ELXSI SYSTEM 6400

SYSTEM ARCHITECTURE

ORDER NO. 9550

ELXSI
2334 Lundy Place, San Jose, California 95131
Telephone: (408) 942-0900 Telex: 172-320

In Asia and Australia:
TATA-ELXSI Pte., Ltd.
37, Joo Koon Circle, Jalan Ahmad Ibrahim, Singapore 2262
Telephone: 261-6727 Telex: RS 34413 Tesing

NOTICE

The information contained in this document is subject to change without notice.

ELXSI makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. ELXSI shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

ELXSI assumes no responsibility for the use or reliability of its software on equipment that is not furnished by ELXSI.

SECOND EDITION

October 1983

The documentation in this manual includes certain features not fully implemented with the current software release. These features are identified on the following page as temporary restrictions. Use only those functions available with the current release. Manual updates will be supplied as new releases implement the features not currently available.

LIST OF EFFECTIVE PAGES

The information on this page identifies the current status of this document with its revisions.

SYSTEM ARCHITECTURE
October 1983

Each new edition represents a total revision of the document. Interim changes are issued as updates to be merged into the manual by the user. All previous changes are incorporated into a new edition. Within the text, changes made since the last edition are marked by revision bars alongside the left margin.

Editions are listed below with corresponding dates:

EDITION	DATE
First Edition	May 1983
Second Edition	October 1983

The following is a summary of revisions effective for this document.

SYSTEM ARCHITECTURE
October 1983

TOPIC	DOCUMENTATION CHANGE	PAGE
Status Codes	Status codes currently implemented	v
Instructions	Opcodes not implemented	vi
Opcode change	Compare inst. opcodes corrected	10-5,6

Features not available with the current release but documented or referred to in this manual are as follows:

- Message system status return codes. The following status codes are a subset of those listed in the manual and are the status codes currently in use.
- The instructions following the list of status codes are currently unimplemented.

MESSAGE SYSTEM STATUS RETURN CODES

```
MSYS funnel Does Not Exist                = 1;
MSYS illegal Channel ID                   = 2;
MSYS funnel Not Disabled                  = 3;
MSYS notification Not Sent                = 5;
MSYS message Too Long                     = 6;
MSYS vector Cannot Be Modified            = 7;
MSYS link Table Is Full                   = 11;
MSYS link Already Created                 = 12;
MSYS no Message In Funnel                 = 14;
MSYS illegal To Disable Funnel ID         = 17;
MSYS specified Funnel Already Dis         = 18;
MSYS specified Funnel Already En          = 21;
MSYS message Contains Link                = 28;
MSYS no Messages On Channels              = 29;
MSYS message Does Not Contain Link        = 30;
MSYS link Does Not Have Hdwr Right        = 31;
MSYS illegal Preferred Link ID            = 32;
MSYS illegal To Move Funnel               = 34;
MSYS link 1 RProcess Dead                 = 36;
MSYS no Msg Buffer Available              = 37;
MSYS too Many Bfrs In Transit             = 38;
MSYS attached Buffer Count Overflow       = 39;
MSYS link Not Def                         = 40;
MSYS unallocated Page                     = 41;
MSYS unallocatable Page                   = 42;
MSYS bad Hi Link Id                       = 43;
MSYS zero Link Id                         = 45;
MSYS neg Data Block Length                = 46;
MSYS funnel Not Enabled                   = 47;
MSYS bad Receive Process Slot No          = 48;
MSYS bad Pointer in Free Link             = 49;
MSYS access Unalloc Page Of Funnel Table  = 50;
MSYS link Cannot Be Sent                  = 51;
MSYS link 1 Target Unit Busy              = 52;
MSYS link 2 Target Unit Busy              = 53;
```

```
MSYS message Not Sent                          = 54;
MSYS Transport Hdw Error                       = 55;
MSYS link Fault                                = 56;
MSYS link 2 Receiving Process Dead             =136;
MSYS link 2 Too Many Attached Buffers          =139;
MSYS link 2 Link not defined                   =140;
MSYS link 2 Unallocated Page                   =141;
MSYS link 2 Unallocatable Page                 =142;
MSYS link 2 Exceeds number of link levels      =143;
MSYS link 2 Zero Link ID                       =145;
MSYS link 2 Funnel Not Enabled                 =147;
MSYS link 2 Target Unit Busy                   =152;
```

UNIMPLEMENTED INSTRUCTIONS

Opcode	Instruction
603	READ.MACH.ID
700	MEMORY.MAN
701	MODIFY.PME
809	EXCH.LINK.FORWARD
80A	FORWARD.MSG
80E	SEND.SMALL.MSG

MANUAL OBJECTIVES

This reference manual provides a description of the instruction set as well as a general background on the ELXSI SYSTEM 6400 Architecture.

INTENDED AUDIENCE

This manual is intended for users who desire to write low level software for the ELXSI. Readers are encouraged to use the higher level ELXSI supplied languages whenever possible for the following reasons:

1. Many of the instructions do not provide data checking for speed and performance advantages. The user may be required to provide specially emitted code for this purpose.

2. The instruction set is designed specifically for compilers rather than for the writer in assembly language.

3. The message system instructions have powerful analogues in the higher level intrinsics that are more convenient.

DOCUMENT STRUCTURE

Chapter 1, "System Overview", provides a general introduction to the ELXSI 6400 features and design.

Chapter 2, "Architecture", characterizes the CPU, memory management, and various internal registers and mechanisms. The chapter provides sufficient context for the user who intends to stay within a process space.

Chapter 3, "Data Representations", describes the ELXSI data types. Included is a brief discussion on the proposed IEEE Floating Point Standard and how it is implemented on the ELXSI.

Chapter 4, "Instruction Set Composition", describes the format and implementation of standard ELXSI primitives.

Chapter 5, "Data Transfer Instructions", describes those instructions that primarily move data. Included are loads and stores, bit field, string, and mutual exclusion.

Chapter 6, "Integer Arithmetic Instructions", describes the integer and arithmetic shift instructions. A general discussion on implementing multiple precision arithmetic with the "carry" mechanism is included.

Chapter 7, "Floating Point Instructions", describes the floating point arithmetic instructions. This chapter assumes in-depth familiarity with chapters 2 and 3.

Chapter 8, "ASCII Arithmetic Instructions", covers the ASCII encoded decimal arithmetic operations.

Chapter 9, "Logical Instructions", describes a wide ranging group of logical instructions.

Chapter 10, "Relational Test Instructions", describes those instructions that perform a specified action based on the outcome of a relational test.

Chapter 11, "Data Conversion Instructions", describes the group of instructions used to convert to different data types.

Chapter 12, "Flow of Control Instructions", includes simple branching, procedure calls, breakpoints and exceptions.

Chapter 13, "Inter-Process Communications", is a general discussion on the message system and process communication structures.

Chapter 14, "Message System Instructions", describes the instructions used for communication between processes.

Chapter 15, "General Instructions", includes instructions for accessing certain internal registers used in the process management, and miscellaneous instructions.

Appendix A, "Alphabetical List of Opcodes", is provided to supplement the index for fast look-ups of instruction descriptions.

Appendix B, "Data Types", illustrates the storage formats for each of the ELXSI data types. This may be considered as a quick reference summary of much of the information in Chapter 3.

Appendix C, "Generalized Addressing Instruction Formats", describes the generalized instruction storage format for each generalized addressing mode.

Appendix D, "Non-generalized Addressing Instruction Formats", describes the storage formats of the non-generalized instructions.

Appendix E, "ELXSI Machine Instruction Descriptors", is a glossary of the symbol notation used in describing how the instructions work.

CONTENTS

CONTENTS

CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

1 SYSTEM OVERVIEW

The ELXSI SYSTEM 6400 is designed for applications that require very high performance, efficient concurrent processing, and system flexibility. The ELXSI solution lies in a highly effective multiprocessor utilization through a process oriented and message based architecture. The message system allows processes to simply "plug in" without concern by the user of machine dependent operations, resulting in a system adaptable to a wide range of applications, capable of evolving in any dimension the user may find necessary, and having an inherently high level of security.

1.1 PROCESS ORIENTED ENVIRONMENT

The ELXSI 6400 features a novel architecture to optimize the concurrent execution of processes. The problem of concurrent execution is related to the management of critical sections and of efficient utilization of system resources. Procedure based systems tend to be limited by the necessity of synchronous job execution, resulting in the wasteful suspension of CPUs in cases where serialized access becomes necessary. This problem is compounded with the addition of more CPUs to the point where the performance rolloff effectively limits the size of the system.

The ELXSI 6400 is a process based system. User tasks and the operating system are hierarchically organized but distributed throughout the machine, executing asynchronously. Rather than using privileged modes as in a procedure based system, resource management and inter-process communication are performed entirely through messages. The program (or user) may be split up among processes that communicate with each other through a message system, each with its own 32-bit address space. As such, a failure in the code of one process cannot destroy other processes, providing firewall protection and information hiding attributes.

1.2 THE OPERATING SYSTEM

The operating system, itself a collection of processes, plugs into the message system and allows the utilization of system resources by the user. Synchronization and allocation of these resources is accomplished through messages, with context associated with a process or a set of processes. This is in contrast with a procedure based system, where synchronization and resource management are performed via shared memory locks handled by privileged procedures. Context in this case is associated with procedures, and a process calls different procedures to accomplish a task.

1.3 INTER-PROCESS COMMUNICATIONS

The communication structure belonging to a process is a defining attribute. This includes the communication paths and the priority of the process relative to other processes. Whether a resource such as the CPU is given over to a process is determined by the message traffic for higher priority processes, if any, relative to the availability of the resources. When a higher priority process has completed execution of all of its messages, the resource is released for the next higher priority process. The effect is better load balancing and system utilization.

1.4 THE VIRTUAL MACHINE

Processes are the smallest executable entities that may be allocated resources. A user may be a process or a collection of processes that communicate with each other through the message system. The ELXSI appears to a process as a virtual machine having 16 64-bit general purpose registers, with other internal registers for control and management of process states. The process may have a total of 4 gigabytes of virtual memory, of which 2 gigabytes are private addressable space for the process. The only communication between processes is through the message system.

Procedures may be invoked within the process space through mechanisms that are both efficient and simple to implement.

1.5 THE MESSAGE SYSTEM

The message system is the means by which processes communicate and is highly amenable to the design of modular programs having processes that easily connect to each other. The communication mechanism is completely transparent to sending processes, allowing process substitution or I/O redirection by the operating system.

Unlike traditional architectures, a message system allows a high degree of security between processes through hardware enforced protection. Specifically, communication between processes may be only by mutual consent, with messages encoded through firmware with the ID of the sending process. Furthermore, since memory is not implicitly shared, a failure of one process, malevolent or otherwise, cannot affect other processes. The high security of this system obviates the need for privileged and user modes with the associated special instructions.

1.6 THE INSTRUCTION SET

The ELXSI instruction set is designed specifically for the emission of very fast code by a compiler. A complete set of primitives, variable length instruction formats, and a wide range of addressing modes optimize flexibility and code compaction. Instructions exist for operations internal to a process and for operations to communicate between processes. The latter, known as message system instructions, have functionally equivalent intrinsics in the high level ELXSI supplied languages.

The instruction set supports several data types including Integer, Numeric string (ASCII), String, Logical, Character, and Floating Point. The Floating Point is based on the proposed IEEE Standard for Binary Floating Point Arithmetic and supports 32-, 64-, and 80-bit floating point operands.

1.7 ELXSI SYSTEM 6400 SPECIFICATIONS

Table 1-1. ELXSI SYSTEM 6400 Specifications

GENERAL	
Multiple CPU Bus Oriented Message Based	1-10 CPUs, 1-4 IOPs
SYSTEM BUS (GIGABUS)	
Cycle Time	25 nanoseconds
Width	64-bit (110 bits total)
Transfer Rate	160-213 Megabytes/second (usable data)
Error Checks	Full internal parity
CENTRAL PROCESSING UNIT	
Type	Microprogrammed general purpose
Logic	LSI/ECL
Word length	64 bits
Cycle time	50 nanoseconds
Registers	16 sets of general purpose & internal registers
Cache	16-Kbyte, 2-way set associative, 32-byte blocks 100 nanosecond access 16 sets of translation look-aside buffers (TLB)
Address space	4 gigabytes
Floating point	IEEE proposed standard, 32-, 64-, 80-bit

MEMORY SYSTEM

Type	64-Kbit MOS with error detection/correction
Minimum size	4 Mbytes
Maximum size	192 Mbytes
Cycle time	400 nanoseconds (128-bit read, 64-bit write)

I/O PROCESSOR

Type	Microprogrammed digital computer
Logic	LSI/ECL
Word Length	64 bits
Cycle Time	50 nanoseconds
I/O Rate	Two 8 Mbyte/second sub-busses

SERVICE PROCESSOR

Type	M68000 based with 256 Kbytes memory

PERIPHERALS

Disk	300 and 474 Mbyte SMD drives
Mag Tape	6250/1600 and 1600/800 bpi at 125 ips
Printers	1200 and 1800 lines per minute

SOFTWARE

Operating System	EMBOS - ELXSI Message Based Operating System
Languages	Pascal, FORTRAN 77, C, COBOL-74, BASIC
DBMS	Relational model

ENVIRONMENTAL SPECIFICATIONS (4-processor configuration)

Power	208V, 3 phase, 47-63 Hz, 60 amps operating (450 amps startup)
Temperature	4-27 degrees Centigrade
Relative Humidity	40-80% non-condensing
Cooling	74,000 BTU/hr.
Dimensions	59"W, 32"D, 70"H
Weight	2500 lbs. (approximate)

2 ARCHITECTURE

This chapter covers those architectural features and mechanisms that are within the user environment of the ELXSI instruction set. Topics in all chapters up to Chapter 13, "Inter-Process Communications", assume a virtual machine as viewed by a single process.

Data is 64 bits in length when in registers, and is of variable length in byte addressable memory. The ELXSI conventions have data fields from left to right, high order to low order, with the most significant bit (or byte) as bit żero (or byte zero). A complete discussion of data representations may be found in Chapter 3, "Data Representations".

2.1 GENERAL PURPOSE REGISTERS

A process has 16 64-bit general purpose registers (GPRs). These registers are used in support of normal ALU operations and for register direct and register indirect addressing. The GPRs in this text are referred to as R0 through R15.

2.1.1 Implicit Register Usage

Certain general purpose registers may be used for parameter passing during a procedure CALL and EXIT, during an interrupt, and when using the IXIT, BXIT and BREAKPOINT instructions. In these cases, R15 is used as a stack pointer or, with the branch through register instructions, as a location for the the return address. Additionally, addressing modes 8 and 9 implicitly use R15 as a base address register.

Operations that place a result into two registers (such as MUL.128 and double extended floating point) must not use R15 as the register containing the low order result, as such operations are illegal. Aside from these restrictions, the reader should use caution so as not to inadvertently destroy the stack pointer.

The implicit usage of registers R0 through R4 is described in Section 2.4, "Interrupts and Breakpoints".

2.2 PROCESS STATUS WORD

The Process Status Word is a 64-bit internal register maintained for each process. Its purpose is to record or modify certain events occurring within a process, such as arithmetic exceptions or the determination of rounding modes. Paired exception bits are provided to allow the user the option of entering special routines on the event of unusual occurrences.

The PSW may be accessed and modified by the instructions READ.STAT and WRITE.STAT. The bits are numbered 0 through 63, with 0 being the high order (leftmost) bit. The table below contains a description of each of these bits.

Bits 1 through 12 simply record or activate some process states. The default values in the table indicate the bit state on process start-up.

Bits 16 through 63 are paired exception bits. An exception is a procedure call through an interrupt, invoked when a system or user defined condition occurs during the execution of an instruction. An access violation is an example of a system defined condition. A relation that is tested by the user through the test and generate exception instructions may be defined such that an exception is generated when the relation is true. This is a user defined condition.

The odd numbered bits in the bit pair are the exception occurred bits. These bits are unconditionally set on the occurrence of the associated exception. For example, an attempted Integer divide by zero will always set bit 19. The exception occurred bits are cleared on process startup; thereafter, the bits are only cleared explicitly by the user through the WRITE.STAT instruction or through the invocation of a system supplied exception handler.

Table 2-1. Process Status Word (Page 1 of 2)

Bit Field	Description
00-01	Floating Point Rounding Mode (RMODE) 00 - Round nearest (RN) - DEFAULT 01 - Round to zero (RZ) 10 - Round to plus infinity (RP) 11 - Round to minus infinity (RM)
02-08	Reserved
09	Integer carry bit 0 - Carry cleared (for ADD) - DEFAULT Carry set (for SUB) 1 - Carry set (for ADD) Carry cleared (for SUB)
10	Decimal carry bit 0 - Carry cleared (for ADD) - DEFAULT Carry set (for SUB) 1 - Carry set (for ADD) Carry cleared (for SUB)
11	Flush Underflows to Zero 0 - Use denormalized numbers - DEFAULT 1 - Flush underflows to zero
12	Single Step Pending 0 - No single step pending - DEFAULT 1 - Single step pending
13-15	Reserved

Table 2-1. Process Status Word (Page 2 of 2)

Bit Pair	Exception Description	Exception Enable Default
16-17	Integer Overflow	1
18-19	Integer Divide by Zero	1
20-21	Floating Point Overflow	0
22-23	Floating Point Underflow	0
24-25	Floating Point Divide by Zero	0
26-27	Floating Point Invalid Operation	0
28-29	Floating Point Inexact Result	0
30-55	Reserved	0
56-57	Software generated exception	1
58-59	Access violation	1
60-61	Illegal or Unimplemented instruction	1
62-63	Reserved	0

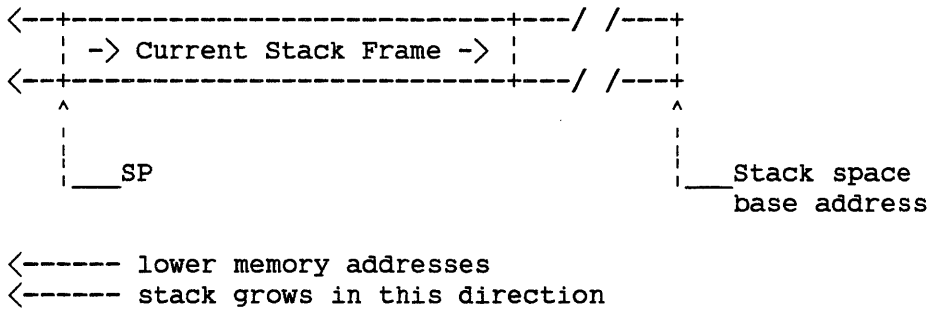
The even numbered bits in the bit pairs are the exception handler enable bits. These bits, when enabled, allow the program to vector to either a system supplied exception handling procedure or to a user supplied procedure on the occurrence of the associated exception. If the exception handler enable bit is disabled, the instruction that generated the exception delivers a default result to the destination operand. The details of this mechanism are discussed in Section 2.5, "Exceptions". The "Exception Enable Default" column in the table specifies the state of the exception handler enable bits on process startup.

2.3 PROCEDURE CALLS

ELXSI procedure calls reference and deallocate stack frames in a manner that is simpler, faster, and with less overhead than traditional approaches. Basically, a procedure explicitly allocates its own stack space before making any procedure calls, and places, rather than pushes, any data items it needs to save onto the stack. When the procedure calls a nested procedure, the CALL instruction places the return PC into the first 32-bit word of the allocated stack space and branches to the nested procedure. The nested procedure likewise allocates its own stack space. On exiting, the nested procedure deallocates its local stack frame (with the EXIT instruction) and the return address in the caller's stack frame is automatically placed into the PC. Control then resumes with the calling procedure. Alternatively, the "branch through register" instructions may be used, but stack frame allocation and deallocation support must be provided by the user.

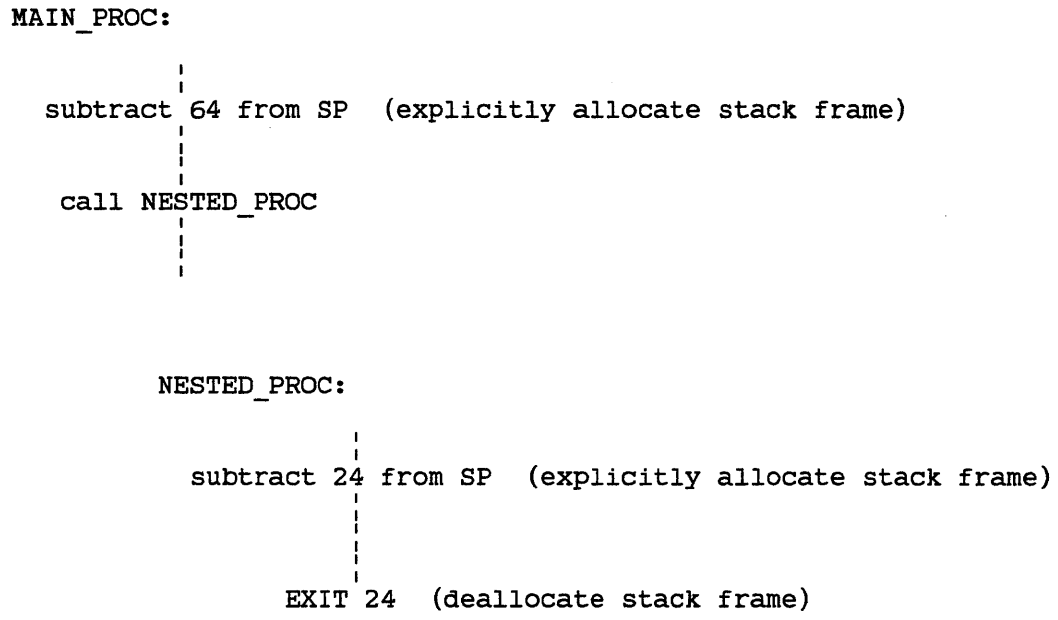
This section discusses the general case of procedure calls and exits. Chapter 12, "Flow of Control Instructions", describes the instructions in detail. The special cases of interrupts and breakpoints may be found in Section 2.4, "Interrupts and Breakpoints".

On process creation, a space in virtual memory is allocated as the stack area that starts at a base address and grows towards low memory. This is illustrated in the below diagram. The Stack Pointer (SP), register 15, points to both the top of the stack and the stack frame base address. When a procedure is entered, it explicitly allocates a stack frame by subtracting from the SP the required stack frame space. The new low address of the stack frame is the stack frame base address, the top of stack, and the contents of SP. (The stack pointer is initialized to the stack space base address on process startup.)

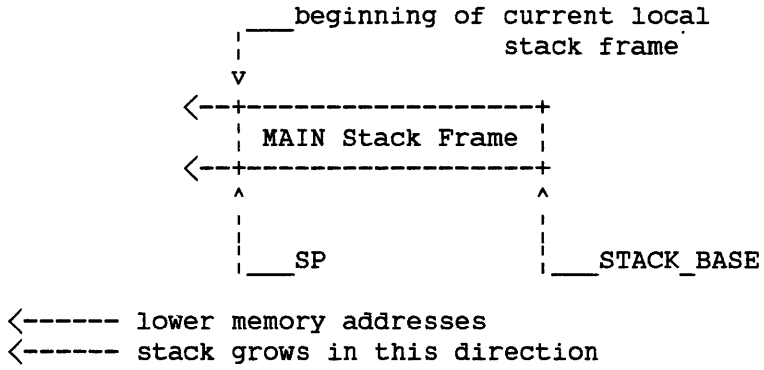


An example of the call/exit protocol is shown below. It has a main procedure with 64 bytes of local storage, calling a nested procedure with 24 bytes of local storage. Upon entry to the main procedure, it is assumed that the stack is empty so that SP points to the stack base.

The outline for the code is



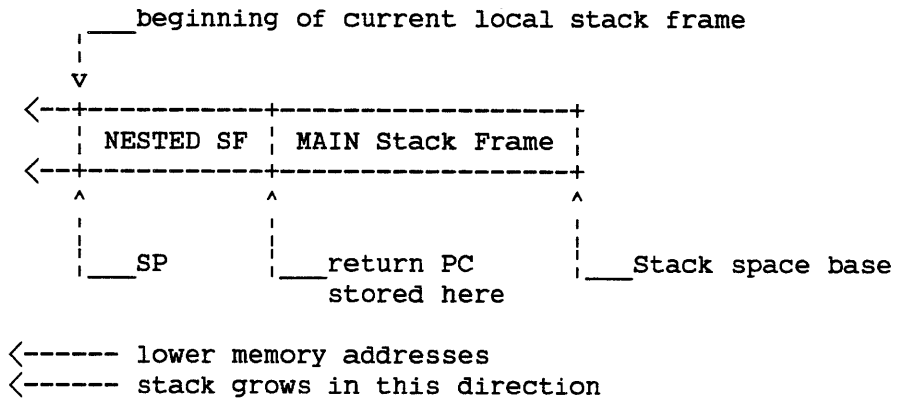
First, the MAIN_PROC code allocates its local stack frame. Since the stack grows towards low memory, this is accomplished by subtracting 64 from SP. SP now points to the beginning of the current local stack frame and to the top of stack, which is now STACK_BASE-64. The stack now looks like:



The first 32 bits of the allocated local stack frame hold the space for the return Program Counter (PC) value. As this location is the current top of stack, it is pointed to by SP. When MAIN_PROC does call NESTED_PROC, the CALL instruction places the return PC in this location and replaces the contents of the Program Counter with the address of the first instruction in NESTED_PROC. The "return PC" is the address

of the instruction following the CALL instruction, that is, the location for resuming execution after the return from the called procedure.

NESTED_PROC allocates its stack frame in a similar fashion. By subtracting 24 from SP, the SP now equals STACK_BASE - (64+24), and the stack looks like:



The last instruction in NESTED_PROC is an EXIT 24 instruction. This instruction first adds 24 to SP, thus deallocating its local stack frame. The SP now points to the top of stack for MAIN_PROC. The 32-bit return PC pointed to by SP is then placed into the program counter, and control is returned to MAIN_PROC.

Note that since the SP always points to the first data item in the current stack frame, other data items may be simply addressed by the base register + displacement addressing modes, which include addressing modes 8 and 9 that are designed for this purpose.

It can be seen that the ELXSI architecture varies from the traditional approach in two respects. First, the local stack frame is assumed to run from low towards high memory addresses, thus allowing SP to serve as both top of stack pointer and current stack frame pointer. The second difference is when and where the local stack frame size is stored.

In summary, data items are addressed by the lowest byte address they occupy in virtual memory, the stack grows toward low memory, and the beginning of a local stack frame is the address of the lowest byte in the frame. The calling procedure must explicitly allocate stack space for itself, otherwise the return address will be destroyed by any nested procedures that are called.

It is good practice to never store data in a space where stack growth may occur, as interrupts use this area to store part of the process state. The data and stack spaces are described in more detail under Memory Organization.

2.4 INTERRUPTS AND BREAKPOINTS

This section presents an overview of the environment in which an interrupt occurs. A more complete description may be found in the discussion on the message system in Chapter 13, "Inter-Process Communications".

Information from sources external to the process may only enter the process through the message system. A process will trap to an interrupt handler if the process is in a state that allows interrupts to occur and if the message is received on a designated interrupt channel. On an interrupt, the execution of the BREAKPOINT instruction, or after the execution of an instruction following the BXIT instruction with single stepping enabled, these events will:

1. Allocate a stack frame to save the process data. For BREAKPOINT/SINGLE STEP, subtract 136 from the SP. For interrupts, subtract 144 from the SP.

2. Place the PC of the next instruction onto the stack. The PC resides in the lower 32 bits of the 64-bit word placed onto the stack.

3. For interrupts only, place the current local priority onto the stack (64 bits - the priority occupies the low order 4 bits).

4. Place Registers RF down to R0 on the stack.

5. Set R0 to the memory address of the saved PC in the stack.

6. Set R1 to the memory address of the saved R0 in the stack.

7. For all interrupts, branch to execute the interrupt handler (and ignore the remaining steps).

If a BREAKPOINT or the result of BXIT with single stepping enabled (Debugger entry) then set R2 to the following entry reason code

Code	Reason
0	BREAKPOINT
1	BXIT with single stepping enabled
2	Execute address break
3	Read data break
4	Write data break

8. Set R3 to the memory address of the instruction just executed (old PC) or to zero if single step.

9. Set R4 to the memory address of the data causing the data break. Set to zero if not a data break.

10. Branch to enter the Debugger.

After the breakpoint/single step save is complete, the stack appears as follows:

Stack after BREAKPOINT/SINGLE STEP save of process data

Definitions:

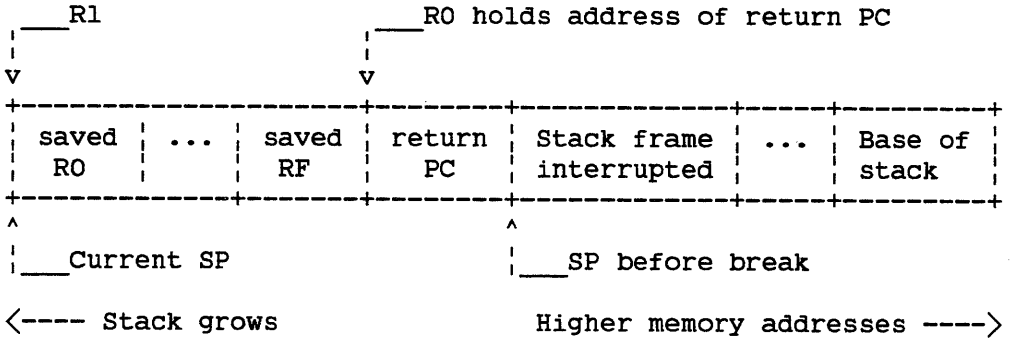
SPb = stack pointer before break occurred
PCb = program counter before break occurred

Register contents:

RF = SPb - 128 - 8 (or current SP)
R0 = SPb - 8 (or current SP+128)
R1 = SPb - 128 - 8 (or current SP)
R2 = reason code
R3 = address of instruction just executed
(zero for single step)
R4 = memory address of data causing break

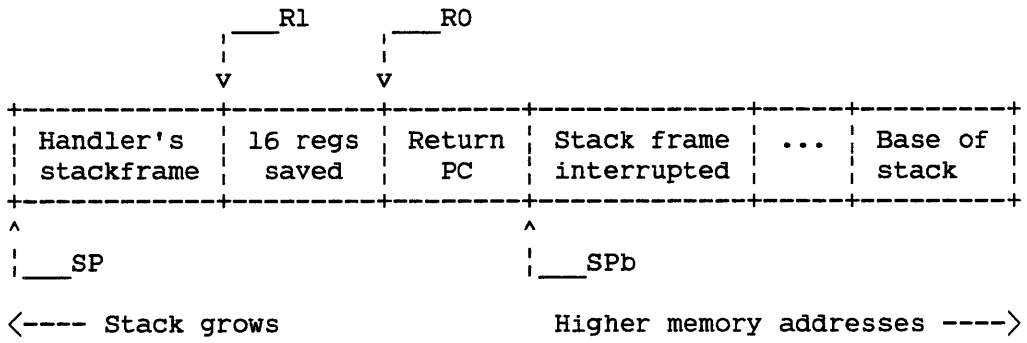
Stack contents:

SP + 128 : saved PC
SP + 120 : saved RF (SP)
etc.
SP + 8 : saved R1
SP + 0 : saved R0



The invoked Debugger procedure can receive the saved PC and GPRs as parameters and may modify the saved state before exiting. After the Debugger has built its stack frame, the stack looks like:

Stack with DEBUGGER Stack Frame added



On exiting from the handler, the BXIT instruction deallocates the handler's stack frame, restores and deallocates the 16 registers and the PC, and returns to the interrupted process.

Following are the stack and registers after the interrupt save is complete:

Stack after INTERRUPT save

Definitions:

SPb = stack pointer before interrupt occurred
PCb = program counter before interrupt occurred
OLP = old local priority

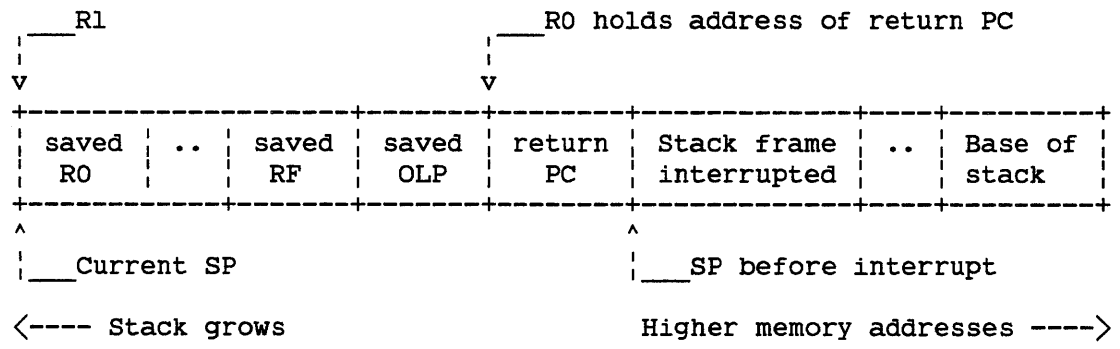
Register contents:

RF = SPb - 128 - 16 (or current SP)
R0 = SPb - 8 (or current SP+128+8)
R1 = SPb - 128 - 16 (or current SP)
R2 = receiving funnel ID (on an interrupting channel)

Stack contents:

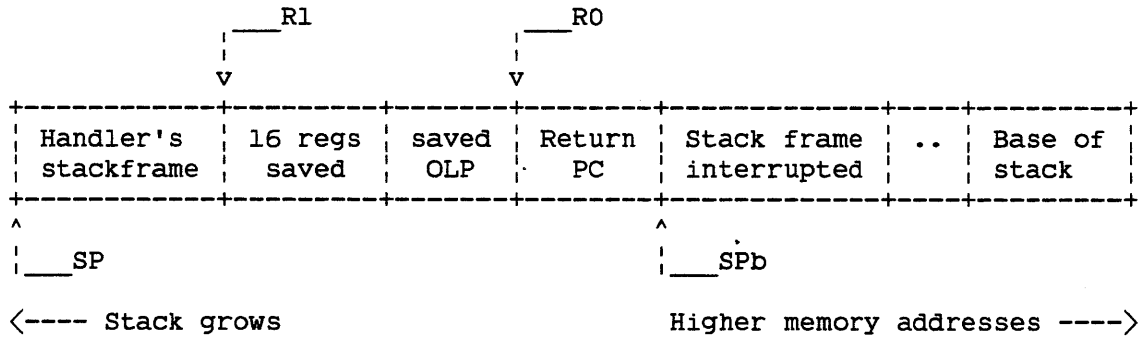
SP + 136 : saved PC
SP + 128 : saved old local priority
SP + 120 : saved RF (SP)
etc.
SP + 8 : saved R1
SP + 0 : saved R0

The stack after the interrupt save is



The invoked interrupt procedure can receive the saved PC, old local priority, and GPRs as parameters, but it is not recommended that interrupt handlers modify the saved state before returning. After the handler has built its stack frame, the stack looks like:

Stack with Handler Stack Frame added



On exiting from the interrupt routine, the IXIT instruction deallocates the handler's stack frame, restores and deallocates the 16 registers, old local priority, and the PC, performs the activities that are required for exiting from an interrupt (such as executing the SET.LOCAL.PRIORITY function), and returns to the interrupted process.

This is note for those readers who may write code which modifies data structures that are shared with an interrupt handler. Compilers written for the ELXSI will in general perform register tracking. This may result in references to a datum shared with an interrupt handler not actually loading that datum from memory, because it already resides in a register. If an interrupt handler is invoked between two such references and modifies the datum, the value in the register may not be current. This is, of course,

Refer to the appropriate ELXSI Language User's Guide for information on how to maintain the integrity of such shared data items.

2.5 EXCEPTIONS

Exceptions provide a convenient mechanism to call a special procedure in the event of some defined condition or relation. These exceptions fall into two general classes: user-interceptable exceptions and non-interceptable exceptions (faults). Non-interceptable exceptions are used by the system for such conditions as page faults. Typically these exceptions are corrected by the system and are transparent to the faulting process.

This section covers the class of user interceptable exceptions. The Program Status Word always records the occurrence of such exceptions. The user may establish several courses of action that include the following:

1. Default. The system will generate a fabricated result for the instruction that generated the exception if the user has disabled the exception handler enable bit associated with the exception.

2. Call the system exception handler. If the user has enabled the exception handler enable bit, a system supplied exception handler is invoked which will enter the Debugger if selected or cause the process to terminate.

3. Call a user supplied exception handler. The user may write an exception handler to replace the system exception handler. Intrinsics are provided for this purpose.

Unless otherwise specified, program control resumes at the instruction following the instruction that generated the exception.

2.5.1 User Interceptable Exceptions

User interceptable exceptions may be classified as follows:

1. Architectural Exceptions. Exceptions of this type are detected by the system hardware and invoke the system exception handler as well as, optionally, a user supplied exception handler. These include all bit pair entries in the PSW listed in Table 2-1.

2. Software Exceptions. Software exceptions are generated by the "generate exception" instructions described in Chapter 10, "Relational Test Instructions", and by the EXCEPTION instruction. The generate exception instructions test a user defined relation and cause an exception if the relation is true.

Additionally, the user may specify exception subclasses within these instructions that will interrupt to user supplied subclass exception handlers.

The difference between these and the architectural exceptions lies in the user requirement to supply the exception handler and to keep track of unique exception types. The general status of this type of exception is maintained in the PSW as bit pair 56-57.

2.5.2 Exception Mechanism

An instruction that causes an exception with the associated exception handler enabled will cause the system to generate a message. This message, directed into a standard funnel of the process, contains the instruction that generated the exception and other information some of which may be defined by the user. Arrival of the message on the funnel causes an interrupt that invokes the system exception dispatcher. The dispatcher may then pass the message, after some modification, to the correct system exception handler or to a user supplied exception handler, which resolves the problem and returns control to the interrupted procedure. The exception software resides in the code space of the process.

The standard exception funnel is essentially an input port that is attached to channel 1 of the same process that generated the exception. Its purpose is to receive incoming exception messages generated by the system. The channel to which the funnel is attached is enabled for interrupts, such that the arrival of a message will interrupt the faulting process and invoke the system exception dispatcher specified by the funnel's interrupt vector.

The system exception dispatcher is the procedure pointed to by the interrupt vector associated with the standard exception funnel. This procedure brings the exception message into the process space, determines the type of exception generated, and either calls the appropriate system supplied exception handler or a user supplied exception handler, if provided. The exception message also contains information on other exceptions that may have occurred for the instruction, and calls these additional procedures based on the priorities of the incoming exceptions.

The called procedure handling the exception may manipulate the data that has been placed on the stack by the exception dispatcher. When the procedure exits, eventually to the interrupted code, the stack containing the corrected information will be deallocated and restored to the appropriate registers. (Target operands in memory are corrected in the exception handling procedures.)

2.5.3 Exception Message Description

The message data block has the following format upon arrival into the standard exception funnel:

Table 2-2. Exception Message Format

	0	1	2	3	4	5	6	7
0	Instruction (72)						-->	
1	-->	unused (48)					res (8)	exc (8)
2	Program Counter (32)				unused (32)			
3	unused (64)							
4	unused (64)							
5	Value of source operand 1 (low) (64)							
6	unused (64)							
7	Value of source operand 2 (low) (64)							
8	Actual result (high) (64)							
9	Actual result (low) (64)							
10	unused (64)							
11	unused (64)							
12	Variant Part (64)							

The fields in the message area have the following meanings:

Instruction: This field contains the entire instruction that caused the exception. The instruction is left justified.

Exception Code: This field has a value from 0 to 255 and uniquely identifies the type of exception to be processed. Refer to Table 2-3 for the exception codes.

Table 2-3. Exception Message Codes

Code	Interceptable Exceptions
0	Integer overflow
1	Integer Divide by Zero
2	Floating Point Overflow
3	Floating Point Underflow
4	Floating Point Divide by Zero
5	Floating Point Invalid Operation
6	Floating Point Inexact Result
7-19	Reserved
20	Software generated exception
21	Access violation
22	Illegal or Unimplemented instruction

The system exception dispatcher extracts this field and adds 256 to the code to pass to the exception handlers. This allows the lower 256 values to be used for subclass exception codes specified in the appendages of the generate software exception instructions.

Program Counter: The Program Counter is the 32-bit virtual address of the instruction that raised the exception.

Variant Part: Several of the exceptions, including the software generated exceptions, provide specific values relevant to the instruction and the exception type.

The parts of the exception message which are valid for each exception are defined below.

INTEGER OVERFLOW

Message Field	Contents of Message Field
exception code	0
instruction	instruction causing exception
program counter	instruction causing exception
source operand 1	none
source operand 2	none
actual result - high	none mul may provide high word of result
actual result - low	mod 2**64 of result
variant part	none

INTEGER DIVIDE BY ZERO

Message Field	Contents of Message Field
exception code	1
instruction	instruction causing exception
program counter	instruction causing exception
source operand 1	none
source operand 2	none
actual result - high	none
actual result - low	none
variant part	none

FLOATING POINT OVERFLOW

Message Field	Contents of Message Field
exception code	2
instruction	instruction causing exception
program counter	instruction causing exception
source operand 1	none
source operand 2	none
actual result - high	valid if extended precision
actual result - low	scaled result
variant part	none
comments	results depend on precision

FLOATING POINT UNDERFLOW

Message Field	Contents of Message Field
exception code	3
instruction	instruction causing exception
program counter	instruction causing exception
source operand 1	none
source operand 2	none
actual result - high	valid if extended precision
actual result - low	scaled result
variant part	none
comments	results depend on precision

FLOATING POINT DIVIDE BY ZERO

Message Field	Contents of Message Field
exception code	4
instruction	instruction causing exception
program counter	instruction causing exception
source operand 1	none
source operand 2	none
actual result - high	none
actual result - low	none
variant part	none

FLOATING POINT INVALID OPERATION

Message Field	Contents of Message Field
exception code	5
instruction	instruction causing exception
program counter	instruction causing exception
source operand 1	system handler may provide
source operand 2	system handler may provide
actual result - high	none
actual result - low	none
variant part	none

FLOATING POINT INEXACT RESULT

Message Field	Contents of Message Field
exception code	6
instruction	instruction causing exception
program counter	instruction causing exception
source operand 1	none
source operand 2	none
actual result - high	valid if extended precision
actual result - low	scaled result
variant part	none
comments	results depend on precision

SOFTWARE EXCEPTION (includes compare and generate exception)

Message Field	Contents of Message Field
exception code	20
instruction	instruction causing exception
program counter	instruction causing exception
source operand 1	sub-code rx for EXCEPTION instr sub-code from appendage for CMP instr
source operand 2	exception data rz for EXCEPTION instr 0 for CMP instr
actual result - high	none
actual result - low	none
variant part	none

ACCESS VIOLATION - data, instruction

Message Field	Contents of Message Field
exception code	21
instruction	0
program counter	data - instruction causing exception instr - last instruction to execute
source operand 1	none
source operand 2	none
actual result - high	none
actual result - low	none
variant part	none
comments	cannot supply instruction because of execute only access rights.

ILLEGAL/UNIMPLEMENTED INSTRUCTION

Message Field	Contents of Message Field
exception code	22
instruction	instruction causing exception
program counter	instruction causing exception
source operand 1	none
source operand 2	none
actual result - high	none
actual result - low	none
variant part	none

2.6 MEMORY ORGANIZATION

This section describes the virtual addressing space available to a process and its allocation.

2.6.1 Virtual Memory Space

A general virtual address is 32 bits in size. The addressing space (4 gigabytes) is considered by the memory manager to have four subspaces of one gigabyte each. These subspaces are Private (2), Public, and Reserved. The two most significant bits of the address are used to identify the address space:

Table 2-4. Virtual Memory Allocation

MSB	Virtual Memory allocation
00	Private Space
01	
10	Public Space
11	Reserved

Private space, of up to two gigabytes for each process, is managed as two separate and independent subspaces but appears to a process as a single, contiguous space. The first subspace, known as P0 (MSB 00), grows towards higher memory, while the second subspace, known as P1 (MSB 01), grows towards lower memory. The memory mapping scheme splits up a process between these two subspaces to allow the code/stack area and the data area to grow towards each other.

The actual use of this private space includes the following types of addressable objects:

- Code. This includes all the private instructions which are to be executed by the process. Private code usually consists of a program together with any unshared libraries. Code is normally marked with execute-only access and, therefore, may not be modified. This provides the advantages of re-entrancy and maintainability.

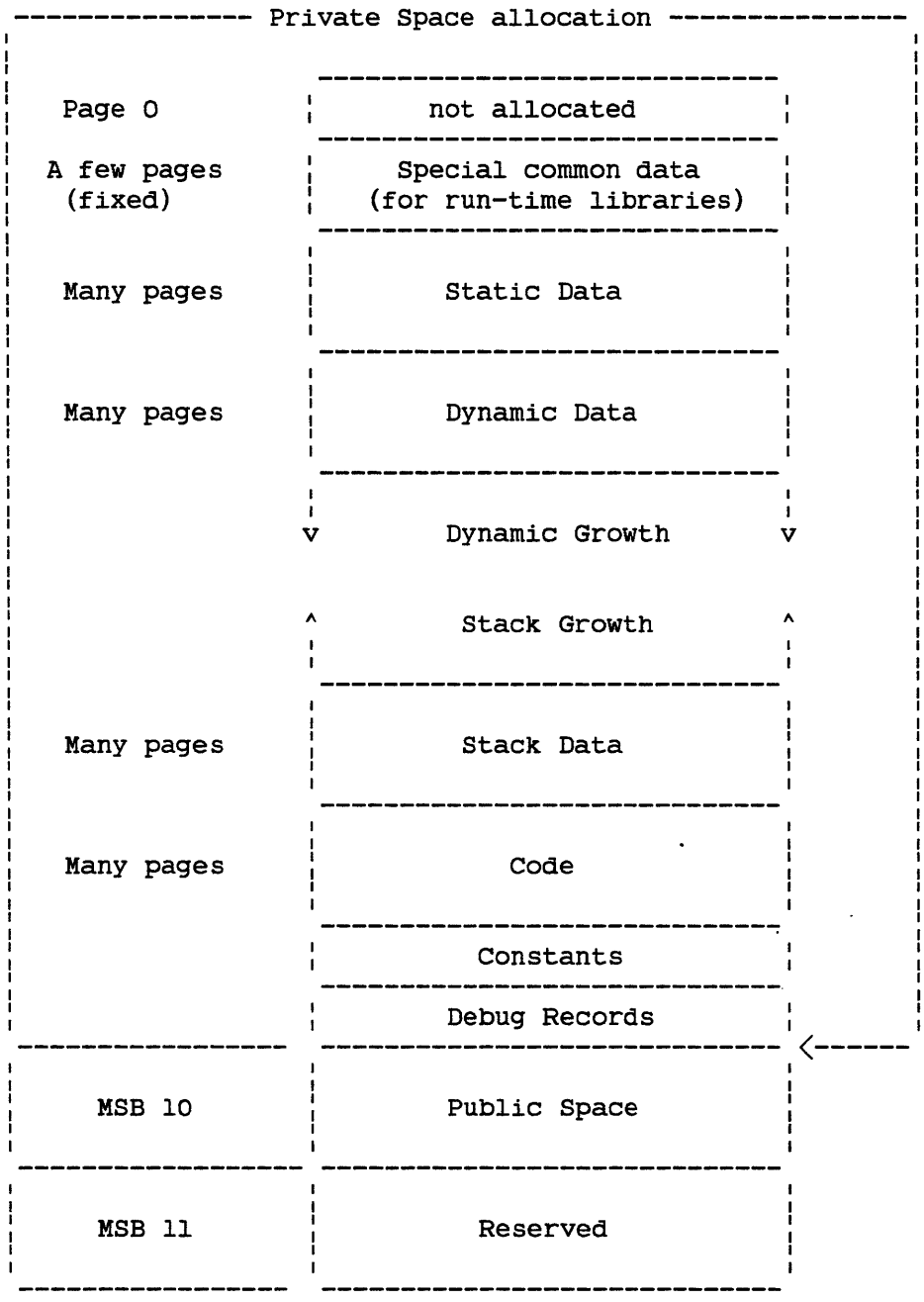
- Static data. Static data has a lifetime equal to that of the program and does not change in size during execution. This corresponds to the global data declarations of most languages. The actual size of static data will vary depending on the program being executed.

- Dynamic Data. Dynamic data changes in size and application during the life of the program. The major uses of dynamic data are for the Pascal heap area, control blocks for the file system, and other run-time library procedures. Management of dynamic data space is the responsibility of the software. For example, the Pascal compiler is responsible for any garbage collection that may be necessary in its heap area.

- Stack Data. Stack data has a lifetime equal to the life of the procedure which defines it. This data includes all variables that are declared locally within a procedure or subroutine. Stack data is added to the stack as a procedure is called, and is deleted from the stack when the procedure terminates.

Stack data and Dynamic data both change in size during the execution of a program. The ultimate size of each of these classes of data is not known before the program is actually executed. For this reason, separate these two classes of data as far as possible and let them grow towards each other; this leaves the largest possible open growth space between them. The standard organization of private space is illustrated below.

Table 2-5. Private Space Allocation



Page zero cannot be allocated. This is to catch common programming bugs that reference address zero.

The special common data area contains pointers for the run time libraries. These pointers are used to locate the so-called "own" variables that actually reside in the dynamic data area. An example of these own variables is the file control blocks used by the in-process part of the file system.

The public subspace, called PUB, (addresses with MSB = 10) is shared with all processes and is used primarily for system library code. The public subspace grows towards higher memory. The last subspace is reserved for future expansion.

2.6.2 Page Maps

For each subspace (such as those described above) there is a map which is used by the hardware to translate a virtual address to a physical address. The Page Map contains Page Map Entries (PMEs) which describe the attributes of a page.

Virtual space is logically broken up into pages of 2 Kbytes (2048 bytes) each. For every virtual page there may be a physical page allocated on disk by the memory manager that holds the data. A page "in memory" is a copy of this data in physical memory, a necessary condition for utilization by the instruction set. A page that is referenced but not "in memory" will generate a page fault, causing the memory manager to bring the page in off the disk so that the CPU can continue executing for the faulting process.

The page map will be large if the space it describes is large. Therefore, it is necessary to break the page map itself into pages to allow effective management of main memory. This is achieved by a hierarchy of page maps. A single level page map is capable of handling small subspaces up to 512 KBytes. When space requirements increase, a second level is created with its first descriptor pointing to the first level page. The other PMEs on this second level define other first level pages as the need for their existence becomes apparent, to a maximum of 128 Mbytes. The third level is created in a similar fashion for subspaces larger than 128 Mbytes. The conceptual similarity of the various sizes of subspace makes it easy to expand a subspace even as it crosses one of the boundaries separating the three levels. For example, if a process needs to get another page added to a 512 Kbyte subspace, it will first be checked to see that no administrative size limits have been exceeded and then the memory manager will create a second-level page map.

2.6.3 Page Map Entry (PME)

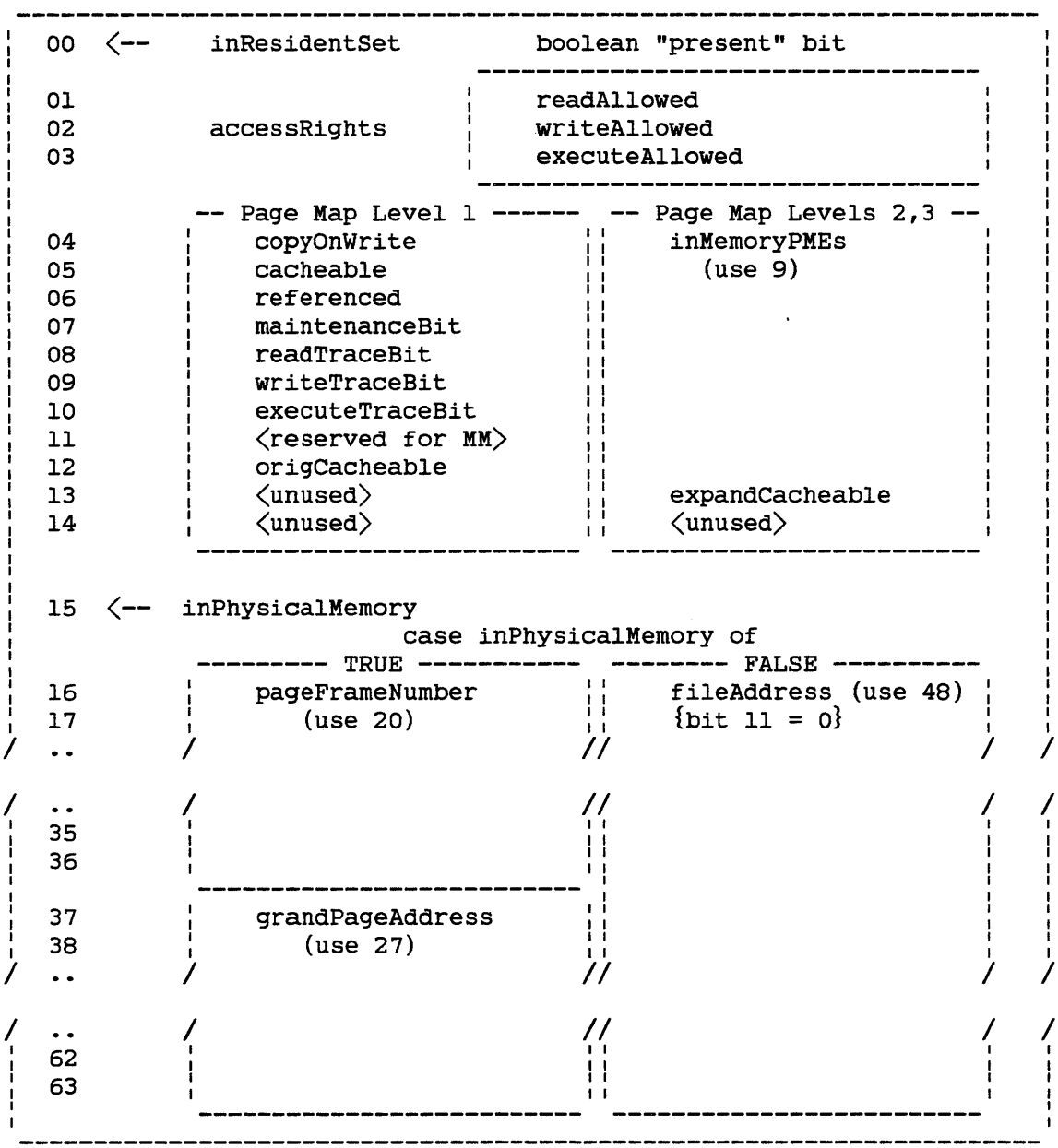
The page map entry is a generalized descriptor of a page in its current state. Refer to Table 2-4 on the following page.

There are essentially two forms of the PME depending on whether the page is in memory or not. This is specified by the "inPhysicalMemory" bit. If the page is in memory, the PME specifies the physical address of the page image. If the page is absent from memory, then the memory manager checks bit 11. If bit 11 = 0, then the PME specifies the disk address of the page. A disk address consists of a disk identifier and a sector address.

The PME may be read with the READ.PME instruction. The following bits may be modified with the MODIFY.PME instruction:

maintenanceBit
readTraceBit
writeTraceBit
executeTraceBit

Table 2-6. Page Map Entry



3 DATA REPRESENTATIONS

The ELXSI architecture supports six classes of data representations, as listed below:

- o INTEGER
- o LOGICAL
- o FLOATING POINT
- o CHARACTER
- o STRING
- o NUMERIC STRING

For the illustrations in this chapter, the most significant bit is bit 0, shown as the leftmost bit. Bytes are represented in a similar fashion in both registers and memory. Data resides in memory from low address to high address, with the low address as byte zero. Register data is always a 64-bit quantity, which means that memory operands are sign or zero extended and right justified when loaded into a register. Addressing formats are discussed in Chapter 4, "Instruction Set Composition".

3.1 FLOATING POINT

ELXSI Floating Point representations conform to the proposed IEEE Standard for Binary Floating-Point Arithmetic. Three floating point number representations are provided: single (32 bits), double (64 bits) and double extended (80 bits). All three forms may exist in either a register or in memory.

A floating point operand is composed of a sign bit, a significand and an exponent. The significand is an unsigned number with a value greater than or equal to zero, and less than two.

The integer bit of the significand is physically present in the extended double precision storage formats, but the bit is not present in the single and double precision forms. The implied state and location of the integer bit in these cases is referred to as the "hidden bit".

The exponent range for single and double precision starts at 1 to enable the encoding of the hidden bit. If the exponent is zero, the hidden bit is inferred to be zero, and the number is denormalized (or zero if the number is equal to zero). The range for Double Extended starts at zero.

3.1.1 Terminology

The following are terms used in this section:

- Biased exponent: The sum of an exponent and a constant (bias) chosen to make the biased exponent's range non-negative.

- Denormalized number: A non-zero floating point number whose exponent is the minimum value for the format, and whose leading significand bit is zero.

- Exponent: The component of a binary floating point number that normally signifies the integer power to which two is raised in determining the value of the number. It is represented by an "e".

- Fraction: The field of the significand that lies to the right of its implied binary point. It is represented by an "f".

- Infinity: The ELXSI uses the affine closure form of infinity. It may be illustrated by a straight number line with a positive infinity at one end and a negative infinity at the other end.

- NaN: Not a number. There are two types of NaN's, signaling and quiet. These symbolic entities are used to check for invalid data or certain arithmetic enhancements.

- Quiet NaN: Quiet NaNs are propagated through all arithmetic operations to allow the retrospective diagnosis of invalid results. These are represented with the first bit to the right of the binary point equal to zero.

- Signaling NaN: Signaling NaNs signal the INVALID OPERATION exception whenever they appear as operands. These allow values for uninitialized variables or arithmetic enhancements, and are represented with the first bit to the right of the binary point equal to one.

- Significand: The component of a binary floating-point number that consists of an explicit or implied leading integer bit to the left of the implied binary point, and a fraction field to the right.

- Sign bit: The most significant bit of an integer or floating point representation that determines the sign of the operand. Otherwise, this bit is the most significant bit of any datum, and is represented by "s".

3.1.2 Data Types

The three floating point types, Single, Double, and Double Extended, are illustrated below in both register and memory form. Single or Double precision operands require one register. Double Extended operands require two registers, R and R+1, with bit 0 of a memory operand aligning with bit 0 of R and bit 127 aligning with bit 63 of R+1.

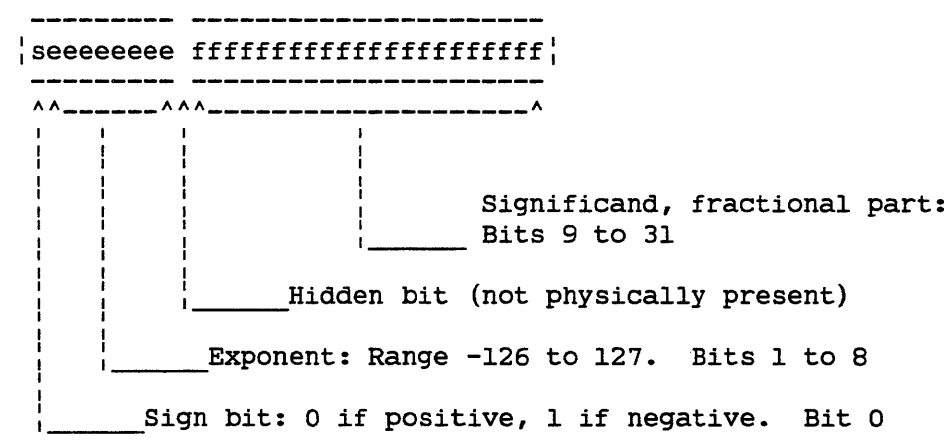
3.1.2.1 Single Precision

The Single Precision floating point number is represented in a 32-bit field with 1 sign bit, 8 exponent bits, 23 significand bits and 1 hidden bit.

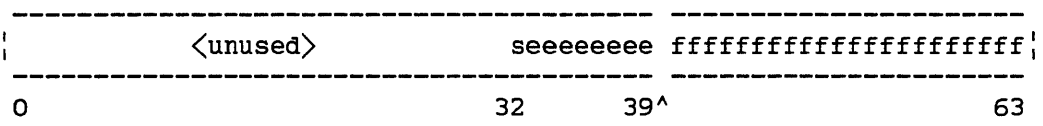
The exponent may range from -126 to 127. The actual exponent is biased by 127 so that it is always positive. As such, the smallest exponent value (-126) is represented by an actual exponent of 1 (or zero if denormalized). An actual exponent of 255 is used for representing infinities and NaN's.

The hidden bit is assumed to be immediately to the left of bit 9. The binary point lies between the hidden bit and bit 9.

Single Precision in Memory



Single Precision in Register



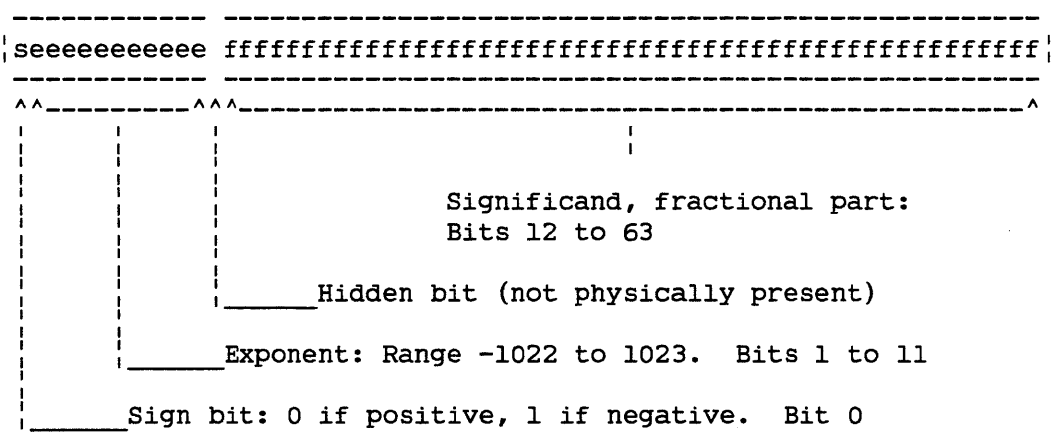
3.1.2.2 Double Precision

The double precision floating point number is represented in a 64-bit field with 1 sign bit, 11 exponent bits, 52 significand bits and 1 hidden bit.

The exponent may range from -1022 to 1023. The actual exponent is biased by 1023 so that it is always positive. As such, the smallest exponent value (-1022) is represented by an actual exponent of 1 (or zero if denormalized). An actual exponent of 2047 is used to represent infinities and NaN's.

The hidden bit is assumed to precede bit 12. The binary point lies between the hidden bit and bit 12.

Double Precision in Memory and Register



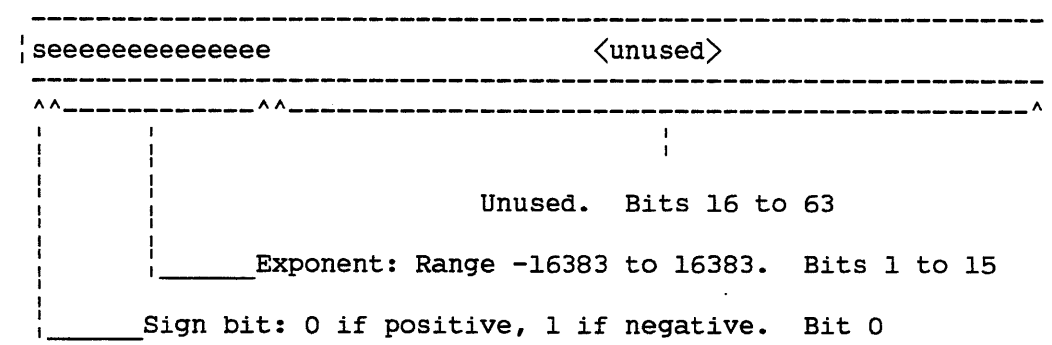
3.1.2.3 Extended Double

The extended double floating point number (80-bit) is represented in a 128-bit field with 1 sign bit, 15 exponent bits, 64 significand bits and 48 unused bits. There is no hidden bit.

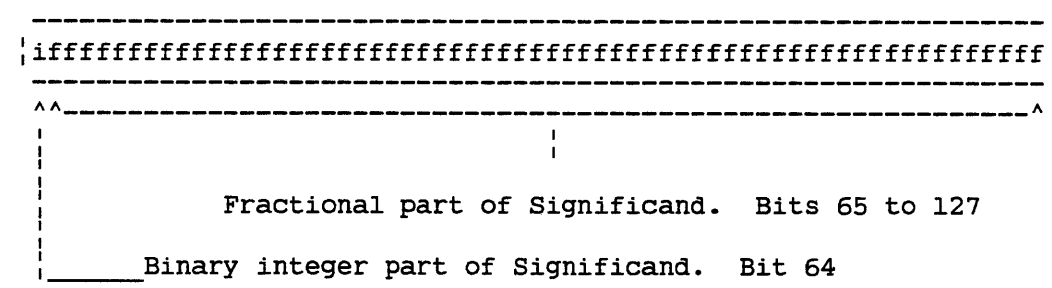
The exponent may range from -16383 to 16383. The actual exponent is biased by 16383 so that it is always positive. As such, the smallest exponent (-16383) is represented by an actual exponent of zero. An actual exponent of 32767 is used to represent infinity and NaN's.

There is no hidden bit. The binary point lies between bit 64 and 65. If bit 64 = 1, the the number is Normal. If the exponent = 0 and bit 64 = 0, then the number is Denormalized. If the exponent is not equal to 0 or 32767, and bit 64 = 0, then the number and all operations on it are undefined.

Double Extended in Memory and Register, High Order Word.



Double Extended in Register and Memory, Low Order Word.



3.1.3 Operand Classes

The five operand classes characterize an operand through values in the exponent and significand fields. This information is summarized in the following table. The "i" and "f" table headers in the Significand field specify the integer and fractional parts respectively. Note that normalized numbers use a biased exponent such that the exponent will always be positive.

Table 3-1. Operand Classes

Operand Class	Data Type	Exponent	Significand i	Significand f	Significand Value
zero	Single Precision	0	x		zero
	Double Precision	0	x		zero
	Extended Double	0	0		zero
denormalized number	Single Precision	0	x		nonzero
	Double Precision	0	x		nonzero
	Extended Double	0	0		nonzero
normal(ized) nonzero number	Single Precision	1 .. 254	x		any
	Double Precision	1 .. 2046	x		any
	Extended Double	0 .. 32766	1		any
infinity	Single Precision	255	x		zero
	Double Precision	2047	x		zero
	Extended Double	32767	0		zero
quiet not-a-number (NaN) symbol	Single Precision	255	x	0	nonzero
	Double Precision	2047	x	0	nonzero
	Extended Double	32767	0	0	nonzero
	Extended Double	32767	1	0	any
signaling not-a-number (NaN) symbol	Single Precision	255	x	1	any
	Double Precision	2047	x	1	any
	Extended Double	32767	0	1	any
	Extended Double	32767	1	1	any

3.1.4 Rounding Modes

There are four rounding modes that may be selected. The rounding modes are determined by the rounding mode bits set in the Process Status Word.

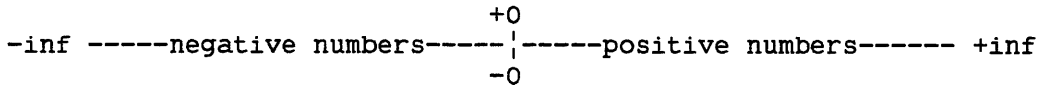
1. Round to nearest even. In the halfway case, round so that the low order bit of the result is zero. Over a large number of roundings the direction will be unbiased. This is the default rounding mode.

2. Round to zero. Truncate magnitude.

3. Round toward positive infinity.

4. Round toward minus infinity.

The two infinities (positive and negative) are considered to be at the opposite ends of a number line. (Positive and negative zero have the same value).



3.1.5 Predicates and Relations

It is possible to compare all floating point numbers in all formats. Comparisons are exact and never overflow or underflow. The four mutually exclusive relations are:

UNORDERED	U	?
LESS THAN	L	<
EQUAL	E	=
GREATER THAN	G	>

An unordered relation occurs when one or more of the operands is a NaN. (A NaN is unordered to everything, including itself.) The relational test instructions use a 4-bit compare condition field (ccf) with the values indicated in the table to test for the corresponding relation.

Table 3-2. Predicates and Relations

PREDICATES			RELATIONS			
ad hoc	symbol	ccf	U	L	E	G
>	G	1	F	F	F	T
=	E	2	F	F	T	F
>=	EG	3	F	F	T	T
<	L	4	F	T	F	F
<>	LG	5	F	T	F	T
<=	LE	6	F	T	T	F
<=>	LEG	7	F	T	T	T
?	U	8	T	F	F	F
?>	UG	9	T	F	F	T
?=	UE	10	T	F	T	F
?>=	UEG	11	T	F	T	T
?<	UL	12	T	T	F	F
?<>	ULG	13	T	T	F	T
?<=	ULE	14	T	T	T	F
?<=>	ULEG	15	T	T	T	T

Refer to Chapter 10, "Relational Test Instructions", for a discussion on the Relational Test instructions.

3.1.6 Exceptions

This section describes the conditions under which the user interceptable architectural exceptions are generated. Also provided are the default results if the exception handler is not enabled, and the results supplied by the system exception handlers.

The following exceptions are mutually exclusive:

- Floating Point Invalid Operation
- Floating Point Divide by Zero
- Floating Point Underflow
- Floating Point Overflow

The Floating Point Inexact Result exception may coincide with either Underflow or Overflow. In this case, the Underflow or Overflow has precedence. If both exceptions occur and only one exception handler is enabled, then the message appropriate to the enabled exception is sent to the exception handler along with the information that both exceptions occurred. If both or neither of the exception handlers are enabled, the action appropriate to the exception with priority is carried out.

3.1.6.1 Floating Point Overflow

If the exception handler is disabled, the result depends on the rounding mode as shown in the following table:

Table 3-3. Floating Point Overflow Result Matrix with Disabled Exception Handler

Rounding Mode	Sign of Result	Returned Value
Round to Nearest Even	x	infinity with sign
Round to Zero	x	largest normalized with sign
Round to Plus Infinity	1	largest normalized with sign
Round to Minus Infinity	0	largest normalized with sign
Round to Plus Infinity	0	infinity with sign
Round to Minus Infinity	1	infinity with sign

If the system exception handler is enabled, the system will calculate the infinitely precise result and round it to the precision of the destination. This result is then scaled and is passed to the exception handler, as follows:

Single Precision	Actual result of exponent minus 192
Double Precision	Actual result of exponent minus 1536
Extended Precision	Actual result of exponent minus 24576

3.1.6.2 Floating Point Underflow

Exponent underflow is detected before rounding. If the exception handler is enabled, then the system will calculate the infinitely precise result rounded to the precision of the destination. The result is then augmented and passed to the exception handler as follows:

Single Precision	Actual result of exponent plus 192
Double Precision	Actual result of exponent plus 1536
Extended Precision	Actual result of exponent plus 24576

If the exception handler is disabled in Flush to Zero mode, then the system will set the Underflow exception and deliver zero as the result. If not in Flush to Zero mode, then the system delivers a denormalized number as the default result. In this case, the Underflow exception is set only if the result is not exact (which will also set the Inexact Result exception).

3.1.6.3 Floating Point Divide by Zero

If the exception handler is disabled, then the result is infinity with appropriate sign (XOR the signs of the operands). The system exception handler will terminate the process or enter the Debugger if enabled.

3.1.6.4 Floating Point Invalid Operation

An Invalid Operation exception occurs if an operation is attempted on an operand for which the operation is not defined, such as a floating point multiply with an unordered operand. If the exception handler is disabled, the default result for an Invalid operation is a Quiet NaN (Not a Number) whose significand is not necessarily related to the significand of either of the operands, even if one of them is a Signaling NaN.

The canonical Quiet NaN is as follows:

Single Precision	7F80 0001
Double Precision	7FF0 0000 2000 0000
Extended Double	7FFF 0000 0000 0000, 8000 0100 0000 0000

The system exception handler will terminate the process or enter the Debugger if enabled.

3.1.6.5 Floating Point Inexact Result

An inexact exception occurs if information is lost by rounding, or if infinity is delivered as the result of operations on finite operands. If the exception handler is disabled, the computed result is delivered. The system exception handler will terminate the process or enter the Debugger if enabled.

3.2 INTEGER

The ELXSI architecture provides for 8-, 16-, 32-, and 64-bit integers. Integers may be either signed or unsigned. If signed, the integer is represented in two's complement form, and the sign bit is bit 0. Integers are low-order adjusted when in registers. Memory operands loaded into registers are sign or zero extended to 64-bits as specified by the operation.

Not all types of integer operations are supported for all types of integers. Please see Section 6.1, "Multiple Precision Integer Arithmetic", for specific usage.

3.2.1 Exceptions

This section describes the conditions under which the user interceptable hardware exceptions are generated. Also provided are the default results if the exception handler is not enabled, and the results supplied by the system exception executioners.

3.2.1.1 Integer Overflow

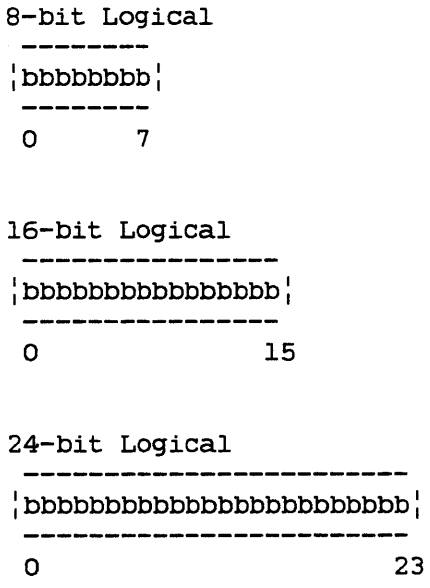
Integer overflow occurs with signed integer operations that generate results exceeding 64 bits, and with certain operations that attempt a store with overflow check. Operations generating an integer overflow will place the result into the target operand before checking the exception handler enable bit in the PSW. High order bits of the result that do not fit into the target operand are lost, as the result exceeds the modulus of the target. The system exception handler will terminate the process or enter the Debugger if enabled.

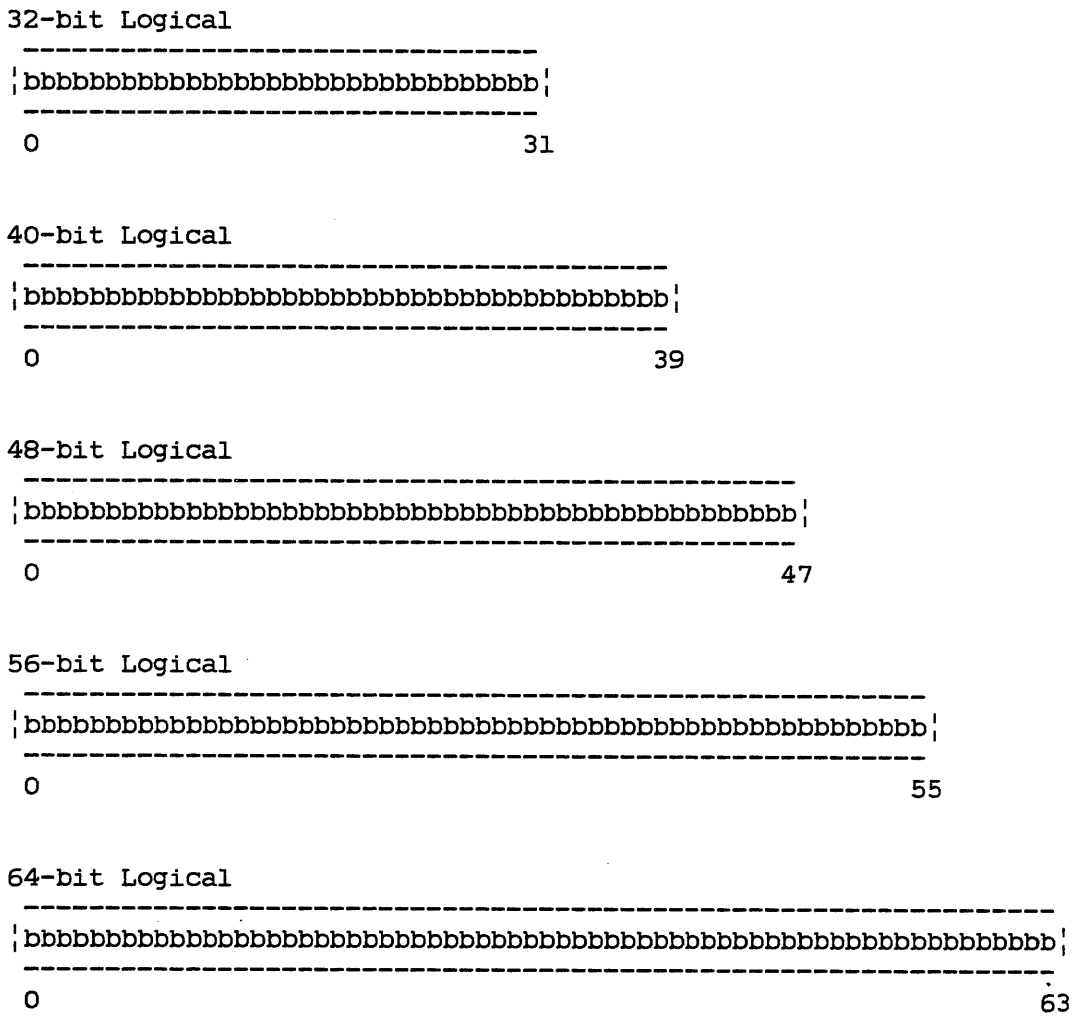
3.2.1.2 Integer Divide by Zero

This exception occurs with an attempt to divide by zero. If the exception handler is disabled, then the target operand remains unchanged without overflow. The system exception handler will terminate the process or enter the Debugger if enabled.

3.3 LOGICAL

A logical datum is a bit-string (array of bits) with length 8-, 16-, 32-, or 64-bits. Bit-strings loaded from memory are low-order justified in the register and are unsigned. Immediate operands are sign extended to 64 bits when in a register.





Chapter 9, "Logical Instructions", provides additional information on utilization by the instruction set.

3.4 CHARACTER

Characters on the ELXSI are ASCII encoded bytes. Bits in a character are numbered from left to right, high order to low order, 0 to 7. For the ASCII representations, bit 0 of the byte is zero. A '1' in bit 0 is presently undefined for an alternate character set.

The ASCII equivalence table may be seen below:

Table 3-4. ASCII Equivalence

Low Nibble		High Nibble							
hex		00	10	20	30	40	50	60	70
	dec	0	16	32	48	64	80	96	112
00	0	NUL	DLE	SP	0	@	P	`	p
01	1	SOH	DC1	!	1	A	Q	a	q
02	2	STX	DC2	"	2	B	R	b	r
03	3	ETX	DC3	#	3	C	S	c	s
04	4	EOT	DC4	$	4	D	T	d	t
05	5	ENQ	NAK	%	5	E	U	e	u
06	6	ACK	SYN	&	6	F	V	f	v
07	7	BEL	ETB	'	7	G	W	g	w
08	8	BS	CAN	(	8	H	X	h	x
09	9	HT	EM	)	9	I	Y	i	y
0A	10	LF	SUB	*	:	J	Z	j	z
0B	11	VT	ESC	+	;	K	[	k	{
0C	12	FF	FS	,	<	L	\	l	|
0D	13	CR	GS	-	=	M	]	m	}
0E	14	SO	RS	.	>	N	^	n	~
0F	15	SI	US	/	?	O	_	o	DEL

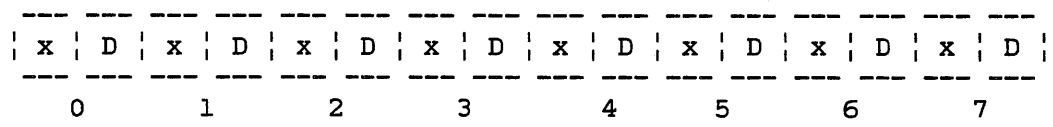
3.5 STRING

Type String is an array of ASCII characters. Strings may start on any byte boundary and have a maximum length of (2**31)-1 characters. Bytes in a character string are numbered from left to right, low memory address to high memory address, starting from 0. Refer to the Character data type for a description of the ASCII character.

3.6 NUMERIC STRING

A numeric string represents an unsigned decimal integer value through a character string. Each byte in the string must contain the ASCII representation of a decimal digit (Hex 30 through 39). When in a register, leading and trailing decimal zeros may be represented by hex 00 or hex 30. However, all ASCII instructions (CVT.IA, ASCII.ADD, ASCII.ADDC, ASCII.SUB, and ASCII.SUBC) return all digits in ASCII display format, that is, all byte values within the range hex 30 to hex 39.

Numeric String in Register



where
x = hex3 or [= 0 if all bits to left or right = 0]
D = hex0 through hex9

4 INSTRUCTION SET COMPOSITION

Instructions on the ELXSI machine operate on a large variety of data types using standard primitives and microcode optimizations. This chapter addresses those instructions that execute within a process. The message system instructions may be found in Chapter 14.

Most instructions within a process are register directed with the design of minimizing accesses to memory. A fast addressing and execution schema results from a variable length instruction format and efficient operand addressing. Basically, the information within the first 3 nibbles specifies the instruction format, the addressing form, and the operation. The first pass at the instruction defines all operands, fetches the memory operand (if any), and hands the operation over to the appropriate hardware to execute. There are no memory indirect operations, and in general, no more than one memory reference, so only one memory cycle is used.

4.1 INSTRUCTION TYPES

There are two overall types of instructions on the ELXSI, generalized and non-generalized. Instructions that typically manipulate register data and require addressing flexibility fall into the generalized class. These include load, store, arithmetic, logical, and data conversion instructions. Instructions that do not need addressing flexibility or are highly specialized fall into the non-generalized class and include flow of control instructions, microcode optimized instructions, and instructions for inter-process communication. Non-generalized instructions minimize the overall instruction set complexity by eliminating unneeded addressing forms.

4.2 ADDRESSING MODES

Addressing modes allow the the user (compiler or human) to optimally address operands for the instruction. Non-memory operands include registers and immediate data. Memory operands may be addressed through register indirects (base and index) and immediate data, or various combinations thereof.

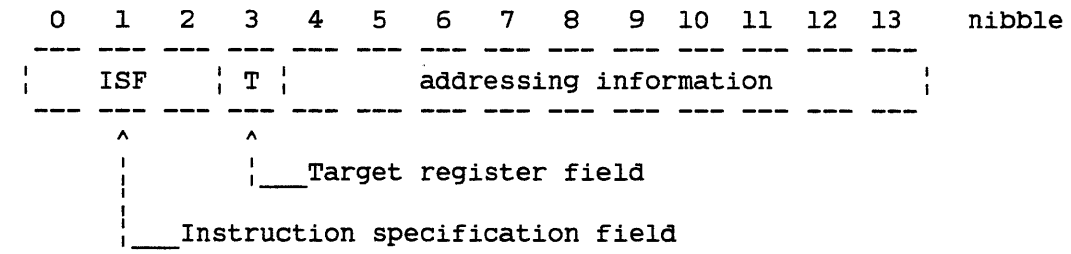
Generalized instructions have 15 addressing modes; 5 of these belong to a subclass known as short-op addressing. The other ten, referred to as generalized addressing modes, enable a wide selection of addressing forms and are characterized by 12-bit opcode fields. A property of the generalized class of instructions is the singular correspondence between an addressing mode and the format and length of the instruction; both are determined from the first byte.

Short-ops have single-byte opcodes and exist as optimizations of frequently used long-op (generalized) addressing forms. There are 5 addressing modes, and there is usually a corresponding long-op instruction for each short-op form.

Non-generalized instructions have addressing forms that are unique to each instruction, and they have no fixed format.

The diagram below shows a typical form of a generalized class instruction with generalized addressing. The Instruction Specification Field (ISF) holds the opcode and the addressing mode. The target register field specifies the register to receive the results of a computation; this register may concurrently be a source. The addressing information area is of variable length and may contain register specifier fields for source operands, an address, or immediate encoded data.

Typical Form of Instruction using Generalized Addressing



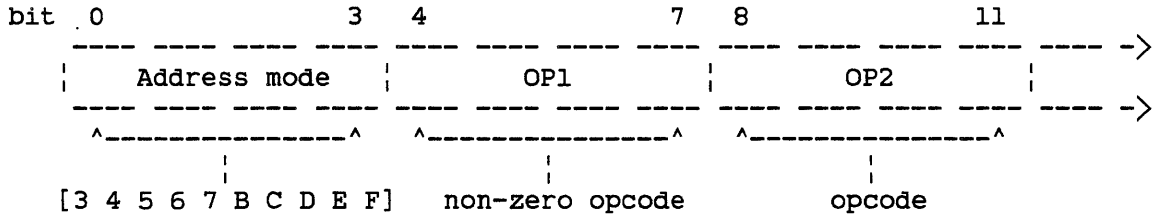
The first 2 nibbles of the opcode are sufficient to establish the addressing class and the addressing mode, if any. This relationship may be seen below:

Addressing class		Nibble 0 is in set	AND	Nibble 1 is
Generalized	=	[3 4 5 6 7 B C D E F]	AND	[non-zero]
Short-op	=	[0 1 8 9 A]	AND	[non-zero]
Non-generalized	=	[opcode]	AND	[zero]

Generalized instructions are always characterized with a non-zero value in nibble 1. When this is true, nibble 0 establishes the addressing mode and, with the exception of appendages, the format and the length of the instruction.

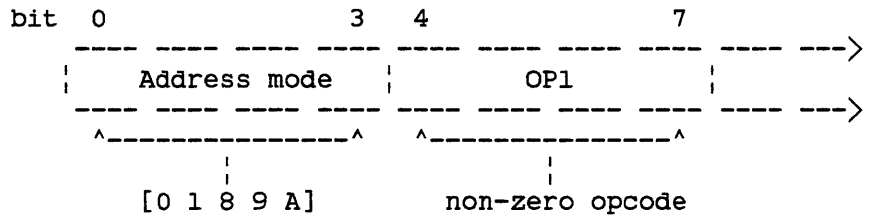
The generalized addressing (long-opcode) form is shown below:

ISF for Generalized Addressing Forms



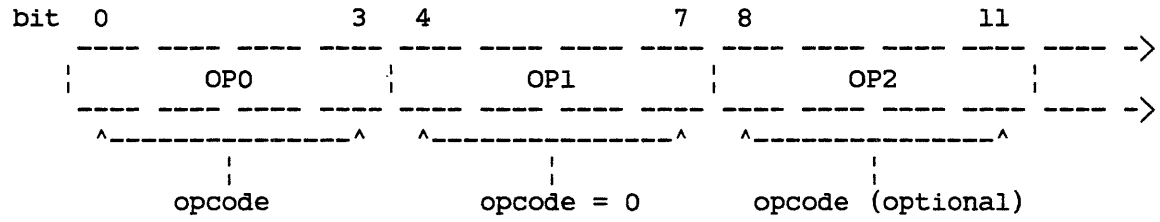
The short-opcode addressing forms, belonging to the class of generalized instructions, use nibble 1 for the opcode and nibble 0 for the addressing mode, as seen below:

ISF for Short Opcode Addressing Forms



The non-generalized class of instructions are characterized by having a zero value in nibble 1. They do not have addressing modes, rather the addressing schema is specific to each instruction. Nibble zero is now part of the opcode. Its usual form may be seen below:

ISF for Non-Generalized Instructions



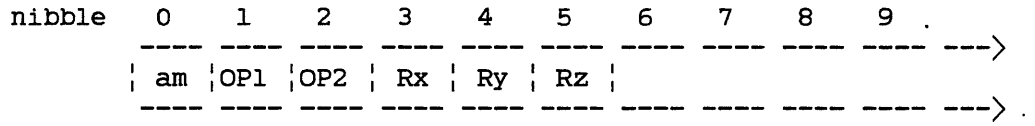
The remainder of these instructions hold register specifier fields, addressing information, immediate encoded values, and appendages as appropriate.

4.3 REGISTER SPECIFIERS

Register specifiers either point to register operands or hold immediate values as an operand. A register operand may be a register whose contents are to be used in, say, an arithmetic operation, or whose contents may point to a memory address. Register specifiers reside in one or more 4-bit fields of an instruction and have a numeric range of ('0' to 'F') hex, representing the general purpose registers R0 to R15.

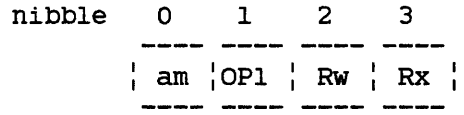
For the generalized instructions, the inclusion of the 4-bit register specifier fields is specific to each addressing mode rather than to the instruction. Regardless of whether the register is operative in a given instruction, the field must be included if it is specified in that addressing mode. This is because instructions have a fixed length characteristic of each addressing mode. For example, instructions that have monadic operations, such as logical NOT, may use only 2 operands, the target and source operand. Operand Ry, usually another source operand, is ignored, although its register specifier field is encoded in the instruction as a requisite of the addressing mode.

Register Specifiers for Generalized Addressing Modes



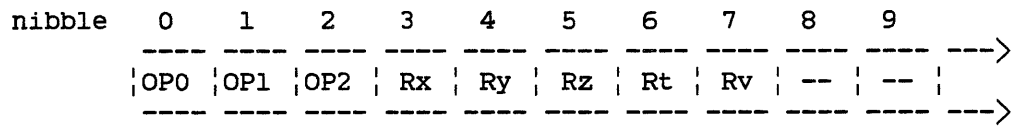
In Generalized Addressing modes, the nibbles labeled Rx, Ry and Rz may be used for register specifiers. Rx is usually the target operand, and in some cases may be both the source and target.

Register Specifiers for Short-Op Addressing Modes



In Short-Op addressing modes, the nibbles labeled Rw and Rx may be used for register specifiers. Rx is usually the target operand, and in some cases may be both source and target.

Register Specifiers for Non-Generalized Instructions



For non-generalized instructions, no strict format is followed, and all labeled nibbles (Rx through Rv) may be used for register fields.

4.4 DESCRIPTION AND USE OF ADDRESSING MODES

This section describes each addressing mode with the memory layout and the assembler usage. The diagrams show memory layouts of the 15 generalized and short opcode addressing modes. The memory layouts of the non-generalized instructions are instruction-specific and are described with the instructions. Use of the given assembler syntax will force the specified addressing mode, rather than allowing the compiler to make the selection.

The symbols have the following meanings:

- A register specifier surrounded by brackets "[]" indicates that the contents of the register point to a memory location. If these symbols are adjacent, then sum the register quantities to find an effective address.

- A symbol having the form "e{}" indicates a displacement added to find an effective address. The number within the brackets indicates the bit length of the displacement quantity encoded in the instruction.

- A symbol having the form "=e{}" indicates that the value encoded in the instruction is to be used as an immediate operand rather than for an address.

Further descriptions may be found in Appendix E.

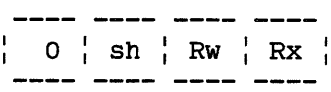
The address space for a process in the ELXSI machine is 32 bits. All effective addresses are 32-bit signed integers, and all address calculations use signed arithmetic. If an operand or an address computation exceeds 32 bits in length (such as a base register), the lower 32 bits are extracted without regard for the value of the unused high order bits. Overflow is ignored (and will not be detected).

There are no restrictions on register usage for establishing an effective address. Any register may be used for the base, index or displacement as the user may find appropriate. To find an effective address, simply add together the contents of the registers specified for the address computation along with any immediate values encoded in the instruction, as applicable to the addressing mode.

Instructions start on any byte boundary and may cross physical memory boundaries.

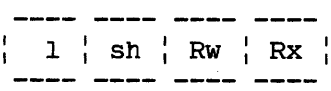
4.4.1 Mode 0: 1 Register, Register (Short)

Assembler usage: op Rx,Rw



4.4.2 Mode 1: 1 Register, Register (Short)

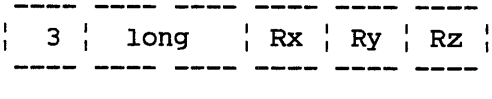
Assembler usage: op Rx,Rw



Modes 0 and 1 are used when both operands reside in registers and the result may replace one of the operands. In general, Rx is both a source and a target, and Rw is a source. These are short opcode versions of mode 3.

4.4.3 Mode 3: 2 Register, Register

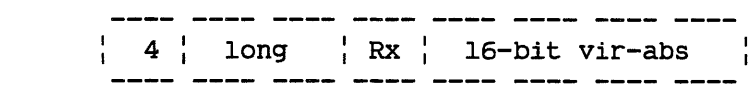
Assembler usage: op Rx,Ry,Rz



Mode 3 is used when the source operands reside in registers and the target is a register. In general, Rx is the target and Ry and Rz are the sources. For monadic operations Ry is usually ignored.

4.4.4 Mode 4: 1 Register, Absolute Memory Address

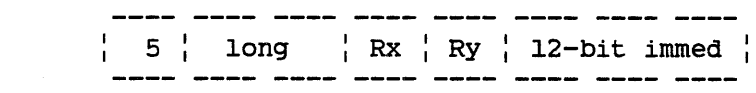
Assembler usage: op Rx,e{16}



Mode 4 uses an unsigned 16-bit integer as the virtual memory address of the data item, and is intended to provide access to global scalars residing in the first 64 Kbytes of the virtual memory space. This mode is an optimization of mode 6.

4.4.5 Mode 5: 2 Register, Immediate

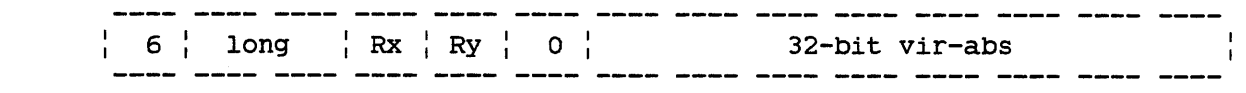
Assembler usage: op Rx,Ry,=e{12}



Mode 5 is used when one of the source operands is immediate. The immediate operand is a 12-bit signed integer. This is an optimization of mode 7. Since most immediate values are small, this instruction will suffice in a majority of the cases.

4.4.6 Mode 6: 2 Register, Long Absolute Memory Address

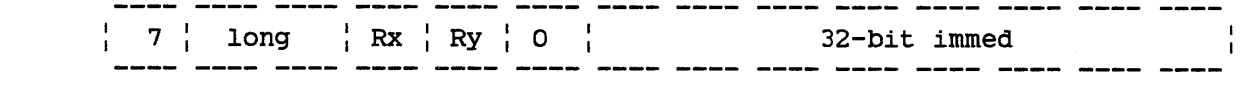
Assembler usage: op Rx,Ry,e{32}



Mode 6 uses a signed 32-bit integer as the virtual memory address of the data item, to provide access to global scalars residing anywhere in the virtual memory space.

4.4.7 Mode 7: 2 Register, Long Immediate

Assembler usage: op Rx,Ry,=e{32}

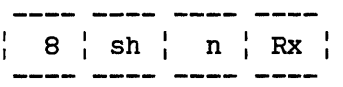


Mode 7 is used in place of the 2 Register, Immediate mode when the immediate operand cannot be held in a 12-bit signed integer.

4.4.8 Mode 8: Stack Pointer Relative (Short)

Assembler usage: op Rx,[SP]e6

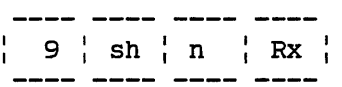
where e = 4n, 0<=n<=15



4.4.9 Mode 9: Stack Pointer Relative (Short)

Assembler usage: op Rx,[SP]e6

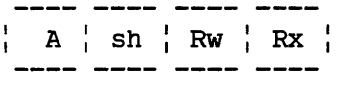
where e = 4n, 0<=n<=15



Modes 8 and 9 provide access to local variables. The effective address ([sp] + 4*n) jumps in multiples of four bytes relative to the stack pointer. These modes are an optimization of modes D and F, and one of them must be used if either the data item does not start on a multiple of 4 bytes from the stack pointer, or the data item is out of range of these modes.

4.4.10 Mode A: 1 Register, Base + Zero Displacement (Short)

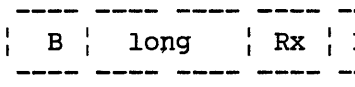
Assembler usage: op Rx,[Rw]



Mode A is used to access based scalars (such as in the Pascal heap), and to reference parameters. This mode is a short opcode optimization of mode B, and has only two operands. The effective address = [Rw].

4.4.11 Mode B: 2 Register, Base + Zero Displacement

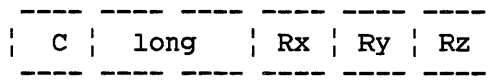
Assembler usage: op Rx,Ry,[Rz]



Mode B is the generalized form of mode A, and is used in a similar fashion. This mode (and mode A) are sometimes referred to as "register indirect". The effective address = [Rz].

4.4.12 Mode C: 1 Register, Base + Index + Zero Displacement

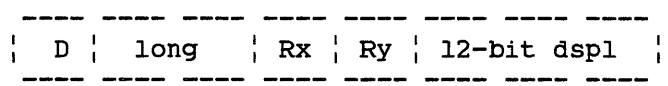
Assembler usage: op Rx,[Ry][Rz]



Mode C is useful for access to elements of arrays passed as parameters to a subroutine. The effective address of the data item = [Ry][Rz], or the sum of the contents of Ry and Rz.

4.4.13 Mode D: 1 Register, Base + 12-bit Displacement

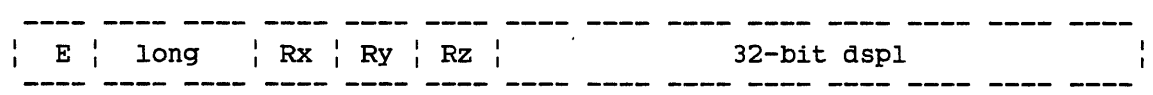
Assembler usage: op Rx,[Ry]e{12}



Mode D is a generalization of modes 8 and 9, and is used to access local scalars and records, or records in the heap. This mode is an optimization of mode F. The effective address of the data item = [Ry] + the 12-bit signed displacement.

4.4.14 Mode E: 1 Register, Base + Index + 32-bit Displacement

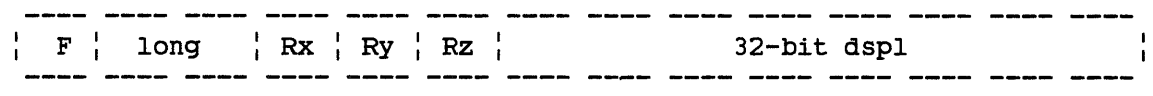
Assembler usage: op Rx,[Ry][Rz]e{32}



Mode E is used for accessing arrays of records passed as parameters. The displacement is used to give access to the elements of the record. The effective address = [Ry][Rz] + 32-bit signed displacement.

4.4.15 Mode F: 2 Register, Base + 32-bit Displacement

Assembler usage: op Rx,Ry,[Rz]e{32}



Mode F provides access to the entire virtual memory, relative to a base register, and is a generalization of modes 8, 9 and D. The 32-bit displacement is signed. It is used to access global arrays, and may be useful for access to mapped files. The effective address of the data item = [Rz] + 32-bit signed displacement.

4.5 IMPLEMENTATION EXAMPLES

The Integer Multiply instructions are used to illustrate some addressing forms, register usage, and instruction encoding at the machine level. Detailed descriptions of each of the addressing modes are in the previous section. Descriptions for the symbols may be found in Appendix F.

Opcode			Implementation
MUL.16	MUL.32	MUL.64	
		0Bx	Rx <- Rx * Rw (short opcode)
		3B8	Rx <- Ry * Rz
498	4A8	4B8	Rx <- Rx * e{16}
		5B8	Rx <- Ry * =e{12}
698	6A8	6B8	Rx <- Ry * e{32}
		7B8	Rx <- Ry * =e{32}
B98	BA8	BB8	Rx <- Ry * [Rz]
C98	CA8	CB8	Rx <- Rx * [Ry][Rz]
D98	DA8	DB8	Rx <- Rx * [Ry]e{12}
E98	EA8	EB8	Rx <- Rx * [Ry][Rz]e{32}
F98	FA8	FB8	Rx <- Ry * [Rz]e{32}

Addressing mode

Source Operands

Target Operand

The MUL instruction finds the product of the source operands and places the result into the register specified by Rx. The instruction may specify two operands or three operands. In 2 operand forms, found in addressing modes [0 4 C D E], Rx is both the target and a source operand, and 'last operand' is the other source operand. In 3 operand forms, found in addressing modes [3 5 6 7 B F], Ry and 'last operand' are the source operands, and Rx is the target operand.

Find the code EB8. Observe that this is a MUL.64 instruction of the generalized addressing type with addressing mode 'E'. The action, from the 'Implementation' column to the right, reads,

> "Multiply the contents of the register specified by Rx with the contents of the effective address, and place the result into the register specified by Rx".

The effective address is the sum of registers Ry + Rz + the 32-bit signed integer displacement encoded in the instruction. The contents of the effective address is a 64-bit Integer (MUL.64).

Find the code EA8. Note that the addressing mode is the same. However, the contents of the effective address is now a 32-bit word (MUL.32). For this instruction, the 32-bit word is sign extended to 64 bits prior to the multiply.

Find the code OBx. This is a MUL.64 instruction with a short opcode addressing format. '0' specifies the addressing mode, and 'x' indicates that the nibble belongs to the register specifier Rw. The action is

> "Multiply the contents of registers Rx and Rw, and place the result into register Rx".

Assume that we have a 16-bit local scalar that we wish to multiply against the contents of register R11, with the result placed in R11. We find that the MUL.16 instruction using addressing mode 'D', is appropriate. The opcode becomes 'D98'. Our base register, [Ry], is register R9. Assuming some quantities:

```
base register  =          0000000000100020
16-bit scalar  =                      8000
12-bit displacement  =                 032
source/target R11    =   2000000000000000
```

The effective address of the scalar is '100052' hex, or the sum of the base register and the 12-bit displacement. The instruction now takes the 16-bit scalar from this location and sign-extends it, as all integer operations are on 64-bit quantities. The 16-bit sign-extended scalar is now

FFFFFFFFFFFF8000

The sign-extended scalar is then multiplied with R11. The instruction generates an INTEGER OVERFLOW exception if a carry is propagated out of the sign bit. This is the case here. The exception is trapped if the INTEGER OVERFLOW trap is enabled in the Process Status Word, otherwise the result is placed into R11 and the integer overflow is ignored.

The MUL.16 instruction used in the previous example would be encoded as follows:

D98B9032

```
where     D  = addressing mode
         98  = opcode for MUL.16
          B  = source/target register
          9  = base register
        032  = displacement
```

5 DATA TRANSFER INSTRUCTIONS

The instructions described in this chapter are used to move data. These instructions are generally classified according to the data types that they operate on, as listed below.

The monadic transfer instructions move data in byte-size quanta between registers, from a register to memory, or from memory to a register.

Bit field insertion and extraction instructions allow the manipulation of data fields having variable length and placement within a register.

Mutual exclusion instructions allow uninterruptable swaps between a register and memory. These are typically used to 'semaphore' between co-operating processes.

The byte string copy instructions allow the movement of string type data.

Monadic Transfer Instructions

LD	Load Register Sign Extended
LDZ	Load Register Zero Extended
ST	Store Register
STI	Store Immediate
STIN	Store Immediate Negated
STV	Store Register with Overflow Check

Bit Field Insertion and Extraction

INSERT	Insert Bit Field
EXTRACT	Extract Bit Field, Sign Extended
EXTRACTZ	Extract Bit Field, Zero Extended

Mutual Exclusion Instructions

EXCH	Exchange Register and Memory
EXCH.AND	
EXCH.OR	

Byte String Copy Instructions

COPYB	Copy Byte String
COPYB.CONST	Copy Constant Byte String

5.1 MONADIC TRANSFER INSTRUCTIONS

These instructions copy byte-oriented data between registers or between a register and memory. LDx instructions move data into a register specified by OP1. STx instructions move data into a memory location specified by 'Last Operand'. Register to register loads for non-fullword operands and register to register stores are undefined.

5.1.1 LD Load Register Sign Extended
5.1.2 LDZ Load Register Zero Extended

Copy 'Last Operand' into Operand 1, low order justified. Sign extend the result in the 64-bit destination operand for LD, and zero extend the result for LDZ.

Opcode				Implementation
LD.8	LD.16	LD.32	LD.64	
			01x	Rx <- Rw
			3B0	Rx <- Rz
480	490	4A0	4B0	Rx <- e{16}
			5B0	Rx <- =e{12}
680	690	6A0	6B0	Rx <- e{32}
		81x		Rx <- [SP]e{6}
		A1x		Rx <- [Rw]
B80	B90	BA0	BB0	Rx <- [Rz]
C80	C90	CA0	CB0	Rx <- [Ry][Rz]
D80	D90	DA0	DB0	Rx <- [Ry]e{12}
E80	E90	EA0	EB0	Rx <- [Ry][Rz]e{32}
F80	F90	FA0	FB0	Rx <- [Rz]e{32}

Opcode							Implementation
LDZ.8	LDZ.16	LDZ.24	LDZ.32	LDZ.40	LDZ.48	LDZ.56	
481	491	4B1	4A1	497	4A7	4B7	Rx <- e{16}
681	691	6B1	6A1	697	6A7	6B7	Rx <- e{32}
91x							Rx <- [SP]e{6}
B81	B91	BB1	BA1	B97	BA7	BB7	Rx <- [Rz]
C81	C91	CB1	CA1	C97	CA7	CB7	Rx <- [Ry][Rz]
D81	D91	DB1	DA1	D97	DA7	DB7	Rx <- [Ry]e{12}
E81	E91	EB1	EA1	E97	EA7	EB7	Rx <- [Ry][Rz]e{32}
F81	F91	FB1	FA1	F97	FA7	FB7	Rx <- [Rz]e{32}

Instruction Specific Exceptions: none

5.1.3 ST Store Register

Copy the rightmost (low order) bits of Operand 1 into 'Last Operand'. Stores into registers are undefined for this instruction.

Opcode								Implementation	
ST.8	ST.16	ST.24	ST.32	ST.40	ST.48	ST.56	ST.64		
460	461	46C	462	46D	46E	46F	463	e{16}	<- Rx
660	661	66C	662	66D	66E	66F	663	e{32}	<- Rx
87x			86x					[SP]e{6}	<- Rx
			A7x					[Rw]	<- Rx
B60	B61	B6C	B62	B6D	B6E	B6F	B63	[Rz]	<- Rx
C60	C61	C6C	C62	C6D	C6E	C6F	C63	[Ry][Rz]	<- Rx
D60	D61	D6C	D62	D6D	D6E	D6F	D63	[Ry]e{12}	<- Rx
E60	E61	E6C	E62	E6D	E6E	E6F	E63	[Ry][Rz]e{32}	<- Rx
F60	F61	F6C	F62	F6D	F6E	F6F	F63	[Rz]e{32}	<- Rx

Instruction Specific Exceptions: none

5.1.4 STV Store Register with Overflow Check

Copy the low order 8, 16, or 32 bits to 'Last Operand', and then check Operand 1 (Rx) for Integer Overflow. Overflow occurs if the upper 57, 49, or 33 bits are not the same. For a discussion on exception handling, refer to Chapter 2, "Architecture".

Opcode			Implementation	
STV.8	STV.16	STV.32		
464	465	466	e{16}	<- Rx
664	665	666	e{32}	<- Rx
		96x	[SP]e{6}	<- Rx
		A6x	[Rw]	<- Rx
B64	B65	B66	[Rz]	<- Rx
C64	C65	C66	[Ry][Rz]	<- Rx
D64	D65	D66	[Ry]e{12}	<- Rx
E64	E65	E66	[Ry][Rz]e{32}	<- Rx
F64	F65	F66	[Rz]e{32}	<- Rx

Instruction Specific Exceptions: INTEGER OVERFLOW

5.1.5 STI Store Immediate

Store the value of the unsigned 4-bit Rx field into the 'Last Operand', and zero extend to the appropriate size. A STORE IMMEDIATE of 0 has the effect of a CLEAR instruction. The range for Operand 1 is (0 to 15).

Opcode				Implementation	
STI.8	STI.16	STI.32	STI.64		
			05x	Rw	<- =Rx (0..15)
			353	Rz	<- =Rx (0..15)
450	451	452	453	e{16}	<- =Rx (0..15)
650	651	652	653	e{32}	<- =Rx (0..15)
		85x		[SP]e{6}	<- =Rx (0..15)
B50	B51	B52	B53	[Rz]	<- =Rx (0..15)
C50	C51	C52	C53	[Ry][Rz]	<- =Rx (0..15)
D50	D51	D52	D53	[Ry]e{12}	<- =Rx (0..15)
E50	E51	E52	E53	[Ry][Rz]e{32}	<- =Rx (0..15)
F50	F51	F52	F53	[Rz]e{32}	<- =Rx (0..15)

Instruction Specific Exceptions: none

5.1.6 STIN Store Immediate Negated

Zero extend and then form the 1's complement of the unsigned 4-bit Rx field, storing the result into 'Last Operand'. The range for immediate operands is (-16 to -1), as the one's complement biases the value of the 4-bit operand by -1.

Opcode				Implementation	
STIN.8	STIN.16	STIN.32	STIN.64		
			15x	Rw	<- 1's cp =Rx
			357	Rz	<- 1's cp =Rx
454	455	456	457	e{16}	<- 1's cp =Rx
654	655	656	657	e{32}	<- 1's cp =Rx
B54	B55	B56	B57	[Rz]	<- 1's cp =Rx
C54	C55	C56	C57	[Ry][Rz]	<- 1's cp =Rx
D54	D55	D56	D57	[Ry]e{12}	<- 1's cp =Rx
E54	E55	E56	E57	[Ry][Rz]e{32}	<- 1's cp =Rx
F54	F55	F56	F57	[Rz]e{32}	<- 1's cp =Rx

Instruction Specific Exceptions: none

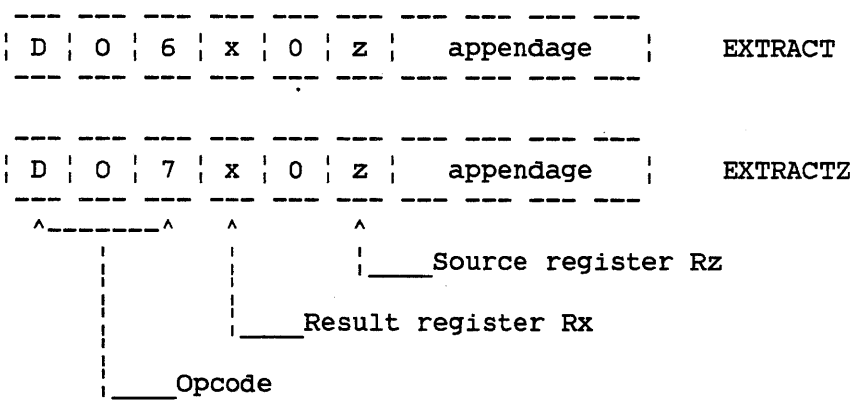
5.2 INSERT AND EXTRACT BIT FIELD OPERATIONS

These instructions move bit fields into or out of registers. The length and starting location of the bit fields are specified in the appendage of the instruction.

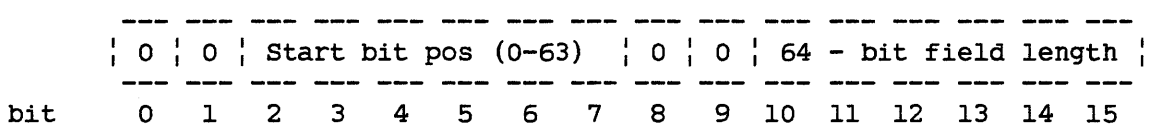
5.2.1 EXTRACT Extract Bit Field, Sign Extended
5.2.2 EXTRACTZ Extract Bit Field, Zero Extended

Extract a bit field from register Rz and store right justified into register Rx. SIGN extend the result for EXTRACT and ZERO extend for EXTRACTZ. The appendage holds a start bit pointer and a length value for the desired extract. The start bit is the leftmost or sign bit position of the bit field, and the length value is 64 minus the bit field length. The range for the start value is 0 to 63. The range for the length value is 1 to 64. The sum of the start bit and the field width must be less than or equal to 64; otherwise, results will be undefined, and implementation dependent side effects will occur.

Opcode assignment	Code	Implementation
EXTRACT	D06	Rx <- Rz:<st:len>, sign extended
EXTRACTZ	D07	Rx <- Rz:<st:len>, zero extended



The appendage has the following format:



There is no data checking of appendages or operands with these instructions. Use with caution.

Instruction Specific Exceptions: None

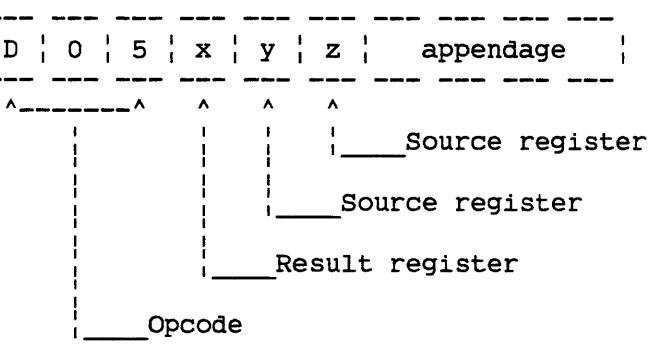
5.2.3 INSERT Insert Bit Field Operation

INSERT takes a right justified bit field from register Rz, displaces it leftward, overlays it onto an image of Ry, and places the result into Rx. The appendage specifies the length of the field and the starting bit position for the overlay. Values in Ry and Rz are unchanged.

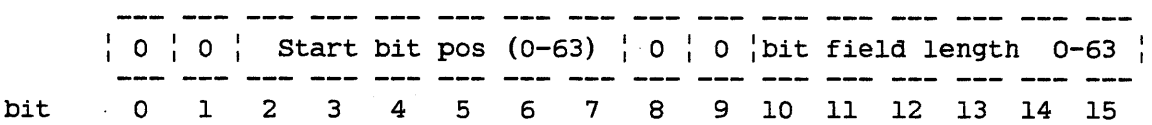
A bit field length of zero is allowed as a NULL case. Use a LD.64 instruction for bit field lengths of 64. High order bits of Rz not in the bit field to be inserted MUST be zero (observe the effect of signed integers in this register). The sum of the start bit and the field width must be less than or equal to 64; otherwise, results will be undefined, and implementation dependent side effects will occur.

Opcode assignment	Code	Implementation
INSERT	D05	Rx <- Rz inserted into Ry

The format of the instruction is



The appendage has the following format:



Typically, these instructions are used to manipulate packed records. Observe the asymmetry of the INSERT and EXTRACT instructions, as INSERT does not check overflow. Also observe the differences in the appendages, keeping in mind that there is no data checking of appendages or operands with these instructions.

Instruction Specific Exceptions: none

5.3 MUTUAL EXCLUSION INSTRUCTIONS

Mutual Exclusion instructions perform logically selective and uninterruptable register-memory swaps. The instructions are valid only on word-aligned, full 64-bit words, that is, the lower 3 bits of the target address must be zero. The operations work on both cacheable and non-cacheable memory operands. In both cases, the execution of the instruction is 'atomic', i.e., the read modify write sequence is indivisible with respect to any other process. If the target is in cache, then only the cache (and not memory) is modified.

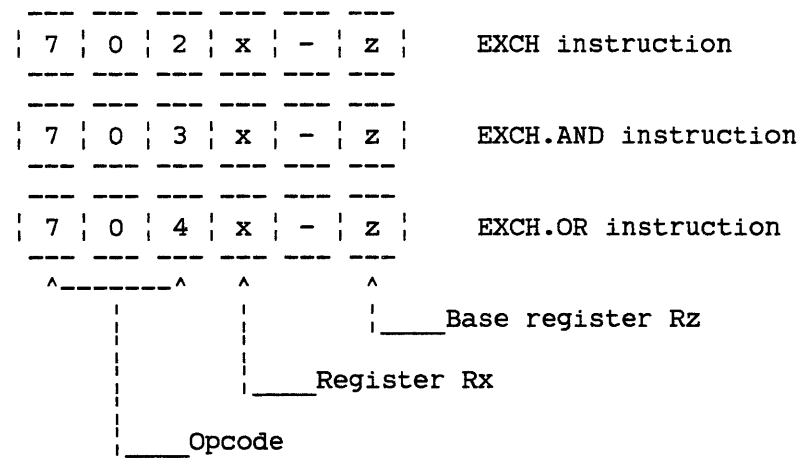
These instructions generally have two types of applications; in the first case to semaphore between processes using a non-cacheable data item as a target, and in the second case to enable a process to coordinate with itself. Using cacheable data items to semaphore between processes is NOT recommended because of critical section problems (one process may access the datum in cache while another process accesses the datum with the same address in memory).

The EXCH operation exchanges Rx with the memory operand pointed to by Rz. For the logical variants, the instruction performs a logical <op> between Rx and the 64-bit memory operand, places the result in Rx, and then EXCHanges Rx with memory. The memory location is specified by a base register.

5.3.1 EXCH Exchange Register and Memory
5.3.2 EXCH.AND
5.3.3 EXCH.OR

Opcode assignment	Code	Implementation
EXCH	702	[Rz] <-> Rx
EXCH.AND	703	[Rz] <-> Rx < Rx AND [Rz] >
EXCH.OR	704	[Rz] <-> Rx < Rx OR [Rz] >

The format of these non-generalized instructions is



Instruction specific exceptions: none

5.4 BYTE STRING COPY INSTRUCTIONS

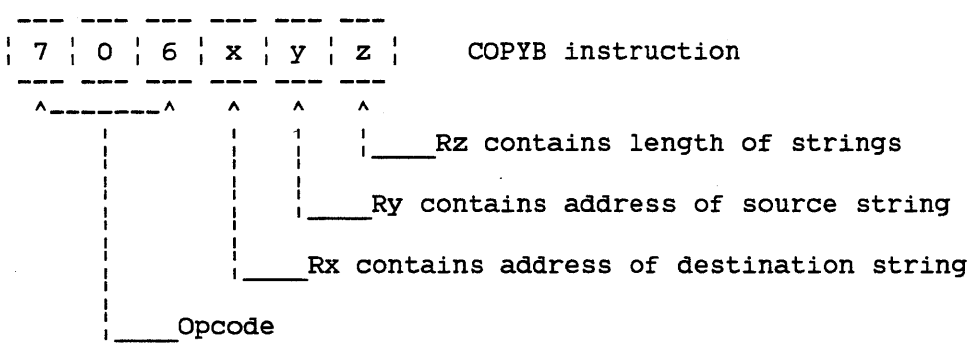
Byte string copy instructions perform memory-to-memory copies of strings (COPYB), or repeatedly copy fullword constants into a string (COPYB.CONST).

5.4.1 COPYB Copy Byte String

Copy the source string into the destination string. Rx contains the address of the destination string, Ry contains the address of the source string, and Rz contains the length of the source and destination strings which are of equal length. The instruction appears to move the string into a temporary location and then to the destination, such that it cannot be used to propagate a fill character through string overlapping.

Opcode assignment	Code	Implementation
COPYB	706	[Rx]:<st> <- [Ry]:<st>, Rz:<len>

The format of this non-generalized instruction is



Instruction Specific Exceptions: none

Special notes:

The contents of Rx, Ry, and Rz are undefined after execution of this instruction.

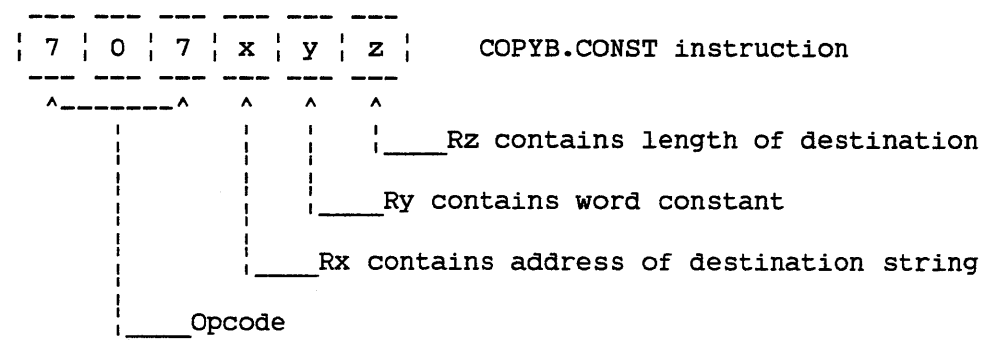
Negative string lengths are treated as if the string has zero length and no data movement takes place. To generate an exception on negative string lengths, use a compare and generate exception instruction.

5.4.2 COPYB.CONST Copy Constant Byte String

Rx contains the address of the destination string, Ry contains an 8-byte source word, and Rz contains the length of the destination string. The instruction appears to perform a word-wise copy into successive destination string words, with the first byte of the source word overlaying the first byte of the destination, until [Rz] bytes have been copied. The length of the destination string need not be a multiple of 8 bytes.

Opcode assignment	Code	Implementation
COPYB.CONST	707	[Rx]:<st> <- Ry, Rz:<len>

The format of this non-generalized instruction is



Instruction specific exceptions: none

Special notes:

The contents of Rx and Rz are undefined after execution of this instruction.

6 INTEGER INSTRUCTIONS

Integer instructions may use 16-bit, 32-bit, or 64-bit source operands to perform 64-bit register arithmetic. Operands less than 64 bits, as specified in the instruction, are first sign extended prior to the operation. The ADD.16 instruction, for example, sign extends the 16-bit source operand to 64 bits and then performs the ADD with 64-bit precision. As such, INTEGER OVERFLOW exceptions are only generated from 64-bit overflows. The overflow is checked in Operand 1 after the result has been placed into the target operand. The MUL.128 and MULU.128 instructions are special cases, operating on 64-bit quantities and returning 128-bit results spanning 2 registers.

To check for overflow in operands less than 64 bits, use a STORE WITH OVERFLOW CHECK instruction with the appropriate modulus. This will generate the exception if the source operand exceeds the size of the target operand.

Integer Instructions

ADD	Integer Addition
ADDUC	Unsigned Integer Addition Generate Carry
ADDI	Integer Addition with Immediate Operand
DIV	Integer Division
DIVR	Reverse Integer Division
MUL	Integer Multiply
MUL	128-Bit Integer Multiply
MULU	128-Bit Unsigned Integer Multiply
NEG	Integer Negate
REM	Remainder of Integer Divide
REMR	Remainder of Reverse Integer Divide
SUB	Integer Subtract
SUBR	Reverse Integer Subtraction
SUBUC	Unsigned Integer Subtract Gen Carry
SUBUCR	Reverse Unsigned Int Sub Gen Carry
SUBI	Integer Subtraction with Immediate

Arithmetic Shift Instructions

SLA	Shift Left Arithmetic
SRA	Shift Right Arithmetic

6.1 MULTIPLE PRECISION INTEGER ARITHMETIC

Normal 64-bit or less arithmetic operations on the ELXSI are simply special cases of multiple precision arithmetic. This is an artifact of the ELXSI carry mechanism and two generic types of instructions; those that operate on unsigned integers and allow carry-out, and those that operate on signed integers and detect overflow. Unsigned integer addition will set the carry bit in the PSW in the event of a carry-out, and otherwise will clear it. Signed integer addition will set the INTEGER OVERFLOW exception in the event of a carry out, and will unconditionally clear the carry bit.

Subtraction is one's complement addition where the subtrahend is complemented and added to the minuend along with the complement of the carry bit. A carry out in unsigned subtraction will clear the carry bit. In this way, the carry may be propagated as the carry bit will again be complemented when added into the next stage. With respect to a carry out, signed subtraction will perform the same action as signed addition.

The user may check the state of the carry bit by retrieving the Process Status Word (READ.STAT) or alternatively, by executing the instruction SUB.64 Rn,Rn. If the carry bit was set, the result is -1, otherwise the result is 0. This is illustrated below with the carry bit set:

```
  10110011   Rn
+ 01001100   complement of Rn
+        0   complement of carry bit in PSW
  --------
  11111111
```

Use of the SUB.64 instruction will always clear the carry bit. The above example will also set the INTEGER OVERFLOW exception if the carry bit tested was clear. If the SUBUC.64 instruction was used instead, the effect would be to toggle the carry bit in the PSW.

In the context of multiple precision arithmetic, the lower stages of the problem would first be computed using the unsigned integer instructions. In this way, a carry may be propagated to the next stage. When the high order stage is reached, the signed integer instructions are used to clear the carry bit and to detect overflow.

An example follows for chained subtraction:

```
minuend =     0000112222222222 2222222222222222
subtrahend =  0000000000000000 3222222222222222
```

```
R0 = low  order word of minuend
R1 = high order word of minuend
R2 = low  order word of subtrahend
R3 = high order word of subtrahend
```

Carry bit in PSW = 0

The first step is to use unsigned subtraction on the low order words of the subtrahend and minuend. The SUBUC.64 instruction using mode 3 is suitable for this. Making these assignments:

```
Rx = target = R4
Ry = low order word of minuend = R0
Rz = low order word of subtrahend = R2
```

The instruction is encoded as 31A402

The result in R4 after execution is F000000000000000 with no carry.

The absence of a carry out sets the carry bit in the PSW. The next step is to subtract the high order words as signed integers using the SUB.64 instruction. This is to enable the detection of overflow. Using addressing mode 3 and making these assignments:

```
Rx = target = R5
Ry = high order word of minuend = R1
Rz = high order word of subtrahend = R3
```

The instruction is encoded as 3BA513

When this instruction is executed, it checks to see if a carry was generated by the previous stage and performs the following:

```
     0000112222222222 = R1
   + FFFFFFFFFFFFFFFF = R3 complemented
     ----------------
     0000112222222221
   +                0   (no carry in)
     ----------------
     0000112222222221 = high order result in R5
```

The final result, concatenating R5 and R4, is

```
     0000112222222221 F000000000000000
```

At this point, the carry bit cleared in the PSW, as the SUB.64 instruction clears the carry. Integer overflow is not generated as the carry did not propagate to the sign bit.

6.2 INTEGER ADDITION

Add the source operands with the carry bit, place the result into Operand 1, and then check for overflow. In the 2 operand forms, Operand 1 is replaced by the sum of Operand 1 and Operand 2. In the 3 operand forms, Operand 1 is replaced by the sum of Operand 2 and Operand 3.

The carry bit is unconditionally cleared for ADD instructions, but set for ADDUC only if a '1' is carried out of bit '0'. Operands are unsigned for ADDUC; signed integer operations may give unexpected results if the carry bit is set.

6.2.1 ADD Integer Addition
6.2.2 ADDUC Unsigned Integer Addition Generate Carry

Opcode				Implementation	
ADD.16	ADD.32	ADD.64	ADDUC.64		
		0Bx		Rx <- Rx+Rw	+ carry
		3B9	319	Rx <- Ry+Rz	+ carry
499	4A9	4B9	419	Rx <- Rx+e{16}	+ carry
		5B9	519	Rx <- Ry+=e{12}	+ carry
699	6A9	6B9	619	Rx <- Ry+e{32}	+ carry
		7B9	719	Rx <- Ry+=e{32}	+ carry
	89x			Rx <- Rx+[SP]e{6}	+ carry
	A9x			Rx <- Rx+[Rw]	+ carry
B99	BA9	BB9	B19	Rx <- Ry+[Rz]	+ carry
C99	CA9	CB9	C19	Rx <- Rx+[Ry][Rz]	+ carry
D99	DA9	DB9	D19	Rx <- Rx+[Ry]e{12}	+ carry
E99	EA9	EB9	E19	Rx <- Rx+[Ry][Rz]e{32}	+ carry
F99	FA9	FB9	F19	Rx <- Ry+[Rz]e{32}	+ carry

Instruction Specific Exceptions, ADD: INTEGER OVERFLOW
ADDUC: none

6.2.3 ADDI Integer Addition with Immediate Operand

Sum Rw with the immediate value in the four bit Rx specification field, place the result into Rw, clear the carry bit, and then check for integer overflow. The range of Rx is 0 to 15.

ADDI has the effect of an INCREMENT instruction if in the form ADDI.64 1,Rw. There is no direct analogue for this instruction in the generalized addressing modes, although an ADD.64 with modes 5 or 7 could be used.

Opcode	Implementation
ADDI.64	
04x	Rw <- Rw + =Rx

Instruction Specific Exceptions: INTEGER OVERFLOW

6.3 INTEGER DIVISION

Divide the source operands, place the result into Operand 1, and then check for integer overflow. DIV and DIVR perform identically with the exception of the source operand division order. In the 2 operand forms, Operand 1 is replaced by Operand 1 divided by Operand 2 (Operand 2 divided by Operand 1 for DIVR). In the 3 operand forms, Operand 1 is replaced by Operand 2 divided by Operand 3 (Operand 3 divided by Operand 2 for DIVR).

6.3.1 DIV Integer Division
6.3.2 DIVR Reverse Integer Division

Opcode			Implementation
DIV.16	DIV.32	DIV.64	
		3BC	Rx <- Ry/Rz
49C	4AC	4BC	Rx <- Ry/e{16}
		5BC	Rx <- Ry/=e{12}
69C	6AC	6BC	Rx <- Ry/e{32}
		7BC	Rx <- Ry/=e{32}
BCC	BAC	BBC	Rx <- Ry/[Rz]
C9C	CAC	CBC	Rx <- Rx/[Ry][Rz]
D9C	DAC	DBC	Rx <- Rx/[Ry]e{12}
E9C	EAC	EBC	Rx <- Rx/[Ry][Rz]e{32}
F9C	FAC	FBC	Rx <- Ry/[Rz]e{32}

Opcode			Implementation
DIVR.16	DIVR.32	DIVR.64	
49D	4AD	4BD	Rx <- e{16}/Ry
		5BD	Rx <- =e{12}/Ry
69D	6AD	6BD	Rx <- e{32}/Ry
		7BD	Rx <- =e{32}/Ry
B9D	BAD	BBD	Rx <- [Rz]/Ry
C9D	CAD	CBD	Rx <- [Ry][Rz]/Rx
D9D	DAD	DBD	Rx <- [Ry]e{12}/Rx
E9D	EAD	EBD	Rx <- [Ry][Rz]e{32}/Rx
F9D	FAD	FBD	Rx <- [Rz]e{32}/Ry

Instruction Specific Exceptions: INTEGER OVERFLOW
INTEGER ZERO DIVIDE

6.4 INTEGER MULTIPLY OPERATIONS

Multiply the source operands, place the result into Operand 1, and then check for integer overflow. In the 2 operand forms, Operand 1 is replaced by Operand 1 multiplied by Operand 2. In the 3 operand forms, Operand 1 is replaced by Operand 2 multiplied by Operand 3.

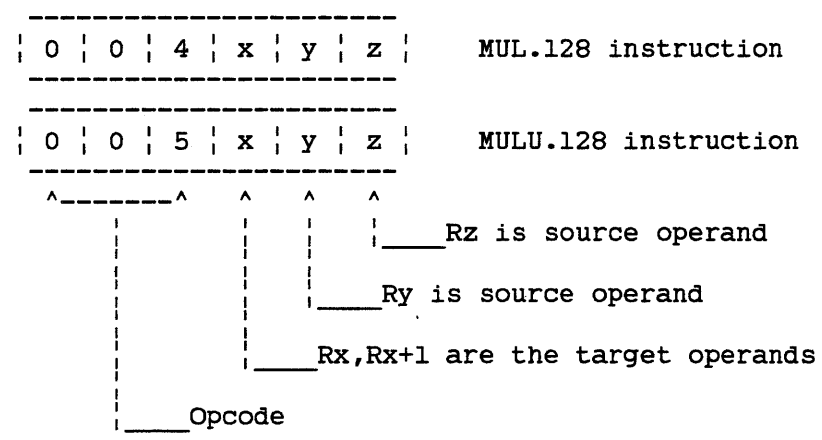
For MUL.128 and MULU.128, multiply the 64-bit source operands in Ry and Rz, with the 128-bit result replacing Rx and Rx+1. Rx receives the high order 64 bits of the result, and Rx+1 receives the low order 64 bits. MUL.128 assumes signed integer operands, MULU.128 assumes unsigned integer operands.

6.4.1 MUL Integer Multiply
6.4.2 MUL.128 128-Bit Integer Multiply
6.4.3 MULU.128 128-Bit Unsigned Integer Multiply

Opcode			Implementation
MUL.16	MUL.32	MUL.64	
		0Bx	Rx <- Rx * Rw
		3B8	Rx <- Ry * Rz
498	4A8	4B8	Rx <- Rx * e{16}
		5B8	Rx <- Ry * =e{12}
698	6A8	6B8	Rx <- Ry * e{32}
		7B8	Rx <- Ry * =e{32}
B98	BA8	BB8	Rx <- Ry * [Rz]
C98	CA8	CB8	Rx <- Rx * [Ry][Rz]
D98	DA8	DB8	Rx <- Rx * [Ry]e{12}
E98	EA8	EB8	Rx <- Rx * [Ry][Rz]e{32}
F98	FA8	FB8	Rx <- Ry * [Rz]e{32}

Opcode assignment	Code	Implementation
MUL.128	B02	Rx,Rx+1 <- Ry * Rz
MULU.128	B03	Rx,Rx+1 <- Ry * Rz

The formats of the non-generalized instructions are



Instruction Specific Exceptions, MUL: INTEGER OVERFLOW
MUL.128: none
MULU.128: none

6.5 NEG INTEGER NEGATE

Replace Operand 1 with the two's complement of 'Last Operand', and then check for integer overflow. The carry bit is unreferenced and unmodified by this instruction.

Opcode			Implementation
NEG.16	NEG.32	NEG.64	
		3B2	Rx <- two's cp of Rz
492	4A2	4B2	Rx <- two's cp of e{16}
692	6A2		Rx <- two's cp of e{32}
B92	BA2		Rx <- two's cp of [Rz]
C92	CA2	CB2	Rx <- two's cp of [Ry][Rz]
D92	DA2	DB2	Rx <- two's cp of [Ry]e{12}
E92	EA2	EB2	Rx <- two's cp of [Ry][Rz]e{32}
F92	FA2		Rx <- two's cp of [Rz]e{32}

Instruction Specific Exceptions: INTEGER OVERFLOW

6.6 REMAINDER OF INTEGER DIVIDE OPERATIONS

Divide the source operands and place the remainder into Operand 1. REM and REMR perform identically with the exception of the order of source operand division. In the 2 operand form, Operand 1 is replaced with the remainder of Operand 1 divided by Operand 2 (Operand 2 divided by Operand 1 for REMR). In the 3 operand form, Operand 1 is replaced with the remainder of Operand 2 divided by Operand 3 (Operand 3 divided by Operand 2 for REMR).

The quotient is rounded towards zero, and the dividend and remainder have the same sign if both are non-zero. This is the FORTRAN remainder.

6.6.1 REM Remainder of Integer Divide
6.6.2 REMR Remainder of Reverse Integer Divide

Opcode			Implementation
REM.16	REM.32	REM.64	
		3BE	Rx <- rem of Ry/Rz
49E	4AE	4BE	Rx <- rem of Rx/e{16}
		5BE	Rx <- rem of Ry/=e{12}
69E	6AE	6BE	Rx <- rem of Ry/e{32}
		7BE	Rx <- rem of Ry/=e{32}
B9E	BAE	BBE	Rx <- rem of Ry/[Rz]
C9E	CAE	CBE	Rx <- rem of Rx/[Ry][Rz]
D9E	DAE	DBE	Rx <- rem of Rx/[Ry]e{12}
E9E	EAE	EBE	Rx <- rem of Rx/[Ry][Rz]e{32}
F9E	FAE	FBE	Rx <- rem of Ry/[Rz]e{32}

Opcode			Implementation
REMR.16	REMR.32	REMR.64	
49F	4AF	4BF	Rx <- rem of e{16}/Rx
		5BF	Rx <- rem of =e{12}/Ry
69F	6AF	6BF	Rx <- rem of e{32}/Ry
		7BF	Rx <- rem of =e{32}/Ry
B9F	BAF	BBF	Rx <- rem of [Rz]/Ry
C9F	CAF	CBF	Rx <- rem of [Ry][Rz]/Rx
D9F	DAF	DBF	Rx <- rem of [Ry]e{12}/Rx
E9F	EAF	EBF	Rx <- rem of [Ry][Rz]e{32}/Rx
F9F	FAF	FBF	Rx <- rem of [Rz]e{32}/Ry

Instruction Specific Exceptions: INTEGER ZERO DIVIDE

6.7 INTEGER SUBTRACT OPERATIONS

Replace Operand 1 with the difference of the source operands, and then check for integer overflow. SUB and SUBR behave identically with exception of the order of source operand subtraction. In 2 operand forms, replace Operand 1 with Operand 1 minus Operand 2 (Operand 2 minus Operand 1 for SUBR). In 3 operand forms, replace Operand 1 with Operand 2 minus Operand 3 (Operand 3 minus Operand 2 for SUBR). The carry bit is cleared with these instructions.

6.7.1 SUB Integer Subtract
6.7.2 SUBR Reverse Integer Subtract

Opcode			Implementation	
SUB.16	SUB.32	SUB.64		
		0Ax	Rx <- Rx-Rw	- carry
		3BA	Rx <- Ry-Rz	- carry
49A	4AA	4BA	Rx <- Ry-e{16}	- carry
		5BA	Rx <- Ry-=e{12}	- carry
69A	6AA	6BA	Rx <- Ry-e{32}	- carry
		7BA	Rx <- Ry-=e{32}	- carry
	8Ax		Rx <- Rx-[SP]e{6}	- carry
	AAx		Rx <- Rx-[Rw]	- carry
B9A	BAA	BBA	Rx <- Ry-[Rz]	- carry
C9A	CAA	CBA	Rx <- Rx-[Ry][Rz]	- carry
D9A	DAA	DBA	Rx <- Rx-[Ry]e{12}	- carry
E9A	EAA	EBA	Rx <- Rx-[Ry][Rz]e{32}	- carry
F9A	FAA	FBA	Rx <- Ry-[Rz]e{32}	- carry

Opcode			Implementation	
SUBR.16	SUBR.32	SUBR.64		
		0Bx	Rx <- Rw-Rx	- carry
49B	4AB	4BB	Rx <- e{16}-Rx	- carry
		5BB	Rx <- =e{12}-Ry	- carry
69B	6AB	6BB	Rx <- e{32}-Ry	- carry
		7BB	Rx <- =e{32}-Ry	- carry
	8Bx		Rx <- [SP]e{6}-Rx	- carry
	ABx		Rx <- [Rw]-Rx	- carry
B9B	BAB	BBB	Rx <- [Rz]-Ry	- carry
C9B	CAB	CBB	Rx <- [Ry][Rz]-Rx	- carry
D9B	DAB	DBB	Rx <- [Ry]e{12}-Rx	- carry
E9B	EAB	EBB	Rx <- [Ry][Rz]e{32}-Rx	- carry
F9B	FAB	FBB	Rx <- [Rz]e{32}-Ry	- carry

Instruction Specific Exceptions: INTEGER OVERFLOW

6.7.3 SUBUC Unsigned Integer Subtract Gen Carry
6.7.4 SUBUCR Reverse Unsigned Integer Subtract Gen Carry

Replace Operand 1 with the difference of the unsigned source operands. SUBUC and SUBUCR behave identically except for the order of source operand subtraction. In 2 operand forms, replace Operand 1 with Operand 1 minus Operand 2 (Operand 2 minus Operand 1 for SUBUCR). In 3 operand forms, replace Operand 1 with Operand 2 minus Operand 3 (Operand 3 minus Operand 2 for SUBUCR). A carry-out clears the carry bit, otherwise it is set.

Opcode		
SUBUC.64	Implementation	
31A	Rx <- Ry-Rz	- carry
41A	Rx <- Rx-e{16}	- carry
51A	Rx <- Ry-=e{12}	- carry
61A	Rx <- Ry-e{32}	- carry
71A	Rx <- Ry-=e{32}	- carry
B1A	Rx <- Ry-[Rz]	- carry
C1A	Rx <- Rx-[Ry][Rz]	- carry
D1A	Rx <- Rx-[Ry]e{12}	- carry
E1A	Rx <- Rx-[Ry][Rz]e{32}	- carry
F1A	Rx <- Ry-[Rz]e{32}	- carry

Opcode SUBUCR.64	Implementation
41B	Rx <- e{16}-Rx - carry
51B	Rx <- =e{12}-Ry - carry
61B	Rx <- e{32}-Ry - carry
71B	Rx <- =e{32}-Ry - carry
B1B	Rx <- [Rz]-Ry - carry
C1B	Rx <- [Ry][Rz]-Rx - carry
D1B	Rx <- [Ry]e{12}-Rx - carry
E1B	Rx <- [Ry][Rz]e{32}-Rx - carry
F1B	Rx <- [Rz]e{32}-Ry - carry

Instruction Specific Exceptions: none

6.7.5 SUBI Integer Subtraction with Immediate Operand

Subtract from Rw the immediate value plus one in the four bit Rx specification field, then place the result in Rw and clear the carry bit. When this operation is completed, check for integer overflow. The range of the effective operand is -16 to -1.

SUBI has the effect of a DECREMENT instruction if in the form SUBI.64 Rw,=1. There is no analogue in the generalized addressing modes for this instruction.

Opcode SUBI.64	Implementation
14x	Rw <- Rw - (=Rx+1)

Instruction Specific Exceptions: INTEGER OVERFLOW

6.8 ARITHMETIC SHIFT INSTRUCTIONS

Arithmetic shift operations operate on 64-bit signed integers. All shift distances are unsigned integers modulo 64.

In two operand forms, Operand 1 is replaced by the value of Operand 1 shifted by Operand 2. In three operand forms, Operand 1 is replaced by the value of Operand 2 shifted by Operand 3. The fill bit is zero for SLA, and is the sign bit in the operand to be shifted for SLR.

6.8.1 SLA Shift Left Arithmetic
6.8.2 SRA Shift Right Arithmetic

Opcode SLA	Implementation
312	Rx <- Ry shifted left by Rz
512	Rx <- Ry shifted left by =e{12}
712	Rx <- Ry shifted left by =e{32}

Opcode SRA	Implementation
313	Rx <- Ry shifted right by Rz
513	Rx <- Ry shifted rignt by =e{12}
713	Rx <- Ry shifted right by =e{32}

Special Notes:

Overflow operations may occur with SLA and is equivalent to overflow in integer multiply. Overflow is checked after the target operand is written with the result. The overflow exception is generated if the sign bit changes as a result of a shift. This allows multiplication by powers of two with the detection of overflow. Note that the Shift Left Logical instruction does not generate this exception.

A shift right by n with the SRA instruction will perform a division by 2**n on a positive number. However, a shift right by n on a negative number will result in a rounding toward negative infinity. Thus it is necessary to adjust the number prior to the shift. To correctly perform a fast division by two on a negative number, precede the shift by n with an ADD of (2**n)-1.

Instruction Specific Exceptions, SLA: INTEGER OVERFLOW
SRA: none

7 FLOATING POINT INSTRUCTIONS

The floating point operations conform to the proposed IEEE Floating Point Standard. The three floating point formats provided are: single (32 bits), double (64 bits), and double extended (80 bits). Refer to Chapter 3, "Data Representations", for a description of the standard and the encodings for numbers and symbols. Refer to Section 2.2, "Process Status Word", for a discussion of exception handling.

FLOATING POINT INSTRUCTIONS

FADD	Floating Point Addition
FDIV	Floating Point Division
FDIVR	Floating Point Division Reversed
FMUL	Floating Point Multiply
FREM	Floating Point Remainder
FSQR	Floating Point Square Root
FSUB	Floating Point Subtraction
FSUBR	Floating Point Subtraction Reversed

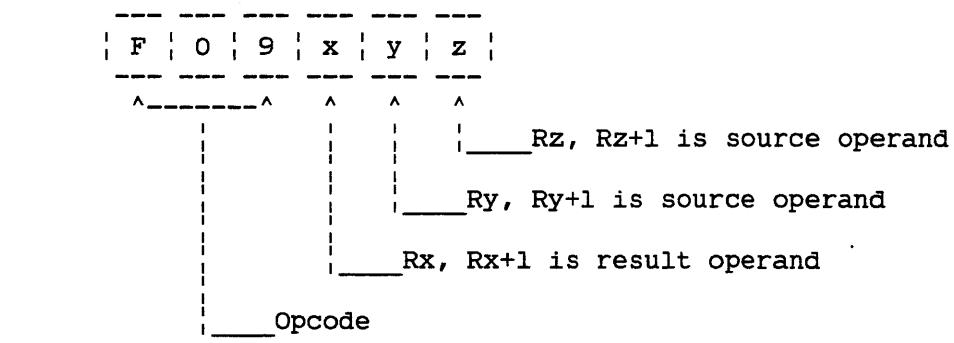
7.1 FADD FLOATING POINT ADDITION

Add the source operands and place into Operand 1. With the two address forms, operands 1 and 2 are the sources. With the three operand forms, operands 2 and 3 are the sources. The result matrix is shown on the following page.

Opcode		
FADD.32	FADD.64	Implementation
19x	1Dx	Rx <- Rx + Rw
3C9	3D9	Rx <- Ry + Rz
4C9	4D9	Rx <- Rx + e{16}
6C9	6D9	Rx <- Ry + e{32}
7C9		Rx <- Ry + =e{32}
99x		Rx <- Rx + [SP]e{6}
BC9	BD9	Rx <- Ry + [Rz]
CC9	CD9	Rx <- Rx + [Ry][Rz]
DC9	DD9	Rx <- Rx + [Ry]e{12}
EC9	ED9	Rx <- Rx + [Ry][Rz]e{32}
FC9	FD9	Rx <- Ry + [Rz]e{32}

Opcode assignment	Code	Implementation
FADD.80	F09	(Rx,Rx+1) <- (Ry,Ry+1) + (Rz,Rz+1)

The format of the non-generalized FADD.80 instruction is



Instruction Specific Exceptions: FLOATING POINT INVALID OPERATION
FLOATING POINT INEXACT RESULT
FLOATING POINT UNDERFLOW
FLOATING POINT OVERFLOW

Table 7-1. Result Matrix for FADD, FSUB and FSUBR

If OP1 is	and OP2 is	then result is	
Zero	Zero	Zero with sign	(1)
Zero	Denorm, Normal	OP2	
Zero	Infinity	OP2	
Denorm, Normal	Zero	OP1	
Denorm, Normal	Denorm, Normal	computed	(1)
Denorm, Normal	Infinity	OP2	
Infinity	Zero	OP1	
Infinity	Denorm, Normal	OP1	
Infinity	Infinity	OP1 or Invalid	(2)

Magnitude subtraction occurs if the operands' signs are different in FADD or the same in FSUB or FSUBR.

(1) If magnitude subtraction produces a zero result, then the sign is determined by the rounding mode. The result is -0 if the rounding mode is toward minus infinity, and +0 otherwise. Magnitude addition preserves the sign of the operands.

(2) Magnitude subtraction of two Infinities causes an Invalid Operation exception.

7.2 FLOATING POINT DIVISION

Divide the source operands and place the result into Operand 1. With the two operand forms, Operand 1 is replaced by Operand 1 divided by Operand 2 (Operand 2 divided by Operand 1 for FDIVR). With the three operand forms, Operand 1 is replaced by Operand 2 divided by Operand 3 (Operand 3 divided by Operand 2 for FDIVR). The result matrix is shown below.

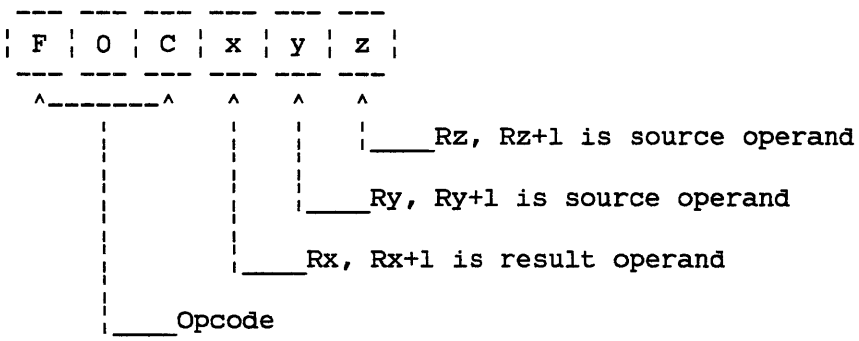
7.2.1 FDIV Floating Point Division
7.2.2 FDIVR Floating Point Division Reversed

Opcode		
FDIV.32	FDIV.64	Implementation
3CC	3DC	Rx <- Ry/Rz
4CC	4DC	Rx <- Rx/e{16}
6CC	6DC	Rx <- Ry/e{32}
7CC		Rx <- Ry/=e{32}
BCC	BDC	Rx <- Ry/[Rz]
CCC	CDC	Rx <- Rx/[Ry][Rz]
DCC	DDC	Rx <- Rx/[Ry]e{12}
ECC	EDC	Rx <- Rx/[Ry][Rz]e{32}
FCC	FDC	Rx <- Ry/[Rz]e{32}

Opcode		
FDIVR.32	FDIVR.64	Implementation
4CD	4DD	Rx <- e{16}/Rx
6CD	6DD	Rx <- e{32}/Ry
7CD		Rx <- =e{32}/Ry
BCD	BDD	Rx <- [Rz]/Ry
CCD	CDD	Rx <- [Ry][Rz]/Rx
DCD	DDD	Rx <- [Ry]e{12}/Rx
ECD	EDD	Rx <- [Ry][Rz]e{32}/Rx
FCD	FDD	Rx <- [Rz]e{32}/Ry

Opcode assignment	Code	Implementation
FDIV.80	F0C	(Rx,Rx+1) <- (Ry,Ry+1)/(Rz,Rz+1)

The format of the non-generalized FDIV.80 instruction is



Instruction Specific Exceptions: FLOATING POINT INVALID OPERATION
FLOATING POINT INEXACT RESULT
FLOATING POINT UNDERFLOW
FLOATING POINT OVERFLOW
FLOATING POINT DIVIDE BY ZERO

Table 7-2. Result Matrix for FDIV and FDIVR

If OP1 is	and OP2 is	then result is
Zero	Zero	Invalid Operation exc.
Zero	Denorm, Normal	Zero with xor signs
Zero	Infinity	Zero with xor signs
Denorm, Normal	Zero	Divide by Zero exception
Denorm, Normal	Denorm, Normal	computed
Denorm, Normal	Infinity	Zero with xor signs
Infinity	Zero	Inf with xor signs
Infinity	Denorm, Normal	Inf with xor signs
Infinity	Infinity	Invalid Operation exc.

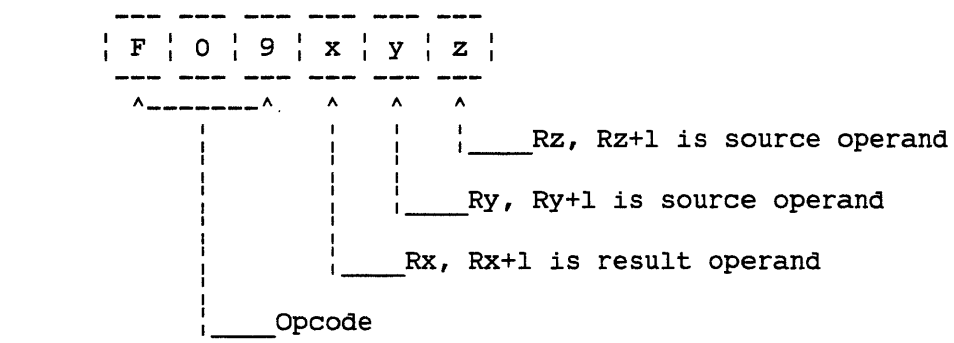
7.3 FMUL FLOATING POINT MULTIPLY

Multiply the source operands and place into Operand 1. With the two address forms, operands 1 and 2 are the sources. With the three operand forms, operands 2 and 3 are the sources. The result matrix is shown on the following page.

Opcode		
FMUL.32	FMUL.64	Implementation
18x	1Cx	Rx <- Rx * Rw
3C8	3D8	Rx <- Ry * Rz
4C8	4D8	Rx <- Rx * e{16}
6C8	6D8	Rx <- Ry * e{32}
7C8		Rx <- Ry * =e{32}
98x		Rx <- Rx * [SP]e{6}
BC8	BD8	Rx <- Ry * [Rz]
CC8	CD8	Rx <- Rx * [Ry][Rz]
DC8	DD8	Rx <- Rx * [Ry]e{12}
EC8	ED8	Rx <- Rx * [Ry][Rz]e{32}
FC8	FD8	Rx <- Ry * [Rz]e{32}

Opcode assignment	Code	Implementation
FMUL.80	F08	(Rx,Rx+1) <- (Ry,Ry+1) * (Rz,Rz+1)

The format of the non-generalized FMUL.80 instruction is



Instruction Specific Exceptions: FLOATING POINT INVALID OPERATION
FLOATING POINT INEXACT RESULT
FLOATING POINT UNDERFLOW
FLOATING POINT OVERFLOW

Table 7-3. Result Matrix for MULTIPLY

If OP1 is	and OP2 is	then result is
Zero	Zero	Zero with xor signs
Zero	Denorm, Normal	Zero with xor signs
Zero	Infinity	Invalid Operation
Denorm, Normal	Zero	Zero with xor signs
Denorm, Normal	Denorm, Normal	computed
Denorm, Normal	Infinity	Inf with xor signs
Infinity	Zero	Invalid Operation
Infinity	Denorm, Normal	Inf with xor signs
Infinity	Infinity	Inf with xor signs

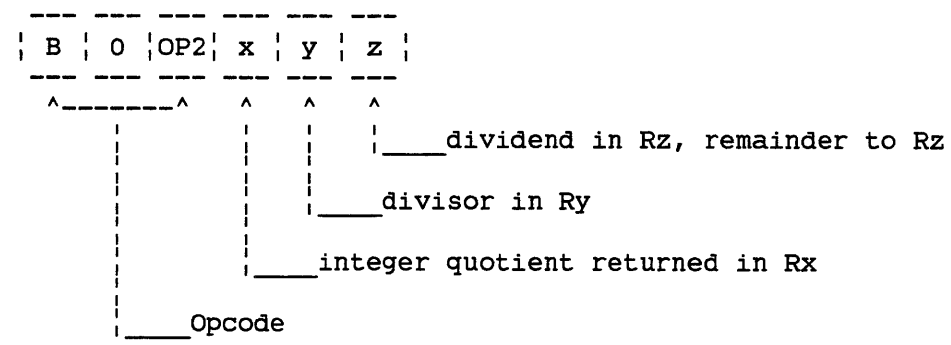
7.4 FREM FLOATING POINT REMAINDER

Divide Rz by Ry (note that FREM is defined by FDIVR) to produce the full-length integer quotient rounded to nearest even. If the rounded quotient is zero, the instruction terminates. Otherwise, Rx is replaced by the low order 64 bits of the rounded two's complement integer quotient and Rz is replaced by the remainder in floating point form. The remainder is defined to be Rz - ((Rz/Ry rounded to the nearest integer) * Ry) so that -0.5 * abs(divisor) <= remainder <= +0.5 * abs(divisor). The remainder need not be rounded because it is exact. A zero remainder has the same sign as the dividend.

The contents of Rx, Rz (and Rz+1 in extended) are undefined if the register specifiers Rx and Rz are the same (Rx and Rz+1 for Double Extended). Integer overflow is ignored when storing the quotient into Rx.

Opcode assignment	Code	Implementation
FREM.32	B04	if Rz/Ry not zero; Rx <- quot, Rz <-rem
FREM.64	B05	if Rz/Ry not zero; Rx <- quot, Rz <-rem
FREM.80	B06	if Rz/Ry not zero; Rx <- quot, Rz <-rem

The format of the non-generalized FREM.xx instruction is



Instruction Specific Exceptions: FLOATING POINT INVALID OPERATION

Table 7-4. Result Matrix for FREM

If OP1 is	and OP2 is	then result is
Zero	Zero	Invalid Operation
Zero	Denorm, Normal	NOP
Zero	Infinity	NOP
Denorm, Normal	Zero	Invalid Operation
Denorm, Normal	Denorm, Normal	computed
Denorm, Normal	Infinity	NOP
Infinity	Zero	Invalid Operation
Infinity	Denorm, Normal	Invalid Operation
Infinity	Infinity	Invalid Operation

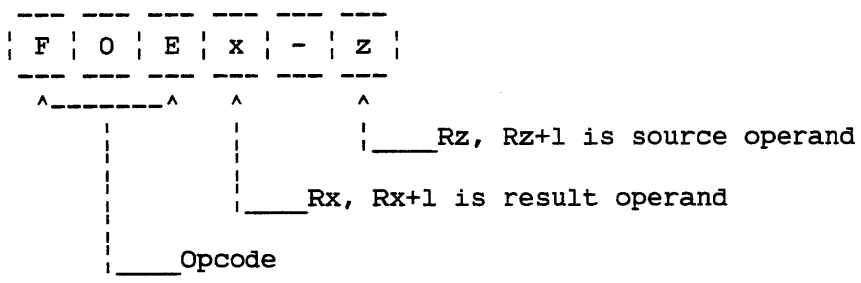
7.5 FSQR FLOATING POINT SQUARE ROOT

Find the square root of the source and place into Operand 1. With the two address forms, Operand 2 is the source. With the three address forms, Operand 3 is the source.

Opcode		
FSQR.32	FSQR.64	Implementation
3CE	3DE	Rx <- Rz**1/2
4CE	4DE	Rx <- e{16}**1/2
6CE	6DE	Rx <- e{32}**1/2
BCE	BDE	Rx <- [Rz]**1/2
CCE	CDE	Rx <- [Ry][Rz]**1/2
DCE	DDE	Rx <- [Ry]e{12}**1/2
ECE	EDE	Rx <- [Ry][Rz]e{32}**1/2
FCE	FDE	Rx <- [Rz]e{32}**1/2

Opcode assignment	Code	Implementation
FSQR.80	F0E	(Rx,Rx+1) <- (Rz,Rz+1)**1/2

The format of the non-generalized FSQR.80 instruction is



Instruction Specific Exceptions: FLOATING POINT INVALID OPERATION
FLOATING POINT INEXACT RESULT

Table 7-5. Result Matrix for FSQR

If OP1 is	then result is
Neg. Infinity	Invalid Operation
Neg. (Denorm, Normal)	Invalid Operation
Zero	OP1 (square root of -0 is -0)
Pos. (Denorm, Normal)	computed
Pos. Infinity	OP1

7.6 FLOATING POINT SUBTRACTION

Place the difference of the source operands into Operand 1. With the two operand forms, Operand 1 is replaced by Operand 1 minus Operand 2 (Operand 2 minus Operand 1 for FSUBR). With the three operand forms, Operand 1 is replaced by Operand 2 minus Operand 3 (Operand 3 minus Operand 2 for FSUBR). The result matrix is shown on the page following the FADD instruction.

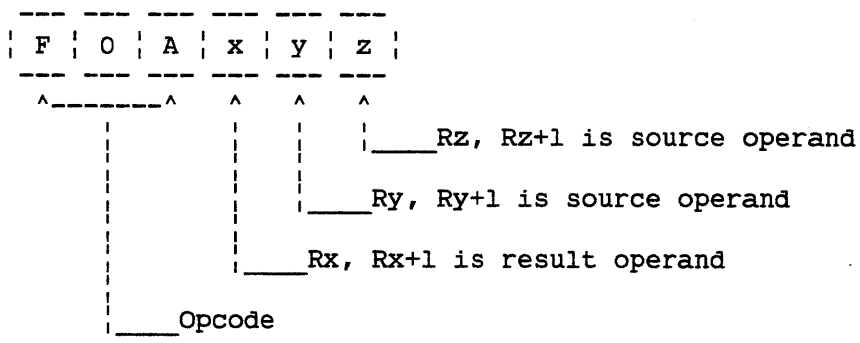
7.6.1 FSUB Floating Point Subtraction
7.6.2 FSUBR Floating Point Subtraction Reversed

Opcode		
FSUB.32	FSUB.64	Implementation
1Ax	1Ex	Rx <- Rx - Rw
3CA	3DA	Rx <- Ry - Rz
4CA	4DA	Rx <- Rx - e{16}
6CA	6DA	Rx <- Ry - e{32}
7CA		Rx <- Ry - =e{32}
9Ax		Rx <- Rx - [SP]e{6}
BCA	BDA	Rx <- Ry - [Rz]
CAC	CDA	Rx <- [Ry] - [Rz]
DAA	DDC	Rx <- [Ry] - e{12}
ECA	EDA	Rx <- [Ry] - [Rz]e{32}
FCA	FDA	Rx <- Ry - [Rz]e{32}

Opcode		Implementation
FSUBR.32	FSUBR.64	
1Ax	1Ex	Rx <- Rw - Rx
4CB	4DB	Rx <- e{16} - Rx
6CB	6DB	Rx <- e{32} - Ry
7CB		Rx <- =e{32} - Ry
9Ax		Rx <- [SP]e{6} - Rx
BCB	BDB	Rx <- [Rz] - Ry
CCB	CDB	Rx <- [Rz] - [Ry]
DCB	DDB	Rx <- e{12} - [Ry]
ECB	EDB	Rx <- [Rz]e{32} - [Ry]
FCB	FDB	Rx <- [Rz]e{32} - Ry

Opcode assignment	Code	Implementation
FSUB.80	F0A	(Rx,Rx+1) <- (Ry,Ry+1) - (Rz,Rz+1)

The format of the non-generalized FSUB.80 instruction is



Instruction Specific Exceptions:
FLOATING POINT INVALID OPERATION
FLOATING POINT INEXACT RESULT
FLOATING POINT UNDERFLOW
FLOATING POINT OVERFLOW

8 ASCII ARITHMETIC INSTRUCTIONS

ASCII arithmetic instructions operate on ASCII encoded decimal strings. Operands are in ASCII format, with the ASCII digits in the set ['00', '30' to '39'] hex. Source operands not conforming to the ASCII format will produce undefined results.

ASCII.ADD	ASCII Addition
ASCII.ADDC	ASCII Addition Generate Carry
ASCII.SUB	ASCII Subtract
ASCII.SUBC	ASCII Subtract Generate Carry

8.1 MULTIPLE PRECISION ASCII SUBTRACTION

ASCII subtraction is implemented through two types of instructions on the ELXSI. The ASCII.SUB instruction returns the magnitude of the result of the subtraction in Rx, with the sign bit of the result returned in Rt. The ASCII.SUBC instruction returns the 9's complement of the result if the result is negative, otherwise it returns the true magnitude of the result.

The ASCII.SUBC instructions would typically be used in a situation where the numeric string to be operated on exceeds the register length, or otherwise where multiple precision subtraction is required. In the majority of cases where ASCII.SUBC is used, the returned result will be positive; if the result is negative, then an additional subtraction is required in order to determine the true magnitude of the result. (See Section 6.1 on multiple precision integer arithmetic for a discussion of the carry mechanism).

As an example to illustrate the postive result case for multiple precision subtraction, 3 is subtracted from 1001. Assuming that the register width is 4 characters,

The example is

```
  0001 0001
- 0000 0003
  ---------
```

Doing the low order subtract,

```
  0001              0001
- 0003  becomes   + 9996  (9's complement of 0003)
  -----           +    1  (complement of decimal carry, initially 0)
                   ----
                   9998
```

Since there is no carry, a 1 is placed in the decimal carry bit (no carry amounts to a borrow from the next segment).

Doing the high order subtract,

```
0001                0001
0000     becomes    9999
                       0  (complement of decimal carry)
                   -----
                    0000
```

Since there is a carry, the result is positive. A zero goes into the sign register. Since this is not a generate carry instruction, a zero is placed in the decimal carry bit. Overflow does not occur since the true result can be represented in the 4 digits.

Another example will illustrate the negative result case for multiple precision subtraction.

The example is

```
  0001 0001
- 0000 0003
  ---------
```

Doing the low order subtract,

```
  1 0001           0001
- 1 0003  becomes  9996
 --------            1  (complement of decimal carry, initially 0)
                   -----
                   9998
```

Since there is no carry, a 1 is placed in the decimal carry bit.

Doing the high order subtract,

```
0001               0001
0001    becomes    9998
                      0  (complement of decimal carry)
                   -----
                   9999
```

Since there is no carry, the result is negative. A one goes into the sign register. Since this is not a generate carry instruction, (ASCII.ADDC) a zero is placed in the decimal carry bit. The target result register receives the absolute value of 9999 (= 0001) as the result is negative. So the result that we have now is

```
negative  0001 9998
```

This is clearly not the result that is desired, since the actual answer is -2. There are two ways to correct this answer, both signaled by a negative result, that is, a 1 in the target sign register. The first is to simply reperform the subtraction, switching the subtrahend and minuend.

The second is to form the 9's complement of the low order segments by subtracting them from 0 with the ASCII.SUBC instruction, and then subtracting 0 from the high order segment, to take into account any borrow that might have been generated.

In the example above, we get

```
  0000           0000
- 9998  becomes  0001
                   1  (complement of decimal carry)
                 -----
                   2
```

Since there is no carry, a 1 is placed in the decimal carry. The high order segment is

```
  0001             0001
- 0000    becomes   9999
                      0  (complement of decimal carry)
                    -----
                    0000
```

Thus, the actual ASCII answer that is wanted is returned

negative 0000 0002

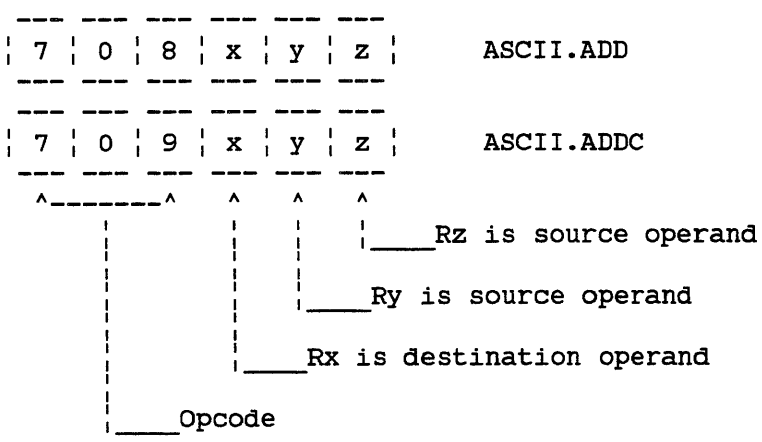
8.2 ASCII ADDITION

Replace Rx with the ASCII decimal sum of Ry + Rz + decimal carry. ASCII.ADD clears the decimal carry-out, whereas ASCII.ADDC sets the carry-out if generated.

8.2.1 ASCII.ADD ASCII Addition
8.2.2 ASCII.ADDC ASCII Addition Generate Carry

Opcode assignment	Code	Implementation
ASCII.ADD	708	Rx <- Ry + Rz + carry
ASCII.ADDC	709	Rx <- Ry + Rz + carry

The formats of these non-generalized instructions are



Instruction Specific Exceptions: None

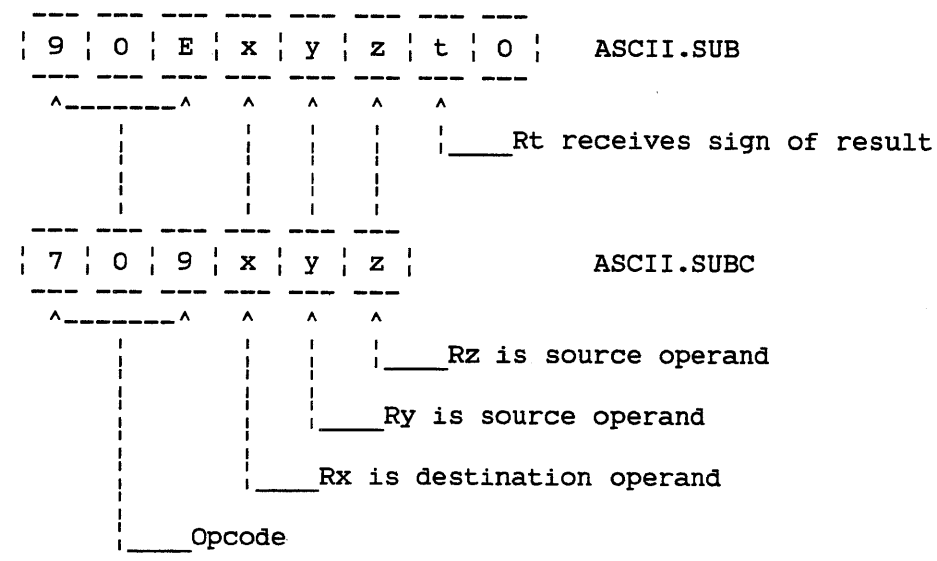
8.3 ASCII SUBTRACTION

Extract the decimal value from Ry and sum with the decimal value 9's complemented from Rz + the complement of the decimal carry. For ASCII.SUB, place the ASCII encoded magnitude of the sum into Rx, place the sign of the result into Rt (0 for a positive result, 1 for a negative result), and clear the decimal carry. For ASCII.SUBC, place the ASCII encoded sum into Rx and set the decimal carry to the complement of the carryout of the sum.

8.3.1 ASCII.SUB ASCII Subtract
8.3.2 ASCII.SUBC ASCII Subtract Generate Carry

Opcode assignment	Code	Implementation
ASCII.SUB	90E	Rx <- mag(Ry - Rz - carry); Rt <- sign
ASCII.SUBC	70B	Rx <- Ry - Rz - carry

The formats of these non-generalized instructions are



It is important to note that the ASCII.SUB instruction behaves differently than the ASCII.SUBC instruction, in that the MAGNITUDE of the result is returned. There is no difference when the result is positive, but when it is negative, ASCII.SUBC returns the 9's complement of the magnitude, while this instruction returns the sign as negative and the true magnitude of the result. This is most often exactly what is preferred when ASCII arithmetic is being performed on ASCII strings of length eight or less. However, when chained arithmetic is being performed because (at least one of) the operands has a length greater than eight digits, special action must be taken if the result of the high order subtraction using ASCII.SUB is negative. When this result occurs, one of two actions must be taken. The first possible action is to perform an ASCII.SUBC on all of the low order segments of the result, subtracting each segment from 0. This will convert the 9's complement values into true magnitude values. Then in order to take in account a possible carry out from this conversion, use ASCII.SUB to subtract 0 from the high order segment of the result. The second possible action is to just reperform the subtraction, reversing the subtrahend and the minuend.

Obviously, the same number of subtractions are performed in both cases, so the choice of method depends on minimizing the number of supporting operations which are required, such as reloading operands.

Instruction Specific Exceptions: none

9 LOGICAL INSTRUCTIONS

All logical operations are performed on 64-bit unsigned quantities. Immediate operands are SIGN extended, and all shift distances are unsigned integers modulo 64.

Logical Instructions

AND	Boolean AND
OR	Logical Or
XOR	Logical Exclusive Or
NOT	Logical Not
SET.BIT	
CLEAR.BIT	
TOGGLE.BIT	
FIND.FIRST	Find First Logical One
ROL	Logical Rotate Left
ROR	Logical Rotate Right

Logical Shift Instructions

SLL	Shift Left Logical
SLL1	Shift Left Logical by 1
SLL2	Shift Left Logical by 2
SLL3	Shift Left Logical by 3
SRL	Shift Right Logical

9.1 FULLWORD LOGICAL OPERATIONS

With 2 operand addressing, operand 1 is replaced by operand 1 <op> 'Last Operand'. With 3 operand addressing, operand 1 is replaced by operand 2 <op> 'Last Operand'. The logical NOT operation replaces operand 1 with the 1's complement of 'Last Operand'.

9.1.1 AND Logical AND
9.1.2 OR Logical OR
9.1.3 NOT Logical NOT
9.1.4 XOR Logical Exclusive OR

Opcode				Implementation
AND	OR	NOT	XOR	
31C	31D	31E	31F	Rx <- Ry <op> Rz
41C	41D	41E	41F	Rx <- Rx <op> e{16}
51C	51D		51F	Rx <- Ry <op> =e{12}
61C	61D	61E	61F	Rx <- Ry <op> e{32}
71C	71D		71F	Rx <- Ry <op> =e{32}
B1C	B1D	B1E	B1F	Rx <- Ry <op> [Rz]
C1C	C1D	C1E	C1F	Rx <- Ry <op> [Ry][Rz]
D1C	D1D	D1E	D1F	Rx <- Ry <op> [Ry]e{12}
E1C	E1D	E1E	E1F	Rx <- Ry <op> [Ry][Rz]e{32}
F1C	F1D	F1E	F1F	Rx <- Ry <op> [Rz]e{32}

Instruction Specific Exceptions: none

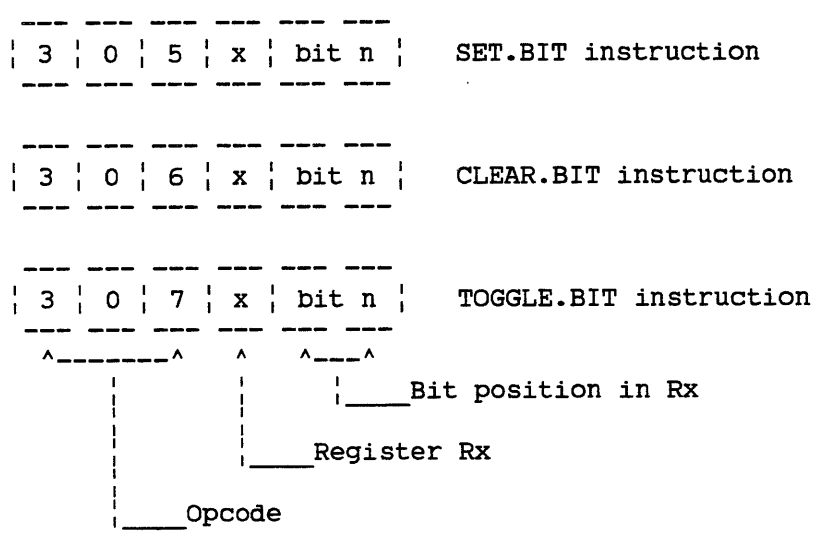
9.2 BIT-WISE LOGICAL OPERATIONS

The bit specified in 'Last Operand' is set for SET.BIT, cleared for CLEAR.BIT, and complemented for TOGGLE.BIT. 'Last operand', a literal encoded in the instruction, is used modulo 64.

9.2.1 SET.BIT
9.2.2 CLEAR.BIT
9.2.3 TOGGLE.BIT

Opcode assignment	Code	Implementation
SET.BIT	305	Rx <- Rx<SET>bit
CLEAR.BIT	306	Rx <- Rx<CLEAR>bit
TOGGLE.BIT	307	Rx <- Rx<COMPLEMENT>bit

The formats for the instructions are



Instruction Specific Exceptions: none

9.2.4 FIND.FIRST Find First Logical One

Starting at bit 0, scan a 64-bit string of 'Last Operand' for the first occurrence of a set bit. If found, the bit position is returned to Operand 1, otherwise a -1 is returned.

Opcode FIND.FIRST	Implementation
317	Rx <- first set bit position Rz
417	Rx <- first set bit position e{16}
617	Rx <- first set bit position e{32}
B17	Rx <- first set bit position [Rz]
C17	Rx <- first set bit position [Ry][Rz]
D17	Rx <- first set bit position [Ry]e{12}
E17	Rx <- first set bit position [Ry][Rz]e{32}
F17	Rx <- first set bit position [Rz]e{32}

Instruction Specific Exceptions: none

9.3 LOGICAL ROTATE OPERATIONS

Rotate an image of Operand 2 by the value in 'Last Operand' and place into operand 1. The unsigned value of Last Operand is used modulo 64.

9.3.1 ROL Logical Rotate Left
9.3.2 ROR Logical Rotate Right

Opcode		
ROL	ROR	Implementation
314	315	Rx <- Ry rotated Rz bits
514	515	Rx <- Ry rotated =e{12} bits
714	715	Rx <- Ry rotated =e{32} bits

Instruction specific exceptions: none

9.4 LOGICAL SHIFT OPERATIONS

Shift an image of Operand 2 by the count in 'Last Operand' and place into operand 1. The unsigned value of 'Last Operand' is used modulo 64. The fill bit is zero.

9.4.1 SLL Shift Left Logical
9.4.2 SLR Shift Right Logical

Opcode		
SLL	SRL	Implementation
310	311	Rx <- Ry shifted by Rz
510	511	Rx <- Ry shifted by =e{12}
710	711	Rx <- Ry shifted by =e{32}

SLL will not cause an overflow exception, whereas SLA -Shift Left Arithmetic- will. The sign bit is not maintained in either instruction.

Instruction Specific Exceptions: none

9.5 LEFT SHIFT OPERATIONS FOR FAST ARRAY INDEXING

SLL1, SLL2 and SLL3 perform left shifts by 1, 2, or 3 to facilitate indexing into 16-, 32-, and 64-bit word arrays. Operand 1 is replaced by the value of 'Last Operand' shifted left by 1, 2, or 3. The fill bit is zero.

9.5.1 SLL1 Shift Left Logical by 1
9.5.2 SLL2 Shift Left Logical by 2
9.5.3 SLL3 Shift Left Logical by 3

Opcode			Implementation	
SLL1.16	SLL1.32	SLL1.64		
394	3A4	3B4	Rx <- Ry	shift left by 1
494	4A4	4B4	Rx <- e{16}	shift left by 1
694	6A4	6B4	Rx <- e{32}	shift left by 1
B94	BA4	BB4	Rx <- [Rz]	shift left by 1
C94	CA4	CB4	Rx <- [Ry][Rz]	shift left by 1
D94	DA4	DB4	Rx <- [Ry]e{12}	shift left by 1
E94	EA4	EB4	Rx <- [Ry][Rz]e{32}	shift left by 1
F94	FA4	FB4	Rx <- [Rz]e{32}	shift left by 1

Opcode			Implementation	
SLL2.16	SLL2.32	SLL2.64		
		0Dx	Rx <- Rw	shift left by 2
395	3A5	3B5	Rx <- Rz	shift left by 2
495	4A5	4B5	Rx <- e{16}	shift left by 2
695	6A5	6B5	Rx <- e{32}	shift left by 2
	8Dx		Rx <- [SP]e{6}	shift left by 2
B95	BA5	BB5	Rx <- [Rz]	shift left by 2
C95	CA5	CB5	Rx <- [Ry][Rz]	shift left by 2
D95	DA5	DB5	Rx <- [Ry]e{12}	shift left by 2
E95	EA5	EB5	Rx <- [Ry][Rz]e{32}	shift left by 2
F95	FA5	FB5	Rx <- [Rz]e{32}	shift left by 2

Opcode			Implementation	
SLL3.16	SLL3.32	SLL3.64		
		0Ex	Rx <- Rw	shift left by 3
396	3A6	3B6	Rx <- Rz	shift left by 3
496	4A6	4B6	Rx <- e{16}	shift left by 3
696	6A6	6B6	Rx <- e{32}	shift left by 3
	8Ex		Rx <- [SP]e{6}	shift left by 3
B96	BA6	BB6	Rx <- [Rz]	shift left by 3
C96	CA6	CB6	Rx <- [Ry][Rz]	shift left by 3
D96	DA6	DB6	Rx <- [Ry]e{12}	shift left by 3
E96	EA6	EB6	Rx <- [Ry][Rz]e{32}	shift left by 3
F96	FA6	FB6	Rx <- [Rz]e{32}	shift left by 3

Use these instructions on the index of the object to be addressed. Load another register with the base address of the array, then use the BASE,INDEX addressing mode. Note the functional equivalence of using SLL1.64 to double the index and using ADD.64 to sum the index to itself.

Instruction Specific Exceptions: none

10 RELATIONAL TEST INSTRUCTIONS

The Relational Test instructions test two operands for a specified relation. If the relation is true, then the instruction performs the operation. If the relation is false, then no action is taken.

Integer Compare Instructions

CMP.BR	Compare Signed Integer and Branch Program Counter Relative
CMPU.BR	Compare Unsigned Integer and Branch Program Counter Relative
CMP	Compare Signed Integer and either Set Register or Generate Exception
CMPU	Compare Unsigned Integer and either Set Register or Generate Exception

Floating Point Compare Instructions

FCMP	Compare Floating Point and either Set Register or Generate Exception
FCMPX	Compare Floating Point Unordered Relation excepted and either Set Register or Generate Exception
FCMP.BR	Compare Floating Point and Branch Program Counter Relative
FCMPX.BR	Compare Floating Point Unordered Relation excepted and Branch Program Counter Relative

Byte String Compare Instructions

CMPB.BR	Compare Byte Strings and Branch Program Counter Relative
CMPB.BR.CONST	Compare Byte String Against Constant and Branch Program Counter Relative
CMPB.TEST	Compare Byte Strings and Generate Test Result

The order of comparison is:

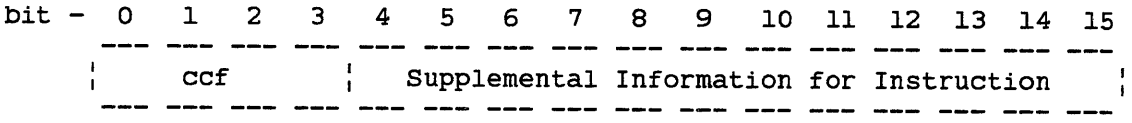
Operand 1 <relation> Operand 2

All integer type compares are performed on 64-bit quantities. The integer compare instructions that use signed integer operands first sign extend all operands to 64 bits before the compare. The unsigned integer compare instructions zero extend all operands before the

compare. A test performed with the unsigned integer compare will find a quantity with the sign bit on greater than a quantity with the sign bit off. Otherwise, these two types of instructions perform identically.

The instructions in this section use a 16-bit appendage. The first four high order bits of the appendage are for the compare condition field (ccf). The remaining 12 bits are used to specify the appropriate action for the instruction should the test be successful. The appendage in its entirety is shown below:

General Appendage Format



The compare condition field uses four bits to specify the relation to be tested. Refer to Table 10-1. The unordered bit is only used for the floating point compares to allow the detection of NaN's. A further discussion on unordered relations may be found in Chapter 3, "Data Representations", and within the descriptions of the floating point compare instructions. For the integer and string compare instructions, the unordered bit (bit 0) is reserved.

The 4-bit compare condition segment has the following format:

Compare Condition Field Format

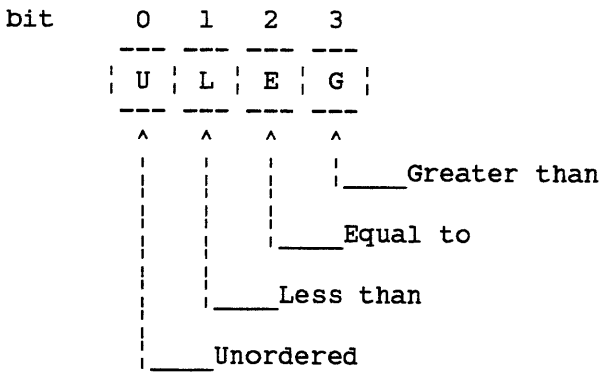


Table 10-1. Table of Compare Condition Field Relations

Code	Floating Point	Integer or String
1	G	G
2	E	E
3	EG	EG
4	L	L
5	LG	LG
6	LE	LE
7	LEG	LEG
8	U	
9	UG	G
10	UE	E
11	UEG	EG
12	UL	L
13	ULG	LG
14	ULE	LE
15	ULEG	LEG

The compare and branch instructions use the remaining 12-bit segment of the appendage to hold a two's complement PC relative offset for the branch.

-COMPARE AND BRANCH APPENDAGE FORMAT-

<two's complement PC relative offset>											
bit 4	5	6	7	8	9	10	11	12	13	14	15

For the compare and set register instructions, the 12-bit segment has the following format.

-COMPARE AND SET REGISTER APPENDAGE FORMAT-

0	0	0	0	0	0	0	0	0	0	0	0
bit 4	5	6	7	8	9	10	11	12	13	14	15

Bit 4 differentiates this instruction from the compare and generate exception form. Bits 5 through 15 are unused and must be '0'. Bits 0-62 of Rx are cleared, then the least significant bit of Rx is set if the compare is true, and is cleared if the compare is false.

For the compare and generate exception instructions, the 12-bit segment has the following format.

-COMPARE AND GENERATE EXCEPTION APPENDAGE FORMAT-

bit	4	5	6	7	8	9	10	11	12	13	14	15

Bit 4 is '1' to identify this format, and bits 5 through 7 are unused and must be '0'. Bits 8 through 15 contain a value which is included in the message to the exception handler.

All Branch Relative instructions use the address of the first byte of the branch instruction as the base for the offset.

10.1 INTEGER COMPARE

The Integer Compare instructions perform a relational test on signed or unsigned integer operands.

10.1.1 Compare Integer and Branch Program Counter Relative

Compare integer operands. If true, branch program counter relative by a 12-bit signed integer byte offset specified in the appendage. The relational test is specified in the compare condition segment.

10.1.1.1 CMP.BR Compare Signed Integers and Branch PC Relative
10.1.1.2 CMPU.BR Compare Unsigned Integers and Branch PC Relative

Opcode					
CMP.BR.8	.16	.32	.64	Implementation	
			033	Rx <rel> Rw	branch
			333	Ry <rel> Rz	branch
430	431	432	433	Rx <rel> e{16}	branch
			533	Ry <rel> =e{12}	branch
630	631	632	633	Ry <rel> e{32}	branch
			733	Ry <rel> =e{32}	branch
		A32		Rw <rel> [Ry][Rz]	branch
B30	B31	B32	B33	Ry <rel> [Rz]	branch
C30	C31	C32	C33	Rx <rel> [Ry]e{12}	branch
D30	D31	D32	D33	Rx <rel> [Ry][Rz]e{32}	branch
E30	E31	E32	E33	Rx <rel> [Ry][Rz]e{32}	branch
F30	F31	F32	F33	Ry <rel> [Rz]e{32}	branch

Opcode					
CMPU.BR.8	.16	.32	.64	Implementation	
			337	Ry <rel> Rz	branch
434	435	436	437	Rx <rel> e{16}	branch
			537	Ry <rel> =e{12}	branch
634	635	636	637	Ry <rel> e{32}	branch
			737	Ry <rel> =e{32}	branch
B34	B35	B36	B37	Ry <rel> [Rz]	branch
C34	C35	C36	C37	Rx <rel> [Ry]e{12}	branch
D34	D35	D36	D37	Rx <rel> [Ry][Rz]e{32}	branch
E34	E35	E36	E37	Rx <rel> [Ry][Rz]e{32}	branch
F34	F35	F36	F37	Ry <rel> [Rz]e{32}	branch

Instruction Specific Exceptions: none

10.1.2 Compare Integers and either Set Register or Generate Exception

Compare integer operands. If true, Set Register or Generate Exception. The "Set Register" occurs when bit 4 of the appendage is '0'. If the compare is true, set Rx equal to one; if the compare is false, clear Rx. The "Generate Exception" occurs when bit 4 of the appendage is '1'. If the compare is true, then generate the exception specified in the appendage. The relational test considers the operands to be signed integers for CMP, and unsigned integers for CMPU.

10.1.2.1 CMP
10.1.2.2 CMPU

Opcode				Implementation
CMP.8	CMP.16	CMP.32	CMP.64	
			323	Rx, exc; <- Ry <rel> Rz
420	421	422	423	Rx, exc; <- Rx <rel> e{16}
			523	Rx, exc; <- Ry <rel> =e{12}
620	621	622	623	Rx, exc; <- Ry <rel> e{32}
			723	Rx, exc; <- Ry <rel> =e{32}
B20	B21	B22	B23	Rx, exc; <- Ry <rel> [Rz]
C20	C21	C22	C23	Rx, exc; <- Rx <rel> [Ry][Rz]
D20	D21	D22	D23	Rx, exc; <- Rx <rel> [Ry]e{12}
E20	E21	E22	E23	Rx, exc; <- Rx <rel> [Ry][Rz]e{32}
F20	F21	F22	F23	Rx, exc; <- Ry <rel> [Rz]e{32}

Opcode				Implementation
CMPU.8	CMPU.16	CMPU.32	CMPU.64	
			327	Rx, exc; <- Ry <rel> Rz
424	425	426	427	Rx, exc; <- Rx <rel> e{16}
			527	Rx, exc; <- Ry <rel> =e{12}
624	625	626	627	Rx, exc; <- Ry <rel> e{32}
			727	Rx, exc; <- Ry <rel> =e{32}
B24	B25	B26	B27	Rx, exc; <- Ry <rel> [Rz]
C24	C25	C26	C27	Rx, exc; <- Rx <rel> [Ry][Rz]
D24	D25	D26	D27	Rx, exc; <- Rx <rel> [Ry]e{12}
E24	E25	E26	E27	Rx, exc; <- Rx <rel> [Ry][Rz]e{32}
F24	F25	F26	F27	Rx, exc; <- Ry <rel> [Rz]e{32}

Instruction Specific Exceptions: none

10.2 FLOATING POINT COMPARE

All floating point compare instructions first test for unordered relations. The FCMP and FCMP.BR instructions generate the INVALID OPERATION exception only if one or both operands are signaling NaN's. The FCMPX and FCMPX.BR instructions generate an INVALID OPERATION exception if one or both operands are quiet or signaling NaN's. A NaN is unordered to everything, including itself.

The hardware exceptions are always generated before any software exceptions specified in the appendage. The relations that may be tested in the compare condition field are as follows:

- UNORDERED
- LESS THAN
- EQUAL TO
- GREATER THAN

The following table specifies the outcome for an unordered relation with the FCMP, FCMP.BR, FCMPX, and FCMPX.BR instructions. The table headers have the following meaning:

E Exception handler enabled for INVALID OPERATION (bit 26 in PSW).
U Unordered bit state in the compare condition field.

E	U		ACTION
0	0	Set bit 27 in PSW.	Terminate this instruction and execute next instruction.
0	1	Set bit 27 in PSW.	Test the operands and take the action specified in the appendage.
1	x	Set bit 27 in PSW.	Take the INVALID OPERATION exception handler.

10.2.1 Compare Floating Point and Branch PC Relative

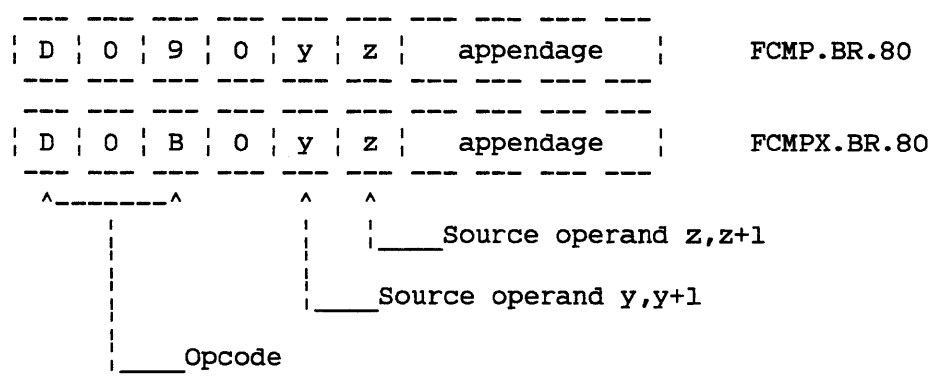
Compare the 32-, 64-, or 80-bit floating point operands. If true, branch PC relative with the 12-bit signed integer offset specified in the appendage.

10.2.1.1 FCMP.BR Compare Floating Point and Branch PC Relative
10.2.1.2 FCMPX.BR Compare Floating Point and Branch PC Relative, Unordered Relation Excepted

Opcode				Implementation	
FCMP.BR.32	.64	FCMPX.BR.32	.64		
13x	12x			Rx <rel> Rw	branch
338	339	33A	33B	Rx <rel> Rz	branch
438	439	43A	43B	Rx <rel> e{16}	branch
538		53A		Ry <rel> =e{12}	branch
638	639	63A	63B	Ry <rel> e{32}	branch
738		73A		Ry <rel> =e{32}	branch
93x				Rw <rel> [SP]e{6}	branch
B38	B39	B3A	B3B	Ry <rel> [Rz]	branch
C38	C39	C3A	C3B	Rx <rel> [Ry]e{12}	branch
D38	D39	D3A	D3B	Rx <rel> [Ry][Rz]e{32}	branch
E38	E39	E3A	E3B	Rx <rel> [Ry][Rz]e{32}	branch
F38	F39	F3A	F3B	Ry <rel> [Rz]e{32}	branch

Opcode assignment	Code	Implementation
FCMP.BR.80	D09	Ry,Ry+1 <rel> Rz,Rz+1, branch
FCMPX.BR.80	D0B	Ry,Ry+1 <rel> Rz,Rz+1, branch

Formats for the non-generalized instructions are



Instruction Specific Exceptions: FLOATING POINT INVALID OPERATION

Table 10-2. Result Matrix for FCMP, FCMPX, FCMP.BR, FCMPX.BR

If OP1 is	and OP2 is	then result is
Zero	Zero	Zero with sign
Zero	Denorm, Normal	OP2
Zero	Infinity	OP2
Denorm, Normal	Zero	OP1
Denorm, Normal	Denorm, Normal	computed
Denorm, Normal	Infinity	OP2
Infinity	Zero	OP1
Infinity	Denorm, Normal	OP1
Infinity	Infinity	OP1 or Invalid

10.2.2 Compare Floating Point and either Set Register or Generate Exception

Compare the 32-, 64-, or 80-bit floating point operands and perform the action specified by bit 4 in the appendage. If bit 4 = '0', then "Set Register" (set Rx equal to '1' if the compare is true, and equal to '0' if false). If bit 4 = '1' and the compare is true then generate the exception specified in the appendage.

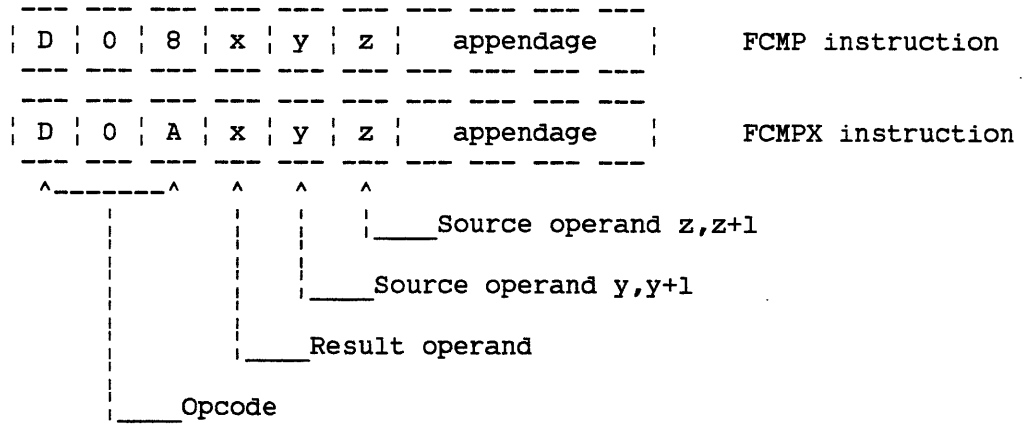
10.2.2.1 FCMP Compare Floating Point and either Set Register or Generate Exception

10.2.2.2 FCMPX Compare Floating Point and either Set Register or Generate Exception, Unordered Relation Excepted

Opcode				Implementation
FCMP.32	.64	FCMPX.32	.64	
328	329	32A	32B	Rx, exc; <- Ry <rel> Rz
428	429	42A	42B	Rx, exc; <- Rx <rel> e{16}
628	629	62A	62B	Rx, exc; <- Ry <rel> e{32}
728		72A		Rx, exc; <- Ry <rel> =e{32}
B28	B29	B2A	B2B	Rx, exc; <- Ry <rel> [Rz]
C28	C29	C2A	C2B	Rx, exc; <- Rx <rel> [Ry][Rz]
D28	D29	D2A	D2B	Rx, exc; <- Rx <rel> [Ry]e{12}
E28	E29	E2A	E2B	Rx, exc; <- Rx <rel> [Ry][Rz]e{32}
F28	F29	F2A	F2B	Rx, exc; <- Ry <rel> [Rz]e{32}

Opcode assignment	Code	Implementation
FCMP.80	D08	Rx, exc; <- Ry,Ry+1 <rel> Rz,Rz+1
FCMPX.80	D0A	Rx, exc; <- Ry,Ry+1 <rel> Rz,Rz+1

The formats for the non-generalized instructions are



Instruction Specific Exceptions: FLOATING POINT INVALID OPERATION

10.3 BYTE STRING COMPARE

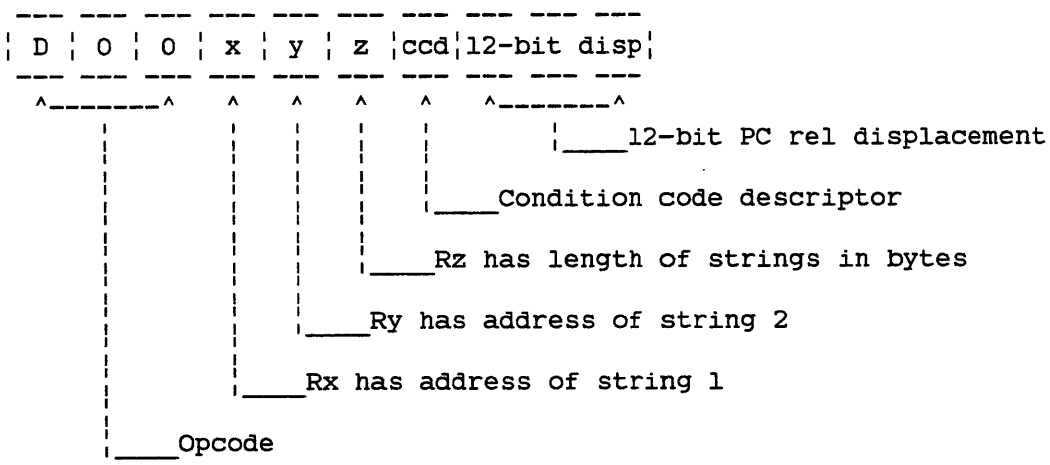
The string operand is tested against another string or a constant.

10.3.1 CMPB.BR Compare Byte Strings and Branch

Compare two byte strings of equal length. The comparison proceeds byte-wise from the respective starting addresses. Rx contains the address of string 1, Ry the address of string 2, and Rz contains the length of the strings in bytes. If string 1 <rel> string 2 is true, then the relative branch is taken. This instruction has an appendage identical in format to the integer CMP.BR instructions.

Opcode assignment	Code	Implementation
CMPB.BR	D00	[Rx:<st>] <rel> [Ry:<st>], Rz:<len>, br

The format for the non-generalized CMPB.BR instruction is



Special Notes:

The contents of Rx, Ry and Rz are unpredictable at the end of the instruction. If defined values for these registers are desired, use a software loop instead of these instructions.

Instruction Specific Exceptions: none

10.3.2 CMPB.BR.CONST Compare Byte String Against Constant and Branch

Compare, word-wise, a byte string against a 64-bit word constant. Rx contains the address of the string, Ry contains a 64-bit word to be used in the compare, and Rz contains the length of the string in bytes. If string 1 <rel> constant is true, then the relative branch is taken. The final compare may be less than a word, i.e., the string need not be a multiple of eight bytes in length. Thus, this instruction appears to truncate on byte boundaries, if necessary, the word constant on the final compare to conform to the length of the string.

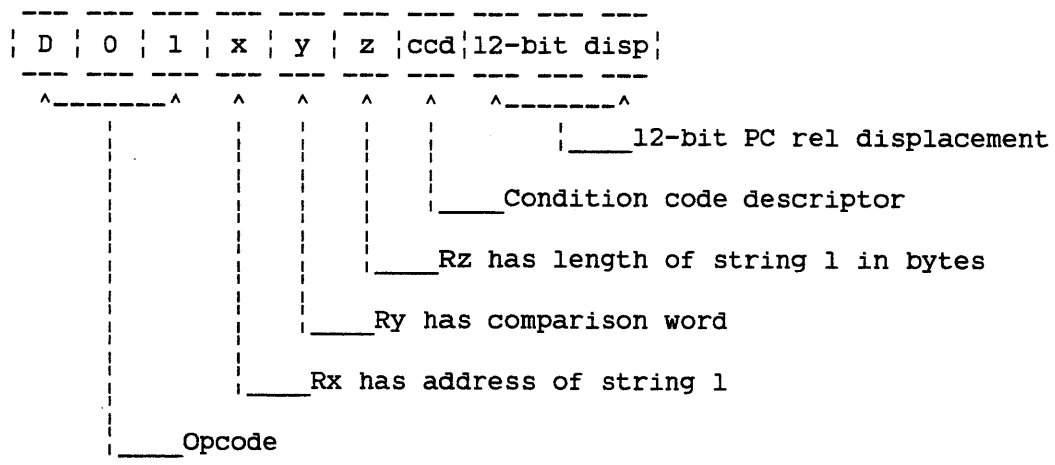
This instruction has an appendage identical in format to the integer CMP.BR instructions.

Opcode assignment	Code	Implementation
CMPB.BR.CONST	D01	[Rx:<st>] <rel> Ry, Rz:<len>, branch

Special Notes:

The contents of Rx and Rz are unpredictable at the end of the instruction. If defined values for these registers are desired, use a software loop instead of these instructions.

The format for the non-generalized CMPB.BR.CONST instruction is



Instruction Specific Exceptions: none

10.3.3 CMPB.TEST Compare Byte Strings and Generate Test Result

Compare, byte-wise, two strings of equal length. Rx contains the starting address of string 1, Ry the starting address of string 2, and Rz contains the length of the strings in bytes. The first inequality of the byte-wise compare, or, if equal, the exhaustion of the string, will set Rx to one of three values:

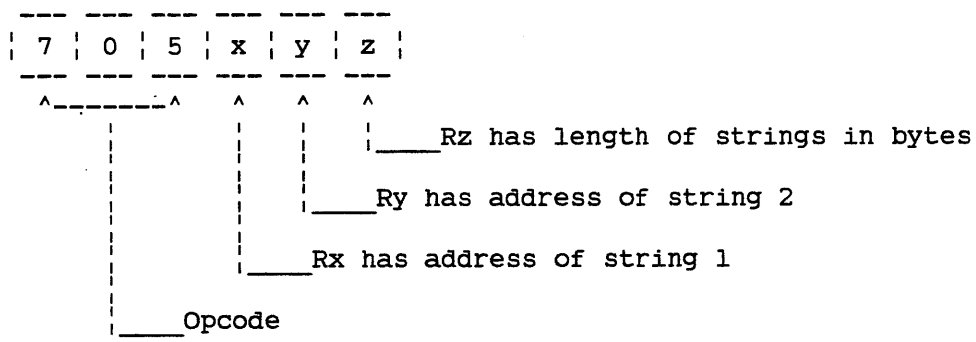
```
If [Rx]:<st>   less than    [Ry]:<st> then set Rx to -1
If [Rx]:<st>    equal to    [Ry]:<st> then set Rx to  0
If [Rx]:<st>  greater than  [Ry]:<st> then set Rx to  1
```

Opcode assignment	Code	Implementation
CMPB.TEST	705	[Rx]:<st> <rel> [Ry]:<st>, Rz:<len>

Special Notes:

The contents of Ry and Rz are unpredictable at the end of the instruction. If defined values for these registers are desired, use a software loop instead of these instructions.

The format for the non-generalized CMPB.TEST instruction is



Instruction Specific Exceptions: none

11 DATA CONVERSION INSTRUCTIONS

These instructions allow conversion across data types, including Double precision, Extended Double precision, Single precision, 64-bit signed Integer and ASCII numeric strings.

CVT.AI	Convert ASCII String to Integer
CVT.DE	Convert from Double to Extended
CVT.DI	Convert from Double to Integer
CVT.DS	Convert from Double to Single
CVT.ED	Convert from Extended to Double
CVT.EI	Convert from Extended to Integer
CVT.ES	Convert from Extended to Single
CVT.IA	Convert from Integer to ASCII
CVT.ID	Convert from Integer to Double
CVT.IE	Convert from Integer to Extended
CVT.IS	Convert from Integer to Single
CVT.SD	Convert from Single to Double
CVT.SE	Convert from Single to Extended
CVT.SI	Convert from Single to Integer
FINP.32	Floating Point Integer Part
FINP.64	Floating Point Integer Part
FINP.80	Floating Point Integer Part

Given the numerical attributes in the operand, the table below shows the expected result of a conversion for each instruction.

Table 11-1. Data Conversion Result Matrix for Operand States

Opcode	Zero	Denorm	Normal	Infinity	QNaN	SNaN
CVT.DE	2	4	3	5	8	6
CVT.DI	1	1	7	6	6	6
CVT.DS	2	9	3	5	10	6
CVT.ED	2	9	3	5	10	6
CVT.EI	1	1	7	6	6	6
CVT.ES	2	9	3	5	10	6
CVT.ID	12	12	12	12	12	12
CVT.IE	13	13	13	13	13	13
CVT.IS	12	12	12	12	12	12
CVT.SD	2	4	3	5	8	6
CVT.SE	2	4	3	5	8	6
CVT.SI	1	1	7	6	6	6
FINP.32	2	11	11	6	6	6
FINP.64	2	11	11	6	6	6
FINP.80	2	11	11	6	6	6

CONVERSION RESULT:

[1] Zero
[2] Zero with same sign
[3] Equivalent number
[4] Equivalent normalized number
[5] Infinity with same sign
[6] Floating Point Invalid Operation
[7] Truncated number
[8] Same Quiet NaN with zeroes appended to significand
[9] Floating Point Underflow
[10] Same Quiet NaN with least significant bits truncated
[11] Rounded to Floating Point Integer
[12] Rounded result
[13] Exact result

Using, for an example, the instruction CVT.DE (Convert Double to Extended), assume

If operand to be converted is a denormalized number, then the conversion result in the destination operand will be an equivalent normalized number. This is conversion result (4) above.

If operand to be converted is a signaling NaN, then no conversion takes place; rather, a FLOATING POINT INVALID OPERATION exception is generated. This is conversion result (6) above.

Note that CVT floating to Integer instructions truncate, while CVT floating to floating instructions obey the rounding mode. However, the FINP instructions may be used to round floating point numbers to Integer in the same floating point format. The instruction sequence

```
FINP.32
CVT.SI
```

will round a floating point number as it is converted to Integer.

11.1 ASCII CONVERSION INSTRUCTIONS

These instructions perform conversions on 64-bit signed Integers and numeric strings. A numeric string is a sequence of ASCII encoded numbers within the set ['30'..'39'] hex.

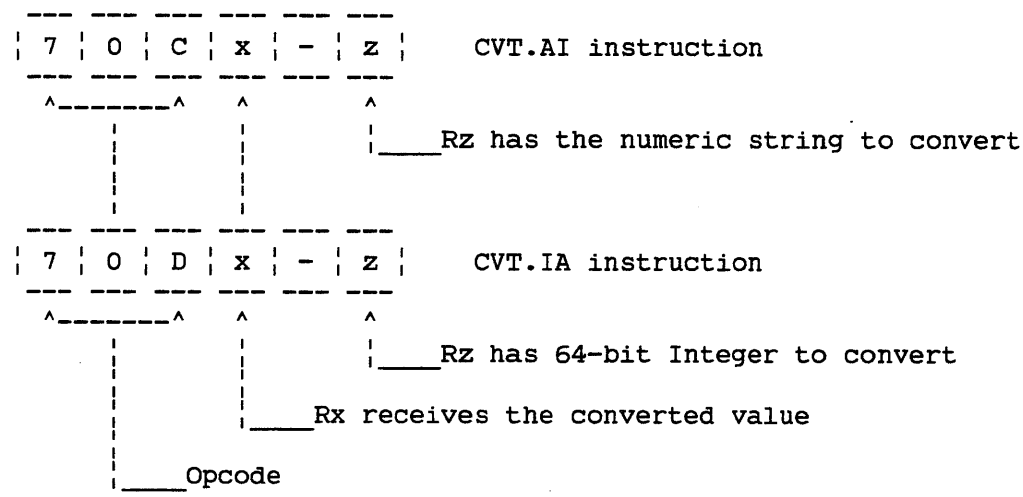
CVT.AI replaces Rx with the conversion of Rz from an eight digit numeric string to a 64-bit signed Integer. Results are unpredictable if a byte in the numeric string is not hex '00' or hex '30' to '39'.

CVT.IA replaces Rx with the conversion of Rz from a 64-bit signed Integer to an eight digit numeric string. Results are unpredictable if the operand is outside the range 0 through 99,999,999.

11.1.1 CVT.IA Convert from Integer to ASCII
11.1.2 CVT.AI Convert from ASCII to Integer

Opcode assignment	Code	Implementation
CVT.AI	70C	Rx <- Rz converted
CVT.IA	70D	Rx <- Rz converted

The formats of these non-generalized instructions are



Instruction Specific Exceptions: none

11.2 CONVERT FROM DOUBLE TO EXTENDED, INTEGER, OR SINGLE

Convert 'Last Operand' from Double to Double Extended (CVT.DE), 64-bit Integer (CVT.DI), or Floating Point Single (CVT.DS). Place the result into Operand 1.

11.2.1 CVT.DE Convert from Double to Extended
11.2.2 CVT.DI Convert from Double to Integer
11.2.3 CVT.DS Convert from Double to Single

Opcode			Implementation	
CVT.DE	CVT.DI	CVT.DS		
3D7	3D3	3D6	Rx(,Rx+1) <- Rz	converted
4D7	4D3	4D6	Rx(,Rx+1) <- e{16}	converted
6D7	6D3	6D6	Rx(,Rx+1) <- e{32}	converted
BD7	BD3	BD6	Rx(,Rx+1) <- [Rz]	converted
CD7	CD3	CD6	Rx(,Rx+1) <- [Ry]e{12}	converted
DD7	DD3	DD6	Rx(,Rx+1) <- [Ry][Rz]e{32}	converted
ED7	ED3	ED6	Rx(,Rx+1) <- [Ry][Rz]e{32}	converted
FD7	FD3	FD6	Rx(,Rx+1) <- [Rz]e{32}	converted

Instruction Specific Exceptions:

EXCEPTION	CVT.DE	CVT.DI	CVT.DS
FP Underflow			X
FP Overflow			X
FP Invalid Operation	X	X	X
FP Inexact Result			X
Integer Overflow		X	

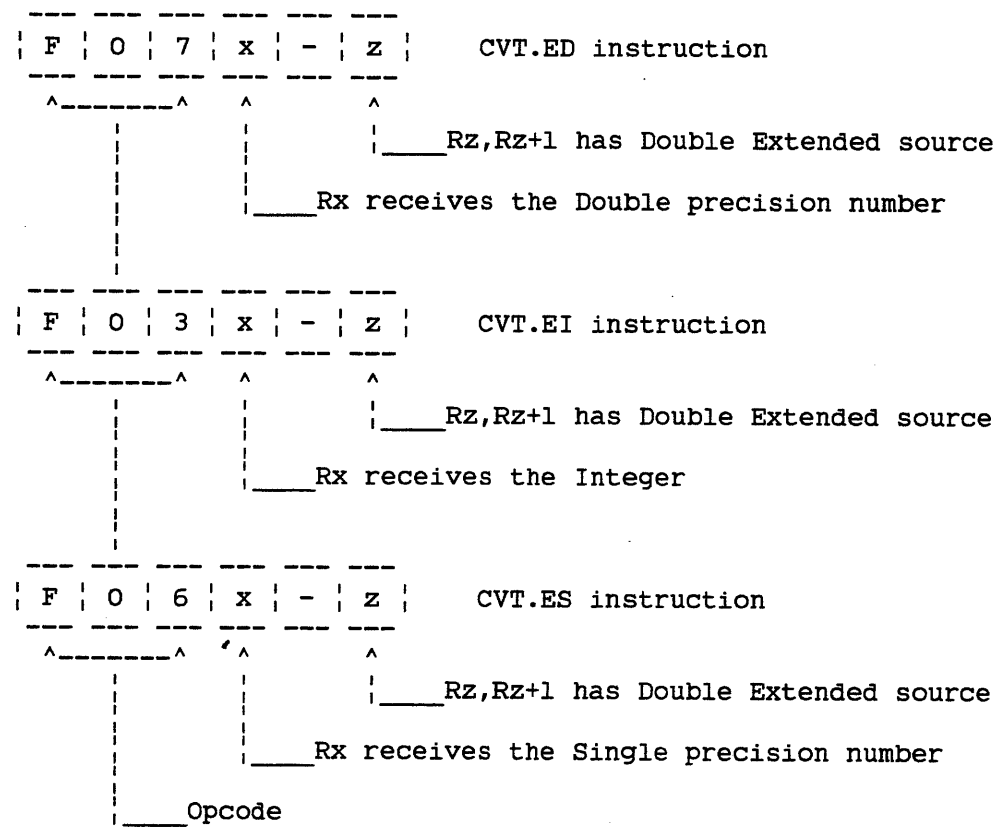
11.3 CONVERT FROM EXTENDED TO DOUBLE, INTEGER, OR SINGLE

Convert 'Last Operand' from Extended to Double (CVT.ED), 64-bit Integer (CVT.EI), or Floating Point Single (CVT.ES). Place the result into Operand 1.

11.3.1 CVT.ED Convert from Extended to Double
11.3.2 CVT.EI Convert from Extended to Integer
11.3.3 CVT.ES Convert from Extended to Single

Opcode assignment	Code	Implementation
CVT.ED	F07	Rx <- Rz,Rz+1 converted
CVT.EI	F03	Rx <- Rz,Rz+1 converted
CVT.ES	F06	Rx <- Rz,Rz+1 converted

The formats of these non-generalized instructions are



Instruction Specific Exceptions:

EXCEPTION	CVT.ED	CVT.EI	CVT.ES
FP Underflow	X		X
FP Overflow	X		X
FP Invalid Operation	X	X	X
FP Inexact Result	X		X
Integer Overflow		X	

11.4 CONVERT FROM INTEGER TO DOUBLE, SINGLE, OR EXTENDED

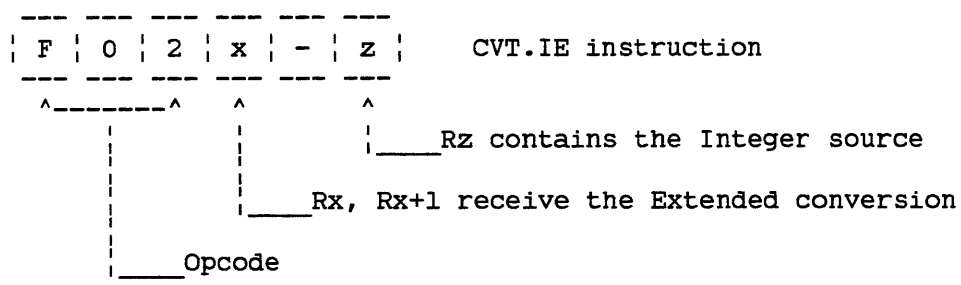
Convert 'Last Operand' from 64-bit signed Integer to Double (CVT.ID), Floating Point Single (CVT.IS), or Extended (CVT.IE). Place the result into Operand 1.

11.4.1 CVT.ID Convert from Integer to Double
11.4.2 CVT.IS Convert from Integer to Single
11.4.3 CVT.IE Convert from Integer to Extended

Opcode		Implementation	
CVT.ID	CVT.IS		
3D2	3C2	Rx <- Rz	converted
4D2	4C2	Rx <- e{16}	converted
6D2	6C2	Rx <- e{32}	converted
BD2	BC2	Rx <- [Rz]	converted
CD2	CC2	Rx <- [Ry][Rz]	converted
DD2	DC2	Rx <- [Ry]e{12}	converted
ED2	EC2	Rx <- [Ry][Rz]e{32}	converted
FD2	FC2	Rx <- [Rz]e{32}	converted

Opcode assignment	Code	Implementation
CVT.IE	F02	Rx,Rx+1 <- Rz converted

The format of the non-generalized CVT.IE instruction is



Instruction Specific Exceptions:

EXCEPTION	CVT.ID	CVT.IE	CVT.IS
FP Inexact Result	X		X

11.5 CONVERT FROM SINGLE TO DOUBLE, INTEGER, OR EXTENDED

Convert 'Last Operand' from Floating Point Single to Double (CVT.SD), 64-bit Integer (CVT.SI), or Extended (CVT.SE). Place the result into Operand 1.

11.5.1 CVT.SD Convert from Single to Double
11.5.2 CVT.SI Convert from Single to Integer
11.5.3 CVT.SE Convert from Single to Extended

Opcode			Implementation	
CVT.SD	CVT.SI	CVT.SE		
3C6	3C3	3C7	(Rx,Rx+1) <- Rz	converted
4C6	4C3	4C7	(Rx,Rx+1) <- e{16}	converted
6C6	6C3	6C7	(Rx,Rx+1) <- e{32}	converted
BC6	BC3	BC7	(Rx,Rx+1) <- [Rz]	converted
CC6	CC3	CC7	(Rx,Rx+1) <- [Ry][Rz]	converted
DC6	DC3	DC7	(Rx,Rx+1) <- [Ry]e{12}	converted
EC6	EC3	EC7	(Rx,Rx+1) <- [Ry][Rz]e{32}	converted
FC6	FC3	FC7	(Rx,Rx+1) <- [Rz]e{32}	converted

Instruction Specific Exceptions:

EXCEPTION	CVT.SD	CVT.SE	CVT.SI
FP Invalid Operation	X	X	X
Integer Overflow			X

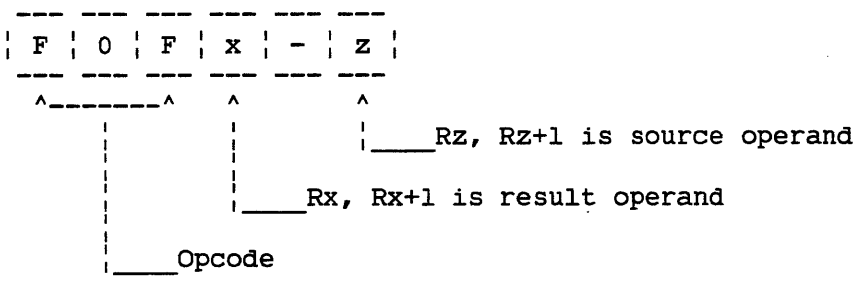
11.6 FINP FLOATING POINT INTEGER PART

With the two operand forms, Operand 2 is rounded at the binary point based on the rounding mode, and the resulting floating point Integer replaces Operand 1. With the three operand forms, Operand 3 is the source and Operand 2 is ignored.

Opcode		Implementation
FINP.32	FINP.64	
3CF	3DF	Rx <- Integer part of Rz
4CF	4DF	Rx <- Integer part of e{16}
6CF	6DF	Rx <- Integer part of e{32}
BCF	BDF	Rx <- Integer part of [Rz]
CCF	CDF	Rx <- Integer part of [Rz]
DCF	DDF	Rx <- Integer part of e{12}
ECF	EDF	Rx <- Integer part of [Rz]e{32}
FCF	FDF	Rx <- Integer part of [Rz]e{32}

Opcode assignment	Code	Implementation
FINP.80	F0F	(Rx,Rx+1) <- Integer part of (Rz,Rz+1)

The format of the non-generalized FINP.80 instruction is



Instruction Specific Exceptions: FLOATING POINT INVALID OPERATION
FLOATING POINT INEXACT RESULT

RESULT MATRIX FOR FINP

If OP1 is	then result is
Zero	Zero. The sign of the operand is preserved.
Denorm, Normal	Computed
Infinite	Invalid Operation

12 FLOW OF CONTROL INSTRUCTIONS

The control instructions may be divided into 4 basic groups: simple control transfer, conditional control transfer, procedural control transfer and control transfers requiring special handling.

In all control instructions (including the compare and branch instructions), any relative offsets refer to the first byte of the instruction.

Simple Control Transfer Instructions

BR.ABS	Branch Absolute
BR.REL	Branch Relative
BR.BACKWARD	Branch Backward Short Relative
BR.FORWARD	Branch Forward Short Relative

Conditional Control Transfer Instructions

BR.<cond>.ABS	Branch Register Conditional absolute
BR.<cond>.REL	Branch Register Conditional relative
BR.B <cond>.SH REL	Branch Backward Register conditional
BR.F <cond>.SH REL	Branch Forward Register conditional

Procedural Control Transfer Instructions

BR.REG	Branch through Register
CALL	Procedure Call Through Stack
CALL.REG	Procedure Call Through Register
EXIT	

Control Transfers Requiring Special Handling

BREAKPOINT	
EXCEPTION	
IXIT	Exit from Interrupt
BXIT	Exit from Break

12.1 UNCONDITIONAL BRANCHES

BR.ABS loads the program counter with the 32-bit displacement encoded in the instruction.

BR.REL adds the 32-bit signed integer offset encoded in the instruction to the program counter.

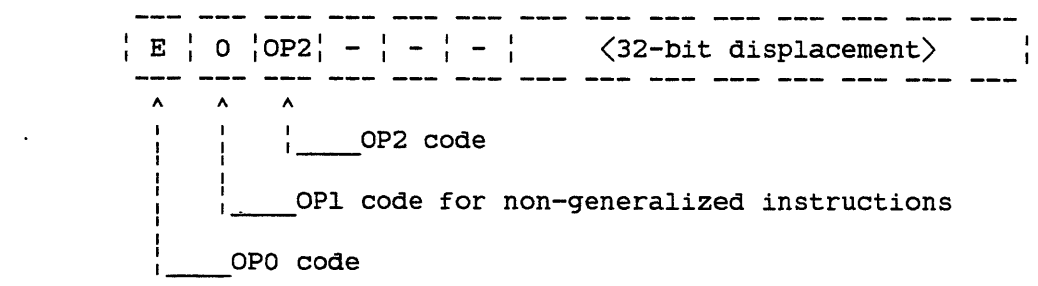
BR.BACKWARD subtracts the unsigned 8-bit displacement from the program counter. The branch range is from 0 to -255 bytes.

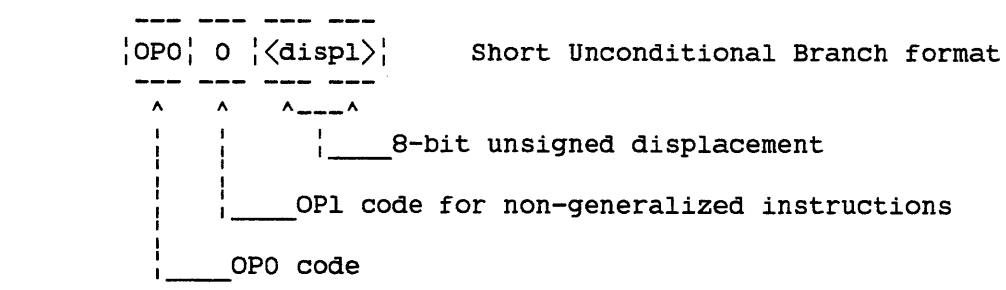
BR.FORWARD adds the unsigned 8-bit displacement to the program counter. The branch range is from 0 to 255 bytes.

12.1.1 BR.ABS Branch Absolute
12.1.2 BR.REL Branch Relative
12.1.3 BR.BACKWARD Branch Backward Short Relative
12.1.4 BR.FORWARD Branch Forward Short Relative

Opcode assignment	Code	Implementation
BR.ABS	E07	PC <- displacement
BR.REL	E0F	PC <- PC + displacement
BR.BACKWARD	A0x	PC <- PC - unsigned displacement
BR.FORWARD	20x	PC <- PC + unsigned displacement

The formats for these instructions are





Instruction Specific Exceptions: none

12.2 BRANCH REGISTER CONDITIONAL LONG INSTRUCTIONS

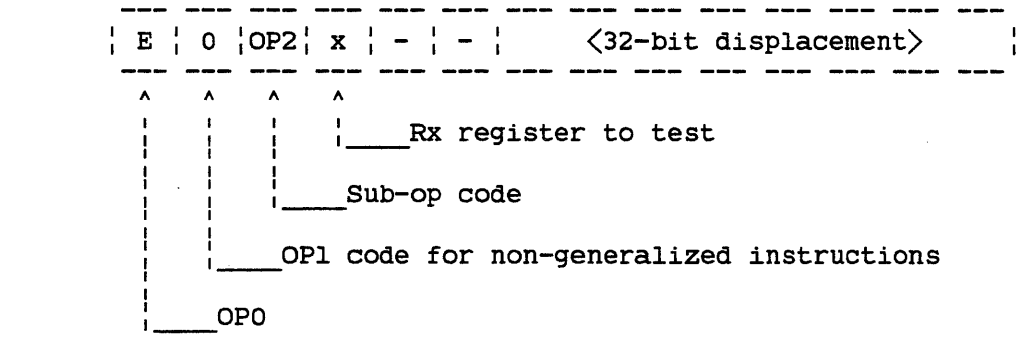
Load (BR.<COND>.ABS) or add (BR.<COND>.REL) the 32-bit signed integer encoded in the instruction to the program counter if the relation [(register) <cond> zero] is true.

12.2.1 BR.<cond>.ABS Branch Register Conditional Absolute
12.2.2 BR.<cond>.REL Branch Register Conditional Relative

Opcode assignment	Code	Implementation
BR.GT.ABS	E01	PC <- signed disp, Rx<GT>0
BR.EQ.ABS	E02	PC <- signed disp, Rx<EQ>0
BR.GE.ABS	E03	PC <- signed disp, Rx<GE>0
BR.LT.ABS	E04	PC <- signed disp, Rx<LT>0
BR.NE.ABS	E05	PC <- signed disp, Rx<NE>0
BR.LE.ABS	E06	PC <- signed disp, Rx<LE>0

Opcode assignment	Code	Implementation
BR.GT.REL	E09	PC <- PC+signed disp, Rx<GT>0
BR.EQ.REL	E0A	PC <- PC+signed disp, Rx<EQ>0
BR.GE.REL	E0B	PC <- PC+signed disp, Rx<GE>0
BR.LT.REL	E0C	PC <- PC+signed disp, Rx<LT>0
BR.NE.REL	E0D	PC <- PC+signed disp, Rx<NE>0
BR.LE.REL	E0E	PC <- PC+signed disp, Rx<LE>0

The instruction format for the long conditional branches is



Instruction Specific Exceptions: none

12.3 BRANCH REGISTER CONDITIONAL SHORT RELATIVE INSTRUCTIONS

Subtract (BR.B.<cond>.SH.REL) or add (BR.F.<cond>.SH.REL) the 8-bit unsigned integer encoded in the instruction to the program counter if the relation [(register) <cond> zero] is true.

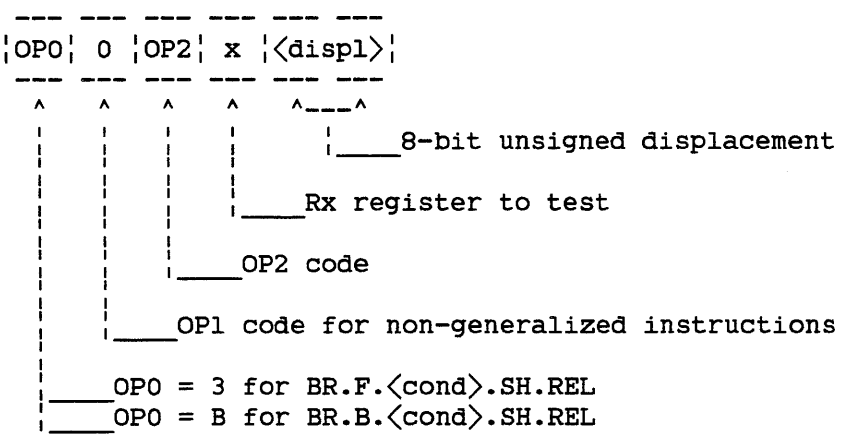
12.3.1 BR.B.<cond>.SH.REL Branch Backward Register Conditional Short Relative

12.3.2 BR.F.<cond>.SH.REL Branch Forward Register Conditional Short Relative

Opcode assignment	Code	Implementation
BR.B.GT.SH.REL	B09	PC <- PC-unsigned disp, Rx<GT>0
BR.B.EQ.SH.REL	B0A	PC <- PC-unsigned disp, Rx<EQ>0
BR.B.GE.SH.REL	B0B	PC <- PC-unsigned disp, Rx<GE>0
BR.B.LT.SH.REL	B0C	PC <- PC-unsigned disp, Rx<LT>0
BR.B.NE.SH.REL	B0D	PC <- PC-unsigned disp, Rx<NE>0
BR.B.LE.SH.REL	B0E	PC <- PC-unsigned disp, Rx<LE>0

Opcode assignment	Code	Implementation
BR.F.GT.SH.REL	309	PC <- PC+unsigned disp, Rx<GT>0
BR.F.EQ.SH.REL	30A	PC <- PC+unsigned disp, Rx<EQ>0
BR.F.GE.SH.REL	30B	PC <- PC+unsigned disp, Rx<GE>0
BR.F.LT.SH.REL	30C	PC <- PC+unsigned disp, Rx<LT>0
BR.F.NE.SH.REL	30D	PC <- PC+unsigned disp, Rx<NE>0
BR.F.LE.SH.REL	30E	PC <- PC+unsigned disp, Rx<LE>0

The instruction format for the short conditional branches is



Instruction Specific Exceptions: none

12.4 PROCEDURAL CONTROL TRANSFER INSTRUCTIONS

There are two types of procedure calls supported by the ELXSI architecture. Their difference lies in where the return Program Counter value is saved (the address of the instruction immediately following the call), and by the architectural support provided for local stack frame deallocation.

In the first and most commonly used type, the return PC value of the calling procedure is placed (rather than pushed) in the 32-bit memory location pointed at by the stack pointer (CALL). When returning from the called procedure, the local stack frame is deallocated automatically, restoring the return PC value (EXIT).

In the alternate procedure call mechanism, the return PC is passed in a register (CALL.REG). To return to the calling procedure, a branch through register is executed (BR.REG). No architectural stack frame deallocation support is provided for the CALL.REG and BR.REG instructions. Any local stack frame handling must be done by specially emitted code.

For a complete description of the procedure calling mechanism, the reader should refer to Section 2.3, "Procedure Calls".

Procedural Control Transfer Instructions

BR.REG	Branch through Register
CALL	Procedure Call Through Stack
CALL.REG	Procedure Call Through Register
EXIT	

12.4.1 BR.REG Branch through Register

Load the program counter from the specified register.

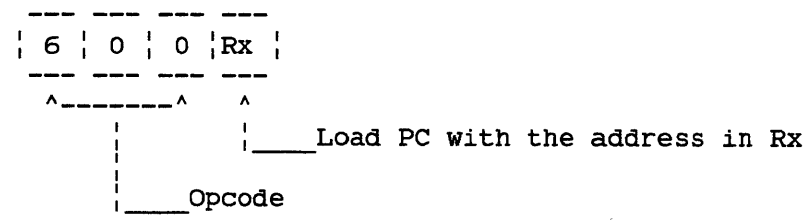
Opcode assignment	Code	Implementation
BR.REG	600	PC <- Rx

Instruction Specific Exceptions: none

Special Notes:

This instruction is used in conjunction with the CALL.REG instruction to return from a procedure call.

The format of the non-generalized BR.REG instruction is

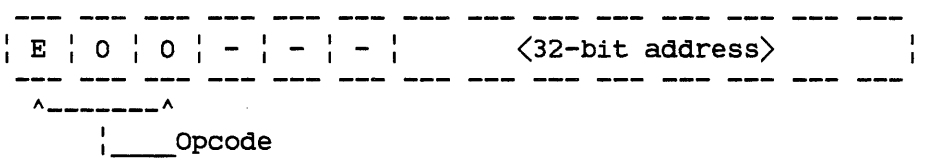


12.4.2 CALL Procedure Call Through Stack

The CALL instruction places the return PC (as an absolute address) into the 32-bit memory location pointed at by register 15 (the stack pointer) and branches absolute to the 32-bit signed integer address encoded in the instruction.

Opcode assignment	Code	Implementation
CALL	E00	[R15] <- PC+7, PC <- signed disp

The format of the non-generalized CALL instruction is



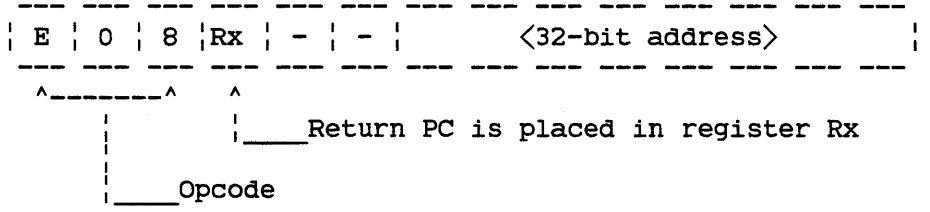
Instruction Specific Exceptions: none

12.4.3 CALL.REG Procedure Call Through Register

The CALL.REG instruction places the return PC value (as an absolute address) into the specified register and branches absolute to the 32-bit signed integer address encoded in the instruction.

Opcode assignment	Code	Implementation
CALL.REG	E08	Rx <- PC+7, PC <- signed disp

The format of the non-generalized CALL.REG instruction is



This instruction may also be used to place the current PC into a specified register. Execute a CALL.REG with the address of the next instruction as the destination of the call. This function may also be accomplished with the LD.64 instruction and the help of the binder.

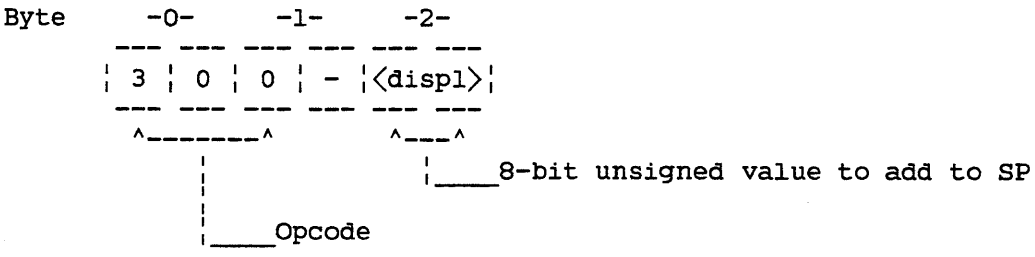
Instruction Specific Exceptions: none

12.4.4 EXIT

The EXIT instruction deallocates the local stack frame or activation record. It adds an unsigned 8-bit displacement to the stack pointer, then places the absolute address from the new top-of-stack into the PC. The displacement, encoded in the instruction, represents the distance from the current stack pointer to the return PC location in the stack.

Opcode assignment	Code	Implementation
EXIT	300	SP <- SP + udisp, PC <- [SP]

The format of the non-generalized EXIT instruction is



EXIT and CALL instructions both assume stack storage for the PC return values, and should be paired likewise. If using a CALL.REG instruction, return through a branch register instruction, and perform local stack frame handling by an add to the SP.

Slightly different code is usually emitted when the stack frame contains dynamic arrays or arrays larger than 2**8 bytes.

12.5 CONTROL TRANSFERS REQUIRING SPECIAL HANDLING

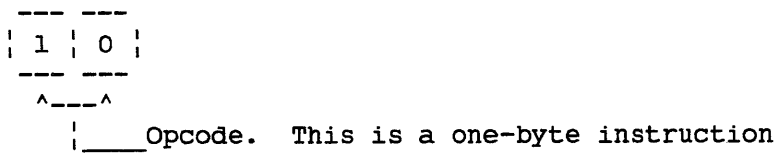
For a more complete description of exception handling, see Chapter 2, "Architecture".

12.5.1 BREAKPOINT

The BREAKPOINT instruction pushes the next instruction address and registers RF through R0 onto the stack, loads registers R0 through R4 with specific information, and loads the PC with a prespecified entry point into the system Debugger. See Section 2.4, "Interrupts and Breakpoints", for a complete description of the data placed in the registers.

Opcode assignment	Code	Implementation
BREAKPOINT	10	see above description

The format of the non-generalized BREAKPOINT instruction is



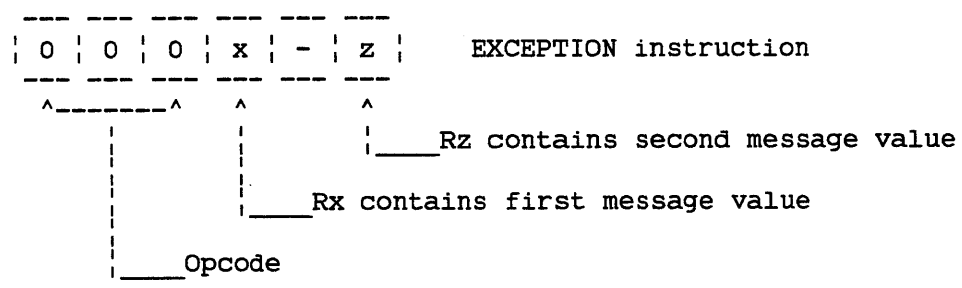
Instruction Specific Exceptions: none

12.5.2 EXCEPTION

Sends a Software Generated Exception class message which includes the concatenated 64-bit values of Rx and Rz. Rx forms the high order 64 bits and Rz forms the low order 64 bits.

Opcode assignment	Code	Implementation
EXCEPTION	000	see above description

The format of the non-generalized EXCEPTION instruction is



12.5.3 IXIT Exit from Interrupt

The IXIT instruction is used to exit from an interrupt routine. First, the local stack frame is deallocated by adding an unsigned 8-bit displacement to the stack pointer. The data in the stack belonging to the interrupted routine is then restored and deallocated from the stack, returning control to the interrupted routine. At the completion of this instruction, the process's local priority will have been restored to its old saved value, or to the highest priority active channel, whichever is higher.

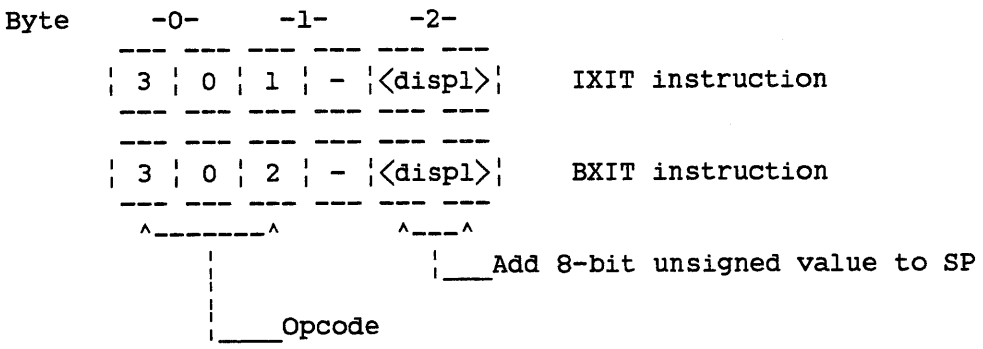
Interrupts push data into the stack in the following order: the next instruction address (as a 64-bit value), the local priority of the interrupted process (as a 64-bit value), and registers RF through R0. See Section 2.4, "Interrupts and Breakponts", for a complete description of the data placed in the stack.

12.5.4 BXIT Exit from Break

The BXIT instruction is used to exit from a routine entered through a BREAKPOINT instruction or by a single step break (usually a Debugger action). The contents of register R0 determines whether or not single stepping is enabled. If R0 = 0, then single stepping is disabled. If R0 = 1, then single stepping is enabled. The only other difference between the IXIT and BXIT instructions lies in the absence of the local priority on the stack if the routine entered is the result of a BREAKPOINT (R0 = 0).

Opcode assignment	Code	Implementation
IXIT	301	see above description
BXIT	302	see above description

The formats for these instructions are



Slightly different code is usually emitted when the stack contains dynamic arrays or arrays larger than 2**8 bytes.

13 INTER-PROCESS COMMUNICATIONS

A structural view of the ELXSI multiprocessor environment is that of time and location independent processes executing concurrently and communicating only through messages.

This chapter provides an introduction to the message system. Chapter 14 provides the details of the message system instructions. EMBOS Programmer's Reference Manual, Volume 3, introduces the concepts of job and process structures and provides a lower level detail of the message system.

13.1 MESSAGE SYSTEM OVERVIEW

The message system serves as the intelligent medium through which messages are sent. Following are some of the communication attributes in the message system environment:

- Process Concurrence. The communication structures that enable messages to be sent and received are created through a protocol that requires the participation and agreement of both processes.

- Transparency. A process may send or receive a message from any participating process (including itself) where the communication path exists to transfer the message. The message system allows the communication to be asynchronous through the provision of a message buffer; thus, messages may be queued until the receiving process is ready to bring the message into its process space. The message system makes no modifications to any data sent between processes.

- Process Substitution. A sending process need not be aware of the physical location or even the real identity of a receiving process. This allows the operating system to substitute processes to allow, for example, I/O redirection from a spooling process.

- Communication Security. All processes are bounded by their 32-bit virtual address space. Security and reliability of the system are enhanced as the communication attributes of a process are enforced through hardware protection. This has the advantages of hiding the data structures and implementation details of a given process from other processes, and of providing firewall protection.

  The assignment of communication links between processes is done on a "need to know" basis. Each process is responsible for controlling the incoming communication paths to itself. It does this by providing these communication paths only to those processes that must send messages to it. Consequently, with the distribution of communication paths managed in this way, each process is only given the communication paths to those processes

that it needs to send messages to. This scheme restricts the range of outside processes that any one process can communicate with, and it is not within the power of that process to expand this range (since no process is allowed to alter the tables that control its outgoing communication paths to other processes or the tables that control the handling of incoming messages from other processes). The only way it can expand this range is with the cooperation of the process that will be the recipient of its messages.

Many system security problems can be traced to malicious processes that have been able to forge their identities. With the ELXSI system, every message sent is tagged by the firmware with the process ID of the sender. The sender has no way of altering the process ID tag in the message, and it cannot alter the location that the firmware uses as the source of the process ID because this location is not even in the sender's address space. Consequently, the sender has no way to forge the process ID tag in the message, and the receiver can be assured that the sender's process ID is authentic.

- No Privileged mode. There is no privileged instruction mode in the ELXSI system. The process address space map, which controls access to memory, and the process link table, which controls access to the processes that manage other system resources, both reside outside any user's address space. Because user access to system address space, system processes, and system resources is so restricted, no privileged instruction mode is needed.

13.1.1 Communication Structures

The communication structure of a process consists of links, funnels, and channels and the tables that they reside in. These tables are all located outside the user process address space, and are not modifiable by the user. The user, however, has read access, via instructions, to the link and funnel tables and can look at individual link and funnel table entries. Only the firmware can write into these tables, and it is capable of adding or deleting table entries or of modifying fields within each of the entries.

Links are the address pointers over which messages are sent. As such, they generally point to outside processes, although they can also point to this process since a process can also send messages to itself. Each link corresponds to a single pointer and contains the process ID and version ID of the target process and the ID of the funnel that is to get the message. The funnel resides on the target process and is the mailbox slot into which the message goes when it is sent. Each funnel corresponds to a single mailbox slot and contains the channel that it is attached to and the head and tail pointers to the list of messages that have been delivered into this funnel, but not yet received (with the Receive instruction) by the target process. The channel that the funnel is tied to provides it with attributes that are essential to the

interrupt system used to modify the instruction execution flow for this process (when interrupting conditions occur) and the priority system used to schedule and allocate system resources for this process vis-a-vis other active processes on this particular processor. Each of these three basic elements of a process's communication structure is described in more detail below.

13.1.1.1 Links

Links serve as address pointers for the routing of messages in the message system. They are used by sending processes to direct messages into the funnels of target processes.

Links are composed of two basic elements: a data structure known as the link table entry (LTE), and an index into the link table entry known as the link ID.

Within the LTE is information on the link attributes, the process ID of the link creator, and the funnel of the link creator that will receive the messages. The link creator is the process that has created the link to enable it to receive messages from other processes.

The link creator passes or copies the link to the link holder, that is, the process from which the link creator wishes to receive messages. The link is passed or copied by sending the LTE information in a message. When the process receives the message, the incoming LTE information is placed into the link table of that process. The link ID specified by the receiver of the message is the index into the link table entry location for placement of the incoming LTE.

When the link holder wishes to send a message, the link ID of the link is incorporated as a parameter in the message. This link ID points to the LTE for that link, which in turn points to the location of the process that is to receive the message.

There may be up to 65,535 links for each process. A link must always be directed into exactly one funnel.

The link creator may specify other attributes of the link, including the rights of a link holder to copy or pass the link to another process, the right to receive notification if the link holder copies, passes, or deletes the link, and other attributes as specified in the link table data structure. If a link is copied, an additional link is directed into the funnel of the link creator, thus several processes may hold copies of the link, all directed into the same funnel.

13.1.1.2 Funnels

Funnels serve essentially as input ports through which a process may receive messages. The funnel ID is specified in the link entries for all of the links that are directed into the funnel. A process may have a total of 255 funnels, all of which may be attached to a given channel. Each funnel has an associated interrupt vector to allow an interrupt to be taken to an interrupt handling procedure if the channel to which the funnel is attached is enabled for interrupts. The interrupt is generated when a message arrives on a funnel attached to a channel which has interrupts enabled.

Messages are received on funnels in FIFO order. However, funnels attached to a given channel essentially have the same priority among themselves. The priority of the channel to which a funnel is attached determines the funnel's priority, and thus the priority of any messages arriving on a funnel attached to that channel.

Any number of links may be directed into a funnel. Funnels must at all times be attached to a channel, but may be moved to another channel if desired.

13.1.1.3 Channels

Each process has 16 channels, numbered 0 to 15, and each channel has an associated local priority that is the same as the channel number. The highest priority channel is channel 0, with the priority decreasing monotonically.

The channel priority map belonging to a process specifies a global priority associated with its local priority. This global priority is the process priority for that channel relative to other processes.

Channels can be interruptable. When a message arrives on a funnel attached to a channel that has interrupts enabled, and that channel is the highest priority active channel, and its priority is higher than the currently executing code within the process, an interrupt is taken to the interrupt handler address specified by the funnel interrupt vector.

An active channel is a channel that has one or more messages pending in one of its attached funnels. The local priority of a process is the channel ID of highest priority active channel. The arrival of a message may raise the local priority of a process but never lower it. Local process priority may be raised or lowered explicitly through an instruction provided for this purpose.

13.1.2 Message Composition

There are several types of messages. In the most typical case, that of a "simple" message, the length of the data is variable, separable into two message blocks if desired, with a maximum total length of 888 bytes. Messages may also contain links to be copied or passed from the sending process, or they may contain data to be forwarded to a tertiary process via an intermediate process. If a link is contained in a message, the remaining length of the message may extend up to 672 bytes.

Messages from a sending process are prepended by the firmware with the unforgeable process ID of the sending process. If the link to the receiving process is undefined, no message will be sent.

13.2 MESSAGE SYSTEM OPERATIONS

This section presents an overview of how communication paths are created between processes, and the general method through which messages are sent and received.

13.2.1 Establishing a Communication Path

A communication path must be established before any message may be sent or received. A process is conferred at birth with a certain number of links and funnels, known as standard links and standard funnels. These are discussed in the Programmer's Reference Manual, Volume 3.

To create a link, a funnel must exist for the link to be directed into. The process then creates a link directed into its own funnel and then passes or copies the link to the requesting process.

It is quite possible that a process (such as a spooler) might want communication in only one direction. The underlying principle of the message system is that processes communicate with each other by mutual consent according to some protocol they agree upon. The message system does not enforce some inherent protocol or synchronization. Rather, it is the manner in which the message system is used that can establish a protocol or synchronization.

13.2.2 Sending Messages

To send a message, the process first constructs a message in its data space that contains both the data to be sent and the link ID over which the message is to be sent. The data area containing the link ID is referred to as the "parameter block", and the data area immediately following as the "message block". When the process is ready to send the message, a SEND type instruction is invoked which copies the message from the sender's virtual address space into the system message buffer space.

The message system prepends, via firmware, the process ID of the sending process to the message. The message system firmware then performs the following:

1. It looks up the corresponding link entry in the link table belonging to the sending process, for which the link ID is the index. If the link entry does not exist, an error message is returned to the sender and the message is not sent. Otherwise the message is copied from the user's virtual address space into a linked group of free system message buffers.

2. Additional information is prepended to the message that specifies the type of message, the funnel ID, the number of bytes of data in the message, and other information depending on the type of message. This, along with the "from process ID", is encoded in a block of information known as "receive control information".

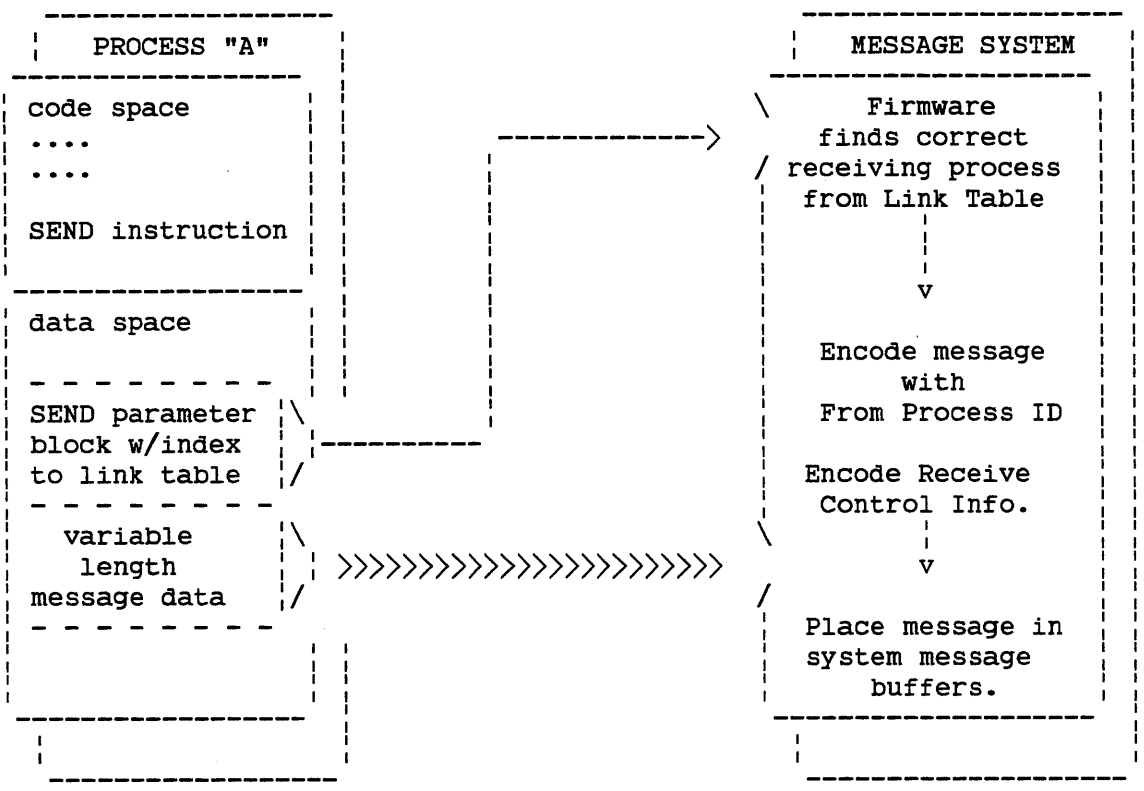
3. A Send Message Operation Bus Information Quantum (BIQ) pair is transmitted to notify the target process that an incoming message is pending for it. This BIQ-pair is fielded by the target unit's Send Message Operation Handler in a procedure called "message delivery". This operation handler responds to the sending process with bad status if the message cannot be delivered or with good status if delivery can take place.

4. Actual delivery of the message occurs when this operation handler attaches the message buffers containing the message to the funnel queue of the funnel specified in the sender's link table entry. At this point, the message continues to reside in the system message buffers until the target process executes a receive.

The important thing to note here is that the SEND instruction transmits only the Send Message Operation BIQ-pair, essentially 2 GigaBus-width words, to the target process to notify it that there is a message pending for it. The BIQ-pair contains the destination funnel ID and a pointer to the message buffer string having the actual message, and this is the information that message delivery uses to attach the message buffers to the funnel queue of the target process. The SEND instruction does not have to transmit the entire message across the GigaBus. Instead, the target process only gets the message transferred to it's virtual address space when the target process executes the receive operation against the message.

The following diagram illustrates the send message action:

Sending a Message

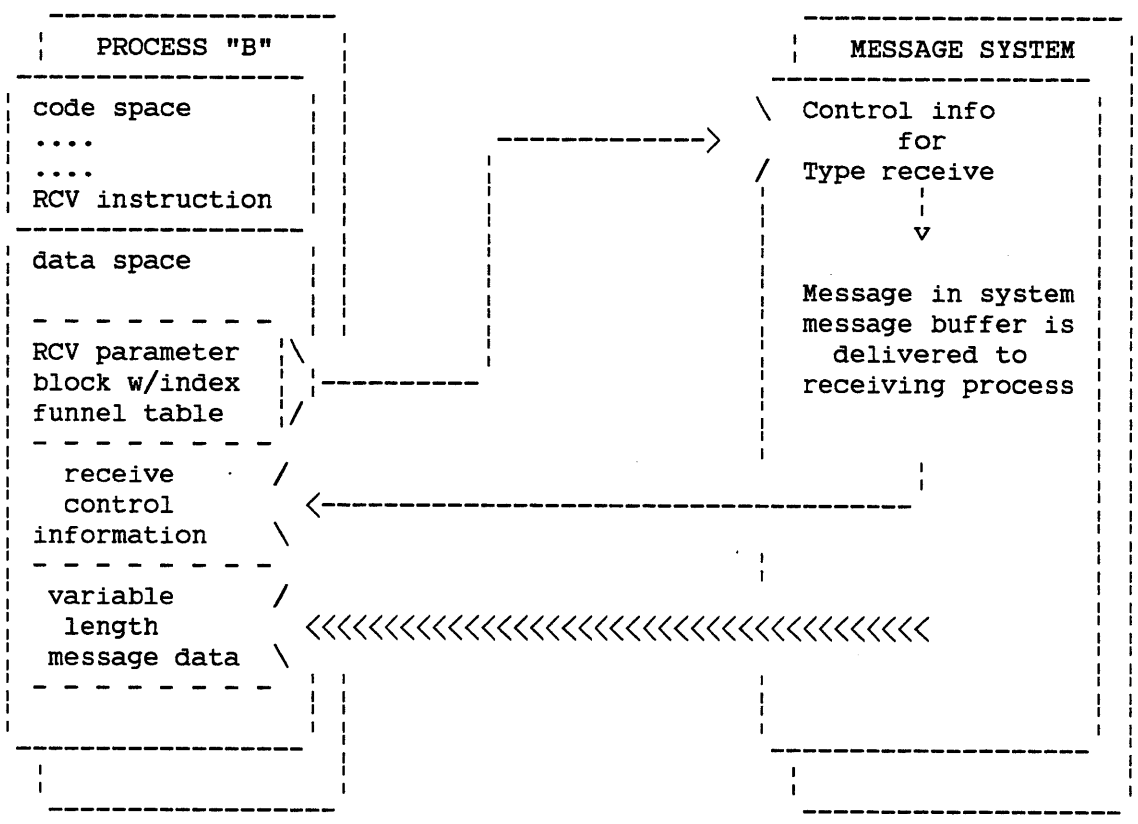


13.2.3 Receiving Messages

To receive a simple message, a process issues the receive instruction, specifies the target virtual data space for the message, and constructs the accompanying parameter block that specifies how the receive is to operate. When the receive is executed, the message, along with the receive control information, is transferred from the system message buffers into the virtual address space of the receiving process. The process can receive a message on a specific funnel by issuing a Receive On Funnel instruction. This would be the kind of receive issued if an incoming message arrived on an interrupting channel because the interrupt handler would know which funnel the message arrived on. Alternatively, the process can scan with the Receive On Channel instruction, and then receive the message from the highest priority active channel selected in the channel mask specified for that instruction. To receive a message containing a link, the process can issue a Receive Link On Funnel or Receive Link On Channel. These instructions allow the process to designate a link table location for storing the link in the message, in addition to the other link parameters.

The receiving process also has the option of suspending itself until a message has arrived on a particular funnel or channel. This can be performed by specifying a receive type of synchronous as opposed to a receive type of asynchronous. A synchronous Receive On Funnel issued against a funnel with no messages on it will cause the process to be blocked until a message does arrive on that funnel. A synchronous Receive On Channel issued against an inactive channel will also cause the process to be suspended until a message is delivered on any funnel that is attached to that channel.

Receiving a Message



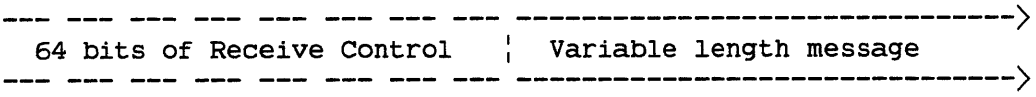
13.3 DATA STRUCTURES

The data structures provided in this section describe the link and funnel table entries and headers as well as data structures for messages. The parameter blocks are described in detail with the associated instructions in Chapter 14, "Message System Instructions".

13.3.1 Message

In general, a message consists of 0 to N bytes of information called message data. The maximum value for N depends upon the particular type of message:

Typical Form of Message in Transit



Instructions also exist for passing, copying, or forwarding links along with a message. The details of these different types of messages are discussed in Chapter 14.

13.3.1.1 Message Types

A simple message may contain from 0 to 888 bytes of message data which may be organized into one or two blocks as the user may determine convenient. Note that 888 bytes is the maximum message data length regardless of the number of blocks used.

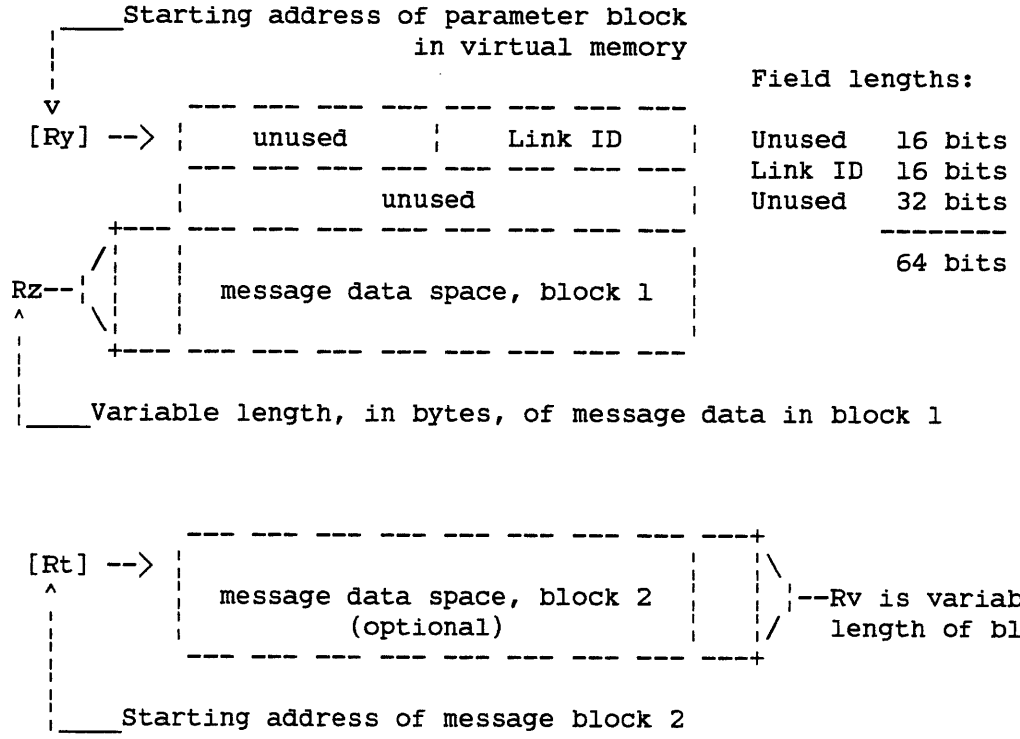
A small message may contain from 0 to 4 bytes of message data which must be organized into one block.

A To Hardware message may contain from 0 to 8 bytes of message data which must be organized into one block.

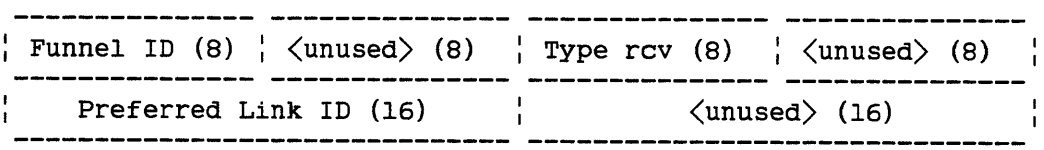
13.3.1.2 Parameter Blocks

Parameter blocks reside in the virtual address data space of a process and contain parameters for certain message system instructions. For a process that wishes to send a simple message, for example, the process constructs a parameter block containing the link ID. The message itself follows after the two words of the parameter block.

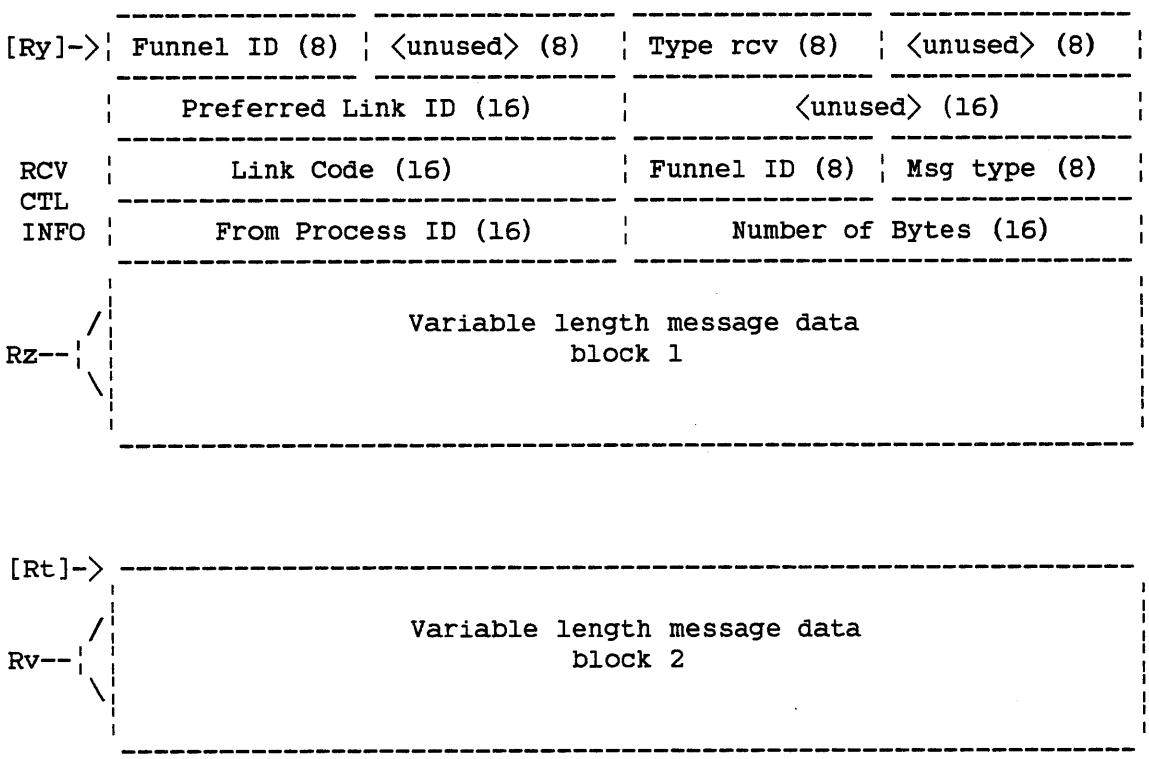
To illustrate usage, a simple message as represented in the virtual space of the sending process is shown below. The registers indicated are the entry parameters for a SEND instruction. The "link ID" identifies the link over which the message will be sent.



The RCV instruction uses the two words in the parameter block to specify the funnel and the type of receive desired. The parameter block is specified by the receiving process at some location in its virtual address space.



A typical message that has been received and placed in the virtual address space of the receiving process is shown below.



The specific formats for the parameter blocks are given with each of the receive instructions. The above fields are defined as follows:

Funnel ID

This 8-bit integer field in the first word of the parameter block has meaning when the instruction is a Receive On Funnel. The field identifies the funnel from which to receive a message, if any.

If the instruction is a Receive On Channel, the high order 16 bits of the first word (shown above as Funnel ID <8> and Unused <8>) contain a channel mask, where bit 0 corresponds to channel 0, bit 1 to channel 1,..., bit 15 to channel 15. This field causes the firmware to attempt to receive a message on every channel that has its corresponding bit set in the channel mask. The firmware starts with the highest priority channel and searches downward from there. It completes the receive when it finds a channel with a message or stops if there are no messages on any of the selected channels.

Type of Receive

This specifies one of three kinds of receive the user may execute. The boolean values occupy the low order 3 bits of the receive type field:

- Synchronous - If there is no message in the specified funnel, the user is blocked until a message is delivered into the funnel. If synchronous is not specified and no message is found, the receive instruction completes normally, and execution proceeds to the next instruction in the instruction stream.

- Interrogate - The message, if any, is copied into the user's virtual address space, but the message remains attached to the funnel. If there is no message, the action is controlled by the Synchronous state.

- Dismiss - This option invokes an implicit SET.LOCAL.PRIORITY instruction that will set the local priority of the process to either 15 or that of the highest priority active channel, whichever is higher.

Preferred Link ID

This field is used when the instruction is a Receive Link On Funnel or a Receive Link On Channel. The value in this field identifies the link table entry location where the link in the message should be placed. If the value is zero, a default location is selected from the list of free link table entries in order to receive the link in the message. The actual link ID of the link selected is returned in this field at the end of the instruction.

The registers specified in the above diagram are the input parameters for the receive type instructions and are used to place the message as described above into the address space of the receiving process.

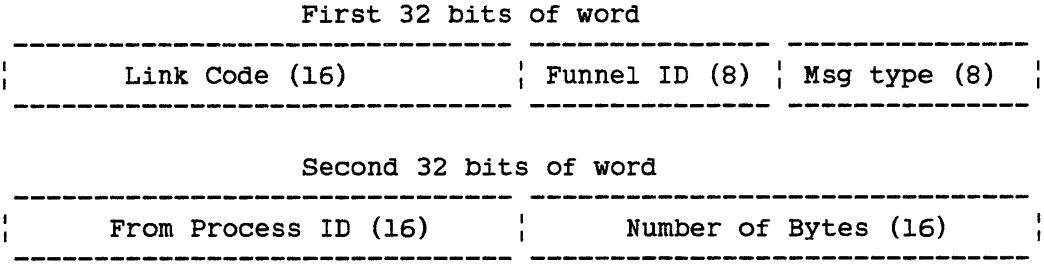
Note that the receiving process may select one or two data blocks, and may establish the lengths of either as desired, independently of the format of the message as it originally existed in the sender's address space. For example, even though the message existed as a single contiguous block in the sender's space, the receiver is free to specify one block to hold the parameter block data and another separate block to hold the message itself. Note, however, that the receive control information is always placed into the field following the receive parameter block specified by [Ry].

13.3.1.3 Receive Control Information

The firmware prepends 8 bytes of what is known as receive control information to the message when it is sent. This information is available to the receiving process and contains the link code of the link upon which the message was sent, the identification number of the funnel into which the message was delivered, the type of the message, the process identification number of the sender and the length of the message data in bytes. The receive control information is written by the firmware and is not forgeable by the the sender.

Although the receive control information is the same for all messages received by a process, the first part of the parameter block is dependent on the type of receive instruction, and is specified accordingly in Chapter 14.

Receive Control Information, message format



The Receive Control fields are defined as follows:

Link Code
: This is an arbitrary value associated with the link over which the message was sent. The sending process is the holder of the link.

Funnel ID
: This is the funnel into which the link is directed, and through which the message was received.

Message Type
: This field characterizes the incoming messages. It is typically used for notifying a link creator when one of its links has been copied, passed, or deleted. The notification message is sent automatically by the system to the link creator when the appropriate notify attribute for the link entry has been selected.

This boolean field is defined as follows (bits 0 through 2 are unused):

Small Message
: The message may contain up to 4 bytes of message data organized into one block. This is a special kind of send in which the receiving process allocates a message buffer for the message, rather than the sending process. Uses bit 3 of this field.

Link Deleted
: This indicates to the receiving process that the sender has deleted a link created by the link creator. Uses bit 4 of this field.

Link Copied
: This indicates to the receiving process (link creator) that a link has been copied. Uses bit 5 of this field

Link Passed
: This indicates to the receiving process (link creator) that a link has been passed on to another process. Uses bit 6 of this field.

Includes Link
: This indicates that the message contains a link. Uses bit 7 of this field.

From Process ID
: This is the ID of the sending process encoded into the message by firmware.

Number of bytes in message
: This is the length of the message data, not including the receive control information, in bytes.

Notice that if the length of the message data is zero, the receive control information will still be sent to the receiving process.

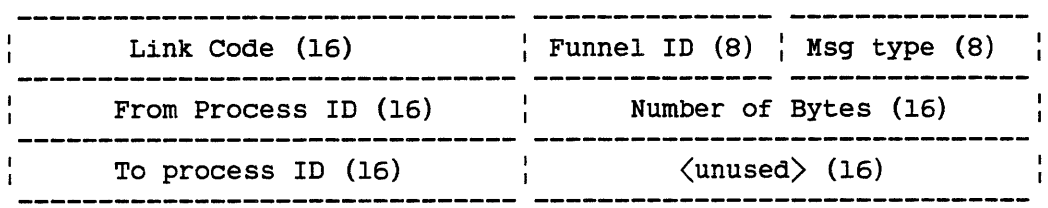
13.3.1.4 Notification

In many situations, it is useful for the creator or owner of a link to know if that link has been copied, passed, or deleted by one or more of the other processes that hold the link. If the link creator sets the notify attributes in the Link Rights field, then the link creator will be notified when the link is passed, copied, or deleted via a notification message sent along the affected link.

The instructions that copy links, pass links, or delete links are also the ones that generate the notification messages. Each of these instructions checks the notify attributes in the Link Rights field to determine if the notification should be sent. The notification is not sent if the corresponding attribute is not set, or if the link holder is also the link owner (It would not be useful for the link owner to

send a notification message to itself). The contents of the link deleted notification message is simply the receive control information that is sent with all messages, with the link deleted attribute set in the message type field. The link copied and the link passed notifications also contain the receive control information with either the link copied or the link passed attribute set in the message type field, along with an additional field that identifies the process to which the link was copied or passed.

Notification Message for Copy and Pass Link Instructions



The fields in the notification message are defined below:

Link Code
: This field specifies the link code of the link that was copied, passed, or deleted. Links are always directed into the funnel of the link creator.

Funnel ID
: This field specifies the funnel into which the link is directed, and upon which the message is received.

Message Type
: This field specifies through boolean values whether the link has been copied, passed, or deleted by the notifying process.

From Process ID
: This field specifies the process ID that executed the instruction to copy, pass, or delete the link.

Number of bytes
: The message block length "number of bytes" will be zero for delete link notification, and 2 for copy and pass link notification. This is because the To Process field is not sent on the delete link notification.

To Process ID
: This field identifies the process which had the link copied or passed to it. This field, and the unused field following it, are not used for the delete link notification message.

13.3.2 Links

The link creator has ultimate control over the link and can specify copy link rights, pass link rights, notification rights, or message rights. A process may have up to 65,535 links. This number may be restricted by the creator of the process.

13.3.2.1 Link Table Entries

A process can be a link owner and a link holder. Owned links are created by the link owner and point only into the funnels of the owning process. Consequently, the link owner is also sometimes called the link creator. Owned links are circular and always point back into the funnels of the owner process. The link owner can send a copy of a link it owns to another process by issuing a Copy Link or Pass Link instruction. The recipient of that link becomes the holder of the link and now has a path for sending messages to the link owner. Consequently, held links are always created by processes other than the link holder and point into the funnels of those other processes (which also happen to be the owners and the creators of these links). They always point away from the holding process and are the communication "links" between that process and the owning processes.

All of the links that belong to a process, whether they are owned or held, are stored in double-word-length packets called link table entries. These entries are organized into a link table, which contains a combination of defined link table entries and free link table entries. The defined link table entries are the ones that contain valid owned and held links. The free link table entries are those that are available for creating new links or for storing additional held links. These free entries are held together in a doubly linked list, for simplicity in adding or removing elements from this list.

The link ID is used as an index to a particular entry in the link table. Links are identified by their corresponding link ID, such that link 1 has link ID 1, link 44 has link ID 44, and so on. Link ID 0 for link 0 is reserved and points to the header entry of the link table. This entry contains information about the rest of the link table, such as the forward and backward pointers to the free list of link table entries, and whether or not more pages can be allocated to the link table.

A process can have up to 65,535 links, but in most cases can operate with far fewer. Thus, pages of memory are allocated to the link table as the links are used up. This is to avoid using large amounts of memory on "empty" link table space filled with free link table entries that would never be used. The task of managing link table space is done by the Link Manager.

At process creation time, each process is given, at the very minimum, a standard set of links. The pages needed to hold these links are allocated and frozen into memory. Any additional pages that are needed later can be allocated as required. However, regardless of the number of links initially given to the process, the page containing the link table header is always allocated. Only the firmware can directly access allocated Link Table pages.

The data structure describing a link contains the following information:

Link Table Entry

linkState
: Defined or undefined. A defined link is a link that has been successfully created. Messages are only allowed to be sent over a defined link.

The following fields have meaning if, and only if, the link State is defined.

toFunnel
: The identification number of the funnel into which the link is directed. A link may be directed into exactly one funnel.

linkRights
: These are specified by the creator of the link at the time of creation. There are three types of rights associated with each link, specified by the creator of the link at the time of link creation:

notificationRights
: These rights require the holder of the link to issue a notification message to the owner of the link when the link is copied, passed, or deleted. These notification rights are as follows:

informOnPass
: Send notification to the link owner when the link holder has passed the link to another process. Passing a link differs from copying a link in that the process passing a link has that link deleted from its link table at the end of the operation. A process that passes a link no longer holds that link, whereas a process that copies a link will continue to hold that link at the completion of the operation.

informOnCopy.
Send notification to the link owner when the link holder has copied the link to another process.

informOnDelete
Send notification to the link owner when the link holder has deleted the link from its link table.

copyLinkRights
These confer upon the holder of the link the right to copy the link and/or the right to pass the link, as follows:

canPass
The link holder is allowed to pass the link to another process.

canCopy The link holder is allowed to copy the link to another process.

msgRights
These confer upon the holder of the link the right of the holder to forward messages on the link and/or the right to send messages to the hardware (firmware) on the link. There rights are in addition to the right to send messages. For example, it is quite legal to send a regular message over a link with the toHardware right and it will be delivered to the software process specified in the link. However, it is not legal to send a message to the hardware over a link that does not have the To Hardware right. These are as follows:

forwardMessage
Messages may be forwarded over this link.

toHardware
Messages to the hardware (firmware) may be sent over this link.

toProcess
This is the process identification number of the process into whose funnel the link is directed.

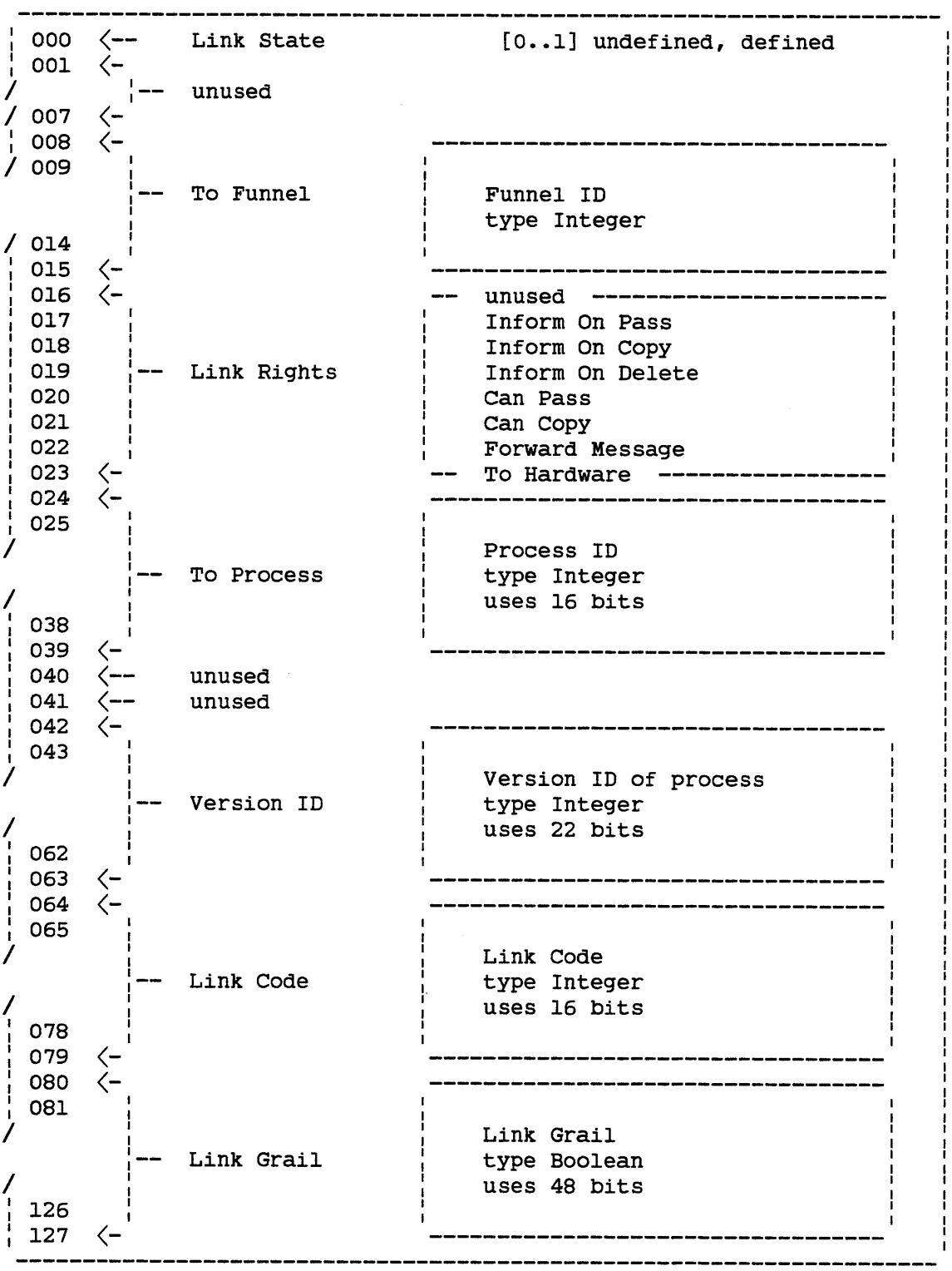
versionID
This is the version number associated with the process identification number in the toProcess field. The version ID prevents links with stale information in them from being used to send messages to new processes with recycled process ID's. For example, process A sends a message to process B and process B's process ID is the same as the toProcess ID in process A's link to process B. If B's version ID is different from the version ID in A's link to B, then process B is not the same process as the one that existed when A's link to B was created, and A is so informed.

linkCode
    This is a 16-bit unsigned number, which the creator of the link may associate with the link. It has meaning only to the creator and holder of the link. The link code is available to the receiver of a message in the receive control information section of the parameter block.

linkGrail
    This is a 48-bit array of booleans associated with the link.

The following table indicates the bits corresponding to each link entry in the link table. These link entries may be read by the READ.LTE instruction.

Table 13-1. Link Table Entry



13.3.2.2 Link Table Header

The link table header resides in the first doubleword of the link table and contains the following information:

linkHeaderState
: This is bit 0 and is always set to zero. The Link Table Header is always tagged as being undefined so it will never be mistaken for a valid defined link table entry.

maxCreatedLinkID
: This specifies the highest number link ID that has been created so far. Uses bits 16 to 31.

fwdFreeLink
: This is the index to the next free link in the free link list. Uses bits 32 to 47 of the first word.

revFreeLink
: This is the index to the previous free link in the free link list. Uses bits 48 to 63 of the first word.

allocatablePages
: This is a boolean bit, bit 0 of the second word, and specifies whether or not additional pages can be allocated to the link table.

allocatablePageCount
: This specifies the number of remaining pages that can be allocated to the link table for this process. Uses bits 8 to 23 of the second word.

13.3.3 Funnels

Each process in the system may have up to 255 funnels, although this number may be restricted by the creator of the process. Each funnel within the process has a unique identification number associated with it called the funnel ID. This number is used as an index into the table that contains the funnel table entries for this process. A valid funnel may have any number of links directed into it.

13.3.3.1 Funnel Table Entries

All the funnels within a process are organized into double-word entries in that process's funnel table. Each funnel table entry, as these are called, is either defined or free. A defined funnel table entry is one that has been created by the process and, as such, has its defined bit set and is attached to a channel. A free funnel funnel table entry is one that is available for creation and, as such, has its defined bit reset and is attached to the list of free funnels. This list consists of a doubly-linked group of free funnel table entries. Like the list of free links, the pointers to the head and tail of this list reside in

the header entry or funnel 0. An entry is removed from the free list when a funnel is created (by means of the CREATE.FUNNEL instruction), and a entry is added to the free list when a funnel is deleted (by means of the DELETE. FUNNEL instruction).

Each funnel table entry is indexed by an identifier called the funnel ID. Funnel ID's span the range 0 to 255. The full complement of funnels for a process can fit on two memory pages. Funnels with ID's 0 to 127 reside on the first page, and those with ID's 128 to 255 reside on the second page. Since most processes will never use even 127 funnels, they will be created with the second page of the funnel table unallocated. The first page will always be allocated, and the funnel table header entry will always be set up.

The data structure defining a funnel contains the following information:

Funnel Table Entry

funnelState
: Defined or undefined. A defined funnel is one that has been successfully created. A link may be configured into a defined funnel. Deleting a defined funnel causes it to become undefined.

The following fields have meaning if and only if the funnel State is defined.

channelID
: The identification number of the channel to which the funnel is attached. All defined funnels must be attached to exactly one channel. When a funnel is created it is attached to channel number 15. The user may move the funnel to any channel (except channel 0) with the Attach Funnel To Channel instruction.

nextFunnel
: This field only has meaning if there are messages queued on this funnel. Then it contains the funnel ID of the next funnel that also has messages queued up and is attached to the same channel. This singly linked pointer string forms what is called the active funnels list for that channel. Only active funnels (those with queued messages) are linked into this list. The funnel at the tail of this list will have zero value in its nextFunnel field.

messageCount
: The number of messages currently delivered to the funnel but not yet received. The count is a 16-bit integer.

interruptVector

The virtual address of the interrupt handler invoked when an interrupt is generated for this funnel. If interrupts are never generated for messages delivered into this funnel, then the interrupt vector may be zero. Note, however, that if an interrupt was ever to be generated with an address of zero, an access violation would occur.

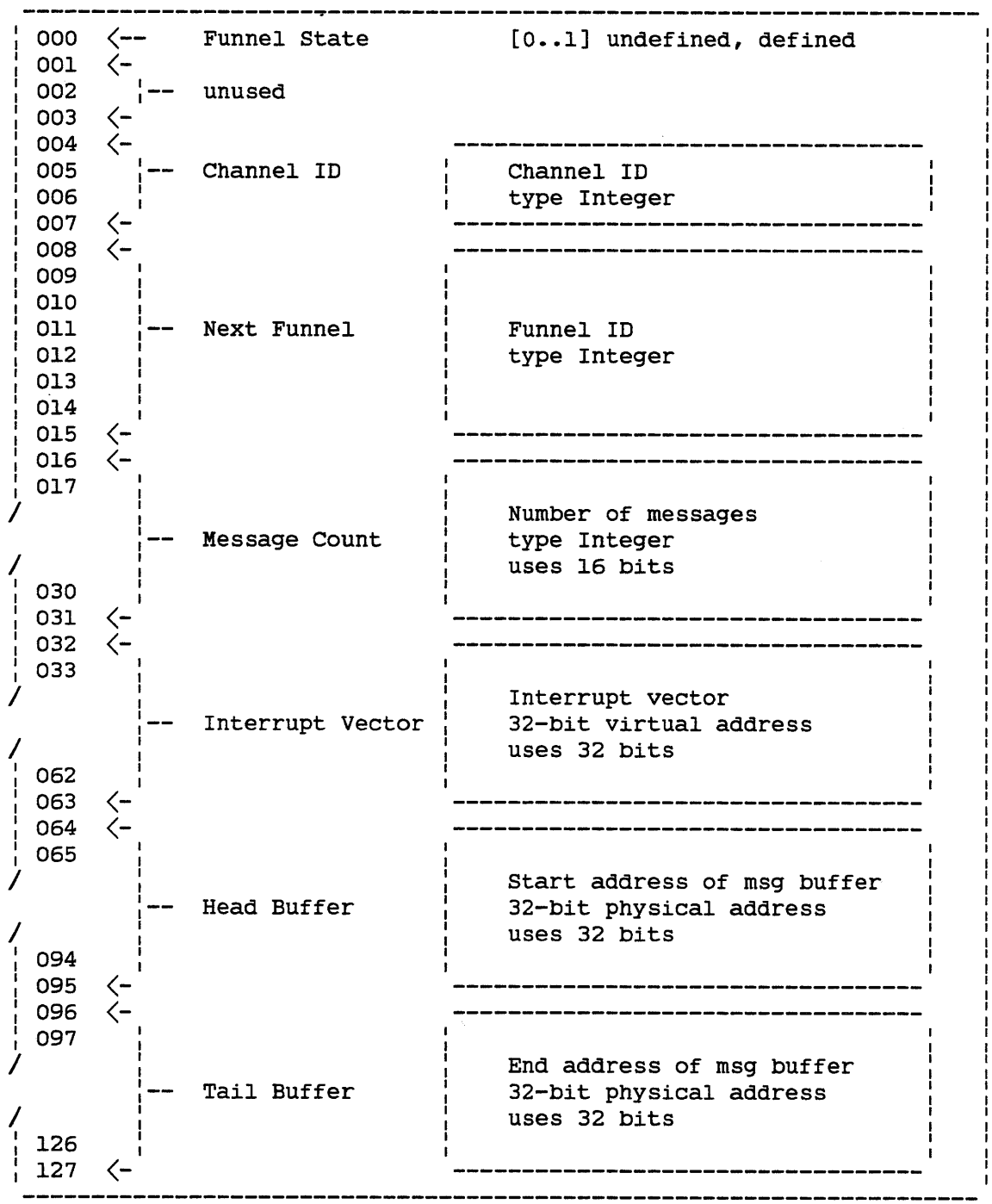
headBuffer

Messages delivered into a funnel, but not yet received, are organized into a singly linked list of system message buffers. HeadBuffer holds the physical address of the first message buffer in this list. When a receive is executed, headBuffer is used to dequeue the first message on this funnel.

tailBuffer

This field holds the physical address of the last system message buffer in the funnel's queue of unreceived messages. When a message is being delivered, tailBuffer is used to append the new message onto the tail of this list. The operations that enqueue and dequeue messages from the funnel queue exist to maintain the list in FIFO order. If no messages have been delivered into the funnel, then the head and tail buffer addresses are zero.

Table 13-2. Funnel Table Entry



13.3.3.2 Funnel Table Header

The funnel table header resides in the first doubleword of the funnel table and contains the following:

funnelHeaderState
: This is bit 0 of the first word and is always set to zero. The funnel table header entry is always tagged as undefined so that it will never be mistaken for a valid funnel table entry.

maxCreatedFunnelID
: This contains the highest numbered funnel ID created so far. This uses bits 16 to 23.

fwdFreeFunnel
: This is the ID (index) of the next free funnel in the list of free funnels. It uses bits 48 to 55.

revFreeFunnel
: This is the ID (index) of the previous free funnel the list of free funnels. This uses bits 56 to 63 of the first word.

allocatablePages
: This is bit 0 of the second word and specifies whether or not a second page can be allocated to the funnel table.

13.3.4 Channels

Channels allow users to organize funnels and prioritize the receiving of messages. Every process has 16 channels (channel 0 through channel 15), and every channel has a message system priority associated with it. The associated priority is the same as the channel number, with priority 0 being the highest and priority 15 the lowest. This priority is called the local priority because its scope is local to that process only. It has direct bearing on what that process will do next and which queued message will be received next. A channel may have from 0 to 255 funnels attached to it. All of the funnels attached to a particular channel will have that channel's priority.

Channels may be interrupting or non-interrupting. Interrupting channels are those that have been enabled for interrupts; non-interrupting channels are those that have been disabled for interrupts.

A channel mask is used to designate which of the 16 channels are interrupting and which are non-interrupting. Bit 0 of this mask is associated with channel 0, bit 1 with channel 1,..., bit 15 with channel 15. Each bit set on indicates that the corresponding channel is enabled for interrupts; each bit set off means that the corresponding channel has interrupts disabled. The user can modify the channel mask by issuing the ENABLE.CHANNEL.INT and DISABLE.CHANNEL.INT instructions.

There are, however, architectural restrictions placed on the interrupt structure:

1. The priority of interrupting channels must be higher than the priority of non-interrupting channels.

2. The above implies that interrupting channels and non-interrupting channels must exist in two separate groups, since an interrupting channel followed by a non-interrupting channel can never be followed by another interrupting channel.

To reiterate, the basic premise of the interrupt structure is this: The arrival of a message on a funnel that is attached to an interrupting channel will cause an interrupt to occur.

Channel 0 is used in special ways by the operating system. As such, it has some special restrictions associated with it:

1. Channel 0 is always interrupting and can never be disabled.

2. No funnel may be attached to channel 0 with the ATTACH.FUNNEL.TO.CHANNEL instruction. The only funnel attached to this channel is the lifeLine funnel (funnelID = 1), and this funnel can neither be disabled nor moved to another channel.

13.4 PRIORITY STRUCTURE OF MESSAGE SYSTEM

The priority structure is an ordering scheme through which execution priority is defined. There are two kinds of priority: Local Priority and Global Priority. Local Priority applies locally to a process and is used to determine which message that process will receive next or what task that process will perform next. Global Priority applies to all living processes on a particular CPU. It crosses process boundaries and is used by the Scheduler to determine the execution order of the processes that are ready to run.

Local Priority and Global Priority are related to each other and are mapped into one another in the channel priority map. This maps a local priority which can range in value from 0 to 15 into a global priority which can range in value from 0 to 255. In general, Local Priority values and Global Priority values track each other in the sense that a high local priority will be mapped into a correspondingly high global priority. There are, however, no architectural constraints that local/global priorities follow such a pattern.

13.4.1 Local Priority

Local Priority Local priority has meaning only within an individual process. Its value can be explicitly modified by the user with the SET.LOCAL.PRIORITY instruction or implicitly modified by the firmware (which performs the set local priority function when delivering messages or exiting from an interrupt with the IXIT instruction). The way the set local priority function works is to modify the local priority of the process to either the specified value or the channel number of the highest priority active channel, whichever is higher in priority. A channel is active when any funnel attached to that channel contains a message. As such, the local priority will often be at the priority of the highest priority active channel. For example, if funnel 200 with a queued message is attached to channel 10, funnel 14 with a queued message is attached to channel 5, and only these two channels contain messages, then the resulting local priority would be 5. Should a message be delivered into a funnel attached to a higher priority channel, then the local priority of the process would be raised. On the other hand, the delivery of a message into a funnel attached to a lower priority channel will not change the local priority.

Message delivery can only raise the local priority of a process. A process generally has its local priority raised to a high level so it can vie for additional system resources (at the expense of other processes) in order to get certain demanding tasks done in a timely manner. (Note that a high local priority value normally translates into a high global priority value, and a high global priority means a high scheduling priority). If the process were left at this high priority, it would continue to usurp system resources that it really did not need. Consequently, once the process has completed its high priority tasks, its local priority must be lowered. The can be done in several ways:

- It can issue the SET.LOCAL.PRIORITY instruction and specify that the local priority be set to the "base" local priority value for this process.

- The RECEIVE instruction that was used to receive the high priority message can specify the DISMISS option which on the completion of the receive operation will perform a a set local priority with a priority value of 15. This will lower the process priority to that of the highest priority active channel, or, if no channels are active, to a local priority of 15.

- If the high local priority has come about because of servicing an interrupt, the lowering of priority will occur automatically when the process returns from the interrupt handler through the IXIT instruction. The last part of this instruction performs a set local priority with the priority value set to the local priority that the process had prior to the delivery of the interrupting message (Chapter 2 describes how the old local priority is placed onto the stack on the occurrence of an interrupt). The process local priority upon return from the

interrupt handler will be the higher of either the restored old local priority or the value of the highest priority active channel.

13.4.2 Global Priority

Global priority is the execution priority of a particular process relative to all other processes in the system. It is used by the operating system to control resource allocation and may be dynamically modified to effect load balancing among all the processes executing in the system. A global priority is assigned to each of the 16 local priorities, and to each of the 16 channels since channel numbers and local priority values have a one-to-one correspondence. The mapping of these assignments is held in the channel priority map of that process's PCB (Process Control Block). The global priorities for each process are assigned to each channel at process creation time. These assignments are not modifiable by the user, but they may be modified by the operating system from time to time.

When a process becomes eligible for execution, it is put into a queue of eligible processes called the Active List. The processes in this list are ordered according to global priority, and those with high global priorities are given preferential treatment by the scheduling mechanism. As a result, high priority processes are allowed to execute sooner and more often than low priority processes. This priority scheme together with the setting of time quantums restricting how long each process can run at a single stretch form the basic components of the scheduling mechanism.

13.5 STANDARD COMMUNICATION PATHS

When a process is started, it has a set of standard links and funnels already created by EMBOS. These links and funnels have ID's in the range 1-20. This range of link and funnel IDs is reserved for EMBOS, so the user should not delete any predefined link or funnel, or create a link or funnel with a preferredLink(Funnel)ID in the range 1-20. Since all links and funnels in the range 1-20 are not presently used, a createLink (createFunnel) with a preferredLink(Funnel)ID of 0 may result in the creation of a link or funnel in the range 1-20. This will cause no problems, since all EMBOS-reserved links and funnels are created before the user's code begins executing.

13.5.1 Standard Links

The standard links for each process consist of special self-links and communcation paths to the process' parent and to certain system processes. The only LinkID relevant to this document is the following:

userExceptionLinkID (linkID = 1)
: This link is directed into the process' userExceptionFunnel (funnelID = 2) and is used by firmware and software to send user interceptable exception messages to the process.

13.5.2 Standard Funnels

The standard funnels for each process consist of funnels into which self-links, and links from the process' parent, and certain system processes are directed. The two funnels relevant to this document are:

userLifelineFunnel (funnelID = 1)
: This funnel is attached to channel 0. The firmware prevents this funnel from being altered in any way. It cannot be deleted, disabled, attached to a different channel, have its interrupt vector changed, or have links created into it. Any receive on the funnel will always find it empty, so a synchronous receive will wait forever (or until an interrupt occurs).

userExceptionFunnel (funnelID = 2)
: This is the funnel into which the userExceptionLink belonging to the process is directed. The funnel is attached to channel ID 1, and has an interrupt vector which points to an EMBOS supplied exception dispatcher. All user interceptible exceptions arrive on this funnel. The exception dispatcher receives all exception messages and invokes the appropriate exception handler.

14 MESSAGE SYSTEM INSTRUCTIONS

The message system instructions are used for implementing and modifying communication structures. These instructions should not be considered as the primary user interface to the message system, as software for this purpose is included in all ELXSI supplied languages.

ATT.FUN.TO.CHAN	Attach Funnel to Channel
COPY.LINK	Copy Link
CREATE.FUN	Create Funnel
CREATE.LINK	Create Link to Funnel
DEL.FUN	Delete Funnel
DEL.LINK	Delete Link
DEL.MSG	Delete Message from Funnel
DISABLE.CHAN.INT	Disable Interrupts on Channel
DISABLE.FUN	Disable Funnel
ENABLE.CHAN.INT	Enable Interrupts on Channel
ENABLE.FUN	Enable Funnel
EXCH.LINK.FORWARD	Exchange Message Link and Forward
FORWARD.MSG	Forward Message
PASS.LINK	Pass Link
RCV	Receive Message
RCV.CHAN	Receive Message on Channel
RCV.LINK	Receive Message with Link
RCV.LINK.ON.CHAN	Receive Message with Link on Channel
READ.FTE	Read Funnel Table Entry
READ.LTE	Read Link Table Entry
SEND	Send Message
SEND.SMALL.MSG	Send Small Message
SEND.TO.HARDWARE	Send Message to Hardware
SET.FUN.INT.VECTOR	Set Funnel Interrupt Vector
SET.LOCAL.PRI	Set Local Priority

The message system instructions return a status code to the calling process. These codes tell the user whether the operation was successful or not, and can also provide information on abnormal conditions encountered during execution of the instruction. The status code is instruction specific and is returned in register Rx as a right justified integer value.

A list of the status codes is provided with each instruction. The column to the right of these codes provide information on whether or not the instruction executed successfully. A "y" indicates that the associated status is merely an advisory. An "n" indicates that the instruction has partially or completely failed.

A summary of the codes is provided below:

Table 14-1. Status Return Codes

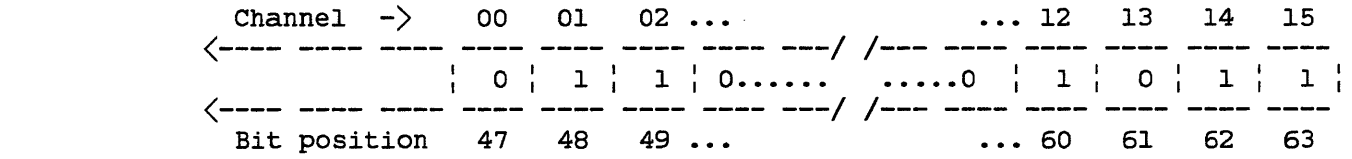
1 =	MSYS Funnel Does Not Exist
2 =	MSYS Illegal Channel ID
3 =	MSYS Funnel Not Disabled
4 =	MSYS Funnel Not Empty
5 =	MSYS Transport Hardware Error On Notification
6 =	MSYS Message Too Long
7 =	MSYS Illegal to Modify Interrupt Vector On This Funnel
8 =	MSYS Funnel Table Is Full
9 =	MSYS Illegal Preferred Funnel ID
10 =	MSYS Funnel Already Created
11 =	MSYS Link Table Is Full
12 =	MSYS Link Already Created
14 =	MSYS No Message In Funnel
15 =	MSYS Illegal To Disable Interrupts
17 =	MSYS Illegal To Disable Funnel ID
18 =	MSYS Specified Funnel Already Disabled
21 =	MSYS Specified Funnel Already Enabled
23 =	MSYS Cannot Forward On Link
28 =	MSYS Message Contains Link
29 =	MSYS No Messages On Channels
30 =	MSYS Message Does Not Contain Link
31 =	MSYS Link Does Not Have Hdwr Right
32 =	MSYS Illegal Preferred Link ID
34 =	MSYS Illegal To Move Funnel
36 =	MSYS Link 1 Receiving Process Dead
37 =	MSYS No Message Buffer Available
38 =	MSYS Too Many Buffers In Transit
39 =	MSYS Link 1 Too Many Attached Buffers
40 =	MSYS Link 1 Link Not Defined
41 =	MSYS Link 1 Unallocated Page
42 =	MSYS Link 1 Unallocatable Page
43 =	MSYS Link 1 Exceeds No Of Link Levels
45 =	MSYS Link 1 Zero Link ID
46 =	MSYS Negative Data Block Length
47 =	MSYS Link 1 Funnel Not Enabled
48 =	MSYS Destination Process Not On Target Unit
49 =	MSYS Bad Pointer In Free Link
50 =	MSYS Access Unallocated Page Of Funnel Table
51 =	MSYS Link Cannot Be Sent
52 =	MSYS Link 1 Target Unit Busy
54 =	MSYS Message Not Sent
54 =	MSYS Transport Hardware Error On Data Message
55 =	MSYS Transport Hardware Error
56 =	MSYS Link Fault
136 =	MSYS Link 2 Receiving Process Dead
139 =	MSYS Link 2 Too Many Attached Buffers
140 =	MSYS Link 2 Link Not Defined
141 =	MSYS Link 2 Unallocated Page
142 =	MSYS Link 2 Unallocatable Page

```
143 =  MSYS Link 2 Exceeds No Of Link Levels
145 =  MSYS Link 2 Zero Link ID
147 =  MSYS Link 2 Funnel Not Enabled
152 =  MSYS Link 2 Target Unit Busy
```

Descriptions of the data structures, such as the Funnel Table Entry (FTE) and the Link Table Entry (LTE), may be found in Chapter 13, "Inter-Process Communications". The message length, when specified, is in bytes. All unused fields must be set to zero.

Channel masks are 16-bit right justified fields that are used to select or deselect channels for particular instructions. Each bit corresponds to a channel. If the bit is set to '1', the indicated function is performed for that channel. Bits set to '0' have no effect.

CHANNEL MASK in Ry, right justified



The channels selected above are channels [0,1,12,14,15]. Observe that the logical relation of this mask to the object is defined by the instruction, whereas a returned operand will be the actual object. For example, assume that the enabled channels in the channel mask happen to be = 1010101010101010 and we perform a DISABLE.CHAN.INT with the above mask. The new word will be 1000101010100000 and the returned (old) word will be 1010101010101010.

14.1 CREATE A COMMUNICATION PATH

To send a message to another process, a link must be established that is directed into the funnel of the receiving process. One may then send messages via the link.

If a process (A) is already sending to process (B) but wishes to receive messages from process (B), process (A) must create a link directed into one of its own funnels, and then copy or pass the link to process (B). The link attributes are discussed in Chapter 13, "Inter-Process Communications".

14.1.1 CREATE.LINK Create Link to Funnel

All links are created with this instruction and are self links, that is, they are directed into one of the creator's own funnels. The link configuration is specified in the create link parameter block whose address is in Ry. The instruction places the configuration into the link table belonging to the process. Upon completion, status is returned in Rx.

Implementation						Addressing mode usage
8	0	2	x	y	0	CREATE.LINK Rx,[Ry]

Opcode

y: This field must be zero

x: Ry parameter block address

Rx = Instruction Specific Return Status

0	Success	[y]
1	Funnel Does Not Exist	[n]
11	Link Table Is Full	[n]
12	Link Already Created	[n]
32	Illegal Preferred Link ID	[n]
50	Access Unallocated Page Of Funnel Table	[n]

If the link ID specified in the parameter block is zero, then the next free link is created, otherwise the specified link is created. The link ID actually created is returned in the parameter block in the link ID field. If [Rx] <> 0, then the link is not created. Following are descriptions of the parameter block fields.

Link ID
: This is the link ID of the link to be created or zero, if the next undefined link ID is to be used. The firmware returns the link ID actually created in this field upon successful completion.

Funnel ID
: This is the funnel ID in the creator's Funnel Table into which the link being created will be directed. The funnel must have been created prior to creating the link.

Link Rights
These are boolean rights associated with the link, presented from high order to low order in the field. The To Hardware Right may never be specified for this instruction. These rights are as follows:

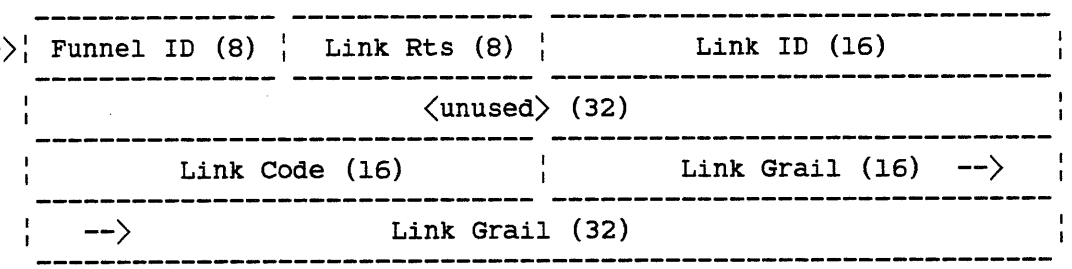
Unused
Inform on Pass (notify link creator when passing link)
Inform on Copy (notify link creator when copying link)
Inform on Delete (notify link creator when deleting link)
Can Pass Link (link can be passed to another process)
Can Copy Link (link can be copied to another process)
Can Forward Message (messages may be forwarded on this link)
To Hardware (messages to hardware may be sent on this link)

Link Code
This is a 16-bit unsigned integer which may contain information specified by the link creator. The default value is 0.

Link Grail
This is an array of 48 booleans. The default is 48 bits of zero.

A full description of these data structures may be found in Chapter 13, "Inter-Process Communications".

CREATE.LINK Parameter Block

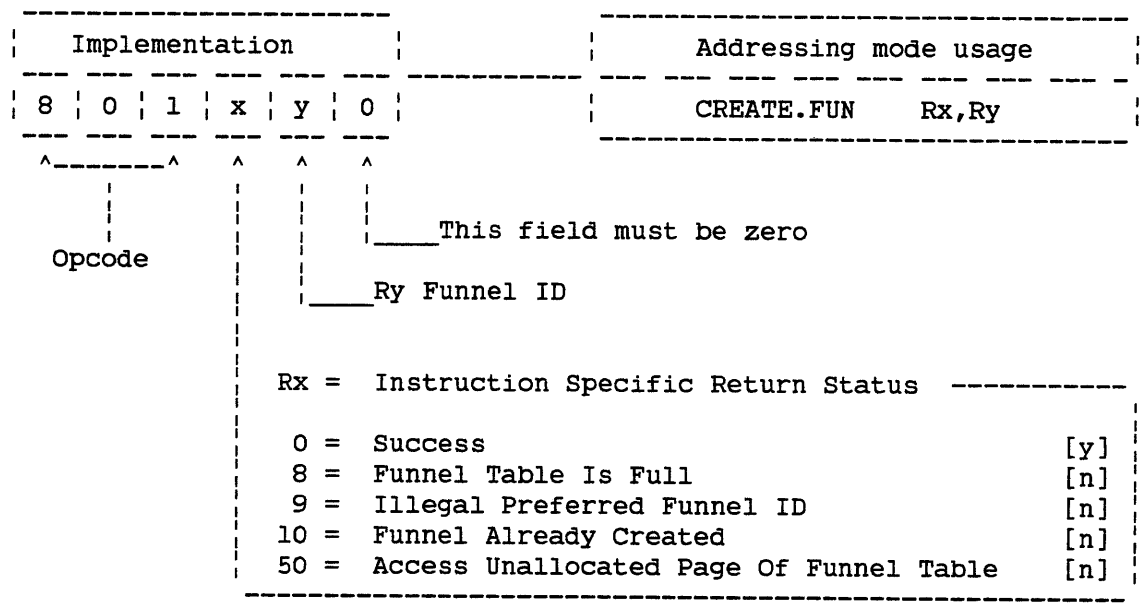


14.1.2 CREATE.FUN Create Funnel

Before a link can be created into a funnel or a message received on a funnel, the funnel must be created. Funnels are always created attached to channel 15.

The funnel specified in Ry is created as follows: if [Ry] = 0, the next funnel on the list of free funnels is used, otherwise the specified funnel is used. The funnel table entry corresponding to the specified funnel ID is removed from the list of free funnels. Its defined bit is set, it is attached to channel 15, and the rest of its fields are set to initial values. The funnel channel map entry for this funnel is updated to show this funnel attached to channel 15.

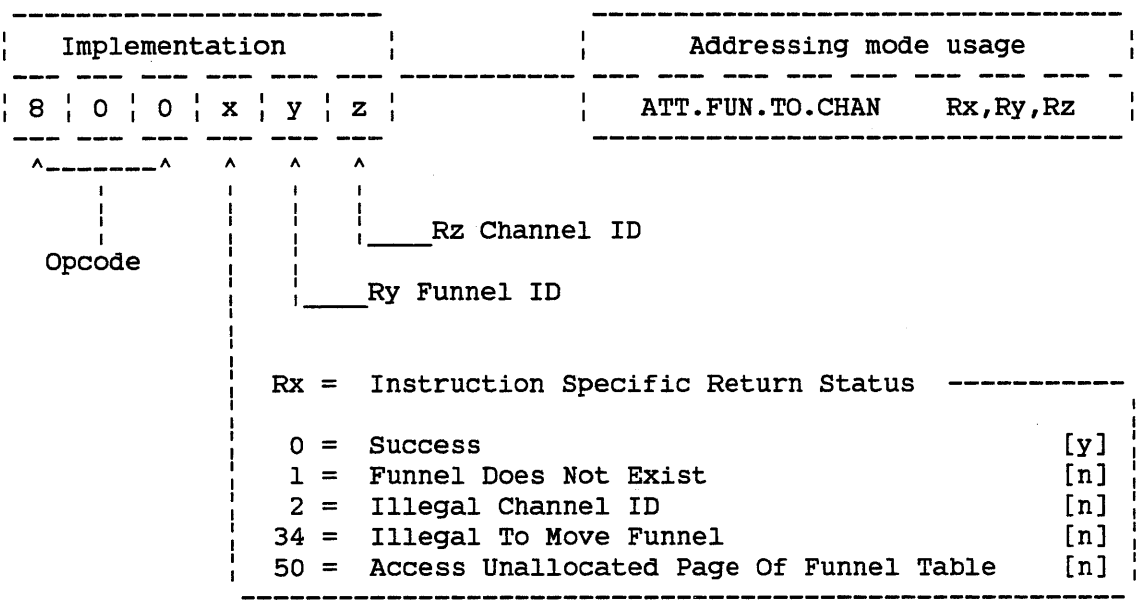
Upon completion, the actual ID of the funnel created is returned in Ry, and the status is returned in Rx. If [Rx] <> 0, then no funnel is created.



14.1.3 ATT.FUN.TO.CHAN Attach Funnel to Channel

All funnels are attached to channel 15 when created. This instruction allows the user to move a funnel to another channel.

The instruction attaches the funnel specified in Ry to the channel specified in Rz. Upon completion, status is returned in Rx and the channel to which the funnel was previously attached is returned in Rz. If [Rx] <> 0, then Rz is unchanged. A funnel cannot be attached to channel 0.



This instruction modifies the funnel table entry and the funnel channel map for the funnel specified in Ry. The addresses of both data structures are known to the system.

The funnel specified in Ry must have been created with the CREATE.FUN instruction prior to executing this instruction.

14.2 DESTROY A COMMUNICATION PATH

These instructions delete the links and funnels belonging to a process. Deleting standard links and funnels may produce unexpected results.

14.2.1 DEL.LINK Delete Link

The specified link is deleted from the link table and returned to the list of free links. The first word in the parameter block contains the link ID to be deleted. If "Inform on Delete" is specified in the link entry for the link to be deleted, a notification message is sent to the creator of the link. Ry contains the address of the parameter block. The notification message will not be sent if the link creator deletes a link that is directed into one of its own funnels.

Implementation								Addressing mode usage
9	0	1	x	y	0	-	-	DEL.LINK Rx,[Ry]

Opcode

y = Ry parameter block address

0 = This field must be zero

Rx = Instruction Specific Return Status

0 =	Success	[y]
36 =	Link 1 Receiving Process Dead	[y]
40 =	Link 1 Link Not Defined	[n]
41 =	Link 1 Unallocated Page	[n]
42 =	Link 1 Unallocatable Page	[n]
43 =	Link 1 Exceeds No Of Link Levels	[n]
45 =	Link 1 Zero Link ID	[n]
47 =	Link 1 Funnel Not Enabled	[y]
55 =	Link 1 Transport Hardware Error	[y]
139 =	Link 1 Too Many Attached Buffers	[y]

DEL.LINK Parameter Block

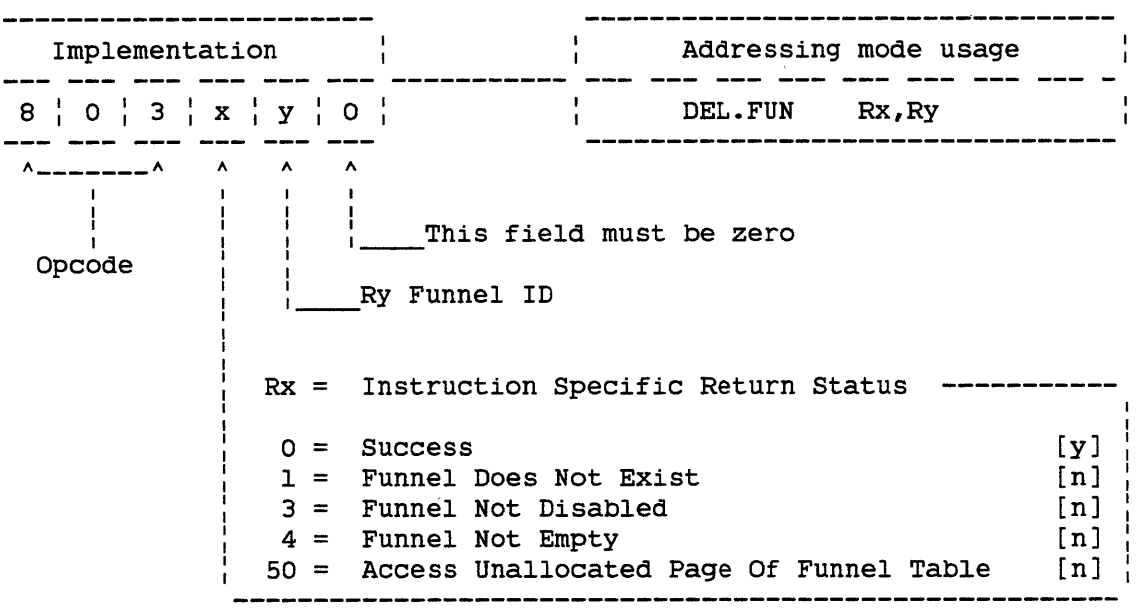
[Ry]->		
	<unused> (16)	Link ID (16)
	<unused> (32)	

Notification Message to Link Creator

	Link Code (16)	Funnel ID (8)	Msg type (8)
	From Process ID (16)	Number of Bytes (16)	

14.2.2 DEL.FUN Delete Funnel

The funnel specified in Ry is deleted by returning its entry in the funnel table to the free list. Prior to deleting a funnel, it must be disabled by executing the DISABLE.FUN instruction, and all messages must be removed from its funnel queue by executing the DELETE.MESSAGE instruction. Upon completion, status is returned in Rx.



14.3 SEND MESSAGES

There are three types of messages a user may send. They are listed below by the instructions used in sending them.

- SEND. This is used to send the typical message. The length of this message may vary from 0 bytes to a maximum of 888 bytes. The message data to be sent can be organized into one or two (not necessarily contiguous) data blocks in the user's virtual space. The message preamble and the message data are transferred into message buffers (see Chapter 13, "Inter-Process Communications") contributed by the sending functional unit.

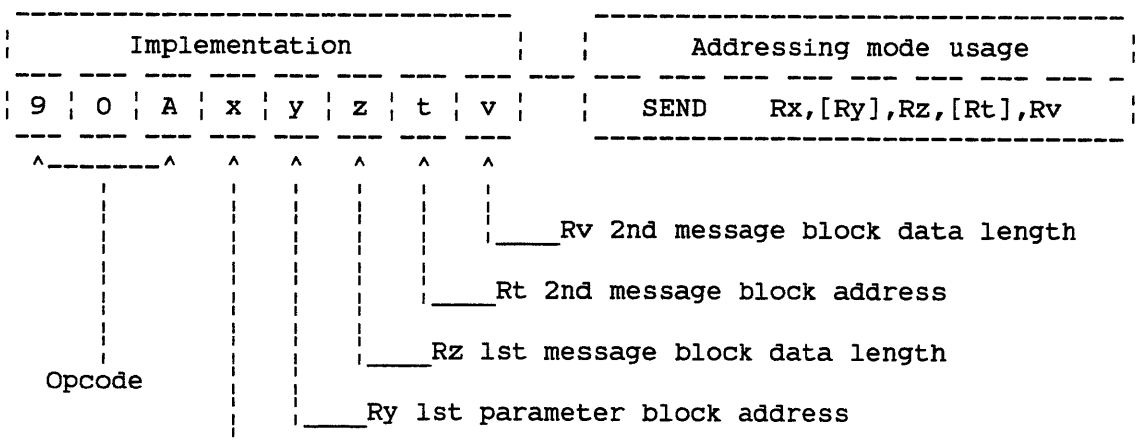
- SEND.SMALL. This type allows the user to place a 32-bit message into a register and send it to another process. The receiving functional unit, rather than the sending functional unit, contributes the message buffer for this kind of send.

- SEND.TO.HARDWARE. This sends a message to the hardware (CPU) being used by the process into which the link is directed. A link can only have the toHardware link right conferred to it via system intrinsics not covered in this section. No message buffer is required to send this 64-bit message to the hardware.

The maximum size of a message is 888 bytes. If a link is contained in the message, as with COPY.LINK or PASS.LINK, the link uses 16 bytes, leaving a remainder of 872 bytes for message data.

14.3.1 SEND Send Message

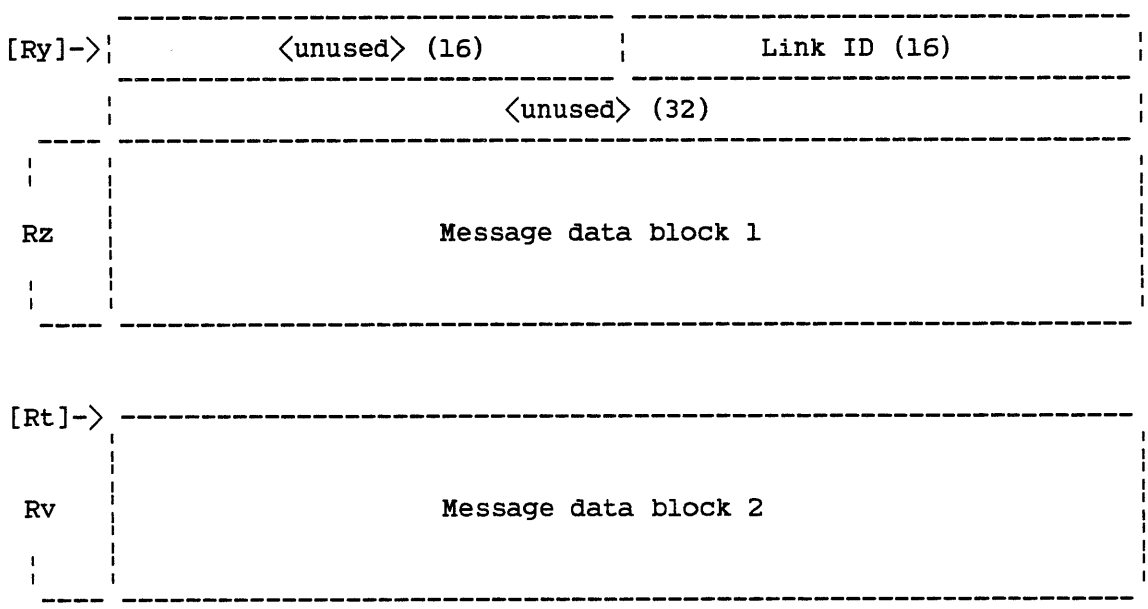
This instruction copies the specified message from the caller's virtual address space to the system message buffers, then attaches the buffer to the funnel specified in the link table entry for the sending process. Ry contains the virtual address of the send parameter block and Rz the length of the message data. If [Rv] <> 0, then the message is in two parts, with Rt containing the virtual address of the second message block and Rv the length. Upon completion, status is returned in Rx.



Rx = Instruction Specific Return Status

0 =	Success	[y]
6 =	Message Too Long	[n]
36 =	Link 1 Receiving Process Dead	[n]
39 =	Link 1 Too Many Attached Buffers	[n]
40 =	Link 1 Link Not Defined	[n]
41 =	Link 1 Unallocated Page	[n]
42 =	Link 1 Unallocatable Page	[n]
43 =	Link 1 Exceeds No Of Link Levels	[n]
45 =	Link 1 Zero Link ID	[n]
46 =	Negative Data Block Length	[n]
47 =	Link 1 Funnel Not Enabled	[n]
55 =	Transport Hardware Error	[n]

SEND Parameter Block



14.3.2 SEND.SMALL.MSG Send Small Message

The data in the low order 32 bits of Rz (bits 32 through 63) are formed into a message, and sent on the link specified in Ry. High order bits 0 through 31 of Rz must be zero. Upon completion, status is returned in Rx.

Implementation							Addressing mode usage								
8	0	E	x	y	z		SEND.SMALL.MSG			Rx,Ry,Rz					

Opcode

Rz Message data

Ry Link ID

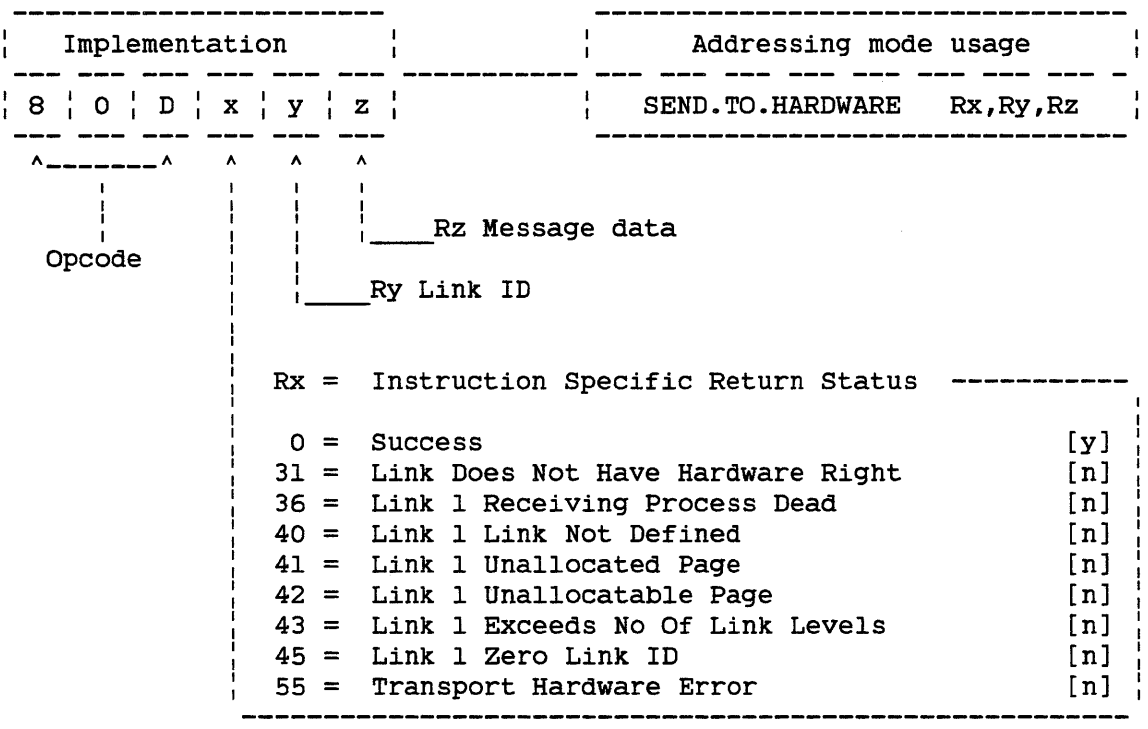
Rx = Instruction Specific Return Status

0 =	Success	[y]
36 =	Link 1 Receiving Process Dead	[n]
37 =	No Message Buffer Available	[n]
40 =	Link 1 Link Not Defined	[n]
41 =	Link 1 Unallocated Page	[n]
42 =	Link 1 Unallocatable Page	[n]
43 =	Link 1 Exceeds No Of Link Levels	[n]
45 =	Link 1 Zero Link ID	[n]
47 =	Link 1 Funnel Not Enabled	[n]
55 =	Transport Hardware Error	[n]

14.3.3 SEND.TO.HARDWARE Send Message to Hardware

This instruction is similar to the SEND.SMALL.MSG instruction, except that the message is not received by the specified target. Instead, the message is interpreted by the CPU that owns the target process. The 64-bit data in Rz is formed into a message, and is sent on the link ID specified in Ry. Upon completion, status is returned in Rx.

The link on which this message is sent must have toHardware rights specified by the link creator.



14.3.4 COPY.LINK Copy Link

This instruction copies a link along with a message from the caller's virtual address space to the system message buffers, then attaches the buffer to the funnel specified in the caller's link table. This instruction is identical to SEND with the addition of a link ID to copy as specified in the parameter block. A link must be defined in order to be copied.

The receiver of a link places the link into its link table. The link attributes of the original and the copied link both have the same funnel, that is, an additional link is now directed into the funnel.

Ry contains the address of the first send parameter block and Rz contains the length of the data (message) in the parameter block. Rt contains the address of the second message block and Rv contains the length of the data in the second message block. If [Rv] = 0, there is no second message block and the contents of Rt have no meaning. Upon completion, the status is returned in Rx.

A notification message is sent to the creator of the link only if "notify on copy" is specified in the link entry for the copied link. A notification message is never sent, regardless of the setting of the "notify on copy" bit, if the owner (creator) of the link is the process invoking COPY.LINK.

Implementation								Addressing mode usage
9	0	0	x	y	z	t	v	COPY.LINK Rx,[Ry],Rz,[Rt],Rv

Opcode

Rv 2nd message block data length

Rt 2nd message block address

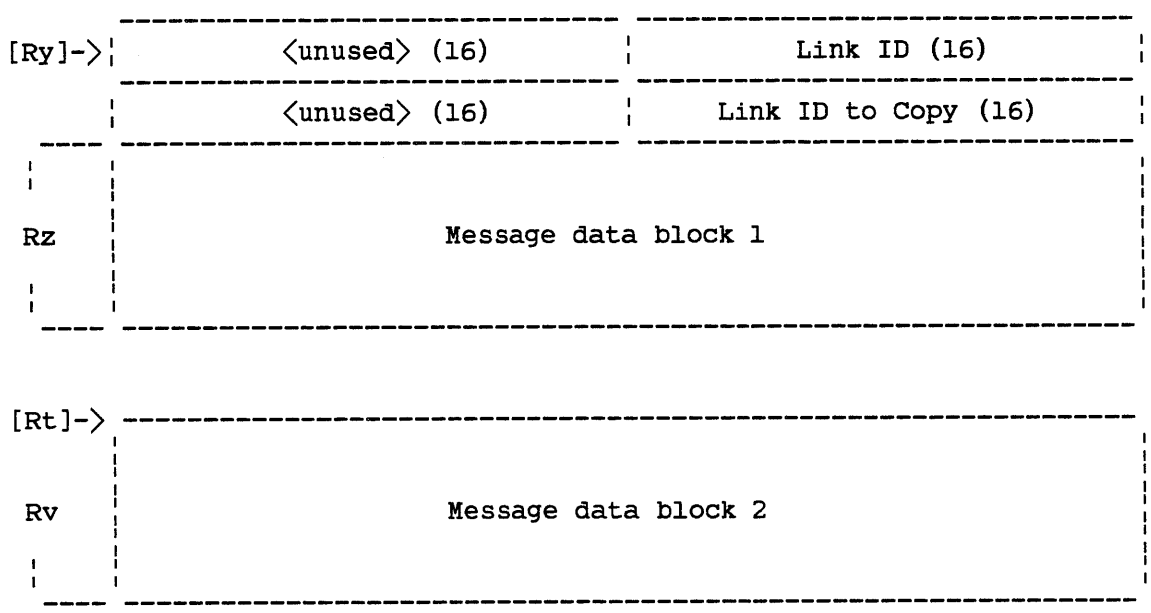
Rz 1st message block data length

Ry 1st parameter block address

Rx = Instruction Specific Return Status

0 =	Success	[y]
5 =	Transport Hardware Error On Notification	[n]
6 =	Message Too Long	[n]
36 =	Link 1 Receiving Process Dead	[n]
39 =	Link 1 Too Many Attached Buffers	[n]
40 =	Link 1 Link Not Defined	[n]
41 =	Link 1 Unallocated Page	[n]
42 =	Link 1 Unallocatable Page	[n]
43 =	Link 1 Exceeds No Of Link Levels	[n]
45 =	Link 1 Zero Link ID	[n]
46 =	Negative Data Block Length	[n]
47 =	Link 1 Funnel Not Enabled	[n]
51 =	Link Cannot Be Sent	[n]
52 =	Link 1 Target Unit Busy	[n]
54 =	Transport Hardware Error On Data Message	[n]
136 =	Link 2 Receiving Process Dead	[n]
139 =	Link 2 Too Many Attached Buffers	[n]
140 =	Link 2 Link Not Defined	[n]
141 =	Link 2 Unallocated Page	[n]
142 =	Link 2 Unallocatable Page	[n]
143 =	Link 2 Exceeds No Of Link Levels	[n]
145 =	Link 2 Zero Link ID	[n]
147 =	Link 2 Funnel Not Enabled	[n]
152 =	Link 2 Target Unit Busy	[n]

COPY.LINK Parameter Block



Notice that execution of the COPY.LINK instruction simply allows the receiving process to place the link in the message into its link table. The link owner now has two links directed into one of its funnels: the original link that the process created or held that was directed into its own funnel, and the copied link now held by the receiver of the message.

The notification message to the creator of the link, if sent, has the following format:

COPY.LINK notification message

Link Code (16)	Funnel ID (8)	Msg type (8)
From Process ID (16)	Number of Bytes (16)	
Receive Process ID (16)	<unused> (16)	

14.3.5 PASS.LINK Pass Link

PASS.LINK is identical to COPY.LINK, except that after the link to be passed is put in the message, the entry for it is deleted from the holder's link table. When the receiver of the link does a receive link, there will still be only one link. Ry contains the address of the first send parameter block and Rz contains the length of the data (message) in the parameter block. Rt contains the address of the second send parameter block and Rv contains the length of the data (message) in the parameter block. If [Rv] = 0, then there is no second parameter block, and the contents of Rt have no meaning. Upon completion, status is returned in Rx.

If the Link Rights field of the link to be passed specifies "inform on pass", then a default notification message will be sent to the creator of the link. (See Section 14.3.4, "COPY.LINK", for diagrams and contents of notification message.) If the "inform on pass" bit is set, and the owner of the link is invoking PASS.LINK, no notification message is sent.

Implementation								Addressing mode usage
9	0	5	x	y	z	t	v	PASS.LINK Rx,[Ry],Rz,[Rt],Rv

Opcode

Ry 1st parameter block address

Rz 1st message block data length

Rt 2nd message block address

Rv 2nd message block data length

Rx = Instruction Specific Return Status

0 =	Success	[y]
5 =	Transport Hardware Error On Notification	[n]
6 =	Message Too Long	[n]
36 =	Link 1 Receiving Process Dead	[n]
40 =	Link 1 Link Not Defined	[n]
41 =	Link 1 Unallocated Page	[n]
42 =	Link 1 Unallocatable Page	[n]
43 =	Link 1 Exceeds No Of Link Levels	[n]
45 =	Link 1 Zero Link ID	[n]
46 =	Negative Data Block Length	[n]
47 =	Link 1 Funnel Not Enabled	[n]
49 =	Link 1 Too Many Attached Buffers	[n]
51 =	Link Cannot Be Sent	[n]
52 =	Link 1 Target Unit Busy	[n]
54 =	Transport Hardware Error On Data Message	[n]
136 =	Link 2 Receiving Process Dead	[n]
139 =	Link 2 Too Many Attached Buffers	[n]
140 =	Link 2 Link Not Defined	[n]
141 =	Link 2 Unallocated Page	[n]
142 =	Link 2 Unallocatable Page	[n]
143 =	Link 2 Exceeds No Of Link Levels	[n]
145 =	Link 2 Zero Link ID	[n]
147 =	Link 2 Funnel Not Enabled	[n]
152 =	Link 2 Target Unit Busy	[n]

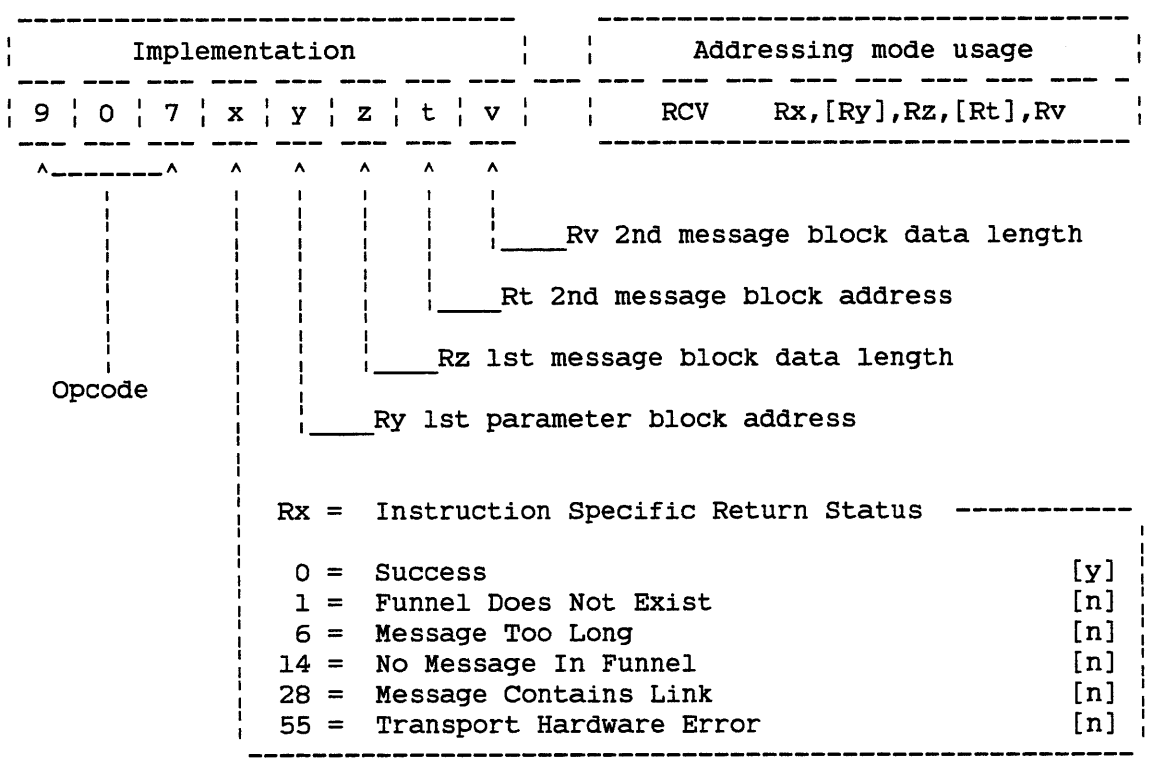
14.4 RECEIVE MESSAGES

The receive instructions provide several means of interfacing to the system message buffers.

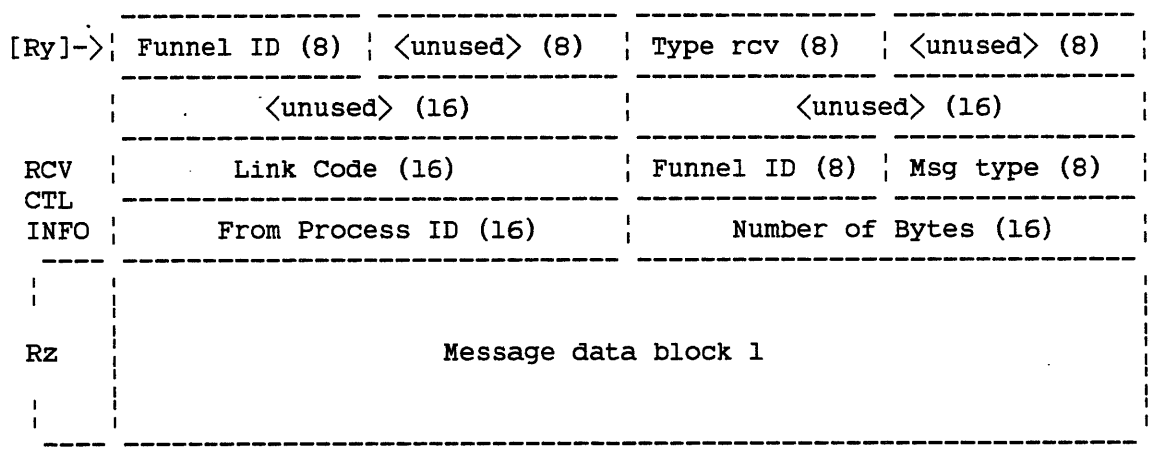
Observe that the microcode adds the length of the parameter block to the number of bytes specified in Rz to determine the address space for the parameter block and the first message block. The parameter block will always be stored at [Ry]. The message itself may be stored at [Ry + 4 words] with Rz length, or stored at [Rt] with Rv length, or split up with the first part of the message at the former address, and the second part at the latter address. The "number of bytes" specified in the receive control part of the parameter block only includes the length of the message data areas, not the parameter block. The user would typically allocate 888 bytes + length of parameter block (16 bytes) to cover worst cases.

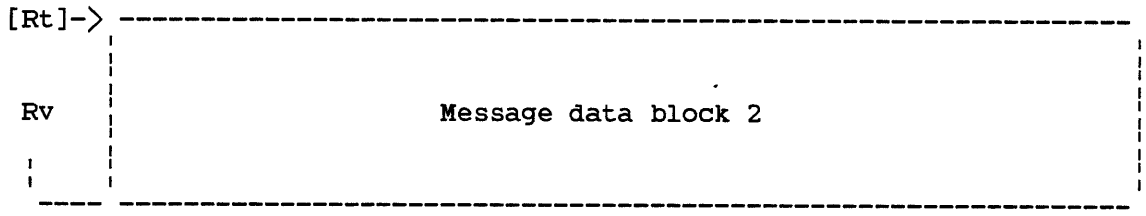
14.4.1 RCV Receive Message

Copies the first message on the specified funnel into the user's virtual address specified in Ry, for the number of bytes specified in Rz. Optionally, if [Rv] <> 0, then the message is copied into two buffers. If so, then Rt contains the virtual address of the beginning of the second buffer and Rv specifies the length of the buffer. Upon completion, status is returned in Rx.



RCV Parameter Block





The parameter fields are defined as follows:

Funnel ID
This is the funnel from which to take the message if there is a message.

Type of receive
This specifies the kind of receive the user wishes to execute and contains the following boolean values:

Synchronous
If there is no message in the specified funnel, the user is blocked until a message is delivered into the funnel. (Bit-7 of field).

Interrogate
The message, if any, is copied into the users virtual address space, but the message remains attached to the funnel. If there is no message, the action is controlled by the synchronous specification. (Bit-6 of field).

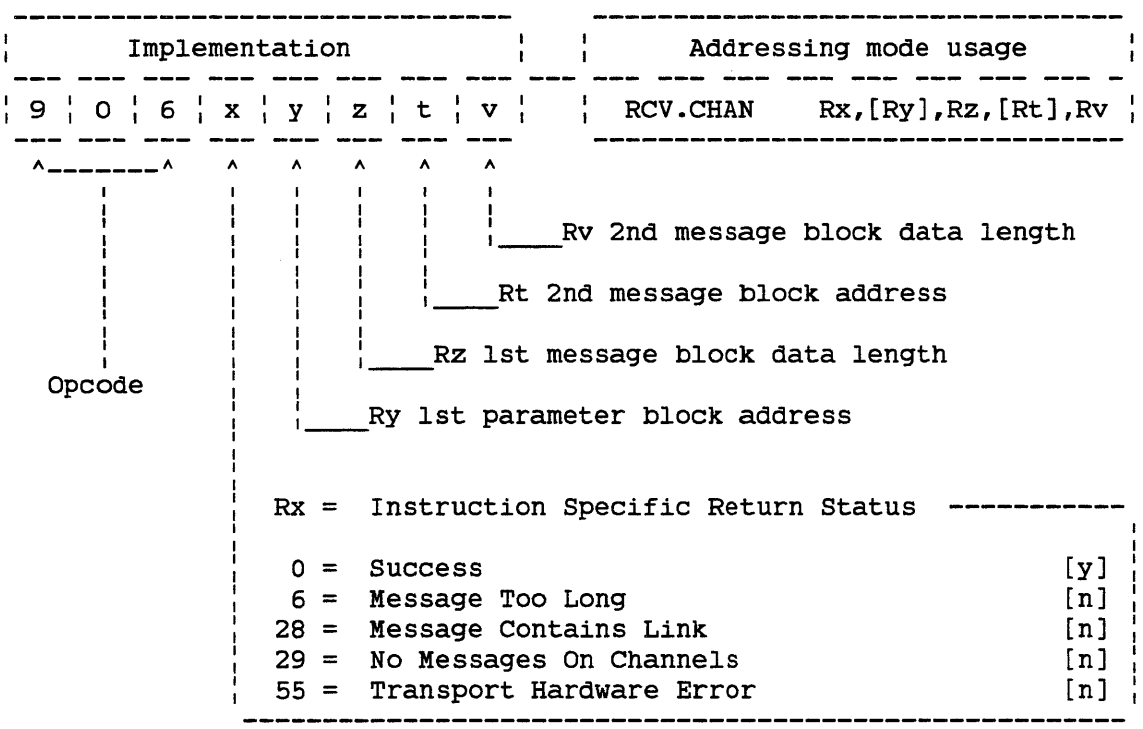
Dismiss
This option implicitly sets the local priority to a value previously specified by the SET.LOCAL.PRI instruction, or to 15 before any modification of local priority by the user. The specification will be invoked prior to the receive. (Bit-5 of field).

If the user executes a RCV and the system detects that the message contains a link, then RCV behaves exactly as for a RCV with interrogate, and returns "message contains link" error (28). If the user wishes to receive the message, the RECEIVE.LINK instruction must be invoked.

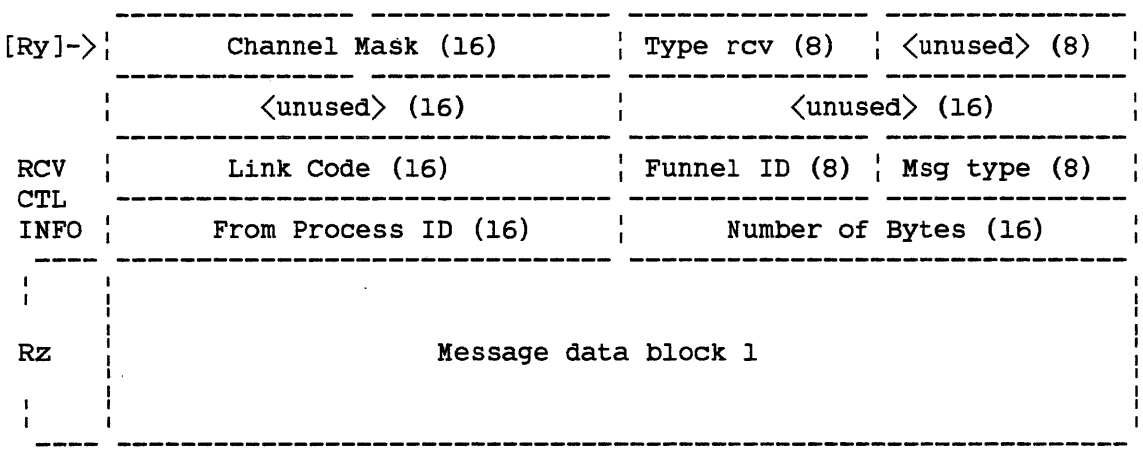
14.4.2 RCV.CHAN Receive Message on Channel

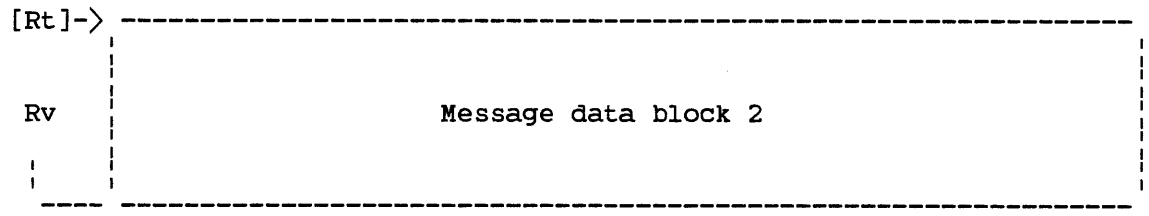
This instruction searches for a message on all channels specified in a channel mask, starting with the highest priority channel. If a message is found, the instruction transfers the message in a manner identical to the RCV instruction. If a message is not found on any specified channel, the action is controlled by the "synchronous" parameter, as follows: If "synchronous" is selected, the instruction will block the user and continue to search the masked channels until a message arrives; if false, the instruction terminates.

The order in which funnels are checked for messages on a given channel is not defined for the user (messages arriving on a funnel are received in FIFO order). The channel mask is specified in the parameter block. The instruction otherwise behaves identically to the RCV instruction on the receipt of a message and for the "type receive" specification.



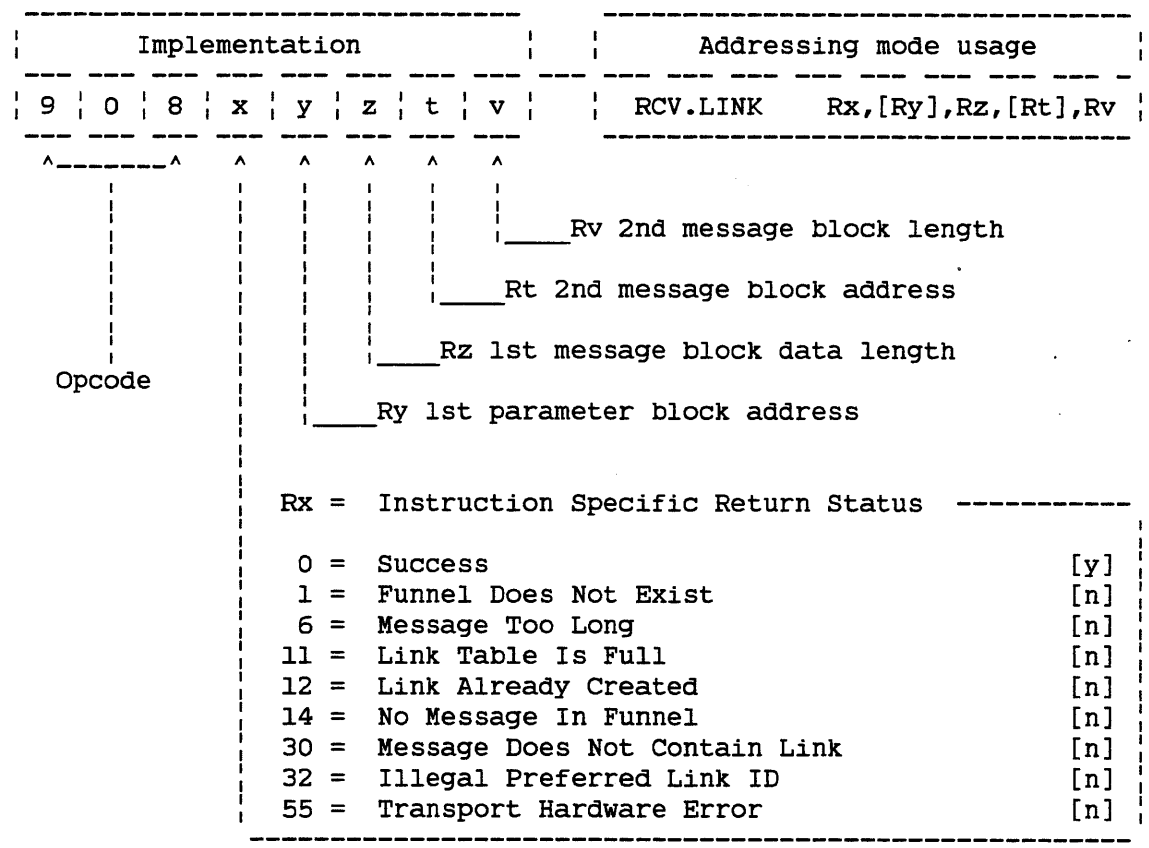
RCV.CHAN Parameter Block



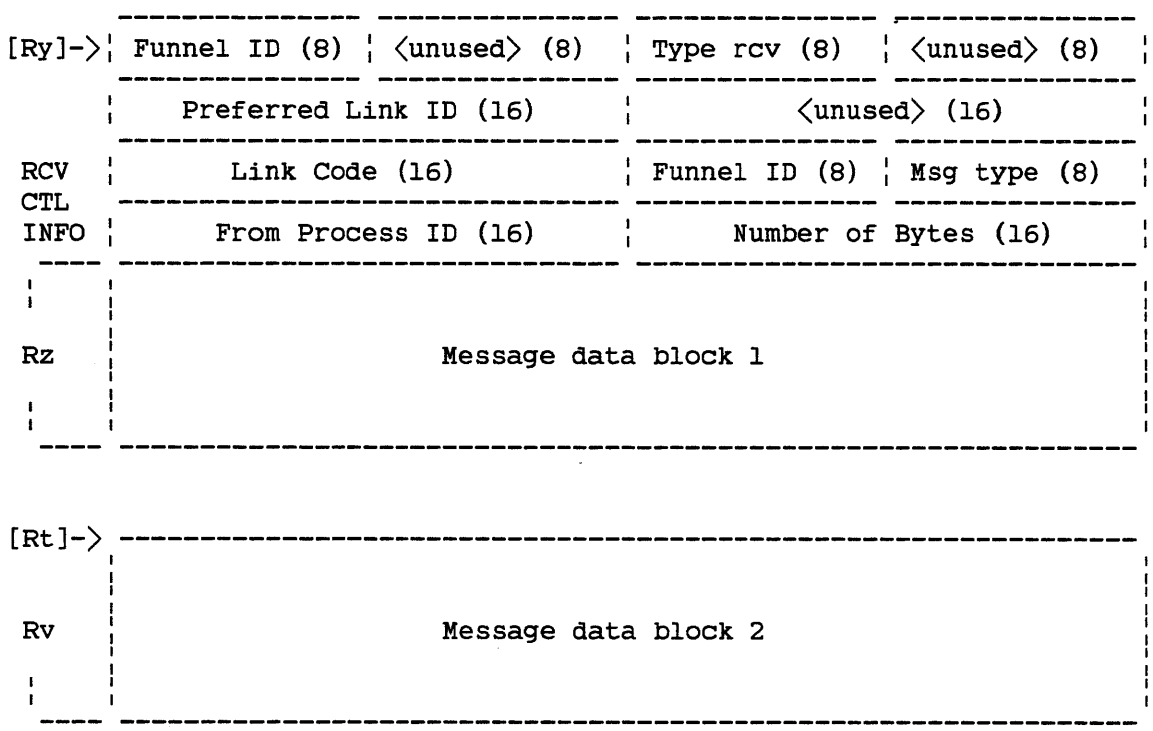


14.4.3 RCV.LINK Receive Message with Link

This instruction is similiar to RCV in the copy of the message into the user's virtual address space. In addition, there is a link table entry in the message, which is put into the user's link table according to the "preferred link ID" specification. As with RCV, Ry contains the virtual address of the parameter block and Rz the length of the message block. If [Rv] <> 0, then the message is copied in two parts, with Rt specifying the virtual address of the second block and Rv the length. Upon completion, status is returned in Rx.



RCV.LINK Parameter Block

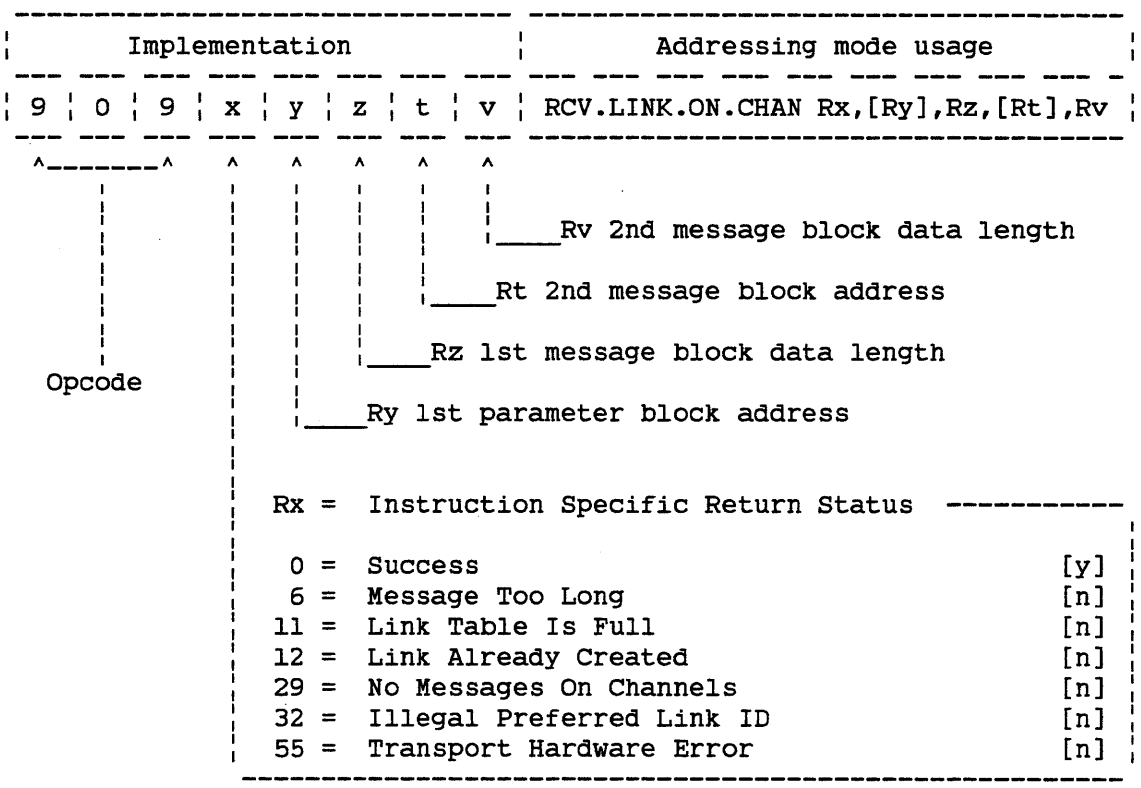


The funnel, receive type and receive control information are the same as for RCV. Preferred link ID is the link ID in the receiver's link table into which the link entry is placed. If the user has no preference, a zero (0) may be specified and the machine will use the next free link in the table. In either case, the machine returns the actual link ID assigned and places it in the Preferred Link ID field of the parameter block.

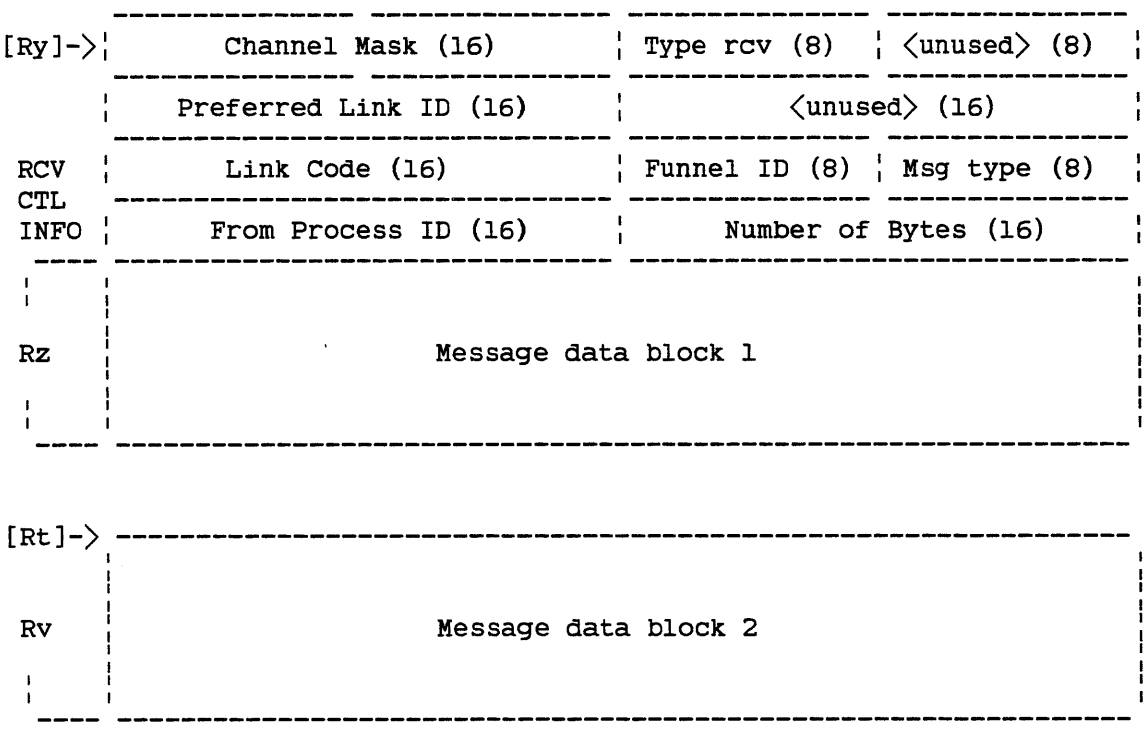
If the user executes a RCV.LINK and the machine detects that the message does not contain a link, the message is transferred exactly as for a RCV, and the "Message Does Not Contain Link" error is returned to the user. If status codes 11, 12, or 32 are returned, the message is copied into the user's address space but the message is not deleted from the funnel and the preferred link is not created.

14.4.4 RCV.LINK.ON.CHAN Receive Message with Link on Channel

This instruction behaves exactly like RCV.LINK, except that a channel mask is specified. (See RCV and RCV.CHAN above.)



RCV.LINK.ON.CHAN Parameter Block



14.5 FORWARD MESSAGES

These instructions forward messages from a funnel and send them out on a link. Links cannot be forwarded from a message sender, rather the EXCH.LINK.FORWARD instruction exchanges the link with one held by the intermediate process, and forwards the new link with the original message.

14.5.1 FORWARD.MSG Forward Message

Take the first message from the funnel specified in the parameter block and send it on the specified link. Ry contains the address of the parameter block. Upon completion, status is returned in Rx. Messages containing links may not be forwarded.

Implementation						Addressing mode usage
8	0	A	x	y	0	FORWARD.MSG Rx,[Ry]

Opcode

y: This field must be zero

0: Ry parameter block address

Rx = Instruction Specific Return Status

0 =	Success	[y]
1 =	Funnel Does Not Exist	[n]
14 =	No Message In Funnel	[n]
23 =	Cannot Forward On Link	[n]
28 =	Message Contains Link	[n]
36 =	Link 1 Receiving Process Dead	[n]
39 =	Link 1 Too Many Attached Buffers	[n]
40 =	Link 1 Link Not Defined	[n]
41 =	Link 1 Unallocated Page	[n]
42 =	Link 1 Unallocatable Page	[n]
43 =	Link 1 Exceeds No Of Link Levels	[n]
45 =	Link 1 Zero Link ID	[n]
47 =	Link 1 Funnel Not Enabled	[n]
55 =	Transport Hardware Error	[n]

FORWARD.MSG Parameter Block

[Ry]->				
	Funnel ID (8)	<unused> (8)	Link ID (8)	<unused> (8)
	<unused> (32)			

14.5.2 EXCH.LINK.FORWARD Exchange Message Link and Forward Message

The instruction takes the first message on the funnel specified by funnel ID, and forwards it on the link ID specified in the parameter block. If the message contains a link, then the link ID contained in the message is saved in "preferred link ID", and "link to be sent" is substituted instead. Upon completion, status is returned in Rx. Ry contains the address of the parameter block.

Implementation							Addressing mode usage								
8	0	9	x	y	0		EXCH.LINK.FORWARD							Rx,[Ry]	

Opcode

This field must be zero

Ry parameter block address

Rx = Instruction Specific Return Status

0 =	Success	[y]
1 =	Funnel Does Not Exist	[n]
11 =	Link Table Is Full	[n]
12 =	Link Already Created	[n]
14 =	No Message In Funnel	[n]
23 =	Cannot Forward On Link	[n]
32 =	Illegal Preferred Link ID	[n]
36 =	Link 1 Receiving Process Dead	[n]
39 =	Link 1 Too Many Attached Buffers	[n]
40 =	Link 1 Link Not Defined	[n]
41 =	Link 1 Unallocated Page	[n]
42 =	Link 1 Unallocatable Page	[n]
43 =	Link 1 Exceeds No Of Link Levels	[n]
45 =	Link 1 Zero Link ID	[n]
47 =	Link 1 Funnel Not Enabled	[n]
51 =	Link Cannot Be Sent	[n]
55 =	Transport Hardware Error	[n]
140 =	Link 2 Link Not Defined	[n]
141 =	Link 2 Unallocated Page	[n]
142 =	Link 2 Unallocatable Page	[n]
143 =	Link 2 Exceeds No Of Link Levels	[n]
145 =	Link 2 Zero Link ID	[n]

EXCH.LINK.FORWARD Parameter Block

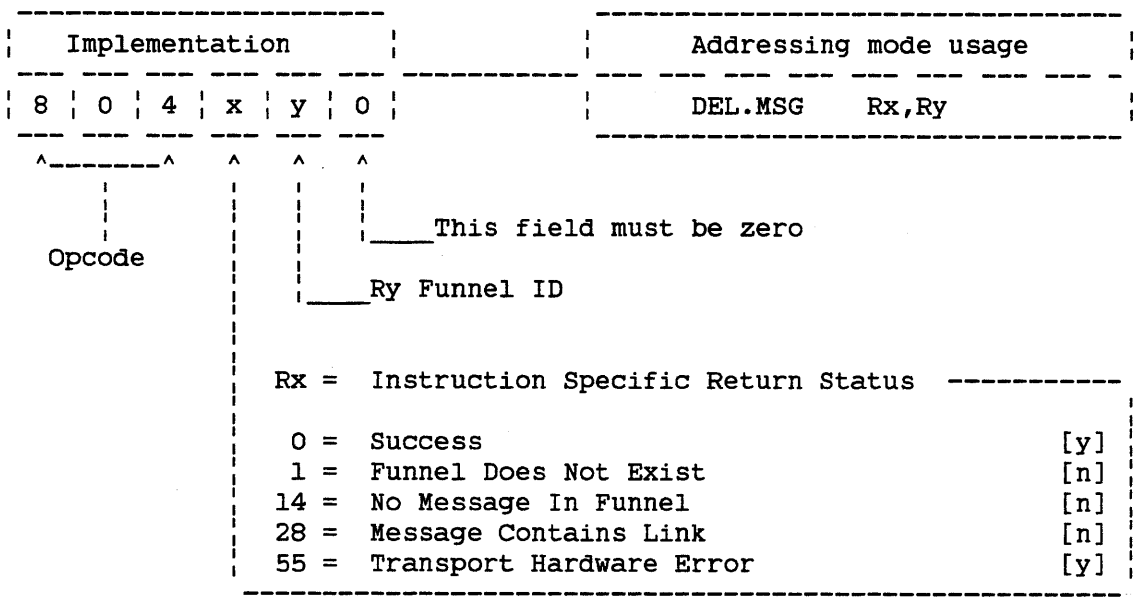
[Ry]->	Funnel ID (8)	<unused> (8)	Link ID (16)
	Preferred Link ID (16)		Link to be sent (16)

14.6 DELETE MESSAGES

This instruction disposes of messages.

14.6.1 DEL.MSG Delete Message from Funnel

Deletes the first message attached to the funnel specified in Ry. Upon completion, status is returned in Rx, and the message buffers holding the deleted message are returned to the free list. Messages containing links may not be deleted.

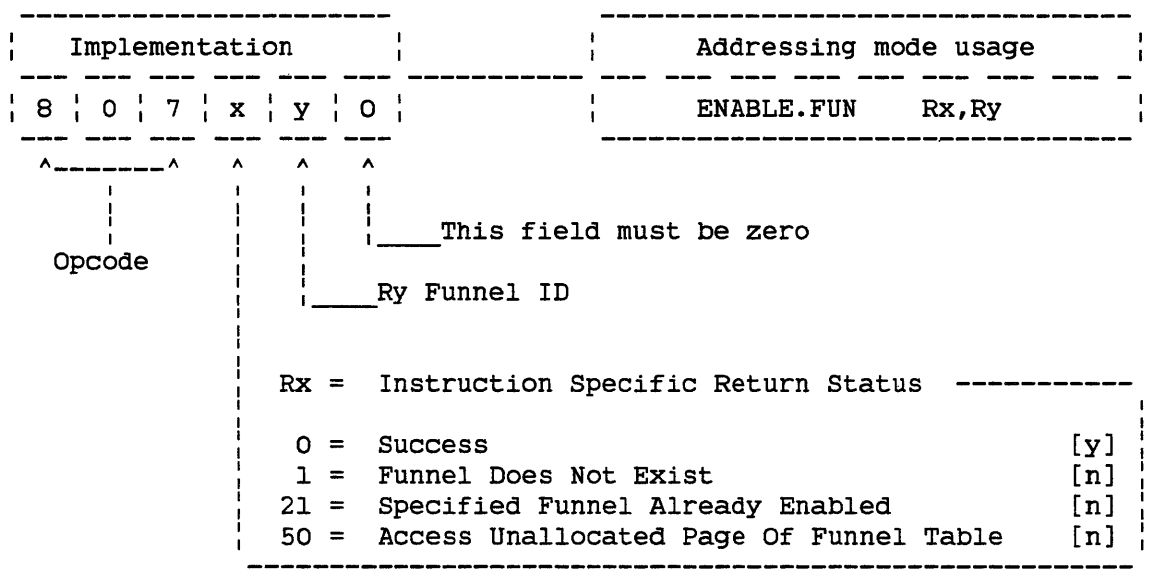


14.7 ENABLE AND DISABLE FUNNEL

These instructions toggle the enabled funnels bit in the Enabled Funnels Map of the Process Control Block. This has the effect of marking the associated funnel as enabled or disabled. The Enabled Funnels Map contains an enabled funnels bit for each of a process' 256 funnels. Chapter 13, "Inter-Process Communications", covers this data structure in detail.

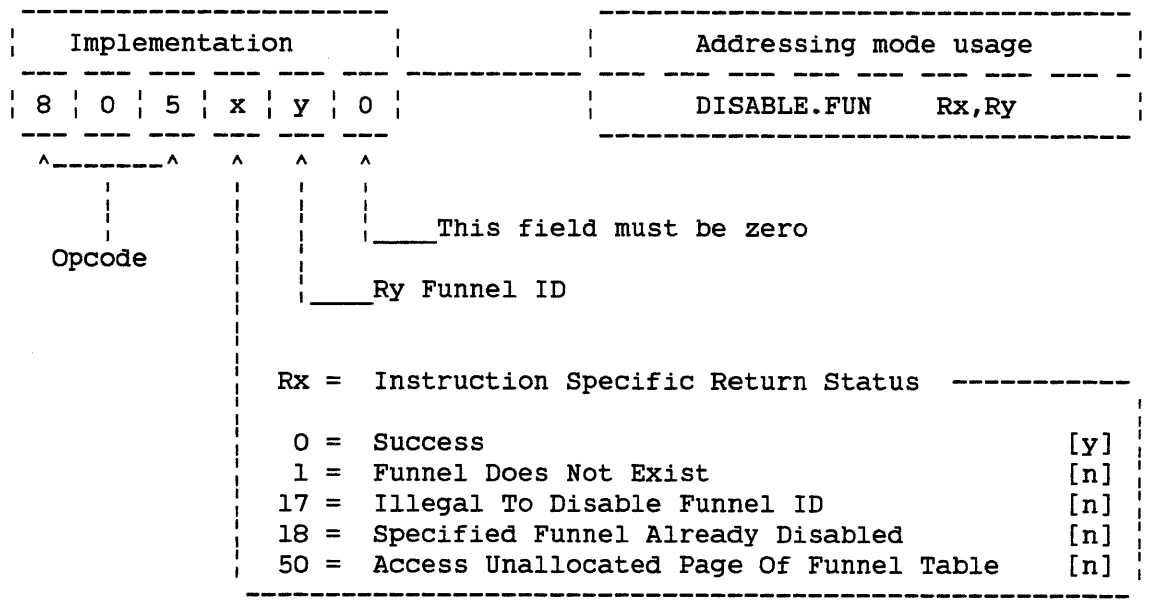
14.7.1 ENABLE.FUN Enable Funnel

The funnel specified in Ry is marked enabled in the enabled funnel map. Any subsequent messages directed to the funnel will be delivered. Status is returned in Rx.



14.7.2 DISABLE.FUN Disable Funnel

The funnel specified in Ry is marked disabled in the enabled funnels map. All messages currently attached to the funnel remain attached, but subsequent messages directed to the funnel will not be attached and bad (non-zero) status will be returned to the sender of the message. Upon completion of the DISABLE.FUN instruction, status is returned in Rx.



14.8 INTERRUPT AND LOCAL PRIORITY

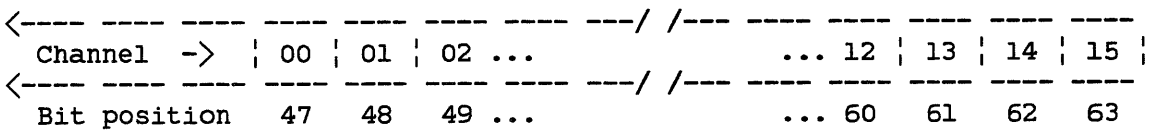
If a channel is enabled for interrupts, any message arriving on a funnel attached to the channel will generate an interrupt. Assuming that no higher priority interrupts are extant, the instruction stream of the process will be interrupted when the firmware loads the Program Counter with the interrupt vector, that is, the address of the trap handler. The trap handling procedure interprets the parameters passed by the message that generated the interrupt. These mechanisms are discussed in Chapter 2, "Architecture".

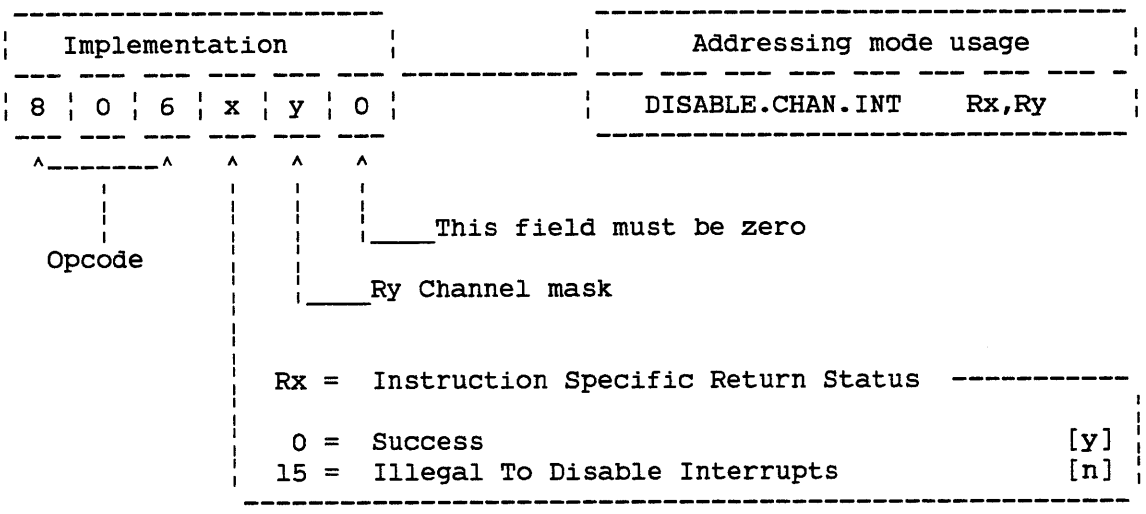
These instructions allow the process to control local priority, the interrupt channel masks, and the interrupt vectors.

14.8.1 DISABLE.CHAN.INT Disable Interrupts on Channel

The channel(s) specified in the channel mask are marked to disable interrupts by setting the associated bit(s) in Ry. Thereafter, messages delivered to funnels attached to such channels will not cause an interrupt. Upon completion, the old channel mask is returned in Ry and status is returned in Rx. If the return status is non-zero, Ry is unchanged. Interrupts may be disabled on any channel except channel 0.

Channel mask in Ry, right justified

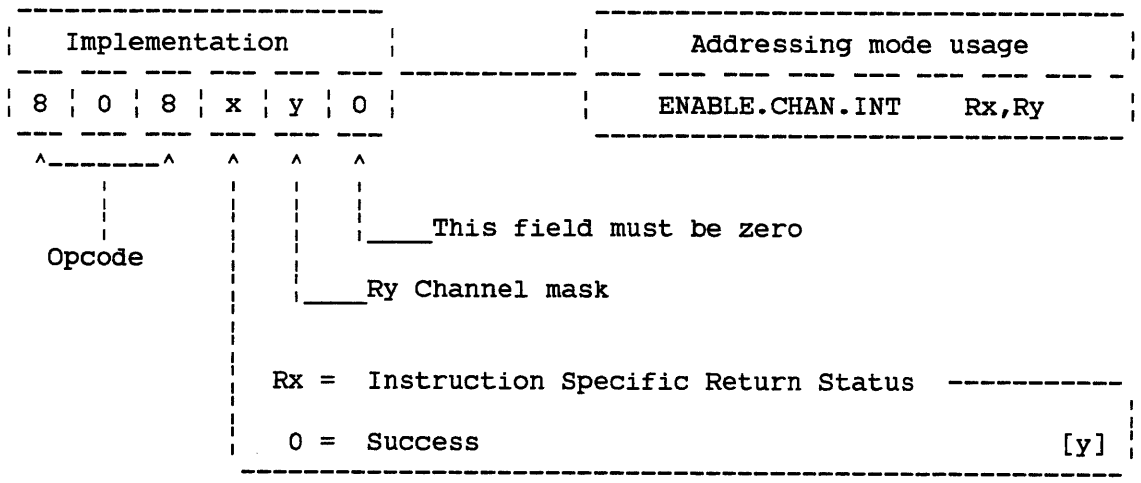




The channel mask is represented in the machine with bits set for enabled channels, and bits cleared for disabled channels. The value returned in Ry for the old channel mask will be displayed accordingly.

14.8.2 ENABLE.CHAN.INT Enable Interrupts on Channel

The channel(s) specified in the channel mask are marked to enable interrupts by setting the associated bit(s) in Ry. Existing messages on the funnels for the enabled channel will cause interrupts as well as the delivery of subsequent messages. On completion, the old channel mask is returned in Ry and status is returned in Rx.

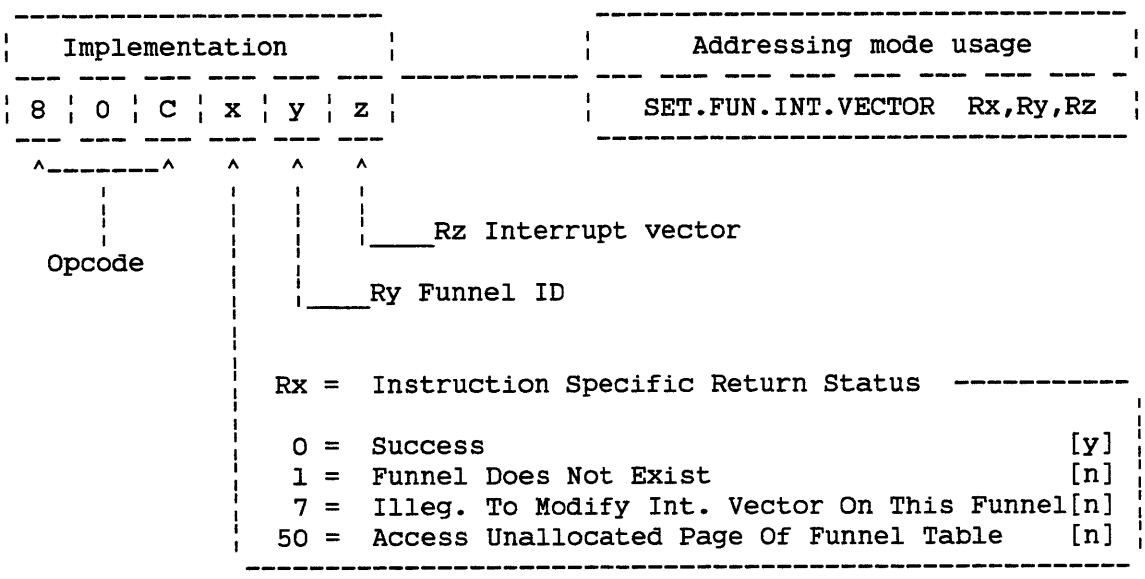


The channel mask is represented in the machine with bits set for enabled channels, and bits cleared for disabled channels. The value returned in Ry for the old channel mask will be displayed accordingly.

14.8.3 SET.FUN.INT.VECTOR Set Funnel Interrupt Vector

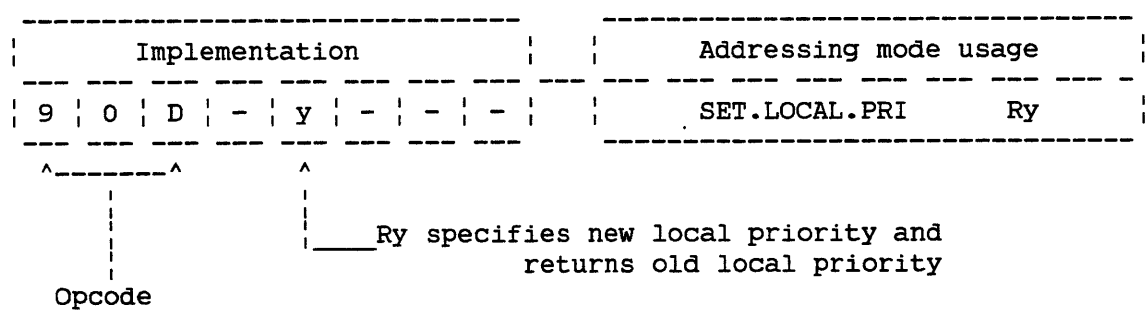
The interrupt vector for the funnel specified in Ry is changed to the value specified in Rz. Upon completion, the old interrupt vector is returned in Rz, and status is returned in Rx. If a non-zero status is returned in Rx, then Rz is unchanged. The interrupt vector for funnel 1 cannot be modified with this instruction.

All vectors point to the system interrupt handlers at process start-up. A value in Rz of zero is illegal, as it will cause an access violation to page zero if an interrupt should occur that uses this vector.



14.8.4 SET.LOCAL.PRI Set Local Priority

The local priority of the caller is modified to either the priority in Ry or the priority of the highest active channel, whichever is higher. An active channel is any channel having a funnel that contains a message. On completion, the old local priority is returned in Ry. No status code is returned with this instruction.

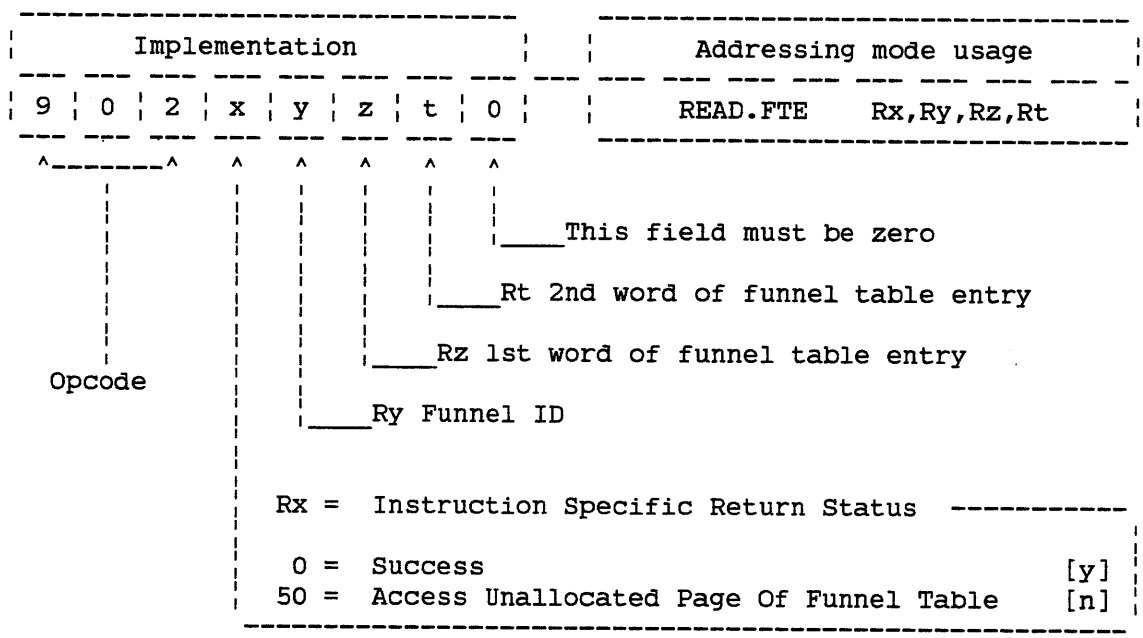


14.9 PROCESS INQUIRY

These instructions read the link and funnel table entries. If the input parameter is zero, the associated link or funnel table header is returned.

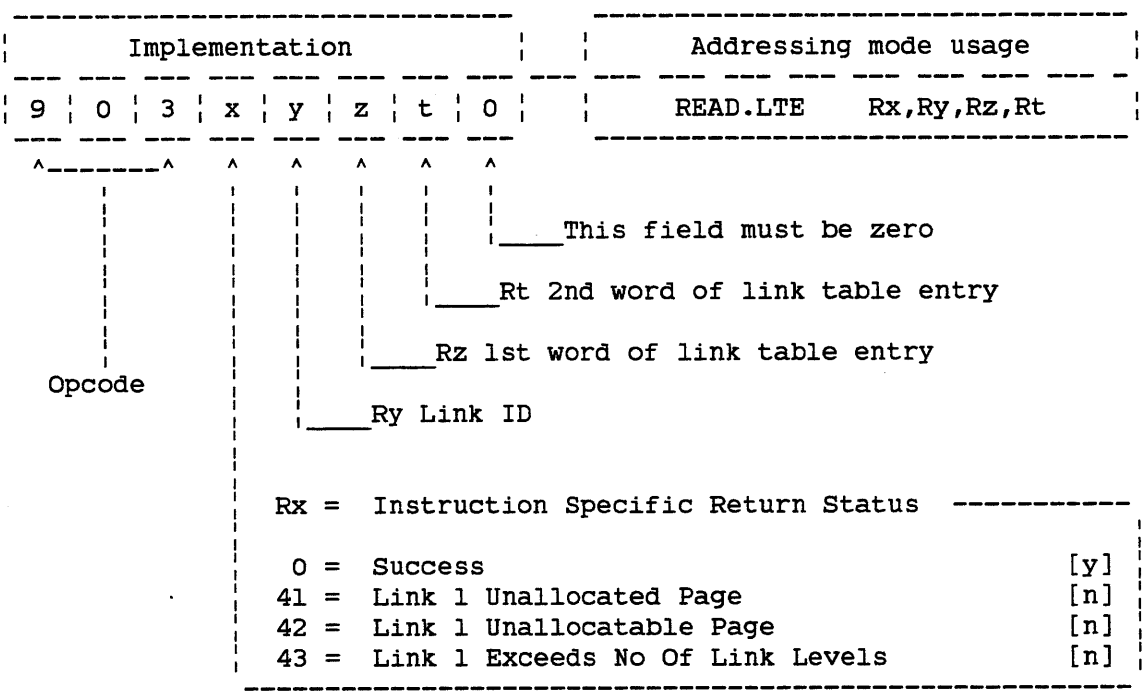
14.9.1 READ.FTE Read Funnel Table Entry

The funnel table entry for the funnel specified in Ry is returned in registers Rz and Rt. If zero is specified, then the funnel table header is returned. Rz holds the first word of the funnel table entry, and Rt holds the second word of the funnel table entry. The status is returned in Rx. Upon completion, if [Rx] is not equal to zero, then the contents of Rz and Rt have no meaning.



14.9.2 READ.LTE Read Link Table Entry

The link table entry for the link specified in Ry is returned in registers Rz and Rt. If zero is specified, then the link table header is returned. Rz holds the first word of the link table entry, and Rt holds the second word of the link table entry. The status is returned in Rx. Upon completion, if [Rx] is not equal to zero, then the contents of Rz and Rt have no meaning.



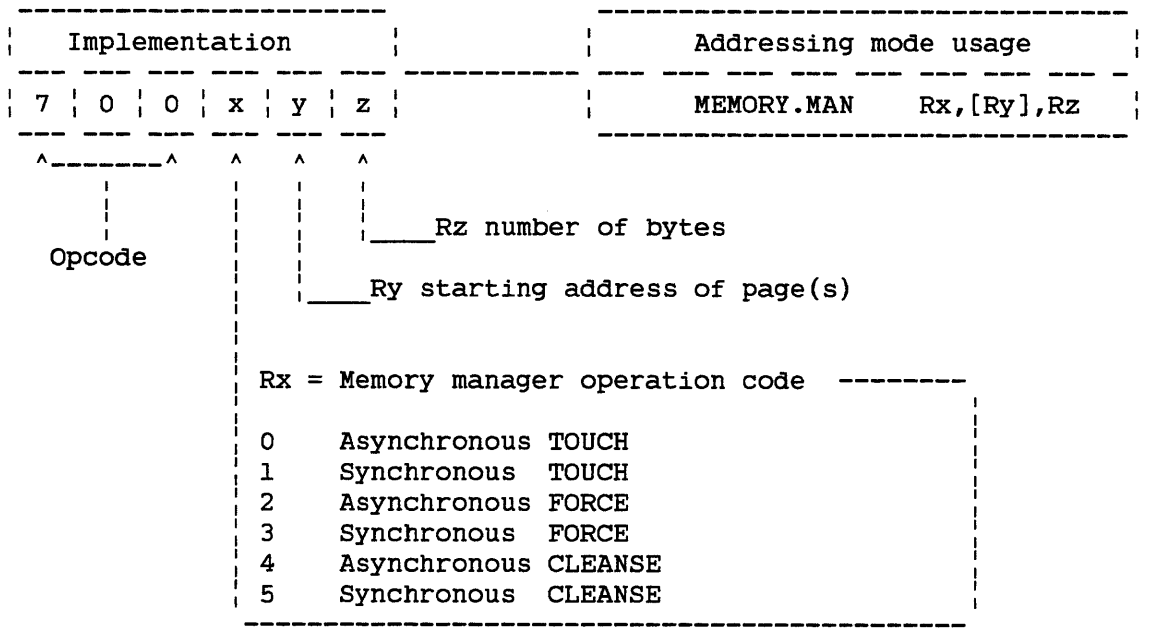
15 GENERAL INSTRUCTIONS

This chapter covers the miscellaneous instructions:

MEMORY.MAN	Memory Management
MODIFY.PME	Modify Page Map Entry
READ.PME	Read Page Map Entry
NOP	No Operation
READ.CPU.TIMER	Read CPU Timer
READ.REAL.TIMER	Read Real Time Clock
READ.STAT	Read Process Status Word
WRITE.STAT	Write Process Status Word
READ.MACH.ID	

15.1 MEMORY.MAN - MEMORY MANAGEMENT

This instruction sends a message to the Memory Manager Process to request an operation on a page or group of pages. The request may be either synchronous or asynchronous. If synchronous, then the process executing this instruction will be stopped until the Memory Manager explicitly restarts it. Rx contains the operation code to be performed by the memory manager, Ry contains the starting address of the virtual page(s), and Rz contains the number of bytes in the page(s).



TOUCH is an asynchronous hint to start a pre-fetch on a page or a group of pages.

FORCE is a request to write a page or a group of pages to disk. The reference bit is not modified. If the page(s) are not dirty, then they are not written.

CLEANSE is a request to reset a page or group of pages to "virgin". This resets the page map entry (PME) for every virtual page having an actual page of disk space. When the page is subsequently referenced, the Memory Manager will not fault the page in off of disk, but instead will allocate a zeroed page in physical main memory. The copy on disk will not be initialized to zero as is normally the case when a page is allocated. This is typically used to clean pages that have not been referenced and do not need to be written to disk.

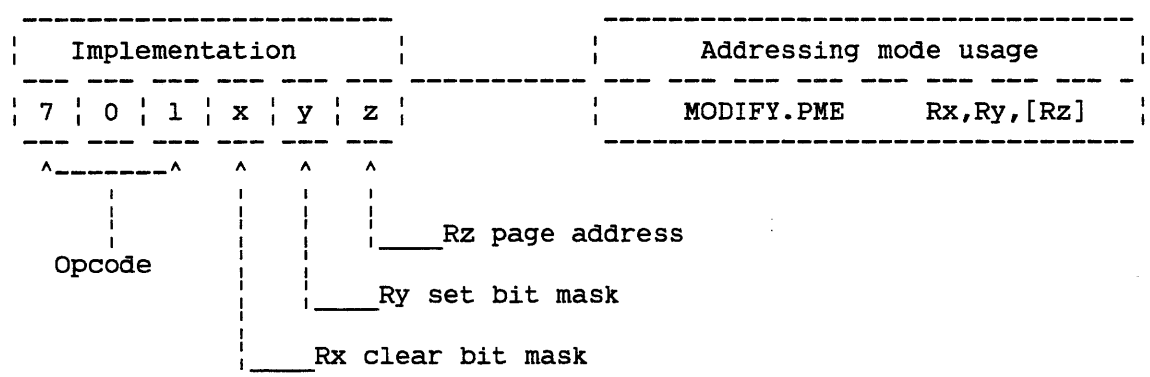
15.2 MODIFY.PME MODIFY PAGE MAP ENTRY

Rx contains the clear bit mask and Ry contains the set bit mask to modify the PME bits. Rz contains the address of the page belonging to the PME. In the bit masks, a bit having the value one indicates that the appropriate operation is to be performed on the corresponding bit in the PME. A bit having the value zero indicates that the corresponding bit in the PME is not to be modified.

The instruction operates as follows: [Rx] and [Ry] are first ANDed with internally maintained masks to restrict the PME bits that the user may modify. The clear mask is applied first. To clear bits, the verified [Rx] is complemented and ANDed with the PME. This result is then ORed with the verified [Ry]. The PME is then rewritten. The only

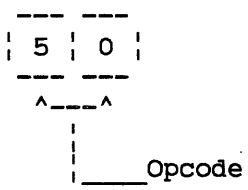
bits which may be modified are the three trace bits and the reference bit. The instruction operates correctly on shared pages.

PME bits	type	mask bit
maintenanceBit	boolean	60
readTraceBit	boolean	61
writeTraceBit	boolean	62
executeTraceBit	boolean	63



15.3 NOP NO OPERATION

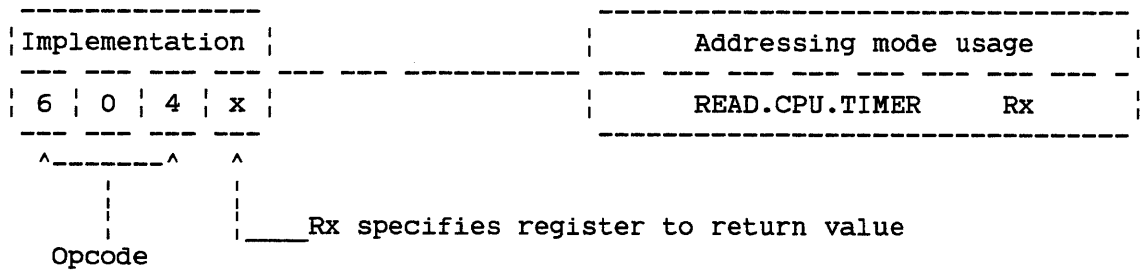
The NOP instruction has no action other than to cause the PC to be incremented to the next instruction.



Instruction Specific Exceptions: none

15.4 READ.CPU.TIMER

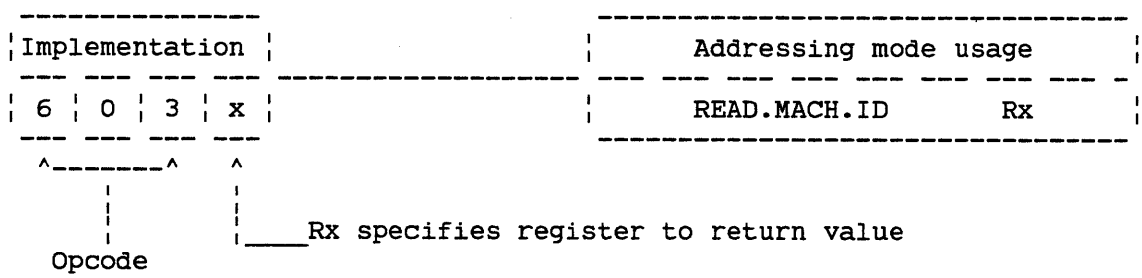
Load the specified register with the 64-bit Process CPU Timer.



Instruction Specific Exceptions: none
Each process has a 64-bit CPU Timer which is initialized at process start-up to zero. The timer is incremented by one every 25 nanoseconds when the process is executing.

15.5 READ.MACH.ID

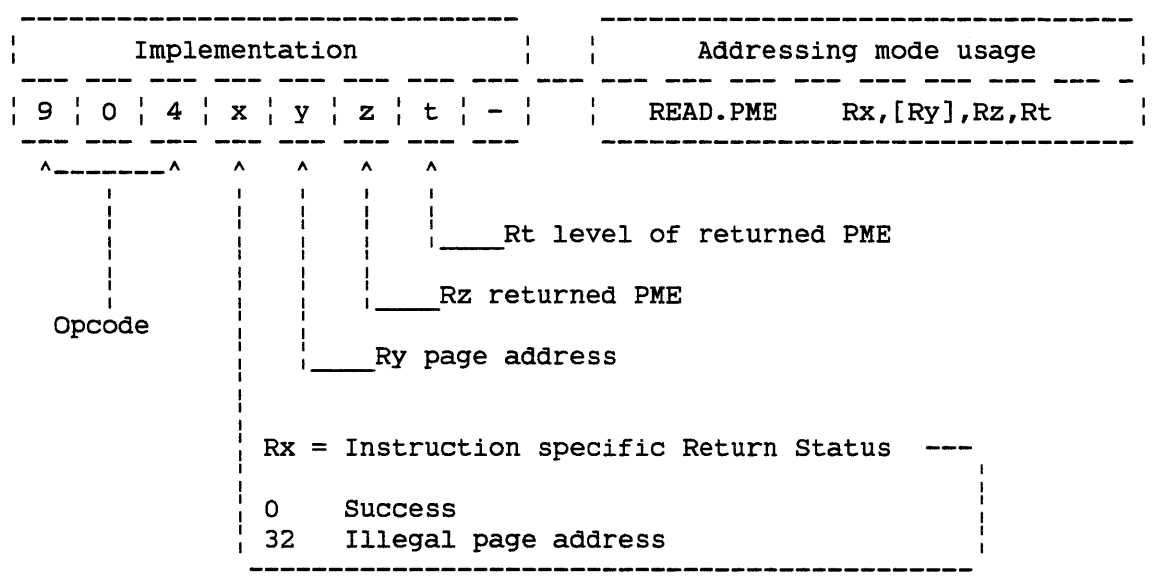
Load the specified register with the 64-bit CPU identification code. This code is unique for each CPU and is assigned when manufactured. The contents of the identification code are currently undefined.



Instruction Specific Exceptions: none

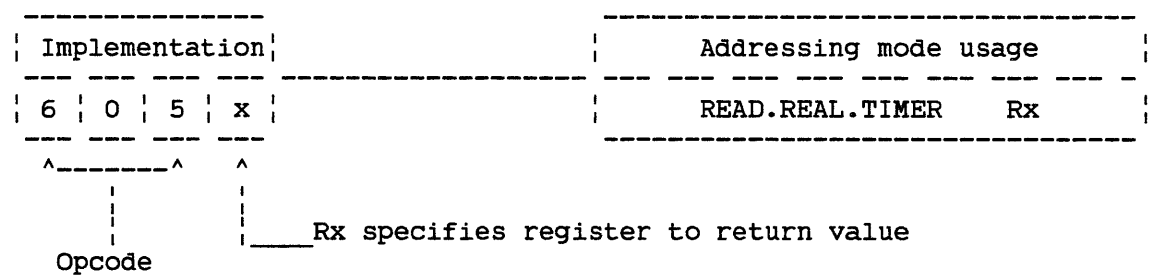
15.6 READ.PME READ PAGE MAP ENTRY

The page map entry for the page containing the byte address in Ry is returned in Rz. The PME level is returned in Rt. A page map entry is always returned, regardless of whether or not the address in Rz is allocatable or not. The status is returned in Rx. If the return value of [Rx] is non-zero, then Rz is not modified.



15.7 READ.REAL.TIMER

Load the specified register with the 64-bit Absolute Timer.



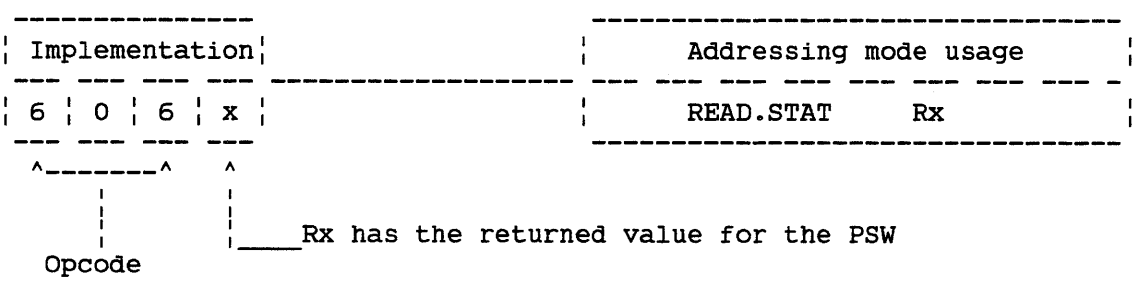
The Service Processor contains a battery powered clock which measures time and date. At cold load of the system, the time and date are converted to a value which represents the absolute number of 25 nanosecond ticks which have occurred since a base time and date. This value is placed into the Real Time Clock at CPU initialization. The Real Time Clock in incremented by 1 every 25 nanoseconds. Since the

clock is 64 bits wide, the clock can time an interval of over 7,200 years (not counting the sign bit).

The base time is March 1, 1600.

15.8 READ.STAT READ PROCESS STATUS WORD

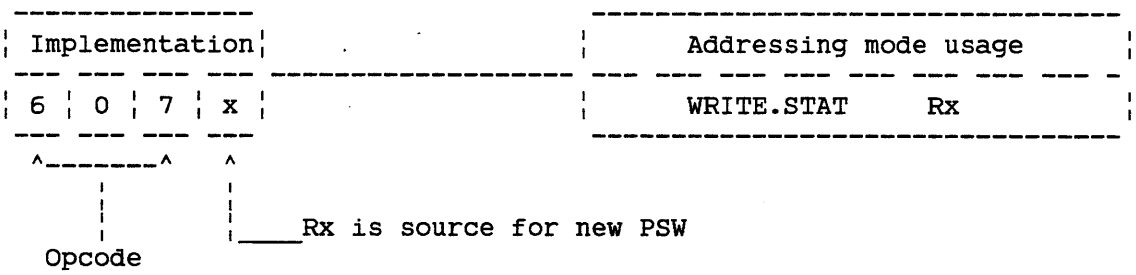
Load the specified register with the 64-bit Process Status Word.



Instruction Specific Exceptions: none

15.9 WRITE.STAT WRITE PROCESS STATUS WORD

Copy the specified register into the 64-bit Process Status Word.

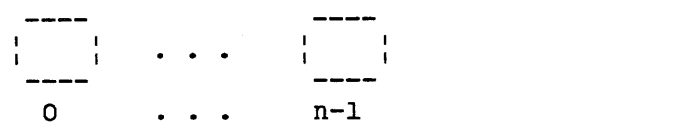


Instruction Specific Exceptions: none

APPENDIX A: ALPHABETICAL LIST OF OPCODES

APPENDIX B: DATA TYPES

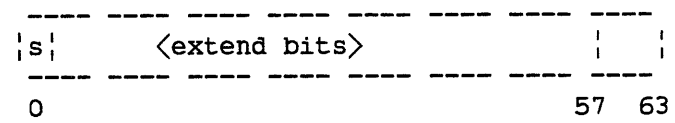
8-bit Character



Character String
1 <= n <= 2**32

8-bit Integer in memory

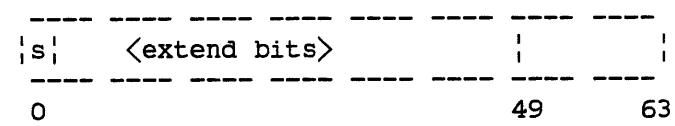
and



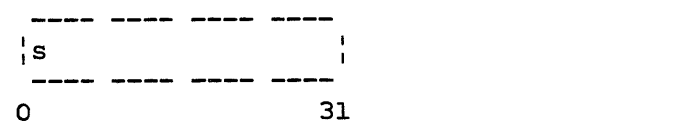
in register

16-bit Integer in memory

and

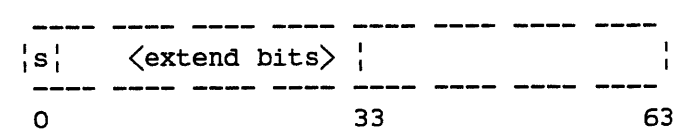


in register

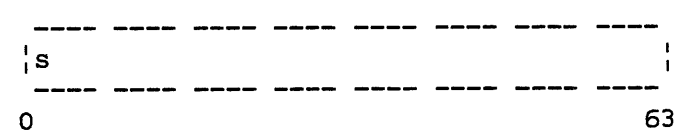


32-bit Integer in memory

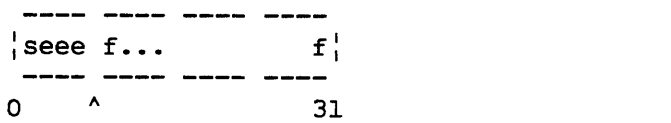
and



in register

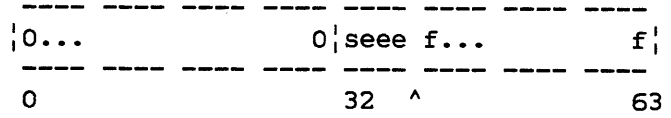


64-bit Integer in memory and register

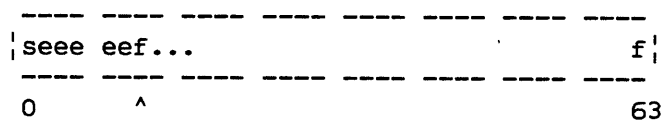


Single Floating in memory

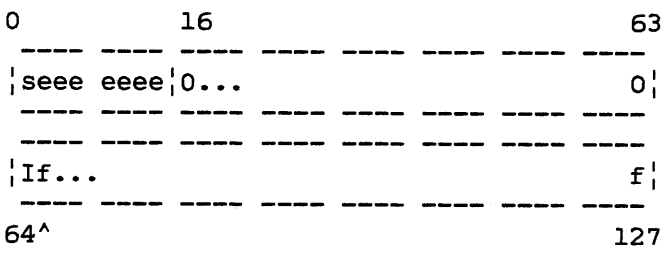
and



in register



Double Floating in memory and register



Double Extended Floating in memory and register

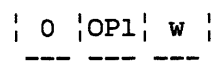
s = sign bit
e = exponent bit(s)
f = fraction (significand) bit
I = Integer bit of significand if no hidden bit
h = apparent position of hidden bit
^ = position of binary point

Numeric string in register

where
x = hex3 or [= 0 if all bits to left or right = 0]
D = hex0 through hex9

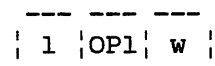
APPENDIX C: GENERALIZED ADDRESSING INSTRUCTION FORMATS

1 reg, reg



op Rx,Rw

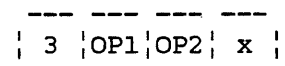
1 reg, reg



op Rx,Rw

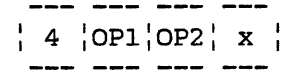
Mode 2: Reserved for future use

2 reg, reg



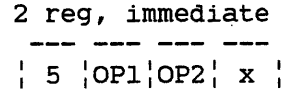
op Rx,Ry,Rz

1 reg, absolute



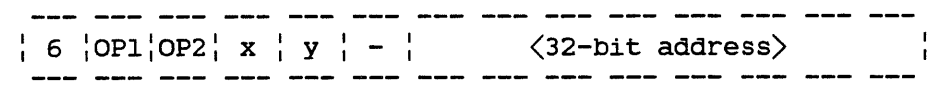
op Rx,e{16}

2 reg, immediate



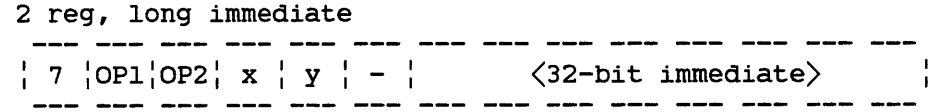
op Rx,Ry,=e{12}

2 reg, long absolute

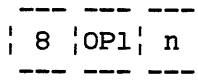


op Rx,Ry,e{32}

2 reg, long immediate



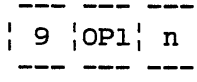
Stack pointer relative



op Rx,[ ]e{6}

where e = 4n, 0 <= n <= 15

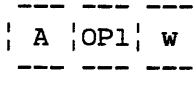
Stack pointer relative



op Rx,[ ]e{6}

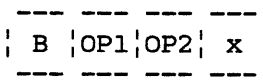
where e = 4n, 0 <= n <= 15

Reg, base + zero displacement



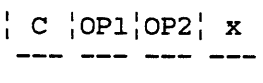
op Rx,[Rw]

2 reg, base + zero displacement



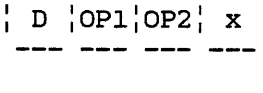
op Rx,Ry,[Rz]

1 reg, base + index + zero displacement



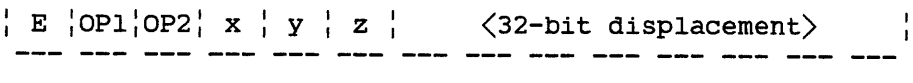
op Rx,[Ry][Rz]

1 reg, base + 12-bit displacement



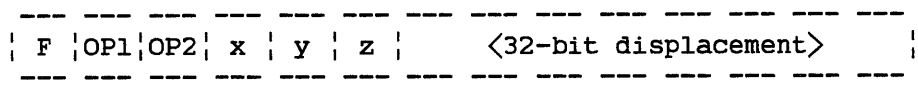
op Rx,[Ry],e{12}

1 reg, base + index + 32-bit displacement



op Rx,[Ry][Rz]e{32}

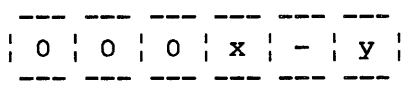
2 reg, base + 32-bit displacement



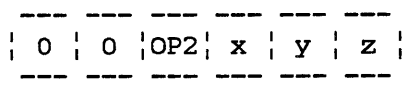

```
                                        op    Rx,Ry,[Rz]e{32}
```

APPENDIX D: NON-GENERALIZED ADDRESSING INSTRUCTION FORMATS

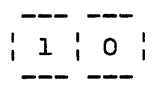
EXCEPTION



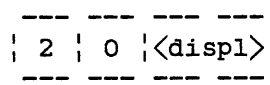
FREM, MUL.128, MULU.128



BREAKPOINT

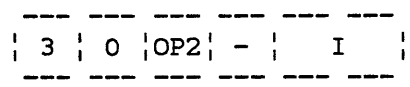


BR.FORWARD



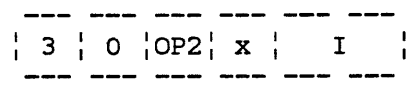
(PC + UNSIGNED <displ>)

EXIT, IXIT, BXIT



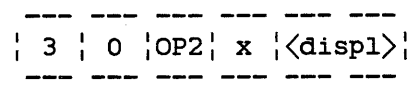
(ADD UNSIGNED I TO GPR 15)

SET.BIT, CLEAR.BIT, TOGGLE.BIT



(BIT I IN GPR x)

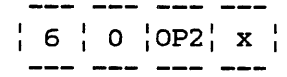
BR.F.<cond>.SH.REL



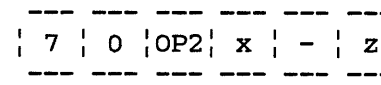
(PC + UNSIGNED <displ>, IF TRUE)

NOP

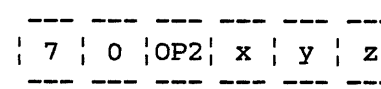
BR.REG, READ.---, WRITE.STAT, CASE



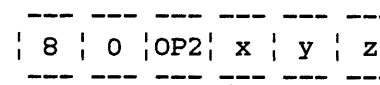
EXCH, EXCH.AND, EXCH.OR, CVT.AI, CVT.IA



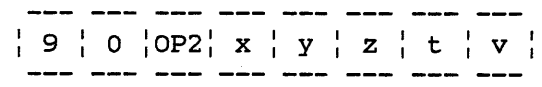
MEMORY.MAN, MODIFY.PME, ASCII.---, COPYB, COPYB.CONST, CMPB.TEST



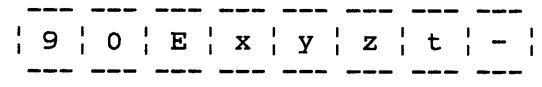
Message System 1



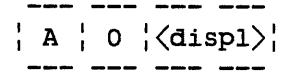
Message System 2 and SET.LOCAL.PRI



ASCII.SUB

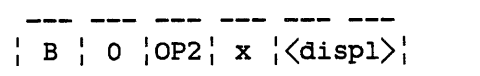


BR.BACKWARD



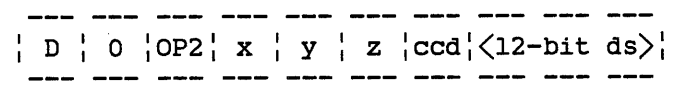
(PC - UNSIGNED <displ>)

BR.B.<cond>.SH.REL

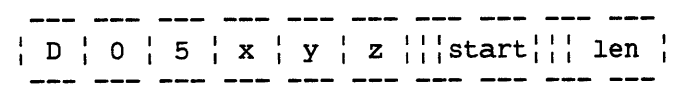


(PC - UNSIGNED <displ>, IF TRUE)

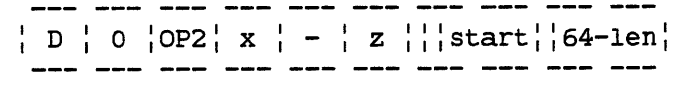
CMPB.BR, CMPB.BR.CONST



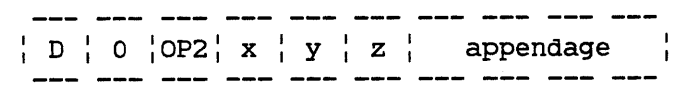
INSERT



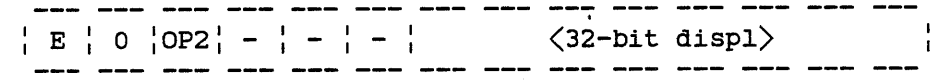
EXTRACT, EXTRACTZ



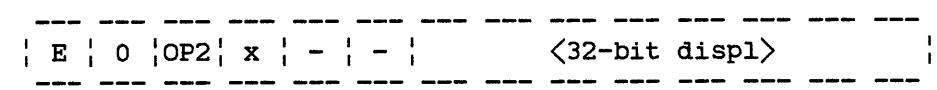
FCMP.BR.80, FCMPX.BR.80, FCMP.80, FCMPX.80



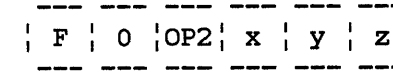
CALL, BR.ABS, BR.REL



CALL.REG, BR.<cond>.ABS, BR.<cond>.REL



CVT.IE, CVT.EI, CVT.ES, Fxxx.80



APPENDIX E: ELXSI MACHINE INSTRUCTION DESCRIPTORS

M — The effective address of an operand.

[M] — The contents of memory location M.

Rspec — Register specifiers reside in the fields tabled below for appropriate instructions.

nibble	2	3	4	5	6	7
register	w	x	y	z	t	v

Rn — The contents of register Rn, where n = 0 to 15.

[Rn] — The contents of register Rn points to a memory location.

<- — The replace descriptor (e.g., [M] <- Rx means "the contents of the memory location pointed at by M is replaced by the contents of Rx").

=Rx — The nibble in the instruction which normally specifies register Rx is interpreted as an immediate operand.

RR — Register to register instruction.

RM — Register to memory or memory to register instruction.

x — In Short Op form, this represents nibble 2 as register Rw. Example: m3x, where m is the addressing mode.

m — Refers to the 1st nibble in the instruction. (nibble 0), For generalized and short opcode instructions, this represents the addressing mode.

OP0 — Refers to the 1st nibble in the instruction (nibble 0), which for non-generalized instructions is the first opcode specifier.

OP1 — Refers to the 2nd nibble in the instruction (nibble 1), which is OPcode specifier 1.

OP2 — Refers to the 3rd nibble in the instruction (nibble 2), which is OPcode specifier 2.

e — Represents an expression which can be evaluated by the assembler and results in an integer value. If an "=" preceeds the "e", then the expression is used as an immediate operand.

{} These delimiters, which may follow an expression, enclose a value that indicates the bit-size of the field which will hold the integer result of the expression. For example, =0{12} would force the usage of the 12-bit immediate mode, while =0{32} would force the 32-bit immediate mode. If this specification is not present, the assembler will use either the 12- or 32-bit mode, depending on the magnitude of the value of the expression.

[] When enclosing a register specification, indicates that the contents of the register are to be used as the address of the operand, rather than as the operand itself. If used without a register specification, the stack pointer (RF) is assumed. This is the manner in which modes 8 and 9 are forced.

, The comma is used to separate operands. Values appearing next to each other in a single operand, if allowed, are summed together to generate the effective address of the operand. Thus, "[R3]1500" would be interpreted as "add 1500 to the contents of register 3 and use this as the effective address of the operand". "[RA][R6]10" would be interpreted as "add 10 to the sum of the contents of register 6, and use this as the effective address of the operand".

[op]:<s:l> The effective operand is the bit string in the operand starting at st and of length len.

Index

ELXSI relies on your observations, comments, and suggestions to ensure the accuracy, readability, and usefulness of its documentation. Please use specific page and paragraph references when appropriate.

What is your overall impression of this manual? Please consider such aspects as accuracy, organization, and writing style.

_____

_____

_____

Does this manual meet your requirements?_____

What did you find most useful?_____

_____

_____

How could it be improved?_____

_____

_____

Please cite any technical errors you have found in this manual._____

_____

_____

Name_____ Street_____

Title_____ City_____

Department_____ State/Country_____

Organization_____ Zip_____

Please return this form to:
ELXSI
Technical Publications
2334 Lundy Place
San Jose, CA 95131