Sept, 1970

GEMINI COMPUTER SYSTEMS

- - - - -

PRINCIPLES OF OPERATION

PRELIMINARY

PREFACE

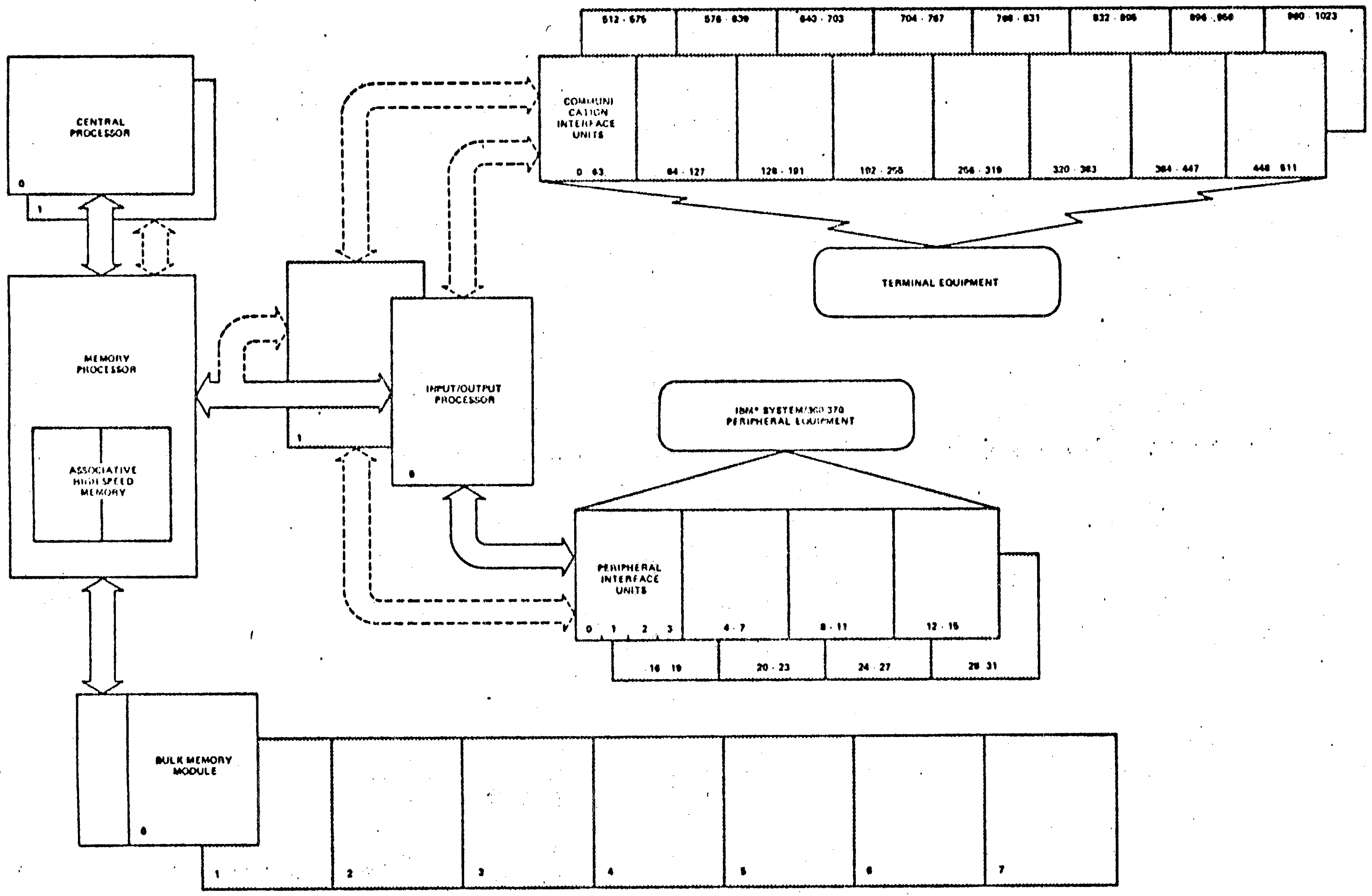
This document is a preliminary version of GEMINI Computer Systems Principles of Operations. It is divided into 9 Chapters.

The first chapter, Systems Architecture, provides an overview of GEMINI Computer Systems. Chapters 2, 3, and 4 on Memory, Addressing, and Instruction Set are fundamental to understanding the programming of GEMINI Systems. The remaining chapters are an amplification of the basic material of the previous chapters.

1.0 SYSTEM ARCHITECTURE

GEMINI Computer Systems are multiprocessing systems organized around a large central memory complex consisting of a single, high-performance bulk memory and a memory processor (see Figure 1.1). For this reason, the architecture of GEMINI Computer Systems is characterized as memory oriented. GEMINI Computer Systems include central processors, input/output processors, and a memory processor. The central processor and the input/output processors can be duplexed for increased throughput and failsoft capability. The software component of the GEMINI Computer Systems is the GEMINI Operational Control Program (OCP). It performs all resource allocation as well as failure diagnosis and recovery, and includes the GEMINI Programming Language (GPL) and a complete set of user services. The Operational Control Program is reentrant and, in a dual central processor system, can be simultaneously executed by both central processors.

Communication between system processors is accomplished by means of an interprocessor control bus, interrupt signals, and bulk memory. The interprocessor control bus is common to all processors and provides programmed communication between a central processor and all the other processors. A central processor uses the interprocessor control bus to send data to and request data from another processor.



*A REGISTERED TRADEMARK OF INTERNATIONAL BUSINESS MACHINES CORPORATION

Figure 1.1 GEMINI Computer System

Central processor response to asynchronous system events is accomplished by a multilevel priority interrupt scheme. The memory processor or an input/output processor can signal the occurrence of events by generating an interrupt that can be serviced by either central processor.

Access to bulk memory for all processors in the system is controlled by the memory processor. A central processor addresses only virtual memory and the memory processor transforms the virtual address into a real bulk memory address. The virtual memory organization makes it possible for a central processor to access memory with contiguous addresses even though the corresponding real bulk memory locations may be noncontiguous. An input/output processor accesses memory by presenting both a virtual address and its corresponding real bulk memory address.

1.1 MEMORY PROCESSOR

The GEMINI Memory Processor controls all accesses to memory. Accesses to bulk memory are automatically buffered by an associative high-speed memory, which is transparent from a programming viewpoint. The contents of high-speed memory are dynamically maintained by a mechanism that replaces those portions of the memory least recently referenced by a central processor. The high-speed memory is accessed by means of a virtual address. Therefore, any requested data that is currently in the high-speed memory is transferred between the high-speed memory and the requesting processor without requiring an address transformation.

When a central processor requests data that is not in the high-speed memory, the virtual address is transformed to a real bulk memory address. The required data is accessed from bulk memory and is used to replace data in the high-speed memory that was least recently used by a central processor. When an input/output processor requests data that is not in the high-speed memory, bulk memory is accessed directly using the real bulk memory address supplied by the requesting processor.

Virtual memory is divided into memory segments, and each segment consists of over a billion virtual byte addresses. The segmenting of virtual memory provides the isolation of unrelated program areas that is required in a multiprogramming environment. A memory segment is divided into over a million pages, each page containing 1024 bytes.

The GEMINI memory processor incorporates individual <u>page access controls</u> to prevent unauthorized accesses to memory by a central processor. Unauthorized write accesses by an input/output processor are similarly prevented by separate write access controls for each page of memory These two access protection mechanisms are independent of each other.

1.2 CENTRAL PROCESSOR

The GEMINI Central Processor is a micro-programmed digital computer which incorporates the following features.

1.2.1 Operating Mode

A central processor operates in a hierarchy of four possible modes: private, subsystem, service, and supervisor mode. The mode determines which instructions can be executed, which memory segments can be accessed, and restricts the accessing of individual pages within segments.

1.2.2 Instruction Format

A GEMINI instruction consists of an operation code, break-point control information, and from 1 to 11 addresses. Depending on the operation code of the instruction, each address can define an operand that is from 1 to 16 bytes in length. All operations have a required number of addresses, and many operations include addresses that can optionally be specified.

In addition to the standard instructions, there are two extended instructions that permit up to 256 user-defined operations and up to 256 additional system-defined operations. The microprograms associated with the extended instructions can reside in <u>either</u> bulk memory or control memory.

1.2.3 <u>Instruction Set</u>

The GEMINI instruction set provides arithmetic operations, byte string operations, list operations, and control operations.

Arithmetic Operations --

The arithmetic operations provided by the standard GEMINI instruction set include:

Computational
Signed binary and decimal integer
Binary magnitude integer
Address
Logical

The computational operations permit the use 8 data types which include binary or decimal, real or complex, and which have fixed or floating-point adjustment. Binary Computational Arithmetic operations have a precision of just over 33 decimal digits and an exponent range of over 19,728 decimal orders of magnitude. Decimal computation arithmetic operations have a maximum precision of 27 digits and an exponent range of 1,999 orders of magnitude. The instruction set provides for optional rounding and normalization of the results of all computational arithmetic operations. If a result is not to be normalized, it can be adjusted to a specified precision.

Signed binary integer arithmetic operations have a maximum precision just over 38 decimal digits, while signed decimal integer operations have a maximum precision of 31 digits. Magnitude binary integer operations have a maximum precision of 128 bits, and address arithmetic operations have a maximum precision of 126 bits. The logical operations can be performed on data that is from 8 to 128 bits in length.

Byte String Operations --

The GEMINI instruction set provides operations that manipulate a byte string or any 8-bit character within the byte string. The byte strings can be from 1 to 16 bytes in length.

[illegible]

Copy operations can be performed with an optionally specified bit displacement and an optionally specified mask. Comparison operations can be performed with an optional mask specification. Other operations include the generating of a string of contiguous 1 or 0 bits, and the counting of the number of binary or decimal digits in a string having a 0 or non-0 value.

List Operations --

The GEMINI instruction set provides list operations that permit the generation and maintenance of stack, queue, and ring structures. The lists can be easily expanded and relocated in memory.

1.2.4 Execution Breakpoints

These breakpoints permit program interrupts to be generated before or after a specified instruction is executed, or when a specified standard or extended operation code is detected. In addition, there are 8 simultaneous address breakpoints providing selective memory access monitoring. There are also breakpoints that monitor statement execution and thus provide source language debugging.

1.2.5 State Vectors

There are two state vectors, namely the processor state vector and the system state vector. The processor state vector contains the information necessary to control instruction execution and report the current status of a central processor. Information such as the mode, the address of the instruction currently being executed, the instruction breakpoint controls, and the interrupt controls are located in the processor state vector. When an interrupt is activated, the required processor state vector information is automatically preserved and subsequently restored when the interrupt servicing routine has completed execution.

Similarly, the system state vector contains the information necessary to control the devices which are external to a central processor and report the current operating status of the system.

1.2.6 Interrupt System

The multilevel priority interrupt system controls both private and shared interrupts. Private interrupts are the result of conditions caused directly by a central processor. Shared interrupts result from the occurrence of asynchronous conditions and in a dual central processor system may be serviced by either central processor.

Individual controls are associated with each interrupt and permit a central processor to service the interrupt or ignore the occurrence of the interrupt condition. In addition, interrupts may be deferred under program control.

1.3 INPUT/OUTPUT PROCESSOR

The GEMINI Input/Output Processor links all peripheral devices and communication lines to the Memory Processor. Peripheral devices are attached through peripheral interface units and communication lines are attached through communication interface units.

Up to 32 peripheral interface units are supported in groups of 4. Each unit has a bandwidth of 5 million bytes per second and is physically and electrically compatible with all IBM System/360 and System/370 device controllers. Any peripheral interface unit can be utilized as a System/360 or System/370 selector, block multiplexor, or byte multiplexor channel. Up to 1024 block multiplexor subchannels are available in 64-unit increments, and up to 256 byte multiplexor subchannels are available in 16-unit increments.

The communication interface units connect communication equipment to the GEMINI system. Up to 1024 communication interface units are supported in 64-unit increments. Each unit has a bandwidth of 9600 bits per second and conforms to EIA RS232C interface specifications.

The combined bandwidth of all peripheral interface units and communication interface units is 12 million bytes per second.

2.0 MEMORY

The GEMINI Computer Systems memory organization is based on a virtual memory concept. This organization allows the use of a virtual memory that is larger than the available real memory, and therefore requires a mechanism that converts virtual addresses to real addresses. In GEMINI Computer Systems, virtual-to-real address transformation is performed by the memory processor.

Related to the accessing of memory is the format in which information is stored. The GEMINI System allows variable-length, multiaddress instructions and accommodates many types of data.

2.1 ORGANIZATION

The basic unit of addressable information in GEMINI Computer Systemsis the <u>byte</u>, which is composed of 8 contiguous bits numbered as shown below:

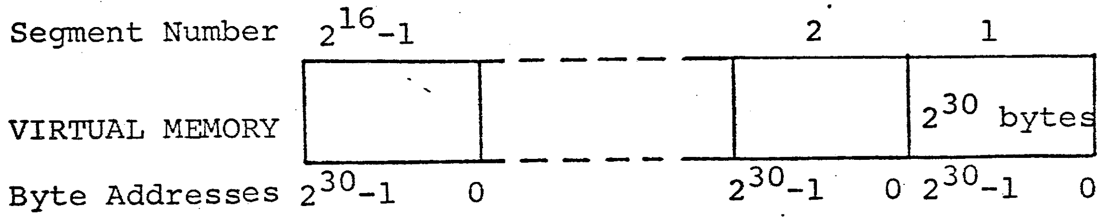
trailing	7	6	5	4	3	2	1	0	leading

Bit position 0 is defined as the leading bit position, and bit position 7 is defined as the trailing bit position.

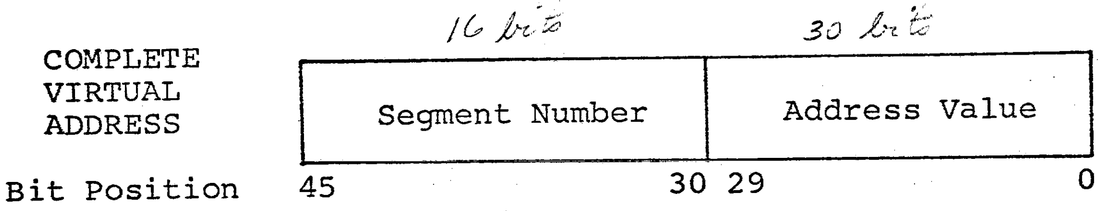
A page of memory is defined as 1024 contiguous bytes numbered from 0 to 1023. The page is the basic unit for assignment and protection of memory.

2.1.1 Virtual Memory

The Virtual memory organization of GEMINI Computer Systems allows a central processor to use contiguous addresses to access programs and data that reside in non-contiguous pages of real memory. Virtual memory is divided into segments so that unique segments can be assigned to distinct tasks executing in a multiprogramming environment. There are 2^{20} virtual pages in a memory segment. A memory segment is defined as a contiguous set of address values ranging from 0 to $2^{30}-1$. There are $2^{16}-1$ memory segments, each of which is identified by a unique segment number. The following shows the organization of virtual memory:



A complete virtual address is required to access virtual memory. A complete virtual address is a 46-bit binary number consisting of 16 high-order bits that specify the segment number and 30 low-order bits that specify the address value.



The segment number specifies the memory segment, and the address value specifies the byte location within the segment The formation of a complete virtual address is discussed in the following sections.

2.1.2 Virtual Address to Real Address Transformation

A page of a memory segment that has been referenced by a task is referred to as an assigned page. A page becomes assigned to a task when the task makes a write access to a previously unreferenced page of a memory segment.

A central processor addresses only virtual memory. The transformation of a complete virtual address to a real memory address is automatically performed by the memory processor. The memory processor utilizes an address transformation table to accomplish the transformation of virtual to real addresses. For each assigned page of virtual memory, the address transformation table contains an entry that consists of the virtual page address, the number of the real page associated with the virtual page, and page access controls that define the memory protection associated with the page.

A virtual page address is formed by using the high-order 36 bits of the complete virtual address, which is in effect dividing the complete virtual address by 1024. The transformation of a virtual address to a real address is shown in Figure 2.1.

A detailed discussion of the assignment of memory pages and the address transformation process is in 9.0 Memory Processor.

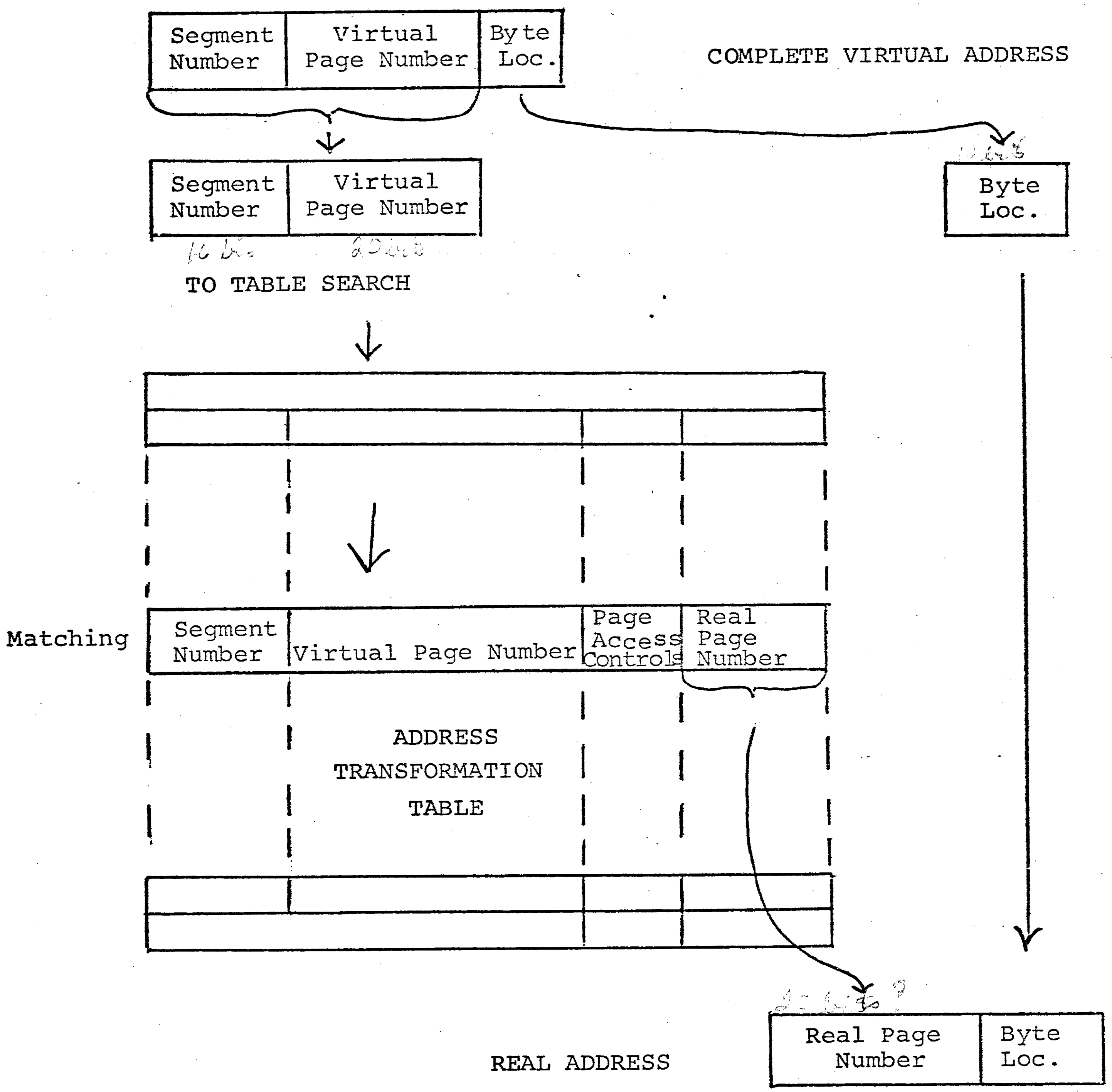


Figure 2.1 Virtual Address to Real Address Transformation

2.2 CENTRAL PROCESSOR MEMORY ACCESSES

In the formation of a complete virtual address, the segment number is obtained from one of the five 16-bit segment number registers, which are located in the processor state vector. An execution mode control in the processor state vector (as described in 2.2.3 Access Mode) determines which segment number registers can be used.

2.2.1 Segment Number Registers

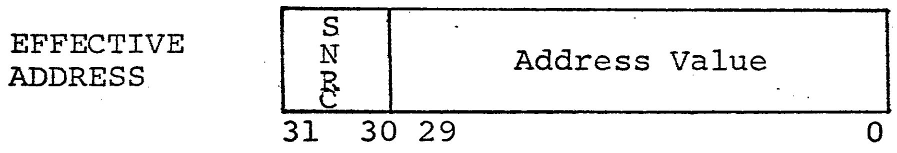
The segment number registers are numbered 0 through 4. The associated memory segments are identified as follows:

Segment Number Register	Memory Segment Nomenclature
0	Private
1	Subsystem
2	Context
3	System (private to a central processor)
4	System (shared by central processors)

Note that the memory segment being referenced by either segment number register 3 or 4 is referred to as the system segment.

2.2.2 Formation of a Complete Virtual Address

A Central Processor accesses virtual memory by first forming an effective address which consists of a 30-bit address value and a 2-bit segment number register code (SNRC). The formation of an effective address is discussed in 3.0 Addressing.

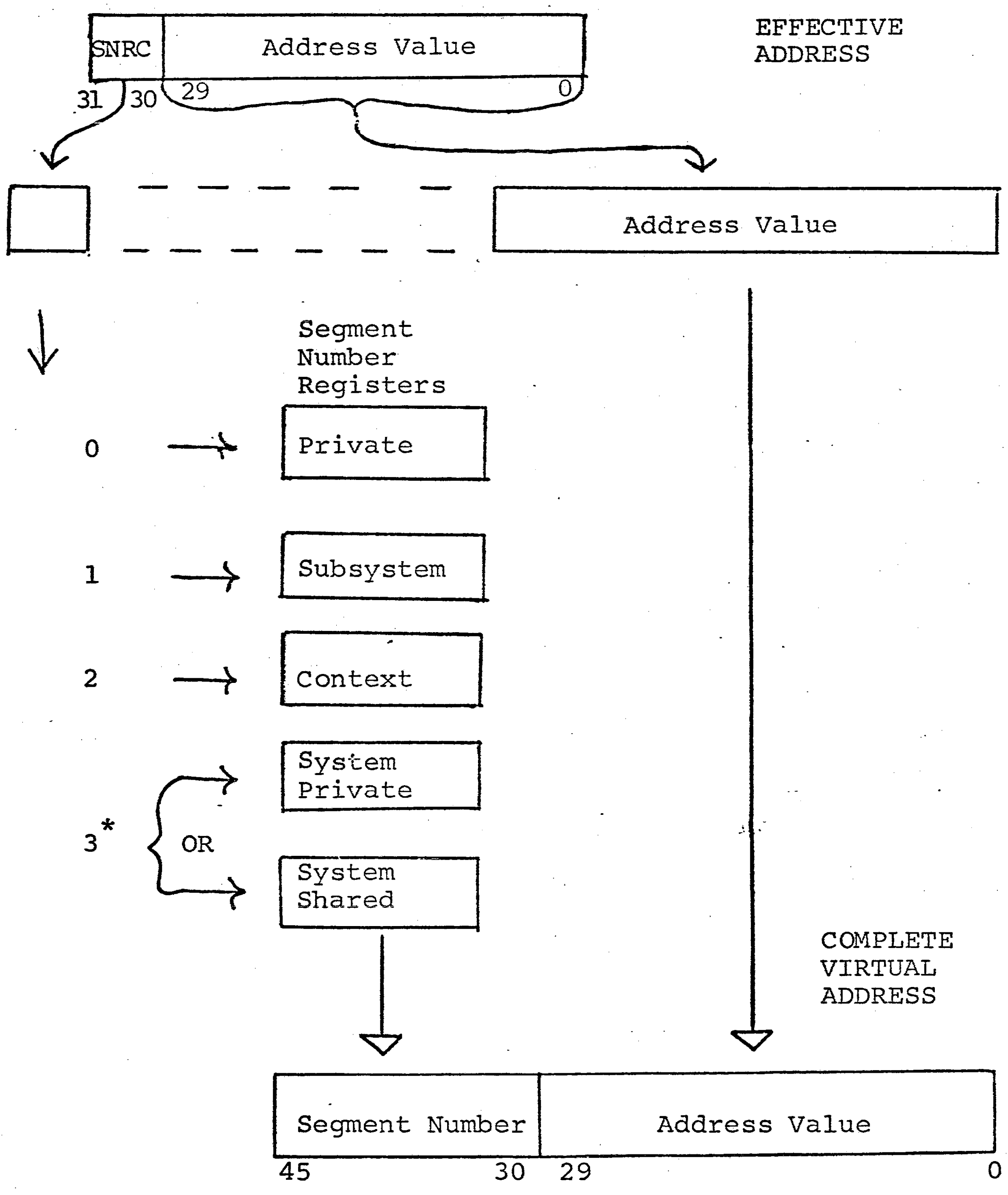


The segment number register code specifies the segment number register that is to be used during the formation of the complete virtual address according to the following convention:

SNRC	Segment Number Register
0	Private (SNR0)
1	Subsystem (SNR1)
2	Context (SNR2)
3	System (SNR3) if address value $< 2^{14}$
	System (SNR4) if address value $\geq 2^{14}$

Note that when the segment number register code equals 3, both the code and the address value are used to specify the segment number.

A complete virtual address is formed by concatenating the address value in the effective address with the contents of the specified segment number register. The formation of a complete virtual address is shown in Figure 2.2.



* If SNRC = 3 and the Address Value $< 2^{14}$ then System Private is chosen for its Segment Number value; else System Shared is chosen.

Figure 2.2 Formation of A Complete Virtual Address

2.2.3 Access Mode

The specification of the segment number registers is restricted by the access mode. The access mode is normally equal to the central processor mode, but in certain circumstances the value for the access mode is specified under program control.

Instruction Access --

An instruction is accessed from the effective address specified by the instruction location register located in the processor state vector. During instruction access access mode is equal to the central processor mode.

The following restrictions apply to the specification of segment number registers during instruction access:

Access Mode	Legal Segment Number Register Specification
Private (0)	Private
Subsystem (1)	Subsystem, Context
Service (2)	System, Context
Supervisor (3)	System, Context

A control located in the processor state vector allows the access mode restrictions on instruction access to be ignored.

Data Access --

Data is accessed at the location specified by the effective address. During data accesses the access mode is equal to one of the following:

- The central processor mode.
- The link mode specified by a subprogram control block. This condition is true if the effective address resulted from an argument addressing calculation. Argument addressing is described in 3.7 Argument Addressing.
- The mode specified by an operand of an ANALYZE instruction. These instructions are discussed in 4.7 Control Operations.

During data accesses the following restrictions apply to the specification of segment number registers:

Access Mode	Legal Segment Number Register Specification
Private (0)	Private
Subsystem (1)	Private, Subsystem, and Context
Service (2)	All registers can be selected
Supervisor (3)	All registers can be selected

A control located in the processor state vector allows the access mode restrictions on data access to be ignored.

2.2.4 Page Access Controls

In addition to the access mode restrictions that constrain the specification of the segment number registers, page access controls are associated with each page of virtual memory. These controls are located in the address transformation table and are shown in Figure 2.1.

The page access controls include a page mode access and a private write access control. The page mode access specifies the minimum mode required to access the page. During all memory accesses, the access mode must be greater than or equal to the page mode access control. When the same memory segment is being referenced by independent tasks which execute in different modes, this control enables the protection of the higher access mode pages.

The private write access allows a page to be protected from write accesses by a central processor. — [illegible]

2.3 DATA AND INSTRUCTION FORMATS

The following describes the data and instruction formats used in GEMINI Computer Systems. The formats of addresses are described in 3.1 Address Formats.

2.3.1 Byte Strings

A <u>byte string</u> is defined as a set of contiguous bytes:

trailing	n	n-1	...	1	0	leading

Byte position 0 is the leading byte and corresponds to the lowest numbered memory address; the trailing byte corresponds to the highest numbered memory address.

2.3.2 <u>Data Encoding</u>

Several methods of data representation are used in GEMINI Computer Systems. The GEMINI Computer Systems data types are composed of one or more of the following basic forms: character, binary or decimal digits, integers, fractions.

Character --

The GEMINI Computer Systems can accommodate any of the 6-, 7-, or 8-bit codes in common usage. However, the code used by the GEMINI Operational Control Program is an 8-bit extension of the 7-bit ASA code for Information Interchange (ANSI X3.4-1968). ASACII is embedded in the GEMINI byte structure as follows:

7	6	5	4	3	2	1	0	Bit Position
0	b_7	b_6	b_5	b_4	b_3	b_2	b_1	GEMINI Representation

Binary/Decimal Digits --

Within the arithmetic data types, either binary or decimal representations of numbers are used. A binary digit can be 0 or 1; a decimal digit consists of four binary digits and can have a numeric value of 0 through 9. A sign digit at the trailing end of a number indicates whether the number is positive or negative. A binary sign digit of 0 indicates a positive value, and a binary sign digit of 1 indicates a negative value. A decimal sign digit of 10 (binary 1010) indicates a positive value, and a decimal sign digit of 11 (binary 1011) indicates a negative value.

Integer --

The integer (a signed or unsigned whole number) assumes a leading (low-order) binary or decimal point. For signed integers the trailing digit is the sign digit, while for the unsigned integers the trailing digit is the highest order digit of the number.

The maximum value of an integer is expressed as:

$$\pm\ (\underline{r}^{\underline{n}}-1)$$

where $\underline{r}$ is the radix (2 for binary, 10 for decimal)
$\underline{n}$ is the number of digits in the operand, excluding the sign digit

Any other integer value represented may be expressed as:

$$\pm\ (\underline{d}_{\underline{n}-1}\underline{r}^{\underline{n}-1} + \underline{d}_{\underline{n}-2}\underline{r}^{\underline{n}-2} + \ldots + \underline{d}_1\underline{r}^1 + \underline{d}_0\underline{r}^0)$$

where $\underline{d}_{\underline{i}}$ is the $\underline{i}$th digit in the operand
$\underline{n}$ is the number of digits in the operand, excluding the sign digit
$\underline{r}$ is the radix (2 for binary, 10 for decimal)

Fraction --

The fraction always has a sign digit in the trailing digit position. The fraction assumes a binary or decimal point between the trailing (high-order) numeric digit and the sign digit. The value of the fraction is expressed as follows:

$$\pm\ (\underline{d}_{\underline{n}-1}\underline{r}^{-1} + \underline{d}_{\underline{n}-2}\underline{r}^{-2} + \ldots + \underline{d}_1\underline{r}^{\underline{n}-1} + \underline{d}_0\underline{r}^{\underline{n}})$$

where $\underline{d}_{\underline{i}}$ is the $\underline{i}$th numeric digit
$\underline{n}$ is the number of numeric digits
$\underline{r}$ is the radix (2 for binary, 10 for decimal)

2.3.3 Arithmetic Operands

There are 3 basic forms of arithmetic data: the computational, integer, and logical. The computational form has an exponent part with integer format and a fixed-point part with fraction format. There are 8 data types that have the computational form; they are distinguished as having BINARY or DECIMAL base, FIXED or FLOAT scale, and REAL or COMPLEX mode. The <u>integer</u> form has only an integer part in the integer format. There are 4 data types that have the integer form: the 2 signed INTEGER data types, distinguished as having BINARY or DECIMAL base, and the 2 unsigned data types, MAGNITUDE and ADDRESS. The <u>logical</u> form has a byte-string format. All bits in each byte in the string are treated independently. The LOGICAL data type has the logical form.

This discussion presents the formats of the data types with the computational and integer forms. The computational data types include real and complex operands, and the integer data types include integer, magnitude, and addressing operands.

Real Operand --

A <u>real operand</u> (see Figure 2.3) is a binary or decimal number consisting of a fixed-point part and an exponent. The exponent has an integer format and occupies the 2 leading bytes; the fixed-point part has a fraction format and occupies the next 1 to 14 bytes. The trailing digit in both fixed-point part and exponent is a sign digit.

The value of the real operand is the product

$$\underline{f}\,\underline{r}^{\underline{e}}$$

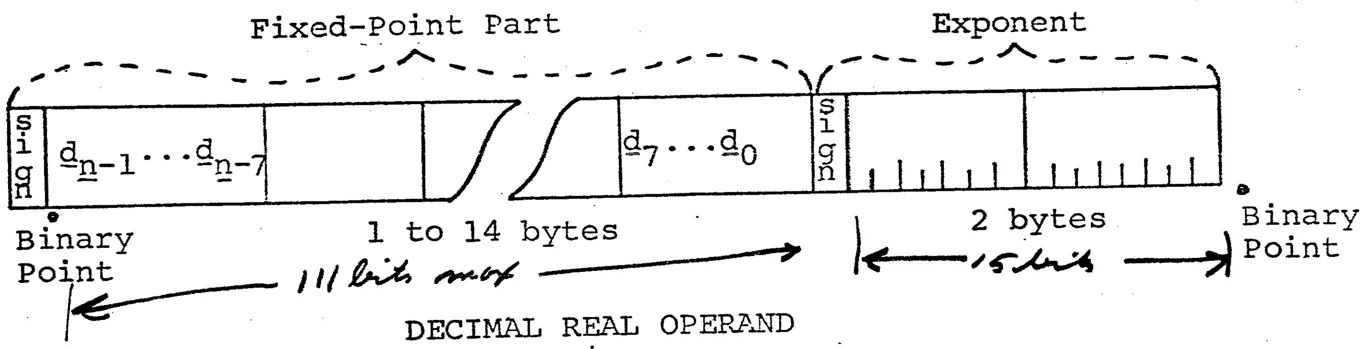
where $\underline{f}$ is the value of the fixed-point part
$\underline{r}$ is the radix (2 or 10)
$\underline{e}$ is the value of the exponent

The range of the real operand is constrained by the value of the exponent. The maximum absolute value of _e_ is $2^{15}-1$ for binary and 10^3-1 for decimal.

A real operand has either a binary format or a decimal format in both exponent and fixed-point parts. For each format there are 2 data types distinguished as FIXED or FLOAT. This distinction serves to specify which of 2 sets of computational controls are used during execution of an operation. (The computational controls are discussed in 4.3 Computational Arithmetic.) Thus, there are 4 data types for real operands:

- BINARY FIXED REAL
- BINARY FLOAT REAL
- DECIMAL FIXED REAL
- DECIMAL FLOAT REAL

BINARY REAL OPERAND



DECIMAL REAL OPERAND

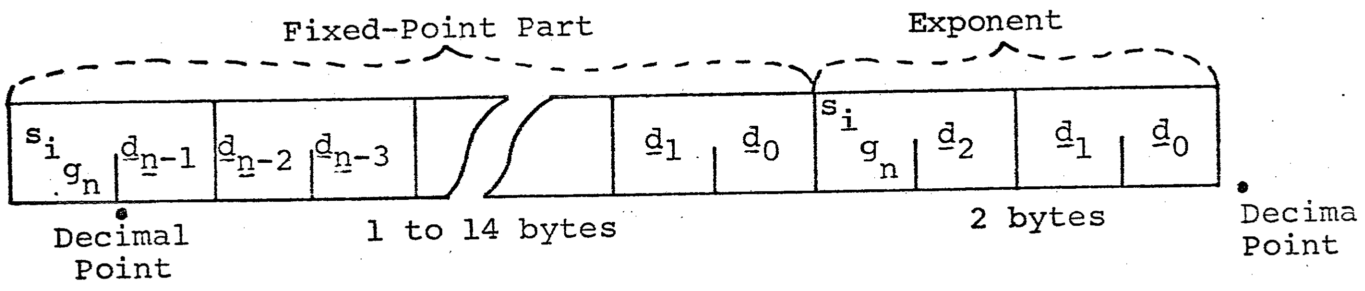


Figure 2.3 Real Operands

Complex Operand --

A <u>complex operand</u> (see Figure 2.4) represents numbers of the form

$$\underline{z} = \underline{x} + \underline{yi}$$

where $\underline{i}$ is the imaginary coefficient and consists of the pair of equal-length real operands $\underline{x}$ and $\underline{y}$. The leading member of the pair is the real component $\underline{x}$, and the trailing real number is the imaginary component $\underline{y}$. The resulting complex operand may be 6 to 32 bytes long in even increments.

The two real operands have the same base (BINARY or DECIMAL) and scale (FIXED or FLOAT). Thus, there are 4 data types for complex operands:

- BINARY FIXED COMPLEX
- BINARY FLOAT COMPLEX
- DECIMAL FIXED COMPLEX
- DECIMAL FLOAT COMPLEX

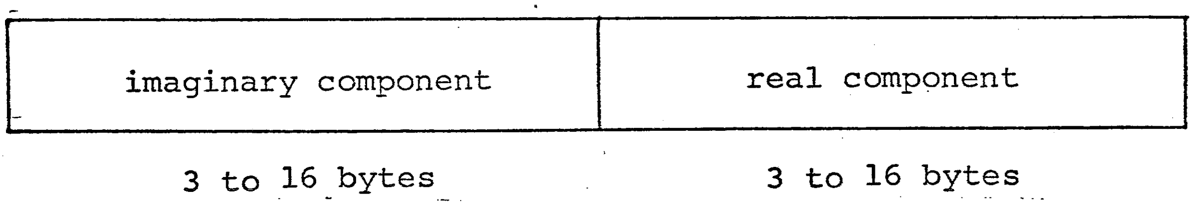


Figure 2.4 Complex Operand

Integer Operand --

An <u>integer operand</u> (see Figure 2.5) is a signed binary or decimal integer from 1 to 16 bytes long. The trailing digit of the operand is a sign digit, and the digit that immediately precedes the sign digit is the highest order digit. A binary or decimal point is assumed to precede the lowest order (leading) digit. There are 2 data types for integer operands:

- BINARY INTEGER
- DECIMAL INTEGER

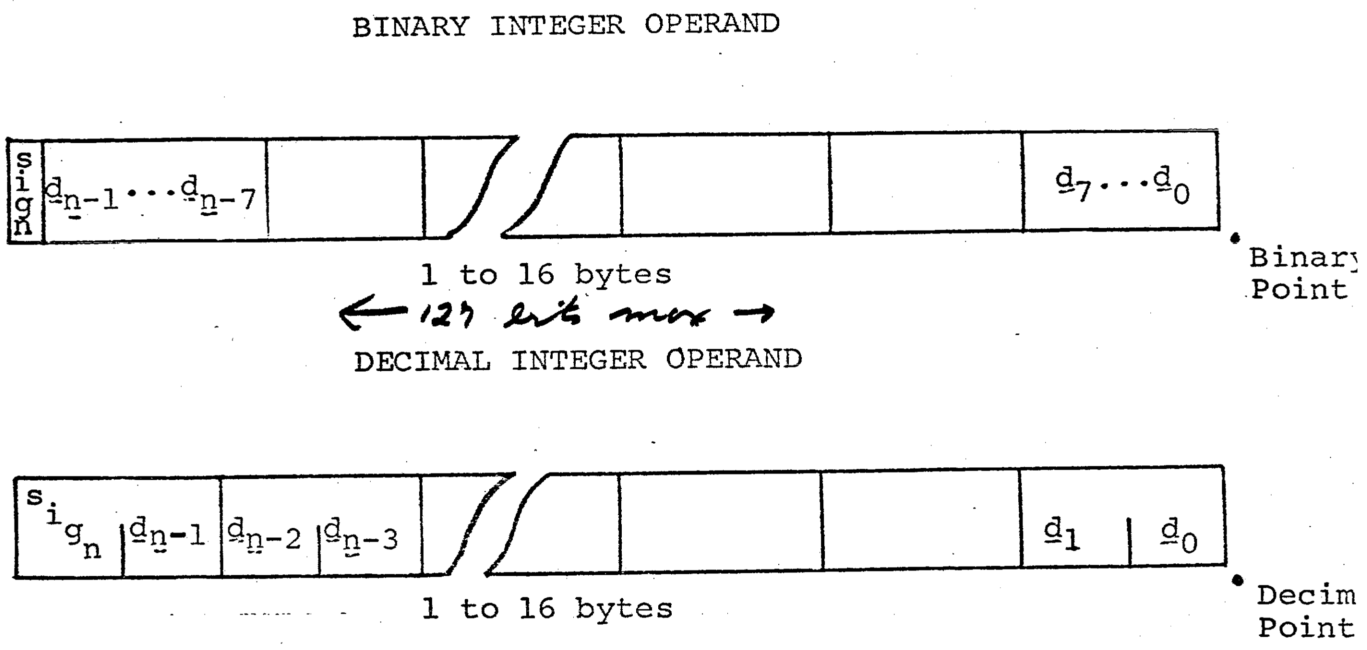


Figure 2.5 Integer Operands

Magnitude Operand --

A magnitude operand (see Figure 2.6) is an unsigned binary integer from 1 to 16 bytes long. The trailing digit position has the highest order of magnitude. All magnitude operands have the MAGNITUDE data type.

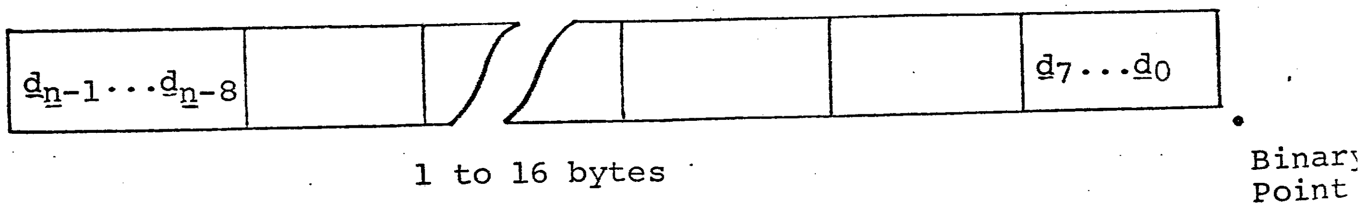

$\underline{d}_{n-1} \cdots \underline{d}_{n-8}$
$\underline{d}_7 \cdots \underline{d}_0$
1 to 16 bytes
Binar
Point

Figure 2.6 Magnitude Operand

Address Operand --

An <u>address operand</u> (see Figure 2.7) is an unsigned binary integer 1 to 16 bytes long. The two trailing bits are ignored in address arithmetic. The two trailing bit positions are treated separately so that segment number register codes remain unaffected by address arithmetic (see 3.1.2 Address Elements). All address operands have the ADDRESS data type.

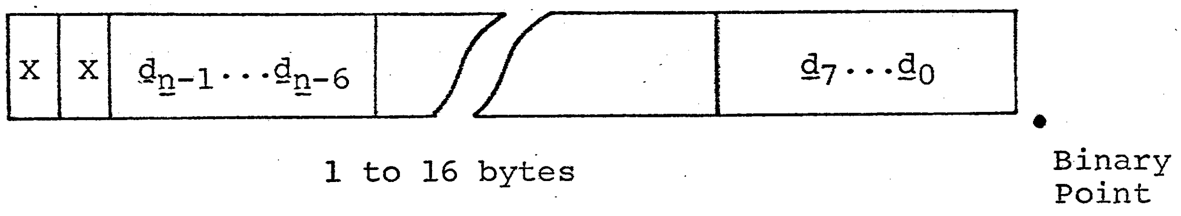


Figure 2.7 Address Operand

2.3.4 Logical

The <u>logical operand</u> is a set of 1 to 16 contiguous bytes. These operands are used in character or bit string manipulations with logical operations (Boolean arithmetic). The instruction set permits bit manipulation within operands that have byte lengths. Logical operands have the LOGICAL data type.

2.3.5 Instructions

GEMINI Computer Systems provide 3 formats for instructions: standard, extended, and null. The formats of these instructions are shown in Figure 2.8.

Standard Instructions --

A standard instruction consists of an operation code followed by an instruction qualifier and a variable-length address string.

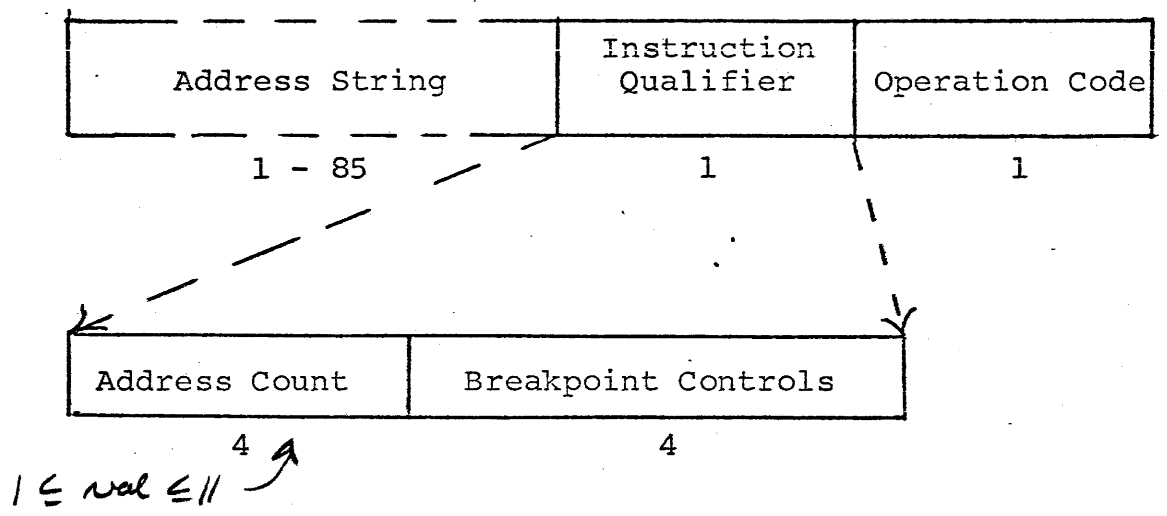
The operation code is one byte long and is used to specify the function to be performed by the instruction.

The instruction qualifier byte contains an address count and breakpoint controls. The address count indicates the number of addresses to be found in the address string and may vary from 1 to 11 in value. The address count is a function of the operation specified by the instruction. The breakpoint controls permit tagging of an instruction for debugging purposes. The breakpoint controls are described in detail in 6.0 Execution Breakpoints.

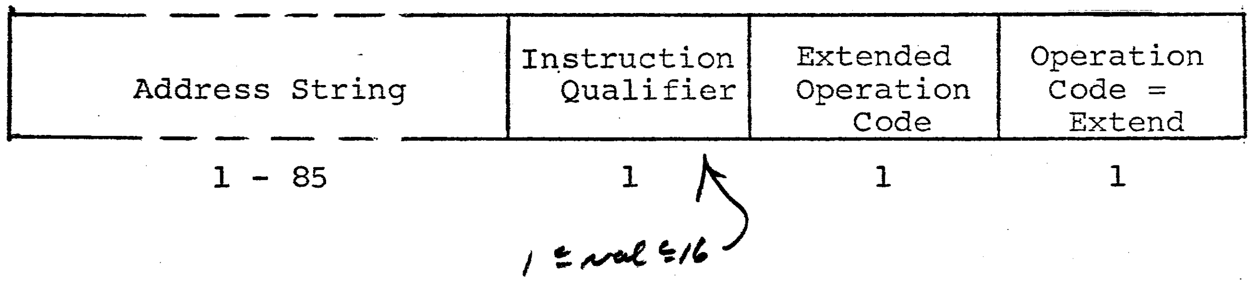
The number of addresses in an address string corresponds to the address count. Each address, depending upon its type, can vary from 1 to 17 bytes in length. Address formats are discussed in 3.1 Address Formats.

?

STANDARD INSTRUCTIONS



EXTENDED INSTRUCTIONS



NULL INSTRUCTION

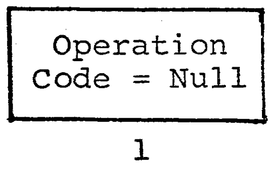


Figure 2.8 Instruction Formats Showing Field Lengths in Bytes

Extended Instructions --

An extended instruction consists of an operation code followed by an extended operation code, an instruction qualifier, and an address string. The operation code specifies that the byte following it is an extended operation code. The instruction qualifier and address string for extended instructions are the same as those defined for standard instructions, except that there can be as many as 16 addresses in the address string.

Null Instruction --

The null instruction consists of an operation code only and specifies that no operation is to be performed.

2.3.6 Control Operand

The instruction set utilizes a variety of implicit operand formats for passing information between called and calling routines; for providing tables of entries to system and subsystem services; and for providing base addresses, pointers, and element lengths for the queues, stacks, and rings of list processing. These formats are presented here for summary reference only; detailed information is given where the formats are discussed in context with related material.

Subprogram Control Block --

The <u>subprogram control block</u> contains information needed by a service program or a subprogram to effect a return to the program that called it and to refer to arguments provided by the calling program. The control block contains addresses of the calling instruction and the return location, and an argument list address for referring to operands used by the subprogram but provided by the calling program. An argument index and link mode are provided for utility and system integrity. The subprogram control block has a 16-byte format, as shown in Figure 2.9.

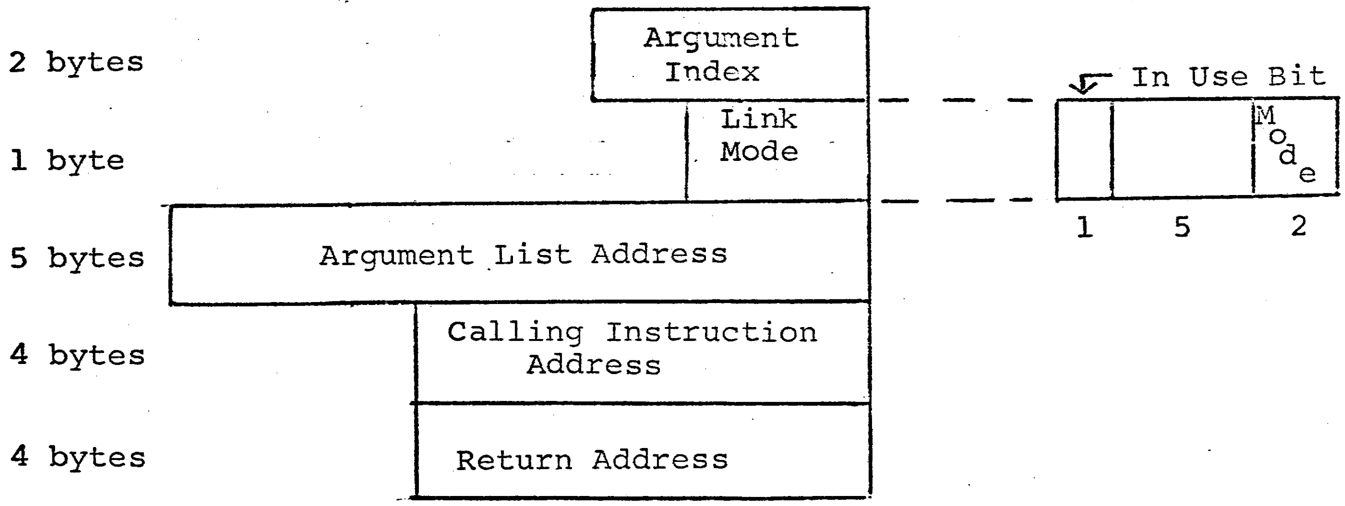


Figure 2.9 Subprogram Control Block

System and Subsystem Service Entry Tables --

The system and all subsystems maintain tables to specify the entry location of the routines that perform system and subsystem services. Each entry in the table is 4 bytes long and provides a 30-bit address value and a 2-bit transition mode value, as shown in Figure 2.10. For the system service entry table the address value points to the entry of a system service routine and has an understood segment number register code of 3. For a subsystem service entry table, the address value points to the entry of a subsystem service routine and has an understood segment number register code of 1. The transition mode specifies a central processor mode for execution of the service.

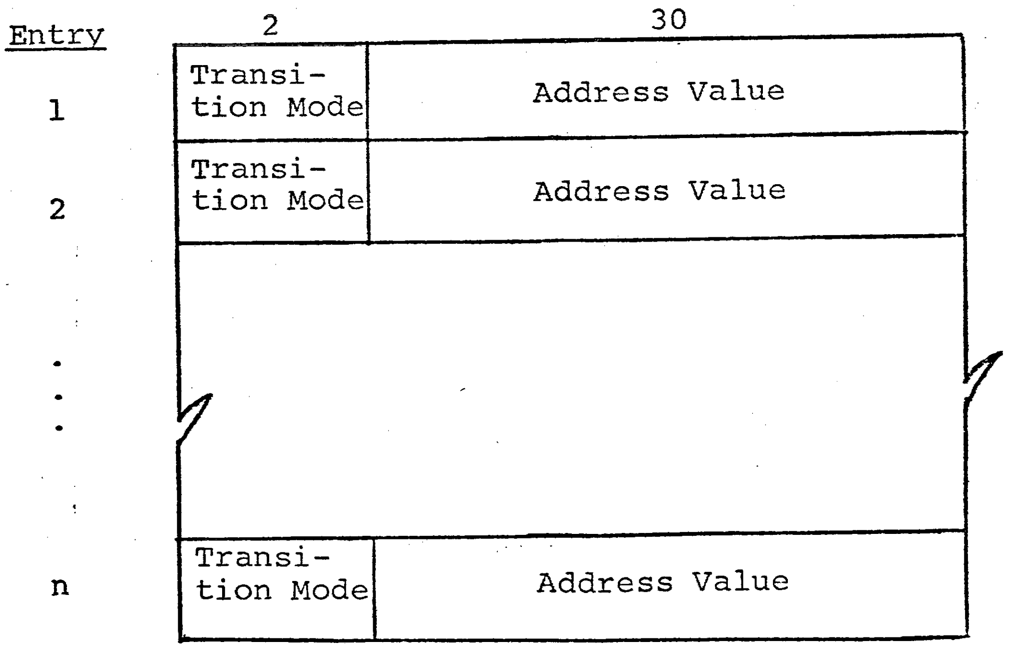


Figure 2.10 Service Entry Table

List and List Domain Control Blocks --

The list control block and the list domain control blocks are used by the list processing operations. The list control block contains addresses and pointers to list elements, while the list domain control block provides characteristics of the list domain such as size, element length, and location of next free element. The formats of these control blocks are given in Figure 2.11.

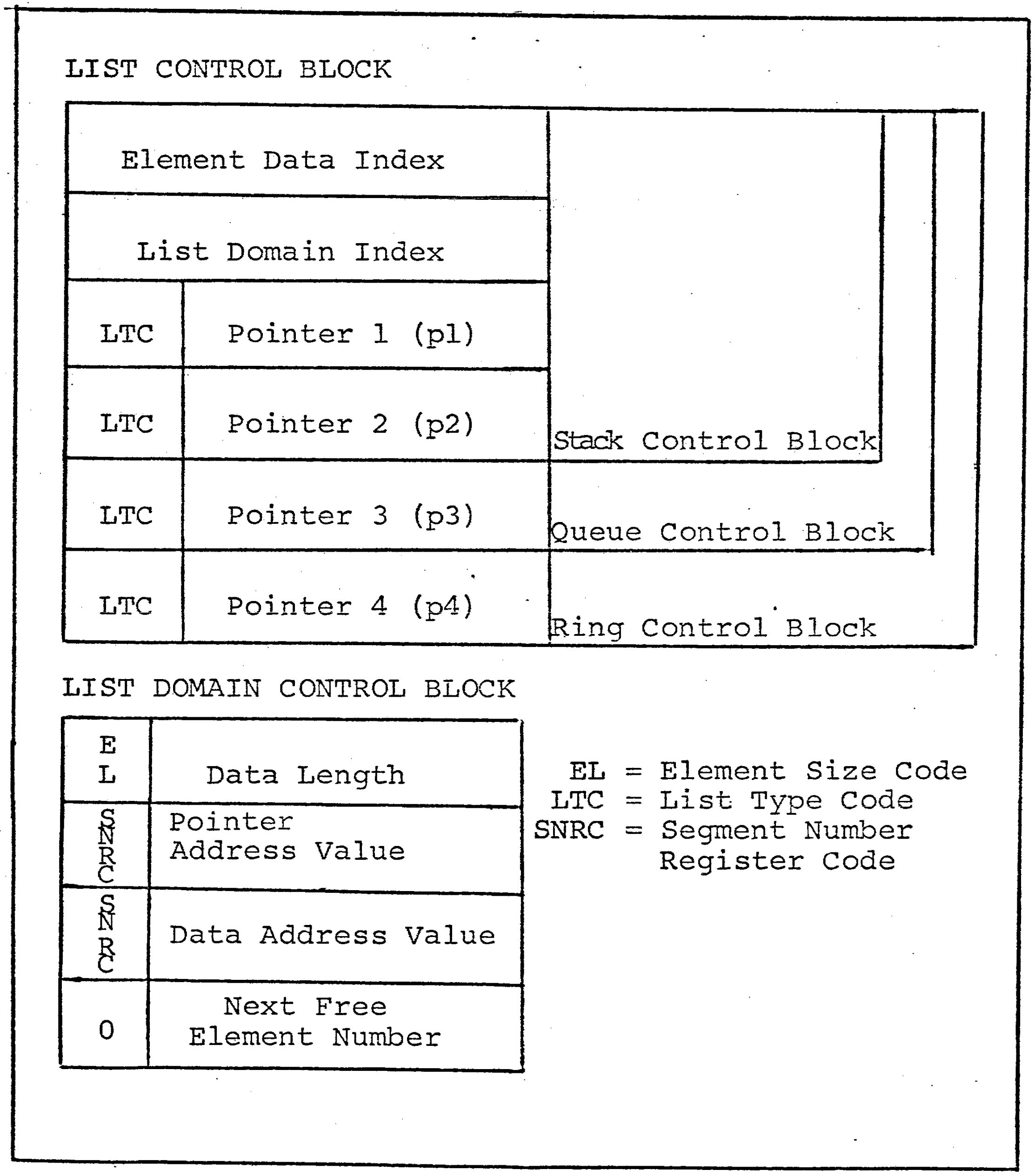


Figure 2.11 List and List Domain Control Blocks

3.0 ADDRESSING

A standard instruction or an extended instruction contains a variable length address string (see Figure 2.8 Instruction Format). The address string contains a variable number of addresses and each address provides the necessary information to define an operand. An operand is a value or a storage location referenced in the execution of an instruction. Each address in the address string either defines an operand or specifies the location of another address which in turn defines the operand.

An address defines an operand either by containing the operand value or by specifying the location of the operand. The location of an operand is referred to by an effective address, which specifies the beginning byte location. Because an operand can be variable in length, an address may also specify the length of the operand. The various methods of defining an operand form 5 classes of addressing:

- Null - the address is omitted from the address string
- Immediate - the address contains the operand value
- Direct - the address specifies the location of an operand
- Indirect - the address specifies the location of another address which in turn defines the operand
- Argument - The address references a byte string that contains a list of addresses. An address in the list defines the operand

Two of the classes, direct and indirect, permit the use of indexing to specify an effective address. When indexing is used, the effective address is calculated using a value in the address modified by a value in an index. An index is a 4-byte field in memory that contains an effective address.

3.1 ADDRESS FORMATS

An address is a byte string consisting of an address qualifier and 0, 1,or 2 of the following elements:

- Base Address
- Index Address
- Argument Index
- Immediate Operand

There are 8 address formats, ranging in length from 1 to 17 bytes. Permissible combinations are depicted in Figure 3.1.

3.1.1 Address Qualifier

The address qualifier is the leading byte of each address and consists of a 4-bit address type code and a 4-bit length code.

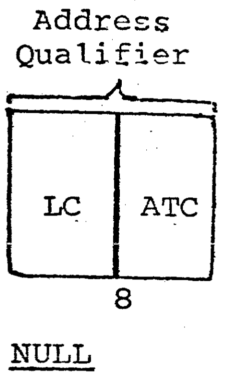
The address type code indicates the class of addressing, the address format, and optionally an indexing method. The address type codes are listed in Table 3.1.

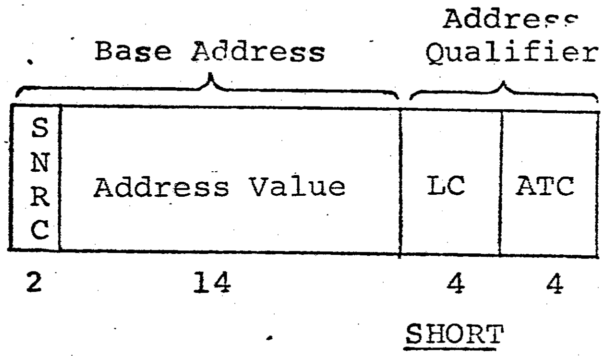
The length code specifies the operand length. Values of 1 through 15 specify, respectively, operand lengths of 1 through 15 bytes. A value of 0 specifies an operand length of 16 bytes, or in certain uses it specifies an address that requires no length specification.

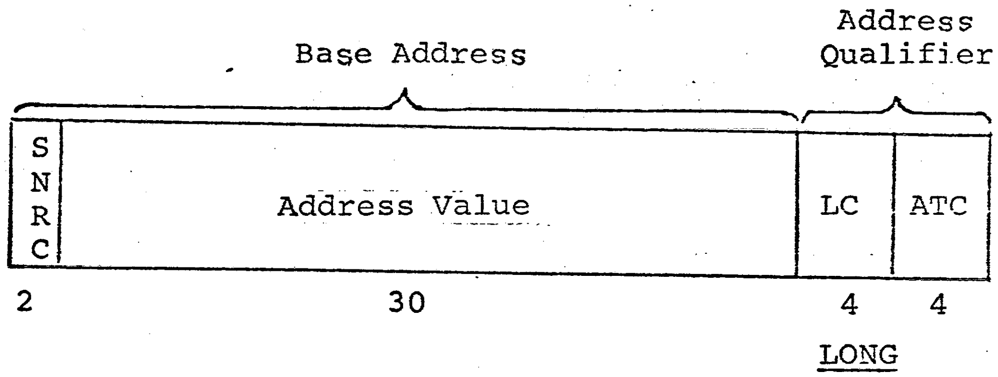
3.1.2 Address Elements

An address value specifies a byte location as a binary magnitude number that is either 14 or 30 bits in length.

ATC	=	Address Type Code
LC	=	Length Code
SNRC	=	Segment Number Register Code







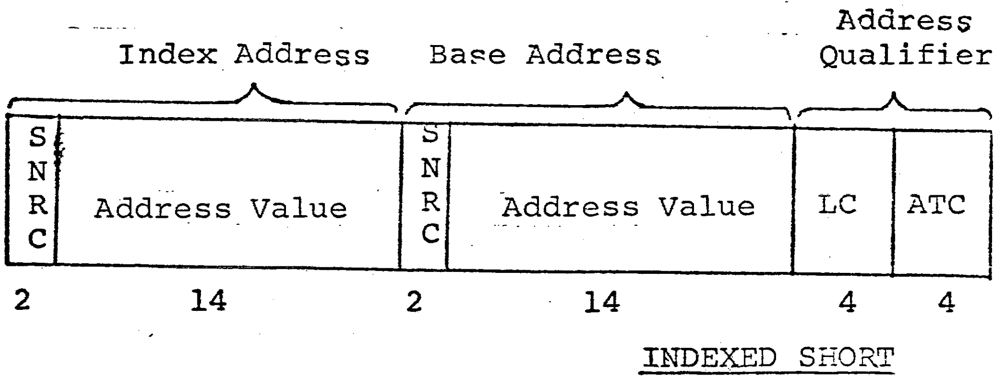
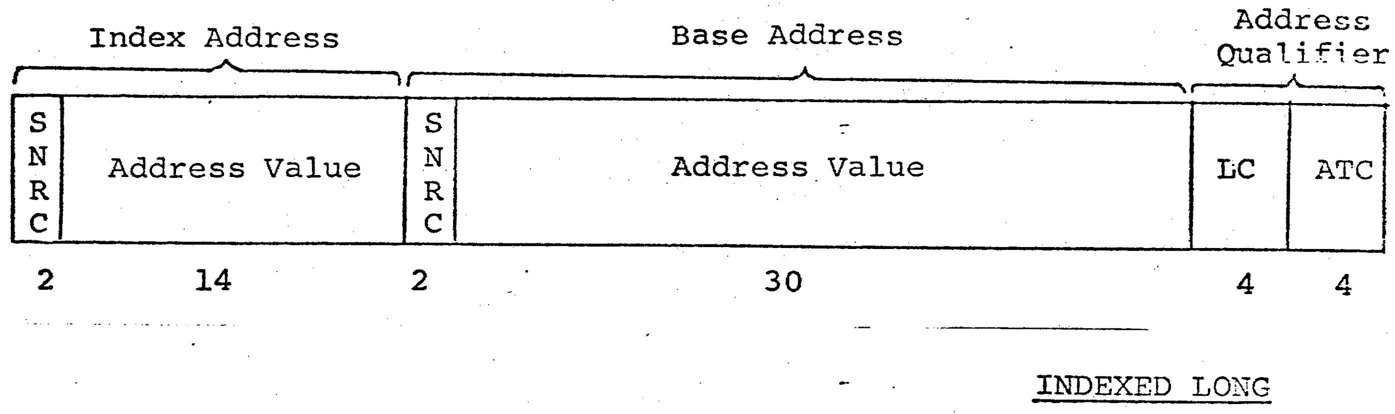
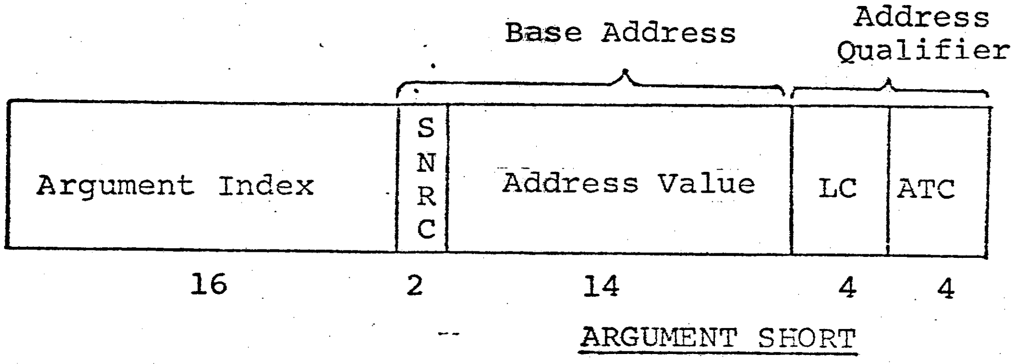
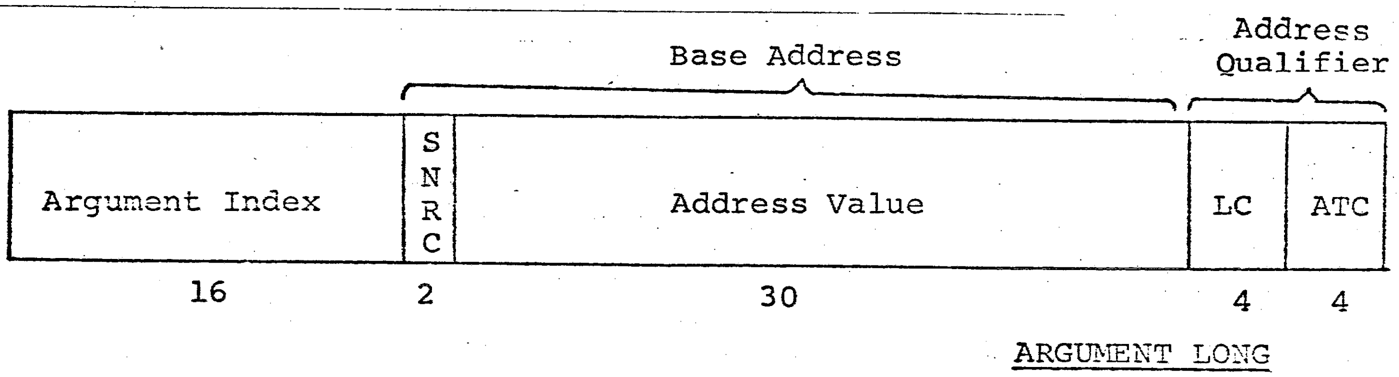


Figure 3.1 Address Formats (1 of 2)







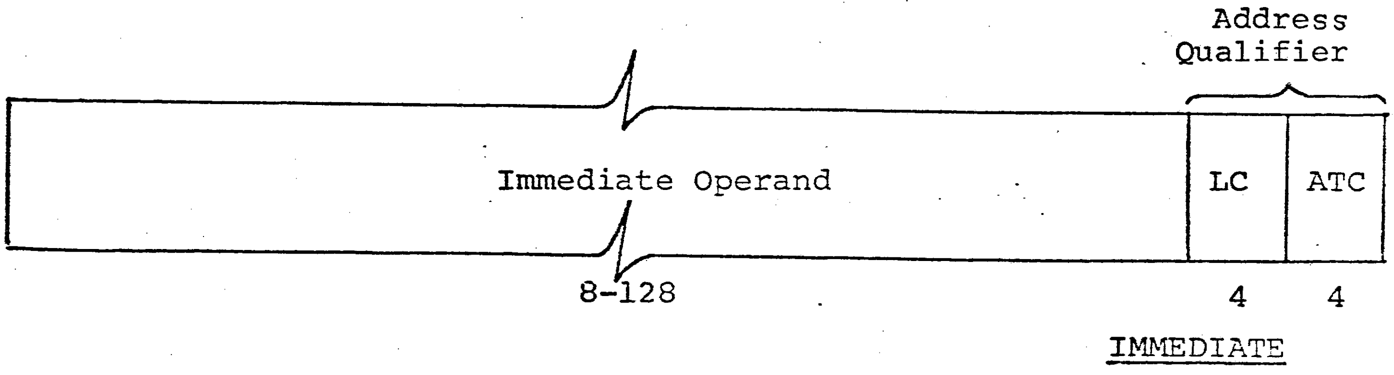


Figure 3.1 Address Formats (2 of 2)

Table 3.1 Address Type Codes

Address Type Code	Class	Format	Addressing Type
0	Null	Null	Null
1	Direct	Short	Short Direct
2	Direct	Indexed Short	Short Direct With Displacement Indexing
3	Direct	Indexed Short	Short Direct With Alignment Indexing
4	Indirect	Short	Short Indirect
5	Argument	Argument Short	Short Argument
6	Indirect	Indexed Short	Short Indirect With Displacement Indexing
7	Indirect	Indexed Short	Short Indirect With Alignment Indexing
8	Immediate	Immediate	Immediate
9	Direct	Long	Long Direct
10	Direct	Indexed Long	Long Direct With Displacement Indexing
11	Direct	Indexed Long	Long Direct With Alignment Indexing
12	Indirect	Long	Long Indirect
13	Argument	Argument Long	Long Argument
14	Indirect	Indexed Long	Long Indirect With Displacement Indexing
15	Indirect	Indexed Long	Long Indirect With Alignment Indexing

A 14-bit number is referred to as a short address value, and a 30-bit number is referred to as a long address value. When a short address value is used, 16 high-order 0's are appended prior to its use in the computation of the effective address.

A segment number register code is a 2-bit binary magnitude number used in the specification of a segment number register. (See discussion in 2.2.2 Formation of a Complete Virtual Address.)

A base address consists of an address value and a segment number register code. When a base address is present in an address format, it immediately follows the address qualifier. A short base address contains a short address value and is 2 bytes long. A long base address contains a long address value and is 4 bytes long.

An index address specifies the location of a 4-byte index and has the same format as a short base address. When an index address is present in an address format, it immediately follows the base address. An index address is always 2 bytes long.

An argument index is a 16-bit binary magnitude number. In an address that specifies argument addressing, the argument index immediately follows the base address.

An immediate operand is a value in the address that is used as an operand in the execution of an instruction. The immediate operand immediately follows the address qualifier.

3.2 INDEXING

Indexing operations can be performed in both direct and indirect addressing. Indexing requires an address format with a base address and an index address. When indexing is performed, the address value specified by the base address is modified by the contents of an index to form the effective address. The segment number register code in the effective address is specified either by the base address or the index.

An index is 4 bytes in length and contains a long address value and a segment number register code. The format of the index is identical to that of an effective address.

All indexing operations are performed using binary magnitude arithmetic which produces a result in the range 0 to $2^{30}-1$. The operation performed is identical to that described in 4.5 Magnitude Operations except that the truncation interrupt is never generated.

3.2.1 Displacement Indexing

In displacement indexing, the address value in the base address is added to the address value in the index to form the address value in the effective address (see Figure 3.2).

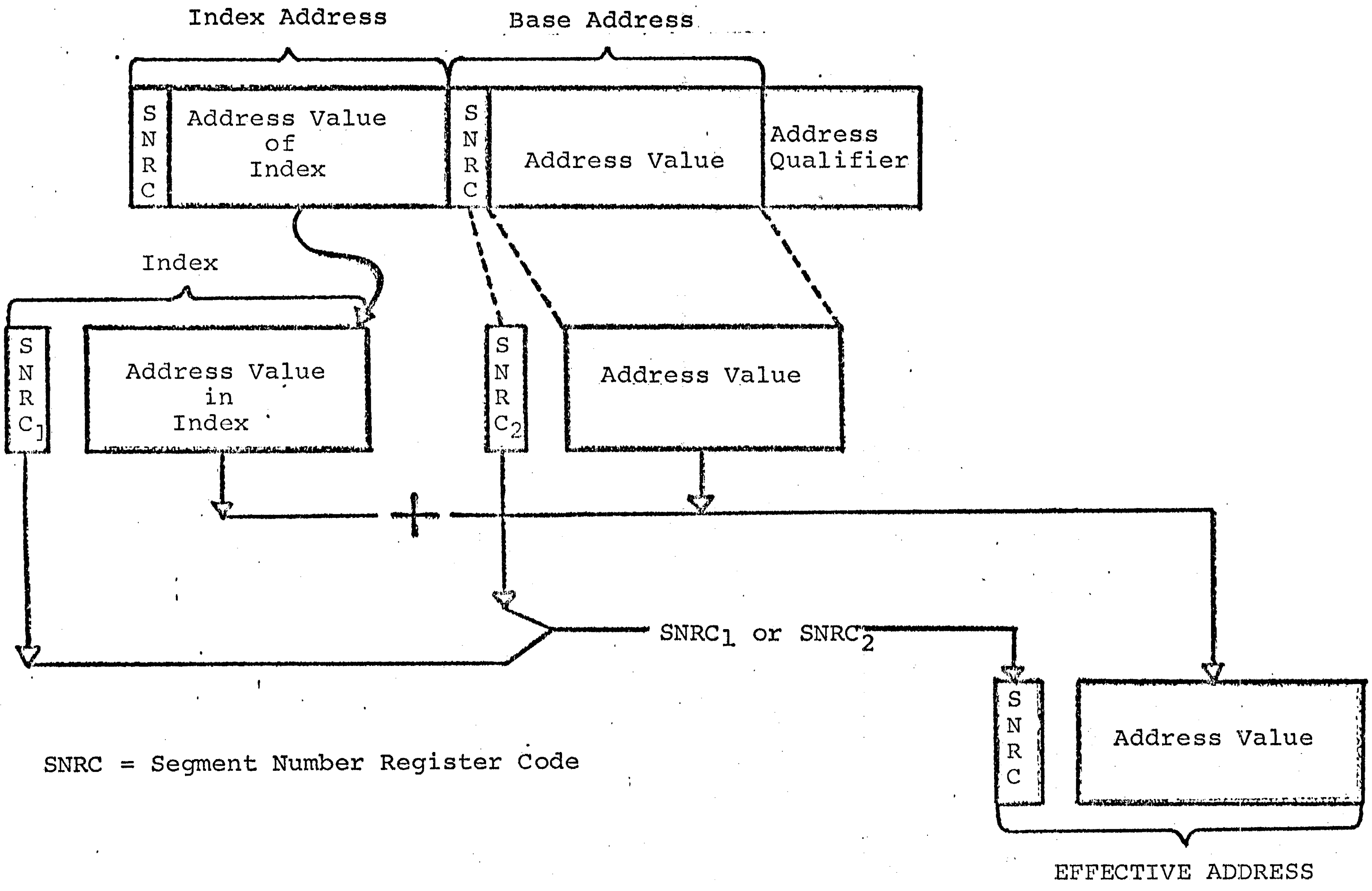


Figure 3.2 Displacement Indexing

3.2.2 <u>Alignment Indexing</u>

In alignment indexing, the address value in the index is multiplied by the length of the operand. The product is then added to the address value in the base address to produce the address value in the effective address (see Figure 3.3

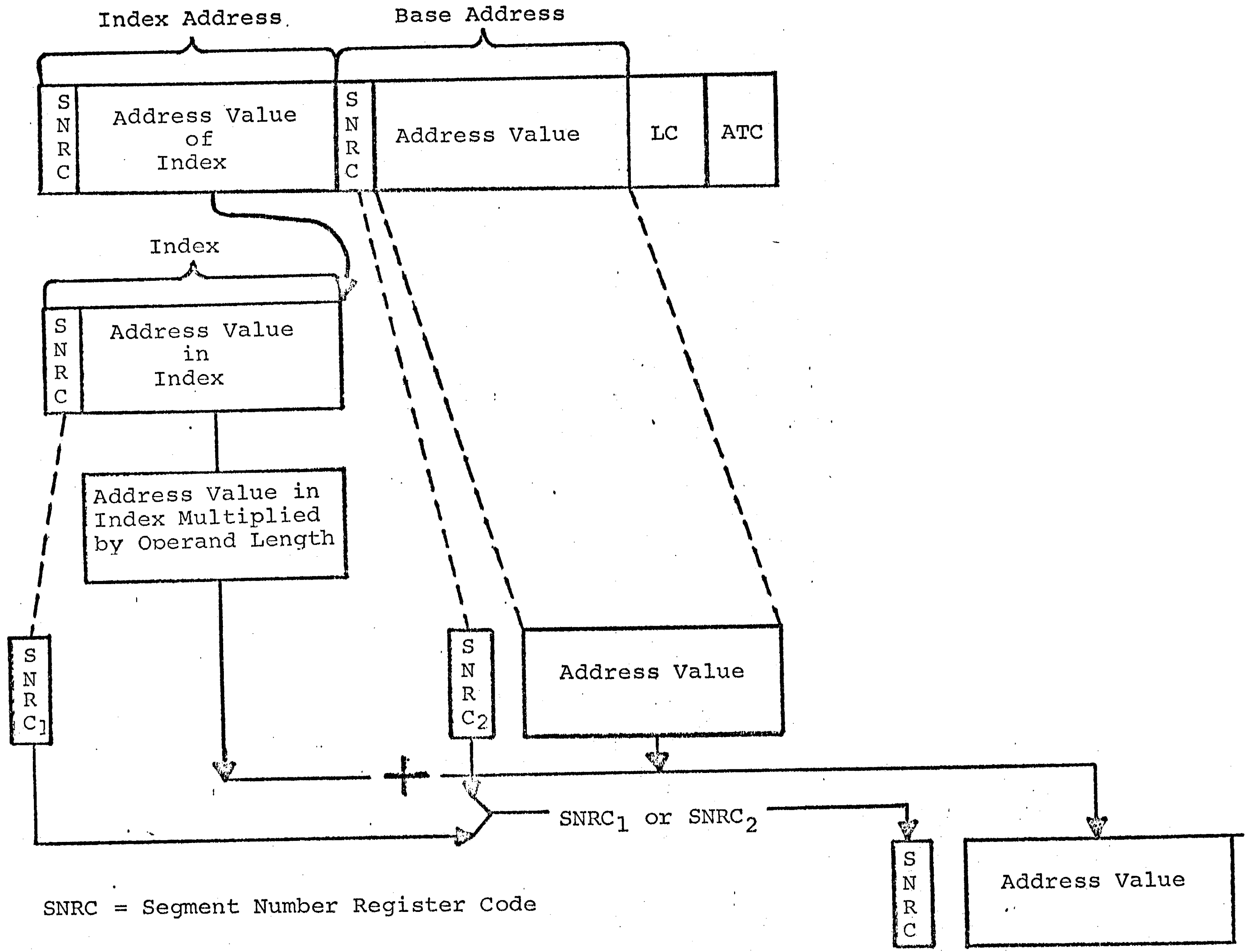
Alignment indexing specifies elements in an array by their position within the array. The positions within an array are numbered 0 through <u>n</u>-1, where <u>n</u> is the total number of elements in the array. The length of the elements in the array is specified by the length code of the instruction that references the array.

3.2.3 <u>Specification of the Segment Number Register Code</u>

When indexing is performed, the segment number register code in the effective address is the segment number register code in the index or the segment number register code in the base address, whichever is greater. However, the segment number register code in the index must be less than or equal to the segment number register code in the index address.

3.3 NULL ADDRESSING

A null address consists of only an address qualifier containing an address type code of 0 and a length code of 0. A null address either indicates the omission of an operand



SNRC = Segment Number Register Code

Figure 3.3 Alignment Indexing

EFFECTIVE ADDRESS

or indicates that the first address in the instruction specifies the effective address. The use of null addressing is discussed in 4.2.2 Implicit Addresses and 4.2.1 Optional Operands.

3.4 IMMEDIATE ADDRESSING

Immediate addressing uses a value in the address,called an immediate operand, as an operand in the execution of the instruction. The data format of the immediate operand is determined by the operation code of the instruction.

One addressing type is associated with immediate addressing. The address format consists of an address qualifier with a length code value from 1 to 8, followed by an operand that is from 1 to 16 bytes in length. Except for complex operations, the length code specifies the length of the immediate value. For complex operations the length code specifies the length of the real component,and the imaginary component is assumed to be the same length; thus, for complex operations the immediate operand is twice the length specified by the length code.

Immediate addressing cannot be used to specify an operand that is used as the destination for the result of an operation nor to specify a location for the transfer of control.

3.5 DIRECT ADDRESSING

Direct addressing specifies the effective address using a base address or a base address modified by a displacement or alignment indexing calculation. The address format consists of an address qualifier, a base address, and optionally an index address. Associated with direct addressing are 6 addressing types:

- Short Direct Type --

  The effective address is specified by a short base address.

- Long Direct Type --

  The effective address is specified by a long base address.

- Short Direct with Displacement Indexing --

  The effective address is specified by a short base address modified by a displacement indexing calculation.

- Long Direct with Displacement Indexing --

  The effective address is specified by a long base address modified by a displacement indexing calculation.

- Short Direct with Alignment Indexing --

  The effective address is specified by a short base address modified by an alignment indexing calculation.

- Long Direct with Alignment Indexing --

  The effective address is specified by a long base address modified by an alignment indexing calculation.

3.6 INDIRECT ADDRESSING

When indirect addressing is used, the address in the address string of the instruction specifies the location of another address. That address, referred to as the final address, defines the operand.

In indirect addressing the location of a final address is specified by a base address or by a base address modified by a displacement or alignment indexing calculation. The address format consists of an address qualifier, a base address, and an optional index address. The length code specifies the length of the final address. Six addressing types specify indirect addressing:

- Short Indirect Type --

  The location of a final address is specified by a short base address.

- Long Indirect Type --

  The location of a final address is specified by a long base address.

- Short Indirect with Displacement Indexing --

  The location of a final address is specified by a short base address modified by a displacement indexing calculation.

- Long Indirect with Displacement Indexing --

  The location of a final address is specified by a long base address modified by a displacement indexing calculation.

- Short Indirect with Alignment Indexing --

  The location of a final address is specified by a short base address modified by an alignment indexing calculation.

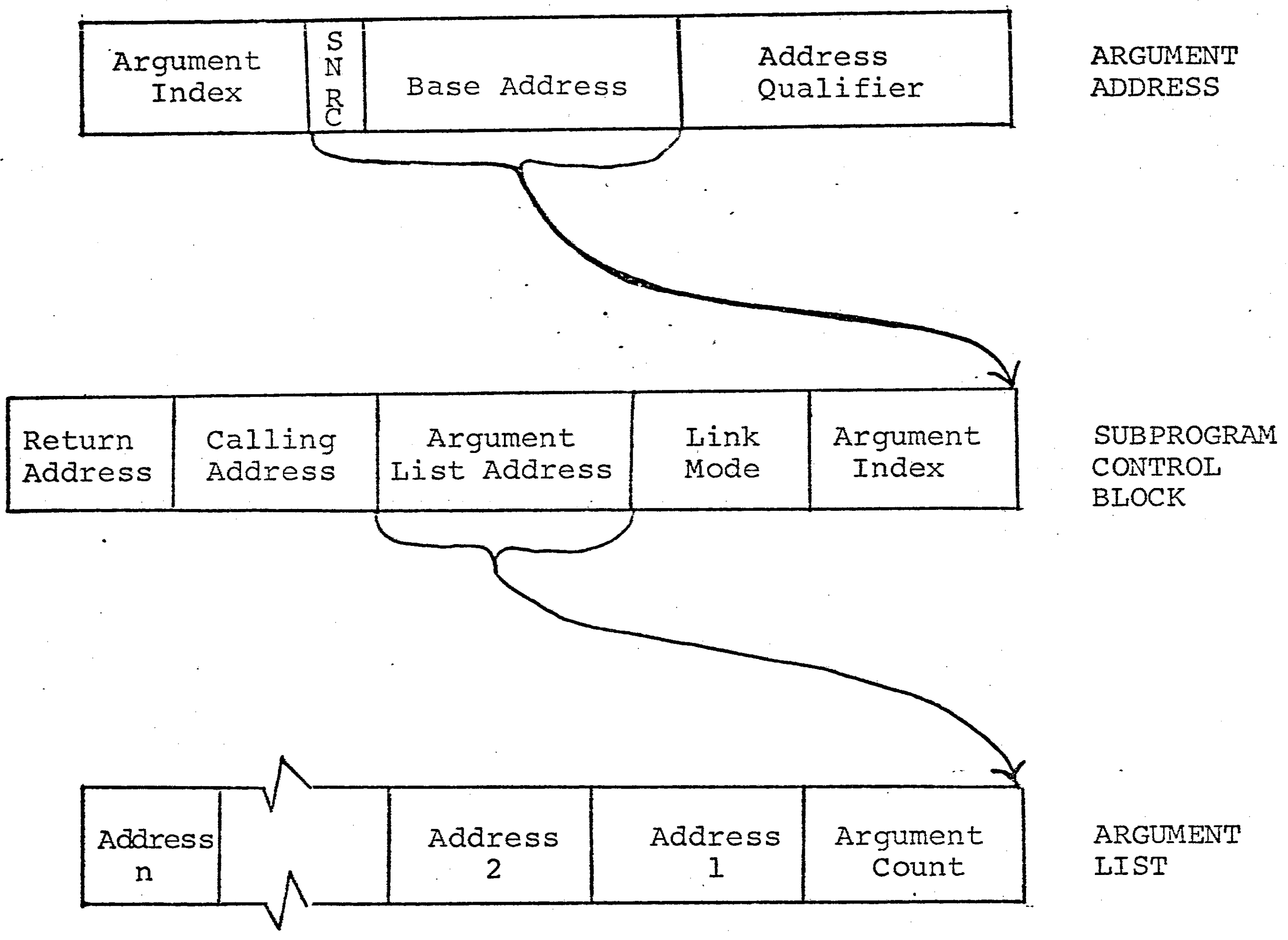
- Long Indirect with Alignment Indexing --

  The location of a final address is specified by a long base address modified by an alignment indexing calculation.

Only one level of indirect addressing is permitted, and the final address must specify null, immediate, or direct addressing.

3.7 ARGUMENT ADDRESSING

Argument addressing (see Figure 3.4) provides a method for passing operands to and from a called subprogram. When argument addressing is used, reference is being made to an operand defined by an address in an argument list. The argument list contains a count of the number of addresses as well as the actual list of addresses. The location of the argument list is specified in a subprogram control block along with control information necessary for argument addressing.



SNRC = Segment Number Register Code

Figure 3.4 Argument Addressing

The addressed operand is accessed through two levels of indirection:

- First, the base address in the instruction specifies the location of the subprogram control block. The subprogram control block specifies the location of the argument list.
- Second, an argument index in the instruction, or the subprogram control block, specifies the appropriate address in the argument list. The alternate argument index in the subprogram control block is chosen whenever the argument index value in the instruction has the value 0.

  The resulting address, referred to as the final address, defines the operand.

3.7.1 Argument Addressing Formats

The argument address has a format that consists of an address qualifier specifying argument addressing and a length code of 0, a base address, and an argument index. The base address specifies the location of a subprogram control block. The argument index occupies 2 bytes and contains a binary magnitude number.

An argument index in the range of 1 to $2^{16}-1$ specifies the position of an address in the argument list. An argument index value of 0 indicates that the alternate argument index in the subprogram control block is to be used for the specification of an address position in the argument list.

There are 2 addressing types that specify argument addressing.

- Short Argument

  A short base address specifies the location of the subprogram control block.

- Long Argument

  A long base address specifies the location of the subprogram control block.

3.7.2 Subprogram Control Block

A subprogram control block, with the exception of the value of the alternate argument index, is automatically generated during the execution of CALL, SYSTEM, and SUB-SYSTEM instructions. The use of this control block is discussed in 4.9 Control Operations. The following topics discuss the argument index, argument list address, and link mode used during argument addressing.

Argument Index --

The argument index field contains an argument index that is used only if the argument index in the argument address is equal to 0. If this argument index is used, its value ranges from 1 to $2^{16}-1$ and specifies, respectively, the first through the last addresses in the argument list. If the argument address contains a nonzero argument index, this field is ignored.

Argument List Address --

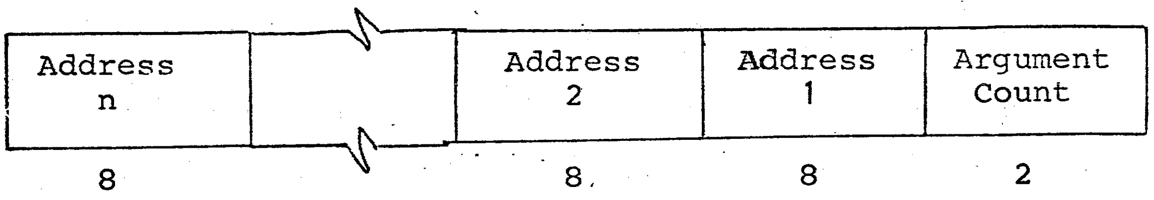
The argument list address field contains a long direct address that specifies the location of an argument list. During argument addressing, the length code specified by the argument list address is ignored.

Link Mode --

The leading 2 bits of the link mode field contain a value in the range 0 through 3 that specifies the access mode used during argument addressing. This field is required only if argument addressing is performed in a central processor mode greater than 0. If the central processor mode is 0, the link mode is assumed also to be 0. The access mode is discussed in 2.2.3 Access Mode.

3.7.3 Argument List

An argument list is a byte string consisting of a 2-byte argument count and from 0 to $2^{16}-1$ fields that are each 8 bytes in length. The argument count specifies the number of fields in the list. Each field contains an address adjusted to the leading byte of the field. The argument list format is shown below:



3.7.4 Mechanism of Argument Addressing

The base address associated with an argument address specifies the location of a subprogram control block. The subprogram control block is accessed and it specifies the location of an argument list. In order to access the argument list, the access must be legal in the mode specified by the link mode. The link mode must be less than or equal to the central processor mode.

An address in the argument list is specified either by the argument index in the argument address or by the argument index in the subprogram control block. If the argument index in the argument address is equal to 0, the argument index in the control block is used. If both argument indexes equal 0, or if the argument index used is greater than the argument count, the illegal argument index interrupt condition occurs.

The location of the referenced address in the argument list is calculated as follows:

Location = ALEA + 2 + 8*(AI-1)

where ALEA is the effective address of the argument list and
AI is the argument index used

The specified address in the argument list is referred to as the _final address_ and it defines the operand. Only one level of argument addressing is permitted and the final address must specify null, immediate, or direct addressing. If the final address references an operand in memory, the access must be legal in the mode specified by the link mode.

4.0 INSTRUCTION SET

The GEMINI instruction set is divided into 8 groups according to the data format on which the instructions operate:

- Computational Arithmetic Operations -- REAL and COMPLEX data
- Integer Arithmetic Operations -- INTEGER data
- Magnitude Arithmetic Operations -- MAGNITUDE and ADDRESS data
- Logical Operations -- LOGICAL data
- Miscellaneous Byte Operations -- byte strings
- Control Operations -- instruction locations
- List Operations -- list control blocks
- Select Operations -- data formats specific to internal and external devices

The notation used to describe the instruction set is defined in 4.1 Notation, and the features that are common to more than one group of instructions are presented in 4.2 Common Characteristics. The succeeding topics (4.3 thru 4.10) enumerate and describe the individual instructions in each of the eight groups listed above.

4.1 NOTATION

This section defines the types of operands that are used in the various instructions and explains the manner in which the individual instructions are discussed.

4.1.1 Operand Type

In the description of each instruction, the addresses that form the address string in the instruction are classified

according to the way the
operation uses the operand that each address specifies.
Under this classification, there are 3 types of operands: destination operands, source operands, and address operands.

Destination Operands --

A destination operand is a byte string in memory whose contents may be altered by the operation. A destination operand is either the receiving field for the result of an operation or a control block that is affected by the operation. For all instructions that have destination operands, the destination operand is specified by the first address in the instruction address string. A destination operand cannot be specified using null addressing or immediate addressing.

Source Operands --

A source operand is a value used by the operation. Source operands are specified by any type of addressing. Null addressing specifies an implicit operand (see 4.2.2 Implicit Addresses) or an omitted optional operand (see 4.2.1 Optional Operands). Immediate addressing specifies the value of the operand. Direct addressing specifies a byte string in memory whose contents are the value of the operand. If indirect or argument addressing is used, the final address may specify null, immediate, or direct addressing.

Address Operands --

An address operand is the effective address of a memory location to which control may be transferred. The contents of that memory location are not read or altered by the operation. An address operand can be specified only by direct addressing in the instruction address string or by direct addressing in the final address when indirect or argument addressing is used.

4.1.2 Instruction Descriptions

Each instruction description consists of the name of the instruction and the following topics:

- Operation
- Syntax
- Program Interrupts
- Semantics

Operation --

This topic lists names for all operations to which the instruction description applies. If only one operation is applicable, its name is identical to the name of the instruction and the operation topic is omitted.

Syntax --

This topic specifies the syntax of the instruction in the format of a GEMINI Programming Language (GPL) statement.

The syntax consists of keywords, special characters, function names, metavariables, and metacharacters.

Keywords appear in capital letters. They are used to identify the instruction or the meaning of particular operands.

Special characters are operators or delimiters. The operators are the traditional arithmetic, relational, and logical operators. The equal sign is used as a relational operator when it appears in a comparison operation and as the assignment operator in noncomparison operations. The delimiters are parentheses and commas. Parentheses are used to group one or more operands into a list. The list may specify the operands of a function, or it may follow a keyword that requires more than one operand specification. Commas are used to separate the operands in the list.

Function names appear in small letters. In function notation, the function name is used instead of a special character to denote an operation. Function notation is used only with the assignment operation; otherwise, keyword notation is used.

Metavariables appear in boldface and represent addresses in the address string. Each metavariable consists of a letter indicating the usage of the operand specified by the address, and a number indicating the position of the address in the instruction address string. The letters used are d for destination operand, s for source operand, and a for address operand. The allowable metavariables are d1, s1 through s11, and a1 through a5. In an actual GPL statement, programmer-specified symbols or literal values would replace these metavariables. The data type of the programmer-specified symbols determines which of a set of operation codes, all with the same syntactic representation, is specified for execution. The data type is ordinarily specified in a GPL DECLARE statement.

Another type of metavariable consists of the name of an option in boldface letters. This type of metavariable is used to specify options that are common to many instructions and whose format is described separately.

The metacharacters are the boldface brackets [ ]. The brackets are used to indicate an option in the address string (see 4.2.1 Optional Operands).

Program Interrupts --

This section of the instruction description names the special interrupt conditions that can occur when one or more of the described operations is executed. If no special interrupt conditions can occur, this topic is omitted. Interrupt conditions that are common to all instructions (see 4.2.6 Program Interrupt) are not included.

Interrupt conditions are listed individually, by class, or by subclass. The class and subclass names refer to the classification of interrupts in 7.6. Program Interrupt Group. When a class or subclass name appears, it indicates that every interrupt condition in that class or subclass can occur when one or more of the operations are executed.

Semantics --

This section gives a description of the instruction execution and how the various operands affect the operations. It does not include a description of steps common to all instructions or to an entire group. Steps common to all instructions are described in 4.2 Common Characteristics. Those common to a group of instructions are discussed before the descriptions of that group.

4.2 COMMON CHARACTERISTICS

Certain features of the instruction set are common to all instructions or to instructions in more than one group of instructions. Those features are explained in the following paragraphs.

4.2.1 Optional Operands

Almost all instructions allow one or more of the possible operands to be omitted. Operands that are not needed for meaningful execution are called optional operands. The omission of an optional operand from an instruction is indicated through null addressing.

In the syntax of an instruction, optional portions are enclosed in brackets. In the following instruction format the operand s4 is optional; it and the keyword 'MASK' may be omitted, and the instruction will still be valid.

IF s2 = s3 [MASK s4] THEN GO TO a1 [iteration]

4.2.2 Implicit Addresses

Many instructions allow the original contents of the destination operand to be used as a source operand. This is called implicit addressing. When implicit addressing is permitted, null addressing in the address string of the instruction, or in the final address when indirect or argument addressing is used, indicates that the source operand is being specified by the destination operand address.

Any source operand that is not defined as an optional operand (see 4.2.1 Optional Operands) can be specified by implicit addressing if the first address in the string is a destination operand. Note that source operands in comparison operations cannot be specified by implicit addressing because the first address is an address operand. In the following example, source operands s2 and s3 are not optional; therefore, they can be specified by implicit addressing.

d1 = s2 + s3 [ADJUST (s4 [, s5])] [iteration]

In the formation of the instruction address string, null addresses that indicate either implicit addressing or omitted optional operands can be excluded from the address string completely if they are not followed by any nonnull addresses. If a nonnull address does appear, all previous null addresses are required in the address string to mark the position of the nonnull address.

4.2.3 Iteration

The arithmetic, magnitude, logical, and miscellaneous byte groups of instructions can be performed iteratively by specifying an iteration count and index increments for appropriate operands. The iteration count is specified by a source operand following the keyword 'PERFORM'. The index increments are specified by a parenthesized list of source operands following the keyword 'INCREMENTS'. One index increment may be specified for each operand in the instruction; therefore, depending on the instruction, there can be 2 to 5 index increments. Each index increment is optional. An omitted index increment is treated as if an index increment with a value of zero had been specified.

When iteration is specified, the instruction is executed the number of times specified by the iteration count except in comparison operations, which are described below. After each execution, every index that is being used to specify the location of an operand is incremented by the corresponding index increment. In this way the successive executions of the instruction can operate with successive elements in an array. When an operand is specified by indirect or argument addressing, only an index in the final address is incremented.

In comparison operations, iteration is discontinued when the comparison condition is met even if the iteration count is not exhausted. If the comparison is true, control is transferred to a location specified in the instruction. Indexes are not incremented when the comparison is true. If the comparison is false, the indexes are incremented. The instruction is then repeated unless the iteration count has been exhausted, in which case control passes to the next sequential instruction.

In all instructions for which iteration is permitted, the iteration operands follow the other operands and are separated from them by a comma. Iteration is optional and the option is indicated in the instruction syntax by enclosing in brackets the metavariable 'iteration' as in the following example:

```
d1 = s2     &  s3     [iteration]
```

If the syntax were not abbreviated by using the metavariable, the option would appear as follows:

```
d1 = s2     &  s3     [, PERFORM s4 INCREMENTS ([s5],
                         [s6], [s7])]
```

The numbering of the iteration operands (s4 through s7 in the above example) varies according to the number of operands that precede them in the instruction format.

A more complete discussion of iteration is contained in 5.0 ITERATIVE EXECUTION.

4.2.4 Suboperation Codes

The exact operation to be performed when an instruction is executed is often specified by a suboperation code. Suboperation codes are always source operands. Suboperation codes are interpreted bit by bit rather than as numeric values. Some bits may be ignored depending on the settings of other bits of the code. Only a certain number of leading bits in the specified operand are examined. All other bits are ignored. The bits that are examined may have any of the possible bit configurations.

4.2.5 Constraints

The operation code of an instruction constrains the operand lengths, the operand values, and the addressing types which can be specified.

Whenever a constraint is violated, a program interrupt condition occurs. The constraints associated with a particular operation are discussed with the description of the operation. Following is a list of the constraints which are common to instruction execution.

Mode Constraints --

The segment number register code constraints which are dependent on central processor mode are discussed in 2.2.3 Access Mode.

Length Code Constraints --

The length code is constrained by the addressing type, and by the data format required for the operation. The constraints associated with an addressing type are discussed in 3.1 Address Formats, and the data formats are discussed in 2.3 Data and Instruction Formats.

Because the length of a control block is a function of the operation being performed, the address which specifies a control block must have an associated length code of 0. Also, whenever an address specifies a location for the transfer of control, the length code must be 0.

Destination Operand Constraints --

A destination operand cannot be specified by null or immediate addressing.

Iteration Constraints --

The iteration count operand cannot exceed $2^{32}-1$. The index increment operands are assumed to be of ADDRESS data type.

Control Transfer Constraints --

A control transfer location cannot be specified by null or immediate addressing.

The segment number register code associated with the control transfer location is constrained by the central processor mode as follows:

- For central processor mode 0 (private mode), only segment number register code 0 (private segment) can be specified .

- For central processor mode 1, only segment number register code 1 or 2 can be specified .

- For central processor modes 2 and 3, only segment number register code 2 or 3 can be specified.

4.2.6 Program Interrupt

During instruction execution, conditions associated with the program interrupt group can occur. Those conditions which pertain to an individual instruction are listed in the description of the instruction, and those conditions which pertain to all instructions are listed in the following paragraphs. Program interrupts are further described in 7.4 Interrupt Processing and in 7.6 Program Interrupt Group.

Page Assignment Class --

These interrupt conditions occur when an attempt is made to write into an unassigned virtual address or to make any kind of access to a page currently in secondary storage. Generally, a real page assignment is made automatically by a service program.

Mode Violation Class --

These conditions occur when an attempt is made to execute a privileged instruction or to address memory while the central processor mode or the current access mode is less than required.

Illegal Addressing Class --

These conditions occur when a read access is attempted from an unassigned virtual address or a write access to a page protected against write access.

Illegal Instruction Class --

These conditions occur when an attempt is made to execute a nonexistent standard or extended operation code or when an instruction qualifier, address qualifier, argument index, or subprogram control block is inconsistent with the operation code being executed or the effective addressing calculation being made.

Address Bounds Breakpoint Class --

These conditions occur when an access is attempted to an address within the bounds specified by an active address bounds register.

Instruction Breakpoint Class --

These conditions occur when the various breakpoint conditions are satisfied and the appropriate controls in the processor state vector and instruction qualifier are set. The breakpoint controls are discussed in 6.0 <u>EXECUTION BREAKPOINTS</u>.

Critical Operand Class --

These conditions occur when an operand specifies a value that is inconsistent with the operation codes of the instruction.

4.3 COMPUTATIONAL ARITHMETIC

There are 8 types of computational arithmetic operations corresponding to the 8 computational data types, which are described in 2.3.3 Arithmetic Operands. These are a complete set of the data types with BINARY or DECIMAL base, FIXED or FLOAT scale, and REAL or COMPLEX mode.

For each computational data type there is a complete set of the following operations: replace, add, subtract, multiply, divide, absolute value, square, inner product, polynomial, if equal, and if not equal. For the 4 REAL types only, there are the additional operations square root, if less than, and if not less than.

Before the result of a computational operation is stored into memory or used in a comparison determination, it is adjusted and truncated. Adjustment is according to one of 3 methods, automatic, normalized, or manual, depending on controls in the processor state vector and in adjustment operands that optionally appear in the instruction. Truncation consists of either simply chopping off the low-order digits of the result or rounding the portion to be retained before chopping off the low-order digits. Controls in the processor state vector and in adjustment operands specify whether or not rounding is done. Rounding is never done in comparison operations.

For each data type there is a set of 2 control bits in the processor state vector, a normalization control and a rounding control. An instruction can override the current settings of these controls by including the optional adjustment operands.

The following sections are an introduction to GEMINI computational arithmetic.

4.3.1 Intermediate Results

All computational arithmetic operations are carried out to produce intermediate results that represent the high order portion of the true result and which are contained in a 116 bit register for the fixed-point part and a separate register for the exponent part. The fixed-point part consists of a sign digit position and 115 binary or 28 decimal digits. Two separate results are produced for the real and imaginary components of complex numbers. The intermediate results are adjusted by aligning both fixed-point and exponent parts according to one of three adjustment methods: normalized, automatic, and manual. The adjusted result is then truncated either by rounding or chopping off low-order digits to produce the final result within a target length. Target length, adjustment methods, and truncation methods are all described in 4.3.2 through 4.3.9.

4.3.2 Target Length

For assignment operations (those in which the value of the result is stored into a destination operand) the target length is the length of the destination operand. For comparison operations the target length is the length of the longer of the two source operands being compared unless adjustment operands are included in the instruction and specify an explicit target length. For complex operations the target length is the same for both real and imaginary components.

4.3.3 Normalized Representation

A REAL data item or a component of a COMPLEX data item is normalized if it has a non-zero digit in the high-order digit position of the fixed-point part or if it is an absolute zero (see 4.3.4 Absolute Zero and Relative Zero). Normalized representation provides a unique way of representing any value of a given data type.

4.3.4 Absolute Zero and Relative Zero

Absolute zero consists of a fixed-point part with a negative sign and all zero digits. The exponent part may have any value, but only an exponent value of +0 is returned by the computational operations for an absolute zero result. Absolute zero is understood to represent zero exactly and is distinguished from relative zero, which is understood to represent zero approximately. Relative zero is represented by a fixed-point part with a positive sign and all zero digits and an exponent part with any legitimate value. In computational operations a relative zero source operand is treated like any non-zero operand with regard to alignment of operands and determination of the exponent of the result.

4.3.5 Automatic Adjustment

When the automatic adjustment method is specified, adjustment of the intermediate result is made according to the exponents and the number of high-order zeros in the source operands. The kind of adjustment made depends on the nature of the operation.

The adjustments are made to provide automatic monitoring of significant digits in the fixed-point part of the result. Automatic significant digit monitoring uses the binary or decimal point in the high-order position of the fixed-point part as a fixed reference point. The varying operand lengths are all treated as if they were shorthand for the longest operand length with low-order digits omitted. The program, however, can use the varying operand lengths to specify how much of the intermediate result is to be stored, including high-order zeros between the decimal or binary point and the first significant digit. The purpose of the automatic adjustments is simply to assure that the reliability of the results of computational operations can be interpreted in terms of the reliability of the source operands.

The automatic adjustments are made as follows, where $\underline{e}_0$, $\underline{e}_1$, $\underline{e}_2$ represent the exponents of the destination operand and the two source operands, respectively; and $\underline{m}_0$, $\underline{m}_1$, $\underline{m}_2$ represent the number of high-order zeros in the fixed-point part of the destination and source operands, respectively. Note that any adjustment of the number of high-order zeros implies a corresponding change in the exponent, and vice versa, so that the same value is maintained while the representation is changed.

- Addition/Subtraction. The source operand with the smaller exponent is adjusted to have the same exponent as the source operand before the addition or subtraction; the result, therefore, has $\underline{e}_0$ equal to the maximum of $\underline{e}_1$ and $\underline{e}_2$. Exception: When one of the source operands is an absolute zero, no adjustment is made on the other source operand.

- Multiplication. The result of multiplication is adjusted to have the same number of high-order zeros as the source operand with the greater number of high-order zeros; the result, therefore, has $\underline{m}_0$ equal to the maximum of $\underline{m}_1$ and $\underline{m}_2$. Exception: When one of the source operands is an absolute zero, the result is absolute zero.

- Division. The result of the division is adjusted to have the same number of high-order zeros as the source operand with the greater number of high-order zeros; the result, therefore, has $\underline{m}_0$ equal to the maximum of $\underline{m}_1$ and $\underline{m}_2$. Exceptions: An absolute zero division causes a Divide by Zero interrupt condition; a relative zero source operand causes a relative zero result with an exponent set as follows:

$$\underline{e}_0 = \underline{e}_1 - \underline{e}_2 - \underline{m}_1 + 2\underline{m}_2 \text{ if } \underline{m}_1 < \underline{m}_2.$$
$$\underline{e}_0 = \underline{e}_1 - \underline{e}_2 + \underline{m}_1 \text{ if } \underline{m}_1 \geq \underline{m}_2.$$

- Square Root. The result is adjusted to have one less high-order zero than the source operand; the result, therefore, has $\underline{m}_0$ equal to $\underline{m}_1 - 1$.

- Square. The result is adjusted to have one more high-order zero than the source operand; the result, therefore, has $\underline{m}_0$ equal to $\underline{m}_1 + 1$.

- Composite Operations. Inner product, polynomial, and all the COMPLEX operations are composites of two or more of the above operations and an automatic adjustment is made after each step of the operation. Exception: In COMPLEX division, the steps to square the components of the divisor and add the squares together always have normalized adjustment.

- Comparison Operations. The comparison is done by means of a subtraction operation; the fixed-point part of the result is adjusted by setting to zeros all the low-order digits beyond the length of the longer source operand. The relation between the two source operands is determined on the basis of the adjusted result.

4.3.6 Normalized Adjustment

When the normalized adjustment method is used, the intermediate result is adjusted to the normalized representation. All zero results are expressed as an absolute zero. All non-zero results are adjusted to have a non-zero digit in the high-order digit position of the fixed-point part, that is $\underline{m}_0$ equals 0.

4.3.7 Manual Adjustment

When the manual adjustment method is used, the instruction specifies an explicit adjustment source operand and the result is unconditionally adjusted as specified. The specified adjustment is either in terms of the exponent, $\underline{e}$; or the number of high-order zeros _m_; or, for comparison operations instead of _m_, the target length in digits, _t_. The specification is either in terms of a specific _e_, _m_, or _t_ value, or a value relative to the one that would result if the automatic method were being used.

4.3.8 Truncation

Truncation is performed on the intermediate result only as the last step before storing the result into the destination operand or determining a relation in a comparison operation. There are two methods of truncation, depending on whether or not a rounding operation is included:

Chopping --

When chopping is specified, low-order digits of the fixed-point part of the intermediate result are unconditionally set to zeros. Only the high-order digits that make up the target length are unaffected by the chopping.

Rounding --

When rounding is specified, one of two methods is used, decimal rounding or binary rounding. In each method the 4 bits (that is 1 decimal digit or 4 binary digits) immediately to the low-order side of the target length are examined.

In decimal rounding, if the decimal digit is between 5 and 9, 1 is added to the magnitude of the fixed-point part at the low-order digit position of the target length; carries are propagated through the entire target length, and a carry out of the high-order significant digit causes a one digit shift to the low-order end with a corresponding increase in the exponent. This is the traditional rounding method.

In binary rounding, if any of the 4 bits are 1's, the digit in the low-order position of the target length is unconditionally set to 1. There are no carries propagated.

Once rounding has been completed, the low-order digits beyond the target length are set unconditionally to zeros, as in chopping.

4.3.9 Specification of Adjustment and Truncation Method

The adjustment method and truncation method to be used in executing a computational arithmetic operation are specified either by the normalization and rounding controls in the processor state vector or by the optional adjustment operands in the instruction. If the adjustment operands are omitted, the processor state vector normalization and rounding controls for the data type of the operation specify adjustment and truncation methods. If the normalization bit is set, the normalized method is used; if the normalization bit is reset, the automatic method is used. If the rounding bit is set, decimal or binary rounding, depending on the base of the data type, is used; if the rounding bit is reset, only chopping is used.

If the adjustment operands are present, they are interpreted as indicated below to specify adjustment and truncation according to processor state vector settings or explicit method.

In all computational instructions optional adjustment operands are permitted. They consist of the keyword 'ADJUST' followed by 2 source operands which are separated by a comma and enclosed in parentheses. The first of the source operands specifies the adjustment suboperation code. The comma and second source operand are needed only to specify a manual adjustment.

The adjustment suboperation code is the leading 7 bits of the first adjustment operand. Of these the leading 5 bits specify the adjustment method, and the other 2 bits control rounding.

Bit Position	Bit Value 0	Bit Value 1
0	PSV Adjustment Control	Instruction Adjustment Control
1 & 2	0 0 Automatic Method 0 1 Normalized Method 1 0 Manual *e* Adjust 1 1 Manual *m* or *t* Adjust	
3	Specific Value	Relative Value
4	Adjustment Value	Adjustment Prototype
5	PSV Rounding Control	Instruction Rounding Control
6	Chopping	Rounding

Figure 4.1 - Adjustment Suboperation Codes

Bit 0 specifies whether the adjustment method is determined from the processor state vector normalization control bit or from bits 1 and 2 of the adjustment suboperation code. Bits 1 and 2 specify whether the automatic method, the normalized method, or one of the manual methods is to be used. The 2 manual methods are exponent adjustment and high-order zero or target length adjustment. If a manual method is being used, bits 3 and 4 specify how the second adjustment operand is interpreted. Bit 3 indicates whether a _specific value_ or a _relative value_ is specified. A specific value indicates the value to which _e_, _m_, or _t_ is to be adjusted. A relative value specifies an adjustment relative to the value of _e_, _m_, or _t_ that would result from automatic adjustment. Bit 4 indicates whether the second adjustment operand is a signed integer _adjustment value_ or an _adjustment prototype_. A prototype is a data item of the same data type as the operation. Its exponent, high-order zeros, or length is used as a model for aligning the result. A relative _m_ or _t_ prototype always specifies an increase in _m_ or _t_ from the automatic adjustment.

Bit 5 indicates whether rounding is determined by the processor state vector rounding control bit or by bit 6. If bit 5 is 1, bit 6 indicates whether or not rounding is done. The data type of the operation determines whether decimal or binary rounding is meant.

COMPUTATIONAL REAL REPLACE

Operation Codes

Binary Fixed Real Replace
Binary Float Real Replace
Decimal Fixed Real Replace
Decimal Float Real Replace

Syntax

d1 = s2 [ADJUST (s3[, s4])] [iteration]

Program Interrupts

Fixed Point Overflow Class (Real Subclass)
Exponent Overflow Class (Real Subclass)
Illegal Decimal Digit

Semantics

The operand specified by the first address is replaced by the operand specified by the second address.

COMPUTATIONAL REAL ADD

Operation Codes

Binary Fixed Real Add
Binary Float Real Add
Decimal Fixed Real Add
Decimal Float Real Add

Syntax

d1 = s2 + s3   [ADJUST (s4[, s5])] [iteration]

Program Interrupts

Overflow Class (Real Subclass)
Fixed Point Overflow Class (Real Subclass)
Exponent Overflow Class (Real Subclass)
Illegal Decimal Digit

Semantics

The operand specified by the second address is added to the operand specified by the third address. The result replaces the destination operand.

COMPUTATIONAL REAL SUBTRACT

Operation Codes

Binary Fixed Real Subtract
Binary Float Real Subtract
Decimal Fixed Real Subtract
Decimal Float Real Subtract

Syntax

<u>d1</u> = <u>s2</u> - <u>s3</u>   [ADJUST (<u>s4</u>[, <u>s5</u>])] [<u>iteration</u>]

Program Interrupts

Overflow Class (Real Subclass)
Fixed Point Overflow Class (Real Subclass)
Exponent Overflow Class (Real Subclass)
Illegal Decimal Digit

Semantics

The operand specified by the third address is subtracted from the operand specified by the second address. The result replaces the destination operand.

COMPUTATIONAL REAL MULTIPLY

Operation Codes

Binary Fixed Real Multiply
Binary Float Real Multiply
Decimal Fixed Real Multiply
Decimal Float Real Multiply

Syntax

d1 = s2 * s3   [ADJUST (s4[, s5])] [iteration]

Program Interrupts

Overflow Class (Real Subclass)
Fixed Point Overflow Class (Real Subclass)
Exponent Overflow Class (Real Subclass)
Illegal Decimal Digit

Semantics

The operand specified by the second address is multiplied by the operand specified by the third address. The result replaces the destination operand.

COMPUTATIONAL REAL DIVIDE

Operation Codes

Binary Fixed Real Divide
Binary Float Real Divide
Decimal Fixed Real Divide
Decimal Float Real Divide

Syntax

d1 = s2 / s3   [ADJUST (s4[, s5])] [iteration]

Program Interrupts

Overflow Class (Real Subclass)
Fixed Point Overflow Class (Real Subclass)
Exponent Overflow Class (Real Subclass)
Divide by Zero Class (Real Subclass)
Illegal Decimal Digit

Semantics

The operand specified by the second address is divided by the operand specified by the third address. The quotient replaces the destination operand.

COMPUTATIONAL REAL SQUARE

Operation Codes

Binary Fixed Real Square
Binary Float Real Square
Decimal Fixed Real Square
Decimal Float Real Square

Syntax

```
d1 = square (s2)    [ADJUST(s3[,s4])]  [iteration]
```

Program Interrupts

Overflow Class (Real Subclass)
Fixed Point Overflow Class (Real Subclass)
Exponent Overflow Class (Real Subclass)
Illegal Decimal Digit

Semantics

The operand specified by the second address is squared and the result replaces the destination operand.

COMPUTATION REAL SQUARE ROOT

Operation Codes

Binary Fixed Real Square Root
Binary Float Real Square Root
Decimal Fixed Real Square Root
Decimal Float Real Square Root

Syntax

$\underline{\underline{d1}}$ = sqrt ($\underline{\underline{s2}}$)   [ADJUST ($\underline{\underline{s3}}$[,$\underline{\underline{s4}}$])]   [iteration]

Program Interrupts

Fixed Point Overflow Class (Real Subclass)
Exponent Overflow Class (Real Subclass)
Square Root Source Operand Negative
Illegal Decimal Digit

Semantics

The square root of the operand specified by the second address is calculated. The result replaces the destination operand.

COMPUTATIONAL REAL IF EQUAL

Operation Codes

Binary Real If Equal
Decimal Real If Equal

Syntax

IF s2 = s3 [ADJUST(s4[,s5])] THEN GO TO a1 [iteration]

Program Interrupts

Illegal Decimal Digit

Semantics

The operand specified by the second address is compared with the operand specified by the third address. If the comparison is true, control is transferred to the location specified by the first address; otherwise no transfer occurs.

COMPUTATIONAL REAL IF NOT EQUAL

Operation Codes

Binary Real If Not Equal
Decimal Real If Not Equal

Syntax

IF s2 ¬= s3 [ADJUST(s4[,s5])] THEN GO TO a1 [iteration]

Program Interrupts

Illegal Decimal Digit

Semantics

The operand specified by the second address is compared with the operand specified by the third address. If the comparison is true, control is transferred to the location specified by the first address; otherwise no transfer occurs.

COMPUTATIONAL REAL IF LESS

Operation Codes

Binary Real If Less
Decimal Real If Less

Syntax

IF s2 < s3 [ADJUST(s4[,s5])] THEN GO TO a1 [iteration]

Program Interrupts

Illegal Decimal Digit

Semantics

The operand specified by the second address is compared with the operand specified by the third address. If the comparison is true, control is transferred to the location specified by the first address; otherwise no transfer occurs.

COMPUTATIONAL REAL IF NOT LESS

Operation Codes

Binary Real If Not Less
Decimal Real If Not Less

Syntax

IF s2 ¬ < s3 [ADJUST(s4[,s5])] THEN GO TO a1 [iteration]

Program Interrupts

Illegal Decimal Digit

Semantics

The operand specified by the second address is compared with the operand specified by the third address. If the comparison is true, control is transferred to the location specified by the first address; otherwise no transfer occurs.

COMPUTATIONAL COMPLEX REPLACE

Operation Codes

Binary Fixed Complex Replace
Binary Float Complex Replace
Decimal Fixed Complex Replace
Decimal Float Complex Replace

Syntax

d1 = s2 [ADJUST(s3[,s4])] [iteration]

Program Interrupts

Fixed Point Overflow Class (Complex Subclass)
Exponent Overflow Class (Complex Subclass)
Illegal Decimal Digit

Semantics

The operand specified by the second address replaces the destination operand.

COMPUTATIONAL COMPLEX ADD

Operation Codes

Binary Fixed Complex Add
Binary Float Complex Add
Decimal Fixed Complex Add
Decimal Float Complex Add

Syntax

d1 = s2 + s3   [ADJUST(s4[, s5])] [iteration]

Program Interrupts

Overflow Class (Complex Subclass)
Fixed Point Overflow Class (Complex Subclass)
Exponent Overflow Class (Complex Subclass)
Illegal Decimal Digit

Semantics

The operand specified by the third address is added to the operand specified by the second address. The result of the addition replaces the destination operand.

COMPUTATIONAL COMPLEX SUBTRACT

Operation Codes

Binary Fixed Complex Subtract
Binary Float Complex Subtract
Decimal Fixed Complex Subtract
Decimal Float Complex Subtract

Syntax

$\underline{d1} = \underline{s2} - \underline{s3}$ [ADJUST($\underline{s4}$[,$\underline{s5}$])] [iteration]

Program Interrupts

Overflow Class (Complex Subclass)
Fixed Point Overflow Class (Complex Subclass)
Exponent Overflow Class (Complex Subclass)
Illegal Decimal Digit

Semantics

The operand specified by the third address is subtracted from the operand specified by the second address. The result of the subtraction replaces the destination operand.

COMPUTATIONAL COMPLEX MULTIPLY

Operation Codes

Binary Fixed Complex Multiply
Binary Float Complex Multiply
Decimal Fixed Complex Multiply
Decimal Float Complex Multiply

Syntax

d1 = s2* s3   [ADJUST(s4[,s5])] [iteration]

Program Interrupts

Overflow Class (Complex Subclass)
Fixed Point Overflow Class (Complex Subclass)
Exponent Overflow Class (Complex Subclass)
Illegal Decimal Digit

Semantics

The operand specified by the second address is multiplied by the operand specified by the third address. The result of the multiplication replaces the destination operand.

COMPUTATIONAL COMPLEX DIVIDE

Operation Codes

Binary Fixed Complex Divide
Binary Float Complex Divide
Decimal Fixed Complex Divide
Decimal Float Complex Divide

Syntax

d1 = s2 / s3   [ADJUST(s4[,s5])] [iteration]

Program Interrupts

Overflow Class (Complex Subclass)
Fixed Point Overflow Class (Complex Subclass)
Exponent Overflow Class (Complex Subclass)
Divide by Zero Class (Complex Subclass)
Illegal Decimal Digit

Semantics

The operand specified by the second address is divided by the operand specified by the third address. The division is performed and the quotient replaces the destination operand.

COMPUTATIONAL COMPLEX ABSOLUTE VALUE

Operation Codes

Binary Fixed Complex Absolute Value
Binary Float Complex Absolute Value
Decimal Fixed Complex Absolute Value
Decimal Float Complex Absolute Value

Syntax

d1 = abs (s2) [ADJUST(s3 [,s4])] [iteration]

Program Interrupts

Overflow Class (Complex Subclass)
Fixed Point Overflow Class (Complex Subclass)
Exponent Overflow Class (Complex Subclass)
Illegal Decimal Digit

Semantics

The absolute value of the complex operand specified by the second address is calculated and replaces the destination operand.

The result is the positive square root of the sum of squares of the real and the imaginary parts. The result is a REAL data type with the same base (BINARY or DECIMAL) and scale (FIXED or FLOAT) as the COMPLEX source operand.

COMPUTATIONAL COMPLEX IF EQUAL

Operation Codes

Binary Complex If Equal
Decimal Complex If Equal

Syntax

```
IF s2 = s3 [ADJUST(s4[,s5])] THEN GO TO a1 [iteration]
```

Program Interrupts

Illegal Decimal Digit

Semantics

The operand specified by the second address is compared with the operand specified by the third address. If the comparison is true, control is transferred to the location specified by the first address; otherwise no transfer occurs.

COMPUTATIONAL COMPLEX IF NOT EQUAL

Operation Codes

Binary Complex If Not Equal
Decimal Complex If Not Equal

Syntax

IF <u>s2</u> ¬ = <u>s3</u> [ADJUST(<u>s4</u> [,<u>s5</u>])] THEN GO TO <u>a1</u> [<u>iteration</u>]

Program Interrupts

Illegal Decimal Digit

Semantics

The operand specified by the second address is compared with the operand specified by the third address. If the comparison is true, control is transferred to the location specified by the first address; otherwise no transfer occurs.

4.4 INTEGER OPERATIONS

Integer arithmetic operations allow arithmetic to be performed using signed binary or decimal integers.

Constraints --

An integer can be from 1 to 16 bytes long. See 2.3.3 Arithmetic Operands for the format discussion. An integer value of -0 is accepted, but is then converted to +0. A value of -0 is not generated by the integer operations.

Common Operations --

Before performance of an operation, the following steps occur:

- The source operands are positioned so that the lowest order digits are aligned.
- If necessary, the source operands are expanded to the same length as the longest operand by inserting 0 digits between the sign digit and the high order digit.

Before storing the result of an operation, the following steps occur:

- The result and the destination operand are positioned so that the low order digits are aligned.
- If the length of the result is greater than the length of the destination, the result is shortened to the same length as the destination by deleting high order digits. If any nonzero digit is deleted, the integer overflow interrupt condition corresponding to the data type occurs.

- The comparison operations subtract the second source source operand from the first before making the test. If the optional mask is specified, the subtraction is performed only with those bit positions corresponding to a 1 in the mask.

INTEGER REPLACE

Operation Codes

Binary Integer Replace
Decimal Integer Replace

Syntax

$\underline{\text{d1}} = \underline{\text{s2}}$ [iteration]

Program Interrupts

Overflow Class (Integer Subclass)
Illegal Decimal Digit

Semantics

The operand specified by the second address replaces the destination operand.

INTEGER ADD

Operation Codes

Binary Integer Add
Decimal Integer Add

Syntax

d1 = s2 + s3   [iteration]

Program Interrupts

Overflow Class (Integer Subclass)
Illegal Decimal Digit

Semantics

The operand specified by the third address is added to the operand specified by the second address. The result of the addition replaces the destination operand.

INTEGER SUBTRACT

Operation Codes

Binary Integer Subtract
Decimal Integer Subtract

Syntax

d1 = s2 - s3   [iteration]

Program Interrupts

Overflow Class (Integer Subclass)
Illegal Decimal Digit

Semantics

The operand specified by the third address is subtracted from the operand specified by the second address. The result of the subtraction replaces the destination operand.

INTEGER MULTIPLY

Operation Codes

Binary Integer Multiply
Decimal Integer Multiply

Syntax

d1 = s2 * s3   [iteration]

Program Interrupts

Overflow Class (Integer Subclass)
Illegal Decimal Digit

Semantics

The operand specified by the second address is multiplied by the operand specified by the third address. The result of the multiplication replaces the destination operand.

INTEGER DIVIDE

Operation Codes

Binary Integer Divide
Decimal Integer Divide

Syntax

d1 = s2 / s3   [iteration]

Program Interrupts

Overflow Class (Integer Subclass)

Illegal Decimal Digit

Semantics

The operand specified by the second address is divided by the operand specified by the third address. The quotient replaces the destination operand.

INTEGER REMAINDER

Operation Codes

Binary Integer Remainder
Decimal Integer Remainder

Syntax

d1 = remainder (s2, s3)   [iteration]

Program Interrupts

Overflow Class (Integer Subclass)
Divide by Zero Class (Integer Subclass)
Illegal Decimal Digit

Semantics

The operand specified by the second address is divided by the operand specified by the third address. The remainder replaces the destination operand. The sign of the remainder is set to be the same as the sign of the operand specified by the second address.

INTEGER SCALE

Operation Codes

Binary Integer Scale
Decimal Integer Scale

Syntax

$\underline{d1}$ = scale ($\underline{s2}$, $\underline{s3}$) [$\underline{\text{iteration}}$]

Program Interrupts

Overflow Class (Integer Subclass)
Illegal Displacement
Illegal Decimal Digit

Semantics

The operand to be scaled is specified by the second address, and the scale factor operand is specified by the third address. The result of the scaling operation replaces the destination operand.

The scale factor operand is a binary or decimal integer depending on the type of integer operation being performed. The sign of the scale factor operand determines the direction of the scaling operation. When the sign is positive, the scale operand is multiplied by $\underline{r}^{\underline{n}}$, and when it is negative, the scale operand is divided by $\underline{r}^{\underline{n}}$. The value of $\underline{r}$ is 2 or 10 depending on the type of integer operation being performed, and $\underline{n}$ is the absolute value of the scale factor operand.

INTEGER IF EQUAL

Operation Codes

Binary Integer If Equal
Decimal Integer If Equal

Syntax

IF _s2_ = _s3_ [MASK _s4_] THEN GO TO _a1_ [_iteration_]

Program Interrupts

Illegal Decimal Digit

Semantics

The operand specified by the second address is compared with the operand specified by the third address. If the comparison is true, control is transferred to the location specified by the first address; otherwise no transfer occurs. If the optional mask is specified by the fourth address the comparison is performed only with those digit positions corresponding to a 1 in the mask.

INTEGER IF NOT EQUAL

Operation Codes

Binary Integer If Not Equal
Decimal Integer If Not Equal

Syntax

IF s2 ¬ = s3 [MASK s4] THEN GO TO a1 [iteration]

Program Interrupts

Illegal Decimal Digit

Semantics

The operand specified by the second address is compared with the operand specified by the third address. If the comparison is true, control is transferred to the location specified by the first address; otherwise no transfer occurs. If the optional mask is specified by the fourth address, the comparison is performed only with those digit positions corresponding to 1 in the mask.

INTEGER IF LESS

Operation Codes

Binary Integer If Less
Decimal Integer If Less

Syntax

IF $\underline{s2}$ < $\underline{s3}$ [MASK $\underline{s4}$] THEN GO TO $\underline{a1}$ [$\underline{\text{iteration}}$]

Program Interrupts

Illegal Decimal Digit

Semantics

The operand specified by the second address is compared with the operand specified by the third address. If the comparison is true, control is transferred to the location specified by the first address; otherwise no transfer occurs. If the optional mask is specified by the fourth address, the comparison is performed only with those digit positions corresponding to a 1 in the mask.

INTEGER IF NOT LESS

Operation Codes

Binary Integer If Not Less
Decimal Integer If Not Less

Syntax

IF s2 ¬ < s3 [MASK s4] THEN GO TO a1 [iteration]

Program Interrupts

Illegal Decimal Digit

Semantics

The operand specified by the second address is compared with the operand specified by the third address. If the comparison is true, control is transferred to the location specified by the first address; otherwise no transfer occurs. If the optional mask is specified by the fourth address, the comparison is performed only with those digit positions corresponding to a 1 in the mask.

4.5 MAGNITUDE OPERATIONS

Magnitude operations allow arithmetic to be performed using unsigned magnitude binary integers.

There are 2 types of magnitude arithmetic operations: magnitude and address. The operations differ only in the length of the operands involved in the calculation.

Constraints --

Both magnitude and address operands can vary from 1 to 16 bytes in length. For address operations, however, the 2 trailing bits are not used during address operations. The format of these 2 data types are discussed in 2.3.3 Arithmetic Operands.

Common Operations --

Magnitude arithmetic operations perform the specified arithmetic operation and then express the result modulo $2^{\underline{n}}$ where $\underline{n}$ is the number of bits in the destination operand.

Before performance of an operation, the following steps occur:

- The source operands are positioned so that the lowest order digits are aligned.
- If necessary, the source operands are expanded to the same length as the longest operand by inserting 0 digits.

Before storing the result of an operation, the following steps occur:

- The result and the destination operand are positioned so that the low order digits are aligned.
- If the length of the result is greater than the length of the destination, the result is shortened to the same length as the destination by deleting high order digits. If any nonzero digit is deleted, the integer overflow interrupt condition corresponding to the data type occurs.
- The comparison operations subtract the second source operand from the first before making the test. If the optional mask is specified, the subtraction is performed only with those bit positions corresponding to a 1 in the mask.

MAGNITUDE REPLACE

Operation Codes

Magnitude Replace
Address Replace

Syntax

d1 = s2 [iteration]

Program Interrupts

Truncation Class (Magnitude Subclass)

Semantics

The operand specified by the second address replaces the destination operand.

MAGNITUDE ADD

Operation Codes

Magnitude Add
Address Add

Syntax

d1 = s2 + s3   [iteration]

Program Interrupts

Truncation Class (Magnitude Subclass)

Semantics

The operand specified by the third address is added to the operand specified by the second address. The result of the addition replaces the destination operand.

MAGNITUDE SUBTRACT

Operation Codes

Magnitude Subtract
Address Subtract

Syntax

d1 = s2 - s3   [iteration]

Program Interrupts

Truncation Class (Magnitude Subclass)

Semantics

The operand specified by the third address is subtracted from the operand specified by the second address. The result of the subtraction replaces the destination operand.

MAGNITUDE MULTIPLY

Operation Codes

Magnitude Multiply
Address Multiply

Syntax

d1 = s2 * s3   [iteration]

Program Interrupts

Truncation Class (Magnitude Subclass)

Semantics

The operand specified by the second address is multiplied by the operand specified by the third address. The result of the multiplication replaces the destination operand.

MAGNITUDE DIVIDE

Operation Codes

Magnitude Divide
Address Divide

Syntax

d1 = s2 / s3   [iteration]

Program Interrupts

Truncation Class (Magnitude Subclass)
Divide by Zero Class (Magnitude Subclass)

Semantics

The operand specified by the second address is divided by the operand specified by the third address. The quotient replaces the destination operand.

MAGNITUDE REMAINDER

Operation Codes

Magnitude Remainder
Address Remainder

Syntax

d1 = remainder (s2, s3) [iteration]

Program Interrupts

Truncation Class (Magnitude Subclass)
Divide by Zero Class (Magnitude Subclass)

Semantics

The operand specified by the second address is divided by the operand specified by the third address. The remainder replaces the destination operand.

MAGNITUDE SCALE

Operation Codes

Magnitude Scale
Address Scale

Syntax

<u>d1</u> = scale (<u>s2</u>,<u>s3</u>) [<u>iteration</u>]

Program Interrupts

Truncation Class (Magnitude Subclass)
Illegal Displacement

Semantics

The operand to be scaled specified by the second address is scaled by a factor 2^n where n the scale factor operand is specified by the third address. The reult of the scaling operation replaces the destination operand.

The scale factor operand n is a binary integer and its sign determines the direction of the scaling operation. When the scale factor operand is positive, the scale operand is multiplied by 2^n, but when it is negative, the scale operand is divided by 2^n. The value of n is the absolute value of the scale factor operand.

MAGNITUDE IF EQUAL

Operation Codes

Magnitude If Equal
Address If Equal

Syntax

IF s2 = s3 [MASK s4] THEN GO TO a1 [iteration]

Semantics

The operand specified by the second address is compared with the operand specified by the third address. If the comparison is true, control is transferred to the location specified by the first address; otherwise no transfer occurs. If the optional mask is specified by the fourth address, the comparison is performed only with those digit positions corresponding to a 1 in the mask.

MAGNITUDE IF NOT EQUAL

Operation Codes

Magnitude If Not Equal
Address If Not Equal

Syntax

IF s2 ¬= s3 [MASK s4] THEN GO TO a1 [iteration]

Semantics

The operand specified by the second address is compared with the operand specified by the third address. If the comparison is true, control is transferred to the location specified by the first address; otherwise no transfer occurs. If the optional mask is specified by the fourth address, the comparison is performed only with those digit positions corresponding to a 1 in the mask.

MAGNITUDE IF LESS

Operation Codes

Magnitude If Less
Address If Less

Syntax

IF s2 < s3 [MASK s4] THEN GO TO a1 [iteration]

Semantics

The operand specified by the second address is compared with the operand specified by the third address. If the comparison is true, control is transferred to the location specified by the first address; otherwise no transfer occurs. If the optional mask is specified by the fourth address, the comparison is performed only with those digit positions corresponding to a 1 in the mask.

MAGNITUDE IF NOT LESS

Operation Codes

Magnitude If Not Less
Address If Not Less

Syntax

IF s2 ¬ < s3 [MASK s4] THEN GO TO a1 [iteration]

Semantics

The operand specified by the second address is compared with the operand specified by the third address. If the comparison is true, control is transferred to the location specified by the first address; otherwise no transfer occurs. If the optional mask is specified by the fourth address, the comparison is performed only with those digit positions corresponding to a 1 in the mask.

4.6 LOGICAL OPERATIONS

Common Characteristics --

A logical operand is a byte string from 1 to 16 bytes long. Operands are aligned at the leading bit positions before performing an operation. For all operations, whenever the 2 source operands are of different lengths, 0's are extended through the trailing bits of the shorter operand until both operands are the same length.

After the operation is performed, one of the following steps may be performed:

- When the length of the result is less than the length of the destination, the result is stored and the trailing bits of the destination are set to 0.
- When the length of the result exceeds the length of the destination, the trailing bytes of the result are discarded. Note that the logical overflow interrupt condition does not occur if all of the truncated bits are 0.

LOGICAL REPLACE

Syntax

    d1 = s2 [iteration]

Program Interrupts

    Logical Truncation

Semantics

    The operand specified by the second address replaces the destination operand.

LOGICAL AND

Syntax

$\underline{d1} = \underline{s2}$ & $\underline{s3}$ [iteration]

Program Interrupts

Logical Truncation

Semantics

An *and* is performed between the corresponding bits of the operands specified by the second and third addresses. The result of the operation replaces the destination operand.

The *and* function is performed according to the following truth table:

$d1_n$	= $s2_n$	& $s3_n$
0	0	0
0	0	1
0	1	0
1	1	1

where *n* is the bit position.

LOGICAL INCLUSIVE OR

Syntax

$\underline{d1} = \underline{s2} \mid \underline{s3}$ [iteration]

Program Interrupts

Logical Truncation

Semantics

An inclusive or is performed between the corresponding bits of the operands specified by the second and third addresses. The result of the operation replaces the destination operand.

The inclusive or function is performed according to the following truth table:

$d1_n$ =	$s2_n$	$s3_n$
0	0	0
1	0	1
1	1	0
1	1	1

where $\underline{n}$ is the bit position.

LOGICAL EXCLUSIVE OR

Syntax

d1 = s2 $\overline{¦}$ s3 [iteration]

Program Interrupts

Logical Truncation

Semantics

An exclusive or is performed between the corresponding bits of the operands specified by the second and third addresses. The result of the operation replaces the destination operand.

The exclusive or function is performed according to the following truth table:

$d1_n$ =	$s2_n$	$s3_n$
0	0	0
1	0	1
1	1	0
0	1	1

where n is the bit position.

LOGICAL NOT AND

Syntax

$\underline{d1} = \underline{s2} \neg \& \underline{s3}$ [$\underline{\text{iteration}}$]

Program Interrupts

Logical Truncation

Semantics

A _not and_ is performed between the corresponding bits of operands specified by the second and third addresses. The result of the operation replaces the destination operand.

The _not and_ function is performed according to the following truth table:

$\underline{d1}_{\underline{n}}$ =	$\underline{s2}_{\underline{n}}$	$\underline{s3}_{\underline{n}}$
1	0	0
1	0	1
1	1	0
0	1	1

where _n_ is the bit position.

LOGICAL NOT INCLUSIVE OR

Syntax

$\underline{d1} = \underline{s2} \neg | \underline{s3}$ [iteration]

Program Interrupts

Logical Truncation

Semantics

A not inclusive or is performed between the corresponding bits of operands specified by the second and third address. The result of the operation replaces the destination operand.

The not inclusive or function is performed according to the following truth table:

$d1_n$ =	$s2_n$	$s3_n$
1	0	0
0	0	1
0	1	0
0	1	1

where n is the bit position.

LOGICAL NOT EXCLUSIVE OR

Syntax

$\underline{d1} = \underline{s2} \neg \mid \underline{s3}$ [iteration]

Program Interrupts

Logical Truncation

Semantics

A not exclusive or is performed between the corresponding bits of operands specified by the second and third address. The result of the operation replaces the destination operand.

The not exclusive or function is performed according to the following truth table:

$d1_n$ =	$s2_n$	$s3_n$
1	0	0
0	0	1
0	1	0
1	1	1

where n is the bit position.

4.7 MISCELLANEOUS BYTE OPERATIONS

A group of miscellaneous byte operations are included for operation on logical operands and byte strings. These are SHIFT, COUNT, GENERATE and COPY.

The operands used in these operations are byte strings from 1 to 16 bytes long.

SHIFT

Syntax

```
d1 = shift (s2 CODE s3 COUNT s4) [iteration]
```

Program Interrupts

Logical Truncation
Illegal Shift Count

Semantics

The suboperation operand is specified by the second address, the shift operand is specified by the third address, and the optional shift-count operand is specified by the fourth address. The result of the Shift operation replaces the destination operand.

The suboperation operand indicates how the shift operand is to be manipulated by specifying according to the following code:

Bit Position		
1	0	Bit Value
Shift	Left	0
Cycle	Right	1

Shift Suboperation Codes

The shift-count operand is required for all functions. The shift-count operand is assumed to be a binary integer that specifies the number of single bit shifts to be performed. In addition, a negative shift-count operand inverts the direction of the shift.

In a shift operation, vacated leading or trailing bit positions are filled with 0's and bits shifted off either end of the operand are lost.

In a cycle operation, bits shifted off either end are copied into the vacated leading or trailing bit positions.

COUNT

Operation

Binary Count
Decimal Count

Syntax

d1 = count (s2 CODE s3) [iteration]

Program Interrupts

Overflow Class (Integer Subclass)
Illegal Decimal Digit

Semantics

The number of digits in the second operand are counted according to the specified CODE function. The count result replaces the destination operand.

The suboperation code (third operand) determines the function to be performed as follows:

Bit Value		Bit Position
1	0	
Count 0 Digits	Count Non Zero Digits	0
00 Count Sign 01 Count Contiguous Leading 10 Count Contiguous Trailing 11 Count All		1 and 2
Include Sign Digit	Exclude Sign Digit	3

Count Suboperation Code

For the sign digit plus (+) is considred to have the value 0 and negative (-) is considered to have the value 1.

GENERATE

Syntax

dl = generate (s2 COUNT s3 [DISP s4]) [iteration]

Program Interrupts

Logical Truncation
Illegal Generate Specification
Illegal Displacement

Semantics

The suboperation operand is specified by the second address, the bit-count operand is specified by the third address, and the optional displacement operand is specified by the fourth address.

The suboperation operand indicates that trailing or leading bits in the destination operand are to be set to 0 or 1 according to the following function:

Bit Value	Bit Position 1	Bit Position 0
0	Leading	0's
1	Trailing	1's

Generate Suboperation Codes

The bit count operand is a positive binary integer which specifies the number of contiguous bits to be modified. The optional displacement operand permits modification of contiguous bits beginning with a bit position displaced from leading or trailing according to the value of the operand. The displacement operand is a positive binary integer with a maximum value equal to 1 less than the number of bits in the destination operand. The bit count plus the displacement value cannot specify a bit position outside the destination operand.

For a leading displacement, the displacement operand specifies the initial bit position to be modified. For a trailing displacement, the displacement operand is subtracted from the bit length of the destination operand to specify the initial bit position.

REVERSE

Syntax

d1 = reverse (s2) [iteration]

Program Interrupts

Logical Truncation

Semantics

The contents of the byte string specified by the second address are reversed bit for bit between leading bit positions and trailing bit positions. The result is stored into the destination operand. If the destination operand is shorter than the source operand, trailing bytes of the result are truncated. If the destination operand is longer than the source operand, the result is extended with trailing 0's.

COPY

Syntax

d1 = copy (s2 [DISP s3] [MASK s4]) [iteration]

Program Interrupts

Logical Truncation
Illegal Displacement

Semantics

The operand specified by the second address replaces the destination operand. The optional displacement operand is specified by the third address, the optional mask is specified by the fourth address. The operand specified by the second address replaces the destination operand. If the length of the destination operand is greater than the length of the source operand, the trailing bytes of the destination operand are not affected.

If the optional mask is specified by the fourth address, bits of the source operand are copied into the destination operand only where the corresponding bit position of the mask is 1. The other bits of the destination operand are not affected. The mask is aligned with the leading bit position of the destination operand.

The displacement operand is a binary integer. Its maximum absolute value for a positive displacement is 1 less than the number of bits in the destination operand, and for a negative displacement is 1 less than the number of bits in the source operand. It causes the source operand to be aligned with the destination operand according to the following table:

Displacement Operand Sign	Bit Position		Bit Position
negative	S(d)	aligned with	D(0)
positive	S(0)	aligned with	D(d)

where: S = source operand

D = destination operand

d = absolute value of the displacement operand.

The bits of the source operand thus aligned are copied into the destination operand. Bit positions of the destination operand not aligned with source operand bits are not affected.

When both the mask and the displacement operand are specified, the mask is aligned with the destination operand, and the displaced source operand is copied into the destination only where the corresponding bit position of the mask is 1.

CONVERT

Syntax

d1 = convert (s2 CODE s3) [ADJUST(s4 [,s5]) [iteration]

Program Interrupts

Overflow Class
Truncation Class
Fixed Point Overflow Class
Exponent Underflow Class
Illegal Decimal Digit

Semantics

The second operand is converted according to the suboperation operand specified by the third address. The result replaces the destination operand.

The suboperation operand is 1 byte and represents two 4-bit codes. The leading 4 bits designate the source operand data type, and the trailing 4 bits designate the destination operand data type. The 4-bit codes are interpreted as shown in Table 4.1 Data Type Codes. If either of the 4-bit codes is one of the null values, no action is taken and the destination operand is unchanged.

The conversion occurs in up to 6 steps:

Source acquisition
Mode conversion
Sign conversion
Base conversion
Scale conversion
Destination storage

Data Type Code	Data Type
0	Real Float Decimal
1	Real Float Binary
2	Real Fixed Decimal
3	Real Fixed Binary
4	Complex Float Decimal
5	Complex Float Binary
6	Complex Fixed Decimal
7	Complex Fixed Binary
8	Integer Decimal
9	Integer Binary
10	Magnitude
11	Address
12	Logical
13	Null
14	Null
15	Null

Table 4.1 Data Type Codes

Source acquisition and destination storage always occur and are the first and last steps, respectively. The 4 remaining intermediate steps may or may not occur, and are variously ordered as shown in Table 4.2 Intermediate Conversion Steps.

The intermediate steps are composed of the following functions:

Mode Conversions

This step deals with the real-complex conversions.

COMPLEX to REAL

Interrupts: none.

Method: Ignore imaginary part.

REAL to COMPLEX

Interrupts: none.

Method: Supply 0 as imaginary part.

Sign Conversions

This step deals with the magnitude-integer conversions.

MAGNITUDE to BINARY INTEGER

Interrupts: Binary Integer Overflow

Method: The source is interpreted as 2's complement binary integer and converted to sign/magnitude binary integer. If the source is a high-order 1 followed by 127 zeros, the Binary Integer Overflow interrupt condition occurs. If the interrupt is not being accepted, the function continues with the result the same as the source.

BINARY INTEGER to MAGNITUDE

Interrupts: none.

Method: The signed source is converted to a 2's complement binary integer which is used as the magnitude result.

Base Conversions

This step deals with the binary-decimal radix conversions. These functions are of the form: target base/constant scale.

BINARY INTEGER to DECIMAL INTEGER

Interrupts: Decimal Integer Overflow

Method: Integer conversion performed from low-order end. If the result overflows 31 decimal digits, the Decimal Integer Overflow interrupt condition occurs.

DECIMAL INTEGER to BINARY INTEGER

Interrupts: none.

Method: Integer conversion performed from low-order end.

BINARY FLOAT to DECIMAL FLOAT

DECIMAL FLOAT to BINARY FLOAT

Interrupts: none.

Method: Internal floating-point conversion is done.

Scale Conversions

This step deals with the floating-integer conversions. These functions are of the form: target scale/constant base. The fixed-float distinction is relevant only to the destination access sequence.

FLOAT DECIMAL to INTEGER DECIMAL

FLOAT BINARY to INTEGER BINARY

Interrupts: Decimal Integer Overflow
Binary Integer Overflow

Method: The source is scaled appropriately. Digits corresponding to the fraction part, if any, are ignored and no interrupt occurs. If there are digits corresponding to the integer part beyond 31 decimal or 127 binary digits, the non-critical Decimal or Binary Integer Overflow interrupt conditions occur.

INTEGER DECIMAL to FLOAT DECIMAL

INTEGER BINARY to FLOAT BINARY

Interrupts: none.

Method: The source is scaled so that there are no high-order zeros. Low-order digits beyond 116 bits are lost.

Source / Destination	Bit	Modulo	Integer Decimal	Integer Binary	Real Decimal	Real Binary	Complex Decimal	Complex Binary	Conversion Step
Bit	–	–	–	–	–	–	–	–	
Address	–	–	bin/int mag	mag	bin/flo int/bin mag	int/bin mag	real bin/flo int/bin mag	real int/bin mag	space radix scale sign
Integer Decimal	–	int dec/int	–	dec/int	int/dec	int/bin dec/int	real int/dec	real int/bin dec/int	space sign scale radix
Integer Binary	–	int	bin/int	–	bin/flo int/bin	int/bin	real bin/flo int/bin	real int/bin	space sign radix scale
Real Decimal	–	int flo/bin dec/flo	flo/dec	flo/bin dec/flo	–	dec/flo	real	real dec/flo	space sign scale radix
Real Binary	–	int flo/bin	bin/int flo/bin	flo/bin	bin/flo	–	real bin/flo	real	space sign radix scale
Complex Decimal	–	int flo/bin dec/flo comp	flo/dec comp	flo/dec comp	comp	dec/flo comp	–	dec/flo	sign scale radix space
Complex Binary	–	int flo/bin comp	bin/int flo/dec comp	flo/bin comp	bin/flo comp	comp	bin/flo	–	sign radix scale space

Performed in the order written reading from top to bottom.

TABLE 4.2 INTERMEDIATE CONVERSION STEPS

4.8 LIST OPERATIONS

List operations provide for the manipulation of stack, queue, and ring list structures. The three types of list structures are classified by the manner in which elements are added to or removed from the list.

Stack structures permit addition or deletion only from the top of the structure.

Queue structures permit addition and deletion from the top of the structure, and addition to the bottom of the structure.

Ring structures permit addition and deletion from arbitrary points within the structure.

Each list structure is embedded in a list domain which consists of one or more list structures, one of which, the _free element stack_, contains all elements in the list domain not assigned to other list structures.

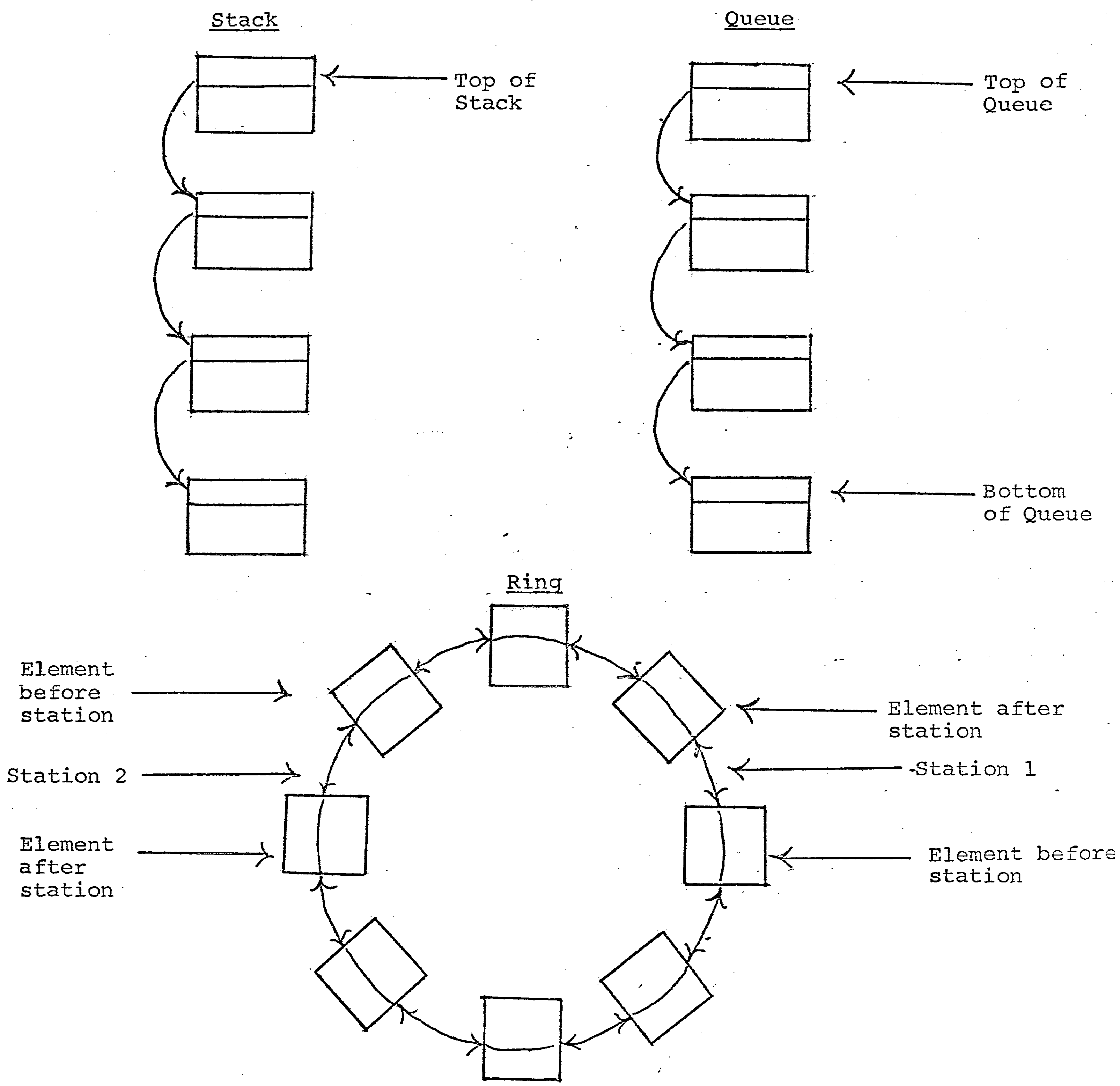


Figure 4. List Structures

4.8.1 List Domain

A list domain consists of two areas of contiguous virtual memory. One is a set of equal length data fields and the other a set of corresponding pointers, also of equal length. An element of a list domain is identified by its number and consists of a data field together with pointer occupying the corresponding position within the pointer area.

A list domain contains one or more list structures, which need not be of the same type.

Linkage --

In a stack structure, the pointer field of an element contains the element number of the element that was added to the stack prior to the addition of the given element. The pointer field of the first element that was added to the stack contains 0.

In a queue structure, the pointer field of an element contains the element number of the element that was added to the bottom of the queue after the given element. The pointer field of the last or bottom element of a queue contains 0. Note that queue pointers are identical to stack pointers.

In a ring structure, the pointer field of an element contains a value that is the 'exclusive or' of the element numbers of the elements before and after the given ring element.

Free Element Stack --

The elements within a list domain that are not linked to another list structure are on the free element stack. When an element is added to a list, it is unlinked from the free element stack and linked to the list. Similarly, elements removed from the list are unlinked from the list and linked to the free element stack. All lists within the same list domain share the same free element stack. Because the pointer field of an element is used for linkage, the elements of any list, including the free element stack, need not occupy contiguous memory locations.

The top of the free element stack is denoted by the <u>next free element number</u> pointer. The pointer field of each element in the free element stack contains the element number of another unused element, and the pointer field of the last available element contains a 0. When the free element stack is empty, the next free element number has the value 0.

4.8.2 List Domain and List Control Block

Each list instruction specifies two operands. The first operand addresses a list control block which contains pointers to elements of its associated list structure in addition to the address of the list domain control block to which it belongs. The second operand is a suboperation code which specifies one or more pointers of the list control block which are required to PUSH an element onto, PULL an element from, or STEP a pointer through the list structure.

List Control Block --

The list control block consists of 4 to 6 four byte fields depending upon the type of list structure; these are a current data index, a list domain control block address, and 2 to 4 pointers for addition or removal of elements of the structure. The list control block differs for each of the three kinds of structures in that 2 pointers are required for a stack, 3 for a queue, and 4 for a ring structure.

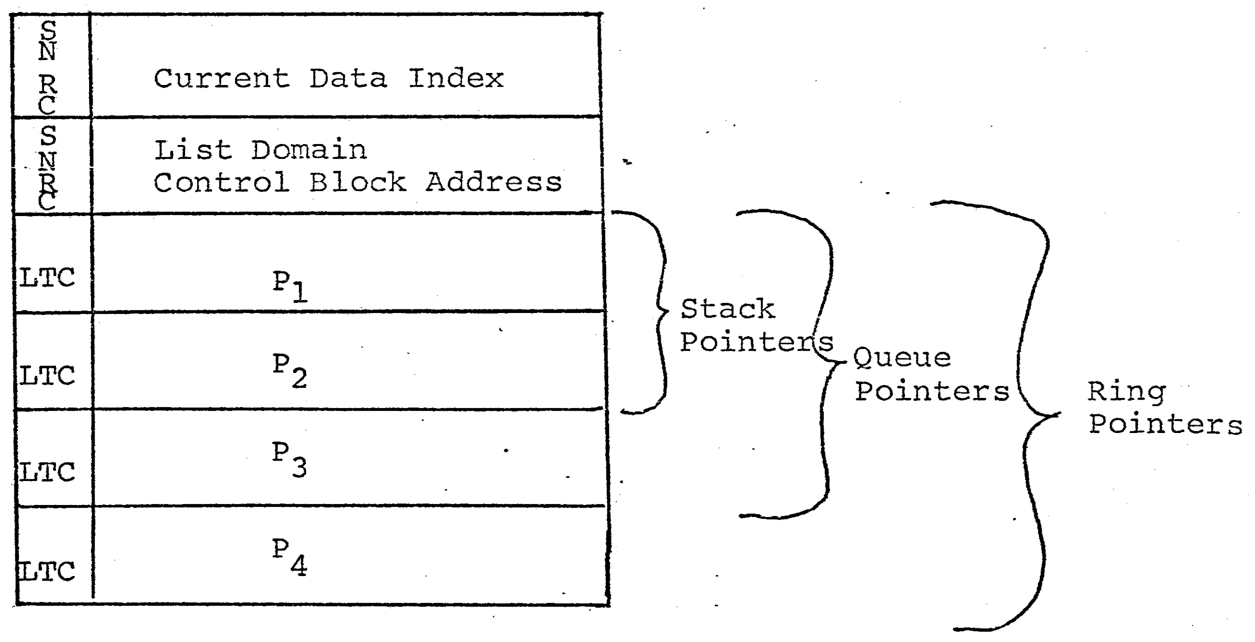


Figure 4. List Control Block Formats

The current data index is the effective address of the data field of the current element being referenced by the list control block. The current element is the one which was adjusted by the most recent list operation.

The list domain control block address specifies the effective address of the list domain control block.

A list type code occupies the 2 trailing bits of each pointer and identify the list type. It ranges in value from 1 through 3 to indicate a stack, queue, or ring, respectively.

The list control block pointers contain element numbers of elements within the list structure, or are zero if the list is empty. The number of pointers required in the list control block and their functions depend on the type of list structure being used:

> A stack control block consists of a current data index, a list domain control block address, and a push/pull pointer (p1) and step pointer (p2) with list type codes equal to 1. The push/pull pointer contains the element number of the element which is at the top of the stack. The step pointer is utilized by the STEP instructicn for data searches.
>
> A queue control block consists of a current data index, a list domain control block address, and a pull pointer (p1), step pointer (p2) and push pointer (p3) with list type codes equal to 2. The pull pointer contains the element number of the element which is at the top of the queue, and the push pointer contains the element number of the element which is at the bottom of the queue. The step pointer is utilized by the STEP instruction for data searches.
>
> A ring control block consists of a current data index, a list domain control block address, and 2 station-1 pointers (p1 and p2) and 2 station-2 pointers (p3 and p4) with list type codes equal to 3.
>
> The first pointer of each station (p1 or p3) specifies element number of the element before the station. The second pointer (p2 or p4) specifies the element number of the element after the station. Any operation may be performed using either of the stations. When both stations are affected by the addition or deletion of an element, both stations are automatically adjusted irrespective of which station was specified by the PUSH or PULL instruction.

List Domain Control Block --

The list domain control block occupies 16 contiguous bytes and contains an element number length code, a data field length, a pointer field base address, a data field base address, and the next free element number.

4 bytes

Element Size Code	Data length
SNRC	Pointer Field Base Address
SNRC	Data Field Base Address
'0'	Next Free Element Number

16 bytes

The first 4 bytes of the list domain control block contain the length of the data fields and a size code for the pointer fields. The first 30 bits contain a binary magnitude number that specifies the data length and the remaining 2 bits are the pointer size code. This code has values 0 through 3 which respectively specify pointer field lengths of 1 to 4 bytes. These pointer lengths allow the maximum element number to be 2^8-1, $2^{16}-1$, $2^{23}-1$, and $2^{30}-1$, respectively. Note that in 4-byte pointer fields, the 2 trailing bits are ignored.

The second 4 bytes of the list domain control block contain the base address of the set of pointers. The next 4 bytes are the base address of the set of data fields. Both of these base addresses have the format of an effective address.

The trailing 4 bytes of the list domain control block contain the next free element number which is the number of the next available element in the free element stack. The 2 trailing bits are always ignored.

4.8.3 Operation Codes

There are 3 list operations: PUSH, PULL, and STEP.

The PUSH operation adds elements and the PULL operation removes elements from a list. The STEP operation adjusts a temporary pointer associated with either a stack or a queue and, thus, allows the list to be searched without disturbing the top or bottom pointer. When the STEP operation is performed on a ring, a station is moved forward (toward the direction of the element after the station) or backward (toward the direction of the element before the station).

Suboperation Codes --

The second operand of a list instruction specifies a suboperation code. The suboperation code indicates whether a stack, queue, or ring is to be manipulated, and when necessary also specifies the ring station to be used in the operation. Following is a list of the functions specified by the suboperation operand according to the list operation being performed:

List Operation	Suboperation Function
PUSH	Push on top of stack
	Push on bottom of queue
	Push on top of queue
	Push before station-1
	Push after station-1
	Push before station-2
	Push after station-2
PULL	Pull from top of stack
	Pull from top of queue
	Pull before station-1
	Pull after station-1
	Pull before station-2
	Pull after station-2
STEP	Step a stack
	Step a queue
	Step station-1 forward
	Step station-1 backward
	Step station-2 forward
	Step station-2 backward

PUSH

Syntax: PUSH d1 CODE s2

Program Interrupts: Free Element Stack Empty
Illegal List Control Block Format
Unassigned Suboperation

Semantics: A list control block is specified by the first address. The next free element in the list domain specified by the list control block is added to the list structure. The next free element number is updated to reference the next free element in the free element stack.

The element data index is set to the effective address of the data field of the element just added to the list. The list control block pointers are affected in the following manner.

- Stacks

The new element is added to the top of the stack by storing the initial contents of the push/pull pointer into the pointer field of the new element and by setting the push/pull pointer to the element number of the new element.

If this is the first element to be added to the stack, the pointer field of the new element is set to 0 and the step pointer is set to the element number of the new element; otherwise, the step pointer is unaffected.

• Queues

Push on bottom of queue --

The new element is added to the bottom of the queue by setting the push (bottom) pointer and the pointer field of the element previously on the bottom of the queue to the element number of the new element. The pointer field of the new element is set to 0. If this is the first element to be added to the queue, the pull (top) pointer, the step, and the push (bottom) pointer are set to the element number of the new element; otherwise the pull and step pointers are unaffected.

Push on top of queue --

The new element is added to the top of the queue by storing the initial contents of the pull (top) pointer into the pointer field of the new element and by setting the pull pointer to the element number of the new element.

If this is the first element to be added to the queue, the pointer field of the new element is set to 0, and the step and push pointers are set to the element number of the new element; otherwise the step and push pointers are unaffected.

• Rings

The new element is added to the ring before or after the station designated by the suboperation operand. This is accomplished by relinking the pointers of the elements immediately before and after the designated station to reference the new element. The pointer field of the new element is set to reference the ring elements which are immediately before and after its position in the ring.

The specified station pointer is updated to contain the element number of the new element.

If this is the first element to be added to the ring, both station-1 pointers and both station-2 pointers are set to the element number of the new element. If station-1 was equal to station-2 prior to the operation, the pointers are modified so that when the instruction completes execution, both station pairs are still equal.

PULL

Syntax:	PULL d1 CODE s2
Program Interrupts:	Empty List Illegal List Control Block Format Illegal List Domain Control Block Format Unassigned Suboperation

Semantics:

The element specified by the designated list control block pointer is removed from the list structure and returned to the free element stack. The next free element is updated to reference the removed element. The list control block is specified by the first operand and the second operand provides suboperation code which designates the appropriate list control block pointer.

The element data index is set to the effective address of the data field of the element just removed from the list.

The list control block pointers are affected in the following manner:

- Stacks

The top element is removed from the stack by setting the push/pull pointer to the element number of the element which was previously the second from the top. If the step pointer was equal to the push/pull pointer before the operation, it is set equal to the new value of the push/pull pointer; otherwise it is unaffected.

When the last element is removed from the stack, the push/pull and step pointers are set to 0.

● Queues

The top element is removed from the queue by setting the pull pointer to the element number of the element which was previously the second from the top. If the step pointer was equal to the pull pointer before the operation, it is set equal to the new value of the pull pointer; otherwise it is unaffected.

If the last element is being removed from the queue, the pull, step, and push pointers are set to 0; otherwise the push pointer is unaffected.

● Rings

An element is removed from the ring before or after the station designated by the suboperation operand. This is accomplished by relinking the pointers of the elements immediately before and after the removed element. The specified station pointer is set to the element number of the element which now occupies the position in the ring of the removed element.

When the last element is removed from the ring, all 4 pointers are set to 0.

If the specified station pointer was equal to one of the pointers of the other station prior to the operation, both station pairs are equal after the instruction is executed.

STEP

Syntax:	STEP <u>d1</u> CODE <u>s2</u>
Program Interrupts:	Empty List Illegal List ontrol Block Format End of List

Semantics:

The list control block pointer specified by the suboperation operand is stepped to the next element position. The list control block is specified by the first operand, and the suboperation code is specified by the second operand.

The effective address of the data field of the element now referenced by the designated list control block pointer is stored into the element data index.

The list control block pointers are affected in the following manner:

- Stacks

The step pointer is moved one element position toward the bottom of the stack. This is accomplished by storing the element number contained in the pointer field of the element being referenced by the step pointer into thestep pointer. The push/pull pointer is not affected. If the step pointer is pointing to the bottom element in the stack before the instruction is executed, an attempt to STEP causes the end of list interrupt.

- Queues

The step pointer is moved one element position toward the bottom of the queue. This is accomplished by storing the element number contained in the pointer field of the element being referenced by the step pointer into the step pointer. The push and pull pointers are unaffected.

• Rings

The specified station is stepped in the direction indicated by the suboperation operand. The element numbers of the elements before and after the new position are placed into the first and second pointers of the station, respectively. The other station is not affected.

When a station is stepped in the forward direction, it is moved toward the element specified by the after pointer. Similarly, when a station is stepped backward, it is stepped in the direction of the element specified by the before pointer.

4.9 CONTROL OPERATIONS

The control operations provide capabilities for unconditional transfer, subprogram control, instruction analysis, and miscellaneous system control functions.

Subprogram Control Block

The CALL, SYSTEM, and SUBSYSTEM instructions transfer control to a subprogram and store linkage information into a subprogram control block. This control block is also required for the execution of the RESTORE and RETURN instructions.

In addition, a subprogram control block is required for argument addressing (see 3.7 Argument Addressing). The CALL, SYSTEM, and SUBSYSTEM instructions can optionally specify the location of an argument list and, if specified, the location is automatically stored into the subprogram control block.

An argument list is a byte string containing a list of addresses and a count of the number of addresses in the list. The called subprogram can access the data specified by the addresses in the list through the use of argument addressing.

A subprogram control block consists of an argument index, a link mode, an argument list address, a return address, and a calling instruction address. Each field is defined below.

Argument Index --
The 2-byte argument index field contains a binary magnitude integer that can specify an address in the argument list. This field, which is used during argument addressing, is not modified during the execution of control operations.

Link Mode --
The 1-byte link mode field contains a mode value in the leading 2 bits and a use flag in the trailing bit. When a CALL, SYSTEM, or SUBSYSTEM instruction is executed, the current value of the central processor mode is stored into the mode field. The use bit is set during the execution of a CALL, SYSTEM, or SUBSYSTEM instruction and is reset during the execution of a RETURN or RESTORE instruction.

Argument List Address --
The 5-byte argument list address field contains the address of an argument list. When a CALL, SYSTEM, or SUBSYSTEM instruction specifies the location of an argument list,

a long direct address with a length code of 2 is stored into this field. The length code is provided so that a called subprogram can access the argument count, in the leading 2 bytes of an argument list, without having to modify the subprogram control block. When the calling instruction does not specify an argument list, a null address is stored into this field.

Calling Instruction Effective Address --
The 4-byte calling instruction address field contains the effective address of the calling instruction. This address is automatically supplied when a CALL, SYSTEM, or SUBSYSTEM instruction is executed. This field makes it possible to identify the calling program.

Return Effective Address --
The 4-byte return address field contains the effective address of a return location. When a CALL, SYSTEM, or SUBSYSTEM instruction specifies a return address, the effective address is stored into this field. When the calling instruction does not specify a return address, the effective address of the next instruction after the calling instruction is stored into this field. The return address field is used during the execution of a RETURN or RESTORE instruction.

Service Call Operations

The *service call instruction*, SYSTEM and SUBSYSTEM, permit a transfer of control to subprograms that can execute in a mode different than that of the calling program.

The subprograms associated with the SYSTEM instruction execute in either the service mode (2) or the supervisor mode (3) and reside in the system segment (3 or 4). Those associated with the SUBSYSTEM instruction execute in subsystem mode (1) and reside in the subsystem segment (1).

Associated with each kind of service call instruction is a *service entry table*. Each entry in the table specifies the minimum execution mode and the location of a subprogram. A service call transfers control to one of the subprograms by specifying an entry in the table.

When a service call instruction is executed, subprogram linkage is automatically stored into a subprogram control block.

Service Entry Table --

The SUBSYSTEM and SYSTEM instructions each require a service entry table. The format of an entry in the table is:

Transition Mode	Address Value	
2	30	bit length

The address value specifies the location of a subprogram. When a SYSTEM call is executed, the subprogram is assumed to be in the system segment (3 or 4); and when a SUBSYSTEM call is executed, the subprogram is assumed to be in the subsystem segment (1).

The transition mode specifies the execution mode of a subprogram called by a service call instruction.

The location and length of a service entry table is specified by registers in the processor state vector. The effective address of the service entry table associated with the SYSTEM call is specified by the system service table base register. Similarly, the effective address of the service entry table associated with the SUBSYSTEM call is specified by the subsystem service table base register.

A service table is variable in length and entries in the table are numbered 0 through n - 1, where n is the total number of entries in the table. The maximum entry number (n-1) is specified by the system service limit register for a system service entry table or by the subsystem service limit register for the SUBSYSTEM service entry table.

A service limit register is 32 bits in length: the leading 30 bits specify the maximum entry number, 1 bit is unused, and the trailing bit specifies whether there are any entries in the associated service entry table. If the trailing bit is set, the associated service entry table is in use. If the trailing bit is reset, the associated service entry table is not in use.

Service Table Index --
The first address of either the SYSTEM or SUBSYSTEM instruction specifies an operand containing the service table index. This index specifies values of 0 through _n_-1, where _n_ is the maximum number of entries in the associated service entry table.

Mode Transition --

When a service call is executed, the central processor mode is changed to the transition mode specified by the selected entry in the service entry table; however, if the central processor mode is greater than the transition mode, it is not changed by the instruction. The called subprogram returns control to the calling program and restores the mode of the calling program by the execution of a RESTORE instruction. The RESTORE is a privileged instruction.

Go To

Syntax: GO TO <u>al</u>

Semantics: Control is transferred to location specified by the address.

Call

Syntax: CALL a2 LINK d1 [ARGUMENTS a3] [RETURN a4]

Semantics: The linkage information is stored into the subprogram control block specified by the destination operand, and control is then transferred to the subprogram entry address specified by the second operand.

If the argument list address is specified, a long direct address containing a length code of 2 is generated and stored into the argument list address field. If the argument list address is not specified, a null address is stored into this field.

If the return address is specified, the effective address of the return location is stored into the return address field. If the return address is not specified, the effective address of the location following the CALL instruction is stored into this field.

The current central processor mode is stored into the leading 2 bits of the link mode field and the use bit is set.

Subsystem

Syntax:	SUBSYSTEM s1 [ARGUMENTS a2] [RETURN a3]
Program Interrupts:	Illegal Subsystem Service Entry Table Index Illegal Transition Mode
Semantics:	The service table index is specified by the first address, the argument list address is specified by the second address, and the optional return address is specified by the third address.

Execution of this instruction causes control to be transferred to a subprogram in the subsystem segment (1) and changes the central processor mode to subsystem mode (1).

The current central processor mode is stored into the leading 2 bits of the link mode field, and the use bit is set.

The execution of this instruction requires a service entry table. The service table index is a binary magnitude number in the range 0 through n-1 where n is the maximum number of entries in the table. The entry specified by the service table index is accessed from the service entry table and the following is performed:

- The central processor mode is set equal to the transition mode, unless it is already higher. The transition mode must be equal to subsystem mode (1).

- Control is transferred to the location in the subsystem segment (1) specified by the leading 30 bits of the specified entry.

Associated with the execution of this instruction is a subprogram control block which occupies locations 1024 through 1039 (the first 16 locations of page 1) of the context segment (2). The argument list address, and the link mode field of the subprogram control block are affected by the execution of this instruction.

If the argument list address is specified, a long direct address containing a length code of 2 is generated and stored into the argument list address field. If the argument list address is not specified, a null address is stored into this field.

System

Syntax: SYSTEM s1 [ARGUMENTS a2] [RETURN a3]

Program Interrupts: Illegal System Service Table Index
Illegal Transition Mode

Semantics: The service table index is specified by the first address, the optional argument list address is specified by the second address, and the optional return address is specified by the third address.

Execution of this instruction causes control to be transferred to a subprogram in the system segment (3 or 4). The central processor mode is changed to either the service mode (2) or the supervisor mode (3).

The current central processor mode is stored into the leading 2 bits of the link mode field and the use bit is set.

The execution of this instruction requires a service entry table. The service table index is a binary magnitude number in the range 0 through n-1 where n is the maximum number of entries in the table. The entry specified by the service table index entry is accessed from the service entry table and the following is performed:

- The central processor mode is set equal to the transition mode, unless it is already higher. The transition mode must be equal to either the service mode (2) or supervisor mode (3).

- Control is transferred to the location in the system segment (3 or 4) specified by the leading 30 bits of the specified entry.

Associated with the execution of this instruction is a subprogram control block that occupies locations 0 through 15 (the first 16 locations of the page 0) of the context segment (2). The argument list address, the return address, the calling instruction address, and the link mode field of the subprogram control block are affected by the execution of this instruction.

If the argument list address is specified, a long direct address containing a length code of 2 is generated and stored into the argument list address field. If the argument list address is not specified, a null address is stored into this field.

If the return address is specified, the effective address of the return location is stored into the return address field. If the return address is not specified, the effective address of the location following the SYSTEM instruction is stored into this field.

Return

Syntax:	RETURN LINK dl
Program Interrupts:	None
Semantics:	A transfer of control occurs to the location specified by the return effective address field of the referenced subprogram control block. The use bit in the referenced subprogram control block is reset.

Restore

Syntax: RESTORE d1

Program Interrupts: Privileged Operation Code
Link Mode Greater than Central Processor Mode

Semantics: The address specifies the location of a subprogram control block. This instruction can only be executed in a mode greater than user mode (0). The mode of the central processor is set to the value specified by the leading 2 bits of the link mode field of the subprogram control block, and control is then transferred to the return effective address specified by the control block. The use bit in the control block is reset.

ANALYZE

Syntax: ANALYZE s3 CODE s2 RESULT d1 [ASC s4] [MODE s5]

Program Interrupts: Illegal Link Mode

Semantics:

The suboperation operand is specified by the second address and its value determines the operands that are specified by the other addresses. The suboperation operand specifies the operation that is performed according to the following table 4. Analyze Functions.

Calculate Element Data Index--

A list control block is specified by the first address and the pointer operand is specified by the third address.

The pointer operand ranges from 0 through 3 and specifies respectively the p1 through p4 pointers of a list control block. The effective address of the data field of the element being referenced by the specified pointer is calculated, and the result is stored into the element data index field of the list control block.

Validate Segment Number Register Code in Index and Copy Index--

An index is specified by the third address and an optional mode operand is specified by the fifth address.

The segment number register code contained in the index is examined to determine if the value is legal in the mode specified by the mode operand. If the mode operand is not present, the segment number register code is validated using the current access mode.

If the segment number register code value is legal, the index is copied into the destination operand.

Function Specified by Suboperation Operand Value (s2)	Length Code (LC1)	s3	LC3	s4	s5
calculate element data index	0	pointer number	0-	--	--
validate SNRC and copy index	3	index	0	Null	MODE*
validate SNRC and copy EA	3	instruction or address	0	ASC*	MODE*
validate SNRC of EA and form long direct type address	5	instruction or address	0	ASC*	MODE*
validate SNRC and copy address	7	instruction or address	0	ASC*	MODE*
validate SNRC & copy argument address	7	subprogram control block	0	--	--
calculate instruction length	0-15	instruction	0	--	--
calculate instruction address displacement	0-15	instruction	0	ASC	--
calculate address length	0-15	instruction or address	0	ASC*	--

Table 4.11 Analyze Functions

*Optional
EA=Effective Address
SNRC=Segment Number Register Code
ASC=Address Selection Code

Validate Segment Number Register Code in Effective Address and Copy Effective Address --

An optional address selection code is specified by the fourth address, and an optional mode operand is specified by the fifth address. The address selection code is a binary magnitude number, ranging in value from 0 through 15, that specifies 1 of the 16 addresses in an address string.

If the address selection code is specified, the third address specifies the location of the operation code in an instruction, and the address selection code specifies an address in the instruction address string. If the address selection code is not specified, the third address specifies the location of the address qualifier in an address.

The specified address is used to calculate an effective address. The segment number register code of the effective address is examined to determine if the value is legal in the mode specified by the mode operand. If the mode is not present, the segment number register code is validated using the current access mode.

If the segment number register code is legal the effective address is copied into the destination operand.

Validate Segment Number Register Code in Effective Address, Form a Long Direct Address --

This operation is the same as that described above for the suboperation value equal to 2, except for the following: A long direct address if formed using the validated effective address and its associated length code. The formed address is stored into the destination operand.

Validate Segment Number Register Code in Address and Copy Address --

An optional address selection code is specified by the fourth address, and an optional mode operand is specified by the fifth address. The address selection code is a binary magnitude number, ranging in value from 0 through 15, that specifies 1 of 16 addresses in an Address string.

If the address selection code is specified the third address specifies the location of the operation code in an instruction and the address selection code specifies an address in the instruction address string. If the address selection code is not specified, the third address specifies the location of the address qualifier in an address.

The specified address is accessed. The segment number register code contained in the base address and the index address, if present, are examined to determine if the value is legal in the mode specified by the mode operand. If the mode operand is not present, the segment number register codes are validated using the current access mode.

If the segment number register code is legal, the complete address is copied into the destination operand. If the address is less than 8 bytes in length, the remaining bytes of the destination operand are filled with 0's.

Validate Segment Number Register Code in Argument Address and Copy Argument Address --

The third address specifies an address within an argument list and must use argument addressing.

The segment number register code contained in the indicated argument address is examined to determine if the segment number register code is legal in the mode specified by the link mode in the subprogram control block. If the ANALYZE instruction is executed in mode 0, the link mode is assumed also to be 0.

If the segment number register code is legal, the complete argument address is copied into the destination operand. If the address is less than 8 bytes long, the remaining bytes of the destination operand are filled with 0's.

Calculate Instruction Length --

The third address specifies the location of the operation code of an instruction.

The length in bytes of the instruction byte string is calculated, and the binary magnitude number replaces the destination operand.

Calculate Instruction Address Displacement --

The location of an instruction is specified by the third address, and an address selection code is specified by the fourth address. The address selection code is a binary magnitude number, ranging in value from 0 through 15, that specifies 1 of 16 addresses in an address string.

The number of bytes of displacement from the leading byte of the instruction to the leading byte of the address specified by the address selection code is calculated. The result, which is a binary magnitude number, replaces the destination operand.

Calculate Address Length --

An optional Address Selection Code is specified by the fourth address. The Address Selection Code is a binary magnitude number, ranging in value from 0 through 15, that specifies 1 of the 16 addresses in an Address String.

If the Address selection code is specified, the third address specifies the location of the operation code in an instruction and the address selection code specifies an address in the instruction. If the address selection code is not specified, the third address specifies the location of the address qualifier in an address.

The length in bytes of the indicated address is calculated. The result, which is a binary magnitude number replaces the destination operand.

SET

Syntax

SET d2 RESULT d1

Program Interrupts

Privileged Operation Code

Semantics

An interprocessor control flag, which is a 1-byte operand specified by the second address, is set to the locked state (all 1's). The previous contents of the flag are stored into the destination operand specified by the first address.

This instruction can only be executed in modes 2 and 3.

Execution of the SET instruction by a central processor prevents accesses to the specified interprocessor control flag by any other processor in the system between the reading of the flag and the setting of the flag.

By convention an interprocessor control flag is in the locked state if any of the bits are 1's. It is in the unlocked state if all bits are 0's.

Null

Syntax:	NULL
Semantics:	This instruction requires a null instruction format as shown in Figure 3.1. Execution of this instruction causes control to be transferred to the next instruction in sequence.

Pass

Syntax: PASS s1, [s2], [s3], [s4], [s5] . . ., [s16]

Semantics: The first address specifies the first source operand. The second through sixteenth addresses specify optional source operands.

Execution of this instruction causes the instruction location register to be incremented by the number of bytes in the instruction string. Control is transferred to the next instruction in sequence.

CONTROL

Syntax

CONTROL ([d1] [, s2]) CODE s3 [DISP s4] [MASK s5]

Program Interrupts

Privileged Operation Code

Semantics

The third operand is a suboperation code that specifies a device internal to a central processor. The device is set to the value of the source operand, which is specified by the second address. The previous setting of the device is stored into the destination operand, which is specified by the first address. The fourth and fifth addresses specify respectively optional displacement and mask operands.

This instruction can be executed only in modes 2 and 3.

The destination and source operands are both optional. If both are omitted, there is no action taken. If only the destination is specified, the device is not altered and its current setting is stored into the destination operand. When only the source operand is present, it specifies the new setting of the device and the previous setting is not saved.

The displacement operand specifies alignment of the byte string that contains the device setting with the byte strings of the source and destination operands. The displacement operand is a signed binary integer. A positive displacement specifies alignment of the leading bit (bit position 0) of the source and destination operands with bit position n of the device byte string, where n is the absolute value of the displacement. A negative displacement specifies alignment of bit position n of the source and destination operands with the leading bit (bit position 0) of the device byte string.

The mask operand specifies which bits in the device byte string participate in the operation. The mask operand is always aligned with the device byte string. When the mask operand is present, bits in the device byte string participate in the operation only if the corresponding bit position in the mask operand is 1.

The devices that can be specified by the suboperation code of a CONTROL instruction are the various processor state operands, the timers, and various diagnostic devices.

SELECT

Syntax

SELECT ([d1] [, s2]) CODE s3

Program Interrupts

Privileged Operation Code

Semantics

The third operand is a suboperation code that specifies a device external to a central processor and an operation to be performed by the device. The source operand specified by the second address is output to the device and the destination operand receives input from the device. The output and input are device-dependent status information. Both the source and the destination operands are optional. If both are omitted, no action is taken.

This instruction can be executed only in modes 2 and 3.

System Extended Operation

Syntax: None

Program Interrupts: Illegal system extended operation code.

Semantics: This operation code requires the extended instruction format as depicted in Figure 2.1 Instruction Formats. The system extended operation code specifies that the following byte is an extended operation code. The system extended operation code can be utilized to specify 1 of 256 operations in addition to those included in the standard GEMINI operation code set. The system extended operation codes will be assigned by Computer Operations, Inc.

Private Extended Operation

Syntax:	None
Program Interrupts:	Illegal Private Extended Operation Code
Semantics:	This operation code requires the extended instruction format as depicted in Figure 2.1 Instruction Formats. The private extended operation code specifies that the following byte is an extended operation code. The private extended operation code can be utilized to specify 1 of 256 operations in addition to those included in the standard GEMINI operation code set.

5.0 ITERATIVE EXECUTION

Iterative execution allows an instruction to be accessed once and executec repeatedly.

In addition, execution operand indexes are automatically incremented permitting the use of the iteration option for array operations and searches.

All instructions except those in the control, list, and select groups can be iteratively executed.

The iteration option is invoked by specifying a set of iteration operands following the execution operands. The execution operands are those that can appear in the instruction when iteration is not specified. The iteration operands include a count, which specifies the maximum number of times the instruction is to be performed, and a set of optional increments corresponding to the execution operands.

After each iteration the instruction may be interrupted. If an interrupt occurs, sufficient information is available in the processor state vector to allow the iteration to be resumed after the interrupt is serviced. If a comparison is being executed and the comparson is satisfied the iteration is terminated.

5.1 ITERATION SPECIFICATION

The iteration option includes an iteration count operand and index increment operands. The iteration count is the maximum number of times the instruction is to be performed. The iteration count operand is assumed to be a binary magnitude number and must be in the range of 0 to $2^{32}-1$. An index increment operand specifies the increment value for an index used in the calculation of the effective address of an execution operand. It is assumed to contain an address value.

The addresses used for the iteration option occupy the trailing positions in the address string of the instruction. The first position occupied by an iteration operand is the first address position available after the execution operands. The maximum number of execution operands in an instruction is 5 and with a full complement of iteration operands the maximum number of operands in an instruction is 11.

The syntax of the iteration option is:

[,PERFORM $\underline{S}_{\underline{n}}$ INCREMENTS ([$\underline{S}_{\underline{n}+1}$],[$\underline{S}_{\underline{n}+2}$],[$\underline{S}_{\underline{n}+3}$],[$\underline{S}_{\underline{n}+4}$],[$\underline{S}_{\underline{n}+5}$])]

where $\underline{n}$ = the first address position available after the required operands and other optional operands have been specified

S_n = the address specifying the iteration count operand. This operand is a binary magnitude number in the range of 0 to $2^{32}-1$.

S_{n+1} = addresses S_n+1 through S_n+5 specify the index increment operands for the first through the fifth address in the address string. These operands have the format of an address operand. Each of these addresses is optional.

Following is an example of the syntax of a computational addition operation specifying the iteration option:

```
d1 = s2 + s3 [,ADJUST(s4,[s5])][,PERFORM s6 INCREMENTS
             ([s7],[s8],[s9],[s10],[s11])]
```

where d1 = the address of the destination operand; its associated index increment is specified by s7.

s2 to s5 = the addresses of the second through fifth source operands. The associated index increments are specified by s8 through s11, respectively.

s6 = the address of the iteration count operand.

An optional address can be omitted entirely from an address string provided that the address is not followed by a non-null address; otherwise, the optional address is specified by the null addressing type. Thus, if the iteration option is not desired, the s6 through s11 addresses are completely omitted from the address string. If both the adjustment option and the iteration option are not desired, addresses s4 through s11 are omitted. If the iteration option is desired, but the adjustment option is not, the <u>s4</u> and <u>s5</u> addresses must be specified by null addresses.

5.2 INDEX INCREMENTATION

If any of the execution operands specify either an alignment or displacement indexing calculation, the indexes are incremented by the value specified by the index increment operands. If an operand is specified by either an indirect or argument addressing type, the incrementation is performed only on an index in the final address. A discussion of the final address is in 3.6 <u>Indirect Addressing</u> and 3.7 <u>Argument Addressing</u>.

The incrementation is performed after each iteration. Thus, the effective address of an operand is computed using the index value generated during the previous iteration.

If an increment value is not provided for an address specifying indexing, an increment of 0 is understood and the index is not modified.

5.3 INITIALIZATION FOR ITERATIVE EXECUTION

When the iteration option is specified, the instruction is accessed only once, and the information required for the iterative execution is stored in the processor state vector. When an iterative instruction is encountered, the following operations are performed:

- The perform state bit is set. This control specified that an iterative instruction is being performed; the bit remains set until the iteration terminates, when it is automatically reset.

- The operation code is stored into the perform operation code field. The instruction qualifier is stored into the perform instruction qualifier field.

- For each instruction address, exclusive of the addresses for the iteration operands, information describing the final address used to specify the operand is stored.

For each final address of an execution operand, the address qualifier is stored into one of the five perform address qualifier fields. The other address information retained depends on the type of addressing used to specify the operand.

If null addressing is specified, the address qualifier is the only information retained.

If immediate addressing is specified, the immediate value is also stored into the perform immediate value field. If the address is not indexed, the final effective address is calculated and stored into the perform effective address field.

If the final address is indexed, the information necessary to perform an effective address calculation after each iteration is contained in the processor state vector:

- The index address is stored into the perform index address field.
- The initial value of the index is stored into the perform index value field. This field is updated during each iteration by the value of the index increment.
- The index increment operand is stored into the perform index increment field.
- The effective address specifying the operand during the first iteration is calculated and stored into the perform effective address field. This field is updated during each iteration and, thus, always specifies the current effective address of the operand.
- The effective address increment field is initialized. This field specifies the increment that is to be added to the effective address field after each iteration. If displacement indexing is used, the effective address increment is equal to the index increment operand. If alignment indexing is used, the effective address increment is equal to the product of the index increment operand multiplied by the length code.

Each index is maintained and incremented in the processor state vector, and when the iteration terminates, the incremented index is stored into memory.

6.4 MECHANISM OF ITERATIVE EXECUTION

When an iterative instruction is executed, the iteration controls in the processor state vector are initialized and the iteration count value is examined. If the iteration count is equal to 0, the instruction is not executed, the iteration is terminated, and control is transferred to the next instruction in sequence. When an iteration is terminated, the perform state bit is reset and the indexes in the processor state vector are stored into memory at the corresponding index address.

If the iteration count is greater than 0, the iterative execution is begun. During each iteration, any result is stored into the specified destination operand. However, indexes are maintained and incremented in the processor state vector until the iteration is terminated.

During each iteration, the following is performed:

- The instruction is executed.
- If a comparison instruction is being executed and the comparison is satisfied, the iteration is terminated. The effective address specified by the control transfer location is stored into the instruction location register.
- If any interrupt requires servicing, the iteration is interrupted. The information required to resume the iteration is contained in the processor state vector. The instruction location register specifies the effective address of the iterative instruction, the perform state bit remains set, and the indexes remain in the processor state vector.
- If neither a comparison is satisfied nor an interrupt occurs, the indexes and the effective addresses are incremented.
- The perform iteration count is decremented by 1 and, if the decremented count is zero, the iteration terminates. The effective address of the next instruction in sequence is stored into the instruction location register. If the decremented count is greater than zero, the above sequence is repeated.

If an iteration terminates because a comparison condition was satisfied, the indexes specify the operands that caused the comparison to be met. Similarly, if a program interrupt condition resulted from an execution, the indexes specify the operands involved in the current iteration. The indexes are incremented prior to the test of the iteration count for termination. For this reason, if an iteration terminates because the iteration count was decremented to zero, the indexes have been incremented beyond the operands involved in the final iteration.

6.0 EXECUTION BREAKPOINTS

Breakpoint controls allow the user to specify conditions for which program interrupts are to be generated. Breakpoint controls are located in the instruction qualifier and the processor state vector. The controls in the instruction qualifier are related only to the particular instruction in which they appear. Those in the processor state vector are related to all instructions. Further, certain processor state vector controls must be set in order for the instruction qualifier controls to be interpreted.

6.1 INSTRUCTION QUALIFIER BREAKPOINT CONTROLS

The breakpoint controls located in the instruction qualifier permit control of the execution of an individual instruction. These controls serve the following purposes:

- To prevent the instruction from being executed. The instruction is bypassed and control is transferred to the next instruction in sequence.
- To define the instruction as the end of a statement. (A statement is defined as an instruction or series of instructions).
- To cause an interrupt either before or after the execution of the instruction.

The instruction qualifier breakpoint controls are interpreted only when the corresponding master breakpoint controls in the processor state vector are set. The processor state vector master breakpoint are explained in 6.2.5 Format of Master Controls. The instruction qualifier breakpoint controls function in conjunction with a breakpoint control mode master control in the processor state vector, and are effective only when the breakpoint control mode is greater than or equal to the central processor mode.

The format of the instruction qualifier is as follows:

Instruction Qualifier Format

Address Count	P	S	A	B
4	1	1	1	1

B-Bit (Before) --

If the B-bit is set in the instruction qualifier, and the enable A-bit and B-bit in instruction qualifier control (B11-bit) is set in the processor state vector, the break before instruction interrupt condition occurs before the execution of the instruction.

A-Bit (After)--

If the A-bit is set in the instruction qualifier and the enable A-bit and B-bit in instruction qualifier control (B11-bit) is set in the processor state vector, the break after instruction interrupt condition occurs after the execution of the instruction.

S-Bit (Statement End) --

If the S-bit is set in the instruction qualifier, the instruction is regarded as the end of a statement. The end of statement control (B12-bit), the instruction count control (B13-bit), and the statement count control (B14-bit) in the processor state vector determine the interpretation of the S-bit, as described in section 6.2.1 Statement Counter.

P-Bit (Pass) --

If the P-bit is set in the instruction qualifier and the enable P-bit in instruction qualifier control (B15-bit) is set in the processor state vector, the instruction is not executed and control is transferred to the next instruction in sequence.

Address Count --

The 4-bit address count field specifies the number of addresses in the address string of the instruction and is not part of the breakpoint controls.

6.2 PROCESSOR STATE VECTOR BREAKPOINT CONTROLS

Controls located in the processor state vector permit interrupts to be generated:

- After each instruction has been executed.
- Before or after an instruction has been executed that contains an address within specified bounds.
- Before or after an instruction has been executed that contains a specified operation code or extended operation code.
- After a specified number of instructions have been executed.

- After a specified number of statements have been executed
- After the completion of the current statement execution

In addition, the processor state vector contains controls that:

- Enable the breakpoint controls in the instruction qualifier
- Specify a breakpoint control mode
- Enable a location counter that specifies the effective address of the previously executed instruction

The processor state vector controls consist of a statement counter, operation code masks, address bounds registers and controls, a prior instruction location register, and the master breakpoint control bits numbered B0 through B16. All breakpoint controls, except the address breakpoint controls are effective only when the breakpoint control mode (indicated by master control bits B2 and B3) is greater than or equal to the central processor mode.

6.2.1 Statement Counter

A statement is a series of one or more instructions, the last of which is indicated by having the S-bit set in the instruction qualifier. If the instruction count control (B14-bit) is set, however, every instruction is considered the end of a statement regardless of the setting of the S-bit in the instruction qualifier. If the end of statement control (B12-bit) is set, the end of statement interrupt condition occurs after execution of an instruction which is the end of a statement. If the statement count control (B14-bit) is set, the statement counter is decremented by 1 after execution of an instruction which is the end of a statement.

The statement counter is a 4-byte register in the processor state vector. It can be given any 32-bit binary magnitude value. When the statement count control (B14-bit) is set, the statement counter is decremented after execution of every end of statement instruction. If decrementing the statement counter results in a 0 result, the statement counter equals 0 interrupt condition occurs. By placing the appropriate initial value into the statement counter and setting the statement count control (B14-bit), it is possible to interrupt execution of a program after a given number of statements has been executed, or after a given number of instructions if the instruction count control (B13-bit) is also set.

6.2.2 Operation Code Masks

Three 256-bit operation code masks are associated, respectively, with standard operation codes, system extended operation codes, and private extended operation codes. An operation code is selected for breakpointing by setting the bit position in the mask that corresponds to the value assigned to the operation code. Thus, each mask can simultaneously select from 1 to 256 unique operations.

The B4 and B5 control bits determine whether a breakpoint occurs before or after the execution of an instruction with a standard operation code selected in the operation code mask. If both the B4 and B5 bits are reset, the standard operation code mask is ignored. Similarly, the B6 and B7 bits control the private extended operation code mask, and the B8 and B9 bits control the system extended operation code mask.

6.2.3 Address Bounds Registers and Controls

Eight address bounds registers detect memory references that are within the bounds of from 1 to 8 pairs of address boundaries. Any number of the registers can be active simultaneously. Each 64-bit register has the following format:

ADDRESS BOUNDS REGISTER	MAC	Upper Address Limit	SNRC	Lower Address Limit
	2	30	2	30

The address bounds are specified by long (30-bit) address values that specify the upper and lower address limits. The segment number register code (SNRC) is discussed in 2.2.2. Formation of a Complete Virtual Address. The memory access code (MAC) specifies the type of memory access that is to be detected and has the following values:

MAC = 0	Compare on instruction access only
MAC = 1	Compare on operand write access only
MAC = 2	Compare on any operand access
MAC = 3	Compare on any access

The 16-bit address bounds register controls select a register and determine if the address breakpoint will occur before or after the execution of the instruction containing a specified address. The bits in the controls are paired, and each pair corresponds to one of the 8 address bounds registers. The leading bit of each pair corresponds to a breakpoint before and the trailing bit corresponds to a breakpoint after the specified address is detected. If both bits are reset, the contents of the associated address bounds register are ignored.

The address bounds register controls are enabled by the B1 master control bit. If this bit is reset, the address bounds register controls, and hence the address bounds registers, are considered inactive.

6.2.4 Prior Instruction Location Register

The prior instruction location register is a 32-bit register enabled by the B0 master control bit. It contains the effective address of the instruction executed before the current instruction. The effective address of the current instruction is always maintained in the instruction location register.

6.2.5 Format of Master Controls

The master breakpoints controls are numbered B0 through B16 and have the format shown in Figure 6.1.

Extended Operation Code Breakpoints				Operation Code Breakpoints		Breakpoint Control Mode		Enable Address Bounds Register	Enable Prior Instruction Register
B9	B8	B7	B6	B5	B4	B3	B2	B1	B0

Perform Breakpoint	Enable Pass	Statement Breakpoints			Instruction Breakpoints	
B16	B15	B14	B13	B12	B11	B10

Figure 6.1 Format of Master Breakpoint Controls

Enable Prior Instruction Location Register Bit --

B0 (enable prior instruction location register). If this bit is set, the location of the previously executed instruction is maintained in the prior instruction location register.

Enable Address Bounds Register Controls Bit --

B1 (enable address bounds register controls). If this bit is set, the address bounds register controls are enabled.

Breakpoint Control Mode (2 bits) --

B2 and B3 (breakpoint control mode). All breakpoint controls, except the address bounds register controls, are effective only if the breakpoint control mode is greater than or equal to the central processor mode.

Operation Code Breakpoints (2 bits) --

B4 (breakpoint before operation code). If this bit is set and an operation code specified by the operation code mask is detected, the <u>break before on operation code</u> interrupt condition occurs before the execution of the instruction that contains the specified operation code.

B5 (breakpoint after operation code). If this bit is set and an operation code specified by the operation code mask is detected, the <u>break after on operation code</u> interrupt condition occurs after the execution of the instruction that contains the specified operation code.

Extended Operation Code Breakpoints (4 bits) --

B6 (breakpoint before system extended operation code). If this bit is set and a system extended operation code specified by the system extended operation code mask is detected, the <u>break before on system extended operation code</u> interrupt condition occurs before the execution of the instruction that contains the specified operation code.

B7 (breakpoint after system extended operation code). If this bit is set and a system extended operation code specified by the system extended operation code mask is detected, the <u>break after on system extended operation code</u> interrupt condition occurs after the execution of the instruction that contains the specified operation code.

B8 (breakpoint before private extended operation code). If this bit is set and a private extended operation code specified by the private extended operation code mask is detected, the <u>break before on private extended operation code</u> interrupt condition occurs before the execution of the instruction that contains the specified operation code.

B9 (breakpoint after private extended operation code). If this bit is set and a private extended operation code specified by the private extended operation code mask is detected, the <u>break after on private extended operation code</u> interrupt condition occurs after the execution of the instruction that contains the specified operation code.

Instruction Breakpoints (2 bits) --

B10 (breakpoint after instruction). If this bit is set, the break after instruction interrupt condition occurs after the execution of each instruction regardless of the state of the A-bit in the instruction qualifier.

B11 (enable A-bit and B-bit in instruction qualifier). If this bit is set, the A-bit and the B-bit in the instruction qualifier are interpreted.

Statement Breakpoints (3 bits) --

B12 (end of statement control). If this bit is set, the end of statement interrupt condition occurs after the execution of an instruction that has the S-bit set in the instruction qualifier.

B13 (instruction count control). If this bit is set, each instruction is regarded as an end of statement.

B14 (statement count control). If this bit is set, the statement counter is decremented after the execution of any instruction that has the S-bit set in the instruction qualifier, or if B13 is set after every instruction. If the statement counter equals 0 after the execution of the instruction, the statement counter equals zero interrupt condition occurs. If the statement counter is nonzero, the instruction in sequence is executed.

Enable Pass Bit (1 bit) --
B15 (enable P-bit in instruction qualifier). If this bit is set, any instruction that has the P-bit set in the instruction qualifier is not executed, and control is passed to the next instruction in sequence.

Perform Breakpoint Control (1 bit) --
B16 (perform breakpoint control). If this bit is set, the break after perform iteration interrupt condition occurs after each iteration of an iterative instruction.

7.0 <u>INTERRUPTS</u>

Interrupts provide the means for the system to respond to specific conditions by causing an interruption of the currently executing program and a transfer of control to an interrupt service program. The automatic sequence performed by the interrupt system in acknowledging a condition and activating a service program is termed <u>processing</u> an interrupt. Simultaneously occurring interrupts are serviced one at a time according to a fixed priority scheme. The sequence of actions taken by the service program that is activated is termed <u>servicing</u> an interrupt.

A central processor controls the processing of interrupts by means of a hierarchy of programmable control elements in the interrupt system. These include accept, hold, dispatch, and lockout controls.

Figure 7.1 shows the organization of the interrupt system in terms of the controls affecting the processing of an interrupt.

7.1 INTERRUPT PRIORITY GROUPS

An interrupt is classified as either shared or private and has an assigned priority level. <u>Private interrupts</u> are the result of conditions internal to a central processor or caused directly by the activity of a central processor.

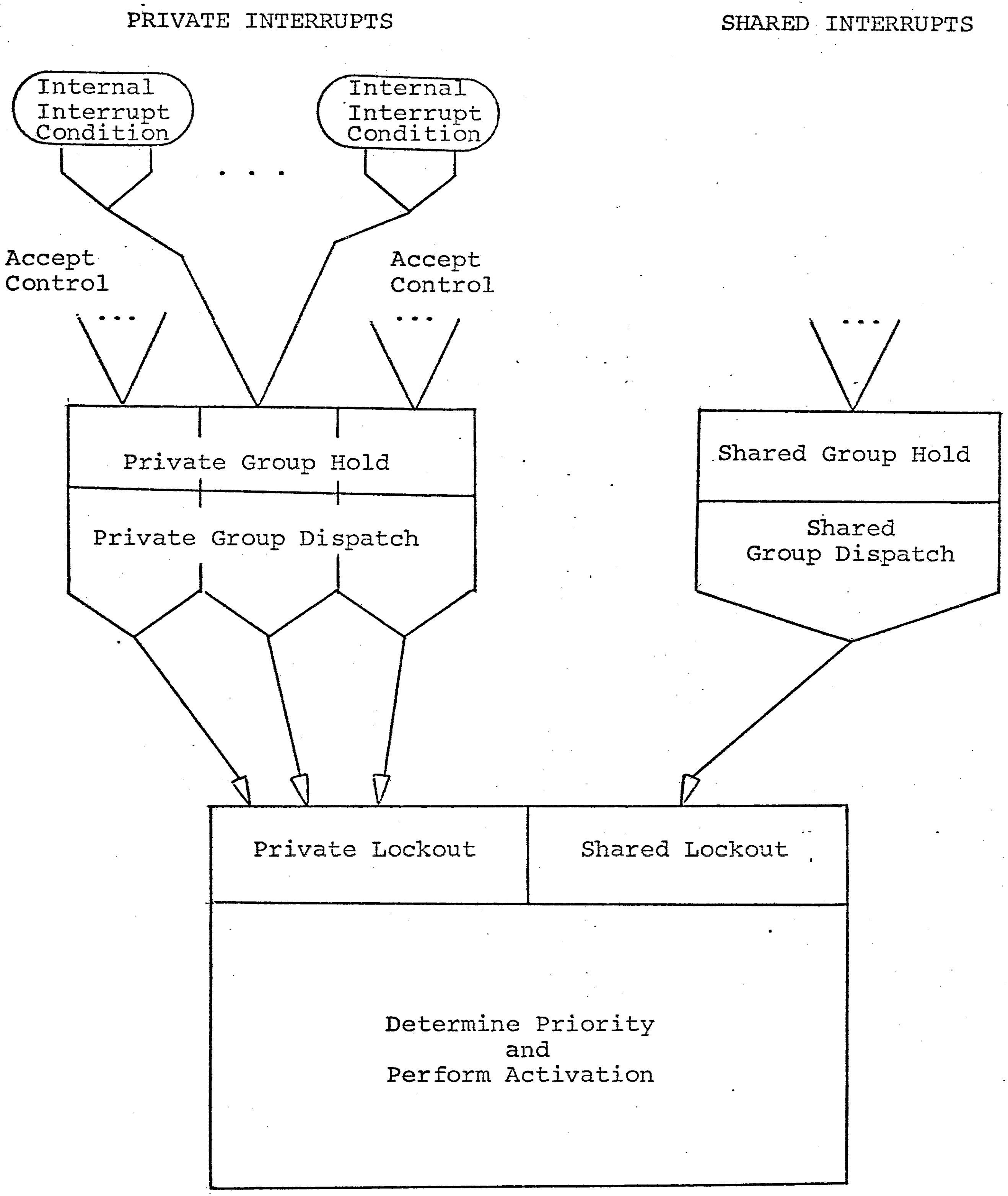


Figure 7.1 Organization of Interrupt Controls

Private interrupts are processed only by the central processor that caused them. Shared interrupts result from the occurrence of asynchronous conditions and, in a dual central processor system, may be processed by either central processor.

The priority level of a particular interrupt assigns the interrupt to one of the interrupt priority groups. Table 7.1 lists the interrupt priority groups in the order of their priority. A larger interrupt priority number indicates a higher priority.

7.2 INTERRUPT CONTROLS

7.2.1 Interrupt Accept Controls

Associated with each interrupt condition is a 1-bit accept control. When the accept control is set, the interrupt is being accepted and the occurrence of the interrupt condition causes the corresponding priority group hold bit to be set.

7.2.2 Group Hold

Associated with each interrupt priority group is a hold bit that can be set by the occurrence of any one of the interrupt conditions included in the interrupt priority group. The hold bit records the occurrence of an interrupt condition.

The hold bits for the private interrupt groups are in the processor state vector.

Table 7.1 Interrupt Priority Groups

GROUP	PRIORITY	PRIVATE/SHARED
Diagnostic	4 and higher	Private or Shared
Shared Processor	3	Shared
Program	2	Private
Statistical Timers	1	Private
Quantum Timer	0	Private

7.2.3 Group Dispatch

There is a dispatch control for each interrupt priority group which can prevent the group from being scanned for activation.

The dispatch bits for the private interrupt groups are in the processor state vector.

7.2.4 Interrupt Lockouts

Two interrupt lockout control bits are in the processor state vector. The private interrupt lockout bit, when set, prevents the private interrupts from being scanned for activation. The shared interrupt lockout bit, when set, prevents the shared interrupts from being scanned for activation by the Central Processor in question. See 7.4.2 Scanning Interrupts for Activation.

7.2.5 Old and New Processor State Operands

For each interrupt priority group there is an old processor state operand 0 effective address and a new processor state operand 0 effective address. The first effective address specifies a memory location into which the 16 bytes of the current processor state operand zero are to be written when the interrupt priority group is activated. The second effective address specifies a memory location from which 16 bytes are read to provide a new processor state operand zero to begin servicing the interrupt. These effective addresses can be altered by CONTROL instructions.

7.3 INTERRUPT FLAGS

The interrupt flags specifically identify the interrupt condition and, when applicable, the addressing circumstances at the time of the interrupt.

The interrupt condition code is a 16-bit number that uniquely identifies each interrupt condition within an interrupt priority group.

The address selection code is a 4-bit number from 1 through 11 that specifies which of the 11 addresses of an instruction caused the interrupt, if it occurred during an effective address computation.

The effective address code is a 2-bit number that, when appropriate, specifies the address being evaluated in an effective address computation when the interrupt occurred, as indicated below:

0 - The address was in the address string of the instruction.

1 - The address was a final address in indirect addressing.

2 - The address was a final address in argument addressing.

3 - The address was in a subprogram control block in argument addressing.

7.4 INTERRUPT PROCESSING

7.4.1 Occurrence of an Interrupt Condition

The processing of an interrupt is initiated by the occurrence of an interrupt condition. If it is not being accepted, the interrupt condition is ignored and no further processing is necessary. If the associated accept control bit is set, occurrence of the condition causes the setting of the corresponding interrupt group hold bit. The setting of the hold bit places the interrupt group in a pending state.

When a hold bit is set, the priority group is pending or active and subsequent occurrences of interrupt conditions of that group cannot be detected. Whenever a subsequent interrupt condition occurs, provision is made for critical interrupt conditions and all spurious program interrupt conditions to be processed by causing a diagnostic interrupt condition.

7.4.2 Scanning Interrupts for Activation

A central processor is in an interruptible state following the completion of each instruction or following each iteration of an iterative instruction. Each time the central processor is in an interruptible state, the interrupts are scanned to determine which, if any, is to be activated.

Only those interrupt priority groups for which the corresponding dispatch bit is set and the associated lockout bit is reset are included in the priority comparison. An interrupt that is pending but not being dispatched remains pending. In order to qualify for activation, an interrupt priority group must be pending, dispatchable, and not locked out.

The final determination for interrupt activation is interrupt priority. The priority comparison includes the currently active interrupts. In order to be activated, an interrupt group must have higher priority than any interrupts that are currently active and it must have the highest priority of all the nonactive groups that are pending, dispatchable, and not locked out.

7.4.3 Interrupt Activation

When an interrupt is activated, the interrupt condition code (see 7.3 Interrupt Flags) is stored into processor state operand zero. The entire processor state operand zero is written into the old processor state operand zero location for the particular priority group that is being activated. The contents of the new processor state operand zero location for that group are read into the processor state vector to provide the processor state controls and instruction location necessary to service the interrupt.

The interrupt has been activated when the current processor state operand zero has been saved and program control has been transferred to the location specified by the new processor state operand zero. The central processor is again in an interruptible state at the completion of the activation. An interrupt remains active until cleared.

7.4.4 Clearing an Interrupt

Once an interrupt is activated, it remains active until it is explicitly cleared. Because an interrupt service program can itself be interrupted by a higher priority interrupt, more than one interrupt can be active at a time. An interrupt is cleared by a CONTROL instruction which both restores the old processor state operand zero and specifies the clearing of the interrupt group.

7.5 SHARED PROCESSOR INTERRUPT GROUP

The shared processor interrupt group, priority group 3, comprises a single interrupt condition. When the shared processor interrupt group is activated, the interrupt condition code in processor state operand zero is set to indicate the unit number associated with the device that originated the interrupt. The unit number is the same one that was used by a central processor when issuing a Start I/O SELECT instruction to the device. When the interrupt has been activated, the service program can use the unit number to issue a Test I/O SELECT instruction to interrogate the device and obtain a channel status operand. The channel status operand contains information about the exact condition for which the device is requesting central processor action.

7.6 PROGRAM INTERRUPT GROUP

The program interrupt group, priority group 2,are conditions which occur as a direct result of instruction execution. When the interrupt is being accepted, the instruction that caused the condition is aborted, the destination operand is not modified, and the instruction location in the old program status operand zero contains the effective address of the instruction that caused the interrupt.

The program interrupts are divided into 2 categories; those that are critical to the continued execution of the instruction, and those that are noncritical. When a critical condition occurs but the interrupt is not being accepted, the instruction is aborted and a diagnostic interrupt occurs. When a noncritical condition occurs and the interrupt is not being accepted, the instruction execution is not aborted but continues to completion as defined for that condition.

All program interrupts are critical unless otherwise specified in the interrupt description. Descriptions of non-critical interrupt conditions explicitly specify the result of the continued instruction execution.

For programming convenience, individual program interrupt conditions that have common characteristics and handling requirements are grouped into <u>classes</u>. The descriptions of the individual conditions are grouped according to these classes. It is also convenient to refer to subclasses within some of the classes. A subclass is simply a cross-section of the class consisting of individual conditions with one or more common components to the condition name, as the <u>binary real</u> overflow subclass of the overflow class, comprising the <u>binary</u> fixed <u>real</u> overflow and <u>binary</u> float <u>real</u> overflow interrupt conditions. Some interrupt conditions have been grouped here by subclass for reference by the instruction operation descriptions in 4.0 <u>INSTRUCTION SET</u>.

7.6.1 <u>Page Assignment Class</u>

Write with Unassigned Virtual Address --

An attempt was made to write into memory using a virtual address that did not have an entry in the address transformation table. The address transformation search key, beginning search index, current search index, effective address and the data to be written are stored into the memory segment specified by segment number register 3.

Write with Secondary Storage Address --

An attempt was made to write into memory using a virtual address that had an entry in the address transformation table with the secondary storage bit set. The address transformation search key, beginning search index, current search index, effective address,and the data to be written are stored into the memory segment specified by segment number register 3.

Read with Secondary Storage Address --

An attempt was made to read from memory using a virtual address that had an entry in the address transformation table with the secondary storage bit set. The address transformation search key, beginning search index, current search index, and effective address are stored into the memory segment specified by segment number register 3.

7.6.2 Statistical Class

Reference to Segment Number Register 0 --

The segment number register code of an address specified segment number register zero. If the interrupt is not being accepted, the specified access is continued.

Reference to Segment Number Register 1 --

The segment number register code of an address specified segment number register 1. If the interrupt is not being accepted, the specified access is continued.

Reference to Segment Number Register 2 --

The segment number register code of an address specified segment number register 2. If the interrupt is not being accepted, the specified access is continued.

Reference to Segment Number Register 3 --

The segment number register code of an address specified system segment number register. If the interrupt is not being accepted, the specified access is continued.

Mode Transition Subclass

The following interrupts result only from the execution of service instructions which cause a transition to a higher mode. The interrupt condition occurs after the mode transition has been completed. If the interrupt is not being accepted, the transition is performed without interrupt.

Mode 0 to Mode 1 Transition
Mode 0 to Mode 2 Transition
Mode 0 to Mode 3 Transition
Mode 1 to Mode 2 Transition
Mode 1 to Mode 3 Transition
Mode 2 to Mode 3 Transition

The following interrupts result only from the execution of RESTORE instructions which cause a transition to mode 0, mode 1 or mode 2. The interrupt condition occurs after the transition has been completed. If the interrupt is not being accepted, the transition is performed without interruption.

Return to Mode 0
Return to Mode 1
Return to Mode 2

7.6.3 Mode Violation Class

Privileged Operation Code --

An attempt was made to execute an operation instruction in a central processor mode which was less than that required by the operation.

Invalid Segment Number Register Selection --

The segment number register code of an address encountered during an effective virtual address calculation had a value of 1,2, or 3 when the current access mode was 0, or a value of 3 when the current access mode was 1;

Index Segment Number Register Code Greater than the Index Address Segment Number Register Code --

During an indexing operation, the segment number register code contained in the index was greater than the segment number register code in the index address that specified the index.

Illegal Control Transfer --

A control transfer address executed in mode 0 specified a segment number register code other than 0; or a control transfer address executed in mode 1 specified a segment number register code other than 1 or 2; or a control transfer address executed in mode 2 or mode 3 specified a segment number register code other than 2 or 3.

Link Mode Greater than Central Processor Mode --

The value of the mode operand specified in an ANALYZE instruction, or the link mode byte in a referenced subprogram control block was greater than the central processor mode.

Page Mode Conflict --

An attempt was made to access a page and the current access mode was less than the mode specified in the access controls. For write accesses the data is stored into the memory segment specified by segment number register 3.

7.6.4 Illegal Addressing Class

Read with Unassigned Virtual Address --

An attempt was made to read from bulk memory using a virtual address that did not have an entry in the address transformation table. The address transformation search key, beginning search index, current search index, and effective address are stored into the memory segment specified by segment number register 3.

Private Write Access Violation --

An attempt was made to write into a page of bulk memory and its associated private write access bit was set. The data is stored into the memory segment specified by segment number register 3.

7.6.5 Illegal Instruction Class

Unassigned Operation Code --

An attempt was made to execute an unassigned operation code.

Unassigned Private Extended Operation Code --

An attempt was made to execute an unassigned private extended operation code.

Unassigned System Extended Operation Code --

An attempt was made to execute an unassigned system extended operation code.

Illegal Instruction Qualifier --

An instruction qualifier contained either an address count that was inconsistent with the operation code .

Unassigned Address Qualifier --

The contents of the address qualifier byte did not correspond to any of the assigned values.

Illegal Address Type Code --

The specified address type code was inconsistent with the operation code of the instruction, or specified an illegal addressing sequence.

Illegal Length Code --

The specified length code was inconsistent with the operation code of the instruction or the address type code of the address.

Illegal Argument Index --

The value of the argument index was greater than the value of the argument count in the argument list being referenced or was 0.

Illegal Subprogram Control Block Format --

The address qualifier of the argument list address in a referenced subprogram control block did not specify null or long direct addressing.

7.6.6 Illegal Suboperation Class

Illegal List Control Block Format --

During the execution of a list operation the list type code in the list control block did not match the list type implied by the suboperation operand.

Unassigned Subsystem Service Entry Table Index --

Either no subsystem service entry table was active, or the operand specified by the first address of a SUBSYSTEM service instruction had a value that was unassigned as a subsystem service entry table index.

Unassigned System Service Entry Table Index --

Either no system service entry table was active, or the operand specified by the first address of a SYSTEM service instruction had a value that was unassigned as a system service entry table index.

7.6.7 Address Bounds Breakpoint Class

Break Before on Address Subclass --

The following conditions were true:

- The B0-bit in processor state operand zero was set.
- An effective address associated with the current instruction was specified by an active address bounds register set for break before.

The instruction is not executed.

Break Before on Address in Address Bounds Register 0
Break Before on Address in Address Bounds Register 1
Break Before on Address in Address Bounds Register 2
Break Before on Address in Address Bounds Register 3
Break Before on Address in Address Bounds Register 4
Break Before on Address in Address Bounds Register 5
Break Before on Address in Address Bounds Register 6
Break Before on Address in Address Bounds Register 7

Break After on Address Subclass --

The following conditions were true:

- The B0-bit in processor state operand zero was set.
- An effective address associated with the current instruction was specified by an active address bounds register set for break after.

The current instruction execution is completed and then the interrupt occurs.

Break After on Address in Address Bounds Register 0
Break After on Address in Address Bounds Register 1
Break After on Address in Address Bounds Register 2
Break After on Address in Address Bounds Register 3
Break After on Address in Address Bounds Register 4
Break After on Address in Address Bounds Register 5
Break After on Address in Address Bounds Register 6
Break After on Address in Address Bounds Register 7

7.6.8 Instruction Breakpoint Class

The instruction breakpoints are controlled by the state of bits located in the instruction qualifier and in processor state operand zero. An interrupt occurs either before or after an instruction execution is completed as is indicated by the name of the interrupt. The execution breakpoints are discussed in 6.0 Execution Breakpoints.

Break Before on Operation Code --

All of the following conditions were true:

- The B4-bit in processor state operand zero was set.
- The central processor mode was equal to or less than the breakpoint control mode.
- The operation code was one of those specified by the operation code mask.

Break After on Operation Code --

All of the following conditions were true:

- The B5-bit in processor state operand zero was set.
- The central processor mode was equal to or less than the breakpoint control mode.
- The operation code was one of those specified by the operation code mask.

Break Before on Private Extended Operation Code --

All of the following conditions were true:

- The B6-bit in processor state operand zero was set.
- The central processor mode was equal to or less than the breakpoint control mode.
- The private extended operation code was one of those specified by the private extended operation code mask.

Break After on Private Extended Operation Code --

All of the following conditions were true:

- The B7-bit in processor state operand zero was set.
- The central processor mode was equal to or less than the breakpoint control mode.
- The private extended operation code was one of those specified by the private extended operation code mask.

Break Before on System Extended Operation Code --

All of the following conditions were true:

- The B8-bit in processor state operand zero was set.
- The central processor mode was equal to or less than the breakpoint control mode.
- The system extended operation code was one of those specified by the system extended operation code mask.

Break After on System Extended Operation Code --

All of the following conditions were true:

- The B9-bit in processor state operand zero was set.
- The central processor mode was equal to or less than the breakpoint control mode.
- The system extended operation code was one of those specified by the system extended operation code mask.

Break Before Instruction --

All of the following conditions were true:

- The B11-bit in processor state operand zero was set.
- The central processor mode was equal to or less than the breakpoint control mode.
- The B-bit was set in the instruction qualifier of the instruction.

Break After Instruction --

Either of the following sets of conditions was true:

- The B10-bit in processor state operand zero was set and the central processor mode was equal to or less than the breakpoint control mode.
- The B11-bit in processor state operand zero was set and the central processor mode was equal to or less than the breakpoint control mode, and the A-bit in the instruction qualifier of the instruction was set.

Break After on Statement Counter Equal Zero --

Either of the following sets of conditions was true:

- The statement counter is decremented after the execution of each instruction regardless of the state of the S-bit in the instruction qualifier when the B13-bit in processor state operand zero is set and the central processor mode is equal to or less than the breakpoint control mode. The interrupt occurs after the execution of the instruction that caused the statement counter to be decremented to 0.

- The statement counter is decremented after the execution of an instruction when: the B14-bit in processor state operand zero is set, and the central processor mode is equal to or less than the breakpoint control mode, and the S-bit is set in the instruction qualifier of the instruction. The interrupt occurs after the execution of the instruction that caused the statement counter to be decremented to 0.

Break After End of Statement --

The following conditions were true:

- The B12-bit in processor state operand zero was set.
- The central processor mode was equal to or less than the breakpoint control mode.
- The S-bit in the instruction qualifier of the instruction was set.

Break After Each Perform Iteration --

The following conditions were true during the iterative execution of an instruction:

- The B16-bit in processor state operand zero was set.
- The central processor mode was equal to or less than the breakpoint control mode.

7.6.9 Critical Operand Class

Iteration Count Greater Than $2^{32}-1$ --

The operand specifying the iteration count for the iterative execution of an operation has a value greater than $2^{32}-1$.

Illegal Decimal Digit --

An illegal decimal digit was detected in a source operand of a decimal operation.

Illegal Displacement --

Any of the following was detected:

- A COPY instruction specified a positive displacement that exceeded the length of the destination operand or a negative displacement that exceeded the length of the source operand.
- A CONTROL instruction specified a displacement that exceeded 127.
- The displacement operand of a GENERATE instruction specified was a negative value or exceeded the length of the destination operand.

Illegal Generate Specification --

Either the bit count itself or the sum of the bit count and displacement specified by the GENERATE instruction exceeded the length of the destination operand.

7.6.10 Non-Critical Operand Class

Square Root Source Operand Negative --

The source operand of the SQUARE ROOT instruction was negative. If the interrupt is not being accepted, the destination operand is assigned the negative of the square root of the absolute value of the source operand.

Illegal Shift Count --

A SHIFT instruction specified a shift count exceeding the bit length of the operand that was being shifted. If this interrupt is not being accepted, a shift count equal to the length of the shift operand will be used.

7.6.11 List Processing Class

Empty List --

The list control block pointers referenced by a PULL or STEP operation were 0.

End of List --

An attempt was made to STEP off the end of the list.

Free Element Stack Empty --

The next free element number was 0 during the execution of a PUSH instruction.

7.6.12 Overflow Class

The result of an operation has an absolute value too large to be represented in the destination operand. If the interrupt is not being accepted, the following values are assigned to the destination operand depending on the data type and adjustment method:

- REAL types:

  Normalized Method - Absolute Zero

  Automatic Method - Fixed-point part of +0, exponent part of maximum positive value.

  Manual Method - Fixed-point part as calculated, exponent part as calculated with high-order digits truncated.

- COMPLEX types:

  As for REAL types, but real and imaginary components treated separately.

- INTEGER types:

  Result as calculated with high-order digits truncated.

Real Overflow Subclass --

Binary Fixed Real Overflow
Decimal Fixed Real Overflow
Binary Float Real Overflow
Decimal Float Real Overflow

Complex Overflow Subclass --

Binary Fixed Complex Overflow Real Part
Binary Fixed Complex Overflow Imaginary Part
Decimal Fixed Complex Overflow Real Part
Decimal Fixed Complex Overflow Imaginary Part
Binary Float Complex Overflow Real Part
Binary Float Complex Overflow Imaginary Part
Decimal Float Complex Overflow Real Part
Decimal Float Complex Overflow Imaginary Part

Integer Overflow Subclass --

Binary Integer Overflow
Decimal Integer Overflow

7.6.13 Truncation Class

The result of an operation cannot be stored in the destination operand without truncating high order (trailing) digits. If the interrupt is not being accepted, the trailing digits of the result are truncated.

Magnitude Truncation Subclass --

Magnitude Truncation
Address Truncation

Logical Truncation Subclass --

Logical Truncation

7.6.14 Fixed Point Overflow Class

The fixed-point part of the result of a REAL operation or of a component of the result of a COMPLEX operation has been adjusted so that high-order non-zero digits are to the left of the radix point. If the interrupt is not being accepted, the result is stored into the destination operand by truncating the digits to the left of the radix point.

Real Fixed Point Overflow Subclass --

Binary Fixed Real Fixed Point Overflow
Decimal Fixed Real Fixed Point Overflow
Binary Float Real Fixed Point Overflow
Decimal Float Real Fixed Point Overflow

Complex Fixed Point Overflow Subclass --

Binary Fixed Complex Fixed Point Overflow Real Part
Binary Fixed Complex Fixed Point Overflow Imaginary Part
Decimal Fixed Complex Fixed Point Overflow Real Part
Decimal Fixed Complex Fixed Point Overflow Imaginary Part
Binary Float Complex Fixed Point Overflow Real Part
Binary Float Complex Fixed Point Overflow Imaginary Part
Decimal Float Complex Fixed Point Overflow Real Part
Decimal Float Complex Fixed Point Overflow Imaginary Part

7.6.15 Exponent Overflow Class

The exponent part of the result of a REAL operation or of a component of the result of a COMPLEX operation is smaller than the minimum exponent value for the radix of the operation. If the interrupt is being ignored, a value is stored into the destination operand according to the adjustment method being used:

- Normalized Method - Absolute Zero
- Automatic Method - Condition never occurs.
- Manual Method - Fixed-point part as calculated, exponent part as calculated with high-order digits truncated.

Real Exponent Underflow Subclass --

Binary Fixed Real Exponent Underflow
Decimal Fixed Real Exponent Underflow
Binary Float Real Exponent Underflow
Decimal Float Real Exponent Underflow

Complex Exponent Under Subclass --

Binary Fixed Complex Exponent Underflow Real Part
Binary Fixed Complex Exponent Underflow Imaginary Part
Decimal Fixed Complex Exponent Underflow Real Part
Decimal Fixed Complex Exponent Underflow Imaginary Part
Binary Float Complex Exponent Underflow Real Part
Binary Float Complex Exponent Underflow Imaginary Part
Decimal Float Complex Exponent Underflow Real Part
Decimal Float Complex Exponent Underflow Imaginary Part

7.6.16 Divide by Zero Class

The divisor in a divide operation had a value of zero. For REAL and COMPLEX operations the divide by zero interrupt condition always occurs when there is an absolute zero divisor, but a relative zero divisor raises the condition only when the normalized adjustment method is specified. When the interrupt is not being accepted, the value stored into the destination operand depends on the data type and adjustment method as follows:

REAL data types:

- Normalized method - Absolute zero
- Automatic Method - Fixed-point part equals +0, exponent equals maximum exponent value.
- Manual method - As for automatic method.

COMPLEX data types:

As for REAL data types, with the same value stored into both components.

INTEGER, MAGNITUDE, and ADDRESS data types:

Zero in the appropriate data type.

Real Divide by Zero Subclass --

Binary Fixed Real Divide by Zero
Decimal Fixed Real Divide by Zero
Binary Float Real Divide by Zero
Decimal Float Real Divide by Zero

Complex Divide by Zero Subclass --

Binary Fixed Complex Divide by Zero
Decimal Fixed Complex Divide by Zero
Binary Float Complex Divide by Zero
Decimal Float Complex Divide by Zero

Integer Divide by Zero Subclass --

Binary Integer Divide by Zero
Decimal Integer Divide by Zero

Magnitude Divide by Zero Subclass --

Magnitude Divide by Zero
Address Divide by Zero

7.7 STATISTICAL COUNTER INTERRUPT GROUP

The statistical counters interrupt group comprises seven interrupt conditions, which occur when any of seven separate

32-bit counters is decremented to 0. The 7 counters have their count decremented by 1 every microsecond when the following conditions are all true:

- The accept bit is set for the interrupt associated with the counter.
- The central processor is in a particular state, as defined for each counter.

When a count reaches 0, the appropriate interrupt condition occurs, but the counter continues to be decremented until the accept bit for the interrupt is reset. At any time a counter may be given a count value in the range 0 to $2^{32}-1$ by means of a CONTROL instruction.

7.7.1 Mode Counters

There is a separate counter for each of the 4 central processor modes. A counter is decremented once every microsecond while the central processor is operating in the associated mode and the interrupt for the count reaching 0 is being accepted.

Mode Zero Count Equals 0
Mode One Count Equals 0
Mode Two Count Equals 0
Mode Three Count Equals 0

7.7.2 Interrupt Active Count Equals 0

This interrupt is associated with a counter that is decremented once every microsecond while any interrupt is active and the interrupt for the count reaching 0 is being accepted.

7.7.3 Auxiliary Count Equals 0

This interrupt is associated with a counter that is decremented once every microsecond while the interrupt for the count reaching 0 is being accepted. The counter is not associated with any particular central processor state.

7.7.4 Page Usage Table Count Equals 0

This interrupt is associated with a counter that is decremented once every microsecond while the interrupt for the count reaching 0 is being accepted and the page usage table is active. The table is active whenever page usage statistics are being kept for pages addressed with one or more segment number register codes. The interrupt also occurs when the page usage table is full and a page number not in the table is addressed. Page usage statistics can be gathered by reading and clearing the entries in the page usage table at specific intervals and whenever the table is full. The statistics are read and cleared by means of the CONTROL instruction.

7.8 QUANTUM TIMER INTERRUPT GROUP

Quantum Timer Equals Zero

The quantum timer equals zero interrupt condition occurs when either of the following are true:

- The counter associated with the quantum timer has been decremented to 0 and the central processor mode is 0 or 1. If the timer decrements to 0 when the central processor mode is not 0 or 1, the interrupt condition does not occur.

- The counter associated with the quantum timer has been decremented to 0 and the central processor mode is 2 or 3, and a CONTROL instruction specifying dismissal was executed.

8.0 MEMORY PROCESSOR

The purpose of the memory processor is to control and process all accesses to bulk memory. The memory processor interfaces with central processors, input/output processors, the interprocessor control bus, the maintenance panel, and bulk memory.

The memory processor reduces the effective access time to bulk memory, transforms virtual addresses to real addresses, and protects memory from unauthorized accesses. Pages of memory can be protected from write access by a central processor or by the input/output processors (which in turn interface with communication terminals and peripheral equipment through communication and peripheral interface units), or by both the central and the input/output processors. In addition a page mode access control is associated with each page of memory and examined during central processor accesses.

8.1 MEMORY PROCESSOR FUNCTIONS

Functionally, the memory processor consists of 3 units: the high speed memory unit, the address transformation control unit, and the interface control unit. The high speed memory functions as a buffer containing those portions of bulk memory that are currently being referenced by the central processors.

The contents of the buffer memory are maintained by a technique that automatically replaces infrequently used information. The address transformation control unit transforms virtual addresses to real addresses. The interface control unit accesses bulk memory and performs error checks on both the data and the addresses. Figure 8.1 provides an overview of memory processor operation.

Memory requests are initiated when a central processor presents a complete virtual address, or when an input/output processor presents both a complete virtual address and a real address to the memory processor. First an attempt is made to access the data from the high speed memory. If the requested data is in the high speed memory buffer and the memory protection controls permit the access, the data transfer is performed between the high speed memory and the requesting processor.

If the data is not in the high speed memory and an input/output processor initiated the request, the access is performed directly to bulk memory using the real address supplied with the request. Provided that the memory protection controls permit the access, the data transfer is performed between bulk memory and the requesting processor. Note that high speed memory is never loaded to satisfy a memory request from an input/output processor.

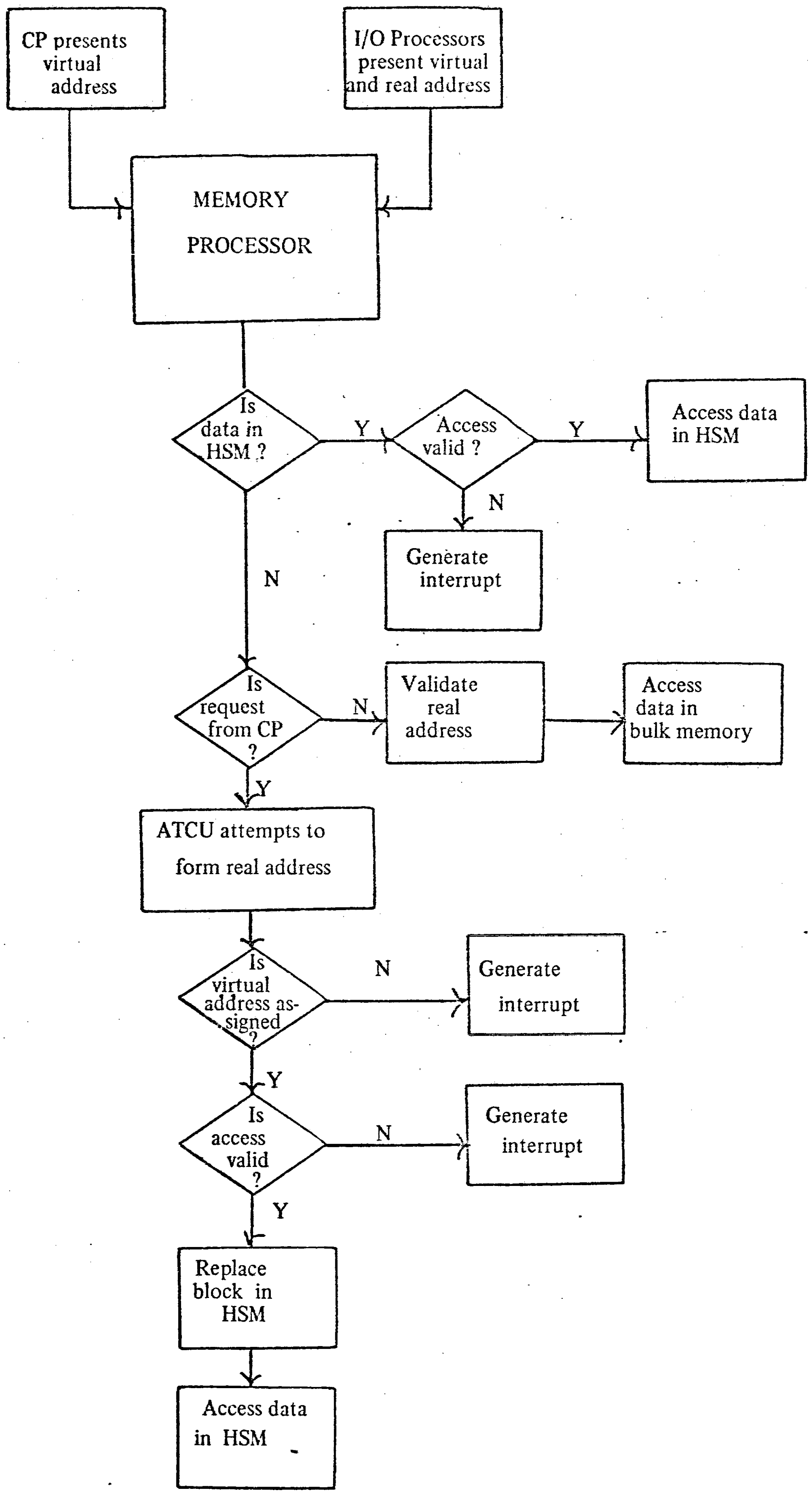


Figure 8.1 Overview of Memory Processor

If the data is not in the high speed memory and a central processor initiated the request, the virtual address is presented to the address transformation control unit. If a real address is associated with the virtual address, the data is accessed from bulk memory and loaded into high speed memory. If the memory protection controls are not violated, the data is transferred between the high speed memory and the requesting processor.

8.2 HIGH SPEED MEMORY UNIT

The high speed memory unit includes a random access buffer memory, a content addressable memory,and a device to manage the replacement of data in the buffer memory.

8.2.1 Random Access Buffer

The random access buffer memory has a size of 8,192 or 16,384 bytes, depending on the model of the memory processor. It reduces effective bulk memory access time by retaining those blocks of memory that are currently being accessed by central processors. Each block is 64 contiguous bytes; thus, 128 or 256 blocks of bulk memory can be held in the high speed buffer memory depending on the model of the processor.

8.2.2 Content Addressable Memory

The content addressable memory of the high speed memory unit identifies the blocks of memory that are currently contained in high speed buffer memory and also specifies the access control information associated with the blocks. A block of memory is identified by its virtual block address. A virtual block address is formed by dividing the complete virtual address by 64; thus, the high order 40 bits of the complete virtual address specify the virtual block address.

8.2.3 High Speed Memory Replacement Device

The maintenance of current data blocks in the high speed buffer memory is controlled by the replacement device. This device maintains an activity list, with each entry in the list corresponding to a block of high speed memory. The list is ordered on a least recently used basis. Although both central processor and input/output processor requests cause high speed memory to be accessed, only central processor requests affect the activity list. When it is necessary to replace a block of high speed memory, the block that was least recently accessed by a central processor is selected for replacement.

8.2.4 High Speed Memory Access

When a virtual block address is presented to the high speed memory unit, the content addressable memory is accessed to determine if the requested block is currently in high speed memory. If the block is in high speed memory and the access controls associated with the block permit the access, a data transfer is performed between the high speed buffer memory and the requesting processor.

When an input/output processor requests data that is not contained in high speed memory, the interface control unit of the memory processor performs the access to bulk memory using the real address supplied by the input/output processor. The data transfer is then performed between bulk memory and the requesting processor.

When a central processor requests data that is not contained in high speed memory, the virtual address is presented to the address transformation control unit. If a real address is currently associated with the virtual address, the requested block is accessed from bulk memory. The data is loaded into the block of high speed memory selected by the high speed memory replacement device.

Although high speed memory is accessed by virtual address, the real address of each block in high speed memory is maintained so that it is unnecessary to transform the virtual address in order to store the block selected for replacement into bulk memory. If the block selected for replacement has not been modified, the contents of the block are not stored since an identical copy of the block exists in bulk memory; however, if a write access occurred to the block selected for replacement, the contents of the block are written into bulk memory using the real address associated with the block.

8.3 ADDRESS TRANSFORMATION CONTROL UNIT

The address transformation control unit includes content addressable and random access memories, a replacement device, and a hashing and search device. The hashing and search device performs searches on the address transformation table (described in 8.3.1).

8.3.1 Address Transformation Table

The address transformation table, located in memory, maintains a record of all assigned pages of virtual memory. It also includes the access control information associated with each page.

Each address transformation table entry is 8 bytes in length and contains a virtual page number, a real page number, a segment number, and a control field.

The control field includes page access controls, a secondary storage control, a use control, and 3 unassigned bits. The format of an address transformation table entry is shown in Figure 8.2.

The page access controls include a private write access control and a page mode access control. These controls are discussed in 2.2.4 Page Access Controls.

The secondary storage control, if set, specifies that the virtual page address corresponds to a page in secondary storage (i.e., on a peripheral device rather than in bulk memory).

The use control, if set, specifies that the entry is active. An entry is considered active if the virtual page address has been assigned to either a real bulk memory page or a page of secondary storage.

The number of entries in the address transformation table is equal to 9/8 times the sum of the number of pages of bulk memory and the number of pages of secondary storage allocated for the storage of virtual memory.

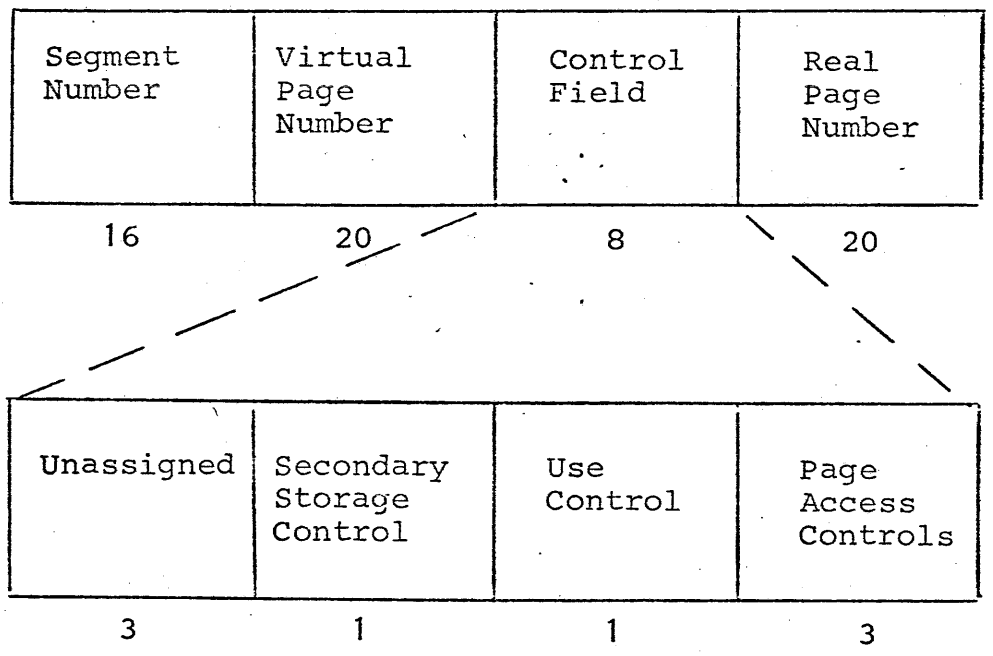


Figure 8.2 Address Transformation Table Entry

The number of possible virtual page addresses is far greater than the number of address transformation table entries, and the virtual page addresses that are in use at any time are unpredictable. To reduce search time, a hashing algorithm is used to assign and to access the virtual page addresses in the address transformation table. The hashing algorithm uniformly distributes throughout the table the virtual page addresses in use at any time. The address transformation control unit uses the algorithm to locate the address transformation table entry corresponding to a given virtual page address or to determine that no such entry exists.

The address transformation table occupies contiguous pages in real and virtual memory. The address transformation control unit has registers that specify both the virtual page address and the real page address of the first of these pages. These registers can be accessed under program control by the execution of a privileged (SELECT) instruction.

8.3.2 Address Transformation Control Unit Content Addressable and Random Access Memories

The content addressable and random access memories function together to reduce the time required to perform an address transformation. These memories maintain the 32 address transformation table entries that have been most recently accessed.

8.3.3 Address Transformation Control Unit Replacemtnt Device

The maintenance of current data in the address transformation control unit is controlled by the replacement device. Similar to the high speed memory replacement device, this device maintains an activity list with each entry in the list corresponding to an entry in the content addressable and random access memories. When an entry must be replaced, the entry that was least recently accessed by a central processor is selected for replacement.

The input to the address transformation control unit is a virtual page address referred to as a search key. The address transformation control unit content addressable memory is accessed, and if the requested virtual page address is present and the access controls are not violated, the associated real page address is obtained from the random access memory and presented to the interface control unit.

If the requested search key is not in the content addressable memory, the hashing and search subunit searches the address transformation table. Interlock flags control the searching of the address transformation table and prevent one central processor from reading the table when the other central processor is modifying it.

The search key is input to the hashing algorithm, which produces a number referred to as the beginning search index. The beginning search index is a byte displacement into the address transformation table that selects the entry where the search is to begin. Note that the beginning search index is not unique to a given search key because the number of possible search keys (i.e., virtual page addresses) exceeds the number of address transformation table entries.

In order to search the address transformation table, the virtual page address and real page address are obtained from the registers that specify the beginning of the address transformation table. These addresses are converted into byte addresses and the beginning search index is added to each of these addresses, forming beginning search addresses. Both the virtual and real beginning search addresses are presented to the high speed memory unit. In a manner similar to the processing of input/output processor requests, the data (the address transformation table entries) is accessed either from high speed memory or directly from bulk memory.

The search of the address transformation table proceeds from lower to higher search addresses, wraps around the end of the table if necessary to continue the search, and terminates when any of the following occurs:

- An entry is encountered containing a virtual page address identical to the search key and assigned to a real page.
- An entry is encountered containing a virtual page address identical to the search key and assigned to a page of secondary storage.
- An unused entry is encountered.

If the requested virtual address has been associated with a real address and the access controls are not violated, the real address is obtained and presented to the interface control unit. The address transformation control unit memories are updated to contain this virtual page address and its corresponding real page address and access controls.

If the virtual page address is assigned to a page of secondary storage, the <u>read with secondary storage address</u> or the <u>write with secondary storage address</u> interrupt is generated for the central processor that presented the search key.

If an entry that is not in use is encountered, either the <u>read with unassigned virtual address</u> or the <u>write with unassigned virtual address</u> interrupt is generated for the processor that presented the search key. It is possible to determine that the search key is unassigned without searching the entire address transformation table due to the manner in which the table entries are deleted.

Whenever a deletion is performed, subsequent entries are rearranged so that there are no unused entries between any active entry and the entry specified by its beginning search index.

8.3.4 Unassigned Page Interrupts

Whenever the read or write with unassigned virtual page interrupt or the read or write with virtual page in secondary storage interrupt is generated, information pertaining to the interrupt condition is stored into the segment specified in segment number register (3) of the central processor that presented the search key. The format of the information and the locations used for storage are shown in Figure 8.3.

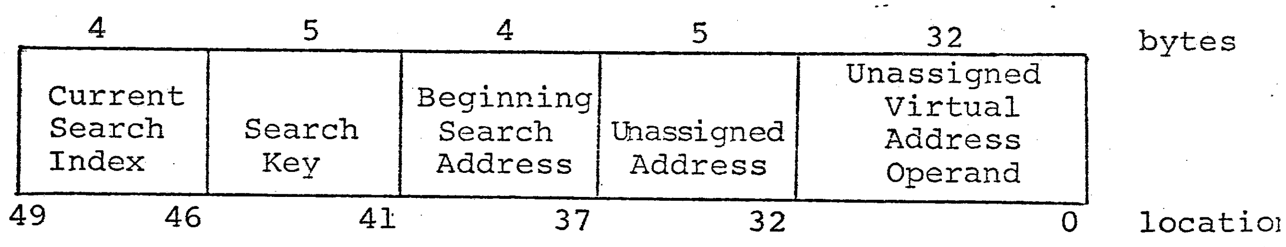


Figure 8.3 Unassigned Page Interrupt Information

If a write access was attempted, the operand is stored into the leading byte positions of the unassigned virtual address operand field. This field is 32 bytes in length and occupies location 0 through 31.

The virtual address causing the interrupt is stored as a long direct address (addressing type code 9) into the unassigned address field. The length code associated with the address is 0 if the interrupt resulted from an attempt to transfer control to an unassigned page. This field is 5 bytes in length and occupies locations 32 through 36.

The beginning search index is stored into the <u>beginning search index</u> field. This field is 4 bytes in length and occupies locations 37 through 40.

The virtual page address causing the interrupt condition is stored into the <u>search key</u> field. This field is 5 bytes in length and occupies locations 41 through 45.

Similar to the beginning search index, the <u>current search index</u> is a byte displacement into the address transformation table and locates the entry being examined when the interrupt condition occurred. The current search index is stored into locations 46 through 49.

8.4 INTERFACE CONTROL UNIT

The <u>interface control unit</u> performs accesses to bulk memory. This unit consists of memory protection controls, error checking circuitry, and bulk memory interface circuitry.

The memory protection controls consist of separate write access locks for each page of bulk memory. These locks prevent unauthorized write accesses for an input/output processor.

The real address is examined, and a diagnostic interrupt is generated if the address exceeds limits determined by the size of bulk memory. If the address is within acceptable limits, a block is accessed from bulk memory.

Each time an access is performed, the error chedking circuitry detects and corrects single-bit errors and can generate a diagnostic interrupt indicating that the concition occurred. All double-bit, and some multiple-bit errors are also detected and reported via the interrupt mechanism.

In addition to the error detection, the real address is validated. Stored with each block of data is the associated real address. When a data access is performed, the requested real address is compared with the real address associated with the block. A diagnostic interrupt is generated if the two addresses are not equal.

9.0 PROCESSOR STATE VECTOR

Each central processor contains registers that control and report the state of the processor. As a group these registers are called the processor state vector. Individual registers are grouped to form processor state operands, which can be accessed under program control with the privileged CONTROL instruction. A processor state operand has a length of 1 to 16 bytes. The contents of processor state operand zero are described in detail in 9.1 Processor State Operand 0. Contents of the other processor state operands are discussed briefly in 9.2 Other Processor State Operands.

9.1 PROCESSOR STATE OPERAND 0

Processor state operand 0 is 16 bytes long. It contains the information and controls that are automatically saved in memory when any interrupt is activated. Servicing of an interrupt begins when new processor state operand zero contents with appropriate control settings required for interrupt servicing are obtained from memory (see 7.4 INTERRUPT PROCESSING).

The registers that make up processor state operand zero are described in 9.1.1 through 9.1.10. Figure 9.1 shows the format of these registers. Note that 6 bits of the 16-byte operand are unused.

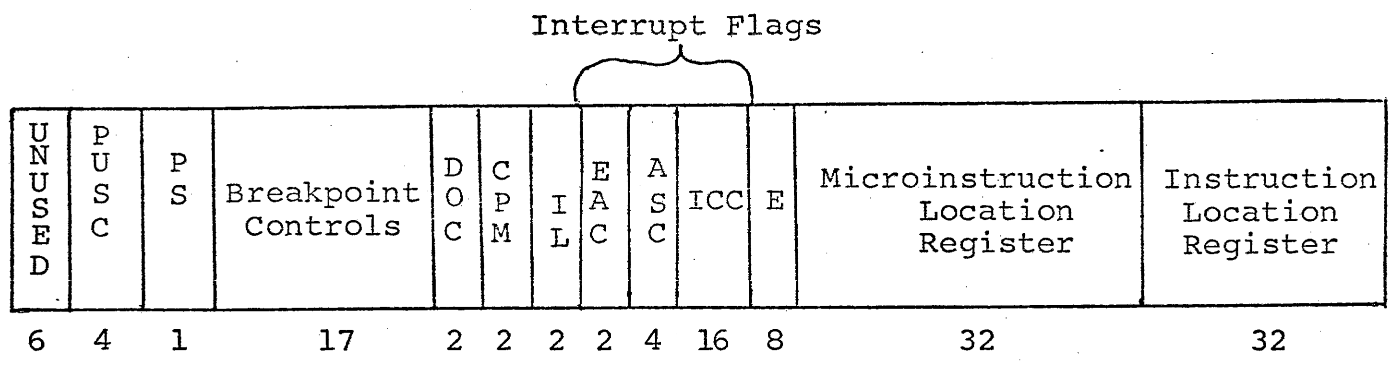


Figure 9.1 Processor State Operand 0

9.1.1 Instruction Location Register

The instruction location register is 32 bits long and contains the effective address of the instruction currently being executed by this central processor. The instruction location register contains a 30-bit address field and a 2-bit segment number register code field.

9.1.2 Microinstruction Location Register

The microinstruction location register is 32 bits long and contains the effective address of the microinstruction currently being executed by this central processor. The microinstruction location register contains a 30-bit address field and a 2-bit segment number register code field.

9.1.3 Emulate Control

The emulate (E) control is 8 bits long and selects one of 256 possible microprogram sets.

9.1.4 Interrupt Flags

The interrupt flags comprise the 16-bit interrupt condition code (ICC), the 4-bit address selection code (ASC), and the 2-bit effective addressing code (EAC). The use of these fields is described in 7.3 Interrupt Flags.

9.1.5 Interrupt Locks

The interrupt locks (IL) consist of 2 control bits. The first, when set, prevents activation of the private interrupt groups for this central processor. The second, when set, prevents activation of the shared interrupt group for this central processor.

9.1.6 Central Processor Mode

The central processor mode (CPM) is a 2-bit field that indicates the mode in which this central processor is currently operating. The central processor mode constrains instruction and data accesses and the execution of privileged instructions.

9.1.7 Diagnostic Override Controls

The diagnostic override controls (DOC) consist of 2 control bits. The first, when set, overrides page mode access restrictions. The second, when set, overrides mode transfer restrictions.

9.1.8 Breakpoint Controls

The breakpoint controls are 17 bits identified as B0 through B16. The names of the individual controls are listed below. The use of these controls is discussed in 6.2 Processor State Vector Breakpoint Controls and in 7.6.7 Address Bounds Breakpoint Class and 7.6.8 Instruction Breakpoint Class.

B0	Enable prior instruction location register
B1	Enable address bounds register controls
B2	Breakpoint control mode
B3	
B4	Operation code breakpoint before
B5	Operation code breakpoint after
B6	Private extended operation code breakpoint before
B7	Private extended operation code breakpoint after
B8	System extended operation code breakpoint before
B9	System extended operation code breakpoint after
B10	Instruction breakpoint after
B11	Enable A-bit and B-bit in instruction qualifier
B12	End of statement control
B13	Instruction counter control
B14	Enable S-bit in instruction qualifier
B15	Enable P-bit in instruction qualifier
B16	Perform breakpoint control

9.1.9 Perform State Control

The perform state (PS) is a single control bit. When set, it indicates that the current instruction is being executed iteratively (see 5.3 Initialization for Iterative Execution).

9.1.10 Page Usage Statistics Controls

The page usage statistics controls (PUSC) are 4 control bits, numbered 0 to 3. They control the keeping of page usage statistics for pages addressed using segment number register codes 0, 1, 2, and 3, respectively.

9.2 OTHER PROCESSOR STATE OPERANDS

The other processor state operands contain control and report registers whose contents do not necessarily have to be saved during interrupt activation. The following sections enumerate these registers.

9.2.1 Segment Number Registers

The segment number registers are five 16-bit registers numbered from 0 to 4. Each contains a segment number that can be used in the formation of a complete virtual address. Segment number register selection is discussed in 2.2 Central Processor Memory Accesses.

9.2.2 Perform Controls

The perform controls contain the information needed for iterative execution of the current instruction (see 5.3 Initialization for Iterative Execution). They include the operation code and instruction qualifier of the instruction being executed iteratively, and the iteration count.
For each of the 5 possible execution operands, the following information is kept in the perform controls: address qualifier, index address, index value, index increment, effective address, and effective address increment. For immediate operands, the operand itself replaces the index value and increment and the effective address and increment.

9.2.3 Breakpoint Registers

The breakpoint registers are used in conjunction with the breakpoint controls in processor state operand zero. These controls include:

- The operation code masks,which allow selection of any operation codes and extended operation codes for breakpointing.
- The 8 address bounds registers,which allow protection of 8 ranges of effective address; the address bounds controls enable address bounds checking.
- The statement count,which provides breakpointing after execution of a specified number of instructions or statements
- The prior instruction register,which contains the effective address of the instruction executed immediately before the current instruction

For a full description of breakpoint controls located in the processor state vector, see 6.2 Processor State Vector Breakpoint Controls.

9.2.4 Interrupt Controls

The interrupt controls include the hold and dispatch controls for the various interrupt priority groups and the accept bits for the individual private interrupt conditions.

9.2.5 Computational Controls

The computational controls comprise a rounding and a normalization bit for each of the 8 computational data types (see 4.3 Computational Arithmetic).

9.2.6 Service Entry Table Registers

The service entry table registers contain a service entry table base address and a service entry table index limit for the system and subsystem service entry tables.

9.2.7 System Controls

The system controls are set to indicate the working state of various private and shared system components, such as the high speed memory unit, the address transformation control unit, and the arithmetic units.

Mar 2003:
From a talk given at UC-Berkeley by
Al Burns from Computer Operations, Inc.
of Costa Mesa, California.