HEWLETT-PACKARD CO.

MISERS

Ltr	Revisions	Date	Appr'd
A	As Issued	12/06/82	DH

			MODEL	STK #
			Focus Machine Instruction Set ERS	
			BY J. Fiasconaro	DATE 09/23/82
LT	P.C. #	APPR / DATE	APPD	SHEET # 1 OF 150
REVISIONS			SUPERSEDES	DWG # A-1FE1-3020-8

TABLE OF CONTENTS

1. INTRODUCTION

The standard machine instruction set for the FOCUS system is patterned after the AMIGO machine instruction set. The changes that have been made are largely the result of architectural differences between AMIGO and FOCUS, in particular, the FOCUS system has 32-bit words in memory, 32-bit registers on the CPU chip, an I/O structure that is different from AMIGO, and chip to chip communication requirements that are different from AMIGO's. In addition, FOCUS supports a different decimal number format and binary floating point number format.

In preparing this document, an attempt has been made to avoid the assumption that the reader is familiar with AMIGO. However, anyone familiar with the instruction set for AMIGO (or for the 3000) will immediately notice the similarities. Care should be taken to avoid confusing the instruction sets.

1.1 OVERVIEW OF THE OPERATING ENVIRONMENT

The operating environment provided by the FOCUS machine instruction set is basically that of a stack oriented machine in which segmentation is used to facilitate memory management. A simplified diagram of the operating environment is shown in Figure 1.1. There are two basic types of segments, program or code segments and data segments. Each executable program resides in one or more code segments. Interrupt service routines also reside in code segments. In addition, each program uses one data segment as an execution stack and at least one data segment as a global data area. The information required to manage these segments is kept in two tables, the Code Segment Table and the Data Segment Table. The Device Reference Table has an entry for each I/O device and this entry contains the Code Segment Table entry number for the service routine and the Data Segment Table entry number for the global data area. Dedicated memory locations contain pointers to the first entry in the Code Segment Table, the Data Segment Table and the Device Reference Table. Other dedicated memory locations contain pointers to the start and end of the Interrupt Control Stack which serves as the execution stack for most interrupt service routines. Finally, there is a dedicated memory location containing a pointer to the current Task Control Block which contains information relevant to the currently

executing task. This information includes the Data Segment Table entry numbers for the stack segment and global data segment. A set of registers on the CPU point to the currently active segments and all addressing is done relative to these registers.

1.1.1 Program, Stack and Data Segments

The machine instructions for each executable program are kept in one or more program segments. As indicated in Figure 1.2, there are three pointers associated with a program segment. The

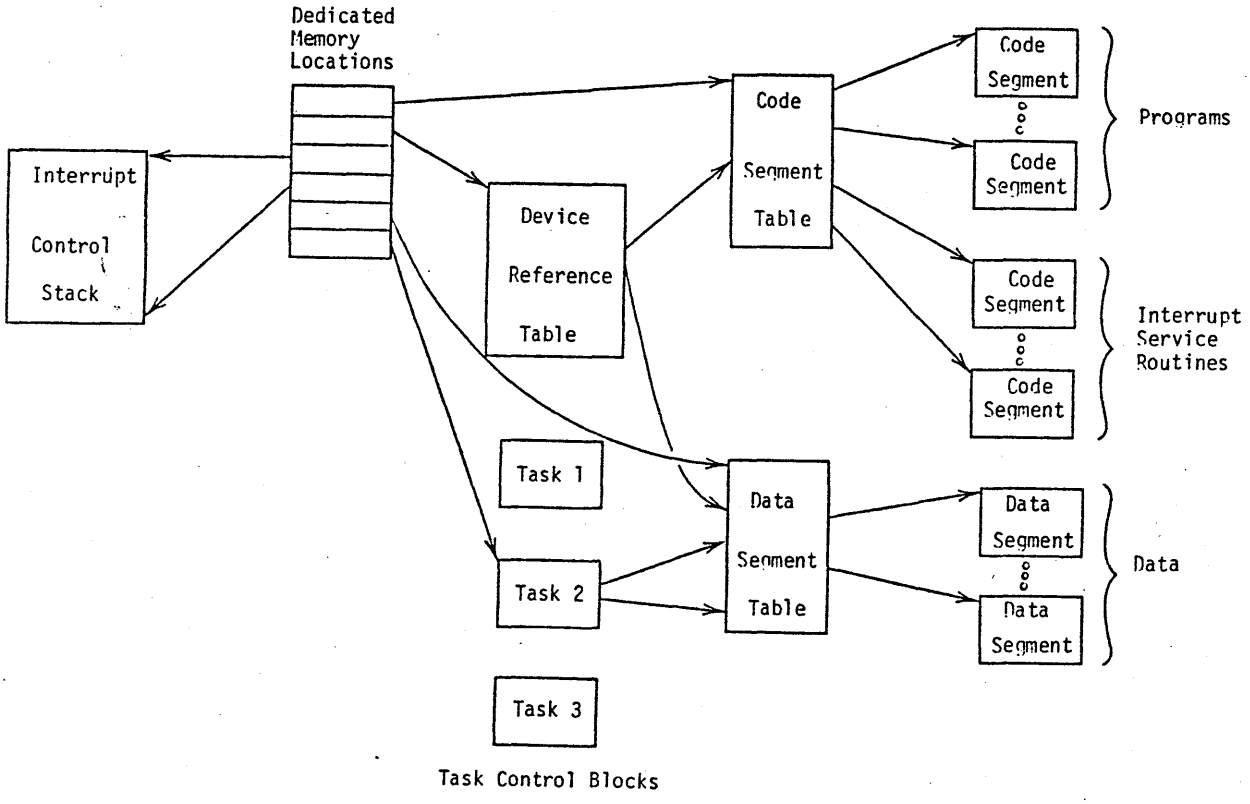


Fig. 1.1 The Operating Environment

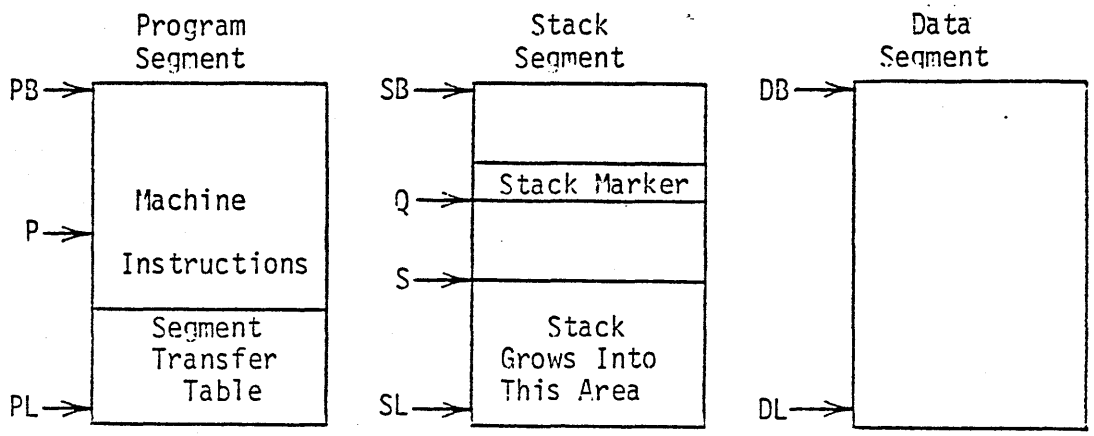


Figure 1.2 The Segmentation Scheme

values of these pointers for the currently executing program are kept in registers on the CPU chip. These registers are the Program Base, PB, register which points to the smallest memory address occupied by the code segment, the Program Limit, PL, register which points to the largest memory address occupied by the code segment, and the Program Counter, P, which points to the current instruction. The PB and PL registers are used for bounds checking all references to the code segment. The Segment Transfer Table contains the information required to transfer control from one procedure to another procedure, possibly in a different code segment.

The memory required for passing parameters, storing local variables, saving the context of the machine on a procedure call, and evaluating mathematical expressions is allocated in a stack segment which is also shown in Figure 1.2. The Stack Base, SB, and Stack Limit, SL, registers are analogous to the PB and PL registers. The Stack Pointer, S, points to the current top-of-stack. On every procedure call the status of the machine is saved in a stack marker. The Q register points to the stack marker closest to the top-of-stack.

Global data is stored in a data segment. The Data Base, DB, and Data Limit, DL, registers are used to bounds check all references to the data segment. The data segment is also shown in Figure 1.2.

Large arrays of local or global data which do not fit conveniently into the stack segment or the global data segment can be placed in one or more external data segments. In this case there are no registers on the CPU chip pointing to the start and end of segments. Data in these segments is accessed indirectly via external data segment pointers stored in the stack and global data segments. All references to these segments are checked by the microcode using information in the Data Segment Table.

1.1.2 Code and Data Segment Tables

The program, stack and data segments for the system are managed through two tables, a Code Segment Table and a Data Segment Table. Stack and data segments are both handled through the Data Segment Table. The entries in the tables are set up by the operating system and used both by the operating system and by the microcode. The table entries contain the length, the current location (which may be in memory or elsewhere), and the status of

each segment. Portions of each table are reserved for operating system code and data segments and the remaining portions are reserved for user code and data segments.

1.1.3 Privileged and Unprivileged Mode

Two modes of operation are provided by the machine instruction set: privileged mode and unprivileged mode. Privileged mode, which is generally available only to operating system software, provides the ability to manipulate all aspects of the machine. Unprivileged mode limits this ability in certain areas. For example, in unprivileged mode it is not possible to write into a program segment, or to modify the limit registers defining the bounds of a segment. An unprivilege user can modify only his own stack, global, and external data segments. In addition, there is a set of machine instructions (e.g. the dispatch instruction, most of the I/O instructions and special load and store instructions) which cannot be executed in unprivileged mode. This mode scheme is designed to make it easier to write operating systems for the FOCUS processing system.

1.2 OVERVIEW OF THE REST OF THE ERS

A complete description of the machine defined by the machine instruction set is presented in Section 2. The registers accessible by the instruction set, the addressing conventions and the standard data formats are described. The segmentation scheme, including the Code and Data Segment Tables, is described in detail in Section 3. A summary of the I/O system is presented in Section 4. Internal and external interrupts and internal traps are described in Section 5. A detailed description of each machine instruction is presented in Section 6. Finally, a sample program is described in Section 7.

2. DESCRIPTION OF THE MACHINE

As indicated in Section 1, there are nine registers on the CPU chip which are used to point to the currently active set of code and data segments. In addition to these registers, the machine instruction set provides access to nine other registers: an Index register, a Status register, a Flags register, a Message register, a Message Mask register, a Breakpoint register, a Slave Address register, a Slave Data register, and a register which points to the current user's data segment table. In order to facilitate the relocatability of code and data segments, all addressing is done relative to some register. Direct, indirect, indexed and indirect-and-indexed addressing modes are provided. Segment bounds checking is done by the microcode on every memory reference. Machine instructions are provided to handle a variety of integer, floating-point number, decimal number and string operations.

2.1 REGISTERS

The registers that are accessible from the machine instruction set fall into four categories: those related to program segments, those related to stack segments, those related to data segments, and those related to the status of the machine. All of these registers, except for the Flags register, are 32-bits wide. The Flags register is 8-bits wide.

2.1.1 P, PB, and PL Registers

There are three registers which point to the currently executing code segment.

- P: Program Counter - contains the absolute address of the instruction being executed. The LSB of P must be 0.
- PB: Program Base Register - contains the smallest absolute memory address occupied by the code segment being executed. The two LSB's of PB must both be 0.
- PL: Program Limit Register - contains the absolute memory address of the last word in the code segment being executed. The two LSB's of PL must both be 0.

2.1.2 SB, SL, S, and Q Registers

There are four registers which point to the current stack segment.

SB: Stack Base Register - contains the smallest absolute memory address occupied by the execution stack.
SL: Stack Limit Register - contains the largest absolute memory address occupied by the stack segment.
S: Stack Pointer - contains the absolute memory address of the top-of-stack.
Q: Stack Marker Pointer - contains the absolute memory address of the stack marker which is closest to the top-of-stack.

The two LSB's of SB must both be 0 and the two LSB's of SL, S, and Q must both be 1.

2.1.3 DB, DL, and DST Registers

There are three registers associated with the current global data segment.

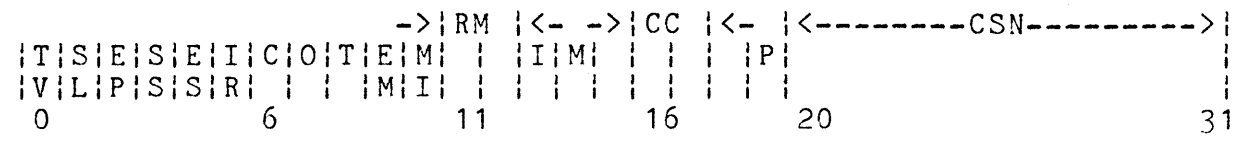
DB: Data Base Register - contains the smallest absolute memory address occupied by the data segment.
DL: Data Limit Register - contains the largest absolute memory address occupied by the data segment.
DST: Data Segment Table Pointer - contains the absolute memory address of the first entry of the current user's Data Segment Table.

The two LSB's of DB and DST must both be 0 and the two LSB's of DL must both be 1.

2.1.4 Other Registers

There are eight other registers accessible from the machine instruction set.

X: Index Register - contains the byte offset to be used when an indexed addressing mode is specified.
STATUS: Status Register - contains the current status of the machine. The format is:



where:

P = Instruction Pending Bit - set to 1 by an instruction if it gets interrupted. This bit is cleared when the instruction is finally completed.

TV = Trace Variables Bit - set to 1 to enable the trace variables trap.

SL = Start of Line Bit - set to 1 to enable the start of line trap.

EP = End of Procedure Bit - set to 1 to enable the end of procedure trap.

SS = Start of Subroutine Bit - set to 1 to enable the start of subroutine trap.

ES = End of Subroutine Bit - set to 1 to enable the end of subroutine trap.

M = Mode Bit - set to 1 for privileged mode.

I = Interrupt Bit - set to 1 to enable Message Register interrupts.

T = Trap Bit - set to 1 to enable user traps.

O = Overflow Bit - set to 1 by the microcode if overflow occurs on an arithmetic operation. This bit is cleared only by the Branch on Overflow, Branch on No Overflow, and Set Registers instructions.

C = Carry Bit - set by the microcode to the value of the carry generated by an arithmetic operation.

MI = Machine Instruction Trap Bit - set to 1 if a trap prior to each machine instruction is desired. (See EM.)

EM = Enable Machine Instruction Trap - set and cleared appropriately by the microcode to allow a trap prior to each machine instruction. This trap occurs only if EM and MI are both 1.

CC = Condition Code - CCG = 00, CCE = 10, CCL = 01 or 11.
(When set by the hardware, only 01 is used for CCL.) There are three interpretations for the Condition Code Bits
CCA sets CC = CCL if operand <0
CC = CCE if operand =0
CC = CCG if operand >0

CCB sets CC = CCL if special ASCII character
CC = CCE if upper or lower case alphabetic ASCII character [a-z and A-Z]
CC = CCG if numerical ASCII character[0-9]
CCC sets CC = CCL if operand 1 < operand 2
CC = CCE if operand 1 = operand 2
CC = CCG if operand 1 > operand 2

RM = Rounding Mode for binary floating point operations - default is 00
00 - Round to Nearest Even
01 - Round toward Plus Infinity
10 - Round toward Zero
11 - Round toward Minus Infinity

IR = Inexact Result Bit - set to 1 by binary floating point instructions if the result is not exact. This bit is cleared only by the Branch on Inexact Result and Set Registers instructions.

CSN= Code Segment Number - the Code Segment Table entry number for the currently executing code segment.

FLAGS: Flags Register - an 8-bit register used in connection with the debug aid instructions(SOL, SOP, EOP, etc.). The flag bits are intended to be used as follows:

Bit 0: Set to 1 if any breakpoints or line traces have been requested anywhere in the current user's program (not just in the current procedure).

Bit 1: Set to 1 if any breakpoints or line traces have been requested anywhere in the operating system (which includes both the strictly operating system code and sharable code).

Bit 2: Set to 1 if the current user is stepping through any of his procedures.

Bit 3: Set to 1 if any operating system procedures are being stepped through.

Bit 4: Set to 1 if line count data is being generated for any of the current user's procedures.

Bit 5: Set to 1 if line count data is being generated for any of the procedures in the operating system.

Bit 6: Set to 1 if any variables in the current user's program are being traced.

Bit 7: Set to 1 if any variables in the operating system are being traced.

These eight bits and the two most significant bits of the code segment number are used to derive an overall debug bit as follows:

MSB'S of CSN	Overall Debug Bit = 1 only if
00	Flag Bits 1 or 3 or 5 or 7 = 1
01	Flag Bits 1 or 3 or 5 or 6 or 7 = 1
10	Flag Bits 1 or 3 or 5 or 6 or 7 = 1
11	Flag Bits 0 or 2 or 4 or 6 = 1

MSG: Message Register - contains a bit for each condition which can interrupt the CPU. A 1 in a bit position causes an interrupt. The bits are defined as follows:

Bit	
0	Memory double bit error
1	MPB slave address error
2	MPB slave data error
3-5	Always zero
6	MPB slave channel write data valid
7	Illegal I/O opcode
8	MPB error detected by I/O
9	Double bit error detected by I/O
10-11	Unused
12	Used by WTC instruction
13	Timer Interrupt
14	I/O Attention Request Acknowledge
15	Memory Controller interrupt
16-31	I/O device interrupts

Consult the I/O, and Memory Controller ERS's for more details.

MASK: Message Mask Register - used to disable bits in the Message register. A 1 in a particular bit position in the Mask register will disable interrupts from the corresponding bit position in the Message register. This does not prevent the Message register bits from being set. Bits 3-5 contain the CPU channel number.

BRKPT: Breakpoint Register - set to the absolute address of a machine instruction. A Breakpoint Trap will occur just prior to executing this instruction.

SAR: Slave Address Register - receives the address half of an address-data pair written to the CPU by some processor on the MPB. Consult the Memory Controller ERS for more details.

SDR: Slave Data Register - receives the data half of an address-data pair written to the CPU by some processor on the MPB. Consult the Memory Controller ERS for more details.

2.2 ADDRESSING CONVENTIONS

In general, specification of an address in memory requires the specification of a register to be used as a base register, and an offset which is to be added to or subtracted from the content of the base register. It is also necessary to indicate whether indirect addressing or indexed addressing or both is desired. Segment bounds checking is performed by the microcode for all memory references.

2.2.1 Top-of-Stack Addressing

Many machine instructions operate on the top elements of the stack and, as a result, require no explicit address specification in the instruction. For example, the integer add instruction adds the top two words in the stack as 2's complement integers, deletes these two words from the stack, and pushes their sum onto the stack.

2.2.2 Direct Addressing

The location of an operand or a branch target may be indicated by a direct addressing convention in which a base register and offset are specified explicitly in the instruction. The offset is always a positive quantity and the indication of whether this offset is to be added to or subtracted from the contents of the base register is contained in the base register specification. The available base register specifications are: P+ and P- which are used for branches and access to literals stored in program segments, SB+ which is used to access information in the stack segment, Q+ which is used for access to local variables, Q- which is used for addressing passed parameters, S- which is used for accessing temporary variables in subroutines, and DB+ and DL- which are used for addressing global variables. The offset is usually a 19-bit byte offset. The exceptions to this rule are outlined in Section 6.2.

2.2.3 Indirect Addressing

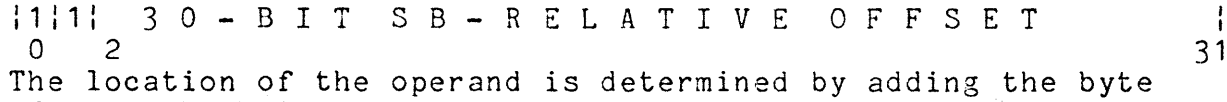
With indirect addressing, which is indicated by one bit in the instruction, the base and offset information in the instruction is utilized as it is in the direct addressing case to fetch a word from memory. However, in the indirect case this word is not the desired operand but is a pointer to the desired operand. Four different pointer formats are used: Self-relative pointers, stack segment pointers, global data segment pointers, and external data segment pointers. The latter three types are

collectively referred to as data segment pointers.

Self-relative pointers have the following format:

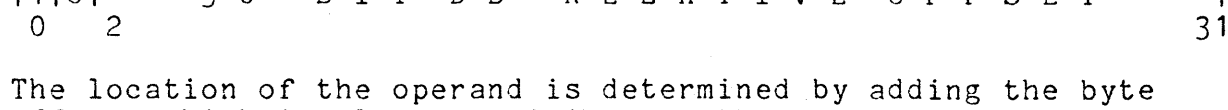
The location of the operand is the location of the pointer plus the offset contained in the pointer; hence the name of self-relative pointer. The offset is a 32-bit 2's complement byte offset. Self-relative pointers are used exclusively in connection with code segments and can be distinguished from machine instructions only by their usage.

Stack segment pointers have the following format:



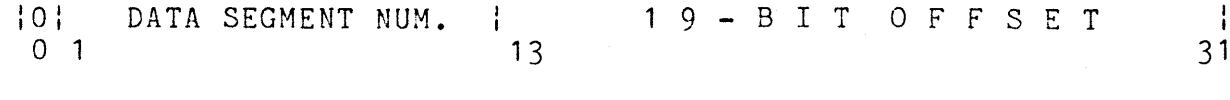
The location of the operand is determined by adding the byte offset, which is always positive, to the contents of the SB register.

Global data segment pointers have the following format:



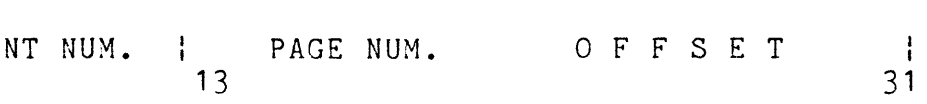
The location of the operand is determined by adding the byte offset, which is always positive, to the contents of the DB register.

Large arrays of data that do not fit conveniently in stack segments or global data segments can be stored in external data segments and accessed indirectly through external data segment pointers stored in stack or global data segments. External data segments can be either paged or unpaged. In the unpaged case the external data segment pointer is interpreted as follows:



The data segment number points to the appropriate entry in the Data Segment Table. This entry contains the current location and length of the data segment. The location of the operand is determined by adding the 19-bit positive byte offset in the pointer to the address of the first word in the data segment.

In the paged case the external data segment pointer is interpreted as follows:



The data segment number points to the appropriate entry in the Data Segment Table. This entry contains the current location and length of the Page Table. The page number points to the appropriate entry in the Page Table. This entry contains the location of the page. The location of the operand is determined by ORing the offset in the pointer with the address of the first word of the page. The boundary between page number and offset in the pointer is movable [see section 3.2.1]. Note also that paged data segments must not be used for stack segments or global data segments.

In the paged case the external data segment pointer can be thought of as a 31-bit virtual address. This virtual address space is split into two pieces, one for system data and one for user data as described in section 3.2.1. In a multi-user system, each user can have his own 30-bit virtual address space. For indexed addressing [see section 2.2.4] the index register is added to the entire original pointer/virtual address to determine a new pointer/virtual address. See section 6.2 for a more detailed description of the evaluation of external data segment pointers.

2.2.4 Indexed Addressing

Indexed addressing provides a convenient means of accessing the individual elements of an array. An address is determined from the base and offset information in the instruction just as it is in the direct case. In the indexed case, however, the value of the index register is added (as a 32-bit 2's complement byte offset) to this address to determine the location of the operand.

In most cases, indexed addressing and indirect addressing are indicated by two separate bits within an instruction. Thus it is possible to indicate both indirect and indexed addressing for the same instruction. For all instructions except branch instructions postindexing is used. This means the indirect pointer is fetched first [using base and offset] and the index register is added in during the address computation. Branch instructions use preindexing so that high-level language "CASE" statements can be

executed efficiently. In this case, the base, offset and index register select one of a set of indirect pointers which is then evaluated (without using the index register) to determine the branch target.

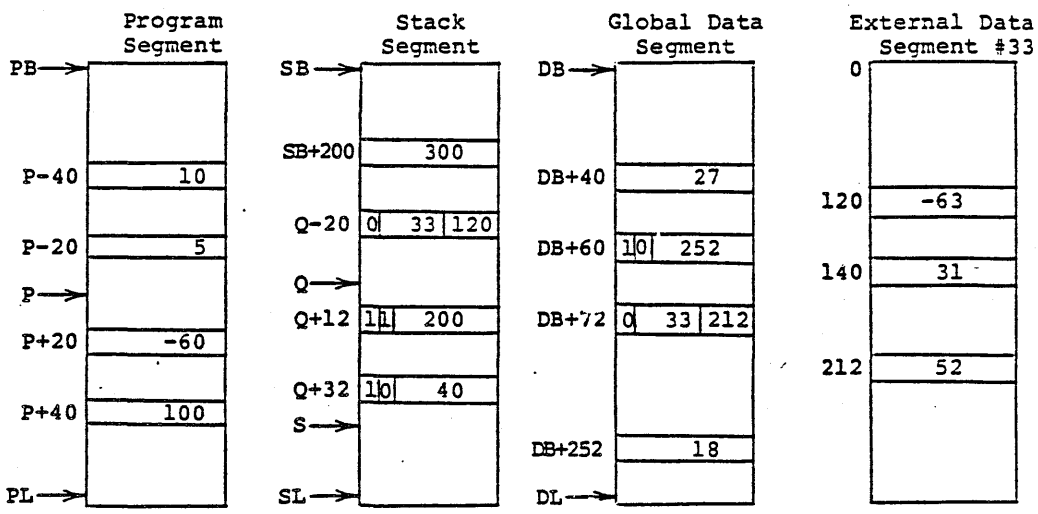
A number of examples of address computation for the load instruction are shown in Figure 2.1. This instruction pushes the word at the computed address onto the stack. All of the offsets (e.g. 20 in P + 20) indicated in the figure are decimal byte offsets. The numbers along the left side of the [unpaged] external data segment are decimal byte offsets within the segment. The index register is interpreted as a byte offset.

2.2.5 Bounds Checking

The microcode detects all attempts to access memory locations outside of the following ranges:

Code Segments	PB to PL
Stack Segments	SB to S
Global Data Segments	DB to DL
Unpaged External Data Segments	Starting Address to Starting Address+Length-1

(For unpaged external data segments, the starting address and



The Index Register contains 20

I stands for Indirect - X stands for Indexed

INSTRUCTION	New TOP-OF-STACK
LOAD P+20	-60
LOAD P+20, I	10
LOAD P+20, X	100
LOAD P+20, I,X	5
LOAD Q+12, I	300
LOAD Q+32, I	27
LOAD Q-20, I	-63
LOAD Q-20, I,X	31
LOAD DB+60, I	18
LOAD DB+72, I	52

Figure 2.1 Examples of Address Computation

length are determined from the Data Segment Table. For chained external data segments, described in Section 3.2.2, the above bounds tests are applied individually to each segment in the chain. See section 6.2 for the bounds checking for paged data segments.) If such an attempt is made, control is passed to the operating system via the internal trap mechanism which is described in Section 5.2. The operating system handles the error.

2.2.6 Alignment of Data in Memory

Memory can be viewed as a large byte array arranged as follows:

Bit Number	0	8	16	24	32	40	48	56	64	72	80	88
Byte Address	0	1	2	3	4	5	6	7	8	9	10	11
Halfword Address	0		2		4		6		8		10	
Word Address	0				4				8			

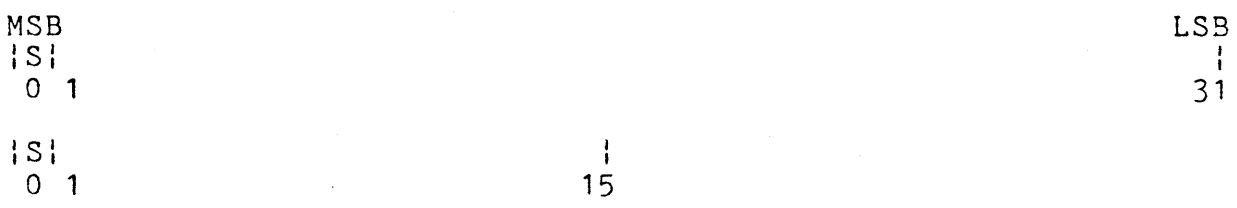
Instructions which operate on word and doubleword operands expect the operands to be aligned on word boundaries (i.e. the two least significant bits of the operand address are ignored and assumed to be zero). Similarly, instructions for halfword operands expect the operands to be aligned on halfword boundaries (i.e. the least significant bit of the operand address is ignored and assumed to be zero). Instructions which operate on bytes can access any byte in memory. Futhermore, all tables known to the Machine Instruction Set and all resident segments and pages must start on a word boundary and must occupy an integer number of words.

2.3 FORMATS

The machine instruction set includes instructions which manipulate integers, floating-point numbers, decimal numbers and byte strings. The formats for these data types are defined in this section along with the format of the stack marker which is used to save the status of the machine on a procedure call.

2.3.1 Integers

The instruction set supports 16-bit and 32-bit two's complement integers which have the following formats:



where S is the sign bit. The instruction set also supports unsigned 32-bit integers.

2.3.2 Floating-Point Numbers

Sign-magnitude representation is used for floating-point numbers. Two formats are supported, a 32-bit format:

and a 64-bit format:

where S is the sign [Plus = 0, Minus = 1] of the binary fraction. The exponent field contains 127 or 1023 plus the actual exponent (power of 2) of the number in the 32-and 64-bit formats respectively. Exponent fields containing all zeros and all ones are "reserved". If the exponent is zero and the fraction is zero then the number is interpreted as a signed zero. If the exponent is zero but the fraction is not zero then the number is assumed to be denormalized. Floating point numbers are usually stored in a normalized form in which the binary point is to the left of the fraction field and an implied leading 1 is to the left of the binary point. If the exponent is all ones and the fraction is zero then the number is regarded as a signed infinity. If the exponent is all ones but the fraction is not zero then the interpretation is "not-a-number" [NAN]. Attempts to operate on denormalized numbers, infinities and NANs will cause a floating point operand trap. [See Appendix F]

The 32-bit format can represent (normalized) numbers with absolute values from about 1.2 X 10^-38 to about 3.4 X 10^38 with

seven significant digits. The 64-bit format can represent [normalized] number with absolute values from about 2.2 X 10^-308 to about 1.8 X 10^308 with almost 16 significant digits.

2.3.3 Decimal Numbers

The instruction set supports conversions to and from the following decimal format:

EXPONENT	0 0 0 0 0	S	D1	D2	D3	D4
0	10	15				31

D5	D6	D7	D8	D9	D10	D11	D12
32							63

D13	D14	D15	D16	D17	0 0 0 0 0 0 0 0 0 0 0 0
64					95

where S is the sign of the decimal number. The exponent field contains a 2's complement integer which is the actual exponent (power of 10) of the number. All decimal numbers are assumed to be in a normalized form in which the decimal point is between D1 and D2 and D1 # 0. The number zero is a special case represented by three words containing all zeros. Each decimal digit is a 4-bit quantity between 0000 and 1001.

The instruction set also supports 8-digit unsigned decimal integers whose format corresponds to the second word of the decimal floating point format.

2.3.4 Byte Strings

A string consists of a word containing the "current length" [C] of a string, followed by a byte array containing the characters of the string. The allocated length of this array is the "maximum length" [M] of the string. The bytes of the string are numbered from 1 to M; only the first C bytes contain valid data. The format is:

CURRENT LENGTH (C)			
BYTE 1	BYTE 2	BYTE 3	BYTE 4
		*	
		*	
		*	
BYTE C-2	BYTE C-1	BYTE C	BYTE C+1
		*	
		*	
		*	
BYTE M-1	BYTE M		

The additional information needed to operate on a string is provided by a "string descriptor". A string descriptor is a 4-word block which must be loaded onto the stack prior to executing a string instruction. The format is:

MAXIMUM LENGTH [M]
POINTER TO CURRENT LENGTH WORD
INDEX OF FIRST BYTE OF INTERET [F]
NUMBER OF BYTES OF INTEREST (N) OR -1

The pointer can either be a data segment pointer or a local program pointer. The maximum length entry should be loaded onto the stack first.

In situations where a string value is being temporarily held on the stack, it is represented by a structure called a "string temporary". The string temp is designed to look like a string plus descriptor; however, an additional "temp size word" is inserted. A string temp is formed by:

1] Pushing a string onto the stack, starting with the current length entry.
2] Pushing the "temp size word" which is equal to the number of words occupied by the string.
3] Pushing a string descriptor for the string.

The "temp size word" is used to make it easy to delete the string temp from the stack with the SUBS instruction.

2.3.5 The Stack Marker

When necessary, the status of the machine is saved in a stack marker in the currently active stack segment. The format of the

stack marker is as follows:

```
                INDEX REGISTER
  PB-RELATIVE RETURN ADDRESS
              STATUS REGISTER
                   DELTA Q
```

When a stack marker is pushed onto the stack, the Q and S registers are updated to point to the right-most byte of the new delta Q entry. Delta Q is the offset back to the old value of Q. The PB-relative return address is simply the address of the first instruction to be executed when the machine status is restored.

3. SEGMENTATION

The segmentation scheme employed in the machine instruction set provides two main capabilities. First, it provides a way to break up a large program or data base into logical pieces which need not all be present in memory all the time. Second, it enables a CPU to handle a number of different programs at the same time so that the best use can be made of all of the system resources. In both cases, all of the decisions related to the handling of system resources are left to the operating system.

Two basic types of segments are utilized, program or code segments and data segments. An overview of these types of segments is presented in Section 1.1. A detailed description is presented in this section.

3.1 CODE SEGMENTS

The machine instructions for each executable program are kept in one or more code or program segments. A portion of each segment, called the Segment Transfer Table, contains the information required to transfer control from one procedure to another procedure, possibly in a different code segment. The entire collection of code segments is managed through a Code Segment Table.

3.1.1 The Code Segment Table

The Code Segment Table (CST) consists of a maximum of 4096 4-word segment descriptor entries. The actual number of entries in use at any point in time is variable. Entries are dynamically allocated by the operating system as programs are loaded and unloaded. Each entry contains control information about the segment, and gives its length and location. Accessing the CST is done mainly via the PCL, EXIT, IXIT and DISP instructions (see Section 6.2.4 for a description of these instructions) and is completely transparent to the user.

The overall structure of the CST is shown in Figure 3.1. The first 1024 entries are reserved for strictly operating system code (i.e. operating system code which does not modify user data). Entry zero is different from all of the others. The first word of this entry contains 16 times the current number of entries

(excluding entry zero) in this portion of the CST. The other three words in entry zero can be used by the operating system. The absolute memory address of the first word in entry zero is kept in a dedicated memory location (see Appendix D). The next 2048 entries are reserved for sharable code (i.e. code which is part of the operating system but which may modify user data). The final 1024 entries are reserved for strictly user code (i.e. code which is supplied entirely by a user).

Each entry in the CST has the following format:

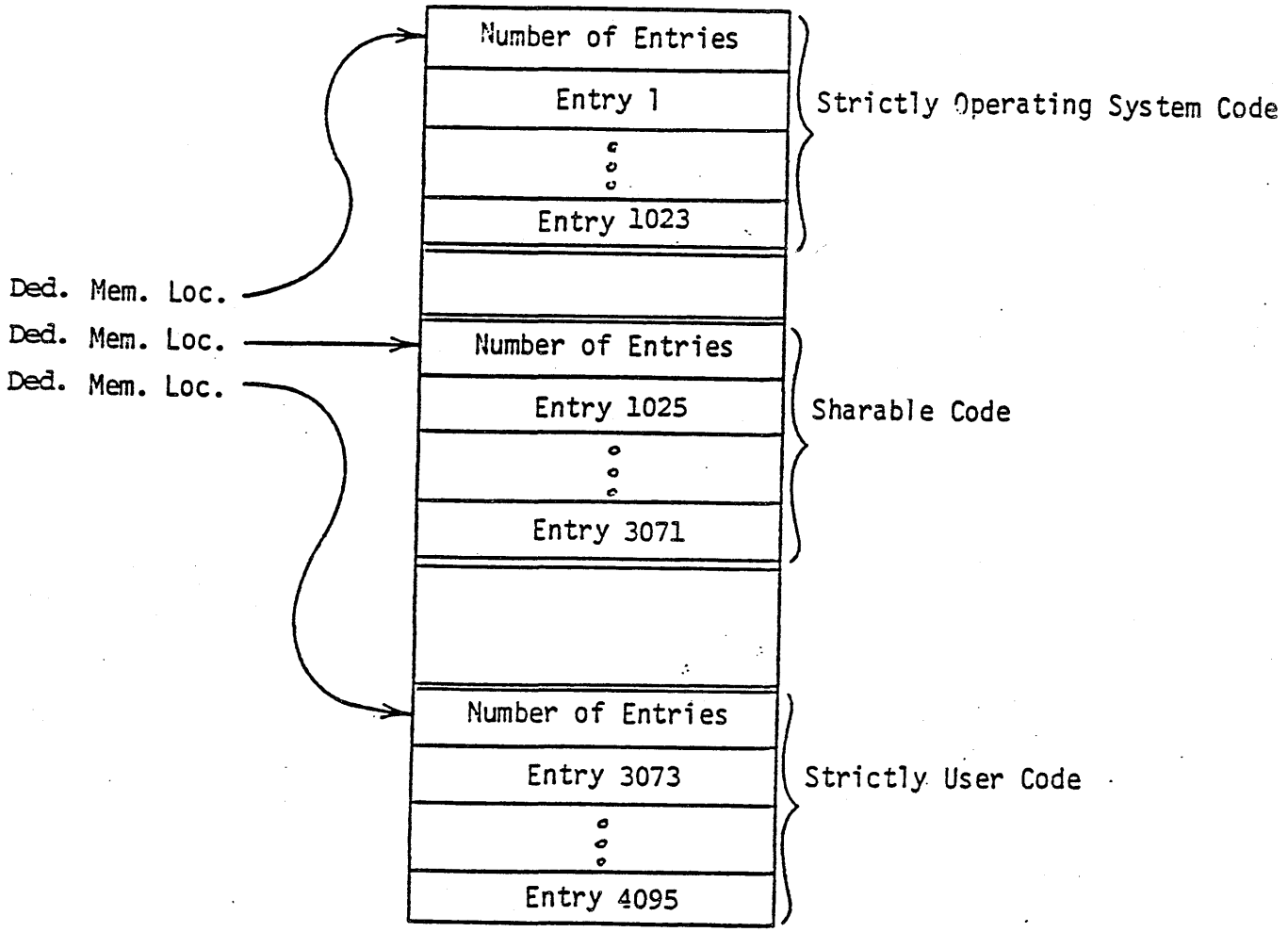


Figure 3.1 The Code Segment Table

0	USE COUNT		R	M	A
		LENGTH - 1			
		LOCATION			

0 1 13 29 31

where:

A = Absence bit - Set to 1 if the code segment is absent from main memory.

M = Mode bit - set to 1 if the segment is to be executed in privileged mode.

R = Reference bit - set to 1 when the segment is accessed

Length = The length of the code segment in bytes. Bits 30 and 31 of Length-1 must both be 1.

Location = The location of the code segment. If A=0 the Location is the absolute memory address of the first word in the segment and bits 30 and 31 must both be 0. If A=1, this entry can be used by the operating system.

Use Count = The number of tasks currently using this segment. This count is incremented when code execution begins in a particular segment and is decremented when control is transferred to another segment. An internal trap is generated if the use count is zero when it is to be decremented or 4095 when it is to be incremented.

Bits 13 - 17 of the first word of each entry and all of the fourth word can be used by the operating system. Bits 18 - 28 of the first word are reserved.

When an instruction needs a CST entry, it reads the first word in the entry and simultaneously sets this word to (2's complement) -1. When an instruction is finished using a CST entry, it restores the first word in the entry to the appropriate value. This temporary -1 condition indicates that a CST entry is currently "owned" by someone. If an instruction reads the first word of an entry and gets a negative number, it will repeatedly read this word until it gets a positive number. The operating system should treat CST entries in a similar manner.

3.1.2 Segment Transfer Table

The words at the end of each program segment are used for the Segment Transfer Table (STT). It is convenient to think of this table as starting at the PL end of the segment and growing toward the PB end of the segment. The entry pointed to by the PL register has the format:

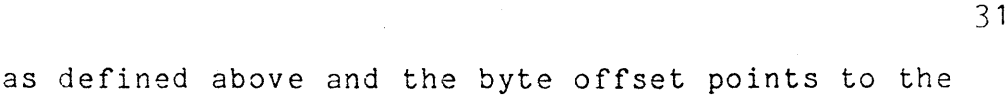
where:

U = Uncallable bit - If the U-bit is 1 and the calling procedure is in unprivileged mode an internal trap (see Section 5.2) is generated.

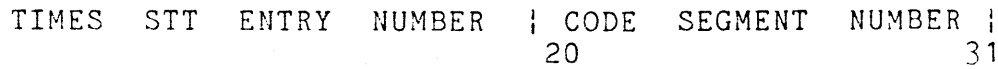
Length = Four times the number of entries in the STT.

Bits 2-12 must all be zero.

All of the other entries in the STT are either local program pointers or external program pointers. There is a local program pointer for each procedure in a code segment. The format of these pointers is:



where U is as defined above and the byte offset points to the start of the procedure. Procedures in other segments are accessed through external program pointers, each of which points either to a local program pointer or to the length entry in the STT of some other segment. The format is:



where the STT entry in the specified code segment either contains a local program pointer which points to the start of the procedure or is the length entry, in which case P is set to PB. The Mode bit in each CST entry specifies the mode for all of the procedures in a segment. The Uncallable bit in STT entries provides for callable and uncallable privileged procedures. All unprivileged procedures are considered to be callable. The instruction set does not support uncallable unprivileged procedures.

An example of the use of the CST and STT by the procedure call (PCL) instruction is shown in Figure 3.2. The PCL instruction is described in Section 6.2.4 In the example, two

situations are described, namely, a procedure call within a seqment and a procedure call to another seqment. First consider the PCLS 2 instruction in segment 23. The second entry in the STT is accessed. This entry contains a local program pointer (with the uncallable bit equal to 0) which points to the location of the next instruction (PB+400). The information in the CST is not needed for this procedure call. Next consider the PCLS 3 instruction in segment 23. The third entry in the STT contains an external program pointer which points to the fourth STT entry in program segment 22. The CST is used to find the location of segment 22 and to update PB and PL. The fourth entry in the STT contains a local program pointer which points to the location of the next instruction (PB + 100). In this case the use count for segment 23 is decremented and the use count for segment 22 is incremented.

3.2 DATA SEGMENTS

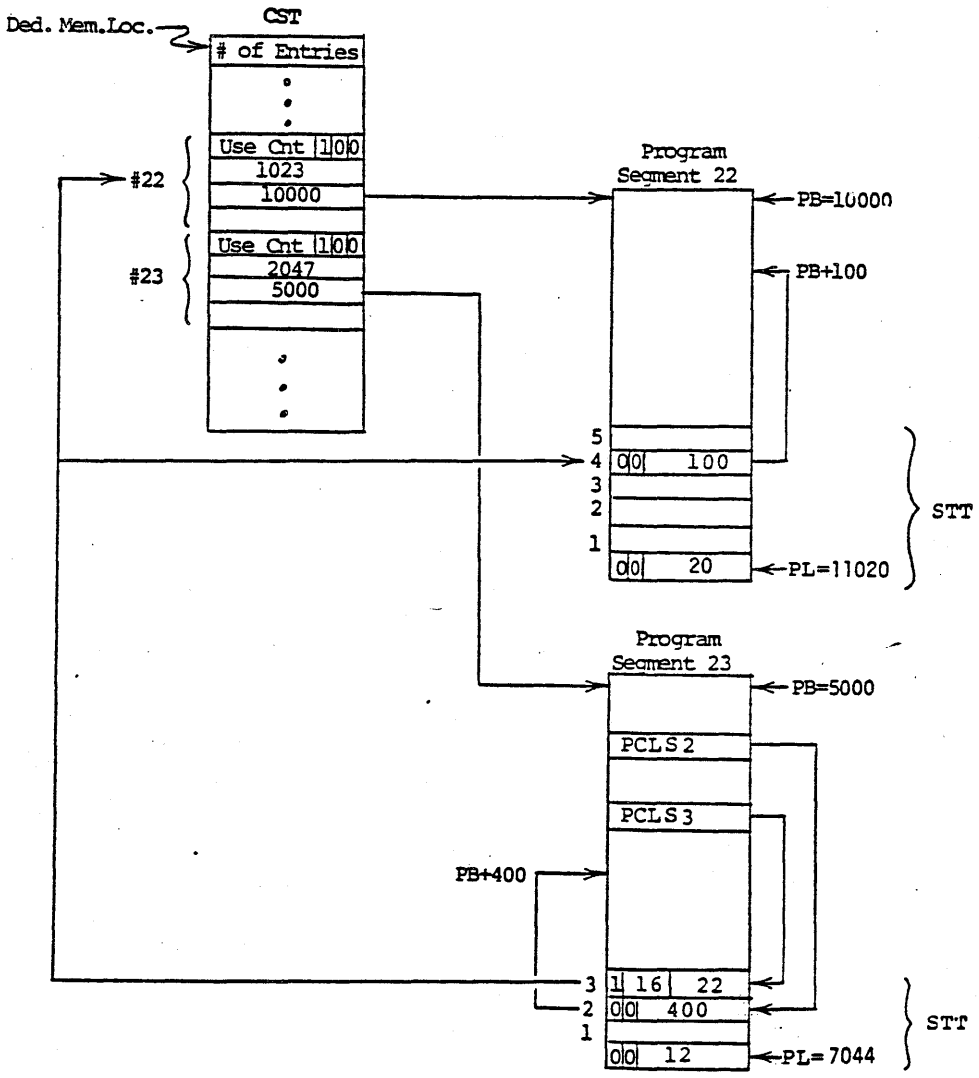


Figure 3.2 Use of CST and STT

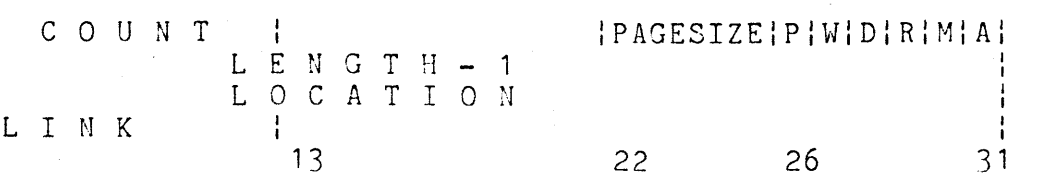
Every program requires at least two data segments: a stack segment and a global data segment. Data such as large arrays and I/O information that cannot be stored conveniently in these two segments can be stored in external data segments. All of these segments are managed through a Data Segment Table.

3.2.1 The Data Segment Table

The Data Segment Table (DST) consists of a maximum of 4096 4-word segment descriptor entries. As with code segments, the actual number of entries in use at any point in time is variable. Entries are dynamically allocated by the operating system as jobs are initiated and terminated. Each entry contains control information about the segment, and gives its length and location.

The overall structure of the DST is shown in Figure 3.3. The first 2048 entries are reserved for operating system data. Entry zero is different from all the others. The first word of this entry contains 16 times the current number of entries (excluding entry zero) in this portion of the DST. The other three words in entry zero can be used by the operating system. Entry zero is assumed to start at memory location 2200 octal. The remaining 2048 entries are reserved for user data. The location of this portion of the current user's DST is kept in the DST register.

Each entry in the DST has the following format:



where:

- A = Absence bit - set to 1 if the data segment or page table is absent from main memory.
- M = Mode bit - set to 1 if access to this [unpaged] segment is allowed only in privileged mode. Ignored for paged segments.
- R = Reference bit - set to 1 when data in the segment or the page table is accessed.
- D = Dirty bit - set to 1 when the [unpaged] data segment is altered. Ignored for paged segments.
- W = Write bit - set to 1 if the [unpaged] data segment can be altered. Ignored for paged segments.
- P = Paged bit - set to 1 if the data segment is paged.

Pagesize = The number of bits in an external data segment pointer which are to be interpreted as offset within a page. The values 0 and 1 must not be used. The size of a page is 2^Pagesize bytes. Ignored for unpaged segments.

Length = The length of the data segment or page table in bytes. Bits 30 and 31 of Length-1 must both be 1.

Location = The location of the data segment or page table. If A=0 the Location is the absolute memory address of

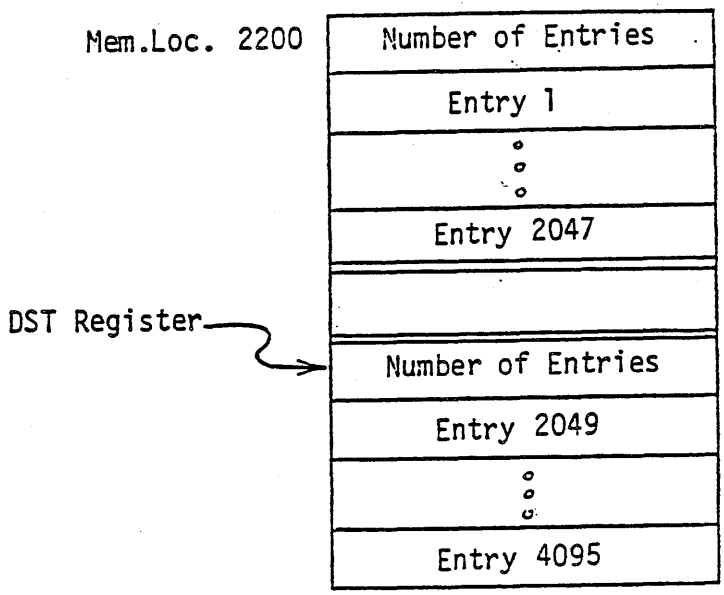


Figure 3.3 The Data Segment Table

the first word in the segment or page table. Bits 30 and 31 must both be 0 for unpaged segments. Page tables must be aligned on the smaller of: page boundaries or page table boundaries. If A=1, this entry can be used by the operating system.

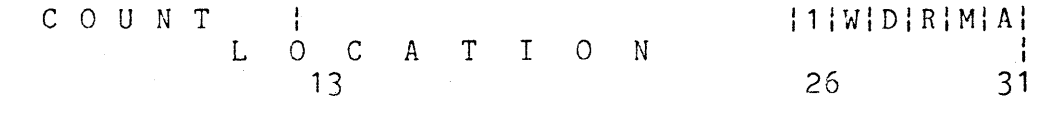
C = Consecutive Link bit. Set to 0 for non-consecutively linked segments. Set to 1 for consecutively linked segments. Ignored for paged segments.

Link = Chained data segment link - set to 0 if a data segment is not linked to any other data segment. If the data segment is linked to one or more other data segments then the link field is not zero. Linked data segments are explained in Section 3.2.2. Ignored for paged segments.

Use Count = The number of tasks currently using this segment. This count is incremented by an instruction upon gaining access to a particular external data segment and is decremented by the instruction when access is no longer needed. (The XGDS instruction also manipulates this count.) An internal trap is generated if the use count is zero when it is to be decremented or 4095 when it is to be incremented.

Bits 13-17 of the first word and bits 13-31 of the fourth word of each entry can be used by the operating system. Bits 18-21 of the first word are reserved.

Each entry in a page table has the following format:



where:

A = Absence bit - set to 1 if the page is absent from main memory.

M = Mode bit - set to 1 if access to this page is allowed only in privileged mode.

R = Reference bit - set to 1 when data in the page is accessed.

D = Dirty bit - set to 1 when the page is altered.

W = Write bit - set to 1 if the page can be altered.

Location = The location of the page. If A=0 the Location is the absolute memory address of the first word in the

page. Bits 32-Pagesize thru 31 must be zero. If A=1, this entry can be used by the operating system.
Use Count = The number of tasks currently using this page.

Bits 13-17 of the first word of each entry can be used by the operating system. Bits 18-25 of the first word are reserved. Note that bit 26 must be a 1.

When an instruction needs a DST entry, it reads the first word of the entry and simultaneously sets this word to -1. If this word is negative, the instruction will repeatedly read this word until it gets a positive number. If the instruction also needs a page table entry, it will perform this same operation on the first word of the page table entry and then write back the first word of the DST entry. When the instruction is finished with the final table entry, it restores the first word of the entry to the appropriate value. The operating system must treat DST and page table entries in a similar manner.

3.2.2 Data Segment Chaining

It may be convenient, from the point of view of memory management, to break up a large array of data into several smaller unpaged external data segments. Two situations exist. First, the size of the array is known and is static. Second, the size of the array is dynamic and is not known beforehand. These two cases are treated separately.

Static arrays can easily be broken into smaller segments of equal size and these segments can be assigned to consecutive DST entries. In this case, the link field in each DST entry (except for the last DST entry in the chain) contains the entry number for the last segment in the chain and the consecutive link bit is set to 1. The last DST entry in the chain has a link field of 0. If the desired offset into the array is known, it is easy to determine the appropriate segment. Consecutively linked segments must all be the same length (within a given chain) and this length must be a power of 2.

The above scheme does not work well for dynamic arrays because it may not be possible to allocate consecutive DST entries as the array grows. In this case the consecutive link bit is set to 0 and the link field in each DST entry contains the entry number for the next segment in the chain. The last DST entry in the chain has a link field of zero. Using this scheme, any

available DST entry number may be allocated for each new data segment. There are no length restrictions for non-consecutively linked segments.

When setting up the information in DST entries for a set of chained segments the following rules must be observed: 1] All of the segments in a particular chain must be in the same part of the data segment table [i.e. either all in the system DST or all in the user DST], and 2] All of the segments in a particular chain must be within the current length of the DST. Violations of these two rules are not detected by the microcode.

4. INPUT AND OUTPUT

Input and output operations are initiated by machine instructions and are carried out either by DMA or by direct I/O. Both 8-bit and 16-bit I/O can be performed. The handling of interrupts, which are also provided, is described in Section 5. This section briefly describes the available DMA and direct I/O capabilities. The I/O ERS should be consulted for additional details.

Direct I/O can be used for 16-bit transfers. For output, the device address is pushed onto the stack followed by the data word in which the desired halfword is right justified. Executing the IOW instruction accomplishes the output and pops the data word from the stack. For input only the device address is loaded onto the stack. The IOR instruction pushes the data (also right justified) read from the I/O channel onto the stack.

Both 8-bit and 16-bit I/O can be done through DMA. The desired starting address, transfer count, and (optional) termination byte/halfword are loaded onto the stack and transferred to the I/O hardware via separate IOW instructions. The direction of the transfer and the size of each transfer are controlled by bits in the DMA Status word which is also sent to the I/O hardware by the IOW instruction. Once a DMA operation is started, it proceeds independently of the CPU which can be interrupted at the completion of the operation. Provisions for automatically double buffering DMA transfers are also provided in the I/O hardware. The current status of a DMA transfer can be tested by reading the current address, current transfer count and current status from the I/O hardware.

5. INTERRUPTS AND TRAPS

There are three mechanisms which have the ability to modify the normal flow of machine instructions: external interrupts, internal interrupts, and traps. External interrupts are generally service requests from I/O devices. These requests are usually acknowledged at the end of a machine instruction. Internal interrupts generally signal some abnormal condition within the system which is not associated with the execution of an instruction. As with external interrupts, internal interrupts are not usualy acknowledged until the end of an instruction. Traps differ from interrupts in that traps result from conditions detected by the microcode during the execution of an instruction. The detailed handling of all three conditions is left to the operating system.

5.1 EXTERNAL INTERRUPTS

The section of microcode which does the initial processing of all external interrupts is called the interrupt handler and is described in this section (a flow chart is provided in Appendix C). Two other important aspects of external interrupts are also discussed, namely the external interrupt structure and the interrupt response time.

5.1.1 External Interrupt Structure

External interrupts are serviced in descending order of priority, i.e. the highest priority interrupt is serviced first. There are 16 hardware-defined priority levels and each I/O device can be assigned any one of these levels. The interrupt structure is such that a higher priority device can preempt a lower priority device. Within a given priority level, interrupts are handled on a first-come, first-served basis. A Mask register is provided for the purpose of masking off different priority levels. The I/O ERS should be consulted for additional details.

5.1.2 Interrupt Response Time

The interrupt response time is defined to be the time that elapses between an interrupt and the start of the execution of the first instruction of the interrupt service routine for the highest priority unmasked interrupt. The CPU looks for unmasked Message

register interrupts [which include all I/O interrupts] at the end of every instruction and at least every 12.5 microseconds in long instructions. At the end of an instruction up to three other conditions may have to be handled by the CPU before handling an I/O interrupt [see section 5.2.1]. In interruptable instructions, I/O interrupts are handled after checking for other Message register interrupts as in the case of interrupts at the end of an instruction. In either case the occurance of stack overflow will add to the interrupt response time. If an I/O interrupt is the only condition that needs to be handled then the interrupt response time is expected to be about 25 microseconds. There are two uninterruptable activities for the CPU. The first is the evaluation of an external data segment pointer into a set of nonconsecutively linked chained data segments (see section 3.2.2). The second is gaining access to a CST or DST entry by "semaphore reading" the first word of the entry (see sections 3.1.1 and 3.2.1)

5.1.3 The Interrupt Handler

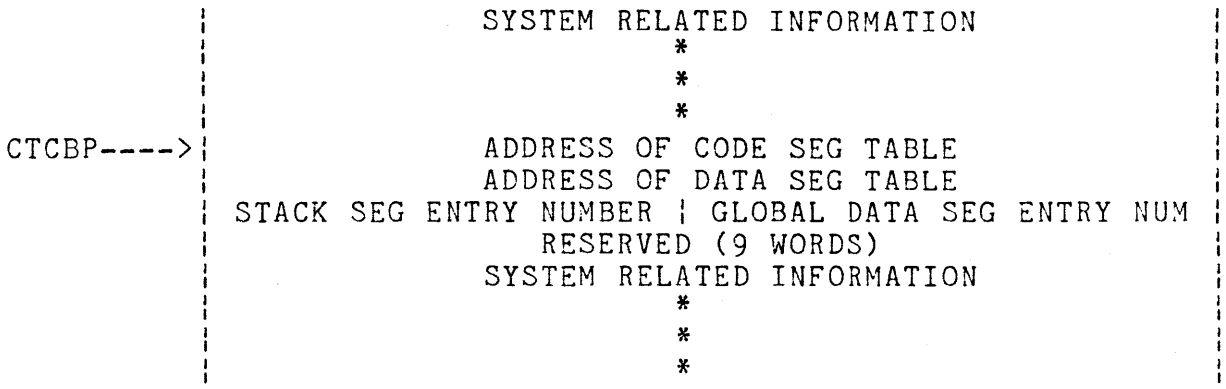
The microcode interrupt handler [see Appendix C] is executed on behalf of a particular device when the following conditions are met:

1) The device has requested an interrupt
2) Interrupts at that priority level are not masked
3) The interrupt bit in the Status register is set to 1
4) No higher priority device is requesting service

Execution of the interrupt handler is begun at the end of the currently executing machine instruction (except for a few instructions like the move/scan instructions which can be interrupted before completion). The first job of the interrupt handler is to determine which device should be serviced. If, in a multi-CPU system, a CPU finds that all interrupting devices have already started receiving service then that CPU simply returns to what it was doing before the interrupt. Otherwise, a device is removed from the list of devices waiting at the highest unmasked priority level. The state of the machine is then stored by pushing a stack marker onto the current stack. Because all external interrupts are handled on a separate execution stack called the Interrupt Control Stack (ICS), i.e. the CPU registers are changed to point to different stack and data segments, it is also necessary to save the information required to restore the current stack and data segments. This information is stored in

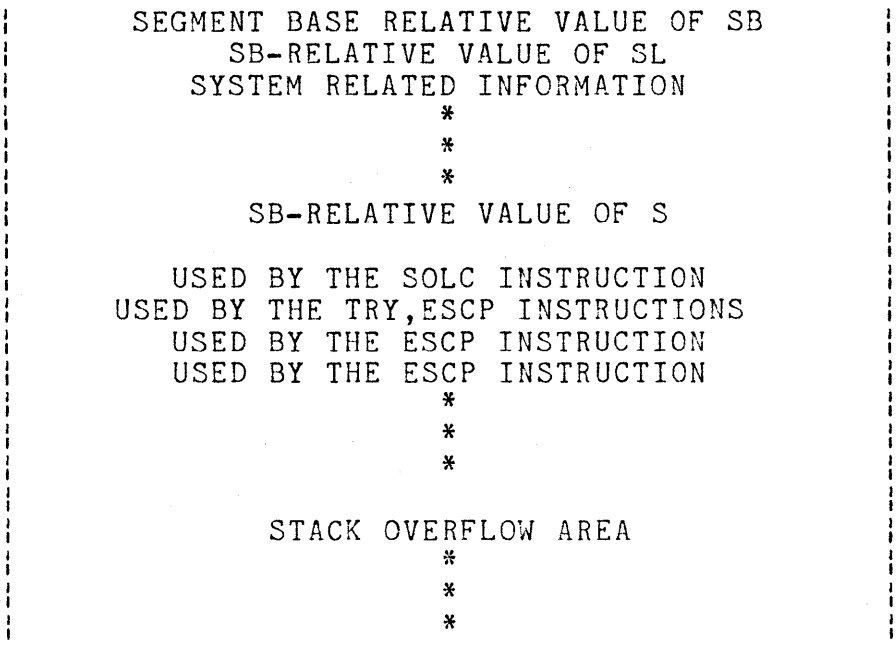
two places. Some information is kept in a Task Control Block (TCB) associated with the interrupted task and some information is kept with the stack segment.

The format of the TCB is as follows:



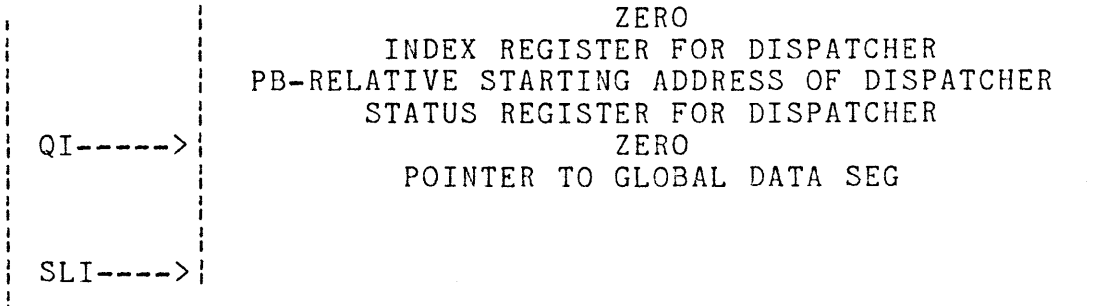
where CTCBP is the Current TCB Pointer which is kept in a dedicated memory location (see Appendix D). The first two words point to the start of the appropriate Code and Data Segment Tables and the next word contains the Data Segment Table entry numbers for the stack segment and the global data segment. The next nine words are reserved for other implementations of this instruction set. One will contain an external data segment pointer whose data segment number points to the first paged data segment in the DST for this task. The operating system can use the remainder of each block for system related information.

The information required to restore the values of the pointers into the stack segment is kept in the first few words of the stack segment. The format for a stack is:



It is important to note that the SB-relative value of S is stored at SB-1. This is done because the value of SB is in the corresponding CPU register, thus allowing the value of S to be stored quickly on an interrupt. The relative value of Q does not have to be saved because it is contained in the stack marker which is pointed to by S. Each stack segment must also have a stack overflow area of at least 64 words between SL and the physical end of the stack segment. This area insures that instructions can run to completion even after causing S to become larger than SL and that all pending traps and interrupts can be processed. The words SB+1 through SB+4 are used by the indicated instructions.

The Interrupt Control Stack is initialized by the operating system to have the following stack-marker-like format:

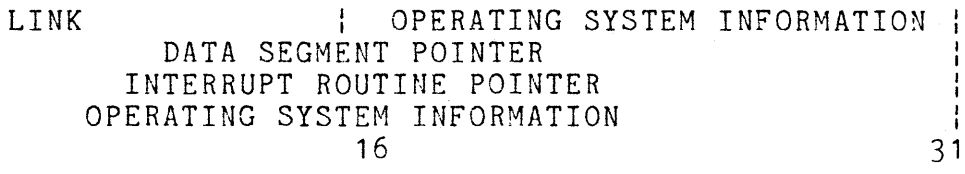


where the PB-relative starting address points to the first executable instruction in the dispatcher (opertating system supplied software for initiating programs). If the code segment number in the Status register entry is not 0 then PB and PL are evaluated through the code segment table. If the code segment number is 0 then PB is set to 0 and PL is set to 3777777774. In either case the dispatcher must be resident. The pointer to the global data segment is a data segment pointer to the global data segment used by the dispatcher. If an external data segment pointer is used, the offset field in the pointer is ignored and the values of DB and DL are obtained from the location and length information in the DST. If a stack or global data segment pointer is used, the offset field is interpreted as the absolute address for DB. DL is set to DB + 1024 (words) for stack pointers or to 3777777777 for global pointers. The pointers QI and SLI are the interrupt values of Q and SL respectively. These two pointers are kept in dedicated memory locations (see Appendix D). The use of the ICS allows an interrupt service routine to "swap out" the stack segment for the interrupted program, if necessary. In addition, the ICS provides a controlled environment for interrupt service routines.

When execution is transferred to the ICS, the interrupt handler sets the hardware ICS flag, which is accessible only to the microcode, and then sets Q to QI, SL to SLI, and SB to QI-64 (words). The location pointed to by SB is used to hold the data segment pointer to the global data segment for the currently executing interrupt service routine. The location SB-1 is used by some instructions (e.g. STOP) to save the SB-relative value of S, and SB+1 through SB+4 are used by other instructions (e.g. SOLC, TRY, and ESCP). The current value of the interrupt mask is pushed onto the stack at Q+2 and the current and all lower priority levels are masked. The device number of the device requiring service is pushed onto the stack at Q+3. The S register is set

equal to Q + 3. All of the locations between SB+1 and QI-5, inclusive, can be used by the operating system.

The device number points into a Device Reference Table which consists of a series of four-word entries of the following format for each device.



The first halfword contains either zero or the address of the DRT entry for the next I/O device waiting for service at this priority level. The operating system must initialize the links to zero. The second word contains a data segment pointer which points to the global data segment [which must be resident] used by the interrupt service routine. If an external data segment pointer is used, the offset field in the pointer is ignored and the values of DB and DL are obtained from the location and length information in the DST. If a stack or global data segment pointer is used, the offset field is interpreted as the absolute address for DB. DL is set to DB+1024 (words) for stack pointers or to 3777777777 for global pointers. The third word contains a program pointer to the interrupt service routine [which must be resident and should not be in code segment 1] for the device. If an external program pointer is used, it is evaluated through the CST as in the case of a PCL instruction. If a local program pointer is used, the uncallable bit is ignored and the offset field is interpreted as the absolute address for P. PB is set to 0, PL is set to 3777777774, and the segment number in the Status register is set to 0. The second halfword and the fourth word are reserved for use by the I/O hardware or the operating system. The interrupt handler decrements the use count for the current code segment, and increments the use count for the new segment. For the special case of code segment 0 the use count is neither incremented nor decremented. The DRT occupies a block of dedicated memory locations. (See Appendix D.)

If an interrupt service routine is interrupted by a higher priority request, a standard stack marker is pushed onto the ICS followed by the data segment pointer to the global data segment of the interrupted routine followed by the current mask word followed by the device number of the higher priority device. The Device

Reference Table entry for the new interrupt determines the code sequence that is executed next. The external data segment pointer for the new global data segment is stored at (SB).

All interrupt service routines end with an interrupt exit (IXIT) instruction. When all interrupts have been serviced, i.e. when Q points to an entry with bits 1-31 all zero (bit 0 is used separately by the DISP instruction), the IXIT instruction determines what is to be done next. Several options are available including returning to the interrupted process, or entering the dispatcher (either at the beginning or where it was interrupted). A flowchart of the IXIT instruction is provided in Appendix C to provide more complete information on its operation.

5.2 INTERNAL INTERRUPTS AND TRAPS

Code segment 1 is reserved for procedures which service internal interrupts and traps. Most of these procedures use the current user's stack while the rest use the Interrupt Control Stack as described in the preceding section. Because of the special role played by code segment 1, it is assumed to be resident and the usual trap conditions are not tested when entering code segment 1 to handle other trap conditions.

5.2.1 Interrupt/Trap Sequence

The following sequence of events occurs when an internal interrupt or trap is detected:

1) The interrupt condition is cleared and an external program pointer is created with the segment number set to 1 and with the Segment Transfer Table (STT) entry number corresponding to the condition which caused the interrupt or trap. These conditions are enumerated in the next section.
2) A stack marker is pushed onto the current stack.
3) The use count for the exited code segment is decremented, and the use count for code segment 1 is incremented.
4) A parameter is pushed onto the current stack at Q+3. If none is required by the interrupt, the parameter is the external program pointer.
5) The PB and PL registers are set up from CST entry 1.
6) All Status register bits are cleared except for the (privileged) mode bit and the code segment number field.
7) The program counter is set from the appropriate STT entry

in code segment 1.
8) Execution of code is begun.

If, at the end of an instruction, more than one trap condition is true then steps 1,2,4, and 7 are repeated and code execution is not begun until the last condition has been handled. The following traps and interrupts can occur at the end of a machine instruction:

1] Trace Variables - tested for first so handled last if other conditions are true.
2] User Traps - tested second.
3] Message Trap - tested third.
4] I/O Interrupts - tested fourth.
5] Instruction Sequencing Bounds Violation - P>PL is tested but only if conditions 1-4 are not true.
6] Stack Overflow - only if conditions 1-5 are not true.
7] Breakpoint Trap - only if conditions 1-6 are not true.
8] Machine Instruction Trap - only if conditions 1-7 are not true.

All tests that are skipped because other conditions are true will be performed again later when the interrupted code sequence is resumed. Stack overflow is also tested just prior to executing the first instruction of the trap or interrupt service routine. If overflow is detected then a stack marker is pushed for the trap or interrupt and the stack overflow trap routine is executed instead. In addition, stack overflow is tested when leaving a stack to switch to the ICS. If overflow is detected in this case then a stack marker for the stack overflow trap is pushed at the base of the ICS and execution of the routine for the condition which caused the switch to the ICS is continued.

5.2.2 Causes of Internal Interrupts and Traps

This section enumerates and defines all of the causes of internal interrupts and traps, the segment transfer table entry number assigned to each, the different passed parameters, and the stack on which each service routine is executed. Unless otherwise specified, there is no special parameter and the service routine executes on the current user's stack. Notes 1 thru 4 appear at the end of this section.

STT# DEFINITION

1) Bounds Violation. An address is outside of the limits for the program, stack or global data segments. See Section

2.2.5. STT#13 is used for external data segments. See note 2.

2) Check Trap. This trap is generated by the CHEK and CRB instructions. The Current Instruction register is passed as a parameter. See notes 1 and 3.

3) Breakpoint Trap. This trap is generated just prior to executing the machine instruction whose absolute address is in the Breakpoint register. See notes 1 and 3.

4) Machine Instruction Trap. This trap occurs prior to every machine instruction if the Machine Instruction Trap Bit in the Status register is set. The Current Instruction register is passed as a parameter. See notes 1, 3 and 4.

5) String Trap. This trap occurs if one of the following errors is detected by a string instruction. The error number is passed in the parameter. See notes 2 and 3.

   1] The maximum length in a string descriptor is negative.
   2] The number of bytes of interest in a string descriptor is unexpectedly negative.
   3] The index of the first byte of interest in a string descriptor is less than or equal to zero.
   4] The specified substring would extend beyond the maximum length in the string descriptor.
   5] The current length of a string is negative.
   6] The current length of a string is greater than the maximum length in the string descriptor.
   7] The index of the first byte of interest in a string descriptor does not point within or immediately adjacent to the current string.
   8] Adding a quantity to a source or target pointer caused the two most significant bits of the pointer to change. In the case of EDS pointers into unpaged segments this trap occurs if any of the thirteen most significant bits changed.
   9] The specified source substring would extend beyond the maximum length of the target if the assignment was made.

   Bits in the parameter word designate the trap condition as follows:

Bits 0-10	0
Bit 11	1 for error in target string 0 for error in source string
Bits 12-15	Error Number
Bits 16-31	Current Instruction Register

6) Unused.

7) Unused.
10) Reset. The hardware reset signal causes this trap. The current instruction is aborted and control is transfered immediately to this service routine which executes on the ICS. Error checking is skipped and use counts are ignored. See note 2. SEE APPENDIX G.
11) Page Table Violation. The page table entry referenced is beyond the current length of the page table. See note 2.
12) Inconsistent Registers. This trap occurs in IXIT and SETR if an attempt is made to setup an inconsistent set of registers (e.g. SL<SB, S<Q, etc.). The Current Instruction register is passed as a parameter. See note 2.
13) External Data Segment Bounds Violation. An address is outside of the limits of an external data segment or a set of linked external data segments. See notes 2 and 4.
14) System Error. An error condition exists under which processing cannot continue. See note 2. The following errors are indicated by the parameter:
    0) Too many PSEB instructions executed
    1) Attempt to decrement a use count which is zero or to increment a use count which is 4095
    2) Bits 1 or 2 in the Message register are set and disabled [either via Mask or the I bit in Status] during an IOR, IOW, IOC, WTC, RFC, or SMSG instruction. This trap does not clear the Message bits.
    3) Unable to complete an IOR within a reasonable amount of time. See note 4.
    4] Unused.
    5-7] Bits 1 or 2 in the Message register became set doing the MPB operaton [including a reasonable number of retries] required by an IOR, IOW, IOC, or SMSG instruction. This trap clears these Message bits and passes their prior values in the two LSB's of the parameter. The slave data error bit is in the LSB.
15) External Data Segment Pointer Violation. An external data segment pointer contains a data segment number of either zero or 2048. See note 2.
16) Pointer Conversion Violation. An attempt was made to form a data segment pointer with an offset which is to large for the type of pointer being used. See note 2.
17) External Program Pointer Violation. An external program pointer contains a code segment number of zero, 1024, or 3072. Or an attempt to EXIT to code segment 1024 or 3072 or

to code segment 0 in unprivileged mode is detected. See note 2.

20) Unimplemented Instruction. An attempt has been made to execute an undefined instruction. The Current Instruction register is passed as a parameter. See notes 1 and 3. SEE APPENDIX G.

21) STT Violation. Two conditions, distinguished by the parameter, can force this trap (See note 2):
    1) An STT entry number is greater than the STT length in the referenced segment.
    2) The STT number in an external program pointer points to an external program pointer in the referenced segment.

22) CST Violation. The CST entry referenced is beyond the current length of the CST. See note 2.

23) DST Violation. The DST entry referenced is beyond the current length of the DST. See note 2.

24) Stack Overflow. An operation caused S to become larger than SL. The service routine is executed on the ICS. It is, in general, very difficult to detemine exactly which instruction caused S to become larger than SL. Nevertheless, correct execution of the offending program can be resumed by increasing the size of its stack and then executing an IXIT instruction. If this trap occurs while on the ICS, SL is increased by 64 words. ADDS, PSHN, SLD, and SCON clear STAT [9:1] before trapping.

25) Stack Underflow. An operation causes S to become less than Q. See note 2.

26) Privileged Mode Violation. Three conditions, distinguished by the parameter, can force this trap (See note 2):
    1) An attempt is made to execute a privileged instruction while in unprivileged mode.
    2) An attempt is made to EXIT or Branch from an unprivileged procedure to a privileged procedure.
    3) An attempt is made in an unprivileged EXIT instruction to change the interrupt bit or to set the instruction pending bit in the Status register.

27) Privileged Mode Data Violation. An attempt is made by an unprivileged procedure to reference data in a privileged data segment. See note 2.

30) Unexpected Pointer Type. An instruction has encountered a pointer type which it cannot handle. See note 2.

31) User Traps. These traps are enabled by the trap bit in the Status register. If this bit is 0 when a trap is encountered, the overflow bit in the Status register is set

and no trap is taken. The overflow bit is cleared if the trap is taken. See notes 1 and 3. This trap occurs whenever both the overflow and trap bits are set [e.g. SETR] but the parameter is correct only for the following conditions:

1) Integer Overflow. The result of an integer operation exceeds the largest representable integer.
2) Integer Divide-by-Zero. The divisor in an integer divide is equal to zero.

Bits in the parameter word designate the trap condition as follows:

Bits 0-23	0
Bits 24-27	0 - Overflow
	1 - Unused
	2 - Invalid Argument (divide-by-zero)
	3-15 - Unused
Bits 28-31 (Result Type)	0 - 16-bit integer
	1 - 32-bit integer
	2 - Unused
	3 - Unused
	4 - 64-bit integer
	5-15 - Unused

32) Illegal Decimal Number. A decimal math instruction has been supplied an illegal operand. The Current Instruction register is passed as a parameter. See note 2.

33) Exponent Size Trap. The DTB, BTD, or POT instruction has been supplied an exponent which is too large. The Current Instruction register is passed as a parameter. See note 2.

34) Floating Point Operand Trap. See notes 1 and 3. This trap is caused by the following operations on binary floating point numbers:

1] Attempts to operate on denormalized numbers, infinities or NAN's.
2] Attempts to divide by zero.
3] Attempts to convert a 64-bit number to a 32-bit number which cannot accommodate the exponent.

Bits in the parameter word designate the trap condition as follows:

Bits 0-9	0
Bits 10-12	Analogous to bits 13-15 but for the second operand
Bit 13	1 if TOS number is a NAN
14	1 if TOS number is an infinity

15	1 if TOS number is denormalized
Bits 16-31	Current Instruction Register

Note: Bits 10-15 are zero for divide-by-zero and conversion errors.

35) Floating Point Result Trap. See notes 1 and 3. This trap is caused by binary floating point overflow or underflow or by the BIR instruction. Bits in the parameter word designate the trap condition as follows:

Bits 0-10	0
Bit 11	LSB of result prior to rounding
12	Round bit of result prior to rounding
13	Sticky bit of result prior to rounding
14	1 if overflow
15	1 if underflow
Bits 16-31	Current Instruction Register

Note: Bits 11-15 are zero if caused by BIR.

36) Unexpected External Data Segment Type. A paged external data segment was encountered when an unpaged segment was expected, and vice versa. See note 2.

37) Absent Code segment. The absence bit in the CST entry for the referenced segment is 1. The CST entry number is passed as a parameter. If this trap occurs on the ICS, the parameter is negated. See notes 2, 3, and 4.

40) Absent Page. The absence bit in the page table entry for the referenced page is 1. The EDS pointer [indexed, if appropriate] is passed as a parameter. If this trap occurs on the ICS, the parameter is negated. See notes 2, 3, and 4.

41) Uncallable Procedure. The uncallable bit in a local program pointer (or in the STT length entry if STT#=0) is set to 1 and this pointer is referenced from another segment running in unprivileged mode. See note 2.

42) Absent Data Segment. The absence bit in the DST entry for the referenced segment is 1. The DST entry number is passed as a parameter. If this trap occurs on the ICS, the parameter is negated. See notes 2, 3, and 4.

43) Absent Page Table. The absence bit in the DST entry for the referenced [paged] segment is 1. The EDS pointer [indexed, if appropriate] is passed as a parameter. If this trap occurs on the ICS, the parameter is negated. See notes 2, 3, and 4.

44) Start-of-Line. This trap is generated by the start-of-line instruction. The 8 LSB's of the instruction are passed as a

parameter. See notes 1 and 3.

45) Variable Trace. The trace bit in the status register is set and the instruction just executed has modified some memory location. A data segment pointer to the first modified byte of memory is passed in Q+3. The second parameter, in Q+2, is a 2's complement integer whose absolute value indicates the number of modified bytes [possibly spanning several pages or segments] and whose sign indicates the direction to the end of the modified section. This trap can be caused by ST, STS, STB, STH, STD, SSI, SBSI, SHSI, SDSI, SBIT, SEML, MVB, MVBW, TRAN, FILB, SAS, SLD, SCON, INCM, DECM, INCF, and DECF. See notes 1 and 3.

46) Start-of-Procedure. The overall debug bit, which is tested by the start-of-procedure instruction, is set. The 8 LSB's of the instruction are passed as a parameter. See notes 1 and 3.

47) End-of-Procedure. The end-of-procedure instruction generates this trap. See notes 1 and 3.

50) Start-of-Subroutine. The start-of-subroutine instruction generates this trap. The 8 LSB's of the instruction are passed as a parameter. See notes 1 and 3.

51) End-of-Subroutine. The end-of-subroutine instruction generates this trap. See notes 1 and 3.

52) Code Segment Violation. An attempt to modify a code segment with unprivileged instructions. See note 2.

53) Branch Violation. An attempt to execute a direct DB, DL, SB, Q, or S relative branch instruction. See note 2.

54) Message Trap. Unmasked message bits in the 16 most significant bits of the Message register have been set and the Interrupt bit in Status is equal to 1. The bits in the message register are cleared and a parameter with the corresponding bits set is placed at Q + 3. See notes 1 and 3. This trap is handled on the ICS.

55) Instruction Sequencing Bounds Violation. This trap occurs if the destination of a PCL, EXIT, IXIT, SXIT, branch instruction, trap, or interrupt is out of bounds and when P>PL is detected during normal instruction sequencing. In this case the segment number and delta P in the stack marker are probably in error.

56) Start-of-Line-Check Trap. This trap is generated by the SOLC instruction. The 8 LSB's of the instruction are passed as a parameter. See notes 1 and 3.

57) Data Segment Write Violation. An attempt to modify a data segment or execute an XGDS with a data segment whose DST entry has the write bit set to zero. See note 2.

Note 1: In these cases the trap condition is detected at the end of an instruction and the offset in the stack marker points to the next instruction.

Note 2: In these cases the trap condition is detected during an instruction and the offset in the stack marker points to this instruction.

Note 3: The instruction sequence which caused this trap is guaranteed to be resumable after the cause of the trap is remedied.

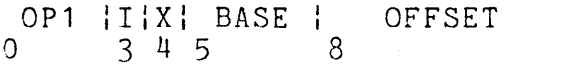
Note 4: This trap clears STAT (9:1) prior to pushing the stack marker.

6. THE MACHINE INSTRUCTION SET

Both 16- and 32-bit instruction formats are used. A total of 254 opcodes are available. The 32-bit instructions are always aligned on full word boundaries in memory. SEE APPENDIX G. The various instruction formats and a detailed description of each instruction are presented in this section.

6.1 INSTRUCTION FORMATS

The basic 16-bit instruction has the format:



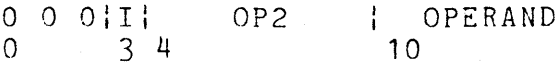
(1)

where OP1 stands for opcode 1, X is the index bit, and I is the indirect bit. The base and offset fields are combined as follows:

BASE	LOCATION
0	P+OFFSET
1	P-OFFSET
2	DB+OFFSET
3	DL-OFFSET
4	Q+OFFSET
5	Q-OFFSET
6	SB+OFFSET
7	S-OFFSET

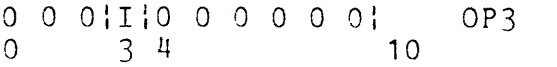
The offset is always a positive quantity in all of the formats.

If opcode 1 is 0, the instruction is interpreted as follows:



(2)

If opcode 2 (OP2) is a 0, the six-bit operand is interpreted as another opcode (OP3). The format is as follows:



(3)

This format is used for stack operations. Many other instructions

require no explicit operand, so OP2=1 also signals the presence of OP3.

```
|0 0 0|I|0 0 0 0 0 1|    OP3    |            (4)
 0     3 4          10         15
```

In formats 2,3,and 4, bit 3 is used as an indirect bit, an index bit or as a special indicator bit depending on the particular instruction.

If opcode 1 is 1, the instruction is interpreted as follows:

```
|0 0 1|   OP2   |   OPERAND     |            (5)
 0     3         8               15
```

where OP2 is only 5-bits long but the operand field is 8-bits long. There is neither an index nor an indirect bit with this format.

If opcode 1 is 7, the instruction is interpreted as a 32-bit instruction. The format is:

```
|1 1 1|   OP2   |I|X| BASE |            O F F S E T                |(6)
 0     3         8 9 10     13                                    31
```

where opcode 2 (OP2) is 5-bits long and the offset field is 19-bits long. The interpretation of the base bits is the same as in the 16-bit case.

6.2 DETAILED DESCRIPTION OF EACH INSTRUCTION

The following conventions and abbreviations are used in this section:

1) All offsets are byte offsets unless otherwise specified.
2) All instructions are unprivileged unless otherwise specified.
3) "Format" refers to the numbered instruction formats in Section 6.1.
4) All opcodes are octal numbers.
5) The three MSB's of all absolute addresses (content of limit registers, parameters for some privileged instructions, etc) must be zero.
6) "Indicators" refer to the carry, overflow, inexact result, and condition code bits in the Status register.
7) A register name, e.g. S, refers to the contents of the register while the name in parentheses, e.g. (S), refers

to the content of the memory location pointed to by the register.

8) Bit fields within a word are denoted by (N:M) where N is the starting bit (bit 0 is always the left-most bit) and M is the number of bits, e.g. S(8:8).

9) TOS stands for top-of-stack. The four top-most words in the stack are referred to as A, B, C, and D. A is the top-most word.

10) CIR is the Current Instruction register which holds each instruction as it is executed. Sixteen-bit instructions are right justified with 0 in the left halfword. It is not directly accessible by the machine instruction set.

11) E is the effective address referenced by an instruction after all indirect and indexed computation. If no explicit calculation of E is presented in an instruction, then E is implicitly calculated according to the addressing conventions presented in Section 2.2 [including data segment chaining described in Section 3.2.2] prior to any other operation in the instruction. Unless otherwise noted, postindexing is used.

12) As indicated in Section 2.2.4, the Index register ,X, contains a byte index.

13) All indirect pointers are full word quantities and must be aligned on full word boundaries in memory.

14) The bit patterns for all of the instructions are summarized in Appendix B.

15) Because the stack is word oriented, an expression like S:=S+1 should be interpreted as incrementing S to point to the next word and not to the next byte.

16) P+1 always refers to the next instruction and not to the next byte in a code segment.

17) The evaluation of a data segment pointer implies a call to the following procedure. The procedure starts with a data segment pointer [PTR] and produces the absolute address [ADDR] referenced by the pointer. In this description, the word "indexed" refers to the appropriate X bit in CIR, "unpriv" refers to the M bit in the Status register and "mod mem inst" refers to a modify memory instruction [i.e. those listed in the description of the Variable Trace trap in section 5.2.2]. The names of the fields in DST and page table entries [described in section 3.2.1] are used throughout and are implicitly updated each time a DST or page table entry is read. For simplicity, the semaphore reading and writing back of table entries [described in section 3.2.1] is not included in this description. The

```
        final values of ADDR, PTR, and OFFSET are available to the
        instructions which call this procedure.

        If PTR (0:2) = 2 then
        Begin
           ADDR:= DB + PTR (2:30)
           If indexed then ADDR:= ADDR + X
           If ADDR < DB or ADDR > DL then Bounds Violation Trap
        End
        If PTR (0:2) = 3 then
        Begin
           ADDR:= SB + PTR (2:30)
           If indexed then ADDR:= ADDR + X
           If ADDR < SB or ADDR > S then Bounds Violation Trap
        End
        If PTR (0:1) = 0 then
        Begin
           If PTR (1:1) = 0 then
           Begin
              TLENGTH: = Length of System DST
           End else
           Begin
              TLENGTH: = Length of User DST
           End
           If PTR (2:11) = 0 then EDS Pointer Violation Trap
           If TLENGTH < 16* PTR (2:11) then DST Violation Trap
           OFFSET := PTR (13:19)
           If indexed then OFFSET := OFFSET + X
           Linked := FALSE
READTBL:Read 4-word DST entry
           If P = 0 then
           Begin
              If OFFSET <0 then EDS Bounds Violation
              If OFFSET < LENGTH-1 then
              Begin
                 If mod mem inst and W=0 then Data Seg Write Viol
                 If unpriv and M=1 then Priv Mode Data Viol
                 PTR (13:19) := OFFSET (13:19)
                 If A=1 then Absent Segment Trap
                 R := 1
                 If mod mem inst then D:=1
                 ADDR:= LOCATION + OFFSET
              End else
              Begin
                 LINKED := TRUE
```

```
                IF C = 0 then
                Begin
                   If LINK = 0 then EDS Bounds Violation
                   OFFSET := OFFSET - LENGTH
                   PTR (1:12) := LINK
                   Go to READTBL
                End else
                Begin
                   PTR (1:12) := PTR (1:12) + OFFSET/LENGTH
                   OFFSET := OFFSET mod LENGTH
                   If PTR (1:12) > LINK then EDS Bounds Violation
                   Go to READTBL
                End
             End
          End else
          Begin
             If LINKED = TRUE then Unexpected EDS Type Trap
             If indexed then
             Begin
                TEMP:= PTR + X
                If TEMP (0:13) # PTR (0:13) then
                Begin
                   If TEMP (0:2)#PTR(0:2) then EDS Bounds Viol
                   PTR:= TEMP
                   If PTR (2:11) = 0 then EDS Pointer Violation
                   If TLENGTH < 16*PTR(2:11) then DST Violation
                   Read 1st 3 words of new DST entry
                   If P = 0 then Unexpected EDS Type Trap
                End
             End
             If A = 1 then Absent Page Table Trap
             PAGENUM:= PTR (13:19 - PAGESIZE)
             OFFSET:= PTR (32-PAGESIZE: PAGESIZE)
             If 8*PAGENUM + 4 > LENGTH-1 then Page Table Viol
             Read 2-word page table entry at LOCATION+8*PAGENUM
             If mod mem inst and W = 0 then Data Seg Write Viol
             If unpriv and M = 1 then Priv Mode Data Violation
             If A = 1 then Absent Page Trap
             R:= 1
             If mod mem inst then D:= 1
             ADDR:= LOCATION OR OFFSET
          End
       End
```

18) The evaluation of a local program pointer involves adding

the 30-bit offset in the pointer to PB and bounds checking the resulting address against PB and PL.

6.2.1 Load and Store Instructions

1. LD (Load) Format 6 Opcode 2=10
LDS Format 1 Opcode 1=2

```
S := S + 1
(S):= (E)
```

The content of E is pushed onto the stack. In the 16-bit version, the 8-bit offset field is interpreted as a word offset while in the 32-bit version, the 19-bit offset is a byte offset.
Indicators = CCA

1.5 LSI (Load Stack Indirect) Format 4 Opcode 3=5

```
DSPTR: = (S)
S:= S-1
(S + 1):= (Evaluation of DSPTR)
S:= S + 1
```

The data segment pointer in the TDS is replaced by the word to which it points [indexed if CIR (19:1) = 1].
Indicators = CCA

2. LDB (Load Byte) Format 6 Opcode 2=14

```
S := S+1
(S)(0:24):= 0
(S)(24: 8):=(E)
```

The content of E is right justified and pushed onto the stack.
Indicators = CCB

2.5 LBSI (Load Byte Stack Indirect) Format 4 Opcode 3=6

```
DSPTR: = (S)
S:= S-1
(S + 1)(0:24):= 0
(S + 1)(24:8):= (Evaluation of DSPTR)
S:= S + 1
```

The data segment pointer in the TOS is replaced by the byte

(right justified) to which it points [indexed if CIR (19:1) = 1].
Indicators = CCB

3. LDD (Load Double) Format 6 Opcode 2=20
LDDS Format 1 Opcode 1=4

```
S := S+2
(S-1):= (E)
(S):= (E+1)
```

The doubleword in E, E+1 is pushed onto the stack, in that order. In the 16-bit version, the 8-bit offset field is interpreted as a word offset. SEE APPENDIX G.
Indicators = CCA

3.5 LDSI (Load Double Stack Indirect) Format 4 Opcode 3=7

```
DSPTR:= (S)
S:= S-1
E:= Evaluation of DSPTR
(S+1, S+2): = (E, E+1)
S:= S+2
```

The data segment pointer in the TOS is replaced by the doubleword to which it points [indexed if CIR (19:1) = 1]. SEE APPENDIX G.
Indicators = CCA

4. LDH (Load Halfword) Format 6 Opcode 2=22

```
S:= S+1
(S)(16:16):=(E)
(S)(0:16):=S(16:1)
```

The content of E is right justified, sign extended, and pushed onto the stack.
Indicators = CCA

4.5 LHSI (Load Halfword Stack Indirect) Format 4 Opcode 3=10

```
DSPTR:= (S)
S:= S-1
(S+1) (16:16):= (Evaluation of DSPTR)
(S+1) (0:16):= (S+1) (16:1)
```

S:= S+1

The data segment pointer in the TOS is replaced by the halfword [right justified and sign extended] to which it points [indexed if CIR (19:1) = 1].
Indicators = CCA

4.6 LBIT [Load Bit] Format 4 Opcode 3 = 2

```
X:= (S) (0:29)
X (0:3):= (S) (0:1)
BITNO:= (S) (29:3)
DSPTR:= (S-1)
S:= S-2
BYTE:= (Evaluation of DSPTR, Indexed)
(S+1):= 0
(S+1) (31:1):= BYTE (BITNO:1)
S:= S+1
```

This instruction expects a 2's complement bit offset in the TOS and a data segment pointer in the second word in the stack. These two words are popped from the stack and the single bit pointed to by the pointer - offset combination is right justified and pushed onto the stack. The left-most bit of the byte pointed to by the pointer is bit 0. Note that CIR (19:1) must be 1 and that X is modified to be the appropriate byte offset.
Indicators = CCA

5. LDXA (Load X onto Stack) Format 3 Opcode 3 = 44

```
S := S+1
(S):= X
```

The content of X is pushed onto the stack.
Indicators = CCA on new TOS.

6. ZERO (Push Zero) Format 3 Opcode 3=6

```
S := S+1
(S):= 0
```

A zero word is pushed onto the stack.
Indicators = CCA

7. DZRO (Double Push Zero) Format 3 Opcode 3=7

```
        S := S+2
        (S-1):= 0
        (S):= 0
```

Two words containing all zeros are pushed onto the stack.
Indicators = CCA

8. LDI (Load Immediate) Format 6 Opcode 2=6
   LDIS Format 5 Opcode 2=14

```
        S:= S+1
        (S):=Immediate Operand
```

The immediate positive quantity in the operand field (CIR(8:24) in the 32-bit version, CIR(24:8) in the 16-bit version) is pushed onto the stack as a positive integer.
Indicators = CCA

9. LNI (Load Negative Immediate) Format 6 Opcode 2=7
   LNIS Format 5 Opcode 2=15

```
        S := S+1
        (S):=Immediate Operand
        (S) := -(S)
```

The two's complement of the immediate positive quantity in the operand field (CIR(8:24) in the 32-bit version, CIR(24:8) in the 16-bit version) is pushed onto the stack as a negative integer.
Indicators = CCA

10. LRA (Load Relative Address) Format 6 Opcode 2=36
    LRAS Format 1 Opcode 1=3

```
        If direct then
        Begin
             If P-relative then (S+1):= E - PB else
             Begin
                  If DB-relative or DL-relative then
                  Begin
                       (S+1):=Base + Offset - DB
                       (S+1)(0:2):= 10
                  End else
                  Begin
                       (S+1):=Base + Offset - SB
```

```
                    (S+1)(0:2):= 11
               End
               If indexed then (S+1):=(S+1)+X
          End
     End else
     Begin
          If P-relative then (S+1):= E - PB else
          Begin
               (S+1):=(Base + Offset)
               If indexed then (S+1):=(S+1)+X
          End
     End
     S:=S+1
```

In direct addressing mode this instruction pushes onto the stack a pointer to the computed address (using byte indexing). For an address in a code segment this is a PB-relative byte offset. For an address in the stack segment the stack segment pointer format is used and for an address in the global data segment the global data segment pointer format is used.

For indirect addressing in code segments the pointer is treated as a self-relative pointer and a PB-relative byte offset is pushed onto the stack. In all other cases, the pointer (possibly indexed) is loaded onto the stack. If indexing causes overflow then a trap to STT entry number 16 in code segment 1 is generated.
Indicators Unaffected

11. LPP (Load Program Pointer) Format 6 Opcode 2=4
LPPS Format 5 Opcode 2=34

In the 16-bit version, Offset=4*CIR(24:8). In the 32-bit version, Offset=CIR(13:19).

```
     If Offset>(PL)(13:19) then STT Violation Trap else
     Begin
          S:=S+1
          (S):=(PL-Offset)
          If (S)(0:1)=0 then
          Begin
               (S):=0
               (S)(0:1):=1
               (S)(1:19):=Offset
```

```
                    (S)(20:12):=STATUS(20:12)
                End
            End
```

The program pointer in the STT at PL-Offset is loaded onto the stack and converted to an external program pointer if it is a local pointer.
Indicators Unaffected

12. PLDA (Privileged Load from Absolute Address) Fmt4 Op3=64

```
XX:=If CIR (19:1)=1 then X else 0
(S):=((S)+XX)
```

The absolute address in the TOS is replaced by the word to which it points [indexed if CIR (19:1)=1]. Warning: This instruction should not be used to access memory between Q and SL in an active stack segment. This is a privileged instruction
Indicators = CCA

12.1 PLBA (Priv Load Byte from Absolute Address) Fmt 4 Op3=47

```
XX:= If CIR(19:1)=1 then X else 0
(S) (24:8):= ((S)+XX)
(S) (0:24):=0
```

The absolute address in the TOS is replaced by the byte (right justified) to which it points [indexed if CIR (19:1)=1]. See warning with PLDA. This is a privileged instruction.
Indicators = CCB

12.2 PLHA (Priv Load Halfword from Absolute Address) Fmt4 Op3=51

```
XX:= If CIR (19:1)=1 then X else 0
(S) (16:16):= ((S)+XX)
(S) (0:16):= (S) (16:1)
```

The absolute address in the TOS is replaced by the halfword (right justified and sign extended) to which it points [indexed if CIR (19:1)=1]. See warning with PLDA. This is a privileged instruction.
Indicators = CCA

13. ST (Store) Format 6 Opcode 2=13

STS Format 1 Opcode 1=5

If P-relative then Code Segment Violation else
(E):=(S)
S:=S-1

The content of the top-of-stack is stored in memory location E. The stack is popped. In the 16-bit version, the 8-bit offset field is interpreted as a word offset while in the 32-bit version, the 19-bit offset is a byte offset.
Indicators Unaffected

13.5 SSI (Store Stack Indirect) Format 4 Opcode 3=25

DSPTR:= (S-1)
S:= S-2
(Evaluation of DSPTR):=(S+2)

The content of the TOS is stored in the memory location pointed to by the data segment pointer [indexed if CIR (19:1)=1] in the second word in the stack. Both words are popped from the stack.
Indicators Unaffected

14. STB (Store Byte) Format 6 Opcode 2=34

If P-relative then Code Segment Violation else
(E):=(S)(24:8)
S:=S-1

The least significant byte of the top-of-stack is stored in memory location E. The stack is popped.
Indicators Unaffected

14.5 SBSI (Store Byte Stack Indirect) Format 4 Opcode 3=26

DSPTR:= (S-1)
S:= S-2
(Evaluation of DSPTR):= (S+2)(24:8)

The least significant byte of the TOS is stored in the memory location pointed to by the data segment pointer (indexed if CIR (19:1)=1) in the second word in the stack. Both words are popped from the stack.
Indicators Unaffected

15. STD (Store Double) Format 6 Opcode 2=35

```
If P-relative then Code Segment Violation else
(E) := (S-1)
(E+1) := (S)
S := S-2
```

The top-most two words on the stack are stored in memory locations E and E+1. The stack is popped. SEE APPENDIX G.
Indicators Unaffected

15.5 SDSI (Store Double Stack Indirect) Format 4 Opcode 3=27

```
DSPTR:= (S-2)
S:= S-3
E:= Evaluation of DSPTR
(E,E+1):= (S+2, S+3)
```

The top-most two words on the stack are stored in the memory location pointed to by the data segment pointer [indexed if CIR (19:1)=1] in the third word in the stack. All three words are popped from the stack. SEE APPENDIX G.
Indicators Unaffected

16. STH (Store Halfword) Format 6 Opcode 2=25

```
If P-relative then Code Segment Violation else
(E):=(S)(16:16)
S:=S-1
```

The right halfword in the TOS is stored in memory location E. The stack is popped.
Indicators Unaffected

16.5 SHSI (Store Halfword Stack Indirect) Fmt 4 Op3=30

```
DSPTR:= (S-2)
S:= S-2
(Evaluation of DSPTR):= (S+2)(16:16)
```

The right halfword in the TOS is stored in the memory location pointed to by the data segment pointer [indexed if CIR(19:1)=1] in the second word in the stack. Both words are popped from the stack.
Indicators Unaffected.

16.6 SBIT (Store Bit) Format 4 Opcode 3=3

```
X:= (S-1)(0:29)
X(0:3):=(S-1)(0:1)
BITNO:=(S-1)(29:3)
DSPTR:= (S-2)
S:= S-3
(Evaluation of DSPTR, Indexed)(BITNO:1):= (S+3)(31:1)
```

This instruction expects a 2's complement bit offset in the second word on the stack and a data segment pointer in the third word in the stack. The least significant bit of the TOS is stored in the bit position pointed to by the pointer-offset combination. The left-most bit of the byte pointed to by the pointer is bit 0. All three words are popped from the stack. Note that CIR (19:1) must be 1 and that X is modified to be the appropriate byte offset.
Indicators Unaffected

17. STAX (Store A in X) Format 3 Opcode 3=43

```
X := (S)
If CIR(19:1)=1 then S := S-1
```

The top word in the stack is stored in the Index register.
If CIR(19:1) = 1, then the TOS is deleted.
Indicators = CCA on the new X

18. PSTA (Privileged Store into Absolute Address)Fmt4 Op3=65

```
XX:= If CIR (19:1)=1 then X else 0
((S-1)+XX):=(S)
S:=S-2
```

The top word of the stack is stored at the address [indexed if CIR (19:1)=1] contained in the second word in the stack. Both words are popped. Warning: This instruction should not be used to access memory between Q and SL in an active stack segment. This is a privileged instruction.
Indicators Unaffected

18.1 PSBA (Priv Store Byte into Absolute Address) Fmt 4 Op3 = 50

```
XX:= If CIR (19:1)=1 then X else 0
((S-1)+XX):= (S)(24:8)
S:= S-2
```

The least significant byte of the TOS is stored at the absolute address [indexed if CIR (19:1)=1] contained in the second word in the stack. Both words are popped. See warning with PSTA. This is a privileged insruction.
Indicators Unaffected.

18.2 PSHA (Priv Store Halfword into Absolute Address) Fmt4 Op3=52

```
XX:= If CIR (19:1)=1 then X else 0
((S-1)+XX):= (S)(16:16)
S:= S-2
```

The right halfword of the TOS is stored at the absolute address [indexed if CIR (19:1)=1] contained in the second word in the stack. Both words are popped. See warning with PSTA. This is a privileged instruction.
Indicators Unaffected

6.2.2 Stack and Register Manipulation Instructions

19. DEL (Delete A) Format 3 Opcode 3=40

```
S:=S-1
```

The TOS is deleted.
Indicators Unaffected

20. DDEL (Double Delete) Format 3 Opcode 3=2

```
S:=S-2
```

The top two words of the stack are deleted.
Indicators Unaffected

21. DELB (Delete B) Format 3 Opcode 3=1

```
(S-1):=(S)
S:=S-1
```

The second word in the stack is deleted and the stack is compressed. The TOS is unchanged.
Indicators Unaffected

22. DUP (Duplicate A) Format 3 Opcode 3=45

```
S := S+1
(S) := (S-1)
```

The top word of the stack is duplicated by pushing a copy of it onto the stack.
Indicators = CCA

23. DDUP (Double Duplicate) Format 3 Opcode 3=46

```
S := S+2
(S-1) := (S-3)
(S) := (S-2)
```

The doubleword in the top two words of the stack is duplicated by pushing a copy of it onto the stack.
Indicators = CCA on doubleword

24. XCH (Exchange A and B) Format 3 Opcode 3=32

```
TEMP:=(S-1)
(S-1):=(S)
(S):=TEMP
```

The top two words of the stack are interchanged.
Indicators = CCA on new TOS

25. DXCH (Double Exchange) Format 3 Opcode 3=16

```
TEMP:=(S-2)
(S-2):=(S)
(S):=TEMP
TEMP:=(S-3)
(S-3):=(S-1)
(S-1):=TEMP
```

The top two doubleword pairs are interchanged on the stack.
Indicators = CCA on new TOS doubleword pair.

26. XAX (Exchange A and X) Format 3 Opcode 3=35

```
TEMP := X
X := (S)
(S) := TEMP
```

The TOS and Index registers are exchanged.

Indicators = CCA on new TOS

27. CAB (Rotate ABC) Format 3 Opcode 3=56

```
TEMP:=(S-2)
(S-2):=(S-1)
(S-1):=(S)
(S):=TEMP
```

The third word in the stack is removed from the stack, the stack is compressed, and the original third word is pushed onto the stack.
Indicators = CCA on the new TOS

28. LDXI (Load X Immediate) Format 5 Opcode 2=12

```
X(0:24) := 0
X(24:8) := CIR (24:8)
```

The immediate positive quantity in the operand field is loaded into the Index register.
Indicators = CCA

29. LDXN (Load X Negative Immediate) Format 5 Opcode 2=13

```
X(0:24) := 0
X(24:8) := CIR(24:8)
X := -X
```

The two's complement of the immediate positive quantity in the operand field is loaded into the Index register as a negative integer.
Indicators = CCA

30. LDX (Load Index) Format 6 Opcode 2=26

```
X:=(E)
```

The content of E is loaded into the Index register.
Indicators = CCA

33. TEST (Test TOS) Format 3 Opcode 3=25

Set condition code in Status register based on TOS. If CIR(19:1)=1, the TOS is deleted.

Indicators = CCA

34. TSTB (Test Byte) Format 3 Opcode 3=27

Set condition code in Status register based on least significant byte in TOS. If CIR(19:1)=1, the TOS is deleted.
Indicators = CCB

34.5 TSTD (Test Double) Format 3 Opcode 3=26

Set condition code in Status register based on the doubleword in the top-most two words in the stack. If CIR(19:1)=1, the doubleword is deleted.
Indicators = CCA

36. ADDS (Add to S) Format 5 Opcode 2=32

```
If CIR(24:8) > 0 then S := S+CIR(24:8) else
Begin
     TEMP := (S)
     S := S-1
     S := S+TEMP
End
```

If the 8-bit operand is zero, the content of the TOS is added to S-1. Otherwise the operand, a positive integer, is added to S. This is a privileged instruction.
Indicators Unaffected

36.5 PSHN (Push N Words) Format 3 Opcode 3=5

```
TEMP:=0
If CIR(19:1)=1 then TEMP:=(S);S:=S-1
While X>0 do
Begin
     S:=S+1
     (S):=TEMP
     X:=X-1
End
```

This instruction pushes either N zeros or N copies of the TOS onto the stack. The value of N is in the Index register. The TOS is duplicated if CIR(19:1)=1.
Indicators = If original X>0 then CCA else unaffected

37. SUBS (Subtract from S) Format 5 Opcode 2=33

```
If CIR(24:8) > 0 then S:= S-CIR(24:8) else
Begin
     TEMP := (S)
     S := S-1
     If TEMP > 0 then S := S-TEMP
End
```

If the 8-bit operand is zero, the content of the TOS, which must be positive, is subtracted from S-1. Otherwise the operand, a positive integer, is subtracted from S.
Indicators Unaffected

38. PSHR (Push Registers) Format 2 Opcode 2=34

```
If CIR(31:1) = 1 then S:=S+1; (S):=S-SB-1
If CIR(30:1) = 1 then S:=S+1; (S):=Q-SB
If CIR(29:1) = 1 then S:=S+1; (S):=STATUS
If CIR(28:1) = 1 then S:=S+1; (S):=DST
If CIR(27:1) = 1 then S:=S+1; (S):=BRKPT
If CIR(26:1) = 1 then S:=S+1; (S):=0; (S)(24:8):=FLAGS
If CIR(19:1) = 1 then
Begin
     S:=S+1; (S):=P
     S:=S+1; (S):=PB
     S:=S+1; (S):=PL
     S:=S+1; (S):=DB
     S:=S+1; (S):=DL
     S:=S+1; (S):=SB
     S:=S+1; (S):=SL
End
```

The registers specified by CIR(19:1) and CIR(26:6) are pushed onto the stack in the sequence shown.
Indicators Unaffected

39. SETR (Set Registers) Format 2 Opcode 2=35

```
If CIR(26:1) = 1 then FLAGS:=(S)(24:8); S:=S-1
If CIR(27:1) = 1 then BRKPT:=(S); S:=S-1
If CIR(28:1) = 1 then DST:=(S); S:=S-1
If CIR(29:1) = 1 then STATUS:=(S); S:=S-1
If CIR(30:1) = 1 then Q:=(S)+SB; S:=S-1
```

If CIR(31:1) = 1 then S:=(S)+SB

The registers specified by CIR(26:6) are set from the stack in the sequence specified. An attempt to set BRKPT, DST, or S in unprivileged mode causes an immediate Mode Violation Trap (no registers are modified). In unprivileged mode only the user-related FLAGS are modified and only STATUS(0:9), STATUS(10:3), and STATUS(16:2) are modified. In privileged mode it is not possible to modify STATUS(9:1), STATUS(19:1), and STATUS(20:12). If the new values of Q and S do not satisfy SB<Q<S<SL then Q and S are restored to their original values and an inconsistent registers trap occurs. Note that DST is not saved on interrupts or traps and IXIT uses the information in the Task Control Block to restore this register.
Indicators Unaffected except by loading the Status register

40. XGDS (Exchange Global Data Segment) Format 4 Opcode 3=14

Ext. Data Seg. Ptr. := (S)
DB:= Seg. Location from DST entry
DL:= DB + Seg Length from DST entry
Increment Use Count for New Segment
TEMP:= ((CTCBP)+2)(20:12)
((CTCBP)+2)(20:12):= (S)(1:12)
(S)(1:12):= TEMP
Decrement Use Count for Old Segment

This instruction expects an external data segment pointer on the stack that points to a new global data segment. The offset field in this pointer is ignored. The DB and DL registers are set from the location and length information in the DST. The Write bit must be set in this DST entry. The segment number of the new global data segment is exchanged with the global data segment number in the current Task Control Block.
Indicators Unaffected

41. ASL (Arithmetic Shift Left) Format 2 Opcode 2=6

Note: In all of the shift instructions CIR(19:1) is an index bit and CNT=6 least significant bits of (CIR(26:6)+(if CIR(19:1)=1 then X else 0))

The TOS is shifted left CNT bits. If the sign bit changes,

overflow is set. This instruction multiplies the TOS by a power of 2.
Indicators=CCA, Overflow

42. ASR (Arithmetic Shift Right) Format 2 Opcode 2=7

See note under ASL. If the TOS is negative then it is negated. The TOS is shifted right CNT bits, filling with zeros. If the original TOS was negative, then the shifted TOS is negated. This instruction divides the TOS by a power of 2. (EXF can be used for sign extension.)
Indicators=CCA

43. LSL (Logical Shift Left) Format 2 Opcode 2=2

See note under ASL. The TOS is shifted left CNT bits, filling with zeros.
Indicators=CCA

44. LSR (Logical Shift Right) Format 2 Opcode 2=3

See note under ASL. The TOS is shifted right CNT bits, filling with zeros.
Indicators=CCA

45. CSL (Circular Shift Left) Format 2 Opcode 2=4

See note under ASL. The TOS is shifted left CNT bits circularly.
Indicators=CCA

46. CSR (Circular Shift Right) Format 2 Opcode 2=5

See note under ASL. The TOS is shifted right CNT bits circularly.
Indicators=CCA

53. EXF (Extract Field) Format 6 Opcode 2=0

J=CIR(27:5) specifies the first (leftmost) bit in the TOS
K=CIR(22:5) specifies 32-the number of bits in the TOS
L=32-K

If indexed then CIR(22:10):= CIR(22:10)+X

(S)(K:L) := (S)(J:L)
If CIR(8:1)=0 then (S)(0:K) := 0 else sign extend L-bit field

Bits J, J+1, J+2,...J+L-1 (all bits greater than 31 have a value of 0) are extracted from the TOS and the TOS is deleted. The L-bit field is right justified with any leading bits 0 if CIR(8:1)=0 or sign extended if CIR(8:1)=1 and this result is pushed onto the stack.
Indicators=CCA on new TOS

54. DPF (Deposit Field) Format 6 Opcode 2=1

J=CIR(27:5) specifies the first (leftmost) bit in the destination field
K=CIR(22:5) specifies 32-the number of bits in the field
L=32-K

If indexed then CIR(22:10):= CIR(22:10) + X
(S-1)(J:L):= (S)(K:L)
(S-1)(0:J):= (S-1)(0:J)
(S-1)(J+L:K-J):= (S-1)(J+L:K-J)
S:= S-1

The L least significant bits of the TOS are placed in bits J, J+1, J+2,...J+L-1 (all bits greater than 31 are lost) in the second word of the stack. The remaining bits of the second word in the stack are unchanged. The TOS is deleted.
Indicators=CCA on new TOS

55. LSXL (Logical Shift X Left) Format 2 Opcode 2=10

See note under ASL. The content of the Index register is shifted left CNT bits.
Indicators = CCA on X

6.2.3 Arithmetic and Logical Instructions

56. ADD (Add) Format 3 Opcode 3=20

(S-1):=(S-1)+(S)+If CIR(19:1)=1 then Carry else 0
S:= S-1

The top two words in the stack are popped from the stack and

added together (with Carry) as two's complement integers. The sum is pushed onto the stack.
Indicators = If CIR(19:1)=0 then CCA, Overflow
If CIR(19:1)=1 then CCA only if result#0, Overflow

57. SUB (Subtract) Format 3 Opcode 3=21

```
(S-1):=(S-1)-(S)-If CIR(19:1)=1 then 1-Carry else 0
S:= S-1
```

The top word in the stack is subtracted (with Carry) from the second word in the stack. This is a two's complement integer subtraction. Both words are popped from the stack and the difference is pushed onto the stack.
Indicators = If CIR(19:1)=0 then CCA, Overflow
If CIR(19:1)=1 then CCA only if result#0, Overflow

58. MPY (Multiply) Format 3 Opcode 3=22

```
(S-1,S):= (S-1)*(S)
If CIR(19:1)=0 then
Begin
     Set Overflow based on (S-1) and (S)(0:1)
     (S-1) := (S)
     S := S-1
End
```

The top two words in the stack are popped from the stack, multiplied together as two's complement integers, and the doubleword product is pushed onto the stack. If CIR (19:1)=0 then overflow is set if the most significant 33 bits of the product are not all zeros or all ones and the most significant half of the product is deleted.
Indicator = CCA, Overflow

59. DIV (Divide) Format 3 Opcode 3=23

```
TEMP := (S-2,S-1)/(S)
(S) := remainder of (S-2,S-1)/(S)
(S-2,S-1) := TEMP
```

The doubleword integer in the second and third words in the stack is divided by the integer in the TOS producing a doubleword quotient and a singleword remainder. The divisor

and dividend are popped from the stack and the quotient and the remainder are pushed onto the stack (in that order). If CIR(19:1)=0 then the sign of the remainder is the same as the sign of the dividend. If CIR(19:1)=1 then the remainder is always positive. In both cases, dividend = quotient*divisor + remainder.
Indicators = CCA on quotient, Overflow.

59.1 DIVS (Divide, Single Word Dividend) Fmt 3 Op3=33

```
TEMP := (S-1)/(S)
(S) := remainder of (S-1)/(S)
(S-1) := TEMP
```

The integer in the second word in the stack is divided by the integer in the TOS producing a singleword quotient and a singleword remainder. The divisor and dividend are popped from the stack and the quotient and remainder are pushed onto the stack (in that order). If CIR(19:1)=0 then the sign of the remainder is the same as the sign of the dividend. If CIR(19:1)=1 then the remainder is always positive. In both cases, dividend = quotient*divisor + remainder.
Indicators = CCA on quotient, Overflow.

60. NEG (Negate) Format 3 Opcode 3=24

```
(S):= -(S)
```

The TOS is replaced by its two's complement.
Indicators = CCA, Overflow

61. CMP (Compare) Format 3 Opcode 3 = 17

```
Set Condition Code based on (S-1) and (S)
S := If CIR(19:1)=1 then S-2 else S-1
```

The condition code is set to CCC as a result of the integer comparison of (S-1), operand 1, and (S), operand 2. Both operands are deleted if CIR(19:1)=1. Otherwise only the TOS is deleted.
Indicators = CCC

61.1 LADD (Logical Add) Format 3 Opcode 3=60

(S-1):=(S-1)+(S)+If CIR(19:1)=1 then Carry else 0
S:=S-1

The top two words are popped from the stack and added together (with Carry) as 32-bit positive integers. The sum is pushed onto the stack.
Indicators = If result#0 then CCG, Carry
If result=0 and CIR(19:1)=0 then CCE, Carry
If result=0 and CIR(19:1)=1 then Carry

61.2 LSUB (Logical Subtract) Format 3 Opcode 3=61

(S-1):=(S-1)-(S)-If CIR(19:1)=1 then 1-Carry else 0
S:=S-1

The top two words are popped from the stack and subtracted (with Carry) as 32-bit positive integers. The difference is pushed onto the stack.
Indicators = If result#0 then CCG, Carry
If result=0 and CIR(19:1)=0 then CCE, Carry
If result=0 and CIR(19:1)=1 then Carry

61.3 LMPY (Logical Multiply) Format 3 Opcode 3=62

(S-1,S):=(S-1)*(S)

The top two words are popped from the stack and multiplied together as 32-bit positive integers. The product is pushed onto the stack.
Indicators = If result#0 then CCG else CCE

61.5 LCMP (Logical Compare) Format 3 Opcode 3=57

Set Condition Code based on (S-1) and (S)
S:=If CIR(19:1)=1 then S-2 else S-1

The condition code is set to CCC as a result of the comparison of (S-1), operand 1, and (S), operand 2, as 32-bit positive integers. Both operands are deleted if CIR(19:1)=1. Otherwise only the TOS is deleted.
Indicators = CCC

68. ADDI (Add Immediate) Format 5 Opcode 2=4

(S):= (S)+CIR(24:8)

The immediate positive quantity CIR (24:8) is added to the TOS in integer form and the sum replaces the TOS.
Indicators = CCA, Overflow

69. SUBI (Subtract Immediate) Format 5 Opcode 2=5

(S):= (S)-CIR(24:8)

The immediate positive quantity CIR(24:8) is subtracted from the TOS in integer form and the difference replaces the TOS.
Indicators = CCA, Overflow

70. MPYI (Multiply Immediate) Format 5 Opcode 2=6

(S):= (S)*CIR(24:8)

The immediate positive quantity CIR(24:8) is multiplied with the TOS in integer form. The least significant word of the doubleword product replaces the TOS. If the most significant 33 bits of the doubleword product are not all zeros or all ones then overflow is set.
Indicators = CCA, Overflow

71. DIVI (Divide Immediate) Format 5 Opcode 2=7

(S):= (S)/CIR(24:8)

The TOS is divided by the immediate positive quantity CIR (24:8) in integer form. The quotient replaces the TOS.
Indicators = CCA, Overflow

72. CMPI (Compare Immediate) Format 5 Opcode 2=3

Set Condtion Code based on (S) and CIR(24:8)
S:= S-1

The condtion code is set to CCC as a result of the integer comparison of (S), operand 1, and the positive quantity CIR(24:8), operand 2. The TOS is deleted.
Indicators = CCC

73. CMPN (Compare Negative Immediate) Format 5 Opcode 2=2

Set Condtion Code based on (S) and -CIR (24:8)
S:= S-1

The condition code is set to CCC as a result of the integer comparison of (S), operand 1, and the two's complement of the positive quantity CIR(24:8), operand 2. The TOS is deleted.
Indicators = CCC

74. ADAX (Add A to X) Format 3 Opcode 3=36

X:= X+(S)
If CIR(19:1)=1 then S := S-1

The TOS is added to the Index register and the sum replaces the original content of the Index register. If CIR(19:1)=1, the TOS is deleted.
Indicators = CCA, and Overflow on new X

75. ADXA (Add X to A) Format 3 Opcode 3=37

(S):= (S)+X

The Index register is added to the TOS and the sum replaces the original TOS.
Indicators = CCA, and Overflow on new TOS

76. ADXI (Add Immediate to X) Format 5 Opcode 2=26

X:= X+CIR(24:8)

The immediate positive quantity CIR(24:8) is added to the Index register in integer form.
Indicators = CCA, and Overflow on X

77. SBXI (Subtract Immediate from X) Format 5 Opcode 2=27

X:= X-CIR(24:8)

The immediate positive quantity CIR(24:8) is subtracted from the index register in integer form.
Indicators = CCA, and Overflow on X

80. CMPM (Compare Memory) Format 6 Opcode 2=15

Set condition Code based on (S) and (E)
S:= S-1

The condition code is set to CCC as a result of the integer comparison of (S), operand 1, and (E), operand 2. The TOS is deleted.
Indicators = CCC

81. INCM (Increment Memory) Format 6 Opcode 2=24

(E):= (E)+1

The content of E is incremented by 1 in integer form.
Indicators = CCA, Overflow

81.1 DECM (Decrement Memory) Format 6 Opcode 2=23

(E) := (E)-1

The content of E is decremented by 1 in integer form.
Indicators = CCA, and Overflow

81.2 INCF (Increment Memory by Four) Format 6 Opcode 2=27

(E) := (E)+4

The content of E is incremented by 4 in integer form.
Indicators = CCA, Overflow

81.3 DECF (Decrement Memory by Four) Format 6 Opcode 2=32

(E) := (E)-4

The content of E is decremented by 4 in integer form.
Indicators = CCA, Overflow

82. FADD (Floating Add) Format 3 Opcode 3=51

(S-3,S-2):= (S-3,S-2)+(S-1,S)
S:= S-2

The two 64-bit floating point numbers contained in the top 4 words of the stack are added together. The top 4 words of the stack are deleted and the two-word sum is pushed onto the stack.
Indicators = CCA, Inexact Result

83. FSUB (Floating Subtract) Format 3 Opcode 3=52

(S-3,S-2):= (S-3,S-2)-(S-1,S)

S:= S-2

The 64-bit floating point number contained in the top two words in the stack is subtracted from the floating point number contained in the next two words in the stack. The top 4 words of the stack are deleted and the 64-bit difference is pushed onto the stack.
Indicators = CCA, Inexact Result

84. FMPY (Floating Multiply) Format 3 Opcode 3=53

(S-3,S-2):= (S-3,S-2)*(S-1,S)
S:= S-2

The two 64-bit floating point numbers contained in the top 4 words in the stack are multiplied togther. The top 4 words of the stack are deleted and the 64-bit product is pushed onto the stack.
Indicators = CCA, Inexact Result

85. FDIV (Floating Divide) Format 3 Opcode 3=54

(S-3, S-2):= (S-3,S-2)/(S-1,S)
S:= S-2

The 64-bit floating point number contained in the third and fourth words in the stack is divided by the floating point number contained in the top two words in the stack. The top 4 words of the stack are deleted and the 64-bit quotient is pushed onto the stack.
Indicators = CCA, Inexact Result

86. FNEG (Floating Negate) Format 3 Opcode 3=55

(S-1,S):= -(S-1,S)

The 64-bit floating point number contained in the top two words of the stack is negated, i.e. subtracted from +0.
Indicators = CCA

87. FCMP (Floating Compare) Format 3 Opcode 3=50

Set Condition Code based on (S-3,S-2) and (S-1,S)
S := If CIR(19:1)=1 then S-4 else S-2

The condition code is set to CCC as a result of the floating point comparison of (S-3,S-2), operand 1, and (S-1,S), operand 2. Both operands are deleted if CIR(19:1)=1. Otherwise only (S-1,S) is deleted.
Indicators = CCC

87.1 FTST (Floating Test) Format 3 Opcode 3=3

Set Condition Code based on (S-1,S)
If CIR (19:1)=1 then S:=S-2

The condition code is set based on the doubleword floating point number in the top two words in the stack. If this number is -0, the result is CCE. If CIR(19:1)=1, the double word is deleted.
Indicators = CCA

88. FADS (Floating Add Single) Format 3 Opcode 3=72

(S-1):= (S-1)+(S)
S:= S-1

The two 32-bit floating point numbers contained in the top two words of the stack are added together. The top two words are deleted and the one-word sum is pushed onto the stack.
Indicators = CCA, Inexact Result

89. FSBS (Floating Subtract Single) Format 3 Opcode 3=73

(S-1):= (S-1)-(S)
S:= S-1

The 32-bit floating point number contained in the TOS is subtacted from the floating point number contained in the second word in the stack. The top two words are deleted and the 32-bit difference is pushed onto the stack.
Indicators = CCA, Inexact Result

90. FMPS (Floating Multiply Single) Format 3 Opcode 3=74

(S-1):= (S-1)*(S)
S:= S-1

The two 32-bit floating point numbers contained in the top

two words in the stack are multiplied together. The top two words are deleted and the 32-bit product is pushed onto the stack.
Indicators = CCA, Inexact Result

91. FDVS (Floating Divide Single) Format 3 Opcode 3=75

(S-1):= (S-1)/(S)
S:= S-1

The 32-bit floating point number contained in the second word in the stack is divided by the floating point number contained in the TOS. The top two words are deleted and the 32-bit quotient is pushed onto the stack.
Indicators = CCA, Inexact Result

92. FNGS (Floating Negate Single) Format 3 Opcode 3=76

(S):= -(S)

The 32-bit floating point number contained in the TOS is negated, i.e. subtracted from + 0.
Indicators = CCA

93. FCPS (Floating Compare Single) Format 3 Opcode 3=77

Set Condition Code based on (S-1) and (S)
S := If CIR(19:1)=1 then S-2 else S-1

The condition code is set to CCC as a result of the floating point comparison of (S-1), operand 1, and (S), operand 2. Both operands are deleted if CIR(19:1)=1. Otherwise only the TOS is deleted.
Indicators = CCC

93.1 FTSS (Floating Test Single) Format 3 Opcode 3=4

Set Condition Code based on (S)
If CIR (19:1)=1 then S:=S-1

The condition code is set based on the singleword floating point number in the TOS. If this number is -0, the result is CCE. If CIR (19:1)=1 the TOS is popped.
Indicators = CCA

93.8 DLAD (Decimal Logical Add) Format 3 Opcode 3=11

```
(S-1):=(S-1)+(S)+If CIR(19:1)=1 then Carry else 0
S := S-1
```

The top two words are popped from the stack and added together (with Carry) as 8-digit unsigned decimal numbers. The sum is pushed onto the stack.
Indicators = If result#0 then CCG, Carry
If result=0 and CIR(19:1)=0 then CCE, Carry
If result=0 and CIR(19:1)=1 then Carry

93.9 DLSB (Decimal Logical Subtract) Format 3 Opcode 3=12

```
(S-1):=(S-1)-(S)-If CIR(19:1)=1 then 1-Carry else 0
S := S-1
```

The top two words are popped from the stack and subtracted (with Carry) as 8-digit unsigned decimal numbers. The difference is pushed onto the stack.
Indicators = If result#0 then CCG, Carry
If result=0 and CIR(19:1)=0 then CCE, Carry
If result=0 and CIR(19:1)=1 then Carry

94.1 DTB (Decimal to Binary) Format 4 Opcode 3=0

```
If CIR(19:1) = 0 then
Begin
     N:=decimal floating point number in (S-2,S-1,S)
     exponent(N):=0
     I:=N*10^16
     S:=S-1
     (S-1,S):=binary floating point equivalent of I
     Condition Code := CCA
End else
Begin
     If -308 < (S)(0:10) < 308 then
     Begin
          S:=S+3
          (S-3,S-2):=binary floating point number A
          (S-1,S):=binary floating point number B
          Condition Code:=CCG
     End else Exponent Size Trap
End
```

If CIR(19:1)=0, this instruction expects a 96-bit decimal floating point number, N, in the top three words in the stack. This number is popped from the stack and the decimal digits, treated as a 17-digit integer, are converted to a binary floating point number which is pushed onto the stack. The condition code bits are set to CCA.

If CIR(19:1)=1, this instruction expects the exponent word of a 96-bit decimal floating point number in the TOS. If the exponent does not satisfy -308<exponent<308 then an exponent size trap occurs. Otherwise, the exponent word is popped from the stack and two 64-bit binary floating point numbers, A and B, are pushed onto the stack. These two numbers are chosen so that A/B = 10^(exponent(N)-16). This offset of 16 is taken into account by this instruction when CIR(19:1)=0. The condition code bits are set to CCG. See Appendix E for a routine which can be used for decimal to binary floating point conversion.
Indicators = Condition Code, Inexact result if CIR(19:1)=0

94.3 BTD (Binary to Decimal) Format 4 Opcode 3=1

```
If CIR(19:1)=0 then
Begin
     I:=(S-2)
     N:=binary floating point number in (S-1,S)
     If N=0 then (S-2):=0 else
     Begin
          If 1<integer part(absolute value(N))<9 then
          Begin
               If -512 < I < 511 then
               Begin
                    (S-2)(0:10):=I(22:10)
                    (S-2)(15:17),(S-1),(S):=decimal
                    equivalent of N
                    Condition Code:=CCE
               End else Exponent Size Trap
          End
          If integer part(absolute value(N)) < 1 then
          Begin
               S:=S-2
               (S):=(S)-1
               Condition Code:=CCL
          End else
          Begin
```

```
                    S:=S-2
                    (S):=(S)+1
                    Condition Code:=CCG
               End
          End
     End else
     Begin
          N:=binary floating point number in (S-1,S)
          If N=0 then S:=S-1;(S):=0 else
          Begin
               S:=S-1
               (S):=integer part(log base ten(N))
               (approximate)
               If exponent(N) < 0 then (S):=(S)-1
          End
          Condition Code:=CCA
     End
```

If CIR(19:1)=0, this instruction expects an integer, I, [produced by this instruction when CIR(19:1)=1] in the third word in the stack and a 64-bit binary floating point number, N, in the top two words in the stack. If N=0 then I is ignored and the result is a three word decimal floating point zero. If the integer part of the absolute value of N is between 1 and 9 then N is converted to a three word decimal floating point number with the exponent of this number set equal to I. If the integer part of the absolute value of N is outside of the range 1 to 9 then N is popped from the stack, I is corrected appropriately, and the condition code is set to CCG or CCL as indicated.

If CIR(19:1)=1, this instruction expects a 64-bit binary floating point number, N, in the top two words in the stack. This number is popped and used to calculate an integer, I, between -308 and 308 which is pushed onto the stack and used to set the condition code. This integer is chosen so that when N is divided by 10^I (see POT instruction) the quotient is a number whose absolute value, A, satisfies 1<A<10 (or A=0 in N=0). Actually, a fast approximation is used to calculate I. For approximately 4% of all floating point numbers, the integer I will be in error by +1. This possible error is taken into account by this instruction when CIR(19:1)=0. See Appendix E for a routine which can be used for binary to decimal floating point conversion.
Indicators = Condition Code

94.5 POT (Power of Ten) Format 4 Opcode 3=53

```
If -308 < (S) < 308 then
Begin
     S:=S+3
     (S-3,S-2):=binary floating point number A
     (S-1,S):=binary floating point number B
     Condition Code:=CCG
End else Exponent Size Trap
```

This instruction converts an integer I, -308<I<308, in the TOS into two binary floating point numbers, A and B, which satisfy A/B = 10^I. See Appendix E.
Indicators = If trap then unaffected else CCG

95. ISC (Integer Size Check) Format 3 Opcode 3=34

```
If (S)(0:17) <> 0 then
Begin
     If (S)(0:17) <> -1 then Overflow:=1
End
```

A word integer in the TOS is checked. If the halfword integer format cannot accomodate the full word integer then overflow is set.
Indicators = CCA, Overflow

96. FLTS (Float Single) Format 3 Opcode 3=70

```
(S):=float(S)
```

The 32-bit integer on the TOS is converted (with rounding) to a 32-bit floating point number. The TOS is deleted and the floating point number is pushed onto the stack.
Indicators = CCA, Inexact Result

97. FLTD (Float Double) Format 3 Opcode 3=42

```
S:=S+1
(S-1,S):=float (S-1)
```

The 32-bit integer in the TOS is converted to a 64-bit floating point number.
Indicators = CCA

98. FIXD (Fix Double) Format 3 Opcode 3=41

(S-1):=If CIR(19:1)=0 then fix and truncate (S-1,S)
else fix and round (S-1,S)
S:=S-1

The 64-bit floating point number contained in the top two words of the stack is popped from the stack and converted to a fixed point number. If CIR(19:1)=0 then the integer part of this number is kept and the fractional part is truncated. If CIR(19:1)=1 then this number is rounded (by adding 0.5 to its magnitude) to an integer which is pushed onto the stack.
Indicators = CCA, Overflow

100. FIXS (Fix Single) Format 3 Opcode 3=71

(S):=If CIR(19:1)=0 then fix and truncate (S) else fix and round (S)

The 32-bit floating point number contained in the TOS is popped from the stack and converted to a fixed point number. If CIR(19:1)=0 then the integer part of this number is pushed onto the stack and the fractional part is truncated. If CIR(19:1)=1 then this number is rounded (by adding 0.5 to its magnitude) to an integer which is pushed onto the stack.
Indicators = CCA, Overflow

100.9 CSDF (Convert Single to Double Word Floating) Fmt 3 Op3=31

S:=S+1
(S):=0
(S-1,S):= convert (S-1)

The 32-bit binary floating point number in the TOS is converted to a 64-bit floating point number. The zero which is pushed onto the stack is left there in the event of a Floating Point Operand Trap.
Indicators = CCA

101. CDSF (Convert Double to Single Word Floating) Fmt 3 Op3=30

(S-1):=convert (S-1,S)
S:=S-1

The 64-bit binary floating point number contained in the top two words in the stack is converted to a 32-bit floating

point number.
Indicators = CCA, Inexact Result

101.1 SRM (Set Rounding Mode) Format 2 Opcode 2=33

```
If CIR(19:1)=0 then STATUS(11:2):=CIR(30:2) else
Begin
     STATUS(11:2):=(S)(30:2)
     S:=S-1
End
```

This instruction sets the Rounding Mode bits in the Status register from the two LSB'S of the instruction if CIR(19:1)=0 or the TOS if CIR(19:1)=1. In the latter case the stack is popped.
Indicators Unaffected

102. AND (Logical AND) Format 3 Opcode 3=67

```
(S-1):= (S-1) AND (S)
S:= S-1
```

The top two words in the stack are popped from the stack, ANDed together and the result is pushed onto the stack.
Indicators = CCA

103. OR (Logical OR) Format 3 Opcode 3=65

```
(S-1):= (S-1) OR (S)
S:= S-1
```

The top two words in the stack are popped from the stack, ORed together and the result is pushed onto the stack.
Indicators = CCA

104. NOT (Logical Complement) Format 3 Opcode 3=64

```
(S):= NOT (S)
```

The TOS is popped from the stack, each bit is complemented and the result is pushed onto the stack.
Indicators = CCA

105. XOR (Exclusive OR) Format 3 Opcode 3=66

(S-1):=(S-1) XOR (S)
S:= S-1

The top two words in the stack are popped from the stack, exclusive ORed together and the result is pushed onto the stack.
Indicators = CCA

107. ANDI (And Immediate) Format 5 Opcode 2=37

(S):= (S) AND CIR(24:8)

The immediate quantity CIR(24:8) is expanded to 32 bits with leading zeros and ANDed with the TOS. The result replaces the TOS.
Indicators = CCA

107.1 XORI (Exclusive OR Immediate) Format 5 Opcode2=36
(S):= (S) XOR CIR(24:8)

The immediate quantity CIR(24:8) is expanded to 32 bits with leading zeros and XORed with the TOS. The result replaces the TOS.
Indicators = CCA

107.2 ORI (Or Immediate) Format 5 Opcode 2=35

(S) := (S) OR CIR(24:8)

The immediate quantity CIR(24:8) is expanded to 32 bits with leading zeros and ORed with the TOS. The result replaces the TOS.
Indicators = CCA

6.2.4 Program Control and Branch Instructions

108. BR (Branch) Format 6 Opcode 2=30

```
XX:= If indexed then X else 0
If P-relative then
Begin
     If direct then P:= E else
     Begin
          P:= P+ Offset + XX + (P+ Offset + XX)
     End
```

```
        End Else
        If not indirect then Branch Violation else
        Begin
             PTR:= BASE+ Offset + XX
             If (PTR)(0:1) = 0 then P:=PB+(PTR)(2:30) else
             Begin
                  P:=Evaluation of Ext. Program Pointer at PTR
                  Decrement Use Count for old segment
                  Increment Use Count for new segment
             End
        End
```

In all cases preindexing is performed. P-relative indirect branches use self-relative pointers while DB, DL, SB, Q, and S-relative indirect branches use either local or external program pointers like those stored in the STT. The uncallable bit in local program pointers is ignored. It is not possible to branch from an unprivileged procedure to a privileged callable procedure. Unconditional P-relative branches can also be performed with the BCC instruction. Indicators Unaffected

108.1 BRSI (Branch Stack Indirect) Format 4 Opcode 3=37

This instruction is a faster implementation of BR S-0,I. In addition, this instruction pops the indirect pointer from the stack. SEE APPENDIX G.

109. BCC (Branch on Condition Code) Format 6 Opcode 2=31
BCCS Format 1 Opcode 1=6

```
        If format 1 then CCF:= CIR (20:3)
        If format 6 then CCF:= CIR (9:3)
        If CC=CCG and CCF(0:1)=1 then P:=E else
        Begin
             If CC=CCE and CCF(1:1)=1 then P:=E else
             Begin
                  If CC=CCL and CCF(2:1)=1 then P:=E
             End
        End
```

In the 16-bit version, the offset field is interpreted as a halfword offset. The sign of the offset is indicated by CIR(23:1) or CIR(12:1). In both cases 0 indicates + and 1 indicates -. Refer to Appendix B. Control is transferred to location E under the following conditions:

If CCF = 0, Never Branch
1, Branch if CC=CCL
2, Branch if CC=CCE
3, Branch if CC=CCL or CCE
4, Branch if CC=CCG
5, Branch if CC=CCG or CCL
6, Branch if CC=CCG or CCE
7, Always Branch
Indicators Unaffected

110. BCY (Branch on Carry) Format 2 Opcode 2=60
Format 2 Opcode 2=64

Note: In all Format 2 branch instructions CIR(19:1) is an indirect bit, CIR(26:6) is the magnitude of the P-relative halfword offset and the sign of the offset is indicated by CIR(23:1), + is 0, - is 1.

If STATUS (6:1) = 1 then P:=E

If the carry bit in the Status Register is a 1 then control is transferred to E.
Indicators = Carry cleared

111. BNC (Branch on No Carry) Format 2 Opcode 2=61
Format 2 Opcode 2=65

If STATUS (6:1)=0 then P:=E

See note under BCY. If the carry bit in the Status register is a 0 then control is transferred to E.
Indicators = Carry cleared

112. BOV (Branch on Overflow) Format 2 Opcode 2=62
Format 2 Opcode 2=66

If STATUS (7:1)=1 then P:=E

See note under BCY. If the overflow bit in the Status register is a 1 then control is transferred to E.
Indicators = Overflow cleared

113. BNO (Branch on No Overflow) Format 2 Opcode 2=63
Format 2 Opcode 2=67

If STATUS (7:1)=0 then P:=E

See note under BCY. If the overflow bit in the Status register is a 0 then control is transferred to E.
Indicators = Overflow cleared.

114. BRZ (Branch on TOS Zero) Format 2 Opcode 2=70
Format 2 Opcode 2=74

```
If (S)=0 then P:=E
S:=S-1
```

See note under BCY. If the TOS is zero then control is transferred to E. The TOS is deleted.
Indicators Unaffected

114.1 BRE (Branch on TOS Even) Format 2 Opcode 2=71
Format 2 Opcode 2=75

```
If (S)(31:1)=0 then P:=E
S:=S-1
```

See note under BCY. If the TOS is even then control is transfered to E. The TOS is deleted.
Indicators Unaffected

114.2 BRO (Branch on TOS Odd) Format 2 Opcode 2=72
Format 2 Opcode 2=76

```
If (S)(31:1)=1 then P:=E
S:=S-1
```

See note under BCY. If the TOS is odd then control is transfered to E. The TOS is deleted.
Indicators Unaffected

114.3 CRB (Compare Range & Branch) Format 2 Opcode 2=73
Format 2 Opcode 2=77

```
Condition Code:= CCE
If (S-1) < (S-2) < (S) then S:=S-2 else
Begin
     If CIR(26:6)=0 then Check Trap else
     Begin
          S:=S-2
          Condition Code:=CCG
```

```
                P:=E
            End
        End
```

See note under BCY. The integer in the third word in the stack is tested to detemine if it is within the interval defined by the upper bound integer in the TOS and the lower bound integer in the second word in the stack. If the integer in the third word in the stack is not within the range then the branch or trap occurs. The bounds parameters are popped from the stack if the integer is within the range or if the branch is taken. They are not popped for the trap case.
Indicators = If branch taken then CCG else CCE

115. BIR (Branch on Inexact Result) Format 2 Opcode 2=40
Format 2 Opcode 2=44

```
If STATUS (5:1)=1 then
Begin
     STATUS (5:1):=0
     If CIR (26:6)=0 then Flt Result Trap else P:=E
End
```

See note under BCY. If the inexact result bit in the Status register is set then it is cleared and the branch or trap is taken.
Indicators Inexact Result Cleared

116. BUN (Branch on Unused Condition Code) Format 2 Opcode 2=41
Format 2 Opcode 2=45

```
If STATUS (16:2)=3 then P:=E
```

See note under BCY. If both of the condition code bits in the Status register are equal to 1 (this pattern is never set by the hardware but may be set by the operating system) then control is transferred to E.
Indicators Unaffected

117. NOP (No Operation) Format 3 Opcode 3=0

Program and data are unchanged.
Indicators Unaffected

118. STOP (Stop) Format 4 Opcode 3=77

```
If SB # 0 then (SB-1):=S-SB
If CIR(19:1)=0 then HALT else
Begin
     TEMP:=MASK
     MASK:=(S)
     S:=S-1
     OLDSTAT:= STATUS
     STATUS (13:1):=1
     Wait for any unmasked Message bit
     MASK:=TEMP
     STATUS:= OLDSTAT
End
```

If CIR(19:1)=0,the machine halts and manual intervention is required to restart it. If CIR(19:1)=1, the machine goes to sleep. In this case operation is resumed if an unmasked Message bit is received. Prior to halting, the Mask register is saved and replaced with the TOS which is deleted and the Interrupt bit in the Status register is set. When operation resumes, the original Mask and Status are restored. In either case the SB-relative value of S is stord at SB-1 if SB#0 (which is true only at initial power-on). The occurence of a message trap after the machine wakes up depends on the original values of Mask and Status. This is a privileged instruction.
Indicators Unaffected.

118.1 WAIT (Wait) Format 4 Opcode 3=76

```
Debug Pad 5:= CIR(19:1)
While (S) # 0 do
Begin
     (S):= (S)-1
End
```

This instruction causes the CPU to stop for a specified period of time. It takes 1 microsecond each time the TOS is decremented. This instruction is interruptable. The value of CIR (19:1) is transferred to debug pad 5 to control the self-test LED.
Indicators Unaffected

120. PCL (Procedure Call) Format 6 Opcode 2=3

PCLS Format 5 Opcode 2=22

In the 16-bit version, Offset=4*CIR(24:8). In the 32-bit version, Offset=CIR(13:19).

```
If Offset = 0 then
Begin
     TEMP:= (S)
     S:= S-1
End else TEMP:= (PL-Offset)
S:= S+4
(S-3):= X
(S-2):= P+1-PB
(S-1):= STATUS
(S):= S-Q
Q:= S
P:= Evaluation of pointer in TEMP
```

A stack marker is pushed onto the stack. Control is transfered to the location evaluated from the program pointer (either local or external) contained in the TOS if Offset=0 or contained in PL-Offset otherwise. If the pointer is local, the U-bit is ignored. A trap occurs if an attempt is made to transfer in unprivileged mode to a code segment with the U-bit set. Furthermore, if an external pointer is referenced, then the appropriate entry in the STT of the called segment must contain a local pointer. If a privileged user calls an unprivileged segment, the machine will remain in privileged mode. See Appendix C for a flowchart of this instruction.
Indicators Unaffected

121. EXIT (Procedure Exit) Format 5 Opcode 2=23

```
S:= Q
Q:= Q-(S)
STATUS:= (S-1)
Set PB and PL from Code Segment Number in STATUS
P:= PB + (S-2)
X:= (S-3)
S:= S-4-CIR(24:8)
```

This instruction is used to return from routines called with the PCL instruction. A routine executing in unprivileged mode may not EXIT to privileged mode, may not change the

interrupt bit, and may not set the instruction pending bit in the Status register. This instruction restores the values of Q, S, PB, PL, P, X and STATUS. The quantity CIR(24:8) is the number of words occupied by passed parameters. These words are popped from the stack. See Appendix C for a flowchart of this instruction.
Indicators reset with the Status register.

122. SXIT (Subroutine Exit) Format 5 Opcode 2=24

P:= PB + (S)
S:= S-1-CIR(24:8)

This instruction is used to return from subroutines which are called with a LRA, BR combination. Only the PB-relative return address is stacked. The quantity CIR(24:8) is the number of words occupied by passed parameters. These words are popped from the stack.
Indicators Unaffected

123. IXIT (Interrupt Exit) Format 4 Opcode 3=74

See Appendix C for a flowchart of this instruction.

This instruction is used to exit from internal traps which are always handled on the ICS, interrupt service routines, and the dispatcher. SEE APPENDIX G. Several alternative routes are possible depending on the state of certain flags. These possibilities include:

1) Dispatcher exit to a new process. This occurs if the dispatcher flag is a one when IXIT is executed.

2) Return to interrupted process. This occurs after the last interrupt has been serviced and (Q) = 0. The dispatcher is not automatically entered.

3) Same as (2) except a DISP instruction was executed while pseudo-disabled (i.e. an attempt was made to enter the dispatcher).

4) Return to an interrupted interrupt service routine. This occurs when a service routine is completed and (Q)(0:1)=0 and (Q)(1:31)#0.

5) Return to interrupted dispatcher. This occurs when (Q)(0:1)=1 (set by the microcode interrupt handler) and (QI)(0:1)=0(i.e. Q is not equal to QI).

6) Same as (5) except a DISP instruction was executed while pseudo-disabled.

7) Start dispatcher instead of returning to interrupted process. This occurs when (QI)(0:1)=1 (set by executing a DISP instruction).

8) Restart dispatcher instead of returning to interrupted dispatcher. This occurs when (QI)(0:1)=1 (set by executing a DISP instruction).

Note: (7) and (8) are treated as identical cases.

IXIT assumes it is executing on the ICS with a stack marker at Q, Q-1, Q-2, and Q-3, a pointer to the global data segment at Q+1, the previous interrupt mask at Q+2 and the device number of the interrupting device at Q+3. The entries in the Task Control Block are used when dispatching a new process and when returning to a interrupted process. This is a privileged instruction.
Indicators Unaffected

124. DISP (Dispatch) Format 4 Opcode 3=16

```
If ICS FLAG = 1 or (QI-4) # 0 then (QI)(0:1) = 1 else
Begin
     S:= S+4
     (S-3):= X
     (S-2):= P+1-PB
     (S-1):= STATUS
     (S):= S-Q
     (SB-1):= S-SB
     Q:= QI
     SL:= SLI
     ICS FLAG:= 1
     S:= Q + 3
     SB:= Q-64
     (QI):=0
     Set DB and DL from Data Segment Pointer in (Q+1)
     (SB):=(Q+1)
     Set PB and PL from Code Segment Number in (Q-1)
```

```
            DISP FLAG:=1
            STATUS:=(Q-1)
            X:=(Q-3)
            P:= PB + (Q-2)
       End
```

This instruction is used to enter the dispatcher directly from a privileged program or from the ICS after servicing all pending interrupts. Execution of this instruction is the only way to enter the dispatcher. This is a privileged instruction.
Indicators = If dispatcher entered then CCE (prior to pushing stack marker) else CCG.

125. PSDB (Pseudo Interrupt Disable) Format 4 Opcode 3=15

```
       (QI-4):= (QI-4) + 1
```

The Dispatcher Disable Count is incremented. The dispatching of new processes is inhibited while this count is not zero. This is a privileged instruction.
Indicators Unaffected

126. PSEB (Pseudo Interrupt Enable) Format 4 Opcode 3=17

```
       If (QI-4) = 0 then System Error Trap
       (QI-4):=(QI-4)-1
       If (QI-4) = 0 then
       Begin
            If (QI)(0:1) = 1 then
            Begin
                 If DISP FLAG=1 then
                 Begin
                      (QI):=0
                      Abort dispatcher & restart
                 End else Execute DISP Instruction
            End
       End
```

If the Dispatcher Disable Count, (QI-4), is zero when this instruction executes then a system error trap occurs. Otherwise the count is decremented. If it now equals zero, all "pseudo-disables" have been removed, and it is now permissible to dispatch a new process. If (QI)(0:1)=1 (it may have been set by executing a DISP instruction while on

the ICS or while pseudo-disabled) then the dispatcher is aborted and restarted if DISP FLAG=1 or the DISP instruction is executed if DISP FLAG=0. If the new value of the dispatcher disable count is not zero or if there have been no dispatch requests, QI(0:1)=0, then the condition code is set to CCL and the next instruction is executed. This is a privileged instruction.
Indicators = CCL or set by DISP instruction

127. SOL (Start of Line) Format 5 Opcode 2=0

If STATUS(1:1)=1 then Start of Line Trap

This debugging aid can be the first instruction of the series of machine instructions corresponding to a line of a high level language program. If the Start of Line Bit is set then a Start of Line Trap occurs.
Indicators Unaffected.

127.5 SOLC (Start of Line Check) Format 5 Opcode 2=31

```
If Q = S then Bounds Violation Trap else
Begin
     (Q+1):= P-PB
     If (SB+1) # 0 then Start of Line Check Trap
End
```

This debugging aid stores P-PB at Q+1 and then traps if the content of SB+1 (word) is not zero. The meaning of the word at SB+1 is defined by the operating system. No bounds tests are made on SB+1. Furthermore Q+1 must not be the same word as SB+1.
Indicators Unaffected

128. SOP (Start of Procedure) Format 5 Opcode 2=1

```
STATUS(0:5) := 0
If Overall Debug Bit = 1 then Start of Procedure Trap
```

This debugging aid can be the first machine instruction in a procedure (after all local variables have been allocated). The debug bits in the Status register are cleared and the overall debug bit is tested. If this bit is a 1 then a Start of Procedure Trap occurs.
Indicators Unaffected

129. EOP (End of Procedure) Format 4 Opcode 3=44

If STATUS(2:1)=1 then End of Procedure Trap

This debugging aid can be placed just before an EXIT or IXIT instruction. An End of Procedure Trap occurs if the End of Procedure Bit in the Status register is a 1.
Indicators Unaffected

130. SOS (Start of Subroutine) Format 5 Opcode 2=16

If STATUS(3:1)=1 then Start of Subroutine Trap

This debugging aid can be the first instruction in a subroutine. If the Start of Subroutine Bit is 1 then a Start of Subroutine Trap occurs.
Indicators Unaffected

131. EOS (End of Subroutine) Format 4 Opcode 3=4

If STATUS(4:1)=1 then End of Subroutine Trap

This debugging aid can be placed just before an SXIT instruction. If the End of Subroutine Bit is 1 then an End of Subroutine Trap occurs.
Indicators Unaffected

6.2.5 Move, Scan, and String Instructions

133. MVB (Move Bytes) Format 2 Opcode 2=51

```
X:=0
SDEC:= CIR(30:2)
D:= If CIR(19:1) = 0 then 1 else -1
If CIR (29:1) = 0 then
Begin
     SBA:= Evaluation of Local Pgm Pointer in (S-2)
End Else
Begin
     SBA:= Evaluation of Data Segment Pointer in (S-2)
End
TBA:= Evaluation of Data Segment Pointer in (S-1)
While (S) > D*X do
Begin
```

```
            (TBA):= (SBA)
            TBA := TBA+D
            SBA := SBA+D
            X:=X+D
        End
        (S):=(S)-D*X
        S:= S-SDEC
```

This instruction expects a byte count in the TOS, a target area pointer in the second word in the stack and a source area pointer in the third word in the stack. The source area pointer can be a local program pointer, in which case the uncallable bit is ignored. If the source area pointer is an external program pointer then an Unexpected Pointer Type Trap will occur. If the direction bit, CIR(19:1), is zero, bytes are moved from progressively increasing byte addresses in the source area to the target area. If the direction bit is one, bytes are moved from progressively decreasing byte addresses. SBA is the source byte address and TBA is the target byte address. The three parameters on the stack cannot be part of either the source area or the target area. A bounds violation will cause premature termination of the instruction and a Bounds Violation Trap. The number of words specified in the SDEC field of the instruction are popped from the stack. Where possible this instruction reads and writes full word quantities instead of individual bytes. (E.g. if D=1 and SBA(30:2)=0 and TBA(30:2)=0 and (S)(30:2)=0 or if D=-1 and SBA(30:2)=3 and TBA(30:2)=3 and (S)(30:2)=0)
Indicators Unaffected

133.1 PMVA (Priv Move Bytes using Absolute Addresses) Fmt 2 Op2=50

```
        X:= 0
        SDEC:= CIR(30:2)
        D:= If CIR(19:1) = 0 then 1 else -1
        TBA:= (S-1)
        SBA:= (S-2)
        While (S) > D*X do
        Begin
            (TBA):= (SBA)
            TBA:= TBA+D
            SBA:= SBA+D
            X:= X+D
        End
```

```
(S):= (S)-D*X
S:= S-SDEC
```

This instruction expects a byte count in the TOS, a target absolute address in the second word in the stack and a source area absolute address in the third word in the stack. The direction bit, CIR(19:1), is used as in MVB. Bytes are moved from the source byte address, SBA, to the target byte address, TBA, until the byte count is equaled. The three parameters cannot be part of the source or target area. No bounds checking is performed. The number of words specified in the SDEC field of the instruction are popped from the stack. Where possible this instruction reads and writes full word quantities instead of bytes (see MVB instruction for details).
Indicators Unaffected

135. SCW (Scan While) Format 2 Opcode 2=52

```
X:=0
SDEC:= CIR(30:2)
D:= If CIR(19:1) = 0 then 1 else -1
If CIR(29:1)= 0 then
Begin
     SBA:= Evaluation of Local Pgm Pointer in (S-2)
End Else
Begin
     SBA:= Evaluation of Data Segment Pointer in (S-2)
End
SCANNING:=TRUE
While SCANNING AND (S)>D*X do
Begin
     (S-1)(0:8):=(SBA)
     (S-1)(8:8):=(SBA) XOR (S-1)(16:8)
     If (SBA)=(S-1)(24:8) then
     Begin
          SBA:=SBA+D
          X:=X+D
     End else SCANNING := FALSE
End
(S):=(S)-D*X
S:= S-SDEC
```

This instruction expects a limit byte count in the TOS, a terminal character in the third byte in the second word in

the stack, a test character in the fourth byte in the second word in the stack, and a source area pointer in the third word in the stack. If the address is valid and the limit byte count is not exceeded, bytes are scanned until the source string presents a character that is different from the test character. The terminating character and the XOR of the terminating character and the terminal character are deposited in the first and second bytes respectively in the second word in the stack. If the direction bit, CIR(19:1), is zero, bytes are scanned in progressively increasing addresses. If the direction bit is one, bytes are scanned in progressively decreasing addresses. An invalid address causes a Bounds Violation Trap. The three parameters on the stack cannot be part of the source area. The number of words specified in the SDEC field of the instruction are popped from the stack.
Indicators = CCB on the last character scanned only if the original TOS>0

136. SCU (Scan Until) Format 2 Opcode 2=53

```
X:=0
SDEC:= CIR(30:2)
D:= If CIR(19:1) = 0 then 1 else -1
If CIR(29:1) = 0 then
Begin
     SBA:= Evaluation of Local Pgm Pointer in (S-2)
End Else
Begin
     SBA:= Evaluation of Data Segment Pointer in (S-2)
End
SCANNING := TRUE
While SCANNING AND S>D*X do
Begin
     (S-1)(0:8):=(SBA)
     (S-1)(8:8):=(SBA) XOR (S-1)(16:8)
     If (SBA)#(S-1)(24:8) AND (SBA)#(S-1)(16:8) then
     Begin
          SBA:=SBA+D
          X:=X+D
     End else SCANNING := FALSE
End
(S):=(S)-D*X
S:= S-SDEC
```

This instruction expects a limit byte count in the TOS, a terminal character in the third byte in the second word in the stack, a test character in the fourth byte in the second word in the stack and a source area pointer in the third word in the stack. If the address is valid and the limit byte count is not exceeded, bytes are scanned until either the terminal or the test character is encountered. The terminating character and the XOR of the terminating character and the terminal character are deposited in the first and second bytes respectively in the second word in the stack. The direction bit, CIR(19:1), is used as in SCW. An invalid address causes a Bounds Violation Trap. The three parameters on the stack cannot be part of the source area. The number of words specified in the SDEC field of the instruction are popped from the stack.
Indicators = CCB on last character scanned only if the original TOS>0

137. CMPB (Compare Bytes) Format 2 Opcode 2=54

```
X:=0
SDEC:= CIR(30:2)
D:= If CIR(19:1) = 0 then 1 else -1
If CIR(29:1)=0 then
Begin
     SBA:= Evaluation of Local Pgm Pointer in (S-2)
End Else
Begin
     SBA:= Evaluation of Data Segment pointer in (S-2)
End
TBA:= Evaluation of Data Segment Pointer in (S-1)
While (S)>D*X AND (SBA)=(TBA) do
Begin
     SBA:= SBA + D
     TBA:= TBA + D
     X:= X + D
End
(S):=(S)-D*X
S:= S-SDEC
```

This instruction expects a byte count in the TOS, a target area pointer in the second word in the stack and a source area pointer in the third word in the stack. If the direction bit, CIR(19:1), is zero, bytes from the source area are compared with bytes from the target area from

progressively increasing addresses. If the direction bit is one, bytes are compared from progressively decreasing addresses. The instruction terminates when a comparison fails or when the byte count in the TOS is reached or when an invalid address causes a Bounds Violation Trap. The Condition Code bits are set to CCE if the original TOS is less than or equal to 0. The three words on the stack cannot be part of either the source area or the target area. The number of words specified in the SDEC field of the instruction are popped from the stack.
Indicators = If byte count termination then CCE else
If (final source byte)>(final target byte)
then CCG else CCL

137.1 TRAN (Translate) Fmt 2 Op2=56

```
X:= 0
SDEC:= CIR(30:2)
D:= If CIR(19:1) = 0 then 1 else -1
If CIR(29:1) = 0 then
Begin
     TBLBA:= Evaluation of Local Pgm Pointer in (S-2)
End else
Begin
     TBLBA:= Evaluation of Data Segment Ptr in (S-2)
End
TBA:= Evaluation of Data Segment Pointer in (S-1)
While (S) > D*X do
Begin
     (TBA):= (TBLBA+(TBA))
     TBA:= TBA + D
     X:= X + D
End
(S):= (S) - D*X
S:= S - SDEC
```

This instruction expects a byte count in the TOS, a pointer to a string of bytes to be translated in the second word on the stack, and a pointer to a 256-byte conversion table [which is not allowed to cross segment or page boundaries] in the third word on the stack. Bytes are read from the string to be translated (using the direction bit as in SCW) one at a time and the byte value is used as an offset into the table. The selected table value replaces the original byte in the string and the address is advanced to the next

byte. The three stacked parameters cannot be part of the string or the conversion table. A bounds violation causes premature termination of the instruction and a Bounds Violation Trap. The number of words specified in the SDEC field of the instruction are popped from the stack. Indicators Unaffected

137.2 FILB (Fill Bytes) Fmt 2 Op2=57

```
X:= 0
SDEC:= CIR(30:2)
If (S) > 0 then
Begin
     TBA:= Evaluation of Data Segment Pointer in (S-1)
     SOURCEBYTE:= (S-2)(24:8)
     While (S) > X do
     Begin
          (TBA):= SOURCEBYTE
          TBA:= TBA+1
          X:= X+1
     End
     (S):= (S)-X
End
S:= S-SDEC
```

This instruction expects a byte count in the TOS, a target area pointer in the second word in the stack and a source byte value in the right most byte of the third word in the stack. The indicated target area is filled with the specified byte value. The three parameters cannot be part of the target area. The number of words specified in the SDEC field of the instruction are popped from the stack. Indicators Unaffected

137.3 SVAL (String Validate) Fmt 2 Op2=26

```
(Fetch source parameters)

MAXLEN:= (S-3)
(S-2) (30:2):= 0
PTR:= (S-2)
FIRST:= (S-1)
LEN:= (S)
If CIR(29:1)=1 then
Begin
```

```
        STRINGADR:= Evaluation of PTR as Data Seg Ptr
    End else
    Begin
        STRINGADR:= Evaluation of PTR as Local Pgm Ptr
    End
    CURLEN:= (STRINGADR)

    (Validate source parameters)

    If MAXLEN < 0 then String Trap
    If (LEN < 0) and (CIR(19:1)=0) then String Trap
    If FIRST <= 0 then String Trap
    If (LEN>=0) and ((FIRST+LEN-1)>MAXLEN) then String Trap
    If CURLEN < 0 then String Trap
    If CURLEN > MAXLEN then String Trap
    If FIRST > CURLEN+1 then String Trap

    (Compute actual length)

    If (LEN < 0) or (LEN > (CURLEN - FIRST + 1)) then LEN:=
    CURLEN - FIRST + 1

    (Return actual length if option bit set)

    If CIR(27:1) = 1 then (S):= LEN
```

This instruction expects a string descriptor in the top 4 words of the stack. The two LSB's of the pointer in the descriptor are set to zero to force word alignment. The descriptor and current length word are subjected to the indicated checks. If any check fails, a String Trap will occur. If CIR(27:1)=1, the length word is updated to contain the actual length. (NB: the checks are not necessarily made in the order given.)
Indicators Unaffected

137.4 SLD (String Load) Fmt 2 Op2=25

```
    Fetch and validate source parameters as in SVAL.
    Compute actual length as in SVAL
    (Compute size of string temp, allocate space)

    TEMPADR:= S - 15 bytes
    NWORDS:= ((LEN + 3) div 4) + 1
    NEWS:= TEMPADR + (NWORDS*4 + 19) bytes
```

If NEWS > SL then Stack Overflow Trap else S:= NEWS

(Fill in string temp length size word, and descriptor)

(TEMPADR):= LEN
(S-4):= NWORDS
(S-3):= LEN
(S-2):= SB-relative pointer to TEMPADR
(S-1):= 1
(S):= -1

(Use MVB to fill in data)

S:= S+3
(S-2):= SPTR+FIRST+3
(S-1):= SB-relative pointer to TEMPADR + 4
(S):= LEN
Execute MVB Instruction with CIR(19:1) = 0,
CIR(29:1) unchanged, and CIR(30:2) = 3

This instruction expects a string descriptor in the top 4 words of the stack. The descriptor and current length word are subjected to the usual checks. If these succeed, the descriptor is replaced by a string temp containing the source substring. The source string variable is not modified. Adding FIRST+3 to SPTR can cause a String Trap (Parameter=8) as described in Section 5.2.2. Note also that the original content of X is destroyed early in the instruction and that the Trace Variable Trap parameter points to the first character moved onto the stack.
Indicators Unaffected

137.5 SAS (String Assign) Fmt 2 Op2=30

Fetch and validate source parameters as in SVAL
Compute actual length as in SVAL
(Source parameters are referred to as
SMAXLEN, SPTR, SFIRST, SLEN, SSTRINGADR, SCURLEN)

(Fetch target parameters)
If CIR(31:1)=1 then
Begin
TDESCADR:= SSTRINGADR - 4 words
If (TDESCADR < Q) or (TDESCADR > S-7) then Bounds Violation Trap

```
            TMAXLEN:= (TDESCADR)
            TPTR:= (TDESCADR+1)
            TFIRST:= (TDESCADR+2)
            TLEN:= (TDESCADR+3)
      End else
      Begin
            TMAXLEN:= (S-7)
            TPTR:= (S-6)
            TFIRST:= (S-5)
            TLEN:= (S-4)
      End
      TPTR(30:2):= 0
      TSTRINGADR:= Evaluation of TPTR as Data Segment Pointer
      TCURLEN:= (TSTRINGADR)

      (Validate target parameters)

      Apply the checks described under SVAL, except that
      CIR(30:1) specifies whether TLEN may be < 0

      (Determine move parameters and new target length)

      If TLEN < 0 then
      Begin
            TCURLEN:= TFIRST + SLEN - 1
            If TCURLEN > TMAXLEN then String Trap
            MVLEN:= SLEN
      End else
      Begin
            TCURLEN:= max(TCURLEN, TFIRST + TLEN - 1)
            MVLEN:= min(SLEN,TLEN)
      End
      If (TSTRINGADR = SSTRINGADR) and (TFIRST > SFIRST) then
      Begin
            CC:= CCL
            MOVESOURCE:= SPTR+SFIRST+MVLEN+2 bytes
            MOVETARGET:= TPTR+TFIRST+MVLEN+2 bytes
      End else
      Begin
            CC:= CCG
            MOVESOURCE:= SPTR+SFIRST+3 bytes
            MOVETARGET:= TPTR+TFIRST+3 bytes
      End

      (Determine blank fill parameters)
```

```
If CIR(28:1)=1 then
Begin
     If TLEN > MVLEN then
     Begin
          FILLTARGET:= TPTR+TFIRST+MVLEN+3
          FILLLEN:= TLEN - MVLEN
     End else FILLLEN:= 0
End

(Pop descriptors and stack blankfill parameters)

If CIR(31:1)=1 then S:=S-4 else S:= S-8
If CIR(28:1)=1 then
Begin
     S:= S + 3
     (S-2):= 32
     (S-1):= FILLTARGET
     (S):= FILLLEN
End

(Update target length word and do move)

(TSTRINGADR):= TCURLEN
S:= S + 3
(S-2):= MOVESOURCE
(S-1):= MOVETARGET
(S):= MVLEN
Execute MVB Instruction with CIR(19:1) = 1 if CCL or 0
if CCG, CIR(29:1) unchanged, and CIR(30:2) = 3
```

This instruction expects a source string descriptor in the top 4 words of the stack. If CIR(31:1)=0, the target descriptor is found in the next 4 words of the stack; if CIR(31:1)=1, the source must be a string temp, and the target descriptor is found immediately below it on the stack. (The target descriptor must be between Q and S.) Both descriptors and current length words are subjected to the usual checks. If these succeed, the target current length word is updated and the proper substring of the source is moved into the proper substring of the target. (The Trace Variable Trap parameter points to the first character moved into the substring of the target.) If CIR(31:1)=0, both descriptors are deleted, otherwise only the source descriptor is deleted. If CIR(28:1)=1,

parameters for a following FILB instruction are pushed onto the stack, so that blank fill can be applied when the target substring may be longer than the source. The source string variable is not modified. Adding a quantity to SPTR or TPTR can cause a String Trap (Parameter=8) as described in Section 5.2.2. Note also that the original content of X is destroyed early in the instruction. SEE APPENDIX G.
Indicators: CC is set according to move direction used

137.6 SCON (String Concatenate) Fmt 2 Op2=27

```
Fetch and validate source parameters as in SVAL
Compute actual length as in SVAL
(Source parameters are referred to as
 SMAXLEN, SPTR, SFIRST, SLEN, SSTRINGADR, SCURLEN)

Fetch and validate target parameters as in SAS

If (TSTRINGADR < Q) or (TSTRINGADR > S-4) then Bounds
Violation Trap

(Compute moveaddresses and new temp length)

If TLEN >= 0 then TCURLEN:=min(TCURLEN, TFIRST+TLEN -1)
MOVESOURCE:= SPTR+SFIRST+3 bytes
MOVETARGET:= TPTR+TCURLEN+4 bytes
TMAXLEN:= TCURLEN + SLEN

(Allocate space for temp)

NWORDS:= ((TMAXLEN + 3) div 4) + 1
NEWS:= TSTRINGADR + (NWORDS*4 + 19) bytes
If NEWS < S then
Begin
     NWORDS:= (S - TSTRINGADR - 19 bytes) div 4
End else
Begin
     If NEWS>SL then Stk Overflow Trap else S:=NEWS
End

(Fill in string temp length, size word, and descriptor)

(TSTRINGADR):= TMAXLEN
(S-4):= NWORDS
(S-3):= TMAXLEN
```

(S-2):= TPTR
(S-1):= TFIRST
(S):= -1

(Move source data)

S:= S+3
(S-2):= MOVESOURCE
(S-1):= MOVETARGET
(S):= SLEN
Execute MVB Instruction with CIR(19:1) = 0,
CIR(29:1) unchanged, and CIR(30:2) = 3

This instruction can best be understood by considering two cases:

1) If CIR(31:1) = 1, two string temps are expected on the stack. They are replaced by a single temp whose value is the concatenation of the values of the two temps. The new temp occupies the same space as the two old temps together did, hence it contains at least 6 words of wasted space.

2) If CIR(31:1) = 0, a source string descriptor is expected in the top 4 words of the stack, and a target string temp is expected just below it. They are replaced by a single string temp whose value is the concatenation of the temp value and source substring. The new temp occupies at least as much space as the old temp and source descriptor together.

The source string variable is not modified. Adding a quantity to SPTR or TPTR can cause a String Trap (Parameter=8) as described in section 5.2.2. Note also that the original content of X is destroyed early in the instruction and that the Trace Variable Trap parameter points to the first character moved onto the stack. Indicators Unaffected

6.2.6 I/O and Interrupt Instructions

138. INT (Interrupts) Format 4 Opcode 3=11

STATUS (13:1) := CIR(19:1)

Message Register Interrupts are enabled or disabled by setting the interrupt bit in the Status register equal to CIR(19:1). This is a privileged instruction.
Indicators Unaffected

140. PMPB (Push MPB Registers) Format 2 Opcode 2=31

```
If CIR(31:1) = 1 then S:=S+1; (S):= MASK
If CIR(30:1)=1 then S:= S+1; (S):= MSG
If CIR (29:1)=1 then
Begin
     S:= S+1
     S(0:29):= 0
     S(29:3):= MASK(3:3)
End
```

This instruction pushes the Mask and Message registers and the (right-justified) CPU channel number onto the stack based on CIR(29:3).
Indicators Unaffected

141. SMPB (Set MPB Registers) Format 2 Opcode 2=32

```
If CIR(29:1)=1 then MASK(3:3):= (S)(29:3); S:= S-1
If CIR(30:1)=1 then
Begin
     I:= 0
     While I<32 do
     Begin
          If (S)(I:1)=1 then MSG (I:1):= 0
          I:= I+1
     End
     S:= S-1
End
If CIR(31:1)=1 then
Begin
     MASK (0:3):= (S)(0:3)
     MASK (6:26):= (S)(6:26)
     S:= S-1
End
```

Based on CIR(29:3), this instruction: 1) sets the CPU channel number (MUST NOT be 0) from the three least significant bits of the TOS and pops the stack, 2)clears individual bits in the Message register if the corresponding

bits in the (new) TOS are set and pops the stack, and 3)stores the (new) TOS in the Mask register (bits 3-5 unaffected) and pops the stack. This is a privileged instruction.
Indicators Unaffected

142. IOW (I/O Write) Format 5 Opcode 2=17

CIR(24:8) contains the I/O operation code. (The current I/O hardware ignores the MSB of the I/O opcode.) For write operations, the TOS contains the data word which is written to the I/O channel and the second word in the stack contains the channel number, bits 26-28, peripheral address, bits 29-31, and the "computed" interface control field, bits 22-25. Bits 0-21 must be zero. Both words are popped if the write is successful. More details are available in the I/O ERS. This is a privileged instruction.
Indicators Unaffected

142.1 IOR (I/O Read) Format 5 Opcode 2=20

For read operations, the TOS contains the channel number, peripheral address, interface control word (see IOW). This word is popped from the stack and the data word read from the I/O channel becomes the new TOS. This is a privileged instruction. (This instruction writes to the designated channel and waits for that channel to write back. The X register is used for the data word of the first write.)
Indicators Unaffected

142.2 IOC (I/O Control) Format 5 Opcode 2=21

For control operations, the TOS is the same as for reads. This word is popped from the stack. This is a privileged instruction.
Indicators Unaffected

6.2.7 Special Instructions

150. LLSH (Linked List Search) Format 2 Opcode 2=36

```
TEST:=(S-1)
CCF:=CIR(29:3)
FLAG:= TRUE
While X>0 AND (S) # 0 AND FLAG = TRUE do
Begin
```

```
                TARGET:=((S)+(S-2))
                If TARGET>TEST AND CCF(0:1)=1 then
                (S):=((S));X:=X-1 else
                Begin
                     If TARGET=TEST AND CCF(1:1)=1 then
                     (S):=((S));X:=X-1 else
                     Begin
                          If TARGET<TEST AND CCF(2:1)=1 then
                          (S):=((S));X:=X-1 else FLAG:= FALSE
                     End
                End
           End
```

The TOS contains an absolute address which points to a link word in a linked list. Each link word contains the absolute address of the next link in the list. The second word in the stack is a test word. The third word in the stack is a byte offset which indicates the position, relative to each link, of a target word. The Index register contains a count. If, at each step, the index register is less than or equal to zero (tested first), or if the TOS/link address is zero, or if the relationship between the test number and the target number does not match the desired relationship specified in CIR (29:3), then the instruction terminates. Otherwise, the TOS is replaced by the address of the next link, the index register is decremented and the instruction repeats. This is a privileged instruction.

Indicators = CCL if terminated on X<0
CCE if terminated on target, test comparison
CCG if terminated on (S)=0

150.1 LINS (List Insert) Format 4 Opcode 3=72

```
           If (S-1)#0 then
           Begin
                FLAG := FALSE
                If (S)=0 then
                Begin
                     (S):=(S-1)
                     FLAG := TRUE
                End
                ((S-1)):=(S)
                BKW:=((S)+(S-2))
                ((S)+(S-2)):=(S-1)+(S-2)
                If not FLAG then
```

```
        Begin
             If BKW # 0 then (BKW-(S-2)):=(S-1)
             ((S-1)+(S-2)):=BKW
        End
   End
```

The TOS contains the absolute address of the "forward" link word of an entry in a doubly linked list. The second word in the stack contains the absolute address of the "forward" link word of a new entry for the list. The third word in the stack contains the 2's complement byte offset from the "forward" to the "backward" link word in each entry. Each "forward" ("backward") link word in the list contains the absolute address of the next "forward" ("backward") link word in the list. Zero is interpreted as the nil pointer. This instruction inserts the new entry prior to the indicated existing entry in the list. If the existing list is empty (indicated by TOS=0) then the new entry is circularly linked to itself and the TOS is set to the value of the second word in the stack. This is a privileged instruction.
Indicators Unaffected

150.2 LDEL (List Delete) Format 4 Opcode 3=73

```
If (S)#0 then
Begin
     FWD := ((S))
     BKW := ((S)+(S-1))
     If BKW#0 then (BKW-(S-1)) := FWD
     If FWD#0 then (FWD+(S-1)) := BKW
End
```

The TOS contains the absolute address of the "forward" link word of an entry in a doubly linked list. The second word in the stack contains the 2's complement byte offset from the "forward" to the "backward" link word in each entry. The list is assumed to be linked as described in LINS. This instruction deletes the indicated entry from the list. This is a privileged instruction.
Indicators = If no entries left in list then CCE else CCG

152. RCS (Read Control Store) Format 4 Opcode 3=31

```
CSA:= X(18:14)
```

```
S:= S+2
(S-1)(0:6):= CSD(0:6)
(S-1)(6:26):= 0
(S):= CSD(6:32)
```

The 14 least significant bits of the index register are interpreted as a control store address, CSA. The content of this location, CSD, is pushed onto the stack with the six most significant bits left justified in the second word on the stack and the remaining 32 bits in the TOS.
Indicators Unaffected

153. SEML (Semaphore Load) Format 6 Opcode 2=33

```
S:= S+1
(S):= (E)       Uninterruptable by
(E):= -1        other processors
```

The content of E is pushed onto the stack and a 2's complement -1 is stored in location E in one uninterruptable operation.
Indicators = CCA on the new TOS

154. PSLA (Priv Semaphore Load from Abs Address) Fmt4 Op3=63

```
E:= (S)+ If CIR(19:1)=1 then X else 0
(S):= (E)       Uninterruptable by
(E):= -1        other processors
```

The TOS contains an absolute address. The content of this address (indexed if CIR(19:1)=1) replaces the TOS and a 2's complement -1 is stored in this (indexed) address in one uninterruptable operation. This instruction should not be used to access memory between Q and SL in an active stack segment. This is a privileged instruction.
Indicators = CCA

155. CPAA (Convert Pointer to Absolute Address) Fmt4 Op3=66

```
DSPTR:= (S)
S:= S-1
(S+1):= Evaluation of DSPTR
S:= S+1
```

The data segment pointer contained in the TOS is replaced by

the absolute address pointed to by the pointer (indexed if CIR(19:1)=1). This is a privileged instruction.
Indicators Unaffected

155.1 VPTR (Validate Pointer) Format 4 Opcode 3=71

```
If (S-1)(31:1)=1 AND STAT(14:1)=0 then CC:=CCL else
Begin
     CC:=CCG
     If (S)(0:2)=2 then
     Begin
          Offset:=(S)(2:30)
          If CIR(19:1)=1 then Offset:=Offset+X
          If Offset < 0 then CC:= CCL
          If Offset > DL-DB then CC:=CCL
     End
     If (S)(0:2)=3 then
     Begin
          Offset:=(S)(2:30)
          If CIR(19:1)=1 then Offset:=Offset+X
          If Offset < 0 then CC:= CCL
          If Offset > (S-2) then CC:=CCL
     End
     If (S)(0:1)=0 then
     Begin
          CC:=CCG
          If (S) is invalid Ext. Data Seg. Ptr. then
          CC:=CCL
          If (S-1)(31:1)=0 AND Data Seg. Mode=1 then
          CC:=CCL
          If (S-1)(30:1)=1 AND Seg. Write Bit=0 then
          CC:=CCL
          If segment is absent then CCE
     End
End
S:=S-1
```

This instruction expects a data segment pointer on the TOS, two mode bits (WRITE=1, PRIV=1) in the least significant bits of the second word in the stack and an SB-relative byte offset in the third word in the stack. Global data segment pointers (possibly indexed) are bounds tested against DL-DB. Stack segment pointers (possibly indexed) are bounds tested against the offset in (S-2) instead of S-SB. External data segment pointers (possibly indexed) are tested against the

information (including bounds, write bit, and mode bit) in the Data Segment Table. Individual pages of a paged data segment are treated exactly like unpaged segments except that the detection of an absent page table will cause a trap. The data segment pointer is popped from the stack.
Indicators = CCE if valid pointer but segment absent
CCL if invalid pointer
CCG if valid pointer and segment not absent

155.5 CPEP (Convert Pointer to Ext Data Seg Ptr) Fmt4 Op3=67

```
If (S)(0:1)=0 and CIR(19:1)=1 then
Begin
     TEMP:= Evaluation of Data Seg. Ptr. in (S)
     If OFFSET (0:13) # 0 then Pointer Conversion Trap
     (S):= PTR
End
If (S)(0:1)#0 then
Begin
     Offset:=(S)(2:30)
     If CIR(19:1)=1 then Offset:=Offset+X
     If (S)(0:2)=2 then SEG:=Global Segment Number else
     Begin
          Offset:=Offset+Seg Base relative value of SB
          SEG:=Stack Segment Number
     End
     If Offset(0:13)#0 then Pointer Conversion Trap
     (S)(0:1):=0
     (S)(1:12):=SEG
     (S)(13:19):=Offset
End
```

This instruction converts a Stack Segment Pointer or a Global Data Segment Pointer on the TOS into an External Data Segment Pointer. (If the TOS contains an External Data Segment Pointer and if indexing is specified then the pointer is evaluated as in the CPTR instruction.) In the case of Stack Segment Pointers, the segment base relative value of SB is added to the offset. In both cases the segment numbers are obtained from the TCB. Consequently, this instruction should not be used while on the ICS. If the External Data Segment Pointer cannot accommodate the specified offset then a Pointer Conversion trap occurs.
Indicators Unaffected

155.6 CPTR (Convert Pointer) Format 4 Opcode 3=70

```
X:= (S)
If (S-1)(0:1)=0 and CIR(19:1)=1 then
Begin
     TEMP:= Evaluation of Data Seg Ptr. in (S-1)
     If OFFSET(0:13)<>0 then Ptr Conversion Viol.
     S:=S-1
     (S):= PTR

End else
Begin
     Offset:=(S-1)(2:30)+(S)
     If Offset(0:2)<>0 then Ptr Conversion Viol.
     S:=S-1
     (S)(2:30):=Offset
End
```

This instruction makes a new pointer to the data referenced by a data segment pointer in the second word in the stack and a 2's complement byte offset in the TOS which is placed in the Index register. The new pointer replaces the original pointer on the stack and the offset is popped from the stack. If the original pointer is an external data segment pointer and if CIR(19:1)=1 then a complete evaluation of the pointer is performed. If the original pointer is an external data segment pointer and CIR(19:1)=0 or if it is a stack or global data segment pointer then the final "offset" is the sum of the offset in the TOS and the "offset" in the original pointer. (This is intended for paged external data segments.) In all cases, the final pointer is the same type as the original pointer. If the pointer format cannot accomodate the final offset then a Pointer Conversion Trap occurs.
Indicators Unaffected

156. CCI (Convert Condition Code to Integer) Format 4 Op3=3

```
S := S+1
If CC = CCL then (S) := -1
If CC = CCE then (S) := 0
If CC = CCG then (S) := 1
```

The condition code states of CCL, CCE, CCG are used to generate a -1, 0, or 1 that is pushed onto the stack.

Indicators Unaffected

156.5 TCC (Test Condition Code) Format 2 Opcode 2=37

```
S:=S+1
(S):=0
CCF:=CIR(29:3)
If CC=CCG AND CCF(0:1)=1 then (S):=1 else
Begin
     If CC=CCE AND CCF(1:1)=1 then (S):=1 else
     Begin
          If CC=CCL AND CCF(2:1)=1 then (S):=1
     End
End
```

Either a 0 or a 1 is pushed onto the stack as follows:

If CCF = 0, Always push 0
1, Push 1 if CC=CCL
2, Push 1 if CC=CCE
3, Push 1 if CC=CCL or CCE
4, Push 1 if CC=CCG
5, Push 1 if CC=CCG or CCL
6, Push 1 if CC=CCG or CCE
7, Always push 1

Indicators Unaffected

157. LIA (Load Intermediate Address) Format 6 Opcode 2=5
LIAS Format 5 Opcode 2=25

In the 16-bit version, Level=CIR(24:2) and Offset=CIR(26:6). The offset is a positive byte offset. In the 32-bit version, Level=CIR(8:4) and Offset=CIR(13:19). The offset is a byte offset and the sign of the offset is specified by CIR(12:1), 0 for +, and 1 for -.

```
Temp:= 0
D:=Level
S:= S+1
(S)(2:30):= Q-SB
(S)(0:2):= 3
While D>0 do
Begin
     If (S) (0:1)= 0 then Temp:= 1
     (S):= (Evaluation of Pointer in (S) - 4)
     D:= D-1
End
```

(S):=(S)+Offset
If Temp= 1 then convert (S) to an Ext. Data Seg. Ptr.

This instruction is designed to support full block structure and subtasking in high-level languages. The word preceding each stack marker is interpreted as an access link which is either a stack segment pointer or an external data segment pointer to the delta Q entry in some other stack marker. Stack segment pointers are interpreted within the context of the segment in which they are located and not necessarily within the context of the current stack segment. The Level is interpreted as a positive integer specifying which access link in the access chain is to be loaded onto the stack. If this operand is zero, a stack segment pointer to the current stack marker is pushed onto the stack. After the final access link is loaded onto the stack, the offset is added to (subtracted from) it. If any external data segment pointers are encountered in the access chain then the final TOS is converted to the appropriate external data segment pointer if it is not already an external data segment pointer. Encountering a paged external data segment will cause an Unexpected EDS Type trap. An attempt to make an External Data Segment Pointer with an offset requiring more than 19 bits causes a Pointer Conversion Trap. All memory accesses are bounds tested against the limits of the segment containing the access link. The lower bound is always SB. The initial upper bound is Q for the current stack and (SB-1) for all stacks referenced by external data segment pointers. Every time an access link is read, its address becomes the new upper bound. In the event that this instruction is interrupted prior to completion, two extra words are pushed onto the current stack. These words are popped when the instruction is resumed.
Indicators Unaffected.

158. RFC (Read from Channel) Format 4 Opcode 3=56

This instruction reads one word from the specified MPB channel. The 29 least significant bits of the TOS specify the MPB address. This address word remains on the stack and the data word read from the specified channel is pushed onto the stack. If CIR(19:1)=1 then secondary address mode is used, otherwise primary address mode is used. This instruction should be used only when reading from Memory Controllers or when an acceptable protocol has been

established with I/O Processors and other CPU's. Consult the MPB ERS for more details. This is a privileged instruction.
Indicators = CCE for successful transaction
CCG if either MSG (1:1) or MSG (2:1) became set
CCL if both MSG (1:1) and MSG (2:1) became set

159. WTC (Write to Channel) Format 4 Opcode 3=57

This instruction writes one word to the specified MPB channel. The 29 least significant bits of the second word in the stack contain the MPB address. The TOS contains the data word which is written to the specified channel. If CIR(19:1)=1 then secondary address mode is used, otherwise primary address mode is used. If the write is successful and if the MSB of the MPB address is 1 then the CPU will wait until MSG (12:1) becomes set, then clean it and continue. This instruction should be used only when writing to Memory Controllers or when an acceptable protocol has been established with I/O Processors and other CPUs. Consult the MPB ERS for more details. This is a privileged instruction.
Indicators same as for RFC.

159.5 RSC (Read Slave Channel) Format 4 Opcode 3=62

S:=S+2
(S-1):=SAR
(S):=SDR

This instruction pushes the two MPB slave channel registers onto the stack. This instruction, which has no effect on MSG(6:1), must not be used when MSG(6:1)=1. That bit in the Message register must be cleared (as is done by the microcode prior to going to the Message Trap handler) before executing this instruction. This is a privileged instruction.
Indicators Unaffected

160. SMSG (Send Message) Format 4 Opcode 3=60

This instruction sends a 25-bit message to the specified MPB channel. The 29 least significant bits of the TOS contain the desired channel address and message. If CIR(19:1)=1 then the message is treated as a broadcast message (which

goes to all channels simultaneously). Consult the MPB ERS for more details. This is a privileged instruction.
Indicators Unaffected.

160.2 ALM (Address Lockout Mode) Format 4 Opcode 3=42

If CIR(19:1)=1, this instruction enables Address Lockout Mode for the CPU'S MPB interface. This is the default mode. If CIR(19:1)=0, this instruction disables Address Lockout Mode for the CPU'S MPB interface. In this mode, processors on the MPB with lower priority than the CPU are no longer guaranteed minimum MPB access rates. Consult the Memory Controller ERS for more details. This is a privileged instruction.
Indicators Unaffected

163. CHEK (Check) Format 6 Opcode 2=2

If not[CIR(8:5) < (S) < CIR(13:19)] then Check Trap

This instruction causes a trap to STT entry number 2 in code segment 1 unless the 32-bit 2's complement integer in the TOS is within the range defined by the 5-bit and 19-bit positive integers in the instruction.
Indicators Unaffected

164. CNTO (Count Ones) Format 4 Opcode 3=54

```
I:=0
COUNT:=0
While I<32 do
Begin
     If (S)(I:1)=1 then COUNT:=COUNT+1
     I:=I+1
End
If CIR(19:1)=1 then S:=S+1
(S):=COUNT
```

This instruction counts the number of ones in the TOS. If CIR(19:1)=0, this count replaces the TOS. Otherwise, this count is pushed onto the stack.
Indicators Unaffected

165. FLMO (Find Leftmost One) Format 4 Opcode 3=55

```
If (S)=0 then I:=32 else
Begin
     I:=0
     While (S)(I:1)=0 do I:=I+1
End
If CIR(19:1)=1 then S:=S+1
(S):=I
```

This instruction computes the bit position of the leftmost one in the TOS. If the TOS is zero, the result is 32. If CIR(19:1)=0, this result replaces the TOS. Otherwise, this result is pushed onto the stack.
Indicators Unaffected

166. TRY (Try) Format 4 Opcode 3=40

```
If CIR(19:1)=1 then
Begin
     S:= S+3
     (S):= S-Q
     (S-1):= STATUS
     (S-2):= (S-3)
     (S-3):= (SB+2)
     (SB+2):= S-SB
End else
Begin
     TEMP:= SB+(SB+2)
     If Q+4 < TEMP < S then
     Begin
          (SB+2):= (TEMP-3)
          S:= TEMP-4
          (SB+4):= 0
     End else Bounds Violation Trap
End
```

This instruction either creates or deletes a pseudo stack marker for the high level language TRY feature. If CIR(19:1)=1, this instruction assumes that a PB-relative recovery address is in the TOS. This word is popped from the stack and pushed back on the stack after the content of SB+2 is pushed onto the stack. Following this, the Status register and a delta-Q entry are pushed onto the stack. The SB-relative value of S is then stored at SB+2. If CIR(19:1)=0, S and (SB+2) are restored to their pre-TRY values and zero is stored at SB+4. No bounds tests are made

on SB+2 or SB+4.
Indicators Unaffected.

167. ESCP (Escape) Format 4 Opcode 3=41

```
If CIR(19:1)=1 then
Begin
     If (SB+4) # 0 then FLAG:= TRUE else FLAG:= FALSE
End else
Begin
     (SB+3):= (S)
     S:= S-1
     FLAG:= TRUE
End
If FLAG= TRUE then
Begin
     (SB+4):= -1
     TEMP:= SB+ (SB+2)
     If SB+12 < TEMP < S then
     Begin
          (SB+2):= (TEMP-3)
          Q:= TEMP
          Execute EXIT 0 Instruction
     End else Bounds Violation Trap
End
```

This instruction supports the high level language ESCAPE feature. If CIR(19:1)=0 then this instruction stores the TOS at SB+3, pops the stack once, stores a -1 at SB+4, restores the word at SB+2, and executes the EXIT instruction (N=0) using the most current pseudo stack marker pushed by the TRY instruction. If CIR(19:1)=1 and if the content of SB+4 is not zero then the same operations take place except that the TOS is not stored at SB+3. No bounds tests are made on SB+2, SB+3, or SB+4. SEE APPENDIX G.
Indicators Unaffected unless EXIT is executed

168. MUC (Modify Use Count) Format 4 Opcode 3=13

```
TEMP:= Evaluation of Data Segment Ptr. in (S)
If Absent seg/page table/page then CC:= CCE else
Begin
     If Ptr. Eval detects any other errors then CC:=
     CCL else
     Begin
```

```
                    CC:= CCG
                    If CIR(19:1)=1 then
                    Begin
                         Increment Use Count for seg/page
                    End else
                    Begin
                         Decrement Use Count for seg/page
                    End
               End
          End
          S:= S-1
```

This instruction expects an external data segment pointer in the TOS and either increments (if CIR(19:1)=1) or decrements the use count for that segment (or page in the case of a paged data segment) if the pointer is valid and the segment (or page or page table) is not absent. The pointer is popped from the stack. This is a privileged instruction.
Indicators = CCE if valid pointer but something absent
CCL if invalid pointer
CCG if valid pointer and nothing absent

169. GTSP (Get Task Segment Pointers) Format 4 Opcode 3=12

```
(CTCBP):= (S)
S:= S+2
(S-2):= 0
(S-2)(1:12):= ((CTCBP)+2)(4:12)
(S-1):= 0
(S-1)(1:12):= ((CTCBP)+2)(20:12)
(S):= 0
(S)(29:3):= MASK(3:3)
DST:= ((CTCBP)+1)
```

This instruction expects an absolute address pointing to a Task Control Block in the TOS. This address is popped from the stack and stored in the dedicated memory location (see Appendix D) for the Current Task Control Block Pointer. The segment numbers for the task's stack and global segments are read from the third word of the TCB, converted into external data segment pointers (offset=0), and pushed onto the stack. The MPB channel number of the CPU is also pushed onto the stack. The DST register is set from the second word of the TCB. This is a privileged instruction.
Indicators Unaffected

7. SAMPLE PROGRAM

This section contains an example of a program written for the FOCUS system. The example chosen is the recursive calculation of N factorial where N is a positive integer or zero. This program is intended to demonstrate some of the general features of the FOCUS instruction set. It is not intended to be an efficient implementation of the calculation of N factorial. The program consists of two parts, the calling sequence and the procedure body. The former is assumed to start at memory location 10000 octal while the latter is assumed to start at memory location 20000 which could be in a different code segment from the calling sequence. The following machine instructions perform the desired task.

LOCATION	INSTRUCTION	COMMENT
10000	ZERO	Allocate Space for Result
2	LDS DB+10	Load Actual Parameter
4	PCLS 2	Call Factorial Procedure
6	STS DB+12	Store Result
10	Rest of Program	
20000	LDS Q-4	Load Passed Parameter
2	CMPI 0	Test it for Zero (Pop Stack)
4	BCCS GL P+3	If not Zero, Branch to 20012
6	LDIS 1	If Zero, Return 1
10	BCCS GEL P+10	Branch to 20030
12	ZERO	Allocate Space for Intermediate Result
14	LDS Q-4	Load Passed Parameter
16	SUBI 1	Decrement Parameter
20	PCLS 2	Recursive Call
22	LDS Q-4	Load Passed Parameter
24	MPY	Multiply By Previous TOS
26	DELB	Delete Upper Half of Product
30	STS Q-5	Store Intermediate Product
32	EXIT 1	Exit this Recursion

The first instruction in the calling sequence allocates space

on the stack for the result produced by the factorial procedure. In this case the result is assumed to be a 32-bit integer. The actual parameter passed to the factorial procedure is also a 32-bit integer which is located in the global data segment. In this case the 16-bit LDS instruction can be used to push the parameter onto the stack. It is further assumed that the second entry in the STT of the segment containing the calling sequence contains the appropriate local or external program pointer to the factorial procedure. Hence a PCLS 2 instruction accomplishes the initial call. After the final EXIT from the factorial procedure the result will be on top of the stack and it can be stored in the global data segment with a 16-bit STS instruction.

The factorial procedure first tests the passed parameter for zero by loading the parameter onto the TOS and executing a CMPI 0 instruction. If the parameter is zero, the CMPI instruction will set the condition code bits in the Status register to CCE. Hence a test for a non-zero parameter is simply a 16-bit BCCS instruction which branches on CCG or CCL. If the parameter is zero, the procedure returns a value of 1 by first loading a 1 on the stack and then executing an unconditional branch to location 20030 which stores the result at Q-5 and EXITs. If the parameter N is not zero the procedure computes N times (N-1)! which it computes by calling itself. The calling sequence is as above, namely, allocate space for the result, push the actual parameter (which is the passed parameter minus 1) onto the stack and call the factorial procedure.

After each EXIT, except for the final return to the initial calling sequence, execution resumes at location 20022 with an intermediate result (N-1)!, in the TOS. This is multitiplied by N, the passed parameter, by executing a LDS Q-4 and a MPY instruction. This new intermediate result is stored at Q-5 before EXITing.

APPENDIX A: Index of the Machine Instructions

The following index is arranged alphabetically by mnemonic. The number preceding each mnemonic refers to the number preceding the description of each instruction in Section 6.2.

74.	ADAX	(Add A to X)
56.	ADD	(Add)
68.	ADDI	(Add Immediate)
36.	ADDS	(Add to S)
75.	ADXA	(Add X to A)
76.	ADXI	(Add Immediate to X)
160.2	ALM	(Address Lockout Mode)
102.	AND	(Logical AND)
107.	ANDI	(And Immediate)
41.	ASL	(Arithmetic Shift Left)
42.	ASR	(Arithmetic Shift Right)
109.	BCC	(Branch on Condition Code)
109.	BCCS	(Branch on Condition Code, 16-bit version)
110.	BCY	(Branch on Carry)
115.	BIR	(Branch on Inexact Result)
111.	BNC	(Branch on No Carry)
113.	BNO	(Branch on No Overflow)
112.	BOV	(Branch on Overflow)
108.	BR	(Branch)
114.1	BRE	(Branch on TOS Even)
114.2	BRO	(Branch on TOS Odd)
108.1	BRSI	(Branch Stack Indirect)
114.	BRZ	(Branch on TOS Zero)
94.3	BTD	(Binary to Decimal)
116.	BUN	(Branch on Unused Condition Code)
27.	CAB	(Rotate ABC)
156.	CCI	(Convert Condition Code to Integer)
101.	CDSF	(Convert Double to Single Word Floating)
163.	CHEK	(Check)
61.	CMP	(Compare)
137.	CMPB	(Compare Bytes)
72.	CMPI	(Compare Immediate)
30.	CMPM	(Compare Memory)
73.	CMPN	(Compare Negative Immediate)
164.	CNTO	(Count Ones)
155.	CPAA	(Convert Pointer to Absolute Address)
155.5	CPEP	(Convert Pointer to Ext. Data Seg. Ptr.)
155.6	CPTR	(Convert Pointer)
114.3	CRB	(Compare Range & Branch)

100.9	CSDF	(Convert Single to Double Word Floating)
45.	CSL	(Circular Shift Left)
46.	CSR	(Circular Shift Right)
20.	DDEL	(Double Delete)
23.	DDUP	(Double Duplicate)
81.3	DECF	(Decrement Memory by Four)
81.1	DECM	(Decrement Memory)
19.	DEL	(Delete A)
21.	DELB	(Delete B)
124.	DISP	(Dispatch)
59.	DIV	(Divide)
71.	DIVI	(Divide Immediate)
59.1	DIVS	(Divide, Single Word Dividend)
93.8	DLAD	(Decimal Logical Add)
93.9	DLSB	(Decimal Logical Subtract)
54.	DPF	(Deposit Field)
94.1	DTB	(Decimal to Binary)
22.	DUP	(Duplicate A)
25.	DXCH	(Double Exchange)
7.	DZRO	(Double Push Zero)
129.	EOP	(End of Procedure)
131.	EOS	(End of Subroutine)
167.	ESCP	(Escape)
53.	EXF	(Extract Field)
121.	EXIT	(Procedure Exit)
82.	FADD	(Floating Add)
88.	FADS	(Floating Add Single)
87.	FCMP	(Floating Compare)
93.	FCPS	(Floating Compare Single)
85.	FDIV	(Floating Divide)
91.	FDVS	(Floating Divide Single)
137.2	FILB	(Fill Bytes)
98.	FIXD	(Fix Double)
100.	FIXS	(Fix Single)
165.	FLMO	(Find Leftmost One)
97.	FLTD	(Float Double)
96.	FLTS	(Float Single)
90.	FMPS	(Floating Multiply Single)
84.	FMPY	(Floating Multiply)
86.	FNEG	(Floating Negate)
92.	FNGS	(Floating Negate Single)
89.	FSBS	(Floating Subtract Single)
83.	FSUB	(Floating Subtract)
93.1	FTSS	(Floating Test Single)
87.1	FTST	(Floating Test)

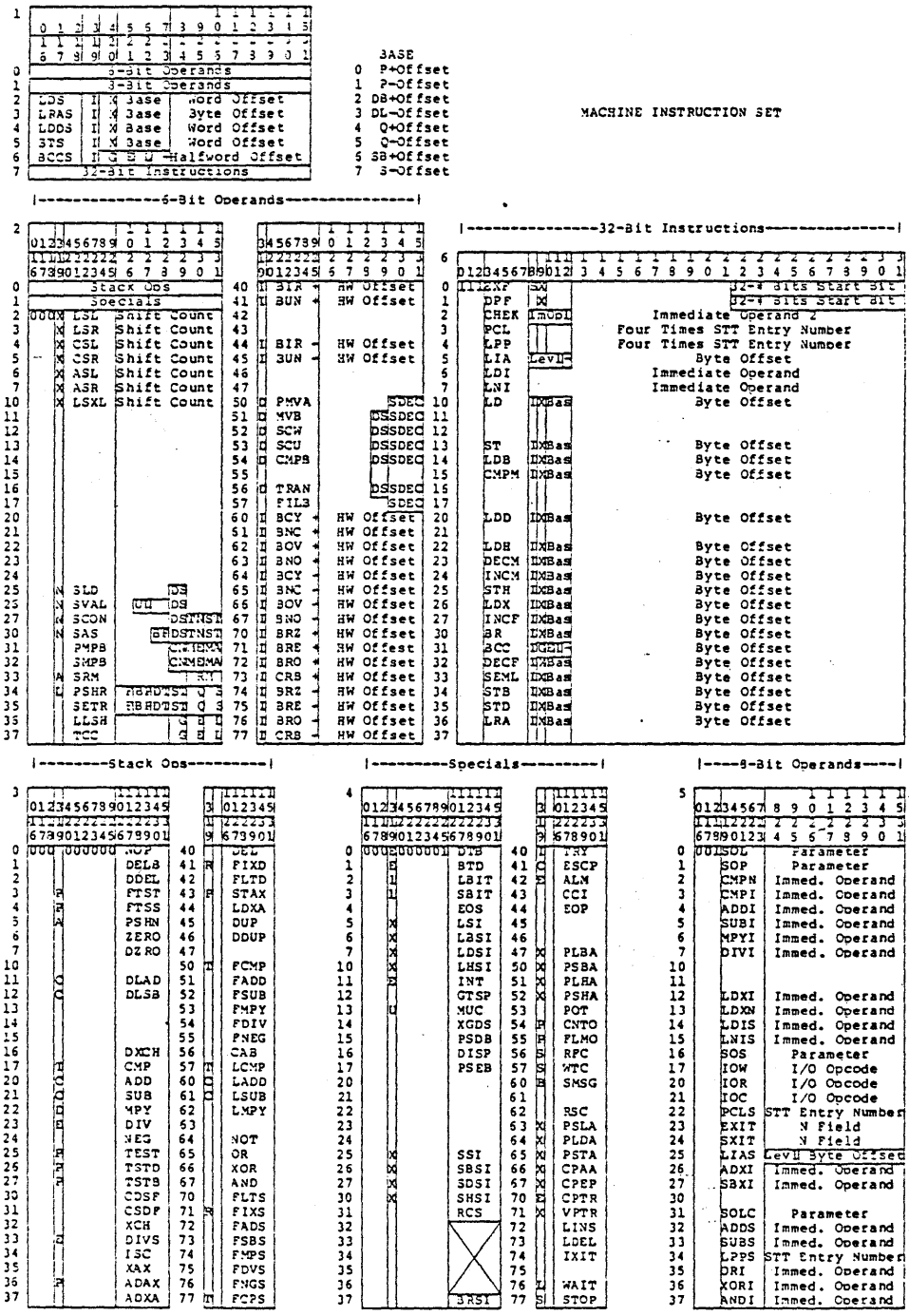
169.	GTSP	(Get Task Segment Pointers)
81.2	INCF	(Increment Memory by Four)
81.	INCM	(Increment Memory)
138.	INT	(Interrupts)
142.2	IOC	(I/O Control)
142.1	IOR	(I/O Read)
142.	IOW	(I/O Write)
95.	ISC	(Integer Size Check)
123.	IXIT	(Interrupt Exit)
61.1	LADD	(Logical Add)
4.6	LBIT	(Load Bit)
2.5	LBSI	(Load Byte Stack Indirect)
61.5	LCMP	(Logical Compare)
1.	LD	(Load)
2.	LDB	(Load Byte)
3.	LDD	(Load Double)
3.	LDDS	(Load Double, 16-bit version)
150.2	LDEL	(List Delete)
4.	LDH	(Load Halfword)
8.	LDI	(Load Immediate)
8.	LDIS	(Load Immediate, 16-bit version)
1.	LDS	(Load, 16-bit version)
3.5	LDSI	(Load Double Stack Indirect)
30.	LDX	(Load Index)
5.	LDXA	(Load X onto Stack)
28.	LDXI	(Load X Immediate)
29.	LDXN	(Load X Negative Immediate)
4.5	LHSI	(Load Halfword Stack Indirect)
157.	LIA	(Load Intermediate Address)
157.	LIAS	(Load Imtermediate Address, 16-bit version)
150.1	LINS	(List Insert)
150.	LLSH	(Linked List Search)
61.3	LMPY	(Logical Multiply)
9.	LNI	(Load Negative Immediate)
9.	LNIS	(Load Negative Immediate, 16-bit version)
11.	LPP	(Load Program Pointer)
11.	LPPS	(Load Program Pointer, 16-bit version)
10.	LRA	(Load Relative Address)
10.	LRAS	(Load Relative Address, 16-bit version)
1.5	LSI	(Load Stack Indirect)
43.	LSL	(Logical Shift Left)
44.	LSR	(Logical Shift Right)
61.2	LSUB	(Logical Subtract)
55.	LSXL	(Logical Shift X Left)
58.	MPY	(Multiply)

70.	MPYI	(Multiply Immediate)
168.	MUC	(Modify Use Count)
133.	MVB	(Move Bytes)
60.	NEG	(Negate)
117.	NOP	(No Operation)
104.	NOT	(Logical Complement)
103.	OR	(Logical OR)
107.2	ORI	(Or Immediate)
120.	PCL	(Procedure Call)
120.	PCLS	(Procedure Call, 16-bit version)
12.1	PLBA	(Priv Load Byte from Absolute Address)
12.	PLDA	(Privileged Load from Absolute Address)
12.2	PLHA	(Priv Load Halfword from Absolute Address)
140.	PMPB	(Push MPB registers)
133.1	PMVA	(Priv More Bytes using Absolute Addresses)
94.5	POT	(Power of Ten)
18.1	PSBA	(Priv Store Byte Into Absolute Address)
125.	PSDB	(Pseudo Interrupt Disable)
126.	PSEB	(Pseudo Interrupt Enable)
18.2	PSHA	(Priv Store Halfword into Absolute Address)
36.5	PSHN	(Push N Words)
38.	PSHR	(Push Registers)
154.	PSLA	(Privileged Semaphore Load from Abs. Address)
18.	PSTA	(Privileged Store into Absolute Address)
152.	RCS	(Read Control Store)
158.	RFC	(Read from Channel)
159.5	RSC	(Read Slave Channel)
137.5	SAS	(String Assign)
16.6	SBIT	(Store Bit)
13.5	SBSI	(Store Byte Stack Indirect)
77.	SBXI	(Subtract Immediate from X)
137.6	SCON	(String Concatenata)
136.	SCU	(Scan Until)
135.	SCW	(Scan While)
15.5	SDSI	(Store Double Stack Indirect)
153.	SEML	(Semaphore Load)
39.	SETR	(Set Registers)
16.5	SHSI	(Store Halfword Stack Indirect)
137.4	SLD	(String Load)
141.	SMPB	(Set MPB registers)
160.	SMSG	(Send Message)
127.	SOL	(Start of Line)
127.5	SOLC	(Start of Line Check)
128.	SOP	(Start of Procedure)
130.	SOS	(Start of Subroutine)

101.1	SRM	(Set Rounding Mode)
13.5	SSI	(Store Stack Indirect)
13.	ST	(Store)
17.	STAX	(Store A in X)
14.	STB	(Store Byte)
15.	STD	(Store Double)
16.	STH	(Store Halfword)
118.	STOP	(Stop)
13.	STS	(Store, 16-bit version)
57.	SUB	(Subtract)
69.	SUBI	(Subtract Immediate)
37.	SUBS	(Subtract from S)
137.3	SVAL	(String Validate)
122.	SXIT	(Subroutine Exit)
156.5	TCC	(Test Condition Code)
33.	TEST	(Test TOS)
137.1	TRAN	(Translate)
34.	TSTB	(Test Byte)
34.5	TSTD	(Test Double)
166.	TRY	(Try)
155.1	VPTR	(Validate Pointer)
118.1	WAIT	(Wait)
159.	WTC	(Write to Channel)
26.	XAX	(Exchange X and A)
24.	XCH	(Exchange A and B)
40.	XGDS	(Exchange Global Data Segment)
105.	XOR	(Exclusive OR)
107.1	XORI	(Exclusive OR Immediate)
6.	ZERO	(Push Zero)

APPENDIX B: Bit Patterns for the Instructions

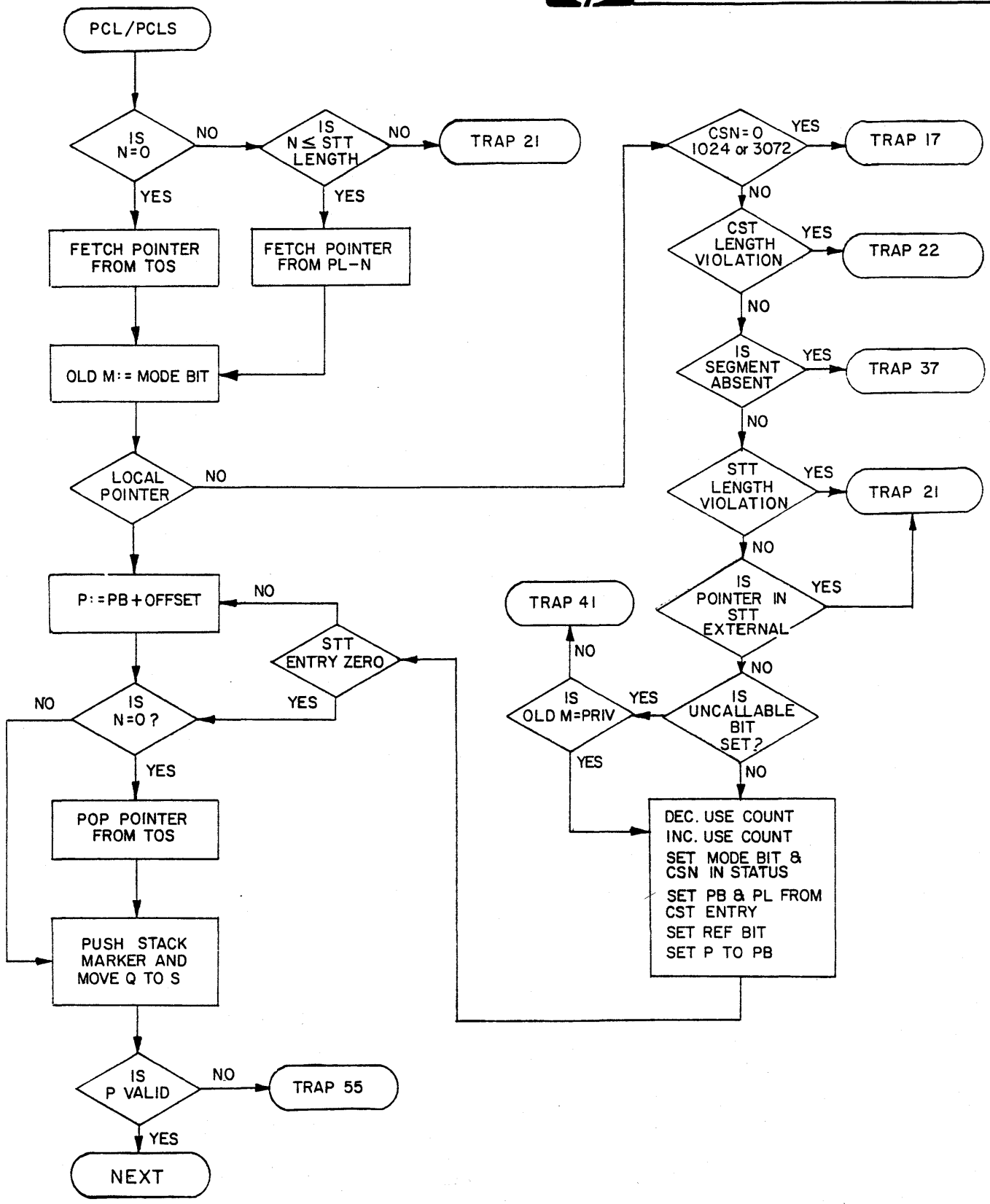
The following page contains a summary of the bit patterns for the machine instructions. In all cases the mnemonic appears in the opcode field and the octal value of the opcode appears to the left of each box. The bits are numbered from left to right across the top of each box. Bit fields for each instruction type that are the same for opcodes 0 to 37 as for opcodes 40 to 77 are not duplicated. The number in the upper left-hand corner of each section is the format number. Refer to Section 6.1 for an explanation of the different instruction formats. All unspecified bits (e.g. bits 10-21 in EXF) must be 0.



			MODEL	STK #
			Focus Machine Instruction Set ERS	
			BY J. Fiasconaro	DATE 09/23/82
LT	P.C. #	APPR	DATE APPD	SHEET # 137 OF 150
REVISIONS			SUPERSEDES	DWG # A-1FE1-3020-8

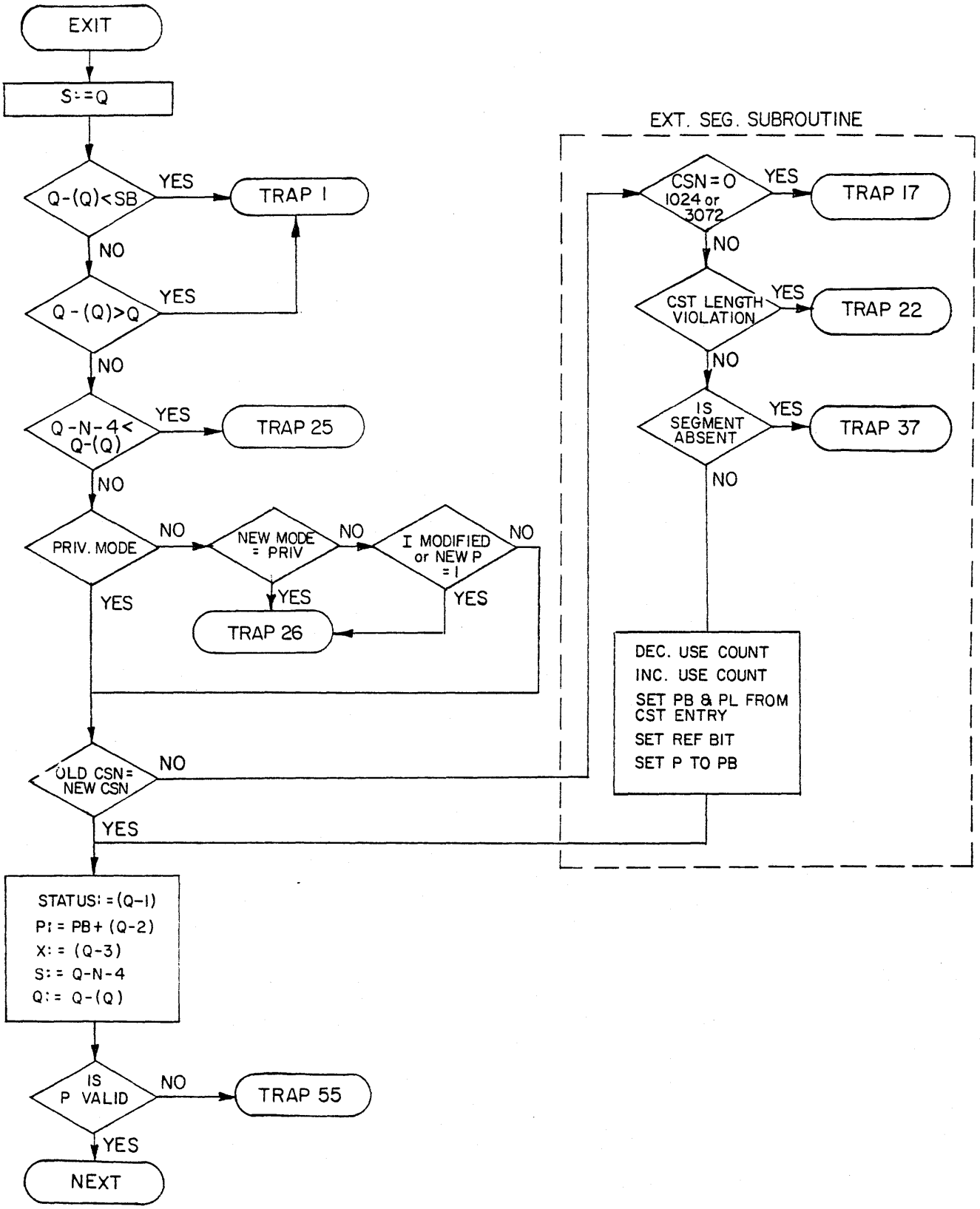
APPENDIX C: Flowcharts for Selected Features

This appendix contains flowcharts for PCL, EXIT, and IXIT instructions and for the interrupt handler.

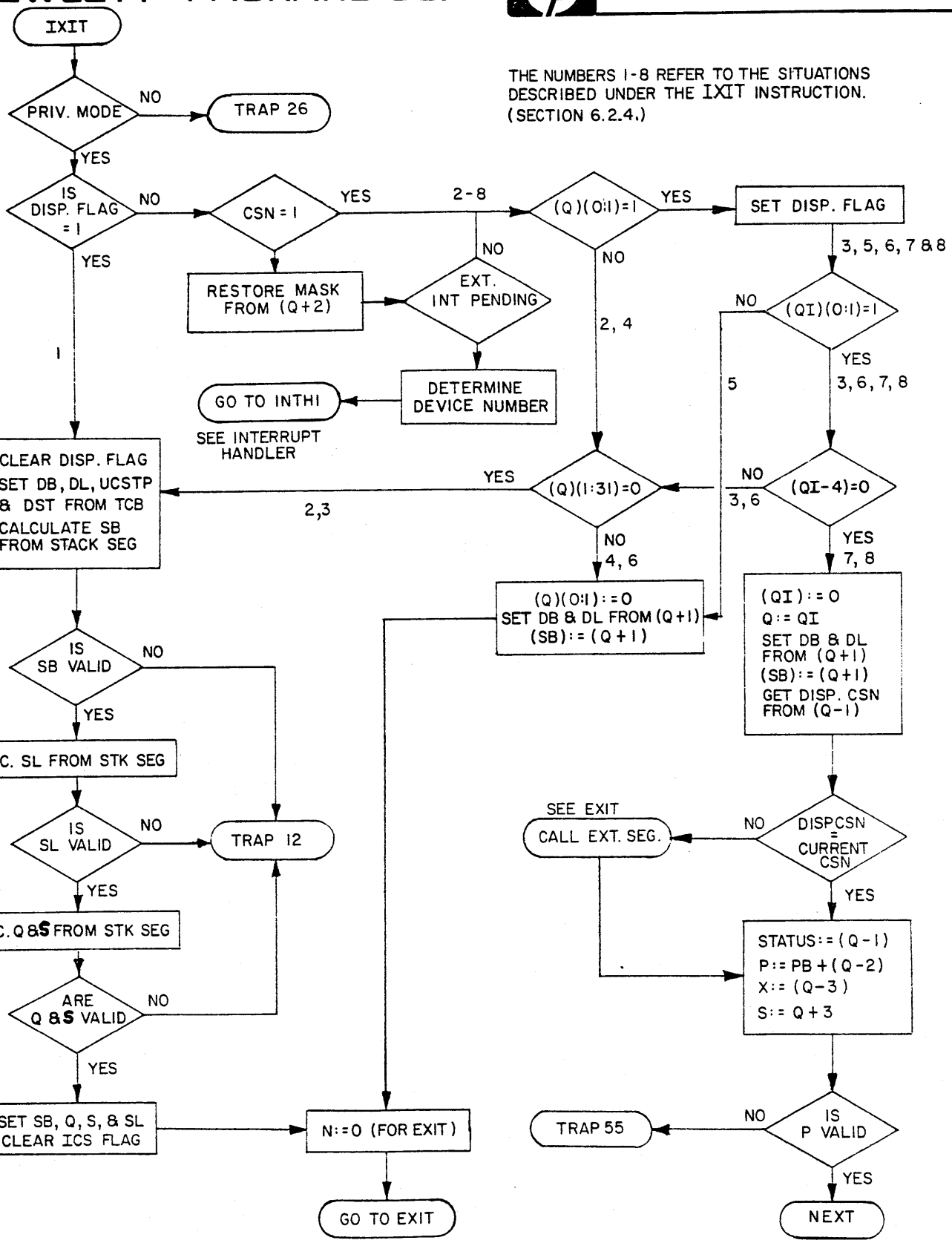


				MODEL	STK NO
				BY	DATE
LTR	PC NO	APPROVED	DATE	APPD	SHEET NO 139 OF

R. REDWICK



				MODEL	STK NO
				BY	DATE
LTR	PC NO	APPROVED	DATE	APPD	SHEET NO 140 OF
		REVISIONS P. BERWICK		SUPERSEDES	DWG NO

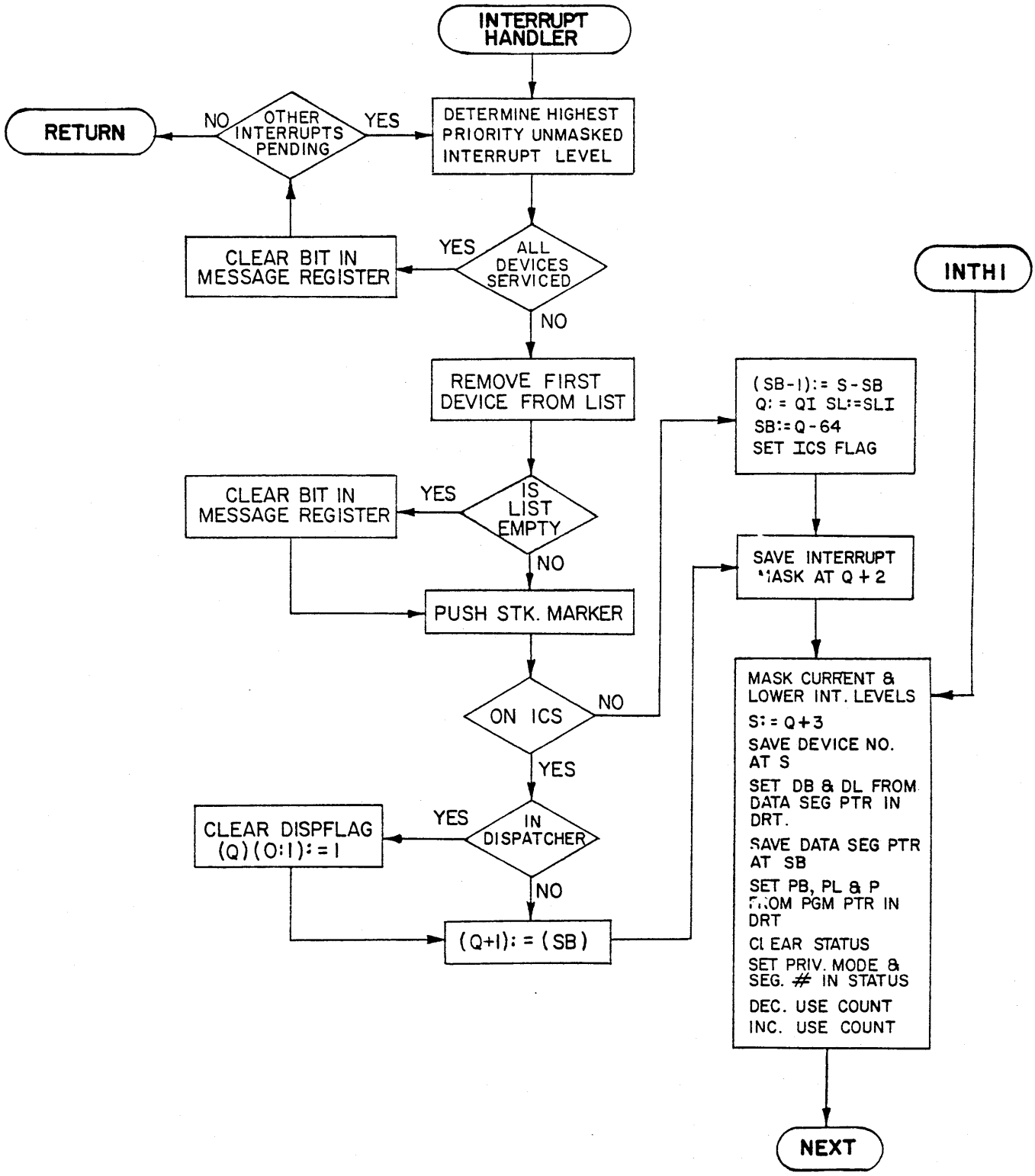


THE NUMBERS 1-8 REFER TO THE SITUATIONS DESCRIBED UNDER THE IXIT INSTRUCTION. (SECTION 6.2.4.)

LTR	PC NO	APPROVED	DATE
		REVISIONS P. BERWICK	

MODEL | STYLE
BY | DATE
APPD | SHEET 141 OF
SUPERSEDES | NEW

7830-006-L



LTR	PC NO	APPROVED	DATE
		REVISIONS P. BERWICK	

MODEL | STK NO

BY | DATE

APPD | SHEET NO 142 OF

SUPERSEDES | DWG NO

APPENDIX D. DEDICATED MEMORY LOCATIONS

LOCATION	USE
0	System Code Segment Table Pointer
4	Sharable Code Segment Table Pointer
10-14	Reserved
20-174	CPU Dedicated Locations
200-1774	Device Reference Table
2000-2174	Head and Tail Pointers
2200-	System Data Segment Table

The CPU Dedicated Locations are blocks of four words each which contain:

Word	1	User Code Segment Table Pointer
	2	Current Task Control Block Pointer
	3	QI (Interrupt Value for Q)
	4	SLI (Interrupt Value for SL)

A CPU on MPB channel N (1<N<7) uses the block starting at location 16*N (decimal). All 3 CST pointers and the current TCB pointer are absolute addresses which must end in 00. As usual, QI & SLI must end in 11.

The head and tail pointers are used by the CPU and the I/O hardware and should not be used by the operating system. The head (tail) pointer contains the absolute address of the Device Reference Table entry for the first (last) device in the list of devices waiting for service at a particular interrupt priority level. If the list is empty, the tail pointer is zero. The head and tail pointers for priority level N (0<N<15) are at locations 1028 + 8*N and 1024 + 8*N (decimal) respectively.

The Device Reference Table (DRT) contains a four-word entry for each possible I/O device. If an I/O device is attached to MPB channel N (1<N<7) and has a peripheral address of P (0<P<7) then its device number is 128*N + 16*P (decimal). This number is also the absolute address of the DRT entry for the device. The format of the DRT entry is given in Section 5.1.3.

The word at location 10 is reserved for other implementations of this instruction set and will contain an external data segment pointer whose data segment number points to the first paged data segment in the system DST.

APPENDIX E. FLOATING POINT CONVERSIONS

The following sequence of machine instructions can be used for decimal floating point to binary floating point conversion. Assume that the decimal number is at Q-6 through Q-4.

```
        LDS   Q-6       Get most significant decimal word
        DTB   E         Convert exponent to "A" and "B"
        LDDS  Q-6       Load whole decimal number
        LDS   Q-4
        DTB             Convert decimal digits to binary number
        DXCH            Swap digits number and "B"
        FDIV            Digits number/"B"
        FMPY            (Digits number/"B")*"A"
        STD   Q-6       Store converted number
```

The following sequence of machine instructions can be used for binary floating point to decimal floating point conversion. Assume that the binary number is at Q-6 and Q-5 with zero at Q-4.

```
        LDDS  Q-6       Get binary number
        BTD   E         Compute exponent guess, "I"
  LOOP  DUP
        POT             Compute "A" and "B", "A"/"B"=10^"I"
        DXCH            Swap "A" and "B"
        LDDS  Q-6       Get binary number again
        DXCH            Swap binary number and "A"
        FDIV            Binary number/"A"
        FMPY            (Binary number/"A")*"B"
        BTD             Convert fraction to decimal
        BCCS  GL LOOP   Repeat Loop if "I" bad
        STS   Q-4       Store decimal results
        STD   Q-6
```

APPENDIX F. IEEE STANDARD FLOATING POINT MATH

This appendix explains which areas of the proposed IEEE standard (Draft 5.11) for floating point computation are directly supported by the FOCUS Machine Instruction Set and the "hooks" provided to allow the operating system to provide for full compliance with the standard.

Numbering of subsections corresponds to the numbering scheme of Draft 5.11 of the proposed standard distributed by the Floating Point Working Group of the Microprocessor Standards Subcommittee of the IEEE Computer Society Computer Standards Committee. (This document was also published in the October 1979 issue of the ACM SIGNUM Newsletter.)

1) Covers the scope of the proposed standard.

2) Defines terms used in the standard document.

3) The instruction set supports both the Single and Double Basic formats for floating point numbers. No Extended format is provided.

4) All four rounding methods described by the standard are supported. (Algorithms for rounding are described in Reference 1.) The rounding mode is selected by bits in the STATUS register which can be changed using the SRM (Set Rounding Mode) instruction.

5) The instruction set includes operations for add, subtract, multiply, divide, negate, compare, and conversions between single and double floating point and 32-bit integer formats. Square root, remainder, and integer-part operations are not provided at the machine instruction level.

   Machine instructions are also provided to support very fast conversions between binary and binary coded decimal numbers (See Appendix E). The algorithms used differ from those given in Reference 1. Procedures to comply fully with the standard's requirements for the binary/decimal conversions can be furnished by system level software.

   Condition codes in the STATUS register are set to reflect the result of an operation, the machine instructions will set these bits to: "00" for ">", "10" for "=", and "01" for

"<". It is expected that the operating system will use the unused encoding, "11", to represent Unordered. The branch-on-condition-code instructions interpret both "01" and "11" as "<". Therefore to ensure proper instruction sequencing any branching based upon floating point operations should use the branch-on-unused-condition-code instruction (BUN, which explicitly tests for the "11" bit pattern) prior to BCC instructions which would branch on either the "<" case or the Unordered case.

6) Operations on infinities, and Not-a-Numbers (NaNs) will result in a Floating Point Operand Trap (#34, see section 5.2). Implementation of Affine and Projective infinity arithmetic modes, and Trapping/Nontrapping NaNs is left to the operating system.

Operations on signed zeros are fully supported. Zero operands (+/-) are detected early during the execution of an instruction and will cause a "fast path" to be taken through the microcode. (This is expected to provide substantial performance benefit to computations involving sparse data structures.)

7) Operations on denormalized numbers will result in a Floating Point Operand Trap (#34, see section 5.2). This allows system software to choose between implementing "gradual underflow" for full standard compliance or "flush to zero" for less demanding applications.

8&9) The operating system is expected to implement the status indicator and user trap enable functions for the five exception conditions defined by the proposed standard.

Since all operations involving infinities, NaNs, and denormalized numbers cause traps, it should be rather straightforward to detect the Invalid Operation exception. Division by zero is also trapped and may be distinguished from division by an invalid operand by examining the bits of the trap parameter, (see section 5.2, trap #34), which indicates the trap's cause. The NaN, infinity, and denormalized bits for the divisor will all be zero.

Overflow and Underflow both cause a Floating Point Result Trap (#35, see section 5.2). There are bits to indicate which of these two conditions caused the trap. Note that the machine instructions will generate an Underflow trap if

the result of an operation cannot be represented as a normalized number. The trapped result will have a correctly rounded significand and wrapped around exponent, and correct sign as described in the standard. The parameter delivered to the trap handler includes bits which indicate the state of the result's least significant bit, rounding bit, and sticky bit prior to applying rounding, these can be used to "un-round" the result if necessary (See Reference 1).

Since the Inexact Result condition will occur rather frequently it would degrade performance severely if it were trapped, therefore any operation which produces an inexact result sets an indicator bit in the STATUS register. To intercept this condition the branch-on-inexact-result (BIR) instruction may be used, this instruction will generate a Floating Point Result Trap if it's branch offset is zero, (Note: overflow and underflow indicator bits in the trap parameter will be zero).

References:

1) "An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic", Jerome T. Coonen, Volume 13, Number 1, IEEE Computer Magazine, January 1980.

APPENDIX G. KNOWN BUGS

The following bugs exist in all CPU's with release dates of Mar. 19, 1982 (i.e. CPU 4.0) and earlier. The release date of a CPU chip can be determined by reading control store location 6 with the RCS instruction. The bottom 32 bits should be interpreted as 8 hex digits in the format YYYYMMDD where YYYY is the year, MM is the month, and DD is the day.

IXIT - Stack overflow can occur if IXIT is interrupted appropriately. This can be avoided by making sure interrupts are off prior to executing each IXIT. This has the side effect of increasing the Interrupt Response Time of the CPU from about 25 microseconds to about 32 microseconds.

INSTRUCTION ALIGNMENT - It is not possible to branch to a 16-bit instruction in the right half of a 32-bit word which has arbitrary data (which may look like the left half of a 32-bit instruction) in the left halfword. All non-sequential instruction fetches (PCL, EXIT, IXIT, SXIT, ESCP, DISP, PSEB, all branch instructions, all traps, and all interrupts) are affected. This situation must be avoided (i.e. if the right halfword is a branch target, the left halfword must look like a 16-bit instruction).

SAS - No trace variable trap occurs if the current length of the source string is zero.

RESET TRAP - The CPU enters this trap handler in secondary address mode. It can be put in primary address mode by doing an IOR instruction in the trap handler.

The following bugs exist in all CPU's with release dates of Nov. 2, 1981 (i.e. CPU 3.3) and earlier. They are fixed in the CPU released on Mar. 19, 1982 (i.e. CPU 4.0).

BRSI - This instruction must not be used with external program pointers. It can cause erroneous instruction sequencing.

ESCP - If CIR(19:1)=0, this instruction can cause the delta Q entry of a pseudo stack marker created by a TRY instruction to be lost in certain cases. This problem can be avoided by preceding this instruction with a DUP if CIR(19:1)=0.

LDD(S), LDSI, STD, SDSI - These instructions can access non-

existent memory (causing a slave address error and slave data error) if the first word accessed is the last word in a segment or page and there is no memory mapped immediately after this first word.

UNIMPLEMENTED INSTRUCTION TRAP - Certain instruction bit patterns, namely, Format 4 Opcode 75, Format 5 Opcode 30, and Format 6 Opcode 37 should, but do not, cause unimplemented instruction traps. They should be avoided.

			MODEL	STK #
			Focus Machine Instruction Set ERS	
			BY J. Fiasconaro	DATE 09/23/82
LT	P.C. #	APPR	DATE	APPD	SHEET # 150 OF 150
REVISIONS			SUPERSEDES	DWG # A-1FE1-3020-8