60459960

CONTROL DATA CORPORATION

CDC® CYBER 170 MODELS 825, 835, AND 855 COMPUTER SYSTEMS

GENERAL DESCRIPTION

RESTRICTED DISTRIBUTION

HARDWARE MAINTENANCE MANUAL

60459960

CONTROL DATA
CORPORATION

CDC® CYBER 170
MODELS 825, 835, AND 855
COMPUTER SYSTEMS

GENERAL DESCRIPTION

RESTRICTED DISTRIBUTION

The material contained herein is the property of Control Data Corporation and is intended solely for use by Control Data personnel. Distribution is restricted to a specific authorized distribution list and lock-and-key control is mandatory. Reproduction and/or distribution outside Control Data is prohibited.

HARDWARE MAINTENANCE MANUAL

REVISION RECORD

REVISION	DESCRIPTION
A	Manual released.
(07-30-82)	
Publication No. 60459960	

REVISION LETTERS I, O, Q, S, X AND Z ARE NOT USED

Address comments concerning this manual to:
Control Data Corporation
Publications and Graphics Division
4201 North Lexington Avenue
St. Paul, Minnesota 55112

or use Comment Sheet in the back of this manual.

LIST OF EFFECTIVE PAGES

New features, as well as changes, deletions, and additions to information in this manual, are indicated by bars in the margins or by a dot near the page number if the entire page is affected. A bar by the page number indicates pagination rather than content has changed.

PAGE	REV
Front Cover	-
Title Page	-
2	A
3/4	A
5	A
6	A
7	A
8	A
9	A
1-1	A
1-2	A
1-3	A
1-4	A
1-5	A
1-6	A
1-7	A
1-8	A
1-9	A
1-10	A
1-11	A
1-12	A
1-13	A
1-14	A
1-15	A
1-16	A
1-17	A
1-18	A
1-19	A
1-20	A
1-21	A
1-22	A
1-23	A
1-24	A
1-25	A
2-1	A
2-2	A
2-3	A
2-4	A
2-5	A
2-6	A
2-7	A
2-8	A
2-9	A
3-1	A
3-2	A
3-3	A
3-4	A
3-5	A
3-6	A
3-7	A
3-8	A
3-9	A
3-10	A
3-11	A
3-12	A
3-13	A
3-14	A
3-15	A
3-16	A
3-17	A
3-18	A
3-19	A
3-20	A
3-21	A
4-1	A
4-2	A
4-3	A
4-4	A
4-5	A
4-6	A
4-7	A
4-8	A
4-9	A
5-1	A
5-2	A
5-3	A
5-4	A
5-5	A
5-6	A
5-7	A
6-1	A
6-2	A
6-3	A
6-4	A
6-5	A
6-6	A
6-7	A
7-1	A
7-2	A
7-3	A
7-4	A
7-5	A
7-6	A
7-7	A
7-8	A
7-9	A
7-10	A
7-11	A
7-12	A
7-13	A
7-14	A
7-15	A
7-16	A
7-17	A
7-18	A
7-19	A
7-20	A
7-21	A
7-22	A
7-23	A
7-24	A
7-25	A
7-26	A
7-27	A
7-28	A
7-29	A
7-30	A
7-31	A
7-32	A
7-33	A
7-34	A
7-35	A
7-36	A
8-1	A
8-2	A
8-3	A
8-4	A
8-5	A
8-6	A
8-7	A
8-8	A
9-1	A
9-2	A
9-3	A
9-4	A
9-5	A
9-6	A
9-7	A
9-8	A
9-9	A
9-10	A
9-11	A
9-12	A
9-13	A
9-14	A
9-15	A
10-1	A
10-2	A
10-3	A
10-4	A
10-5	A
10-6	A
10-7	A
10-8	A
10-9	A
10-10	A
10-11	A
10-12	A
10-13	A
10-14	A
10-15	A
10-16	A
10-17	A
10-18	A
10-19	A
10-20	A
10-21	A
10-22	A
10-23	A
10-24	A
10-25	A
10-26	A
10-27	A
10-28	A
10-29	A
10-30	A
10-31	A
10-32	A
10-33	A
10-34	A
10-35	A
10-36	A
10-37	A
10-38	A
10-39	A
10-40	A
10-41	A
10-42	A
10-43	A
10-44	A
10-45	A
10-46	A
10-47	A
10-48	A
10-49	A
10-50	A
10-51	A
10-52	A
10-53	A
10-54	A
10-55	A
10-56	A
10-57	A
10-58	A
10-59	A
10-60	A
10-61	A
10-62	A
10-63	A
10-64	A
10-65	A
10-66	A
10-67	A
10-68	A
10-69	A
10-70	A
10-71	A
10-72	A
10-73	A
10-74	A
10-75	A
10-76	A
10-77	A
10-78	A
10-79	A
10-80	A
10-81	A
10-82	A
10-83	A
10-84	A
10-85	A
10-86	A
10-87	A
10-88	A
11-1	A
11-2	A
11-3	A
11-4	A
11-5	A
11-6	A
11-7	A
12-1	A
12-2	A
12-3	A
12-4	A
12-5	A
12-6	A
12-7	A
12-8	A
12-9	A
12-10	A
12-11	A
12-12	A
12-13	A
12-14	A
12-15	A
12-16	A
12-17	A
12-18	A
Comment Sheet	A
Back Cover	-

CONTENTS

FIGURES

This document presents an overview of the CYBER 180 system, primarily from a hardware perspective. A brief software overview, however, is contained in section 12.

INTRODUCTION

The CYBER 180 mainframes support two architectures:

- The CYBER 180 architecture (Virtual State) with 64-bit central memory words, virtual memory management, 16-bit PP instructions and 16-bit I/O channels, and so forth. CYBER 180 is the native mode of the processor.

- The CYBER 170 architecture (Real State) with 60-bit central memory words, 12-bit PP instructions and 12-bit I/O channels, and so forth. The CYBER 170 environment requires support from the CYBER 180 state of the processor.

Both the CYBER 170 and CYBER 180 environments may be present at the same time with the processor executing in either environment. The CYBER 170 environment allows current CYBER 170 application codes to be efficiently run on the CYBER 180 mainframes. Either CYBER 170 NOS or NOS/BE may be installed in the CYBER 170 environment of CYBER 180. This allows users to install CYBER 180 mainframes and utilize the CYBER 170 application packages and other existing CYBER 170 programs and then migrate tasks to the CYBER 180 environment as desired. The CYBER 180-based operating system is termed NOS/VE.

The CYBER 180 mainframe consists of a processor, central memory, and I/O unit (figure 1-1).

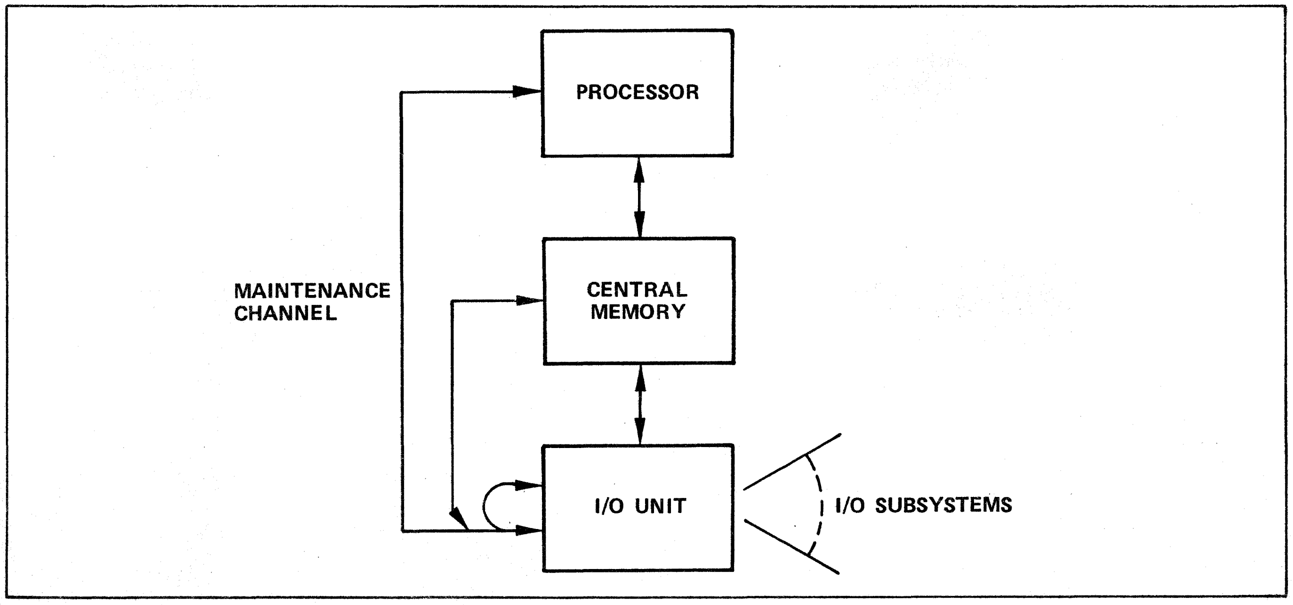


Figure 1-1. CYBER 180 Mainframe

The first four CYBER 180 systems have been termed S1, S2, S3 and Theta. The smallest of the four systems, S1, is approximately equal to a CYBER 172 in performance. Each of the larger systems is targeted at a 3X increase in performance resulting in Theta being approximately 27 times more powerful than S1. The four systems with their respective Series 800 product designations and central memory sizes are as follows:

S1	825	2 to 8 M Bytes
S2	835	4 to 16 M Bytes
S3	855	4 to 16 M Bytes
Theta	885	4 to 32 M Bytes

CENTRAL PROCESSOR

The CYBER 180 processor has the following primary operational registers:

Program Address (P) Register	-	64 bits
16 Address (A) Registers	-	48 bits each
16 Operand (X) Registers	-	64 bits each

These and other registers associated with a specific process (or job) are contained in a 52 word (64 bit words) exchange package. This package contains all the information required for the processor to initiate or restart a process. These registers represented in the exchange package are called process registers as they are associated with a specific process (or job). There are also 14 registers in the processor which contain information relative to the processor rather than to a specific process. Processor registers include items such as page table information, options installed, and error logging. Two of these processor registers, Monitor Process State (MPS) and Job Process State (JPS) contain the real memory addresses for the current exchange package for monitor and job state respectively. Thus, an exchange operation from CYBER 180 monitor to job will store the environment of the monitor process in an exchange package at MPS and initiate the process whose exchange package is located at JPS. This monitor to job exchange (figure 1-2) is initiated by an Exchange Instruction within the monitor process. The subsequent job to monitor exchange may be initiated by either an Exchange instruction or by the occurrence of some condition requiring monitor intervention such as external interrupt, page fault, and power warning.

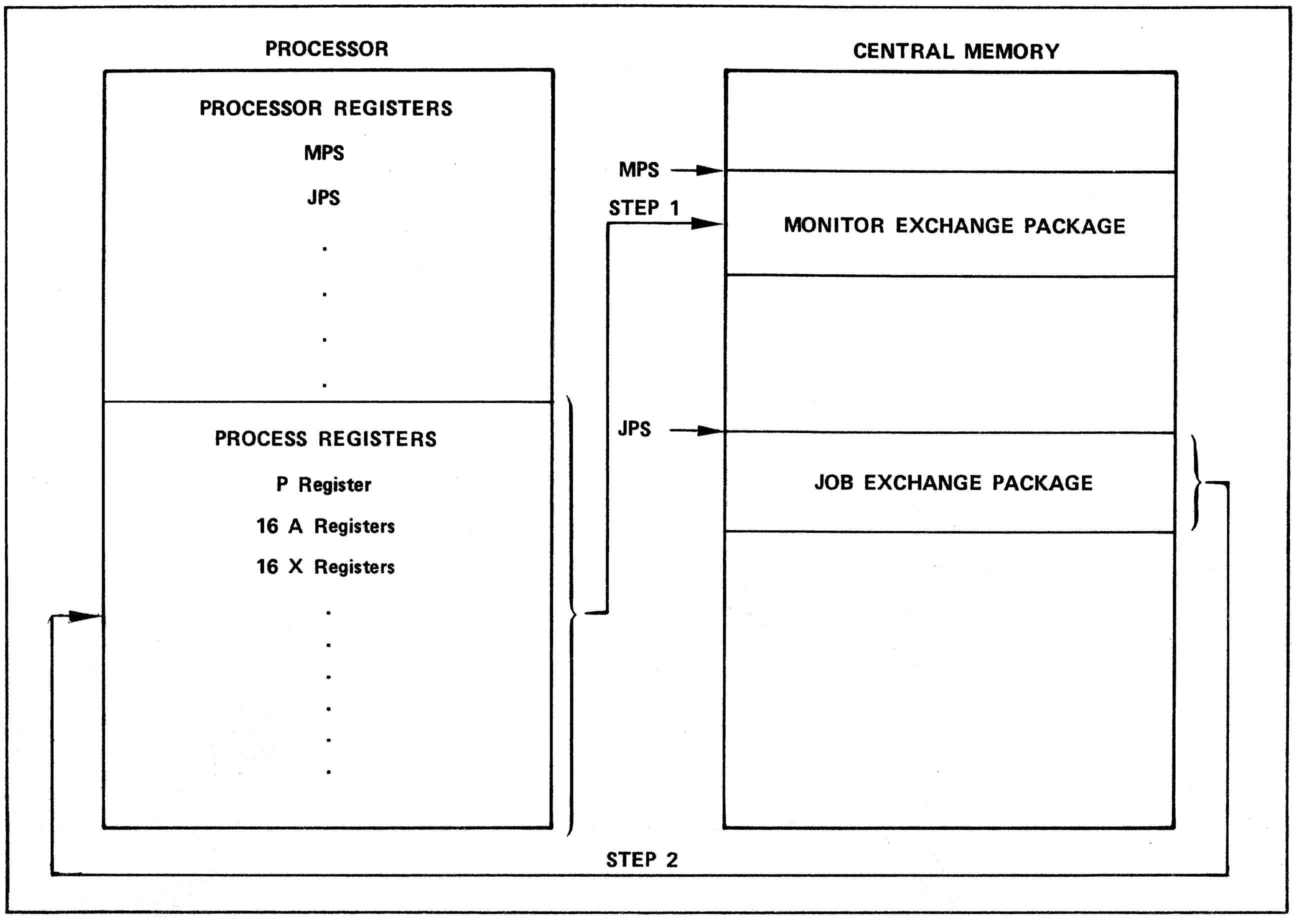


Figure 1-2. Monitor to Job Exchange

VIRTUAL MEMORY

Central processor references to central memory (other than an exchange operation) involve a virtual mapping algorithm. The transformation from the 48-bit virtual address used by the process (or job) to the actual address involves 2 steps as described in the following paragraphs:

1. Security and access validation (Does this process have permission to access this address?)

2. Virtual Memory Management (Where does the desired information actually reside?)

The 48 bit Process Virtual Address (PVA) (figure 1-3) consists of the following:

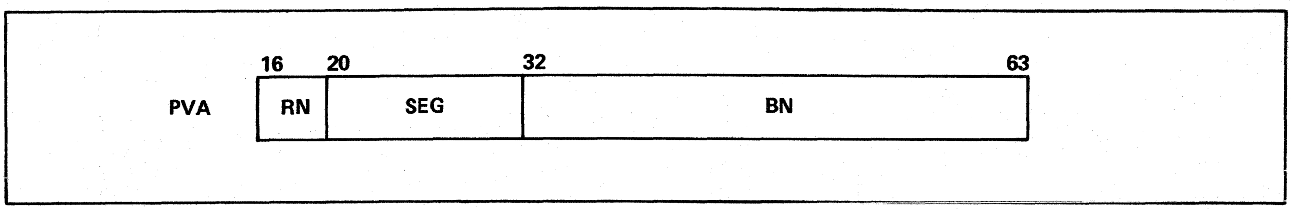


Figure 1-3. Process Virtual Address

RN: Ring Number (0-15) for security validation
SEG: Segment Number identifying 1 of 4096 segments potentially available within a single process´s address space. Each of these segments has security properties such as read only and execute only.
BN: Byte Number identifying a specific byte on the leftmost byte of a word or string within a specific segment.

NOTE

All registers are numbered left to right with bit 63 as the rightmost bit.

The first step of this virtual mapping algorithm is organized on a segment basis because security attributes are assigned per segment. The segment number is used to obtain a Segment Descriptor from the Segment Table for the process. This Segment Descriptor contains security information for the segment in this specific process. For example, a segment could be read only in one process while reads and writes both could be allowed from another process. This security information plus the nature of the reference (RNI, read, write, and so forth) are used to verify the validity of the request relative to security considerations.

When the request is invalid the process will be halted and the operating system notified. When the request is valid, an Active Segment Identifier (ASID) is obtained from the Segment Descriptor. This ASID identifies the segment on a system basis. Thus, many processes may individually use segments such as 0, 1, or 2, and have the virtual algorithm map these into unique system addresses. This also allows segments named uniquely in different processes to map into a single segment (with potentially different access privileges). This ASID replaces the Ring Number and Segment Number in the PVA to produce a System Virtual Address (SVA) (figure 1-4).

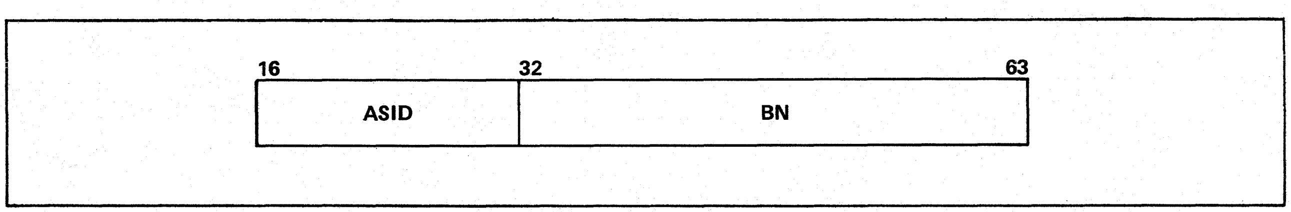


Figure 1-4. System Virtual Address

ASID: Active Segment Identifier
BN: Byte Number from the PVA

The second step of the virtual mapping algorithm is organized on a page basis because physical memory is managed on a page basis. This step locates the required portion (or page) of the segment within the system. Those pages of the various active segments which are currently in central memory are listed in the System Page Table. The SVA (produced in step 1) is used to access this table to determine if the page is in central memory, and if so, where. When the page is in central memory, the desired references are completed and processor execution continues. When the page is not in central memory, the process in execution is interrupted. Control is then given to the operating system which makes the necessary arrangements to have the desired page brought into central memory after which the interrupted process may be resumed.

The virtual mapping algorithm, terminating in a Real Memory Address (RMA) for central memory, may be summarized as shown in figure 1-5.

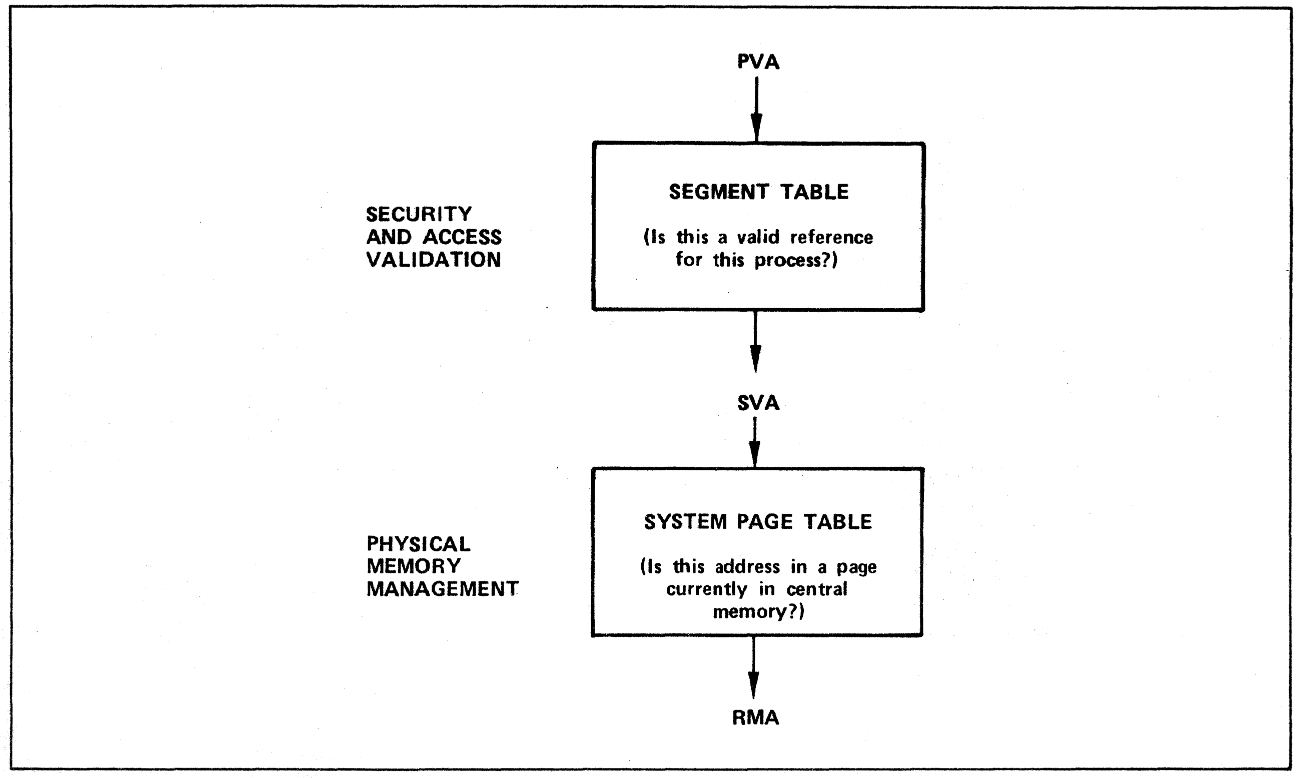

PVA
SECURITY AND ACCESS VALIDATION
SEGMENT TABLE
(Is this a valid reference for this process?)
SVA
PHYSICAL MEMORY MANAGEMENT
SYSTEM PAGE TABLE
(Is this address in a page currently in central memory?)
RMA

Figure 1-5. PVA To RMA Transformation

Since both the Segment Table and System Page Map reside in central memory, it is obvious that some type of hardware assist will be required to achieve high performance. This hardware assistance will vary according to the performance requirements of a specific system; however, figure 1-6 illustrates a typical approach implemented in several processors.

A cache-like Segment Map containing the Segment Descriptors for the most recently accessed segments is first accessed followed by a reference to the Segment Table in central memory only when the Segment Descriptor is not in the Segment MAP. A Page Map similarily supports the SVA to RMA translation process. In addition the SVA is used in parallel to the Page MAP reference to access a cache memory organized by SVA. When the cache contains the data, the access to the Page MAP is terminated. When the cache does not contain the data, the access through the Page MAP and on to central memory is continued.

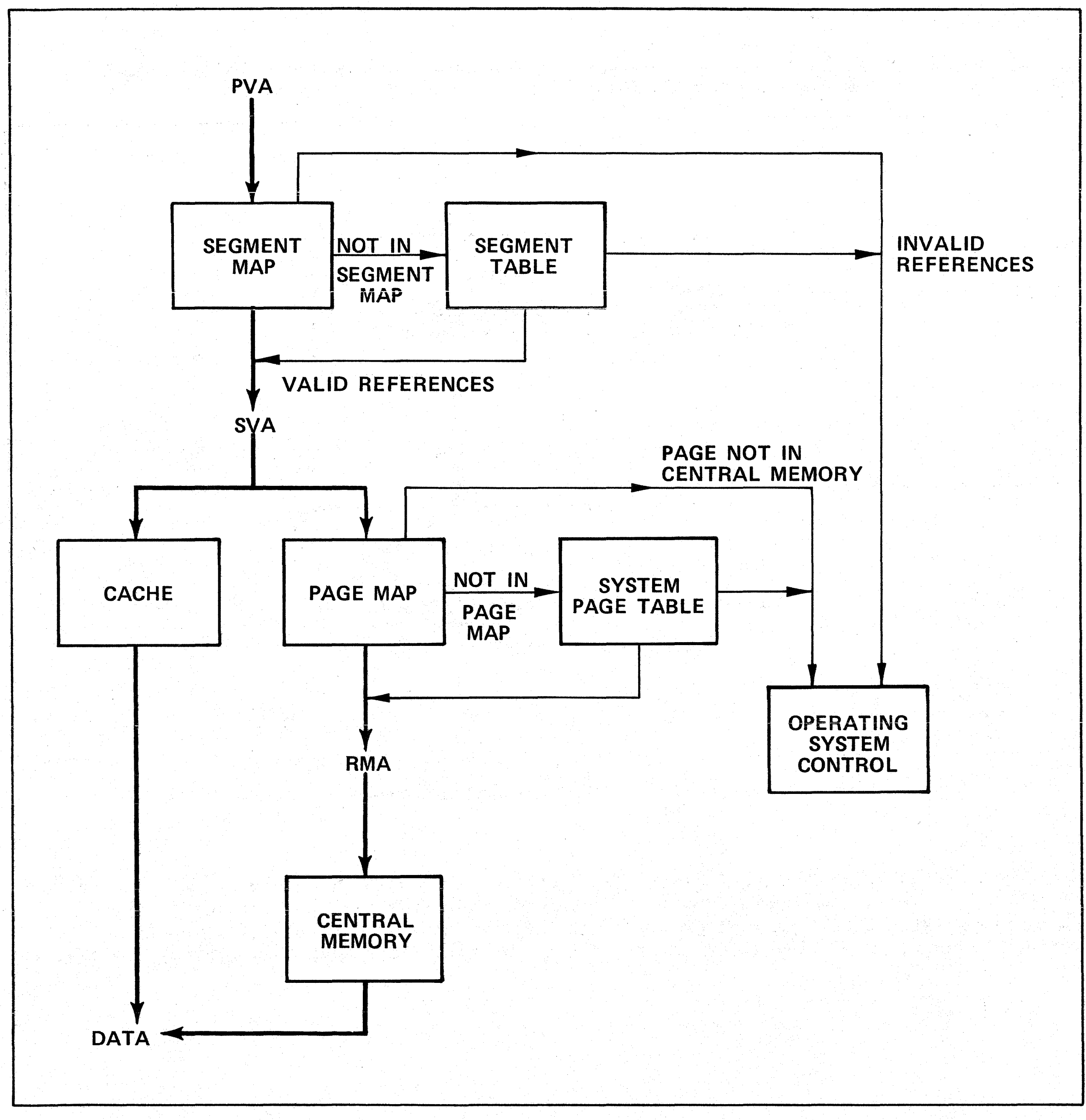


Figure 1-6. Typical MAP/CACHE Usage

INSTRUCTION SET

The CYBER 180 processor instruction set consists of the following 149 instructions:

General (Load, Store, Branch)	76
System (Call, Return, and so forth)	19
Floating Point	16
BDP	18
Vector (Theta systems only)	20
	149

The 76 general instructions include load and store operations on words (64 bits), bytes and bits; integer arithmetic operations; branches, copies, address arithmetic, direct enters and shift operations.

The system instructions include CALL, RETURN and POP instructions to facilitate the use of modular code. The EXCHANGE, COMPARE/SWAP, PROCESSOR INTERRUPT and TEST AND SET BIT support task management and interprocessor communication. The COPY FREE COUNTER provides access to a 48-bit microsecond timer for system accounting. The COPY TO/FROM STATE REGISTER instructions allow the reading and writing of certain processor and process state registers. The LOAD PAGE TABLE INDEX and PURGE BUFFER instructions allows the operating system to maintain the MAP, cache and System Page Table. The BRANCH ON CONDITION REGISTER instruction allows the testing and alteration of the registers involved in the interrupt structure. The KEYPOINT instruction allows both trace and timing data to be gathered on a process.

The single-precision floating-point instructions operate on 64-bit floating point operands consisting of a 49-bit sign/magnitude coefficient (48-bit positive number plus sign bit) and a 15-bit exponent allowing a range of 2^{+4095} to 2^{-4096} plus representations for Indefinite, Infinite and Zero. The floating-point operations are two address of the form, Xk replaced by Xk + Xj (in contrast to CYBER 170 which is Xi replaced by Xj + Xk). There are also two convert instructions which allow conversion between single-precision operands and 64-bit integers. The double-precision floating-point instructions support use of double length coefficients (96 bits plus sign). The BRANCH and COMPARE instructions complete this subset.

The BDP instructions use descriptor fields directly following the instruction in the code stream to describe fields in central memory involved in the BDP operations. For example, the DECIMAL SUM instruction reads two decimal fields from central memory, sums them and replaces one of the two input fields with the result. There are 16 different data formats which may be specified for these fields including binary, packed and unpacked decimal. Not all types of data formats are acceptable input for every instruction.

The vector instructions perform integer, logical and single-precision floating-point operations on data fields in central memory up to 512_{10} words in length. These data fields are always contiguous except for summation which produces a single output operand and the Gather, Scatter pair of instructions which are designed specifically for noncontiguous operations.

Refer to section 10 for more information.

CENTRAL MEMORY

The central memories for the CYBER 180 systems are available in the following sizes:

M1	825	2, 4 or 8 MB
M2	835	4, 8, 12 or 16 MB
M3	855	4, 8, 12 or 16 MB
THETA	885	8, 12, 16, 24 or 32 MB

These memories all include Single Error Correction/Double Error Detection (SEC/DED) as is described under RAM. Each memory contains one port for the I/O unit, one port for the processor and one port for a second processor.

I/O UNIT (IOU)

The IOU contains up to 20 peripheral processors (PP´s) and up to 24 I/O channels (either CYBER 170 12-bit channels and/or CYBER 180 16-bit channels). The PP´s each have a 4K x 16-bit memory and are capable of accessing central memory and any I/O channels.

These PP´s execute both a 12-bit instruction set compatible with CYBER 170 and a new 16-bit instruction set. These are implemented such that the 12-bit instruction set is a subset of the 16-bit instruction set. The 12-bit instruction set allows appropriate I/O system support of the CYBER 170 state in the mainframe such that CYBER 170 NOS and NOS/BE are supported. The 12-bit instruction set includes 60-bit read/write operations for central memory, the required processor interrupt signals and the ability to utilize the 12-bit CYBER 170 channels.

The 16-bit instruction set allows support of the CYBER 180 state in the mainframe and includes 64-bit read/write operations for central memory and the ability to utilize the 16-bit CYBER 180 channels.

MAINTENANCE CHANNEL

The maintenance channel provides an access from any PP to a set of maintenance registers within a specific system element such as processor, memory or I/O unit. These registers are:

- control registers independent of a specific process
- error logging and status registers
- performance data registers

A typical system is illustrated in figure 1-7.

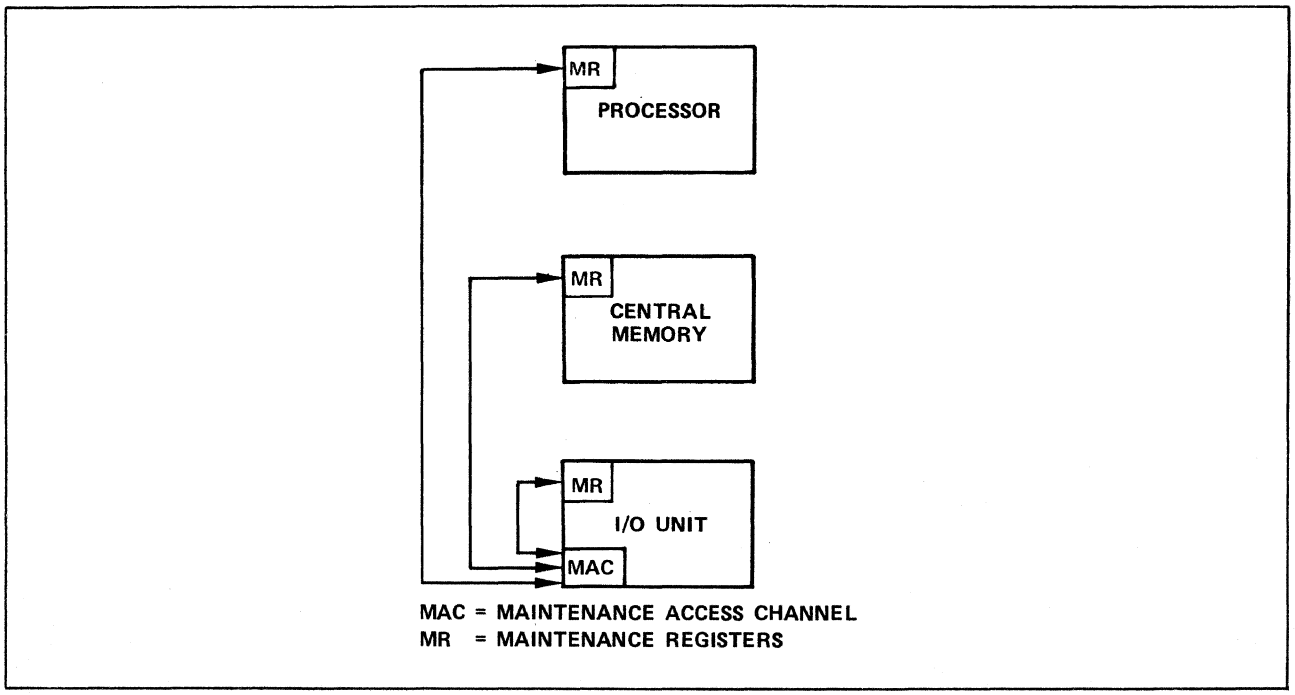


Figure 1-7. Typical Maintenance Channel

The PP may use the maintenance channel to perform operations such as:

- writing the processor registers such as Page Table Address
- reading the Corrected Error Logs from central memory
- reading the Options Installed Register in the IOU to determine the number of PPs
- initializing all system elements
- causing the processor to perform a exchange operation loading the exchange package from the location pointed to by MPS, setting the CYBER 180 Monitor Flag and beginning execution
- causing the processor to halt, allowing the reading of selected error logs followed by resumption of processor execution
- causing the processor to halt execution and then to perform a an exchange operation storing an exchange package into memory

CALL/RETURN OPERATIONS

The CALL, RETURN and POP operations efficiently support structured languages like PASCAL. The CALL instruction simply stores a copy of its current environment such as register contents or flags into central memory and then branches to a new process and resumes execution. When the new or CALLed process is complete, the execution of a RETURN instruction retrieves the environment from central memory and resumes execution immediately following the CALL instruction. The environment stored into central memory is called a STACK FRAME and is similar in format to an exchange package. This STACK FRAME is stored into an area called a STACK which may contain many STACK FRAMES as successive processes are CALLed without intervening RETURN operations. The POP instruction facilitates removal of STACK FRAMES from the STACK in circumstances when a RETURN is not appropriate.

The TRAP operation very closely parallels a CALL except that the TRAP is initiated by the interrupt structure rather than by an explicit instruction.

INTERRUPT STRUCTURE

The CYBER 180 Interrupt Structure is organized around two 16-bit registers; the Monitor Condition Register (MCR) and the User Condition Register (UCR). These registers allow the recording of those program anomalies or other events potentially important enough to justify the interruption of the process currently being executed. Figures 1-8 and 1-9 list the conditions represented in each register and outlines the action to be taken for each. Each bit in the two registers has an associated mask bit which allows some selection of the action to be taken. The actions to be taken include:

HALT - The processor stops execution.
EXCH - An exchange to CYBER 180 monitor mode.
TRAP - A CALL-like operation to another process without
performing an exchange.
STACK - Record condition but take no further action at this time.

				ASSOCIATED MONITOR MASK REGISTER BIT SET				MASK BIT CLEAR
				TRAP ENABLED		TRAP DISABLED		TRAP ENABLED OR DISABLED
P REG	BIT NUMBER AND DEFINITION			JOB MODE	MONITOR MODE	JOB MODE	MONITOR MODE	JOB OR MONITOR MODE
–	48	Detected Uncorrectable Error	Mon	EXCH	TRAP	EXCH	HALT	HALT
–	49	Unassigned		EXCH	TRAP	EXCH	HALT	HALT
P+	50	Short Warning	Sys	EXCH	TRAP	EXCH	STACK	STACK
P	51	Instruction Specification Error	Mon	EXCH	TRAP	EXCH	HALT	HALT
P	52	Address Specification Error	Mon	EXCH	TRAP	EXCH	HALT	HALT
P+	53	170 Exchange Request	Sys	EXCH	TRAP	EXCH	STACK	STACK
P	54	Access Violation	Mon	EXCH	TRAP	EXCH	HALT	HALT
P	55	Environment Specification Error	Mon	EXCH	TRAP	EXCH	HALT	HALT
P+	56	External Interrupt	Sys	EXCH	TRAP	EXCH	STACK	STACK
P	57	Page Table Search Without Find	Mon	EXCH	TRAP	EXCH	HALT	HALT
P+	58	System Call	Status - This bit is a flag only and does not cause any hardware action.					
P+	59	System Interval Timer	Sys	EXCH	TRAP	EXCH	STACK	STACK
P/P+*	60	Invalid Segment/Ring Number Zero	Mon	EXCH	TRAP	EXCH	HALT	HALT
P	61	Outward Call/Inward Return	Mon	EXCH	TRAP	EXCH	HALT	HALT
P+	62	Soft Error Log	Sys	EXCH	TRAP	EXCH	STACK	STACK
–	63	Trap Exception	Status - This bit is a flag only and does not cause any hardware action.					

* P, unless P+ for RNO on loads

Figure 1-8. Monitor Condition Register

				ASSOCIATED USER MASK REGISTER BIT SET				MASK BIT CLEAR
				TRAP ENABLED		TRAP DISABLED		TRAP ENABLED OR DISABLED
P REG	BIT NUMBER AND DEFINITION			JOB MODE	MONITOR MODE	JOB MODE	MONITOR MODE	JOB OR MONITOR MODE
P	48	Privileged Instruction Fault	Mon	TRAP	TRAP	EXCH	HALT	These mask bits are permanently set.
P	49	Unimplemented Instruction	Mon	TRAP	TRAP	EXCH	HALT	
P	50	Free Flag	User	TRAP	TRAP	STACK	STACK	
P+	51	Process Interval Timer	User	TRAP	TRAP	STACK	STACK	
P	52	Inter-ring Pop	Mon	TRAP	TRAP	EXCH	HALT	
P	53	Critical Frame Flag	Mon	TRAP	TRAP	EXCH	HALT	
P+	54	Keypoint	User	TRAP	TRAP	STACK	STACK	
P	55	Divide Fault	User	TRAP	TRAP	STACK	STACK	STACK
P	56	Debug	User	TRAP	TRAP	STACK	STACK	STACK
P	57	Arithmetic Overflow	User	TRAP	TRAP	STACK	STACK	STACK
P+	58	Exponent Overflow	User	TRAP	TRAP	STACK	STACK	STACK
P+	59	Exponent Underflow	User	TRAP	TRAP	STACK	STACK	STACK
P+	60	F. P. Loss of Significance	User	TRAP	TRAP	STACK	STACK	STACK
P	61	F. P. Indefinite	User	TRAP	TRAP	STACK	STACK	STACK
P	62	Arithmetic Loss of Significance	User	TRAP	TRAP	STACK	STACK	STACK
P	63	Invalid BDP Data	User	TRAP	TRAP	STACK	STACK	STACK

Figure 1-9. User Condition Register

A simple example is the occurrence of power failure warning with the processor in job mode which will be indicated by the setting of bit 50 in the MCR. This in turn causes an Exchange to CYBER 180 monitor which in turn allows the processor to take the appropriate action in view of imminent shutdown. A process generated condition such as Arithmetic Overflow (UCR 57) on the other hand may simply be ignored or stacked by the process. Figure 1-10 illustrates the different methods of transition between various CYBER 180 processes. The CYBER 180 Monitor initiates task A which in turn CALLS task B which at a later point initiates a RETURN to task A. At some other point in time a condition occurs in task A which causes a TRAP to the trap handler which subsequently returns control to task A.

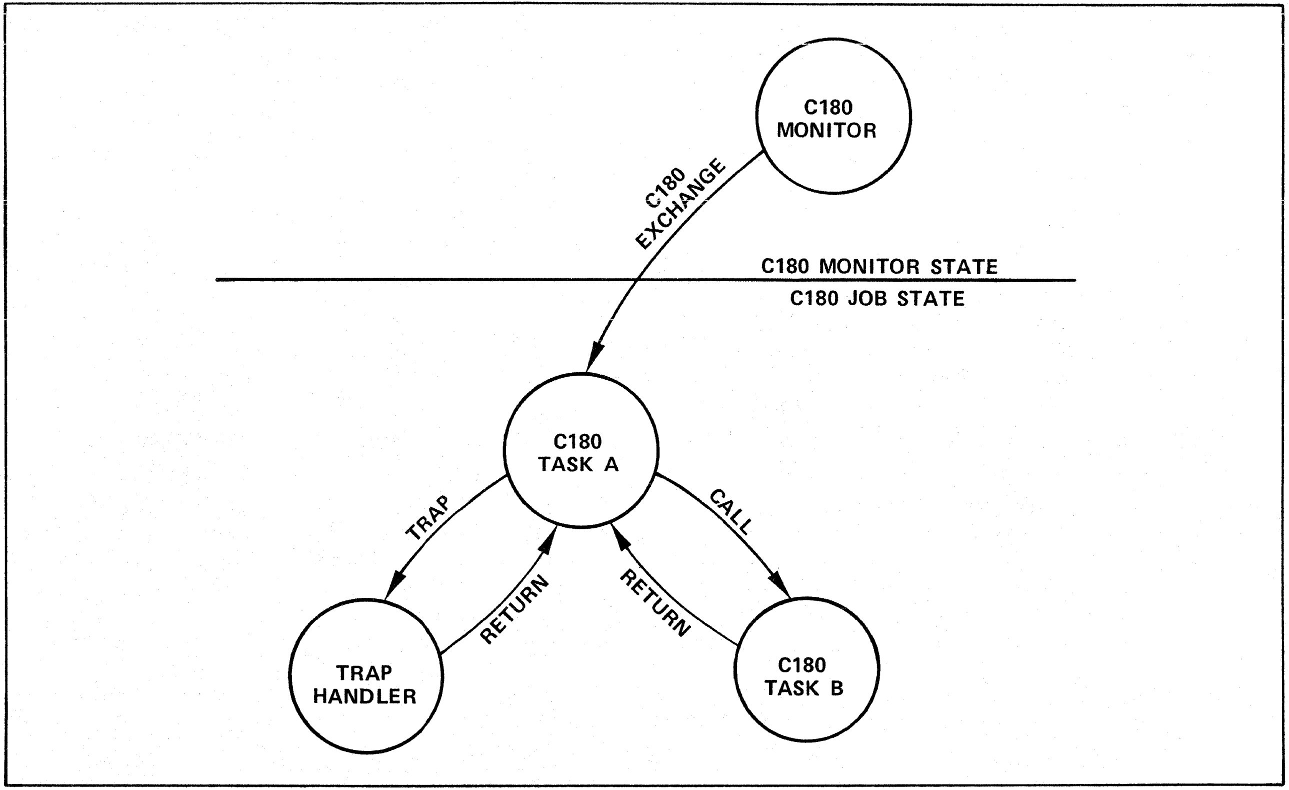


Figure 1-10. Transitions Between CYBER 180 Tasks

Figure 1-11 illustrates the initiation of task D which CALLs task B which in turn CALLs task C which subsequently passes control back to task D through a series of two RETURN operations. Note that task B may be shared by A and D.

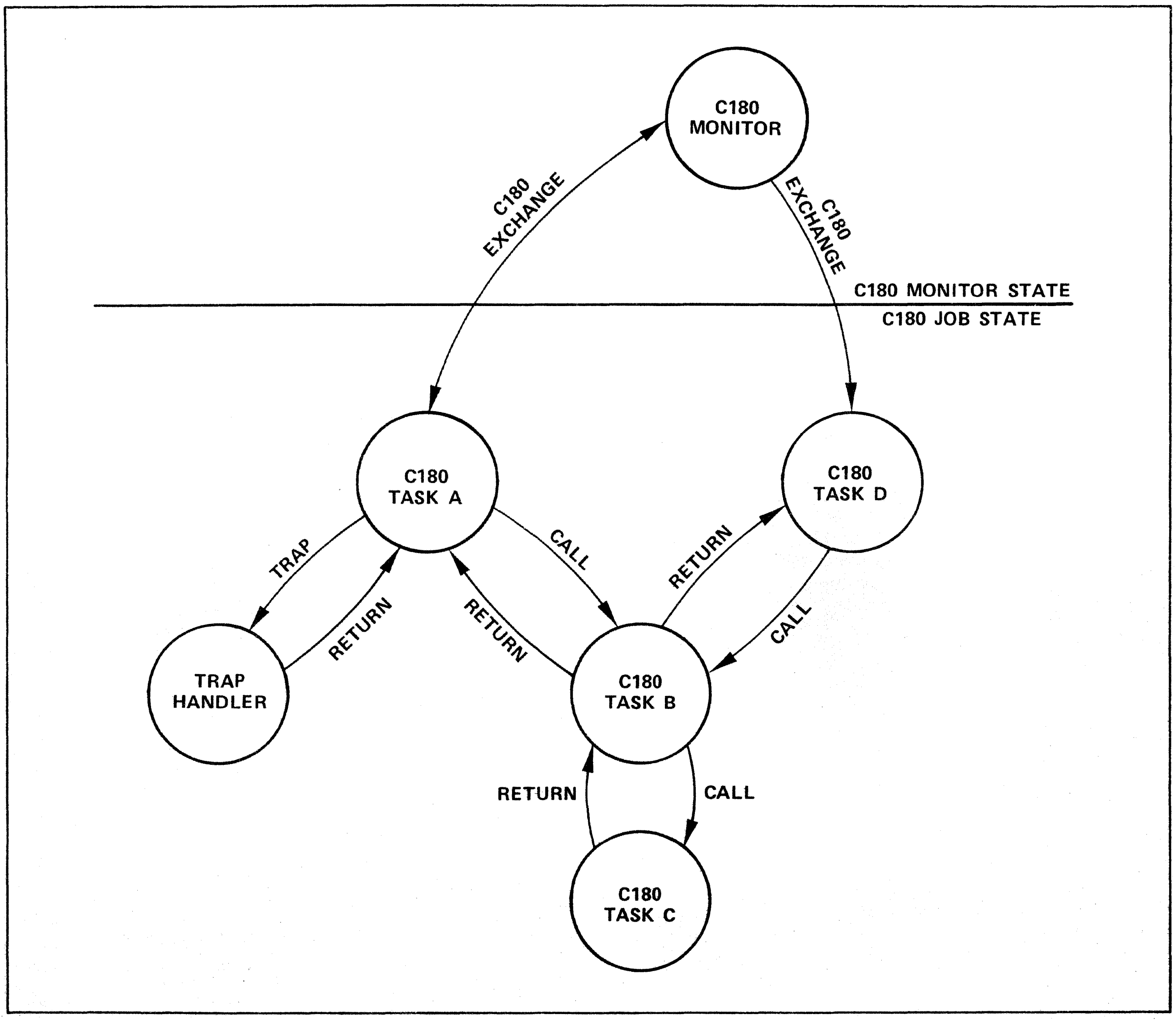


Figure 1-11. Multiple Level Call

C170 SUPPORT

The ability to execute CYBER 170 software on the CYBER 180 mainframes is an important part of the CYBER 180 strategy. The CYBER 170 enviroment is supported somewhat like a special purpose CYBER 180 job as illustrated in figure 1-12. Both CYBER 170 monitor and job state exist within the CYBER 180 job state. CYBER 170 state is not allowed to exist within CYBER 180 monitor state. Transitions between CYBER 170 monitor and CYBER 170 job do not require any intervention by a CYBER 180 task. The CYBER 170 Exchange Requests from the PP´s and CYBER 170 Central Exchange Jump to MA cause CYBER 170 exchanges directly from CYBER 170 job to monitor. CYBER 170 Central Exchange Jumps to Bj+K cause CYBER 170 exchanges directly from CYBER 170 monitor to job.

While the CYBER 170 Exchange Requests from PP´s are handled within the CYBER 170 environment, all other interrupts will cause a transition (TRAP or Exchange) to the CYBER 180 state. There is a CYBER 180 task named Executive Interface (EI) which is directly involved in the support of the CYBER 170 environment and handles any traps from within the CYBER 170 environment. For example, the CMU instructions are not executed directly in the CYBER 170 environment but instead cause a Trap to EI which simulates the CMU operation and then executes a Return to the subsequent CYBER 170 instruction.

Thus, the CYBER 170 environment within the CYBER 180 mainframes consists of the hardware plus EI and the CYBER 180 monitor.

There are several extensions to the CYBER 170 processor instruction set and architecture to allow a 2-million word central memory (the instruction space of a specific job still being limited to 131K in CYBER 170). These are extensions rather than modifications, thus, CYBER 170 user jobs that ran under NOS or NOS/BE are directly transportable to the CYBER 180 mainframes with NOS or NOS/BE installed. The operating systems are basically identical to those running on CYBER 170 mainframes with minor changes to take advantage of the extensions. There is also the capability to declare part of central memory as Unified Extended Memory (UEM) and thus to access it via the 011, 012, 014, 015 instructions in a manner analogous to ECS on the CYBER 170´s and LCME on the CYBER 176. This 2-million word address space, however allocated (CM or UEM), is one CYBER 180 segment with the ASID of FFFF).

The PPs are capable of running dual state also. When executing 12-bit instructions and using the 12-bit CYBER 170 channels, the PP´s are directly compatible to CYBER 170 mainframes with the addition of a Relocation (R) register to allow access to the entire CYBER 170 2-million word address space. The PP when running 12-bit instruction may initiate 60-bit read or write operations with central memory. The 60-bit write operations will automatically cause the purge of appropriate cache entries where necessary. (The byte number plus the knowledge that 60-bit writes imply an ASID of FFFF allow this purge).

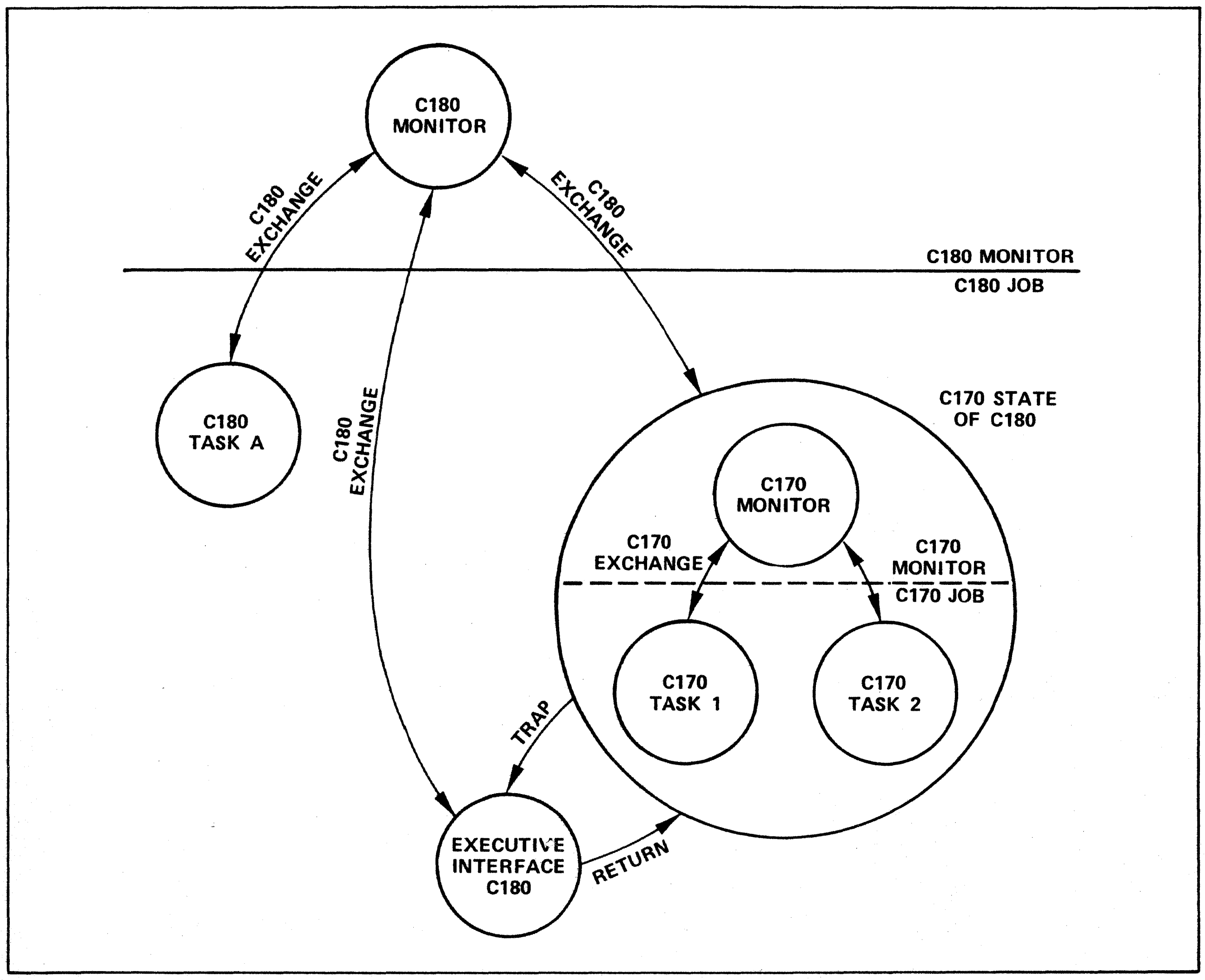


Figure 1-12. CYBER 170 Processor Environment

RAM

A key part of the CYBER 180 strategy is to achieve a significant step forward in the reliability, availability and maintainability of these systems. This has been an integral part of the hardware design process.

Reliability features are intended to reduce the number of hardware and software failures and reduce the risk of a minor component fault becoming a major equipment or system failure. Availability features provide alternate solutions to failures rather than immediate correction, thus allowing the system to remain available to users until a later time when proper corrections can be made. Maintainability features make the system easier to maintain by improving error isolation and ease of correction. Because these features sometimes overlap, there is no attempt made here to separate them according to reliability, availability, or maintainability. Some of these features are:

- Confidence level tests run against critical system elements during system initialization.
- Maintenance registers to set certain conditions and report error status.
- Parity checking on major data and address paths.
- SECDED in central memory.
- Reconfiguration of central memory to bypass a failing area.
- Reconfiguration of peripheral processors to bypass a failing processor.
- On-line diagnostics that can be run concurrent with customer usage.
- An engineering file to predict potential failures.
- Isolation diagnostics to narrow down the cause of failure.
- Remote technical assistance.

CYBER 170/180 SIMILARITIES/DIFFERENCES SUMMARY

CPU

CYBER 170	CYBER 180
60-bit word	64-bit word
Word addressing	Byte/word addressing (8 bytes/word, byte = 8 bits)
8 X registers (60 bits) 8 B registers (18 bits) 8 A registers	16 X registers (64 bits) No B registers 16 A registers (48 bits) (store/load instructions)
1´s complement arithmetic	2´s complement arithmetic in CYBER 180 State, 1´s complement arithmetic in CYBER 170 State
CYBER 170 instruction set	CYBER 180 + CYBER 170 instruction set mode bits Virtual Machine Identifier (VMID) in CYBER 180 exchange package enables CYBER 180 or CYBER 170 State Instructions
Register-to-register operations	Register-to-register operations
Some character handling instructions	Full set of character handling instructions
	Vector instructions (memory to memory) on high-performance models

MEMORY

CYBER 170	CYBER 180
Maximum 131K word user address space	4096 times 2**31 byte user virtual address space
RA/FL relocation	Hardware-segmented memory (maximum 4096 segments per user address space)
17-bit word address within RA/FL defined address space	Two-part virtual address-segment number (12 bits)-signed byte offset into segment (32 bits) space
262K maximum system executable memory	64-Mbyte (potentially 2**31 byte) executable real memory 2,147,483,648
Memory moves + swapping to manage memory	Hardware paged + swapping to manage memory

PPs

CYBER 170	CYBER 180
Up to 2x10 12-bit PPU´s	Up to 4x5 16-bit PP´s
Up to 2x12 12-bit channels	Up to 6x4 12/16-bit channels
Executes 12-bit PPU code	Executes 12-bit, 16-bit, or mixture, PP code (upward compatible with CYBER 170)
Memory size is 4K x 12 bits	Memory size is 4K x 16 bits
12-bit wide data channels	12-and/or 16-bit wide data channels
60-bit access to central memory to central memory	64-(4x16 bits) and 60-(5x12 bits) bit access
18-bit central memory address for PPU read/write	28-bit central memory address for PP read/write
Real central memory addressing	Real central memory addressing
500-ns major cycle time	250-ns major cycle time
16-words long deadstart panel	16-words long deadstart panel + 512-word read only memory usable at deadstart

SYSTEM

CYBER 170	CYBER 180
No shared memory among user address spaces (defined by its RA/FL)	Segments sharable among user address spaces (code and data sharing possible)
Code/data mixed within user´s address space	Segments can be read/write/ execute, or combination - globally sharable code
Exchange operation to go to CPU Monitor	Exchange operation to go to CPU Monitor
CPU supports CYBER 170 instruction set	CPU supports coexisting CYBER 180 and CYBER 170 instruction sets. VMID field, within CYBER 180 exchange package, is used to switch between CYBER 170/180 State instruction sets. The CYBER 170 environment for CYBER 170 NOS or CYBER 170 NOS/BE is established within the CYBER 180 job space and then state switching may be accomplished by an exchange or trap operation and Call or Return instructions. CYBER 170 external interrupts are supported and handled within the CYBER 170 environment.
System runs at System Control Points or in PPU´s, CPU Monitor routes RA+1 requests	Most CYBER 180 system code runs within user address space and obeys the same calling calling, loading/linking conventions.
	-Levels of system code are protected by a hardware supported hierarchical ring mechanism from less capable code modules (15 ring levels are provided).
	-System code can be directly called by RETURN Jump like CALL instruction without software assist.
Subsystems (that is, Telex, Magnet, Data Manager, and so forth) are protected by RA/FL mechanisms from each otheror user, can be called only via CPU Monitor	Subsystems are protected by hardware supported (key/lock) mechanisms from each other, directly callable by user code without software assist

CYBER 180 HARDWARE SUMMARY

CONFIGURATION

Currently four CYBER 180 systems are in the design/implementation phase:

- S1 (P1 processor, M1 memory, I1 IOU)
- S2 (P2 processor, M2 memory, I2 IOU)
- S3 (P3 processor, M3 memory, I2 IOU)
- THETA (processor and memory, I2 IOU)

Performance range (single processor);

P1 approximately 1 x CYBER 172, Theta approximately 36 x CYBER 172

Dual, symmetric multiprocessing in CYBER 180 state

One, maximum 20 PP, I/O unit per system

Maximum memory	M2 - 16 Mbytes M3 - 32 Mbytes
Memory ports	M2 - 4 ports each 64-bits wide M3 - 4 ports each 64-bits wide

CONSTRUCTION

PP´s microcoded via ROM control store, 5 per barrel

Processors microcoded via RAM control store

P2 - ECL10K logic

P3/Theta - LSI and F100K logic

Convection/Freon cooled

MAINTENANCE

Dedicated maintenance PP termed MCU - Maintenance Control Unit

8 bit channel protocol to a maintenance channel (that is, 8 bits of data within the rightmost 8 bits of PP´s word is used only)

- loads processors´ microcode.
- read/write processor´s/memory´s maintenance registers.

MEMORY

Phased

SECDED error correction

16K chips in M2 and in M3

Memory functions:

- to route interrupts from
  processor to processor
  or
  PP to processor (new 16 bit instruction)
- to interlock memory

Byte (8 bits) addressable

Top 4 bits of CYBER 180´s 64 bit memory word are not used in CYBER 170 State

PROCESSORS

32 program accessible registers

16 X registers (also used for indexing)
16 A registers (48-bit logical/virtual address)

Exchange package

- Address of CYBER 180 CPU Monitor´s exchange package (MPS) is set at Deadstart.
- Job Mode Exchange Package address (JPS) is set by CYBER 180 CPU Monitor.

Exchange Interrupts initiate exchange to CPU Monitor

Trap Interrupts initiate partial exchange within job´s Address Space and certain registers are saved at the address contained in the A0 register.

Fast cache per processor

- Not cross connected to other processor, must be software managed in CYBER 180 State.
- PP initiated central memory writes invalidate corresponding CPU´s cache entries on 60-bit transfers.

VIRTUAL MEMORY/PROTECTION

A logical/virtual address consists of three parts:

- 31-bit byte offset into a segment.
- 12-bit segment number, this is a word offset to a Segment Descriptor, within the Segment Table defined by an Exchange Package.
- 4-bit ring number (refer to the following explanation).

A logical address is translated to a real memory address through a one-per-address-space Segment Table (its real memory address is defined by words 34 and 35 of an exchange package) plus a one-per-system global Page Table. CYBER 180 CPU Monitor runs within its own Address Space and can only access a small part of the physical memory. No direct real memory addressing.

Protection is on a segment basis: read/write/execute plus rings/keys (refer to the following explanation).

A special Binding Section is used to transfer control between separately compiled code modules. The new P address and the address to the list of the called code module´s respective Binding Section entries is extracted by a CALL instruction from the Binding Section. The new Binding Section´s address is left in register A3 on completion of the CALL instruction initiated transfer.

CALL instruction saves the current execution environment in virtual memory at the address within A0 register. The number of registers saved is specified by a mask in X0 register. The CYBER 180 RETURN instruction is used to return to the previous execution environment and restore the saved registers.

All linkage information local to an Address Space is placed into a code module specific Binding Section by the CYBER 180 Loader (for example, address of common and data sections, pointers to external variables, linkage for external entry points) therefore, the code module remains globally sharable since it does not contain any address space specific local information.

Hierarchical protection among segments within an address space is by a hardware supported ring mechanism.

- Instruction counter contains a 4-bit ring number (that is, it is a logical address).
- this ring number can only be changed by a CALL, RETURN, or EXCHANGE instruction initiated transfer.
- Segment Description Table Entry contains two ring numbers; R1 and R2.
- Hardware checks P against R1 and R2 when executing code from a segment and tests the Ring Number of the A register against R1 (RN ≤ R2) for a valid write access and against R2 (RN ≤ R2) for a valid read access.
- Interrupt to CPU Monitor, with Access Violation Monitor condition bit set when incorrect match is detected, perform memory access operation when everything is satisfactory.

Keys provide nonhierarchical isolation between segments, within one address space, at one ring level of protection. Majority of subsystem code (like Telex, Magnet, Data Manager, Cobol Message Control System) runs in User Job Address Spaces. Without keys/locks, an error within one subsystem could damage global data owned and managed by another subsystem within the same hierarchical level of ring protection.

The one-per-address-space Segment Table and the system global page table are not accessed on every memory reference. Once a logical/virtual address is translated to a real memory address it is saved in a small one-per-processor Segment/Page Map and reused on any subsequent access to the same address.

PERIPHERAL PROCESSORS

Upward compatible with CYBER 170 PPU´s.

16-bit memory word

- CYBER 170 instructions and 12-bit mode data are contained within the rightmost 12 bits.
- 12/16-bit mode instructions are differentiated by:
  leftmost bit = 0 for CYBER 170 compatible 12-bit instructions.
  = 1 for new CYBER 180 16-bit mode instructions.

60 and 64-bit mode central memory read/write instructions

- CYBER 170 compatible 60-bit transfers from/to 5x12-bit PP words. Top 4 bits of the 16-bit PP word are set to zeros.
- new 64-bit mode instructions transfer to/from 4x16-bit PPU words.

28-bit central memory address

- central memory addressing is by real memory addresses.
- new relocation register which contains a base address which is added to a relative, 18-bit central memory address to form an absolute address. New 12-bit instructions to load and store it.

12 and 16-bit external channel interfaces (hard wired), if a 12-bit external channel interface is used on a channel the most significant 4 bits of the internal channel word are cleared.

Packed input/output instructions transmit 3x16-bit memory words to/from 12-bit external devices as four channel words.

New 16-bit instruction to interrupt CYBER 180 CPU´s, old 26X instruction is still provided for controlling the execution of the CPU in CYBER 170 State.

CYBER 170 STATE OPERATION

Microcode and hardwired logic (that is, in addition to that required for the execution of CYBER 180 State instructions), depending on the processor model, provide coexisting support for both CYBER 180 and CYBER 170 instructions within a CYBER 180 CPU.

CYBER 170 PPU´s are upward compatible with PP´s in the CYBER 180 I/O Unit. CYBER 170 compatible 12-bit instructions are contained within the lower 12 bits of the new 16-bit PP memory word.

CYBER 180 hardware is capable of supporting several variant CYBER 170/180 Dual State User/System Operational Environments. For illustrative purposes, this section assumes a Cooperative Operational Environment since this mode requires the full support of CYBER 180´s Dual State hardware.

CPU State Switching

VMID within a CYBER 180 exchange package enables CYBER 180 or CYBER 170 State instructions

VMID = 0 for CYBER 180.
       1 for CYBER 170.

CYBER 180 CPU Monitor can directly manipulate VMIDs.

CYBER 180 or CYBER 170 environment is established by either

- exchanging, with the correct VMID set for CYBER 180 or CYBER 170 State, from CYBER 180 Monitor to CYBER 180 Job Mode

or

- executing a CALL instruction within a CYBER 180 Job's Address Space and transfer control directly to the CYBER 170 environment

or

- trapping, which may establish VMID=0 from either CYBER 170 or CYBER 180 State but may not establish a VMID=1

or

- executing a CYBER 180 RETURN instruction. The target environment is established by restoring the register images contained in CYBER 180 virtual memory at the address within A2 register, (that is, for VMID=0 return is into CYBER 180 State, for VMID=1 return is to CYBER 170 State).

The target CYBER 170 exchange package must reside within CYBER 180's A and X registers for the CALL initiated mode switch, or contained within the CYBER 180 exchange package for the first alternative, previous to initiating the mode switching operation or within the Stack Frame Save Area for Return. The CYBER 170 State P address and the new VMID (that is, VMID=1) are both extracted from the Binding Section for the CALL mode of switching.

CYBER 170 Central Memory Image

One CYBER 180 segment contains the CYBER 170 memory image. The ASID for this segment is FFFF.

Pages of the CYBER 170 memory segment must be at consecutive memory addresses, starting at Real Memory Address zero.

All CYBER 170 addresses are relocated by the RA register defined by the active exchange package and used as offset into the CYBER 170 segment. CYBER 180's virtual memory mechanisms (that is, segment and page table) are used to translate this logical/virtual address into real memory address.

Pages corresponding to the CYBER 170 segment must have page descriptors in the CYBER 180 Page Table.

CYBER 170 memory segment is managed by a minimally altered NOS or NOS/BE operating system.

Central Memory Extended (CME)

Maximum memory size available to the system is two million words.

RAc and FLc are 21-bit registers which are used to address the memory.

Maximum memory available to a single job is 131K words (17-bit address).

Central memory extended is contained within a single contiguous segment along with CYBER 170 central memory.

Soft Extended Core Storage (ECS)

Maximum size of soft ECS and central memory (including CME) is two million words.

RAe and FLe are used to allocate soft ECS to a job.

Block copy instructions (011,012) and single word load/store instructions (014,015) are used to reference soft ECS.

Soft ECS is contained within a single contiguous segment along with CYBER 170 central memory.

CPU Management

Unified CPU Dispatcher assigns processor, within CYBER 180 or CYBER 170 CPU Monitor, to the highest priority CYBER 170 or CYBER 180 Control Point.

One CYBER 180 Address Space/Control Point is defined as the CYBER 170 Emulator Job. It is dispatched by the CYBER 180 CPU Monitor when either

- the currently highest priority Control Point is executing in CYBER 170 State

or

- a PP initiated Exchange Request Monitor Condition is detected (that is, bit 53 of the Monitor Condition register is found set by the CYBER 180 CPU Monitor).

CYBER 170 CPU Monitor (NOS or NOS/BE variant) services CYBER 170 State PP or RA+1 requests and either dispatches a CYBER 170 Control Point or returns to the CYBER 180 Monitor.

CYBER 170 Exchange Requests

PP´s initiated CYBER 170 State Exchange Request (that is, by a 26X instruction) either

- changes the CYBER 170 State Execution environment, as defined for a CYBER 170 processor (that is, enter CYBER 170 CPU Monitor within the CYBER 170 memory segment via standard CYBER 170 State exchange operations, or do nothing), while the CPU is executing within the CYBER 170 environment

or

- sets bit 53 of the Monitor Condition Register and halts the PP while the processor is in CYBER 180 State. Setting a bit in the Monitor Condition Register results in an exchange to CYBER 180 CPU Monitor which immediately dispatches the Emulator Job. CYBER 170 State is thus entered and exchange to the CYBER 170 CPU Monitor takes place, the requesting PP is also restarted by the hardware after the A170 exchange, if necessary, takes place.

To understand CYBER 180 completely, it is necessary to get a firm grasp of the three main areas:

Virtual Memory Mechanism

Interrupt System

Call/Return Mechanism

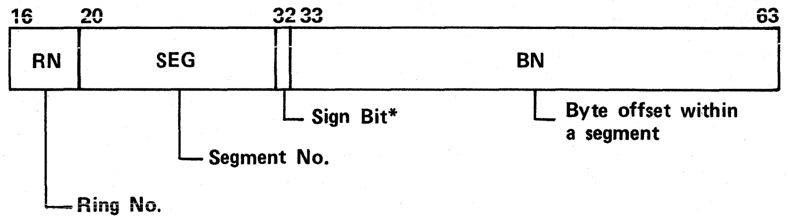
Until a firm comprehension of all three of these areas has been obtained, it is not possible to tie them together and appreciate the whole.

This section deals with the basic concepts of the CYBER 180 virtual memory mechanism. Virtual memory was originally conceived as a solution to the overlay problem. However, as technology has advanced, it has evolved into a solution to the security problem. This is the primary purpose of the CYBER 180 virtual memory mechanism. Each executing task operates in its own, unique address space which is divided into a number of segments. Each segment may be 2^{31-1} bytes (2 billion bytes) long. It will be seen later that it is the segment which forms the basis of the security and protection mechanisms. It is important to understand the difference between segments and pages. The segment is the unit of virtual memory management. It has attributes, such as length, access privileges and other features peculiar to the protection scheme as will be seen later. The page is the unit of real memory management. Pages do not have attributes. They are present in the hardware to assist the software (operating system) with the management of the very large real memories which may be supported by CYBER 180. The page size is a variable which is set during system initialization and is constant from one deadstart until the next.

The only addresses available to software are virtual memory addresses. CYBER 180 processors do not have a real memory address mode, and the only places they are used are in hardware tables used in address translation. This section describes the address translation mechanism. It is of interest to study this mechanism, and there are certain software responsibilities for the operating system to optimize the process. In the more general sense however, it is important to understand the role of segments, and what a proliferation of segments, which are all active concurrently, can do to the performance of the system. In addition, programmers must maintain good locality of reference. That is, all code that is being used at one time should be collected in one place (in virtual memory). Likewise, all data in use at one time should be collected in one place. The importance of this cannot be overstressed since the hardware depends on this good locality of reference to minimize address translation time.

ADDRESS TRANSLATION

The fundamental address, available to a programmer, is the PROCESS VIRTUAL ADDRESS (PVA). To the user, this appears as a segment number and a byte offset within the segment. It also includes a ring number which is part of the protection mechanism discussed later (figure 2-1).



*Serves to prevent address increments from exceeding the segment.

Figure 2-1. Process Virtual Address

The segment number is a 12-bit field which is used as an index into a Process Segment Table which is created by the operating system for each process (task(1)) which is active in the system.

(1) The terms process and task are synonomous. A task is the unit of execution of the CYBER 180 operating system. A process is the hardware term for a task.

Segment numbers are assigned sequentially from zero. This is not essential, but is natural and the hardware has been optimized assuming this to be the case. Segment descriptor table entries (SDE) are 64-bits long, and contain information relating to the privileges and protection of that segment. They also contain an ASID which is a 16-bit identifier used and created by the operating system which identifies each active segment uniquely on a system-wide basis.

Address translation takes place in two steps. The first step translates a PVA to a system virtual address (SVA). The process segment descriptor table (SDT) is used for this purpose. It will be seen later that the concept of the system virtual address is extremely important. It forms the basis for code sharing, and processor cache memories are organized on SVA´s - not real memory addresses.

In the first step of the translation the hardware takes the ASID from the entry in the SDT pointed to by the SEG field of the PVA. It then catenates this with the BN field of the PVA to form the SVA (figure 2-2).

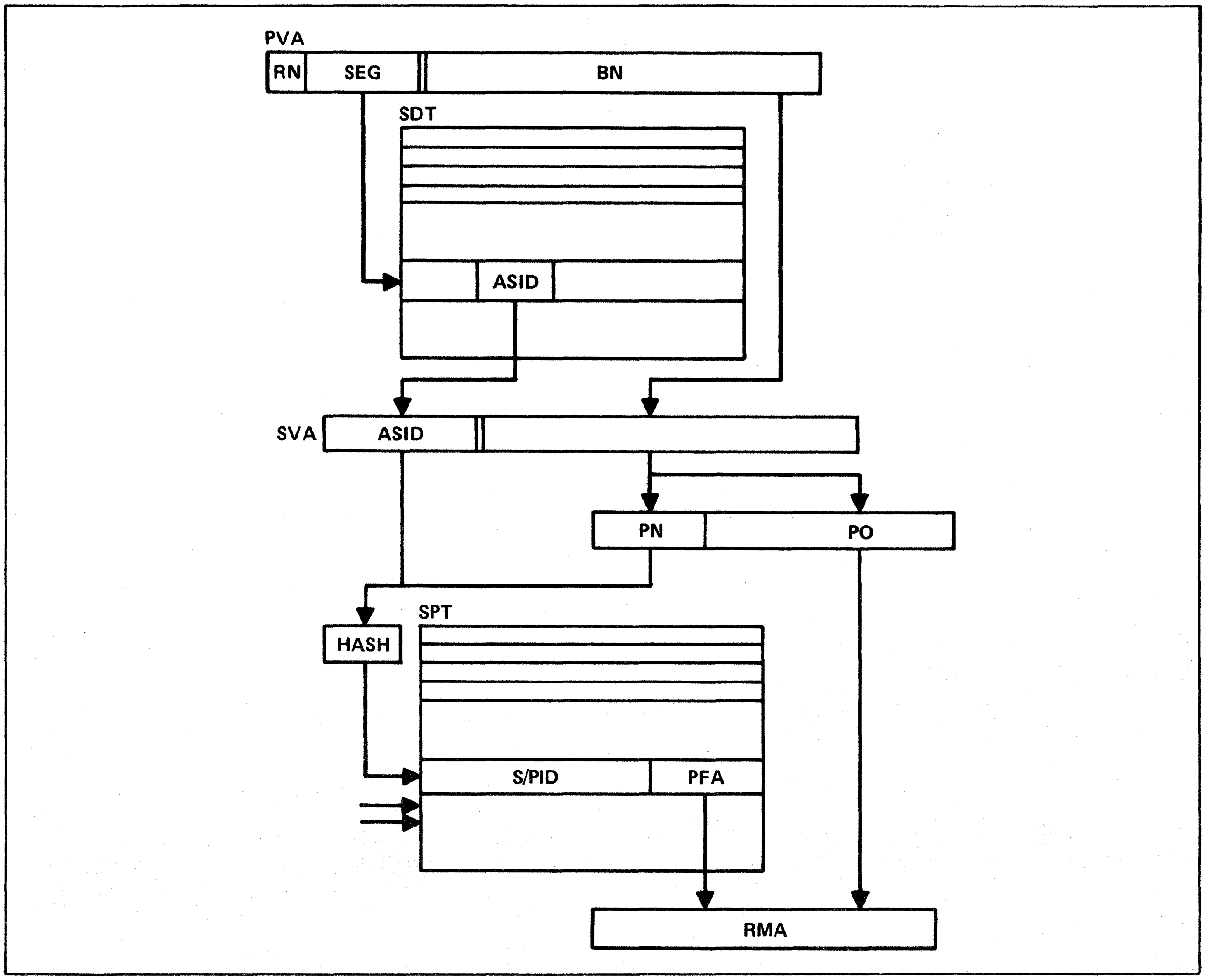


Figure 2-2. Address Translation

The processor views the BN as a Page Number (PN) and a byte offset within that page, or Page Offset (PO). To determine where (or if) the page resides in real memory a further access to the System Page Table (SPT) is made. Since many more pages exist in virtual memory than in real memory, a hashing algorithm is used to compute the page table index. On CYBER 180 the page number and the ASID are hashed via an exclusive OR; this is discussed in more detail later. The page table index is used to select a candidate entry from the system page table. The table is then searched forward linearly until a valid page with the desired entry is found, or until 32 entries have been searched. If the search terminates without a hit, the page is assumed not to be in central memory and a page fault is indicated.

Once a hit is made, the page frame address (PFA) in the page table is catenated with the page offset from the SVA to form a 32-bit RMA. Figure 2-3 illustrates the formation of PN and PO for a 4096 Byte Page.

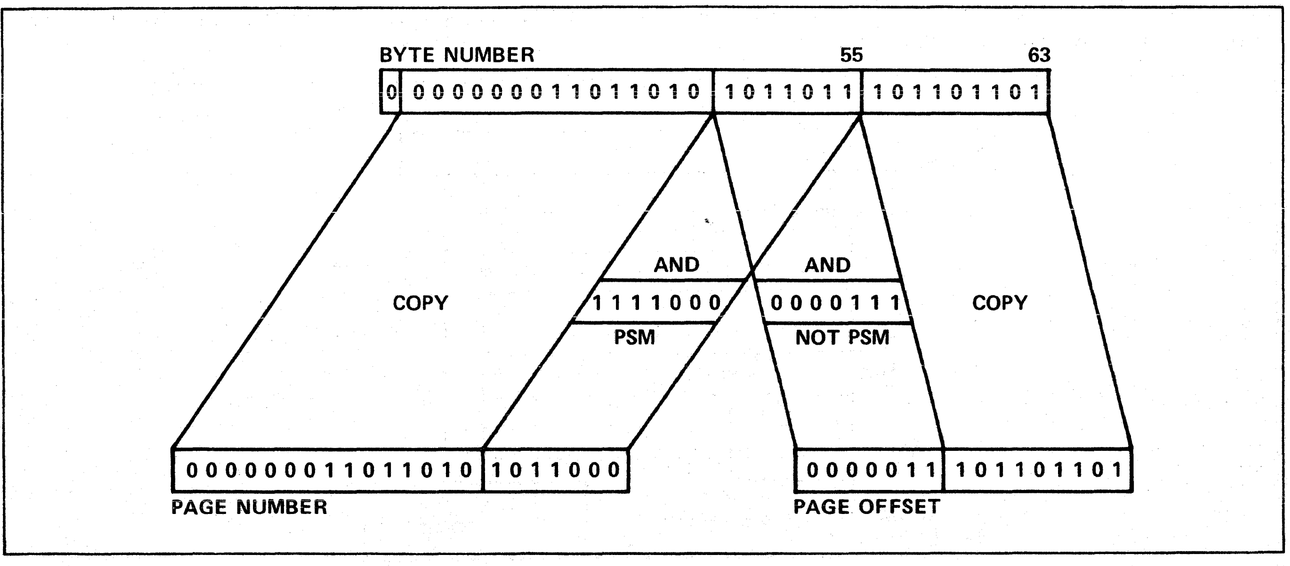


Figure 2-3. Formation of Page Number and Page Offset (For a 4096 Byte Page)

PAGE SIZE

The page size is determined by the value of the Page Size Mask (PSM). This is a 7-bit register which expresses the page size in multiples of 512 bytes such that the page size is given by:

$$\text{page size} = 2^{9} \times 2^{7-(+/PSM)}$$

Where the PSM is a solid mask extending from left to right. A PSM of zero indicates the largest page size (64KB). The term (+/PSM) expresses the summation of the one bits (Pop Count) in the PSM.

HASHING ALGORITHMS

The hashing algorithm takes an exclusive OR of the low-order 16-bits of the page number and the ASID. If the page number is only 15-bits long, then a zero is catenated to the lefthand end to make it 16-bits. Since the resulting hash must be as random as possible, the most random low order bits of one quantity should be exclusively OR´ed with the most random high order bits of another. The low order 16-bits of the PN are the most random part of that quantity and the same will be true of the ASID if it is assigned sequentially starting from zero. This is not desirable and it is incumbent upon the operating system to ensure the appropriate randomness in the ASID. This may be achieved by assigning ASID´s from zero on up and then inverting the bits. Hence, the first 16 ASID´s should be:

	HEX	BINARY
1	0000	0000 0000 0000 0000
2	8000	1000 0000 0000 0000
3	4000	0100 0000 0000 0000
4	C000	1100 0000 0000 0000
5	2000	0010 0000 0000 0000
6	A000	1010 0000 0000 0000
7	6000	0110 0000 0000 0000
8	E000	1110 0000 0000 0000
9	1000	0001 0000 0000 0000
10	9000	1001 0000 0000 0000
11	5000	0101 0000 0000 0000
12	D000	1101 0000 0000 0000
13	3000	0011 0000 0000 0000
14	B000	1011 0000 0000 0000
15	7000	0111 0000 0000 0000
16	F000	1111 0000 0000 0000

and so on. Another technique which could be used by the operating system is to use a pseudo-random number generator for ASID assignment.

The result of the hash is AND´ed to the Page Table Length (PTL) and four zeros catenated to form the page table index (figure 2-4).

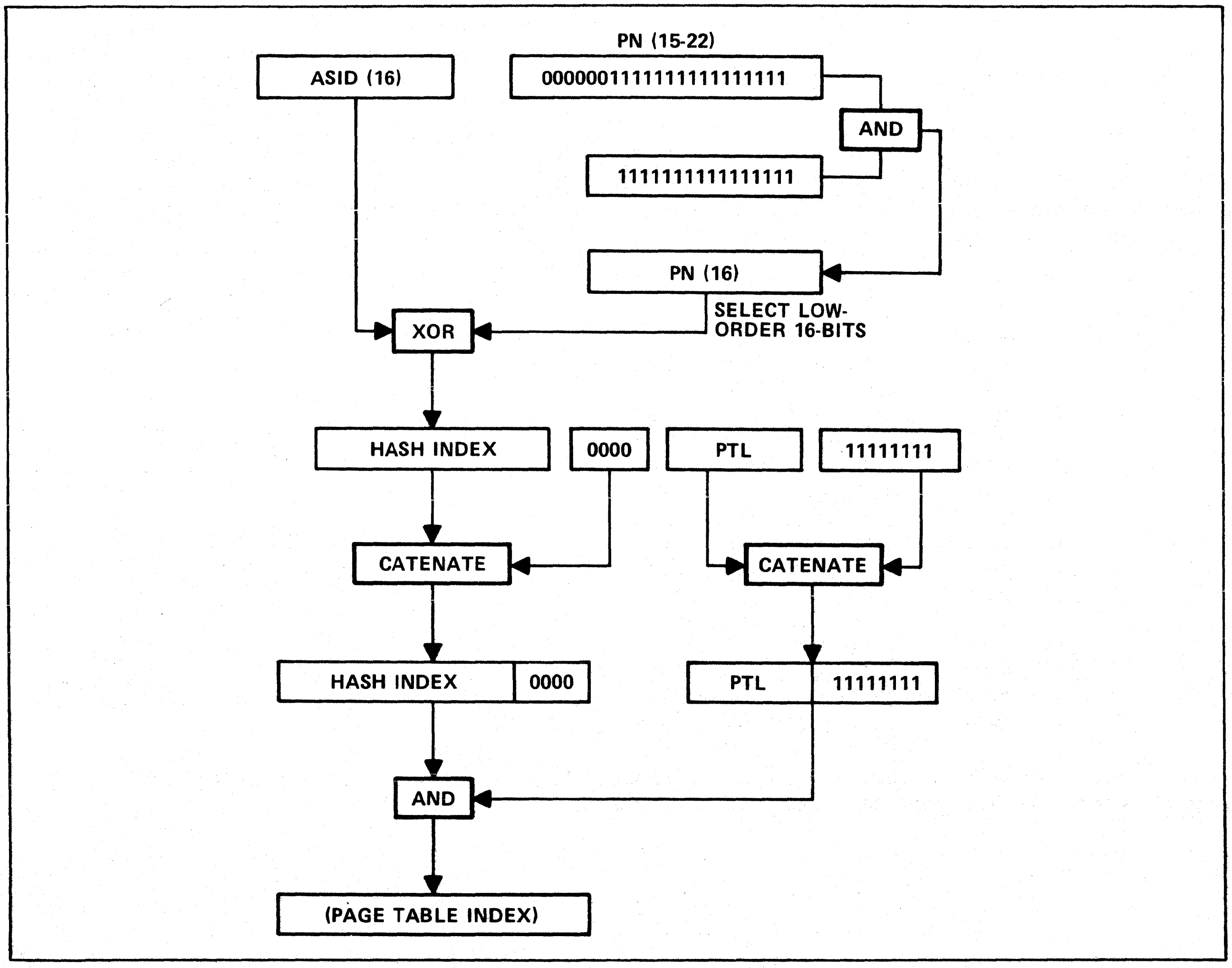


Figure 2-4. Hashing Algorithm

PAGE TABLE SEARCH

Since the hashing algorithm is a many-to-one mapping it is entirely possible for two different pages to hash to the same page table entry (PTE). To ensure that the correct entry has been found the ASID and Page Number portion of the SVA is compared with the System Page ID held in the PTE. If they do not compare equal, then a linear search is initiated which is controlled by bits 0 & 1 of the PTE. The search continues until 32 entries have been searched, the correct entry found, or until an end of search condition is indicated by bit one (figure 2-5).

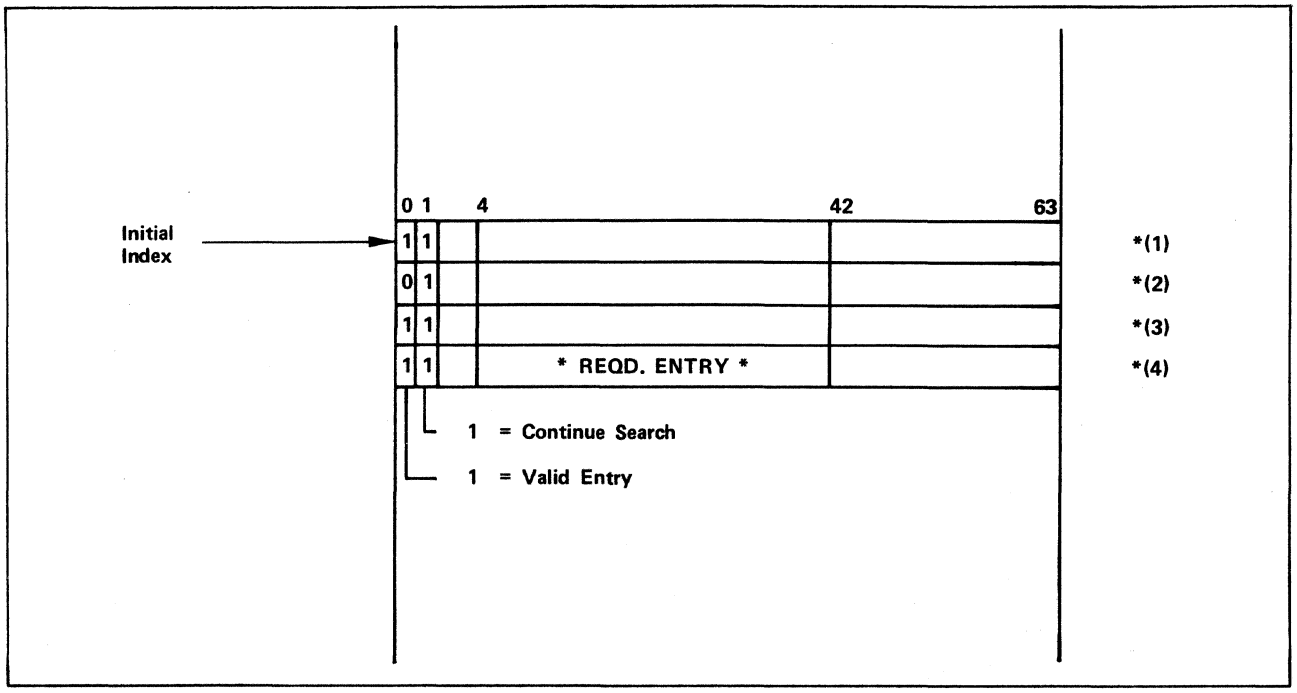


Figure 2-5. Page Table Search Example

*(1) The first entry accessed is valid (bit 0=1) but has the wrong Segment Page Identifier (SPID). Since bit 1=1 the next entry is checked.

*(2) The second entry is invalid but search can continue.

*(3) The third entry is the same as the first.

*(4) The fourth entry matches the required SPID. The sequence terminates at this point. The continue bit (bit 1) does not necessarily indicate an end of search at this point since other multiple entries with this hash or an adjacent hash may be present.

The algorithm for setting the continue search bit is self-evident. When an entry is invalidated its continue bit is checked. If it is set then no further action is necessary since it is part of a chain to an entry further down in the Page Table. If it is zero, then the table may be searched backwards to clear out possible continue bits for the, now, invalid entry. If the previous entry had its continue bit clear, then the process terminates since there is no chain to investigate. If the continue bit was set, then it is cleared and a check made to determine whether further continue bits can be cleared. Conditions for further clearing are: an ASID of zero (a null entry by software convention); an ASID of nonzero which hashes directly to this entry - in which case the continue chain being cleared could have started higher up (figure 2-6). Note that a special system instruction (Load Page Table Index) has been defined to aid in this process. This instruction is described in a later section.

In summary, the hardware uses the Segment Descriptor Table to translate a PVA into an SVA. It then uses the System Page Table to translate the SVA into an RMA. The SDT and the SPT are hardware tables which are constructed and managed by software. Naturally, if every reference to memory required at least two additional memory references, the processors would execute extremely slowly. It will be seen later on that a number of hardware buffers are utilized to eliminate this overhead.

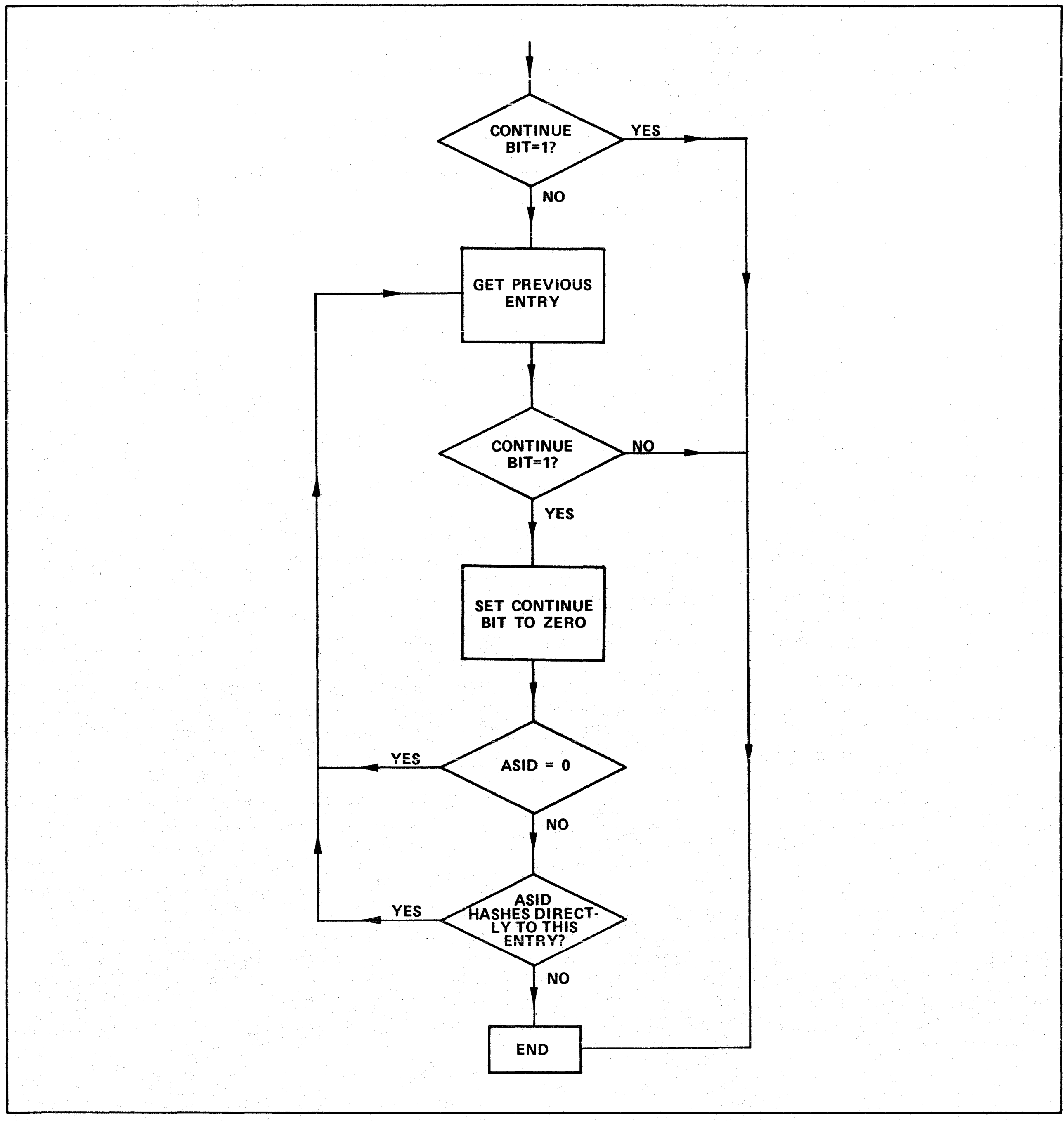


Figure 2-6. Page Table Search Flowchart

It is assumed that the Operating System assigns 2 to 4 times as many entries in the page table as there are pages in real memory. This is to accommodate coincident hash indexes with the minimum search. Since the general environment contains two processors care must be taken when changing page table entries and the special interlock instructions must be used for this purpose. Details of this usage can be found in a later section which deals with the system instructions.

The only mode of operation of the hardware is a virtual address mode. There are no instructions which deal directly with real memory addresses. The hardware has been designed with dynamic paging in mind. That is pages are brought into memory on a demand basis and page table entries are purged based on a least recently used (LRU) algorithm. This algorithm is the responsibility of the operating system. However, two flags are kept in the PTE to help in the process. These are kept in the VM field (bits 2-3) and have the following meaning:

(i) Whenever a page is used (read, written or executed) the hardware sets bit 2 in the PTE.

(ii) Whenever a page is modified (written) the hardware sets bit 3 in the PTE.

Combinations of bits 2-3 have the following meanings:

00 - New page, unused and unmodified
01 - Unused but modified (see note below)
10 - Used but not modified
11 - Used and modified

Pages are chosen as candidates for purging based on the value of this VM field and their LRU status. Since any page which has been modified must be written to mass storage when it is purged, modified pages will typically have a higher resistance to purging. The status unused but modified can arise from software algorithms. The VM bits are never cleared by hardware. They are cleared by software when pages are purged and to force updates to the LRU status of all pages. In this latter mechanism, it is expected that the Operating System will periodically zero all used bits in the page table. This will effectively reset the LRU status. Ensuing activity will automatically update this status.

Although the hardware has been designed with dynamic paging in mind, it is not a prerequisite. In particular, when running in a pure CYBER 170 State, static paging will be used. The entire CYBER 170 environment will be assigned to a single CYBER 180 segment which operates in CYBER 180 job mode. Pages in this segment and in real memory have a one-to-one correspondence, and once initialized, the page table will not change. CYBER 170 will operate in a virtual memory segment which has a size corresponding to the amount of real memory in the system. The ASID is set to HEX FFFF and SEG to zero, a pseudo RMA mode will exist within the hardware. Note that this is a pseudo mode since the hardware will still go through the address translation mechanism. However, there will be no page faults.

Security is a subject, particularly as it pertains to computer systems, which is rapidly gaining in importance. Being cognizant of this fact, CYBER 180 has been designed to meet the most stringent security requirements of the industry. The degree of security achieved by a CYBER 180 system is controlled by the software exploitation of the hardware features. The hardware has been designed to provide facilities which will detect breaches of security, or attempted breaches of security. However, it is the software which really controls the desired level of security. If a rigid set of conventions is not followed, then loopholes will exist which will no doubt be detected by ingenious users. The way in which the software and hardware must play together is similar to other disciplines which must be followed if a system is to be totally secure. These disciplines embrace the installation management, the operators, the administration. In fact they embrace the entire organization. The computer is only one small, albeit important, part of this whole.

The responsibilities of the organization are not discussed here. Instead the discussion is confined to the hardware and software facilities provided by CYBER 180 systems. It will be divided into two major areas: the first deals with the software facilities and their interfaces to the end user, and the second deals with the hardware facilities and their use by the software.

SOFTWARE FACILITIES

ACCESS CONTROL

A basic objective of NOS/VE is to provide efficient and safe services to multiple users simultaneously and asynchronously. Levels of service to be provided range from complete isolation of users from each other to controlled sharing between cooperating users. In order to allow this range of service levels, the system has adopted a general access control strategy or security model which serves as the conceptual basis for the detailed implementation of all the access control mechanism in the system.

The access control strategy is based on a conceptual access control matrix. Rows of the matrix represent all possible users of the system. In the access control matrix these are called subjects. Columns of the matrix represent all possible system resources that can be accessed by a subject. In the access control matrix these are called objects, such as subjects, files, and equipment. Each element in the matrix identified by a subject-object pair contains the valid kinds of access or access rights that the subject has to that particular object. Figure 3-1 illustrates access control.

	SUBJECT A	SUBJECT B	FILE C	FILE D	TAPE DRIVE E
SUBJECT A		ADMIN.	OWNER R,W		OWNER USE
SUBJECT B			R	OWNER R,X	

Figure 3-1. Access Control Example

In the example, subject A is the administrator of subject B, the owner of file C and of tape drive E. Subject A can read and write file C and use tape drive E. Subject B can read file C and owns and can read and execute file D. Every access any subject makes to any object is validated via the access control matrix. The access is permitted only if the corresponding access right is in the appropriate element in the access control matrix.

Obviously, the operating system cannot maintain a physical matrix which is consulted on every access. A variety of features of the system architecture interact to implement the conceptual access control matrix. Some of the major features of the NOS/VE implementation of the access control architecture are listed below:

User identification and validation

- A user must be known before gaining access to the system.
- The resources a user can use are a function of user controls, project controls and the current state of the system.
- An attribute of every user is the lowest ring number of execution.
- Modification to the user validation information may only be performed by the system, account and project administrators who control the user´s installation.

File system

- All files in the system, local or permanent, are owned by a single user.
- Access to permanent files by any other user besides the owner is regulated by an access control list that is associated with each file. The access control list contains the names and access rights of all users permitted to access the file.
- All files, local or permanent, have one or more ring brackets associated with them which are used as qualifiers to file access.
  - If a file is readable, then it possesses a read bracket which defines those rings in which it can be read.
  - If a file is writable, then it possesses a write bracket which defines those rings in which it can be written.
  - If a file is executable, then it possesses an execute bracket which defines those rings in which it may execute and a call bracket which defines those rings from which it may be called.

  The ring brackets associated with a file are specified by the owner of the file. However, the file system will not allow any user to specify any ring bracket of higher privilege than the ring in which the user is executing.
- All files, local or permanent, possess certain attributes that describe the contents of the file. The ring brackets associated with reading, writing, executing and calling are all file attributes. Whether a given user has read, write or execute permission to another user´s permanent file is determined by the access control list of the file. The combination of these two factors allow rings to be a unit of protection recognizable system wide. The reason this is useful to the operating system is that it wishes to discriminate between system code and nonsystem code running in a user job regardless of the user on whose behalf the job is executing. Since the user´s installation administrators control the assignment of ring numbers in the validation files, the user controls the extent and connotation of his installation ring usage. If an installation chooses to associate different rings with different security classifications, it may do so. If it wishes to run all users at a single ring, then the only use of rings will be to protect the operating system from users´ programs.

Segment management

- For a file accessed through the system virtual memory mechanism, the file protection attributes maintained by the file system are used by segment management to build the segment descriptor table entries used by the CPU address translation logic when referencing the segment. The attributes the file system software have maintained are continuously enforced by the hardware when the file is being referenced.
- The loader accesses all object libraries through the segment level access facility of the file and memory management systems. It also uses segment management to create the transient segments that are used for the data areas of the executing program. The loader is responsible for creating these segments with the correct protection attributes to assure proper execution and protection of the program.

An example illustrates how these mechanisms interact to effect access control in NOS/VE. Consider two users, TOM and BILL. TOM has been validated by the installation to execute in ring nine. BILL has been validated to run in ring eleven. TOM develops an application and stores it in the permanent file catalog. Since TOM was executing in ring nine when he cataloged his application, it has an execute bracket of (nine,nine) by default. In order to allow BILL to use his application, TOM must set the call bracket of his application permanent file to eleven and give BILL execute permission in the access control list of the file. Since TOM is the owner of his application, he is the only user permitted to set its call bracket and place entries in its access control list.

In order to use TOM´s application, BILL must first ATTACH the file for execute access. This will succeed because TOM has placed BILL in the access control list of the file and specified execute access. BILL then executes a program which uses TOM´s application. BILL´s program is loaded in ring eleven and TOM´s program is loaded in ring nine. Because TOM´s program has a call bracket that extends to ring eleven, BILL´s program can call TOM´s and use the service it provides.

HARDWARE FACILITIES

One of the primary design goals of CYBER 180 systems was to improve the overall system reliability. In the past a limiting factor has been the operating system. This software is large - of the order of one million lines of code - and is error prone. An error in today´s systems often causes those systems to crash, interrupting normal service. Since it is unlikely that such a vast quantity of code can be generated error free, other solutions must be sought. In the CYBER 180 the solution chosen is to give each user his or her own copy of the operating system; then if a particular copy of the system fails it will cause nothing more serious than a single job to abort. With this approach, the operating system and the user´s code become an entity, and facilities must be provided to separate and protect operating system modules from user modules and from each other. This is the primary reason for the CYBER 180 security system. A second major objective is to provide controlled access to all code and data. To this end users are protected from each other, and can be protected from the system.

The key to this protection mechanism is the CYBER 180 virtual memory mechanism. In particular, the virtual memory segment is the basic element which is protected. However, before the attributes of this segment are discussed, it is necessary to understand how segments are arranged in virtual memory for utilization by a user.

VIRTUAL MEMORY USER ADDRESS SPACE

The concept of an Address Space is vital to CYBER 180. It is simply the set of addresses known to an executing process. On CYBER 170 this would be the set of real memory addresses embraced by RA and FL. On CYBER 180 it is the set of virtual memory addresses specified by the entries in the SDT. Each process executing in a CYBER 180 system has a unique SDT. The address and length of this table are specified by process state registers held in the exchange package used to define the environment of the exchange interval for each task. Each entry in the SDT consists of a full word and describes all of the attributes of one segment. Just as CYBER 170 users are constrained to an address space by the RA/FL mechanism, so are CYBER 180 users constrained to an address space by the SDT. This, then, is the basic element of protection. The discussion of the protection mechanisms which follow describes the protection offered within a virtual memory address space.

It is useful at this stage to see what happens when a user attempts to access code or data segments not in his address space. The only addresses known to the user are process virtual addresses (PVAs). Refer to figure 3-2. Each PVA has three components: a ring number, a segment number, and a byte number.

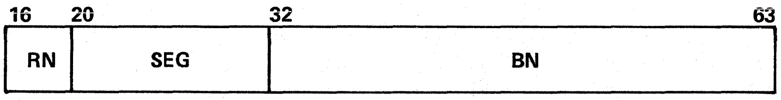


Figure 3-2. Process Virtual Address

The ring number is discussed later in this section. The segment number is assigned by the operating system (segment manager) in ascending sequential order starting with zero. These are the names, and are the only names by which the user knows his segments. They act as an index into the process segment table which contains entries only for those segments to which the user has access rights. The byte number field simply denotes a byte offset within a segment. The only way a user can attempt to access a segment which is not contained within his address space is by specifying a segment number greater than any assigned by the operating system for that process. However, when an attempt is made to reference that segment an exchange interrupt results since the segment number is greater than the Segment Table Length (STL). The STL is set by the operating system and held in a process state register which can be read but not written - it is set by an exchange jump. The hardware performs this basic test for every reference which is made to memory.

Whenever an exchange jump occurs, a switch of address spaces occurs. The operating system monitor runs in its own, unique address space. This is not true of the bulk of the operating system. Services such as those offered by Record Manager reside within the user's address space. Further protection mechanisms, which are described below, come into play to protect these parts of the system from the user and vice versa. It is important to conceptualize and remember that all this happens in virtual memory. Conceptually, operating system segments which reside in the user address space exist as multiple copies in virtual memory. To optimize the use of real memory the operating system will typically keep a single copy of the code in real memory which will be shared by several users. The individual users will be unaware of this since they are only aware of what happens in virtual memory. There is no possible breach of security here since each user is totally unaware of the existence of other users.

SEGMENT ATTRIBUTES

Once a user has been confined to an address space, the segment becomes the basis of the security mechanism. Each segment has a set of attributes associated with it which are recorded in the Segment Descriptor Entry (SDE) in the SDT. These attributes are its global system name, its access attributes, its rings and its global and local locks. A segment also has a length associated with it, which is not kept in the SDE. Since these are recorded in the SDE, and the SDE is unique to a given process, it is possible for a segment to be shared by more than one process, yet have different attributes for each process. The format of an SDE is shown in figure 3-3:

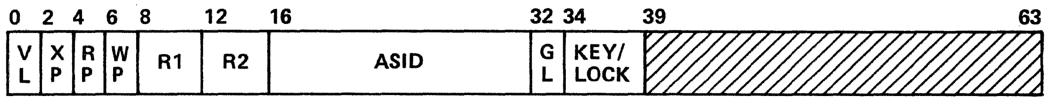


Figure 3-3. Segment Descriptor Entry (SDE)

The first four fields determine the access privileges for the segment. Values for these fields are as follows:

VL : 00 - Invalid entry. A segment which is no longer being used by a process does not have to have its SDE removed from the SDT but simply invalidated. If segments are viewed as files, their entries would be invalidated when the files (segments) are closed and purged.

01 - Reserved.

10 - Regular segment. This denotes an active segment for the executing process.

11 - Cache by-pass segment. It is important to keep certain tables and interlock words in a cache by-pass segment. An example is the exchange package. The exchange jump mechanism works from a real memory address, hence data in cache memory, which memories are addressed via a System Virtual Address (SVA), does not get updated.

XP : 00 - Nonexecutable segment. This is a data segment which would normally have either read or write access.

01 - Nonprivileged executable segment. Some instructions can only be executed if they reside in a segment having the attributes of local or global privilege. In this way certain operations are restricted for use by the Operating System and cannot be invoked accidentally by a user executing garbage - for example, literals.

10 - Local-privileged executable segment. Code contained in segments with that attribute may execute all unprivileged instructions and all instructions restricted to local privilege. In particular, trap handlers will have at least local privilege since the trap enable flip-flop and the trap enable delay flip-flop can only be set by a system copy instruction, and these flags can only be written in local privileged mode.

11 - Global-privileged executable segment. Code contained in segments having global privilege may execute all instructions except those restricted to monitor mode. This includes those instructions restricted to local privileged mode, which is a subset of global privilege.

RP : 00 - Nonreadable segment. Such a segment will either have write privilege or execute privilege. An execute privilege segment would normally have only that attribute. However, literals may be stored in the segment and read from the segment with load instructions provided for that purpose. The load instructions always load from an address relative to P - the program address counter. In this case the read access is implicitly equated to the execute access of the segment.

01 - Read under control of the Key/Lock mechanism. For a segment controlled by a lock, this control may selectively apply to either the read or write privilege of the segment. This code indicates that reads are under key/lock control.

10 - Read not under the control of Key/Lock. This is a normal read privilege assigned to the segment.

11 - Binding Section - Read not under the control of Key/Locks. Binding Sections, which contain pointers to external procedures and data, always have read privilege, and are never subject to Key/Lock control.

WP :00 - Nonwritable segment. Typically, all executable segments are nonwritable. This is because CYBER 180 code is usually organized into pure procedures. A user could generate a code segment from assembly language which modified code. However, this code segment would not be sharable with other users.

01 - Write controlled by the Key/Lock mechanism. This is the write counterpart of RP=01 and indicates that the segment is writable but only if the Key/Lock access is correct.

10 - Write not under control of Key/Lock. This is a normal, writable data segment.

11 - Reserved.

The XP, RP and WP are the first level of protection offered within a user address space. Figure 3-4 illustrates segment protection within an address space.

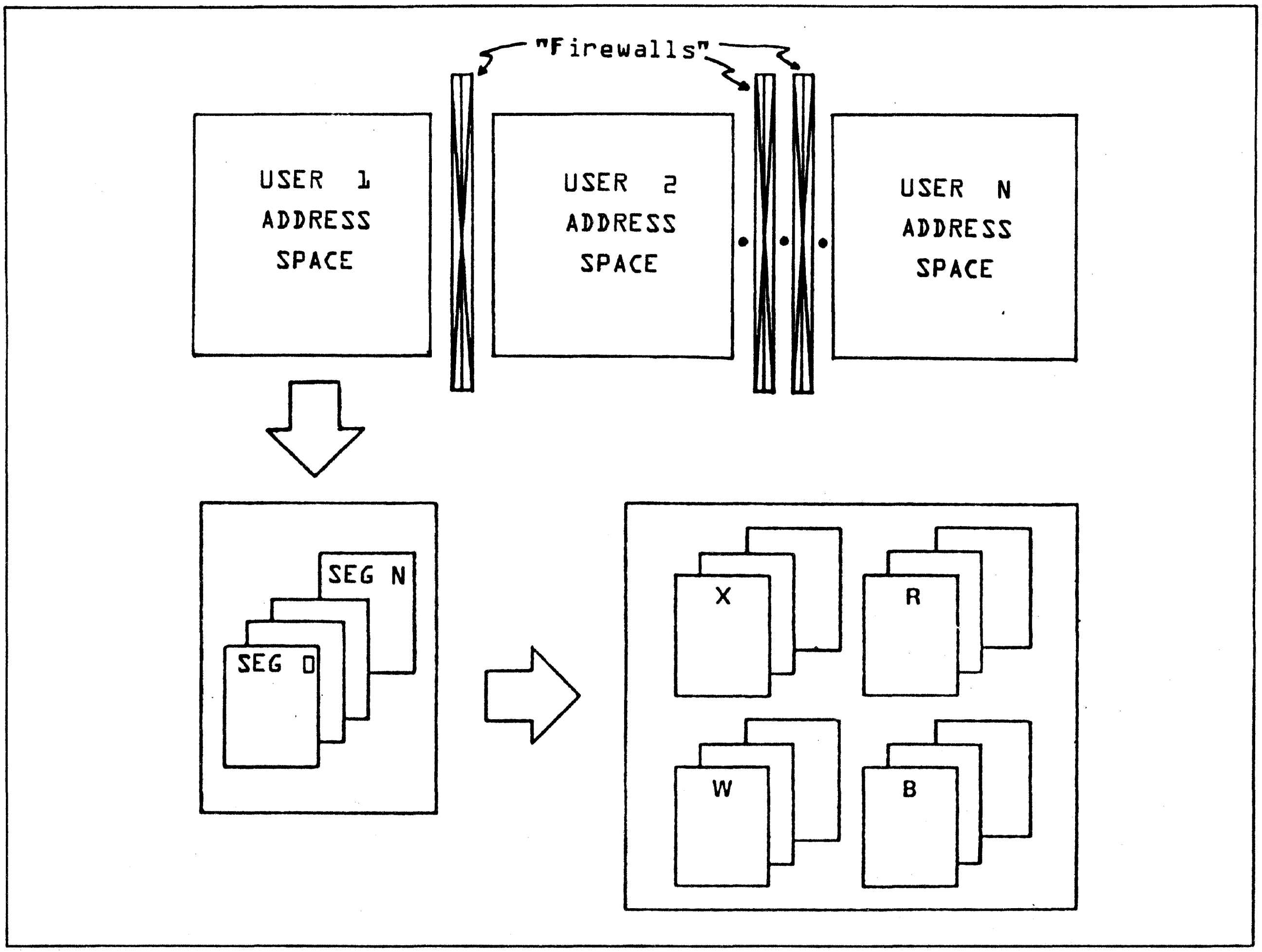


Figure 3-4. Segment Protection Within An Address Space

Hence, users are first absolutely constrained to their address space. Within this space code and data are organized into segments, then the segments are assigned various privileges. Most of the operating system and entire subsystems are expected to exist in the user address space. These mechanisms, therefore, are necessary to ensure the appropriate security is maintained at all times. Notice that even privileged portions of the operating system, such as trap handlers with local privilege, can reside in the same address space as a user who may be executing completely unchecked code - and therefore very unreliable code.

This level of protection will guarantee that code segments are not arbitrarily over-written by users, leading to unpredictable results, and it will guarantee that read-only data segments are not destroyed either willfully or accidentally. However, this level of protection is insufficient by itself. For example, users could read and write segments of the operating system, or users with inadequate security clearances could gain access to private data segments. To accommodate these aspects of security three further mechanisms are provided.

RINGS OF PROTECTION

To provide a separation between code and data segments, and prevent unauthorized access, each address space is organized into a series of Rings of Protection. A maximum of fifteen such rings are permitted in an address space. They may be regarded as separate machine states having differing privileges. The rings are organized hierarchically such that the lower the ring number the higher the privilege, Ring 1 having the highest privilege. In general, code residing in Ring n can read and write segments in Ring n and higher numbered rings. In addition, code in Ring n can call procedures in Ring n and lower numbered rings, although such calling is carefully controlled. The ring protection mechanism is controlled by the R1 and R2 fields in the SDE, the R3 field in the Code Base Pointer (CBP) and the ring number carried in all PVA´s - particularly those held in A Registers and P Registers. Code and Data Segments need not reside in a single ring but may exist in several rings. When this occurs the segment is said to reside in a Ring Bracket. The extent of this ring bracket is defined by the R1 and R2 fields of the SDE.

There are four Ring Brackets which are associated with each and every segment. These rings brackets are for read, write, execute and call. The first three of these are truly segment attributes and are described by the SDE, the fourth - the call bracket - is associated with the segment by the operating system with no loss in generality. In practice, the operating system does not only associate these ring brackets with segments but associates them with every file in the system, either local or permanent, regardless whether or not a given file is a segment. The checking which is performed by the hardware is described below for each type of access.

Execute Access

A segment which resides in several rings has its execute bracket described by the R1 and R2 fields of the SDE. Thus if:

$$\text{SDE.R1} \leq \text{P.RN} \leq \text{SDE.R2}$$

then the segment is a member of the execute bracket. If control is transferred to the segment from within the execute bracket, then the ring number in the P Register (P.RN) is unchanged. If control is transferred from outside the ring bracket (via an inter-ring call), then P.RN is always set to SDE.R2. Calls, if permitted, can only be made inward. The hardware validates execute access once and only once when a segment is entered. The fact that calls can only be made inward (or to the same ring) often appears confusing, but the reason is quite straightforward. Care must always be taken when crossing domains of protection to ensure that no security violation occurs. This is particularly true when traversing from one domain to another with higher privilege. If an outward call were permitted then its counterpart, an inward return, would also have to be permitted. However, a return is an unsolicited GO TO, which implies a traversal of domains of protection without the necessary control. So it is really an inward return which must be prevented.

Call Access

Two main checks are exercised by the hardware when a call is made. The first ensures that the call is an inward call:

$$\text{PVA.RN} \geq \text{SDE.R1}$$

where PVA.RN is the ring number of the PVA containing the entry point of the called procedure, and SDE.R1 is the lower range of the ring bracket of the called procedure.

The second check ensures that the caller has adequate privilege to call the callee:

$PVA.RN \leq CBP.R3$

where CBP.R3 is the CBP gate ring number.

Hence, callee can restrict entry to the procedure such that he may only be called from certain rings. Now one more case is of interest. That is, where a routine is called on behalf of another caller. This can happen when a caller calls on a more privileged procedure legitimately, but then requests that the callee in turn call a third procedure to which callee has access but caller does not. Via a high level language, this is constructed very simply by a pointer to procedure. To prevent this form of unauthorized call, the hardware performs an additional check:

$Aj.RN \leq CBP.R3$

where Aj.Rn is the ring number of the pointer used to access the binding section containing the relevant CBP. This A register will not have more privilege than the code requesting the call. It is that code which must reside within callee´s call ring bracket. In practice, since P.RN will always be less than or equal to Aj.RN the hardware only has to perform the latter test.

Read Access

An executing procedure may read a segment providing the following is true:

$PVA.RN \leq SDE.R2$

where PVA.RN is the ring number of the pointer (held in an A Register) used to access virtual memory, and SDE.R2 is the outermost ring number for the segment being accessed.

Thus, a procedure may read a segment from a ring of equal or lower privilege than its own. For the read access to be successful, of course, the segment must have the read attribute associated with it.

Write Access

An executing procedure may write a segment providing the following is true:

$PVA.RN \leq SDE.R1$

where PVA.RN is the ring number of the pointer (held in the A Register) used to access virtual memory, and SDE.R1 is the innermost ring number of the ring bracket for the segment being accessed.

These four major ring brackets are shown in figure 3-5:

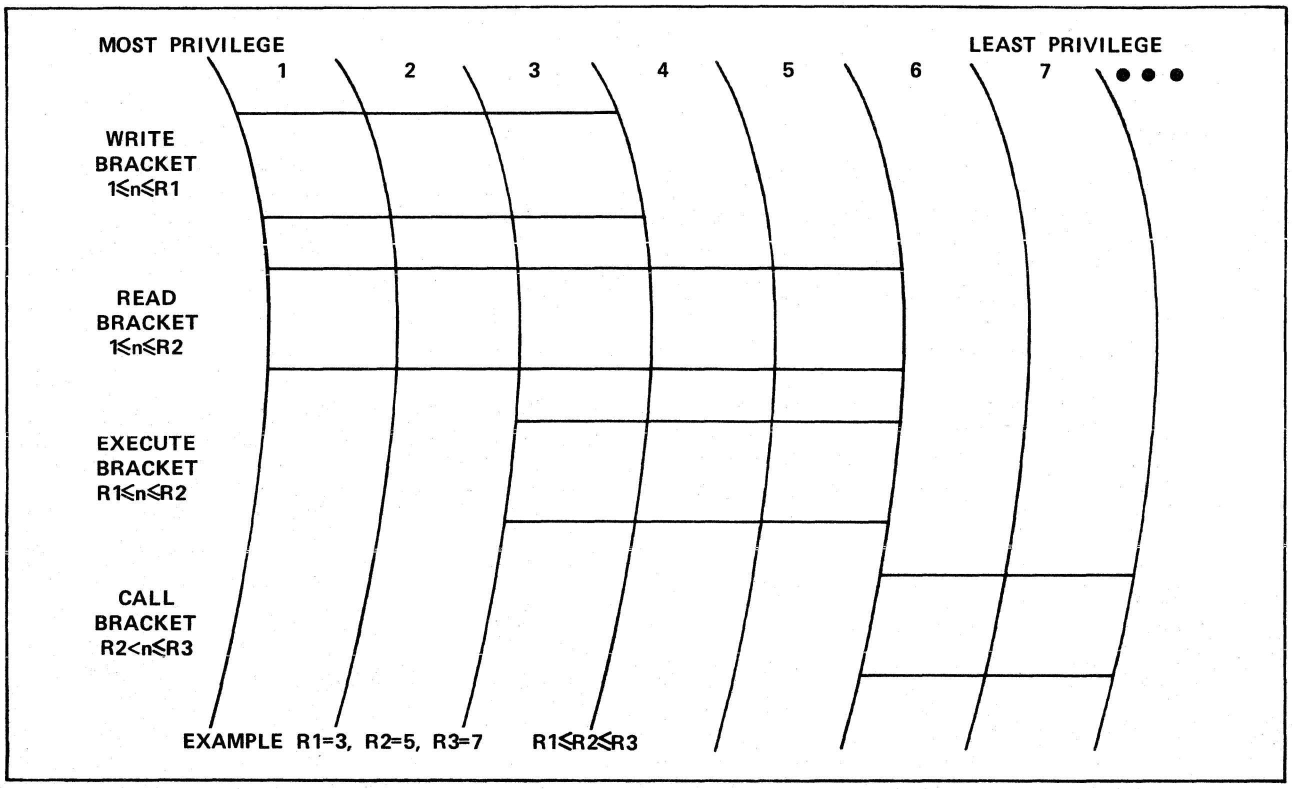


Figure 3-5. Ring Brackets

In this example the R1, R2 and R3 parameters, which define the ring brackets for a particular segment, have been set as follows:

R1 = 3
R2 = 5
R3 = 7

The segment may be written from another segment if that other segment resides in ring 3 or in a lower numbered ring (ring 1 or 2).

The segment may be read from another segment, providing the other segment resides in ring 5 or in a lower numbered ring (rings 1 to 4).

The segment may execute in rings 3, 4 or 5. If the segment is called from ring 3 it will execute in ring 3. If it is called from ring 5 it will execute in ring 5 and so on. If it is called from ring 6 (or a ring numbered greater than 6), then it will execute in ring 5 - the least privileged of rings 3, 4, and 5.

The segment may be called from either rings 6 or 7. Segments may always be called from other segments in the same ring that they are in. Consequently, the segment may be called from rings 3 through 7. It may not be called from any rings greater than 7, which are outside the call bracket. Neither may it be called from a ring number less than three since this would constitute an outward call. This case is covered in section 7, where Call/Return is discussed. Call/Return is the primary mechanism for crossing protection boundaries within an address space.

Figure 3-6 shows how ring brackets are used:

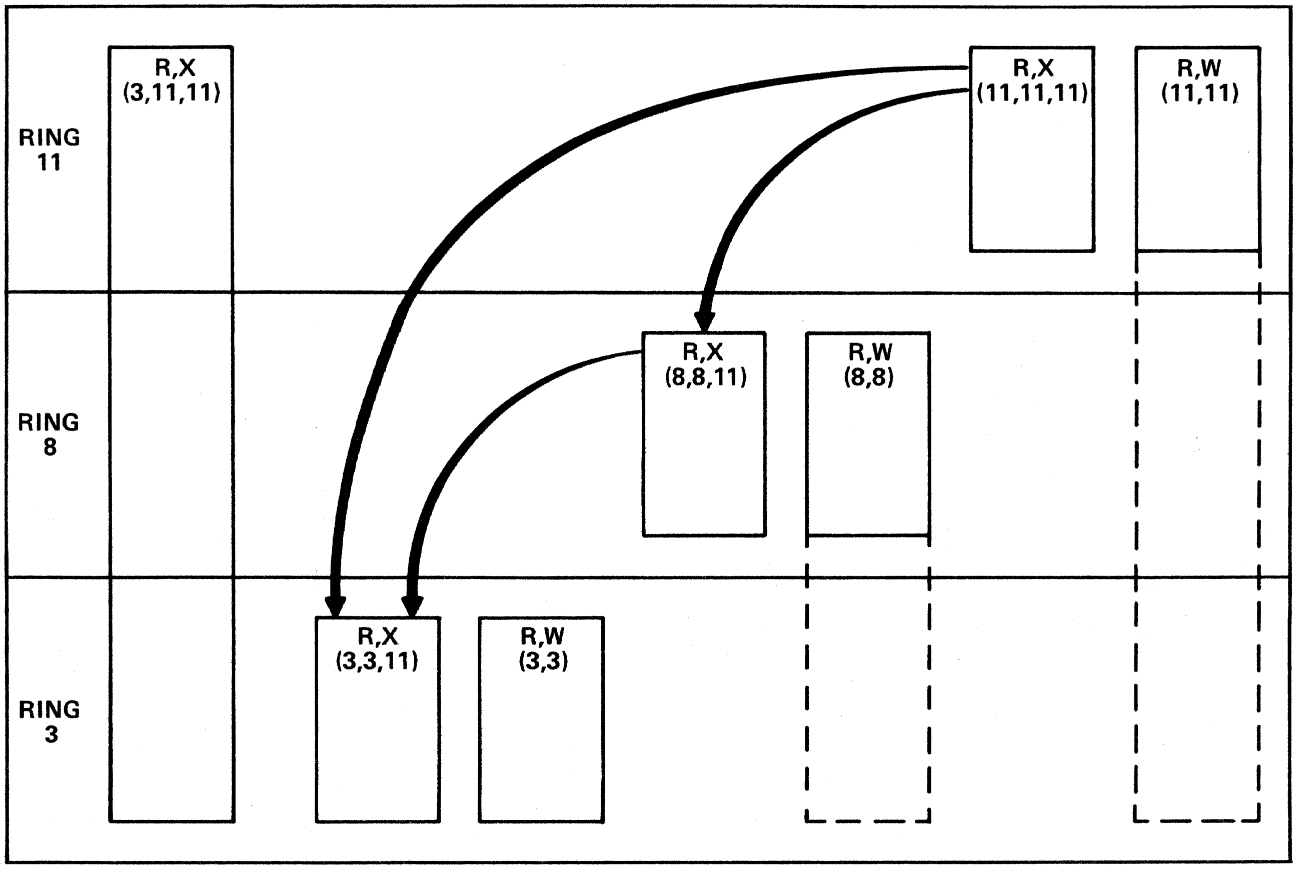


Figure 3-6. Example of Ring Brackets

In this example the extremities of the ring brackets for each segment are denoted by the numbers in parentheses as: (R1,R2,R3). The procedure in ring 11 may call on the procedure in ring 8 since the call bracket for this latter procedure has been set to 11. The procedure in ring 8 may read and write into the data segment belonging to the procedure in ring 11, since the segment has Read/Write access and its R1 and R2 fields have both been set to 11. In this way the procedure in ring 11 may pass parameters to and receive results from the procedure in ring 8. Likewise, the procedure in ring 3 may be called from either the procedure in ring 8 or the procedure in ring 11, because the ring 3 procedure has its call bracket equal to ring 11. The logical extensions of the data segments in each ring are indicated by dotted lines in the diagram. Notice that there are no extensions to the execute segments since the R1 and R2 fields restrict execution of the segment to a particular ring. Thus, the procedure in ring 11 may only be executed in ring 11, the procedure in ring 8 may only be executed in ring 8, and the procedure in ring 3 may only be executed in ring 3.

On the left side of the figure there is an execute segment having R1 and R2 fields equal to 3 and 11 respectively. Consequently, this procedure may be executed in any ring between 3 and 11 inclusive. Such a procedure might be a trap handler or the FORTRAN math library. A user executing in ring 11 may call on square root, for example, which would then execute in ring 11. Similarily, a procedure in ring 8 utilizing square root would have it execute in ring 8. In other words, the square root procedure always executes with the privilege of the caller. This is required, since it is acting on behalf of the caller.

Now the segment may only be read, written or executed if it has the appropriate access permission associated with the segment. In other words for a segment to be written from another segment it must contain write permission and the other segment must reside in a ring from which the first segment may be written.

Within a single address space, therefore, rings of protection provide a mechanism for protecting sensitive code and data. Two cases are of particular interest.

The first of these deals with the need to know. A procedure should have access only to those procedures and data segments necessary to do its task. Now remember that the ring mechanism is hierarchical. That is, the lower the ring number, the higher the privilege. Consequently, the ring mechanism is very attractive in a military environment where security clearances are also hierarchical. A higher clearance (lower ring number) allows access to more documents, but a smaller number of individuals (segments) are granted such a clearance.

The second case is concerned with degrees of potential damage. The segments of a system may be effectively segregated into two or more rings according to the damage that may be wrought when these segments are misused. The segments whose misuse is likely to cause the greatest damage are given lower ring numbers. By means of this segregation the bulk of the operating system may reside within the user´s address space and yet be protected from the vagaries of undebugged user code. If part of the operating system does fail, then the damage may be contained and cause nothing worse than the user job to abort. Hence, rings will be used extensively by the operating system for damage control, and also made available for the user to create an hierarchical security structure. Figure 3-7 illustrates the resulting user address space.

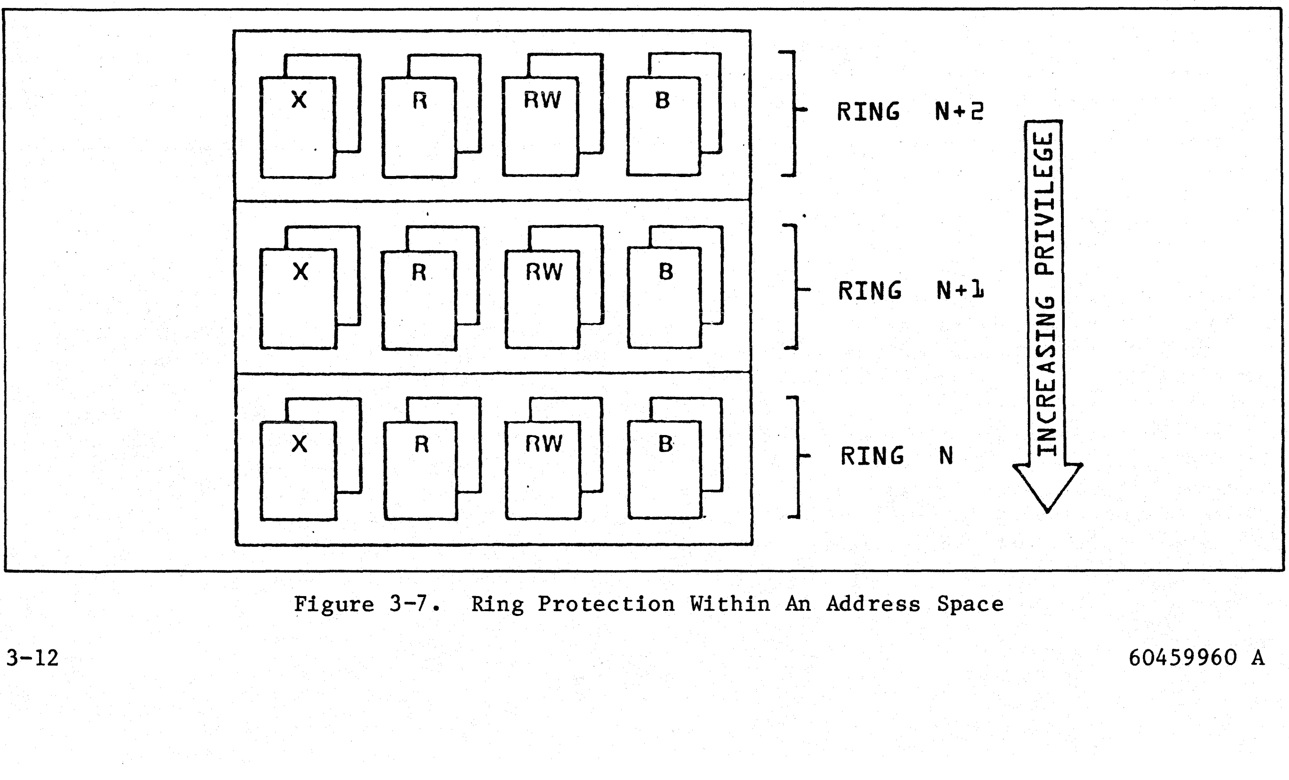


Figure 3-7. Ring Protection Within An Address Space

The address space, which is the basic unit of protection, now has two further protective mechanisms. The first restricts the type of access to a segment, and the second limits the region from which a segment may be accessed.

RING NUMBERS IN POINTERS

The only addresses with which programmers deal are Process Virtual Addresses (PVAs). Refer to figure 3-8. A set of 16 address registers (A Registers) exists in the hardware to hold these addresses during instruction execution.

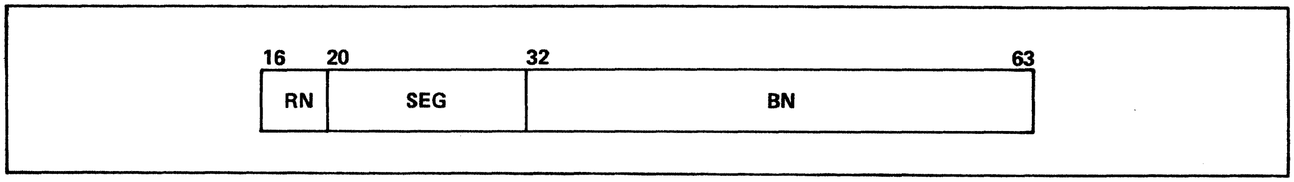


Figure 3-8. Process Virtual Address

The SEG and BN fields designate the segment being addressed and the byte offset within that segment respectively. The RN field is the ring number associated with the access being made. This ring number is most important to the ring security in the system since it is common for a procedure to perform work on behalf of another, less privileged procedure. When this happens it is important that the more privileged procedure does not act with greater authority than has been assigned to the caller. To this end, whenever an A Register is loaded, either explicitly (via a Load or Copy instruction), or implicitly (via a Return or Pop instruction) the hardware places the ring number with least privilege into the register. A comparison is made between the ring number of an A Register being used for the load, the ring number of the pointer being loaded and the ring number of the R1 field of the SDE associated with the A Register used to load the pointer (refer to figure 3-9). The largest of these three ring numbers is entered into the destination A Register.

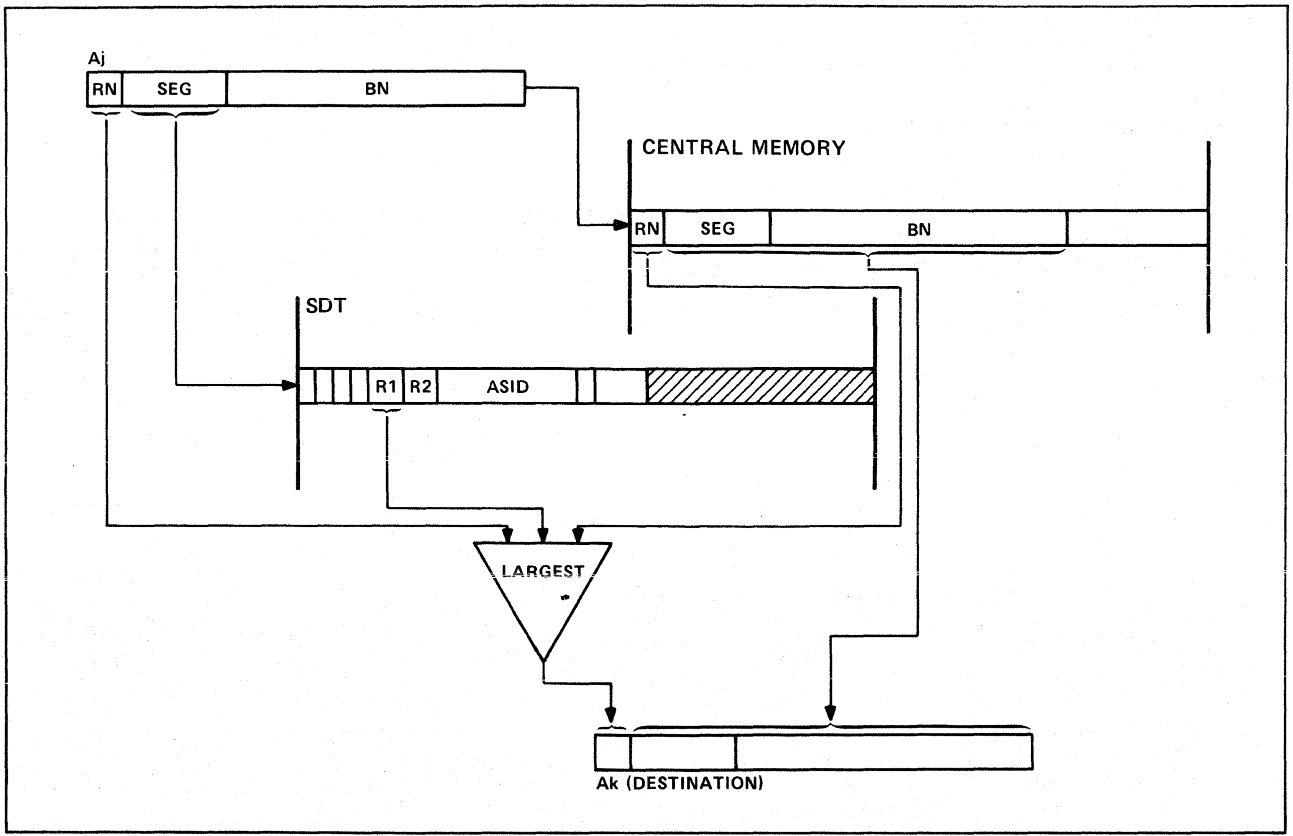


Figure 3-9. A Register Ring Voting

When an A Register is loaded via a Copy from an X Register, a comparison is made between the Ring Number of the pointer held in the X Register and the Ring Number held in the P Register, the larger of the two values being used. Since this cannot be as rigorous a test as that used for loading A Registers, care must be exercised in its use. For example, if a procedure calls on a second procedure in a more privileged ring, and a pointer or pointers are passed via loading an X Register and copying the X Register to an A Register, then the callee may end up acting on behalf of caller, with more privilege than caller is allowed. However, when this happens callee - the more privileged procedure - is at fault. It is incumbent upon the more privileged - and therefore more trustworthy - procedure to maintain the security of the system down through his level. If this fundamental software convention is not followed, there is nothing the hardware can do to maintain system integrity.

GLOBAL AND LOCAL KEY/LOCKS

Keys and locks provide two other protection mechanisms in the hardware. These deal with mutually suspicious code segments which reside within the same ring of protection. Here again, two cases are of interest: the protection of local data (where Local Key/Locks are used), and the isolation of competing applications. Some examples will help to clarify the problems being addressed by these mechanisms. Following these examples there is a description of how the hardware functions, and then a final example to tie the entire concept together.

Local Key/Lock Usage

An example of Local Key/Lock usage may be taken from a math function which has been developed and is being marketed by some organization. Since the function is general purpose it will typically reside in the same ring of execution as the user who is calling it. However, the developer may wish to restrict access to coefficients he has derived and which exist in a separate (read-only) segment. Hence, the scenario is one in which a read-only data segment can only be accessed from a given code segment or segments in a given ring. This type of access protection is accomplished by means of Local Keys and Locks being applied to all segments in that ring.

Global Key/Lock Usage

Notice that the purpose of Local Key/Locks is to protect local data (data which is used by a particular procedure or procedures). Global Key/Locks, in turn, are used for isolation rather than for protection. For example, two proprietary applications for competing organizations may coexist within a user address space. The applications are regarded as subsystems and, as such, have been placed in a ring of execution of more privilege than that of the end user. Within this ring it is necessary to isolate the applications so that they may neither call each other, nor read nor write data from one to the other. Global Key/Locks are used to achieve this isolation.

Key/Lock Hardware Mechanism

There is a lock associated with every segment. It is described by a six-bit field in the SDE, hence up to 64 different locks may coexist. Whenever a segment is executed, the lock associated with that segment becomes the current key. The various values which locks may have assume no hierarchical significance, as with rings. Of importance only is whether the key/locks are the same or different. Two bits in the SDE describe whether the lock associated with the segment is a Local Lock or a Global Lock, or whether the single six-bit lock value acts as both a Local and Global Lock. The format of SDE bits 32-39 is shown in figure 3-10:

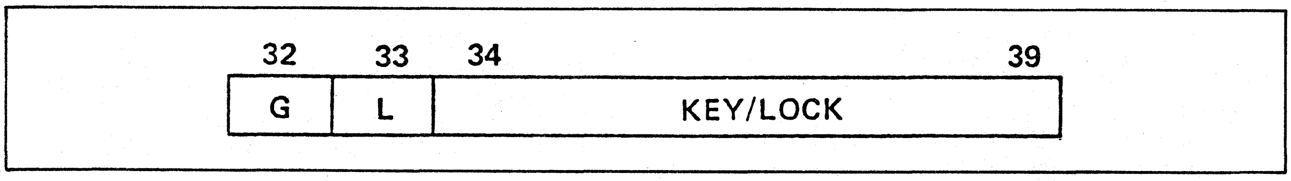


Figure 3-10. Format of SDE Bits 32-39

The G and L fields have the following meaning:

Procedure		Global	Local
G	L		
0	0	Master Key	Master Key
0	1	Master Key	6-bit Key
1	0	6-bit Key	Master Key
1	1	6-bit Key	6-bit Key

Data			
G	L		
0	0	No Lock	No Lock
0	1	No Lock	6-bit Lock
1	0	6-bit Lock	No Lock
1	1	6-bit Lock	6-bit Lock

A Master Key will fit any Lock, and any Key will fit a No Lock. In general, access to one segment from another segment will only be granted if the first segment has no lock (Local and Global), or if the second segment has a Master Key (Local and Global), or if the Global and Local Keys exactly match the Global and Local Locks. These tests, which are executed by the hardware, are in addition to these already described for rings and type of access. However, the Key/Lock tests are performed selectively as controlled by the RP and WP fields in the SDE. Hence, even though a Global and/or Local Lock may have been specified for a segment, the test for read access will only apply when the Lock applies to read access as indicated by an RP value of 01. Write accesses are similarly controlled by WP.

In addition to the tests performed for read and write access, Global Key/Lock tests are performed on Calls and Returns. A call is permitted when callee has a Master Key, when caller has No Lock, or when callee's Key exactly equals caller's Lock. The Key transformations which take place on a call are summarized as follows:

Caller's Global Key	Callee's Global Lock	New Global Key
0	0	0
0	K2	K2
K1	0	K1
K1	K2	K1 if K1 = K2 else Access Violation

On a Return, the hardware checks (against caller's SDE) to ensure that caller's Global Key, obtained from the Stack Frame Save Area, is equal to Caller's Global Lock. The following combinations are permitted:

New Global Key (from SFSA)	Global Lock (from SDE)
0	0
K1	K1
K1	0

All other Key/Lock transformations result in an Access Violation.

The new Local Key on a call is always taken unconditionally from Callee's Local Key value in the SDE. On Return the hardware verifies that the Local Key obtained from the SFSA exactly matches the Local Lock taken from Caller's SDE.

Key/Lock Example

Figure 3-11 illustrates the usage of Key/Lock values and should help to clarify the mechanism.

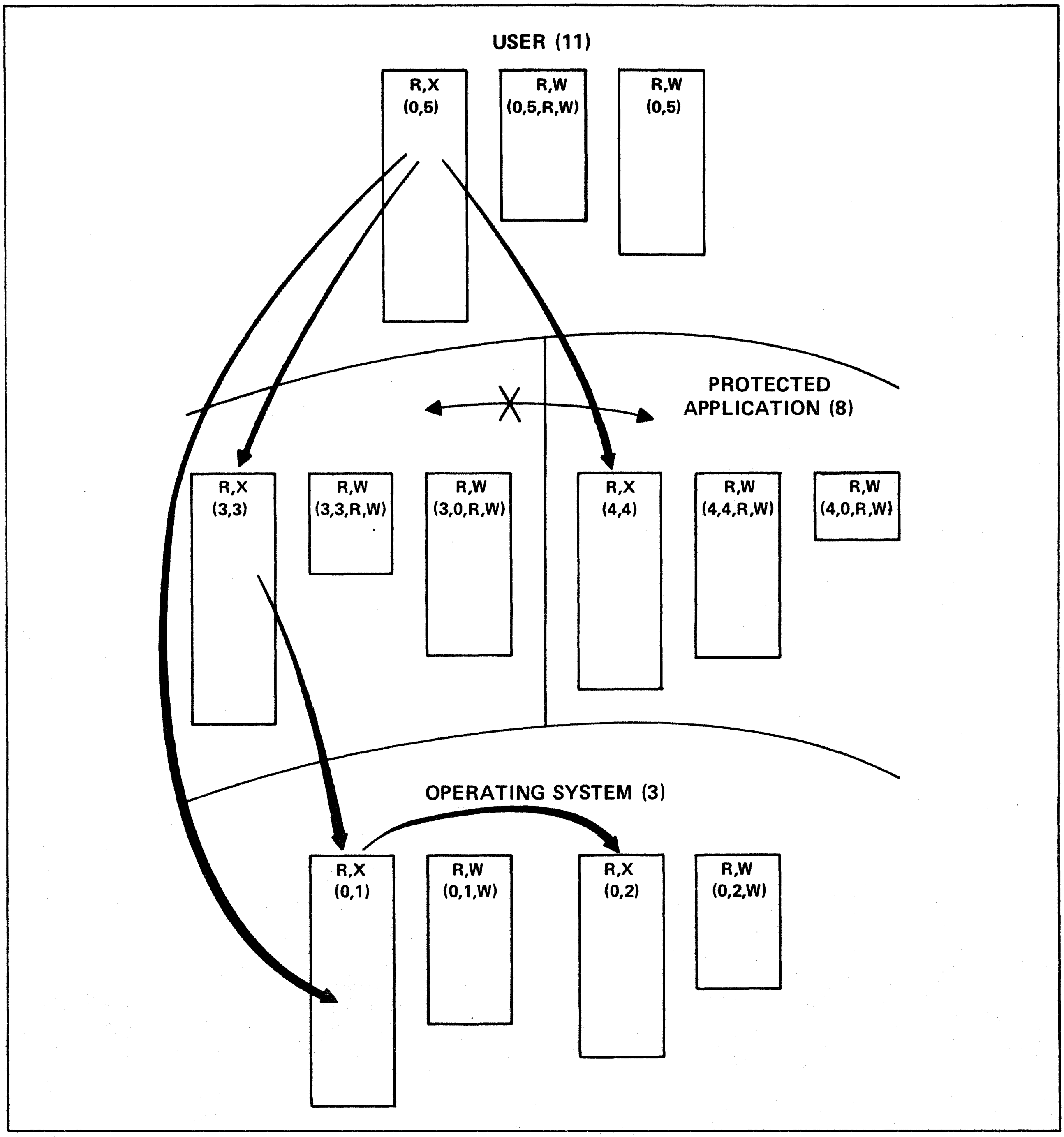


Figure 3-11. Key/Lock Example

In this example, three rings of protection are utilized. Ring 3 is the most privileged ring, where parts of the operating system reside. Ring 8 contains two applications which must be isolated from each other, but callable from a user in Ring 11. The information in parentheses defines the Global Lock, Local Lock and whether these Locks apply to read accesses (R), write accesses (W) or both.

In order to isolate the applications from each other they are assigned different Global Key/Locks. Likewise, for them to be called by a user in Ring 11, and to be able to call the operating system in Ring 3, the user and the operating system are assigned Global Key/Lock values equal to zero. In other words the user and the operating system have Master Global Keys - No Global Locks. This is consistent with the general software convention for Global Locks: Subsystems are confined to rings greater than those for the operating system but less than those for the user. Each Subsystem in these rings is assigned a unique Global Key/Lock value. This limits the total number of Subsystems within a single user address space to 63. In the example, Application A cannot read or write Application B´s data segments since the Global Key/Lock values are different. Likewise the applications cannot call each other. Consequently, they have been totally isolated from each other, even though they reside in the same ring of protection. However, the user, in Ring 11 may call on either application and on the operating system, since the user has a Global Key/Lock of zero - a Master Global Key. When a call is made on an application it runs with its own Global Key, thus ensuring continued isolation. On return to the user, the user´s Global key is restored to him. The user cannot read or write the applications´ ring. When the operating system is called from the user it runs with a Master Global Key (user has a Master Global Key, operating system has No Global Lock). Hence, nominally, the operating system can read and write the applications data segments. However, when the operating system is called from an application, then it executes with the Global Key of that application and cannot read or write the other application´s data segments. Hence when the operating system executes on behalf of a protected or isolated application, it executes with the privilege of that application, and consequently cannot be tricked by that application into giving it access to data from which it is otherwise isolated.

Local Key/Locks work rather differently. Whenever a call is made to a procedure in another segment, callee executes with his own Local Key. This value is associated with the Local Lock of the data segments accessed by that procedure. In the example, the user has a read/write data segment to which Key/Lock verification applies. It has a unique Local Key/Lock value of five. Consequently, the applications in Ring 8, which have Local Keys of three and four respectively, and the operating system which executes in Ring 3 with a Local Key value of either one or two cannot access this data segment. This is true even though the data segment is available for reading and writing, and resides in Ring 11 - a ring with very little privilege. Similarly, the operating system cannot read or write either of the applications´ data segments since they have different Local Key/Lock values from the operating system, and each other. Hence, Local Key/Locks are used to protect local data regardless of the ring structure in use.

By software convention the operating system segments (both code and data) are assigned nonzero Local Key/Locks. This has the added advantage that various modules of the operating system can be protected from each other. In the example, there are two modules, both in Ring 3, which can call each other and can read each others data segments. However, the data segments can only be written from the module to which they belong. This is a very powerful debug aid for the operating system. In today´s systems it is not uncommon for one module of the operating system to accidentally destroy data belonging to another module. The damage is not discovered until the second module is called, by which time the culprit is unidentifiable. Through the use of Local Key/Locks the culprit can be identified at the time the data was over-written, or at the time when this was attempted.

Since the SDE contains a single Key/Lock value for Global and Local Key/Locks, and since the algorithms for transforming Global Key values on a call are different for the two types of Key/Locks, it is necessary to maintain two separate values for the current key. The current key is maintained in the P Register and figure 3-12 indicates that two separate six-bit fields exist in the P Register for this purpose.

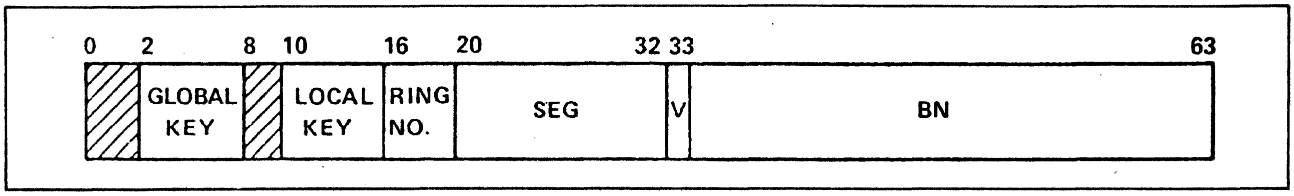


Figure 3-12. Program Address Register

In summary, there are three basic forms of protection from within a user space: the type of access to a segment, the ring protection mechanism, and Global and Local Key/Lock values. For every access attempted, all three of these tests must be successful. If any one of them fails, an Access Violation interrupt results, and the user is exchanged out of his address space into the Operating System monitor address space where appropriate action results. The complete, protected user address space is illustrated in figure 3-13:

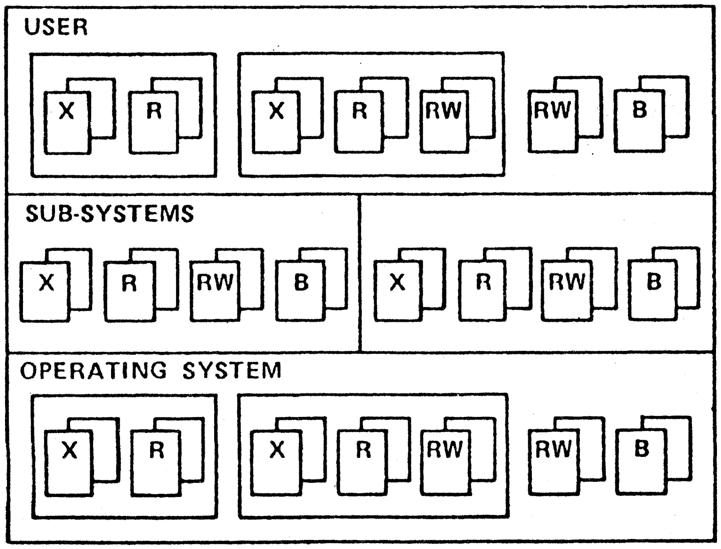


Figure 3-13. Conceptualization of a User Address Space

The following flowcharts (figures 3-14 and 3-15) describe the complete virtual memory address translation and access control.

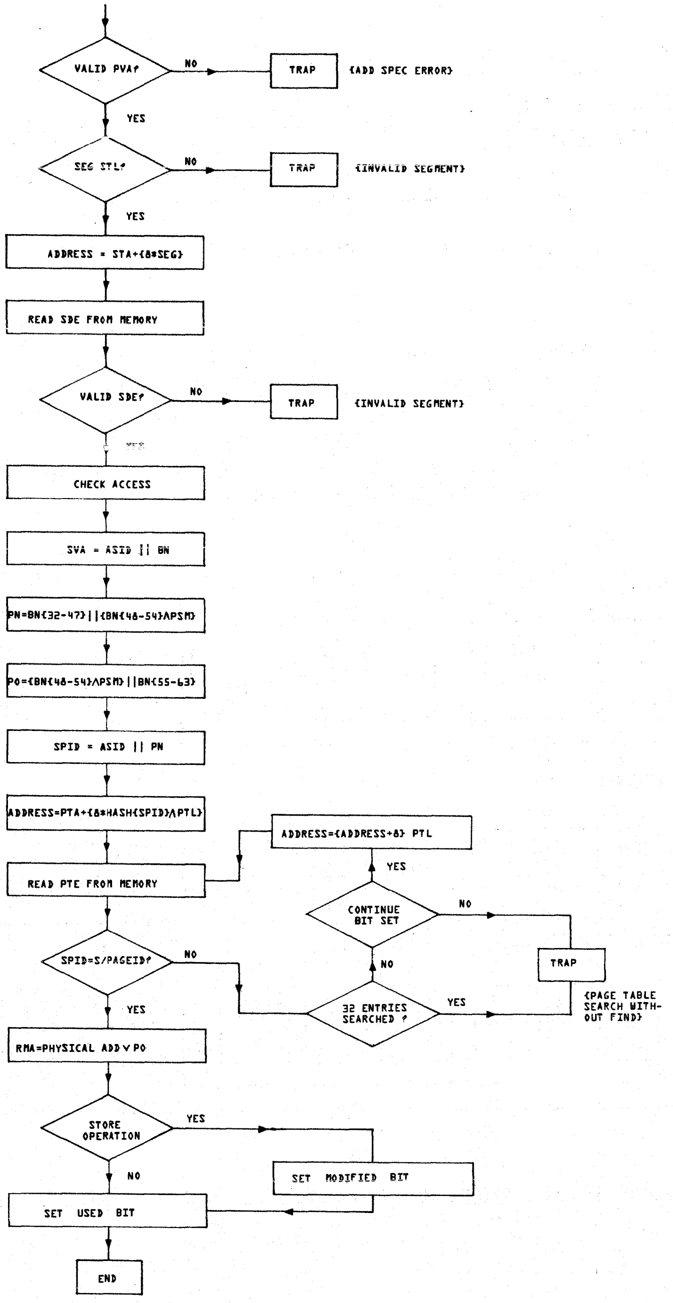


Figure 3-14. Virtual Memory Address Translation Flowchart

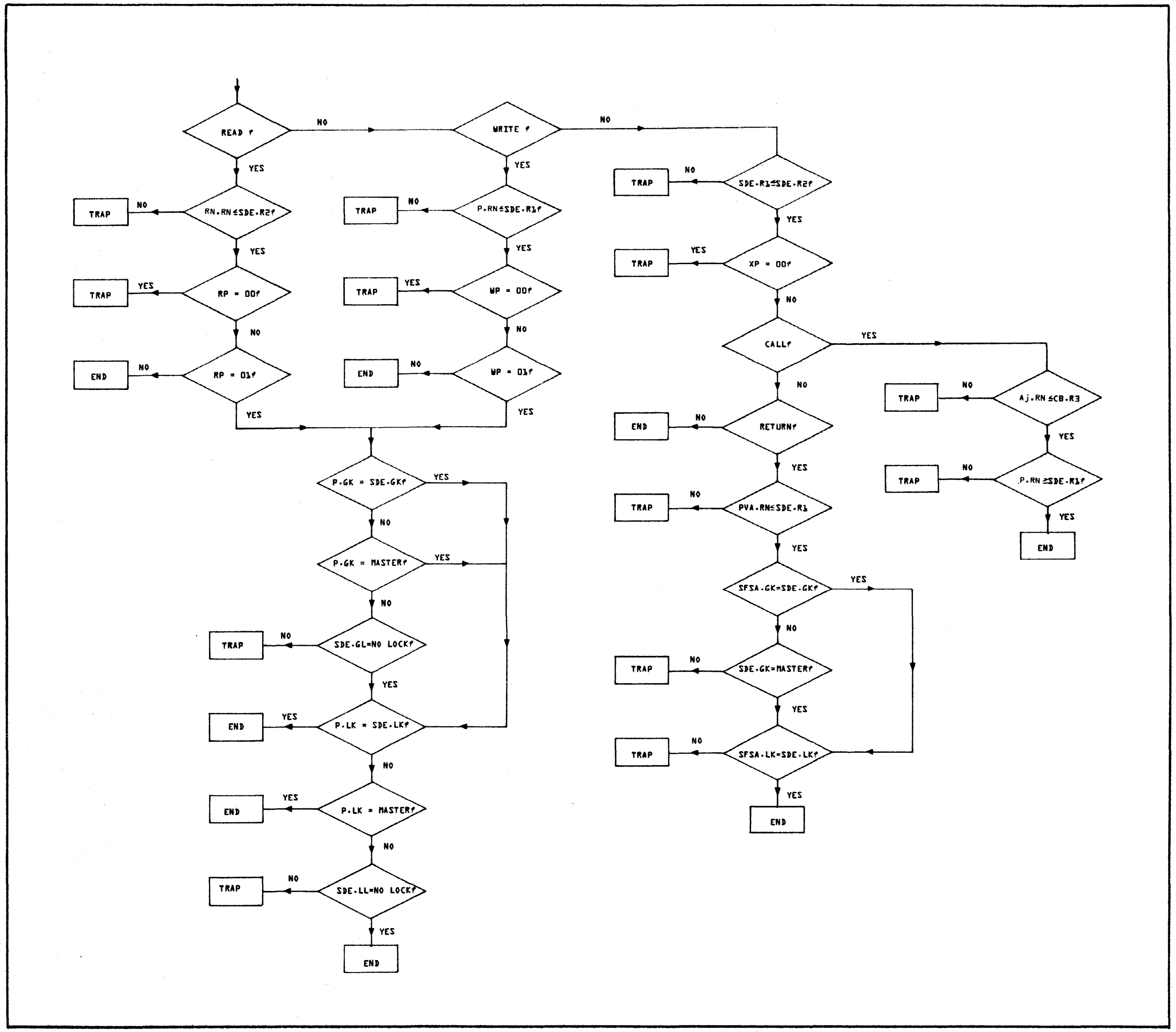


Figure 3-15. Virtual Memory Protection Flowchart

To minimize the time necessary to translate a PVA to an RMA a number of hardware buffer memories are utilized. The description given here is based on P3 buffer memories. The organization varies from processor to processor, but the fundamental concepts are the same.

Figure 4-1 gives a pictorial representation of these buffer memories. The Segment MAP contains the most recently used entries from the Process Segment Table. In the first stage of address translation the processor uses this MAP to translate the PVA to an SVA. This SVA is then transmitted to the cache memory and the Page MAP. Each of these buffers are organized on the basis of the SVA, the Page MAP containing the most recently used entries in the SPT, and the cache containing the most recently used words in system virtual memory. Simultaneously, a search is made of the Page MAP and cache. If a cache hit occurs, then no further action is required. However if the required data is not in cache, then the search of the Page MAP is relevant. If a hit occurs the required address translation completes and central memory may be accesssed via the appropriate RMA. Only when there is no hit in the page MAP must the processor actually search the SPT in real memory.

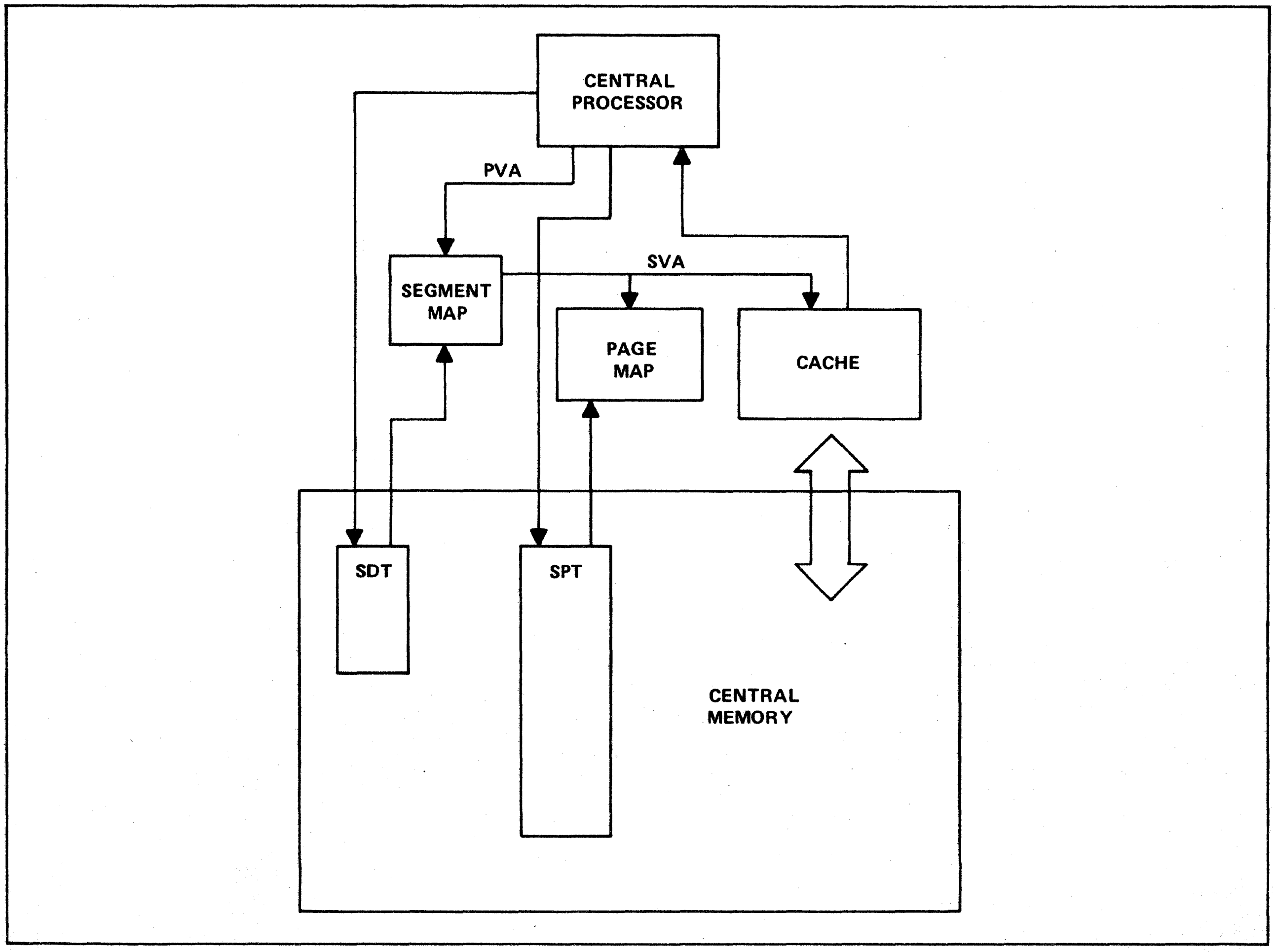


Figure 4-1. CYBER 180 Buffer Memories

SEGMENT MAP

The purpose of the segment MAP is to translate a segment number (SEG) to an Active Segment Identifier (ASID). This is the first step in the address translation mechanism and translates a PVA to an SVA. Figure 4-2 illustrates the general process. A set associative technique is employed whereby an index is used to select a set, and then an associative (simultaneous) comparison is made between each entry in the set and the required segment number. P3 has 16 such sets in its Segment MAP, each set having two members. To index into the MAP the lower four bits of the segment number are used as a hash index. These bits are the most random part of the segment number.

The hash index identifies one of two entries in the segment MAP which are candidates for translation of the given SEG. The segment MAP simultaneously compares the set tag entries with the mode of operation (job/monitor) and the upper eight bits of the SEG. If a hit is made, then the ASID is taken from the segment descriptor word held in the MAP. If no hit occurs, then the ASID must be fetched from the segment descriptor table in real memory.

The tag field of the segment MAP contains a bit to indicate which entry in the two sets is the least recently used (LRU). There are only two candidates. This entry is then used to receive the new segment descriptor. The tag field does not contain the segment table address (STA). Instead two registers are used: one for the job STA and one for monitor. During an exchange to monitor state the monitor STA is compared to that obtained from the monitor exchange package. Similarly with the job STA when an exchange to job state occurs. If the values do not compare, then all entries for either job or monitor in the segment MAP are invalidated.

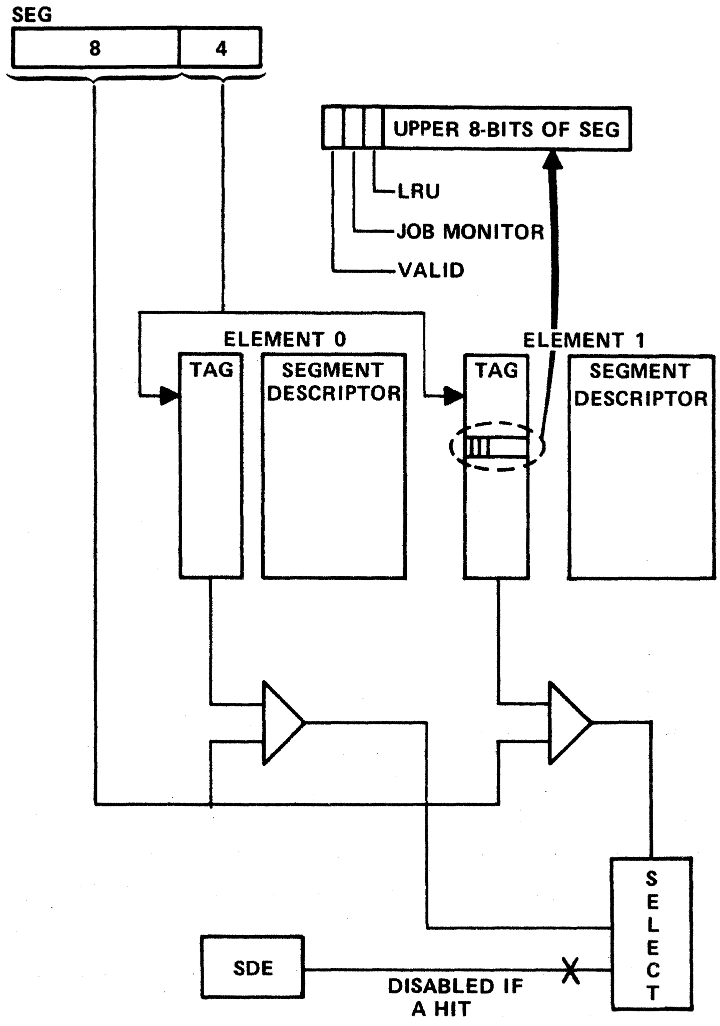


Figure 4-2. Segment Map Operation

The most recently used segment numbers appear in the MAP. Hence, the more segments used by a process the less likely it will be to find the entry in the MAP. The system performs most efficiently if the MAP entries for monitor and job are not hashed to the same location in the MAP. This is best handled by the operating system assigning job segment numbers sequentially from zero, and monitor segment numbers from FFF downwards sequentially. This has the affect of creating a 32 entry buffer which is filled from the top with job segment descriptors, and from the bottom with monitor segment descriptors (figure 4-3). The choice of a starting segment number for monitor need not be FFF, but should be of the form XXF. In fact, FFF will probably be used for some special purpose by the operating system, and, in any case, would maximize the dead space in the monitor segment table. The practical choice for the starting segment number is computed from:

(number of monitor segments) .OR. 00F

in which case the maximum number of dead entries will be 15.

When the MAP is degraded (due to a parity error) one set is eliminated. This means that the probability of a miss is heightened and performance degrades.

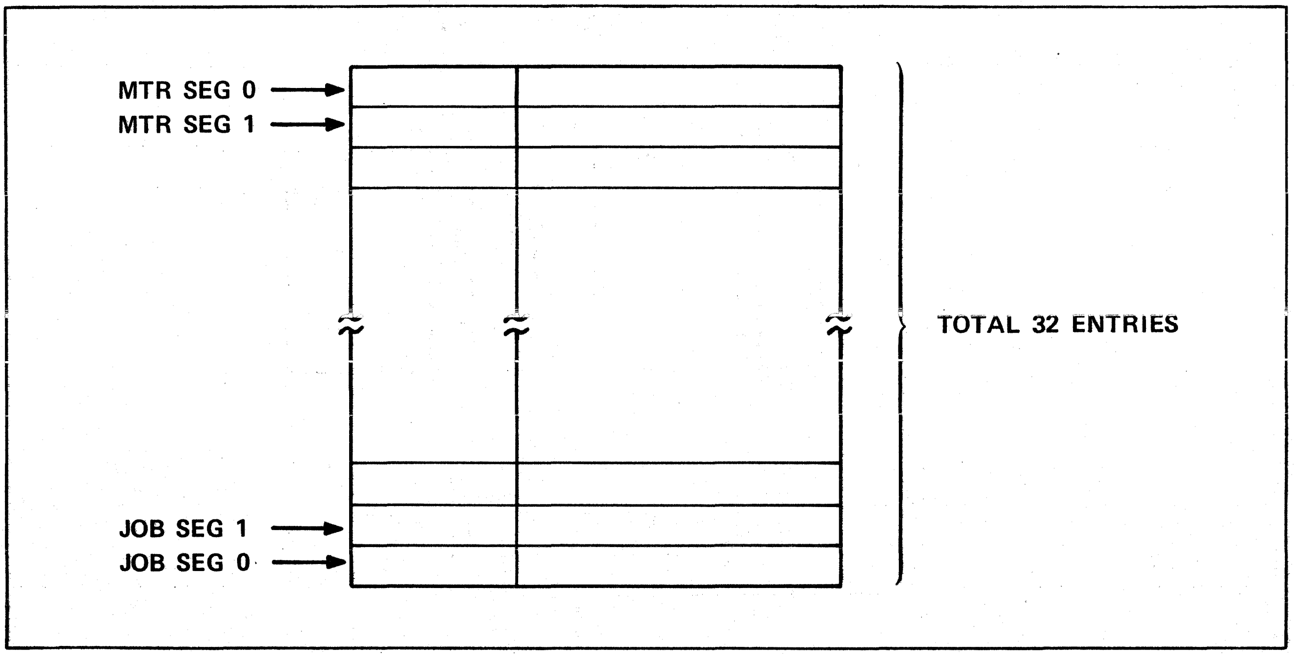


Figure 4-3. Segment Map Allocation

PAGE MAP

The purpose of the page MAP (figure 4-4) is to translate the SVA from the segment MAP into an RMA. As with the segment MAP a set associative technique is used. In this case there are 32 sets, and the low order five bits of the page number are used as a hash index to select a set. The page number is formed from the byte number by executing a logical product with the page size mask. Depending on the page size the page number will not be right justified, and the hardware performs the necessary justification before extracting the hash index.

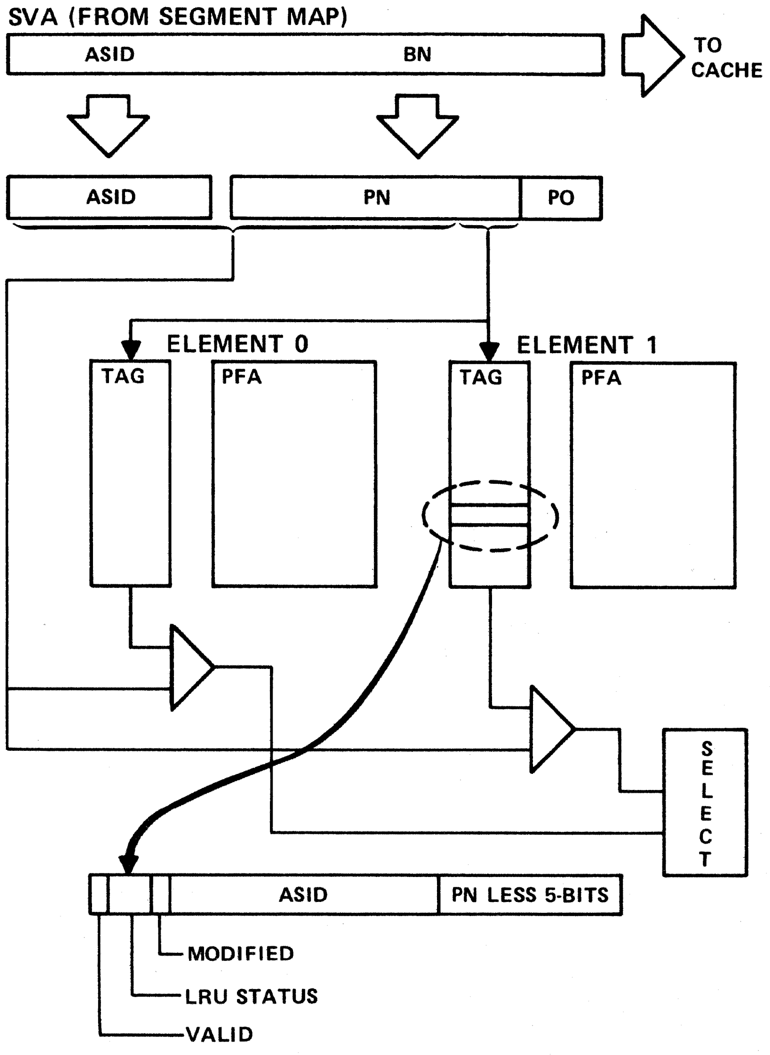


Figure 4-4. Page Map Operation

The page MAP simultaneously compares the set tag entries with the high-order 33-bits of the SVA. Note that the valid bit is not included in this operation. Invalid PVA´s (and therefore invalid SVA´s) do not get this far in the translation mechanism. If a hit is made, then the RMA is formed from the SVA in the page MAP data table, and the page offset. Otherwise, a page table search is initiated.

The tag field in the page MAP contains two bits to indicate which entry in the two sets is the LRU. There are only two candidates. However, two bits are allocated on P3 to allow for up to four entries per set.

The most recently used pages appear in the page MAP. Hence, the more pages used by a process, the less likely it will be to find the entry in the MAP. When the page MAP is degraded, one group of entries is eliminated. The probability of a hit is reduced, and performance degrades.

The modified bit is carried in the page MAP, but not the used bit. Actions taken on a hit and a miss are described below, and clarify the setting of these bits:

1) MAP Miss - Page Table Hit

Read : (i) Set the used bit in the PTE
(ii) Copy the modify bit to the MAP
(iii) Copy the addresses to the MAP

Write : (i) Set used and modify bits in the PTE
(ii) Copy the modify bit to the MAP
(iii) Copy the addresses to the MAP

2) MAP Hit

Read : (i) Simply form the RMA - no page table access is necessary

Write : (i) If the modify bit is set in the MAP, then the process is identical to read.

(ii) If the modify bit is not set in the MAP, then a page table search is required to set the modify bit in the PTE. The same bit is set in the MAP. At this time the modify bit in the PTE and in the MAP is set, and the used bit is set in the PTE.

The MAP and the cache perform similar functions. Once the segment MAP has formed an SVA it sends it simultaneously to the page MAP and the cache. If there is a cache hit, and the operation is a read, then there is no need to access the page MAP, data being read directly from cache.

On a read the cache hit overrides everything. Consequently, it is possible to to get a cache hit even when the relevant page is not in central memory. This is because the cache is organized on the SVA. The operating system must ensure that cache accurately reflects the contents of system virtual memory at all times. Whether the data actually resides on disk or in real memory is immaterial. On writes, the situation is different. CYBER 180 processors always write through cache. This means that the appropriate entry in cache is either updated or purged on a write. Actual implementation is processor model dependent. On P3 when a cache hit occurs on a write, if it is a full-word write, then the word is updated. For a partial-word write, the word is purged. On another processor cache is updated regardless of the nature of the write.

CACHE MEMORY

CYBER 180 supports very large, cost effective memories. It achieves this at the expense of some memory speed, and to make up for this loss of speed a buffer memory, (cache memory), is placed in the faster processors. The most recently used words in system virtual memory are held in a much smaller, faster memory. The management of this memory is in figure 4-5. A set-associative technique is used to control entries in the cache. On P3 a maximum of four entries per associative set are employed. An entry in a set consists of a tag field, which identifies the entry, and 32 bytes (4 words) of data which are termed a BLOCK. There are 256 sets on P3.

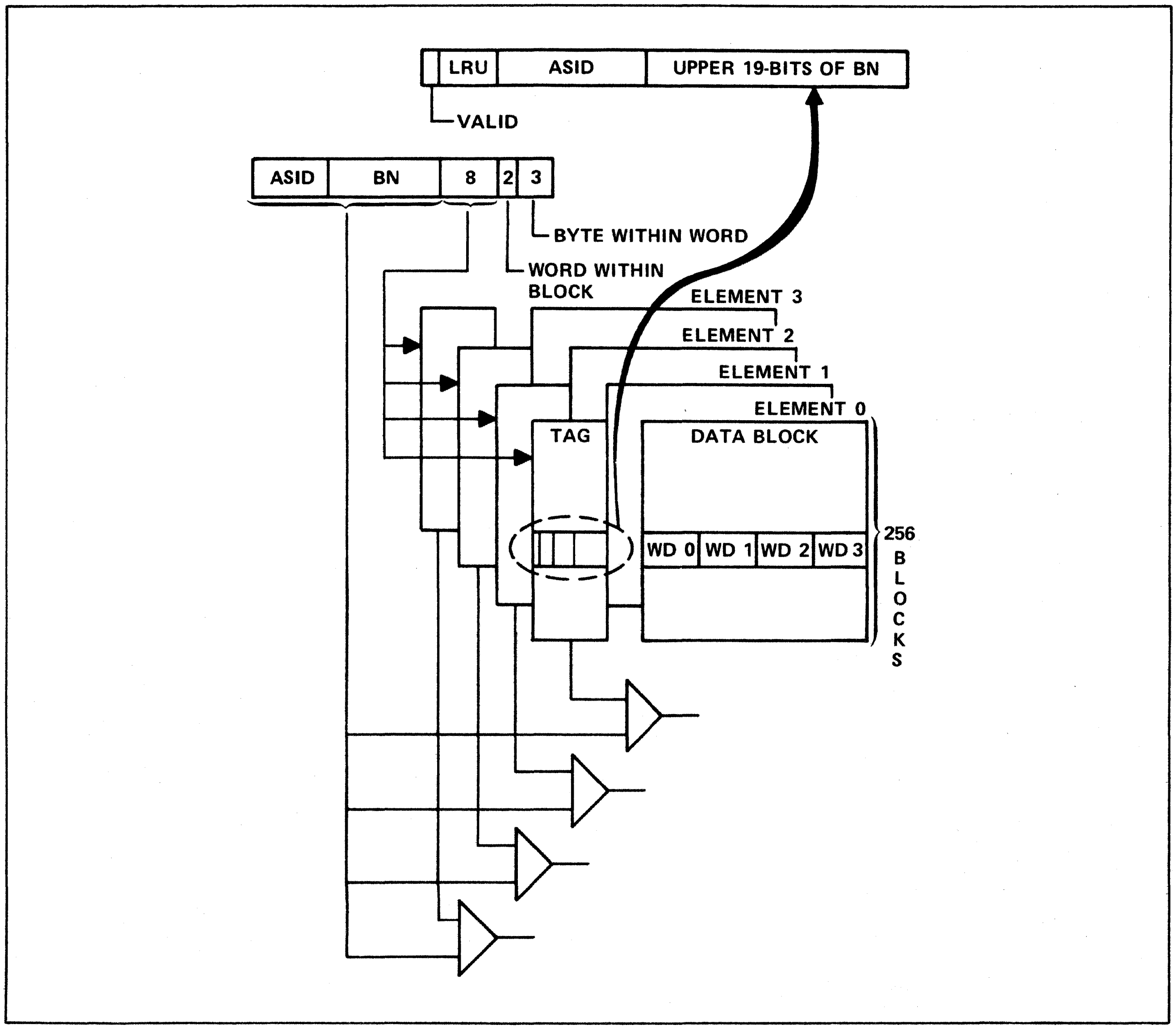


Figure 4-5. Cache Memory Operation

Bits 51-58 of the SVA are used as a hash index into the sets. These represent the most random part of the SVA. The low-order five bits of the SVA represent the word within block, and the byte within word respectively. Note that the ASID does not enter into the hash index computation. This is deliberate since in CYBER 170 State only a single segment (ASID = FFFF) is used, and this has no randomness.

Once a set has been selected, a simultaneous comparison of the upper 35 bits of the SVA and the tag entries is made. If there is a hit, and the entry is valid, that entry is used. If there is no hit, then a set is chosen for the new entry and the appropriate words read up. Entries are chosen first on the basis of their validity, and then on their LRU status. Whenever a new entry is made in a set an entire block (four words) is read up, starting with the required word and proceeding left to right unless the instruction is a right to left (BDP numeric) type. Cache regards central memory as a series of four-word blocks which always start on a block boundary.

If, at any time, cache is not busy after it found a hit, then it automatically looks ahead one block. If it gets a hit, then the sequence ends, otherwise it initiates a read on that block.

Cache will always be organized on SVA for C180 processors.

SOFTWARE IMPLICATIONS

There are several software implications in the use of the cache and the MAP´s, particularly in a multiprocessing environment. It is incumbent upon the operating system software to ensure that stale data does not exist at any time in the MAP or the cache (CYBER 180 physical I/O and memory writes performed by another processor do not update automatically the processor local cache). The following guidelines should be followed by the software:

1) Whenever a page table entry is changed the Page MAP must be purged. Not only the Page MAP in the processor updating the page tables but in the second processor, if available. Care must be exercised by the software at this time, and to some extent, the hardware depends on the software to take certain precautions . The reason for this is the noninterruptibility of the CYBER 180 instructions. Before an instruction is placed in execution it is prevalidated. The hardware ensures that all pages required to complete the execution of the instruction are in memory before execution commences. Once execution starts, the processor assumes that the pages it requires will remain there. Hence, a second processor must not delete a page from memory without first notifying the other processor. A typical sequence of events is:

   (1) Set the invalid bit in the PTE - This ensures that an instruction cannot start which requires this page, but that it can complete if it has already started. In other words, the processor ignores the valid bit once an instruction has been prevalidated.

   (2) Send an interrupt to the second processor asking to purge MAP.

   (3) First processor waits for acknowledgement from second processor that MAP has been purged.

   (4) First processor updates the page table entry.

   (5) First processor sets valid bit in PTE.

   Since the valid bit was dropped prior to sending the interrupt, no instruction can be started using the page which is absent or deleted. An instruction making such a reference would cause a page fault, and this page fault will not be processed until the in progress page table update has been completed. This is another interlock which must be set up by the O/S software. That is, only one processor can execute a page table update at one time.

   Notice also that when a page table update is made, cache memory need not be purged if the operation is a write, since writes always write through cache memory, prevalidation will ensure that the page exists in memory. If the operation is a read, even though the page has been purged from memory, the copy in cache memory is still good, and the hardware will use this copy as has already been described.

2) There is a danger, in a multiprocessor environment, of the cache becoming stale whenever a processor is assigned to a job. At this time, the O/S should check the LPID (Last Processor ID) field in the Job Exchange package against the processor ID (PID). If the quantities are not identical, then cache must be purged.

These are not the only times when cache and the MAP must be purged. It will be seen in a later section (Purge Buffer) that similar problems arise during I/O. The points made here are merely illustrative. There has to be a cooperative effort between the hardware and software, and great care must be exercised when designing for the multiprocessor environment.

This chapter discusses processor state and process state registers. Processor state registers define the operational state of the processor without regard to a specific process. Process state registers define a specific process.

PROCESSOR STATE REGISTERS

Each processor has a set of registers which define the operational state of the processor. These registers are described fully in the MIGDS, however, several points are of interest here:

1. CYBER 180 has an exchange mechanism, similar in function to CYBER 170, which executes quite differently from CYBER 170. Whereas on CYBER 170 a true exchange occurs (that is, the operating registers are stored in memory and loaded with the contents of those same memory cells), on CYBER 180 the operating registers (process state registers) are stored in one area of memory and loaded from a different area in memory. Since an exchange jump always changes the operating mode from job to monitor, or vice versa, two exchange packages are located in memory: a monitor exchange package and a job exchange package. These exchange packages are located at real memory addresses specified by the Job Process State (JPS) and the Monitor Process State (MPS) registers. They must not be located at the same address, nor must they overlap. Finally, they must be on a double word boundary. To this end, the least significant four bits of the JPS and MPS are ignored (treated as zeros - figures 5-1 and 5-2).

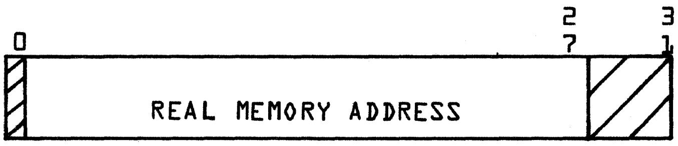


Figure 5-1. JPS and MPS Registers

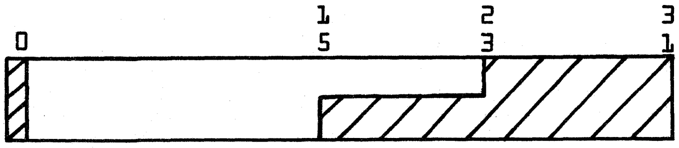


Figure 5-2. PTA Register

2. Two registers the Page Table Address (PTA) and the Page Table Length (PTL) specify the size of the Page Table. The Page Table must be located on a boundary which is zero modulo the Page Table Length. The reason for this is that the hardware accesses the Page Table frequently and computes an index for this purpose. To find the address of the required entry, instead of adding the index to the PTA it is simply catenated - a much faster operation. Depending on the page table length the low-order 9-17 bits of the PTA must be set to zero.

   The PTL, which indicates the length of the Page Table, is simply used as a mask which is used to ensure that a hash index with the page table remains within the bounds of the page table. Its use is described in the section dealing with virtual memory.

3. The Page Size Mask (PSM) specifies the page size to be used. The page size may be chosen from 512 bytes to 64K bytes. However, typical page sizes are expected to be 2KB and 4KB. As with the PTL, the use of the PSM is discussed fully in the virtual memory section.

4. Two registers deal with equipment identification - the Element ID (EID) and the Processor ID (PID). The first is a unique, world-wide identification, the format is shown in figure 5-3. The second (the PID) is a abbreviated version which uniquely identifies an equipment within a system. The PID is used on exchanges to identify the Last Processor ID (LPID), and is used in a self-discovery process during system initialization. A third register - Options Installed (OI) - completes the description of the equipment. This is a 64-bit register which indicates the number of PP´s, cache memory size, ports to central memory, and so forth.

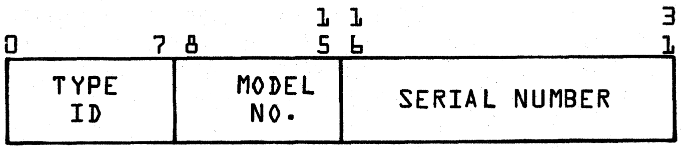


Figure 5-3. Element ID Register

5. There is a 32-bit microsecond counter - the System Interval Timer (SIT) - which counts down, and is used to establish job time slices.

6. One final register is of interest at this stage and that is the Virtual Machine Capability List (VMCL). Many of the CYBER 180 processors are microprocessors and the microcode may describe various machines which are termed virtual machines. CYBER 180 is one such virtual machine but many others are possible, in particular CYBER 170. This 16-bit register controls the virtual machines the user (customer) is permitted to run. For example, a CYBER 180 customer who has not purchased the CYBER 170 emulator is prevented from executing CYBER 170 code via the register.

The remaining processor state registers (there are several) deal with the operational status of the processor and its maintenance. Many of these registers are model dependent.

Access to these registers is controlled. Most registers can be read and written from the Maintenance Control Unit (MCU), and can be read from the processor. However, registers can only be written when the appropriate privilege has been granted. Access to the registers is illustrated in the figure 5-4.

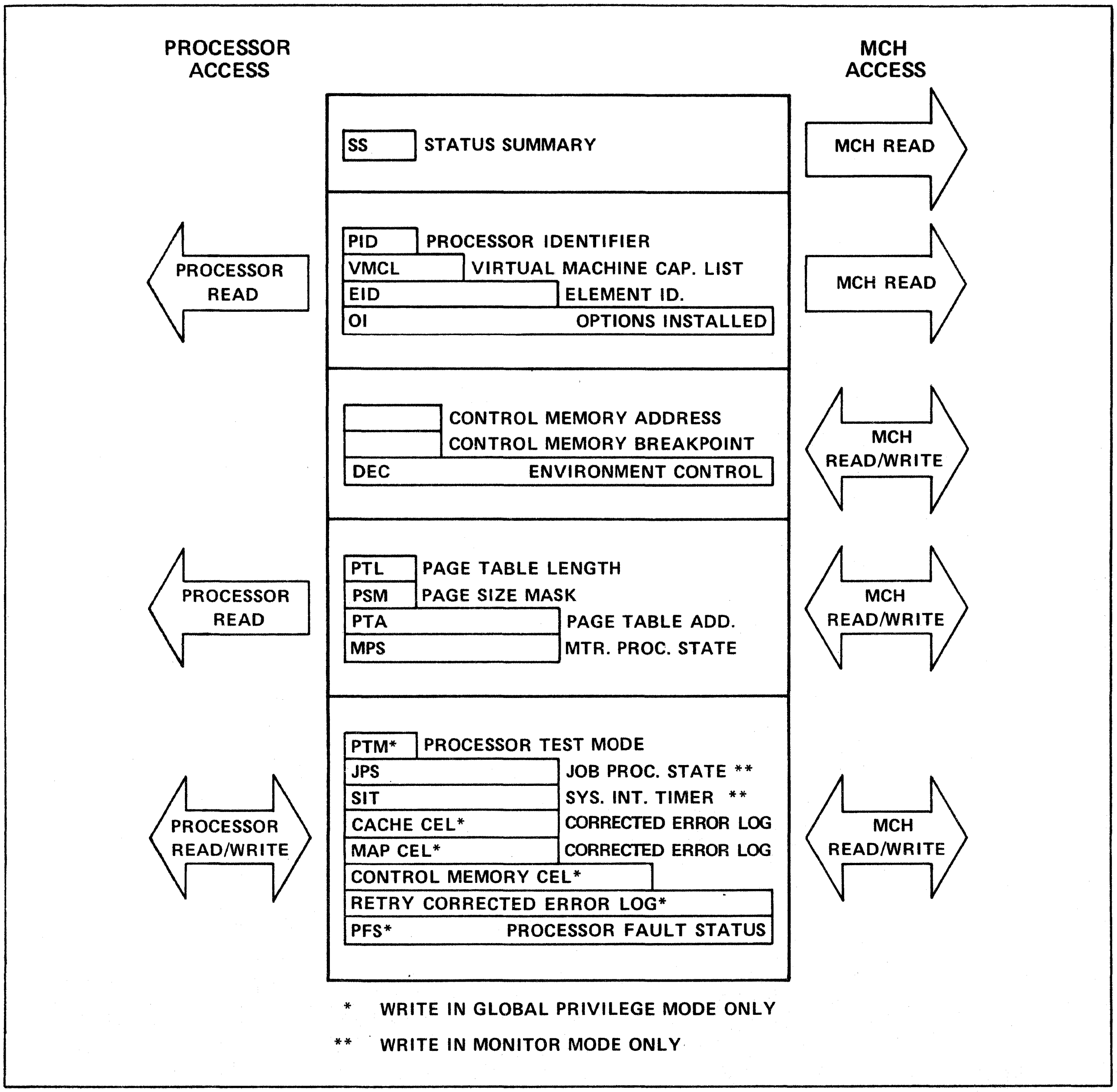
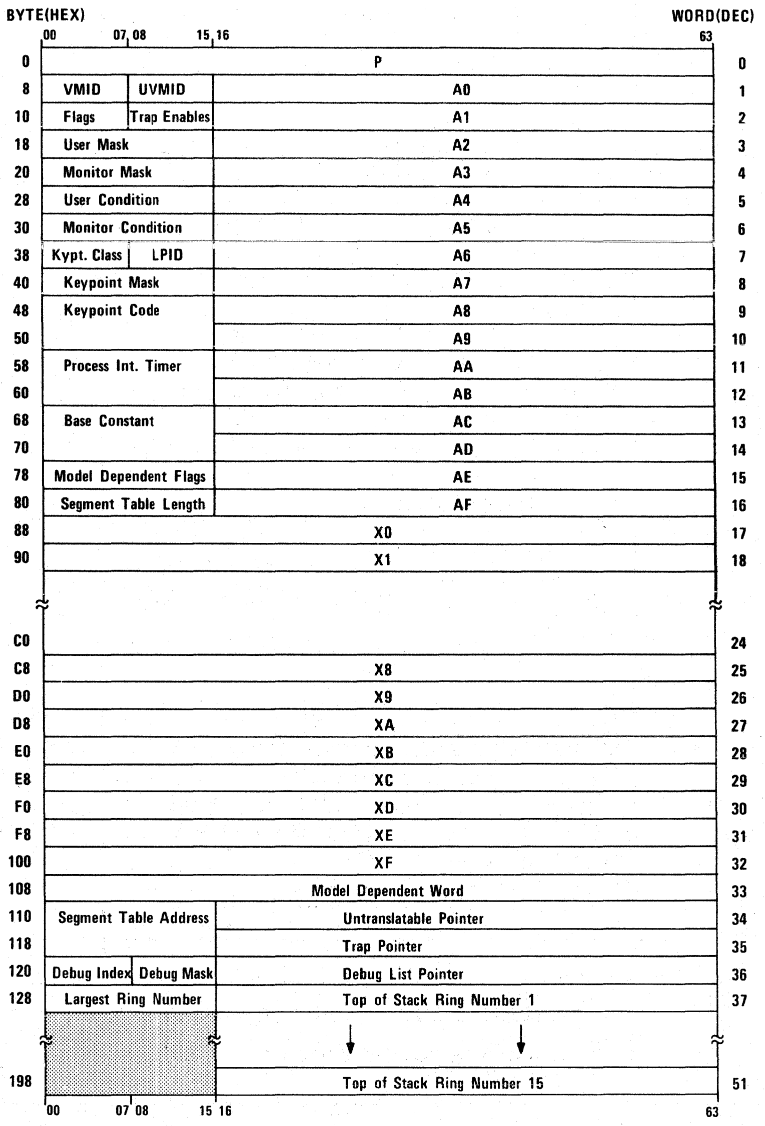


Figure 5-4. Processor State Registers

PROCESS STATE REGISTERS

There is a large set of registers which define each process state, these include the P, A, and X registers. These registers completely describe the operational environment of a job or process, and if the process is interrupted for any reason that environment must be captured in order for processing to resume after the interrupt has been dealt with. This is accomplished by the exchange mechanism during which all the process state registers are saved in an exchange package (figure 5-5), and a fresh set of registers (defining the process exchanged to) are loaded from a second exchange package.



BYTE(HEX)	00 07	08 15	16 63	WORD(DEC)
0	P			0
8	VMID	UVMID	A0	1
10	Flags	Trap Enables	A1	2
18	User Mask		A2	3
20	Monitor Mask		A3	4
28	User Condition		A4	5
30	Monitor Condition		A5	6
38	Kypt. Class	LPID	A6	7
40	Keypoint Mask		A7	8
48	Keypoint Code		A8	9
50			A9	10
58	Process Int. Timer		AA	11
60			AB	12
68	Base Constant		AC	13
70			AD	14
78	Model Dependent Flags		AE	15
80	Segment Table Length		AF	16
88	X0			17
90	X1			18
≈				≈
C0				24
C8	X8			25
D0	X9			26
D8	XA			27
E0	XB			28
E8	XC			29
F0	XD			30
F8	XE			31
100	XF			32
108	Model Dependent Word			33
110	Segment Table Address		Untranslatable Pointer	34
118			Trap Pointer	35
120	Debug Index	Debug Mask	Debug List Pointer	36
128	Largest Ring Number		Top of Stack Ring Number 1	37
≈	▨	▨	↓ ↓	≈
198	▨	▨	Top of Stack Ring Number 15	51

Figure 5-5. CYBER 180 Exchange Package (CYBER 180 Process)

The process state registers are summarized below. The 33 basic oL and if the process is Perating registers (P, A and X registers) are described elsewhere in this document. This section will cover the remaining process state registers. The VMID designates the virtual machine to which control is being transferred. VMID's of zero (CYBER 180) and one (CYBER 170) have been defined for the CYBER 180 processors. The UVMID is a register used to designate an invalid (undefined) VMID to which the processor attempted to transfer control. If an exchange jump is attempted to a nonexistent virtual machine, then the exchange completes, and a second exchange interrupt occurs immediately on an Environment Specification Error. This is when the UVMID is set to identify the fault to the operating system.

A series of flags are located in word 2 of the exchange package. These are: The Critical Frame Flag (CFF); the On Condition Flag (OCF); the Keypoint Enable Flag (KEF); and two flags to control trap interrupts. These are primarily software flags which are carried by the hardware. Their usage is described in later sections of this document.

The User and Monitor Mask Registers and Condition Registers are used to control interrupts and are discussed fully in the section dealing with interrupts. Similarly, the Keypoint Class, Keypoint Mask and Keypoint Code Registers are described in the section dealing with Keypoint. These registers control the keypoint process.

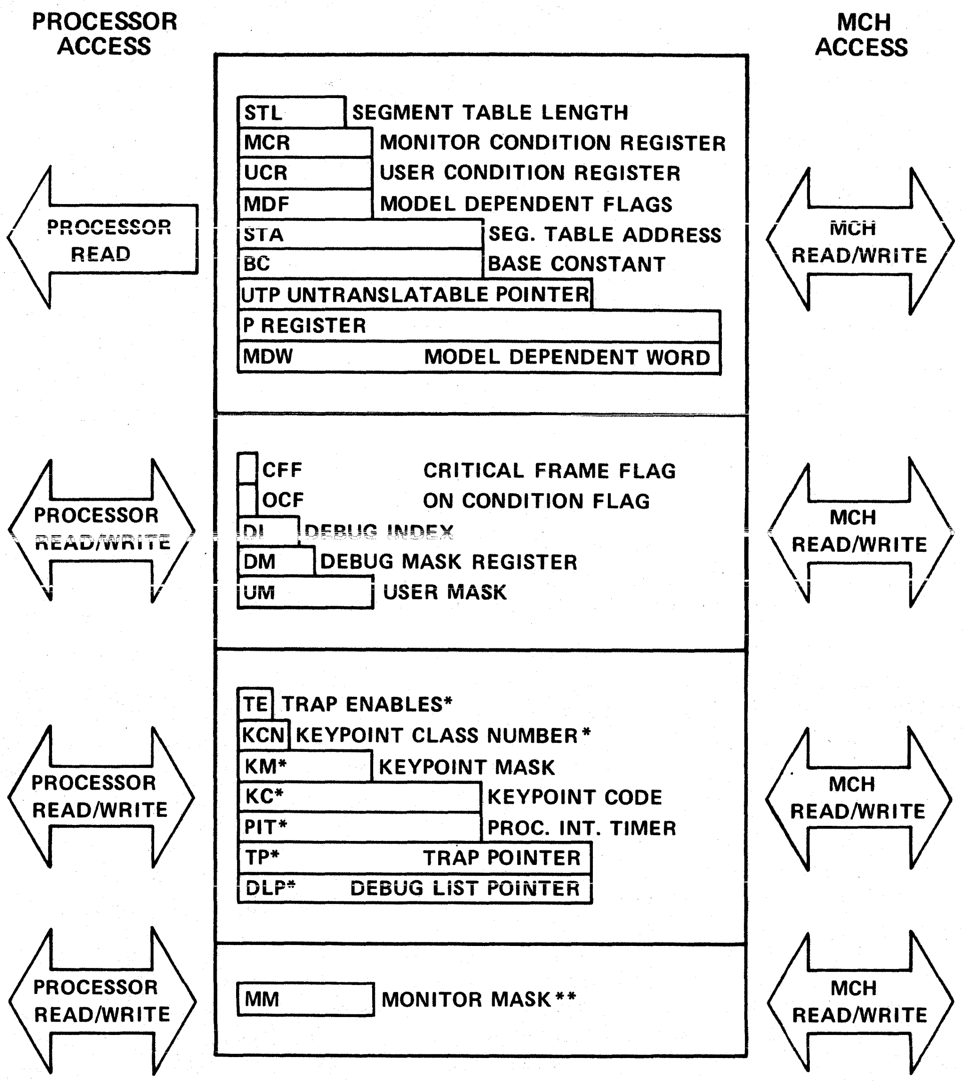
The LPID (Last Processor ID) has already been introduced (refer to Cache Memory). It records the PID of the processor executing a given exchange interval. The PIT (Processor Interval Timer) is a 32-bit microsecond timer analagous to the SIT. It counts down, at a microsecond rate, and interrupts the processor whenever it reaches zero. It is used for timing within a given task (or process).

The Base Constant is a register used by the O/S as an index to a control point area for an executing task. The STA and STL (Segment Table Address and Length) specify the RMA and length of the SDT to the hardware. Remember, the SDT is a hardware table used in the virtual memory address translation and, as such, it must be located at a real memory address. The combination of the STA and STL also uniquely define the task address space.

The model dependent flags and word are used by the hardware, typically, to help in hardware checkout. They do not have any particular significance to the software. The Debug Index, Debug Mask and Debug List Pointer are used to control the debug facility, and are discussed fully in a later section.

The Trap Pointer carries the address of the trap handler to be used by an executing task. It is discussed in the section dealing with interrupts. Likewise, the Untranslatable Pointer (UTP) is also covered in the interrupt section. This register holds the pointer or address which could not be translated, causing an exchange to O/S monitor. Finally, there are 15 Top of Stack Pointers, one for each ring of execution. Their utilization is covered in the section dealing with Call/Return. On most processors (at least P1-P3) these pointers are not kept in live registers but reside in the exchange package in central memory. The Largest Ring Number Register has been included in the event that the Top of Stack Pointers are kept in live registers. In which case the hardware could be organized such that the exchange mechanism would only have to exchange those pointers actually in use in the process.

As with the processor state registers, access to the process state registers is carefully controlled. This access is illustrated in figures 5-6 and 5-7.



* WRITE IN LOCAL PRIVILEGED MODE ONLY

** WRITE IN MONITOR MODE ONLY

NOTE: ONLY THOSE PROCESS STATE REGISTERS WHICH MAY BE ACCESSED VIA THE PROCESSOR COPY INSTRUCTION OR VIA THE MAINTENANCE CHANNEL ARE SHOWN.

Figure 5-6. Process State Registers

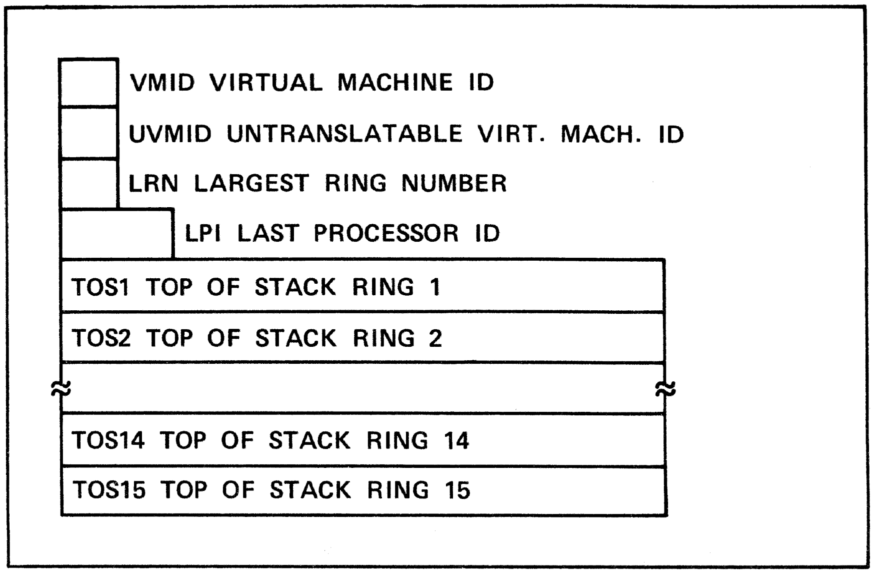


Figure 5-7. Process State Registers Accessed by Exchange Operation

First of all, the CYBER 180 interrupt system is hierarchical. That is, a process may be interrupted and control transferred to an operating system interrupt handler. Depending on the status of this new environment, it may be interrupted itself but via a different mechanism. The two basic interrupt mechanisms are termed: exchange interrupts and trap interrupts. Both forms of interrupt save the current environment (as described by the process state registers) and transfer control to some other code module. In the case of an exchange interrupt, control transfers from a user or subsystem address space to the monitor address space. Trap interrupts, on the other hand, are processed within the address space of the current process.

Trap interrupts are controlled by two process state registers: the trap enable flip-flop (TEF) and the trap enable delay flip-flop (TED). The settings of these registers are controlled by the exchange mechanism. Hence, it is the software designer´s choice whether a monitor exchange interrupt is handled with traps enabled or disabled. This is an important design decision as will be seen later on.

Two pairs of process state registers are used to monitor interrupts and control the actions taken when a condition arises which may interrupt a process. These are the Monitor Condition Register (MCR) and the Monitor Mask Register (MM), and the User Condition Register (UCR) and User Mask Register (UM). The condition registers are normally filled with zeros. Each bit in the registers corresponds to a particular interrupt condition and when that condition is encountered, the bit is set to indicate that fact. For each bit in the condition registers, there is a corresponding bit in the mask registers and when both bits are set, an interrupt is taken. In other words, the processor takes the logical product of the two register pairs, and then takes an interrupt if the result is nonzero (figure 6-1).

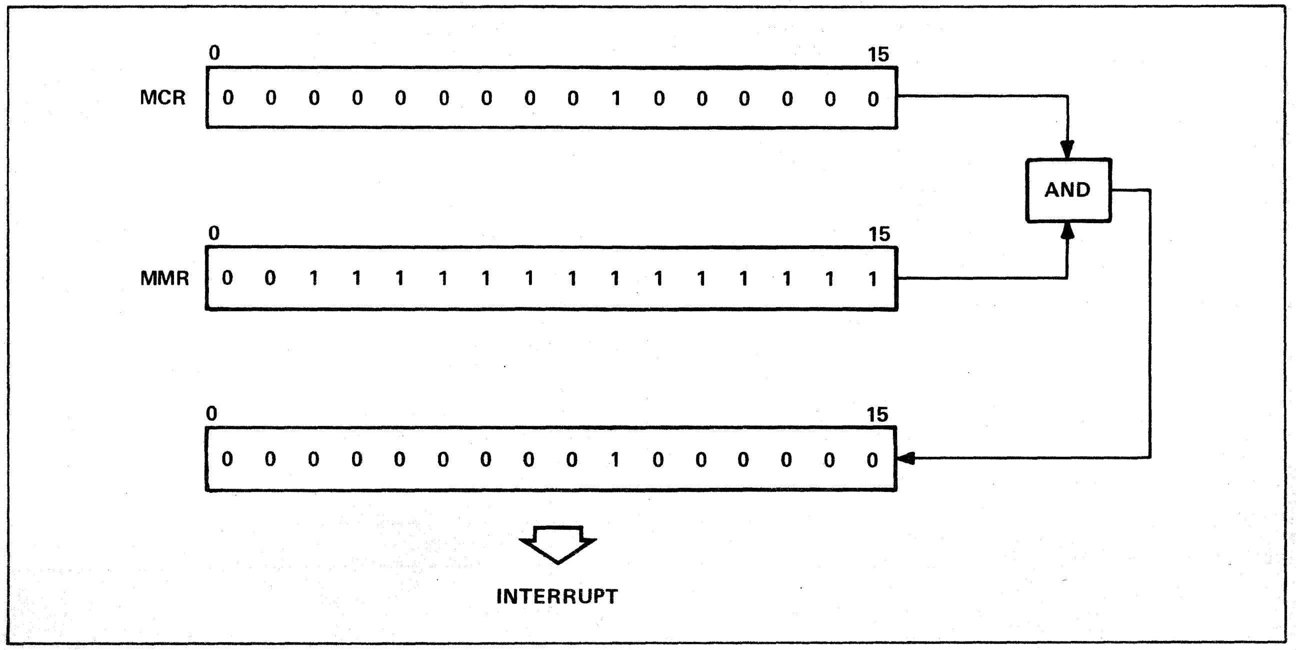


Figure 6-1. Basic Interrupt Mechanism

Now although there are only two condition registers (for the monitor and user), there are really four classes of conditions. They have been grouped into two registers simply for software convenience. The four classes are monitor conditions, system conditions, user conditions, and status indicators (figure 6-2). Conditions which are signaled in the MCR have a higher priority than (are acted on before) those flagged in the UCR. Notice that the MCR contains all system conditions, flags and most of the monitor conditions. The UCR contains all user conditions and some monitor conditions. The monitor conditions which are in the UCR are there so that the user may process them via a trap interrupt from within the user address space.

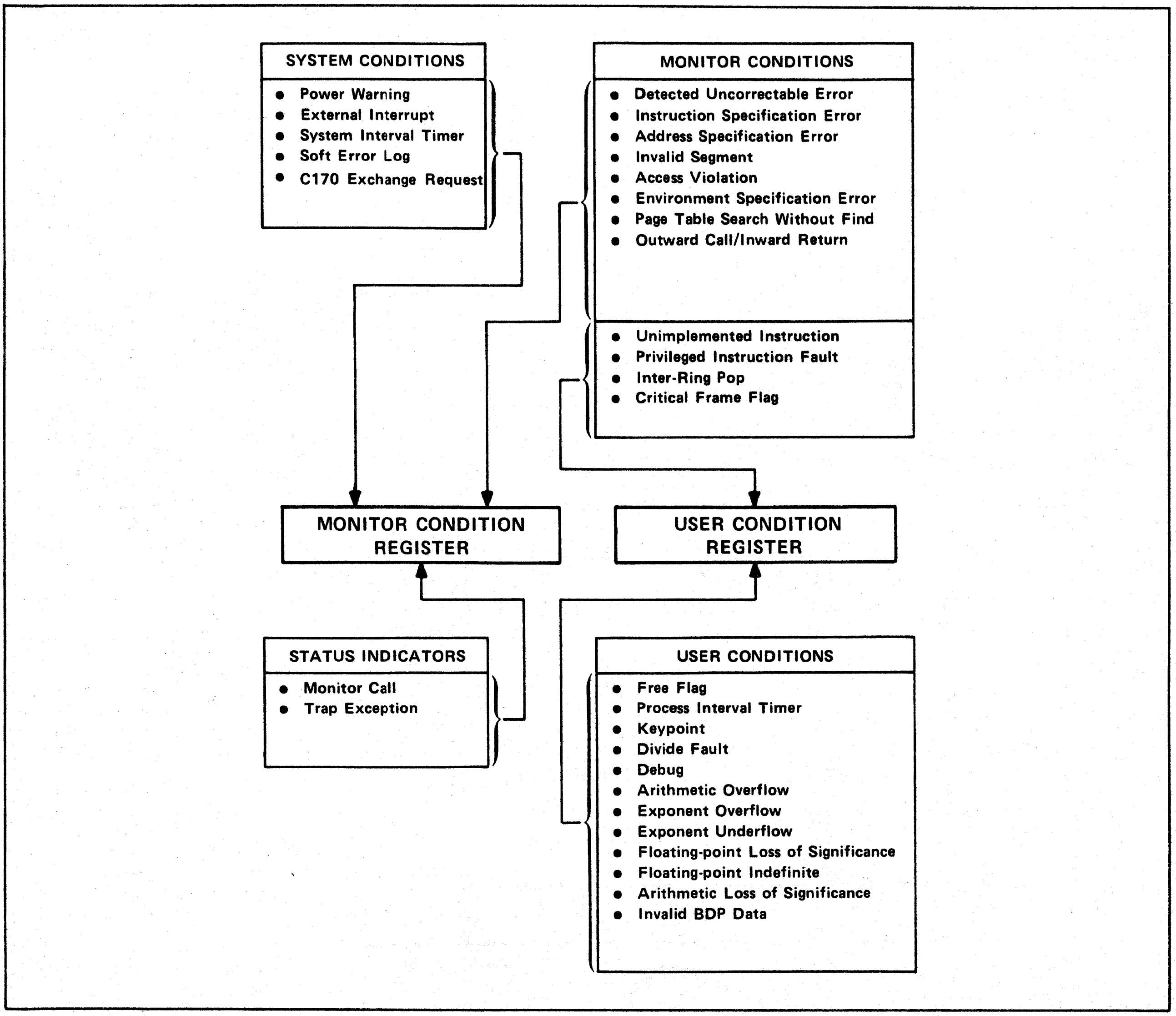
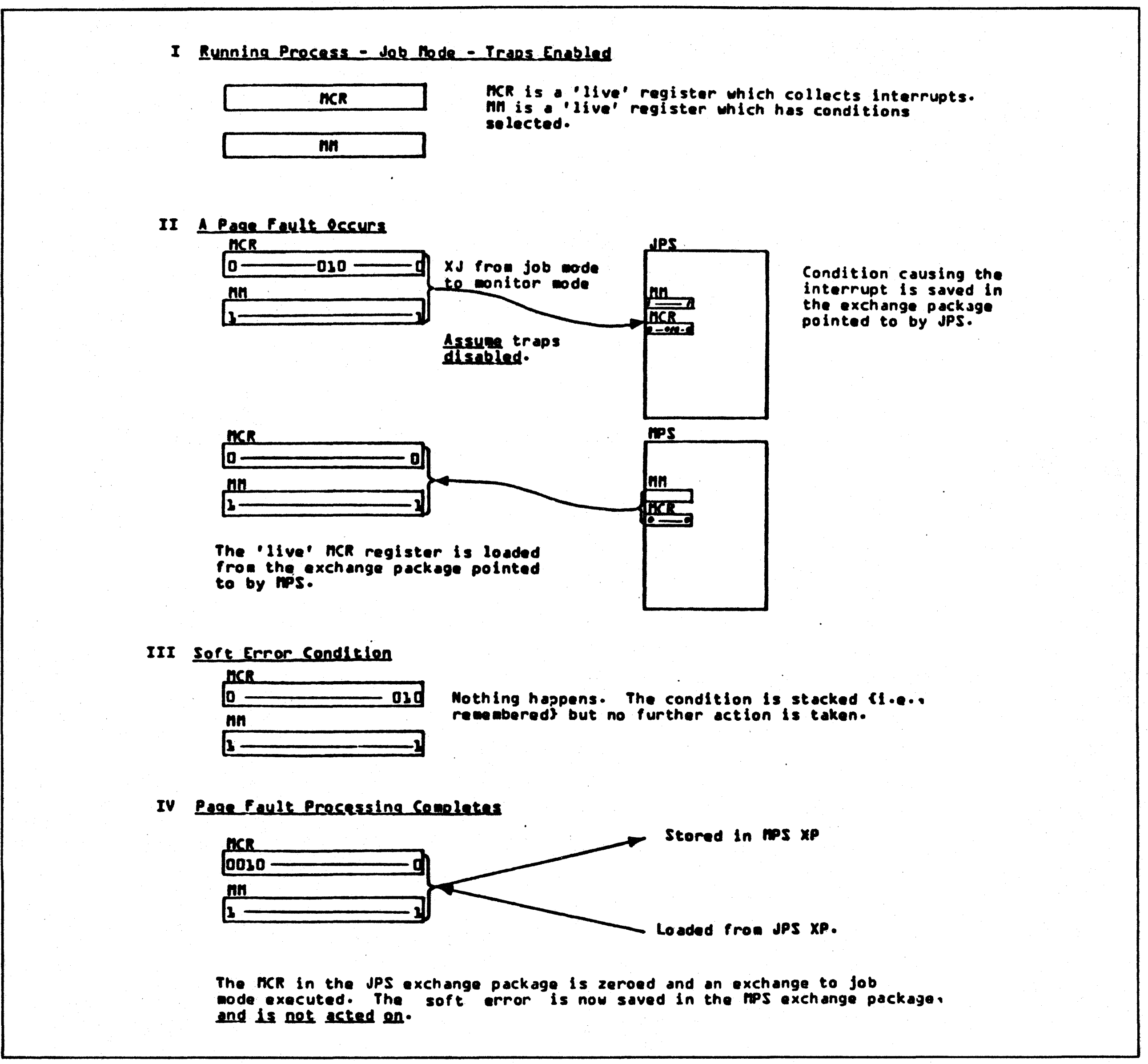


Figure 6-2. Interrupt Conditions

Monitor conditions are organized such that they are typically only encountered in job mode, the exceptions being uncorrectable errors which can occur at any time. When these conditions arise, an exchange jump from job mode to monitor mode takes place. A recurrence of the same condition (or another monitor condition) causes the processor to halt when traps are disabled. That is, with the exception of hardware diagnostics, the code executed in monitor state is arranged so that these conditions cannot arise. System conditions, on the other hand, occur any time, cause an exchange interrupt from job state to monitor state, and are stacked when encountered in monitor mode with trap disabled. This means that care must be taken when processing an interrupt to ensure that conditions are not lost.

Consider the following situation: The machine is in job mode, traps enabled and a page fault occurs (figure 6-3). During the processing of the page fault (in monitor mode), a soft error occurs. If traps are disabled, then this condition is simply remembered (stacked). When the page fault processing completes if an exchange is taken back to the process originally interrupted or another process, then the soft error is lost. It is stored away in the monitor exchange package. There is only one way to guarantee that this condition is not lost and that is to run in monitor mode, traps enabled. Testing the live MCR does not suffice, since subsequent to this test, an exchange back to job state must be made, and there is a finite time between the test and the point where the exchange is committed.



I Running Process - Job Mode - Traps Enabled

MCR is a 'live' register which collects interrupts. MM is a 'live' register which has conditions selected.

II A Page Fault Occurs

XJ from job mode to monitor mode

Assume traps disabled.

Condition causing the interrupt is saved in the exchange package pointed to by JPS.

The 'live' MCR register is loaded from the exchange package pointed to by MPS.

III Soft Error Condition

Nothing happens. The condition is stacked (i.e., remembered) but no further action is taken.

IV Page Fault Processing Completes

Stored in MPS XP

Loaded from JPS XP.

The MCR in the JPS exchange package is zeroed and an exchange to job mode executed. The soft error is now saved in the MPS exchange package, and is not acted on.

Figure 6-3. Examples of Interrupts

However, it is not necessary to have traps enabled for all monitor mode processing. The preferable sequence is to enter monitor with traps enabled, immediately disable traps, complete processing of the interrupt, enable traps, and return to job mode. Any conditions which have arisen during the interrupt processing are handled via an appropriate trap handler.

The interrupt system is hierarchical. The hierarchy does have a meaning and should be used. For conditions logged in the Monitor Condition Register, the hierarchy is:

```
               +--> STACK
             --+
EXCHANGE -> TRAP
             --+
               +--> HALT
```

Thus, an interrupt occurring in C180 job mode will cause an exchange to C180 monitor. An interrupt in C180 monitor with traps enabled will cause a trap. An interrupt in C180 monitor with traps disabled will cause either a stack or halt depending on the specific interrupt. It is incumbent upon the system to spend as little time as possible processing interrupts with traps disabled because a higher priority interrupt may be pending. Some care is necessary when designing the Operating System in this area.

The interrupt processing, as it affects the MCR, is very similar for the UCR. This register collects user conditions which typically lead to a trap interrupt. These conditions are best handled from within the user´s address space - built by a system routine. The hierarchy for these conditions is simply:

TRAP -> STACK

Thus, an interrupt with traps enabled will cause a trap whether in C180 job or monitor; an interrupt with traps disabled will be stacked.

In other words, the condition may be acted on or remembered. However, interrupt handlers are organized such that these conditions cannot arise, hence stacking will not occur very often. As has been previously stated, the relationship between the UCR and the User Mask Register (UM) is the same as that between the MCR and MM. If a particular condition has not been selected by the user in the UM then, effectively, it is stacked indefinitely. Certain instructions (floating-point arithmetic) yield results which could differ depending on the settings in the User Mask. This occurs when end-cases such as exponent overflow and underflow are encountered. Also held in the UCR are four monitor conditions. The hierarchy for these is:

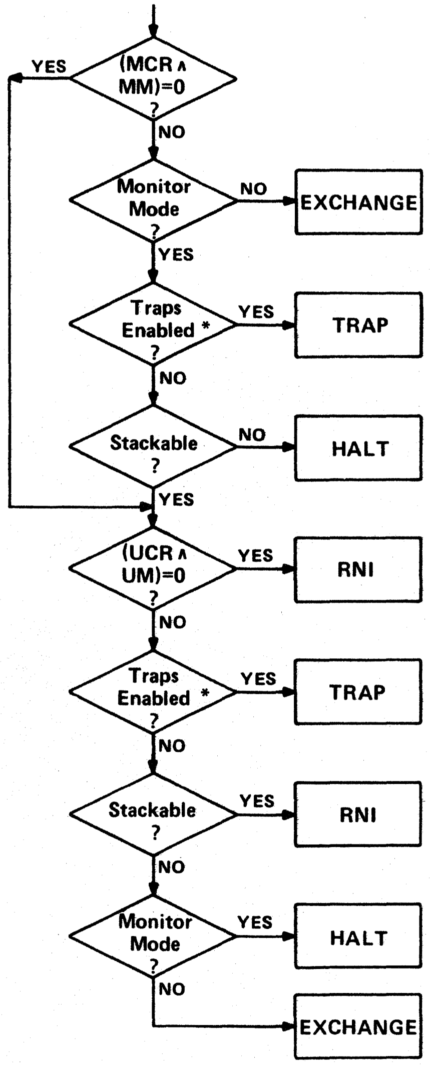
TRAP -> EXCHANGE -> HALT

Thus, an interrupt with traps enabled will cause a trap whether in C180 job or monitor. An interrupt with traps disabled will cause an exchange to C180 monitor when an interrupt occurs in C180 job mode and a halt when an interrupt occurs in C180 monitor mode.

The exchange and halt conditions should normally arise very infrequently or not at all since the interrupt handlers can be organized to prevent this. These monitor conditions have been placed in the UCR for specific reasons. For example, a trap on an unimplemented instruction is intended to be used for a software simulation of an instruction which is not in the repertoire of CYBER 180. This simulation must take place from within the users address space. Other monitor conditions in the UCR will have to wait until the system instructions have been discussed, in particular CALL/RETURN.

A more detailed discussion of the interrupt system where each condition is considered is postponed until the stack processing characteristics of CYBER 180 are described. Some final points will help to clarify the general process at this stage:

1. The overall scheme of events is represented in figure 6-4. In this flowchart stacked conditions lead to an RNI (Read Next Instruction). As indicated in the previous paragraph, this is a conceptual process only.



* TRAPS ENABLED MEANS THE TRAP ENABLE FLIP-FLOP (TEF) IS SET AND THE TRAP ENABLE DELAY FLIP-FLOP (TED) IS CLEAR.

Figure 6-4. Interrupt Flowchart

2. Conceptually, the hardware checks for interrupts before, during and after instructions. In actuality, only uncorrectable errors can occur at any point, and the wrap-up after one instruction and prevalidation for the next can become essentially a single process.

3. The hardware typically collects interrupts not between instructions, but between the instructions´ points of no return. There comes a point in every CYBER 180 instruction when something is written (memory, register file, and so forth). Once this happens, the instruction is committed, and, with the exception of hardware faults, interrupt conditions which arise apply up to the next point of no return (figure 6-5).

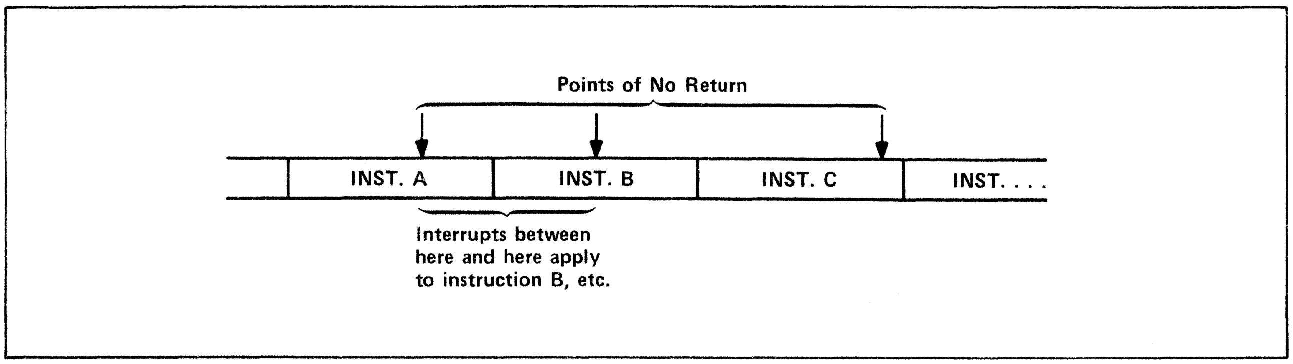


Figure 6-5. P3 Pipelined Instruction Stream

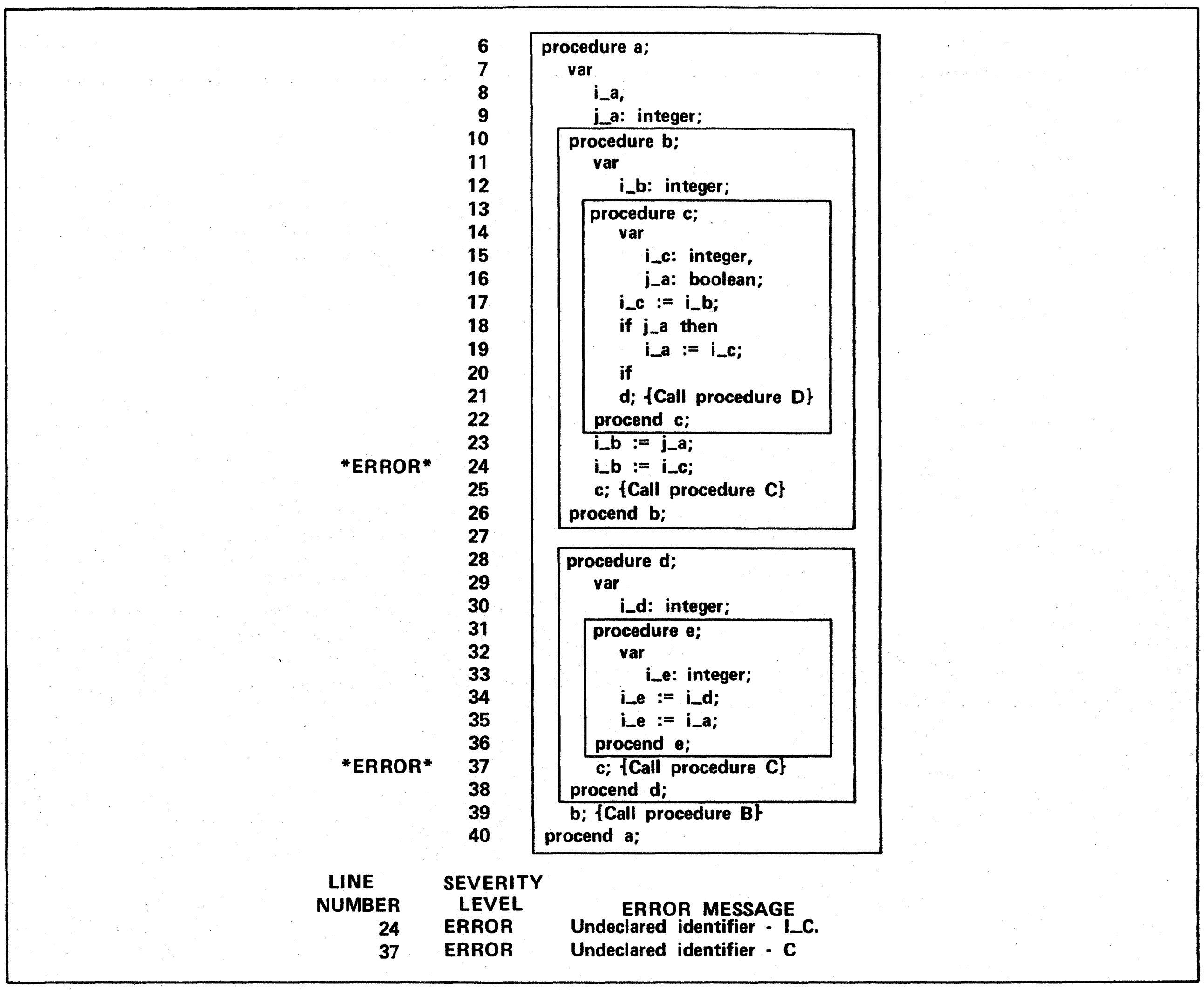
The concept of a point of no return is important, since hardware errors which occur before this point can be retried. If the retry is successful, then a soft error condition is recorded, otherwise a Detected Uncorrectable Error (DUE) is flagged.

CALL/RETURN/POP MECHANISM 7

The CYBER 180 CALL/RETURN mechanism is the technique for crossing protection boundaries within an address space. It is also used for transferring control between procedures (subroutines). It is designed to satisfy the requirements of block structured languages permitting recursive calls such as CYBIL - the implementation language for CYBER 180.

SOFTWARE CONSIDERATIONS

Before describing the CALL/RETURN mechanism, a short introduction to block structured languages is in order. Procedures (subroutines) in a block structured language are organized into a series of nested blocks (figure 7-1). In each set of blocks, variables are related. Variables are classified into two types: static and dynamic. Static variables are allocated to fixed memory addresses and tend to be used throughout a program. Dynamic variables are allocated to different memory address each time a procedure is called. This allocation occurs in a stack. A stack is an area of memory which can grow and shrink dynamically, in accordance with the demands. Each time a procedure is called a new stack frame for that procedure is created. On CYBER 180 much of the management of this stack is accomplished by the hardware of the CALL/RETURN mechanism. The objective is to contain the code for a given function in a compartment for which there are controlled modes of entry. The variables used by this compartment (or block) are generated each time the block is entered and are erased when the block is exited.



```
                 6   procedure a;
                 7     var
                 8       i_a,
                 9       j_a: integer;
                10     procedure b;
                11       var
                12         i_b: integer;
                13       procedure c;
                14         var
                15           i_c: integer,
                16           j_a: boolean;
                17         i_c := i_b;
                18         if j_a then
                19           i_a := i_c;
                20         if
                21         d; {Call procedure D}
                22       procend c;
                23       i_b := j_a;
     *ERROR*    24       i_b := i_c;
                25       c; {Call procedure C}
                26     procend b;
                27
                28     procedure d;
                29       var
                30         i_d: integer;
                31       procedure e;
                32         var
                33           i_e: integer;
                34         i_e := i_d;
                35         i_e := i_a;
                36       procend e;
     *ERROR*    37       c; {Call procedure C}
                38     procend d;
                39     b; {Call procedure B}
                40   procend a;

     LINE       SEVERITY
     NUMBER      LEVEL        ERROR MESSAGE
        24      ERROR      Undeclared identifier - I_C.
        37      ERROR      Undeclared identifier - C
```

Figure 7-1. Example of Block Structure

In the diagram, two sets of nested blocks are shown in module A. These are (B,C) and (D,E). The replacement statements in procedure C involve variables described in the program module A and in the procedures B and C. Knowledge of the whereabouts of these variables is maintained by a static link which is held in the stack frame for each procedure. This linkage is called static since it is known by the compiler at compile time and never changes.

Figure 7-2 illustrates the stack mechanism. The process starts by creating a stack frame for the dynamic variables in the module A. A Current Stack Frame pointer (CSF) points to the beginning of this stack frame and a Dynamic Space Pointer (DSP) points to the next available (free) space in the stack. On CYBER 180, the stack frame and the pointers are established by software. When procedure B is called, procedure A´s environment is saved and a stack frame created for procedure B. A dynamic link is created pointing to procedure A´s stack frame, and a static link pointing (in this case) to the same stack frame. The dynamic link is termed the Previous Save Area pointer (PSA) and is automatically updated on a CALL and RETURN by the CYBER 180 hardware.

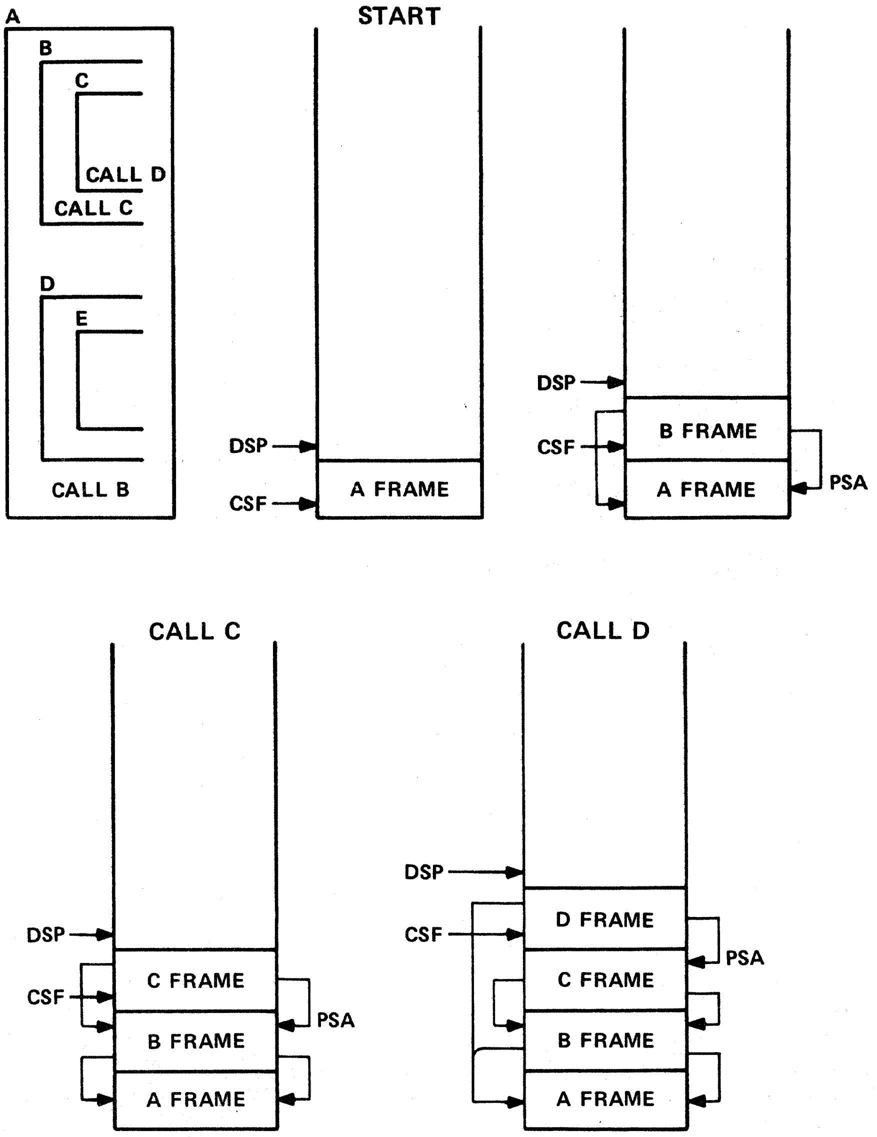


Figure 7-2. Stack Frame Manipulation by Call/Return

A call on procedure C follows in much the same way, and again the static and dynamic links simply point to the previous stack frame. However, when procedure C calls on procedure D, the dynamic link (for stack management) points to the previous stack frame, whereas the static link points to the stack frame for module A, but not to those declared in blocks B and C which are contained within A but do not contain D. The reason for this is that procedure D is a block within the base module, A, and procedure D has access to variables declared in module A, but not to those declared in blocks B or C.

On each procedure call the DSP is updated to point to the next available space within the stack. This is a software function on CYBER 180, and represents the reservation of an area in the stack which is large enough to accommodate all of the dynamic variables for a given procedure - a quantity which is known only to the software.

Since each time a procedure is called, the caller´s environment is saved, it is easy to see that a procedure may be reentered or called recursively. This is true providing all code is organized into pure procedures. That is, no code modification is permitted.

The CALL/RETURN mechanism provides facilities for protection, dynamic linking and virtual machine switching. These features of CALL and RETURN are developed separately because of their importance. First, it is necessary to understand the hardware support for the basic mechanism.

Consider an executing procedure (procedure A) which calls a second procedure, procedure B. Refer to figure 7-3. Four parameters are of interest:

- Top of Stack (TOS) Pointer - in exchange package.
- Dynamic Space Pointer (DSP)- held in A0.
- Current Stack Frame (CSF) - held in A1.
- Previous Stack Frame (PSF) - held in A2.

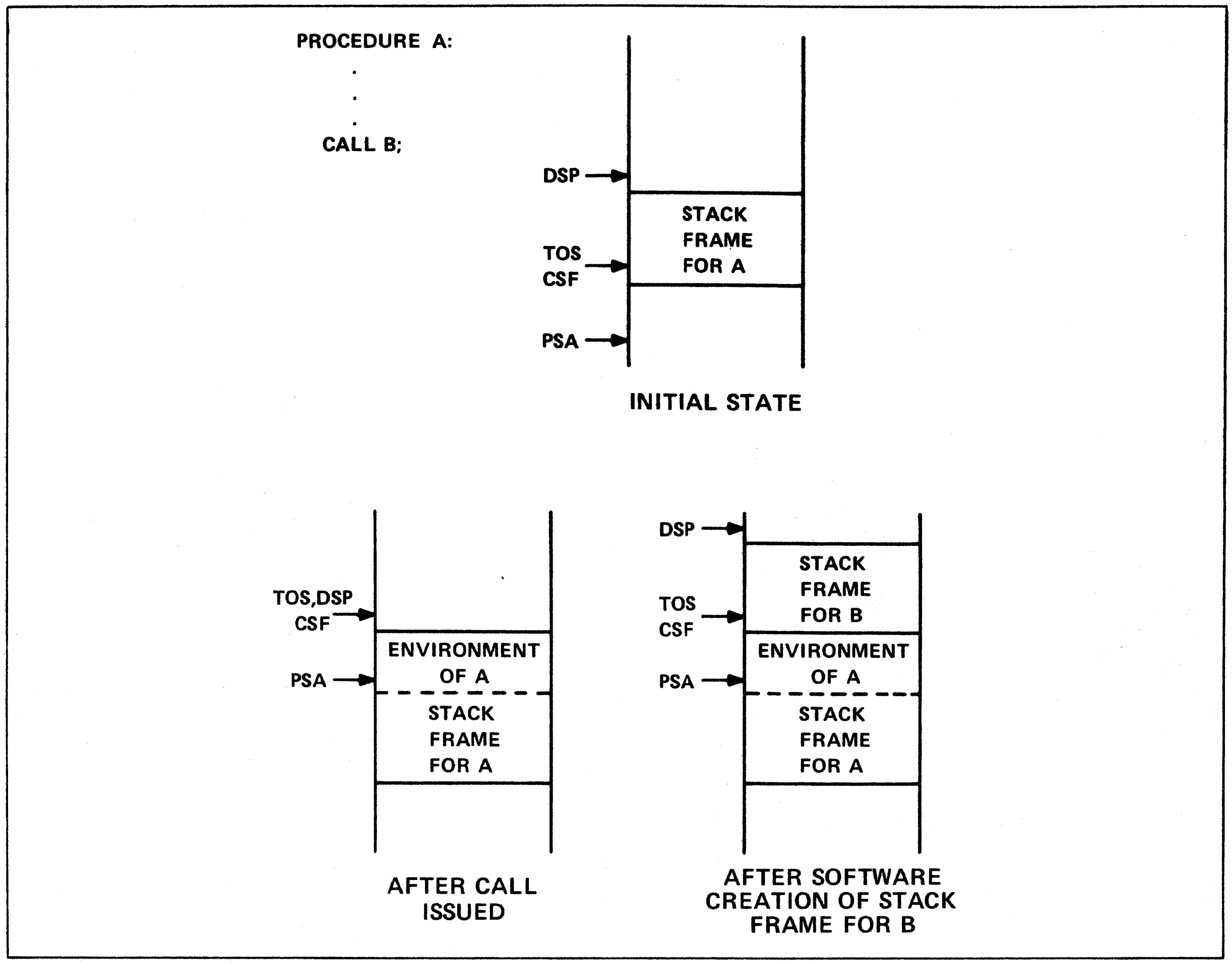


Figure 7-3. Basic Call Mechanism

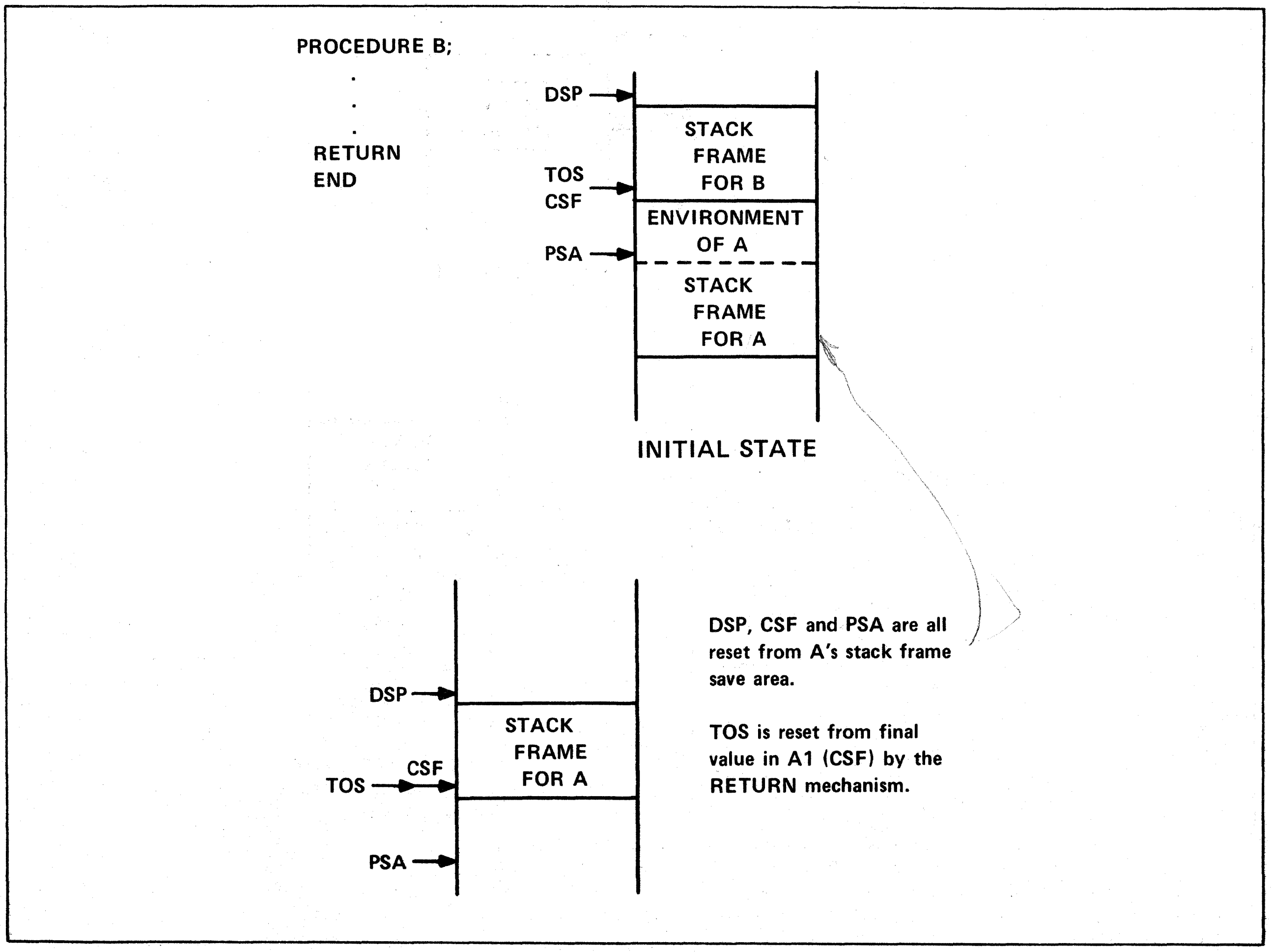
These quantities are pointing within the stack as indicated prior to the CALL. When the CALL is issued, the following steps occur:

1. Caller's environment is saved in caller's stack frame, and TOS is updated to reflect the next free space in the stack.

2. PSA is set to DSP. A2 <- A0

3. CSF and DSP are set to TOS.

The next step is for the software to create a stack frame to hold dynamic variables for procedure B. At this time, the four key parameters are pointing into the stack for procedure B precisely as they had been for procedure A. Return can be accomplished easily since the pointers to A´s stack frame have been saved (in A0-A2) in the stack frame save area (figure 7-4).



DSP, CSF and PSA are all reset from A's stack frame save area.

TOS is reset from final value in A1 (CSF) by the RETURN mechanism.

Figure 7-4. Basic Return Mechanism

CALL - THE BASIC MECHANISM

A stack is created by the operating system for each ring of execution. A TOS pointer for each of these stacks is kept in the exchange package. Whenever a procedure calls another procedure, the caller´s environment is saved in the stack frame save area (figure 7-5). The first four words of this area are stored unconditionally, the remaining words are stored under the control of the caller. The caller formats a Stack Frame Descriptor in X0-Right prior to issuing the CALL. The descriptor specifies which X and A registers are to be saved, in addition to those saved by default. Registers saved must be contiguously numbered. In the case of A registers, since A0-A2 are saved unconditionally, it is only necessary to specify the upper limit of the contiguous list. The descriptor is analagous to that used by load/store multiple instructions, which are described later. It must be supplied, and the terminal A register designator must be greater or equal to two. If no X registers are to be saved then the terminal X register designator (Xt) should be less than the starting X register designator (Xs). When callee returns to caller, these registers are automatically restored. Hence, the operation of the hardware strongly suggests a software calling convention whereby the caller saves the environment.

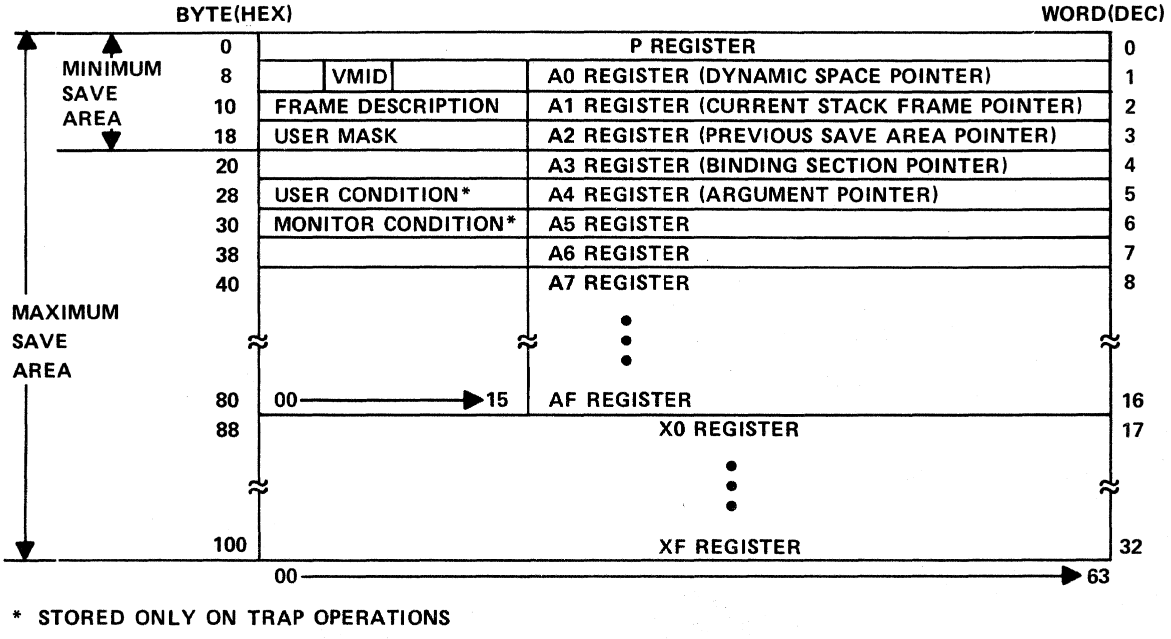


Figure 7-5. Stack Frame Save Area

CYBER 180 supports two forms of the CALL instruction which may be loosely regarded as general purpose (CALL INDIRECT) and special purpose (CALL RELATIVE) calls. The CALL INDIRECT may call into a different segment in a different ring, and as will be shown later, into a different virtual machine. Whereas the CALL RELATIVE calls into the same environment. Although the same basic mechanism applies to both forms of the call, the general purpose version must guarantee the privacy of the callee and caller who may have quite different privileges.

The flowcharts given at the end of this section describe these instructions completely but the following basic steps are followed:

1) Caller's environment is saved
2) Caller's stack frame is pushed
3) The P register is updated to point to the first instruction of the callee to be executed.

There is a single return instruction which simply inverts this process:

1) Callee´s stack frame is popped.
2) Caller´s environment is restored.
3) The P Register is updated (from caller´s environment) so that it points to the first instruction following the original CALL, to be executed.

General Notes:

1) Caller´s environment is saved in caller´s stack. In fact, it is saved at the top of caller´s stack. To minimize the execution time for a CALL this environment is stored on word boundaries. If the top of stack happens not to be on a word boundary, then the stack frame save area will be forced to a word boundary by the CALL instruction.

2) Callee´s stack frame is not created per se. The CSF pointer is updated to point to the first entry in the stack frame, but it is the responsibility of callee, via software, to reserve the appropriate amount of space in the stack. It is also recommended that an integral number of words be reserved for that purpose.

3) Since the CALL RELATIVE calls to a word boundary, every procedure (subroutine) must start on a word boundary. While this is not strictly necessary for external procedures, when the process of binding is described, the reason for this convention will become apparent.

RETURN - THE BASIC MECHANISM

The basic return mechanism, pops callee´s stack frame and restores caller´s stack frame as the active frame. In other words, the environment which exists following the execution of a RETURN instruction is precisely that which existed prior to the execution of the associated CALL instruction. Figure 7-6 illustrates the changes which occur in the stack when this sequence is followed:

```
CALL   (intra-ring)
CALL   (inter-ring) from ring 11 to ring 3)
RETURN
RETURN
```

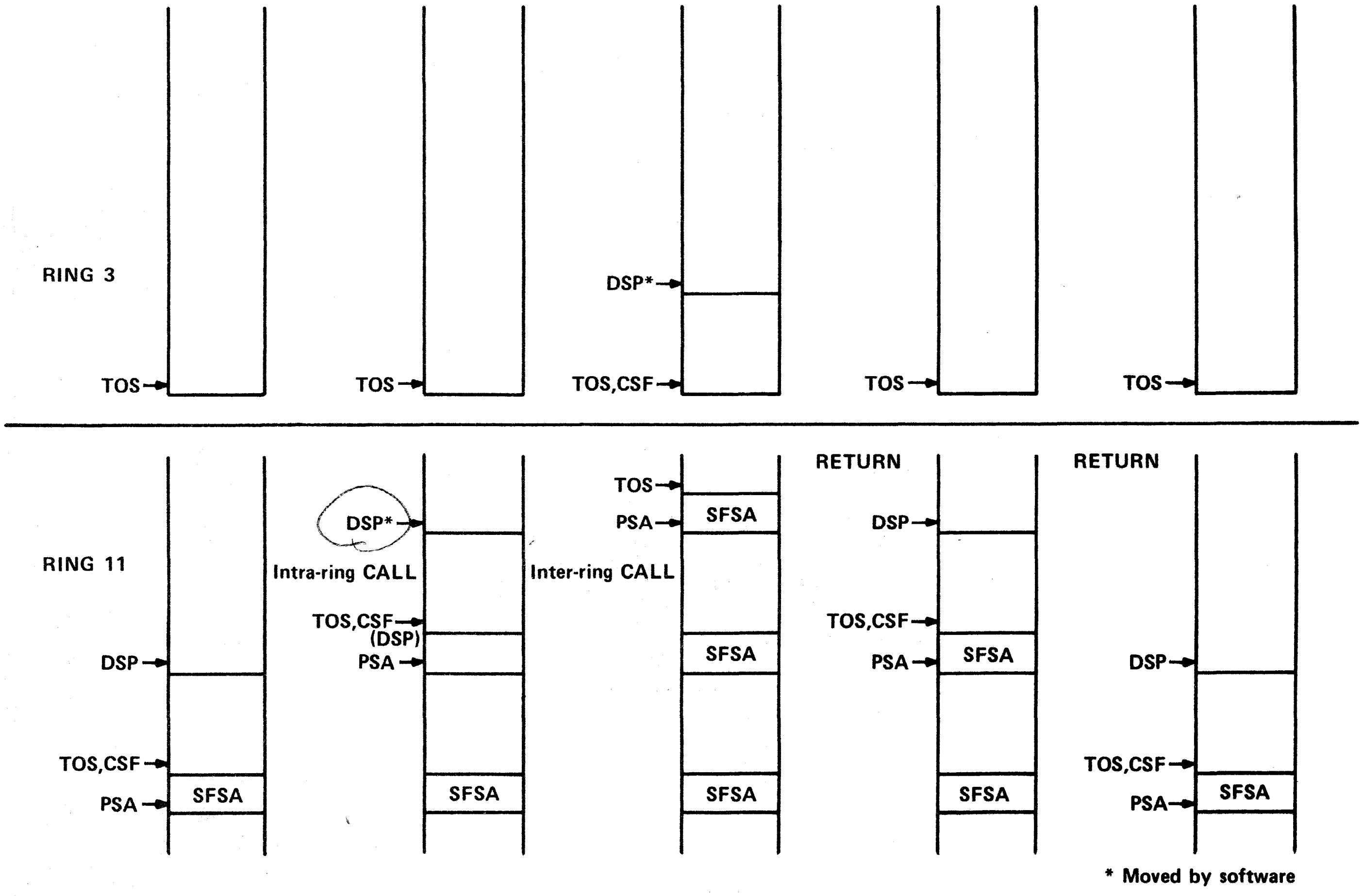


Figure 7-6. Call/Return

Since calls are typically to inner rings, returns are typically to outer, less privileged rings. Care must be exercised to ensure that callee´s greater privileges are not transmitted back to caller. Callee´s ring number may appear in any A Register used by callee - not just those saved by caller. To ensure that this ring number does not get returned to caller, a check is made by the return instruction to ensure that no A Register is returned to caller with a ring number that exceeds caller´s ring number. This process is termed rippling, and figure 7-7 illustrates the process. Caller´s overall privileges, maintained in the P Register are automatically restored when caller´s P Register is loaded from the stack frame save area. A check is made to ensure that the global and local keys which are loaded are identically equal to those to be found in caller´s Segment Descriptor Entry.

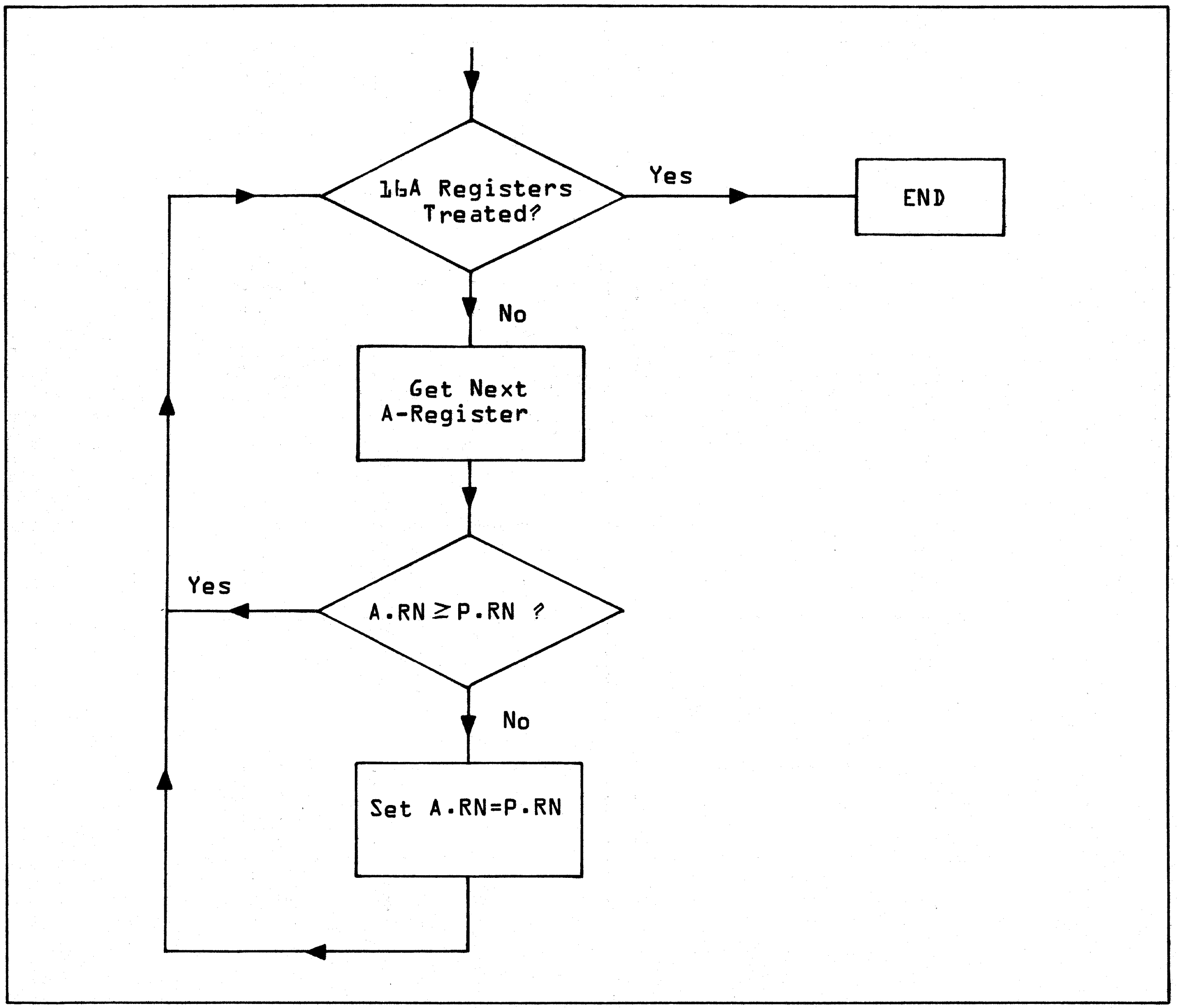


Figure 7-7. Rippling

General Notes:

1) Processes start execution, typically, in their outermost ring. Stacks in all rings will be empty except for the one in the primary ring of execution. As calls are made inward entries are made in other stacks which will be emptied as RETURN´s are issued. The question might reasonably be asked, "Why have fifteen stacks?". Again when the security of the system is considered the reason for this becomes obvious. Since the stack holds the dynamic variables for an executing process, that process has Read/Write access to the stack. If there were only a single stack then an executing process could make a call to a procedure in an inner ring, and then access that procedure´s dynamic variables, which would be at the top of the stack. The only way to prevent this would be for callee to zero out all dynamic variables used. This would be prohibitively time consuming.

2) CALL and RETURN are time consuming operations and are designed to satisfy the general architectural requirements of CYBER 180. In particular, the generalized form of CALL (CALL INDIRECT) should only be used when an external procedure call is made to a procedure in another segment. When binding is discussed it is seen that the Binder actually assists with this task.

The flowcharts at the end of this section describe the overall process for RETURN.

POP - THE BASIC MECHANISM

There are times, typically in the presence of an error, or a nonlocal GOTO, when it is necessary to eliminate an entry or a number of entries from a particular stack. Since these entries will have been created by a series of calls, a similar series of returns will accomplish the required purge. However, when the purging is to be completed without executing intervening instructions this can only be achieved by an appropriate software sequence, or by issuing a POP instruction which has been provided for this purpose. The POP instruction simply moves the CSF, PSA and TOS pointers eliminating the stack frame but not changing the P-counter. Figure 7-8 is an example wherein calls have been made three deep into the structure of a program and then the entire set of calls aborted. POP´s can only be issued within the current ring of execution. Access violations are not checked, and if a POP is attempted across rings (as indicated by the ring number in A2-PSA), then the instruction execution is inhibited and the program interrupted.

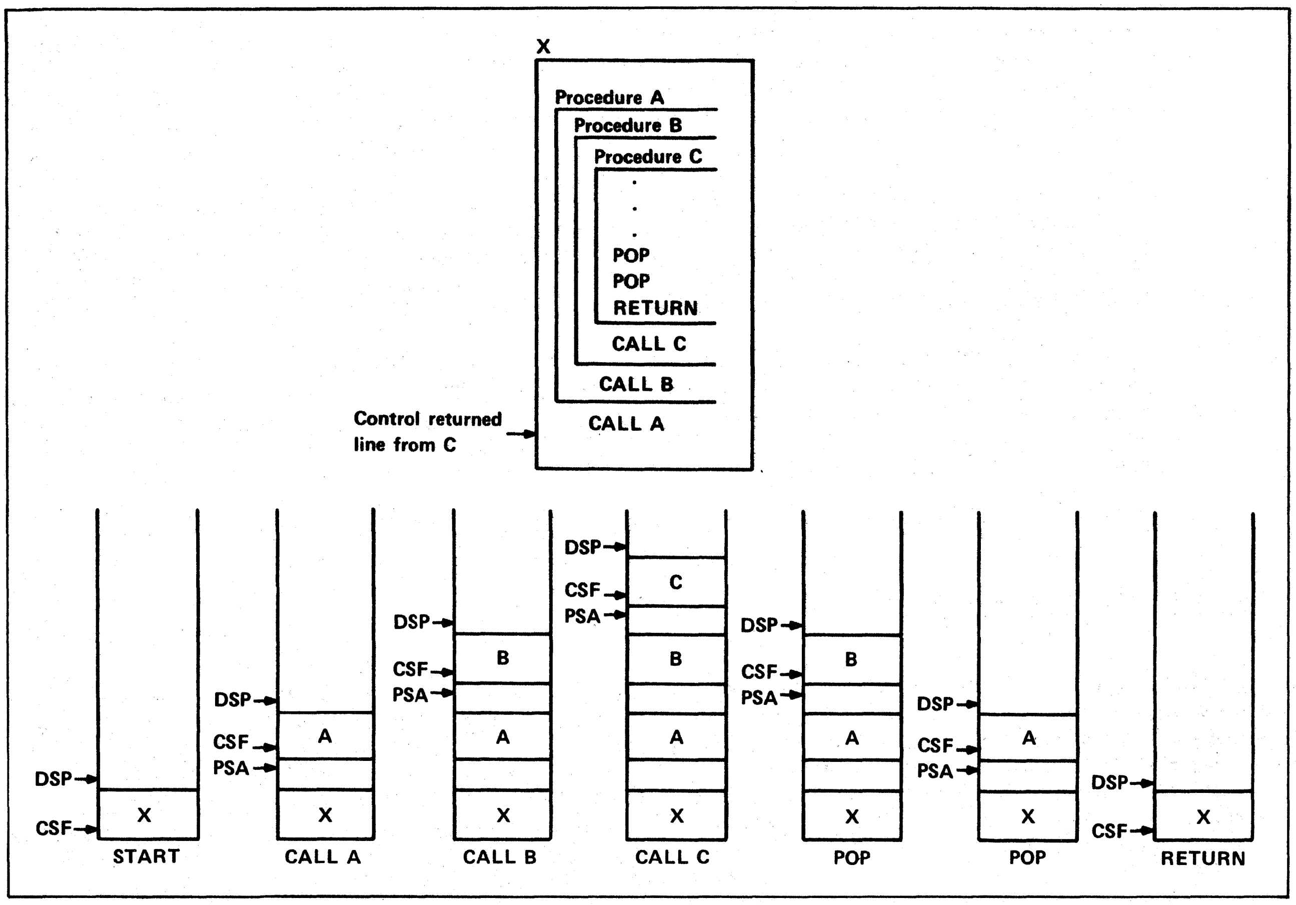


Figure 7-8. Example of POP Instruction

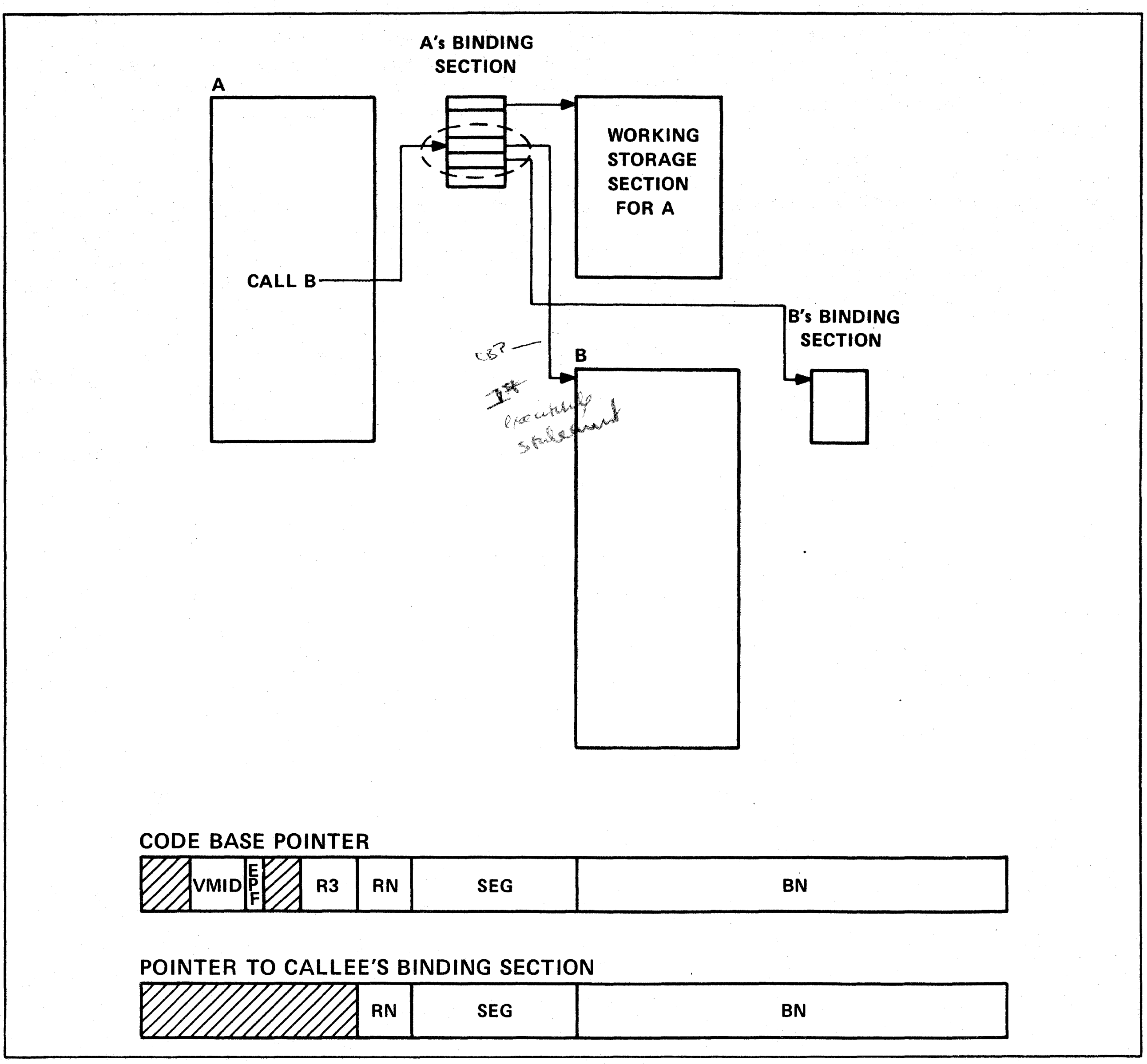
THE BINDING SECTION - CODE SHARING

It is important that the entry to procedures be carefully controlled. That is, procedures must receive control only at those points they expect to receive control - their entry points. To make this possible, procedures are not entered directly, but are entered via a pointer to the procedure. This pointer is held in a Binding Section. All such pointers are placed in the Binding Section by the Loader, and the CALL mechanism then guarantees that the call is made via a Binding Section.

The objective on CYBER 180 is to have one copy of a code segment in memory which is shared by several users. Each user has a copy of each code sequence required in his virtual memory address space. For example, the FORTRAN compiler exists only one time in real memory, but depending on which user has the CPU it operates on different compilation units. There must be nothing in the code segment which makes a direct reference to data which is modified. This is accomplished by placing pointers to such data in a Binding Section which is created along with each code module, and then give the address of the Binding Section to the callee when the procedure call is invoked.

Each task executing then, has some code, which it may be sharing with other tasks, and some data which is typically unique to itself. When a compiler compiles some source code, it compiles offsets into the Binding Section and directives to the loader for building the Binding Section. It is then the responsibility of the loader to link all code modules and build the necessary Binding Sections. This process is described more fully in the section on software.

The Binding Section is in a separate segment and is identified uniquely by its segment descriptor entry (refer to section on virtual memory). It typically contains pointers to external procedures and pointers to working storage areas which hold static variables. That is, variables which do not appear in a stack frame. When a procedure calls on another procedure which is defined externally, then the call points into the Binding Section. The Binding Section (by convention) has one, or two, full word entries which contain a Code Base Pointer (CBP) and a pointer to the callee's Binding Section (figure 7-9). The CBP points to the first executable statement in procedure D. The VMID and R3 fields in the CBP are discussed shortly. The EPF (external procedure flag) field in the one state indicates that the procedure being called is an external procedure, and therefore the next entry in the Binding Section is the pointer to callee's Binding Section. This field is nothing more than a flag to differentiate between single-word and double-word entries in the Binding Section.



CODE BASE POINTER

▨	VMID	EPF	▨	R3	RN	SEG	BN

POINTER TO CALLEE'S BINDING SECTION

▨	RN	SEG	BN

Figure 7-9. Call Indirect Example

Whenever a call is made to a procedure in another ring, this form of the call instruction (via the Binding Section) must be used. However, for critical (intrasegment, intraring) calls a shorter form of the call instruction should be used. This form finds the first executable statement of the caller at P plus an offset and obviates the need for a CBP. The offset used by this instruction is a 16-bit long word offset, hence all procedures must start on a word boundary.

In order to make code sharing possible, each task sharing the code must have its own data. The location of this data is defined through the Binding Section, which, in turn, is defined via the Process Segment Table. The Segment Table Address is defined by the Exchange Package for that process. Hence, each instance of a process has an exchange package which describes the process state registers for that exchange interval. This defines the whereabouts of the code and data to be used by the process, via the virtual memory mechanism. Different tasks using the same code will have their own Segment Tables and will have unique entries in the System Page Table for their Binding Sections and data. However, the page table entry for the code segment will be shared by all tasks sharing the code (figure 7-10). The way this happens is quite simple if one remembers the basic virtual memory address translation mechanism.

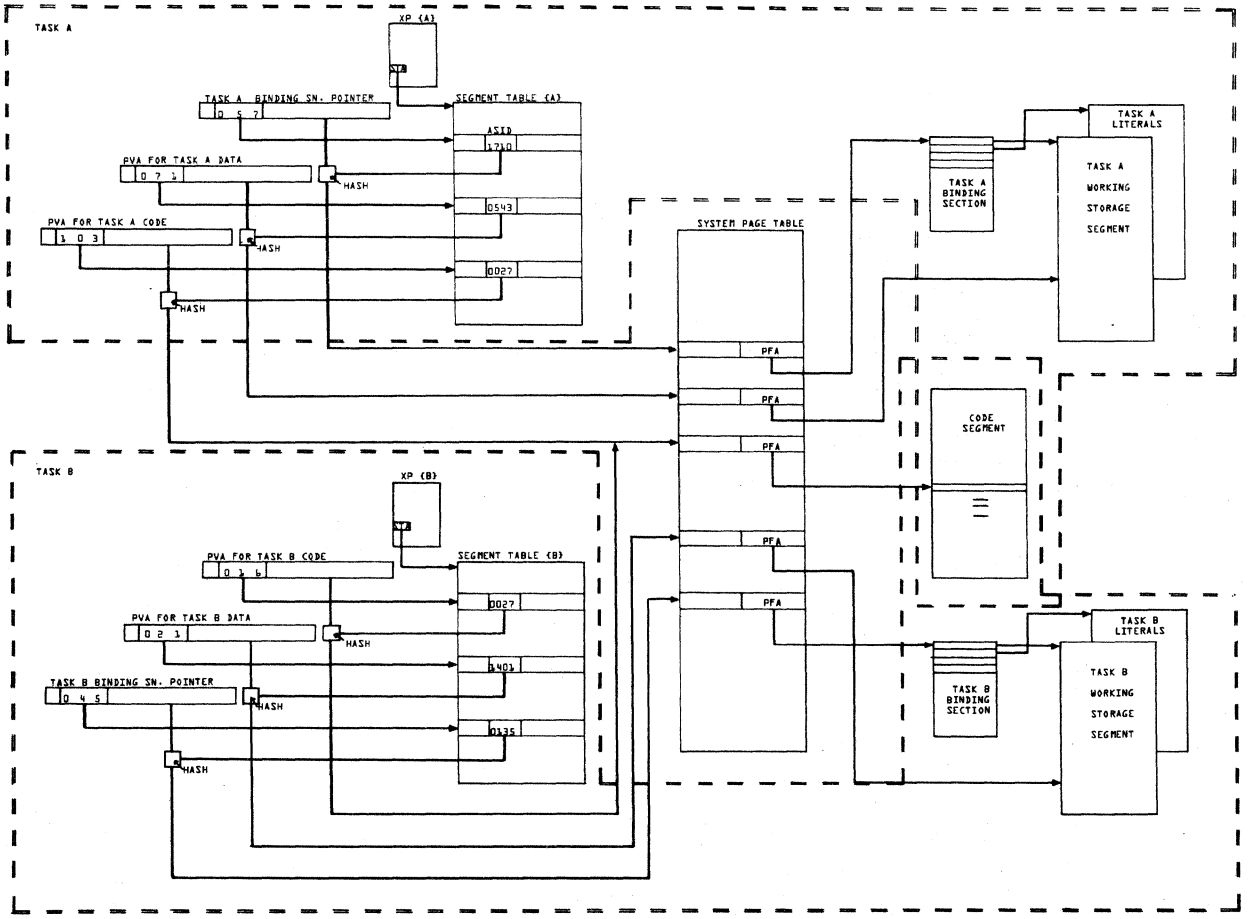


Figure 7-10. Code Sharing

The key to code sharing lies with concept of the SVA. Code which is being shared actually resides as two conceptually separate copies in two address spaces. When this code is referenced (via a PVA) the first step is to translate the PVA into an SVA. The operating system arranges for code which is to be shared to have common SVAs. The translation to a real memory address will then result in the same locations in central memory regardless of the process requesting the translation. This is accomplished by assigning common ASID´s to shared code segments, which happens the first time the segment is referenced. Neither the originator of the code being shared, nor any users of it need be aware that the code is being shared. The only contingency is that code be organized into pure procedures. Code sharing per se, is unrelated to CALL/RETURN. However, the separation of code and data, the absence of direct references to the data in the code, and the Binding section all play their part. When a procedure is called from another procedure, caller gives callee´s Binding Section to him. That is, caller carries a pointer to callee´s Binding Section as a parameter of the call. It is by this mechanism that shared code (that is, code shared in real memory) receives different data sets on which to work. The whereabouts of these Binding Sections is determined by the Loader which loads multiple copies of the code which will ultimately be shared, into virtual memory (figure 7-11).

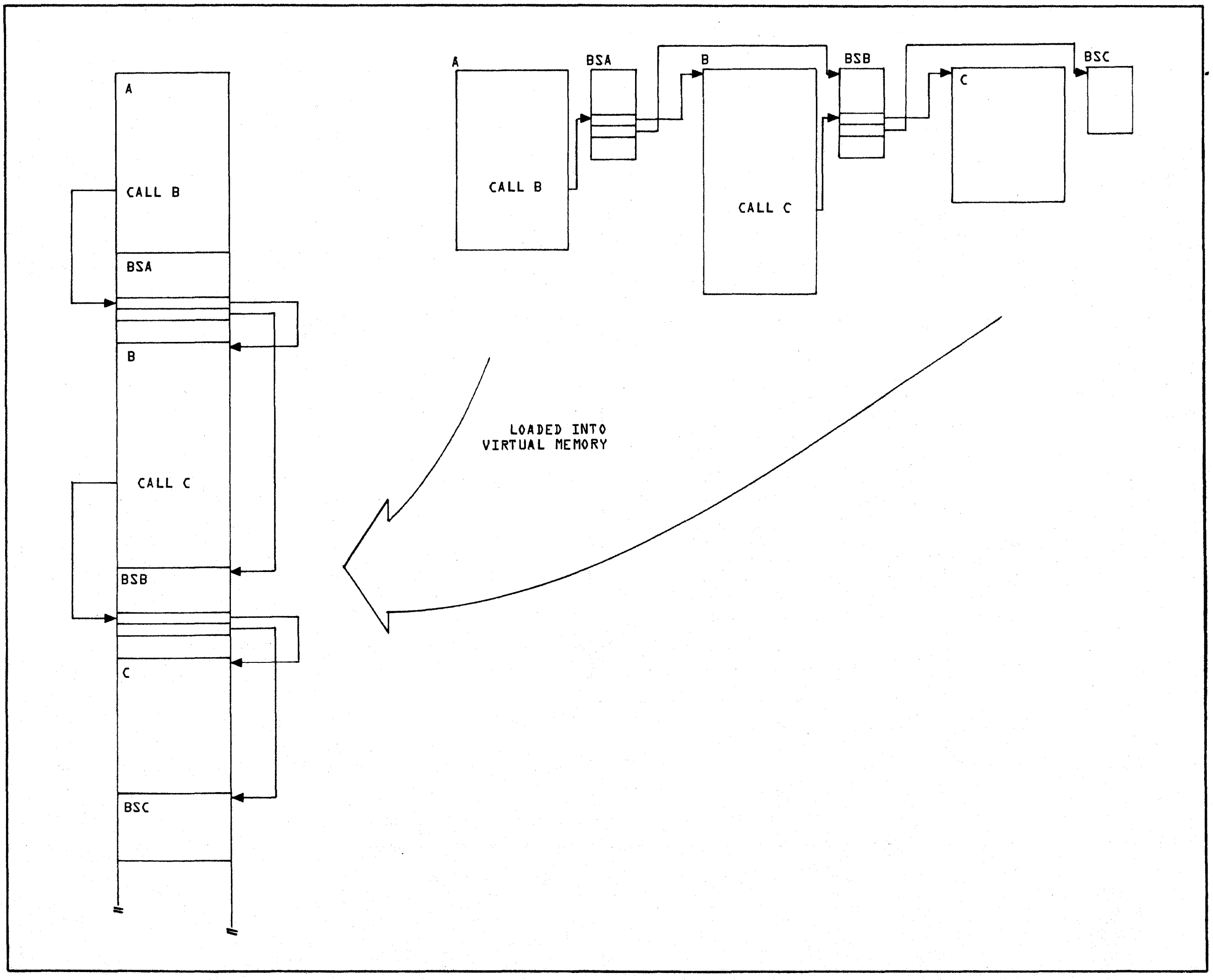


Figure 7-11. Loading Mechanism

Two further areas, software conventions and parameter passing, need to be discussed before the basic mechanism can be summarized. Parameter passing is discussed first. In general, when a procedure is called from another procedure, parameters need to be passed between them. The general parameter passing technique selected for CYBER 180 is to pass an argument list pointer to the callee. Typically, this argument list pointer points to a list of pointers which in turn point to data to be referenced by the called procedure. By convention, the argument pointer is held in A4 and the convention is supported by the hardware which transfers the argument list pointer to A4 during the execution of a CALL instruction.

Two other pointers are used by procedures, namely the Binding Section pointer and the static link. Of these, the Binding Section pointer is by far the more important and by convention is held in A3. As with the argument list pointer, this software convention is supported by the hardware. The choice of registers A3 and A4 to hold these quantities simplifies the saving and restoring of them during CALL´s and RETURN´s since the instructions always saves a contiguous set of A Registers, and A0-A2 are always saved by a CALL. The static link is not always required, and for those cases where it is needed it is carried by software and the hardware has no part in its maintenance.

FLAGS

There are two flags which are handled by the CALL/RETURN/POP instructions. These are the On-Condition Flag (OCF) and the Critical Frame Flag (CFF). They are software flags which are reset by the hardware on each call to a new procedure.

ON-CONDITION FLAG

The end-user causes the OCF to be set by requesting that a particular code sequence be executed when a chosen error arises. This is generally done via a high level language, and the compiler generates the code necessary to set the OCF and generate a dummy stack frame for the On-Condition processing (figure 7-12). A pointer in the user´s stack frame points to this dummy. All exception conditions are typically selected by the process monitor. When one arises a trap interrupt occurs and the trap handler searches the stack for the presence of an OCF which is set. On-Conditions are set by a particular procedure. When a CALL is made, the OCF associated with the calling procedure is saved (in the stack frame save area) as part of caller´s environment, and the OCF is cleared. If an appropriate exception arises, then it will be handled by caller´s On-Condition action, unless callee had also requested specific action to be taken on the same exception. The following should be remembered:

- Actions to be taken on exceptions are specified by the user. They are recorded in a dummy stack frame, and by setting the OCF.
- Actions are established by a given procedure but carry across procedure calls.
- Each procedure may have its own unique set of On-Conditions.

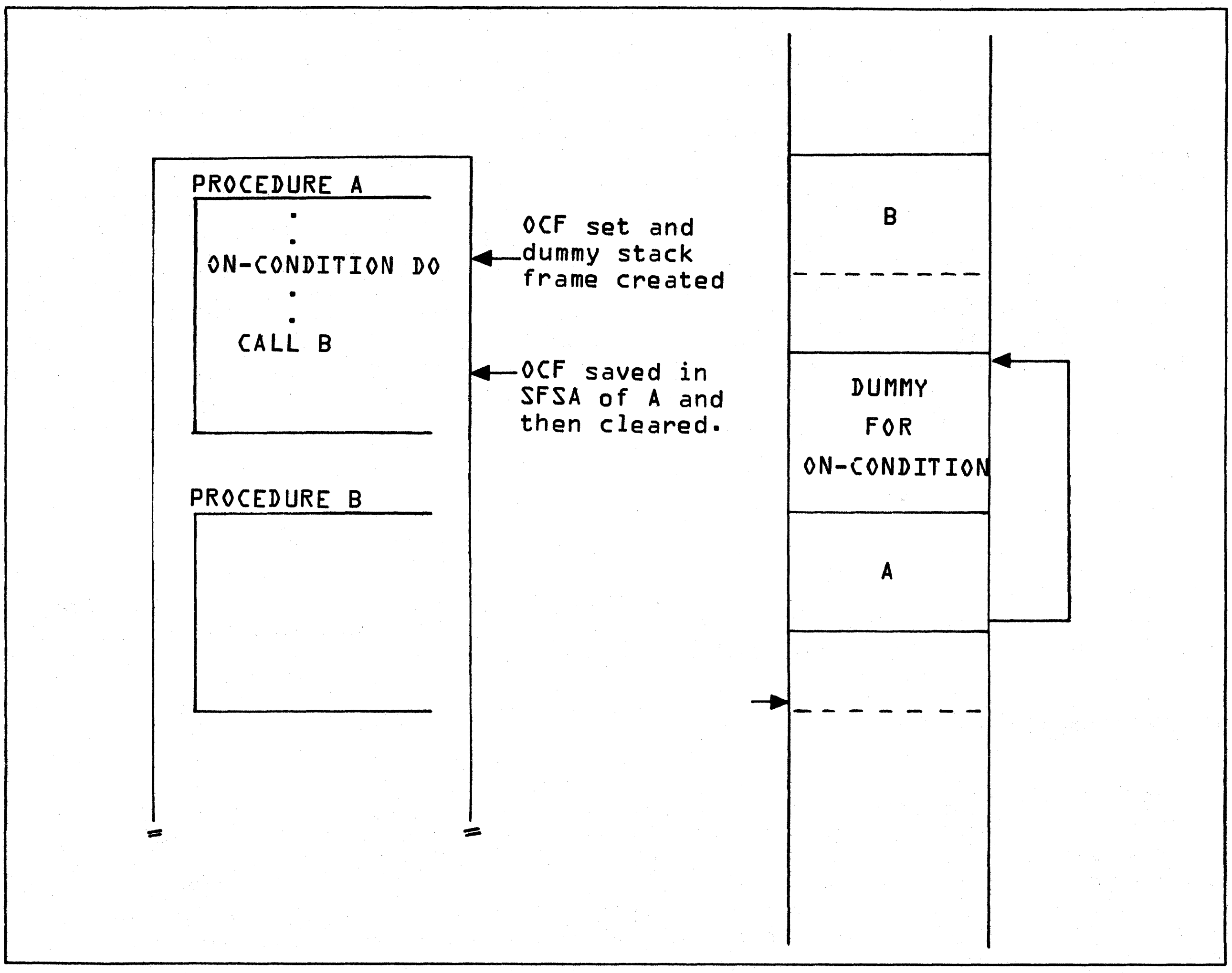

PROCEDURE A
ON-CONDITION DO
CALL B
PROCEDURE B
OCF set and dummy stack frame created
OCF saved in SFSA of A and then cleared.
B
DUMMY FOR ON-CONDITION
A

Figure 7-12. On-Condition Handling

CRITICAL FRAME FLAG

The critical frame flag is a software device for declaring a procedure critical. The term critical is used to denote the fact that some tidy-up is required before leaving the procedure is question. In other words exit from the procedure must take place in an orderly manner. An example will help to clarify this:

> Imagine a job running under the control of a subsystem. The job may open a file or set some locks which must be closed or cleared before the job is terminated. If the job terminates abnormally the standard tidy-up procedure would pop the stack frames in use prior to returning to the subsystem for final exit. However, in the case where particular action is required before a stack frame is eliminated, a different path must be followed. The critical frame flag is used to alert the subsystem in control of this situation. When the locks are set or files opened, the critical frame flag is also set, and subsequently saved, in the procedures stack frame save area, whenever it calls on

another procedure. An attempt to pop a stack frame with the critical frame flag set is detected by the hardware and a trap interrupt taken. The trap handler hands control back to the subsystem which (by an investigation of user´s stack) can perform the necessary tidy-up operations.

OUTWARD-CALLS/INWARD RETURNS

Calls may only be made within the same ring of protection or to an inner ring (figure 7-13). However, there are various circumstances which require the execution of a call to an outer ring. For example, when an end-user job is initiated a call must be made to the outer ring where the user program resides. Since the hardware prohibits the execution of this call it must be accomplished by the software.

When an outward call is attempted, an interrupt occurs and the following steps are taken (assuming the machine is in job mode, traps enabled):

1) An exchange interrupt occurs (outward-call).

2) The exchange interrupt handler sets the free-flag and issues an exchange. This will cause control to return to the original outward call instruction. However, before it can be reissued:

3) A trap interrupt occurs (free-flag). This is really an implicit call into the stack in caller´s ring of execution. The free-flag is nothing more than a mechanism for converting an exchange interrupt into a trap interrupt.

4) The trap handler creates two dummy frames in callee´s stack. The first dummy frame is callee´s eventual stack frame, the second is created simply to be popped via a return which will transfer control to callee.

5) The trap handler executes a RETURN to callee. This pops the second dummy stack frame (figure 7-13).

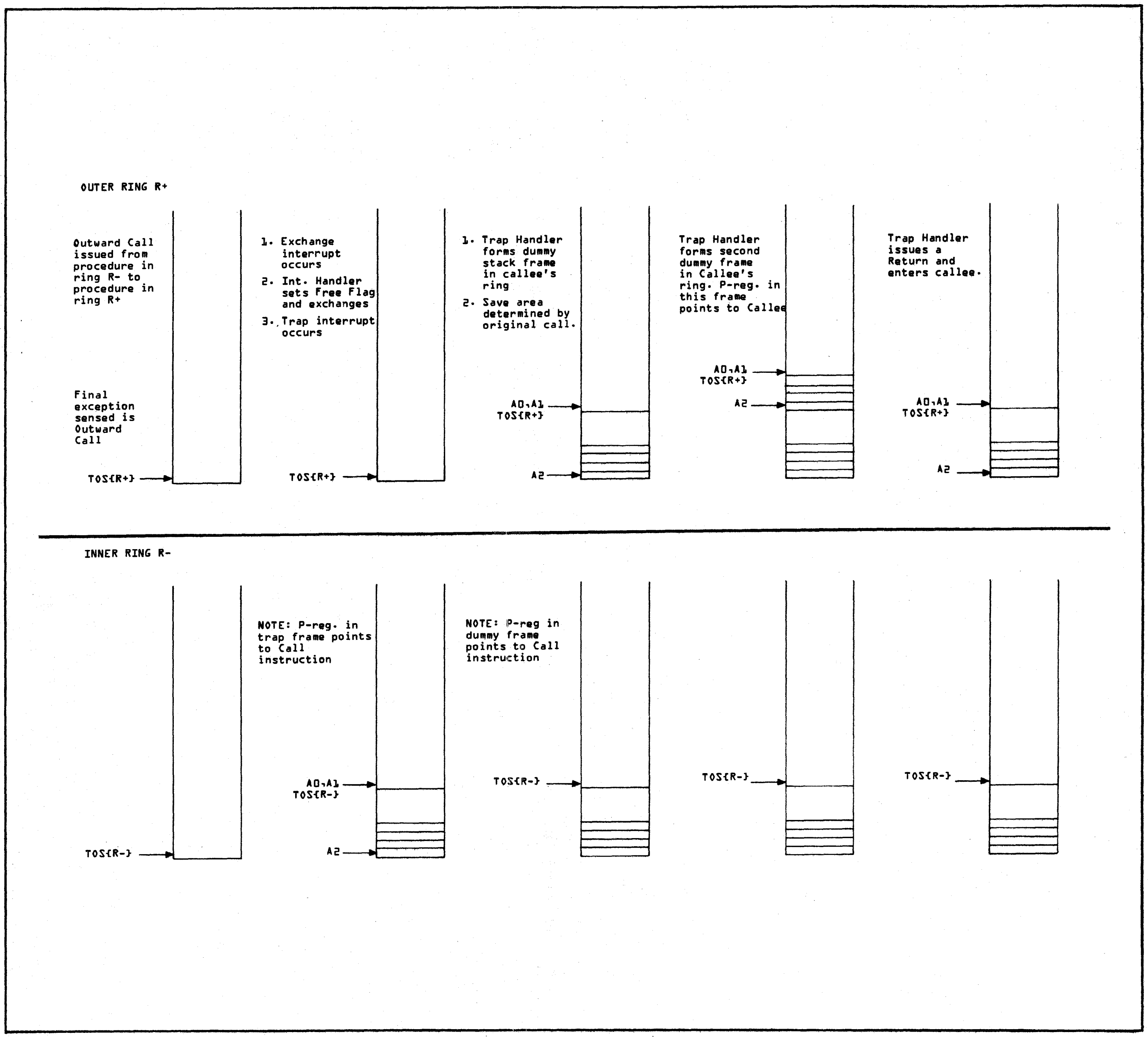

OUTER RING R+
Outward Call issued from procedure in ring R- to procedure in ring R+
Final exception sensed is Outward Call
TOS(R+)
1. Exchange interrupt occurs
2. Int. Handler sets Free Flag and exchanges
3. Trap interrupt occurs
TOS(R+)
1. Trap Handler forms dummy stack frame in callee's ring
2. Save area determined by original call.
A0,A1
TOS(R+)
A2
Trap Handler forms second dummy frame in Callee's ring. P-reg. in this frame points to Callee
A0,A1
TOS(R+)
A2
Trap Handler issues a Return and enters callee.
A0,A1
TOS(R+)
A2
INNER RING R-
TOS(R-)
NOTE: P-reg. in trap frame points to Call instruction
A0,A1
TOS(R-)
A2
NOTE: P-reg in dummy frame points to Call instruction
TOS(R-)
TOS(R-)
TOS(R-)

Figure 7-13. Outward Call

Hence, to execute an outward call, the interrupt handler software arranges to an outward return to callee. An inward return is the reverse of this procedure and consists of executing an inward call to the original caller as follows:

1) An exchange interrupt occurs (inward call).

2) The exchange interrupt handler sets the free-flag and issues an exchange. This causes control to return to the original inward return instruction. However, before it can be reissued:

3) A trap interrupt occurs (free-flag). This is really an implied call into the stack in callee´s ring of execution.

4) The trap handler calls on Task Monitor in caller´s ring of execution.

   Task Monitor:

5) Eliminates three frames from callee´s stack.

6) Adjusts its own stack frame to point to caller´s stack frame

7) Issues a return.

On the inward return the implicit call to replace the return is actually an inward call to Task Monitor (figure 7-14).

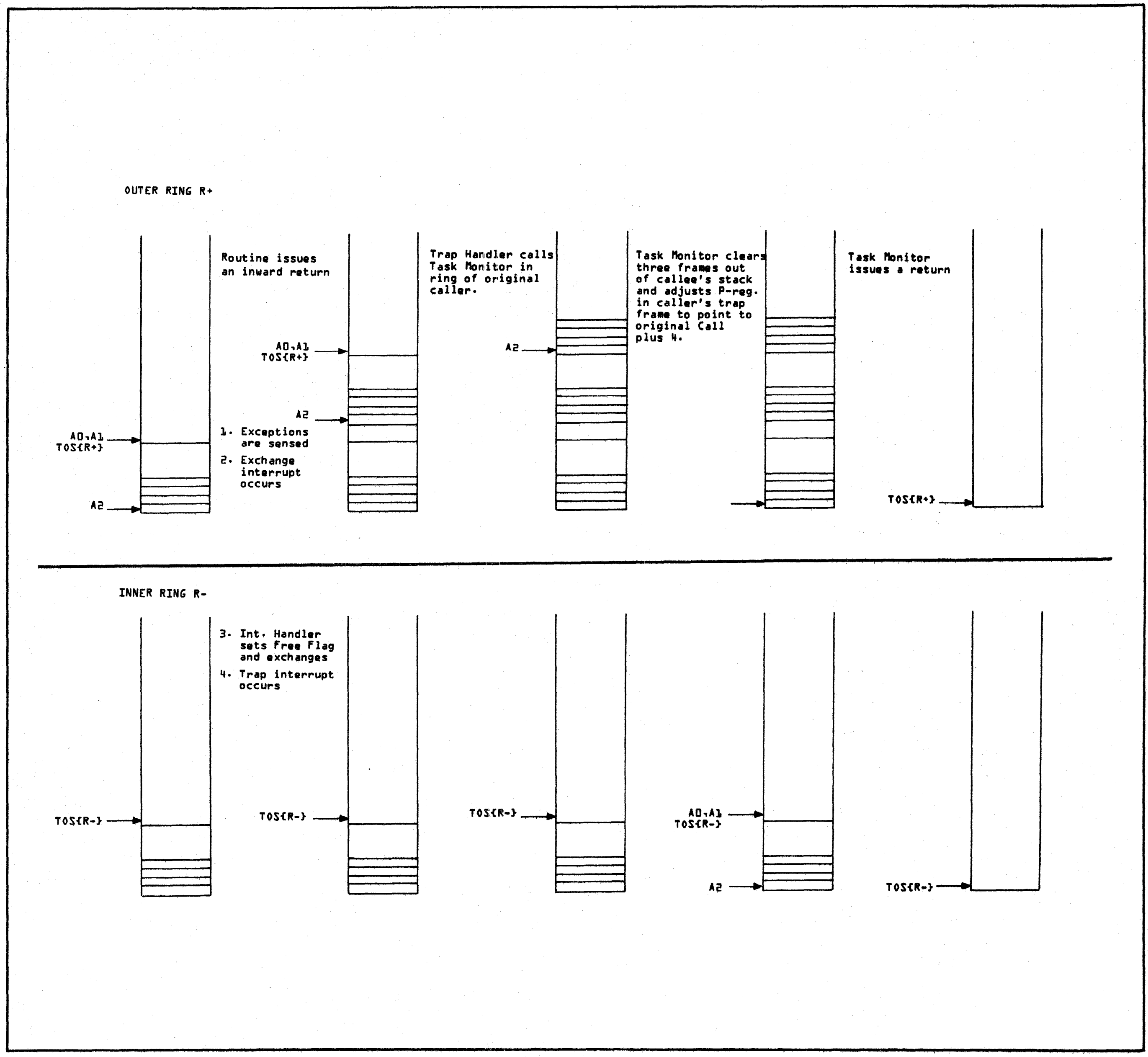
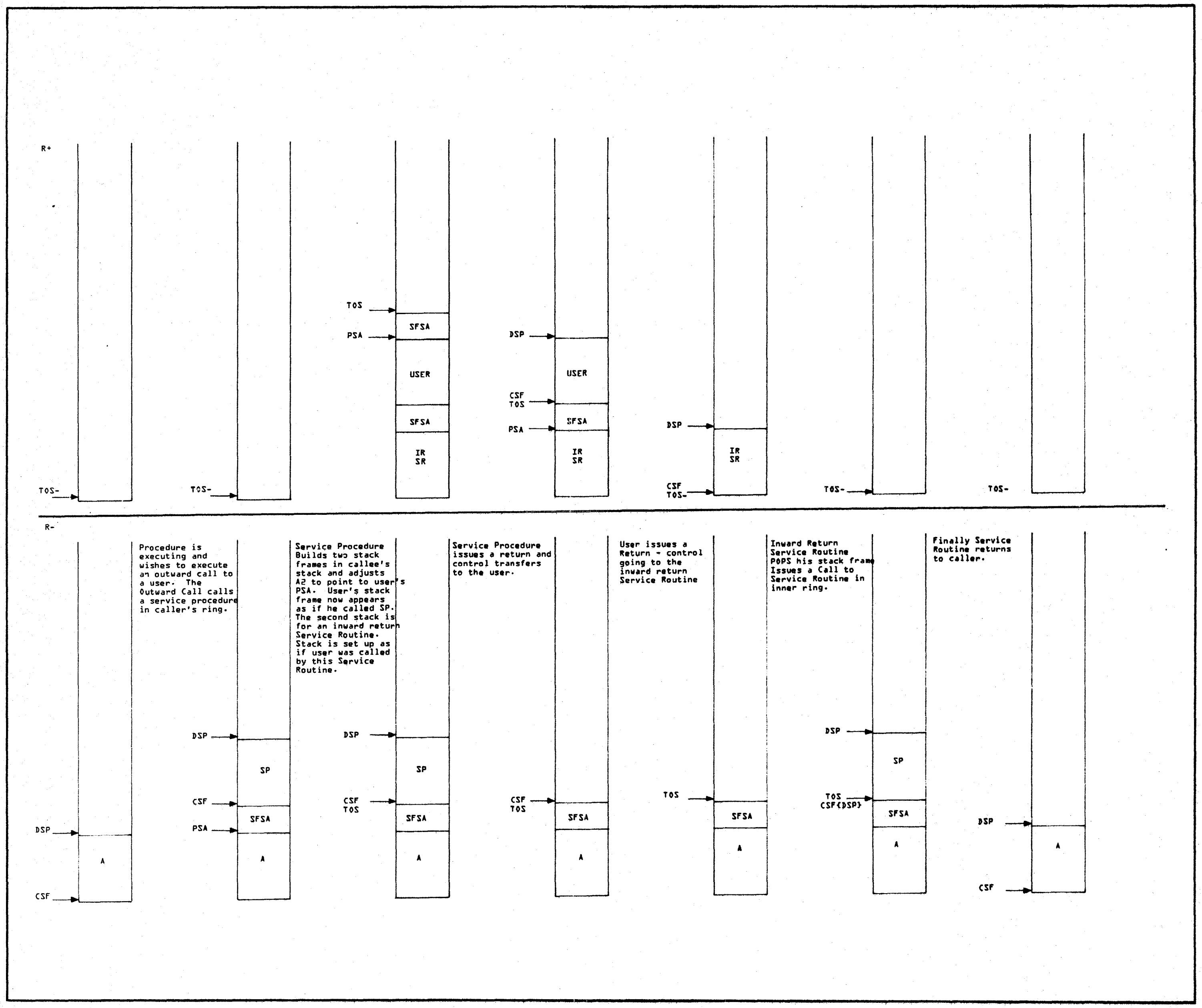

OUTER RING R+
Routine issues an inward return
AO,A1 TOS(R+)
A2
1. Exceptions are sensed
2. Exchange interrupt occurs
Trap Handler calls Task Monitor in ring of original caller.
AO,A1 TOS(R+)
A2
A2
Task Monitor clears three frames out of callee's stack and adjusts P-reg. in caller's trap frame to point to original Call plus 4.
Task Monitor issues a return
TOS(R+)
INNER RING R-
3. Int. Handler sets Free Flag and exchanges
4. Trap interrupt occurs
TOS(R-)
TOS(R-)
TOS(R-)
AO,A1 TOS(R-)
A2
TOS(R-)

Figure 7-14. Inward Return

The general case of an outward call and inward return has been treated here. It can be a time consuming process, and it should be used sparingly. Fortunately, the operating system generally knows ahead of time that an outward call is to be issued and need not bother to take the exchange jump and force a trap interrupt. Instead, the functions performed by the trap handler can be performed by the caller before issuing an outward call, and an outward return can be issued directly to transfer control to callee. Similarly, when an outward call is made, it is known that there will be a subsequent inward return. The functions performed by Task Monitor can be performed by caller prior to issuing the outward call. This can be accomplished by caller calling on an outward call Service Procedure which creates two stack frames in callee´s stack such that it appears as though callee was called by a Service Procedure in his own ring of execution, and called the original (outward call) Service Procedure. The outward call Service Procedure then returns to callee to transfer control. Callee subsequently returns to an inward return Service Procedure in his own ring of execution, which pops its own stack frame before making an inward call on the original outward call Service Procedure. This procedure then returns to the original caller (figure 7-15).



Procedure is executing and wishes to execute an outward call to a user. The Outward Call calls a service procedure in caller's ring.

Service Procedure Builds two stack frames in callee's stack and adjusts A2 to point to user's PSA. User's stack frame now appears as if he called SP. The second stack is for an inward return Service Routine. Stack is set up as if user was called by this Service Routine.

Service Procedure issues a return and control transfers to the user.

User issues a Return - control going to the inward return Service Routine

Inward Return Service Routine POPS his stack frame Issues a Call to Service Routine in inner ring.

Finally Service Routine returns to caller.

Figure 7-15. OS Call

OBJECT MODULE BINDING

Whenever a procedure is compiled or assembled, directives are compiled with it to enable the loader to create a Binding Section. In fact, all references to Working Storage, external procedures, and so forth, are compiled as offsets into the Binding Section. Hence, when a program is executed, there will be multiple Binding Sections (one per procedure). There is nothing wrong with this except that procedures which are called from several other procedures will have an entry in several Binding Sections. This is wasteful in terms of space. Also, many calls to external procedures will translate into calls within the same segment (intrasegment calls). These are really only, external procedure calls at compile time. They become internal procedures at execute time. The difference is that an external procedure call must be made with a CALL INDIRECT via the Binding Section, whereas an internal procedure call can be made with the more efficient CALL RELATIVE instruction. The Object Library Generator minimizes these space and time inefficiencies.

The Object Library Generator performs two major functions:

1) It eliminates redundancy by taking all procedures in a module to be bound and placing them in a single code section. It also combines all Binding Sections into a single Binding Section and eliminates redundant entries to external procedures.

2) Since many calls to external procedures will translate to calls to internal procedures during the coalescing of the Binding Sections, the CALL INDIRECT instructions are converted to CALL RELATIVE instructions for these procedures, and their entries eliminated completely from the Binding Section (figure 7-16).

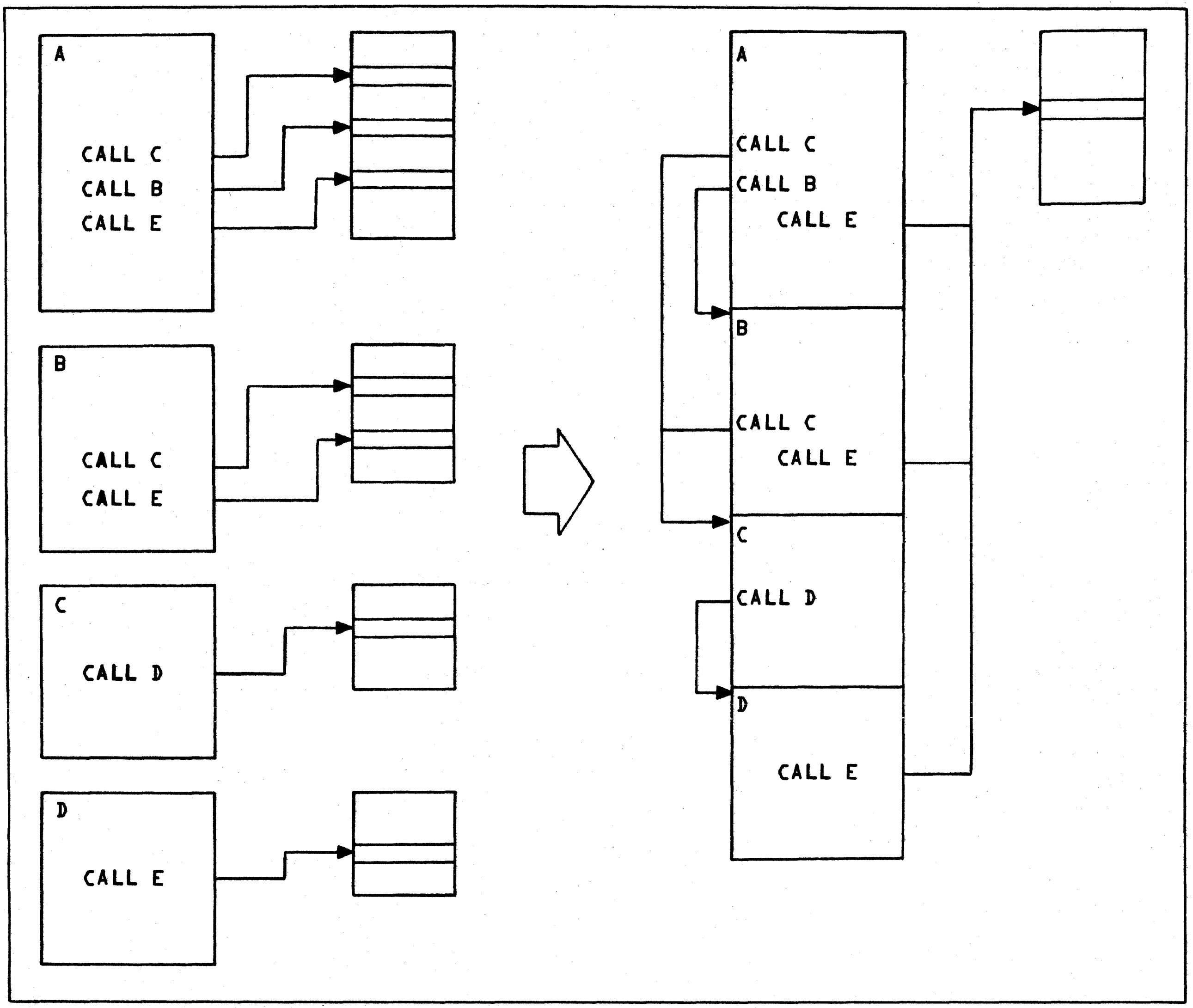


Figure 7-16. Binding Process

This second major function of the Object Library Generator imposes a restriction on the format of the call instructions which have been designed with this purpose in mind. Since they have similar formats, all that need be done by the Object Library Generator is to change the operation code from B5 to B0, set the desired value in the Q-field and force the j-field to three, which is the conventional register for the Binding Section (figure 7-17).

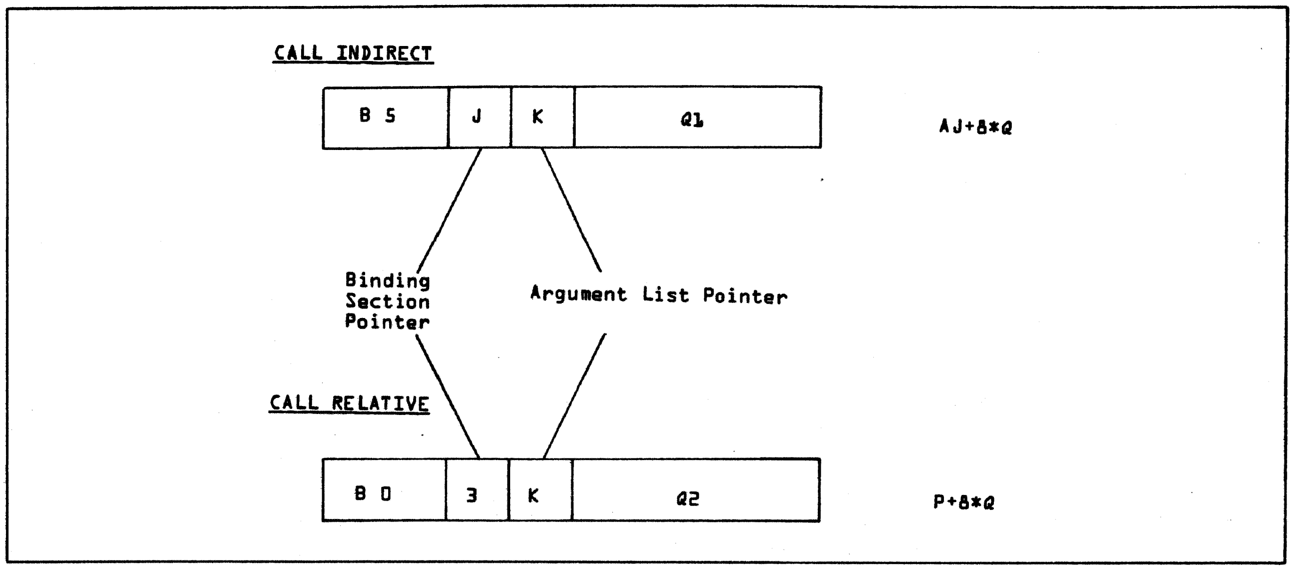


Figure 7-17. Conversion from Call Indirect to Call Relative

VIRTUAL MACHINES

CYBER 180 provides a capability to support several virtual machines. The two most important of these are the native machine (CYBER 180) and CYBER 170. The call mechanism permits a procedure being executed on one virtual machine to call another procedure which will execute with a different virtual machine. The exact mechanism which accomplishes this machine switch will not be described here but will be covered in a separate section on virtual machines.

ZERO RING NUMBER

In the section on virtual memory it was explained that there are fifteen rings of protection on CYBER 180. These are numbered 1-15. Ring number zero has been reserved for a special purpose, namely: Dynamic Linking. Traditionally, a program has been written as a series of subroutines or procedures. These subroutines are compiled separately, then linked together with a loader prior to their execution. Depending on the system, all subroutines referenced had to be present before the program could be placed in execution. Frequently, this restriction is levied even though all subroutines may not be used. This happens to be one way of solving the problem of linking, loading and placing a program into execution, but it is by no means the only one. Certainly, this alternative is open to CYBER 180 and may be a common method invoked by the user. However, CYBER 180 provides another option which is to link and load a procedure the first time it is called and not before. If a procedure is referenced but never called, it need never go through the linking and loading mechanism, and will never require that memory be allocated to it. This process is known as Dynamic Linking, and a ring number of zero is reserved to denote an unlinked pointer.

Ring numbers of zero can occur in one of two ways: in an attempt to load a pointer into an A register, and in an attempt to call on an unlinked procedure. The CYBER 180 hardware automatically detects this condition and causes an exchange interrupt to be taken if the machine is in job mode. The exchange interrupt handler can then schedule the appropriate operating system procedure to form the necessary link and load the required procedure.

When a ring number zero is detected on a load instruction the following sequence occurs:

1) The load instruction completes, loading the invalid pointer into the appropriate A register with a ring number determined by the normal ring number contention mechanisms. Refer to figure 7-18.

2) An exchange interrupt occurs. The P Register stored in the exchange package (at JPS) points to the instruction following the load instruction.

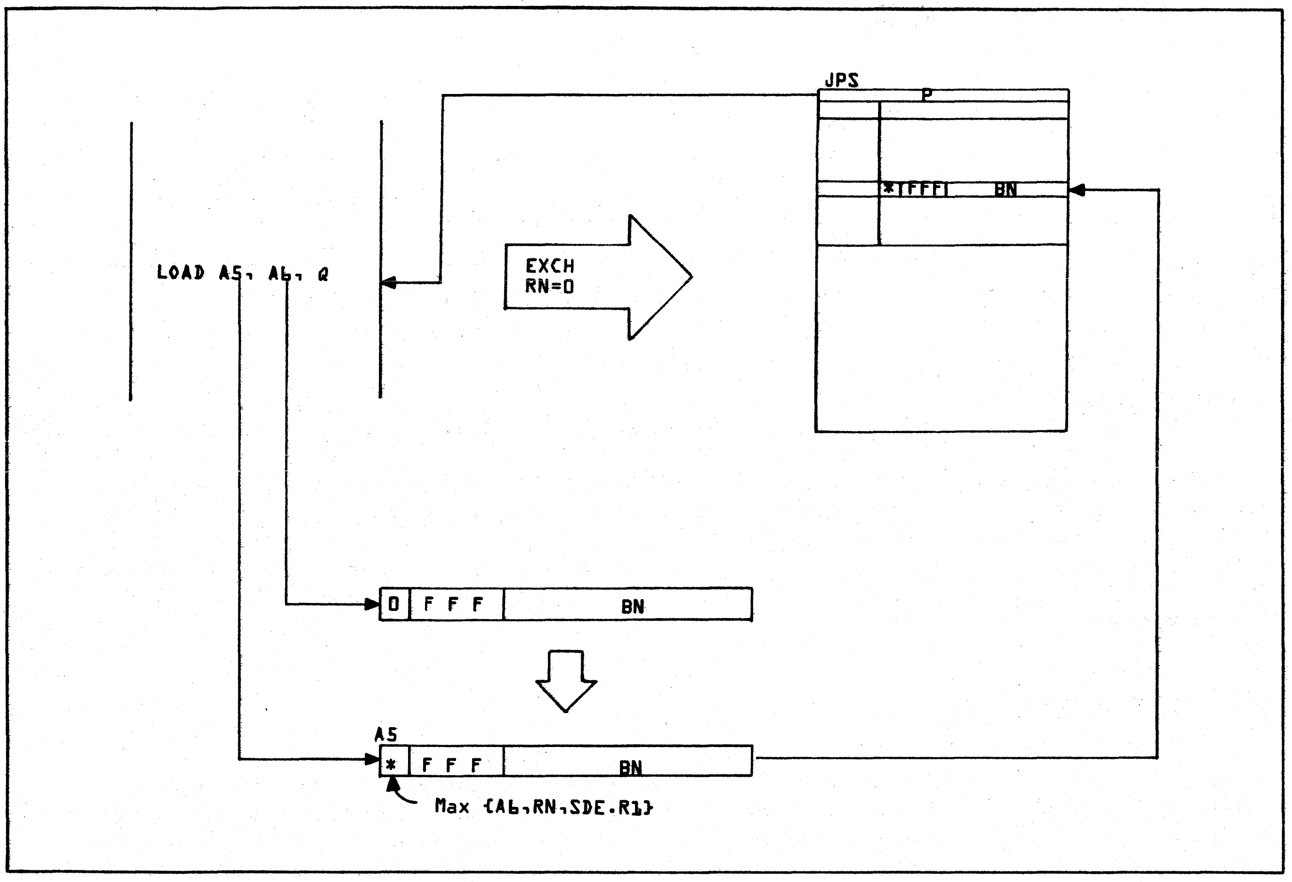


Figure 7-18. Ring Number Zero on Load A

When a ring number zero is detected on a CALL instruction the following sequence occurs:

1) The execution of the call instruction is inhibited.

2) An exchange interrupt is taken. The P Register stored in the exchange package (at JPS) points to the CALL instruction iL

M the necessary link an question, and the Untranslatable Pointer Register stored in the same exchange package, contains the CBP (with a ring number of zero) from the Binding Section which caused the interrupt (figure 7-19).

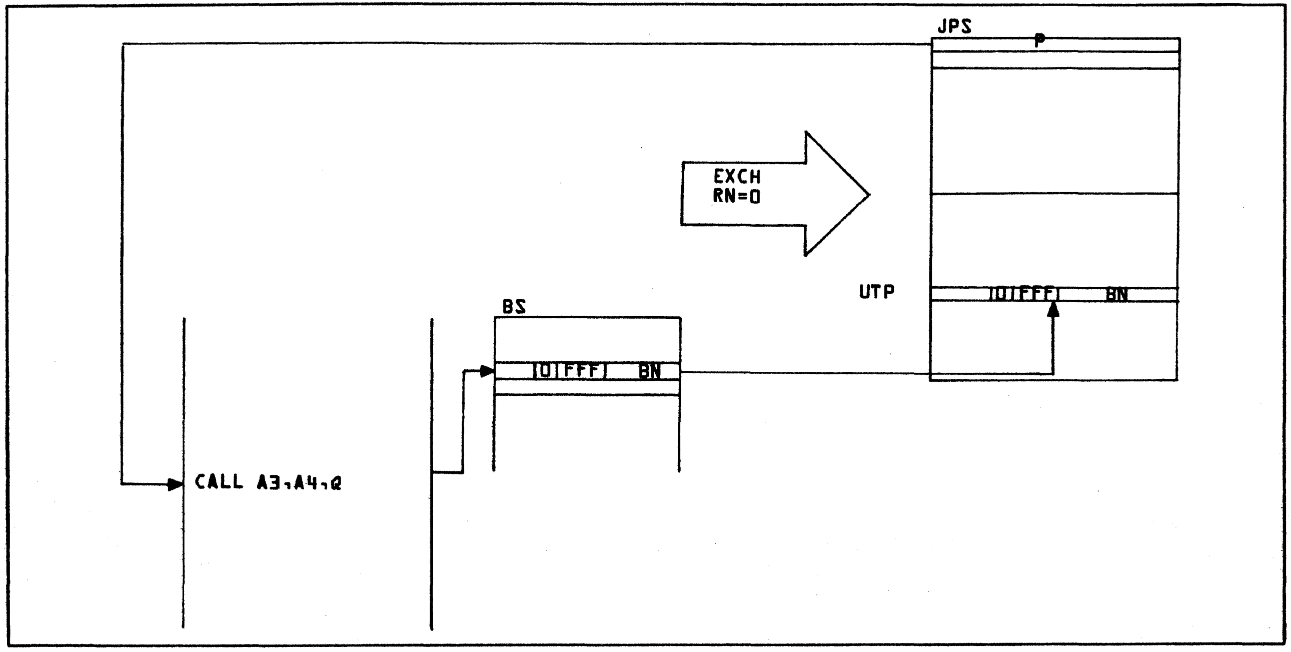


Figure 7-19. Ring Number Zero on Call

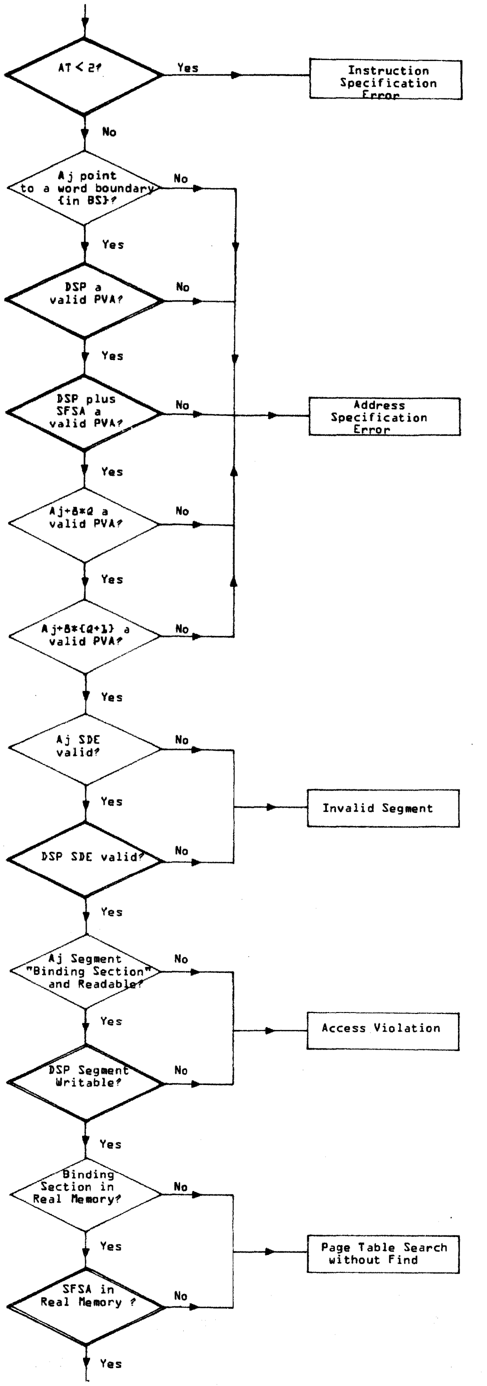
The Untranslatable Pointer (UTP) is the key to handling this exception condition which is combined with an Invalid Segment exception. The sequence the interrupt handler should follow on sensing an Invalid Segment condition is to check on the Untranslatable Pointer Register. If this has a zero ring number plus a segment number of all ones, then a ring number zero condition has been detected by a CALL instruction (as opposed to an Invalid Segment). The Untranslatable Pointer, by software convention contains a dummy Segment Number of all ones (to flag an unlinked pointer) and the Byte Offset contains a pointer to loader tables which contain information necessary to form the required link. The appropriate entry is made in the Binding Section containing the unlinked Code Base Pointer and an exchange jump executed, which will cause the CALL to be reissued.

If the UTP contained no indication of the fault (this must be established by software convention) then the individual A Registers (at JPS) must be scanned for the fake segment number. The zero ring number will no longer exist in the register or registers in question, since it will be eliminated by the ring number voting mechanism. The register or registers in question are loaded with the correct segment number and byte offset and an exchange jump issued to continue processing. Remember to scan all A Registers since several of them could have zero ring numbers if a Load Multiple instruction is used.

Dynamic linking is an option provided by the CYBER 180 hardware. It provides an alternative to conventional loading techniques. However, there is no need to support this particular technique by software. That is an operating system design decision. Nevertheless, without hardware support of this nature the choice would not be available.

FLOWCHARTS OF THE OVERALL PROCESS

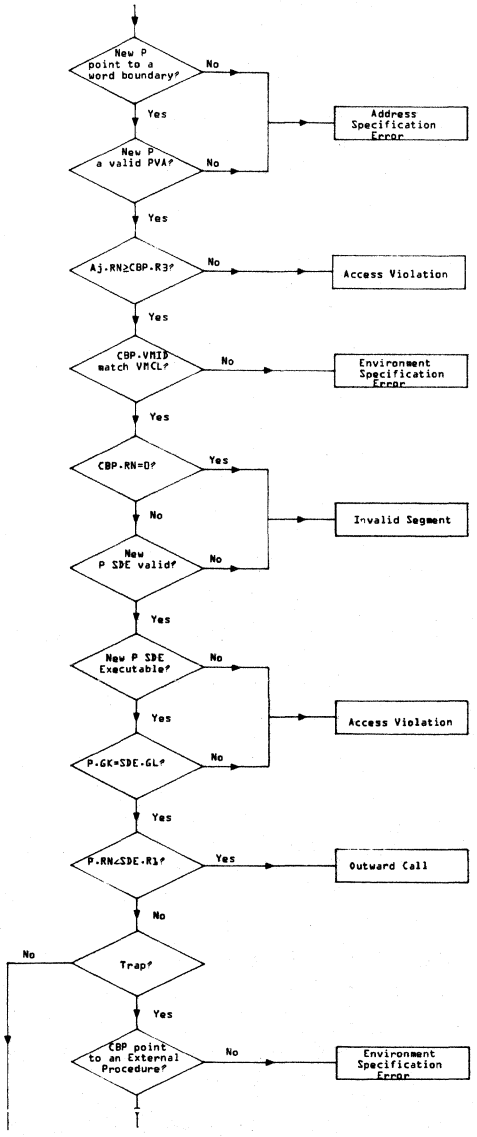
Figures 7-20, 7-21, and 7-22 describe the overall process for CALL, RETURN and POP. Included in the flow-chart for CALL are those steps which are unique to a trap interrupt. Remember a trap interrupt is nothing more than an unsolicited call in which all the A registers and X registers are saved. There are some additional steps. In particular the condition causing the trap is erased from either the User Condition Register or the Monitor Condition Register and those registers are captured in the Stack Frame Save Area.



CALL/TRAP

NOTE: Double edged boxes apply to CALL DIRECT, CALL RELATIVE and TRAP.
Single edged boxes apply to CALL DIRECT and TRAP.

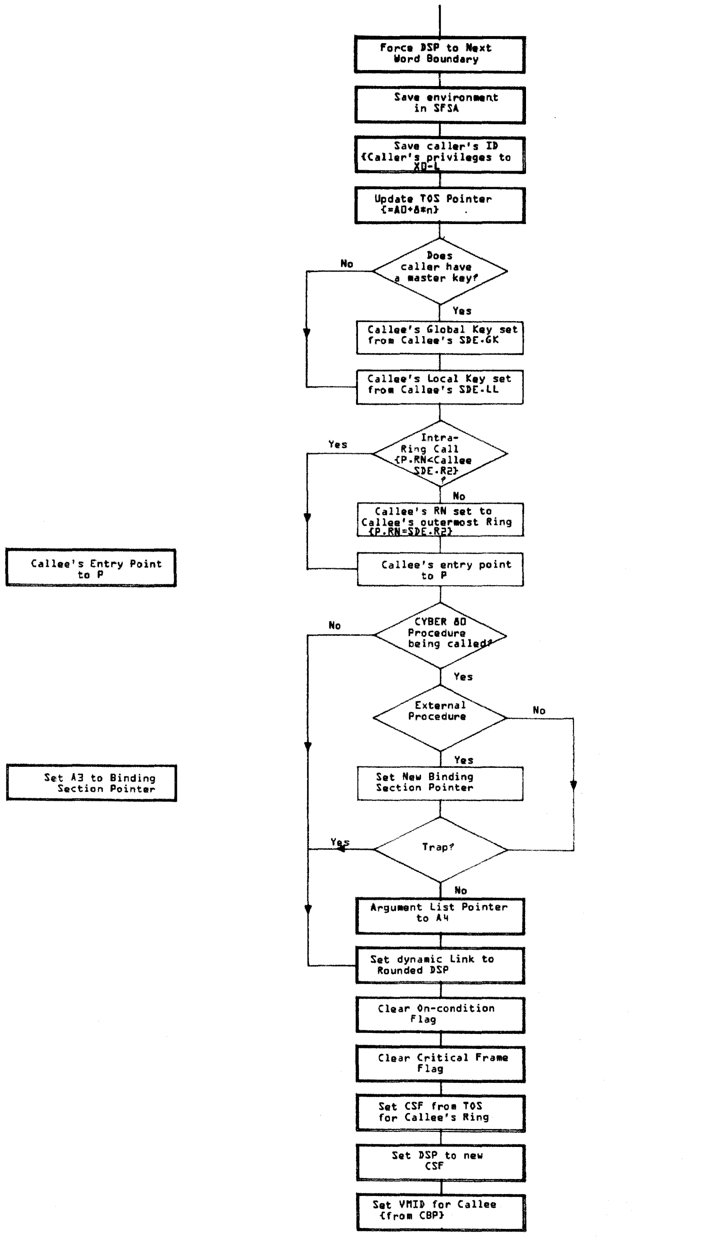
Figure 7-20. CALL/TRAP (Sheet 1 of 3)



CALL/TRAP

NOTE: Double edged boxes apply to CALL DIRECT, CALL RELATIVE and TRAP.
Single edged boxes apply to CALL DIRECT and TRAP.

Figure 7-20. CALL/TRAP (Sheet 2 of 3)


Force DSP to Next Word Boundary
Save environment in SFSA
Save caller's ID {Caller's privileges to XD-L
Update TOS Pointer {=AD+8*n}
Does caller have a master key?
No
Yes
Callee's Global Key set from Callee's SDE.GK
Callee's Local Key set from Callee's SDE.LL
Intra-Ring Call {P.RN<Callee SDE.R2} ?
Yes
No
Callee's RN set to Callee's outermost Ring {P.RN=SDE.R2}
Callee's Entry Point to P
Callee's entry point to P
CYBER 80 Procedure being called?
No
Yes
External Procedure
No
Yes
Set A3 to Binding Section Pointer
Set New Binding Section Pointer
Trap?
Yes
No
Argument List Pointer to A4
Set dynamic Link to Rounded DSP
Clear On-condition Flag
Clear Critical Frame Flag
Set CSF from TOS for Callee's Ring
Set DSP to new CSF
Set VMID for Callee {from CBP}

NOTE: Double edged boxes apply to CALL DIRECT, CALL RELATIVE and TRAP.
Single edged boxes apply to CALL DIRECT and TRAP.

Figure 7-20. CALL/TRAP (Sheet 3 of 3)

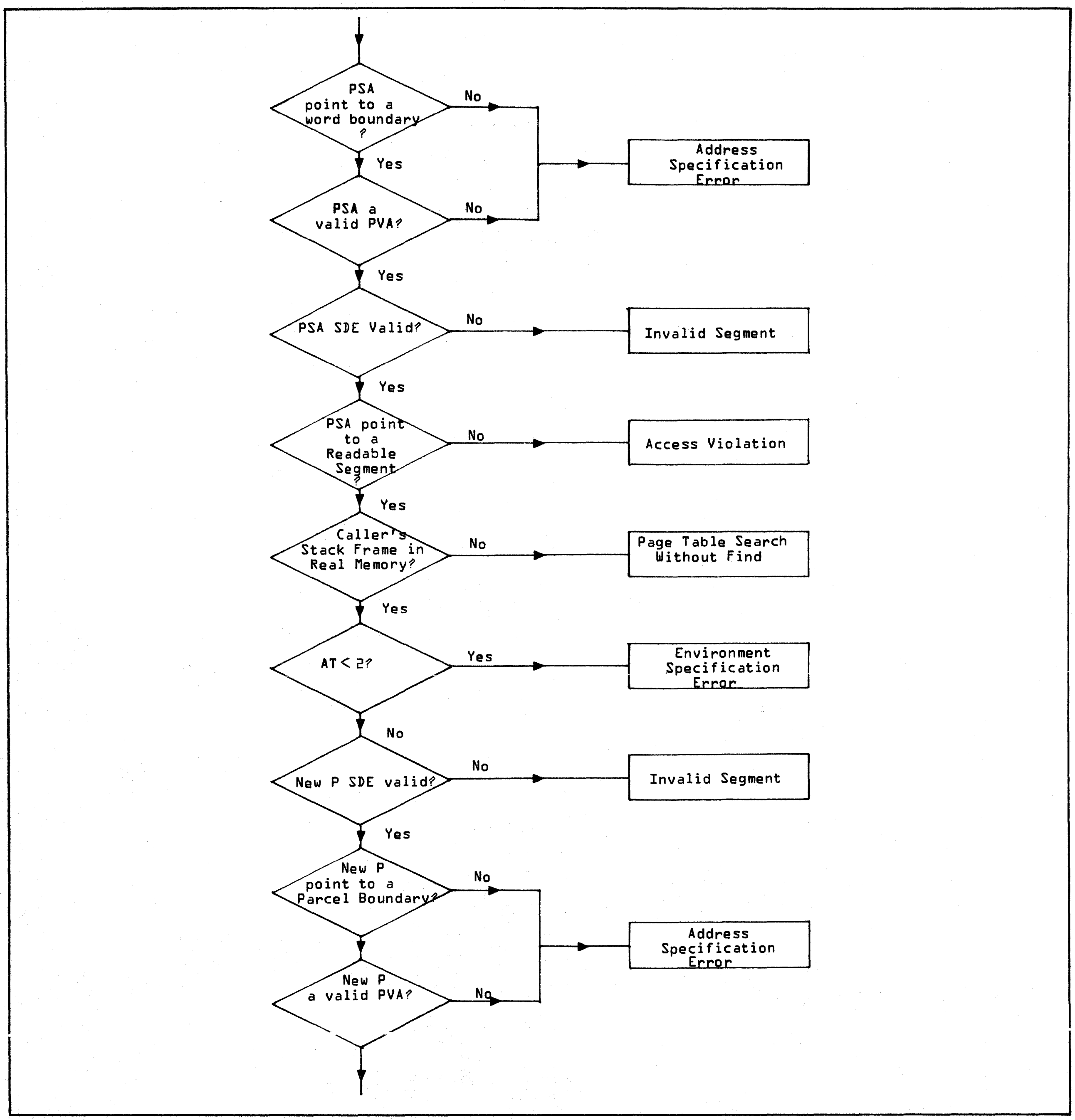

PSA point to a word boundary ?
No
Yes
Address Specification Error
PSA a valid PVA?
No
Yes
PSA SDE Valid?
No
Invalid Segment
Yes
PSA point to a Readable Segment ?
No
Access Violation
Yes
Caller's Stack Frame in Real Memory?
No
Page Table Search Without Find
Yes
AT < 2?
Yes
Environment Specification Error
No
New P SDE valid?
No
Invalid Segment
Yes
New P point to a Parcel Boundary?
No
Address Specification Error
New P a valid PVA?
No

Figure 7-21. RETURN (Sheet 1 of 2)

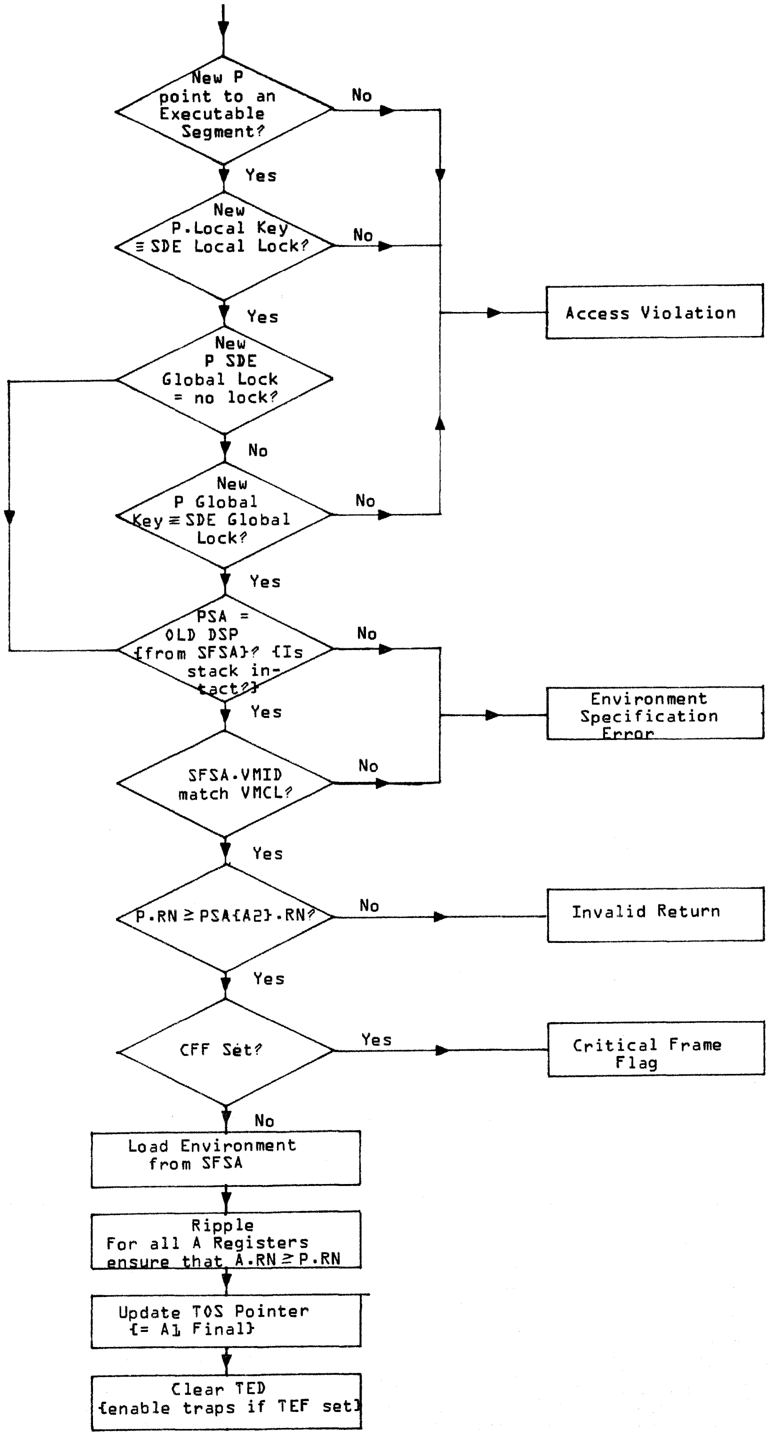

New P point to an Executable Segment?
No
Yes
New P.Local Key ≡ SDE Local Lock?
No
Yes
Access Violation
New P SDE Global Lock = no lock?
No
New P Global Key ≡ SDE Global Lock?
No
Yes
PSA = OLD DSP {from SFSA}? {Is stack in-tact?}
No
Yes
Environment Specification Error
SFSA.VMID match VMCL?
No
Yes
P.RN ≥ PSA{A2}.RN?
No
Invalid Return
Yes
CFF Set?
Yes
Critical Frame Flag
No
Load Environment from SFSA
Ripple For all A Registers ensure that A.RN ≥ P.RN
Update TOS Pointer {= A1 Final}
Clear TED {enable traps if TEF set}

Figure 7-21. RETURN (Sheet 2 of 2)

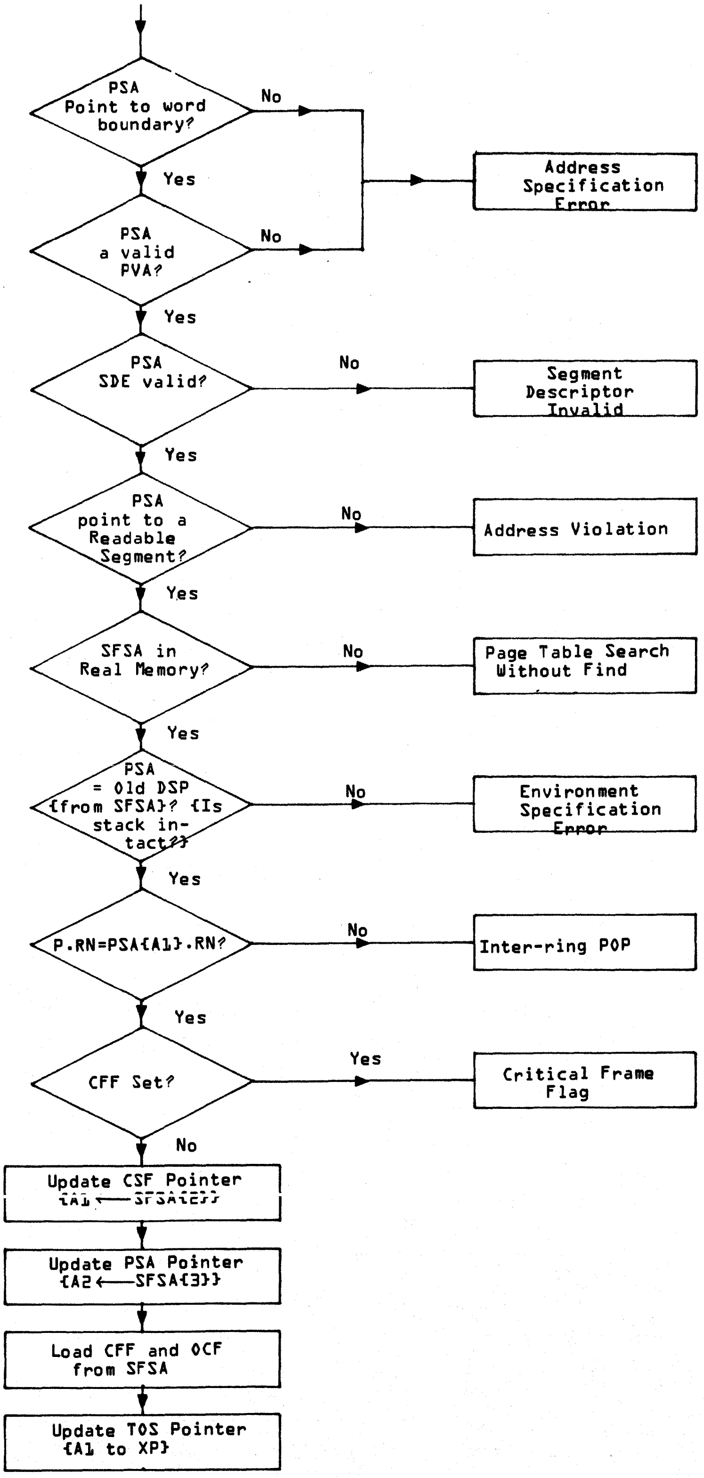


Figure 7-22. POP

The foregoing sections described the basic protection mechanisms provided by the CYBER 180 hardware. This included the primary protection afforded by address space as well as other protection mechanisms within the address space. It is now necessary to describe the techniques for crossing protection boundaries. Two techniques are available, one for switching between address spaces and one for crossing protection boundaries within an address space.

CHANGING ADDRESS SPACES

The exchange jump is used to transfer control from one address space to another. When an exchange jump occurs the machine state changes between Job Mode and Monitor Mode. This state is controlled by a flip-flop which cannot be cleared or set by software other than by an exchange jump when the flip-flop is complemented. CYBER 180 processors are always deadstarted into Monitor Mode via a half exchange. The operating system monitor is the most privileged module of the operating system. It resides in its own address space and has additional, special privileges because it operates in a unique machine state. It is the most trustworthy piece of code in the system. The operating system monitor establishes users' operating environments (by defining their exchange packages) and, consequently, establishes in part their level of security. This concept of trustworthiness is very important to CYBER 180 systems. In general, the lower the ring of execution, the more trustworthy is a code module. CYBER 180 hardware provides the tools necessary to construct a system with any desired level of security. Nevertheless, those hardware facilities are only as good as the software which uses them. For a system to be truly secure, software conventions must be enforced. These conventions form part of the overall architectural design of the system. In concert with this theme the hardware does very little checking on the operating system monitor. In particular, no ring number checks are performed during an exchange jump. If the monitor elects to increase a user's authority (by assigning an A Register Ring Number lower than his ring of execution), then the user runs with that greater privilege. Because of this and other reasons, the operating system monitor should be an extremely small, thoroughly debugged piece of code.

PROTECTION BOUNDARIES WITHIN AN ADDRESS SPACE

Call/Return is the primary mechanism for crossing protection boundaries within an address space. It is the only mechanism for crossing ring boundaries. Two conditions must be satisifed before crossing a protection boundary. First, the caller must be permitted to make the call; second, the callee must not act on behalf of caller with more authority than caller. (The following discussion assumes that the reader is familiar with the basic Call/Return mechanism.) In Call/Return, the caller always provides the callee with his privileges. As a result, it is only possible to make a call from a more privileged ring to a less privileged ring of execution. If the reverse happens then callee, in a less privileged ring than caller, would receive caller's privileges and there would be an immediate, potential breach in security. The hardware prevents this eventuality by detecting attempts either to call outward to a ring of less privilege, or to return inward to a ring of more privilege. Such an attempted breach in security causes an exchange interrupt into the monitor address space.

Most of the information pertaining to security is managed by hardware and is contained in hardware tables - albeit they are constructed by software. The main such table is the segment descriptor table (SDT). Whenever a call is made to another segment across a protection boundary, the transfer must take place in a controlled manner. To accomplish this, calls across protection boundaries do not take place directly but use instead an indirect address (PVA) held in a pointer in a binding section. By software convention, binding sections are not writable in user rings, and are constructed by the Loader based on directives issued by compilers and assemblers. The hardware ensures that all calls across protection boundaries take place via a binding section entry. An access violation interrupt causes an exchange to monitor mode if an attempt is made to bypass this mechanism.

Many other security checks are performed by the hardware during a call. Some are fairly straightforward. For example:

- The Stack Frame Save Area must be in a segment which has write permission.
- Callee´s entry point (obtained from the binding section) must be in a segment which has execute permission.
- Caller´s Global Key must be identical to Callee´s Global Lock, unless either Caller has a Master Global Key, or callee has no Global Lock.

The hardware also ensures that the Caller is within Callee´s call Bracket - as described in the section on Rings of Protection. The pointer to callee´s entry point in the binding section is named a Code Base Pointer (CBP) and has the following format (figure 8-1):

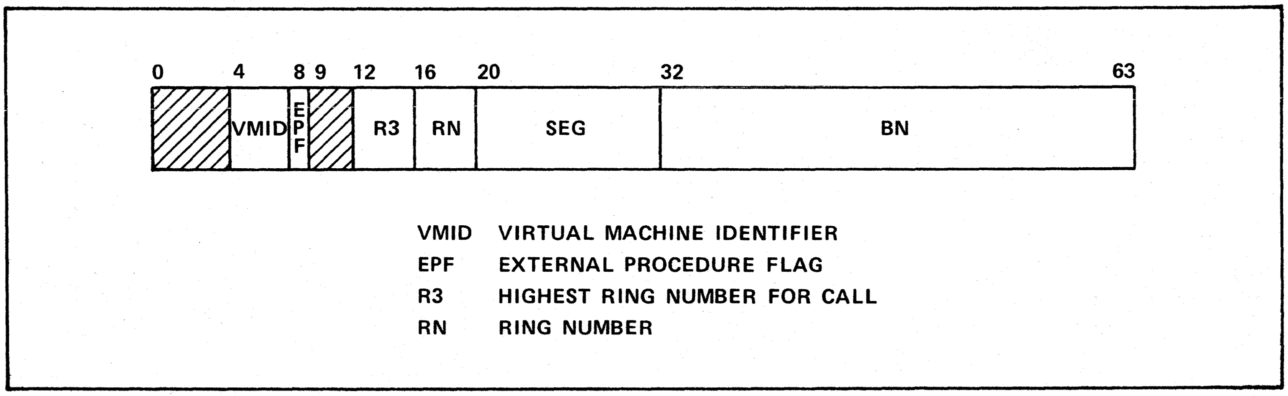

0
4
8
9
12
16
20
32
63
VMID
EPF
R3
RN
SEG
BN
VMID VIRTUAL MACHINE IDENTIFIER
EPF EXTERNAL PROCEDURE FLAG
R3 HIGHEST RING NUMBER FOR CALL
RN RING NUMBER

Figure 8-1. Code Base Pointer - CBP

A call is permitted providing

PVA.RN ≤ CBP.R3

The first check performed by the hardware during a Call ensures that Caller´s ring number (held in the P Register) is within Callee´s call bracket. That is:

P.RN ≤ CBP.R3

In practice this check is made implicitly. An explicit check is made against the Aj ring number as described below. Of itself this check is insufficient since a caller could ask a more privileged procedure to call a third procedure on his behalf to which he does not normally have access.

In figure 8-2, procedure A resides in ring 13 and procedure B resides in ring 11.

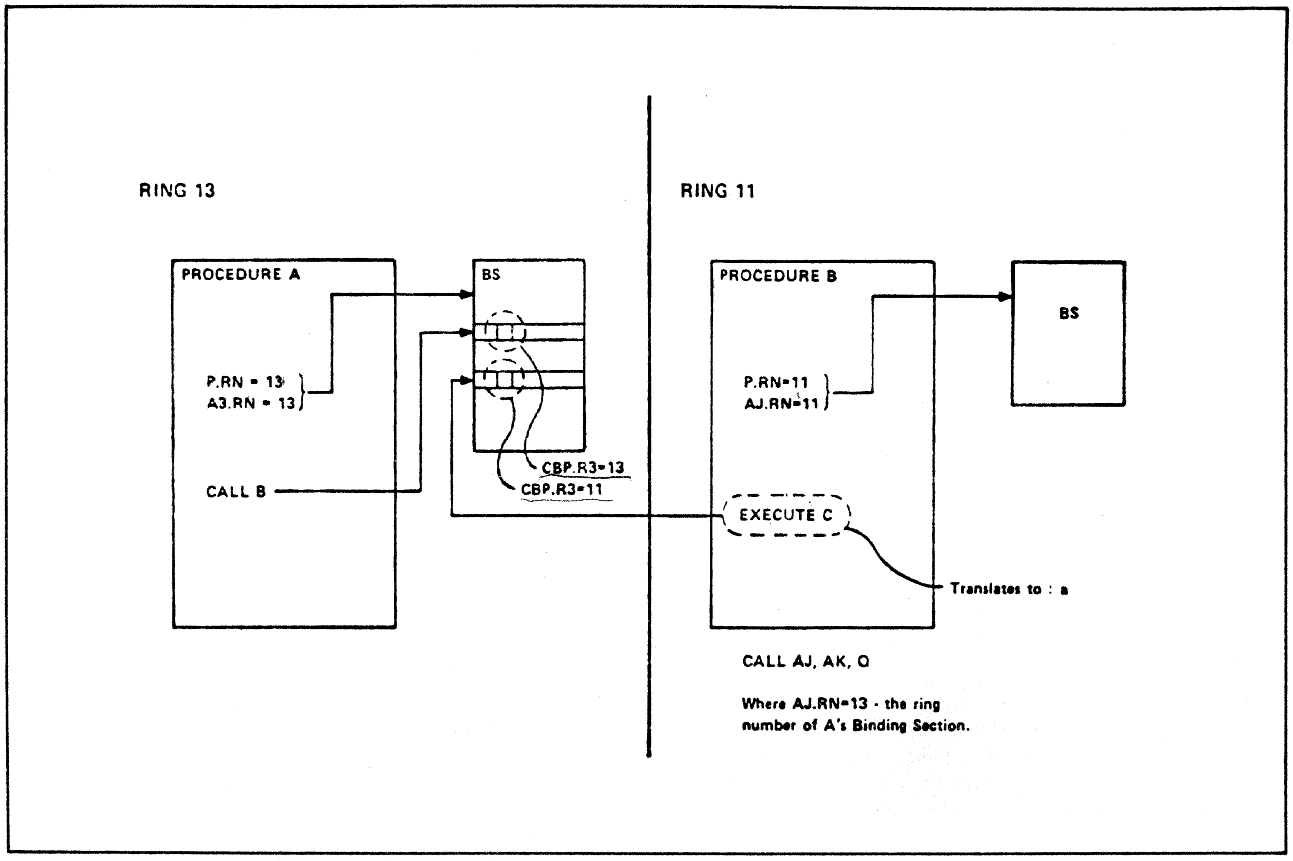


Figure 8-2. Calling a Procedure on Behalf of Another Procedure

Procedure A is allowed to call procedure B since it is within procedure B´s call bracket. Similarly, procedure B is allowed to call on procedure C. However, procedure A is not allowed to call on procedure C, and if procedure B, acting on behalf of procedure A, is asked to call call procedure C, via procedure A´s binding section, then the call must be disallowed. The hardware detects this condition by ensuring that

✱ Aj.RN $\leq$ CBP.R3

where Aj.RN is the ring number contained in the pointer to A´s binding section. This ring number will be greater than or equal to procedure A´s ring of execution, even though it is being used by procedure B. A combination of the hardware ring number voting mechanism and software conventions ensures that the correct ring number has been entered into Register Aj.

Two final security actions take place during a call. First, a software convention which is supported by the hardware places the Argument List Pointer in Register A4. This register contains caller´s Ring number (or a ring number greater than caller´s). This is guaranteed by the hardware. However, software is responsible for ensuring that all parameters used from caller, by callee are referenced via this argument list pointer. This is just one example where a software convention comes into play to ensure that the correct level of security is maintained. The important fact to note is that it is always the more privileged procedure which must enforce the software convention. The final security action during a call is for the hardware to copy caller´s P-left (bits 0-31) into Register X0-left. This provides callee with an unforgeable copy of caller´s privileges - Ring, Global Key, Local

Key and Segment Number. If the more trustworthy callee wants to make absolutely sure he is not being tricked in any way by caller, he should make use of this data to validate all accesses to code and data made on behalf of caller.

So far only the call mechanism has been discussed. However, eventually, the called procedure must return to the caller. At this time care must be taken to ensure that caller does not receive more privilege than he is entitled to receive. The caller's environment is saved in caller's stack, which is a read/write segment within the user's address space which can be modified. It is important to ensure that the ring numbers and key/lock values restored to caller reflect his original privileges. When a return is issued the hardware performs the following tests to ensure that caller's privileges are correct:

1) Caller's stack must reside in a readable segment. This is determined by ensuring that the segment number in A2 (PSA) points to a segment which has read access.

2) Caller's code segment must have execute access. This is determined from the new P Register obtained from the previous stack frame save area.

3) Caller's Local key must be identical to the Local Key of the caller's code segment.

4) Caller's Global Key must be identical to the Global Lock of the caller's code segment, provided the associated segments Global Lock is not a No Lock.

5) For each A Register loaded from the SFSA the normal ring voting procedure is applied. The Key here is that the A Register used to load the remaining A Registers is A2 which callee received directly from caller.

6) For each A Register not loaded from the SFSA, the ring number shall be forced to be at least as great as caller's ring of execution. Hence, if caller did not elect to save certain A Registers, but callee used them, they may have callee's ring number in them. The mechanism of forcing them to at least caller's ring, which is known as Rippling, ensures that there is no breach in security.

The hardware performs two further tests in addition to these. The first test ensures that the initial value held in A2 (PSA) exactly equals the value of A0 (DSP) stored in the stack frame save area. This is known as a stack check. Strictly speaking this is not an access violation, but it does indicate that if the caller's SFSA is inconsistent, it has probably been overwritten. The hardware flags an environment specification error rather than an access violation in this case. The second test which is relevant ensures that callee returns to an outer ring. That is, caller's ring number must not be less than the ring number initially held in A2 (PSA).

Many of the security checks performed by the hardware require that A2 is intact. Since this is handed to callee by the hardware with at least caller's ring number, this provides very tight security protection provided callee does not use A2 for any general purpose in his procedure. Here again, a software convention must be followed to maintain system security. In general, A Registers A0-A2 should be reserved for stack manipulation only.

INTERSEGMENT BRANCH

It was mentioned earlier that the Call/Return mechanism is the primary mechanism employed for crossing protection boundaries. It is the only mechanism available for crossing rings. However, another instruction, Intersegment Branch, may be used to transfer control from one segment to another. Since such a transfer of control involves transgressing a Key/Lock protection boundary the hardware must ensure that the correct Key/Lock transformations occur. The execution of this instruction is illustrated in figure 8-3.

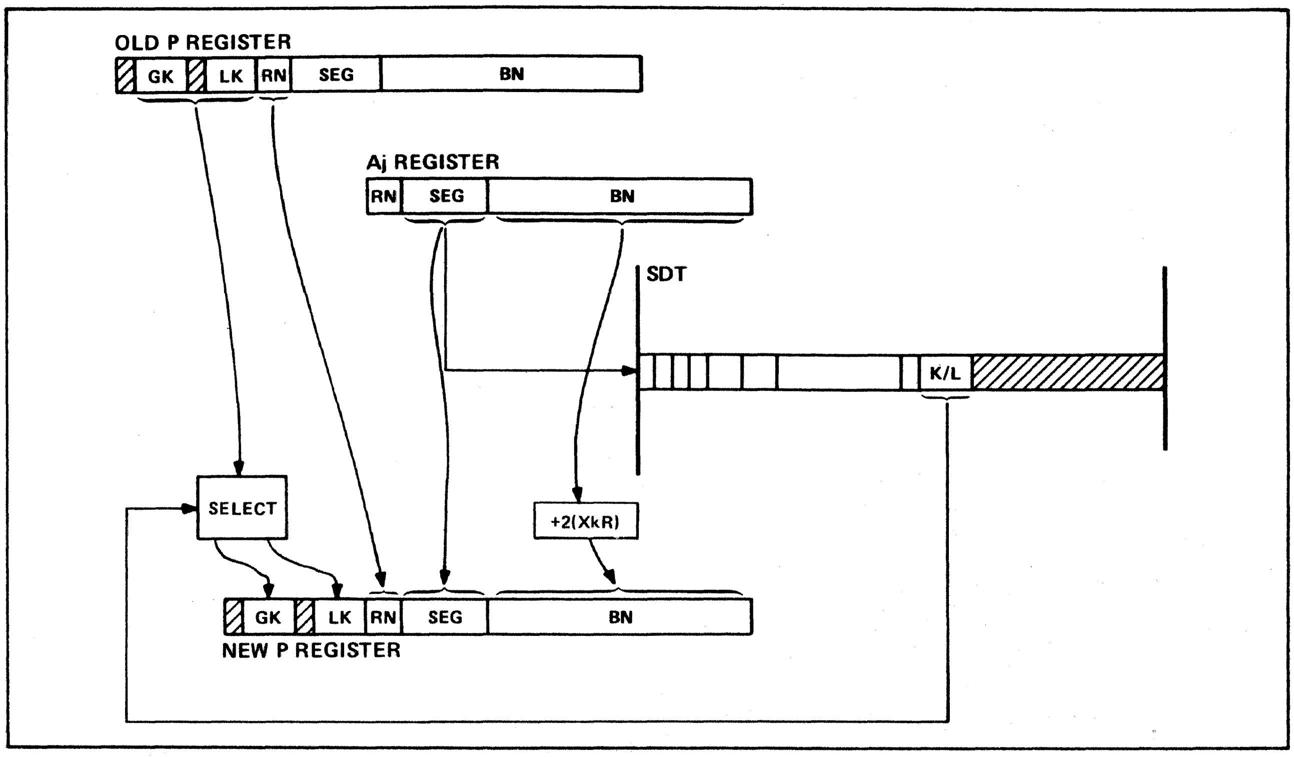


Figure 8-3. Intersegment Branch

Notice that the new P Register ring number is forced to the value in the old P Register. Ring boundaries cannot be crossed by this instruction. In addition, the new P Register Local Key is taken from the associated SDE Local Lock - an executing procedure always runs with its own Local Key. Global Key/Lock transformations follow the rules established for calling a procedure. That is, the new Global Key must be identically equal to the old Global Key unless the old Global Key was a Master Key or the new Global Key was obtained from a No Lock. In summary:

Old Global Key	New SDE Global Lock	New Global Key
0	0	0
0	K2	K2
K1	0	K1
K1	K2	Access Violation

This gives rise to an apparent anomaly. If procedure A with a Master Global Key transfers control to procedure B with a Nonmaster Global Lock, then procedure B will execute with the Nonmaster Global Key. If procedure B subsequently transfers control back to procedure A, procedure A will then execute with a Nonmaster Global Key, even though it is entitled to the Master Global Key. Here again, software conventions come into play. If the previous discussion on Key/Locks is referenced, it will be noticed that, by convention, for rings containing segments with Nonmaster Global Keys, all segments will have a Nonmaster Global Key. All other segments (in other rings) will have Master Global Keys. Hence, the situation described above should never arise since the Intersegment branch instruction never crosses a ring boundary.

WHEN HARDWARE CHECKS OCCUR

The hardware makes the following checks for access violations on each occurrence of the actions listed below:

Read Access to a segment:

(a) The segment must have read access.

(b) The segment must be readable from the ring of the procedure making the access (this is via the ring number of the A Register used to make this access).

(c) The current Local Key exactly equals the Local Lock of the segment, in the absence of a Master Local Key or No Lock.

(d) The current Global Key exactly equals the Global Lock of the segment, in the absence of a Master Global Key or No Lock.

Write Access to a segment:

(a) The segment must have write access.

(b) The segment must be writable from the ring of the procedure making the access (this is via the ring number of the A Register used to make the access).

(c) The current Local Key exactly equals the Local Lock of the segment, in the absence of a Master Local Key or No Lock.

(d) The current Global Key exactly equals the Global Lock of the Segment, in the absence of a Master Global Key or No Lock.

Call to an external procedure:

(a) The CBP must be in a binding section.

(b) The current Stack Frame Save Area must be in a segment which has write access.

(c) The procedure being called must be in a segment which has execute access.

(d) Caller must be within callee´s call bracket.

(e) Caller´s Global Key must be exactly equal to callee´s Global Lock, in the absence of a Master Global Key or No Lock.

(f) The call must not be an outward call.

Return from an external procedure:

(a) The previous Stack Frame Save Area must be in a segment which has read access.

(b) The procedure to which control is returned must be in a segment which has execute access.

(c) The final Local Key (obtained from the P Register in the SFSA) exactly equals the associated segment´s (caller´s) Local Lock.

(d) The final Global Key (obtained from the P Register in the SFSA) exactly equals the associated segment´s (caller´s) Global Lock, provided the associated segment´s Global Lock is not a No Lock.

(e) The return must be an outward return.

In addition, for each A Register ring number which is less than the final P Register ring number, the associated A Registers ring number is set equal to the P Register ring number.

First instruction issued from a new segment:

(a) The segment must have execute access. This check is not repeated for further instructions issued from the same segment. Normally, the check occurs during the execution of the instruction which transferred control to the new segment - that is during the call or intersegment branch.

Branch to a new segment:

(a) The current Global Key (in the P Register) exactly equals the associated segment´s (branch to) Global Lock, in the absence of a Master Global Key or No Global Lock.

These are not the only checks performed by the hardware during the execution of these instructions. These are just the checks which are made to ensure that an access violation is not being attempted. Many other checks are made to ensure that the hardware functions correctly. For example, all branches must be to parcel boundaries, and all calls must be to word boundaries.

SOFTWARE CONVENTIONS

The hardware provides the mechanism necessary to construct a secure system. However, it is the software usage of the hardware which determines the ultimate level of security. For the system to be completely secure, the software must adhere to several conventions. Some of these have been discussed in the previous sections, they are now summarized in this section.

RINGS OF PROTECTION

Since the ring protection mechanism is hierarchical, the higher the privilege assigned to a procedure (the lower the ring number), the more trustworthy that procedure must be. This has two implications: the more privileged a procedure, the more thoroughly it must be checked out. The operating system monitor, which is the most privileged procedure in the system, should be kept as small as possible and thoroughly checked out. Secondly, it is always incumbent on the more privileged procedure to ensure that its own integrity is not jeopardized. In particular, care must be exercised when a procedure acts on behalf of a less privileged procedure. In this case whenever data is referenced via caller´s arguments, callee must reference this data through directly loaded A Registers. In other words callee must ensure that the hardware A Register ring voting is exercised whenever caller´s pointers are used. This is as opposed to loading a pointer in an X Register and then switching this into an A Register (using a Copy X to A instruction) when callee´s ring number could result in caller´s pointer. Since most software will be developed in a high level language, it is the compilers which must adhere to this convention.

KEY/LOCKS

Two types of Key/Locks are provided to protect local code and data and to isolate mutually suspicious subsystems. For these nonhierarchical mechanisms to function as desired, their values must be assigned in accordance with certain conventions regarding the allocation of Key/Lock values.

First of all for Global Key/Locks: for every segment in rings isolating subsystems from each other the Global Lock must be set to a nonzero value. Segments in all other rings must carry a Global Lock value equal to zero. This ensures that users may freely call on subsystems and the operating system, subsystems may freely call on the operating system, yet subsystems are totally isolated from each other.

In general, user, subsystem and system procedures will be assigned with a nonzero Local Lock. That is, no procedure will have a Master Local Key. This ensures that data can be restricted to be written or accessed by only local procedures. Typically, all nonlocal data are not controlled.

CONTROLLING PROCEDURES

As has already been described, much of the security of the system is ensured by the hardware. The hardware utilizes various hardware tables, in particular, the Segment Descriptor Table. These tables are constructed by software procedures. These procedures are very trustworthy; they will execute in low numbered rings but not necessarily Ring 1. They should be developed in such a way that they are self-contained, as small as possible, and impossible to tamper with unless the most stringent security checks have been taken and passed. The security mechanisms which have already been described will take care of security problems when the procedures are being executed. However, when they are modified, either statically or dynamically, a combination of installation procedures and operating system services must be brought into play to ensure that the security of the system is maintained.

USER RESPONSIBILITIES

The hardware and software mechanisms which interplay to provide system-wide security and protection have been described. At first glance it may appear that the utilization of these facilities places a heavy burden on the end-user. Fortunately, this is not the case, although an onus is placed on the installation management. Much of the security of the system is centered on the operating system file system. Every file carries with it the four ring brackets - for Read, Write, Execute and Call - which have already been described. These ring brackets are assigned based on the privilege which the user has been validated. Hence, before a user can log in to the system, in either batch or interactive mode, that user must be known to the system. He will identify himself via a user number and a password. These parameters will direct the system to a validation file containing the privileges of the user.

The normal end-user should be totally unaware of his ring of execution and whether or not his code and data segments carry nonzero Local Locks. For an end-user the Global Lock will be zero. If the user desires to protect some local data, then suitable directives to the operating system will cause the setting of the appropriate Local Lock values. Again the actual value of these Local Locks is of no concern to the user. Consequently, the average end-user who is, for example, running FORTRAN codes need not be concerned with the security mechanisms of the system. At the same time these mechanisms will be in play to isolate him from other users and from the system.

INTERRUPTS PART II 9

The section on CALL/RETURN should be thoroughly understood before proceeding with this section. When the subject of interrupts was introduced their hierarchical nature was described. This hierarchy involved two types of interrupts: exchange interrupts and trap interrupts. In an exchange interrupt the state of the machine changes from job mode to monitor mode, all process state registers are saved in one area of memory and loaded from another area. Included in the process state registers is the P Register and execution continues after the exchange at the address pointed to by the P Register.

In a trap interrupt, although the purpose is similar (that is, to stop the normal sequence of operation and transfer control to another instruction sequence in such a way that the original sequence can be restarted at the point that it was interrupted) the mechanism is quite different. In fact, a trap interrupt is an implicit CALL. Not all the process state registers are saved and very few are loaded with different values. A maximum stack frame save area is created and all A Registers and X Registers are saved in it, along with other key process state registers. The P Register is saved, and processing continues at the address given by a Code Base Pointer (CBP) in the Binding Section of the interrupted process. The address of this CBP is given by the Trap Pointer, and the CBP must point to an external procedure for the trap interrupt to complete.

Trap interrupts, therefore, transfer control to an address within the address space of the executing process. This is important because the trap handler will normally have to make reference to flags and data held in user´s stack. In fact, the outward-call/inward-return mechanism described in the last section was conducted primarily in the user address space even though it was initiated by an exchange interrupt. The free-flag was used to cause an interrupt to take place in user´s address space.

Of major importance to the trap interrupt operation is the management of the condition registers and trap control flags: the trap enable flip-flop (TEF) and trap enable delay (TED). When the trap interrupt is taken the User and Monitor Condition registers are stored in the stack frame save area and the bit (or bits) which cause the interrupt are cleared from the appropriate condition register. Hence, these registers are reset on the trap and can start collecting new fault conditions in an unambiguous manner. Also, when the trap interrupt is taken, traps are disabled - the TEF is cleared.

To reenable interrupts, two mechanisms are available. Simply setting the TEF via a COPY instruction will accomplish this. However, this is not the normal technique used. The trap interrupt is an implicit CALL, and the continuation of normal processing is accomplished by a RETURN instruction. Part of the RETURN mechanism reenables interrupts. The sequence of events is to set the TEF and the TED (by a single COPY instruction), then issue the RETURN. When the TED is set traps are disabled regardless of the setting of the TEF. The RETURN instruction clears the TED which, if the TEF is set, reenables interrupts. Since the TED is cleared only upon completion of the RETURN instruction, problems associated with enabling traps in one instruction step, then returning in a second step, are avoided.

INTERRUPT CONDITIONS

Now that the basic interrupt mechanism has been described, we can proceed to the individual interrupt conditions. CYBER 180 interrupts are precise. That is, the interrupt handler can always refer back exactly to the instruction which caused the interrupt, or which was being executed when the interrupt occurred. However, depending on the nature of the interrupt, the method for tracing back to the instruction in question varies.

A basic architectural philosophy of CYBER 180 is that an instruction is not interrupted during its execution. Conditions which would prevent an instruction from executing are checked before the instruction is committed. The concept of a point of no return was introduced in an earlier section on interrupts and this is an important concept. Any exception conditions detected before the point of no return will prevent the instruction from executing, an interrupt will be taken, and the P Register, at the time of the interrupt, will point to the instruction which could not be executed.

MONITOR CONDITION REGISTER (MCR)

Figure 1-8 lists the conditions recorded in the Monitor Condition Register. Following are some notes on these conditions.

Detected Uncorrectable Error (DUE)

This interrupt indicates that an uncorrectable error has been detected in either the processor or the memory on a reference generated by the processor.

Major data paths, registers, control memories all carry either parity or SECDED. Any error which is detected before the point of no return of an instruction causes the instruction to be retried. A retry counter (one counter which applies to all errors) may be set. If the instruction retry is unsuccessful, then a Detected Uncorrectable Error is recorded. The P Register saved by the interrupt, points to the instruction that was in execution (but before its point of no return) at the time the interrupt occurred. If the error arose after the point of no return, then it will be handled by the complete portion of the instruction execution. In this case the P Register saved by the interrupt points to the instruction following the one which was in execution when the error was detected. This means that there is no way of resuming the instruction stream after the interrupt. However, since the state of the process which was executing is undefined there is little point in doing this.

To aid recovering processors at this point the Processor Not Damaged (PND) flag is set if the fault occurred before the point of no return. When this flag is set the process environment is intact even though further processing may be impossible. This fact may be utilized by the damaged assessor to effect a subsequent restart of the process.

Memory malfunctions are included in this condition. An understanding of the types of errors which can arise in memory may help in an assessment as to the best way to handle them. A simplified picture of memory error detection is shown in figure 9-1. Data transmissions between a processor and memory are checked for correct parity at the processor port, the memory port and at the memory array paks. Memory itself, such as chips and bank logic, has a SECDED.

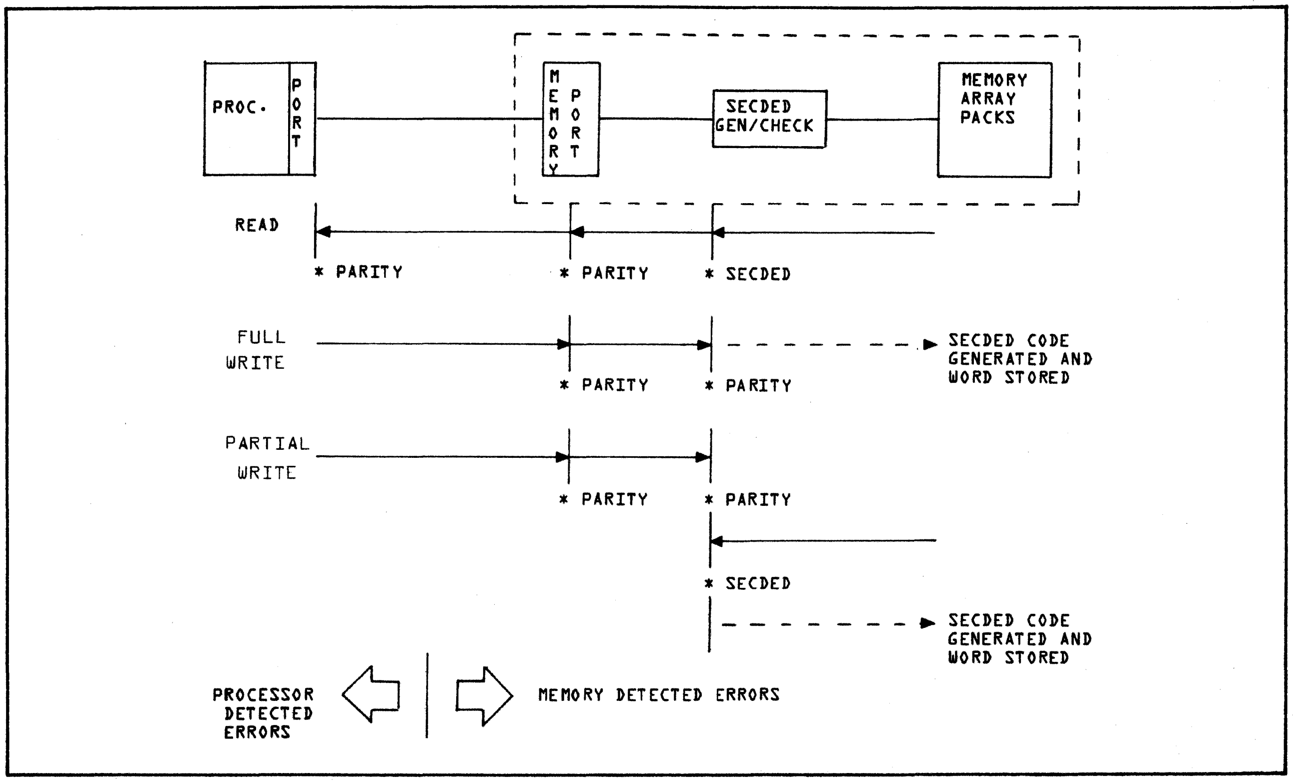


Figure 9-1. Memory Error Detection

A read request is essentially a synchronous process. The requesting processor must wait for the data transmission to complete. The transmission does a SECDED check and then is parity checked at the memory port before being routed to the processor. Errors detected up to this point result in a memory detected malfunction. The transmission is then parity checked at the processor port and an error here results in a processor detected malfunction. The now incorrect data continues to its destination and the process in execution should be handled as previously described.

Two forms of write request are of interest: a partial write, in which only a portion of a 64-bit central memory word is written; and a full-word write, in which a 64-bit word is stored in central memory.

On a partial write, the word being modified must first be fetched from central memory, then rewritten. The data transmission is checked for parity at the memory port and again at the memory array paks (actually at the SECDED generator). The word to be modified is then fetched and checked for SECDED. Finally it is updated, has a new SECDED code generated for it, and then is saved in central memory. Any error which is detected is recorded as a memory detected malfunction.

On a full-word write the sequence is the same as for the partial write, except that the steps where the word is read from central memory and updated are omitted. Hence on a full-word write only parity errors can be detected.

A write request is essentially an asynchronous event. The processor issues the request to memory and continues processing. Any errors which are detected (by memory) are reported back to the processor at a point in the processing which is not associated with the write operation which failed. Hence, it is virtually impossible to relate back to the instruction which is affected by the error. It is pointless therefore, to continue execution of the process in question.

It is not a bad strategy when the errors are encountered, to assume that a user job was being processed, and attempt to take the interrupt. If the error was transient or in a part of the machine that can be bypassed, then the task can be aborted and processing can proceed, maybe after an appropriate reconfiguration has taken place. If a second occurrence of the failure is encountered during the interrupt, the processor will either try to trap or will halt. In the extreme case the processor will halt.

When a DUE is present there may be other bits set in the MCR/UCR as a result of the error, all of which should be disregarded.

Not Assigned

This bit is not set implicitly by any hardware condition, but may be set or cleared explicitly by software on Exchange or Branch on Condition Register as any other condition register bit. When set explicitly, this bit causes program interruptions in a manner identical to bit 48 of the MCR.

Short Warning

A short warning interrupt is one of several asynchronous interrupts (external events) which can arise. In all cases like these the P Register saved by the interrupt points to the next instruction in sequence to be executed. In other words, it is always detected at the next point of no return encountered. A short warning interrupt indicates that within a minimum of 2.5 seconds a system critical component will fail and will automatically shut itself down. It is up to the operating system to take the necessary steps within this timeframe to ensure an orderly restart. System critical components include as a minimum the MG set (main power supply) and all mainframe elements (processors, memories and the IOU). In addition, customers have an option to purchase a Configuration Environment Monitor (CEM) which will detect and report impending shutdowns in key peripheral equipment such as the system disk(s) and controller(s). This interrupt signals an impending shutdown of a key equipment. It may be a power failure, but it could be a high-temperature condition or some other condition which is likely to cause damage to the equipment unless prompt action is taken. This interrupt will never cause the processor to halt.

The short warning bit in the MCR remains set as long as the condition holds. That is, even though an exchange interrupt occurs, and a new copy of the MCR is obtained, the power warning bit remains set. Hence, if the operating system monitor is entered with the traps enabled, an immediate trap results.

There is a second indication of a short warning which is intended for use in CYBER 170 State. A bit is reserved for that purpose in the Processor Status Summary Register. The process of recording the condition is basically the same as that for the MCR. As long as the situation holds, the condition remains recorded in the Status Summary Register. In the event that it clears (a transient power loss) the condition goes away. This enables software to monitor for restart conditions.

Instruction Specification Error

This is one of a class of errors where either the user has made an error (such as executing data) or is deliberately trying to tamper with the system. In either event an exchange interrupt is taken with the P Register saved (at JPS) pointing to the instruction which caused the error. The only case where a user may be deliberately trying to destroy the system is when the user attempts to execute a monitor instruction in job mode. These special instructions for use only by monitor are described in a later section. The interrupt enables the operating system to abort the job and report to the end-user the precise instruction, and address within the process being executed which caused the fault.

Address Specification Error

Certain instructions require a particular form of an address to be used. If the required form is not used this interrupt will occur, and the operating system can follow the actions suggested for an instruction specification error. Here also the P Register saved (at JPS) points to the instruction with the faulty address. In addition, the faulty address is loaded into the Untranslatable Pointer Register (UTP).

C170 Exchange Request

CYBER 180 is designed such that it may execute the instructions not only of CYBER 180 but of other machines as well. CYBER 170 is the most important of these. On CYBER 170 the IOU can initiate an exchange jump in the CPU. However, when this happens on CYBER 180 it can only be executed if the CYBER 170 virtual machine is being executed. If the CYBER 180 virtual machine is being executed, then an exchange request interrupt occurs and the CYBER 180 monitor must then exchange to the CYBER 170 virtual machine in order for the request to be satisfied. This is an asynchronous interrupt and the P Register stored (at JPS) by the interrupt is set accordingly.

Access Violation

This interrupt occurs when a user attempts to access code or data to which he has not been granted access privelege. The CYBER 180 protection mechanism is described fully in the section dealing with virtual memory. It is a mechanism which is built into the hardware and any attempt to circumvent it leads to this interrupt. This is the same as an instruction specification error in that the P Register (saved at JPS) points to the instruction which attempted to violate the protection mechanism.

Environment Specification Error

This interrupt indicates that the environment has been destroyed in some way, typically by a programming error. The destruction is such that an illogical or impossible situation develops and further processing is impossible. The most common cause of this is the destruction of the information in the stack by a user. Since the stack has read/write access and contains dynamic variables along with link information for calls and returns, this is not uncommon. Further processing is impossible and the operating system must abort the job. This interrupt behaves exactly as an instruction specification error with one exception. In most cases the P Register saved during the interrupt points to the instruction which caused the error. However, this error can arise when the processor is attempting to trap or exchange on another interrupt. If a trap was being attempted but could not complete, then an exchange will be attempted if the machine is in job mode. If the exchange is successful, then the P Register saved (at JPS) points to the instruction which originally caused the trap and the trap exception bit will be set. The operating system must abort the job at this time, but should check for the trap exception and then report to the user:

a) the instruction executed or about to be executed when the original interrupt occurred

b) the nature of the original interrupt

c) the final reason for the job abort - which is the environment specification error

If the machine is in monitor mode when the trap exception occurs it will halt, since the monitor´s environment has been destroyed.

For exchange interrupts the situation is different. If an exchange from monitor to job is attempted such that the job in question is not permitted to execute the given virtual machine to which it is exchanging, then the following happens:

a) The exchange from monitor to job completes and an environment specification error is detected.

b) An exchange is taken immediately from job to monitor.

That is, the environment specification error is associated with the job and recorded in the exchange package stored at JPS. Also, the P Register saved in this exchange package is identical to that which was loaded from JPS when the original exchange from monitor to job was attempted.

If a virtual machine mismatch occurs on an attempted exchange from job to monitor, then the hardware must have failed in some serious, undetected manner. The processor has no recourse other than to halt. This situation is unlikely to occur and is also difficult to detect. There will be no indication in the processor error logs and there will be no indication in either the exchange package at JPS or that at MPS. An investigation of the PVA in the P Register at the time of the halt (by the MCU) and an investigation of the Monitor Condition Register (and Monitor Mask Register), followed by a check on the registers controlling virtual machine switching should reveal the nature of the problem. If unexplained processor halts are to be avoided, then the code in the MCU should include a check for these conditions.

External Interrupt

An external interrupt is an asynchronous interrupt. It is a signal to a processor that another processor requires some action to be performed. Precisely which processor is making the request and the nature of the request must be relayed by software convention. A message buffer must be set up in central memory to contain this information. Once the request is satisfied, an exchange jump back to job continues normal processing.

Page Table Search Without Find

This is a simple page fault - a user has tried to access a page which is not in real memory. The operating system must arrange for the page to be brought into memory before processing can continue. This condition is always caught in the prevalidation of an instruction, that is, before the point of no return. Consequently, the P Register saved by the interrupt (at JPS) points to the instruction which could not be executed because of the missing page. In addition, the Untranslatable Pointer register (UTP) contains the address (PVA) which gave the page fault. Hence, in order to satisfy the page fault the operating system does not have to trace back through the code being executed. It can gain all the information it requires from the UTP. Once the page fault is satisfied, an exchange back to job continues normal processing. Page faulting does not always result in loading a fresh page in memory. The operating system must apply various safeguards to ensure that a process that is running away in a write loop does not consume all of real memory. Typically, this can be done by limiting the size of the segments being used by the user.

One last point: certain code segments of the operating system must be wired down, that is, not paged. This is to avoid the recursion of faults which could otherwise occur. For example, the Page Fault Handler, itself, cannot get a page fault. Such a condition normally causes a processor halt via the hierarchy mechanism of the CYBER 180 interrupt system.

System Call

Unlike the conditions discussed to this point, the system call condition is not an interrupt condition but is a flag for the operating system monitor. The value of the corresponding bit in the Monitor Mask register has no affect on the setting of this flag. A process executing in job mode may need to make a request on the operating system monitor for some action. To do this, the process stores a request message in a message buffer, then issues an exchange jump. This switches the machine state from job to monitor, and to all intents and purposes appears to monitor as if an exchange interrupt occurred. An investigation of the MCR at JPS reveals the system call flag set, and the necessary action is taken. The P Register saved during the exchange (at JPS) points to the instruction following the exchange jump, hence an exchange back to job continues normal processing. Unlike true interrupt conditions, if this flag is set by the special system instruction which modifies the MCR, then an interrupt does not result.

Care should be taken by the interrupt handler to ensure that only the MCR is checked for that condition. This is different from the normal mechanism in which the logical product of the MM and MCR is checked. The simplest procedure for the operating system to follow is to ensure that the MM bit (bit 10) is always set on exchange to job.

System Interval Timer

The System Interval Timer (SIT) resides in a processor state register. It is the single timer for the entire system. It is a 32-bit counter which is decremented once every microsecond. When it counts to zero this interrupt is taken. This is an asynchronous interrupt and the P Register stored at JPS points to the instruction following the one being executed when the SIT decremented to zero. The SIT is intended to be used for time slicing and accounting. Once the counter has decremented to zero it does not stop counting. That is, it will next decrement to -1 (2**32 -1) and continue decrementing.

Invalid Segment/Ring Number Zero

The invalid segment condition bit in the MCR combines two conditions. The first of these is a true invalid segment and the second is an unlinked pointer, ring number zero. An invalid segment condition arises when either a segment descriptor table entry (SDE) has been flagged as an invalid entry (VL field = 00), or when the Segment Table Length (STL) has been exceeded. This latter condition occurs when the Segment Number (SEG) portion of a PVA is greater than STL, that is, when SEG > STL. For these conditions the P Register stored at JPS points to the instruction which attempted the central memory access which gave rise to the condition.

A ring number zero condition arises when an unlinked pointer has been loaded. The operating system must arrange for the loader to form the necessary links as described previously in the section dealing with dynamic loading. In all cases the unlinked pointer is placed in the Untranslatable Pointer Register (UTP) and contains all the information necessary for the operating system to form the appropriate link. When an unlinked pointer is loaded, then the load completes before the interrupt is taken. Hence, the P Register stored at JPS points to the instruction following the load instruction which loaded the unlinked pointer. This means the unlinked pointer was loaded into an A Register and the operating system must take care to replace this register value with the correct, linked pointer.

Outward Call/Inward Return

The ring hierarchy has been established such that procedures in inner rings may access code and data in outer rings (rings with higher numbers and lower privilege), and procedures in outer rings may CALL on procedures in inner rings in a controlled manner. This has been described in the section on CALL/RETURN. This condition has been provided to prevent a user from attempting an outward call or an inward return, and thereby causing a possible security breach. The P Register stored at JPS points to the CALL or RETURN instruction in question, and the operating system must either abort the job or simulate the required call. This latter process has already been described.

Soft Error Log

This condition bit sets when the processor encounters a hardware error which was corrected by the hardware. This includes correctable errors in the processor itself and may include, if selected, single bit errors in central memory. Examples in the processor include a successful instruction retry, cache or MAP parity errors, and so forth. These vary from processor type to processor type. These correctable errors are also reported in the Status Summary register (for processor or memory) hence it is not necessary for the processor to be interrupted on every correctable error. The choice may be made to ignore this interrupt (by not setting the appropriate bit in the Monitor Mask register) and treat these errors in an asynchronous manner via the Maintenance Control Unit (MCU) - a designated PP in the IOU. The P Register stored at JPS points to the next instruction to be executed. That is, if the interrupt is taken, then the operating system monitor after processing the interrupt has only to issue an exchange to continue normal processing.

Trap Exception

Trap exception is similar to system call in that it is not an interrupt condition but is a flag to the operating system. The flag indicates that for some reason a trap interrupt was attempted but could not be completed because a condition was encountered which prevented it. Hence, at least two other bits will be set in the MCR whenever the Trap Exception bit is set, one being the bit which prevented the trap from completing, and the other being the bit that caused the trap. An example of this process is an arithmetic overflow encountered and a trap attempted. However, in attempting to store the Stack Frame Save Area a page fault in the Stack is detected and an exchange interrupt taken. After satisfying the page fault the operating system exchanges back to the user (taking care to clear the Trap Exception bit in the MCR), whereupon the trap will take place since the condition has not been removed from the UCR. The P Register stored at JPS contains the PVA which would have been laid down in the Stack Frame Save Area had the trap been successful.

General Notes on the MCR:

- The condition bits in the MCR have been sequenced in a priority order from left to right with the most serious conditions towards the leftmost end of the register. The recommended (but not mandatory) order of processing is in this sequence. Consequently, the fatal system conditions occupy the first three bits in the register and the trap exception flag the last.

- Whenever an invalid pointer is encountered for whatever reason, an interrupt occurs and the invalid pointer is placed in the Untranslatable Pointer Register.

- Interrupts which may occur in multiples of particular interest are the four which cause entries in the UTP: Invalid Segment/Ring Number Zero (ISG), Address Specification Error (ASE), Access Violation (AV), and Page Table Search Without Find (PSWF). If these occur in combination, then the following precedence applies to the PVA entered in the UTP: ISG, ASE, AV, PSWF.

USER CONDITION REGISTER (UCR)

Figure 1-9 lists the conditions recorded in the User Condition Register. Following are some notes on these conditions. Rather than repeat information, the arithmetic conditions have been grouped into classes which describe their behavior.

Privileged Instruction Fault

This is really a monitor condition and could have been implemented in the MCR. However, by recording it in the UCR there is an opportunity for handling the condition from within the user's address space. This is the first of four monitor conditions which are recorded in the UCR. They are all characterized by the fact that they cannot be stacked and are termed the nonstackable conditions. In practice this condition, which arises because a user has attempted to execute a privileged instruction in a nonprivileged mode, will normally be handled by the operating system directly. For a discussion on the privileged modes of operation of CYBER 180, refer to section on system instructions.

The P Register stored in the Stack Frame Save Area (SFSA) points to the instruction which gave the fault. The execution of this instruction is inhibited.

Unimplemented Instruction

This is the second monitor condition which is flagged in the UCR. It provides a capability for emulating a model dependent instruction with suitable software. Since the emulation should occur from within the user's address space a trap rather than an exchange is taken. The P Register stored in the SFSA points to the illegal instruction which caused the trap.

Free Flag

An example of the use of this flag is given in the section on CALL/RETURN mechanism. In that case an outward CALL was simulated by the operating system from within the user's address space. The transition from the monitor's address space to the user's address space was made by setting the Free Flag in the exchange package at JPS, and executing an exchange jump from monitor to job. In the prevalidation of the next instruction to be executed in the user's job, the Free Flag is detected and a trap taken. The P Register stored in the SFSA points to the next instruction to be executed in the user's code. This UCR condition is unique in that it takes priority over MCR conditions which may arise at the same time.

Process Interval Timer

The Process Interval Timer (PIT) is a 32-bit counter which decrements once every microsecond. Each process has a unique counter when it is in execution. Whenever a PIT reaches zero a condition bit sets in the UCR, and if enabled, a trap is taken. The PIT continues counting at this time. One microsecond after the PIT has zeroed, the counter assumes a value of -1 (2**32-1) and the decrementing continues. This condition is an asynchronous interrupt similar to the SIT which is recorded in the MCR. The P Register saved in the SFSA points to the next instruction to be executed. In other words, when the trap handler has completed its processing and issued a RETURN, normal instruction execution resumes.

Inter-ring POP

This is the third monitor condition which is recorded in the UCR. The POP instruction is described in the section on CALL/RETURN mechanism. Its function is to dispose of stack frames, typically during tidy-up when a process is being terminated. The POP instruction merely moves pointers (DSP,CSF,PSA,TOS) which point at a given stack. It does not contain any of the safeguards required when crossing rings. Hence, if a ring crossing is attempted in the tidy-up process, a trap is taken and software procedures are invoked to ensure the ring crossing takes place in a controlled manner. The P Register saved in the SFSA points to the POP instruction which attempted the ring crossing, and whose execution was inhibited.

Critical Frame Flag

This is the fourth and final monitor condition which is recorded in the UCR. The Critical Frame Flag (CFF) is a software flag which is acted on by the hardware. Software sets this flag to prevent the disposal of certain stack frames which may be shared by separate tasks running in the same address space. The flag is cleared by CALL and trap so that each instance of a procedure begins in a noncritical state. It is likewise restored on a POP or a RETURN in order for the criticality of the current stack frame to be determined. This condition provides an interrupt into the user´s address space and the trap handler must determine how the stack frame can be disposed. In other words the criticality of the stack frame is set by software convention, and any alteration of the criticality or disposal of the stack frame must be under the control of the same software. The P Register, saved in the SFSA, points to the RETURN or POP instruction which attempted to eliminate the critical stack frame.

Keypoint

The keypoint condition indicates that software is to collect hardware performance data at this point of the program. For a full discussion of this topic see the section dealing with Performance Monitoring. In this case the P Register saved in the SFSA points to the instruction following the keypoint instruction which caused the trap.

General Notes:

a) The first seven conditions recorded in the UCR have been described above. They comprise four monitor conditions and three user conditions. The bits in the User Mask Register (UM) corresponding to these seven conditions are permanently selected by the hardware. Hence, if one of these conditions arises, and traps are enabled, then a trap will be taken.

b) For the four, nonstackable, monitor conditions, execution of the instruction causing the trap is always inhibited. Furthermore, the offending instruction is rarely if ever executed. However, since the P Register saved in the SFSA points to the offending instruction, the trap handler must advance the value of the P Register saved in the SFSA, before issuing a RETURN.

Debug

This condition indicates that a debug condition was met such as a storage reference made or branch taken. For a full discussion of the facilities in this area, see the section on debug. The P Register saved in the SFSA points to the instruction which caused the trap to occur.

Invalid BDP Data

This condition indicates that a BDP instruction encountered data which did not match the required format. In this case the P Register saved in the SFSA points to the instruction which encountered the invalid BDP data. For a full discussion of required formats, refer to section on BDP instructions.

Arithmetic Conditions

The remaining seven user conditions are arithmetic conditions.

Divide Fault
Arithmetic Overflow
Floating Point Indefinite
Arithmetic Loss of Significance
Exponent Overflow
Exponent Underflow
Floating Point Loss of Significance

They fall into two classes depending on whether the P Register stored in the SFSA points to the instruction which caused the fault or whether it points to the instruction following the one which caused the fault. In addition, the instruction may or may not be executed before the trap is taken. In general, the intent of CYBER 180 is to be able to identify the instruction which caused the fault. This means that the P Register saved in the SFSA normally points to the instruction in question. This is particularly important since it is impossible to back up the instruction stream when an instruction has executed and the P Register has been advanced. (It follows automatically that the instruction execution is normally inhibited.) However, in the case of floating-point instructions the following is apparent. If the result is not indefinite or infinite, but is an exponent overflow or underflow condition, then it will be a true value even though it is out-of-range of the standard floating-point numbers. In the general scheme of events this is unimportant, but code can be provided to correct the situation without loss.

The vector instructions require some special attention because of the multiple operands involved. Floating-point vectors may encounter several (up to four) of these conditions. The vector instruction execution is not inhibited and the interrupt occurs after the completion of the vector instruction.

- If a single condition is encountered, the P Register saved in the SFSA will follow the pattern for scalar instructions.

  P points to the following instruction for:

  Exponent Overflow
  Exponent Underflow
  Floating Point Lost of Significance

  P points to the vector instruction which encountered the interrupt for:

  Divide Fault
  Arithmetic Overflow
  Floating Point Indefinite
  Arithmetic Loss of Significance

- If multiple conditions are encountered which require different values of P, the P will always be set to point to the instruction following the vector instruction which encountered the multiple conditions.

Since the conditions themselves are self-explanatory the discussion below groups them into two major categories and is restricted to irregularities.

Conditions Where the Instruction is Inhibited

For this class of arithmetic faults, the execution of the instruction which caused the fault is inhibited, and the P Register saved in the SFSA points to that instruction. Exceptions are noted in the following text. The conditions which fall into this category are:

- Divide Fault (integer, decimal, floating-point)
- Arithmetic Overflow (integer, decimal)
- Floating-point Indefinite
- Arithmetic Loss of Significance (integer,decimal)

General Notes:

a) Floating-point indefinite falls into this category since it can arise on a branch instruction (32-bits) as well as on an arithmetic operation (16-bits). Hence, it is not possible to backup the instruction stream.

b) A divide fault occurs either on a divide by zero or when the divisor is an unnormalized floating-point number. The latter case does not necessarily result in a divide fault, but the single and double precision quotient operations do not prenormalize. (However, all floating-point operations postnormalize to the extent that normalized numbers will emerge if normalized numbers are input to the floating-point unit.) Also, if traps are disabled or the divide fault interrupt is not selected, then the instruction is still inhibited and execution continues at the next instruction in sequence.

c) Traps on user conditions have been included for convenience. They may be selected or disabled by the user via the UM, and cause an interruption to the normal execution sequence. When the condition has not been selected but is encountered, the appropriate bit is still set in the UCR, and may be tested and cleared by a special instruction: Branch on Condition Register. This instruction is discussed in the section dealing with system instruction.

Conditions Where the Instruction is Executed

For this class of arithmetic faults, the execution of the instruction which caused the fault is completed, and the P register, saved in the SFSA, points to the next instruction. The conditions which fall into this category are:

-Exponent Overflow
-Exponent Underflow
-Floating Point Loss of Significance

General Notes:

1) It is important that the instructions during which the condition arose are executed, since the CYBER 180 floating point format has been chosen such that, even though an exponent overflow or underflow occurs, a true result is returned. This is explained in more detail in the section on floating-point instructions. It provides a programmer with the opportunity to scale variables and continue processing if desired.

SIMULATED INTERRUPTS

There are two ways in which an interrupt can be generated artificially. These are by setting a bit in the UCR (by a Branch on Condition Register instruction), and by setting a bit in the UM when the corresponding bit is already set in the UCR. In both instances, the P Register saved in the SFSA points to the instruction following that which set the bit in either a Mask Register or a Condition Register, that is, following a Branch on Condition Register (BCR) or a copy. It is not good practice to set bits in a UCR. This facility has been included as a diagnostic aid to verify that the interrupt system is functioning correctly.

MULTIPLE INTERRUPTS

When more than one interrupt condition arises at one time the following rules apply:

a) Exchange interrupts are always serviced before trap interrupts by the hardware.

b) When multiple interrupts of the same type occur simultaneously, they are all recorded in the condition registers. The precise mechanism is processor model dependent: some processors recording all coincident conditions simultaneously, others recording them one at a time. The interrupt handlers must accommodate multiple, simultaneous interrupts.

Multiple interrupts arise because of the asynchronous nature of certain interrupt conditions. Since the P Register saved by the interrupt is intended to point to either the instruction which caused the fault or to the following instruction, it is important to understand what is contained in that register. The following general rules should help.

a) A Detected Uncorrectable Error (DUE) always takes precedence and leaves the P Register in an undefined state.

b) The synchronous interrupts take priority over the asynchronous interrupts.

Care must be taken when designing and generating interrupt handlers, and these rules must be applied at all times. The following example will clarify the pitfalls.

When an asynchronous interrupt occurs (for example, a PIT), the P Register saved in the SFSA points to the next instruction to be executed. Hence, the trap is taken, processed, and a RETURN issued, which continues normal processing. However, if a PIT occurs simultaneously with, for example, an unimplemented instruction, then the P Register saved in the SFSA points to the unimplemented instruction. If only the PIT is acted on, then the RETURN will cause the unimplemented instruction fault to be detected again. However, if only the unimplemented instruction was acted on the PIT would never be seen. This is because on a trap the UCR is saved in the SFSA and the live UCR is zeroed. It is the live UCR which carries back across the return. This is different from the exchange mechanism when a fresh copy of the condition registers is invoked (either from the exchange package at MPS, or that at JPS) for each exchange interval. If an exchange condition is processed, but the bit in the condition register in the exchange package in memory is not cleared, then an exchange loop will follow.

Probably the safest rule to follow is to process all conditions which have arisen, and which have been selected, at one time.

Finally, software cooperation is needed when interrupts are caused artificially (for example, by setting the UM dynamically). If the normal action taken by the interrupt handler is to advance the P counter, then in this case an instruction will be omitted. For example, in this sequence

```
          ENTE    XF,X'E6'
          ENTP    XE, 1
          CPYXS   XE, XF
          LX      XF, A5, ABC
```

the LX instruction may be spaced over by the interrupt handler. To avoid such an occurrence, it is recommended that a do nothing (CPYXX X0,X0) be inserted immediately following the instruction causing the trap - in this case (CPYXS XE,XF).

Complete details of the operation of the central processor instructions for CYBER 180 may be found in the MIGDS. It is not the intention to repeat all the information here. This section will confine itself to features of these instructions which are unique to CYBER 180, and it will highlight some characteristics about which questions have frequently been asked.

REGISTERS

CYBER 180 has 33 general purpose registers. These are the P Register, 16 64-bit X Registers, and 16 48-bit A Registers (figure 10-1).

The P Register or Program Address Register contains the PVA of the instruction in central memory during the time it is read, interpreted and executed by the processor. It also carries information relating to the privacy of the code segment being executed. This information contains the current ring of execution and the global and local keys of the process.

The 16 X Registers are named X0-XF using hexadecimal notation. They are general purpose registers used to hold logical quantities, signed integers and signed floating-point numbers. They have a left half and a right half and there are instructions which operate on the entire 64-bit register, and other instructions which operate only on the lower 32-bits (bits 32-63) in which case the register is referred to as X-Right. Instructions which operate on the lower 32-bits of an X register do not modify the upper 32 bits.

The 16 A Registers are used to hold PVA´s to address operands in central memory. They are referred to as A0-AF and are identical to the low-order 48-bits of the P Register. There is no implied relationship between the A Register and X Register as on CYBER 170.

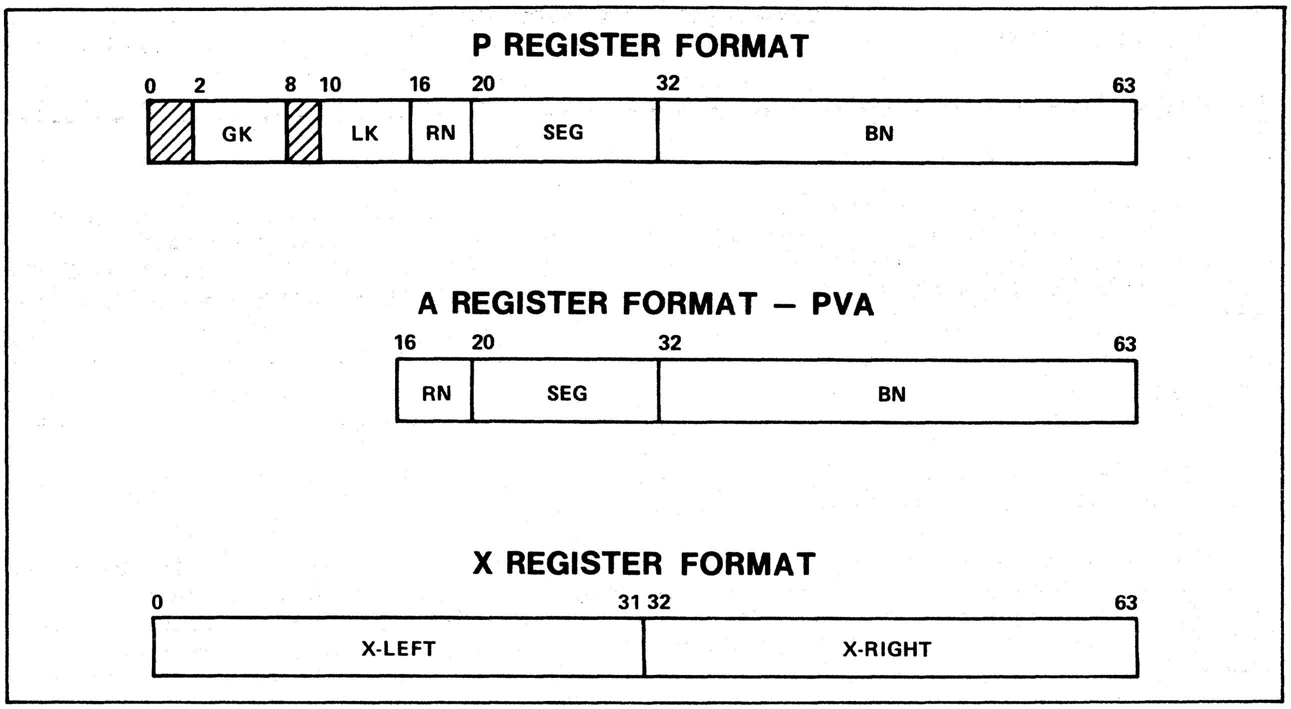


Figure 10-1. General Purpose Registers

GENERAL STRUCTURE

Bit numbering on CYBER 180 processors is left to right zero origin. This system applies to words in memory, bytes in words, bits in bytes, bits in registers, and so forth.

CYBER 180 is a byte addressable machine - unlike CYBER 170 which is a word addressable machine. For this reason there is no No Operation (NOP) instruction. All instructions consist of an integral number of bytes and instruction parcels can span word boundaries or page boundaries but not segment boundaries. Each segment is a unique entity and has no relation to the segments numbered immediately prior to or following it.

CYBER 180 instructions are noninterruptible. The processor always prevalidates an instruction before executing it. It is not possible to have a page fault during the execution of an instruction. This design philosophy has been chosen to simplify the development of CYBER 180 processors and is directly responsible for restrictions on the lengths of certain operands. These restrictions are particularly in evidence in the BDP instructions.

Another major difference between CYBER 180 and CYBER 170 is that CYBER 180 instructions are basically 2-address instructions whereas CYBER 170 instructions are basically 3-address.

INSTRUCTION GROUPS

There are four groups of instructions on CYBER 180 as follows:

General Instructions (76) - Load/Store, Integer Arithmetic, Logicals, Branches, Enters, Copies, Address Arithmetic, Shifts, and so forth.

BDP Instructions (18) - Moves, Compares, Decimal Arithmetic, Translate, Edit.

Floating-Point Instructions (16) - Floating-Point Arithmetic, Branches, Compare.

System Instructions (19) - Subroutine link, page table management, cache management, maintenance register copies, interlocks, and so forth.

These groups are discussed separately in that sequence.

NOTE

This document does not include a discussion of CYBER 180 vector instructions.

GENERAL INSTRUCTIONS

Four instruction formats (figure 10-2), one one-parcel and three two-parcel instructions are used for the general instructions. These instructions are termed jkiD, SjkiD, jk and jkQ instructions. j,k and i refer to register subscript designators such as Xj, Ak, Xi. Q is always a signed (2´s complement) 16-bit constant, D is an unsigned 12-bit constant (except when used in the shift and scale instructions), and S is a suboperation.

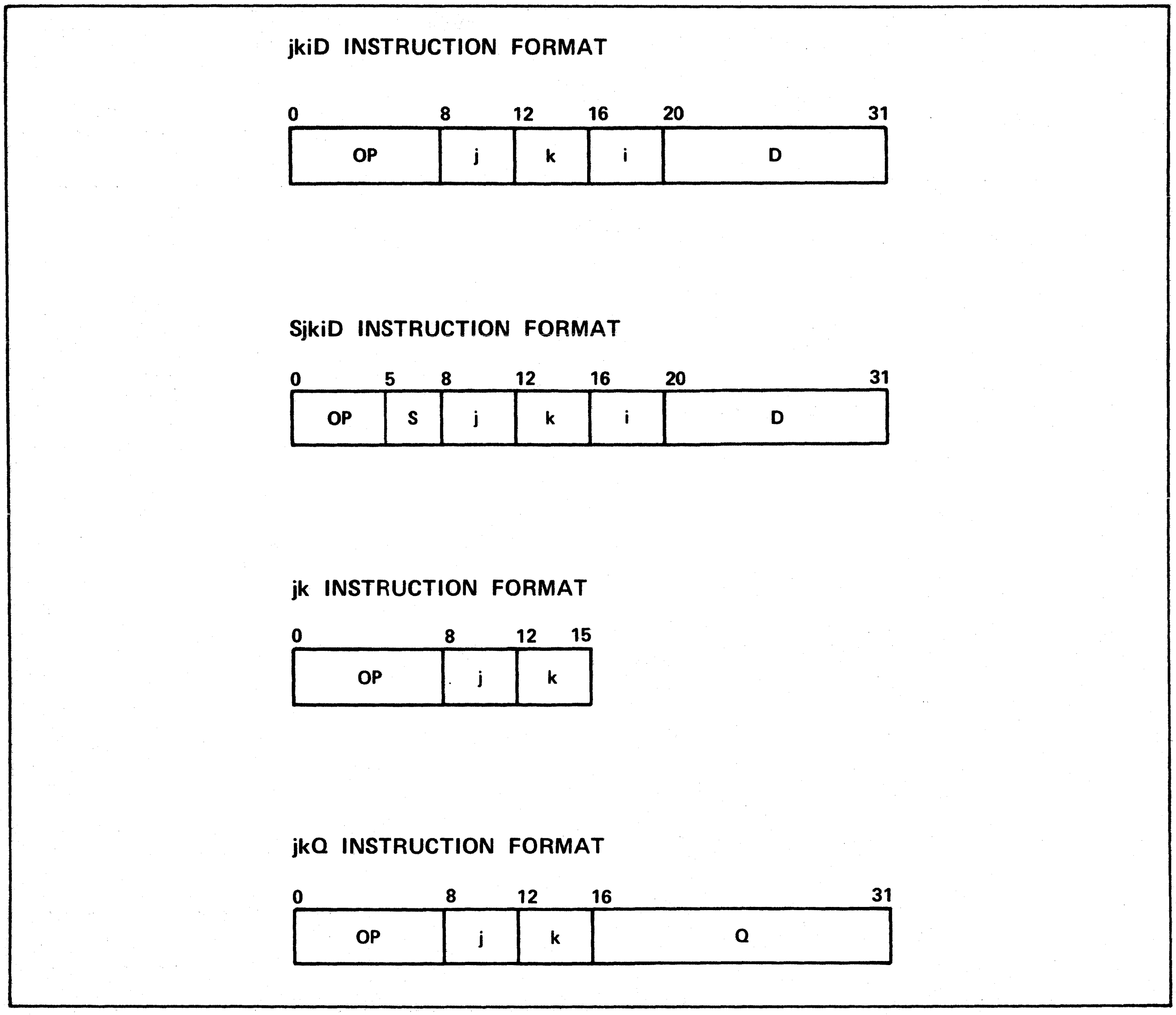


Figure 10-2. General Instruction Formats

LOAD/STORE BYTES, WORDS, BITS

These instructions load data into and store data from X Registers. Points to be noted are:

1) If X0 is specified as a register for index arithmetic, it is interpreted as no index.

2) If the information to be loaded is less than a full register (64-bits) then the information is loaded right justified, zero fill.

3) If a 64-bit word is loaded or stored the data must be located on a full word boundary or an address specification error will be flagged.

4) The address of the data to be loaded or stored is the address of the leftmost byte in memory for load/store byte.

LOAD/STORE A REGISTER

A Registers are used to address operands to be fetched from or stored in central memory. When these operations occur a security check is made to ensure that the privacy of the data is maintained. Part of that security check consists of ensuring that data is not accessed outside a segment´s ring bracket. The ring number in the PVA held in an A Register is used for this purpose. It is possible to change a ring number in a PVA held in central memory. However, that PVA can only be used via an A Register, hence when an A Register is loaded, the processor ensures that a ring number smaller than that permitted is not entered in the register. It accomplishes this by selecting the largest ring number from:

(a) The six bytes addressed in central memory.

(b) The current Aj Register.

(c) The SDE.R1 pointed to by the segment number in Aj Register.

and inserting that into the PVA.RN of the destination register (figure 10-3).

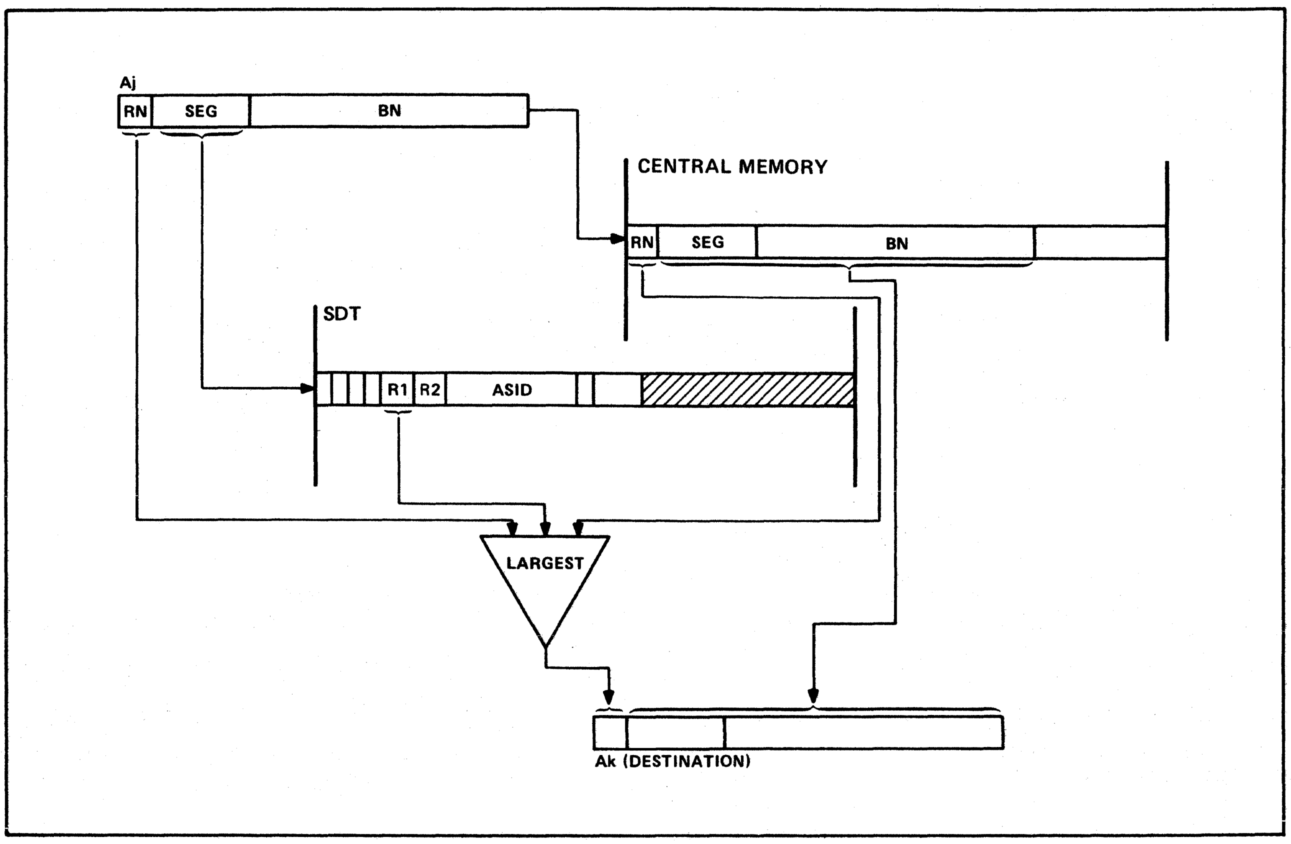


Figure 10-3. A Register Ring Voting

The importance of the ring number assignment is discussed, in depth, in the section on security. In that section, software conventions are also described which are necessary if the integrity of the system is to be maintained.

LOAD/STORE MULTIPLE REGISTERS

These instructions permit the simultaneous loading and storing of a set of A Registers and a set of X Registers. Points to note are:

- One X Register (X0) is lost to the process itself. It holds the designations of the register sets to be stored or loaded.

- Registers to be saved or restored must lie on a full-word boundary in central memory. Failure to specify a full-word address will result in an address specification error.

- A single set of contiguous A Registers, and a single set of contiguous X Registers may be saved or loaded by these instruction.

- When a set of A Registers is loaded, ring number validation occurs as described in the section on single A Register loading.

- If a single A Register or a single X Register is to be loaded/stored, then the start and end designators must be equal.

- If no A Register or no X Register is to be loaded/stored, then the starting designator must be set greater than the end designator.

INTEGER ARITHMETIC

Integer arithmetic is fairly standard. A full set of operations is provided for both the 32-bit integers and the 64-bit integers.

- Integer arithmetic is 2's complement arithmetic. Hence (-X .NE. NOT X), minus zero (-0) does NOT exist, and the magnitude of the largest negative number is one greater than the magnitude of the largest positive number.

- The major difference between the 32-bit operands and the 64-bit operands is the point at which overflow is detected.

- No rounding occurs on any of the instructions.

- The branch tests for less than and less than or equal to are generated by switching the operands in the greater than and greater than or equal to tests.

- The results of an integer compare are placed in bits 32 and 33 of register X1-Right. These bit settings are 00, 01 and 11 for Xj equal to, greater than and less than Xk respectively (figure 10-4). The remainder of X1-Right is cleared, hence a nonzero branch on X1-Right detects a not equal result.

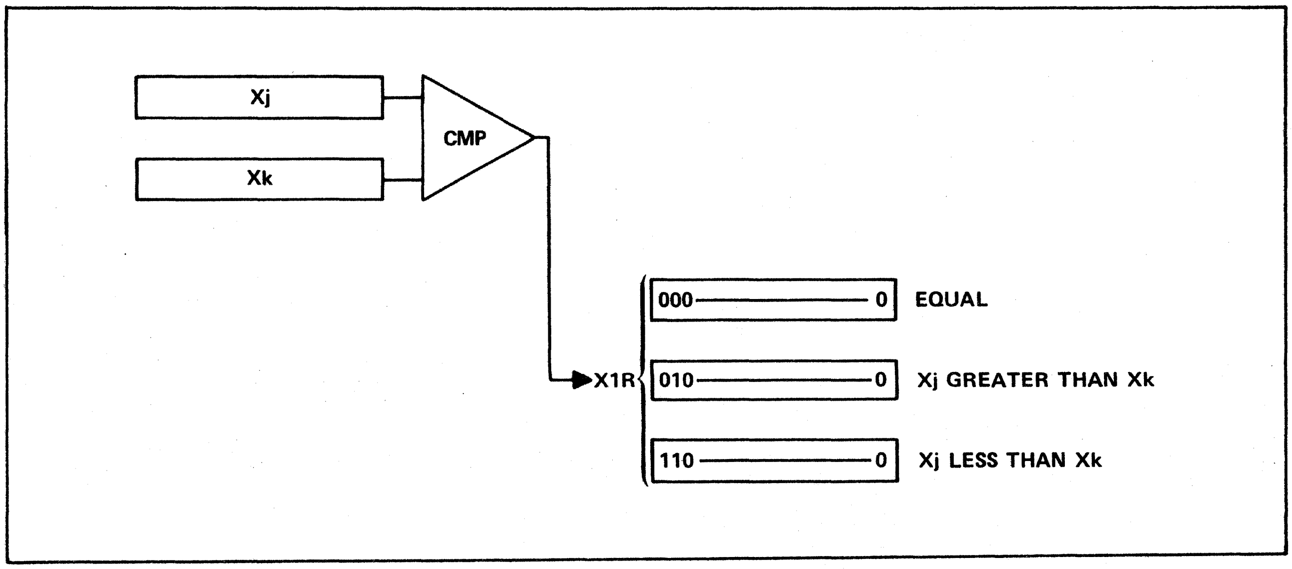


Figure 10-4. Results of Integer Compare

- Register X0 (or X0-Right) cannot be tested explicitly by a branch or compare instruction, since the use of this register by any of these instructions is interpreted as zero. The designation X0 is used to test other registers for a nonzero or zero value. If it is necessary to test X0 it must first be copied to another register.

- The range of 64-bit operations is provided to satisfy the requirements of FORTRAN, and other processors dealing with large integer values. In particular, numerical analysis, commonly conducted in unnormalized floating point arithmetic on CYBER 170 and other machines, is expected to be performed using this arithmetic. 32-bit integers are designed for use in index and address arithmetic, the overflow condition arising at the upper byte number address within a segment.

- All branches must be to a parcel boundary or an address specification error is detected.

BRANCH INSTRUCTIONS

Conditional with Increment

This instruction (figure 10-5), which operates on 64-bit operands is intended for use by FORTRAN in DO loop compilations. Points to note are:

- Whenever the branch is taken, Xk is incremented by one.

- If Xj is specified as X0 then all zeros is assumed by the hardware. The intent here is to use a negative index to count through zero.

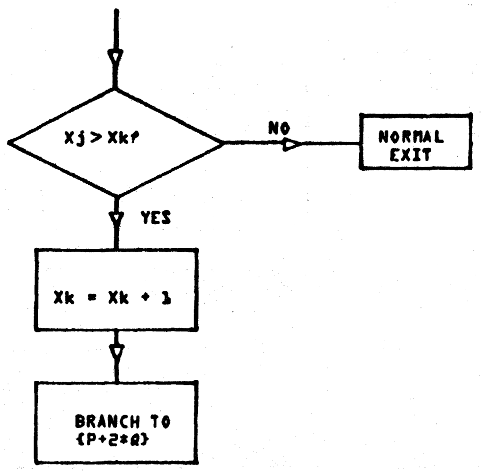


Figure 10-5. Conditional Branch with Increment

Conditional, Ak

The purpose of this instruction is to compare two A Registers (figure 10-6). However, since A Registers contain only PVA´s, and PVA´s have three components, special actions are required. Points to note are:

- The ring numbers of the PVA´s form no part of the comparison.
- Segment number fields (SEG) are compared strictly on an equality/nonequality basis.
- Byte numbers (BN) are compared as signed 32-bit, 2´s complement integers.

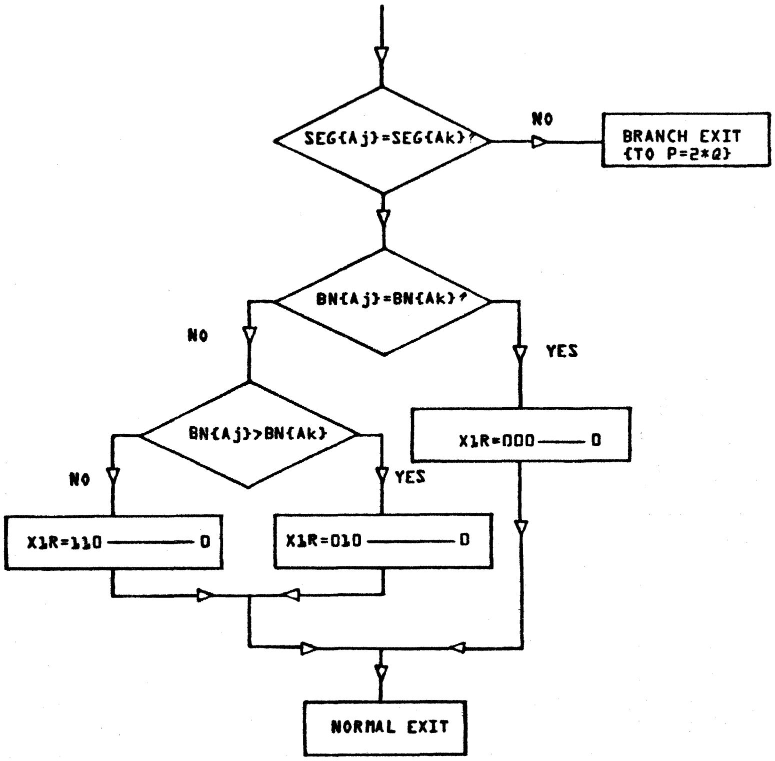


Figure 10-6. Branch Conditional, Ak

Unconditional Branch, Indexed

The reach of this instruction is controlled by the index value in Xk-Right. It is expected that the 24-bit operand, enter instruction will be used in conjunction with this branch.

Unconditional Branch, (A) Indexed

The branches discussed so far are all branches made relative to the current value of the P Register. The branches occur within a segment and no checking on the privacy has to be made when the branch is taken since the segment is the basis of the privacy in the system. One instruction, the Intersegment Branch (figure 10-7), is provided to enable branches to be taken directly from one segment to another.

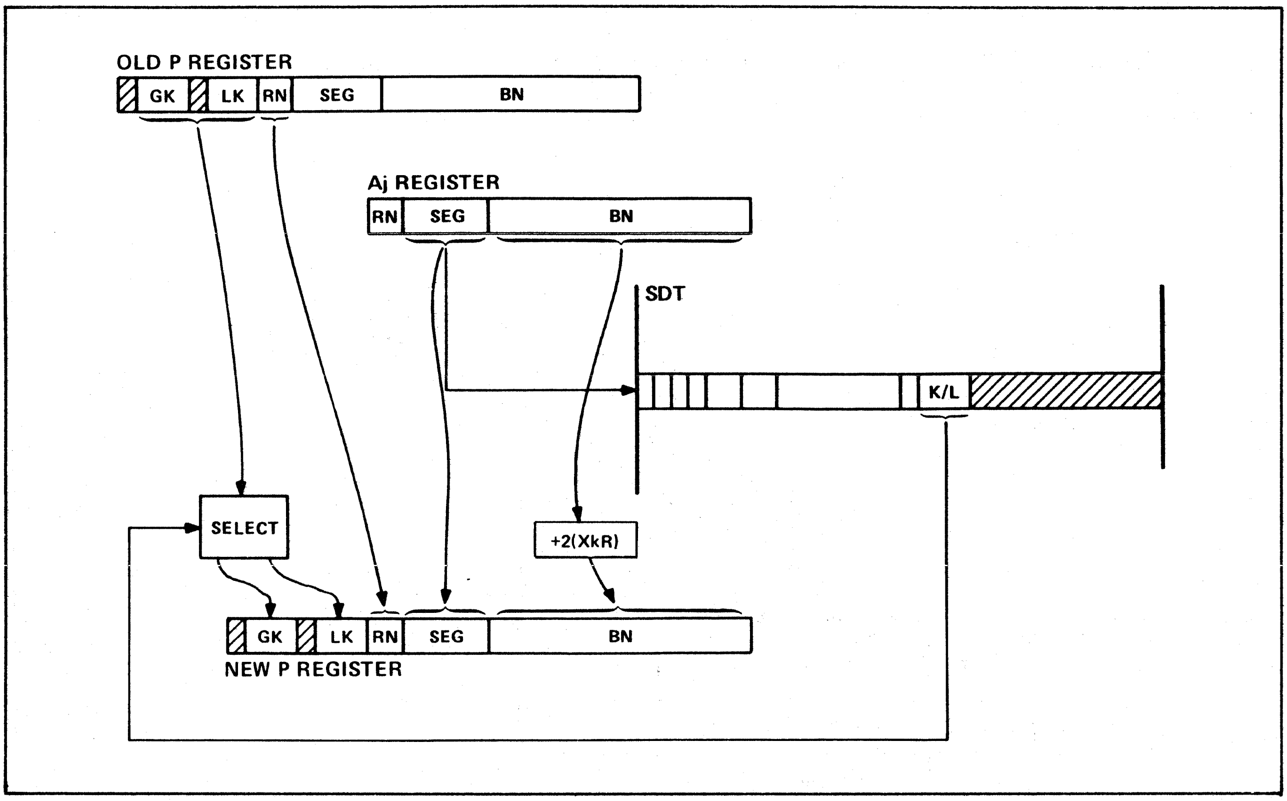


Figure 10-7. Intersegment Branch

The following actions occur in the execution of this instruction (figure 10-8).

1) The P Register byte number is set to the byte number of Aj plus an index (2*Xk-Right).

2) The P Register segment number is set to the Aj segment number.

3) The P Register ring number is not changed.

4) The P Register global keys are set as follows:

Old P-Reg	Segment Descriptor	New P-Reg
G-Key	(Aj SEG) G-Key	G-Key
0	0	0
0	K	K
K	0	K
K	K	K

The new P Register local key is taken from the segment descriptor local lock.

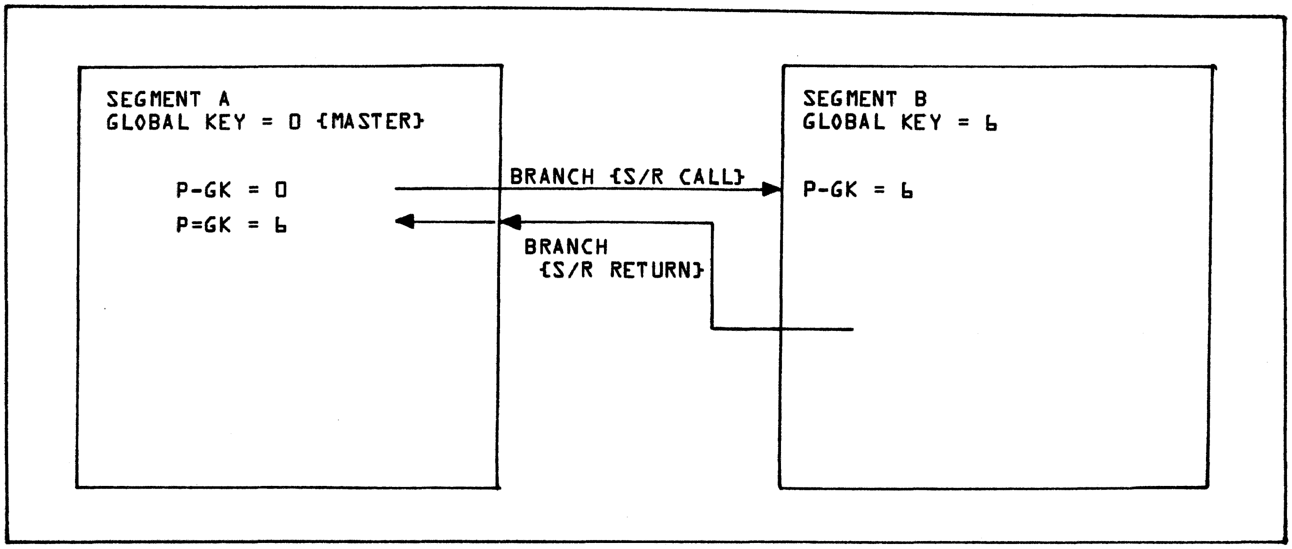


Figure 10-8. Intersegment Branch - Global Key Settings

This instruction can be used as a short subroutine call. In general, the code for CYBER 180 is collected into a series of pure procedures (nonmodifiable code) which may be entered recursively. Special CALL and RETURN instructions are provided for the purpose. However, since these instructions require considerable access validation, and stack processing (retention of the environment) the overhead is relatively high. To circumvent this overhead for short general purpose subroutines (that is, functions), the intersegment branch can be used. When used in this manner, the return address must be loaded into an A Register prior to the branch. However, care must be exercised. The global key of the new P Register is always set to the lower privilege of the two segments involved. Hence, if an intersegment branch is made to a segment with less privilege than the current one, when a return is made (via another intersegment branch) the original P Register (for the original segment) will have a global key with less privilege than it started out with (figure 10-8). Local keys will not exhibit this characteristic. This instruction must work in this manner, since a user can never increase his privileges without the express action of the operating system. Since, privilege is inviolate from the hardware point of view, care must be exercised.

In practice, this situation should never arise due to the intended usage of global Key/Locks and the associated software conventions. Refer to section on Key/Lock mechanism. Hence, procedure calls within the same ring between mutually suspicious subsystems are not permitted, and calls inter-ring must use the CALL/RETURN mechanism.

One last point, the next instruction fetch is included in this instruction validation procedure. Hence, if the segment being branched to does not have execute access, or its ring bracket is inside that of the old P Register ring number, then an access violation will result.

COPY INSTRUCTIONS

These instructions are straightforward. Copy X to X, A to A, X-Right to X-Right, A to X and X to A. The only point of note concerns the Copy X to A. Here again the question of privacy arises. More privilege must not be granted via the ring number portion of the PVA than is permitted. To prevent this the hardware enters the larger of the ring numbers found in the P Register and the X Register into the destination A Register (figure 10-9).

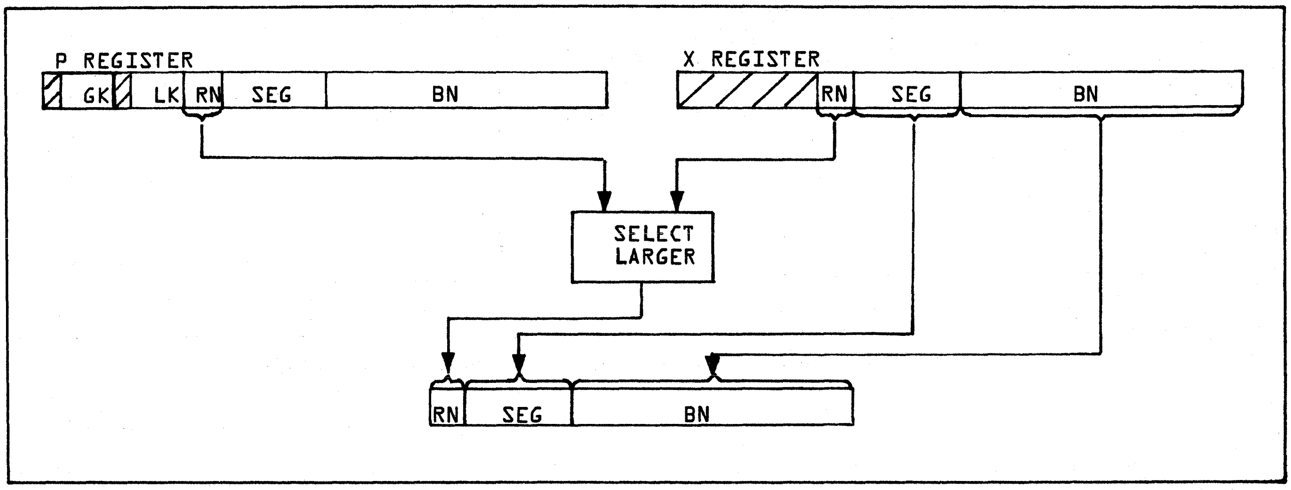


Figure 10-9. Copy X to A Operation

ADDRESS ARITHMETIC

These instructions modify the value of the PVA held in an A Register. The ring number field is never changed and, in general, the segment number is replaced and the byte number modified. Points to note are:

- The Copy P with Indexing and Displacement will copy the P Register, less its keys and ring number into an A Register if the X Register index specified is X0 and the displacement is zero.

- The Copy A with Displacement, Modulo is intended to modify an A Register and force the resulting byte number field to a byte parcel, half-word or full-word boundary (figures 10-10 and 10-11). This is done by truncating the appropriate number of bits in the least significant portion of the BN field. The most common usage is expected to be to force to the next boundary. In this case the D field of the instruction would be set to the complement of the j-field. For example, to force an address to the next full-word boundary D would be 7 and j zero. In this instruction the D field is a positive integer constant. It is zero extended, not sign extended to the left when it is added to the BN field.

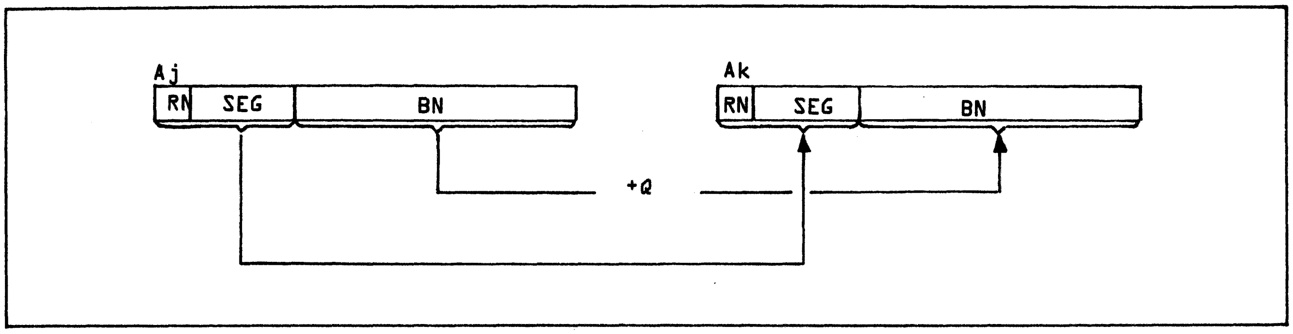


Figure 10-10. Address Increment

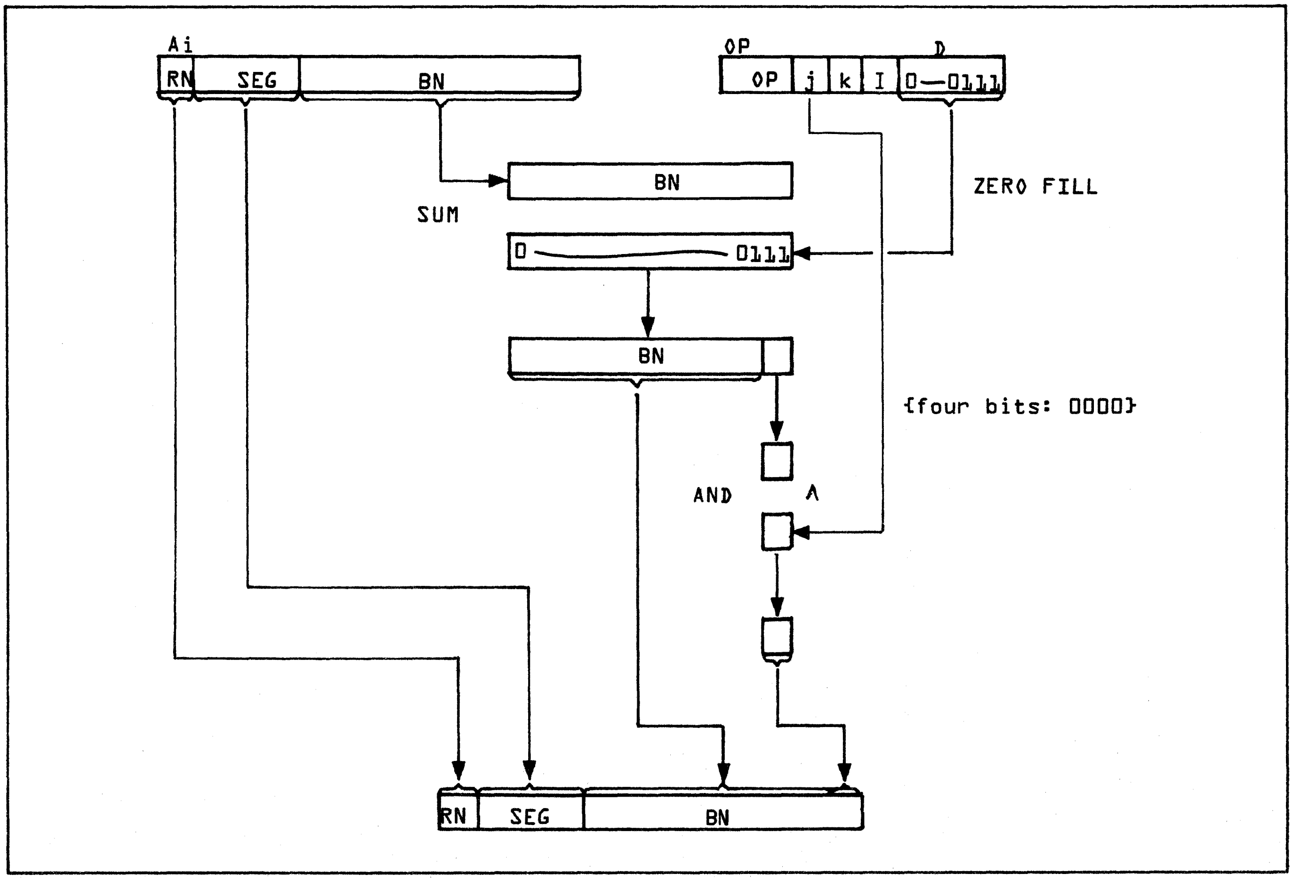


Figure 10-11. Address Increment, Modulo

ENTER INSTRUCTIONS

These instructions, in general, operate on 64-bit X Registers. Three instructions enter positive or negative 4-bit values obtained from the j-field of the instruction, and enter a 16-bit signed integer into the 64-bit register.

Four other instructions deal with entering values into X0 and X1. The first two instructions enter 8-bit logical quantitites into X0 and X1 respectively. The third instruction enters a 24-bit quantity into X1. The value entered is a 24-bit signed integer and it is intended that this instruction be used in conjunction with the unconditional indexed branch instruction to extend the reach of the branch. The range of the 24-bit integer is:

$$-2^{23} \le m \le (2^{23}-1)$$

and since the branch works on a parcel boundary (indexed by 2*X1-Right) the effective reach is:

$$-2^{24} \le \text{reach} \le (2^{24}-2)$$

or roughly +/- 16,000,000 bytes. This is not a full segment but it is an enormous reach. This is important since the reach of the conditional branch instruction is limited to the size of the Q-field. In bytes this is:

$$-2^{16} \le \text{reach} \le (2^{16}-2)$$

or roughly +/- 65,000 bytes, which is not enough for a generalized compilation process. When difficulties are encountered in this area, the conditional branch should be inverted and used in conjunction with an unconditional branch (figure 10-12).

```
             BRREQ      X5,X6,LABEL ----+
                                        |
                becomes :               v Reach > 2^16

          +--BRRNE      X5,X6,LAB1
          |  ENTC       X1,(LABEL-*)/2
          |  BRREL      X1       ----+
     LAB1 +-> --                     |
              --                     v Long Reach > 2^24
```

Figure 10-12. Long Reach Conditional Branch

The fourth instruction enters a 24-bit quantity into X0. It is intended for use in conjunction with the CALL instruction. A descriptor value (in the low-order 16-bits) and an 8-bit quantity can be entered with a single instruction. The 8-bit quantity will be used by software to indicate the number of parameters to be transmitted on the call.

The final enter instruction is the only instruction in the repertoire that pertains to the left half of an X Register exclusively. Three options are provided (figure 10-13).

1) Clear Xk-Left - set to zeros

2) Set Xk-Left - set to ones

3) Set Xk-Left to the sign of Xk-Right.

The last option is expected to be the most commonly used and drags the sign of the Xk-Right, or converts a half-word integer into a full-word integer.

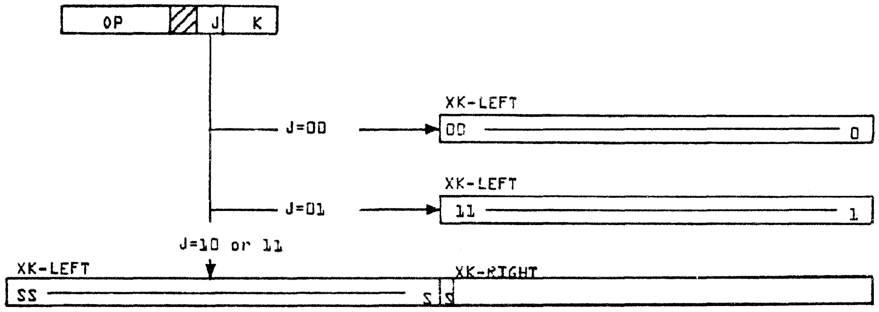


Figure 10-13. Enter Signs

SHIFTS

Circular and End-Off shifts are provided for the full X Register and End-Off shifts for the right half of an X Register. Points to note are:

- Shift counts are formed from the summation of Xi-Right and D. These quantities are signed 2's complement 8-bit integers.
- The sign of the 8-bit shift count determines the direction of the shift. Positive is interpreted as left shift.
- Left shift end-off has zero fill in the right of the register.
- Right shift end-off is with sign-extension. In other words this is an arithmetic shift.
- There is no logical right shift end-off.

The hardware checks the sign of the 8-bit shift count to determine the direction of the shift, and complements the count if it is negative. It then extracts either the low-order 6-bits (for full-word shifts) or the low-order 5-bits (for half-word shifts). Consequently, left shifts range from 0-63 (or 31) and right shifts range from 1-64 (or 32).

LOGICAL OPERATIONS

The logical instructions operate on 64-bit quantities and are standard. Five operations are provided:

1) Logical Sum - OR

2) Logical Difference - EOR

3) Logical Product - AND

4) Logical Complement - NOT

5) Logical Inhibit - AND NOT

The truth tables for these operations are shown in the accompanying diagram.

OR	EOR	AND	NOT	AND NOT
0011	0011	0011	1111	0011
0101	0101	0101	0101	0101
----	----	----	----	----
0111	0110	0001	1010	0010

LOGICAL TRUTH TABLES

BIT STRING OPERATIONS

These instructions pertain to a contiguous string of bits in a 64-bit (full word) X Register. The address and size of the bit string is specified by a 12-bit designator which is interpreted as a starting bit address and a number of bits (length).

```
 0          5 6          11
┌────────────┬────────────┐
|Starting Bit| Length - 1 |
|  Address   |            |
└────────────┴────────────┘
```

The instructions all use a bit string descriptor formed by adding a 12-bit offset (unsigned, positive integer) to the right half of an X Register with no overflow detection. If the sum of the starting bit address and the length is greater than 63 an instruction specification error is flagged. Use of X0 as an index is interpreted as all zeros.

Three instructions are provided:

1) Isolate Bit Mask - Forms a solid mask in a 64-bit X Register (figure 10-14).

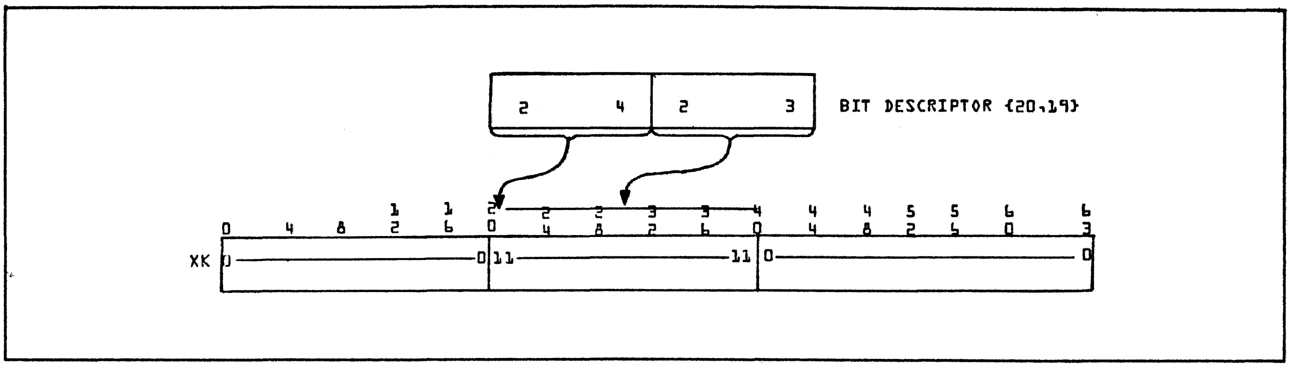


Figure 10-14. Isolate Bit Mask

2) Isolate to Xk - Extracts a string of bits from Xj as specified by the bit string designator and places them in XK right justified zero fill (figure 10-15).

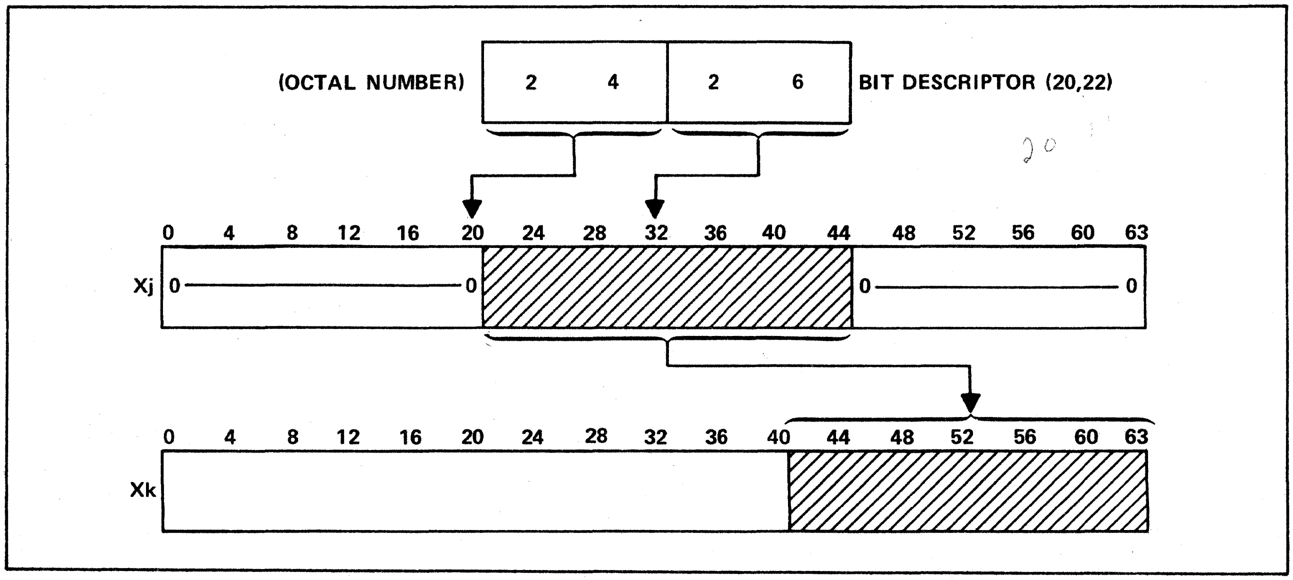


Figure 10-15. Isolate to Xk

3) Insert into Xk - Performs the inverse of the previous instruction, taking the right most bits from Xj as specified by the length field in the bit string descriptor, and inserts them into Xk as per the bit string descriptor. All other bits in Xk are unchanged.

MARK TO BOOLEAN

This instruction is intended to take the result of a compare instruction which is in X1-Right bits 32 and 33, and convert it to a Boolean (True/False) result. Results from all compare instructions are as follows.

Relation	X1-Right		Value
	32	33	
XK = XJ	0	0	0
XK > XJ	0	1	1
XK < XJ	1	1	3

However, relational expressions are frequently encountered with the form:

(A ≠ B)

The Mark to Boolean is designed to produce a Boolean result for any relational expression. It does this by using the j-field of the instruction to define the relational expression being evaluated.

The value of the j-field relates to the value of the X1-Right comparison result. Four values are possible (0,1,2,3) of which three (0,1,3) are used by the compare instructions. Each bit in the j-field points to a value in the X1-Register. For example; 1000 points to value 0, 0100 points to value 1, 1001 points to values 0 and 3 (figure 10-16).

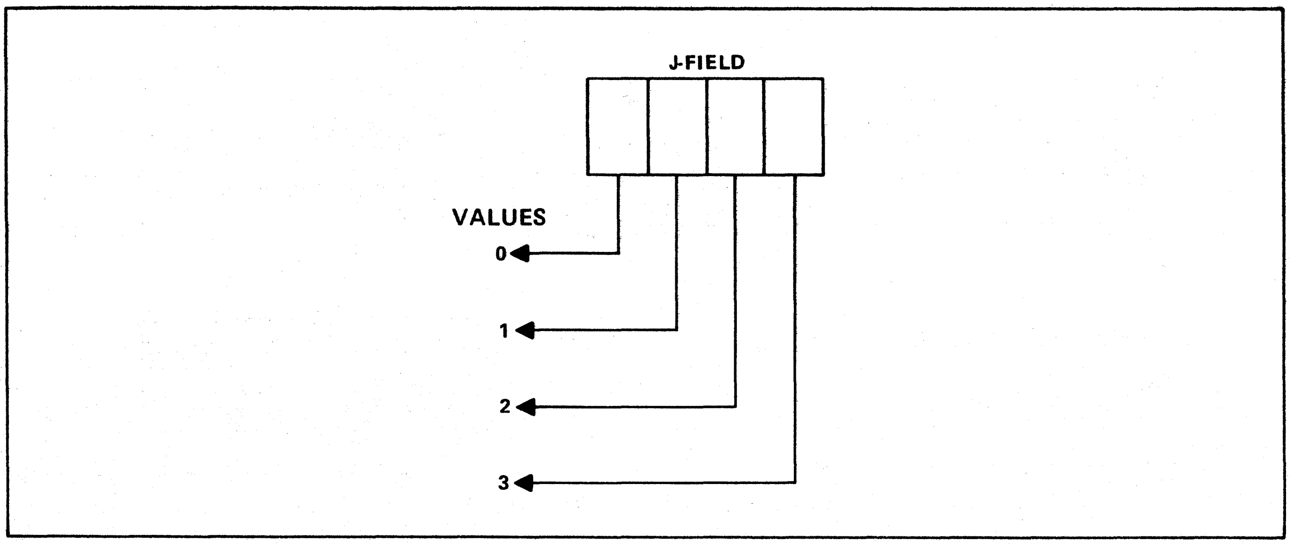


Figure 10-16. Mark to Boolean J-Field Usage

If the value pointed to by the j-field is true, then a true result is set in the XK-Register by setting bit 0 (sign bit) and clearing the remaining 63 bits. Otherwise a false value is indicated by clearing XK. Only six relational values are of interest: <, >, ≠, =, ≤, and ≥. Hence only six values for j are required. These are 1,4,5,8,9 and C for <, >, ≠, =, ≤, and ≥ respectively (figure 10-17).

j-FIELD		TEST	TEST	
HEX	BINARY	(LITERAL)	ACTUAL	REQD
0	0000	NONE	NONE	NO
1	0001	<	<	YES
2	0010	NONE	NONE	NO
3	0011	<	<	NO
4	0100	>	>	YES
5	0101	> & <	≠	YES
6	0110	>	>	NO
7	0111	> & <	≠	NO
8	1000	=	=	YES
9	1001	< & =	≤	YES
A	1010	=	=	NO
B	1011	< & =	≤	NO
C	1100	> &=	≥	YES
D	1101	>,< & =	ALL	NO
E	1110	> & =	≥	NO
F	1111	>, < & =	ALL	NO

Figure 10-17. Mark to Boolean Tests

BDP INSTRUCTIONS

The BDP instructions utilize three instruction formats: jK plus two descriptors, jKiD plus two descriptors and jKiD plus one descriptor (figure 10-18).

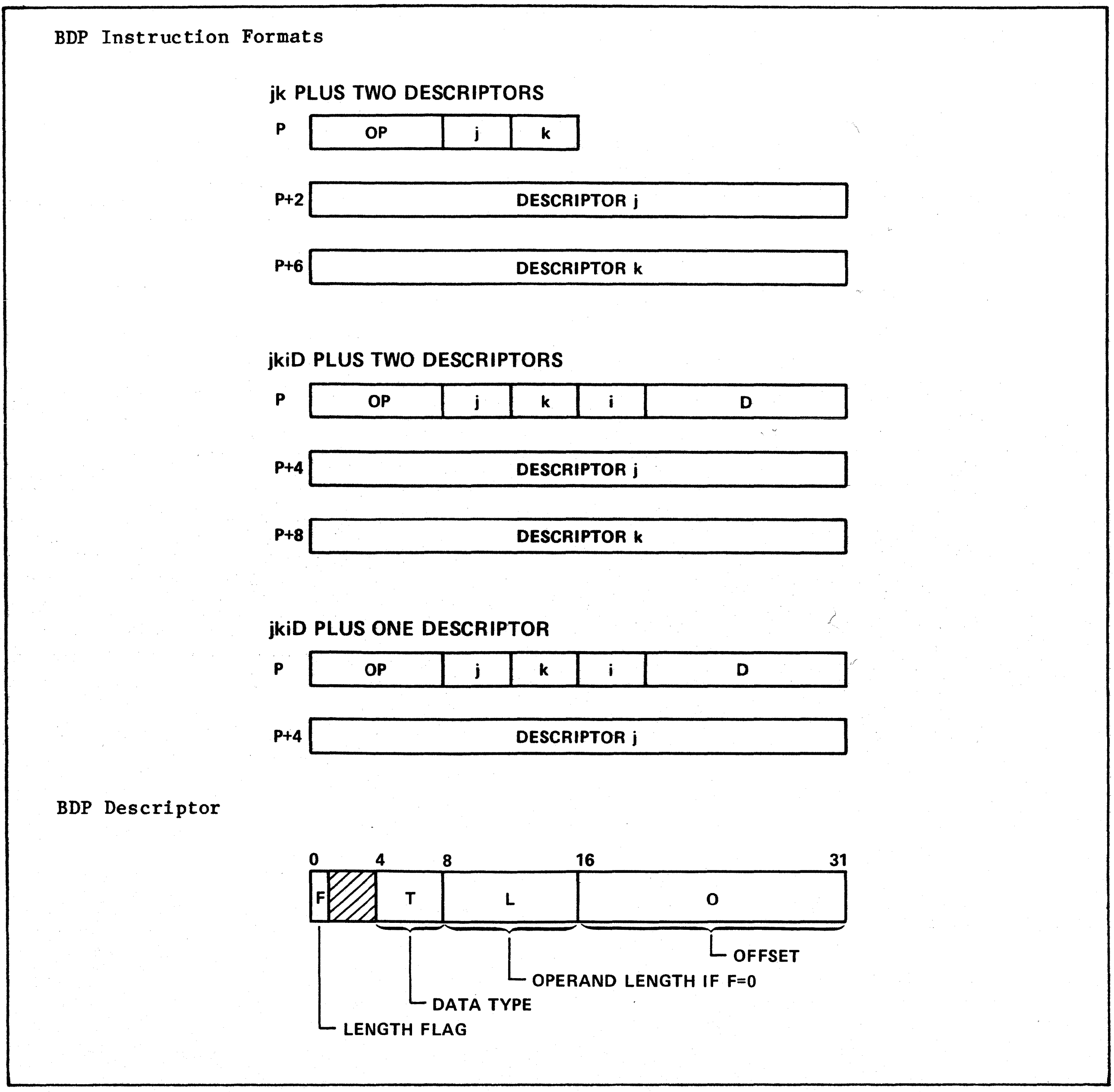


Figure 10-18. BDP Instruction Formats and BDP Descriptor

2byte

Data descriptors are 32-bits long, situated on a parcel boundary, are in the main instruction stream and contain information about the location, size and type of the data. The following points should be noted:

- BDP data is always referenced by the BDP instruction via data descriptors.

- Data descriptors form part of the code stream and since CYBER 180 is designed to operate on pure procedures (no code modification), the data descriptors cannot be modified.

- The length of BDP operands may be specified in the descriptor (F=0), or may be treated as a variable (F=1) located in either X0 or X1 for the j-descriptor and K-descriptor (first and second) respectively. Values of length in the descriptor from 00-FF correspond to lengths of 0-255 bytes. When the length is obtained from an X Register (X0 or X1) bits 55-63 are used values 000-100 corresponding to lengths of 0-256 respectively. Lengths greater than 256 are illegal. This restriction arises from the philosophy of noninterruptible instructions.

- The maximum lengths of operands are a function of the operand type as follows.

Packed Decimal	(Types 0-3,12,13)	- 19 Bytes
Unpacked Decimal	(Types 4-8)	- 38 Bytes
Binary	(Types 10,11,14,15)	- 8 Bytes
Alphanumeric	(Type 9)	- 256 Bytes

- Instructions are typically 2-Address. This means that one of the operands is usually modified by the operation, such as:

  A = A+B

  Since BDP operations function on a memory-to-memory basis that can lead to problems if care is not used.

- When source and destination data fields overlap (other than exactly overlay each other) the BDP instructions result in undefined data.

BDP DATA TYPES

Sixteen data types have been defined for use by the BDP instructions (figure 10-19). They include:

- Four Packed Decimal Types
- Five Unpacked Decimal Types
- One Alphanumeric Type (ASCII)
- Two Binary Types
- Four Translated Types (2 Packed and 2 Binary)

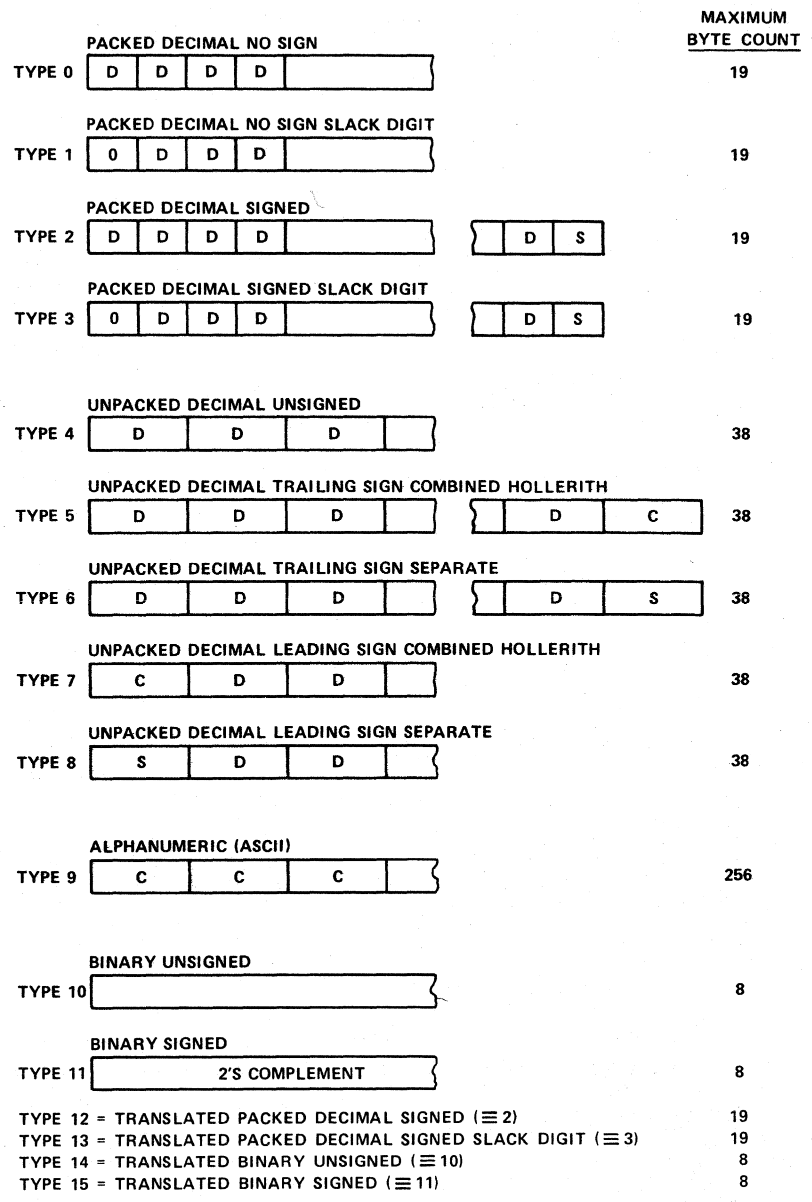

MAXIMUM
BYTE COUNT
PACKED DECIMAL NO SIGN
TYPE 0
D D D D
19
PACKED DECIMAL NO SIGN SLACK DIGIT
TYPE 1
0 D D D
19
PACKED DECIMAL SIGNED
TYPE 2
D D D D
D S
19
PACKED DECIMAL SIGNED SLACK DIGIT
TYPE 3
0 D D D
D S
19
UNPACKED DECIMAL UNSIGNED
TYPE 4
D D D
38
UNPACKED DECIMAL TRAILING SIGN COMBINED HOLLERITH
TYPE 5
D D D
D C
38
UNPACKED DECIMAL TRAILING SIGN SEPARATE
TYPE 6
D D D
D S
38
UNPACKED DECIMAL LEADING SIGN COMBINED HOLLERITH
TYPE 7
C D D
38
UNPACKED DECIMAL LEADING SIGN SEPARATE
TYPE 8
S D D
38
ALPHANUMERIC (ASCII)
TYPE 9
C C C
256
BINARY UNSIGNED
TYPE 10
8
BINARY SIGNED
TYPE 11
2'S COMPLEMENT
8
TYPE 12 = TRANSLATED PACKED DECIMAL SIGNED (≡2)
19
TYPE 13 = TRANSLATED PACKED DECIMAL SIGNED SLACK DIGIT (≡3)
19
TYPE 14 = TRANSLATED BINARY UNSIGNED (≡10)
8
TYPE 15 = TRANSLATED BINARY SIGNED (≡11)
8

Figure 10-19. BDP Data Types

The following points should be noted:

- The only character code recognized by the hardware is ASCII.

- The hardware does not operate on digit boundaries where digits are 4-bit packed decimal quantities. The concept of the slack digit has been invented to accommodate packed decimal quantities with odd numbered lengths. This is strictly an internal data representation.

- Sign conventions are as follows:

Packed Decimal : + HEX A,B,C,E,F - C preferred
- HEX D

Unpacked Decimal : + HEX 2B (ASCII +)
Separate - HEX 2D (ASCII -)

Unpacked Decimal :
Combined

ASCII	HEX	Interpretation	Comments
1-9	31-39	+1 - +9	
A-I	41-49		Preferred
J-R	4A-4F	-1 - -9	
	50-52		
{	7B	+0	Preferred
0	30	+0	
&	26	+0	
}	7D	-0	Preferred
-	2D	-0	

- Both +0 and -0 can occur and are treated as equivalent.

- Operations need not be on the same type operands.

Translated Data Types

There are four translated data types which are:

- Translated Packed Decimal Signed with and without slack digit.

- Translated Binary signed and unsigned.

The purpose of these data types is to be able to handle EBCDIC data via the hardware. Now the only characters recognized within the CYBER 180 system boundaries are ASCII characters. However, outside the system boundaries EBCDIC data is very important. It is the intent on CYBER 180 to translate EBCDIC data character by character as it crosses the system boundary to its ASCII equivalent. (Typically this data will be on magnetic tape, and the translation will occur on the fly in the magnetic tape controller). For the purposes of this translation the incoming data will be translated eight bits at a time regardless of data type. This means that integer values (signed and unsigned) and packed decimal values will be translated to a meaningless jumble. However, the translation algorithm is well defined. CYBER 180 processors recognize these data types, translate them to meaningful data, operate on them, and, if necessary translate the results back to meaningless jumble (figure 10-20).

The only data type not handled by this technique is Packed Decimal Unsigned. Unpacked Decimal Data translates correctly from EBCDIC to its ASCII equivalent.

The translation algorithms are straightforward and are defined in the MIGDS. They will not be restated here.

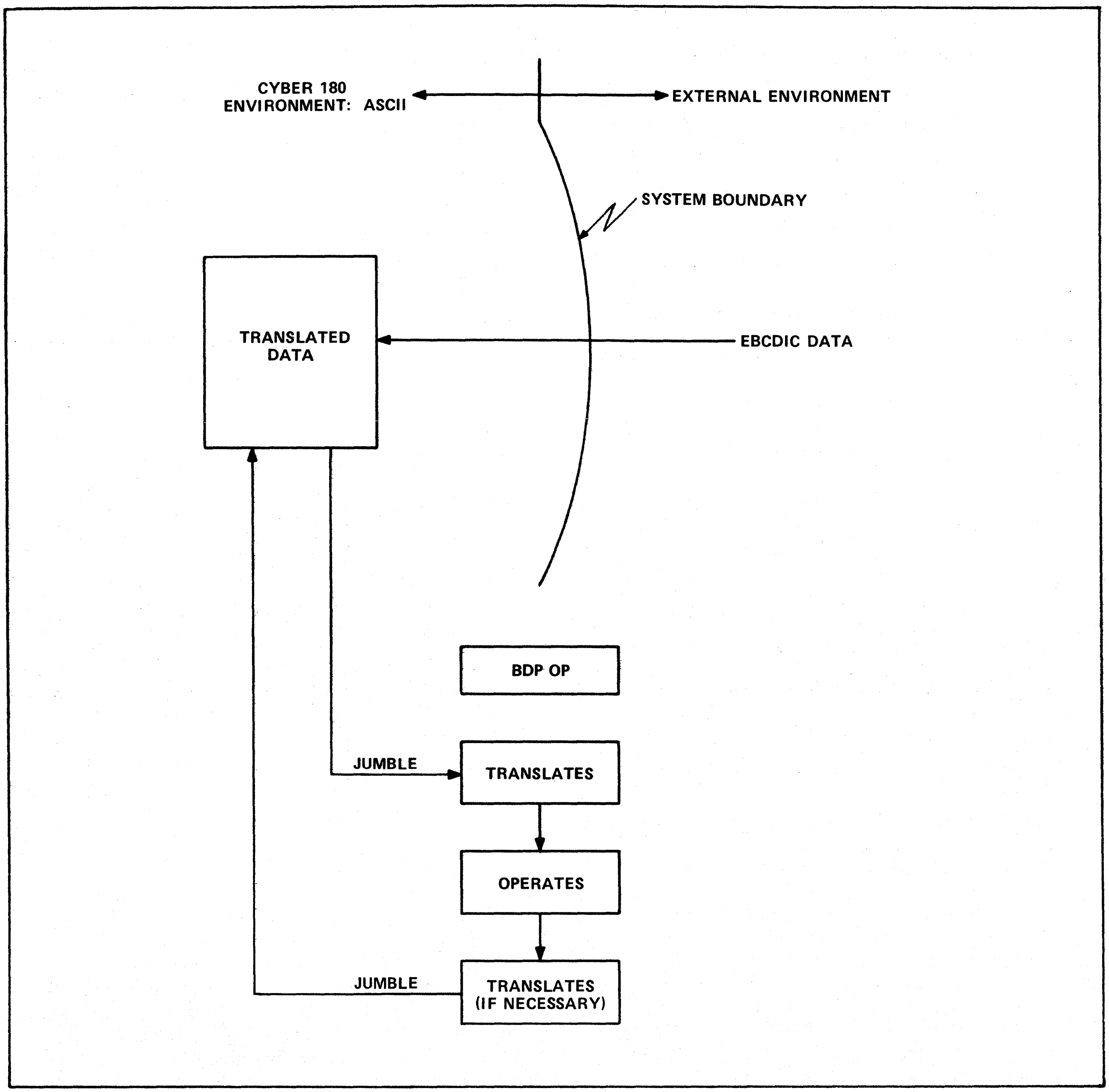
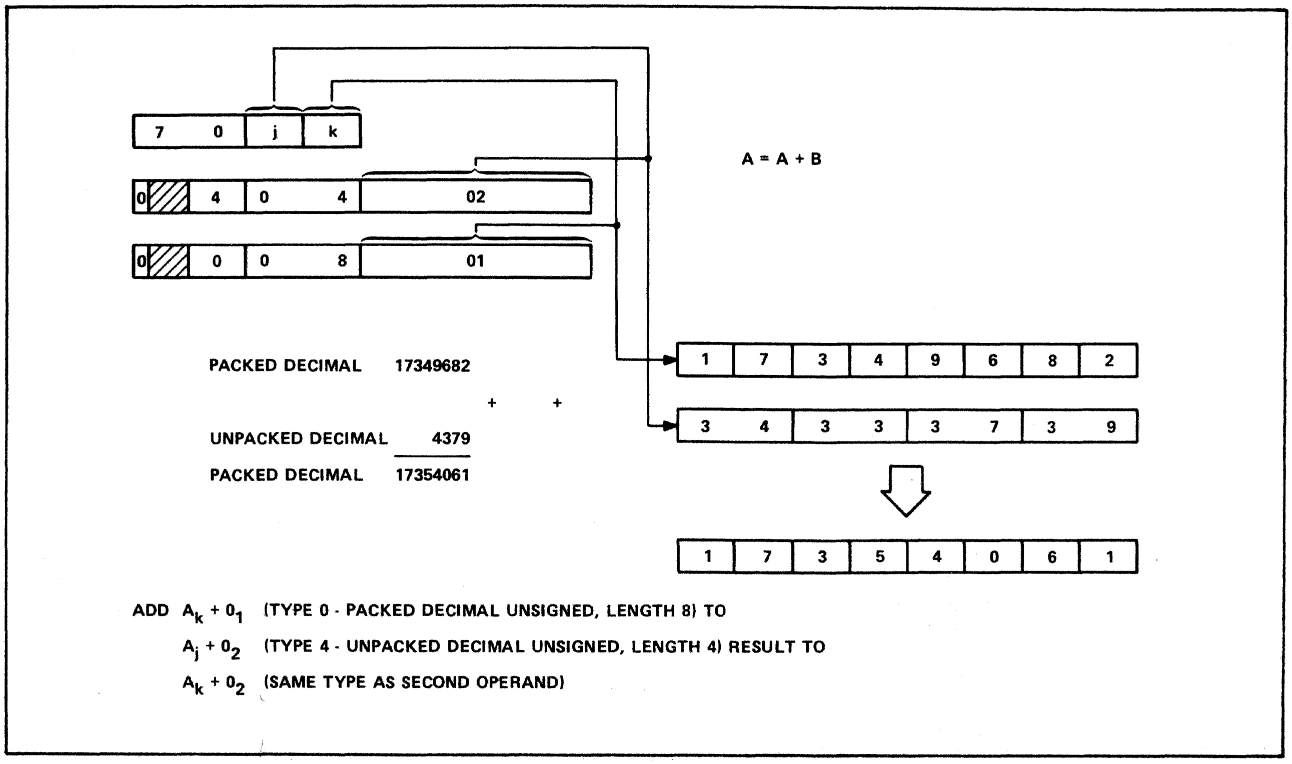


Figure 10-20. Translated Data Types

Figure 10-21 is an example of a decimal add.



ADD $A_k + 0_1$ (TYPE 0 - PACKED DECIMAL UNSIGNED, LENGTH 8) TO
$A_j + 0_2$ (TYPE 4 - UNPACKED DECIMAL UNSIGNED, LENGTH 4) RESULT TO
$A_k + 0_2$ (SAME TYPE AS SECOND OPERAND)

Figure 10-21. Decimal Add Example

NUMERIC OPERATIONS

General points of interest are:

- These operations, in general, work right to left. Cache memory operation takes this into account when loading words into a block in cache memory.
- When the results exceed the length of the destination field an Arithmetic Loss of Significance or an Arithmetic Overflow condition is detected, depending on the instruction.
- Leading zeros are supplied or leading digits truncated to accommodate unequal source and destination field lengths. (With 2-Address instructions, one operand serves as both source and destination).
- Destination operand lengths of zero are treated as NOP´s, except that error sensing will occur unless the source operand field length is also zero.

Arithmetic

Four operations are provided: sum, difference, product and quotient. Points to note are:

- Data types which may be freely mixed are: 0-6, 12 and 13. All other data types must be converted to one of these data types via a Numeric Move prior to the operation.

- If the operands are of unequal length, then the shorter one has leading zeros appended to equalize the lengths. If significant digits cannot be stored because the length of the result is greater than the length of the destination field, the leading digits are truncated and an arithmetic overflow is flagged. The only exception to this rule is a divide by zero, when a divide fault is flagged, and the destination field is unchanged.

Scaling

Two instructions are provided to facilitate multiplying and dividing by ten (scaling). Several points should be noted:

- Scaling is accomplished by shifting the decimal quantity left or right. The shift count is a signed 32-bit integer formed from [Xi]+D. The sign of the shift count gives the direction of the shift, left being positive. The shift count is taken from the least significant 8-bits of the shift count field (the 2's complement of these if a right shift).

- Shifting is end-off, zero fill regardless of direction.

- If the destination field is longer than the source field then zeros are appended to the left of the result.

- When nonzero digits are shifted end-off, or the source field is truncated to fit the destination field an arithmetic loss of significance is flagged.

- Rounding on right shifts is accomplished by adding five to the last digit shifted end-off and propogating the carry.

- Alphanumerics are treated as type 4 (unpacked decimal, unsigned) and cannot be used as a destination field.

- Signs are transferred (not shifted) unless the result is zero in which case the sign is positive. Preferred signs are always used in the destination field.

Move

The following points should be noted:

- Decimal move operates right to left.

- All data types can be freely mixed in this instruction.

- A primary use of this instruction is to convert from one data type to another.

- Alphanumerics are treated as type 4 (unpacked decimal, unsigned).

- If nonzero digits are truncated to fit the destination field, then an arithmetic loss of significance is flagged.

Compare

The following points should be noted:

- The instruction performs an algebraic comparison.
- If the lengths of the operands are unequal, then zeros are appended to the left hand end of the shorter operand prior to the comparison.
- Data types 0-6, 12 and 13 may be freely mixed in this instruction.
- The result of the comparison is returned in X1 in the standard CYBER 180 manner:

  Source = destination: X1-Right = 000---0
  Source > destination: X1-Right = 010---0
  Source < destination: X1-Right = 110---0

BYTE INSTRUCTIONS

These instructions:

- Operate on alphanumeric quantities (type 9).
- Operate left to right.
- Use ASCII blanks as a fill character when unequal operand lengths are encountered.
- Destination field lengths of zero result in a NOP except that exception sensing on nonzero source field lengths will occur.

Compare

Two compare instructions are provided: uncollated and collated. In the former, each character is treated as an 8-bit absolute value. The following points should be noted:

- The operation proceeds from left to right.
- Trailing spaces are appended to equalize field lengths.
- The results of the comparison are left in X1-Right in the standard CYBER 180 manner, that is:

  Source = Destination : X1-Right = 000---0
  Source > Destination : X1-Right = 010---0
  Source < Destination : X1-Right = 110---0

  In addition, the sequence number of the unequal bytes is placed in X0-Right.

- Collated compare proceeds as the uncollated until an inequality is detected, at which stage the unequal characters are translated according to the collate table (at (Aj)+D) and the translated characters compared. If these compare equal, then the comparison continues (figures 10-22 and 10-23).

- The collate table is provided by the user and contains 256 characters. The character to be translated is used as an index into this table. The complete table must always be provided since instructions are always prevalidated. If the complete table did not exist, it is possible to get a page fault during the prevalidation which could lead to a job abort.

- The collated compare instructions can require up to seven pages to be resident in memory before execution starts. This is more than any other instruction. Seven pages come from the instruction itself, two for each operand and two for the collate table, assuming that all these quantities cross page boundaries, which is possible.

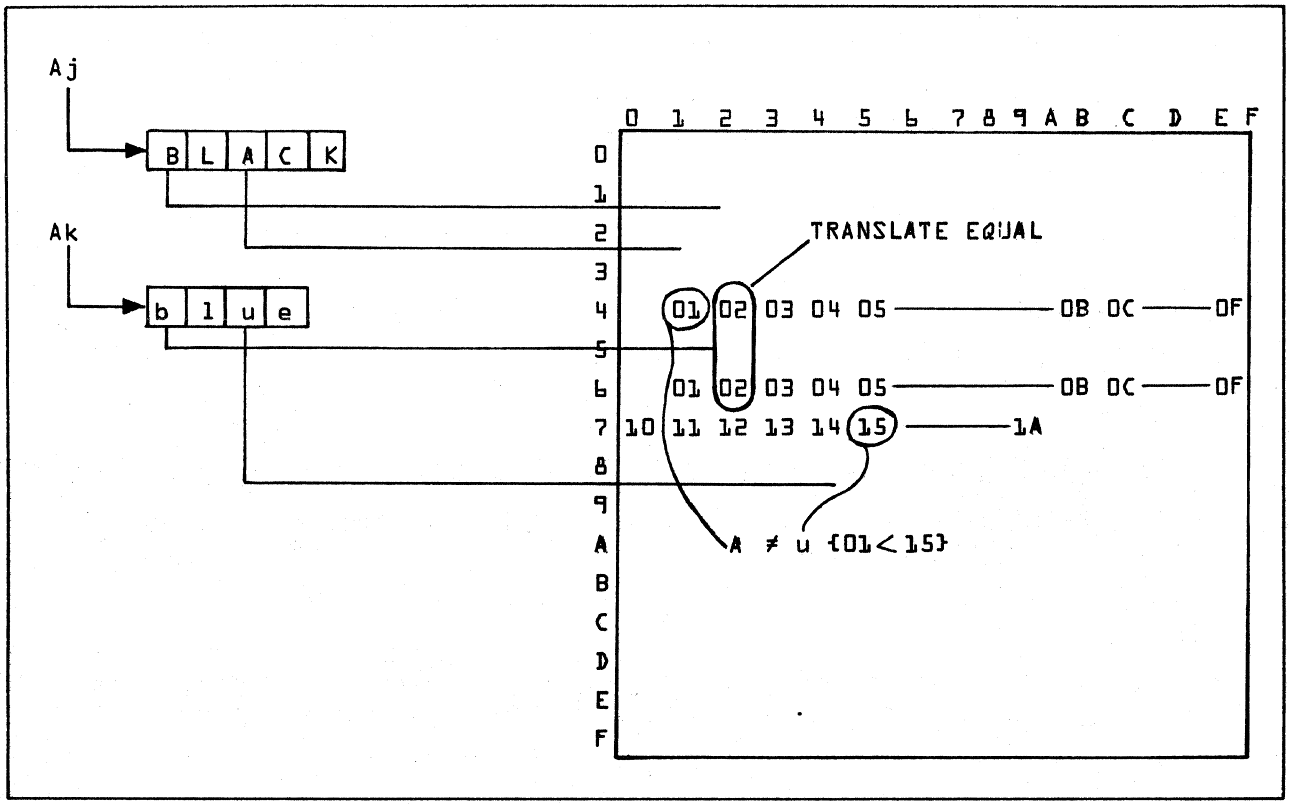


Figure 10-22. Collated Compare Operation

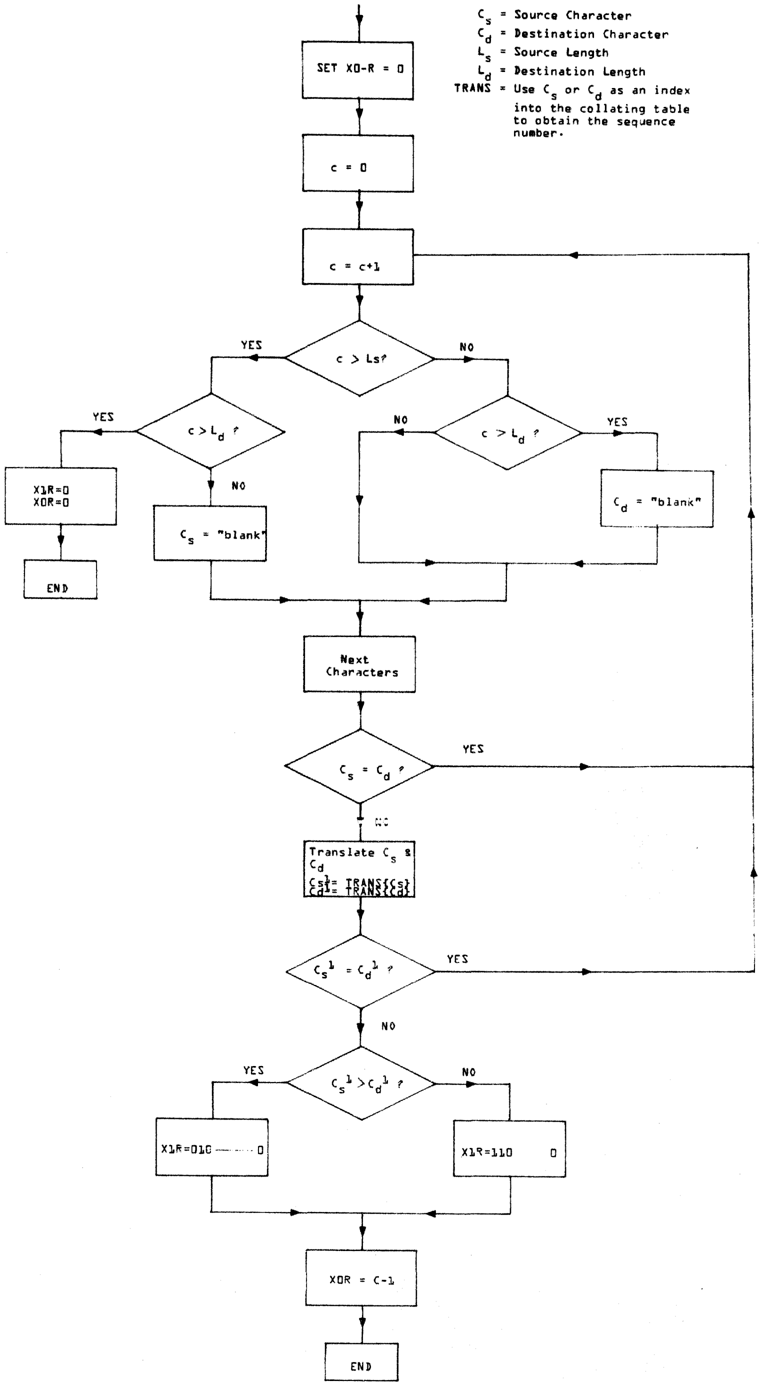


Figure 10-23. Collated Compare Operation Flowchart

Byte Scan

This instruction is intended to examine a character string for the presence or absence of a character or characters. It is similar in function to a repeated equality search. However, its operation is the reverse of an equality search. Instead of comparing a search character with each character in a character string, the character string is passed over a set of characters to determine whether or not there is a match. In fact an actual set of characters is not specified for the scan, but a bit map indicating the presence or absence of a character in the set is established by the user. Characters are treated left to right in the source field and each character is used as an index into the bit map. If the bit is ON (=1), then the operation terminates, otherwise the next character is taken. Points to note are:

- Each character in the source field is treated as an 8-bit absolute value regardless of the data type specified in the descriptor field.

- The sequence number of the character causing the instruction to terminate is returned in X0-Right.

- The character which caused the instruction to terminate is returned in X1-Right.

- If the scan terminates by exhausting all characters in the source field, then X0-Right contains the length of the original byte string, and X1-Right is set to 80000000 (HEX) (sign bit set, remainder cleared).

- If the bit map is complemented, then the operation switches from a Byte Scan While Nonmember (figure 10-24) to a Byte Scan While Member.

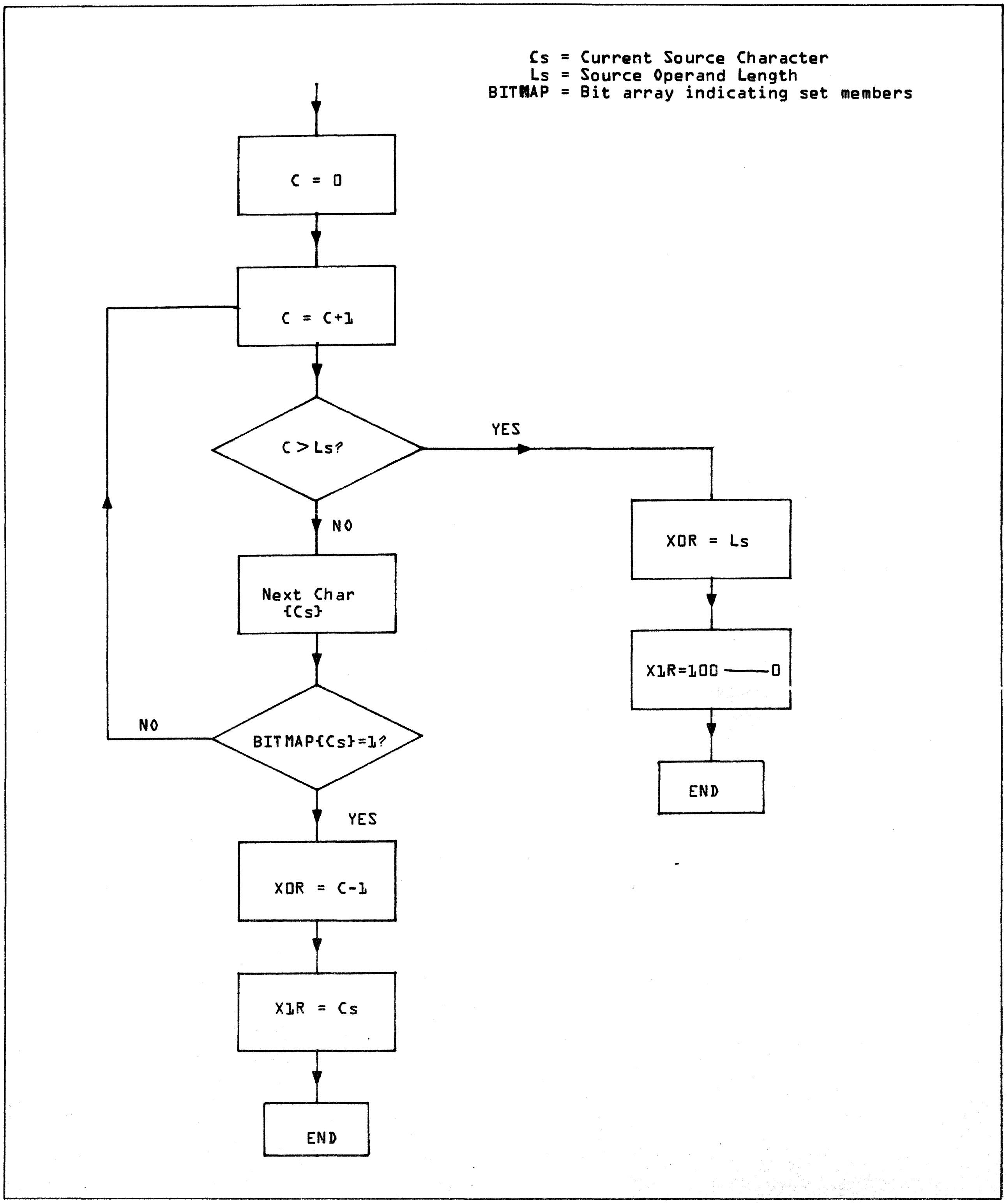


Figure 10-24. Scan While Nonmember Operation

Some examples will help to clarify the operation. Remembering that the ASCII representation in hex, for A-Z = 41-5A, for a-z = 61-7A and blank = 20.

Example 1. Search a character string for the first blank.

Since we are only looking for a blank here, we want the scan to stop on a blank, which means the bit map should have a one in the 20(HEX) position and zeros everywhere else. In other words the bit map set, is a set of one element.

```
      0 1 2 3 4 5 6 7 8 9 A B C D E F      HEX
    ┌─────────────────────────────────┐
    | 0 0-------------------------0 0 |    0000
  1 | 0 0-------------------------0 0 |    0000
  2 | 1 0-------------------------0 0 |    8000
  3 | 0 0-------------------------0 0 |    0000
  4 |               ..                |    0000
  5 |               ..                |    0000
  6 |               ..                |    0000
  7 |               ..                |    0000
  8 |               ..                |    0000
  9 |               ..                |    0000
  A |               ..                |    0000
  B |               ..                |    0000
  C |               ..                |    0000
  D |               ..                |    0000
  E | 0 0-------------------------0 0 |    0000
  F | 0 0-------------------------0 0 |    0000
    └─────────────────────────────────┘
```

Example 2. Search a character string for the first nonblank; discard leading blanks.

This is the converse of the previous example, and the bit map set is the complement of the previous one. That is, ones everywhere except in the 20 (hex) position.

```
      0 1 2 3 4 5 6 7 8 9 A B C D E F          HEX
    ┌─────────────────────────────────┐
  0 | 1 1-------------------------1 1|         FFFF
  1 | 1 1-------------------------1 1|         FFFF
  2 | 0 1-------------------------1 1|         7FFF
  3 | 1 1-------------------------1 1|         FFFF
  4 |               ..               |         FFFF
  5 |               ..               |         FFFF
  6 |               ..               |         FFFF
  7 |               ..               |         FFFF
  8 |               ..               |         FFFF
  9 |               ..               |         FFFF
  A |               ..               |         FFFF
  B |               ..               |         FFFF
  C |               ..               |         FFFF
  D |               ..               |         FFFF
  E | 1 1-------------------------1 1|         FFFF
  F | 1 1-------------------------1 1|         FFFF
    └─────────────────────────────────┘
```

Example 3. Search a character string for the first nonalphabetic, nonblank character. Set bits 41-5A (hex), 61-7A (hex) and 20 (hex) equal to zero and all else ones. The bit map is everything except blank and the alphabetics.

```
      0 1 2 3 4 5 6 7 8 9 A B C D E F          HEX
    ┌─────────────────────────────────┐
  0 | 1 1-------------------------1 1|         FFFF
  1 | 1 1-------------------------1 1|         FFFF
  2 | 0 1-------------------------1 1|         7FFF
  3 | 1 1-------------------------1 1|         FFFF
  4 | 0 0------------------------- 0 0|        0000
  5 | 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1|         001F
  6 | 0 0-------------------------0 0|         0000
  7 | 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1|         001F
  8 | 1 1-------------------------1 1|         FFFF
  9 |               ..               |         FFFF
  A |               ..               |         FFFF
  B |               ..               |         FFFF
  C |               ..               |         FFFF
  D |               ..               |         FFFF
  E | 1 1-------------------------1 1|         FFFF
  F | 1 1-------------------------1 1|         FFFF
    └─────────────────────────────────┘
```

Translate

The translate instruction translates from one 8-bit character representation to a second 8-bit character representation via a translate table. Points to note are:

- Translation occurs left to right. If the source field is longer than the destination field, the rightmost characters are truncated. If the source field is shorter than the destination field, the destination field is filled in its rightmost characters with translated blanks.

- The data type field is ignored. Each 8-bit character is interpreted as an absolute 8-bit quantity.

- Translation occurs exactly as for collated compare. Each character in the source field is treated as an index into a 256 byte translate table.

- All values in the translate table must be supplied if a job is not to be aborted prematurely.

Move Bytes

This instruction simply performs a memory-to-memory move of a string of characters. Points to note are:

- The operation proceeds left to right.

- Unequal source and destination field lengths are accommodated by truncating trailing characters, or filling trailing characters with blanks (20(hex)).

- If the source and destination fields overlap in any way other than exactly, then the results will be undefined.

Edit

The EDIT instruction is intended for use primarily in COBOL compilations. It takes a source data field and formats it for subsequent display. It is a very complex instruction, the execution of which is controlled by an edit mask (located at (Ai+D)) which consists of a sequence of micro-operations (figure 10-25).

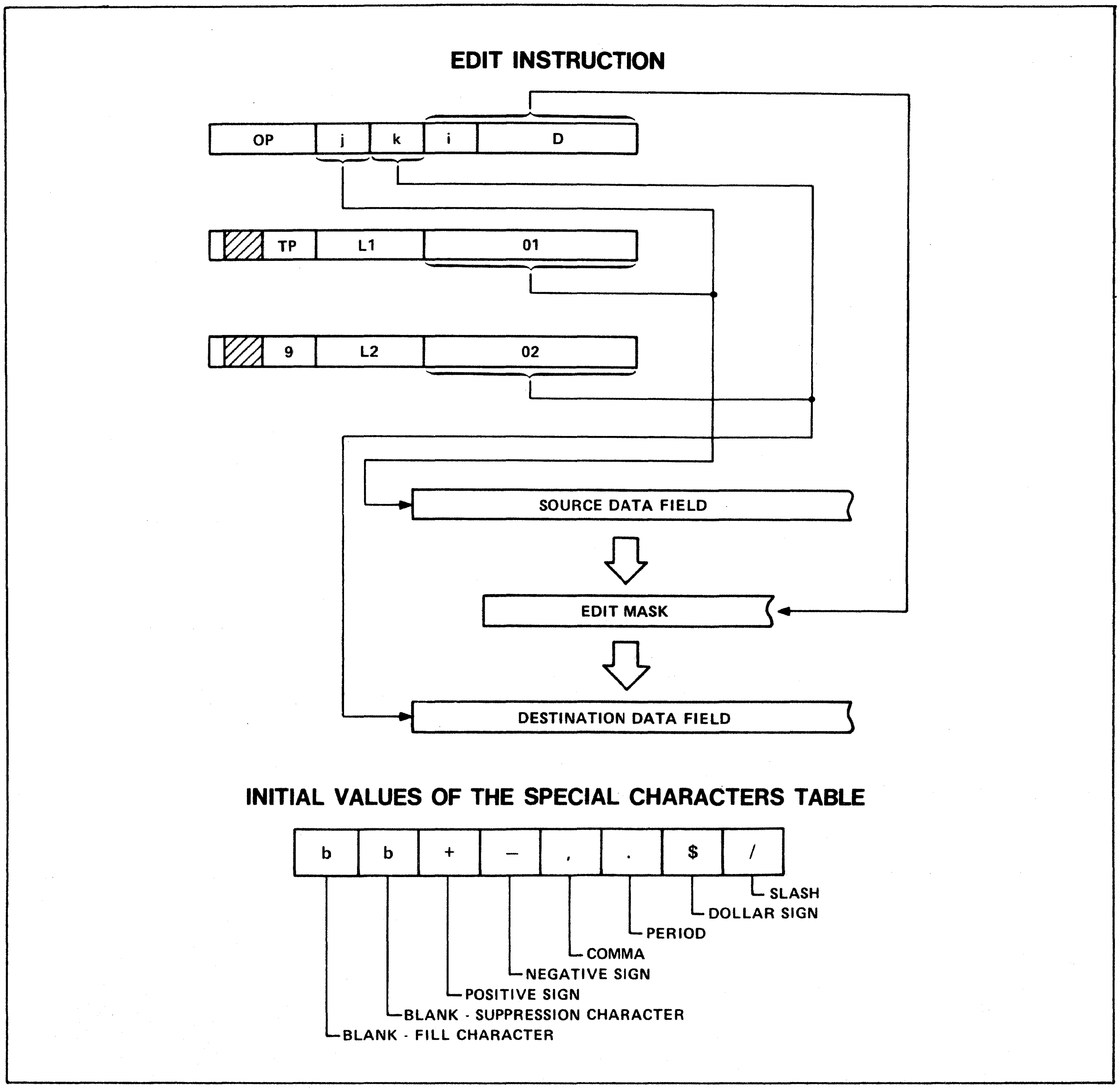


Figure 10-25. Edit Instruction

Points to note are:

- The destination field is always alphanumeric (type 9).
- With the exception of binary data (types 10,11,14 & 15) all data types are acceptable in the source field.
- A number of special tables and flags are made available to the edit instruction by the hardware. These are described in the key to symbols for flow-charts.

- The source field sign is skipped by micro-ops addressing the source field. For combined signs, the numeric value only is interpreted.

- If the length of the edit mask is zero or one, then the instruction results in no operation.

The Edit Mask

The edit mask consists of a string of 8-bit bytes each containing a micro-operation code (MOP) and a specification value (SV) (figure 10-26). The first byte of the edit mask contains the length of the edit mask including itself. Hence, up to 254 MOP´s may be specified per edit instruction.

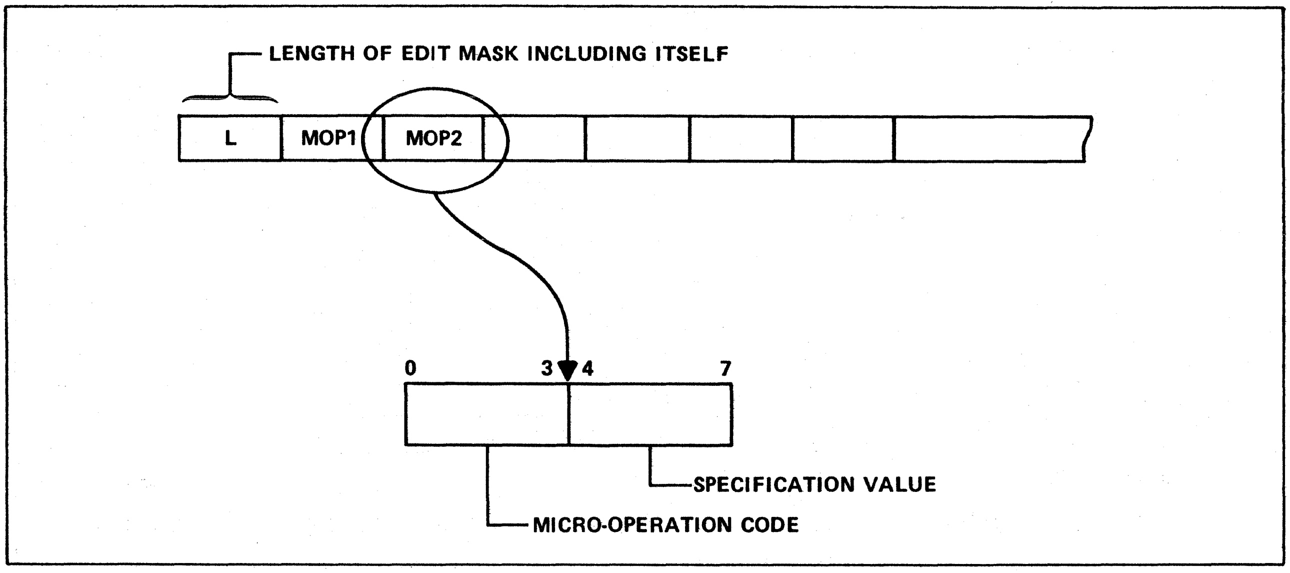


Figure 10-26. Edit Mask

General Notes on EDIT:

- Edit is a complex instruction. It is really a set of instructions which are built up in the edit mask, much like an instruction stack. Each micro-operation directs some control over the destination field, so that source data can be formatted in a generalized manner.

- Edit is best understood by working through examples. In appendix C of the MIGDS there are numerous examples of edit masks and their affect on a variety of source fields. These examples will not be repeated here but are recommended reading for anyone wishing to understand this instruction fully.

Figures 10-27 through 10-43 describe the operation of the edit instruction itself, and the operation of each micro-op.

Key to Symbols:

i Index for the source field in bytes for data type 9 and in digits for all other data types (skipping slack digits on data types 1, 3,and 13 and skipping separate sign on data types 2, 3, 6, 8 and 12).

j Index for destination field, initialized to 0.

k Index for mask, initialized to 0.

SCi Source character addressed by base of source field indexed by i.

SDi Source digit addressed by base of source field indexed by i.

DCj Destination byte addressed by base of destination field indexed by j.

MCk Mask byte addressed by base of mask field indexed by k.

ES End supression toggle (initialized False and then set True when end supression occurs).

ZF Zero field toggle (initialized True and then set False when nonzero source digit is processed).

SN Sign toggle (initialized False and then set True if source field is negative).

SCT Special character table (initialized by hardware as indicated in edit overview diagram, and may be read/written by certain MOPs).

SCT_n The (n+1)th entry in the SCT (n must be 0-7).

SV Specification value.

SM Symbol, which is a string of 0-15 characters which may be created and inserted into the destination field. It is initialized to zero length and once used must be recreated before further use.

SM_c The cth symbol character.

LS Length of source field in digits (or in characters for type 9).

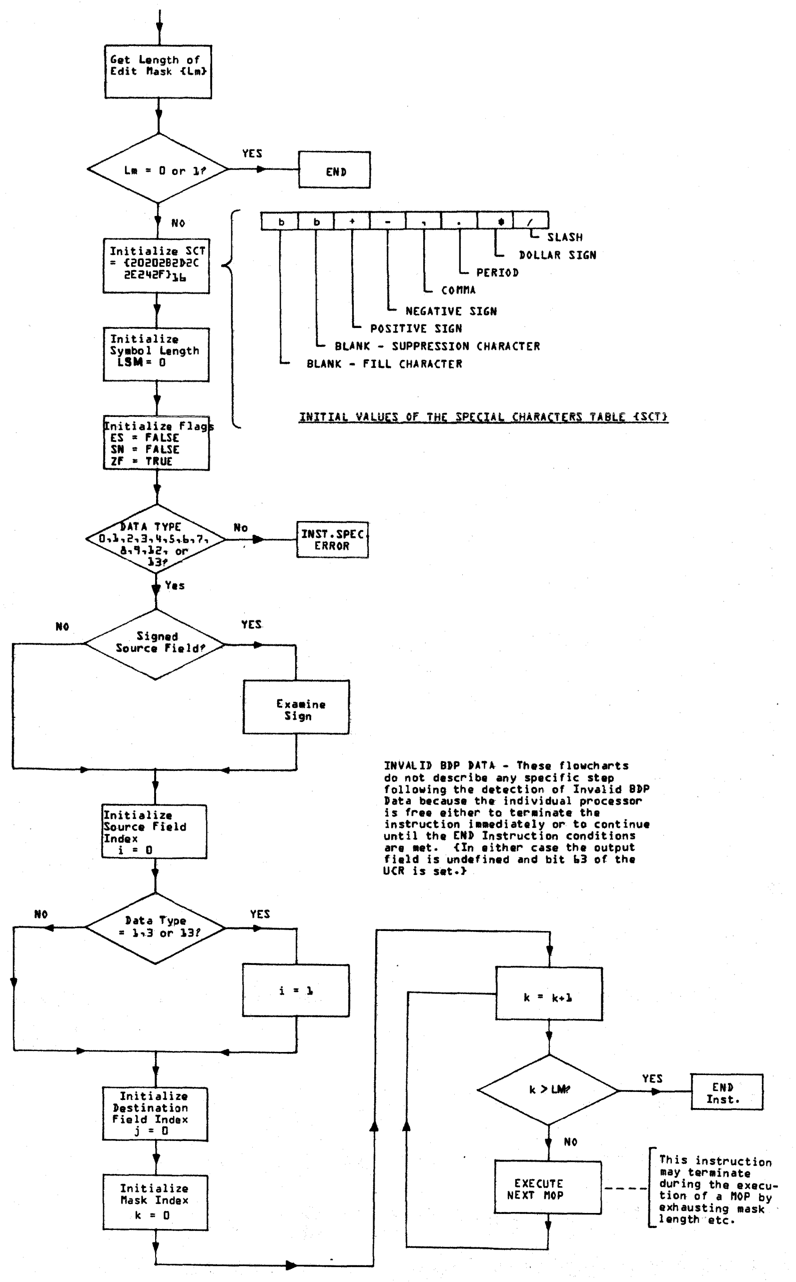
LM Length of edit mask in bytes.

LD Length of destination of field in bytes.

LSM Length of the symbol in bytes, initialized to 0.

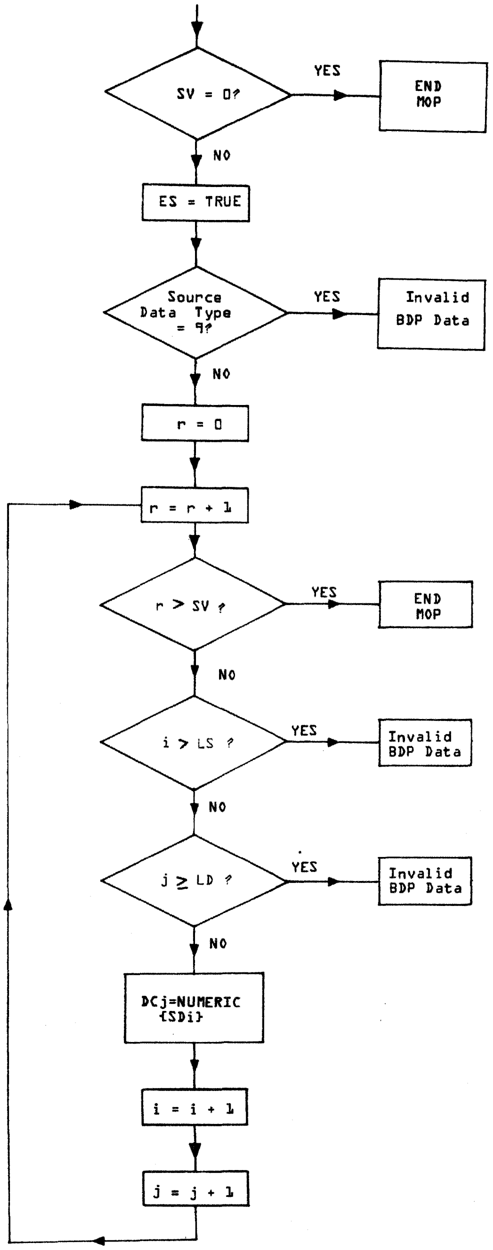
r Loop counting mechanism associated with SV and SM.

t Loop counting mechanism associated with LSM and SM.


Get Length of Edit Mask {Lm}
Lm = 0 or 1?
YES
END
NO
Initialize SCT = {20202B2D2C 2E242F}16
Initialize Symbol Length LSM = 0
Initialize Flags ES = FALSE SN = FALSE ZF = TRUE
b
b
+
-
,
.
$
/
SLASH
DOLLAR SIGN
PERIOD
COMMA
NEGATIVE SIGN
POSITIVE SIGN
BLANK - SUPPRESSION CHARACTER
BLANK - FILL CHARACTER
INITIAL VALUES OF THE SPECIAL CHARACTERS TABLE {SCT}
DATA TYPE 0,1,2,3,4,5,6,7, 8,9,12, or 13?
No
INST.SPEC. ERROR
Yes
NO
Signed Source Field?
YES
Examine Sign
Initialize Source Field Index i = 0
NO
Data Type = 1,3 or 13?
YES
i = 1
Initialize Destination Field Index j = 0
Initialize Mask Index k = 0
k = k+1
k > LM?
YES
END Inst.
NO
EXECUTE NEXT MOP
This instruction may terminate during the execution of a MOP by exhausting mask length etc.

INVALID BDP DATA - These flowcharts do not describe any specific step following the detection of Invalid BDP Data because the individual processor is free either to terminate the instruction immediately or to continue until the END Instruction conditions are met. {In either case the output field is undefined and bit 63 of the UCR is set.}

Figure 10-27. Edit Overview, Including Initialization



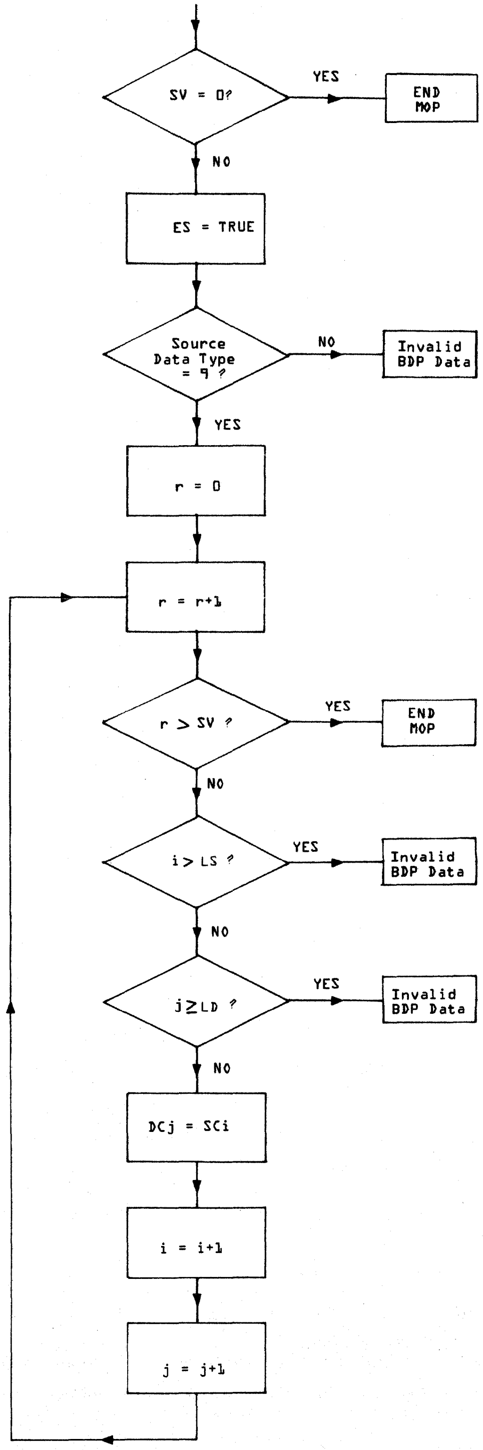
Notes:

1. This MOP translates 1-15 digits in the source field to their equivalent ASCII characters and copies them to the destination field.

2. The function NUMERIC is flow-charted elsewhere, and insures that the data being translated is valid, numeric data.

3. Set ES true if SV ≠ 0.

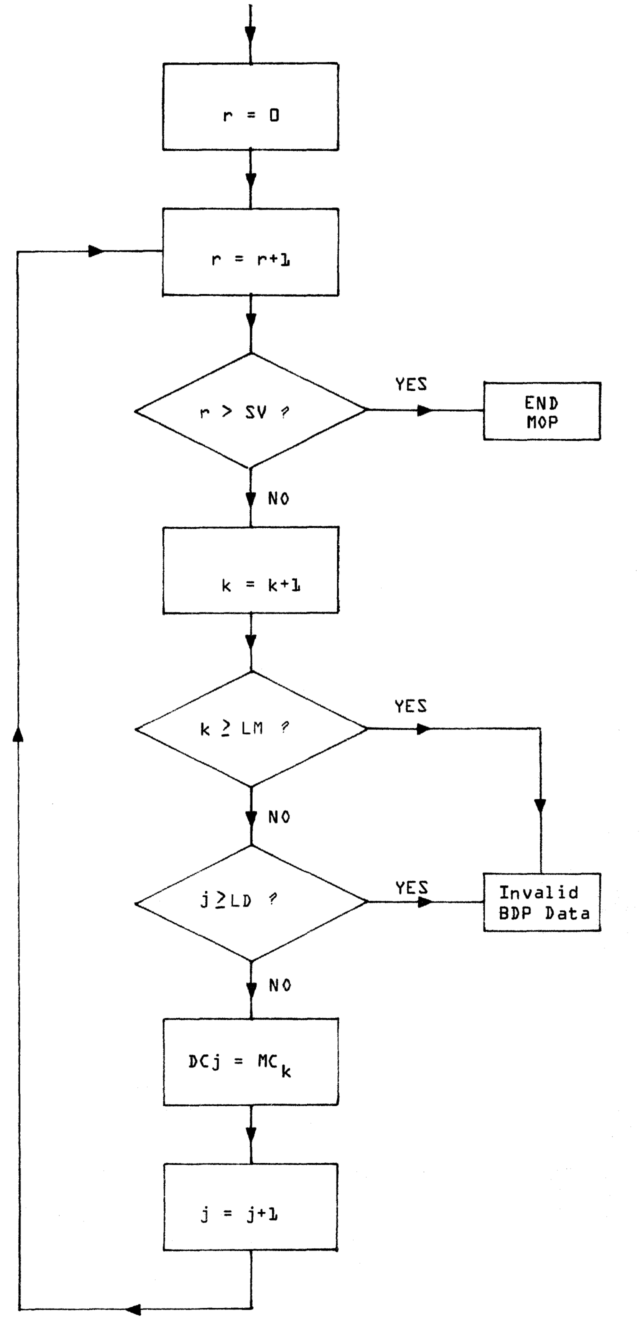
Figure 10-28. MOP 0 - Move Source Digits



Notes:

1. This MOP copies 1-15 characters from the source field to the destination field.

Figure 10-29. MOP 1 - Move Source Characters

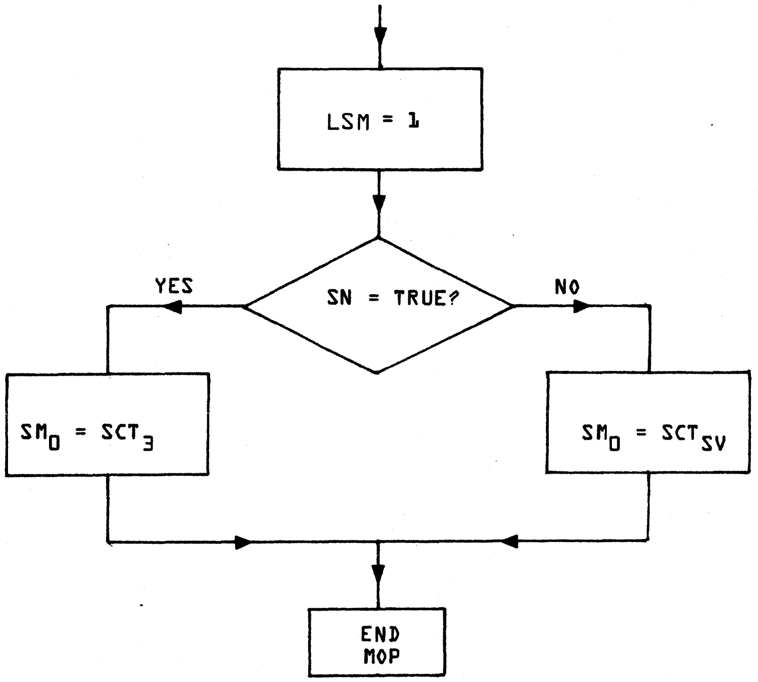


Notes:

1. This MOP moves 1-15 characters from the edit mask {in essence a micro-op string} to the destination field.

2. Any of the source data types 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 12 or 13 are legal for this MOP.

Figure 10-30. MOP 4 - Move Mask Characters



Notes:

1. This MOP sets the symbol to a single character representing the sign of the source data field.

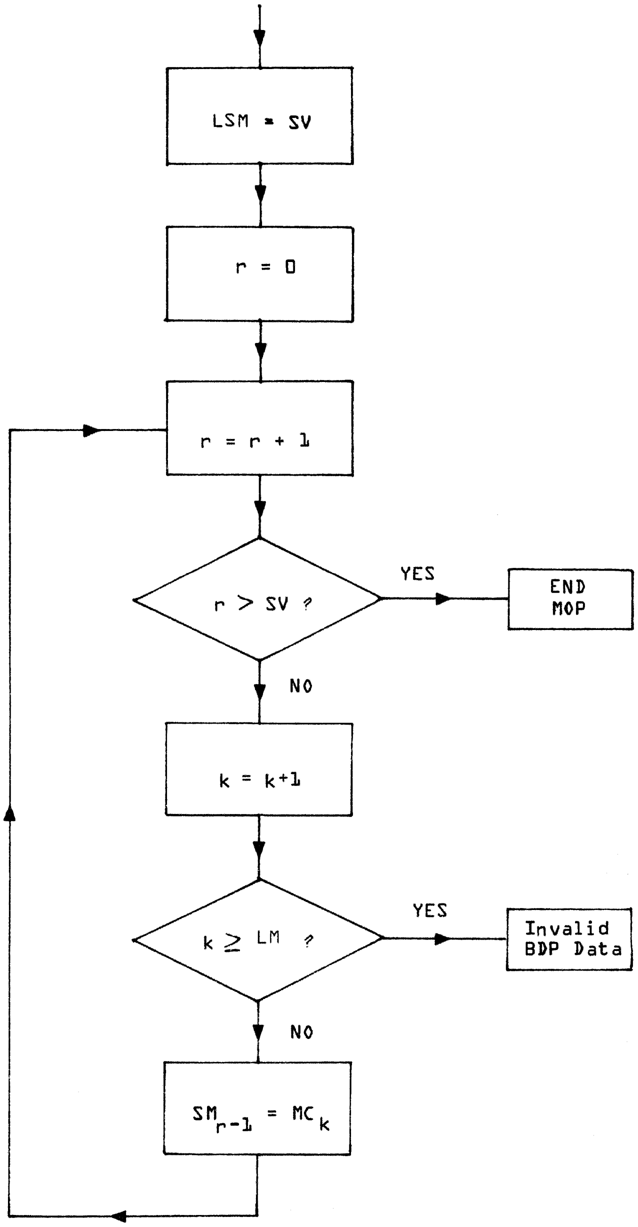
2. If the source data field is negative, then the sign is either set to minus (default value in the SCT) or to the value which has been stored in SCT(3).

3. If the source data field is positive, then the sign is set to a value selected from the SCT indexed by the least significant three bits of the specification value. Assuming a default SCT SV would normally have a value equal to 1 or 2 corresponding to blank and plus.

4. All values of SV are legal although only the rightmost three bits are interpreted when SV is used as an index.

5. Any of source data types 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 12 or 13 are legal for this MOP.

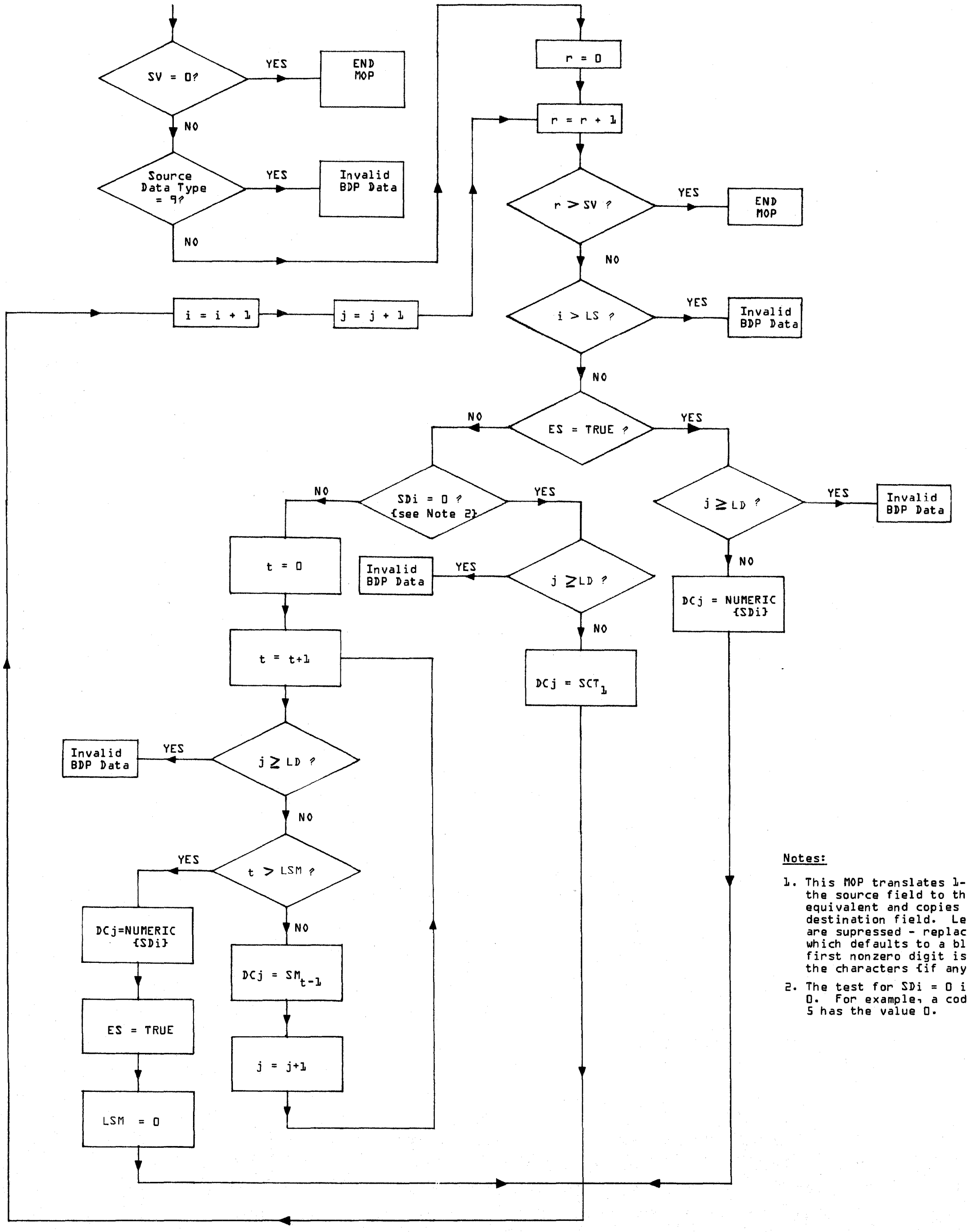
Figure 10-31. MOP 5 - Select Sign as Symbol



Notes:

1. This MOP copies 1-15 characters from the edit mask to the symbol.
2. All values of SV are legal for this MOP.
3. Any of the source data types 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 12 or 13 are legal for this MOP.

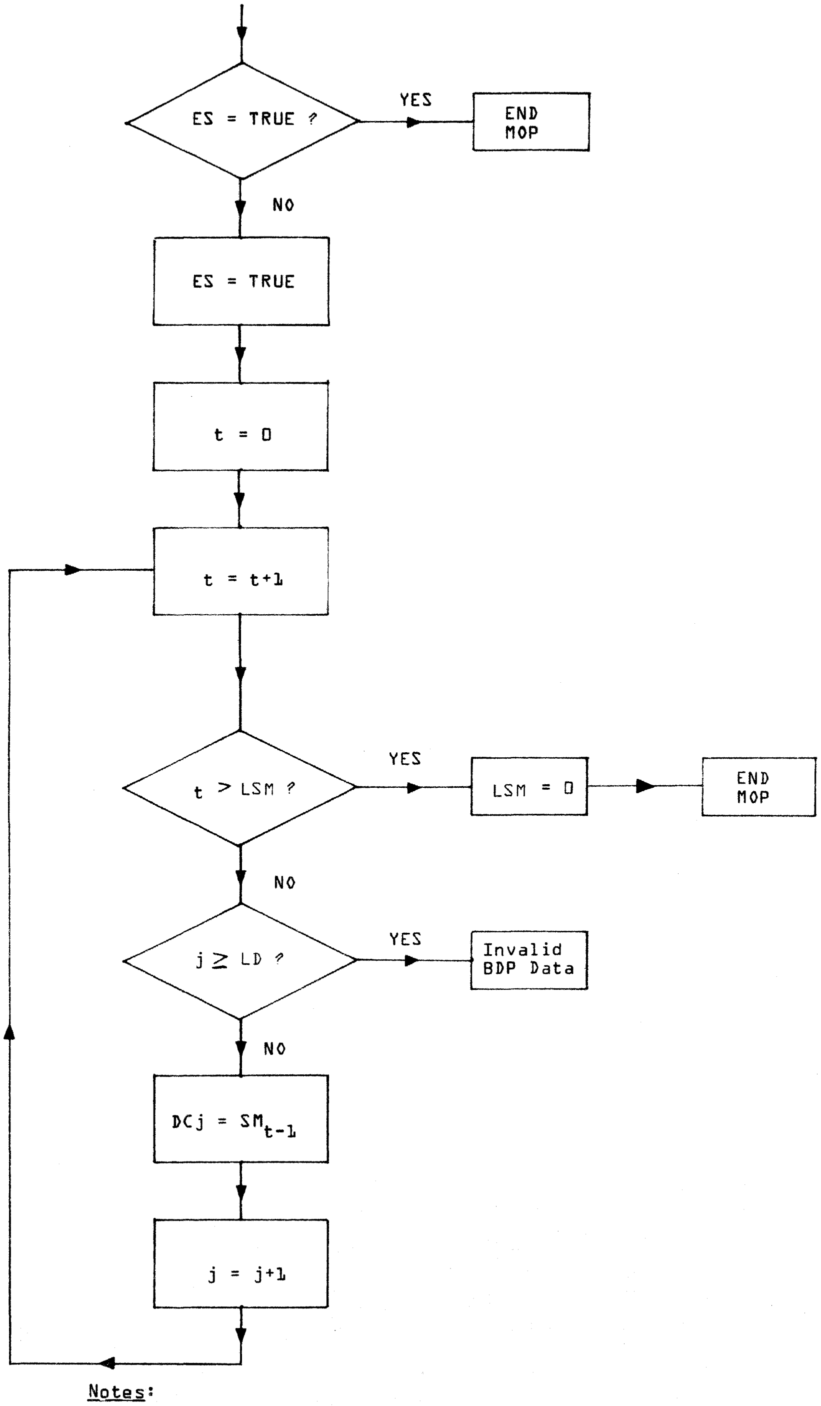
Figure 10-32. MOP 6 - Select Mask Characters as Symbols



Notes:

1. This MOP translates 1-15 digits from the source field to their ASCII equivalent and copies them to the destination field. Leading zeros are supressed - replaced by SCT{1} which defaults to a blank - and the first nonzero digit is preceded by the characters {if any} in the symbol.
2. The test for SDi = 0 is for the value 0. For example, a code of 3C on type 5 has the value 0.

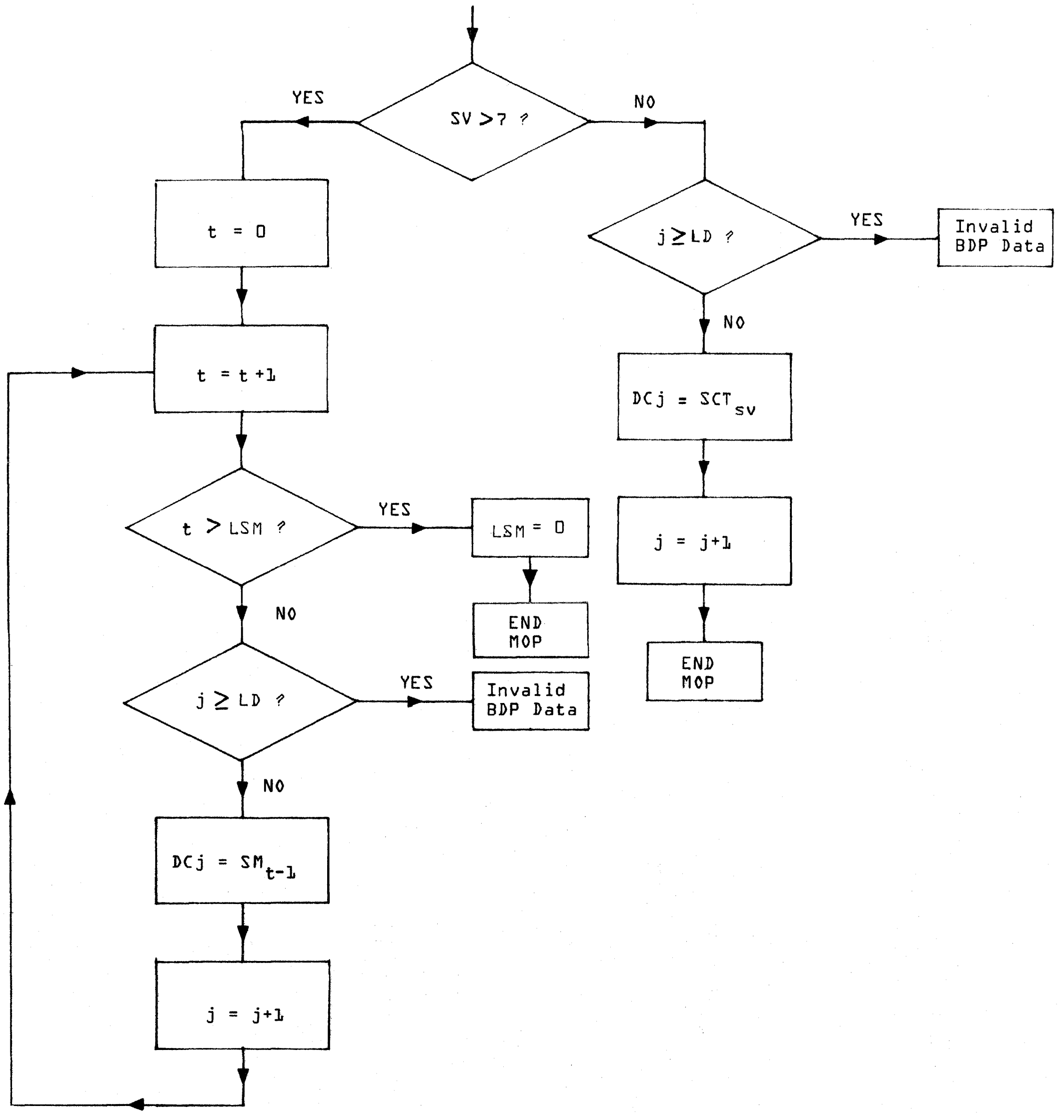
Figure 10-33. MOP 7 - Move Source Digits or Suppress with Floating Symbol



Notes:

1. If the End Suppression flag (ES) is not set, then this MOP copies the characters in the symbol to the destination field.
2. Any of source data types 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 12 or 13 are legal for this MOP.

Figure 10-34. MOP 8 - End Float



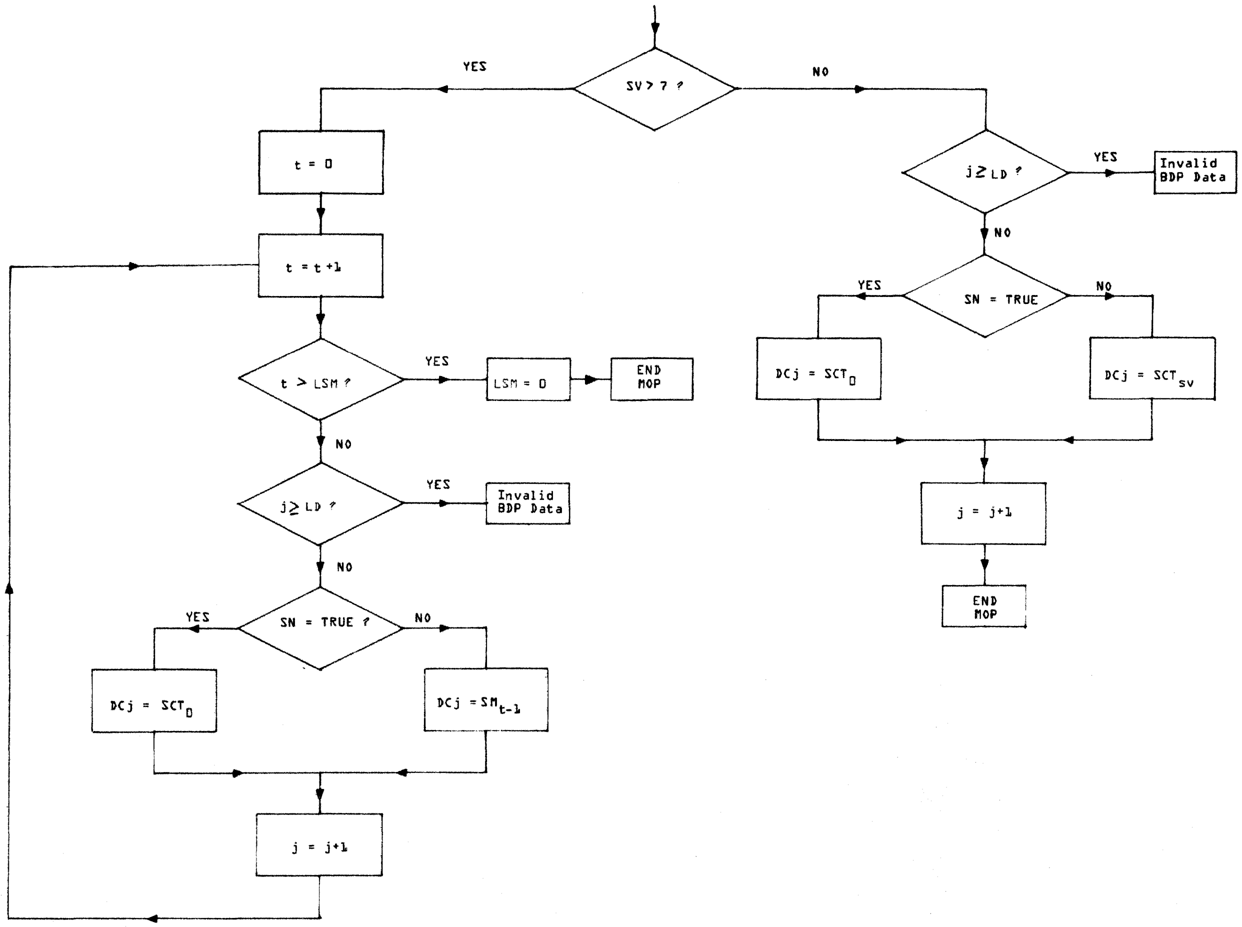
<u>Notes</u>:

1. This MOP either inserts the symbol characters or a character from the SCT into the destination field.

2. This MOP is controlled by the SV field. The most significant bit of this field is used as a flag. If set, then the symbol is inserted into the destination field, otherwise the SVth character from the SCT is inserted into the destination field.

3. Any of source data types 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 12 or 13 are legal for this MOP.

Figure 10-35. MOP 9 - Insert Symbol or SCT Character



Notes:

1. SV > 7

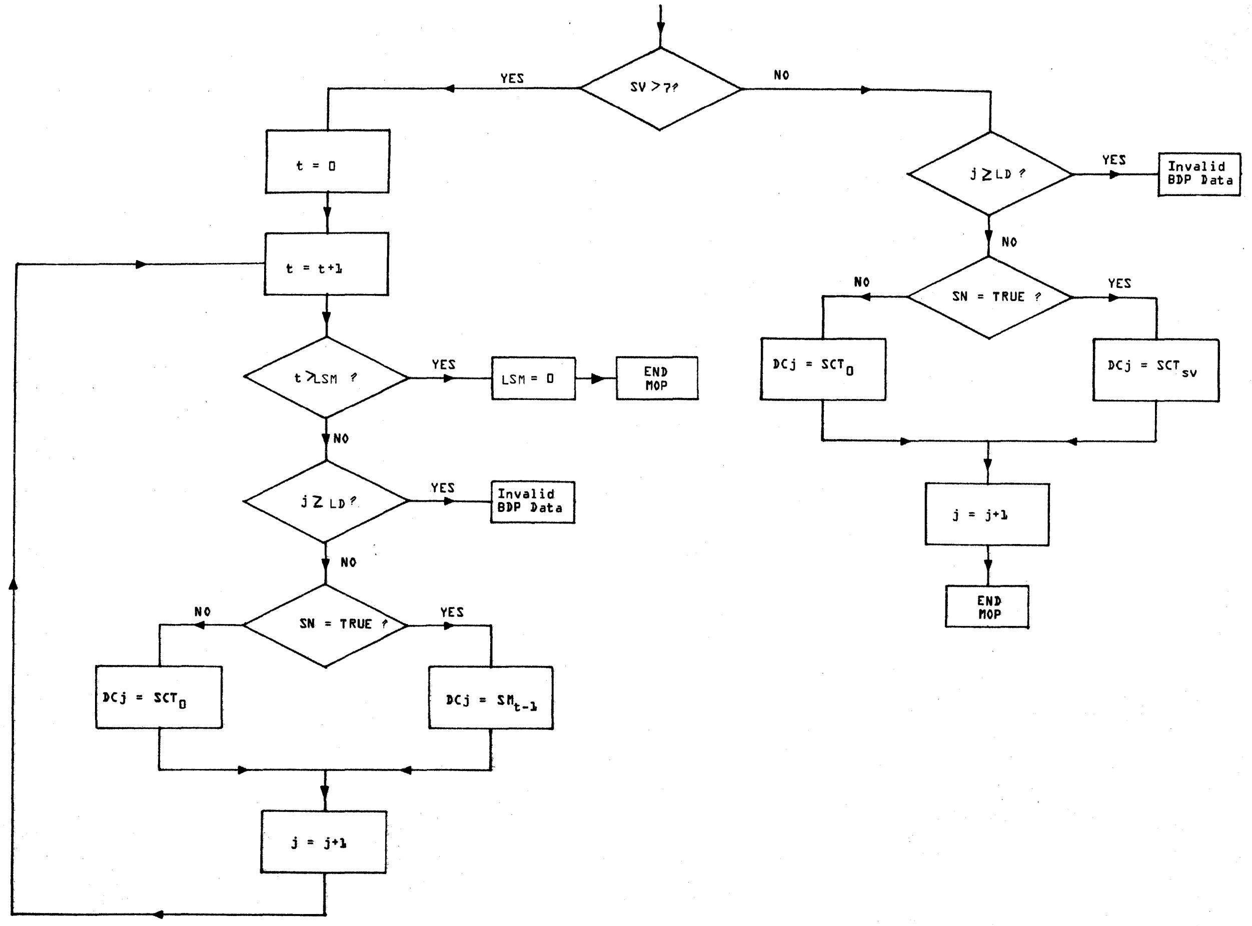
   Copy the Symbol to the destination field when the source field is positive, otherwise copy SCT$_0$ once for each character in the Symbol.

2. SV ≤ 7

   Copy SCT$_{SV}$ once to the destination field when the source field is positive, otherwise copy SCT$_0$ once.

3. Any of source data types 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 12 or 13 are legal for this MOP.

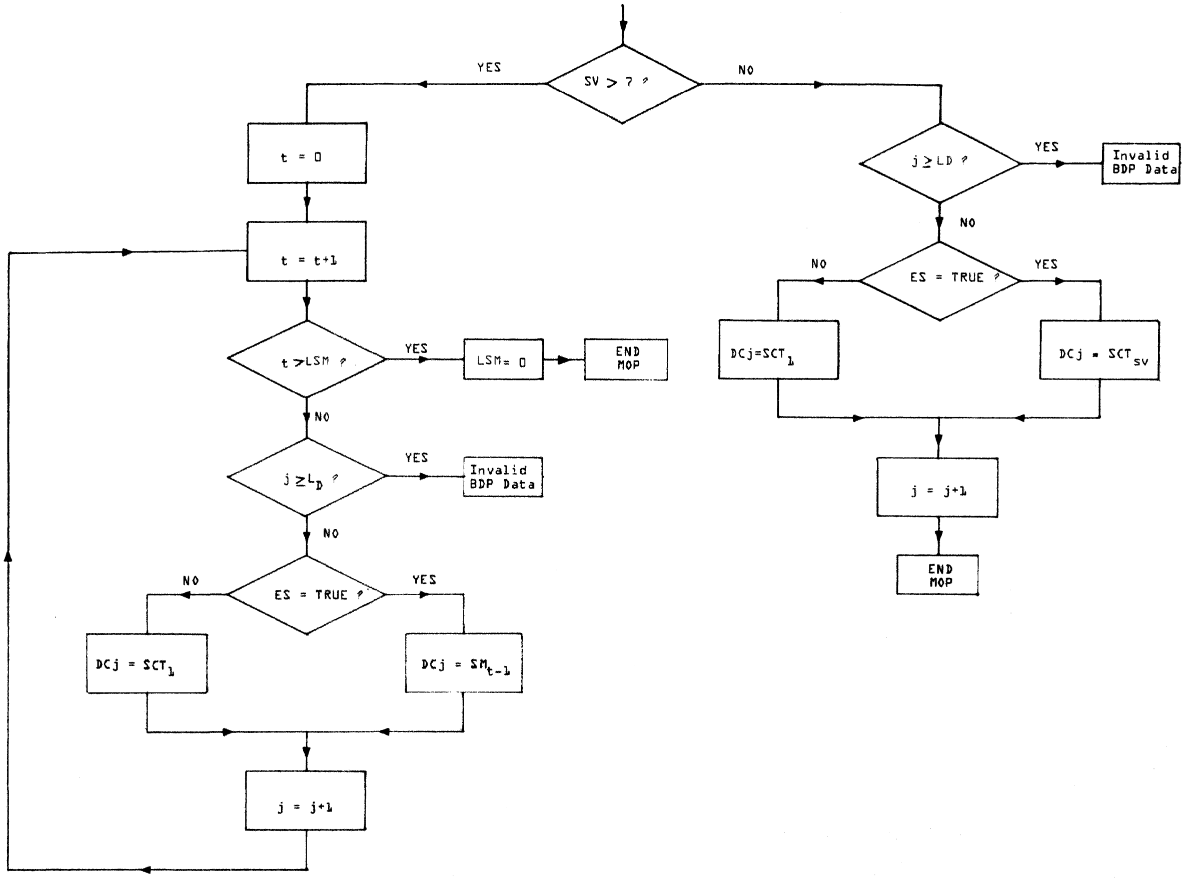
Figure 10-36. MOP A - Insert Symbol or SCT Character if Source is Positive, Elsewhere Insert Blanks



Notes: 1. This MOP is identical in all respects to MOP A except that the blank insertion occurs for a <u>positive</u> rather than negative source field.

2. Any of source data types 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 12 or 13 are legal for this MOP.

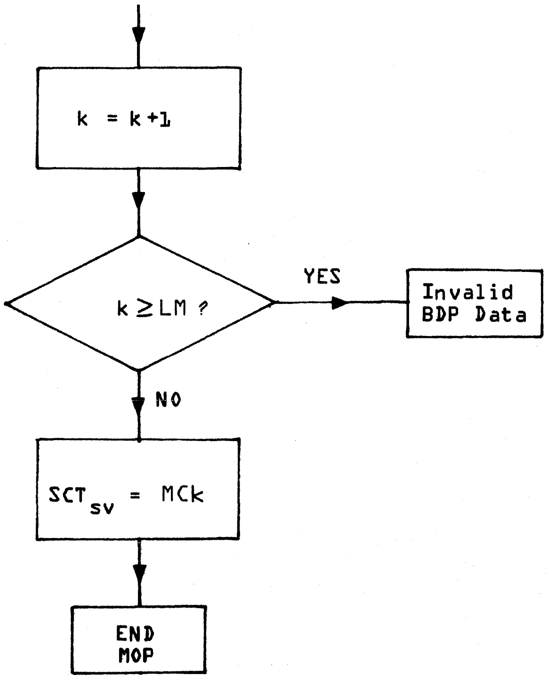
Figure 10-37. MOP B - Insert Symbol or SCT Character if Source is Negative, Else Insert Blanks



Notes: 1. This MOP is identical in all respects to MOP A except that the blank insertion occurs only when zero suppression is in effect.

2. Any of source data types 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 12, or 13 are legal for this MOP.

Figure 10-38. MOP C - Insert Symbol or SCT Character, Unless Suppression

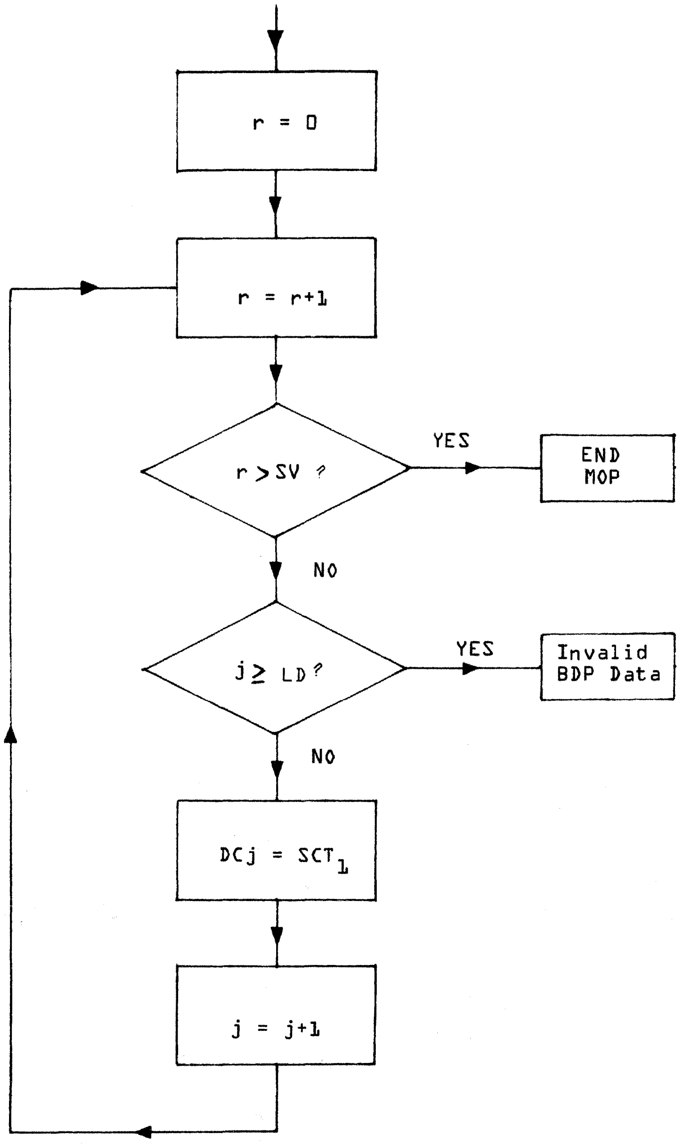


Notes: 1. This MOP copies the next character from the edit mask into the SVth character of the SCT.

2. Only the low-order three bits of the SV are used by this MOP.

3. Any of source data types 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 12 or 13 are legal for this MOP.

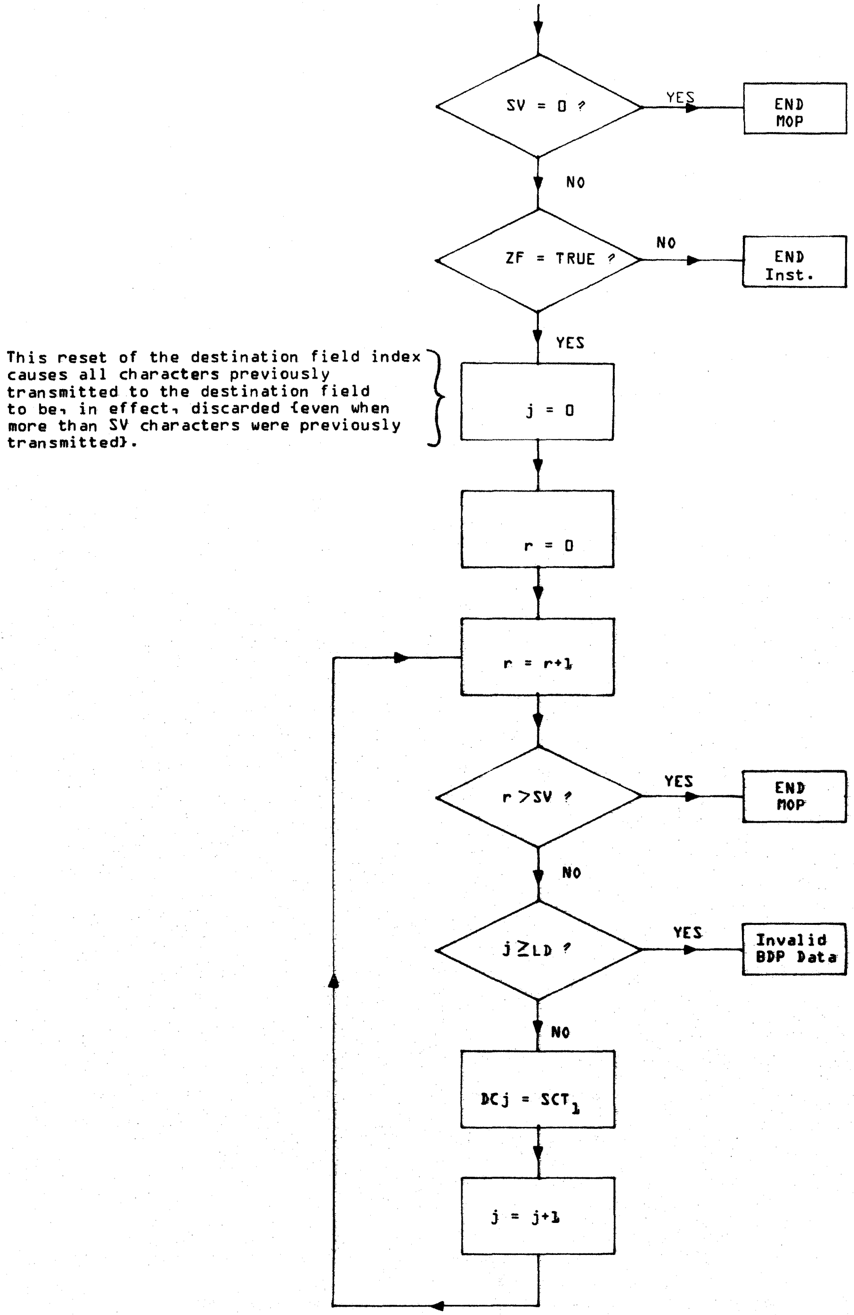
Figure 10-39. MOP D - Write SCT Entry



Notes: 1. This MOP copies the suppression character {from SCT{1}} into the destination field SV times.

2. Any of source data types 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 12, or 13 are legal for this MOP.

Figure 10-40. MOP E - Spread Suppression Character



This reset of the destination field index causes all characters previously transmitted to the destination field to be, in effect, discarded {even when more than SV characters were previously transmitted}.

Notes: 1. This MOP functions only for source fields with a zero value. A non-zero source field causes the termination of the edit instruction {not just this MOP}.

2. For a zero source field, SV suppression characters are copied into the destination field.

3. Any of source data types 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 12 or 13 are legal for this MOP.

Figure 10-41. MOP F - Reset and Suppression Zero Field

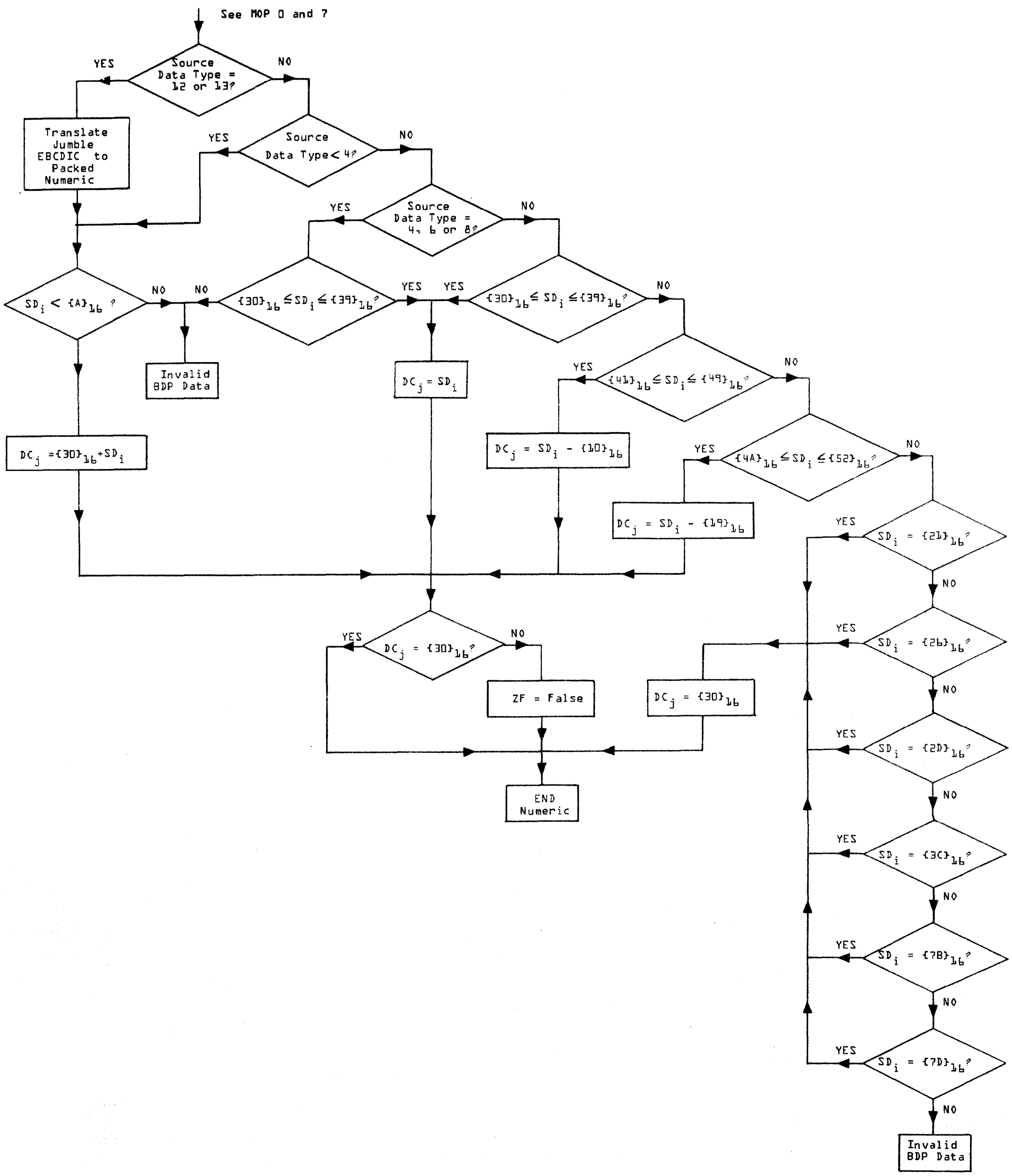


Figure 10-42. Numeric Function

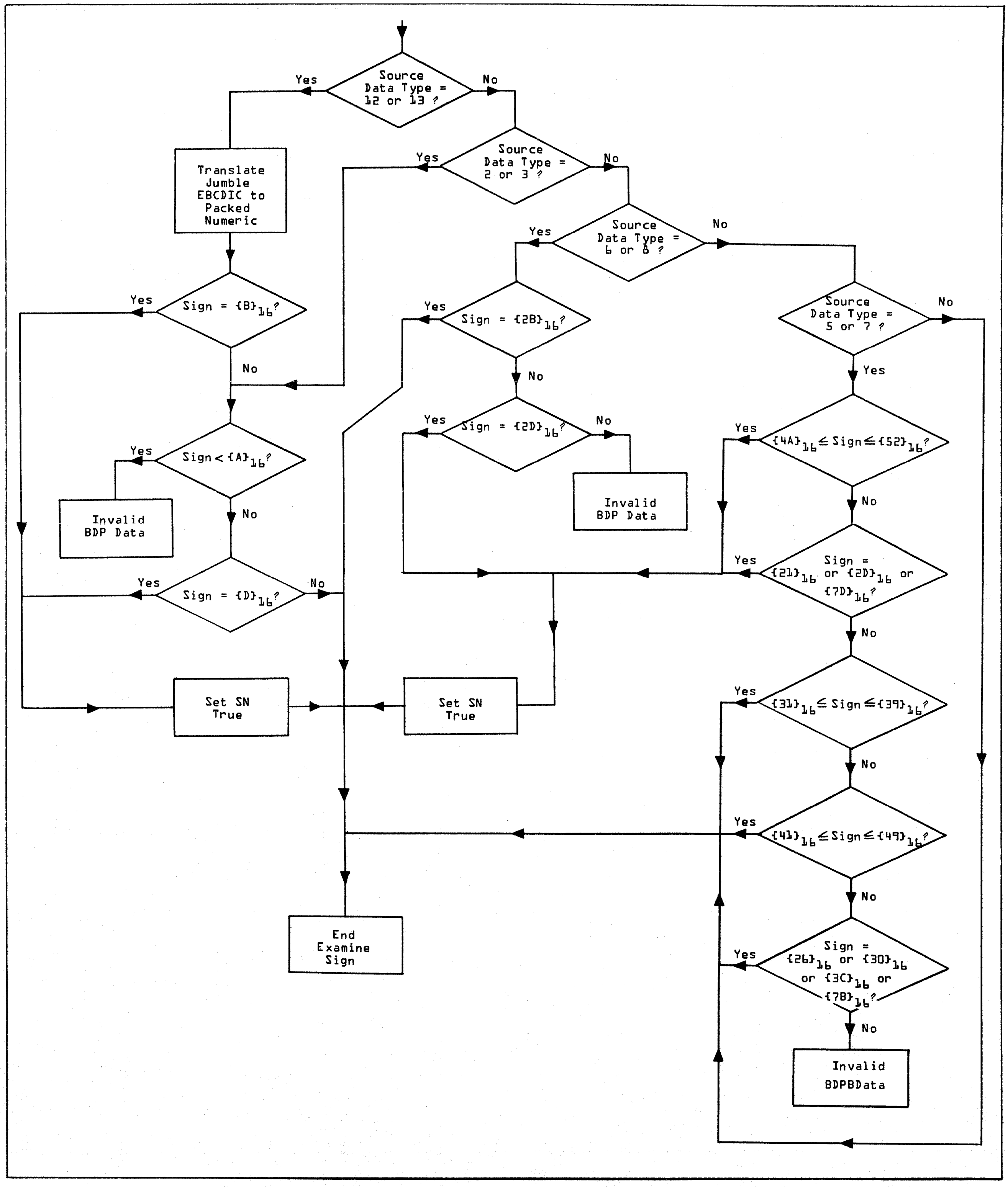


Figure 10-43. Examine Sign

Calculate Subscript

This instruction facilitates the computation of offsets in a multidimensioned array (figure 10-44). To understand how this instruction works it is necessary to formulate the equations used to calculate an index into an array. First of all it is assumed that the index values have an arbitrary origin. That is an array may be declared with an index that runs from a minimum to a maximum value such as: 2-10, -5-4, 0-15, 1-100, and so forth. Also it is assumed that an array may have an arbitrary number of dimensions, although repeated use of this instruction is required if the number of dimensions is greater than two.

The general expression for calculating a zero origin index into a multi-dimensional array $A * B * C * D * \ldots$ is:

$$([(a-A)(b.-B+1)+(b-B)](c.-C+1)+(c-C))(d.-D+1)+(d-D)\ldots.$$

Where $a.$ = maximum value of a

and A = minimum value of a, and so forth.

For example, take the array defined as follows

(3..10, 5..7, 8..9) (figure 10-44)

In this case: $a. = 9$; $A = 8$;

$b. = 7$; $B = 5$;

$c. = 10$; $C = 3$;

and the 0-origin index is given by

$$[(a-8)(3) + (b-5)]\ (8) + (c-3)$$

For example, the index of the element (6,6,8) [(c,b,a)] is:

$$[(8-8).3 + (6-5)]\ 8 + (6-3) = 11$$

Calculate Subscript is designed to compute the index of a 2-dimensional array. This is given by:

$$i = (a-A)\ (b.-B+1) + (b-B)$$

The terms A, B and b. are constants, as is the expression b., A, B. The terms B and b. and the expression b.-B+1 are called the minimum and maximum values, and the size respectively.

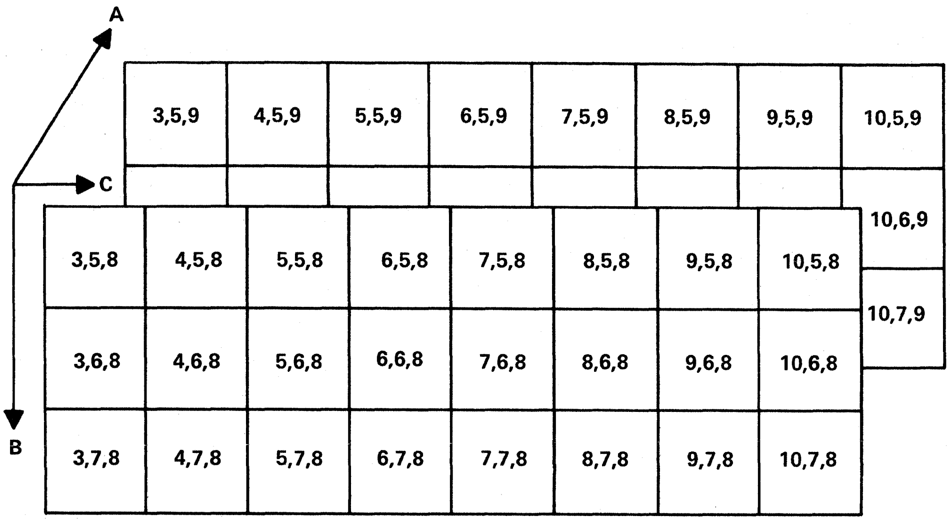


Figure 10-44. ARRAY (3..10, 5..7, 8..9)

To form the required index the quantities a, b, b., A, B and SIZE need to be specified. In the Calculate Subscript instruction the maximum, minimum and size quantities are supplied by a table found at (Ai)+D called the Subscript Range Table (SRT) (figure 10-45). The raw index values (a and b) are found in the source field (Aj) and destination register (Xk). The instruction proceeds as follows (figure 10-46):

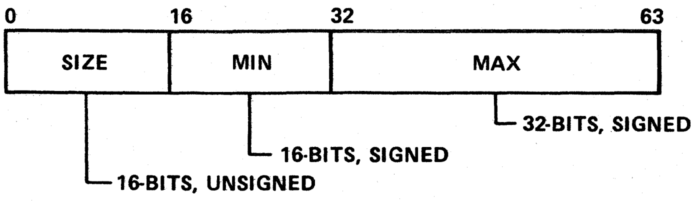


Figure 10-45. Subscript Range Table

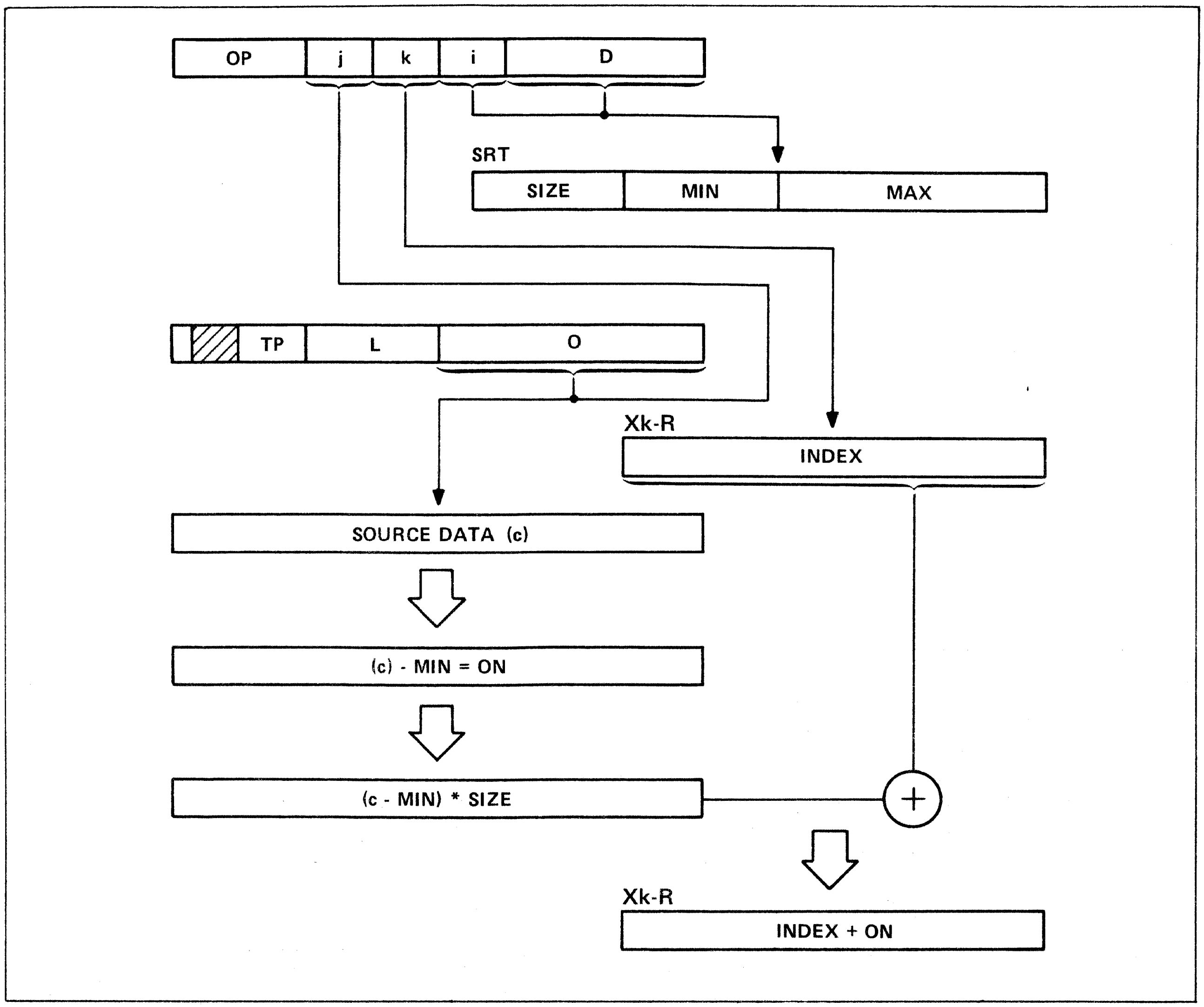


Figure 10-46. Calculate Subscript Operation

1. Take the binary value of the source field and subtract MIN from this value. (a-A) = occurrence number (ON)

2. Check that the occurrence number lies in the range:

$$0 \leq \text{ON} \leq \text{MAX}$$

3. Multiply this value by SIZE. [(a-A)(b.-B+1)].

4. Add this product (ON * SIZE) to the contents of Xk Right.

Points to note are:

- The SRT must be situated on a full word boundary. Failure to do so results in an Address Specification Error.

- The instruction performs a range check (step 3). If the index is out of range, then an Invalid BDP Data condition is detected.

- Since the contents of Xk are expected to be used as an index to access an array element, Xk is expected to hold the starting index of the most frequently changing variable corrected for a zero origin. In other words Xk should hold (b-B).

- For multiply dimensioned arrays where more than two dimensions are involved the general formula is viewed as follows:

  $\text{index} = (a-A) + (b-B)(a.-a) + (c-A)(a.-A)(b.-B) + \ldots$

  The Calculate Subscript instruction may be used twice as follows:

  (a) Set Xk to (a-A).

  (b) Calculate subscript and add - Aj will point to b, (Ai) + D will point to SRT(1).

  (c) Calculate subscript and add - Aj will point to c, (Ai) + D will point to SRT(2).

The SRT entries are:

Size	Min	Max
(a.-A+1)	A	(b.-B)
(a.+1-A)(b.+1-B)	B	(c.-C)

- Source field data types which are permitted are 0-6, 10 and 11.

A flowchart for Calculate Subscript is shown in figure 10-47.

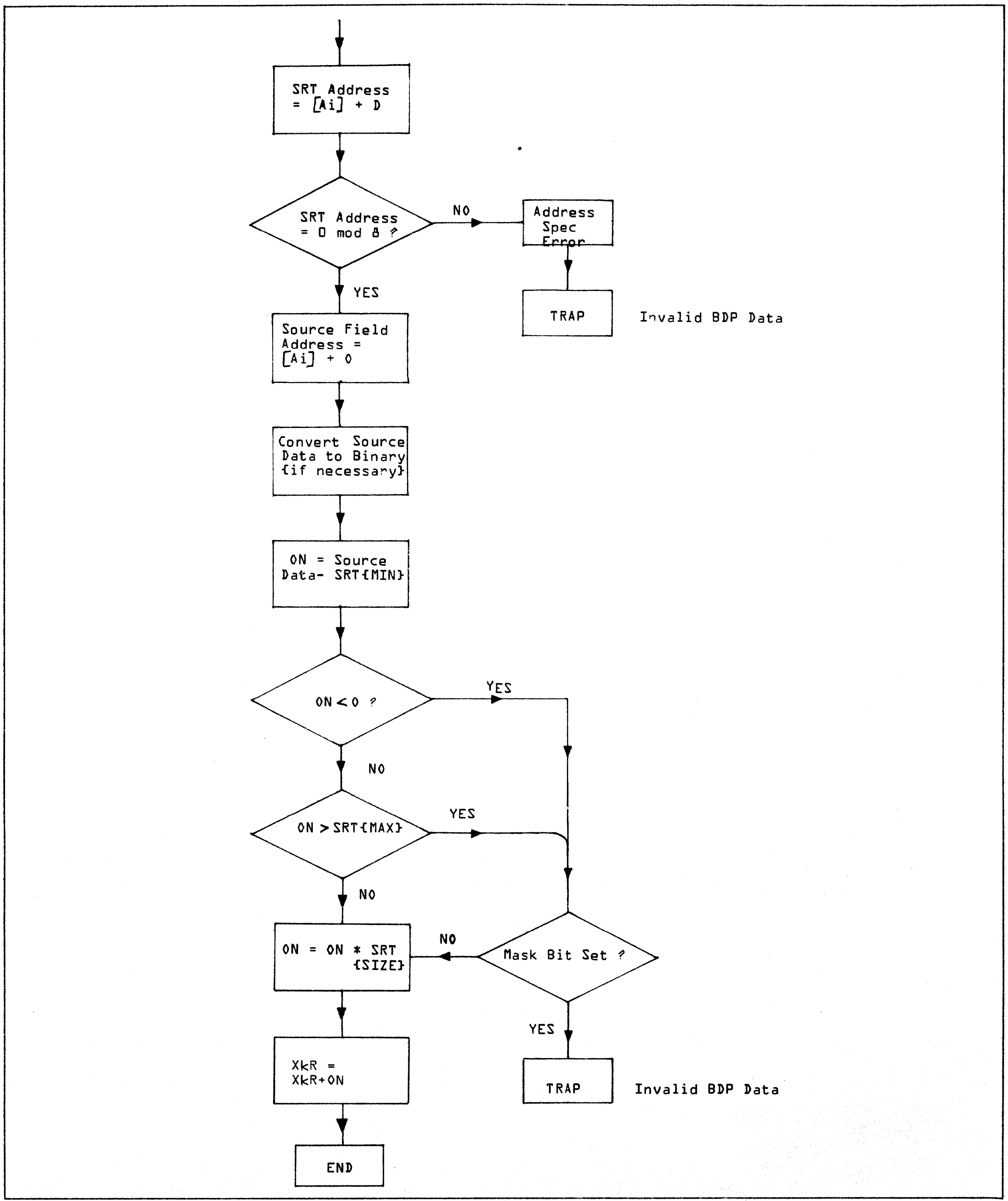


Figure 10-47. Calculate Subscript and Add

Immediate Data

There are three instructions in this group which operate with a single byte of source data held in the instruction itself. The instructions move or add to, or compare with a destination data element. Since there is a single descriptor the j-field, which is usually used with a descriptor to locate the source date, has a special function, namely that of determining the data type of the immediate operand.

For move and compare an operand is formed from the immediate operand which is controlled by the destination data type, and is either moved to or compared with the destination field. The lower order two bits of the j-field determine how this operand is formed as shown in figure 10-48:

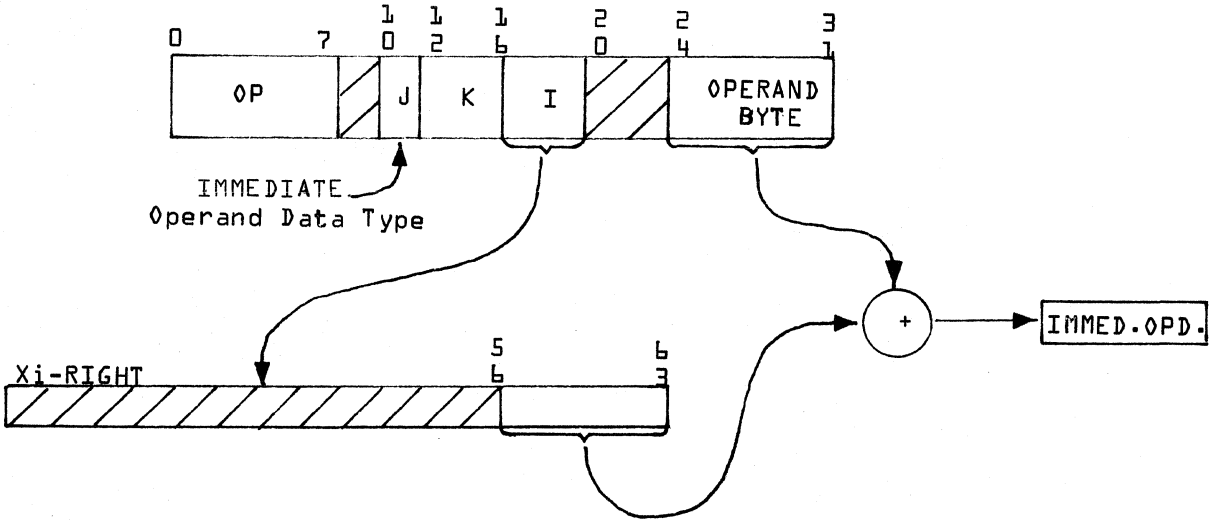


Figure 10-48. Immediate Data BDP Instructions

j-field	Source Data Type	Destination Data Type	Action
00	10	10,11,14,15	Binary: operand is right justified zero filled.
01	4	0-6,12,13	Decimal: operand is right justified zero filled, with sign (plus) supplied as required.
10	9	Ignored	Alphanumeric: immediate operand is repeated throughout the receiving field.
11	9	Ignored	Alphanumeric: operand is left justified, blank filled.

For Add, only the first two of these values for j have any meaning. In other words, only the low order bit of the j-field is used.

Points to note are:

- Once the immediate operand has been formed, these instructions perform exactly as their numeric counterparts.
- Unauthorized data types yield an Instruction Specification Error.
- On Immediate Add, if an overflow condition occurs, it is recognized and the program interrupt taken as required.
- The results of the compare are specified in X1-Right in the standard CYBER 180 manner:

  Source = destination : X1-Right = 00---0
  Source > destination : X1-Right = 01---0
  Source < destination : X1-Right = 11---0

FLOATING-POINT INSTRUCTIONS

There are sixteen instructions which operate on floating-point variables. Before describing them, it is necessary to understand the CYBER 180 floating-point format. In designing this format the objective was to yield the same results as CYBER 170 when operating with the same data using the same instructions. Since CYBER 170 has a 60-bit word and CYBER 180 has a 64-bit word there will be differences unless 4-bits of the CYBER 180 word are ignored. This was considered unacceptable, hence some trade-offs were made. The basic objective was used to define the magnitude of the fraction, end-cases will occur at different times on the two machines. General points of interest are:

- Single precision floating-point numbers consist of a sign, a signed exponent, and a 48-bit fraction (figure 10-49). Double precision floating-point numbers consist of a sign, a signed exponent and a 96-bit fraction (figure 10-50).

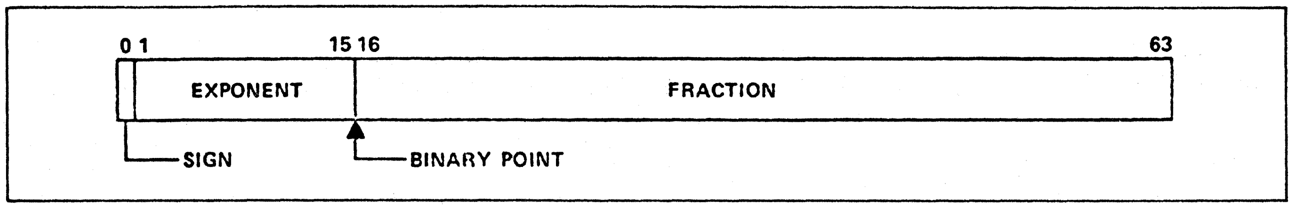


Figure 10-49. Single Precision Floating-Point Format

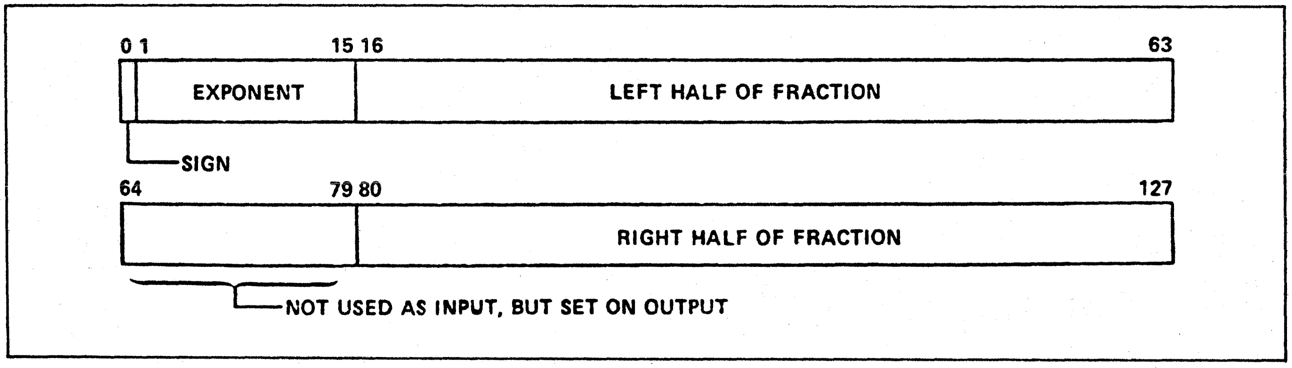


Figure 10-50. Double Precision Floating-Point Format

- The binary point occurs to the left of the fraction - as opposed to CYBER 170 where it is on the right.

- The representation is signed-magnitude. The fraction is always a positive value. It also means that both plus zero (+0) and minus zero (-0) are represented.

- The exponent is a 15-bit biased quantity which has two bits reserved immediately to the right of the most significant (bias) bit. These bits are used to flag the special conditions of overflow (infinite), underflow (zero), and indefinite.

- Actual exponent values take 12-bits as opposed to 10-bits on CYBER 170 and this affects end cases.

- There are no rounding instructions on CYBER 180. This will lead to differences between CYBER 180 and CYBER 170 for those CYBER 170 procedures using rounded arithmetic - notably the FORTRAN math library.

- There is a full complement of instructions which operate on double precision floating-point numbers. Typically these execute in microcode and have not been included for reasons of speed, but for convenience. Hence, it is not practical time-wise to simulate rounded single-precision arithmetic using these instructions.

- The sign and exponent of the lower half of a double precision floating-point number are ignored on input to a double precision operation and set equal to the sign and exponent of the upper half on output. This convention simplifies the task of programming a double precision floating-point compare for which there is no hardware instruction.

The FORTRAN statement:

```
     DOUBLE  A,B
       .
       .
       .
     IF(A-B) 10,20,30
```

can be compiled as follows, assuming A and B to be in registers X2-X5.

```
      BRFEQ   X2,X4,ATMP#     .upper halves equal
      BRFGT   X2,X4,A30#      .A>B (A-B +ve)
A10#  BSS     0               .A<B (A-B -ve)
       .
       .
       .
ATMP# BRFEQ   X3,X5,A20#      .A=B (A-B zero)
      BRFGT   X3,X5,A30#      .A>B (A-B +ve)
      BRXEQ   X0,X0,A10#      .A<B (A-B -ve)
```

which is a reasonably compact sequence.

Some care must be taken when constructing double-precision numbers to adhere to the hardware conventions, or comparisons may yield spurious results. The lower half of a double-precision quantity may have a fraction consisting entirely of zeroes, but have an exponent which is in range. These numbers are denoted by the symbol Z3 and are really unnormalized zeros. If Z3 is compared to Zero (a word of all zeros) the result may be less than or greater than, depending on the sign of Z3. However, Z3 compares equal to itself.

- Unlike CYBER 170, there is no explicit normalization operation. However, when normalized operands are presented to the floating-point units, normalized numbers result. In particular, add and subtract instructions perform a full post-normalization. hence, a floating add of zero will suffice to normalize a floating-point number.

Results emitted by the floating-point units vary in accordance with the user mask (UM) register setting. When an operation yields either an exponent overflow or exponent underflow, a predetermined result of either infinity or zero will be returned unless the UM bit is set which corresponds to that condition. In these cases the true result is returned. This is made possible by the representation chosen for floating-point variables and is discussed more fully below in the section dealing with nonstandard floating-point numbers. When both the UM bit is set and traps are enabled, the floating-point operations exhibit a unique feature: they complete execution before the trap is taken. This is made possible by two factors. First, the floating-point representation accommodates all out-of-range numbers which can be generated by the hardware. Next, all floating-point arithmetic instructions are 16-bit instructions. Consequently, there can be no loss of precision as a result of executing the instruction. This is true for underflow and overflow results and for floating-point loss of significance. It is not true for indefinite, which can arise in both 16-bit and 32-bit instructions (branches). A predetermined result will always be returned for indefinite unless the corresponding UM bit is set and traps are enabled, in which case instruction execution is inhibited.

Nonstandard Numbers

Standard floating-point numbers have biased exponents in the range (hex.):

$(3000) \leq e < (5000)$

This leaves numbers with exponents in the ranges (hex.) of:

$(0000) \leq e < (3000)$ and

$(5000) \leq e < (7FFF)$

for the representation of nonstandard numbers. These gaps in the range have the following meanings (hex.):

$(7000) \leq e \leq (7FFF)$: INDEFINITE

$(5000) \leq e < (7000)$: INFINITE

These numbers with exponent overflow may be generated by the hardware.

$(1000) \leq e < (3000)$: UNDERFLOW

These numbers with exponent overflow may be generated by the hardware.

$(0000) \leq e < (1000)$: ZERO

With the single exception of the hex. number (0000), these numbers cannot be generated by the hardware.

Input operands having exponents in the range (hex.) of:

$(0000) \leq e < (3000)$

are interpreted as Zero by the hardware.

The full range of floating-point numbers is tabulated (figure 10-51), followed by an illustration of the nonstandard numbers (figure 10-52). A word consisting entirely of zeros may be received as input to, or issued as output from, a floating-point unit as a true zero. By hardware convention, however, zero is a nonstandard floating-point number.

	HEXADECIMAL EXPONENT INCLUDING CO-EFFICIENT SIGN	ACTUAL EXPONENT (TO THE BASE 2)	INPUT ARGUMENTS	
COEFFICIENT SIGN EQUAL TO 0 (POSITIVE NUMBERS)	7XXX	– – – –	INDEFINITE	
	6FFF ↑ 5000	$2^{12,287}$ ↑ $2^{4,096}$	INFINITE	
	4FFF ↑ 4000 3FFF ↓ 3000	2^{4095} ↑ 2^{0} 2^{-1} ↓ $2^{-4,096}$	STANDARD	NUMBERS IN THIS RANGE WITH ZERO COEFFICIENTS ARE TERMED +Z3
	2FFF ↓ 1000	$2^{-4,097}$ ↓ $2^{-12,288}$	ZERO	+Z2
	0XXX	– – – –	ZERO	+Z1
COEFFICIENT SIGN EQUAL TO 1 (NEGATIVE NUMBERS)	8XXX	– – – –	ZERO	–Z1
	9000 ↑ AFFF	$2^{-12,288}$ ↑ $2^{-4,097}$	ZERO	–Z2
	B000 ↑ BFFF C000 ↓ CFFF	$2^{-4,096}$ ↑ 2^{-1} 2^{0} ↓ $2^{4,095}$	STANDARD	NUMBERS IN THIS RANGE WITH ZERO COEFFICIENTS ARE TERMED –Z3
	D000 ↓ EFFF	2^{4096} ↓ $2^{12,287}$	INFINITE	
	FXXX	– – –	INDEFINITE	

Figure 10-51. Floating-Point Representation

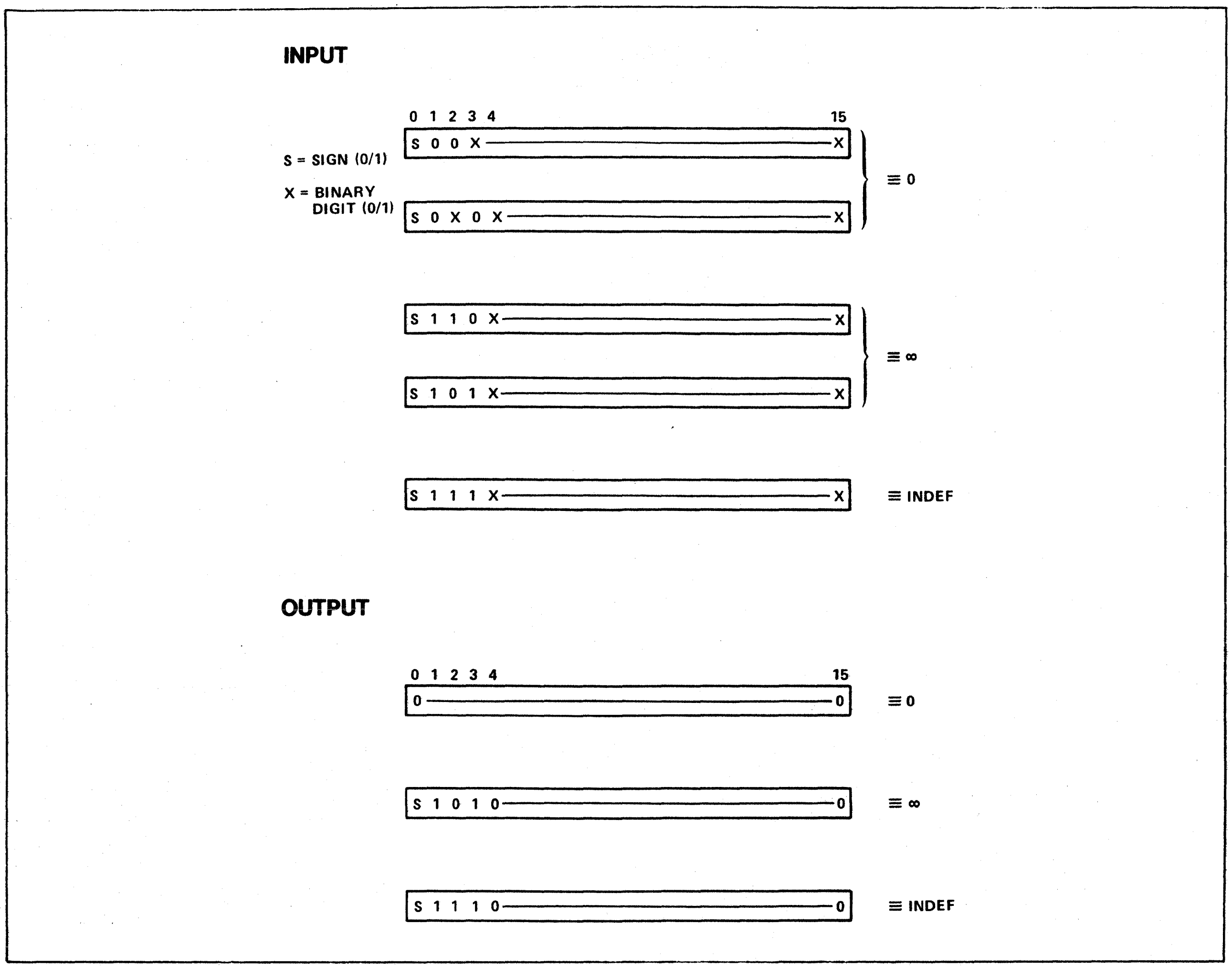


Figure 10-52. Nonstandard Floating-Point Numbers

CONVERT INSTRUCTIONS

Two instructions are provided to translate between integers and floating-point numbers. Points to note are:

- When converting from integer, numbers outside the range:

  -2^{48} through $(2^{48} -1)$

  are truncated in their rightmost bits. ONLY

- When converting to integer, numbers which are: CNFI

  -indefinite
  -infinite
  -have actual (unbiased) exponents ≤ 0
  -have fractions = 0

  are converted to zero.

  Numbers in the range:

  $-(2^{63} - 2^{15})$ through $(2^{63} - 2^{15})$

  are converted exactly, and numbers outside this range have the least significant 64-bits of the result returned as the result with a floating-point loss of significance condition detected.

ADD/SUBTRACT - SINGLE AND DOUBLE PRECISION

Points to note are:

- These are 2-address instructions.

- Double precision operands are located in registers Xj and Xj+1, Xk and Xk+1. If j or k = F then (j+1) or (k+1) = 0.

- Indefinite or infinite source operands yield indefinite or infinite results (figure 10-53).

- Nonstandard numbers (in the gap) are input as zero.

- These instructions post-normalize. In fact, they provide the mechanism for normalizing unnormalized numbers.

- When an exponent overflow occurs one of two things happens depending on the condition of the mask bit corresponding with this condition. If the mask bit is not set: no interrupt occurs and the standard form for infinity is output.

  If the mask bit is set: the exponent along with its bias, and the normalized fraction along with its sign is returned as the result. Even though an overflow condition has been detected, the result is still a true and accurate floating-point representation of the result. An interrupt is taken and actions which follow are then determined by the code compiled (by the user) to handle this interrupt.

- When an exponent underflow occurs, precisely the same actions take place as with an exponent overflow, except that when the mask bit (for exponent underflow) is not set a result of zero (all zeros) is returned. With the mask bit set, the result will be true result in the gap between zero and the standard floating point numbers.

  IZ3

- When the operation results in a zero fraction (including the overflow bit) a result of all zeros is returned if the mask bit associated with floating-point loss of significance is clear. If the mask bit is set, then the exponent along with its bias, is returned with a zero fraction, and a floating-point loss of significance is recorded. The exception to this rule occurs when both operands are zero, as defined by their exponents. No loss of significance occurs in this case. If both operands are zero, however, and at least one of them has a standard exponent and a zero fraction (Z3), then a floating-point loss of significance results.

Add

Xj Xk	W	+∞	-∞	∓IND
W	S	+∞	-∞	IND
+∞	+∞	+∞	IND	IND
-∞	-∞	IND	-∞	IND
∓IND	IND	IND	IND	IND

Subtract

Xj Xk	W	+∞	-∞	∓IND
W	D	-∞	+∞	IND
+∞	+∞	IND	+∞	IND
-∞	-∞	-∞	IND	IND
∓IND	IND	IND	IND	IND

Figure 10-53. Add/Subtract - Nonstandard Floating-Point Numbers

PRODUCT - SINGLE AND DOUBLE PRECISION

Points to note are:

- This is a 2-address instruction (A=A*B).
- Indefinite, infinite or zero source operands yield indefinite, infinite or zero results (figure 10-54).
- Nonstandard numbers (in the gap) are input as zero.
- This operation performs the following function:

$$A.2^{a} * B.2^{b} = A.B.2^{(a+b)}$$

Xk \ Xj	+N	-N	+0	-0	+∞	-∞	∓IND
+N	+P	-P	0	0	+∞	-∞	IND
-N	-P	+P	0	0	-∞	+∞	IND
+0	0	0	0	0	IND	IND	IND
-0	0	0	0	0	IND	IND	IND
+∞	+∞	-∞	IND	IND	+∞	-∞	IND
-∞	-∞	+∞	IND	IND	-∞	+∞	IND
∓IND	IND	IND	IND	IND	IND	IND	IND

Figure 10-54. Multiply - Nonstandard Floating-Point Numbers

In single precision it forms a 96-bit product from two 48-bit source operands then normalizes one bit position. This means that normalized input yields normalized output. However, if the source operands are unnormalized, the state of the product is unknown. The reason for this one bit normalization is that the smallest normalized coefficient is a half. If this is squared, the smallest product results (namely a quarter) which is unnormalized by one bit position.

- Exponent underflow and exponent overflow are handled as previously with add/subtract. Since the exponents are added in this case, the largest positive exponent that can be obtained is 8190 and the largest negative exponent is -8192. Both of these quantities fall in the range of nonstandard numbers. Hence, if they are generated, and the associated mask bit is set, a correct value is returned.

4095 4096
4095 4096
8190 -8192

QUOTIENT - SINGLE AND DOUBLE PRECISION

Points to note are:

- This is a 2-address instruction (A=A/B).

- Indefinite, infinite and zero source operands yield indefinite, infinite or zero results (figure 10-55).

Xj Xk	+N	-N	+0	-0	+∞	-∞	∓IND
+N	+Q	-Q	DIVIDE FAULT - EXECUTION INHIBITED	DIVIDE FAULT - EXECUTION INHIBITED	0	0	IND
-N	-Q	+Q			0	0	IND
+0	0	0			0	0	IND
-0	0	0			0	0	IND
+∞	+∞	-∞			IND	IND	IND
-∞	-∞	+∞			IND	IND	IND
∓IND	IND	IND			IND	IND	IND

Figure 10-55. Divide - Nonstandard Floating-Point Numbers

- Nonstandard numbers (in the gap) are input as zero.
- This operation performs the following function:

$$A.2^{a} / B.2^{b} = (A/B) . 2^{(a-b)}$$

In single precision a 96-bit dividend is formed by appending 48-zeros to the low order bit of the second operand. A 48-bit quotient is formed which is normalized one bit position. This means that normalized input yields normalized output. However, if the source operands are unnormalized, then the state of the quotient is unknown. The reason for this one bit normalization is that the largest normalized fraction is

$$1 - 2^{-48}$$

and the smallest normalized fraction is a half. The quotient formed by dividing these quantities is

$$2 - 2^{-47}$$

which overflows by one bit.

- Two cases are of special interest. These are a divisor of zero, and a resulting quotient which would be equal to or greater than 2.0. In those cases, the instruction execution is inhibited, a divide fault indicated and the corresponding interrupt taken if the mask bit is set.

BRANCH AND COMPARE INSTRUCTIONS

Two single-precision floating-point quantities may be compared for: equal, not equal, greater than, greater than or equal. When the condition is met either a branch may be taken or the results of the comparison may be returned to register X1-Right. When the result of the comparison is transmitted to X1-Right in the standard CYBER 180 manner:

Xj = Xk,	X1-Right = 000---0
Xj > Xk,	X1-Right = 010---0
Xj < Xk,	X1-Right = 110---0

In addition, an indefinite condition is flagged by the value 100---0 in X1-Right. Points to note are:

1) Operands compared by branch instructions will have the same results as those compared by the compare instruction. All of those instructions examine the exponents and signs of the input operands to determine the nature of the comparison. The result is determined directly if the exponent of either operand shows it to be zero, infinite, or indefinite. This is also true if the operand signs differ. While this approach minimizes execution time, some anomalies may occur if the operands are unnormalized. Of particular interest is the Z3 operand, which has an in-range exponent but a zero fraction. Z3 is an unnormalized zero, but since these preliminary decisions are based solely on examination of the exponent, comparisons involving Z3 yield arbitrary results.

   When Z3 is compared to a true zero (as defined by its exponent), the result (<,>) is determined by the sign of Z3. An equal comparison is not possible. When Z3 is compared to an in-range quantity, however, the result (<,>,=) is based not only on the sign, but in the case of the signs being equal, it is based on the magnitude of the exponent as well. In the peculiar case of Z3 having a large exponent (but a zero fraction) and the second operand having a small exponent (but a nonzero fraction), this second operand will have its fraction driven to zero during exponent equalization, and the quantitites will compare equal. Comparisons involving Z3 are unique in that they may yield results which differ from other similar operations, such as subtracting the two quantities being compared and then branching on the result. Figures 10-56 and 10-57 at the end of this section should help clarify these and other issues relating to floating-point instructions.

2) If X0 is specified as a register, it is interpreted as all zeros. This means that the contents of X0 cannot be tested explicitly.

3) If either operand is indefinite, a floating-point indefinite condition is recorded and a normal exit taken. falls thru

4) When both operands are standard floating-point numbers, a floating-point subtract is executed to determine the results of the comparison.

Xk \ Xj	+N	-N	+0	-0	+∞	-∞	±IND
+N	D	<	<	<	>	<	IND
-N	>	D	>	>	>	<	IND
+0	>	<	=	=	>	<	IND
-0	>	<	=	=	>	<	IND
+∞	<	<	<	<	IND	<	IND
-∞	>	>	>	>	>	IND	IND
±IND	IND	IND	IND	IND	IND	IND	IND

Figure 10-56. BRANCH and COMPARE - Nonstandard Floating-Point Numbers

Exception Branch

This instruction tests a single precision floating-point number for an exception condition (figure 10-57). Conditions sensed are exponent overflow, exponent underflow and indefinite and then are determined by the exponent value as described previously.

A user may elect not to set the mask bits in the user condition register to force an interrupt when an exception condition arises. Instead, the conditions may be sensed in line via use of that instruction.

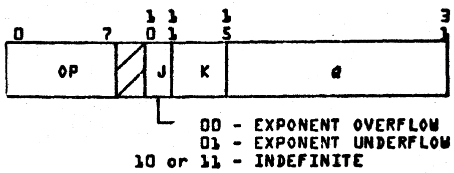


Figure 10-57. Exception Branch

SYSTEM INSTRUCTIONS

INTRODUCTION

The CYBER 180 instruction repertoire includes 16 instructions which are designed to facilitate operating system processing. These instructions include the CALL/RETURN/POP instructions which have already been discussed. While some of these are specific to the operating system, others,like CALL/RETURN, have a more general utility. A mechanism has been defined whereby the operating system specific instructions cannot be executed by users in a destructive or accidental manner. This mechanism is described in the following paragraphs.

PRIVILEGED STATES OF EXECUTION

There are two major machine states, which are referred to as monitor mode and user mode. When the machine is in the monitor mode, exchange interrupts are disabled. In addition, certain system instructions have their execution restricted to this mode. The switch between the two states can only be made with an exchange jump which may be issued either explicitly, as in the case of a monitor to job switch or a system call by a user, or implicitly in the form of an exchange interrupt in job mode. This machine state cannot be altered by any means other than an exchange jump. It is set at deadstart/initialization time by master clearing the processor, and it is recorded in the Environment Control Register (DEC) for maintenance purposes.

When the processor is in monitor mode, typically, traps are disabled, which, in turn, means that interrupts are locked out. Since the CYBER 180 systems are designed to provide a minimum response time to an external stimulus (external interrupt) it is important to minimize the time consumed with interrupts locked-out. If operating system instructions are restricted to monitor mode, then those processors using these instructions would have to run in monitor mode and interrupt response times would increase. For this reason certain other privileged states have been introduced. These privileged states are termed: global, local and unprivileged and are structured hierarchically. The controlling unit for these states is the code segment, the states being established by the Segment Descriptor Entries (SDE´s) for those code segments (figure 10-58).

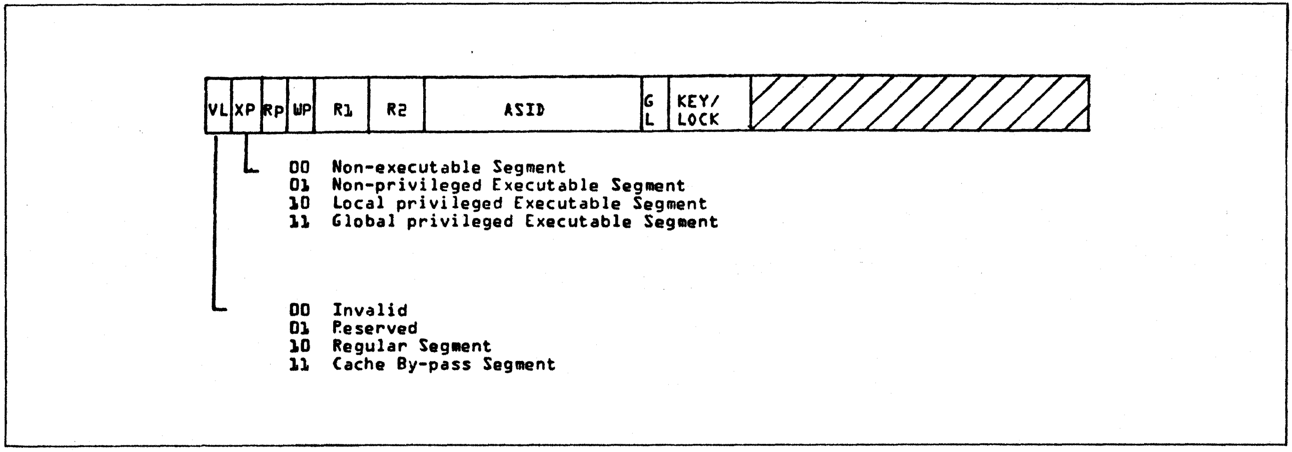

VL
XP
RP
WP
R1
R2
ASID
G L
KEY/ LOCK
00 Non-executable Segment
01 Non-privileged Executable Segment
10 Local privileged Executable Segment
11 Global privileged Executable Segment
00 Invalid
01 Reserved
10 Regular Segment
11 Cache By-pass Segment

Figure 10-58. Segment Description Table Entry - SDE

This hierarchical concept of privileged states has the following advantages:

a. The amount of code which must be executed with interrupts disabled can be kept to an absolute minimum, which guarantees an interrupt response time.

b. The operating system monitor can be restricted to a small number of critical functions. This minimizes the amount of code which must be developed and maximizes the mean time between failures for this code. In fact, the goal is to have no software errors in this code.

c. Access to certain registers, for example hardware maintenance registers, can be restricted to users who have the need to access (in particular write) these registers.

As a result, the normal end-user will run in an unprivileged mode, whereas certain portions of the operating system will run with various privileges, and maintenance routines (on-line diagnostics) will have the privilege necessary to access the hardware maintenance registers. A detailed description of the system instructions and their intended usage now follows.

EXCHANGE JUMP

The exchange jump instruction is used to switch the processor between monitor mode and job mode. During an exchange jump the current state of the machine, as described by the process state registers, is saved in central memory, and a new state is created by loading the process state registers from another area in central memory. Unlike CYBER 170 machines, CYBER 180 machines do not swap the contents of central memory and the process state registers. Instead, the registers are saved in one area and loaded from another. These areas in central memory are specified by two processor state registers: Monitor Process State (MPS); and Job Process State (JPS). These registers hold Real Memory Addresses. This is very important and has software implications. The exchange jump mechanism is one of the very few operations which bypasses the virtual memory mechanism. In turn, this means that it bypasses cache for those processors which have cache buffers. Hence, if care is not exercised stale data can end up in cache memory (which works off an SVA) after an exchange jump. This could be handled by purging the appropriate part of cache. However, the recommendation is that exchange packages be placed in cache bypass segments so that they never get loaded into cache (refer to figure 10-58).

Exchange jumps can occur explicitly or implicitly as follows:

a) If the processor is in monitor mode, an exchange jump will place the processor in job mode, and instruction execution will commence at the PVA specified by the content of the P Register in word zero of the exchange package at JPS.

b) If the processor is in job mode, a user may issue an exchange jump, which will place the processor in monitor mode at an address specified by the content of the P Register found in word zero of the exchange package at MPS. This is termed a System Call and sets bit 58 in the MCR to differentiate it from other monitor fault conditions.

c) When the processor is in job mode, a monitor fault condition as determined by the MCR will cause an exchange interrupt which will place the processor in monitor mode.

It is important to understand the concept of an address space as defined by the entries in a Segment Descriptor Table (SDT) for a process. Each user has an address space which contains all the code being executed in that user process and the data on which that code is operating. In addition, the user address space will include the operating system services required for the process and at least one segment for communication with the operating system address space. The user address space and the operating system address space are distinct entities. It is impossible for the user to access the operating system address space and it is extremely difficult for the operating system to access the user address space. The translation of a PVA involves a table look-up in the SDT and since the user´s SDT is inactive when the operating system monitor is executing, the automatic address translation does not function. The operating system has created the user SDT. Hence it can simulate the address translate mechanism, but this would be tedious and impractical. This is one reason why trap interrupts have been provided so that those conditions which must be handled from within the user´s address space cause interrupts to that environment.

When a system call is issued by a user it is necessary for the user to communicate with the operating system such that Monitor can determine the reason for the call. There are a number of ways this can be accomplished. One way is to communicate the required information across the address spaces in X Registers. This is perhaps the simplest technique, although it has the drawback that the amount of information which can be transmitted is limited. Another technique is for the operating system to create a Segment which exists in both the operating system and user address spaces. This is simple to achieve and suggests a general communication medium for jobs and the operating system monitor.

The exchange jump mechanism can be used to effect a virtual machine switch. On CYBER 180 this mechanism, along with the CALL/RETURN mechanism is the technique used to switch between CYBER 180 native state and CYBER 170 state. This process is discussed fully in the section dealing with CYBER 170 state. When monitor exchanges to job and makes a virtual machine switch, a common exchange package is established between the two virtual machines. The Virtual Machine Capability List (VMCL) determines which virtual machines exist in a given processor. If an exchange is made to a virtual machine which does not exist (a VMID/VMCL mismatch), then an Environment Specification Error is detected, and, on completion of the exchange jump, an exchange interrupt occurs. The converse of this is theoretically impossible since the initialization process always commences in a CYBER 180 environment. However, a hardware failure could generate this condition, in which case the processor will halt, since monitor conditions always cause an exchange interrupt from job mode to CYBER 180 monitor mode.

KEYPOINT

CYBER 180 hardware has built-in facilities for gathering system performance data. These facilities are activated by the keypoint instruction. Performance data may be gathered either by software or by an optional hardware performance monitoring facility (PMF). Software retrieval is governed by the Keypoint Enable Flag (KEF) - a process state register which is set for an exchange interval by the operating system. If the KEF is set and traps are enabled, then an interrupt will occur on keypoint instructions under control of the Keypoint Mask (KM) register. It is then the responsibility of the trap interrupt routine to save any required data. Hardware retrieval of performance data is controlled by register 22 in the PMF, and is independent of any retrieval by software. The hardware records the keypoint class, keypoint code, and time of day. This information may be recovered by software over the maintenance channel. In addition to this data, the PMF can monitor other specific events such as cache hits, MAP hits and Page Table hits, which could not be measured readily via software techniques.

Keypoint instructions are activated under the control of the Keypoint Mask (KM) and selected by the Keypoint Class Number (KCN). Complete identification of the instruction is by Keypoint Code within KCN.

The j-field of the instruction contains the KCN which is used as a bit index into the Keypoint Mask. That is, a KCN of five indexes bit 5 of the KM - the sixth bit from the left-hand end of the KM. If this bit is set, then performance data is collected. This data typically includes: time of day; KCN; and Keypoint Code (figure 10-59).

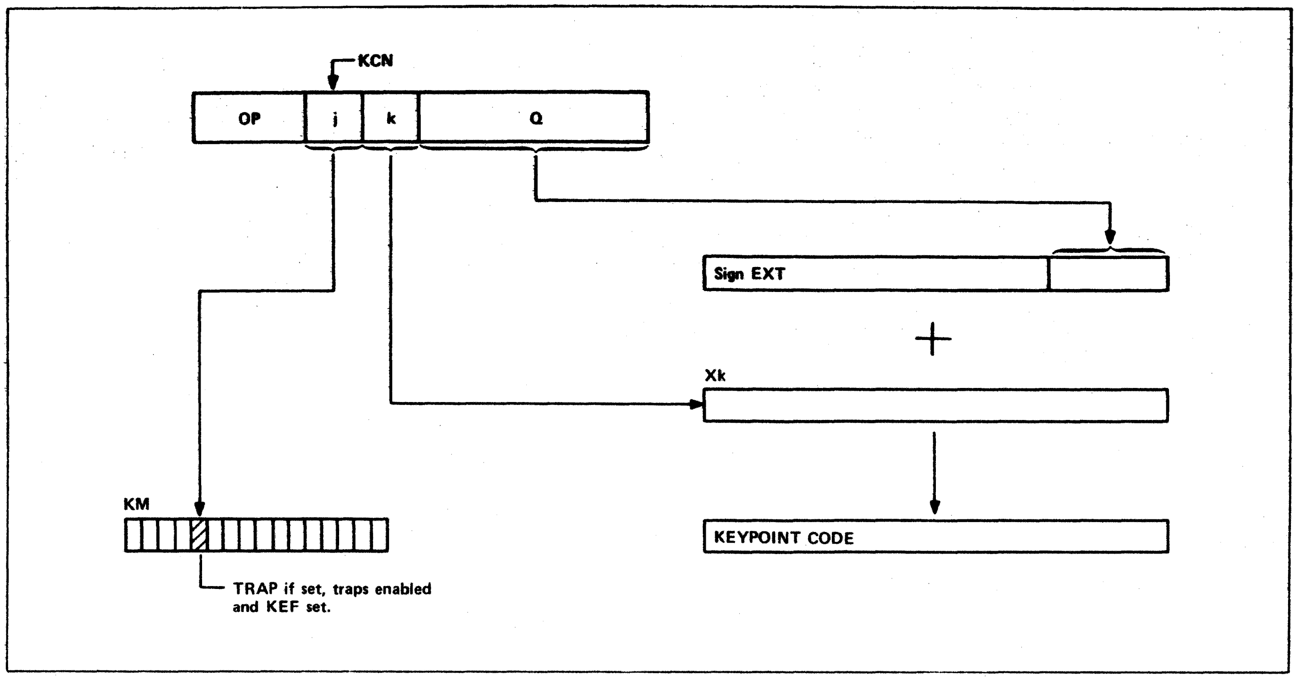


Figure 10-59. Keypoint Operation

In general, Keypoint instructions are placed throughout code segments where performance data is gathered. For example, performance data on I/O routines may be required. A Keypoint Class, as specified by the KCN could be established for all I/O routines and Keypoint instructions inserted as appropriate with this KCN. The individual routines would then be identified uniquely by the Keypoint Code formed from Register-Xk and the Q-field of the keypoint instruction.

COMPARE SWAP

There are two techniques which are employed by CYBER 180 to interlock data in central memory which is accessed/modified by more than one processor. One functions on a word basis and the other on a bit basis. The general theory of the interlock instructions is to establish a convention whereby a zero indicates unlocked and a one indicates locked. To set a lock, a one is set in a register which is then exchanged with the interlock in central memory. If the result in the register is a one, then the lock was already set, otherwise it has been set and the process can continue.

The actual operation of the Compare/Swap instruction is more complicated than this model (figure 10-60) although the theory is the same. In the Compare/Swap a quantity in register Xk is compared with the interlock word in central memory whose PVA is contained in register Aj. If they are equal, then the lock can be set, and this is accomplished by storing X0 in the interlock word. If they are unequal, then the interlock word is loaded into register Xk to indicate that the lock was already set. In addition to this basic Compare/Swap, or compare and set lock, a check is made to determine whether the interlock is locked by hardware. If it is, then a branch exit is taken to (P+2*Q).

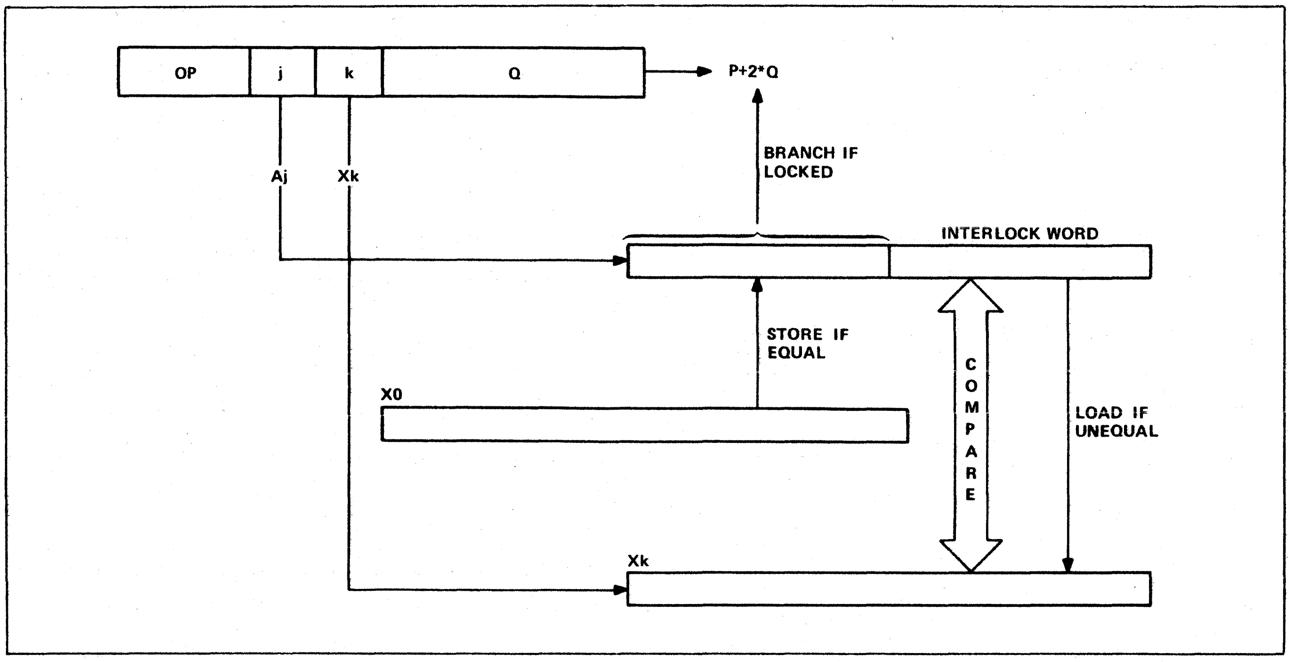


Figure 10-60. Compare/Swap Operation

A single interlock pattern has been reserved for the hardware in this respect. This interlock pattern is a word which consists of all ones in its first 32 bit positions. The need for this hardware interlock arises from the fact that the total operation does not take place in memory. Rather the processor uses a memory exchange function to exchange the interlock word with a word having all ones in its leftmost 32 bits. This sets the hardware lock if it was not already set. It then proceeds to complete the remainder of the operation in the processor, finally storing the appropriate word into the interlock, thereby clearing the hardware lock. Software is prevented from setting the hardware lock by this instruction. If X0-left contains all ones then an Instruction Specification Error is detected. This hardware interlock process ensures that during the execution of this instruction no other processor can attempt the same instruction on the same interlock word. In other words, once the instruction has been committed it will run to conclusion without interference. Instructions which exhibit this characteristic are called Nonpreemptive Instructions.

In a monoprocessor environment, a similar problem can arise if memory accesses take place out of sequence with respect to the sequence in which they were issued. It is for this reason that these instructions are preserialized and postserialized with respect to memory accesses on the part of the given processor. All memory requests issued by the processor are satisfied before the interlock instruction is issued, and the memory accesses for the interlock are completed before the next instruction is issued. On the memories designed for S2 and S3, requests are satisfied in the sequence they are issued. However, this may not be true for a common bulk memory with a slow access and many banks. For these memories it is conceivable that only those requests for the same bank will be sequenced. This is an important concept, since this function is not always performed by the hardware and it is sometimes incumbent upon the programmer to ensure the desired serialization is carried out.

General Notes:

- Since this instruction swaps the contents of a register and central memory, it requires that the interlock word be in a segment which has both Read and Write access.

- The interlock word must reside on a word boundary. This is in order that the 64-bit memory exchange function can be used. An Address Specification Error results if this condition is not met.

- The operation (exchange) occurs in central memory and bypasses cache. Cache is bypassed on the read portion and purged on the write. Since the purpose is to interlock processors a common point must be utilized. Cache memory is peculiar to a given processor and, therefore, cannot be used and should be bypassed.

- To simplify the debug operation the following assumptions are made:

  - The operands are not locked (by hardware).

  - The operands are used for both read and write access.

  - The branch reject address is not used as an argument for debug.

TEST AND SET BIT

This is the second instruction available for interlocking processors. Unlike the Compare/Swap, the Test and Set Bit instruction functions on a single bit in memory. The operation is straight-forward. An interlock bit in central memory is loaded into register Xk, right justified, zero filled, and the bit is set in central memory. A subsequent investigation of register Xk will determine whether the lock has been set. If Xk is zero, the the lock has been set, otherwise it was already locked (figure 10-61).

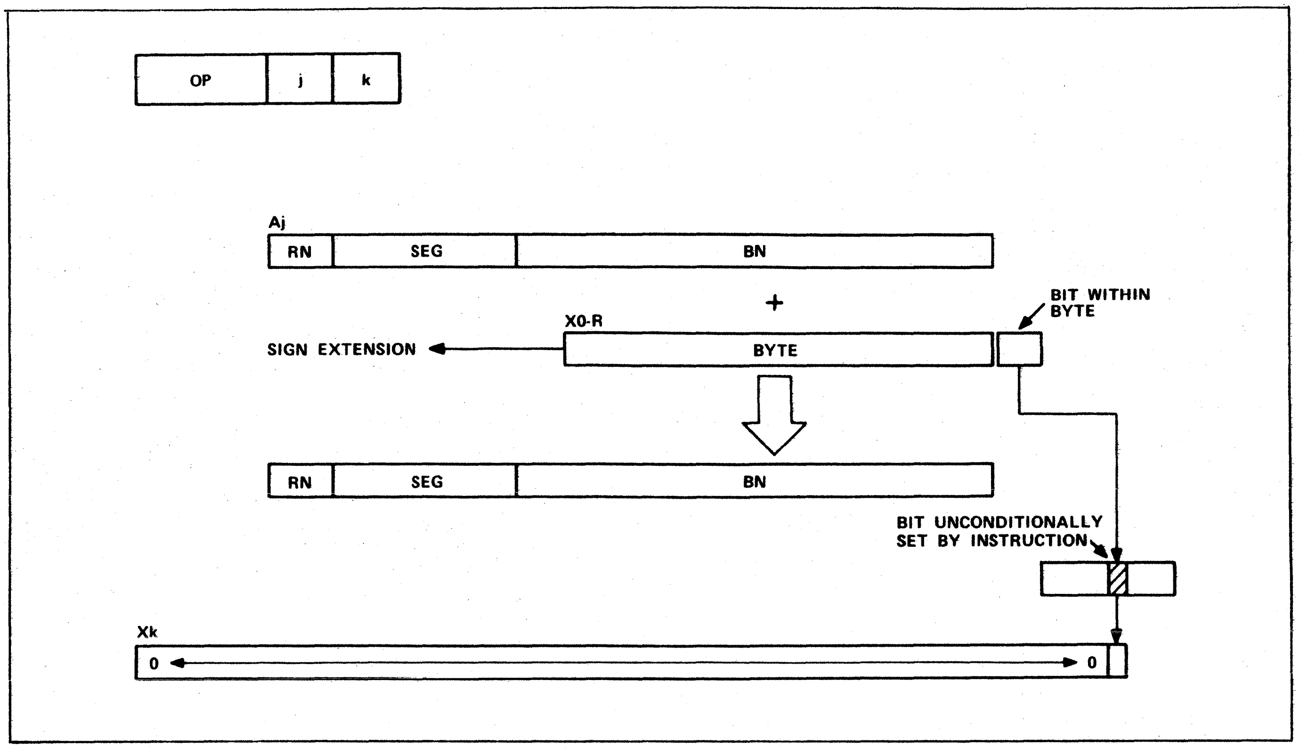


Figure 10-61. Test and Set Bit

This instruction is nonpreemptive. No accesses to the byte containing the interlock bit are permitted from any port on central memory during the execution of this instruction. It also is preserialized and postserialized with respect to this processor as described under Compare/Swap. Note, however, that the instruction unconditionally sets the lock. If it was already set, then action depending on the lock must be postponed. To clear a lock, the store bit instruction, which is described under the general instructions, must be used. Since the Store Bit instruction has a general utility it is undesirable to penalize it by pre- and postserialization. Nevertheless the requirement remains to preserialize when clearing a lock. This may be achieved by issuing a Test and Set Bit instruction before a Store Bit to clear a lock. This resets the already set lock and postserializes thereby effectively preserializing the following instruction.

General Notes:

- The byte containing the interlock bit must reside in a segment which has both Read and Write access since the instruction both loads (into Xk) and stores into the interlock bit.

- Here again cache should be bypassed to ensure that the interlock bit resides in a memory common to all registers. The instruction bypasses cache on read and purges the entry on a write.

- The bit address is formed from the contents of Registers Aj and X0. Aj carries a byte address, X0-Register carries a bit address which is treated as a byte address and a bit address within that byte. The byte address portion is sign extended in its leftmost three bits and added to the Byte Number field of Aj to form the byte address of the interlock byte.

TEST AND SET PAGE

The purpose of this instruction is to give the Real Memory Address (RMA) corresponding to a PVA, provided as an argument to the instruction. If the required page is not in memory, then a flag is returned to indicate this fact. The PVA is initially contained in register Aj and the RMA returned in Xk-Right. When the page is not in real memory Xk-R is returned with the sign bit (bit-32) set. For pages in memory the used bit in the PTE is set. The assumption is that the page will be used almost immediately, and that used pages are aged via a least frequently used algorithm, hence the page tends to be held in real memory.

When the page is not in memory a page fault does not occur. In general, page fault sensing and access violations on Aj do not take place. However, Address Specification errors, and Invalid Segment sensing do occur.

Probably the most prevalent usage of this instruction is to determine an RMA for I/O, since the PP´s in the IOU can only access real memory addresses.

COPY FREE RUNNING COUNTER

There is a one microsecond clock in central memory whose value may be read by this instruction. This clock, termed the Free Running Counter, is classified as a central memory maintenance register, and is the only such register having access via the memory port. The remainder are accessed via the Maintenance Channel (MCH). The Free Running Counter may be written (but not read) from the MCH, and may be read by the CPU via the memory port using this instruction. The counter is a 48-bit register which is read into register Xk, right justified, zero filled. The address of the counter (register B0) is supplied in bits 56-63 of register Xj. In addition, bit 33 of Xj is used to designate which of two memory ports accessible by the processor shall be used.

LOAD PAGE TABLE INDEX

This instruction is included in the repertoire specifically to aid in the management of the page table. Starting with an SVA it determines whether or not a given entry is in the page table, and if so, where it is.

To use this instruction an SVA must first be derived from a PVA by software. This quantity is used by the instruction out of register Xj. The instruction then uses the virtual memory mechanism to search the page table for the required entry. The only difference between the virtual memory mechanism and the instruction is that the instruction ignores the valid bit in the PTE. This is to facilitiate the implementation of the algorithm for clearing continue bits in the page table. This algorithm is described in the section on virtual memory and requires determining whether a given entry hashed directly to the PTE, or was some distance from the direct hash entry.

The instruction returns the index of the PTE in Xk-Right. This is true even when the entry is not found, in which case it returns the index of the last entry searched. X1-Right contains the count of entries searched and a flag (bit-32) to indicate whether the required entry was found (figure 10-62). An investigation of the quantity answers the hash-direct question, and obviates the need for software to simulate the hashing algorithm.

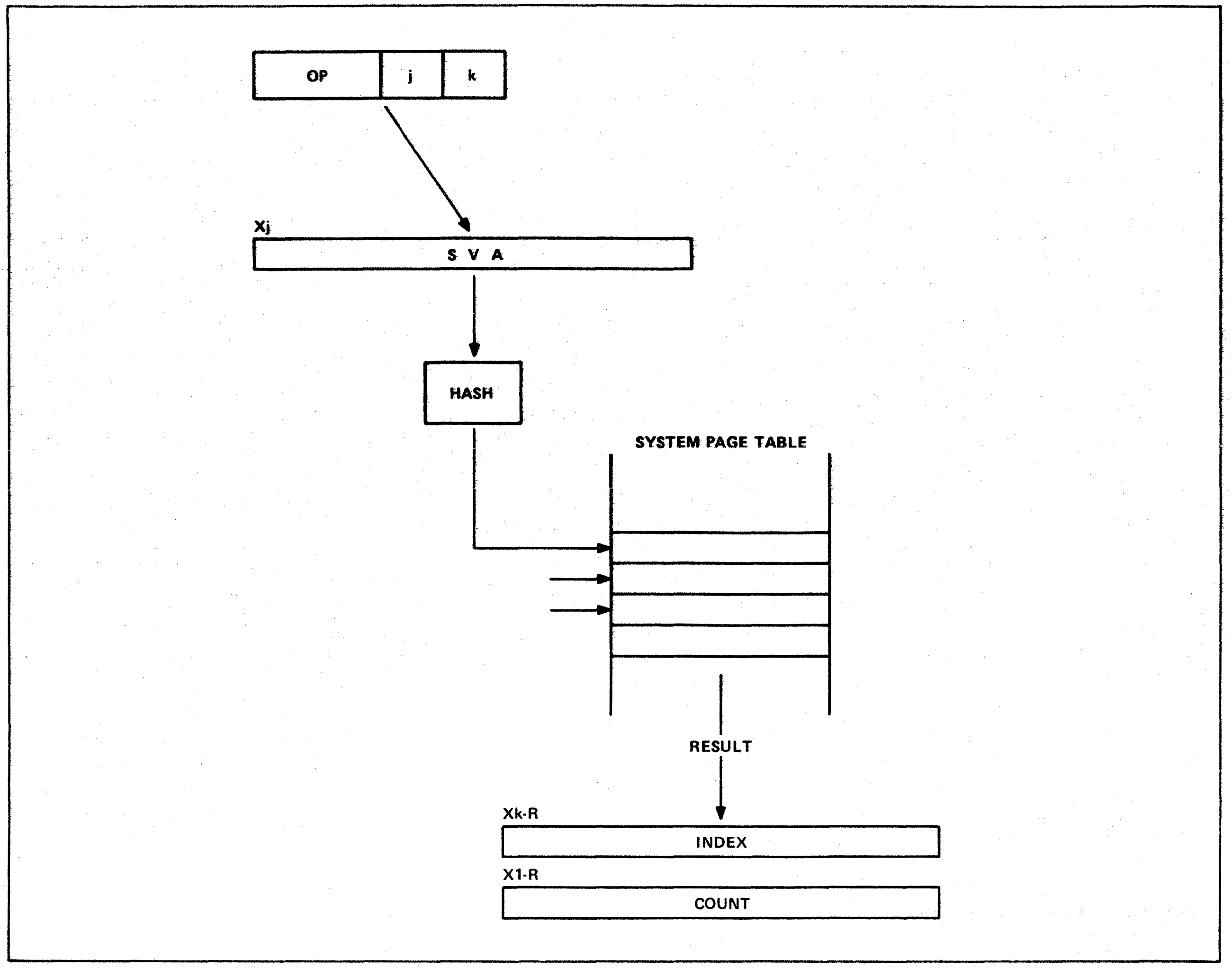


Figure 10-62. Load Page Table Index

The instruction is permitted to execute only when the processor is in Local or Global privileged mode. Although page table management is an operating system function, it is not necessarily prudent to do this management in monitor mode when interrupts would be disabled. Instead, a system job is created which has special privileges; in this instance the job would have at least local privilege.

PROCESSOR INTERRUPT

This instruction provides the capability for one processor to interrupt one or more processors connected to the same memory. The receiving processor has the external interrupt bit (bit-56) set in its Monitor Condition Register and reacts according to the state of that processor. The actual interrupt is routed from memory port to memory port or ports. Register Xk-Right is used by the instruction to determine which memory the interrupts are routed through. Bit 33 of Xk-Right selects one of two memory ports connected to the interrupting processor, and bits 56-63 select the ports to which the interrupt will be sent.

General Notes:

1) Interrupts may be sent to or received from the IOU. When an interrupt is sent to the IOU it is ignored.

2) A software convention must be developed so that the interrupted processor or processors can determine the reason for the interrupt. For this reason it is necessary for the interrupting processor to preserialize the instruction. That is, all memory references on behalf of the interrupting processor are satisfied before the instruction is issued to ensure that the message has been stored successfully before the interrupt is sent.

3) It is possible for a processor to interrupt itself, although this may have limited utility.

4) Examples of the use of this instruction are:

   - One processor changing the state of the world. That is, a processor creates a new page table and switches to its use. A second processor would be idled while the switch was in progress.

   - The IOU wants to alert a processor to the fact that is has completed an I/O operation.

   - The MCU (a special PP in the IOU) wants a processor to perform some maintenance function on its behalf.

These are all examples of system functions. For this reason this instruction carries with it a Global Privilege.

BRANCH ON CONDITION REGISTER

This instruction provides the means for testing, setting and clearing bits in the condition registers. Testing of bits, and modifying bits in the User Condition Register (UCR) are unprivileged operations. However, setting or clearing bits in the Monitor Condition Register (MCR) can only occur from monitor mode.

Although the instruction can be used to test for the presence or absence of a bit in the condition register, the most common application will probably be to test for a bit set, and if set clear it and branch. This provides a mechanism for checking for interrupts without paying the penalty for a trap or an exchange interrupt. Since this instruction can set a bit in a condition register it can cause an interrupt. When the bit is set it will appear to the hardware that the condition arose. Whenever a bit is set a branch exit is taken, and it is this branch address which is saved in the exchange package or stack frame save area as the interrupt occurs. A subsequent exchange or return instruction will then cause processing to resume at the branch address. The ability to set a bit in a condition register has been included primarily for diagnostic purposes. However, no special privileges apply to the UCR since a user cannot cause wanton destruction of the system through that register. Setting a bit in the UCR will cause an interrupt, providing the corresponding bit is set in the User Mask Register and traps are enabled. Similarly for the MCR except for the two flags which are held in that register. These are the System Call (bit-58) and Trap Exception (bit-63). If either of these bits is set in the MCR then there will not be an interrupt, regardless of the state of the machine and the value of the corresponding bits in the MM.

COPY

The copy instructions provide the means for copying state registers (both processor and process state registers) to X Registers and vice versa. Most state registers are numbered uniquely such that the register number serves as an address, both for the copy instructions and for access of the registers over the Maintenance Channel. The copy instructions function by taking the address of the state registers from the last eight bits of register Xj, and then copying to/from register Xk. The value in Xk will always be right justified and zero filled. The following general notes are of interest:

- Not all state registers are available to the processor. For example, the Status Summary register can only be accessed via the MCH - and then only in a Read Mode.

- Except for those registers which cannot be accessed the Copy to Xk instruction is nonprivileged.

- The Copy from Xk instruction, that is, write to a state register, carries various privileges depending on the address of the state register. An attempt to write a state register from a segment with the wrong privilege causes an interrupt.

- Attempts to write a read-only register or to write a nonexistent register act as no operations.

- Attempts to read a nonexistent register result in all zeros being returned.

- Certain process state registers (defined in the exchange package) do not have addresses, and cannot be read/written via the copy instructions. In fact, these registers can only be set via the exchange jump mechanism. Figure 10-63 illustrates the accessibility to the registers.

REGISTER GROUP	REGISTER NUMBER(S)	REGISTER NAME	COPY ACCESS PRIVILEGES		MCU ACCESS
			COPY FROM STATE REGISTER	COPY TO STATE REGISTER	
			READ	WRITE	
00-0F	00	STATUS SUMMARY	NO ACCESS	NO ACCESS	R
10-1F	10	ELEMENT ID	UNPRIV.		
	11	PROCESSOR ID			
	12	OPTIONS INSTALLED			
	13	VIRTUAL MACHINE CAPABILITY LIST			
20-2F 30-3F	21	PMF KEYPOINT	NO ACCESS	NO ACCESS	R/W
	22	PMF BUFFER			
	30	DEPENDENT ENVIRONMENT CONTROL			
	31	CONTROL STORE ADDRESS			
	32	CONTROL STORE BREAKPOINT			
40-4F 50-5F	40	P REGISTER	UNPRIV.	NO ACCESS	R/W
	41	MONITOR PROCESS STATE POINTER			
	42	MONITOR CONDITION REGISTER			
	43	USER CONDITION REGISTER			
	44	UNTRANSLATABLE POINTER			
	45	SEGMENT TABLE LENGTH			
	46	SEGMENT TABLE ADDRESS			
	47	BASE CONSTANT			
	48	PAGE TABLE ADDRESS			
	49	PAGE TABLE LENGTH			
	4A	PAGE SIZE MASK			
	50	MODEL DEPENDENT FLAGS			
	51	MODEL DEPENDENT WORD			
60-6F 70-7F	60	MONITOR MASK REGISTER	UNPRIV.	MONITOR	R/W
	61	JOB PROCESS STATE POINTER			
	62	SYSTEM INTERVAL TIMER			
80-8F 90-9F A0-AF B0-BF	80-8F	PROCESSOR FAULT STATUS	UNPRIV.	GLOBAL	R/W
	90	RETRY CORRECTED ERROR LOG			
	91	CONTROL STORE CORRECTED ERROR LOG			
	92	CACHE CORRECTED ERROR LOG			
	93	MAP CORRECTED ERROR LOG			
	A0	PROCESSOR TEST MODE			
C0-CF D0-DF	C0-C3	TRAP ENABLES	UNPRIV.	LOCAL	R/W
	C4	TRAP POINTER			
	C5	DEBUG POINTER			
	C6	KEYPOINT MASK			
	C7	KEYPOINT CODE			
	C8	KEYPOINT CLASS NUMBER			
	C9	PROCESS INTERVAL TIMER			
	CA-CB	KEYPOINT ENABLE FLAG			
E0-EF F0-FF	E0-E1	CRITICAL FRAME FLAG	UNPRIV.	UNPRIV.	R/W
	E2-E3	ON CONDITION FLAG			
	E4	DEBUG INDEX			
	E5	DEBUG MASK REGISTER			
	E6	USER MASK REGISTER			

Figure 10-63. Processor Register Definitions and Accesses

- Various flip-flops may be set by the copy instructions and some special provisions for these have been made. When a state register is read, it is always possible address Xj and the destination Xk to be the same register. When writing, however, two registers are usually used. For the single-bit registers multiple addresses have been supplied such that a single register can act as both the address and data. For example, E0 and E1 are two addresses for the critical frame flag. A copy to CFF with a value of E0 will clear the CFF, and with E1 will set it (figure 10-64). This concept has been carried even further for the Trap Enable Flip-flop (TEF) and the Trap Enable Delay (TED), when four addresses are reserved for the combined two registers which may be set simultaneously by a single copy instruction (figure 10-64). This is a very important fact when the process of enabling traps after a trap interrupt is considered. Traps are enabled when the TEF is set, and the TED is clear. When a trap interrupt occurs the TEF clears. To both the TEF and the TED are set and a Return issued. The setting of the two flags can take place in one instruction.

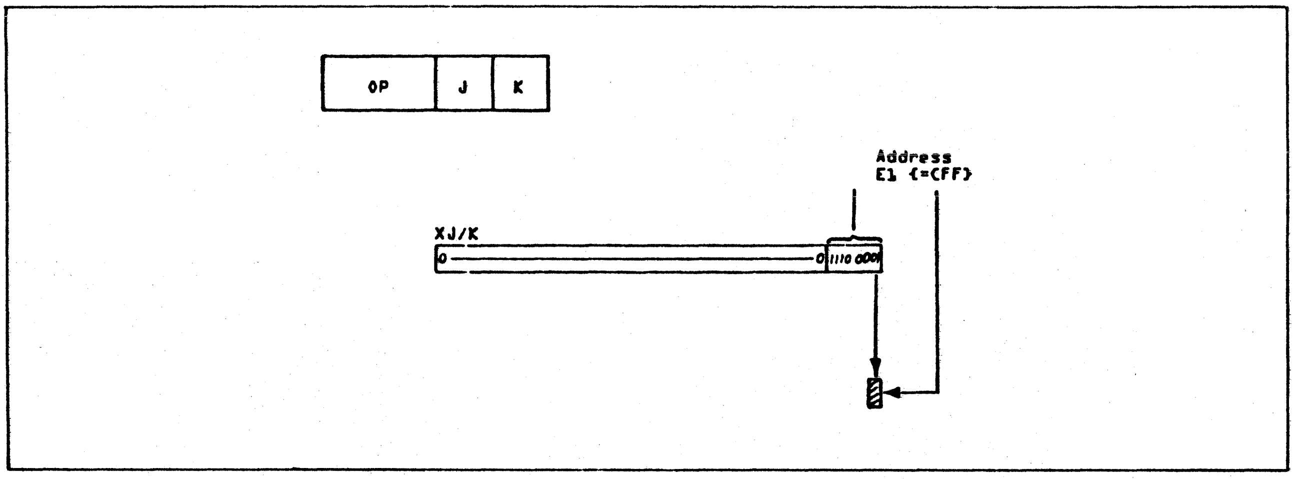


Figure 10-64. Copy to Single Bit Register

PURGE BUFFER

The cache and MAP buffers, present on certain CYBER 180 processors, were described in an earlier section. They are fast buffer memories designed to give rapid access to frequently used data, and contain copies of data held in central memory. Whenever the copy and the base data are different, the copy is said to be stale, and must be purged from the buffer in question. Some purging is provided automatically by the hardware, but there are many instances where the onus is placed on the software to ensure that the appropriate buffer or buffers are purged. The Purge Buffer instruction has been provided for this purpose. The instruction has several options which are controlled by the k-field of the instruction.

These include purging either the MAP or the cache, and either totally or selectively by an SVA or a PVA. Purging the cache via a PVA is an unprivileged instruction; all other forms of the instruction carry a Local privilege. This is to protect against a user purging entries in another user´s address space. Technically, such an act would not constitute a violation of the security in the system, but it could degrade the performance of another job.

Selective purging in the cache is for all entries in a 512-byte block, and all entries in a segment. For the MAP, all entries for a page or for a segment may be purged.

The instruction preserializes and postserializes. This is simply to ensure that there are no outstanding memory requests from this processor when the buffer is purged, and that an outstanding request issued before the purge, does not complete after the purge.

For those models which do not have cache or MAP buffers this instruction is treated as a no operation.

Care should be exercised deciding when to use the Purge Buffer instruction and deciding how much to purge. If no data or too little data is purged results could be fatal and erratic. If too much is purged, performance could be affected. Certainly the latter is less serious and an initial version of an operating system erring in this direction probably would be acceptable. The main point is to be aware of those times when stale data is likely to accumulate in a buffer memory. These are described in the following paragraphs.

Cache

a) Whenever an ASID is reassigned

b) In a multiprocessor environment, whenever one processor modifies a page which is used (shared) by a second processor

Cache memory contains a copy of the most recently used words (code and data in most models) in system virtual memory. Consequently, software must ensure that the cache always reflects accurately what is in system virtual memory. Since virtual memory is represented by the file system on CYBER 180, real memory acts as a cache to virtual memory. When a page is reassigned in real memory the image of the old page in virtual memory does not change. As a result, cache need not be purged. By organizing cache memories as system virtual addresses, the number and frequency of necessary purges are minimized.

MAP

a) Whenever a used or valid bit is cleared in the page table

b) Whenever a segment´s attributes change

c) Whenever an ASID is deleted

In general, the Segment MAP contains a copy of the SDE and the Page MAP contains a copy of the PTE. Hence, whenever anything in an SDE or a PTE changes the MAP should be purged.

EXECUTE ALGORITHM

The Execute Algorithm instruction is a device for reserving an op code for model dependent instructions. For example, a customer may buy a pop. count instruction similar to that available on CYBER 170. On a machine having no special instructions which utilize this feature, execution of the instruction results in an unimplemented instruction error. In addition, op. codes BE and BF have been reserved for customer use, such as for software simulation of instructions which do not exist within the CYBER 180 instruction repertoire.

PROGRAM ERROR

An operation code consisting entirely of zeros has been reserved. Execution of the instruction yields an instruction specification error. The idea behind the reservation is that when programs run away and start executing data and so forth, it is likely that zero bytes will be encountered - hence the condition.

SCOPE LOOP SYNC

Execution of this instruction triggers a signal at an external test point which is suitable for synchronizing test equipment. Systems with a central memory refresh counter also issue a refresh counter resynch function, followed by a read of word 0 of the current C180 exchange package. Systems without central memory refresh only provide the external test point signal.

DEBUG

CYBER 180 processors provide a debug facility which assists programmers debugging at the machine code level in C180 State. The operation of debug is fairly complex since, in affect, it provides an interrupt capability during an instruction execution. In practice, the instruction is not executed until the debug processing is complete, although prevalidation of the instruction may complete prior to the debug. As a result of this flexibility the state of the process must be retained across interrupts, and several process state registers have been defined for this purpose.

The user may elect to debug based on a number of conditions. These are:

- Whenever data is read from a specified area in virtual memory
- Whenever data is written into a specified area in virtual memory
- Whenever an instruction is fetched from a specified area in virtual memory
- Whenever a branch is made to a specified area in virtual memory
- Whenever either a CALL INDIRECT or a CALL RELATIVE instruction is issued to a procedure in a specified area in virtual memory

These constitute five debug conditions and for any instruction issued the user may elect to debug on any combination of these conditions, for up to 32 different areas in virtual memory. The conditions are specified in a debug list (figure 11-1), which is provided by the user, and further controlled by the Debug Mask Register (DM). Finally, debugging is activated by setting bit 56 in the User Mask Register (UM) and enabling traps.

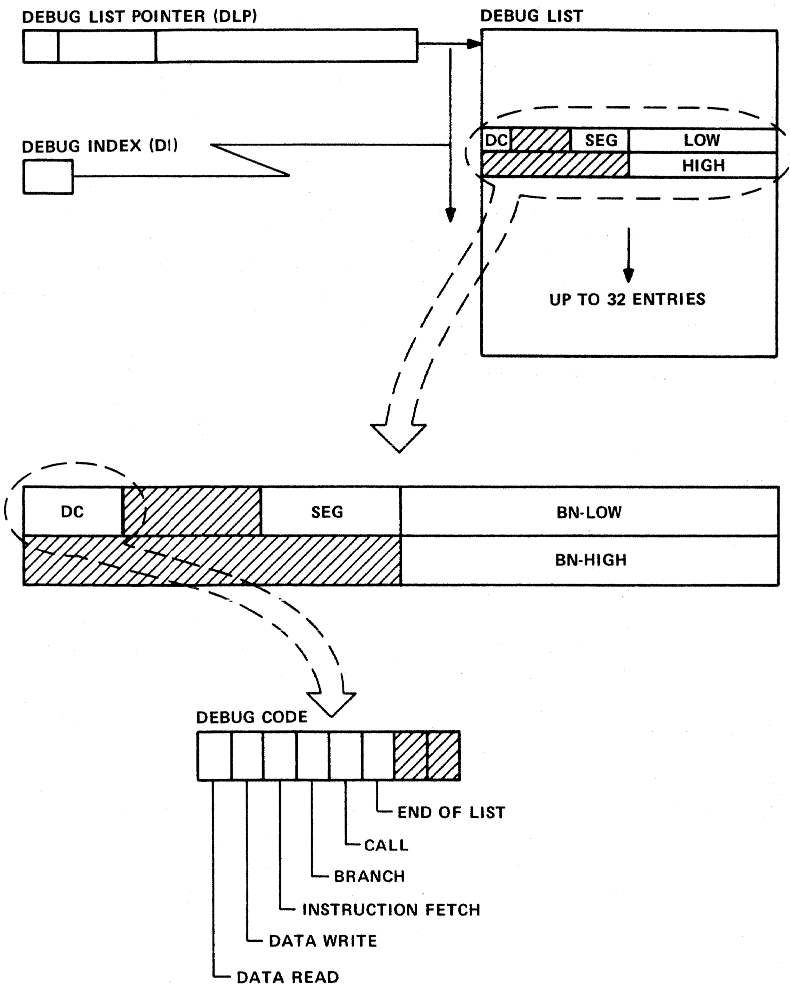


Figure 11-1. Debug List

Each entry in the debug list consists of two words, which must be placed on word boundaries. The two words describe:

> The debug conditions; the segment in virtual memory to which the conditions apply; and the range of addresses (byte numbers) in the segment to which the debug conditions are restricted (figure 11-2).

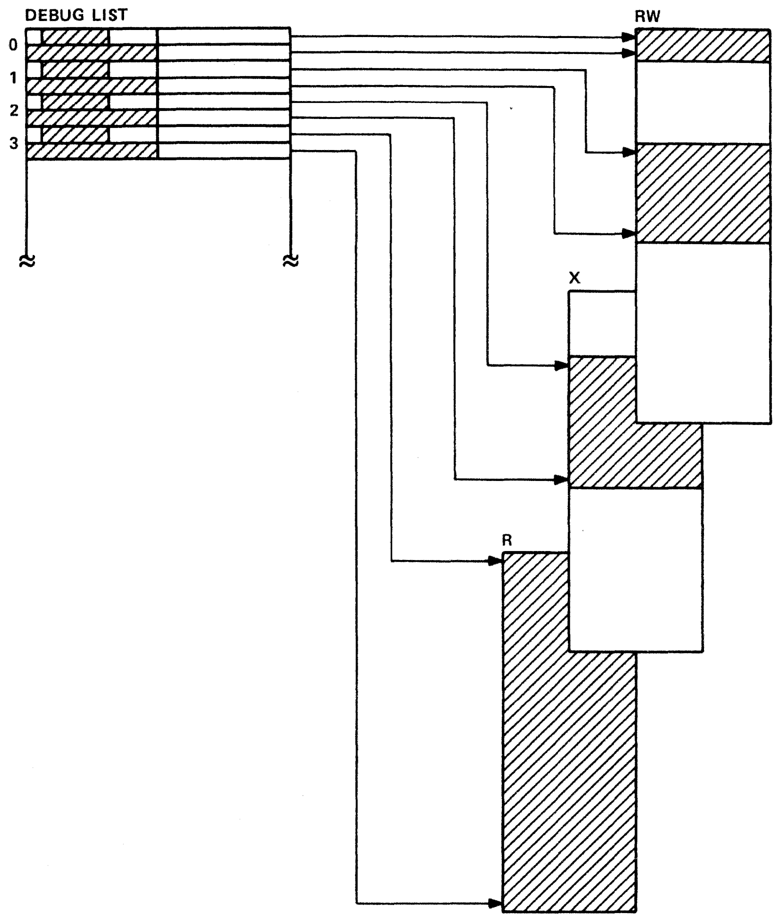


Figure 11-2. Debug List Entries

The Debug Code (DC) may be selectively activated at run time, by setting the DM appropriately. Refer to figure 11-3. For each condition set in the DC there is a corresponding condition select in the DM. Debugging occurs only when there is a coincidence between a condition bit in the DC, and a condition select bit in the DM.

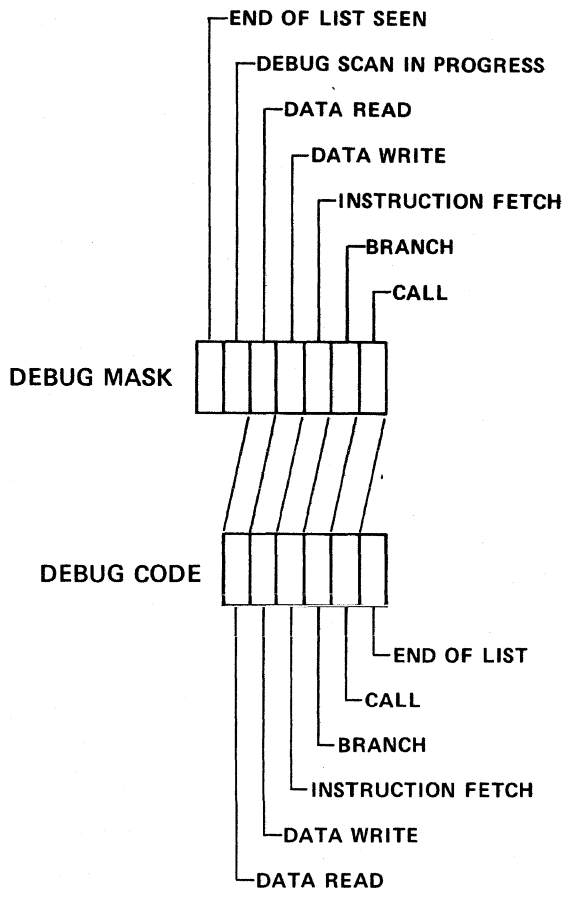


Figure 11-3. Debug Condition Select

A user may insert a debug list into his program, set the DM to select a range of conditions, but run in a normal mode by not choosing to trap on debug. In this case the overhead due to debugging is zero. Performance degradation due to debug testing will occur whenever the first two items in the following list are true; however, a debug trap will not occur unless all of the following are true:

1) Traps are enabled.

2) Bit-56 in the UM is set (debugging selected).

3) The process is C180.

4) The DM and DC registers both select a test that is satisfied for the current instruction.

5) The end of list seen flag in the DM is not set.

When these conditions are met the debug list is scanned and trap interrupts taken, as required, by the hardware setting bit 56 in the UCR.

The hardware utilizes three process state registers to control scanning of the debug list. These are: the debug list pointer (DLP); the debug index (DI); and the DM. The DLP gives the starting address of the debug list, the DI keeps track of the position within that list, and the DM contains two flags to control the initiation and termination of the debug. The first of these flags is the Debug Scan in Progress flag which controls the start of the process. Conceptually, whenever an instruction is executed this flag is cleared. Then when the next instruction is issued and debug is active, the processor starts scanning the debug list from the beginning. The second flag is the End of List Seen flag, and controls the termination of debugging for the current instruction. This flag is set when either 32 entries in the debug list have been scanned, or when an end of list bit is encountered in a debug code. Again, conceptually, the flag is cleared whenever an instruction is executed. The complete, conceptual, hardware process is shown in figure 11-4. These flags are primarily hardware flags which have been included in a process state register so that the hardware can remember where it is when an interrupt is taken. If they are set by software they could perturb the operation of debug.

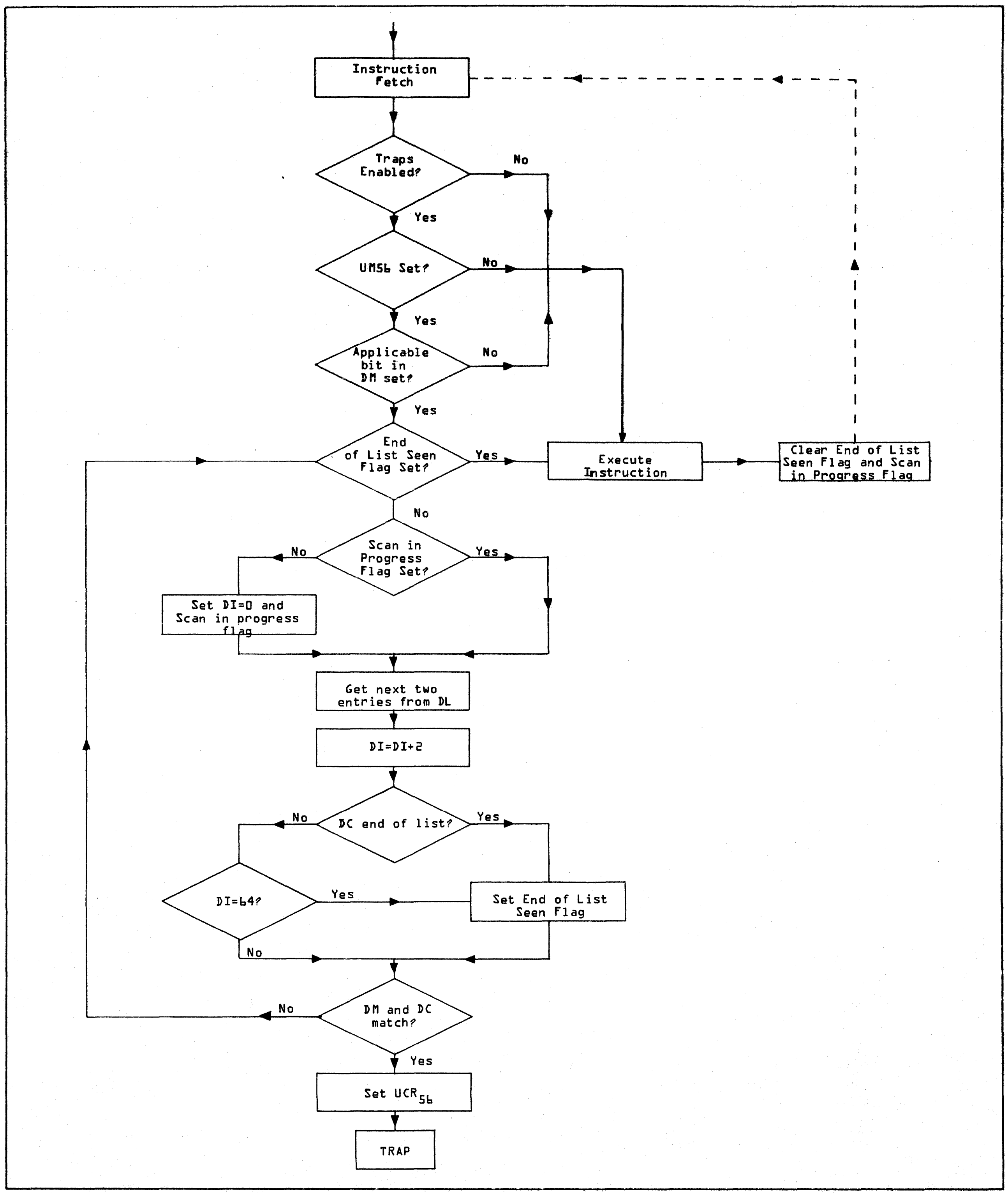

Instruction Fetch
Traps Enabled?
No
Yes
UM56 Set?
No
Yes
Applicable bit in DM set?
No
Yes
End of List Seen Flag Set?
Yes
Execute Instruction
Clear End of List Seen Flag and Scan in Progress Flag
No
Scan in Progress Flag Set?
No
Yes
Set DI=0 and Scan in progress flag
Get next two entries from DL
DI=DI+2
DC end of list?
No
Yes
DI=64?
Yes
Set End of List Seen Flag
No
DM and DC match?
No
Yes
Set UCR56
TRAP

Figure 11-4. Conceptual Debug Procedure

General Notes:

1) Debugging only occurs when traps are enabled. Hence, the interrupt handlers cannot make use of this facility. However, monitor mode code can make use of debug - providing traps are enabled.

2) Debug list entries beyond the 32nd are ignored regardless of whether or not an end of list seen flag has been encountered in the DC.

3) The user must have been granted local privilege in order to alter the DLP. In other words, local privilege is required to specify a different debug list in the same process.

4) Several instructions apply to more than one debug condition. For example, many BDP instructions can trap on both Read and Write since they have both source and destination operands in memory. One instruction, CALL INDIRECT, applies to four debug conditions (it is a CALL, it reads from the Binding Section, it writes into the Stack Frame Save Area, and it is fetched).

5) Also, some instructions have more than one operand which is checked for a debug trap. Typically, these are instructions which specify the address of a table to be used in conjunction with two operands (for example, Translate and Edit).

6) Exchange jumps, which branch to a value found in an exchange package at a real memory address, do not cause a debug trap on branch. Similarly, Compare and Swap does not cause a debug trap on branch when it rejects as a result of a hardware lock set.

7) When a debug trap occurs the P Register stored in the Stack Frame Save Area points to the instruction which would have been executed had the debug trap not been taken. Also when a debug trap is taken, the DI will have an odd value. That is, it will point to the second word of a word pair entry in the debug list. However, while the debug list is being scanned, interrupts are enabled, and should an asynchronous interrupt occur, such as PIT, and External Interrupt, then the DI may be either odd or even.

This section gives a brief overview of the system software for Control Data´s CYBER 180 program. Included is a discussion of the operating system and product set. Specifically excluded is any discussion of maintenance software.

OPERATING SYSTEM (NOS/VE)

The internal name for the new operating system for CYBER 180 native state is NOS/VE.

PRODUCT SET

LANGUAGE PROCESSORS

The initial release of CYBER 180 software will include two user languages: FORTRAN and COBOL. Subsequent releases will contain the following languages: APL, ALGOL-60, ALGOL-68, BASIC, PASCAL, PL/1 and JOVIAL.

FORTRAN

CYBER 180 FORTRAN is a reimplementation in CYBIL of FORTRAN 5, which is an existing CYBER 170 FORTRAN product written in assembly language.

COBOL

CYBER 180 COBOL is a reimplementation in CYBIL of the existing CYBER 170 COBOL 6.

SUPPORT SERVICES

The CYBER 180 support services include Sort/Merge, Basic and Advanced Access Methods, and Loader. The Basic Access Methods and the loader are parts of the NOS/VE operating system.

Sort/Merge

CYBER 180 Sort/Merge is a reimplementation in CYBIL of the CYBER 170 SORT 5 Product.

Data Management

Data Management for CYBER 180 will be provided by the Information Management Facility (IMF180) and is based on the existing DMS170 and also on the European Data Management System (EDMS). The first release of this product will be with the second release of the operating system.

USER INTERFACES

There are two levels of user interface on CYBER 180. The command interface is the view of CYBER 180 as seen by all users from interactive or batch jobs. It is analogous to the control card interface on CYBER 170. The program interface is the view of CYBER 180 as seen by all programmers: system programmers, application programmers, and user programmers. It is analogous to the macro interface on CYBER 170.

COMMAND INTERFACE

The key design criteria for the Command Interface is to provide a usable consistent interface across all modes of access to the system by all users, operators or system maintenance personnel. A major factor in achieving consistency is the use of a System Command Language (SCL) which provides a common syntax, control statements and procedure mechanism that are used by all commands and command utilities.

NOS/VE Command Summary

In order to give an overview of the capabilities of NOS/VE, a list of some of the available commands and their more important general characteristics is provided.

System Access

All users that access the system must have been previously registered as valid users of the system by an installation administrator and must identify themselves each time they use the system in interactive or batch mode. The installation administrator can organize users into groups called families and within families into groups according to account and project.

System access command summary:

```
LOGIN
LOGOUT
SET_PASSWORD
```

File System

All users have a master catalog associated with them which is conventionally given the same name as their user name. The master catalog can contain an arbitrary number of files and subcatalogs. Each subcatalog can also contain an arbitrary number of files or subcatalogs which allows users to build their own hierarchy of files.

Each file consists of data and a set of attributes which describe the data that are used to cause the data in the file to be properly interpreted by programs accessing the file. Both data and its attributes are maintained in the catalog or subcatalog in which the file is located.

File system command summary:

```
CREATE_FILE
DELETE_FILE
CREATE_CATALOG
DELETE_CATALOG
CREATE_FILE_PERMIT
DELETE_FILE_PERMIT
CREATE_CATALOG_PERMIT
DELETE_CATALOG_PERMIT
ATTACH_FILE
DETACH_FILE
DISPLAY_FILE
SET_FILE_ATTRIBUTES
DISPLAY_FILE_ATTRIBUTES
CHANGE_FILE_ATTRIBUTES
COPY_FILE
COMPARE_FILE
```

Job Management

A job is the basic mechanism for organizing work to be performed by the system. A job runs on behalf of a single user and is used as the accounting envelope and unit of scheduling.

Job management command summary:

```
SUBMIT_JOB
TERMINATE_JOB
PRINT_FILE
DISPLAY_JOB_STATUS
DISPLAY_PRINT_STATUS
JOB/JOBEND
```

Resource Management

Resource management command summary:

```
RESERVE_RESOURCE
RELEASE_RESOURCE
REQUEST_MAGNETIC_TAPE
REQUEST_TERMINAL
SET_JOB_LIMIT
```

Program Execution Commands

Program execution is the process of combining and executing a number of separately produced modules. It is the basic means by which users perform the work they require.

Program execution command summary:

```
EXECUTE_TASK
name call
SET_PROGRAM_ATTRIBUTES
DISPLAY_PROGRAM_ATTRIBUTES
```

Program Compilation Commands

Program compilation command summary:

FTN
COBOL
CYBIL

SCL Procedure and Control Commands

In addition to providing a consistent syntax for all system supplied commands, SCL also provides the facility for users to extend the system in a consistent manner. This is accomplished by the SCL procedure capability.

SCL procedure and control command summary:

```
PROC
PROCEND
procedure name call
INCLUDE_FILE
EXIT_PROC
IF/ELSE/ELSEIF/IFEND
FOR/FOREND
LOOP/LOOPEND
WHILE/WHILEND
REPEAT/UNTIL
WHEN/WHENEND
CREATE_VARIABLE
DELETE_VARIABLE
```

Command Utilities

SOURCE CODE MAINTENANCE
EDITOR
OBJECT CODE MAINTENANCE
INTERACTIVE DEBUGGER (SYMBOLIC FOR CYBIL ONLY)
ACCOUNT, PROJECT, MEMBER AND USER ADMINISTRATION

PROGRAM INTERFACE

User and application programmers access the system primarily via FORTRAN and COBOL. User programs written in these standardized languages should migrate easily from CYBER 170 to CYBER 180. System programmers access the system via CYBIL, the implementation language for CYBER 180. Most of NOS/VE will be written in CYBIL, with assembly language only being used either to take advantage of specific machine capabilities that are otherwise unavailable or to achieve high performance in critical areas.

The program interface to NOS/VE is accessed via CYBIL procedure calls. The parameters and data structures conform to the CYBIL rules for variables, constants and types. Therefore, in order to understand the program interface to the system, a working knowledge of CYBIL is required.

CYBIL

CYBIL, the implementation language for CYBER 180, is derived from the programming language PASCAL developed by Dr. Niklaus Wirth of Switzerland. The four major declarations of CYBIL are constants (1, 2... A, B, etc.), variables (cells in memory that contain a value), procedures, and types. The concept of type, which was popularized by Dr. Wirth, allows a programmer to declare his own types of data. This appears frequently in the NOS/VE interfaces, where type declarations define the values the operating system procedures operate on.

Declarations may be specified globally (essentially allocated at load time) or within a procedure (allocated when a procedure is called), which is the general concept of block structure.

BASIC SYNTAX

The module is the basic unit of compilation in a CYBIL program.

```
MODULE<module name>
 <declaration>
 <declaration>
 <declaration>
    .  .  .
MODEND<module name>;
```

It is essentially a list of global declarations, where each declaration is either a constant, variable, procedure, or type declaration.

The format for a procedure declaration follows:

```
PROCEDURE(<attributes>)<procedure name>(<parameter definition>)
 <declaration>
 <declaration>
 <declaration>
    .  .  .
 <statement>
 <statement>
 <statement>
    .  .  .
PROCEND<procedure name>;
```

This includes a list of declarations local to the procedure (constant, variable, procedure or type) followed by a series of statements. These declarations are activated each time this particular procedure is called and deactivated when the procedure returns, which is part of the block structuring capability of CYBIL. The statements available in CYBIL are typical high-level language statements such as assignment, IF THEN ELSE, FOR, WHILE, and procedure reference.

TYPE DECLARATIONS

The use of programmer-defined types denotes the permissible values that a variable may assume, and the structuring method of the variable. Because these type declarations are so explicit, a certain amount of internal checking such as access verification is possible at compile time. Hence, many errors which have traditionally been found at execution time are detected by the compiler at compile time. When programming in CYBIL, it is a common experience for a program to execute as soon as an error-free compilation has been obtained, even though it might yield incorrect results.

Fixed Types

- INTEGER (64 bits)
- CHARACTER (8-bit ASCII character)
- ORDINAL
- BOOLEAN (logical operator)
- SUBRANGE
- POINTER

Fixed type declarations are built into the language. The Ordinal type is a delineated list of constants that is a symbolic way of representing values. With the Subrange type, the programmer can specify a subset of permissible values for an integer variable. The Pointer is a dereferenced version of any one of the fixed, structured or adaptable types.

Structured Types

- SETS
- STRINGS
- ARRAY
- RECORD

Structured type declarations enable the programmer to define his own types. Sets define a series of values that are either present or absent. The Record type, used frequently in NOS/VE documentation, is a series of fields each of predefined or user defined type. Such a recursive type definition is used to build arbitrary and unique structures.

Storage Types

- HEAP
- SEQUENCE

A Heap is a memory data structure that is used for the random allocation and deallocation of other variables at execution time. A Sequence is used for the sequential allocation and/or accessing of other variables at execution time.

Adaptable Types

- ADAPTABLE STRING
- ADAPTABLE ARRAY
- ADAPTABLE RECORD
- ADAPTABLE HEAP
- ADAPTABLE SEQUENCE

Adaptable type declarations are those whose bound is determined at program execution. This applies to variable-length strings or arrays, or to records which contain variable-length strings or arrays. Adaptable heap and sequence refer to blocks of memory whose sizes are determined at execution time.

EXAMPLES OF DECLARATIONS

Type Declarations

The following module of CYBIL code shows some examples of type declarations (figure 12-1), and how they relate to variable and procedure declarations.

```
module type_declarations_example;

type
   ordinal_example = (attached, opened, closed, detached);
type
   record_example = record
      o:   ordinal_example,
      i:   integer,
      b:   boolean,
   recend;
type
   array_type_example = array [1 .. 10] of record_example;

{No memory allocated yet}

var
   i1:   integer,
   i2:   integer,
   b1:   boolean,
   b2:   boolean,
   a1:   array_type_example,
   a2:   array_type_example;

procedure example;
   a1[3].o := opened;
   a2[3].o := attached;
procend example;

modend type_declaration_example;
```

Figure 12-1. TYPE Declarations

The first example on lines 7 and 8 is an ordinal type declaration. This particular ordinal type has four values: attached, opened, closed or detached. The other two examples on lines 9 through 16 are both structured type declarations. The record type has three fields: o, i, and b. The o-field is of the type ordinal example, which means it may be attached, opened, closed, or detached (see line 8). The i-field is an integer type, and the b-field is a Boolean type. The third declaration (lines 15 and 16) is an array type, which is defined as an array of 1 to 10 of the user-defined type record example (refer to line 10).

These three examples show how types may be cascaded to form arbitrarily complex structures. The program does not allocate memory, but simply defines three kinds of values that are associated with the variables shown in lines 20 through 26.

The key word var in line 20 indicates the allocation of variables. In this instance there are six variables of three different types. The first two, i1 and i2, are both integer types. The next two, b1 and b2, are both boolean types. The last two variables, a1 and a2, are both array types as defined by the user in line 16.

The procedure declaration in line 29 specifies that the value of o-field of the third element in array a1 be set to opened. Line 30 specifies that the o-field of the third element of array a2 be attached. Both of these declarations are consistent with the list of permissible ordinal values specified in line 8.

CYBIL Declarations

The following module of CYBIL code shows examples of all declarations (figure 12-2), how they may be nested, and what sort of range-checking is done by the compiler.

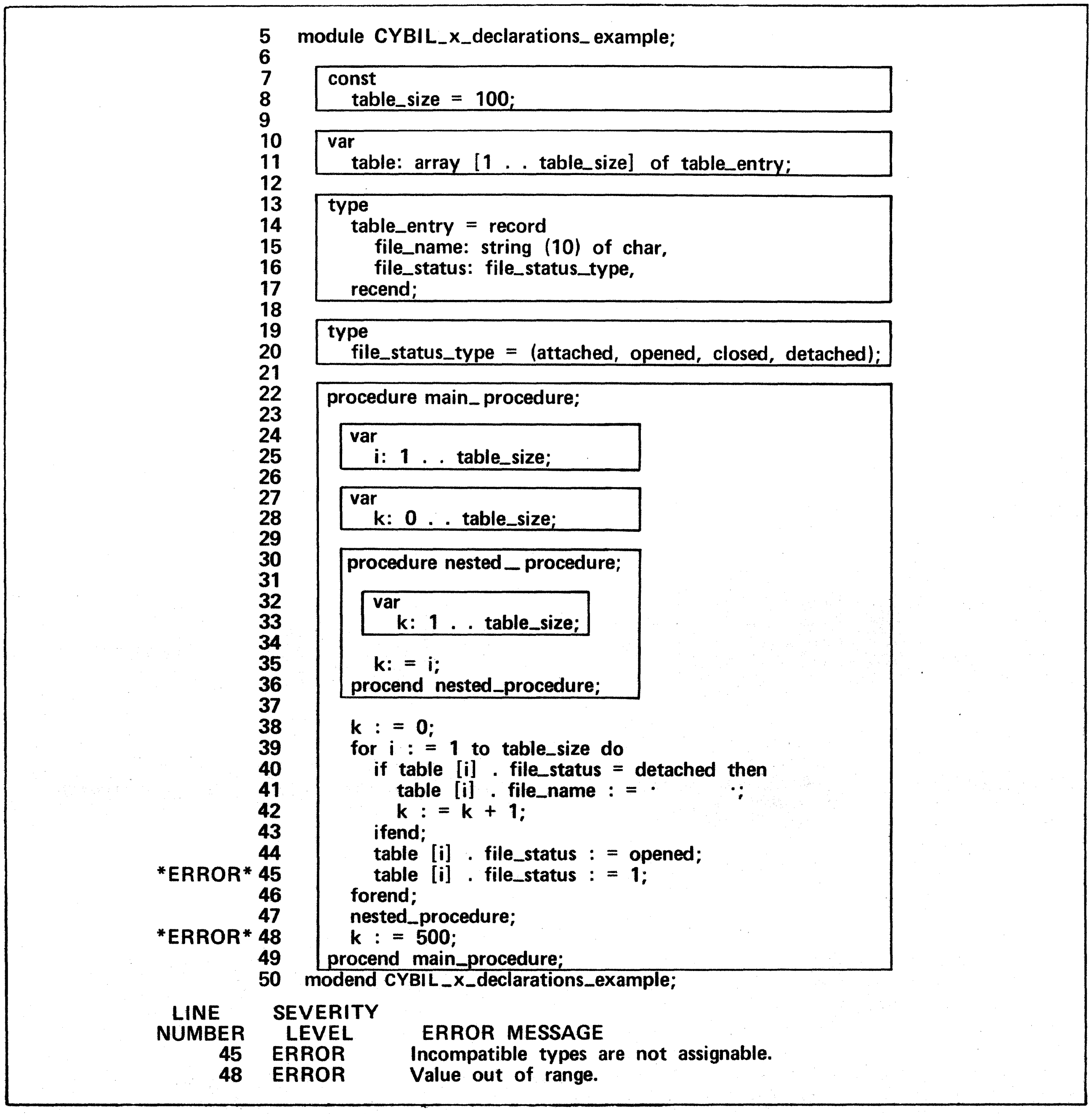
This module contains five major declarations: a constant (lines 7 and 8) a variable (lines 10 and 11), two types (lines 13 through 17, and lines 19 and 20), and a procedure (lines 22 through 49).

Within the procedure declaration are three more declarations: two variables (lines 24 and 25, and lines 27 and 28), and a nested procedure (lines 30 through 36). These declarations are followed by a series of statements (lines 38 through 48). Within the nested procedure are a variable declaration (lines 32 and 33), and one statement (line 35).

In line 44, table (i) is a variable defined in line 11 as an array of the type specified by the record in lines 14 through 17. The second field in that record (line 16) is file status, which is the ordinal type specified in line 20. Thus, in line 44, when the file status of table (i) is designated as opened, the compiler accepts this statement as valid because opened is one of the ordinal values declared in line 20. In line 45, however, the compiler rejects as an error the assignment of 1 (which is the internal data mapping for opened). This makes the code more readable by minimizing unnecessary references to other documentation, in this case to look up the definition of 1.

Another compiler check is shown in line 48 where the subrange k is assigned a value of 500. According to line 28, however, the variable k can assume any value from 0 to the constant table size, which is specified as 100 in line 8. As a result, 500 is beyond the range of 0 to 100, and the compiler rejects this statement as an error.

These and other features of CYBIL make it possible to do a large amount of program debugging at compile time.



```
          5   module CYBIL_x_declarations_example;
          6
          7     const
          8       table_size = 100;
          9
         10     var
         11       table: array [1 . . table_size] of table_entry;
         12
         13     type
         14       table_entry = record
         15         file_name: string (10) of char,
         16         file_status: file_status_type,
         17       recend;
         18
         19     type
         20       file_status_type = (attached, opened, closed, detached);
         21
         22     procedure main_procedure;
         23
         24       var
         25         i: 1 . . table_size;
         26
         27       var
         28         k: 0 . . table_size;
         29
         30       procedure nested_procedure;
         31
         32         var
         33           k: 1 . . table_size;
         34
         35         k: = i;
         36       procend nested_procedure;
         37
         38       k : = 0;
         39       for i : = 1 to table_size do
         40         if table [i] . file_status = detached then
         41           table [i] . file_name : = '          ';
         42           k : = k + 1;
         43         ifend;
         44         table [i] . file_status : = opened;
*ERROR*  45         table [i] . file_status : = 1;
         46       forend;
         47       nested_procedure;
*ERROR*  48       k : = 500;
         49     procend main_procedure;
         50   modend CYBIL_x_declarations_example;

  LINE    SEVERITY
 NUMBER    LEVEL        ERROR MESSAGE
     45   ERROR       Incompatible types are not assignable.
     48   ERROR       Value out of range.
```

Figure 12-2. CYBIL Declarations

CYBER 180 OPERATING SYSTEM (NOS/VE)

SYSTEM STRUCTURE

NOS/VE is designed to run predominately in the central processor. There are several reasons for this. One is to take advantage of higher level implementation languages and lower central memory costs in order to increase the productivity of the software development process. Another aspect is the greater reliability derived from protection and security mechanisms such as virtual memory translation which are bypassed by the peripheral processors. NOS/VE maintainability is also enhanced if its code is not dispersed in the peripheral processors. In addition, localizing NOS/VE in the central processor will eliminate the architectural necessity of including peripheral processors in successor product lines.

The virtual memory mechanism of CYBER 180 is an advantage both to user programmers and to CDC´s system programmers. Much of the operating system and most of the product set software executes in virtual memory. NOS/VE and user jobs run in the same environment without special constraints. Thus, the system can use itself and take advantage of the same software made available to the users.

Initially, CYBER 180 systems will operate in dual state, that is with two separate operating systems running in the same mainframe: NOS/VE and NOS/170, or NOS/VE and NOS/BE. These two operating systems communicate across a memory link, but do not perform a very dynamic sharing of the hardware. Peripheral processors and central memory are partitioned. Certain peripheral equipments are associated with CYBER 170 state and others with CYBER 180 state. Dual state operation is a requirement on initial CYBER 180 systems, and NOS/VE may be accessed only by coming through the CYBER 170 operating system.

NOS/VE is composed of a series of jobs. These may be user jobs, operator jobs, or jobs that are part of the operating system. Within each job the unit of execution is the task. Each executable task has an exchange package and segment descriptor table associated with it. Tasks are further broken down into a series of object modules that are compilations of some higher level language such as CYBIL.

CPU Monitor

The CPU monitor is the most privileged part of NOS/VE, residing in ring 1. It sees exchange packages and segment descriptor tables for a series of tasks. These tasks, whether from user jobs or system jobs, are all handled alike by the CPU monitor. The monitor communicates to the task through the signal buffer.

The basic responsibilities of CPU monitor are task dispatching, physical I/O, and exchange interrupt processing.

Task Attributes and Components

In addition to the exchange package, segment descriptor table and signal buffer, each task has a set of standard operating system routines for use during execution. The most privileged is the task monitor in ring 1, which handles traps and communicates with the CPU monitor using an Exchange instruction or by sending a signal via the CPU monitor to a system job. Task services occupy rings 2 to 6, and include record manager, Loader, file manager, buffer manager, and segment manager. Rings 7 to 15 are available for user, application and system modules. Procedure calls are used to communicate between task monitor and task services and, in turn, between task services and module code.

System Jobs

Tasks in the system job are responsible for multiplexing all of the users across the resources of the system. These include job initiator, job terminator, system operator and page manager. Page manager manipulates the page table, selecting which pages are in which working set, and so forth.

MEMORY MANAGEMENT

There are two concepts of memory in the CYBER 180: virtual memory and real memory. Virtual memory is divided into segments, and is the view of memory as seen by all end users, by the software product set, and by a significant portion of NOS/VE. Real memory is divided into pages, and is the low-level view of memory as seen by that part of NOS/VE that is responsible for manipulating system page tables and other similar functions.

Even though virtual memory provides a large address space, system programmers can not ignore their requirement to write efficient, high-performance code. To accomplish this, procedures that reference one another should be placed as close together as possible. This increases the probability that they will be in the same page. Procedure and data references within a single page (or low number of pages) increase performance by minimizing working set size and eliminating unnecessary page swapping. This concept of locality of reference is important to follow, even though the programmer is relieved of many other memory management details.

Virtual Memory Management

The following NOS/VE components are responsible for virtual memory management.

File System

The file system maintains the ring and key attributes for all local and permanent files. It also provides segment level access to files, where files are made available in a segment of address space and are accessible with loads and stores.

NOTE

Files are also accessible through the record manager with gets and puts.

Segment Manager

The segment manager adds and removes segments from the task address space and assigns the active segment identifiers (ASID).

Compilers

Compilers are responsible for generating object modules. An object module is produced from language-dependent units of source code, for example, a main program or subroutine in FORTRAN, or a module in CYBIL. Each object module is separated into multiple object text sections that are separately manipulable at load time. The object modules produced by compilers typically contain three kinds of sections: executable code (that is, instructions), working storage (read only and read write data), and binding information (virtual addresses).

Loader

The Loader links modules in virtual memory. It also assigns process virtual addresses (PVA). This includes the ring number, segment number, and byte number (the offset within a segment), all of which are assigned at load time, not compile time. The Loader also enforces binding segment conventions, for example, that procedure descriptors are aligned on word boundaries and that they only contain valid protection information.

Object Library Generator

The object library generator takes the raw compiler output and reformats these object modules into load modules, which can be loaded more efficiently. It also creates object libraries, and binds multiple modules that were compiled separately into single loadable units.

Real Memory Management

The following NOS/VE components are responsible for real memory management.

Page Management

Page management is responsible for maintaining job working sets, that is the number of pages of virtual memory that must be in real memory for the job to progress. This solves the same problem that C170 systems solve with overlays, however in C180, the system rather than the user is responsible for the details of the solution.

Job Scheduler

Job scheduler is responsible for sharing the amount of memory among all the job working sets. This is similar to the CYBER 170 function of swapping.

FORTRAN LGO EXAMPLE

In this example, the user is validated to execute in ring 11. The user calls the FORTRAN compiler and compiles two programs, one named Main and the other named Sub. The user puts the output of both Main and Sub in LGO, and then executes that program. The format of the file LGO is shown in figure 12-3.

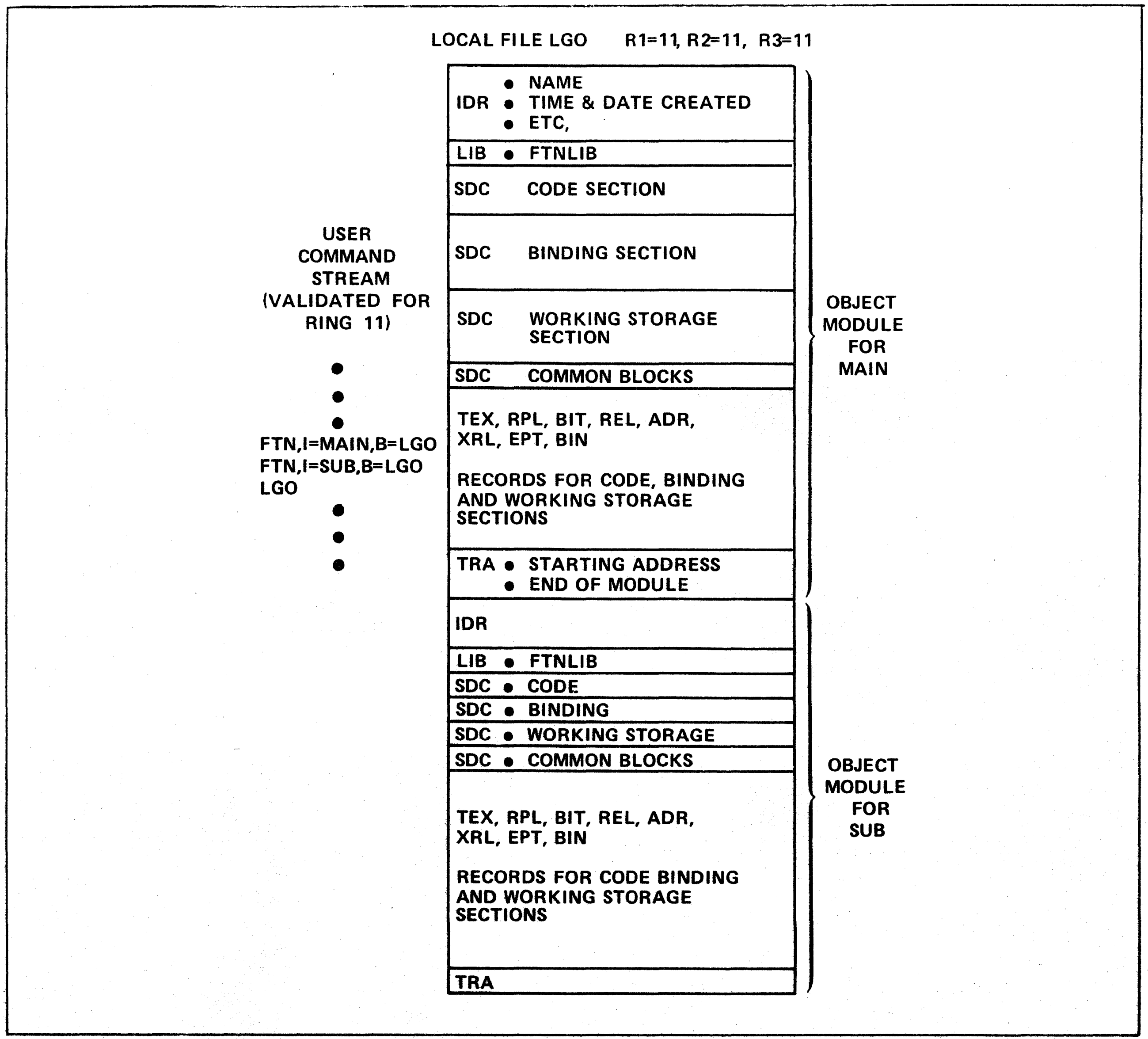


Figure 12-3. Local File LGO

Note that the Rl, R2, and R3 ring numbers are all ring 11, which are the default ring numbers derived from the user´s validated ring number, in this case ring 11. The user is not required to specify these segment attributes, and their assignment is usually a function of the installation manager.

The LGO file itself is comprised of two object modules, one for Main and one for Sub. Each module has three separate sections: code, binding, and working storage. These are followed by a series of interpretive records that provide the values and fill in the data in those three sections. Finally in each module, the transfer record gives the optional starting procedure name and specifies that it is the end of the module.

Several segments are created when the LGO is executed.

Process Segment Number	Name
10	code segment
11	binding segment
12	working storage segment
13	object library file
14	stack segment

It is important to understand the difference between a section and a segment. A section is a part of a module (figure 12-4). At load time, sections from all modules having the same attributes are allocated contiguously in the same segment. Thus, a process segment may contain similar sections from different modules, which minimizes fragmentation. Segments are closely associated with physical memory, and are the basic unit of allocation in virtual memory.

The ring numbers (11 in this example) are derived from the ring numbers of the LGO file and applied appropriately to these process segments. The code segment contains instructions and has Read and Execute privilege. The binding segment contains addresses of working storage and of other routines to be called (in this case Sine and Random). The binding segment is readable but not user-writable. All addresses in the binding segment are valid, and if one, for instance, should point to an address of a system routine that goes across rings, the user cannot overwrite that address. The working storage segment contains static data associated with the various modules.

The object library file is a segment-level access file stored in process segment 13. The file is accessed using Load and Store machine instructions to reference directly segment 13. It is particularly important, therefore, to protect against invalid accesses to this type of file. Because this is a read-execute file, the file system software prohibits any attempted writes to that file, and when the file is included in a task address space, the access attributes in the associated segment descriptor are used by the hardware to prevent any write access.

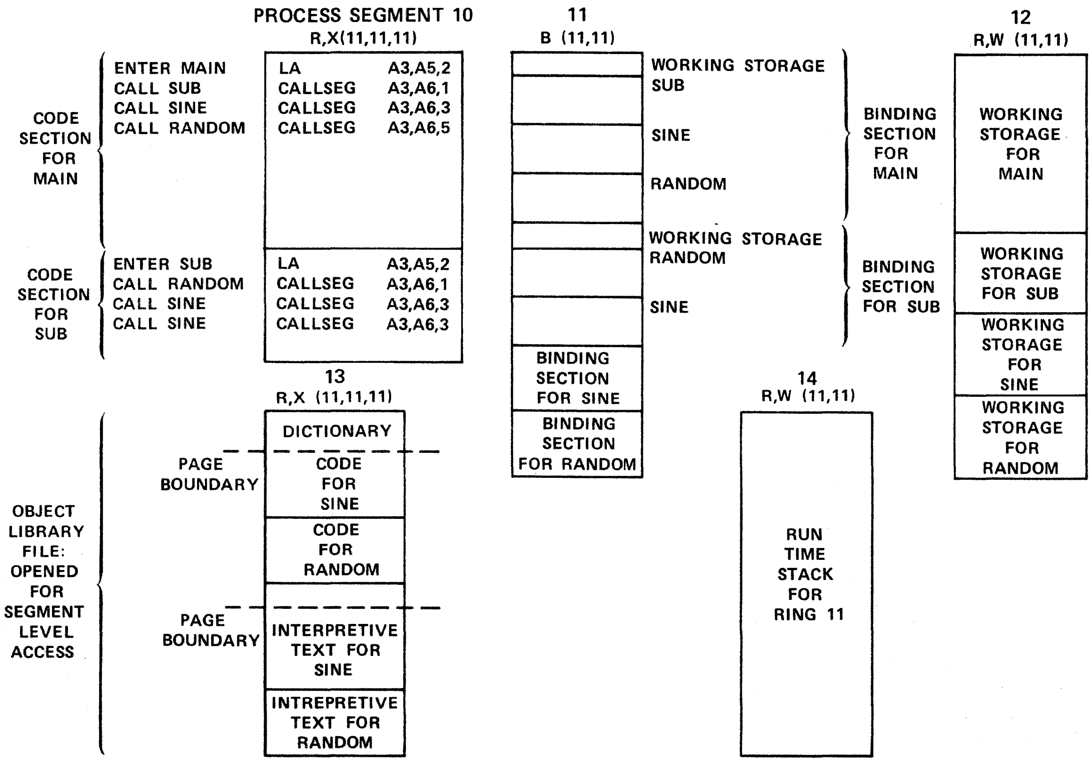


Figure 12-4. FORTRAN LGO Example

The object library generator has reformatted the object code for subroutines Sine and Random into load modules that can be executed. Also included in the object library file is interpretive text for Sine and Random. This directs the Loader on where to locate and how to initialize the binding sections and working storage sections for Sine and Random. All of the code is together in one part of the file, and all of the interpretive text is together in another part of the file. This is done for locality of reference, that is to minimize references across page boundaries.

Figure 12-5 illustrates the various pointers which are established. The pointers with broken arrows are generated by the compiler, and are offsets into the Binding Section. The pointers shown with unbroken arrows are supplied by the Loader. They all reside in the Binding Section (which contains only addresses), and forge the necessary link between a program and the data on which it is to operate.

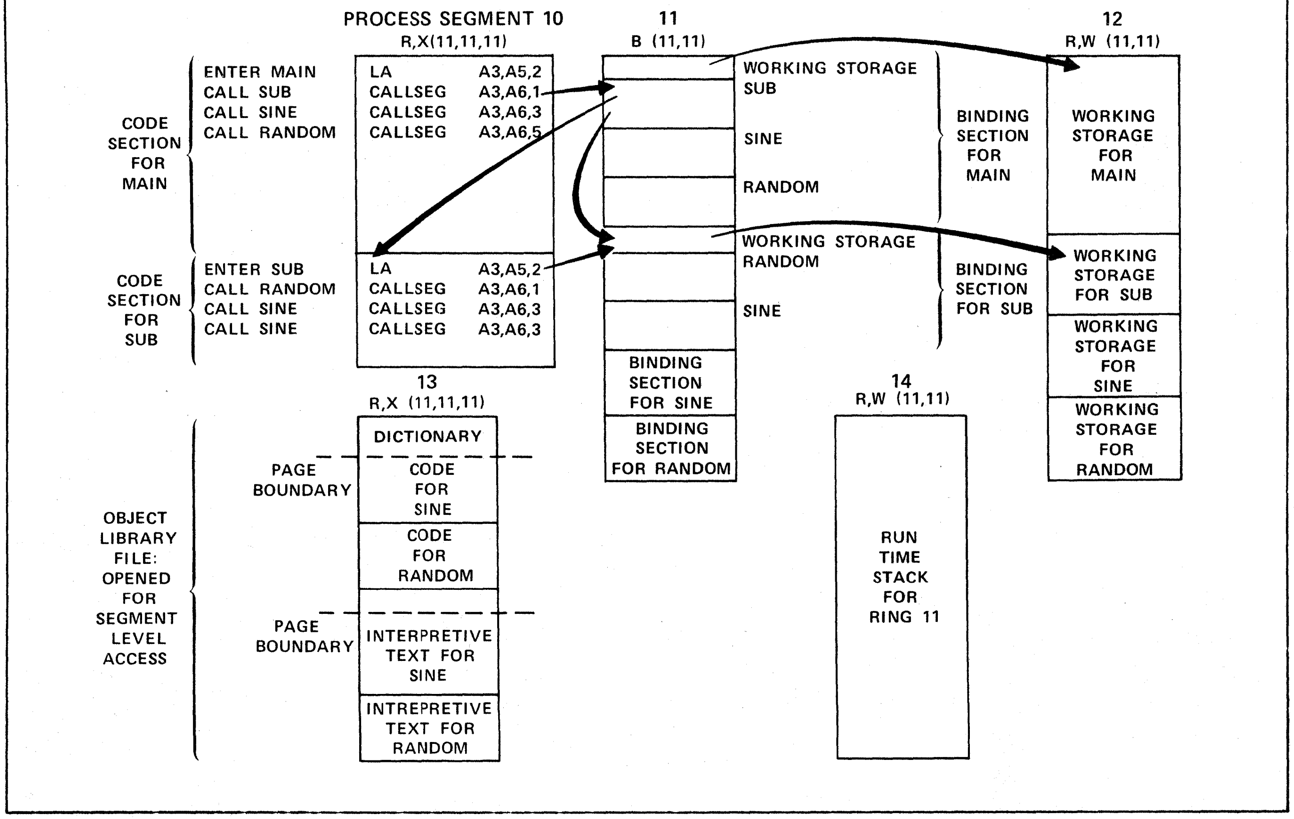


Figure 12-5. FORTRAN LGO Example, Pointers

When the program is debugged and ready for production operation, Loader overhead is reduced by running LGO through the object library generator, turning it into a segment-level access file. The two modules in LGO (Main and Sub) are bound into a single loadable module (New). Once again, the ring brackets of LGO (11, 11, 11) are carried forward to the new object library file.

The format of this new object library is shown in greater detail in figure 12-6.

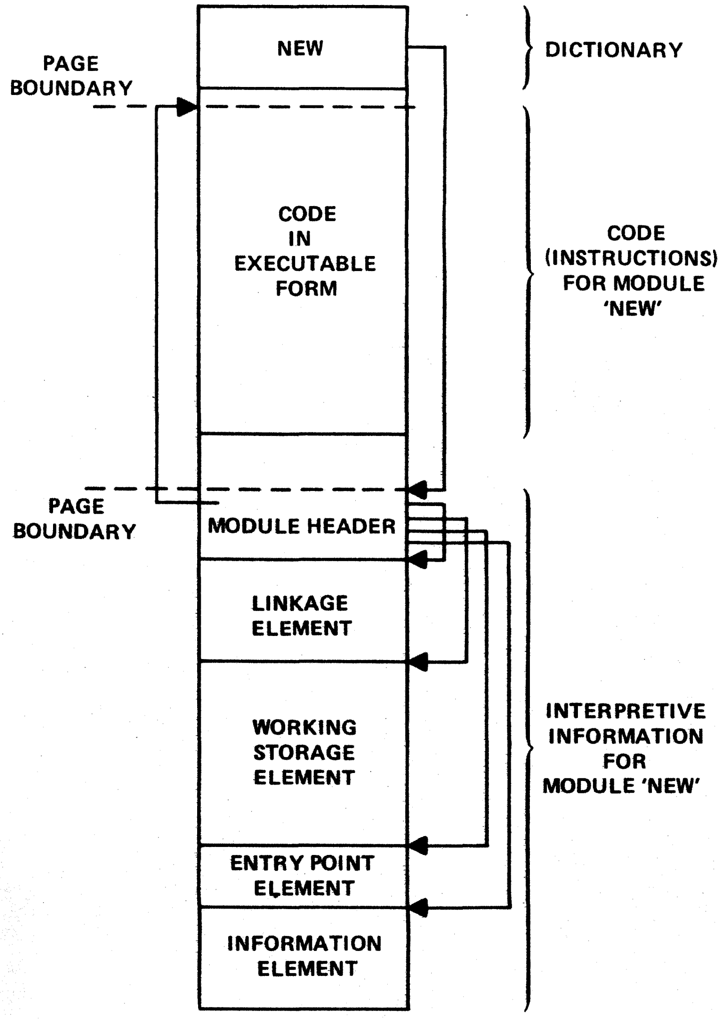


Figure 12-6. Object Library Format - Segment Level Access File

First is the entry point name dictionary for the single module named New. This points to the header, which tells the Loader where to find the other interpretive information for this module. The module code is in executable form, and is physically separated from the interpretive information. This segment level access file is now contained in process segment 10 as shown in figure 12-7.

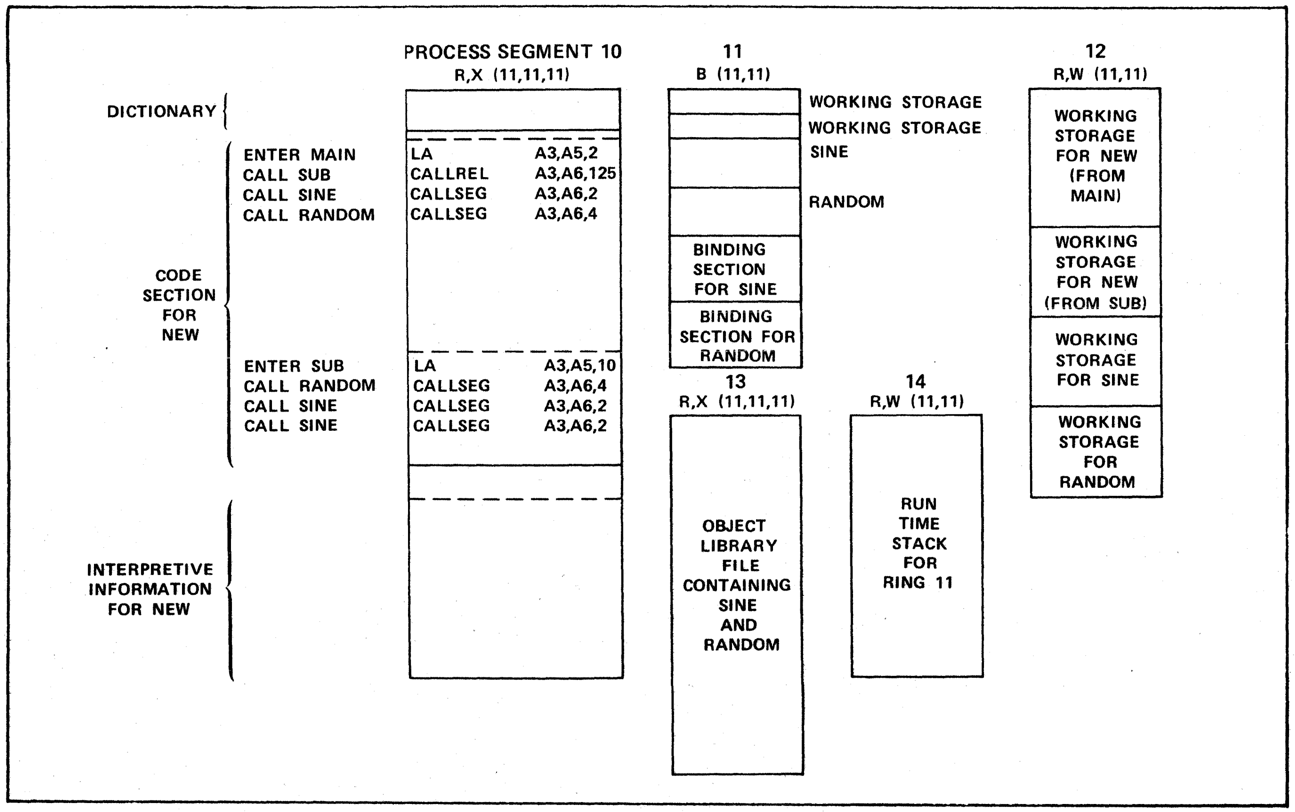


Figure 12-7. FORTRAN LGO Example, Conclusion

In the previous example, Main and Sub were compiled separately. This resulted in redundant procedure descriptors for both Sine and Random in the binding segment. Since all calls to the operating system are procedure calls, every system call a module makes will result in a binding system entry. When several modules are combined together, however, the object library generator compresses any redundant entries into single copies within the new binding segment, and modifies the code accordingly.

The working storage segment remains the same as before because all of the static data from Main and from Sub is needed by the single module New.

If this figure is compared with the previous figure showing the load image, some important differences can be noted. Not only has the Binding Section for New been compressed, but the call from Main to Sub has been modified to a CALLREL (Call Relative) from a CALLSEG (Call Indirect). The Call Relative is a shorter form of the general call instruction which is reserved for intrasegment calls when protection boundaries are not crossed.

COMMENT SHEET

MANUAL TITLE: CDC CYBER 170 Models 825, 835, and 855 Computer Systems Hardware Maintenance Manual

PUBLICATION NO.: 60459960 REVISION: A

NAME: ______________________________

COMPANY: ______________________________

STREET ADDRESS: ______________________________

CITY: ______________ STATE: ______________ ZIP CODE: ______________

This form is not intended to be used as an order blank. Control Data Corporation welcomes your evaluation of this manual. Please indicate any errors, suggested additions or deletions, or general comments below (please include page number references).

□ Please Reply □ No Reply Necessary

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

FOLD FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 8241 MINNEAPOLIS, MINN.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION

Publications and Graphics Division
ARH219
4201 North Lexington Avenue
Saint Paul, Minnesota 55112

FOLD FOLD

CUT ALONG LINE

CORPORATE HEADQUARTERS, P.O. BOX 0, MINNEAPOLIS, MINN. 55440
SALES OFFICES AND SERVICE CENTERS IN MAJOR CITIES THROUGHOUT THE WORLD

LITHO IN U.S.A.

CONTROL DATA CORPORATION