MAN2798

P400 SYSTEM REFERENCE
User Guide

Revision PRELIM. A
13 JULY 1976

PRIME
Computer, Inc.
145 Pennsylvania Ave.
Framingham, Mass. 01701

Performance characteristics are
subject to change without notice.

CONTENTS

ILLUSTRATIONS

Figure	Title	Page

There are no illustrations in this document.

TABLES

In the period prior to the publication of the official PRIME 400 System Reference Manual, this document is intended to serve as an interim programmer's reference manual for systems containing the PRIME 400 central processing unit. This document contains the information immediately necessary to allow both systems programmers as well as application programmers to begin utilizing the advanced features of the PRIME 400. As this document discusses only the differences between the PRIME 400 and earlier processors, the reader is expected to have a working familiarity with the PRIME 300.

RELATED DOCUMENTS

The following listed documents may provide useful supplemental information to the reader of this User Guide.

Document Title	Order No.
Prime 100-200-300 System Reference	MAN 1671
Macro Assembler User Guide	MAN 1673
FORTRAN IV User Guide	MAN 1674
PRIMOS Interactive User Guide	MAN 2602
PRIMOS Computer Room User Guide	MAN 2603
PRIMOS File System User Guide	MAN 2604
Program Development Software User Guide	MAN 1879

DOCUMENT PRODUCTION

This document was produced on line via the editing and storage features of the Prime PRIMOS operating system. Hard copy suitable for printing has been produced via the RUNOFF command, available with software revision Rev 8.0 of the PRIMOS operating system. User comments, reactions, and suggestions on this document format and content are solicited, as always.

SECTION 1

INTRODUCTION

1.1 INTRODUCTION TO THIS DOCUMENT.

In the period prior to the publication of the official PRIME 400 System Reference Manual, this document is intended to serve as an interim programmer's reference manual for systems containing the PRIME 400 central processing unit. This document contains the information immediately necessary to allow both systems programmers as well as application programmers to begin utilizing the advanced features of the PRIME 400. As this document discusses only the differences between the PRIME 400 and earlier processors, the reader is expected to have a working familiarity with the PRIME 300.

This document is provided for information purposes only and is not a specification. All information contained herein is subject to change without notice.

1.2 INTRODUCTION TO THE PRIME 400 PROCESSOR.

The PRIME 400 is a two-board processor designed to plug into the present chassis, to drive all current and planned peripheral devices and controllers, to interface with all present 32K and future memories, to operate all present user-space software, and to obey the compatibility and family constraints of the PRIME Computer User Plan. The processor is very fast (built with high technology) and has segmented addressing (for modern software organization and a very large address space).

Thus, the PRIME 400 does two things. First, it provides a high-end product with the speed and capacity to handle very demanding new applications, such as large data bases, multi-task real-time control, and distributed networks supporting large-scale mainframes. Second, it provides a compatible growth path for existing or proposed PRIME 300 installations. In summary, the PRIME 400 preserves the customer's existing investment in hardware and software while providing a range of new speed and capacity features for greatly enhanced performance in new applications.

1.3 COMPATIBILITY.

Compatibility is a stringent goal in the PRIME 400 product offering. The new processor is absolutely hardware compatible with the present chassis, present power supplies, present 32K memories, all present peripheral device controllers, and the software-visible decor of the PRIME 100/200/300. On the software side, all existing user-space

programming operates without change on the PRIME 400 processor, all present data file structures are preserved without change, and aspects of upward and downward compatibility are maintained.

Notwithstanding the above, certain architectural advantages such as segmentation cannot be downward compatible with respect to programs designed to utilize them effectively. A segmented addressing space provides the basis for a simpler and more effective operating system--a combined DOS, DOS/VM, and RTOS/VM operating system known as PRIMOS. As PRIMOS takes heavy advantage of the advanced features of the PRIME 400 processor, it itself is not downward compatible. However, PRIMOS supports all existing DOS/VM commands, the existing DOS/VM file structures, and all PRIME 100/200/300 addressing and execution modes. Thus all existing user-space programs (including saved memory images) run under PRIMOS without modification. Furthermore, PRIMOS can be used to write, develop, and run new downward-compatible programs which can be interchanged with PRIME 100/200/300 environments at any time.

In one sense then, the management of the downward compatibility of segmentation is handled much the same way as the compatibility of the paging feature of the PRIME 300, which is not available on the PRIME 100/200. That is, an operating system which does take advantage of the feature is provided, which is compatible with previous operating systems, and which allows user-space programs which are indifferent to the feature to be treated in a completely upward- and downward-compatible fashion.

In another sense though, the handling of segmentation is different from the paging feature of the PRIME 300. That is because, in addition to itself taking advantage of the feature, PRIMOS also passes on to the user-space environment the ability to fully utilize segmentation when desired.

1.4 PERFORMANCE.

The PRIME 400 performance is between two and three times that of the PRIME 300, especially in benchmark situations. For comparison, some PRIME 300 and PRIME 400 instruction times are shown in Table 1.1. Note that on the PRIME 300 the ADD instruction in the worst case (which is the usual case) takes 2480 nanoseconds, because of page-translation time (160 ns), 600 ns memory, and the use of relative mode (in which the index operation costs 440 ns). Thus the normal PRIME 300 ADD instruction under DOS/VM takes 2480 ns. By comparison, the best case ADD for the PRIME 400 takes 560 ns, for an improvement factor of 4.4. The comparison of worst to best is fair because on the PRIME 400 the best case is readily achievable in ordinary programming and benchmarks. The average PRIME 400 ADD time (assuming an 85 percent hit rate in the cache and interleaved memories) is 920 ns, which is 2.7 times better than the PRIME 300--a very substantial improvement.

Other integer arithmetic improvements are characterized by the MPY

instruction, which improves by 9640/4200 = 2.3 times. Floating-point improvements are characterized by the FAD and FMP instructions, which improve by 8990/4220 = 2.1 times and 25280/9000 = 2.8 times respecively, which are again very substantial savings.

I/O performance is improved in four ways: shorter latency time (the time an I/O controller must wait for service after requesting it); faster data rates (shorter data transfer time when service is granted); many more direct memory access (DMA) channels (in which control information is stored in registers rather than in memory); and entirely new modes (for greater I/O efficiency). Table 1.2 compares the times for the PRIME 300 and PRIME 400 I/O modes.

The architectural features which give these performance improvements are as follows:

1. Cache. A 1024-word bipolar buffer between the central processor and memory reduces the effective memory access time from 680 ns to 240 ns. It also eliminates (completely overlaps) the time required for paging and segmentation translation.

2. 32-bit arithmetic and logic unit. Arithmetic performed on full 32-bit quantities greatly reduces time for arithmetic and floating-point operations. The 32-bit adder also speeds up relative address formation.

3. Faster control unit. The new microcode structure for the control unit allows very fast steps and reduces the number of steps required. For example, a PRIME 400 ADD instruction requires only two steps, as opposed to five on the PRIME 300.

4. Registers. The live-register set is increased from 32 16-bit registers on the PRIME 300 to 128 32-bit registers. This allows multiple register sets for very fast process exchange.

5. Interleaved memory. On the PRIME 400, main memory can be interleaved, which speeds up sequential access and reduces the cache miss rate.

TABLE 1.1.
COMPARISON OF PRIME 300 AND PRIME 400
INSTRUCTION EXECUTION TIMES.

PRIME 300 TIMES:

instruction	440 ns mem paging off 32S mode	600 ns mem paging off 32S mode	600 ns mem paging on 32S mode	600 ns mem paging on 32R mode
ADD M,1	1560	1880	2040	2480
ADD R (note 1)	1760	1820	1900	1900
DAD M,1	2800	3280	3440	3880
MPY M,1	8720	9040	9200	9640
FAD M,1 (note 2)	-	-	-	8990 +480A+720N
FMP M,1	-	-	-	25280

PRIME 400 TIMES:

All times for interleaved 600 ns memories, and include segmentation and paging translation times. The assumed cache hit rate is 85 percent, with a 1200 ns cache fault time (doubleword fetch).

instruction	best case	average case
ADD M,1	560	920
MPY M,1	4200	4560
FAD M,1 (note 1)	3500 +160A+160N	4220 +160A+160N
FMP M,1	8280	9000

ALL TIMES IN NANOSECONDS

NOTE 1: ADD from a register, R<8.

NOTE 2: A = number of required adjust cycles; N = number of required normalization cycles.

TABLE 1.2.
COMPARISON OF PRIME 300 AND PRIME 400 I/O TIMES.

mode	input data transfer time	output data transfer time
(PRIME 300, 440 ns memory)		
DMT, first word	2760	2600
DMT, later words	800	880
DMA, first word	2860	3000
DMA, later words	1120	1080
DMC, first word	4940	4980
DMC, later words	3440	3480
(PRIME 400, any memory)		
DMT, first word	1400	1640
DMT, later words	800	880
DMA, first word	1400	1640
DMA, later words	800	880
DMC, first word	2520	2920
DMC, later words	2080	2760
DMQ	5000	5000

ALL TIMES IN NANOSECONDS

The "first word" times refer to the first word of a block of words to be transferred at the maximum I/O rate. The "later words" times refer to all words of the block after the first word.

1.5 INPUT/OUTPUT OPERATION.

Compatibility requires that all PRIME 300 I/O modes be fully supported on the present I/O bus. Thus, I/O through the A-register as well as the DMA, DMC, and DMT direct modes of operation are fully supported, but with improved performance as discussed in Section 1.4. In addition, several new features are added:

1. Mapped I/O through segment 0.
2. Remote I/O bus extender and an I/O bus switch.
3. New direct-memory queue (DMQ) mode for stream I/O.
4. 32 DMA channels instead of 8.
5. Very fast DMC data rate.
6. Interrupts which automatically initiate process exchange.

The mapped I/O feature allows easy I/O access to the entire 2**22 (4 million) words of physical memory, even though the I/O bus retains its former 18-bit address width. The mapping feature causes I/O accesses to memory to undergo segmentation and paging translation just as processor references; the PRIMOS operating system is responsible for keeping the necessary virtual-to-physical correspondence in effect for the duration of the transfer. This mapping also aids the operating system in performing file transfers.

The remote I/O bus extender allows the addition of up to four remote backplanes, each of which can drive ten I/O controllers, along a 30-foot cable out from the processor. The I/O bus switch allows the switching of controllers among several processors.

A new direct memory queue (DMQ) mode provides a ring-structured memory buffer for the reception and transmission of stream I/O (I/O in which data is transferred in continuous streams of bits, characters, or words, rather than in discrete records). This mode allows the asynchronous multi-line controller to queue messages without the need for extensive software management of "tumble tables" on receive, nor character-time interrupts on transmit. The DMQ mode substantially reduces the PRIMOS overhead in dealing with user-terminal I/O.

The large register set of the PRIME 400 provides for 32 DMA channels. Also, since the cache is used to hold DMC cell pairs, repetitive DMC transfers occur very quickly, as shown in Table 1.2.

For interrupts, a new central processor mode is defined which allows an interrupt signal to be processed as an automatic notify (wakeup) of a process without causing an actual program interruption. The mode automatically issues the proper interrupt-clearing instructions to the signalling controller. This mode allows very fast process exchange times and greatly reduces the overhead of the multiple-priority

scheduling schemes common to the RTOS and PRIMOS operating systems.

Overall, PRIME 400 I/O performance is considerably enhanced over the PRIME 300.

1.6 FIRMWARE ENHANCEMENTS.

The PRIME 400 uses a new microcode structure with the following salient features:

1. 64-bit microcode word width.
2. IBM-style multiway branches.
3. 16K words of microcode address space.
4. Stack of depth 16.
5. Future availability of an extended control storage (XCS) option.

The 64-bit width of the new microcode allows more functions to be controlled in parallel, and thus reduces the number of microcode steps necessary to perform a given function. For example, the ADD instruction executes in two microcode steps on the PRIME 400 as opposed to five steps on the PRIME 300. The IBM-style multiway branches are also important because they are very fast.

The 16K address space allows for considerable future expansion of the microcode. The present two-board PRIME 400 provides 2K 64-bit words of on-board programmable read-only memory (PROM) using 2K PROM parts. However, the board layout will accommodate a 4K PROM part when it becomes available, giving 4K words of on-board PROM.

Microcode can also be expanded with an extended control storage (XCS) board, to be available as an option in the future. The XCS board will provide:

1. PROM extension of an additional 2K words (at least).
2. 1K words (at least) of program-writable control store.
3. Parity checking on all microcode words.
4. A simulate mode for writable control store (as in the PRIME 300 writable control storage option).
5. A port for connection of a PROM programmer.

The writable control storage will be loaded internally under program control or by I/O operations. The PRIME 400 instruction set has two

addressable and eight generic instructions reserved for a direct decoding into writable control storage. The extended control storage option is being designed specifically to support customer microprograms as well as packaged microcode systems, such as a business instruction set, a fast Fourier transform processor, a matrix operation package, etc.

1.7 INTEGRITY ENHANCEMENTS.

The PRIME 400 is equipped with several new integrity features, representing a considerable improvement over the PRIME 300. These features include:

1. Parity checking on processor registers and the cache.
2. As an option to be available, an error detecting and correcting code on each main memory word.
3. Improved program control over the disposition of machine and parity checks.
4. Recording of the origin and status of every machine and parity check signal in a diagnostic status word.
5. A non-destructive VIRY instruction.
6. As an option to be available, a field-engineering panel with a ring-buffer remembering the last 64 microcode addresses fetched by the processor.

Parity is maintained and checked on all the live-registers (128 32-bit registers) of the processor and of the data in the cache. Parity is also checked on all external busses. When the extended control store option is provided, there will also be a parity check on each 64-bit microcode word.

A further option to be available on main memory boards is an error detecting and correcting code on each memory word. The code is capable of correcting all single errors and detecting some double errors. When correction is possible, it is done automatically in the memory on-the-fly, with no delay to the processor. If a correctable error occurs during instruction execution, a check signal which may be requested by the software (see the discussion of the machine check modes below) is held off until the completion of the instruction to allow the computation in progress to benefit from the corrected value; following the check, the operating system can elect to continue the computation regardless of whether or not the hardware or the software elected to run a diagnostic routine in the meanwhile. Correctable errors which occur during direct-memory I/O operation (DMA, DMC, DMT, DMQ) are simply corrected and cause no check signal ever, to maximize the likelihood of completing the I/O transfer successfully.

Uncorrectable errors cause a check signal immediately if during instruction execution, or following completion of the current instruction if during direct-memory I/O, or else are completely ignored (depending upon the machine check mode). As discussed below, all check signals are accompanied by a complete description of the detected error in the diagnostic status word for analysis by the check handler.

The PRIME 400 gives the software improved control over the disposition of check signals. A two-bit machine check mode field is provided which allows the software to run the processor in one of four check modes. The machine check mode field is the last two bits of the processor modals, and is set with the LPSW instruction. The four modes are:

00: "None". The processor is not in an error reporting mode. Errors set a program-testable flag but no check is signalled. The diagnostic status word is not set.

01: "Memory parity". The processor sets the diagnostic status word and generates a check signal for all memory parity errors (and all uncorrectable memory errors detected by the error detecting and correction option, if installed), both during instruction execution and also during direct-memory I/O. Correctable memory errors are ignored and processor parity failures set a program-testable flag in this mode.

10: "Quiet". The processor sets the diagnostic status word and generates a check signal for all detected errors other than a correctable memory error. Correctable memory errors are ignored in this mode.

11: "Record". The processor sets the diagnostic status word and generates a check signal for all detected errors in this mode. In the case of a correctable memory error, the check signal is held off until the instruction in progress completes, to allow the software the option of resuming the computation following servicing of the check. Correctable memory errors which occur during direct-memory I/O are always ignored, even in this mode, in order to allow the I/O transfer to complete successfully when possible with the correction.

The diagnostic status word is a 96-bit field set by the processor whenever it detects an error which should result in a check signal to the software. The software handling the check signal can read the diagnostic status word to learn the origin of the signal and take appropriate action.

A check is either a memory parity error or else a machine check. There are three circumstances which can cause a memory parity check. The first is detection of a main memory parity error (or an uncorrectable main memory error, if the error detecting and correcting option is installed) during instruction execution when the processor is not in machine check mode 00 ("none"). The second is occurrence of a correctable main memory error (the error detecting and correcting

option must be installed) during instruction execution when the processor is in machine check mode 11 ("report"). The last is detection of a main memory parity error (or an uncorrectable error, with the correcting option installed) during direct-memory I/O when the processor is not in machine check mode 00 ("none"). When the error detecting and correcting option is installed, corrected errors during I/O execution are always ignored, never set the diagnostic status word, and never signal a check.

A machine check is caused by detection of a parity error on a processor internal register or on an external bus when the processor is in machine check mode 10 ("quiet") or 11 ("record"). When the processor is running in modes 00 ("none") or 01 ("memory parity"), processor parity errors do not set the diagnostic status word and do not cause a check signal, but do set a program-testable flag.

The VIRY instruction triggers a series of microprograms that can verify the integrity of the internal processor components without being destructive to the state of the user's program in execution. This greatly eases restart of the interrupted computation following a check, even if the check handler desires to perform verification.

1.8 IMPLEMENTATION.

The PRIME 400 is implemented as a two-board processor with the boards connected by a 200-signal bus across the top-hat connectors. The top-hat bus carries microcode control signals as well as the full-width source and destination busses for the arithmetic and logic unit (B and D busses). The extended control storage option is packaged as a third board, also connecting via the top-hat bus.

The standard two processor boards hold approximately 330 dual-inline packages each and utilize eight-layer construction. The multilayer construction is used to provide the necessary ground plane for the high-speed logic signals as well as to achieve interconnections at the high package density. The logic family is TTL throughout, mostly Schottky-clamped. No emitter-coupled logic is used. All logic packages are industry standard multiple-sourced units.

SECTION 2

PROGRAM-VISIBLE DECOR

The program-visible portion of the PRIME 400 architecture is described in this section. First, the virtual-memory addressing space (segmentation and paging features) which underlies the PRIME 400 design is discussed. Then, the PRIME 400 instruction and register sets are described in turn. Finally, a number of special features and unique capabilities which contribute to the power of the PRIME 400 product are covered.

2.1 VIRTUAL MEMORY STRUCTURE.

Physical memory on the PRIME 400 can be as large as 4,194,304 (2**22) 16-bit words. The virtual space is 268,435,456 (2**28) 16-bit words. The mapping of virtual space to physical space includes both segmentation and paging. The page size is 1024 words. The segment size is 0 to 65536 words in units of 1024 words. There are 4096 segments to a virtual space. The segments are in four groups of 1024 segments each. There are four descriptor table address registers (DTARs), which point to tables containing segment descriptor words (SDWs), which point to tables containing page map entries (PMNTs), which point to physical pages of memory. Thus a 28-bit virtual address contains 2 bits of descriptor table selection, 10 bits of segment selection, and 16 bits of word selection. It should be noted that the hardware-implemented automatic process-exchange mechanism does not affect the contents of DTARs 0 and 1 and, therefore, all processes share the same first 2048 segments of virtual address space and have the second 2048 segments as private space. Finally, the presence of both paging and segmentation permits the separation of memory management from operating system management. Table 2.1 shows the formats of descriptor table address registers, segment descriptor words, and page map entries.

A descriptor table has from 1 to 1024 entries, must begin on an even word, and must not cross a 65536-word boundary. A page table always has 64 entries and must not cross a 65536-word boundary. Pages must begin on a 1024-word boundary.

There must be no missing memory locations in the first 65536 words of physical memory.

Virtual memory operation is under control of bit 14 of the processor modals, loadable under program control via the LPSW instruction. When this bit is off, no paging or segment translation is performed. The low-order 22 bits of each virtual effective address are taken as a physical address directly.

TABLE 2.1
VIRTUAL MEMORY FORMATS

DESCRIPTOR TABLE ADDRESS REGISTER FORMAT
(32 bits)

SSSSSSSSSSDDDDDD
-DDDDDDDDDDDDDD

1-10: 1024 minus descriptor table size (SSS...S).
11-16, 18-32: High-order 21 bits of 22-bit physical address descriptor table origin, low bit taken as zero (DDD...D).
17: Not used.

SEGMENT DESCRIPTOR WORD FORMAT
(32 bits)

PPPPPPPPPP------
FAAABBBCCCPPPPPP

17: Fault if 1 (F).
18-20: Access allowed from ring 1 (AAA).
000: No access.
001: Gate (for procedure call).
010: Read.
011: Read and write.
100, 101: Reserved.
110: Read and execute.
111: Read, write, and execute.
21-23: Reserved for future expansion (BBB).
24-26: Access allowed from ring 3, same code as above (CCC).
27-32, 1-10: High-order 16 bits of the 22-bit physical address of the page table origin (PPP...P).
11-16: Reserved, must be zero.

PAGE MAP ENTRY
(16 bits)

VRUSAAAAAAAAAAA

1: Valid: page resident if 1, fault if 0 (V).
2: Referenced: set by hardware when page is referenced (R).
3: Unmodified: reset by hardware when page is modified (U).
4: Shared (inhibit usage of cache buffer): set by software when memory page is shared among processors (S).
5-16: High-order 12 bits of physical page address, low-order 10 bits are taken as zero (AAA...A).

2.2 PRIME 400 INSTRUCTION SET.

For downward compatibility with the PRIME 300, the PRIME 400 provides all of the former addressing modes: 16S, 32S, 32R, and 64R. When running in one of these modes, the PRIME 400 decodes instructions and develops effective addresses in the same way as the PRIME 300. Memory addresses 0 through 7 map onto the live registers in the same way, and the program can directly or indirectly reference up to 65536 words of memory. Support of these four modes allows existing DOS/VM user-space programs to run under the PRIMOS operating system on the PRIME 400 without change. The 16-bit effective addresses generated in the PRIME 300 modes are expanded to a full 28-bit virtual address by automatic concatenation of a 12-bit segment number established when the PRIME 300 mode is entered. Thus, when a DOS/VM user program is run under PRIMOS, its entire 65536-word address space appears as a single segment to the PRIME 400.

To permit usage of multiple segments and the new instructions defined for the PRIME 400, a new virtual-addressing mode is provided: 64V. When run in 64V mode, the PRIME 400 provides access to multiple segments via 32-bit and 48-bit indirect words, four 32-bit base registers, and two 64-bit field address and length registers. In addition, a second index register is provided, and all combinations of base registers, indirection, and pre- and post-indexing by either index register can be specified in instruction addresses.

Instruction decoding in 64V mode proceeds as follows. As in the other modes, sequential decoding begins with bits 3-6 of the instruction word. If these bits are all zero, the instruction is of the generic class, and all the remaining bits of the instruction word are also interpreted as part of the operation code. The PRIME 400 generic instructions include the PRIME 300 generics and many new ones in addition. Table 2.2 lists the PRIME 400 generics and their functions.

If bits 3-6 are not zero, the instruction is of the memory-reference class, bit 7 is the sector bit, S, and bits 8-16 are a displacement, D, in two's complement notation (-256 <= D < +256). If the sector bit is one and the displacement is in the range -256 <= D < -224, then instruction word bits 13-14, WW, are interpreted as an extension of the operation code, selecting a column of Table 2.3 for execution. If the sector bit is a zero or the displacement is not in the range -256 <= D < -224, then execution occurs from the first column of Table 2.3, as if the extension bits had been zero. Table 2.4 explains the meanings of the memory-reference instructions.

In all memory-reference instructions, bit 1, I, specifies indirect addressing. In all but those with bits 3-6 equal to 1101, bit 2, X, specifies indexing. When bits 3-6 equal 1101, bit 2 is used as an extension of the operation code, and such instructions cannot be indexed.

TABLE 2.2.
PRIME 400 GENERIC INSTRUCTIONS.

This table summarizes all PRIME 400 generic operations, grouped by function. Instructions marked "RESTRICTED" can be executed only in ring zero. Rings are discussed in Section 2.3.

The "type" column indicates the format and/or function of the operation as follows.

AP: Three-word operation, the last two words of which are an AP address pointer.
BR: Two-word operation, the second word of which is a word number within the current procedure segment to which to branch.
CON: Single-word control operation.
FLD: Single-word field operation.
FOPR: Single-word floating-point operation.
FSKP: Single-word floating-point skip operation.
IG: Single-word integrity operation.
IO: Single-word input/output operation.
LOG: Single-word logicize operation.
MODE: Single-word mode operation.
OPR: Single-word miscellaneous operation.
SH: Single-word shift operation.
SKP: Single-word skip operation.

The "C" column indicates the effect of the operation on the C-bit and the L-bit as follows.

-: C and L are unchanged by the operation.
1: C is unchanged, L is carry.
2: C is overflow, L is carry.
3: C is overflow, L is indeterminant.
4: C is shift extension, L is indeterminant.
5: C is a result of the operation, L is indeterminant.
6: C and L are indeterminant.
7: C and L are loaded by the operation.
8: C is cleared, L is indeterminant.

The "cc" column indicates the effect of the operation on the condition codes as follows.

-: Condition codes are unchanged by the operation.
1,4: Condition codes indicate the result of the arithmetic operation or compare.
5: Condition codes are indeterminant.
6: Condition codes are loaded by the operation.
7: Condition codes indicate the result of the operation.

mnem	opcode	type	C	cc	description
(MISCELLANEOUS AP OPERATIONS.)					
WAIT	000315	AP	-	-	WAIT ON SEMAPHORE AT AP. RESTRICTED.
CALF	000705	AP	7	6	PROCEDURE CALL FROM FAULTING PROC. TO FAULT HANDLER.
LPSW	000711	AP	7	6	LOAD PROGRAM STATUS WORD (SEGN,WORDN,KEYS, MODALS). RESTRICTED.
RSAV	000715	AP	-	-	SAVE REGISTERS (GENERAL,FLOATING,TEMPORARY BASE).
RRST	000717	AP	-	-	RESTORE REGISTERS (GENERAL,FLOATING,TEMPORARY BASE).
STAC	001200	AP	-	7	STORE A CONDITIONAL ON B=[EA16]. CCEQ= SUCCESS ELSE FAIL.
STLC	001204	AP	-	7	STORE L CONDITIONAL ON E=[EA32]. CCEQ= SUCCESS ELSE FAIL.
NFYE	001210	AP	6	5	NOTIFY ON SEMAPHORE AT AP. USE FIFO QUEUEING. NO CAI. RESTRICTED.
NFYB	001211	AP	6	5	NOTIFY ON SEMAPHORE AT AP. USE LIFO QUEUEING. NO CAI. RESTRICTED.
INEN	001214	AP	6	5	NOTIFY DURING INTERRUPT CODE. USE FIFO QUEUEING. NO CAI. RESTRICTED.
INBN	001215	AP	6	5	NOTIFY DURING INTERRUPT CODE. USE LIFO QUEUEING. NO CAI. RESTRICTED.
INEC	001216	AP	6	5	NOTIFY DURING INTERRUPT CODE. USE FIFO QUEUEING. DO CAI. RESTRICTED.
INBC	001217	AP	6	5	NOTIFY DURING INTERRUPT CODE. USE LIFO QUEUEING. DO CAI. RESTRICTED.
RTQ	141714	AP	-	6	REMOVE FROM TOP OF QUEUE. ON EMPTY, A=0 AND CC'S SET EQ.
RBQ	141715	AP	-	6	REMOVE FROM BOTTOM OF QUEUE. ON EMPTY, A=0 AND CC'S SET EQ.
ABQ	141716	AP	-	6	ADD TO BOTTOM OF QUEUE. CCEQ = FULL ELSE NOT FULL.
ATQ	141717	AP	-	6	ADD TO TOP OF QUEUE. CCEQ = FULL ELSE NOT FULL.
TSTQ	141757	AP	-	6	TEST QUEUE. A SET TO # ITEMS IN QUEUE. CC'S SET EQ ON EMPTY.
(BRANCH OPERATIONS.)					
BLE	140610	BR	-	4	BRANCH ON A REGISTER .LE. 0.
BGT	140611	BR	-	4	BRANCH ON A REGISTER .GT. 0.
BEQ	140612	BR	-	4	BRANCH ON A REGISTER .EQ. 0.
BNE	140613	BR	-	4	BRANCH ON A REGISTER .NE. 0.
BLT	140614	BR	-	4	BRANCH ON A REGISTER .LT. 0.
BLLT	140614	BR	-	4	BRANCH ON L REGISTER .LT. 0.
BLGE	140615	BR	-	4	BRANCH ON L REGISTER .GE. 0.
BGE	140615	BR	-	4	BRANCH ON A REGISTER .GE. 0.
BLLE	140700	BR	-	4	BRANCH ON L REGISTER .LE. 0.
BLGT	140701	BR	-	4	BRANCH ON L REGISTER .GT. 0.

BLEQ	140702	BR	-	4	BRANCH ON L REGISTER .EQ. Ø.
BLNE	140703	BR	-	4	BRANCH ON L REGISTER .NE. Ø.
BDY	140724	BR	-	-	BRANCH ON DECREMENTED Y.
BDX	140734	BR	-	-	BRANCH ON DECREMENTED X.
BIY	141324	BR	-	-	BRANCH ON INCREMENTED Y.
BIX	141334	BR	-	-	BRANCH ON INCREMENTED X.
BCLE	141600	BR	-	-	BRANCH ON CONDITION CODE .LE.
BCGT	141601	BR	-	-	BRANCH ON CONDITION CODE .GT.
BMEQ	141602	BR	-	-	BRANCH ON MAGNITUDE-CONDITIONS L,CC .EQ.
BCEQ	141602	BR	-	-	BRANCH ON CONDITION CODE .EQ.
BCNE	141603	BR	-	-	BRANCH ON CONDITION CODE .NE.
BMNE	141603	BR	-	-	BRANCH ON MAGNITUDE-CONDITIONS L,CC .NE.
BCS	141604	BR	-	-	BRANCH ON CBIT SET.
BCGE	141605	BR	-	-	BRANCH ON CONDITION CODE .GE.
BFLE	141610	BR	-	4	BRANCH ON FLOATING ACCUMULATOR .LE. Ø.
BFGT	141611	BR	-	4	BRANCH ON FLOATING ACCUMULATOR .GT. Ø.
BFEQ	141612	BR	-	4	BRANCH ON FLOATING ACCUMULATOR .EQ. Ø.
BFNE	141613	BR	-	4	BRANCH ON FLOATING ACCUMULATOR .NE. Ø.
BFLT	141614	BR	-	4	BRANCH ON FLOATING ACCUMULATOR .LT. Ø.
BFGE	141615	BR	-	4	BRANCH ON FLOATING ACCUMULATOR .GE. Ø.
BCLT	141704	BR	-	-	BRANCH ON CONDITION CODE .LT.
BCR	141705	BR	-	-	BRANCH ON CBIT RESET.
BMGE	141706	BR	-	-	BRANCH ON MAGNITUDE-CONDITIONS L,CC .GE.
BLS	141706	BR	-	-	BRANCH ON LINK SET.
BLR	141707	BR	-	-	BRANCH ON LINK RESET.
BMLT	141707	BR	-	-	BRANCH ON MAGNITUDE-CONDITIONS L,CC .LT.
BMGT	141710	BR	-	-	BRANCH ON MAGNITUDE-CONDITIONS L,CC .GT.
BMLE	141711	BR	-	-	BRANCH ON MAGNITUDE-CONDITIONS L,CC .LE.

(CONTROL OPERATIONS.)

HLT	000000	CON	-	-	HALT COMPUTER OPERATION. RESTRICTED.
SVC	000505	CON	-	-	SUPERVISOR CALL.
IRTN	000601	CON	7	6	INTERRUPT RETURN, NO CAI. RESTRICTED.
IRTC	000603	CON	7	6	INTERRUPT RETURN, DO CAI. RESTRICTED.
ARGT	000605	CON	-	-	ARGUMENT TRANSFER (USED WITH PCL).
PRTN	000611	CON	7	6	PROCEDURE RETURN.
ITLB	000615	CON	-	-	INVALIDATE STLB ENTRY, L HAS VIRTUAL ADDRESS. RESTRICTED.
LPID	000617	CON	-	-	LOAD PROCESS ID FROM A01-A12. RESTRICTED.
CGT	001314	-	6	5	COMPUTED GO TO.

(FIELD OPERATIONS.)

EAFA	001300	AP	-	-	EFFECTIVE ADDRESS TO FIELD REGISTER ZERO.
ALFA	001301	FLD	6	5	ADD L TO FIELD ADDRESS REGISTER ZERO.
LDC	001302	FLD	-	7	LOAD CHAR TO A REG. AS SPECIFIED BY FIELD ADDRESS REG. ZERO.
LFLI	001303	-	-	-	LOAD FIELD LENGTH REGISTER IMMEDIATE ZERO.
EAFA	001310	AP	-	-	EFFECTIVE ADDRESS TO FIELD REGISTER ONE.
ALFA	001311	FLD	6	5	ADD L TO FIELD ADDRESS REGISTER ONE.
LDC	001312	FLD	-	7	LOAD CHAR TO A REG. AS SPECIFIED BY FIELD ADDRESS REG. ONE.

```
LFLI 001313 -    -  -   LOAD FIELD LENGTH REGISTER IMMEDIATE ONE.
STFA 001320 AP   -  -   STORE FIELD ADDRESS REGISTER ZERO.
TLFL 001321 FLD  -  -   TRANSFER L TO FIELD LENGTH REGISTER ZERO.
STC  001322 FLD  -  7   STORE CHAR FROM A REG. AS SPECIFIED BY FIELD
                        ADDRESS REG. ZERO.
TFLL 001323 FLD  -  -   TRANSFER FIELD LENGTH REG. TO L REG. ZERO.
STFA 001330 AP   -  -   STORE FIELD ADDRESS REGISTER ONE.
TLFL 001331 FLD  -  -   TRANSFER L TO FIELD LENGTH REGISTER ONE.
STC  001332 FLD  -  7   STORE CHAR FROM A REG. AS SPECIFIED BY FIELD
                        ADDRESS REG. ONE.
TFLL 001333 FLD  -  -   TRANSFER FIELD LENGTH REG. TO L REG. ONE.

(FLOATING-POINT OPERATIONS.)

FDBL 140016 FOPR -  -   CONVERT SINGLE FLOATING TO DOUBLE FLOATING.
                        FAC=>DFAC.
FCM  140530 FOPR 3  5   FLOATING COMPLEMENT.  -FAC=>FAC.
INTA 140531 FOPR 3  5   CONVERT FLOATING TO INTEGER.  INT(FAC)=>A.
FLTA 140532 FOPR 3  5   CONVERT INTEGER TO FLOATING.  FLOT(A)=>FAC.
INTL 140533 FOPR 3  5   CONVERT FLOATING TO LONG INTEGER.  INT(FAC)=>L.
FRN  140534 FOPR 3  5   FLOATING ROUND UP.
FLTL 140535 FOPR 8  5   CONVERT LONG INTEGER TO FLOATING.
                        FLOT(L)=>FAC.
FLOT 140550 FOPR 6  5   CONVERT DP INTEGER WITH HOLE TO FLOATING.
                        FLOT(A,B)=>FAC.
INT  140554 FOPR 3  5   CONVERT FLOATING TO DP INTEGER WITH HOLE.
                        INT(FAC)=>A,B.
DFCM 140574 FOPR 3  5   DOUBLE FLOATING COMPLEMENT.  -DFAC=>DFAC.

(FLOATING-POINT SKIP OPERATIONS.)

FSZE 140510 FSKP -  4   FLOATING SKIP IF .EQ. 0.
FSNZ 140511 FSKP -  4   FLOATING SKIP IF .NE. 0.
FSMI 140512 FSKP -  4   FLOATING SKIP IF .LT. 0.
FSPL 140513 FSKP -  4   FLOATING SKIP IF .GE. 0.
FSLE 140514 FSKP -  4   FLOATING SKIP IF .LE. 0.
FSGT 140515 FSKP -  4   FLOATING SKIP IF .GT. 0.

(INTEGRITY OPERATIONS.)

RMC  000021 IG   -  -   RESET MACHINE CHECK FLAG.  RESTRICTED.
VIRY 000311 IG   5  6   EXECUTE VERIFICATON ROUTINE.  RESTRICTED.
LMCM 000501 IG   -  -   LEAVE MACHINE CHECK MODE.  RESTRICTED.
EMCM 000503 IG   -  -   ENTER MACHINE CHECK MODE.  RESTRICTED.
MDEI 001304 IG   -  -   MEMORY DIAGNOSTIC ENABLE INTERLEAVE.
                        RESTRICTED.
MDII 001305 IG   -  -   MEMORY DIAGNOSTIC INHIBIT INTERLEAVE.
                        RESTRICTED.
MDRS 001306 IG   -  -   MEMORY DIAGNOSTIC READ SYNDROME BITS.
                        RESTRICTED.
MDWC 001307 IG   -  -   MEMORY DIAGNOSTIC LOAD WRITE CONTROL REGISTER.
                        RESTRICTED.
```

MDIW	0013??	IG	- -	MEMORY DIAGNOSTIC WRITE INTERLEAVED. L=>[E]. RESTRICTED.

(INPUT/OUTPUT OPERATIONS.)

ENB	000401	IO	- -	ENABLE INTERRUPTS. RESTRICTED.
CAI	000411	IO	- -	CLEAR ACTIVE INTERRUPT. RESTRICTED.
INH	001001	IO	- -	INHIBIT INTERRUPTS. RESTRICTED.

(LOGICIZE OPERATIONS.)

LLT	140410	LOG	- 4	LOGICIZE ON A REG. LT. IF A.LT.0, 1=>A ELSE 0=>A.
LLLT	140410	LOG	- 4	LOGICIZE ON L REG. LT. IF L.LT.0, 1=>A ELSE 0=>A.
LLE	140411	LOG	- 4	LOGICIZE ON A REG. LE. IF A.LE.0, 1=>A ELSE 0=>A.
LNE	140412	LOG	- 4	LOGICIZE ON A REG. NE. IF A.NE.0, 1=>A ELSE 0=>A.
LEQ	140413	LOG	- 4	LOGICIZE ON A REG. EQ. IF A.EQ.0, 1=>A ELSE 0=>A.
LGE	140414	LOG	- 4	LOGICIZE ON A REG. GE. IF A.GE.0, 1=>A ELSE 0=>A.
LLGE	140414	LOG	- 4	LOGICIZE ON L REG. GE. IF L.GE.0, 1=>A ELSE 0=>A.
LGT	140415	LOG	- 4	LOGICIZE ON A REG. GT. IF A.GT.0, 1=>A ELSE 0=>A.
LF	140416	LOG	- 5	LOGICIZE FALSE. 0=>A.
LT	140417	LOG	- 5	LOGICIZE TRUE. 1=>A.
LFLT	141110	LOG	- 4	LOGICIZE ON FLOATING LT. IF FAC.LT.0, 1=>A ELSE 0=>A.
LFLE	141111	LOG	- 4	LOGICIZE ON FLOATING LE. IF FAC.LE.0, 1=>A ELSE 0=>A.
LFNE	141112	LOG	- 4	LOGICIZE ON FLOATING NE. IF FAC.NE.0, 1=>A ELSE 0=>A.
LFEQ	141113	LOG	- 4	LOGICIZE ON FLOATING EQ. IF FAC.EQ.0, 1=>A ELSE 0=>A.
LFGE	141114	LOG	- 4	LOGICIZE ON FLOATING GE. IF FAC.GE.0, 1=>A ELSE 0=>A.
LFGT	141115	LOG	- 4	LOGICIZE ON FLOATING GT. IF FAC.GT.0, 1=>A ELSE 0=>A.
LCLT	141500	LOG	- -	LOGICIZE ON COND CODE LT. IF CC.LT.,1=>A ELSE 0=>A.
LCLE	141501	LOG	- -	LOGICIZE ON COND CODE LE. IF CC.LE.,1=>A ELSE 0=>A.
LCNE	141502	LOG	- -	LOGICIZE ON COND CODE NE. IF CC.NE.,1=>A ELSE 0=>A.
LCEQ	141503	LOG	- -	LOGICIZE ON COND CODE EQ. IF CC.EQ.,1=>A ELSE 0=>A.
LCGE	141504	LOG	- -	LOGICIZE ON COND CODE GE. IF CC.GE.,1=>A ELSE 0=>A.
LCGT	141505	LOG	- -	LOGICIZE ON COND CODE GT. IF CC.GT.,1=>A ELSE 0=>A.

```
LLLE 141511 LOG  -  4   LOGICIZE ON L REG. LE.  IF L.LE.Ø, 1=>A
                        ELSE Ø=>A.
LLNE 141512 LOG  -  4   LOGICIZE ON L REG. NE.  IF L.NE.Ø, 1=>A
                        ELSE Ø=>A.
LLEQ 141513 LOG  -  4   LOGICIZE ON L REG. EQ.  IF L.EQ.Ø, 1=>A
                        ELSE Ø=>A.
LLGT 141515 LOG  -  4   LOGICIZE ON L REG. GT.  IF L.GT.Ø, 1=>A
                        ELSE Ø=>A.
```

(MODE OPERATIONS.)

```
SGL  ØØØØØ5 MODE -  -   ENTER SINGLE-PRECISION MODE.
DBL  ØØØØØ7 MODE -  -   ENTER DOUBLE-PRECISION MODE (NOT USEFUL IN
                        64V MODE).
E64V ØØØØ1Ø MODE -  -   ENTER P4ØØ MODE.
E16S ØØØØ11 MODE -  -   ENTER P3ØØ 16K SECTORED MODE.
E32S ØØØØ13 MODE -  -   ENTER P3ØØ 32K SECTORED MODE.
ESIM ØØØ415 MODE -  -   ENTER STANDARD INTERRUPT MODE.  RESTRICTED.
EVIM ØØØ417 MODE -  -   ENTER VECTORED INTERUPT MODE.  RESTRICTED.
E64R ØØ1Ø11 MODE -  -   ENTER P3ØØ 64K RELATIVE MODE.
E32R ØØ1Ø13 MODE -  -   ENTER P3ØØ 32K RELATIVE MODE.
```

(MISCELLANEOUS OPERATIONS.)

```
NOP  ØØØØØ1 OPR  -  -   NO OPERATION.
PIMA ØØØØ15 OPR  3  5   LONG INTEGER TO SHORT INTEGER CONVERSION.
                        L=>A.  IEX ON PRECISION LOSS.
SCA  ØØØ041 OPR  -  -   LOAD P-3ØØ SHIFT COUNTER INTO A REG.
INK  ØØØ043 OPR  -  -   INPUT P-3ØØ KEYS INTO A REG.
NRM  ØØØ1Ø1 OPR  -  -   NORMALIZE A,B AS ON P-3ØØ (NOT USEFUL IN 64V
                        MODE).
RTN  ØØØ1Ø5 OPR  -  -   RETURN FROM P-3ØØ RECURSIVE PROCEDURE (NOT
                        USEFUL IN 64V MODE).
CEA  ØØØ111 OPR  -  -   CALCULATE EFFECTIVE ADDRESS.  A AS EA=>A.  (NOT
                        USEFUL IN 64V MODE.)
PIDA ØØØ115 OPR  -  -   SHORT INTEGER TO LONG INTEGER CONVERSION.
                        A=>L.
IAB  ØØØ2Ø1 OPR  -  -   INTERCHANGE A REG. WITH B REG.  A=>B & B=>A.
PIM  ØØØ2Ø5 OPR  -  -   CONVERT DP INTEGER WITH HOLE TO SHORT INTEGER.
PID  ØØØ211 OPR  -  -   CONVERT SHORT INTEGER TO DP INTEGER WITH HOLE.
PIML ØØØ3Ø1 OPR  3  5   CONVERT 64BIT INTEGER TO LONG INTEGER.
                        (L,E)=>L.
PIDL ØØØ3Ø5 OPR  -  -   CONVERT LONG INTEGER TO 64 BIT INTEGER.
OTK  ØØØ4Ø5 OPR  7  6   OUTPUT A REG. TO P-3ØØ KEYS AND SHIFT COUNTER.
TKA  ØØ1ØØ5 OPR  -  -   TRANSFER KEYS TO A.
TAK  ØØ1Ø15 OPR  7  6   TRANSFER A TO KEYS.
STEX ØØ1315 OPR  6  5   STACK EXTEND.  L REG. HAS EXTENT.
WCS  ØØ16XX -    -  -   WCS ENTRANCES.  UII ON NO WCS OR WCS NOT
                        LOADED.  MAY BE MICROPROGRAMMED TO BE
                        RESTRICTED.
CRL  14ØØ1Ø OPR  -  -   CLEAR L REGISTER.  Ø=>L.
```

	140014	OPR	-	-	Obsolete. Clears both the B-register and the least-significant word of the DFAC. See Section 2.8.
CRB	140015	OPR	-	-	CLEAR B REGISTER. Ø=>B.
CHS	140024	OPR	-	-	CHANGE SIGN OF A REGISTER.
CRA	140040	OPR	-	-	CLEAR A REGISTER. Ø=>A.
SSP	140100	OPR	-	-	SET SIGN OF A REG. PLUS. Ø=>ABIT1.
XCA	140104	OPR	-	-	INTERCHANGE AND CLEAR A. A=>B, Ø=>A.
S1A	140110	OPR	2	1	SUBTRACT 1 FROM A REGISTER. A-1=>A.
IRX	140114	OPR	-	-	INCREMENT X AND SKIP IF Ø.
RCB	140200	OPR	5	-	RESET CBIT. Ø=>CBIT.
XCB	140204	OPR	-	-	INTERCHANGE AND CLEAR B REG. B=>A, Ø=>B.
DRX	140210	OPR	-	-	DECREMENT X AND SKIP IF Ø.
CAZ	140214	OPR	1	1	COMPARE A WITH Ø. SKIP Ø,1,2 INST. IF A >,=,< Ø.
A2A	140304	OPR	2	1	ADD 2 TO A REGISTER. A+2=>A.
S2A	140310	OPR	2	1	SUBTRACT 2 FROM A REGISTER. A-2=>A.
TAB	140314	OPR	-	-	TRANSFER A TO B REG. A=>B.
CSA	140320	OPR	5	-	COPY SIGN OF A. A1=>CBIT, Ø=>A1.
CMA	140401	OPR	-	-	ONE'S COMPLEMENT A REGISTER.
TCA	140407	OPR	2	1	TWO'S COMPLEMENT A REGISTER. -A=>A.
SSM	140500	OPR	-	-	SET SIGN OF A REG. MINUS. 1=>A1.
TAX	140504	OPR	-	-	TRANSFER A REG. TO X REG. A=>X.
TAY	140505	OPR	-	-	TRANSFER A REG. TO Y REG. A=>Y.
SCB	140600	OPR	5	-	SET CBIT. 1=>CBIT.
TBA	140604	OPR	-	-	TRANSFER B REG. TO A REG. B=>A.
ADLL	141000	OPR	2	1	ADD LINK TO L REGISTER.
TXA	141034	OPR	-	-	TRANSFER X REG. TO A REG. X=>A.
CAR	141044	OPR	-	-	CLEAR A REG. RIGHT BYTE.
CAL	141050	OPR	-	-	CLEAR A REG. LEFT BYTE.
TYA	141124	OPR	-	-	TRANSFER Y REG. TO A REG. Y=>A.
ICL	141140	OPR	-	-	INTERCHANGE BYTES OF A REG. AND CLEAR LEFT BYTE.
A1A	141206	OPR	2	1	ADD 1 TO A REG. A+1=>A.
TCL	141210	OPR	2	1	TWO'S COMPLEMENT L. -L=>L.
ACA	141216	OPR	2	1	ADD CBIT TO A REG. CBIT+A=>A.
ICR	141240	OPR	-	-	INTERCHANGE BYTES OF A REG. AND CLEAR RIGHT BYTE.
ICA	141340	OPR	-	-	INTERCHANGE BYTES OF A REG.
CRE	141404	OPR	-	-	CLEAR E. Ø=>E.
CRLE	141410	OPR	-	-	CLEAR L AND E. Ø=>L, Ø=>E.
ILE	141414	OPR	-	-	INTERCHANGE L AND E. L=>E & E=>L.
(SHIFT OPERATIONS.)					
LRL	0400XX	SH	4	5	LONG RIGHT LOGICAL.
LRS	0401XX	SH	4	5	LONG RIGHT SHIFT (LONG INTEGER ARITHMETIC IN 64V MODE, ELSE DP INTEGER WITH HOLE).
LRR	0402XX	SH	4	5	LONG RIGHT ROTATE.
ARL	0404XX	SH	4	5	A RIGHT LOGICAL.
ARS	0405XX	SH	4	5	A RIGHT SHIFT (SHORT INTEGER ARITHMETIC).
ARR	0406XX	SH	4	5	A RIGHT ROTATE.
LLL	0410XX	SH	4	5	LONG LEFT LOGICAL.

LLS	0411XX	SH	4	5	LONG LEFT SHIFT (LONG INTEGER ARITHMETIC IN 64V MODE, ELSE DP INTEGER WITH HOLE).
LLR	0412XX	SH	4	5	LONG LEFT ROTATE.
ALL	0414XX	SH	4	5	A LEFT LOGICAL.
ALS	0415XX	SH	4	5	A LEFT SHIFT (SHORT INTEGER ARITHMETIC).
ALR	0416XX	SH	4	5	A LEFT ROTATE.

(SKIP OPERATIONS.)

SKP	100000	SKP	-	-	SKIP ONE WORD.
SRC	100001	SKP	-	-	SKIP IF CBIT RESET.
SR4	100002	SKP	-	-	SKIP IF SENSE SWITCH 4 RESET. RESTRICTED.
SR3	100004	SKP	-	-	SKIP IF SENSE SWITCH 3 RESET. RESTRICTED.
SR2	100010	SKP	-	-	SKIP IF SENSE SWITCH 2 RESET. RESTRICTED.
SR1	100020	SKP	-	-	SKIP IF SENSE SWITCH 1 RESET. RESTRICTED.
SSR	100036	SKP	-	-	SKIP IF SENSE SWITCHES 1,2,3 AND 4 RESET. RESTRICTED.
SZE	100040	SKP	-	-	SKIP IF A REG. .EQ. 0.
SLZ	100100	SKP	-	-	SKIP IF A REG. BIT 16 .EQ. 0.
SMCR	100200	SKP	-	-	SKIP IF MACHINE CHECK RESET.
SGT	100220	SKP	-	-	SKIP IF A REG. .GT. 0.
SNR	10024X	SKP	-	-	SKIP IF SENSE SWITCH N RESET. RESTRICTED.
SAR	10026X	SKP	-	-	SKIP IF A REG. BIT N RESET.
SPL	100400	SKP	-	-	SKIP IF A REG. .GE. 0.
SSC	101001	SKP	-	-	SKIP IF CBIT SET.
SS4	101002	SKP	-	-	SKIP IF SENSE SWITCH 4 SET. RESTRICTED.
SS3	101004	SKP	-	-	SKIP IF SENSE SWITCH 3 SET. RESTRICTED.
SS2	101010	SKP	-	-	SKIP IF SENSE SWITCH 2 SET. RESTRICTED.
SS1	101020	SKP	-	-	SKIP IF SENSE SWITCH 1 SET. RESTRICTED.
SSS	101036	SKP	-	-	SKIP IF SENSE SWITCHES 1,2,3 AND 4 SET. RESTRICTED.
SNZ	101040	SKP	-	-	SKIP IF A REG. NE. 0.
SLN	101100	SKP	-	-	SKIP IF A REG. BIT 16 SET.
SMCS	101200	SKP	-	-	SKIP IF MACHINE CHECK SET.
SLE	101220	SKP	-	-	SKIP IF A REG. .LE. 0.
SNS	101240	SKP	-	-	SKIP IF SENSE SWITCH N SET. RESTRICTED.
SAS	101260	SKP	-	-	SKIP IF A REG. BIT N SET.
SMI	101400	SKP	-	-	SKIP IF A REG. .LT. 0.

TABLE 2.3.
PRIME 400 MEMORY-REFERENCE INSTRUCTIONS
(WHEN IN 64V MODE).

instruction bits 3-6	instruction bits 13-14 (if S=1 and -256<=D<-224) 00	01	10	11
0001	JMP	EAL	XEC	-
0010	LDA	FLD	DFLD	LDL
0011	ANA	STLR	ORA	ANL
0100	STA	FST	DFST	STL
0101	ERA	LDLR	-	ERL
0110	ADD	FAD	DFAD	ADL
0111	SUB	FSB	DFSB	SBL
1000	JST	-	PCL	-
1001	CAS	FCS	DFCS	CLS
1010	IRS	MIA	EAXB	-
1011	IMA	MIB	EALB	-
1100	JSY	EIO	JSXB	-
1101*	STX	FLX	DFLX	-
1101**	LDX	LDY	STY	JSX
1110	MPY	FMP	DFMP	MPL
1111	DIV	FDV	DFDV	DVL

Use column 00 if S (bit 7) is 0 or if D (bits 8-16) is not in the range -256 <= D < -224.

*: Use this row if bit 2 of the instruction word is a zero. These instructions cannot be indexed.

**: Use this row if bit 2 of the instruction word is a one. These instructions cannot be indexed.

TABLE 2.4.
PRIME 400 MEMORY-REFERENCE INSTRUCTION DESCRIPTIONS.

This list summarizes all PRIME 400 memory-reference and programmed input/output instructions, sorted by operation code. The first number in the "opcode" column is the octal representation of instruction bits 3-6. The second number is the octal representation of bits 13-14 (bits 13-14 are inspected only if bits 6-11 are 11000, i.e., S=1 and -256 <= D < -224). Instructions marked "restricted" can be executed only in ring zero.

The "type" column indicates the format of the operation as follows.
MR: Memory-reference operation.
PIO: Programmed input/output operation.

The "C" column indicates the effect of the operation on the C-bit and the L-bit as follows.
-: C and L are unchanged by the operation.
1: C is unchanged, L is carry.
2: C is overflow, L is carry.
3: C is overflow, L is indeterminant.
6: C and L are indeterminant.
7: C and L are loaded by the operation.

The "cc" column indicates the effect of the operation on the condition codes as follows.
-: Condition codes are unchanged by the operation.
1: Condition codes indicate the result of the arithmetic operation or compare.
5: Condition codes are indeterminant.
6: Condition codes are loaded by the operation.
7: Condition codes indicate the result of the operation.

The "avail" column indicates in which addressing modes the operation is available as follows.
S: The operation is available in 16S and 32S modes.
R: The operation is available in 32R and 64R modes.
V: The operation is available in 64V mode.

mnem	opcode	type	C	cc	avail	description
JMP	01	MR	-	-	SRV	Unconditional jump. EA => PB,P.
EAA	01 01	MR	-	-	R	Effective address to A-register. EA => A.
EAL	01 01	MR	-	-	V	Load effective address. EA => A,B.
XEC	01 02	MR	-	-	RV	Execute instruction at effective address. Not all instructions can be executed.
ENTR	01 03	MR	-	-	R	Enter P-300 recursive procedure (use with CREP and RTN). (S) => [(S)-EA]16, (S)-EA => S.
LDA	02	MR	-	-	SRV	Load A-register. [EA]16 => A.
DLD	02(DP)	MR	-	-	SR	Double load. [EA]32 => A,B.
FLD	02 01	MR	-	-	RV	Floating load. [EA]32 => FAC.
DFLD	02 02	MR	-	-	RV	Double floating load. [EA]64 => DFAC.

LDL	Ø2 Ø3	MR	-	-	V	Load long. [EA]32 => A,B.
JEQ	Ø2 Ø3	MR	-	-	R	Jump if equal. If (A) .EQ. Ø, EA => P.
ANA	Ø3	MR	-	-	SRV	AND. (A) .AND. [EA]16 => A.
STLR	Ø3 Ø1	MR	-	-	V	Store long into register-file location EA.
ORA	Ø3 Ø2	MR	-	-	V	OR. (A) .OR. [EA]16 => A.
ANL	Ø3 Ø3	MR	-	-	V	AND long. (A,B) .AND. [EA]32 => A,B.
JNE	Ø3 Ø3	MR	-	-	R	Jump if not equal. If (A) .NE. Ø, EA => P.
STA	Ø4	MR	-	-	SRV	Store A-register. (A) => [EA]16.
DST	Ø4(DP)	MR	-	-	SR	Double store. (A,B) => [EA]32.
FST	Ø4 Ø1	MR	6	6	RV	Floating store. (FAC) => [EA]32.
DFST	Ø4 Ø2	MR	-	-	RV	Double floating store. (DFAC) => [EA]64.
STL	Ø4 Ø3	MR	-	-	V	Store long. (A,B) => [EA]32.
JLE	Ø4 Ø3	MR	-	-	R	Jump if less-equal. If (A) .LE. Ø, EA => P.
ERA	Ø5	MR	-	-	SRV	Exclusive OR. (A) .XOR. [EA]16 => A.
LDLR	Ø5 Ø1	MR	-	-	V	Load long from register-file location EA.
ERL	Ø5 Ø3	MR	-	-	V	Exclusive OR long. (A,B) .XOR. [EA]32 => A,B.
JGT	Ø5 Ø3	MR	-	-	R	Jump if greater. If (A) .GT. Ø, EA => P.
ADD	Ø6	MR	2	1	SRV	Add. (A) + [EA]16 => A.
DAD	Ø6(DP)	MR	2	1	SR	Double add (with hole). (A,B) + [EA]32 => A,B.
FAD	Ø6 Ø1	MR	3	5	RV	Floating add. (FAC) + [EA]32 => FAC.
DFAD	Ø6 Ø2	MR	3	5	RV	Double floating add. (DFAC) + [EA]64 => DFAC.
ADL	Ø6 Ø3	MR	2	1	V	Add long (no hole). (A,B) + [EA]32 => A,B.
JLT	Ø6 Ø3	MR	-	-	R	Jump if less. If (A) .LT. Ø, EA => P.
SUB	Ø7	MR	2	1	SRV	Subtract. (A) - [EA]16 => A.
DSB	Ø7(DP)	MR	2	1	SR	Double subtract (with hole). (A,B) - => A,B.
FSB	Ø7 Ø1	MR	3	5	RV	Floating subtract. (FAC) - [EA]32 => FAC.
DFSB	Ø7 Ø2	MR	3	5	RV	Double floating subtract. (DFAC) - [EA]64 => DFAC.
SBL	Ø7 Ø3	MR	2	1	V	Subtract long (no hole). (A,B) - [EA]32 => A,B.
JGE	Ø7 Ø3	MR	-	-	R	Jump if greater-equal. If (A) .GE. Ø, EA => P.
JST	1Ø	MR	-	-	SRV	Jump and store. (P) => [EA]16, EA+1 => P.
CREP	1Ø Ø2	MR	-	-	R	Call P-3ØØ recursive procedure (use with ENTR and RTN). (P) => [(S)+1]16, EA => P.
PCL	1Ø Ø2	MR	7	6	V	P-4ØØ procedure call.
CAS	11	MR	1	1	SRV	Compare. Skip Ø,1,2 instructions if (A) >,=,< [EA]16.
FCS	11 Ø1	MR	6	5	RV	Floating compare. Skip Ø,1,2 instructions if (FAC) >,=,< [EA]32.
DFCS	11 Ø2	MR	6	5	RV	Double floating compare. Skip Ø,1,2 instructions if (DFAC) >,=,< [EA]64.
CLS	11 Ø3	MR	1	1	V	Compare long (no hole). Skip Ø,1,2 instructions if (A,B) >,=,< [EA]32.
IRS	12	MR	-	-	SRV	Increment, replace, and skip if zero.
MIA	12 Ø1	MR	-	-	V	Microcode entrance. May be microprogrammed to be restricted.

EAXB	12 Ø2	MR	-	-	V	Effective address to temporary base register. EA => XB.
IMA	13	MR	-	-	SRV	Interchange memory and A-register.
MIB	13 Ø1	MR	-	-	V	Microcode entrance. May be microprogrammed to be restricted.
EALB	13 Ø2	MR	-	-	V	Effective address to linkage base register. EA => LB.
JSY	14	MR	-	-	V	Jump and set Y-register. (P) => Y, EA => P.
OCP	14	PIO	-	-	SR	Output control pulse. Restricted.
EIO	14 Ø1	MR	-	7	V	Execute EA as I/O instruction. Restricted. Set cc equal if P-3ØØ would have skipped.
JSXB	14 Ø2	MR	-	-	V	Jump and set temporary base register. (PB,P) => XB, EA => PB,P.
STX	15	MR	-	-	SRV	Store X-register. (X) => [EA]16.
FLX	15 Ø1	MR	-	-	RV	Load floating index. 2*[EA]16 => X.
DFLX	15 Ø2	MR	-	-	V	Load double floating index. 4*[EA]16 => X.
JDX	15 Ø2	MR	-	-	R	Jump on decremented X-register zero.
JIX	15 Ø3	MR	-	-	R	Jump on incremented X-register zero.
MPY	16	MR	3	1	SR	Multiply (with hole). (A) * [EA]16 => A,B.
MPY	16	MR	-	1	V	Multiply (integer, no hole). (A) * [EA]16 => A,B.
FMP	16 Ø1	MR	3	5	RV	Floating multiply. (FAC) * [EA]32 => FAC.
DFMP	16 Ø2	MR	3	5	RV	Double floating multiply. (DFAC) * [EA]64 => DFAC.
MPL	16 Ø3	MR	-	1	V	Multiply long (integer, no hole). (A,B) * [EA]32 => A,B,EH,EL.
DIV	17	MR	3	5	SR	Divide (with hole). (A,B) / [EA]16 => A; remainder => B.
DIV	17	MR	3	5	V	Divide (integer, no hole). (A,B) / [EA]16 => A; remainder => B.
FDV	17 Ø1	MR	3	5	RV	Floating divide. (FAC) / [EA]32 => FAC.
DFDV	17 Ø2	MR	3	5	RV	Double floating divide. (DFAC) / [EA]64 => DFAC.
DVL	17 Ø3	MR	3	5	V	Divide long (integer, no hole). (A,B,EH,EL) / [EA]32 => A,B; remainder => EH,EL.
SMK	17ØØ2Ø	PIO	-	-	SR	Set interrupt masks. Restricted.
SKS	34	PIO	-	-	SR	Skip if condition set. Restricted.
LDX	35	MR	-	-	SRV	Load X-register. [EA]16 => X.
LDY	35 Ø1	MR	-	-	V	Load Y-register. [EA]16 => Y.
STY	35 Ø2	MR	-	-	V	Store Y-register. (Y) => [EA]16.
JSX	35 Ø3	MR	-	-	RV	Jump and set X-register. (P) => X, EA => P.
INA	54	PIO	-	-	SR	Input to A-register. Restricted.
OTA	74	PIO	-	-	SR	Output from A-register. Restricted.

2.3 PRIME 400 EFFECTIVE ADDRESS CALCULATION.

Effective address calculation in 64V mode proceeds as follows: A PRIME 400 virtual address is 30 bits long total, comprising a 2-bit ring number field, a 12-bit segment number field, and a 16-bit word number field. The segment and word number fields define what word of virtual memory is to be accessed, and the ring field governs what access privileges will be granted.

For the purposes of regulating access, the PRIME 400 has three rings (numbered 0, 1, and 3) of nested, sequentially more restricted privilege. Ring 0 is the most privileged, and ring 3 the least. Restricted instructions can be executed only in ring 0. If the effective address has a ring field of 0 or if virtual memory operation is turned off by bit 14 of the processor modals, access is unrestricted (reading, writing, and executing are permitted, regardless of what segment is being accessed). If the effective address has a ring field of 1 or 3, the access granted is determined by bits 18-20 or 24-26 respectively of the accessed segment's segment descriptor word (SDW), as shown in Table 2.1. Attempted references which are in violation of the granted access privileges are suppressed and an access violation fault occurs.

The four base registers provided in the PRIME 400 are referred to as shown in Table 2.5. Each base register is 32 bits wide and holds a complete virtual address as shown in Table 2.5. Indirect words fetched during instruction address preparation also specify (either explicitly or implicitly) a complete virtual address. Their format is also shown in Table 2.5. Note that there are two types of indirect words: 16-bit type, which are referenced by one-word instructions (S=0 or D not in the range -256 <= D < -224) and which always implicitly refer to the same segment as the instruction; and 48-bit type, which are referenced by two-word instructions (S=1 and -256 <= D < -224) and which explicitly refer to a segment by number.

Address calculation is broken down into many cases below. In all of these cases, the following symbols are used:

I	Instruction bit 1, the indirect bit.
X	Instruction bit 2, the index bit (except X is always considered zero when bits 3-6 are 1101, the instructions which cannot be indexed).
S	Instruction bit 7, the sector bit.
D	Instruction bits 8-16, read as a two's complement displacement (-256 <= D < +256). In a very few instances (all explicitly noted below), D is treated as an unsigned integer (0 <= D < +512).

LIVE	The number of accessible live registers. If virtual memory is in operation (bit 14 of the processor modals), the value of LIVE is 8. If virtual memory is not in operation, the value is 32.
(X)	Contents of the X-register (16-bits).
(Y)	Contents of the Y-register (16-bits).
(PB)R	Ring number field of the procedure base register (2 bits).
(PB)S	Segment number field of the procedure base register (12 bits).
(PB)W	Always taken as 16 bits of zeros. The procedure base register behaves as if its word offset is always zero.
(SB)R, (SB)S, (SB)W	The ring number, segment number, and word number fields of the stack base register.
(LB)R, (LB)S, (LB)W	The ring number, segment number, and word number fields of the temporary base register.
(P)	Contents of the program counter after it has been incremented past the current instruction (16 bits).
(EA)R, (EA)S, (EA)W	The ring number, segment number, and word number fields of the calculated effective address.
(temp)R, (temp)S, (temp)W	The ring number, segment number, and word number of a temporary effective address from which an indirect word must be fetched.
(live register ?)	The 16-bit contents of live register number ? (Ø<=?<LIVE).
[r,s,w]	The 16-bit contents of word w of segment s, from ring r (i.e., ring r must have read access to segment s or else a fault occurs).
(base register ?)R, (base register ?)S, (base register ?)W	The ring number, segment number, and word number fields of base register ?, where ? of Ø is the procedure base, 1 is the stack

	base, 2 is the linkage base, and 3 is the temporary base. The word number portion of the procedure base is always taken as zero.
RING	The current ring of execution (2 bits). This is not necessarily the same as (PB)R, since (PB)R can be changed at any time without special privileges.
SEG	The current procedure segment number (12 bits). As with RING, this is not necessarily the same as (PB)S.
[r,s,w]R	Bits 2-3 of word w of segment s, from ring r (the ring number field of a 48-bit indirect word).
[r,s,w]S	Bits 5-16 of word w of segment s, from ring r (the segment number field of a 48-bit indirect word).
[r,s,w]W	Bits 1-16 of word w+1 of segment s, from ring r (the word number field of a 48-bit indirect word).
A	16-bit word offset from the second word of the instruction (present only when S=1 and -256<=D<-224).
Y	Instruction bit 12, the extended index bit (inspected only when S=1 and -256<=D<-224).
ZZ	Instruction bits 15-16, the base register bits (inspected only when S=1 and -256<=D<-224). ZZ of Ø is the procedure base, 1 is the stack base, 2 is the linkage base, and 3 is the temporary base.

All additions shown below are modulo 65536, i.e., 16-bit arithmetic with overflows ignored. On all fetches shown for indirect words, read access is required of the segment containing the indirect word. Access to live registers (which are not contained in any segment) is always read, execute, and write.

(S=Ø AND I=Ø CASES.)

In these cases, access is to any of 256 locations offset from either the stack base register or the linkage base register. Indexing can modify which location is selected. These cases allow an efficient,

one-word format of instruction addressing to the most often accessed locations of a procedure stack or linkage area.

Case 1. S=Ø, I=Ø, X=Ø.

If Ø<=D<LIVE, then the effective address is live register D.
If LIVE<=D, then the effective address is given by
    (EA)R <- MAX (RING, (SB)R);
    (EA)S <- (SB)S;
    (EA)W <- (SB)W + D.
If D<Ø, then the effective address is given by
    (EA)R <- MAX (RING, (LB)R);
    (EA)S <- (LB)S;
    (EA)W <- (LB)W + D + 512.
Note that use of D+512 is the same as interpreting D as an unsigned integer.

Case 2. S=Ø, I=Ø, X=1.

If D<Ø, then the effective address is given by
    (EA)R <- MAX (RING, (LB)R);
    (EA)S <- (LB)S;
    (EA)W <- (LB)W + D + 512 + (X).
Else if D+(X)<LIVE, then the effective address is live register D+(X).
Else, the effective address is given by
    (EA)R <- MAX (RING, (SB)R);
    (EA)S <- (SB)S;
    (EA)W <- (SB)W + D + (X).
Note that the decision to use the linkage base register is made prior to indexing.

(S=Ø AND I=1 CASES.)

In these cases, all references, both intermediate and final, are to the live registers (addresses less than LIVE) or to the procedure segment (all others). Indirect words are of the 16-bit format.

Case 3. S=Ø, I=1, X=Ø.

If Ø<=D<LIVE, then the location of the 16-bit indirect word is live register D, so set (temp)W via
    (temp)W <- D
and go to Case 4 to perform the indirect cycle.
Else, the location of the 16-bit indirect word is word D of the procedure segment, considering D as an unsigned 9-bit number (Ø<=D<+512), so set (temp)W via
    (temp)W <- D treated as an unsigned 9-bit number (Ø<=D<+512)
and go to Case 5 to perform the indirect cycle.

Case 4. Indirect through live register (temp)W.

If (live register (temp)W)<LIVE, then the effective address is live register (live register (temp)W).
Else, the effective address is in the procedure segment and is given by
(EA)R <- RING;
(EA)S <- SEG;
(EA)W <- (live register (temp)W).

Case 5. Indirect through 16-bit word of procedure segment.

If [RING, SEG, (temp)W] < LIVE, then the effective address is live register [RING, SEG, (temp)W].
Else, the effective address is in the procedure segment and is given by
(EA)R <- RING;
(EA)S <- SEG;
(EA)W <- [RING, SEG, (temp)W].

Case 6. S=Ø, I=1, X=1.

If Ø<=D<64, pre-indexing is selected. In this case, if D+(X)<LIVE, then the 16-bit indirect word is in live register D+(X), so set (temp)W via
(temp)W <- D + (X)
and go to Case 4 to perform the indirect cycle. But if LIVE<=D+(X), then the 16-bit indirect word is in the procedure segment at word D+(X), so set (temp)W via
(temp)W <- D + (X)
and go to Case 5 to perform the indirect cycle.

On the other hand, if D is not in the range Ø<=D<64, then post-indexing is selected, and the 16-bit indirect word is in the procedure segment at word D, treating D as an unsigned 9-bit number (Ø<=D<+512). In this case, if [RING, SEG, D unsigned]+(X) is less than LIVE, the effective address is live register [RING, SEG, D unsigned]+(X). But if [RING, SEG, D unsigned]+(X) is greater than or equal to LIVE, the effective address is in the procedure segment and is given by
(EA)R <- RING;
(EA)S <- SEG;
(EA)W <- [RING, SEG, D unsigned]+(X).

(S=1 AND -224<=D<+256 CASES.)

In these cases, indexing is always post-indexing, and all references are either to a live register (for word number less than LIVE) or to the procedure segment (for word number greater than or equal to LIVE). Any indirects are through 16-bit indirect words. The intermediate

address is relative to the program counter after it has been advanced to the next instruction.

Case 7. S=1, -224<=D<+256, I=Ø.

In this case there is no indirection, so the intermediate address is the final address. If X=Ø there is no indexing, so the intermediate address is

(temp)W <- (P) + D;

but if X=1 there is indexing, and the intermediate address is

(temp)W <- (P) + D + (X).

If (temp)W<LIVE then the effective address is live register (temp)W. But if LIVE<=(temp)W, then the effective address is given by

(EA)R <- RING;
(EA)S <- SEG;
(EA)W <- (temp)W.

Case 8. S=1, -224=D<+256, I=1, X=Ø.

The intermediate address is (P)+D.

If (P)+D<LIVE, the 16-bit indirect word is in live register (P)+D. In this case, if (live register (P)+D)<LIVE, the effective address is live register (live register (P)+D). But if LIVE<=(live register (P)+D), the effective address is given by

(EA)R <- RING;
(EA)S <- SEG;
(EA)W <- (live register (P)+D).

On the other hand, if LIVE<=(P)+D, the 16-bit indirect word is in the procedure segment at word (P)+D. In this case, if [RING, SEG, (P)+D]<LIVE, the effective address is live register [RING, SEG, (P)+D]. But if LIVE<=[RING, SEG, (P)+D], the effective address is given by

(EA)R <- RING;
(EA)S <- SEG;
(EA)W <- [RING, SEG, (P)+D].

Case 9. S=1, -224<=D<+256, I=1, X=1.

The intermediate address is (P)+D.

If (P)+D<LIVE, the 16-bit indirect word is in live register (P)+D. In this case, if (live register (P)+D)+(X)<LIVE, the effective address is live register (live register (P)+D)+(X). But if LIVE<=(live register (P)+D)+(X), the effective address is given by

(EA)R <- RING;
(EA)S <- SEG;
(EA)W <- (live register (P)+D) + (X).

On the other hand, if LIVE<=(P)+D, the 16-bit indirect word is in the

procedure segment at word (P)+D. In this case, if [RING, SEG, (P)+D]+(X)<LIVE, the effective address is live register [RING, SEG, (P)+D]+(X). But if LIVE<=[RING, SEG, (P)+D]+X), the effective address is given by

(EA)R <- RING;
(EA)S <- SEG;
(EA)W <- [RING, SEG, (P)+D] + (X).

(S=1 AND -256<=D<-224 CASES.)

In these cases, the instruction is extended by an additional 16-bit word, A, giving a full 65536 range of word number, and bit 12, Y, and bits 15-16, ZZ, of the instruction word team up with bits 1, I, and 2, X, to specify all 32 combinations of base registers, direct or indirect, and pre- or post-indexing by either the X-register or the Y-register. Also in these cases, all indirect words are of the 48-bit format, and neither indirect words nor final effective addresses ever land in the live registers.

Case 10. S=1, -256<=D<-224, I=0, X=0, Y=0.

Direct addressing. The effective address is given by

(EA)R <- MAX(RING, (base register ZZ)R);
(EA)S <- (base register ZZ)S;
(EA)W <- (base register ZZ)W + A.

Case 11. S=1, -256<=D<-224, I=0, X=0, Y=1.

Indexing by the Y-register. The effective address is given by

(EA)R <- MAX(RING, (base register ZZ)R);
(EA)S <- (base register ZZ)S;
(EA)W <- (base register ZZ)W + A + (Y).

Case 12. S=1, -256<=D<-224, I=0, X=1, Y=0.

Indexing by the X-register. The effective address is given by

(EA)R <- MAX(RING, (base register ZZ)R);
(EA)S <- (base register ZZ)S;
(EA)W <- (base register ZZ)W + A + (X).

Case 13. S=1, -256<=D<-224, I=0, X=1, Y=1.

Indirect. The location of the 48-bit indirect word is given by

(temp)R <- MAX(RING, (base register ZZ)R);
(temp)S <- (base register ZZ)S;

(temp)W <- (base register ZZ)W + A.

The effective address is given by
(EA)R <- MAX(RING, [(temp)R, (temp)S, (temp)W]R);
(EA)S <- [(temp)R, (temp)S, (temp)W]S;
(EA)W <- [(temp)R, (temp)S, (temp)W]W.

Case 14. S=1, -256<=D<-224, I=1, X=Ø, Y=Ø.

Pre-indexed by the Y-register. The location of the 48-bit indirect word is given by
(temp)R <- MAX(RING, (base register ZZ)R);
(temp)S <- (base register ZZ)S;
(temp)W <- (base register ZZ)W + A + (Y).

The effective address is given by
(EA)R <- MAX(RING, [(temp)R,(temp)S, (temp)W]R);
(EA)S <- [(temp)R, (temp)S, (temp)W]S;
(EA)W <- [(temp)R, (temp)S, (temp)W]W.

Case 15. S=1, -256<=D<-224, I=1, X=Ø, Y=1.

Post-indexed by the Y-register. The location of the 48-bit indirect word is given by
(temp)R <- MAX(RING, (base register ZZ)R);
(temp)S <- (base register ZZ)S;
(temp)W <- (base register ZZ)W + A.

The effective address is given by
(EA)R <- MAX(RING, [(temp)R, (temp)S, (temp)W]R);
(EA)S <- [(temp)R, (temp)S, (temp)W]S;
(EA)W <- [(temp)R, (temp)S, (temp)W]W + (Y).

Case 16. S=1, -256<=D<-224, I=1, X=1, Y=Ø.

Pre-indexed by the X-register. The location of the 48-bit indirect word is given by
(temp)R <- MAX(RING, (base register ZZ)R);
(temp)S <- (base register ZZ)S;
(temp)W <- (base register ZZ)W + A + (X).

The effective address is given by
(EA)R <- MAX(RING, [(temp)R, (temp)S, (temp)W]R);
(EA)S <- [(temp)R, (temp)S, (temp)W]S;
(EA)W <- [(temp)R, (temp)S, (temp)W]W.

Case 17. S=1, -256<=D<-224, I=1, X=1, Y=1.

Post-indexed by the X-register. The location of the 48-bit indirect word is given by

(temp)R <- MAX(RING, (base register ZZ)R);
(temp)S <- (base register ZZ)S;
(temp)W <- (base register ZZ)W + A.

The effective address is given by

(EA)R <- MAX(RING, [(temp)R, (temp)S, (temp)W]R);
(EA)S <- [(temp)W, (temp)S, (temp)W]S;
(EA)W <- [(temp)W, (temp)S, (temp)W]W + (X).

TABLE 2.5.
PRIME 400 ADDRESS CALCULATION FORMATS.

PRIME 400 BASE REGISTERS

number	mnemonic	name
0	PB	Procedure base.
1	SB	Stack base.
2	LB	Linkage base.
3	XB	Temporary base.

PRIME 400 EFFECTIVE ADDRESS
OR
BASE REGISTER CONTENTS

0RR0SSSSSSSSSSS
WWWWWWWWWWWWWWW

1:	Not used, will be zero.
2,3:	Ring of privilege (0, 1, or 3) (RR).
4:	Not used, will be zero.
5-16:	Segment number (SSS...S).
17-32:	Word number (WWW...W).

INDIRECT WORD FOR
PRIME 400 ONE-WORD INSTRUCTIONS

This form of indirect word is used when referenced by instruction words with S (bit 7) of 0 or D (bits 8-16) not in the range -256 <= D < -224.

WWWWWWWWWWWWWWW

1-16:	Word number within procedure segment (WWW...W).

INDIRECT WORD
FOR PRIME 400 TWO-WORD INSTRUCTIONS

This form of indirect word is used in the PRIME 400 when referenced by instruction words with S (bit 7) of 1 and D (bits 8-16) in the range -256 <= D -224.

```
FRRESSSSSSSSSSS
WWWWWWWWWWWWWWW
BBBB----------
```

1: Pointer fault if set (F). In the fault case, the entire first word (bits 1-16) forms a fault code, and no other bits are inspected.

2,3: Ring of privilege (RR).

4: If zero, no third word is present and the bit number (BBBB) of the effective address is taken as zero. If one, the third word is present and gives the bit number (E).

5-16: The segment number portion of the effective address (SSS...S).

17-32: Word number (WWW...W).

33-36: If bit 4 (E) is a one, the bit number (BBBB).

37-48: Must be zero.

TABLE 2.6.
PRIME 300/400 REGISTER CORRESPONDENCE.

This table expresses the relation between the PRIME 300 and PRIME 400 program-visible registers, their memory addresses for use in short-form instructions, and their register-file address assignments for use in LDLR and STLR instructions. Not all registers can be referenced all ways. Note that some assignments overlap.

relative register-file address (LDLR/STLR)	memory address (short-form instructions)	register usage
-	7	P (program counter)
2 H	1	A (accumulator)
2 L	2	B (double-precision and long accumulator extension)
3 H,L	-	EH,EL (accumulator extension for MPL,DVL)
5 H	3	S (stack), Y (alternate index)
7 H	Ø	X (index)
1Ø H	13	-
1Ø,11	-	FARØ,FLRØ (field address and length register Ø)
12,13	-	FAR1,FLR1 (field address and length register 1)
12 H	4	FAC (floating accumulator, mantissa high)
12 L	5	FAC (mantissa middle)
13 H	6	FAC (exponent)
13 L	-	FAC (mantissa low, double-precision)
14 H,L	-	PB (procedure base)
15 H,L	14,15	SB (stack base)
16 H,L	16,17	LB (linkage base)
17 H,L	-	XB (temporary base)
2Ø H	1Ø	(high half of DTAR3)
2Ø H,L	-	DTAR3 (descriptor table address, segments 3Ø72-4Ø95)
21 H,L	-	DTAR2 (segments 2Ø48-3Ø71)
22 H,L	-	DTAR1 (segments 1Ø24-2Ø47)
23 H,L	-	DTARØ (segments Ø-1Ø23)
24 H,L	-	keys, modals (see Table 2.9)
25 H,L	-	OWNER (address of process control block of process owning register contents)
26 H	11	FCODE (fault code)
27 H,L	-	FADDR (fault address)
27 L	12	(fault address word number)
3Ø H	-	process 1Ø24-microsecond c.p.u. timer

Register-file addresses used in LDLR and STLR instructions are doubleword addresses. The notation "2 H" means the high or left 16 bits of register-file address 2, while "2 L" means the low or right 16 bits.

Registers are addressed by memory-reference instructions when the effective address is less than LIVE (Section 2.3) and the processor is in other than 64V mode or is in 64V mode cases 3 through 9 (Section 2.3).

The following registers should not be written into by STLR instructions, or anomalous behavior will result.

PB: The procedure base should be changed only via LPSW or programmed transfers of control.

keys: The keys should be changed only via LPSW or the various mode control operations.

modals: The modals should be changed only via LPSW or the various mode control operations. In no case should an LPSW ever attempt to change the current register set bits of the modals.

2.4 GENERIC-AP INSTRUCTIONS.

There has arisen a need for more instructions in the PRIME 400 which develop effective addresses than can be accommodated in the memory-reference class. These instructions are provided by a class of generic operation codes which are followed by a so-called AP pointer. The format of an AP pointer is:

```
BBBBI-ZZ--------
WWWWWWWWWWWWWWWW
```

where bits 1-4 (BBBB) are the bit number field, bit 5 (I) is the indirect bit, bits 7-8 (ZZ) are the base register field, and bits 17-32 (WWW...W) are the word number field. The remaining bits must be zero. The effective address defined by an AP pointer is calculated as follows.

If I is zero, there is no indirect cycle, and the effective address is

```
(EA)R <- MAX( RING , (base register ZZ)R );
(EA)S <- (base register ZZ)S;
(EA)W <- (base register ZZ)W + WWW...W;
(EA)B <- BBBB.
```

Otherwise, if I is one, there is an indirect cycle. The address of the indirect word is

```
(temp)R <- MAX( RING , (base register ZZ)R );
(temp)S <- (base register ZZ)S;
(temp)W <- (base register ZZ)W + WWW...W;
```

and the effective address is

```
(EA)R <- MAX( RING , [(temp)R,(temp)S,(temp)W]R );
(EA)S <- [(temp)R,(temp)S,(temp)W]S;
(EA)W <- [(temp)R,(temp)S,(temp)W]W;
(EA)B <- [(temp)R,(temp)S,(temp)W]B.
```

Note that: no bit numbers are ever extracted from base registers (they do not have bit number fields); a bit number resulting in the effective address can come only from the AP pointer itself (if it is not indirect) or the fetched indirect word (if it is indirect); no bit numbers are added; the bit number field of an indirect AP pointer is ignored; only a single level of indirection is allowed; and the indirect words referenced are of the 48-bit form shown in Table 2.5.

Like memory-reference class instructions, most generic-AP instructions do not actually make use of a bit number even if a nonzero bit number is inadvertently coded for them. Word referencing instructions continue to reference complete words. The only two instructions in the PRIME 400 which make use of the bit number in their effective address are Effective Address to Field Address Register Zero (001300) and Effective Address to Field Address Register One (001310).

2.5 FIELD MANIPULATION INSTRUCTIONS.

The PRIME 400 is augmented by four new registers for the manipulation

of variable-length bit and character fields. They are Field Address Register Zero (FARØ), Field Length Register Zero (FLRØ), Field Address Register One (FAR1), and Field Length Register One (FLR1). The two Field Address Registers appear to the software to be 40 bits wide each, organized as 2 bits of ring number (FARn)R, 12 bits of segment number (FARn)S, 22 bits of word number (FARn)W, and 4 bits of bit number (FARn)B. In the PRIME 400 implementation, the hardware requires that the upper 6 bits of the word number be zero (segments on the PRIME 400 are limited to 2**16 words). The two Field Length Registers appear to be 27 bits wide, and the hardware presently requires that their upper 6 bits be zero. A further implementation restriction is that Field Address and Length Registers One overlap the Floating Point Accumulator (FAC). Thus the software must be aware that field-manipulation instructions destroy the contents of the Floating Point Accumulator, and conversely floating-point operations destroy the contents of the Field Address and Length Registers.

Sixteen instuctions are provided in the PRIME 400 for the manipulation of the Field Address and Length Registers. In summary, the instructions are:

EAFA Ø	Effective Address to Field Address Register Ø;
EAFA 1	Effective Address to Field Address Register 1;
ALFA Ø	Add Long to Field Address Register Ø;
ALFA 1	Add Long to Field Address Register 1;
LFLI Ø	Load Immediate Field Length Register Ø;
LFLI 1	Load Immediate Field Length Register 1;
TLFL Ø	Transfer Long to Field Length Register Ø;
TLFL 1	Transfer Long to Field Length Register 1;
STFA Ø	Store Field Address Register Ø;
STFA 1	Store Field Address Register 1;
TFLL Ø	Transfer Long from Field Length Register Ø;
TFLL 1	Transfer Long from Field Length Register 1;
LDC Ø	Load Character from Field Ø;
LDC 1	Load Character from Field 1;
STC Ø	Store Character into Field Ø;
STC 1	Store Character into Field 1.

The instructions are described below.

EAFA Ø (GEN-AP ØØ13ØØ) Effective Address to Field Address Register Ø.

(FARØ)R,S,W,B <- (EA)R,S,W,B.

The complete effective address of the instruction, including the bit number portion, is placed in Field Address Register Ø. Field Length Register Ø is unchanged.

EAFA 1 (GEN-AP ØØ131Ø) Effective Address to Field Address Register 1.

(FAR1)R,S,W,B <- (EA)R,S,W,B;

(FAC) <- undefined.

EAFA 1 is similar to EAFA Ø. Note that the Floating Point Accumulator is overlaid.

ALFA Ø (GEN ØØ13Ø1) Add Long to Field Address Register Ø.

(FARØ)W,B <- (FARØ)W,B + (A,B).

The 32-bit integer (with no hole) in the combined A-,B-register is added to the 26-bit unsigned combined word- and bit-number fields of Field Address Register Ø. All but the low-order 2Ø bits of the sum must be zero. The low-order 26 bits of the sum replace the word- and bit-number fields of Field Address Register Ø. The effect is that of adding a calculated bit offset from the A-,B-register to the address in Field Address Register Ø. Field Length Register Ø is unchanged.

ALFA 1 (GEN 1311) Add Long to Field Address Register 1.

(FAR1)W,B <- (FAR1)W,B + (A,B);
(FAC) <- undefined.

ALFA 1 is similar to ALFA Ø.

LFLI Ø (GEN-2 ØØ1303) Load Immediate Field Length Register Ø.

(FLRØ) <- Ø...||(second word of instruction).

The unsigned 16-bit integer in the second instruction word is loaded into the low-order bits of Field Length Register Ø, the high-order bits of which are cleared. The effect is that of loading a field length with is known at compile-time and is less than 65536. The interpretation of the length value depends upon following instructions (character-type operations treat it as a length in characters; bit-type operations treat it as a length in bits). Field Address Register Ø is unchanged.

LFLI 1 (GEN-2 ØØ1313) Load Immediate Field Length Register 1.

(FLR1) <- Ø...||(second word of instruction);
(FAC) <- undefined.

LFLI 1 is similar to LFLI Ø.

TLFL Ø (GEN ØØ1321) Transfer Long to Field Length Register Ø.

(FLRØ) <- (A,B).

The unsigned integer (with no hole) in the combined A-,B-register is transferred to Field Length Register Ø. The high-order 11 bits of the A-register must be zero in order that the high-order 6 bits of the Field Length Register will be zero. The effect is that of loading a computed length. The 21 bits of the Field Length Register which are allowed to be nonzero permit a length of 2**2Ø to be expressed, which is the maximum needed in the PRIME 4ØØ (the bit-length of the largest allowable segment).

TLFL 1 (GEN ØØ1331) Transfer Long to Field Length Register 1.

(FLR1) <- (A,B);
(FAC) <- undefined.

TLFL 1 is similar to TLFL Ø.

STFA Ø (GEN-AP ØØ132Ø) Store Field Address Register Ø.

[(EA)]R,S,W,B <- (FARØ)R,S,W,B.

The contents of all fields of Field Address Register Ø are stored into memory as a hardware indirect-word pointer. If the bit-number field (FARØ)B is zero in the Field Address Register, only the first two words of the pointer are stored (and its E bit is reset); otherwise, all three words must be stored (E will be set). This instruction together with Effective Address to Field Address Register Ø (EAFA Ø) and Add Long to Field Address Register Ø (ALFA Ø) provide a completely general means of manipulating arbitrary bit addresses in the PRIME 4ØØ. In addition, this instruction permits access to the residual address following a field operation whenever this is meaningful. Finally, it can also be used to access the input field address in an unimplemented-instruction package designed to simulate any desired field operations.

STFA 1 (GEN-AP ØØ133Ø) Store Field Address Register 1.

[(EA)]R,S,W,B <- (FAR1)R,S,W,B.

STFA 1 is similar to STFA Ø.

TFLL Ø (GEN ØØ1323) Transfer Long from Field Length Register Ø.

```
(A,B)  <- (FLRØ).
```

The contents of Field Length Register Ø are transferred to the combined A-,B-register as an unsigned long integer (with no hole). The high-order bits of the A-register are cleared. This instruction permits access to the residual length following a field operation (when meaningful), and also can be used in an unimplemented-instruction package simulating field operations to access the input length.

TFLL 1 (GEN ØØ1333) Transfer Long from Field Length Register 1.

```
(A,B)  <- (FLR1).
```

TFLL 1 is similar to TFLL Ø.

LDC Ø (GEN ØØ1302) Load Character from Field Ø.

```
If (FLRØ)  > Ø then
     (A)Ø1-Ø8 <- Ø...;
     (A)Ø9-16 <- (the  8-bit character addressed by (FARØ)R,S,W,B,
                 ignoring the last three bits of (FARØ)B);
     (FARØ)W,B <- (FARØ)W,B + 8;
     (FLRØ)  <- (FLRØ)  - 1;
     condition codes <- unequal.
Else if (FLRØ)  = Ø then
     (A)  <- Ø...;
     condition codes <- equal;
     (FARØ)  and (FLRØ)  unchanged.
```

If the character field described by Field Address and Length Registers Ø is of nonzero length, its first character is loaded into the A-register right-justified and zero-bit padded, the condition codes are set to "unequal", and Field Address and Length Registers Ø are stepped over the character (the address is advanced by 8 bits and the length is decreased by 1 character). On the other hand, if the character field is of zero length, the A-register is cleared, the condition codes are set to "equal", and the address and length registers are unchanged. As the last three bits of the field address are ignored, fields are always treated as character-aligned by this instruction.

LDC 1 (GEN ØØ1312) Load Character from Field 1.

```
If (FLR1)  > Ø then
     (A)Ø1-Ø8 <- Ø...;
     (A)Ø9-16 <- (the  8-bit character addressed by (FAR1)R,S,W,B,
```

```
                ignoring the last three bits of (FAR1)B);
     (FAR1)W,B <- (FAR1)W,B + 8;
     (FLR1)  <- (FLR1)  - 1;
     condition codes <- unequal;
     (FAC)  <- undefined.
Else if (FLR1)  = Ø then
     (A)  <- Ø...;
     condition codes <- equal;
     (FAR1)  and (FLR1)  unchanged.
```

LDC 1 is similar to LDC Ø.

STC Ø (GEN ØØ1322) Store Character into Field Ø.

```
If (FLRØ)  > Ø then
     (the 8-bit character addressed by (FARØ)R,S,W,B,
      ignoring the last three bits of (FARØ)B)  <- (A)Ø9-16;
     (FARØ)W,B <- (FARØ)W,B + 8;
     (FLRØ)  <- (FLRØ)  - 1;
     condition codes <- unequal.
Else if (FLRØ)  = Ø then
     condition codes <- equal;
     (FARØ)  and (FLRØ)  unchanged.
```

If the character field described by Field Address and Length Registers Ø is of nonzero length, its first character is replaced by the character in the right of the A-register, the condition codes are set to "unequal", and Field Address and Length Registers Ø are stepped over the character (the address is advanced by 8 bits and the length is decreased by 1 character). On the other hand, if the character field is of zero length, the condition codes are set to "equal", and the address and length registers are unchanged. As the last three bits of the field address are ignored, fields are always treated as character-aligned by this instruction.

STC 1 (GEN ØØ1332) Store Character into Field 1.

```
If (FLR1)  > Ø then
     (the 8-bit character addressed by (FAR1)R,S,W,B,
      ignoring the last three bits of (FAR1)B)  <- (A)Ø9-16;
     (FAR1)W,B <- (FAR1)W,B + 8;
     (FLR1)  <- (FLR1)  - 1;
     condition codes <- unequal;
     (FAC)  <- undefined.
Else if (FLR1)  = Ø then
     condition codes <- equal;
     (FAR1)  and (FLR1)  unchanged.
```

STC 1 is similar to STC Ø.

2.6 PROCEDURE CALL.

The PRIME 400 procedure call mechanism permits procedures to call one another, facilitates argument passing, permits ring crossing of the protection mechanism, and permits shared, reentrant, and/or recursive code. In the PRIME 400, procedure call performs the functions of JST, F$AT, and SVC in the PRIME 300. The effective address of the procedure call instruction is an entry control block (ECB). The entry control block contains the information required to set up the keys and base registers, perform argument transfer, and do stack segment managment. Stack segment management includes saving the current procedure base, linkage base, stack base, and keys and also allocating space for dynamic variables. An individual stack frame may not cross a segment boundary. The ECB format is shown in Table 2.7. The stack frame format is shown in Table 2.8.

A stack is a collection of one or more segments in which stack frames are allocated as part of the procedure call mechanism. Frames are allocated and deleted in a strict last-in/first-out order within a single stack. In general, all procedures executing in one ring share the same stack, while procedures executing in different rings use different stacks.

The segment number of the first segment in a stack serves to identify the stack. This segment is called the stack root. The first two words in this segment contain a segment number/word number pointer that addresses the location following the last frame allocated on the stack. The third and fourth word of each segment in a stack contain a pointer to the next segment of the stack, if one has been allocated. When there is not sufficient room to allocate a new frame in the segment pointed to by the free pointer, the extension pointer is used to step to the next segment in the stack. If none has been allocated, a stack overflow fault occurs.

Stack frames are backward threaded only (each frame points to its caller's frame). The state of the caller (return location, stack base register, linkage base register, keys) is saved in the called frame. To perform a call or a return, no reference to the caller's frame is required.

PCL (MR 10 02) Procedure Call.

The procedure call instruction (PCL) is a memory-reference instruction that addresses the entry control block of the procedure being called. The instruction performs the following sequence of operations.

It computes the ring number of the called procedure;

it allocates a stack frame for the called procedure;

it saves the caller's critical state information (program counter,

stack base register, linkage base register, and keys) in the new stack frame;

it loads the critical state for the called procedure;

it evaluates the caller's argument template list, storing a list of final effective addresses in the new stack frame.

The actual order in which these operations take place is determined by the requirement that the instruction be restartable if a fault or interrupt occurs during its execution. To avoid completely restarting the instruction when a fault occurs during argument transfer, the program counter is advanced to the first instruction of the called procedure before the argument list is evaluated. This instruction must be an Argument Transfer (ARGT, GEN 000605), which restarts the argument list evaluation from the point at which it was interrupted. When the transfer is complete, the program counter is stepped to the instruction following the ARGT. The argument transfer process uses the X- and Y-registers and the temporary base register to save control information during the transfer.

The detailed execution of a procedure call is as follows.

Ring number calculation. The ring number of the called procedure depends upon the caller's access privileges to the segment containing the addressed entry control block. No ring change takes place if the caller has READ access. If the caller has GATE access, the ring number is taken from the ring number field in ECB.PB without weakening. In this case, the entry control block must start on a 16-word boundary to ensure that a proper block is being referenced. An access violation occurs if neither of the above cases applies.

Stack frame allocation. The stack root is obtained from the entry control block. If zero, the stack root is fetched from the caller's stack frame. The free pointer is fetched from the first two words of the stack root. If there is sufficient room in the segment pointed to by the free pointer for a frame of the size required by the entry control block, the stack frame starts at the free pointer value, and the free pointer is advanced over the new frame. If there is not sufficient room there for the new frame, the extension pointer in words 2 and 3 of the segment pointed to by the free pointer is examined. If zero, a stack overflow fault is generated. If nonzero, it is taken as a new free pointer, and the process is repeated.

Frame header fill-in. The flag word of the new frame is cleared. The caller's program counter, stack base register, linkage base register, and keys are stored in the frame. The saved program counter includes the caller's ring and segment number. At this point, the saved program counter points following the procedure call instruction. When argument transfer is complete, the pointer will be updated to follow the entire calling sequence.

Called procedure state load. The called procedure's program counter,

linkage base register, and keys are loaded from the entry control block. The stack base register is set to the address of the frame created by the procedure call instruction.

Argument transfer. The procedure call instruction is followed by a sequence of argument transfer templates which define the argument list for the called procedure. Argument transfer templates are described next.

ARGUMENT TRANSFER TEMPLATES.

The list of argument transfer templates following the procedure call instruction is evaluated to generate a list of actual argument pointers in the new frame. The format of each argument transfer template is shown in Table 2.7. Each argument pointer may require one or more templates for its generation. The last template for each argument has its S (store) bit set. The last template for the last argument in the list has its L (last) bit set to terminate the argument transfer.

Each template specifies the calculation of an address by specifying a base register, a word and bit displacement from that register, and an optional indirection. If further offsets or indirections are required to generate the final argument address, the template will not have its store bit set, and the address calculated so far will be placed in the temporary base register (ring, segment, word numbers) and X-register (bit number) for access by the next template.

Each time a template with its store bit set is encountered, the calculated address is stored in the next argument pointer position in the new stack frame. If the address has a zero bit offset, the address is stored in the two-word indirect format (with the E-bit reset, see Table 2.5). Otherwise it is stored in the three-word format (E-bit set). In either case, three words are allocated to each pointer in the argument list.

If the caller's template list generates fewer arguments than are expected by the callee (as specified in the entry control block), argument pointers containing the pointer-fault bit set and all other bits reset (pointer-fault code 100000, "omitted argument") are stored for the missing arguments. On the other hand, if the caller's list generates more arguments than are specified by the callee, the surplus arguments are ignored. If the called procedure attempts to reference an omitted argument, other than to simply pass it on in another call, it will experience a pointer fault. If it passes on an omitted argument in another call, the argument will appear omitted to the newly called procedure.

The calling and the called procedure must agree on whether or not arguments are expected. If no arguments are expected (as specified in the entry control block), the procedure call instruction must not be followed by any argument transfer templates; but if arguments are

expected, a template list must follow the call. If a call intends to omit all expected arguments, it may be followed by an argument transfer template with its last bit set but with its store bit reset. Procedures which specify no arguments in their entry control blocks must not begin with ARGT instructions.

ARGT (GEN 000605) Argument Transfer.

The Argument Transfer operation must be the first executable instruction of any procedure which is defined as accepting arguments in its entry control block. It serves as a holding point for the program counter while argument transfer is taking place into the new frame. The program counter is advanced past it when argument transfer is complete. Procedures which specify zero arguments in their entry control blocks should not begin with ARGT operations.

STEX (GEN 001315) Stack Frame Extend.

The Stack Frame Extend operation is used by a procedure which wishes to obtain additional space in the procedure stack for automatic variables. Such space is automatically deallocated and reclaimed for other uses when the procedure returns, just like the original frame created when the procedure was entered. The combined A-,B-register (with no hole) specifies the desired contiguous size of the extension in words. The size is rounded up to an even number of words by the firmware. The address of the extension is returned as a segment number/word number pointer in the combined A-,B-register. It is possible that the extension may not be contiguous with the initial frame (there may have been insufficient room left in the same segment). Any number of extensions may be made. This instruction as well as Procedure Call can cause a stack overflow fault.

PRTN (GEN 000611) Procedure Return.

The Procedure Return instruction deallocates the stack frame created for the executing procedure and returns to the environment of the procedure that called it. The stack frame is deallocated by storing the current stack base register into the free pointer. The caller's state is restored by loading his program counter, stack base register, linkage base register, and keys from the frame being left. The ring number in the program counter is weakened with the current ring number to allow outward returns but prevent inward returns.

TABLE 2.7.
ENTRY AND ARGUMENT CONTROL FORMATS.

ENTRY CONTROL BLOCK
(addressed by PCL instruction)
(16 words long)

offset	name	usage
+Ø,+1	ECB.PB	Pointer (ring, segment, word number) to the first executable instruction of the called procedure.
+2	ECB.SFSIZE	Stack frame size to create (in words). Must be even.
+3	ECB.ROOTSN	Stack root segment number. If zero, keep same stack.
+4	ECB.ARGDISP	Displacement in new frame of where to build argument list.
+5	ECB.NARGS	Number of arguments expected.
+6,+7	ECB.LB	Pointer (ring, segment, word) to be loaded as called procedure's linkage base (location of called procedure's linkage frame less '4ØØ).
+8	ECB.KEYS	C.p.u. keys desired by called procedure.
+9...+15		Reserved, must be zero.

Entry Control Blocks which are gates must begin on a Ø mod 16 boundary, and must specify a new stack root.

ARGUMENT TRANSFER TEMPLATE
(follows PCL instruction)
(2 words long each)

```
BBBBI-ZZLS------
WWWWWWWWWWWWWWWW
```

1-4: Bit number (BBBB).
5: Indirect (I).
7-8: Base register (ZZ).
9: Last template for this call (L).
1Ø: Last template for this argument, store argument address (S).
17-32: Word number (WWW...W).
6,11-16: Reserved, must be zero.

Operation of each template:

```
(temp)R <- MAX( RING , (base register ZZ)R );
(temp)S <- (base register ZZ)S;
(temp)W,B <- (base register ZZ)W,B + WWW...W,BBBB;
```

where (base register ZZ)B is taken as (X)Ø1-Ø4 when ZZ is 3 (the X-register behaves as a bit offset extension to the temporary base register), otherwise as zero (base registers have no bit fields). A

carry from the bit number addition propagates into the word number field. RING is the ring of execution of the caller.
Next, if I=1 (indirect):
(temp)R <- MAX(RING , [(temp)R,S,W]R);
(temp)S,W,B <- [(temp)R,S,W]S,W,B.
Then, if S=1 (store this argument):
(next argument pointer
in new stack frame) <- (temp)R,S,W,B.
Else if S=Ø (don't store):
(XB)R,S,W <- (temp)R,S,W;
(X) <- (temp)B||Ø....
If L=1 (last) then argument list is done, else go on to next template.

TABLE 2.8.
STACK FORMATS.

STACK SEGMENT HEADER
(4 words long)

offset	usage
+Ø,+1	Free pointer (segment number/word number of available location at which to build next frame). Must be even.
+2,+3	Extension segment pointer (segment number/word number of location at which to build next frame when current segment overflows). If zero, a stack overflow fault occurs when current segment overflows.

STACK FRAME HEADER
(1Ø words long)

offset	usage
+Ø	Flag bits (set to zero when frame is created).
+1	Stack root segment number (for locating free pointer).
+2,+3	Return pointer (segment number/word number of location following call and argument sequence which created this frame).
+4,+5	Caller's saved stack base register.
+6,+7	Caller's saved linkage base register.
+8	Caller's saved keys.
+9	Word number of location following call (or beginning of argument transfer templates).

2.7 DOUBLE-PRECISION INTEGER CHANGES.

When the PRIME 400 is operating in PRIME 300 mode (16S, 32S, 32R, or 64R), its double-precision integers are of the same format and accessible in the same way as those of the PRIME 300. These integers have a so-called "hole" in the middle because bit 17 (the high-order bit of the second word) is required to be zero. This gives them a precision of 30 bits plus a sign, and a range of -2**31 = -2,147,483,648 to 2**31-1 = 2,147,483,647.

When in PRIME 400 mode (64V), the processor has another form of integer available, the so-called "long" integer, with no hole. This form allows a precision of 31 bits plus a sign, and a range of -2**32 = -4,294,967,296 to 2**32-1 = 4,294,967,295. This form is used (in 64V mode) by the LDL, ANL, STL, ERL, ADL, SBL, CLS, MPY, DIV, MPL, and DVL instructions. In 64V mode, MPY multiplies two 16-bit integers to produce a 32-bit "long" product with no hole. DIV divides a 32-bit "long" dividend with no hole by a 16-bit integer. MPL multiplies two 32-bit "long" integers with no holes and produces a 64-bit extended integer product with no holes in the combined A-, B-, EH-, and EL-registers (see Table 2.6). DVL is the inverse of MPL. Some generic instructions are provided to clear and swap quantities in and out of the EH- and EL-registers.

In 64V mode, the double-precision bit of the keys can be set and reset with the DBL, SGL, and OTK instructions, but the bit has no effect on the LDA, STA, ADD, and SUB operations. Thus, DLD, DST, DAD, and DSB are not available in 64V mode. Also, in 64V mode the multiply instructions (MPY and MPL) never overflow because there is always room for the product.

Generic operations which specifically manipulate the double-precision integer format with a hole (PID, PIM, FLOT, INT, and NRM) except for the shifts (LRS and LLS) continue to operate on that format even when executed in 64V mode, but their use is not recommended. Similarly, generic operations which specifically manipulate the long-integer format with no hole (STLC, BL.., ALFA, TLFL, TFLL, INTL, FLTL, LL.., PIMA, PIDA, PIML, PIDL, STEX, ADLL, and TCL) also operate on that format when the processor is not in 64V mode, but are not recommended. The shifts LRS and LLS operate with no hole in 64V mode but with a hole in other than 64V mode. Also note that PIMA and PIML overflow on a precision loss, while PIM ignores a precision loss. FRAC is not provided on the PRIME 400 and causes an unimplemented-instruction fault in any mode.

2.8 DOUBLE-PRECISION FLOATING-POINT CHANGES.

In the PRIME 400 the B-register is no longer a portion of the double-precision floating-point accumulator. The B-register is assigned register file location 2 L (see Table 2.6) and continues to be addressed as memory location 2 by short-form memory-reference

instructions. The double-precision floating-point accumulator is assigned register file locations 12 H, 12 L, 13 H, and 13 L, the first three words of which can be addressed as memory locations 4, 5, and 6 by short-form instructions. The last word of the double-precision floating-point accumulator, 13 L, which contains the least-significant bits of the mantissa, cannot be addressed by short-form instructions.

In order that existing PRIME 300 programs continue to work, the old operation code for Clear B-Register (CRB, GEN 140014) on the PRIME 400 clears both the B-register and also the last word of the double-precision floating-point accumulator. The Clear B-Register instruction is assigned a new operation code (140015), which on the PRIME 400 clears only the B-register (and does not affect the floating accumulator), but which on a PRIME 300 also functions as clearing the B-register. Furthermore, a new operation FDBL is provided (GEN 140016), which on the PRIME 400 clears only the last word of the double-precision floating-point accumulator (and not the B-register), but which on a PRIME 300 also functions as clearing the B-register. New programs should use CRB (140015) to clear the B-register and FDBL (140016) to clear the low-order bits of the floating mantissa.

The accuracy of some floating-point operations on the PRIME 400 has been improved, so their results will not, in general, compare equal to the last bit with those of the PRIME 300.

2.9 CONDITION-CODES AND L-BIT.

The C-bit of the PRIME 300 is joined by an L-bit and two condition-code bits in the PRIME 400. The C-bit continues to have the same meaning as in the PRIME 300. The L-bit and the condition codes are set by any arithmetic or shift operation except IRS, IRX, and DRX. Tables 2.2 and 2.4 show exactly which instructions have an effect on the C-bit, L-bit, and condition codes.

The L-bit is equal to the carry out of the most significant bit of an arithmetic operation. It is valuable for simulating multiple-precision operations and for performing unsigned comparisons (following a CAS or a SUB).

The two condition-code bits are designated "EQ" and "LT". EQ is set if and only if the 16-bit or 32-bit visible portion of the result is zero (that is, if overflow occurs, EQ reflects the state of the result after truncation rather than before). LT reflects the extended sign of the result (before truncation, if overflow), and is set if the result is negative.

Many instructions have been added to the PRIME 400 set to test and branch on the L-bit and the condition codes. The condition codes appear in the PRIME 400 keys but not in the PRIME 300 keys (see next section).

2.10 KEYS AND MODALS.

The notion of keys on the PRIME 300 has been considerably expanded in the PRIME 400. Status associated with a process (as opposed to the processor) is collected in a 16-bit register known as the "keys". Status associated with the processor (and not with any particular process) is collected in another 16-bit register known as the "modals". The formats of the keys and the modals are shown in Table 2.9.

Note that the C-bit, double-precision bit, and addressing-mode bits are still in their PRIME 300 positions in the keys, but the shift count has been removed to make room for other things. For compatibility with the PRIME 300, the INK and OTK instructions still read and set the shift count (low-order 8 bits of the floating-point accumulator exponent) instead of accessing bits 9-16 of the PRIME 400 keys register. For manipulating the PRIME 400 keys, the processor is provided with the LPSW, TKA, and TAK instructions.

The double-precision bit in the PRIME 400 may be set and reset by DBL and SGL (as well as all other keys operations), but in 64V mode this bit has no effect on the LDA, STA, ADD, and SUB operations. In other than 64V mode, double-precision works as on the PRIME 300.

The L-bit and the condition-code bits have been described in the preceding section.

The addressing-mode bits have been expanded to three to allow the addition of the new 64V mode (bits 4-6 = 110).

The floating-exception and integer-exception bits allow better program control over when arithmetic exceptions cause faults. When keys bit 7 is a zero, floating-point arithmetic exceptions cause the processor to fault; when bit 7 is a one, floating-point arithmetic exceptions merely set the C-bit. The PRIME 400 also allows faults on integer arithmetic exceptions. When keys bit 8 is a one, any integer arithmetic operation which sets the C-bit on overflow (codes 2 and 3 in the C-bit column of Tables 2.2 and 2.4) also causes an integer exception fault; when bit 8 is a zero, no fault occurs. Note that for bit 7 it is a zero which requests the fault and for bit 8 it is a one. Integer exception faults share the same vector assignment as floating-point exception faults. The integer exception fault code is '001400 (hexadecimal 0300).

The in-dispatcher and save-done bits are managed by the process-exchange mechanism exclusively. Software should not attempt to alter their state.

The modals now make visible the processor state. As before, the enable bit and the vectored-interrupt bit of the modals can be manipulated with the specific instructions provided for that purpose (ENB, INH, ESIM, EVIM). But now all bits of the modals can be read or written with the LDLR and LPSW instructions.

Bits 9-11 of the modals reflect which register set the processor is using at the moment. The PRIME 400 has two user register sets, designated 2 and 3. The process-exchange mechanism manages the switching of register sets automatically as it switches processes. Under no circumstances should software ever attempt to change the settings of these bits. See the discussion of the LPSW instruction in Section 2.14.

Bit 12 of the modals controls mapped I/O. When this bit is a zero, all I/O addresses are physical addresses. When it is a one, all I/O addresses are mapped by the segmentation and paging hardware as if they were in segment zero. It is the responsibility of the operating system to see that the desired virtual-to-physical mapping remains in effect for the duration of the I/O transfer.

Bit 13 of the modals set to one turns on the PRIME 400 automatic process-exchange mechanism. This mechanism is discussed in the next section.

Bit 14 of the modals set to one turns on segmentation and paging mapping. When this bit is a zero, the processor continues to develop full 28-bit virtual addresses in its usual way, but the low-order 22 bits of them (6 bits of the segment number and 16 bits of the word number) are used directly without translation to address physical memory. Thus, when this bit is a zero, the machine behaves as if its address space was 64 contiguous segments of 65536 words each, mapped one-to-one onto physical memory.

Bits 15 and 16 of the modals control the response of the processor to detected integrity failures. The four machine-check modes are described in Section 1.7.

The keys and modals registers have a special hardware implementation on the PRIME 400 alongside the register file. Software must never attempt to write into the keys or the modals with the STLR instruction. The only valid way to change either the keys or the modals is to use the LPSW instruction (see Section 2.14), the keys operations OTK and TAK, or the various special-case instructions designed to manipulate specific bits of the status. Furthermore, even LPSW should not be used to alter the state of the in-dispatcher and save-done bits of the keys or the register-set bits of the modals. These bits are managed by the hardware and firmware exclusively.

TABLE 2.9.
KEYS AND MODALS.

PRIME 400 KEYS (16 bits)

CDLMMMFXNZ----IS

1:	C-bit. (C)
2:	Double-precision bit (SGL, DBL). (D)
3:	L-bit (see Section 2.9). (L)
4-6:	Addressing mode; 0=16S, 1=32S, 2=64R, 3=32R, 6=64V. (MMM)
7:	Allow floating-point exception faults if zero. (F)
8:	Allow integer exception faults if one. (X)
9:	Condition code LT (negative result). (N)
10:	Condition code EQ (zero result). (Z)
11-14:	Reserved, must be zero.
15:	In dispatcher (set/reset only by process exchange). (I)
16:	Save done (set/reset only by process exchange). (S)

These keys are found in register file cell 24 H of the current register set, and are referenced by the LPSW, TKA, and TAK instructions.

INK AND OTK KEYS (16 bits)

CDLMMMFXSSSSSSSS

1-8:	Same as in PRIME 400 keys above. (CDLMMMFX)
9-16:	Shift count (low-order 8 bits of the floating-point accumulator exponent register, register file 13 H). (SSS...S)

These keys are referenced by the INK and OTK instructions.

MODALS (16 bits)

EV------CCCIPSMM

1:	Interrupts enabled. (E)
2:	Vectored-interrupt mode. (V)
3-8:	Reserved, must be zero.
9-11:	Current register set (set/reset only by process exchange). (CCC)
12:	Mapped I/O mode. (I)
13:	Process-exchange mode. (P)
14:	Segmentation mode. (S)
15-16:	Machine-check mode. (MM)

2.11 PROCESS EXCHANGE.

A process is a logically continuously executing sequence of code. Physically a process may be halted for indeterminant lengths of time, either by an interrupt or by explicitly requesting suspension until a specific event occurs.

The data bases included in the process exchange mechanism are process control blocks (PCBs), the ready list, semaphores (in the sense of Dijkstra), and wait lists. Each process must have a process control block describing the process. All PCBs in the system are in a single dedicated segment. The minimum size for a PCB is 64 words. The maximum number of separate processes is 1023. Table 2.10 gives the PCB format. Movement between the ready list and the wait lists is controlled by use of the NOTIFY and WAIT instructions. These instructions reference a semaphore. After executing either instruction, the highest priority process on the ready list is executed. A NOTIFY (Dijkstra's "V" operation) decrements the semaphore counter and a WAIT (Dijkstra's "P" operation) increments the counter. Thus a NOTIFY may cause a process to move from non-ready to ready and a WAIT may cause a process to move from ready to non-ready.

A process is considered either ready to execute or not ready. If ready, the process is on the ready list. If not ready, the process is on the wait list of some semaphore. Coordination between processes takes place through semaphores. A semaphore defines an event whose meaning is shared among two or more processes. A semaphore takes two 16-bit locations in memory: a counter of WAITs on the event and the location of the first PCB awaiting the event. Negative counts indicate the event has already happened.

A wait list consists of a semaphore plus the PCBs of any processes awaiting its event.

The ready list is a logically two-dimensional structure consisting of strings of PCBs of processes which are ready to execute. Each PCB contains a level indicator giving the priority of the process. Multiple processes can exist on the same priority level. Processes within a level are strung with the PCB link word.

The Process Exchange mechanism is composed of three data bases and two basic instruction primitives. The data bases are the ready list, wait lists, and Process Control Blocks (PCB). The basic instruction primitives are WAIT and NOTIFY. In addition, there is an independent mechanism for controlling the usage of two register sets which is related to, but not part of, the ready list data base.

The ready list is a two-dimensional list structure used for priority scheduling and dispatching of processes. The entire ready list data base (excluding live registers) and all PCB's are contained in a single segment. The segment number of this segment is contained in a 16-bit register called OWNERH. Within the segment, all pointers and addresses (except fault vectors and wait list pointers) are 16-bit

word number quantities.

The two-dimensionality of the ready list is achieved with a linear array of list headers for each priority level composed of a Beginning of List (BOL) pointer and an End of List (EOL) pointer.

Each pointer is the 16-bit word number address of a PCB (in the same segment as the ready list). The PCB's on each priority level list are forward-threaded through a 16-bit link word, and as many PCB's as desired can be threaded together on each priority level to form the ready list. A process' priority level is both determined by and encoded as the address of a BOL pointer in the ready list. Priority order is determined by arithmetic comparison, i.e., smaller numbers (addresses) are higher priorities. As a result, priority level list headers must be allocated in contiguous memory at system startup time.

The end of the ready list is determined by a BOL containing a 1 (PCB addresses must be even). An empty level is indicated by a BOL containing Ø. The 32-bit registers PPA (Pointer to Process A) and PPB (Pointer to Process B) are a speed-up mechanism for locating the next process to dispatch. PPA always contains both the level (BOL pointer) and PCB address (designated level A and PCBA) of the currently active process. PPB points to the NEXT process to be run when process A 'goes away'. PPA not only points to the currently active process, but, by definition, level A is the highest level in the system. It is possible for PPB and PPA to be 'invalid'. This condition is indicated by a PCB address of Ø. It is important NOT to disturb the level portions, especially level A since, even if invalid, level A indicates the highest level that WAS in the system and therefore determines where in the ready list to begin a scan, if necessary (PPB invalid), for the next process to run. In a completely idle system, both PPA and PPB will be invalid and, upon completion of the ready list scan, the u-code will go into a 'wait for interrupt' loop.

It is important to notice that there is no word number pointer to the first priority level in the ready list. The ready list allocator, which starts the process exchange mechanism, knows where the list begins and, during execution, level A (in PPA) will always point to either the highest level currently in the system or the last known highest level and, hence, acts as an effective ready list begin pointer. In addition, level B will always be higher than the second level to run. That is, a PCB can never be on a level higher than level B unless it is the only PCB higher than level B (i.e., level A).

Two 'queuing' algorithms are implemented for the ready list, FIFO and LIFO.

Every PCB in the system will always be somewhere. If it is not on the ready list, then, by definition, it will be on a wait list. A wait list is defined by a 32-bit semaphore consisting of a 16-bit counter (C) and a 16-bit word number BOL pointer. Since the ready list and all PCB's reside in one segment (OWNERH), and only PCB's go onto wait lists, a segment number is not needed in the semaphore. However,

semaphores themselves can be anywhere and, in general, are NOT in the PCB segment. Notice that the last block on the wait list contains a Ø link word. Notice also that the semaphore contains only a BOL pointer.

The 'queuing' algorithm for wait lists is process priority queuing. That is, the priority level of a PCB will determine where on the wait list the PCB will be queued. For PCB's of equal priority, the algorithm becomes FIFO.

The contents of a process control block (PCB) are shown in Table 2.10. The PCB can be broken into the following logical sections which are ordered as shown:

a. Control

Ø - level (pointer to BOL in ready list)
1 - link (pointer to next PCB or Ø)
2,3 - SN/WN of Wait List this block is currently on (SN=Ø indicates on ready list)
4 - abort flags used to generate Process Fault when PCB is dispatched.
Current bit assignments 1-15: MBZ
16: process interval timer overflow
5,7 - reserved

b. Process State

8,9 - Process elapsed timers (must be maintained by software that resets the live interval timer)
10,13 - DTAR2 and DTAR3 (never saved, always restored)
14 - Process Interval Timer with 1.Ø24 msec resolution
15 - Reserved
16 - Save mask - used to avoid saving and restoring registers = Ø
Bits 1- 8: GRØ-GR7 (2 words each)
9-12: FPØ-FP1 (4 registers, 2 words each)
13-16: Base Registers(PB,SB,LB,XB)
17 - Keys
18,33 - GRØ-GR7
34,41 - FPØ-FP1
42,49 - Base Registers (PB,SB,LB,XB)

Note that although all the registers are assigned locations within the PCB, only non-zero registers will actually be saved which results in a compacted list which can only be determined by the bits in the save mask. In general, the saved registers (those not equal to Ø) will be between words 18 and 49. The order of the registers, however, is fixed as above.

c. Fault (See section on Faults for a description of the use of this portion of the PCB)

5Ø,59 - Fault Vectors: SN/WN pointers to fault tables for

Ring Ø, Ring 1, Page Fault and Ring 3 fault handlers

60,62 - Concealed Fault Stack Header

63,.. - Concealed Stack - 6 word entries. (This stack need not start at word 63).

There are two basic instruction primitives for the process exchange mechanism: NOTIFY and WAIT. In addition, NOTIFY has two variants. These instructions, similar to Dijkstra's P and V operators, are essentially 'interlock' mechanisms. These instructions are three-word (48-bit) 'instructions' as follows:

Instruction (16-bit universal generic)
32-bit AP-pointer to semaphore address

As suggested by the names, WAIT is used to wait for an event (CP, time, unit record device available, whatever) and NOTIFY is used to signal that an event has occurred. In particular, a WAIT is used to wait for a NOTIFY and a NOTIFY is used to alert a process which is waiting.

Coordination is achieved by means of a semaphore containing a counter and a BOL pointer. The semaphore and the PCB's waiting for the event of that semaphore constitute a wait list. The counter, if greater than Ø, indicates the number of PCB's on the wait list. If negative, it indicates the number of processes that can obtain the resource. Semaphores fall into two categories: public and private. A public semaphore is used to coordinate several processes together or control a system resource. Private semaphores are used by a single process to coordinate its own activities. For example, if a disk request is made, a process will wait on a private semaphore for the disk operation to complete. The disk process will then notify the semaphore upon completion. The distinguishing characteristics of a private semaphore is that only 1 PCB can ever be on that wait list. A public semaphore can have many different PCB's on its wait list.

The operation of WAIT is as follows: the semaphore counter is incremented and, if greater than Ø, (resource not available/event has not occurred), the PCB is removed from the ready list and added to the specified wait list. If the counter is less than or equal to Ø, the process continues. If the PCB is put on the wait list, the general registers are NOT saved and the register set is made available. Therefore, a process can NEVER depend on the general registers being intact after a WAIT. In fact, from the point of view of an executing process, a WAIT appears as a NOP which destroys the registers. In addition, WAIT will turn off the process timer.

The NOTIFY instruction has two flavors:

NFYE: use FIFO queuing op code Bit 16 = Ø
NFYB: use LIFO queuing op code Bit 16 = 1

The instructions differ ONLY in the ready list queuing algorithm used. The operation of NOTIFY is as follows: the semaphore counter is

decremented and the notifying process continues. If the counter is less than Ø, no action is taken, but if greater than or equal to Ø, a PCB is removed from the top of the wait list and added to the ready list. No explicit action is ever taken on the notifying process, only the notified semaphore. If a notified process is of higher priority than the notifying process, the latter will be effectively 'interrupted', but will remain on the ready list.

The dispatcher is the root of the process exchange mechanism and is responsible for determining the next process to run (be dispatched), and assigning that process a register set. There is considerable overlap with NOTIFY and WAIT in functionality related to maintaining the ready list. For example, both NOTIFY and WAIT update PPA and PPB as appropriate, but the dispatcher scans the ready list if PPA is invalid. Register file management, including any necessary saves and restores, are the sole province of the dispatcher.

Upon entry, the dispatcher first asks if PPA is valid (PCBA nonzero). If it is, the process is assigned a register set and dispatched. If PPA is not valid (PCBA zero), a scan of the ready list is initiated from the level of PPA, which is always valid. If a PCB is found, PPA is adjusted and the process dispatched. If the ready list is empty, the dispatcher idles. Whenever a process is dispatched the process timer is turned on.

In each register set, a register, designated OWNER, contains a pointer to the PCB of the process which owns the set. OWNER is a full 32-bit pointer and OWNERH is used throughout the system to determine the segment number of the ready list and PCB's. Obviously, OWNERH must be the same in both register sets. In addition, the low order bit of the keys register (KEYSH) is used to indicate whether the register set is available. The bit is called the Save Done bit and, if set, indicates that the register set and its copy in the owner's PCB are identical (a save has been done). This bit is set by the save routine (called from WAIT or the dispatcher) and reset when a process is dispatched. Whether a register set is available (SD=1) or not, it is always owned. Therefore, if a process goes away (either as a result of a WAIT or the notification of a higher level process) and comes back again immediately and, if that process still owns the register set, a restore operation is not necessary. If a register set switch is necessary, the process timer is turned off. The dispatcher is the only code which switches register sets.

The PRIME 400 contains four distinct register files. Each file is further divided into halves, each 32 locations (registers) long, and each 16 bits wide. One half is referred to as the high half and the other as the low half. Since both halves are addressed together, each register file contains 32 32-bit registers or 64 16-bit registers. The register files, numbered from Ø, are used as follows:

RFØ - u-code scratch and system registers
RF1 - 32 DMA channels
RF2 - User register set

RF3 - User register set

This layout of register files allows easy expansion to eight register files, thus adding four new user register sets. All user register sets have the same internal format and the DMA register file simply consists of 32 channel registers. Channel register ´20 within RF1 is equivalent to the PRIME 300 DMA registers ´20 and ´21. Channel register ´22 is mapped to ´22 and ´23. In this way, the mapping proceeds for each even register in RF1 to channel register ´36, mapped to ´36 and ´37. All other RF1 registers represent additional DMA channels over the PRIME 300. Table 2.6 shows the internal layout of the user register sets (RF2, RF3). Note that all user register sets contain the segment number of the Ready List/PCB segment (OWNERH) and a cell for the modals (KEYSL). It is necessary, before entering process exchange mode, to set OWNERH in ALL register sets to the proper value and to NEVER alter it thereafter. Although all register sets contain a cell for the modals, only the current register set (CRS) contains the valid modals. It is therefore necessary, whenever register sets are switched, to copy the modals into the new register set. Currently, only the Dispatcher switches register sets. CRS is defined and specified by the three bit field labeled ´CRS´ in the modals. Since this field can span up to eight register files, but two are used for u-code scratch and DMA, user register sets are numbered from 2 - 7. Of course, only 2 and 3 are currently implemented. Thus, for the PRIME 400, the CRS field must always have bit 9 off, bit 10 on, and bit 11 selects the register set (as if 0 and 1 were the numbers). In fact, the u-code will only look at bit 11.

Direct register file addressing (not using CRS) is accomplished either with the LDLR/STLR instructions or via the control panel. The Register Files are ordered sequentially with an absolute address of 0 addressing RF0-register 0 (u-code scratch/system file), ´40 addressing RF1-register 0 (DMA file), ´100 addressing RF2-register 0 (user set 2), and ´140 addressing RF3-register 0 (user set 3).

Cell 30 H of the current register set is a 16-bit wide 1024-microsecond up-counting process c.p.u. timer. The dispatcher turns it on before dispatching a process and turns it off before saving a process into its PCB or swapping register sets. On each tick, u-code increments the live interval timer (TIMER) in RF(CRS). When that overflows, bit 16 in the PCB abort flags is set to cause a process fault. It is the responsibility of software that resets the interval timer to maintain the elapsed timer.

At various points in the dispatcher a check for interrupt pending (fetch cycle trap) is made. As a result, interrupts can occur either in the fetch cycle or in the dispatcher. The possible Fetch Cycle traps are:

1. External interrupt and memory increment.
2. CP-timer increment and possible overflow.
3. Power failure.
4. Halt switch on control panel.

5. End-of-instruction trap.

The end-of-instruction trap occurs either from an ECC corrected error or from a missing memory module, memory parity, or machine check during I/O. In all cases, if the check handling software returns (via LPSW instruction), the possible destinations are either the fetch cycle or the dispatcher (PB in PSW not a real program counter). In order to guarantee the proper destination, bit 15 of the keys (KEYSH) is used to indicate if the trap was detected by the dispatcher (bit 15=1). This bit is set by the dispatcher upon detecting a trap and is reset when a process is actually dispatched (return to fetch cycle).

TABLE 2.10.
PROCESS CONTROL BLOCK FORMAT.

octal offset	field length (16-bit words)	field description
0	1	Level (priority).
1	1	Link to next PCB of same priority.
2	1	Wait-list segment number (zero if ready).
3	1	Wait-list word number.
4	1	Abort flags.
5	3	Reserved.
10	2	Elapsed timer.
12	2	Descriptor table address register 2.
14	2	Descriptor table address register 3.
16	1	Interval timer (live).
17	1	Reserved.
20	1	Save mask.
21	1	Keys.
22	2	General register 0.
24	2	General register 1.
26	2	General register 2.
30	2	General register 3.
32	2	General register 4.
34	2	General register 5.
36	2	General register 6.
40	2	General register 7.
42	4	Floating-point register 0.
46	4	Floating-point register 1.
52	2	Procedure base register.
54	2	Stack base register.
56	2	Linkage base register.
60	2	Temporary base register.
62	2	Fault vector, ring 0.
64	2	Fault vector, ring 1.
66	2	Reserved.
70	2	Fault vector, ring 3.
72	2	Page fault vector.
74	1	Concealed stack FIRST.
75	1	Concealed stack NEXT.
76	1	Concealed stack LAST.
77-up		Concealed fault stack entries, six words per entry (see Section 2.12).

The data saved in locations 22 through 61 has a fixed order, but is compacted toward low addresses over doublewords of zero. The save mask in location 20 has a zero bit for each doubleword of zero omitted and a one bit for each nonzero doubleword stored. Locations are fixed again starting at 62.

2.12 TRAPS, INTERRUPTS, FAULTS, AND CHECKS.

Four words used frequently are 'trap', 'interrupt' (or 'external interrupt'), 'fault', and 'check'. The meanings of these terms are carefully distinguished for the PRIME 400. Software breaks in execution are divided into three main categories referred to as 'interrupts', 'faults', and 'checks'. The word 'trap', on the other hand, refers to a break in execution flow on the u-code level.

Traps can occur for many reasons, not all of which cause software visible action, and are always processed on the u-code level. Some traps may directly or indirectly cause breaks in software execution, but not all software breaks are the result of a trap.

On the PRIME 300 and in the PRIME 400 when process-exchange mode is not turned on, interrupts, faults, and checks used the same protocol to get to their respective software handlers, namely they caused a vector through a dedicated sector 0 location (JST* vector). On the PRIME 400 when process exchange mode is enabled, the three categories use different protocols both from the PRIME 300 and each other. Roughly, the three terms are used to describe:

1. Interrupt - a signal has been received from a device in the external world (including clocks) indicating that the device either needs to be serviced or has completed an operation. In general, an interrupt is not the result of an operation initiated by the currently executing software and will not be processed by that software (though, of course, it may).

2. Fault - a condition has been detected that requires software intervention as a direct result of the currently executing software. In general, faults can be handled by the current software, though in many cases common supervisor code within the current process handles the fault. Also, in general, an external device in the real world is not directly involved in either the cause or cure of a fault condition. Often, however, external devices are involved indirectly as, for example, in performing a page turn operation as a result of a page fault.

3. Check - an internal CP consistency problem has been detected which requires software intervention. The condition could be either an integrity violation, reference to a memory module which does not exist, or a power failure. By contrast, a reference to a page which is not resident or an arithmetic operation which causes an exception is a FAULT condition.

External Interrupts.

External Interrupts operate in either of two modes depending upon whether process exchange is turned on. If process exchange is off, all interrupts are treated as PRIME 300 interrupts. In all cases, except memory increment, the address presented by the controller (or '63 if in standard interrupt mode) is used as the address in segment 0 of a 16-bit vector. This vector, in turn, points to interrupt response code (IRC), also in segment 0, which is entered via a simulated JST* through the vector. Thus, the current program counter (RPL) is stored in (vector) and execution begins at location (vector) +1 with interrupts inhibited, but with no other keys or modals changed. If in vectored interrupt mode, it is the responsibility of the software to do a CAI. In either mode, the full RP is saved in the register PSWPB. Software must store PSWPB before allowing another interrupt.

If process exchange mode is on, an entirely different mechanism operates. In all cases, except memory increment, the address presented by the controller is used as a 16-bit word number offset into the interrupt segment (#4). This segment is guaranteed to be in memory, but STLB misses may occur. The current PB (actually RP) and KEYS (keys and modals) are saved in the u-code scratch registers PSWPB and PSWKEYS. The machine is then inhibited and the IRC begins execution in 64V mode. It is the responsibility of the IRC to issue a CAI. It is important to note that the IRC in the interrupt segment does not belong to any process. PPA points to the PCB of the interrupted process and, in fact, no PCB exists for the IRC. Also, except for PB and KEYS, no registers are saved. In fact, even PSWPB and PSWKEYS are in the register file and not in memory. As a result, the IRC cannot do an enable and must return to the process exchange mechanism (i.e., the dispatcher) as soon as possible. Because of all these restrictions on what the immediate IRC can do, as well as the fact that it does not belong to any process, it is referred to as phantom interrupt code. Unless the job of servicing an interrupt is very simple, phantom interrupt code can do little more than turn off the controller's interrupt mask, issue a CAI, and NOTIFY the real IRC.

A memory increment interrupt is handled the same regardless of the state of process exchange. The address presented by the controller is used as the 16-bit word number in segment 0 (I/O segment) of a 16-bit memory cell to be incremented. If the counter does not overflow (-1->0), the u-code simply returns. With process exchange off, the return is always to the fetch cycle. With process exchange on, the return is either to the fetch cycle or the dispatcher, depending upon where the interrupt was detected. When detecting an interrupt, the dispatcher always insures that RP=PB and that all live keys=KEYS. When memory increment returns, it does so to the top of the dispatcher without having touched PB or KEYS. In this way, memory increment is guaranteed not to destroy any vital information needed by the dispatcher. If the memory cell counter does overflow, an End-of-Range signal is generated and then memory increment returns. The subsequent EOR interrupt will then be treated like any other external interrupt.

Phantom interrupt code has two options for the actions it can take. If the servicing required by the interrupt is very simple, phantom code can completely process the interrupt and return to the dispatcher. If the servicing required is more complex, the phantom code must turn off the controller's interrupt mask and NOTIFY the remainder of the IRC. In the first case, PB and KEYS must be restored from PSWPB and PSWKEYS and then the dispatcher must be entered directly. Since PB cannot be restored in phantom code (the program counter will point to the instruction in phantom code) and the dispatcher cannot be entered directly (no such instruction exists), the special instruction, IRTN, a 16-bit generic, is executed to perform these functions. After entering the dispatcher via an IRTN, the dispatcher does not know that an interrupt occurred.

In order to NOTIFY a process, phantom code must insure that PB and KEYS are restored before issuing the NOTIFY. The special instruction, INOTIFY, performs the restore and then does the NOTIFY. As NOTIFY, INOTIFY is a three-word generic with two flavors, INOTIFYB and INOTIFYE where the beginning of list option has bit 16=1 and the end of list option has bit 16=Ø in the opcode.

Phantom Interrupt code can issue a CAI in one of two ways. Either an explicit CAI instruction may be issued or the IRTN/INOTIFY instructions can issue it. Bit 15 of the IRTN/INOTIFY instructions is interpreted as follows:

Bit 15 = Ø do not issue CAI
1 issue CAI

In all, there are four INOTIFY instructions as follows:

Name	Bit 15	16	Function
INEC	1	Ø	End + CAI
INEN	Ø	Ø	End + no CAI
INBC	1	1	Beginning + CAI
INBN	Ø	1	Beginning + no CAI

Faults.

Faults are CPU events which are synchronous with and, in a loose sense, caused by software. Eleven fault classes have been defined for the PRIME 400. Several of these classes are further subdivided into distinct types. Of the eleven, three are completely new for the PRIME 400 and, of the other eight, three have expanded meaning when in PRIME 400 mode. The eleven fault classes and their meanings are:

Fault	PRIME 400	PRIME 300
RXM	Restrict mode violation	same
Process	Abort flags word .NE. 0 in PCB on dispatch	N.A.
Page	Page Fault (Page not in memory)	same
SVC	N.A.	Supervisor Call
UII	Unimplemented instruction	same
ILL	Illegal instruction	same
Access	Violation of segment access rights	Page write violation
Arithmetic	All FLEX + IEX (Integer Exception)	FLEX
Stack	Stack overflow/underflow	Procedure Stack(S-Reg) Underflow
Segment	1: Segment # too big	N.A.
	2: Missing segment (SDW fault bit set)	N.A.
Pointer	Fault bit in pointer set	N.A.

The fault handling mechanism consists of two data bases and the CALF instruction. The u-code is in turn divided into a set of 'front-ends' for each fault class and a common fault handler.

The fault data bases consist of the fault vectors and concealed stack in the PCB and the fault tables pointed to by the PCB vectors. Table 2.11 shows these data bases as well as the mapping of PRIME 300 faults to PRIME 400 faults. Also shown in this figure is the differential action taken according to fault class (e.g., what ring to process the fault in) and the set up the u-code 'front end' must do before going to the common fault handler.

The underlying philosophy of the four fault vectors is that while some faults may need to be processed by ring 0 code, others may be adequately handled in the current ring of the faulting process. The vectors are in the PCB to allow different processes to have different fault handlers. For example, process A may wish to use a system fault routine to handle pointer faults (dynamic linker) while process B may wish to define its own algorithms for resolving pointer faults. Notice that it is always possible for a 'current ring' fault handler to call a ring 0 procedure if the need arises. Note also that page fault has its own vector despite the fact that ring 0 is entered. For the special case of page fault, only a single, system-wide processor will be used and all PCB page fault vectors will point to the same place.

The concealed stack, also in the PCB, is used to allow fault on fault conditions. For example, it is quite possible to get a segment fault while processing a segment fault. The only fault which cannot cause another fault of any type is page fault. Each frame of the concealed stack contains the PB and keys (KEYSH) of the faulting procedure as

well as a fault code (to distinguish different types within each class) and a fault address, if appropriate. The stack itself is circular and must have allocated sufficient frames to handle the longest possible sequence of fault on fault that can occur in ring Ø. Such a sequence might be: Pointer (link) fault -> Segment fault -> Stack fault -> Segment fault -> Page fault. Note that this particular sequence occurs before any software fault handler is entered. Also, the first segment fault enters ring Ø, so at least a five-level stack is necessary if the original link fault is to be processed correctly. Each frame of the concealed stack is six words long, organized as follows:

+Ø,+1 Program counter (segment number/word number);
+2 Keys;
+3 Fault code (FCODE in Table 2.11);
+4,+5 Fault address (segment number/word number, FADDR in Table 2.11).

The second data base consists of four distinct fault tables, each pointed to by a PCB fault vector. Each entry in the table consists of four words of which the first three must be a CALF instruction. Only the page fault table must be locked to memory and only the ring Ø table must be in a pre-defined (SDW exists) segment (otherwise, segment fault might recurse infinitely). Naturally, the ring Ø table, as well as the PCB, is carefully audited by ring Ø procedures.

The CALF instruction has two major functions. First, to avoid holding off interrupts for too long, the CALF instruction defines a restart point in fault handling since it has a PB (i.e., it is a macro-machine instruction). As a result, it is quite possible to suspend a process in the middle of getting to a software fault handler. Second, it allows a straightforward mechanism to simulate a procedure call from the faulting procedure (at the instruction causing the fault) to the fault handler.

The instruction itself is a three-word generic in which the second and third words are a 32-bit AP-pointer to the fault handler. To simulate the procedure call, the PB and KEYS from the concealed stack are placed in the fault handler's stack frame along with the other base registers (only the PB and KEYS have been changed to point to the CALF and to enter 64V addressing mode) to be used by the standard procedure return (PRTN) instruction. In addition, the fault code and address are placed in the fault handler's stack as words '12, '13, and '14. After the information is moved from the concealed stack it is popped. The flag word ('Ø) of the new frame is set to 1 instead of Ø to distinguish the frame as created by CALF. The entry control block addressed by the CALF must specify no arguments. It may be a gate or not.

The fault handler is a u-code routine that is entered from the various fault class 'front ends' and, based on process exchange mode, either simulates a PRIME 3ØØ type fault (JST* through segment Ø fault vectors) or performs the PRIME 4ØØ fault protocol which includes setting up a concealed stack frame, switching to 64V mode, and determining, on the basis of information provided by the 'front end', which fault vector to

use and setting PB to point to the proper CALF in the fault table. Note that for PRIME 300 faults, the full RP is also saved in the u-code scratch register PSWPB and the machine is inhibited for one instruction if in Ring 0.

Checks.

Checks, unlike faults, are CPU events which are asynchronous with, and are not caused by, normal instruction execution. Rather, they are events which are either invisible (e.g., an ECC corrected error) or fatal (e.g., missing memory module) to the currently executing procedure and perhaps the CPU entirely (e.g., machine check). Checks essentially represent processor faults as opposed to process or procedure faults. Four check classes have been defined as follows:

CHeck	header loc	First instruction of handler	DSW set?
power failure	4/'200	4/'204	no
memory parity	4/'270	4/'274	yes
machine check	4/'300	4/'304	yes
missing memory module	4/'310	4/'314	yes

Unlike faults which can be stacked and interrupts which cause a process to be suspended, each check class has a single save area (check block) consisting of eight words in the interrupt segment (#4) in which PB and KEYS (high and low) are saved in the first four locations (check header) and the remaining four locations contain software code (probably a JMP). In addition to the memory data base, three 32-bit registers are used as a diagnostic status word (DSW) to help a software check handler sort out what happened. Table 2.12 shows the format of the DSW.

Check reporting (traps) is controlled by the two low order bits in the modals (KEYSL). The possible modes are:

MCM = 0 no reporting
1 report memory parity (uncorrected) only
2 report unrecovered errors only
3 report all errors

The check trap can result in two possible actions depending upon the type of check that occurred and the type of u-code which was trapped. If the trapped code was either DMX, PIO, or external interrupt processing (unless the error was a machine check for RCM parity), or if the check was for an ECC corrected (ECCC) error, the end-of-instruction flag is set, REOIV is set to the proper offset/vector, MCM is set to 0 (except ECCC which sets it to 2), and a u-code RTN to the trapped step is executed. In this way, the IO bus is always left in a clean state. In all other cases, the check to software occurs immediately.

The common check handler is entered from various check 'front ends' and, based on process exchange mode, either simulates a PRIME 300 type check (JST* through segment 0 check vectors) or performs the PRIME 400 check protocol which includes setting up the check header, inhibiting the machine, and switching to 64V addressing mode. In either mode, MCM is set to 0 before going to software.

Check-handling software has the responsibility for clearing the Diagnostic Status Word after each check. If the software does not clear the DSW, later checks will overwrite some of the data from preceding checks. Enough independent fields are allowed in the DSW to remember one each of the longest chain of checks which can occur before software gets control, except that the RMA and PB of the last check only can be saved. If a missing-memory-module check has occurred, then it was the last, and the saved RMA and PB go with it. If not, then if either a machine check or an ECC-uncorrected memory-parity check occurred (these are mutually exclusive), then it was the last and its RMA and PB are in the DSW. Otherwise, the saved RMA and PB belong to the ECC-corrected memory-parity check.

In the event that the ECC memory option is not installed, all memory-parity errors are treated as ECC-uncorrected errors.

TABLE 2.11.
FAULT PROCESSING.

Column 1 is the vector location in segment zero for an indirect JST when process-exchange mode is off. Column 3 is the offset within the fault vector of the applicable CALF when process-exchange mode is on. The "ring" column shows whether the fault is handled in the ring of occurrence or in ring zero. In the "saved P-counter" column, "current" means the saved P-counter is not reset back to the beginning of the most recently attempted instruction; "backed" means that it is.

PX off vector	fault type	PX on offset	ring	saved P-counter	FCODE	FADDR
62	restricted instruction	Ø	current	backed	-	-
63	process	4	zero	current	abort flags	-
64	page	1Ø	zero	backed	-	address
65	SVC	14	current	current	-	-
66	unimplemented instruction	2Ø	current	backed	current P-ctr	eff address
72	illegal instruction	4Ø	current	backed	current P-ctr	eff address
73	access violation	44	zero	backed	-	address
74	arithmetic exception	5Ø	current	current	excep code	operand addr
75	stack overflow	54	zero	backed	-	last stk seg
76	segment	6Ø	zero	backed	1=# too big 2=fault bit	address
77	pointer	64	current	backed	ptr 1st word	addr of ptr

Exception codes for arithmetic exceptions are as on the PRIME 3ØØ with the addition of code ´ØØ14ØØ (hexadecimal Ø3ØØ) for integer exception (see Section 2.1Ø).

TABLE 2.12.
DIAGNOSTIC STATUS WORD.
(6 words long)

Set on all checks except power failure as follows (the Diagnostic Status Word is untouched by a power failure check):

Bits 1-32 (register file ′34 absolute): DSWRMA.

Bits 33-64 (register file ′35 absolute): DSWSTAT.
IHPMKKKWCUBPPPXO
A-SSSSSN--TTTTTT

Bits 65-96 (register file ′36 absolute): DSWPB.

bits		meaning, validity
1-32	(DSWRMA)	Memory address register. Valid if and only if a machine check occurred but not a missing-memory-module check, or else RMA invalid (bit 49) is reset on a missing-memory-module or memory-parity check. Invalid if and only if no check has occurred, or else RMA invalid is set on a missing-memory-module or ECC-uncorrected check. In the event of multiple checks, DSWRMA is the RMA of the missing-memory-module check if any, else of the machine or ECC-uncorrected check (they are mutually exclusive) if any, otherwise of the ECC-corrected check.
33	I	Check immediate. The check could not be held off until end-of-instruction. Always valid.
34	H	Machine check. Always valid. If set, bits 37-40 are valid.
35	P	Memory-parity check. Always valid. If set, bit 56 is valid.
36	M	Missing-memory-module check. Always valid. If set, bits 49 and 56 are valid.
37-39	KKK	Machine-check code. Valid only if bit 34 is set. Parity failure on 0=peripheral data (BPD) output, 1=peripheral address (BPA) input, 2=memory data (BMD) output, 3=cache data (RCD), 4=peripheral address (BPA) output, 5=RDX-BPD input, 6=memory address (BMA), 7=register file.
40	W	Not RCM parity. Reset if and only if there is an RCM parity error and the extended control storage option is installed. Valid only if bit 34 is set.
41	U	ECC-uncorrected memory-parity check (or, any memory-parity check when the ECC memory option is not installed). Always valid. If set, bit 35 is set, and bit 56 is valid.
42	C	ECC-corrected memory-parity check. Always valid. If set, bit 35 is set, and bits 51-56 are valid.

43	B	Backup count invalid. Always valid. If reset, bits 44-46 are valid.
44-46	PPP	RP (P-counter) backup count. Amount to subtract from DSWRP to find the beginning of the most recently attempted instruction. Valid only if bit 43 is reset.
47	X	Check occurred during DMX service. Always valid.
48	O	Check occurred during DMX service, programmed input/output, or interrupt microcode. Always valid.
49	A	RMA invalid. If set, no RMA is available in DSWRMA. Valid if and only if a missing-memory-module check occurred, or else a memory-parity check occurred but not a machine check. Invalid if and only if there was no check, or else a machine check occurred without a missing-memory-module.
50		Reserved.
51-55	SSSSS	ECC-corrected syndrome bits. Valid only if bit 42 is set.
56	N	Memory module number (failing memory module in case of interleaved memories). Valid only if bit 35 or bit 36 is set. If both bits are set, bit 56 is the module number which goes with the missing-memory-module check.
57-58		Reserved.
59-64	TTTTTT	Microverify failing test number. Valid only following failure during Master Clear or VIRY instruction.
65-96	(DSWPB)	Extended program counter (ring, segment, word). Always valid. In the event of multiple checks, DSWPB is the program counter of the missing-memory-module check if any, else of the machine or ECC-uncorrected check (they are mutually exclusive) if any, otherwise of the ECC-corrected check.

2.13 QUEUES AND DMQ.

Queue structures on the PRIME 400 are double-ended queues ("deques", to quote Knuth), and are used for both input/output (DMQ mode, physical queues) and interprocess communications (virtual queues). Each queue is implemented by an array of 2**K words for data and a four-word control block. The data block format is shown in Table 2.13 and the control block format in Table 2.14.

The data block is constrained to be of length 2**K for some 4<=K<=16 and the origin of the queue is constrained to be M*2**K. These restrictions on the data block allow the beginning and ending of the data block to be easily inferred from the read or write pointer. Let us define MASK to be a word with K '1' bits on the right and 16-K '0' bits on the left, i.e., MASK=2**K-1: Then if P points inside the data block, then

ORIGIN = P .AND. (.NOT. MASK)

and

END = P .OR. MASK

The control block entries mean as follows:

Top (Read) Ptr:	Points to the datum at the head of the queue;
Bottom (Write) Ptr:	Points to the cell after the datum at the tail of the queue;
Segment:	Six bits of address extension or else segment number;
MASK:	=2**K-1 defines the size of the queue data block.

Notice carefully that the queue could contain from 0 to 2**K entries, but to reserve the condition Top-Ptr = Bottom-Ptr for empty, we must define the queue to be full when it has 2**K-1 entries: i.e., there is always one slot empty.

The DMQ mode of I/O is defined by a DMX request of BPCMO...4=00001 for input and BPCMO...4=00000 for output. In the input mode, a word is added to the bottom of the queue if there is room, else an EOR (End of Range) signal is returned to the controller. In output mode, data is taken from the top of the queue or, if empty, a zero word is output along with EOR. Note that EOR is not put out with the word that empties the queue as with DMA. All memory operations bypass cache. An important special case is output when the queue is empty, which requires only two reads (the Read-Ptr and Write-Ptr), a comparison, and a speedy exit. This efficiency consideration accounts for the peculiar ordering of words in the control block.

The DMQ modes assume that the BPA address refers to a control block in

segment zero which in turn refers to a data block in physical memory.

The instructions provided for queue manipulation are of the generic-AP class, in which a following AP-pointer (see Section 2.4) provides the address of the queue control block.

Data is in the A-register and the results of the operation are given in the condition code bits for later testing. No Wait or Notify action is taken by the instruction per se. The instructions are:

ATQ	P	Add to Top of Queue
ABQ	P	Add to Bottom of Queue
RTQ	P	Remove from Top of Queue
RBQ	P	Remove from Bottom of Queue
TSTQ	P	TeST Queue

The Ptr refers to a control block in virtual space which is shown in Table 2.14. The virtual queue control block differs from the physical in that a segment number is provided instead of a physical address. Ring zero privilege is required to manipulate physical queues. Also, the ring number determines the privilege of access into both the control block and the data block.

The algorithms for queue operation are as follows (T1,T2,T3,T4 and T5 are temporary registers):

A. RTQ or DMQ output

1. T1 <- Top
2. T2 <- Bottom
3. if T1 = T2 exit, Queue Empty, EOR
4. T3 <- Segment
5. T4 <- Mask
6. A <- (T1)
7. Top <- T1 .AND. .NOT. T4 .OR. (T1 + 1) .AND. T4

Note that EOR is determined after only two memory references and the top pointer is updated after the data is removed. Similarly, for input the algorithm is:

B. ABQ or DMQ input

1. T1 <- Top
2. T2 <- Bottom
3. T3 <- Segment
4. T4 <- Mask
5. T5 <- T2 .AND. .NOT. T4 .OR. (T2 + 1) .AND. T4
6. if T1 = T2 exit, Queue Full, EOR
7. (T2) <- A
8. Bottom <- T5

Note that here all four control words must be fetched before any operation or testing can take place. Also note that the data is

inserted before the pointer is updated. This insures that the sequence ABQ/DMQ-output and DMQ-output/RTQ can work without interlock in either microcode or software. The other two algorithms are:

C. RBQ

```
1.  T1 <- Top
2.  T2 <- Bottom
3.  if T1 = T2  exit, Queue Empty
4.  T3 <- Segment
5.  T4 <- Mask
6.  T2 <- T2 .AND. .NOT. T4 .OR. (T2 - 1) .AND. T4
7.  A <- (T2)
8.  Bottom <- T2
```

D. ATQ

```
1.  T1 <- Top
2.  T2 <- Bottom
3.  T3 <- Segment
4.  T4 <- Mask
5.  T1 <- T1 .AND. .NOT. T4 .OR. (T1 - 1) .AND. T4
6.  (T1) <- A
7.  Top <- T1
```

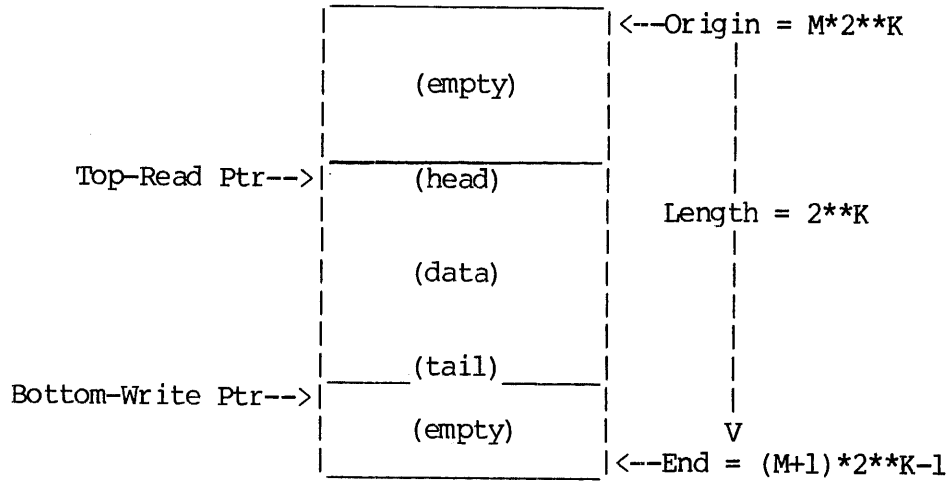
In addition, the queue can be tested by the instruction TSTQ which calculates the length of the data queue and compares the result with Ø and Mask. Interestingly, the length of the data queue is:

$$L = (Bottom - Top) \ .AND. \ MASK$$

whether the data is wrapped or not!

TABLE 2.13.
QUEUE DATA STRUCTURES.

QUEUE DATA BLOCK, DATA NOT WRAPPED



QUEUE DATA BLOCK, DATA WRAPPED

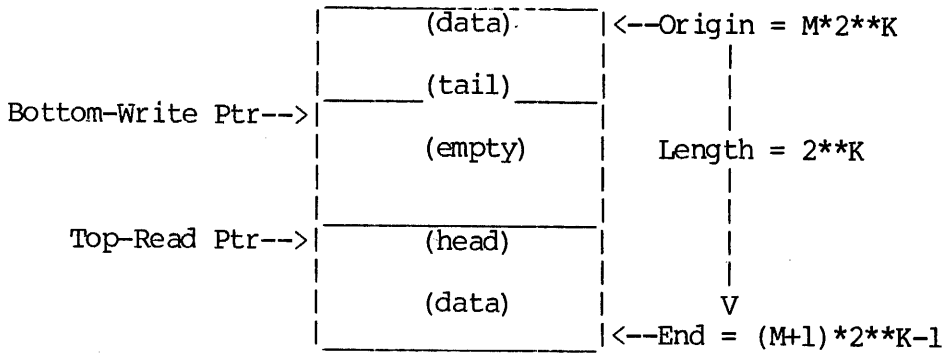


TABLE 2.14.
QUEUE CONTROL BLOCK.
(4 words long)

```
TTTTTTTTTTTTTTTT
BBBBBBBBBBBBBBBB
V---SSSSSSSSSSS
MMMMMMMMMMMMMMM
```

bits	function
1-16:	Top (read) pointer (TTT...T).
17-32:	Bottom (write) pointer (BBB...B).
33:	Virtual/physical control bit, virtual queue if set, physical queue if reset (V).
34-36:	Reserved, must be zero.
37-48:	Segment number of queue data block if a virtual queue, high-order physical address bits if a physical queue (SSS...S).
49-64:	Mask of value 2**K-1 (MMM...M).

2.14 OTHER NEW INSTRUCTIONS.

The PRIME 400 has numerous other new instructions. Most of them are self-explanatory, but a few which are not so obvious are described in this section.

XEC (MR 01 02) Execute.

Not all instructions can be executed by the Execute instruction. The instructions which can be executed differ in the PRIME 400 from those in the PRIME 300.

No multiple-word instructions can be executed properly. All one-word instructions can be executed properly except JMP, JST, and address-mode-changing generics. Instructions which skip do so relative to the XEC instruction. Effective-address calculation of memory-reference instructions is relative to the location of the executed instruction. Instructions which are illegal or unimplemented cause illegal or unimplemented instruction faults when executed. On any fault or interrupt the saved program counter is relative to the XEC instruction.

LPSW (GEN-AP 000711) Load Program Status Word.

Load Program Status Word is a restricted operation which can change the status of the processor. It can be executed only in ring zero. The instruction addresses a four-word block containing an extended program counter (ring, segment, and word numbers) in the first two words, keys in the third word, and modals in the fourth. The program counter and keys of the running process are loaded from the first three words, then the processor modals are loaded from the fourth. If the new keys have the in-dispatcher bit (bit 16) off, the current process continues in execution but at a location defined by the new program counter. If the new keys have the in-dispatcher bit on, the dispatcher is entered to dispatch the highest priority ready process. Whenever the current process again becomes the highest priority ready process, it will then resume execution at the point defined by its new program counter. The modals are associated with the processor and not the process, so in either case the new modals are effective immediately.

This instruction is used to load the four words of the register file which cannot be correctly loaded with the STLR instruction: the program counter (ring, segment, and word number), the keys, and the modals. The STLR instruction should not be used to set these words, as it does not update the separate hardware registers in which the processor maintains duplicate information to achieve higher performance.

The LPSW instruction must never attempt to change the

current-register-set bits of the modals (bits 9-11). This implies that, unless for some reason the current register set in effect for the execution of the program is known with certainty, any program wishing to execute an LPSW must inhibit interrupts (to prevent an unexpected process and register exchange), read the register set currently in effect from the present modals (as with an LDLR ´24), mask those register-set bits into the modals to be loaded, and then finally execute the LPSW. Fortunately, in both usual applications of LPSW the needed register-set bits are predictable: when LPSW is first used after Master Clear to turn on process-exchange mode, the current-register-set bits should be Ø1Ø (the processor always comes out of Master Clear in register set 2); and when LPSW is used to return from a fault, check, or interrupt handled by inhibited code, whatever register-set bits were stored away by the fault, check, or interrupt are still correct and can simply be reloaded.

SimilarLy, except to reload status correctly stored on a fault, check, or interrupt, an LPSW should never attempt to set either the save-done bit (bit 15) or the in-dispatcher bit (bit 16) of the keys. The initial LPSW following a Master Clear should have both these bits off.

LDLR (MR Ø5 Ø1) Load Long from Register File.
STLR (MR Ø3 Ø1) Store Long into Register File.

These instructions load or store a 32-bit quantity in the combined A-,B-register from or into any cell of the register file. The register-file cell selected is determined by the effective address of the LDLR/STLR (these instructions do not access main memory). Both absolute and relative register-file addressing is provided. In absolute addressing, any particular cell of the 128 in the entire register file can be specified. In relative addressing, any cell of the 32 which are the current register set can be specified, and the hardware uses the current-register-set bits of the modals to convert the relative address to absolute before the access.

Only the word-number portion of the effective address of the LDLR/STLR is inspected. Bit 2 of the word number determines whether absolute or relative addressing is used. If bit 2 is reset, addressing is relative to the current register set. In this case, bits 12-16 of the word number select one of the 32 cells of the current register set. Bits 1 and 3-11 of the word number are reserved and must be zero. The layout of each user register set (there are two in the PRIME 4ØØ) is shown in Table 2.6.

If bit 2 of the word number is set, absolute register-file addressing is used. In this case, bits 1Ø-16 of the word number select one of the 128 cells of the entire register file. Bits 1 and 3-9 of the word number are reserved and must be zero. Cells ´Ø through ´37 absolute of the register file are microcode temporaries. The only microcode temporaries of interest to the programmer are:

'02 TR2 (see Section 3);
'03 TR3 (see Section 3);
'21 PBSAVE (see Section 2.12);
'30 PSWPB (see Section 2.12);
'31 PSWKEYS (see Section 2.12);
'32 PPA, level and PCB location (see Section 2.11);
'33 PPB, level and PCB location (see Section 2.11);
'34 DSWRMA (see Table 2.12);
'35 DSWSTAT (see Table 2.12);
'36 DSWPB (see Table 2.12).

Register-file addresses '40 through '77 absolute are direct-memory-access (DMA) channel cells. Register-file addresses '100 through '137 absolute are user register set 2, and addresses '140 through '177 absolute are user register set 3.

The LDLR/STLR instructions are partially restricted. In ring zero, the full functionality described above is available. In other than ring zero, specification of an absolute address or of a relative address higher than '17 causes a restricted mode violation.

CGT (GEN 001314) Computed Go To.

The Computed Go To instruction implements the FORTRAN computed GO TO construct. The instruction word is followed by N further words, the first of which must contain the integer N and the remaining N-1 of which contain word numbers within the procedure segment of possible branch addresses. If the contents of the A-register are less than one or greater than N-1, no branch is taken and control passes to the instruction following the branch list. If the contents of the A-register are between one and N-1 inclusive, the corresponding branch address is selected and control passes to that word number within the procedure segment.

STAC (GEN-AP 001200) Store A-Register Conditionally.
STLC (GEN-AP 001204) Store Long Conditionally.

The Store A-Register Conditionally instruction stores the contents of the A-register into memory if and only if the present contents of the addressed memory word are identical to that of the B-register. The comparison and store are guaranteed not to be separated by the execution of any other c.p.u. instructions. That is, it is not possible for any other instruction to change the contents of the addressed memory word after the comparison has been made but before the store takes place. The condition-code bits are set "equal" if the store takes place, otherwise "unequal".

The Store Long Conditionally stores the 32-bit contents of the combined A-,B-register into a memory pair if and only if the present contents of

the addressed pair are identical to that of the combined EH-,EL-register (the extended accumulator). Again, the condition-code bits are set "equal" if the store occurs, otherwise "unequal".

These instructions are provided to aid cooperating sequential processes in the manipulation of shared data. They often permit removal of mutually exclusive critical sections, hence possibly indefinite delays, from algorithms which would otherwise have required them.

Neither of these instructions is interlocked against direct-memory input/output. Hence, these instructions should not be used to interlock a process with a DMA, DMC, or DMQ channel.

SECTION 3

CONTROL PANEL

The control panel for the PRIME 400 is the same physical panel used for the PRIME 300. Its functionality was enhanced by improving the u-code in the CP. All switches and selectors operate exactly as for the PRIME 300 with the exception of the sense switches in the up position. Table 3.1 is a diagram of the functionality of the switches. Notice that with all switches down, any FETCH/STORE operations are to/from memory-mapped. As long as segmentation mode is not turned on, mapped and absolute are the same, thus preserving compatibility. If SS4 down were absolute, address traps could not occur and would thus be incompatible. Notice also that SS5-16 in the up position changes meaning depending upon SS4. When mapped, all 12 switches are read as a 12-bit segment number. When absolute, SS11-16 are used as the 6 high order bits of the 22-bit physical address. To address any PRIME 300 registers, all sense switches should be placed in the down position and addresses between 0 and '37 specified.

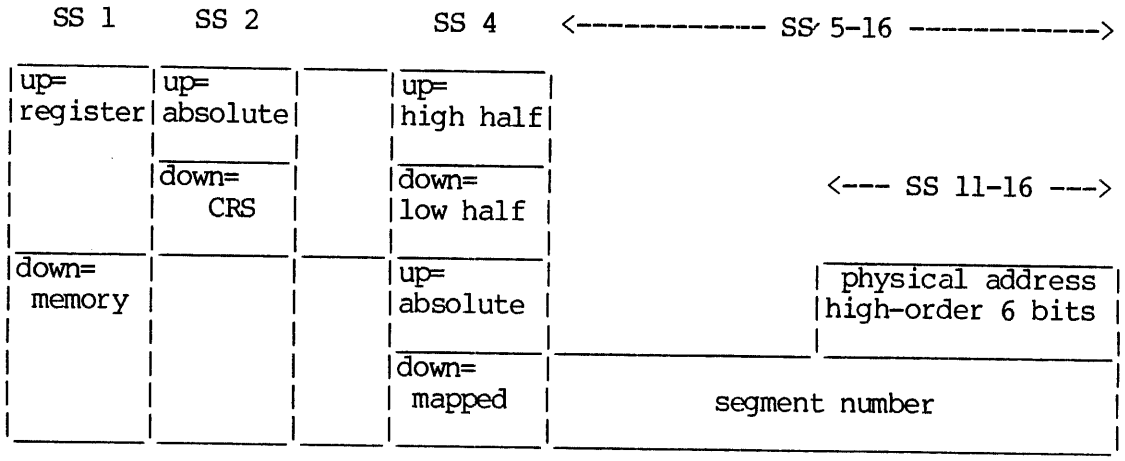
PRIME 400 registers are accessed by raising SS1. Then, if SS2 is down, the low order 5 bits of the address are used to access 32-bit registers 0-'37 within CRS. If SS2 is raised, the full 7 bit address is used to access any register in any register file. The addresses, as shown in Figure 16, are 0-'37=u-code scratch/system, '40-'77=DMA, '100-'137=User set 2, and '140-'177=User set 3. SS4 is used to access either the high half (up) or the low half (down) of the selected register. For all register accesses, the Y+1 functions will advance the register address before the access, exactly as for memory accesses. Wrap around will occur on the appropriate number of bits, since any bits of higher order are ignored for the access.

The control panel data register is TR2H and the address register is TR3. Upon entering the control panel routine, RP is saved in TR3 and (RP) is saved in TR2H. In addition, the keys (KEYSH) are updated to reflect accurately the live keys. Thereafter, TR3H is not altered by the control panel itself so RPH is always remembered. However, on exit, PBH is used to update RPH and KEYS is used to update all the keys. As a result, single stepping can change segments as well as keys and modals.

The only exception to the control panel entry protocol is that if a Fault, Check, or external Interrupt attempts to vector through a vector containing 0 in PRIME 300 mode, the following registers will contain:

RP:	address of 'trapped' instruction
PBH:	SN of 'trapped' instruction
KEYSH:	proper keys
TR2H:	(data) 0
TR3:	(address) 0|0
TR2L:	address, in segment 0, of the 'vector' containing 0

TABLE 3.1.
CONTROL PANEL.



With all switches down, the control panel works exactly as for the PRIME 300 following a Master Clear or a HLT. It is necessary to make mapped (SS 4 down) memory references to generate address traps (access registers as memory, as in short-form instructions). If segmentation mode is on, mapped references are to segment zero unless some other segment number is entered in SS 5-16. When accessing the register file (SS 1 up), only the low-order 5 bits (SS 2 down) or 7 bits (SS 2 up) of the address are used for register selection; the "Y+1" functions increment the address for registers in the same way as for memory.

INDEX

PRIME

PRIME Computer, Inc., 145 Pennsylvania Avenue, Framingham, Massachusetts 01701