PRIME

REFERENCE GUIDE SYSTEM ARCHITECTURE & INSTRUCTIONS IDR3060

P/N MAN3251-001

PRIME SOFTWARE DOCUMENTATION

HIGH LEVEL LANGUAGE PROGRAMMER'S GUIDES

- **FORTRAN IV**
  PDR3057
  PTU47
- **COBOL**
  PDR3056
  PTU48
- **RPGII**
  IDR3031
  FDR3275*
  PTU49
- **BASIC/VM**
  IDR3058
- **INTERPRETIVE BASIC**
  IDR1813

ASSEMBLY LANGUAGE REFERENCE GUIDES

- **SYSTEM ARCHITECTURE INSTRUCTIONS**
  IDR3060
  MAN1812*
- **PRIME MACRO ASSEMBLER**
  PDR3059
  PTU50

OPERATING SYSTEM REFERENCE GUIDES

- **PRIMOS COMMANDS**
  PDR3108
  FDR3250*
- **SYSTEM ADMINISTRATOR GUIDE**
  IDR3109
- **FILE SYSTEM**
  PDR3110
  PTU51
- **SOFTWARE LIBRARY**
  PDR3106
  PTU52

SOFTWARE SUBSYSTEM REFERENCE GUIDES

- **DATA BASE MANAGEMENT**
  IDR3043
  IDR3044
  IDR3045
  IDR3046
  PTU55
- **EDITOR & RUNOFF**
  FDR3104
- **MIDAS**
  PDR3061
  PTU54
- **SPSS**
  PDR3173
- **FORMS**
  IDR3040
  PTU45
  PTU53

COMMUNICATIONS SUBSYSTEM REFERENCE GUIDES

- **RJE/2780**
  PDR3067
- **HASP**
  IDR3107

PRIME DOCUMENTATION TYPES

IDR **Initial Documentation Release:** provides usable, accurate advanced information.

PDR **Preliminary Documentation Release:** provides more complete and accurate information about the product, but is not in final format.

FDR **Final Documentation Release:** a complete product description; edited, formatted and presented in Prime's highest standards. **The Programmer's Companion*** is another type of FDR; a series of pocket-size, quick reference guides on Prime software products.

PTU **Prime Technical Update:** interim updates to existing documents.

PREFACE

This document is an IDR (Initial Documentation Release) on System Architecture and Instructions for the Prime 100 through 500. This reference guide is being written in three stages:

1. IDR stage

   - Includes the completely revised instruction set and supporting data structures.
   - Incorporates all the remaining material from the superseded manuals and PETS.

2. PDR stage

   - Updates the instruction set summary.
   - Includes a complete revision of all system architecture descriptions.

3. FDR stage

   - Revises and updates the PDR in a final typeset version.

Parts II and III of this IDR contain the Prime 100 through 500 instructions set definitions with supporting data and instruction formats, grouped according to addressing mode. Combined with the PMA User Guide, it provides the programmer with the information necessary to program in the PMA assembly language.

Part I and the Appendices contain all the information from the superseded manuals which was not rewritten for Parts II and III. This section will be completely rewritten for the PDR.

The manuals and PETS which are superseded by this manual are:

MAN 1671 - System Reference User Guide
MAN 2798 - P400 System Reference
PET -300 - Introduction to P500
PET -381 - Prime P500 Extended Instruction Set

All correspondence on suggested changes to this docment should be directed to:

Rosemary Shields, Technical Writer
Technical Publications Department
Prime Computer, Inc.
#3 Newton Executive Park
Newton, Massachusetts Ø2165

We wish to thank the members of the SYSTEM ARCHITECTURE AND INSTRUCTION GUIDE team and also the non-team members, both customer and Prime, who contributed to and reviewed this IDR.

First printing July 1978

PART I - OVERVIEW

PART II - INSTRUCTION SUMMARY - V, R, AND S MODES

APPENDICES

PART ONE

OVERVIEW

SECTION 1

INTRODUCTION

This reference guide emphasizes the Prime 350, 400 and 500. However, the capabilities of the Prime 100, 200 and 300 may be considered as a subset of the larger processors.

THE CENTRAL PROCESSORS

There are six Prime processors: the single-user Prime 100 and Prime 200; the 31-user Prime 300 and 350; the 63-user Prime 400; and the top-of-the-line Prime 500. They are completely upward compatible, insuring easy and economical growth.

The Prime 500 supports 32 million bytes of virtual memory, 8 million bytes of 600 nanosecond MOS main memory, 2.4 billion bytes of disk storage, and 63 simultaneous users. It provides high system throughput for both business and computation applications with its parallel logic, high-speed double precision floating-point arithmetic unit, and high speed business instruction set. The processor's performance is enhanced by an 80-nanosecond cache memory, error correcting MOS memory, stack architecture, 128 32-bit registers, automatic machine state switching, and a performance-optimized microcode structure.

The Prime 400 can be substituted for the Prime 500 where applications do not require the 500's high-speed floating-point computation or firmware enhancements for COBOL execution. Both processors offer the same paged and segmented memory management techniques; up to 32 million bytes of virtual memory, and up to 8 million bytes of main memory; both support the same peripheral system, including disk subsystems with a capacity of up to 2.4 billion bytes; both provide extensive microprogrammed system integrity features; and both are capable of supporting up to 63 simultaneous users.

The Prime 350 combines the large-scale capabilities of the shared virtual memory Prime 500 and Prime 400 with the economy of the Prime 300. It has up to 2 million bytes of virtual memory that can be shared by up to 31 simultaneous users, each running programs up to 768K bytes long.

The Prime 300 is a smaller-scale processor that provides an economical balance of features for timesharing systems with up to 31 users. A paged memory management system gives each user a 128K-byte virtual address space. Main memory can be expanded in 64K-byte increments to a maximum of 512K bytes.

The Prime 200 and 100 provide low cost central processor resources for dedicated applications such as single-user computation and real-time data acquisition and control. The Prime 200 is a higher performance version of the Prime 100 and in addition to a 128K-byte memory capacity, offers optional high-speed multiply-divide, and byte parity checking throughout the processor and main memory.

SOFTWARE FIRST: THE PRIME BLUEPRINT

Prime central processors offer higher throughput, faster response, and greater convenience than others because of Prime's unique "Software First" approach to computer system design.

"Software First" means that Prime designs its system software first and then develops a family of processors that offer combinations of performance features, instruction sets and memory capacities that guarantee optimum software performance.

Since all Prime software is based on a common file structure and a uniform operating system (called PRIMOS) the software is the blueprint for hardware design. As a result, each processor has specific features that enhance software performance and provide a more responsive system.

For example, a process exchange hardware/firmware scheme automatically allocates Prime 350, 400 and 500 resouces to the highest priority process. Dual register sets in the Prime 400 and 500 increase throughput. Procedure calls to reentrant subroutines minimize system response time. Stack procedure instructions simplify recursive and reentrant programming procedures. COBOL decimal arithmetic, character manipulation, and character editing instructions are implemented in firmware on the Prime 500. And to enhance FORTRAN compiler throughput, all Prime processors feature a unique set of 'logicize' instructions that automatically convert comparision results directly to truth table values.

COMPATIBILITY

Since a common and uniform software system came first, the processors that run the software are compatible. Programs developed on smaller processors can be preserved completely when upgrading to a larger system. Program development for small scale systems can be steamlined by using the expanded file handling and memory resources of a larger multi-user system.

Prime central processors are not only logically compatible, but also electromechanically compatible. This means that upgrading a system's central processor performance is as simple as swapping circuit boards. For example, a Prime 400 can be upgraded to a Prime 500 by replacing the dual Prime 400 processor boards with the Prime 500 three-board set. All existing software will run on the new processor without modification--and faster.

CENTRAL PROCESSOR FEATURES

Prime central processors have many important features, some unique, some in common. Tables 1-1 through 1-4 itemize specific processor

features, detailed performance characteristics, and operating specifications.

Microprogrammed Logic

All Prime computers use microprogrammed logic. Microprogramming frequently used subroutines, algorithms, and special purpose instructions improves the processor's speed and reduces main memory storage requirements.

MOS Memory

From its beginning, Prime has been committed to using state-of-the-art MOS technology in its memories. As a result, Prime was first to package 64K bytes on a single board using 4K chips. Today, Prime continues to advance the state-of-the-art by also offering 256K bytes on a single board using 16K chips. The packaging density means Prime users can have up to 8 million bytes of main memory--more than many mainframes--in an extremely compact configuration (1 megabyte occupies a mere four inches of vertical cabinet space).

System Integrity Features

Memory byte parity is standard on the Prime 200, 300, 350, and 400 central processors (the Prime 500 has standard error correcting memory). Every word in main memory is treated as a pair of eight-bit bytes, each with a ninth parity bit. Memory parity is checked when a word is read, and generated when a word is written. Errors are reported to the operating system for appropriate action.

Processor byte parity is standard on all Prime processors except the Prime 100. Parity is checked or generated for all data transfers on the bus between the main memory and processor, among all processor registers, on all internal processor busses, and on the bus between the processor and all I/O devices.

Microverification routines are optional on the Prime 200 and standard on the Prime 300, 350, 400, and 500. They are a series of microprograms that verify the processor's integrity by checking and excercising all processor components, other than basic clocks and control circuits. They are activated automatically on the Prime 300 by a processor parity error. On the Prime 350, 400, and 500 they are activated by a system master-clear, or can be initiated under user program control. When one of the microverification routines detects a fault condition, it is reported to the operating system for appropriate action. If the system is halted, the control panel lights display the number of the test that failed.

Table 1-1. Central Processor Features

Central Processor	100	200	300	350	400	500
Feature Availability						
No. of programmable DMA channels			8	8	32	32
No. of addressable 32-bit registers				72	128	128
Extended direct memory access (DMC,DMT)	O	O	S	S	S	S
Direct Memory queue (DMQ)				S	S	S
Full control panel	S	S	S	S	S	S
Unimplemented instruction trap	S	S	S	S	S	S
High-speed register file (32 addressable 16-bit registers)	S	S	S			S
8 general purpose registers						
Hardware multiply/divide and double precision arithmetic	O	O	S	S	S	S
Automatic program loaders (standard devices)	O	O	S	S	S	S
Microverification routines		O	S	S	S	S
Processor byte parity		S	S	S	S	S
Memory byte parity		S	S	S	S	S
Error correcting memory				O	O	S
Single- and double-precision floating-point arithmetic		O	O	S	S	S
32-bit arithmetic logic unit				S	S	S
32-bit integer arithmetic				S	S	S
64-bit integer arithmetic				S	S	S
Virtual memory capability (paging)			S			
Virtual memory capability (paging and segmentation)				S	S	S
Stack processing instructions			S	S	S	S
2K-byte cache (80 nanosecond cycle time)				S	S	S
Hardware process exchange				S	S	S
Ring protection structure				S	S	S
Business instructions						S
Fast floating-point arithmetic						S

Table 1-2. Operational Characteristics

Central processor	100	200	300	350	400	500
Word Size: Memory bits	16	16	16	16	16	16
Internal	16	16	16	32	32	32
Instruction size	basic format 16 bits;extended format, 32 bits					
Addressing	direct, indexed and indirect in sectored and relative modes;					
Minimum-maximum						
main memory (bytes)						
(K=1,024)	8-128	8-128	64-512	64K-512M	128K-8M	256K-8M
Memory access time (ns)	680	600	600	600	600	600
Memory increment						
per board (bytes)	8K,16K,32K	8K,16K,64K	64K	64K	64K,256K	256K
Maximum program						
size (bytes)	128K per program			768K	32M	32M
Maximum virtual						
memory space (bytes/user)			128K	2 M	32M	32M
I/O data path (bits)	16	16 plus 2 parity	16 plus 2 parity	16 plus 2 parity	16 plus 2 parity	16 plus 2 parity
Maximum DMTI/O						
rate (MB/sec)	1.3	2	2.5	2.5	2.5	2.5
Addressable registers						
in high-speed register	32 (includes index register, accumulators, stack					
file	register, DMA addresses, etc.)			72	128	128
Standard instructions	112	117	145	319	318	517
Optional instructions	9	37	19			
Instruction types	memory reference, input/output, generic, shift					
Typical instruction						
times (us):						
Add to memory	2.44	1.96	1.56	1.28	0.56	0.56
Skip on condition	2.84-3.30	2.04-2.32	1.92	1.48	1.20	1.20
Hardware multiply	14	10.48	8.50	6.56	4.20	4.20
Hardware divide	18.2-19.6	13.68-14.72	13.50	7.28	4.76	4.76
Single Precision						
Floating-Point add		9.35+.48A+ .8n	9.25	11.62	5.18	3.72
Floating-Point multiply		27.82	25.20	17.96	9.00	4.02
Floating-Point divide		39.46	37.90	22.52	11.92	6.04
Double Precision						
Floating-Point add				11.74	6.46	4.80
Floating-Point multiply				21.59	20.14	6.46
Floating-Point divide				14.80	24.04	8.68

Table 1-3. Electromechanical Specifications

Chassis Capacity (boards)	5	10	17	24
Chassis dimensions (WxHxD in CM)	45.6x26.7x49.5	45.6x40x49.5	45.6x66.7x49.5	45.6x80x49.5
Weight (including fans and power supply) (kg)	22.7	24.9	45.6	49.8
Operating temp. range (°C)	0°-50°	0°-50°	0°-50°	0°-50°
Max.rel. humidity (no cond.)	95%	95%	95%	95%
Mounting	table top or rack	rack	rack	rack/cabinet
Typical heat dissipation (BTU/hr.)	2,000	3,600	4,000	6000
Voltage range (VAC)	190-250	190-250	190-250	190-250
Hz (single-phase)	47-63	47-63	47-63	47-63
Amps (typical)	5	9	10	15
Power supply	100 amp. main supply, chassis-mounted			100 Amp supply, chassis mounted, two per chassis

Instruction Set

The machine language instruction set permits data manipulation by bit, byte, word and multi-word. Included in the standard Prime instruction set is a group of memory reference instructions that minimize register overhead, and a group of 'logicize' instructions that enhance compiler efficiency by converting comparison relationships directly to truth values.

While each central processor provides a different mix of standard and optional instructions, Prime maintains instruction set compatibility by using unimplemented instruction interrupt hardware and a virtual instruction package containing software equivalents of unimplemented machine-language instructions. This means, for example, programs written to use the Prime 500 instruction set can be executed on other Prime central processors, such as the Prime 400 or 350. And as applications grow from smaller Prime processors to larger, more powerful Prime processors, the basic instruction set used in the original code can use the more sophisticated large processor performance features.

Virtual Memory

Prime's implementation of virtual memory is transparent to system users. This means Prime virtual memory computer users can execute large programs without concern for memory management. Program overlays are no longer needed. Each user is free to create, test, modify, and execute programs without regard for how system resources are managed to perform these functions. Programs written for single user Prime 100 and Prime 200 systems can be executed by virtual memory Prime 300, 350, 400, or 500 systems without modification.

On the Prime 300, each of up to 31 simultaneous users is allocated a 128K-byte virtual address space. Special hardware automatically maps virtual addresses to physical memory. Each user has a page map in main memory that maintains correspondence between the original 'virtual' addresses and the 'real' or physical addresses they are translated into.

On the Prime 350, up to 31 users can share a 2 million byte virtual memory, and on the Prime 400 and Prime 500, up to 63 simultaneous users can share a 32 million byte virtual memory. Both segmentation and paging are used to manage virtual memory facilities. Each user's address space consists of a private segment for user programs, and another that is shared with the operating system. By embedding operating system functions in each user's virtual memory space, all operating system functions are immediately available as if they were an integral part of a user's program, greatly reducing system overhead.

Floating-Point Arithmetic

A general-purpose floating-point arithmetic unit on the Prime 300, 350, and 400 provides direct hardware execution of floating-point

instructions in the Prime instruction repertoire. Floating-point arithmetic is done in either single-or double-precision formats. In the single-precision (32-bit) format, two words are used to store the mantissa and characteristic, and accuracy is maintained to seven significant digits. The double-precision (64-bit) format uses four words and maintains accuracy to 14 significant digits.

The Prime 500's high speed floating-point arithmetic executes double-precision arithmetic about three times faster than the Prime 400. One reason is the use of parallel logic in the floating-point arithmetic unit. Binary multiplication is done four bits at a time, division is done three bits at a time, and addition 48 bits at a time. This is significantly faster than the single-bit algorithms traditionally used for multiply and divide. Users upgrading from any Prime central processor can run all existing programs without modification.

Input/Output

Direct-to-memory input/output operations are supported by three types of program-assignable I/O channels: Direct Memory Access Channels (DMA), Direct Memory Channels (DMC), and Direct Memory Transter (DMT) channels.

The Prime 300 and 350 have 8 DMA channels; Prime 400 and 500 central processors have thirty-two program-assignable DMA channels controlled by high-speed channel address registers, providing high throughput with a minimum of CPU control overhead. DMA channels support a maximum data rate of 2.5 million bytes per second, and are typically used for high speed peripherals such as fast disk devices.

DMC channels, controlled by channel address words in the first 8K bytes of main memory, offer up to 2048 channels for medium-speed I/O transfers such as serial data communications transfer, with a maximum transfer rate of 960K bytes per second.

DMT channels are provided for high-speed device controllers, such as the controllers for moving-head disks, that execute channel control programs. The maximum DMT throughput rate is 2.5 million bytes/second.

In addition to these direct-to-memory channels, a Direct Memory Queue (DMQ) mode of operation on the Prime 350, 400 and 500 provides a circular queue for handling communication devices. The queue reduces operating system overhead by eliminating interrupt handling on a character-by-character basis.

Cache Memory

A high-speed (80 nanosecond access) 2K-byte, bipolar cache memory on the Prime 350, 400 and 500 acts as a high-speed buffer between the central processor and main memory. The cache uses a sophisticated algorithm to determine and get the data the processor is most likely to use next, increasing the apparent speed of the main memory to near that

of the processor. This algorithm provides an average cache 'hit rate' of about 85 per cent. Memory mapping is overlapped with cache memory access, further reducing total instruction execution times.

High-Speed Integer Arithmetic Unit

All integer arithmetic and logical operations on the Prime 350, 400 and 500 are done in the central processor's 32-bit wide arithmetic unit. Using a 32-bit, rather than a 16-bit, format significantly speeds the execution of double-precision integer arithmetic. The arithmetic unit also efficiently handles complex address formation, such as base-plus-displacement, and indexed addressing.

Interleaved Main Memory

With interleaved main memory, consecutive memory locations are on separate memory boards. This two-way interleaving speeds sequential memory accesses and maximizes the cache hit rate. In effect, interleaving provides 32-bit transfers between memory and central processor.

Dual Register Sets

Prime 400 and 500 central processors have 128 32-bit registers for increased throughput. These handle a variety of functions, such as controlling 32 high-speed DMA channels and storing machine states in dual register sets during process exchange operations. The central processor's process exchange mechanism dynamically and automatically manages register set assignment to processes.

Process Exchange

A combination of hardware and firmware automatically allocates Prime 350, 400 and 500 central processor resources to the highest priority process in a queue of processes ready for execution.

Process exchange handles the swapping of machine states necessary for coordinating between processes ready for execution and those waiting for a specific event to occur. Firmware within the process exchange mechanism automatically dispatches the next ready process for execution, without software intervention.

Stack Architecture

Programs on the Prime 350, 400 and 500 operate in a three segment environment: a stack segment containing all local variable values, an instruction or procedure segment, and a linkage segment that contains statically allocated variables and linkages to common data. PRIMOS provides highly efficient addressing modes to access stack and linkage variables. Hardware CALL and RETURN instructions eliminate the overhead of software stack management routines.

The Prime 350, 400 and 500's stack structure optimizes the efficiency of operations like parameter passing, subroutine and procedure calls, arithmetic expression evaluation, and dynamic allocation of temporary storage and context switching.

Business Instructions

The Prime 500 has high-level support for ANSI 74 COBOL and other business oriented languages through comprehensive hardware/firmware instructions designed for decimal arithmetic, character field manipulation, and editing operations.

Decimal arithmetic instructions support packed or unpacked signed numbers up to 18 digits long. They handle operands with different data types and scale factors automatically during add, subtract, multiply, divide, and comparision operations. Users can specify rounding on numeric operations, and instructions provide binary/decimal and decimal/binary conversions.

The Prime 500 does character operations on field sizes of virtually any length. Operations for moves, compares, translates, and searches automatically handle justification, truncation, and padding. Numeric and character editing instructions produce fields in ANSI 74 COBOL-like picture formats.

SECTION 2

THE PRIME 400

OVERVIEW OF THE PRIME 400 PROCESSOR

The Prime 400 is a two-board processor designed to plug into the standard Prime 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 constraints of the Prime Computer family. The processor is very fast (built with high technology) and has segmented addressing (for modern organization and a very large address space).

Thus, the Prime 400 does two things. First, it provides a 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 speed and capacity features for greatly enhanced performance in new applications.

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 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 disk, virtual memory, and real-time operating system known as PRIMOS IV. As PRIMOS IV takes heavy advantage of the advanced features of the Prime 400 processor, it is not downward compatible. However, PRIMOS IV supports all former PRIMOS commands, the existing PRIMOS 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 IV without modification. Furthermore, PRIMOS IV 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, 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 is provided which takes advantage of the segmentation feature, is compatible with previous operating systems, and allows user-space programs which are indifferent to segmentation 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 full utilize segmentation when desired.

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 2-1. 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 PRIMOS III 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 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 respectively, 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 2-2 compares the times for the Prime 300 and Prime 400 I/O modes.

Table 2-1. 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	660 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 +480+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 2-2. Comparison of Prime 300 and Prime 400 I/O Times

mode	input data transfer time	output data transfer time
(PRIME300, 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.

The architectural features which gives 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.

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. In addition, several new features are added:

1. Mapped I/O through segment 0.

2. Remote I/O bus extender and 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 each 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 continous 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 2-2.

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 III operating systems.

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

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.

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 set 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 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 desired to perform verification.

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 2-3 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 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.

PROCESS EXCHANGE ENVIRONMENT

A process is a logically continuous 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 control block describing the process. All PCBs in the system are in a single dedicated segment. The minimum size of a PCB is 64 words. The maximum number of separate processes is 1023. Table 2-4 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

Table 2-3. Virtual Memory formats

DESCRIPTOR TABLE ADDRESS REGISTER FORMAT
(32 bits)

SSSSSSSSSSDDDDDD
-DDDDDDDDDDDDDDD

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).

Table 2-4. 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	Tempory 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 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. After executing either instruction, the highest priority process on the ready list is executed. A NOTIFY (Dijkstra's "V" operation) decrements the sempahore 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 waitlist of some semaphore. A semaphore defines an event whose meaning is shared amont two or more processes. Coordination between processes takes place through sempahores. 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 sempahore 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 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 microcode 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, 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 priorty 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), shown in Table 2-4, 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 - Segment number/word number of Wait List this block is currently on (Segment number=Ø indicates on ready list)
4 - abort flags used to generate process fault when PCB is dispatched.

Current bit assignments 1-15: must be zero
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 1.024 msec resolution
15 - Reserved
16 - Save mask - used to avoid saving and restoring registers = Ø

Bits 1-8: GR0-GR7 (2 words each)
9-12: FP0-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)

50,59 - Fault Vectors: Segment number/word number pointer 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).

Wait and Notify

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 0, 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 activiites. 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 one 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 0, (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 0, 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 opcode Bit 16 = 0
NFYB: use LIFO queuing opcode 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 0, no action is taken, but if greater than or equal to 0, 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 set 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 cleared when a process is dispatched. Whether a register set is avaiable (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 sets. Each set 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 set contains 32 32-bit registers or 64 16-bit registers. The register sets, numbered from 0, are used as follows:

0 - microcode scratch and system registers
1 - 32 DMA channels
2 - User register set
3 - User register set

This layout of register sets allows easy expansion to eight register set, thus adding four new user register sets. All user register sets

have the same internal format and the DMA register set simply consists of 32 channel registers. Channel register '20 within RS1 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 RS1 to channel register '36, mapped to '36 and '37. All other RS1 registers represent additional DMA channels over the Prime 300. Table 2-4 shows the internal layout of the user register sets (RS2, RS3). 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 microcode scratch and DMA, user register sets are number 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 microcode will only look at bit 11.

Direct register set addressing (not using CRS) is accomplished either with the LDLR/STLR instructions or via the control panel. The register sets are ordered sequentially with an absolute address of 0 addressing RS0-register 0 (microcode scratch/system set), '40 addressing RS1-register 0 (DMA set), '100 addressing RS2-register 0 (user set 2), and '140 addressing RS3-register 0 (user set 3).

Cell 30H of the current register set is a 16-bit wide 1024-microsecond up-counting process CPU 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, microcode increments the interval timer (TIMER) in RS (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 clears the interval timer to maintain the elasped 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 a 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 cleared when a process is actually dispatched (return to fetch cycle).

PROCEDURE CALL ENVIRONMENT

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 efffective 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 management. 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.

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.

Procedure Call Instructions (PCL)

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.

1. Computes the ring number of the called procedure.

2. Allocates a stack frame for the called procedure.

3. Saves the caller's critical state information (program counter, stack base register, linkage base register, and keys) in the new stack frame.

4. Loads the critical state for the called procedure.

5. 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), 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 transter:. 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 instructions is evaluated to generate a list of actual argument pointers in the new frame. 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 number) 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. Otherwise it is stored in the three-word format with the E-bit set. In either case, three words are allocated to each pointer in the argument list.

If the caller's template list generates a 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.

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 microcode level.

Traps can occur for many reasons, not all of which cause software visible action, and are always processed on the microcode 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, although 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 interupts operate in either of two modes depending upon whether or not 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 microcode 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 set 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 Ø (I/O segment) of a 16-bit memory cell to be incremented. If the counter does not overflow (-1->Ø), the microcode 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 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 instruction 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 microcode 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 2-5 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 microcode '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 0. 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 0, 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:

+0,+1 Program counter (segment number/word number);

+2 Keys;

+3 Fault code (FCODE in Table 2-7);

+4,+5 Fault address (segment number/word number, FADDR in Table 2-5).

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 0 table must be in a pre-defined (SDW exists) segment (otherwise, segment fault might recurse infinitely). Naturally, the ring 0 table, as well as the PCB, is carefully audited by ring 0 procedures.

The CALF instruction has two major functions. First, to avoid holding off interrupts for too long, the CALF instruciton 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 ('0) of the new frame is set to 1 instead of 0 to distinquish the

Table 2-5. 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	0	current	backed	-	-
63	process	4	zero	current	abort flags	-
64	page	10	zero	backed	-	address
65	SVC	14	current	current	-	-
66	unimplemented instruction	20	current	backed	current P-ctr	eff address
72	illegal instruction	40	current	backed	current P-ctr	eff address
73	access violation	44	zero	backed	-	address
74	arithmetic exception	50	current	current	excep code	operand addr
75	stack overflow	54	zero	backed	-	last stk seg
76	segment	60	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 300 with the addition of code '001400 (hexadecimal 0300) for integer exception

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 microcode routine that is entered from the various fault class 'front ends' and, based on process exchange mode, either simulates a Prime 300 type fault (JST* through segment 0 fault vectors) or performs the Prime 400 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 inthe fault table. Note that for Prime 300 faults, the full RP is also saved in the microcode 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 2-6 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 microcode which was

Table 2-6. Diagnostic Status Word

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.

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 Ø (except ECCC which sets it to 2), and a microcode RTN to the trapped step is executed. In this way, the I/O 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 Ø check vectors) or performs the Prime 400 protocol which includes setting up the check header, inhibiting the machine, and switching to 64V addressing mode. In either mode, MCM is set to Ø 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 occured (these are mutally 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 options is not installed, all memory-parity errors are treated as ECC-uncorrected errors.

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 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 'Ø' 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 of 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,m 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 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 2-10. 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 <- .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 compare the result with 0 and Mask. Interestingly, the length of the data queue is:

L = (Bottom - Top) .AND. MASK

whether the data is wrapped or not!

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 microcode 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. Figure 2-1 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 compatiblity. If SS$ 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=microcode 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 accesss, 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 accurately reflect the keys. Thereafter, TR3H is not altered by the control panel itself is 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

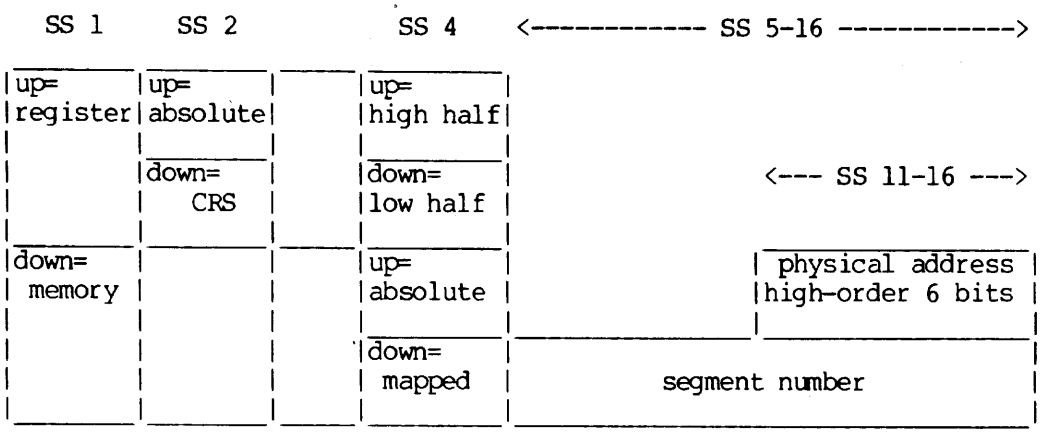
KEYSH: proper keys

TR2H: (data) 0

TR3: (address) 0|0

TR2L: address, in segment 0, of the 'vector' containing 0

Figure 2-1. 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 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.

SECTION 3

PRIME 500

The Prime 500 is a general register 32-bit machine. This architecture is supported by eight 32-bit fixed-point and logical accumulator registers, seven of which can be used for indexing, two 64-bit floating point registers which overlap the two 64-bit field registers, and four 32-bit base registers.

The Prime 500 contains two full register sets with the Prime 400 registers mapped onto them. Figure 1 shows one set of Prime 500 registers with the Prime 400 assignments.

The instruction set for the P500 contains three types of instructions:

1. Memory Reference - each of these instructions can specify any register as a target (source or destination depending on the instruction), any base register plus a 16-bit word-number displacement. The appendix shows this type of memory reference (MR) instruction. The MR format can specify all combinations of single-level indexing and indirect addressing. These options are specified by the tag modifier (TM) field. The TM field is also used to specify register-register (RR) format and the 3 classes of immediate. The appendix shows all of these formats. The MR class of instructions include both full-word (32-bit) and half-word (16-bit) operand types.

2. Register Generics - these are instructions which operate on the specified target register. This class includes the branch instructions which use the second half-word to specify a PB dislacement for the branch address. All the remaining instructions occupy a half-word.

3. Non-Register Generics - this class of instructions includes all of the control instructions including mode change. These half-word instructions have op codes that overlap the Prime 400 generics.

PRIME 500 Register	PRIME 400 Register	
GR0	-	General Register 0
GR1	-	General Register 1 Index register 1
GR2	A,B (L)	General Register 2 Index register 2
GR3	E	General Register 3 Index register 3
GR4	-	General Register 4 Index register 4
GR5	Y,-	General Register 5 Index register 5
GR6	-	General Register 6 Index register 6
GR7	X,-	General Register 7 Index Register 7
FR0H	-	Floating Register 0 (high 32 bits) Field Register 0 high
FR0L	-	Floating Register 0 (low 32 bits) Field Register 0 low
FR1H	FRH	Floating Register 1 (high 32-bits) Field Register 1 high
FR1L	-	Floating Register 1 (low 32 bits) Field Register 1 low
PB	PB	Procedure Base
SB	SB	Stack Base
LB	LB	Link Base
XB	XB	Auxiliary Base

XIS MICRO-CODE

The Prime 500 XIS board is controlled using the same 64-bit field which controls the rest of a Prime 400-Prime 500.

The board is enabled to do anything if and only if the U/A field = 2 or 3 (RCM 29 = 1) and the UIS field = 0 (RCM 31 = 33 = 0) and the IS field does not = 0 (RAM 10 -> 13 has a 1). If the board is enabled, the BB, ALU, RF, BD, IS, and RP/REA u-code fields are reinterpreted to control the XIS board.

The field and bit encodings have been carefully chosen to permit both the current and XIS boards to be active at the same time. The overlap is chosen to permit loading and unloading the XIS board.

The IS field, clock field, cache control field, traps, and next address fields are not affected by the XIS control. However, the XIS microcode

uses these fields. For example, the next address field is used by the XIS microcode for its next address calculations.

EXTENDED INSTRUCTION SET

The Prime 500 Extended Instruction Set contains 15 new instructions to support commercial data processing applications. The instructions are divided into two basic groups, decimal arithmetic and character field processing. The instructions make use of the Prime 400/500 field address and length registers to provide for byte addressing and restartability after interrupts and page faults.

PART TWO

INSTRUCTION SUMMARY V, R, S MODES

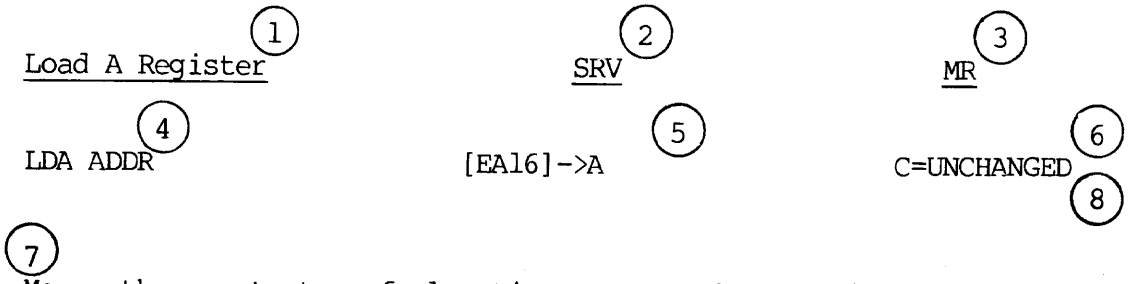
SECTION 4

CONVENTIONS

INSTRUCTION DESCRIPTION CONVENTIONS

This manual describes each of the instructions in the context of the mode where they are first used. To avoid duplicate descriptions while facilitating retrieval, each instruction is described once, but listed in as many categories as appropriate.

The illustration below shows the format we are using.



Move the contents of location ADDR to the A-register. The original contents of the A-register are lost.

(1)- Instruction name

(2)- Addressing modes where it is legal - S, R, V in this case

(3)- Format of instruction - MR is memory reference

(4)- PMA format (numeric followed by argument, if required)

(5)- Instruction control flow in algebraic notation.
[ ] = contents of
( ) = bit number
-> = replaces

(6)- State of condition codes, C-bit, or L-bit

(7)- Description of instruction

(8)- R if restricted; blank if not restricted

FUNCTION GROUP DEFINITIONS

The instruction definitions are grouped by primary function, such as fixed point arithmetic. Table 4-1 below contains the definitions for all the function groups and modes. If you wish to find a particular instruction, Appendix C contains an alphabetic list.

Table 4-1. FUNCTION DEFINITIONS

DEFINITION	S	R	V	I
Addressing Mode	X	X	X	X
Branch			X	X
Character			X	X
Clear field	X	X	X	
Decimal Arithmetic			X	X
Field Register			X	X
Floating Point Arithmetic		X	X	X
Integer Arithmetic	X	X	X	X
Integrity	X	X	X	X
Input/Output	X	X	X	X
Keys	X	X	X	X
Logical Operations	X	X	X	X
Logical Test and Set	X	X	X	X
Machine Control	X	X	X	X
Move	X	X	X	X
Program Control and Jump	X	X	X	X
Process Exhange			X	X
Queue Control			X	X
Shift	X	X	X	X
Skip	X	X	X	X

FORMAT DEFINITIONS

Each instruction has a format. The formats and their meaning are summarized in Table 4-2. Their specific bit definitions are defined in the instruction groups where they are first used.

Table 4-2. FORMAT DEFINITIONS

MNEMONIC	DEFINITION	S	R	V	I
GEN	Generic	X	X	X	
AP	Address Pointer			X	X
BRAN	Branch			X	
CHAR	Character			X	
DECI	Generic Decimal			X	
PIO	Programmed I/O	X	X	X	
SHFT	Shift	X	X	X	
MR	Memory Reference - non I-mode	X	X	X	
MRFR	Memory Reference - Floating Register				X
MRNR	Memory Reference Non Register				X
RGEN	Register Generic				X

GENERAL DATA STRUCTURES

Table 4-3. Data Structures - summarizes all the data structures manipulated by instructions.

Table 4-3. DATA STRUCTURES

CLASS	IDENTITY			
	S	R	V	I
Integer (Unsigned)				
16-bit	X	X	X	X
32-bit			X	X
Integer (Signed)				
16-bit	X	X	X	X
31-bit	X	X		
32-bit			X	X
Floating Point				
32-bit		X	X	X
64-bit		X	X	X
Decimal			X	X
Character string			X	X
Word				
16-bit	X	X	X	
32-bit				X
Halfword - 16 bit				X
Byte	X	X	X	X
Indirect Pointer(IP)				
16-bit	X	X	X	X
32-bit			X	X
48-bit			X	X
Address Pointer (AP)			X	X
Stacks				
Segment Header			X	X
Frame Header			X	X
Argument Template			X	X
Entry Control Block			X	X
Queue Control Block			X	X

PROCESSOR CHARACTERISTICS

Table 4-4, Processor Characteristics, lists the program visible portions of the hardware.

Table 4-4. Processor Characteristics

Class	Identity			
	S	R	V	I
Registers				
100/200/300	X	X		
400/500			X	X
Field Registers			X	X
Floating Registers		X	X	X
Keys				
300	X	X		
400/500			X	X
C-Bit	X	X	X	X
L-Bit			X	X
Condition Codes			X	X
Modals			X	X

SECTION 5

FORMATS - SRV

DATA STRUCTURES

Word Length

16 bits

Byte Length

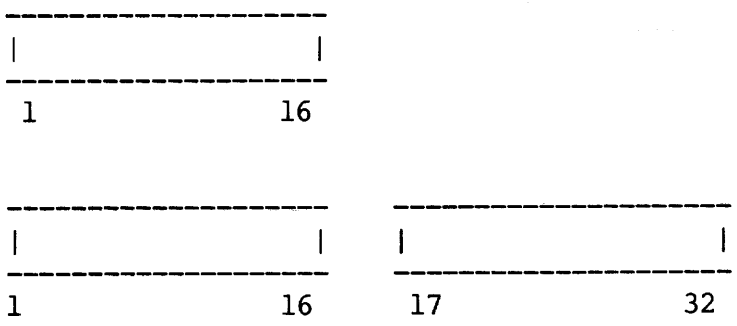
8 bits

Character Strings

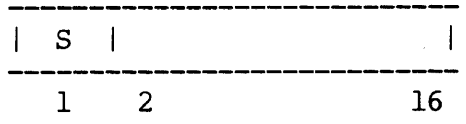
Variable length collection of bytes from 1 to 2**17-1.

Numbers:

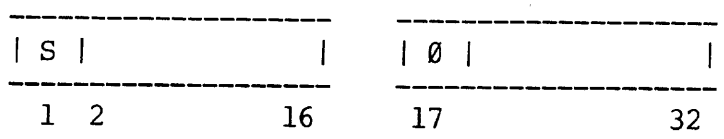
- Unsigned 16 and 32 bit integers



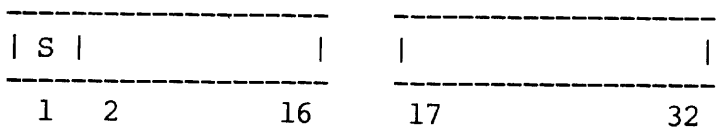
- Signed 16-bit integers



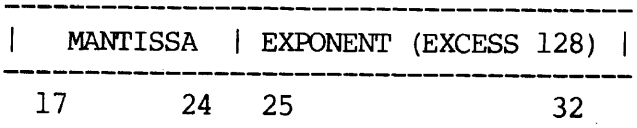
- Signed 31-bit integers (S,R-modes)



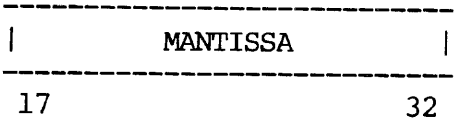
- Signed 32-bit integers (V-mode)

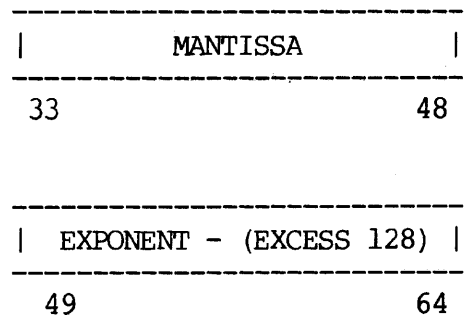


- Floating Point - Single Precision, 32 bits (R,V-modes)



- Floating Point - Double Precision, 64 bits (R,V-modes)



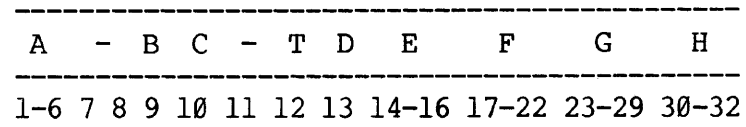


- Decimal - one to 63 digits in five forms (V-mode)

Decimal Control Word Format (V-Mode)

To specify the characteristics of the operation to be performed, most decimal arithmetic instructions require a control word to be loaded in the L register.

The general format is as follows:



Where:

A - Field 1, number of digits

E - Field 1, decimal data type

B - If set, sign of field 1 is treated as opposite of its actual value.

C - If set, sign of field 2 is treated as opposite of its actual value. (XAD, XMP, XDV, XCM only)

D - Round flag (XMV only)

F - Field 2, number of digits

H - Field 2, decimal data type

G - Scale differential (XAD, XMV, XCM only)

T - Generate positive results always

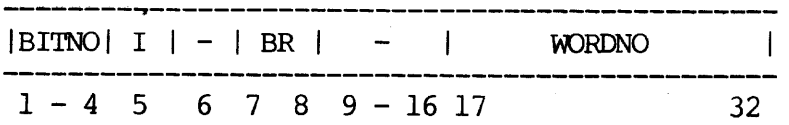
- - Unused, must be zero

The fields used by each instruction are listed in the instruction descriptions. Fields not used by an instruction must be zero.

The scale differential specifies the difference in decimal point alignment between the operator and fields for some instructions. This field is treated as a signed 7 bit two's complement number. Its value is specified as Fx=F1-F2, where Fx is the number of fractional digits in Field x. A positive value indicates a right shifting of Field 1 with respect to Field 2, and a negative value indicates a left shifting.

Address Pointer (AP) (V-Mode)

Two word pointer which follows GENAP instructions.



BITNO (Bits 1-4) - Bit number

I (Bit 5) - Indirect bit

BR (Bits 7-8) - Base register

ØØ = Procedure Base (PB)

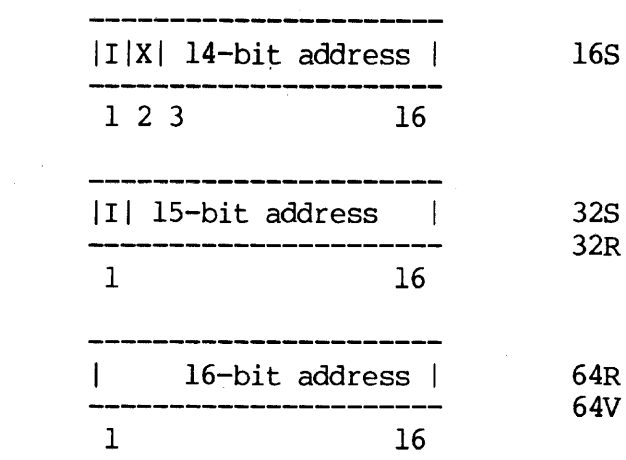
Ø1 = Stack Base (SB)

1Ø = Link Base (LB)

11 = Temporary Base (XB)

WORDNO (Bit 17-32) - Word number offset from base register contents

Indirect Word - One Word Memory Reference



Indirect Pointer - Two Word Memory Reference (IP) (V-Mode)

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

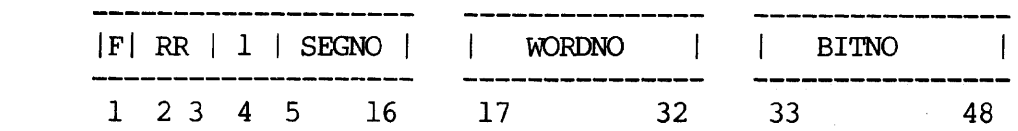
RR (Bits 2-3) - Ring of privilege - controls access rights

Bit 4 = Ø - No third word. Bit number portion of effective address is zero.

SEGNO (Bits 5-16) - The segment number portion of the effective address

WORDNO (Bit 17-32) - The word number portion of the effective address.

Indirect Pointer - Three Word Memory Reference (IP) (V-Mode)



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

RR (Bits 2-3) - Ring of privilege - controls access rights.

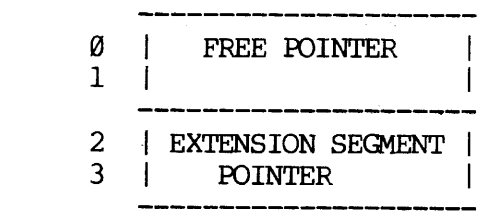
Bit 4 = 1 - The third word is present and gives the bit number portion of the effective address.

SEGNO (Bits 5-16) - The segment number portion of the effective address.

WORDNO (Bit 17-32) - The word number portion of the effective address.

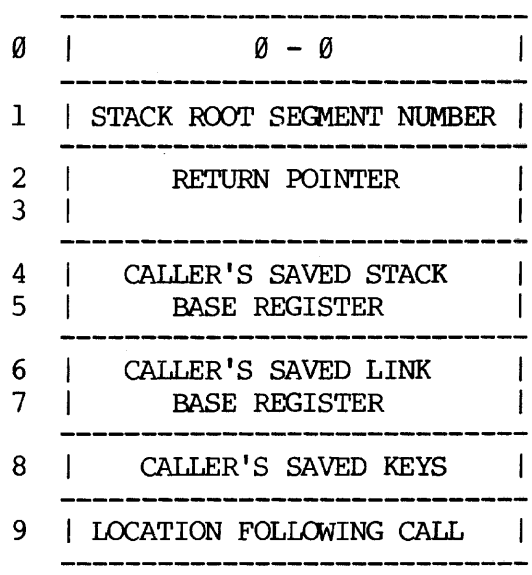
BITNO (Bits 33-36) - The bit number portion of the effective address.

Stack Segment Header (V-Mode)



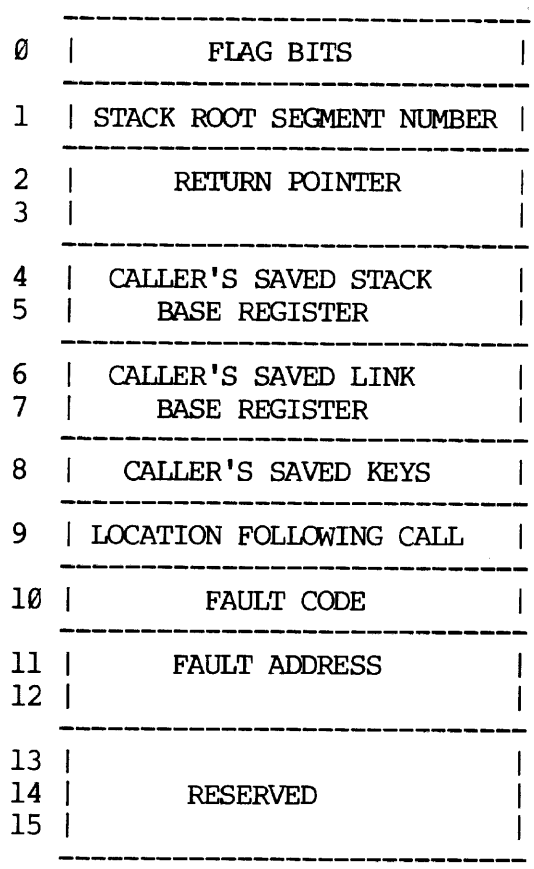
Word	Meaning
Ø,1	Free pointer - segment number/word number of available location at which to build next frame. Must be even.
2,3	Extension segment pointer - segment number/word number of locations at which to build next frame when current segment overflow. If zero, a stack overflow fault occurs when current segment overflows.

PCL Stack Frame Header (V-Mode)



Word	Meaning
0	Flag bits - set to zero by PCL 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 link base register
8	Caller's saved keys
9	Word number of location following call - beginning of argument transfer templates, if any

CALF Stack Frame Header (V-Mode)

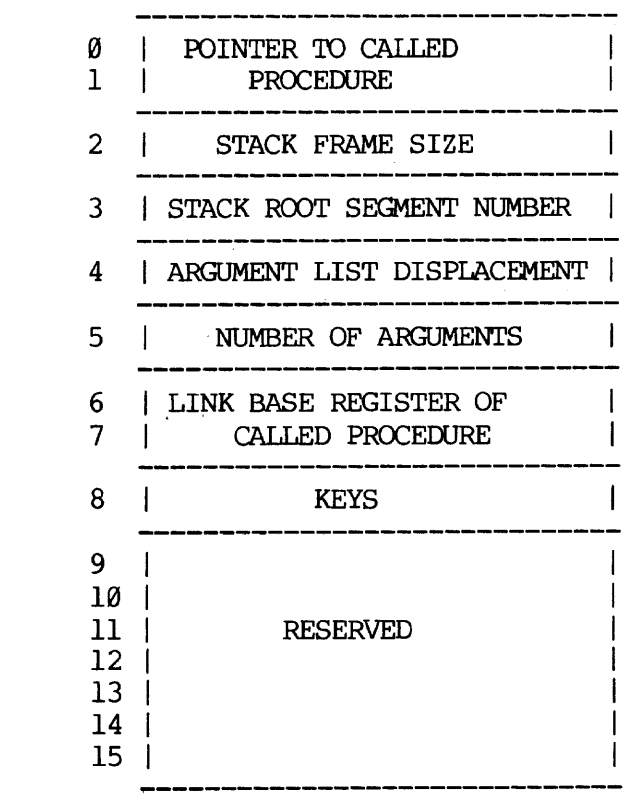


Word	Meaning
0	Flag bits - set to one by CALF fault
1	Stack root segment number - for locating free pointer
2,3	Return pointer - segment number/word number of location following call and argument sequence which created this frame
4,5	Caller's saved stack base register
6,7	Caller's saved link base register
8	Caller's saved keys
9	Word number of location following call - beginning of argument transfer templates, if any
10	Fault code

11,12 Fault address

13-15 Reserved

Entry Control Block (V-Mode)

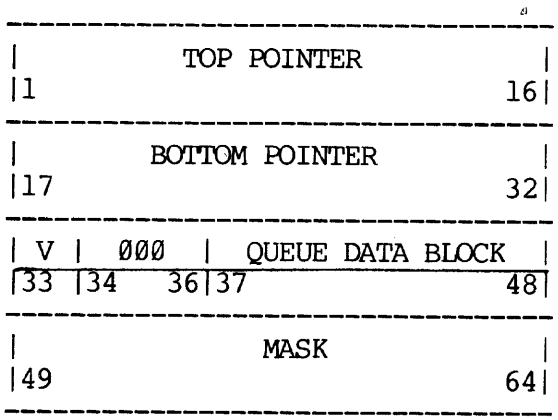


Word	Meaning
0,1	Pointer (ring, segment, word number) to the first executable instruction of the called procedure.
2	Stack frame size to create (in words). Must be even.
3	Stack root segment number. If zero, keep same stack.
4	Displacement in new frame of where to build argument list.
5	Number of arguments expected.
6,7	Called procedure's link base (location of called procedure's linkage frame less '400).
8	CPU keys desired by called procedure.

9-15 Reserved, must be zero.

Entry control blocks which are gates must begin on a Ø modulo 16 boundary, and must specify a new stack root.

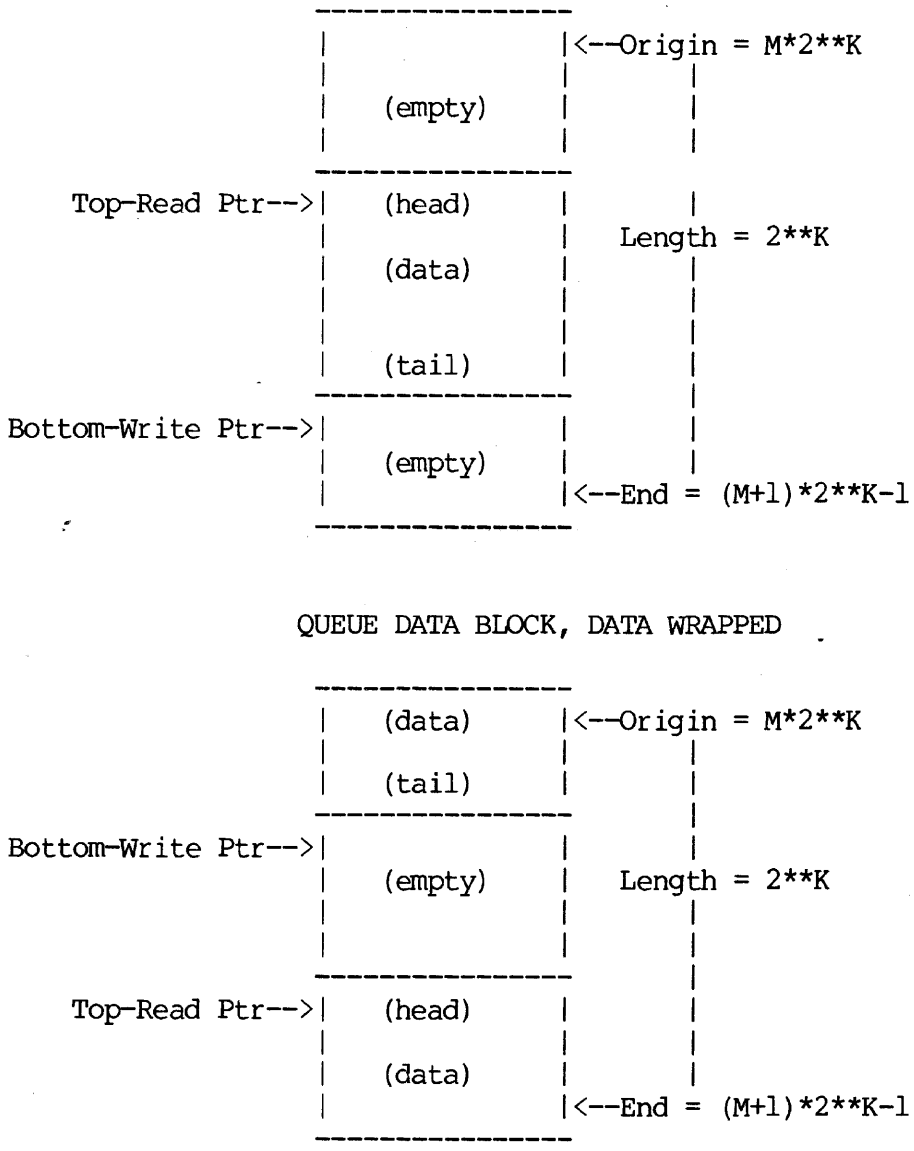
Queue Control Block (V-Mode)



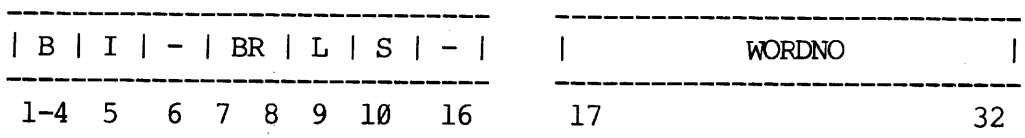
Bits	Meaning
1-16	Top pointer-read
17-32	Bottom Pointer-Write
33 (V)	Virtual/physical control bit Ø = physical queue 1 = virtual queue
34-36	Reserved - must be zero
37-48	Queue data block address Segment number if virtual queue High order physical address bits if physical queue
49-64	Mask - value 2**K-1

Queue control blocks must start on even word boundaries.

Figure 5-1. Queue Data Structures



Argument Transfer Template (V-Mode)



B (Bits 1-4) - Bit number

I (Bit 5) - Indirect

BR (Bits 7-8) - Base register

ØØ = Procedure base (PB)
Ø1 = Stack base (SB)
1Ø = Link base (LB)
11 = Temporary base (XB)

L (Bit 9) - Last template for this call

S (Bit 1Ø) - Store argument address. Last template for this argument.

WORDNO (Bits 17-32) - Word number offset from base register

PROCESSOR CHARACTERISTICS

Registers (S-Mode)

Prime 100, 200 and 300 registers are 16 bits wide. All the program visible registers are physically located in high speed memory and are addressed as memory locations 0-37. In restricted mode (normal user operation) only 0-7 are accessable.

Memory Address	Register Designation	Function
0	X	Index Register
1	A	Arithmetic Register
2	B	Extension Arithmetic Register
3		
4		
5		
6	VSC	Visible Shift Count
7	P	Program Counter
10	PMAR (Prime 300 only)	Page Map Address Register
11	FCODE	
12	FAR	Fault Address Register
13-17	Reserved	
20-37	DMA 1-8	Word Pairs for DMA channels (address and word counts)

Registers (R-Mode)

Prime 100, 200 and 300 registers are 16 bits wide. All the program visible registers are physicaly located in high speed memory and are addressed as memory locations 0-37. In restricted mode (normal user operation) only 0-7 are accessable.

Memory Address	Register Designation	Function
0	X	Index Register
1	A	Arithmetic Register
2	B	Extension Arithmetic Register
3	S	Stack Register
4	FLTH	Floating Point Accumulator - High
5	FLTL	Floating Point Accumulator - Low
6	FEXP	Floating Point Exponent
7	P	Program Counter
10	PMAR (Prime 300 only)	Page Map Address Register
11	FCODE	Fault code
12	PFAR	Page Fault Address Register
13-17	Reserved for microprogram	
20-37	DMA 1-8	Word Pairs for DMA channels (address and word counts)

Registers (V-Mode)

Prime 400/500 registers are 32 bits wide. See Section 2. Short form instructions reference the same registers as in Rmode.

All other instructions use the LDLR and STLR relative register addresses.

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

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.

V-Mode Register Description:

SCRATCH RSØ			DMX RS1			CURRENT REGISTER SET (CRS) RS2	RS3		
ADR	HIGH	LOW	ADR	HIGH	LOW	ADR	ADR	HIGH	LOW
Ø	TRØ	-	4Ø	-	-	1ØØ	14Ø	GRØ:OLT2	-
1	TR1	-	41	-	-	1Ø1	141	GR1:PTS	-
2	TR2	-	42	-	-	1Ø2	142	GR2(1,A,LH)	(2,B,LL)
3	TR3	-	43	-	-	1Ø3	143	GR3(EH)	(EL)
4	TR4	-	44	-	-	1Ø4	144	GR4	-
5	TR5	-	45	-	-	1Ø5	145	GR5(3,S,Y)	-
6	TR6	-	46	-	-	1Ø6	146	GR6	-
7	TR7	-	47	-	-	1Ø7	147	GR7(Ø,X)	-
1Ø	RDMX1	-	5Ø	-	-	11Ø	15Ø	FAR1(13)	-
11	RDMX2	-	51	-	-	111	151	FLR1	-
12	-	RATMPL	52	-	-	112	152	FAR2(4)	(5)
13	RSGT1	-	53	-	-	113	153	FLR2:VSC(6)	-
14	RSGT2	-	54	-	-	114	154	PB	-
15	RECC1	-	55	-	-	115	155	SB(14)	(15)
16	RECC2	-	56	-	-	116	156	LB(16)	(17)
17	-	REOIV	57	-	-	117	157	XB	-
2Ø	ZERO	ONE	6Ø	(2Ø)	(21)	12Ø	16Ø	DTAR3(1Ø)	-
21	PBSAVE	-	61	-	-	121	161	DTAR2	-
22	RDMX3	-	62	(22)	(23)	122	162	DTAR1	1
23	RDMX4	-	63	-	-	123	163	DTARØ	-
24	C377	-	64	(24)	(25)	124	164	KEYS	(MODALS)
25	-	-	65	-	-	125	165	OWNER	-
26	-	-	66	(26)	(27)	126	166	FCODE(11)	-
27	-	-	67	-	-	127	167	FADDR	(12)
3Ø	PSWPB	-	7Ø	(3Ø)	(31)	13Ø	17Ø	TIMER	-
31	PSWKEYS	1	71	-	-	131	171	-	-
32	PPA:PLA	PCBA	72	(32)	(33)	132	172	-	-
33	PPB:PLB	PCBB	73	-	-	133	173	-	-
34	DSWRMA	-	74	(34)	(35)	134	174	-	-
35	DSWSTAT	-	75	-	-	135	175	-	-
36	DSWPB	-	76	(36)	(37)	136	176	-	-
37	RSAVPTR	-	77	-	-	137	177	-	-

NOTICE - Numbers in parentheses ( ) show P3ØØ Address Mapping

Definitions

TR	Temporary Registers
	TR7 - Saved return pointer on a crash (automatic save)
RDMX	Register DMX
	RDMX1 - Used by DMC, buffer start pointer
	RDMX2 - REA at time of DMX trap
	RDMX3 - Save RD during DMQ
	RDMX4 - Used as working register
RATMPL	Read Address Trap Map to rP Low
RSGT	Register Segmentation Trap
	RSGT1 - SDW2 / address of Page Map
	RSGT2 - contents of Page Map / SDW2

REOIV	Register End of Instruction Vector
ZERO/ONE	Constants
PBSAVE	Procedure Base SAVE saved return pointer when return pointer used elsewhere
C377	Constant
PSWPB	Processor Status Word Procedure Base return pointer for interupt return (also used for Prime 300 compatibility)
PSWKEYS	Processor Status Word KEYS KEYS for interupt return (also used for Prime 300 compatibility)
PPA	Pointer to Process A
PLA	Pointer to Level A
PCBA	Process Control Block A
PPB	Pointer to Process B
PLA	Pointer to Level B
PCBB	Process Control Block B
DSWRMA	Diagnostic Status Word RMA RMA at last Check Trap
DSWSTAT	Diagnostic Status Word STATus
DSWPB	Diagnostic Status Word Procedure Base Return pointer or PBSAVE at last check
RSAVPTR	Register SAVE Pointer Location of Register Save Area after Halt

GR	General Register
OLT2	Old Length and Type
PTS	Pointer To Sign
FAR1	Field Address Register 1
FLR1	Field Length Register 1
FAR2	Field Address Register 2
FLR2	Field Length Register 2
PB	Procedure Base PBH - RPH PBL - 0
SB	Stack Base
LB	Link Base
XB	Temporary (auxiliary) base
DTAR	Descriptor Table address registers
KEYS	See below
MODALS	See below
OWNER	Pointer to PCB of process owning this register set
FCODE	Fault CODE
FADDR	Fault ADDRess
TIMER	1-millisecond process timer (used for time-slice)

V-Mode Register Usage:

STLR/ LDLR	Address Trap	Usage
-	7	P (program counter)
2 H	1	A (accumulator, high half of L)
2 L	2	B (double-precision, low half of L)
3 H,L	-	EH,EL (accumulator extension for MPL DVL)
5 H	3	Y (alternate index), S (stack)
7 H	0	X (index)
10 H	13	-
10,11	-	(field address and length register 0)
12,13	-	(field address and length register 1)
12 H	4	(floating accumulator, mantissa high)
12 L	5	(mantissa middle)
13 H	6	(exponent)
13 L	-	(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)
20 H	10	(high half of DTAR3)
20 H,L	-	DTAR3 (descriptor table address, segments 3072-4095)
21 H,L	-	DTAR2 (segments 2048-3071)
22 H,L	-	DTAR1 (segments 1024-2047)
23 H,L	-	DTAR0 (segments 0-1023)
24 H,L	-	keys, modals
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)
30 H	-	process 1024-microsecond c.p.u timer

Floating Point Register - Single Precision (R-Mode)

Register	Contents	
'04	|S| MANTISSA	
	1 2	16
'05	MANTISSA	
	17	32
'06	EXPONENT (EXCESS 128)	

Floating Point Register - Double Precision (R-Mode)

Register	Contents	
'04	|S| MANTISSA	
	1 2	16
'05	MANTISSA	
	17	32
'02	MANTISSA	
	33	48
'06	EXPONENT (EXCESS 128)	
	49	64

Floating Point Register - Single Precision (V-Mode)

Register	Contents		
12H	|S|	MANTISSA	
	1 2		16
12L	|	MANTISSA	|
	17		32
13H	|	EXPONENT (EXCESS 128)	|
	33		48

Floating Point Register - Double Precision (V-Mode)

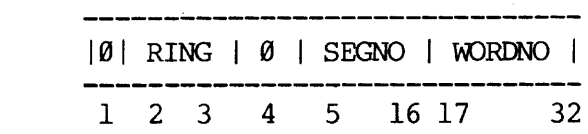
Register	Contents		
12H	|S|	MANTISSA	
	1 2		16
12L	|	MANTISSA	|
	17		32
13L	|	MANTISSA	|
	33		48
13H	|	EXPONENT (EXCESS 128)	|
	49		64

Base Registers (V-Mode)

The four base registers:

Procedure Base Register	PB
Stack Base Register	SB
Link Base Register	LB
Temporary Base Register	XB

are discussed in Section 2, Prime 400 Architecture. Their format is:



RING (Bits 2-3) - Ring Number

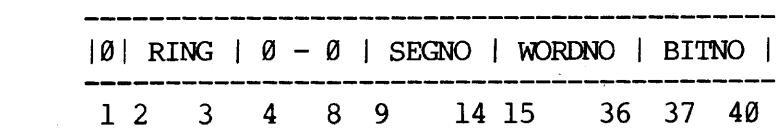
SEGNO (Bits 5-16) - Segment Number

WORDNO (Bits 17-32) - Word Number

Field Registers (V-Mode)

There are two address registers and two length register for the manipulation of variable length fields. They overlap the floating point accumulator.

Field Address Register



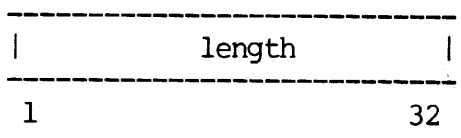
RING (Bits 2-3) - Ring Number

SEGNO (Bits 9-14) - Segment Number

WORDNO (Bits 15-36) - Word Number

BITNO (Bits 37-40) - Bit Number

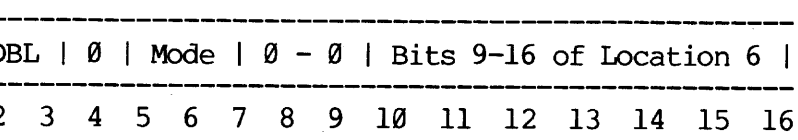
Field Length Register



The meaning of the value in the field length register depends on the data type being used. For a discussion of the available data types see the decimal and character instruction descriptions.

Keys (S,R-Modes)

Process status information is available in a word called the keys, which can be read or set by the program. If format is as follows:



C (Bit 1) - Set by arithmetic error conditions

DBL (Bit 2) - Single Precision, 1 - Double Precision.

MODE (Bits 4-6) - The current addressing mode as follows:

000 = 16S

001 = 32S

011 = 32R

010 = 64R

110 = 64V

100 = 32I

C-Bit (S,R-Modes)

Bit 1 in the keys. Set by arithmetic error conditions (Bit 1).

Keys (V-Mode)

Process status information is collected in a 16-bit register known as the keys. It may be referenced by the LPSW, TKA, and TAK instructions.

C	Ø	L	MODE	F	X	LT	EQ	DEX	Ø - Ø	I	S
1	2	3	4-6	7	8	9	10	11	12 - 14	15	16

C (Bit 1) - C-Bit

L (Bit 3) - L-Bit

MODE (Bits 4-6) - Addressing Mode:

ØØØ = 16S
ØØ1 = 32S
Ø11 = 32R
Ø1Ø = 64R
11Ø = 64V
1ØØ = 32I

F (Bit 7) - Floating point exception disable:

Ø = take fault
1 = set C-bit

X (Bit 8) - Integer Exception enable

Ø = set C-bit
1 = take fault

LT (Bit 9) - Condition code bits:
EQ (Bit 1Ø)

LT = negative

EQ (Bit 1Ø)

DEX (Bit 11) - Decimal exception enable

Ø = set C-bit
1 = take fault

I (Bit 15) - In dispatcher - set/cleared only by process exchange

S (Bit 16) - Save done - set/cleared only by process exchange

C-Bit (V-Mode)

Set by error conditions in arithmetic operations.

L-Bit (V-Mode)

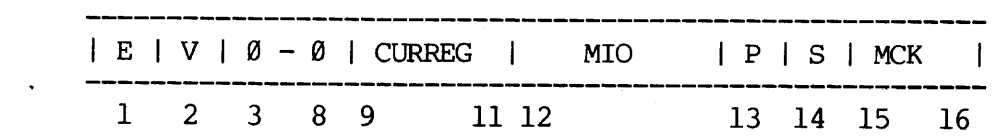
Set by an arithmetic or shift operation except IRS, IRX, DRX. Equal to carry out of the most significant bit (bit 1) of an arithmetic operation. It is valuable for simulating multiple - precision operations and for performing unsigned comparisons following a CAS or a SUB.

Condition Code Bits (V-Mode)

The two condition-code bits are designated "EQ" and "LT". EQ is set if and only if the result is zero; 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.

Modals (V-Mode)

Processor status is collected in another 16-bit register known as the "modals".



E (Bit 1) - Interrupts enabled

V (Bit 2) - Vectored-interrupt mode

CURREG (Bits 9-11) - Current register set (set/cleared only by process exchange)

MIO (Bit 12) - Mapped I/O mode

P (Bit 13) - Process-exchange mode

S (Bit 14) - Segmentation mode

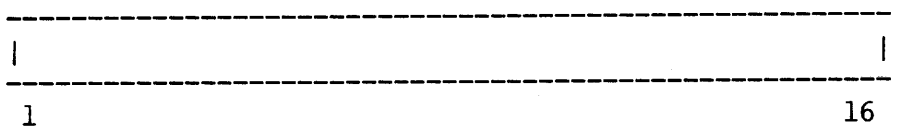
MCK (Bits 15-16) - Machine-check mode

Note

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, the keys operations OTK and TAK, or the various special-case instructions designed to manipluate specific bits of the status. Furthermore, even LPSW should not be used to alter the in-dispatcher and save-done bits of the keys or the register-set bits of the modals.

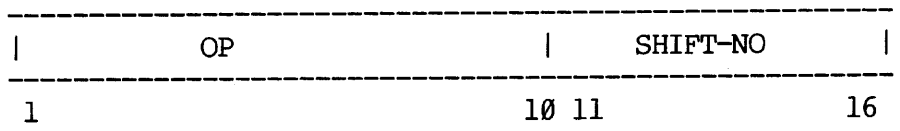
INSTRUCTION FORMATS

GENERIC



The entire instruction word is an opcode. Bits 3-6 are always zero

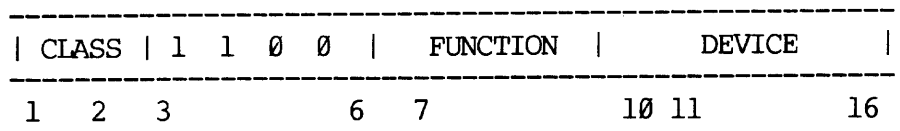
SHIFT



OP (Bits 1-10) - Opcode - Bits 3-6 are always zero

SHIFT-NO (Bits 11-16) - Two's complement of the number of places to be shifted.

I/O (S,R-Modes)



CLASS (Bits 1-2) - Type of I/O instruction

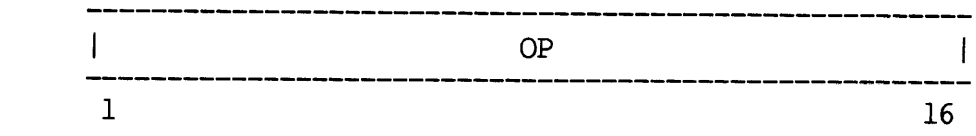
00 = Control
01 = Sense
10 = Input
11 = Output

Bits 3-6 - 1100

FUNCTION (Bits 7-10) - Subdivision of class. Device dependent

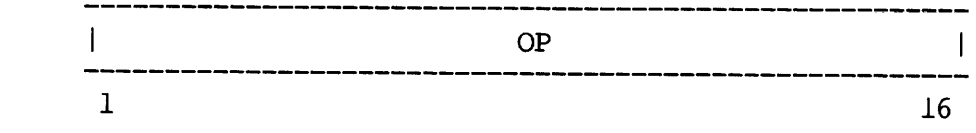
DEVICE (Bits 11-16) - Device type

DECIMAL (V-Mode)



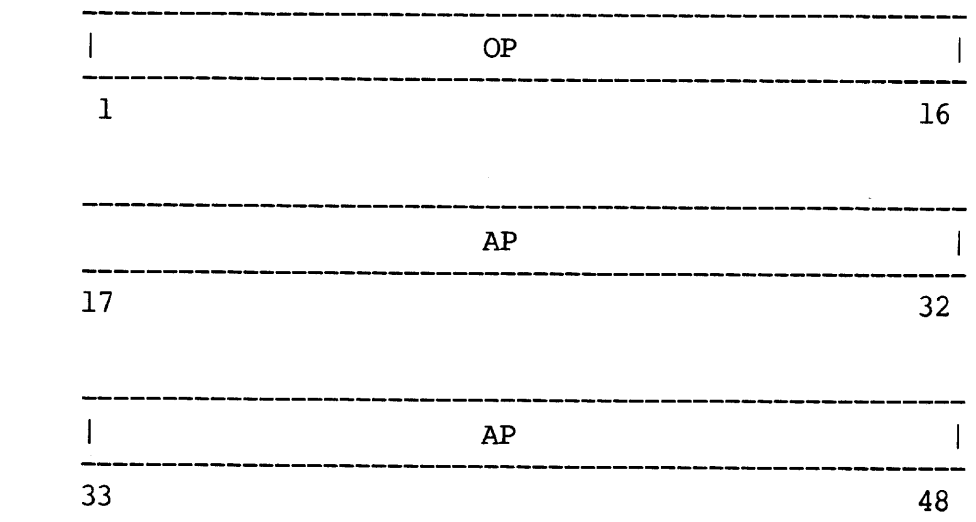
OP (Bits 1-16) - Opcode. This instruction uses previously set up field registers and a previously set up control word in register L (see decimal control word in Data Structures).

CHARACTER (V-Mode)



OP (Bits 1-16) - Opcode. This instruction uses previously set up field registers.

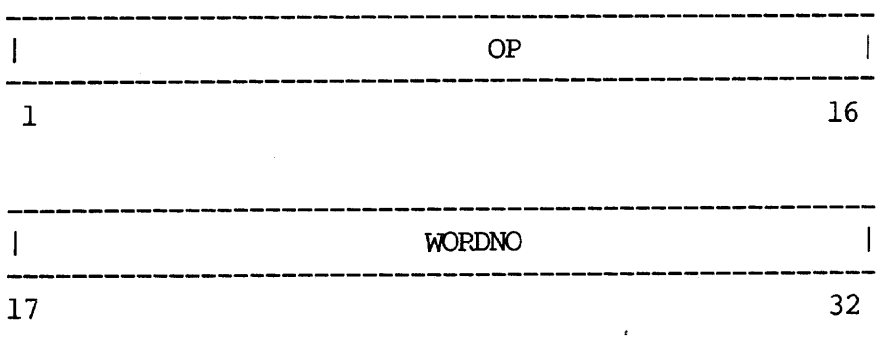
GENERIC AP (V-Mode)



OP (Bits 1-16) - Opcode

AP Bits (17-48) - Address Pointer - see AP in Data Structures

BRANCH (V-Mode)



OP (Bits 1-16) - Opcode

WORDNO (Bits 17-32) - Word number offset from procedure base register.

Memory Reference

See Effective Addressing Formation in Section 6 - Memory Addressing.

SECTION 6

MEMORY ADDRESSING

BACKGROUND CONCEPTS

Memory is addressed as a set of continuous word locations. The number of words that can be addressed by an instruction, and the way in which the address is calculated depends on the current addressing mode of the machine and the location of the address relative to the instruction.

In turn, the addressing modes of the machine differ in the size of the instruction word, the number of bits allotted to the provisional address displacement, and the number and meaning of the bits allotted to the operation code.

To reduce the number of memory references, designers wish to do as much as possible in one word. For example, in the S and R addressing modes, a one word memory reference instruction has nine bits (512 words) of direct addressability, four bits for operation codes, one bit for indirection, one bit for indexing, and one bit to control out-of-range addresses.

Within each addressing mode, there are the following tradeoffs:

1) Size of program address space

2) Levels of indirection

3) Levels of indexing

4) Whether indexing is performed before or after indirection

5) Number of operation codes available.

Through the discussion of the S, R, and V addressing modes, we shall show how these variables are defined.

Memory Organization

Sectors: (S-Mode and R-Mode when S=0). A sector is a contiguous group of 512 words. S-Mode memory reference instructions have nine bits (D field) of addressability to any location in a sector and one bit, the S-bit, to specify Sector 0 (S=0) or the current sector (S=1). D and S together give 10 bits, or 1024 words, of direct addressability.

Relative Reach: (R-Mode and V-Mode when S=1). When S=1 the D field is interpreted as a signed number in the range -255 to +256. When D≤240 (R-Mode) or D≤224 (V-Mode) the number is treated as a code, not as a displacement. When -240<D<256 (R-Mode) or -224<D<256 (V-Mode), the address is relative to the program counter.

Segmentation: (V-Mode and I-Mode). See Sections 2 and 3 - Prime 400 and Prime 500 architecture for a discussion of segmentation.

Effective Address Formation

Each memory reference instruction calculates an effective address. This calculation and its results vary depending on addressing mode and and instruction format; variables include pre and post indexing, indirection, and base registers. For maximum clarity, we discuss the classes by format types and present addressing mode flowcharts. Both the format discussions and the addressing mode flowcharts are cross referenced to each other. Table 6-1 summarizes the format classes and gives the addressing modes where they are used.

Indexing: In general, if the X-bit of the instruction is set, the contents of the index register are added to the D-field. If the indirect bit is set, the address mode and D-field determine whether indexing occurs before or after indirection. The result is truncated to the number of bits permitted by the addressing modes, and the high order bits are cleared. In V-Mode, there are two index registers, X and Y. The displacement field determines which to use and how to use it.

Note

The index register may be preset by the program to any value between -32768 and +32767.

Indirection: In general, if the I-bit is set, the D-field plus index, if any, is an intermediate address. The indirect address word at that location may, depending on the address mode, also contain X and I bits. The specific addressing mode discussion gives the details.

Address Truncation (SR): After effective address formation is complete, the resulting address is truncated to the number of bits appropriate to the addressing mode in effect:

Table 6-1. Memory Reference Instruction Format

Type	No. Words	(1) S	(2) D	(3) CB	Mode
Basic	1	0	0 - '777	--	SR
Sector Relative	1	1	0 - '777	--	S
Procedure Relative	1	1	-241 to +256	--	R
			-224 to +256	--	V
Stack Postincrement/ Predecrement	1	1	-256 to -241	2,3	R
Base Register Relative	1	0	0 - '777	--	V
Long Reach	2	1	-256 to -241	0,2	R
Stack Relative	2	1	-256 to -241	1,3	R
Base Registers	2	1	-256 to -224	--	V

(1) S - Sector Bit. Bit 7 in both one and two word memory reference instructions. The meaning varies, depending on the addressing mode, but in general is used to control out of range addresses.

(2) D - Displacement field. Bits 8-16 in the instruction word. Bit 8 is sign bit except in Basic, Sector Relative, and Base Register types of instruction.

(3) CB - Class Bits. Bits 15 and 16 of the R mode two word instructions distinguish between Long Reach and Stack Relative instruction types.

Mode	Addressing Bits	Size of Addressable Memory
16S	14	16K
32S	15	32K
32R		
64R	16	64K

Since the higher order bits of the address are zeroes, an address cannot be formed that addresses a memory location beyond the range of the current addressing mode. However, it is possible for an executing program to increment the program counter out of the current range (instead of overflowing to zero).

Instruction Range

The range that an instruction can directly address is called its addressing range. The assembler and the loader analyze the assembler statement and set up both in-range and out-of-range addresses. In the discussion below we shall examine the sectored and relative address ranges and how they are set up prior to execution. The segmentation concepts and address ranges are discussed in Section 2.

Sectored: In S-mode, the memory reference instructions can address any location in sector 0 or in the sector of the instruction. When S=1, the nine bit displacement field is a location in the current sector. When S=0, the nine bit displacement is in sector 0.

The software uses the S-bit to control out-of-range addresses in the following manner: the assembler does a preliminary analysis of the relation of the displacement field (expression or symbol) to the instruction location, and passes this information to the loader, which sets up the final instruction for execution. The loader puts the object code received from the assembler together with any other required routines (such as subroutines), resolves external linkages and sets up sector 0, the communication and linkage area.

Sector 0 can also be directly addressed by the program, a useful feature for handling common data fields.

Examples:

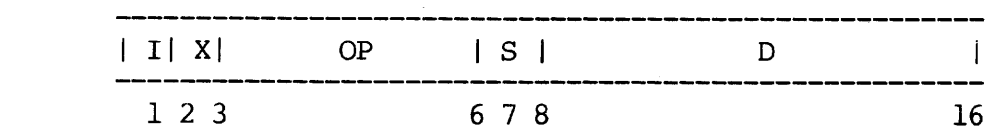
ASSEMBLER NOTATION	LOCATION OF ADDR		
	SECTOR Ø	SAME SECTOR	OTHER
LDA ADDR	S=Ø I=Ø D=location in sector Ø	S=1 I=Ø D=displacement in same sector	S=Ø I=1 D=first available link in sector Ø. At that location an indirect word is constructed with I=Ø, pointing at ADDR with a full 14 (16S) or 15 (32S, 32R) or 16 (64R) bit indirect address
LDA ADDR,*	S=Ø I=1 D=location in sector Ø which contains a pointer defined by the program	S=1 I=1 D=location in same sector. It must contain pointer defined by program.	S=Ø I=1 D=first available link in sector Ø. At that location an indirect word is constructed with I=1 and a full 14 (16S) or 15 (32S, 32SR) bit indirect pointer to ADDR. Not permitted in 64R.

Relative: In R-mode when S=1, the D field is interpreted as a signed number in the range -225 to +256. When the two high order bits are one (D<240) the number is treated as a code, not as a displacement. When -240<D<256 the address is relative to the program counter.

The loader analyzes the displacement field and if the effective address will be out of relative range (-256 -> +256) sets S=0, I=1, and the displacement field to point to the address word in sector zero. Thus, in 64R, if the address is is out of range, no indirection is possible because the loader uses the instruction word indirect bit.

MEMORY REFERENCE INSTRUCTION FORMATS

BASIC (One word, S-bit=Ø) 16S
32S
32R
64R



I (Bit 1) - Indirect Bit

X (Bit 2) - Index Bit

OP (Bits 3-6) - Opcode

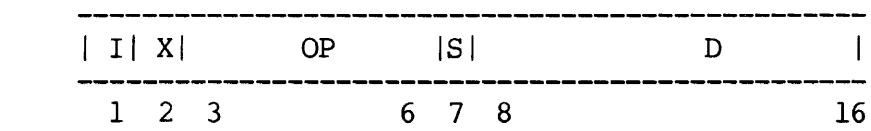
S (Bit 7) - Sector Bit =Ø

D (Bits 8-16) - Displacement in sector Ø

The D-field is a displacement in sector Ø. The effective address is equal to bits 8-16 of the instruction, with bits Ø-7 equal to zero. Indexing and indirection are a function of the I and X bits and the addressing mode.

Addresing Mode	I	X	S	D	EA	Type
16S	Ø	Ø	Ø	Ø to '777	Ø|D	Direct
	Ø	1	Ø	Ø to '777	Ø|D+X	Indexed
	1	Ø	Ø	Ø to '777	I(Ø|D)	Indirect
	1	1	Ø	Ø to '777	I(Ø|D+X)	Indirect, preindexed
32S	Ø	Ø	Ø	Ø to '777	Ø|D	Direct
32R	Ø	1	Ø	Ø to '777	Ø|D+X	Indexed
64R	1	Ø	Ø	Ø to '777	I(Ø|D)	Indirect
	1	1	Ø	Ø to '77	I(Ø|D+X)	Indirect, preindexed
	1	1	Ø	1ØØ to '777	I(Ø|D)+X	Indirect, postindexed

SECTOR RELATIVE (One word, S-bit=1) 16S
32S



I (Bit 1) - Indirect Bit

X (Bit 2) - Index Bit

OP (Bits 3-6) - Opcode

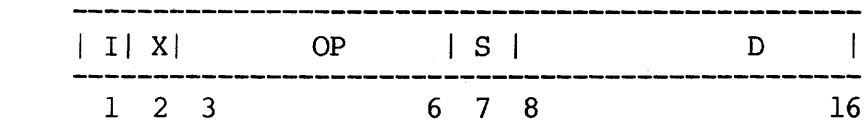
S (Bit 7) - Sector Bit = 1

D (Bits 8-16) - Displacement within current sector.

The D-field is a displacement in the current sector. The effective address is formed by concatinating the D-field bits with the higher order bits of the program counter (P). Indexing and indirection are a function of the I and X bits and the addressing mode. Bits 1 and 2 (16S) or 1 (32S) of the final effective address are cleared. In effect, the program counter gives the sector number and the D-field, the location within the sector.

Addressing Mode	I	X	S	D	EA	Type
16S	Ø	Ø	1	Ø to '777	P|D	Direct
	Ø	1	1	Ø to '777	P|D+X	Indexed
	1	Ø	1	Ø to '777	I(P|D)	Indirect
	1	1	1	Ø to '777	I(P|D+X)	Indirect, preindexed
32S	Ø	Ø	1	Ø to '777	P|D	Direct
	Ø	1	1	Ø to '777	P|D+X	Indexed
	1	Ø	1	Ø to '777	I(P|D)	Indirect
	1	1	1	Ø to '777	I(P|D)+X	Indirect, postindexed

PROCEDURE RELATIVE (One word, S-bit=1) 32R
64R
64V



I (Bit 1) - Indirect Bit

X (Bit 2) - Index Bit

OP (Bits 3-6) - Opcode

S (Bit 7) - Sector Bit =1

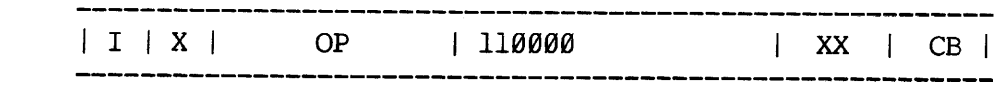
D (Bits 8-16) - Location relative to the program counter

64V = -224 to +256
64R = -240 to +256

Addressing is relative to the current program counter value, which is the current instruction location plus 1. The effective address is formed by adding the value of the D-field to the updated program counter value (P). Indirection and indexing are a function of the I and X bits and the addressing mode.

Addressing Mode	I	X	S	D	EA	Type
32R 64R	0	0	1	-240 to +256	P+D	Direct
	0	1	1	-240 to +256	P+D+X	Indexed
	1	0	1	-240 to +256	I(P+D)	Indirect
	1	1	1	-240 to +256	I(P+D)+X	Indirect, postindexed
64V	0	0	1	-224 to +256	P+D	Direct
	0	1	1	-224 to +256	P+D+X	Indexed
	1	0	1	-224 to +256	I(P+D)	Indirect
	1	1	1	-224 to +256	I(P+D)+X	Indirect, postindexed

STACK PREDECREMENT, POSTINCREMENT (One word, S-bit=1) 32R 64R



I (Bit 1) - Indirect Bit

X (Bit 2) - Index Bit

OP (Bits 3-6) - Opcode

Bits 7-12 = 110000

XX (Bits 13-14) - Opcode extension

CB (Bits 15-16) - Class Bits

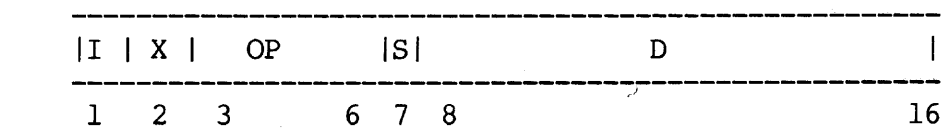
These classes use the stack pointer (SP) as the address displacement, and perform an auxiliary postincrement or predecrement of the pointer. Instructions using these address methods are always one-word instructions.

Addressing Mode	I	X	S	Class Bits 15,16	EA	Type
32R 64R	0	0	1	2	SP	Postincrement
	0	1	1	2	I(SP)+X	Postincrement, indirect, post-indexed
	1	0	1	2	I(SP)	Postincrement, indirect
	0	0	1	3	SP-1	Predecrement
	0	1	1	3	I(SP-1)+X	Predecrement indirect, post-indexed
	1	0	1	3	I(SP-1)	Predecrement, indirect

Note

If a fault occurs during the execution of these classes, anomolous behavior can result.

BASE REGISTER RELATIVE (One word, S-bit=0) 64V



I (Bit 1) - Indirect Bit

X (Bit 2) - Index Bit

OP (Bits 3-6) - Opcode

S (Bit 7) - Sector Bit = Ø

D (Bits 8-16) - Location relative to selected base register.

This format provides 64V with one word based memory reference instructions, using the D-field to encode both base and displacement.

All indirection will be through 16-bit pointers in the procedure segment and the final effective address of indirect instructions will be in the procedure segment.

The effective address calculation is:

I	X	S	D	Address	Type
Ø	Ø	1	Ø-'7	register location	Direct
			'1Ø-'377	SB+D	
			'4ØØ-'777	LB+D	
Ø	1	1	Ø-'377	if D+X<'1Ø then EA= register location else SB+D+X	Indexed
			'4ØØ-'777	LB+D+X	
1	Ø	1	Ø-'7	REG*	Indirect
			'1Ø-'777	PB|D*	
1	1	1	Ø-'77	[PB|D+X]*	Indirect preindexed
1	1	1	'1ØØ-'777	[PB|D]*+X	Indirect postindexed

PB = procedure base register

LB = link base register

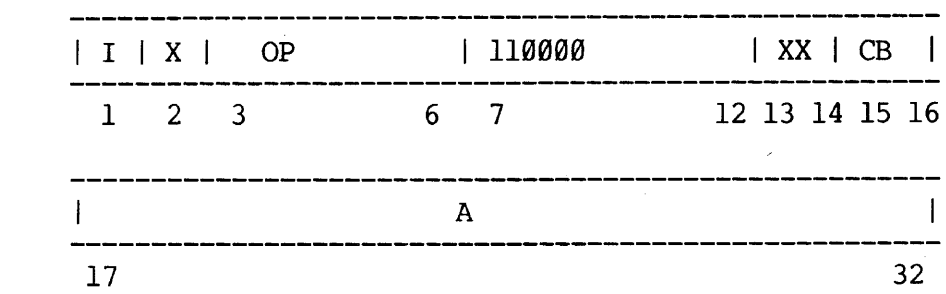
SB = stack base register

X = Index register

D = Displacement field

REG = R-Mode registers, i.e., A,B,X, etc.

LONG REACH (Two word, S-bit=1) 32R
64R



I (Bit 1) - Indirect Bit

X (Bit 2) - Index Bit

OP (Bits 3-6) - Opcode

Bits 7-12 - 110000

XX (Bits 13-14) - Opcode extension

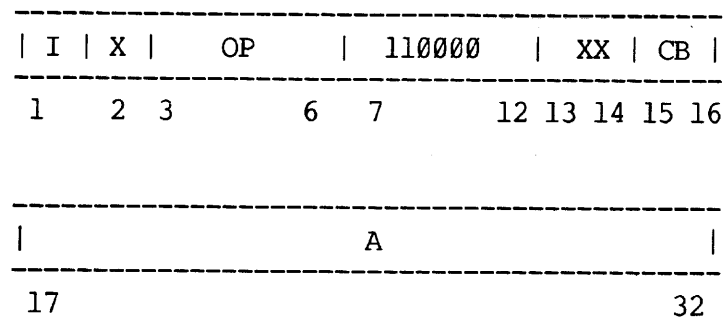
CB (Bits 15-16) - Class Bits

A (Bits 17-32) - Address word

The 16-bit address word in the location following the instruction plus the I and X bits in the instruction combine in effective address calculation. The direct instruction reach is extended to 32K words (32R) or 64K words (64R), since the address is in the word following the instruction. In 32R, bit 1 is zero. In 64R, all 16 bits are used.

Addressing Mode	I	X	S	CB	EA	Type
32R	0	0	1	0	A	Direct
64R	0	1	1	0	A+X	Indexed
	1	0	1	0	I(A)	Indirect
	1	1	1	0	I(A+X)	Indirect, preindexed
	1	1	1	2	I(A)+X	Indirect, postindexed

STACK RELATIVE (Two Word, S-bit=1) 32R 64R



I (Bit 1) - Indirect Bit

X (Bit 2) - Index Bit

OP (Bits 3-6) - Opcode

Bits 7-12 - 110000

XX (Bits 13-14) - Opcode extension

CB (Bits 15-16) - Class Bits

A (Bits 17-32) - Address word

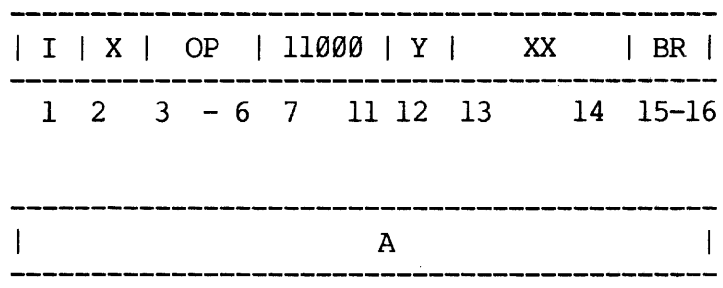
This class is identical to two-word long reach except that the contents of the stack pointer (SP) are added to the address word following the instruction word during the initial effective address calculation.

Indexing and indirection take place under control of the I and X bits and the addressing mode.

Addressing

Mode	I	X	S	CB	EA	Type
32R						
64R	0	0	1	1	A+SP	Direct
	0	1	1	1	A+SP+X	Indexed
	1	0	1	1	I(A+SP)	Indirect
	1	1	1	1	I(A+SP+X)	Indirect, preindexed
	1	1	1	3	I(A+SP)+X	Indirect, postindexed

TWO WORD MEMORY REFERENCE 64V



I (Bit 1) - Indirect bit

X (Bit 2) - X bit

OP (Bit 3-6) - Opcode

Y (Bit 12) - Y bit

XX (Bits 13 -14) - Opcode extension

BR (Bits 15-16) - Base register: 00=PB, 01=SB, 10=LB, 11=XB

A (Bits 17-32) - 16-bit word displacement relative to the base selected by the BR bits.

I, X, Y and BR combine to give all 32 possible address combinations:

- direct
- indexed by X
- indexed by Y
- indirect
- pre-indexed by X
- pre-indexed by Y
- postindexed by X
- postindexed by Y

All indirect words are either 32 or 48 bit format and the final effective address is always a memory address (never a register). Table 6-3 shows all possible combinations.

Table 6-3. V-Mode Two Word Memory Reference

I	X	Y	BR	Effective Address	Meaning
0	0	0	0	[PB]|D	
			1	[SB]+D	Direct
			2	[LB]+D	
			3	[XB]+D	
0	0	1	0	[PB]|D+[Y]	
			1	[SB]+D+[Y]	Indexed by Y
			2	[LB]+D+[Y]	
			3	[XB]+D+[Y]	
0	1	0	0	[PB]|D+[X]	
			1	[SB]+D+[X]	
			2	[LB]+D+[X]	Indexed by X
			3	[XB]+D+[X]	
0	1	1	0	[PB]|D*	
			1	[SB]+D*	Indirect
			2	[LB]+D*	
			3	[XB]+D*	
1	0	0	0	[PB|D+Y]*	
			1	[SB+D+Y]*	Pre-indexed by Y
			2	[LB+D+Y]*	
			3	[XB+D+Y]*	
1	0	1	0	[PB]|D*+[Y]	
			1	[SB]+D*+[Y]	Post-indexed by Y
			2	[LB]+D*+[Y]	
			3	[XB]+D*+[Y]	
1	1	0	0	[PB]|D+[X]*	
			1	[SB]+D+[X]*	Pre-indexed by X
			2	[LB]+D+[X]*	
			3	[XB]+D+[X]*	
1	1	1	0	[PB]|D*+[X]	
			1	[SB]+D*+[X]	Post-indexed by X
			2	[LB]+D*+[X]	
			3	[XB]+D*+[X]	

Note

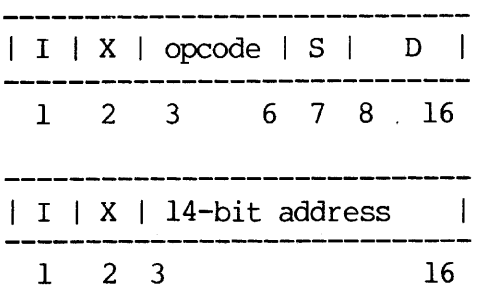
LDX and STX instructions may only be direct or indirect.

ADDRESSING MODE SUMMARIES AND FLOW CHARTS

16S SUMMARY

Address Length: 14 bits; 16K word address space

Format:



Indexing: Multiple levels. In an indirect word, the index calculation is done before the indirection.

Indirection: Multiple levels.

I	X	S	D	EA	Assembler Notation	Type
0	0	0	0 to '777	0|D	LDA ADDR	Direct
0	1	0	0 to '777	0|D+X	LDA ADDR,1	Indexed
1	0	0	0 to '777	I(0|D)	LDA ADDR,*	Indirect
1	1	0	0 to '777	I(0|D+X)	LDA ADDR,1*	Indirect, preindexed
0	0	1	0 to '777	P|D	LDA ADDR	Direct
0	1	1	0 to '777	P|D+X	LDA ADDR,1	Indexed
1	0	1	0 to '777	I(P|D)	LDA ADDR,*	Indirect
1	1	1	0 to '777	I(P|D+X)	LDA ADDR,1*	Indirect, preindexed

Table Description

P	= contents of program counter prior to instruction fetch (pointing at instruction)
Ø|D	= displacement into sector Ø. Sector bits of effective address (bits 3-8) are zero.
P|D	= displacement in current sector formed by concatenation of sector bits from program counter with displacement field in instruction word.
X	= contents of index register.
I(expression)	= treat the effective address as indirect address
ADDR	= location addressed by the LDA

Note

If D is Ø-'7 and S=Ø, the effective address is a register.

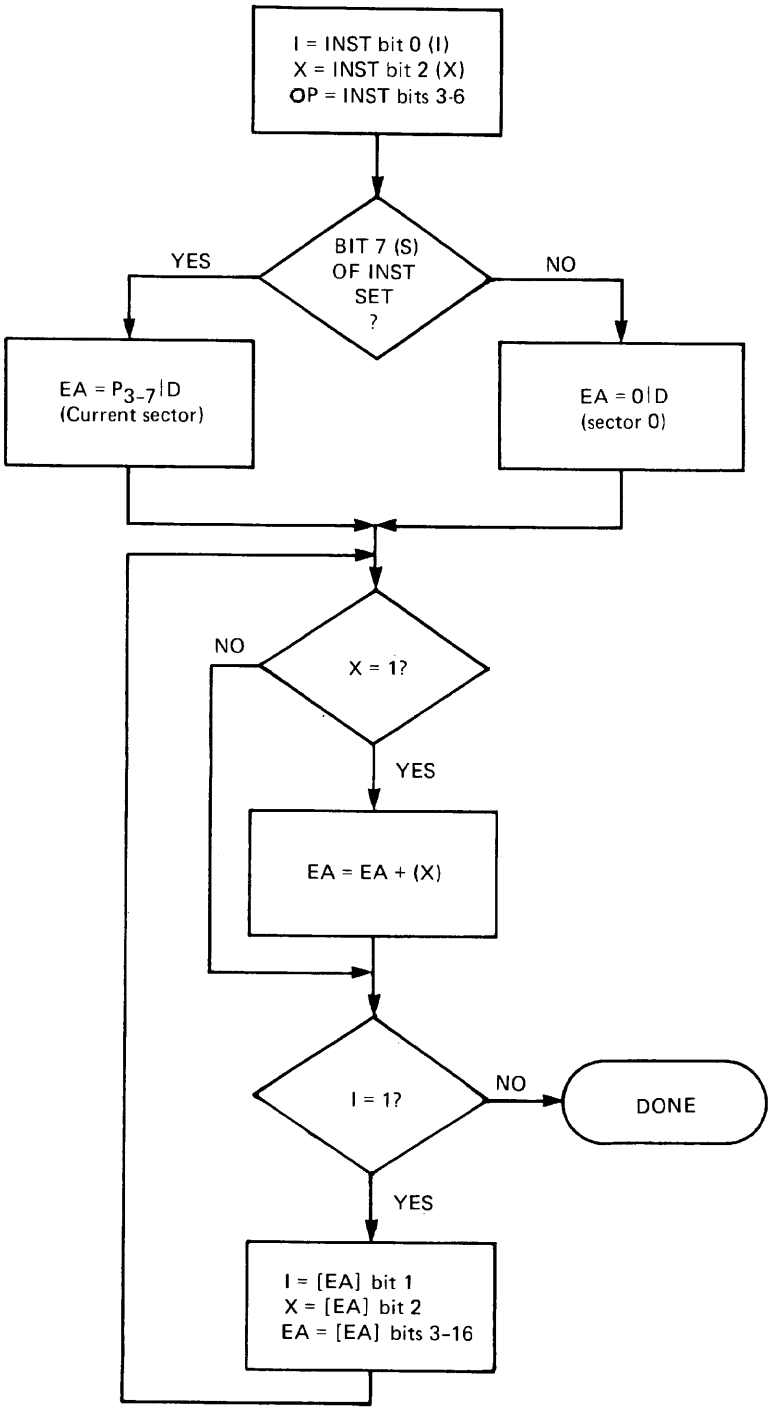
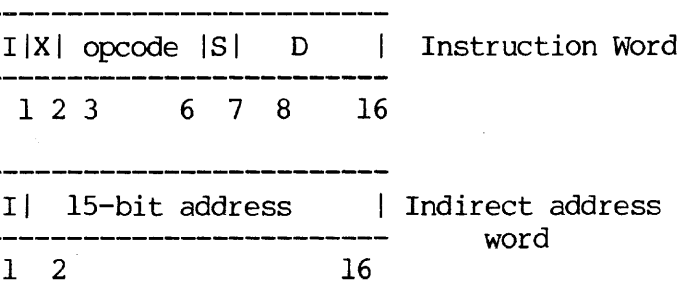


Figure 6-1. 16S Address Calculation

32S (INCLUDES 32R WHEN S=Ø) SUMMARY

Address Length:15 bits; 32K word address space

Format:



Indexing: One level. The 15-bit indirect address word eliminates the X bit. Done after all indirection is complete, except for the special case shown in the table below.

Indirection: Multiple levels.

I	X	S	D	EA	Assembler Notation	Type
Ø	Ø	Ø	Ø to '777	Ø|D	LDA ADDR	Direct
Ø	1	Ø	Ø to '777	Ø|D+X	LDA ADDR,1	Indexed
1	Ø	Ø	Ø to '777	I(Ø|D)	LDA ADDR,*	Indirect
1	1	Ø	Ø to '77	I(Ø|D+X)	LDA ADDR,1*	Indirect, preindexed
1	1	Ø	1ØØ to '777	I(Ø|D)+X	LDA ADDR,*1	Indirect postindexed
Ø	Ø	1	Ø to '777	P|D	LDA ADDR	Direct
Ø	1	1	Ø to '777	P|D+X	LDA ADDR,1	Indexed
1	Ø	1	Ø to '777	I(P|D)	LDA ADDR,*	Indirect
1	1	1	Ø to '777	I(P|D)+X	LDA ADDR,1*	Indirect postindexed

Table Description

P = contents of program counter prior to instruction fetch (pointing at instruction)

Ø|D = displacement into sector Ø. The contents of D when S=Ø.

P|D = displacement in current sector formed by concatenation of sector bits from program counter with D.

X = contents of index register.

I(expression) = treat the effective address as indirect address.

ADDR = location addressed by the LDA.

Note

If D is 0-'7 and S=0, the effective address is a register.

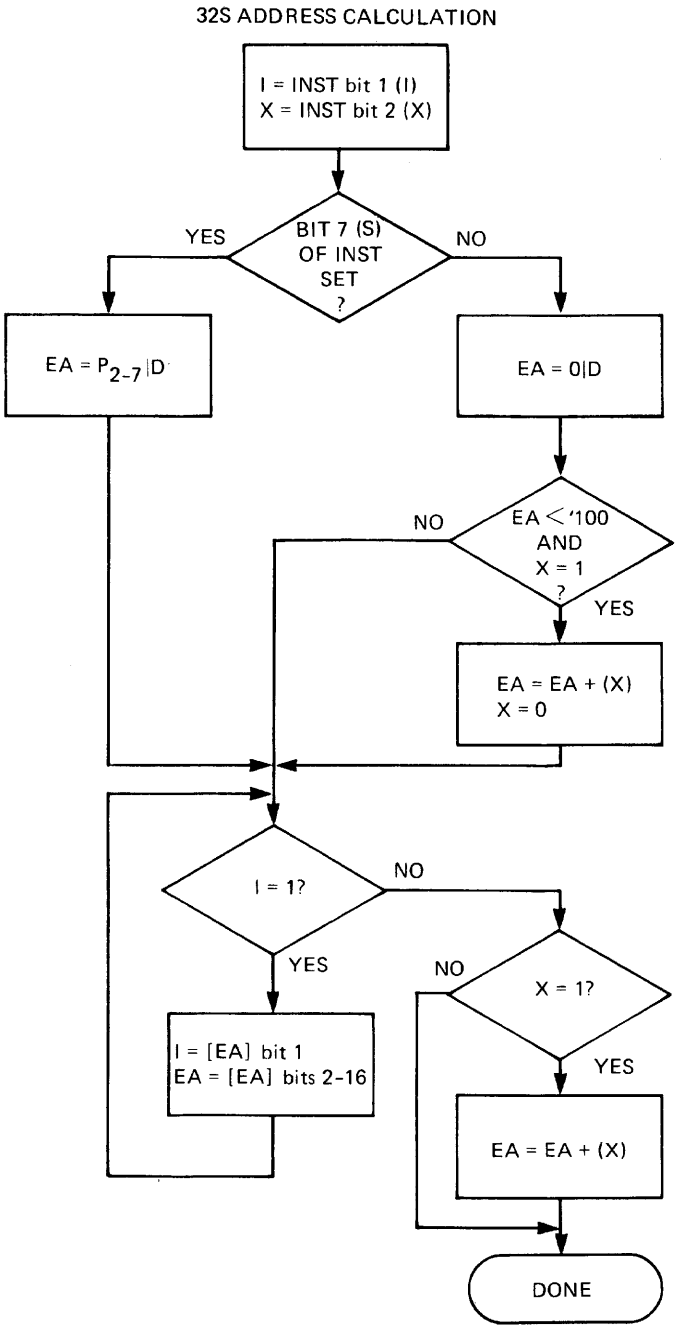
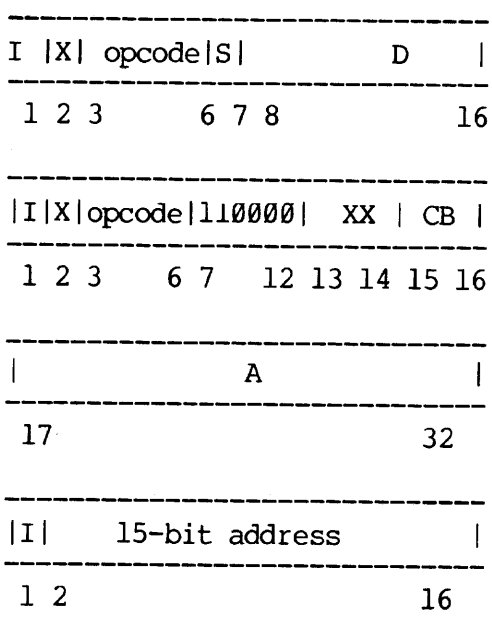


Figure 6-2. 32S Address Calculation

32R SUMMARY

Address Length: 15 bits; 32K word address space

Format:



Indexing: One level.

Indirection: Multiple levels.

I	X	S	CB	D	EA	Assembler Notation	Type
Ø	Ø	Ø	--	Ø to '777	Ø|D	LDA ADDR	Direct
Ø	1	Ø	--	Ø to '777	Ø|D+X	LDA ADDR,1	Indexed
1	Ø	Ø	--	Ø to '777	I(Ø|D)	LDA ADDR,*	Indirect
1	1	Ø	--	Ø to '77	I(Ø|D+X)	LDA ADDR,1*	Indirect, preindexed
1	1	Ø	--	'1ØØ to '777	I(Ø|D)+X	LDA ADDR,*1	Indirect, postindexed
Ø	Ø	1	--	-24Ø to +256	P+D	LDA ADDR	Direct
Ø	1	1	--	-24Ø to +256	P+D+X	LDA ADDR,1	Indexed
1	Ø	1	--	-24Ø to +256	I(P+D)	LDA ADDR,*	Indirect
1	1	1	--	-24Ø to +256	I(P+D)+X	LDA ADDR,*1	Indirect, postindexed
Ø	Ø	1	2	-------	SP	LDA @+	Postincrement
Ø	1	1	2	-------	I(SP)+X	LDA @+,*1	Postincrement, indirect, postindexed
1	Ø	1	2	-------	I(SP)	LDA @+,*	Postincrement, indirect
Ø	Ø	1	3	-------	SP-1	LDA -@	Predecrement
Ø	1	1	3	-------	I(SP-1)+X	LDA -@,*1	Predecrement indirect, postindexed
1	Ø	1	3	-------	I(SP-1)	LDA -@,*	Predecrement,

Ø	Ø	1	Ø	-------	A	LDA% ADDR	Direct, long reach indirect
Ø	1	1	Ø	-------	A+X	LDA% ADDR,X	Indexed, long reach
1	Ø	1	Ø	-------	I(A)	LAD% ADDR,*	Indirect, long reach
1	1	1	2	-------	I(A+X)	LDA% ADDR,X*	Indirect, preindexed long reach
1	1	1	2	-------	I(A)+X	LDA% ADDR,*X	Indirect, postindexed long reach
Ø	Ø	1	1	-------	A+SP	LDA @+ADDR	Direct, stack relative
Ø	1	1	1	-------	A+SP+X	LDA @+ADDR,X	Indexed, stack relative
1	Ø	1	1	-------	I(A+SP)	LDA @+ADDR,*	Indirect, stack relative
1	1	1	1	-------	I(A+SP+X)	LDA @+ADDR,X*	Indirect, preindexed, stack relative
1	1	1	3	-------	I(A+SP)+X	LDA @+ADDR,*X	Indirect, postindexed, stack relative

Table Description

P = contents of program counter after instruction fetch (pointing at instruction plus 1)

Ø|D = displacement into sector Ø. Sector bits of effective address (bits 3-8) are zero.

X = contents of index register

I(expression) = treat the effective address as indirect address

SP = stack pointer

ADDR = location addressed by the LDA

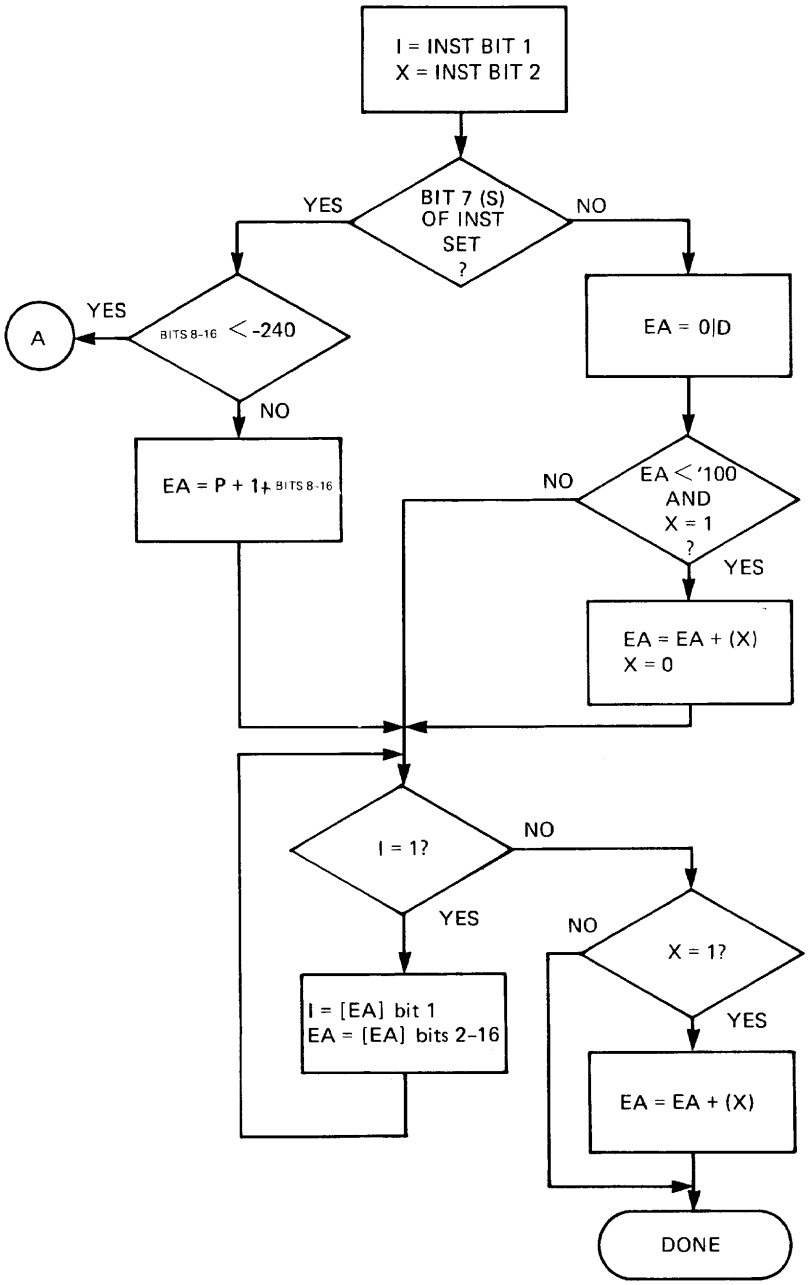


Figure 6-3. 32R Address Calculation (1 of 5)

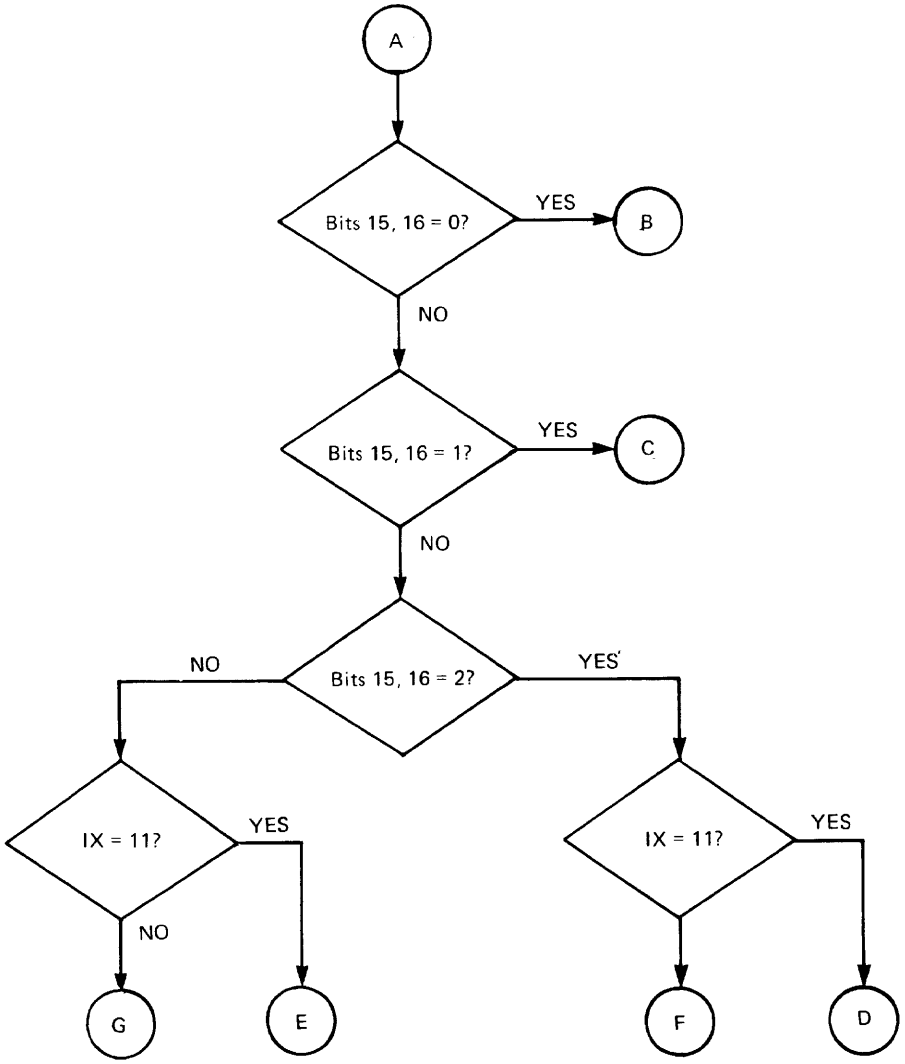


Figure 6-3. 32R Address Calculation (2 of 5)

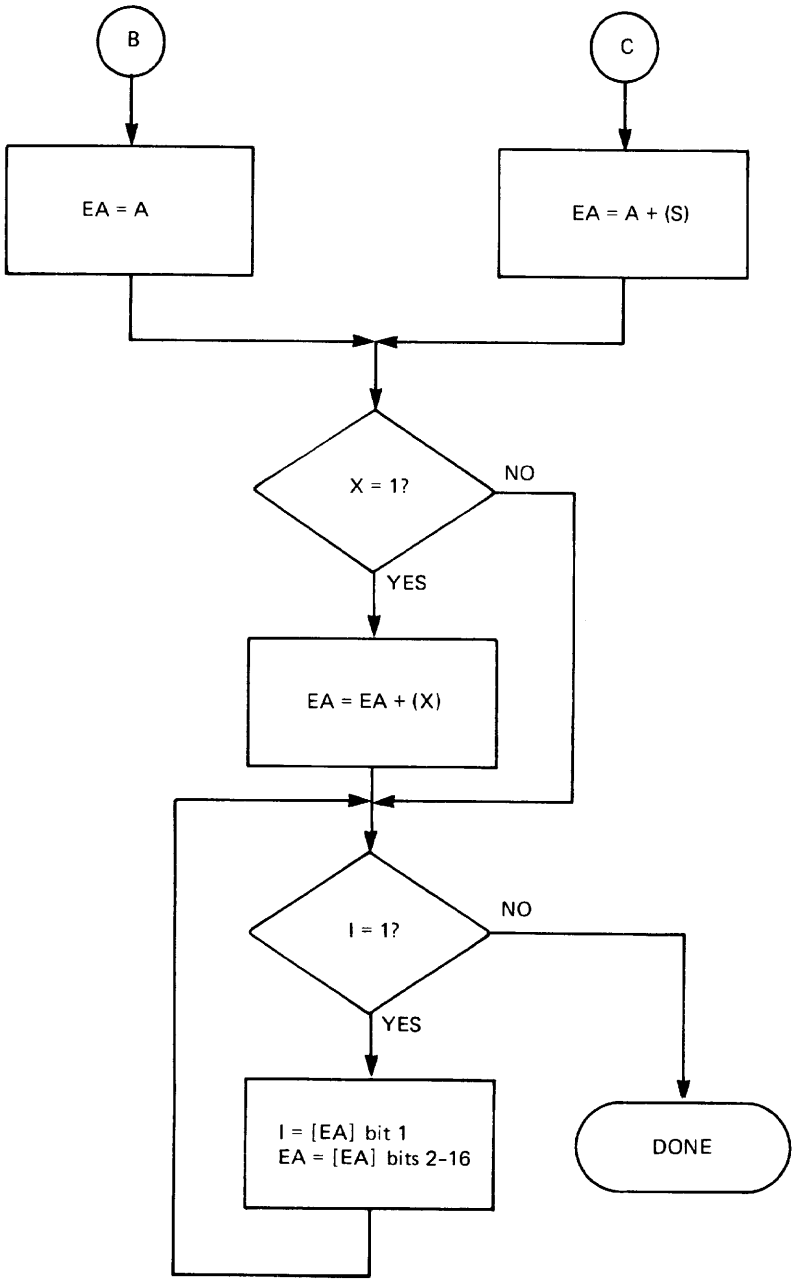


Figure 6-3. 32R Address Calculation (3 of 5)

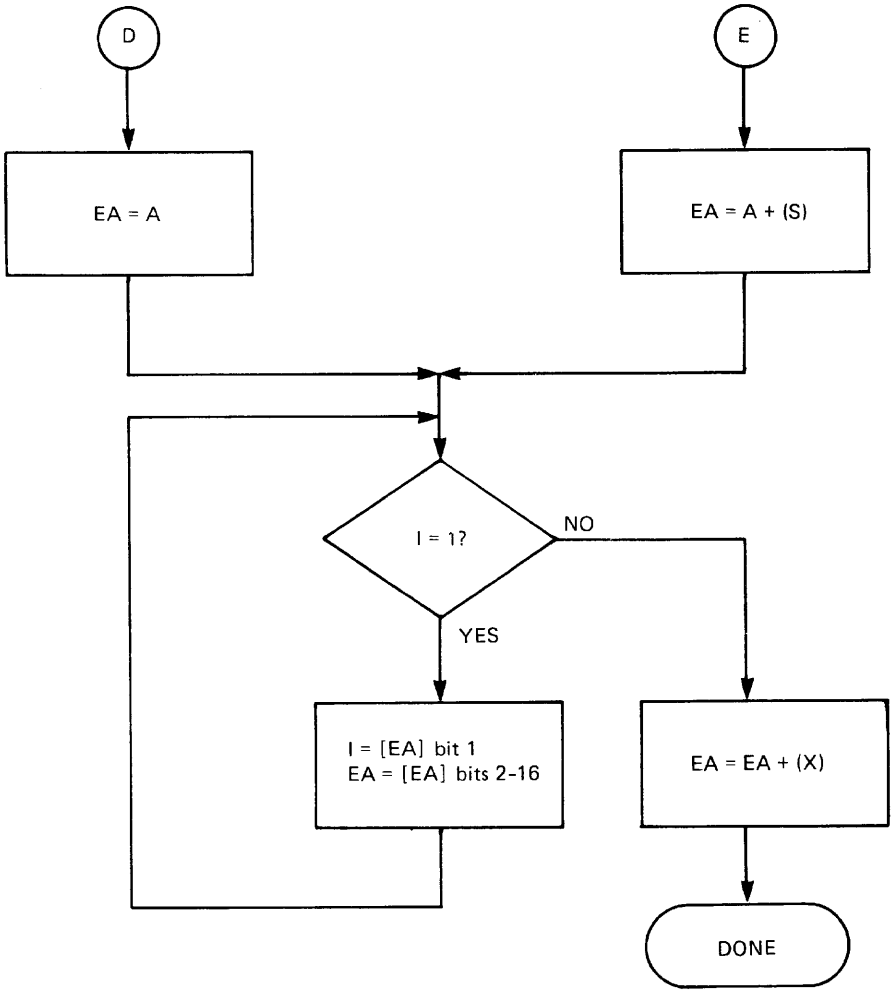


Figure 6-3. 32R Address Calculation (4 of 5)

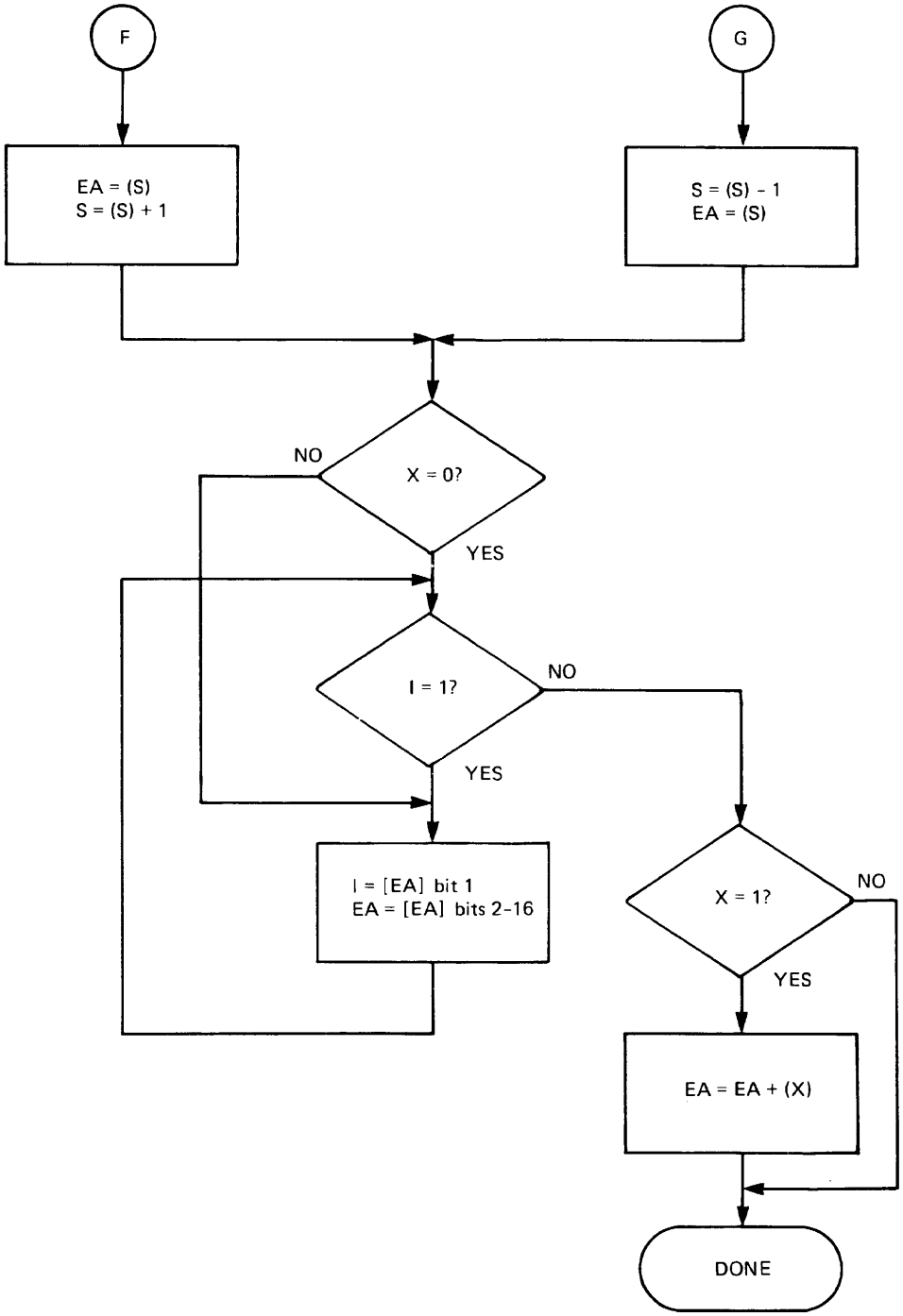
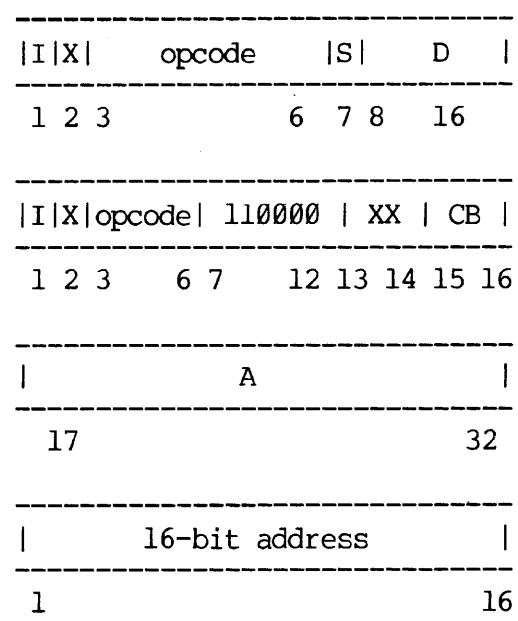


Figure 6-3. 32R Address Calculation (5 of 5)

64R SUMMARY

Address Length: 16 bits; 64K word address space

Format:



Indexing: One level.

Indirection: One level.

I	X	S	CB	D	EA	Assembler Notation	Type
0	0	0		0 to '777	0|D	LDA ADDR	Direct
0	1	0		0 to '777	0|D+X	LDA ADDR,1	Indexed
1	0	0	--	0 to '777	I(0|D)	LDA ADDR,*	Indirect
1	1	0	--	0 to '77	I(0|D+X)	LDA ADDR,1*	Indirect, preindexed
			--				
1	1	0	--	'100 to '777	I(0|D)+X	LDA ADDR*1	Indirect, postindexed
0	0	1	--	-240 to +256	P+D	LDA ADDR	Direct
0	1	1	--	-240 to +256	P+D+X	LDA ADDR,1	Indexed
1	0	1	--	-240 to +256	I(P+D)	LDA ADDR,*	Indirect
1	1	1	--	-240 to +256	I(P+D)+X	LDA ADDR,*1	Indirect, postindexed
0	0	1	2	-------	SP	LDA @+	Postincrement
0	1	1	2	-------	I(SP)+X	LDA @+,*1	Postincrement, indirect, postindexed
1	0	1	2	-------	I(SP)	LDA @+,*	Postincrement, indirect
0	0	1	3	-------	SP-1	LDA -@	Predecrement
0	1	1	3	-------	I(SP-1)+X	LDA -@,*1	Predecrement indirect, postindexed
1	0	1	3	-------	I(SP-1)	LDA -@,*	Predecrement,

Ø	Ø	1	Ø	-------	A	LDA% ADDR	Direct, long reach indirect
Ø	1	1	Ø	-------	A+X	LDA% ADDR,X	Indexed, long reach
1	Ø	1	Ø	-------	I(A)	LDA% ADDR,*	Indirect, long reach
1	1	1	Ø	-------	I(A+X)	LDA% ADDR,X*	Indirect, long reach preindexed
1	1	1	Ø	-------	I(A)+X	LDA% ADDR,*X	Indirect, long reach postindexed
Ø	Ø	1	1	-------	A+SP	LDA @+ADDR	Direct, stack relative
Ø	1	1	1	-------	A+SP+X	LDA @+ADDR,X	Indexed, stack relative
1	Ø	1	1	-------	I(A+SP)	LDA @+ADDR,*	Indirect, stack relative
1	1	1	1	-------	I(A+SP+X)	LDA @+ADDR,X*	Indirect, preindexed, stack relative
1	1	1	3	-------	I(A+SP)+X	LDA @+ADDR,*X	Indirect, postindexed, stack relative

Table Description

P = contents of program counter after instruction fetch (pointing at instruction plus 1)

Ø|D = displacement into sector Ø. Sector bits of effective address (bits 3-8) are zero.

X = contents of index register

I(expression) = treat effective address as indirect address

SP = stack pointer

ADDR = location addressed by the LDA

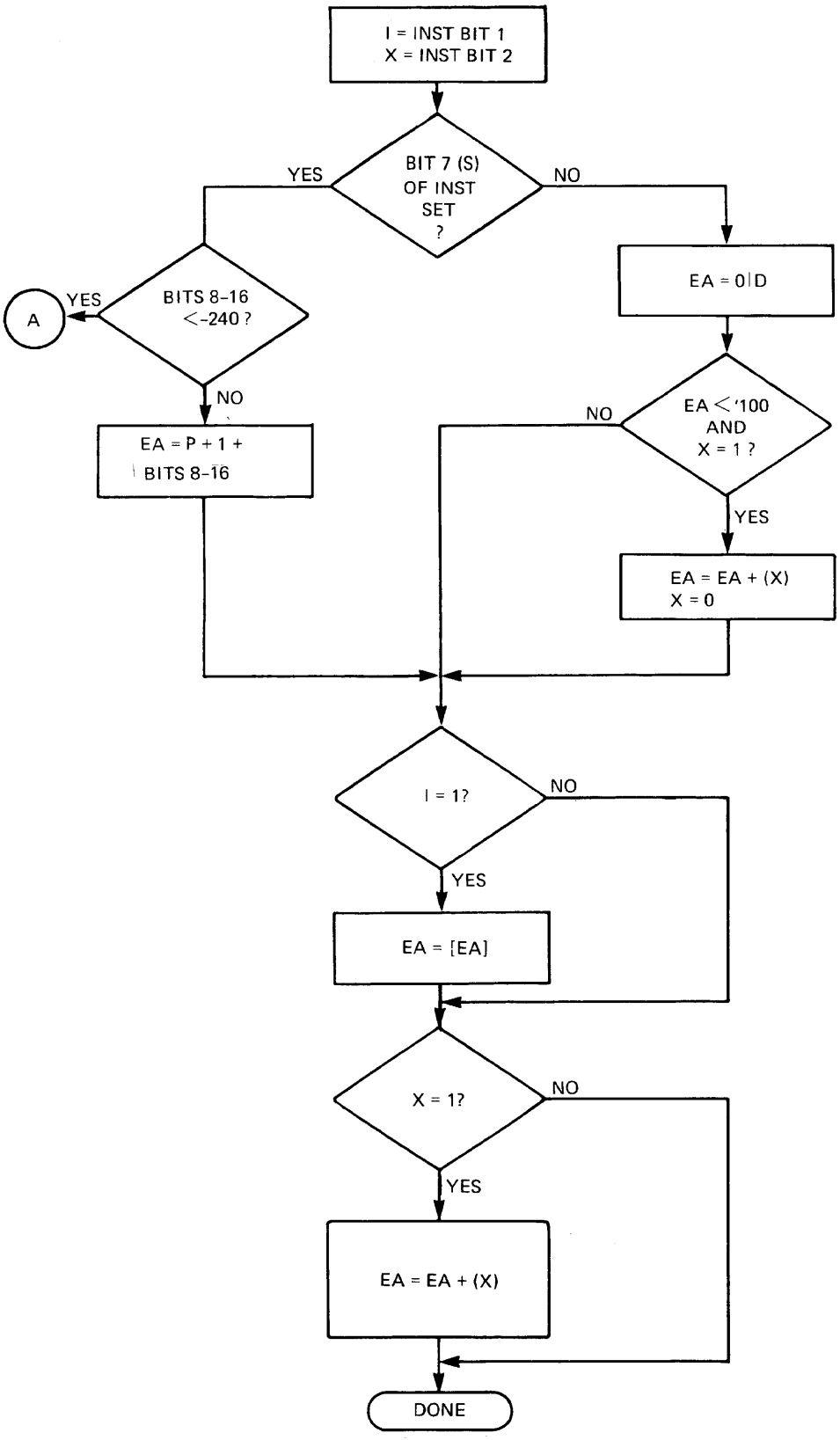


Figure 6-4. 64R Address Calculation (1 of 5)

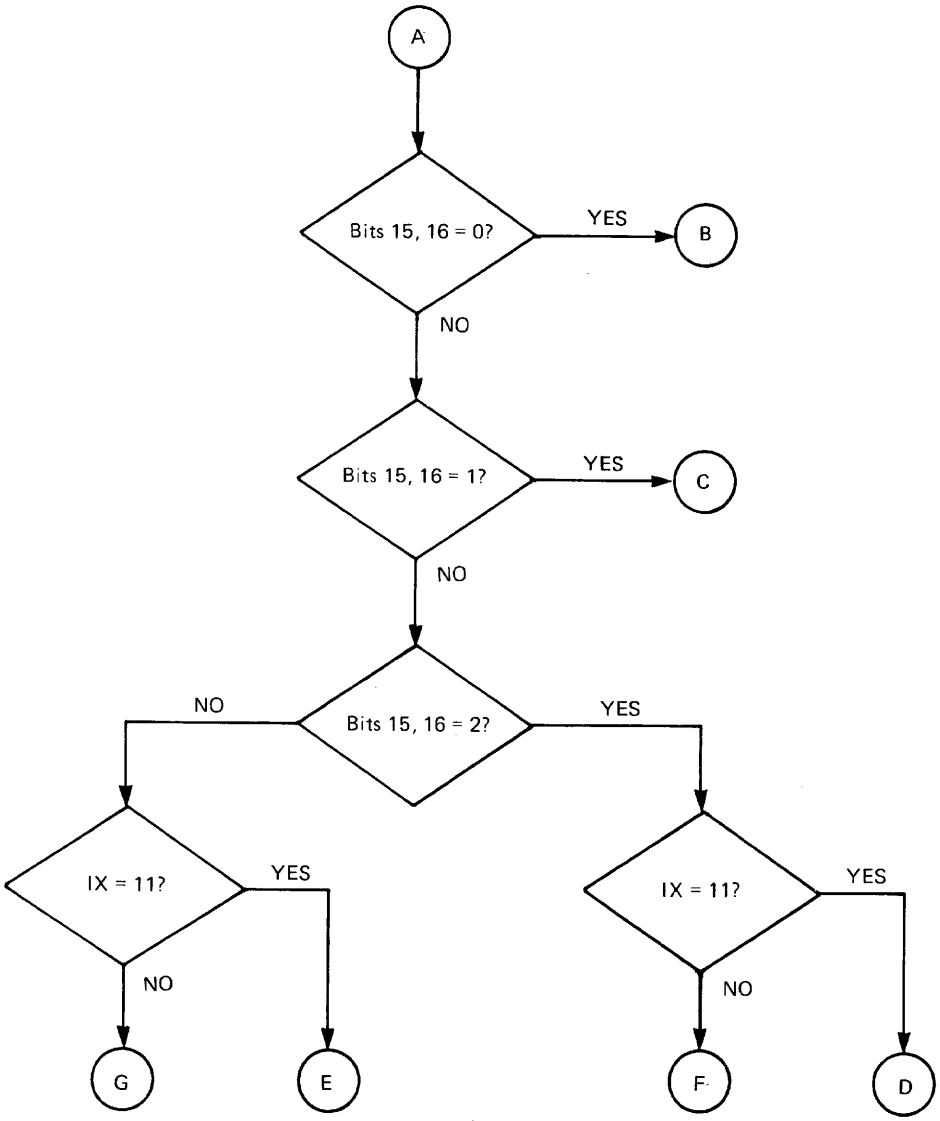


Figure 6-4. 64R Address Calculation (2 of 5)

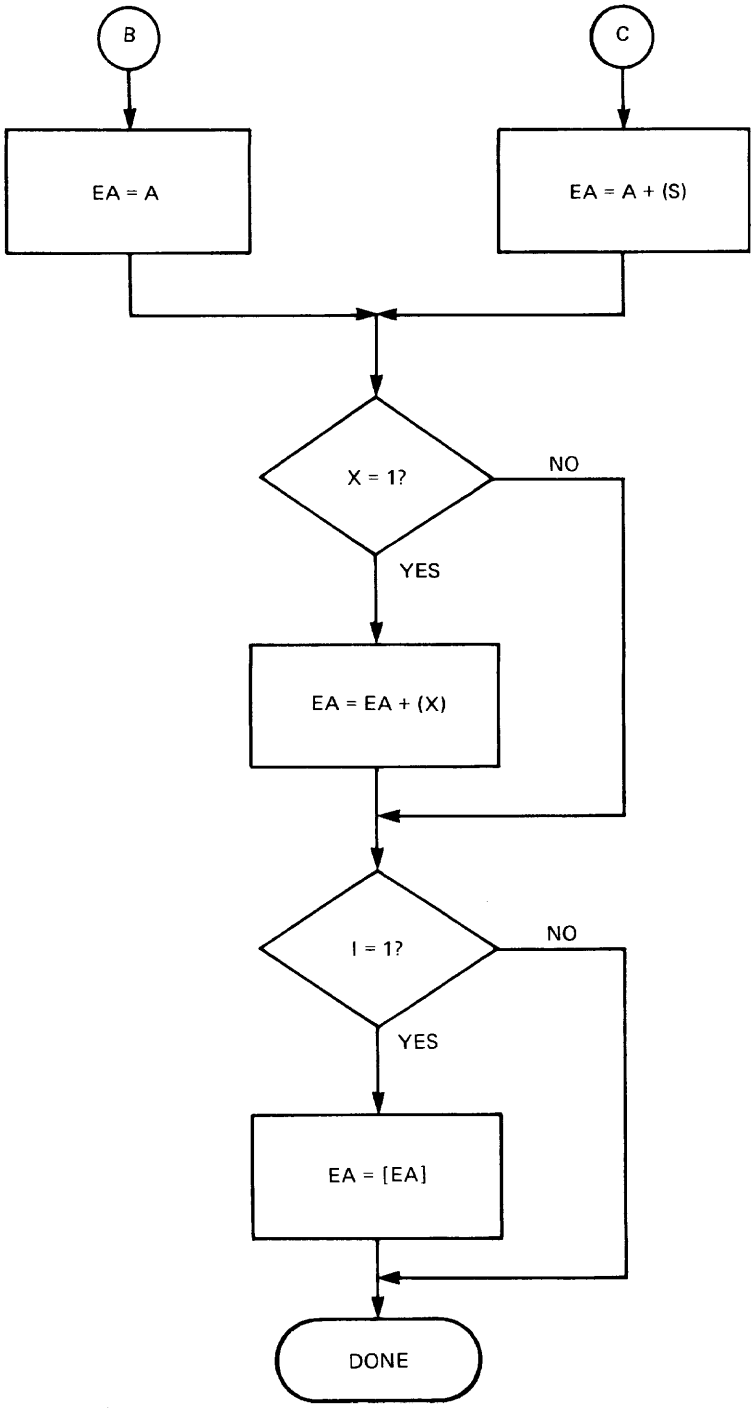


Figure 6-4. 64R Address Calculation (3 of 5)

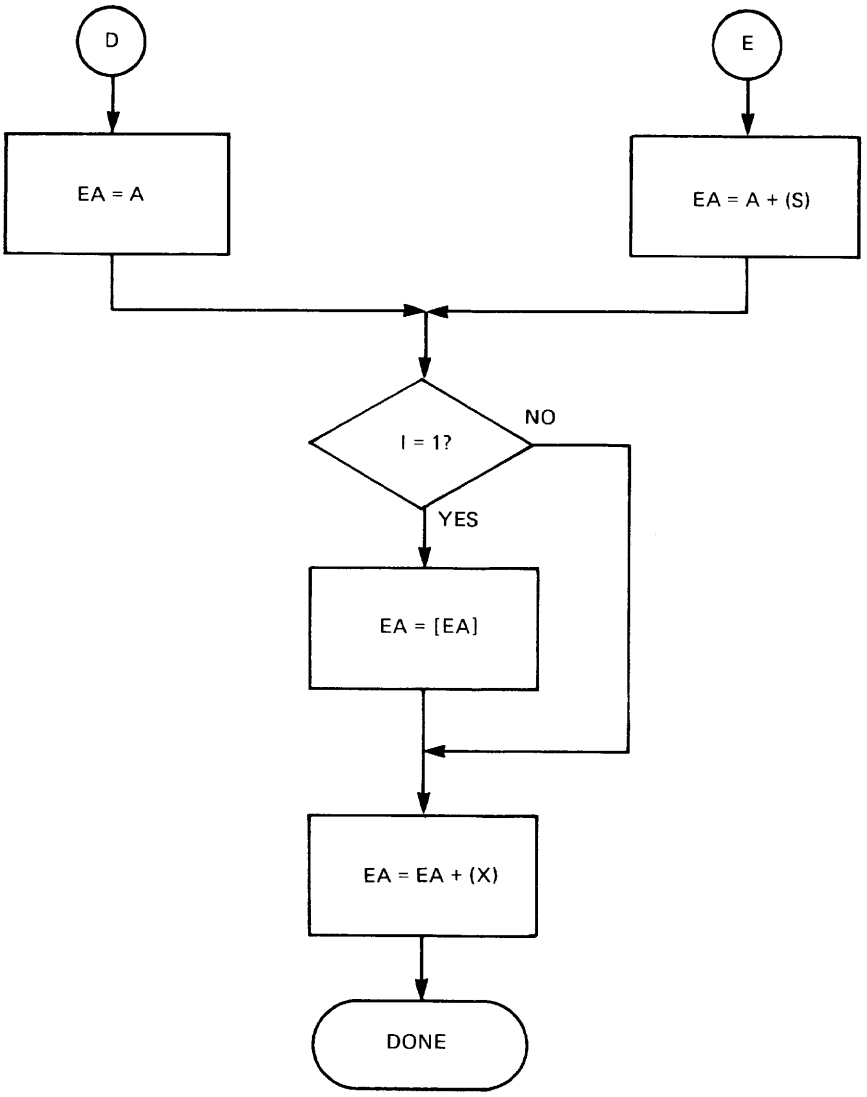


Figure 6-4. 64R Address Calculation (4 of 5)

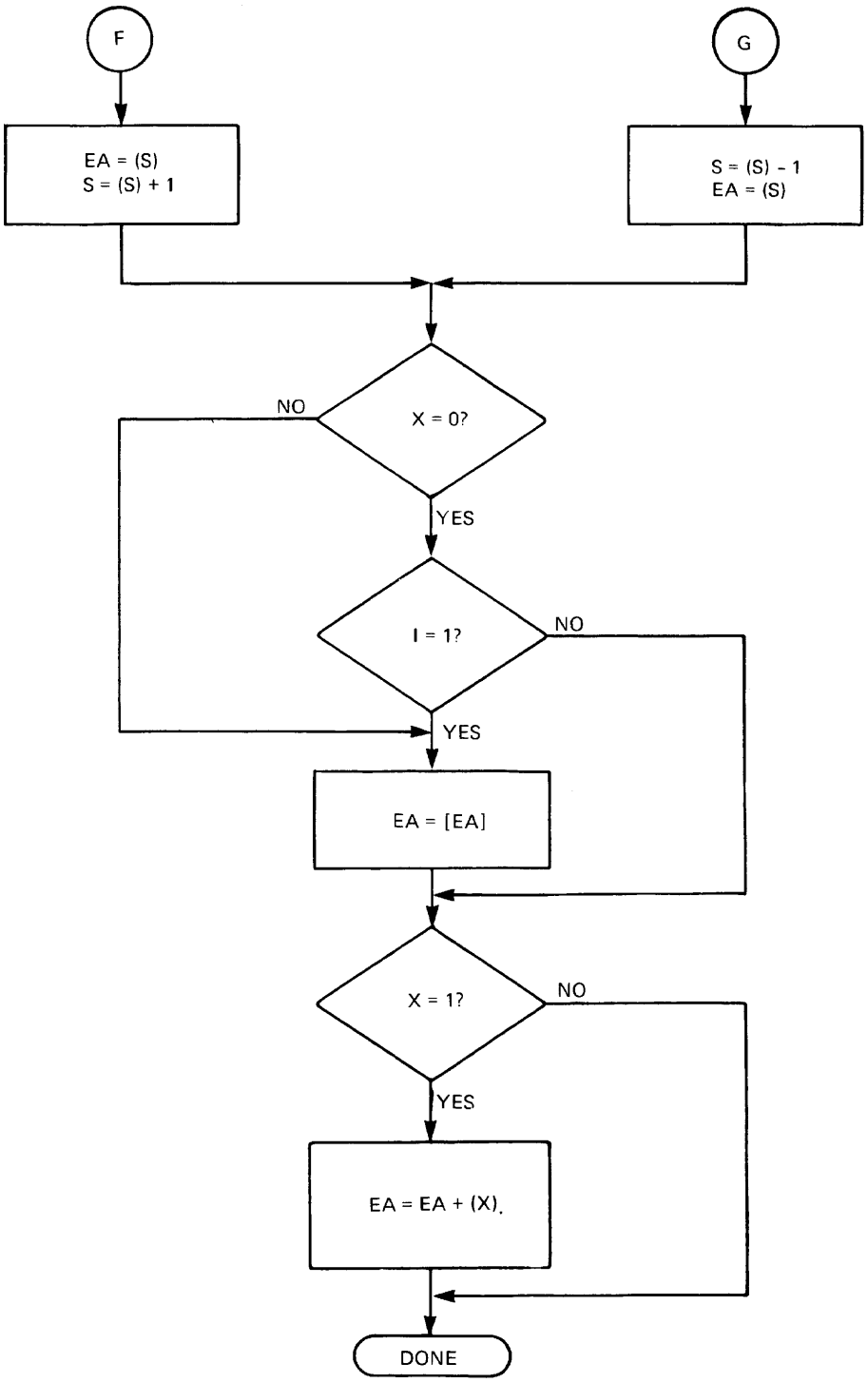
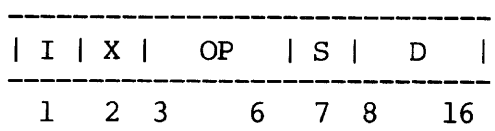


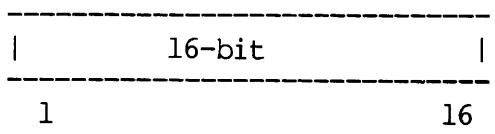
Figure 6-4. 64R Address Calculation (5 of 5)

64V PROCEDURE RELATIVE (One Word, S=1)

Address length: 16 bits; 64K word address space

Format:





Indexing: One level

Indirection: One level

I	X	S	D	EA	Type
0	0	1	-224 to +256	P+D	Direct
0	1	1	-224 to +256	P+D+X	Indexed
1	0	1	-224 to +256	I(P+D)	Indirect
1	1	1	-224 to +256	I(P+D)+X	Indirect, postindexed

Table Description

P = contents of program counter after instruction fetch (pointing at instruction plus one).

D = procedure segment displacement.

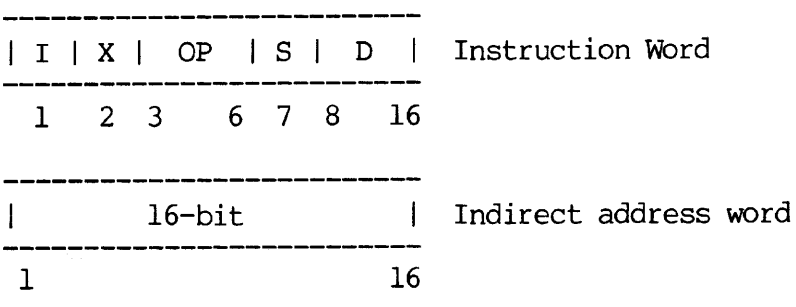
X = contents of X register.

I(expression) = treat effective address as indirect address

64V BASE REGISTER RELATIVE (One Word, S=0)

Address Length: 3 64K segments

Format:



Indexing: One level

Indirection: One level

I	X	S	D	EA	Type
0	0	0	0-'7	register location	Direct
			'10-'377	SB+D	
			'400-'377	LB+D	
0	1	0	0-'377	if D+X<'10 then EA=register location else SB+D+X	Indexed
			'400-'777	LB+D+X	
1	0	0	0-'7	I(REG)	Indirect
			'10-'777	I(PB|D)	
1	1	0	0-'77	I(PB|D+X)	Indirect, preindexed
1	1	0	'100-'777	I(PB|D)+X	Indirect, postindexed

REG = R-mode registers, i.e., A, B, X, etc.

PB = procedure base register

LB = link base register

SB = stack base register

X = Index register

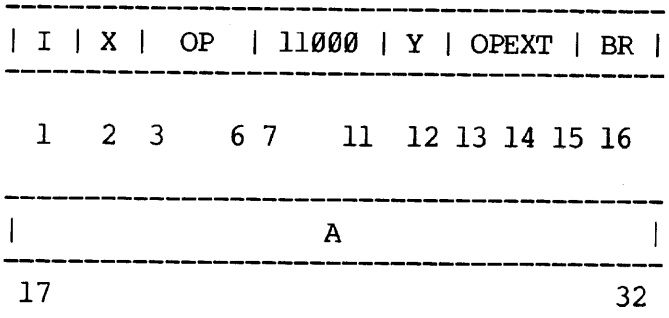
D = Displacement field

I(expression) = treat effective address as indirect address

64V TWO WORD MEMORY REFERENCE

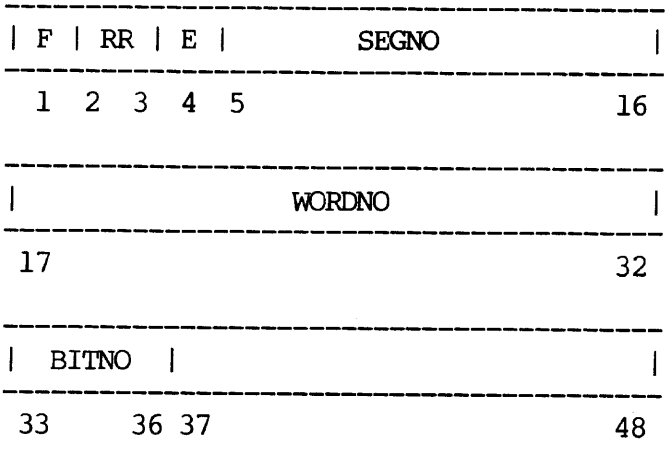
Address Length: 28 bits; 4096 64K segments

Format:



Indexing: X and Y

Indirection: 48 bit word



I	X	Y	BR	Effective Address	Meaning
0	0	0	0	PB|D	
			1	SB+D	Direct
			2	LB+D	
			3	XB+D	
0	0	1	0	PB|D+Y	
			1	SB+D+Y	Indexed by Y
			2	LB+D+Y	
			3	XB+D+Y	
0	1	0	0	PB|D+X	
			1	SB+D+X	
			2	LB+D+X	Indexed by X
			3	XB+D+X	
0	1	1	0	I(PB|D)	
			1	I(SB+D)	Indirect
			2	I(LB+D)	
			3	I(XB+D)	
1	0	0	0	I(PB|D+Y)	
			1	I(SB+D+Y)	Pre-indexed by Y
			2	I(LB+D+Y)	
			3	I(XB+D+Y)	
1	0	1	0	I(PB|D)+Y	
			1	I(SB+D)+Y	Post-indexed by Y
			2	I(LB+D)+Y	
			3	I(XB+D)+Y	
1	1	0	0	I(PB|D+X)	
			1	I(SB+D+X)	Pre-indexed by X
			2	I(LB+D+X)	
			3	I(XB+D+X)	
1	1	1	0	I(PB|D)+X	
			1	I(SB+D)+X	Post-indexed by X
			2	I(LB+D)+X	
			3	I(XB+D)+X	

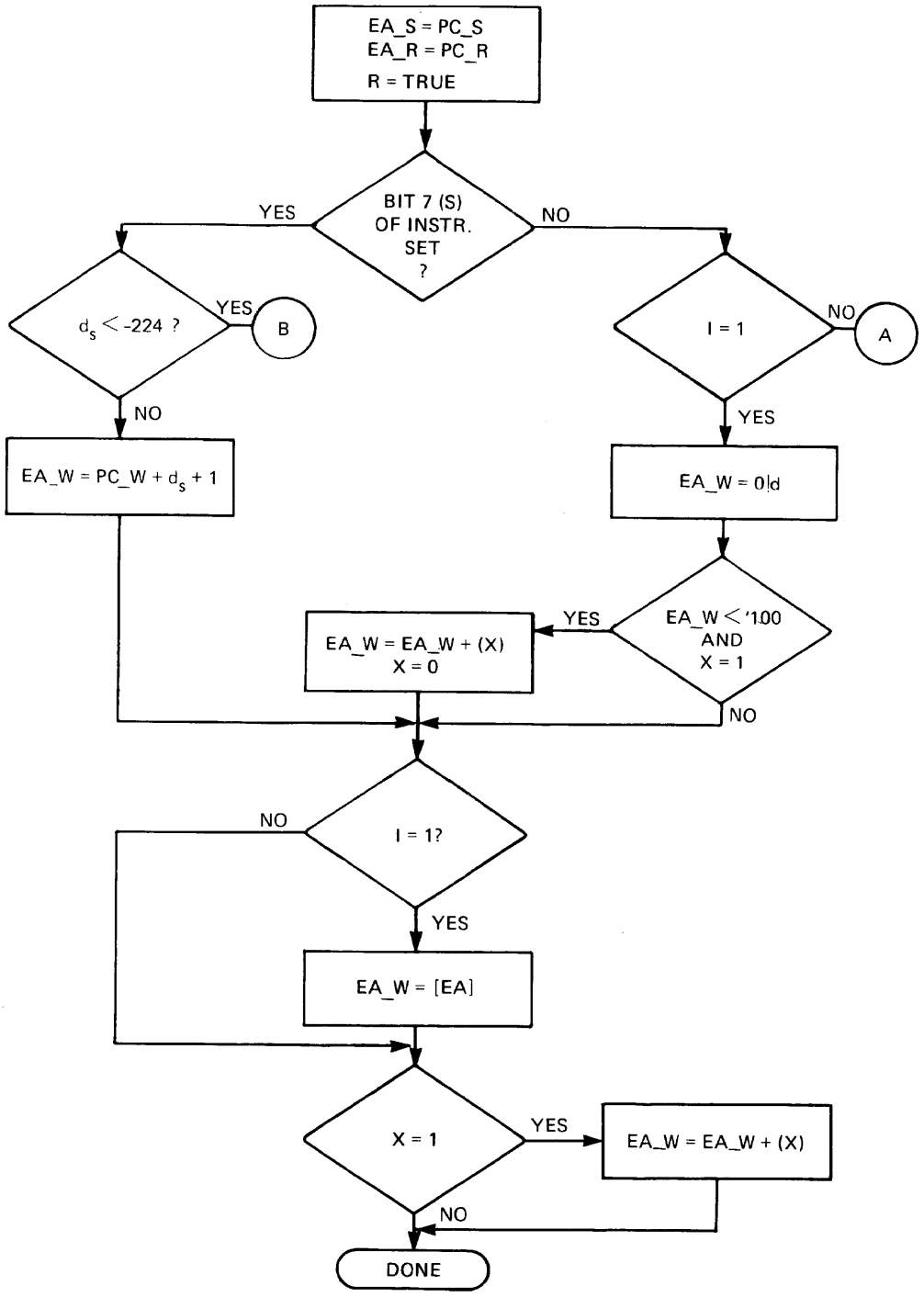


Figure 6-5. 64V Address Calculation (1 of 3)

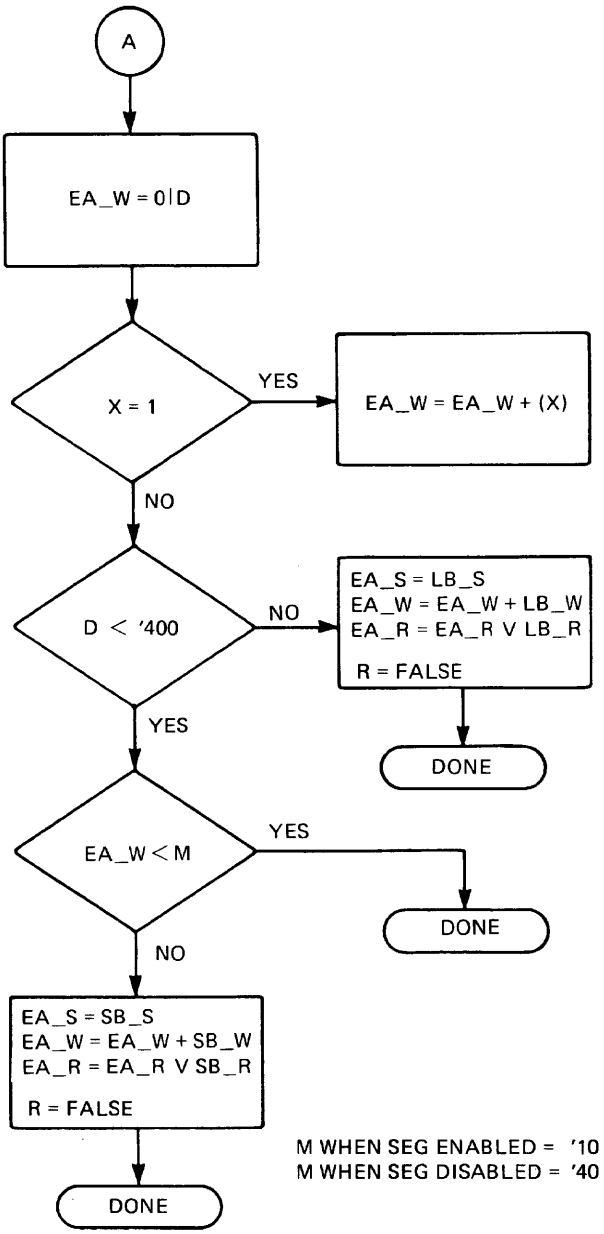


Figure 6-5. 64V Address Calculation (2 of 3)

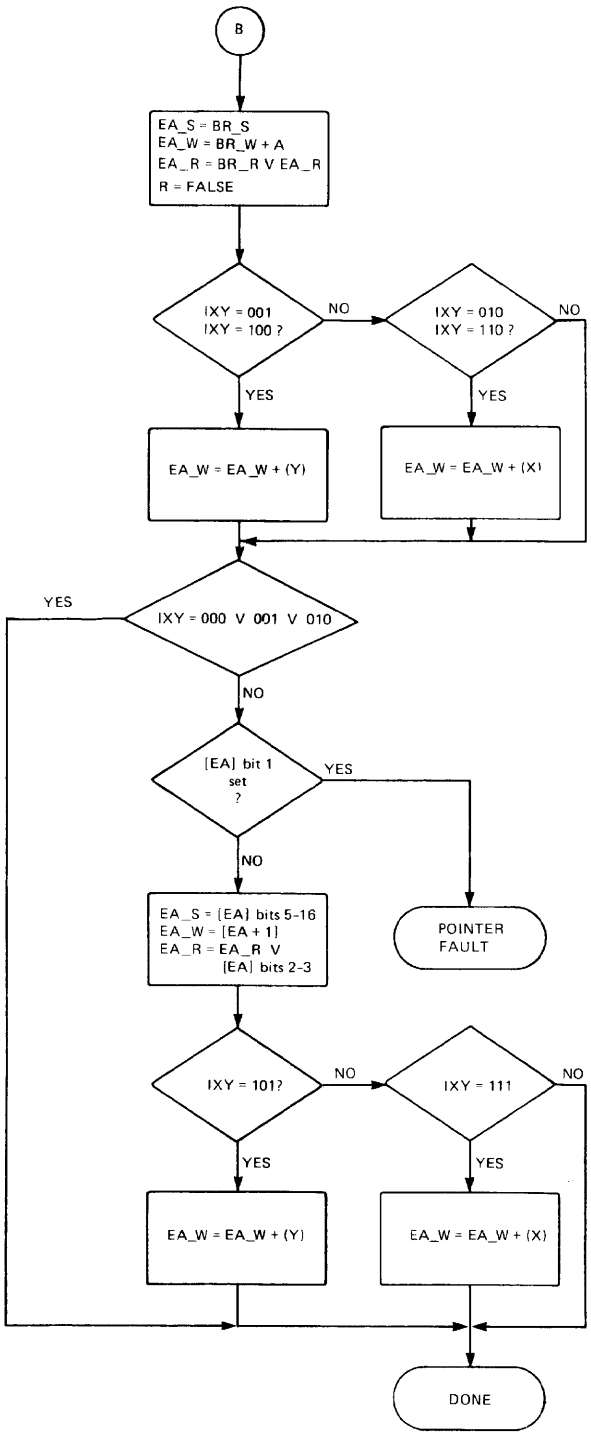


Figure 6-5. 64V Address Calculation (3 of 3)

SECTION 7

INSTRUCTION DEFINITIONS - SRV

ADMOD - Addressing Mode

Set the addressing mode of the machine.

Enter 16S Mode SRV GEN

E16S

Use 16S address calculations to form subsequent effective addresses and enable S-mode interpretation of instruction. See section on address resolution for details.

Enter 32S Mode SRV GEN

E32S

Use 32S address calculations to form subsequent effective addresses and enable S-mode interpretation of instructions. See section on address resolution for details.

Enter 32R Mode SRV GEN

E32R

Use 32R address calculations to form subsequent effective addresses and enable R-mode interpretation of instructions. See section on address resolution for details.

Enter 64R Mode SRV GEN

E64R

Use 64R address calculations to form subsequent effective addresses and enable R-mode interpretation of instructions. See section on address resolution for details.

Enter 64V Mode SRV GEN

E64V

Use 64V address calculations to form subsequent effective addresses and enable 64V-mode interpretation of instructions. See section on address resolution for details.

Enter 32I Mode SRV GEN

E32I

Use 32I address calculations to form subsequent effective addresses and enable 32I-mode interpretation of instructions. See section on address resolution for details.

BRAN - Branch V BRAN

The branch instructions are two word generics which test the contents of a register or the result of a previous ARITHMETIC or COMPARE operation, as indicated by the condition codes (CC), the C-bit, and the L-bit.

Word 1 = opcode and conditional test

Word 2 = 16-bit direct word address within the current procedure segment

Condition code branches test six conditions based on the LT bit, the EQ bit, and the opcode.

Condition	Meaning
<	branch if LT bit set and EQ bit cleared
≤	branch if LT bit set or EQ bit set
=	branch if EQ bit set
≠	branch if EQ bit cleared
≥	branch if LT bit cleared or EQ bit set
>	branch if LT bit cleared and EQ bit cleared

Test Condition Code and Branch

These instructions have the following format:

Branch if condition code { LT, LE, EQ, NE, GE, GT } Ø

For example: BCLT ADDR means Branch to ADDR if the condition code is less than zero (LT bit set and EQ bit cleared).

BCLT ADDR	if CC<Ø, then ADDR->PC
BCLE ADDR	if CC≤Ø, then ADDR->PC
BCEQ ADDR	if CC=Ø, then ADDR->PC
BCNE ADDR	if CC≠Ø, then ADDR->PC
BCGE ADDR	if CC≥Ø, then ADDR->PC
BCGT ADDR	if CC>Ø, then ADDR->PC

Test Magnitude Condition and Branch

These instructions have the following format:

Branch to ADDR if L=1 and condition code $\left\{\begin{matrix} LT \\ LE \\ EQ \\ NE \\ GE \\ GT \end{matrix}\right\}$ Ø

For example: BMLT ADDR means Branch to ADDR if the L-bit is set and condition code is less than Ø (LT bit set and EQ bit cleared).

```
BMLT ADDR        if L=1 and CC<Ø, then ADDR->PC
BMLE ADDR        if L=1 and CC≤Ø, then ADDR->PC
BMEQ ADDR        if L=1 and CC=Ø, then ADDR->PC
BMNE ADDR        if L=1 and CC≠Ø, then ADDR->PC
BMGE ADDR        if L=1 and CC≥Ø, then ADDR->PC
BMGT ADDR        if L=1 and CC>Ø, then ADDR->PC
```

Test C-Bit and Branch

Branch if C-Bit $\left\{\begin{matrix} Ø \\ 1 \end{matrix}\right\}$

- Branch if C-bit Reset (equals zero)

  BCR ADDR

  if C-bit=Ø, then ADDR->PC

- Branch if C-bit Set (equals one)

  BCS ADDR

  if C-bit=1, then ADDR->PC

Test L-Bit

Branch if L-Bit $\left\{\begin{matrix} Ø \\ 1 \end{matrix}\right\}$

- Branch if L-bit Reset (equals zero)

  BLR ADDR

  if L-bit=Ø, then ADDR->PC

- Branch if L-bit Set (equals one)

BLS ADDR

if L-bit=1, then ADDR->PC

Branch on Register

These instructions have the following format:

Branch if { A-Register / L-Register / Floating-Register } { LT / LE / EQ / NE / GE / GT } Ø

For example: BLT ADDR means Branch to ADDR if the contents of the A register is less than zero (LT bit is set and EQ bit is cleared).

BLT ADDR	if A<Ø, then ADDR->PC
BLE ADDR	if A≤Ø, then ADDR->PC
BEQ ADDR	if A=Ø, then ADDR->PC
BNE ADDR	if A≠Ø, then ADDR->PC
BGE ADDR	if A≥Ø, then ADDR->PC
BGT ADDR	if A>Ø, then ADDR->PC
BLLT ADDR	if L<Ø, then ADDR->PC
BLLE ADDR	if L≤Ø, then ADDR->PC
BLEQ ADDR	if L=Ø, then ADDR->PC
BLNE ADDR	if L≠Ø, then ADDR->PC
BLGE ADDR	if L≥Ø, then ADDR->PC
BLGT ADDR	if L>Ø, then ADDR->PC
BFLT ADDR	if F<Ø, then ADDR->PC
BFLE ADDR	if F≤Ø, then ADDR->PC
BFEQ ADDR	if F=Ø, then ADDR->PC
BFNE ADDR	if F≠Ø, then ADDR->PC
BFGE ADDR	if F≥Ø, then ADDR->PC
BFGT ADDR	if F>Ø, then ADDR->PC

Increment or Decrement X or Y and Branch

{ Increment / Decrement } { X / Y } by 1 then branch to ADDR if result = Ø

BIX ADDR	X+1->X; if X=Ø then ADDR->PC
BIY ADDR	Y+1->Y; if Y=Ø then ADDR->PC
BDX ADDR	X-1->X; if X=Ø then ADDR->PC
BDY ADDR	Y-1->Y; if Y=Ø then ADDR->PC

Computed GOTO V GEN

CGT n If 1≤A<n
then [PC+A]->PC
else PC+n->PC

Instruction word followed by n further words:

Word 1 contains integer n

Words 2-n contain branch addresses within the current procedure segment.

If the contents of register A is less than n and greater than or equal to 1, then control passes to the address in PC+A; otherwise no branch is taken and control passes to PC+n.

CHAR - Character String Operations

These instructions use the field address and length registers (FAR, FLR) which have been set up by field operation instructions prior to the use of these instructions. Character string operations perform memory to memory operations on variable length character fields. The FAR is used as a byte pointer and the bit offset (low order 3 bits) is ignored.

Date Type: Characters are 8-bit bytes. The format is unspecified and may be determined by programmer, e.g., ASCII, EBCDIC, etc. The translate instruction (ZTRN), for example uses a table set up by the programmer to translate one character code into another.

Load Character

V

CHAR

LDC {1 / 0}

CC=NE=succeed
CC=EQ=field empty

If the specified FLR is nonzero, load the single character pointed to by the specified FAR into A register bits 9-16. A-reg bits 1-8 are cleared. The specified FAR is advanced 8 bits to the next character, and the FLR is decremented by 1. Set condition code NE.

If the specified FLR is zero, then set the condition code EQ.

Store Character

V

CHAR

STC {1 / 0}

CC=NE=succeed
CC=EQ=abort

Store bits 9-16 of the A register into the character pointed to by the selected FAR. The FAR is advanced 8 bits to the next character, and the FLR is decremented by 1. Set the condition code NE.

If the specified FLR is zero, set the condition code EQ and do not store.

Move Character Field V CHAR

ZMV

Move characters from field 0 to field 1, going from left to right. If the source field is shorter than the destination field, the destination field is padded with ASCII blanks ('240). If the source field is longer than the destination field, the remainder of the source field is not moved. The FAR's and FLR's are left in an undefined state by this operation.

Setup:

FAR 0 = source field address (byte-aligned)
FLR 0 = source field length in bytes
FAR 1 = destination field address length (byte aligned)
FLR 1 = destination field length in bytes

Move Equal Length Fields V CHAR

ZMVD

Move characters from field 0 to field 1. There is no padding or truncation since only the number of characters to be moved is specified.

Setup:

FAR 0 = source field address (byte-aligned)
FAR 1 = destination field address (byte-aligned)
FLR 1 = number of characters to move

Fill Field V CHAR

ZFIL

Store the character contained in bits 9-16 of the A register into each character of field 1.

Setup:

A [9-16] = character to fill
FAR 1 = destination field address (byte aligned)
FLR 1 = destination field length in bytes

Translate Character Field V CHAR

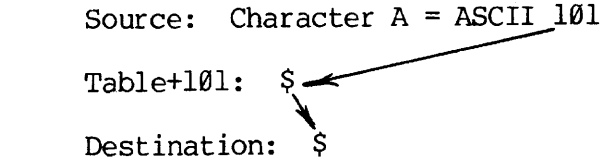
ZTRN

Use each character in field 0 as an index into the 256 byte table addressed by the XB register. Store each selected table character in the successive characters of field 1. Source and destination length are the same, specified by FLR1.

Setup:

FAR 0 = source field address (byte aligned)
FAR 1 = destination field address (byte aligned)
FLR 1 = number of characters to translate and move
XB = address of 256-byte translate table

Example:



Compare Character Field V CHAR

ZCM

Compare field 0 to field 1 and set condition codes based on the results. If the fields are not of equal length, the shorter field is logically padded with ASCII blanks ('240).

Setup:

FAR 0 = field 0 address (byte aligned)
FLR 0 = length of field 0 in characters
FAR 1 = field 1 address (byte aligned)
FLR 1 = length of field 1 in characters

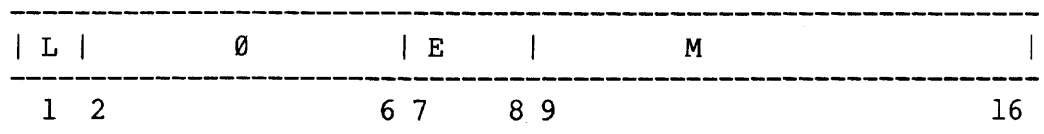
Condition code	Result
EQ	field 0 = field 1
LT	field 0 < field 1

Edit Character Field V CHAR

ZED

Move characters from field Ø into field 1 under the control of an edit program pointed to by XB. Movement stops when the source field is exhausted or when the end of the edit program is reached.

Edit Program Word



L = Last entry if set

Ø = Must be zero

E = Edit opcode

M = Edit modifier

Opcode(E)	Mnemonic	Definition
Ø	CPC	copy M characters from source to destination
1	INL	insert literal character M
2	SKC	skip M characters
3	BLK	supply M blanks (ASCII '240)

Setup:

FAR Ø = address of source field (byte aligned)
FAR 1 = address of destination field (byte aligned)
FLR 1 = number of characters to move and edit
XB = address of edit program

CLEAR - Clear Register

Clear A Right Byte	SRV	GEN
CAR	Ø->A(9-16)	

Clear bits 9-16 of register A without affecting bits 1-8.

Clear A Left Byte	SRV	GEN
CAL	Ø->A(1-8)	

Clear bits 1-8 of register A without affecting bits 9-16.

Clear the A Register	SRV	GEN
CRA	Ø->A	

Reset the contents of the A register to zero.

Clear the B Register	SRV	GEN
CRB	Ø->B	

Reset the contents of the B register to zero.

Clear Long	SRV	GEN
CRL	Ø->L	

Reset the contents of the L register to zero.

Clear E	V	GEN
CRE	Ø->E	

Reset the contents of E to zero.

Clear L and E	V	GEN
CRLE	Ø->L Ø -> E	

Reset the contents of L and E to zero.

DECI - Decimal Arithmetic

These instructions use the field address and length registers which have been set up by field operation instructions prior to the use of the decimal arithmetic instruction. The general setup is:

```
EAFA 0,Source field address
EAFA 1,Destination field address
LDL  Control word (described below)
decimal operation
```

Variations on this pattern are discussed in the appropriate instruction.

Decimal Data Types

The decimal instruction set operates on five types of decimal data. Table 7-1 summarizes the characteristics of each type:

Table 7-1. Decimal Data Types

Type	Code	Size of Decimal Digit	Comments
Leading Separate Sign	0	8	A plus sign (+) or a space represents a positive number. Operations generate +. A minus sign (-) represents negative number.
Trailing Separate Sign	1	8	
Packed Decimal	3	4	Use 4-bit nibble to represent each digit, followed by sign nibble. Positive sign represented by hex C in sign nibble. Negative in hex D. Requires odd number of digits and must start on byte boundary.
Leading Embedded Sign	4	8	A single character represents a digit and the sign of the field. When more than one character is listed, all will be recognized, but only first will be given in result field.
Trailing Embedded Sign	5	8	Embedded sign characters are as follows: Digit — Positive — Negative 0 — 0,+,{ — - } 1 — 1 A — J 2 — 2 B — K 3 — 3 C — L 4 — 4 D — M 5 — 5 E — N 6 — 6 F — O 7 — 7 G — P 8 — 8 H — Q 9 — 9 I — R

Arithmetic Instruction Register Usage (I-Mode only)

All arithmetic instructions use general registers GR0, GR1, GR3, GR4, and GR6, FLR0, FLR1 as scratch registers. These registers are not guaranteed to remain the same if an arithmetic instruction is executed.

Control Word Format

To specify the characteristics of the operation to be performed, most decimal arithmetic instructions require a control word to be loaded in the L register.

The general format is as follows:

A	-	B	C	-	T	D	E	F	G	H
1-6	7	8	9	10	11	12	13	14-16	17-22	23-29	30-32

Where:

A - Field 1, number of digits

E - Field 1, decimal data type (see Table 7-1)

B - If set, sign of field 1 is treated as opposite of its actual value.

C - If set, sign of field 2 is treated as opposite of its actual value. (XAD, XMP, XDV, XCM only)

D - If set, then round (XMV only)

F - Field 2, number of digits

H - Field 2, decimal data type

G - Scale differential (XAD, XMV, XCM only)

T - Generate positive results always

- - Unused, must be zero

The fields used by each instruction are listed in the instruction descriptions. Fields not used by an instruction must be zero.

The scale differential specifies the difference in decimal point alignment between the operator and fields for some instructions. This field is treated as a signed 7 bit two's complement number, where a positive value indicates a right shifting of Field 1 with respect to Field 2, and a negative value indicates a left shifting.

Decimal Exception (DEX)

There are two ways that an exception is handled. If the program is running in decimal exception mode, then a directed fault (similar to floating exception) is taken with the following fault codes:

	DEX TYPE (HIGH)	SUB CODE (LOW
Overflow	7	0
Divide by zero	7	1
Conversion	7	2

When not in decimal exception mode, the C bit is set and execution continues with the next instruction.

Decimal Add V DECI

XAD

A	-	B	C	E	F	G	H
1-6	7	8	9	10-11	12-13	14-16	17-22	23-29	30-32

Add the source field to the destination field and place the results in the destination field. The control word determines:

1) The operation - addition or subtraction

2) The scaling of the results

Operations:

B and C field control whether the operation is an add or subtract

B	C	Operation
0	0	+ Source + Destination
0	1	+ Source - Destination
1	0	- Source + Destination
1	1	- Source - Destination

Scaling: G Field

The scale differential field in the control word is used to adjust

field 1 in relation to field 2. If the scale differential is greater than zero, low order digits in field 1 will only affect the initial borrow from the low order digit of field 2. If the scale differential is less than zero, field 1 is considered to be logically extended with low order zeros when applied to field 2.

Decimal Multiply V DECI

XMP

A	-	B	C	-	T		F	G	H
1-6	7	8	9	10	11	12	13	14-16	17-22	23-29	30-32

Multiply destination field by source field and put the result in the destination field. To avoid overflow the destination field must have the same number of leading zeros as the length of the source field. The G field (scale differential) must contain the number of multiplier (source field) digits.

Multiply calculates source field times destination field. The product has a field length of 'destination field length (i.e., length of Field 2) + the number of multiplier digits'.

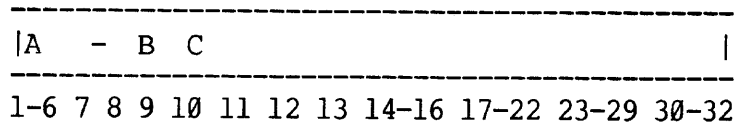
The product field is left justified in the destination field. The maximum partial product added in per traverse of the multiplicand is source digits + multiplier digits processed. Note also that there is an implied weighting of the partial product Field that is 10 **r, where r = multiplier digits.

The condition codes are set to reflect the state of the resultant field. The C bit is set on overflow (i.e., F2 is not source digits longer than the multiplier field) if not in DEX mode. If in DEX mode, a directed vector is taken.

The temporary base register is used by the instruction and may change.

Decimal Divide V DECI

XDV



Divide destination field by source field, placing both the quotient and remainder in the destination field.

Decimal Data Type - trailing sign embedded only. To allow room for both quotient and remainder the destination field must contain the same number of leading zeros as the length of the source field.

After divide the destination field contains quotient of length (destination length - source length) followed by remainder of source length.

Exceptions

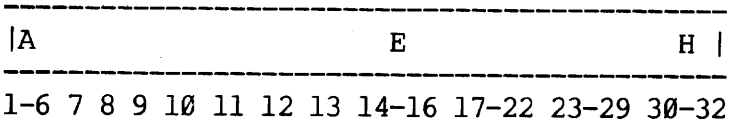
1) Source = 0

2) Not trailing embedded

3) Destination $\leq$ source.

Decimal to Binary Conversion V DECI

XDTB



XDTB converts the decimal field to binary. The length of the binary field is specified in the H field of the control word as follows:

0 - 16 Bits, returned in A

1 - 32 Bits, returned in L

2 - 64 Bits, returned in L/E

A conversion error exception is taken on overflow. The condition codes are undefined for this operation.

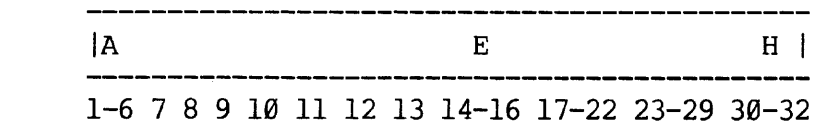
Field Address Register 2 is not used by this instruction and can be used as an accumulator for indexed pointers.

XDTB Characteristics:

This instruction returns a 16, 32 or 64 bit integer in either the A, L, or L/E registers, depending on the destination field type.

Binary to Decimal Conversion V DECI

XBTD



XBTD converts a 16, 32 or 64 bit signed binary number to decimal. The H field in the control word specifies the length and location of the binary source as follows:

0 - 16 Bits, located in EH

1 - 32 Bits, located in E

33 - 64 Bits, located in FP1

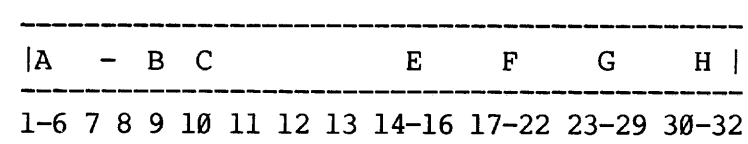
The condition codes are undefined for this operation. A conversion error exception is taken on overflow - see decimal exception.

XBTD Characteristics:

This instruction converts the binary field present in EH, E or FPR1 (depending on field type) into a decimal field. Unlike the rest of the decimal arithmetic instructions, XBTD returns the decimal field in what elsewhere is known as the "source" field address register.

Decimal Compare V DECI

XCM



XCM sets the condition codes to reflect the comparison Field 2 :: Field 1. The scale difference applies as in XAD.

The condition codes are set as follows:

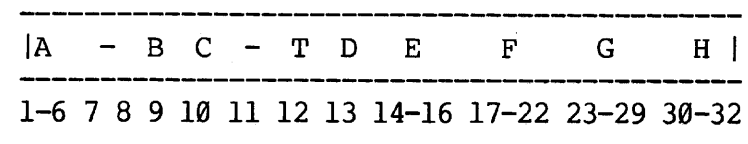
GT = Field 2 > Field 1

EQ = Field 2 = Field 1

LT = Field 2 < Field 1

Decimal Move V DECI

XMV



XMV moves source to destination, changing the sign if the B bit in the control word is set, and rounding if the D bit is set and G, the scale differential, is greater than zero. If the scale differential is negative then zeros are supplied before field 1 is used for a source. The condition codes are set to reflect the state of the destination after the move.

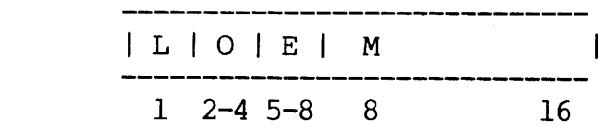
Numeric Edit V DECI

XED

Processes an edit sub-program addressed by the temporary base register to control the editing of the source field into the destination field. The source field must have leading separate sign, and must have the same number of digits and the same decimal point alignment as called for by the edit sub-program. Normal setup for the instruction would consist of a decimal move to correct the type, length, and alignment of the number to be edited. The A register must equal one if the source

field is 0; otherwise the A register must be 0.

The edit sub-program consists of a list of words formatted as follows:



Where:

L = Last entry if set

E = Edit opcode

M = Edit modifier

The XED instruction maintains several internal variables during its processing which are used to control the operation. These variables are:

- Zero suppress character - initial value is blank (ASCII '240).
- Floating edit character - initially not defined
- Sign of the source field - established by fetching the first character of the source field.
- Significance flag - records the end of zero suppression.

Edit Sub Operations

Opcode	Mnemonic	Definition
00	ZS	zero suppress next M digits. Digits are consecutively fetched from the source field and the significance flag is checked. If the significance flag is set, the digit is copied to the destination field. If the significance flag is clear and the digit is non-zero, the significance flag is set, the floating character inserted (if it is currently defined), and the digit is copied. Otherwise the zero suppress character is substituted for the zero digit in the destination field.
01	IL	insert literal M in destination field.
02	SS	set zero suppress character to M
03	ICS	insert literal M if the significance flag set; otherwise insert zero suppress character
04	ID	insert M digits. If significance flag is clear, it is set and the floating edit character inserted (if currently defined). Then copy M digits into the destination field.
05	ICM	insert M if sign is minus; otherwise insert zero suppress character
06	ICP	insert M if sign is plus; otherwise insert zero suppress character.
07	SFC	set floating character to M
10	SFP	set floating character to M if sign plus; otherwise set floating edit character to zero suppress character.
11	SFM	set floating character to M if sign minus; otherwise set floating edit character to zero suppress character.
12	SFS	set floating character to sign

13	JZ	jump M+1 locations ahead in edit sub program if source field equals zero.
14	FS	fill next M characters with zero suppress character
15	SF	set significance flag
16	IS	insert sign

FIELD - Field Operations

These instructions set up and manipulate the field address and length registers. These registers are used by both the decimal and character string instructions. The interpretation of the value in the field length registers depends on the data type and instruction using them.

Store Field Address Register V AP

STFA {1 / Ø} ,ADDR FAR {1 / Ø} ->[EA]32 or [EA]48

Store the contents of the field address register into ADDR as a hardware indirect pointer.

If bit number field of the field address register is zero, store the first two words of the pointer and clear the pointer's extend bit.

If bit number field of the field address register is non-zero, store all three words of the pointer and set the pointer's extend bit.

Transfer L-Register to Field Length Register V GEN

TLFL {Ø / 1} L->[FLR {Ø / 1}]

Transfer the 32-bit unsigned integer in the L register into the selected field length register. The high order 11 bits of L must be zero to make the high order 6 bits of the field length register equal to zero. This instruction is used to load a value computed at execution time into a field length register. The maximum allowable field length is 2**20 (21 bits) - the number of bits in a 64K segment.

Effective Address to Field Address Register V AP

EAFA {1 / Ø} ,ADDR [EA]->[FAR {1 / Ø}]

Place the complete effective address, including the bit portion, in the selected field address register. The associated field length register is unchanged.

Load Field Length Register Immediate V BRAN

LFLI {1 / 0},DATA DATA->[FLR {1 / 0}]

Place the 16-bit unsigned integer in the 2nd word of the instruction into the field length register. Clear the high order bits. This instruction loads a constant which is 65535 or less into a field length register.

The associated field address register is unchanged.

Add Long Integer to Field Address V GEN

ALFA {0 / 1}

Add the 32-bit integer in register L, which represnts an offset in bits, to the 26-bit unsigned word and bit number fields of the field address register. The low-order 26 bits of the sum replace the word and bit number fields of the field address register. All but the low order 20 bits of the sum must be zero.

Example:

To advance FAR 0 by 3 bytes, place 24 into the L register and execute ALFA 0.

Transfer Field Length Register to L V GEN

TFLL {0 / 1}

Transfer the contents of the field length register to the L register as an unsigned 32-bit integer. Clear the high order 11 bits of L.

FLPT - Floating Point Arithmetic

See Section 5 for a description of the processor dependent register formats and the floating point data structures.

Normalization

The result of every floating point calculation is normalized. In normal form, the most significant digit of the mantissa follows the binary point. If an operation produces a mantissa that is smaller than normal, the mantissa is shifted left until the most significant bit differs from the sign bit, and the exponent is decreased by one for each shift. Bits vacated at the right are filled by zeros. If the result of an operation overflows the mantissa, it is shifted right one place, the overflow bit is made the most significant bit, and the exponent is increased by 1.

Floating Point Exceptions

On the Prime 300, error conditions that arise during floating point operations are detected and handled by floating exception (FLEX) interrupt. These are enabled when the software makes physical memory location '74 non-zero (i.e., inserts a pointer to a routine that identifies and processes floating point error conditions). When any one of several types of error occurs, the CPU interrupts through location '74 to the error handlng routine.

If location '74 is zero, FLEX interrupts do not occur; instead, the C bit is set. The user can test the C bit after possible error situations and take action as appropriate.

All FLEX interrupts vector through location '74 and locations '11 and '12 are set in certain cases to indicate the type of error condition. Table 7-2 shows the codes currently assigned.

In the basic arithmetic operations, increasing the exponent in the floating point register beyond 32639 is an overflow; decreasing it below -32896 is an underflow. Note that the exception is detected during an overflow or underflow of the full 16-bit exponent in the floating point register.

An attempt to store a single-precision number with an exponent greater than 127 or less than -128 in the two-word memory format results in a different type of exception. The number in the floating point register is not altered by the FST operation and so can be recovered if necessary.

Other detected exceptions are an attempt to divide by zero or to form an integer exceeding the capacity of the combined A|B register (±30 bits or about ±1 billion decimal).

On the Prime 400/500, the floating point exception is a fault rather than an interrupt. For a discussion of the difference between faults, checks and interrupts, see Section 2.

Table 7-2. Floating Exception Codes

Register 11		Register 12	Type of Exception
Single Pre.	Double Pre.		
$100	$200	--	Overflow/Underflow (Exponent exceeds approx. 10 ±9800)
$101	$201	--	Division by zero
$102	--	(EA)	Attempt to store single precision exponent exceeding 8-bit memory format (>127, <-128)
$103	--	--	Attempt to form INT exceeding capacity of concatenated A and B registers (approx. ±1 billion).

Note: $ indicates hexadecimal codes

Table 7-3. Floating Point Mantissa and Exponent Ranges

Field	Single Precision-Memory	Single Precision-Register	Double Precision
Mantissa			
Bits	23 + Sign	31 + Sign	47 + Sign
Precision	±8,388,607	±2,147,483,647	±140,737,488,355,327
Exponent			
Bits	8	16	16
Range	-128 to +127	-32896 to +32639	-32896 to +32639
	± 38 ($10^{\pm 38}$)	+9,823,-9902 ($10^{+9,823,-9902}$)	+9823,-9902 ($10^{+9823,-9902}$)

Floating Load RV MR

FLD ADDR [EA]32->F

Load the double precision number contained in the two successive words at ADDR into the floating point register.

Floating Store RV MR

FST ADDR F->[EA]32

Store the single precision floating point number contained in the floating point register in two memory words starting at ADDR. Bits 24-31 of the 31 bit mantissa are truncated when written into the 23-bit capacity memory storage. However, the mantissa may be rounded to bit 24 by a FRN instruction which adds 1 to bit 24 if bit 25 is 1. If the floating point register contains an exponent outside the 8-bit range (-128<E<+127), set C or initiate a floating exception.

Floating Add RV MR

FAD ADDR F+[EA]32->F

Add the floating point number at ADDR to the contents of the floating point register and leave the resulting floating point number in the floating point register. Addition of floating point numbers is accomplished by right shifting the smaller number by the difference in the exponents. After alignment, the mantissas are added.

If there is an overflow from the most significant bit (not the sign), the sum mantissa is shifted right one place, the exponent is incremented by one and the overflow bit becomes the high-order bit in the normalized mantissa. If the result is otherwise not in normal form (as when numbers with unlike signs are added), the result is normalized. If there is an exponent under/overflow (<-32896, >+32639) set the C Bit or take a floating exception.

Floating Subtract RV MR

FSB ADDR F-[EA]32->F

Subtract the contents of ADDR from the floating point register by aligning exponents, and proceding as in FAD except that the [EA]32 is subtracted from the floating point register.

Floating Multiply RV MR

FMP ADDR F*[EA]32->F

Multiply the contents of the floating point register by the contents of ADDR and place the product in the floating point register with the mantissa normalized. If there is an exponent under/overflow, the C bit is set or floating exception is initiated.

Floating Divide RV MR

FDV ADDR F/[EA]32->F

Divide the contents of the floating point register by the number in ADDR and place the quotient in the floating point register with the mantissa normalized.

If there is an exponent under/overflow or division by zero, the C bit is set or a floating exception is initiated.

Compare and Skip RV MR

FCS ADDR If F>[EA]32, then PC->PC
If F=[EA]32, then PC+1->PC
If F<[EA]32, then PC+2->PC

If the contents of the floating point register is greater than the contents of ADDR, execute the next instruction.

If the contents of the floating point register equals the contents of ADDR, skip the next location in instruction sequence and execute the instruction at second location following.

If the contents of the floating point register is lss than the contents of ADDR, skip next two locations in instruction sequence and execute

the instruction at third instruction location.

Fix As Fraction	RV	GEN
FRAC	FRAC(F)->(A|B)	

Convert the fractional part of the floating point number in FAC to a binary fraction in the concatenated A and B registers with the binary point between A1 and A2.

Float	RV	GEN
FLOT	Float(A|B)->F	

Take the 31-bit integer in the combined A|B register and convert it into a normalized floating point number in the floating point register.

Convert Integer to Float	V	GEN
FLTA	FLOT(A)->F	

Convert the 16 bit integer in register A to a single precision floating point number in the floating point register.

Convert Long Integer to Float	V	GEN
FLTL	FLOT(L)->FAC	

Convert the 32 bit integer in register L to a single precision floating point number in the floating point register.

Fix as Integer	RV	GEN
INT	Int(F)->A|B	

Convert the integer part of floating point number in the floating point register to a 31 bit integer in the A|B register with the binary point following bit 31. If the floating point register contains a number too

large to be represented in the 31-bit integer format, the C-bit is set or a floating exception is initiated.

Convert Float to Integer V GEN

INTA INT(FAC)->A

Convert the single precision floating point number in the floating point register into a 16 bit in integer in register A. The fractional part of floating point register is lost. Overflow occurs if the value in floating point register is less than -(2**15) or greater than 2**15-1, and sets the C-bit or generates a FLEX.

Convert Float to Long Integer V GEN

INTL INT(FAC)->L

Convert the single precision floating point number in the floating point register into a 32 bit integer in register L. The fractional part of FAC is lost. Overflow occurs if the value in the floating point register is less than -(2**31) or greater than 2**31-1 and sets the C-bit or generates a FLEX.

Complement RV GEN

FCM -F->F

Two's complement the mantissa of the floating point register and normalize if necessary. Overflow sets C or generates a floating exception.

Round Up RV GEN

FRN

If bit 25 of the mantissa in the floating point register is 1, add 1 to bit 24 and reset 25. Overflow sets C or generates a floating exception.

Floating Skip If Zero RV GEN

FSZE

If the floating point register is equal to zero, skip next location.

Floating Skip If Not Zero RV GEN

FSNZ

If the floating point register is not equal to zero, skip next location.

Floating Skip If Minus RV GEN

FSMI

If the floating point register is less than 0, skip next location.

Floating Skip If Plus RV GEN

FSPL

If the floating point register is greater than 0, skip next location.

Floating Skip If Less or Equal Than Zero RV GEN

FSLE

If floating point register is less than or equal to zero, skip next location.

Floating Skip If Greater Than Zero RV GEN

FSGT

If floating point register is greater than zero, skip next location.

Double Precision Floating Load RV MR

DFLD ADDR [EA]64->F

Load the double precision floating point number contained in the four memory words at ADDR into the floating point register.

Double Precision Floating Store RV MR

DFST ADDR F->[EA]64

Store the double precision floating point number contained in the floating point register into the location specified by ADDR. Exponent and mantissa bit capacities are the same so that no floating point exceptions are possible.

Double Precision Floating Add RV MR

DFAD ADDR F+[EA]64->F

Add the double precision number starting at ADDR to the double precision number in the floating point register and leave the result in the floating point register. (Same procedure as FAD except a 47-bit mantissa is produced.)

Double Precision Floating Subtract RV MR

DFSB ADDR F-[EA]64->F

Subtract the double precision floating point number starting at ADDR from the double precision floating point number in the floating point register. (Same procedure as FSB except a 47-bit mantissa is produced.)

Double Precision Floating Multiply RV MR

DFMP ADDR F*[EA]64->F

Multiply the contents of the floating point register by the contents of ADDR and place the products in the floating point register with the mantissa normalized. If there is an exponent under/overflow, the C bit is set or floating exception is initiated.

Double Precision Floating Divide RV MR

DFDV ADDR F/[EA]64->F

Divide the contents of the floating point register by the number in ADDR and place the quotient in the floating point register with the mantissa normalized.

If there is an exponent under/overflow or division by zero, the C bit is set or a floating exception is initiated.

Double Precision Floating Point Compare and Skip RV MR

DFCS ADDR if F>[EA]64 then PC->PC
if F=[EA]64 then PC+1->PC
if F<[EA]64 then PC+2->PC

If the contents of the floating point register is greater than the contents of ADDR, execute the next instruction.

If the contents of the floating point register equals the contents of ADDR, skip the next location in instruction sequence and execute the instruction at second location following.

If the contents of the floating point register is less than the contents of ADDR, skip next two locations in instruction sequence and execute the at third instruction location following.

Double Precision Floating Complement RV GEN

DFCM -F->F

Two's complement the precision mantissa in floating point register and normalize if necessary. Overflow sets C or generates a floating exception.

Floating Load Index RV MR

FLX ADDR [EA]16*2->X

Double the contents of the effective address and load the result into the index register X. This instruction facilitates indexing sequences that involve double-word memory reference operations. It works directly for two-word indexing, e.g., 31-bit or 32-bit integer or floating point.

Double Floating Load Index V MR

DFLX ADDR [EA]16*4->X

Quadruple the contents of the effective address and load the result into the index register X. This instruction is useful for addressing arrays or tables of element size four words.

Convert Single to Double Float V GEN

FDBL F->F

Convert the single precision floating point number in the floating point register to a double precision precision floating point number in the floating point register.

INT - Integer Arithmetic

These instructions operate on 16, 31-bit and 32-bit signed integers.

Single Precision	SR	GEN

SGL

Return to single precision mode. Subsequent LDA, STA, ADD and SUB instructions handle 16-bit integers.

Add	SRV	MR

ADD ADDR A+[EA]16->A

Add the 16-bit integer at ADDR to the 16-bit integer in register A and put the result into register A. If the sum is greater than 2**15 or less than or equal to -2**15, set C; otherwise, clear C. In the first overflow case, the result has a minus sign, but a magnitude in positive form equal to the sum minus 2**15; in the second, the result has a plus sign, but a magnitude in negative form equal to the sum plus 2**15.

Add One to A	SRV	GEN

AlA A+1->A

Add 1 to the 16-bit integer in register A and put the result into A. If the number incremented is 2**15-1, set C and give a result of -2**15; otherwise clear C.

Add 2 to A	SRV	GEN

A2A A+2->A

Add 2 to the 16-bit integer in register A and put the result into A. If the number incremented is 2**15-2 or 2**15-1, set C and give a result of -2**15 or -(2**15-1); otherwise clear C.

Subtract	SRV	MR
SUB ADDR	A-[EA]16->A	

Subtract the 16-bit integer at ADDR from the 16-bit integer in register A and put the result into register A. If the difference is >2**15 or <-2**15, set C; otherwise clear C. In the first overflow case, the result has a minus sign but a magnitude in positive form equal to the difference minus 2**15; in the second, the result has a plus sign but a magnitude in negative form equal to the difference plus 2**15.

Subtract One from A	SRV	GEN
S1A	A-1->A	

Subtract 1 from the 16-bit integer in register A and put the result into A. If the number decremented is -2**15, set C and give a result of 2**15-1; otherwise clear C.

Subtract 2 from A	SRV	GEN
S2A	A-2->A	

Subtract 2 from the 16-bit integer in register A and put the result into A. If the number decremented is -(2**15-1) or -2**15, set C and give a result of 2**15-2; otherwise clear C.

Multiply	SR	MR
MPY ADDR	A*[EA]16->A|B	

Multiply the 16-bit integer in register A by the 16-bit integer at ADDR, and put the 31-bit integer result into registers A and B. If both the multiplier and multiplicand are -2**15 then set C; otherwise clear C.

Position Following Integer Multiply SR GEN

PIM B(2-16)->A(2-16)

Convert the 31-bit integer in registers A|B to a 16-bit integer in A by moving bits 2-16 of B into bits 2-16 of A. Overflow if a loss of precision would result.

Normalize SR GEN

NRM A1 A2...A16 B1 B2...B16 <- 0

Shift the 31-bit integer in registers A and B left arithmetically, bringing zeros into bit 16 of B, bypassing bit 1 of B, leaving bit 1 of register A unaffected, and dropping bits out of bit 2 of register A until bit 2 of register A is in the state opposite that bit 1 of register A. Since the only data shifted out of bit 2 of register A is equal to the sign, no information is lost. Place the number of shifts performed in bits 9-16 of the keys.

Load Shift Count into A SR GEN

SCA keys(9-16)->A(9-16)
0->A(1-8)

Load the contents of bits 9-16 of the keys into bits 9-16 of register A and clear bits 1-8 of register A.

By shifting until bit 2 differs from the sign, normalization produces a fraction in the range 1/2 to (1-LSB) or -(1/2+LSB) to -1. Saving the number of shifts allows the program to determine any change in the order of magnitude of a result due to a fixed point operation on the fractions of floating point operands. The program can then use the information stored in the keys to adjust the exponent. Finally, the result is put in proper format by shifting the fraction to the correct position and inserting the exponent in the high order word.

Divide	SR	MR

DIV ADDR A|B/[EA]16->A;REM->B

Divide the 31-bit integer in register A|B by the 16-bit integer at ADDR and put the quotient into A, and the remainder into B. Barring overflow, the results are defined such that A*[ADDR]+B equals the original A|B and the remainder in B has the same sign as the dividend. Hence, -42 divided by 5 gives A=-8 and B=-2. Overflow occurs (and the C-bit is set) whenever the quotient is less than -(2**15) or greater than 2**15-1; the A|B register is unchanged.

•

Position for Integer Divide	SR	GEN

PID A(2-16)->B(2-16)
Ø->B(1);A(1)->A(2-16)

Convert the 16-bit integer in register A to a 31-bit integer in A|B by moving the contents of bits 2-16 of register A to bits 2-16 of register B, clearing bit 1 of register B and extending the sign in bit 1 of A through bits 2-16 of A.

Double Precision	SR	GEN

DBL

Enter double precision mode. Subsequent LDA, STA, ADD and SUB instructions handle 31-bit integers.

Double Add SR MR

DAD ADDR A|B+[EA]31->A|B

Add the 31-bit integer at ADDR and ADDR+1 to the 31-bit integer in registers A|B, and put the result into A|B. If the sum is ≥2**30 or <-2**30, set C; otherwise, clear C. In the first overflow case, the result has a minus sign but a magnitude in positive form equal to the sum minus 2**30; in the second, the result has a plus sign but a magnitude in negative form equal to the sum plus 2**30.

By definition, bit 1 of the low order word of a 31-bit integer must be 0. The instruction executes only in double precision mode.

Double Subtract SR MR

DSB ADDR A|B-[EA]31->A|B

Subtract the 31-bit integer at ADDR and ADDR+1 from the 31-bit integer in registers A|B, and place the result into A|B. If the difference is ≥2**30 or <-2**30, set C; otherwise, clear C. In the first overflow case, the result has a minus sign but a magnitude in positive form equal to the difference minus 2**30; in the second, the result has a plus sign but a magnitude in negative form equal to the difference plus 2**30.

Bit 1 of the low order word of a 31-bit integer must be 0. The instruction executes only in double precision mode.

To negate one 31-bit integer, simply subtract it from zero.

Two's Complement A SRV GEN

TCA -A->A

Form the twos complement of the contents of register A and put the result into register A. If the result is -2**15, set C and give a result of -2**15; otherwise clear C.

Add C-Bit to A | SRV | GEN

ACA A+C-bit->A

Add the C-bit to the contents of register A and put the result into A (C is treated as same order of magnitude as bit 16 of A). If the number originally in A is 2**15-1, set C and give a result of -2**15; otherwise clear C.

Set Sign Plus | SRV | GEN

SSP Ø->A(1)

Clear bit 1 of register A without affecting the rest of the register.

Set Sign Minus | SRV | GEN

SSM 1->A(1)

Set bit 1 of register A to one without affecting the rest of the register.

Change Sign | SRV | GEN

CHS -A(1)->A(1)

Complement bit 1 of register A without affecting the rest of the register.

Copy Sign of A | SRV | GEN

CSA A(1)->C-bit
Ø->A(1)

Make C equal to bit 1 of register A and clear bit 1 of A without affecting the rest of the register. Used when using single precision arithmetic to do double precision work.

Add Long V MR

ADL ADDR L+[EA]32->L

Add the 32-bit integer at ADDR to the 32-bit integer in register L and put the result into L. If the sum is greater than 2**31 or less than-2**31, set C; otherwise, clear C. In the first overflow case, the result has a minus sign, but a magnitude in positive form equal to the sum minus 2**31; in the second, the result has a plus sign, but a magnitude in negative form equal to the sum plus 2**31.

Subtract Long V MR

SBL ADDR L-[EA]32->L

Subtract the 32-bit integer at ADDR from the 32-bit integer in register L and put the result into the L register. If the difference is greater than +2**31 or less than -2**31, set C; otherwise clear C. In the first overflow case, the result has a minus sign but a magnitude in positive form equal to the difference minus 2**31; in the second, the result has a plus sign but a magnitude in negative form equal to the difference plus 2**31.

Multiply V MR

MPY ADDR A*[EA]16->L

Multiply the 16-bit integer in register A by the 16-bit integer at ADDR, and put the 32-bit integer result into L. This operation never overflows because there is always room for the product.

Position Following Integer Multiply V GEN

PIMA L(17-31)->A(1-16)

Convert the 32-bit integer in L to a 16-bit integer in register A by moving bits 17-32 of L into bits 1-16 of A. Overflow if a loss of precision would result.

Multiply Long V MR

MPL ADDR L*[EA]32->L|E

Multiply the 32-bit integer in register L by the 32-bit integer at ADDR, and put the 64-bit integer result into L|E. This operation never overflows becuase there is always room for the product.

Position Following Integer Multiply-Long V GEN

PIML L|E(33-64)->L(1-32)

Convert the 64-bit integer in registers L|E to a 32-bit integer in L by moving bits 33-64 of register L|E into bits 1-32 of register L. Overflow if a loss of precision would result.

Divide V MR

DIV ADDR L/[EA]16->A;REM->B

Divide the 32-bit integer in register L by the 16-bit integer at ADDR and put the quotient into A, and the remainder into B. Barring overflow, the results are defined such that A*[ADDR]+B equals the original L and the remainder in B has the same sign as the dividend. Hence, -42 divided by 5 gives A=-8 and B=-2.

Overflow occurs (and the C-bit is set) whenever the quotient is less than -(2**15) or greater than 2**15-1. The PIDA instruction is useful for placing 16-bit dividends into L.

Divide Long V MR

DVL ADDR L|E/[EA]32->L;REM->E

Divide the 64-bit integer in registers L|E by the 32-bit integer at ADDR and put the quotient into L, and the remainder into E. Barring overflow, the results are defined such that L*[ADDR]+E equals the original L|E and the remainder in E has the same sign as the dividend. Hence, +42 divided by -5 gives L=-8 and E=+2.

Overflow occurs (and the C-bit is set) whenever the quotient is less than -(2**31) or greater than 2**31-1.

Position For Integer Divide V GEN

PID A(1-16)->L(17-32)
A(1)->A(2-16)

Convert the 16-bit integer in register A to a 32-bit integer in register A by moving bits 1-16 of A to bits 17-32 of L and extending the sign in bit 1 of L through bits 2-16 of A.

Position For Integer Divide-Long V GEN

PIDL L->E
L(1)->L(2-32)

Convert the 32-bit integer in register L to a 64-bit integer in register E by moving the contents of L to E and extending the sign in bits 1 of L through bits 2-32 of L. PIDL is useful for placng 32 bit operands in L|E.

Two's Complement Long V GEN

TCL -L->L

Form the two's complement of the contents of register L and put the result into L. If the result is -2**31, set C and give a result of -2**31; otherwise clear C.

Add L bit to L V GEN

ADLL L+keys(L)->L

Add the link bit (L-bit in the keys) to the contents of the L register and put the result into the L register. Overflow may be set.

This instruction is useful in implementing multiple precision arithmetic.

COMPARE V MR

CLS ADDR if L>[EA]32 then PB+1->PB
if L=[EA]32 then PB+2->PB

```
if L<[EA]32 then PB+3->PB
```

If the contents of the L register is greater than the contents of ADDR, execute the next instruction.

If the contents of the L register equals the contents of ADDR, skip the next location in instruction sequence and execute the instruction at second location following.

If the contents of the L register is less than the contents of ADDR, skip next two locations in instruction sequence and execute the instruction at third location following.

Compare A and Skip — SRV — MR

```
                if A>[EA]16 then PC=PC
CAS ADDR        if A=[EA]16 then PC+1->PC
                if A<[EA]16 then PC+2->PC
```

If the contents of the A register is greater than the contents of ADDR, execute the next instruction.

If the contents of the A register equals the contents of ADDR, skip the next location in instruction sequence and execute the instruction at the second location following.

If the contents of the L register is less than the contents of ADDR, skip the next two locations in instruction sequence and execute the instruction at the third location following.

Compare A with Zero — SRV — GEN

```
                if A>0 then PC=PC
CAZ             if A=0 then PC+1->PC
                if A<0 then PC+2->PC
```

If the contents of the A register is greater than zero, execute the next instruction.

If the contents of the A register is equal to zero, skip the next location in instruction sequence and execute the instruction at second location following.

If the contents of the A register is less than zero, skip the next location in instruction sequence and execute the instruction at third location following.

INTGY - Hardware Integrity Check

Verify	SRV	GEN
VIRY		R

Execute the verification routine, and if there is a failure of any kind, go on to the next instruction with the number of the test that failed in register A. If there are no errors, skip the next instruction in sequence.

If the processor does not have the verification routine, this instruction executes as no-op.

Enter Machine Check Mode	SRV	GEN
EMCM		R

In machine check mode the microprogram responds to a machine parity error by causing a machine check interrupt if there is a non-zero vector in the interrupt locaion. If this locaion is zeo the machine halts.

Leave Machine Check Mode	SRV	GEN
LMCM		R

A machine parity error sets the machine check flag, but does not cause a check.

Clear Machine Check	SRV	GEN
RMC		R

Clear the machine check flag.

Skip on Machine Check Reset	SRV	GEN
SMCR		R

If the machine check flag is zero (indicating no machine detected

parity error), skip the next instruction in sequence. (When the processor is in machine check mode, this instruction has no meaning and executes as skip.)

Skip on Machine Check Set	SRV	GEN
SMCS		R

If the machine check flag is set (indicating a machine detected parity error), skip the next instruction in sequence. (When the processor is in machine check mode, this instruction has no meaning and executes as a NOP).

Inhibit Interleave	V	GEN
MDII		R

Inhibit the memory diagnostic interleave capability.

Write Interleaved	V	GEN
MDIW		R

Write interleaved memory diagnostic

Read Syndrome Bits	V	GEN
MDRS		R

Read memory diagnostic syndrome bits

Load Write Control Register	V	GEN
MDWC		R

Write memory diagnostic control register

Verify the XIS Board V GEN

VXIS R

VXIS executes a Prime 500 microcode diagnostic routine that checks the integrity of the XIS board. If the XIS board is not functional, the machine will not skip the next instruction and the A register will hold the failed micro-diagnostic test number. If the machine passes the verify instruction, the next instruction is skipped.

The codes and tests are:

' 72 Data Move Test - Load and Unload XIS Board

' 73 Normalize Test - Adjust Test

' 74 Binary Multiply

' 75 Binary Divide

' 76 Decimal Arithmetic

I/O - Input/Output

Output from A	SR	PIO
OTA FUNC|DEV		R

Output data from register A to DEV. FUNC tells the device which operation to perform. If the device does not respond ready, then do not perform the transfer but instead execute the next instruction in sequence. If the device responds ready, then perform the transfer and skip the next instruction in sequence. The processor sends the contents of register A to DEV which performs whatever control operations are appropriate to the function and the device.

The number of bits actually accepted by the device depends on the type of information, the size of the device register, the mode of operation, etc. The contents of register A are unaffected.

An OTA instruction for any device except device '20 uses a ready test and the skipping procedure as stated in the description of the instruction. An OTA to device '20 makes no test and does not skip.

Skip if Satisfied	SR	PIO
SKS FUNC|DEV		R

FUNC (bits 7-10) defines a condition to be tested by the SKS. When the condition is satisfied, the device specified by DEV (bits 11-16) responds ready, and the next instruction in sequence is skipped.

Input to A	SR	PIO
INA FUNC|DEV		R

Input data from device DEV into register A. FUNC determines the type of data. If the device does not respond ready, then do not perform the transfer, but execute the next instruction in sequence. If the device responds ready, then perform the transfer specified by FUNC and skip the next instruction in sequence. To perform the function specified by FUNC, the processor reads the information from DEV into register A and performs whatever control operations are appropriate to the function and the device. Depending on FUNC, the information read may be data, status, an address, a word count, or anything else.

The number of bits brought into register A depends on the type of

information, the size of the device register, the mode of operation, etc.

INA instructions for any device except device '20 use a ready test and skip the next instruction if the device was ready.

Output Control Pulse	SR	PIO
OCP FUNC|DEV		R

Send a control pulse for the function specified by FUNC (bits 7-10) to the device specified by DEV (bits 11-16). This instruction never skips and is used for such functions as initializing a disk controller, or starting a transfer.

INTERRUPT PROGRAMMING

The instructions that control the interrupt system are all of the type with a full word op-code, but associated with the system are two I/O instructions that deal with the mask used for setting up the interrupt enable flags in certain devices. When power is turned on or the computer is cleared from the control panel, the processor is automatically in standard interrupt mode with interrupts inhibited.

Enable Interrupt	SRV	GEN
ENB		R

Enable the external interrupt system so the processor will respond to interrupt requests over the I/O bus. This instruction takes effect following execution of the next sequential instruction.

Inhibit Interrupts	SRV	GEN
INH		R

Inhibit the external interrupt system so the processor will not respond to interrupt requests over the I/O bus. This instruction takes effect immediately.

Enter Standard Interrupt Mode SRV GEN

ESIM

Enter standard interrupt mode so that all interrupts are made through location '63.

Enter Vectored Interrupt Mode SRV GEN

EVIM

Enter vectored interrupt mode so that the interrupt priority of a device is determined by its position on the I/O bus (with lower devices having higher priority) and each interrupt is made through the location specified by the interrupting device.

Clear Active Interrupt SRV GEN

CAI

Terminate the presently active interrupt so that the processor can recognize interrupt requests from devices of lower priority (in higher slots) than the device for which the current interrupt is being held. This instruction is effective only in vectored interrupt mode.

Send Mask SR PIO

SMK R

Set up the interrupt enable flags in the device according to the mask in register A (a 1 in a mask bit sets the flag in the device corresponding to that bit; a 0 clears it). Note that this instruction is equivalent to OTA '0020 and never skips.

The bits in the mask and the devices assigned to them are as follows (note that the mask does not necessarily control the interrupt enable flags in all devices):

1	
2	
3	
4	Moving head disk (certain models)
5	
6	
7	
8	Fixed head disk
9	Paper tape reader
10	Paper tape punch
11	Teletypwriter
12	
13	
14	
15	
16	Real time clock

This instruction and IMK are included only for compatibility.

Input Mask	SR	PIO
IMK		R

Input interrupt mas bits to A register from all implemented devices. Used to see which devices have interrupts enabled. For compatibility only.

Execute I/O	V	MR

EIO ADDR

Perform the I/O instruction represented by the effective address, e.g., X = '04 EIO '131000,X will execute an INA with FUNC = '10 and DEV = '04.

KEYS-Status Keys

See Section 5 for the format of the keys.

Input Keys	SR	GEN
INK	keys->A	

Read the keys into register A.

Output Keys	SR	GEN
OTK	A->keys	

Set up the keys from the contents of register A. Each bit position in register A corresponds to the bit position in the keys, e.g., bit 1 of register A becomes the C-bit in the keys.

Set C-Bit	SRV	GEN
SCB	1->C	

Set the C-bit in the keys.

Reset C-Bit	SRV	GEN
RCB	0->C	

Clear the C-bit in the keys.

Transfer A to Keys	V	GEN
TAK	A->keys	

Transfer the contents of register A to the keys register. If the new value of the keys specifies a different addressing mode, note that the new mode takes effect on the next instruction.

Transfer Keys to A V GEN

TKA keys->A

Transfer the contents of the keys register to register A.

LOGIC - Logical Operations

Exclusive OR to A | SRV | MR

ERA ADDR | A.XOR.[EA]16->A

EXCLUSIVE OR the contents of location ADDR with the contents of register A and place the result in register A. A given bit of the result is 1 if the corresponding bits of the operands differ; otherwise the resulting bit is 0.

A Bit	Memory Bit	Resulting Bit
0	0	0
0	1	1
1	0	1
1	1	0

Complement A | SRV | GEN

CMA | .NOT. A->A

Form the ones complement of the contents of register A and put the result in register A. Each one becomes a zero; each zero becomes a one.

AND to A | SRV | MR

ANA ADDR | A.AND.[EA]16->A

AND the contents of location ADDR with the contents of register A and place the result in register A. A given bit of the result is 1 if the corresponding bits of both operands are 1; otherwise the resulting bit is 0.

A Bit	Memory Bit	Resulting Bit
0	0	0
0	1	0
1	0	0
1	1	1

AND Long V MR

ANL ADDR L.AND.[EA]32->L

AND the contents of register L with the 32-bit quantity at ADDR, putting the result in L.

Inclusive OR V MR

ORA A.OR.[EA]16->A

INCLUSIVE OR the contents of register A with the 16-bit quantity at ADDR, putting the result in A.

Exclusive OR Long V MR

ERL ADDR L.XOR.[EA]32->L

EXCLUSIVE OR the contents of register L with the 32-bit quantity at ADDR, putting the result in L.

LTSTS - Logical Test and Set

Logical Test and Set (Logicize) SRV GEN

If the test is satisfied, then set the A register equal to 1.

If the test is not satisfied, then set the register equal to Ø.

These instructions simplify the analysis of complex logical expressions.

For example, LLT means if the contentss of the A register is less than Ø, then set A equal to one; else clear A.

LLT	if CC < Ø, then 1->A; else Ø->A
LLE	if CC ≤ Ø, then 1->A; else Ø->A
LEQ	if CC = Ø, then 1->A; else Ø->A
LNE	if CC ≠ Ø, then 1->A; else Ø->A
LGE	if CC ≥ Ø, then 1->A; else Ø->A
LGT	if CC > Ø, then 1->A; else Ø->A

Logic set A True

LT 1->A

Set A equal to one.

Logic set A False

LF Ø->A

Set A equal to zero.

Logical Test and Set (Logicize) V GEN

If the test is satisfied, then set the A register equal to 1.

If the test is not satisfied, then set the A register equal to Ø.

The logical test and set instructions simplify the analysis of complex logical expressions.

Their format is:

If { Condition Code / L Register / Floating Point Register } { LT / LE / EQ / NE / GE / GT } Ø, then 1 -> A; else Ø -> A

For example: LCLT means if the condition code is less than zero (LT bit is set and EQ is cleared), set the A register equal to 1 else set A equal to Ø.

LCLT	if CC < Ø, then 1->A; else Ø->A
LCLE	if CC $\leq$ Ø, then 1->A; else Ø->A
LCEQ	if CC = Ø, then 1->A; else Ø->A
LCNE	if CC $\neq$ Ø, then 1->A; else Ø->A
LCGE	if CC $\geq$ Ø, then 1->A; else Ø->A
LCGT	if CC > Ø, then 1->A; else Ø->A
LLLT	if L < Ø, then 1->A; else Ø->A
LLLE	if L $\leq$ Ø, then 1->A; else Ø->A
LLEQ	if L = Ø, then 1->A; else Ø->A
LLNE	if L $\neq$ Ø, then 1->A; else Ø->A
LLGE	if L $\geq$ Ø, then 1->A; else Ø->A
LLGT	if L > Ø, then 1->A; else Ø->A
LFLT	if F < Ø, then 1->A; else Ø->A
LFLE	if F $\leq$ Ø, then 1->A; else Ø->A
LFEQ	if F = Ø, then 1->A; else Ø->A
LFNE	if F $\neq$ Ø, then 1->A; else Ø->A
LFGE	if F $\geq$ Ø, then 1->A; else Ø->A
LFGT	if F > Ø, then 1->A; else Ø->A

MCTL - Machine Control

Enter Paging Mode and Jump (Prime 300)	SR	MR
EPMJ ADDR	EA->PC	R

EPMJ is a two-word instruction. The first word is the opcode; the second word contains a 16-bit address pointing to the final effective address which is transferred to the program counter; the associative memory registers are cleared, and paging mode is enabled.

Leave Paging Mode and Jump (Prime 300)	SR	MR
LPMJ ADDR	EA->PC	R

LPMJ is a two-word instruction. The first word is the opcode; the second word contains a 16-bit address pointing to the final effective address which is transferred to the program counter. Paging mode is disabled.

Enter Restricted Execution Mode and Jump (Prime 300)	SR	MR
ERMJ ADDR	EA->PC	R

ERMJ is a two-word instruction. The first word is the opcode; the second word contains a 16-bit address pointing to the final effective address which is transferred to the program counter; restricted execution mode is enabled, and interrupts are enabled.

Enter Virtual Mode and Jump (Prime 300)	SR	MR
EVMJ ADDR	EA->PC	R

EVMJ is a two-word instruction that has the effect of an EPMJ and ERMJ combined. The first word in the opcode; the second word contains a 16-bit address pointing to the final effective address which is transferred to the program counter; the associative memory registers are cleared, paging mode is enabled, restricted execution mode is enabled, and interrupts are enabled.

Enter Paging Mode and Jump to XCS (Prime 300) SR MR

EPMX ADDR EA->PC

EPMX is a two-word instruction. The first word is the opcode; the second word contains a 16-bit pointer to the location of the micro-instruction. Paging is enabled.

Enter Restricted Execution Mode and Jump to XCS (Prime 300) SR MR

ERMX ADDR EA->PC

ERMX is a two word instruction. The first is the opcode. The second word contains a 16-bit pointer to the location of the micro instruction. Restricted execution mode and interrupts are enabled.

Leave Paging Mode and Jump to XCS (Prime 300) SR MR

LPMX ADDR EA->PC

LPMX is a two-word instruction. The first word is the opcode. The second word contains a 16-bit pointer to the location of the micro instruction. Paging is disabled.

Enter Virtual Mode and Jump to XCS (Prime 300) SR MR

EVMX ADDR EA->PC

EVMX is a two-word instruction. The first word, that has the effect of an EPMX and ERMX combined, is the opcode; the second word contains a 16-bit pointer to the location of the micro-instruction. Paging, interrupts, and restricted execution mode are enabled.

Supervisor Call SRV GEN

SVC

An addressing mode independent method of making an operating system request. It is also independent of operating system. The call protocol is such that an operation code (request) followed by argument pointers (16-bit word number - on the Prime 400/500, segment number is

the segment in which the SVC resides) is made available to the operating system. PRIMOS II, III, IV and V have defined a uniform set of operation codes to provide operating system independent services. See PDR3108, System Commands, for definition of op codes and arguments.

Note

On the Prime 100-300, the SVC is treated as an interrupt. On the Prime 350-500, the SVC is treated as a fault.

Halt	SRV	GEN

HLT

Halt the processor with the STOP indicator lit on the control panel and the program counter pointing to the next instruction in sequence (the instruction that would have been executed had the HLT been replaced by a no-op). The data lights display the next instruction.

No Operation	SRV	GEN

NOP PC+1->PC

Do nothing, but go on to the next instruction.

Save Registers	V	AP

RSAV ADDR

Save the general, floating and XB registers in the save area starting at location ADDR. Only those general and floating point registers which are not zero are saved. A save mask is generated which identifies the registers which are not zero. Registers which are zero are not stored into the save area; their location remains untouched.

The format of the RSAV area is:

Save Mask

RF13H
RF13L
RF12H
RF12L

RF11H
RF11L
RF10H
RF10L
RF07H
RF07L
RF06H
RF06L
RF05H
RF05L
RF04H
RF04L
RF03H
RF03L
RF02H
RF02L
RF01H
RF01L
RF00H
RF00L
XBH
XBL

legend:

RFxxH = Current register file location xx High Half

RFxxL = current register file location xx Low Half.

Save Mask:

	R	R	R	R	R	R	R	R	R	R	R	R
MBZ	13	12	11	10	7	6	5	4	3	2	1	0
1-4	5	6	7	8	9	10	11	12	13	14	15	16

The size of the RSAV area varies between 3 words and 27 words.

Restore Registers V AP

RRST ADDR

Restore the general, floating and XB registers from the save area starting at location ADDR. The format of the save area is as for RSAV, above.

Invalidate STLB Entry	V	GEN
ITLB		R

Invalidate the Segmentation Translation Lookaside Buffer (STLB) entry whose address is in L. This instruction must be executed whenever the page table entry for the given address is changed.

If a Segment Descriptor Word (SDW) or a Descriptor Table Address Register (DTAR) is changed, usually the entire STLB must be invalidated. This can be done by executing ITLB once for each page of any single segment (except segment 0).

If the segment number portion of L is zero, the I/O T.L.B. entry corresponding to address L is invalidated.

Load Process ID	V	GEN
LPID	A->RPID	R

Load the process id register from bits 1-12 of Register A.

Load Program Status Word	V	AP
LPSW ADDR		R

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 at location ADDR containing a 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 dos 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 010 (the processor is always initialized to 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 eload 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.

Control Extended Control Store V GEN

CXCS

Move the A register to control register on writable control store board.

Microcode Indirect A V MR

MIA ADDR

Microcode entrance.

Microcode Indirect B V MR

MIB ADDR

Microcode entrance.

Writable Control Store RV GEN

WCS

Reserved set of 64 op codes to serve as microcode entrances.

Load Writable Control Store V GEN

LXCS

Load writable control store portion of extended control store board from the memory block pointed to by XB. The control register loaded by CXCS modifies this instruction.

MOVE - Move Data

Interchange Characters in A	SRV	GEN
ICA	A(1-8) <-> A(9-16)	

Interchange the two bytes (characters) of register A (move the contents of bits 1-8 to bits 9-16 and the contents of bits 9-16 to bits 1-8).

Interchange and Clear Left	SRV	GEN
ICL	A(1-8)->A(9-16) Ø->A(1-8)	

Interchange the two bytes of register A and then clear the left byte (bits 1-8).

Interchange and Clear Right	SRV	GEN
ICR	A(1-16)->A(1-8) Ø->A(1-16)	

Move the contents of register A bits 9-16 to bits 1-8 and clear bits 1-16. The original contents of bits 1-8 are lost.

Interchange the A and B Registers	SRV	GEN
IAB	A<--->B	

Move the contents of register A to register B and the contents of register B to register A.

Exchange and Clear the A Register	SRV	GEN
XCA	A->B Ø->A	

Exchange (swap) the A and B registers; then clear A.

Exchange and Clear the B Register | SRV | GEN

XCB — B->A, 0->B

Exchange (swap) the B and A registers; then clear register B.

Load the A Register | SRV | MR

LDA ADDR — [EA]16->A

Load the contents of location ADDR into the A register. The contents of ADDR are unaffected; the previous contents of the A register are lost.

Store the A Register | SRV | MR

STA ADDR — A->[EA]16

Store the contents of the A register in location ADDR. The contents of the A register are unaffected; the previous contents of ADDR are lost.

Interchange Memory and the A Register | SRV | MR

IMA ADDR — [EA]16<->A

Store the contents of the A register in location ADDR and load the original contents of location ADDR into the A register; the original contents of the A register and ADDR are swapped.

Load Index Register SRV MR

LDX ADDR [EA]16->X

Load the contents of location ADDR into the index register. The contents of ADDR are unaffected, the previous contents of the index register are lost. This instruction cannot itself specify indexing, although an address word retrieved in the effective address calculation may do so in 16S mode.

Store Index Register SRV MR

STX ADDR X->[EA]16

Store the contents of the index register in location ADDR. The contents of the index register are unaffected and the previous contents of ADDR are lost. This instruction cannot itself specify indexing, although an address word retrieved in the effective address calculation may do so in 16S mode.

Double Load SR MR

DLD ADDR [EA]32->A|B

Load the contents of location ADDR into register A and the contents of location ADDR+1 into register B. The contents of memory are unaffected, the original contents of registers A and B are lost. This instruction executes only in double precision mode.

Double Store SR MR

DST ADDR A|B->[EA]32

Store the contents of register A in location ADDR and the contents of register B in location ADDR+1. The contents of registers A and B are unaffected, the original contents of the specified memory locations are lost. This instruction executes only in double precision mode.

Transfer A to B V GEN

TAB A->B

Move the contents of A to B.

Transfer B to A V GEN

TBA B->A

Move the contents of B to A.

Transfer A to X V GEN

TAX A->X

Move the contents of A to X.

Transfer X to A V GEN

TXA X->A

Move the contents of X to A.

Transfer A to Y V GEN

TAY A->Y

Move the contents of A to Y.

Transfer Y to A V GEN

TYA Y->A

Move the contents of Y to A.

Store L Into Addressed Register V MR

STLR ADDR L->register(EA)W

Stores the contents of L into the register location specified by ADDR. There are three cases of this instruction which are summarized below. Only the word portion of the effective address, ((EA)W), is used.

Ring 0 and Bit 2 of (EA)W = 1 R

(EA)W(10-16) - Absolute register number from 0 to '177.

Ring 0 and Bit 2 of (EA)W = 0 R

(EA)W(12-16) - Register 0-37 in current register set.

Ring other than 0

(EA)W (1-12) must = 0

(EA)W(13-16) - Register 0-17 in current register set.

Load L From Addressed Register V MR

LDLR ADDR register(EA)W->L

Copies the contents of the register specified by the word number portion of ADDR into L. There are three cases of this instruction which are summarized below. Only the word portion of the effective address, (EA)W, is used.

Ring 0 and Bit 2 of (EA)W = 1 R

(EA)W(10-16) - Absolute register number from 0-'177.

Ring 0 and Bit 2 of (EA)W = 0 R

(EA)W - Register 0-'37 in the current register set.

Ring Other than 0

(EA)W(1-12) must = 0.

(EA)W(13-16) - Register 0-'17 in the current register set.

Interchange L and E | V | GEN

ILE | L<-->E

Swap the contents of registers L and E.

Load Long | V | MR

LDL ADDR | [EA]32->L

Move the 32-bit quantity at location ADDR to register L.

Load Y | V | MR

LDY ADDR | [EA]32->Y

Move the 16 bit quantity at location ADDR to register Y. Cannot be indexed.

Store Long | V | MR

STL ADDR | L->[EA]32

Store the contents of register L into the 32-bit long word at location ADDR.

Store Y | V | AP

STY ADDR | Y->[EA]32

Store the contents of Y into the location specified by ADDR. Cannot be indexed.

Store A Conditionally V AP

STAC ADDR if [EA]16=B then A->[EA]16

Store the contents of A into location ADDR, if and only if, the contents of location ADDR equals the contents of B.

The comparison and store are guaranteed not to be separated by the execution of any other cpu 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".

Store L Conditionally V MR

STLC ADDR if [EA]32=E then L->[EA]32

Store the contents of L into the 32-bit location at ADDR if and only if the contents of location ADDR equals the contents of E.

STLC and STAC 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.

Both of these instructions are interlocked against direct-memory input/output. Hence, these instructions may be used to interlock a process with a DMA, DMC or DMQ channel, or to interlock a memory location possibly being accessed by I/O.

PCTLJ - Program Control and Jump

Compute Effective Address	SR	GEN

CEA

Interpret the contents of the A register as a 16-bit indirect address word in the current addressing mode, calculate the effective address, and place the final effective address back in the A register.

Jump	SRV	MR
JMP ADDR	EA->PC	

Transfer conrol to location ADDR by loading ADDR into the program counter and continue sequential operation from that location.

Jump and Store	SR	MR
JST ADDR	PC->[EA]16 EA+1->PC	

Call a subroutine by storing the contents of the program counter (which points to the next location after the JST instruction) in location ADDR. Continue execution at location ADDR+1. In non-restricted mode, interrupts are inhibited for one instruction cycle following a JST.

The return address is truncated according to the addressing mode before it is stored, and higher-order bits of the memory location are not altered. It is thus possible to preset the I or X bits of such locations:

Mode	Preset Allowed
16S	I, X
32S, 32R	I
64R	-

The usual procedure for calling a subroutine is to use a JST with an effective address that specifies the subroutine's starting location. Since the return address is saved at the entry point, a subsequent return can be made to the instruction following the JST by an indirect JMP through the entry point.

Note

Cannot be used in shared code.

Jump If Equal to Zero R MR

JEQ ADDR if A=Ø, then EA->PC

If the contents of the A register are equal to zero, then load ADDR into the program counter and continue sequential operation from that location.

Jump If Not Equal to Zero R MR

JNE ADDR if A=Ø, then EA->PC

If the contents of the A register are not equal to zero, then load ADDR into the program counter and continue sequential operation from that location.

Jump If Less Than or Equal to Zero R MR

JLE ADDR if A≤Ø, then EA->PC

If the contents of the A register are less than or equal to zero, then load ADDR into the program counter and continue sequential operation from that location.

Jump If Greater Than Zero R MR

JGT ADDR if A>Ø, then EA->PC

If the contents of the A register are greater than zero, then load ADDR into the program counter and continue sequential operation from that location.

Jump If Less Than Zero R MR

JLT ADDR if A<0, then EA->PC

If the contents of the A register are less than zero, then load ADDR into the program counter and continue sequential operation from that location.

Jump If Greater Than or Equal to Zero R MR

JGE ADDR if A≥0, then EA->PC

If the contents of the A register are greater than or equal to zero, then load ADDR into the program counter and continue sequential operation from that location.

Jump and Decrement Index R MR

JDX ADDR X=X-1
if X=0, then EA->PC;else PC=PC+1

Decrement the contents of the index register by one; then, if the contents of X are not equal to zero, load ADDR into the program counter and continue sequential operation from that location. Otherwise, execute the next sequential instruction.

Jump and Increment Index R MR

JIX ADDR X=X+1
if X=0, then EA->PC; else PC=PC+1

Increment the contents of the index register by one; then, if the contents of X are not equal to zero, load ADDR into the program counter and continue sequential operation from that location. Otherwise, execute the next sequential instruction.

Jump and Store Return in Index R MR

JSX ADDR PC+1->X
EA->PC

Increment the program counter by one and load into the index register. Load ADDR into the program counter and continue sequential operation from that location.

Procedure Stack Control R

This group of instructions simplifies programming of pure procedures, recursive or reentrant subroutines, and dynamic storage allocation. CREP saves the program counter in the current stack frame and transfers control to a subroutine. ENTR creates an n-word stack frame by altering the stack pointer (S Register), and links the new frame with the previous one. RTN undoes the work of both CREP and ENTR by deleting the current frame and restoring the saved program counter value for the calling program.

Stack frames created by recursive or reentrant procedures are assumed to contain n+2 words, where n is the number of locations required for variable or parameter storage during an invocation of the subroutine. The other two words are reserved for the frame linkword (inserted by ENTR) and a return address (inserted by CREP).

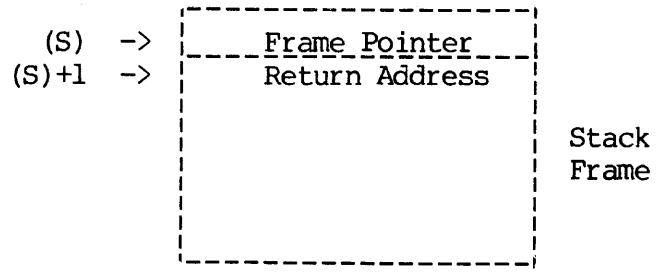
Call Recursive Entry Procedure R MR

CREP ADDR

Increment the programs counter, P, and load P+1 into the location following the one specified by the current stack pointer. Load ADDR into the program counter and continue execution from that location.

```
(P)+1 -> [(S) +1]
EA -> (P)
```

The CREP instruction performs subroutine linkage for recursive or reentrant procedures. CREP stores the return address in the second word of a stack frame created by the ENTR instruction, rather than in the destination address as in a JST:



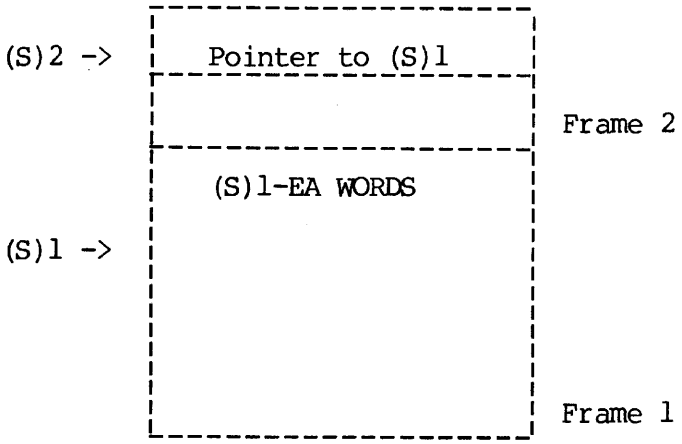
Enter Recursive Procedure Stack R MR

ENTR N

Alter the stack pointer by subtracting the value of N and store the previous value of S in the new location.

(S)1 -> [(S)1-EA]
(S)1-EA -> (S)2

The ENTR instruction allocates a block of memory as a stack frame containing N locations:



The frame is created by subtracting N from the stack pointer contents, (S)1, to form (S)2, and then storing (S)1 at that address. Thus, the first word of the frame points to the previous frame.

Return from Recursive Procedure R GEN

RTN

Fetch the return address from word 2 of the previous stack frame and load the result in the program counter; then transfer word 1 (the pointer to the preceding stack frame) to the S register.

(S)+1 -> P
(S) -> S

If the return address is 0, (S) is unchanged and a PSU (Procedure Stack Underflow) fault is taken (interrupt through location '75 in physical memory is taken on the Prime 300).

Execute RV MR

XEC ADDR

Execute the instruction at location ADDR, but do not transfer control to that location. Not all instructions can be executed by the instruction.

No multi-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. On any fault or interrupt, the saved program counter is relative to the XEC instruction.

Jump and Set Y V MR

JSY ADDR (PC)W->Y
EA->PB

Save the word number of the program counter in the Y register and transfer control to location ADDR. Only the word number portion of the return address is saved, JSY may (usually) only be used to call subroutines that reside in the same procedure segment.

Jump and Set XB V MR

JSXB ADDR PC->XB
EA->PB

Save the 32-bit contents of the program counter in XB, and transfer control to location ADDR. JSXB may be used to make both intersegment and intrasegment subroutine calls.

Effective Address to A Register R MR

EAA ADDR EA->A

Calculate the effective address and load it into register A. The contents of ADDR are unaffected and the original contents of register A are overwritten and lost.

Effective Address to L V MR

EAL ADDR EA->L

Calculate the effective address and put it into the L register.

Effective Address to LB V MR

EALB ADDR EA->LB

Calculate the effective address and put it into the link base, LB.

Effective Address to XB V MR

EAXB ADDR EA->XB

Calculate the effective address and put it into the temporary base, XB.

Procedure Call V MR

PCL ADDR

Call procedure whose ECB is at ADDR.

Step 1. Calculation of Target Ring Number

a. If the caller has Read access to the segment containing the ECB (segment number of ADDR), new ring=current ring.

b. If the caller has Gate access to the segment containing the ECB, new ring=ring field of ECB(PB) The ECB must start on a modulo-16 boundary in this case.

If neither a. nor b. holds, an access violation results.

Step 2. Stack Frame Allocation

a. If ECB(STACK ROOT)=0, then stack root=ECB(ROOTSN)

b. Fetch the free pointer at location 0 of segment (stack root). If there is sufficient room remaining (size needed given by ECB(SFSIZE), allocate frame here and update free pointer in segment stack root.

c. If no room in this segment, fetch the extension pointer at location 2 of the segment pointed to by free pointer. If 0,

generate stack overflow fault. Else, use extension pointer as a new free pointer and go to step b.

Step 3. New Frame Header Setup

a. The flag word (word 0) is set to 0.

b. The caller's PB, SB, LB and keys are saved in the frame header. The ring of field of PB properly reflects the ring of execution of the caller. The saved PB at this moment points to the word following the PCL instruction. It will be updated when argument transfer (if any) complete to point beyond the argument templates. Word '11 of the stack frame is set to the word number of this initial value of saved PB (i.e., points to PCL+2).

Step 4. Callee State Load

The callee's PB, LB, and keys are loaded from the entry control block, except that the ring field of PB has no effect if the ECB is not a gate. The SB register is set to point at the new stack frame.

Step 5. Argument Transfer

If ECB(NARGS) is 0, this step is skipped. Otherwise, the one or more AP's (argument templates) following the PCL instruction are processed to load argument pointers into the callee's stack frame. At least one AP must follow PCL if the callee expects arguments; no AP may follow if the callee expects no arguments. The saved PB in callee's stack frame is updated to point beyond the AP's when argument transfer is done. See the ARGT instruction for a description of argument transfer.

Argument Transfer V GEN

ARGT

The Argument Transfer operation must be the first executable instruction of any procedure which is defined by its entry control block as accepting arguments. 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 must not begin with an ARGT.

The list of argument transfer templates following the caller's PCL 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 Section 5. 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 (XB) 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. The first argument pointer position is specified in the procedure's ECB. If the address has a zero bit offset, the address is stored in the two-word indirect format (with the E-bit clear). 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 argments 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.

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.

Stack Extend V GEN

STEX

Obtains 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 L register specifies the

desired contiguous size of the extension in words. The size is rounded up to an even number of words. The address of the extension is returned as a segment number/word number pointer in the L 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 can cause a stack overflow fault.

Procedure Return V GEN

PRTN

Deallocates the current stack frame 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 The current stack frame consists of the frame created upon entry to the current procedure plus all extensions created during the execution of the current procedure.

PRCEX - Process Exchange - (Restricted) V AP

There are seven process exchange instructions:

INBC
INBN
INEC
INEN
NFYB
NFYE
WAIT

See Section 2 for a complete discussion of the process exchange mechanism.

QUEUE - Queue Management Instructions

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

Data is to or from register A and the results of the operation are given in the condition code bits for later testing.

ADDR refers to a control block in virtual space. 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; any non-ring zero attempt to access physical queues will result in a restrict mode violation fault. Also the ring number determines the privilege of access into both the control block and the data block.

Add to Top of Queue V AP

ATQ ADDR

Add the contents of the A-register to the top of the queue defined by the QCB (Queue Control Block) at ADDR. The condition codes are set EQ if the queue is full e.g., the word could not be added.

Add to Bottom of Queue V AP

ABQ ADDR

Add the contents of the A-register to the bottom of the queue defined by the QCB at ADDR. The condition codes are set EQ if the queue is full e.g., the word could not be added.

Remove from Top of Queue V AP

RTQ ADDR

Remove a single word from the top of the queue defined by the QCB at ADDR, and place it in the A-register. If the queue is empty, set A=0 and condition codes EQ.

Remove from Bottom of Queue V AP

RBQ ADDR

Remove a single word from the bottom of the queue defined by the QCB at ADDR, and place it in the A-register. If the queue is empty, set A=0 and condition codes EQ.

Test Queue V AP

TSTQ ADDR

Set the A-register to the number of items in the queue defined by the QCB at ADDR. If the queue is empty, set condition codes EQ.

SHIFT - Shift Group

Shifting is the movement of the contents of a register bit-to-bit. The instructions in this group shift or rotate right or left the contents of A or the contents of A and B treated as a single register with A on the left. Although these instructions are similar in format and operation, functionally some are logical and others arithmetic.

A shift is logical or arithmetic simply in terms of the way the data word is interpeted: a logical shift treats it as a string of bits whereas an arithmetic shift treats it as a signed number.

Rotation is a cyclic logical shift such that information rotated out at one end is put back in at the other. The last bit rotated in at the right or left is also saved in C.

In a logical or left shift, the contents of the register or registers are moved bit-to-bit with Ø's brought in at the end being vacated. Information shifted out at the other end is lost.

A right arithmetic shift fills the vacated left positions with the sign bit. The C-bit reflects the last bit shifted out.

A left arithmetic shift includes the sign, but interprets a sign change as overflow. It fills the vacated right positions with Ø's and sets the C-bit on overflow.

Hence, arithmetic shifting is equivalent to multiplying or dividing the number by a power of 2, provided no information is lost. These operations also use the C-bit to detect the loss of any bit of significance in a left arithmetic shift, and in all other cases to save the last bit shifted out.

In a shift instruction word, bits 3-6 are all Ø's and the group is indicated by Ø1 in bits 1 and 2. Bits 7-1Ø indicate the particular type of shift, and bits 11-16 specify the twos complement of the number of places to be shifted. Mnemonics are available for the individual types, so the opcode may be regarded as the left four digits of the instruction word, with the word completed by adding the right two digits for the number of places. Note that the mnemonics are constructed using "logical" to mean a logical shift and "shift" to mean specifically an arithmetic shift.

A Left Logical SRV SHFT

ALL n

```
C <-- A  -- A <-- Ø
      1     16
```

Shift the contents of register A left n places, bringing zeros into bit 16; data shifted out of bit 1 are lost, except that the last bit shifted out is saved in C.

A Right Logical SRV SHFT

ARL n

```
Ø -> A  -- A   -> C
     1     16
```

Shift the contents of register A right n places, bringing zeros into bit 1; data shifted out of bit 16 are lost, except that the last bit shifted out is saved in C.

Long Right Logical SRV SHFT

LRL n

```
Ø -> A  -- A     -> B  -- B   -> C
     1     16        1     16
```

Shift the contents of register A and B right n places, bringing zeros into bit 1 of register A. Bit 16 of register A is shifted into bit 1 of register B. Data shifted out of bit 16 of register B are lost, except that the last bit shifted out is saved in C.

Long Left Logical SRV SHFT

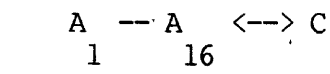
LLL n

```
C <- A  -- A   <- B  -- B   <- Ø
     1     16     1     16
```

Shift the contents of registers A and B left n places, bringing zeros into bit 16 of register B. Bit 1 of register B is shifted into bit 16 of register A; data shifted out of bit 1 of register A are lost, except that the last bit shifted out is saved in C.

A Left Rotate SRV SHFT

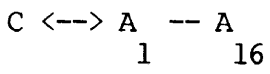
ALR n



Shift the contents of register A left n places, rotating bit 1 into bit 16. The last bit rotated back in at the right is also saved in C.

A Right Rotate SRV SHFT

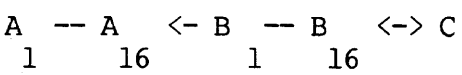
ARR n



Shift the contents of register A right n places, rotating bit 16 into bit 1. The last bit rotated back in at the left is also saved in C.

Long Left Rotate SRV SHFT

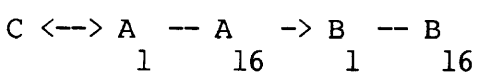
LLR n



Shift the contents of registers A and B left n places, rotating bit 1 of register A into bit 16 of register B. Bit 1 of register B shifts into bit 16 of register A. The last bit rotated from register A back to B is also saved in C.

Long Right Rotate SRV SHFT

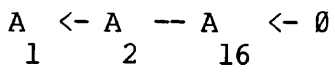
LRR n



Shift the contents of register A and B right n places, rotating bit 16 of register B into bit 1 of register A. Bit 16 of register A is shifted into bit 1 of register B. The last bit rotated from register B back to register A is also saved in C.

A Left Shift SRV SHFT

ALS n



Shift the contents of register A left arithmetically n places, bringing zeros into bit 16. Data shifted out of bit 1 are lost. The C-bit is initially cleared. If the sign (bit 1) changes state, set C. A sign change indicates that a bit of significance (a one in a positive number, a zero in a negative) has been shifted out of the magnitude part.

A Right Shift — SRV — SHFT

ARS n

```
A  -> A  -- A   -> C
 1    2     16
```

Shift the contents of register A right arithmetically n places, leaving the sign (bit 1) unaffected, but shifting it into the magnitude part, zeros in a positive number, ones in a negative. Data shifted out of bit 16 are lost, except that the last bit shifted out is saved in C.

Long Left Shift — SR — SHFT

LLS n

```
A  <- A  -- A      B     B  -- B   <- Ø
 1    2     16     1     2     16
```

Shift the contents of the 31-bit integer in register A|B left arithmetically n places, bringing zeros into bit 16 of register B, bypassing bit 1 of register B; Bit 2 of register B is shifted into bit 16 of register A. Data shifted out of bit 1 of register A are lost. If the sign (bit 1 of register A) changes state, set C; otherwise, clear C. A sign change indicates that a bit of significance (a one in a positive number, a zero in a negative) has been shifted out of the magnitude part.

Long Right Shift — SR — SHFT

LRS n

```
A  -> A  -- A      B     B  -- B   -> C
 1    2     16     1     2     16
```

Shift the contents of the 31-bit integer in register A|B right arithmetically n places, leaving bit 1 of register A unaffected, bypassing bit 1 of register B, and shifting the sign (bit 1 of register A) into the magnitude part (zeros in a positive number, ones in a negative). Bit 16 of register A is shifted into bit 2 of register B; data shifted out of B bit 16 are lost, except that the last bit shifted out is saved in C.

Long Left Shift V SHFT

LLS n

```
L <- L --- L <- 0
 1     2      32
```

Shift the contents of the 32-bit integer in the L register left arithmetically n places, bringing zeros into bit 32. Data shifted out of bit 1 are lost. If the sign (bit 1) changes state, set C; otherwise clear C.

Long Right Shift V SHFT

LLS n

```
L <- L --- L -> C
 1     2      32
```

Shift the contents of the 32-bit integer in the L register right arithmetically n places, leaving bit 1 unaffected. Data shifted out of bit 32 are lost, except that the last bit shifted out is saved in C.

SKIP - Conditional Skip

Skip if A Greater Than Zero SRV GEN

SGT if A>0 then PC+1->PC

If the contents of register A is greater than zero, skip the next instruction in sequence.

Skip if A Less Than or Equal to Zero SRV GEN

SLE if A≤0 then PC+1->PC

If the number contained in A is less than or equal to zero, skip the next instruction in sequence.

Skip on A Bit Set SRV GEN

SAS n if A(n)=1 then PC+1->PC

If bit n in register A is 1, skip the next instruction in sequence.

Note

The assembler will convert n to the octal equivalent of the bit number minus one.

Increment Memory, Replace, and Skip SRV MR

IRS ADDR [EA]16+1->[EA]16;
if [EA]16=0 then PC+1->PC

Add 1 to the contents of location ADDR and place the result back in ADDR. Skip the next instruction in sequence if the result is zero.

Increment and Replace Index SRV GEN

IRX X+1->X;
if X=0 then PC+1->PC

Add 1 to the contents of the index register and place the result back in that register. Skip the next instruction in sequence if the result is zero.

Decrement and Replace Index SRV GEN

DRX X-1->X
if X=0 then PC+1->PC

Subtract 1 from the contents of the index register and place the result back in that register. Skip the next instruction in sequence if the result is zero.

Skip on A Bit Reset SRV GEN

SAR n if A(n)=0 then PC+1->PC

If bit n in the A register is 0, skip the next instruction in sequence.

Note

The assembler will convert n to octal equivalent of the bit number minus one.

Skip on Sense Switch Set SRV GEN

SNS n if sense switch n=1 then PC+1->PC

If sense switch n is on (up), skip the next instruction in sequence.

Skip on Sense Switch Reset SRV GEN

SNR n if sense switch n=0 then PC+1->PC

If sense switch n is off (not up), skip the next instruction in sequence.

Skip Group

SKP n

Skip conditions are selected by individual bits or combinations of them.

- Bits 1-6 are always 100000.
- Bit 7=1 means if true, skip the next instruction.
- Bit 7=0 means if false, skip the next instruction.
- Bit 9=0 means test a combination of bits.

The various conditions, the bits that select them and the mnemonics and opcodes for them are given in Table 7-1.

Table 7-1. Combination Skip Group

Mnemonic	Selector Bits	Bit 7	Skip on Condition	Op Code
NOP		1	None (no-op)	'101000
SKP		0	Skip unconditionally	'100000
SMI	8	1	A Minus (A(1)= 1)	'101400
SPL	8	0	A Plus (A(1)= 0)	'100400
SLN	10	0	LSB Nonzero (A(16)= 1)	'101100
SLZ	10	0	LSB Zero (A(16)= 0)	'100100
SNZ	11	1	A Nonzero	'101040
SZE	11	0	A Zero	'100040
SS1	12	1	Sense Switch 1 Set	'101020
SR1	12	0	Sense Switch 1 Clear	'100020
SS2	13	1	Sense Switch 2 Set	'101010
SR2	13	0	Sense Switch 2 Clear	'100010
SS3	14	1	Sense Switch 3 Set	'101004
SR3	14	0	Sense Switch 3 Clear	'100004
SS4	15	1	Sense Switch 4 Set	'101002
SR4	15	0	Sense Switch 4 Clear	'100002
SSS	12-15	1	Any of Sense Switches 1-4 Set	'101036
SSR	12-15	0	Any of Sense Switches 1-4 Reset	'100036
SSC	16	1	Set C	'101001
SRC	16	0	Clear C	'100001

Skip conditions can be combined using SKP and giving the bit 7-16 configuration for the combination in the address field.

PART THREE

INSTRUCTION SUMMARY I MODE

SECTION 8

FORMATS - I-MODE

32I mode implements the general register architecture of the Prime-500. The data types and instruction formats combine to provide an extensive range of addressing types.

DATA STRUCTURES

Word Length

32 bits

Halfword Length

16 bits

Byte Length

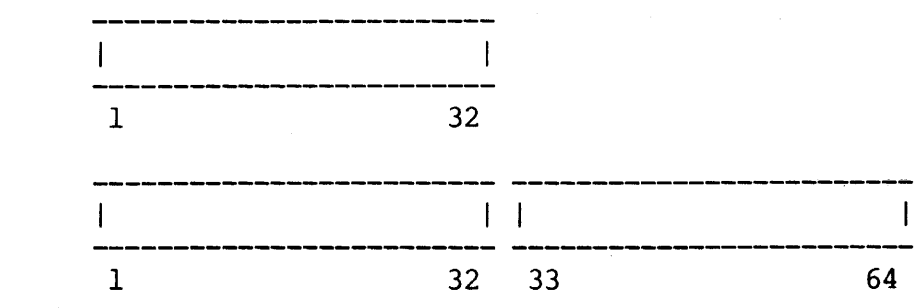
8 bits

Character Strings

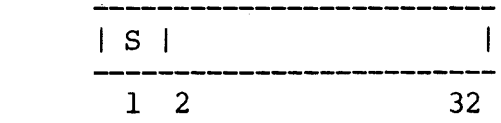
Variable length collection of bytes from 1 to 2**17-1.

Numbers

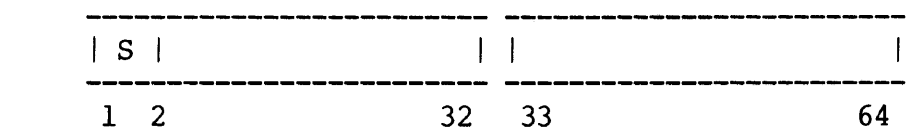
- Unsigned 32 and 64 bit integers



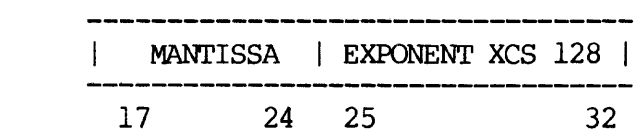
- Signed 32-bit integers



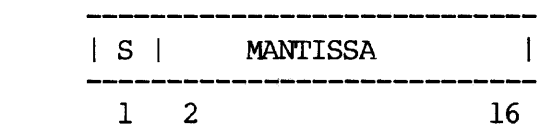
- Signed 64-bit integers

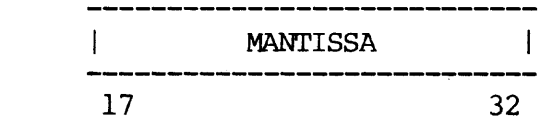


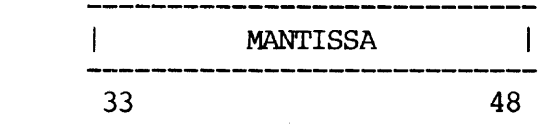
- Floating Point - Single Precision 32 bits

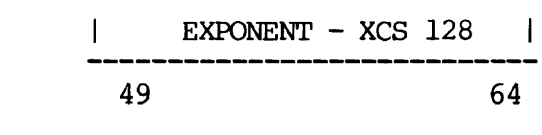


- Floating Point - Double Precision 64 bits









- Decimal - one to 63 digits in five forms

Decimal Control Word Format:

To specify the characteristics of the operation to be performed, most decimal arithmetic instructions require a control word to be loaded in general register 2.

The general format is as follows:

A	-	B	C	-	T	D	E	F	G	H
1-6	7	8	9	10	11	12	13	14-16	17-22	23-29	30-32

Where:

A - Field 1, number of digits

E - Field 1, decimal data type

B - If set, sign of field 1 is treated as opposite of its actual value.

C - If set, sign of field 2 is treated as opposite of its actual value. (XAD, XMP, XDV, XCM only)

D - Round flag (XMV only)

F - Field 2, number of digits

H - Field 2, decimal data type

G - Scale differential (XAD, XMV, XCM only)

T - Generate positive results always

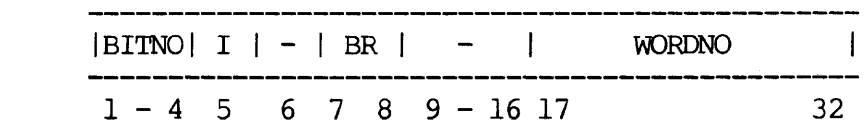
- - Unused, must be zero

The fields used by each instruction are listed in the instruction descriptions. Fields not used by an instruction must be zero.

The scale differential specifies the difference in decimal point alignment between the operator and fields for some instructions. This field is treated as a signed 7 bit two's complement number. A positive value indicates a right shifting of Field 1 with respect to Field 2, and a negative value indicates a left shifting.

Address Pointer (AP)

Two word pointer which follows AP instructions.



BITNO (Bits 1-4) - Bit number

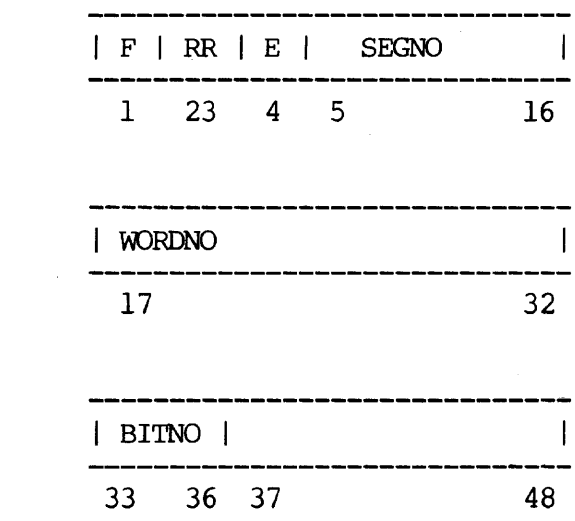
I (Bit 5) - Indirect bit

BR (Bits 7-8) - Base register

- ØØ = Procedure Base (PB)
- Ø1 = Stack Base (SB)
- 1Ø = Link Base (LB)
- 11 = Temporary Base (XB)

WORDNO (Bit 17-32) - Word number offset from base register contents

Indirect Pointer - Two Word Memory Reference (IP)



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

RR (Bits 2-3) - Ring of privilege - controls access rights

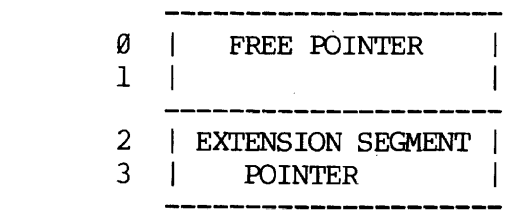
E (Bit 4) - Extend bit. If zero, no third word is present and the bit number of the effective address is taken as zero. If one, the third word is present and gives the bit number.

SEGNO (Bits 5-16) - The segment number portion of the effective address

WORDNO (Bit 17-32) - The word number portion of the effective address.

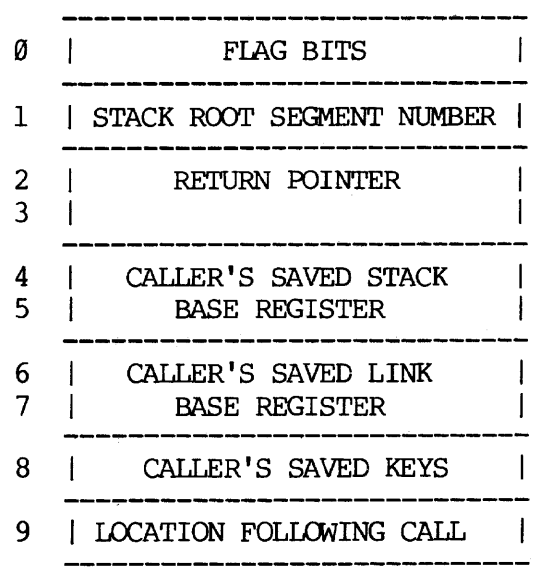
BITNO (Bits 33-36) - The bit number if E is a one.

Stack Segment Header



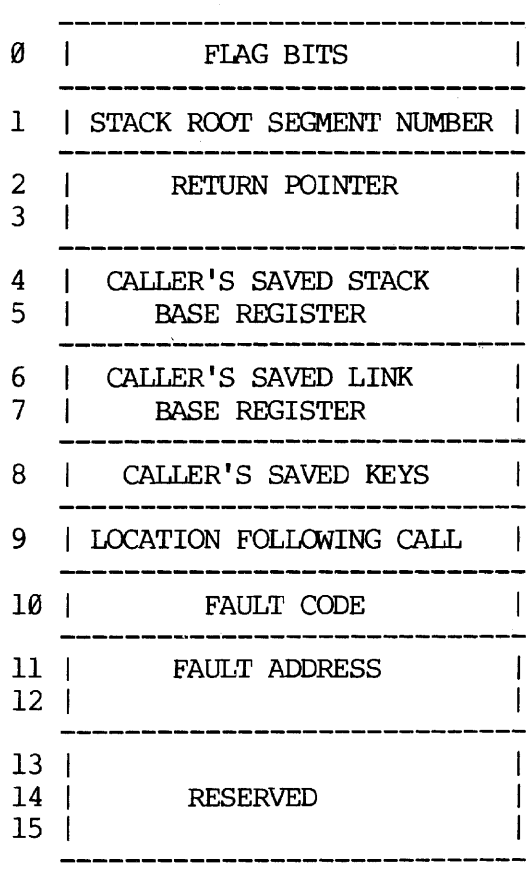
Word	Meaning
0,1	Free pointer - segment number/word number of available location at which to build next frame. Must be even.
2,3	Extension segment pointer - segment number/word number of locations at which to build next frame when current segment overflow. If zero, a stack overflow fault occurs when current segment overflows.

PCL Stack Frame Header



Word	Meaning
0	Flag bits - set to zero by PCL 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 link base register
8	Caller's saved keys
9	Word number of location following call - beginning of argument transfer templates, if any

CALF Stack Frame Header

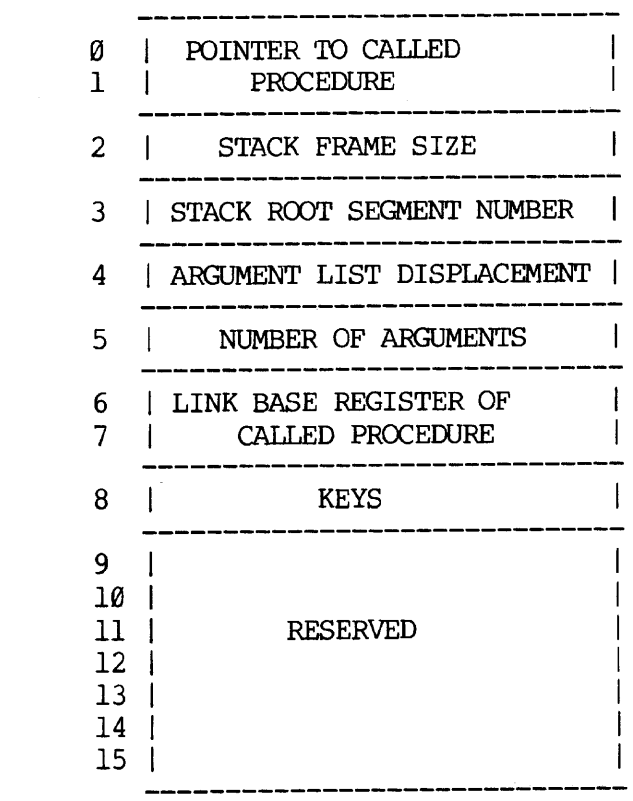


Word	Meaning
0	Flag bits - set to ones by CALF fault
1	Stack root segment number - for locating free pointer
2,3	Return pointer - segment number/word number of location following call and argument sequence which created this frame
4,5	Caller's saved stack base register
6,7	Caller's saved link base register
8	Caller's saved keys
9	Word number of location following call - beginning of argument transfer templates, if any
10	Fault code

11 Fault address

13-15 Reserved

Entry Control Block

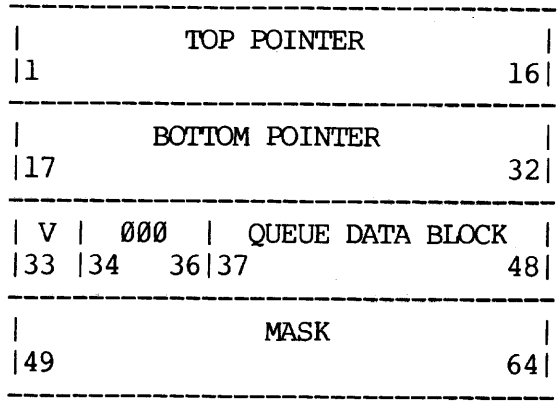


Word	Meaning
0,1	Pointer (ring, segment, word number) to the first executable instruction of the called procedure.
2	Stack frame size to create (in words). Must be even.
3	Stack root segment number. If zero, keep same stack.
4	Displacement in new frame of where to build argument list.
5	Number of arguments expected.
6,7	Called procedure's link base (location of called procedure's linkage frame less '400).
8	CPU keys desired by called procedure.

9-15 Reserved, must be zero.

Entry control blocks which are gates must begin on a Ø modulo 16 boundary, and must specify a new stack root.

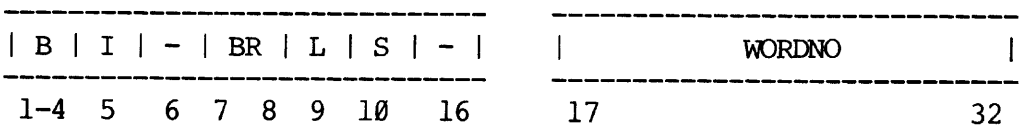
Queue Control Block



Bits	Meaning
1-16	Top Pointer-read
17-32	Bottom Pointer-Write
33 (V)	Virtual/physical control bit Ø = physical queue 1 = virtual queue
34-36	Reserved - must be zero
37-48	Queue data block address Segment number if virtual queue High order physical address bits if physical queue
49-64	Mask - value 2**K-1

Queue control blocks must start on even word boundaries.

Argument Transfer Template:



B (Bits 1-4) - Bit number

I (Bit 5) - Indirect

BR (Bits 7-8) - Base register

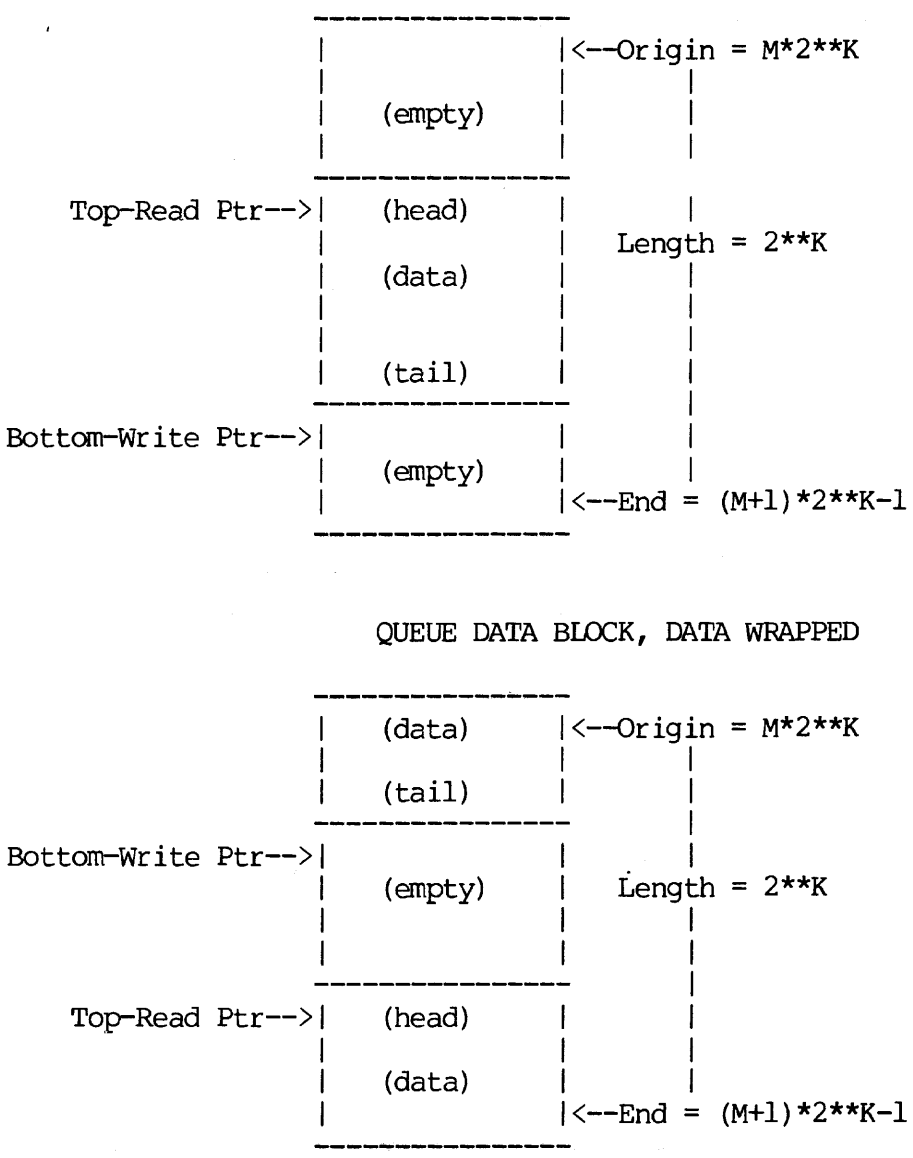
ØØ = Procedure base (PB)
Ø1 = Stack base (SB)
1Ø = Link base (LB)
11 = Temporary base (XB)

L (Bit 9) - Last template for this call

S (Bit 1Ø) - Store argument address. Last template for this argument.

WORDNO (Bits 17-32) - Word number offset from base register

Figure 8-1. Queue Data Structures



PROCESSOR CHARACTERISTICS

I-Mode Register Description:

SCRATCH RS0			DMX RS1			CURRENT REGISTER SET (CRS) RS2	RS3		
ADR	HIGH	LOW	ADR	HIGH	LOW	ADR	ADR	HIGH	LOW
0	TR0	-	40	-	-	100	140	GR0:OLT2	-
1	TR1	-	41	-	-	101	141	GR1:PTS	-
2	TR2	-	42	-	-	102	142	GR2(1,A,LH)	(2,B,LL)
3	TR3	-	43	-	-	103	143	GR3(EH)	(EL)
4	TR4	-	44	-	-	104	144	GR4	-
5	TR5	-	45	-	-	105	145	GR5(3,S,Y)	-
6	TR6	-	46	-	-	106	146	GR6	-
7	TR7	-	47	-	-	107	147	GR7(0,X)	-
10	RDMX1	-	50	-	-	110	150	FAR1(13)	-
11	RDMX2	-	51	-	-	111	151	FLR1	-
12	-	RATMPL	52	-	-	112	152	FAR2(4)	(5)
13	RSGT1	-	53	-	-	113	153	FLR2:VSC(6)	-
14	RSGT2	-	54	-	-	114	154	PB	-
15	RECC1	-	55	-	-	115	155	SB(14)	(15)
16	RECC2	-	56	-	-	116	156	LB(16)	(17)
17	-	REOIV	57	-	-	117	157	XB	-
20	ZERO	ONE	60	(20)	(21)	120	160	DTAR3(10)	-
21	PBSAVE	-	61	-	-	121	161	DTAR2	-
22	RDMX3	-	62	(22)	(23)	122	162	DTAR1	1
23	RDMX4	-	63	-	-	123	163	DTAR0	-
24	C377	-	64	(24)	(25)	124	164	KEYS	(MODALS)
25	-	-	65	-	-	125	165	OWNER	-
26	-	-	66	(26)	(27)	126	166	FCODE(11)	-
27	-	-	67	-	-	127	167	FADDR	(12)
30	PSWPB	-	70	(30)	(31)	130	170	TIMER	-
31	PSWKEYS	1	71	-	-	131	171	-	-
32	PPA:PLA	PCBA	72	(32)	(33)	132	172	-	-
33	PPB:PLB	PCBB	73	-	-	133	173	-	-
34	DSWRMA	-	74	(34)	(35)	134	174	-	-
35	DSWSTAT	-	75	-	-	135	175	-	-
36	DSWPB	-	76	(36)	(37)	136	176	-	-
37	RSAVPTR	-	77	-	-	137	177	-	-

NOTICE - Letters in parentheses ( ) show V-Mode correspondence
- Numbers in parentheses ( ) show P300 Address Mapping

Definitions

TR Temporary Registers
TR7 - Saved return pointer on a crash (automatic save)

RDMX Register DMX
RDMX1 - Used by DMC, buffer start pointer
RDMX2 - REA at time of DMX trap
RDMX3 - Save RD during DMQ
RDMX4 - Used as working register

RATMPL Read Address Trap Map to rP Low

RSGT	Register Segmentation Trap RSGT1 - SDW2 / address of Page Map RSGT2 - contents of Page Map / DSW2
REOIV	Register End of Instruction Vector
ZERO/ONE	Constants
PBSAVE	Procedure Base SAVE saved return pointer when return pointer used elsewhere
C377	Constant
PSWPB	Processor Status Word Procedure Base return pointer for interupt return (also used for Prime 300 compatibility)
PSWKEYS	Processor Status Word KEYS KEYS for interupt return (also used for Prime 300 compatibility)
PPA	Pointer to Process A
PLA	Pointer to Level A
PCBA	Process Control Block A
PPB	Pointer to Process B
PLA	Pointer to Level B
PCBB	Process Control Block B
DSWRMA	Diagnostic Status Word RMA RMA at last Check Trap
DSWSTAT	Diagnostic Status Word STATus
DSWPB	Diagnostic Status Word Procedure Base Return pointer or PBSAVE at last check
RSAVPTR	Register SAVE Pointer Location of Register Save Area after Halt

GR	General Register
OLT2	Old Length and Type
PTS	Pointer To Sign
FAR1	Field Address Register
FLR1	Field Length Register
FAR2	Field Address Register
FLR2	Field Length Register
PB	Procedure Base PBH - RPH PBL - 0
SB	Stack Base
LB	Link Base
XB	Temp. (auxiliary) base
DTAR	Descriptor Table adr. reg.
KEYS	See below
MODALS	See below
OWNER	OWNER
FCODE	Fault CODE
FADDR	Fault ADDRess
TIMER	TIMER

General Register - 32 bits

The eight general registers are numbered from 0 - 7. 1 - 7 may be used for index registers. All are used as fixed point and logical accumulators in register to memory and register to register operations.

Floating Point Registers - 64 bits

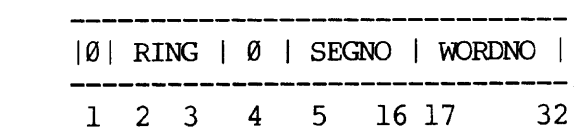
The two floating point registers are numbered 0 and 1. They are used as single and double precision accumulators in register to memory and register to register operations. The two foating point registers overlap the two field length address registers on the Prime 500 and care must be used in moving between floating point and field registers.

Base Registers - 32-bits

The four base registers:

Procedure Base Register	PB
Stack Base Register	SB
Link Base Register	LB
Temporary Base Register	XB

are discussed in Section 2, Prime 400 Architecture. Their format is:



RING (Bits 2-3) - Ring Number

SEGNO (Bits 5-16) - Segment Number

WORDNO (Bits 17-32) - Word Number

Field Registers - 64 bits

The field registers, numbered 0 and 1, have the same meaning as in V-mode. They are distinguished from the floating point registers by the instructions which use them, but exercise caution in moving from one instruction type to the other. The floating point registers overlap the field registers.

Keys

Process status information is collected in a 16-bit register known as the "keys". It may be referenced by the LPSW, TKA, and TAK instructions.

```
-----------------------------------
|C|0|L| MODE |F|X|LT|EQ| 0 - 0 |I|S|
-----------------------------------
 1 2 3  4-6   7 8  9 10 11-14  15 16
```

C (Bit 1) - C-Bit

L (Bit 3) - L-Bit

MODE (Bits 4-6) - Addressing Mode:

000 = 16S
001 = 32S
011 = 32R
010 = 64R
110 = 64V
100 = 32I

F (Bit 7) - Floating point exception disable:

0 = take fault
1 = set C-bit

X (Bit 8) - Integer Exception enable

0 = set C-bit
1 = take fault

LT (Bit 9) - Condition code bits:

EQ (Bit 10)

LT = negative
EQ = equal

DEX (Bit 11) - Decimal exception enable
0 = set C-bit
1 = take fault

I (Bit 15) - In dispatcher - set/cleared only by process exchange

S (Bit 16) - Save done - set/cleared only by process exchange

C-Bit

Set by error conditions in arithmetic operations.

L-Bit

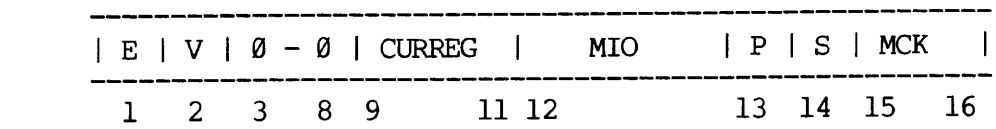
Set by an arithmetic or shift operation except IRS, IRX, DRX. Equal to carry out of the most significant bit (bit 1) of an arithmetic operation. It is valuable for simulating multiple - precision operations and for performing unsigned comparisons following a CAS or a SUB.

Condition Code Bits

The two condition-code bits are designated "EQ" and "LT". EQ is set if and only if the result is zero; 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.

Modals

Processor status is collected in another 16-bit register known as the "modals".



E (Bit 1) - Interrupts enabled

V (Bit 2) - Vectored-interrupt mode

CURREG (Bits 9-11) - Current register set (set/cleared only by process exchange)

MIO (Bit 12) - Mapped I/O mode

P (Bit 13) - Process-exchange mode

S (Bit 14) - Segmentation mode

MCK (Bits 15-16) - Machine-check mode

INSTRUCTION FORMATS

The three primary instruction formats and their subcategories are discussed below.

Non Register Generic:

These instructions are a subset of the V-mode generics and are processed in the same way.

Register Generic:

These instructions operate on the specified register, which may be a general, field, or floating register. This class includes the branch instructions, where the branch address, in the second word, is a 16-bit procedure base displacement.

Memory Reference:

Table 8-1 below summarizes the differences among the types, and Table 8-2 describes the meaning of various tag modes (address formation rules).

Table 8-1

Inst. Type	Data Type	Data Location (2nd word)	Bit Layout (see Table 18-2)
MRNR	Fixed Point Logical	Memory	1 4 7 10 12 16 17-36 ooo 110 ooo TT SSS BB D D
MRGR	Fixed Point Logical	Immediate Register Memory	1 7 10 12 16 17-36 ooo ooo RRR TT SSS BB D D
MRFR	Floating	Immediate Register Memory	1 4 7 9 10 12 16 17-36 ooo 110 ooo TT SSS BB D D

Bit 9=0 if single
Bit 9=1 if double
OP = opcode

Table 8-2. Address Formation
Special Case Selection

T	S	B	Effective Address/Instruction Type
3	>Ø	-	(D+B)*+S (indirect,post-index)
3	Ø	-	(D+B)* (indirect)
2	>Ø	-	(D+B+S)* (pre-index,indirect)
2	Ø	-	(D+B)* (indirect)
1	>Ø	-	D+B+S (indexed)
1	Ø	-	D+B (direct)
Ø	>Ø	Ø	REG-REG (S specifices source Register)
Ø	Ø	1	Immediate Type 1
Ø	>Ø	1	Immediate Type 2
Ø	Ø	2	Immediate Type 3
Ø	1	2	Floating Reg Source (FRØ)
Ø	2	2	Undefined (will not generate UII)
Ø	3	2	Floating Reg source (FR1)
Ø	4-7	2	Undefined (will not generated UII)
Ø	-	3	Undefined (will not generate UII)

MEMORY REFERENCE - ADDRESS FORMATION

See Section 6, V Mode effective address calculation flow charts. Immediate and register-to register operations are discribed below:

Immediate Requirements:

1. Half word general register instruction requires a 16 bit literal (no L suffix).

2. Full word general register instruction requires a 32-bit literal (with L suffix) with high order 16 bits = 0

3. Floating point register instruction (both single and double precision) requires a floating point literal.

Register to Register Requirements:

1. Address field = absolute value, either:

   a. 0 or 1 if the instruction format is MRFR, or

   b. 0 - 7 if the instruction format is MRGR

2. No indirection

3. No indexing

SECTION 9

I-MODE INSTRUCTIONS

ADMOD

Defined in Section 7.

E16S	Enter 16S Mode
E32R	Enter 32R Mode
E32S	Enter 32S Mode
E64R	Enter 64R Mode
E64V	Enter 64V Mode
E32I	Enter 32I Mode

BRAN - Branch

The branch instructions are two word register generics which test the contents of a register or the result of a previous ARITHMETIC or COMPARE operation as indicated by the condition codes (CC), the C-bit, and the L-Bit.

Word 1 = opcode and register number

Word 2 = 16-bit address within the current segment.

The bit layout is:

1	2	3	4	5	6	7	8	9	10 - 16
Ø	Ø	1	Ø	Ø	Ø	R	R	R	opcode

Condition code branches test six conditions based on the LT bit, the EQ bit, and the opcode.

Condition	Meaning
<	branch if LT bit set and EQ bit cleared
≤	branch if LT bit set or EQ bit set
=	branch if EQ bit set
≠	branch if EQ bit cleared
≥	branch if LT bit cleared or EQ bit set
>	branch if LT bit cleared and EQ bit cleared

Test Relation to Ø and Branch if True

These instructions have the following format:

Branch if {Register, Half-Register, Floating-Register} {LT, LE, EQ, NE, GE, GT} Ø

For example: BRLT R,ADDR means Branch to ADDR if Register less than zero.

BRLT R,ADDR	if R<Ø, then ADDR->PC	CC=Result
BRLE R,ADDR	if R≤Ø, then ADDR->PC	CC=Result
BREQ R,ADDR	if R=Ø, then ADDR->PC	CC=Result
BRNE R,ADDR	if R≠Ø, then ADDR->PC	CC=Result

BRGE R,ADDR	if R≥0, then ADDR->PC	CC=Result
BRGT R,ADDR	if R>0, then ADDR->PC	CC=Result
BHLT RH,ADDR	if RH<0, then ADDR->PC	CC=Result
BHLE RH,ADDR	if RH≤0, then ADDR->PC	CC=Result
BHEQ RH,ADDR	if RH=0, then ADDR->PC	CC=Result
BHNE RH,ADDR	if RH≠0, then ADDR->PC	CC=Result
BHGE RH,ADDR	if RH≥0, then ADDR->PC	CC=Result
BHGT RH,ADDR	if RH>0, then ADDR->PC	CC=Result
BFLT F,ADDR	if F<0, then ADDR->PC	CC=Result
BFLE F,ADDR	if F≤0, then ADDR->PC	CC=Result
BFEQ F,ADDR	if F=0, then ADDR->PC	CC=Result
BFNE F,ADDR	if F≠0, then ADDR->PC	CC=Result
BFGE F,ADDR	if F≥0, then ADDR->PC	CC=Result
BFGT F,ADDR	if F>0, then ADDR->PC	CC=Result

Branch on Incremented or Decremented Register

These instructions have the following format:

{Increment / Decrement} {Register / Half Register} by {1 / 2 / 4} then branch if result = 0

For example: BRI1 R,ADDR means increment the contents of the register by 1 and then branch to ADDR if the result equals zero.

BRI1 R,ADDR	R+1 ->R; if R=0, then ADDR->PC	CC=Result
BRI2 R,ADDR	R+2 ->R; if R=0, then ADDR->PC	CC=Result
BRI4 R,ADDR	R+4 ->R; if R=0, then ADDR->PC	CC=Result
BHI1 RH,ADDR	RH+1->RH; if RH=0, then ADDR->PC	CC=Result
BHI2 RH,ADDR	RH+2->RH; if RH=0, then ADDR->PC	CC=Result
BHI4 RH,ADDR	RH+4->RH; if RH=0, then ADDR->PC	CC=Result
BRD1 R,ADDR	R-1->R; if R=0, then ADDR->PC	CC=Result
BRD2 R,ADDR	R-2->R; if R=0, then ADDR->PC	CC=Result
BRD4 R,ADDR	R-4->R; If R=0, then ADDR->PC	CC=Result
BHD1 RH,ADDR	RH-1->RH; if RH=0, then ADDR->PC	CC=Result
BHD2 RH,ADDR	RH-2->RH; if RH=0, then ADDR->PC	CC=Result
BHD4 RH,ADDR	RH-4->RH; if RH=0, then ADDR->PC	CC=Result

Test Register Bit and Branch

- Branch if Register Bit Reset (Equals Zero)

  BRBR R,BITNO,ADDR

  if R(BITNO)=0, then ADDR->PC

- Branch if Register Bit Set (Equals One)

BRBS R,BITNO,ADDR

if R(BITNO)=1, then ADDR->PC

Computed GOTO I RGEN

CGT R,n if 1≤R<n
then [PC+R]->PC
else PC+n->PC

Instruction word followed by n further words:

Word 1 contains integer n

Words 2-n contain branch addresses within the current

Procedure segment

If the contents of register R is less than n and greater than or equal to 1, then control passes to the address in PC+R; otherwise no branch is taken and control passes to PC+n.

CHAR - Character Operations

Load Character

I RGEN

LDC {0 / 1},R

If the specified field length register is nonzero, load the single character pointed to by the specified field address register into R, bits 9-16. R bits 1-8 are cleared. The low order 3 bits of the bit offset in the field address register are ignored, implying that the character must be byte aligned. The specified field address register is advanced 8 bits to the next character, and the field length register is decremented by 1. Set condition code NE (clear EQ).

If the specified field length register is zero, then set the condition code EQ.

Store Character

I RGEN

STC {0 / 1},R

Store bits 9-16 of register R into the character pointed to by the selected field address register. The low order 3 bits of the bit offset of the field address register are ignored, implying that the character must be byte aligned. The field address register is advanced 8 bits to the next character, and the field length register is decremented by 1. Set the condition code NE (clear EQ).

If the specified field length register is zero, set the condition code EQ and do not store.

Summary of Instructions Defined in Section 7

ZCM	Compare Character Field
ZFIL	Fill Character Field
ZMV	Move Character Field
ZMVD	Move Equal Length Fields
ZTRN	Translate Character Fields
ZED	Character Edit

CLEAR - Clear

Clear Register	I	RGEN
CR R	Ø->R	C=unchanged L=unchanged CC=unchanged

Fill R with zeros.

Clear Left Halfword	I	RGEN
CRHL R	Ø->RH	C=unchanged L=unchanged CC=unchanged

Fill bits 1-16 of R with zeros.

Clear Right Halfword	I	RGEN
CRHR R	Ø->RL	C=unchanged L=unchanged CC=unchanged

Fill bits 17-32 of R with zeros.

Clear High Byte 1	I	RGEN
CRBL R	Ø->RH(1-8)	C=unchanged L=unchanged CC=unchanged

Fill Bits 1-8 of R with zeros.

Clear High Byte 2 I RGEN

CRBR R Ø->RH(9-16)

C=unchanged
L=unchanged
CC=unchanged

Fill bits 9-16 of R with zeros.

Zero Memory Fullword I MRNR

ZM ADDR Ø->[EA]32

C=unchanged
L=unchanged
CC=unchanged

Fill contents of ADDR with zeros.

Zero Memory Halfword I MRNR

ZMH ADDR Ø->[EA]16

C=unchanged
L=unchanged
CC=unchanged

Fill contents of ADDR with zeros.

DECI - Decimal Arithmetic

Defined in Section 7

XAD	Decimal Add
XBTD	Binary to Decimal Conversion
XCM	Decimal Compare
XDTB	Decimal to Binary Conversion
XDV	Decimal Divide
XMP	Decimal Multiply
XMV	Decimal Move
XED	Numeric Edit

FIELD - Field Operations

Add Register to Field Address Register — I — RGEN

ARFA {0} ,R — R+FAR->FAR
{1}

Add the contents of R to the field address register, putting the result in the field address register.

Transfer Field Length Register to Register — I — RGEN

TFLR {0} ,R — FLR->R
{1}

Move the contents of the field length register to R.

Transfer Register To Field Length Register — I — RGEN

TRFL {0} ,R — R->FLR
{1}

Move the contents of R to the field length register.

Summary of Instructions Defined in Section 7

EARA 0 Load Field Address Register 0
EAFA 1 Load Field Address Register 1
LFLI 0 Load Field Length Register 0
LFLI 1 Load Field Length Register 1
STFA 0 Store Field Address Register
STFA 1 Store field Address Register

FLPT - Floating Point Arithmetic

Single Precision - 32 bits

Floating Add I MRFR

FA {0},ADDR F+[EA]32->F
{1}

Add the floating point number at ADDR to the contents of the floating point number in the specified floating point register, and leave the resulting floating point number in the floating point register. Addition of floating point numbers requires that their exponents be the same power of two. This is accomplished by right shifting the smaller number by the difference in the exponents. After alignment, the mantissas are added.

If there is an overflow from the most significant bit (not the sign), the sum mantissa is shifted right one place, the exponent is incremented by one and the overflow bit becomes the high-order bit in the normalized mantissa. If the result is otherwise not in normal form (as when numbers with unlike signs are added), the result is normalized. If there is an exponent under/overflow (<-32896, >+32639) set the C-bit or take a floating point exception.

Floating Subtract I MRFR

FS {0},ADDR F-[EA]32->F
{1}

Subtract the contents of ADDR from the specified floating point register by aligning exponents, and proceding as in FA except that the contents of ADDR is subtracted from floating point register.

Floating Multiply I MRFR

FM {0},ADDR F*[EA]32->F
{1}

Multiply the contents of the floating point register by the contents of ADDR and place the product in the floating point register with the mantissa normalized. If there is an exponent under/overflow, the C-bit is set or floating exception is initiated.

Floating Divide I MRFR

FD {0 / 1},ADDR F/[EA]32->F

Divide the contents of the specified floating point register by the number in ADDR and leave the normalized quotient in the floating point register.

If there is an exponent under/overflow or division by zero, the C-bit is set or a floating exception is initiated.

Floating Compare I MRFR

FC {0 / 1},ADDR F::[EA]32

Compare the contents of the specified floating point register with the contents of ADDR and set the condition codes accordingly.

Floating Complement I RGEN

FCM {0 / 1} -F->F

Two's complement the mantissa of the specified floating point register and normalize if necessary. Overflow will set the C-bit or intiate a floating exception.

Floating Load I MRFR

FL {0 / 1},ADDR [EA]32->F

Load the floating point number contained in ADDR into the specified floating point register.

Floating Store I MRFR

FST {0|1},ADDR F->[EA]32

Store the single precision floating point number contained in the specified floating point register in ADDR. Bits 24-31 of the 31 bit mantissa are truncated when written into the 23-bit capacity memory storage. However, the mantissa may be rounded to bit 24 by a FRN instruction which adds 1 to bit 24 if bit 25 is 1. If the floating point register contains an exponent outside the 8-bit range (-128<E<+127), set C or initiate a floating exception.

Floating Round I RGEN

FRN {0|1}

If bit 25 of the mantissa in specified floating point register is 1, add 1 to bit 24 and reset 25. Overflow sets C or generates a floating point exception.

Convert Integer to Floating Point I RGEN

FLT {0|1},R Float(R)->F

Convert the integer in R to a normalized floating point number in the specified floating point register.

Convert Halfword Integer to Floating Point I RGEN

FLTH {0|1},R Float(RH)->F

Convert the halfword integer in RH to a normalized floating point number in the specified floating point register.

Convert Floating Point to Integer I RGEN

INT {0},R Int(F)->R
{1}

Convert the floating point number in the specified floating point regis ter to an integer in R.

Convert Floating Point to Halfword Integer I RGEN

INTH {0},R Int(F)->RH
{1}

Convert the floating point number in the specified floating point register to a halfword integer in RH.

Double Precision - 64 Bits

Double Floating Add I MRFR

DFA {0},ADDR F+[EA]64->F
{1}

Add the contents of ADDR to the contents of the specified floating point register and put the result in the floating point register.

Double Floating Subtract I MRFR

DFS {0},ADDR F-[EA]64->F
{1}

Subtract the contents of ADDR from the contents of the specified floating point register and put the result in the floating point register.

Double Floating Multiply I MRFR

DFM {0},ADDR F*[EA]64->F
{1}

Multiply the double precision floating point number in the specified floating point register by the double precision floating number starting at ADDR and leave the result in the floating point register. Exponents are added and, after mantissas are multiplied, the product is normalized.

An exponent under/overflow sets the C-bit or initiates a floating point exception.

Double Floating Divide I MRFR

DFD {0},ADDR F/[EA]64->F
{1}

Divide the double precision floating point number in the specified floating point register by the double precision floating point number starting at ADDR and leave the result in the floating point register. Exponents are subtracted, and after the divisor mantissa is divided into the dividend mantissa, the quotient is normalized.

An under/overflow or an attempt to divide by zero sets the C bit or initiates a floating point exception.

Double Floating Compare I MRFR

DFC {0},ADDR F::[EA]64
{1}

Compare the contents of ADDR with the contents of the specified floating point register and set the condition codes accordingly.

Double Floating Complement I RGEN

DFCM {0} -F->F
{1}

Two's complement the double precision mantissa in the specified floating point register and normalize if necessary. Overflow sets the C-bit or initiates a floating point exception.

Double Floating Load I MRFR

DFL {0},ADDR [EA]64->F
{1}

Load the double precision number contained in the four memory words at ADDR into the specified floating point register.

Double Floating Store I MRFR

DFST {0},ADDR F->[EA]64
{1}

Store the contents of the specified floating point register into the four memory words at ADDR.

Convert Single to Double I RGEN

DBLE {0},R F->F
{1}

Convert single precision floating point number in the specified floating point register to double precision floating point number in the floating point register.

INT - Integer Arithmetic

C=overflow if and only if IEX=0 (See KEYS in Section 8)

Add Fullword	I	MRGR
A R,ADDR	R+[EA]32->R	C=overflow L=carry CC=arithmetic result

Add the 32-bit integer at ADDR to the 32-bit integer in register R, and put the result into R.

Add Halfword	I	MRGR
AH R,ADDR	RH+[EA]16->RH	C=overflow L=carry CC=arithmetic result

Add the 16-bit integer at ADDR to the 16-bit integer in bits 1-16 of register R and put the result into bits 1-16 of R.

Subtract Fullword	I	MRGR
S R,ADDR	R-[EA]32->R	C=overflow L=carry CC=arithmetic result

Subtract the 32-bit integer ADDR from 32-bit integer in register R, and put the result into R.

Subtract Halfword	I	MRGR
SH R,ADDR	RH-[EA]16->RH	C=overflow L=carry CC=arithmetic result

Subtract the 16-bit integer at ADDR from the 16-bit integer in bits 1-16 of register R and put the result into bits 1-16 of R.

Multiply Fullword	I	MRGR
M R,ADDR	R*[EA]32->R|R+1	C=overflow L=unspecified CC=arithmetic result

Multiply the 32-bit integer in register R by the 32-bit integer at ADDR and put the 64-bit result into R and R+1. The least significant bit is in bit position 64. R must be an even register.

Position After Multiply	I	RGEN
PIM R	R+1->R	C=overflow L=carry CC=arithmetic result

Convert the 64-bit integer in registers R and R+1 to a 32-bit integer in R by moving the contents of R+1 to R. Overflow if a loss of precision would result. Bit 1 of R+1 must be the same as R.

Multiply Halfword	I	MRGR
MH R,ADDR	RH*[EA]16->R	C=overflow L=unspecified CC=arithmetic result

Multiply the 16-bit integer in bits 1-16 of register R by the 16-bit integer at ADDR and put the 32-bit result into R. The least significant bit is in bit position 32.

Position Half Register After Multiply	I	RGEN
PIMH R	RL->RH	C=overflow L=unspecified CC=arithmetic result

Convert the 32-bit integer in register R to a 16-bit integer in bits 1-16 of register R by moving the contents of bits 17-32 of R to bits 1-16 of R. Overflow if a loss of precision would result.

Divide Fullword	I	MRGR
D R,ADDR	R|R+1/[EA]32->R REMAINDER->R+1	C=overflow/div by 0 L=unspecified CC=arithmetic result

Divide the 64-bit integer in registers R and R+1 by the 32-bit integer at ADDR, and put the result in R and the remainder in R+1. The least significant bit of the dividend is in bit 64. Overflow if the quotient is less then -(2**31) or greater than 2**31-1. R must be an even register.

Position For Integer Divide	I	RGEN
PID R	R->R+1 R(1) -> R(2-32)	C=unchanged L=unchanged CC=unchanged

Convert the 32-bit integer in register R to a 64 integer in registers R and R+1 by moving the contents of R to R+1, and extending the sign in bit 1 of R through bits 2-32 of R.

Divide Halfword	I	MRGR
DH R,ADDR	R/[EA]16->RH Remainder->RH	C=overflow/div by 0 L=unspecified CC=arithmetic result

Divide the 32-bit integer in register R by the 16-bit integer at ADDR, and put the result into bits 1-16 of R and the remainder into bits 17-32 of R. The least significant bit of the dividend is in bit 32. Overflow if the quotient is less than -(2**15) or greater than 2**15-1.

Position Half Register For Integer Divide	I	RGEN
PIDH R	RH->RL R(1)->R(2-16)	C=unchanged L=unchanged CC=unchanged

Convert the 16-bit integer in bits 1-16 of register R to 32-bit integer in R by moving the contents of bits 1-16 of R to bits 17-32 of R, and extending the sign in bit 1 through bits 2-16 of R.

Change Sign	I	RGEN
CHS R	-R(1)->R(1)	C=unchanged L=unchanged CC=unchanged

Change bit 1 of register R to its opposite.

Two's Complement Register	I	RGEN
TC R	-R+1->R	C=overflow L=unspecified CC=arithmetic result

Replace the contents of register R by its two's complement.

Two's Complement Half Register	I	RGEN
TCH R	-RH+1->RH	C=oveflow L=unspecified CC=arithmetic result

Replace the contents of bits 1-16 of register R by its two's complement.

Increment Memory Fullword	I	MRNR
IM ADDR	[EA]32+1->[EA]32	C=overflow L=unchanged CC=arithmetic result

Add one to the 32-bit integer at ADDR and put the result into ADDR.

Increment Memory Halfword	I	MRNR
IMH ADDR	[EA]16+1->[EA]16	C=overflow L=unchanged CC=arithmetic result

Add one to the 16-bit integer at ADDR and put the result into ADDR.

Increment Register by 1	I	RGEN
IR1 R	R+1->R	C=overflow L=carry CC=arithmetic result

Add one to the contents of register R and put the result in R.

Increment Register by 2	I	RGEN
IR2 R	R+2->R	C=overflow L=carry CC=arithmetic result

Add two to the contents of register R and put the result in R.

Increment Half Register by 1	I	RGEN
IH1	RH+1->RH	C=overflow L=carry C=arithmetic result

Add one to the contents of bits 1-16 of register R and put the result into R.

Increment Half Register by 2	I	RGEN
IH2 R	RH+2->RH	C=overflow L=carry CC=arithmetic result

Add two to the contents of bits 1-16 of register R and put the result into R.

Decrement Memory Fullword	I	MRNR
DM ADDR	[EA]32-1->[EA]32	C=unchanged L=unchanged CC=arithmetic result

Subtract one from the 32-bit integer at ADDR and put the result into ADDR.

Decrement Memory Halfword	I	MRNR
DMH ADDR	[EA]16-1->[EA]16	C=overflow L=unchanged CC=arithmetic result

Subtract one from the 16-bit integer at ADDR and put the result into ADDR.

Decrement Register by 1	I	RGEN
DR1 R	R-1->R	C=overflow L=carry CC=arithmetic result

Subtract one from the contents of R and put the result into R.

Decrement Register by 2	I	RGEN
DR2 R	R-2->R	C=overflow L=carry CC=arithmetic result

Subtract two from the contents of R and put the result into R.

Decrement Half Register by 1	I	RGEN
DH1 R	RH-1->RH	C=overflow L=carry CC=arithmetic result

Subtract one from the bits 1-16 of register R and put the results into

bits 1-16 of register R.

Decrement Half Register by 2	I	RGEN
DH2 R	RH-2->RH	C=overflow L=carry CC=arithmetic result

Subtract two from the bits 1-16 of register RH and put the result into bits 1-16 of register R.

Compare Fullword	I	MRGR
C R,ADDR	R::[EA]32, Set CC.	

Arithmetically compare the 32-bit integer in R with the 32-bit integer at ADDR and set the condition codes to reflect the results.

Compare Halfword	I	MRGR
CH R,ADDR	RH::[EA]16; Set CC.	

Arithmetically compare bits 1-16 of register R with the 16-bit integer at ADDR and set the condition codes to reflect the results.

Copy Sign	I	RGEN
CSR R	R(1)->C 0->R(1)	

Copy the sign bit of register R, (bit 1), into C and zero R(1).

Set Sign Minus	I	RGEN
SSM R	1->R(1)	

Set the sign bit of register R, (bit 1), equal to one.

Set Sign Plus	I	RGEN
SSP R	0->R(1)	

Set the sign bit of register R, (bit 1), equal to zero.

Test Memory	I	MRNR
TM ADDR	[EA]32::0; set CC	

Test the contents of ADDR and set condition code accordingly.

Add Link to Register	I	RGEN
ADLR R	if keys (L)=1 then R+1->R	

If the L bit is set in the keys then add 1 to the contents of register R.

INTGY

Defined in Section 7.

MDII	Inhibit Interleaved
MDIW	Write Interleaved
MDRS	Read Syndrome Bits
MDWC	Load Write Control Register
EMCM	Enter Machine Check Mode
RMC	Clear Machine Check
VIRY	Verify
LMCM	Leave Machine Check Mode

I/O - Input/Output

Execute I/O I MRNR

EIO ADDR

Interpret the low order 16 bits of ADDR as a Prime 400 PIO instruction. If the addressed device responds ready, the condition codes will be set EQ; otherwise they will be set NE.

summary of Instructions From Section 7

IRTC	Interrupt Return
IRTN	Interrupt Return
CAI	Clear Active Interrupt
ENB	Enable Interrupts
ESIM	Enter Standard Interrupt Mode
EVIM	Enter Vectored Interrupt Mode
INH	Inhibit Interrupts

KEYS - Status Keys

Moves keys to and from registers.

Input Keys	I	RGEN
INK R	keys->R	

Save contents of keys in R.

Output Keys	I	RGEN
OTK R	R->keys	

Restore keys from R.

LOGIC - Logical Operations

Complement Register — REGN

CMR R .NOT.R->R

Complement the contents of R.

Complement Half Register — RGEN

CMH RH .NOT.RH->RH

Complement the contents of RH.

AND Fullword — MRGR

N R,ADDR R.AND.[EA]32->R

AND the contents of R and ADDR and put the result into R.

AND Halfword

NH R,ADDR RH.AND.[EA]16->RH

AND the contents of RH and ADDR and put the result into RH.

OR Fullword — I — MRGR

O R,ADDR R.OR.[EA]32->R

OR the contents of R and ADDR and put the result into R.

OR Halfword — I — MRGR

OH R,ADDR RH.OR.[EA]16->RH

OR the contents of RH and ADDR and put the result into RH.

Exclusive OR Fullword I MRGR

X R,ADDR R.XOR.[EA]32->R

Exclusive OR the contents of R and ADDR and put the result into R.

Exclusive OR Halfword I MRGR

XH R,ADDR RH.XOR.[EA]16->RH

Exclusive OR the contents of RH and ADDR and put the result into RH.

LTSTS - Logical Test and Set

Logical Test and Set (Logicize) I RGEN

If the test is satisfied, then set the register equal to 1.

If the test is not satisfied, then set the register equal to Ø.

These instructions simplify the analysis of complex logical expressions.

The general format is:

If { Condition Codes / Register / Half Register / Floating-Point Register } { LT / LE / EQ / NE / GE / GT } Ø, then 1 -> R; else Ø -> R

For example: LCLT R means, if the condition code is less than zero then set R equal to one, else set R equal to zero.

LCLT R	if CC < Ø, then 1 -> R; else Ø -> R
LCLE R	if CC ≤ Ø, then 1 -> R; else Ø -> R
LCEQ R	if CC = Ø, then 1 -> R; else Ø -> R
LCNE R	if CC ≠ Ø, then 1 -> R; else Ø -> R
LCGE R	if CC ≥ Ø, then 1 -> R; else Ø -> R
LCGT R	if CC > Ø, then 1 -> R; else Ø -> R
LLT R	if R < Ø, then 1 -> R; else Ø -> R
LLE R	if R ≤ Ø, then 1 -> R; else Ø -> R
LEQ R	if R = Ø, then 1 -> R; else Ø -> R
LNE R	if R ≠ Ø, then 1 -> R; else Ø -> R
LGE R	if R ≥ Ø, then 1 -> R; else Ø -> R
LGT R	if R > Ø, then 1 -> R; else Ø -> R
LHLT R	if RH < Ø, then 1 -> R; else Ø -> R
LHLE R	if RH ≤ Ø, then 1 -> R; else Ø -> R
LHEQ R	if RH = Ø, then 1 -> R; else Ø -> R
LHNE R	if RH ≠ Ø, then 1 -> R; else Ø -> R
LHGE R	if RH ≥ Ø, then 1 -> R; else Ø -> R
LHGT R	if RH > Ø, then 1 -> R; else Ø -> R
LFLT R	if F < Ø, then 1 -> R; else Ø -> R
LFLE R	if F ≤ Ø, then 1 -> R; else Ø -> R
LFEQ R	if F = Ø, then 1 -> R; else Ø -> R
LFNE R	if F ≠ Ø, then 1 -> R; else Ø -> R
LFGE R	if F ≥ Ø, then 1 -> R; else Ø -> R
LFGT R	if F > Ø, then 1 -> R; else Ø -> R

Logic set False I RGEN

LF R

Set R equal to zero.

Logic set True I RGEN

LT R

Set R equal to one.

MCTL - Machine Control

Defined in Section 7.

ITLB	Invalidate STLB entry
LPID	Load Process ID
LPSW	Load Program Status Word
RSAV	Register Save
HLT	Halt

MOVE - Move Data

These instructions move data from one location to another.

Interchange Register and Memory - Fullword

I | MRGR

I R,ADDR

R<->[EA]32

Swap the contents of R and ADDR.

Interchange Register and Memory - Halfword

I | MRGR

IH R,ADDR

RH<->[EA]16

Swap the contents of RH and ADDR.

Interchange Register Halves

I | RGEN

IR R

RH<->RL

Swap halves of R.

Interchange Bytes

I | RGEN

IRB R

RH(1-8)<->RH(9-16)

Swap bits 1-8 of RH with bits 9-16 of RH.

Interchange Bytes and Clear Left

I | RGEN

ICBL R

RH(1-8)<->RH(9-16);

Ø->[RH(1-8)]

Swap the bits 1-8 and bits 9-16 of RH. Then set bits 1-8=Ø.

Interchange Bytes and Clear Right I RGEN

ICBR R RH(9-16)<->RH(1-8);
Ø->RH(9-16)

Swap bits 9-16 and bits 1-8 of RH. Then set bits 9-16=Ø.

Interchange Halfwords and Clear Left I RGEN

ICHL R RH<->RL;
Ø->RH

Swap halves of R and set RH=Ø.

Swap Halfwords and Clear Right I RGEN

ICHR R RH<->RL;
Ø->RL

Swap halves of R and set RL=Ø.

Store Fullword I MAGR

ST R,ADDR R->[EA]32

Store the contents of R into ADDR.

Store Halfword I MRGR

STH R,ADDR RH->[EA]16

Store the contents of RH into ADDR.

Store Conditional Fullword I AP

STCD R,ADDR if R+1=[EA]32 then R->[EA]32

If the contents of R+1 equals the contents of ADDR, then store the contents of R into ADDR.

Store Conditional Halfword I AP

STCH R,ADDR if RL=[EA]16 then RH->[EA]16

If the contents of RL equal the contents of ADDR, then store the contents of RH into ADDR.

Load Fullword I MRGR

L R,ADDR [EA]32->R

Load the contents of ADDR into R.

Load Halfword I MRGR

LH R,ADDR [EA]16->RH

Load the contents of ADDR into RH.

Load Halfword Left Shifted by 1 I MRGR

LHL1 R,ADDR [EA]16.LS.1=>RH

Left shift the contents of ADDR by 1 and put the result into RH.

Load Halfword Left Shifted by 2 I MRGR

LHL2 R,ADDR [EA]16.LS.2=>RH

Left shift the contents of ADDR by 2 and put the result into RH.

Load Addressed Register I MRGR

LDAR R,ADDR

Stores the contents of R into the register specified by ADDR. There are three special cases of this instruction which are summarized below. Only the word portion of the effective address is used.

Ring 0 and Bit 2 of word portion = 1 (Restricted)

Bits 10-16 - Absolute register number from 1-12 (see discussion of register sets in Section 8, FORMATS - I-MODE).

Ring 0 and Bit 2 of word portion = 0 (Restricted if Register > '17)

Bits 12-16 - Register 0-37 in current register set.

Ring 1 or 3

Bits 1-12 must = 0

Bits 13-16 - Register 0-17 in current register set.

Store Addressed Register I MRGR

STAR R,ADDR

Stores the contents of the register specified by the contents of ADDR into R. There are three special cases of this instruction which are summarized below. Only the word portion of the effective address is used.

Ring 0 and Bit 2 of word portion = 1 (Restricted)

Bits 10-16 - Absolute register number from 1-128. (See discussion of register sets in Section 8, FORMATS - I-MODE).

Ring 0 and Bit 2 of word portion = 0 (Restricted '20 => '37)

Bits 12-16 - Register 0-37 in the current register set.

Ring 1 or 3

Bits 1-12 must = 0

Bits 13-16 - Register 0-17 in the current register set.

PCTLJ - Program Control and Jump

These instructions transfer control to a different location. They differ from branch instructions in the ability to move across segments. They differ among themselves in the complexity of operations performed and in the handling of the return address.

Jump	I	MRNR
JMP ADDR	EA->PC	

Jump to ADDR.

Jump to Subroutine	I	MRGR
JSR R,ADDR	PCL->RH; EA->PC	

Jump to ADDR and save the 16-bit word number position of the return address in RH.

Jump and Set XB	I	MRNR
JSXB ADDR	PC->XB EA -> PC	

Jump to ADDR and save the full 32-bit return address in XB.

Effective Address to Link Base	I	MRNR
EALB ADDR	EA->LB	

Store the effective address of ADDR in the link base register.

Effective Address to Register I MRGR

EAR R,ADDR EA->R

Store the effective address of ADDR in R.

Effective Address to Temporary Base I MRNR

EAXB ADDR EA->XB

Store the effective address of ADDR in the temporary base register.

Summary of Instructions Defined in Section 7

PCL	Procedure Call
ARGT	Argument Transfer
PRTN	Procedure Return
SVC	Suprevisor Call
STEX	Stack Extend
RSAV	Register Save
RRST	Register Restore

QUEUE - Queue Management

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

Data is to or from general register 2 and the results of the operation are given in the condition code bits for later testing.

ADDR refers to a control block in virtual space. 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; any non-ring zero attempt to access physical queues will result in a restrict mode violation fault. Also, the ring number determines the privilege of access into both the control block and the data block.

Add to Top of Queue — I — AP

ATQ ADDR

Add the contents of general register 2 to the top of the queue defined by the QCB (Queue Control Block) at ADDR. The condition codes are set EQ if the queue is full e.g., the word could not be added.

Add to Bottom of Queue — I — AP

ABQ ADDR

Add the contents of general register 2 to the bottom of the queue defined by the QCB at ADDR. The condition codes are set EQ if the queue is full e.g., the word could not be added.

Remove from Top of Queue — I — AP

RTQ ADDR

Remove a single word from the top of the queue defined by the QCB at ADDR, and place it in general register 2. But if the queue is empty, set general register 2=0 and condition codes EQ.

Remove from Bottom of Queue I AP

RBQ ADDR

Remove a single word from the bottom of the queue defined by the QCB at ADDR, and place it in general register 2. But, if the queue is empty, set general register 2=0 and condition codes EQ.

Test Queue I AP

TSTQ ADDR

Set general register 2 to the number of items in the queue defined by the QCB at ADDR. If the queue is empty, set condition codes EQ.

PRCEX

Defined in Section 7.

INBC	Interrupt Notify
INBN	Interrupt Notify
INEC	Interrupt Notify
INEN	Interrupt Notify
NFYB	Notify
NFYE	Notify
WAIT	Wait

SHIFT - Shift Data

Register Shifts

Rotate I MRGR

ROT R,ADDR

Rotates the bits in R. The low order 16 bits of ADDR tell how many bits to shift, in what direction and whether full or halfword.

Bit 1 = 0 = left

Bit 1 = 1 = right

Bit 2 = 0 = word (32)

Bit 2 = 1 = halfword

Bits 3-16 = no. of bits to shift

Shift Arithmetic I MRGR

SHA R,ADDR

Shift R arithmetically, leaving bit 1 of the register untouched. The low order 16 bits of ADDR tell how many bits to shift, in what direction and whether full or halfword.

Bit 1 = 0 = left

Bit 1 = 1 = right

Bit 2 = 0 = word (32)

Bit 2 = 1 = halfword

Bits 3-16 = no. of bits to shift

Shift Logical I MRGR

SHL R,ADDR

Shift all bits in R, including bit 1. The low order 16 bits of ADDR tell how many bits to shift, in what direction and whether full or halfword.

Bit 1 = 0 = left

Bit 1 = 1 = right

Bit 2 = 0 = word (32)

Bit 2 = 1 = halfword

Bits 3-16 = no. of bits to shift

Shift Register Left 1 I RGEN

SL1 R

Shift R left one bit.

Shift Register Left 2 I RGEN

SL2 R

Shift R left two bits.

Shift Register Right 1 I RGEN

SR1 R

Shift R right one bit.

Shift Register Right 2 I RGEN

SR2 R

Shift R right two bits.

Half Register Shifts

Shift Half Register Left 1 I RGEN

SHL1 R

Shift RH left one bit.

Shift Half Register Left 2 I RGEN

SHL2 R

Shift RH left two bits.

Shift Half Register Right 1 I RGEN

SHR1 R

Shift RH right one bit.

Shift Half Register Right 2 I RGEN

SHR2 R

Shift RH right two bits.

A P P E N D I C E S

APPENDIX A

BASIC FEATURES OF THE PRIME 100, 200 AND 300

This section describes the basic architecture and program-visible features of the Prime 100, 200 and 300. Appendix B describes the advanced features of the Prime 300.

PROCESSOR ORGANIZATION

From the user's point of view, the central processor is the control unit for the entire system; it performs all arithmetic, logical, and data handling operations, manages address calculations, and sequences the program. It is connected to the memory by a memory bus and to the peripheral equipment by I/O, data, address and control busses. The processor (Figure A-1) consists of a set of high speed hardware registers addressed by a register set, an arithmetic logic unit, and other registers such as the Y and M memory buffers that are connected to the memory and I/O busses. Microprocessing logic manipulates data contained in these system elements to execute each instruction.

Microprogram Control

Processor arithmetic operations are performed by manipulating data contained in the register set in conjunction with the arthmetic logic unit. Processor arithmetic operations, data transfers to and from main memory and peripheral I/O operation are all controlled by a microprogram stored in read-only memory. The microprogram is a separate program stored in increments of 256 52-bit micro-instructions. The microprocessor executes one or two micro-instructions during each machine cycle to execute user-level instructions, calculate addresses, accomplish interrupts, oversee I/O transfers, and in general perform internal system control functions.

High Speed Register Set

The first 32 memory locations (0-'37) are high speed hardware that permits multi-step instruction op-codes, (.e.g., multiply, double-precision) to proceed at several times the memory cycle time under microprogram control. The X, A and B registers can be addressed symbolically or as memory locations.

A detailed discussion of microprogramming and the associated registers is given in the Microcoders Handbook (MAN1940).

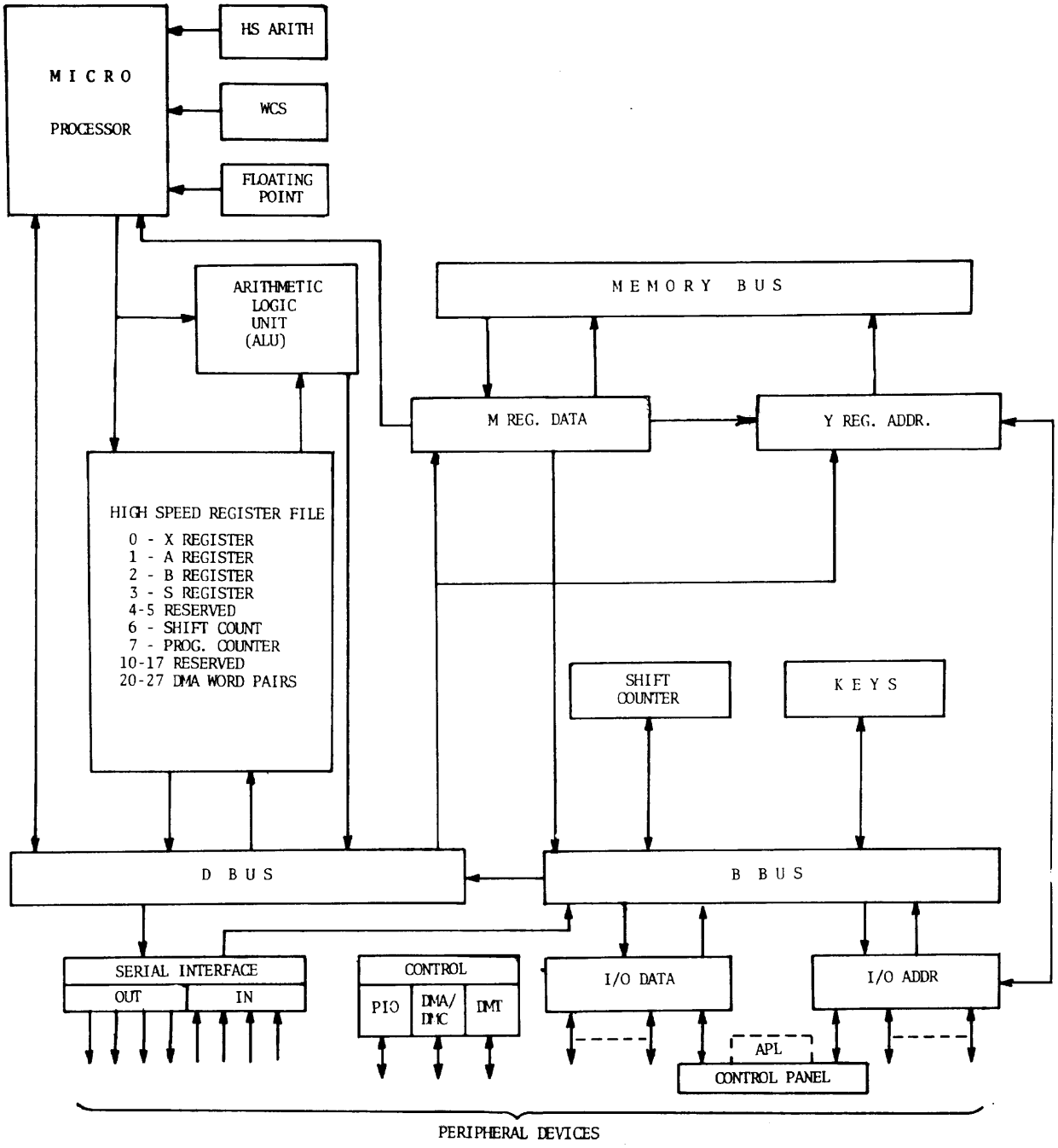


Figure A-1. CPU Block Diagram

CENTRAL PROCESSOR DESCRIPTION

The central processors can be thought of as having a processor within a processor. From the user standpoint, the outer processor is a stored program digital computer consisting of a control unit, main memory, arithmetic, and I/O logic. In a microprogrammed computer, however, the function of the control unit is implemented with an inner control memory containing an orderly arrangement of instruction sub-elements. These sub-elements, called micro-instructions, are arranged into a series of steps (a microprogram) to execute a user level or outer processor level instruction.

The inner processor or micro-processor also contains a control unit, memory and I/O facilities. It too contains a program address register, fetches instructions, and executes them. It is even capable of being interrupted from normal instruction level sequences in order to handle I/O, power failure, machine checks, etc.

In order to achieve speed in executing user level instructions and minimize random discrete logic, the micro-instruction word is 52 bits wide and is expandable to 64 bits wide. The micro-instruction is divided into 12 fields, with each field controlling a portion of the processors operation.

The microprogram resides in a read-only control memory (ROM), which makes it impervious to power outages and programming errors.

Every function that the outer processor would normally perform is controlled by a series of the micro-instruction steps. This includes fetching user-level instructions from memory, incrementing the program address register, and executing the instruction. Unlike the outer processor, the microprogram never stops. Even when the outer processor is executing a HALT instruction, the microprogram is monitoring the control panel and is ready to respond to control panel input. The control panel is an I/O device and the switch settings are interpreted by the CPU as data words, with each bit having a particular function. The microprogram decodes the sense switch data and then controls user program execution, and displays data and program addresses on the control panel displays.

STANDARD CPU FUNCTIONS

Sequential Instruction Execution

The address for a memory access is held in register 7, and data read from memory or about to be stored into memory is held in register M. The processor performs a program by executing instructions retrieved from consecutive memory locations counted by the program counter (P register), register 7 on the block diagram. As one instruction is being fetched, P is incremented by 1 so that the next instruction is

normally taken from the next consecutive location. Sequential program flow is altered by changing the contents of P, either by incrementing it an extra time in a test-skip instruction or by replacing its contents with the value specified by a jump instruction.

Addressable Registers

A general-purpose arithmetic unit and high-speed register set are used by the outer processor to perform machine-language functions and store transient data and control information. The arithmetic unit performs arithmetic and logical operations while the 32 addressable registers in the register set handle such functions as address indexing, stack processing, program sequencing, and control direct-to-memory I/O data transfers. All processing is done on 16-bit words, with all bits in a word processed in parallel.

Arithmetic Register

All computations are performed using the ALU and the arithmetic or A register. Data can be moved in either direction between A and any memory location via the D bus and the M register. The contents of a memory location can be combined arithmetically or logically with the contents of A. The A register also serves as the data connection with the programmed I/O bus, via the D bus and B bus. A secondary arithmetic register, the B register, serves as a right extension of A for double length operations. The processor also has a single-bit register, the C register (or carry bit), that is set on overflow in arithmetic operations and is loaded with the last bit dropped out of A or B in shift operations.

Referencing Memory

Each memory reference instruction calculates an effective address that is stored in the Y register. This calculation may include indirection, where an address calculated at an intermediate step is used to retrieve another address, and may include indexing, where a fixed quantity is added to a given address. The index register (X) as well as the S (stack) register may be used for storing the indexing quantity. The S register is used for push-pop stack operations as well as fully recursive reentry procedures. The recursive procedure is essentially an indexing technique that is performed independently of and addition to the indexing in the effective address calculation involving X.

MOS Memory

Instructions and data are stored in MOS main memory. Prime systems use MOS memory exclusively and depending on which processor is used, memory access times of 680, 600 or 440 nanoseconds, and maximum capacities of 64K or 256K words are available. A system's main memory can be expanded in modular increments of 8K or 32K words on a single 16" X 18" circuit board.

Automatic Power Monitor with Battery Backup

Any configuration of processor and MOS memory can be equipped with an optional power monitor system to preserve the memory's contents if AC power is interrupted, and automatically restart program execution when power is restored. Four major functions are handled by the power monitor option: sensing line voltage not within operational limits, issuing an interrupt at the onset of power failure, battery refreshing of MOS memory, and automatic restart when power is restored.

The battery backup system includes one or more 20 Amp-hour, sealed gelelectrolyte cells and an automatic charger. The batteries are housed on a rack-mountable panel which can be installed in any position in a system rack or cabinet.

Direct-To-Memory Data Transfers

Data transfers between the main memory and high-speed devices can be performed through the use of the programmable, eight-channel direct memory access (DMA) system, standard on all Prime processors. The number of direct memory channels and the maximum data rate can be expanded with DMC and DMT modes of operation (optional on the Prime 100 and 200, and standard on the 300). DMC, which provides up to 2,000 direct memory channels, is similar to DMA, except that where DMA uses registers in the high-speed register set to store control information, DMC uses main memory locations. DMT is used with certain high-speed device controllers in which the controllers themselves monitor direct memory transfers with minimal processor intervention.

Processor Serial I/O Ports

In addition to the data transfers handled via the I/O bus, EIA binary signals up to 9600 baud, can be handled by a four-channel, bit-serial, full-duplex interface which is an integral part of all Prime processors. By means of programmed control of this interface, serial data can be transmitted on four output lines and simultaneously received on four input lines. The interface operates on EIA standard levels, and all lines are easily accessible at the back edge-connector strip of the processor board.

Vectored Priority Interrupts

A flexible interrupt processing capability is a standard feature on all Prime processors and augments programmed control of I/O data transfers. I/O processing on an interrupt basis frees the central processor for other activities between data transfers, and automatically resolves processing priorities when multiple activities require servicing at the same time. An interrupt vectoring technique minimizes interrupt response time by assigning each interrupt source a program selectable memory location for subroutine entry. Interrupt priorities are established by the physical sequence in which device controllers are plugged into the back plane.

INSTRUCTION EXECUTION

Refer to the block diagram, Figure A-1, to supplement this discussion of instruction execution.

High-Speed Register Set

All processor register are physically located in a high-speed register set and logically addressed as if they were MOS memory locations. Memory addresses 0-37 are reserved for this purpose and correspond to the following registers:

Memory Address	Register Designation	Function
0	X	Index Register
1	A	Arithmetic Register
2	B	Extension Arithmetic Register
3	S	Stack Register
4	FLTH	Floating Point
5	FLTL	Accumulator
6	VSC	Visible Shift Count/Floating Point Exponent
7	P	Program Counter
10	PMAR	Page Map Address Register
11	Reserved for microprogram	
12	PFAR	Page Fault Address Register
13-17	Reserved for microprogram	
20-37	DMA-18	Word Pairs for DMA channels (address and word counts)

Transfer of Information

The simplest CPU operation is the transfer of information from one register to another register or a series of registers; for example, to transfer the contents of the A register in the register set to register M and thence to the memory bus. To do this, the A register must be selected; the register set must be allowed on the bus D; the resultant data on bus D must be put into the M register and its effective address must be calculated and stored in the Y register, then the M register and Y register address must be transferred to the memory bus. Finally, the data on the memory bus must be transferred to memory at the specific memory address. The program counter (register 7) must have been incremented when the instruction was fetched. This process is roughly a Store A (STA) instruction.

Conversely, information may be taken from memory and moved back down to a register in the register set. This process is roughly equivalent to a Load A (LDA) instruction. To do this, information must be transferred from the memory to memory bus to the M register which must be selected as the source of bus D via a transfer through bus B. Then the register set must be used as the source of the information on bus D. Finally, the P counter (register set 7) must have been incremented when the instruction was fetched. These operations are accomplished by

selecting and setting the proper microcode fields then executing the microcode. For details of the fields that are set and how to construct microcode information, refer to the Microcoder's Handbook (MAN 1940).

Transfers Using the ALU

To add two values, the first of which is in the M register and the second of which is in the A register and then load the result into the A register, it is necessary to first get the correct data to the inputs of the arithmetic logic unit (ALU). This is done by selecting the M register as the source of the B bus and the A register as the register set register. Next, the ALU must be conditioned to add. This is done by selecting the microcode fields for addition (Refer to the Microcoder's Handbook). After the add operation, the results have to be loaded back into the A register by selected the ALU as the source of bus D and the A register as the destination of the information on bus D.

Shifting

Shifting is controlled by microcode. This includes both the type of shift and the end conditions. It is accomplished by using the information in the VSC register (register 6) as the source for the information on the D Bus. Each output of the VSC register is shifted (right or left) one place before being placed into the D bus; the shift counter is used to keep count of the number of shifts. This counter is created outside of the register set and can be loaded from bus B and read in as the low order half of bus B. The shift counter incrementation takes place at the end of the shift cycle.

MEMORY CYCLING

MOS memory provides an optimum combination of high speed, simple plug-in expansion and high density packaging. Memory cycle times are either 600 or 750 nanoseconds on the Prime 300, 750 nanoseconds on the 200 and 1 microsecond on the 100. A single etched circuit board provides 8K words available in the increments up to 32K per board with integral byte parity. Memory capacity is expandable to 64K in 8K increments on all Prime computers, and to 256K in 32K increments of 750 ns memory on 300-series machines.

The main memory is addressed as a set of contiguous word locations whose addresses range from 0 to '177777 or 65,536. (Memory locations are always specified by their octal addresses.) The number of words that can be addressed by an instruction, and the location of those words relative to the instruction depend on which of two addressing modes - sectored or relative - the machine is operating in. In either mode, contiguous word locations are organized into fixed-length groups called sectors.

Sectored and Relative Addressing Modes

In sectored mode addressing, all sectors are 512 words long and an instruction may directly address either the locations in sector zero (locations 0 -'777) or the locations in the sector in which the instruction is stored. Relative mode addressing permits direct references to locations in sector zero, as in sectored mode, or references to locations in a range relative to the contents of the program counter P (P-239 to P+256). Sixteen unused addresses from P-240 to P-256 are interpreted as special addressing codes that provide additional methods of address formation such as stack register operation, base-plus-displacement and direct addressing of any location from 0 to '177777.

Automatic Memory Refresh

The computer's semiconductor memory is continually refreshed by a sequence of staggered refresh cycles, each of which refreshes 1/32 of the entire memory. Although refreshing does take some time from the program, the effect is usually negligible as the microprogrammed processor logic continues in operation while the refreshing is in progress.

Reserved Memory Locations

Locations '40-'57 are reserved for eight direct memory channel (DMC) data words and eight channel control words. Locations '60 through '74 are dedicated for specific interrupts, both internal (i.e., memory parity errors and illegal instructions) or external (peripheral deviced interrupts). Locations '100-'177 are set aside for vectored interrupts from peripheral devices (i.e., the locations used for a particular interrupt is typically '100 plus the code of the device causing the interrupt).

INTERRUPT AND TRAP HANDLING

Traps and Interrupts

Traps result in branching in the microcode. Interrupts result in branching in the executing program. Some traps also cause interrupts.

There are external and internal interrupts. Internal interrupts are those caused by traps, such as unimplemented instruction interrupts, etc. External interrupts are caused by real-time interrupt requests from device controllers plugged into the backplane. External interrupts can be enabled or disabled by the INH and ENB instructions.

External interrupts have two modes, vectored and standard, selected by the EVIM and ESIM instructions. In standard mode, an indirect JST through location '63 is executed. In vectored mode, the indirect JST

is through a vector address provided by the interrupting controller. In both modes, interrupt priority is determined by the backplane.

Interrupts

There are 13 different interrupt vectors allowed in the Prime 100/200. They fall into several broad classes: hardware monitoring, external, and software aids.

All of these interrupts have some properties in common. First, all of the interrupts check their vector location to see if it is zero before going indirect through it. If it is zero, HLT (halt) is executed. Second, the vector is interpreted as a 16 bit absolute address independent of addressing mode in force. Third, the program counter is deposited at the address pointed to by the vector and execution begins at the next address. Fourth, the non-visible keys are changed by clearing out the 'system clear' and 'permit external interrupts' flops. Fifth, all vectors do an absolute vector.

Hardware Monitoring

These interrupts as a class check on the operability of the system and give the user warning of past or approaching failures:

1. Missing Memory Module

   The memory does not exist at a location accessed. This interrupt may be used to determine memory size. It may result from the CEA instruction as well as any memory reference instruction.

   The interrupt cannot be inhibited and deposits the P counter pointing to the next instruction to be executed. The machine check flag is cleared by this interrupt.

2. Memory Parity

   An error has been detected in the memory data most recently read.

3. Machine check

   An internal data transfer or I/O bus transfer generated at parity error.

4. Parity Fail

   This uninhibitable interrupt is taken when a failure of system power is detected. The interrupt is through location '60 and is given 1 millisecond before an internal system clear signal is given. If location '60 has an address other than zero in it, an interrupt to that location will be executed. If the contents of location '60 is zero, a halt occurs.

Systems with battery backup can minimize the effect of power loss by saving applicable data registers, terminating peripheral tranfers and setting up for an auto restart at location '1000 when power is restored.

External Interrupts

These interrupts serve as the normal asynchronous sources of external stimuli to the processor. Included in this class are all of the normal periphral interrupts.

1. Real Time Clock (Increment)

   This interrupt does not interrupt program execution. However, it does increment location '61 of memory every 16.6 milliseconds (20 milliseconds for 50 Hz systems). On incrementing to zero, an external interrupt through location '63 is requested. Incrementing the clock is not affected by the ENB and INH instructions, but can be started and stopped using programmed I/O.

2. Real Time Clock (Overflow)

   This is a standard external interrupt. (See 3.)

3. Interrupt (Compatible Mode)

   This interrupt is for all external devices. It can be enabled or inhibited using the ENB and INH instructions respectively The actual device interrupting must be determined by a polling method. External interrupts are automatically inhibited by this interrupt. External interrupts come here if the processor is in compatible mode.

4. Interrupt (Vectored Mode)

   Identical in function to compatible mode, this method is used if the processor has been put into the vectored interrupt mode. ENB and INH word as before.

   This time, however, each interrupt uses a vector specified by the controller (normally '100 + Device Address) and the vector can be anywhere in the first 64K memory.

Software Aids

These vectors serve as a link to tie user developed software to Prime developed software along a clearly defined path. In addition, standard software can use these traps to run efficiently on large Prime machines while still running successfully on smaller Prime machine.

SVC (SerVice Call): This interrupt is a convenient way of unambiguously demanding the attention of the executive software. Argument transfer will typically be done using the computer words in memory that follow the SVC.

Prime executive software defines the SVC calls. The advantage of using SVC is: an SVC works the same in normal, restricted, or virtual execution mode. Thus standard software is able to run in different execution environments.

Restricted Execution Violation: This interrut is enabled by executing an ERM and disabled by any interrupt (including SVC).

If enabled, this interrupt occurs whenever a restricted user executes any I/O (including ISI and OSI) instructions, or machine mode change of any non-visible key, or over n levels of indirection (n = a convenient number), or execution of a HALT. This feature is found only in systems with virtual memory.

UII (Unimplemented Instruction): To permit upward compatible software, Prime has reserved octal codes that when executed cause an unimplemented instruction interrupt. On the Prime 100 and 200, Multiply and Divide are examples of instructions that cause this uninhibitable interrupt.

As a result, a package that decodes and software-implements these instructions, can be added. To help this unimplemented instruction (UII) package, the program counter contents is saved so that a deposited program counter always points to the instruction that caused the interrupt.

ILL (Illegal Instruction): To permit customer use of special op codes which act as UII's, Prime has defined many codes as illegal. Execution of these causes an interrupt similar to the UII (Unimplemented Instruction package). The difference is that an instruction that is unimplemented can easily become implemented in the future by microcode changes. Illegal instructions, however, will remain illegal.

Internal Interrupts

Besides the use of interrupts to handle the peripheral equipment, a number of internal processor situations can interrupt the program. The action taken in response to an internal interrupt is essentially the same as for an external interrupt, but many of the conditions associated with the latter are not applicable to the former. All internal interrupts are vectored regardless of the mode of the external interrupt.

Although a particular type of internal interrupt may be inhibited at its source, it is never affected by the enabling or inhibiting of external interrupts as a class; e.g., a memory parity error can cause an interrupt only if the processor is in machine check mode, but with that mode in effect, an error always causes an interrupt even if

external interrupts are inhibited. All internal interrupts have priority over external interrupts simply by virtue of the circumstances they represent. Among internal interrupts, priority is a function of logical necessity.

In response to an internal interrupt, the processor vectors through a specific location. If the 16-bit absolute address in this location is zero, the processor halts. If the address is nonzero, the processor inhibits external interrupts saves the P register in the location and resumes normal program execution at the location following that in which the P register was stored. Since an internal interrupt has nothing to do with the bus priority structure, the service routine need not give a CIA upon completion.

Internal interrupts are used to monitor the hardware and aid in software execution. Interrupt locations and conditions that generate interrupts through them are as follows:

'60 Power Failure - incoming power is not up to specification. This vector must be left unimplemented (zero) unless the processor has the memory save option.

'61 Real Time Clock Counter - this is not an internal interrupt at all, but is used as a counter by the real time clock.

'62 Restricted execution in VM. (Prime 300)

'63 External interrupts use this location.

'64 Page Fault. (Prime 300)

'65 Supervisor Call - an interrupt to this service.

INPUT/OUTPUT

As shown on the block diagram, a Prime computer system can be connected to a variety of peripheral devices. Generally, I/O Data is transferred to and from the B bus from the serial interface or AMLC or SMLC devices. Device types other than the serial interface interact with the B bus through an I/O data buffer and I/O buffer, similar to the way in which the CPU interacts with the memory bus through the memory data and address buffers. Serial input is routed to to the B bus; however, serial output is directed from the D bus directly to the serial interface buffer. Note also, that the control panel has a buffer and is treated as an I/O device; thus setting sense switches can input information directly into the CPU.

Instructions

Instructions in the I/O class govern the transfer of data to and from the peripheral equipment, and also perform some functions in the processor. The class comprises four types of instructions for sending control pulses out to a device, testing conditions in a device for a skip, and moving data or other information out to a device or in from it. An instruction in the I/O class is designated by 1100 in bits 3-6, and the type is indicated by bits 1 and 2; hence the four types of I/O instructions have op codes '14, '34, '54 and '74. Bits 7-10 specify the particular function the instruction is to perform, and bits 11-16 select the device that is to respond to the instruction. The format thus allows sixty-four codes for addressing devices ('00-'77) and sixteen for specifying functions ('00-'77) that a given type of I/O instruction can perform using the addressed device.

Device code '20 is used for communication with the control panel and for controlling interrupts and the real time clock. The other sixty-three codes are available for external devices, but many are assigned to standard equipment.

The meanings of the function codes differ with the type of instruction and the type of device, although some are common to all devices. With the control type of instruction, the function code 00 usually "turns on" or "starts" the device (with whatever meaning that term may have vis-a-vis the particular device), and code '17 initializes the device, making it ready for use. An I/O skip instruction invariably uses function code 00 to determine whether a device is ready and code '04 to determine whether it is requesting an interrupt. The data instructions, in and out, generally use code 00 specifically for real data - as against moving control information, word counts, addresses, or status.

Typically a device interface has a 6-bit device selection network, ready and interrupt enable flags, and logic nets that supply the device code, the device identification, and the number of the slot in which the interface is mounted. The selection network decodes bits 11-16 of the instruction so that only the addressed device responds to signals sent by the processor over the I/O bus. The ready flag indicates just that: the device is ready - meaning it has just completed a task requiring some response by the processor, or it is idle and may be used. Considering devices at the simplest level, the program places an output device in operation by giving a data-out instruction that resets ready and sends the first unit of data - a word or character depending on how the device handles information. When the device has processed the unit of data, it sets ready to indicate that it is ready to receive new data for output. With an input device, the program responds by giving a data-in instruction that not only brings in the data but also resets ready and tells the device to read more data; to end the process the program must actually issue a control command to stop the device. With either type of device, the setting of ready requests an interrupt if the interrupt enable flag is set. If the program does not wish to use the device, it can reset interrupt enable to prevent the

idle state of the device from continually requesting an interrupt.

Every device can supply its device code for use by the interrupt system (although a more complex device may be set up to supply an interrupt addreses specified by the program rather than using its own device code). The program can read the slot number in order to determine the position of any device on the I/O bus (this determines priority with respect to the vectored interrupt) and can read the identification number of each device. The latter number not only identifies the type of device, but also indicates any modification from the standard, which one it is if several of the same type are connected to the bus.

The four basic I/O instruction types:

OCP	Output Control Pulse
SKS	Skip if Condition Satisfied
INA	Input to A Register
OTA	Output from A Register

Control Panel Communication

The program can communicate with the operator via the control panel by virtue of the fact that it can address the panel as an I/O device. With the following instructions, the program reads the contents of the switch register as data or as sense switches and loads a data register whose contents can be displayed in the lights (in no case is a ready test necessary). The instructions assigned to the control panel are:

INA '1620	Read Sense Switches
INA '1720	Read Data Switches
OTA '1720	Load Lights

Processor Serial Interface

Besides the many peripheral devices connected to the I/O bus and controlled by I/O instructions, there is a basic serial interface that is built into the processor and is controlled by special instructions. By means of this device, the program can control the transmission of serial data on four output lines and can receive serial data simultaneously over four input lines. The program handles output by periodically changing the contents of a 4-bit output register in which each bit is connected to a separate output line thus, successive changes in the register contents produce bit-by-bit serial transmission over the lines. Data is received by sampling the input lines to pick up bit-by-bit-serial input. The device operates entirely on EIA standard levels and the lines are available at the back edge connector of the processor board. The program supplies data to and receives data from the lines via A bits 13-16, where line 1 corresponds to bit 13. Input and output are handled by the two instructions:

OSI	Output Serial Interface
ISI	Input Serial Interface

The lines may be used for anything that involves transmission or reception of binary EIA signal. An output line could be used to control a light to signal the operator; an input line might be connected to a switch, allowing a person or a device to supply a binary signal that can be sampled at appropriate times by the program. The lines can also be used for standard data communication where the program is entirely responsible for all timing, for constructing characters with appropriate start and stop bits, and for stripping the data out of received characters. For output, the usual procedure is simply to change the signal on the output line for each bit in a serial transmission. The program determines character length and transmission frequency, and can actually run the output lines at different rates - as would be the case were one line being used for serial transmission and another to control a signal light. Whenever any bit of the output register is changed, information previously given for the other lines must be repeated to keep the appropriate signals on them.

For input, both the frequency and character length must be known. In conventional data communications, an idle line is constantly marking (continuous 1s) and the beginning of an asynchronous character is indicated by a starting space (a zero bit). The usual procedure is to sample the line at five times the bit rate. Upon reading a zero on a line that has been idle, the program should assume it has discovered only a possible space; if a zero is still read at the next two sample times, it can be assumed that the line has a true space rather than a transient, and transmission has started. The program should then read the line at every fifth sample time so that reading is centered within each bit time. If a number of lines are operating, the program must keep track of them separately, i.e., the program must keep the read times centered on each line independently of the others. With sophisticated software, the serial interface could actually be used for a complete data communication channel with even the automatic answering of incoming calls in a private network or the public dial telephone system. For such an arrangement, one input line would be used for data and the others for modem control signals such as ring indicator, clear to send, carrier detected, and data set ready. Output would require three lines: one for data, and two for the control signals request to send and data terminal ready.

External Interrupt

Many I/O devices must be serviced infrequently relative to the processor speed and only a small amount of processor time is required to service them, but they must be serviced within a short time after they request it. Failure to do so within the specified time (which varies among devices) can result in loss of information and certainly results in operating the device below its maximum speed. The external priority interrupt is designed with these considerations in mind, i.e., the use of interruptions in the current program sequence facilitates concurrent operation of the main program and a number of peripheral devices. The interrupt system also allows conditions internal to the processor (traps) to interrupt the program, but here we are concerned only with external interrupts.

Interrupt requests by a device are governed by its interrupt ready and interrupt enable flags. When a device completes an operation it sets the ready flag, and this action requests an interrupt if interrupt enable is set - if interrupt enable has been cleared by the program, the device cannot request an interrupt. The program controls the enabling flags by means of OCP instructions; moreover, the flags in some devices are also connected to the I/O bus data lines, so the program can set up the enabling flags in all such devices at once by means of a mask sent over the bus.

At appropriate times the processor synchronizes any requests that are then being made. Once a request has been synchronized, the device that made it must wait for an interrupt to start. Although the interrupt signal on the bus is disabled once an interrupt starts, the request made by the device remains until the program clears ready or interrupt enable. if the program does clear interrupt enable in a device, that device not only cannot request an interrupt when its ready flag sets, but any request it has already made is voided, so it is no longer waiting for an interrupt (and no I/O skip instruction can determine that it had requested one). However, if ready is left set, setting interrupt enable restores the request.

Before beginning each instruction, the processor takes care of all direct memory requests, including any additional requests that are made while direct memory transfers are being handled. When no more devices are requesting access, the processor starts an interrupt if the external interrupt system is enabled and a device that has priority is requesting an interrupt. The way in which the hardware handles an interrupt and the way in which the program should respond depends upon the interrupt mode.

Standard Interrupt Mode: In standard mode, any device that can make an interrupt request has priority to interrupt any program, even an interrupt service routine, unless the interrupt system is inhibited. The processor starts to service an interrupt by inhibiting the interrupt system so no further interrupts can be started, saving P (which points to the next instruction) in the location addressed by the contents of location '63, and begins the interrupt service routine by resuming normal instruction execution at the location following that in which P was stored.

CAUTION

> The contents of any interrupt location ('63 for the standard interrupt) are always interpreted as a 16-bit absolute address. Therefore, when setting up interrupt locations, the program must make sure not to use addresses larger than available memory.

The service routine should determine which device requires service, save the keys and any parts of the register set that it will use, and service the device. The device can be identified by means of SKS instructions that test for interrupt requests. The program may leave the interrupt inhibited while servicing the device (or devices), or it can enable interrupts and establish a priority structure to allow higher priority devices to interrupt the current routine.

There are two ways in which the program can structure device priority. The service routine establishes a basic priority by the order in which it tests the devices. It can also define higher and lower priorities by setting up the interrupt enable flags in the devices and then reenabling the interrupt. In this way, any device whose interrupt enable flag is clear cannot interrupt the current routine and is therefore defined as being of lower priority, whereas a device that is allowed to interrupt is defined as being of higher priority.

After servicing a device (or all devices found to be interrupting by an SKS chain), the routine should restore the preinterrupt states of the keys and the register set, enable the interrupt, and return to the interrupted program by jumping indirect through the location in which P was stored. If the routine allows interrupts by higher priority devices, then before returning to the interrupted program it should reenable lower priority devices that were not allowed to interrupt the current routine, but will be allowed to interrupt the program to which the processor is returning.

Vectored Interrupt Mode: In vectored mode, the processor responds to an interrupt request from a specific device and has a built-in priority structure such that lower priority devices cannot interrupt while the processor is holding an interrupt for a device of higher priority. The conditions for starting an interrupt are therefore the same as those given for the standard case with one exception: if the processor is already in an interrupt routine, it will go on to the next instruction even if interrupts are enabled, unless the requesting device is of higher priority than that for which the current interrupt is being held. When an interrupt is started and several devices are making requests simultaneously, the processor responds to that requesting device that has the highest priority (mounted in the lowest-numbered slot).

As in standard mode, the processor inhibits further interrupts, saves P as specified by the contents of an interrupt location, and proceeds with the service routine at the position following that in which P was stored. However, unlike a standard interrupt, here there is no fixed interrupt location - instead the location is specified by the device to which the processor is responding. In most cases, the device specifies an address '100 greater than its device code, but a complex device may have an address register for this purpose so that the program can specify the location through which the device will interrupt.

Since the system uses a location unique to each device, there is no need for testing, and the service routine acts only for the

interrupting device (it should of course save keys and registers as usual). There is also a built-in priority determined by bus position, so even if the routine allows interrupts, no device higher on the bus can do so (in other words, all devices in higher-numbered slots are of lessor priority). Moreover, the program can still pick and choose among the nearer devices by adjusting the individual interrupt enable flags. Hence in vectored mode, devices of higher interrupt priority can interrupt a given routine once interrupts are reenabled.

When returning to the interrupted program, the routine must restore the preinterrupt state and either reenable interrupts or reestablish the appropriate priority structure. Furthermore, a routine for a vectored interrupt must also give a specific instruction (CAI) to clear the presently active interrupt so the processor can then respond to requests from devices of lower interrupt priority.

Interrupt Programming: The instructions that control the interrupt system are all of the type with a full word op code, but associated withh the system are two I/O instructions that deal with the mask used for setting up the interrurpt enable flags in certain devices. When power is turned on or the computer is cleared from the control panel, the processor is automatically in standard interrupt mode with interrupts inhibited. The instructions are:

ENB	Enable Interrupt
INH	Inhibit Interrupts
ESIM	Enter Standard Interrupt Mode
EVIM	Enter Vectored Interrupt Mode
CAI	Clear Active Interrupt
SMK	Send Mask
IMK	Input Mask

Timing: The time a device must wait for an interrupt to start depends on how many devices are using interrupts, how long the service routines are for devices of higher priority, and whether the direct memory channels are in use. In vectored mode, a single device will shut out all others of lower priority until a CAI instruction is executed; and the direct memory channels shut out all interrupts when they operate at the maximum rate. If the DMA channels are not in use and only one device is using interrupts, it need never wait longer than the time required for the processor to finish the instruction that is being performed when the request is made. Without delays caused by indirect addressing, the maximum interrupt waiting time is the latency given in the Specification table below.

Programming Suggestions: If the program has little computing to do and is using only one or two fast I/O devices or several slow ones, it may not be necessary to use the interrupt at all. On the other hand, if there are many calculations to perform and the program is using a fast device or data is being processed using several slower devices, then the interrupt is necessary. The critical factors in determining whether to use the interrupt, and in what ways the program should determine priority, are what the program is doing besides input/output

and the time required by the service routines.

A convenient method for handling a large number of priority levels is to use a push-pop stack for saving the machine state. This obviates setting aside so many specific locations for saving registers, and makes it very easy for a routine at any level in a sequence of nested routines to restore the state for the interrupted program.

For those who do program interrupt routines, there are several rules to remember:

1. An interrupt cannot be started until the current instruction is finished. Therefore, do not use lengthy indirect address chains if a device that requires very fast service can request an interrupt.

2. The service routine should save the keys and any parts of the register set that it will use.

3. The JST and ENB instructions delay external interrupt servicing for one full instruction cycle, as do the ILL and UII internal interrupts.

4. The principal function of an interrupt routine is to respond to the situation that caused the interrupt, (e.g., computations that can be performed outside the routine should not be included within it).

5. Before returning to the interrupted program, the routine should restore the keys and the register set, and in vectored mode it must give a CAI.

Direct Memory Access

Handling data transfers between external devices and memory under programmed I/O control requires the execution of several instructions for each word transferred. To allow greater transfer rates, the processor contains eight direct memory channels through which devices, at their own request, can gain direct access to memory using a minimum of processor time. At rates lower than the maximum, the channels free the processor to allow execution of a program concurrently with data transfers for high speed devices such as disk and magnetic tape.

To control a direct memory transfer, the program sets up a device to use a particular channel and sets up a pair of memory locations to define the channel. The channels use locations '20-'37 in the register set, with locations '20 and '21 governing channel one, '22 and '23 governing channel two, and so on to '36 and '37*. To set up the device, the program gives an OTA that supplies the controller the address of the first channel location to be used. The program places a 12-bit word count in the first location, and the address of the first word to be transferred in the second.

FIRST LOCATION

-WORD COUNT	RESERVED
1 2 3 4 5 6 7 8 9 10 11 12	13 14 15 16

SECOND LOCATION

ADDRESS
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

*The processor permits any contiguous pair of locations in the register set to be used, although some locations, such as the program counter or those reserved for microprogram functions, are obviously not appropriate for this purpose. The programmer can use X, A, B, S, and certain other locations when necessary.

The word count is in bits 1-12 and is the twos complement of the number of words to be transferred; the maximum number of words in a single block on one channel is therefore 4096, produced by a negative count of zero (a single device can handle larger blocks by stepping through successive channels). The contents of the second address are interpreted as a 16-bit absolute address regardless of memory size.

When the device requires data service, it requests access to memory via its channel. Between instructions and at various points within an instruction, the processor can pause to handle a transfer. If several devices are waiting for service simutaneously, the first to receive it is the one that is mounted in the lowest-numbered slot. Whenever the processor pauses to handle a DMA request, it handles all pending requests before resuming the instruction, starting an interrupt, or going on to the next instruction.

To service a channel request, the processor accesses the location specified by the channel address, sends its contents out over the bus or stores in it a word taken from the bus as specified by the device, and increments both the address and the word count by one. When the word count overflows (goes to zero), the processor signals the device that the block is complete. Typically, complex device controllers such as those for fixed and moving head disks can automatically chain DMA channels thereby facilitating scatter/gather data transfers.

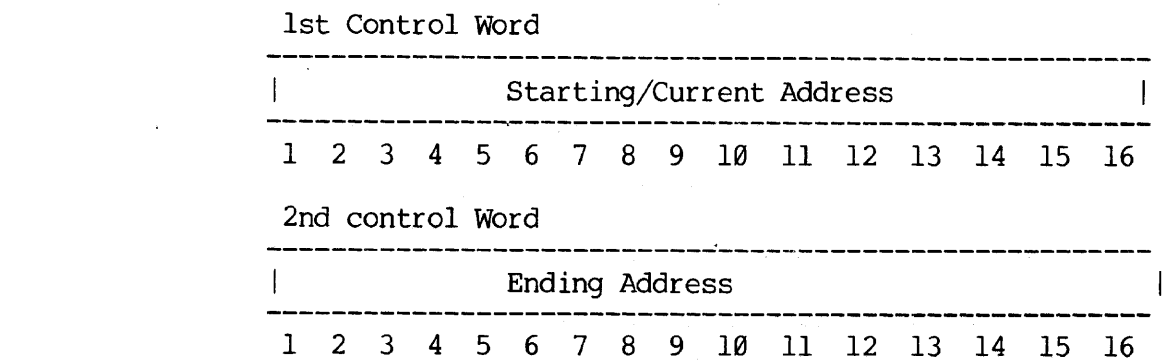
Timing: The time a device must wait for channel access depends on when its request is made within an instruction and how many devices of higher priority are also requesting access; a given device must wait until all devices of higher priority have been serviced, so the highest priority device can preempt all processor time if it requests access at the maximum rate. The microprogram must save certain registers to service the channel, and although it can pause within an instruction, it cannot take direct memory requests while starting an interrupt, so

the worst case waiting time for the highest priority device is 3-4 microseconds for an isolated transfer. But once an initial transfer can be handled at the rate of one every 1.2 microseconds; this allows a maximum of 833,333 words per second, but at this rate all other processing activity is suspended.

Direct Memory Channel, Direct Memory Transfer (DMC, DMT

The DMC and DMT modes of input/output operation extend the speed and flexibility of the standard DMA system available in all Prime computers. DMC is used to extend the number of direct memory channels (up to 2000) and the maximum block size handled by each channel (up to 64K words). DMT increases the maximum direct memory data rate to one million words per second. In contrast to programmed I/O, which requires the execution of several instructions for each word transferred, direct memory transfers reduce the number of instructions needed for I/O control, allow multiple high speed transfers to be handled concurrently and permit processing to be overlapped with I/O operations.

DMC Operation: DMC transfers are controlled in much the same way as DMA transfers; the program specifies a particular channel for an I/O device via an OTA to the device's controller and sets up a pair of control words to establish at what memory location the transfer will begin and how many words will be transferred. Unlike the DMA system, which uses preassigned pairs of high-speed registers to control the DMA channels, DMC transfers are controlled by pairs of adjacent locations in main memory. This permits up to 2000 DMC channels to be specified using memory locations between '64 to '7776. The first word of a control word pair contains the starting address for the transfer, and the second word contains the ending address as illustrated below.



Data blocks of up to 64K words can be transferred at input and output rates of up to 271,739 and 268,817 words per second. At the maximum input and output rates, processing is suspended, while at lower rates processing and I/O transfers are overlapped.

When a device requires DMC service, it requests access to memory via its specified channel. The DMC microcode automatically synchronizes with the instruction currently being executed and causes the processor to pause either at some point within the instruction or upon its

completion. If several devices request servicing simultaneously, the order in which the requests are acknowledged is determined by the priority relationship among the device controllers and the central processor. In general, the controller closest to the processor in the chassis (i.e., mounted in the lowest numbered slot) is given highest priority, while the controller in the highest slot position is assigned the lowest priority.

To service a channel request, the processor accesses the location specified by the first channel control word (starting or current address) and either reads or writes a word as specified by the device controller. The current address is incremented by one after each channel request is serviced until the current address is equal to the ending address, signaling that the block transfer is complete.

Chaining: DMC channels may be chained together to facilitate scatter/gather data transfers. An OTA 14XX loads a device controller with the required DMC set up information from the A register as shown below. A one in bit 5 specifies a DMC transfer (a zero specifies DMC).

Chain No.	1	Channel Address
1 2 3 4	5	6 7 8 9 10 11 1 13 14 15 16

The chain number specifies how many DMC channels in addition to one specified by the channel address portion of the word will be used for a data transfer. A chain number of zero causes the transfer to terminate after one end of range. A chain number greater than zero causes the controller to wait for that number of ends of range plus one before terminating the transfer. In this case, the channel address is automatically incremented by two after each end of range, thereby automatically switching control to the next higher DMC channel.

DMT Operation: Certain controllers are capable of providing the necessary memory addresses for direct memory transfers without using external control words stored in the processor or memory as with DMA or DMC transfers. This permits all channel control functions to be completely overlapped with processor and memory functions thereby increasing the computer's maximum input and output rates to 1,086,956 and 1,041,666 words per second respectively. When operating in the DMT mode, the controller automatically places the memory address of each word to be transferred directly on the I/O bus and terminates the transfer when the end of range has been reached. Because of its high speed and low control overhead, the DMT mode can multiplex data on a word-by-word basis.

Specification Summary*

	DMC	DMT
Maximum Transfer Rate (processing suspended)		
Input (words/sec.)	271,739	1,086,956
Output (words/sec.)	268,817	1,041,666
Interruption To Processing Per Word Transferred at Max.		
Input Rate at Max.	3.68 microsec	920 nanosec
Output Rate	3.72 microsec	960 nanosec
Interleaved Input	4.7 microsec	2 microsec
Interleaved Output	4.7 microsec	2 microsec

*Assumes microsec memory cycle time.

DATA INTEGRITY FEATURES

The following paragraphs summarize data integrity features available on Prime systems, and the purpose of traps, and interrupts within the central processor.

The Prime 200 and 300 CPU's include several levels of automatic, program-independent data integrity check features:

Memory Parity	Checks parity of every 8-bit byte read from high-speed memory. If machine check mode is in effect, an interrupt through location '67 is taken.
Machine Check Mode	Enabled or disabled by EMCM and LMCM instructions. Enables memory parity interrupts and microverification, if present.
Microverification	Optional microcode test routines that test the logic of the entire CPU.
Interrupt	Associated with the user level program. An interrupt performs a control transfer to a location specified by the location associated with the type of interrupt found. This amounts to an indirect JST.
Trap	Associated with microcode. A trap transfers micro-control to a specific trap catching microroutine. Some traps also generate interrupts; others do not.
Memory Parity Error	A parity error in a word read from memory.
Machine Check Error	A parity error in any other situation (in a register, over the I/O bus, etc.).

Memory parity and machine checks are standard on the Prime 200 and 300 and microverification is optional on both processors. These features are not implemented in the Prime 100.

Machine Check Functions

Occurrence of either memory parity error or machine check error in any Prime processor always sets the machine check flag, depending on the type of processor (does it have microverification or not?), its operation mode (normal operating mode or machine check mode), type of error (memory parity or machine check), and type of failure (solid or transient).

Normal Operating Mode (Enabled by MASTER CLEAR or LMCM Instruction

Memory Parity Error: Memory parity error in any Prime machine operating in normal operating mode always sets the machine check flag. There is no interrupt to the operating program. To check for parity error, the operating program may use the SMCS (Skip on Machine Check Set) or SMCR (Skip on Machine Check Reset) instruction. It is then up to the system programmer to handle this problem. Master clear or RMC (Reset Machine Check) can be used to clear the flag.

Machine Check Error: The same procedure as for memory parity error applies.

Machine Check Mode (Enabled by EMCM)

Memory Parity Error: In any Prime processor operating in machine check mode, a memory parity error sets the machine check flag. This causes a microcode trap that executes a microroutine to reset the machine check flag and causes a program interrupt through location '67. Response to this interrupt is decided by the system interrupt service routine.

Machine Check Error (CPU without microverification): A machine check error occurring in a Prime Type 211 or Type 215 central processor running in machine check mode and causes the processor to halt (indicated by the control panel STOP light). If the operator turns the function selector to STOP/STEP, all address lights will be lit.

Machine Check Error (CPU with microverification): A machine check error occurring in a Prime computer with microverification running in machine check mode, initiates execution of the microprogram verification routine to check (verify) proper operation of the processor. The verification routine always clears the machine check flag.

Parity Errors

Several alternative ways of detecting and recovering from parity errors are provided by Prime hardware.

Prime 200 and 300 series computers detect memory parity errors by checking byte parity on each memory read operation. Byte parity errors that occur during data transfers between CPU registers, the backplane and the arithmetic unit, are all classified as machine check parity errors. Both memory and machine check parity errors set the machine check flag.

In the normal operating mode, Prime 200 and 300 computers resemble the Prime 100, which has no parity check hardware. The user may employ the SMCS (Skip on Machine Check Set) and SMCR (Skip on Machine Check Reset) instructions to sense parity errors by testing the machine check flag and may provide subroutines to handle parity errors.

Special instructions (EMCM, LMCM) are provided that cause the computer to enter the machine check mode. In the machine check mode, when a memory parity error sets the machine check flag; a microcode program resets the flag and causes an interrupt through location '67.

Depending on a program's sensitivity to memory parity errors, a user may choose to provide reentry points and a service routine to repeat the calculation or a user may choose some other solution.

In Prime 300 computers, memory is organized into 512-word sectors or pages; and the virtual memory paging technique enables the user to edit out and work around a defective page if interrupts through location '67 occur consistently from a particular area in memory. Also, operating system software checks for bad memory and takes appropriate action to work around that memory. Systems User Guide).

In processors without microverification, machine check parity errors cause the processor to halt, as indicated by the control panel stop light. In this case, turning the function selector to the STOP/STEP position lights all the ADDRESS lights. This action confirms that a CPU parity error has occurred.

In Prime 200 and 300 series machines with the microverify option, a machine check error activates the microcode verification program. This program runs a series of tests on individual registers in the processor, arithmetic unit and I/0 bus. If the entire microverify routine is cycled without a failure being diagnosed in a particular circuit, the parity error is assumed to have been caused by a transient condition. The microverify routine then clears the register set and machine Status Keys and causes interrupt through location '70. The program can then resume execution after the machine state is restored if the user program has been set up to handle this situation.

If the microverification routine encounters a nontransient circuit failure, it continues to cycle as long as the failure persists; and the number of the test is displayed in the ADDRESS lights when the function selector is set at the RUN or LOAD postion. The processor leaves the machine check mode and reenters the normal operating mode when it encounters the LMCM (Leave Machine Check Mode) instruction. Thus, there are two operating modes: normal and machine check. In normal mode, parity errors do not influence program flow unless explicit instructions are inserted into the user's program. In the machine check mode, a parity error during memory read causes an interrupt through location '67 that may be acted upon at the user's discretion; otherwise, program execution continues. In processors without microverification, a machine check parity error halts the machine because processor parity errors are assumed to be more serious than memory parity errors. In machines with the microverify option, if the CPU passes the tests performed by the microverify routine; the assumption is that the trouble was a transient one and processing resumes.

When power is turned-on (and when the MASTER CLEAR button is pressed), processors with microverification perform a CPU circuit integrity check. Then, the computer operates in normal mode. The machine check flag signifying a memory/CPU parity error is ignored in this case.

The microverification routine can be executed at any time by means of the VIRY instruction to assure the integrity of processor circuits. Similarly, the user can assure himself that the CPU is functioning properly by turning on the power and pressing MASTER CLEAR to initiate the microverify sequence.

MICROVERIFICATION

The optional microverify feature provides the Prime processor with a self-test capability. This self-test capability consists of a set of microcode routines that verify the operations that can be successfully performed by the processor. These tests are carefully constructed to verify successively larger portions of the CPU hardware, always building on those portions that are already verified. Table A-1 lists the verification routines and describes the microcode logic that the test exercises.

One of the fundamental tasks of the microverify feature is to verify that the machine checking hardware (Machine Check) is detecting both good and bad parity correctly; this is done with Tests 4, 5, and 6. Machine Check checks byte parity on internal and external data paths. Parity is generated only when necessary (i.e., on shift and ALU operations), and parity is normally transmitted from one register to another unchanged. Each processor register includes parity checking logic. This parity checking normally detects all single bit parity failures and detects looping multiple faults after a few data patterns have been used. A parity error detected by the machine check hardware traps to the microprocessor and causes the verify microcode to be executed.

The verify microcode exercises the processor's control unit as well as the ALU, the registers, and the various data paths (See Figure A-1). Because address and data busses of both memory and I/Ø (bus D, memory bus, bus B, etc.) are tri-state and because all but the memory address bus are bi-directional, a failed component (board) anywhere in the system can stop all data transfer to and from a bus. Two microverify tests (11 and 12) explicitly check for this condition by verifying that both ones and zeroes can be placed upon the busses.

Microverify status is displayed at the control panel indicator lights.

Microverification routines provide a powerful and flexible means of verifying data integrity and preventing the propagation of erroneous data within the system. The Microverification Routines consist of microprogrammed firmware sequences that can test the logic of the

Table A-1. Verification Routines

Test No.	Test Exercises:
0	ALU-0, condition code Jump on not equal
0	RM, RY, EMIT, RA (Register A) all can be set=0 ALU (subtract) Internal busses - transmit 0
1	BB can be loaded from RY
2	Modals and traps are tested to verify clear
3	RX can be incremented BD can transmit bits on lower byte RSC words (it counts and loads correctly) RCM emit 0 can be done
4	RY parity detection Control unit 16 way branch BB, BD data transmission Jump logic
5	RM parity detection Control unit 16 way branch BB, BD data transmission Jump logic
6	Register set parity detection and all tested in 5
7	ALU = -1 BD shift left, BD parity generate RF parity check RSC increment Jump
7	Carry bit, Load, set and clear ALU = subtract Jump Logic
10	Register set with various patterns of bits RSC increment BB, BD various sources and patterns Jump logic
11	I/0 busses, BPA, BPD are able to transmit a one and zero in each bit. Right shift RM, RY, RF data and parity

12	Memory busses, BMA, BMD. Memory Location 5 BMD is tested for a 1 and Ø in each bit Memory timing must work
13	Discovers a parity failure in tests 7-12

entire CPU, verify the reliability of the computer's error detection logic, and test the operation of all data registers, peripheral address and data bus lines, memory address and data bus lines, and the high-speed register set. In addition to these operational tests, the Microverification Routines can also force selected error conditions to occur and then verify that the CPU properly detects those conditions.

Operation

Since the microverification routines are implemented in the CPU's microcode, they are always resident within the system yet do not require memory space for storage. A microverification sequence is initiated whenever the system is cleared from the control panel (Master Clear), a machine check flag is set (hardware detection of a CPU error), or a VIRY (Verify) instruction is executed. For greater operating flexibility, initiation of the sequence following a machine check can be enabled or disabled under program control. (See EMCM and LMCM instructions in Section 7.) This is an important feature since if microverification is enabled for machine checks, the detection of a processor error automatically suspends normal processing for as long as the error condition exists. In certain situations, the user may wish to continue processing to predetermined check points and at such points initiate microverification under program control. When microverification is enabled for machine checks and a transient error is detected, the machine will automatically resume normal operation when proper operational status has been verified.

The result of a pass through the microverification routines depends on CPU status and the method of entering the routines. The various alternatives are summarized in Figure A-1.

Transient Failure

If the entire routine runs (verification routine did not find an error), the failure may have been a transient one, therefore the micro-routine clears the keys and register set, and issues a programmed interrupt through location '70. This returns control to the system program (which can provide for recovery and continued operation).

Solid Failure

The micro-processor will recycle through the microverification routines as long as the failure exists (indefinitely). The number of the failing test is displayed in the address lights.

The VIRY Instruction

The self-test (microverify) routines described in this section may be initiated by a VIRY instruction. The VIRY instruction skips on no error and returns with the failed test number on error. The microverify routines are also initiated by a MASTER CLEAR and by a machine check error.

Table A-2 shows how entry and exit of the verify test operates.

Table A-2. Microverify Entry and Exit

ENTRY to microverify routines will be upon:	EXIT from microverify routines will be upon:
System Clear (MASTER CLEAR)	No Error
CPU parity error (Machine Check)	Successful pass or first error detected.
VIRY instruction	Successful pass or first error detected.

POWER MONITOR AND AUTOMATIC RESTART OPTION

The power monitor and related features combine to provide automatic restart from memory after a power failure has been corrected and AC power is restored. Four distinct and interrelated functions are provided by this option: sensing of line voltage not within operating specifications, storing of processor status information when power fails, battery refreshing of MOS memory, and automatic restart when AC power is restored.

Operation

When the computer is running, AC power is constantly monitored to assure that it satisfies the computer's voltage requirement. Should voltage drop below the specified limit (95 VAC) an automatic power failure interrupt is executed through location '60. This gives the program approximately one millisecond to prepare for loss of AC power. At the end of this interval, a system clear is generated to prevent random logic transitions from altering memory as power is going down. The back-up battery is used to refresh the contents of the MOS memory and provide power to essential processor logic. Use of the battery is indicated by the flashing of the STOP light on the computer's control panel. When power returns to proper operating specifications, the processor automatically restarts at location '1000 and the battery begins recharging.

Equipment Configuration

The equipment consists of two separate components: an assembly with panel which mounts on the computer's power supply, and the battery which may be mounted at either the front or back of the equipment rack. The assembly which mounts on the power supply contains the battery charging circuitry and PC to DC conversion circuitry to supply

16.2 volts $\pm$ 0.5 volts, 2.5 amps
3.5 volts $\pm$ 0.5 volts, 2.0 amps
5.0 volts $\pm$ 0.25 volts, 7.0 amps

with bendback and overvoltage protections. The panel associated with this assembly contains a full-charge indicator, meter terminals, and battery terminals.

The battery is a sealed, gel-electrolyte unit which can be stored or charged in any position. Two 20 Amp hour cells can be housed in the battery mounting, which requires 7" of panel space and a depth of 8".

Specification Summary

Prime input voltage requirement: 95 to 125 VAC, 47 to 63Hz.

Allowable Temporary Voltage Drops:

% Drop From 120 VAC	Maxumum Allowable Duration
100%	12.0 msec.
40%	20.8 msec.
24%	480.0 msec.

Battery: 20 Amp-Hour

Operating Temperature: 0 to 50 C

Battery Back-up Time:

Memory	1 Battery	2 Batteries
4K	6.7 Hrs.	13.4 Hrs.
8K	6	12
24K	3.1	6.2
16K	4.3	8.7
32K	2.4	4.9

AUTOMATIC PROGRAM LOAD

The automatic program load feature enables the operator to load programs from devices such as fixed and moving head disks and paper tape simply by initiating a hardware bootstrap from the control panel. They may also be used to reload programs when power is restored following a power failure. These features save considerable time and effort by eliminating the tedious and error-prone procedure of manually keying in a bootstrap loader one word at a time.

There are three basic types of automatic loaders, one for the fixed-and moving head disks, one for magnetic tape, and one for the ASR and high speed paper tape readers.

All versions are implemented as part of the control panel and the operator uses sense switches to specify the input device.

The disk version reads the contents of sector 0 of the selected disk, storing the words beginning at location '770. After reading the data, the processor begins normal program execution at location '1000. (The program executed; i.e., the data read in from the disk, is entirely at the discretion of the programmer).

The magnetic tape version reads the first record from magnetic tape unit 0 into memory beginning at location '770. After reading the data, the processor begins normal program execution at location '1000. Like the disk, the data read from tape is entirely at the discretion of the programmer.

The paper tape version reads any Prime self-loading tape. Tapes of the assembler, linking loader, text editor and other basic programers are available in self-loading format. Also, any tape punched by the memory dump and load program (MDL) is in the self-loading format and its data is stored in the same part of memory from which it was punched.

SPECIFICATION SUMMARY

Operating Characteristics

SSW 14 15 16 Function Selected

0 0 0=Start at '1000

0 0 1=APL from TTY

0 1 0=APL from HSR

0 1 1=APL from FHD

1 0 0=APL from MHD

1 0 1=APL from Magnetic Tape

Data Rate - Input Device Dependent

APPENDIX B

PRIME 300 ADVANCED FEATURES

The Prime 300 is totally compatible with the Prime 100 and 200. All of the features described in Appendix A are present in the Prime 300.

In addition to the core functionality described in Appendix A, the Prime 300 provides several advanced features: an extended instruction set, virtual memory management hardware, and optional writable control store.

PRIME 300 EXTENDED INSTRUCTIONS

This group of instructions is hardware-implemented only in the Prime 300 and above. The op-codes are obtained by using the extended R-mode instruction format. However, all instructions of this group (except XEC) can be implemented on the Prime 100 or 200 through the Unimplemented Instruction Interrupt (UII) and a UII subroutine library.

Extended Jump Instructions

The Prime 300 introduces nine jump instructions in the R-mode extended, two-word instruction set. (These instructions are also available in the Prime 350 and above in R-mode.) Six of them are conditional on whether the contents of the A register are equal to zero, greater than zero, etc. Others combine a jump with incrementing, decrementing and storing the index register. The instructions are:

JEQ	Jump If Equal to Zero
JNE	Jump If Not Equal to Zero
JLE	Jump If Less Than or Equal to Zero
JGT	Jump If Greater Than Zero
JLT	Jump If Less Than Zero
JGE	Jump If Greater Than or Equal to Zero
JDX	Jump and Decrement Index
JIX	Jump and Increment Index
JSX	Jump and Store Return in Index

Procedure Stack Control

This group of instructions simplifies programming of pure procedures, recursive or reentrant subroutines, and dynamic storage allocation. ENTR alters a stack pointer (S Register) to create an n-word stack frame, and links the new frame with the previous one. CREP saves the program counter in the current stack frame and transfers control to a subroutine. RTN undoes the work of both CREP and ENTR by deleting the current frame and restoring the saved program counter value for the calling program.

ENTR	Enter Recursive Procedure Stack
CREP	Call Recursive Entry Procedure
RTN	Return from Recursive Procedure

Other Extended Instructions

EAA	Effective Address to A Register
XEC	Execute Effective Address Contents as Next Instruction
FLX	Load Double Word Index

VIRTUAL MEMORY

The virtual memory feature (VM) greatly expands the processing and storage resources of a Prime 300. It adds the following basic capabilities:

- Expansion of memory addressing to 262,144 words.
- Hardware protection of software integrity:

  Specific areas within a task (user-level) can be protected against being altered by the task itself

  Tasks can be protected against access and alterations by other tasks

  Executive routines (base and supervisory level) can be protected against alteration by user-level tasks

- Automatic swapping of program segments (pages) between memory and disk.

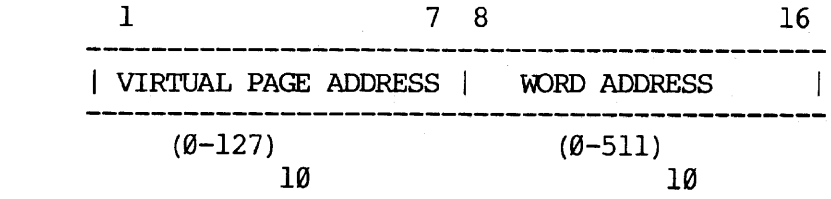
VM's capabilities facilitate:

- Multi-user time-shared disk operating systems.
- Multi-tasking real time operating systems
- Foreground/Background systems with real time multi-user or multi-tasking in the (protected) foregound and batch operations in the background.
- Execution of single programs larger than 64K with or without a disk.

Paging

Paging is a technique of segmenting memory into a fixed length of 512 words or 'pages', intercepting memory acccess, and translating the access from a 'virtual' address to a 'real' or physical address. This translation expands the normal 16 bit address field (2**16= 65,536 words) to 18 bits (2**18=262,144 words). Each translation references a 'page map' that contains the 'real' address for each of the 'virtual' addresses. When paging operations are enabled, the processor is said to be in the 'Paging Mode'.

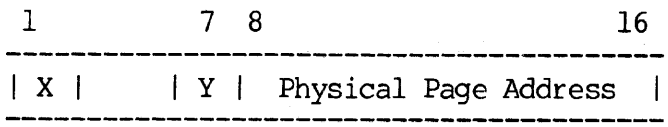
A page is a 512-word contiguous address space whose starting address is a multiple of 512. Normally on a Prime 200 or 300 the maximum address space or segment available to a program is 2**16=65,536 words because of the inherent 16-bit address field. Consider the format of a 16-bit effective address associated with a program segment:



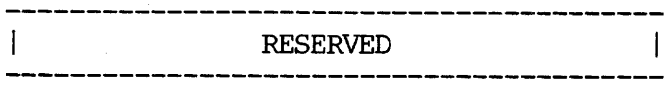
Page Map

In paging mode, the virtual page address (VPA) points to an entry in the page map. That entry contains the real page address and indicates if the page is in memory and if it is 'write-protected'. The contents of the map are created by the base-level executive software. Each user-level program has a map that normally consists of 128 entries - one for each virtual page address. The format for each map entry is:

1st word



2nd word



X: page in memory; 1 = yes, Ø = no

Y: page is write protected; 1 = yes, Ø = no

The second word is usually used by the supervisor for the page's disk address. It can also be used for a second interleaved map.

A page map consists of 128 entries (two words per entry). When a map is in memory, it starts at a multiple of 256 or 256 n +1. To activate a user-level program in the page mode, its map must be in memory and the map's starting address loaded into register '1Ø, the page map address register (PMAR).

As illustrated in Figure B-1, when the user-level program initiates a fetch to an effective address, the following sequence of events occurs:

The virtual page address is doubled and 'added' to the PMAR to create an address that points to the appropriate entry in the map. The physical page address becomes the high order nine bits of the physical memory address; the word address becomes the low order nine bits. This is the full 18 bit physical memory address.

Content Associative Memory Registers (CAM)

The process described above requires an extra memory cycle for each 'virtual' memory reference. The VM features has four Content Associative Memory (CAM) Registers that reduce the overhead to 8Ø ns per memory reference. The CAM registers contain a copy of the four last referenced page map entries. (See Figure B-2.) The contents of these registers are inspected before the memory map. If the CAM registers contain the required map entry, the overhead is 8Ø ns; if not, the memory map is accessed and copied into the particular CAM register that has gone the longest time since it was last accessed. Most programs spend most of the time within a page and would usually find the map entry in CAM. Prime has measured the performance of typical programs operating under its operating systems and found that only 3 percent to 4 percent of map references are not found in CAM.

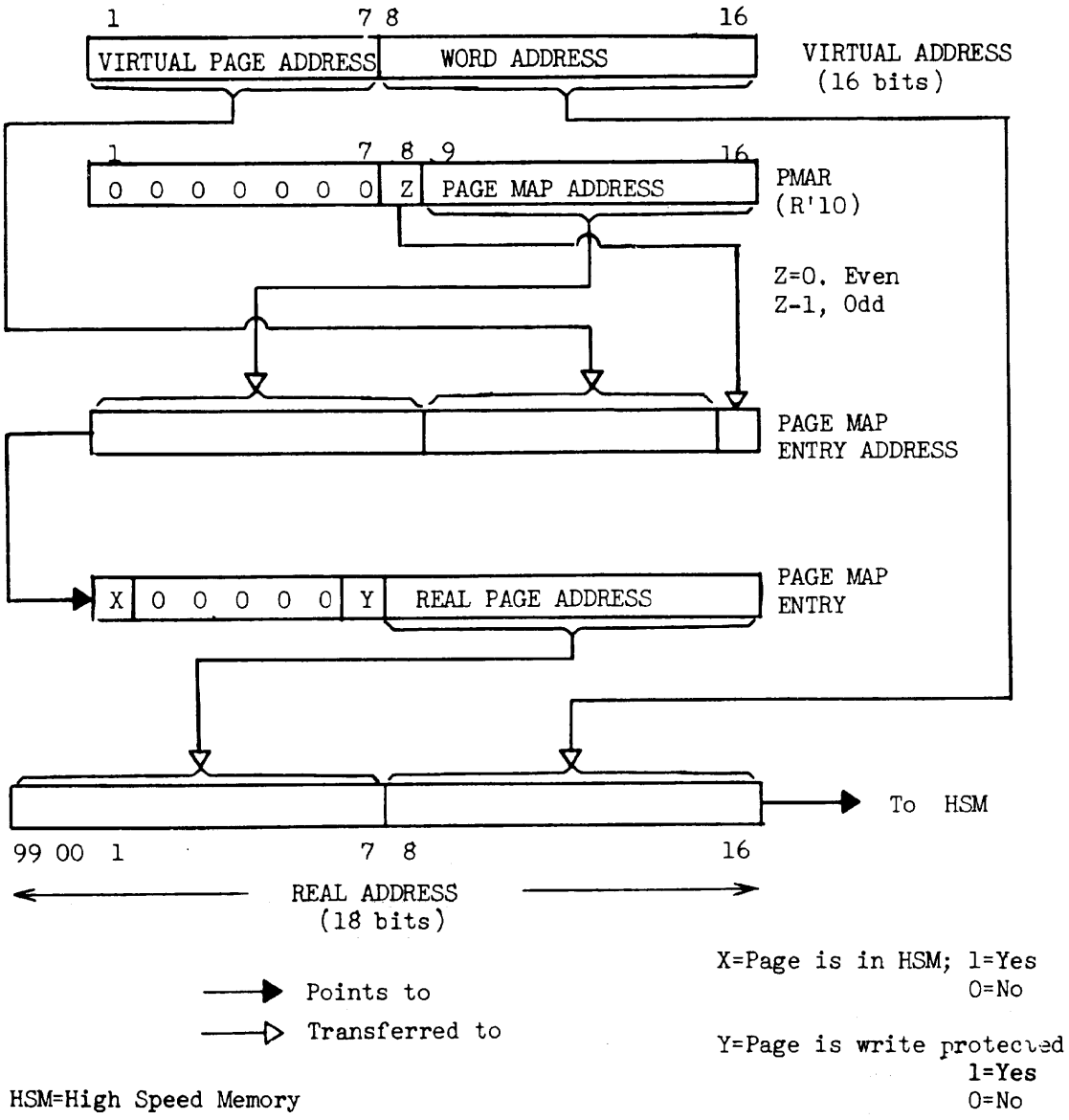
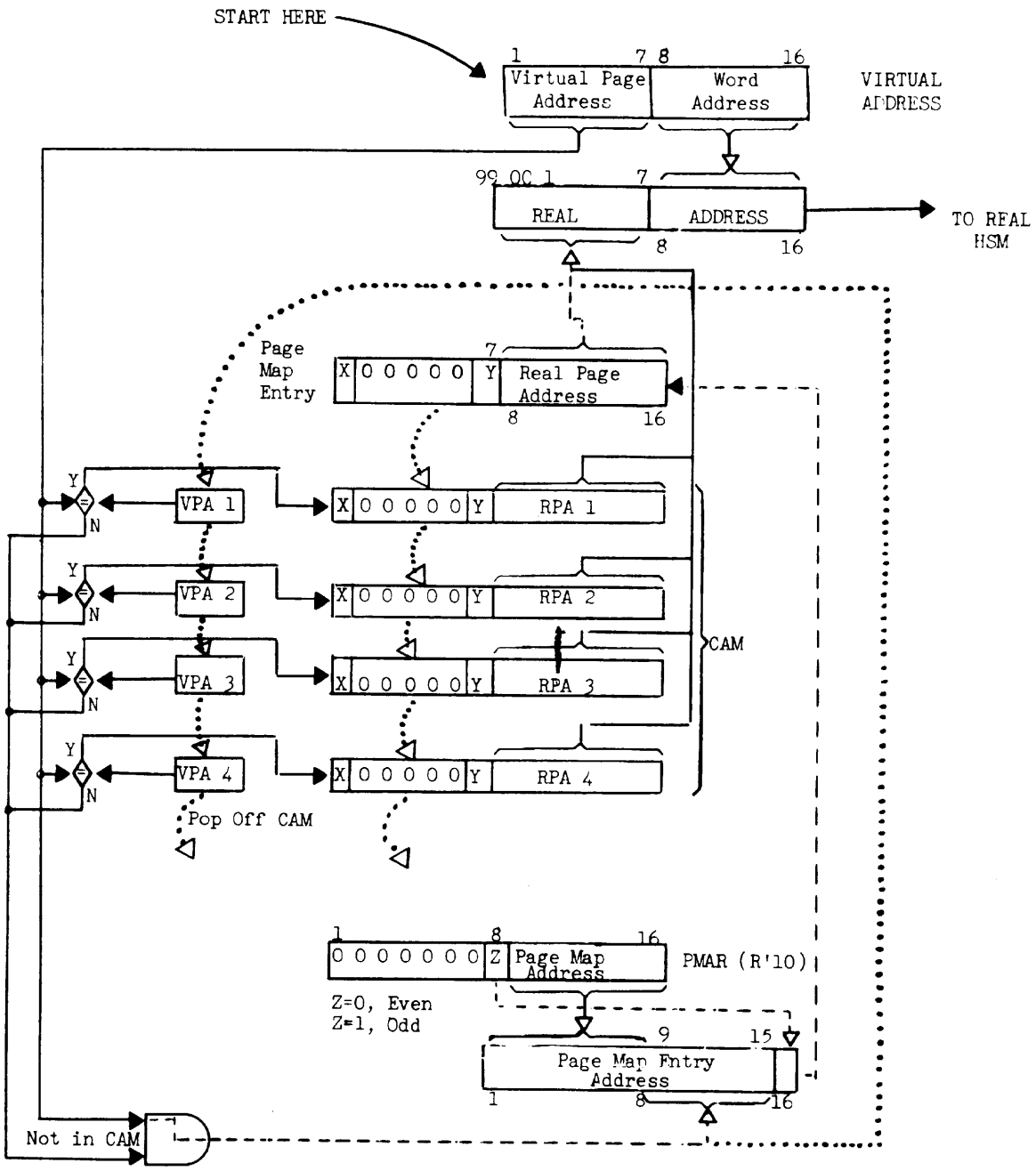


Figure B-1. Physical Address Formation



*If reference was in CAM, that entry is pushed to the top of the CAM

CAM = Content Associative Memory registers
VPA = Virtual Page Address
RPA = Real Page Address
HSM = High Speed Memory
PMAR = Page Map Address Register

Figure B-2. Virtual Memory Effective Address Formation

Page-turning

A disk-based system can effectively expand the 'size' of memory to be as big as the storage size of the disk. This is done by swapping program segments in and out of memory or page-turning. This is a practical technique for many applications since a program usually executes one portion at a time. To make page-turning efficient and practical, the executive program is automatically notified when a task tried to access a page not in memory. The information of which pages are in or out of memory is stored in the page map. This condition is called a 'page fault'. The executive is also told what page the program tried to access and then brings the required page into memory. This feature means that user-level programs can be written without concern for paging; i.e., paging is the executive's responsibility and is transparent to the user-level programs.

The page map entries have a bit that indicates if the page is in memory or on the disk. When a program accesses a new page and its map entry indicates it is not in memory, a page-fault interrupt occurs and the virtual address causing the page-fault is loaded into register '12. The base-level executive program must respond to the interrupt, decide what physical space to use for the new page, load the page off the disk into that space, update the page map entry, and return control over the user-level program. This procedure is referred to as page-turning. One technique used in page-turning is to use the second word of a page map entry to store the page's disk address.

The page map can also specify which pages can be altered and those that cannot. Thus, the map is consulted for each write operation. If a task tries to write into a protected page, the executive is automatically notified of the attempted violation.

Hardware Memory Protection

Virtual Memory provides three levels of protection for maintaining program integrity. The first is inherent in the paging mode operation. Each user-level program is associated with a page map. To activate that program, the executive must load the PMAR with the maps physical address and turn control over to the program. That program can access only those pages in its map.

To reallocate the computer's resources to another program, the executive need only change the PMAR to a new map address. Time-sharing and multi-tasking is facilitated in this manner. The reallocation process can be initiated by an external interrupt, specifically the real time clock, or by a call to the executive from a user-level program.

Within a user-level program, pages can be protected against being altered. This is done by the executive setting the write protect bit in each of the protected page's map entries. When a protected page is accessed by an instruction that would alter its contents, a page write violation interrupt is generated.

A third level of protection is provided by restricting selected programs from executing instructions that would alter the processors control state.

Restricted Execution

User-level programs operate in the restricted execution mode (RXM). In the RXM mode, the executive is automatically notified when the user-level program attempts to execute any one of a class of I/O, interrupt and control instructions. This insures a delineation of responsibilities between the user-level programs and the executive program that controls all I/O, interrupts, and processor modes. In this way, 'stand alone' user-level programs needn't be re-written to run under a time-sharing executive, and they can run without danger of alteration to the executive or other tasks.

The executive program would normally enable the restricted execution mode (RXM) as part of its transfer to a user-level program. This mode is exited by an external interrupt or when a restricted instruction is attempted. The later case causes a RXM interrupt and, as with an external interrupt, would normally branch to the base-level executive program.

Hierarchy of Processing States

Three district processing states can be defined as a direct result of the paging mode and the restricted execution mode. These are:

User-level: usually the 'application' software operating in both the paging and restricted execution mode.

Supervisory-level: A portion of the executive program that needn't be resident in memory and operates in the paging and non-restricted execution modes.

Base-level: A fundamental portion of the executive program that must reside in memory and operates in the non-paging and non-restricted execution modes.

Processing State	Paging Mode	Restricted Execution Mode
User-level	ON	ON
Supervisor-level (executive)	ON	OFF
Base-level (executive)	OFF	OFF

The base-level program is always resident in memory and is responsible for handling:

1. Interrupts (see Table B-1)

    a. I/O devices
    b. Real Time Clock
    c. Page-fault
    d. Restricted execution fault
    e. Disk transfers

2. Bookkeeping

    a. Operating state
    b. Memory allocation

The supervisor level is part of the operating system executive and is a continuation of the base-level executive. The supervisory level can be page-turned and therefore is inherently slower to respond to stimuli than the base. It can be used for such functions as file management and internal operating system commands such as:

- Attach a user file space to a terminal
- Read batch commands from a specified file and execute
- Load a memory image file into memory
- Save memory on a user file

Virtual Memory Instructions

The following instructions control page, virtual and restricted execution modes. They are not emulated by UII software on the Prime 100 or 200, which lack the virtual memory hardware, and so cause an illegal instruction trap.

EPMJ	Enter Paging Mode and Jump
LPMJ	Leave Paging Mode and Jump
ERMJ	Enter Restricted Execution Mode and Jump
EVMJ	Enter Virtual Mode and Jump

WRITABLE CONTROL STORE

The Prime 300 is microprogrammed using a 64-bit wide microprocessor as the control unit. This microprocessor or "inner processor" has a control store containing microinstructions arranged as microprograms. These execute the computer's or "outer processor's" machine instructions, I/O and control panel functions. WSC extends the Prime

Table B-1. Virtual Memory Interrupts

Octal Vector Location	Description
'65	Supervisor Call. Generated by the execution of the SVC instruction. The contents of the program counter is copied into the location addressed in '65. The program counter points to the location following the SVC instruction.
'64	Page-fault. This interrupt occurs when the page map indicates the page required by the executing instruction is not in memory. The program counter pointing to the faulted instruction is copied into the location addressed in '64. The address of the requested page is copied to location '12. The interrupt can occur only when the paging mode is enabled. When a page fault interrupt occurs, paging mode is disabled.
'73	Page Write Violation. This interrupt occurs when the page map indicates the page about to be written into is write protected. The program counter point to the violating instruction is copied into the location addressed in '73. The interrupt can occur only when paging mode is enabled. When a page write violation interrupt occurs, the paging mode is disabled.
'62	Restricted Execution Violation. This interrupt occurs when the following types of instructions try to execute:

I/O	INTERRUPTS	CONTROL
OCP	ENB	HLT
SKS	INH	EMCM
INA	ESIM	LMCM
OTA	EVIM	RMC/RMP
ISI	CAI	VIRY
OSI	SMK	EPMJ
	INK	LPMJ
		EVMJ
		ERMJ

The program counter pointing to the violating instruction is copied into the location addressed '62. The interrupt can occur only when restricted execution mode is enabled. When a restricted execution violation interrupt occurs, the restricted execution mode is disabled.

300's basic control store (512 words of PROM, Programmable Read Only Memory) with 256 words of RAM (Random Access Memory). Microprograms for WCS are written using Prime's Micro Assembler and are loaded from main memory using a Prime-supplied loader. Control is turned over to WCS microprograms by executing a set of special 'Jump to WCS' instructions.

WCS Capabilities

The capabilities of WCS are a function of both the inner and outer processors. The outer processor has 32 general purpose registers, an arithmetic and logic unit, main memory interface, I/O bus interface and an 8-bit auxiliary counter. This general architecture is enhanced by hardware assists to speed the execution of skips, instruction fetches, and decode operations.

The inner processor contains a control unit and control store. It features a three-deep push/pop stack and condition testing to assist in subroutines, and microprogram trapping. The microprocessor fetches and executes microinstructions from control store. The 64-bit wide microinstructions format permits multiple functions to be performed by a single instruction. A narrower word may require several instructions to accomplish an equivalent task. Also, the generalized structure of the outer processor broadens the spectrum of tasks that can be implemented with the powerful inner processor.

Microprogramming can be used to gain a speed advantage and minimize main memory storage compared to in-line programming with the standard instruction set. A microprogrammed subroutine eliminates individual instruction fetches from main memory. A typical microinstruction executes in 280 ns - much faster than main memory cycle time. Thus, when main memory references are minimized, execution speed increases. Furthermore, since a single in-line instruction transfers control to a WCS subroutine, there can also be a more efficient use of main memory.

XCS Board

The Prime microprocessor can address up to 4096 words of control store. To extend the control store past the 512 words located on the central processor (CP) board, a second board is used. The extended control store (XCS) board connects to the CP board in adjacent chassis slots with three short ribbon cables attached to the boards' rear edge connectors.

Both WCS (RAM) and floating point arithmetic (PROM) can use the XCS board depending upon which type processor and option has been ordered.

Programming

Loading WCS: The XCS board also interfaces to the I/O bus to facilitate the loading of WCS from main memory. This is accomplished by first loading the A register with the starting address of WCS and transferring this to the XCS board with an OTA instruction. Data is

transferred to WCS by a sequence of LDA and OTA instructions. The WCS control logic automatically packs four sequential 16-bit words into one 64-bit microinstruction, and advances to the next WCS location.

"Jump to WCS" Instructions: Four "Jump to WCS" instructions have been added to the Prime 300 series processors:

EPMX Enter Page Mode and Jump to XCS

EVMX Enter Virtual Mode and Jump to XCS

(Virtual mode = page mode and restricted execution mode)

ERMX Enter Restricted Execution Mode and Jump to SCS

LPMX Leave Page Mode and Jump to SCX

All of the above are 2 word instructions. The first word is the op code; the second is a pointer to a main memory location containing the address of the microinstruction. This format permits pure procedures to be separated from variables (the control store address may be a variable) and facilitates recursive, re-entrant programming. To transfer control to WCS takes approximately 2.5 microseconds. These are restricted instructions and cause a trap when executed with restricted execution mode enabled. This feature gives the executive software the ability to decide how to handle user-level requests for WCS access.

Use of PMA: Prime's Macro Assembler (PMA) can be used to create special mnemonic representations for microprograms. These are actually one-instruction macro's: a "Jump to WCS" with a unique pointer and WCS address. Once the macro has been defined, the user can refer to his microprogram as a mnemonic, the way he would to a standard Prime instruction.

Software

Micro Assembler: Prime's micro assembler allows the user to create microprograms with full symbolic assembly capabilities. Symbolic source code is assembled and object code created in a format to be loaded into WCS and/or printed in hexadecimal format. The micro assembler runs under PRIMOS and requires a system with at least 32K words of main memory. The Text Editor (ED) is used to enter and modify source statements.

Loader: This enables a PRIMOS user to load WCS from main memory.

Test and Verification: Test and verification routines are also provided.

Microprogramming Course: A one week microprogramming course must be attended as a prerequisite to installing and using the WCS feature. The course is designed to give the student hands-on experience writing, debugging, and executing microprograms. For the student to properly benefit from the course, he should at least familiarize himself with the reading material that will be provided him prior to the course. Preferably he will be experienced in machine level programming, logic design, and computer architecture.

APPENDIX C

INSTRUCTION SUMMARY

This appendix contains a complete list of instructions for the Prime 100 through 500. Each instruction is followed by its octal code, format, function information on addressing mode and hardware availability, and a one line description of the instruction.

The columns in the list are as follows:

R RESTRICTIONS

blank - regular instruction

R - instruction causes a restricted mode violation fault if executed in other than ring 0 (restricted mode violation interrupt on Prime 300)

P - instruction may cause a fault depending on address

W - writable control store instruction, may be programmed in wcs to cause a fault

M - Machine specific - use only on specified CPU. Usually an instruction reserved for operating system, such as EPMJ.

MNEM - a mnemonic name recognized by the assembler PMA.

OPCODE - Octal operation code of the instruction. The codes are indented so that I/O instructions are isolated from generics, and the memory reference and register instructions of the P500 are sorted apart from the MR instructions of the P100-400.

RI - Register (R) and Immediate (I) forms available (P500 memory reference instructions only); Y = YES, N = NO.

FORM Format of instruction:

MNEMONIC	DEFINITION
GEN	Generic
AP	Address Pointer
BRAN	Branch

CHAR	Character
DECI	Decimal
PIO	Programmed I/O
SHFT	Shift
MR	Memory Reference - non I-mode
MRG	Memory Reference - General Register
MRFR	Memory Reference - Floating Register
MRNR	Memory Reference Non Register
RGEN	Register Generic

FUNC Function of instruction:

MNEMONIC	DEFINITION
ADMOD	Addressing Mode
BRAN	Branch
CHAR	Character
CLEAR	Clear field
DECI	Decimal Arithmetic
FIELD	Field Register
FLOAT	Floating Point Arithmetic
INT	Integer
INTGY	Integrity
IO	Input/Output
KEYS	Keys
LOGIC	Logical Operations
LTSTS	Logical Test and Set
MCTL	Machine Control
MOVE	Move
PCTLJ	Program Control and Jump
PRCEX	Process Exchange
QUEUE	Queue Control
SHIFT	Register shift
SKIP	Skip

MODE Addressing modes in which instruction functions as defined:

S	16S or 32S
R	32R or 64R
V	64V (P400-P500)
I	32I (P500)

1 2 3 4 5 How instruction is implemented on each CPU (100 thru 500).

Codes are:

- Not implemented. Do not use this mnemonic on this CPU.

H Implemented by standard hardware.

O Implemented by hardware option or UII library if option is not present.

U Implemented by UII library.

C - How instruction affects C and L bits. codes are:

- C and L are unchanged
1 C = unchanged, L = carry
2 C = overflow status, L = carry
3 C = overflow status, L = indeterminant
4 C = status returned by specific shift, L = indeterminant
5 C = set by instruction, L = last bit out of A1 for left
L = last bit out of B16 for right, including short
6 C = indeterminant, L = indeterminant
7 C = loaded by instruction, L = loaded by instruction
8 C = reset by instruction, L = indeterminate

CC - How instruction affects condition codes. codes are:

- condition codes are not altered
1 condition codes are set to reflect the result of arithmetic operation or compare
4 condition codes are set to reflect result of branch, compare
or logicize operand state.
5 condition codes are indeterminant
6 condition codes are loaded by instruction
7 special results are shown in condition codes for this instruction

DESCRIPTION - a brief description of the instruction

R	MNEM	OP CODE	RI	FORM	FUNC	MODE		1	2	3	4	5	C	CC	DESCRIPTION
	A	02	YY	MRGR	INT		I	-	-	-	-	H	2	1	Add Fullword
	A1A	141206		GEN	INT	S R V		H	H	H	H	H	2	1	Add One to A
	A2A	140304		GEN	INT	S R V		H	H	H	H	H	2	1	Add Two to A
	ABQ	141716		AP	QUEUE	V		-	-	-	H	H	-	6	Add to Bottom of Queue
	ABQ	(70)134		AP	QUEUE		I	-	-	-	-	H	-	7	Add to Bottom of Queue
	ACA	141216		GEN	INT	S R V		H	H	H	H	H	2	1	Add C-Bit to A
	ADD	06		MR	INT	S R V		H	H	H	H	H	2	1	Add
	ADL	06 03		MR	INT	V		-	-	-	H	H	2	1	Add Long
	ADLL	141000		GEN	INT	V		-	-	-	H	H	2	1	Add Link Bit to L
	ADLR	(70)014		RGEN	INT		I	-	-	-	-	H	-	7	Add Link to R
	AH	12	YY	MRGR	INT		I	-	-	-	-	H	2	1	Add Halfword
	ALFA 0	001301		GEN	FIELD	V		-	-	-	H	H	6	5	Add Long Integer to Field Address
	ALFA 1	001311		GEN	FIELD	V		-	-	-	H	H	6	5	Add Long Integer to Field Address
	ALL	0414XX		SHFT	SHIFT	S R V		H	H	H	H	H	4	5	A Left Logical
	ALR	0416XX		SHFT	SHIFT	S R V		H	H	H	H	H	4	5	A Left Rotate
	ALS	0415XX		SHFT	SHIFT	S R V		H	H	H	H	H	4	5	A Left Shift
	ANA	03		MR	LOGIC	S R V		H	H	H	H	H	-	-	AND
	ANL	03 03		MR	LOGIC	V		-	-	-	H	H	-	-	AND Long
	ARFA	(70)161,171		RGEN	FIELD		I	-	-	-	-	H	-	7	Update Field Address Register
	ARGT	000605		GEN	PCTLJ	V	I	-	-	-	H	H	-	-	Argument Transfer
	ARL	0404XX		SHFT	SHIFT	S R V		H	H	H	H	H	4	5	A Right Logical
	ARR	0406XX		SHFT	SHIFT	S R V		H	H	H	H	H	4	5	A Right Rotate
	ARS	0405XX		SHFT	SHIFT	S R V		H	H	H	H	H	4	5	A Right Shift
	ATQ	141717		AP	QUEUE	V		-	-	-	H	H	-	6	Add to Top of Queue
	ATQ	(70)135		AP	QUEUE		I	-	-	-	-	H	-	7	Add to Top of Queue
	BCEQ	141602		BRAN	BRAN	V		-	-	-	H	H	-	-	Branch if CC = 0
	BCGE	141605		BRAN	BRAN	V		-	-	-	H	H	-	-	Branch if CC $\geq$ 0
	BCGT	141601		BRAN	BRAN	V		-	-	-	H	H	-	-	Branch if CC > 0
	BCLE	141600		BRAN	BRAN	V		-	-	-	H	H	-	-	Branch if CC $\leq$ 0
	BCLT	141604		BRAN	BRAN	V		-	-	-	H	H	-	-	Branch if CC < 0
	BCNE	141603		BRAN	BRAN	V		-	-	-	H	H	-	-	Branch if CC .NE. 0
	BCR	141705		BRAN	BRAN	V		-	-	-	H	H	-	-	Branch if C-Bit = 0
	BCS	141704		BRAN	BRAN	V		-	-	-	H	H	-	-	Branch if C-Bit = 1
	BDX	140734		BRAN	BRAN	V		-	-	-	H	H	-	-	Decrement X; Branch if X = 0
	BDY	140724		BRAN	BRAN	V		-	-	-	H	H	-	-	Decrement Y; Branch if Y = 0
	BEQ	140612		BRAN	BRAN	V		-	-	-	H	H	-	4	Branch if A = 0
	BFEQ	141612		BRAN	BRAN	V		-	-	-	H	H	-	4	Branch if F = 0
	BFEQ	(50)122		RGEN	BRAN		I	-	-	-	-	H	-	4	Branch if F = 0
	BFGE	141615		BRAN	BRAN	V		-	-	-	H	H	-	4	Branch if F $\geq$ 0
	BFGE	(50)125		RGEN	BRAN		I	-	-	-	-	H	-	4	Branch if F $\geq$ 0
	BFGT	141611		BRAN	BRAN	V		-	-	-	H	H	-	4	Branch if F > 0
	BFGT	(50)121		RGEN	BRAN		I	-	-	-	-	H	-	4	Branch if F > 0
	BFLE	141610		BRAN	BRAN	V		-	-	-	H	H	-	4	Branch if F $\leq$ 0
	BFLE	(50)120		RGEN	BRAN		I	-	-	-	-	H	-	4	Branch if F $\leq$ 0
	BFLT	141614		BRAN	BRAN	V		-	-	-	H	H	-	4	Branch if F < 0
	BFLT	(50)124		RGEN	BRAN		I	-	-	-	-	H	-	4	Branch if F < 0
	BFNE	141613		BRAN	BRAN	V		-	-	-	H	H	-	4	Branch if F .NE. 0
	BFNE	(50)123		RGEN	BRAN		I	-	-	-	-	H	-	4	Branch if F .NE. 0
	BGE	140615		BRAN	BRAN	V		-	-	-	H	H	-	4	Branch if A $\geq$ 0
	BGT	140611		BRAN	BRAN	V		-	-	-	H	H	-	4	Branch if A > 0
	BHD1	(50)144		RGEN	BRAN		I	-	-	-	-	H	-	-	Decrement H by One; Branch if H = 0
	BHD2	(50)145		RGEN	BRAN		I	-	-	-	-	H	-	-	Decrement H by Two; Branch if H = 0
	BHD4	(50)146		RGEN	BRAN		I	-	-	-	-	H	-	-	Decrement H by Four; Branch if H = 0
	BHEQ	(50)105		RGEN	BRAN		I	-	-	-	-	H	-	4	Branch if H = 0
	BHEQ	(50)112		RGEN	BRAN		I	-	-	-	-	H	-	4	Branch if H = 0

R	MNEM	OP CODE	RI	FORM	FUNC	MODE		1	2	3	4	5	C	CC	DESCRIPTION
	BHGT	(50)111		RGEN	BRAN		I	-	-	-	-	H	-	4	Branch if H > 0
	BHI1	(50)140		RGEN	BRAN		I	-	-	-	-	H	-	-	Increment H by One; Branch if H = 0
	BHI2	(50)141		RGEN	BRAN		I	-	-	-	-	H	-	-	Increment H by Two; Branch if H = 0
	BHI4	(50)142		RGEN	BRAN		I	-	-	-	-	H	-	-	Increment H by One; Branch if H = 0
	BHLE	(50)110		RGEN	BRAN		I	-	-	-	-	H	-	4	Branch if H ≤ 0
	BHLT	(50)104		RGEN	BRAN		I	-	-	-	-	H	-	4	Branch if H < 0
	BHNE	(50)113		RGEN	BRAN		I	-	-	-	-	H	-	4	Branch if H is not equal to 0
	BIX	141334		BRAN	BRAN	V		-	-	-	H	H	-	-	Increment X and Branch
	BIY	141324		BRAN	BRAN	V		-	-	-	H	H	-	-	Increment Y and Branch
	BLE	140610		BRAN	BRAN	V		-	-	-	H	H	-	4	Branch if A ≤ 0
	BLEQ	140702		BRAN	BRAN	V		-	-	-	H	H	-	4	Branch if L = 0
	BLGE	140615		BRAN	BRAN	V		-	-	-	H	H	-	4	Branch is L ≥ 0
	BLGT	140701		BRAN	BRAN	V		-	-	-	H	H	-	4	Branch if L > 0
	BLLE	140700		BRAN	BRAN	V		-	-	-	H	H	-	4	Branch if L ≤ 0
	BLLT	140614		BRAN	BRAN	V		-	-	-	H	H	-	4	Branch if L < 0
	BLNE	140703		BRAN	BRAN	V		-	-	-	H	H	-	4	Branch if L .NE. 0
	BLR	141707		BRAN	BRAN	V		-	-	-	H	H	-	-	Branch if L-Bit = 0 (Reset)
	BLS	141706		BRAN	BRAN	V		-	-	-	H	H	-	-	Branch if L-Bit = 1 (Set)
	BLT	140614		BRAN	BRAN	V		-	-	-	H	H	-	4	Branch if A < 0
	BMEQ	141602		BRAN	BRAN	V		-	-	-	H	H	-	-	Branch if Magnitude = 0
	BMGE	141706		BRAN	BRAN	V		-	-	-	H	H	-	-	Branch if Magnitude is ≥ 0
	BMGT	141710		BRAN	BRAN	V		-	-	-	H	H	-	-	Branch if Magnitude is > 0
	BMLE	141711		BRAN	BRAN	V		-	-	-	H	H	-	-	Branch if Magnitude is ≤ 0
	BMLT	141707		BRAN	BRAN	V		-	-	-	H	H	-	-	Branch if Magnitude is < 0
	BMNE	141603		BRAN	BRAN	V		-	-	-	H	H	-	-	Branch if Magnitude is .NE. 0
	BNE	140613		BRAN	BRAN	V		-	-	-	H	H	-	4	Branch if A .NE. 0
	BRBR	(50)040-077		BRAN	BRAN		I	-	-	-	-	H	-	-	Branch if bit n = 0
	BRBS	(50)000-037		RGEN	BRAN		I	-	-	-	-	H	-	-	Branch if R bit n = 1
	BRD1	(50)134		RGEN	BRAN		I	-	-	-	-	H	-	-	Decrement R by One; Branch if R=0
	BRD2	(50)135		RGEN	BRAN		I	-	-	-	-	H	-	-	Decrement R by Two; Branch if R=0
	BRD4	(50)136		RGEN	BRAN		I	-	-	-	-	H	-	-	Decrement R by Four; Branch if R=0
	BREQ	(50)102		RGEN	BRAN		I	-	-	-	-	H	-	4	Branch if R = 0
	BRGE	(50)105		RGEN	BRAN		I	-	-	-	-	H	-	4	Branch if R ≥ 0
	BRGT	(50)101		RGEN	BRAN		I	-	-	-	-	H	-	4	Branch if R > 0
	BRI1	(50)130		RGEN	BRAN		I	-	-	-	-	H	-	-	Increment R by one and branch if 0
	BRI2	(50)131		RGEN	BRAN		I	-	-	-	-	H	-	-	Increment R by 2 and branch if 0
	BRI4	(50)132		RGEN	BRAN		I	-	-	-	-	H	-	-	Increment R by 4 and branch if 0
	BRLE	(50)100		RGEN	BRAN		I	-	-	-	-	H	-	4	Branch if R ≤ 0
	BRLT	(50)104		RGEN	BRAN		I	-	-	-	-	H	-	4	Branch if R < 0
	BRNE	(50)103		RGEN	BRAN		I	-	-	-	-	H	-	4	Branch if R .NE. 0
	C	61	YY	MRGR	INT		I	-	-	-	-	H	1	1	Compare Fullword
R	CAI	000411		GEN	IO	S R V	I	H	H	H	H	H	-	-	Clear Active Interrupt
	CAL	141050		GEN	CLEAR	S R V		H	H	H	H	H	-	-	Clear A Left
	CALF	000705		AP	PCTLJ	V		-	-	-	H	H	7	6	Call Fault Handler
	CAR	141044		GEN	CLEAR	S R V		H	H	H	H	H	-	-	Clear A Right
	CAS	11		MR	SKIP	S R V		H	H	H	H	H	1	1	Compare A and Skip
	CAZ	140214		GEN	SKIP	S R V		H	H	H	H	H	1	1	Compare A with Zero
	CEA	000111		GEN	PCTLJ	S R		H	H	H	H	H	-	-	Compute Effective Address

R	MNEM	OP CODE	RI	FORM	FUNC	MODE	1	2	3	4	5	C	CC	DESCRIPTION
	CGT	001314		BRAN	BRAN	V	-	-	-	H	H	6	5	Computed GOTO
	CGT	(70)026		BRAN	BRAN	I	-	-	-	-	H	-	7	Computed GOTO
	CH	71	YY	MRGR	INT	I	-	-	-	-	H	1	1	Compare Halfword
	CHS	140024		GEN	INT	S R V	H	H	H	H	H	-	-	Change Sign
	CHS	(70)040		RGEN	INT	I	-	-	-	-	H	-	-	Change Sign
	CLS	11 03		MR	LOGIC	V	-	-	-	H	H	1	1	Compare
	CMA	140401		GEN	LOGIC	S R V	H	H	H	H	H	-	-	Complement A
	CMH	(70)045		RGEN	LOGIC	I	-	-	-	-	H	-	-	Complement
	CR	(70)056		RGEN	CLEAR	I	-	-	-	-	H	-	-	Clear
	CRA	140040		GEN	CLEAR	S R V	H	H	H	H	H	-	-	Clear A
	CRB	140015		GEN	CLEAR	S R V	H	H	H	H	H	-	-	Clear B
	CRBL	(70)062		RGEN	CLEAR	I	-	-	-	-	H	-	-	Clear High Byte 1 (Left)
	CRBR	(70)063		RGEN	CLEAR	I	-	-	-	-	H	-	-	Clear High Byte 2 (Right)
	CRE	141404		GEN	CLEAR	V	-	-	-	H	H	-	-	Clear E
	CREP	10 02		MR	PCTLJ	R	U	U	H	H	H	-	-	Call Recursive Entry Procedure
	CRHL	(70)054		RGEN	CLEAR	I	-	-	-	-	H	-	-	Clear Left Half
	CRHR	(70)055		RGEN	CLEAR	I	-	-	-	-	H	-	-	Clear Right Half
	CRL	140010		GEN	CLEAR	S R V	H	H	H	H	H	-	-	Clear Long
	CRLE	141410		GEN	CLEAR	V	-	-	-	H	H	-	-	Clear L and E
	CSA	140320		GEN	MOVE	S R V	H	H	H	H	H	5	-	Copy Sign of A
	CSR	(70)041		RGEN	MOVE	I	-	-	-	-	H	-	-	Copy Sign
R	CXCS	001714		GEN	MCTL	V	-	-	-	H	H	-	-	Extended Control Store
	D	62	YY	MRGR	INT	I	-	-	-	-	H	3	1	Divide Fullword
	DAD	06		MR	INT	S R	O	O	H	H	H	2	1	Double Add
	DBL	000007		GEN	INT	S R	H	H	H	H	H	-	-	Double Precision
	DBLE	(70)106,116		RGEN	FLPT	I	-	-	-	-	H	-	-	Convert Single to Double
	DFA	15,17	YY	MRFR	FLPT	I	-	-	-	-	H	3	1	Double Floating Add
	DFAD	06 02		MR	FLPT	R V	U	U	H	H	H	3	5	Double Floating Add
	DFC	05,07	YY	MRFR	FLPT	I	-	-	-	-	H	-	1	Double Floating Compare
	DFCM	140574		GEN	FLPT	R V	-	O	O	H	H	3	5	Double Floating Complement
	DFCM	(70)144,154		RGEN	FLPT	I	-	-	-	-	H	3	1	Double Floating Complement
	DFCS	11 02		MR	FLPT	R V	U	U	H	H	H	6	5	Double Floating Compare and Skip
	DFD	31,33	YY	MRFR	FLPT	I	-	-	-	-	H	3	1	Double Floating Divide
	DFDV	17 02		MR	FLPT	R V	U	O	O	H	H	3	5	Double Floating Divide
	DFL	01,03	YY	MRFR	FLPT	I	-	-	-	-	H	-	-	Double Floating Load
	DFLD	02 02		MR	FLPT	R V	U	U	H	H	H	-	-	Double Floating Load
	DFLX	15 02		MR	FLPT	V	-	-	-	H	H	-	-	Load Double Floating Index
	DFM	25,27	YY	MRFR	FLPT	I	-	-	-	-	H	3	1	Double Floating Multiply
	DFMP	16 02		MR	FLPT	R V	U	O	O	H	H	3	5	Double Floating Multiply
	DFS	21,23	YY	MRFR	FLPT	I	-	-	-	-	H	3	1	Double Floating Subtract
	DFSB	07 02		MR	FLPT	R V	U	U	H	H	H	3	5	Double Floating Subtract
	DFST	11,13	NN	MRFR	FLPT	I	-	-	-	-	H	-	-	Double Floating Store
	DFST	04 02		MR	FLPT	R V	U	U	H	H	H	-	-	Double Floating Store
	DH	72	YY	MRGR	INT	I	-	-	-	-	H	3	1	Divide Halfword
	DH1	(70)130		RGEN	INT	I	-	-	-	-	H	2	1	Decrement by One
	DH2	(70)131		RGEN	INT	I	-	-	-	-	H	2	1	Decrement by Two
	DIV	17		MR	INT	V	-	-	-	H	H	3	5	Divide
	DIV	17		MR	INT	S R	O	O	H	H	H	3	5	Divide
	DLD	02		MR	MOVE	S R	O	O	H	H	H	-	-	Double Load
	DM	60	NN	MRNR	INT	I	-	-	-	-	H	-	1	Decrement Fullword
	DMH	70	NN	MRNR	INT	I	-	-	-	-	H	-	1	Decrement Halfword
	DR1	(70)124		RGEN	INT	I	-	-	-	-	H	2	1	Decrement by One
	DR2	(70)125		RGEN	INT	I	-	-	-	-	H	2	1	Decrement by Two
	DRX	140210		GEN	SKIP	S R V	H	H	H	H	H	-	-	Decrement and Replace Index
	DSB	07		MR	INT	S R	O	O	H	H	H	2	1	Double Subtract
	DST	04		MR	MOVE	S R	O	O	H	H	H	-	-	Double Store
	DVL	17 03		MR	INT	V	-	-	-	H	H	3	5	Divide Long
	E16S	000011		GEN	ADMOD	S R V I	H	H	H	H	H	-	-	Enter 16S Mode

R	MNEM	OP CODE	RI	FORM	FUNC	MODE	1	2	3	4	5	C	CC	DESCRIPTION
	E32I	001010		GEN	ADMOD	S R V	-	-	-	-	H			Enter 32I Mode
	E32R	001013		GEN	ADMOD	S R V I	H	H	H	H	H	-	-	Enter 32R Mode
	E32S	000013		GEN	ADMOD	S R V I	H	H	H	H	H	-	-	Enter 32S Mode
	E64R	001011		GEN	ADMOD	S R V I	H	H	H	H	H	-	-	Enter 64R Mode
	E64V	000010		GEN	ADMOD	S R V I	-	-	-	H	H	-	-	Enter 64V Mode
	EAA	01 01		MR	MOVE	R	U	U	H	H	H	-	-	Effective Address to A
	EAFA 0	001300		AP	FIELD	V I	-	-	-	H	H	-	-	Load Field Address Register 0
	EAFA 1	001310		AP	FIELD	V I	-	-	-	H	H	-	-	Load Field Address Register 1
	EAL	01 01		MR	MOVE	V	-	-	-	H	H	-	-	Effective Address to L
	EALB	42	NN	MRNR	PCTLJ	I	-	-	-	-	H	-	-	Effective Address to Link Base
	EALB	13 02		MR	PCTLJ	V	-	-	-	H	H	-	-	Effective Address to LB
	EAR	63	NN	MRGR	PCTLJ	I	-	-	-	-	H	-	-	Effective Address to Register
	EAXB	52	NN	MRNR	PCTLJ	I	-	-	-	-	H	-	-	Effective Address to Temporary Base
	EAXB	12 02		MR	MOVE	V	-	-	-	H	H	-	-	Effective Address to Temporary Base
	EIO	34	NN	MRGR	IO	I	-	-	-	-	H	-	2	Execute I/O
R	EIO	14 01		MR	IO	V	-	-	-	H	H	-	-	Execute I/O
R	EMCM	000503		GEN	INTGY	S R V I	-	H	H	H	H	-	-	Enter Machine Check Mode
R	ENB	000401		GEN	IO	S R V I	H	H	H	H	H	-	-	Enable Interrupts
	ENTR	01 03		MR	PCTLJ	R	U	U	H	H	H	-	-	Enter Recursive Procedure Stack
	EPMJ	000217		MR	MCTL	S R	-	-	H	-	-	-	-	Enter Paging Mode and Jump
M	EPMX	000237		MR	MCTL	S R	-	-	H	-	-	-	-	Enter Paging Mode and Jump to XCS
	ERA	05		MR	LOGIC	S R V	H	H	H	H	H	-	-	Exclusive OR to A
	ERL	05 03		MR	LOGIC	V	-	-	-	H	H	-	-	Exclusive OR to L
M	ERMJ	000701		MR	MCTL	S R	-	-	H	-	-	-	-	Enter Restricted Execution Mode and Jump
M	ERMX	000721		MR	MCTL	S R	-	-	H	-	-	-	-	Enter Restricted Execution Mode and Jump to WCS
R	ESIM	000415		GEN	IO	S R V I	H	H	H	H	H	-	-	Enter Standard Interrupt Mode
R	EVIM	000417		GEN	IO	S R V I	H	H	H	H	H	-	-	Enter Vectored Interrupt Mode
M	EVMJ J	000703		MR	MCTL	S R	-	-	H	-	-	-	-	Enter Vectored Mode and Jump
M	EVMX	000723		MR	MCTL	S R	-	-	H	-	-	-	-	Enter Virtual Mode and Jump to WCS
	FA	14,16 YY		MRFR	FLPT	I	-	-	-	-	H	3	1	Floating Add
	FAD	06 01		MR	FLPT	R V	U	U	H	H	H	3	5	Floating Add
	FC	04,06 YY		MRFR	FLPT	I	-	-	-	-	H	-	1	Floating Compare
	FCM	140530		GEN	FLPT	R V	-	O	O	H	H	3	5	Floating Complement
	FCM	(70)100,110		RGEN	FLPT	I	-	-	-	-	H	3	1	Floating Complement
	FCS	11 01		MR	FLPT	R V	U	U	H	H	H	6	5	Floating Compare and Skip
	FD	30,32 YY		MRFR	FLPT	I	-	-	-	-	H	3	1	Floating Divide
	FDBL	140016		GEN	FLPT	V	-	-	-	H	H	-	-	Convert Single to Double Float
	FDV	17 01		MR	FLPT	R V	U	O	O	H	H	3	5	Floating Divide
	FL	00,02 YY		MRFR	FLPT	I	-	-	-	-	H	-	-	Floating Load
	FLD	02 01		MR	FLPT	R V	U	U	H	H	H	-	-	Floating Load
	FLOT	140550		GEN	FLPT	R V	-	O	O	H	H	6	5	Float
	FLT	(70)105,115		RGEN	FLPT	I	-	-	-	-	H	-	-	Convert Integer to Floating
	FLTA	140532		GEN	FLPT	V	-	-	-	H	H	3	5	Convert Integer to Floating
	FLTH	(70)102,112		RGEN	FLPT	I	-	-	-	-	H	-	-	Convert Halfword to Floating

R	MNEM	OP CODE	RI	FORM	FUNC	MODE	1	2	3	4	5	C	CC	DESCRIPTION
	FLTL	140535		GEN	FLPT	V	-	-	-	H	H	8	5	Convert Long Integer to Floating
	FLX	15 01		MR	FLPT	R V	U	U	H	H	H	-	-	Load Double Word Index
	FM	24,26	YY	MRFR	FLPT	I	-	-	-	-	H	3	1	Floating Multiply
	FMP	16 01		MR	FLPT	R V	U	O	O	H	H	3	5	Floating Multiply
	FRN	140534		GEN	FLPT	R V	U	O	O	H	H	3	5	Floating Round
	FRN	(70)107,117		RGEN	FLPT	I	-	-	-	-	H	3	1	Floating Round
	FS	20,22	YY	MRFR	FLPT	I	-	-	-	-	H	3	1	Floating Subtract
	FSB	07 01		MR	FLPT	R V	U	U	H	H	H	3	5	Floating Subtract
	FSGT	140515		GEN	FLPT	R V	-	O	O	H	H	-	4	Floating Skip if > 0
	FSLE	140514		GEN	FLPT	R V	-	O	O	H	H	-	4	Floating Skip ≤ 0
	FSMI	140512		GEN	FLPT	R V	-	O	O	H	H	-	4	Floating Skip if Minus
	FSNZ	140511		GEN	FLPT	R V	-	O	O	H	H	-	4	Floating Skip if Not Zero
	FSPL	140513		GEN	FLPT	R V	-	O	O	H	H	-	4	Floating skip if Plus
	FST	10,12	NN	MRFR	FLPT	I	-	-	-	-	H	-	-	Floating Store
	FST	04 01		MR	FLPT	R V	U	U	H	H	H	6	6	Floating Store
	FSZE	140510		GEN	FLPT	R V	-	O	O	H	H	-	4	Floatin Skip if Zero
R	HLT	000000		GEN	MCTL	S R V I	H	H	H	H	H	-	-	Halt
	I	41	YN	MRGR	MOVE	I	-	-	-	-	H	-	-	Interchange Register and Memory-Fullword
	IAB	000201		GEN	MOVE	S R V	H	H	H	H	H	-	-	Interchange A and B
	ICA	141340		GEN	MOVE	S R V	H	H	H	H	H	-	-	Interchange Characters in A
	ICBL	(70)065		RGEN	MOVE	I	-	-	-	-	H	-	-	Interchange Bytes and Clear Left
	ICBR	(70)066		RGEN	MOVE	I	-	-	-	-	H	-	-	Interchange Bytes and Clear Right
	ICHL	(70)060		RGEN	MOVE	I	-	-	-	-	H	-	-	Interchange Halves and Clear Left
	ICHR	(70)061		RGEN	MOVE	I	-	-	-	-	H	-	-	Interchange Halves and Clear Right
	ICL	141140		GEN	MOVE	S R V	H	H	H	H	H	-	-	Interchange and Clear Left
	ICR	141240		GEN	MOVE	S R V	H	H	H	H	H	-	-	Interchange and Clear Right
	IH	51	YN	MRGR	MOVE	I	-	-	-	-	H	-	-	Interchange Memory and Register-Halfword
	IH1	(70)126		RGEN	INT	I	-	-	-	-	H	2	1	Increment by One
	IH2	(70)127		RGEN	INT	I	-	-	-	-	H	2	1	Increment by Two
	ILE	141414		GEN	MOVE	V	-	-	-	H	H	-	-	Interchange L and E
	IM	40	NN	MRNR	INT	I	-	-	-	-	H	-	1	Increment Fullword
	IMA	13		MR	MOVE	S R V	H	H	H	H	H	-	-	Interchange Memory and A
	IMH	0	NN	MRNR	INT	I	-	-	-	-	H	-	1	Increment Halfword
R	INA	54		PIO	IO	S R	H	H	H	H	H	-	-	Input to A
R	INBC	001215		AP	PRCEX	V I	-	-	-	H	H	6	5	Interrupt Notify
R	INBN	001215		AP	PRCEX	V I	-	-	-	H	H	6	5	Interrupt Notify
R	INEC	001216		AP	PRCEX	V I	-	-	-	H	H	6	5	Interrupt Notify
R	INEN	001214		AP	PRCEX	V I	-	-	-	H	H	6	5	Interrupt Notify
R	INH	001001		GEN	IO	S R V I	H	H	H	H	H	-	-	Inhibit Interrupts
	INK	000043		GEN	KEYS	S R	H	H	H	H	H	-	-	Input Keys
	INK	(70)070		RGEN	KEYS	I	-	-	-	-	H	-	-	Save Keys
	INT	140554		GEN	FLPT	V	-	O	O	H	H	3	5	Fix as Integer
	INT	(70)103,113		RGEN	FLPT	I	-	-	-	-	H	-	-	Convert Floating to Integer
	INTA	140531		GEN	FLPT	V	-	-	-	H	H	3	5	Convert Floating to Integer
	INTH	(70)101,111		RGEN	FLPT	I	-	-	-	-	H	-	-	Convert Floating to Halfword Integer
	INTL	140533		GEN	FLPT	V	-	-	-	H	H	3	5	Convert Floating to Integer Long
	IR1	(70)122		RGEN	INT	I	-	-	-	-	H	2	1	Increment by One
	IR2	(70)123		RGEN	INT	I	-	-	-	-	H	2	1	Increment by Two
	IRB	(70)064		RGEN	MOVE	I	-	-	-	-	H	-	-	Interchange Bytes
	IRH	(70)057		RGEN	MOVE	I	-	-	-	-	H	-	-	Interchange Halves

R	MNEM	OP CODE	RI	FORM	FUNC	MODE	1	2	3	4	5	C	CC	DESCRIPTION
	IRS	12		MR	SKIP	S R V	H	H	H	H	H	-	-	Increment Memory Replace and Skip
R	IRTC	000603		GEN	IO	V I	-	-	-	H	H	7	6	Interrupt Return
R	IRTN	000601		GEN	IO	V I	-	-	-	H	H	7	6	Interrupt Return
	IRX	140114		GEN	SKIP	S R V	H	H	H	H	H	-	-	Increment and Replace Index
R	ITLB	000615		GEN	MCTL	V I	-	-	-	H	H	-	-	Invalidate STLB entry
	JDX	15 02		MR	PCTLJ	R	U	U	H	H	H	-	-	Jump and Decrement Index
	JEQ	02 03		MR	PCTLJ	R	U	U	H	H	H	-	-	Jump if = 0
	JGE	07 03		MR	PCTLJ	R	U	U	H	H	H	-	-	Jump or $\geq$ 0
	JGT	05 03		MR	PCTLJ	R	U	U	H	H	H	-	-	Jump if $>$ 0
	JIX	15 03		MR	PCTLJ	R	U	U	H	H	H	-	-	Jump and Increment Index
	JLE	04 03		MR	PCTLJ	R	U	U	H	H	H	-	-	Jump if $\leq$ 0
	JLT	06 03		MR	PCTLJ	R	U	U	H	H	H	-	-	Jump if $<$ 0
	JMP	51	NN	MRNR	PCTLJ	I	-	-	-	-	H	-	-	Jump
	JMP	01		MR	PCTLJ	S R V	H	H	H	H	H	-	-	Jump
	JNE	03 03		MR	PCTLJ	R	U	U	H	H	H	-	-	Jump if .NE. 0
	JSR	73	NN	MRGR	PCTLJ	I	-	-	-	-	H	-	-	Jump to Subroutine
	JST	10		MR	PCTLJ	S R	H	H	H	H	H	-	-	Jump and Store
	JSX	35 03		MR	PCTLJ	R V	H	H	H	H	H	-	-	Jump and Store Return in Index
	JSXB	61	NN	MRNR	PCTLJ	I	-	-	-	-	H	-	-	Jump and Set XB
	JSXB	14 02		MR	PCTLJ	V	-	-	-	H	H	-	-	Jump and Set XB
	JSY	14		MR	PCTLJ	V	-	-	-	H	H	-	-	Jump and Store Y
	L	01	YY	MRGR	MOVE	I	-	-	-	-	H	-	-	Load
	LCEQ	141503		GEN	LTSTS	V	-	-	-	H	H	-	-	Test CC Equal to 0 and Set A
	LCEQ	(70)153		RGEN	LTSTS	I	-	-	-	-	H	-	-	Test CC = 0 and Set R
	LCGE	141504		GEN	LTSTS	V	-	-	-	H	H	-	-	Test CC $\leq$ 0 and Set A
	LCGE	(70)154		RGEN	LTSTS	I	-	-	-	-	H	-	-	Test CC $\geq$ 0 and Set R
	LCGT	141505		GEN	LTSTS	V	-	-	-	H	H	-	-	Test CC $>$ 0 and Set A
	LCGT	(70)155		RGEN	LTSTS	I	-	-	-	-	H	-	-	Test CC > 0 and Set R
	LCLE	141501		GEN	LTSTS	V	-	-	-	H	H	-	-	Test CC Less than 0 or Equal to A
	LCLE	(70)151		RGEN	LTSTS	I	-	-	-	-	H	-	-	Test CC $\leq$ 0 and Set R
	LCLT	141500		GEN	LTSTS	V	-	-	-	H	H	-	-	Test CC $<$ 0 and Set A
	LCLT	(70)150		RGEN	LTSTS	I	-	-	-	-	H	-	-	Test CC < 0 and Set R
	LCNE	141502		GEN	LTSTS	V	-	-	-	H	H	-	-	Test CC .NE. 0 and Set A
	LCNE	(70)152		RGEN	LTSTS	I	-	-	-	-	H	-	-	Test CC .NE. 0 and Set R
	LDA	02		MR	MOVE	S R V	H	H	H	H	H	-	-	Load A
	LDAR	44	NN	MRGR	MOVE	I	-	-	-	-	H	-	-	Load Addressed Register
	LDC	(70)162,172		RGEN	CHAR	I	-	-	-	-	H	-	7	Load Character
	LDC 0	001302		CHAR	CHAR	V	-	-	-	H	H	-	7	Load Character
	LDC 1	001312		CHAR	CHAR	V	-	-	-	H	H	-	7	Load Character
	LDL	02 03		MR	MOVE	V	-	-	-	H	H	-	-	Load Long
P	LDLR	05 01		MR	MOVE	V	-	-	-	H	H	-	-	Load From Addressed Register
	LDX	35		MR	MOVE	S R V	H	H	H	H	H	-	-	Load Index
	LDY	35 01		MR	MOVE	V	-	-	-	H	H	-	-	Load
	LEQ	140413		GEN	LTSTS	S R V	H	H	H	H	H	-	4	Test A = 0; Set A
	LEQ	(70)003		RGEN	LTSTS	I	-	-	-	-	H	-	4	Test R = 0 and Set R
	LF	140416		GEN	LTSTS	S R V	H	H	H	H	H	-	5	Logic set A False
	LF	(70)016		RGEN	LTSTS	I	-	-	-	-	H	-	4	Logic set R False
	LFEQ	141113		GEN	LTSTS	V	-	-	-	H	H	-	4	Test F = 0; Set A
	LFEQ	(70)023,033		RGEN	LTSTS	I	-	-	-	-	H	-	4	Test F = 0; Set R
	LFGE	141114		GEN	LTSTS	V	-	-	-	H	H	-	4	Test F $\geq$ 0; Set A
	LFGE	(70)024,034		RGEN	LTSTS	I	-	-	-	-	H	-	4	Test F $\geq$ 0; Set A
	LFGT	141115		GEN	LTSTS	V	-	-	-	H	H	-	4	Test F $>$ 0; Set A
	LFGT	(70)025,035		RGEN	LTSTS	I	-	-	-	-	H	-	4	Test F > 0; Set R
	LFLE	141111		GEN	LTSTS	V	-	-	-	H	H	-	4	Test F $\leq$ 0; Set A
	LFLE	(70)021,031		RGEN	LTSTS	I	-	-	-	-	H	-	4	Test F $\leq$ 0; Set R
	LFLI 0	001303		BRAN	FIELD	V I	-	-	-	H	H	-	-	Load Field Length Register 0
	LFLI 1	001313		BRAN	FIELD	V I	-	-	-	H	H	-	-	Load Field Length

R	MNEM	OP CODE	RI	FORM	FUNC	MODE	1	2	3	4	5	C	CC	DESCRIPTION
														Register 1
	LFLT	141110		GEN	LTSTS	V	-	-	-	H	H	-	4	Test F < 0; Set A
	LFLT	(70)020,030		RGEN	LTSTS	I	-	-	-	-	H	-	4	Test F < 0; Set R
	LFNE	141112		GEN	LTSTS	V	-	-	-	H	H	-	4	Test F .NE. 0; Set A
	LFNE	(70)022,032		RGEN	LTSTS	I	-	-	-	-	H	-	4	Test F .NE. 0; Set R
	LGE	140414		GEN	LTSTS S R V		H	H	H	H	H	-	4	Test A $\geq$ 0; Set A
	LGE	(70)004		RGEN	LTSTS	I	-	-	-	-	H	-	4	Test R $\geq$ 0; Set R
	LGT	140415		GEN	LTSTS S R V		H	H	H	H	H	-	4	Test A > 0; Set A
	LGT	(70)005		RGEN	LTSTS	I	-	-	-	-	H	-	4	Test R > 0; Set R
	LH	11	YY	MRGR	MOVE	I	-	-	-	-	H	-	-	Load Halfword
	LHEQ	(70)013		RGEN	LTSTS	I	-	-	-	-	H	-	4	Test RH = 0; Set RH
	LHGE	(70)004		RGEN	LTSTS	I	-	-	-	-	H	-	4	Test RH $\geq$ 0; Set RH
	LHGT	(70)015		RGEN	LTSTS	I	-	-	-	-	H	-	4	Test RH > 0; Set R
	LHL1	04	YN	MRGR	MOVE	I	-	-	-	-	H	-	-	Load Halfword Left Shifted by 1
	LHL2	14	YN	MRGR	MOVE	I	-	-	-	-	H	-	-	Load Halfword Left Shifted by 2
	LHLE	(70)011		RGEN	LTSTS	I	-	-	-	-	H	-	4	Test RH $\leq$ 0; Set RH
	LHLT	(70)000		RGEN	LTSTS	I	-	-	-	-	H	-	4	Test RH < 0; Set RH
	LHNE	(70)012		RGEN	LTSTS	I	-	-	-	-	H	-	4	Test RH .NE. 0; Set RH
	LLE	140411		GEN	LTSTS S R V		H	H	H	H	H	-	4	Test A $\leq$ 0; Set A
	LLE	(70)001		RGEN	LTSTS	I	-	-	-	-	H	-	4	Test R $\leq$ 0; Set R
	LLEQ	141513		GEN	LTSTS	V	-	-	-	H	H	-	4	Test L = 0; Set A
	LLGE	140414		GEN	LTSTS	V	-	-	-	H	H	-	4	Test L $\geq$ 0; Set A
	LLGT	141515		GEN	LTSTS	V	-	-	-	H	H	-	4	Test L > 0; Set A
	LLL	0410XX		SHFT	SHIFT S R V		H	H	H	H	H	4	5	Long Left Logical
	LLLE	141511		GEN	LTSTS	V	-	-	-	H	H	-	4	Test L $\leq$ 0; Set A
	LLLT	140410		GEN	LTSTS	V	-	-	-	H	H	-	4	Test L < 0; Set A
	LLNE	141512		GEN	LTSTS	V	-	-	-	H	H	-	4	Test L .NE. 0; Set A
	LLR	0412XX		SHFT	SHIFT S R V		H	H	H	H	H	4	5	Long left Rotate
	LLS	0411XX		SHFT	SHIFT S R V		H	H	H	H	H	4	5	Long Left Shift
	LLT	140410		GEN	LTSTS S R V		H	H	H	H	H	-	4	Test A < 0; Set A
	LLT	(70)000		RGEN	LTSTS	I	-	-	-	-	H	-	4	Test R < 0; Set R
R	LMCM	000501		GEN	INTGY S R V	I	-	H	H	H	H	-	-	Leave
	LNE	140412		GEN	LTSTS S R V		H	H	H	H	H	-	4	Test A .NE. 0; Set A
	LNE	(70)002		RGEN	LTSTS	I	-	-	-	-	H	-	4	Test R .NE. 0; Set R
R	LPID	000617		GEN	MCTL	V	I	-	-	-	H	H	-	-	Load Process ID
M	LPMJ	000215		MR	MCTL S R			-	-	H	-	-	-	-	Leave Paging Mode and Jump
M	LPMX	000235		MR	MCTL S R			-	-	H	-	-	-	-	Leave Paging Mode and Jump to XCS
R	LPSW	000711		AP	MCTL	V	I	-	-	-	H	H	7	6	Load Program Status Word
	LRL	0400XX		SHFT	SHIFT S R V		H	H	H	H	H	4	5	Long Right Logical Shift
	LRR	0402XX		SHFT	SHIFT S R V		H	H	H	H	H	4	5	Long Right Rotate
	LRS	0401XX		SHFT	SHIFT S R V		H	H	H	H	H	4	5	Long Right Shift
	LT	140417		GEN	LTSTS S R V		H	H	H	H	H	-	5	Set A = 1
	LT	(70)017		RGEN	LTSTS	I	-	-	-	-	H	-	4	Set R = 1
R	LWCS	001710		GEN	MCTL	V		-	-	-	H	H	-	-	Load Writable Control Store
	M	42	YY	MRGR	INT	I	-	-	-	-	H	3	1	Multiply Fullword
R	MDII	001305		GEN	INTGY	V	I	-	-	-	H	H	-	-	Inhibit Interleaved
R	MDIW	001324		GEN	INTGY	V	I	-	-	-	H	H	-	-	Write Interleaved
R	MDRS	001306		GEN	INTGY	V	I	-	-	-	H	H	-	-	Read Syndrome Bits
R	MDWC	001307		GEN	INTGY	V	I	-	-	-	H	H	-	-	Load Write Control Register
	MH	52	YY	MRGR	INT	I	-	-	-	-	H	3	1	Multiply Halfword
	MIA	64	NN	MRGR	MCTL	I	-	-	-	-	H	-	-	Microcode Entrance
M	MIA	12 01		MR	MCTL	V		-	-	-	H	H	-	-	Microcode Entrance
	MIB	74	NN	MRGR	MCTL	I	-	-	-	-	H	-	-	Microcode Entrance
M	MIB	13 01		MR	MCTL	V		-	-	-	H	H	-	-	Microcode Entrance
	MPL	16 03		MR	INT	V		-	-	-	H	H	-	1	Multiply Long
	MPY	16		MR	INT	V		-	-	-	H	H	-	1	Multiply
	MPY	16		MR	INT S R			O	O	H	H	H	3	1	Multiply
	N	03	YY	MRGR	LOGIC	I	-	-	-	-	H	-	-	AND

R	MNEM	OP CODE	RI	FORM	FUNC	MODE			1	2	3	4	5	C	CC	DESCRIPTION
R	NFYB	001211		AP	PRCEX		V	I	-	-	-	H	H	6	5	Notify
R	NFYE	001210		AP	PRCEX		V	I	-	-	-	H	H	6	5	Notify
	NH	13	YY	MRGR	LOGIC			I	-	-	-	-	H	-	-	AND Halfword
	NOP	000001		GEN	MCTL	S	R V		H	H	H	H	H	-	-	No Operation
	NRM	000101		GEN	INT	S	R		H	H	H	H	H	-	-	Normalize
	O	23	YY	MRGR	LOGIC			I	-	-	-	-	H	-	-	OR
R	OCP	14		PIO	IO	S	R		H	H	H	H	H	-	-	Output Control Pulse
	OH	33	YY	MRGR	LOGIC			I	-	-	-	-	H	-	-	OR Halfword
	ORA	03 02		MR	LOGIC		V		-	-	-	H	H	-	-	Inclusive OR
R	OTA	74		PIO	IO	S	R		H	H	H	H	H	-	-	Output from A
	OTK	000405		GEN	KEYS	S	R		H	H	H	H	H	7	6	Restore Keys
	OTK	(70)071		RGEN	KEYS			I	-	-	-	-	H	7	S	Restore Keys
	PCL	41	NN	MRNR	PCTLJ			I	-	-	-	-	H	-	-	Procedure Call
	PCL	10 02		MR	PCTLJ		V		-	-	-	H	H	7	6	Procedure Call
	PID	000211		GEN	INT	S	R		O	O	H	H	H	-	-	Position for Integer Divide
	PID	(70)052		RGEN	INT			I	-	-	-	-	H	-	-	Position for Integer Divide
	PIDA	000115		GEN	INT		V		-	-	-	H	H	-	-	Position for Integer Divide
	PIDH	(70)053		RGEN	INT			I	-	-	-	-	H	-	-	Position for Integer Divide
	PIDL	000305		GEN	INT		V		-	-	-	H	H	-	-	Position Long for Integer Divide
	PIM	000205		GEN	INT	S	R		O	O	H	H	H	-	-	Position After Multiply
	PIM	(70)050		RGEN	INT			I	-	-	-	-	H	2	1	Position After Multiply
	PIMA	000015		GEN	INT		V		-	-	-	H	H	3	5	Position After Multiply
	PIMH	(70)051		RGEN	INT			I	-	-	-	-	H	2	1	Position After Multiply
	PIML	000301		GEN	INT		V		-	-	-	H	H	3	5	Position After Multiply Long
	PRTN	000611		GEN	PCTLJ		V	I	-	-	-	H	H	7	6	Procedure Return
	RBQ	141715		AP	QUEUE		V		-	-	-	H	H	-	6	Remove From Bottom of Queue
	RBQ	(70)133		AP	QUEUE			I	-	-	-	-	H	-	7	Remove From Bottom of Queue
	RCB	140200		GEN	KEYS	S	R V		H	H	H	H	H	5	-	Clear C-Bit (Reset)
R	RMC	000021		GEN	INTGY	S	R V	I	-	H	H	H	H	-	-	Clear Machine Check
	ROT	24	NN	MRGR	SHIFT			I	-	-	-	-	H	4	-	Rotate
	RRST	000717		AP	MCTL		V	I	-	-	-	H	H	-	-	Register Restore
	RSAV	000715		AP	MCTL		V	I	-	-	-	H	H	-	-	Register Save
	RTN	000105		GEN	PCTLJ	S	R		H	H	H	H	H	-	-	Return
	RTQ	141714		AP	QUEUE		V		-	-	-	H	H	-	6	Remove From Top of Queue
	RTQ	(70)132		AP	QUEUE			I	-	-	-	-	H	-	7	Remove From Top of Queue
	S	22	YY	MRGR	INT			I	-	-	-	-	H	2	1	Subtract
	S1A	140110		GEN	INT	S	R V		H	H	H	H	H	2	1	Subtract One from A
	S2A	140310		GEN	INT	S	R V		H	H	H	H	H	2	1	Subtract Two from A
	SAR	10026X		GEN	SKIP	S	R V		H	H	H	H	H	-	-	Skip on A Bit Clear
	SAS	101260		GEN	SKIP	S	R V		H	H	H	H	H	-	-	Skip on A Bit Set
	SBL	07 03		MR	INT		V		-	-	-	H	H	2	1	Subtract Long
	SCA	000041		GEN	INT	S	R		H	H	H	H	H	-	-	Load Shift Count into A
	SCB	140600		GEN	KEYS	S	R V		H	H	H	H	H	5	-	Set C-Bit in Keys
	SGL	000005		GEN	INT	S	R		H	H	H	H	H	-	-	Single Precision
	SGT	100220		GEN	SKIP	S	R V		H	H	H	H	H	-	-	Skip if A Greater Than Zero
	SH	32	YY	MRGR	INT			I	-	-	-	-	H	2	1	Subtract Halfword
	SHA	15	NN	MRGR	SHIFT			I	-	-	-	-	H	4	-	Shift Arithmetic
	SHL	05	NN	MRGR	SHIFT			I	-	-	-	-	H	4	-	Shift Logical
	SHL1	(70)076		RGEN	SHIFT			I	-	-	-	-	H	4	1	Shift H Left One
	SHL2	(70)077		RGEN	SHIFT			I	-	-	-	-	H	4	1	Shift H Left Two
	SHR1	(70)120		RGEN	SHIFT			I	-	-	-	-	H	4	1	Shift H Right One
	SHR2	(70)121		RGEN	SHIFT			I	-	-	-	-	H	4	1	Shift H Right Two
	SKP	100000		GEN	SKIP	S	R V		H	H	H	H	H	-	-	Skip
R	SKS	34		PIO	IO	S	R		H	H	H	H	H	-	-	Skip if Satisfied
	SL1	(70)072		RGEN	SHIFT			I	-	-	-	-	H	4	1	Shift R Left One

R	MNEM	OP CODE	RI	FORM	FUNC	MODE		1	2	3	4	5	C	CC	DESCRIPTION
	SL2	(70)073		RGEN	SHIFT		I	-	-	-	-	H	4	1	Shift R Left Two
	SLE	101220		GEN	SKIP	S R V		H	H	H	H	H	-	-	Skip if A Less Than or Equal to Zero
	SLN	101100		GEN	SKIP	S R V		H	H	H	H	H	-	-	Skip if LSB (A(16)=1) Nonzero
	SLZ	100100		GEN	SKIP	S R V		H	H	H	H	H	-	-	Skip if LSB (A(16)=0) Zero
	SMCR	100200		GEN	INTGY	S R V		-	H	H	H	H	-	-	Skip on Machine Check Clear
	SMCS	101200		GEN	INTGY	S R V		-	H	H	H	H	-	-	Skip on Machine Check Set
	SMI	101400		GEN	SKIP	S R V		H	H	H	H	H	-	-	Skip if A Minus
R	SMK	170020		PIO	IO	S R		H	H	H	H	H	-	-	Send Mask
R	SNR	10024X		GEN	SKIP	S R V		H	H	H	H	H	-	-	Skip on Sense Switch Clear
R	SNS	101240		GEN	SKIP	S R V		H	H	H	H	H	-	-	Skip on Sense Switch Set
	SNZ	101040		GEN	SKIP	S R V		H	H	H	H	H	-	-	Skip if A Non-Zero
	SPL	100400		GEN	SKIP	S R V		H	H	H	H	H	-	-	Skip if A Plus
R	SR1	100020		GEN	SKIP	S R V		H	H	H	H	H	-	-	Skip if Sense Switch 1 Clear
	SR1	(70)074		RGEN	SHIFT		I	-	-	-	-	H	4	1	Shift R Right One
R	SR2	100010		GEN	SKIP	S R V		H	H	H	H	H	-	-	Skip if Sense Switch 2 Clear
	SR2	(70)075		RGEN	SHIFT		I	-	-	-	-	H	4	1	Shift R Right Two
R	SR3	100004		GEN	SKIP	S R V		H	H	H	H	H	-	-	Skip if Sense Switch 3 Clear
R	SR4	100002		GEN	SKIP	S R V		H	H	H	H	H	-	-	Skip if Sense Switch 4 Clear
	SRC	100001		GEN	SKIP	S R V		H	H	H	H	H	-	-	Skip if C-Bit is Clear
R	SS1	101020		GEN	SKIP	S R V		H	H	H	H	H	-	-	Skip if Sense Switch 1 Clear
R	SS2	101010		GEN	SKIP	S R V		H	H	H	H	H	-	-	Skip if Sense Switch 2 Clear
R	SS3	101004		GEN	SKIP	S R V		H	H	H	H	H	-	-	Skip if Sense Switch 3 Clear
R	SS4	101002		GEN	SKIP	S R V		H	H	H	H	H	-	-	Skip if Sense Switch 4 Clear
	SSC	101001		GEN	SKIP	S R V		H	H	H	H	H	-	-	Skip if C-Bit is Set
	SSM	140500		GEN	INT	S R V		H	H	H	H	H	-	-	Set Sign Minus
	SSM	(70)042		RGEN	INT		I	-	-	-	-	H	-	-	Set Sign Minus
	SSP	140100		GEN	INT	S R V		H	H	H	H	H	-	-	Set Sign Plus
	SSP	(70)043		RGEN	INT		I	-	-	-	-	H	-	-	Set Sign Plus
R	SSR	100036		GEN	SKIP	S R V		H	H	H	H	H	-	-	Skip if Any Sense Switch is Clear
R	SSS	101036		GEN	SKIP	S R V		H	H	H	H	H	-	-	Skip if Any Sense Switch is Set
	ST	21	NN	MRGR	MOVE		I	-	-	-	-	H	-	-	Store
	STA	04		MR	MOVE	S R V		H	H	H	H	H	-	-	Store A
	STAC	001200		AP	MOVE	V		-	-	-	H	H	-	7	Store A Conditionally
	STAR	54	NN	MRGR	MOVE		I	-	-	-	-	H	-	-	Store Addressed Register
	STC	(70)166,176		RGEN	CHAR		I	-	-	-	-	H	-	7	Store Character
	STC 0	001322		CHAR	CHAR	V		-	-	-	H	H	-	7	Store Character
	STC 1	001332		CHAR	CHAR	V		-	-	-	H	H	-	7	Store Character
	STCD	(70)137		RGEN	MOVE		I	-	-	-	-	H	-	7	Store Conditional Fullword
	STCH	(70)136		RGEN	MOVE		I	-	-	-	-	H	-	7	Store Conditional Halfword
	STEX	001315		GEN	PCTLJ	V		-	-	-	H	H	6	5	Stack Extend
	STEX	(70)027		RGEN	PCTLJ		I	-	-	-	-	H	-	7	Stack Extend
	STFA 0	001320		AP	FIELD	V	I	-	-	-	H	H	-	-	Store Field Address Register
	STFA 1	001330		AP	FIELD	V	I	-	-	-	H	H	-	-	Store Field Address Register
	STH	31	NN	MRGR	MOVE		I	-	-	-	-	H	-	-	Store Halfword
	STL	04 03		MR	MOVE	V		-	-	-	H	H	-	-	Store Long

R	MNEM	OP CODE	RI	FORM	FUNC	MODE	1	2	3	4	5	C	CC	DESCRIPTION
	STLC	001204		AP	MOVE	V	-	-	-	H	H	-	7	Store L Conditionally
P	STLR	03 01		MR	MOVE	V	-	-	-	H	H	-	-	Store L into Addressed Register
	STX	15		MR	MOVE	S R V	H	H	H	H	H	-	-	Store X
	STY	35 02		MR	MOVE	V	-	-	-	H	H	-	-	Store Y
	SUB	07		MR	INT	S R V	H	H	H	H	H	2	1	Subtract
	SVC	000505		GEN	PCTLJ	S R V I	H	H	H	H	H	-	-	Supervisor Call
	SZE	100040		GEN	SKIP	S R V	H	H	H	H	H	-	-	Skip if A Zero
	TAB	140314		GEN	MOVE	V	-	-	-	H	H	-	-	Transfer A to B
	TAK	001015		GEN	KEYS	V	-	-	-	H	H	7	6	Move A to Keys
	TAX	140504		GEN	MOVE	V	-	-	-	H	H	-	-	Transfer A to X
	TAY	140505		GEN	MOVE	V	-	-	-	H	H	-	-	Transfer A to Y
	TBA	140604		GEN	MOVE	V	-	-	-	H	H	-	-	Transfer B to A
	TC	(70)046		RGEN	INT	I	-	-	-	-	H	3	1	Two's Complement R
	TCA	140407		GEN	INT	S R V	H	H	H	H	H	2	1	Two's Complement A
	TCH	(70)047		RGEN	INT	I	-	-	-	-	H	3	1	Two's Complement H
	TCL	141210		GEN	INT	V	-	-	-	H	H	2	1	Two's Complement Long
	TFLL 0	001323		GEN	FIELD	V	-	-	-	H	H	-	-	Transfer Field Length Register to L
	TFLL 1	001333		GEN	FIELD	V	-	-	-	H	H	-	-	Transfer Field Length Register to L
	TFLR	(70)163,173		RGEN	FIELD	I	-	-	-	-	H	-	7	Move Field Length
	TKA	001005		GEN	KEYS	V	-	-	-	H	H	-	-	Move Keys to A
	TLFL	001321		GEN	FIELD	V	-	-	-	H	H	-	-	Transfer L to Field Length Register
	TLFL	001331		GEN	FIELD	V	-	-	-	H	H	-	-	Transfer L to Field Length Register
	TM	44	NN	MRNR	MCTL	I	-	-	-	-	H	-	1	Test Memory
	TRFL	(70)165,175		RGEN	FIELD	I	-	-	-	-	H	-	7	Update Field Length
	TSTQ	141757		AP	QUEUE	V	-	-	-	H	H	-	6	Test Queue
	TSTQ	(70)104		AP	QUEUE	I	-	-	-	-	H	-	7	Test Queue
	TXA	141034		GEN	MOVE	V	-	-	-	H	H	-	-	Transfer X to A
	TYA	141124		GEN	MOVE	V	-	-	-	H	H	-	-	Transfer Y to A
R	VIRY	000311		GEN	INTGY	S R V I	-	O	H	H	H	5	6	Verify
R	WAIT	000315		AP	PRCEX	V I	-	-	-	H	H	-	-	Wait
W	WCS	0016XX		GEN	MCTL	R V	-	-	-	O	O	-	-	Writeable Control Store
	X	43	YY	MRGR	LOGIC	I	-	-	-	-	H	-	-	Exclusive OR
	XAD	001100		DECI	DECI	V I	-	-	-	U	H	X	X	Decimal Add
	XBTD	001145		DECI	DECI	V I	-	-	-	U	H	X	X	Binary to Decimal Conversion
	XCA	140104		GEN	MOVE	S R V	H	H	H	H	H	-	-	Exchange and Clear A
	XCB	140204		GEN	MOVE	S R V	H	H	H	H	H	-	-	Exchange and Clear B
	XCM	001102		DECI	DECI	V I	-	-	-	U	H	X	X	Decimal Compare
	XDTB	001146		DECI	DECI	V I	-	-	-	U	H	X	X	Decimal to Binary Conversion
	XDV	001107		DECI	DECI	V I	-	-	-	U	H	X	X	Decimal Divide
	XEC	01 02		MR	PCTLJ	R V	-	-	H	H	H	-	-	Execute
	XED	001112		DECI	DECI	V I	-	-	-	-	H	X	X	Numeric Edit
	XH	53	YY	MRGR	LOGIC	I	-	-	-	-	H	-	-	Exclusive OR Halfword
	XMP	001104		DECI	DECI	V I	-	-	-	U	H	X	X	Decimal Multiply
	XMV	001101		DECI	DECI	V I	-	-	-	U	H	X	X	Decimal Move
	ZCM	001117		CHAR	CHAR	V I	-	-	-	U	H	X	X	Compare Character Field
	ZED	001111		CHAR	CHAR	V I	-	-	-	-	H	X	X	Character Edit
	ZFIL	011116		CHAR	CHAR	V I	-	-	-	U	H	X	X	Fill Character Field
	ZM	43	NN	MRNR	CLEAR	I	-	-	-	-	H	-	-	Clear Fullword
	ZMH	53	NN	MRNR	CLEAR	I	-	-	-	-	H	-	-	Clear Halfword
	ZMV	001114		CHAR	CHAR	V I	-	-	-	U	H	X	X	Move Character Field
	ZMVD	001115		CHAR	CHAR	V I	-	-	-	U	H	X	X	Move Equal Length Fields
	ZTRN	001110		CHAR	CHAR	V I	-	-	-	U	H	X	X	Translate Character Fields