Chapter 42

VAX-11/780—A Virtual Address Extension to the DEC PDP-11 Family[1]

W. D. Strecker

Introduction

Large Virtual Address Space Minicomputers

Perhaps the most useful definition of a minicomputer system is based on price: depending on one's perspective such systems are typically found in the $20K to $200K range. The twin forces of market pull—as customers build increasingly complex systems on minicomputers—and technology push—as the semiconductor industry provides increasingly lower cost logic and memory elements—have induced minicomputer manufacturers to produce systems of considerable performance and memory capacity. Such systems are typified by the DEC PDP-11/70. From an architectural point of view, the characteristic which most distinguishes many of these systems from larger mainframe computers is the size of the virtual address space: the immediately available address space seen by an individual process. For many purposes the 65K byte virtual address space typically provided on minicomputers (such as the PDP-11) has not been and probably will not continue to be a severe limitation. However, there are some applications whose programming is impractical in a 65K byte virtual address space, and perhaps most importantly, others whose programming is appreciably simplified by having a large virtual address space. Given the relative trends in hardware and software costs, the latter point alone will insure that large virtual address space minicomputers play an increasingly important role in minicomputer product offerings.

In principle, there is no great challenge in designing a large virtual address minicomputer system. For example, many of the large mainframe computers could serve as architectural models for such a system. The real challenge lies in two areas: compatibility—very tangible and important; and simplicity—intangible but nonetheless important.

The first area is preserving the customer's and the computer manufacturer's investment in existing systems. This investment exists at many levels: basic hardware (principally busses and peripherals); systems and applications software; files and data bases; and personnel familiar with the programming, use, and operation of the systems. For example, just recently a major computer manufacturer abandoned a major effort for new computer architectures in favor of evolving its current architectures [McLean, 1977].

The second intangible area is the preservation of those attributes (other than price) which make minicomputer systems attractive. These include approachability, understandability, and ease of use. Preservation of these attributes suggests that simply modelling an extended virtual address minicomputer after a large mainframe computer is not wholly appropriate. It also suggests that during architectural design, tradeoffs must be made between more than just performance, functionality, and cost. Performance or functionality features which are so complex that they appreciably compromise understanding or ease of use must be rejected as inappropriate for minicomputer systems.

VAX-11 Overview

VAX-11 is the Virtual Address eXtention of PDP-11 architecture [Bell et al., 1970; Bell and Strecker, 1976]. The most distinctive feature of VAX-11 is the extension of the virtual address from 16 bits as provided on the PDP-11 to 32 bits. With the 8-bit byte the basic addressable unit, the extension provides a virtual address space of about 4.3 gigabytes which, even given rapid improvement in memory technology, should be adequate far into the future.

Since maximal PDP-11 compatibility was a strong goal, early VAX-11 design efforts focused on literally extending the PDP-11: preserving the existing instruction formats and instruction set and fitting the virtual address extension around them. The objective here was to permit, to the extent possible, the running of existing programs in the extended virtual address environment. While realizing this objective was possible (there were three distinct designs), it was felt that the extended architecture designs were overly compromised in the areas of efficiency, functionality, and programming ease.

Consequently, it was decided to drop the constraint of the PDP-11 instruction format in designing the extended virtual address space or *native mode* of the VAX-11 architecture. However, in order to run existing PDP-11 programs, VAX-11 includes a PDP-11 *compatibility mode*. Compatibility mode provides the basic PDP-11 instruction set less only privileged instructions (such as HALT) and floating point instructions (which are optional on most PDP-11 processors and not required by most PDP-11 software).

In addition to compatibility mode, a number of other features to preserve PDP-11 investment have been provided in the VAX-11 architecture, the VAX-11 operating system VAX/VMS, and the VAX-11/780 implementation of the VAX-11 architecture. These features include:

1 The equivalent native mode data types and formats are identical to those on the PDP-11. Also, while extended, the VAX-11 native mode instruction set and addressing modes

[1]*AFIPS Proc. NCC*, 1978, pp. 967–980.

are very close to those on the PDP-11. As a consequence VAX-11 native mode assembly language programming is quite similar to PDP-11 assembly language programming.

2 The VAX-11/780 uses the same peripheral busses (Unibus and Massbus) as the PDP-11 and uses the same peripherals.

3 The VAX/VMS operating system is an evolution of the PDP-11 RSX-11M and IAS operating systems, offers a similar although extended set of system services, and uses the same command languages. Additionally, VAX/VMS supports most of the RSX-11M/IAS system service requests issued by programs executing in compatibility mode.

4 The VAX/VMS file system is the same as used on the RSX-11M/IAS operating systems permitting interchange of files and volumes. The file access methods as implemented by the RMS record manager are also the same.

5 VAX-11 high level language compilers accept the same source languages as the equivalent PDP-11 compilers and execution of compiled programs gives the same results.

The coverage of all these aspects of VAX-11 is well beyond the scope of any single paper. The remainder of this paper discusses the design of the VAX-11 native mode architecture and gives an overview of the VAX-11/780 system.

VAX-11 Native Architecture

Processor State

Like the PDP-11, VAX-11 is organized around a general register processor state. This organization was favored because access to operands stored in general registers is fast (since the registers are internal to the processor and register accesses do not need to pass through a memory management mechanism) and because only a small number of bits in an instruction are needed to designate a register. Perhaps most importantly, the registers are used (as on the PDP-11) in conjunction with a large set of addressing modes which permit unusually flexible operand addressing methods.

Some consideration was given to a pure stack based architecture. However it was rejected because real program data suggests the superiority of two or three operand instruction formats [Myers, 1977b]. Actually VAX-11 is quite stack oriented, and although it is not optimally encoded for the purpose, can easily be used as a pure stack architecture if desired.

VAX-11 has 16 32-bit general registers (denoted R0-R15) which are used for both fixed and floating point operands. This is in contrast to the PDP-11 which has eight 16-bit general registers and six 64-bit floating point registers. The merged set of fixed and floating registers were preferred because it simplifies programming and permits a more effective allocation of the registers.

Four of the registers are assigned special meaning in the VAX-11 architecture:

1 R15 is the *program counter* (PC) which contains the address of the next byte to be interpreted in the instruction stream.

2 R14 is the *stack pointer* (SP) which contains the address of the top of the processor defined stack used for procedure and interrupt linkage.

3 R13 is the *frame pointer* (FP). The VAX-11 procedure calling convention builds a data structure on the stack called a stack frame. FP contains the address of this structure.

4 R12 is the *argument pointer* (AP). The VAX-11 procedure calling convention uses a data structure called an argument list. AP contains the address of this structure.

The remaining element of the user visible processor state (additional processor state seen mainly by privileged procedures is discussed later) is the 16-bit *processor status word* (PSW). The PSW contains the N, Z, V, and C condition codes which indicate respectively whether a previous instruction had a negative result, a zero result, a result which overflowed, or a result which produced a carry (or borrow). Also in the PSW are the IV, DV, and FU bits which enable processor trapping on integer overflow, decimal overflow, and floating underflow conditions respectively. (The trapping on conditions of floating overflow and divide by zero for any data type are always enabled.)

Finally, the PSW contains the T bit which when set forces a trap at the end of each instruction. This trap is useful for program debugging and analysis purposes.

Data Types and Formats

The VAX-11 data types are a superset of the PDP-11 data types. Where the PDP-11 and VAX-11 have equivalent data types the formats (representation in memory) are identical. Data type and data format identity is one of the most compelling forms of compatibility. It permits free interchange of binary data between PDP-11 and VAX-11 programs. It facilitates source level compatibility between equivalent PDP-11 and VAX-11 languages. It also greatly facilitiates hardware implementation of and software support of the PDP-11 compatibility mode in the VAX-11 architecture.

The VAX-11 data types divide into five classes:

1 Integer data types are the 8-bit *byte*, the 16-bit *word*, the 32-bit *longword*, and the 64-bit *quadword*. Usually these data types are considered signed with negative values represented in two's complement form. However, for most purposes they can be interpreted as unsigned and the

VAX-11 instruction set provides support for this interpretation.

2 Floating data types are the 32-bit *floating* and the 64-bit *double* floating. These data types are binary normalized, have an 8-bit signed exponent, and have a 25- or 57-bit signed fraction with the redundant most significant fraction bit not represented.

3 The *variable bit field* data type is 0 to 32 bits located arbitrarily with respect to addressable byte boundaries. A bit field is specified by three operands: the address of a byte, the starting bit position P with respect to bit 0 of that byte, and the size S of the field. The VAX-11 instruction set provides for interpreting the field as signed or unsigned.

4 The *character string* data type is 0 to 65535 contiguous bytes. It is specified by two operands: the length and starting address of the string. Although the data type is named "character string," no special interpretation is placed on the values of the bytes in the character string.

5 The *decimal string* data types are 0 to 31 digits. They are specified by two operands: a length (in digits) and a starting address. The primary data type is *packed decimal* with two digits stored in each byte except that the byte containing the least significant digit contains a single digit and the sign. Two ASCII character decimal types are supported: *leading separate* sign and *trailing embedded* sign. The leading separate type is a "+," "−," or "<blank>" (equivalent to "+") ASCII character followed by 0 to 31 ASCII decimal digit characters. A trailing embedded sign decimal string is 0 to 31 bytes which are ASCII decimal digit characters except for the character containing least significant digit which is an arbitrary encoding of the digit and sign.

All of the data types except field may be stored on arbitrary byte boundaries—there are no alignment constraints. The field data type, of course, can start on an arbitrary bit boundary.

Attributes of and symbolic representations for most of the data types are given in Table 1 and Fig. 1.

Instruction Format and Address Modes

Most architectures provide a small number of relatively fixed instruction formats. Two problems often result. First, not all operands of an instruction have the same specification generality. For example, one operand must come from memory and another from a register; or one must come from the stack and another from memory. Second, only a limited number of operands can be accommodated: typically one or two. For instructions which inherently require more operands (such as field or string instructions), the additional operands are specified in ad hoc ways: small literal fields in instructions, specific registers or stack positions, or packed in fields of a single operand. Both these problems lead to increased programming complexity: they require superfluous move type instructions to get operands to places where they can be used and increase competition for potentially scarce resources such as registers.

To avoid these problems two criteria were used in the design of the VAX-11 instruction format: (1) all instructions should have the "natural" number of operands and (2) all operands should have the same generality in specification. These criteria led to a highly

Table 1 Data Types

Data type	*Size*	*Range (decimal)*	
Integer		*Signed*	*Unsigned*
Byte	8 bits	−128 to +127	0 to 255
Word	16 bits	−32768 to +32767	0 to 65535
Longword	32 bits	-2^{31} to $+2^{31}-1$	0 to $2^{32}-1$
Quadword	64 bits	-2^{63} to $+2^{63}-1$	0 to $2^{64}-1$
Floating point		$\pm 2.9 \times 10^{-3}$ *to* 1.7×10^{38}	
Floating	32 bits	approximately seven decimal digits precision	
Double Floating	64 bits	approximately sixteen decimal digits precision	
Packed decimal string	0 to 16 bytes (31 digits)	numeric, two digits per byte sign in low half of last byte	
Character string	0 to 65535 bytes	one character per byte	
Varaible-length bit field	0 to 32 bits	dependent on interpretation	

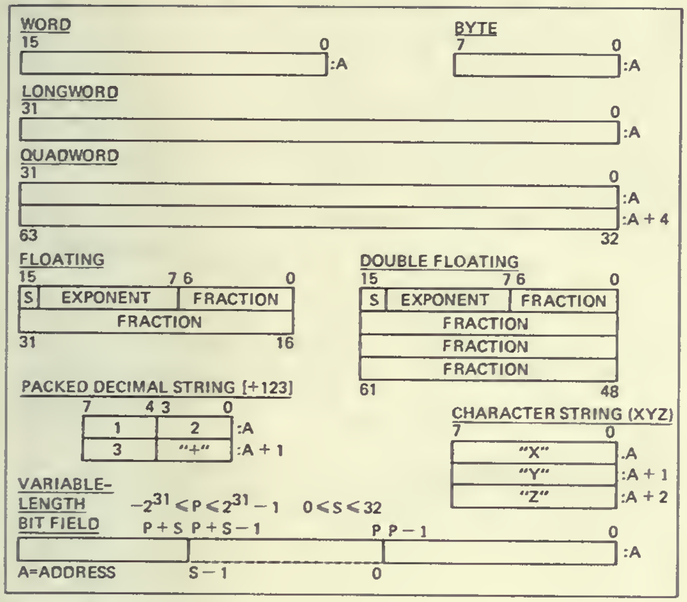

Fig. 1. Data formats.

variable instruction format. An instruction consists of a one or two[1] byte *opcode* followed by the specifications for n operands ($n \geq 0$) where n is an implicit property of the opcode. An operand specification is one to 10 bytes in length and consists of a one or two byte *operand specifier* followed by (as required) zero to eight bytes of *specifier extension*. The operand specifier includes the address mode and designation of any registers needed to locate the operand. A specifier extension consists of a displacement, an address, or immediate data.

The VAX-11 address modes are with one exception a superset of the PDP-11 address modes. The PDP-11 address mode autodecrement deferred was omitted from VAX-11 because it was rarely used.

Most operand specifiers are one byte long and contain two 4-bit fields: the high order field (bits 7:4) contains the address mode designator and the lower field (bits 3:0) contains a general register designator. The address modes include:

1 *Register mode* in which the designated register contains the operand.

2 *Register deferred mode* in which the designated register contains the address of the operand.

3 *Autodecrement mode* in which the contents of the designated register are first decremented by the size (in bytes) of the operand and then used as the address of the operand.

[1]No currently defined instructions use two byte opcodes.

4 *Autoincrement mode* in which the contents of the designated register are first used as the address of the operand and are then incremented by the size of the operand. Note that if the designated register is PC, the operand is located in the instruction stream. This use of autoincrement mode is called *immediate* mode. In immediate mode the one to eight bytes of data are the specifier extension.

Autoincrement mode can be used sequentially to process a vector in one direction and autodecrement mode used to process a vector in the opposite direction. Autoincrement, register deferred, and autodecrement modes can be applied to a single register to implement a stack data structure: autodecrement to "push," autoincrement to "pop," and register deferred to access the top of the stack.

5 *Autoincrement deferred mode* in which the contents of the designated register are used as the address of a longword in memory which contains the address of the operand. After this use, the contents of the register are incremented by four (the size in bytes of the longword address). Note that if PC is the designated register, the absolute address of the operand is located in the instruction stream. This use of autoincrement deferred mode is termed *absolute* mode. In absolute mode the 4-byte address is the specifier extension.

6 *Displacement mode* in which a displacement is added to the contents of the designated register to form the operand address. There are three displacement modes depending on whether a signed byte, word, or longword displacement is the specifier extension. These modes are termed byte, word, and longword displacement respectively. Note that if PC is the designated register, the operand is located relative to PC. For this use the modes are termed byte, word, and longword *relative* mode respectively.

7 *Displacement deferred mode* in which a displacement is added to the designated register to form the address of a longword containing the address of the operand. There are three displacement deferred modes depending on whether a signed byte, word, or longword displacement is the specifier extension. These modes are termed byte, word, and longword displacement respectively. Note that if PC is the designated register, the operand address is located relative to PC. For this use the modes are termed byte, word, and longword *relative deferred* mode respectively.

8 *Literal mode* in which the operand specifier itself contains a 6-bit literal which is the operand. For integer data types the literal encodes the values 0–63; for floating data types the literal includes three exponent and three fraction bits to give 64 common values.

9 *Index mode* which is not really a mode but rather a one byte prefix operator for any other mode which evaluates to a memory address (i.e., all modes except register and literal). The index mode prefix is cascaded with the operand specifier for that mode (called the base operand specifier) to form an aggregate two byte operand specifier. The base

operand specifier is used in the normal way to evaluate a base address. A copy of the contents of the register designated in the index prefix is multiplied by the size (in bytes) of the operand and added to the base address. The sum is the final operand address. There are three advantages to the VAX-11 form of indexing: (a) the index is scaled by the data size and thus the index register maintains a logical rather than a byte offset into an indexed data structure, (b) indexing can be applied to any of the address modes which generate memory addresses and this results in a comprehensive set of indexed addressing methods, and (c) the space required to specify indexing and the index register is paid only when indexing is used.

The VAX-11 assembler syntax for the address modes is given in Fig. 2. The bracketed ([]) notation is optional and the programmer rarely needs to be concerned with displacement sizes or whether to choose literal or immediate mode. The programmer writes the simple form and assembler chooses the address mode which produces the shortest instruction length.

In order to give a better feeling for the instruction format and assembler notation, several examples are given in Figs. 3 to 5. In Fig. 3 is an instruction which moves a word from an address which is 56 plus the contents of R5 to an address which is 270 plus the contents of R6. Note, that the displacement 56 is representable in a byte while the displacement 270 requires a word. The instruction occupies 6 bytes. In Fig. 4 is an instruction which adds 1 to a longword in R0 and stores the result at a memory address which is the sum of A and 4 times the contents of R. This instruction occupies 9 bytes. Finally, in Fig. 5 is a return from subroutine instruction. It has no explicit operands and occupies a single byte.

The only significant instance where there is non-general specification of operands is in the specification of targets for

Literal (Immediate)	{S↑ / I↑} *# constant*	
Register	R*n*	
Register Deferred	(R*n*)	Indexed [R*x*]
Autodecrement	− (R*n*)	Indexed [R*x*]
Autoincrement	(R*n*)+	Indexed [R*x*]
Autoincrement Deferred	@ (R*n*)+	Indexed [R*x*]
(Absolute)	@ *# address*	Indexed [R*x*]
Displacement	{B↑ / W↑ / L↑} *displacement* (R*n*) / *address*	Indexed [R*x*]
Displacement Deferred	@ {B↑ / W↑ / L↑} *displacement* (R*n*) / *address*	Indexed [R*x*]

n = 0 through 15
x = 0 through 14

Fig. 2. Assembler syntax.

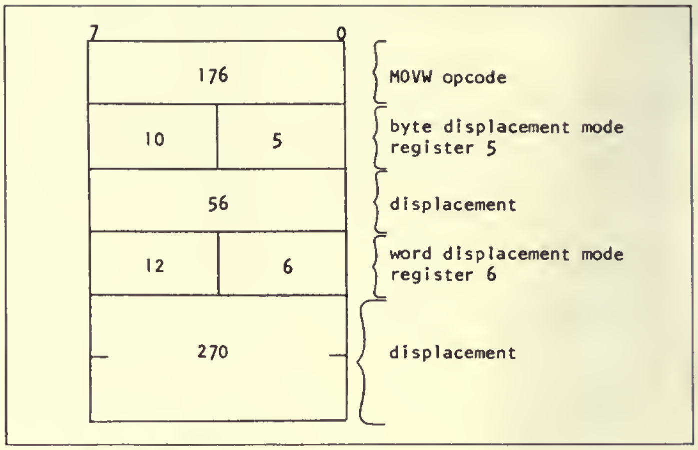
7
0
176
MOVW opcode
10
5
byte displacement mode register 5
56
displacement
12
6
word displacement mode register 6
270
displacement

Fig. 3. MOVW 56 (R5), 270 (R7).

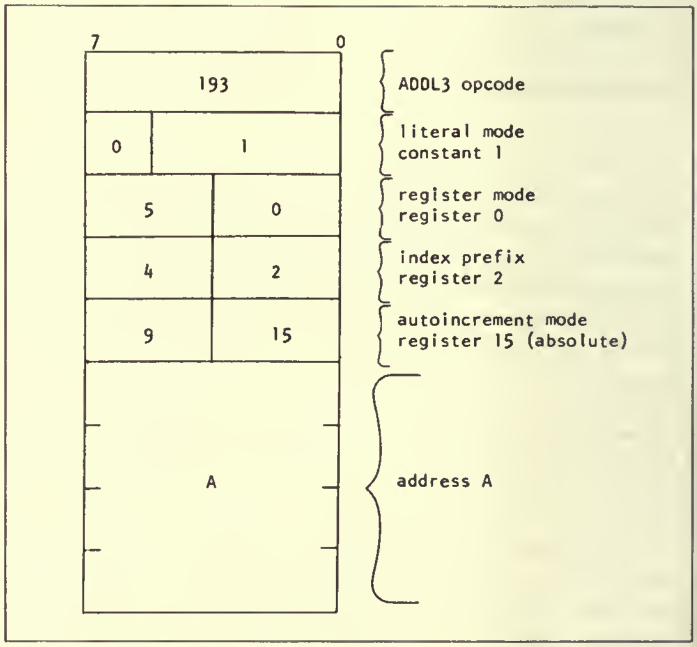

Fig. 4. ADDL3 #1, R0, @ #A [R2].

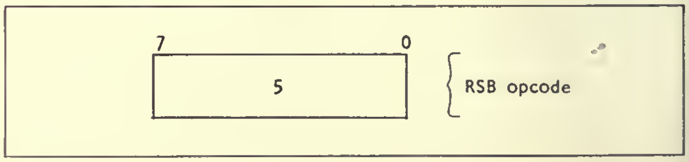

Fig. 5. RSB.

branch instructions. Since invariably the target of a branch instruction is a small displacement from the current PC, most branch instructions simply take a one byte PC relative displacement. This is exactly as if byte displacement mode were used with the PC used as the register, except that the operand specifier byte is not needed. Because of the pervasiveness of branch instructions in code, this one byte saving results in a non-trivial reduction in code size. An example of the branch instruction branch on equal is given in Fig. 6.

Instruction Set

A major goal of the VAX-11 instruction set design was to provide for effective compiler generated code. Four decisions helped to realize this goal:

1 A very regular and consistent treatment of operators. Thus, for example, since there is a divide longword instruction, there are also divide word and divide byte instructions.

2 An avoidance of instructions unlikely to be generated by a compiler.

3 Inclusion of several forms of common operators. For example the integer add instructions are included in three forms: (a) one operand where the value one is added to an operand, (b) two operands where one operand is added to a second, and (c) three operands where one operand is added to a second and the result stored in a third. Since the VAX-11 instruction format allows fully general specifications of the operands, VAX-11 programs often have the structure (though not the encoding) of the canonic program form proposed in Flynn [1977].

4 Replacement of common instruction sequences with single instructions. Examples of this include procedure calling, multiway branching, loop control, and array subscript calculation.

The effect of these decisions is reflected in several observations. First, despite the larger virtual address and instruction set support for more data types, compiler (and hand) generated code for VAX-11 is typically smaller than the equivalent PDP-11 code for algorithms operating on data types supported by the PDP-11. Second, of the 243 instructions in the instruction set about 75 percent are generated by the VAX-11 FORTRAN compiler. Of the instructions not generated, most operate on data types not part of the FORTRAN language.

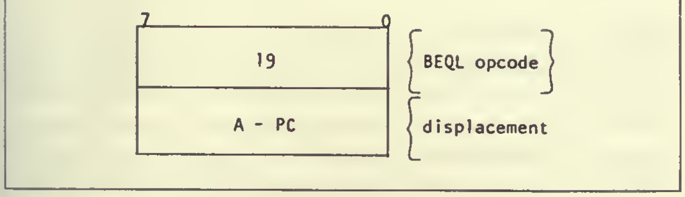

Fig. 6. BEQL A.

A complete list of the VAX-11 instructions is given in Appendix 1. The following gives an overview of the instruction set.

1 *Integer logic and arithmetic*—Byte, word, and longword are the primary data types. A fairly conventional group of arithmetic and logical instructions is provided. The result generating dyadic arithmetic and logical instructions are provided in two and three operand forms. A number of optimizations are included: clear which is a move of zero; test which is a compare against zero; and increment and decrement which are an optimization of add one and subtract one respectively. A complete set of converts is provided which covers both the integer and the floating data types. In contrast to other architectures only a few shift type instructions are provided: it was felt that shifts are mostly used for field isolation which is much more conveniently done with the field instructions described later. In order to support greater than longword precision integer operations, a few special instructions are provided: extended multiply and divide and add with carry and subtract with carry.

2 *Floating point instructions*—Again a conventional group of instructions are included with result producing dyadic operators in two and three operand forms. Several specialized floating point instructions are included. The extended modulus instruction multiplies two floating operands together and stores the integer and fraction parts of the product in separate result operands. The polynomial instruction computes a polynomial from a table of coefficients in memory. Both these instructions employ greater than normal precision and are useful in high accuracy mathematical routines. A convert rounded instruction is provided which implements the ALGOL rather than FORTRAN conventions for converting from floating point to integer.

3 *Address instructions*—The move address instructions store in the result operand the effective address of the source operand. The push address optimizations push on the stack (defined by SP) the effective address of the source operand. The latter are used extensively in subroutine calling.

4 *Field instructions*—The extract field instructions extract a 0 to 32-bit field, sign- or zero-extend it if it is less than 32 bits, and store it in a longword operand. The compare field instructions compare a (sign- or zero-extended if necessary) field against a longword operand. The find first instructions find the first occurrence of a set or clear bit in a field.

5 *Control instructions*—There is a complete set of conditional branches supporting both a signed and, where appropriate, an unsigned interpretation of the various data types. These branches test the condition codes and take a one byte

PC relative branch displacement. There are three unconditional branch instructions: the first taking a one byte PC relative displacement, the second taking a word PC relative displacement, and the third—called jump—taking a general operand specification. Paralleling these three instructions are three branch to subroutine instructions. These push the current PC on the stack before transferring control. The single byte return from subroutine instruction returns from subroutines called by these instructions. There is a set of branch on bit instructions which branch on the state of a single bit and, depending on the instruction, set, clear, or leave unchanged that bit.

The add compare and branch instructions are used for loop control. A step operand is added to the loop control operand and the sum compared against a limit operand. The result of the comparison determines whether the branch is taken. The sense of the comparison is based on the sign of the step operand. Optimizations of loop control include the add one and branch instructions which assume a step of one and the subtract one and branch instructions which assume a step of minus one and a limit of zero.

The case instructions implement the computed go to in FORTRAN and case statements in other languages. A selector operand is checked to see that it lies in range and is then used to select one of table of PC relative branch displacements following the instruction.

6 *Queue instructions*—The queue representation is a doubly linked circular list. Instructions are provided to insert an item into a queue or to remove an item from a queue.

7 *Character string instructions*—The general move character instruction takes five operands specifying the lengths and starting addresses of the source and destination strings and a fill character to be used if the source string is shorter than the destination string. The instruction functions correctly regardless of string overlap. An optimized move character instruction assumes the string lengths are equal and takes three operands. Paralleling the move instructions are two compare character instructions. The move translated characters instruction is similar to the general move character instruction except that the source string bytes are translated by a translation table specified by the instruction before being moved to destination string. The move translated until escape instruction stops if the result of a translation matches the escape character specified by one of its operands. The locate and skip character instructions find respectively the first occurrence or non-occurrence of a character in a string. The scan and span instructions find respectively the first occurrence or non-occurrence of a character within a specified character set in a string. The match characters instruction finds the first occurrence of a substring within a string which matches a specified pattern string.

8 *Packed decimal instructions*—A conventional set of arithmetic instructions is provided. The arithmetic shift and round instruction provides decimal point scaling and rounding. Converts are provided to and from longword integers, leading separate decimal strings, and trailing embedded decimal strings. A comprehensive edit instruction is included.

VAX-11 Procedure Instructions

A major goal of the VAX-11 design was to have a single system wide procedure calling convention which would apply to all inter-module calls in the various languages, calls for operating system services, and calls to the common run time system. Three VAX-11 instructions support this convention: two call instructions which are indistinguishable as far as the called procedure is concerned and a return instruction.

The call instructions assume that the first word of a procedure is an *entry mask* which specifies which registers are to be used by the procedure and thus need to be saved. (Actually only R0-R11 are controlled by the entry mask and bits 15:12 of the mask are reserved for other purposes.) After pushing the registers to be saved on the stack, the call instruction pushes AP, FP, PC, a longword containing the PSW and the entry mask, and a zero valued longword which is the initial value of a condition handler address. The call instruction then loads FP with the contents of SP and AP with the argument list address. The appearance of the stack frame after the call is shown in the upper part of Fig. 7.

The form of the argument list is shown in the lower part of Fig. 7. It consists of an argument count and list of longword arguments which are typically addresses. The CALLG instruction takes two operands: one specifying the procedure address and the other specifying the address of the argument list assumed arbitrarily located in memory. The CALLS instruction also takes two operands: one the procedure address and the other an argument count. CALLS assumes that the arguments have been pushed on the stack and pushes the argument count immediately prior to saving the registers controlled by the entry mask. It also sets bit 13 of the saved entry mask to indicate a CALLS instruction was used to make the call.

The return instruction uses FP to locate the stack frame. It loads SP with the contents of FP and restores PSW through PC by popping the stack. The saved entry mask controls the popping and restoring of R11 through R0. Finally if the bit indicating CALLS was set, the argument list is removed from the stack.

Memory Management Design Alternatives

Memory management comprises the mechanisms used (1) to map the virtual addresses generated by processes to physical memory addresses, (2) to control access to memory (i.e., to control whether a process has read, write, or no access to various areas of memory), and (3) to allow a process to execute even if all of its virtual address space is not simultaneously mapped to physical memory (i.e., to

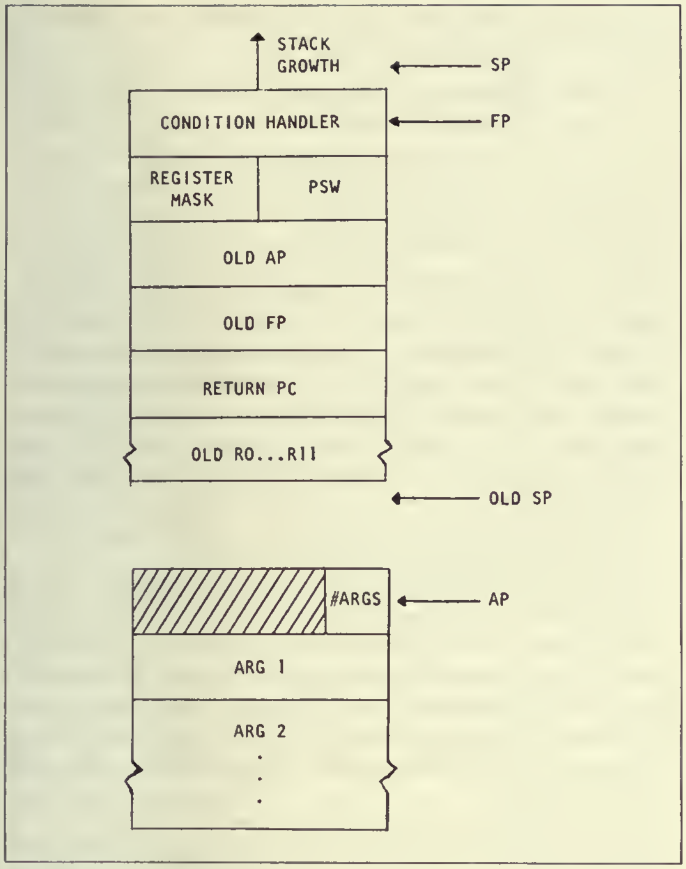

Fig. 7. Stack frame.

provide so called virtual memory facilities). The memory management proved to be the most difficult part of the architecture to design. Three alternatives were pursued and full designs were completed for the first two alternatives and nearly completed for the third. The three alternatives[1] were:

1 A paged form of memory management with access control at the page level and a small number (4) of hierarchical access modes whose use would be dedicated to specific purposes. This represented an evolution of the PDP-11/70 memory management.

2 A paged and segmented form with access control at the segment level and a larger number (8) of hierarchical access modes which would be used quite generally. Although it differed in a number of ways, the design was motivated by the Multics [Organick, 1972; Schroeder and Saltzer, 1971] architecture and the Honeywell 6180 implementation.

3 A capabilities [Needham, 1972; Needham and Walker, 1977] form with access control provided by the capabilities and the ability to page larger objects described by the capabilities.

[1]It should not be construed that memory management is independent of the rest of the architecture. The various memory management alternatives required different definitions of the addressing modes and different instruction level support for addressing.

The first alternative was finally selected. The second alternative was rejected because it was felt that the real increase in functionality provided inadequately offset the increased architectural complexity. The third alternative appeared to offer functionality advantages that could be useful over the longer term. However, it was unlikely that these advantages could be exploited in the near term. Further it appeared that the complexity of the capabilities design was inappropriate for a minicomputer system.

Memory Mapping

The 4.3 gigabyte virtual address space is divided into four regions as shown in Fig. 8. The first two regions—the *program* and *control* regions—comprise the per process virtual address space which is uniquely mapped for each process. The second two regions—the *system* region and a region reserved for future use—comprise the system virtual address space which is singly mapped for all processes.

Each of the regions serves different purposes. The program region contains user programs and data and the top of the region is a dynamic memory allocation point. The control region contains operating system data structures specific to the process and the user stack. The system region contains procedures which are common to all processes (such as those that comprise the operating system and RMS) and (as will be seen later) page tables.

A virtual address has the structure shown in the upper part of Fig. 9. Bits 8:0 specify a byte within a 512 byte *page* which is the

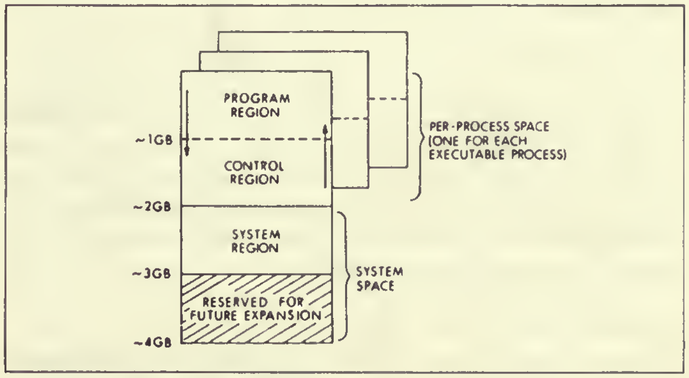
PROGRAM REGION
~1GB
CONTROL REGION
~2GB
SYSTEM REGION
~3GB
RESERVED FOR FUTURE EXPANSION
~4GB
PER-PROCESS SPACE (ONE FOR EACH EXECUTABLE PROCESS)
SYSTEM SPACE

Fig. 8. Virtual address space.

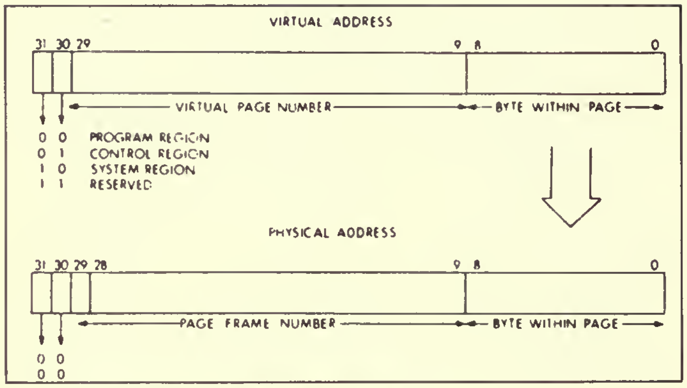

Fig. 9. Virtual and physical addresses.

basic unit of mapping. Bits 29:9 specify a *virtual page number* (VPN). Bits 31:30 select the virtual address region. The mechanism of mapping consists of using the region select bits to select a *page table* which consists of *page table entries* (PTEs). After a check that it is not too large, the VPN is used to index into the page table to select a PTE. The PTE contains either (1) 21-bit physical *page frame number* which is concatenated with the nine low order byte in page bits to form a 30-bit physical address shown in the lower part of Fig. 9, or (2) an indication that the virtual page accessed is not in physical memory. The latter case is called a *page fault.* Instruction execution in the current procedure is suspended and control is transferred to an operating system procedure which will cause the missing virtual page to be brought into physical memory. At this point instruction execution in the suspended procedure can resume transparently.

The page table for the system region is defined by the *system base register* which contains the physical address of the start of the system region page table and the *system length register* which contains the length of the table. Thus the system page table is contiguous in physical memory.

The per process space page tables are defined similarly by the *program* and *control region base registers* and *length registers.* However, the base registers do not contain physical addresses: rather, they contain system region virtual addresses. Thus the per process page tables are contiguous in the system region virtual address space and are not necessarily contiguous in physical memory. This placement of the per process page tables permits them to be paged and avoids what would otherwise be a serious physical memory allocation problem.

Access Control

At a given point in time a process executes in one of four *access modes.* The modes from most privileged to least are called *kernel, executive, supervisor* and *user.* The use of these modes by VAX/VMS is as follows:

1 Kernel—Interrupt and exception handling, scheduling, paging, physical I/O, etc.

2 Executive—Logical I/O as provided by RMS.

3 Supervisor—The command interpreter.

4 User—User procedures and data.

The accessibility of each page (read, write, or no access) from each access mode is specified in the PTE for that page. Any attempt to improperly access a page is suppressed and control is transferred to an operating system procedure. The accessibility is assumed hierarchically ordered: if a page is writable from any given mode, it is also readable; and if a page is accessible from a less privileged mode, it is accessible from a more privileged mode. Thus, for example, a page can be readable and writable from kernel mode, only readable from executive mode, and inaccessible from supervisor and user modes.

A procedure executing in a less privileged mode often needs to call a procedure which executes in a more privileged mode: e.g., a user program needs an operating system service performed. The access mode is changed to a more privileged mode by executing a change mode instruction which transfers control to a routine executing at the new access mode. A return is made to original access mode by executing a return from exception or interrupt instruction (REI).

The *current access mode* is stored in the *processor status longword* (PSL) whose low order 16 bits comprise the PSW. Also stored in the PSL is the *previous access mode*; i.e., the access mode from which the current access mode was called. The previous mode information is used by the special probe instructions which validate arguments passed in cross access mode calls.

Procedures running at each of the access modes require a separate stack with appropriate accessibility. To facilitate this, each process has four copies of SP which are selected according to the current access mode field in the PSL. A procedure always accesses the correct stack by using R14.

In an earlier section, it was stated that the VAX-11 standard CALL instruction is used for all calls including those for operating system services. Indeed procedures do call the operating system using the CALL instruction. The target of the CALL instruction is the minimal procedure consisting of an entry mask, a change mode instruction, and a return instruction. This access mode changing is transparent to the calling procedure.

Interrupts and Exceptions

Interrupts and exceptions are forced changes in control flow other than that explicitly indicated by the executing program. The distinction between them is that interrupts are normally unrelated to the currently executing program while exceptions are a direct consequence of program execution. Examples of interrupt conditions are status changes in I/O devices while examples of exception conditions are arithmetic overflow or a memory management access control violation.

VAX-11 provides a 31 priority level interrupt system. Sixteen levels (16-31) are provided for hardware while 15 levels (1-15) are provided for software. (Level 0 is used for normal program execution.) The current *interrupt priority level* (IPL) is stored in a field in the PSL. When an interrupt request is made at a level higher than IPL, the current PC and PSL are pushed on the stack and new PC obtained from a vector selected by the interrupt requester (a new PSL is generated by the CPU). Interrupts are serviced by routines executing with kernel mode access control. Since interrupts are appropriately serviced in a system wide rather than a specific process context, the stack used for interrupts is defined by another stack pointer called the *interrupt stack pointer*. (Just as for the multiple stack pointers used in process context, an interrupt routine accesses the interrupt stack using R14.) An interrupt service is terminated by execution of an REI instruction which loads PC and PSL from the top two longwords on the stack.

Exceptions are handled like interrupts except for the following: (1) since exceptions arise in a specific process context, the kernel mode stack for that process is used to store PC and PSL and (2) additional parameters (such as the virtual address causing a page fault) may be pushed on the stack.

Process Context Switching

From the standpoint of the VAX-11 architecture, the process state or context consists of:

1 The 15 general registers R0-R13 and R15.
2 Four copies of R14 (SP): one for each of kernel, executive, supervisor, and user access modes.
3 The PSL.
4 Two base and two limit registers for the program and control region page tables.

This context is gathered together in a data structure called a *process control block* (PCB) which normally resides in memory. While a process is executing, the process context can be considered to reside in processor registers. To switch from one process to another it is required that the process context from the previously executing process be saved in its PCB in memory and the process context for the process about to be executed to be loaded from its PCB in memory. Two VAX-11 instructions support context switching. The save process context instruction saves the complete process context in memory while the load process context instruction loads the complete process context from memory.

I/O

Much like the PDP-11, VAX-11 has no specific I/O instructions. Rather, I/O devices and device controllers are implemented with a set of registers which have addresses in the physical memory address space. The CPU controls I/O devices by writing these registers; the devices return status by writing these registers and the CPU subsequently reading them. The normal memory management mechanism controls access to I/O device registers and a process having a particular device's registers mapped into its address space can control that device using the regular instruction set.

Compatibility Mode

As mentioned in the VAX-11 overview, compatibility mode in the VAX-11 architecture provides the basic PDP-11 instruction set less privileged and floating point instructions. Compatibility mode is intended to support a user as opposed to an operating system environment. Normally a compatibility mode program is combined with a set of native mode procedures whose purpose is to map service requests from some particular PDP-11 operating system environment into VAX/VMS services.

In compatibility mode the 16-bit PDP-11 addresses are zero-extended to 32-bits where standard native mode mapping and access control apply. The eight 16-bit PDP-11 general registers overmap the native mode general registers R0-R6 and R15 and thus the PDP-11 processor state is contained wholly within the native mode processor state.

Compatibility mode is entered by setting the compatibility mode bit in the PSL. Compatibility mode is left by executing a PDP-11 trap instruction (such as used to make operating service requests), and on interrupts and exceptions.

VAX-11/780 Implementation

VAX-11/780

The VAX-11/780 computer system is the first implementation of the VAX-11 architecture. For instructions executed in compatibility mode, the VAX-11/780 has a performance comparable to the PDP-11/70. For instructions executed in native mode, the -11/780 has a performance in excess of the -11/70 and thus represents the new high end of the -11 (LSI-11, PDP-11, VAX-11) family.

A block diagram of the -11/780 system is given in Fig. 10. The system consists of a central processing unit (CPU), the console subsystem, the memory subsystem, and the I/O subsystem. The

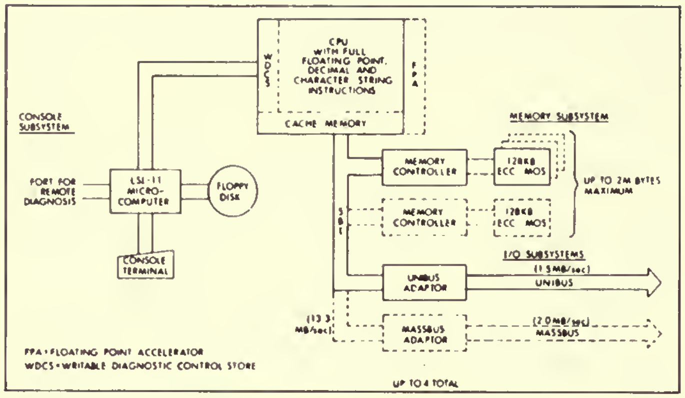

Fig. 10. VAX-11/780 system.

CPU and the memory and I/O subsystems are joined by a high speed synchronous bus called the *Synchronous Backplane Interconnect* (SBI).

CPU

The CPU is a microprogrammed processor which implements the native and compatibility mode instruction sets, the memory management, and the interrupt and exception mechanisms. The CPU has 32-bit main data paths and is built almost entirely of conventional Schottky TTL components.

To reduce effective memory access time the CPU includes an 8K byte write through cache or buffer memory. The cache organization is 2-way associative with an 8-byte block size. To reduce delays due to writes, the CPU includes a write buffer. The CPU issues the write to the buffer and the actual memory write takes place in parallel with other CPU activity.

The CPU contains a 128 entry address translation buffer which is a cache of recent virtual to physical translations. The buffer is divided into two 64 entry sections: one for the per process regions and one for the system region. This division facilitates permitting the system region translations to remain unaffected by a process context switch.

A fourth buffer in the CPU is the 8-byte instruction buffer. It serves two purposes. First, it decomposes the highly variable instruction format into its basic components and, second, it constantly fetches ahead to reduce delays in obtaining the instruction components.

The CPU includes two standard clocks. The programmable real-time clock is used by the operating system for local timing purposes. The time-of-year clock with its own battery backup is the long term time references for the operating system. It is automatically read on system startup to eliminate the need for manual entry of data and time.

The CPU includes 12K bytes of writable diagnostic control store (WDCS) which is used for diagnostic purposes, implementation of certain instructions, and for future microcode changes. As an option for very sophisticated users, another 12K bytes of writable control store is available.

A second option is the floating point accelerator (FPA). Although the basic CPU implements the full floating point instruction set, the FPA provides high speed floating point hardware. It is logically invisible to programs and only affects their running time.

Console Subsystem

The console subsystem is centered around an LSI-11 computer with 16K bytes of RAM and 8K bytes of ROM (used to store the LSI-11 bootstrap, LSI-11 diagnostics, and console routines). Also included are a floppy disk, an interface to the console terminal, and a port for remote diagnostic purposes.

The floppy disk in the console subsystem serves multiple purposes. It stores the main system bootstrap and diagnostics and serves as a medium for distribution of software updates.

SBI

The SBI is the primary control and data transfer path in the -11/780 system. Because the cache and write buffer largely decouple the CPU performance from the memory access time, the SBI design was optimized for bandwidth and reliability rather than the lowest possible access time.

The SBI is a synchronous bus with a cycle time of 200 nsec. The data path width of the SBI is 32 bits. During each 200 nsec cycle either 32 bits of data or a 30-bit physical address can be transferred. Since each 32-bit read or write requires transmission of both address and data, two SBI cycles are used for a complete transaction. The SBI protocol permits 64-bit reads or writes using one address cycle and two data transfer cycles: the CPU and I/O subsystem use this mode whenever possible. For read transactions the bus is reacquired by the memory in order to send the data: thus the bus is not held during the memory access time.

Arbitration of the SBI is distributed: each interface to the SBI has a specific priority and its own bus request line. When an interface wishes to use the bus, it asserts its bus request line. If at the end of a 200 nsec cycle there are no interfaces of higher priority requesting the bus, the interface takes control of the bus.

Extensive checking is done on the SBI. Each transfer is parity checked and confirmed by the receiver. The arbitration process and general observance of the SBI protocol are checked by each SBI interface during each SBI cycle. The processor maintains a running 16-cycle history of the SBI: any SBI error condition causes this history to be locked and preserved for diagnostic purposes.

Memory Subsystem

The memory subsystem consists of one or two memory controllers with up to 1M bytes of memory on each. The memory is organized in 64-bit quadwords with an 8-bit ECC which provides single bit error correction and double bit error detection. The memory is built of 4K MOS RAM components.

The memory controllers have buffers which hold up to four memory requests. These buffers substantially increase the utilization of the SBI and memory by permitting the pipelining of multiple memory requests. If desired, quadword physical addresses can be interleaved across the memory controllers.

As an option, battery backup is available which preserves the contents of memory across short term power failures.

I/O Subsystem

The I/O subsystem consists of buffered interfaces or adapters between the SBI and the two types of peripheral busses used on PDP-11 systems: the Unibus and the Massbus. One Unibus adapter and up to four Massbus adapters can be configured on a VAX-11/780 system.

The Unibus is a medium speed multiplexor bus which is used as a primary memory as well as peripheral bus in many PDP-11 systems. It has an 18-bit physical address space and supports byte and word transfers. In addition to implementing the Unibus protocol and transmitting interrupts to the CPU, the Unibus adapter provides two other functions. The first is mapping 18-bit Unibus addresses to 30-bit SBI physical addresses. This is accomplished in a manner substantially identical to the virtual to physical mapping implemented by the CPU. The Unibus address space is divided into 512 512-byte pages. Each Unibus page has a page table entry (residing in the Unibus adapter) which maps addresses in that page to physical memory addresses. In addition to providing address translation, the mapping permits contiguous transfers on the Unibus which cross page boundaries to be mapped to discontiguous physical memory page frames.

The second function performed by the Unibus adapter is assembling 16-bit Unibus transfers (both reads and writes) into 64-bit SBI transfers. This operation (which is applicable only to block transfers such as from disks) appreciably reduces SBI traffic due to Unibus operations. There are 158-byte buffers in the Unibus adapter permitting 15 simultaneous buffered transactions. Additionally there is an un-buffered path through the Unibus adapter permitting an arbitrary number of simultaneous un-buffered transfers.

The Massbus is a high speed block bus used primarily for disks and tapes. The Massbus adapter provides much the same functionality as the Unibus adapter. The physical addresses into which transfers are made are defined by a page table: again this permits contiguous device transfers into discontiguous physical memory.

Buffering is provided in the Massbus adapter which minimizes the probability of device overruns and assembles data into 64-bit units for transfer over the SBI.

References

Bell and Strecker [1976]; Bell et al. [1970]; Flynn [1977]; Levy and Eckhouse [1980]; McLean [1977]; Myers [1977b]; Needham [1972]; Needham and Walker [1977]; Organick [1972]; Schrocker and Saltzer [1971].

APPENDIX 1 VAX-11 INSTRUCTION SET

Integer and Floating Point Logical Instructions

MOV-	Move(B,W,L,F,D,Q)†
MNEG-	Move Negated(B,W,L,F,D)
MCOM-	Move Complemented(B,W,L)
MOVZ-	Move Zero-Extended(BW,BL,WL)
CLR-	Clear(B,W,L=F,Q=D)
CVT-	Convert(B,W,L,F,D)(B,W,L,F,D)
CVTR-L	Convert Rounded(F,D) to Longword
CMP-	Compare(B,W,L,F,D)
TST-	Test(B,W,L,F,D)
BIS-2	Bit Set(B,W,L)2-Operand
BIS-3	Bit Set(B,W,L)3-Operand
BIC-2	Bit Clear(B,W,L)2-Operand
BIC-3	Bit Clear(B,W,L)3-Operand
BIT-	Bit Test(B,W,L)
XOR-2	Exclusive OR(B,W,L)2-Operand
XOR-3	Exclusive OR(B,W,L)3-Operand
ROTL	Rotate Longword
PUSHL	Push Longword

Integer and Floating Point Arithmetic Instructions

INC-	Increment(B,W,L)
DEC-	Decrement(B,W,L)
ASH-	Arithmetic Shift(L,Q)
ADD-2	Add(B,W,L,F,D)2-Operand
ADD-3	Add(B,W,L,F,D)3-Operand
ADWC	Add with Carry
ADAWI	Add Aligned Word Interlocked

†B = byte, W = word, L = longword, F = floating, D = double floating, Q = quadword, S = set, C = clear.

SUB-2	Subtract(B,W,L,F,D)2-Operand
SUB-3	Subtract(B,W,L,F,D)3-Operand
SBWC	Subtract with Carry
MUL-2	Multiply(B,W,L,F,D)2-Operand
MUL-3	Multiply(B,W,L,F,D)3-Operand
EMUL	Extended Multiply
DIV-2	Divide(B,W,L,F,D)2-Operand
DIV-3	Divide(B,W,L,F,D)3-Operand
EDIV	Extended Divide
EMOD-	Extended Modulus(F,D)
POLY-	Polynomial Evaluation(F,D)

Index Instruction

INDEX	Compute Index

Packed Decimal Instructions

MOVP	Move Packed
CMPP3	Compare Packed 3-Operand
CMPP4	Compare Packed 4-Operand
ASHP	Arithmetic Shift Round and Packed
ADDP4	Add Packed 4-Operand
ADDP6	Add Packed 6-Operand
SUBP4	Subtract Packed 4-Operand
SUBP6	Subtract Packed 6-Operand
MULP	Multiply Packed
DIVP	Divide Packed
CVTLP	Convert Long to Packed
CVTPL	Convert Packed to Long
CVTPT	Convert Packed to Trailing
CVTTP	Convert Trailing to Packed
CVTPS	Convert Packed to Separate
CVTSP	Convert Separate to Packed
EDITPC	Edit Packed to Character String

Character String Instructions

MOVC3	Move Character 3-Operand
MOVC5	Move Character 5-Operand
MOVTC	Move Translated Characters
MOVTUC	Move Translated Unit Character
CMPC3	Compare Characters 3-Operand
CMPC5	Compare Characters 5-Operand
LOCC	Locate Character
SKPC	Skip Character
SCANC	Scan Characters
SPANC	Span Characters
MATCHC	Match Characters

Variable-Length Bit Field Instructions

EXTV	Extract Field
EXTZV	Extract Zero-Extended Field
INSV	Insert Field
CMPV	Compare Field
CMPZV	Compare Zero-Extended Field
FFS	Find First Set
FFC	Find First Clear

Branch on Bit Instructions

BLB-	Branch on Low B(S,Cl)
BB-	Branch on Bit(S,Cl)
BBS-	Branch on Bit Set and(S,Cl)Bit
BBC	Branch on Bit Clear and(Set,Clear)Bit
BBSSI	Branch on Bit Set and Set Bit Interlocked
BBCCI	Branch on Bit Clear and Clear Bit Interlocked

Queue Instructions

INSQUE	Insert Entry in Queue
REMQUE	Remove Entry from Queue

Address Manipulation Instructions

MOVA-	Move Address(B,W,L=F,Q=D)
PUSHA-	Push Address(B,W,L=F,Q=D)on Stack

Processor State Instructions

PUSHR	Push Registers on Stack
POPR	Pop Registers from Stack
MOVPSL	Move from Processor Status Longword
BISPSW	Bit Set Processor Status Word
BICPSW	Bit Clear Processor Status Word

Unconditional Branch and Jump Instructions

BR-	Branch with(B,W)Displacement
JMP	Jump

Branch on Condition Code

BLSS	Less Than
BLSSU	Less Than Unsigned
(BCS)	(Carry Set)
BLEQ	Less Than or Equal
BLEQU	Less Than or Equal Unsigned
BEQL	Equal
(BEQLU)	(Equal Unsigned)
BNEQ	Not Equal
(BNEQU)	(Not Equal Unsigned)
BGTR	Greater Than
BGTRU	Greater Than Unsigned
BGEQ	Greater Than or Equal
BGEQU	Greater Than or Equal Unsigned
(BCC)	(Carry Clear)
BVS	Overflow Set
BVC	Overflow Clear

Loop and Case Branch

ACB-	Add, Compare and Branch(B,W,L,F,D)
AOBLEQ	Add One and Branch Less Than or Equal
AOBLSS	Add One and Branch Less Than
SOBGEQ	Subtract One and Branch Greater Than or Equal
SOBGTR	Subtract One and Branch Greater Than
CASE-	Case on(B,W,L)

Subroutine Call and Return Instructions

BSB	Branch to Subroutine with(B,W,) Displacement
JSB	Jump to Subroutine
RSB	Return from Subroutine

Procedure Call and Return Instructions

CALLG	Call Procedure with General Argument List
CALLS	Call Procedure with Stack Argument List
RET	Return from Procedure

Access Mode Instructions

CHM	Change Mode to (Kernel,Executive,Supervisor, User)
REI	Return from Exception or Interrupt
PROBER	Probe Read
PROBEW	Probe Write

Privileged Processor Register Control Instructions

SVPCTX	Save Process Context
LDPCTX	Load Process Context
MTPR	Move to Process Register
MFPR	Move from Processor Register

Special Function Instructions

CRC	Cyclic Redundancy Check
BPT	Breakpoint Fault
XFC	Extended Function Call
NOP	No Operation
HALT	Halt