Table of Contents

1.0 PART ONE - THE BASIC MACHINE

1.1 Introduction

This is a proposal for a new machine.

The machine is quite similar to a PDP10. It is assumed that the reader is intimately familiar with the PDP10. This document shall only describe the differences. Anything which is not covered in this document can be assumed to function in the same manner as a PDP10.

Indeed, the machine is very similar to a PDP10. There are, however, some major differences. We do not expect that there will be a single program that will run on the new machine without at least some modification. PDP10 compatibility was not our primary goal. Instead, our goal was to produce the best machine possible. Compatibility is a good idea only when it does not sacrifice quality.

We do not anticipate that programs will run without modification. Some programs will require more modification than others. Many will require a complete rewrite. The operating system itself falls into this last category. Neither TOPS10 nor TOPS20 can be modified to run on the new machine. The new machine will run an operating system of a totally new design. It is not our purpose here to design the operating system, just the machine.

We will define the instruction set, but only that portion that applies to user mode. We have tried to avoid defining those things which are unique to exec mode. We have not defined the format of the I/O registers nor the page maps. We do, however, cover a few aspects of exec (e.g. op-codes 31-37). We have tried to keep these to a minimum.

The machine comes in four flavors: the basic machine and three options (the vector option, the stack option, and the arithmetic string option).

1.2 Bits

This is a 32 bit machine not 36. The bits are numbered from left to right as follows:

```
!0 0!0 0 0!0 0 0!1 1 1!1 1 1!1 1 2!2 2 2!2 2 2!2 3 3!3 3 3!3 3 3!
!0 1!2 3 4!5 6 7!0 1 2!3 4 5!6 7 0!1 2 3!4 5 6!7 0 1!2 3 4!5 6 7!
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! !
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
! !     !     !     !     !     ! !     !     !     !     !     !
                                !
             LH                 !               RH
```

Note that the bits are divided into groups of three. This is an octal machine not Hex.

Note that the bits are numbered in octal not decimal. This will eliminate a long standing source of confusion. No longer will people be confused by the default radix of the POINT pseudo op (nor the B shifts). Why should the radix be any different than that used anywhere else?

A 32 bit word is easily divided into 4 bytes of 8 bits each. This is not, however, a byte oriented machine. It is a word oriented machine.

Although the word size is 32, the address size is 31. The machine supports a 31 bit virtual address. The address specifies a word number not a byte number.

1.3 Effective Address Calculation

Refer to the following diagrams:

Instruction format:

```
!0                1!1     1!1!1                                    3!
!0                0!1     4!5!6                                    7!
+-----------------+-------+-+-------------------------------------+
!     OPCODE      !  AC   !0!                  Z                  !
+-----------------+-------+-+-------------------------------------+
!    9 bits       !4 bits ! !               18 bits               !

!0                1!1     1!1!1!1     2!2                          3!
!0                0!1     4!5!6!7     2!3                          7!
+-----------------+-------+-+-+-------+---------------------------+
!     OPCODE      !  AC   !1!I!   X   !             Y             !
+-----------------+-------+-+-+-------+---------------------------+
!    9 bits       !4 bits ! ! !4 bits !          13 bits          !
```

Indirect Word:

```
!0!0                                                               3!
!0!1                                                               7!
+-+---------------------------------------------------------------+
!0!                             Z                                 !
+-+---------------------------------------------------------------+
! !                          31 bits                              !

!0!0!0     0!0     1!1                                             3!
!0!1!2     5!6     1!2                                             7!
+-+-+-------+-------+---------------------------------------------+
!1!I!  X1   !  X2   !                      Y                      !
+-+-+-------+-------+---------------------------------------------+
! ! !4 bits !4 bits !                   22 bits                   !
```

Effective address calculation proceeds as follows:

1. If bit 15 (the Mode bit) is zero, then bits 16-37 give the effective address. This quantity is not sign extended, bits 0-15 of E are set to zero.

2. If bit 15 is one, then bits 16-37 are interpreted as I, X, and Y. The Y field is sign extended and added to the contents of the index register specified by X. The index register is taken as a full 32 bit quantity (the LH is not ignored). Note that registers 0 and 17 cannot be used as index registers. X=0 indicates that indexing is not to

take place. X=17 indicates that the PC should be used as an index register (this is known as "PC relative addressing").

Programmers of the VAX will potentially misconstrue the above statement. We do not mean to imply that the PC is addressable as register 17. Register 17 and the PC are two totally independent quantities. The programmer is free to use register 17 as a general purpose register. He may store any value that he wishes in register 17. He may not, however, use register 17 for the purposes of indexing. If the programmer puts a 17 in the X field, he will not get the value of register 17. He will, instead, get the value of the PC. In all other contexts register 17 will functions normally. Register 17 is much like register 0: It is a general purpose register but cannot be used for indexing.

Perhaps "PC relative addressing" is a poor choice of words. The addressing isn't always relative to the PC. It's sometimes relative to other quantities. X=17 merely indicates that relative addressing is to take place. The base of relativity is taken from context (but its usually the PC). During an XCT instruction, for example, X=17 indicates that the addressing is relative to the location of the instruction being executed and not relative to the location of the XCT itself.

Those of you who doubt the usefulness of position independent addressing should refer to section 1.6 (Programming Environment).

3. If bit 16 (The I bit) is one, then indirect addressing takes place. An indirect word is fetched from the word specified by X and Y. If bit 0 (the Mode bit) in the indirect word is zero, then bits 1-37 give the effective address. This field is not sign extended. 31 bits is the largest virtual address you can have.

   If bit 0 of the indirect word is one, then bits 1-37 are interpreted as I, X1, X2, and Y. The Y field is sign extended.

   Note that there are two index registers. This is convenient for accessing two dimensional arrays. Note, however, that a two dimensional array cannot be accessed in position independent code. One of the two index registers will be set to 17.

Note that X=17 indicates that addressing is relative to the location that the indirect word was fetched from and not relative to the PC.

In general, X=17 indicates that the MA register should be added to the effective address. The MA register contains the address of the last location fetched from memory. In most cases MA will contain a copy of the PC. It's possible, however, that the MA will contain: 1). The address of an instruction being XCT'ed 2). The address of an indirect word 3). The address of a byte pointer

Without using an indirect word, the effective address can specify any quantity from -2**12 up to +2**18-1. By using an indirect word, the effective address can specify any quantity from -2**21 up to +2**31-1.

Effective address calculation, as on the PDP10, ignores all overflows.

At this point the diligent reader may wish to turn to appendix 5.2. This section contains a statistical analysis of the effective address calculation. How often is the mode bit zero? How often is it one? How often is the I bit used? How often is X used?

The diligent reader may also wish to study appendix 5.1 . This section documents the algorithm used by the microcode to compute the effective address.

1.4 Effective Address Examples

It is important to stress that the effective address is a full 32 bit quantity. Perhaps some examples will clarify:

1. The instruction MOVEI 1,-1 causes the assembler to generate the instruction 10043017777. This instruction sets all bits in AC 1.

2. The instruction ADDI 1,-47 generates the instruction 13443017731. It is equivalent to SUBI 1,+47 .

3. The instruction CAIE 1,-2 is not equivalent to CAIE 1,777776 . The former generates 14103017776 . The latter generates 14102777776 . The former skips if AC1 is 37777777776 . The latter skips if AC1 is 00000777776.

4. The instruction TRO 1,-1 is not equivalent to TRO 1,177777 (see section 1.11.19). The latter sets the RH to ones. The former sets both halves to ones.

5. MOVEI T1,(T2) is not equivalent to MOVEI T1,@T2 . They differ if the sign bit of T2 is on.

1.5 AOBJN Loops

Note what happens when a memory reference is attempted and bit 0 of the effective address is one. The machine only supports 31 bit virtual addresses. Therefore bit 0 should always be zero. If, however, bit 0 is one, then the entire LH of the address is ignored. The result is a 16 bit address.

This feature will enable some of the older PDP10 programs to run without modification. Study the following example:

```
        HRLZI   T1,-TABLEN
        SETZ    T2,
LOOP:   ADD     T2,TAB(T1)
        AOBJN   T1,LOOP
```

This is, at best, a kludge. It does, however, allow the program to run provided that the program is loaded at an address below 2**16.

Note that the process of truncating the LH does not take place at the time of effective address calculation. Instead it takes place at the time the actual reference is made (i.e. at the time of virtual to physical address translation). Example: The instruction MOVEI T1,-1 sets T1 to 37777777777 (not 177777). The instruction MOVE T1,-1 sets T1 to the contents of location 177777 (not location 37777777777).

1.6 Programming Environment

One cannot understand the machine fully without knowing the programming environment under which the machine is intended to run. Key in this environment is the usage of multiple sharable segments.

Consider, for a moment, the total amount of memory used by SCAN and WILD. The majority of all TOPS10 utilities have copies of SCAN and WILD linked into them. The amount of space used is enormous.

Clearly, there would be a tremendous savings if a single copy of SCAN and WILD could be shared amongst the utilities. This requires, however, that a given process be allowed to attach to more than one sharable segment. For example, a user running DIRECT might attach to four different segments: SCAN, WILD, HELPER, and DIRECT itself. Meanwhile other programs might attach to FOROTS, LIBOL, DBMS, SORT, the scientific subroutine library, GLXLIB, and a host of user written subroutines.

The question quickly becomes one of virtual address space. At what address will SCAN exist? Where will WILD be?

Hypothetically lets assign an address of 500000 to SCAN. Lets put WILD at 512000, HELPER at 521000, SORT at 523000, etc.

Will there be enough virtual address space to fit all the possible subroutines that anyone would ever want to attach to? Clearly the answer is "No". Regardless of how large the address space, its only a matter of time before the user base exceeds it.

To solve the problem, then, we must take a radically different tact. The solution we have chosen is a simple one: For each of the users sharing a particular segment, the segment will exist at a different virtual address.

It isn't the user who picks the virtual address, it's the monitor. The user, for example, might tell the monitor merely that he wishes to attach to SCAN. The monitor would pick the next available address and report this to the user.

The ramifications of this approach are great. Firstmost, each of the segments must be written to be position independent. This has had a profound influence on the design of the instruction set.

Lets digress now for a moment and discuss what we mean by the word "segment". Our usage is quite different than the TOPS10 concept of a HISEG. Our concept is a superset of the HISEG concept.

Each segment shall be divided into two regions:

1). The Code Region - This region consists of zero or more pages. The region is reentrant: it's both write locked and shareable. As the name implys, the region is typically used to store code.

2). The Data Region - This region consists of zero or more pages. The region is write enabled. It is not shareable. The data region is used to store local variables.

Note that the code region is analagous to the TOPS10 concept of a HISEG. The data region is analagous to the concept of a LOWSEG. The addressing, however, is quite different. TOPS10 seperates the HISEG and the LOWSEG by a wide gap in addresses. The HISEG is loaded at 400000 and the LOWSEG is loaded at 0. On the new machine, however, the code region is immediately adjacent to the data region.

Example: A segment like SCAN would take about 8 pages. These would be 8 consecutive pages. Two of the pages would be data pages. The other six pages would be code pages.

SCAN contains a call to HELPER. The HELPER segment, however, is position independent. The address that HELPER is loaded at will not be known in advance. The address cannot be hard coded into SCAN.EXE . We therefore reserve an extra word in SCAN's data region. The monitor will store the address of HELPER in this location as soon as the information is known. SCAN will use this location as an indirect word whenever it wishes to call HELPER.

Note that the indirect word must be in the data region and not the code region. The address stored at this location is potentically different for each user of SCAN and HELPER.

How does the monitor know where to store the address of HELPER? It finds this information in the EXE directory of the SCAN segment. It is one of two new things that we've added to the directory. We've added two lists: the INTERN list and the EXTERN list.

The INTERN list is a list of all the entry points to the segment. Each item in the list contains two pieces of information: 1). The symbolic name of the entry point, and 2). The offset within the segment of the entry point.

The EXTERN list is a list of all the external subroutines called by this segment. Each item in the list contains three pieces of information: 1). The symbolic name of the subroutine, 2). The offset within this segment where the address of the subroutine is to be stored, and 3). The filespec of the segment where the subroutine can be found.

To continue our example, we see that DIRECT has two items in its EXTERN list: SCAN and WILD. SCAN, in turn, has one item in its EXTERN list: HELPER. Both HELPER and WILD are "leaf nodes" (they don't call anything). The monitor will load all four segments when the user says "RUN DIRECT". The user of DIRECT need not be aware of which segments the program calls.

One might suspect that this process will make the RUN command quite slow. In actuallity, however, the difference will not be noticeable. The frequently used segments will all tend to stay in core. The monitor need not load them, it merely attaches to them (at least for the code region). The data region, however, might need to be loaded from the EXE file upon each invocation. But this can be avoided if the data region is "null" (i.e. it is known that the data region initially contains nothing but zeros). Even if non-null we can still attach to an existing copy of the data region. We merely mark the pages as "copy on write".

Disclaimer: Our usage of SCAN and WILD should be taken merely as an example. We do not mean to imply that the new system will support anything remotely similar to present day SCAN or WILD.

1.7 Assembler/Linker

Programming on this machine can be made significantly easier if we assume several changes in the assembler/linker.

1.7.1 Automatic Generation Of Links -

In position independent code, if the location referenced by the effective address of an instruction is not within plus or minus 2**12 of the PC then an indirect word must be used. The indirect word must be located within 2**12 of the PC but the location pointed to by the indirect word can be anywhere within 2**21.

These indirect words will not be coded manually. The assembler and/or linker will insert them automatically whenever it sees that the target isn't within 2**12. The programmer need not even be aware that this is taking place. He need not be aware of the distance to the target. The programmer, for example, might code "JRST FOO##". If need be, the assembler/linker will automatically convert this to "JRST @[FOO##]".

Terminology: We shall use the term "link word" to refer to an indirect word which was inserted automattically. This will distinguish it from an indirect word which was deliberately coded.

1.7.2 GAP -

The assembler will have a new pseudo op: GAP. It will be used to indicate an unreachable position in the code. I.E. A position where the assembler is free to insert link words (if need be). Example:

```
        SKIPE   T1      ;IF THEN ELSE
        JRST    FOO
        ...
        JRST    BAR
        GAP             ;UNREACHABLE CODE, INSERT LINKS HERE
FOO:    ...
BAR:
```

Note that it is not necessary to insert GAPs unless the module is bigger than 2**12. For small modules the link words can all be inserted at the end.

One way of thinking of it is that GAP merely subdivides a PSECT into several smaller PSECTS. Each of the resulting PSECTs is smaller that 2**12. Thus there's always room to insert the link words at the end of the PSECT.

The assembler will support two types of PSECTS: code PSECTS and data PSECTS (see section 1.6 p8). Code PSECTS are those which are loaded into the code region of a segment. Data PSECTS are those which are loaded into the data region. The GAP pseudo op will be supported for both types of PSECTS.

1.7.3 Who Inserts Link Words -

Some of the link words can be inserted by the assembler. Others, however, must be done by the linker. With a global symbol, for example, it is not known until link time whether the symbol is within 2**12. It is not known whether a link word will be needed.

Despite the fact that the assembler is able to do some of the links, it is our recommendation that the assembler not attempt this. We recommend that link words be inserted only by the linker. This will consolidate the code in a single place.

1.7.4 Binding The Mode -

One unusual property of this machine is its ambiguity. For a given assembly language statement, the effective address can often be coded in several different ways. Which one should be generated?

To decide this, one must first answer two questions: 1). Where is the code to be loaded? and 2). Should position independent addressing be used?

We propose that these issues should not be resolved until link time. The assembler should not attempt to resolve them. In fact the assembler should not generate any code for bits 15-37 of any instruction (at least not in those cases where the effective address is relocatable). Instead the assembler will place two pieces of information into the REL file: The PSECT number being referenced, and the offset into the PSECT. Its ultimately the linker who chooses the addressing mode. Its the linker who generates the code for bits 15-37 of the instruction. The linker will have a variety of switches to control the decision making process.

1.7.5 Literals -

Consider the following example: There exists a module which is slightly larger than 2**12. In this module there is an integer literal which is referenced ten times. Nine of the references are within 2**12 of the literal pool. These nine references do not require link words. The tenth, however, might conceivably use a link word (which is inserted at a GAP). But that would be a silly way to code it. Instead of inserting a link word we should insert a second copy of the literal. The resulting code would take the same amount of space but would run much faster.

The plan requires, however, that it be the linker who resolves the literals. The assembler can no longer fulfill this function as the assembler doesn't know which references require link words.

The assembler must define, in the REL file, the value of each literal. Its up to the linker to decide where to place each of the literals. It doesn't necessarily place them all in a single pool. Moreover, many of the literals will be duplicated in several pools.

This plan has the added advantage that literals can be shared across modules.

1.7.6 Linker Optimization -

Note that the efficiency of a program can be influenced by the order that the modules are linked. A pair of modules with frequent references to each other should be loaded in the same 2**12 of address space. This will avoid needlesss link words.

In a large program it can be extremely difficult to decide which order to load the modules. We could theoretically write an optimizing linker which would make the decision for us. Our research shows, however, that such a linker isn't really necessary (see section 5.2). It seems that link words are fairly rare. We therefore suspect that the difference between an optimized linker and an unoptimized linker would not be substantial.

We recommend that the initial implementation not be optimized. At a later date, however, an optimizing linker would definitely make a good project.

The optimizing linker might use any of a wide variety of algorithms. The better ones are likely to be combinitorial (and therefore quite slow). We believe, however, that heuristics will be found that run quite fast and produce results which are close to optimal.

One proposed heuristic is to choose the order based on reference density. Load first the module which resolves the greatest number of outstanding references. But give preference to small modules. In other words, look not at the actual number of references but rather the density of references. Load first the module with the highest density.

1.7.7 Symbol Names -

The assembler/linker must support symbols longer than 6 characters. Some of the opcodes, in fact, are longer.

1.7.8 Indirect Words -

The pseudo op "Z" will generate an indirect word (see section 1.3). The syntax will be:

```
Z       [@][addr][(X1[,X2])]
```

1.8 Rules For Op-code Assignment

1.8.1 Duplicate Op-codes -

One of the most reknown aspects of the PDP10 is its duplicate op-codes. For example code 670 (TDO) is identical to code 434 (OR). This is a remnant leftover from the PDP6. The PDP6 was a hardwired machine. By assigning duplicate op-codes, the designers were able to save alot of combinitorial logic. Besides, there were op-codes to spare.

Over the years, however, numerous op-codes have been added. There are few left now and we can no longer afford to waste a single one. Moreover, all the modern machines are microprogrammed. Dispatch is now handled by a RAM and the assignment of opcodes has no affect on the amount of logic used.

The new machine will have no duplicate op-codes. The assembler will map both TDO and OR to the same op-code (code 434). Code 670 will be recycled and used for a new instruction. The other duplicates will be handled in a similar manner.

The following is a table of duplicate op-codes. The instructions are grouped into equivalence classes. Note that the grouping is slightly different on the new machine than it was for the PDP10. (see also section 6.0: the opcode index).

op-code	Both	PDP10 only	New machine only
300	CAI		BLN
	TRN		BRN
	TLN		DSKP
	TDN		TSKP
	TSN		QSKP
	JUMP		HSKP
	CAM		
	SETA		
	SETAI		
	SETMM		
201	MOVEI	HRRZI(1)	
	SETMI	MOVMI(2)	
205		MOVSI(3)	
		HRLZI(3)	
304	CAIA		BLNA
	TRNA		BRNA
	TLNA		DSKPA
	TDNA		TSKPA
	TSNA		QSKPA

	CAMA	HSKPA
201(4)	SETZ SETZI	HLLZI(5) HLRZI(5) HLLEI(5) HLREI(5)
N/A(6)	SETO SETOI	
N/A(7)	SETCA SETCAI	
434	OR TDO	
660	TRO ORI	
202	MOVEM SETAM SETAB	
200	MOVE SETM SETMB	
203	MOVES SKIP HLLS HRRS	
430	XOR TDC	
640	TRC XORI	
630	TDZ ANDCM	
620	TRZ ANDCMI	

Footnotes:

1. On a PDP10, HRRZI is equivalent to MOVEI. This is not true on the new machine: a MOVEI instruction doesn't necessarily set the LH of AC to zero.

2. On the PDP10, MOVMI is equivalent to MOVEI. On the new machine, however, the sign bit of the effective address isn't necessarily zero and therefore the instructions are not equivalent.

3. On a PDP10, HRLZI is equivalent to MOVSI. This is not true on the new machine: a MOVSI instruction doesn't necessarily set the RH of AC to zero.

4. (see section 1.11.10).

5. On a PDP10, HLLZI, HLRZI, HLLEI, and HLREI are all equivalent to SETZ. On the new machine, however, none of these instructions is equivalent to any of the others.

6. Not applicable (see section 1.11.11).

7. Not applicable (see section 1.11.17).

1.8.2 New Names -

The AOBJN instruction will be known by a new name: AOBJL. There will be no change in functionality, only a change in name. For compatibility, the assembler will recognize both mneumonics, but AOBJL is the preferred name.

Likewise AOBJP will be known as AOBJGE.

The ANDCB group will be known as NOR.

The ORCB group will be known as NAND.

FLTR will be known as FLT.

1.8.3 I/O Instructions -

The PDP10 reserves op-codes 700-777 for the I/O instructions. These instructions will not exist on the new machine. Instead, the new machine will perform I/O in a fashion similar to the PDP11. Each of the registers in each of the I/O controllers will be directly addressable by referencing the correct location in physical memory. The operating system will be free to map these physical addresses to any virtual addresses it may desire. No doubt the operating system will choose a bank of virtual addresses in the lowest 2**18 of memory so that they can be referenced without a link word.

It is not our purpose here to define the formats of any of the I/O registers. It's likely, however, that they would be VAX compatible.

Note that the new machine doesn't have a "User I/O" bit. If the operating system wishes to allow a user to perform I/O operations it need merely map the I/O registers into the user's virtual address space.

Any memory reference instruction can be used to manipulate the I/O registers. Of particular interest, however, are the instructions B{L,R}{N,O,Z,C}{-,N,E,A}

1.8.4 EOP -

Op-code 400 is known as the "EOP" instruction. The effective address of this instruction is treated as an extension of the op-code. The effective address specifies which subfunction is to be performed. We can therefore support a large number of functions while only using a single op-code. Each of these functions has a single operand: an AC.

1.8.5 ACOP -

Op-codes 254-256 are known as "ACOP's". In these instructions the AC field is treated as an extension of the op-code. The AC field specifies which subfunction will be performed. We can therefore support a large number of functions while only using a small number of op-codes. Each of these functions has a single operand: E.

On the PDP10 there are numerous instructions for which the AC field is ignored (e.g. JUMPA, SETCMM, etc). The new machine will continue to support these instructions, but the AC field will no longer be ignored. These instructions will be implemented as ACOP functions. The programmer need not be aware of this fact. The syntax of his source code will not change. JSR, for example, will merely be OPDEFed to "JRST 5,".

1.8.6 Priveledged Instructions -

On the PDP10 op-codes 31-37 are LUUO's. On the new machine, however, they are LUUO's only in user mode. In exec mode they are something quite different.

On the PDP10 there are several op-codes which are legal only in exec mode (e.g. PXCT, MAP, etc). On the new machine these instructions have been moved to 31-37.

Note that one of these instructions (op-code 31) is an ACOP type instruction (the AC field is an extension of the op-code).

1.9 Stack Pointers

The machine supports two types of stack pointers: (types 0 and 1)

1.9.1 Type 0 -

```
!0!0                                                          3!
!0!1                                                          7!
+-+-----------------------------------------------------------+
!0!                          addr                             !
+-+-----------------------------------------------------------+
```

It is anticipated that type 0 stack pointers will be the most popular.

Type zero is indicated by a zero in bit 0. Bits 1-37 contain the address of the current word on the stack. A PDL overflow can occur only if the address is incremented above 2**31-1. The stack must therefore be placed at the extreme top of the virtual address space.

If a second type 0 stack is desired, the program can protect against overflow by making inaccessable the pages at either end of the stack.

1.9.2 Type 1 -

```
!0!0                          1!2                             3!
!0!1                          7!0                             7!
+-+----------------------------+------------------------------+
!1!           -N               !            addr              !
+-+----------------------------+------------------------------+
```

Type 1 is indicated by a 1 in bit 0. Bits 1-17 contain a count of the number of words that remain after the current word. This count is expressed as a negative number. Thus the entire left half (bits 0-17) contains the two's complement of the count. Bits 20-37 contain the address of the current word on the stack. This address must fall below 2**16. The stack may not be placed at a higher address.

Type 1 stack pointers are intended only for PDP10 compatibility.

1.10 New Instructions

The new machine supports a wide variety of instructions that did not exist on the PDP10. The following is a list of miscellaneous new instructions.

1.10.1 MOVEIA - MOVE Immediate And Always Skip (op-code 310) -

This instruction is just like MOVEI except that the instruction skips.

1.10.2 PUSHI - PUSH Immediate (op-code 314) -

The PUSHI instruction is similar to the PUSH instruction except that E itself is pushed instead of C(E).

1.10.3 T{R,L,D}{U,NU} -

Test and skip if Unanimous.

The AC is compared against a mask. For each of the bits which is one in the mask, the corresponding bit in the AC must also be a one. The test must be unanimous. The selected bits in AC must all be ones.

For each of the bits which is zero in the mask, the corresponding bit in the AC is ignored.

Legend:

R - Right - The mask is E.
L - Left - The mask is a copy of E with the LH and RH swapped.
D - Direct - The mask is C(E).

U - Skip if Unanimous
NU - Skip if Not Unanimous.

Note that

```
T?U     T1,foo
```

Is equivalent to:

```
T?C     T1,foo
T?CE    T1,foo
```

Note that

```
T?NU    T1,foo
```

Is equivalent to:

```
T?C     T1,foo
T?CN    T1,foo
```

Note the difference between T?NN and T?U. T?NN will skip if any of the selected bits is one. T?U will only skip if all of the selected bits are one.

The instruction TRU T1,-1 is equivalent to CAIE T1,-1. Both skip if T1 is all ones.

Mneumonic	Op-code
TRU	600
TRNU	604

TLU	601
TLNU	605
TDU	610
TDNU	614

1.10.4 B{L,R}{Z,O,N,C}{N,E,-,A} -

Each of these instructions manipulates a single bit in the word addressed by E. Bits 11-14 of the instruction are not interpreted as an AC number. Instead they are interpreted as a bit number.

Legend:

L - Test the bit in the LH (i.e. AC=0 means bit 0).
R - Test the bit in the RH (i.e. AC=0 means bit 20).

Z - Zero the bit.
O - Set the bit to one.
N - Don't change the bit.
C - Complement the bit.

N - Skip if the bit was originally Non-zero.
E - Skip if the bit was originally zero.
A - Always skip.
blank - Never skip.

Example: The instruction "BLO 3,FOO" will set bit 3 in location FOO. The instruction "BRNE 3,FOO" will skip if bit 23 in location FOO is zero. The instruction "BLCN 0,FOO" will complement the sign bit of location FOO and skip if the sign bit was originally on.

Note that on a 32 bit machine it takes 5 bits to express a bit number. The AC field, however, is only 4 bits wide. To get around this problem, the class "L" instructions have all been assigned even op-codes. The class "R" instructions have all been assigned odd op-codes. Thus bits 10-14 of the instruction give the actual bit number (all five bits). Thus the source statement "BLNN 23,T1" will cause the assembler to generate the code "BRNN 3,T1".

A major use of this instruction group is to manipulate bits in the I/O page.

Mneumonic	Op-code
BLN	300
BRN	300
BLNE	702
BRNE	703
BLNA	304
BRNA	304
BLNN	706
BRNN	707
BLZ	710
BRZ	711

BLZE	712
BRZE	713
BLZA	714
BRZA	715
BLZN	716
BRZN	717
BLC	720
BRC	721
BLCE	722
BRCE	723
BLCA	724
BRCA	725
BLCN	726
BRCN	727
BLO	730
BRO	731
BLOE	732
BROE	733
BLOA	734
BROA	735
BLON	736
BRON	737

1.10.5 SSTEP - Single STEP (op-code 256-1) -

This instruction is intended for the soul purpose of implementing $X in DDT.

The C(E) is taken as the address of an instruction to be executed. Note the difference between XCT and SSTEP. In XCT, C(E) is the instruction itself. In SSTEP, C(E) is the address of the instruction.

SSTEP is unlike XCT in another important respect. In SSTEP, if the instruction being executed attempts to alter the PC, then the PC isn't actually changed. Instead, C(E) is updated. If a normal (non-skip) instruction is executed, then C(E) will be incremented once (as the PC normally is). A skip instruction will increment C(E) by two. A jump instruction (JRST, JUMP??, etc) stores the effective address of the JRST. A subroutine call (e.g. PUSHJ) places the address of the PUSHJ plus one on the stack and overwrites C(E) with the address of the subroutine.

Example:

```
BAR:        SSTEP   MYPC
            ...
MYPC:       FOO
            ...
FOO:        JSP     T1,GOO
            ...
GOO:        ...
```

Location MYPC is overwritten with the address of GOO, T1 is set to FOO+1, and the next instruction is taken from BAR+1.

The instruction is implemented by the microcode with little or no support from the hardware. The microcode stores E and the original PC in internal registers. It then moves C(E) into the PC and executes a normal instruction cycle. Upon instruction exit, the modified PC is stored at location E and the original PC is restored.

Note that the SSTEP instruction interprets C(E) as an indirect word (the sign bit is not ignored). Thus C(E) may be coded as a position independent pointer. Note, however, that the sign bit of C(E) is always set to zero when the modified PC is stored.

The modified PC is stored at the address specified by the original E. If C(E) is an indirect word that points to a second indirect word, it's the original E that determines where the modified PC is stored.

If one SSTEP attempts to execute a second SSTEP then the chain is aborted and C(E) is set to -1. An SSTEP may execute an XCT, and an XCT may execute a SSTEP.

Note that if the target instruction page faults, the page fault PC is that of the SSTEP instruction and not that of the target instruction.

1.10.6 EA - Effective Address (op-code 123) -

The effective address of the EA instruction specifies the address of a second instruction. Fetch the second instruction and compute its effective address. Place the result in AC.

If the EA instruction had existed on the PDP10, then

```
        EA      T1,FOO          ;CASE ONE
```

would have been equivalent to:

```
        MOVEI   T1,@FOO         ;CASE TWO
```

On the new machine, however, the two are not equivalent. Example: Consider the instruction:

```
FOO:    MOVE    T2,47
```

Case one (above) puts a 47 into T1. However case two puts "MOVE T2,47" into T1 (note that the sign bit of location FOO is zero, the op-code is 200).

Think of it this way: The new machine has two types of effective addresses: The 19 bit format and the 32 bit format. The former variety appears as the low order 19 bits of every instruction (see section 1.3). The 32 bit format is used for indirect words. The purpose of the EA instruction is to specify an indirect word that uses the 19 bit format instead of the 32 bit format.

The EA instruction is used heavily by DDT.

1.10.7 BUS{-,I,M,B} - Backward SUBtract. -

The BUS instruction is similar to the SUB instruction except that the order of the operands is reversed. SUB computes AC-C(E), whereas BUS computes C(E)-AC.

Mneumonic	Op-code	What
BUS	140	AC=C(E)-AC
BUSI	141	AC=E-AC
BUSM	142	C(E)=C(E)-AC
BUSB	143	AC=C(E)-AC

1.10.8 IVID{-,I,M,B} - Backward Integer DIVide. -

The IVID instruction is similar to the IDIV instruction except that the order of the operands is reversed. IDIV computes AC/C(E), whereas IVID computes C(E)/AC.

Mneumonic	Op-code	What
IVID	150	AC=C(E)/AC (no remainder)
IVIDI	151	AC=E/AC
IVIDM	152	C(E)=C(E)/AC
IVIDB	153	C(E)=AC=C(E)/AC

1.10.9 IDV[I] -

The IDV instruction is similar to the IDIV instruction except that IDV does not return a remainder. AC+1 is unchanged.

Mneumonic	Op-code
IDV	100
IDVI	101
IDVM	232

Note: The assembler will recognize the mneumonic "IDVM" and map it equal to "IDIVM". Neither returns a remainder.

1.10.10 UMAP - User MAP (op-code 032) -

This instruction is similar to MAP except that the user page map is used instead of the exec page map.

All indirect words and index registers specified by the effective address calculation are fetched from exec virtual space.

This instruction is privileged. It's legal only from exec mode. If executed from user mode, it's an LUUO.

1.10.11 XJSR - EXtended Jump To Subroutine (opcode 031-0) -

```
C(E)=FLAGS
C(E+1)=PC+1
PC=E+2
```

1.10.12 XRET - EXtended RETurn From Subroutine (opcode 031-1) -

```
FLAGS=C(E)
PC=C(E+1)
```

1.10.13 XDIS - DISMISS Interrupt (opcode 031-3) -

Same as XRET except dismiss the current interrupt (if any).

1.10.14 XPCW - Exchange PCW (opcode 031-2) -

```
C(E)=FLAGS
C(E+1)=PC+1
FLAGS=C(E+2)
PC=C(E+3)
```

1.10.15 BBLT - Backward BLT (opcode 256-4) -

The effective address gives the location of a three word argument block:

```
E+0/        FF
 +1/        FT
 +2/        LT
```

Words FF through FF+(LT-FT)-1 are moved to FT through LT respectively.

BBLT is just like BLT except that FF+(LT-FT)-1 is the first word transfered instead of FF. (see section 1.11.3).

1.10.16 EOP -

Op-code 400 is known as the "EOP" instruction. The effective address of this instruction is treated as an extension of the op-code. The effective address specifies which subfunction is to be performed. We can therefore support a large number of functions while only using a single op-code. Each of these functions has a single operand: an AC.

The effective address is decoded as follows:

```
!1        1!2       2!2                2!3        3!3        3!
!6        7!0       1!2                7!0        3!4        7!
+----------+---------+-----------------+----------+----------+
!          ! 01=PSAV !        NL       !    FR    !    LR    !
!          +---------+-----------------+----------+----------+
!    00    !   00    !  group number   !  function number    !
!          +---------+-----------------+---------------------+
!          !   10    !              reserved                 !
!          !   11    !                                       !
+----------+---------+---------------------------------------+
! 01=SAVE  !                                                 !
! 10=REST  !                    bit mask                     !
! 11=PSAVE !                                                 !
+----------+-------------------------------------------------+
```

The vast majority of all EOP functions have zeros in bits 16-21. These functions are divided into groups. Bits 22-27 give the group number. Bits 30-37 denote the function number within that group.

1.10.16.1 SAV - SAVe AC's (on The Stack) -

The SAV instruction is one of the many functions of EOP. It is represented by an octal 0075 in bits 16-27 of the effective address. Bits 30-37 of the effective address are decoded as follows:

```
!3      3!3      3!
!0      3!4      7!
+-------+-------+
!  FR   !  LR   !
+-------+-------+
```

The register denoted by FR is pushed onto the stack. Register FR+1 is then pushed. Then FR+2, ..., etc. The process stops when the register denoted by LR is pushed onto the stack.

Note that for the purposes of this instruction register 17 is said to be followed by register 0. Thus if FR=16 and LR=1, four registers will be pushed: 16, 17, 0, and 1 (in that order).

As with all stack instructions, the stack pointer is taken from the register denoted by bits 11-14 of the instruction (not bits 11-14 of the effective address).

Whether it be a MACRO or not, the assembler will recognize the syntax:

```
        SAV     P,FR,LR
```

1.10.16.2 RST - ReSTore AC's (from The Stack) -

The RST instruction is an EOP function. It is represented by an octal 0074 in bits 16-27 of the effective address. Bits 30-37 of the effective address are decoded as follows:

```
!3      3!3      3!
!0      3!4      7!
+-------+--------+
!  FR   !  LR    !
+-------+--------+
```

The instruction is the inverse operation from SAV. The registers are popped back off the stack. Note that since LR was the last register pushed, it is now the first register popped.

The assembler will recognize the syntax:

```
RST     P,FR,LR
```

1.10.16.3 PSAV - Popping SAVe -

The PSAV instruction is an EOP function. The effective address is decoded as follows:

```
!1      2!2           2!3     3!3     3!
!6      1!2           7!0     3!4     7!
+-------+-----------+-------+-------+
!0 0 0 1!     NL      !  FR   !  LR   !
+-------+-----------+-------+-------+
```

Registers FR through LR are pushed onto the stack as they would be by the SAV instruction. The stack pointer is then adjusted by the quantity +NL (see ADJSP). The sign bit is then lit in the effective address register and this modified value is pushed onto the stack.

The assembler will recognize the syntax:

```
PSAV    P,FR,LR[,NL]
```

Example: We are all familiar with the PDP10 subroutine SAVE4.

```
PUSHJ   P,SAVE4
```

is equivalent to:

```
PSAV    P,P1,P4
```

Note that the POPJ instruction will automatically undo the effects of the PSAV instruction (see section 1.11.4: POPJ).

Note that the purpose of the NL field is to allocate space on the stack for the storage of local variables.

1.10.16.4 SAVE - SAVE AC's (on The Stack) -

This instruction is an EOP. The effective address is decoded as follows:

```
!1 1!2                            3!
!6 7!0                            7!
+---+-------------------------------+
!0 1!          Bit mask             !
+---+-------------------------------+
```

Bits 20-37 are a bit mask which indicates what AC's are to be pushed onto the stack. Bit 20 in the mask corresponds to AC 0, Bit 37 to AC 17, etc.

Note: This instruction is significantly slower than the SAV instruction. Whenever possible SAV should be used instead of SAVE.

The assembler will recognize the syntax:

```
        SAVE    P,a,b,c,...
```

Note that AC 0 is the first to be pushed and AC 17 is the last. Thus:

```
        SAVE    P,P1,P2,P3,P4
```

is equivalent to:

```
        SAV     P,P1,P4
```

1.10.16.5 REST - RESTore AC's (from The Stack) -

This instruction is an EOP. The effective address is decoded as follows:

```
!1 1!2                              3!
!6 7!0                              7!
+---+-------------------------------+
!1 0!           Bit mask            !
+---+-------------------------------+
```

This instruction is the inverse operation of SAVE. The registers are popped back off the stack.

1.10.16.6 PSAVE - Popping SAVE -

This instruction is an EOP. The effective address is decoded as follows:

```
!1 1!2                             3!
!6 7!0                             7!
+---+--------------------------------+
!1 1!           Bit mask             !
+---+--------------------------------+
```

The registers specified by the bit mask are pushed onto the stack exactly as they would be by the SAVE instruction. The sign bit is then lit in the effective address register and this modified value is then pushed onto the stack.

Note that the POPJ instruction will automatically undo the effects of the PSAVE instruction (see section 1.11.4: POPJ).

1.11 Modifications To Existing Instructions

Most of the instructions on the new machine function exactly the same as the equivalent instruction on the PDP10. There are a few, however, that function slightly different. The following is a list of these differences.

1.11.1 SETOM -

On the PDP10, the AC field was ignored. Not so on the new machine. If the AC field is zero, the instruction behaves as it did on the PDP10. If the AC is non-zero, however, then AC gets a copy of C(E) as it was before being set to ones. The manipulation of C(E) is performed as an uninterruptable read pause write. This will be useful for the implementation of interlocks.

1.11.2 SETZM -

(See SETOM) Iff the AC is non-zero then AC gets a copy of the original C(E).

1.11.3 BLT -

BLT is totally different than it was on the PDP10. BLT is now function 3 of op-code 256. E gives the address of a three word argument block:

```
E+0/    FF      ;FIRST FROM
E+1/    FT      ;FIRST TO
E+2/    LT      ;LAST TO
```

Locations FF through FF+(LT-FT)-1 are copied to FT through LT. Note that location FF is the first moved.

Each of the three words in the argument block is interpreted as an indirect word. If relative addressing is specified then each is relative to a different address. FF is relative to E+0. LT is relative to E+2.

See also BBLT (section 1.10.15).

1.11.4 POPJ -

The instruction begins by POPing the top word off the stack. The microcode places it in an internal register called X.

FOO:

If the sign bit of X is zero, add the effective address of the POPJ instruction to X. Branch to the resulting address (i.e. copy the modified value of X to the PC).

If, however, the sign bit of X is one, decode the rest of the word as follows:

```
!0!0                          1!1!2 2!2         2!3     3!3     3!
!0!1                          6!7!0 1!2         7!0     3!4     7!
+-+---------------------------+-+---+-----------+-------+-------+
! !                           !0!   !    NL     !  FR   !  LR   !
!1!          Ignored          +-+---+-----------+-------+-------+
! !                           !1!          Bit Mask             !
+-+---------------------------+-+-------------------------------+
```

Bit 17 tells us whether a PSAV or a PSAVE has been done (see section 1.10.16). In either case, the process is reversed. A PRST or PREST is simulated. We then pop the next word off the stack and place it in register X. Note that the effective address register is still unchanged. It still contains the effective address from the POPJ instruction. GOTO FOO.

Note: On the PDP10, E was ignored. On the new machine, however, it is used to implement skip returns. Example:

```
        AOS     (P)
        POPJ    P,
```

Is roughly equivalent to:

```
        POPJ    P,1
```

Note that the later does not actually modify any location on the stack.

1.11.5 ADJSP -

The ADJSP instruction is the same as on the PDP10 except that E is taken as a 32 bit signed integer instead of an 18 bit signed integer.

1.11.6 Doubleword Integers -

The format for a double precision integer is different than that on the PDP10. On the PDP10, bit 0 of the second word is not a significant bit. It is a copy of bit 0 of the first word (the sign bit). On the new machine, however, bit 0 of the second word is a significant bit.

Instructions affected: DADD, DSUB, DMUL, DDIV, DMOVN, DMOVNM, MUL{-,I,M,B}, DIV{-,I,M,B}.

1.11.7 JSR, JSP, PUSHJ, And POPJ -

These instructions no longer save the flags. The PC is taken as a full 32 bit quantity.

1.11.8 JSA -

```
C(E)=C(AC)
C(E+1)=C(AC+1)
C(AC)=PC+1
C(AC+1)=E
PC=E+2
```

1.11.9 JRA -

```
PC=C(AC)
C(AC)=C(C(AC+1))
C(AC+1)=C(C(AC+1)+1)
```

1.11.10 SETZ -

On the PDP10, the effective address of SETZ is ignored (but most programmers leave it zero). On the new machine, however, it is required that E be zero. The assembler, in fact, maps the mneumonic SETZ to the same op-code as MOVEI. "SETZ T1," is mapped to "MOVEI T1,0"

1.11.11 SETO -

The assembler maps the mneumonic SETO into "MOVEI -1".

1.11.12 XCT -

On the PDP10, an AC field of zero meant XCT. A non-zero AC meant PXCT. The new machine doesn't have a previous context execute. Use ULDB and UDPB instead. The XCT instruction is now function 0 of ACOP 256.

1.11.13 MAP -

This instruction has changed op-code from 257 to 033. It is still legal only from exec. If executed from user mode it used to be trapped as an MUUO. It's now an LUUO instead.

1.11.14 JUMPA -

On the PDP10, the AC field of JUMPA is ignored. Not so on the new machine. The mneumonic JUMPA now maps to the same op-code as JRST. The AC field of JRST is not ignored.

On the PDP10 JUMPA is used by DDT for inserting patchs. On the new machine DDT should use JRST instead.

1.11.15 JSR -

On the PDP10 the AC field was ignored. On the new machine JSR is "JRST 5,". Note that JSR saves a full word PC. It does not store flags.

1.11.16 SETCMM -

On the PDP10 the AC field was ignored. On the new machine SETCMM is function 5 of ACOP 256.

1.11.17 SETCA -

On the PDP10 the effective address was ignored. On the new machine the assembler maps the mneumonic "SETCA" to "XORI -1".

1.11.18 JFCL -

The JFCL instruction is not supported on the new machine. The user should, instead, perform a bit test of location .JBTRP (see section 1.17).

Note that the assembler will continue to recognize the mneumonic JFCL. It will be mapped, however, to the same op-code as CAI.

1.11.19 T{R,L,D,S}{N,Z,O,C}{-,A,N,E} -

TR?? is almost like TD?? except that TD?? deals with C(E) whereas TR?? deals with E itself. Both deal in full 32 bit quantities. TR?? does not ignore the left half of E.

TL?? is almost like TS??. TS?? deals with a copy of C(E) that has had its halves swapped. TL?? deals with a copy of E that has had its halves swapped. Both deal in full 32 bit quantities. TL?? does not ignore the left half of E.

1.12 Byte Pointers

1.12.1 Format -

The hardware has provisions for eight different types of byte pointers. Bits 0-2 of the BP indicate the type. At present, only two types are defined (type 4 and type 5). An attempt to use any of the remaining types will cause trap 7 (illegal operand).

```
!0   0!0       0!1       1!1!1                                  3!
!0   2!3       7!0       4!5!6                                  7!
+-----+---------+---------+-+-----------------------------------+
!  4  !    S    !    P    !0!                Z                  !
+-----+---------+---------+-+-----------------------------------+


!0   0!0       0!1       1!1!1!1      2!2                       3!
!0   2!3       7!0       4!5!6!7      2!3                       7!
+-----+---------+---------+-+-+-------+-------------------------+
!  4  !    S    !    P    !1!I!   X   !           Y             !
+-----+---------+---------+-+-+-------+-------------------------+


!0   0!0       0!1       1!1                                    3!
!0   2!3       7!0       4!5                                    7!
+-----+---------+---------+-------------------------------------+
!  5  !    S    !    P    !                 C                   !
+-----+---------+---------+-------------------------------------+
!                         INDIRECT WORD                         !
+---------------------------------------------------------------+
```

1.12.1.1

In a type 4 byte pointer, bits 15-37 specify the effective address. The format of these bits is exactly the same as that of the basic instruction format. If relative addressing is used, the base address is that of the byte pointer, not the PC.

In a type 5 byte pointer, the second word specifies the effective address. The format of this word is exactly the same as that of an indirect word. Note that if relative addressing is used, the base address is that of the second word, not the first word.

1.12.1.2

As on the PDP10, the S field indicates the number of bits per byte. An S field of zero, however, indicates a 32 bit byte (a fullword).

1.12.1.3

The P field indicates the number of bits to the right of the target byte. This is exactly the same as the P field on the PDP10. Note, however, that unlike the PDP10, there is no way to specify the ficticous byte just to the left of a word (on the PDP10, programmers would set P to ^D36). You must instead specify the last byte of the previous word.

1.12.1.4

In a type 5 byte pointer, the C field indicates a count of the number of bytes remaining after the current byte. This field is used by the instructions {ILDB,IDPB,LDB,DPB,IBP}{W,L} (see section 1.12.2.1).

1.12.1.5

The assembler will recognize two pseudo ops: POINT and POINTR.

POINT will build a type 4 byte pointer. The format is identical to that on the PDP10: POINT S,E,N (where N=37-P).

POINTR will build a type 5 byte pointer. The format is: POINTR S,E,N[,C]. If the C field is omitted it defaults to 2**18-1 (which is the largest positive number that will fit in the C field).

1.12.1.6

Consider one of the subtle differences between a type 4 and type 5 byte pointer.

```
BP4:    POINT   40,@.+1,37
        Z       E

BP5:    POINTR  40,E,37
```

If an ILDB instruction is executed, the result(s) would be:

```
BP4:    POINT   40,@.+2,37
        Z       E

BP5:    POINTR  40,E+1,37,2**18-2
```

The resulting BP4 is probably not what the programmer intended.

1.12.2 Byte Instructions -

The new machine supports all of the PDP10's byte instructions. In addition, it supports a wide variety of new ones.

1.12.2.1 {ILDB,IDPB,IBP,LDB,DPB}{-,A,W,L} -

The last character of each mneumonic indicates under what circumstances the instruction will skip:

W - skip if Win.

Skip if the resulting C field is greater than or equal to zero (i.e. skip if the string is not yet exhausted).

L - skip if Loose

Skip if the resulting C field is less than zero (i.e. skip if the string is exhausted).

A - Always skip

blank - never skip

(same as the equivalent instruction on the PDP10).

The instructions {ILDB,IDPB}{W,L} begin by incrementing the byte pointer. In doing so, they decrement the C field. The instructions then take one of two possible actions depending on the sign bit of the resulting C field (bit 15). If the sign bit is 0, the target byte is loaded/stored. If the sign bit is 1, the target byte is not referenced (ILDB{W,L} does not alter the AC).

The instructions {ILDB,IDPB}{-,A} decrement the C field but ignore the result. The target byte is always loaded/stored.

The instructions {LDB,DPB}{W,L} take one of two possible actions depending on the sign bit of the C field (they do not modify the C field). If the sign bit is 0, the target byte is loaded/stored. If the sign bit is 1, the target byte is not referenced (LDB{W,L} does not alter the AC).

Note: If the byte pointer is a type 4 byte pointer (no C field) then we assume an infinite supply of bytes. The "W" class instruction will always skip. The "L" class instruction will never skip.

Example: Consider the following well known PDP10 subroutine:

```
CI:     SOSGE   IBUF+.BFCTR
         JRST   CI2
        ILDB    T1,IBUF+.BFPTR
```

```
         AOS      (P)
         POPJ     P,
CI2:     ...
```

This could be coded as:

```
CI:      ILDBL    T1,IBUF+.BFPTR
          POPJ    P,1
         ...
```

Mneumonic	Op-code	
ILDB	134	(same as PDP10)
ILDBA	257	(new)
ILDBW	264	(new)
ILDBL	324	(new)
IDPB	136	(same as PDP10)
IDPBA	330	(new)
IDPBW	417	(new)
IDPBL	435	(new)
IBP	133	(same as PDP10)
IBPA	256-10	(new)
IBPW	256-11	(new)
IBPL	256-12	(new)
LDB	135	(same as PDP10)
LDBA	543	(new)
LDBW	611	(new)
LDBL	615	(new)
DPB	137	(same as PDP10)
DPBA	650	(new)
DPBW	670	(new)
DPBL	740	(new)

1.12.2.2 DPBI - DePosit Byte Immediate (op-code 251) -

This instruction is similar to DPB except that AC is deposited instead of C(AC). The AC field (bits 11-14 of the instruction) is taken as a 4 bit number. This number is deposited as specified by the byte pointer. If S is greater than 4, the unused bits are zeroed. The 4 bit wide number is not sign extended.

Example: The instruction:

```
DPBI    ^D8,[POINT 5,FOO,7]
```

Will set bits 3-7 of location FOO to eight (this could be particularly handy if FOO is a byte pointer).

1.12.2.3 LDBX - LoaD Byte EXtended (op-code 104) -

This instruction is similar to LDB except that the byte is sign extended.

Example: Given the byte pointer:

```
FOO:        POINT   3,[47],37
```

The instruction "LDB T1,FOO" will set T1 to 7. The instruction "LDBX T1,FOO" will set T1 to -1.

1.12.2.4 B{AOS,SOS}{-,L,E,LE,A,GE,N,G} -

The AC field is treated as an extension to the op-code. Thus all 16 functions share the same code (number 255).

E gives the address of a byte pointer. The byte is incremented or decremented. Example:

```
        BAOS??  E
```

Is equivalent to:

```
        LDBX    T1,E
        AOS     T1
        DPB     T1,E
        SKIP??  T1
```

Mneumonic	AC Field
BAOS	0
BAOSL	1
BAOSE	2
BAOSLE	3
BAOSA	4
BAOSGE	5
BAOSN	6
BAOSG	7
BSOS	10
BSOSL	11
BSOSE	12
BSOSLE	13
BSOSA	14
BSOSGE	15
BSOSN	16
BSOSG	17

1.12.3 ADJBP -

As on the PDP10, ADJBP is the same op-code as IBP (code 133). If the AC field is zero, the instruction is interpreted as IBP. IF the AC field is non-zero, the instruction is interpreted as ADJBP. The contents of that AC is taken as a signed byte count. The byte pointer is advanced forward or backward by that number of bytes. Unlike the PDP10, the AC is unchanged. The new machine modifies the byte pointer in the location specified by E (including the C field). To determine if the C field has expired use the instruction "BL? 15,E".

Note that ADJBP preserves byte alignment but IBP does not.

1.12.3.1 DECBP - DECrement Byte Pointer (op-code 256-6) -

DECBP is the inverse of IBP.

DECBP does not preserve byte alignment. If you wish to preserve byte alignment use ADJBP instead.

1.12.3.2 U{LDB,DPB} -

ULDB - User LDB (op-code 036)
UDPB - User DPB (op-code 037)

The new machine does not have a PXCT instruction. PXCT has been replaced by ULDB and UDPB. These instructions are just like LDB and DPB except that the data is taken from user space instead of exec space. Note that the byte pointer itself is in exec space. The effective address calculation specified by the byte pointer is carried out entirely in exec space. The only thing that comes from user space is the actual data.

Note that if the target address is in the range 0 to 17 then the user AC set is used. Thus the effective address calculation uses the exec AC set but the target data comes from the user AC set. Note that by loading the FLAGS register the monitor can select which AC set to use as the exec set and which to use as the user set.

This facility is not as extensive as PXCT, but it is perfectly satisfactory for 99% of the cases. If the monitor wants to do something more extensive (like BLT) it can always map the user's page in the exec map.

Consider the case where the argument block to a UUO has a pointer to a second argument block. The user would like this pointer to be in position independent format. PXCT would be useful to compute the ultimate effective address. Without PXCT, however, we suggest the monitor contain a subroutine to simulate effective address calculation. The subroutine will execute quickly as most effective address calculations are simple ones.

These instructions are legal only from exec mode. If executed from user mode they are LUUOs.

Note that it is indeed possible for these instructions to get a proprietary violation (see section 1.15). The instruction will trap if the reference is not legal for the segment which issued the current UUO.

1.12.3.3 PHY{LDB,DPB} -

PHYLDB - PHYsical LDB (op-code 034)
PHYDPB - PHYsical DPB (op-code 035)

These instructions are similar to LDB and DPB except that the data is taken from physical address space instead of virtual address space. Note that the byte pointer itself is in virtual address space. The effective address calculation specified by the byte pointer is carried out entirely in the virtual address space. The only thing that comes from physical space is the actual data byte.

Note that if the target address is in the range 0 to 17 then physical locations 0-17 are referenced (not the exec AC set).

These instructions are priveledged. They can be executed only from exec mode. If executed from user mode they are LUUOs.

1.13 Floating Point

The machine supports three types of floating point numbers: Single, Double, and triple.

1.13.1 Single -

The format of a single precision floating point number is exactly the same as that on the PDP10 except that there are four less bits of precision. Therefore:

Bits	
0	Sign bit
1-10	(8 bits) Exponent (excess 200)
11-37	(23 bits) Fraction

Negative numbers are expressed as the two's complement of the entire word.

1.13.2 Double -

The double precision format is exactly like the single precision format except that it has an extra 32 bits of precision (one word). Unlike the PDP10, bit zero of the second word is not ignored. It is a meaningful data bit just like any other.

Bits	
0	Sign bit
1-10	(8 bits) Exponent
11-77	(55 bits) Fraction

1.13.3 Triple -

The triple precision format has 25 additional bits of precision and 7 additional bits of exponent:

Bits	
0	Sign bit
1-17	(15 bits) Exponent
20-137	(80 bits) Fraction

1.13.4 Obsolete -

The machine does not support unrounded single precision floating point. The op-codes for F{AD,SB,MP,DV}{-,M,B} are obsolete and have been recycled. The assembler, however, still recognizes these mneumonics and maps them into the equivalent rounded instruction (F{AD,SB,MP,DV}R{-,M,B} respectively).

The old style KA10 double precision floating point format is no longer supported. The instructions DFN, UFA, FADL, FSBL, FMPL, and DFVL (op-codes 131, 130, 141, 151, 161, and 171 respectively) are obsolete. The op-codes have been recycled.

1.13.5 Immediate -

The instructions F{AD,SB,MP,DV}[R]I function slightly different than they did on the PDP10. On the PDP10, the halves of E were swapped before use. On the new machine, E is shifted left ^D14 bits before use. This will allow for the greatest utilization of the 18 significant bits of E (1 sign bit, 8 bits of exponent, and 9 bits of fraction).

1.13.6 Op-code Assignment -

There are 16 arithmetic instructions for single precision numbers: F{AD,SB,MP,DV}{-,I,M,B}; 4 instructions for double precision: DF{AD,SB,MP,DV}; and 4 instructions for triple precision: TF{AD,SB,MP,DV}.

Opcode	Mneumonic
144	FAD
145	FADI
146	FADM
147	FADB
154	FSB
155	FSBI
156	FSBM
157	FSBB
164	FMP
165	FMPI
166	FMPM
167	FMPB
174	FDV
175	FDVI
176	FDVM
177	FDVB
110	DFAD
111	DFSB
112	DFMP
113	DFDV
102	TFAD
103	TFSB
106	TFMP
107	TFDV

1.13.7 Complex Numbers -

All three types of floating point are supported for complex numbers. Complex integers are not supported. In core, complex numbers are represented by an ordered pair in which the real part is stored first and the imaginary part is stored second. In single precision format the real part is in word 0, and the imaginary part is in word 1. In double precision format the real part is in words 0-1, and the imaginary part is in words 2-3. In triple precision format the real part is in words 0-2, and the imaginary part is in words 3-5.

Once you've learned the mneumonics for the floating point instructions the complex instructions are easy: merely replace the "F" with a "C". Thus the mneumonics for complex arithmetic are: {-,D,T}C{AD,SB,MP,DV}.

```
(E+FI) = (A+BI) op (C+DI)

OP
---
AD - Add: E=A+C F=B+D
SB - Subtract: E=A-C F=B-D
MP - Multiply: E=AC-BD F=BC+AD
DV - Divide: E=(AC+BD)/(CC+DD) F=(BC-AD)/(CC+DD)
```

OPCODE	MNEUMONIC
746	CAD
747	CSB
750	CMP
751	CDV
752	DCAD
753	DCSB
754	DCMP
755	DCDV
756	TCAD
757	TCSB
760	TCMP
761	TCDV

1.13.8 New Instructions (Floating And Complex) -

The following is a list of instructions which are used in connection with floating point:

1.13.8.1 FMOVEI - Floating MOVE Immediate (op-code 401) -

E is shifted 14 bits to the left and the result is placed in the AC.

1.13.8.2 {T,Q,H}MOVE[M] -

TMOVE - Triple MOVE (op-code 130)

```
C(AC)=C(E)
C(AC+1)=C(E+1)
C(AC+2)=C(E+2)
```

TMOVEM - Triple MOVE to Memory (op-code 131)

```
C(E)=C(AC)
C(E+1)=C(AC+1)
C(E+2)=C(AC+2)
```

QMOVE - Quad MOVE (op-code 742)

```
C(AC)=C(E)
C(AC+1)=C(E+1)
C(AC+2)=C(E+2)
C(AC+3)=C(E+3)
```

QMOVEM - Quad MOVE to Memory (op-code 743)

```
C(E)=C(AC)
C(E+1)=C(AC+1)
C(E+2)=C(AC+2)
C(E+3)=C(AC+3)
```

HMOVE - Hex MOVE (op-code 744)

```
C(AC)=C(E)
C(AC+1)=C(E+1)
C(AC+2)=C(E+2)
C(AC+3)=C(E+3)
C(AC+4)=C(E+4)
C(AC+5)=C(E+5)
```

HMOVEM - Hex MOVE to Memory (op-code 745)

```
C(E)=C(AC)
C(E+1)=C(AC+1)
C(E+2)=C(AC+2)
C(E+3)=C(AC+3)
C(E+4)=C(AC+4)
C(E+5)=C(AC+5)
```

Note that MOVE[M] is used for single precision floating point numbers. DMOVE[M] is used for double precision floating point numbers or single precision complex numbers. TMOVE[M] is used for triple precision floating point numbers. QMOVE[M] is used for double precision complex numbers. HMOVE[M] is used for triple precision complex numbers.

1.13.8.3 {D,T}SKP{-,L,E,LE,A,GE,N,G} {Q,H}SKP{-,E,A,N} -

These instructions are similar to SKIP{-,L,E,LE,A,GE,N,G}. Instead of testing a single word in memory, they test a Doubleword, Tripleword, Quadword, or Hexword (respectively). Unlike SKIP, the data is not copied into the AC. The AC field is decoded as part of the op-code.

Note that for Quadwords and Hexwords it is not possible to test for {L,LE,GE,G}. These concepts are not meaningful in the context of complex numbers.

Mneumonic	Opcode	
DSKP	300	(same as CAI)
DSKPL	331	(same as SKIPL)
DSKPE	254-2	
DSKPLE	254-3	
DSKPA	304	(same as CAIA)
DSKPGE	335	(same as SKIPGE)
DSKPN	254-6	
DSKPG	254-7	
TSKP	300	(same as CAI)
TSKPL	331	(same as SKIPL)
TSKPE	254-12	
TSKPLE	254-13	
TSKPA	304	(same as CAIA)
TSKPGE	335	(same as SKIPGE)
TSKPN	254-16	
TSKPG	254-17	
QSKP	300	(same as CAI)
QSKPE	254-10	
QSKPA	304	(same as CAIA)
QSKPN	254-11	
HSKP	300	(same as CAI)
HSKPE	254-14	
HSKPA	304	(same as CAIA)
HSKPN	254-15	

1.13.8.4 {D,T,Q,H}SETZ[M] -

Zero the Doubleword, Tripleword, Quadword, or Hexword.

Mneumonic	Opcode
DSETZM	256-13
TSETZM	256-14
QSETZM	256-15
HSETZM	256-16
DSETZ	EOP-1
TSETZ	EOP-2
QSETZ	EOP-3
HSETZ	EOP-4

Note that it is better to code "{D,T,Q,H}SETZ AC," than "{D,T,Q,H}SETZM AC". The SETZ instruction is potentially faster than the corresponding SETZM (it has the potential to zero all the affected AC's simultaneously). Moreover, the prefetch of the next instruction can start sooner with SETZ than with SETZM.

Note that the SETZ group uses modulus 16 arithmetic but the SETZM group does not. Thus "TSETZ 17," will zero registers 17,0, and 1. But "TSETZM 17" will zero register 17 and core locations 20 and 21.

1.13.8.5 {-,D,T}[C]NEG -

1.13.8.5.1 NEG (EOP-7) -

NEGate the AC (take the two's complement).

```
NEG     T1,
```

is equivalent to:

```
MOVN    T1,T1
```

Note that the NEG instruction is faster than its equivalent (it allows the prefetch of the next instruction to start sooner).

1.13.8.5.2 DNEG (EOP-10) -

NEGate (take the two's complement) of the doubleword AC,AC+1.

```
DNEG    T1,
```

is equivalent to:

```
DMOVN   T1,T1
```

Note that the DNEG instruction is faster than the equivalent DMOVN.

1.13.8.5.3 TNEG (EOP-11) -

NEGate (take the two's complement) of the tripleword AC,AC+1,AC+2.

1.13.8.5.4 CNEG (EOP-12) -

NEGate the complex number in AC and AC+1.

```
CNEG    T1,
```

is equivalent to:

```
NEG     T1,
NEG     T1+1,
```

1.13.8.5.5 DCNEG (EOP-13) -

NEGate the pair of doublewords begining at AC.

```
DCNEG   T1,
```

is equivalent to:

```
DNEG    T1,
DNEG    T1+2,
```

1.13.8.5.6 TCNEG (EOP-14) -

NEGate the pair of triplewords begining at AC.

```
TCNEG   T1,
```

is equivalent to:

```
TNEG    T1,
TNEG    T1+3,
```

1.13.8.6 Conversions -

The EOP instruction supports twenty different functions for the purpose of converting between various number systems. The function codes are listed in the following table:

FROM \ TO	I	DI	F	DF	TF
I	X	20	21	22	23
DI	24	X	25	26	27
F	30	31	X	32	33
DF	34	35	36	X	37
TF	40	41	42	43	X

I - Integer
DI - Double Integer
F - Floating (single precision)
DF - Double Floating
TF - Triple Floating

Example: The instruction CDIDF converts a number from doubleword integer format into double precision floating point format. This instruction is function 26 of EOP.

Func	Name	Notes
20	CIDI	C(AC+1)=C(AC) C(AC)=0
21	CIF	same as FLTR
22	CIDF	
23	CITF	
24	CDII	C(AC)=C(AC+1) *
25	CDIF	
26	CDIDF	
27	CDITF	
30	CFI	same as FIX *
31	CFDI	*
32	CFDF	same as SETZ AC+1,
33	CFTF	
34	CDFI	*
35	CDFDI	*
36	CDFF	(rounded)
37	CDFTF	
40	CTFI	*
41	CTFDI	*
42	CTFF	(rounded)
43	CTFDF	(rounded)

* = Sets one of the overflow bits and/or traps if the conversion is not possible.

1.14 EXTEND (opcode 256-7)

The EXTEND instruction is completely different from that on the PDP10.

The format of the extend instruction is as follows:

```
EXTEND  [BYTE (^D9)OPCODE(4)AC(^D19)E2
         additional arguments
         ...]
```

The effective address of the EXTEND instruction specifies the location of an argument block. The argument block is two or more words long. The exact length depends on which function is specified.

The format of words E+1 through E+n varies from function to function. The format of word E+0, however, remains constant. The format of E+0 is identical to the basic instruction format (see section 1.3). The first nine bits specify a function code. The next 4 bits specify an AC. The last nineteen bits specify an effective address.

Note that the EXTEND instruction has two effective addresses. To avoid confusion we refer to them by seperate names: "E" and "E2". We use the name "E" to denote the original effective address (the location of the argument block). "E2" refers to the effective address specified by bits 15-37 of E+0.

Note that the term "AC" is not deemed to be confusing. There's only one AC not two. Bits 11-14 of the original instruction are not interpreted as an AC. They are, instead, part of the op-code. The term "AC" shall be used only to refer to bits 11-14 of E+0.

Most of the arguments (E+1, E+2, E+3, ...) are interpreted as indirect words (see section 1.3). Thus any value between -2**21 and +2**31-1 can be specified. By using 17 as an index register, position independent addressing can be acheived. Moreover, by setting the indirect bit, one need not specify the actual value. One can instead specify the address where the argument is to be found.

1.14.1 SMOVE - String Move (EXTEND 1) -

```
E+0/    BYTE    (^D9)1(4)0(^D19)PAD
 +1/    addr of BP A
 +2/    addr of BP B
```

Sting B is copied to string A.

Both byte pointers must be type 5. Any other type code will result in an "illegal operand" trap. Note that the argument block contains the address of the byte pointer and not the byte pointer itself.

If string B is shorter than string A, it is padded by inserting extra bytes at the end of the string. E2 specifies the value of the pad byte.

If string B is longer than string A, it is truncated by dropping bytes from the end of the string.

The following piece of pseudo code documents the algorithm used by the microcode:

```
        ;EA     E2,0(E)         ;PERFORMED BY EXTEND'S DISPATCH
        MOVEI   T1,@1(E)
        MOVE    A1,0(T1)
        MOVEI   A2,@1(T1)
        MOVEI   T1,@2(E)
        MOVE    B1,0(T1)
        MOVEI   B2,@1(T1)
SMOVE1: ILDBW   T1,B1
        MOVE    T1,E2
        IDPBL   T1,A1
        JRST    SMOVE1
```

1.14.2 BSMOVE - Backward String MOVE (EXTEND 2) -

BSMOVE is just like SMOVE except that the bytes are moved in reverse order. The difference in order will only matter if the two strings overlap.

BSMOVE is to SMOVE as BBLT is to BLT.

1.14.3 CONCAT - Concatenate Two Strings (EXTEND 3) -

This instruction is almost the same as SMOVE except that it does not insert PAD characters if the target string is shorter than the source.

```
E+0/    BYTE    (^D9)3(^D23)0
 +1/    addr of BP A
 +2/    addr of BP B
```

Both byte pointers must be type 5.

Note that E+1 is the addr of byte pointer A and not the byte pointer itself. At the conclusion of the instruction byte pointer A is incremented by the number of bytes in string B. Byte pointer B is unchanged.

1.14.4 SRCH{E,N} - Search A String -

```
E+0/    BYTE    (^D9)OPCODE(4)AC(^D19)CHAR
 +1/    addr of BP
```

SRCHE - (EXTEND 4) Search the string specified by BP for the character specified by CHAR. Stop the search upon finding the first byte equal to CHAR. If AC is non-zero, store in AC the byte number where the target character was found (zero if the character was not found). The instruction skips iff the target character was found. Note that BP should initially point to the ficticious byte just before the first byte to be tested (i.e. the C field of the BP contains the number of bytes to be tested).

SRCHN - (EXTEND 5) Search the string for the first byte which is not equal to CHAR. If AC is non-zero, store in AC the byte number of the first byte not equal to CHAR (zero if the entire string was equal to CHAR). The instruction skips iff there is at least one byte not equal to CHAR.

Note that BP must be a type 5 byte pointer.

Note that its perfectly legal to use 32 bit bytes (full words).

1.14.5 SPAT{E,N} Search A String (with Pattern Matching) -

```
E+0/    BYTE    (^D9)OPCODE(4)AC(^D19)<ADDR OF BIT MASK>
 +1/    addr of BP
 +2/    MIN
 +3/    MAX
```

These instructions are quite similar to SRCH{E,N}. Instead of searching for a single character, however, we search for any of a group of characters. This group is specified by a bit mask. The first bit in the mask corresponds to the character whoose value is MIN. The last bit in the mask corresponds to the character whoose value is MAX. The length of the mask (in bits) is MAX-MIN+1.

SPATE is function 6 of EXTEND. SPATN is function 7.

Example: Pseudo code for SPATE

```
SPATE:  EA      E2,0(E)
        MOVEI   T1,@1(E)
        MOVE    A1,0(T1)
        MOVEI   A2,@1(T1)
        MOVEI   MN,@2(E)
        MOVEI   MX,@3(E)
        MOVEI   K,1
SPATE1: ILDBW   T1,A1
        JRST    SPATE2
        CAML    T1,MN
        CAMLE   T1,MX
        JRST    SPATE3
        SUB     T1,MN
        LSHC    T1,-5
        LSH     T2,-^D27
        MOVNS   T2
        MOVSI   T3,100000          ;SET SIGN BIT
        LSH     T3,(T2)
        ADD     T1,E2
        TDNE    T3,(T1)
        JRST    SPATE4
SPATE3: AOJA    K,SPATE1
SPATE2: SETZ    K,
SPATE4: LDB     T1,[POINT 4,0(E),14] ;AC
        SKIPE   T1
        MOVEM   K,(T1)
        SKIPN   K
        -                          ;NON-SKIP
        -                          ;SKIP
```

1.14.6 SCOMP{-,L,E,LE,A,GE,N,G} - String COMPare -

```
E+0/    BYTE    (^D9)OPCODE(4)AC(^D19)PAD
 +1/    addr of BP A
 +2/    addr of BP B
```

Compare string A with string B. Stop upon finding the first pair of bytes that are unequal. If AC is non-zero, store in AC the byte number of the first pair of bytes that are unequal. Skip the next instruction depending on which string is greater.

Both byte pointers must be type 5. Any other type code will result in an "illegal operand" trap.

Note that SCOMP and SCOMPA are not no-ops. AC returns the byte number of the first byte which is unequal.

Using the AC returned by SCOMP, an ADJBP will point to the first byte that differs.

Name	EXTEND	Skip
SCOMP	10	never
SCOMPL	11	A<B
SCOMPE	12	A=B
SCOMPLE	13	A<=B
SCOMPA	14	always
SCOMPGE	15	A>=B
SCOMPN	16	A<>B
SCOMPG	17	A>B

1.15 Conceal

The PDP10 had a flag bit called "PUBLIC". On the new machine this flag has been extended into a three bit field. The field is called "CONCEAL". A value of zero in the field is roughly equivalent to the PUBLIC bit being lit. A non-zero value means that the program is concealed. Note that there are seven different flavors of concealment.

This allows the program to attach to seven different segments, each of which contains proprietary code. Each segment is protected from unauthorized access by the user. Each segment is protected from each of the others.

If the program is running in PUBLIC mode (i.e. CONCEAL=0) then it can only reference those pages which are PUBLIC.

If, however, the program is running with CONCEAL=X then it can reference both PUBLIC pages and those pages with CONCEAL=X.

The one exception to these rules is the PORTAL instruction. Any program can reference any word in any page (regardless of the value of CONCEAL) but only if the reference is a fetch for execution and only if the word contains a PORTAL instruction. The PORTAL instruction is used to declare the legal entry points to a concealed segment.

Although the CONCEAL field is only 3 bits wide, this does not mean that the program can only attach to seven different segments. He can attach to much more than this. But only seven of the segments can be concealed. As concealed segments are rather rare, this should not be restrictive.

Note that the conceal field is only meaningful in user mode. All references are considered legal if the machine is in exec mode.

Note that ULDB and UDPB are considered to be user mode references and it is indeed possible for these instructions to get a proprietary violation. This feature can be disabled by lighting the "CONCEAL Disable Bit" in the FLAGS register (see section 1.16).

1.16 FLAGS

On the PDP10, the "PC" was divided into 2 halves. The RH contained the program counter and the LH contained flags. On the new machine each half has been expanded into a full 32 bit register.

Note that the FLAGS register is accessable only from exec. A user mode program can neither read nor write the FLAGS register.

Note that the FLAGS register does not contain any overflow bits. These have been moved elsewhere (see section 1.17).

Format of the FLAGS register:

Bit(s)	What
0	User Mode (sign bit)
1-2	spare
3	User Mode Address Break Inhibit If on, prevents user references from causing address breaks. The bit is not intended as an equivalent of bit 8 in the KL10 PC. It is, instead, an equivalent for bit 4 in the KI10 "DATAO PAG,". The bit is not a one shot. When lit the bit stays lit forever. To simulate a one shot use the SSTEP instruction. The bit has no affect on exec mode references nor physical references.
4	Load Exec AC Set When the FLAGS register is written, this bit determines whether bits 5-7 are loaded. A one in bit 4 causes bits 5-7 to be loaded as the new exec AC set. A zero in bit 4 causes bits 5-7 to be ignored (the exec AC set is not changed). When the FLAGS register is read, bit 4 is always on.
5-7	Exec AC Set
10	Load User AC Set

Similar to bit 4 but controls bits 11-13 instead of bits 5-7.

11-13 User AC Set

14-17 spare

20 Disable CONCEAL

If this bit is on, the CONCEAL field is ignored. All references are considered legal.

The bit is intended so that CONCEAL may be temporarily disabled during a routine that does lots of ULDB's and UDPB's. Most routines that do these instructions, however, do not wish to ignore CONCEAL. It is desired for the instruction to trap if the reference is not legal for the segment that issued the current UUO.

21 Load CONCEAL

Similar to bit 4 but controls bits 22-24 instead of bits 5-7.

22-24 CONCEAL

25 spare

26-27 Load PI Enables

These bits control the usage of bits 30-37. When the FLAGS register is written, bits 26-27 are interpreted as follows:

00 - No change to PI enables. Ignore bits 30-37.

01 - Turn on those enables selected by bits 30-37.

10 - Turn off those enables selected by bits 30-37.

11 - Load the PI enables from bits 30-37.

When the FLAGS register is read, bits 26 and 27 are always on.

30 PI System

31 PIA 1

32 PIA 2

33 PIA 3

34 PIA 4

35	PIA 5
36	PIA 6
37	PIA 7

1.17 TRAPS

The microcode maintains an internal register called TRAPS. This register is a bit mask which indicates the types of failures that have occurred during this instruction (integer overflow, floating overflow, no divide, etc). Normally this register will be zero. If, however, the register is non-zero at the conclusion of the instruction, the microcode will take special actions. The following code is executed at the conclusion of every instruction:

```
        JUMPE   TRAPS,NEXT          ;NO TRAPS GOTO NEXT INSTRUCTION
GOO:    IORB    TRAPS,.JBTRP        ;SET BITS IN USER CORE
        AND     TRAPS,.JBNBL        ;ENABLED FOR THIS TRAP?
        JUMPE   TRAPS,NEXT          ;NO
        TDNE    TRAPS,.JBMOD        ;YES, WHO FIELDS THE TRAP?
        JRST    FOO                 ;MONITOR
        MOVEM   PC,.JBTOP           ;USER, STORE OLD PC
        MOVE    PC,.JBTNP           ;GET NEW PC
        SETZ    TRAPS,              ;CLEAR THEM
        JRST    NEXT                ;GOTO NEXT INSTRUCTION
FOO:    MOVEM   FLAGS,.ESTOP        ;STORE OLD PC
        MOVEM   PC,.ESTOP+1
        MOVE    FLAGS,.ESTNP        ;GET NEW PC
        MOVE    PC,.ESTNP+1
        SETZ    TRAPS,              ;CLEAR THEM
        JRST    NEXT                ;GOTO NEXT INSTRUCTION
```

It is important to note that the instruction at GOO is an IORB and not an IORM. The user trap routine must be careful to clear .JBTRP before dimissing the current trap else havoc may occur on subsequent traps. (See also section 2.5: MAT).

1.17.1 Trap Types -

Bit	Trap
0	Integer Overflow
1	Integer No Divide
2	Floating Overflow
3	Floating Underflow
4	Floating No Divide
5	String Overflow
6	String No Divide
7	Illegal Operand
10	PDL Overflow

1.18 JOBDAT

Word	Name	What
0-17	-	ACs
20	.JBDAT	Addr of 1st word beyond JOBDAT.
21	.JBTRP	Bit mask of traps which have occurred.
22	.JBNBL	Bit mask of enabled traps. A zero bit means that the error condition is ignored. If the bit is a one, however, a trap will occur.
23	.JBMOD	Bit mask that tells who fields the trap: A zero bit means that the user fields the trap (via .JBTNP). If the bit is a one, however, then the monitor fields the trap (via .ESTNP). Note that even the traps which occur in exec mode have the choice of being fielded either by .JBTNP or .ESTNP.
24	.JBTOP	Trap Old PC.
25	.JBTNP	Trap New PC.
26	.JBUUO	Upon encountering an LUUO, the microcode will store the opcode and AC field in this location. Only bits 0-14 of this location are meaningful. Bits 15-37 are indeterminate. Note that in the case of an XCT of an LUUO it would be non-trivial to determine the opcode via .JBUOP
27	.JBUUE	LUUO Effective Address.
30	.JBUOP	LUUO Old PC
31	.JBUNP	LUUO New PC
32-n	-	Fields defined by the monitor.

1.19 Page Maps

It is not our purpose to define the address translation mechanism. We consider this to be implementation dependant.

We will only define those portions of the page map which are not related to address translation. We do not assign actual word numbers. The offsets are considered implementation dependant. We define only the field names and the number of words occupied by each field.

1.19.1 UPMP - User Page Map Page -

Name	Words	What
.USK	10	These words are used by the microcode to store status information when an interruptable instruction is interrupted. The information tells exactly how far the instruction got before it was interrupted. In order to restart the instruction at the correct spot, the microcode will retrieve these words from the UPMP. (See section 2.5 for an example). The format of these words is different for each instruction. In general, however, word 0 is zero if the instruction is to start at the beginning. Word 0 is non-zero if the instruction is to restart in the middle. This is not to say that .USK is inspected each time an interruptable instruction begins. It is inspected only if the RESTART flag is on. The RESTART flag is lit by the XDIS instruction. Example: see section 2.5 (the MAT instruction). Note that for the sake of simplicity the MAT instruction is the only one where we have included a complete description of .USK . The reader should assume, however, that all interruptable instructions function in a manner consistent with the MAT instruction.

1.19.2 EPMP - Exec Page Map Page -

Name	Words	What
.ESTOP	2	Trap Old PCW (FLAGS and PC).
.ESTNP	2	Trap New PCW (FLAGS and PC).
.ESK	10*10	Eight blocks of eight words each. There's one block for each interrupt level (plus one for UUO level). Each block is a copy of .USK (used to restart an interruptable instruction).

2.0 PART II: THE VECTOR OPTION

The vector instructions are considered to be an option to the basic machine. Each of the instructions can be implemented purely in the microcode. They are best implemented, however, by building special purpose hardware.

It is not envisioned that the custom hardware would exist in the initial implementation of the machine. The instructions would initially be implemented in the microcode. A pipeline version wouldn't come until much later.

Even without special purpose hardware, the instruction will still run much faster than if the equivalent were coded in assembly.

2.1 VMOVE - MOVE A Vector (EXTEND 200)

The VMOVE instruction will move vector B to vector A. The format of the argument block is as follows:

```
E+0/    BYTE    (^D9)200(4)0(^D19)N
 +1/    addr of vector A
 +2/    delta A
 +3/    addr of vector B
 +4/    delta B
 +5/    number of items
```

As always, each of these parameters is interpreted as an indirect word (see section 1.3). Thus the programmer can either specify the actual value or the address where the value is to be found.

The E+2 parameter specifies the distance (in words) between two items in vector A. The E+4 parameter specifies the distance between two items in vector B. Parameter N specifies the number of words per item. N can have almost any value, but the most popular values are as follows:

N	Datum
1	integer or single precision floating point
2	double floating or single complex
3	triple precision floating point
4	double precision complex
6	triple precision complex

The following piece of pseudo code documents the algorithm used by the microcode:

```
VMOVE:  MOVEI   AO,@1(E)        ;ADDR OF VECTOR A
        MOVEI   BO,@3(E)        ;ADDR OF VECTOR B
        MOVEI   CO,@5(E)        ;NUMBER OF ITEMS
        MOVEI   DA,@2(E)        ;DELTA A
        MOVEI   DB,@4(E)        ;DELTA B
        EA      N,0(E)          ;E2
VMOVE2: MOVE    AI,AO           ;ADDR OF ITEM A
        MOVE    BI,BO           ;ADDR OF ITEM B
        MOVE    CI,N            ;WORDS PER ITEM
VMOVE1: MOVE    T1,(AI)         ;COPY AN ITEM
        MOVEM   T1,(AO)
        ADDI    AI,1
        ADDI    AO,1
        SOJG    CI,VMOVE1
        ADD     AO,DA           ;STEP TO NEXT ITEM
        ADD     BO,DB
        SOJN    CO,VMOVE2       ;LOOP
```

Some examples will help to clarify. Consider the following fortran program:

```
C MOVE A VECTOR
  REAL X(100),Y(100)
  DO 1 I=1,100
1 X(I)=Y(I)
```

This could be coded as:

```
E+0/    BYTE    (^D9)200(4)0(^D19)1
 +1/    X
 +2/    1
 +3/    Y
 +4/    1
 +5/    ^D100
```

The program:

```
C MOVE A MATRIX
  REAL X(100,5),Y(100,5)
  DO 1 I=1,100
  DO 1 J=1,5
1 X(I,J)=Y(I,J)
```

Could be coded as:

```
E+0/    BYTE    (^D9)200(4)0(^D19)1
 +1/    X
 +2/    1
 +3/    Y
 +4/    1
 +5/    ^D500
```

```
C MOVE A "COLUMN"
  REAL X(100,5),Y(100,5)
  DO 1 I=1,100
1 X(I,1)=Y(I,2)
```

Could be coded as follows: (if in row major order)

```
E+0/    BYTE    (^D9)200(4)0(^D19)1
 +1/    X
 +2/    5
 +3/    Y+1
 +4/    5
 +5/    ^D100
```

Or it could be coded as follows: (if in column major order)

```
E+0/    BYTE    (^D9)200(4)0(^D19)1
 +1/    X
 +2/    1
 +3/    Y+^D100
 +4/    1
 +5/    ^D100
```

```
C MOVE A "ROW" TO A "COLUMN"
  REAL X(100,100),Y(100,100)
  DO 1 I=1,100
1 X(I,2)=Y(1,I)
```

Could be coded as follows: (if in row major order)

```
E+0/    BYTE    (^D9)200(4)0(^D19)1
 +1/    X+1
 +2/    ^D100
 +3/    Y
 +4/    1
 +5/    ^D100
```

```
C MOVE A DOUBLE "ROW" TO A DOUBLE "COLUMN"
  DOUBLE PRECISION X(100,100),Y(100,100)
  DO 1 I=1,100
1 X(I,2)=Y(1,I)
```

Could be coded as follows: (row major)

```
E+0/    BYTE    (^D9)200(4)0(^D19)2
 +1/    X+2
 +2/    ^D200
 +3/    Y
 +4/    2
 +5/    ^D100
```

```
C MOVE REAL PART OF "COLUMN" TO "ROW"
  COMPLEX Y(100,100)
  REAL X(100,100)
  DO 1 I=1,100
1 X(2,I)=REAL(Y(I,1))
```

Could be coded as follows: (row major)

```
E+0/    BYTE    (^D9)200(4)0(^D19)1
 +1/    X+^D100
 +2/    1
 +3/    Y
 +4/    ^D200
 +5/    ^D100
```

2.2 V{MUL,DIV,ADD,SUB,{F,DF,TF,C,DC,TC}{AD,SB,MP,DV}}{2,3}

These instructions are used to perform arithmetic operations upon pairs of vectors. The format of the argument block is as follows:

```
E+0/    BYTE     (^D9)201(4)0(^D19)FUNC
 +1/    number of items
 +2/    addr of vector A
 +3/    delta A
 +4/    addr of vector B
 +5/    delta B
 +6/    addr of vector C          ;included only if 3 operands
 +7/    delta C                   ;included only if 3 operands
```

The E2 field gives the function code (each of these instructions is implemented as a function of EXTEND 201). Note that some of the functions have two operands and some have three (hence the last character of the mneumonic).

Name	FUNC	Op	Datum
VADD2	0	A=A+B	INTEGER
VADD3	1	C=A+B	INTEGER
VSUB2	2	A=A-B	INTEGER
VSUB3	3	C=A-B	INTEGER
VMUL2	4	A=A*B	INTEGER
VMUL3	5	C=A*B	INTEGER
VDIV2	6	A=A/B	INTEGER
VDIV3	7	C=A/B	INTEGER
VFAD2	10	A=A+B	SINGLE FLOATING
VFAD3	11	C=A+B	SINGLE FLOATING
VFSB2	12	A=A-B	SINGLE FLOATING
VFSB3	13	C=A-B	SINGLE FLOATING
VFMP2	14	A=A*B	SINGLE FLOATING
VFMP3	15	C=A*B	SINGLE FLOATING
VFDV2	16	A=A/B	SINGLE FLOATING
VFDV3	17	C=A/B	SINGLE FLOATING
VDFAD2	20	A=A+B	DOUBLE FLOATING
VDFAD3	21	C=A+B	DOUBLE FLOATING
VDFSB2	22	A=A-B	DOUBLE FLOATING
VDFSB3	23	C=A-B	DOUBLE FLOATING
VDFMP2	24	A=A*B	DOUBLE FLOATING
VDFMP3	25	C=A*B	DOUBLE FLOATING
VDFDV2	26	A=A/B	DOUBLE FLOATING
VDFDV3	27	C=A/B	DOUBLE FLOATING
VTFAD2	30	A=A+B	TRIPLE FLOATING
VTFAD3	31	C=A+B	TRIPLE FLOATING
VTFSB2	32	A=A-B	TRIPLE FLOATING
VTFSB3	33	C=A-B	TRIPLE FLOATING
VTFMP2	34	A=A*B	TRIPLE FLOATING
VTFMP3	35	C=A*B	TRIPLE FLOATING
VTFDV2	36	A=A/B	TRIPLE FLOATING
VTFDV3	37	C=A/B	TRIPLE FLOATING
VCAD2	40	A=A+B	SINGLE COMPLEX

VCAD3	41	C=A+B	SINGLE COMPLEX
VCSB2	42	A=A-B	SINGLE COMPLEX
VCSB3	43	C=A-B	SINGLE COMPLEX
VCMP2	44	A=A*B	SINGLE COMPLEX
VCMP3	45	C=A*B	SINGLE COMPLEX
VCDV2	46	A=A/B	SINGLE COMPLEX
VCDV3	47	C=A/B	SINGLE COMPLEX
VDCAD2	50	A=A+B	DOUBLE COMPLEX
VDCAD3	51	C=A+B	DOUBLE COMPLEX
VDCSB2	52	A=A-B	DOUBLE COMPLEX
VDCSB3	53	C=A-B	DOUBLE COMPLEX
VDCMP2	54	A=A*B	DOUBLE COMPLEX
VDCMP3	55	C=A*B	DOUBLE COMPLEX
VDCDV2	56	A=A/B	DOUBLE COMPLEX
VDCDV3	57	C=A/B	DOUBLE COMPLEX
VTCAD2	60	A=A+B	TRIPLE COMPLEX
VTCAD3	61	C=A+B	TRIPLE COMPLEX
VTCSB2	62	A=A-B	TRIPLE COMPLEX
VTCSB3	63	C=A-B	TRIPLE COMPLEX
VTCMP2	64	A=A*B	TRIPLE COMPLEX
VTCMP3	65	C=A*B	TRIPLE COMPLEX
VTCDV2	66	A=A/B	TRIPLE COMPLEX
VTCDV3	67	C=A/B	TRIPLE COMPLEX

Note that E+3 or E+5 may be zero, meaning that the operand isn't really a vector at all. It is, instead, a scalar. The microcode will optimize these cases.

Note that the two operand instruction is not read-pause-write.

The following piece of pseudo code documents the algorithm used by the microcode for VFMP2 and VFMP3:

```
;ENTER HERE FOR VFMP3
VFMP3:  MOVEI   C,@6(E)         ;ADDR OF VECTOR C
        MOVEI   DC,@7(E)        ;DELTA C
        JRST    VFMPA

;ENTER HERE FOR VFMP2
VFMP2:  MOVEI   C,@2(E)         ;VECTOR C SAME AS VECTOR A
        MOVEI   DC,@3(E)
VFMPA:  MOVEI   A,@2(E)         ;ADDR OF VECTOR A
        MOVEI   DA,@3(E)        ;DELTA A
        MOVEI   B,@4(E)         ;ADDR OF VECTOR B
        MOVEI   DB,@5(E)        ;DELTA B
        MOVEI   N,@1(E)         ;NUMBER OF ITEMS
        JUMPE   DB,VFMPC        ;SCALAR B?
        JUMPN   DA,VFMPB        ;SCALAR A?
        EXCH    A,B             ;SWAP THEM
        MOVE    DA,DB

;HERE IF VECTOR B IS ACTUALLY A SCALAR
VFMPC:  MOVE    X,(B)           ;PICK UP SCALAR
```

```
VFMPD:  MOVE    T1,(A)          ;MULTIPLY
        FMP     T1,X
        MOVEM   T1,(C)
        ADD     A,DA            ;STEP TO NEXT ITEM
        ADD     C,DC
        SOJN    N,VFMPD         ;LOOP
        JRST    DONE

;HERE IF DA AND DB BOTH NON-0
VFMPB:  MOVE    T1,(A)          ;MULTIPLY
        FMP     T1,(B)
        MOVEM   T1,(C)
        ADD     A,DA            ;STEP TO NEXT ITEM
        ADD     B,DB
        ADD     C,DC
        SOJN    N,VFMPB         ;LOOP
        JRST    DONE
```

2.3 V{ADD,MUL,{F,DF,TF,C,DC,TC}{AD,MP}}

Compute the sum (or product) of all the items in a vector and store the result in AC. The format of the argument block is as follows:

```
E+0/    BYTE    (^D9)202(4)AC(^D19)FUNC
 +1/    number of items
 +2/    addr of vector
 +3/    delta item
```

Each of these instructions is a function of EXTEND 202:

Name	FUNC	Op	Datum
VADD	0	+	INTEGER
VMUL	1	*	INTEGER
VFAD	2	+	SINGLE FLOATING
VFMP	3	*	SINGLE FLOATING
VDFAD	4	+	DOUBLE FLOATING
VDFMP	5	*	DOUBLE FLOATING
VTFAD	6	+	TRIPLE FLOATING
VTFMP	7	*	TRIPLE FLOATING
VCAD	10	+	SINGLE COMPLEX
VCMP	11	*	SINGLE COMPLEX
VDCAD	12	+	DOUBLE COMPLEX
VDCMP	13	*	DOUBLE COMPLEX
VTCAD	14	+	TRIPLE COMPLEX
VTCMP	15	*	TRIPLE COMPLEX

Example: Pseudo code for VFAD:

```
VFAD:   MOVEI   N,@1(E)
        MOVEI   A,@2(E)
        MOVEI   DA,@3(E)
        SETZ    S,
LOOP:   FAD     S,(A)
        ADD     A,DA
        SOJN    N,LOOP
        LDB     AC,[POINT 4,0(E),14]
        MOVEM   S,(AC)
```

2.4 {-,D,T}POLY

Evaluate a polynomial (single, double, or triple precision floating point).

```
Y = C0 + C1*X + C2*X**2 + C3*X**3 + ...
```

```
E+0/    BYTE    (^D9)OPCODE(4)AC(^D19)0
 +1/    number of items
 +2/    addr of vector
 +3/    delta item
```

The value of X is taken from the AC. The polynomial is computed using a vector of coeffiecients. The result is placed back in the AC.

Name	EXTEND
POLY	203
DPOLY	204
TPOLY	205

Example: Pseudo code for POLY:

```
POLY:   MOVEI   N,@1(E)
        MOVEI   A,@2(E)
        MOVEI   DA,@3(E)
        LDB     AC,[POINT 4,0(E),14]
        MOVE    X,(AC)
        SETZ    Y,
        FMOVEI  Z,1.0/40000      ;Z=1.0
LOOP:   MOVE    T1,(A)
        FMP     T1,Z
        ADD     Y,T1
        FMP     Z,X
        ADD     A,DA
        SOJG    N,LOOP
        MOVEM   Y,(AC)
```

2.5 {-,F,DF,TF,C,DC,TC}MAT

Multiply matrix A by matrix B according to the rules of linear algebra. Place the result in matrix C (C=A*B).

Note:

number of cols in mat A = number of rows in mat B
number of rows in mat A = number of rows in mat C
number of cols in mat B = number of cols in mat C

Each matrix can be stored in either row major order or column major order.

The format of the argument block is as follows:

```
E+0/    BYTE    (^D9)OPCODE(^D23)0
 +1/    addr of matrix A
 +2/    delta col A
 +3/    delta row A
 +4/    addr of matrix B
 +5/    delta col B
 +6/    delta row B
 +7/    addr of matrix C
 +10/   delta col C
 +11/   delta row C
 +12/   cols in A, rows in B
 +13/   rows in A, rows in C
 +14/   cols in B, cols in C
```

Name	EXTEND	Datum
MAT	206	INTEGER
FMAT	207	SINGLE FLOATING
DFMAT	210	DOUBLE FLOATING
TFMAT	211	TRIPLE FLOATING
CMAT	212	SINGLE COMPLEX
DCMAT	213	DOUBLE COMPLEX
TCMAT	214	TRIPLE COMPLEX

The following piece of pseudo code uses 15 registers. It documents the algorithm that the microcode would use to implement FMAT. DFMAT, on the other hand, would require 17 registers. TFMAT would require 19. Etc.

The example, however, should not be taken literally. When implemented in microcode, we would use additional registers. For example, E+3 would be loaded into a register just like E+2 is loaded into DAC. E+3 would not be fetched from core each time it was needed.

```
        S=0     ;SUM
        T1=1    ;TEMPS (USED ONLY DURING RESTART)
        T2=T1+1
        T3=T2+1
        CO=T1   ;START OF COL IN C (OUTTER LOOP)
        CI=T2   ;CURRENT ITEM IN C (INNER LOOP)
        BO=4    ;START OF COL IN B (OUTTER LOOP)
        BI=T3   ;CURRENT ITEM IN B (INNER LOOP)
        AO=5    ;START OF ROW IN A (OUTTER LOOP)
        AI=6    ;CURRENT ITEM IN A (INNER LOOP)
        K=7     ;COUNT OF MULTIPLYS
        KO=10   ;COUNT OF COLS LEFT IN C (OUTTER LOOP)
        KM=11   ;COUNT OF ROWS LEFT IN C (MIDDLE LOOP)
        KI=12   ;COUNT OF ROWS LEFT IN B (INNER LOOP)
        DAC=13  ;DELTA A COL (E+2)
        DBR=14  ;DELTA B ROW (E+6)
        X=15    ;CURRENT ITEM
        E=16    ;EFFECTIVE ADDR

;ENTER HERE
FMAT:   MOVEI   DAC,@2(E)       ;DELTA A COL
        MOVEI   DBR,@6(E)       ;DELTA B ROW
        SKIPE   K,.USK          ;RESTART?
        JRST    FMAT5           ;YES
        MOVEI   BO,@4(E)        ;ADDR OF 1ST COL IN B
        MOVEI   CO,@7(E)        ;ADDR OF 1ST COL IN C
        MOVEI   KO,@14(E)       ;COLS IN B, COLS IN C
;START OF OUTTER LOOP:
FMAT3:  MOVEI   AO,@1(E)        ;ADDR OF 1ST ROW IN A
        MOVE    CI,CO           ;START OF COL IN C
        MOVEI   KM,@13(E)       ;ROWS IN A, ROWS IN C
;START OF MIDDLE LOOP:
FMAT2:  MOVE    AI,AO           ;START OF ROW IN A
        MOVE    BI,BO           ;START OF COL IN B
        MOVEI   KI,@12(E)       ;COLS IN A, ROWS IN B
        SETZ    S,
;START OF INNER LOOP:
FMAT1:  BLNE    n,m             ;INTERRUPT PENDING?
        JRST    FMAT4           ;YES
FMAT6:  MOVE    X,(AI)          ;GET ITEM FROM A
        FMP     X,(BI)          ;TIMES ITEM FROM B
        FAD     S,X             ;ADD TO SUM
        ADDI    K,1             ;BUMP COUNT
        ADD     AI,DAC          ;NEXT ITEM IN ROW A
        ADD     BI,DBR          ;NEXT ITEM IN COL B
        SOJN    KI,FMAT1        ;LOOP
        MOVEM   S,(CI)          ;STORE SUM IN C
        ADDI    AO,@3(E)        ;NEXT ROW IN A
        ADDI    CI,@11(E)       ;NEXT ROW IN C
        SOJN    KM,FMAT2        ;LOOP
        ADDI    BO,@5(E)        ;NEXT COL IN B
        ADDI    CO,@10(E)       ;NEXT COL IN C
        SOJN    KO,FMAT3        ;LOOP
        JRST    DONE
```

```
;HERE IF INTERRUPT (OR PAGE FAULT)
FMAT4:  MOVEM   S,.USK+1        ;SAVE SUM
        SKIPE   X,TRAPS         ;ANY TRAPS SO FAR?
        IORM    X,.JBTRP        ;YES, STORE THEM
        MOVEM   K,.USK          ;SAVE COUNT
        JRST    FOO             ;GO SERVICE INTERRUPT

;HERE IF INSTRUCTION IS RESTARTED
FMAT5:  MOVEI   KI,@12(E)       ;COLS IN A, ROWS IN B
        MOVEI   KM,@13(E)       ;ROWS IN A, ROWS IN C
        MOVEI   KO,@14(E)       ;COLS IN B, COLS IN C
        MOVE    T2,K            ;COUNT
        SETZM   .USK            ;RESET COUNT
        IDIV    T2,KI           ;T3=ROW NUMBER IN B
        MOVE    T1,T2           ;ITEM NUMBER IN C
        IDIV    T1,KM           ;T1=COL NUMBER IN C
                                ;T2=ROW NUMBER IN C
        SUB     KI,T3           ;ROWS LEFT IN B
        SUB     KM,T2           ;ROWS LEFT IN C
        SUB     KO,T1           ;COLS LEFT IN C
        MOVE    AO,T2           ;ROW NUMBER IN A
        IMULI   AO,@3(E)        ;OFFSET
        ADDI    AO,@1(E)        ;ADDR
        MOVE    AI,T3           ;COL NUMBER IN A
        IMUL    AI,DAC          ;OFFSET
        ADD     AI,AO           ;ADDR
        MOVE    BO,T1           ;COL NUMBER IN B
        IMULI   BO,@5(E)        ;OFFSET
        ADDI    BO,@4(E)        ;ADDR
        ;MOVE   BI,T3           ;ROW NUMBER IN B
        IMUL    BI,DBR          ;OFFSET
        ADD     BI,BO           ;ADDR
        ;MOVE   CO,T1           ;COL NUMBER IN C
        IMULI   CO,@10(E)       ;OFFSET
        ADDI    CO,@7(E)        ;ADDR
        ;MOVE   CI,T2           ;ROW NUMBER IN C
        IMULI   CI,@11(E)       ;OFFSET
        ADD     CI,CO           ;ADDR
        MOVE    X,.JBTRP        ;RESTORE TRAPS
        MOVEM   X,TRAPS
        MOVE    S,.USK+1        ;GET SUM BACK
        JRST    FMAT6           ;CONTINUE
```

Note that instead of using .USK+1 to store the sum, we could use 0(CI) instead. This would work just fine in the case of an interrupt. But wouldn't work as well in the case of a page fault (as the reference to 0(CI) might cause a second page fault). This problem with the page fault case can be corrected, however, by changing the instruction at FMAT1-1 to "SETZB S,(CI)". This will insure that if the page fault is going to occur, it does so during a favorable window.

Regardless, the .USK+1 approach is deemed better than the 0(CI) approach. Its slightly faster and its less of a kludge.

Note the manner in which traps are handled. When an overflow occurs the instruction does not abort. It is not until the conclusion of the instruction that the microcode tests whether or not the user is enabled to trap this overflow. Even if the instruction is interrupted and restarted, the existence of the overflow condition will not cause a trap until the instruction comes to full completion.

Note the manner in which TRAPS is reloaded from .JBTRP. If the bit is on in .JBTRP, we make no attempt to decipher whether it was this instruction that lit the bit. It is assumed that the user will clear the bit in .JBTRP each time the trap occurs.

3.0 PART III: STRING ARITHMETIC

The "String Arithmetic Option" provides a wide variety of EXTEND instructions to perform arithmetic operations upon strings of digits.

All of the strings take the following format:

```
!  byte 0   !  byte 1   !  byte 2   !           !  byte N   !
+-----------+-----------+-----------+-----------+-----------+
!   sign    ! 1st digit ! 2nd digit !    ...    ! Nth digit !
+-----------+-----------+-----------+-----------+-----------+
```

Note that it takes N+1 bytes to represent an N digit signed number.

All of the instructions take their arguments in the form of a type 5 byte pointer. The BP points to the sign byte. Therefore the C field contains a count of the digits (it contains N not N+1). Example: The string

```
FOO:    ASCII   "+0047"
```

would be represented by the byte pointer:

```
        POINTR  8,FOO,7,4
```

Note that the string doesn't necessarily have to be ASCII, and doesn't necessarily have to be base ten. Any arbitrary number system can be used: any radix, and any character set. To define a number system you must code a four word parameter block known as the NSB (Number System Block). The format of the NSB is as follows:

```
NSB+0/  radix
   +1/  character code for "0"
   +2/  character code for "+"
   +3/  character code for "-"
```

Note that each of these words is interpreted as an indirect word (see section 1.3). Thus you always have a choice: you can specify the actual value or you can specify a pointer to the value.

Note that by merely specifing three characters ("0", "+", and "-") you have completely defined the character set. These are the only three characters needed to perform arithmetic.

Note that a string is considered negative if the sign byte is anything other than "+". It doesn't necessarily have to be "-". If, however, the machine generates a minus sign, it will use the specified value of "-".

Note that hexadecimal cannot be represented. The sixteen possible digits are not consecutive character codes.

BCD can be represented by specifying the byte size as 4, the radix as ^D10, and "0" as ^D0.

One of the most popular techniques, however, is to set the byte size to ^D32, the radix to ^D10**9, and "0" to ^D0.

Note that any character outside the range "0" to "0"+radix-1 is regarded as a zero. Thus spaces are taken as zeros.

3.1 Unsigned Arithemetic

If NSB+3 is equal to NSB+2 ("-" is equal to "+") then the string is taken to be "unsigned". This is not to say that the sign byte does not exist. You must, as always, allocate space for the sign byte. The value of the sign byte, however, is totally ignored. The string is always considered positive.

In practice, the tendency is to specify a byte pointer that points to the last byte of the previous field. This is a "ficticious sign byte". The ficticious byte is never referenced.

3.2 S{ADD,SUB,MUL,DIV}{2,3}

These instructions are used to perform arithmetic operations upon strings of digits. Strings can be added, subtracted, multiplied, or divided.

The instructions are implemented as functions 100 through 107 of EXTEND. Some of the functions have two operands (known as operand A and operand B). Some of the functions have three operands (known as A, B, and C).

Code	Name	Function	
100	SADD2	A=A+B	
101	SADD3	C=A+B	
102	SSUB2	A=A-B	
103	SSUB3	C=A-B	
104	SMUL2	A=A*B	*
105	SMUL3	C=A*B	
106	SDIV2	A=A/B	*
107	SDIV3	C=A/B	*

* = To complete these instructions the microcode will require scratch space. This space will be allocated on the stack. The stack pointer is specified by the AC field (bits 11-14 of E+0).

The format of the argument block for these instructions is as follows:

```
E+0/    BYTE    (^D9)OPCODE(4)AC(^D19)NSB
 +1/    addr of BP A
 +2/    addr of BP B
 +3/    addr of BP C    ;included only if 3 operands
```

Note that all 3 byte pointers must be type 5 byte pointers. Note that type 5 byte pointers are two words long. Note that the argument block does not contain the byte pointers themselves but rather the addresses where the byte pointers can be found. This results in a savings provided that each string is referenced at least twice. In a typical program the average number of references per variable is significantly greater than two.

Note that the byte pointers are not incremented, decremented, or altered in any way by these instructions.

When substracting one unsigned string from another, an overflow trap will occur if the result is negative. Unsigned strings are not allowed to be negative.

Example: Consider the string X whoose value is "+0023". We wish to evaluate the expression "Y=(X+3)*47". This could be coded as follows:

```
         ...
         EXTEND  OP1
         EXTEND  OP2
         ...
OP1:     SADD3   MYNSB
         BPX
         BP3
         BPY
OP2:     SMUL2   P,MYNSB
         BPY
         BP47
BPX:     POINTR  8,X,7,4
BPY:     POINTR  8,Y,7,4
BP3:     POINTR  8,LIT3,7,1
BP47:    POINTR  8,LIT47,7,2
X:       ASCII   "+0023"
Y:       BLOCK   2
LIT3:    ASCII   "+3"
LIT47:   ASCII   "+47"
MYNSB:   ^D10
         "0"
         Z       @MYPLUS
         "-"
MYPLUS:  "+"
```

The following piece of pseudo code is intended to document the algorithm used by the microcode for the instructions SSUB2, SADD2, SSUB3, and SADD3:

```
;ENTER HERE FOR SSUB2 AND SADD2
SSUB2:  MOVEIA  F,F.MB              ;FLAG SUBTRACTION
SADD2:  SETZ    F,                  ;FLAG ADDITION
        MOVEI   T1,@1(E)            ;GET BP A
        MOVE    A1,0(T1)
        MOVEI   A2,@1(T1)
        DMOVE   C1,A1               ;BP C IS SAME AS BP A
        JRST    SADD

;ENTER HERE FOR SSUB3 AND SADD3
SSUB3:  MOVEIA  F,F.MB              ;FLAG SUBTRACTION
SADD3:  SETZ    F,                  ;FLAG ADDITION
        MOVEI   T1,@1(E)            ;GET BP A
        MOVE    A1,0(T1)
        MOVEI   A2,@1(T1)
        MOVEI   T1,@3(E)            ;GET BP C
        MOVE    C1,0(T1)
        MOVEI   C2,@1(T1)
SADD:   MOVEI   T1,@2(E)            ;GET BP B
        MOVE    B1,0(T1)
        MOVEI   B2,@1(T1)
        ;EA     E2,0(E)             ;PERFORMED BY EXTEND'S DISPATCH
        MOVEI   CP,@2(E2)           ;CHARACTER FOR "PLUS"
        MOVEI   CM,@3(E2)           ;CHARACTER FOR "MINUS"
        CAMN    CP,CM               ;UNSIGNED?
        JRST    SADDA               ;YES
        LDB     T1,A1               ;IS STRING A NEGATIVE?
        CAME    T1,CP
        TRO     F,F.MA              ;YES
        LDB     T1,B1               ;IS STRING B NEGATIVE?
        CAME    T1,CP
        TRC     F,F.MB              ;YES
SADDA:  MOVEI   R,@0(E2)            ;RADIX
        MOVEI   CZ,@1(E2)           ;CHARACTER FOR "ZERO"
        EA      KA,A1               ;COUNT OF DIGITS IN STRING A
        ;LDBX   KA,[POINT ^D19,A1,37]    ;ANOTHER WAY
        EA      KB,B1               ;COUNT OF DIGITS IN STRING B
        TRNU    F,F.MA+F.MB         ;BOTH STRINGS NEGATIVE?
        TRCA    F,F.MA+F.MB+F.MC ;YES, THEN C WILL BE NEGATIVE
        TRNN    F,F.MA+F.MB         ;JUST ONE STRING NEGATIVE?
        JRST    SADDJ

;HERE IF JUST ONE STRING IS NEGATIVE
;FIND WHICH NUMBER IS BIGGER
        CAMN    KA,KB               ;BOTH STRINGS SAME LENGTH?
        JRST    SADDE               ;YES
        CAML    KA,KB               ;WHICH STRING IS LONGER?
        JRST    SADDB               ;STRING A IS LONGER
        EXCH    KA,KB               ;STRING B IS LONGER, EXCHANGE THEM
        EXCH    A1,B1
        EXCH    A2,B2
```

```
        TRC     F,F.MA+F.MB
;HERE WHEN STRING A IS THE LONGER STRING
SADDB:  SUB     KA,KB           ;DIFFERENCE OF LENGTHS
;WE EXPECT THAT STRING A WILL HAVE LEADING ZEROS (KA OF THEM)
SADDC:  ILDB    T1,A1           ;GET NEXT DIGIT FROM STRING A
        SUB     T1,CZ           ;IS IT ZERO? (I.E. NOT A DIGIT)
        SKIPL   T1
        CAML    T1,R
        CAIA
        JUMPN   T1,SADDG
        SOJN    KA,SADDC        ;YES, LOOP

;HERE WHEN BOTH STRINGS ARE SAME LENGTH.
;CHECK WHICH STRING HAS LARGER VALUE.
SADDE:  ILDBW   T1,A1           ;GET NEXT DIGIT FROM STRING A
        JRST    SADDG           ;STRING EXHAUSTED
        SUB     T1,CZ           ;LEGAL DIGIT?
        SKIPL   T1
        CAML    T1,R
        SETZ    T1,             ;NO, MUST BE SPACE
        ILDB    T2,B1           ;GET NEXT DIGIT FROM STRING B
        SUB     T2,R            ;LEGAL DIGIT?
        SKIPL   T2
        CAML    T2,R
        SETZ    T2,             ;NO, MUST BE SPACE
        CAMN    T1,T2           ;SAME?
        JRST    SADDE           ;YES, KEEP LOOKING

;SEARCH ENDS UPON FINDING FIRST DIFFERENCE
        CAML    T1,T2           ;WHICH IS BIGGER?
        JRST    SADDI           ;A
        EXCH    A1,B1           ;B, EXCHANGE THEM
        EXCH    A2,B2
        TRC     F,F.MA+F.MB
;HERE WHEN STRING A HAS THE BIGGER VALUE
SADDI:  DECBP   B1              ;BACK UP
SADDG:  DECBP   A1
        EA      KA,A1           ;GET COUNT BACK
        EA      KB,B1
;HERE WHEN THERE IS AT MOST ONE STRING WHICH IS NEGATIVE.
;IF THERE IS, IN FACT, A NEGATIVE STRING THEN IT IS KNOWN THAT
;STRING A HAS THE LARGER MAGNITUDE.
SADDJ:  EA      KC,C1           ;COUNT OF DIGITS IN STRING C
        ADJBP   KA,A1           ;SKIP TO END OF STRING
        ADJBP   KB,B1
        ADJBP   KC,C1
        SETZ    C,              ;INITIAL VALUE OF CARRY

LOOP:   JUMPE   KA,LOOP3        ;BRANCH IF STRING A IS EXHAUSTED
        LDB     T1,A1           ;GET NEXT DIGIT FROM STRING A
        DECBP   A1              ;BACK UP
        SOS     KA
        SUB     T1,CZ           ;LEGAL DIGIT?
        SKIPL   T1
        CAML    T1,R
```

```
LOOP3:  SETZ    T1,             ;NO, MUST BE SPACE
        JUMPE   KB,LOOP4        ;BRANCH IF STRING B IS EXHAUSTED
        LDB     T2,B1           ;GET NEXT DIGIT FROM STRING B
        DECBP   B1              ;BACK UP
        SOS     KB
        SUB     T2,CZ           ;LEGAL DIGIT?
        SKIPL   T2
        CAML    T2,R
LOOP4:  SETZ    T2,             ;NO, MUST BE SPACE
        TRNE    F,F.MA+F.MB     ;ONE STRING NEGATIVE?
        MOVNS   T2              ;YES, SUBTRACT B FROM A
        ADD     T1,C            ;ADD CARRY
        ADD     T1,T2           ;ADD THE TWO DIGITS
        JUMPL   T1,LOOP1        ;BRANCH IF BORROW OUT
        CAML    T1,R            ;CARRY OUT?
        MOVEIA  C,1             ;YES
        MOVEIA  C,0             ;NO
        SUB     T1,R            ;YES
        JRST    LOOP2
LOOP1:  ADD     T1,R            ;BORROW OUT
        SETO    C,
LOOP2:  JUMPN   KC,LOOP6        ;BRANCH UNLESS C EXHAUSTED
        SKIPE   T1              ;OVERFLOW?
        TRO     F,F.OVR         ;YES
        JRST    LOOP5
LOOP6:  ADD     T1,CZ           ;CONVERT TO CHARACTER CODE
        DPB     T1,C1           ;STORE DIGIT
        DECBP   C1              ;BACK UP
        SOJN    KC,LOOP         ;LOOP
LOOP5:  JUMPN   KA,LOOP
        JUMPN   KB,LOOP
        CAMN    CP,CM           ;UNSIGNED?
        JRST    UNSIGN          ;YES
        TRZN    F,F.MA+F.MC     ;IS SIGN NEGATIVE?
        SKIPA   T1,CP           ;POSITIVE
        MOVE    T1,CM           ;NEGATIVE
        DPB     T1,C1           ;STORE SIGN
UNSIGN: SKIPL   C               ;OVERFLOW?
        TRNE    F,F.MA+F.OVR    ;UNSIGNED SUBTRACT OVERFLOW?
        MOVEI   C,1             ;YES
        ASH     C,^D31          ;LIGHT OVERFLOW IF CARRY
;DONE
```

3.3 ASMOVE - Arithmetic String MOVE (EXTEND 110)

The format of the argument block is:

```
E+0/    BYTE    (^D9)110(4)0(^D19)NSB
 +1/    addr of BP A
 +2/    addr of BP B
```

String B is moved to string A.

In the process of moving it, the string gets "normalized" (i.e. "spaces" are converted to true zeroes, and the sign byte is set to either true minus or true plus).

3.4 CNS - Convert Number System (EXTEND 111)

The format of the argument block is:

```
E+0/    BYTE    (^D9)111(4)0(^D19)NSBA
 +1/    addr of BP A
 +2/    addr of BP B
 +3/    NSBB
```

This instruction is exactly like ASMOVE except that it has two number system blocks. In the process of moving the string, it is converted from one number system to another. NSBA is used for string A, NSBB is used for string B.

Note that word 0 of NSBB is ignored (the radix). This instruction cannot be used to convert radixs, just character sets.

An overflow will result if an attempt is made to convert a signed negative number into an unsigned number.

This instruction is equivalent to IBM's PACK and UNPACK.

3.5 ASC{-,L,E,LE,A,GE,N,G}

Arithmetic String Compare (EXTEND 112). The format of the argument block is:

```
E+0/    BYTE    (^D9)112(4)AC(^D19)NSB
 +1/    addr of BP A
 +2/    addr of BP B
```

Compare string A with string B. Depending upon the result, the EXTEND instruction may skip.

Note that the AC field is decoded as an extension to the op-code:

AC	Name	Function
0	ASC	never skip
1	ASCL	skip if A<B
2	ASCE	skip if A=B
3	ASCLE	skip if A<=B
4	ASCA	always skip
5	ASCGE	skip if A>=B
6	ASCN	skip if A<>B
7	ASCG	skip if A>B

3.6 CTB - Convert To Binary (EXTEND 113)

```
E+0/    BYTE    (^D9)113(4)AC(^D19)NSB
 +1/    addr of BP
```

The numeric string is converted to a binary integer which is placed in the AC. If the resulting number is outside the range -2**31 to +2**31-1 then an overflow will result.

3.7 CFB - Convert From Binary (EXTEND 114)

```
E+0/    BYTE    (^D9)114(4)AC(^D19)NSB
 +1/    addr of BP
```

The binary integer in AC is converted to a numeric string.

4.0 PART IV: THE STACK OPTION

The stack instructions are considered to be an option to the basic machine.

The vast majority of these instructions begin with the letter "K" or "P". Those that begin with "K" are EOP's. They deal exclusively with items already on the stack. Those that begin with "P", however, are not EOP's. The operands for these instructions are not necessarily on the stack (at least not when the instruction begins).

Not all the EOP's occur in the same group (see section 1.10.16). As a rule of thumb (there are exceptions): Bits 22-24 of the group number tell you how many words are popped off the stack at the beginning of the instruction. Bits 25-27 of the group number tell you how many words are pushed back onto the stack at the conclusion of the instruction.

In the following discussion we shall often refer to items "K0" and "K1". By K0 we mean the top item on the stack. By K1 we refer to the second item on the stack. The term "item" should not be confused with "word". K0, for example, does not mean the top word, it's the top item. Each item consists of one or more words. When dealing with triple precision complex numbers, for example, each item is six words long.

4.1 K{ADD,SUB,BUS,MUL,DIV,VID,MOD,DOM}

These instructions pop two integers off the stack, perform an arithmetic operation upon them, and push the result back onto the stack. The KADD instruction, for example, adds the top two integers on the stack.

Name	Group	Func	What
KADD	21	6	K1+K0
KSUB	21	7	K1-K0
KMUL	21	10	K1*K0
KDIV	21	11	K1/K0
KBUS	21	12	K0-K1
KVID	21	13	K0/K1
KMOD	21	14	remainder K1/K0
KDOM	21	15	remainder K0/K1

Example: The following piece of code computes the expression "Y=3*(X+1)-4*(Z-1)":

```
        PUSHI   P,3
        PUSH    P,X
        PUSHI   P,1
        KADD    P,
        KMUL    P,
        PUSHI   P,4
        PUSH    P,Z
        PUSHI   P,1
        KSUB    P,
        KMUL    P,
        KSUB    P,
        POP     P,Y
```

4.2 K{F,DF,TF,C,DC,TC}{AD,SB,BS,MP,DV,VD}

These instructions perform floating point operations upon items on the stack. They pop two items off the stack, perform an operation, and push the result back onto the stack.

Name	Group	Func	What	Datum
KFAD	21	0	K1+K0	SINGLE FLOATING
KFSB	21	1	K1-K0	SINGLE FLOATING
KFMP	21	2	K1*K0	SINGLE FLOATING
KFDV	21	3	K1/K0	SINGLE FLOATING
KFBS	21	4	K0-K1	SINGLE FLOATING
KFVD	21	5	K0/K1	SINGLE FLOATING
KDFAD	42	0	K1+K0	DOUBLE FLOATING
KDFSB	42	1	K1-K0	DOUBLE FLOATING
KDFMP	42	2	K1*K0	DOUBLE FLOATING
KDFDV	42	3	K1/K0	DOUBLE FLOATING
KDFBS	42	4	K0-K1	DOUBLE FLOATING
KDFVD	42	5	K0/K1	DOUBLE FLOATING
KTFAD	63	0	K1+K0	TRIPLE FLOATING
KTFSB	63	1	K1-K0	TRIPLE FLOATING
KTFMP	63	2	K1*K0	TRIPLE FLOATING
KTFDV	63	3	K1/K0	TRIPLE FLOATING
KTFBS	63	4	K0-K1	TRIPLE FLOATING
KTFVD	63	5	K0/K1	TRIPLE FLOATING
KCAD	42	6	K1+K0	SINGLE COMPLEX
KCSB	42	7	K1-K0	SINGLE COMPLEX
KCMP	42	10	K1*K0	SINGLE COMPLEX
KCDV	42	11	K1/K0	SINGLE COMPLEX
KCBS	42	12	K0-K1	SINGLE COMPLEX
KCVD	42	13	K0/K1	SINGLE COMPLEX
KDCAD	76	0	K1+K0	DOUBLE COMPLEX
KDCSB	76	1	K1-K0	DOUBLE COMPLEX
KDCMP	76	2	K1*K0	DOUBLE COMPLEX
KDCDV	76	3	K1/K0	DOUBLE COMPLEX
KDCBS	76	4	K0-K1	DOUBLE COMPLEX
KDCVD	76	5	K0/K1	DOUBLE COMPLEX
KTCAD	77	0	K1+K0	TRIPLE COMPLEX
KTCSB	77	1	K1-K0	TRIPLE COMPLEX
KTCMP	77	2	K1*K0	TRIPLE COMPLEX
KTCDV	77	3	K1/K0	TRIPLE COMPLEX
KTCBS	77	4	K0-K1	TRIPLE COMPLEX
KTCVD	77	5	K0/K1	TRIPLE COMPLEX

4.3 {D,T,Q,H}{PUSH,POP}

These instructions push and pop a doubleword, tripleword, quadword, or hexword (respectively):

Name	Opcode	Words
DPUSH	424	2
TPUSH	425	3
QPUSH	414	4
HPUSH	416	6
DPOP	426	2
TPOP	427	3
QPOP	415	4
HPOP	741	6

Example: The following piece of code computes the expression "Y=X*Z" using double precision floating point:

```
        DPUSH   P,X
        DPUSH   P,Z
        KDFMP   P,
        DPOP    P,Y
```

4.4 FPUSHI - Floating PUSH Immediate (opcode 320)

E is shifted 14 bits to the left and the result is pushed onto the stack. Thus the instruction "FPUSHI P,X" is equivalent to the sequence:

```
FMOVEI  T1,X
PUSH    P,T1
```

4.5 P{ADD,SUB,MUL,DIV}[I]

Name	Opcode	Function
PADD	160	K0=K0+C(E)
PADDI	161	K0=K0+E
PSUB	162	K0=K0-C(E)
PSUBI	163	K0=K0-E
PMUL	170	K0=K0*C(E)
PMULI	171	K0=K0*E
PDIV	172	K0=K0/C(E)
PDIVI	173	K0=K0/E

The instruction "PSUB P,X", for example, is equivalent to the sequence:

```
PUSH    P,X
KSUB    P,
```

Example: The following code will recompute the expression from section 4.1: Y=3*(X+1)-4*(Z-1)

```
PUSH    P,X
PADDI   P,1
PMULI   P,3
PUSH    P,Z
PSUBI   P,1
PMULI   P,4
KSUB    P,
POP     P,Y
```

Note how much shorter this version is.

4.6 PF{AD,SB,MP,DV}[I]

These instructions are similar to the previous group except that they use single precision reals instead of integers.

Name	Opcode	Function
PFAD	450	K0=K0+C(E)
PFADI	451	K0=K0+E
PFSB	474	K0=K0-C(E)
PFSBI	475	K0=K0-E
PFMP	700	K0=K0*C(E)
PFMPI	701	K0=K0*E
PFDV	704	K0=K0/C(E)
PFDVI	705	K0=K0/E

Example: The instruction "PFADI P,X" is equivalent to the sequence:

```
FPUSHI  P,X
KFAD    P,
```

4.7 PADDM - Popping ADD To Memory (opcode 462)

Pop a word off the stack and add it to C(E) (i.e. C(E)=C(E)+K0). Thus the instruction "PADDM P,X" is equivalent to the sequence:

```
POP     P,T1
ADDM    T1,X
```

Note that the instruction uses read-pause-write.

4.8 PFADM - Popping Floating Add To Memory (opcode 503)

This instruction is similar to PADDM except that it uses floating point.

4.9 PUSHZ (opcode 431)

If E is positive, push E words of zeros onto the stack. If E is negative, PUSHZ is the same as ADJSP.

4.10 PLDB - Popping LoaD Byte (opcode 420)

Load a byte and push it onto the stack.

Example: The instruction "PLDB P,X" is equivalent to the sequence:

```
LDB     T1,X
PUSH    P,T1
```

4.11 PDPB - Popping DePosit Byte (opcode 421)

Pop a word off the stack and do a deposit byte.

Example: The instruction "PDPB P,X" is equivalent to the sequence:

```
POP     P,T1
DPB     T1,X
```

4.12 PMOVEM - Popping MOVE To Memory (opcode 247)

Store the top word of the stack at the location specified by E. Do not pop the word off the stack. The word remains on the top of the stack and the stack pointer is not changed.

Example:

```
PMOVEM  AC,E
```

is equivalent to:

```
MOVE    T1,0(AC)
MOVEM   T1,E
```

4.13 PUPJ (EOP 11-3)

This instruction is a cross between a PUSHJ and a POPJ.

The top word on the stack is interpreted as the address of a subroutine. POP the address off the stack and PUSHJ to it. The net effect is that the PC is exchanged with the top word on the stack.

This instruction is quite useful when implementing coroutines.

4.14 K{-,D,T}SKP{-,L,E,LE,A,GE,N,G}

Pop one, two, or three words off the stack and skip depending on their value:

Name	EOP	Words	Skip if
KSKP	10-0	1	never
KSKPL	10-1	1	K0<0
KSKPE	10-2	1	K0=0
KSKPLE	10-3	1	K0<=0
KSKPA	10-4	1	always
KSKPGE	10-5	1	K0>=0
KSKPN	10-6	1	K0<>0
KSKPG	10-7	1	K0>0
KDSKP	20-0	2	never
KDSKPL	20-1	2	K0<0
KDSKPE	20-2	2	K0=0
KDSKPLE	20-3	2	K0<=0
KDSKPA	20-4	2	always
KDSKPGE	20-5	2	K0>=0
KDSKPN	20-6	2	K0<>0
KDSKPG	20-7	2	K0>0
KTSKP	30-0	3	never
KTSKPL	30-1	3	K0<0
KTSKPE	30-2	3	K0=0
KTSKPLE	30-3	3	K0<=0
KTSKPA	30-4	3	always
KTSKPGE	30-5	3	K0>=0
KTSKPN	30-6	3	K0<>0
KTSKPG	30-7	3	K0>0

Note that KSKP, KDSKP, and KTSKP are not no-ops. They pop words off the stack.

4.15 K{Q,H}SKP{-,E,A,N}

Pop four or six words off the stack and skip depending on their value:

Name	EOP	Words	Skip if
KQSKP	40-0	4	never
KQSKPE	40-1	4	K0=0
KQSKPA	40-2	4	always
KQSKPN	40-2	4	K0<>0
KHSKP	60-0	6	never
KHSKPE	60-1	6	K0=0
KHSKPA	60-2	6	always
KHSKPN	60-2	6	K0<>0

Note that K{Q,H}SKP{L,LE,G,GE} are not supported. These concepts are not meaningful for complex numbers.

4.16 Stack Boolean

The stack option supports all 16 of the boolean functions:

Name	EOP	K0=0011 K1=0101
KSETZ	21-20	0000
KAND	21-21	0001
KANDC1	21-22	0010
KSET0	21-23	0011
KANDC0	21-24	0100
KSET1	21-25	0101
KXOR	21-26	0110
KOR	21-27	0111
KNOR	21-30	1000
KEQV	21-31	1001
KSETC1	21-32	1010
KORC1	21-33	1011
KSETC0	21-34	1100
KORC0	21-35	1101
KNAND	21-36	1110
KSETO	21-37	1111

4.17 K{-,D,T,C,DC,TC}NEG

Negate the top item on the stack:

Name	EOP	Datum
KNEG	11-0	single
KDNEG	22-0	double
KTNEG	33-0	triple
KCNEG	22-1	complex single
KDCNEG	44-0	complex double
KTCNEG	66-0	complex triple

4.18 Stack Conversions

The stack option supports 12 instructions for converting data types: K{I,F,DF,TF}{I,F,DF,TF}. Any of the four supported data types can be converted to any of the other types.

Supported Types:

I - Integer
F - Floating (single precision)
DF - Double Floating
TF - Triple Floating

Example: the instruction KDFI converts from double precision floating point to integer.

Name	EOP	Notes
KIF	11-1	
KIDF	12-0	
KITF	13-0	
KFI	11-2	*
KFDF	12-1	
KFTF	13-1	
KDFI	21-16	*
KDFF	21-17	rounded
KDFTF	23-0	
KTFI	31-0	*
KTFF	31-1	rounded
KTFDF	32-0	rounded

* = Sets one of the overflow bits and/or traps if the conversion is not possible.

Note that double integers are not supported by any stack instruction.

4.19 SWAP (EOP Group 73)

This class of instruction takes up an entire EOP group.

Bits 30-37 are decoded as follows:

```
!3!3   3!3!3   3!
!0!1   3!4!5   7!
+-+-----+-+-----+
!0!  N  !0!  M  !
+-+-----+-+-----+
```

The purpose of this instruction is to swap the top two items on the stack. The top item on the stack is taken as being N words long. The next item is taken as being M words long.

Note that if either N or M is zero, the instruction is a no-op.

Note that bits 30 and 34 are ignored. The largest item you can swap is 7 words long. This instruction is difficult to implement in microcode, and we deliberately wish to restrict the size of the largest item. In practice, we believe the largest item will be six words long (a complex tripleword).

4.20 KILL (EOP Group 72)

This class of instruction takes up an entire EOP group.

Bits 30-37 are decoded as follows:

```
!3     3!3     3!
!0     3!4     7!
+-------+-------+
!   N   !   M   !
+-------+-------+
```

This instruction is similar to SWAP. The top item on the stack is taken as being N words long. The second item is taken as being M words long. The purpose of the instruction is to delete the second item from the stack.

Note that if M=0, the instruction is a no-op. If N=0, the instruction is equivalent to "ADJSP P,-M".

Among other things, this instruction is used to take the imaginary part of a complex number (deleting the real part).

5.0 APPENDIXS

5.1 Effective Address Calculation

The microcode uses 4 registers to compute the effective address: IR, E, MA, and MB. The algorithm is as follows:

```
BEGIN:   ;ENTER HERE WITH THE INSTRUCTION IN IR AND THE ADDRESS
         ;THAT THE INSTRUCTION WAS FETCHED FROM IN MA

         IF IR(15)=1 GOTO MODE1
         E(16:37)=IR(16:37)
         E(0:15)=0
         GOTO EXIT

MODE1:   E(23:37)=IR(23:37)
         E(0:22)=IR(23)
         IF IR(17:22)=0 GOTO NOX
         IF IR(17:22)=17 GOTO RELX
         E=E+MEMORY(IR(17:22))
         GOTO NOX

RELX:    E=E+MA
NOX:     IF IR(16)=0 GOTO EXIT

I:       MA=E
         MB=MEMORY(MA)
         IF MB(0)=1 GOTO IMODE1
         E=MB
         GOTO EXIT

IMODE1:  E(12:37)=MB(12:37)
         E(0:11)=MB(12)
         IF MB(2:5)=0 GOTO INOX1
         IF MB(2:5)=17 GOTO IRELX1
         E=E+MEMORY(IR(2:5))
         GOTO INOX1

IRELX1:  E=E+MA
INOX1:   IF MB(6:11)=0 GOTO INOX2
         IF MB(6:11)=17 GOTO IRELX2
         E=E+MEMORY(IR(6:11))
         GOTO INOX2

IRELX2:  E=E+MA
INOX2:   IF MB(1)=1 GOTO I

EXIT:    ;THE EFFECTIVE VIRTUAL ADDRESS IS NOW IN REGISTER E
```

5.2 Statistics

The single most important aspect of any machine is the method of calculating the effective address. Before settling upon the current scheme we did a fair amount of analysis of existing PDP10 programs. The survey presented here is one conducted upon the first K of FILFND (to be precise, the first 1039 instructions in 701 FILFND). We believe this to be a representative sample.

Each of the 1039 instructions was divided into one of 32 categories (numbered in octal from 0 to 37). The first digit of the category number summarises the I and X fields of the instruction:

Code	Means
0	No index register, not indirect
1	Index register only
2	Indirect only
3	Both index and indirect

The second digit of the category number summarises the Y field:

Code	Y field contained
0	An AC number
1	An unrelocated expression (absolute)
2	An address in the LOWSEG
3	An address in the HISEG which is "close" (within 2**12)
4	An address in the HISEG which is "far" (outside 2**12)
5	An address in the HISEG which is external (don't know how close)
6	The Y field is ignored (e.g. SETZ)
7	The address of a literal

Thus codes 0-1 are absolute. Codes 2-5 and 7 are relocatable.

Of the 32 categories, only 10 were actually observed to occur:

Type	Refs	Percent
00	149	14%
01	118	11%
02	46	4%
03	231	22%
05	202	19%
06	45	4%
07	7	1%
11	210	20%

12	30	3%
33	1	0%

	1039	

Of the 1039 instructions, 798 had a zero in the first digit. I.E. 77% are neither indexed nor indirect.

Of these 798, 486 were relocatable (61%)

Of these 486, 46 were in the LOWSEG (9%)

Note that FILFND is only 6416 octal words big (including literals). Thus all PC relative references are close (i.e. within plus or minus 2**12). There were no type ?4 references at all. FILFND is considered a large module.

Of the 202 references to HISEG externals, there were only 83 externals involved. I.E. There are 2.43 references to each one. This should cut down on the number of links.

Of the 46 unindexed references to the lowseg, there were only 22 lowseg locations involved (2.09 references to each location).

Of the 30 indexed references to the lowseg, only 18 links are required. There are 1.67 references to each link. This ratio is different from that for unindexed lowseg references because there would have to be a seperate link each time a different index AC is used.

Of these indexed references to the lowseg, there are 15 references to the JBT tables (half the type 12 references). These 15 references require 11 links (61% of the type 12 links). A large portion of these links could be avoided if the JBT items were moved to the PDB. That is to say you can't fit all the JBT tables in the lowest 2**12 words memory, but you can indeed fit all of JBTPDB. References to these data items would then take two words each (one instruction to reference JBTPDB, and one instruction to reference the data item itself). Two words isn't very good but the other approach isn't much better. It too takes about two words (one for the instruction that references the JBT table, and one for the link). Few of the links are shared, each reference needs a link of its own.

Of the 83 links to hiseg externals, 13 are to "literal byte pointers" (22 references to 13 links). By "literal byte pointer" we mean items like UNYK4S. The main reason these byte pointers were coded as globals in COMMOD instead of making them into local literals was to save typing. On the new machine, however, it might be better to keep them as local literals. One way or another you're going to tie up a word of memory: either for the local literal or for the link to the external. Given a choice, the local literal is better because it executes faster. We assume there will be a mechanism for intermodule literal pools. A literal in one module could be shared by another module if the other module was close. If the other module were not close, LINK would build two copies of the literal (one for each module). Alot of typing could be saved if these global literals could be referenced by name.

Of the 202 references to hiseg externals (type 05), 33 of them were to CPOPJ1. All of these references would be eliminated by the proposed change to the POPJ instruction.

Notes:

1. FILFND does not have a lowseg so all lowseg items are external. These are counted as type 02 not type 05.

2. References to .CP??? are counted as lowseg references despite the fact that the address involved is above 400000. The address is, however, below MONORG.

3. All literals are counted as "close". FILFND is smaller than 2**12 words so there would be plenty of room for the literals. There was exactly one reference to each literal.

4. We have assumed all externals are far but this isn't necessarily true. Some might reference a close module.

5. The number of references to each link would no doubt increase if a larger sample were taken. 1039 instructions, however, isn't shabby.

6. References to links appear to be fairly localized. A given page of listing might have numerous references to a particular link, where as the entire remainder of the module might have few if any.

7. It would be interesting to do a study to find out which types are executed most frequently.

Conclusions:

These conclusions are based on the existing code and do not assume the usage of any of the new instructions:

1. If the entire monitor (HISEG and LOWSEG) were loaded in the first 2**18 of memory (as it was in 701) then the only reference types that require links are 12 and 33. There would be 31 references to 19 links. The monitor would increase in size by 19 words per 1039 instructions (1.8%).

But because the word size is smaller, the net number of bits would actually decrease by 9.5%.

2. If the monitor's HISEG were made position independent and placed above 2**18 then reference types 05, 12, and 33 would require links. There would be 233 references to 102 links. The monitor size would increase by 102 words per 1039, or 9.8%. The size in bits, however, would decrease by 2.4%.

3. (The worst case) If both the monitor's HISEG and LOWSEG were loaded above 2**18, then reference types 02, 05, 12 and 33 would require links. There would be 279 references to 124 links. The monitor size would increase by 124 words per 1039, or 11.9%. The size in bits, however, would decrease by 0.5%.

5.3 Alternatives

The single most important aspect of any machine is the method of calculating the effective address. Before settling upon the current scheme, many alternatives were considered. Each has its own trade offs. Some schemes are particularly good for certain types of addressing, but not so good for others. The goal is to find a scheme that works fairly well for all the common addressing modes.

There are many objections to the scheme we have chosen. We shall discuss two of them:

5.3.1

The first drawback of the current scheme is that it can't index into an array whoose position is PC relative. Rather, it can do the index but a link word is required.

At first glance this seems like a serious drawback. The more we think about it, however, the less serious it seems.

Consider the alternatives: Instead of using AC 17 for position independent addressing we could have invented a new bit:

```
!1!1!1!2     2!2                      3!
!5!6!7!0     3!4                      7!
+-+-+-+-------+-----------------------+
!1!R!I!  X    !          Y            !
+-+-+-+-------+-----------------------+
   ^          !       12 bits         !
   !
   !
   +-- new bit
```

The new bit, iff on, indicates position independent addressing. The current PC (or MA) is added to the effective address. The cost of this bit, however, is enormous. It chops the Y field from 13 bits to 12 bits (a 12 bit Y field means a relative address of plus or minus 2**11).

One bit may not sound like much but this particular bit is a crucial one. The size of the average REL file is somewhere between 2**11 and 2**12. Rather: files larger than 2**12 are quite rare but files larger than 2**11 are common.

Moreover, we do not expect that the need for position independent indexing will be a great one:

It is expected that only subroutines will use position independent addressing. The main program will be loaded in the lowest 2**18 of memory. Subroutines will be loaded above 2**18. Subroutines will use position independent addressing, the main program will not.

It is also expected that very few subroutines will have arrays of their own. Most subroutines will have arrays passed to them as arguments (actually only the address of the array is passed). Thus the subroutine can't do normal indexing anyway. Position independent indexing is not needed.

There are, however, a few subroutines that do have arrays of their own. But efficiency dictates that the array should be allocated on the stack:

Consider a program with X subroutines. Assume that the branching factor is N (i.e. that the typical subroutine calls N other subroutines). The value of N varies greatly but is typically greater than 2. At any given instant only LOGn(X) subroutines are active (LOG base N of X). If the program allocates the arrays statically, then the amount of space used is Y*X (where Y is the average array space per subroutine). If, however, the arrays are allocated dynamically then the amount of space used is Y*LOGn(X). This can result is a tremendous savings in space. Thus it is expected that most subroutines will allocate their arrays dynamically (this also means that the size of the array can be passed to the subroutine as an argument. The subroutine doesn't have to reserve extra space for the worst possible case). Given that the arrays are allocated dynamically, the subroutine cannot do normal indexing. The ability to do PC relative indexing would not be of any help.

"Own variables" are the one exception to this rule. Own variables are those variables belonging to a subroutine which are preserved from one invocation to the next.

Clearly own variables cannot be allocated on the stack. But own variables are fairly rare and own arrays are even rarer.

5.3.2

The second drawback of the current effective address scheme is that we "loose" a register (register 17). The loss is a serious one. We are not proud of it.

May we point out, however, that even the PDP10 looses a register when it does position independent addressing. Consider the following PDP10 program (a typical case):

```
        MOVSI   T1,(JRST (X))
        JSP     X,T1
        PHASE   0
        ...
FOO:    ...
        JRST    FOO(X)
        ...
        DEPHASE
```

Register X is dedicated for the soul purpose of position independent addressing. It cannot be used for anything else. It is effectively lost.

Note that on the new machine, however, we don't loose the register completely. Its just that we can't use it for indexing. It's still available for all other uses.

Moreover, consider the dilemma faced by DDT. On the PDP10 DDT tries to type out the instruction:

```
        JRST    n(X)
```

DDT does not know what value register X will have when the instruction is ultimately executed. Therefore DDT can't type the symbolic name of the location being referenced.

On the new machine, however, DDT knows exactly what is meant by

```
        JRST    n(17)
```

The usage of register 17 is precisely defined by the hardware and DDT knows this. DDT can therefore convert to a symbolic name.

OPCODE List

The following is a list of opcode assignments (sorted by number).

Note: In cases where several mneumonics are listed for the same opcode, the one which is listed first is the prefered name.

Opcode	PDP10	New Machine (if different than PDP10)
000	illegal	
1-30	LUUO	
031	LUUO	XOP (exec only)
032	LUUO	UMAP (exec only)
033	LUUO	MAP (exec only)
034	LUUO	PHYLDB (exec only)
035	LUUO	PHYDPB (exec only)
036	LUUO	ULDB (exec only)
037	LUUO	UDPB (exec only)
40-77	MUUO	
100	UJEN	IDV
101	-	IDVI
102	GFAD	TFAD
103	GFSB	TFSB
104	JSYS	LDBX
105	ADJSP	
106	GFMP	TFMP
107	GFDV	TFDV
110	DFAD	
111	DFSB	
112	DFMP	
113	DFDV	
114	DADD	
115	DSUB	
116	DMUL	
117	DDIV	
120	DMOVE	
121	DMOVN	
122	FIX	
123	EXTEND	EA
124	DMOVEM	
125	DMOVNM	
126	FIXR	
127	FLTR	FLT,FLTR
130	UFA	TMOVE
131	DFN	TMOVEM
132	FSC	
133	IBP+ADJBP	
134	ILDB	
135	LDB	
136	IDPB	
137	DPB	
140	FAD	BUS

OPCODE LIST (SORTED BY NUMBER)

141	FADL	BUSI
142	FADM	BUSM
143	FADB	BUSB
144	FADR	FAD,FADR
145	FADRI	FADI,FADRI
146	FADRM	FADM,FADRM
147	FADRB	FADB,FADRB
150	FSB	IVID
151	FSBL	IVIDI
152	FSBM	IVIDM
153	FSBB	IVIDB
154	FSBR	FSB,FSBR
155	FSBRI	FSBI,FSBRI
156	FSBRM	FSBM,FSBRM
157	FSBRB	FSBB,FSBRB
160	FMP	PADD
161	FMPL	PADDI
162	FMPM	PSUB
163	FMPB	PSUBI
164	FMPR	FMP,FMPR
165	FMPRI	FMPI,FMPRI
166	FMPRM	FMPM,FMPRM
167	FMPRB	FMPB,FMPRB
170	FDV	PMUL
171	FDVL	PMULI
172	FDVM	PDIV
173	FDVB	PDIVI
174	FDVR	FDV,FDVR
175	FDVRI	FDVI,FDVRI
176	FDVRM	FDVM,FDVRM
177	FDVRB	FDVB,FDVRB
200	MOVE	MOVE,SETM,SETMB
201	MOVEI	MOVEI,SETMI,SETZ,SETZI
202	MOVEM	MOVEM,SETAM,SETAB
203	MOVES	MOVES,SKIP,HLLS,HRRS
204	MOVS	
205	MOVSI	
206	MOVSM	
207	MOVSS	
210	MOVN	
211	MOVNI	
212	MOVNM	
213	MOVNS	
214	MOVM	
215	MOVMI	
216	MOVMM	
217	MOVMS	
220	IMUL	
221	IMULI	
222	IMULM	
223	IMULB	
224	MUL	
225	MULI	
226	MULM	

227	MULB	
230	IDIV	
231	IDIVI	
232	IDIVM	IDIVM,IDVM
233	IDIVB	
234	DIV	
235	DIVI	
236	DIVM	
237	DIVB	
240	ASH	
241	ROT	
242	LSH	
243	JFFO	
244	ASHC	
245	ROTC	
246	LSHC	
247	-	PMOVEM
250	EXCH	
251	BLT	DPBI
252	AOBJP	AOBJGE,AOBJP
253	AOBJN	AOBJL,AOBJN
254	JRST	(an ACOP)
255	JFCL	BAOS (an ACOP)
256	XCT	(an ACOP)
257	MAP	ILDBA
260	PUSHJ	
261	PUSH	
262	POP	
263	POPJ	
264	JSR	ILDBW
265	JSP	
266	JSA	
267	JRA	
270	ADD	
271	ADDI	
272	ADDM	
273	ADDB	
274	SUB	
275	SUBI	
276	SUBM	
277	SUBB	
300	CAI	CAI,TRN,TLN,TDN,TSN,JUMP,CAM,SETA,SETAI,SETMM,BLN,BRN DSKP,TSKP,QSKP,HSKP,JFCL
301	CAIL	
302	CAIE	
303	CAILE	
304	CAIA	CAIA,TRNA,TLNA,TDNA,TSNA,CAMA,BLNA,BRNA, DSKPA,TSKPA,QSKPA,HSKPA
305	CAIGE	
306	CAIN	
307	CAIG	
310	CAM	MOVEIA
311	CAML	
312	CAME	

313	CAMLE	
314	CAMA	PUSHI
315	CAMGE	
316	CAMN	
317	CAMG	
320	JUMP	FPUSHI
321	JUMPL	
322	JUMPE	
323	JUMPLE	
324	JUMPA	ILDBL
325	JUMPGE	
326	JUMPN	
327	JUMPG	
330	SKIP	IDPBA
331	SKIPL	SKIPL,DSKPL,TSKPL,QSKPL,HSKPL
332	SKIPE	
333	SKIPLE	
334	SKIPA	
335	SKIPGE	SKIPGE,DSKPGE,TSKPGE,QSKPGE,HSKPGE
336	SKIPN	
337	SKIPG	
340	AOJ	
341	AOJL	
342	AOJE	
343	AOJLE	
344	AOJA	
345	AOJGE	
346	AOJN	
347	AOJG	
350	AOS	
351	AOSL	
352	AOSE	
353	AOSLE	
354	AOSA	
355	AOSGE	
356	AOSN	
357	AOSG	
360	SOJ	
361	SOJL	
362	SOJE	
363	SOJLE	
364	SOJA	
365	SOJGE	
366	SOJN	
367	SOJG	
370	SOS	
371	SOSL	
372	SOSE	
373	SOSLE	
374	SOSA	
375	SOSGE	
376	SOSN	
377	SOSG	
400	SETZ	EOP

OPCODE LIST (SORTED BY NUMBER)

401	SETZI	FMOVEI
402	SETZM	
403	SETZB	
404	AND	
405	ANDI	
406	ANDM	
407	ANDB	
410	ANDCA	
411	ANDCAI	
412	ANDCAM	
413	ANDCAB	
414	SETM	QPUSH
415	SETMI	QPOP
416	SETMM	HPUSH
417	SETMB	IDPBW
420	ANDCM	PLDB
421	ANDCMI	PDPB
422	ANDCMM	
423	ANDCMB	
424	SETA	DPUSH
425	SETAI	TPUSH
426	SETAM	DPOP
427	SETAB	TPOP
430	XOR	XOR,TDC
431	XORI	PUSHZ
432	XORM	
433	XORB	
434	OR	OR,TDO
435	ORI	IDPBL
436	ORM	
437	ORB	
440	ANDCB	NOR,ANDCB
441	ANDCBI	NORI,ANDCBI
442	ANDCBM	NORM,ANDCBM
443	ANDCBB	NORB,ANDCBB
444	EQV	
445	EQVI	
446	EQVM	
447	EQVB	
450	SETCA	PFAD
451	SETCAI	PFADI
452	SETCAM	
453	SETCAB	
454	ORCA	
455	ORCAI	
456	ORCAM	
457	ORCAB	
460	SETCM	
461	SETCMI	
462	SETCMM	PADDM
463	SETCMB	
464	ORCM	
465	ORCMI	
466	ORCMM	

467	ORCMB	
470	ORCB	NAND,ORCB
471	ORCBI	NANDI,ORCBI
472	ORCBM	NANDM,ORCBM
473	ORCBB	NANDB,ORCBB
474	SETO	PFSB
475	SETOI	PFSBI
476	SETOM	
477	SETOB	
500	HLL	
501	HLLI	
502	HLLM	
503	HLLS	PFADM
504	HRL	
505	HRLI	
506	HRLM	
507	HRLS	
510	HLLZ	
511	HLLZI	
512	HLLZM	
513	HLLZS	
514	HRLZ	
515	HRLZI	
516	HRLZM	
517	HRLZS	
520	HLLO	
521	HLLOI	
522	HLLOM	
523	HLLOS	
524	HRLO	
525	HRLOI	
526	HRLOM	
527	HRLOS	
530	HLLE	
531	HLLEI	
532	HLLEM	
533	HLLES	
534	HRLE	
535	HRLEI	
536	HRLEM	
537	HRLES	
540	HRR	
541	HRRI	
542	HRRM	
543	HRRS	LDBA
544	HLR	
545	HLRI	
546	HLRM	
547	HLRS	
550	HRRZ	
551	HRRZI	
552	HRRZM	
553	HRRZS	
554	HLRZ	

555	HLRZI	
556	HLRZM	
557	HLRZS	
560	HRRO	
561	HRROI	
562	HRROM	
563	HRROS	
564	HLRO	
565	HLROI	
566	HLROM	
567	HLROS	
570	HRRE	
571	HRREI	
572	HRREM	
573	HRRES	
574	HLRE	
575	HLREI	
576	HLREM	
577	HLRES	
600	TRN	TRU
601	TLN	TLU
602	TRNE	
603	TLNE	
604	TRNA	TRNU
605	TLNA	TLNU
606	TRNN	
607	TLNN	
610	TDN	TDU
611	TSN	LDBW
612	TDNE	
613	TSNE	
614	TDNA	TDNU
615	TSNA	LDBL
616	TDNN	
617	TSNN	
620	TRZ	TRZ,ANDCMI
621	TLZ	
622	TRZE	
623	TLZE	
624	TRZA	
625	TLZA	
626	TRZN	
627	TLZN	
630	TDZ	TDZ,ANDCM
631	TSZ	
632	TDZE	
633	TSZE	
634	TDZA	
635	TSZA	
636	TDZN	
637	TSZN	
640	TRC	TRC,XORI
641	TLC	
642	TRCE	

643	TLCE	
644	TRCA	
645	TLCA	
646	TRCN	
647	TLCN	
650	TDC	DPBA
651	TSC	
652	TDCE	
653	TSCE	
654	TDCA	
655	TSCA	
656	TDCN	
657	TSCN	
660	TRO	TRO,ORI
661	TLO	
662	TROE	
663	TLOE	
664	TROA	
665	TLOA	
666	TRON	
667	TLON	
670	TDO	DPBW
671	TSO	
672	TDOE	
673	TSOE	
674	TDOA	
675	TSOA	
676	TDON	
677	TSON	
700	-	PFMP
701	-	PFMPI
702	-	BLNE
703	-	BRNE
704	-	PFDV
705	-	PFDVI
706	-	BLNN
707	-	BRNN
710	-	BLZ
711	-	BRZ
712	-	BLZE
713	-	BRZE
714	-	BLZA
715	-	BRZA
716	-	BLZN
717	-	BRZN
720	-	BLC
721	-	BRC
722	-	BLCE
723	-	BRCE
724	-	BLCA
725	-	BRCA
726	-	BLCN
727	-	BRCN
730	-	BLO

OPCODE LIST (SORTED BY NUMBER)

731	-	BRO
732	-	BLOE
733	-	BROE
734	-	BLOA
735	-	BROA
736	-	BLON
737	-	BRON
740	-	DPBL
741	-	HPOP
742	-	QMOVE
743	-	QMOVEM
744	-	HMOVE
745	-	HMOVEM
746	-	CAD
747	-	CSB
750	-	CMP
751	-	CDV
752	-	DCAD
753	-	DCSB
754	-	DCMP
755	-	DCDV
756	-	TCAD
757	-	TCSB
760	-	TCMP
761	-	TCDV
762-776	-	spare
777	illegal	

Opcode 031: (XOP - Exec only)

AC	Name
0	XJSR
1	XRET
2	XPCW
3	XDIS
4-17	spare

Opcode 254:

AC	Name
0	JRST
1	PORTAL
2	DSKPE
3	DSKPLE

OPCODE LIST (SORTED BY NUMBER)

4	HALT
5	JSR
6	DSKPN
7	DSKPG
10	QSKPE
11	QSKPN
12	TSKPE
13	TSKPLE
14	HSKPE
15	HSKPN
16	TSKPN
17	TSKPG

Opcode 255:

AC	Name
0	BAOS
1	BAOSL
2	BAOSE
3	BAOSLE
4	BAOSA
5	BAOSGE
6	BAOSN
7	BAOSG
10	BSOS
11	BSOSL
12	BSOSE
13	BSOSLE
14	BSOSA
15	BSOSGE
16	BSOSN
17	BSOSG

Opcode 256:

AC	Name
0	XCT
1	SSTEP
2	spare
3	BLT
4	BBLT
5	SETCMM
6	DECBP
7	EXTEND

OPCODE LIST (SORTED BY NUMBER)

10	IBPA
11	IBPW
12	IBPL
13	DSETZM
14	TSETZM
15	QSETZM
16	HSETZM
17	spare

Opcode 400: (EOP)

Group	Func	Name
0	0	spare
0	1	DSETZ
0	2	TSETZ
0	3	QSETZ
0	4	HSETZ
0	5	spare
0	6	spare
0	7	NEG
0	10	DNEG
0	11	TNEG
0	12	CNEG
0	13	DCNEG
0	14	TCNEG
0	15-17	spare
0	20	CIDI
0	21	CIF
0	22	CIDF
0	23	CITF
0	24	CDII
0	25	CDIF
0	26	CDIDF
0	27	CDITF
0	30	CFI
0	31	CFDI
0	32	CFDF
0	33	CFTF
0	34	CDFI
0	35	CDFDI
0	36	CDFF
0	37	CDFTF
0	40	CTFI
0	41	CTFDI
0	42	CTFF
0	43	CTFDF
10	0	KSKP
10	1	KSKPL
10	2	KSKPE

OPCODE LIST (SORTED BY NUMBER)

10	3	KSKPLE
10	4	KSKPA
10	5	KSKPGE
10	6	KSKPN
10	7	KSKPG
11	0	KNEG
11	1	KIF
11	2	KFI
11	3	PUPJ
12	0	KIDF
12	1	KFDF
13	0	KITF
13	1	KFTF
20	0	KDSKP
20	1	KDSKPL
20	2	KDSKPE
20	3	KDSKPLE
20	4	KDSKPA
20	5	KDSKPGE
20	6	KDSKPN
20	7	KDSKPG
21	0	KFAD
21	1	KFSB
21	2	KFMP
21	3	KFDV
21	4	KFBS
21	5	KFVD
21	6	KADD
21	7	KSUB
21	10	KMUL
21	11	KDIV
21	12	KBUS
21	13	KVID
21	14	KMOD
21	15	KDOM
21	16	KDFI
21	17	KDFF
21	20	KSETZ
21	21	KAND
21	22	KANDC1
21	23	KSET0
21	24	KANDC0
21	25	KSET1
21	26	KXOR
21	27	KOR
21	30	KNOR
21	31	KEQV
21	32	KSETC1
21	33	KORC1
21	34	KSETC0
21	35	KORC0
21	36	KNAND
21	37	KSETO
22	0	KDNEG

OPCODE LIST (SORTED BY NUMBER)

22	1	KCNEG
23	0	KDFTF
30	0	KTSKP
30	1	KTSKPL
30	2	KTSKPE
30	3	KTSKPLE
30	4	KTSKPA
30	5	KTSKPGE
30	6	KTSKPN
30	7	KTSKPG
31	0	KTFI
31	1	KTFF
32	0	KTFDF
33	0	KTNEG
40	0	KQSKP
40	1	KQSKPE
40	2	KQSKPA
40	3	KQSKPN
42	0	KDFAD
42	1	KDFSB
42	2	KDFMP
42	3	KDFDV
42	4	KDFBS
42	5	KDFVD
42	6	KCAD
42	7	KCSB
42	10	KCMP
42	11	KCDV
42	12	KCBS
42	13	KCVD
44	0	KDCNEG
60	0	KHSKP
60	1	KHSKPE
60	2	KHSKPA
60	3	KHSKPN
63	0	KTFAD
63	1	KTFSB
63	2	KTFMP
63	3	KTFDV
63	4	KTFBS
63	5	KTFVD
66	0	KTCNEG
72	-	KILL
73	-	SWAP
74	-	RST
75	-	SAV
76	0	KDCAD
76	1	KDCSB
76	2	KDCMP
76	3	KDCDV
76	4	KDCBS
76	5	KDCVD
77	0	KTCAD
77	1	KTCSB

77	2	KTCMP
77	3	KTCDV
77	4	KTCBS
77	5	KTCVD

EXTEND: (Opcode 256-7)

Op	Name
1	SMOVE
2	BSMOVE
3	CONCAT
4	SRCHE
5	SRCHN
6	SPATE
7	SPATN
10	SCOMP
11	SCOMPL
12	SCOMPE
13	SCOMPLE
14	SCOMPA
15	SCOMPGE
16	SCOMPN
17	SCOMPG
100	SADD2
101	SADD3
102	SSUB2
103	SSUB3
104	SMUL2
105	SMUL3
106	SDIV2
107	SDIV3
110	ASMOVE
111	CNS
112-0	ASC
112-1	ASCL
112-2	ASCE
112-3	ASCLE
112-4	ASCA
112-5	ASCGE
112-6	ASCN
112-7	ASCG
113	CTB
114	CFB
200	VMOVE
201-0	VADD2
201-1	VADD3
201-2	VSUB2
201-3	VSUB3
201-4	VMUL2

OPCODE LIST (SORTED BY NUMBER)

201-5	VMUL3
201-6	VDIV2
201-7	VDIV3
201-10	VFAD2
201-11	VFAD3
201-12	VFSB2
201-13	VFSB3
201-14	VFMP2
201-15	VFMP3
201-16	VFDV2
201-17	VFDV3
201-20	VDFAD2
201-21	VDFAD3
201-22	VDFSB2
201-23	VDFSB3
201-24	VDFMP2
201-25	VDFMP3
201-26	VDFDV2
201-27	VDFDV3
201-30	VTFAD2
201-31	VTFAD3
201-32	VTFSB2
201-33	VTFSB3
201-34	VTFMP2
201-35	VTFMP3
201-36	VTFDV2
201-37	VTFDV3
201-40	VCAD2
201-41	VCAD3
201-42	VCSB2
201-43	VCSB3
201-44	VCMP2
201-45	VCMP3
201-46	VCDV2
201-47	VCDV3
201-50	VDCAD2
201-51	VDCAD3
201-52	VDCSB2
201-53	VDCSB3
201-54	VDCMP2
201-55	VDCMP3
201-56	VDCDV2
201-57	VDCDV3
201-60	VTCAD2
201-61	VTCAD3
201-62	VTCSB2
201-63	VTCSB3
201-64	VTCMP2
201-65	VTCMP3
201-66	VTCDV2
201-67	VTCDV3
202-0	VADD
202-1	VMUL
202-2	VFAD

OPCODE LIST (SORTED BY NUMBER)

202-3	VFMP
202-4	VDFAD
202-5	VDFMP
202-6	VTFAD
202-7	VTFMP
202-10	VCAD
202-11	VCMP
202-12	VDCAD
202-13	VDCMP
202-14	VTCAD
202-15	VTCMP
203	POLY
204	DPOLY
205	TPOLY
206	MAT
207	FMAT
210	DFMAT
211	TFMAT
212	CMAT
213	DCMAT
214	TCMAT

OPCODE List

The following is a list of opcode assignments (sorted by name).

Name	New OPCODE	Page Num	PDP10 OPCODE (if different from new machine)
ADD	270B10		
ADDB	273B10		
ADDI	271B10		
ADDM	272B10		
ADJBP	133B10	56	
ADJSP	105B10	44	
AND	404B10		
ANDB	407B10		
ANDCA	410B10		
ANDCAB	413B10		
ANDCAI	411B10		
ANDCAM	412B10		
ANDCB	440B10		
ANDCBB	443B10		
ANDCBI	441B10		
ANDCBM	442B10		
ANDCM	630B10	14	420B10
ANDCMB	423B10		
ANDCMI	620B10	14	421B10
ANDCMM	422B10		
ANDI	405B10		
ANDM	406B10		
AOBJGE	252B10	16	-
AOBJL	253B10	16	-
AOBJN	253B10		
AOBJP	252B10		
AOJ	340B10		
AOJA	344B10		
AOJE	342B10		
AOJG	347B10		
AOJGE	345B10		
AOJL	341B10		
AOJLE	343B10		
AOJN	346B10		
AOS	350B10		
AOSA	354B10		
AOSE	352B10		
AOSG	357B10		
AOSGE	355B10		
AOSL	351B10		
AOSLE	353B10		
AOSN	356B10		
ASC	112B10	109	-
ASCA	ASC 4,	109	-
ASCE	ASC 2,	109	-
ASCG	ASC 7,	109	-

OPCODE LIST (SORTED BY NAME)

ASCGE	ASC 5,	109	-
ASCL	ASC 1,	109	-
ASCLE	ASC 3,	109	-
ASCN	ASC 6,	109	-
ASH	240B10		
ASHC	244B10		
ASMOVE	110B10	107	-
BAOS	255B10	55	-
BAOSA	BAOS 4,	55	-
BAOSE	BAOS 2,	55	-
BAOSG	BAOS 7,	55	-
BAOSGE	BAOS 5,	55	-
BAOSL	BAOS 1,	55	-
BAOSLE	BAOS 3,	55	-
BAOSN	BAOS 6,	55	-
BBLT	XCT 4,	33	-
BLC	720B10	23	-
BLCA	724B10	23	-
BLCE	722B10	23	-
BLCN	726B10	23	-
BLN	300B10	23	-
BLNA	304B10	23	-
BLNE	702B10	23	-
BLNN	706B10	23	-
BLO	730B10	23	-
BLOA	734B10	23	-
BLOE	732B10	23	-
BLON	736B10	23	-
BLT	XCT 3,	42	251B10
BLZ	710B10	23	-
BLZA	714B10	23	-
BLZE	712B10	23	-
BLZN	716B10	23	-
BRC	721B10	23	-
BRCA	725B10	23	-
BRCE	723B10	23	-
BRCN	727B10	23	-
BRN	300B10	23	-
BRNA	304B10	23	-
BRNE	703B10	23	-
BRNN	707B10	23	-
BRO	731B10	23	-
BROA	735B10	23	-
BROE	733B10	23	-
BRON	737B10	23	-
BRZ	711B10	23	-
BRZA	715B10	23	-
BRZE	713B10	23	-
BRZN	717B10	23	-
BSMOVE	002B10	73	-
BSOS	BAOS 10,	55	-
BSOSA	BAOS 14,	55	-
BSOSE	BAOS 12,	55	-
BSOSG	BAOS 17,	55	-

OPCODE LIST (SORTED BY NAME)

BSOSGE	BAOS 15,	55	-
BSOSL	BAOS 11,	55	-
BSOSLE	BAOS 13,	55	-
BSOSN	BAOS 16,	55	-
BUS	140B10	28	-
BUSB	143B10	28	-
BUSI	141B10	28	-
BUSM	142B10	28	-
CAD	746B10	63	-
CAI	300B10		
CAIA	304B10		
CAIE	302B10		
CAIG	307B10		
CAIGE	305B10		
CAIL	301B10		
CAILE	303B10		
CAIN	306B10		
CAM	300B10	14	310B10
CAMA	304B10	14	314B10
CAME	312B10		
CAMG	317B10		
CAMGE	315B10		
CAML	311B10		
CAMLE	313B10		
CAMN	316B10		
CDFDI	EOP 35	71	-
CDFF	EOP 36	71	-
CDFI	EOP 34	71	-
CDFTF	EOP 37	71	-
CDIDF	EOP 26	71	-
CDIF	EOP 25	71	-
CDII	EOP 24	71	-
CDITF	EOP 27	71	-
CDV	751B10	63	-
CFB	114B10	110	-
CFDF	EOP 32	71	-
CFDI	EOP 31	71	-
CFI	EOP 30	71	-
CFTF	EOP 33	71	-
CIDF	EOP 22	71	-
CIDI	EOP 20	71	-
CIF	EOP 21	71	-
CITF	EOP 23	71	-
CMAT	212B10	96	-
CMP	750B10	63	-
CNEG	EOP 12	69	-
CNS	111B10	108	-
CONCAT	003B10	74	-
CSB	747B10	63	-
CTB	113B10	110	-
CTFDF	EOP 43	71	-
CTFDI	EOP 41	71	-
CTFF	EOP 42	71	-
CTFI	EOP 40	71	-

OPCODE LIST (SORTED BY NAME)

DADD	114B10	44	
DCAD	752B10	63	-
DCDV	755B10	63	-
DCMAT	213B10	96	-
DCMP	754B10	63	-
DCNEG	EOP 13	69	-
DCSB	753B10	63	-
DDIV	117B10	44	
DECBP	XCT 6,	57	-
DFAD	110B10	62	
DFDV	113B10	62	
DFMAT	210B10	96	-
DFMP	112B10	62	
DFSB	111B10	62	
DIV	234B10	44	
DIVB	237B10	44	
DIVI	235B10	44	
DIVM	236B10	44	
DMOVE	120B10		
DMOVEM	124B10		
DMOVN	121B10		
DMOVNM	125B10		
DMUL	116B10	44	
DNEG	EOP 10	69	-
DPB	137B10	52	
DPBA	650B10	52	-
DPBI	251B10	54	-
DPBL	740B10	52	-
DPBW	670B10	52	-
DPOLY	204B10	95	-
DPOP	426B10	114	-
DPUSH	424B10	114	-
DSETZ	EOP 1	68	-
DSETZM	XCT 13,	68	-
DSKP	300B10	67	-
DSKPA	304B10	67	-
DSKPE	JRST 2,	67	-
DSKPG	JRST 7,	67	-
DSKPGE	335B10	67	-
DSKPL	331B10	67	-
DSKPLE	JRST 3,	67	-
DSKPN	JRST 6,	67	-
DSUB	115B10	44	
EA	123B10	27	-
EOP	400B10	34	-
EQV	444B10		
EQVB	447B10		
EQVI	445B10		
EQVM	446B10		
EXCH	250B10		
EXTEND	XCT 7,	72	123B10
FAD	144B10	62	140B10
FADB	147B10	62	143B10
FADI	145B10	62	-

OPCODE LIST (SORTED BY NAME)

FADM	146B10	62	142B10
FADR	144B10	62	
FADRB	147B10	62	
FADRI	145B10	62	
FADRM	146B10	62	
FDV	174B10	62	170B10
FDVB	177B10	62	173B10
FDVI	175B10	62	-
FDVM	176B10	62	172B10
FDVR	174B10	62	
FDVRB	177B10	62	
FDVRI	175B10	62	
FDVRM	176B10	62	
FIX	122B10		
FIXR	126B10		
FLT	127B10	16	-
FLTR	127B10		
FMAT	207B10	96	-
FMOVEI	401B10	64	-
FMP	164B10	62	160B10
FMPB	167B10	62	163B10
FMPI	165B10	62	-
FMPM	166B10	62	162B10
FMPR	164B10	62	
FMPRB	167B10	62	
FMPRI	165B10	62	
FMPRM	166B10	62	
FPUSHI	320B10	115	-
FSB	154B10	62	150B10
FSBB	157B10	62	153B10
FSBI	155B10	62	-
FSBM	156B10	62	152B10
FSBR	154B10	62	
FSBRB	157B10	62	
FSBRI	155B10	62	
FSBRM	156B10	62	
FSC	132B10		
HALT	JRST 4,		
HLL	500B10		
HLLE	530B10		
HLLEI	531B10		
HLLEM	532B10		
HLLES	533B10		
HLLI	501B10		
HLLM	502B10		
HLLO	520B10		
HLLOI	521B10		
HLLOM	522B10		
HLLOS	523B10		
HLLS	203B10	14	503B10
HLLZ	510B10		
HLLZI	511B10		
HLLZM	512B10		
HLLZS	513B10		

HLR	544B10		
HLRE	574B10		
HLREI	575B10		
HLREM	576B10		
HLRES	577B10		
HLRI	545B10		
HLRM	546B10		
HLRO	564B10		
HLROI	565B10		
HLROM	566B10		
HLROS	567B10		
HLRS	547B10		
HLRZ	554B10		
HLRZI	555B10		
HLRZM	556B10		
HLRZS	557B10		
HMOVE	744B10	65	-
HMOVEM	745B10	65	-
HPOP	741B10	114	-
HPUSH	416B10	114	-
HRL	504B10		
HRLE	534B10		
HRLEI	535B10		
HRLEM	536B10		
HRLES	537B10		
HRLI	505B10		
HRLM	506B10		
HRLO	524B10		
HRLOI	525B10		
HRLOM	526B10		
HRLOS	527B10		
HRLS	507B10		
HRLZ	514B10		
HRLZI	515B10		
HRLZM	516B10		
HRLZS	517B10		
HRR	540B10		
HRRE	570B10		
HRREI	571B10		
HRREM	572B10		
HRRES	573B10		
HRRI	541B10		
HRRM	542B10		
HRRO	560B10		
HRROI	561B10		
HRROM	562B10		
HRROS	563B10		
HRRS	203B10	14	543B10
HRRZ	550B10		
HRRZI	551B10		
HRRZM	552B10		
HRRZS	553B10		
HSETZ	EOP 4	68	-
HSETZM	XCT 16,	68	-

OPCODE LIST (SORTED BY NAME)

HSKP	300B10	67	-
HSKPA	304B10	67	-
HSKPE	JRST 14,	67	-
HSKPGE	335B10	67	-
HSKPL	331B10	67	-
HSKPN	JRST 15,	67	-
IBP	133B10		
IBPA	XCT 10,	52	-
IBPL	XCT 12,	52	-
IBPW	XCT 11,	52	-
IDIV	230B10		
IDIVB	233B10		
IDIVI	231B10		
IDIVM	232B10		
IDPB	136B10	52	
IDPBA	330B10	52	-
IDPBL	435B10	52	-
IDPBW	417B10	52	-
IDV	100B10	30	-
IDVI	101B10	30	-
IDVM	232B10	30	-
ILDB	134B10	52	
ILDBA	257B10	52	-
ILDBL	324B10	52	-
ILDBW	264B10	52	-
IMUL	220B10		
IMULB	223B10		
IMULI	221B10		
IMULM	222B10		
IVID	150B10	29	-
IVIDB	153B10	29	-
IVIDI	151B10	29	-
IVIDM	152B10	29	-
JFCL	300B10	48	255B10
JFFO	243B10		
JRA	267B10	45	
JRST	254B10		
JSA	266B10	45	
JSP	265B10	45	
JSR	JRST 5,	45	264B10
JUMP	300B10	14	320B10
JUMPA	254B10	47	-
JUMPE	322B10		
JUMPG	327B10		
JUMPGE	325B10		
JUMPL	321B10		
JUMPLE	323B10		
JUMPN	326B10		
KADD	EOP 10406	112	-
KAND	EOP 10421	124	-
KANDC0	EOP 10424	124	-
KANDC1	EOP 10422	124	-
KBUS	EOP 10412	112	-
KCAD	EOP 21006	113	-

OPCODE LIST (SORTED BY NAME)

KCBS	EOP 21012	113	-
KCDV	EOP 21011	113	-
KCMP	EOP 21010	113	-
KCNEG	EOP 11001	125	-
KCSB	EOP 21007	113	-
KCVD	EOP 21013	113	-
KDCAD	EOP 37000	113	-
KDCBS	EOP 37004	113	-
KDCDV	EOP 37003	113	-
KDCMP	EOP 37002	113	-
KDCNEG	EOP 22000	125	-
KDCSB	EOP 37001	113	-
KDCVD	EOP 37005	113	-
KDFAD	EOP 21000	113	-
KDFBS	EOP 21004	113	-
KDFDV	EOP 21003	113	-
KDFF	EOP 10417	126	-
KDFI	EOP 10416	126	-
KDFMP	EOP 21002	113	-
KDFSB	EOP 21001	113	-
KDFTF	EOP 11400	126	-
KDFVD	EOP 21005	113	-
KDIV	EOP 10411	112	-
KDNEG	EOP 11000	125	-
KDOM	EOP 10415	112	-
KDSKP	EOP 10000	122	-
KDSKPA	EOP 10004	122	-
KDSKPE	EOP 10002	122	-
KDSKPG	EOP 10007	122	-
KDSKPGE	EOP 10005	122	-
KDSKPL	EOP 10001	122	-
KDSKPLE	EOP 10003	122	-
KDSKPN	EOP 10006	122	-
KEQV	EOP 10431	124	-
KFAD	EOP 10400	113	-
KFBS	EOP 10404	113	-
KFDF	EOP 5001	126	-
KFDV	EOP 10403	113	-
KFI	EOP 4402	126	-
KFMP	EOP 10402	113	-
KFSB	EOP 10401	113	-
KFTF	EOP 5401	126	-
KFVD	EOP 10405	113	-
KHSKP	EOP 30000	122	-
KHSKPA	EOP 30002	122	-
KHSKPE	EOP 30001	122	-
KHSKPN	EOP 30003	122	-
KIDF	EOP 5000	126	-
KIF	EOP 4401	126	-
KILL	EOP 35000	128	-
KITF	EOP 5400	126	-
KMOD	EOP 10414	112	-
KMUL	EOP 10410	112	-
KNAND	EOP 10436	124	-

OPCODE LIST (SORTED BY NAME)

KNEG	EOP 4400	125	-
KNOR	EOP 10430	124	-
KOR	EOP 10427	124	-
KORC0	EOP 10435	124	-
KORC1	EOP 10433	124	-
KQSKP	EOP 20000	122	-
KQSKPA	EOP 20002	122	-
KQSKPE	EOP 20001	122	-
KQSKPN	EOP 20003	122	-
KSET0	EOP 10423	124	-
KSET1	EOP 10425	124	-
KSETC0	EOP 10434	124	-
KSETC1	EOP 10432	124	-
KSETO	EOP 10437	124	-
KSETZ	EOP 10420	124	-
KSKP	EOP 4000	122	-
KSKPA	EOP 4004	122	-
KSKPE	EOP 4002	122	-
KSKPG	EOP 4007	122	-
KSKPGE	EOP 4005	122	-
KSKPL	EOP 4001	122	-
KSKPLE	EOP 4003	122	-
KSKPN	EOP 4006	122	-
KSUB	EOP 10407	112	-
KTCAD	EOP 37400	113	-
KTCBS	EOP 37404	113	-
KTCDV	EOP 37403	113	-
KTCMP	EOP 37402	113	-
KTCNEG	EOP 33000	125	-
KTCSB	EOP 37401	113	-
KTCVD	EOP 37405	113	-
KTFAD	EOP 31400	113	-
KTFBS	EOP 31404	113	-
KTFDF	EOP 15000	126	-
KTFDV	EOP 31403	113	-
KTFF	EOP 14401	126	-
KTFI	EOP 14400	126	-
KTFMP	EOP 31402	113	-
KTFSB	EOP 31401	113	-
KTFVD	EOP 31405	113	-
KTNEG	EOP 15400	125	-
KTSKP	EOP 14000	122	-
KTSKPA	EOP 14004	122	-
KTSKPE	EOP 14002	122	-
KTSKPG	EOP 14007	122	-
KTSKPGE	EOP 14005	122	-
KTSKPL	EOP 14001	122	-
KTSKPLE	EOP 14003	122	-
KTSKPN	EOP 14006	122	-
KVID	EOP 10413	112	-
KXOR	EOP 10426	124	-
LDB	135B10	52	
LDBA	543B10	52	-
LDBL	615B10	52	-

OPCODE LIST (SORTED BY NAME)

LDBW	611B10	52	-
LDBX	104B10	54	-
LSH	242B10		
LSHC	246B10		
MAP	033B10		
MAT	206B10	96	-
MOVE	200B10	14	
MOVEI	201B10	14	
MOVEIA	310B10	20	-
MOVEM	202B10	14	
MOVES	203B10	14	
MOVM	214B10		
MOVMI	215B10		
MOVMM	216B10		
MOVMS	217B10		
MOVN	210B10		
MOVNI	211B10		
MOVNM	212B10		
MOVNS	213B10		
MOVS	204B10		
MOVSI	205B10		
MOVSM	206B10		
MOVSS	207B10		
MUL	224B10	44	
MULB	227B10	44	
MULI	225B10	44	
MULM	226B10	44	
NAND	470B10	16	-
NANDB	473B10	16	-
NANDI	471B10	16	-
NANDM	472B10	16	-
NEG	EOP 7	69	-
NOR	440B10	16	-
NORB	443B10	16	-
NORI	441B10	16	-
NORM	442B10	16	-
OR	434B10		
ORB	437B10		
ORCA	454B10		
ORCAB	457B10		
ORCAI	455B10		
ORCAM	456B10		
ORCB	470B10		
ORCBB	473B10		
ORCBI	471B10		
ORCBM	472B10		
ORCM	464B10		
ORCMB	467B10		
ORCMI	465B10		
ORCMM	466B10		
ORI	660B10	14	435B10
ORM	436B10		
PADD	160B10	116	-
PADDI	161B10	116	-

OPCODE LIST (SORTED BY NAME)

PADDM	462B10	118	-
PDIV	172B10	116	-
PDIVI	173B10	116	-
PDPB	421B10	119	-
PFAD	450B10	117	-
PFADI	451B10	117	-
PFADM	503B10	118	-
PFDV	704B10	117	-
PFDVI	705B10	117	-
PFMP	700B10	117	-
PFMPI	701B10	117	-
PFSB	474B10	117	-
PFSBI	475B10	117	-
PHYDPB	035B10	59	-
PHYLDB	034B10	59	-
PLDB	420B10	119	-
PMOVEM	247B10	120	-
PMUL	170B10	116	-
PMULI	171B10	116	-
POLY	203B10	95	-
POP	262B10		
POPJ	263B10	43	
PORTAL	JRST 1,	78	
PSAV	EOP 40000	37	-
PSAVE	EOP 600000	40	-
PSUB	162B10	116	-
PSUBI	163B10	116	-
PUPJ	EOP 4403	121	-
PUSH	261B10		
PUSHI	314B10	20	-
PUSHJ	260B10	45	
PUSHZ	431B10	118	-
QMOVE	742B10	65	-
QMOVEM	743B10	65	-
QPOP	415B10	114	-
QPUSH	414B10	114	-
QSETZ	EOP 3	68	-
QSETZM	XCT 15,	68	-
QSKP	300B10	67	-
QSKPA	304B10	67	-
QSKPE	JRST 10,	67	-
QSKPGE	335B10	67	-
QSKPL	331B10	67	-
QSKPN	JRST 11,	67	-
REST	EOP 400000	39	-
ROT	241B10		
ROTC	245B10		
RST	EOP 36000	36	-
SADD2	100B10	102	-
SADD3	101B10	102	-
SAV	EOP 36400	35	-
SAVE	EOP 200000	38	-
SCOMP	010B10	77	-
SCOMPA	014B10	77	-

OPCODE LIST (SORTED BY NAME)

SCOMPE	012B10	77	-
SCOMPG	017B10	77	-
SCOMPGE	015B10	77	-
SCOMPL	011B10	77	-
SCOMPLE	013B10	77	-
SCOMPN	016B10	77	-
SDIV2	106B10	102	-
SDIV3	107B10	102	-
SETA	300B10	14	424B10
SETAB	202B10	14	427B10
SETAI	300B10	14	425B10
SETAM	202B10	14	426B10
SETCA	32001017777	47	-
SETCAB	453B10		
SETCAM	452B10		
SETCM	460B10		
SETCMB	463B10		
SETCMI	461B10		
SETCMM	XCT 5,	47	462B10
SETM	200B10	14	414B10
SETMB	200B10	14	417B10
SETMI	201B10	14	415B10
SETMM	300B10	14	416B10
SETO	20041017777	46	-
SETOB	477B10		
SETOM	476B10	41	
SETZ	201B10	14	400B10
SETZB	403B10		
SETZI	201B10	14	401B10
SETZM	402B10	41	
SKIP	203B10	14	330B10
SKIPA	334B10		
SKIPE	332B10		
SKIPG	337B10		
SKIPGE	335B10		
SKIPL	331B10		
SKIPLE	333B10		
SKIPN	336B10		
SMOVE	001B10	73	-
SMUL2	104B10	102	-
SMUL3	105B10	102	-
SOJ	360B10		
SOJA	364B10		
SOJE	362B10		
SOJG	367B10		
SOJGE	365B10		
SOJL	361B10		
SOJLE	363B10		
SOJN	366B10		
SOS	370B10		
SOSA	374B10		
SOSE	372B10		
SOSG	377B10		
SOSGE	375B10		

SOSL	371B10		
SOSLE	373B10		
SOSN	376B10		
SPATE	006B10	76	-
SPATN	007B10	76	-
SRCHE	004B10	75	-
SRCHN	005B10	75	-
SSTEP	XCT 1,	25	-
SSUB2	102B10	102	-
SSUB3	103B10	102	-
SUB	274B10		
SUBB	277B10		
SUBI	275B10		
SUBM	276B10		
SWAP	EOP 35400	127	-
TCAD	756B10	63	-
TCDV	761B10	63	-
TCMAT	214B10	96	-
TCMP	760B10	63	-
TCNEG	EOP 14	69	-
TCSB	757B10	63	-
TDC	430B10	14	650B10
TDCA	654B10	48	
TDCE	652B10	48	
TDCN	656B10	48	
TDN	300B10	14	610B10
TDNA	304B10	14	614B10
TDNE	612B10	48	
TDNN	616B10	48	
TDNU	614B10	21	-
TDO	434B10	14	670B10
TDOA	674B10	48	
TDOE	672B10	48	
TDON	676B10	48	
TDU	610B10	21	-
TDZ	630B10		
TDZA	634B10	48	
TDZE	632B10	48	
TDZN	636B10	48	
TFAD	102B10	62	-
TFDV	107B10	62	-
TFMAT	211B10	96	-
TFMP	106B10	62	-
TFSB	103B10	62	-
TLC	641B10	48	
TLCA	645B10	48	
TLCE	643B10	48	
TLCN	647B10	48	
TLN	300B10	14	601B10
TLNA	304B10	14	605B10
TLNE	603B10	48	
TLNN	607B10	48	
TLNU	605B10	21	-
TLO	661B10	48	

OPCODE LIST (SORTED BY NAME)

TLOA	665B10	48	
TLOE	663B10	48	
TLON	667B10	48	
TLU	601B10	21	-
TLZ	621B10	48	
TLZA	625B10	48	
TLZE	623B10	48	
TLZN	627B10	48	
TMOVE	130B10	65	-
TMOVEM	131B10	65	-
TNEG	EOP 11	69	-
TPOLY	205B10	95	-
TPOP	427B10	114	-
TPUSH	425B10	114	-
TRC	640B10		
TRCA	644B10	48	
TRCE	642B10	48	
TRCN	646B10	48	
TRN	300B10	14	600B10
TRNA	304B10	14	604B10
TRNE	602B10	48	
TRNN	606B10	48	
TRNU	604B10	21	-
TRO	660B10		
TROA	664B10	48	
TROE	662B10	48	
TRON	666B10	48	
TRU	600B10	21	-
TRZ	620B10		
TRZA	624B10	48	
TRZE	622B10	48	
TRZN	626B10	48	
TSC	651B10	48	
TSCA	655B10	48	
TSCE	653B10	48	
TSCN	657B10	48	
TSETZ	EOP 2	68	-
TSETZM	XCT 14,	68	-
TSKP	300B10	67	-
TSKPA	304B10	67	-
TSKPE	JRST 12,	67	-
TSKPG	JRST 17,	67	-
TSKPGE	335B10	67	-
TSKPL	331B10	67	-
TSKPLE	JRST 13,	67	-
TSKPN	JRST 16,	67	-
TSN	300B10	14	611B10
TSNA	304B10	14	615B10
TSNE	613B10	48	
TSNN	617B10	48	
TSO	671B10	48	
TSOA	675B10	48	
TSOE	673B10	48	
TSON	677B10	48	

TSZ	631B10	48	
TSZA	635B10	48	
TSZE	633B10	48	
TSZN	637B10	48	
UDPB	037B10	58	-
ULDB	036B10	58	-
UMAP	032B10	31	-
VADD	202B10+0	94	-
VADD2	201B10+0	91	-
VADD3	201B10+1	91	-
VCAD	202B10+10	94	-
VCAD2	201B10+40	91	-
VCAD3	201B10+41	91	-
VCDV2	201B10+46	91	-
VCDV3	201B10+47	91	-
VCMP	202B10+11	94	-
VCMP2	201B10+44	91	-
VCMP3	201B10+45	91	-
VCSB2	201B10+42	91	-
VCSB3	201B10+43	91	-
VDCAD	202B10+12	94	-
VDCAD2	201B10+50	91	-
VDCAD3	201B10+51	91	-
VDCDV2	201B10+56	91	-
VDCDV3	201B10+57	91	-
VDCMP	202B10+13	94	-
VDCMP2	201B10+54	91	-
VDCMP3	201B10+55	91	-
VDCSB2	201B10+52	91	-
VDCSB3	201B10+53	91	-
VDFAD	202B10+4	94	-
VDFAD2	201B10+20	91	-
VDFAD3	201B10+21	91	-
VDFDV2	201B10+26	91	-
VDFDV3	201B10+27	91	-
VDFMP	202B10+5	94	-
VDFMP2	201B10+24	91	-
VDFMP3	201B10+25	91	-
VDFSB2	201B10+22	91	-
VDFSB3	201B10+23	91	-
VDIV2	201B10+6	91	-
VDIV3	201B10+7	91	-
VFAD	202B10+2	94	-
VFAD2	201B10+10	91	-
VFAD3	201B10+11	91	-
VFDV2	201B10+16	91	-
VFDV3	201B10+17	91	-
VFMP	202B10+3	94	-
VFMP2	201B10+14	91	-
VFMP3	201B10+15	91	-
VFSB2	201B10+12	91	-
VFSB3	201B10+13	91	-
VMOVE	200B10	87	-
VMUL	202B10+1	94	-

VMUL2	201B10+4	91	-
VMUL3	201B10+5	91	-
VSUB2	201B10+2	91	-
VSUB3	201B10+3	91	-
VTCAD	202B10+14	94	-
VTCAD2	201B10+60	91	-
VTCAD3	201B10+61	91	-
VTCDV2	201B10+66	91	-
VTCDV3	201B10+67	91	-
VTCMP	202B10+15	94	-
VTCMP2	201B10+64	91	-
VTCMP3	201B10+65	91	-
VTCSB2	201B10+62	91	-
VTCSB3	201B10+63	91	-
VTFAD	202B10+6	94	-
VTFAD2	201B10+30	91	-
VTFAD3	201B10+31	91	-
VTFDV2	201B10+36	91	-
VTFDV3	201B10+37	91	-
VTFMP	202B10+7	94	-
VTFMP2	201B10+34	91	-
VTFMP3	201B10+35	91	-
VTFSB2	201B10+32	91	-
VTFSB3	201B10+33	91	-
XCT	256B10	47	
XDIS	XOP 3,	32	-
XJSR	XOP 0,	32	-
XOP	031B10	18	-
XOR	430B10		
XORB	433B10		
XORI	640B10	14	431B10
XORM	432B10		
XPCW	XOP 2,	32	-
XRET	XOP 1,	32	-

INDEX