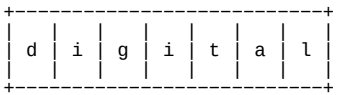


To: Architecture distribution

Date: 24 Jun 83
From: Mike Uhler
Dept: L.S.E.G.
DTN: (8−)231−6448
Loc/Mail stop: MRO1−2/E85
Net mail: UHLER at IO

Subject: Extended addressing
Revision: 5

Revision history

Revision	Date	Changes
5	07−Jul−83	Make one−word global byte pointers legal in section 0. Make JSA and JRA illegal for inter−section use.

1.0 Introduction

As we attempted to implement the EBOX microcode for Jupiter, we quickly discovered that there were some serious deficiencies in the documentation for what we were trying to implement. The most serious deficiency was in the area of rules for extended addressing, especially for the exception conditions. We found ourselves implementing the EBOX microcode partially from the Processor Reference Manual (PRM) but, more often than not, from a collection of old documentation, memos, and recollections of what was decided in the design of extended addressing.

After spending two weeks attempting to decipher what the rules should be and comparing them with the KL10 implementation, I wound up with a large collection of notes concerning aspects of extended addressing which are either not documented or poorly documented. Several meetings of the PDP−10 Architecture Committee ensued, and this memo is an attempt to formalize my notes.

The intent of the memo is to provide a description of extended addressing as defined by the PDP−10 architecture. This material really belongs in the Processor Reference Manual, and every attempt will be made to get it included in the next release of the manual. Note that certain implementations of the PDP−10 architecture don't always conform to the descriptions given in the memo. These are descriptions of what SHOULD be implemented, not necessarily what IS implemented. However, all future PDP−10 processors should conform to these descriptions.

In order to make it easier for the reader, I've also added a lot of background, definitions, and descriptions of extended addressing that are found in other references. This additional discussion should make the overall structure of extended addressing more clear.

In order to avoid swamping the reader with too much detail at any point, I sometimes intentionally ignore or understate certain important aspects of the examples that I use. These items are generally covered later in the memo. I also occasionally forward reference topics. Because of this organization, it may be best to make a quick first pass through the memo to pick out the highlights and then go back and make a more detailed pass.

This memo assumes that the reader has at least a basic knowledge of the PDP−10 instruction set, the notation used to describe instructions, and the format of an instruction word. Readers who do not have this knowledge are referred to sections 1.4 through 1.6 of the Processor Reference Manual and to the Macro Assembler Reference Manual.

2.0 Reference materials

The primary source of information about the instruction set is the Processor Reference Manual. Unfortunately, there are some inaccuracies and some omissions in the sections related to extended addressing. The "Extended Effective Address Calculation" flow chart on page 1-30 of the PRM is the best "description" of the effective address calculation algorithms and it is attached to this memo for the convenience of the reader.

The KL10 Engineering Functional Spec contains several chapters related to this topic and has some interesting insights. Especially interesting are chapters 2.2, "User Interface to Extended Addressing", and 2.3, "Monitor Calling (MUUO, PXCT)". Along with these chapters is a hand-drawn flow chart by Tom Hastings entitled "Flow for Extended Addressing" that clears up several questions about EA-calc algorithms, especially in the area of PXCT. A copy of this flow chart is attached.

Old memos describing the design of extended addressing and the implementation of extended addressing in TOPS-20 are also somewhat helpful.

Finally, the KL10 microcode contains a few helpful comments about exception conditions in that implementation of extended addressing. It is in no sense "light reading", however.

3.0 Historical summary of extended addressing

PDP-10 processors prior to the model B KL10 implemented a virtual address space of 256K words. As programs and the operating systems grew, it became apparent that a virtual address space that was limited to 256K was insufficient for future expansion. Sometime in late 1973, an Extended Addressing Design Group was formed to evaluate proposals for increasing the virtual address space of the PDP-10. By early 1975, this group had agreed upon one proposal, and this proposal was documented in chapter 2.2 of the KL10 Engineering Functional Spec.

This proposal increased the size of the virtual address space from 256K words to 1 billion words by expanding the size of a virtual address from 18 bits to 30 bits. The virtual address space is logically divided into 4096 sections of 256K words each. The program may use these sections as separate logical entities or treat them as one large contiguous address space. Instructions, however, must explicitly transfer control between sections; they may not "fall" into the next section.

The increase in the size of the virtual address space was accompanied by an increase in the size of PC, from 18 to 30 bits. This increase allowed a program to execute in any of the extended sections. The contents of bits 6-17 of PC were termed the "PC section".

In order to allow an instruction to specify a full 30-bit virtual address, the rules for indexing and indirection were modified when PC section was non-zero. In addition, new instructions were defined to allow a program to jump to other sections.

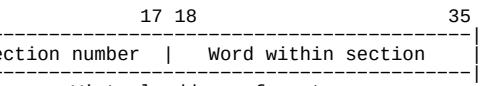
To insure compatibility with programs written for non-extended processors, section zero is treated exactly as it is on non-extended processors. This means that if a program is executing in section zero, nearly all instructions behave exactly as they would if the program were executed on a non-extended machine. Programs running in section zero cannot reference data in any other section (with one exception) and entry into another section is possible only with a few instructions (e.g., XJRSTF, XJRST, etc.).

The first processor to implement extended addressing was the model B KL10. Due to hardware restrictions, this processor implemented only 32 of the 4096 sections of virtual address space. References to virtual sections above the implemented range cause a page fail trap to the monitor. The KC10 implements the full 30-bit virtual address space.

4.0 Definition of terms

Before we start looking at extended addressing, let's define some terms:

- A virtual address is a 30-bit address used to reference a word in an address space. Although the address space can be considered to be one large, contiguous space, it is probably easier to consider it to be broken into sections of 256K words each. Bits 6-17 of the virtual address then specify the section number and bits 18-35 specify the word within the section. A virtual address looks like:



Virtual address format

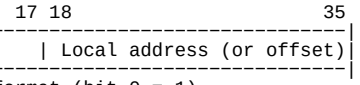
PC has the same format as a virtual address.

- An address word is a word containing I, X, and Y fields (see the PRM for definitions for these fields) in either IFIW or EFIW (see below) format. An effective address calculation takes such a word as input. Thus, instructions, indirect words, and byte pointers are all examples of address words.

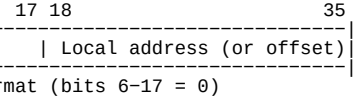
- A local address is an 18-bit in-section address that, when combined with a default section number, specifies a full 30-bit address. The section number is supplied by something other than the address word or index register.

- A global address is a 30-bit address that supplies its own section number. Therefore, no default section need be applied.

- A local index is an 18-bit displacement or address obtained from an index register used in an effective address calculation in section zero, or from an index register used in a non-zero section that has bit 0=1 or bits 6-17 equal zero. In a non-zero section, an index register containing a local index has one of the following formats:

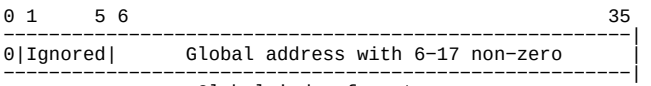


Local index format (bit 0 = 1)



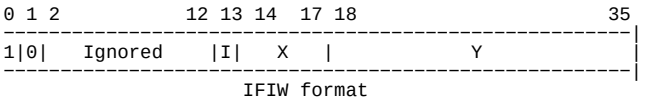
Local index format (bits 6-17 = 0)

- A global index is a 30-bit displacement or address obtained from an index register used in an effective address calculation in a non-zero section, that has bit 0=0 and bits 6-17 non-zero. An index register containing a global index looks like:



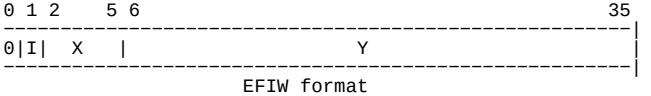
Global index format

- An instruction format indirect word (IFIW) is any indirect word in section zero, or an indirect word in a non-zero section that has bit 0=1 and bit 1=0 (instructions being executed are always interpreted in IFIW format). In this format, bit 13 is the indirect bit, bits 14-17 are the index register address, and bits 18-35 are the local memory address. An IFIW in a non-zero section looks like:



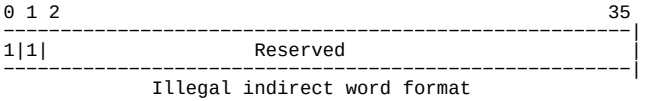
IFIW format

- An extended format indirect word (EFIW) is any indirect word in a non-zero section that has bit 0=0. In this format, bit 1 is the indirect bit, bits 2-5 are the index register address, and bits 6-35 are the global memory address. An EFIW looks like:



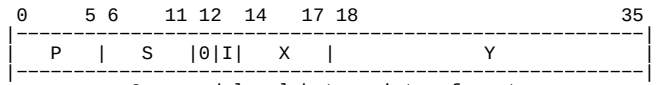
EFIW format

- An illegal indirect word is any indirect word in a non-zero section that has both bits 0 and 1 set to a 1. This type of indirect word is reserved for use by future hardware. If an EA-calc encounters this type of indirect word in a non-zero section, it will generate a page fail. The monitor cannot perform any user service as a result of this trap, including trapping to the user, since this would cause possible compatibility problems with future machines. An illegal indirect word looks like:



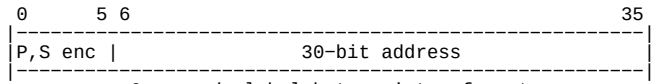
Illegal indirect word format

- A one-word local byte pointer is any byte pointer whose P field is less than or equal to 36 and that has bit 12=0. In this type of byte pointer, bits 13-35 have the same format as an IFIW, and bits 0-11 specify the size and position of the byte. A one-word local byte pointer looks like:



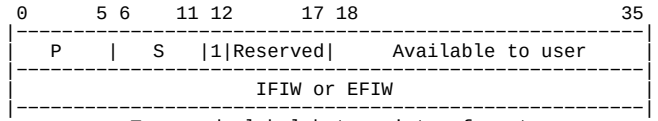
One-word local byte pointer format

- A one-word global byte pointer is any byte pointer whose P field is greater than 36. In this type of byte pointer, bits 0-5 are an encoded representation of the size and position of the byte and bits 6-35 supply a full 30-bit address of the word containing the byte. A one-word global byte pointer looks like:



One-word global byte pointer format

- A two-word global byte pointer is any byte pointer in a non-zero section whose P field is less than or equal to 36 and which has bit 12=1. As its name implies, this type of byte pointer consists of two words where bits 0-11 of the first word give the size and position of the byte and bit 12 must be a 1. The second word is either an IFIW or an EFIW and, when EA-calc'ed, supplies the address of the word containing the byte. A two-word global byte pointer looks like:



Two-word global byte pointer format

- A local stack pointer is any stack pointer in section zero, or a stack pointer in a non-zero section that has bit 0=1 or bits 6-17 equal zero before incrementing or decrementing (exactly like a local index). Incrementing or decrementing such a stack pointer will operate on both halves of the pointer independently, suppressing carries out of bit 18.

- A global stack pointer is a stack pointer in a non-zero section that has bit 0=0 and bits 6-17 non-zero before incrementing (exactly like a global index). Incrementing or decrementing such a stack pointer will treat the entire word as a 30-bit quantity.

5.0 Effective Address Calculations

No discussion of extended addressing is complete without talking about EA-calc's. An effective address calculation is performed on every instruction before it is executed. In addition, some instructions perform additional EA-calc's during the processing of the instruction (e.g. byte instruction EA-calc of the byte pointer).

5.1 Description of the EA-calc algorithm

The basic operation of an EA-calc is to process a so-called address word by adding the Y field of the word to the contents of the optional index register to compute a modified address. If the indirect bit is set in the address word, another word is fetched from the memory location addressed by the computed address and the entire process repeats until a word is found with the indirect bit not set. Sound simple? Well, let's look at the operation in a bit more detail.

The address word can be of two different formats, IFIW or EFIW (an instruction is treated as an IFIW when it is EA-calc'ed). In addition, an index can be of two different formats, local or global. Note that in section zero, all address words are IFIWs and all indices are local by definition. The complexity involved in the EA-calc algorithm is the result of these multiple formats.

Since the indirect bit simply causes another address word to be fetched and the EA-calc process to be repeated, we can fully characterize an EA-calc by looking at the combinations of IFIW, EFIW, and indices in local and global format. Let's look at these combinations one at a time.

5.1.1 No indexing

If no index register is specified in the address word, the EA-calc is strictly a function of the Y field in the address word. For an IFIW, the result is a local address. For example, both

```
1,,100/ MOVE 1,200
```

and

```
1,,100/ MOVE 1,@150
1,,150/ 400000,,200
```

compute a local effective address of 200. In the first case, the only address word is the instruction itself, which is treated as an implicit IFIW. In the second case, there are two address words, the instruction and the indirect word, and the indirect word is in the IFIW format.

For an EFIW, the result is a full 30-bit global address. For example,

```
1,,100/ MOVE 1,@[1,,200]
```

computes a global effective address of 1,,200 because the indirect word has a global format.

5.1.2 IFIW with local index

If the address word is an IFIW and the index is local, the result is a local address. The 18-bit address is computed by adding the Y field to the right half of the contents of the index register. For example:

```
1,,100/ MOVE 1,[-1,,10]
1,,101/ MOVE 2,@[400001,,200]
```

The indirect word has an IFIW format, so bits 14-17 specify the index register address. Since the contents of the index register are negative, it is a local index and the EA-calc is performed by adding the Y field (200) to the right half of the index register (10) to produce a local effective address of 210.

5.1.3 IFIW with global index

If the address word is an IFIW and the index is global, the result is a 30-bit global address. The address is computed by adding bits 6-35 of the contents of the index register to the value of the Y field, that has been sign-extended from bit 18 into bits 6-17. For example:

```
1,,100/ MOVE 1,[2,,10]
1,,101/ MOVE 2,-2(1)
```

The second instruction word has an implicit IFIW format, so bits 14-17 specify the index register address. Since the left half of the index register is positive non-zero, it is a global index and the EA-calc is computed by adding the Y field, after sign-extending it from bit 18 into bits 6-17 (7777,,-2), to bits 6-35 of the contents of the index register (2,,10), producing a global effective address of 2,,6.

Note that the sign extension allows Y to be used as a positive or negative constant offset to the global address in an index register. This offset is limited to +/- 128K.

5.1.4 EFIW with global index

If the address word is an EFIW, the index is always assumed to have the global format and the result is a 30-bit global address. The address is computed by adding bits 6-35 of the contents of the index register to bits 6-35 of the Y field. For example:

```
1,,100/ MOVE 1,[2,,10]
1,,101/ MOVE 2,@[010002,,200]
```

The indirect word has an EFIW format, so bits 2-5 specify the index register address. The index is always global, so the EA-calc is computed by adding the Y field (2,,200) to bits 6-35 of the contents of the index register (2,,10) to produce a global effective address of 4,,210.

5.1.5 References to section zero

Note that the only way to reference section zero from a non-zero section is via an EFIW format indirect word with bits 6-17 equal zero. Indexing alone cannot be used to reference section zero, because an index with bits 6-17 equal zero is treated as a local address to the section from which the last address word was fetched.

5.1.6 Summary of EA-calc rules

The preceding sections can be summarized by the table that follows. This table gives the computation done for all combinations of address words and index registers formats plus an indication as to whether the result is local or global.

Address Word Type	Index Reg Type	IFIW	EFIW
None		Y[18:35] Local	Y[6:35] Global
Local	Local	Y[18:35]+(XR)[18:35] Local	Not Defined (Actually the case below)
Global	Local	Y[18]*7777,,Y[18:35]+(XR)[6:35] Global	Y[6:35]+(XR)[6:35] Global

5.2 Results of an EA-calc

When the microcode performs an EA-calc, it is simply following the rules described above and shown graphically in the EA-calc flow chart from the PRM. The result of this EA-calc is a 30-bit address and a 1-bit flag that indicates the address is local or global. These two pieces of information must be considered together whenever the results of the EA-calc are used; it is seldom, if ever, correct to consider the address without also considering the local/global bit.

Every EA-calc carries a default section along during the calculation of the effective address. The initial default section for an EA-calc of an instruction is PC section. More generally, the default section is initially that from which the first address word was fetched. This default section is changed from the initial value if the EA-calc follows a global address into another section. In fact, the default section is always the section from which the last address word was fetched.

If a local address is calculated using the rules given above, the default section is applied to complete the 30-bit address. If a global address is calculated, the default section is not used.

The last iteration of the EA-calc (the computation done on the last address word that doesn't have the indirect bit set) determines whether or not the result of the EA-calc is local or global. If the result of the last iteration is a local address, the result of the EA-calc is local. Similarly, if the result of the last iteration is global, so is the entire EA-calc. The transitions of the local/global flag are indicated on the PRM flow chart by notations such as "E Global".

The significant thing to remember is that a local EA-calc still results in a 30-bit address, even though 12 bits (the section number) were not explicitly supplied to the EA-calc routines as part of an address word or an index register.

- An effective address calculation always computes 31 bits of information: a 30-bit address, and a 1-bit local/global flag.

5.3 Simple EA-calc examples

In the examples above, we ignored the fact that EA-calc's always produce a 30-bit address when we said that the result was a local address n. In the following examples, we emphasize that a full 30-bit address is produced. Consider the following instruction:

```
0,,200/ MOVE 1,100
```

The EA-calc for this instruction results in a local EA. Therefore, the EA-calc computes the 30-bit address as 0,,100 and

the 1−bit local/global flag as local. Since the EA is local, we know that the section number was defaulted from something, in this case, the PC section. We say that the effective address is 0,,100 LOCAL (this notation is used throughout the rest of this discussion to specify all 31 bits of information).

Let's consider a slightly more complex example:

```
        1,,200/ MOVE 1,@300

        1,,300/400000,,100
```

As in the previous example, the effective address calculation computes a local address of 100. Since the address word was fetched from section 1, the result of the EA−calc is 1,,100 LOCAL.

Let's look at a global EA−calc:

```
       1,,100/ MOVE 1,@[2,,200]
```

In this case, the effective address calculation produces a global address of 2,,200 GLOBAL and no default section need be applied.

6.0 Use of the local/global flag

There are two uses for the local/global flag. First, it is used to determine if the address is actually an AC. If the address is local, and bits 18−35 are in the range 0 to 17, inclusive, the address references an AC, independent of bits 6−17. This means that a program can reference the ACs while running in any section, as long as the reference is local.

Second, the local/global flag determines how to increment or decrement the address. If the address is local, incrementing or decrementing it suppresses carries from bit 17 to bit 18 and vice versa. That is, the address always wraps around in the current section if the right half is incremented past 2^18−1 or decremented past 0. A global address is handled as a full 30−bit quantity and overflow or underflow of the right half can affect the left half section number.

6.1 AC references

Let's look at several examples that make use of the local/global flag. First, let's compare what happens to AC references for local and global effective addresses.

```
        2,,100/ MOVE 1,@[400000,,5]
```

The EA−calc for this instruction yields 2,,5 LOCAL, where the section number was defaulted to 2. Is this memory location 2,,5 or AC 5? Because the EA−calc is local, the rule says that it is an AC reference and not a memory reference. On the other hand, the EA−calc for

```
        2,,100/ MOVE 1,@[2,,5]
```

results in an EA of 2,,5 GLOBAL. Since the EA is global, this is a memory reference and not an AC reference.

- EA−calc's which yield local addresses, where bits 18−35 of EA are in the range 0−17, inclusive, always refer to the ACs independent of the section number.

Finally, there is the concept of "global AC address". This concept allows a program running in any non−zero section to make a global reference to the ACs by computing a global address in the first 16 (decimal) locations of section 1. Consider the following example:

```
        2,,100/ MOVE 1,@[1,,5]
```

The EA−calc yields 1,,5 GLOBAL and because of the "global AC address" rule, the reference is to AC 5 instead of memory location 1,,5.

- An EA−calc which yields a global address to locations 0−17, inclusive, of section 1, refers to the ACs and not to memory. Such an address is called a global AC address.

6.2 Incrementing EA

Another use for the local/global flag computed as the result of an EA−calc is to determine how to increment the effective address. Let's look at two examples using DMOVE, one computing a local EA and one computing a global EA.

```
        2,,100/ DMOVE 1,@[400000,,777777]
```

The EA−calc for this instruction results in an effective address of 2,,777777 LOCAL. The DMOVE instruction fetches two contiguous words from E and E+1, but what is E+1 in this case? Since the EA−calc resulted in a local address, incrementing E is done section−local, resulting in 2,,0 LOCAL for E+1. But this is a local reference to the ACs, so the two references for E and E+1 go to 2,,777777 (memory) and 2,,0 (AC). Note that the state of the local/global flag is maintained during the incrementing of EA.

- Incrementing or decrementing a local address is always done relative to the original section, i.e., the addresses "wrap around" in section.

- Incrementing a local address whose in−section part is 777777 causes the address to wrap around into the ACs.

Let's look at the corresponding global case:

```
        2,,100/ DMOVE 1,@[2,,777777]
```

In this case, the EA−calc yields 2,,777777 GLOBAL. Because this is a global address, incrementing E to get the second word results in a reference to 3,,0 GLOBAL. Since this isn't a local address, the reference is made to memory location 3,,0 and not to AC 0.

- Incrementing or decrementing a global address affects the entire address; i.e., section boundaries are ignored.

- The process of incrementing or decrementing an address, whether the address is local or global, preserves the state of the local/global flag.

7.0 Multi−section EA−calc's

So far we have considered only EA−calc's that remain in one section. If the program is running in a non−zero section, a global quantity encountered during the EA−calc (from either an index register or indirect word) can cause the EA−calc to "change sections". An example will make this more clear:

```
        3,,100/ MOVE 1,@[200002,,100]
        2,,100/ 3,,200
```

The EA−calc for this instruction computes a global address of 2,,100 from the indirect word in the literal. Since the indirect bit is set in this word (bit 1 is the indirect bit in an EFIW), the EA−calc routine fetches the word at 2,,100 and continues the EA−calc. The final result of the EA−calc yields 3,,200 GLOBAL. This isn't a very interesting example, because it doesn't demonstrate the significance of the section change, so let's look at a slightly different example:

```
        3,,100/ MOVE 1,@[200002,,100]
        2,,100/ 400000,,200
```

In this example, the first part of the EA−calc remains the same and the routine fetches the word at 2,,100. In this case, however, the result of the EA−calc yields a local address instead of a global one. But what section is the address local to? The rule says that a local address is always local to the section from which the address word was fetched. Since the EA−calc changed from section 3 to section 2 when the last address word was fetched, the EA−calc is relative to section 2 and the EA−calc yields 2,,200 LOCAL.

- The default section for a local address is always that from which the address word was fetched.

Now that we've seen what happens to EA−calc's that cross section boundaries, let's see what happens if the EA−calc enters section zero:

```
        3,,077/ MOVEI 3,1
        3,,100/ MOVE 1,@[200000,,100]
        0,,100/ 3,,200
```

As with the example above, the EA−calc for this instruction fetches the word at 0,,100 and continues. But since the EA−calc entered section zero, this word is treated as an IFIW instead of an EFIW. Therefore, the 3 in the left half of 0,,100 is interpreted as the index register field instead of a global section number. Since AC 3 contains a 1, the EA−calc yields 0,,201. In addition, the last address word was fetched from section zero, so the result is a local address.

- An effective address calculation which "falls" into section zero always results in an effective address that is local (to section zero). Furthermore, the effective address calculation can never "get out" of section zero once it enters it because all addresses in section zero are treated as local. Further operations obey section zero rules.

8.0 Special case instructions

Other than modifications to the EA-calc algorithms when the PC is in a non-zero section, most instructions are unaffected by the addition of extended addressing. However, there are a few classes of instructions that behave differently on an extended machine from the way they would on a non-extended machine. This section describes the behavior of each class of instruction that has this characteristic.

Examples in this section sometimes use the POINT pseudo-op to describe a byte pointer. For those readers who do not know what this pseudo-op generates, a description can be found in the Macro manual.

8.1 Byte instructions

The effective address calculation for a byte instruction addresses the byte pointer word(s). The instruction then does another EA-calc on the byte pointer after determining which one of the three possible byte pointer formats was supplied.

8.1.1 Byte pointer interpretation

The algorithm for determining the type of the byte pointer is as follows:

Byte pointer decode algorithm

The "Section 0?" test in the flow chart is based on where the first word of the two-word global byte pointer was fetched from

and not on PC section. This is an important distinction if the byte instruction and the byte pointer are not in the same section.

- For byte instructions, the test for the possibility of a two-word global byte pointer is done based on the section from which the first word of the byte pointer was fetched. That is, if the section from which the first word of the byte pointer was fetched is non-zero, the byte pointer may be global.

8.1.2 Byte pointer EA-calc

The default section for the byte pointer EA-calc is initially that from which the byte pointer was fetched. Once again, this may be different from PC section if the instruction and byte pointer are in different sections. If we realize that the byte pointer is really an address word, this is an extension of the rule that says local addresses are local to the section from which the address word was fetched. For example:

```
3,,100/ LDB 1,@[2,,100]
2,,100/ POINT 6,200,0
```

In this example, the byte instruction is fetched from section 3. The EA-calc for the instruction follows an EFIW into section 2 and the byte pointer is fetched. The byte pointer is in one-word local format, so the EA-calc of the byte pointer results in a local address. But is the address local to section 3 (section containing the byte instruction) or 2 (section containing the byte pointer)? The rule says that byte pointer EA-calc's start off local to the section from which the byte pointer was fetched, so the EA-calc is local to section 2. The result of the EA-calc is therefore 2,,200 LOCAL.

Note that, while the initial default section may be that containing the byte pointer, the default section may change if the EA-calc encounters a global quantity. For example:

```
3,,100/ LDB 1,@[2,,100]
2,,100/ POINT 6,@[200004,,100],0
4,,100/ 400000,,200
```

As in the previous example, the byte pointer is fetched from section 2. The byte pointer has the indirect bit set, so the byte pointer EA-calc follows the EFIW in the literal (which also has the indirect bit set) into section 4, where the final address word is fetched from location 4,,100. This final address word is an IFIW, so the result of the EA-calc is a local address. Even though the byte pointer EA-calc started in section 2, the result of the EA-calc is local to section 4, because that's where the last address word was fetched from. The byte pointer EA-calc results in an effective address of 4,,200 LOCAL.

- For byte instructions, the initial default section for the byte pointer EA-calc is the section from which the byte pointer was fetched, which may not be the same section as that containing the byte instruction. Further, if the EA-calc results in a local address, the address is local to the section from which the last address word in the effective address calculation was fetched.

8.2 EXTEND instructions

Like the byte instructions, certain EXTEND instructions perform another EA-calc for the byte pointer (MOVSxx, CMPSxx, CVTBDx, CVTDBx, and EDIT). The AC field of the EXTEND instruction addresses a block of ACs, that contain the byte pointers. In addition, some EXTEND instructions perform an EA-calc on the extended opcode word, which is interpreted in IFIW format. The extended opcode word is addressed by the effective address of the EXTEND instruction.

8.2.1 Byte pointer interpretation

The algorithm for determining the byte pointer format is the same as that described for byte instructions with one exception. For EXTEND instructions, the "Section 0?" test in the flow chart is based on PC section.

- For EXTEND instructions, the test for the possibility of a two-word global byte pointer is done based on PC section. That is, if PC section is non-zero, the byte pointers may be global.

8.2.2 Byte pointer EA-calc

The default section for the byte pointer EA-calc is initially PC section even if other parts of the EXTEND instruction are in other sections. For example:

```
3,,100/ MOVEI 1,5             ;Source length
3,,101/ MOVE 2,[POINT 7,200]  ;Source byte pointer
3,,102/ MOVEI 4,5             ;Destination length
3,,103/ MOVE 5,[POINT 7,300]  ;Destination byte pointer
3,,104/ SETZB 3,6             ;Clear 2nd word of BPs
3,,105/ EXTEND 1,@[2,,100]

2,,100/ MOVSLJ                ;Extended opcode is MOVSLJ
2,,101/ 0                     ;Fill character is 0
```

In this example, the EXTEND instruction is in section 3 and the EA−calc of the instruction follows an EFIW into section 2. The EA−calc's for the one−word local byte pointers in ACs 2 and 5 generate local addresses of 200 and 300 respectively. But are they local to section 3 (PC section) or to section 2 (section containing the extended opcode)? Because the byte pointers are fetched from the ACs, which are implicitly in PC section, the EA−calc is relative to PC section. Once again, this is a conceptual extension to the rule that local addresses are local to the section from which the address word (in this case, the byte pointer) was fetched.

As with byte instructions, the default section of the EA−calc may change if the EA−calc encounters a global quantity. An example of this for the EXTEND instruction would be analogous to that for byte instructions given above.

- For EXTEND instructions, the initial default section for the byte pointer EA−calc is PC section.

One interesting aspect of this rule is demonstrated by the following example:

```
3,,100/ MOVEI 1,5             ;Source length
3,,101/ MOVE 2,[POINT 7,200]  ;Source byte pointer
3,,102/ MOVEI 4,5             ;Destination length
3,,103/ MOVE 5,[POINT 7,300]  ;Destination byte pointer
3,,104/ SETZB 3,6             ;Clear 2nd word of BPs
3,,105/ EXTEND 1,@[0,,100]

0,,100/ MOVSLJ                ;Extended opcode is MOVSLJ
0,,101/ 0                     ;Fill character is 0
```

In this example, the EXTEND instruction is in a non−zero section (3) and the extended opcode is in section zero. Even though part of the processing of the instruction fell into section zero, the EA−calc of the byte pointers is still done relative to PC section. Hence, the result is the same as in the previous example.

8.2.3 Extended opcode EA−calc

Some EXTEND instructions also perform an EA−calc on the extended opcode word. In this case, the default section for the EA−calc is initially the section from which the extended opcode word was fetched. For example:

```
3,,100/ MOVEI 1,5             ;Source length
3,,101/ MOVE 2,[POINT 7,200]  ;Source byte pointer
3,,102/ MOVEI 4,5             ;Destination length
3,,103/ MOVE 5,[POINT 7,300]  ;Destination byte pointer
3,,104/ SETZB 3,6             ;Clear 2nd word of BPs
3,,105/ EXTEND 1,@[2,,100]

2,,100/ MOVST 200             ;Extended opcode is MOVST
2,,101/ 0                     ;Fill character is 0
```

As in the last example, the EXTEND instruction EA−calc follows an EFIW into section 2 to fetch the extended opcode word from location 2,,100. In this example, the extended opcode turns out to be a MOVST which addresses a translation table with the result of the EA−calc of the word. This EA−calc results in a local address which is local to the section from which the address word was fetched. Therefore, the table is read from locations starting at 2,,200 LOCAL.

- The initial default section for the EA−calc of the extended opcode word under an EXTEND instruction is that from which the extended opcode word was fetched.

8.2.4 EDIT pattern and mark addresses

In addition to byte pointer type determination, the EDIT instruction under EXTEND interprets the pattern string and mark addresses differently based on PC section. If PC section is zero, both addresses are limited to 18−bit addresses in section zero and the result of setting bits 6−17 non−zero is undefined. Conversely, if PC section is non−zero, both addresses are treated as full 30−bit global addresses and no default sections are applied. An example of this is too complex to be given here and will be left as an exercise to the reader.

8.3 JSP and JSR

In a non−extended machine, these two instructions store the flags and an 18 bit PC before jumping to the effective address. This is also true if the instructions are executed in section zero of an extended machine. Because this format is insufficient to store a full 30−bit address, the operation of the instructions is modified when the PC is in a non−zero section. Instead of storing the

flags and PC, these instructions store the full 30-bit PC (actually PC+1), omitting the flags. For example:

```
2,,100/ JSP 1,200
```

stores 2,,101 in AC 1 before jumping to location 2,,200. Similarly,

```
2,,100/ JSR 200
```

stores 2,,101 in 2,,200 before jumping to location 2,,201. Note that for JSR, the PC is stored in the word addressed by the effective address even if that address is in another section, e.g.,

```
2,,100/ JSR @[3,,200]
```

In this case, the EA-calc for the JSR results in an effective address of 3,,200 GLOBAL. Therefore, 2,,101 (PC+1) is stored in 3,,200 (EA) before jumping to 3,,201 (EA+1).

An interesting aspect of this is demonstrated by the following example:

```
2,,100/ JSP 1,@[0,,100]
```

Because the PC is in a non-zero section, the instruction stores 2,,101 in AC 1 and then jumps to location 0,,100. But an attempt to return to the caller in section 2 via the usual JRST (1) instruction would fail, because the EA-calc of the return instruction, done in section zero, would fail to produce a 30-bit global address. As a result, it is difficult to write a subroutine in section zero that can be called via JSP or JSR from an arbitrary section.

A final example illustrates the difference between a local and global EA for JSR:

```
2,,200/ JSR 777777
```

The EA-calc for this case results in a value of 2,,777777 LOCAL. Therefore, 2,,201 (PC+1) is stored in 2,,777777 (EA) and the destination of the jump is 2,,0 (EA+1 local). This is consistent with the rule that local addresses always wrap around in section when incremented.

The global analogy is as follows:

```
2,,200/ JSR @[2,,777777]
```

In this case, the result of the EA-calc is 2,,777777 GLOBAL so the instruction stores 2,,201 (PC+1) into location 2,,777777 (EA) as in the last example. The difference is in the destination of the jump. Because the effective address is global, incrementing it produces 3,,0 GLOBAL (EA+1 global) as the destination of the jump.

See the section on instruction fetches below for additional information on these two cases.

- If PC is in a non-zero section, the JSP and JSR instructions store a full 30-bit PC in the appropriate place instead of storing flags and PC. This is true even if the destination of the jump is in section zero.

8.4 Stack instructions

In a non-extended machine (and an extended machine in section zero), the stack pointer typically contains a negative control count in the left half and an 18-bit address in the right half. Such a stack pointer is called a local stack pointer. Because this format is insufficient to hold a full 30-bit stack address, an additional format for stack pointers is allowable when the PC is in a non-zero section. In this format (called a global stack pointer), the stack pointer is positive, bits 6-17 are non-zero, and bits 6-35 of the word are interpreted as the global address of the stack.

If the stack pointer is in local format, the stack address is local to PC section. For example:

```
2,,100/ MOVE 17,[-100,,200]
2,,101/ PUSH 17,300
```

Because the left half of the stack pointer is negative, it is in local format. Therefore, the stack address is 2,,200 LOCAL, because the stack is local to PC section.

- Local stack pointers are always local to PC section.
- The test for the possibility of a global stack pointer is done based on PC section. That is, if PC section is non-zero, the stack pointer may be global.

Note that a PUSH-type stack operation done on a local stack pointer that has overflowed (i.e., the left half of the pointer has gone to zero) changes the stack pointer to global format.

The type of stack pointer also determines how the stack address is incremented or decremented. For example, consider the following:

```
2,,100/ MOVE 17,[-100,,777777]
2,,101/ PUSH 17,200
```

The stack pointer in this example is local, so the stack address is 2,,777777 LOCAL. When the PUSH instruction increments the pointer, it does so section-local, resulting in an incremented stack address of 2,,0 LOCAL (which actually references AC 0). The stack pointer would then look like -77,,0.

Let's look at the same example with a global stack pointer:

```
2,,100/ MOVE 17,[2,,777777]
2,,101/ PUSH 17,200
```

With a global stack pointer, the increment is done globally, resulting in an incremented stack address of 3,,0 GLOBAL (which is memory location 0 in section 3). The stack pointer would then look like 3,,0.

- o Incrementing or decrementing a local stack pointer wraps around in section. Conversely, the same operation on a global stack pointer may cross section boundaries.

In addition to the requirement for a global stack pointer to specify a full 30-bit stack address, the operation of the PUSHJ and POPJ instructions is modified when the PC is in a non-zero section. Like JSP and JSR, PUSHJ stores a full 30-bit PC (again, actually PC+1) on the stack, omitting the flags. Similarly, POPJ restores a full 30-bit PC from the stack instead of an 18-bit PC local to PC section. Let's look at some examples:

```
2,,100/ MOVE 17,[-100,,200]
2,,101/ PUSHJ 17,400
```

Because PC section is non-zero, the PUSHJ stores 2,,102 on the stack at location 2,,201, which was addressed by a local stack pointer, and then jumps to location 2,,400. An updated stack pointer of -77,,201 is stored back into AC 17. Similarly:

```
2,,400/ MOVE 17,[-77,,201]
2,,401/ POPJ 17,
```

restores the full 30-bit PC from stack location 2,,201 (addressed by the local stack pointer) and then stores an updated stack pointer of -100,,200 back into AC 17.

This behavior has some interesting aspects, as the next example demonstrates:


```
2,,100/ MOVE 17,[2,,200]
2,,101/ PUSHJ @[0,,300]
```

Because PC is in a non-zero section, the PUSHJ instruction stores a full 30-bit PC (2,,102) on the stack at location 2,,201 (addressed by the global stack pointer). The jump is then made into section zero. But an attempt to return to the caller with a POPJ instruction will result in bedlam. In the first place, the global stack pointer will be interpreted as a local one in section zero. In addition, POPJ will assume that the stack word contains flags and PC and restore an 18-bit PC, local to section zero.

As this example demonstrates, it isn't very practical to call subroutines in section zero, from a non-zero section, using the normal call/return conventions.

- o If PC is in a non-zero section, the PUSHJ instruction stores a full 30 bit PC on the stack. This is true even if the destination of the jump is in section zero and regardless of the format of the stack pointer.

- o If PC is in a non-zero section, the POPJ instruction always restores a full 30-bit PC from the stack.

8.5 JSA and JRA

These instructions use a format that is incompatible with extended addressing. Because they are also considered an obsolete method for subroutine call/return, no attempt has been made to find an alternate format for these instructions when executed in a non-zero section.

For compatibility with section zero programs, these two instructions continue to work in non-zero sections. However, their use is restricted to intra-section operation, and all inter-section use is undefined.

In the case of JSA, the effective address is calculated in the normal manner. However, if the EA-calc results in an address outside of PC section, the action of the instruction is undefined. For example, the results of:

```
2,,100/ JSA 1,@[3,,200]
```

are undefined because the effective address is in section 3 and PC section is section 2. Note that a JSA which computes a global effective address which addresses the last word of PC section is also undefined. Let's look at an example of why this is true:

```
2,,100/ JSA 1,@[2,,777777]
```

In this case, the microcode would store the contents of AC into 2,,777777 and attempt to jump to E+1. But because EA is global, the computation of E+1 would result in 3,,0 GLOBAL which is outside of PC section.

The normal usage of JRA is of the form JRA AC,(AC) and the operation of the instruction is defined to take this into account. After the normal effective address calculation is performed, PC section is appended to the in-section addresses in AC to form the address of where the old contents of AC were stored and the new PC address. This forces all references to be in PC section. For example,

```
2,,201/ MOVE 1,[200,,101]
2,,202/ JRA 1,(1)
```

restores AC from location 2,,200 (PC section plus contents of AC left) and then jumps to 2,,101 (EA in PC section).

These definitions for JSA and JRA are consistent with the operation of the instructions in section zero.

- The use of JSA and JRA in a non-zero section is restricted to the case where the EA-calc results in an address in PC section. All inter-section usage is undefined.

8.6 LUUOs

In a non-extended machine, LUUOs trap via a pair of locations (40 and 41) in exec or user virtual memory. Because this scheme is insufficient to support extended addressing, the operation of LUUOs is modified if the PC is in a non-zero section. In this circumstance, the LUUO is processed through a four-word block which is addressed by a word in the exec or user process tables. See the PRM for more details.

- If PC is in a non-zero section, LUUOs trap through a four-word block addressed by a location in the EPT (exec LUUO) or UPT (user LUUO).

8.7 BLT

The format used for source and destination addresses by BLT is insufficient to represent two 30-bit addresses. As a result, the XBLT instruction was added to the instruction set to allow block transfers from one arbitrary 30-bit address to another. Despite

this, BLT is still useful for intra-section block transfers, and the operation of the instruction has been changed slightly.

The initial source address is constructed by taking the 18-bit address in the left half of the AC and appending it to the section number and local/global flag from the effective address. Similarly, the initial destination address is constructed from the 18-bit address in the right half of the AC and the section number and local/global flag from the effective address. This means that transfers are always to and from the same section as that specified by the effective address, which need not necessarily be the same as PC section. Source and destination addresses are then incremented, section-local (even if EA is global) until the destination address is equal to EA. For example:

```
2,,100/ MOVE 1,[200,,300]
2,,101/ BLT 1,@[3,,302]
```

In this example, the EA-calc for the BLT results in 3,,302 GLOBAL. Using the rules above, the initial source and destination addresses would be 3,,200 GLOBAL and 3,,300 GLOBAL. Therefore, the following transfer would take place:

```
3,,200 => 3,,300
3,,201 => 3,,301
3,,202 => 3,,302
```

Let's look at an example that demonstrates the significance of incrementing the addresses section-local:

```
2,,100/ MOVE 1,[777776,,300]
2,,101/ BLT 1,@[3,,302]
```

As in the previous example, EA is 3,,302 GLOBAL and the initial destination address is 3,,300 GLOBAL. In this case, the initial source address is 3,,777776 GLOBAL and the following transfer takes place:

```
3,,777776 => 3,,300
3,,777777 => 3,,301
3,,0      => 3,,302
```

Note that the source address was incremented section-local even though it was a global address.

It is important to note that the local/global flag must be included in constructing the initial source and destination addresses even though the addresses are always incremented section-local. This is because the check for an AC reference is done by including this flag. Let's look at two examples, one whose EA is local and one whose EA is global:

```
        2,,100/ MOVE 17,[1,,200]
        2,,101/ BLT 17,201
```

In this case, the result of the EA−calc for the BLT is 2,,201 LOCAL. Therefore, the initial source and destination addresses are 2,,1 LOCAL and 2,,200 LOCAL, respectively. Because the source is a local address whose in−section part is in the range 0−17, it references AC 1. Now let's look at the global case:

```
        2,,100/ MOVE 17,[1,,200]
        2,,101/ BLT 17,@[2,,201]
```

In this case, the result of the EA−calc for the BLT is 2,,201 GLOBAL. Therefore, the initial source and destination addresses are 2,,1 GLOBAL and 2,,200 GLOBAL, respectively. In this case, the source address references memory location 2,,1 instead of the ACs because the effective address is global. In both cases, however, the addresses are incremented section−local.

- The initial source and destination addresses for BLT are constructed by appending the appropriate half of the AC to the section number and local/global flag from the effective address. Incrementing of source and destination addresses is always done section−local independent of the state of the local/global flag. However, the determination of AC reference is done via the normal rules by including the local/global flag.

8.8 XBLT

The XBLT instruction is the one exception to the rule that a section zero program cannot reference data in non−zero sections. In this one case, the contents of AC+1 (source pointer) and AC+2 (destination pointer) are always treated as 30−bit global addresses, even if the PC is in section zero. This means that a program running in section zero can allocate a non−zero section and XBLT code or data into it without having to jump into a non−zero section to do it.

- The source and destination addresses for XBLT are always interpreted as full 30−bit global addresses, even if the PC is in section zero.

This means that the final addresses left in AC+2 and AC+3 at the end of the XBLT may be inaccessible by other instructions in section zero. For example:


```
        0,,100/ MOVEI 1,777777        ;Word count
        0,,101/ MOVEI 2,20            ;Source address
        0,,102/ MOVE 3,[2,,100]       ;Destination address
        0,,103/ EXTEND 1,[XBLT]
```

In this example, the transfer is from 0,,20 to 2,,100, and the number of words transferred is 256K−1. The final source and destination addresses left in ACs 2 and 3 are 1,,17 and 3,,77 respectively.

- For XBLT, the final values stored in AC+2 and AC+3 for source and destination addresses are computed by adding the initial word count to the initial source and destination addresses. This computation is the same in all sections, including section zero.

8.9 JRSTF

If the PC is in a non−zero section, JRSTF traps as an MUUO. This is because JRSTF is usually used with an indirect word or index register with PC flags in the left half. It is quite likely that these flags would be mistaken for a global section number.

- If PC is in a non−zero section, JRSTF traps as an MUUO. XJRSTF should be used in a non−zero section.

8.10 XMOVEI and XHLLI

Unlike other immediate instructions that use only 18 bits of the effective address, these two instructions operate on all 30 bits of EA. XMOVEI returns the full 30−bit effective address in AC. XHLLI stores the section number of the effective address in the left half of AC, leaving the right half unchanged.

One important implication of these two instructions is that they convert a local reference to an AC in any non−zero section into the global form. For example:

```
        2,,100/ XMOVEI 1,6
```

The EA−calc of the XMOVEI results in 2,,6 LOCAL, which is a local reference to AC 6. This result is then converted to the global AC address of 1,,6 before being loaded into AC 1.

This conversion is not done if the AC reference is local to section zero. For example:

```
        2,,100/ XMOVEI 1,@[200000,,6]
```

In this example, the EA−calc follows an indirect EFIW into section zero. The result of the EA−calc is therefore 0,,6 LOCAL, which is a local reference to AC 6. Because the effective address is in section zero, it is not converted to the global form and 0,,6 is stored in AC 1.

- o If the effective address of an XMOVEI or XHLLI is a local reference to an AC in a non−zero section, the AC address is converted to a global AC address before being loaded into AC.

8.11 XCT

With the exception of the modification of the EA−calc rules in a non−zero section, the XCT instruction operates in the same manner as on a non−extended machine. The operation of the instruction being executed, however, may be affected. This section describes these cases and gives examples to demonstrate them.

8.11.1 Default section for EA−calc

If an instruction is executed by an XCT, the initial default section for the EA−calc of that instruction is the section from which the instruction was fetched. This may be different from PC section if the XCT and the executed instruction are in different sections. For example:

```
        3,,100/ XCT @[2,,100]

        2,,100/ MOVE 1,200
```

In this example, the XCT instruction is in section 3 and the executed instruction is in section 2. The Ea−calc for the MOVE yields a local address, which is local to the section from which the MOVE was fetched. Therefore, the result of the EA−calc is 2,,200 LOCAL. This rule allows one to XCT an instruction in another section and have local references generated by the executed instruction be local to the section containing the instruction.

- o The initial default section for the EA−calc of an instruction executed by XCT is that from which the instruction was fetched.

8.11.2 Relationship with skip and jump instructions

When a skip instruction is XCTed, the skip is always relative to PC section, i.e., the section containing the XCT (first XCT if there is a chain of XCTs). This is true even if the skip instruction is in another section. For example:

```
        3,,100/ XCT @[2,,300]

        2,,300/ SKIPA 1,200
```

In this example, an XCT in section 3 executes a skip instruction in section 2. Because this instruction always skips, the next instruction is taken from location 3,,102 (PC+2), not 2,,302 (instruction+2). However, the EA−calc of the SKIPA instruction results in 2,,200 LOCAL, so the contents of location 200 in section 2 are stored in AC.

- o If an XCT executes a skip instruction, the skip is always relative to PC section, even if the skip instruction is in another section.

The following example demonstrates the effect of XCTing a jump instruction:

```
        3,,100/ XCT @[2,,100]

        2,,100/ JRST 200
```

In this example, an XCT in section 3 executes a jump instruction in section 2. The EA−calc for the JRST results in an address local to section 2, so the next instruction is taken from 2,,200, not 3,,200.

- o If an XCT executes a jump instruction that jumps, the next instruction is fetched from the effective address of the jump. This is true even if the XCT and the jump are in different sections and the EA−calc of the jump results in a local address whose section is different from PC section.

8.11.3 PC storing instructions

When an XCT executes an instruction that stores PC as part of the operation of the instruction (e.g., PUSHJ, JSP, etc.), the value stored is relative to PC section (i.e., the XCT) and not the section of the executed instruction. For example:

```
3,,100/ XCT @[2,,200]

2,,200/ JSP 1,300
```

In this example, an XCT in section 3 executes a JSP in section 2. The next instruction is fetched from location 2,,300 because the EA−calc of the JSP is local to section 2. However, the PC stored in AC 1 is 3,,101 (XCT+1), not 2,,201 (JSP+1).

- If an XCT executes an instruction that stores PC as part of its execution, the value stored is relative to the XCT and not the executed instruction.

8.11.4 Local stack references

When an XCT executes a stack instruction that uses a local stack pointer, the stack pointer is local to PC section and not to that containing the stack instruction. For example:

```
3,,077/ MOVE 17,[−100,,300]
3,,100/ XCT @[2,,200]

2,,200/ PUSH 17,400
```

In this example, an XCT in section 3 executes a PUSH in section 2. Since the EA−calc for the PUSH results in a local address, the datum to be pushed is in the same section as the PUSH instruction (at location 2,,400). However, the stack pointer is local to PC section, not the section containing the PUSH. Therefore, the datum is stored on the stack at location 3,,301.

- If an XCT executes a stack instruction whose stack pointer is local, the stack is local to PC section, not the section containing the stack instruction.

8.11.5 Generalizations for XCT

The examples above cover specific relationships between XCT and the executed instruction. There are really two generalizations (one of which was given above) that can be made about XCT, as follows:

1. The initial default section for the EA−calc of an XCTed instruction is that from which the instruction was fetched, and not the section from which the XCT was fetched.

2. Any test of PC section for determining whether section zero rules or non−zero section rules apply is done based on the section from which the XCT instruction was fetched (the first one if there is a chain of XCTs). That is, PC section doesn't change because an XCT executes an instruction in another section.

9.0 Summary of default sections for EA-calc

After covering all the special case instructions, it is worthwhile to summarize the rules regarding the initial default section number for EA-calc's. The initial default section for any EA-calc is that from which the address word was fetched. This is true for the simple cases as well as the more complex cases. The following table gives the initial default section for the various kinds of EA-calc:

EA-calc class	Initial default section
Instruction	PC section
XCTed instruction	Section containing the executed instruction
Byte instruction byte pointer	Section containing the byte pointer
EXTEND instruction byte pointer	PC section
EXTEND instruction opcode word	Section containing the opcode word
Local stack pointer	PC section

10.0 Section zero vs. non-zero section rules

As the previous discussion of special case instructions indicates, some instructions do different things based on a test for section zero. However, this test isn't always on PC section. We have intentionally left out examples that demonstrate some of the boundary conditions that make extended addressing hard to document to avoid confusing the reader before the simple cases are understood. This section includes examples of these boundary conditions, and summarizes the rules for testing to see if section zero rules apply.

The first example illustrates the test for the possibility of a global byte pointer:

```
        3,,100/ LDB 1,@[0,,200]
        0,,200/ 000640,,300
        0,,201/ 400000,,400
```

In this example, the byte instruction is in section 3 and the byte pointer is in section 0. Note that bit 12 is set in the byte pointer which, if global byte pointers are allowed, would indicate a two-word global byte pointer. Is this byte pointer interpreted as a one-word local or two a word global byte pointer? The rule given in a previous section says that the test is made based on the section from which the byte pointer was fetched. Therefore, bit 12 is ignored, the byte pointer is interpreted in one-word local format, and the byte is fetched from the word at location 0,,300.

Let's look at a similar case involving both XCT and EXTEND:

```
      3,,100/ MOVEI 1,5             ;Source length
      3,,101/ MOVE 2,[440740,,500]  ;Source b.p. (1st wd)
      3,,102/ MOVE 3,[5,,100]       ;Source b.p. (2nd wd)
      3,,103/ MOVEI 4,5             ;Destination length
      3,,104/ MOVE 5,[440740,,300]  ;Destination b.p. (1st wd)
      3,,105/ MOVE 6,[5,,200]       ;Destination b.p. (2nd wd)
      3,,106/ XCT @[0,,100]         ;Execute EXTEND in section 0

      0,,100/ EXTEND 1,200

      0,,200/ MOVSLJ                ;Extended opcode is MOVSLJ
      0,,201/ 0                     ;Fill character is 0
```

In this example, the XCT is in section 3 and the entire EXTEND instruction is in section zero. Both the source and destination byte pointers have bit 12 set, which means they may be interpreted as two-word global pointers. But are they? The rule given in a previous section says that the test is made based on PC section, which is non-zero. Therefore, the byte pointers are two-word global and the string is moved from 5,,100 to 5,,200. If this seems like an anomaly, remember that the test is based on PC section because the byte pointers are fetched from the ACs. References to ACs addressed by the AC field of the instruction are

always made in PC section.

A final example combines an XCT with a JSR:

```
3,,100/ XCT @[0,,200]

0,,200/ JSR 300
```

In this example, the XCT is in section 3 and the JSR is in section zero. The EA-calc of the JSR is local to section zero, so the destination of the jump is 0,,301. But what is stored in 0,,300? The rule given in a previous section says that the test is based on PC section. Therefore, we store a full 30-bit PC (3,,101) into location 0,,300.

- The test for section zero rules vs. non-zero section rules is done based on PC section for all cases except byte instructions. This is true even if the instruction is an XCT which executes an instruction in another section (including section zero).

- The test for section zero rules vs. non-zero section rules for a byte instruction is done based on the section from which the byte pointer was fetched.

It is important to realize that PC section may be different from that containing the instruction being executed if an XCT (or chain of XCTs) is involved. PC section is always that from which the original instruction (the XCT if that instruction is involved) was fetched. This is a subtle distinction, but it is important in testing for section zero rules.

11.0 Special consideration for ACs

On the PDP-10, the ACs are both general purpose registers and also part of the virtual address space of every program. This dual use is convenient but also confusing when one is attempting to understand the rules of extended addressing. This section describes some of the aspects of the relationship between extended addressing and the use of the ACs.

11.1 AC references

An AC can be referenced in one of four ways as follows:

1. As a general purpose register through the AC field of an instruction.

2. As an index register through the index register field of an instruction or indirect word.

3. As a local memory reference to the first 16 (decimal) locations of any section.

4. As a global memory reference to the first 16 (decimal) locations of section 1.

In this discussion, we are concerned with the last two uses.

The rules for extended addressing say that memory references in section zero are always local. Therefore, a section zero memory reference can reference the ACs only if it is to the first 16 (decimal) locations in section zero. On the other hand, a memory reference in a non-zero section can reference the ACs in two different ways. If the memory reference is local, the ACs appear in the virtual address space of every section as the first 16 locations. For example, both

```
2,,100/ MOVE 1,2
```

and

```
5,,100/ MOVE 1,2
```

reference AC 2 even though the addresses are local to different sections.

In addition, the ACs may be referenced in a section-independent way via a reference to global address 1,,n, where n is in the range 0-17, inclusive. This means that an AC address can be passed between two routines running in a non-zero section, even if the routines are in different sections. For example:

```
5,,100/ MOVE 16,[1,,6]     ;Get global AC address for AC
5,,101/ PUSHJ 17,@[3,,200] ; 6 and call routine
          :
          :
3,,200/ MOVE 1,(16)        ;Use global XR to fetch data
```

In this example, the calling routine in section 5 places the global AC address for AC 6 into AC 16 and calls a routine in section 3. Because 1,,6 is a global AC address, the called routine interprets the index in global format and the data is fetched from AC 6.

Note that an address of the form 1,,n, where n is in the range 0−17, will always reference the ACs, whether the address is local or global. If the address is local, the reference is a local reference to the ACs in section 1. If the address is global, it is a global AC reference to the ACs.

- An address of the form 1,,n, where n is in the range 0−17, inclusive, refers to the ACs whether it is a local or global address. Therefore, such an address can be used to refer to the ACs even if the state of the local/global bit is not known.

11.2 Instruction fetches

All instruction fetches are made as local references, even though the PC is a full 30−bit address. Therefore, an instruction is fetched from the ACs whenever bits 18−35 of PC are in the range 0−17, inclusive, independent of the section number. Consider the following example:

```
1,,100/ XJRST [3,,2]
```

This instruction sets the PC to 3,,2. However, the next instruction fetch will come from AC 2 because it is made as a local reference.

This behavior can have some implications for instructions that also store information before changing PC. Consider the following example:

```
1,,100/ JSR @[3,,2]
```

The JSR stores the current PC into memory location 3,,2 and then changes the PC to 3,,3. The next instruction is then fetched from AC 3 because of the local reference, but the old PC is in memory and must be fetched with a global reference.

- Instruction fetches from C(PC) are always made as local references even if PC was previously set to a global address. This means that instruction fetches from the first 16 (decimal) locations of any section cause the instruction to be fetched from the ACs.

11.3 Storing PC

If an instruction that stores PC as part of its execution is fetched from the ACs, the PC is stored as a full 30−bit address if PC is in a non−zero section. For example:

```
3,,100/ MOVE 4,[JSP 2,200]
3,,101/ JRST 4
```

In this example, the MOVE instruction stores a JSP into AC 4, and the JRST instruction computes a local effective address that references the ACs. PC is set to 3,,4, but the next instruction is fetched from AC 4 because instruction fetches are always made as local references. Therefore, the next instruction to be executed is the JSP. Because PC section is non−zero (it is still 3), the JSP must store a full 30−bit PC into AC 2. The important thing to realize is that PC is 3,,4 and is not 0,,4 (a section zero AC address) or 1,,4 (a global AC address). Therefore the JSP stores 3,,5 (remember, it stores PC+1) into AC 2 and jumps to 3,,200.

- If an instruction that is fetched from AC stores PC as part of its execution, the PC stored is a full 30−bit address including PC section, if PC section is non−zero.

11.4 Storing EA for LUUO, MUUO and page fails

When an LUUO or MUUO is executed or an instruction page fails, the microcode stores some information about the exception in a block addressed by a word fetched from the UPT or EPT. The information stored includes the effective address (or reference address in the case of page fail) for the instruction that caused the exception. If the resulting effective address is a local reference to an AC in a non−zero section, the microcode converts this address to a global AC reference before storing it in the block. This is the same rule used for XMOVEI and XHLLI.

- If the effective address of an LUUO or MUUO, or an instruction that causes a page fail results in a local reference to the ACs in a non−zero section, the microcode converts the local AC reference to a global AC address before storing the result.

11.5 An example

Consider the following example that brings together all of these rules:

```
3,,100/ MOVE 6,[001000,,10]
3,,101/ JRST 6
```

In this example, the MOVE stores an LUUO (opcode 001) into AC 6 and the JRST sets PC to 3,,6. The following list indicates the significant actions that are performed to process the LUUO:

1. The EA−calc for the LUUO is performed and the result is 3,,10 LOCAL.

2. Because PC section is non−zero, the LUUO must be processed through a four−word block addressed by a location in the UPT.

3. PC+1 must be stored as a full 30−bit address, including section number. The value stored is 3,,7.

4. Because the EA−calc of the LUUO resulted in a local reference to AC 10, it must be converted to a global AC address before it is stored in the block. The value stored is therefore 1,,10.

12.0 PXCT

When the monitor is invoked by an MUUO, page fail, etc., the address space of the process that caused the invocation is potentially different from that of the monitor. In order to provide a communications mechanism between the monitor and the so−called "previous context", the PXCT (for Previous context XCT) instruction was defined. Although PXCT is normally considered as a separate topic from extended addressing, there are interactions between the two that make it desirable to talk about them together.

Because PXCT is legal only in exec mode, there is no need to define a new opcode for the instruction. Rather, the normal XCT opcode is used, and a non−zero AC field distinguishes a PXCT from a normal XCT. The opcode name PXCT is simply a notational convenience to emphasize that the executed instruction is making previous context references.

12.1 Previous context

For the purposes of this discussion, "previous context" is defined by three processor state variables: Previous Context Section (PCS), Previous Context User (PCU), and Previous AC Block (PAB). PCS is a 12−bit state register (5 on the KL10) that gives the value of PC section in the previous context at the time of the event that invoked the monitor. PCU is a 1−bit register that indicates that the previous context was user mode (as opposed to exec mode). PAB is a 3−bit register that gives the AC block number used by the previous context (there are typically multiple AC blocks implemented by a machine, 8 in both KL10 and KC10. The so−called "current ac block" is addressed by another 3−bit state register called Current AC Block, or CAB). Therefore, the previous context includes both the address space and ACs that were in use at the time of the event that invoked the monitor.

When a context change occurs as the result of an MUUO, page fail, interrupt, etc., the previous context state variables are set according to a set of rules that are defined for each type of context change. The specific rules aren’t important for the purpose of this discussion and the reader is referred to other sources for more information. The important point is that the state variable are set as the result of the context change.

In addition to being set on a context change, the monitor may also set the state variables explicitly when it desires to make an asynchronous reference to previous context.

These previous context state registers then direct references to the previous context as described below. Note that the previous context need not always be user mode. It is exec mode in cases where the monitor makes a request of itself, such as the execution of an MUUO by the monitor.

12.2 Use of the previous context state variables

The state registers PCS, PCU, and PAB hold information necessary to make a previous context memory or AC (as memory or index register) reference. This section describes the use for each register.

PCS is a 12−bit state variable that gives the value of PC section in the previous context. It is used in the PXCT EA−calc algorithm as described below to provide a default section number for a local EA−calc. It is also used as the basis for the test for section zero in some instructions that behave differently in non−zero sections as described below. (For most instructions, the effect is as if the instruction were executed in previous context.)

PCU is a 1−bit state variable that indicates that the previous context was user mode. PCU is used to select the address space for a previous context memory reference. That is, if the reference is to previous context and PCU is set, the reference is made to the user address space as mapped through the UPT. Conversely, if the reference is to previous context and PCU is not set, the reference is to the exec address space as mapped through the EPT.

PAB is a 3−bit state variable that gives the AC block number for the previous AC block. If an index register or AC is referenced in previous context, PAB gives the number of the AC block containing the data.

12.3 References to previous context

The PXCT mechanism allows the monitor to execute an instruction such that certain references of the executed instruction are made to the previous context. Conceptually, these references are made as if the PXCTed instruction were being executed in the previous context.

It is important to understand exactly which operations are modified by PXCT. The instruction fetch and EA−calc of the PXCT instruction and the fetch of the executed instruction are always done in current context. In addition, all AC references (as the result of bits 9−12 of the executed instruction) are made to the current context ACs. The only difference between an instruction executed under PXCT and one that is not is the way certain memory and index register references are made. In particular, the EA−calc of the executed instruction may reference indirect words and index registers in previous context. Also, memory and AC references made as the result of the EA−calc may be to previous context. Exactly which references are made in previous context is determined by the type of instruction that is being executed and by the bits set in the AC field of the PXCT instruction.

12.4 Applicable instructions

Not all instructions may be executed via PXCT. The use of PXCT is limited to instructions that are useful to the monitor, and no attempt is made to trap those cases that aren't applicable. The instructions that may be executed are as follows:

MOVE class instructions
Halfword class instructions
EXCH
XMOVEI, XHLLI
BLT (with restrictions), XBLT
Arithmetic (integer and floating point) instructions
Boolean instructions
DMOVE class instructions
CAI and CAM class instructions
SKIP, AOS, and SOS class instructions
Logical test instructions
PUSH and POP (with restrictions)
Byte class instructions
MOVSLJ (with restrictions)
MAP

All other instructions are inapplicable, and the results of executing an inapplicable instruction are undefined. Note that this list explicitly excludes all instructions that jump.

12.5 Interpretation of the AC field bits

The four bits of the AC field of the PXCT instruction determine which memory references of the executed instruction are made to previous context. For most PXCTed instructions, the AC field bits are logically grouped into two pairs (9−10 and 11−12) to control how EA−calc and data references are performed. Within each pair, the first bit (the generic "E control bit") causes index register and address word references to come from previous context during an EA−calc. The second bit (the generic "D control bit") causes data fetches as the result of instruction execution to come from previous context. When considered as a whole, bits 9−12 of the AC field are named "E1", "D1", "E2", and "D2" but the generic names ("E" and "D") may be used when it is clear which bits control the reference in question.

Not all executed instructions use both pairs of bits. In fact, the great majority of applicable instructions use only bits 9 and 10; bit 9 for the EA−calc of the PXCTed instruction and bit 10 for the data reference made as the result of that EA−calc. A notable example of the use of bits 11 and 12 to control previous context references is the byte instructions. In this case, bit 11 controls the EA−calc of the byte pointer and bit 12 controls the data reference to the word containing the byte. Some instructions use other combinations of bits, e.g., BLT, EXTEND (MOVSLJ and XBLT), and stack instructions.

The previous context memory references controlled by each AC field bit may be summarized by the following table:

Bit	References made in previous context if bit is 1
9 (E1)	Effective address calculation of instruction (index registers, indirect words).
10 (D1)	Memory operands specified by EA, whether fetch or store (e.g, PUSH source, POP or BLT destination); byte pointer.
11 (E2)	Effective address calculation of byte pointer; source in EXTEND (e.g., XBLT or MOVSLJ source); effective address calculation of source byte pointer in EXTEND (MOVSLJ).
12 (D2)	Byte data; source in BLT; destination in EXTEND (e.g., XBLT or MOVSLJ destination); effective address calculation of destination byte pointer in EXTEND (MOVSLJ).

There are obviously a limited number of valid combinations of AC field bits for those instructions that may be PXCTed. The following table gives the legal combinations. The "AC" column gives the AC field value for the equivalent bits, e.g., the AC column would contain a 4 for a 0 1 0 0 bit string.

Instructions	AC	E1 9	D1 10	E2 11	D2 12	References
General	4	0	1	0	0	Data
	14	1	1	0	0	E, data
PUSH, POP	4	0	1	0	0	Data
	14	1	1	0	0	E, data
Immediate	10	1	−	0	0	E (no data reference)
BLT	5	0	1	0	1	Source data, destination data
	15	1	1	0	1	E, source data, destination data
XBLT	2	0	0	1	0	Source data
	1	0	0	0	1	Destination data
	3	0	0	1	1	Source data, destination data
Byte	1	0	0	0	1	Byte data
	3	0	0	1	1	Pointer E, byte data
	7	0	1	1	1	Pointer, pointer E, byte data
	17	1	1	1	1	E, pointer, pointer E, byte data
MOVSLJ	1	0	0	0	1	Destination pointer E, destination data
	2	0	0	1	0	Source pointer E, source data
	3	0	0	1	1	Source pointer E, destination pointer E, source data, destination data

Note that BLT, PUSH, POP, and MOVSLJ have restrictions on what memory references can be PXCTed. For BLT, all references, optionally including the EA−calc, must be done in previous context. The results of PXCTing a BLT where source but not destination or destination but not source is in previous context are undefined. The LDPAC and STPAC instructions should be used to transfer the previous ACs to and from current context. In all other cases, XBLT must be used to transfer data between current and previous context.

For PUSH and POP, the stack must always be in current context. This means that previous context references for PUSH and POP are limited to the EA−calc and data reference made to the location addressed by the EA−calc. PUSH and POP therefore reduce to the "general" case.

For MOVSLJ, if source or destination data is in previous context, the source or destination byte pointer EA−calc must be done in previous context also. If the monitor wishes to force a current context EA−calc for a previous context data reference, it can compute the effective address of the byte word and use a one− or two−word global byte pointer. The microcode will still do the EA−calc in previous context, but no previous context defaults will be applied.

12.6 Modifications to the EA−calc algorithm

The appropriate "E" and "D" control bits from the AC field of the PXCT instruction are used to modify an EA−calc done on the executed instruction or a subsequent EA−calc done by the instruction (e.g., byte pointer). This modification involves pre− and post−processing the normal effective address calculation algorithms to conditionally include PCS at two points.

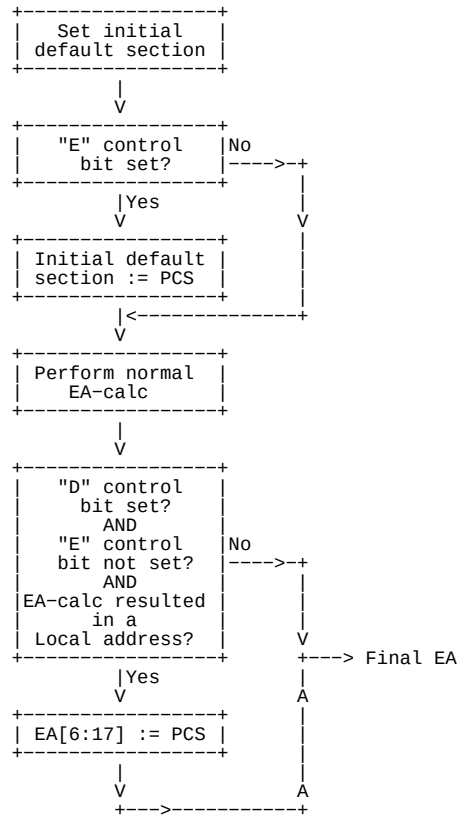
If the appropriate "E" control bit is set, the initial default section for the EA−calc is set to PCS. Since the "E" control bit also controls previous context indirect word and index register references, this means that the entire EA−calc is done in previous context. If the "E" control bit is not set, the initial default

section for the EA−calc is that from which the address word was fetched, and the EA−calc is done in current context.

When the normal EA−calc is completed, the resulting value is post−processed. If the result of the EA−calc was a local address AND the "E" control bit was not set AND the "D" control bit was set, the section number of the EA−calc is replaced by PCS. Note that the local/global flag remains local if this is done.

The application of PCS at the end of the EA−calc may seem to make no sense at first glance, so let's take a closer look at it. Remember that the purpose of PXCT is to allow the monitor to reference data in the previous context as if the user had supplied it. If the user supplies a local address in, for example, a JSYS argument, the monitor should make the data reference local to the section in which the user was running. By applying PCS at the end of the EA−calc as indicated above, the microcode automatically makes the reference to the correct section.

This algorithm may be described by the following flow chart:



PXCT EA−calc algorithm

Assume that PCS is 1 and consider the following example:

```
2,,100/ PXCT 4,[MOVE 1,100]
```

MOVE is one of the "general" class of opcodes, so bits 9 and 10 of the PXCT AC field control the previous context references. In this example, bit 9 (The "E1" bit) is off and bit 10 (the "D1" bit) is on. Therefore, the EA-calc is done in current context with a result of 2,,100 LOCAL. Because the "D1" bit is on, the "E1" bit is off, and the result of the EA-calc is local, the PXCT EA-calc algorithm applies PCS to bits 6-17 of the EA-calc. The final effective address is therefore 1,,100 LOCAL and the data reference is made to that location in previous context.

Let's look at another example. Assume that PCS is 2 and that the following locations exist in previous context:

```
2,,200/ 200003,,300

3,,300/ 400000,,400
```

In current context, the following instruction is executed:

```
1,,100/ PXCT 14,[MOVE 1,@200]
```

In this example, both the "E1" and "D1" bits are on in the PXCT AC field. Therefore, the EA-calc is done in previous context and the initial default section for the EA-calc is set to 2 (PCS). Location 2,,200 in previous context contains an indirect EFIW that the EA-calc follows into section 3. The final address word fetched from previous context location 3,,300 is in IFIW format, so the result of the EA-calc is local to the section from which the address word was fetched. The result of the EA-calc is 3,,400 LOCAL. Because the "D1" bit is also set, the MOVE fetches data from previous context location 3,,400.

A final example demonstrates the result of an EA-calc that references an AC. Assume that PCS is 3.

```
2,,100/ PXCT 4,[MOVE 1,2]
```

As with the first example, the EA-calc is done in current context and PCS is applied to bits 6-17 of the result to produce an effective address of 3,,2 LOCAL. Just as in the non-PXCT case, this is a local reference to AC 2. Because the "D1" bit is set, the reference is made to previous context AC 2 in the AC block specified by PAB.

- The EA-calc of a PXCTed instruction may be pre- or post-processed as directed by the AC field control bits of the PXCT instruction. Except for this additional processing, the EA-calc algorithms and results are exactly the same as for the non-PXCT case. This includes the uses for the local/global flag.

12.7 Section zero vs. non-zero section rules

Of the instructions that may be PXCTed, there are three types (stack, byte, and MOVSLJ) that operate differently in non-zero sections and section zero. When one of these instructions is PXCTed, the test for zero/non-zero rules may not be the same as the test when there is no PXCT involved. The interaction of PXCT with each of the instruction types is covered separately below.

12.7.1 Stack instructions

When no PXCT is involved, the test for the possibility of a global stack pointer is done based on PC section. When a PUSH or POP instruction is PXCTed, the previous context references are limited to the EA-calc and the datum addressed by the EA-calc, and the stack reference is always made in current context. Because the stack is in current context, the interpretation of the stack pointer type is made based on the current context PC section and is not dependent on PCS. For example, assume that PCS is 0.

```
2,,100/ MOVE 1,[3,,1000]
2,,101/ PXCT 4,[PUSH 1,200]
```

In this example, PC section is non-zero and the stack pointer in AC 1 has a global format. The test to determine whether the stack pointer is allowed to be global is still made based on PC section (even though there is a PXCT involved), and not on PCS. Therefore, the stack pointer is indeed global and previous context location 0,,200 is pushed onto the stack in current context location 3,,1001.

- When a stack instruction (PUSH, POP) is PXCTed, the test for the possibility of a global stack pointer is done based on PC section.

- When a stack instruction is PXCTed, local stack pointers are always local to PC section.

12.7.2 Byte instructions

Normally, the byte instruction test for the possibility of global byte pointers is done based on the section from which the byte pointer was fetched. When a byte instruction is PXCTed, this rule continues to apply, with extensions to include the possibility that the byte pointer may be fetched from previous context. This is best explained with several examples.

Assume that PCS is 0 and that the following locations exist in previous context:

```
0,,100/ 400000,,200

0,,200/ 12
```

In current context, the following instruction is executed:

```
2,,300/ PXCT 3,[LDB 1,400]

2,,400/ 000640,,0
2,,401/ 400020,,100
```

For PXCT of byte instructions, bits 9 (E1) and 10 (D1) direct the EA-calc of the byte instruction and the fetch of the byte pointer. Bits 11 (E2) and 12 (D2) direct the EA-calc of the byte pointer and the fetch of the word containing the byte. In this example, the "D1" bit is off, so the byte pointer is fetched from current context location 2,,400. Bit 12 is on in the byte pointer, and a test must be made to see if it may be global. The byte pointer is global because it was fetched from current context section 2, and the fact that PCS is zero is not considered.

The "E2" bit and the "D2" bit of the PXCT AC field are both on, so the byte pointer EA-calc is done in previous context. The second word of the two-word global byte pointer has the indirect bit set, and the next address word is fetched from previous context location 0,,100. The final result of the EA-calc is 0,,200 LOCAL in previous context and bits 30-35 of that word are extracted and placed in current context AC 1.

Let’s look at a similar example in which the byte pointer is also fetched from previous context. Once again assume that PCS is 0 and the previous context contains the following locations:

```
0,,400/ 000640,,100
0,,401/ 400000,,200

0,,100/ 10
0,,200/ 20
```

In current context, the following instruction is executed:

```
2,,300/ PXCT 7,[LDB 1,400]
```


In this case, the "D1" bit of the PXCT AC field is set, so the byte pointer is fetched from previous context location 0,,400. As in the last example, bit 12 is set in the byte pointer. But because the byte pointer was fetched from previous context section 0, bit 12 is ignored and the byte pointer is interpreted in one-word local format. The EA-calc is done in previous context and results in an effective address of 0,,100 LOCAL. The byte is then fetched from bits 30-35 of previous context location 100.

- When a byte instruction is PXCTed, the test for the possibility of a global byte pointer is done based on the section from which the byte pointer was fetched. This is true independent of whether the byte pointer is fetched from current or previous context.

This interpretation, while correct architecturally, causes some problems for TOPS-20 as it is implemented today because TOPS-20 copies byte pointers from the previous context into current context. Ideally, when a JSYS does a byte instruction on behalf of the user, the byte pointer would be interpreted exactly as if the user had executed the byte instruction. Thus, if the byte pointer were fetched from section 0, it would be interpreted as a local pointer; if it were fetched from any other section, it would be interpreted as possibly being global. This can be accomplished by using PXCT 7, as indicated in the example above.

Because TOPS-20 copies the byte pointer from the previous context into current context, one that looks like a global byte pointer will be interpreted as a global byte pointer even if it is fetched from previous context section zero. This is because the monitor typically runs in a non-zero section and the PXCTed byte instruction fetches the byte pointer from current context. Hence the test for the possibility of a global byte pointer is made based on current context section rather than previous context section.

12.7.3 EXTENDed MOVSLJ instruction

If no PXCT is involved, the MOVSLJ test for the possibility of a global byte pointer is made based on PC section. If a PXCT is involved, the test is more complex because it is based on PC section if the PXCT control bit for the byte pointer is off and on PCS if the PXCT control bit is on. For example, assume that PCS is zero and that previous context contains the following locations:

```
0,,200/ ASCII|ABCDE|

0,,300/ ASCII|FGHIJ|
```

In current context, the following instruction sequence is executed:

```
        3,,100/ MOVEI 1,5               ;Source length
        3,,101/ DMOVE 2,[440740,,200    ;Source BP (word 1)
                         400000,,300]   ;Source BP (word 2)
        3,,102/ MOVEI 4,5               ;Destination length
        3,,103/ DMOVE 5,[440740,,400    ;Destination BP (word 1)
                         400000,,500]   ;Destination BP (word 2)
        3,,104/ PXCT 2,[EXTEND 1,600]   ;PXCT the MOVSLJ

        3,,600/ MOVSLJ                  ;Extended opcode is MOVSLJ
        3,,601/ 0                       ;Fill character is 0
```

In this example, the "E2" bit is set in the PXCT AC field, which indicates that the source EA-calc and string reference are to be made to previous context. Conversely, the "D2" bit is off, which indicates that the destination EA-calc and string references are to be made to current context.

Because the source-in-previous control bit is set in the PXCT AC field, the test for the possibility of a global source byte pointer is made based on PCS. In this case, PCS is zero, so bit 12 is ignored in the byte pointer and it is interpreted in one-word local format. The byte pointer EA-calc results in 0,,200 LOCAL in previous context.

On the other hand, the destination-in-previous control bit is not set, so the test for the possibility of a global destination byte pointer is made based on PC section. Since PC section is non-zero and bit 12 is set, the byte pointer is interpreted in two-word global format, and the byte pointer EA-calc results in 3,,500 LOCAL in current context.

The result is to transfer the string "ABCDE" from previous context location 0,,200 to current context location 3,,500.

- When a MOVSLJ instruction is PXCTed, the test for the possibility of a global byte pointer is done based on PC section if the appropriate PXCT control bit is off. If the bit is on, the test is done based on PCS.

APPENDIX A

EA-CALC FLOWCHARTS

The following pages contain the EA-calc flowcharts from the Processor Reference Manual (page 1-30) and from the KL10 Engineering Functional Spec.

INDEX