```
+---------------------------+
|   |   |   |   |   |   |   |
| d | i | g | i | t | a | l |
|   |   |   |   |   |   |   |
+---------------------------+
```

i n t e r o f f i c e

m e m o r a n d u m

To: Jupiter design group

Date: 11 May 83
From: Mike Uhler
Dept: Jupiter Engineering
DTN: (8-)231-6448
Loc/Mail stop: MR01-2/E85
Net mail: UHLER at 10

Subject: Overview of the Jupiter EBOX and IBOX

1.0 Introduction

2.0 The PDP-10 instruction set

One of the features of the PDP-10 instruction set is its regularity. Although there are 400+ user-mode instructions, the instructions are logically divided into a relatively small number of instruction classes (e.g. compare, test, full word, etc.). Even within each class, the instructions are regularly defined.

Because of this regularity, one can often make generalizations about the operations that must be performed on any instruction. This section discusses some of these generalizations.

2.1 Instruction operands

One can logically divide the instruction set up into six categories based on the memory operand requirements for the instruction. The classes are as follows:

1. Immediate instructions. This class consists of instructions which neither fetch an operand from C(EA) nor store a result back to that location. Note that this class includes more instructions than those that use EA as a number (e.g., MOVEI) in that instructions like SETZ are immediate instructions, but do not use EA.

2. Single memory operand instructions. This class consists of instructions which fetch a single memory operand from C(EA), e.g. MOVE. The memory operand may be optionally write-tested as it is read because certain instructions in this class (e.g., AOS) store results back to C(EA) also.

3. Single memory store instructions. This class consists of instructions that store a single result to C(EA). They need not fetch the word from C(EA) because the instruction does not use that operand as input data to the instruction (e.g., MOVEM).

4. Double memory operand instructions. This class is similar to class 2 except that two memory operands are fetched from C(EA) and C(EA+1) (e.g., DMOVE). There is no write-test required in this class because no instruction both fetches a double memory operand and then stores back a double memory result.

5. Double memory store instructions. This class is similar to class 3 except that two results are stored to C(EA) and C(EA+1) (e.g., DMOVEM).

6. Complex instructions. This class consists of the "other" instructions, that is, those that don't fit into any of the classes given above. The number of instructions in this class is small enough to list explicitly, and they are as follows:

```
LUUO     MUUO
EXTEND
IBP      ADJBP    ILDB     LDB      IDPB     DPB
BLT
XCT
ADJSP    PUSHJ    PUSH     POP      POPJ     PUSHM
POPM     PUSHI
JRST     JFCL     JFFO
JSR      JSP      JSA      JRA
Privileged instructions
```

2.2 Jump instructions

The PDP-10 architecture includes several classes of instructions that jump, that is, that can change PC to something other than .+1 or .+2. Some of these instructions are unconditional jumps and some are conditional jumps.

A useful observation about the conditional jump instructions is that they always (with one exception) jump based on the data contained in the AC specified by the AC field of the instruction. Unlike architectures which jump based on condition codes, data directed jumps have some possibly interesting implications for IBOX prefetch. More on this later.

Not only do the conditional jump instructions jump based on data in AC, they uniformly make the decision based on a test for proximity to zero (i.e., < 0, <= 0, = 0, /= 0, >= 0, or > 0).

Some instructions add one to or subtract one from AC before making the proximity test for zero. If one wants to make the jump decision based on the original contents of AC rather than the contents after a +1 or -1 operations is carried out, one must have the capability to test for proximity to -1, 0, or +1.

The result is that we have the following 18 possible tests:

```
<  -1        <  0        <  +1
<= -1        <= 0        <= +1
=  -1        =  0        =  +1
/= -1        /= 0        /= +1
>= -1        >= 0        >= +1
>  -1        >  0        >  +1
```

There are a few exceptions to this rule and it's worth listing all the jump instructions.

2.2.1 Unconditional jumps

The unconditional jump instructions are as follows:

```
JRST
PUSHJ   POPJ
JSP     JSR
JSA     JRA
JUMPA   AOJA    SOJA
```

Note that all instructions in this class don't jump to EA (e.g., JSR and POPJ).

2.2.2 Conditional jumps that don't make a proximity test

The conditional jump instructions that don't strictly adhere to the proximity text on the full AC are:

AOBJN	Jump if AC<0:17> < -1
AOBJP	Jump if AC<0:17> >= -1
JFFO	Jump if AC /= 0
JFCL	Jump if any specified flag was set

Note that AOBJN and AOBJP can simply test bit 0 of the result if the test is made after the +1,,1 operation.

2.2.3 Conditional jumps that make a proximity test

Finally, there are several classes of conditional jump instructions that do make a proximity test. The instructions and the test that they use are as follows:

Instruction within class

		L	LE	E	N	GE	G
	JUMPx	< 0	<= 0	= 0	/= 0	>= 0	> 0
Class	AOJx	< -1	<= -1	= -1	/= -1	>= -1	> -1
	SOJx	< +1	<= +1	= +1	/= +1	>= +1	> +1

Note that all three classes reduce to the JUMPx case if the +1 or -1 operation is performed before the test.

2.3 Skip instructions

The PDP-10 architecture also defines instructions that skip, i.e., they can change PC to .+2. As with the jump instructions, there are unconditional and conditional skip instructions.

The conditional skip instructions are a bit more complex than the conditional jump instructions in that the the test condition is not simply a proximity test against a specified value. In fact, the conditional skip instructions can be logically divided into compare, skip, test, and miscellaneous sub-classes. Let's consider each of these separately.

2.3.1 Unconditional skips

The unconditional skips consist of the "skip always" instructions in the Compare, Skip, and Test instructions. They are:

CAIA	CAMA		
SKIPA	AOSA	SOSA	
TLNA	TLZA	TLOA	TLCA
TRNA	TRZA	TROA	TRCA
TDNA	TDZA	TDOA	TDCA
TSNA	TSZA	TSOA	TSCA

2.3.2 Conditional skips - Compare

The compare instructions compare C(AC) with either 0,,EA or C(EA) and skip if the appropriate condition is true. The conditions are the same as those for the conditional jump instructions. The instructions and the appropriate tests are given in the following table:

	CAIx	CAMx
L	C(AC) < 0,,EA	C(AC) < C(EA)
LE	C(AC) <= 0,,EA	C(AC) <= C(EA)
E	C(AC) = 0,,EA	C(AC) = C(EA)
N	C(AC) /= 0,,EA	C(AC) /= C(EA)
GE	C(AC) >= 0,,EA	C(AC) >= C(EA)
G	C(AC) > 0,,EA	C(AC) > C(EA)

Note that these can be converted into a proximity test against zero if the appropriate second operand (0,,EA or C(EA)) is subtracted from C(AC). There are other algorithms that don't require a subtract (e.g., see the KL10).

2.3.3 Conditional skips - Skip

The skip instructions are exactly symmetric with the JUMPx, AOJx, and SOJx conditional jump instructions. The data being tested against -1, 0, or +1 is in C(EA) rather than in C(AC) as with the jump instructions. These instructions, and the proximity test that they user, are as follows:

		Instruction within class					
		L	LE	E	N	GE	G
	SKIPx	< 0	<= 0	= 0	/= 0	>= 0	> 0
Class	AOSx	< -1	<= -1	= -1	/= -1	>= -1	> -1
	SOSx	< +1	<= +1	= +1	/= +1	>= +1	> +1

Note that, as with the conditional jump instructions, all three cases reduce to the SKIPx case if the +1 or -1 operation is performed before the test.

2.3.4 Conditional skips - Test

The test instructions logically AND C(AC) with a specified mask and skip if the result meets the specified condition. The instructions and the masking operation for each are as follows:

	E	N
TRxx	(C(AC) AND 0,,E) = 0	(C(AC) AND 0,,E) /= 0
TLxx	(C(AC) AND E,,0) = 0	(C(AC) AND E,,0) /= 0
TDxx	(C(AC) AND C(EA)) = 0	(C(AC) AND C(EA)) /= 0
TSxx	(C(AC) AND SC(EA)) = 0	(C(AC) AND SC(EA)) /= 0

Note that in the TSxx case, the notation SC(EA) means the datum fetched from EA with the halves swapped.

2.3.5 Conditional skips - Miscellaneous

This class of conditional skips is quite small. The only instructions in this class are certain complex EXTEND instructions and some of the privileged instructions (e.g., SNPI).

3.0 The IBOX

The job of an IBOX is to prefetch instructions and operands along an instruction stream that the EBOX or FPA will execute. If there were no conditional jump or skip instructions in the instruction set, this would be a straight-forward task because the IBOX would always know which instruction the EBOX or FPA would execute next.

The presence of conditional jump or skip instructions makes the job more difficult. In the general case, the IBOX might have to "guess" what the EBOX will do to a conditional jump or skip instruction. This is especially true if the IBOX is fetching far ahead of EBOX execution.

There have been several schemes in the past for solving the "where will the EBOX go?" problem, including:

- o Static branch prediction based on the opcode.
- o Dynamic branch prediction based on both the opcode and what the EBOX did with the instruction the last "n" times that it was executed.
- o Multi-stream prefetching to attempt to follow all possible paths.
- o "Pause and wait" schemes that forced the IBOX to wait for the EBOX to make a decision.

Combinations of these schemes have also been used.

As indicated in the previous section, the PDP-10 architecture has the advantage that the large majority of conditional jumps and skips jump or skip based on the data in AC or C(EA). The current scheme for the Jupiter IBOX is to make use of this fact and have the IBOX determine what the EBOX will do with the instruction as it sends it to the EBOX for execution. Note that such a scheme will never be 100% effective because the EBOX may be modifying AC or C(EA) as the IBOX is making the test. On the other hand, compiler technology exists to re-order instruction streams so that such conflicts are avoided. We may gain significant benefit in the future by teaching programmers the implications of programming for a pipelined machine design.

3.1 Functions that the IBOX provides.

As currently envisioned, the IBOX is responsible for setting up instructions for EBOX or FPA execution. It also provides certain other services to the EBOX. In particular, the IBOX does the following:

1. Instruction prefetch. The IBOX prefetches all instructions for EBOX and FPA execution. To do this effectively, it must

know what the EBOX is going to do with conditional jump and skip instructions. The result is that the IBOX needs the logic necessary to make the jump/skip tests that were listed in the section on the instruction set above.

An interesting aspect of such a decision is that the EBOX need not execute any instruction which affects only PC. Studies from the Dolphin project indicate that such instructions may be as high at 30% of the total executed instructions.

2. Instruction EA-calc. The IBOX performs all EA-calc's, including those that involve indirect chains. Because of finite conflict compare on indirect words, the IBOX must pause and wait for the EBOX to complete execution of the current instruction if the EA-calc includes multi-level indirect words.

   Because many indirect words are in the ACs, the IBOX must have the ability to fetch the indirect word from either memory or the ACs.

   Note that the IBOX may find an illegal indirect word (bit 0=bit1=1 if fetched from a non-zero section) at some point in the EA-calc. If it does, the IBOX should terminate the EA-calc and provide some sort of indication to the EBOX so that it can start a page fail trap to the monitor.

3. Instruction operand fetch and/or writability test. Based on the instruction opcode, the IBOX should fetch and/or write-test one or two operands at E or E+1.

4. Instruction setup for complex instructions. For a subset of performance critical instructions, the IBOX should perform more extensive setup for EBOX or FPA execution. This is discussed in more detail below.

5. Requests for service from the EBOX. The IBOX must service a limited number of EBOX requests (previously called ICMDs). The most obvious request is one to load a new PC. This is also discussed in more detail below.

3.2 Operand setup for simple instructions

As mentioned in the section on the instruction set, the vast majority of instructions can be divided into five classes based on the operand requirements of the instruction. The following list indicates what the IBOX must do for each class:

1. Immediate instructions. Since no operands are required, the IBOX need only provide the effective address. Note that the effective address is always 31 bits of information; a 30-bit effective address (including default section number, if

necessary) and a 1-bit local/global flag.

2. Single memory operand instructions. For this class, the IBOX must provide the effective address and the memory operand from C(EA). Depending on the opcode, the operand fetch may include an optional write-test.

3. Single memory store instructions. For this class, the IBOX must provide the effective address and write-test the memory location at EA.

4. Double memory operand instructions. For this class, the IBOX must provide EA and EA+1, and fetch the two memory operand from C(EA) and C(EA+1).

5. Double memory store instructions. for this class, the IBOX must provide EA and EA+1, and write-test the memory locations at EA and EA+1.

Note that in all cases, EA+1 is computed following the rules of extended addressing. That is, if the original EA is local (according to the local/global flag), the result is also local and the addition wraps in-section. As with EA, the EA+1 result is also a 31-bit quantity.

For all instructions, the EBOX must have access to the PC of the instruction. This is a global requirement that is independent of the instruction being executed.

3.3 Operand setup for complex instructions

In the section on the instruction set above, a number of instructions were classified as "complex". In general, this group of instructions does not fit into one of the five categories of "simple" instructions from the point of view of IBOX prefetch.

In most cases, the instructions in this class are also critical to the performance of the machine. For this reason, special handling of these instructions by the IBOX is warrented. The discussion that follows treats each instruction independently.

3.3.1 LUUO

The EBOX processing of LUUOs differs depending on whether PC is in section zero or a non-zero section. From the standpoint of IBOX setup, however, the requirements are similar. The IBOX must provide EA and the opcode and AC field of the instruction so that they may be stored in the LUUO block. The preferred format for the opcode and AC field is BYTE (18)0(9)opcode(4)AC(5)0, but this is not critical.

Note that a section zero LUUO is processed by XCTing the instruction in location 41 of the address space in which the instruction was executed. This means that the EBOX must be able to request that the IBOX perform an XCT of an instruction at a specified location as the next instruction.

3.3.2 MUUO

For MUUOs, the IBOX must provide EA and the opcode and AC field of the instruction. The prefered format is the same as for LUUO.

3.3.3 EXTEND

With the exception of a few pieces of necessary information, the EXTEND instruction simply addresses another instruction. The necessary information from the EXTEND instruction itself is the opcode (for initial EBOX microcode address), the AC field of the instruction (the block of ACs is addressed by the AC field of the EXTEND and not the AC field of the EXTENDed instruction), and EA of the EXTEND.

For EXTEND, the IBOX should compute EA and fetch the EXTENDed instruction word. The opcode of the EXTENDed instruction word should be used to address a second early-decode array that is exactly analogous to the one used to provide information about the un-EXTENDed instruction set. The IBOX then uses this information to process the instruction in a manner similar to that done for a normal instruction.

Note that the EXTENDed opcode must also be latched to be used as an EBOX dispatch. To avoid using up 512 microinstructions as the destination of the EXTENDed opcode dispatch, there should be a valid bit in the early-decode array that indicates whether the opcode is assigned or not. If it is not, the EXTENDed opcode should be forced to a single value (possibly 0 since that value is already illegal).

The EBOX processing of the EXTEND instruction would then consist of executing one EFAST and one ESLOW microinstruction from location 123. The first ESLOW location would dispatch on the latched EXTENDed opcode directly to the processing routine in a (potentially) 1-of-512 dispatch.

The list of EXTENDed opcodes can be broken down into three classes: G-floating conversion, string, and XBLT. The IBOX processing for each of these classes is covered separately below.

3.3.3.1 G-floating conversion

All EXTENDed instructions in this class are either single or double memory operand instructions. In addition, they are also executed by the FPA. As a result, the IBOX should compute EA of the EXTENDed instruction and fetch one or two memory operands from EA and EA+1.

3.3.3.2 String

The string instructions are a mixture of complex operations performed by the EBOX. For these instructions, the IBOX should compute EA of the EXTENDed instruction and provide that, along with EA of the EXTEND instruction to the EBOX for execution. Because the string instructions have the potential for storing to multiple locations, the IBOX should stop prefetching and wait for the EBOX to complete execution of the instruction before continuing.

3.3.3.3 XBLT

XBLT is an extended case of BLT. There is nothing that the IBOX can provide to the EBOX in terms of instruction setup. See the section on BLT below for a discussion of the EBOX processing of BLT.

3.3.4 IBP

For IBP, the IBOX should compute EA and fetch the byte pointer from EA and EA+1 (if two-word global) with write-test. It provides EA and the byte pointer to the EBOX for execution.

3.3.5 ADJBP

ADJBP is an IBP with a non-zero AC field. At present, it's not clear what the IBOX should do for this instruction.

3.3.6 ILDB

For ILDB, the IBOX should compute EA and fetch the byte pointer from EA and EA+1 (if two-word global) with write-test. It then increments the Y field of the byte pointer if necessary, EA-calc's the byte pointer and fetches the byte data word. It provides EA, the byte pointer, the byte data word to the EBOX for execution.

3.3.7 LDB

For LDB, the IBOX should compute EA and fetch the byte pointer from EA and EA+1 (if two-word global). It then EA-calc's the byte pointer, and fetches the byte data word. It provides EA, the byte pointer, and the byte data word to the EBOX for execution.

3.3.8 IDPB

For IDPB, the IBOX should compute EA and fetch the byte pointer from EA and EA+1 (if two-word global) with write-test. It then increments the Y field of the byte pointer if necessary, EA-calc's the byte pointer, and fetches the byte data word with write-test. It provides EA, the byte pointer, the address of the byte data word, and the byte data word to the EBOX for execution.

3.3.9 DPB

For DPB, the IBOX should compute EA and fetch the byte pointer from EA and EA+1 (if two-word global). It then EA-calc's the byte pointer, and fetches the byte data word with write-test. It provides EA, the byte pointer, the address of the byte data word, and the byte data word to the EBOX for execution.

3.3.10 BLT

BLT is an instruction that can store to many locations. As a result, the IBOX must stop prefetching until the EBOX completes execution. Other than performing the EA-calc and presenting the result to the EBOX for execution, there is nothing that the IBOX should do to complete setup for the instruction.

During EBOX execution of the instruction, however, the IBOX plays an important role. The goal is to use EBOX/IBOX cooperation to maintain maximum word transfer rate by causing the IBOX to increment the read/write addresses for the next request while the EBOX is completing the current request. The exact coupling must be thought out in more detail.

3.3.11 XCT

There is no particular reason that XCT has to be executed by the EBOX, and the IBOX should do all of the processing for the instruction. If the instruction was executed in exec mode, the IBOX should latch the AC field bits for PXCT control of the XCTed instruction. Finally, the IBOX should compute EA for the XCT and fetch the new instruction from EA (without changing PC) and process it as if it were a normal instruction fetch.

Note that the PXCT control bits that were latched from the AC field of the XCT in exec mode must be cleared when the XCTed instruction is completed.

3.3.12 ADJSP

Because there are no memory references involved in ADJSP, the IBOX setup is limited to computing EA and setting the stack global flag as appropriate.

The stack global flag is set based on the format of the stack pointer in AC and is used for EBOX main ALU carry control and microcode dispatch. It may be more efficient to set this flag in the EBOX based on the currently addressed AC.

3.3.13 PUSHJ

For PUSHJ, the IBOX should compute EA (the jump address), set the stack global flag as appropriate, compute the new address of top-of-stack by incrementing the address portion of the stack pointer, and write-test the top-of-stack location. It provides the new address of top-of-stack to the EBOX for execution.

3.3.14 PUSH

For PUSH, the IBOX should compute EA and fetch the datum to be pushed from that location. It also sets the stack global flag as appropriate, computes the new address of top-of-stack by incrementing the address portion of the stack pointer, and write-tests the new top-of-stack location. It provides the datum to be pushed and the new address of top-of-stack to the EBOX for execution.

3.3.15 POP

For POP, the IBOX should compute EA and write-test that location. It also sets the stack global flag as appropriate, computes the address of top-of-stack from the stack pointer in AC and fetches the datum to be popped from that location. It provides the datum to be popped and EA to the EBOX for execution.

3.3.16 POPJ

For POPJ, the IBOX should set the stack global flag as appropriate, compute the address of top-of-stack from the stack pointer in AC, and fetch the datum from that location. The next instruction is taken from the location addressed by the word fetched from top-of-stack. It provides the datum from top-of-stack to the EBOX for execution.

3.3.17 PUSHM

For PUSHM, the IBOX should set the stack global flag as appropriate, compute EA, and fetch the control word from that location. It provides EA and the word fetched from EA to the EBOX for execution.

Because PUSHM may write to multiple locations, the IBOX should stop prefetching and wait for the EBOX to complete execution of the instruction before continuing.

3.3.18 POPM

For POPM, the IBOX should set the stack global flag as appropriate and compute EA. It provides EA to the EBOX for execution.

Because POPM may write to multiple ACs, the IBOX should stop prefetching and wait for the EBOX to complete execution of the instruction before continuing.

3.3.19 PUSHI

For PUSHI, the IBOX should compute EA and set the stack global flag as appropriate. It also computes the new address of top-of-stack by incrementing the address portion of the stack pointer, and write-tests the new top-of-stack location. It provides EA and the new address of top-of-stack to the EBOX for execution.

3.3.20 JRST

The AC field of the JRST instruction is used as a 1-of-16 decode to determine how to process the instruction. The most common use of JRST is the JRST 0, case which is an unconditional jump to EA. All others are more complex.

The IBOX should special-case the JRST 0, case and simply fetch the next instruction from C(EA). Since this case affects nothing other than PC, it need not be executed by the EBOX at all. Because the operation of the other JRST decodes are so varied, the IBOX should probably stop prefetching and wait for the EBOX to complete execution of the instruction if something other than JRST 0, is seen.

3.3.21 JSR

For JSR, the IBOX should compute EA and write-test that location. It should treat the instruction as an unconditional jump that jumps to EA+1. It provides EA to the EBOX for execution.

3.3.22 JSP

For JSP, the IBOX should compute EA and provide that to the EBOX for execution. The next instruction will be taken from that location.

3.3.23 JSA and JRA

Because the interpretation of the effective address of these instructions is non-standard, the IBOX should simply compute EA and provide it to the EBOX for execution. The IBOX should then stop prefetching and wait for the EBOX to complete execution of the instruction.

3.3.24 Privileged (I/O) instructions

The privileged I/O instructions are those instructions which are used to control functions which are both internal and external to the processor. The first four instructions in this class use the AC field of the instruction as a 1-of-16 decode to determine how to process the instruction. The function to be performed has been assigned to one of the opcodes based on their instruction setup requirements.

The rest of the privileged I/O instructions use an individual opcode for each function. These instructions are covered separately below.

3.3.24.1 APRO (opcode 700) - Complex instructions

This class of instructions includes all those which are too complex to group into one of the other classes. This complexity may be because the instruction stores to multiple locations, because it is a conditional skip, etc. For this class of instructions, the IBOX should compute EA and provide that to the EBOX for execution. It should then stop prefetching and wait for the EBOX to complete execution of the instruction.

3.3.24.2 APR1 (opcode 701) - Immediate

This class of instructions includes all those which neither fetch from, nor store to the location addressed by EA. For this class, the IBOX should compute EA and provide that to the EBOX for exectution. No instructions in this class skip or jump.

3.3.24.3 APR2 (opcode 702) - Operand fetch from EA

This class of instructions includes all those which fetch a single memory operand from EA. For this class, the IBOX should compute EA and fetch the operand from that location. It provides EA and the operand to the EBOX for execution. No instructions in this class skip or jump.

3.3.24.4 APR3 (opcode 703) - Result store to EA

This class of instructions includes all those which store a single memory result to EA. For this class, the IBOX should compute EA and write-test that location. It should provide EA to the EBOX for execution. No instructions in this class skip or jump.

3.3.24.5 APR4 (opcode ???) - Double result store to EA and EA+1

This class of instructions includes all those which store a double memory result to EA and EA+1. For this class, the IBOX should compute EA and EA+1, and write-test both locations. It should provide EA and EA+1 to the EBOX for execution. No instructions in this class jump or skip.

3.3.24.6 UMOVE

The UMOVE instruction performs the EA-calc in current context and then fetches the computed location from previous context. If the IBOX has the capability to unconditionally force a previous context memory reference, it should treat UMOVE as a MOVE that fetches the operand from previous context. In this case, the IBOX would provide EA and the operand to the EBOX for execution.

If the IBOX cannot force an unconditional previous context memory reference, it should provide EA to the EBOX for execution and the EBOX will make the operand reference.

3.3.24.7 UMOVEM

The UMOVE instruction performs the EA-calc in current context and stores the result into the location in previous context. If the IBOX has the capability to unconditionally force a previous context write-test, it should treat UMOVEM as a MOVEM that stores the result into previous context.

If the IBOX cannot force an unconditional previous context write-test, it will have to stop prefetching until the EBOX can complete execution of the instruction.

3.3.24.8 PMOVE and PMOVEM

For PMOVE and PMOVEM, the IBOX should compute EA and fetch the physical EA-calc word from C(EA). It provides EA and the physical EA-calc word to the EBOX for execution.

3.3.24.9 RNGB and RNGBW

The RNGB and RNGBW instructions are immediate instructions which may skip based on some complex events. The IBOX should compute EA and then assume that they always skip. This is a safe assumption because the non-skip case is an exception condition that rarely happens.

If the EBOX determines that the instruction will not skip, it must do a LOAD PC.

3.3.24.10 SNBSY

The SNBSY instruction is an immediate instruction which may skip based on some complex events. The IBOX should compute EA and then assume that the instruction always skips. This is probably a safe assumption because the BUSY signal on the I/O bus should be asserted for minimum time relative to the total time spent executing instructions.

If the EBOX determines that the instruction will not skip, it must do a LOAD PC.

3.3.24.11 LDPAC and STPAC

The LDPAC and STPAC instructions are complex instructions that reference multiple memory or AC locations. For these instructions, the IBOX should provide EA to the EBOX for execution and then stop prefetching until the EBOX completes execution of the instruction.

3.3.24.12 INSQHI, INSQTI, REMQHI, and REMQTI

For the queue instructions, the IBOX should compute EA and fetch the physical EA-calc word from C(EA). It provides EA and the physical EA-calc word to the EBOX for execution.

Because the queue instruction write to multiple locations, the IBOX should stop prefetching and wait for the EBOX to complete execution of the instruction.

3.4 Summary of operand setup

This section uses symbolic notation to summarize the information presented in English in the previous section. The following pseudo-registers are assumed to exist:

- o PC. This is a 30-bit register that addresses the instruction being executed by the EBOX. Note that this register is not changed as the result of XCT or EXTEND.
- o EA1 and EA2. These are two 31-bit registers that hold 30-bit addresses plus a 1-bit local/global flag. The local/global flag is set according to the extended addressing rules during an EA-calc.
- o OP2, OP3, and OP4. These are three 36-bit data registers that hold operands for the instruction.
- o INS. This is a 13-bit register that contains the opcode and AC field of the instruction.

The notation that is used should be fairly self-explanatory. ":=" is used to indicate that the entity on the right-hand-side is latched in the pseudo-register specified on the left-hand-side.

The function "MEM[xxx]" indicates that a memory reference is made to the location indicated by the argument.

The function "WT[xxx]" indicates that a write test is performed on the location specified by the argument.

Braces (e.g. {WT[xxx]}) indicate that the item contained in the braces is optional based on the exact instruction involved.

Immediate instructions:

```
PC := address of instruction
EA1 := EA-calc
```

Single memory operand instructions:

```
PC := address of instruction
EA1 := EA-calc
OP2 := MEM[EA1] {WT[EA1]}
```

Single memory store instructions:

```
PC := address of instruction
EA1 := EA-calc
WT[EA1]
```

Double memory operand instructions:

```
PC := address of instruction
EA1 := EA-calc
EA2 := EA1 + 1
OP2 := MEM[EA1]
OP3 := MEM[EA2]
```

Double memory store instructions:

```
PC := address of instruction
EA1 := EA-calc
EA2 := EA1 + 1
WT[EA1]
WT[EA2]
```

LUUO:

```
PC := address of instruction
EA1 := EA-calc
INS := opcode and AC field
```

MUUO:

```
        PC := address of instruction
        EA1 := EA-calc
        INS := opcode and AC field
```

EXTEND (G-floating conversion with single memory operand):

```
        PC := address of instruction
        EA1 := EA-calc of G-floating conversion opcode
        OP2 := MEM[EA1]
```

EXTEND (G-floating conversion with double memory operand):

```
        PC := address of instruction
        EA1 := EA-calc of G-floating conversion opcode
        EA2 := EA1 + 1
        OP2 := MEM[EA1]
        OP3 := MEM[EA2]
```

EXTEND (string):

```
        PC := address of instruction
        EA1 := EA-calc of EXTEND
        EA2 := EA-calc of EXTENDed opcode
```

EXTEND (XBLT)

```
        PC := address of instruction
        EA1 := EA-calc of EXTEND
```

IBP:

```
        PC := address of instruction
        EA1 := EA-calc
        OP2 := MEM[EA1] WT[EA1]
        {OP3 := MEM[EA1+1] WT[EA1+1]} (if two-word global)
```

ILDB:

```
        PC := address of instruction
        EA1 := EA-calc
        OP2 := MEM[EA1] WT[EA1]
        {OP3 := MEM[EA1+1] WT[EA1+1]} (if two-word global)
        EA2 := EA-calc of byte pointer
        OP4 := MEM[EA2]
```

LDB:

```
        PC := address of instruction
        EA1 := EA-calc
        OP2 := MEM[EA1]
        {OP3 := MEM[EA1+1]} (if two-word global)
        EA2 := EA-calc of byte pointer
        OP4 := MEM[EA2]
```

IDPB:

```
        PC := address of instruction
        EA1 := EA-calc
        OP2 := MEM[EA1] WT[EA1]
        {OP3 := MEM[EA1+1] WT[EA1+1]} (if two-word global)
        EA2 := EA-calc of byte pointer
        OP4 := MEM[EA2] WT[EA2]
```

DPB:

```
        PC := address of instruction
        EA1 := EA-calc
        OP2 := MEM[EA1]
        {OP3 := MEM[EA1+1]} (if two-word global)
        EA2 := EA-calc of byte pointer
        OP4 := MEM[EA2] WT[EA2]
```

PUSHJ:

```
        PC := address of instruction
        EA1 := EA-calc
        EA2 := top-of-stack + 1
        WT[EA2]
```

PUSH:

```
        PC := address of instruction
        EA1 := EA-calc
        OP2 := MEM[EA1]
        EA2 := top-of-stack + 1
        WT[EA2]
```

POP:

```
        PC := address of instruction
        EA1 := EA-calc
        WT[EA1]
        EA2 := top-of-stack
        OP3 := MEM[EA2]
```

POPJ:

```
        PC := address of instruction
        EA1 := EA-calc
        EA2 := top-of-stack
        OP3 := MEM[EA2]
```

PUSHM:

```
        PC := address of instruction
        EA1 := EA-calc
        OP2 := MEM[EA1]
```

POPM:

```
        PC := address of instruction
        EA1 := EA-calc
```

PUSHI:

```
        PC := address of instruction
        EA1 := EA-calc
        EA2 := top-of-stack + 1
        WT[EA2]
```

JSR:

```
        PC := address of instruction
        EA1 := EA-calc
        WT[EA1]
        EA2 := EA1 + 1
```