The OIS Processor Principles of Operation

REVIEW DRAFT - COMMENTS PLEASE

April 9, 1977
Version 2.0

This document describes the interior architecture of the OIS System Element Digital Processor. It includes a description of the virtual storage system, the instruction set, and the input-output facilities.

XEROX

INFORMATION TECHNOLOGY GROUP
SYSTEMS DEVELOPMENT DIVISION
3406 Hillview Ave. / Palo Alto / California 94304

Table of Contents

Input-Output

Introduction

This document describes the interior architecture of the OIS System Element Digital Processor. It includes a description of the virtual storage system, the instruction set, and the input-output facilities.

It is required that all System Element Digital Processors implemented for OIS be compatible with this architecture. This will allow common software systems to be constructed which will operate on all members of the family, as well as providing for a common input-output interface. It will also allow reimplementation of the processor to occur when it is economically advantageous.

This document does not specify an implementation for any instance of the OIS processor; It does specify those principles which must be adhered to to guarantee software compatibility at the instruction set level, and input-output compatibility at the level of the devices.

This document will be modified from time to time, as implementation of the initial instances of the OIS processor family occurs, and the Mesa language implementation is refined. We expect to stabilize both the architecture and this document in early 1977, so that final product development may proceed without significant impact from them.

Information Forms and Syntactic Conventions

Throughout this document, a number of conventions are used, which are described in this section.

Number system

Numeric quantities are expressed in decimal unless otherwise specified. The suffix B is used to indicate octal.

* is used to indicate multiplication, ** is used to indicate exponentiation:

```
5D3 = 5000 = 5*10**3
3B5 = 300000B = 3*8**5
```

For large multiples of a power of 2, K is used to designate 2**10, and M is used to designated 2**20:

```
32K = 32*2**10 = 2**15 =     32768,
1M = 1*2**20    = 2**20 = 1048576
```

Special Characters

<x> means "contents of x".

Square brackets [ ] are used to indicate indexing or to delimit the arguments of a function:

> x[3] = <x+3> means the contents of location x+3, i.e. the third element of the vector x
> hf[2] means the value returned by the function hf with argument 2.

Double commas are used to indicate the concatenation of two fields. If x is a 3-bit field and y is a 5-bit field, then x,,y is an eight-bit field with x in its high order bits.

Terms

a *word* is a sixteen bit quantity. Bit 0 is the most significant bit, bit 15 is the least significant bit. When diagrammed, bit 0 is on the left.

A *doubleword* is a thirty-two bit quantity, with bits numbered from 0 to 31. In main storage, the least significant bits (16-31) of a doubleword are stored in location n, the most significant bits (0-15) are stored in location n+1. When a doubleword appears on the evaluation stack, the most significant bits are on the top of the stack, the least significant bits are in the second position.

A *byte* is an eight bit quantity. Bit 0 is the most significant bit, bit 7 is the least significant bit. When diagrammed, bit 0 is on the left.

A *field* is a contiguous group of bits within a word or larger field. The bits are numbered from the left starting at 0. For example, the field consisting of the least significant byte of x is indicated with x[8:15]. If the field is named, the contruct p.f, where p is the address of the word containing the field and f is the field name, is sometimes used to represent the value of the field.

A *pointer* is the address (or displacement from a designated base address) of the first location of a contiguous region of virtual memory. There are a number of different formats for pointers which are described below.

A *page* is a contiguous block of 256 16-bit words, the first word of which begins at an address which equals 0 mod 256.

A *Procedure* is a body of code which performs a single function.

A *Code Segment* is a collection of procedures which are compiled together.

A *Process* is a group of operations and the data on which they operate which can (at least conceptually) execute in parallel with other processes. A process is defined by the contents of memory and by a *state block* which is loaded into the processor registers when the process is run. All processes supported by the OIS processor share a common virtual address space.

A *Main Data Space* (MDS) is a contiguous region of virtual memory associated with one or more processes. It has a maximum size of 64K words, and will be described in detail in subsequent sections. By hardware convention it always begins on a page boundary, hence its address is always 0 mod 256. MDS will be used as the abbreviation for Main Data Space, mds as the designator of the register which points to MDS, and mds pointer as defined below.

Virtual Storage

Introduction

All implementations of the OIS Processor will provide a virtual memory system (VMS) which supports a linear virtual address space of 2**24 sixteen-bit words. This will allow the development of complex software systems which are, to a large extent, configuration-independent.

The virtual memory system has several purposes:

> It provides *address translation* between virtual addresses generated by a program and real memory addresses used by the memory hardware.
>
> It provides *dynamic relocation* of information so that objects need not occupy fixed locations in main storage throughout their existence, but may be moved between secondary storage and any unoccupied area of main storage as required.
>
> It provides *protection* for areas of the address space. Although many of the protection mechanisms normally provided by hardware are provided in the OIS environment by the Mesa compiler's type checking machinery, some degree of protection in the hardware is desirable, primarily to detect errors rather than to defend against hostile action.

There are four primary components of the virtual memory system: First, there must be hardware and/or firmware to do the address translation. Second, there must be storage for the translation information. Third, there must be a secondary storage device which holds the majority of the information contained in the virtual space. Finally, there is a body of software, usually associated with the operating system, which is responsible for transferring information between main and secondary storage.

Mesa's Use of the Virtual Memory System

The OIS processor instruction set has been designed for efficient execution of the Mesa language, which will be used for all OIS programming. A primary Mesa design goal was to provide a space-efficient representation for code. As a result, a large fraction of the memory reference instructions make use of implicit or explicit base registers which point to frequently referenced structures. Thus, the amount of address information required in an instruction is small. In addition, several commonly referenced structures are constrained to begin on (256 word) page boundaries in memory, and can thus be represented by sixteen-bit pointers rather than full addresses.

Although the instruction set makes use of a number of formats for addresses, all are short forms of a full 24-bit virtual address. There is no method specified to bypass the address translation process and directly reference a real main storage location.

Address Translation

The address translation process is identical for all memory references, i.e. for instructions, operands, and I/O operations.

The CPU generates a 24-bit virtual address, usually by adding one or more offsets to an implicit or explicit base register. The generation of this effective virtual address is described later in this document for each instruction.

The 24-bit address is then passed to the address translation hardware, which will attempt to translate it into a real storage address. If the translation is successful, and if no access protection checks are violated, the reference is made and execution continues. If a protection violation occurs, the memory management software will be notified via a trap, and the offending instruction will not be executed.

Since all operations are implemented such that they can be restarted if a memory fault occurs, the memory manager can simply bring the required information into main storage and restart the offending instruction. The interface between the translation hardware/firmware and the memory management software is thus reduced to two traps, *Page Fault* and *Write Protect*, and the Mesa instructions required to manipulate the map.

All implementations of the processor must provide a virtual address space of not less than 22 bits, and there must be a mechanism (described later) to report the maximum size of the virtual space in a particular model to the software, as well as a provision for causing a *PageFault* trap on any attempt to reference locations outside the virtual space provided.

Translation between virtual addresses generated by the processor or I/O system and real addresses used by the memory is done by a *map* implemented in hardware. The map accepts a 24-bit virtual address from the requester, and delivers a real address to the storage modules. The real address is from 18 to 20 bits, depending on the amount of real storage provided in the particular model. Mapping takes place in one page (256 word) quanta, i.e. the least significant eight bits of the virtual address bypass the mapping hardware.

In addition to the information necessary for address translation, the map also contains three bits, Dirty, Write Protected, and Referenced, which provide the memory management software with information about each page. The Dirty bit is set by the mapping hardware when a store is done to a non-write protected page. The Write Protect bit prohibits stores into a particular page, and reports an error if a store is attempted. The Referenced bit is set when any access is made to a page.

Requests

The memory system is capable of accepting four types of requests: I/O fetches, I/O stores, Processor fetches, and Processor stores. Error conditions arising from these requests are reported to the requester in one of two ways. If the request came from an I/O device controller, the controller is notified that an error occurred, and the reference is not done. The controller will take whatever action is required (usually halting the data transfer), and will report the error to the processor in its next status report. If the error arose as a result of a Mesa instruction or operand reference and a trap will be generated. The trap parameter for all memory-related traps is the 16-bit virtual page number.

When a request is received by the memory system, the address is range checked (in configurations providing less than 24 bits of virtual space), then sent to the map. The following table shows the possible outcomes for a request based on its type and the original state of the flag bits.

Map flag bits:	Request Type / Result (W,D,Ref):			
(W,D,Ref)	Processor Fetch	Processor Store	I/O Fetch	I/O Store
0 0 0	0 0 1	0 1 1	0 0 1	0 1 1
0 0 1	0 0 1	0 1 1	0 0 1	0 1 1
0 1 0	0 1 1	0 1 1	0 1 1	0 1 1
0 1 1	0 1 1	0 1 1	0 1 1	0 1 1
1 0 0	1 0 1	1 0 0 (1)	1 0 1	1 0 0 (2)
1 0 1	1 0 1	1 0 1 (1)	1 0 1	1 0 1 (2)
1 1 X (3)	1 1 X (4)	1 1 X (4)	1 1 X (5)	1 1 X (5)

(1) Inhibit the store. Cause the *WriteProtect* trap.
(2) Inhibit the store. Return violation to the I/O controller, which reports it in its status.
(3) This state means *vacant*, i.e. the requested virtual page is not in real memory.
(4) Page is not in real memory, cause *PageFault* trap.
(5) Page is not in real memory. Return violation to I/O controller, which reports it in its status.

Operations on the Map

The processor will provide two operations for dealing with the map (as Mesa instructions). In what follows, v is a 16-bit virtual page number, r is a 12-bit real page number, and f is the 3-bit flag value:

> Associate[r,v,f] Makes a correspondence in the map between real page r and virtual page v, and sets the flag bits in the map entry to f.
>
> SetFlags(v,f) Sets the flag bits associated with the map entry for virtual page v to f, and returns the old value of the flags. If the entry for v is not in the map or is not currently associated with a real page, the operation returns *vacant* (old flags = 6). The operation consisting of reading the old flags and setting the new value must be indivisible.

The software must not attempt to map two different pages in the virtual space into the same real page {restriction imposed by associative implementations}.

Mapping Examples

To bring virtual page v into main storage, the software will do:

Obtain a free real page, say page r.

Associate[m,r,0] -- Map r into a virtual page m known only to the memory manager. This 'hides' the page from all software other than the memory manager during the time the page is being read in from the disk.

Read the page from the disk into m.

Associate[m,r,vacant] -- Remove the page from m.

Associate[v,r,new flags] --Make the page available to the requester.

To remove a virtual page v from real page r, the software will do:

OldFlags ← SetFlags[v,r,WriteProtected]

if OldFlags.dirty then WritePage[...] -- If the page was dirty, write it to the disk. Since the page is now write protected, no stores into the page are possible during the write.

SetFlags[v,vacant] -- Release the page

The software will require one or more auxiliary tables which contain information about the allocation and state of virtual and real storage. These structures need not be known to the hardware in any manner.

Pointer Formats

All addresses generated by the processor are 24-bit virtual addresses. These addresses are stored in a variety of ways, and specific terms are used to describe each format. This section describes the various pointer formats and the terms used for them. This information is also shown schematically in figure 1.

A *24-bit pointer* is a doubleword which contains the 24-bit virtual address in bits 8-31. Bits 0-7 are zero.

A *page pointer* is the most significant sixteen bits of a 24-bit virtual address. Page pointers are stored in a single word.

The OIS processor supports a number of processes, each of which has associated with it a *Main Data Space.* The MDS is a 64K word (maximum) region of the virtual space which is pointed to by the register mds. Note that although each process has only one MDS, a number of processes may share a particular MDS. Since an MDS is constrained to begin on a page boundary by hardware convention, mds contains a page pointer. Since many operations make use of 16-bit displacements which are added to mds to form a full 24-bit virtual address, we will make use of the term *mds pointer* to describe such a displacement. There is nothing unique about an mds pointer - the term is used solely for brevity to indicate that the pointer is a 16-bit displacement relative to the page pointer content of mds.

A *32-bit pointer* is a doubleword containing a page pointer in bits 0-15, and a word displacement relative to the start of the page in bits 16-31.

The term *long pointer* is used when an operation will accept either a 24-bit or a 32-bit pointer. To allow the pointer type to be determined from its value, the convention is used that no object which may be described by a 32-bit pointer will be placed in the first 64K of the virtual space. Thus, if bits 0-7 of a long pointer are zero, the pointer is a 24-bit pointer, otherwise it is a 32-bit pointer.

Central Processor

Data Structures and Associated Registers

The Mesa language makes use of a number of structures with defined formats which are known both to the control transfer instructions (hardware) and to the compiler. The location of a number of these structures relative to the beginning of MDS or relative to the beginning of virtual memory are given by constants whose values are given in Appendix A (in most cases, the precise values of these constants have not been determined at present. Where this is the case, approximate values are given and the approximation is indicated). These structures are shown in figure 2, and include:

Code Segments (C Register)

A code segment contains the instructions for a group of procedures which were compiled as a unit (a module), plus an entry vector which contains the information necessary to find the code associated with each procedure in the module and to allocate a local frame of the appropriate size for the procedure. The register C contains a 24-bit pointer to the base of the currently active code segment.

In most cases, the information required to find the code associated with a procedure (the entry vector item) occupies a single word in the entry vector:

bit 0: 0
bits 1:4: Frame size index
bits 5:15: C-relative byte pointer to the code for this entry

In this case, the frame size index must be such that it can be represented in four bits, the code must lie within 2048 bytes of the base of the code segment, and the procedure must not have any defaulted parameters. If this is not the case, the entry vector item has a different format:

bit 0: 1
bits 1:15: C-relative word pointer to the code for this entry

For this format, the code must lie within 32K words of the base of the code segment, and the code must start on an even byte. The remaining information required to run the code is contained in the word which precedes the code itself:

bits 0:3: Information for defaulting parameters (mechanism unspecified as yet).
bits 4:15: Frame size index. This number is either the frame size index or the size of the frame (in words) if fsi>MaxAllocSlot-1 (see "Frame Allocation")

Since the code pointer (the initial PC) is a byte pointer, and is held in a 16-bit word the maximum size of a code segment is 2**16 = 65536 bytes or 32768 words. Further, since the code segment is pointed to by a 24-bit pointer it is generally (by software convention) disjoint from all Main Data Spaces. The maximum size of a frame is limited by this mechanism to 4096 words.

By software convention, code segments are read-only, and are modified only by the Mesa debugger's breakpoint machinery.

Global frames (G Register)

A global frame is a designated area in MDS. It contains a 24-bit pointer to a code segment, and contains all global variables and external linkage information required by an instance of that code segment. The register G contains an mds pointer to the currently active global frame.

Global frames are created each time that an instance of a module is required. This may occur dynamically at runtime, but usually will be done only when a number of modules are *bound* into a functional configuration. There may thus be more than one global frame per code segment.

Local Frames (L Register)

A local frame is a designated area in MDS. It contains all the local state for a procedure. It is created when a procedure is called, and destroyed (usually) when the procedure returns control to its caller. The register L contains an mds pointer to the currently active local frame. The local frame contains all the information required to continue execution of a procedure whose execution was suspended (when, for example, it calls another procedure).

System Dispatch (sd)

The system dispatch table occupies the same designated area in every MDS. The constant sd specifies the offset (in all main data spaces) of the system dispatch table. The system dispatch table contains *control links* for commonly used runtime procedures, and is used only by the KFCB instruction and by traps.

Global Frame Table (gft)

The global frame table occupies the same designated area in every MDS. The constant gft specifies the offset (in all main data spaces) of the global frame table. Each entry in the global frame table is an mds pointer to a global frame (G). The global frame table is accessed by using the GFT index portion of a standard procedure descriptor (see "Control Links").

Allocation Vector (av)

The allocation vector occupies the same designated area in every MDS. The constant av specifies the offset (in all main data spaces) of the allocation vector. The allocation vector is used primarily for dynamic allocation of local frames. A pool of frames of the most frequently used sizes is maintained by the software. This pool is accessed via the allocation vector av, each entry of which is the head of a list of frames of a fixed size. The frame size index in the entry vector of a code segment provides an index into the allocation vector which is used to locate a frame of the required size when a procedure is entered. The frame itself contains this index as well, so that it can be returned to the appropriate list when the procedure returns. There is a mechanism for indirection which allows the last frame in a list to point to the list for some larger frame size (see "Frame Allocation"). An attempt to allocate a frame from a totally empty list results in a trap.

The Processor Stack

Many of the roles normally filled by central registers or accumulators in some machines are filled in the OIS processor by the *processor stack.* This stack is an array of sixteen-bit registers accessed indirectly via a pointer register stkp.

The precise number of registers in the stack has not yet been detemined. In this document, the parameter stkmax is used to designate this value.

The registers comprising the stack are designated stk[1] through stk[stkmax]. The instruction set makes use of two prmitive operations, *push* and *pop*, which write and read 16-bit words to the stack register addressed by stkp. In the instruction descriptions which follow, push[x] means:

```
stkp ← stkp+1
stk[stkp] ← x
```

Pop[x] means:

```
x ← stk[stkp]
stkp ← stkp-1
```

The stack pointer points to the highest numbered occupied stack location (the 'top of stack'). The stack is empty if stkp=0, full if stkp = stkmax+1. Although the Mesa compiler normally keeps track of the depth of the stack and will not compile operations which underflow or overflow it, a trap is provided by the hardware to detect an attempt to cause underflow (pop when stkp=0) or overflow (push when stkp=stkmax+1). If this is attempted, the trap *StackError* is generated, and the stack pointer is not modified.

In addition to the push and pop operations, some instructions need to be able to modify the stack pointer without affecting the values on the stack, others need to be able to address locations in the stack relative to the stack pointer {note that implementation constraints are likely to preclude the latter capability. The capability is a logical requirement, and need not be implemented in precisely this way}.

The stack is used for expression evaluation and for passing arguments to procedures. The load instructions push words from memory onto the stack, the store instructions pop the stack into memory. The conditional jump instructions pop the top one or two items from stack, test them in various ways, and branch based on the result of the test. The arithmetic operations pop their operands from the stack, calculate a result, and push it onto the stack.

Some operations leave infrequently used results 'above the stack', i.e. in stack locations beyond the one pointed to by stkp. A Mesa instruction (Push) is provided to recover these quantities, if required, by incrementing stkp. Another instruction (Pop) is provided to discard the top element of the stack by decrementing stkp.

The Machine State

The OIS processor supports a total of sixteen hardware-scheduled Mesa processes. The structure of the process switching system which controls the selection of the currently active process is desribed in a later section of this document. Each of these hardware-scheduled processes has a main data space of up to 64K words which contains the local storage for the process, as well as the tables described earlier. The state for a hardware-scheduled process is accessed via an entry in the 16 word *Process State Vector.* The process state vector must begin on a page boundary. The constant psv is a page pointer to the process state vector. Each entry in the psv is the (16-bit) psv-relative displacement of a state block containing the state of the process. Each of the state blocks contains the following quantities, which are sufficient to completely specify the process:

stkp: The evaluation stack pointer (right justified in the word)

stk[1], stk[2], ..., stk[stkmax]: The stack itself

dest: A control link (usually a local frame pointer), which is used by the processor to obtain all the portions of the process state which are kept in machine registers while the process is running. The quantities which are located, either directly or indirectly, from dest are:

G: Pointer to the current global frame (an mds pointer)

C: Pointer to the current code segment (a 24-bit pointer)

PC: The program counter (a byte displacement relative to C)

L: A pointer to the current local frame (an mds pointer)

The precise manner in which these quantities are located from dest is described under "Process Switching".

mds: A page pointer to the main data space of the process

Instruction Formats and Classes

The Mesa instruction set is divided into four principal classes:

Loads and stores
Data modification instructions
Jumps
Control transfers

Instructions are from one to three bytes in length; the opcode is always the first byte. The second and third bytes, if used, are designated α and β respectively. In situations in which both α and β are used as a 16-bit quantity, the designation $\alpha\beta$ is used.

Currently, the instruction set contains more than 256 opcodes. The intent is to reserve a single opcode for an OPERATE instruction, and encode a number of infrequently used opcodes into the α byte of this instruction. This will yield a total of 511 opcodes, not all of which will be used. If the processor attempts to execute an unimplemented opcode, the trap *UnimplementedInstruction* is generated.

In the description of the instructions, the format used is:

Instruction Name
Mnemonic (length in bytes):
description of the instruction's effects.

L,G,and C refer to the *values* of the L,G, and C registers.

The octal opcodes of all instructions are summarized in Appendix A.

The pseudo-language used to describe the effects of instructions is provided for precision, and is not intended to suggest actual implementation, although the sequence in which the atomic operations which comprise instructions are executed is often important. A number of temporary values (e.g. *temp,pointer,data*) are used in the descriptions.

Load/Store Instructions

These instructions transfer data between the evaluation stack and one or more locations in main storage. Instructions are provided for accessing partial words, full words, and doublewords.

The final effective address of all load/store instructions is a 24-bit virtual address. It is the responsibility of the virtual storage system to translate this virtual address into a real address, verify that the location referenced is present in main storage, and apply the appropriate protection checks. These translation operations are *not* described in detail in the descriptions of the instructions.

Many of the instructions are optimized to access the main data space using a 16-bit mds pointer rather than a full 24-bit address. This is indicated explicitly in the instruction descriptions by including mds in the effective address calculation. When used in this way, mds is to be interpreted as a 24-bit value consisting of the page pointer followed by eight zeroes.

When an instruction makes use of a doubleword pointer, the value is treated as a *long pointer*, i.e. either of the formats for specifying a 24-bit address in a doubleword may be used. It is the responsibility of the hardware to interpret this pointer properly, as described under "Pointer Formats". In the instruction descriptions, long pointers are designated "lpointer" (= lpointerh,,lpointerl).

Load/Store Global Word

The load global instructions read a word from the global frame and push it onto the stack. The compiler sorts the references to global variables in a module by frequency, and assigns the eight most frequently referenced variables to the first eight globals in the frame. These variables are accessed using single byte instructions. The remaining globals are referenced using a two-byte instruction in which α indicates the offset into the frame, or with a single byte instruction which uses the top element of the stack as the offset. Note that the first global variable is in G[globalbase] (globalbase is a small constant, see Appendix A), and the single byte loads include this offset. The instructions which use α as the displacement do so relative to G.

Load Global n, n=0-7
LGn (1):
push[<mds+G+n+globalbase>]

Load Global Byte
LGB (2):
push[<mds+G+α>]

The store global instructions store the top element of the stack into the global frame. They take the displacement from the opcode, or from α.

Store Global n, n=0-3
SGn (1):
pop[mds+G+n+globalbase]

Store Global Byte
SGB (2):
pop[mds+G+α]

Load/Store Local Word

The Load Local instructions read a word from the local frame and push it onto the stack. The compiler sorts the references to local variables in a procedure by frequency, and assigns the eight most frequently referenced variables to the first eight locals in the frame. These variables are accessed using single byte instructions. The remaining locals are referenced using a two-byte instruction in which α indicates the offset into the frame. Note that the first local variable is in L[localbase] (localbase is a small constant, see Appendix A), and the single byte loads include this offset. The instructions which use α as the displacement do so relative to L.

Load Local n, n=0-7
LLn (1):

```
push[<mds+L+n+localbase>]
```

Load Local Byte
LLB (2):

```
push[<mds+L+α>]
```

Load Local 0 and 0
LL00 (1):

```
push[<mds+L+localbase+0>]
push[0]
```

Load Local 1 and 0
LL10 (1):

```
push[<mds+L+localbase+1>]
push[0]
```

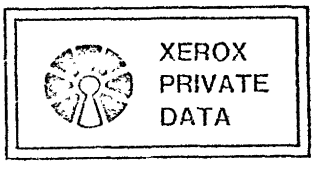
The Store Local instructions store the top element of the stack into the local frame. They take the displacement from the opcode or from α.

Store Local n, n=0-7
SLn (1):

```
pop[mds+L+n+localbase]
```

Store Local Byte
SLB (2):

```
pop[mds+L+α]
```



This instruction is identical to the Store Local instructions except that the stack pointer is not decremented, leaving the stored value on top of the stack.

Put Local n, n=0-3
PLn (1):

```
pop[mds+L+n+localbase]
stkp ← stkp + 1
```

Load Immediate

These instructions push constants onto the stack.

Load Immediate n, n=0-10
LIn (1):

```
push[n]
```

Load Immediate Negative One
LIN1 (1):

```
push[-1]
```

Load Immediate Byte
LIB (2):

```
push[α]
```

Load Immediate Word
LIW (3):

```
push[αβ]
```

Load/Store Global Doubleword

Load Double Global Byte
LDGB (2):

push[<mds+G+α>]
push[<mds+G+α+1>]

Store Double Global Byte
SDGB (2):

pop[mds+G+α+1]
pop[mds+G+α]

Load/Store Local Doubleword

Load Double Local 0
LDL0 (1):

push[<mds+L+localbase+0>]
push[<mds+L+localbase+1>]

Load Double Local Byte
LDLB (2):

push[<mds+L+α>]
push[<mds+L+α+1>]

Load Double Local Swapped 0
LDLS0 (1):

push[<mds+L+localbase+1>]
push[<mds+L+localbase+0>]

Store Double Local 0
SDL0 (1):

pop[mds+L+localbase+1]
pop[mds+L+localbase+0]

Store Double Local Byte
SDLB (2):

pop[mds+L+α+1]
pop[mds+L+α]

Read/Write Word

These instructions use the top element of the stack as a pointer, add to it a displacement from the opcode or α, and do a push or pop.

Read n, n=0-4
Rn (1):

```
pop[pointer]
push[<mds+n+pointer>]
```

Read Byte
RB (2):

```
pop[pointer]
push[<mds+α+pointer>]
```

Read Byte and Load Local 0
RBLL0 (2):

```
pop[pointer]
push[<mds+pointer+α]
push[<mds+L+localbase+0>]
```

Write n, n=0-2
Wn (1):

```
pop[pointer]
pop[mds+n+pointer]
```

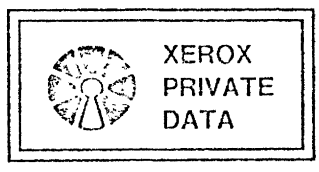
Write Byte
WB (2):

```
pop[pointer]
pop[mds+α+pointer]
```

The following two instructions are similar to Wn and WB, except that the order of their operands on the stack is reversed so that the pointer may be recovered with a *Push* instruction:

Write Swapped 0
WS0 (1):

```
pop[data]
pop[pointer]
<mds+pointer> ← data
```



Write Swapped Byte
WSB (2):

```
pop[data]
pop[pointer]
<mds+pointer+α> ← data
```

The following two instructions are similar to Wn and WB, except that the order of their operands on the stack is reversed and the pointer is left on the stack for a subsequent instruction:

Put Swapped 0
PS0 (1):

```
pop[data]
pop[pointer]
<mds+pointer> ← data
stkp ← stkp + 1
```

Put Swapped Byte
PSB (2):

```
pop[data]
pop[pointer]
<mds+pointer+α> ← data
stkp ← stkp + 1
```

The following instructions interpret the top two elements of the stack as a long pointer, add α to it, and do a push or pop.

Read Byte Long
RBL (2):

```
pop[lpointerh]
pop[lpointerl]
push[<α+lpointer>]
```

Write Byte Long
WBL (2):

```
pop[lpointerh]
pop[lpointerl]
pop[α+lpointer]
```

Read/Write Doubleword

These instructions take a pointer from the stack and do a doubleword push or pop. The *byte* versions use α as a displacement relative to the pointer:

Read Double 0
RD0 (1):

```
pop[pointer]
push[<mds+pointer>]
push[<mds+pointer+1>]
```

Read Double Byte
RDB (2):

```
pop[pointer]
push[<mds+pointer+α>]
push[<mds+pointer+α+1>]
```

Write Double 0
WD0 (1):

```
pop[pointer]
pop[mds+pointer+1]
pop[mds+pointer]
```

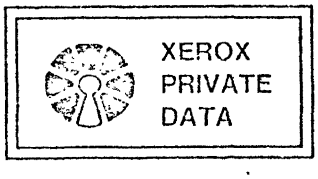
Write Double Byte
WDB (2):

```
pop[pointer]
pop[mds+pointer+α+1]
pop[mds+pointer+α]
```

PSD0 and PSDB take their operands from the stack in reverse order and leave the pointer on the stack for a subsequent instruction:

Put Swapped Double 0
PSD0 (1):

```
pop[data1]
pop[data2]
pop[pointer]
<mds+pointer+1> ← data1
<mds+pointer> ← data2
stkp ← stkp + 1
```



Put Swapped Double Byte
PSDB (2):

```
pop[data1]
pop[data2]
pop[pointer]
<mds+pointer+α+1> ← data1
<mds+pointer+α> ← data2
stkp ← stkp + 1
```

WSD0 and WSDB take their operands from the stack in reverse order, so that the pointer may be recovered by a *Push* instruction:

Write Swapped Double 0
WSD0 (1):

```
pop[data1]
pop[data2]
pop[pointer]
<mds+pointer+1> ← data1
<mds+pointer> ← data2
```

Write Swapped Double Byte
WSDB (2):

```
pop[data1]
pop[data2]
pop[pointer]
<mds+pointer+α+1> ← data1
<mds+pointer+α> ← data2
```

The following instructions interpret the top two elements of the stack as a long pointer, and do a doubleword read or write.

Read Double Byte Long
RDBL (2):

```
pop[lpointerh]
pop[lpointerl]
push[<lpointer+α>]
push[<lpointer+α+1>]
```

Write Double Byte Long
WDBL (2):

```
pop[lpointerh]
pop[lpointerl]
pop[lpointer+α+1]
pop[lpointer+α]
```

Read/Write Indexed

These instructions consider α as a pair of numbers encoded in four bit fields. The displacement in the first field and the item from the top of stack are added to the local selected by the second field and a push is performed at that location.

Read Indexed by Local Pair
RXLP (2):

```
pop[index]
pointer ← <mds+L+localbase+α[0:3]>
push[<mds+pointer+index+α[4:7]>]
```

Write Indexed by Local Pair
WXLP (2):

```
pop[index]
pointer ← <mds+L+localbase+α[0:3]>
pop[mds+pointer+index+α[4:7]]
```

Read Indexed by Local Pair Long
RXLPL (2):

```
pop[index]
lpointerl ← <mds+L+localbase+α[0:3]>
lpointerh ← <mds+L+localbase+α[0:3]+1>
push[<lpointer+index+α[4:7]>]
```

Write Indexed by Local Pair Long
WXLPL (2):

```
pop[index]
lpointerl ← <mds+L+localbase+α[0:3]>
lpointerh ← <mds+L+localbase+α[0:3]+1>
pop[lpointer+index+α[4:7]]
```

Read/Write Indirect

These instructions add a displacement from the opcode to a local variable, and do a push or pop to that location.

Read Indirect Local n, n= 0-3
RILn (1):

push[<mds+<mds+L+localbase>+n>]

This instruction is similar to RILn, except that the pointer is in the local specified in the first 4 bits of α, and the offset is taken from second four bit field of α.

Read Indirect Local Pair
RILP (2):

push[<mds+<mds+L+localbase+α[0:3]>+α[4:7]>]

Writing version of RILP.

Write Indirect Local Pair
WILP (2):

pop[<mds+L+localbase+α[0:3]>+α[4:7]]

This instruction is similar to RILP, except that the pointer is taken from the global frame.

Read Indirect Global Pair
RIGP (2):

push[<mds+<mds+G+globalbase+α[0:3]>+α[4:7]>]

Read Indirect Local Pair Long
RILPL (2):

lpointerl ← <mds+L+localbase+α[0:3]>
lpointerh ← <mds+L+localbase+α[0:3]+1>
push[<lpointer+α[4:7]>]

Read Indirect Global Pair Long
RIGPL (2):

lpointerl ← <mds+G+globalbase+α[0:3]>
lpointerh ← <mds+G+globalbase+α[0:3]+1>
push[<lpointer+α[4:7]>]

Write Indirect Local Pair Long
WILPL (2):

```
lpointerl ← <mds+L+localbase+α[0:3]>
lpointerh ← <mds+L+localbase+α[0:3]+1>
pop[lpointer+α[4:7]]
```

Write Indirect Global Pair Long
WIGPL (2):

```
lpointerl ← <mds+G+globalbase+α[0:3]>
lpointerh ← <mds+G+globalbase+α[0:3]+1>
pop[lpointer+α[4:7]]
```

These instructions optimize double indirection.

Read Indirect Indirect Local 0 Pair
RIILP (2):

```
pointer ← <mds+L+localbase>
push[<mds+<mds+pointer+α[0:3]>+α[4:7]>]
```

Read Indirect Indirect Pair
RIIP (2):

```
pop[pointer]
push[<mds+<mds+pointer+α[0:3]>+α[4:7]>]
```

Read/Write String

These operations take a byte index into a string from the top element of the stack, a pointer to the string from the second element, and read or write a single character (byte) from the string. There are also versions which use the second and third elements of the stack as a long pointer. A Mesa string has the format shown in figure 3 (the instructions do not make use of the first two words of the string).

Read String
RSTR (1):

```
pop[index]
pop[pointer]
if index odd do --odd index means right byte
        push[<mds+pointer+2+index/2> and 377B]
else do
        push[(<mds+pointer+2+index/2> and 177400B) rshift 8]
```

Read String Long
RSTRL (1):

```
pop[index]
pop[lpointerh]
pop[lpointerl]
if index odd do --odd index means right byte
        push[<lpointer+2+index/2> and 377B]
else do
         push[(<lpointer+2+index/2> and 177400B) rshift 8]
```

Write String
WSTR (1):

```
pop[index]
pop[pointer]
pop[data]
if index odd do
        <mds+pointer+2+index/2>←<mds+pointer+2+index/2>
        and 177400B) or (data and 377B) --odd index means right byte
else do
        <mds+pointer+2+index/2>←<mds+pointer+2+index/2> and 377B) or (data and
        177400B) --even index means left byte
```

Write String Long
WSTRL (1):

```
pop[index]
pop[lpointerh]
pop[lpointerl]
pop[data]
if index odd do
        <lpointer+2+index/2>←<lpointer+2+index/2>
        and 177400B) or (data and 377B) --odd index means right byte
else do
        <lpointer+2+index/2>←<lpointer+2+index/2>
        and 377B) or (data and 177400B) --even index means left byte
```

Read/Write Field

These instructions use the top element of the stack plus α as a displacement, and push or pop the field described by β. There are also versions which use the top two elements of the stack as a long pointer.

A field descriptor is an eight-bit byte. The left four bits give the position of the field in a word, the right four bits indicate its size. If *pos,,size* is a field descriptor, the first bit of the field it describes is bit (16-*pos*-*size*), the last bit is (15-*pos*), i.e. *pos* indicates the amount by which a word must be right-shifted to extract the field, and *size* indicates the width of the mask which must be applied to the word. Operations which read a field leave it right justified on the stack; Operations which write a field store the rightmost *size* bits in the correct position in the word, and leave the remaining bits unchanged.

Read Field
RF (3):

```
pop[pointer]
temp ← <mds+pointer+α>
mask ← 2**β[4:7]-1
push[(temp rshift β[0:3]) and mask]
```

Read Field Long
RFL (3):

```
pop[lpointerh]
pop[lpointerl]
temp ← <lpointer+α>
mask ← 2**β[4:7]-1
push[(temp rshift β[0:3]) and mask]
```

This instruction is similar to RF, except that the pointer is in local zero, and the field size and field position are taken from two four bit fields of α.

Read Indirect Local 0 Field
RILF (2):

```
pointer ← <mds+L+localbase>
temp ← <mds+pointer>
mask ← 2**α[4:7]-1
push[(temp rshift α[0:3]) and mask]
```

This instruction is a combination of RILP and RF.

Read Indirect Local Pair Field
RILPF (3):

```
pointer ← <mds+L+localbase+α[0:3]>
temp ← <mds+pointer+α[4:7]>
mask ← 2**β[4:7]-1
push[(temp rshift β[0:3]) and mask]
```

Write Field
WF (3):

```
pop[pointer]
pop[data]
temp ← <mds+pointer+α>
mask ← (2**β[4:7]-1) lshift β[0:3]
data ← data lshift β[0:3]
<mds+pointer+α> ← (temp and not mask) or (data and mask)
```

Write Field Long
WFL (3):

```
pop[lpointerh]
pop[lpointerl]
pop[data]
temp ← <lpointer+α>
mask ← (2**β[4:7]-1) lshift β[0:3]
data ← data lshift β[0:3]
<lpointer+α> ← (temp and not mask) or (data and mask)
```

The Write Swapped Field instruction is similar to WF, except that the operands are reversed on the stack so that the pointer may be recovered by a *Push* instruction:

Write Swapped Field
WSF (3):

```
pop[data]
pop[pointer]
temp ← <mds+pointer+α>
mask ← (2**β[4:7]-1) lshift β[0:3]
data ← data lshift β[0:3]
<mds+pointer+α> ← (temp and not mask) or (data and mask)
```

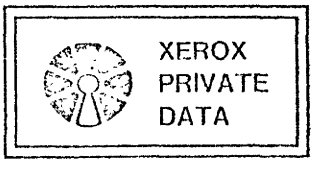
The Put Swapped Field instruction is similar to WF, except that the operands are reversed on the stack and the pointer is left on the stack for a subsequent instruction:

Put Swapped Field
PSF (3):

```
pop[data]
pop[pointer]
temp ← <mds+pointer+α>
mask ← (2**β[4:7]-1) lshift β[0:3]
data ← data lshift β[0:3]
<mds+pointer+α> ← (temp and not mask) or (data and mask)
stkp ← stkp + 1
```

Read Field 0
RF0 (2):

```
pop[pointer]
temp ← <mds+pointer>
mask ← 2**α[4:7]-1
push[(temp rshift α[0:3]) and mask]
```



The Read Bit instructions are similar to RF, except that the size portion of the field descriptor is implicitly 1, and the offset and field position are taken from two four bit fields of α. The pointer may be either on the stack or in local 0.

Read Indirect Local 0 Bit
RILBIT (2):

```
pointer ← <mds+L+localbase>
temp ← <mds+pointer+α[0:3]>
push[(temp rshift α[4:7]) and 1]
```

Read Bit
RBIT (2):

```
pop[pointer]
temp ← <mds+pointer+α[0:3]>
push[(temp rshift α[4:7]) and 1]
```

Data Modification Instructions

These instructions pop their operands from the stack, perform an operation, and push the result.

Add
ADD (1):

```
pop[x]
pop[y]
push[x+y]
```

The top two elements of the stack taken as two's complement numbers are added. The result is pushed onto the stack.

Subtract
SUB (1):

```
pop[x]
pop[y]
push[y-x]
```

The top element of the stack is subtracted from second element using two's complement arithmetic. The result is pushed onto the stack.

Double Add
DADD (1):

```
pop[x]
pop[y]
pop[t]
pop[u]
push[u+y] (c1←carry)
push[t+x+c1] (c2←carry)
push[c2]
stkp← stkp-1
```

The two doublewords on the stack are added and pushed. The carry resulting from the 32-bit addition is left above the top of the stack (in bit 15), so that it may be recovered by a *Push* instruction if required.

Double Subtract
DSUB (1):

```
pop[x]
pop[y]
pop[t]
pop[u]
push[u-y] (c1←carry)
push[t-x-c1] (c2←carry)
push[c2]
stkp← stkp-1
```

The doubleword on the top of the stack is subtracted from the doubleword in the second stack position, and the result is pushed. The carry resulting from the 32-bit subtraction is left above the top of the stack (in bit 15), so that it may be recovered by a *Push* instruction if required.

Multiply
MUL (1):

```
pop[x]
pop[y]
push[(x*y)[16:31]]
push[(x*y)[0:15]]
stkp ← stkp-1
```

The top two elements of the stack are multiplied, and the result, which is a 32-bit quantity, is pushed onto the stack with the most significant 16 bits in the top element, and the least significant 16 bits in the second element. The stack pointer is then decremented, so that the least significant 16 bits occupy the top of stack. In most cases a 16-bit product will be desired, which is the result. If a full 32-bit product is needed, a *Push* instruction may be used to recover the most significant bits. The operands and the result are treated as two's complement numbers, and the sign of the result is calculated according to the rules of algebra.

Unsigned Multiply
UMUL (1):

```
pop[x]
pop[y]
push[(x*y)[16:31]]
push[(x*y)[0:15]]
stkp ← stkp-1
```

The top two elements of the stack are multiplied, and the result, which is a 32-bit quantity, is pushed onto the stack with the most significant 16 bits in the top element, and the least significant 16 bits in the second element. The stack pointer is then decremented, so that the least significant 16 bits occupy the top of stack. In most cases a 16-bit product will be desired, which is the result. If a full 32-bit product is needed, a *Push* instruction may be used to recover the most significant bits. The operands and the result are treated as unsigned numbers.

Double
DBL (1):

```
pop[x]
push[x lshift 1]
```

The top element of the stack is left shifted by 1.

Divide
DIV (1):

```
pop[x]
pop[y]
quot,,rem ← y/x
push[quot]
push[rem]
stkp←stkp-1
```

The top element of the stack is used as a 16-bit signed divisor, the second element is taken as a 16 bit signed dividend. The division is performed, and the 16-bit quotient is pushed onto the stack. The remainder is left above the top of stack. Divisor and dividend are treated as two's complement numbers, and the signs of the quotient and remainder are calculated according to the following rules:

Dividend	Divisor	Quotient	Remainder
positive	positive	positive	positive
positive	negative	negative	positive
negative	positive	negative	negative
negative	negative	positive	negative

The division is not performed and the trap *ZeroDivisor* is generated if the divisor is zero.

Long Divide
LDIV (1):

```
pop[x]
pop[y]
pop[z]
quot,,rem ← y,,z/x
push[quot]
push[rem]
stkp ← stkp-1
```

The top element of the stack is used as a 16-bit signed divisor, the second and third elements are used as a 32-bit signed dividend (with the least significant bits in the third element). The division is performed, and the 16-bit quotient is pushed onto the stack. The remainder is left above the top of stack. Divisor and dividend are treated as two's complement numbers, and the signs of the quotient and remainder are calculated according to the rule for DIV.

If the magnitude of the most significant half of the dividend is greater than that of the divisor, the trap *DivideCheck* is generated, indicating that the quotient will not fit into a single word. If the divisor is zero, the trap *ZeroDivisor* is generated. If either trap occurs, the division is not performed.

Unsigned Divide
UDIV (1):

```
pop[x]
pop[y]
pop[z]
quot,,rem ← y,,z/x
push[quot]
push[rem]
stkp←stkp-1
```

The top element of the stack is used as a 16-bit unsigned divisor, the second and third elements are used as a 32-bit unsigned dividend (with the least significant bits in the third element). The division is performed, and the 16-bit quotient is pushed onto the stack. The remainder is left above the top of stack. Divisor and dividend are treated as unsigned numbers.

If the magnitude of the most significant half of the dividend is greater than that of the divisor, the trap *DivideCheck* is generated, indicating that the quotient will not fit into a single word. If the divisor is zero, the trap *ZeroDivisor* is generated. If either trap occurs, the division is not performed.

The following instruction negates (2's complement) the value on top of stack.

Negate
NEG (1):

```
pop[x]
push[-x]
```

The following instruction adds 1 to the value on top of the stack.

Increment
INC (1):

```
pop[x]
push[x+1]
```

The following instruction adds 2 to the value on top of the stack.

ADD 2
ADD2 (1):

```
pop[temp]
push[temp+2]
```

The following instruction subtracts one from the value on top of the stack.

DECrement
DEC (1):

```
pop[temp]
push[temp-1]
```

The following instruction adds sign-extended α to the value on top of the stack.

ADD Sign-extended Byte
ADDSB (2):

```
pop[temp]
push[temp + (if α[0] = 1 then α + 177400B else α)]
```

The following instruction calculates and pushes the bitwise logical *and* of the top two elements of the stack.
And
AND (1):

```
pop[x]
pop[y]
push[x and y]
```

The following instruction calculates and pushes the bitwise logical *or* of the top two elements of the stack.
Or
OR (1):

```
pop[x]
pop[y]
push[x or y]
```

The following instruction calculates and pushes the bitwise *exclusive or* of the top two elements of the stack.

Exclusive OR
XOR (1):

```
pop[x]
pop[y]
push[x xor y]
```

The following instructions compute the maximum or minimum of the top two elements of the stack, respectively.

signed MAXimum
MAX (1):

```
pop[temp1]
pop[temp2]
IF temp1 > temp2 THEN push[temp1] ELSE push[temp2]
```

signed MINimum
MIN (1):

```
pop[temp1]
pop[temp2]
IF temp1 < temp2 THEN push[temp1] ELSE push[temp2]
```

Unsigned MAXimum
UMAX (1):

```
pop[temp1]
pop[temp2]
IF temp1 > temp2 THEN push[temp1] ELSE push[temp2]
```

Unsigned MINimum
UMIN (1):

```
pop[temp1]
pop[temp2]
IF temp1 < temp2 THEN push[temp1] ELSE push[temp2]
```

Jump Instructions

All Jump instructions are PC relative. The program counter is a 16-bit byte displacement relative to the code segment base. The even byte is in bits 0-7 of a word, the odd byte is in bits 8-15 of a word. During the execution of an instruction, the PC points to the instruction, so that if a trap occurs, the pc does not have to be backed up before it is stored. The effect of this is that the PC is incremented, *then* an instruction byte is accessed.

Unconditional Jumps

These instructions obtain a displacement from the opcode, α, or $\alpha\beta$, and add it to the PC value.

Jump +n, n=2-9
Jn (1):

PC←PC+n

Jump Byte
JB (2):

PC←PC+α (α is sign extended, providing a range of -128 to +127 bytes relative to the JB)

Jump Word
JW (3):

PC←PC+$\alpha\beta$

Conditional Jumps

These instructions compare the top element of the stack with the second element, and branch to a location determined by the opcode or α if the comparison is true. The operators assume two's complement operands, and the comparisons are signed.

Jump Equal +n, n=2-9
JEQn (1):

```
pop[y]
pop[x]
if x=y then PC←PC+n
```

Jump Equal Byte
JEQB (2):

```
pop[y]
pop[x]
if x=y then PC←PC+α
```

Jump Not Equal +n, n=2-9
JNEn (1):

```
pop[y]
pop[x]
if x#y then PC←PC+n
```

Jump Not Equal Byte
JNEB (2):

```
pop[y]
pop[x]
if x#y then PC←PC+α
```

Jump Less Byte
JLB (2):

```
pop[y]
pop[x]
if x<y then PC←PC+α
```

Jump Greater Equal Byte
JGEB (2):

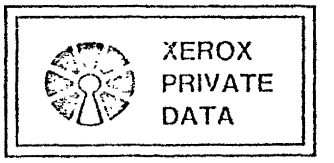
```
pop[y]
pop[x]
if x>=y then PC←PC+α
```

Jump Greater Byte
JGB (2):

```
pop[y]
pop[x]
if x>y then PC←PC+α
```

Jump Less Equal Byte
JLEB (2):

```
pop[y]
pop[x]
if x=<y then PC←PC+α
```



These instructions compare the top element of the stack with the second element, and branch to a location determined by the α if the comparison is true. The comparisons are unsigned.

Jump Unsigned Less Byte
JULB (2):

```
pop[y]
pop[x]
if x<y then PC←PC+α
```

Jump Unsigned Greater Equal Byte
JUGEB (2):

```
pop[y]
pop[x]
if x>=y then PC←PC+α
```

Jump Unsigned Greater Byte
JUGB (2):

```
pop[y]
pop[x]
if x>y then PC←PC+α
```

Jump Unsigned Less Equal Byte
JULEB (2):

```
pop[y]
pop[x]
if x=<y then PC←PC+α
```

The following instructions compare the top of stack with zero, and branch to the location given by α if the comparison is true.

Jump Zero Byte
JZB (2):

```
pop[x]
if x=0 then PC←PC+α
```

Jump Not Zero Byte
JNZB (2):

```
pop[x]
if x#0 then PC←PC+α
```

The following instructions compare the top of stack with NIL (177777B), and branch to the location given by α if the comparison is true.

Jump Nil Byte
JNB (2):

```
pop[x]
if x=-1 then PC←PC+α
```

Jump Not Nil Byte
JNNB (2):

```
pop[x]
if x#-1 then PC←PC+α
```

The following instructions compare the top of stack with α, and branch to the location given by β if the comparison is true.

Jump EQual Byte Byte
JEQBB (3):

```
pop[x]
if x=α then PC←PC+β
```

Jump Not Equal Byte Byte
JNEBB (3):

```
pop[x]
if x#α then PC←PC+β
```

The following instructions compare the top of stack with the first four bits of α, and branch to the location given by the second four bits if the comparison is true.

Jump Greater Pair
JGP (2):

```
pop[x]
if x>α[0:3] then PC←PC+α[4:7] +2
```

Jump Less Pair
JLP (2):

```
pop[x]
if x<α[0:3] then PC←PC+α[4:7] +2
```

The following instructions combine the effects of RBITF and JZ.

Jump Bit EQual Pair Byte
JFEQB (3):

```
pop[pointer]
temp ← <mds+pointer+α[0:3]>
temp ← (temp rshift α[4:7]) and 1
if temp = 0 then PC←PC+β
```

Jump Bit Not Equal Pair Byte
JFNEB (3):

```
pop[pointer]
temp ← <mds+pointer+α[0:3]>
temp ← (temp rshift α[4:7]) and 1
if temp # 0 then PC←PC+β
```

Jump Indexed Byte/Word

These instructions provide a space and time-efficient method of doing the dispatch needed by a *case* statement when the density of cases is high.

The top element of the stack defines the upper limit of a range of values, the second element contains a value to be tested. If the value is in the range (i.e. stk[stkp-1]<stk[stkp] unsigned), it is used to index a table of PC displacements in the code segment, and a PC relative jump is done using this displacement. The JIB instruction uses a table of byte displacements, JIW uses a table of word displacements.

Jump Indexed Byte
JIB (3):

```
pop[y]
pop[x]
if x < y then do

        disp ← <C+αβ+x/2> --get the table entry from the code segment

        if x and 1 = 0 then disp ← disp rshift 8 --select the appropriate byte

        else disp ← disp and 377B

        PC←PC+disp
```

Jump Indexed Word
JIW (3):

```
pop[y]
pop[x]
if x < y then do

        disp ← <C+αβ+x> --get the table entry from the code segment

        PC←PC+disp
```

FOR Loop Control Instructions

The following instructions are used for the inner-most FOR loop. They use a 2-word block to hold the PC and end-condition. STFOR initializes the block and ENDFOR uses it to test for termination. The 2 and 3 byte variations of STFOR exist so the compiler can simply test to see if it is compiling the innermost loop rather than waiting to see if the scope of the loop is sufficiently small to allow use of the 2 byte variant. Local zero is always the loop control variable.

STart FOR loop Byte
STFORB (2):

```
<mds+L+localbase+8> ← PC+2
pop[mds+L+localbase+9]    -- end condition
pop[temp]
<mds+L+localbase> ← temp-1
PC←PC + α
```

STart FOR loop Word
STFORW (3):

```
<mds+L+localbase+8> ← PC+3
pop[mds+L+localbase+9]    -- end condition
pop[temp]
<mds+L+localbase> ← temp-1
PC←PC + αβ
```

signed END FOR loop
ENDFOR (1):

```
temp ← <mds+L+localbase>
temp ← temp+1
if temp < <mds+L+localbase+9> then
    begin <mds+L+localbase> ← temp; PC ← <mds+L+localbase+8>; end
```

Unsigned END FOR loop
UENDFOR (1):

```
temp ← <mds+L+localbase>
temp ← temp+1
if temp < <mds+L+localbase+9> then
    begin <mds+L+localbase> ← temp; PC ← <mds+L+localbase+8>; end
```

Miscellaneous Instructions

The PUSH instruction allows an item which was previously popped to be recovered:

Push
PUSH (1):

```
stkp ← stkp+1
```

The POP instruction discards the top value on the stack:

Pop
POP (1):

```
stkp ← stkp-1
```

The EXCH instruction exchanges the top two items on the stack:

Exchange
EXCH (1):

```
pop[x]
pop[y]
push[x]
push[y]
```

The DUP instruction duplicates the item on the top of the stack:

Duplicate
DUP (1):

```
pop[x]
push[x]
push[x]
```

The following instructions add G and L respectively to α.

Global Address Byte
GADRB (2):

```
push[G+α]
```

Local Address Byte
LADRB (2):

```
push[L+α]
```

The following instruction is provided to support the Mesa signalling machinery. It is a 2-byte instruction, but the value of α is ignored by the hardware:

Catch
CATCH (2):

```
noop
```

Allocate
ALLOC (1):

```
pop[temp]
push[alloc[temp]]
```

This instruction takes a frame size index (see "Frame Allocation") and returns a pointer to a frame of the requested size from the heap in the main data space.

Free
FREE (1):

```
pop[temp]
freeframe[temp]
```

This instruction takes a pointer to a frame and returns the frame to the list given by <temp-1> (see "Frame Allocation")

Shift
SHIFT (1):

```
pop[count]
pop[data]
if count < 0 then
    push[data rshift count]
else do
    push[data lshift count]
```

The first element of the stack contains a shift count, the second element contains the data to be shifted. A positive count implies a left shift. Bits which shift off the end of the word are lost, and zeroes are shifted into the word as necessary.

The BLT instruction takes a destination address, word count, and source address from the top three elements of the stack. It copies the source block to the destination block. Low addresses are transferred first, and no check is made for overlap of the source and destination blocks. If a process switch or trap occurs during a BLT, the stored PC points to the BLT instruction, so that it will resume correctly.

Block Transfer
BLT (1):

```
while stack[stkp-1] > 0 do
        <mds+stack[stkp]> ← <mds+stack[stkp-2]> --destination ← source
        stack[stkp] ← stack[stkp]+1 --increment destination
        stack[stkp-2] ← stack[stkp-2]+1 --increment source
        stack[stkp-1] ← stack[stkp-1]-1 --decrement count
        (test for process switch)
stkp ← stkp-3
```

The BLTR instruction has the same effect as the BLT instruction but the operands are reversed on the stack.

BLock Transfer Reversed
BLTR (1):

```
while stack[stkp-1] > 0 do
        <mds+stack[stkp-2]> ← <mds+stack[stkp]> --destination ← source
        stack[stkp-2] ← stack[stkp-2]+1 --increment destination
        stack[stkp] ← stack[stkp]+1 --increment source
        stack[stkp-1] ← stack[stkp-1]-1 --decrement count
        (test for process switch)
stkp ← stkp-3
```

Block Transfer Long

This instruction is similar to BLT, except that the first and third elements on the stack are long pointers.

Block Transfer Long
BLTL (1):

```
while stack[stkp-2] > 0 do
        <stack[stkp]..stack[stkp-1]> ← <stack[stkp-3]..stack[stkp-4]> --destination
        ←source

        stack[stkp]..stack[stkp-1] ← (stack[stkp]..stack[stkp-1])+1 --increment
        destination (doubleword increment)

        stack[stkp-3]..stack[stkp-4] ← (stack[stkp-3]..stack[stkp-4])+1 --increment
        source (doubleword increment)

        stack[stkp-2] ← stack[stkp-2]-1 --decrement count
        (test for process switch)
stkp ← stkp-5
```

The RR and WR instructions allow access to the static registers of the processor. The register to be accessed is determined by α:

α:	register:	meaning:
1	gft	Global Frame Table
2	WW	Wakeups Waiting
3	AP	Active Processes
4	RP	Ready Processes
5	CPN	Current Process Number
6	WDC	Wakeups Disabled Counter
7	mds	Main Data Space
8	Restart	Reason for System Restart

Write Register
WR (2):

pop[reg[α]]

Read Register
RR (2):

push[reg[α]]

The ROR and RAND instructions are provided to eliminate race conditions in the software when changing the contents of the registers associated with the process switching system. These instructions must be atomic, i.e. no change can occur to the register between the time it is read and written.

Register OR
ROR (2):

pop[temp]
push[reg[α]]
reg[α] ← reg[α] or temp

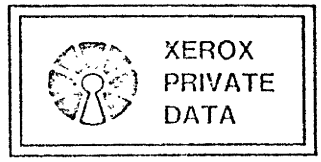
Register AND
RAND (2):

pop[temp]
push[reg[α]]
reg[α] ← reg[α] and temp

The following instructions manipulate the map. See "Operations on the Map"

Associate
ASSOC (1):

pop[tempr] -- bits 0-2 contain f (i.e. W,D,Ref). Bits 4-15 contain a real page number
pop[tempv] -- a 16-bit virtual page number
Assign real page r to virtual page v in the map. Set the flag bits in the map entry to f.



SetFlags
SETF (1):

pop[vp] --a virtual page number
pop[tempf] --flag bits in tempf[0:2]
tempr ← Map entry for virtual page vp. If there is no such entry, tempr[0:2] ← 6 (vacant) and tempr[4:15] ← 0. If the entry exists, bits 0:2 are the flags, bits 4:15 are the real page number. If a non-vacant map entry exists for virtual page vp, set its flag bits from tempf[0:2]
push[tempr] --return old flags and real page number
Operations which might modify the map entry must be disallowed between the time the entry is read and the time it is subsequently updated (the operation must be atomic).

Bit Boundary Block Transfer

The BitBLT (Bit Boundary Block Transfer) instruction has two primary uses: The first is to move regions of storage containing a rectangular area of a display bitmap from one location in memory to another. This corresponds to moving a portion of an image on a display screen. The second principal application is character scan conversion, in which a region of a *font* (a data structure containing the bitmap representation for characters) is transferred to a particular location in a region of storage containing a display bitmap. The instruction has a number of other potential applications, but these two are expected to dominate.

Display Bitmap Format

The format of a display bitmap in storage is shown in figure 4a. By convention, the origin is at the upper left corner of the screen. The x coordinate increases to the right, y increases downward (this corresponds to the direction of scan in the display). The bitmap is composed of w pixels horizontally, h pixels vertically. Each pixel is represented by a single bit in storage. Although w need not be a multiple of 16, each scan line must start on a word boundary (the final 16-(w mod 16) bits of the last word of a scan line are not used). If the bitmap starts at location a, and the number of words per scan line is k, the second scan line starts at location a+k, the third at location a+2*k, etc. If x,y is the coordinate of a pixel, the address of the pixel is a+k*y+x/16, and its bit number is x mod 16. This assumes square pixels, i.e. equal vertical and horizontal resolution. If the display controller supports variable resolution, adjustment is necessary. Also, the display will usually be *interlaced* (all even scan lines displayed first, then all odd scan lines). It is the responsibility of the display controller to deal with this - the bitmap does not represent the interlace in any way.

Font Format

Figure 4b shows the bitmap representation for a single character. There are several methods available for packing the required information into a font; One representation is shown in figure 4c. In addition to the packed bit image, the font includes information which allows the software to find the bitmap for a particular character given its ASCII code, and determine the height, width and baseline of the character. Exact details of the font format are unimportant for the present discussion, and will be omitted.

BitBLT

An *item* is a contiguous string of bits of *width* w. The BitBLT instruction fetches an item from a *source address* sa,,sb (a bit address consisting of a source word address and a bit number), then stores it at a *destination address* da,,db (a bit address). The instruction also allows specification of a *function* to be performed on the source and destination data before storing. Possibilities are {this list is not exhaustive}:

f	Operation
0	dest ← source
1	dest ← dest *or* source
2	dest ← dest *and* source
3	dest ← dest *xor* source
4	dest ← dest *and not* source

After each item is transferred, the source and destination addresses are incremented by two quantities sai and dai, which are the (signed) bit offsets for accessing the next source and destination items. The instruction also requires specification of the total number of items to be transferred.

To deal with the situation in which the source and destination blocks overlap, we adopt the convention that if the item width is negative, sa,,sb is the bit address of the bit following the last bit of the item, and the items are transferred from high addresses to low addresses (see Figure 4d).

BitBLT takes its arguments from the stack, and the Mesa compiler must ensure that the stack is empty except for these arguments at the time the instruction is executed. The arguments and their positions on the stack are:

```
stack[stkp]:    nitems (16 bit unsigned item count)
stack[stkp-1]:  bits 0:3 = sb,  bits 8:15 = sa[0:7]
stack[stkp-2]:  sa[8:23]
stack[stkp-3]:  bits 0.3 = db,  bits 8:15 = da[0:7]
stack[stkp-4]:  da[8:23]
stack[stkp-5]:  sai  (16 bit 2's complement)
stack[stkp-6]:  dai  (16 bit 2's complement)
stack[stkp-7]:  bits 0:3 = function,  bits 4:15 = item width in bits (2's complement)
stack[stkp-8]:  intermediate state (must be initialized to 0 by the program)
```

After transferring each item, the instruction updates the parameters on the stack:

```
nitems ← nitems-1
sa ← sa+sai
da ← da+dai
```

The word containing intermediate state is provided to allow a process switch or page fault to occur during the transfer of a multiword item. This word will contain the number of bits processed for the current item if the instruction is interrupted, and will allow the instruction to continue from its point of interruption when control returns.

BitBLT Examples

To copy the rectangular area A in figure 4a to the origin of the screen, the parameters for BitBLT are:

```
sa= a+6*k+3, sb = 12
da = a, db = 0
sai = dai = 16*k
nitems = 4, width = 75
```

To scan convert the character in figure 4a at the display origin, the parameters are:

```
sa = start address in font,   sb = 0
da = a,   db=0
sai = 6, dai = 16*k
nitems = 8, width = 6
```

Extensions to BitBLT

Since the BitBLT instruction has significant setup overhead if the number of words transferred is small, we may wish to provide additional instructions which set up its arguments given some amount of higher level information (e.g. an ASCII character and a pointer to a font), in a manner analogous to the way the XFER primitive is used. Neither the necessity for these instructions nor their format has been determined.

Control Transfers

Control Links

Most of the control transfer instructions take as an argument a 16 bit *Control Link*. The least significant two bits of the control link determine its type:

Bits 14:15	Meaning
0	The control link is a frame pointer (an mds pointer). This convention forces frames to lie on 4-word boundaries.
1,3	Bits 0:15 are a Procedure Descriptor (see below).
2	The control link is an indirect pointer (an mds pointer)

Objects must be allocated in storage such that indirect and frame pointers automatically have the correct F field, i.e. frames must be at addresses which are 0 mod 4, indirect words must be at addresses which are 2 mod 4. This convention is enforced by the software.

Procedure Descriptors

A procedure descriptor is used in many control transfer operations to obtain the global frame pointer G, the code segment pointer C, and the PC value for a procedure. It consists of two fields:

Bits 0:10	Global Frame Table Index (gfti)
Bits 11:14	entry number (en)

The GFT index is used to retrieve the global frame pointer from the Global Frame Table in the main data space. Since the global frame must be located on a four-word boundary, the least significant two bits of the GFT entry are not used to point to the Global Frame. Instead they are used in conjunction with the entry number to obtain an index into the entry vector of the code segment associated with the global frame. This allows code segments to contain up to 64 procedures. The entry vector contains the starting PC value for the procedure (in C-relative form), and the frame size index for the frame required by the procedure.

Precisely:

```
G ← <mds+gft+gfti>[0:13]*4 -- global frame pointer (an mds pointer)
C ← <mds+G+4>,,<mds+G+3> -- 24-bit code pointer
evx ← <mds+gft+gfti>[14:15]*16 + en +2 -- entry vector index
The PC value depends on whether the short or long form of an entry vector item is used:
evi ← <tC+evx> --the entry vector item
if evi <0 then do
        tPC ← evi[1:15]*2 --program counter
        fsi ← <tC+tPC/2-1>[4:15] --frame size index or frame size
else do
        tPC ← evi[5:15]
        fsi ← evi[1:4]
```

Stored Program Counters

All control transfer instructions except RET and LSTF begin by storing the PC at L[1] in the local frame. The PC is the byte offset relative to the code segment base C of the instruction which is to be executed when the local frame is resumed. This convention limits the size of a code segment to 32K words.

Frame Allocation

Some of the control transfer instructions and the ALLOC instruction allocate and free frames from the heap accessed via the allocation vector av. These instructions make use (conceptually) of two primitive operations, *alloc*[fsi], and *freeframe*[frame]. The former takes a frame size index and returns a frame of the requested size (or larger if indirection occurs) from the heap in the main data space. If alloc cannot satisfy the request, it causes a trap. *Freeframe* takes a frame pointer and returns the frame to the appropriate list in av. The structure of av and the heap is shown in figure 2b.

The allocation vector begins at location av in all MDS's. It contains a vector of pointers to the various frame sizes made available by the software. Since by convention frames begin on four word boundaries the last two bits of these pointers are not needed for pointing to a frame. Instead, they are used as a flag according to the following conventions:

Flag Meaning:

0: This is a normal frame pointer.
1: The list for this size frame is empty.
2: This entry is an indirect pointer in the form of an av-relative displacement of the frame size which should be used instead of this size. (This is customarily placed in the last entry of a list of a given size if it is desired to use a larger frame size should this frame size list be exhausted.)
3: Pointer to a normal frame, and decrement Wakeup Disable Counter when encountered.

Bits 0:13 of an allocation vector entry are the fp field; fp*4 usually points to the frame which will next be allocated when an allocation request for a frame of the appropriate size is received. The frames for each size are arranged in a linked list. (Note that the pointers are to what is apparently the second word of the frame in figure 2b. Since these pointers become the frame pointers, the frame which is used by the firmware and software actually begins at this location. The word which contains the frame size index actually precedes the frame, hence may be thought of as being in location -1 relative to the beginning of the frame. This word must be preserved by the software so that it is available for use by the *freeframe* primitive. Hence, although it is not strictly speaking part of the frame, it must be preserved as long as the frame exists.) When an allocation occurs fp*4 is returned to the requester, and the contents of the word to which it pointed (including the flag bits) are brought into the allocation vector. Thus, frames are allocated from, and ultimately returned to, the head of the linked list, and the allocation vector entry usually points to the next frame to be allocated. The last frame in a list either contains an end of list flag (f = 1) or an indirect flag (f = 2). When this frame is finally pointed to by the allocation vector and an allocation occurs for this frame size, it is allocated, and its pointer is stored in the allocation vector. Should another request for a frame of this size be received before a freeframe operation occurs for a frame of this size, this flag will then take effect. If it is an indirect flag (f = 2) fp is used as a frame size index to access another

(presumably larger) frame size for allocation. If it is an end of list flag (f = 1) a trap occurs.

The intention is that av will contain a limited number of frame sizes which will be sufficient for the majority of requests. To support (infrequent) requests for frames which are larger than the size normally accommodated by av, it is possible for the fsi to contain the frame size directly. If fsi is greater than MaxAllocSlot-*1* (a constant whose value has not yet been determined), an attempt is made to allocate the frame from av[MaxAllocSlot]. This av slot is usually empty, and the allocation attempt will cause the trap *AllocationListEmpty[fsi: FrameSizeIndex]*. The allocation trap handler (software) will note that the request is for a 'large frame', acquire storage for the frame, and add it to the list av[MaxAllocSlot]. The trap handler will then return, restarting the instruction which did the original *alloc*, which will now succeed. To ensure that av[MaxAllocSlot] remains empty, the software will arrange large frames such that they are freed onto the list av[LargeReturnSlot]. (LargeReturnSlot is another as-yet-undetermined parameter). This slot is never used for allocation, and it is the responsibility of the software to deal with the frames which are freed onto this list.

Since a number of processes may share the same main data space (and thus share the same av and heap), it is necessary to disable wakeups when an allocation trap occurs to ensure that the trap handler will not be preempted before it has made more frame space available. Wakeups must be disabled between the time the *alloc* fails and the time its subsequent reexecution succeeds (after the trap handler has provided more frame space). The mechanism specified here accomplishes this goal without complicating the trap mechanism, i.e. the allocation trap handler does not need to be aware of the previous state of the process switching system (wakeups enabled or disabled). It is possible for nested allocation traps to occur to a reasonable level, providing that the trap handler sets up another frame of a size suitable for itself before taking any action which might cause another allocation trap.

Wakeups are disabled when the (hardware) counter WDC is nonzero (WDC contains a count of the number of reasons wakeups are disabled). When the alloc primitive cannot satisfy a request, it increments WDC and causes a trap in the normal manner. When the trap handler supplies more frame storage, it arranges av so that a subsequent attempt to allocate a frame of the size indicated by fsi will yield a frame pointer with a flag field of 3 (rather than 0, which is the normal case -- refer to figure 2). Whenever *alloc* encounters a type 3 flag, it returns the frame to the requester, but also decrements WDC, which will reenable wakeups if they were originally enabled (and the trap was not nested).

In detail, the *alloc* and *freeframe* primitives do:

```
alloc:
        ofsi ← fsi --save original fsi for possible trap
        if fsi > MaxAllocSlot-1 then fsi ← MaxAllocSlot --request is for a 'large' frame
        frame ← <mds+av+fsi>  --the head of the list
        while frame[14:15] = 2 do frame ← <mds+av+frame/4> --indirection
        if frame[14:15] = 1 then do

                if WDC = 255 then trap[WakeupError] --counter would overflow
                WDC←WDC+1 --disable wakeups
                trap[AllocationListEmpty,ofsi] --ofsi is the trap parameter
```

```
else do --flag = 0 or 3

        <mds+av+fsi> ← <mds+(frame and 177774B)> --note that if the reference to the
        frame causes a page fault, the store into av will not occur, and the instruction
        which includes the alloc will be restarted after the fault is fixed.
        if frame[14:15] = 3 then

                if WDC = 0 then trap[WakeupError] --counter would underflow
                WDC ← WDC-1 --reenable wakeups

        return (frame and 177774B)

freeframe:

        fsi ← <mds+frame-1> --fsi is stored one location before the frame
        <mds+frame> ← <mds+av+fsi> --add the frame to the head of the list
        <mds+av+fsi> ← frame
```

XFER

Most of the control transfer instructions, the trap mechanism, and the processing switching facility make use of the primitive operation:

```
XFER[dest: control link, source: control link,xtype: xfertype, TrapParameter: integer]
where:
xfertype: TYPE = {freetype, nofreetype, traptype, pswitchtype}
```

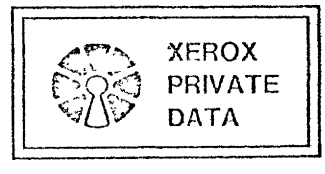
The differences between instructions have to do with the way in which the source and destination links are generated, whether or not the local frame is to be freed, the handling of the source and destination links, and whether the trap parameter (which is required only if xtype=traptype) is stored into the local frame.

The idea of XFER is that the basic primitive may be used to construct a variety of control disciplines which can work together, since they all use the same primitive operation and data structures.

In detail, the XFER primitive does:

(In what follows, tX is used to designate a temporary value for register X, used in situations in which X cannot yet be modified due to the possibility of a trap) -

```
XFER:
        tdest ← dest
        while dest[14:15]=2 do dest ← <mds+dest> --destination link is indirect
        if dest[15] = 1 then do --destination link is a procedure descriptor

                gfti ← dest[0:10] -- extract the two fields of the procedure descriptor
                en ← dest[11:14]
                tG ← <mds+gft+gfti>[0:13]*4 -- obtain global frame pointer
                tC ← <mds+tG+4>,,<mds+tG+3> -- obtain 24-bit code pointer

                if tC is odd then trap[CodeMappedOut,tdest] --code segments normally start on an
                even word boundary.  This trap implies that the code is not in the virtual space,
                and is not related to the VMS's page fault traps.

                evx ← <mds+gft+gfti>[14:15]*16+en+2 -- obtain entry vector index
                evi ← <tC+evx> --the entry vector item
                if evi <0 then do
                        tPC ← evi[1:15]*2 --program counter (byte displacement from tC)
                        fsi ← <tC+tPC/2-1>[4:15] --frame size index or frame size
                else do
                        tPC ← evi[5:15]
                        fsi ← evi[1:4]

                tL ← alloc[fsi] --memory references beyond this point cannot page fault, since
                alloc references the first location of the frame, and the first four locations of the
                frame will not cross a page boundary (since frames are placed on four word
                boundaries by the system software)
                <mds+tL> ← tG  --initialize new frame's static link
                <mds+tL+2> ← source --store return link

        else do --destination link is a frame pointer

                if dest[0:13]=0 then trap[NullDestinationLink,tdest]
                tL ← dest
                tG ← <mds+tL> --the first location in the frame
                tPC ← <mds+tL+1> --the second location in the frame
                tC ← <mds+tG+4>,,<mds+tG+3> --24-bit code pointer
```



```
        if tC is odd then trap[CodeMappedOut,tdest] --code segments normally start on an
        even word boundary.  This trap implies that the code is not in the virtual space,
        and is not related to the VMS's page fault traps.

CONT:
        test for a breakpoint return (see "Breakpoints")
        SELECT xtype FROM
                =freetype =>BEGIN; freeframe[L]; stack[stkp+1]←source; stack[stkp+2]←tdest; END;
                =nofreetype =>BEGIN; stack[stkp+1]←source; stack[stkp+2]←tdest; END;
                =traptype =>L[3]←TrapParameter;
        ENDCASE;
        L←tL --update processor registers
        G←tG
        PC←tPC
        C←tC
```

In the normal case, the source and original destination links are left above the top of the stack so that the context which is getting control can use them if it wants to.

Control Transfer Instructions

The LINKB instruction is executed on entry to nested procedures to establish the back link to the enclosing context. It recovers the destination link of the last XFER, subtracts α, and stores the result in local 0.

Link Byte
LINKB (2):

```
<mds+L+localbase> ← stack[stkp+2] - α
```

The following instruction is used to create local procedure descriptors and signal descriptors. It constructs a 16-bit descriptor from the global frame index of the current frame and α. G[GFTIoffset] contains the GFT index of the current global frame in procedure descriptor form, i.e., with GFTI in bits 0:10.

Descriptor
DESCB (2):

```
push[<mds+G+GFTIoffset> + α]
```

The following instructions create procedure descriptors and signal descriptors. They construct a 16-bit descriptor from the global frame index of the frame on top of the stack and α.

Descriptor Stack
DESCBS (2):

```
pop[pointer]
push[<mds+pointer+GFTIoffset> + α]
```

Call Descriptor Stack
FDESCBS (2):

```
L[1]←PC+2
pop[pointer]
XFER[<mds+pointer+GFTIoffset> + α, L, nofreetype]
```

Local Function Calls

These instructions are used to call a procedure in the current code segment. The local function call instructions are optimizations of the XFER mechanism made possible by the fact that a code segment is compiled as a single entity. The compiler can thus build the information necessary to find the procedure to be called into the code itself, rather than having to wait until the context is bound, as is the normal case. n is the entry number:

Local Function Call n, n=1-8
LFCn (1):

```
        L[1] ← PC+1 --store the PC
LFC:

        evi ← <C+n+2> --the entry vector item
        if evi <0 then do
                tPC ← evi[1:15]*2 --program counter (byte displacement from C)
                fsi ← <tC+tPC/2-1>[4:15] --frame size index or frame size
        else do
                tPC ← evi[5:15]
                fsi ← evi[1:4]

        tL ← alloc[fsi] --get a frame. No page faults can occur beyond this point.
        <mds+tL> ← G  --save static link in the new frame
        <mds+tL+2> ← L --save return link in the new frame
        L ← tL
        PC ← tPC
```

Local Function Call Byte
LFCB (2):

```
        L[1] ← PC+2 --store the PC
        n ← α
        go to LFC
```

Global Function Calls

These instructions access a control link in the global frame, and do an XFER using it:

Global Function Call n, n=0-15
GFCn (1):

```
        L[1] ← PC+1 --Store the PC
        XFER[<mds+G+22+n>,L,nofreetype] --The (22+n)th global contains the destination link, the
        source link is the frame pointer
```

Global Function Call Byte
GFCB (2):

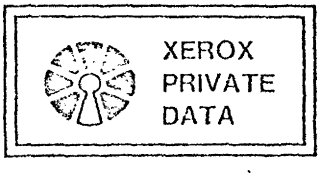
```
        L[1] ← PC+2 --Store the PC
        XFER[<mds+G+α>,L,nofreetype] --the α-th global contains the destination link, the source
        link is the frame pointer
```

Stack Function Call

This instruction XFERs via the control link on the top of the stack.

Stack Function Call
SFC (1):

```
L[1] ← PC+1 --Store the PC
pop[temp]
XFER[temp,L,nofreetype]
```



Kernel Function Call

This instruction XFERs to the function whose control link is in the α-th position of the system dispatch table. The system dispatch table sd (which starts in the same location in all main data spaces) contains contains control links for these *kernel procedures*. The offsets in sd of control links for commonly used Mesa runtime procedures are known to the compiler, which allows it to build non-local linkage information into the code.

Kernel Function Call Byte
KFCB (2):

```
L[1] ← PC+2 --Store the PC.
XFER[<mds+sd+α>,L,nofreetype]
```

Return

These instructions are used to return from a procedure.

Return
RET (1):

```
XFER[<mds+L+2>,0,freetype]
```

The following instructions return -1 and 0 respectively.

RETurn NIL
RETNIL (1):

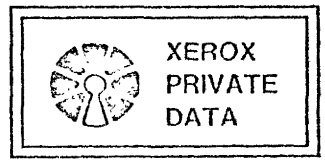
```
push[-1]
XFER[<mds+L+2>,0,freetype]
```

RETurn Zero
RETZ (1):

```
push[0]
XFER[<mds+L+2>,0,freetype]
```

Port Out

This instruction is used to transfer control through a *Port*, which is a two word area in the main data space. PORTO instructions are always immediately followed statically by PORTI instructions, as shown in figure 5. Ports are used to provide, among other things, a coroutine control discipline. Port calls are compatible with procedure calls, in that control can leave a context using the port discipline and enter a context which uses a procedure discipline and vice versa; the various cases are shown in figures 5a-5c. There are two PORTO instructions, with different opcodes but identical effects. The purpose of this is to allow the software to determine the intended usage of the PORTO when a control fault occurs.



The instruction does:

Port Out
PORTO (1):
PORTOB (1):

```
L[1] ← PC+1 --Store the PC
pop[temp] --get pointer to port
<mds+temp> ← L --set the inport to point to the current context.
XFER[<mds+temp+1>,temp,nofreetype]
```

Port In

This instruction saves the return link (which was left above the stack by the PORTO) in the outport, and clears the inport:

Port In
PORTI (1):

```
<mds+stack[stkp+2]> ← 0 --clear the inport .
If stack[stkp+1] # 0 then <mds+stack[stkp+2]+1> ← stack[stkp+1] --save the source link
in the outport unless the XFER was a procedure return
```

Traps

Traps indicate the occurence of infrequent exceptional conditions encountered in the course of instruction access or execution. Unlike process wakeups, traps notify the software of *internal* conditions which require special action, rather than *external* conditions. In some cases, the trap indicates that an error which precludes continued execution has occurred (e.g. *UnimplementedInstruction*). In other cases, the trap will cause the system software to take some action and continue the normal execution sequence. To avoid complexity in the software, we have adopted the view that when a trap occurs, the machine state will be brought (as nearly as possible) to the value it had at the start of the instruction which caused the trap. In particular, the stack will have its initial values, except in the case of an interruptible instruction such as BLT, which may have made considerable progress and then trapped. These instructions will stop in such a way that they can be restarted, in a manner identical to the situation on a process switch. When a page fault occurs on a single word store or the first word of a doubleword store, main storage will be unaffected. If a page fault occurs on the second word of a doubleword store, the first word may have been placed in memory (the result of this case is unpredictable).

Types of Trap

The following list is a summary of the traps which may be generated by the processor. The number preceding the trap name is the *trapnumber*, which is used as an index into the System Dispatch table to select the proper procedure to handle the trap. The quantity in parentheses, if present, is the trap parameter, which provides the handler with additional information concerning the trap. The possible traps are:

0 Breakpoint
1 WriteProtect (virtual page number being accessed)
2 PageFault (virtual page number being accessed)
3 AllocationListEmpty (frame size index)
4 NullDestinationLink (original destination link)
5 Unimplemented Instruction
6 StackError
7 WakeupError
8 ZeroDivisor
9 DivideCheck
10 BlockError
11 CodeMappedOut (original destination link)
12 HardwareError
13-23 Reserved for Expansion

Trap Processing

When a trap occurs, the action invoked is very similar to the XFER operation, with the following differences:

> The trap mechanism stores the trap parameter into the fourth word (L[3]) of the handler's frame, rather than passing it on the stack as do normal XFERs. This is done because some instructions leave information above the stack and therefore the entire stack must be preserved when a trap occurs. Note that this convention implies that the control link in sd cannot be a pointer to a global frame, since this word is used to hold the code pointer C in this case.

The precise actions which must be taken by the processor to cause a trap are:

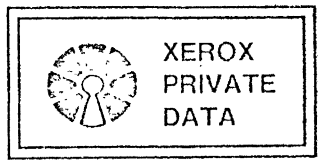
> Instructions which do a *pop* followed by a *push* must abort the push, i.e. the stack must be restored to its condition at the start of the trapped instruction.
>
> Restore the stackpointer to the value it had at the start of the instruction.
>
> if PC # 0 then L[1] ← PC --The stored PC points to the instruction which was in execution when the trap condition was detected, or to the instruction which was about to be executed if the trap occurred as a result of an instruction fetch. Normally, PC can never be zero during normal instruction execution. However if a trap occurs during a process switch, the processor may not have acquired a valid local frame, and storing the PC in this case would not make sense. This situation is discussed more fully under "Process Switching".
>
> XFER[dest,L,traptype,trap parameter]

Since the processor will roll instructions back, either to their starting point or to a well-defined intermediate state, all traps *appear* to occur between instructions. If a given instruction causes more than one type of trap, the traps will occur sequentially, and the processor will attempt to restart the instruction when the handler for each type of trap returns. Because instructions are restarted, rather then being continued from the point at which a trap condition is detected, there is no necessity to consider the effects of multiple traps on a single instruction, nor does the trap handler need to concern itself with the continuation of particular instructions.

The first instruction executed by the trap handler will be a *Dumpstack*, which will save the trapped instruction's stack in the local frame of the handler:

Dumpstack
DSTK (2):

```
pstate ← mds+L+α
<pstate> ← stkp --store the stack and stkp into the local frame
for i = 1 to stkmax do <pstate+i> ← stk[i]
Stkp ← 0 --reset the stackpointer
```



When the handler is ready to continue execution of the interrupted program, there are two types of Loadstate instructions available for this purpose, one which frees the handler's frame, and one which does not:

Load State
LST (2):

```
L[1] ← PC+2 --Store the PC
pstate ← mds+L+α --pointer to the state in the local frame
stkp ← <pstate>
for i = 1 to stkmax do stk[i] ← <pstate+i>
x←<pstate+stkmax+1> --a control link
y←<pstate+stkmax+2> --also a control link
XFER[x,y,nofreetype]
```

Load State and Free
LSTF (2):

```
pstate ← mds+L+α
stkp ← <pstate> --pointer to the state in the local frame
for i = 1 to stkmax do stk[i] ← <pstate+i>
x←<pstate+stkmax+1> --a control link
y←<pstate+stkmax+2> --also a control link
XFER[x,y,freetype]
```

Breakpoints

The single byte *Break* instruction provides a unique trap when it is encountered in the instruction stream:

Break
BRK (1):
trap[*Breakpoint*]

The Mesa debugger (software) sets a breakpoint by replacing an instruction with a BRK instruction. When the processor attempts to execute this instruction, a trap results.

When the debugger continues execution from a breakpoint, it does so with an LSTF or LST instruction. The PC of the broken context will point to the BRK instruction. Before resuming this context, the debugger will store the bytecode to be executed (the bytecode which was replaced by the BRK) in the most significant byte of the saved stackpointer (a normally unused field). It is the responsibility of the LSTF or LST instruction which resumes the broken context to inspect this byte and cause it to be executed in place of the BRK instruction if it is nonzero.

Process Switching

The processor is capable of switching control among sixteen Mesa processes. Priority scheduling of these processes is done by the hardware. It is expected that the processes scheduled by the hardware will be used for device handlers which require low latency, and that one of the levels will be used to implement a more general software scheduler for a larger number of processes.

Process States

Each of the sixteen processes which are scheduled by the hardware is described by an entry in the Process State Vector (see Figure 2a). Each psv entry is a pointer to the state information required by the processor when it runs the process.

A process from which no work is required is *blocked.* A blocked process does not compete for the use of the processor. When an external event occurs which requires service from a process, it requests a *wakeup.* If the process is *active,* i.e. permitted to run, the wakeup causes it to become *ready.* The sixteen processes supported by the hardware have fixed priorities; The highest priority ready process will acquire the processor, its state will be loaded into the machine registers from the information in psv, and it will run. When it finishes its work, it will block itself, which will relinquish the processor to a lower priority process providing that no new wakeups have been requested for the process since it last became ready.

External events which may cause wakeups include signals from device controllers implemented in hardware or microcode, and instructions explicitly executed by a program. The precise electrical and timing requirements for signalling an external wakeup must be a part of the functional specification of a particular processor model. The instruction ROR is provided to allow a program to generate a wakeup. This instruction may be used to *or* a bit mask into WW, and is provided so that this operation can be done atomically, i.e. with the assurance that no other activity may affect the WW register from the time this instruction initiates until it completes.

While a process has control of the processor, it may be preempted by a higher priority process. Preemption does not affect the state of the process, but only suspends it until it again has highest priority.

To avoid losing wakeups, a *wakeup-waiting* flag is associated with each process. When a blocked, active process receives a wakeup, it becomes ready, and its wakeup-waiting flag remains cleared. If a ready process receives a wakeup, the wakeup-waiting flag is set. When the process subsequently attempts to block itself, the wakeup-waiting flag will be cleared and the process will remain ready. This mechanism is provided by a scheduler implemented in hardware or microcode.

The eight-bit counter WDC is provided to disable process switching. If WDC#0, process switching is disabled. Instructions are provided to increment and decrement WDC, and it is automatically incremented and decremented by the frame allocation mechanism (see "Frame Allocation"). The instructions which manipulate WDC are:

Increment Wakeup Disable Counter
IWDC (1):

```
if WDC = 255 then trap[WakeupError]
WDC ← WDC+1
```

Decrement Wakeup Disable Counter
DWDC (1):

```
if WDC = 0 then trap[WakeupError]
WDC ← WDC-1
```

WDC is initialized to 1 by system reset (wakeups are disabled). An attempt to decrement it beyond zero or increment it beyond 255 will fail and cause the trap *WakeupError*.

Registers

The scheduler uses three sixteen bit registers:

RP: Contains bits corresponding to processes which are ready.

AP: Contains bits for processes which are active, i.e. permitted to run. Processes which have zeros in AP never run.

WW: Bits in this register are set by device controllers or by the processor to request wakeups.

The most significant bit (bit 0) of these registers corresponds to the lowest priority process (number 0); the least significant bit (bit 15) corresponds to the highest priority process (number 15).

In addition, there is a four bit register CPN which holds the number of the process which is currently running on the processor, and the eight-bit counter WDC which contains a count of the number of reasons wakeups are disabled.

Scheduler

In the description which follows, HPMASK[n] = 2**(15-n)-1, i.e. a mask with ones in bit positions corresponding to processes of higher priority than process n, and BITNUM[x] is the bit number (0-15) of the leading one bit in x.

When an external agent (i.e. a device controller, or other process) wishes to awaken a process, it ORs one or more bits into WW. At the beginning of every Mesa instruction, WW and AP and HPMASK[CPN] is tested, and if it is nonzero and (i.e. if a process switch is to occur), control is diverted to the scheduler.

The hardware scheduler will not be activated unless a wakeup occurs for an active process of higher priority than the one which is running, or a BLOCK instruction is executed. In the latter case, if no wakeups are pending in WW and no process is ready, control will remain in the scheduler and no process will be run. If a wakeup is received for a process of lower priority than the running process, it will be saved in WW until the running process blocks.

The scheduler does:

Schedule:

```
temp ← WW and AP and not RP --processes which are about to become ready...

RP ← RP or temp --do so...

WW ← WW and not temp --and their wakeup is cleared

Go to Schedule if RP=0 --Nothing to do, wait for a wakeup

temp ← BITNUM[RP and AP] --The number of the highest priority ready active process

Continue running the current process if temp = CP --this can occur only if control entered
the scheduler from the BLOCK instruction and there was a wakeup waiting for the process.

L[1] ← PC --store the PC in the local frame of the process being preempted

psvp = psv*256 + <psv*256 + CPN> --pointer to the state block for the process being
preempted
<psvp> ← stkp --dump the state of the current process
for i = 1 to stkmax do <psvp+i> ← stk[i]
<psvp+stkmax+1> ← L --store the local frame pointer
<psvp+stkmax+2> ← mds

PC ← 0 --clear the PC so that if a trap occurs before control gets to the new process, the
trap machinery will not store the PC

CPN ← temp

psvp = psv*256 + <psv*256 + CPN> --pointer to the state block for the new process
stkp ← <psvp> --load the state of the new process
for i = 1 to stkmax do stk[i] ← <psvp+i>
L ← <psvp+stkmax+1> --load the destination link. Usually, this will be a frame pointer, but
it may be an arbitrary control link. The destination link is placed in L so that if a trap occurs
before the new process has acquired a legitimate frame and PC, the trap handler will return
properly.

mds ← <psvp+stkmax+2>

XFER[L,0,pswitchtype]
```

When a process has completed its work, it executes a *Block* instruction, which does:

Block
BLOCK (1):

```
if WDC# 0 then trap[BlockError] --it is an error to execute a BLOCK while wakeups are
disabled
RP ← RP and not 2**(15-CPN)
go to Schedule
```

If no new wakeups have come in since the last wakeup was recognized, the process will be suspended.

Interruptible Instructions

Most instructions are uninterruptible, with process switching occurring between instructions. Some instructions (e.g. BLT) which are potentially time consuming, must be capable of being interrupted. These instructions must be implemented such that their intermediate state is indistinguishable from their initial state. When an interruption occurs during an interruptible instruction, the PC is adjusted to point to the interrupted instruction. When the process containing the interruptible instruction is restarted, the instruction will resume from the point at which it was suspended.

Errors and Error Handling

This section describes the facilities which must be provided in the processor for logging and reporting hardware-related errors detected by the processor and the memory system (device or controller detected errors are not described in this section). Although this specification does not enumerate all possible errors (since this will depend on the implementation), it does provide a reporting standard to which all portions of the system capable of detecting errors are expected to conform.

Types of Errors

Hardware errors are divided into three categories depending on their severity:

Type 1: Soft errors which do not result in loss of data. These errors are logged, but no further action is taken. Means are provided to disable the processing of type 1 errors so that a permanent error will not consume excessive time due to the logging and reporting activity.

Type 2: Hard errors which result in data loss, but from which the software may be able to recover. These errors are logged, and the source of the error is notified, so that recovery may be attempted.

Type 3: Hard errors from which no recovery is possible. These errors cause an immediate system restart. A type 2 error which occurs during the logging of another type 2 error is a type 3 error, as are type 2 errors which encounter a full logging buffer while attempting to record the state of the error.

The presently identified errors are:

Error	Type
Main storage single bit error	1
Main storage double bit error	2
I/O bus parity error	2
Control store parity error	3
Internal bus parity error	3

Error Logging

Type 1 and type 2 errors are logged in two ring buffers which are located in the first 64K of the virtual space. The software will set up these buffers at initialization time. The (fixed) locations *Type1ErrorBuffer* and *Type2ErrorBuffer* will contain pointers to two ring buffer descriptors, each of which contain four 16-bit pointers:

first:	pointer to the first location in the buffer
last:	pointer to the last location in the buffer+1
in:	location into which the next log entry will be written
out:	location from which the next log entry will be read

A buffer is empty if in=out. Error log entries are of variable length, and may wrap around the end of the buffer. The least significant byte of the first word of a log entry will contain the length of the entry in words, but all other information in a log entry, including an indication of the error type, is error-specific, and must be specified in detail for each type of error a particular implementation can detect.

If the hardware attempts to log a type 1 error and there is insufficient space in the

buffer, the entry is discarded and no further action is taken. If the hardware attempts to log a type 2 error and there is no room in the buffer, a type 3 error is generated and the system is restarted.

Software Notification of Errors

The software is not notified of the occurrence of type 1 errors, but is expected to poll the error buffer at appropriate intervals and empty the buffer if any errors are present.

The software is notified of type 2 errors in one of two ways. If the error arose as a result of the execution of Mesa code, the error is logged and the parameterless trap *HardwareError* is generated. The trap handler may inspect the log entry and take whatever action it deems necessary. If an error is detected by the system but arose as a result of an I/O operation which does not involve the processor, it is logged and the controller which caused the error is notified. The controller is expected to take the appropriate action. Usually this will involve halting any data transfer in progress, but in all cases, the controller will report the error in its next status report to the processor. Errors which arise solely as a result of I/O activity and which are detected by the controllers are also sent to the processor as status information (and are not logged).

Restart Register

The system may be restarted (bootstrapped) for a number of reasons:

The power has been turned on and has stabilized
The user has pressed the 'start' button
The software has initiated a restart
The watchdog timer has expired
A type 3 error has occurred

To allow the software (and perhaps the firmware) to take the appropriate action after a restart, a machine register (Restart) is provided to save the reason for the restart across the bootstrap activity. Bits in this register are set by hardware (or firmware) when various conditions which cause a restart are detected. The specific standard bit assignments for this register are:

Bit	Description
0	Power-on Restart
1	Start Button Pressed
2	Watchdog Timer Expired
3	Software Restart
4	Type2 Error became Type3
5	Control Memory Parity Error (Type3 Error)
6	Internal Bus Parity Error (Type3 Error)

As a part of its initialization, the processor will check the status of the power system, and set Power On and clear the other bits if this is the first restart after power has stabilized. If power was stable across the restart, the register will correctly reflect the reason for the restart.

The software which initializes the system is expected to read the contents of the restart register, take whatever action is appropriate, and clear the register in anticipation of the next restart.

Input-Output

Introduction

The input/output system provides facilities which accommodate a diverse set of I/O devices with significant performance differences and complexities and a large degree of configuration flexibility.

The input/output system is implementation-independent from the point of view of the software. The intent is to achieve software compatibility across processor configurations.

The performance differences and complexities of devices leads to device-specific I/O facilities, primarily in the amount and kind of information being transmitted between software and the I/O system. However, the facilities to effect the transfer are not device-specific. The above does not preclude the possibility that similar devices will be handled by the software in a similar fashion.

The portability of I/O handling software also depends on software which is essentially timing independent and which can adjust itself to the various device configurations.

Common I/O Handling

The facility consists of two types of I/O, Direct and Channel I/O.

In the case of DIO the input or output operation is simply the execution of an INPUT or OUTPUT instruction involving the transfer of a word of data from the top of stack to the controller or from the controller to the top of the stack.

The CIO operation is composed of a sequence of activities which are summarized here and described in detail below:

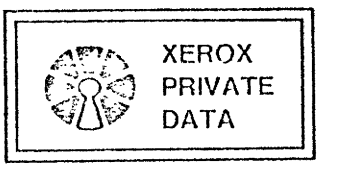
Setting up control information (software)
Starting the operation (software)
Transferring data between the device and memory
(Hardware and/or firmware)
Initiating a wakeup upon completion (Hardware and/or
firmware)
Reading status (software)

The concept of an implementation independent I/O facility makes it necessary to specify only those facilities which are accessible to the software while maintaining implementation flexibility at the processor and controllers.

The facilities for implementing CIO are the INPUT and OUTPUT instructions, a dedicated I/O address space of 256 locations, a dedicated page of virtual memory (I/O page), and process wakeups which occur as a result of specific events relative to the CIO operation.

Controllers and Devices

The software performs I/O operations through device controllers. Controllers connect devices to the processor and memory.



The complexity of the controllers varies to support the needs of devices and particular processor and memory implementations. Various processor configurations may implement controllers for similar devices in different ways, depending on the device, memory, and processor bandwidths.

I/O Addresses, Priorities and the I/O page

The I/O system contains up to 256 independently addressable I/O registers which can be read and loaded through INPUT and OUTPUT instructions at the Mesa level. The I/O address is an eight-bit quantity. Bits 0-3 address a controller and bits 4-7 address one of 16 possible registers within that controller.

Controller address 0 has been assigned to the processor and any special device controllers which are considered to be part of the processor. Controller address 15 has been assigned to an error handling function. The remaining 14 controller addresses are available for device controllers.

The mechanism for reading and loading I/O registers through INPUT and OUTPUT instructions can also be used by microprograms to read and load the registers.

The controller address assignment, except for addresses 0 and 15, is a function of the priority relationship of the controllers in a given configuration. Address 15 has the highest priority and address 0 the lowest.

Virtual memory page 0 is permanently assigned to the I/O system and is allocated to device controllers in the same manner as I/O addresses. These locations hold control information prior to the start of an I/O operation and status which reflects the result of the I/O operation at its completion. The table below shows the relationship between I/O registers, priorities and locations in the I/O page.

The assignment of the 16 I/O registers and 16 words of memory for a controller is device-specific and is defined for each device controller separately.

I/O Address	Assignment	I/O Virtual Memory Page address	Priority
0- 15	processor	0- 15	0 (lowest)
16- 31	controller 1	16- 31	1
32- 47	controller 2	32- 47	2
48- 63	controller 3	48- 63	3
64- 79	controller 4	64- 79	4
80- 95	controller 5	80- 95	5
96-111	controller 6	96-111	6
112-127	controller 7	112-127	7
128-143	controller 8	118-143	8
144-159	controller 9	144-159	9
160-175	controller 10	160-175	10
176-191	controller 11	176-191	11
192-207	controller 12	192-207	12
208-223	controller 13	208-223	13
224-239	controller I4	224-239	14
240-255	fault handling function	240-255	15 (highest)

Input/Output Instructions

The INPUT and OUTPUT instructions transfer a single word of data between an I/O register and the stack. The I/O register address is the top element of the stack.

```
INPUT(1):
    pop[temp]
    push[<I/O register temp>]

OUTPUT(1):
    pop[temp]
    pop[I/O register temp]
```

Process Wakeups

The processor must provide a means for controllers to generate process wakeup requests for the sixteen Mesa processes scheduled by the processor.

A 16-bit mask loaded into a controller register via an OUTPUT instruction or stored into one of the sixteen locations in the I/O page assigned to the controller specifies the process(es) to which wakeups are to be directed when specific events such as I/O completion or device faults occur.

The mechanism for initiating the process wakeup is controller and processor implementation dependent.

Channel I/O Operation

This section describes a typical Channel I/O (CIO) operation. The functional specification of the controllers defines the number and use of I/O registers, I/O page locations, and wakeups.

Control Information

The collection of I/O control information is stored in an I/O Control Block (IOCB). The IOCB information must be established prior to the initiation of an I/O operation and made available to the specific controller. The size and content of the IOCB is device-specific and may include information such as

Pointer to next control block if chaining is implemented
I/O command (i.e.read, write)
Pointer to buffer
Size of buffer
Process wakeup mask for I/O completion and fault handling
Unit address for multi-unit device controllers
Data address

This information is made available to the controller by OUTPUT instructions which directly load registers in the controller, or by storing the information into locations in the I/O page known to the controller. At minimum, the software must provide a pointer to the IOCB in a word in the I/O page prior to initiating the I/O operation.

Initiation

The state and availability of a controller can be obtained by reading the controller status word stored in an I/O register or in the I/O page. The initiation of an I/O operation is reflected in the controller status after some implementation-dependent amount of time. Timing-independent software must avoid sensing the status immediately after initiation.

I/O initiation can occur in two ways:

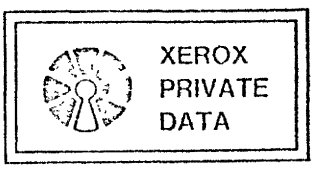
If the controller is idle after an I/O operation, the setting of a specific bit(s) in an I/O register with an OUTPUT instruction may start the I/O operation.

If the controller is never idle but performs some housekeeping at periodic intervals, then the I/O initiation may be accomplished by storing the IOCB pointer into a specific word of the I/O page. When the controller finds a non-zero value in that location, it will initiate the I/O operation. An example of such a controller is a disk controller which does some amount of processing on every sector pulse. The controller updates the current sector address and examines a specific location for an I/O initiation.

Data Transfer

The data transfer between the device and memory is controlled by the contents of the IOCB.

The handling of fixed and variable length blocks, byte, word, and multi-word boundary processing, detection of incorrect length and data chaining is controller-specific. The I/O facility does not preclude the implementation of such features.



During the data transfer, memory related faults may occur which must be handled by the controller. The action taken by the controller depends on the nature of the device. Immediate termination and the setting of the appropriate status bits followed by a process wakeup is the normal mode of fault handling as described under "Error Handling". If data transfer must continue to prevent loss of media position, for example on magnetic tapes, a suitable alternative fault handling approach must be implemented.

Termination and Process Wakeups

Unless periodic software polling is preferred for some devices, normal and abnormal termination of an I/O operation will be signalled via process wakeups after the ending status is stored in the appropriate I/O registers or locations in the I/O page.

The processes which shall receive wakeup requests upon termination are under software control.

Status Information

Termination status must provide the software at the Mesa level with sufficient information to identify the kind of termination (normal or abnormal), what software recovery steps are necessary and what user intervention actions are required.

The status described above is summary status in support of software I/O handling. Every controller must also maintain detailed diagnostic status which identifies the specific failures which led to an abnormal termination. The software shall be able to sense the fault status for error logging.

Dedicated Addresses and Functions

This section describes the assignment of block 0 and block 15 I/O registers and words in the I/O page and the associated functions. As indicated above, these blocks are reserved for the processor and special device controllers which are part of the processor.

I/O page Block 0

The assignment of block 0 (words 0 through 15) of the I/O page is as follows:

Word	Description
0-1	Time of Day The processor maintains a 32-bit time value which is incremented at one millisecond intervals.
2	MaxVM
3	MaxRM These locations are loaded as part of initialization with the maximum size of the virtual and real address spaces. The size is expressed in pages (0 indicates 2**16).
4-15	Unassigned

I/O page Block 15

The assignment of block 15 (words 240 through 255) of the I/O page is as follows:

Word	Description
240-243	I/O Controller addresses These locations contain a left-justified 56 bit value for establishing I/O controller addresses during boot and I/O reset. Each set of four bits define the address for a controller. Bits 0 through 3 define the address for the first controller, bits 4 through 7 for the second etc.
244	Type1 Error Buffer Pointer - see "Error Logging"
245	Type2 Error Buffer Pointer - see "Error Logging"
246-255	Unassigned

Block 0 I/O Registers

The assignment of I/O registers 0 through 15 is as follows:

I/O Address	Description
0-1	Processor Identification Input Register

These registers contain a unique 32 bit processor identification number. The particular implementation for attaching the identification numbers to the processor must be specified in the design specification.

2	Character Printer Input Register
3	Character Printer Output Register

One input and one output register have been assigned to the character printer. The software is expected to poll the input register at the appropriate intervals and determine from the state information obtained from the input register when the next command may be sent to the printer via the output register. The specific assignment of bits must be described in the design specification based on the selected printer implementation.

4	RS 232 Input Register
5	RS 232 Output Register

One input and one output register have been assigned to the RS 232 communication interface. The specific bit assignment must be described in the design specification.

6-15 Unassigned

Block 15 I/O Registers

The assignment of I/O addresses 240 through 255 is as follows:

I/O Address	Description
240-253	Unassigned
254	Diagnostic Readout Register (output)

This register is available to the firmware and software to display error conditions detected by diagnostics, during system bootstrapping, or when the control program is unable to communicate to the user via other means. This register will drive a set of indicators.

255 System Control Register (output)

The system control output register is assigned to system control functions as specified below:

Bit	Description
0:	I/O Reset. Setting this bit will cause a global I/O reset to all controllers. I/O controller addresses are reestablished during. I/O reset from the I/O Page, words 240 through 243.
1:	Restart Watchdog Timer. Setting this bit will restart the watchdog timer. The software must restart the watchdog timer at appropriate intervals in order to avoid a watchdog timer system restart.
2:	Software Boot. Setting this bit will cause a software initiated system boot.
3:	Disable processing of type 1 errors (this bit is set during initialization).

Block 1 through 14 I/O Registers

The assignment and use of I/O registers within a block is controller-specific with the exception of register 0 in every block. Register 0 contains the controller type identification number as well as indicators of installed optional features.

The software shall be able to issue INPUT instructions to these I/O registers, i.e. 16,32,48 ...208, and 224 and determine from the registers the number and type of controllers connected to the processor.

I/O Controller Configuration

In addition to special device controllers which are considered to be part of the processor, up to 14 I/O controllers may be connected to the processor I/O bus.

The processor I/O bus provides a common signal, timing and protocol interface to which all controllers which are part of a specific processor implementation must adhere. Based on this approach, a given controller may be connected to any one of the available controller positions.

Configuration flexibility is achieved through soft controller addressing (bits 0 - 3 of the I/O address). Tentative controller addresses are established at system initialization time for the purpose of locating potential load devices by reading register 0 of every controller. The load sequence may then modify these addresses once the physical arrangement of the controllers is known, and load the

software.

Once the software is loaded, the controller address assignment may be changed by the software (via I/O reset) to order the priority of the I/O controllers appropriately.

APPENDIX A

Mesa Instruction Set Summary

Numeric values for opcodes have not been assigned at this time.

Values of Processor Constants (*= value not determined - value given is approximate)

Name	Value
av	0*
MaxAllocSlot	20*
LargeReturnSlot	Determined by software
sd	22*
gft	46*
stkmax	8*
psv	1
localbase	4*
globalbase	10*

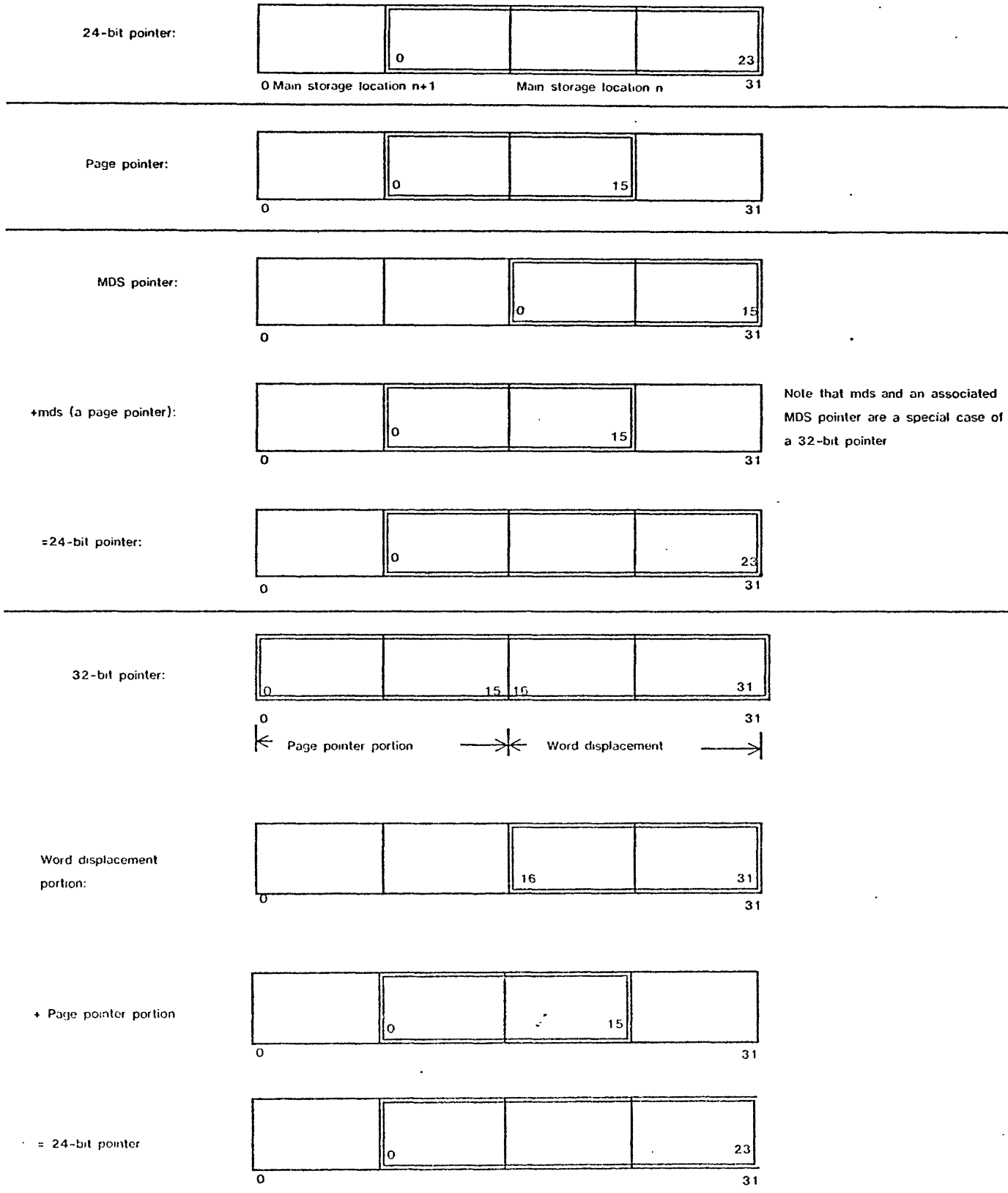


Figure 1: Pointer Formats

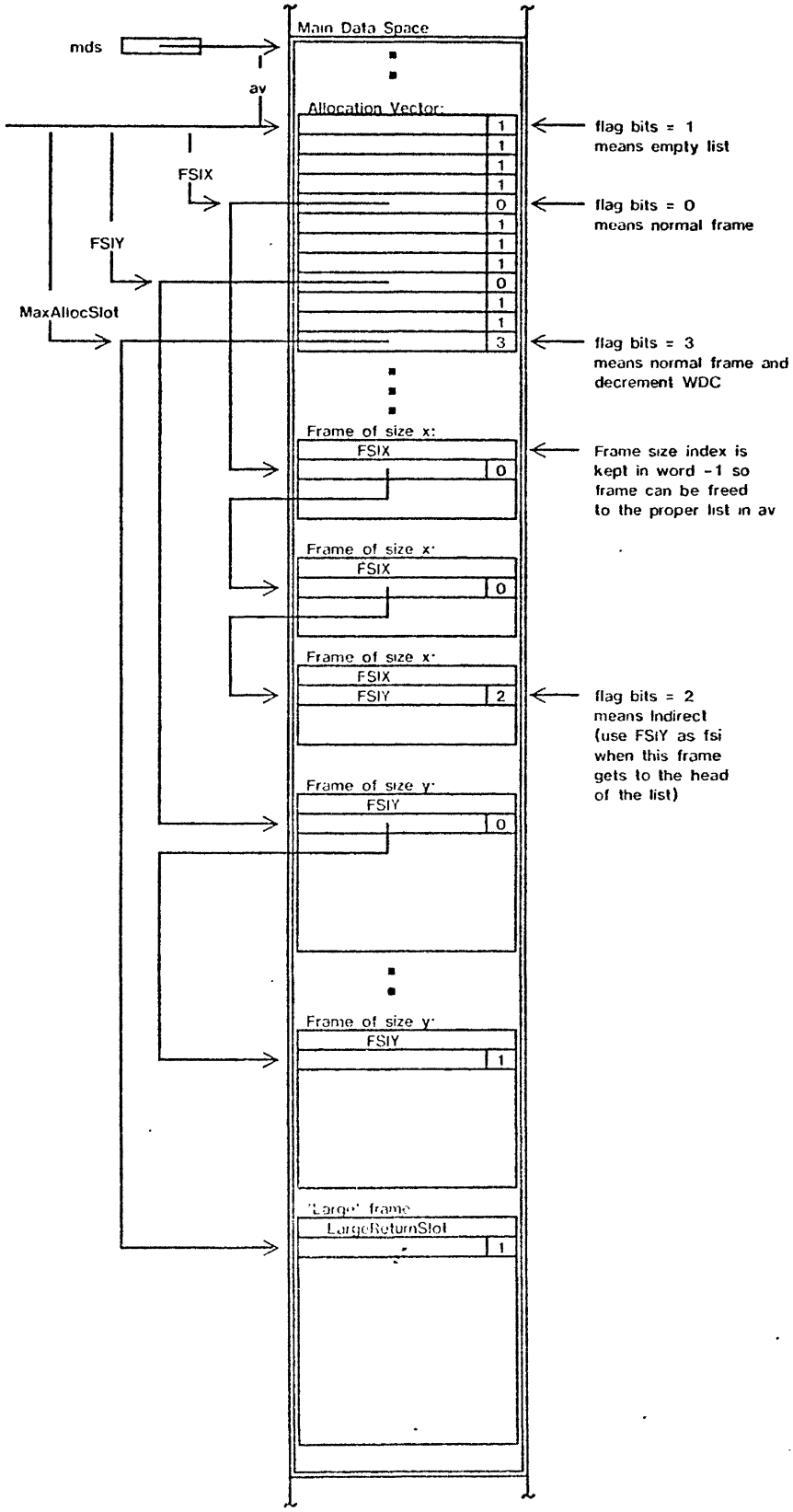
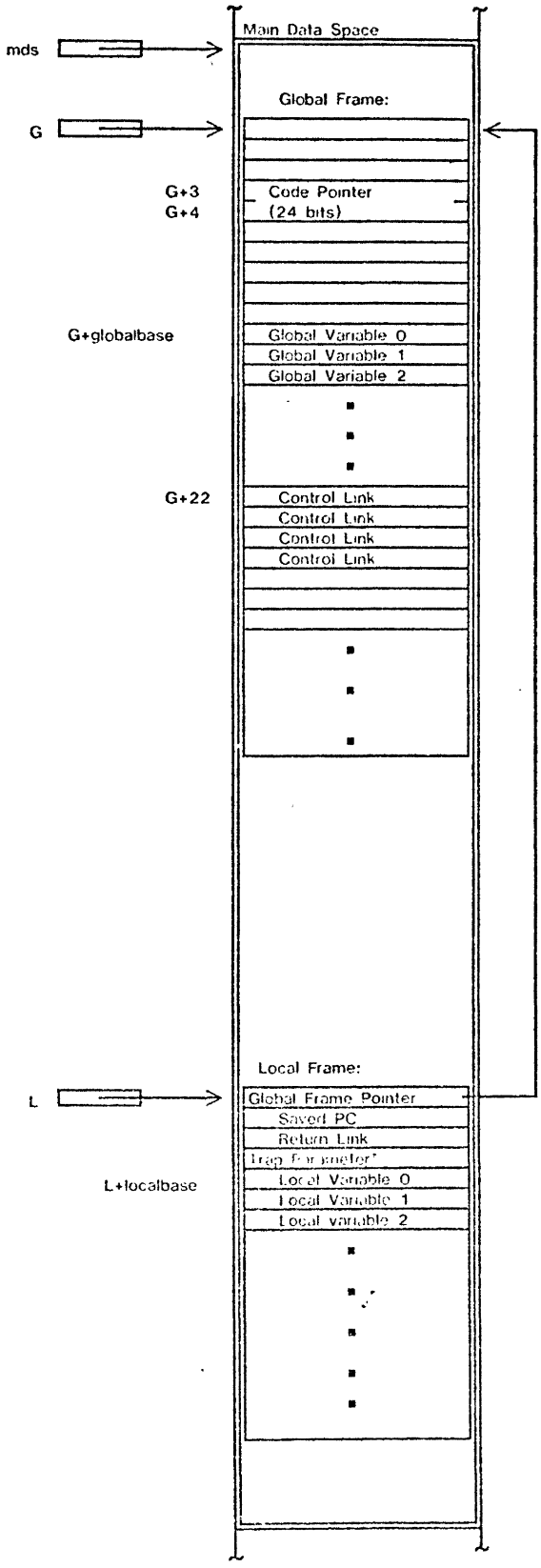


Figure 2b: Allocation Vector and Frames



* Trap Handlers only

Figure 2c: Local and Global Frames

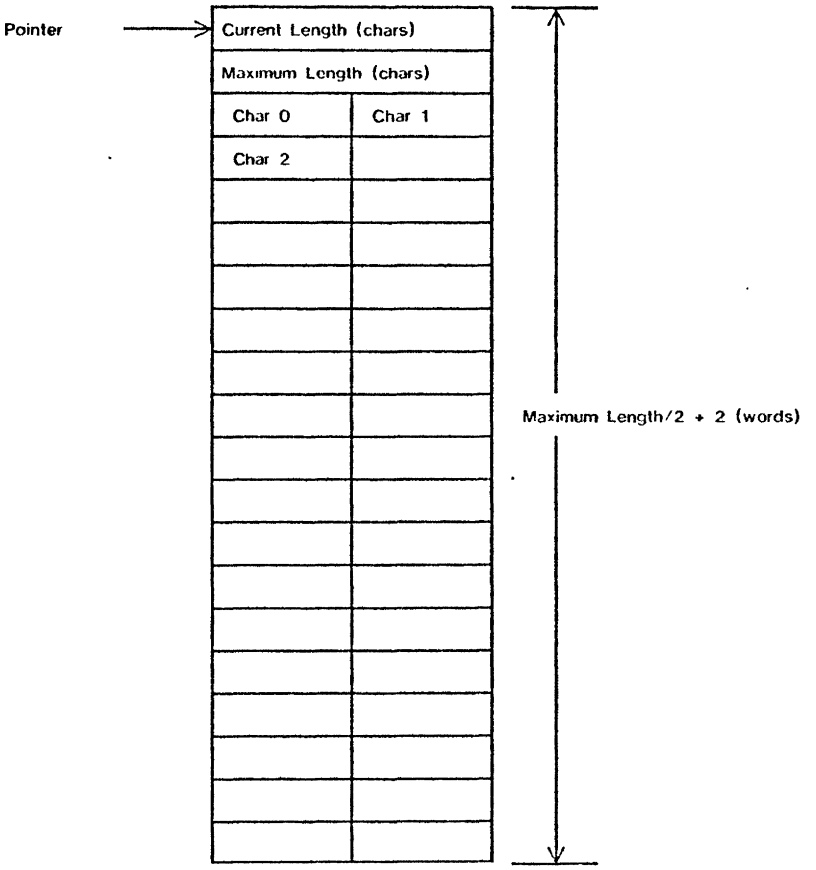


Figure 3: A Mesa String

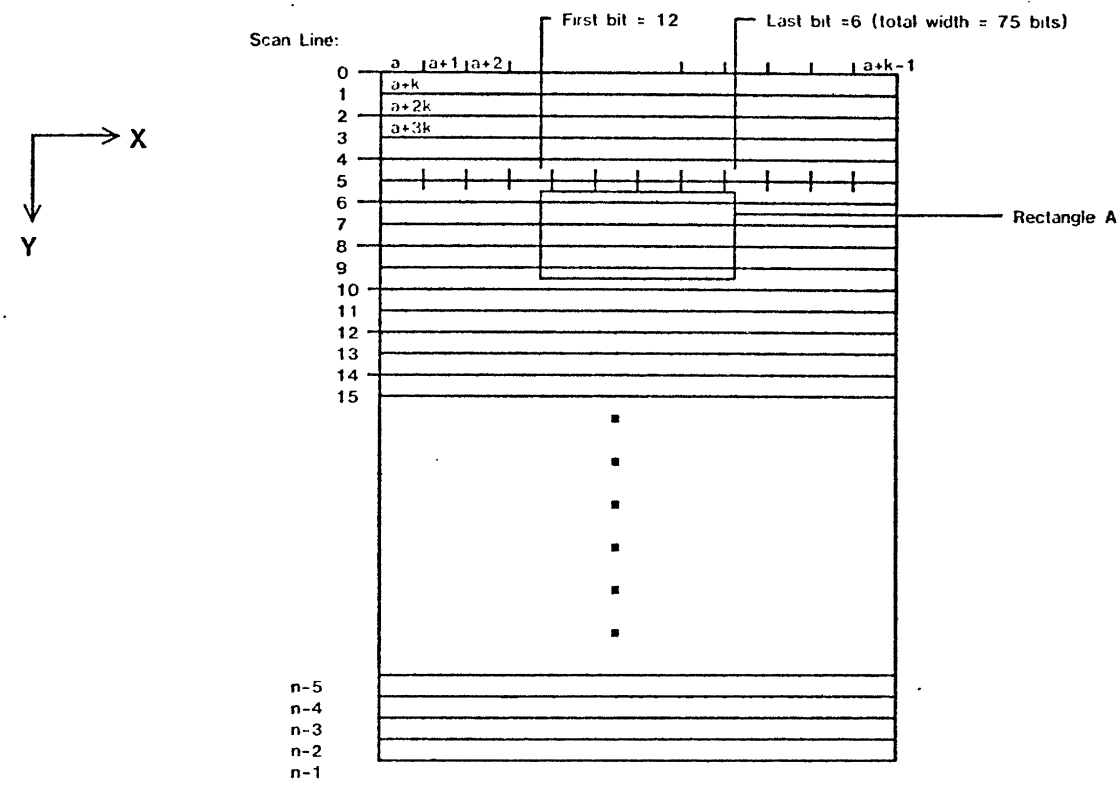


Figure 4a: Display Format

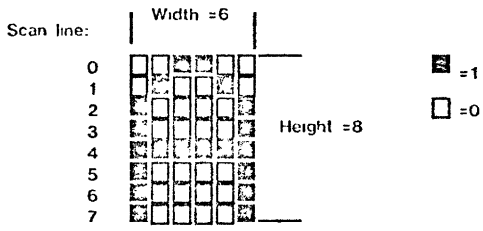


Figure 4b: A Character Bitmap

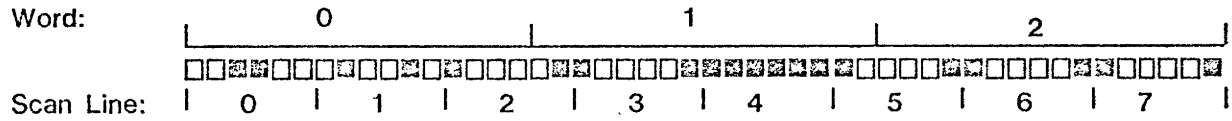


Figure 4c: Character Representation in Storage

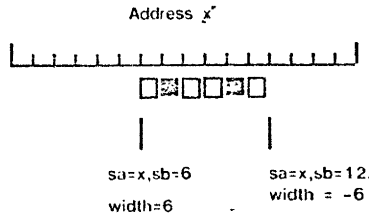
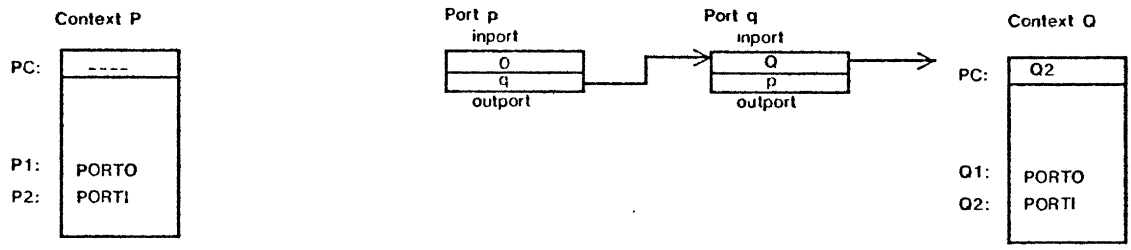
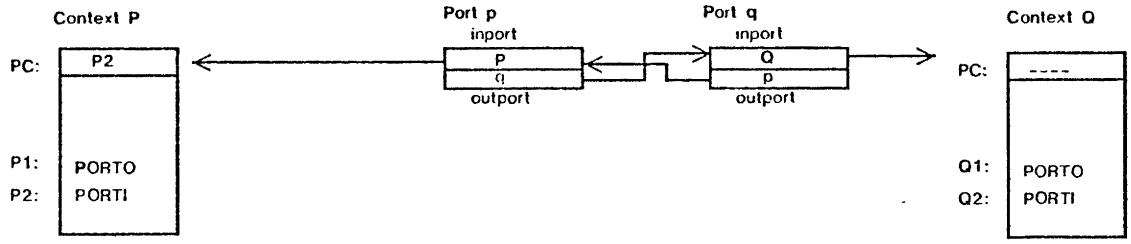


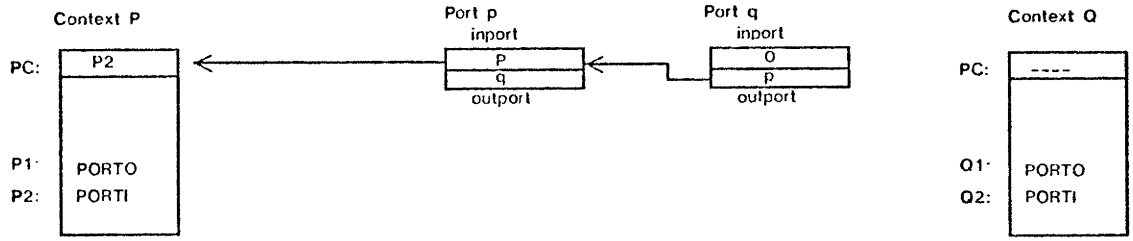
Figure 4d: Interpretation of Negative Item Widths



(a) Q has transferred to P via the PORTO at Q1.
Control is in P, but not yet at P1
Q is pending on q.

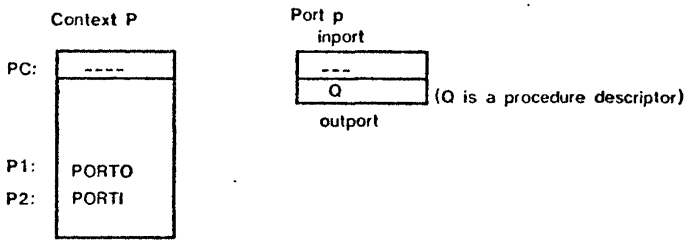


(b) P has executed the PORTO at P1, and control has passed to Q.
Q has not yet executed the PORTI at Q2.
P is pending on p.

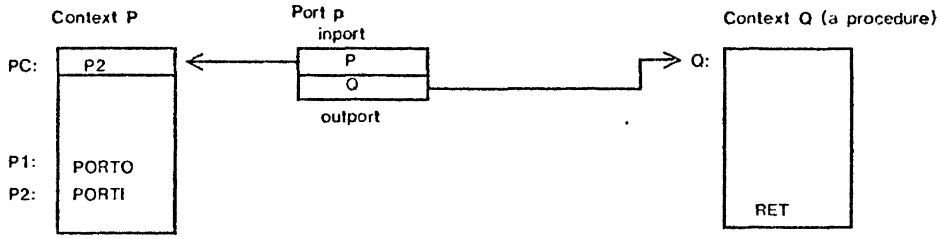


(c) Q has executed the PORTI at Q2, saving the link (an indirect link)
in q outport. Any attempt to transfer to Q through q will fault, since q inport = 0
P is pending on p.

Figure 5a: Port to Port Control Discipline



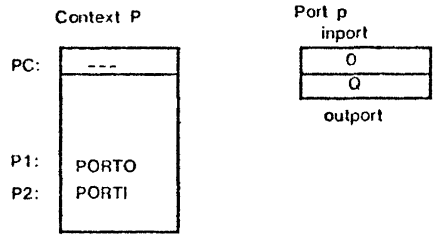
(a) Control is in P, before P1



(b) P has executed the PORTO at P1, and control is in Q

The link saved in Q's frame is p (an indirect link)

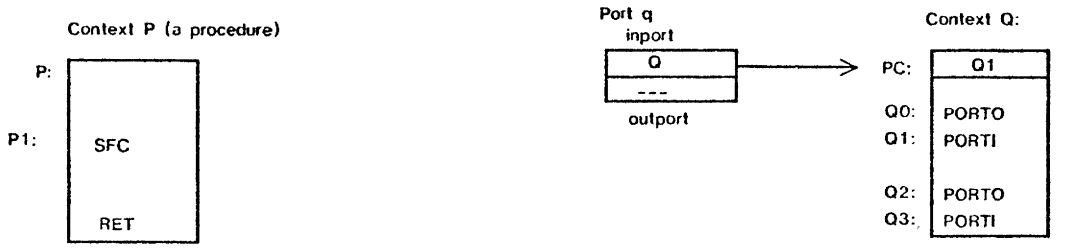
P is pending on p



(c) Q has executed the RET, which does XFER[p,0]

P has executed the PORTI at P2, which has cleared p.inport.

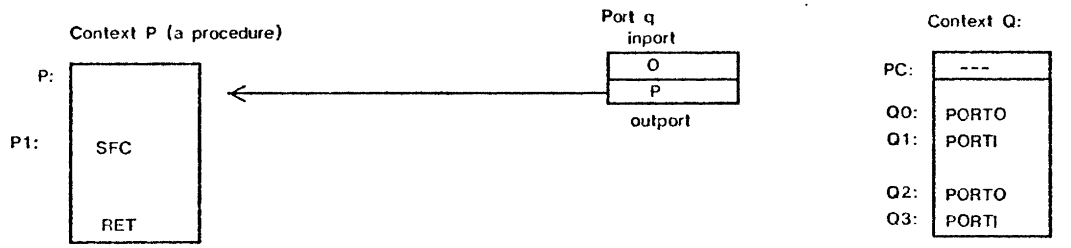
Figure 5b: Port to Procedure Control Discipline



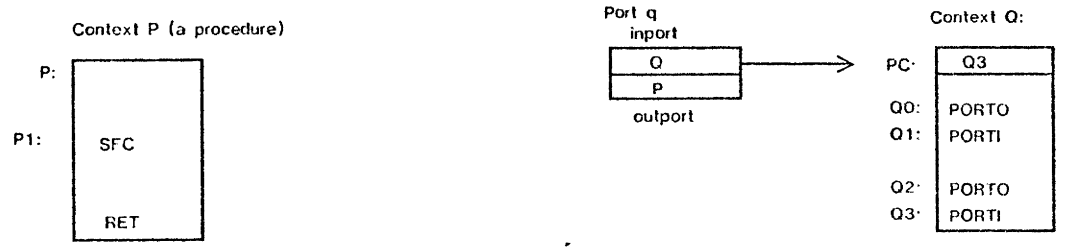
(a) Control is in P, about to execute the SFC at P1.

The stack contains an indirect control link pointing to q.

Q is pending on q.



(b) Control has passed to Q, and the PORTI at Q1 has been executed.
q.outport contains P (a frame pointer)



(c) Q has executed the PORTO at Q2, and control has returned to P.
Q is again pending on q.

Figure 5c: Procedure to Port Control Discipline