XEROX

Mesa Processor Principles of Operation

Version 4.0
May 1985

Office Systems Division
2400 Geng Road
Palo Alto, California 94303

Table of contents

Illustrations

1 Introduction

This document defines the architecture of the Mesa processor. It specifies the processor's virtual memory structure, its instruction interpreter, and the Mesa instruction set. (The organization of the input/output system is described separately.) The Mesa processor is part of a larger plan for the development and construction of integrated Office Information Systems.

The *Principles of Operation* does not discuss the implementation of any particular Mesa processor, of which there are several models, differing primarily in underlying technology, configuration, and performance. It does specify the overall design that must be followed to ensure software compatibility at the instruction set level. The architecture allows common software systems to be constructed that operate on all versions of the processor; it also allows reimplementation of the processor when it is technically or economically advantageous to do so.

The *Principles of Operation*, often called the *PrincOps* (pronounced *prince ops*), is composed of nine additional chapters, an Appendix, a list of references, and four indices.

DATA TYPES Basic data types: UNSPECIFIED, BIT, NIBBLE, BYTE; basic logical and arithmetic operators; numeric types: CARDINAL, INTEGER, REAL; long types; pointer types; type conversion.

MEMORY ORGANIZATION Virtual memory; memory mapping; memory map instructions; major data structures; main data spaces; control transfer data structures; local and global frames; processor registers; evaluation stack; register instructions.

INSTRUCTION INTERPRETER Instruction formats; instruction fetch; address calculation; instruction execution; opcode dispatch; exceptions: traps, faults, interrupts; initial state.

STACK INSTRUCTIONS Stack primitives; check instructions; unary operations; logical operations; arithmetic operations; comparison operations; floating point operations.

JUMP INSTRUCTIONS Unconditional jumps; equality jumps; signed inequality jumps; unsigned inequality jumps; indexed jumps.

ASSIGNMENT INSTRUCTIONS Immediate instructions; frame instructions: local access, global access; pointer instructions: direct, indirect; string instructions; field instructions.

BLOCK TRANSFERS Word boundary block transfers, checksum; byte boundary block transfers; bit boundary block transfers: bit transfer utilities, bit block transfer, text block transfer.

CONTROL TRANSFERS Control links: frame links, indirect links, procedure descriptors; frame allocation; control transfer primitive (XFER); control transfer instructions: local calls, external calls, nested calls, returns, coroutine transfers; traps; breakpoints; xfer traps.

PROCESSES Process data structures; process instructions; process queue management; scheduling; faults; interrupts; timeouts.

APPENDIX Values of constants; register indexes; fixed memory locations; fault queue indexes; system data indexes; opcode assignments.

Section 1.1 lists the major technical characteristics of the processor; §1.2 defines some frequently used and often confusing terminology; §1.3 explains the coding conventions used in describing the operation of the processor.

1.1 Technical Summary

All Mesa processors have the following characteristics that distinguish them from other computers:

1.1.1 High-Level Languages

The Mesa architecture is designed to efficiently execute high-level languages in the style of Algol, Mesa, and Pascal. That is, constructs of the programming language such as modules, procedures, and processes all have concrete representations in the processor and main memory, and the instruction set includes opcodes to implement these language constructs efficiently (for example, in procedure call and return). The processor does not "directly execute" any particular high-level programming language, however.

1.1.2 Compact Program Representation

The Mesa instruction set is designed primarily for a compact, dense representation of programs. Instructions are variable in length. The most frequently used operations and operands are encoded in a single-byte opcode; less frequently used combinations are encoded in two bytes, and so on. The instructions themselves are chosen based on their frequency of use, and this design principle leads to an asymmetrical instruction set. For example, there are eight different instructions that can be used to store variables into local frames in memory, but only four that load local variables onto the stack from memory; this occurs because typical programs perform many more stores than loads. In some cases, a particular operation is performed so infrequently that it is not provided as an instruction, and must therefore be programmed in software (for example, a quad word add). There are other cases in which an instruction is provided for an infrequently used operation because the function performed is required at the instruction set level for technical or efficiency reasons (such as for disable interrupts or checksum calculations).

1.1.3 Compact Data Representation

The instruction set of the processor includes a wide variety of operations for accessing partial and multiword fields of the memory's basic information unit, the sixteen-bit word. Except for system data structures defined by the architecture, there are no alignment restrictions on the allocation of variables, and data structures are generally assumed to be tightly packed in memory.

1.1.4 Read Only Relocatable Code

The instructions associated with a program are organized separately from the data it declares in a structure called a *code segment.* All *code segments* are entirely read-only. They are also relocatable without modification, since no information in a code segment depends on its location in virtual memory.

1.1.5 Stack Machine

The Mesa processor is a stack machine; it has no general-purpose registers. (It does include special-purpose registers for maintaining processor status and state.) The evaluation stack is used as the destination for load instructions, the source for store instructions, and as both the source and the destination for arithmetic instructions. It is also used for parameter passing. The primary motivation for a stack is not to simplify code-generation, but to achieve compact program representation. Since the stack is assumed as the source and destination of one or more operands, specifying operand locations requires no bits in the instruction – they are implied by the opcode.

1.1.6 Control Transfers

The architecture is designed to support modular programming. It therefore suitably optimizes the transfers of control between modules. The Mesa processor implements all transfers with a single primitive called XFER, which is a generalization of the notion of a procedure or subroutine call. All of the standard procedure-calling conventions (such as call by value, call by reference (result), and call by name) and all transfers of control between contexts (procedure call and return, nested procedure calls, coroutine transfers, traps, and process switches) are implemented using the XFER primitive. To support arbitrary control-transfer disciplines, activation records (called *frames*) are allocated by XFER from a heap rather than a stack; this method also allows the heap to be shared by several processes.

1.1.7 Process Mechanism

The architecture is designed for applications that expect a large amount of concurrent activity. The Mesa processor provides for the simultaneous execution of up to one thousand asynchronous, preemptable processes on a single processor. The process mechanism implements monitors and condition variables to control the synchronization and mutual exclusion of processes along with the sharing of resources and information among them. Scheduling is event-driven, rather than time-sliced. Interrupts, timeouts, and communication with I/O devices are also supported by the process mechanism. Support for multiple processors is under development.

1.1.8 Virtual Memory

The Mesa processor provides a single large, uniformly-addressed virtual memory, shared by all processes. The memory is addressed linearly as an array of 2^{32} sixteen bit words, and, for mapping purposes, is further organized as an array of 2^{24} pages of 256 words each; it has no other programmer-visible substructure. Each page can be individually write-protected, and the processor records the fact that a page has been written into or referenced.

1.1.9 Protection

The architecture is designed for the execution of cooperating, not competing, processes. There is no protection mechanism (other than the write-protected page) to limit the sharing of resources among processes. There is no "supervisor mode" nor "privileged" instructions.

1.2 Terminology

In this section, the terms *architecture*, *processor*, and *programmer* are defined. Several stylistic conventions used throughout the *PrincOps* are also explained in this and the following sections.

Note: Paragraphs beginning with the word "**Note**" contain comments intended for the reader of this manual. They generally point out or explain an important convention concerning the document or the code contained within it; they do not describe the Mesa processor itself.

1.2.1 Architecture

As used in the *PrincOps*, the term *architecture* refers to the characteristics of the processor, as seen by a programmer writing executable instructions for the machine. The term does not refer to the way the processor is integrated with other hardware and software components to form a computer system. Nor does it refer to the way hardware and firmware components might be integrated to form any specific implementation of the processor.

Note: Paragraphs beginning with the phrase "**Design Note**" contain important points about the design of the processor architecture. They will be of interest both to implementors and to programmers.

1.2.2 Processor

The term *processor* actually refers to a particular implementation of the Mesa processor (or more likely, to *all* such implementations). A processor is the collection of hardware and microcode that behaves in a manner consistent with the description contained in this manual. Each Mesa processor must achieve the same result as the code appearing here, although the implementation, and in many cases even the algorithm, may be different.

For example, as mentioned above, every Mesa processor includes an evaluation stack whose behavior is described in §3.3.2. There are several ways to implement a stack, varying in such details as where the stack pointer points (to the top element? to the next available entry?) and how the checks for underflow and overflow are made. The processor may use any implementation so long as the stack pointer *as seen by the programmer* always points to the next available entry on the stack, as described in §3.

Note: Paragraphs beginning with the phrase "**Implementation Note**" describe implementation techniques. Generally, these suggest efficiency improvements or point out restrictions in addition to those contained in the code. They are directed primarily at the microcoder and the hardware designer who will implement the processor.

1.2.3 Programmer

The term *programmer* refers to the person writing instructions to be executed by the processor. Because the Mesa processor is designed for use in conjunction with high-level languages, the programmers in question are usually authors of compiler code generators or low-level operating system functions. A "typical" applications programmer sees the processor not at the instruction set level, but rather at the programming language level.

Note: Paragraphs beginning with the phrase "**Programming Note**" are intended primarily for the programmer. Techniques for exploiting a feature of the processor are described in such paragraphs. Also, they often begin with the phrase "It is the programmer's responsibility to ensure that ..." or "It is illegal for a program to ...". As discussed above, these notes are often directed to the authors of the compiler or the operating system, rather than to a "typical" programmer.

Note: The statements contained in programming notes concerning the legality of a program and the conditions that it (and the programmer) must satisfy all have the same intention; they mean that if the condition is violated, the program may produce undefined results, and further, that the results obtained during execution may be different on different implementations of the processor.

1.3 Conventions

The PrincOps describes the processor using Mesa itself; the data structures and algorithms are written in the Mesa language. Familiarity with the Mesa Language Manual [2] is assumed. The term *code* is used to refer to the Mesa source code in this document that describes the behavior of the processor; it does not refer to the programs that execute on the machine. Likewise, the term *routine* is used to refer to a part of the code (usually a procedure), whereas *program* and *procedure* refer to the instructions being executed by the processor. These distinctions must constantly be kept in mind.

Coding conventions beyond standard Mesa style are described in this section. The code in this document makes several assumptions that are not generally known to a Mesa programmer; these assumptions are outlined below. Certain language features normally available (notably those involving pointer dereferencing, variant records, coroutines, and processes) are not allowed in PrincOps code, primarily to simplify and clarify the descriptions. These omissions are also identified.

1.3.1 Type Checking

One of the primary reasons for describing the Mesa processor in Mesa is to bring type-checking to bear on the specification; in particular, all of the code contained in this document has been compiled by the Mesa compiler to verify its syntactic and type correctness. However, because main memory is inherently typeless, it has not been possible to utilize the language's type system fully, while keeping the description simple and short. In particular, all references to main memory yield values of (some variant of) type UNSPECIFIED, and there are many pointers to UNSPECIFIED in the code. When possible, the values are assigned to temporary variables to make their types clear, but the temporaries may not have a concrete realization in the processor.

1.3.2 Type Representation

The code assumes that the underlying representation of the basic types is binary, that unsigned numeric quantities are represented in true binary notation, and that signed quantities use two's-complement representation.

1.3.3 Subrange Types

Except for INTEGER, all subranges used in the code have a lower bound of zero. As a result, there is no automatic biasing of subranges. Subranges occupy only the number of bits required to store their range of values when represented as binary numbers. With one exception (the stack pointer; see §3.3.2), subranges occupy exactly an integral number of bits; that is, the upper bound of all subranges is one less than a power of two.

1.3.4 Enumerated Types

Enumerated types are represented by assigning (binary) zero to the first value, one to the second value, two to the third, and so on; all enumerated types in this document are declared MACHINE DEPENDENT. As with subranges, enumerated types occupy only the number of bits required to represent all of their values. All enumerations occupy an integral number of bits (that is, the number of elements in each enumeration is a power of two).

1.3.5 Pointers

Both virtual- and real-memory addresses are represented by LONG POINTER types; short POINTERs are converted to long pointers before dereferencing. The dereferencing operator (↑) is used to follow a pointer and obtain the word(s) it addresses in real memory. This operator is applied only to LONG POINTERs that have been converted from virtual into real addresses.

1.3.6 Arrays and Records

Arrays and records are used to define the structures used by the processor. Their semantics are as described in the Mesa Language Manual[1], except that their components are restricted to the types defined in §2. Two additional conventions are used in the code.

First, each array element is at least one word long and is always a multiple of the word size. No additional packing is specified; that is, there are no packed arrays that would lead to hidden addressing calculations below the level of a word address. In addition, the fields of a record always account for all of the bits in the words occupied by the structure, whether or not they are used. No extra bits remain free. All records are declared MACHINE DEPENDENT so that this will be checked by the compiler.

Second, two distinguished field names are used in defining record structures. The name reserved, usually accompanied by an initial value, indicates that neither the processor nor the programmer uses the field. If an initial value is given, the processor may assume that the field always has that value, and the programmer must ensure its integrity. On the other hand, the name available indicates that the field may be used by the programmer, and therefore must be left undisturbed by the processor.

1.3.7 Type Conversion

Conversion between types is modeled using the Mesa operators INTEGER, CARDINAL, and LONG, together with a number of built-in routines. Details of the conversions between specific types are contained in §2.4.

1.3.8 Built-in Routines

The code assumes the implementation of a number of basic routines for which a more detailed description is not given; for example, the And and Or routines are used as implementations of the standard logical operators, and are not described further. A complete list of these routines is contained in §2.

1.3.9 Control Flow

Although the code is written as a collection of routines, there are several cases where procedure calls and returns can be replaced by jumps in the implementation. For example, simple opcodes are described as single routines, but the calls that invoke them could be replaced by jumps (or more likely, dispatches) from the main loop of the instruction interpreter. Their returns could be replaced by jumps back to opcode fetch. Similarly, certain utility routines are always invoked at the logical end of opcode processing, and therefore need not actually be called as procedures. A jump to their beginning addresses could be used instead. These routines could return by jumping back to the location in the interpreter to which the opcode would have returned, had a true procedure call been used.

1.3.10 Signals and Errors

Signals are used for global transfers of control across one or more procedure calls. There is only one signal declared in the code: Abort (§4.1). It is used to describe exception processing. This signal is raised in exactly two places: by the trap and the fault routines. It is caught by the main loop of the instruction interpreter, and is never restarted.

An unnamed ERROR is used to indicate conditions that must be established by the programmer but need not be verified by the processor (for example, the Setup routine in §8.2.2.3).

On some occasions, these two conventions are used together; a few routines catch Abort and raise ERROR (for example, the SaveProcess and LoadProcess routines in §10.4.2.1). This combination indicates that it is illegal for the routine to suffer a trap or fault (as when all the memory it references must be resident).

Implementation Note: None of the statements in the code that contain an unnamed ERROR need be included in an implementation of the processor. If a program could cause the code to generate an ERROR, the program is illegal. It may produce undefined results.

1.3.11 Instruction Descriptions

Each instruction is defined as a separate routine. The name of the routine is the same as the mnemonic for the opcode. Complex instructions are broken down into multiple routines, which are assumed to be nested in their parent opcode routine unless they are shared by more than one instruction. The generic form of an instruction description is illustrated below.

OPCODE Name

Details not covered in the summary of the instruction class appear here.

```
OPCODE: PROCEDURE =
  BEGIN
  ...
  END;
```

Fine points and notes appear here.

1.4 Indices

There are four indexes in the *PrincOps*: the Primary Index, Mesa Code Index, which is an index of the types, constants, and routines contained in the code, and the Opcode Names Index and Opcode Mnemonics Index, which are indexes of the opcode routines organized by instruction names and mnemonics. In these indexes, bold face page numbers indicate where the primary, defining information can be found; plain page numbers designate further references.

2 Data Types

This chapter describes the characteristics of the basic data types used by the Mesa processor. The descriptions contained here give only the essential properties of the types. The legal operations on each type are explicitly enumerated elsewhere (and are more restrictive than allowed by the Mesa language).

Note: The routines appearing in this chapter are solely defined for use by the code contained in subsequent chapters. These routines need not be implemented by the processor (although several will be), since they are not available directly to the programmer.

For example, the following routine is useful in describing the processor but is not provided as an instruction:

```
Log: PROCEDURE [count: CARDINAL] RETURNS [CARDINAL];
```

This routine returns the number of bits required to store the number of values given by its argument, assuming an unsigned, unbiased binary representation.

2.1 Basic Data Types

The primary unit of storage is the sixteen-bit word.

```
WordSize: CARDINAL = 16;
```

The most significant bit of a word is numbered zero; the least significant bit is numbered fifteen:

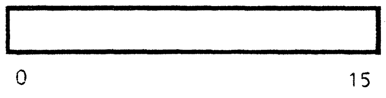


Figure 2.1 Sixteen-Bit Word

Variables of most common types occupy a single word.

2.1.1. Unspecified

Unless they have other specific properties (described in the following sections), words are declared as type UNSPECIFIED, a type that is essentially just a bit string. Only a subset of the basic operators (§2.1.3) apply to this type; in particular, arithmetic operators are not used with UNSPECIFIEDS.

```
BLOCK: TYPE = ARRAY [0..0) OF UNSPECIFIED;
```

A BLOCK is used as a placeholder to represent a region of storage of indeterminate size.

2.1.2 Bit, Nibble, Byte

The following types, which occupy one, four, and eight bits respectively, are used to represent the substructure of a word:

```
BIT: TYPE = [0..2);
NIBBLE: TYPE = [0..16);
BYTE: TYPE = [0..256);
```

A byte is often interpreted as two adjacent nibbles; likewise, a word can be interpreted as two adjacent bytes. The structures NibblePair and BytePair reflect these interpretations:

```
NibblePair: TYPE = MACHINE DEPENDENT RECORD [left (0: 0..3), right (0: 4..7): NIBBLE];

BytePair: TYPE = MACHINE DEPENDENT RECORD [left (0: 0..7), right (0: 8..15): BYTE];
```

Two routines are used to extract the bytes of a word:

```
HighByte: PROCEDURE [u: UNSPECIFIED] RETURNS [BYTE] =
  BEGIN
  pair: BytePair = u;
  RETURN[pair.left];
  END;

LowByte: PROCEDURE [u: UNSPECIFIED] RETURNS [BYTE] =
  BEGIN
  pair: BytePair = u;
  RETURN[pair.right];
  END;
```

The architecture also defines several double word (thirty-two bit) types; see §2.3.

2.1.3 Basic Operators

Fundamental operators are defined for all types: they are assignment (←) and comparison for equality (=) and inequality (#). In addition, the processor implements the primitive operations found in most ALUs; these include the logical operations Not, And, Or, Xor, and Shift, as well as the arithmetic operators negation (-), addition (+), subtraction (-), and ArithShift. Note that comparison (for other than equality) is not considered a basic operator, since its result depends on whether the operands are signed or unsigned (§2.2).

Such operations can not be performed on UNSPECIFIEDs, which are neither signed nor unsigned.

2.1.3.1 Basic Logical Operators

The following standard logical operations on bit strings are primitive:

```
Not: PROCEDURE [UNSPECIFIED] RETURNS [UNSPECIFIED];
Odd: PROCEDURE [UNSPECIFIED] RETURNS [BOOLEAN];
```

Odd returns TRUE if the least significant bit of its argument is one, and FALSE otherwise.

```
And: PROCEDURE [UNSPECIFIED, UNSPECIFIED] RETURNS [UNSPECIFIED];
Or: PROCEDURE [UNSPECIFIED, UNSPECIFIED] RETURNS [UNSPECIFIED];
Xor: PROCEDURE [UNSPECIFIED, UNSPECIFIED] RETURNS [UNSPECIFIED];

Shift: PROCEDURE [data: UNSPECIFIED, count: INTEGER] RETURNS [UNSPECIFIED];
Rotate: PROCEDURE [data: UNSPECIFIED, count: INTEGER] RETURNS [UNSPECIFIED];
```

In Shift, data is shifted the number of bits specified by count; the shift is to the left if count is positive and to the right if it is negative. If count is zero, the result is the value of data unchanged; if the absolute value of count is greater than fifteen, the result of the operation is zero. In all cases, zeros are supplied to vacated bit positions. In Rotate, data is rotated the number of bits given by count; it is left-rotated if count is positive and right-rotated if negative. If count is zero, data is returned unchanged.

2.1.3.2 Basic Arithmetic Operators

The basic arithmetic operators, negation, addition, and subtraction, assume a two's-complement binary representation. If overflow is ignored, the result can be considered either signed or unsigned (see also the section on arithmetic types below).

The following shift routine is also used in the code, but is not provided as an instruction:

```
ArithShift: PROCEDURE [data: INTEGER, count: INTEGER] RETURNS [INTEGER];
```

This operation is similar to logical shift, except that when shifting right, a copy of bit zero (the sign bit) is shifted into the left of data; when shifting left, bit zero is undisturbed.

2.2 Numeric Types

The numeric types include signed and unsigned fixed point numbers. There is also a provision for a floating point representation of real numbers (see §2.2.3). The operations on numeric types include the fundamental operators (§2.1.3), the basic arithmetic operators (§2.1.3.2), and the comparison operators (**<**, **<=**, **>=**, and **>**), plus multiplication (*), division (/), and remainder (MOD).

2.2.1 Cardinal

Unsigned numbers are of type CARDINAL and occupy a single word. The values zero through 65,535 are represented using true binary notation. All operations performed on cardinals produce unsigned results in the range given above.

2.2.2 Integer

Signed numbers are of type INTEGER and occupy a single word. The values -32,768 through 32,767 are represented using two's-complement binary notation. All operations performed on INTEGERs produce signed results according to the rules of algebra. For multiplication, the product is negative if exactly one of the multiplicand or the multiplier is negative and the other operand is not zero.

Multiplicand	Multiplier	Product
positive	positive	positive
positive	negative	negative
negative	positive	negative
negative	negative	positive

For division and remainder, the dividend and the remainder have the same sign; that is, the results satisfy the following equation: dividend = quotient * divisor + remainder.

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

2.2.3 Real

Except that they occupy two and four words respectively, the formats of REAL and LONG REAL types are not defined by the architecture.

Design Note: Adoption of the proposed IEEE floating point standard [2] is currently in progress.

2.3 Long and Pointer Types

The processor implements several long (double-word) types, as well as both short and long pointer types. The representations of these types are defined as extensions of the types described above.

2.3.1 Long Types

The architecture supports double-word configurations of the types UNSPECIFIED, CARDINAL, INTEGER, and POINTER (see below). These types occupy thirty-two bits, wherein the most significant bit of a double word is numbered zero, and the least significant bit is numbered thirty-one.

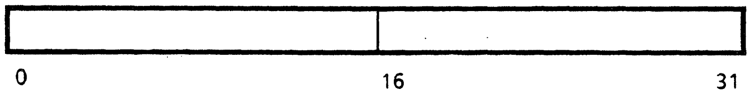


Figure 2.2 Thirty-two Bit Double Word

When these types are stored in memory, the low-order (least significant) sixteen bits occupy the first memory word (at the lower numbered address), and the high-order (most significant) sixteen bits occupy the second memory word (at the higher memory address).

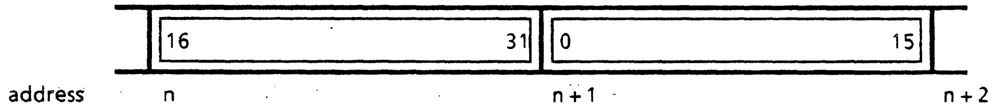


Figure 2.3 Double Word in Memory

Design Note: This inconsistent convention is solely for the convenience and efficiency of operations that use the evaluation stack (§3.3.2).

The following constructs are used to extract the subcomponents of an arbitrary long type:

```
Long: TYPE = MACHINE DEPENDENT RECORD [
  low (0), high (1): UNSPECIFIED];

HighHalf: PROCEDURE [u: LONG UNSPECIFIED] RETURNS [UNSPECIFIED] =
  BEGIN
  long: Long = LOOPHOLE[u];
  RETURN[long.high];
  END;

LowHalf: PROCEDURE [u: LONG UNSPECIFIED] RETURNS [UNSPECIFIED] =
  BEGIN
  long: Long = LOOPHOLE[u];
  RETURN[long.low];
  END;
```

All of the operations applicable to UNSPECIFIED, CARDINAL, and INTEGER types are also valid for their long counterparts, with the same semantics and restrictions, except for the range of the results. In addition, shifting operations are defined for long types:

```
LongShift: PROCEDURE [data: LONG UNSPECIFIED, count: INTEGER]
  RETURNS [LONG UNSPECIFIED];

LongArithShift: PROCEDURE [data: LONG INTEGER, count: INTEGER]
  RETURNS [LONG INTEGER];
```

2.3.2 Pointer Types

Values of type POINTER and LONG POINTER are memory addresses occupying single and double words, respectively. In addition to the fundamental operations, all of the basic logical operators and the basic arithmetic operators, addition and subtraction, can be applied to pointers. For arithmetic purposes, pointers are always unsigned.

Design Note: Like all long types, the components of LONG POINTERs appear in memory with the least significant word occupying the lower-numbered memory address (§2.3.1).

2.4 Type Conversion

This section defines the conversions performed between the types defined above. Except for the operators already defined (HighByte, LowByte, etc.), and for the cases involving the numeric and pointer types described below, conversions between operands are performed by the standard assignment operator (←), which means "copy the bits".

2.4.1 Assignment

In the statement left ← right, if either operand is more than sixteen- bits wide, both must be the same width. Otherwise, if right is shorter than left, sufficient high-order zeros are supplied. In general, when operands smaller than a word appear in an expression, they are considered to be embedded in a word by zero extending (*not* sign extending), just as in the expression "int + 6", the constant is assumed to be extended as necessary. If more than sixteen bits are required, the lengthening of an operand is always made explicit (using LONG, defined below). Likewise, if bits other than zeros are required, a built-in operation is used. For example:

```
SignExtend: PROCEDURE [z: BYTE] RETURNS [INTEGER] =
  BEGIN
  RETURN[IF z IN [0..177B] THEN z ELSE z-400B];
  END;
```

SignExtend defines the conversion of a signed byte to a sixteen-bit integer.

The shortening of an operand is always indicated explicitly by using LowByte, LowHalf, or some other explicitly coded function (see, for example, the ReadField and WriteField routines defined in §7.5).

2.4.2 Signed/Unsigned Conversions

Conversions between signed and unsigned numbers of the same length are performed using the operators CARDINAL and INTEGER. Given i: INTEGER and c: CARDINAL, the following examples illustrate their usage:

```
i ← INTEGER[c];        -- check c < = LAST[INTEGER]
c ← CARDINAL[i];       -- check i > = FIRST[CARDINAL]
```

The INTEGER conversion implies a check that the cardinal is less than 32,768 (yielding an ERROR if it fails); the CARDINAL conversion implies a check that the integer is non-negative. With appropriate change in range, the same conversions also apply to LONG CARDINAL and LONG INTEGER.

2.4.3 Short/Long Conversions

Conversions from short to long are performed using the LONG operator. Given i: INTEGER, li: LONG INTEGER, c: CARDINAL, and lc: LONG CARDINAL, the following table defines the conversion rules:

```
li ← LONG[i];        -- sign extend
li ← LONG[c];        -- supply high-order zeros
lc ← LONG[i];        -- check non-negative
lc ← LONG[c];        -- supply high-order zeros
```

The LONG operator applied to an UNSPECIFIED always produces a LONG UNSPECIFIED by prefixing high-order zeros.

Conversions from long to short values are performed using the built-in routines defined above (for example, LowByte or LowHalf). These operations do not check for loss of significant bits. If a check is required, it appears explicitly in the code.

2.4.4 Pointer Conversions

Conversions of constants to pointers are performed using a **LOOPHOLE**. The mapping of virtual to real memory addresses is the subject of §3.1.1; conversions from short to long pointers involve special addressing considerations described in §3.2.1.

3 Memory Organization

This chapter describes the memory structures of the Mesa processor. It discusses the virtual memory, distinguished regions of the virtual memory called *Main Data Spaces*, and the programmer-accessible memories of the processor. This chapter also identifies most of the data structures residing in these memories used by the processor. The chapters on control transfers (§9) and the process mechanism (§10) define other structures in detail.

3.1 Virtual Memory

All Mesa processors implement a large virtual address space. Virtual memory is organized as a single uniform array of words shared by all processes, addressed by thirty-two bit *virtual addresses*. A virtual address is mapped into a real address before an actual fetch or store operation occurs. Virtual addresses are represented by either *long* or *short pointers*.

For mapping purposes, virtual and real memory are further structured as a uniform array of *pages*. A page is a contiguous array of 256 words whose address is a multiple of the page size. (Therefore, it lies on a *page boundary*.)

```
    PageSize: CARDINAL = 256;
    PageNumber: TYPE = LONG CARDINAL; -- [0..2^24)
    Page: TYPE = ARRAY [0..PageSize) OF UNSPECIFIED;
```

Note: Although a PageNumber is actually a subrange of LONG CARDINAL, the current version of the Mesa language does not support this feature.

Both virtual and real memory consist of up to 2^{24} pages (2^{32} sixteen-bit words).

```
    RealPageNumber: TYPE = PageNumber;
    VirtualPageNumber: TYPE = PageNumber;
```

A block of 256 pages aligned on a 64K-word boundary is called a *bank*. Unless otherwise noted, all memory sizes are stated in units of sixteen-bit words; for example, banks are 64K words.

A virtual address occupies two words: the smallest virtual address is zero and the largest is 2^{32}-1. The most significant bit of the address is numbered zero. The least significant is numbered thirty-one. Virtual addresses are represented by values of type LONG POINTER. As with all LONG data types, when a long pointer is stored in memory, the least significant word appears at the lower-numbered address (§2.3.1).

Within distinguished regions of the virtual memory, called *Main Data Spaces*, data can be referenced using *short pointers*. These addresses occupy a single word and are represented by values of type POINTER. §3.2 discusses the structures contained in Main Data Spaces and the conversion of short to long pointers .

Design Note: Mesa processors may implement virtual and real address spaces smaller than 2^{32} words (see below). Regardless of the actual size of virtual memory, long pointers always occupy two words, and the unused bits must be zero.

3.1.1. Virtual Memory Mapping

Virtual addresses are mapped into real addresses via the *mapping mechanism*. The implementation of the mapping operations is processor-dependent, and it is modeled in this document by the operations ReadMap and WriteMap. Logically, these operations implement an array of real page numbers indexed by virtual page number, except that the array can have holes, allowing for an associative or hashed implementation.

The address-translation process is identical for *all* memory accesses, whether they originate from the processor or from I/O devices. There is no method for bypassing the address-translation mechanism and directly referencing a main memory location using a real address. The virtual-to-real mapping can always be determined using the map instructions (§3.1.2).

Like virtual memory, real memory is referenced by thirty-two bit addresses in the range [0..2^{32}-1]. However, the real address space is not necessarily contiguous or complete; there may be gaps where no real memory resides, and some models of the processor may implement less than 2^{32} words of real memory. The size of a gap in the real address space must always be a multiple of the page size and begin on a page boundary.

The mapping mechanism identifies the real page that corresponds to a given virtual page (if any). Each virtual page is mapped individually, and a contiguous region of virtual memory does not necessarily correspond to a contiguous block of real memory. A thirty-two bit virtual address is mapped into a thirty-two bit real address, as illustrated in Figure 3.1.

The mapping mechanism is described by an array, indexed by the virtual page number, containing the associated real page numbers. It also contains the access flags, some of which are processor-dependent. The flags have the following format, although some reserved bits may not be present in all implementations of the processor:

```
MapFlags: TYPE = MACHINE DEPENDENT RECORD [
  reserved (0: 0..12): UNSPECIFIED[0..17777B],
  protected (0: 13..13): BOOLEAN,
  dirty (0: 14..14): BOOLEAN,
  referenced (0: 15..15): BOOLEAN];
```

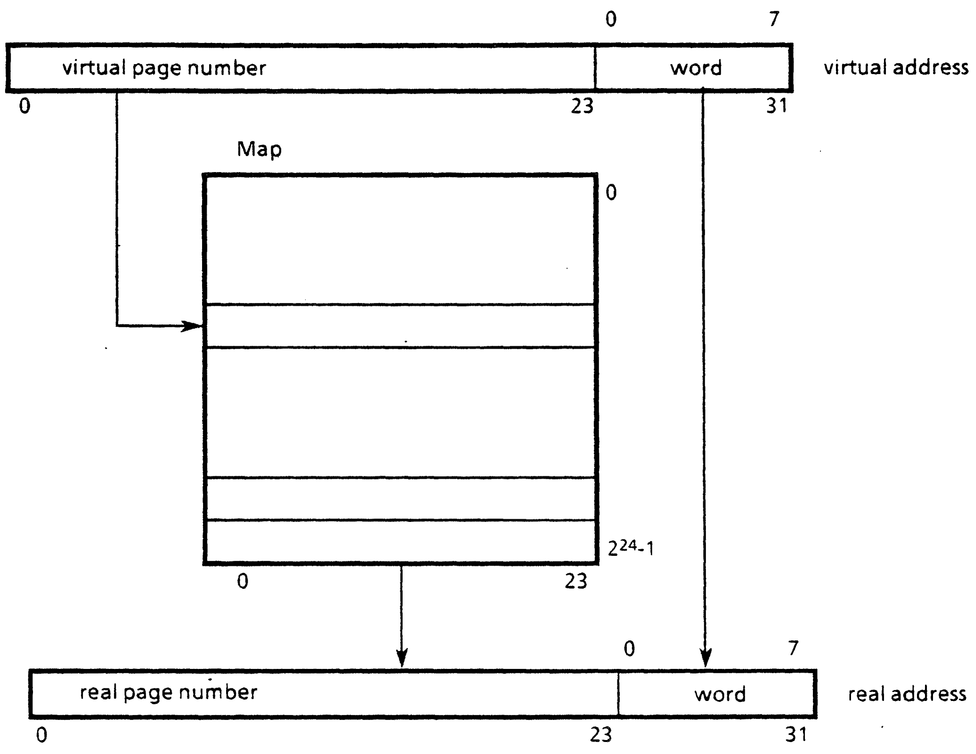


Figure 3.1 Virtual Memory Mapping

Design Note: The processor returns zero as the value of each unimplemented reserved bit. Any unimplemented bits supplied by the programmer when writing the map are ignored.

The flags encode access properties of the real page, if one is assigned. These bits can be read and written both by the processor and by the programmer using the map instructions defined in §3.1.2. Three of the flags, *protected*, *dirty*, and *referenced*, are defined by the architecture. Other processor-dependent flags (up to thirteen) may also be defined. The write-protect bit (protected) is set by the programmer. It prohibits writing into the page, causing a *write-protect fault* if a store is attempted. The dirty bit (dirty) is set by the processor if a store is done into a non-write protected page. The referenced bit (referenced) is set by the processor on any read or write access of a word within the page.

A distinguished encoding of the flags called *vacant* signifies that the virtual page is not present in real memory (that is, it is unmapped). If a read or write operation is performed on a page with flag bits set to vacant, a *page fault* occurs.

```
Vacant: PROCEDURE [flags: MapFlags] RETURNS [BOOLEAN] =
  BEGIN
  RETURN[flags.protected AND flags.dirty AND ~flags.referenced];
  END;
```

The mapping operation is defined by the following routine, which maps a virtual address into a real address. Both types of addresses are represented by LONG POINTERs, but the Mesa dereferencing operator (↑) is applied only to real addresses in the instruction descriptions that follow.

```
Map: PROCEDURE [virtual: LONG POINTER, op: {read, write}] RETURNS [real: LONG POINTER] =
  BEGIN
  mf: MapFlags;
  rp: RealPageNumber;
  adrs: LONG CARDINAL = LOOPHOLE[virtual];
  vp: VirtualPageNumber = adrs/PageSize;
  wa: LONG CARDINAL = adrs MOD PageSize;
  [flags: mf, real: rp] ← ReadMap[vp];
  IF Vacant[mf] THEN PageFault[virtual];
  IF op = write THEN
    IF mf.protected THEN WriteProtectFault[virtual]
    ELSE mf.dirty ← TRUE;
  mf.referenced ← TRUE;
  WriteMap[virtual: vp, flags: mf, real: rp];
  RETURN[LOOPHOLE[rp*PageSize + wa]];
  END;
```

Note: The PageFault and WriteProtectFault routines do not return control to Map. Instead, they raise the Abort signal (§4.1).

Implementation Note: Operations on the map must be atomic with respect to accesses by other processors and I/O devices. That is, after the ReadMap has taken place, other accesses must be prohibited until the following WriteMap completes (or a fault occurs).

Design Note: When accessing data structures declared to be resident in real memory, the processor need not maintain the dirty and referenced flags. The resident structures are the Process Data Area and the State Vectors that it points to (§10.1.1).

The operations ReadMap and WriteMap are implementation-dependent, and are described by the following interface:

```
ReadMap: PROCEDURE [virtual: VirtualPageNumber]
  RETURNS [flags: MapFlags, real: RealPageNumber];

WriteMap: PROCEDURE [
  virtual: VirtualPageNumber, flags: MapFlags, real: RealPageNumber];
```

These operations have the following properties (see also §3.1.2):

It is illegal to attempt to map more than one virtual page to the same real page. This restriction allows an associative or hashed implementation of the map in which there is only one map entry for each real memory page.

In the case where ReadMap returns flags indicating vacant, the value of the real page number returned by the operation is undefined.

In the case where WriteMap is supplied with flags indicating vacant, the value of the real page number supplied by the caller is ignored.

Implementation Note: Each implementation of the processor may handle out-of-bounds virtual addresses differently. The hardware may be designed to make ReadMap and WriteMap return the appropriate flags for this condition. Otherwise, SetMap and GetMapFlags can be used to do address checking.

Programming Note: The maximum size of virtual memory can be determined by attempting to map a real page to each possible virtual page and then checking the flags of its map entry for vacant.

3.1.2 Memory Map Instructions

The *map instructions* are used to maintain the correspondence between virtual and real pages. The SetMap instruction replaces an entry in the map. GetMapFlags reads the flags and real page number from a map entry, given a virtual page number; SetMapFlags reads an entry and updates it with new flags obtained from the stack, provided the flags do not indicate vacancy. Note that SetMap and SetMapFlags must atomically update the map, and that no mapping operations may occur while the map is being updated.

Implementation Note: The atomicity requirements on SetMap and GetMapFlags may be replaced with a rule allowing only one processor in the system ever to write the map. Such a rule would imply that the privileged processor must preset the map flags for each addressing operation performed by the other processors or I/O controllers.

Programming Note: It is illegal to map more than one virtual page to the same real page, so the results of such an operation are undefined. A detailed discussion of the properties of the map can be found in the previous section.

The stack and the Push(Long) and Pop(Long) routines are defined in §3.3.2.

SM Set Map

If the flags specify that the page is vacant, Set Map ignores the real page number, except to pop it from the stack.

```
SM: PROCEDURE =
  BEGIN
  mf: MapFlags = Pop[];
  rp: RealPageNumber = PopLong[];
  vp: VirtualPageNumber = PopLong[];
  WriteMap[virtual: vp, flags: mf, real: rp]
  END;
```

GMF Get Map Flags

If the flags returned indicate a vacant map entry, the real page number is undefined.

```
GMF: PROCEDURE =
  BEGIN
  mf: MapFlags;
  rp: RealPageNumber;
  vp: VirtualPageNumber = PopLong[];
  [flags: mf, real: rp] ← ReadMap[vp];
```

```
        Push[mf];
        PushLong[rp];
        END;
```

SMF Set Map Flags

If the old flags indicate a vacant entry, the real page number is undefined, and the new flags taken from the stack are ignored.

```
    SMF: PROCEDURE =
        BEGIN
        mf: MapFlags;
        rp: RealPageNumber;
        newMf: MapFlags = Pop[];
        vp: VirtualPageNumber = PopLong[];
        [flags: mf, real: rp] ← ReadMap[vp];
        Push[mf];
        PushLong[rp];
        IF ~Vacant[mf] THEN
            WriteMap[virtual: vp, flags: newMf, real: rp];
        END;
```

Programming Note: SetMapFlags cannot change the status of an entry from vacant to non-vacant, since that would require a new real page number. SetMap must be used for this purpose.

3.1.3 Virtual Memory Access

In the code that follows, the Fetch and Store routines are used to perform the mapping operation. They return a LONG POINTER that is the real address produced by Map. This notation allows the use of the Mesa operator (↑) to dereference the pointers. Wherever this operator appears, a real memory access takes place, and all real memory accesses are denoted by this operator. There are no virtual address dereferences in this document, and the code does not make use of Mesa's implicit pointer-dereferencing rules. Notice that wherever calls on Fetch or Store appear, a page fault or write-protect fault might result (in the form of an Abort signal; see §10.4.3).

```
    Fetch: PROCEDURE [virtual: LONG POINTER]
        RETURNS [real: LONG POINTER] =
        BEGIN
        RETURN[Map[virtual, read]];
        END;

    Store: PROCEDURE [virtual: LONG POINTER]
        RETURNS [real: LONG POINTER] =
        BEGIN
        RETURN[Map[virtual, write]];
        END;
```

To allow convenient access to double-word structures, the following operation is also defined, which checks for faults on each word of the data:

```
ReadDbl: PROCEDURE [virtual: LONG POINTER]
  RETURNS [data: LONG UNSPECIFIED] =
  BEGIN
  temp: Long;
  temp.low ← Fetch[virtual]↑;
  temp.high ← Fetch[virtual + 1]↑;
  RETURN[LOOPHOLE[temp]];
  END;
```

Design Note: There are currently no requirements to provide a double-word store operation that is atomic with respect to page faults (see §4.6.1).

3.1.4 Virtual Memory Data Structures

This section summarizes the data structures of the architecture that are global to the entire virtual memory. Other structures are local to and replicated in each Main Data Space, and are described in §3.2.2. The overall structure of virtual memory is illustrated in figure 3.2.

3.1.4.1 Reserved Locations

A contiguous area beginning at page zero of virtual memory is reserved for the booting process (see §4.7). Page zero normally is not used thereafter. It should not be allocated by the programmer, so that most uses of zero long pointers will cause faults.

```
PageZero: LONG POINTER = LOOPHOLE[LONG[0]];
```

During booting, the initialization process may construct and store in main memory information about the machine configuration, device and controller identification, diagnostic information, and error status, in addition to the software necessary for initial program bootstrap. The format, content, and size of this area is processor-dependent; however, it is always contained within the first 64K of virtual memory (called bank zero) beginning at page zero.

```
BootArea: LONG POINTER TO BootData = LOOPHOLE[LONG[0]];
BootData: TYPE = BLOCK;
```

On most processors, one or more pages are reserved for communication between I/O devices or controllers and the processor. The format, content, size, and location of this area is processor-dependent; however, it is always contained within the first 64K of virtual memory.

```
IOArea: LONG POINTER TO IOData;
IOData: TYPE = BLOCK;
```

Other reserved virtual memory locations may be assigned by the programmer. Real memory locations may be reserved for I/O devices, but these areas must be mappable to any virtual memory address by the programmer.

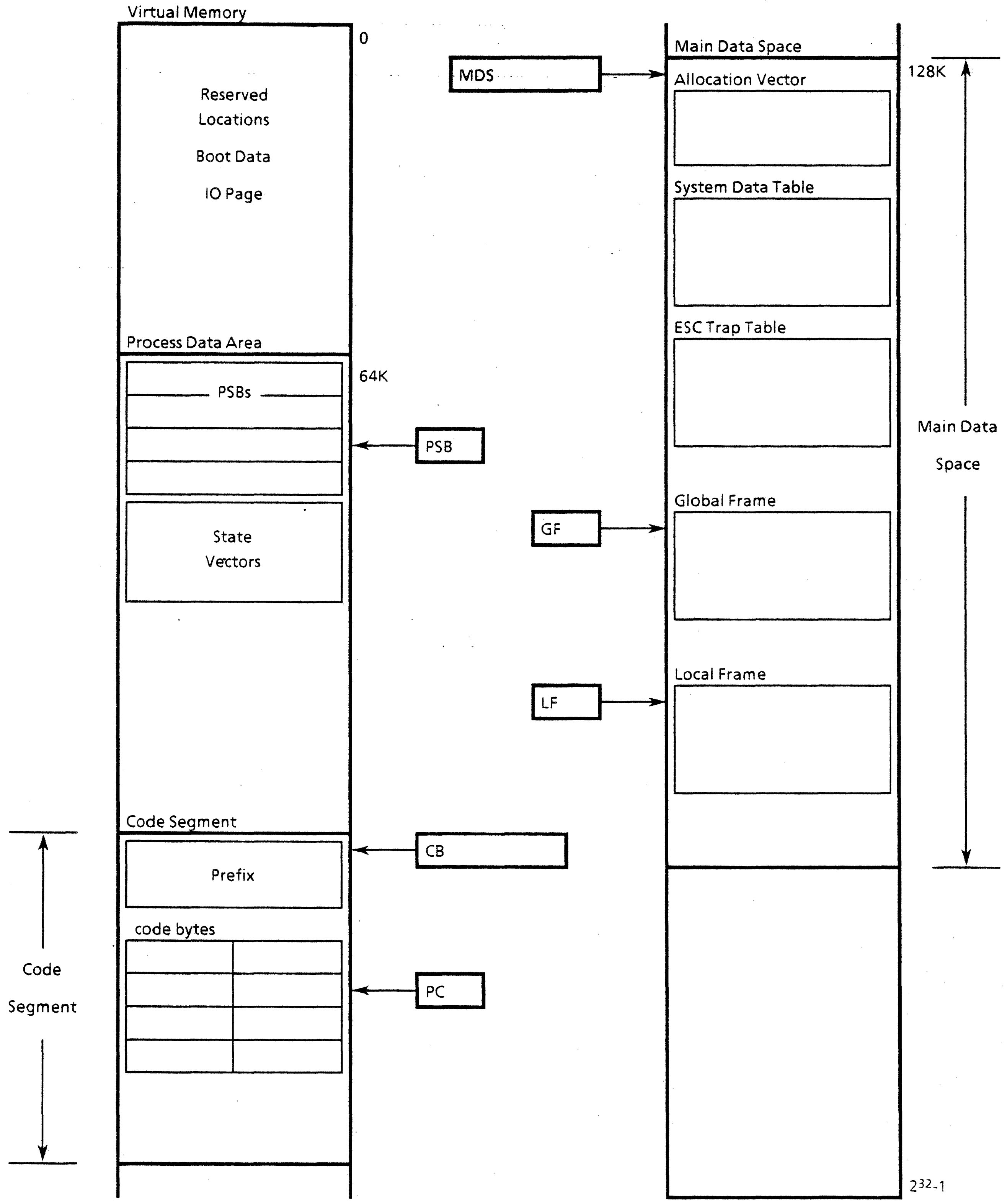


Figure 3.2 Virtual Memory Structure

3.1.4.2 Process Data Structures

The Process Data Area (PDA) contains information recording the state of each process and a pool of state vectors used to save the state of a preempted process. Substructures of the PDA support the handling of faults, interrupts, and timeouts.

```
PDA: LONG POINTER TO ProcessDataArea;
```

The structure and content of the Process Data Area are described in §10.1.1. The location of the PDA is defined in Appendix A.

3.1.4.3 Code Segments

An arbitrary number of *code segments* may be allocated anywhere in virtual memory other than in the reserved locations. (The BootArea may include code, however.) A code segment contains read-only instructions and constants for the procedures that comprise a Mesa program module; it is never modified during the course of normal execution. A code segment is relocatable without modification; no information in a code segment depends on its location in virtual memory.

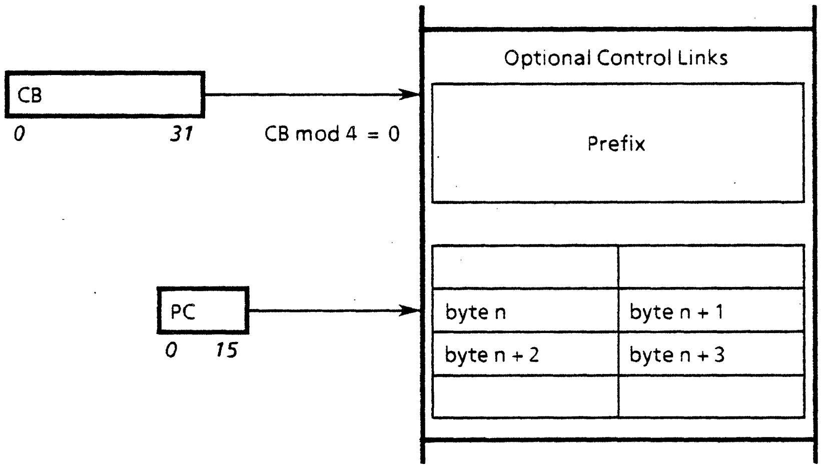


Figure 3.3 Code Segment

The beginning of the currently-executing code segment is pointed to by the CB register (the *code base*, a long pointer). The code segment is quad-word aligned (that is, CB modulo 4 = 0), and no part of a code segment may cross a 64K word boundary. These restrictions must be enforced by the programmer.

```
CodeSegment: TYPE = MACHINE DEPENDENT RECORD [
   available (0): ARRAY [0..4) OF UNSPECIFIED,
   code (4): BLOCK];
```

Note: The array code is of zero length, and is just a place-holder in the record declaration for the beginning of the actual code bytes for the code segment.

In addition to a small amount of space availiable to the programmer, the code segment may contain a number of *control links* located immediately before the word pointed to by the code base (control links are described in §9.1.). These links are used to call procedures or reference variables in other program modules. The links must also be contained in the same 64K bank as the rest of the code segment.

The program counter (PC) points to instruction or operand bytes in the code segment. It is a byte offset, relative to the code base. The code bytes are addressed left-to-right within a word, byte zero being bits zero through seven, byte one being bits eight to fifteen. Since the program counter is sixteen bits, it can reference up to 64K bytes of code starting at the code base. (This is the maximum size of a code segment.)

Design Note: It is illegal for a program to unmap the page to which the PC currently points, to clear the referenced bit, or to modify the dirty bit of that page. It is also illegal to write into the current code segment pointed to by CB.

Implementation Note: These restrictions allow the processor to cache a portion of the current instruction stream and the map entry of the current code page, and to set its map flags only when the page is first referenced.

Several instructions use the code base in conjunction with a *word* offset into the code segment to obtain operands. These instructions call the following routine:

```
ReadCode: PROCEDURE [offset: CARDINAL]
  RETURNS [UNSPECIFIED] =
  BEGIN
  RETURN[Fetch[CB + LONG[offset]] ↑ ];
  END;
```

Programming Note: Because code segments do not cross 64K boundaries, the calculation CB + LONG[offset] can be implemented as a short addition of the offset to the least significant word of the code base, with no possibility of overflow into the high-order word.

3.2 Main Data Spaces

A Main Data Space (MDS) is a contiguous region of 64K words of virtual memory. All *short pointers* access memory locations within an MDS, and all local and global frames (§3.2.2.2) are allocated within an MDS. The purpose of Main Data Spaces is to allow the most commonly used data structures to be referenced by single word rather than long (double-word) pointers.

```
MdsHandle: TYPE = LONG POINTER TO MainDataSpace;
MainDataSpace: TYPE = BLOCK;
```

Main Data Spaces are 64K-word aligned and thus do not cross 64K word boundaries. Several MDSs can be allocated in virtual memory, but only one is current; its address is contained in the thirty-two bit MDS register (§3.3.1).

3.2.1 Main Data Space Access

A *short pointer* is a sixteen-bit quantity that addresses a location within the current MDS relative to its base. To construct a thirty-two bit virtual address from a short pointer, it is simply added to the **MDS** register. The following routine is used to perform this conversion:

```
LengthenPointer: PROCEDURE [ptr: POINTER] RETURNS [LONG POINTER] =
  BEGIN
  offset: CARDINAL = LOOPHOLE[ptr];
  RETURN[MDS + LONG[offset]];
  END;
```

Design Note: Because the MDS is 64K-word aligned and short pointers are restricted to a 64K range within the MDS, a concatenation operation can replace the addition that appears above. All operations on short pointers are performed modulo 2^{16}, ignoring overflow.

Programming Note: A Main Data Space can be less than 64K, but the processor is ignorant of the size of the current MDS. It is up to the programmer to ensure that short pointers do not exceed the actual size of the MDS.

The Lengthen Pointer instruction converts a short pointer to a long pointer using LengthenPointer. Notice that it treats zero (the standard value of NIL) as a special case.

LP Lengthen Pointer

```
LP: PROCEDURE =
  BEGIN
  ptr: POINTER = Pop[];
  PushLong[
    IF ptr = LOOPHOLE[0] THEN LONG[0]
    ELSE LengthenPointer[ptr]];
  END;
```

The routines below are defined in terms of Fetch, Store and ReadDbl (§3.1.3) for mapping short pointers. They use the value of the **MDS** register to lengthen the short pointers.

```
FetchMds: PROCEDURE [ptr: POINTER]
  RETURNS [real: LONG POINTER] =
  BEGIN
  RETURN[Fetch[LengthenPointer[ptr]]];
  END;

StoreMds: PROCEDURE [ptr: POINTER]
  RETURNS [real: LONG POINTER] =
  BEGIN
  RETURN[Store[LengthenPointer[ptr]]];
  END;

ReadDblMds: PROCEDURE [ptr: POINTER]
  RETURNS [data: LONG UNSPECIFIED] =
  BEGIN
```

```
RETURN[ReadDbl[LengthenPointer[ptr]]];
END;
```

3.2.2 Main Data Space Data Structures

The following sections summarize the data structures contained in each Main Data Space. These include page zero, the control-transfer data structures, and local and global frames. Other structures in the MDS may be allocated by the programmer.

3.2.2.1. Reserved Locations

The data structures given in this section are located at fixed addresses in each Main Data Space; when not specified here, their locations are defined in Appendix A.

Page zero of each MDS is not normally used (except during the booting process; see §4.7). It should not be allocated by the programmer, so that most references through zero short pointers will cause faults.

Each Main Data Space contains an Allocation Vector (AV), a System Data table (SD), and an ESC Trap Table (ETT). These data structures are used by the control-transfer mechanism described in detail in §9. A brief summary is contained below:

```
AV: POINTER TO AllocationVector;
```

The procedure-calling mechanism allocates space for local variables dynamically from a *frame heap*, rather than a stack. This method allows for arbitrary control-transfer disciplines in addition to simple call-return. It also allows several processes to share the same heap. The Allocation Vector is used to maintain the heap and to simplify the allocation and deallocation of frames, so that the common cases can be implemented by the processor. The details of the allocation mechanism are described in §9.2.

```
SD: POINTER TO SystemData;
```

The System Data table is used to contain pointers (in the form of control links; see §9.1) to the trap-handling routines called when the processor determines that execution of the current instruction cannot be completed. (Details of the trap mechanism are described in §9.5.) This table is also used to contain pointers (also in the form of control links) to commonly used runtime facilities (see §9.4.2).

```
ETT: POINTER TO EscTrapTable;
```

The ESC Trap table is used to contain pointers (in the form of control links; see §9.1) to trap handling routines called when the program executes an ESC or ESCL opcode which is implemented in software. (Details of the trap mechanism are described in §9.5.)

3.2.2.2 Local and Global Frames

To minimize the amount of addressing information needed to specify the location of an operand (and to maximize locality of reference), variables declared in Mesa programs and procedures are stored in *frames* and referenced relative to the beginning of these structures. Frames are contiguous linear structures in virtual memory that reside entirely within a Main Data Space. They are referenced relative to the **MDS** register using short pointers.

Programming Note: Except for the restriction that frames are contained entirely within a Main Data Space, the maximum size of a frame is not specified by the architecture.

Frames are of two types. *Global frames* contain the global variables declared in a program module. They are allocated statically when the module is loaded. *Local frames* contain the local variables declared in a procedure. They are allocated dynamically when the procedure is called, and they are deallocated when it returns. In addition to local and global variables, frames contain a small amount of overhead information used to record the size of the frame along with the linkages between procedure calls, between procedures and their containing modules, and between modules and their corresponding code segments. The details are described below.

Global Frames

A global frame represents an instance of a program module. It contains the module's globally-declared variables, preceded by a few overhead words. The first global variable is called *global zero*, the next *global one*, and so on.

```
GlobalFrameHandle: TYPE = POINTER TO GlobalVariables;
GlobalVariables: TYPE = BLOCK;
```

Design Note: Global zero must be quad-word aligned. The overhead words and the first four global variables must lie in the same page. This alignment simplifies page-faulting.

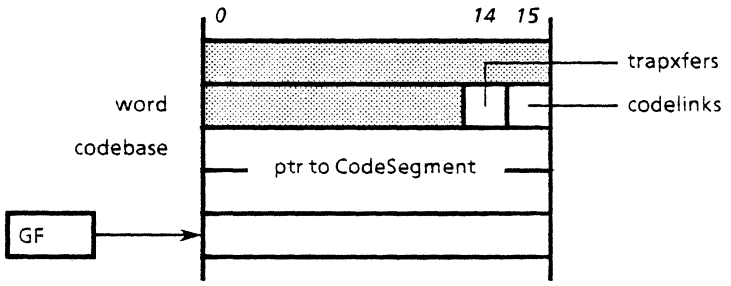


Figure 3.4 Global frame

The overhead words contain the location of the code segment from which instructions will be fetched when the module executes (the codebase field). These words also contain the flag bits trapxfers and codelinks used during control transfers (§9.3). Their remaining fields are available for use by the software.

```
GlobalFrameBase: TYPE = POINTER TO GlobalOverhead;

GlobalWord: TYPE = MACHINE DEPENDENT RECORD [
   available (0: 0..13): [0..37777B],
    trapxfers (0: 14..14): BOOLEAN,
    codelinks (0: 15..15): BOOLEAN];

  GlobalOverhead: TYPE = MACHINE DEPENDENT RECORD [
    available (0): UNSPECIFIED,
    word (1): GlobalWord,
    codebase (2): LONG POINTER TO CodeSegment,
    global (4): GlobalVariables];
```

Note: The array GlobalVariables is of zero length, It is just a place-holder in the record declaration for the starting address of the global variables.

The following routine is used to convert a global frame handle to a pointer to its overhead words:

```
  GlobalBase: PROCEDURE [frame: GlobalFrameHandle]
    RETURNS [GlobalFrameBase] =
    BEGIN
    RETURN[LOOPHOLE[frame-SIZE[GlobalOverhead]]];
    END;
```

Design Note: If a program modifies the overhead words of its own global frame, this modification may have no effect on their values as seen by the processor until the next control transfer into that module. This feature allows the processor to cache this (read-only) information in internal registers.

In addition to the overhead words, a number of *control links* can be located immediately before the global frame's overhead words. These links are used to call procedures and to reference variables in other program modules. Control links and the codelinks bit are discussed in §9. Also, see the Load Link instruction in §9.4.2.

Local Frames

A local frame represents an instance of a procedure. It contains the procedure's locally-declared variables, preceded by a few overhead words. The first local variable is called *local zero*, the next *local one*, and so on.

```
  LocalFrameHandle: TYPE = POINTER TO LocalVariables;
  LocalVariables: TYPE = BLOCK;
```

Design Note: Local zero and the overhead words must be quad-word aligned. The overhead words and the first four local variables must lie in the same page. This alignment simplifies page-faulting.

The overhead words contain the byte-relative location in the code segment from which instructions will be fetched when the procedure executes (the pc field). The globallink points to the procedure's global frame. It is used to gain access to the procedure's global variables. (It points to global zero, not to the overhead words.) The returnlink is a control link that normally points to the local frame of the procedure that created the current local frame (by a transfer of control; see §9.3). The overhead words also contain the frame's size, represented by a frame-size index (§9.2). The remaining field is available for use by the software.

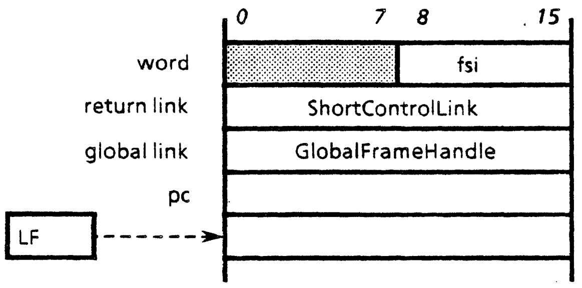


Figure 3.5 Local frame

```
LocalFrameBase: TYPE = POINTER TO LocalOverhead;

LocalWord: TYPE = MACHINE DEPENDENT RECORD [
   available (0: 0..7): BYTE,
   fsi (0: 8..15): FSIndex];

LocalOverhead: TYPE = MACHINE DEPENDENT RECORD [
   word (0): LocalWord,
   returnlink (1): ShortControlLink,
   globallink (2): GlobalFrameHandle,
   pc (3): CARDINAL,
   local (4): LocalVariables];
```

Note: The array LocalVariables is of zero length. It is just a place-holder in the record declaration for the starting address of the local variables.

The following routine is used to convert a local frame handle into a pointer to its overhead words.

```
LocalBase: PROCEDURE [frame: LocalFrameHandle]
   RETURNS [LocalFrameBase] =
   BEGIN
   RETURN[LOOPHOLE[frame-SIZE[LocalOverhead]]];
   END;
```

3.2.3 Frame Overhead Access

The overhead instructions are used to access overhead words of local and global frames. Read Overhead Byte obtains a pointer to a frame from the stack and moves a word from the overhead of that frame to the stack. Write Overhead Byte obtains a pointer to a frame from the stack and moves a word from the stack to the overhead of that frame.

Programming Note: Overhead words of frames must be accessed only by the overhead instructions. This restriction allows implementations to cache overhead words.

Programming Note: The programmer must ensure that the alpha byte (§4.2) in the overhead instructions is within the interval [1..4].

ROB Read Overhead Byte

```
ROB: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
  ptr: POINTER = Pop[];
  IF alpha ~IN [1..4] THEN ERROR;
  Push[FetchMds[ptr-alpha]↑];
  END;
```

WOB Write Overhead Byte

```
WOB: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
  ptr: POINTER = Pop[];
  IF alpha ~IN [1..4] THEN ERROR;
  StoreMds[ptr-alpha]↑ ← Pop[];
  END;
```

3.3 Processor Memories

This section describes those processor memories (usually called registers) visible to the programmer. An implementation of the architecture will typically include other internal registers as well. Several *control registers* are used to direct the execution of programs. The *evaluation stack*, which replaces the general-purpose registers found in most processor architectures, is also described. The data and status registers available to the programmer are listed, and the instructions used to access registers are defined.

3.3.1 Control Registers

The registers described below designate the process currently being executed by the processor, the Main Data Space in which it is executing, the frames to which it has direct access, and the location of the instructions being interpreted. All of the registers declared as pointer types contain virtual-memory addresses.

The state of the current process is recorded in a Process State Block (PSB). Its index into a table of such blocks located in the Process Data Area is contained in the register **PSB**.

```
PSB: PsbIndex;
```

The process state block contains, among other things, a pointer to the MDS in which the process is running. It also contains either a pointer to the process' local frame or a pointer to another structure (a State Vector) containing the process' evaluation stack (§3.3.2) and frame pointer. Details of the process structures are contained in §10.

The virtual address of the current Main Data Space is contained in the **MDS** register. The value of this register normally is changed only by a process switch (see §10). It can also be read and written using the register instructions (§3.3.4).

```
MDS: MdsHandle;
```

Implementation Note: Because the address of the MDS is always a multiple of 64K, MDS can be implemented using a sixteen-bit register.

Before a program can be run, an execution environment called a *context* must be established for it. In addition to a Main Data Space, a context includes:

- a pointer to the program's code segment,
- a pointer to its current instruction (the program counter), and
- pointers to its local and global data.

This information is stored in the overhead words of the program's local and global frame (§3.2.2.2).

The processor includes dedicated registers (described below) that contain pointers to the current local frame, global frame, and code segment, along with the current program counter. These registers are updated by the control-transfer instructions described in §9. Notice that a local frame is sufficient to determine all of the other registers: given a local frame pointer, the program counter is obtained from its pc field, the global frame pointer from its globallink field, and the code segment address from the global frame's codebase field. For this reason, the terms *frame* and *context* are often used interchangeably in the *PrincOps*, as are the terms *control transfer* and *context switch*.

The address of the local frame of the current context is contained in the sixteen-bit register LF (a short pointer). Its value is obtained directly from a Process State Block, from a State Vector, or via a control transfer (§9).

```
LF: LocalFrameHandle;
```

To access the overhead words of the current local frame, the procedure LocalBase[LF] is used. The register LF points to local zero, not to the overhead words. The format of a local frame is defined in §3.2.2.2.

The address of the global frame of the current context is contained in the sixteen bit register GF (a short pointer). Its value is obtained using LocalBase[LF].globallink.

```
GF: GlobalFrameHandle;
```

To access the overhead words of the current global frame, the procedure GlobalBase[GF] is used; the register GF points to global zero, not to the overhead words. The format of a global frame is defined in §3.2.2.2.

The address of the code segment of the current context is contained in the register CB (the code base, a long pointer). Its value is obtained using GlobalBase[GF].codebase. The format of a code segment is described in §3.1.4.3.

```
CB: LONG POINTER TO CodeSegment;
```

The current offset into the code stream is contained in the register PC (the *program counter*). It contains the byte offset, relative to the code base CB of the next byte to be fetched. It is obtained initially (on a control transfer) using LocalBase[LF].pc or from an entry vector.

```
PC: CARDINAL;
```

Except during execution of jump instructions and control transfers, the **PC** is maintained by the instruction-fetch routine GetCodeByte described in §4.3.

3.3.2 Evaluation Stack

The *Evaluation Stack* (usually just called the stack) is an array of registers normally accessed in a *last in, first out* manner. It is used as the source and destination of most transfers to and from memory. It is also used to pass parameters and results from one context to another during control transfers. Most arithmetic and logical operators also obtain their operands from the stack and return their results to the stack.

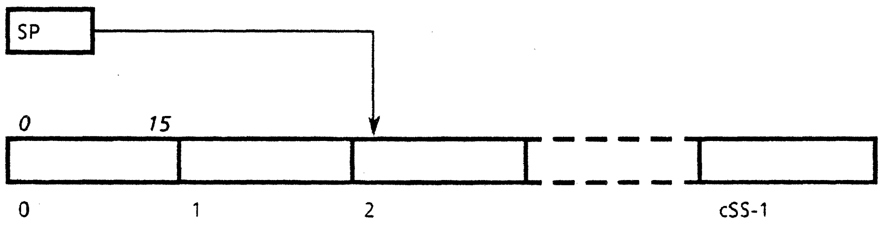


Figure 3.6 Evaluation Stack

The stack is represented by an array of StackDepth words and the Push and Pop routines. The variable **SP** (the *stack pointer*) indexes the next word above the top of the stack, so that the stack is empty when **SP** = 0 and full when **SP** = StackDepth. The StackCount routine returns the number of words currently on the stack. The value of cSS, the stack size, is given in Appendix A.

```
StackDepth: CARDINAL = cSS;
StackPointer: TYPE = [0..StackDepth];

SP: StackPointer;

stack: ARRAY [0..StackDepth) OF UNSPECIFIED;

StackCount: PROCEDURE RETURNS [StackPointer] =
  BEGIN
  RETURN[SP];
  END;
```

Push (PushLong) adds a word (two words) to the top; Pop (PopLong) removes the top word (two words) from the stack. If a push or pop would cause the stack pointer to be incremented or decremented out of range, a trap is generated.

Programming Note: The state of the stack after a stack error is undefined. Such an error is always fatal: it is illegal to resume a program that has generated a stack error (§4.6.1, §9.5).

Implementation Note: A stack error must always be detected by the processor, but it need not be reported during the execution of the instruction that caused it. This allows for pipelining in the arithmetic unit of the processor. A speedy report about the stack error (by the processor) is helpful for debugging, however. Otherwise, if another process switch occurs, the wrong process may be indicated as having a problem.

```
Push: PROCEDURE [data: UNSPECIFIED] =
  BEGIN
  IF SP = StackDepth THEN StackError[];
  stack[SP] ← data;
  SP ← SP + 1;
  END;

Pop: PROCEDURE RETURNS [UNSPECIFIED] =
  BEGIN
  IF SP = 0 THEN StackError[];
  SP ← SP-1;
  RETURN[stack[SP]];
  END;

PushLong: PROCEDURE [data: LONG UNSPECIFIED] =
  BEGIN
  Push[LowHalf[data]];
  Push[HighHalf[data]];
  END;

PopLong: PROCEDURE RETURNS [LONG UNSPECIFIED] =
  BEGIN
  long: Long;
  long.high ← Pop[];
  long.low ← Pop[];
  RETURN[LOOPHOLE[long]];
  END;
```

Note that double-word quantities are placed on the stack so that the least significant word is at the lower-numbered stack index (that is, on the bottom).

The stack is the primary source of instruction operands and the primary destination of results. The load instructions push words from memory onto the stack. The store instructions pop the stack into memory. The conditional jump instructions test words on the top of the stack and branch based on the result. The arithmetic instructions (or operands) pop their operands from the stack and push a result back onto the stack. Indirect instructions find their pointers on the stack.

Normally, the stack may contain results of previous computations that are to be combined with the result of the current instruction by execution of the operations following. However, a few instructions are *minimal stack*; that is, they require that the stack be empty except for their operands. These instructions call the following routine after popping their operands from the stack:

```
MinimalStack: PROCEDURE =
  BEGIN
  IF SP#0 THEN StackError[];
  END;
```

Some operations leave results or partial results *above the top of the stack*, that is, at stack[SP] and stack[SP + 1]. These results are not normally used, but they can be obtained using a Recover instruction (§5.1), which increments the stack pointer without disturbing the stack's contents. There is a corresponding Discard instruction that discards the top element of the stack by decrementing the stack pointer. These instructions are implemented using the following routines:

```
Recover: PROCEDURE =
  BEGIN
  IF SP = StackDepth THEN StackError[];
  SP ← SP + 1;
  END;

Discard: PROCEDURE =
  BEGIN
  IF SP = 0 THEN StackError[];
  SP ← SP-1;
  END;
```

The Multiply instruction (§5.5) provides an example of the use of Recover. It leaves the most significant word of a double-word result above the top of the stack. This allows a single instruction to function as both a single- and a double-precision operation.

In no cases are more than two words left above the top of the stack, and at most SP + 2 elements of the stack need be stored when its contents are saved (see §9.5.3).

Since the stack may not actually be implemented as an array, certain words left above the top of the stack may be lost. In particular, if more than two Recover instructions are executed sequentially, the excess Recovers may yield undefined results. In addition,even if they are not actually destroyed by the instruction, the original stack operands may not be recovered if the instruction changed the contents of the stack *and* changed the value of the stack pointer. (For example, one can not in general recover the original dividend after a divide, but the operand of a store instruction can always be recovered.) Exceptions to this rule are those instructions that leave results explicitly above the top of the stack. These values can always be obtained by Recovers (for example, MUL, and DIV).

Implementation Note: The intention of these restrictions is to allow the top few elements of the stack to be implemented using fast registers as a cache. The restrictions limit the cases in which the contents of the cache must be written back to the stack.

3.3.3 Data and Status Registers

The following additional data and status registers are accessible to the programmer using the register instructions (§3.3.4). In some models of the processor, these actually may be implemented in main memory or other auxiliary storage.

Each processor has a unique identification number guaranteed to be different from all others. This register is read-only.

```
PID: READONLY ARRAY [0..4) OF UNSPECIFIED;
```

Design Note: Currently, only 48 bits are implemented, and the first (high-order) word must be zero.

Most models of the processor include a maintenance panel for displaying error and status information to service personnel. This register is optional; if it is present, it is a write-only register.

```
MP: CARDINAL;
```

The *interval timer* allows high-resolution measurements of program performance and external events. It is incremented by one, modulo 2^{32}, at a constant rate. The units of the timer, called pulses, are processor-dependent, but must be in the range 1-100 microseconds.

```
IT: LONG CARDINAL;
```

The *wakeup mask register* contains a bit mask indicating which interrupt levels are assigned for internal use by the processor (see §10.4.4). It is read only.

```
WM: READONLY CARDINAL;
```

The *wakeup pending register* records the occurrence of wakeups that will later be translated into interrupts by the processor (see §10.4.4).

```
WP: CARDINAL;
```

The *wakeup disable counter* is used to control interrupt processing (see §10.4.4).

```
WDC: CARDINAL;
```

The *process timeout counter* is used to time out waiting processes (see §10.4.5).

```
PTC: CARDINAL;
```

The *xfer trap status* is used to control trapping of control transfers (see §9.5.5).

```
XTS: CARDINAL;
```

Additional data and status registers may be present in the processor and available to the programmer. Inclusion of the IEEE standard floating-point instructions may add such registers. Details of the format and content of these registers are under development.

3.3.4 Register Instructions

The *register* instructions read and write the contents of the programmer-visible registers defined in the previous sections.

RRIT **Read Register IT**

```
RRIT: PROCEDURE =
  BEGIN
  PushLong[IT];
  END;
```

RRMDS **Read Register MDS**

```
RRMDS: PROCEDURE =
  BEGIN
  Push[HighHalf[MDS]];
  END;
```

RRPSB **Read Register PSB**

```
RRPSB: PROCEDURE =
  BEGIN
  Push[Handle[PSB]];
  END;
```

RRPTC **Read Register PTC**

```
RRPTC: PROCEDURE =
  BEGIN
  Push[PTC];
  END;
```

RRWDC **Read Register WDC**

```
RRWDC: PROCEDURE =
  BEGIN
  Push[WDC];
  END;
```

RRWP **Read Register WP**

```
RRWP: PROCEDURE =
  BEGIN
  Push[WP];
  END;
```

RRXTS **Read Register XTS**

```
RRXTS: PROCEDURE =
  BEGIN
  Push[XTS];
  END;
```

WRIT **Write Register IT**

```
WRIT: PROCEDURE =
  BEGIN
```

```
    IT ← PopLong[];
    END;
```

WRMDS **Write Register MDS**

```
WRMDS: PROCEDURE =
    BEGIN
    MDS ← LongShift[LONG[Pop[]], WordSize];
    END;
```

WRMP **Write Register MP**

```
WRMP: PROCEDURE =
    BEGIN
    MP ← Pop[];
    END;
```

WRPSB **Write Register PSB**

```
WRPSB: PROCEDURE =
    BEGIN
    PSB ← Index[Pop[]];
    END;
```

WRPTC **Write Register PTC**

```
WRPTC: PROCEDURE =
    BEGIN
    PTC ← Pop[];
    time ← IT;
    END;
```

WRWDC **Write Register WDC**

```
WRWDC: PROCEDURE =
    BEGIN
    WDC ← Pop[];
    END;
```

WRWP **Write Register WP**

```
WRWP: PROCEDURE =
    BEGIN
    WP ← Pop[];
    END;
```

WRXTS **Write Register XTS**

```
WRXTS: PROCEDURE =
    BEGIN
    XTS ← Pop[];
    END;
```

Programming Note: Reading (writing) a write-only (read-only) register yields undefined results.

Programming Note: Writing the MDS register does not modify the other registers that define the current context. Matching local and global frames must exist in the new MDS at the MDS relative locations pointed to by the LF and GF registers.

The following table lists the sections in which each of these registers is defined (if it is not discussed in detail above). Unless otherwise noted, the register can be both read and written by the programmer.

PSB	Current Process State Block handle (§10.1.1).
MDS	Current Main Data Space address.
PID	Quad word processor id, read only.
MP	Maintenance panel, write only.
IT	Double word interval timer.
WM	Wakeup mask register, read only (§10.4.4).
WP	Wakeup pending register (§10.4.4.1).
WDC	Wakeup disable counter (§10.4.4.3).
PTC	Process timeout counter (§10.4.5).
XTS	Xfer trap status (§9.5.5).

Programming Note: The current local frame register (LF) and global frame register (GF) can be read using the LA0 and GA0 instructions (§7.2). They can be written by a control transfer (§9.4). The current code base (CB) can be obtained from the global frame.

4 Instruction Interpreter

This chapter describes the operation of the Mesa instruction interpreter. Only the main loop is contained here; the individual instructions are covered in other chapters. The instruction formats, instruction fetch, effective address calculation, and opcode dispatch are defined in this chapter. A description of exception processing (traps, faults, and interrupts) is also included. In the last section, the initial state of the processor is defined.

4.1 Interpreter

After initialization, the processor repeatedly interprets instructions as coded in the Processor routine. This task includes checking for pending interrupts and possible timeouts before the execution of each instruction.

```
Abort: ERROR = CODE;

Processor: PROCEDURE =
  BEGIN
  Initialize[];
  DO ENABLE Abort => LOOP;
    interrupt: BOOLEAN ← CheckForInterrupts[];
    timeout: BOOLEAN ← CheckForTimeouts[];
    IF interrupt OR timeout
      THEN Reschedule[preemption: TRUE]
      ELSE IF running THEN Execute[];
    ENDLOOP;
  END;
```

The initial state of the processor and its memories is defined in §4.7. The checks for pending interrupts and timeouts are discussed briefly in §4.6.2 and more throughly in §10.4. Reschedule and running are also defined there. The Execute routine defines instruction fetch, instruction decode, and opcode dispatch. (Details are in §4.3 and §4.5.)

In the event of an exception, the trap and fault routines use the signal Abort to return control to the main loop. In this situation, the processor continues executing instructions using the machine state established by the trap or fault routine; the intermediate state of the current opcode routine (and any routines that it has called) is discarded as a result of catching the signal. Abort is the only signal defined in the code. It is raised only by the trap and fault routines (see §4.6.1), and it is unwound only by the Processor routine.

4.2 Instruction Formats

The Mesa instruction set is composed of variable-length instructions of one, two, or three bytes in length. The most frequently used operations are encoded in a single byte. The first byte is always part of the opcode, and since there are more than 256 instructions, some extended opcodes occupy two or more bytes (the assignment of opcode values is given in the Appendix). In addition to the opcode, an instruction can contain one or two operand bytes, called alpha and beta. Additional operands may be present on the evaluation stack. The following illustration shows the possible instruction formats:

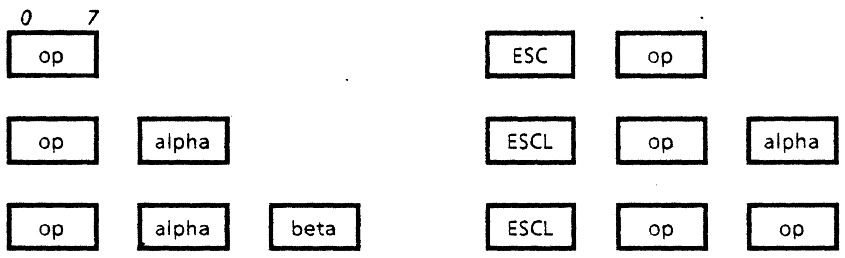


Figure 4.1 Instruction Formats

Design Note: The maximum size of an instruction is three bytes. Therefore, extended opcodes of two (or three) bytes can have only one (zero) operand bytes. This restriction establishes a minimum size for the optional instruction buffer (see §4.3).

Design Note: There are two escape opcodes, one for two-byte instructions (ESC) and one for three-byte instructions (ESCL). In both cases, the second byte is an extended opcode. This feature enables the programmer to determine instruction lengths by examining only the first instruction byte.

Design Note: The opcode values zero and 255 are reserved for internal use by the processor implementation. The programmer must ensure that these values will never appear as opcodes in an instruction stream contained in a code segment.

4.3 Instruction Fetch

Opcode and operand bytes are fetched from the code segment by the GetCodeByte routine. It maintains the program counter PC by incrementing it each time a byte is fetched. Thus the PC always points just beyond the current instruction byte (unless it is explicitly modified by the instruction, as in a jump). The beginning of the current instruction is pointed to by the variable savedPC, defined with **Execute** below (§4.5).

```
GetCodeByte: PROCEDURE RETURNS [UNSPECIFIED[0..377B]] =
  BEGIN
  even: BOOLEAN = (PC MOD 2) = 0;
  word: BytePair = ReadCode[PC/2]; PC ← PC + 1;
```

```
    RETURN[IF even THEN word.left ELSE word.right];
    END;
```

All operations on the PC are performed modulo 2^{16}, ignoring overflow. Backward jumps are performed by adding large unsigned positive numbers to the PC; this is equivalent to adding signed, two's-complement negative numbers, since overflow is ignored.

The following routine is used to fetch a word from the code segment; note that the bytes that make up the word may cross a word boundary in the instruction stream.

```
GetCodeWord: PROCEDURE RETURNS [UNSPECIFIED] =
  BEGIN
  word: BytePair;
  word.left ← GetCodeByte[];
  word.right ← GetCodeByte[];
  RETURN[word];
  END;
```

Implementation Note: As written, the **GetCodeByte** routine fetches a full word from memory each time it is called. To avoid these extra memory references, and generally to speed execution, most models of the processor implement an instruction buffer which holds the next few bytes of the code stream. Take care to ensure that page faults do not occur until the word containing the requested byte would have been accessed by calls on the above routines.

4.4 Address Calculation

There are no fixed address fields or addressing modes defined by the instruction format, since the address computation performed by an instruction is determined entirely by its opcode. However, there are some common patterns shared by several instructions. In general, addresses are computed in one of the following ways:

If the operands are on the evaluation stack, the stack pointer is used to address them (§3.3.2, §5).

If the operands are in the current local or global frame, the registers LF and GF are combined with an offset taken from the stack, the opcode, or the operand bytes alpha and beta (§7.2).

Operands elsewhere in memory are referenced using pointers taken from the stack or from the local or global frame. These pointers are usually combined with offsets obtained from the stack, the opcode, or the operand bytes alpha and beta (§7.3-5).

Operands located in the code are referenced using an offset from the code base CB or the PC. These offsets are taken from the stack, the opcode, or the operand bytes alpha and beta (§6, §7.5.1, and §8.1).

Programming Note: The architecture includes provisions for mapping code segments out of virtual as well as real memory (§9.5.1). Absolute virtual pointers into the code segment should therefore be used with caution. The processor never uses such pointers (except for CB), and all references to the code are relative to the code base. Furthermore, interruptible instructions (§4.6.2) must not save the code base as part of their intermediate state.

A complete definition of address calculation appears in the description of each instruction contained in the chapters that follow.

4.5 Instruction Execution

The following routine defines the initial processing of each opcode. Subsequent actions appear in separate routines defined for each instruction (or instruction class). If an opcode other than ESC or ESCL is unimplemented, an OpcodeTrap is generated (§9.5). Unimplemented ESC or ESCL opcodes generate an EscOpcodeTrap (§9.5). Identifiers beginning with z represent values of single byte opcodes, identifiers beginning with a represent values of ESC opcodes, and identifiers beginning with b represent values of ESCL opcodes. These conventions are fully listed in Appendix A.

```
break: BYTE;
savedPC: CARDINAL;
savedSP: StackPointer;

Execute: PROCEDURE =
  BEGIN
  savedPC ← PC;  savedSP ← SP;
  Dispatch[GetCodeByte[]];
  END;

Dispatch: PROCEDURE [opcode: BYTE] =
  BEGIN
  SELECT opcode FROM
    zLL0 = > LLn[0];
    zLL1 = > LLn[1];
    . . .
    zLLB = > LLB[];
    . . .
    . . .
    zBRK = > BRK[];
    zESC = >
      SELECT opcode ← GetCodeByte[] FROM
        aMW = > MW[];
        aMR = > MR[];
        . . .
        . . .
        ENDCASE = > EscOpcodeTrap[opcode];
    zESCL = >
      SELECT opcode ← GetCodeByte[] FROM
        bROB = > ROB[];
        . . .
        . . .
        ENDCASE = > EscOpcodeTrap[opcode];
    ENDCASE => OpcodeTrap[opcode];
  END;
```

Execute begins by saving the current values of the program counter and stack pointer before each instruction is executed (and before its opcode is fetched). This method allows the trap and fault routines to restore these variables to their original values so that the aborted instruction can be restarted (see below). Most jump instructions also use savedPC as the base for relative addressing (§6). The break byte and Dispatch are used by the breakpoint mechanism (§9.5.4).

4.6 Exceptions

An exception occurs when the processor determines that execution of the current instruction should be halted (perhaps before it has even begun), and that some other context or process should be run. There are three types of exceptions: traps, faults, and interrupts.

4.6.1 Traps and Faults

A *trap* results when the processor detects some condition that will not allow the current instruction to complete successfully (for example, a stack error or a zero divisor). An enumeration of these conditions is contained in §9.5.1. A trap causes the current context to be saved by a *trap routine*, a part of the processor. The trap then invokes a software *trap handler* using a transfer of control much like a procedure call (an XFER; see §9.3). This transfer does not change the current process, the Main Data Space, or the evaluation stack. The XFER may itself cause a trap (or a fault). Details of the context-switching mechanism and the trap routines are contained in §9.

An instruction experiences a *fault* if it causes a page fault (§3.1.1), a write protect fault (§3.1.1), or a frame allocation fault (§9.2). A fault calls a *fault routine*, which causes a process switch. This changes not only the current context, including the evaluation stack, but also the MDS and PSB registers as well, and makes a software *fault handler* ready, if it is not currently ready. (A trap handler, on the other hand, runs in the same process, and uses the same Main Data Space, as the context that caused the trap.) Details of the process switching mechanism and the fault routines are contained in §10.4.

Exceptions can occur at any time during the execution of an instruction. Therefore, care must be taken to define the state of the context when an exception occurs. This precaution allows the programmer to correct the cause of the problem (if possible) and restart the instruction that trapped, faulted, or was interrupted. Stated more precisely: if an exception occurs, the processor state, including the current context, the evaluation stack, and the other registers (defined in §3.3) must be restored to a state which, when used to restart the context, can result in the completion of the instruction as though the exception had not occurred. This is call the *Restart rule*, and like all good rules, there are exceptions involving fatal errors from which processing can not be resumed. They are discussed below.

The restart rule is intended to allow trap and fault handlers themselves to experience traps and faults. No matter how deeply nested the processor becomes in exception routines, only the last exception handler actually will get control. This handler will see the state of a context or a process that was not the original state or context. It only sees the state or context (which was excepted) of which it got control. When the trap or fault handler completes exception processing, it restarts the context in which its exception occurred. Occasionally the restart causes another trap or fault; it can even cause the same trap again.

The restart rule applies under a wide range of conditions. One extreme includes recoverable FAULTs, such as page faults, where the entire processor state from the beginning of the opcode is restored. For some cases of faults, the restart rule is used to restore only a partial state of the processor. At the other extreme are fatal traps. The processor is responsible only for leaving as much information as possible for debugging purposes. The following paragraphs cover these instances in more detail.

Generally, the processor adheres to the restart rule by restoring itself to what its state had been at the beginning of the current instruction. Because Execute saves the initial values of the stack pointer and program counter, the opcode routines are free to remove operands from the stack and fetch operand bytes, knowing that the trap and fault routines (§9.5.2, §10.4.3) will restore the SP and PC in the event of an exception (interrupts are handled differently; §4.6.2). The opcodes do not, however, push results back onto the stack. That method would destroy the original operands! They must be preserved until all possibility of a trap or fault has passed. Instead, the opcode routines use temporary variables to hold intermediate results until exceptions are no longer possible (for example, see RD0, RDB, RDL0, and RDLB in §7.3.1.1 and XFER in §9.3).

Another approach to recovering from a fault is to save and restore the processor state at any point where a fault may occur. This restores only that part of the state that had been altered. Because of its complexity, this approach is used only for lengthy opcodes such as those that transfer large blocks of data (see below).

Certain changes in the state of the computation are allowed even if a subsequent trap or fault could occur. In particular, any operation that is idempotent, and therefore by definition can be performed any number of times with the same result, can be completed before a subsequent operation that may cause a trap or fault. Likewise, instructions that update multiword structures in memory are not required to do so atomically. The restart rule can be interpreted to mean that the entire instruction is re-executed, without considering the effects of other processes that may have executed between the time of the exception and the resumption of the instruction. For example, instructions that store LONG types are not required to update the double word atomically. They can store the first word, and if a fault occurs on the second word, establish a state that will cause both words to be stored (the first for a second time) when the instruction is restarted; the modified locations need *not* be restored to their original values when the fault occurs (see SLD0, SLDB, WDBL in §7).

Programming Note: Multiword structures that must be updated atomically with respect to page faults should not be allocated across a page boundary. If a data structure requires synchronized access by several processes, the locking mechanism provided by monitors (§10) can be used.

For instructions that operate on large multiword structures, efficiency considerations discourage the strict interpretation of the restart rule. Strict interpretation would imply that the entire operation be started over. The BLT instruction (§8), which copies a block of up to 64K words from a source to a destination address, is an example. Each time around its main loop, it modifies the processor state so that the word transferred is no longer specified by the instruction operands. If an exception occurs during the next iteration, resumption results in transfer of the remaining words, without disturbing the data previously moved. All of the block-transfer instructions use this algorithm. Because of potentially long execution times, they also check for interrupts in their main loops.

Finally, there is an exception to the restart rule: some traps are considered fatal, so a context that experiences these traps can never be resumed (correctly). The state of the context is therefore undefined. Fatal traps have names that end with "Error" (e.g., StackError, RescheduleError). Depending on the type of error, either the current context, the current process, or the entire system may be unresumable (see §9.5).

Implementation Note: Although the state of a context that experiences a fatal trap is undefined, the processor should make an attempt to establish a context that is meaningful to the programmer for debugging purposes. Ideally, the program counter should point to the offending instruction, and the stack and stack pointer should reflect its operands. At a minimum, the value of the local frame pointer LF should be available, since little debugging is possible without it.

4.6.2 Interrupts

An *interrupt* occurs in response to a request for service, called a *wakeup*, from an external device or controller. Its effect is to notify a condition variable, which may make a software *interrupt handler* ready (if one is waiting). This may in turn cause a process switch (depending on the handler's priority; see §10.4).

As mentioned above, interrupts are handled differently with respect to the restart rule. Wakeups are buffered in the status register WP (wakeups pending), but otherwise ignored by most instructions (except the block transfers). If the main loop of the Processor routine detects a non-zero value in this register, execution of the current instruction is not started, and a possible process switch occurs, the details of which are described in §10.4. Because the check for interrupts is made before instruction execution begins (in fact, before the opcode is fetched), most instructions are not concerned with the possibility of an interrupt. However, the block transfer instructions (§8) are implemented in a fashion that allows their execution to be suspended and later resumed, as explained above.

4.7 Initial State

The routine below defines the initial conditions when the processor first starts execution; it sets up the processor registers and the current context. Initialize presumes that the memory map has been set to reflect all of the available real memory. Each real memory page must be entered in exactly one map entry corresponding to some existing virtual page, but not all mapped virtual memory need be contiguous. The protected flag of these map entries must be FALSE, but the dirty and referenced flags may be in an undefined state. The flags of all other map entries not assigned to real pages must be vacant (see §3.1.1). The reserved locations in the BootArea and the IOArea must be mapped.

Design Note: Not all of the real memory attached to the machine needs to be made available to the processor at initialization. In particular, so called *buffered devices* that require dedicated real memory will normally make this memory known to the software through the device implementation.

Initialization turns off interrupts, clears the process timeout registers, clears the XFER trap status register, and sets the running flag (§10). It empties the stack, initializes the MDS to the first 64K, and clears the break byte (§9.5.4). Initialize then performs an XFER through a fixed location in the System Data table. (sBoot is defined in Appendix A.)

```
Initialize: PROCEDURE =
  BEGIN
  -- Process registers
  WP ← 0;
  WDC ← 1;
  XTS ← 0;
  time ← IT;
  running ← TRUE;
  -- Context initialization
  SP ← 0;
  break ← 0;
  PSB ← 0;
  MDS ← LOOPHOLE[LONG[0]];
  XFER[dst: @SD[sBoot], src: 0, type: call];
  END;
```

Note that a trap or fault during the initial XFER results in an undefined machine state. (The call from the main program to Initalize is outside the scope of the Abort catch phrase.)

Programming Note: Except for the execution of the XFER (whose destination is an indirect control link), initialization does not reference main memory or disturb its contents.

5 Stack Instructions

The stack instructions manipulate the evaluation stack; except for the code stream, they do not reference memory. All of the usual logical and arithmetic operations (including some comparisons) are described in this chapter, as are primitives for manipulating the stack pointer and the order of the elements on the stack. Support for range checking and NIL pointer checking is also included. The basic arithmetic operators and the Not, And, Or, Xor, Shift, LongNot, LongAnd, LongOr, LongXor, LongShift, and SignExtend routines are defined in §2. The stack routines Push, Pop, Recover, and Discard are discussed in §3.3.2.

Note: None of the stack instructions operate on a stack element "in place"; they always remove their operands to check for stack underflow.

5.1 Stack Primitives

The instructions below are used to maintain the stack. Recover obtains the word above the top of the stack; Recover Two obtains the double word above the top of the stack. Discard removes the word on top of the stack; Discard Two removes the double word on top of the stack. Exchange interchanges the order of the top two words of the stack; Double Exchange interchanges the order of the top two double words of the stack. Duplicate makes a copy of the top word of the stack; Double Duplicate makes a copy of the top double word of the stack. Exchange Discard interchanges the order of the top two words on the stack, then removes the top word.

REC **Recover**

```
REC: PROCEDURE =
  BEGIN
  Recover[];
  END;
```

REC2 **Recover Two**

```
REC2: PROCEDURE =
  BEGIN
  Recover[];
  Recover[];
  END;
```

DIS **Discard**

```
DIS: PROCEDURE =
  BEGIN
  Discard[];
  END;
```

DIS2 **Discard Two**

```
DIS2: PROCEDURE =
  BEGIN
  Discard[];
  Discard[];
  END;
```

There are limitations on how many Recover instructions can follow particular instructions, and on when they may not yield meaningful results. See §3.3.2 for a discussion of these restrictions. ,

EXCH **Exchange**

```
EXCH: PROCEDURE =
  BEGIN
  v: UNSPECIFIED = Pop[];
  u: UNSPECIFIED = Pop[];
  Push[v]; Push[u];
  END;
```

DEXCH **Double Exchange**

```
DEXCH: PROCEDURE =
  BEGIN
  v: LONG UNSPECIFIED = PopLong[];
  u: LONG UNSPECIFIED = PopLong[];
  PushLong[v]; PushLong[u];
  END;
```

DUP **Duplicate**

```
DUP: PROCEDURE =
  BEGIN
  u: UNSPECIFIED = Pop[];
  Push[u]; Push[u];
  END;
```

DDUP **Double Duplicate**

```
DDUP: PROCEDURE =
  BEGIN
  u: LONG UNSPECIFIED = PopLong[];
  PushLong[u]; PushLong[u];
  END;
```

EXDIS **Exchange Discard**

```
EXDIS: PROCEDURE =
  BEGIN
  u: UNSPECIFIED = Pop[];
  v: UNSPECIFIED = Pop[];
  Push[u];
  END;
```

5.2 Check Instructions

The *check* instructions are used to implement runtime checks on array indexes, subranges, and pointers. They restore their first parameter to the stack for use by subsequent instructions, providing that the check succeeds (or for use by the trap handler, if the check fails).

BNDCK **Bounds Check**

```
BNDCK: PROCEDURE =
  BEGIN
  range: CARDINAL = Pop[];
  index: CARDINAL = Pop[];
  Push[index];
  IF index > = range THEN BoundsTrap[];
  END;
```

BNDCKL **Bounds Check Long**

```
BNDCKL: PROCEDURE =
  BEGIN
  range: LONG CARDINAL = PopLong[];
  index: LONG CARDINAL = PopLong[];
  PushLong[index];
  IF index > = range THEN BoundsTrap[];
  END;
```

NILCKL **Nil Check Long**

```
NILCKL: PROCEDURE =
  BEGIN
  ptr: LONG POINTER = PopLong[];
  nil: LONG POINTER = LOOPHOLE[LONG[0]];
  PushLong[ptr];
  IF ptr = nil THEN PointerTrap[];
  END;
```

BoundsTrap and PointerTrap are defined in §9.5.1.

5.3 Unary Operations

The *unary* instructions operate on the top single- or double-word element of the stack. They treat their operands as signed (two's complement) or unsigned binary numbers.

Programming Note: As long as overflow is ignored and does not occur, these instructions can be used both for signed and unsigned operations. Mesa does no overflow checking.

NEG Negate

```
NEG: PROCEDURE =
  BEGIN
  i: INTEGER = Pop[];
  Push[-i];
  END;
```

INC Increment

```
INC: PROCEDURE =
  BEGIN
  s: CARDINAL = Pop[];
  Push[s + 1];
  END;
```

DINC Double Increment

```
DINC: PROCEDURE =
  BEGIN
  s: LONG CARDINAL = PopLong[];
  PushLong[s + 1];
  END;
```

DEC Decrement

```
DEC: PROCEDURE =
  BEGIN
  s: CARDINAL = Pop[];
  Push[s-1];
  END;
```

ADDSB Add Signed Byte

```
ADDSB: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
  i: INTEGER = Pop[];
  Push[i + SignExtend[alpha]];
  END;
```

DBL **Double**

```
DBL: PROCEDURE =
  BEGIN
  u: UNSPECIFIED = Pop[];
  Push[Shift[u, 1]];
  END;
```

DDBL **Double Double**

```
DDBL: PROCEDURE =
  BEGIN
  u: LONG UNSPECIFIED = PopLong[];
  PushLong[LongShift[u, 1]];
  END;
```

TRPL **Triple**

```
TRPL: PROCEDURE =
  BEGIN
  s: CARDINAL = Pop[];
  Push[s * 3];
  END;
```

LINT **Lengthen Integer**

```
LINT: PROCEDURE =
  BEGIN
  i: INTEGER = Pop[];
  Push[i];
  Push[IF i < 0 THEN -1 ELSE 0];
  END;
```

SHIFTSB **Shift Signed Byte**

```
SHIFTSB: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
  u: UNSPECIFIED = Pop[];
  shift: INTEGER = SignExtend[alpha];
  IF shift ~IN [-15..15] THEN ERROR;
  Push[Shift[u, shift]];
  END;
```

5.4 Logical Operations

The *logical* instructions perform bitwise logical functions on the top two single- or double-word elements of the stack. The And instruction and Inclusive and Exclusive Or are defined in terms of the primitives in §2.1.3.1. The Shift instruction shifts u by ABS[shift] bits, left if shift is positive, right if shift is negative. Bits shifted off either end of u are lost; zeroes are shifted into u as necessary. A shift count greater than fifteen always yields a zero result.

AND **And**

```
AND: PROCEDURE =
  BEGIN
  v: UNSPECIFIED = Pop[];
  u: UNSPECIFIED = Pop[];
  Push[And[u, v]];
  END;
```

DAND **Double And**

```
DAND: PROCEDURE =
  BEGIN
  v: LONG UNSPECIFIED = PopLong[];
  u: LONG UNSPECIFIED = PopLong[];
  PushLong[LongAnd[u, v]];
  END;
```

IOR **Inclusive Or**

```
IOR: PROCEDURE =
  BEGIN
  v: UNSPECIFIED = Pop[];
  u: UNSPECIFIED = Pop[];
  Push[Or[u, v]];
  END;
```

DIOR **Double Inclusive Or**

```
DIOR: PROCEDURE =
  BEGIN
  v: LONG UNSPECIFIED = PopLong[];
  u: LONG UNSPECIFIED = PopLong[];
  PushLong[LongOr[u, v]];
  END;
```

XOR **Exclusive Or**

```
XOR: PROCEDURE =
  BEGIN
  v: UNSPECIFIED = Pop[];
  u: UNSPECIFIED = Pop[];
  Push[Xor[u, v]];
  END;
```

DXOR **Double Exclusive Or**

```
DXOR: PROCEDURE =
  BEGIN
  v: LONG UNSPECIFIED = PopLong[];
  u: LONG UNSPECIFIED = PopLong[];
```

```
        PushLong[LongXor[u, v]];
        END;
```

SHIFT **Shift**

```
    SHIFT: PROCEDURE =
        BEGIN
        shift: INTEGER = Pop[];
        u: UNSPECIFIED = Pop[];
        Push[Shift[u, shift]];
        END;
```

DSHIFT **Double Shift**

```
    DSHIFT: PROCEDURE =
        BEGIN
        shift: INTEGER = Pop[];
        u: LONG UNSPECIFIED = PopLong[];
        PushLong[LongShift[u, shift]];
        END;
```

ROTATE **Rotate**

```
    ROTATE: PROCEDURE =
        BEGIN
        rotate: INTEGER = Pop[];
        u: UNSPECIFIED = Pop[];
        Push[Rotate[u, rotate]];
        END;
```

Programming Note: The Not function is obtained by using Xor with one operand set to ones.

5.5 Arithmetic Operations

The following instructions perform arithmetic functions on the top two single- or double-word elements of the stack. They treat their operands as signed (two's complement) or unsigned binary numbers, and leave their results on the stack.

ADD **Add**

```
    ADD: PROCEDURE =
        BEGIN
        t: CARDINAL = Pop[];
        s: CARDINAL = Pop[];
        Push[s + t];
        END;
```

SUB Subtract

```
SUB: PROCEDURE =
  BEGIN
  t: CARDINAL = Pop[];
  s: CARDINAL = Pop[];
  Push[s-t];
  END;
```

The Double Add and Double Subtract instructions take thirty-two bit signed or unsigned operands and push a thirty-two bit result.

DADD Double Add

```
DADD: PROCEDURE =
  BEGIN
  t: LONG CARDINAL = PopLong[];
  s: LONG CARDINAL = PopLong[];
  PushLong[s + t];
  END;
```

DSUB Double Subtract

```
DSUB: PROCEDURE =
  BEGIN
  t: LONG CARDINAL = PopLong[];
  s: LONG CARDINAL = PopLong[];
  PushLong[s-t];
  END;
```

Programming Note: If overflow is ignored, the result of an add or subtract instruction can be considered to be either signed or unsigned.

The Add Double to Cardinal and Add Cardinal to Double instructions take a thirty-two bit and a sixteen-bit operand and push a thirty-two bit result.

ADC Add Double to Cardinal

```
ADC: PROCEDURE =
  BEGIN
  t: LONG CARDINAL = PopLong[];
  s: CARDINAL = Pop[];
  PushLong[LONG[s] + t];
  END;
```

ACD Add Cardinal to Double

```
ACD: PROCEDURE =
  BEGIN
  t: CARDINAL = Pop[];
  s: LONG CARDINAL = PopLong[];
```

```
    PushLong[s + LONG[t]];
    END;
```

The Multiply instruction computes the thirty-two bit product of the top two elements of the stack. The least significant word of the product is pushed onto the stack; the most significant word is left above the top of the stack, so it can be obtained using a Recover instruction.

MUL Multiply

```
MUL: PROCEDURE =
    BEGIN
    t: CARDINAL = Pop[];
    s: CARDINAL = Pop[];
    PushLong[LONG[s]*t];
    Discard[];
    END;
```

Programming Note: If the most significant word of the product is not recovered, the operation can be considered *either* signed or unsigned if overflow is ignored; otherwise it is unsigned.

DMUL Double Multiply

```
MUL: PROCEDURE =
    BEGIN
    t: LONG CARDINAL = PopLong[];
    s: LONG CARDINAL = PopLong[];
    PushLong[s*t];
    END;
```

The *divide* instructions divide a signed (unsigned) sixteen-bit dividend by a signed (unsigned) sixteen-bit divisor. The quotient is pushed onto the stack, and the remainder is left above the top of the stack so it can be obtained using a Recover instruction. In SDIV, the signs of the results are computed according to the rules of algebra (§2.2.2). In all divide instructions, a DivZeroTrap occurs if the divisor is zero (see §9.5.1).

SDIV Signed Divide

```
SDIV: PROCEDURE =
    BEGIN
    k: INTEGER = Pop[];
    j: INTEGER = Pop[];
    IF k = 0 THEN DivZeroTrap[];
    Push[j/k];
    Push[j MOD k];
    Discard[];
    END;
```

UDIV Unsigned Divide

```
UDIV: PROCEDURE =
  BEGIN
  t: CARDINAL = Pop[];
  s: CARDINAL = Pop[];
  IF t = 0 THEN DivZeroTrap[];
  Push[s/t];
  Push[s MOD t];
  Discard[];
  END;
```

The Long Unsigned Divide instruction divides an unsigned thirty-two bit dividend by an unsigned sixteen-bit divisor. The sixteen-bit quotient is pushed onto the stack, and the remainder is left above the top of the stack so it can be obtained using a Recover instruction.

LUDIV Long Unsigned Divide

```
LUDIV: PROCEDURE =
  BEGIN
  t: CARDINAL = Pop[];
  s: LONG CARDINAL = PopLong[];
  IF t = 0 THEN DivZeroTrap[];
  IF HighHalf[s] >= t THEN DivCheckTrap[];
  Push[LowHalf[s/LONG[t]]];
  Push[LowHalf[s MOD LONG[t]]];
  Discard[];
  END;
```

A DivCheckTrap (§9.5.1) is generated if the most significant word of the dividend is greater than the divisor, indicating that the quotient would overflow sixteen bits.

The *double divide* instructions divide a signed (unsigned) thirty-two bit dividend by a signed (unsigned) thirty-two bit divisor. The quotient is pushed onto the stack, and the remainder is left above the top of the stack so it can be obtained using a Recover Two instruction. In SDDIV, the signs of the results are computed according to the rules of algebra (§2.2.2). In all divide instructions, a DivZeroTrap occurs if the divisor is zero (see §9.5.1).

SDDIV Signed Double Divide

```
SDDIV: PROCEDURE =
  BEGIN
  k: LONG INTEGER = PopLong[];
  j: LONG INTEGER = PopLong[];
  IF k = 0 THEN DivZeroTrap[];
  PushLong[j/k];
  PushLong[j MOD k];
  Discard[],
    Discard[];
    END;
```

UDDIV **Unsigned Double Divide**

```
UDDIV: PROCEDURE =
    BEGIN
    t: LONG CARDINAL = PopLong[];
    s: LONG CARDINAL = PopLong[];
    IF t = 0 THEN DivZeroTrap[];
    PushLong[s/t];
    PushLong[s MOD t];
    Discard[];
    Discard[];
    END;
```

5.6 Comparison Operations

The *double compare* instructions compare two thirty-two bit signed or unsigned operands and push zero, one, or minus one depending on whether the operands compare equal, greater, or less.

DCMP **Double Compare**

```
DCMP: PROCEDURE =
    BEGIN
    k: LONG INTEGER = PopLong[];
    j: LONG INTEGER = PopLong[];
    Push[
        SELECT TRUE FROM
            j > k => 1,
            j < k => -1,
            ENDCASE => 0];
    END;
```

UDCMP **Unsigned Double Compare**

```
UDCMP: PROCEDURE =
    BEGIN
    t: LONG CARDINAL = PopLong[];
    s: LONG CARDINAL = PopLong[];
    Push[
        SELECT TRUE FROM
            s > t => 1,
            s < t => -1,
            ENDCASE => 0];
    END;
```

5.7 Floating Point Operations

The floating point instruction set is currently under development (see §2.2.3).

6

Jump Instructions

The jump instructions are of four types: unconditional, conditional, indexed, or absolute. The conditional jumps test against zero or compare two signed or unsigned operands. The indexed jumps index tables of displacements found in the current code segment. They are used to implement case statements.

All jumps are program-counter-relative, and all displacements are measured in bytes, relative to the first byte of the instruction (recorded in savedPC). The following example shows the two possible successors of a Jump Less Byte instruction (defined in §6.3):

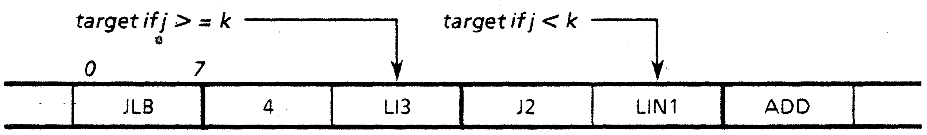


Figure 6.1 Jump Addressing

Note: Most of the jump opcodes add signed displacements, obtained by sign-extending alpha, to the unsigned PC. The only unsigned jump displacements are in the Jump Indexed instructions. All but JIW have their displacement in [-32768, 32767]. Arithmetic on the PC is always performed modulo 2^{16}, and overflow is ignored.

6.1 Unconditional Jumps

These instructions add a small constant, a sign-extended byte, or an INTEGER to the PC.

Jn **Jump n**

```
Jn: PROCEDURE [n: [2..8]] =
  BEGIN
  PC ← savedPC + n;
  END;
```

JB **Jump Byte**

```
JB: PROCEDURE =
    BEGIN
    disp: BYTE = GetCodeByte[];
    PC ← savedPC + SignExtend[disp];
    END;
```

JW **Jump Word**

```
JW: PROCEDURE =
    BEGIN
    disp: INTEGER = GetCodeWord[];
    PC ← savedPC + disp;
    END;
```

The Jump Stack instruction sets the PC to the value popped from the stack:

JS **Jump Stack**

```
JS: PROCEDURE =
    BEGIN
    PC ← Pop[];
    END;
```

The Catch instruction is used by the software to mark the code and indicate (in alpha) the catch phrase index. Except for its effects on the PC, Catch is a no-op.

CATCH **Catch**

```
CATCH: PROCEDURE =
    BEGIN
    alpha: BYTE = GetCodeByte[];
    END;
```

6.2 Equality Jumps

The equality jumps compare the top two elements of the stack or the top element and a constant (alpha or zero) for equality and jump accordingly. The jump pair opcodes compare the top of the stack with a four-bit field of alpha, and add a second four-bit field of alpha to the PC if the comparison succeeds.

JZn **Jump Zero n**

```
JZn: PROCEDURE [n: [3..4]] =
    BEGIN
    u: UNSPECIFIED = Pop[];
    IF u = 0 THEN PC ← savedPC + n;
    END;
```

JNZn **Jump Not Zero n**

```
JNZn: PROCEDURE [n: [3..4]] =
  BEGIN
  u: UNSPECIFIED = Pop[];
  IF u # 0 THEN PC ← savedPC + n;
  END;
```

JZB **Jump Zero Byte**

```
JZB: PROCEDURE =
  BEGIN
  disp: BYTE = GetCodeByte[];
  data: UNSPECIFIED = Pop[];
  IF data = 0 THEN PC ← savedPC + SignExtend[disp];
  END;
```

JNZB **Jump Not Zero Byte**

```
JNZB: PROCEDURE =
  BEGIN
  disp: BYTE = GetCodeByte[];
  data: UNSPECIFIED = Pop[];
  IF data # 0 THEN PC ← savedPC + SignExtend[disp];
  END;
```

JEB **Jump Equal Byte**

```
JEB: PROCEDURE =
  BEGIN
  disp: BYTE = GetCodeByte[];
  v: UNSPECIFIED = Pop[];
  u: UNSPECIFIED = Pop[];
  IF u = v THEN PC ← savedPC + SignExtend[disp];
  END;
```

JNEB **Jump Not Equal Byte**

```
JNEB: PROCEDURE =
  BEGIN
  disp: BYTE = GetCodeByte[];
  v: UNSPECIFIED = Pop[];
  u: UNSPECIFIED = Pop[];
  IF u # v THEN PC ← savedPC + SignExtend[disp];
  END;
```

JDEB **Jump Double Equal Byte**

```
JDEB: PROCEDURE =
  BEGIN
  disp: BYTE = GetCodeByte[];
  v: LONG UNSPECIFIED = PopLong[];
```

```
   u: LONG UNSPECIFIED = PopLong[];
   IF u = v THEN PC ← savedPC + SignExtend[disp];
   END;
```

JDNEB **Jump Double Not Equal Byte**

```
JDNEB: PROCEDURE =
   BEGIN
   disp: BYTE = GetCodeByte[];
   v: LONG UNSPECIFIED = PopLong[];
   u: LONG UNSPECIFIED = PopLong[];
   IF u # v THEN PC ← savedPC + SignExtend[disp];
   END;
```

JEP **Jump Equal Pair**

```
JEP: PROCEDURE =
   BEGIN
   pair: NibblePair = GetCodeByte[];
   data: UNSPECIFIED = Pop[];
   IF data = pair.left THEN PC ← savedPC + pair.right + 4;
   END;
```

JNEP **Jump Not Equal Pair**

```
JNEP: PROCEDURE =
   BEGIN
   pair: NibblePair = GetCodeByte[];
   data: UNSPECIFIED = Pop[];
   IF data # pair.left THEN PC ← savedPC + pair.right + 4;
   END;
```

JEBB **Jump Equal Byte Byte**

```
JEBB: PROCEDURE =
   BEGIN
   byte: UNSPECIFIED = GetCodeByte[];
   disp: BYTE = GetCodeByte[];
   data: UNSPECIFIED = Pop[];
   IF data = byte THEN PC ← savedPC + SignExtend[disp];
   END;
```

JNEBB **Jump Not Equal Byte Byte**

```
JNEBB: PROCEDURE =
   BEGIN
   byte: UNSPECIFIED = GetCodeByte[];
   disp: BYTE = GetCodeByte[];
   data: UNSPECIFIED = Pop[];
   IF data # byte THEN PC ← savedPC + SignExtend[disp];
   END;
```

6.3 Signed Jumps

The signed jump instructions compare the top two elements of the stack as two's complement signed operands and add a sign-extended alpha to the **PC** if the comparison succeeds.

JLB **Jump Less Byte**

```
JLB: PROCEDURE =
  BEGIN
  disp: BYTE = GetCodeByte[];
  k: INTEGER = Pop[];
  j: INTEGER = Pop[];
  IF j < k THEN PC ← savedPC + SignExtend[disp];
  END;
```

JLEB **Jump Less Equal Byte**

```
JLEB: PROCEDURE =
  BEGIN
  disp: BYTE = GetCodeByte[];
  k: INTEGER = Pop[];
  j: INTEGER = Pop[];
  IF j <= k THEN PC ← savedPC + SignExtend[disp];
  END;
```

JGB **Jump Greater Byte**

```
JGB: PROCEDURE =
  BEGIN
  disp: BYTE = GetCodeByte[];
  k: INTEGER = Pop[];
  j: INTEGER = Pop[];
  IF j > k THEN PC ← savedPC + SignExtend[disp];
  END;
```

JGEB **Jump Greater Equal Byte**

```
JGEB: PROCEDURE =
  BEGIN
  disp: BYTE = GetCodeByte[];
  k: INTEGER = Pop[];
  j: INTEGER = Pop[];
  IF j >= k THEN PC ← savedPC + SignExtend[disp];
  END;
```

6.4 Unsigned Jumps

The unsigned jump instructions compare the top two elements of the stack as unsigned operands and add a sign-extended alpha to the **PC** if the comparison succeeds.

JULB **Jump Unsigned Less Byte**

```
JULB: PROCEDURE =
  BEGIN
  disp: BYTE = GetCodeByte[];
  v: CARDINAL = Pop[];
  u: CARDINAL = Pop[];
  IF u < v THEN PC ← savedPC + SignExtend[disp];
  END;
```

JULEB **Jump Unsigned Less Equal Byte**

```
JULEB: PROCEDURE =
  BEGIN
  disp: BYTE = GetCodeByte[];
  v: CARDINAL = Pop[];
  u: CARDINAL = Pop[];
  IF u <= v THEN PC ← savedPC + SignExtend[disp];
  END;
```

JUGB **Jump Unsigned Greater Byte**

```
JUGB: PROCEDURE =
  BEGIN
  disp: BYTE = GetCodeByte[];
  v: CARDINAL = Pop[];
  u: CARDINAL = Pop[];
  IF u > v THEN PC ← savedPC + SignExtend[disp];
  END;
```

JUGEB **Jump Unsigned Greater Equal Byte**

```
JUGEB: PROCEDURE =
  BEGIN
  disp: BYTE = GetCodeByte[];
  v: CARDINAL = Pop[];
  u: CARDINAL = Pop[];
  IF u >= v THEN PC ← savedPC + SignExtend[disp];
  END;
```

6.5 Indexed Jumps

The indexed jumps update the PC from a table of byte displacements located in the code segment at offset base from the code base CB. If index is less than limit, the index added to base is used to extract a displacement from a table located in the current code segment. This displacement is then added to the PC. If index is out of range, no jump occurs. Jump Indexed Byte uses a table of eight bit entries, Jump Indexed Word uses sixteen-bit entries. The entries in both tables contain displacements measured in bytes. Note that in JIB, the displacement is *not* sign-extended.

JIB **Jump Indexed Byte**

```
JIB: PROCEDURE =
  BEGIN
  disp: BytePair;
  base: CARDINAL = GetCodeWord[];
  limit: CARDINAL = Pop[];
  index: CARDINAL = Pop[];
  IF index < limit THEN
    BEGIN
    disp ← ReadCode[base + index/2];
    PC ← savedPC + (
      IF (index MOD 2) = 0 THEN disp.left ELSE disp.right);
    END;
  END;
```

JIW **Jump Indexed Word**

```
JIW: PROCEDURE =
  BEGIN
  disp: CARDINAL;
  base: CARDINAL = GetCodeWord[];
  limit: CARDINAL = Pop[];
  index: CARDINAL = Pop[];
  IF index < limit THEN
    BEGIN
    disp ← ReadCode[base + index];
    PC ← savedPC + disp;
    END;
  END;
```

The ReadCode routine is defined in §3.1.4.3.

7

Assignment Instructions

The assignment instructions move words, double words, bytes, and arbitrary fields of words between the stack and memory. These include the immediate instructions, which obtain their operands from the code stream, the frame instructions, used to access local and global variables, and the instructions that dereference pointers (direct and indirect). The string and field instructions read and write substructures smaller than a word.

Design Note: In instructions that access both the stack and memory, if both a fault error and a stack error are possible, it is undefined which will occur first.

7.1 Immediate Instructions

The immediate instructions load one- or two-word constants onto the stack. Operands (if any) are obtained from the code stream.

LIN1 **Load Immediate Negative One**

```
LIN1: PROCEDURE =
  BEGIN
  Push[177777B];
  END;
```

LINI **Load Immediate Negative Infinity**

```
LINI: PROCEDURE =
  BEGIN
  Push[100000B];
  END;
```

LID0 **Load Immediate Double Zero**

```
LID0: PROCEDURE =
  BEGIN
  PushLong[LONG[0]];
  END;
```

LIn **Load Immediate n**

```
LIn: PROCEDURE [n: [0..10]] =
  BEGIN
  Push[n];
  END;
```

LIB **Load Immediate Byte**

```
LIB: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
  Push[alpha];
  END;
```

LINB **Load Immediate Negative Byte**

Note that alpha is *not* sign-extended.

```
LINB: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
  Push[BytePair[377B, alpha]];
  END;
```

LIHB **Load Immediate High Byte**

```
LIHB: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
  Push[BytePair[alpha, 0]];
  END;
```

LIW **Load Immediate Word**

```
LIW: PROCEDURE =
  BEGIN
  u: UNSPECIFIED = GetCodeWord[];
  Push[u];
  END;
```

7.2 Frame Instructions

The local and global frame instructions move one or two words between the stack and the frame. The opcodes differ primarily in their addressing modes: for frequently addressed variables, the offset of the variable in the frame is given by the instruction's opcode. Less frequently referenced frame variables are addressed by a one-byte offset obtained from alpha. Instructions are also provided for generating the address of a local or global variable.

7.2.1 Local Frame Access

The *load local, store local,* and *put local* instructions provide access to the local frame variables of the current context. The local-address instructions each generate a short pointer to a local variable, given its offset in the frame.

LAn **Local Address n**

```
LAn: PROCEDURE [n: [0..3,6,8]] =
  BEGIN
  Push[LF + n];
  END;
```

LAB **Local Address Byte**

```
LAB: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
  Push[LF + alpha];
  END;
```

LAW **Local Address Word**

```
LAW: PROCEDURE =
  BEGIN
  word: UNSPECIFIED = GetCodeWord[];
  Push[LF + word];
  END;
```

Programming Note: Local variables at offsets larger than 255 words from the base of the frame can be accessed by generating their addresses on the stack and then using the direct pointer instructions defined in §7.3.1.

7.2.1.1 Load Local

The load *local* instructions move one or two words onto the stack from the local frame.

LLn **Load Local n**

```
LLn: PROCEDURE [n: [0..11]] =
  BEGIN
  Push[FetchMds[LF + n]↑];
  END;
```

LLB **Load Local Byte**

```
LLB: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
  Push[FetchMds[LF + alpha]↑];
  END;
```

LLDn **Load Local Double n**

```
LLDn: PROCEDURE [n: [0..8,10]] =
  BEGIN
  Push[FetchMds[LF + n]↑];
  Push[FetchMds[LF + n + 1]↑];
  END;
```

LLDB **Load Local Double Byte**

```
LLDB: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
  Push[FetchMds[LF + alpha]↑];
  Push[FetchMds[LF + alpha + 1]↑];
  END;
```

7.2.1.2 Store Local

The *store local* instructions move one or two words from the stack to the local frame.

SLn **Store Local n**

```
SLn: PROCEDURE [n: [0..10]] =
  BEGIN
  StoreMds[LF + n]↑ ← Pop[];
  END;
```

SLB **Store Local Byte**

```
SLB: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
  StoreMds[LF + alpha]↑ ← Pop[];
  END;
```

SLDn **Store Local Double n**

```
SLDn: PROCEDURE [n: [0..6,8]] =
  BEGIN
  StoreMds[LF + n + 1]↑ ← Pop[];
  StoreMds[LF + n]↑ ← Pop[];
  END;
```

SLDB **Store Local Double Byte**

```
SLDB: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
  StoreMds[LF + alpha + 1]↑ ← Pop[];
```

```
    StoreMds[LF + alpha]↑ ← Pop[];
    END;
```

7.2.1.3 Put Local

The *put local* instructions move one or two words from the stack into the local frame, leaving its operands on the stack.

PLn **Put Local n**

```
PLn: PROCEDURE [n: [0..3]] =
    BEGIN
    SLn[n];
    Recover[];
    END;
```

PLB **Put Local Byte**

```
PLB: PROCEDURE [n: [0..3]] =
    BEGIN
    SLB[];
    Recover[];
    END;
```

PLD0 **Put Local Double Zero**

```
PLD0: PROCEDURE =
    BEGIN
    SLDn[0];
    Recover[];
    Recover[];
    END;
```

PLDB **Put Local Double Byte**

```
PLDB: PROCEDURE =
    BEGIN
    SLDB[];
    Recover[];
    Recover[];
    END;
```

7.2.1.4 Add Local

The Add Local Zero to Immediate Byte instruction adds local zero and a small constant from alpha and pushes the sum on the stack.

AL0IB **Add Local Zero to Immediate Byte**

```
AL0IB: PROCEDURE =
    BEGIN
    alpha: BYTE = GetCodeByte[];
    Push[FetchMds[LF]↑ + alpha];
    END;
```

7.2.2 Global Frame Access

The *load global* and *store global* instructions provide access to the global frame variables of the current context. The global address instruction generates a short pointer to a global variable, given its offset in the frame.

GAn **Global Address n**

```
GAn: PROCEDURE [n: [0..1]] =
    BEGIN
    Push[GF + n];
    END;
```

GAB **Global Address Byte**

```
GAB: PROCEDURE =
    BEGIN
    alpha: BYTE = GetCodeByte[];
    Push[GF + alpha];
    END;
```

GAW **Global Address Word**

```
GAW: PROCEDURE =
    BEGIN
    word: UNSPECIFIED = GetCodeWord[];
    Push[GF + word];
    END;
```

Programming Note: Global variables at offsets larger than 255 words from the base of the frame can be accessed by generating their addresses on the stack and then using the direct pointer instructions defined in §7.3.1.

7.2.2.1 Load Global

The *load global* instructions move one or two words onto the stack from the global frame.

LGn **Load Global n**

```
LGn: PROCEDURE [n: [0..2]] =
    BEGIN
    Push[FetchMds[GF + n]↑];
    END;
```

LGB **Load Global Byte**

```
LGB: PROCEDURE =
    BEGIN
    alpha: BYTE = GetCodeByte[];
    Push[FetchMds[GF + alpha] ↑ ];
    END;
```

LGDn **Load Global Double n**

```
LGDn: PROCEDURE [n: [0,2]] =
    BEGIN
    Push[FetchMds[GF + n] ↑ ];
    Push[FetchMds[GF + n + 1] ↑ ];
    END;
```

LGDB **Load Global Double Byte**

```
LGDB: PROCEDURE =
    BEGIN
    alpha: BYTE = GetCodeByte[];
    Push[FetchMds[GF + alpha] ↑ ];
    Push[FetchMds[GF + alpha + 1] ↑ ];
    END;
```

7.2.2.2 Store Global

The *store global* instructions move one or two words from the stack to the global frame.

SGB **Store Global Byte**

```
SGB: PROCEDURE =
    BEGIN
    alpha: BYTE = GetCodeByte[];
    StoreMds[GF + alpha] ↑ ← Pop[];
    END;
```

SGDB **Store Global Double Byte**

```
SGDB: PROCEDURE =
    BEGIN
    alpha: BYTE = GetCodeByte[];
    StoreMds[GF + alpha + 1] ↑ ← Pop[];
    StoreMds[GF + alpha] ↑ ← Pop[];
    END;
```

7.3 Pointer Instructions

The pointer instructions are divided into two types: direct and indirect. They move a word or pair of words between the stack and memory using a pointer obtained from the stack or from the local or global frame. Most pointer instructions have variants that dereference either short or long pointers.

Implementation Note: In the long-pointer variants of these instructions, any addition to the pointer must be calculated using double-word arithmetic, to account for the case in which ptr + offset may carry into the most significant word of the pointer.

7.3.1 Direct Pointer Instructions

The *direct pointer* instructions obtain a pointer from the stack and move a single or double word stack operand to or from the specified location. The pointer is usually modified by a small offset contained in the opcode or alpha.

7.3.1.1 Read Direct

The *read direct* instructions obtain a long or short pointer from the stack, add to it a small displacement from the opcode or alpha, and perform a single- or double-word push to the stack from memory.

Rn **Read n**

```
Rn: PROCEDURE [n: [0..1]] =
  BEGIN
  ptr: POINTER = Pop[];
  Push[FetchMds[ptr + n]↑];
  END;
```

RB **Read Byte**

```
RB: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
  ptr: POINTER = Pop[];
  Push[FetchMds[ptr + alpha]↑];
  END;
```

RL0 **Read Long Zero**

```
RL0: PROCEDURE =
  BEGIN
  ptr: LONG POINTER = PopLong[];
  Push[Fetch[ptr]↑];
  END;
```

RLB **Read Long Byte**

```
RLB: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
  ptr: LONG POINTER = PopLong[];
  Push[Fetch[ptr + LONG[alpha]]↑];
  END;
```

RD0 **Read Double Zero**

```
RD0: PROCEDURE =
  BEGIN
  ptr: POINTER = Pop[];
  u: UNSPECIFIED = FetchMds[ptr]↑;
```

```
    v: UNSPECIFIED = FetchMds[ptr + 1]↑;
    Push[u]; Push[v];
    END;
```

RDB **Read Double Byte**

```
  RDB: PROCEDURE =
    BEGIN
    alpha: BYTE = GetCodeByte[];
    ptr: POINTER = Pop[];
    u: UNSPECIFIED = FetchMds[ptr + alpha]↑;
    v: UNSPECIFIED = FetchMds[ptr + alpha + 1]↑;
    Push[u]; Push[v];
    END;
```

RDL0 **Read Double Long Zero**

```
  RDL0: PROCEDURE =
    BEGIN
    ptr: LONG POINTER = PopLong[];
    u: UNSPECIFIED = Fetch[ptr]↑;
    v: UNSPECIFIED = Fetch[ptr + 1]↑;
    Push[u]; Push[v];
    END;
```

RDLB **Read Double Long Byte**

```
  RDLB: PROCEDURE =
    BEGIN
    alpha: BYTE = GetCodeByte[];
    ptr: LONG POINTER = PopLong[];
    u: UNSPECIFIED = Fetch[ptr + LONG[alpha]]↑;
    v: UNSPECIFIED = Fetch[ptr + LONG[alpha] + 1]↑;
    Push[u]; Push[v];
    END;
```

RC **Read Code**

```
  RC: PROCEDURE =
    BEGIN
    alpha: BYTE = GetCodeByte[];
    offset: CARDINAL = Pop[];
    Push[ReadCode[offset + alpha]];
    END;
```

7.3.1.2 WriteDirect

The *write direct* instructions obtain a long or short pointer from the stack, add to it a small displacement from the opcode or alpha, and perform a single- or double-word pop from the stack to memory.

W0 **Write Zero**

```
W0: PROCEDURE =
  BEGIN
  ptr: POINTER = Pop[];
  StoreMds[ptr]↑ ← Pop[];
  END;
```

WB **Write Byte**

```
WB: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
  ptr: POINTER = Pop[];
  StoreMds[ptr + alpha]↑ ← Pop[];
  END;
```

WLB **Write Long Byte**

```
WLB: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
  ptr: LONG POINTER = PopLong[];
  Store[ptr + LONG[alpha]]↑ ← Pop[];
  END;
```

WDB **Write Double Byte**

```
WDB: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
  ptr: POINTER = Pop[];
  StoreMds[ptr + alpha + 1]↑ ← Pop[];
  StoreMds[ptr + alpha]↑ ← Pop[];
  END;
```

WDLB **Write Double Long Byte**

```
WDLB: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
  ptr: LONG POINTER = PopLong[];
  Store[ptr + LONG[alpha] + 1]↑ ← Pop[];
  Store[ptr + LONG[alpha]]↑ ← Pop[];
  END;
```

7.3.1.3 Put Swapped Direct

The *put swapped direct* instructions obtain a short pointer from the stack, add to it a small displacement from the opcode or alpha, and perform a single- or double-word store to memory. They leave the pointer on the stack for use by subsequent instructions.

("Swapped" refers to the order of the address and the data on the stack. The data is given first (from the TOS down) followed by the address for swapped instructions).

PSB Put Swapped Byte

```
PSB: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
  u: UNSPECIFIED = Pop[];
  ptr: POINTER = Pop[];
  StoreMds[ptr + alpha]↑ ← u;
  Recover[];
  END;
```

PSD0 Put Swapped Double Zero

```
PSD0: PROCEDURE =
  BEGIN
  v: UNSPECIFIED = Pop[];
  u: UNSPECIFIED = Pop[];
  ptr: POINTER = Pop[];
  StoreMds[ptr + 1]↑ ← v;
  StoreMds[ptr]↑ ← u;
  Recover[];
  END;
```

PSDB Put Swapped Double Byte

```
PSDB: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
  v: UNSPECIFIED = Pop[];
  u: UNSPECIFIED = Pop[];
  ptr: POINTER = Pop[];
  StoreMds[ptr + alpha + 1]↑ ← v;
  StoreMds[ptr + alpha]↑ ← u;
  Recover[];
  END;
```

PSLB Put Swapped Long Byte

```
PSLB: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
  u: UNSPECIFIED = Pop[];
  ptr: LONG POINTER = PopLong[];
  Store[ptr + LONG[alpha]]↑ ← u;
  Recover[];
  Recover[];
  END;
```

PSDLB **Put Swapped Double Long Byte**

```
PSDLB: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
  v: UNSPECIFIED = Pop[];
  u: UNSPECIFIED = Pop[];
  ptr: LONG POINTER = PopLong[];
  Store[ptr + LONG[alpha] + 1]↑ ←v;
  Store[ptr + LONG[alpha]]↑ ←u;
  Recover[];
  Recover[];
  END;
```

7.3.2 Indirect Pointer Instructions

The *indirect pointer* instructions obtain a pointer from the local or global frame or the stack and move a single-space or double-word stack operand to or from the specified location. The pointer is modified by a small offset contained in the opcode or alpha.

7.3.2.1 Read Indirect

The *read indirect* instructions perform a single- or double-word push using a pointer obtained from the local or global frame. Most of these instructions treat alpha as a pair; pair.left specifies the offset of the pointer in the frame, and pair.right is added to the pointer.

RLI0n **Read Local Indirect Zero n**

```
RLI0n: PROCEDURE [n: [0..3]] =
  BEGIN
  ptr: POINTER = FetchMds[LF]↑;
  Push[FetchMds[ptr + n]↑];
  END;
```

RLIP **Read Local Indirect Pair**

```
RLIP: PROCEDURE =
  BEGIN
  pair: NibblePair = GetCodeByte[];
  ptr: POINTER = FetchMds[LF + pair.left]↑;
  Push[FetchMds[ptr + pair.right]↑];
  END;
```

RLILP **Read Local Indirect Long Pair**

```
RLILP: PROCEDURE =
  BEGIN
  pair: NibblePair = GetCodeByte[];
  ptr: LONG POINTER = ReadDblMds[LF + pair.left];
```

```
    Push[Fetch[ptr + LONG[pair.right]]↑];
    END;
```

RGIP **Read Global Indirect Pair**

```
RGIP: PROCEDURE =
    BEGIN
    pair: NibblePair = GetCodeByte[];
    ptr: POINTER = FetchMds[GF + pair.left]↑;
    Push[FetchMds[ptr + pair.right]↑];
    END;
```

RGILP **Read Global Indirect Long Pair**

```
RGILP: PROCEDURE =
    BEGIN
    pair: NibblePair = GetCodeByte[];
    ptr: LONG POINTER = ReadDblMds[GF + pair.left];
    Push[Fetch[ptr + LONG[pair.right]]↑];
    END;
```

RLDI00 **Read Local Double Indirect Zero Zero**

```
RLDI00: PROCEDURE =
    BEGIN
    ptr: POINTER = FetchMds[LF]↑;
    u: UNSPECIFIED = FetchMds[ptr]↑;
    v: UNSPECIFIED = FetchMds[ptr + 1]↑;
    Push[u];  Push[v];
    END;
```

RLDIP **Read Local Double Indirect Pair**

```
RLDIP: PROCEDURE =
    BEGIN
    pair: NibblePair = GetCodeByte[];
    ptr: POINTER = FetchMds[LF + pair.left]↑;
    u: UNSPECIFIED = FetchMds[ptr + pair.right]↑;
    v: UNSPECIFIED = FetchMds[ptr + pair.right + 1]↑;
    Push[u];  Push[v];
    END;
```

RLDILP **Read Local Double Indirect Long Pair**

```
RLDILP: PROCEDURE =
    BEGIN
    pair: NibblePair = GetCodeByte[];
    ptr: LONG POINTER = ReadDblMds[LF + pair.left];
    u: UNSPECIFIED = Fetch[ptr + LONG[pair.right]]↑;
    v: UNSPECIFIED = Fetch[ptr + LONG[pair.right] + 1]↑;
    Push[u];  Push[v];
    END;
```

7.3.2.2 Write Indirect

The *write indirect* instructions perform a single- or double-word pop to memory using a pointer obtained from the local frame. They treat alpha as a pair, in which pair.left specifies the offset of the pointer in the frame, and pair.right is added to the pointer.

WLIP **Write Local Indirect Pair**

```
WLIP: PROCEDURE =
  BEGIN
  pair: NibblePair = GetCodeByte[];
  ptr: POINTER = FetchMds[LF + pair.left]↑;
  StoreMds[ptr + pair.right]↑ ← Pop[];
  END;
```

WLILP **Write Local Indirect Long Pair**

```
WLILP: PROCEDURE =
  BEGIN
  pair: NibblePair = GetCodeByte[];
  ptr: LONG POINTER = ReadDblMds[LF + pair.left];
  Store[ptr + LONG[pair.right]]↑ ← Pop[];
  END;
```

WLDILP **Write Local Double Indirect Long Pair**

```
WLDILP: PROCEDURE =
  BEGIN
  pair: NibblePair = GetCodeByte[];
  ptr: LONG POINTER = ReadDblMds[LF + pair.left];
  Store[ptr + LONG[pair.right] + 1]↑ ← Pop[];
  Store[ptr + LONG[pair.right]]↑ ← Pop[];
  END;
```

7.4 String Instructions

The *string* instructions read or write eight-bit bytes contained in packed arrays. The address of the word containing the byte is computed as the sum of a short or long pointer obtained from the stack plus a byte offset divided by two. The offset is the sum of an index taken from the stack and the instruction's alpha byte.

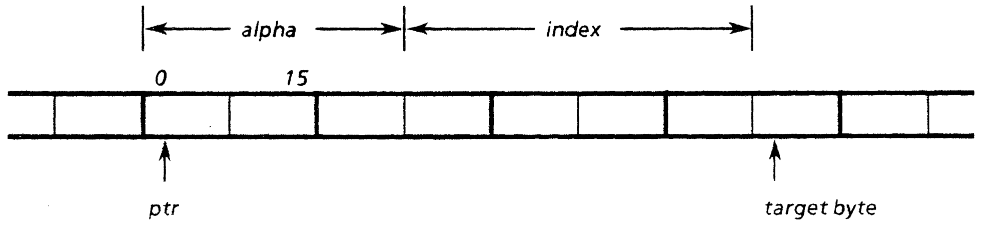


Figure 7.1 String Indexing

The least significant bit of the offset selects the byte that is read or written; zero specifies the most significant byte. The data byte is obtained from the stack, ignoring the high-order byte of the stack word, or written to the stack, clearing the high-order byte.

The following routines are used by the string instructions (and elsewhere) to fetch and store a byte:

```
FetchByte: PROCEDURE [ptr: LONG POINTER, offset: LONG CARDINAL]
  RETURNS [BYTE] =
  BEGIN
  word: BytePair = Fetch[ptr + offset/2]↑;
  RETURN[IF (offset MOD 2) = 0 THEN word.left ELSE word.right];
  END;

StoreByte: PROCEDURE [ptr: LONG POINTER, offset: LONG CARDINAL, data: BYTE] =
  BEGIN
  word: BytePair = Fetch[ptr + offset/2]↑;
  Store[ptr + offset/2]↑ ← IF (offset MOD 2) = 0
    THEN BytePair[data, word.right]
    ELSE BytePair[word.left, data];
  END;
```

Implementation Note: In the long-pointer variants of these instructions, the offset must be calculated using double-word arithmetic, to account for the case in which alpha + index may carry into the most significant word of the pointer.

7.4.1 Read String

The read string instructions clear the high-order byte of the data word written to the stack.

RS **Read String**

```
RS: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
  index: CARDINAL = Pop[];
  ptr: POINTER = Pop[];
  Push[FetchByte[ptr: ptr, offset: alpha + index]];
  END;
```

RLS **Read Long String**

```
RLS: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
  index: CARDINAL = Pop[];
  ptr: LONG POINTER = PopLong[];
  Push[FetchByte[ptr: ptr, offset: LONG[alpha] + LONG[index]]];
  END;
```

7.4.2 Write String

The write string instructions ignore the high-order byte of the data word obtained from the stack.

WS **Write String**

```
WS: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
  index: CARDINAL = Pop[];
  ptr: POINTER = Pop[];
  data: BYTE = LowByte[Pop[]];
  StoreByte[ptr: ptr, offset: alpha + index, data: data];
  END;
```

WLS **Write Long String**

```
WLS: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
  index: CARDINAL = Pop[];
  ptr: LONG POINTER = PopLong[];
  data: BYTE = LowByte[Pop[]];
  StoreByte[ptr: ptr, offset: LONG[alpha] + LONG[index], data: data];
  END;
```

7.5 Field Instructions

The *field* instructions either read or write a field of a word in memory. The word is usually addressed by a short or long pointer found on the stack. The *read indirect* operations obtain the required pointer from the local frame.

The field is described by a *field specifier* or a *field descriptor*, which is usually found in the alpha and beta bytes of the instruction. The stack operations take their field descriptors from the stack. Field specifiers are defined as follows:

```
FieldSpec: TYPE = MACHINE DEPENDENT RECORD [
  pos (0: 0..3): NIBBLE,
  size (0: 4..7): NIBBLE];
```

The pos specifies the most significant bit of the field (the most significant bit of a word is bit zero), and size is one less than the width of the field in bits (a field never has zero width). Figure 7.2 illustrates some examples of field specifiers.

Note that fields described by field specifiers do not cross word boundaries.

In addition to field specifiers, some instructions include an offset, a quantity added to the pointer to obtain the address of the word containing the field. This offset is included in a field descriptor.

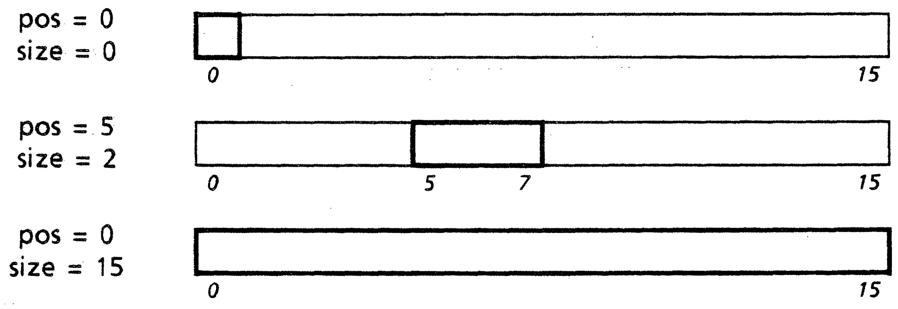


Figure 7.2 Field Specifiers

```
FieldDesc: TYPE = MACHINE DEPENDENT RECORD [
   offset (0: 0..7): BYTE,
   field (0: 8..15): FieldSpec];
```

The following routines are used by the field instructions (and elsewhere) to perform the basic functions of field extraction and insertion:

```
MaskTable: ARRAY [0..WordSize) OF UNSPECIFIED = [
   1, 3, 7, 17B, 37B, 77B, 177B, 377B, 777B, 1777B,
   3777B, 7777B, 17777B, 37777B, 77777B, 177777B];

ReadField: PROCEDURE [source: UNSPECIFIED, spec: FieldSpec]
   RETURNS [UNSPECIFIED] =
   BEGIN
   shift: CARDINAL[0..WordSize);
   IF spec.pos + spec.size + 1 > WordSize THEN ERROR;
   shift ← WordSize-(spec.pos + spec.size + 1);
   RETURN[And[Shift[source, -shift], MaskTable[spec.size]]];
   END;

WriteField: PROCEDURE [dest: UNSPECIFIED, spec: FieldSpec, data: UNSPECIFIED]
   RETURNS [UNSPECIFIED] =
   BEGIN
   mask: UNSPECIFIED;
   shift: CARDINAL[0..WordSize);
   IF spec.pos + spec.size + 1 > WordSize THEN ERROR;
   shift ← WordSize-(spec.pos + spec.size + 1);
   mask ← Shift[MaskTable[spec.size], shift];
   data ← And[Shift[data, shift], mask];
   RETURN[Or[And[dest, Not[mask]], data]];
   END;
```

Design Note: If an instruction contains a field specifier in which pos + size + 1 > WordSize, the results are undefined.

7.5.1 Read Field

The *read field* instructions push a field from a word in memory onto the stack. They right-justify the word and supply high-order zeros if necessary.

RF **Read Field**

```
RF: PROCEDURE =
  BEGIN
  desc: FieldDesc = GetCodeWord[];
  ptr: POINTER = Pop[];
  Push[ReadField[FetchMds[ptr + desc.offset] ↑, desc.field]];
  END;
```

R0F **Read Zero Field**

```
R0F: PROCEDURE =
  BEGIN
  spec: FieldSpec = GetCodeByte[];
  ptr: POINTER = Pop[];
  Push[ReadField[FetchMds[ptr] ↑, spec]];
  END;
```

RLF **Read Long Field**

```
RFL: PROCEDURE =
  BEGIN
  desc: FieldDesc = GetCodeWord[];
  ptr: LONG POINTER = PopLong[];
  Push[ReadField[Fetch[ptr + LONG[desc.offset]] ↑, desc.field]];
  END;
```

RL0F **Read Long Zero Field**

```
RL0F: PROCEDURE =
  BEGIN
  spec: FieldSpec = GetCodeByte[];
  ptr: LONG POINTER = PopLong[];
  Push[ReadField[Fetch[ptr] ↑, spec]];
  END;
```

RLFS **Read Long Field Stack**

```
RLFS: PROCEDURE =
  BEGIN
  desc: FieldDesc = Pop[];
  ptr: LONG POINTER = PopLong[];
  Push[ReadField[Fetch[ptr + LONG[desc.offset]] ↑, desc.field]];
  END;
```

RCFS **Read Code Field Stack**

```
RCFS: PROCEDURE =
  BEGIN
  desc: FieldDesc = Pop[];
  offset: CARDINAL = Pop[];
  Push[ReadField[ReadCode[offset + desc.offset], desc.field]];
  END;
```

Programming Note: The Read Code Field Stack instruction is used for accessing constant structures (arrays and records) located in the current code segment when the offset of the word containing the field is not constant.

RLIPF **Read Local Indirect Pair Field**

```
RLIPF: PROCEDURE =
  BEGIN
  pair: NibblePair = GetCodeByte[];
  spec: FieldSpec = GetCodeByte[];
  ptr: POINTER = FetchMds[LF + pair.left]↑;
  Push[ReadField[FetchMds[ptr + pair.right]↑, spec]];
  END;
```

RLILPF **Read Local Indirect Long Pair Field**

```
RLILPF: PROCEDURE =
  BEGIN
  pair: NibblePair = GetCodeByte[];
  spec: FieldSpec = GetCodeByte[];
  ptr: LONG POINTER = ReadDblMds[LF + pair.left];
  Push[ReadField[Fetch[ptr + LONG[pair.right]]↑, spec]];
  END;
```

7.5.2 Write Field

The *write field* instructions pop a value from the stack into a field of a word in memory. The value is right-justified in the field, ignoring leftover significant bits. Write Swapped Zero Field takes the pointer and the data in the opposite order on the stack, so that the pointer can be obtained using a Recover instruction.

WF **Write Field**

```
WF: PROCEDURE =
  BEGIN
  desc: FieldDesc = GetCodeWord[];
  ptr: POINTER = Pop[];
  data: UNSPECIFIED = Pop[];
  StoreMds[ptr + desc.offset]↑ ←WriteField[
    FetchMds[ptr + desc.offset]↑, desc.field, data];
  END;
```

W0F Write Zero Field

```
W0F: PROCEDURE =
  BEGIN
  spec: FieldSpec = GetCodeByte[];
  ptr: POINTER = Pop[];
  data: UNSPECIFIED = Pop[];
  StoreMds[ptr]↑ ← WriteField[FetchMds[ptr]↑, spec, data];
  END;
```

WLF Write Long Field

```
WLF: PROCEDURE =
  BEGIN
  desc: FieldDesc = GetCodeWord[];
  ptr: LONG POINTER = PopLong[];
  data: UNSPECIFIED = Pop[];
  Store[ptr + LONG[desc.offset]]↑ ← WriteField[
    Fetch[ptr + LONG[desc.offset]]↑, desc.field, data];
  END;
```

WL0F Write Long Zero Field

```
WL0F: PROCEDURE =
  BEGIN
  spec: FieldSpec = GetCodeByte[];
  ptr: LONG POINTER = PopLong[];
  data: UNSPECIFIED = Pop[];
  Store[ptr]↑ ← WriteField[Fetch[ptr]↑, spec, data];
  END;
```

WLFS Write Long Field Stack

```
WLFS: PROCEDURE =
  BEGIN
  desc: FieldDesc = Pop[];
  ptr: LONG POINTER = PopLong[];
  data: UNSPECIFIED = Pop[];
  Store[ptr + LONG[desc.offset]]↑ ← WriteField[
    Fetch[ptr + LONG[desc.offset]]↑, desc.field, data];
  END;
```

WS0F Write Swapped Zero Field

```
WS0F: PROCEDURE =
  BEGIN
  spec: FieldSpec = GetCodeByte[];
  data: UNSPECIFIED = Pop[];
  ptr: POINTER = Pop[];
  StoreMds[ptr]↑ ← WriteField[FetchMds[ptr]↑, spec, data];
  END;
```

7.5.3 Put Swapped Field

The *put swapped field* instructions leave the pointer on the top of the stack.

PS0F **Put Swapped Zero Field**

```
PS0F: PROCEDURE =
  BEGIN
  WS0F[];
  Recover[];
  END;
```

PSF **Put Swapped Field**

```
PSF: PROCEDURE =
  BEGIN
  desc: FieldDesc = GetCodeWord[];
  data: UNSPECIFIED = Pop[];
  ptr: POINTER = Pop[];
  StoreMds[ptr + desc.offset]↑ ← WriteField[
    FetchMds[ptr + desc.offset]↑, desc.field, data];
  Recover[];
  END;
```

PSLF **Put Swapped Long Field**

```
PSLF: PROCEDURE =
  BEGIN
  desc: FieldDesc = GetCodeWord[];
  data: UNSPECIFIED = Pop[];
  ptr: LONG POINTER = PopLong[];
  Store[ptr + LONG[desc.offset]]↑ ← WriteField[
    Fetch[ptr + LONG[desc.offset]]↑, desc.field, data];
  Recover[];
  Recover[];
  END;
```

8

Block Transfers

The block transfer instructions move multiword structures from a source address to a destination address, or they compare two multiword structures for equality. They include word block transfers, word block comparisons, byte block transfer, bit block transfer, and text block transfer. The last two operations are designed specifically for manipulating rectangles and text on a bitmap display.

Because of potentially long execution times, all of the block transfer instructions check for pending interrupts (§4.6.2). When a wakeup is detected, they save their intermediate state on the stack and back up the PC so that, when the instruction is restarted, it will continue transferring from the point of interruption. The check for interrupts is made once per iteration of the main loop (the InterruptPending routine is defined in §10.4.4). An implementation of the processor may make this check less often, if the frequency is consistent with the interrupt latency requirements in §10.4.4.1.

8.1 Word Boundary Block Transfers

The word block transfer instructions pop a count along with (short or long) source and destination pointers from the stack. They move words from the source to the destination. If the source and destination addresses are the same, there will still be a transfer.

Block Transfer and Block Transfer Long move words from the source to the destination in the forward direction (from low to high addresses). If the source and destination blocks overlap and the destination address is greater than the source address, words must be transferred one at a time from the source into the overlap area. This method causes words in the non-overlap area to be duplicated throughout the destination block. If the destination address is less than the source address or there is no overlap, then words do not have to be transferred one at a time, allowing possible speed improvements.

BLT **Block Transfer**

```
BLT: PROCEDURE =
  BEGIN
  DO
    dest: POINTER = Pop[];
    count: CARDINAL = Pop[];
    source: POINTER = Pop[];
```

```
        IF count = 0 THEN EXIT;
        StoreMds[dest] ↑ ← FetchMds[source] ↑ ;
        Push[source + 1];
        Push[count-1];
        Push[dest + 1];
        IF InterruptPending[] THEN GOTO Suspend;
        REPEAT
          Suspend = > PC ← savedPC;
        ENDLOOP;
      END;
```

BLTL **Block Transfer Long**

In Block Transfer Long, the source and destination addresses are long pointers.

```
    BLTL: PROCEDURE =
      BEGIN
      DO
        dest: LONG POINTER = PopLong[];
        count: CARDINAL = Pop[];
        source: LONG POINTER = PopLong[];
        IF count = 0 THEN EXIT;
        Store[dest] ↑ ← Fetch[source] ↑ ;
        PushLong[source + 1];
        Push[count-1];
        PushLong[dest + 1];
        IF InterruptPending[] THEN GOTO Suspend;
        REPEAT
          Suspend = > PC ← savedPC;
        ENDLOOP;
      END;
```

Block Transfer Long Reversed moves words from the source to the destination in the backward direction (from high to low addresses). If the source and destination blocks overlap and the destination address is less than the source address, words must be transferred one at a time from the source into the overlap area, causing words in the non-overlap area to be duplicated throughout the destination block. As with BLT and BLTL, If the destination address is less than the source address or there is no overlap, then words do not have to be transferred one at a time.

BLTLR **Block Transfer Long Reversed**

```
    BLTLR: PROCEDURE =
      BEGIN
      DO
        dest: LONG POINTER = PopLong[];
        count: CARDINAL = Pop[];
        source: LONG POINTER = PopLong[];
        IF count = 0 THEN EXIT;
        Store[dest + count] ↑ ← Fetch[source + count] ↑ ;
        PushLong[source];
        Push[count-1];
      PushLong[dest];
      IF InterruptPending[] THEN GOTO Suspend;
      REPEAT
         Suspend = > PC ← savedPC;
      ENDLOOP;
   END;
```

The Block Transfer Code and Block Transfer Code Long instructions move words from a source block in the code segment addressed by an offset from the current code base CB. In Block Transfer Code the destination address is a short pointer. In Block Transfer Code Long the destination address is a long pointer. The ReadCode routine is defined in §3.1.4.3.

BLTC **Block Transfer Code**

```
BLTC: PROCEDURE =
   BEGIN
   DO
      dest: POINTER = Pop[];
      count: CARDINAL = Pop[];
      source: CARDINAL = Pop[];
      IF count = 0 THEN EXIT;
      StoreMds[dest]↑ ← ReadCode[source];
      Push[source + 1];
      Push[count-1];
      Push[dest + 1];
      IF InterruptPending[] THEN GOTO Suspend;
      REPEAT
         Suspend = > PC ← savedPC;
      ENDLOOP;
   END;
```

BLTCL **Block Transfer Code Long**

```
BLTCL: PROCEDURE =
   BEGIN
   DO
      dest: LONG POINTER = PopLong[];
      count: CARDINAL = Pop[];
      source: CARDINAL = Pop[];
      IF count = 0 THEN EXIT;
      Store[dest]↑ ← ReadCode[source];
      Push[source + 1];
      Push[count-1];
      PushLong[dest + 1];
      IF InterruptPending[] THEN GOTO Suspend;
      REPEAT
         Suspend = > PC ← savedPC;
      ENDLOOP;
   END;
```

Implementation Note: If an interrupt occurs, the value of the (virtual) code base may be different when the instruction is resumed; therefore, it should not be part of the intermediate state saved on the stack.

Implementation Note: Since code segments are read-only, there can be no overlap in the block transfer code instructions. Therefore words do not have to be transfered one at a time, nor do they have to be transfered in the forward direction.

CKSUM Checksum

The Checksum instruction incrementally updates a single word checksum based on the contents of the source block. The updated checksum is returned on the stack.

```
CKSUM: PROCEDURE =
  BEGIN
  cksum: CARDINAL;
  DO
    source: LONG POINTER = PopLong[];
    count: CARDINAL = Pop[];
    cksum ← Pop[];
    IF count = 0 THEN EXIT;
    Push[Checksum[cksum, Fetch[source]↑]];
    Push[count-1];
    PushLong[source + 1];
    IF InterruptPending[] THEN GOTO Suspend;
    REPEAT
      Suspend = > PC ← savedPC;
      RETURN;
    ENDLOOP;
  IF cksum = 177777B THEN cksum ← 0;
  Push[cksum];
  END;
```

The checksum is a ones' complement add-and-left-cycle as computed by the following routine.

```
Checksum: PROCEDURE [chksum: CARDINAL, data: CARDINAL] RETURNS [CARDINAL] =
  BEGIN
  temp: CARDINAL;
  temp ← chksum + data;
  IF chksum > temp THEN temp ← temp + 1;
  IF temp > = 100000B THEN temp ← temp*2 + 1 ELSE temp ← temp*2;
  RETURN[temp];
  END;
```

8.2 Block Comparisons

The block comparison instructions pop a count and pointers from the stack. They compare two blocks of memory, returning TRUE if they are equal and FALSE otherwise. In Block Equal Long, the blocks are addressed by long pointers.

BLEL Block Equal Long

```
BLEL: PROCEDURE =
  BEGIN
  DO
    ptr1: LONG POINTER ← PopLong[];
    count: CARDINAL ← Pop[];
    ptr2: LONG POINTER ← PopLong[];
    IF count = 0 THEN
      BEGIN Push[TRUE]; EXIT; END;
    IF Fetch[ptr1]↑ # Fetch[ptr2]↑ THEN
      BEGIN Push[FALSE]; EXIT; END;
    PushLong[ptr2 + 1];
    Push[count-1];
    PushLong[ptr1 + 1];
    IF InterruptPending[] THEN GOTO Suspend;
    REPEAT
      Suspend => PC ← savedPC;
    ENDLOOP;
  END;
```

BLECL Block Equal Code Long

In Block Equal Code Long, one block is addressed by a long pointer and the other is addressed by an offset from the current code base **CB**. (The ReadCode routine used here is defined in §3.1.4.3.)

```
BLECL: PROCEDURE =
  BEGIN
  DO
    ptr: LONG POINTER ← PopLong[];
    count: CARDINAL ← Pop[];
    offset: CARDINAL ← Pop[];
    IF count = 0 THEN
      BEGIN Push[TRUE]; EXIT; END;
    IF Fetch[ptr]↑ # ReadCode[offset]↑ THEN
      BEGIN Push[FALSE]; EXIT; END;
    Push[offset + 1];
    Push[count-1];
    PushLong[ptr + 1];
    IF InterruptPending[] THEN GOTO Suspend;
    REPEAT
      Suspend => PC ← savedPC;
    ENDLOOP;
  END;
```

Implementation Note: If an interrupt occurs, the value of the (virtual) code base may be different when the instruction is resumed; therefore, it should not be part of the intermediate state saved on the stack.

Implementation Note: In the block comparison instructions, it does not matter whether words are fetched in forward or backward order.

8.3 Byte Boundary Block Transfers

The byte block transfer instructions pop a count, short source and destination offsets, and long source and destination pointers from the stack. They move bytes from the source to the destination. If the source and destination addresses are the same, there will still be a transfer.

Byte Block Transfer moves bytes from the source to the destination in the forward direction (from low to high addresses). If the source and destination blocks overlap and the destination address is greater than the source address, bytes must be transferred one at a time from the source into the overlap area, so bytes in the non-overlap area are duplicated throughout the destination block. If the destination address is less than the source address or there is no overlap, then bytes do not have to be transferred one at a time. Some speed improvements become possible as a result.

The FetchByte and StoreByte routines are defined in §7.4.

BYTBLT Byte Block Transfer

```
BYTBLT: PROCEDURE =
  BEGIN
  DO
    sourceOffset: CARDINAL ← Pop[];
    sourceBase: LONG POINTER ← PopLong[];
    count: CARDINAL ← Pop[];
    destOffset: CARDINAL ← Pop[];
    destBase: LONG POINTER ← PopLong[];
    IF count = 0 THEN EXIT;
    sourceBase ← sourceBase + LONG[sourceOffset/2];
    sourceOffset ← sourceOffset MOD 2;
    destBase ← destBase + LONG[destOffset/2];
    destOffset ← destOffset MOD 2;
    StoreByte[
      destBase, LONG[destOffset], FetchByte[sourceBase, LONG[sourceOffset]]];
    IF sourceOffset = 1 THEN
      BEGIN sourceBase ← sourceBase + 1; sourceOffset ← 0; END
    ELSE sourceOffset ← 1;
    IF destOffset = 1 THEN
      BEGIN destBase ← destBase + 1; destOffset ← 0; END
    ELSE destOffset ← 1;
    PushLong[destBase];
    Push[destOffset];
    Push[count-1];
    PushLong[sourceBase];
    Push[sourceOffset];
    IF InterruptPending[] THEN GOTO Suspend;
    REPEAT
      Suspend => PC ← savedPC;
    ENDLOOP;
  END;
```

Byte Block Transfer Reversed moves bytes from the source to the destination in the backward direction (from high to low addresses). As with other Block Transfer instructions, if the source and destination blocks overlap and the destination address is less than the source address, bytes must be transferred one at a time from the source into the overlap area. Bytes in the non-overlap area are duplicated throughout the destination block. However, if the destination address is less than the source address or there is no overlap, bytes do not have to be transferred one at a time.

BYTBLTR Byte Block Transfer Reversed

```
BYTBLTR: PROCEDURE =
  BEGIN
  DO
    sourceOffset: CARDINAL ← Pop[];
    sourceBase: LONG POINTER ← PopLong[];
    count: CARDINAL ← Pop[];
    destOffset: CARDINAL ← Pop[];
    destBase: LONG POINTER ← PopLong[];
    IF count = 0 THEN EXIT;
    sourceBase ← sourceBase + LONG[sourceOffset/2];
    sourceOffset ← sourceOffset MOD 2;
    destBase ← destBase + LONG[destOffset/2];
    destOffset ← destOffset MOD 2;
    StoreByte[
      destBase, LONG[destOffset], FetchByte[sourceBase, LONG[sourceOffset]]];
    IF sourceOffset = 0 THEN
      BEGIN sourceBase ← sourceBase - 1; sourceOffset ← 1; END
    ELSE sourceOffset ← 0;
    IF destOffset = 0 THEN
      BEGIN destBase ← destBase - 1; destOffset ← 1; END
    ELSE destOffset ← 0;
    PushLong[destBase];
    Push[destOffset];
    Push[count-1];
    PushLong[sourceBase];
    Push[sourceOffset];
    IF InterruptPending[] THEN GOTO Suspend;
    REPEAT
      Suspend = > PC ← savedPC;
    ENDLOOP;
  END;
```

8.4 Bit Boundary Block Transfers

The bit boundary block transfer instructions include Bit Block Transfer (BITBLT), for operating on rectangular arrays of bits in memory, and Text Block Transfer (TXTBLT), for converting arrays of characters into their bitmap representations.

8.4.1 Bit Transfer Utilities

The transfer instructions described below operate on arbitrary bit boundaries. The following structure is used to address bits:

```
BitAddress: TYPE = MACHINE DEPENDENT RECORD [
  word (0): LONG POINTER,
  reserved (2: 0..11): [0..7777B] ← 0,
  bit (2: 12..15): [0..WordSize)];
```

The Bump routine is used to increment (or decrement) a bit address by a bit offset.

```
Bump: PROCEDURE [address: BitAddress, offset: LONG INTEGER] RETURNS [BitAddress] =
  BEGIN
  offset ← offset + LONG[address.bit];
  RETURN[
    BitAddress[
      word: address.word + LongArithShift[offset, -Log[WordSize]],
      bit: And[LowHalf[offset], WordSize-1]]];
  END;
```

Implementation Note: Because the reserved field of a bit address is guaranteed to be zero, the extraction address.bit can be replaced by a word access.

The following routines are used to read and write individual bits within a word. The source (destination) is specified by a base bit address and a bit offset. The ReadField and WriteField routines are defined in §7.5.

```
ReadBit: PROCEDURE [address: BitAddress, offset: INTEGER] RETURNS [BIT] =
  BEGIN
  spec: FieldSpec;
  address ← Bump[address, LONG[offset]];
  spec ← FieldSpec[pos: address.bit, size: 0];
  RETURN[ReadField[Fetch[address.word]↑, spec]];
  END;

WriteBit: PROCEDURE [address: BitAddress, offset: INTEGER, bit: BIT] =
  BEGIN
  spec: FieldSpec;
  word: UNSPECIFIED;
  address ← Bump[address, LONG[offset]];
  word ← Fetch[address.word]↑;
  spec ← FieldSpec[pos: address.bit, size: 0];
  Store[address.word]↑ ← WriteField[word, spec, bit];
  END;
```

8.4.2 Bit Block Transfer

The BITBLT instruction manipulates rectangular arrays of bits. It accesses source bits and destination bits, performs a function on them, and stores the result in the destination bits.

Successive bit pairs are obtained by scanning a source bit stream and a destination bit stream. The instruction operates successively on lines of bits called *items*; it processes width bits from a pair of lines, and then moves down to the next item by adding srcBpl (bits per line) to the source address and dstBpl to the destination address. It continues until it has processed height lines.

Figure 8.1 illustrates a possible configuration of source and destination rectangles, which are always of the same size and dimensions, embedded in separate bitmaps. Approximately half of the items have been moved to the destination, and the location of the next item is highlighted in the source bitmap and shown as a dotted line in the destination bitmap.

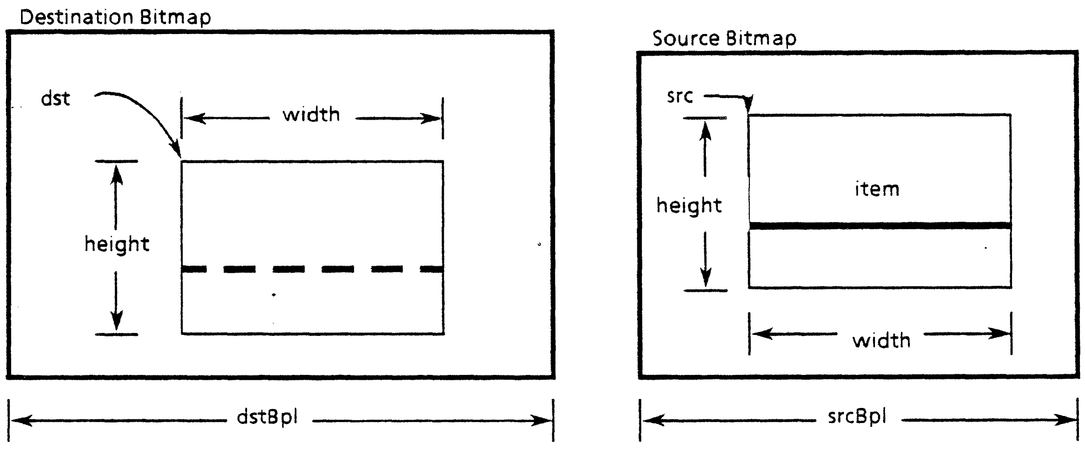


Figure 8.1 BitBlt Source and Destination

8.4.2.1. BitBlt Arguments

The argument to Bit Block Transfer is a short pointer to a record containing the source and destination bit addresses and bits per line, the width and height (in bits) of the rectangle to be operated on, and a word of flags that indicate the operation to be performed. The width and height of the rectangle are restricted to a maximum of 32,767. The argument record must be aligned on a sixteen-word boundary.

Note: Review the section on Gray Flag for the relationship between SrcBpl and the gray flag in BitBltFlags.

```
BitBltArg: TYPE = MACHINE DEPENDENT RECORD [
  dst (0): BitAddress,
  dstBpl (3): INTEGER,
  src (4): BitAddress,
  srcBpl (7): INTEGER,
  width (8): CARDINAL,
  height (9): CARDINAL,
  flags (10): BitBltFlags,
  reserved (11): UNSPECIFIED ← 0];
```

The flag bits specify the direction of the operation, the overlap of the operands, whether the source is interpreted as a gray brick, and the function to be performed on the source and destination bits.

```
BitBltFlags: TYPE = MACHINE DEPENDENT RECORD [
   direction (0: 0..0): Direction,
   disjoint (0: 1..1): BOOLEAN,
   disjointItems (0: 2..2): BOOLEAN,
   gray (0: 3..3): BOOLEAN,
   srcFunc (0: 4..4): SrcFunc,
   dstFunc (0: 5..6): DstFunc,
   reserved (0: 7..15): [0..777B] ←0];
```

Source and Destination Functions

The following routine describes the functions available for combining the source and destination rectangles (arg is the argument record). These functions are also shown in figure 8.2.

```
SrcFunc: TYPE = MACHINE DEPENDENT {null, complement};
DstFunc: TYPE = MACHINE DEPENDENT {null, and, or, xor};

Function: PROCEDURE [dst, src: BIT] RETURNS [BIT] =
   BEGIN
   src ← SELECT arg.flags.srcFunc FROM
      null = > src,
      complement = > Not[src],
      ENDCASE = > ERROR;
   dst ← SELECT arg.flags.dstFunc FROM
      null = > src,
      and = > And[src, dst],
      or = > Or[src, dst],
      xor = > Xor[src, dst],
      ENDCASE = > ERROR;
   RETURN[dst];
   END;
```

		dst			
		n	a	o	x
src	n	s	s·d	s + d	s⊕d
	c	~s	~s·d	~s + d	~s⊕d

Figure 8.2 Source and Destination Functions

The src field has two options; the null selection indicates using the source rectangle "as is" for the destination function. The complement selection will invert the source bits in the destination function.

The dst field determines the function to be used for changing bits in the destination rectangle. The null selection causes the destination to be "replaced" with the source bits. There is no boolean operation in this case. Anding the destination bits with the source bits leaves only those bits in common in the destination. "Painting" the destination requires oring. This operation will leave the union of the two sets of bits in the destination. The last function is the xor. It essentially masks out the matching bits leaving the union but not the intersection of the bits in the destination rectangle.

Direction Flag

The direction flag indicates whether the operation should take place forward (left to right, from low to high memory addresses) or backward (right to left, from high to low memory addresses). This allows an unambiguous specification of overlapping BitBlts, as in scrolling.

```
Direction: TYPE = MACHINE DEPENDENT {forward, backward};
```

If the direction is backward, the source and destination addresses point to the beginning of the *last* item of the blocks to be processed, and the source and destination bits per line must be *negative*. This restricts the width of the bitmaps involved to a maximum of 32,767 bits.

Adjustments of the arguments required by a change in direction are performed by the ComputeDirection routine which appears after the BITBLT opcode.

Disjoint Flag

If the operation's source and destination are completely disjoint, the implementation performs the operation in the most efficient horizontal and vertical directions, given by the following processor dependent variables:

```
xPreference, yPreference: Direction;
```

Both the direction and the disjointItems flags in the argument record are ignored when disjoint is set.

DisjointItems Flag

If the individual items of the source and destination are disjoint, but the rectangles otherwise overlap, the disjointItems flag should be set (and the disjoint flag should be clear). The implementation can then perform the operation so that, within each item, the bits are processed in the most efficient horizontal direction. The items are processed in the order indicated by direction.

If neither disjoint nor disjointItems is set, the implementation processes the items and the bits within items in the direction indicated by the direction flag.

Programming Note: Correct specification of disjoint and disjointItems is the responsibility of the programmer. The implementation makes no attempt to verify claims about overlapping source and destination arguments. If, in the course of instruction execution, a bit is used as a destination bit and then subsequently as a source bit, the results are undefined.

Gray Flag

The gray flag allows repetitive bit patterns to be specified in a condensed format. The usual application is for generation of various shades of gray on the display, but any repetitive pattern within the limits stated below may be supplied.

If the gray option is specified, the srcBpl field of the argument record is reinterpreted as follows:

```
GrayParm: TYPE = MACHINE DEPENDENT RECORD [
  reserved (0: 0..3): NIBBLE ← 0,
  yOffset (0: 4..7): NIBBLE,
  widthMinusOne (0: 8..11): NIBBLE,
  heightMinusOne (0: 12..15): NIBBLE];
```

The fields grayParm.widthMinusOne and grayParm.heightMinusOne define the width (less one) in words and height (less one) in bits, respectively, of a *gray brick* located at arg.src. Note, the term "brick" refers to a rectangular area containing the gray pattern to be copied. Conceptually, this brick is replicated horizontally and vertically to tile a plane of dimensions arg.width by arg.height. This plane becomes the source rectangle of the operation. The brick is a maximum of sixteen words wide and sixteen lines high. Patterns, therefore, are also limited to a repetition rate of sixteen in each direction. To guarantee correct repeatability of the pattern in the horizontal direction, the width of the gray brick (in bits) is usually a multiple of the repetition rate. The height of the gray brick is usually equal to the vertical repetition rate.

Proper alignment of the gray pattern with the destination bitmap requires the initial x- and y-offsets into the brick along with its width and height. The initial x-offset is derived from arg.src as follows: arg.src.word always points to the beginning of the first line to be transferred (not to the origin of the gray brick). The x-offset of the first bit to be transffered is supplied by arg.src.bit. This bit is always in the first word of the line. The initial y-offset is the number of lines down from the origin of the brick. It is specified by grayParm.yOffset. Subtracting the y-offset times the brick width from arg.src.word gives the origin of the gray brick.

Design Note: Since the brick is word-aligned and the repetition rate is sixteen or less, the initial x-offset can never exceed fifteen.

Design Note: The gray case is always forward and completely disjoint (disjointItems is ignored).

Design Note: Allowing grayParm.widthMinusOne to be greater than zero allows gray patterns having repetition rates of other than 1, 2, 4, 8, or 16 in the horizontal direction. Patterns with other repetition rates may be desirable, but are not mandatory. While the BitBlt code allows values greater than zero for grayParm.widthMinusOne, the initial implementation is restricted to a value of zero.

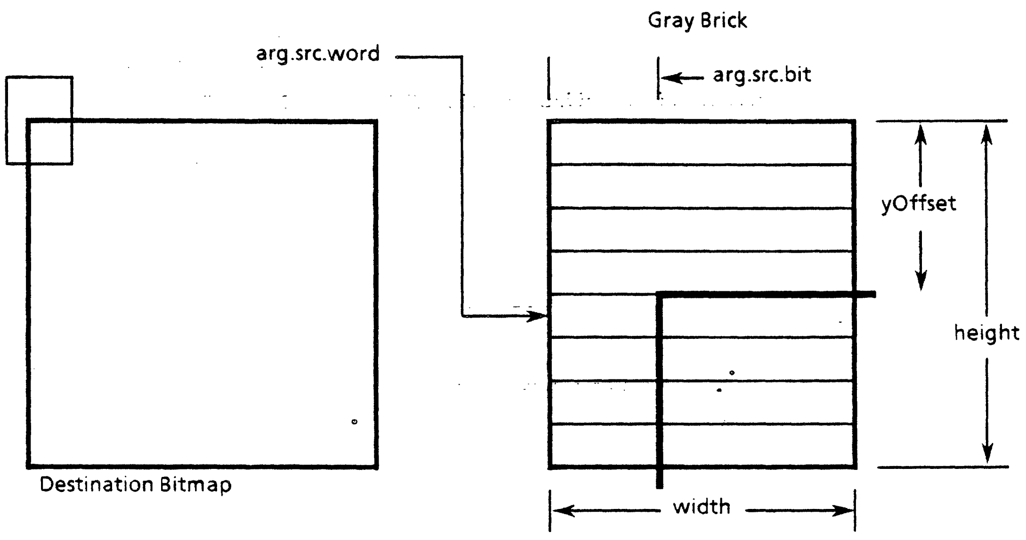


Figure 8.3 Gray Brick

8.4.2.2. Interruptibility

The Bit Block Transfer instruction checks for interrupts after it completes each item. If a pending interrupt is detected, the current state of the BITBLT is saved on the stack. When the instruction is restarted, the stack count is used to distinguish the restart case. The actual format of the stack is processor-dependent. The following routines are assumed to save and restore the intermediate state:

```
PushState, PopState: PROCEDURE;
```

Design Note: If any of the values of the arguments (in memory) change between the time of an interrupt and the subsequent restart of the instruction, the effects of the instruction are undefined. This allows the original values in the argument record to be saved as part of the intermediate state.

8.4.2.3. BitBlt Routines

BITBLT Bit Block Transfer

```
BITBLT: PROCEDURE =
  BEGIN
  line: CARDINAL;
  arg: BitBltArg;
  grayParm: GrayParm;
  lastGray: [0..15);
  grayWidth: INTEGER;
  grayBump: LONG INTEGER;
  xDirection, yDirection: Direction;
  IF StackCount[] = 1 THEN Setup[] ELSE PopState[];
  MinimalStack[];
  WHILE line IN [0..arg.height) DO
```

```
        BitBltItem[];
        arg.src ← Bump[arg.src,
          IF arg.flags.gray THEN
            IF (line MOD grayParm.heightMinusOne + 1) = lastGray
              THEN grayBump
              ELSE LONG[grayWidth]
          ELSE LONG[arg.srcBpl]];
        arg.dst ← Bump[arg.dst, LONG[arg.dstBpl]];
        line ← line + (IF yDirection = forward THEN 1 ELSE -1);
        IF InterruptPending[] THEN GO TO Suspend;
        REPEAT
          Suspend => {PushState[]; PC ← savedPC};
        ENDLOOP;
      END;

  BitBltItem: PROCEDURE =
      BEGIN
      offset, pos: INTEGER;
      offset ← IF xDirection = forward THEN 0 ELSE arg.width-1;
      THROUGH [0..arg.width) DO
        pos ← IF arg.flags.gray THEN
          ((offset + arg.src.bit) MOD ABS[grayWidth])-arg.src.bit ELSE offset;
        WriteBit[
          arg.dst, offset, Function[ReadBit[arg.dst, offset], ReadBit[arg.src, pos]]];
        offset ← offset + (IF xDirection = forward THEN 1 ELSE -1);
        ENDLOOP;
      END;
```

The routines given below are used to set up the BitBlt operation on first entry. They fetch the argument record, perform error checks on its fields, choose a direction, adjust the arguments accordingly, and compute the gray brick boundaries.

```
  Setup: PROCEDURE =
      BEGIN
      ptr: POINTER TO BitBltArg = Pop[];
      arg ← FetchBitBltArg[ptr];
      IF arg.flags.reserved # 0 OR arg.reserved # 0
      OR arg.src.reserved # 0 OR arg.dst.reserved # 0
      OR (~arg.flags.gray AND arg.srcBpl = 0) OR arg.dstBpl = 0
      OR arg.width > 32767 OR arg.height > 32767
        THEN ERROR;
      IF arg.flags.gray THEN
        BEGIN
        grayParm ← LOOPHOLE[arg.srcBpl];
        IF grayParm.widthMinusOne # 0
        OR grayParm.reserved # 0
        OR arg.flags.direction # forward
        OR ~arg.flags.disjoint OR arg.dstBpl < 0
          THEN ERROR;
        grayWidth ← INTEGER[(grayParm.widthMinusOne + 1)*WordSize];
        grayBump ← -grayWidth*grayParm.heightMinusOne;
        END
```

```
ELSE
   IF (arg.flags.direction = forward AND (arg.srcBpl < 0 OR arg.dstBpl < 0))
   OR (arg.flags.direction = backward AND (arg.srcBpl > 0 OR arg.dstBpl > 0))
      THEN ERROR;
ComputeDirection[];
IF arg.flags.gray THEN
   lastGray ← IF yDirection = forward
      THEN grayParm.heightMinusOne-grayParm.yOffset
      ELSE grayParm.yOffset;
line ← IF yDirection = forward THEN 0 ELSE arg.height-1;
END;

FetchBitBltArg: PROCEDURE [ptr: POINTER TO BitBltArg] RETURNS [BitBltArg] =
   BEGIN
   ArgIndex: TYPE = [0..SIZE[BitBltArg]);
   temp: ARRAY ArgIndex OF UNSPECIFIED;
   IF And[ptr, 17B] # 0 THEN ERROR;
   FOR i: ArgIndex IN ArgIndex DO
      temp[i] ← FetchMds[ptr + i]↑;
      ENDLOOP;
   RETURN[LOOPHOLE[temp]];
   END;

ComputeDirection: PROCEDURE =
   BEGIN
   yDirection ← xDirection ← arg.flags.direction;
   IF arg.flags.disjoint AND yDirection # yPreference THEN
      BEGIN
      yDirection ← yPreference;
      IF arg.flags.gray THEN
         BEGIN
         arg.src ← Bump[arg.src,
            grayWidth*((grayParm.yOffset + arg.height-1)
            MOD (grayParm.heightMinusOne + 1)-grayParm.yOffset)];
         grayWidth ← -grayWidth;
         grayBump ← -grayBump;
         END
      ELSE
         BEGIN
         arg.src ← Bump[arg.src, arg.srcBpl*(arg.height-1)];
         arg.srcBpl ← -arg.srcBpl;
         END;
      arg.dst ← Bump[arg.dst, arg.dstBpl*(arg.height-1)];
      arg.dstBpl ← -arg.dstBpl;
      END;
   IF arg.flags.disjoint OR arg.flags.disjointItems THEN xDirection ← xPreference;
   END;
```

Implementation Note: The products computed in Setup and ComputeDirection are thirty-two bit signed numbers (LONG INTEGERS) produced by multiplying an integer by a cardinal. Since the cardinal is known to be less than 32,768, a short-integer multiply can be used.

Implementation Note: Much of the complexity in ComputeDirection comes from reversing direction in the gray case. This can be avoided if yPreference is forward, since all legal gray BitBlts specify a forward direction.

8.4.3 Text Block Transfer

The Text Block Transfer instruction operates on an array of characters. It implements three functions useful for generating the font representation of the text in a bitmap.

Function: TYPE = MACHINE DEPENDENT {display, format, resolve, unused};

The format function is used to calculate the number of characters that will fit on a line and the number of spaces that may be added to the line, given its right margin (in micas). The display function converts characters to their font representation in the destination bitmap, optionally widening or narrowing pad characters to perform line justification. The resolve function is used to record the horizontal bit position of the origin of each character in the bitmap; it also handles justification. This function is used when determining which character in a text line has been selected with a pointing device.

Note: In the following section, the directional references used refer to the association with conventional Xerox bitmap displays. The top left of a CRT display is considered the point of origin for x- and y-coordinates. The x-coordinate increases horizontally from left to right across the screen. The y-coordinate increases vertically from the top of the screen to the bottom. So for instance, referring to the bit-position in left-to-right, top-to-bottom order is only for conceptual purposes.

8.4.3.1 Font Representation

The font determines the height and width (in bits) of the characters and the bit pattern to be transferred. The font also contains two flag bits for each character: the first specifies whether the character is a *pad character* (widenable for justification), and the second specifies whether the character is a *stop character* (terminating a TextBlt operation).

The precise format of the font is private to the implementation; the following types and routines are used in the TXTBLT code to access the font. FontRecord contains the font information TXTBLT needs. rasters indicates the font's raster specification. spacingWidths specifies the width of a character in bits. printerWidths gives the printing width of the character. The flags consist of pad and stop. pad is set to TRUE if the character is a pad character; stop is set to TRUE if the character is a stop character. rasterInfos includes the number of bits to the left or right of the character's origin and specifies the offset of the character's raster. height is the height of the font measured in bits; it is constant for all characters. To allow for kerned fonts, DisplayWidth returns the width of the entire font representation of the character, which includes the left and right kerning not included in the spacingWidth. Bit returns the individual bits of a character's font representation.

FontHandle points to the font information TextBlt needs. FontRecord is the concrete type of a Font. FontRecord must be aligned on a sixteen-word boundary.

```
Font: TYPE;
FontHandle: TYPE = LONG POINTER TO Font;

fontRecordAlignment: NATURAL = 16;

FontRecord: TYPE = MACHINE DEPENDENT RECORD [
   rasters(0): FontRasters,
   spacingWidths(2): SpacingWidths,
   printerWidths(4): PrinterWidths,
   flags(6): FlagsArray,
   rasterInfos(8): RasterInfos,
   height(10): CARDINAL];
```

The following types make up FontRecord:

```
FontBitsPtr: TYPE = LONG BASE POINTER TO ARRAY [0..0) OF UNSPECIFIED;

FontRasters: TYPE = LONG BASE POINTER TO <<rasters>> ARRAY [0..0) OF WORD;
```

The data at FontRasters is a base pointer for the character raster data. For a particular character, RasterInfo.offset (defined below) is added to this base to get the address of the character's raster.The raster format includes the scan lines within the dimensions given by spacingWidths and height. The scan lines are tightly packed together, so that the last bit of a scan line is immediately followed by the first bit of the next. The height of the raster is constant for all characters. Each raster begins on a word boundary.

The memory order of the bits in the raster correspond to the memory order in which TextBlt will paint them into the destination bitmap. Said another way, TextBlt paints the first scan line of the raster into the appropriate place in the first scan line of the destination bitmap, and so on. Similarly, the first bit of a raster's scan line is painted into the appropriate first bit of the scan line in the destination bitmap, and so on.

The first scan line in memory corresponds to the top line on the screen (of Xerox conventional bitmap displays). The first bit of the scan line corresponds to the left pixel of the line. For this case, the first scan line in the raster will be the topmost row of the character, and the first pixel (most significant bit) of a scan line will be the leftmost pixel of its row.

```
Byte: TYPE = CARDINAL [0..255];

SpacingWidths: TYPE = LONG POINTER TO PACKED ARRAY Byte OF SpacingWidth;
SpacingWidth: TYPE = Byte;

PrinterWidths: TYPE = LONG POINTER TO ARRAY Byte OF PrinterWidth;
PrinterWidth: TYPE = CARDINAL;

FlagsArray: TYPE = LONG POINTER TO PACKED ARRAY Byte OF Flags;

Flags: TYPE = MACHINE DEPENDENT RECORD [
   pad(0:0..0): BOOLEAN,
   stop(0:1..1): BOOLEAN];
```

The pad flag allows the character to have its width increased or decreased (in bits) for line justification. The stop flag will specify a stop character to terminate a TextBlt operation.

```
    RasterInfos: TYPE = LONG POINTER TO ARRAY Byte OF RasterInfo;

    RasterInfo: TYPE = MACHINE DEPENDENT RECORD [
      leftKern: BOOLEAN,
      rightKern: BOOLEAN,
      offset: RasterOffset];
```

If RasterInfo.leftKern = TRUE, the character's raster has one column preceeding the char's origin, and will be written into the destination bitmap with one column preceeding the current position (bitPos). If RasterInfo.rightKern = TRUE, the raster extends one column past the spacing width into the space for the next character; that character's raster should begin coincident with the current character's last column (one column preceeding where it would normally go). RasterInfo.offset is the offset for the address of the character's raster.

```
RasterOffsetDomain: TYPE = CARDINAL [0..37777B];
RasterOffset: TYPE = FontRasters RELATIVE POINTER [0..37777B] TO <<raster>> UNSPECIFIED;

RasterOffsetFromDomain: PROC [domain: RasterOffsetDomain]
   RETURNS [RasterOffset] = INLINE {RETURN[LOOPHOLE[domain]]};

RasterDomainFromOffset: PROC [offset: RasterOffset]
   RETURNS [RasterOffsetDomain] = INLINE {RETURN[LOOPHOLE[offset]]};

maxLeftKern: CARDINAL = 1;
maxRightKern: CARDINAL = 1;
```

MaxLeftKern and maxRightKern support kerning up to one pixel in the respective direction.

Design Note: Although the architecture permits it, the specification is not intended to encourage the creation of a different font format for each implementation of the processor. A new format may be specified only if significant performance improvement can be gained and is required.

8.4.3.2. TextBlt Arguments and Results

TextBlt's static arguments are passed via a short pointer to a record. The argument record must be aligned on a sixteen-word boundary. Arguments updated and returned by TextBlt are passed (and returned) on the stack (see the opcode description below).

```
    TxtBltArg: TYPE = MACHINE DEPENDENT RECORD [
      reserved (0: 0..13): [0..37777B] ← 0,
      function (0: 14..15): Function,
      last (1): CARDINAL,
      text (2): LONG POINTER TO ARRAY CARDINAL OF BytePair,
      font (4): FontHandle,
      dst (6): LONG POINTER,
      dstBpl (8): CARDINAL,
      margin (9): CARDINAL,
        space (10): INTEGER,
        coord (11): LONG POINTER TO ARRAY CARDINAL OF CARDINAL];
```

TextBlt proceeds through the characters of arg.text from index through arg.last unless a stop character is encountered (for example, note that index, shown in the TextBlt Routines section, is a byte offset). It maintains the bitPos (postion of the first bit of the character's raster) and the printPos (postion of the first bit of the printed character) of the origin of each character, and increments the count of the number of pad characters processed. During the format function, the scan is also terminated before the right arg.margin (in micas) is passed. The display function ors the character's font bits into the destination bitmap specified by arg.dst and arg.dstBpl (bits per line). The resolve function saves the bitPos of the origin of each character in the array arg.coord.

Programming Note: Because of kerning, the display function may place bits into the destination bitmap to the left of the bitPos of the leftmost character and to the right of the right margin. It is the programmer's responsibility to initialize the bitPos to allow for the left kerning of the first character, and to supply a bitmap wide enough to allow for the maximum possible right kerning. (At present this maximum is one bit.)

Justification can be accomplished using the display and resolve functions with appropriate settings of the arg.space and count values; arg.space is added to the width of every pad character (it may be negative), and count is incremented each time a pad character is encountered (it may also be initially negative). Since the amount of white space to be absorbed by (or squeezed out of) pad characters is rarely an even multiple of the number of pad characters, pad characters encountered have arg.space + 1 added to their widths as long as count is negative. Thus if sixteen bits need to be added to the width of the line in order to justify it, but it contains only thirteen pad characters, arg.space would be set to one, and count would be initialized to *negative* three. This operation will result in widening the first three pad characters by two bits each and the remaining ten pad characters by one bit each.

TextBlt returns, in place of the argument pointer on the stack, an indication of its completion condition: *normal* if the last character was processed, *margin* if the right margin was reached (format only), and *stop* if a terminating character was detected.

```
        Result: TYPE = MACHINE DEPENDENT {normal, margin, stop, unused};
```

The display font (arg.font), character array (arg.text), and destination bitmap (arg.dst) must not cross 64K boundries. (For bitmaps larger than 64K, the display lines can be created in a private buffer and transferred to the display bitmap using the Bit Block Transfer instruction.) As in BITBLT, the maximum value of dstBpl is 32,767. This limitation also applies to horizontal bit positions. The effects of the instruction are undefined if there is any overlap in memory among the arguments (arg), display font (arg.font), character array (arg.text), widths array (arg.coord), or the destination bitmap (arg.dst).

8.4.3.3. Interruptibility

The Text Block Transfer instruction checks for interrupts after it processes each character. As with all block transfers, the intermediate state of the operation is saved on the stack when an interrupt is detected. This saving operation consists of pushing the updated values of the original arguments. The actual format of the stack can be processor-dependent.

Design Note: If any of the values of the arguments (in memory) change between the time of an interrupt and the subsequent restart of the instruction, the effects of the instruction are undefined. The original values in the argument record are thereby allowed to be saved as part of the intermediate state.

8.4.3.4. TextBlt Routines

TXTBLT **Text Block Transfer**

```
TXTBLT: PROCEDURE =
  BEGIN
  result: Result;
  arg: TxtBltArg;
  font: FontRecord;
  ptr: POINTER TO TxtBltArg = Pop[];
  count: INTEGER ← Pop[];
  printPos: CARDINAL ← Pop[];
  bitPos: CARDINAL ← Pop[];
  index: CARDINAL ← Pop[];
  MinimalStack[];
  arg ← FetchTxtBltArg[ptr];
  IF arg.reserved # 0 OR arg.dstBpl > 32767 THEN ERROR;
  UNTIL index > arg.last DO
    char: BYTE;
    IF arg.function = resolve THEN Store[@arg.coord[index]]↑ ←bitPos;
    char ← FetchChar[arg.text, index];
    IF font.flags[char].stop THEN GO TO Stop;
    IF printPos + font.printerWidths[char] > arg.margin
      THEN GOTO Margin;
    IF arg.function = display THEN DisplayChar[bitPos, char];
    bitPos ← bitPos + font.spacingWidths[char];
    printPos ← printPos + font.printerWidths[char];
    IF font.flags[char].pad THEN
      BEGIN
      IF arg.function = (display OR format) THEN
        BEGIN
        bitPos ← bitPos + arg.space;
        IF count < 0 THEN bitPos ← bitPos + 1;
        END;
      count ← count + 1;
      END;
    index ← index + 1;
    IF InterruptPending[] THEN GO TO Suspend;
    REPEAT
      Suspend =>
        BEGIN
        PushState[ptr];
        PC ← savedPC;
```

```
            GO TO Done;
            END;
          Stop = > result ← stop;
          Margin = > result ← margin;
          FINISHED = > result ← normal;
        ENDLOOP;
      PushState[result];
      EXITS Done = > NULL;
      END;
```

The routines below are used by the TXTBLT code. They fetch the argument record, fetch a character from the text array, and move a character from the display font into the destination bitmap. The type BitAddress and the routines Bump, ReadBit, and WriteBit are defined in §8.4.1.

```
    FetchTxtBltArg: PROCEDURE [ptr: POINTER TO TxtBltArg] RETURNS [TxtBltArg] =
      BEGIN
      ArgIndex: TYPE = [0..SIZE[TxtBltArg]);
      temp: ARRAY ArgIndex OF UNSPECIFIED;
      IF And[ptr, 17B] # 0 THEN ERROR;
      FOR i: ArgIndex IN ArgIndex DO
        temp[i] ← FetchMds[ptr + i]↑;
        ENDLOOP;
      RETURN[LOOPHOLE[temp]];
      END;

    FetchChar: PROCEDURE [
        ptr: LONG POINTER TO ARRAY CARDINAL OF BytePair, index: CARDINAL]
      RETURNS [BYTE] =
      BEGIN
      data: BytePair = Fetch[@ptr[index/2]]↑;
      RETURN[IF (index MOD 2) = 0 THEN data.left ELSE data.right];
      END;

    DisplayChar: PROCEDURE [pos: CARDINAL, char: BYTE] =
      BEGIN
      count: CARDINAL ← 0;
      dst: BitAddress ← [word: arg.dst, bit: 0];
      width: CARDINAL = DisplayWidth[arg.font, char];
      pos ← pos - {IF font.rasterInfo[char].leftKern THEN 1 ELSE 0};
      THROUGH [0..font.height) DO
        FOR inc: CARDINAL IN [0..width) DO
          bit: BIT ← Bit[arg.font, char, count];
          offset: INTEGER ← INTEGER[pos + inc];
          WriteBit[dst, offset, Or[bit, ReadBit[dst, offset]]];
          count ← count + 1;
          ENDLOOP;
        dst ← Bump[dst, LONG[INTEGER[arg.dstBpl]]];
        ENDLOOP;
      END;
```

```
Bit: PROC [font: FontHandle, char: Byte, scanLine, pixel: CARDINAL] RETURNS [BIT] = {
  raster: LONG POINTER TO PACKED ARRAY OF BIT =
  LOOPHOLE[font.raster + font.rasterInfos[char].offset];
  bit: CARDINAL = scanLine*displayWidth[font, char] + pixel;
  RETURN[raster[bit]]};
```

[This definition of Bit has been recast above in terms of explicit scanline and pixel.]

```
DisplayWidth: PROC [font: FontHandle, char: Byte] RETURNS [CARDINAL] = {
  RETURN[font.spacingWidths[char]
   + (IF font.rasterInfo[char].leftKern THEN 1 ELSE 0)
   + (IF font.rasterInfo[char].rightKern THEN 1 ELSE 0)]};
```

Programming Note: The programmer should ensure that the calculation pos - {IF font.rasterInfo[char].leftKern THEN 1 ELSE 0} does not underflow,that is, the pos of the first character must allow for its left kerning. The programmer must also ensure that the maximum offset does not exceed 32,767.

Implementation Note: Because the destination bits per line does not exceed 32,767, conversion of arg.dstBpl to a long integer can be performed by supplying high-order zeros. Likewise, the conversion of pos + inc to an integer need not be range-checked.

Design Note: For short (or narrow) characters, considerable optimization of the DisplayChar inner loops is possible by adding information to the font format (the starting vertical location and the height of each character are examples). Because the character is ored into the destination, the white space surrounding the character need not actually be stored in the bitmap. Note, however, that such optimizations may substantially increase the amount of storage required for the font.

```
PushState: PROCEDURE [data: UNSPECIFIED] =
  BEGIN
  Push[index];
  Push[bitPos];
  Push[printPos];
  Push[count];
  Push[data];
  END;
```

PushState handles the stack both for the intermediate state (in the case of an interrupt) and for the final results of the instruction. In the former case the last item pushed is the pointer to the argument record, in the latter case the last item is the result of the TextBlt.

9 Control Transfers

Control transfers are a generalization of the notion of a procedure or subroutine call. In the Mesa architecture, there is a single primitive called XFER, which effects a control transfer from one context to another (§9.3). Variations of this primitive are used to implement procedure calls, nested procedure calls, returns, coroutine transfers, traps, and process switches. Included in XFER is a mechanism for the allocation and destruction of local frames (activation records). This mechanism is described in §9.2.

Contexts are created (and destroyed) by transfers of control, the most common of which is a procedure call. Instructions that implement various forms of procedure call (local, external, nested, etc.) and procedure return are described in §9.4. The processor also implements a general coroutine facility based on *ports* (§9.4.5). Ports allow contexts to transfer control without destroying their state. The ports mechanism may be used to implement, among other things, non-preemptive scheduling of contexts. (Preemptive scheduling is the subject of §10, which discusses the process mechanism.)

Strictly speaking, the contents of the evaluation stack are also part of the state of a context, but the stack is *not* saved or restored by a control transfer. (It is preserved by a process switch; see §10.4.2.) Instead, the stack is used to pass parameters and return results from one context to another. Before and after each transfer, the source and destination contexts must agree on the number and type of the stack elements. Because traps (§9.5) are also implemented as control transfers, there are cases in which the configuration of the stack is not known by the destination context. Therefore, certain instructions save and restore the contents of the evaluation stack and the stack pointer (§9.5.3). To implement a breakpoint mechanism for debugging, these instructions also preserve the context's break byte (§9.5.4).

Furthermore, control transfers do not modify the **MDS** (Main Data Space) or **PSB** (Process Status Block) registers. These registers are controlled by the process-switching mechanism (§10). Transfers of control are limited to contexts residing in the current Main Data Space.

9.1 Control Links

Contexts are represented by *control links*, which have one of three formats. The simplest form is a *frame link*, which is a pointer to a local frame. It represents a context as described above. An *indirect link* is a pointer to a control link, and is used to establish linkages for nested procedures and ports. An indirect link is converted to a context by dereferencing it. Finally, control links of type *procedure descriptor* are used to represent contexts that do not yet exist. They contain all the information necessary to create the context, as well as to transfer control to it after it has been created.

Control transfers take as an argument a destination control link. The least significant bits of the control link determine the type of transfer to be performed. They are encoded as follows:

```
ControlLink: TYPE = LONG UNSPECIFIED;

ShortControlLink: TYPE = UNSPECIFIED;

LinkType: TYPE = {frame, procedure, indirect};

TaggedControlLink: TYPE = MACHINE DEPENDENT RECORD [
  data (0: 0..13): [0..37777B],
  tag (0: 14..15): [0..3],
  fill (1): UNSPECIFIED];

ControlLinkType: PROCEDURE [link: ControlLink] RETURNS [LinkType] =
  BEGIN
  cl: TaggedControlLink = LOOPHOLE[link];
  RETURN[
    SELECT cl.tag FROM
      0 => frame,
      2 => indirect,
      ENDCASE => procedure];
  END;
```

The internal structure of each of the variants of a control link is described below. The use of control links during transfers of control is covered in the section on XFER (§9.3).

9.1.1 Frame Control Links

A *frame control link* is used to transfer control to an existing context (for example, a return from a procedure). The link is a pointer to the local frame of the context.

```
FrameLink: TYPE = LocalFrameHandle;

MakeFrameLink: PROCEDURE [link: ControlLink] RETURNS [FrameLink] =
  BEGIN
  IF ControlLinkType[link] # frame THEN ERROR;
  RETURN[LowHalf[link]];
  END;
```

Note that the frame handle points to the beginning of the frame variables, not to the overhead words. Frame links always point to local frames, never to global frames.

Programming Note: To ensure that the tag bits of a frame control link have the proper values, frames must be allocated at addresses that are zero modulo four.

Programming Note: The high-order word of a FrameLink is not used and may be left uninitialized by the programmer.

9.1.2 Indirect Control Links

An *indirect control link* is a short pointer to a control link.

```
IndirectLink: TYPE = POINTER TO ControlLink;

MakeIndirectLink: PROCEDURE [link: ControlLink] RETURNS [IndirectLink] =
  BEGIN
  IF ControlLinkType[link] # indirect THEN ERROR;
  RETURN[LowHalf[link]];
  END;
```

Indirect control links are used to establish linkages between contexts when calling nested procedures (§9.4.3) and ports (§9.4.5).

Programming Note: To ensure that the tag bits of an indirect control link have the proper values, control links pointed to by indirect links must be allocated at addresses that are two modulo four.

Programming Note: The high-order word of an IndirectLink is not used and may be left uninitialized by the programmer.

9.1.3 Procedure Descriptors

A *procedure descriptor* is used to create a new context. It contains the information necessary to obtain the global frame pointer **GF**, the code segment pointer **CB**, the local frame pointer **LF**, and the starting **PC** value for the procedure. It consists of two fields:

```
ProcDesc: TYPE = MACHINE DEPENDENT RECORD [
  taggedGF(0):UNSPECIFIED,
  pc (1): CARDINAL];

MakeProcDesc: PROCEDURE [link: ControlLink] RETURNS [ProcDesc] =
  BEGIN
  IF ControlLinkType[link] # procedure THEN ERROR;
  RETURN[LOOPHOLE[link]];
  END;
```

The taggedGF is the global frame pointer or'ed with 1. This result becomes the new value of **GF**. The code base is then obtained from the global frame using **GF**. The **PC** field contains the starting byte **PC** for the procedure (relative to the code base **CB**). The first byte of code contains the frame size index (fsi) for the local frame required. This index is used to allocate a local frame whose address is loaded into the **LF** register (see §9.2).

The following is a sketch of this process (ignoring traps and other types of control transfers). §9.3 contains a complete description.

```
proc: ProcDesc;
...
```

```
GF ← And[proc.taggedGF, 177776B];
CB ← ReadDblMds[@GlobalBase[GF].codebase];
PC ← proc.pc;
fsi: FSIndex = GetCodeByte[];
PC ← PC + 1;
LF ← Alloc[fsi];
```

The next section describes frame allocation in more detail.

9.2 Frame Allocation

The procedure call-return mechanism and the Allocate Frame and Free Frame instructions allocate and free frames from the *Frame Heap* in the Main Data Space. The heap is accessed using a structure called the *Allocation Vector*, which begins at a fixed location in each Main Data Space. Allocate and Free Frame make use of two more primitive operations, Alloc and Free. Allo takes a frame size index and returns the address of a frame of the requested size (or larger if indirection occurs). If Alloc cannot satisfy the request, it causes a fault (§10.4.3). Free takes a frame pointer and returns the frame to the appropriate list in the Allocation Vector. The structure of the Allocation Vector and the Frame Heap is illustrated in figure 9.1.

9.2.1 Frame Allocation Vector

To implement heap allocation, each Main Data Space contains a preallocated pool of frames of the most frequently-used sizes. The pool is organized as a vector of pointers to lists of frames of the same size, and frame sizes are encoded by their index into this vector. Since frames begin on four-word boundaries, the low-order two bits of the pointers comprising the lists are not needed to address the frames. Instead, these bits are used as a tag, encoded as shown below:

```
AV: POINTER TO AllocationVector = LOOPHOLE[mAV];
AllocationVector: TYPE = ARRAY FSIndex OF AVItem;

FSIndex: TYPE = [0..256);

AVItem: TYPE = MACHINE DEPENDENT RECORD [
   data (0: 0..13): [0..37777B],
   tag (0: 14..15): MACHINE DEPENDENT {frame, empty, indirect, unused}];
```

The frames of each size are arranged in a linked list, with the AVItem at the head of the list. If the tag field is frame (zero), the AVItem points to the first frame on the list, which will be the next one of this size allocated. The following routine is used to construct the frame pointer in this case:

```
AVFrame: PROCEDURE [avi: AVItem]
   RETURNS [LocalFrameHandle] =
   BEGIN
   IF avi.tag # frame THEN ERROR;
   RETURN[LOOPHOLE[avi]];
   END;
```

Note that the pointer addresses the first local variable of the frame. The frame actually begins at the frame address minus the size of the frame overhead (§3.2.2.2).

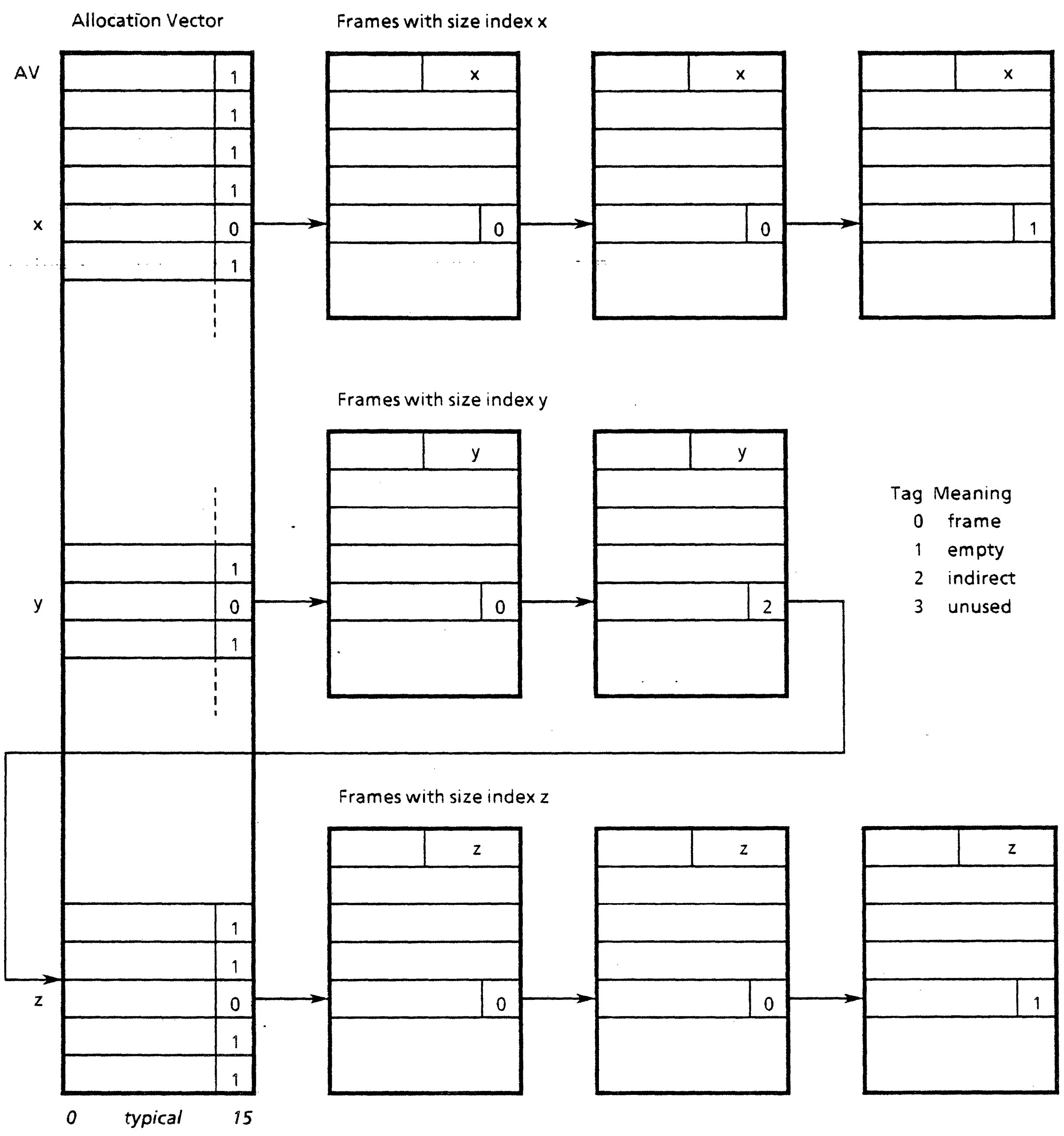


Figure 9.1 Frame Heap

When allocation occurs, AVFrame[AVItem] is returned to the requester. The contents of the word it points to (an AVItem, including tag bits) replaces the AV entry. Thus, frames are allocated from (and returned to) the head of the list, and the Allocation Vector entry usually points to the next frame to be allocated. The following routine decodes an AVItem into a pointer to the next item on the list:

```
AVLink: PROCEDURE [avi: AVItem]
  RETURNS [POINTER TO AVItem] =
  BEGIN
  IF avi.tag # frame THEN ERROR;
  RETURN[LOOPHOLE[avi]];
  END;
```

The last frame in a list contains either an end-of-list tag (empty) or an indirect tag (indirect). When the last frame is allocated, its AVItem is stored in the Allocation Vector as described above. At the next request for a frame of that size, if the tag of the AVItem is indirect, its data field is used as a frame size index to access another (larger) frame list. This list may in turn contain an indirect tag. An indirect AVItem normally is placed in the last frame on a list, because a larger frame size may be needed if that list becomes exhausted. If the tag is empty, an exception occurs. Since several processes may share the same Main Data Space (and thus share the same Allocation Vector and Frame Heap), a fault rather than a trap is generated (§10.4.3).

The frame heap should contain enough frames to satisfy the majority of allocation requests. In response to an exception, the programmer can supply more frames of the appropriate size and retry the operation.

9.2.2 Frame Allocation Primitives

The Alloc and Free primitives are defined below. Note that the FrameFault routine, defined in §10.4.3, does not return control to Alloc; instead, it raises the Abort signal (§4.1).

```
Alloc: PROCEDURE [fsi: FSIndex] RETURNS [LocalFrameHandle] =
  BEGIN
  item: AVItem;
  slot: FSIndex ← fsi;
  DO
    item ← FetchMds[@AV[slot]]↑;
    IF item.tag # indirect THEN EXIT;
    IF item.data > LAST[FSIndex] THEN ERROR;
    slot ← item.data;
    ENDLOOP;
  IF item.tag = empty THEN FrameFault[fsi];
  StoreMds[@AV[slot]]↑ ← FetchMds[AVLink[item]]↑;
  RETURN[AVFrame[item]];
  END;

Free: PROCEDURE [frame: LocalFrameHandle] =
  BEGIN
  word: LocalWord = FetchMds[@LocalBase[frame].word]↑;
  item: AVItem = FetchMds[@AV[word.fsi]]↑;
  StoreMds[frame]↑ ← item;
  StoreMds[@AV[word.fsi]]↑ ← frame;
  END;
```

Note that no assignments are performed until all possibility of a trap or fault has passed. Routines that call Alloc and Free are mindful of the fact that they have side-effects on the Frame Heap.

Programming Note: The Allocation Vector can be shorter than the maximum size specified above. In that case, it is the programmer's responsibility to ensure that no fsi greater than the size of the AV is used, since the processor does no dynamic bounds checking on fsi's.

Design Note: The actual frame sizes associated with each frame size index and the distribution of the number of frames of each size allocated is established by the programmer. Programmers can also designate certain fsi's for special purposes by ensuring that those indices are not generated in the fsi field of code-segment entry vectors. In addition, classes of frames (for instance, resident and non-resident frames) can be designated by using separate ranges of index values for each class. The architecture need not be aware of any of these properties of fsi's.

9.2.3 Frame Allocation Instructions

The Alloc and Free primitives are used by the control-transfer instructions. Frames can also be allocated from the heap by programmers using the following instructions:

AF **Allocate Frame**

```
AF: PROCEDURE =
  BEGIN
  fsi: FSIndex = Pop[];
  Push[Alloc[fsi]];
  END;
```

FF **Free Frame**

```
FF: PROCEDURE =
  BEGIN
  frame: LocalFrameHandle = Pop[];
  Free[frame];
  END;
```

Programming Note: Programmers can utilize all of the storage obtained by an Allocate Frame instruction, including frame overhead words. However, the frame size index in the overhead region must be preserved so that it is available for use by the Free primitive.

9.3 Control Transfer Primitive

The *control transfer* instructions, the trap mechanism, and the process-switching facility all make use of a primitive operation called XFER. The various forms of XFER are distinguished by the ways they generate the source and destination arguments, whether the current local frame is to be freed, their processing of the source and destination links, and some subtle differences in handling traps (discussed in §9.5).

The idea behind XFER is that a single primitive may be used to construct a variety of control disciplines, including procedure calls and returns, nested procedure calls, coroutine transfers, traps, and process switches.

Note: In the XFER code, nPC and nLF designate the new values of PC and LF, required because these registers cannot be modified due to the possibility of a trap or fault. Such temporaries for GF and CB are unnecessary, since the trap and fault routines do not reference them (§9.5.2, §10.4.3).

```
XferType: TYPE = MACHINE DEPENDENT {return(0), call(1), localCall(2), port(3), xfer(4),
trap(5), processSwitch(6), unused(7)};

XFER: PROCEDURE [
  dst: ControlLink, src: ShortControlLink, type: XferType, free: BOOLEAN ← FALSE] =
  BEGIN
  nPC: CARDINAL;
  nLF: LocalFrameHandle;
  push: BOOLEAN ← FALSE;
  nDst: ControlLink ← dst;
  IF type = trap AND free THEN ERROR;
  WHILE ControlLinkType[nDst] = indirect DO
    link: IndirectLink ← MakeIndirectLink[nDst];
    IF type = trap THEN ERROR;
    nDst ← ReadDblMds[link];
    push ← TRUE;
    ENDLOOP;

  . . .
```

In the case of an indirect control link, XFER follows the pointer until a frame link or procedure descriptor is located. The initial processing of a procedure descriptor was explained in §9.1.3; XFER completes the control transfer by initializing the local frame returned by Alloc with a pointer to the procedure's global frame and the procedure's return link (the source of the XFER).

If either the global frame pointer or pc are zero, the procedure descriptor is unbound and an UnboundTrap occurs. A CodeTrap is generated if the code base obtained from the global frame is odd. Recall also that the Alloc routine may generate a FrameFault (§9.2).

```
SELECT ControlLinkType[nDst] FROM
  procedure =>
    BEGIN
    word: BytePair;
    proc: ProcDesc = MakeProcDesc[nDst];
    GF ← And[proc.taggedGF, 177776B];
    IF GF = LOOPHOLE[0] THEN UnboundTrap[dst];
    CB ← ReadDblMds[@GlobalBase[GF].codebase];
    IF Odd[LowHalf[CB]] THEN CodeTrap[GF];
    nPC ← proc.pc;
    IF nPC = 0 THEN UnboundTrap[dst];
    word ← ReadCode[nPC/2];
    nLF ← Alloc[IF nPC MOD 2 = 0 THEN word.left ELSE word.right];
```

```
nPC ← nPC + 1;
StoreMds[@LocalBase[nLF].globallink] ↑ ← GF;
StoreMds[@LocalBase[nLF].returnlink] ↑ ← src;
END;
```

. . .

Design Note: It is important to notice that, after the Alloc completes, references to the new local frame nLF can not cause page faults, since Alloc references the first variable of the frame, and the frame overhead words are guaranteed to be in the same page.

Programming Note: It is illegal for a program to write-protect any page pointed to by the frame allocation vector. Such frames would be successfully removed from the frame heap by Alloc, but would be lost when the subsequent write-protect fault occurred.

Programming Note: CodeTraps can be used to implement a variety of features that depend on detecting all control transfers into a program. For example, this mechanism can be used by software to map code segments out of *virtual* memory. It can also be used (in conjunction with an auxiliary bit) to detect the first time control is passed to a program.

To effect a frame transfer, XFER loads the state from the destination, obtaining the PC and global frame from the local frame and the code base from the global frame. In addition to UnboundTrap and CodeTrap, a frame transfer can also result in a ControlTrap if the destination frame is zero. This trap is used to implement port linkages (§9.4.5), as well as to detect uninitialized control links.

```
frame =>
  BEGIN
  frame: FrameLink = MakeFrameLink[nDst];
  IF frame = LOOPHOLE[0] THEN ControlTrap[src];
  nLF ← frame;
  GF ← FetchMds[@LocalBase[nLF].globallink] ↑ ;
  IF GF = LOOPHOLE[0] THEN UnboundTrap[dst];
  CB ← ReadDblMds[@GlobalBase[GF].codebase];
  IF Odd[LowHalf[CB]] THEN CodeTrap[dst];
  nPC ← FetchMds[@LocalBase[nLF].pc] ↑ ;
  IF nPC = 0 THEN UnboundTrap[dst];
  IF type = trap THEN
    BEGIN
    StoreMds[@LocalBase[nLF].returnlink] ↑ ← src;
    DisableInterrupts[];
    END;
  END;
ENDCASE;
```

. . .

If the destination is a frame and the XFER is performing a trap, it saves the source of the transfer in the return link of the destination and disables interrupts (see §9.5.2).

If the original destination was an indirect control link, the *original* source and destination links are left above the top of the stack so that the context receiving control can access them (for example, the Link Byte instruction §9.4.3 or the Port In instruction §9.4.5).

Finally, XFER will optionally free the current local frame to the frame heap. This feature is used by the return instructions and XFER and Free to implement context destruction.

```
IF push THEN
  BEGIN
  Push[LowHalf[dst]]; Push[src];
  Discard[]; Discard[];
  END;
IF free THEN Free[LF];
LF ← nLF; PC ← nPC;
CheckForXferTraps[dst: dst, type: type];
END;
```

Design Note: Because the current frame is running, references to LF by Free cannot cause page faults (but, Free may fault on the Allocation Vector).

Design Note: Page faults due to Free cannot occur (see §9.5.2) because traps never free the source frame. If traps did specify free = TRUE, the trapped context would be discarded.

Programming Note: It is illegal for a program to unmap or write-protect its current local or global frame or to modify explicitly the dirty or referenced map flags of either frame's first page using the map instructions (§3.1.2).

9.4 Control Transfer Instructions

Several types of control transfers are implemented using the XFER primitive: local function calls, external function calls, nested function calls, returns, and port calls are described in this section. The use of XFER to implement traps is discussed in §9.5.

All control transfers except Return (§9.4.4) begin by storing the PC in the local frame. When stored, the PC points to the beginning of the next instruction. The stored value is therefore the byte offset of the instruction to be executed when the current local frame is resumed. (If the instruction causes a trap or fault, this value of the PC will be overwritten with savedPC by the trap or fault routine, so that the instruction will be restarted.)

9.4.1 Local Function Calls

Local function calls are used to transfer to a procedure located in the current code segment. These instructions are optimizations of the XFER mechanism, made possible by the fact that a particular code segment is compiled as a single unit. Using these instuctions, the compiler can build the information necessary to find the procedure that should be called into the code itself, rather than by using a procedure descriptor, which would be the normal case.

The Local Function Call instruction is an optimized version of the procedure descriptor form of XFER that does not modify the current global frame pointer or code base. In this case, there is no possibility of a CodeTrap.

LFC **Local Function Call**

```
LFC: PROCEDURE =
  BEGIN
  word: BytePair;
  nPC: CARDINAL;
  nLF: LocalFrameHandle;
  StoreMds[@LocalBase[LF].pc]↑ ← PC;
  nPC ← GetCodeWord[];
  IF nPC = 0 THEN UnboundTrap[LOOPHOLE[LONG[0]]];
  word ← ReadCode[nPC/2];
  nLF ← Alloc[IF nPC MOD 2 = 0 THEN word.left ELSE word.right];
  nPC ← nPC + 1;
  StoreMds[@LocalBase[nLF].globallink]↑ ← GF;
  StoreMds[@LocalBase[nLF].returnlink]↑ ← LF;
  LF ← nLF;  PC ← nPC;
  CheckForXferTraps[
    dst: LOOPHOLE[ProcDesc[taggedGF: Or[GF, 1], pc: PC-1]], type: localCall];
  END;
```

Design Note: As in XFER, after the Alloc completes, references to the new local frame nLF cannot cause page faults.

9.4.2 External Function Calls

The *external function calls* transfer control to a control link contained in the global frame or code segment. The following routine is used to obtain the link:

```
FetchLink: PROCEDURE [offset: BYTE] RETURNS [ControlLink] =
  BEGIN
  word: GlobalWord = FetchMds[@GlobalBase[GF].word]↑;
  RETURN[
    IF word.codelinks THEN ReadDbl[CB-LONG[(offset + 1)*2]]
    ELSE ReadDblMds[GlobalBase[GF]-(offset + 1)*2]];
  END;
```

Links are stored either just before the overhead words of the global frame or in an area preceding the code segment. A bit in the flag word of the current global frame indicates the link location (§3.2.2.2).

Design Note: If the links are stored in the code segment, they must be contained in the same 64K bank as the code base. This ensures that the calculation of the address of the link will not underflow in the low-order word or cause a borrow from the high-order word. Since frames are always completely contained in a Main Data Space, this calculation is also accurate if the links are stored in the global frame. Note, however, that the links do not necessarily lie in the same page as the beginning of the global frame or code segment.

The external function calls (and some other instructions) all make use of the routine below, which first saves the PC in the current frame and then XFERs to the destination, supplying LF as the source. The standard default for free is specified.

```
Call: PROCEDURE [dst: ControlLink] =
  BEGIN
  StoreMds[@LocalBase[LF].pc]↑ ←PC;
  XFER[dst: dst, src: LF, type: call];
  END;
```

Single-byte external call instructions are provided for the first thirteen external links. A two-byte version uses alpha as the link number, which allows up to 256 external references per module.

EFCn External Function Call n

```
EFCn: PROCEDURE [n: [0..12]] =
  BEGIN
  Call[FetchLink[n]];
  END;
```

EFCB External Function Call Byte

```
EFCB: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
  Call[FetchLink[alpha]];
  END;
```

The Stack Function Call instruction XFERS to a control link obtained from the top of the stack.

SFC Stack Function Call

```
SFC: PROCEDURE =
  BEGIN
  link: ControlLink = PopLong[];
  Call[link];
  END;
```

The following instruction XFERS to control links in the System Data table described in §9.5. The SD contains control links for kernel procedures that implement runtime support routines. Control links for trap handlers (§9.5.1) are also contained in the SD.

KFCB Kernel Function Call Byte

```
KFCB: PROCEDURE =
  BEGIN
  alpha: SDIndex = GetCodeByte[];
  Call[ReadDblMds[@SD[alpha]]];
  END;
```

9.4.3 Nested Function Calls

The Link Byte instruction is executed on entry to nested procedures. It establishes the static link to the enclosing context (the local frame of the lexically enclosing procedure).

Link Byte recovers the original destination link of the last XFER (normally an indirect control link) from above the stack, subtracts alpha, and stores the result in local zero.

LKB Link Byte

```
LKB: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
  link: ShortControlLink;
  Recover[]; link ← Pop[];
  StoreMds[LF]↑ ← link - alpha;
  END;
```

Programming Note: Because only control transfers through indirect control links leave source and destination links above the top of the stack, the LKB instruction must be executed after control transfers are made using indirect control links.

Programming Note: Nested procedure variables can be represented not by a procedure descriptor, but by a pointer to a procedure descriptor (an indirect link). In this case, the descriptor is allocated in the local frame of the enclosing procedure at an offset known to the programmer. This offset is used in the LKB instruction at the beginning of the nested procedure, which then uses local zero as the static link to access the enclosing procedure's variables. In order for the pointer to the descriptor to be recognized as an indirect control link, the descriptor must be allocated at an address equal to two modulo four.

9.4.4 Returns

The following instructions are used to return from a procedure, freeing its frame. Note that the PC is not saved in the current local frame, since the frame is about to be deallocated.

RET Return

```
RET: PROCEDURE =
  BEGIN
  dst: ControlLink = LONG[FetchMds[@LocalBase[LF].returnlink]↑];
  XFER[dst: dst, src: 0, type: return, free: TRUE];
  END;
```

Programming Note: Although there are separate local and external function call instructions, there is a single return, so that a context need not be concerned with how it was called.

9.4.5 Coroutine Transfers

The coroutine instructions are used to transfer control through *ports*, which are two-word structures located in the Main Data Space. Ports have the following runtime structure:

```
PortLink: TYPE = POINTER TO Port;

Port: TYPE = MACHINE DEPENDENT RECORD [
  inport (0): FrameLink,
    unused (1): UNSPECIFIED,
    outport (2): ControlLink];
```

Ports are allocated in memory so that PortLinks are indirect control links. That is, they are found at addresses of two modulo four. When control is directed into a port, the inport is used as the destination of the XFER. If the inport is non-zero, it contains a pointer to a frame that is said to be *pending* on the port (ready to receive control). Otherwise, a ControlTrap results. When control is directed out of a port, the outport is used as the destination of the XFER. The outport may contain any type of control link, including a frame, a procedure, or the address of another port. If it contains zero, the port is not linked to another context, and a ControlTrap results. Port calls are compatible with procedure calls, because control can leave a context using a port call and enter a context that uses a procedure call discipline (and vice versa). The various cases are shown in figures 9.2-4.

The Port Out instruction obtains the destination port link from the stack. It saves the current PC and sets the inport to the current context. It then XFERs to the outport, specifying the port itself (not the current context) as the source of the transfer. The Port Out instruction is always immediately followed statically by a Port In instruction, as shown in figure 9.2.

PO **Port Out**

```
PO: PROCEDURE =
  BEGIN
  reserved: unspecified = Pop[];
  port: PortLink = Pop[];
  StoreMds[@LocalBase[LF].pc]↑ ← PC;
  StoreMds[@port.inport]↑ ← LF;
  XFER[dst: ReadDblMds[@port.outport], src: port, type: port];
  END;
```

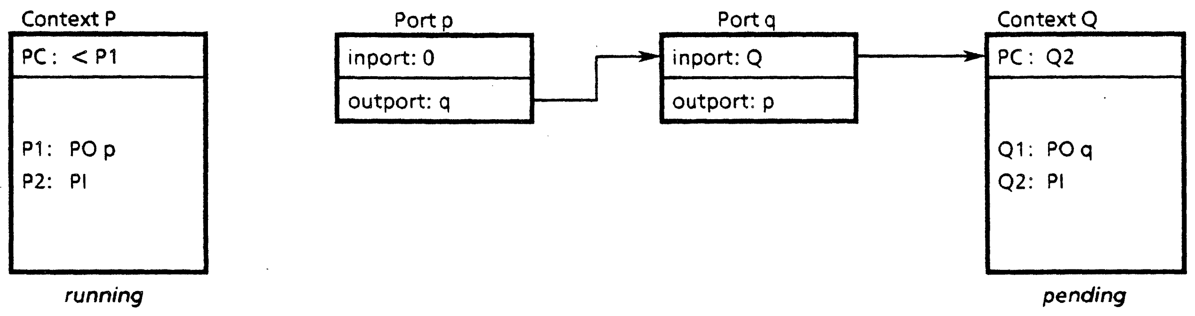
POR **Port Out Responding**

```
POR: PROCEDURE =
  BEGIN
  PO[];
  END;
```

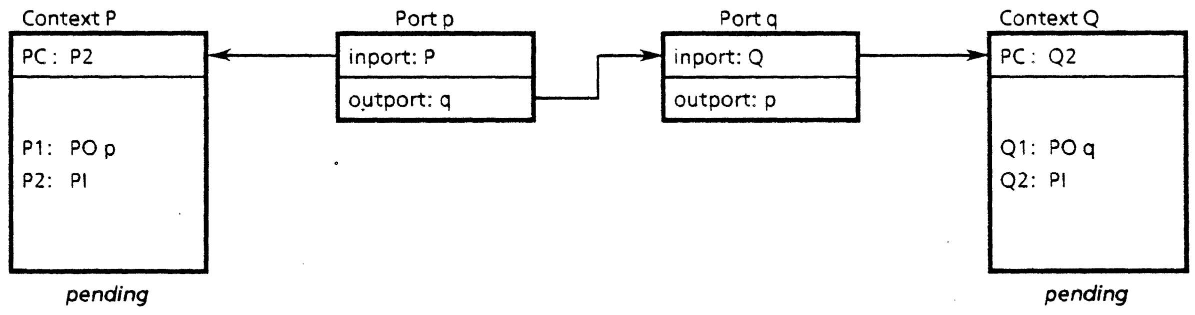
Programming Note: There are two Port Out instructions, with different opcodes but identical semantics. The trap handler uses them to determine the intended usage of the port if a ControlTrap occurs on the transfer. By convention, a *sending* port uses PO, and a *responding* port uses POR [2].

Programming Note: The high-order word of a PortLink is not used and may be left uninitialized by the programmer.

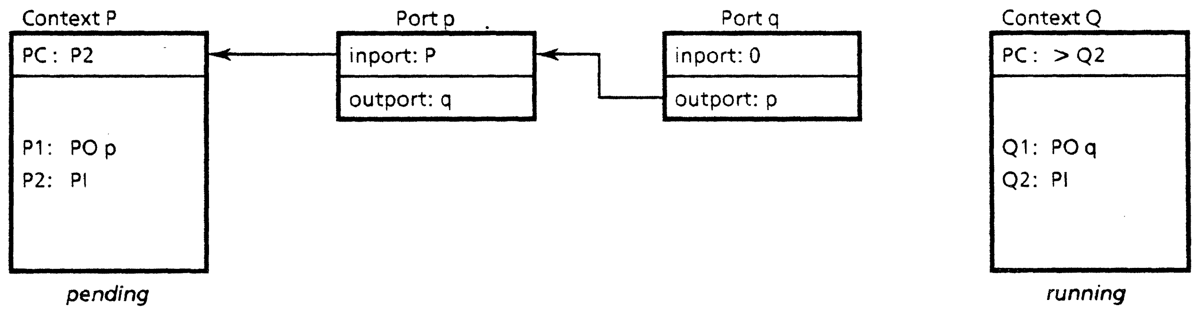
The Port In instruction saves the return link in the outport; that link had been left above the stack by the XFER invoked by the preceeding transfer instruction. Note that if the return link is zero, the Port In was preceded dynamically by a Return (or Return Zero), and the link is not saved. The instruction also clears the inport, so that any transfer directed at the port will cause a control trap.



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

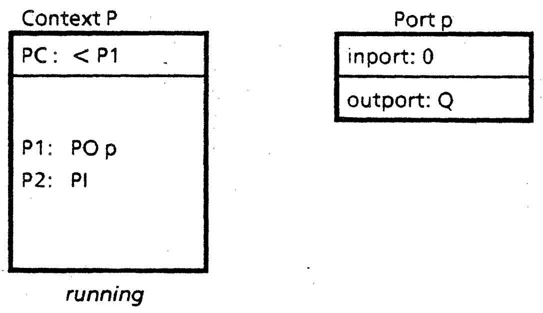


(2) P has executed the PO at P1, and control has passed to Q.
P is now pending on p, but Q has not yet executed the PI at Q2.

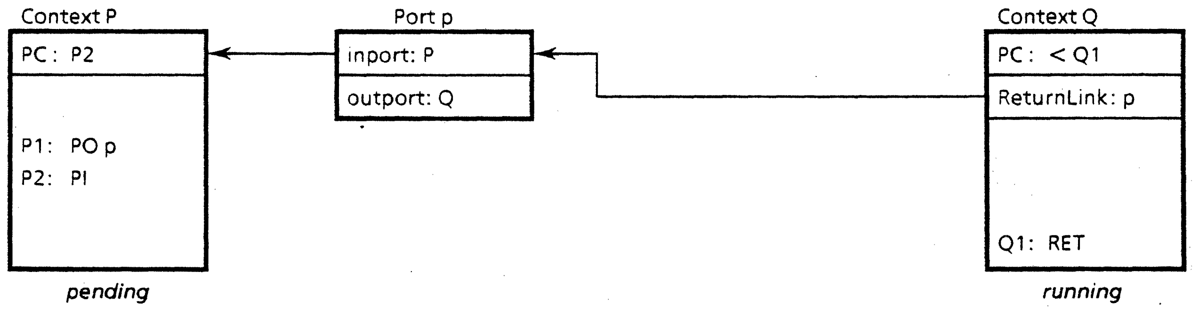


(3) Q has executed the PI at Q2, saving the link (an indirect link to p) in q.outport. Any attempt to transfer to Q through q will trap, since q.inport has been cleared. P remains pending on p.

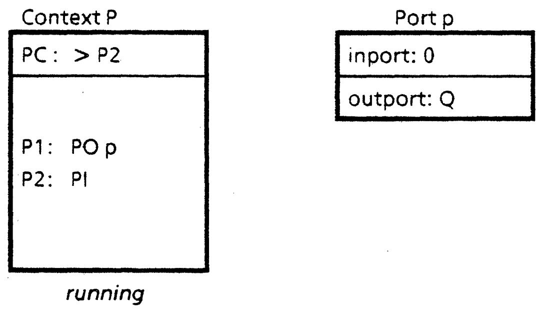
Figure 9.2 Port to Port Control Transfers



(1) Control is in P, before P1; p.outport
contains a procedure descriptor for context Q.

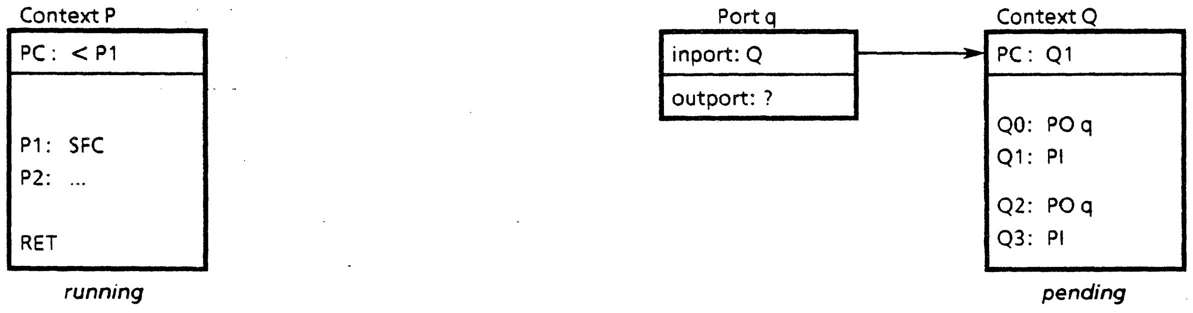


(2) P has executed the PO at P1, and Q has been created.
P is now pending on p, and Q's return link points to p.

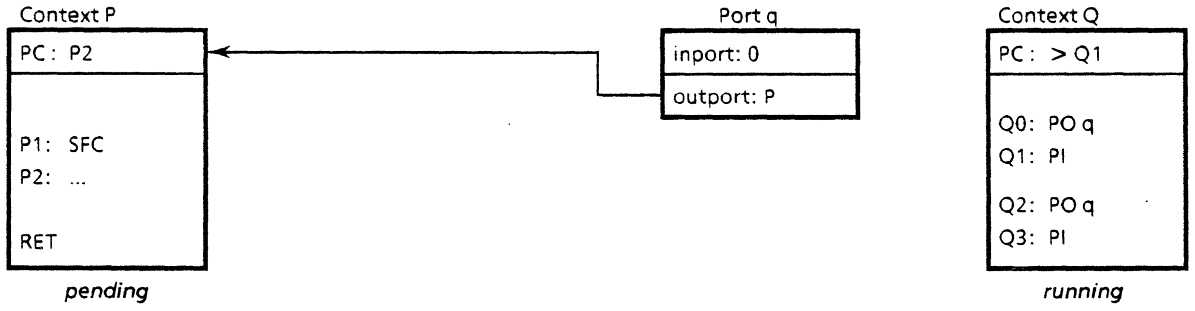


(3) Q has executed the RET, which called XFER[dst: p, src: 0].
P has executed the PI at P2, which has cleared p's inport.

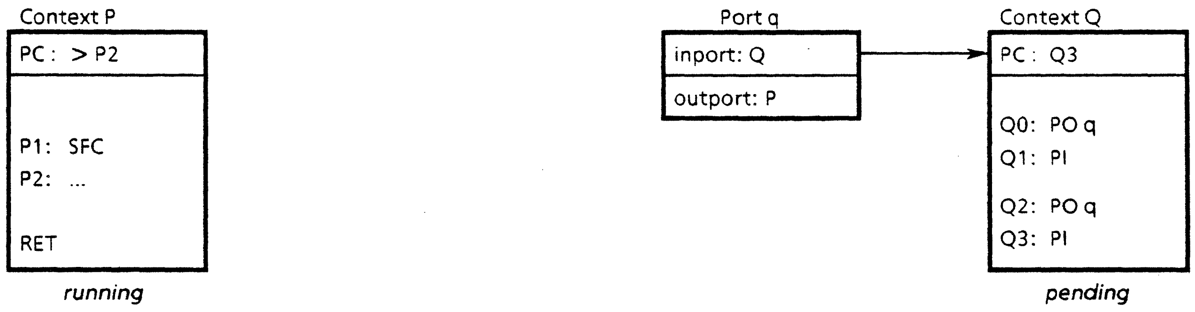
Figure 9.3 Port to Procedure Control Transfers



(1) Control is in P (a procedure), about to execute the SFC at P1.
The stack contains an indirect link to q. Q is pending on q.



(2) Control has passed to Q, and the PI at Q1 has been executed.
The inport has been cleared, and the outport contains P (a frame handle).



(3) Q has executed the PO at Q2, returning control to P
through the outport. Q is again pending on q.

Figure 9.4 Procedure to Port Control Transfers

PI **Port In**

```
PI: PROCEDURE =
  BEGIN
  port: PortLink;
  src: ShortControlLink;
  Recover[]; Recover[];
  src ← Pop[]; port ← Pop[];
  StoreMds[@port.inport]↑ ← 0;
  IF src # 0 THEN
    StoreMds[@port.outport]↑ ← src;
  END;
```

Programming Note: Because only control transfers made through indirect control links leave source and destination links above the top of the stack, the PI instruction must only be executed after control has been transferred using an indirect control link.

Programming Note: If a pre-emption occurs before the Port In completes execution, another process may enter the port, since a frame is still pending on it. This would eventually lead to a situation in which two processes were executing in the same local frame. For this reason, ports cannot be shared by multiple processes.

9.4.6 Link Instructions

The *load link* instruction loads a control link (or other data) from the global frame or the code segment onto the stack.

LLKB **Load Link Byte**

```
LLKB: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
  PushLong[FetchLink[alpha]];
  END;
```

The *read link* instructions perform a single- or double-word push using a pointer obtained from the link area.

RKIB **Read Link Indirect Byte**

```
RKIB: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
  ptr: LONG POINTER = FetchLink[alpha];
  Push[Fetch[ptr]↑];
  END;
```

RKDIB **Read Link Double Indirect Byte**

```
RKDIB: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
```

```
ptr: LONG POINTER = FetchLink[alpha];
Push[Fetch[ptr]↑];
Push[Fetch[ptr + 1]↑];
END;
```

9.5 Traps

Traps indicate the occurrence of exceptional conditions encountered in the course of instruction execution. In some cases, a trap indicates that a serious error has occurred, one which precludes continued execution of the context (for example, a StackError). In other cases, the trap will cause a software *trap handler* to take some corrective action and continue normal execution (e.g., CodeTrap). The trap handler is invoked using an XFER, which itself might generate a nested trap.

Both the XFER code in §9.3 and the trap routines defined in this section obey the restart rule given in §4.6.1 by using temporary variables and recursion. This stringency is particularly important in the case of breakpoints, code traps, and XFER traps when they are mixed with frame allocation faults and page faults. (Fault processing is discussed in §10.4.3.)

9.5.1 Trap Routines

The following paragraphs summarize the traps that can be generated by the processor. Trap handlers for traps other than EscOpcodeTrap are represented by control links located at preassigned indexes in the System Data table, which resides at a fixed address within each Main Data Space. The location of the System Data table and the assignments of indexes to trap conditions are given in Appendix A.

```
SD: POINTER TO SystemData = LOOPHOLE[mSD];
SystemData: TYPE = ARRAY SDIndex OF ControlLink;

SDIndex: TYPE = [0..256);
```

Trap handlers for unimplemented ESC or ESCL opcodes are represented by control links located in the ESC Trap table, which resides at a fixed address within each Main Data Space. The ESC Trap table is indexed by opcode value. Its location is given in Appendix A.

```
ETT: POINTER TO EscTrapTable = LOOPHOLE[mETT];
EscTrapTable: TYPE = ARRAY BYTE OF ControlLink;
```

A brief explanation of each trap is given below, together with a description of its parameters and a reference to the section(s) that call the trap routine. Traps are listed alphabetically.

```
BoundsTrap: PROCEDURE [] = {TrapZero[@SD[sBoundsTrap]]};
```

The Bounds Check instruction generates this trap in response to an out-of-range value (§5.2).

```
BreakTrap: PROCEDURE [] = {TrapZero[@SD[sBreakTrap]]};
```

This trap is invoked by the Break instruction (§9.5.4).

```
CodeTrap: PROCEDURE [gf: GlobalFrameHandle] = {
  TrapOne[@SD[sCodeTrap], gf]};
```

An XFER generates this trap if the code base of the destination context is odd. The parameter is the global frame of the new context (§9.3).

```
ControlTrap: PROCEDURE [src: ShortControlLink] = {
  TrapOne[@SD[sControlTrap], src]};
```

This trap is generated by an XFER if the destination of a frame transfer is zero. The parameter is the source of the original transfer (§9.3).

```
DivCheckTrap: PROCEDURE [] = {TrapZero[@SD[sDivCheckTrap]]};
```

The Long Unsigned Divide instruction generates this trap if the quotient would overflow a single word (§5.5).

```
DivZeroTrap: PROCEDURE [] = {TrapZero[@SD[sDivZeroTrap]]};
```

An attempt to divide by zero generates this trap (§5.5).

```
EscOpcodeTrap: PROCEDURE [opcode: BYTE] = {TrapOne[@ETT[opcode],
opcode]};
```

This trap is generated when instruction execution detects an unimplemented ESC or ESCL opcode (§4.5).

```
InterruptError: PROCEDURE [] = {TrapZero[@SD[sInterruptError]]};
```

This trap is generated by the Disable and Enable Interrupts instructions if the wakeup disable counter **WDC** would underflow or overflow as a result of the operation (see §10.4.4.3).

```
OpcodeTrap: PROCEDURE [opcode: BYTE] = {TrapOne[@SD[sOpcodeTrap], opcode]};
```

This trap is generated when instruction execution detects an unimplemented opcode (§4.5).

```
PointerTrap: PROCEDURE [] = {TrapZero[@SD[sPointerTrap]]};
```

This trap is generated when the Nil Check Long instruction detects a zero pointer (§5.2).

```
ProcessTrap: PROCEDURE [] = {TrapZero[@SD[sProcessTrap]]};
```

The Monitor Reentry instruction generates this trap when the process's abort bit has been set (§10.2.4).

```
RescheduleError: PROCEDURE [] = {TrapZero[@SD[sRescheduleError]]};
```

This trap is generated by the scheduler when the scheduler was entered at an inappropriate time. Inappropriate entry may be caused by interrupts having been disabled while a process opcode causing scheduler entry was being executed. If interrupts were disabled and a page fault, write-protect fault, or frame fault caused entry to the scheduler, the RescheduleError may also be called (§10.4.1).

```
StackError: PROCEDURE [] = {TrapZero[@SD[sStackError]]};
```

Any instruction that manipulates the evaluation stack may cause this trap if the stack routines detect that the stack pointer **SP** would become illegal as a result of the operation (§3.3.2).

```
UnboundTrap: PROCEDURE [dst: ControlLink] = {
  TrapTwo[@SD[sUnboundTrap], dst]};
```

An XFER generates this trap whenever it encounters a zero destination control link or a zero PC. The parameter is the destination of the original transfer (§9.3).

```
HardwareError: PROCEDURE [] = {TrapZero[@SD[sHardwareError]]};
```

Miscellaneous machine-dependent hardware errors generate this trap.

Design Note: *There are three exceptions to the restart rule: in the event of a stack error, a reschedule error, or an interrupt error, the state of the processor is undefined.* These three traps represent fatal software errors from which it is generally impossible to resume execution. In a stack error (§3.3.2), the trapped context is not resumable. In a reschedule error (§10.4.1) or an interrupt error (§10.4.4.3), no process may continue execution.

Programming Note: Resuming a context that has experienced a stack error produces undefined results, as does continuation of process-scheduling after a reschedule error or an interrupt error. The processor reports these three conditions for debugging purposes only; they never occur in correct programs.

9.5.2 Trap Processing

When a trap occurs, the action taken is similar to a normal control transfer to the trap handler, with the following differences:

The trap mechanism stores the trap parameters starting at local zero in the handler's frame. There can be up to four words of parameters (there are currently at most three). Because some instructions, including many XFERs, leave information above the top of the stack, the entire stack must be preserved unmodified; therefore, trap parameters cannot be passed on the stack.

For similar reasons, the XFER performed by the trap routine does not save its source and destination links above the top of the stack (see the description of XFER in §9.3).

Programming Note: This implies that trap handlers can not in general be nested procedures or ports, since these programs begin with instructions (Link Byte and Port In) that access the control links left above the top of the stack by the previous XFER.

If the transfer performed by the trap routine specifies a frame as its destination, XFER stores the source of the transfer in the return link of the trap handler and disables interrupts.

Design Note: Because several processes can share a Main Data Space (and therefore share the trap handlers in its System Data table), fixed-frame trap handlers are not re-entrant and must run with interrupts disabled. They also cannot cause traps or faults.

Programming Note: The use of fixed-frame trap handlers should be restricted to the most primitive performance monitoring and debugging functions.

Design Note: Because traps are implemented as control transfers, the MDS register and the PSB register are not modified by trap processing. This implies that the trap handler runs in the same MDS (and in the same process) as the trapped context.

The precise actions that must be taken by the processor when a trap occurs are shown by TrapZero (no parameters), TrapOne (one parameter), TrapTwo (one long parameter), and the common Trap routine.

```
TrapZero: PROCEDURE [ptr: POINTER TO ControlLink] =
  BEGIN
  Trap[ptr];
  ERROR Abort;
  END;

TrapOne: PROCEDURE [ptr: POINTER TO ControlLink, parameter: UNSPECIFIED] =
  BEGIN
  Trap[ptr];
  StoreMds[LF]↑ ← parameter;
  ERROR Abort;
  END;

TrapTwo: PROCEDURE [ptr: POINTER TO ControlLink, parameter: LONG UNSPECIFIED] =
  BEGIN
  Trap[ptr];
  StoreMds[LF]↑ ← LowHalf[parameter];
  StoreMds[LF + 1]↑ ← HighHalf[parameter];
  ERROR Abort;
  END;

Trap: PROCEDURE [ptr: POINTER TO ControlLink] =
  BEGIN
  handler: ControlLink = ReadDblMds[ptr];
  PC ← savedPC;  SP ← savedSP;
  IF ValidContext[] THEN StoreMds[@LocalBase[LF].pc]↑ ← PC;
  XFER[dst: handler, src: LF, type: trap];
  END;
```

Design Note: The storing of the trap parameter(s) cannot cause a page fault, because the XFER has already guaranteed the presence of the trap handler's first four locals (and its overhead words).

The trap routine must check that the context is valid before saving the PC. This covers the occurrence of a trap during a process switch before a valid frame has been obtained (see §10.4.1).

Design Note: Because the frame is currently running, saving the PC can not cause a page fault.

Since the processor will always re-establish the initial state of the current instruction, all traps *appear* to occur between instructions. If an instruction causes more than one trap, the traps will occur sequentially, and the processor will restart the instruction when the handler for each trap returns. Because instructions are restarted rather than continued from the point of an exception, there is no need for a trap handler to consider the effects of multiple traps on a single instruction, nor does the handler need to concern itself with the continuation of partially completed instructions.

9.5.3 Trap Handlers

The complete state of a context includes not only the current local frame LF, but the evaluation stack (§3.3.2) and the break byte (§9.5.4) as well. Instructions are provided to dump and load this state using a *state vector*, defined as follows:

```
StateHandle: TYPE = LONG POINTER TO StateVector;

StateWord: TYPE = MACHINE DEPENDENT RECORD [
  break (0: 0..7), stkptr (0: 8..15): BYTE];

StateVector: TYPE = MACHINE DEPENDENT RECORD [
  stack (0): ARRAY [0..StackDepth) OF UNSPECIFIED,
  word (14): StateWord,
  frame (15): LocalFrameHandle,
  data (16): BLOCK];
```

Design Note: The size of a state vector is processor dependent (§10.4.2.1). Its minimum size, cSV, is given in Appendix A.

The *state* instructions make use of the routines below to save and restore the break byte, the stack pointer (and savedSP), and the evaluation stack. Since there are never more than two valid entries above the top of the stack, a maximum of SP + 2 entries need be saved.

```
SaveStack: PROCEDURE [state: StateHandle] =
  BEGIN
  FOR sp: StackPointer IN [0..MIN[SP + 2, StackDepth]) DO
    Store[@state.stack[sp]] ↑ ← stack[sp];
    ENDLOOP;
  Store[@state.word] ↑ ← StateWord[break, SP];
  SP ← savedSP ← 0;
  break ← 0;
  END;

LoadStack: PROCEDURE [state: StateHandle] =
  BEGIN
  word: StateWord = Fetch[@state.word] ↑ ;
  FOR sp: StackPointer IN [0..MIN[word.stkptr + 2, StackDepth]) DO
    stack[sp] ← Fetch[@state.stack[sp]] ↑ ;
    ENDLOOP;
  SP ← savedSP ← word.stkptr;
  break ← word.break;
  END;
```

The first instruction executed by a trap handler will normally be a Dump Stack, which will empty the stack by saving its contents in the handler's local frame, at an offset given by alpha.

DSK Dump Stack

```
DSK: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
  state: POINTER TO StateVector = LOOPHOLE[LF + alpha];
  SaveStack[LengthenPointer[state]];
  END;
```

Programming Note: Since trap parameters are stored starting at local zero, the programmer must arrange that the state vector referenced by the Dump Stack instruction is not allocated in the first four frame locations.

When the handler is ready to continue execution of the trapped context, it must reload the stack. The Load Stack instruction is available for this purpose.

LSK Load Stack

```
LSK: PROCEDURE =
  BEGIN
  alpha: BYTE = GetCodeByte[];
  state: POINTER TO StateVector = LOOPHOLE[LF + alpha];
  LoadStack[LengthenPointer[state]];
  END;
```

Since the entire stack is reloaded, it is not necessary to preserve the stack's old contents in case of faults while reloading. If the Load Stack does fault, the stack may be only partially loaded, but the entire operation will be retried when execution resumes.

After the stack is reloaded, the handler must resume the trapped context. There are two instructions available for this purpose: one that frees the handler's frame, and one that enables interrupts. The programmer provides the source and destination of a control transfer.in the trap handler's frame. Typically, the source is zero, and the destination is the original trapped context. This will retry the instruction that caused the trap.

```
TransferDescriptor: TYPE = MACHINE DEPENDENT RECORD [
  src (0): ShortControlLink,
  reserved (1): UNSPECIFIED ← 0,
  dst (2): ControlLink] ;
```

XF XFER and Free

```
XF: PROCEDURE =
  BEGIN
  ptr: POINTER TO TransferDescriptor = LOOPHOLE[LF + GetCodeByte[]];
  XFER[
    dst: ReadDblMds[@ptr.dst], src: FetchMds[@ptr.src]↑, type: xfer, free: TRUE];
  END;
```

XE XFER and Enable

```
XE: PROCEDURE =
  BEGIN ENABLE Abort = > ERROR;
  ptr: POINTER TO TransferDescriptor = LOOPHOLE[LF + GetCodeByte[]];
  StoreMDS[@LocalBase[LF].pc]↑ ← PC;
  XFER[dst: ReadDblMds[@ptr.dst], src: FetchMds[@ptr.src]↑, type: xfer];
  EnableInterrupts[];
  END;
```

Programming Note: It is the programmer's responsibility to ensure that XFER and Enable does not cause a trap or fault (see §9.5.2). This instruction is intended for use by fixed frame trap handlers, which must run with interrupts disabled.

Programming Note: Some traps indicate error conditions that will normally be handled by software executed *in place of* the trapped instruction; re-execution may therefore be inappropriate. For example, the unimplemented instruction trap handler may choose to emulate the effects of the offending instruction in software. In this case, it is always possible for the trap handler to complete the instruction by advancing the program counter of the trapped context (and perhaps also adjusting the contents of the stack).

9.5.4 Breakpoints

The single byte Break instruction causes a distinguished trap when encountered in the instruction stream. It is used for program debugging. To install a breakpoint, the programmer replaces the opcode of the broken instruction with a Break instruction, remembering the original opcode value. When the processor attempts to execute the broken instruction, a trap to a software-supplied breakpoint handler results. (The processor's break byte is normally zero.) BreakTrap is defined in §9.5.2.

BRK Break

```
BRK: PROCEDURE =
  BEGIN
  IF break = 0 THEN BreakTrap[]
  ELSE
    BEGIN
    Dispatch[break];
    break ← 0;
    END;
  END;
```

As with all traps, the first instruction of the breakpoint trap handler should be a Dump Stack, which will save the state of the broken context. To proceed from a break point, the programmer can replace the Break instruction with the original opcode; however, it is usually desirable to leave the breakpoint in place, by first inserting the original opcode into the break field of the state vector of the broken context, then performing a Load Stack instruction. The Load Stack sets the processor's break byte from the state vector. The Break instruction notices a non-zero break byte and dispatches on it (§4.5), rather than performing a breakpoint trap; it then clears the break byte when the broken instruction completes execution.

Design Note: It is important to notice that the break byte is not cleared until the broken instruction has completed execution *successfully*. In particular, if the dispatch on the original opcode results in a trap or fault, the break byte must be saved with the state of the trapped or faulted process, and the break byte associated with the new context (established by the trap or fault routine) must remain undisturbed. In the code above, the Abort signal raised by the trap or fault routine prevents execution of the statement break ←0.

Programming Note: It is possible to place a breakpoint anywhere *except* in the current breakpoint trap handler itself. Also note that, if the break byte is ever set to zBRK, infinite recursion occurs.

9.5.5 Xfer Traps

A method of trapping to software on each transfer of control, conditioned by the trapxfers flag in the global frame of the destination, is available. It is intended for performance measurement and debugging.

```
CheckForXferTraps: PROCEDURE [dst: ControlLink, type: XferType] =
  BEGIN
  IF Odd[ XTS] THEN
    BEGIN
    word: GlobalWord = FetchMds[@GlobalBase[GF].word] ↑ ;
    IF word.trapxfers THEN
      BEGIN
      XTS ← Shift[XTS, -1];
      Trap[@SD[sXferTrap] ! Abort = > ERROR];
      StoreMds[LF] ↑ ← LowHalf[dst];
      StoreMds[LF + 1] ↑ ← HighHalf[dst];
      StoreMds[LF + 2] ↑ ← type;
      ERROR Abort;
      END
    END
  ELSE XTS ← Shift[XTS, -1];
  END;
```

10 Processes

This chapter describes the process mechanism implemented by the Mesa architecture. It includes a description of the data types and structures used to support processes, monitor locks, condition variables, and fault queues. It also defines the process instructions, the process queue-management routines, and the scheduling algorithms. The last section on scheduling includes a description of the state-vector allocation performed by the scheduler, as well as a discussion of exceptional conditions that invoke the scheduler (faults, interrupts, and timeouts) and the processing that they receive.

The process facilities are used for controlling the execution of multiple processes and guaranteeing mutual exclusion. The intended application of the process mechanism is the management of access to shared resources (such as the processor). Asynchronous communication with I/O devices is also supported by the process mechanism.

The process implementation is based on queues of small objects called *Process State Blocks* (PSBs), each representing a single process. When a process is not running, its PSB records the state associated with the process, including the process' Main Data Space, its evaluation stack (possibly), and the frame (context) it was last executing. Only in the case of a pre-emption is the stack saved in a state vector as part of the process state. In other cases, the stack is known to be empty. PSBs also record the process' priority and a few flag bits associated with the process (see §10.1.2).

When a process is running, its state is contained in the processor's control registers described in §3.3.1. These registers include all of those that constitute a context (including the evaluation stack), plus the **PSB** and **MDS** registers. The **PSB** register points to the process' PSB, and the **MDS** register addresses its Main Data Space. These registers are modified when a process switch takes place.

The contents of the **MDS** register is normally modified only by a process switch (it can also be read and written using the register instructions defined in §3.3.4). Several processes can share a single Main Data Space, or an MDS can be restricted to contain a single process. As long as the **MDS** register contains a legal value, the processor can execute programs in an environment containing no processes (that is, one in which the content of the **PSB** register and the current PSB are undefined). The processor begins execution in this state (§4.7).

Each Process State Block is a member of exactly one process queue. There is one queue for each monitor lock, condition variable, and fault handler in the system. A process that is not suspended on a monitor lock, waiting on a condition variable, or faulted is either running or is on the ready queue, waiting for the processor. The semantics of each monitor and condition queue are assigned by the programmer. Except for the ready queue and the fault queues, there are no fixed assignments of queues to resources.

The primary effect of the process instructions and the scheduler is to move PSBs back and forth between the ready queue and a monitor or condition queue. A process moves from the processor to a monitor queue when it attempts to enter a locked monitor. It moves from the monitor queue to the ready queue when the monitor is unlocked (by some other process). Similarly, a process moves from the processor to a condition queue when it waits on a condition variable, and moves to the ready queue when the condition variable is notified, or when the process has timed out.

Each time a process is requeued, the scheduler is invoked. The scheduler saves the state of the current process in its PSB and state vector, chooses the highest-priority runnable process, removes that process from the ready queue, and loads its state into the processor registers. To simplify the scheduler's task, all of the process queues are kept sorted by priority.

Certain exceptional conditions result in process switches rather than traps. These also manipulate the process queues and call the scheduler. Faults (in particular, page faults, write protect faults, frame allocation faults) cause the current process to be placed on a fault queue. The fault's associated condition variable is then notified. An interrupt causes one of the sixteen preassigned condition variables to be notified. Finally, a timeout causes a waiting process to be made ready, even though the condition variable on which it is waiting was not notified by another process.

10.1 Data Structures

A global data structure is used to store the Process State Blocks. This section describes that global data structure, called the Process Data Area (PDA), as well as the details of Process State Blocks, monitor locks, condition variables, and process queue. Details of queue management, however, are postponed until §10.3.

10.1.1 Process Data Area

The Process Data Area is located at a fixed address in virtual memory and is 64K-aligned (the value of mPDA is given in Appendix A):

```
PDA: LONG POINTER TO ProcessDataArea = LOOPHOLE[mPDA];
```

The Process Data Area contains all of the process structures except for monitor locks and condition variables, which are allocated by the programmer. The PDA has the following structure:

```
ProcessDataArea: TYPE = MACHINE DEPENDENT RECORD [
  SELECT OVERLAID * FROM
    header = > [
      ready: Queue,
    count: CARDINAL,
    unused: UNSPECIFIED,
    available: ARRAY [0..5) OF UNSPECIFIED,
    state: StateAllocationTable,
    interrupt: InterruptVector,
    fault: FaultVector],
  blocks = > [
    block: ARRAY [0..0) OF ProcessStateBlock],
  ENDCASE];
```

The PDA contains a resident array of Process State Blocks indexed by a PsbIndex. The initial elements of the array are allocated to other global state information. The first PSB is at index StartPsb, after the PDA header. A zero index is used to denote the null process.

```
PsbNull: PsbIndex = 0;
PsbIndex: TYPE = [0..1024);

StartPsb: PsbIndex =
  (SIZE[ProcessDataArea] + SIZE[ProcessStateBlock]-1)/SIZE[ProcessStateBlock];
```

The index of the currently running process is maintained in a processor register, which is also accessible to the programmer.

```
PSB: PsbIndex;
```

The header of the Process Data Area includes the ready queue and a count of the number of PSBs (not including overhead) in the PDA. There is also a small block available to the programmer, a table of pointers to state vectors used to save the context and stack of pre-empted processes (§10.4.2), an array of condition variables assigned to interrupt levels (§10.4.4), and a structure used to represent fault queues and their associated condition variables (§10.4.3). The PSBs and state vectors follow the header.

Design Note: The count field is used only to determine the number of processes involved in the timeout scan (§10.4.5). Additional PSBs may be allocated by the programmer.

A PSB is active if it is running or is on a process queue; only active PSBs may be referenced by the process instructions.

Programming Note: The programmer typically allocates a fixed number of PSBs, stores this number in the header as the count, and makes the PSBs active (by placing them on the ready queue) as processes are created. Because the timeout scan examines all the PSBs indicated by the count field, the timeout value associated with each inactive process must be set to zero (§10.4.5).

All of the pointers contained in the Process Data Area (and in the State Allocation Table) are relative to the starting location of the PDA (just as short pointers are relative to the origin of an MDS). Like Main Data Spaces, the PDA is aligned on a 64K boundary.

```
LengthenPdaPtr: PROCEDURE [ptr: POINTER] RETURNS [LONG POINTER] =
  BEGIN
  offset: CARDINAL = LOOPHOLE[ptr];
```

```
    RETURN[PDA + LONG[offset]];
    END;

  OffsetPda: PROCEDURE [ptr: LONG POINTER] RETURNS [POINTER] =
    BEGIN
    IF HighHalf[ptr - PDA] # 0 THEN ERROR;
    RETURN[LowHalf[ptr - PDA]];
    END;

  FetchPda: PROCEDURE [ptr: POINTER] RETURNS [LONG POINTER] =
    BEGIN
    RETURN[Fetch[LengthenPdaPtr[ptr]]];
    END;

  StorePda: PROCEDURE [ptr: POINTER] RETURNS [LONG POINTER] =
    BEGIN
    RETURN[Store[LengthenPdaPtr[ptr]]];
    END;
```

Implementation Note: Because the PDA is 64K-word aligned and its relative pointers are restricted to a 64K range, a concatenation operation can replace the addition that appears above, and an extraction of the low-order word can replace the subtraction.

It is often convenient to reference PSBs using PDA-relative pointers rather than indexes. The following routines convert between the two representations:

```
  PsbHandle: TYPE = POINTER TO ProcessStateBlock;

  Handle: PROCEDURE [index: PsbIndex] RETURNS [PsbHandle] =
    BEGIN
    RETURN[LOOPHOLE[index*SIZE[ProcessStateBlock]]];
    END;

  Index: PROCEDURE [handle: PsbHandle] RETURNS [PsbIndex] =
    BEGIN
    RETURN[LOOPHOLE[handle]/SIZE[ProcessStateBlock]];
    END;
```

Design Note: All of the process data structures containing a PsbIndex are laid out so that the low-order bit of the index lies at bit twelve of a word. Because PSBs are eight-word aligned, this allows the conversion from index to handle to be performed by masking out the low-order three and the high-order three bits of the word containing the index. Thus, the 10-bit index is extracted from a 16-bit word and converted into a PDA-relative pointer.

Implementation Note: Because the Process Data Area and the tables pointed to by it are resident, the processor need not maintain the dirty and referenced map flags of the pages containing these structures (§3.1.1).

10.1.2 Process State Blocks

A PSB records the state of a process, and it therefore contains the following fields:

A *link word*, used for maintaining the queue structure. This word also contains the process' priority and some flag bits.

A *flag word* containing fields used primarily by the programmer. This word also includes a single flag bit and a cleanup link used for maintaining the queue structure.

A *process context representation*, which is either a pointer to the frame the process was last executing, or a pointer to a state vector containing the evaluation stack and the frame pointer.

The *timeout value* associated with each process (see §10.4.5).

The high-order sixteen bits of the (64K-aligned) address of the Main Data Space in which the process is running.

Two words of sticky flags for floating-point operations.

Process State Blocks are therefore eight words long, and are always eight-word aligned.

```
ProcessStateBlock: TYPE = MACHINE DEPENDENT RECORD [
  link (0): PsbLink,
  flags (1): PsbFlags,
  context (2): POINTER,
  timeout (3): Ticks,
  mds (4): CARDINAL,
  available (5): UNSPECIFIED,
  sticky (6): LONG UNSPECIFIED];
```

The link word, in addition to the process' priority and a queue link, contains three flag bits: failed indicates that the last instruction executed by the process was an unsuccessful Monitor Entry (§10.2.1) or Monitor Reentry (§10.2.4); permanent indicates that the PSB's context is a state vector that is permanently assigned to the process; preempted records whether the PSB was pre-empted. If it was pre-empted, the saved context includes the evaluation stack and the frame pointer, otherwise it contains only the frame pointer.

```
Priority: TYPE = [0..7];

PsbLink: TYPE = MACHINE DEPENDENT RECORD [
  priority (0: 0..2): Priority,
  next (0: 3..12): PsbIndex,
  failed (0: 13..13): BOOLEAN,
  permanent (0: 14..14): BOOLEAN,
  preempted (0: 15..15): BOOLEAN];
```

The flag word, in addition to a cleanup link and several fields available to the programmer, contains flags that indicate whether the process is waiting on a condition queue and also whether there is an abort pending for the process. If the latter flag is set (by the programmer), the Monitor Reentry instruction causes a trap (§10.2.4).

```
PsbFlags: TYPE = MACHINE DEPENDENT RECORD [
  available (0: 0..2): [0..7],
  cleanup (0: 3..12): PsbIndex,
  reserved (0: 13..13): BIT ← 0,
    waiting (0: 14..14): BOOLEAN,
    abort (0: 15..15): BOOLEAN];
```

Design Note: Process State Blocks must be resident in real memory, and may not be write-protected. No reference to a PSB may cause a fault.

10.1.3 Monitor Locks

Unlike Process State Blocks, monitor locks are allocated by the programmer. They serve as queue headers, and therefore contain a PsbIndex pointing into the body of the queue (the queue structure is defined in §10.1.5). A monitor queue contains all of the processes suspended on the monitor lock.

In addition to the queue pointer, a monitor lock includes a lock bit; when set, the lock bit indicates that some process is executing inside the monitor. If another process attempts to enter the monitor while the lock is set, that process is suspended by placing it on the monitor queue.

```
Monitor: TYPE = MACHINE DEPENDENT RECORD [
    reserved (0: 0..2): [0..7] ← 0,
    tail (0: 3..12): PsbIndex,
    available (0: 13..14): [0..3],
    locked (0: 15..15): BOOLEAN];
```

Design Note: Monitor locks need not be resident (although some may be).

10.1.4 Condition Variables

Condition variables are also allocated by the programmer. Like monitor locks, they serve as queue headers, and contain a PsbIndex pointing into the body of the queue. A condition queue contains all of the processes waiting on the condition.

In addition to the queue pointer, a condition variable contains two flag bits: abortable is set (by the programmer) if processes waiting on the condition can be aborted (§10.2.4); wakeup is set in response to a fault (§10.4.3) or an interrupt (§10.4.4).

```
Condition: TYPE = MACHINE DEPENDENT RECORD [
    reserved (0: 0..2): [0..7] ← 0,
    tail (0: 3..12): PsbIndex,
    available (0: 13..13): BIT,
    abortable (0: 14..14): BOOLEAN,
    wakeup (0: 15..15): BOOLEAN];
```

Design Note: Except for those contained in the PDA, condition variables need not be resident.

10.1.5 Process Queues

A queue is represented by a long pointer to a queue header, declared as type Queue. A queue header is either a monitor lock, a condition variable, a fault queue, or the ready field of the Process Data Area. The header contains a field called tail, which is the index of the *last* PSB on the queue (queue entries are always Process State Blocks).

```
QueueHandle: TYPE = LONG POINTER TO Queue;
Queue: TYPE = MACHINE DEPENDENT RECORD [
   reserved1 (0: 0..2): [0..7] ← 0,
   tail (0: 3..12): PsbIndex,
   reserved2 (0: 13..15): [0..7]];
```

Design Note: The formats of monitor locks and condition variables are carefully designed to match the structure of a Queue, so that they can function as queue headers. When functioning as queue headers, they define additional flags in the reserved2 field.

The last entry on the queue is chained to the first through its link field, which is also a PSB index, and each entry is chained to the next using its link field. If the queue contains one entry, the header points to it, and it is linked to itself. An empty queue is represented by a null index in the queue header. An overall diagram showing a hypothetical arrangement of PSB's in the various queues is contained in Figure 10.1.

10.2 Process Instructions

The process instructions are used to enter and exit monitors, to wait on condition variables and subsequently to re-enter the monitor. They are also used to notify and broadcast condition variables. Two primitives for manipulating the process queues are also available. The process instructions are all *minimal stack* (§3.3.2); that is, their operands always begin at the bottom of the stack. This minimizes the cases in which a State Vector is needed to save the stack of a process (§10.4.2).

10.2.1 Monitor Entry

The Monitor Entry instruction is executed near the beginning of each monitor entry procedure. It either sets the monitor lock, or if the monitor is already locked, causes the current process to be suspended by placing it on the monitor queue.

ME **Monitor Entry**

```
ME: PROCEDURE =
   BEGIN
   m: LONG POINTER TO Monitor = PopLong[];
   mon: Monitor;
   MinimalStack[];
   mon ← Fetch[m]↑;
   IF ~mon.locked THEN
      BEGIN
      mon.locked ← TRUE;
      Store[m]↑ ← mon;
      Push[TRUE];
      END
   ELSE EnterFailed[m];
   END;
```

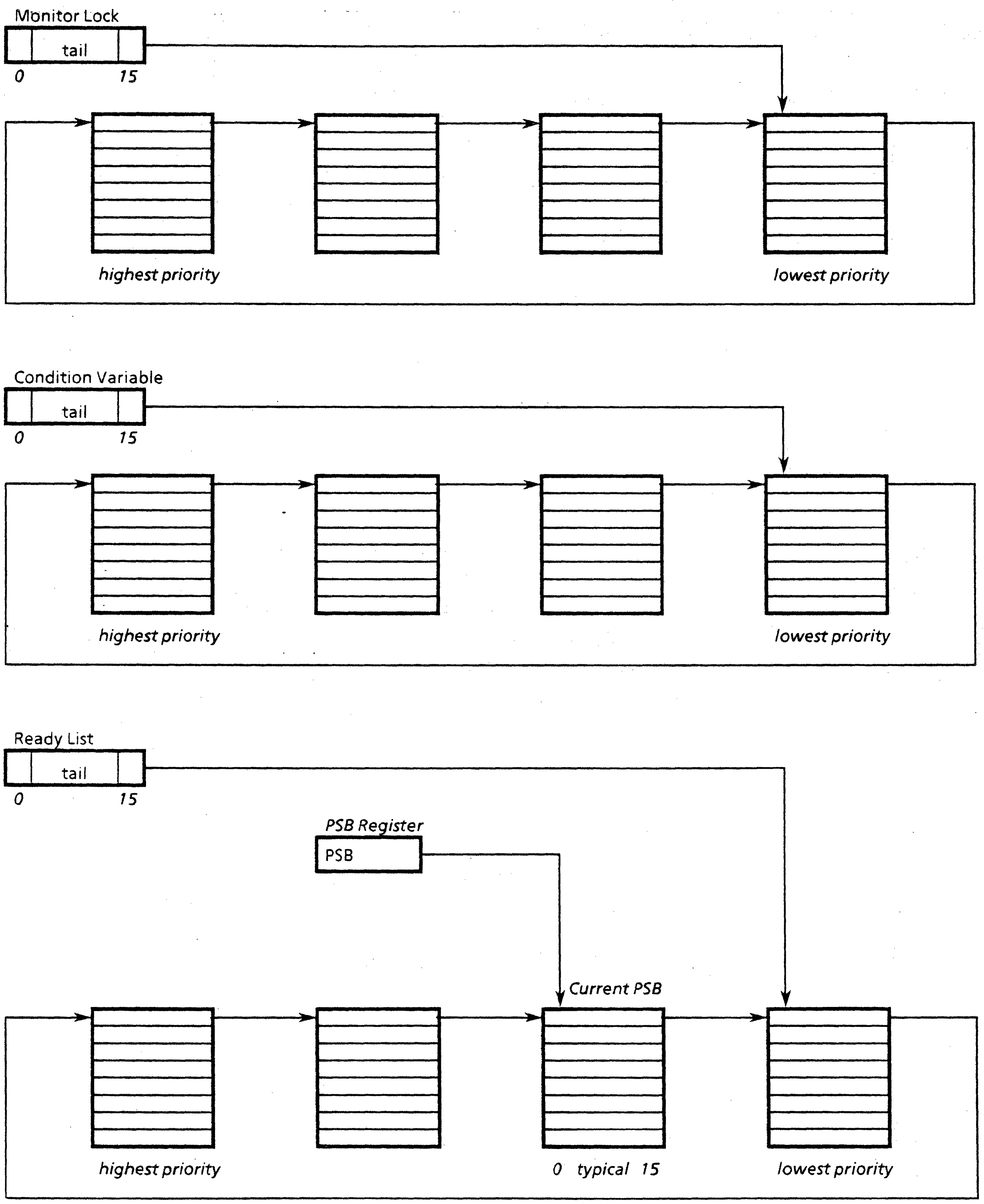


Figure 10.1 Process Queue Structures

If the monitor was entered successfully, Monitor Entry returns TRUE on the stack, which is tested by a Jump Zero Byte instruction immediately following the ME.

If monitor entry was not successful, the PSB's failed bit is set, and the current process is moved to the monitor queue. The scheduler is then invoked (§10.4.1).

```
EnterFailed: PROCEDURE [m: LONG POINTER TO Monitor] =
  BEGIN
  link: PsbLink ← Fetch[@PDA.block[PSB].link]↑;
  link.failed ← TRUE;
  Store[@PDA.block[PSB].link]↑ ← link;
  Requeue[src: @PDA.ready, dst: m, psb: PSB];
  Reschedule[];
  END;
```

When the process later becomes ready, Reschedule notices that its failed bit had been set, and places FALSE onto the evaluation stack. FALSE causes the Jump Zero instruction following the ME to loop back to the instructions to acquire the monitor lock. This allows for a situation in which some other process has locked the monitor between the time a suspended process is made ready and the time it executes the Monitor Entry. This same technique is used by the Monitor Reentry instruction (§10.2.4), which also calls EnterFailed.

10.2.2 Monitor Exit

The Monitor Exit instruction is executed at the end of each monitor entry procedure. It unlocks the monitor and causes the highest-priority process suspended on the monitor queue (if any) to be made ready.

MX **Monitor Exit**

```
MX: PROCEDURE =
  BEGIN
  m: LONG POINTER TO Monitor = PopLong[];
  MinimalStack[];
  IF Exit[m] THEN Reschedule[];
  END;
```

The Exit routine clears the monitor lock and checks the contents of the monitor queue. If the queue is not empty, the first process on the queue is made ready. The Exit routine is also used by the Monitor Wait instruction (§10.2.3).

```
Exit: PROCEDURE [m: LONG POINTER TO Monitor] RETURNS [requeue: BOOLEAN] =
  BEGIN
  mon: Monitor ← Fetch[m]↑;
  IF mon.locked = FALSE THEN ERROR;
  mon.locked ← FALSE;
  Store[m]↑ ← mon;
  IF requeue ← (mon.tail # PsbNull) THEN
    BEGIN
    link: PsbLink = Fetch[@PDA.block[mon.tail].link]↑;
    Requeue[src: m, dst: @PDA.ready, psb: link.next];
    END;
  END;
```

Programming Note: The programmer should ensure that a Monitor Exit instruction is executed only when the monitor is locked.

10.2.3 Monitor Wait

The Monitor Wait instruction is executed within a monitor to wait on a condition variable. It is always followed (statically) by a monitor re-entry sequence, which computes the monitor and condition pointers and executes a Monitor Reentry instruction (§10.2.4). Monitor Wait first unlocks the monitor, as in Monitor Exit. It then moves the current process onto the condition queue (also setting its waiting bit and timeout value) and calls the scheduler.

MW **Monitor Wait**

```
MW: PROCEDURE =
  BEGIN
  t: Ticks = Pop[];
  c: LONG POINTER TO Condition = PopLong[];
  m: LONG POINTER TO Monitor = PopLong[];
  flags: PsbFlags;
  cond: Condition;
  requeue: BOOLEAN;
  MinimalStack[];
  CleanupCondition[c];
  requeue ← Exit[m];
  flags ← Fetch[@PDA.block[PSB].flags]↑;
  cond ← Fetch[c]↑;
  IF ~flags.abort OR ~cond.abortable THEN
    BEGIN
    IF cond.wakeup THEN
      BEGIN
      cond.wakeup ← FALSE;
      Store[c]↑ ← cond;
      END
    ELSE
      BEGIN
      Store[@PDA.block[PSB].timeout]↑ ←
        IF t = 0 THEN 0
        ELSE MAX[1, LowHalf[LONG[PTC] + LONG[t]]];
      flags.waiting ← TRUE;
      Store[@PDA.block[PSB].flags]↑ ← flags;
      Requeue[src: @PDA.ready, dst: c, psb: PSB];
      requeue ← TRUE;
      END;
    END;
  IF requeue THEN Reschedule[];
  END;
```

There are two conditions under which the process executing the wait is not moved to the condition queue, but instead remains on the ready list: when the PSB of the process indicates that there is an abort pending, or when the condition variable indicates that there is a wakeup waiting (§10.4.4.2).

Monitor Wait also sets the timeout value of the process to the current value of the process timeout counter plus the time interval supplied on the stack. A value of zero indicates that the process should not be timed out while waiting. Timeout processing is described more completely in §10.4.5; CleanupCondition is defined in §10.3.2.

10.2.4 Monitor Reentry

Monitor Reentry is used to re-enter a monitor after a wait. If the monitor is locked, the process will be placed on the monitor queue as in the Monitor Entry instruction. Reentry differs from entry because Monitor Reentry will clean up the condition variable and clear the PSB's cleanup link.

MR **Monitor Reentry**

```
MR: PROCEDURE =
  BEGIN
  c: LONG POINTER TO Condition = PopLong[];
  m: LONG POINTER TO Monitor = PopLong[];
  mon: Monitor;
  MinimalStack[];
  mon ← Fetch[m]↑;
  IF ~mon.locked THEN
    BEGIN
    flags: PsbFlags;
    CleanupCondition[c];
    flags ← Fetch[@PDA.block[PSB].flags]↑;
    flags.cleanup ← PsbNull;
    Store[@PDA.block[PSB].flags]↑ ← flags;
    IF flags.abort THEN
      BEGIN
      cond: Condition = Fetch[c]↑;
      IF cond.abortable THEN ProcessTrap[];
      END;
    mon.locked ← TRUE;
    Store[m]↑ ← mon;
    Push[TRUE];
    END
  ELSE EnterFailed[m];
  END;
```

Monitor Reentry, like Monitor Entry, is always followed by a Jump Zero Byte instruction. The Jump Zero Byte loops back to the instructions to acquire the monitor lock. This loop allows for a situation in which some other process still holds or has just locked the monitor between the time the notify (or timeout) signal causes the process to be made ready and the time it executes the Monitor Reentry.

If the monitor is not locked, Monitor Reentry checks for a pending abort. If the condition variable allows aborts, a process trap is generated, so the monitor is not entered. Trap processing is described in detail in §9.5.

Programming Note: If the process trap handler intends to resume the trapped context, it must ensure that the monitor lock is acquired. This preserves the invariant that the lock is held when control is (textually) inside the monitor.

Programming Note: Between a Monitor Wait and the subsequent Monitor Reentry, a process must not execute another Monitor Wait. In particular, the program used to compute and load the monitor and condition pointers and the associated timeout interval onto the stack (and any trap routines invoked by this program) must not wait. See the programming note in section §10.3.2.

10.2.5 Notify and Broadcast

The Notify Condition and Broadcast Condition instructions are used to wake up processes waiting on condition variables. A notify moves the first entry on a condition queue to the ready queue. A broadcast makes all entries on the queue ready.

NC **Notify Condition**

```
NC: PROCEDURE =
  BEGIN
  c: LONG POINTER TO Condition = PopLong[];
  cond: Condition;
  MinimalStack[];
  CleanupCondition[c];
  cond ← Fetch[c]↑;
  IF cond.tail # PsbNull THEN
    BEGIN
    WakeHead[c];
    Reschedule[];
    END;
  END;
```

If the condition queue is empty, a notify has no effect except to clean up the queue (§10.3.2).

BC **Broadcast Condition**

```
BC: PROCEDURE =
  BEGIN
  c: LONG POINTER TO Condition = PopLong[];
  requeue: BOOLEAN;
  MinimalStack[];
  CleanupCondition[c];
  FOR cond: Condition ← Fetch[c]↑, cond ← Fetch[c]↑ WHILE cond.tail # PsbNull DO
    WakeHead[c]; requeue ← TRUE;
    ENDLOOP;
    IF requeue THEN Reschedule[];
    END;
```

The preceeding routine performs the equivalent of a Notify on each process on the condition queue. WakeHead is used to remove the head of the queue each time around the loop. If the condition queue is empty, a broadcast has no effect except to cleanup the queue (§10.3.2).

The following routine is used by the instructions Notify and Broadcast, and by the routine NotifyWakeup (§10.4.4.2). It moves the first PSB from a condition queue to the ready queue and clears the waiting flag.

```
WakeHead: PROCEDURE [c: LONG POINTER TO Condition] =
  BEGIN
  cond: Condition = Fetch[c]↑;
  link: PsbLink = Fetch[@PDA.block[cond.tail].link]↑;
  flags: PsbFlags ← Fetch[@PDA.block[link.next].flags]↑;
  flags.waiting ← FALSE;
  Store[@PDA.block[link.next].flags]↑ ← flags;
  Store[@PDA.block[link.next].timeout]↑ ← 0;
  Requeue[src: c, dst: @PDA.ready, psb: link.next];
  END;
```

WakeHead also clears the timeout value of the process, so that it will not be timed out while running. Timeouts are discussed in §10.4.5.

10.2.6 Requeue

The Requeue instruction gives programmers access to the process mechanism's queue-handling primitives. It removes a process from the source queue and inserts it (according to priority) into the destination queue, unconditionally invoking the scheduler.

REQ Requeue

```
REQ: PROCEDURE =
  BEGIN
  psb: PsbHandle = Pop[];
  dstque: QueueHandle = PopLong[];
  srcque: QueueHandle = PopLong[];
  MinimalStack[];
  Requeue[src: srcque, dst: dstque, psb: Index[psb]];
  Reschedule[];
  END;
```

Note that the Requeue instruction takes a PsbHandle, not an index.

Programming Note: In Requeue, the programmer should ensure that the psb is on the source queue (or that the source queue is zero).

10.2.7 Set Process Priority

The Set Process Priority instruction allows the programmer to change the priority of the current process.

SPP Set Process Priority

```
SPP: PROCEDURE =
  BEGIN
  priority: Priority = Pop[];
  link: PsbLink;
  MinimalStack[];
  link ← Fetch[@PDA.block[PSB].link]↑;
  link.priority ← priority;
  Store[@PDA.block[PSB].link]↑ ← link;
  Requeue[src: @PDA.ready, dst: @PDA.ready, psb: PSB];
  Reschedule[];
  END;
```

10.3 Queue Management

This section defines the small number of primitives used to maintain the process queues. In particular, operations are defined to remove a PSB from a queue (Dequeue) and to insert a PSB into a queue in priority order (Enqueue). Section 10.3.2 discusses cleanup links.

10.3.1 Queuing Procedures

The Requeue routine is used to maintain the process queue structures. It removes the process indexed by psb from the source queue src and inserts it into a destination queue dst according to its priority. Requeue is implemented using the two more primitive operations Dequeue and Enqueue.

```
Requeue: PROCEDURE [src, dst: LONG POINTER, psb: PsbIndex] =
  BEGIN
  IF psb = PsbNull THEN ERROR;
  Dequeue[src, psb];
  Enqueue[dst, psb];
  END;
```

First, Dequeue is invoked to remove the psb from the source queue. Dequeue traverses src looking for the process immediately preceeding psb (called prev), so that the psb can be removed from the queue. Dequeue then updates the queue header, if it points to the psb being removed. The algorithm is complicated by the fact that the location of the queue header (condition variable) of the source queue may not be known (src = 0). This condition occurs when a waiting process is timed out (§10.4.5) and can possibly occur when the programmer executes a Requeue instruction. In this latter case, the psb's cleanup link is set to the original value of its link field, pointing back to the source queue from which it will later be removed by CleanupCondition (described at the end of this section).

```
Dequeue: PROCEDURE [src: LONG POINTER, psb: PsbIndex] =
  BEGIN
```

```
link: PsbLink;
prev: PsbIndex;
queue: Queue;
que: QueueHandle = src;
IF que # LOOPHOLE[0] THEN queue ← Fetch[que]↑;
link ← Fetch[@PDA.block[psb].link]↑;
IF link.next = psb THEN prev ← PsbNull
ELSE
   BEGIN
   temp: PsbLink;
   prev ← IF que = LOOPHOLE[0] THEN psb ELSE queue.tail;
   DO
      temp ← Fetch[@PDA.block[prev].link]↑;
      IF temp.next = psb THEN EXIT;
      prev ← temp.next;
      ENDLOOP;
   temp.next ← link.next;
   Store[@PDA.block[prev].link]↑ ← temp;
   END;
IF que = LOOPHOLE[0] THEN
   BEGIN
   flags: PsbFlags ← Fetch[@PDA.block[psb].flags]↑;
   flags.cleanup ← link.next;
   Store[@PDA.block[psb].flags]↑ ← flags;
   END
ELSE IF queue.tail = psb THEN
   BEGIN
   queue.tail ← prev;
   Store[que]↑ ← queue;
   END;
END;
```

Enqueue inserts the psb into the destination queue dst in priority order. First, it checks for the simple case, when dst is empty. Second, Enqueue tries to add psb to the end of dst if its priority is less than or equal to the priority of the last entry in the queue. Failing that, it searches the destination queue and eventually inserts the psb after all other processes of equal or higher priority, just before the first process of lower priority.

```
Enqueue: PROCEDURE [dst: LONG POINTER, psb: PsbIndex] =
  BEGIN
  que: QueueHandle = dst;
  queue: Queue ← Fetch[que]↑;
  link: PsbLink ← Fetch[@PDA.block[psb].link]↑;
  IF queue.tail = PsbNull THEN
     BEGIN
     link.next ← psb;
     Store[@PDA.block[psb].link]↑ ← link;
     queue.tail ← psb;  Store[que]↑ ← queue;
     END
  ELSE
     BEGIN
     currentlink, nextlink: PsbLink;
```

```
        prev: PsbIndex ← queue.tail;
        currentlink← Fetch[@PDA.block[prev].link]↑;
        IFcurrentlink.priority >= link.priority THEN
          BEGIN
          queue.tail ← psb;
          Store[que]↑ ← queue;
          END
        ELSE
          DO
            nextlink ← Fetch[@PDA.block[currentlink.next].link]↑;
            IF link.priority > nextlink.priority THEN EXIT;
            prev ←currentlink.next;  currentlink← nextlink;
            ENDLOOP;
        link.next ←currentlink.next;  Store[@PDA.block[psb].link]↑ ←link;
        currentlink.next ← psb;  Store[@PDA.block[prev].link]↑ ←currentlink;
        END;
      END;
```

10.3.2 Cleanup Links

The CleanupCondition routine must be invoked before accessing a condition queue, since its queue pointer may not be correct. Inaccuracy occurs when the tail of a condition queue (pointed to by the header) is removed from the queue by a timeout: the location of the header is unknown in that case, so the pointer cannot be properly updated. (This situation may also occur when using the Requeue instruction.) Fortunately, in addition to the link described above, each PSB also contains a second queue link, called the cleanup link, which is used to maintain the queue structures when the location of the queue header is not known.

When Dequeue detects this situation (the source queue is zero), it sets the PSB's cleanup link to the old value of its link field, which points to the next PSB on the queue. CleanupCondition finds the correct head of the condition queue by following the cleanup link into the queue from which the PSB was removed. From there, it locates the tail to which the condition variable should point. Notice, however, that the cleanup link might point to a PSB that also has its cleanup link set because it was also removed from the queue by a timeout! CleanupCondition therefore follows the cleanup links until there are no more, declares the resulting PSB to be the head of the queue, and then follows the normal queue links until the tail is found.

```
CleanupCondition: PROCEDURE [c: LONG POINTER TO Condition] =
  BEGIN
  link: PsbLink;
  flags: PsbFlags;
  psb, head: PsbIndex;
  cond: Condition ← Fetch[c]↑;
  IF (psb ← cond.tail) # PsbNull THEN
    BEGIN
    flags ← Fetch[@PDA.block[psb].flags]↑;
    IF flags.cleanup # PsbNull THEN
      BEGIN
      DO
```

```
          IF flags.cleanup = psb THEN
            BEGIN
            cond.wakeup ← FALSE;
            cond.tail ← PsbNull;
            Store[c]↑ ← cond;
            RETURN;
            END;
          psb ← flags.cleanup;
          flags ← Fetch[@PDA.block[psb].flags]↑;
          IF flags.cleanup = PsbNull THEN EXIT;
          ENDLOOP;
        head ← psb;
        DO
          link ← Fetch[@PDA.block[psb].link]↑;
          IF link.next = head THEN EXIT;
          psb ← link.next;
          ENDLOOP;
        cond.tail ← psb;
        Store[c]↑ ← cond;
        END;
      END;
    END;
```

Note that CleanupCondition itself updates only the condition variable; the cleanup links in the PSBs removed from the condition queue are reset by the Monitor Reentry instruction (§10.2.4).

Programming Note: Between a Monitor Wait and the subsequent Monitor Reentry, a process must not execute another Monitor Wait; in particular, the program used to compute and load the monitor and condition pointers and the timeout interval onto the stack (and any trap routines invoked by that program) must not wait. If the first wait times out, the PSB's cleanup link will be set. Any subsequent wait would destroy the original cleanup link. Also, any fault that occurs between a Monitor Wait and the subsequent Monitor Reentry will result in the process being requeued, first, to a fault service queue and, later, to the Ready Queue again. For example, a page fault on the code page is possible. The first of these requeues is carried out without calling CleanUpCondition, and the second must also avoid CleanUpCondition, or the cleanup link will be smashed. For this reason, the process must removed from the fault service queue to the Ready Queue by means of the FEQ opcode; Notify Condition and Broadcast Condition must not be used.

Design Note: CleanupCondition is an idempotent operation; that is, cleaning up a condition variable that is already clean has no effect. It is therefore permissible for an instruction to clean up a condition variable before checking for other possible traps or faults.

Design Note: Because only processes waiting on condition variables can be timed out, there is no need for a corresponding routine to clean up monitor locks.

10.4 Scheduling

The scheduler implements overall changes in the machine state called *process switches*. Process switches result when the current process yields control of the processor and a higher-priority process is ready, or when the current process is removed from the ready queue, either by its own commission or by a pre-emption. The current process may stop running as a result of performing any of the following actions:

- attempting to enter a monitor,
- exiting a monitor,
- waiting on a condition variable,
- attempting to reenter a monitor, or
- executing a REQ or SPP instruction.

In addition, any of the following pre-emptions may cause the current process to stop running:

- a fault, which notifies a fault handler (§10.4.3),
- an interrupt, which notifies a condition variable (§10.4.4), or
- a timeout, which makes a waiting process ready (§10.4.5).

All of these conditions result in a possible process switch by calling the scheduler (Reschedule) described in the next section. In all cases, process switches take place between instructions. Either the current instruction is completed, or the state of the processor at the beginning of the current instruction is restored. See §4.6.1 for a precise statement of the restart rule.

10.4.1 Scheduler

The scheduler is invoked whenever a process has moved to or from the ready queue. In particular, Reschedule is called by the process instructions that call Requeue and by the Fault, Interrupt, and TimeoutScan routines (described in later sections).

Reschedule finds the highest priority runnable process in the ready queue, saves the state of the current process (if any), and loads the state of the new process into the processor registers. In the case that no runnable process exits, the scheduler sets running to false. This action causes the main loop (§4.1) to cease instruction execution (it continues to check for interrupts and timeouts).

To be runnable, the process must have a state vector in which to save the evaluation stack and context, in case the process is pre-empted, by a fault, interrupt, or timeout. Reschedule therefore checks whether the process has a permanent state vector (link.permanent = TRUE), the process' context already points to a state vector (link.preempted = TRUE) or a state vector for the process' priority is otherwise available (see §10.4.2.2).

```
running: BOOLEAN;

Reschedule: PROCEDURE [preemption: BOOLEAN ← FALSE] =
  BEGIN
  link: PsbLink;
  psb: PsbIndex;
  queue: Queue;
  IF running THEN SaveProcess[preemption];
  queue ← Fetch[@PDA.ready]↑;
  IF queue.tail = PsbNull THEN GO TO BusyWait;
  link ← Fetch[@PDA.block[queue.tail].link]↑;
  DO
    psb ← link.next; link ← Fetch[@PDA.block[psb].link]↑;
    IF link.permanent OR link.preempted OR ~EmptyState[link.priority] THEN EXIT;
    IF psb = queue.tail THEN GO TO BusyWait;
    ENDLOOP;
  PSB ← psb; PC ← savedPC ← 0;
  LF ← LoadProcess[]; running ← TRUE;
  XFER[dst: LONG[LF], src: 0, type: processSwitch];
  EXITS
    BusyWait =>
      BEGIN
      IF ~InterruptsEnabled[] THEN RescheduleError[];
      running ← FALSE;
      END;
  END;
```

Design Note: It is an invariant of the design that PDA.state[PDA.block[PSB].link.priority] # 0 OR link.permanent. That is, a state vector must be available in case the current process is preempted (see §10.4.2).

After saving the state of the current process (if any), Reschedule clears the PC register (and resets savedPC) to indicate that a process switch is in progress. This is necessary in case the subsequent XFER causes a trap or fault, which normally saves the PC in the current frame. This must be avoided if the frame is mapped out or the PC has not yet been loaded (because of a fault on the global frame, for example).

Implementation Note: Any invalid value of the PC can be used in the implementation, as indicated by the following routine, which returns FALSE if a process switch is in progress. Eight is the minimum size of a code segment entry vector.

```
ValidContext: PROCEDURE RETURNS [BOOLEAN] =
  BEGIN
  RETURN[PC >= SIZE[CodeSegment]*2];
  END;
```

If no runnable process can be found on the ready queue, Reschedule sets running to FALSE, causing the instruction interpreter (§4.1) to enter a "busy wait" loop, waiting for an interrupt (§10.4.4) or a timeout (§10.4.5). When in this state, interrupts must be enabled; otherwise, a trap occurs in the context of the last running process (§9.5).

Programming Note: A RescheduleError is fatal. If the trap handler attempts to resume normal process switching, the results are undefined. This condition is reported by the processor for debugging purposes only.

10.4.2 Process State

A process' state is saved in its PSB and perhaps also in one of a number of state vectors allocated to the process' priority level. A state vector preserves the process' stack and local frame pointer, and also has room for a fault parameter (the format is defined in §9.5.3). The process state saving and restoring routines and a description of the algorithms for state vector allocation and deallocation are included in this section.

10.4.2.1 Saving and Loading Process State

The SaveProcess and LoadProcess routines are used by the scheduler to save the state of a running process and to reload the state of a ready process. The SaveStack and LoadStack routines are defined in §9.5.3.

The stack is empty when a process is moved to or from the ready queue by the process instruction. For that reason, a state vector is needed only in the case of a pre-emption caused by a fault, an interrupt, or a timeout. The state vector is obtained using the AllocState routine described in the next section. Otherwise, the state of the process is contained entirely within the PSB.

```
SaveProcess: PROCEDURE [preemption: BOOLEAN] =
  BEGIN ENABLE Abort = > ERROR;
  link: PsbLink ← Fetch[@PDA.block[PSB].link] ↑ ;
  IF ValidContext[] THEN StoreMds[@LocalBase[LF].pc] ↑ ← PC;
  IF link.preempted ← preemption THEN
    BEGIN
    state: StateHandle;
    IF ~link.permanent THEN state ← AllocState[link.priority]
    ELSE state ← LengthenPdaPtr[Fetch[@PDA.block[PSB].context] ↑ ];
    SaveStack[state];
    Store[@state.frame] ↑ ← LF;
    IF ~link.permanent THEN
      Store[@PDA.block[PSB].context] ↑ ← OffsetPda[state];
    END
  ELSE
    IF ~link.permanent THEN Store[@PDA.block[PSB].context] ↑ ← LF
    ELSE
      BEGIN
      state: StateHandle ←
        LengthenPdaPtr[Fetch[@PDA.block[PSB].context] ↑ ];
      Store[@state.frame] ↑ ← LF;
      END;
  Store[@PDA.block[PSB].link] ↑ ← link;
  END;
```

Design Note: The ENABLE for Abort indicates that SaveProcess (and AllocState and SaveStack) can not generate page faults (or write-protect faults). The state vector must be resident because the fault parameter is saved in it. (see §10.4.3).

SaveProcess must check that the context is valid before saving the PC in the current local frame. This check covers the case of a trap or fault during a process switch that has not yet obtained a valid context. SaveStack handles the correct setting of savedSP as well as the stack pointer SP.

Programming Note: Saving the process state does not update the mds field of the PSB. If the program modifies the MDS register, it should also update the current PSB (if that is the effect desired).

The LoadProcess routine reverses the actions of SaveProcess, freeing the state vector if one was allocated. Note that LoadStack sets savedSP as well as SP to the value obtained from the state vector.

```
LoadProcess: PROCEDURE RETURNS [frame: LocalFrameHandle] =
  BEGIN ENABLE Abort = > ERROR;
  mds: CARDINAL;
  link: PsbLink ← Fetch[@PDA.block[PSB].link] ↑ ;
  frame ← Fetch[@PDA.block[PSB].context] ↑ ;
  IF link.preempted THEN
    BEGIN
    state: StateHandle ← LengthenPdaPtr[frame];
    LoadStack[state];
    frame ← Fetch[@state.frame] ↑ ;
    IF ~link.permanent THEN FreeState[link.priority, state];
    END
  ELSE
    BEGIN
    IF link.failed THEN
      BEGIN
      Push[FALSE]; link.failed ← FALSE;
      Store[@PDA.block[PSB].link] ↑ ← link;
      END;
    IF link.permanent THEN
      BEGIN
      state: StateHandle ← LengthenPdaPtr[frame];
      frame ← Fetch[@state.frame] ↑ ;
      END;
    END;
  mds ← Fetch[@PDA.block[PSB].mds] ↑ ;
  MDS ← LongShift[LONG[mds], WordSize];
  END;
```

Design Note: The ENABLE for Abort indicates that LoadProcess (and LoadStack and FreeState) can not generate page faults (or write-protect faults); until the LoadProcess has completed, the state vector is unavailable for reuse by a subsequent fault.

LoadProcess checks for the presence of the failed bit set by the Monitor Entry and Monitor Reentry instructions. In this case, it pushes FALSE onto the stack so that the following Jump Zero Byte instruction will loop back to the monitor entry sequence (see §10.2.1).

Implementation Note: In the case of a fault (see §10.4.3), some implementations of the processor may save additional processor state in the state vector for use by a subsequent load. However, except for the size of the state vector required (which must be constant for all processes), the presence of this additional information *must be invisible to the programmer.*

10.4.2.2 State Vector Allocation

State vectors are allocated much like frames, using an array of pointers to lists of free state vectors called the State Allocation Table (SAT). A separate list is provided for each priority level, but unlike frame allocation, there is no provision for indirect lists. All pointers in the SAT are relative to the base of the Process Data Area.

```
StateAllocationTable: TYPE = ARRAY Priority OF POINTER TO StateVector;
```

The scheduler uses the following routine to ensure that a state vector is available at the correct priority level before running a process:

```
EmptyState: PROCEDURE [pri: Priority] RETURNS [BOOLEAN] =
  BEGIN
  state: POINTER TO StateVector = Fetch[@PDA.state[pri]]↑;
  RETURN[state = LOOPHOLE[0]];
  END;
```

The allocation routine simply returns the element of the array indexed by the requested priority, updating the array to point to the next item on the list. FreeState returns the state vector to the head of the list in the obvious way.

```
AllocState: PROCEDURE [pri: Priority] RETURNS [state: StateHandle] =
  BEGIN
  offset: POINTER = Fetch[@PDA.state[pri]]↑;
  IF offset = LOOPHOLE[0] THEN ERROR;
  state ← LengthenPdaPtr[offset];
  Store[@PDA.state[pri]]↑ ← Fetch[state]↑;
  RETURN[state];
  END;

FreeState: PROCEDURE [pri: Priority, state: StateHandle] =
  BEGIN
  Store[state]↑ ← Fetch[@PDA.state[pri]]↑;
  Store[@PDA.state[pri]]↑ ← OffsetPda[state];
  END;
```

There is no provision for trapping or faulting when a state-vector list is empty. The scheduler guarantees that, when it runs a process, there is a state vector available for the subsequent pre-emption that may occur.

Programming Note: Because there must be one state vector for each pre-emptible process, the number of state vectors in each list determines the degree of pre-emptive multi-programming allowed at the corresponding priority-level.

10.4.3 Faults

A *fault* is an exception that causes a process switch. There are three such exceptions: a page fault, a write protect fault, and a frame-allocation fault. Each type of fault has an associated queue where faulted processes are kept, as well as an associated fault-handler, represented by a condition variable that is notified when the fault occurs. This information is organized as a substructure of the PDA.

```
FaultVector: TYPE = ARRAY FaultIndex OF FaultQueue;

FaultIndex: TYPE = [0..8);

FaultQueue: TYPE = MACHINE DEPENDENT RECORD [
  queue (0): Queue,
  condition (1): Condition];
```

Fault processing is logically much like trap processing (§9.5.2), with the following differences:

- The fault parameter is saved in the state vector of the current (faulted) process, rather than in the frame of the trap handler.
- The fault results in a process switch rather than a control transfer. The fault handler can therefore run in a Main Data Space different from the faulted process.

The precise actions that must be taken by the processor when a fault occurs are shown by FaultOne (single word parameter), FaultTwo (double word parameter), and the common Fault routine.

```
FaultOne: PROCEDURE [fi: FaultIndex, parameter: UNSPECIFIED] =
  BEGIN
  psb: PsbIndex = Fault[fi];
  state: POINTER TO StateVector = Fetch[@PDA.block[psb].context]↑;
  StorePda[@state.data[0]]↑ ← parameter;
  ERROR Abort;
  END;

FaultTwo: PROCEDURE [fi: FaultIndex, parameter: LONG UNSPECIFIED] =
  BEGIN
  psb: PsbIndex = Fault[fi];
  state: POINTER TO StateVector = Fetch[@PDA.block[psb].context]↑;
  StorePda[@state.data[0]]↑ ← LowHalf[parameter];
  StorePda[@state.data[1]]↑ ← HighHalf[parameter];
  ERROR Abort;
  END;

Fault: PROCEDURE [fi: FaultIndex] RETURNS [PsbIndex] =
  BEGIN
  faulted: PsbIndex = PSB;
```

```
Requeue[src: @PDA.ready, dst: @PDA.fault[fi].queue, psb: faulted];
[] ← NotifyWakeup[@PDA.fault[fi].condition];
PC ← savedPC; SP ← savedSP;
Reschedule[preemption: TRUE];
RETURN[faulted];
END;
```

Fault saves the PSB index of the faulted process, moves the process to the appropriate fault queue, and notifies the fault handler using NotifyWakeup (because there is no monitor controlling access to the fault condition variables; see §10.4.4.2). It then restores the PC and SP and invokes the scheduler. The fault routines store their parameters in the state vector of the faulted process. The three possible faults have the following parameters:

```
FrameFault: PROCEDURE [fsi: FSIndex] = {FaultOne[qFrameFault, fsi]};
```

A frame of the requested size (or larger) could not be allocated. The value of the parameter is the frame size index requested by XFER or the Allocate Frame instruction (§9.2.2).

```
PageFault: PROCEDURE [ptr: LONG POINTER] = {FaultTwo[qPageFault, ptr]};

WriteProtectFault: PROCEDURE [ptr: LONG POINTER] = {
  FaultTwo[qWriteProtectFault, ptr]};
```

An access to an unmapped page (a store into a write -rotected page) was attempted. The value of the fault parameter is the virtual address used by the memory reference that faulted (§3.1.1).

The sizes of fault parameters determine the value of cSV, the minimum size of a state vector; it is defined in the Appendix.

Because the fault routine always re-establishes the initial state of a faulted instruction, multiple faults on a single instruction are invisible to the programmer, and there is no need for the fault handler to concern itself with possible partial side effects of the instruction or with the continuation of partially completed instructions.

Programming Note: After correcting the cause of the fault, the fault handler must use the Requeue instruction (instead of NC or BC) to remove the faulted process from the fault queue and make it ready.

10.4.4 Interrupts

An array of reserved condition variables is allocated in the Process Data Area for servicing interrupts on one of sixteen interrupt-levels. An interrupt is implemented by notifying one of these conditions:

```
InterruptVector: TYPE = ARRAY InterruptLevel OF InterruptItem;

InterruptLevel: TYPE = [0..WordSize);

InterruptItem: TYPE = MACHINE DEPENDENT RECORD [
  condition (0): Condition, available (1): UNSPECIFIED];
```

When an external event occurs that requires service from an interrupt process, a wakeup is generated. Wakeups include signals from devices and controllers, and perhaps also internal signals within the processor (for example, clocks or timers). Wakeups are held pending until completion of the current instruction, when they are translated into interrupts by the processor.

Design Note: Some interrupt levels may be reserved for internal use by the processor. One level typically is used to implement the check for timeouts (§10.4.5). The (read-only) wakeup mask register indicates the reserved levels (defined in Appendix A):

```
WM: UNSPECIFIED = cWM;
```

This register has the same format as **WP**; bit *i* corresponds to interrupt level *i* (see below).

10.4.4.1 Checking for Interrupts

Because interrupts occur only between instructions, wakeups received during the execution of the current instruction are buffered in the wakeup pending register **WP**. A device or controller requesting service from the process assigned to interrupt level *i* sets bit *i* of this register to one. It must do so atomically with respect to reads and writes of this register performed by other devices and the processor.

```
WP: UNSPECIFIED;
```

Before fetching and executing each instruction, the processor calls CheckForInterrupts, which invokes the following routine to test for the presence of an interrupt:

```
InterruptPending: PROCEDURE RETURNS [BOOLEAN] =
  BEGIN
  RETURN[WP # 0 AND InterruptsEnabled[]];
  END;
```

Implementation Note: For maximum execution speed, the **WP** # 0 test should always be done before the InterruptsEnabled[] test, since interrupts are almost always enabled.

The **WP** register saves only one wakeup request for each interrupt level per instruction execution, and the check for pending wakeups is made once before the beginning of each instruction (except for interruptible instructions; see below) by the main loop of the instruction interpreter (§4.1). A particular implementation of the processor may check for interrupts more or less often than shown in the description, as long as it meets the following requirements:

> Interrupts must always appear to the programmer to happen between the execution of two instructions. If a wakeup is noticed after the execution of an instruction has begun, the processor must be restored to its state at the beginning of the instruction before the wakeup can be translated into an interrupt; alternately, the processor may complete the current instruction before processing the wakeup. Certain instructions with long execution times are interruptible, for example, the block transfer instructions (§8). They check for pending wakeups during execution and make special provisions for restart in the event of an interrupt.

The worst case response to the highest-priority interrupt will occur when the interrupt request is raised in conjunction both with the timeout scan and with an opcode that requires a long execution time without interrupt checks. To avoid making response time worse than it must be, opcodes should check for interrupts at intervals small compared to the timeout scan.

Programming Note: The software must be written to be robust in the face of lost wakeups. The timeout mechanism described in the next section is designed to assist the programmer with this requirement.

10.4.4.2 Interrupt Processing

The Interrupt routine translates each pending wakeup into a notify of the condition associated with that wakeup's interrupt level. Higher interrupt levels are always processed first. Notice that the interrupt level is independent of the priority of the process waiting on the condition variable. The level affects only the order in which interrupt processes with the same priority are moved to the ready queue.

```
CheckForInterrupts: PROCEDURE RETURNS [BOOLEAN] =
  BEGIN
  IF InterruptPending[]
    THEN RETURN[Interrupt[]]
    ELSE RETURN[FALSE];
  END;

Interrupt: PROCEDURE RETURNS [BOOLEAN] =
  BEGIN
  mask: UNSPECIFIED ← 1;
  wakeups: UNSPECIFIED;
  requeue: BOOLEAN ← FALSE;
  wakeups ← WP;  WP ← 0;
  FOR level: InterruptLevel DECREASING IN InterruptLevel DO
    IF And[wakeups, mask] # 0 THEN
      NotifyWakeup[@PDA.interrupt[level].condition] OR
        requeue ← requeue;
    mask ← Shift[mask, 1];
    ENDLOOP;
  RETURN[requeue];
  END;
```

Implementation Note: The two assignment operations wakeups ← WP and WP ← 0 must be performed atomically with respect to devices that write into the wakeup pending register.

Both faults and interrupts move a process to the ready queue by performing a variation of the standard notify operation (§10.2.5). Condition variables in the interrupt vector (and in the fault vector) make use of an additional bit. It records a wakeup in case no process is waiting on the condition. This record is necessary because another process (the device, in this case) can notify the condition without entering the monitor that normally protects a condition variable. If the device notifies the interrupt process between the time the process decides to wait on the condition variable (for example, it checks the status and finds the device busy) and the time the process actually executes the wait instruction, the notify would be lost. To prevent this, the processor converts a pending wakeup (and a fault) into a NotifyWakeup, which sets a wakeup bit in the condition variable if no process is waiting on the condition.

```
NotifyWakeup: PROCEDURE [c: LONG POINTER TO Condition] RETURNS [BOOLEAN] =
  BEGIN
  cond: Condition;
  requeue: BOOLEAN ← FALSE;
  CleanupCondition[c];
  cond ← Fetch[c]↑;
  IF cond.tail = PsbNull THEN
    BEGIN
    cond.wakeup ← TRUE;
    Store[c]↑ ← cond;
    END
  ELSE
    BEGIN
    WakeHead[c];
    requeue ← TRUE;
    END;
  RETURN[requeue];
  END;
```

The Monitor Wait instructions (§10.2.3) do not wait when the wakeup bit is set.

Programming Note: Since interrupts (and faults) perform a notify rather than a broadcast, only a single process should be waiting on the conditions in the interrupt vector and the fault queue.

10.4.4.3 Disabling Interrupts

Generation of interrupts is controlled by the wakeup disable counter **WDC**. It counts the number of times interrupts have been disabled. The minimum value of WdcMax is given in Appendix A.

```
WDC: CARDINAL;
WdcMax: CARDINAL = cWDC;

InterruptsEnabled: PROCEDURE RETURNS [BOOLEAN] =
  BEGIN
  RETURN[WDC = 0];
  END;

DisableInterrupts: PROCEDURE =
  BEGIN
  WDC ← WDC + 1;
  END;

EnableInterrupts: PROCEDURE =
  BEGIN
```

```
    WDC ← WDC - 1;
    END;
```

The instructions shown below allow the programmer to disable and enable interrupts. They generate a trap if the wakeup disable counter would be incremented or decremented out of range (§9.5).

DI **Disable Interrupts**

```
DI: PROCEDURE =
  BEGIN
  IF WDC = WdcMax THEN InterruptError[];
  DisableInterrupts[];
  END;
```

EI **Enable Interrupts**

```
EI: PROCEDURE =
  BEGIN
  IF WDC = 0 THEN InterruptError[];
  EnableInterrupts[];
  END;
```

Programming Note: A counter (rather than a flag) allows the programmer to disable and enable interrupts without regard to the previous state of the register, provided that the maximum value of the counter is not exceeded.

10.4.5 Timeouts

A process can be timed out by the processor, so it does not wait indefinitely on a condition queue. When a process executes a Monitor Wait instruction (§10.2.3), it specifies a time interval that limits the amount of time the process will spend in the condition queue. If the process is still on the queue after this interval has elapsed, it will be made ready by the processor. When the process next executes, it appears to the programmer as if it had received a notify. Thus, lost notifies will not cause processes to wait forever.

To implement timeouts, each PSB has a timeout field which indicates when its corresponding process should be timed out. The Monitor Wait instruction (§10.2.3) generates this value by adding its time interval operand to the current value of the time (obtained from a processor register; see below). A value of zero indicates that the process should not be timed out. Unless a process is waiting on a condition queue, its timeout is always zero; that is, only waiting processes can be timed out.

Timeouts are measured in ticks, where the conversion between ticks and real time is processor-dependent. A tick is on the order of 40 milliseconds. The upper and lower limits on the size of a tick are specified in Appendix A. TimeOutInterval is the size of the timeout interval measured in the units of the interval timer IT (§3.3.3).

```
Ticks: TYPE = CARDINAL;
TimeOutInterval: LONG CARDINAL;
```

Design Note: The size of the timeout interval should be chosen so that the overhead of checking for process timeouts is acceptably low. Consider the expected number of processes in the system and the time required to perform the timeout scan, along with the minimum and maximum available timeout intervals.

The current value of the time, measured in ticks, is kept in a programmer-accessible processor register called the *process timeout counter*. The accuracy of this timer measured against real time is not specified by the architecture.

```
PTC: Ticks;
```

Programming Note: Because the accuracy of the PTC is unspecified, programmers should use the interval timer IT whenever accurate knowledge of real or elapsed time is necessary. Moreover, there is no guarantee that a timeout will occur within the interval specified by a wait instruction, only that it will occur at approximately that time.

The CheckForTimeouts routine is called by the main loop of the processor (§4.1). It checks to see if a timeout interval has elapsed by comparing the current value of the interval timer with its value at the last call (saved in the private global variable time). If interrupts are enabled and a timeout interval has elapsed, the processor increments the process timeout counter and calls TimeoutScan to check for PSBs that should be timed out.

```
time: LONG CARDINAL;

CheckForTimeouts: PROCEDURE RETURNS [BOOLEAN] =
  BEGIN
  temp: LONG CARDINAL = IT;
  IF InterruptsEnabled[]
  AND temp - time > = TimeOutInterval THEN
    BEGIN
    time ← temp; PTC ← PTC + 1;
    IF PTC = 0 THEN PTC ← PTC + 1;
    RETURN[TimeoutScan[]];
    END
  ELSE RETURN[FALSE];
  END;
```

Programming Note: The Process Timeout Counter does not tick while interrupts are disabled. If interrupts remain disabled for an extended period, the processor makes no attempt to timeout processes that should have been notified during that period.

Implementation Note: The implementation need not follow the above algorithm exactly, as long as it appears to the programmer that a timeout occurs only between instructions, and that the scan occurs at intervals of approximately one timeout interval. In particular, the scan may be initiated by an interrupt internal to the processor, rather than by a call from within the main loop, or the scan can be done in parallel with the processor, as long as the timeout itself is properly synchronized with instruction execution.

The timeout scan examines the timeout of each process in the timeout vector. Processes with zero timeouts are ignored. If the timeout is equal to the current value of the process timeout counter, the timeout is cleared and the process is moved from an unknown condition queue to the ready queue. At the end of the scan, if any processes have been made ready, the routine returns TRUE.

```
TimeoutScan: PROCEDURE RETURNS [BOOLEAN] =
  BEGIN
  requeue: BOOLEAN ← FALSE;
  count: CARDINAL = Fetch[@PDA.count]↑;
  FOR psb: PsbIndex IN [StartPsb..StartPsb + count) DO
    timeout: Ticks ← Fetch[@PDA.block[psb].timeout]↑;
    IF timeout # 0 AND timeout = PTC THEN
      BEGIN
      flags: PsbFlags ← Fetch[@PDA.block[psb].flags]↑;
      flags.waiting ← FALSE;
      Store[@PDA.block[psb].flags]↑ ← flags;
      Store[@PDA.block[psb].timeout]↑ ← 0;
      Requeue[src: LOOPHOLE[LONG[0]], dst: @PDA.ready, psb: psb];
      requeue ← TRUE;
      END;
    ENDLOOP;
  RETURN[requeue];
  END;
```

Because the condition queue containing the process is unknown, zero is passed as the source queue to Requeue, which makes special provisions for handling this case (§10.3).

Design Note: It is the programmer's responsibility to ensure that the count reflects the PSBs that should be examined, and that the timeout of any PSB not representing an active process is set to zero (see §10.1.1).

A

Values of Constants

Appendix A defines the values of constants referenced in the preceding chapters. Where applicable, each constant description identifies the section where the constant is defined.

1. Miscellaneous Constants

The stack size defines the maximum number of sixteen-bit words contained in the evaluation stack. Notice that the stack pointer must represent values in the range [0..cSS] inclusive (§3.3.2).

cSS Stack Size

```
cSS: CARDINAL = 14;
```

The minimum size of a state vector is defined by cSV (§9.5.3). It includes enough space for the control link used by the Load Stack instructions (LSK) and for the longest fault parameter (§10.4.3). The actual size is processor dependent (§10.4.2.1).

cSV State Vector Size

```
cSV: CARDINAL = SIZE[StateVector] + MAX[
    SIZE[ControlLink], SIZE[FSIndex], SIZE[LONG POINTER]];
```

The contents of the wakeup mask register is processor dependent. It records the interrupt levels reserved for internal use by the processor (§10.4.4).

cWM Wakeup Mask

```
cWM: CARDINAL;
```

The maximum value of the wakeup disable counter is processor dependent, but must be greater than or equal to cWDC (§10.4.4).

cWDC Wakeup Disable Counter

```
cWDC: CARDINAL = 7;
```

The minimum and maximum durations of a tick, measured in milliseconds, are given by the following constants (§10.4.5).

cTick Mininum and Maximum Tick Size

```
cTickMin: CARDINAL = 15;
cTickMax: CARDINAL = 60;
```

2. Constant Memory Locations

The following constants define the locations of the fixed data structures of the architecture. Notice that the MDS data structures appear at the same relative location in each Main Data Space.

To keep it from interfering with the booting process, the Process Data Area is located at the beginning of the second bank (§10.1.1).

mPDA Process Data Area

```
mPDA: LONG CARDINAL = 200000B;
```

The frame Allocation Vector starts at page one in each Main Data Space (§9.2.1).

mAV Allocation Vector

```
mAV: CARDINAL = 400B;
```

The System Data table starts at page two in each Main Data Space (§9.5.1).

mSDSystem Data Table

```
mSD: CARDINAL = 1000B;
```

The ESC Trap table begins at page three in each Main Data Space. It is a maximum of four pages long (§9.1.3.1).

mETT ESC Trap Table

```
mETT: CARDINAL = 2000B;
```

3. Fault Queue Indexes

Three of the possible eight entries in the fault vector are used by the processor. The remaining entries are reserved (§10.4.3).

qFQ Fault Queue Indexes

```
qFrameFault: CARDINAL = 0B; qPageFault: CARDINAL = 1B;
qWriteProtectFault: CARDINAL = 2B;
```

4. System Data Table Indexes

Fifteen of the possible 192 entries in the System Data table are used by the processor. Other entries in the range [0..37B] are reserved. The remaining entries are available for use by the software (§9.5.1). See also the KFCB instruction (§9.4.2).

sSD System Data Table Indexes

```
sBoot: CARDINAL = 1B;              sBoundsTrap: CARDINAL = 16B;
sBreakTrap: CARDINAL = 0B;         sCodeTrap: CARDINAL = 7B;
sControlTrap: CARDINAL = 6B;       sDivCheckTrap: CARDINAL = 13B;
sDivZeroTrap: CARDINAL = 12B;      sInterruptError: CARDINAL = 14B;
sOpcodeTrap: CARDINAL = 5B;        sPointerTrap: CARDINAL = 17B;
sProcessTrap: CARDINAL = 15B;      sRescheduleError: CARDINAL = 3B;
sStackError: CARDINAL = 2B;        sUnboundTrap: CARDINAL = 11B;
sXferTrap: CARDINAL = 4B;          sHardwareError: CARDINAL = 10B;
```

Design Note: The location of sBoot must be equal to two modulo four, so that the initial XFER (§4.7) will interpret it as an indirect control link (§9.1.2).

5. Opcode Assignments

The opcode assignments, which are the same for each implementation of the processor, are under development.

B

Opcodes

Approximate conventions in opcode names:

initial letter:

J -	Jump. Conditional and unconditional jump instructions
L -	Load. Simple load of local or global variable
P -	Put. Store (simple or indirect) but leave the value on the stack
R -	Read. Indirect load through pointer; if pointer is on stack, replace it with value.
S -	Store. Simple store of local or global variable
W -	rite. Indirect store through pointer

terminal letter(s):

B -	Byte. The next byte is an eight-bit datum
BB -	Byte Byte. The next two bytes are a pair of eight-bit data
F -	Field. The next two bytes are an eight-bit word offset and two four-bit numbers indicating position and size of a subword field. When the F is preceded by a number, that number is the offset and only the pos/size byte follows.

n - Represents a family of instructions having a range of values for n.
P - Pair. The next byte is a pair of four-bit data
W - Word. The next two bytes are a sixteen-bit datum (high byte first)

other letters:

C -	Code.
D -	Double. Refers to operations on 32-bit data.
FC -	Function Call.
G -	Global.
I -	Indirect.
L -	Local.
L -	Long. Refers to operations using 32-bit pointers.
S -	Signed.
S -	Stack.
S -	String. Involves indexing memory as an array of bytes.
S -	Swapped. The "normal" order of data then pointer for Write instructions is reversed.
U -	Unsigned. For 16- and 32-bit unsigned comparisons.

Description conventions:

Data values and pointers are sixteen-bit unless identified as 32-bit. When the potential for confusion exists, both 16- and 32-bit quantities are noted. All sixteen-bit pointers are MDS relative.

The first operand byte is called alpha, and the second is called beta. In the case of subdivided bytes, the nibbles are called alpha.left and alpha.right (or beta.left and beta.right). If two operand bytes are considered as a sixteen-bit quantity, it is called alphabeta and is equal to alpha*256+beta. Sign-extended operand bytes are sxalpha and sxbeta. The high and low bytes of sixteen-bit data are called data.high and data.low.

L is the address of the current local frame; G is the address of the current global frame; C is the address of the current code segment. L and G are sixteen-bit MDS relative pointers.

In the unobvious cases, stack content at the beginning of the instruction will be given as "Stack has x, y, z." in bottom to top order. 32-bit quantities are stored in the stack with the least significant word on the bottom; in memory the least significant word is stored in the lower address.

LLn **Load Local n** (n = 0..11)
Load from location L + n.

LLB **Load Local Byte**
Load from location L + alpha.

LLDn **Load Local Double n** (n = 0..8, 10)
Load from locations L + n and L + n + 1.

LLDB **Load Local Double Byte**
Load from locations L + alpha and L + alpha + 1.

SLn **Store Local n** (n = 0..10)
Store into L + n

SLB **Store Local Byte**
Store into L + alpha.

SLDn **Store Local Double n** (n = 0..6, 8)
Store into L + n + 1 and L + n.

PLn **Put Local n** (n = 0..3)
Stack has data. Store data into L + n; leave data on stack.

PLB **Put Local Byte**
Stack has data. Store data into L + alpha; leave data on stack.

PLDn **Put Local Double n** (n = 0)
Stack has 32-bit data. Store data into locations L + n and L + n + 1; leave data on stack.

PLDB **Put Local Double Byte**
Stack has 32-bit data. Store data into locations L + alpha and L + alpha + 1; leave data on stack.

LGn **Load Global n** (n = 0..2)
Load from G + n.

LGB **Load Global Byte**
Load from G + alpha.

LGDn **Load Global Double n** (n = 0, 2)
Load from locations G + n and G + n + 1.

LGDB **Load Global Double Byte**
Load from locations G + alpha and G + alpha + 1.

SGB **Store Global Byte**
Store into G + alpha.

BNDCK **Bounds Check**
Stack has value, limit. IF value ~IN [0..limit) THEN trap ELSE discard limit.

BRK **Breakpoint**
IF resuming from debugger THEN execute broken opcode ELSE trap.

Rn **Read n** (n = 0..1)
Stack has pointer. Load from pointer + n.

RB **Read Byte**
Stack has pointer. Load from pointer + alpha.

RLn **Read Long n** (n = 0)
Stack has 32-bit pointer. Load from pointer + n.

RLB **Read Long Byte**
Stack has 32-bit pointer. Load from pointer + alpha.

RDn **Read Double n** (n = 0)
Stack has pointer. Load from pointer + n and pointer + n + 1.

RDB **Read Double Byte**
Stack has pointer. Load from locations pointer + alpha and pointer + alpha + 1.

RDLn **Read Double Long n** (n = 0)
Stack has 32-bit pointer. Load from pointer + n and pointer + n + 1.

RDLB **Read Double Long Byte**
Stack has 32-bit pointer. Load from pointer + alpha and pointer + alpha + 1.

Wn **Write n** (n = 0)
Stack has data, pointer. Store data into pointer + n.

WB **Write Byte**
Stack has data, pointer. Store data into pointer + alpha.

PSB **Put Swapped Byte**
Stack has pointer, data. Store data into pointer + alpha; leave pointer on stack (but not data).

WLB **Write Long Byte**
Stack has data, 32-bit pointer. Store data into pointer + alpha.

PSLB **Put Swapped Long Byte**
Stack has 32-bit pointer, data. Store data into pointer + alpha; leave pointer on stack (but not data).

WDB **Write Double Byte**
Stack has 32-bit data, pointer. Store data into pointer + alpha and pointer + alpha + 1.

PSDn **Put Swapped Double n** (n = 0)
Stack has pointer, 32-bit data. Store data into pointer + n and pointer + n + 1;leave pointer on stack (but not data).

PSDB **Put Swapped Double Byte**
Stack has pointer, 32-bit data. Store data into pointer + alpha and pointer + alpha + 1; leave pointer on stack (but not data).

WDLB **Write Double Long Byte**
Stack has 32-bit pointer, 32-bit data. Store data into pointer + alpha and pointer + alpha + 1.

PSDLB **Put Swapped Double Long Byte**
Stack has 32-bit pointer, 32-bit data. Store data into pointer + alpha and pointer + alpha + 1; leave pointer on stack (but not data).

RLI0n **Read Local Indirect Zero n** (n = 0..3)
Pointer is in L + 0. Load from pointer + n.

RLIP **Read Local Indirect Pair**
Pointer is in L + alpha.left. Load from pointer + alpha.right.

RLILP **Read Local Indirect Long Pair**
32-bit pointer is in L + alpha.left and L + alpha.left + 1. Load from pointer + alpha.right.

RLDI0n **Read Local Double Indirect Zero n** (n = 0)
32-bit pointer is in L + 0 and L + 1. Load from pointer + n and pointer + n + 1.

RLDIP **Read Local Double Indirect Pair**
Pointer is in L + alpha.left. Load from pointer + alpha.right and L + alpha.right + 1.

RLDILP **Read Local Double Indirect Long Pair**
32-bit pointer is in L + alpha.left and L. + alpha.left + 1. Load from pointer + alpha.right and L + alpha.right + 1.

RGIP **Read Global Indirect Pair**
Pointer is in G + alpha.left. Load from pointer + alpha.right.

RGILP **Read Global Indirect Long Pair**
32-bit pointer is in G + alpha.left and G + alpha.left + 1. Load from pointer + alpha.right.

WLIP **Write Local Indirect Pair**
Pointer is in L + alpha.left. Store into pointer + alpha.right.

WLILP **Write Local Indirect Long Pair**
32-bit pointer is in L + alpha.left and L + alpha.left + 1. Store into pointer + alpha.right.

WLDILP **Write Local Double Indirect Long Pair**
Stack has 32-bit data. 32-bit pointer is in L + alpha.left and L + alpha.left + 1.Store data into pointer + alpha.right and pointer + alpha.right + 1.

RS **Read String**
Stack has pointer, index. Fetch word from pointer + (index/2); IF index MOD 2 = 0 THEN push word/256 ELSE push word MOD 256.

RLS **Read Long String**
Stack has 32-bit pointer, index. Fetch word from pointer + (index/2); IF index MOD 2 = 0 THEN push word.high ELSE push word.low.

WS **Write String**
Stack has data, pointer, index. Fetch word from pointer + (index/2); IF index MOD 2 = 0 THEN word.high ← data ELSE word.low ← data; Store word at pointer + (index/2).

WLS **Write Long String**
Stack has data, 32-bit pointer, index. Fetch word from pointer + (index/2); IF index MOD 2 = 0 THEN word.high ← data ELSE word.low ← data; Store word at pointer + (index/2).

R0F **Read Zero Field**
Stack has pointer. Fetch word from pointer + 0; push subword described by alpha.

RF **Read Field**
Stack has pointer. Fetch word from pointer + alpha; push subword described by beta.

RL0F **Read Long Zero Field**
Stack has 32-bit pointer. Fetch word from pointer + 0; push subword described by alpha.

RLF **Read Long Field**
Stack has 32-bit pointer. Fetch word from pointer + alpha; push subword described by beta.

RLFS **Read Long Field Stack**
Stack has 32-bit pointer, fieldDesc. Fetch word from pointer+fieldDesc.high; push subword described by fieldDesc.low.

RLIPF **Read Local Indirect Pair Field**
Pointer is in L+alpha.left. Fetch word from pointer+alpha.right; push subword described by beta.

RLILPF **Read Local Indirect Long Pair Field**
32-bit pointer is in L+alpha.left. Fetch word from pointer+alpha.right; push subword described by beta.

W0F **Write Zero Field**
Stack has data, pointer. Fetch word from pointer+0; put data in subword described by alpha; store word to pointer+0.

WF **Write Field**
Stack has data, pointer. Fetch word from pointer+alpha; put data in subword described by beta; store word to pointer+alpha.

PSF **Put Swapped Field**
Stack has pointer, data. Fetch word from pointer+alpha; put data in subword described by beta; store word to pointer+alpha; leave pointer on stack (but not data).

PS0F **Put Swapped Zero Field**
Stack has pointer, data. Fetch word from pointer+0; put data in subword described by alpha; store word to pointer+0; leave pointer on stack (but not data).

WS0F **Write Swapped Zero Field**
Stack has pointer, data. Fetch word from pointer+0; put data in subword described by alpha; store word to pointer+0.

WL0F **Write Long Zero Field**
Stack has data, 32-bit pointer. Fetch word from pointer+0; put data in subword described by alpha; store word to pointer+0.

WLF **Write Long Field**
Stack has data, 32-bit pointer. Fetch word from pointer+alpha; put data in subword described by beta; store word to pointer+alpha.

PSLF **Put Swapped Long Field**
Stack has 32-bit pointer, data. Fetch word from pointer+alpha; put data in subword described by beta; store word to pointer+alpha; leave pointer on stack (but not data).

WLFS **Write Long Field Stack**
Stack has data, 32-bit pointer, fieldDesc. Fetch word from pointer+fieldDesc.high; put data in subword described by fieldDesc.low; store word to pointer+alpha.

SLDB **Store Local Double Byte**
Store into L+alpha.

SGDB **Store Global Byte**
Store into G+alpha.

LLKB **Load Link Byte**
Fetch from G-2 and test link location bit.
0 => Load from G-6-(2*alpha) and G-6-(2*alpha)+1
1 => Load from C-2-(2*alpha) and C-2-(2*alpha)+1.

RKIB **Read Link Indirect Byte**
Fetch 32-bit pointer as described in LLKB; load from pointer+0.

RKDIB **Read Link Double Indirect Byte**
Fetch 32-bit pointer as described in LLKB; load from pointer+0 and pointer+1.

LKB **Link Byte**
Recover word from above the top of stack; store word-alpha in L+0.

SHIFT **Shift**
Stack has data, count. Shift data by count bits; left if count is positive; right if count is negative.

SHIFTSB **Shift Signed Byte**
Stack has data. Shift data by sign-extended alpha; left if positive; right if negative.

CATCH **Catch**
Two byte noop used by Mesa runtime system.

Jn **Jump n** (n = 2..8)
Unconditional jump. New PC is PC + n.

JB **Jump Byte**
Unconditional jump. New PC is PC + sxalpha.

JW **Jump Word**
Unconditional jump. New PC is PC + alphabeta.

JEP **Jump Equal Pair**
Stack has data. IF data = alpha.left THEN new PC is PC + alpha.right + 2.

JEB **Jump Equal Byte**
Stack has data1, data2. IF data1 = data2 THEN new PC is PC + sxalpha.

JEBB **Jump Equal Byte Byte**
Stack has data. IF data = alpha THEN new PC is PC + sxbeta.

JNEP **Jump Not Equal Pair**
Stack has data. IF data # alpha.left THEN new PC is PC + alpha.right + 2.

JNEB **Jump Not Equal Byte**
Stack has data1, data2. IF data1 # data2 THEN new PC is PC + sxalpha.

JNEBB **Jump Not Equal Byte Byte**
Stack has data. IF data # alpha THEN new PC is PC + sxbeta.

JLB **Jump Less Byte**
Stack has data1, data2. IF data1 < data2 (signed) THEN new PC is PC + sxalpha.

JGEB **Jump Greater Equal Byte**
Stack has data1, data2. IF data1 >= data2 (signed) THEN new PC is PC + sxalpha.

JGB **Jump Greater Byte**
Stack has data1, data2. IF data1 > data2 (signed) THEN new PC is PC + sxalpha.

JLEB **Jump Less Equal Byte**
Stack has data1, data2. IF data1 <= data2 (signed) THEN new PC is PC + sxalpha.

JULB **Jump Unsigned Less Byte**
Stack has data1, data2. IF data1 < data2 (unsigned) THEN new PC is PC + sxalpha.

JUGEB **Jump Unsigned Greater Equal Byte**
Stack has data1, data2. IF data1 >= data2 (unsigned) THEN new PC is PC + sxalpha.

JUGB **Jump Unsigned Greater Byte**
Stack has data1, data2. IF data1 > data2 (unsigned) THEN new PC is PC + sxalpha.

JULEB **Jump Unsigned Less Equal Byte**
Stack has data1, data2. IF data1 <= data2 (unsigned) THEN new PC is PC + sxalpha.

JZn **Jump Zero n** (n = 3..4)
Stack has data. IF data = 0 THEN new PC is PC + n.

JZB **Jump Zero Byte**
Stack has data. IF data = 0 THEN new PC is PC + sxalpha.

JNZn **Jump Not Zero n** (n = 3..4)
Stack has data. IF data # 0 THEN new PC is PC + n.

JNZB **Jump Not Zero Byte**
Stack has data. IF data # 0 THEN new PC is PC + sxalpha.

JDEB **Jump Double Equal**
Stack has 32-bit data1, 32-bit data2. IF data1 = data2 THEN new PC is PC + sxalpha.

JDNEB **Jump Double Not Equal**
Stack has 32-bit data1, 32-bit data2. IF data1 # data2 THEN new PC is PC + sxalpha.

JIB **Jump Indexed Byte**
Stack has index, limit. IF index < limit THEN { fetch disp from C + alphabeta + (index/2); new PC is PC + IF index MOD 2 = 0 THEN disp.high ELSE disp.low}.

JIW **Jump Indexed Word**
Stack has index, limit. IF index < limit THEN { fetch disp from C + alphabeta + index; new PC is PC + disp}.

REC **Recover**
Recover the value above the top of stack, i.e. increment the stack pointer without changing any stack values.

REC2 **Recover 2**
Recover the two values above the top of stack, i.e. increment the stack pointer by two without changing any stack values.

DIS **Discard**
Discard the top value on the stack, i.e. decrement the stack pointer.

DIS2 **Discard 2**
Discard the top two values on the stack, i.e. decrement the stack pointer by two.

EXCH **Exchange**
Interchange the top two values on the stack.

DEXCH **Double Exchange**
Interchange the top two 32-bit values on the stack.

DUP **Duplicate**
Duplicate the top value on the stack.

DDUP **Double Duplicate**
Duplicate the top 32-bit value on the stack.

EXDIS **Exchange Discard**
Equivalent to the sequence EXCH; DIS.

NEG **Negate**
Stack has data. Push 0-data.

INC **Increment**
Stack has data. Push data + 1.

DEC **Decrement**
Stack has data. Push data-1.

DINC **Double Increment**
Stack has 32-bit data. Push data + 1

DBL **Double**
Stack has data. Push data*2.

DDBL **Double Double**
Stack has 32-bit data. Push data*2.

TRPL **Triple**
Stack has data. Push data*3.

AND **And**
Stack has data1, data2. Push data1 AND data2.

IOR **Inclusive Or**
Stack has data1, data2. Push data1 OR data2.

ADDSB **Add Signed Byte**
Stack has data. Push data + sxalpha.

ADD **Add**
Stack has data1, data2. Push data1 + data2.

SUB **Subtract**
Stack has data1, data2. Push data1-data2.

DADD **Double Add**
Stack has 32-bit data1, 32-bit data2. Push data1 + data2.

DSUB **Double Subtract**
Stack has 32-bit data1, 32-bit data2. Push data1-data2.

ADC **Add Double to Cardinal**
Stack has 32-bit data1, 16-bit data2. Push data1 + data2.

ACD **Add Cardinal to Double**
Stack has 16-bit data1, 32-bit data2. Push data1 + data2.

AL0IB **Add Local Zero Immediate Byte**
Fetch data from L + 0; Push data + alpha.

MUL **Multiply**
Stack has data1, data2. Push the 32-bit value data1*data2 then decrement the stack pointer. The effect is to leave the 16-bit product on the stack and the high 16 bits of the product above the stack.

DCMP **Double Compare**
Stack has 32-bit data1, 32-bit data2. Compare data1 and data2 (signed) and push -1 if data1 < data2; 0 if data1 = data2; + 1 if data1 > data2.

UDCMP **Unsigned Double Compare**
Stack has 32-bit data1, 32-bit data2. Compare data1 and data2 (unsigned) and push -1 if data1 < data2; 0 if data1 = data2; + 1 if data1 > data2.

LIn **Load Immediate n** (n = 0..10)
Push n.

LIN1 **Load Immediate Negative 1**
Push -1.

LINI **Load Immediate Negative Infinity**
Push -32768.

LIB **Load Immediate Byte**
Push alpha.

LIW **Load Immediate Word**
Push alphabeta.

LINB **Load Immediate Negative Byte**
Push alpha-256. (I.e. set the high byte to all ones).

LIHB **Load Immediate High Byte**
Push alpha*256.

LID0 **Load Immediate Double 0**
Push two words of zero.

LAn **Local Address n** (n = 0..3, 6, 8)
Push 16-bit value L + n (not the contents of).

LAB **Local Address Byte**
Push 16-bit value L + alpha.

LAW **Local Address Word**
Push 16-bit value L + alphabeta

GAn **Global Address n** (n = 0..1)
Push 16-bit value G+n.

GAB **Global Address Byte**
Push 16-bit value G+alpha.

GAW **Global Address Word**
Push 16-bit value G+alphabeta.

EFCn **External Function Call n** (n = 0..12)
Fetch link n as described in LLKB and XFER to it.

EFCB **External Function Call Byte**
Fetch link alpha as described in LLKB and XFER to it.

LFC **Local Function Call**
Do the last half of XFER (frame allocation and set new PC) using alphabeta as the PC of the new procedure.

SFC **Stack Function Call**
Stack has 32-bit controlLink. XFER to controlLink.

RET **Return**
Fetch returnLink from L-1; XFER to returnLink with a source of zero.

KFCB **Kernel Function Call Byte**
Fetch link from SD[alpha] and XFER to it.

ME **Monitor Enter**
Stack has 32-bit pointer to monitor lock. IF the monitor is unlocked THEN lock it ELSE enqueue current process on monitor queue and reschedule.

MX **Monitor Exit**
Stack has 32-bit pointer to monitor lock. Unlock the monitor; IF the queue is not empty, wake up the first waiting process and reschedule.

BLT **Block Transfer**
Stack has sourcePointer, count, destPointer. Transfer count words beginning at sourcePointer to locations beginning at destPointer. This instruction is interruptable; if interrupted it leaves updated values on the stack and the isntruction is restarted after the interrupt with this new data.

BLTL **Block Transfer Long**
Stack has 32-bit sourcePointer, 16-bit count, 32-bit destPointer. See BLT for remainder of description.

BLTC **Block Transfer Code**
Stack has 16-bit offset, count, destPointer. sourcePointer is C+offset; see BLT.

BLTCL **Block Transfer Code Long**
Stack has 16-bit offset, count, 32-bit destPointer. sourcePointer is C+offset; see **BLT**.

LP **Lengthen Pointer**
Stack has pointer. IF pointer=0 THEN push 0 ELSE push MDS.

ESC **Escape**
Use alpha for a secondary opcode dispatch (see below). This instruction is always exactly two bytes long. Unimplemented instructions trap through the ESCTrapTable.

ESCL **Escape Long**
Like ESC except this instruction is always exactly three bytes long.

RESRVD **Reserved**
Opcodes 0 and 255 are reserved for implementation dependent uses, e.g. one implementation implemented interrupts by forcing the next opcode dispatch to go to opcode 0 and sorting out the state there. These codes are never generated by the compiler and should not be assigned any programmer accessible function.

Escape opcode alpha bytes:

There are two Escape opcodes, ESC and ESCL. These decode their alpha bytes from a single eight-bit address space. The sole purpose of having two different instructions is so that all Mesa opcodes have one fixed length. This permits (relatively) simple hardware to pre-fetch and pre-decode instructions based only on their first byte. In the list below, codes beginning with "a" are used exclusively with ESC to form a two-byte opcode. Codes beginning with "b" are used exclusively with ESCL (and the following beta byte) to form a three-byte opcode.

aMW **Monitor Wait**
Stack has 32-bit monitorPointer, 32-bit conditionPointer, 16-bit timeout. Unlock monitor (and wakeup waiting process if any); enqueue current process on condition with timeout value; reschedule. Exceptions: IF condition has wakeup waiting OR process has been aborted THEN current process continues to run.

aMR **Monitor Reenter**
Stack has 32-bit monitorPointer, 32-bit conditionPointer. IF monitor locked THEN enqueue on monitor queue and reschedule ELSE {test for aborting and trap if appropriate; lock monitor and proceed}.

aNC **Notify Condition**
Stack has 32-bit conditionPointer. Wakeup the first process on the condition queue and reschedule if awakened.

aBC **Broadcast Condition**
Wakeup all process on the condition queue and reschedule if any awakened.

aREQ **Requeue**
Stack has 32-bit queuePointer1, 32-bit queuePointer2, 16-bit process. Dequeue process from queue1 and enqueue it on queue2. IF either queue was the ready list THEN reschedule.

aSM **Set Map**
Stack has 32-bit virtualPage, 32-bit realPage, flags. Set up the indicated virtual to real mapping.

aSMF **Set Map Flags**
Stack has 32-bit virtualPage, flags. Set the flags in the indicated map entry.

aGMF **Get Map Flags**
Stack has 32-bit virtualPage. Push the flags of the indicated virtual page.

aAF **Allocate Frame**
Stack has index. Allocate a frame from the indicated list (as in a procedure call) and push the 16-bit address of the frame.

aFF **Free Frame**
Stack has pointer. Fetch index from pointer-3 and link the frame onto the indicated list.

aPI **Port In**
Recover source and portAddress from above stack. Store 32-bit zero at portAddress + 0; IF source # 0 THEN extend it with a high order zero and store at portAddress + 2 and portAddress + 3.

aPO **Port Out**
Stack has portAddress. Store L at portAddress + 0; fetch link from portAddress + 2 and portAdress + 3 and XFER to it with a source of portAddress.

aPOR **Port Out Responding**
Identical to Port Out. The distinction between the two is used by a trap handler to decide how to recover from port faults.

aSPP **Set Process Priority**

Stack has priority. Dequeue current process from ready queue; change its priority; enqueue process on ready queue and reschedule.

aDI **Disable Interrupts**

Increment the WakeupDisableCounter thus disabling any interrupt processing and process timeouts. Trap if the counter overflows.

aEI **Enable Interrupts**

Decrement the WakeupDisableCounter; if the new value is zero, interrupts are now enabled. Trap if the counter underflows.

aXOR **Exclusive Or**

Stack has data1, data2. Push data1 XOR data2.

aDAND **Double And**

Stack has 32-bit data1, 32-bit data2. Push data1 AND data2.

aDIOR **Double Inclusive Or**

Stack has 32-bit data1, 32-bit data2. Push data1 OR data2.

aDXOR **Double Exclusive Or**

Stack has 32-bit data1, 32-bit data2. Push data1 XOR data2.

aROTATE **Rotate**

Stack has data, count. Rotate data by count MOD 16 bits; left if count is positive; right if count is negative.

aDSHIFT **Double Shift**

Stack has 32-bit data, count. Shift data by count bits; left if count is positive; right if count is negative.

aLINT **Lengthen Integer**

Stack has data. Sign extend data from 16 to 32 bits.

aJS **Jump Stack**

Stack has newpc. Unconditional jump to newpc (relative to C as all pc values are).

aRCFS **Read Code Field Stack**

Stack has offset, fieldDesc. Fetch word from C+offset+fieldDesc.high; push subword described by fieldDesc.low.

bRC **Read Code**

Stack has offset. Load from C+offset.

aUDIV **Unsigned Divide**

Stack has data1, data2. Push quotient and remainder from data1/data2; decrement the stack pointer so as to leave the remainder above the stack.

aLUDIV **Long Unsigned Divide** (should be Double, not Long)

Stack has 32-bit data1, 16- bit data2. Push quotient and remainder from data1/data2; decrement the stack pointer so as to leave the remainder above the stack. Trap if the quotient will not fit in 16 bits.

bROB **Read Overhead Byte**

Stack has pointer. Load from pointer-beta. All access to local and global frame overhead is done through this instruction (and aWOB) so that the processor could cache this data.

bWOB **Write Overhead Byte**

Stack has data, pointer. Store data at pointer-beta. All access to local and global frame overhead is done through this instruction (and aROB) so that the processor could cache this data.

bDSK **Dump Stack**

Dump the evaluation stack and stack pointer starting at L+beta. No more than two values above the top of stack need be stored.

bXE **Xfer and Enable**

XFER using destination stored at L+beta and L+beta+1 and source stored at

L+beta+2 and L+beta+3; at the end of the XFER, decrement the WakeupDisableCounter.

bXF **Xfer and Free**
XFER using destination stored at L+beta and L+beta+1 and source stored at L+beta+2 and L+beta+3; at the end of the XFER, free the current local frame.

bLSK **Load Stack**
Load the evaluation stack and stack pointer from locations starting at L+beta. No more than two values above the top of stack need be loaded.

aBNDCKL **Bounds Check Long**
Stack has 32-bit value, 32-bit limit. IF value ~IN [0..limit) THEN trap ELSE discard limit.

aNILCK **NIL Check**
Stack has pointer. IF pointer = 0 THEN trap ELSE leave pointer on stack.

aNILCKL **NIL Check Long**
Stack has 32-bit pointer. IF pointer = 0 THEN trap ELSE leave pointer on stack.

aBLTLR **Block Transfer Long Reverse**
Stack has 32-bit sourcePointer, 16-bit count, 32-bit destPointer. Working backwards through memory, transfer count words beginning at sourcePointer+count-1 to locations beginning at destPointer+count-1. This instruction is interruptable; if interrupted it leaves updated values on the stack and the isntruction is restarted after the interrupt with this new data.

aBLEL **Block Equal Long**
Stack has 32-bit pointer1, 16-bit count, 32-bit pointer2. Compare count words beginning at pointer1 and pointer2; push 1 if all words are equal; 0 otherwise. This instruction is interruptable like BLT.

aBLECL **Block Equal Code Long**
Stack has 16-bit offset, 16-bit count, 32-bit pointer2. Like BLEL with pointer1 = C+offset.

aCKSUM **Checksum**
Computes the XNS protocol checksum of a block of words. Interruptable.

aBITBLT **Bit Block Transfer**
Stack has pointer to parameters record. Performs many operations on rectangular areas in memory. Used mostly in conjunction with bitmap display data. Known in some quarters as RasterOp.

aTXTBLT **Test Block Transfer**
Stack has pointer to parameters record. Specialized code for measuring and displaying textual data in bitmapped display memory.

aBYTBLT **Byte Block Transfer**
Stack has 32-bit destPointer, destIndex, count, 32-bit sourcepointer, sourceIndex. Like BLT except operates on byte sequences with potentially different alignments.

aBYTBLTR **Byte Block Transfer Reverse**
Stack has 32-bit destPointer, destIndex, count, 32-bit sourcepointer, sourceIndex. Like BYTBLT except operates on byte sequences from high to low addresses.

aVERSION **Version**
Returns processor dependent data about the type of processor and the version of microcode.

aDMUL **Double Multiply**
Stack has 32-bit data1, 32-bit data2. Push 32-bit product data1*data2; overflow is ignored.

aSDIV **Signed Divide**
Stack has 32-bit data1, 32-bit data2. Push quotient and remainder from

datal/data2; decrement the stack pointer so as to leave the remainder above the stack.

aSDDIV	**Signed Double Divide**	Stack has 32-bit datal, 32-bit data2. Push quotient and remainder from datal/data2; decrement the stack pointer by 2 so as to leave the remainder above the stack.
aUDDIV	**Unsignedigned Double Divide**	Stack has 32-bit datal, 32-bit data2. Push quotient and remainder from datal/data2; decrement the stack pointer by 2 so as to leave the remainder above the stack.

-- Floating Point (IEEE standard format, 32-bit only)

aFADD	**Floating Add**	
aFSUB	**Floating Subtract**	
aFMUL	**Floating Multiply**	
aFDIV	**Floating Divide**	
aFCOMP	**Floating Compare**	Returns -1, 0, +1 like DCMP.
aFIX	**Fix**	Returns 32-bit integer
aFLOAT	**Float**	Takes 32-bit integer.
aFIXI	**Fix to Integer**	Returns 16-bit integer.
aFIXC	**Fix to Cardinal**	Returns 16-bit unsigned number.
aFSTICKY	**Floating Sticky Flags**	Sets floating point sticky flags, returns old value.
aFREM	**Floating Remainder**	Returns the fractional part of a quotient.
aROUND	**Round**	Rounds operand to 32-bit integer.
aROUNDI	**Round to Integer**	Rounds to 16-bit integer.
aROUNDC	**Round toCardinal**	Rounds to 16-bit unsigned number.
aFSQRT	**Floating Square Root**	
aFSC	**Floating Scale**	

-- Cedar collector and allocator

aRECLAIMREF
aALTERCOUNT
aRESETSTKBITS
aGCSETUP
a144
aENUMERATERECLAIMABLE
a146
aCREATEREF
a150
aREFTYPE
aCANONICALREFTYPE
aALLOCQUANTIZED
aALLOCHEAP

aFREEOBJECT
aFREEQUANTIZED
aFREEPREFIXED

-- Read / Write Registers

aWRPSB **Write Register PSB**
Set the id of the current process.

aWRMDS **Write Register MDS**
Set the current value of MDS.

aWRWP **Write Register Wakeups Pending**
Bit mask of interrupts not yet processed.

aWRWDC **Write Register Wakeup Disable Counter**
Controls the taking of interrupts.

aWRPTC **Write Register Process Tick Counter**
The counter which controls process timeouts.

aWRIT **Write Register Interval Timer**
The high resolution (1-100 microseconds).

aWRXTS **Write Register Xfer Trap Status**
Governs the taking of a trap on evry XFER.

aWRMP **Write Register Maintenance Panel**
Four digit decimal LED display.

aRRPSB
aRRMDS
aRRWP
aRRWDC
aRRPTC
aRRIT
aRRXTS

-- Processor Dependent Instructions

aINPUT **I/O Input**
Stack has 16-bit I/O register number. Returns 16-bit data.

aOUTPUT **I/O Output**
Stack has 16-bit I/O register number, 16-bit data.

aLOADRAMJ **Load Control Store and Jump**
Overwrite your own control store and execute the new code.

-- Dandelion Instructions

aBANDBLT **Band Block Transfer**
Specialized code for printers.

Mesa Code Index

OpCode Mnemonics

OpCode Names

Primary Index

References

[1] *Mesa Language Manual.* Version 3.0. [November 1984].

[2] IEEE Standard for Binary Floating-Point Arithmetic. Draft 10.0. [To be published Summer, 1985].