The
Connection Machine
System

Paris Reference Manual

Version 6.0
February 1991

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, February 1989
Revised, February 1991

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142–1264
(617) 234–1000/876–1111

Contents

List of Figures

Customer Support

Thinking Machines Customer Support encourages customers to report errors in Connection Machine operation and to suggest improvements in our products.

When reporting an error, please provide as much information as possible to help us identify and correct the problem. A code example that failed to execute, a session transcript, the record of a back-trace, or other such information can greatly reduce the time it takes Thinking Machines to respond to the report.

To contact Thinking Machines Customer Support:

U.S. Mail:	Thinking Machines Corporation Customer Support 245 First Street Cambridge, Massachusetts 02142–1264
Internet Electronic Mail:	customer–support@think.com
Usenet Electronic Mail:	ames!think!customer-support
Telephone:	(617) 234–4000 (617) 876–1111

For Symbolics Users Only

The Symbolics Lisp machine, when connected to the Internet network, provides a special mail facility for automatic reporting of Connection Machine system errors. When such an error occurs, simply press Ctrl-M to create a report. In the mail window that appears, the `To:` field should be addressed as follows:

`To:` customer–support@think.com

Please supplement the automatic report with any further pertinent information.

Part I
Paris Concepts

Version 6.0, February 1991

Chapter 1

Introduction

Paris is a low-level instruction set for programming the Connection Machine computer system. It is the lowest-level protocol by which the actions of Connection Machine processors are directed by the front-end computer. Paris is sometimes referred to as a "macroinstruction set" for the Connection Machine system because it is comparable in power to the (macro)instruction sets of typical sequential processors such as the VAX, and to distinguish it from the "microinstruction set" (microcode) that is executed by the Connection Machine system sequencer and the "nanoinstruction set" that is directly executed by the individual hardware Connection Machine processors.

Paris is intended primarily as a base upon which to build higher-level languages for the Connection Machine system. It provides a large number of operations similar to the machine-level instruction set of an ordinary computer. Paris supports primitive operations on signed and unsigned integers and floating-point numbers, as well as message-passing operations and facilities for transferring data between the Connection Machine processors and the front-end computer.

The Paris user interface consists of a set of macros, functions, and variables to be called from user code. The macros and functions direct the actions of the Connection Machine system by sending macroinstructions to the Connection Machine sequencer, and the variables allow the user program to find out information about the Connection Machine system such as the number of processors available.

Several different versions of the user interface are provided: one for the Lisp programming language, one for C, and one for Fortran. These interfaces are functionally identical; they differ only in conforming to the syntax and data types of one language or the other.

Chapter 2

Virtual Machine Architecture

An important property of the Connection Machine architecture is *scalability*. At present, a single Connection Machine system can have 16,384 or 32,768 or 65,536 physical (hardware) processors, of which any single user can use a portion containing 8,192 or 16,384 or 32,768 or 65,536 processors. (See figure 2.1 for an illustration of 65,536 processors.) In most cases the same software can be executed unchanged on Connection Machine systems (or portions) with different numbers of physical processors; the number of processors affects only the size of the problem that can be handled.

Paris enhances this scalability by presenting to the user an abstract version of the Connection Machine hardware. The most important feature is the *virtual processor* facility, whereby each physical processor is used to simulate some number of virtual processors. A program can be written assuming *any* appropriate number of processors (but not fewer than the number of physical processors); these virtual processors are then mapped onto physical processors. In this way a program can be executed unchanged on Connection Machine systems with different numbers of physical processors, even if it requires a certain minimum number of processors, with an essentially linear trade-off between number of physical processors and execution time. (There is a memory trade-off as well: the memory of a physical processor is divided among the virtual processors it supports.)

For the remainder of this chapter, when we refer to "the Connection Machine" or "the machine" we mean that portion of a Connection Machine system to which the user is attached. For example, if a user is attached to a 16,384 processor portion of a 65,536 processor Connection Machine, the expression "the machine" refers only to the user's 16,384 processors.

The Connection Machine hardware supports two mechanisms for interprocessor communication. The more general mechanism is the *router*, which allows data to be sent from any processor directly to any other processor; indeed, many processors can send data to many other processors simultaneously. The less general mechanism is redundant, but optimizes an important case for speed. It organizes the processors as an n-dimensional grid and allows every processor to send data to its immediate neighbors in the grid. This mechanism is called the *NEWS grid*, from the initials of the four directions in a two-dimensional grid: North, East, West, and South. Using these hardware mechanisms, Paris provides identical virtual mechanisms within the virtual processor framework.

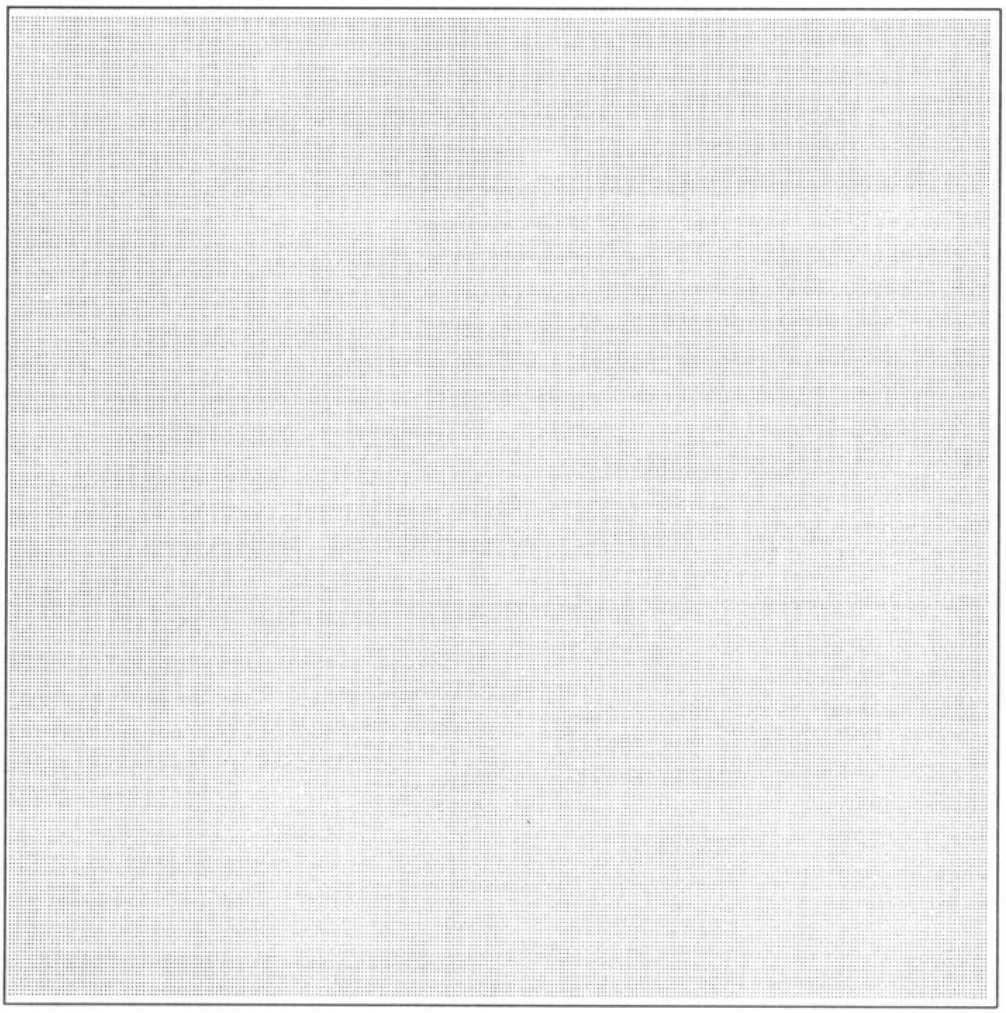


Figure 2.1: 65,536 processors

2.1 Virtual Processors and Virtual Processor Sets

The data parallel programming method associates one processor with each element of a data set. In the virtual processor abstraction provided by Paris, we associate one virtual processor, or VP, with each element of a data set. The set of all virtual processors associated with a data set is called a *virtual processor set*, or *VP set*. For example, consider an image-processing problem that deals with an image of 65,536 pixels, shaped in a 512×128 rectangle. Each pixel is an element of the data set that makes up the image. Thus we would write a program using one VP set of size 65,536: one VP for each pixel.

Because a single problem may be composed of more than one data set, Paris allows for the simultaneous existence of more than one VP set. For example, a text retrieval program might wish to deal with articles at some times, and with words in the articles at other times. This problem is most conveniently modeled with two VP sets, the first corresponding to the data set of all articles (one VP per article) and the second corresponding to the data set of all words (one VP per word).

VP sets are created and deleted through function calls to Paris. The size of a VP set (the number of virtual processors in the VP set) is fixed at the time of the VP set's creation.

Although multiple VP sets may co-exist, only one VP set may be active at any time. This VP set is known as the *current VP set*. All VP sets other than the current VP set are latent; that is, they can not execute any instructions. We say that Paris operates within the current VP set. Paris provides a function `CM:set-vp-set` for setting the current VP set.

2.2 Mapping VP Sets to the Physical Machine

When a Paris program is run, the virtual processors in the user's program are mapped onto the machine's physical processors. The size of the VP set(s) and the size of the physical machine determine how many virtual processors are assigned to each physical processor. In effect, each Connection Machine processor and its memory are shared among the virtual processors they support.

These concepts are further elaborated in the following sections. The time-slicing of the Connection Machine processors is covered in the section "VP Ratios"; the sharing of physical memory among virtual processors is covered in the section "Fields." Communication and related concepts follow.

2.3 VP Ratios

Let p denote the number of Connection Machine physical processors, and let $|X|$ denote the number of virtual processors in a VP set X.

For each VP set X, each physical processor is assigned the task of simulating $|X|/p$ virtual processors. This number $|X|/p$ is called the *virtual processor ratio*, or *VP ratio*, of VP set X. We denote the VP ratio of VP set X as $vpr(X)$. The virtual processor ratio must always be a power of two.

What exactly does this mean? When the machine is operating within VP set X, each instruction in the user's program is executed $vpr(X)$ times by each physical processor, that is, once for every virtual processor. This is completely transparent to the user. A change of

VP set changes the VP ratio to be that of the newly current VP set; if the program changes from VP set X to VP set Y, each instruction after that will be executed $vpr(Y)$ times.

This method of assigning virtual processors to physical processors "spreads out" a VP set as much as possible; the VP ratio for each VP set is as low as possible. The burden of handling a VP set is shared by the entire physical machine.

As an example, suppose we have two VP sets A and B, where $|A| = 64\text{K}$ and $|B| = 256\text{K}$. Suppose we run our program on a Connection Machine system with 64K physical processors ($p = 64\text{K}$). Then $vpr(a) = 64\text{K}/64\text{K} = 1$, and $vpr(b) = 256\text{K}/64\text{K} = 4$. When executing within VP set A, each instruction is executed once by each physical processor. When executing within VP set B, each instruction is executed four times by each physical processor.

If the same program were to be run on a Connection Machine system with only 16K physical processors ($p = 16\text{K}$), then we would have $vpr(a) = 64\text{K}/16\text{K} = 4$, and $vpr(b) = 256\text{K}/16\text{K} = 16$. When executing within VP set A, each instruction would be executed four times by each physical processor. When executing within VP set B, each instruction would be executed 16 times by each physical processor.

This description of "execute once for each virtual processor" applies most accurately to operations such as arithmetic that can take place within each virtual processor independently of other virtual processors. Operations that perform communication are more complicated, but the idea is the same: each physical processor performs all necessary execution steps on behalf of each virtual processor that is to participate in the operation.

As far as the user is concerned, physical processors are hardly visible. Paris is designed to allow the programmer to think entirely in terms of the virtual processor as the basic unit of computational power.

2.4 Fields

At the time of its creation, a VP set has no associated memory (except for its flags). This is the same as saying that no VP in the VP set has any memory, because the memories of all virtual processors in a VP set are always of the same size and layout. Paris provides functions to allocate and deallocate memory to a VP set.

Memory is handled in units called *fields*. Conceptually, a field is simply some number of consecutive bits. A field can be of any size greater than zero bits. When a field is allocated, it has an initial size specified by the user. When we speak of allocating a field to a VP set, we mean allocating a field to each VP in the VP set.

A field is referenced through a *field* ID. Paris returns a unique field ID for each new field that is allocated, and all Paris calls that require a reference to a field take a field ID as a parameter.

How does this abstraction of fields get mapped into physical Connection Machine memory? Again, the concept of VP ratios is important. Just as a Connection Machine physical processor takes responsibility for $vpr(X)$ virtual processors for each VP set X in the user's program, those same physical processors (more precisely, their memories) take responsibility for the fields of those same virtual processors. A single physical memory contains $vpr(X)$ copies of every field in VP set X, $vpr(Y)$ copies of every field in VP set Y, and so on for every VP set in the user's program.

There are two types of fields: heap fields and stack fields. The distinction between them has to do with the storage management strategy employed in the physical memory supporting the virtual processors. Heap fields are the more flexible of the two, but they also have the higher overhead. Heap fields may be allocated and deallocated in any order. Allocation of heap fields to VP set X may be freely intermixed with allocations to VP set Y, and so on. Deallocations need pay no attention to the VP set to which a field belongs, nor to the order in which other allocations and deallocations were done.

Stack fields may be allocated in any order, without regard to VP set. However, stack fields must be deallocated in the reverse order in which they were allocated. This rule applies globally to all fields in all VP sets. Thus, if a program allocates a field f_1 in VP set A, and then allocates a field f_2 in VP set B, and then allocates a field f_3 in VP set A, they must be deallocated in the order f_3, f_2, f_1.

2.5 Processor Addresses

Paris supports two different sorts of addresses for virtual processors: the *send address*, which is used for general purpose communication among virtual processors, and the *NEWS address*, which describes a VP's position in the n-dimensional grid used to optimize nearest-neighbor communication.

A virtual processor has one send address and one NEWS address at all times. Send addresses and NEWS addresses are specific to a VP set; that is, every VP in a VP set has a unique send address and a unique NEWS address, but it is possible for a VP in another VP set to have the same send address or NEWS address. Since Paris always operates within a single VP set, there is normally no ambiguity as to which VP is meant by a given address. For communication across VP sets, Paris has other means of uniquely identifying the intended destination VP.

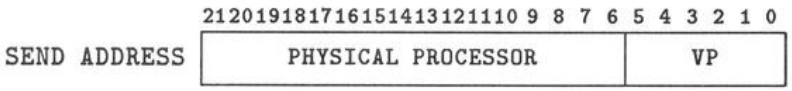
2.6 Send Addresses

Send addresses are used as arguments to Paris communication operations to identify virtual processors that are to supply or receieve data. The Paris operation CM:my-send-address allows every VP in a VP set to find out its own send address.

The send address for a VP is composed of two parts, the physical part and the virtual part. The physical part indicates the location in the CM of the physical processor supporting that VP. The virtual part indicates which VP in that VP set on that physical processor is being addressed. The virtual part is in the less significant bits of the send address.

The size (in bits) of a send address for a VP set depends on two things. The physical size of the machine determines the size of the physical part of the send address. The VP ratio for the VP set determines the size of the virtual part.

For example, in a 64K = 2^{16} Connection Machine, the send addresses for VP set Q with $vpr(Q) = 64 = 2^6$ require 22 bits: 16 bits for the physical part, and 6 bits for the virtual part. In this example, send addresses range from 0 to $2^{22} - 1$.



In this release of Paris, VP ratios must be a power of two. This results in a contiguous address space for send addresses (that is, there are no "holes"). However, this feature is likely to change in the future (thereby allowing a VP ratio to be any integer, not just a power of two). We recommend that no Paris program be written so as to require send addresses to occupy a contiguous range. In particular, we discourage arithmetic on send addresses. Paris provides functions for manipulating send addresses in a "safe" manner. Arithmetic is better done on NEWS addresses; if a total order on all processors is required, please note that a NEWS grid may be one-dimensional.

2.7 NEWS Addresses

A NEWS address is an n-tuple of coordinates $x_0, x_1, \ldots, x_{N-1}$, which specifies a VP's position in an n-dimensional Cartesian-grid geometry. The number of bits required to specify each coordinate depends on the size of that dimension in the geometry. NEWS addresses are treated in more detail below when we discuss geometries.

The Paris operation CM:my-news-coordinate-1L allows every VP in a VP set to find out its own NEWS coordinate along a given axis. Paris also provides functions for producing a send address from a NEWS address, and vice versa. There are a number of variations on these functions to handle only specific dimensions. All addresses are interpreted within the current VP set.

2.8 Communication across VP Sets

Communication across VP sets takes place via the Paris send and get operations and their variants. These operations each accept only a send address as the indicator of the remote VP; NEWS addresses are not allowed. The send address must be of the proper size for the remote VP set; that is, it must have as many bits as are necessary to specify a send address in that VP set, which may be different from the number of bits needed to specify a send address in the current VP set.

We have noted that send addresses are not unique across all VP sets in a program, but that communication across VP sets is unambiguous anyway. This is because every call to a Paris send or get operation also takes a field in a remote VP set as an argument. A field is always associated with exactly one VP set, and this fact allows Paris to determine the remote VP intended as a send destination or a get source.

2.9 Geometries

A *geometry* is an abstract description of an n-dimensional grid of elements. It specifies n, the number of dimensions (also known as the *rank* of the geometry), and it specifies the length of each dimension. There are other aspects of a geometry that may be specified by the Paris user, but we first elaborate on the more basic issues.

The rank of a geometry is an integer between 1 and 31, inclusive. This is the same as saying that a geometry can describe anything from a 1-dimensional grid to a 31-dimensional grid. We number the dimensions of a grid from 0 to the rank minus 1, so we say that a 1-dimensional grid has only dimension 0, a two-dimensional grid has dimensions 0 and 1, etc.

The size of a dimension must be a power of two. The product of the sizes of all dimensions of a geometry specifies the total number of elements in the geometry. For example, a three-dimensional geometry of size $16 \times 512 \times 2$ contains 16,384 elements in all.

Paris provides functions for defining geometries. See section 5.2. A geometry is defined in the abstract, but it has no use until it is associated with a VP set, via another Paris function. Associating a geometry with a VP set defines a "shape," or organization, for the virtual processors of the VP set.

At the time of a VP set's creation, it is associated with some geometry. The geometry specifies the size of the VP set and its conceptual organization in n-space. A VP set is always associated with exactly one geometry, but it may be associated with different geometries over time. Paris provides a function for associating a geometry with a VP set (and implicitly dis-associating the previous one). See section 5.1. In this way, the user can "reshape" a VP set. The only restriction is that all geometries associated with a VP set be of the same total size, since a VP set is not allowed to change size. For example, a VP set originally associated with a $16 \times 512 \times 2$ geometry can later be associated with a 64×256 geometry, since the total number of virtual processors described by both of these geometries is the same (16,384 in this example).

The NEWS address of a virtual processor depends completely on the geometry currently associated with its VP set. Thus, while the send addresses of virtual processors remain constant for the life of a VP set, the NEWS addresses of those same virtual processors can vary as the geometry is changed. When a VP set has a three-dimensional geometry, NEWS addresses for that VP set have three coordinates: x_0, x_1, x_2. When that VP set changes to a two-dimensional geometry, NEWS addresses for that VP set have two coordinates: x_0, x_1.

Given a VP set and given a geometry as we have described it so far (a rank and the size of each dimension), there are many ways for Paris to assign virtual processors to physical processors. However, not all mappings will provide equally efficient communication among the virtual processors of a VP set. Paris allows the user to specify more information than just rank and size of dimensions when creating a geometry. These additional pieces of geometry information we call *ordering* and *weight*, and we discuss them in more detail below.

It should be said, however, that the specification of these properties of a geometry affects only the efficiency of inter-VP communication, and therefore the performance of the program. Choosing suboptimal values will never cause an otherwise correct program to execute in an erroneous manner. Also, for some problems (those involving little or no communication among virtual processors of a VP set) it does not matter how the user specifies these properties. Paris provides a function for creating geometries that does not require specification of ordering or weight information.

Each dimension of a geometry is given an *ordering*. The ordering of a dimension specifies how NEWS coordinates for that dimension are mapped onto physical processors. There are currently two possible orderings: NEWS ordering and send-address ordering. (There may be

more in the future.) Different dimensions of a geometry may be given different orderings.

The NEWS ordering specifies the embedding of the grid into the physical (hardware) n-dimensional grid such that processors with adjacent NEWS coordinates are in fact neighbors within the physical grid. The send-address ordering specifies that if processor A has a smaller NEWS coordinate than processor B (in the specified dimension), then A also has a smaller send address than B. Paris functions that provide nearest-neighbor communication (the CM:get-from-news family of functions, for example) perform best with NEWS ordering. Send ordering is useful for applications such as Fast Fourier Transform; under the send ordering, processors that are nearest neighbors within the physical grid have grid coordinates that differ by various powers of two.

What is the weight of a dimension for? Whenever the VP ratio of a VP set is greater than 1, some number of virtual processors are co-resident on a physical processor. If these virtual processors happen to all be in the same dimension of their geometry, communication among them will be even faster than if they were neighbors in the physical NEWS grid. Communication among virtual processors assigned to the 16 physical processors on a Connection Machine chip is also faster than communication between chips, even if the processors concerned are neighbors in the physical NEWS grid.

Paris can lay out virtual processors on physical processors in such a way as to take advantage of intra-processor and intra-chip communication, provided the Paris user knows which dimension(s) of the geometry will sustain the heaviest communication. (By communication, we mean also operations such as scan and spread). Thus, Paris provides an operation for creating geometries with an indication (the *weight*) of which dimension will have the heaviest communication, which will be second heaviest, etc. Paris then maps the virtual processors onto the physical processors in such a way as to favor the dimensions with the heaviest communication.

2.10 Flags

Each Paris virtual processor has an assortment of one-bit flags. These flags are represented as fields that are specially associated with VP sets. These fields are automatically created when the VP set is created by CM:allocate-vp-set.

Many Paris operations store into these flags rather than, or in addition to, storing results into explicitly supplied argument fields. For example, the CM:s-add-2-1L operation adds one signed integer to another, but also stores information into the carry flag and the overflow flag.

The entire set of flags for each virtual processor is as follows.

- The *context-flag* indicates which virtual processors are active within the current VP set. Nearly all Paris operations are *conditional*; the operation is effectively carried out only in those processors whose *context-flag* is 1, and processors whose *context-flag* is 0 are unaffected. Some operations are always unconditional.

- The *test-flag* holds the result of numeric comparisons and other tests, or indicates which operations failed because of bad operands.

- The *carry-flag* holds the carry in and carry out for some integer arithmetic operations. A few operations use the *carry-flag* as an implicit input.

- The *overflow-flag* indicates which operations produced results that the destination field was too small to contain. Many Paris operations can affect the *overflow-flag*.

Chapter 3

Data Formats

A data item always consists of a string of bits having consecutive addresses. Such a bit string is called a *field*. The term *field* is also used to refer to a collection of fields, one for each virtual processor.

Many Paris operations may be regarded as interpreting bit fields as being of particular data types or formats. Currently Paris provides operations that regard the contents of bit fields as structured according to the following data types:

- signed integers, represented in two's-complement format
- unsigned integers, represented in straight binary format
- floating-point numbers, represented in a format close to that specified by IEEE standard 754 for floating-point arithmetic
- complex floating-point numbers, represented as two floating-point numbers, the real part and the imaginary part
- send-addresses, which are unsigned integers that label virtual processors for communication purposes
- NEWS coordinates, which are unsigned integers, tuples of which label virtual processors within a Cartesian grid for communication purposes

The Connection Machine system allows unusual flexibility in that the hardware does not enforce any particular length or alignment requirements. Paris supports integers and floating-point numbers of almost any size. (However, certain sizes of floating-point number allow particularly efficient execution by the hardware floating-point accelerator, and certain sizes of integer allow certain other operations to be particularly efficient.)

Most Paris operations operate on fields within a virtual processor, delivering results to other fields within that virtual processor. Frequently we speak of one data item, but really mean to speak of many instances of that data item, one for each selected processor, to be considered or operated on in parallel. For example, when we say that an operation sets a flag when a field has such-and-so value, we mean that *within each processor* a separate decision is made: whether to set that processor's flag based on the value of the field within that processor.

3.1 Bit Fields

A bit field is specified by a bit address a and a positive length n; the field consists of the bits with addresses a through $a + n - 1$, inclusive. Therefore the address of a field is the same as that of the lowest-addressed bit.

3.2 Signed Integers

A signed integer is specified in the same way as a simple bit field, by a bit address a and a positive length n. The signed integer is represented in two's-complement form, and so a signed integer of length n can take on values in the range $-(2^{(n-1)})$ through $2^{(n-1)} - 1$, inclusive. The least significant bit has address a, and the most significant (sign) bit has address $a + n - 1$.

All arithmetic on signed integers is performed in a strict wraparound mode. As a rule, if the result of an operation overflows the destination field, the *overflow-flag* is set, and the destination receives as many low-order bits of the true result as will fit. For example, using 4-bit signed arithmetic, multiplying 4 by -7 will produce the 4-bit result 4 (and also set the *overflow-flag*), because the two's-complement representation of -28 is ...111111100100, of which the four low-order bits are 0100, or 4. Signed-integer operations that do not overflow leave the *overflow-flag* unchanged.

In order to simplify the Connection Machine microcode, this arbitrary restriction is imposed: the length n may not be zero or one. In addition, certain operations on signed integers cannot handle operands whose length is greater than the value of the variable CM:*maximum-integer-length*; see section 3.7.

3.3 Unsigned Integers

An unsigned integer is specified in the same way as a simple bit field: by a bit address a and a positive length n. The unsigned integer is represented in stright binary form, and so an unsigned integer of length n can take on values in the range 0 through $2^n - 1$, inclusive. The least significant bit has address a, and the most significant bit has address $a + n - 1$.

All arithmetic on unsigned integers is performed in a strict wraparound mode, modulo 2^n. As a rule, if the result of an operation overflows the destination field, the *overflow-flag* is set, and the destination receives as many low-order bits of the true result as will fit. For example, using 4-bit unsigned arithmetic, multiplying 4 by 7 will produce the 4-bit result 12 (and also set the *overflow-flag*), because the two's-complement representation of 28 is ...00000011100, of which the four low-order bits are 1100, or 12. Unsigned-integer operations that do not overflow clear the *overflow-flag*.

Unsigned integers, unlike signed integers, may be of length zero or one as well as of larger sizes. (Note that an unsigned integer of length zero is considered to have the value 0.) However, certain operations on unsigned integers cannot handle operands whose length is greater than the value of the variable CM:*maximum-integer-length*; see section 3.7.

3.4 Floating-Point Numbers

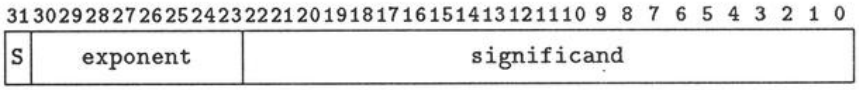
A floating-point data item is specified by three parameters: a bit address a, a significand length s, and an exponent length e. The total number of bits in the representation is $s+e+1$, and the data item occupies the bits with addresses a through $a+s+e$, inclusive.

The significand occupies bits a through $a+s-1$, with the least significant bit at address a. A hidden-bit representation is used, and so the significand is normally interpreted as having a 1-bit as its most significant bit implicitly just above the bit at address $a+s-1$. If the exponent field is all zero-bits, however, then the hidden bit is taken to be 0.

The exponent occupies bits $a+s$ through $a+s+e-1$, with the least significant bit at address $a+s$. An excess-$(2^{e-1}-1)$ representation is used.

The sign bit occupies bit $a+s+e$, and is 1 for a negative number and 0 for a positive number. Overall, a sign-magnitude representation is used, so inverting the sign of a floating-point number merely involves flipping the sign bit. Note that there is both a plus zero and a minus zero.

When $s = 23$ and $e = 8$, this is equivalent to the IEEE standard 754 single-precision format, which looks like this:



When $s = 52$ and $e = 11$, the Paris floating-point format is equivalent to IEEE standard 754 double-precision format. The IEEE standard single-extended and double-extended formats can also be accommodated by suitable choices of s and e.

While the Paris floating-point *format* is equivalent to the IEEE standard format, it must be emphasized that the Paris implementation does not support equivalent *operations* at this time.[1] "Soft" underflow (using denormalized numbers for the result) is not supported. Rounding is performed correctly in all cases, using the round-to-nearest mode; the several rounding modes are not supported. The not-a-number (NAN) values are not supported. The standard exceptions and flags are not all supported. It is strongly recommended that a user of Paris always use the IEEE standard formats unless careful analysis of the application (such as a need for speed or additional exponent range) indicates that another format is required and adequate.

The format of a floating-point operand must obey certain restrictions. The length s must be greater than 0 and not greater than CM:*maximum-significand-length*. The length e must be greater than 1 and not greater than CM:*maximum-exponent-length*. See section 3.7. These restrictions are additionally imposed: $e \geq 2$, $s \geq 1$, and $2^{e-1} \geq s+1$. Values for s and e not satisfying these restrictions will cause unpredictable results.

[1] Thinking Machines Corporation does intend to support all standard IEEE arithmetic operations in a future software release.

3.5 Complex Floating-Point Numbers

A complex floating-point data item is specified by three parameters exactly like those for a floating-point data item: a bit address a, a significand length s, and an exponent length e. The data item consists of two consecutive floating-point data items, with the real part at address a and the imaginary part at address $a + s + e + 1$. The total number of bits in the representation is $2(s + e + 1)$, and the data item occupies the bits with addresses a through $a + 2(s + e) + 1$, inclusive.

3.6 Send Addresses

Every virtual processor in a VP set has an identifiying *send address*, a kind of serial number that distinguishes it from all other virtual processors in that VP set. These addresses are used to perform general interprocessor communication. For example, in the CM:send-1L operation, each virtual processor provides a message and the send address of some other processor, and that message is sent to the specified processor (all such messages effectively being sent in parallel).

The number of bits in a send address depends on the VP set, or rather upon the geometry of that VP set. The function CM:geometry-send-address-length may be used to determine the length in bits of a send address for a given geometry. Suppose that for geometry G this function returns m; then a send address a for a virtual processor in a VP set with geometry G is an unsigned integer such that $0 \leq a < 2^m$. (Programs should not, however, rely on the fact that every integer k such that $0 \leq k < 2^m$ is a valid send address. In a future release of Paris the space of send addresses may contain "holes"; this could occur when the total number of virtual processors in the geometry is not a power of two, an extension that Thinking Machines is contemplating for the future.)

3.7 Configuration Variables

The current configuration of the machine is reflected in a few global variables. Programs may refer to these so they can adapt to various sizes of machine. These variables are set by the cold boot procedure. They should never be set by the user, as there are dependencies among them, which, if violated, will result in errors. Some variables are fixed by the hardware, while others depend on the arrangement of virtual processors set up by the attach or cold boot process. Some variables represent implementation restrictions.

CM:*current-vp-set*

> The VP set ID for the current VP set is always available in this variable. For example, to determine the total number of processors in the current VP set, one might say (in Lisp syntax)
>
> ```
> (CM:geometry-total-processors
>   (CM:vp-set-geometry CM:*current-vp-set*))
> ```
>
> or (in C syntax)

```
CM_geometry_total_processors(CM_vp_set_geometry(CM_current_vp_set))
```

or (in Fortran syntax)

```
CM_GEOMETRY_TOTAL_PROCESSORS(CM_VP_SET_GEOMETRY(CM_CURRENT_VP_SET))
```

CM:*physical-processors-limit*

The total number of physical processors available for use.

CM:*physical-processors-length*

The base-2 logarithm of the total number of physical processors, that is, the minimum length in bits for an unsigned integer field that can contain the number of any physical processor.

CM:*physical-memory-limit*

The amount of physical memory per physical processor, including memory that is set aside for system use. **Note:** Also see the dictionary entry for CM:available-memory, which indicates how much Connection Machine memory is available for user programs.

CM:*physical-memory-length*

The base-2 logarithm of the amount of physical memory per physical processor.

CM:*maximum-integer-length*

Because of implementation restrictions, a few operations on signed and unsigned integers cannot handle operands longer than the value of CM:*maximum-integer-length*.

Experimentation might reveal that in certain cases some of these operations succeed when applied to operands that are longer than this variable, but that fact is not guaranteed in succeeding software releases.

The value of CM:*maximum-integer-length* is never smaller than 128.

CM:*maximum-significand-length*

Because of implementation restrictions, a few operations on floating-point numbers cannot handle operands with significands longer than a certain size.

Experimentation might reveal that in certain cases some of these operations succeed when applied to operands that are longer than specified by these variables, but that fact is not guaranteed in succeeding software releases.

The value of CM:*maximum-significand-length* is never smaller than 96.

CM:*maximum-exponent-length*

Because of implementation restrictions, a few operations on floating-point numbers cannot handle operands with exponents longer than a certain size.

Experimentation might reveal that in certain cases some of these operations succeed when applied to operands that are longer than specified by these variables, but that fact is not guaranteed in succeeding software releases.

The value of CM:*maximum-exponent-length* is never smaller than 32.

CM:*heap-compression-enabled*

When this variable is true (T, 1), automatic heap compression is enabled. See the dictionary entry for CM:compress-heap for information on explicit heap compression.

CM:*heap-compression-messages-enabled*

This variable determines whether a message is issued when heap compression occurs.

CM:*max-number-of-timers*

This represents the maximum number of timers that can be allocated by any one program using the CM:timer- functions.

CM:*no-field*

The value of this variable is a dummy field ID suitable for use as an argument to CM:send-1L and related instructions to indicate that no *notify* field is to be used, or to CM:scan-with-... operations to indicate an unused *sbit* argument when the *smode* argument is :none.

Chapter 4

Operation Formats

Paris operations are executed at the direction of a program running in the front-end machine. For each operation there is a function or macro that, when called, causes the Connection Machine hardware to perform the operation.

4.1 Field Id's

Most Paris operations operate on bit fields in the memories of the data processors. A bit field is specified by a *field id*, a data object that serves to identify the field. A Paris operation that allocates memory for a new field will generate and return a new field id; this field id may then be used as an argument to other Paris operations.

For example, in Lisp one might create a new heap field and then unconditionally initialize its contents to 5.0 in the following manner:

```
(let ((fld (CM:allocate-heap-field 32)))      ;Allocate
  (CM:f-move-const-always-1L fld 5.0 23 8)    ;Initialize
  ...)
```

In C the same operation would look like this:

```
{
    CM_field_id_t fld = CM_allocate_heap_field(32);  /* Allocate */
    CM_f_move_const_always_1L(fld, 5.0, 23, 8);      /* Initialize */
    ...
}
```

And in Fortran:

```
C  Declare the variable
      INTEGER FLD
      ...
C  Allocate and initialize
      FLD = CM_ALLOCATE_HEAP_FIELD(32)
      CM_F_MOVE_CONST_ALWAYS_1L(FLD, 5.0, 23, 8)
      ...
```

4.2 Constant Operands

Certain operations accept as an operand a single datum computed within the front end that is broadcast to all of the Connection Machine processors as part of the operation. Such operations have -constant in their names (or -const, in the case of certain compound operations). As a rule, every operation with -constant in its name has a counterpart without -constant in its name.

For example, to CM:f-add-constant-2-1L there corresponds CM:f-add-2-1L. These operations do exactly the same thing except that the first two operands to CM:f-add-2-1L are field id's for fields containing floating-point numbers, whereas CM:f-add-constant-2-1L takes a field id and a front-end floating-point number. This latter value is broadcast to all (active) processors and then used in the same way that a second field would be used by CM:f-add-2-1L. Here are examples of their use in Lisp:

```
(CM:f-add-2-1L x y 23 8)                ;Add field y into field x
(CM:f-add-constant-2-1L x 2.7 23 8)     ;Add 2.7 into field x
```

The same examples in C:

```
CM_f_add_2_1L(x, y, 23, 8);             /* Add field y into field x */
CM_f_add_constant_2_1L(x, 2.7, 23, 8);  /* Add 2.7 into field x */
```

The same examples in Fortran:

```
C  Add field y into field x
      CM_F_ADD_2_1L(X, Y, 23, 8)
C  Add 2.7 into field x
      CM_F_ADD_CONSTANT_2_1L(X, 2.7, 23, 8)
```

4.3 Unconditional Operations

Most Paris operations are conditional: they take place only in processors that have a 1 in the *context-flag*. But sometimes it is necessary to perform operations unconditionally (that is, without respect to the *context-flag*). A number of Paris operations have unconditional versions, generally named by inserting -always in the name of the conditional function. For example, CM:s-move-always-1L is the unconditional equivalent of CM:s-move-1L.

Paris operations that deal directly with the *context-flag* are inherently unconditional. For the sake of brevity, the names of these operations do not contain -always. Any Paris operation that has -context in its name deals with the *context-flag* and is implicitly unconditional despite the fact that -always does not also appear in its name. One example is CM:set-context.

A few other Paris operations also have only unconditional forms but do not have names containing -always. These are typically specialized communications operations whose names are already so long that inserting -always would exceed the limit on the length of a name. One example is CM:u-read-from-news-array-1L.

4.4 Naming Conventions

Lisp, C, and Fortran impose different sets of rules and conventions on how functions and variables are to be named. The description of Paris in this document strikes a compromise among these languages. All names in this document are presented in Lisp syntax, but carefully observing capitalization, to which C is sensitive even though Fortran and Lisp are not. The Paris Dictionary contains a simple set of rules for converting a Lisp name into the corresponding C or Fortran name.

The rest of this section describes the general rules that were used to achieve a regular naming system for Paris operations. It is not necessary to know these rules to use Paris, but a passing familiarity may help you to remember an exact operation name without having to look it up, or to recognize the argument format from the operation name.

The name of every Paris operation is limited to 32 characters and begins with CM: (in Lisp) or CM_ (in C and Fortran). It also contains one or more words that are the "main description" of the operation, such as add or send or read-from-news-array.

Between the leading CM: or CM_ and the main operation may be one or more prefixes. The prefix fe- indicates an operation performed entirely on the front end (often such an operation has a parallel counterpart without the fe- prefix). Examples of this correspondence are CM:extract-news-coordinate and CM:fe-extract-news-coordinate. If an fe- prefix is present, it appears before all other prefixes.

Other prefixes indicate the type of data to be operated upon:

c-	complex number
f-	floating-point number
s-	signed integer
u-	unsigned integer

For example, CM:f-add-2-1L adds floating-point numbers, whereas CM:s-add-2-1L add signed integers.

If there is more than one type prefix, then the first type applies to the result of the operation, and the other(s) apply to certain source operands, usually the last one(s). For example, CM:s-f-truncate-2-2L produces a signed integer result from a floating-point source.

Some operations include in their names the name of another operation. In this case the embedded operation may have a type prefix. An example is CM:spread-with-f-add-1L. (The name of such an embedded operation is usually preceded by with-, but exceptions occur when this would make names too long, as in CM:multispread-f-multiply-1L, an operation that is not yet implemented but may be in the future.)

There are four groups of *suffixes* for operation names: -constant, -always, number of fields, and number of lengths. They always appear (if at all) in this order.

A number-of-fields suffix is simply a digit (preceded by a hyphen or underscore), such as -3. It tells how many source and destination arguments an instruction requires. The destination arguments are fields; the source arguements are fields, or in some cases constants. In many cases there are sets of similar operations differing primarily in their argument format. For example, CM:f-multiply-3-1L takes three fields and stores the floating-point product of the second and third fields into the first field, whereas CM:f-multiply-2-1L takes only two fields, and stores their product back into the first field (thereby overwriting one source value).

These two formats are distinguished by a suffix indicating the number of arguments that are fields (in this case -3 or -2). As a rule, this suffix is supplied only if it is necessary to distinguish two or more possible formats. (Note that "field-like" arguments, such as the constant used in place of a field in CM:f-multiply-constant-2-1L, are included in the number-of-fields count.)

A number-of-lengths suffix is simply a digit (preceded by a hyphen or underscore) followed by a capital L, such as -3L. This suffix indicates how many length arguments are required. Such arguments indicate the lengths of field arguments. For example, CM:s-add-3-3L takes three field arguments followed by three corresponding length arguments; but CM:s-add-3-1L takes three field arguments and a single length argument that describes the length of all three fields. Note that the format of a floating-point field is described by *two* arguments (significand length and exponent length), but these two arguments are lumped together and counted as a single length. As a rule this suffix always appears in the name of any operation that takes one or more field length arguments.

To summarize, the name of a Paris operation is more or less of this form:

CM:[fe-]{f- | s- | u-}*⟨main name⟩[⟨embedded name⟩][-constant][-always][-m][-nL]

An effort has been made to use full English words in the names of Paris operations. The 32-character limitation on the total length of names has made it necessary to use certain abbreviations universally:

c-	complex floating-point
divinto	divide into
fe-	front end
f-	floating-point
max	maximum
min	minimum
mod	modulo
rem	remainder
s-	signed integer
subfrom	subtract from
u-	unsigned integer

Some of these are standard abbreviations, of course, used in many programming languages. Paris also uses standard abbreviated names for mathematical operations (tan for the tangent function, for example).

Paris uses certain additional abbreviations in the names of compound operations:

mult	multiply
const	constant
sub	subtract
a	always

An example is CM:f-mult-const-sub-const-a-1L.

4.5 Argument Order

An attempt has been made to keep argument order consistent. The following rules of thumb apply.

Arguments that are fields come first. If there is a destination field it always comes first.

Length fields usually come last. They appear in the same order as the fields to which they apply, but if both integer and floating-point fields appear then the floating-point length arguments appear last. For some complex communication operations, such as scan operations, certain control arguments follow the lengths.

Chapter 5

Instruction Set Overview

This chapter provides a quick guided tour of the entire Paris instruction set, organized by categories of functionally related operations. The names of the operations are presented in the form of charts that bring out the combinatorial structure of the instruction set. Alternatives are stacked vertically between braces, and the symbol $\sim$ indicates a choice that adds no characters to the operation name.

The next chapter, the Paris Dictionary, is organized alphabetically by operation name, and provides detailed descriptions of all the operations.

5.1 VP Sets

CM: { allocate-vp-set | deallocate-vp-set | physical-vp-set | is-vp-set-valid | set-vp-set | set-vp-set-geometry | vp-set-geometry }

These operations create, destroy, and otherwise manipulate VP sets.

The operation CM:allocate-vp-set creates a new VP set having a specified geometry (which must be created first). The operation CM:deallocate-vp-set may be used to inform the Paris interface that the user program will not use a VP set any longer.

Of particular importance is CM:set-vp-set, which selects a given VP set as the current VP set.

Given a VP set, the operation CM:vp-set-geometry returns the geometry associated with that VP set.

5.2 Geometries

CM: { create-detailed-geometry, create-geometry, deallocate-geometry, geometry-axis-length, geometry-axis-off-chip-bits, geometry-axis-off-chip-pos, geometry-axis-on-chip-bits, geometry-axis-on-chip-pos, geometry-axis-ordering, geometry-axis-vp-ratio, geometry-coordinate-length, geometry-rank, geometry-send-address-length, geometry-total-processors, geometry-total-vp-ratio }

These operations create, destroy, and otherwise manipulate geometries. Note the many operations that inquire about the shape of the geometry and various axis attributes.

5.3 Interned Geometries and vp Sets

Paris supports a special class of geometry and VP set objects: *interned* objects. The interning facility is especially useful to compiler writers because interned objects may be accessed by description rather than by ID and are automatically reused as needed.

CM: { intern-geometry, intern-detailed-geometry, intern-identical-vp-set }

These operations create interned geometries and VP sets.

Note that interned geometries and VP sets are substantively different kinds of objects from their uninterned couterparts. For instance, a geometry created with CM:create-geometry is never interchangeable with a geometry created with CM:intern-geometry.

5.4 Fields

CM: { add-offset-to-field-id | allocate-heap-field | allocate-heap-field-vp-set | allocate-stack-field | allocate-stack-field-vp-set | deallocate-heap-field | deallocate-stack-through | field-vp-set | is-field-in-heap | is-field-in-stack | is-field-valid | is-stack-field-newer | next-stack-field-id }

These operations create, destroy, and otherwise manipulate fields. Fields are used to contain data to be operated upon in parallel. Most Paris operations require one or more fields as arguments.

CM:available-memory

This instruction indicates the number of bits of memory, per virtual processor, currently available for allocation on either the heap or stack.

CM:compress-heap

Automatic heap compression is enabled by default. Programmers can control heap compression explicitly by setting the configuration variable CM:*heap-compression-enabled* to NIL (false, 0) and then calling the above instruction to control fragmentation.

5.5 Copying Fields

A number of operations are provided simply to copy data from one place to another.

CM: { s- | u- | f- | c- } move { { ~ | -constant | -zero } { ~ | -always } { -2L | -1L } }

The two-length versions of the move operations allow for sign-extension (or truncation) of signed integers, zero-extension (or truncation) of unsigned integers, and changes of range or precision for floating-point numbers.

$$\text{CM:} \left\{ \begin{array}{ll} \text{move-reversed} & \left\{ \begin{array}{c} \sim \\ \text{-always} \end{array} \right\} \\ \text{swap} & \left\{ \begin{array}{c} \sim \\ \text{-always} \end{array} \right\} \text{-2} \end{array} \right\} \text{-1L}$$

The move-reversed operation reverses the order of the bits in a field as it copies them. The swap operation exchanges the contents of two fields.

$$\text{CM:cross-vp-move} \left\{ \begin{array}{c} \sim \\ \text{-always} \end{array} \right\} \text{-1L}$$

The cross-vp-move instruction copies all or a portion of one multidimensional block of data from the current VP set into a similarly shaped region in another VP set.

5.6 Field Aliasing

$$\text{CM:} \left\{ \begin{array}{l} \text{change-field-alias} \\ \text{is-field-an-alias} \\ \text{make-field-alias} \\ \text{remove-field-alias} \\ \text{set-field-alias-vp-set} \end{array} \right\}$$

These operations create, destroy, and manipulate field aliases. A *field alias* is a field ID that references a field already referenced by at least one other field ID. By using field aliases, it is possible to reference the same Connection Machine memory field from within different VP sets.

5.7 Bitwise Boolean Operations

$$\text{CM:} \left\{ \begin{array}{l} \text{logand} \\ \text{logior} \\ \text{logxor} \\ \text{logeqv} \\ \text{lognand} \\ \text{lognor} \\ \text{logandc1} \\ \text{logandc2} \\ \text{logorc1} \\ \text{logorc2} \end{array} \right\} \left\{ \begin{array}{c} \sim \\ \text{-constant} \\ \text{-always} \\ \text{-const-always} \end{array} \right\} \left\{ \begin{array}{l} \text{-2-1L} \\ \text{-3-1L} \end{array} \right\}$$

CM:lognot { -1-1L | -2-1L }

Paris provides all ten non-trivial bitwise boolean operations on two operands, as well as the logical NOT operation that inverts all bits.

5.8 Operations on Flags

Special operations are provided for operating on the flags.

CM: { load- | store- | clear- | set- | invert- | logand- | logior- | global-logand- | global-logior- | global-count- } { test | overflow }

Flags can be loaded from or stored into another field; cleared to zero or set to one; inverted; or combined with another field via logical AND or OR. One may also determine whether any processor, or all processors, have a flag set, or count the number of processors that have a flag set.

CM:clear-all-flags { ~ | -always }

For convenience, a special compound operation is provided for clearing all the flags except the context.

CM: { { load- | store- | clear- | set- | invert- | logand- | logior- | global-logand- | global-logior- | global-count- } context | logand-context-with-test }

The context flag is distinguished from the others, in that operations on the context flag are always unconditional, while most operations on the other flags are conditional (that is,

depend on the state of the context flag).

5.9 Operations on Single Bits

Each of the following operations takes exactly one one-bit field as its operand.

CM: { clear- / set- / global-logand- / global-logior- / global-count- } bit { ~ / -always }

These operations on single-bit fields are provided purely for the sake of efficiency. For example,

CM:clear-bit x

has the same effect as

CM:u-move-constant-1L x, 0, 1

but requires only one operand to be processed instead of three. Paris also provides unconditional forms of all these operations.

5.10 Unary Arithmetic Operations

Paris supports most of the unary arithmetic operations one might expect to find in a computer instruction set, as well as a number that are unusual. Most of them are provided in both one-operand and two-operand formats. The one-operand format treats the destination field as also the source operand; the result replaces the input. The two-operand format has a separate source operand, and ignores the previous contents of the destination field. (As a rule, the two-operand format operates correctly if the two operands are the same field, but may be slower than using the one-operand format.)
For signed and unsigned integers there are negation and integer square root. Absolute value and signum are provided for signed operands only, as these operations are degenerate in the unsigned case.

CM: { { { s- / u- } { negate / isqrt } / s- { abs / s-signum } } { -1-1L / -2-1L / -2-2L } }

The integer-length operation is a modified base-2 logarithm, useful for determining the minimum number of bits required to represent an integer in signed or unsigned form. The logcount operation counts the number of 1-bits in a binary representation (or, in the signed case, it counts the bits that differ from the sign bit).

$$\text{CM:} \left\{\begin{matrix}\text{s-}\\ \text{u-}\end{matrix}\right\} \left\{\begin{matrix}\text{integer-length}\\ \text{logcount}\end{matrix}\right\} \text{-2-2L}$$

A shift instruction performs an arithmetic shift by a specified number of bit positions. Paris supports shifts on either signed or unsigned source fields.

$$\text{CM:} \left\{\begin{matrix}\text{s}\\ \text{u}\end{matrix}\right\} \text{-s-shift} \left\{\begin{matrix}\text{-2}\\ \text{-constant-3}\end{matrix}\right\} \text{-2L}$$

Operations are provided for converting to and from a Gray code representation of binary integers.

$$\text{CM:u-} \left\{\begin{matrix}\text{from}\\ \text{to}\end{matrix}\right\} \text{-gray-code} \left\{\begin{matrix}\text{-1-1L}\\ \text{-2-1L}\end{matrix}\right\}$$

These Paris instructions support converting floating-point numbers between the IEEE format used in the Connection Machine system and VAX floating-point format.

$$\text{CM:f-} \left\{\begin{matrix}\text{ieee-to-vax}\\ \text{vax-to-ieee}\end{matrix}\right\} \text{-1L}$$

Some unary operations take a floating-point operand and produce an integer result, or vice versa. The float operations convert an integer to a floating-point representation. There are several different ways to convert a floating-point number to an integer, reflecting different possible choices for rounding or truncation; floor and truncate provide two such cases.

$$\text{CM:} \left\{\begin{matrix}\text{f-} \left\{\begin{matrix}\text{s-}\\ \text{u-}\end{matrix}\right\} \text{float}\\ \\ \text{s-} \quad \text{f-} \left\{\begin{matrix}\text{floor}\\ \text{truncate}\end{matrix}\right\}\end{matrix}\right\} \left\{\text{-2-2L}\right\}$$

Floating-point and complex absolute value, negation, and square root are provided.

$$\text{CM:} \left\{\begin{matrix}\text{c-}\\ \text{f-}\end{matrix}\right\} \left\{\begin{matrix}\text{abs}\\ \text{negate}\\ \text{sqrt}\end{matrix}\right\} \left\{\begin{matrix}\text{-1-1L}\\ \text{-2-1L}\end{matrix}\right\}$$

Floating-point floor, ceiling, truncation, rounding, and signum operations are available.

$$\text{CM:f-}\left\{\begin{array}{l}\text{f-floor}\\\text{f-ceiling}\\\text{f-truncate}\\\text{f-round}\\\text{f-signum}\end{array}\right\}\left\{\begin{array}{l}\text{-1-1L}\\\text{-2-1L}\end{array}\right\}$$

Complex signum, conjugate, and reciprocal operations are provided.

$$\text{CM:c-}\left\{\begin{array}{l}\text{c-signum}\\\text{c-conjugate}\\\text{c-reciprocal}\end{array}\right\}\left\{\begin{array}{l}\text{-1-1L}\\\text{-2-1L}\end{array}\right\}$$

These two unary operations on complex operands yield floating-point destination values. One calculates the absolute value and the other calculates the phase of each complex source value.

$$\text{CM:f-c-}\left\{\begin{array}{l}\text{abs}\\\text{phase}\end{array}\right\}\ \text{-2-1L}$$

For both floating-point and complex numbers, Paris provides a complete set of transcendental and trigonometric functions, including hyperbolic functions and their inverses.

$$\text{CM:}\left\{\begin{array}{l}\text{f}\\\text{c}\end{array}\right\}\left\{\begin{array}{l}\text{-exp}\\\text{-ln}\\\left\{\begin{array}{l}\sim\\\text{-a}\end{array}\right\}\left\{\begin{array}{l}\text{-sin}\\\text{-cos}\\\text{-tan}\\\text{-sinh}\\\text{-cosh}\\\text{-tanh}\end{array}\right\}\end{array}\right\}\left\{\begin{array}{l}\text{-1-1L}\\\text{-2-1L}\end{array}\right\}$$

In addition, the cis instruction is available. It yields a complex field in which the real part is the cosine of the floating-point source and the imaginary part is the sine of the source.

CM:c-f-cis-2-1L

5.11 Binary Arithmetic Operations

Paris includes most of the binary arithmetic operations one might expect to find in a computer instruction set, as well as a number that are unusual. Most of them are provided

in both two-operand and three-operand formats. The two-operand format treats the destination field as also one of source operands; the result replaces the first input. The three-operand format has two separate source operands, and ignores the previous contents of the destination field. (As a rule, the three-operand format operates correctly if the destination field is the same as one or both source fields, but may be slower than using a two-operand format.)

For signed and unsigned integers, the usual addition, subtraction, and multiplication operations are provided, as well as max and min operations that store the larger or smaller of the two inputs.

There is no single integer division operation; four are provided by the signed and unsigned round and truncate instructions, whose names reflect the rounding or truncation that must occur when integer division is not exact. Conceptually there are four corresponding remainder operations, but only the two most commonly used are provided in Paris: rem, which corresponds to truncate division; and mod, which corresponds to floor division.

CM: { s, u } { -add, -subtract, -multiply, -max, -min, -floor, -ceiling, -truncate, -round } { -3-3L, { ~, -constant } { -2-1L, -3-1L } }

CM: { s-, u- } { rem, mod } { ~, -constant } { -2-1L, -3-1L }

Subtraction is not commutative, and so for efficiency the special case of reverse subtraction is provided. (Division is not commutative, either, but is a sufficiently expensive operation that the relative cost of a separate instruction to copy a constant into a temporary field first is small. Paris therefore does not provide integer reverse division operations.)

CM: { s, u } -subfrom { -constant { -2-1L, -3-1L } }

Paris allows addition and subtraction on integers hundreds of bits long; but in case that is not enough, the usual add-carry and subtract-borrow operations, which use the carry flag as an implicit input, are provided to allow efficient programming of very high precision integer arithmetic. Since the add-carry and subtract-borrow instructions take the *carry-flag* as input as well as setting it upon completion, these instructions can be chained. (The one exception to this rule are the -add-carry-3-3L instructions, which do not set the *carry-flag*

because it is unclear what carry means in the 3L case.)

$$\text{CM:} \left\{\begin{matrix}\text{s-}\\ \text{u-}\end{matrix}\right\} \left\{\begin{matrix}\text{add-carry}\\ \text{subtract-borrow}\end{matrix}\right\} \left\{\begin{matrix}\text{-3-3L}\\ \text{-2-1L}\\ \text{-3-1L}\end{matrix}\right\}$$

The add-flags operation performs an addition and sets the flags but stores no sum. This is useful in a few specialized situations, such as CORDIC-type calculations.

$$\text{CM:} \left\{\begin{matrix}\text{s-}\\ \text{u-}\end{matrix}\right\} \text{add-flags-2-1L}$$

For floating-point and complex numbers, the usual addition, subtraction, multiplication, and division operations are provided. Note that there are unconditional versions of these operations in Paris; these can be much faster than the conditional versions when floating-point hardware is used.

$$\text{CM:} \left\{\begin{matrix}\text{c-}\\ \text{f-}\end{matrix}\right\} \left\{\begin{matrix}\text{add}\\ \text{subtract}\\ \text{multiply}\\ \text{divide}\end{matrix}\right\} \left\{\begin{matrix}\sim\\ \text{-constant}\\ \text{-always}\\ \text{-const-always}\end{matrix}\right\} \left\{\begin{matrix}\text{-2-1L}\\ \text{-3-1L}\end{matrix}\right\}$$

For floating-point numbers, max and min operations are provided, along with floating-point remainder and modulo division operations, and a floating-point exponentiation instruction.

$$\text{CM:f} \left\{\begin{matrix}\text{-max}\\ \text{-min}\\ \text{-mod}\\ \text{-rem}\\ \text{-f-power}\end{matrix}\right\} \left\{\begin{matrix}\sim\\ \text{-constant}\end{matrix}\right\} \left\{\begin{matrix}\text{-2-1L}\\ \text{-3-1L}\end{matrix}\right\}$$

Subtraction and division are not commutative, and so for efficiency special cases of reverse subtraction and reverse division are provided for floating-point and complex floating-point operands. (Unlike the integer case, floating-point division is sufficiently fast and sufficiently common that these special cases are worthwhile.)

$$\left\{\begin{matrix}\text{CM:c-}\\\text{CM:f-}\end{matrix}\right\}\left\{\begin{matrix}\text{subfrom}\\\text{divinto}\end{matrix}\right\}\left\{\begin{matrix}\left\{\begin{matrix}\sim\\\text{-always}\end{matrix}\right\}\text{-2-1L}\\\left\{\begin{matrix}\text{-constant}\\\text{-const-always}\end{matrix}\right\}\left\{\begin{matrix}\text{-2-1L}\\\text{-3-1L}\end{matrix}\right\}\end{matrix}\right\}$$

Other useful floating-point operations include scaling, as well as exponentiating to an integer power.

$$\text{CM:f}\left\{\begin{matrix}\text{-s}\\\text{-u}\end{matrix}\right\}\left\{\begin{matrix}\text{-power}\\\text{-scale}\end{matrix}\right\}\left\{\begin{matrix}\text{-2-2L}\\\text{-3-2L}\\\text{-constant-2-1L}\\\text{-constant-3-1L}\end{matrix}\right\}$$

Paris supports integer exponentiation instructions for both signed and unsigned operands.

$$\text{CM:}\left\{\begin{matrix}\text{s}\\\text{u}\end{matrix}\right\}\left\{\begin{matrix}\text{-s}\\\text{-u}\end{matrix}\right\}\left\{\begin{matrix}\text{-power-3-3L}\\\text{-power-constant-2-1L}\\\text{-power-constant-3}\quad\left\{\begin{matrix}\text{-1L}\\\text{-2L}\end{matrix}\right\}\end{matrix}\right\}$$

Exponentiation of complex number is supported for powers of any data type.

$$\text{CM:c-}\left\{\begin{matrix}\text{c-}\\\text{f-}\\\text{s-}\\\text{u-}\end{matrix}\right\}\text{power}\left\{\begin{matrix}\text{-2-1L}\\\text{-3-1L}\\\text{-constant-2-1L}\\\text{-constant-3-1L}\end{matrix}\right\}$$

The exp operations calculate e^s for complex operands and 2^s for floating-point operands, where s is the value of the *source* field and e is the base of the natural logarithms.

$$\text{CM:}\left\{\begin{matrix}\text{c}\\\text{f}\end{matrix}\right\}\text{-exp}\left\{\begin{matrix}\text{-1-1L}\\\text{-2-1L}\end{matrix}\right\}$$

Instructions are provided that calculate the base 2 or base 10 logarithm of a floating-point source field or the natural logarithm of a complex source field.

$$\text{CM:} \left\{ \begin{matrix} \text{f-log2} \\ \text{f-log10} \\ \text{c-ln} \end{matrix} \right\} \left\{ \begin{matrix} \text{-1-1L} \\ \text{-2-1L} \end{matrix} \right\}$$

A two-input arctangent operation is provided.

CM:f-atan2-3-1L

5.12 Optimized Floating-Point Computations

Paris supports compound floating-point operations that are functionally identical to sequences of simpler floating-point operations. The compound operations are provided purely for the sake of efficiency; they can be implemented so to exploit floating-point hardware more cleverly.

These compound operations perform calculations of the following forms: $xa + b$, $xa - b$, $(x + a)b$, and $(x - a)b$, where x is always a field in memory, and a and b may each be either a field or a constant.

$$\text{CM:f} \left\{ \begin{matrix} \text{-mult} \left\{ \begin{matrix} \sim \\ \text{-const} \end{matrix} \right\} \left\{ \begin{matrix} \text{-add} \\ \text{-sub} \\ \text{-subf} \end{matrix} \right\} \left\{ \begin{matrix} \sim \\ \text{-const} \end{matrix} \right\} \\ \left\{ \begin{matrix} \text{-add} \\ \text{-sub} \\ \text{-subf} \end{matrix} \right\} \left\{ \begin{matrix} \sim \\ \text{-const} \end{matrix} \right\} \text{-mult} \left\{ \begin{matrix} \sim \\ \text{-const} \end{matrix} \right\} \end{matrix} \right\} \left\{ \begin{matrix} \sim \\ \text{-always} \\ \text{-a} \end{matrix} \right\} \text{1L}$$

Note: Where using the term -always in an unconditional instruction name would cause the name to exceed the 32 character limit for Paris instruction names, the implementation uses the term -a instead. In the above chart, this is the case only for instructions that contain const twice. An example is CM:f-sub-const-mult-const-a-1L.

These compound instructions combine floating-point multiplication with reverse subtraction in a variety of ways. The unconditional versions may be faster than the conditional versions. (Note that the name CM:subf-const-mult-const-a-1L uses -a instead of -always in order to stay within the 32-character Paris operation name length limit.)

$$\text{CM:f} \left\{ \begin{matrix} \text{-mult-subf} \\ \text{-mult-const-subf} \\ \text{-subf-const-mult} \end{matrix} \right\} \left\{ \begin{matrix} \sim \\ \text{-const} \end{matrix} \right\} \left\{ \begin{matrix} \sim \\ \text{-always} \end{matrix} \right\} \text{-1L}$$

5.13 Arithmetic Comparisons

Paris supports the usual six comparison operations $=$, $\neq$, $<$, $\leq$, $>$, and $\geq$ for integers and floating-point numbers. Each is available in three forms: compare two fields, compare a field to a constant, and compare a field to zero. The integer operations also allow integer fields of differing length to be compared.

CM: { s / u } { -eq / -ne / -lt / -le / -gt / -ge } { -2L / { ~ / -constant / -zero } -1L }

CM:f- { eq / ne / lt / le / gt / ge } { ~ / -constant / -zero } -1L

CM:c- { eq / ne } { ~ / -constant / -zero } -1L

5.14 Pseudo-Random Number Generation

Paris provides a built-in generator of uniformly distributed pseudo-random numbers. Use these instructions to generate unsigned integers over a specified range, or floating-point numbers in the range from 0.0 (inclusive) to 1.0 (exclusive).

CM: { u- / f- } random -1L

CM:initialize-random-generator

5.15 Arrays

Often it is convenient to treat a large field as an array of smaller fields. These operations allow each virtual processor to index independently into its own array.

CM: { aref | aref32 { ~ | -shared } { ~ | -always } | aset | aset32 { ~ | -shared } } -2L

Three kinds of arrays are supported. An ordinary array is laid out in memory exactly as one would expect: each processor contains its own array elements, concatenated end-to-end to form one large field.

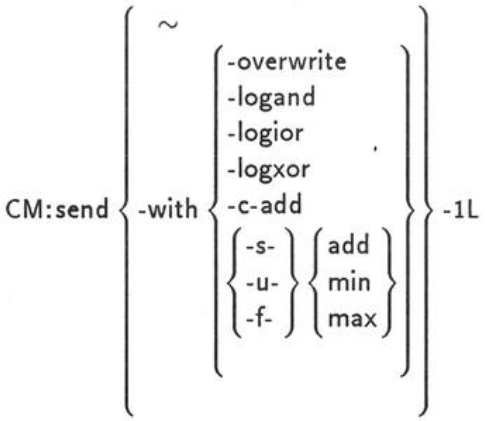
A *slicewise* array is laid out in such a way that an array element logically belonging to one processor is actually stored in memory belonging to 32 processors. The total amount of memory involved is the same, of course, but because the data is laid out in this peculiar manner ordinary Paris operations (such as CM:f-add-2-1L, for example) cannot properly operate on slicewise array elements directly. Only special operations designed to operate on slicewise arrays can properly fetch or store slicewise array elements. Examples are CM:aref32-2L and CM:aset32-2L. These special operations are much faster than the corresponding operations on ordinary arrays.

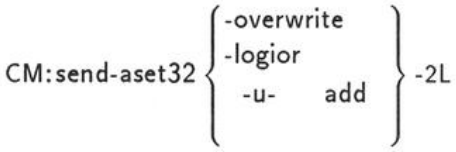
A *shared* array is shared among all the virtual processors occupying a group of 32 physical processors. This can save a great deal of memory, and is useful for lookup tables that are the same for all processors. Of course, care is required when storing into such arrays. In principle this sharing concept could be supported in both ordinary and fast versions, but in fact Paris provides special operations only for fast shared arrays.

Paris also provides, for efficiency, certain compound operations that combine communication with access to a fast array.

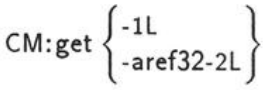
5.16 General Communication

The router functions (**send** and **get**) transmit data in a general fashion that allows any processor to communicate directly with any other processor.





CM:send-to-queue32-1L



CM:my-send-address

Every processor within a VP set is identified by an unsigned binary integer called its *send-address.* If processor A is to send a message M to processor B, then procesor A must contain the send-address of processor B as well as the data M to be sent.

For efficiency, Paris includes compound operations that combine general communication with a fast array reference (aref32 or aset32) within the addressed processor.

5.17 NEWS Communication

The NEWS functions (`send-to-news` and `get-from-news`) organize the processors into a multidimensional rectangular grid, and transmit data from every processor to its neighbor along a specified grid axis. The NEWS operations are considerably more efficient, when applicable, than using the general router mechanism.

The following operations copy data from each processor to the adjacent processor along any NEWS axis.

$$\text{CM: } \left\{\begin{array}{l}\text{get-from-}\\ \text{send-to-}\end{array}\right\} \text{news} \left\{\begin{array}{c}\sim\\ \text{-always}\end{array}\right\} \text{-1L}$$

The instructions in the chart below all work with NEWS coordinates.

$$\text{CM: } \left\{\begin{array}{l}\text{my-news-coordinate}\\ \text{extract-news-coordinate}\\ \text{deposit-news-coordinate}\\ \text{deposit-news-constant}\\ \text{make-news-coordinate}\end{array}\right\} \text{-1L}$$

The operation *my-news-coordinate* stores the NEWS coordinate of each selected processor along a specified NEWS axis into a destination field within that processor.

The operation *extract-news-coordinate* defines the mapping between send-addresses and NEWS coordinates. If g is a geometry, a is an axis number, and s is a send-address, then *extract-news-coordinate*(g, a, s) is the coordinate within geometry g of processor s along the NEWS axis described by a.

A related operation, *deposit-news-coordinate*, may be used to construct a send-address given a set of coordinates by incrementally modifying a send-address one coordinate at a time. If g is a geometry, s is a send-address (for a processor in that geometry), a is an axis number, and c is a coordinate along that axis, then *deposit-news-coordinate*(g, s, a, c) is a new send address s' such that

$$\textit{extract-news-coordinate}(g, a', s') = \begin{cases} c, & \text{if } a' = a \\ \textit{extract-news-coordinate}(g, a', s), & \text{if } a' \neq a \end{cases}$$

In other words, *deposit-news-coordinate*(g, s, a, c) computes a new send-address that has exactly the same NEWS coordinates as s *except* for the coordinate on axis a, which is altered to be c.

Another related operation, *make-news-coordinate*, constructs, within each selected processor, the send-address of a processor that has a specified coordinate along a specified NEWS axis, with all other coordinates zero. If g is a geometry, a is an axis number, and c is a coordinate along a, then *make-news-coordinate*(g, a, c) is s, the send-address of the processor with coordinate c along the NEWS axis a within geometry g and with all other coordinates held at zero. Thus, given a set of zero coordinates of $rank(g)$, s',

$$\textit{make-news-coordinate}(g, a, c) = \textit{deposit-news-coordinate}(g, s', a, c) = s$$

In other words, *make-news-coordinate* is the same as *deposit-new-coordinate* except that it does not need a send-address operand.

The following routines define the relationship between a processor whose send-address is k and its neighbors in a NEWS grid.

function *news-neighbor*$(g, k, axis, direction)$ is
  return *news-relative*$(g, k, axis, direction, 1)$

function *news-relative*$(g, k, axis, direction, distance)$ is
  case *direction* of
    :upward : let $x = (extract\text{-}news\text{-}coordinate(g, axis, k) + distance)$
    :downward : let $x = (extract\text{-}news\text{-}coordinate(g, axis, k) - distance)$
  let $x' = x \bmod geometry\text{-}axis\text{-}length(g, axis)$
  return $deposit\text{-}news\text{-}coordinate(g, k, axis, x')$

5.18 Power of Two NEWS

One special-purpose instruction performs near-neighbor communication between processors that are separated by a particular distance. That distance must be a power of two, measured in intervening processors and inclusive of the source processor.

CM:get-from-power-two { ~ / -always } -1L

5.19 NEWS with Floating-Point Combiners

A series of special-case combining operations that use NEWS communication are supported. These instructions calculate a form of binary addition, subtraction, and multiplication in which one operand is retrieved from a NEWS neighbor of the destination field.

CM:f-news {
{ -add / -sub / -mult } { ~ / -always } { -2-1L / -3-1L }
{ -add-const / -sub-const } { ~ / -a } -3-1L
{ ~ / -const } -mult-4-1L
-mult-const { ~ / -a } -4-1L
-mult { ~ / -const } { -add / -sub } -4-1L
}

5.20 Scan, Reduce, Spread, and Multispread

The spread-from-processor operation provides a simple way to take the value found in one processor and replicate it throughout the machine.

CM:spread-from-processor-{~ | a-} 1L

Extending this idea, the following operations provide extremely powerful combinations of communication and computation in regular patterns on multidimensional grids.

CM: {scan-with | reduce-with | spread-with | multispread} {-copy | -logand | -logior | -logxor | -c-add | {-s- | -u- | -f-} {add | min | max}} -1L

CM:scan-with-f-multiply -1L

CM:enumerate -1L

In a scan operation, every selected processor receives the result of combining source fields from many processors. The reduce and spread operations are special cases of scans that are particularly useful and can be made especially fast. The multispread and enumerate operations generalize the spread operations.

A scan operation requires that a NEWS axis be specified. The processors are thereby divided into disjoint ordered sets of processors called *scan classes*. Two processors belong to the same scan class if their NEWS coordinates differ only along one axis, and they are ordered by their coordinates along that axis. Only active processors participate in a scan operation; all scan and scan-like operations are conditional. The set of active processors along a NEWS axis is called the *scan subclass*.

The scan result computed for a given processor may be produced by combining values from all processors within a scan subclass. That is, all active processors along a specified axis may contribute to the result for each processor along that axis. However – and more usefully – a scan subclass may be divided into pieces called *scan sets*, such that each processor belongs to just one scan set.

The scan set chosen for each processor is controlled by the *smode* operand and by the purpose it assigns to the *sbit* operand.

- If *smode* is :segment-bit, then the *sbit* field is interpreted as a "segment bit."

The segment bit divides a scan class unconditionally (that is, without respect to context) into segments, and a separate scan operation is done within each segment. Operationally speaking, a processor (active or not) is the lowest-addressed processor in a segment if either it is the lowest-addressed processor in its scan class or if its *sbit* field value is 1.

There are two remarkable points here. First, the way in which a segment bit divides a scan class does not depend on either the *context-flag* or the direction of the scan. Second, values from one segment never contribute to the result for any processor in another segment.

- If *smode* is :start-bit, then the *sbit* field is is interpreted as a "start bit."

  Operationally speaking, in each selected processor in which this bit is 1, the scan operation will start over again. The start bit therefore divides a scan subclass into pieces, and a scan operation is done within each piece, or scan set. These pieces differ from the segments determined by a segment bit.

  There are three remarkable points here. First, the start bit is examined only in selected processors. Second, the way in which a start bit divides a scan subclass depends on the direction of the scan. In an upward scan, a processor with a start bit of 1 is the first participant in a scan set that includes its neighbor with the next higher coordinate along the specified NEWS axis; in a downward scan, the same processor begins a scan set that includes its neighbors with lower NEWS axis coordinates.

  Third, for an exclusive scan, a selected processor whose start bit is 1 will receive the identity for the combining operation only if no other selected processor in the same scan subclass precedes it in the ordering; otherwise, it will receive the combined values from all processors in the piece preceding it in the ordering. (Exclusive scans are described below.)

- If *smode* is :none, then there is no need for a one-bit field, and the *sbit* operand is ignored. The scan set for a processor k is the entire scan subclass for k.

A scan operation furthermore behaves as if all the processors in the specified scan set were passed over ("scanned") in linear order; therefore the result computed for a given processor, k, depends only on processors below it in the ordering, or only on processors above it, depending on the direction of the scan. The *direction* and *inclusion* operands determine which processors within the scan set can potentially contribute to the result for k. This final, most narrowed set of potential contributors is called the *scan subset* for k.

If *direction* is :upward, then the scan subset for processor k will contain only processors below k in the ordering. If *direction* is :downward, then the scan subset for k will contain only processors above k in the ordering.

If *inclusion* is :exclusive, then the scan subset for processor k will not contain k itself. If *inclusion* is :inclusive, then the scan subset for k will contain k itself.

The set of processors whose *source* fields actually do contribute to the *dest* field of processor k is called the *scan subset* for k. This will be a subset of the scan set for k (possibly the entire scan set).

These concepts are embodied in the following pseudo-code routines, which are used in the Paris Dictionary to describe the behavior of the scan, spread, reduce, rank, and multispread operations.

Consider representing several NEWS coordinate values in a single integer called a *multi-coordinate.* We can define two operations, *extract-multi-coordinate* and *deposit-multi-coordinate*, for accessing and altering multi-coordinates. They are analogous to *extract-news-coordinate* and *deposit-news-coordinate*, the difference being simply that a multi-coordinate contains values for several news coordinates.

Suppose that g is a geometry, A is an axis-set, and s and t are send-addresses, and let

$$s' = \textit{deposit-multi-coordinate}(g, s, A, \textit{extract-multi-coordinate}(g, A, t))$$

Then s' is the same as s except that coordinates for axes in A have been replaced by corresponding coordinates extracted from t. More formally,

$$\textit{extract-news-coordinate}(g, a, s') = \begin{cases} \textit{extract-news-coordinate}(g, a, s), & \text{if } a \notin A \\ \textit{extract-news-coordinate}(g, a, t), & \text{if } a \in A \end{cases}$$

The Paris instruction CM:multispread-copy-1L actually requires a multi-coordinate as an argument and the instruction CM:fe-extract-multi-coordinate constructs a multi-coordinate. Beyond this, the notion of a multi-coordinate providess a useful conceptual building block in the following pseudo-code definitions.

Now we can define scan classes in terms of the more general concept of a *hyperplane*, which is any subset of the processors obtained by holding some NEWS coordinates fixed while letting the others range freely over their respective axes.

function $\textit{hyperplane}(g, k, \textit{axis-set})$ is
  let $\textit{other-axes} = \{\, a \mid 0 \le a < \textit{rank}(g) \,\} \setminus \textit{axis-set}$
  let $c = \textit{extract-multi-coordinate}(g, \textit{other-axes}, k)$
  return $\{\, m \mid m \in \textit{current-vp-set} \wedge \textit{extract-multi-coordinate}(g, \textit{other-axes}, m) = c \,\}$

function $\textit{scan-class}(g, k, \textit{axis})$ is
  return $\textit{hyperplane}(g, k, \{\, \textit{axis} \,\})$

function $\textit{scan-subclass}(g, k, \textit{axis})$ is
  return $\{\, m \mid m \in \textit{scan-class}(g, k, \textit{axis}) \wedge \textit{context-flag}[m] = 1 \,\}$

function *scan-set*$(g, k, axis, direction, smode, sbit)$ is
  let C = *scan-subclass*$(g, k, axis)$
  function *coord*(s) = *extract-news-coordinate*$(g, axis, s)$
  case (*smode*) of
    (:none) :
      return C
    (:segment-bit) :
      let $Q = \{ m \mid m \in hyperplane(g, k, \{ axis \}) \wedge (sbit[m] = 1) \}$
      return $\{ m \mid m \in C \wedge \neg\exists j : (j \in Q \wedge coord(m) < coord(j) \leq coord(k)) \}$
    (:start-bit) :
      let $Q = \{ m \mid m \in hyperplane(g, k, \{ axis \}) \wedge (sbit[m] = 1) \}$
      case (*direction*) of
        (:upward) :
          return $\{ m \mid m \in C \wedge \neg\exists j : (j \in (C \cap Q) \wedge coord(m) < coord(j) \leq coord(k)) \}$
        (:downward) :
          return $\{ m \mid m \in C \wedge \neg\exists j : (j \in (C \cap Q) \wedge coord(k) \leq coord(j) < coord(m)) \}$

function *scan-subset*$(g, k, axis, direction, inclusion, smode, sbit)$ is
  let S = *scan-set*$(g, k, axis, direction, smode, sbit)$
  function *coord*(s) = *extract-news-coordinate*$(g, axis, s)$
  case (*direction*, *inclusion*) of
    (:upward, :exclusive) : return $\{ m \mid m \in S \wedge coord(m) < coord(k) \}$
    (:upward, :inclusive) : return $\{ m \mid m \in S \wedge coord(m) \leq coord(k) \}$
    (:downward, :exclusive) : return $\{ m \mid m \in S \wedge coord(m) > coord(k) \}$
    (:downward, :inclusive) : return $\{ m \mid m \in S \wedge coord(m) \geq coord(k) \}$

A spread operation is like a scan, except that rather than producing "intermediate" or "running" results by using scan sets, every processor gets the result of combining the values from every active processor in the scan class.

A reduce operation is like a spread, except that instead of storing the result in every active processor in the scan class, it stores the result into only one specified processor of the scan class.

A multispread operation is like a spread, but allows hyperplanes of any rank, not just of rank 1, to serve as the scan classes. In this manner, for example, a single value within each hyperplane can be replicated throughout its hyperplane.

The following table shows the results computed for various operand combinations for a scan with unsigned addition over a set of values all of which are 1.

`scan-with-u-add`		*context-flag*	1 1 1 1 0 0 0 0 1 1 0 0 1 1 1 0
		sbit	0 0 1 0 0 0 1 0 0 0 0 0 0 1 0 0
		source	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
direction	*inclusion*	*smode*	
:upward	:exclusive	:none	0 1 2 3 4 5 6 7 8 →
:downward	:exclusive	:none	8 7 6 5 4 3 2 1 0 ←
:upward	:inclusive	:none	1 2 3 4 5 6 7 8 9 →
:downward	:inclusive	:none	9 8 7 6 5 4 3 2 1 ←
:upward	:exclusive	:segment-bit	0 1 0 1 → 0 1 2 0 1 →
:downward	:exclusive	:segment-bit	1 0 1 0 ← 2 1 0 1 0 ←
:upward	:inclusive	:segment-bit	1 2 1 2 → 1 2 3 1 2 →
:downward	:inclusive	:segment-bit	2 1 2 1 ← 3 2 1 2 1 ←
:upward	:exclusive	:start-bit	0 1 2 1 2 3 4 5 1 →
:downward	:exclusive	:start-bit	2 1 5 4 3 2 1 1 0 ←
:upward	:inclusive	:start-bit	1 2 1 2 3 4 5 1 2 →
:downward	:inclusive	:start-bit	3 2 1 5 4 3 2 1 1 ←

5.21 Global Reduction Operations

A global operation combines a number of values in much the same manner as a scan or reduce operation, but delivers the result to the front end rather than storing it in a processor field.

CM:global { -logand | -logior | -logxor | -c-add | { -s- | -u- | -f- } { add | min | max } | u-max { -s- | -u- } -intlen } -1L

All the usual combining operations are provided. In addition, the compound operation max-intlen is provided for efficiency; it is much faster than than a separate integer-length operation followed by a global-max operation.

5.22 Memory Data Transfers

These operations simply transfer data between a field in the processor array and the front end.

CM: { s- / u- / f- } { read-from / write-to } { -processor / -news-array } -1L

CM:c- { read-from / write-to } -processor -1L

The operations read-from-processor and write-to-processor each transfer a single datum (integer or floating-point).

The operations read-from-news-array and write-to-news-array can transfer entire arrays or subarrays. Their implementation is optimized for relatively high throughput.

5.23 Sorting

Paris provides operations for sorting data based on integer or floating-point keys.

CM: { f- / s- / u- } rank-2-L

The rank operation does not actually put records into sorted order. Instead, it produces ranking information from which appropriate send addresses can be calculated; a send operation can then be used to put the records in order. This allows the ranking operation to deal only with sort keys and not with entire records.

5.24 Timing Paris Code

A set of instructions beginning with CM:timer- provide a timing facility with microsecond precision.

CM:timer- { clear / start / stop / print / read-starts / read-elapsed / read-cm-busy / read-cm-idle / read-run-state / set-starts }

From the Lisp/Paris interface, this timing facility is incorporated in the macro CM:time, which may be wrapped around code in order to time it.

5.25 The LEDS

One of the most attractive features of a Connection Machine system is the array of blinking lights on the faces of its cabinet. The following operation specifies whether the lights are to be blinked automatically, or turned on and off under user program control.

CM:set-system-leds-mode

These operations turn lights on and off according to the contents of a one-bit data field.

CM:latch-leds { ~ | -always }

5.26 Front End Operations

Programs that use Paris operations frequently need to perform certain calculations on the front end that are not easily expressed in the host programming language. These operations are provided as part of the Paris library interface; they deal primarily with Gray codes and NEWS coordinates.

CM:fe- { from-gray-code | to-gray-code | extract-news-coordinate | extract-multi-coordinate | deposit-news-coordinate | make-news-coordinate }

5.27 Environmental Interface

These operations pertain to allocating, deallocating, initializing, and debugging the Connection Machine.

CM:
- attach
- attached
- cold-boot
- detach
- init
- power-up
- reset-timer
- set-safety-mode
- start-timer
- stop-timer
- time
- warm-boot

The attach operation is used to attach the front end process to a specified portion of all Connection Machine processors.

The attached operation returns true if the front end process actually has Connection Machine processors attached for use.

The cold-boot operation is used to initialize the Connection Machine hardware allocated to the executing front end.

The detach operation frees attached Connection Machine processors from the currect front end process.

The init operation is used by the C/Paris and Fortran/Paris interfaces to initialize the Connection Machine hardware.

The power-up operation resets the Nexus, causing all front-end computers to become logically detached from the Connection Machine system.

The set-safety-mode operation allows the user to specify the level of run-time error checking to be performed by the Paris interface.

The time family of operations are used to measure both the execution and the elapsed time taken by other operations.

The warm-boot operation is used by the Lisp/Paris interface to reinitialize the Connection Machine system without disturbing user memory.

Chapter 6

The C/Paris Interface

Paris is used as a set of variables, subroutines, and macros within a program that may be written in any one of a number of languages. This chapter explains how to call Paris instructions from C programs.

6.1 C/Paris Header Files

Type specification statements required for programs that access the C/Paris interface are given in the header file named

`/usr/include/cm/paris.h`

This header file contains four kinds of declarations that provide an environment for calling Paris instructions from C.

- Type declarations define new data types (struct types, for example) needed for communication with certain Paris operations.
- Function declarations define the result types of all C/Paris function subprograms.
- Variable declarations define configuration variables that provide access to the state of the Connection Machine system.
- #define statements define symbolic numeric constants to be used as arguments to certain C/Paris subprogram calls.

These declarations are discussed in more detail in the following sections.

6.2 C/Paris Instruction Names and Argument Types

This section describes how to call these instructions from C and what types of arguments to pass them.

The instruction names and other names that appear in this document are spelled in a form acceptable to Lisp (an arbitrary choice in order to have *some* common denominator for the dictionary). Each name is easily converted to the corresponding C name using the following two-part rule:

- If the Lisp name begins with a colon, add "CM" to the front.
- Drop all asterisks, and convert all colons and hyphens to underscores.

This usually results in a name written in mixed case (some letters uppercase and some lowercase). The name must be written in exactly that way, for C identifiers are case-sensitive. (Although Lisp is *not* case-sensitive, all identifiers appearing in Lisp form in this document are written in mixed case so as to produce the correct C name after applying the conversion rules.)

Chapter 9 describes each of the Paris instructions in terms of its arguments, its effect on operand fields residing in Connection Machine memory, and the result (if any) that it returns to the front end. The same argument name is often used in several different instruction definitions, but arguments with the same name always have the same type (as viewed by the front-end C program). For example, *dest* is used throughout to represent the field ID of a destination field; the field itself may be a floating-point or an integer field, the width of which is specified by other arguments to the instruction, but to the C program the argument is always simply a field ID.

Following is a brief description of the major classes of arguments that can be passed to subprograms of the C/Paris interface.

6.2.1 Id Types

These are values that should be treated as abstract entities, or "black boxes." They are created using special Paris instructions, and their actual values have no significance to the calling C program; they are simply tokens that may be passed to other Paris routines.

VP set ID

A value representing a virtual processor set. Its C type is CM_vp_set_id_t.

geometry ID

A value representing a geometry with a particular shape. Its C type is CM_geometry_id_t.

field ID

A value representing a field allocated on the CM. Its C type is CM_field_id_t.

6.2.2 Operand Field Addresses

Most Paris operations require one or more field IDs to indicate one or more regions of Connection Machine memory to be processed. Such field IDs are obtained from memory allocation calls. Their C type is CM_field_id_t.

dest, *source*, *source1*, *source2*

These field IDs specify fields to be used as source or destination operands of an instruction.

send-address

This argument specifies a field that itself contains, within each processor, the send address of a processor (possibly the same one, possibly another).

news-coordinate

This argument specifies a field that itself contains, within each processor, the NEWS coordinate of a processor (possibly the same one, possibly another).

notify

A field ID for a 1-bit field to hold a result indicating receipt of a message by a send instruction.

sbit

A field ID for a 1-bit field that indicates how Paris scan operations should divide processors into logical groups.

6.2.3 Immediate Operands

These arguments are scalar values that participate in Paris operations as if they were first copied to every Connection Machine processor and then operated upon as if a field ID had been supplied. Paris operations that take "immediate" operand values of this sort usually have "constant" or "const" in their names.

source-value, source2-value

A (front-end) value or variable to be supplied as input to an instruction on the CM. The type of value passed depends on the instruction to which it is passed. The C type of such an immediate operand is long for a signed integer value, unsigned long for a signed integer value, or double for a floating-point value.

send-address-value

An integer, the send address of a single particular processor. The C type of such an immediate operand is CM_sendaddr_t.

news-coordinate-value An integer, the NEWS coordinate of a single particular processor. The C type of such an immediate operand is unsigned long.

6.2.4 Operand Field Lengths

These are integer values that specify the widths of source and destination operand fields on the CM. Their C type is unsigned.

len, slen, slen1, slen2, dlen

An integer value designating the length (in bits) of a source field that will be treated by the operation as a bit field, a signed integer, or an unsigned integer. It is not unusual for this value to be 32 to match the size of C long variables on the front end, but other lengths may be used as well—longer ones for additional precision, shorter ones for improved speed.

s, *ds*, *ss*

An integer value designating the significand length of a floating-point field. For single-precision (C type float) fields, this value should be 23; for double-precision (C type double) fields, the value should be 52.

e, *de*, *se*

An integer value designating the exponent length of a floating-point field. For single-precision (C type float) fields, this value should be 8; for double-precision (C type double) fields, the value should be 11.

6.2.5 Miscellaneous Signed and Unsigned Values

Both signed and unsigned Paris quantities are represented in C by variables and values whose C type is unsigned long. These are variously referred to, depending on their roles within particular operations, under the following names:
offset, *axis*, *axis-length*, *coordinate*, *rank*, *multi-coordinate*

6.2.6 Bit Sets and Masks

Arguments representing sets taken from universes of up to 31 elements are represented as integer values, where the bit whose value is 2^j is 1 to indicate that element j is in the set. Their C type is unsigned long.

At present, the only universe of interest in Paris is *axis-mask*, the set of axes for a given geometry.

6.2.7 Vectors of Integers

These arguments should be represented as C one-dimensional arrays whose elements are of C type unsigned. The maximum size of these vectors is 31.
axis-vector, *start-vector*, *offset-vector*, *end-vector*, *dimension-vector*

6.2.8 Multi-dimensional Front-end Arrays

Multi-dimensional front-end arrays of any C integer or floating-point type can be transferred to and from CM memory using a single instruction (see section 5.22).

front-end-array A pointer to a front-end array is passed simply by mentioning the name of the array.

6.2.9 Symbolic Values

The symbolic constants defined in #define statements in the C/Paris header file should be used when supplying values for these arguments:

direction

One of the values CM_upward or CM_downward, indicating the direction of a scan, NEWS, or other instruction.

inclusion

One of the values CM_exclusive or CM_inclusive, indicating the boundaries of a scan instruction.

smode

One of the values CM_none, CM_start_bit, or CM_segment_bit, indicating how a scan operation is to be partitioned.

There are other symbolic values as well, but these are the most important. All names are formed by the standard rule: starting from a Lisp name such as :start-bit, add "CM" to the front and then convert colons and hyphens to underscores, yielding CM_start_bit.

6.3 C/Paris Configuration Variables

The configuration variables provide access to information about the configuration of the Connection Machine system. See section 3.7 for a list. The C/Paris interface makes these variables accessible through variables declared in the C/Paris header file. They are initialized in an application program by a call to the subroutine CM_init and should not be changed by an application program.

Each configuration variable is a numeric value that is constant over the course of a session (from one cold boot operation to the next), or varies from one Connection Machine configuration to another. For example, CM_physical_processors_limit is a value that depends upon the size of the Connection Machine to which the application is attached.

Numeric values that are constant for a given release of the CM System Software are given in #define statements.

6.4 Calling Paris from C

This section describes how to build C programs that access the Paris instruction set using the C/Paris interface. Such programs must manage the dynamic allocation and deallocation of Connection Machine fields directly. This section describes the form of C main programs and subprograms that call the C/Paris interface, as well as the steps involved in compiling and linking such programs.

The following code fragment illustrates the structure of a C main program that calls Paris instructions.

```
#include <cm/paris.h>
:
main() {
  CM_init();
  :
  CM_paris_instruction(...);
  :
  if ( CM_configuration_variable > limit ) ...
```

```
  ⋮
}
```

Note that the call to CM_init is required prior to any other calls to Paris instructions.

The following code fragment illustrates the structure of a C subroutine subprogram that calls Paris instructions.

```
#include <cm/paris.h>
⋮
float test() {
  ⋮
  CM_paris_instruction(...);
  ⋮
  if ( CM_configuration_variable > limit ) ...
  ⋮
}
```

It looks exactly like a main program in its use of Paris, *except* that a subprogram should not call CM_init.

Use the following command to compile and link these program units:

```
% cc main.c test.c -lparis -lm
```

Note that there should be no space between the -l option and its argument.

Chapter 7

The Fortran/Paris Interface

Paris is used as a set of variables and subroutines within a program that may be written in any one of a number of languages. This chapter explains how to call Paris instructions from Fortran programs, especially those compiled by VAX Fortran and Sun Fortran.

The Fortran/Paris interface is itself an interface to C/Paris (see chapter 6).

7.1 Fortran/Paris Header Files

Type specification statements required for programs that access the Fortran/Paris interface are given in the header file named

`/usr/include/cm/paris-configuration-fort.h`

This header file contains three kinds of declarations that provide an environment for calling Paris instructions from Fortran.

- Type specification statements define the result types of all Fortran/Paris function subprograms.
- A declaration of a common block named cmval defines configuration variables that provide access to the state of the Connection Machine system.
- PARAMETER statements define symbolic numeric constants to be used as arguments to certain Fortran/Paris subprogram calls.

These declarations are discussed in more detail in the following sections.

7.2 Fortran/Paris Instruction Names and Argument Types

This section describes how to call these instructions from Fortran and what types of arguments to pass them.

The instruction names and other names that appear in this document are spelled in a form acceptable to Lisp (an arbitrary choice in order to have *some* common denominator for the dictionary). Each name is easily converted to the corresponding Fortran name using the following two-part rule:

- If the Lisp name begins with a colon, add "CM" to the front.
- Drop all asterisks, and convert all colons and hyphens to underscores.

It is also permissible to convert names to entirely uppercase letters if desired, as Fortran identifiers are not case-sensitive.

Chapter 9 describes each of the Paris instructions in terms of its arguments, its effect on operand fields residing in Connection Machine memory, and the result (if any) that it returns to the front end. The same argument name is often used in several different instruction definitions, but arguments with the same name always have the same type (as viewed by the front-end Fortran program). For example, *dest* is used throughout to Represent the field ID of a destination field; the field itself may be a floating-point or an integer field, the width of which is specified by other arguments to the instruction, but to the Fortran program the argument is always simply a field ID.

Following is a brief description of the major classes of arguments that can be passed to subprograms of the Fortran/Paris interface.

7.2.1 Id Types

These are integer values that should be treated as abstract entities, or "black boxes." They are created using special Paris instructions, and their actual values have no significance to the calling Fortran program; they are simply tokens that may be passed to other Paris routines. Their Fortran type is INTEGER.

VP set ID

An integer value representing a virtual processor set.

geometry ID

An integer value representing a geometry with a particular shape.

field ID

An integer value representing a field allocated on the CM.

7.2.2 Operand Field Addresses

Most Paris operations require one or more field IDs to indicate one or more regions of Connection Machine memory to be processed. Such field IDs are obtained from memory allocation calls. Their Fortran type is INTEGER.

dest, *source*, *source1*, *source2*

These field IDs specify fields to be used as source or destination operands of an instruction.

send-address

This argument specifies a field that itself contains, within each processor, the send address of a processor (possibly the same one, possibly another).

news-coordinate

This argument specifies a field that itself contains, within each processor, the NEWS coordinate of a processor (possibly the same one, possibly another).

notify

A field ID for a 1-bit field to hold a result indicating receipt of a message by a send instruction.

sbit

A field ID for a 1-bit field that indicates how Paris scan operations should divide processors into logical groups.

7.2.3 Immediate Operands

These arguments are scalar values that participate in Paris operations as if they were first copied to every Connection Machine processor and then operated upon as if a field ID had been supplied. Paris operations that take "immediate" operand values of this sort usually have "constant" or "const" in their names.

The Fortran type of such an immediate operand must be INTEGER for an integer value, and DOUBLE PRECISION for a floating-point value.

source-value, source2-value

A (front-end) value or variable to be supplied as input to an instruction on the CM. The type of value passed depends on the instruction to which it is passed.

send-address-value

An integer, the send address of a single particular processor.

news-coordinate-value An integer, the NEWS coordinate of a single particular processor.

7.2.4 Operand Field Lengths

These are integer values that specify the widths of source and destination operand fields on the CM. Their Fortran type is INTEGER.

len, slen, slen1, slen2, dlen

An integer value designating the length (in bits) of a source field that will be treated by the operation as a bit field, a signed integer, or an unsigned integer. It is not unusual for this value to be 32 to match the size of Fortran INTEGER variables on the front end, but other lengths may be used as well—longer ones for additional precision, shorter ones for improved speed.

s, ds, ss

An integer value designating the significand length of a floating-point field. For single-precision (Fortran type REAL) fields, this value should be 23; for double-precision (Fortran type DOUBLE PRECISION) fields, the value should be 52.

e, *de*, *se*

An integer value designating the exponent length of a floating-point field. For single-precision (Fortran type REAL) fields, this value should be 8; for double-precision (Fortran type DOUBLE PRECISION) fields, the value should be 11.

7.2.5 Miscellaneous Signed and Unsigned Values

Both signed and unsigned Paris quantities are represented in Fortran by variables and values whose Fortran type is INTEGER. These are variously referred to, depending on their roles within particular operations, under the following names:
offset, *axis*, *axis-length*, *coordinate*, *rank*, *multi-coordinate*

7.2.6 Bit Sets and Masks

Arguments representing sets taken from universes of up to 31 elements are represented as integer values, where the bit whose value is 2^j is 1 to indicate that element j is in the set. Their Fortran type is INTEGER.

At present, the only universe of interest in Paris is *axis-mask*, the set of axes for a given geometry.

7.2.7 Vectors of Integers

These arguments should be represented as Fortran one-dimensional INTEGER arrays. The maximum size of these vectors is 31.
axis-vector, *start-vector*, *offset-vector*, *end-vector*, *dimension-vector*

7.2.8 Multi-dimensional Front-end Arrays

Multi-dimensional front-end arrays of Fortran type LOGICAL, INTEGER, REAL, or DOUBLE PRECISION can be transferred to and from CM memory using a single instruction (see section 5.22).

front-end-array

Such an array is passed simply by mentioning the name of the array.

7.2.9 Symbolic Values

The symbolic constants defined in PARAMETER statements in the Fortran/Paris header file should be used when supplying values for these arguments:

direction

One of the values CM_upward or CM_downward, indicating the direction of a scan, NEWS, or other instruction.

inclusion

One of the values CM_exclusive or CM_inclusive, indicating the boundaries of a scan instruction.

smode

One of the values CM_none, CM_start_bit, or CM_segment_bit, indicating how a scan operation is to be partitioned.

There are other symbolic values as well, but these are the most important. All names are formed by the standard rule: starting from a Lisp name such as :start-bit, add "CM" to the front and then convert colons and hyphens to underscores, yielding CM_start_bit.

7.3 Fortran/Paris Configuration Variables

The configuration variables provide access to information about the configuration of the Connection Machine system. See section 3.7 for a list. The Fortran/Paris interface makes these variables accessible through variables declared in the common block named cmval, defined by the Fortran/Paris header file. They are initialized in an application program by a call to the subroutine CM_init and should not be changed by an application program.

Each configuration variable is a numeric value that is constant over the course of a session (from one cold boot operation to the next), or varies from one Connection Machine configuration to another. For example, CM_physical_processors_limit is a value that depends upon the size of the Connection Machine to which the application is attached. Most of these configuration variables are declared to be of Fortran type INTEGER.

Numeric values that are constant for a given release of the CM System Software are also given in PARAMETER statements.

7.4 Calling Paris from Fortran

This section describes how to build Fortran programs that access the Paris instruction set using the Fortran/Paris interface. Such programs must manage the dynamic allocation and deallocation of Connection Machine fields directly. This section describes the form of Fortran main programs and subprograms that call the Fortran/Paris interface, as well as the steps involved in compiling and linking such programs.

The following code fragment illustrates the structure of a Fortran main program that calls Paris instructions.

```
      PROGRAM main
C     VAX Fortran or Sun Fortran
      :
      INCLUDE '/usr/include/cm/paris-configuration-fort.h'
      CALL CM_init()
      :
      CALL CM_paris_instruction(...)
      :
      IF ( CM_configuration_variable .GT. limit ) ...
      :
      END
```

Note that the call to CM_init is required prior to any other calls to Paris instructions.

The following code fragment illustrates the structure of a Fortran subroutine subprogram that calls Paris instructions.

```
      SUBROUTINE test
C     VAX Fortran or Sun Fortran
      :
      INCLUDE '/usr/include/cm/paris-configuration-fort.h'
      :
      CALL CM_paris_instruction(...)
      :
      IF ( CM_configuration_variable .GT. limit ) ...
      :
      END
```

It looks exactly like a main program in its use of Paris, *except* that a subprogram should not call CM_init.

Using VAX Fortran, the following command compiles and links these program units to run on the Connection Machine Model 2:

```
% fort main.for test.for -lparisfort -lparis
```

Note that there should be no space between the -l option and its argument.

Using Sun Fortran, the following command compiles and links these program units to run on the Connection Machine Model 2:

```
% f77 main.f test.f -lparisfort -lparis
```

Note that there should be no space between the -l option and its argument.

Chapter 8

The Lisp/Paris Interface

Paris is used as a set of variables, subroutines, and macros within a program that may be written in any one of a number of languages. This chapter explains how to call Paris instructions from Lisp programs.

8.1 Lisp/Paris Instruction Names and Argument Types

This section describes how to call these instructions from Lisp and what types of arguments to pass them.

The instruction names and other names that appear in this document are spelled in a form acceptable to Lisp (an arbitrary choice in order to have *some* common denominator for the dictionary).

Although Lisp is *not* case-sensitive, all identifiers appearing in Lisp form in this document are written in mixed case so as to produce the correct C name after applying certain conversion rules. The Lisp programmer may write names entirely in uppercase letters or entirely lowercase letters, if desired.

Chapter 9 describes each of the Paris instructions in terms of its arguments, its effect on operand fields residing in Connection Machine memory, and the result (if any) that it returns to the front end. The same argument name is often used in several different instruction definitions, but arguments with the same name always have the same type (as viewed by the front-end Lisp program). For example, *dest* is used throughout to represent the field ID of a destination field; the field itself may be a floating-point or an integer field, the width of which is specified by other arguments to the instruction, but to the Lisp program the argument is always simply a field ID.

Following is a brief description of the major classes of arguments that can be passed to subprograms of the Lisp/Paris interface.

8.1.1 Id Types

These are values that should be treated as abstract entities, or "black boxes." They are created using special Paris instructions, and their actual values have no significance to the calling Lisp program; they are simply tokens that may be passed to other Paris routines.

VP **set** ID

An integer value representing a virtual processor set.

geometry ID

A structure of type CM:geometry ID representing a geometry with a particular shape.

field ID

An integer value representing a field allocated on the CM.

8.1.2 Operand Field Addresses

Most Paris operations require one or more field ID's to indicate one or more regions of Connection Machine memory to be processed. Such field ID's are obtained from memory allocation calls. Their Lisp type is integer.

dest, *source*, *source1*, *source2*

These field IDs specify fields to be used as source or destination operands of an instruction.

send-address

This argument specifies a field that itself contains, within each processor, the send address of a processor (possibly the same one, possibly another).

news-coordinate

This argument specifies a field that itself contains, within each processor, the NEWS coordinate of a processor (possibly the same one, possibly another).

notify

A field ID for a 1-bit field to hold a result indicating receipt of a message by a send instruction.

sbit

A field ID for a 1-bit field that indicates how Paris scan operations should divide processors into logical groups.

8.1.3 Immediate Operands

These arguments are scalar values that participate in Paris operations as if they were first copied to every Connection Machine processor and then operated upon as if a field ID had been supplied. Paris operations that take "immediate" operand values of this sort usually have "constant" or "const" in their names.

The Lisp type of such an immediate operand is integer for an integer value, or float for a floating-point value (any of the several kinds of Common Lisp floating-point numbers may be supplied).

source-value, source2-value

A (front-end) value or variable to be supplied as input to an instruction on the CM. The type of value passed depends on the instruction to which it is passed.

send-address-value

An integer, the send address of a single particular processor.

news-coordinate-value An integer, the NEWS coordinate of a single particular processor.

8.1.4 Operand Field Lengths

These are integer values that specify the widths of source and destination operand fields on the CM. Their Lisp type is integer.

len, *slen*, *slen1*, *slen2*, *dlen*

An integer value designating the length (in bits) of a source field that will be treated by the operation as a bit field, a signed integer, or an unsigned integer. It is not unusual for the programmer to choose this value to match the size of Lisp fixnum variables on the front end, but other lengths may be used as well—longer ones for additional precision, shorter ones for improved speed.

s, *ds*, *ss*

An integer value designating the significand length of a floating-point field. Floating-point numbers of any size are supported, but certain values must be used for good performance on the hardware floating-point accelerator. For single-precision (Lisp type single-float) fields, this value should be 23; for double-precision (Lisp type double-float) fields, the value should be 52.

e, *de*, *se*

An integer value designating the exponent length of a floating-point field. Floating-point numbers of any size are supported, but certain values must be used for good performance on the hardware floating-point accelerator. For single-precision (Lisp type single-float) fields, this value should be 8; for double-precision (Lisp type double-float) fields, the value should be 11.

8.1.5 Miscellaneous Signed and Unsigned Values

Both signed and unsigned Paris quantities are represented in Lisp by variables and values whose Lisp type is integer. These are variously referred to, depending on their roles within particular operations, under the following names:
offset, *axis*, *axis-length*, *coordinate*, *rank*, *multi-coordinate*

8.1.6 Bit Sets and Masks

Arguments representing sets taken from universes of up to 31 elements are represented as integer values, where the bit whose value is 2^j is 1 to indicate that element j is in the set. Their Lisp type is integer.

At present, the only universe of interest in Paris is *axis-mask*, the set of axes for a given geometry.

8.1.7 Vectors of Integers

These arguments should be represented as Lisp vectors (one-dimensional arrays); they may be specialized vectors, capable of holding integers only, or general vectors, capable of holding any Lisp objects but into which only integers happen to have been stored. The maximum size of these vectors is 31.

axis-vector, *start-vector*, *offset-vector*, *end-vector*, *dimension-vector*

8.1.8 Multi-dimensional Front-end Arrays

Multi-dimensional front-end arrays, whether specialized or general, can be transferred to and from CM memory using a single instruction (see section 5.22).

front-end-array

Such an array is passed simply by mentioning the name of the array.

8.1.9 Symbolic Values

These symbolic constants should be used when supplying values for these arguments:

direction

One of the values :upward or :downward, indicating the direction of a scan, NEWS, or other instruction.

inclusion

One of the values :exclusive or :inclusive, indicating the boundaries of a scan instruction.

smode

One of the values :none, :start-bit, or :segment-bit, indicating how a scan operation is to be partitioned.

There are other symbolic values as well, but these are the most important.

8.2 Lisp/Paris Configuration Variables

The configuration variables provide access to information about the configuration of the Connection Machine system. See section 3.7 for a list. The Lisp/Paris interface makes these variables available. They are initialized in an application program by a call to subroutine CM:cold-boot and should not be changed by an application program.

Each configuration variable is a numeric value that is constant over the course of a session (from one cold boot operation to the next), or varies from one Connection Machine configuration to another. For example, CM:*pysical-processors-limit* is a value that depends upon the size of the Connection Machine to which the application is attached.

8.3 Calling Paris from Lisp

This section describes how to build Lisp programs that access the Paris instruction set using the Lisp/Paris interface. Such programs must manage the dynamic allocation and deallocation of Connection Machine fields directly. This section describes the form of Lisp main programs and subprograms that call the Lisp/Paris interface, as well as the steps involved in compiling and linking such programs.

The following code fragment illustrates the structure of a Lisp function program that calls Paris instructions.

```
(defun test (...)
  :
  (CM:paris-instruction ...)
  :
  (if (> CM:configuration-variable limit) ...)
  :
  )
```

Remember that CM:cold-boot should be called *once* before beginning a computation that uses Paris; it is not appropriate to call CM:cold-boot on entrance to every function.