VCODE Reference Manual (Version 1.1)

Guy E. Blelloch   Siddhartha Chatterjee   Fritz Knabe
Jay Sipelstein   Marco Zagha

July 1990

CMU-CS-90-146 2

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

A shorter version of this report will appear in the *Third Symposium on the Frontiers of Massively Parallel Computation*, October 8–10, 1990, University of Maryland, College Park, MD.

Abstract

This report introduces VCODE, an intermediate language for data-parallel computations. VCODE is designed to allow easy porting of data-parallel languages, such as C*, PARALATION LISP, and Fortran 8x, to a wide class of parallel machines. It is designed with the joint goals of being simple, expressive, and efficiently implementable. It contains about 50 instructions, most of which manipulate arbitrarily long vectors of atomic values, and includes a set of segmented instructions that are crucial for implementing data-parallel languages that permit nested parallelism, such as PARALATION LISP and CM-Lisp. The report outlines the VCODE language, discusses many of the design issues, illustrates how data-parallel languages can be mapped onto it, and describes how it can be implemented on massively parallel machines. A complete definition of VCODE is given in the appendix.

This research was supported by the Defense Advanced Research Projects Agency (DOD) and monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-87-C-1499, ARPA Order No. 4976, Amendment 20.

510.7808
[illegible]
[illegible]

Keywords: Portable parallel programming, data parallelism, intermediate language, vector models, segmented operations

1 Introduction

Historically, data-parallel languages were associated with SIMD machines, and the erroneous notion of data parallelism being useful for only a small class of structured applications led to its virtual exclusion as a programming model for MIMD machines. More recently, however, researchers have demonstrated that data-parallel languages are useful for applications as diverse as parsing [22], DNA sequence comparison [7, 14], object recognition [26], VLSI design [8, 6, 13] and computer graphics [5], and most of the parallel algorithms found in the literature for grids, hypercubes and Parallel-RAM models are either data-parallel or easily converted into such a form. Furthermore, the semantic simplicity of data-parallel languages, and their ability to easily express large amounts of parallelism, make them scalable, easy to understand, and easy to debug. For these reasons, it has been argued that for many applications data-parallel languages should be the languages of choice for all parallel machines, whether SIMD or MIMD [2, 20, 18]. In fact, some of the languages recently developed for MIMD machines, such as AL [25] (for the WARP machine), Apply [9], and the Fortran 8x vector extensions [1] (which are being implemented on several MIMD machines), are data-parallel.

This recent attempt to port data-parallel languages to a variety of architectures has raised a large range of intriguing compiler issues. Among these issues are the automatic layout of data on distributed memory machines [12], identification and removal of unnecessary synchronization on MIMD machines, work distribution, load balancing, flattening nested parallelism [3], and many issues that appear in compilers for vector machines, such as loop fusion [27]. In the long run, such global compiler issues are likely to be the dominant factor contributing to the performance of parallel machines. At present the compiler work on these issues is sparse.

VCODE was designed as a testbed for a systematic study of the compiler issues that arise in data-parallel languages. VCODE is a small, simple language that could ultimately serve as a portable intermediate representation for high-level data-parallel languages. We are currently implementing compilers that map VCODE onto a variety of parallel machines, and studying the effects of architectural features like memory organization (shared/private, hierarchical/flat), control structure (SIMD/MIMD, vector hardware), and interconnection topology (bus, grid, hypercube) on the structure and functionality of the compilers (see Figure 1). A large part of the research effort is to determine what components of the compilers are common among the machines, and to structure a common compiler to take advantage of this.

We chose not to use any of the existing data-parallel languages as they either lack features we plan to study, or are too cumbersome. For example, we could not use C* [19] or Fortran 8x because they do not support nested parallelism—the ability to define a parallel routine and then use it multiple times in parallel nested inside another parallel routine. On the other hand, we did not want to use CM-Lisp [23] or PARALATION LISP [20], which support nested parallelism, because it is unclear that the primitives they provide can be implemented efficiently, and the languages are large enough that it would take many man-years of work to implement a full compiler. VCODE was designed with three principles in mind:

1. **simplicity:** a small number of instructions with a simple syntax and semantics.
2. **efficiency:** each instruction can be implemented efficiently.
3. **expressiveness:** capable of easily expressing the features of existing data-parallel languages.

Currently, we have interpreters for VCODE on the Cray Y-MP, the Encore Multimax, and the Thinking Machines Corporation Connection Machine, and are working on an implementation for the Carnegie Mellon/Intel iWarp [4]. These share a common front end that handles program control and memory management, and specialized back ends that implement the VCODE primitives through a common library interface. Work is also in progress on compilers for these machines. We have used the initial implementation

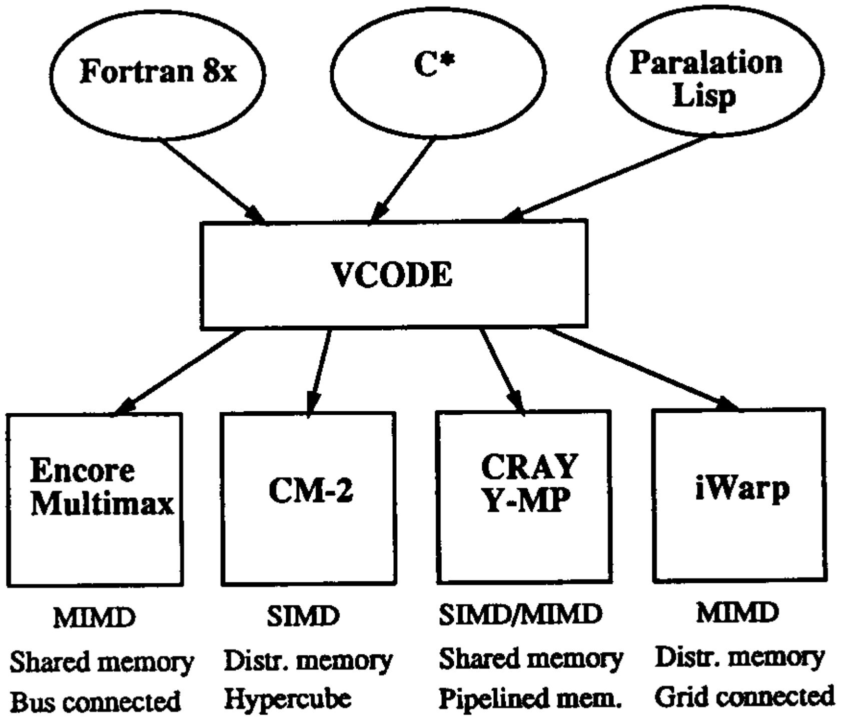


Figure 1: Using VCODE to map a variety of data-parallel languages onto a variety of parallel machines.

in a graduate course on parallel algorithms and have brought up many kernel algorithms including a set of tree, graph, sparse matrix and string operations.

This paper describes VCODE (Section 2), discusses and justifies many of the design decisions involved in the definition of VCODE (Section 3), demonstrates how other data-parallel languages may be translated into VCODE (Section 4), and discusses implementation issues on various machines (Section 5). A reference manual for VCODE is given in Appendix A.

2 VCODE

VCODE can be summarized as follows:

- It is based on a stack and heap model and resembles PCODE [17], a serial intermediate language designed for PASCAL.
- It contains a small set of instructions, most of which operate on one-dimensional vectors of atomic values. The remainder are for control and memory management.
    - It contains instructions for rearranging the elements of vectors. These can be used for vector-based indexing in C*, Fortran 8x and APL [11], for the β operation in CM-Lisp, and for the *move* operation in PARALATION LISP.
    - It has instructions that can operate on segmented vectors. This permits the implementation of the nested parallelism allowed in CM-Lisp, SETL [21] and PARALATION LISP.
- It can dynamically create and destroy arbitrarily long vectors. This is necessary for all the above languages.
- Unlike machine-oriented languages, it provides no notion of processors, either physical or virtual. This allows the interpreter or compiler to choose appropriate strategies for process mapping and memory management depending on the target machine.

VCODE is conveniently defined in terms of a machine architecture, the vector stack machine. This is a standard stack machine with the addition of a heap, except that each memory location, both stack and heap, contains an arbitrarily long vector of atomic values. Each vector has a length associated with it, and vectors in different memory locations can have different lengths. Instructions operate on whole vectors at a time. For example, the + instruction pops the top two vectors (of equal length and appropriate type) from the stack, adds corresponding elements together, and returns the result vector to the top of the stack.

There are four basic types in VCODE: boolean vectors, integer vectors, floating-point vectors and segment descriptor vectors. All vectors are homogeneous. The first three types are self-explanatory, and correspond to data vectors. The segment descriptor is a vector that specifies a partitioning of one or more data vector into *segments*. Various possible concrete representations are possible for this type, such as an integer vector containing the lengths of the segments, an integer vector containing the starting points of the segments, or a boolean vector set to T at segment starting points and to F elsewhere. In this paper we will use the lengths representation. Thus, the two vectors:

$$A = [\mathrm{s}\quad \mathrm{y}\quad \mathrm{l}\quad \mathrm{l}\quad \mathrm{a}\quad \mathrm{b}\quad \mathrm{l}\quad \mathrm{e}\quad \mathrm{s}]$$

$$L = [3\quad 2\quad 4]$$

(where A is the data vector and L the segment descriptor in lengths form) would represent the following partition:

```
A' = [s y l] [l a] [b l e s].
```

VCODE instructions fall into two classes, *vector instructions* and *memory/control instructions*. The vector instructions operate on a fixed number of vectors from the top of the stack and return their result to the top of the stack. The memory/control instructions are used for moving vectors to and from the heap, for stack manipulations, and for program control. Figure 2 lists some of the VCODE instructions.

The vector instructions can be further divided into the following subclasses:

- Elementwise instructions: These are vector versions of arithmetic and logical operations. There is also a `select` instruction that selects from one or the other of two data vectors based on a boolean vector of flags.
- Scan instructions: These take a source vector and a segment descriptor and return a vector whose elements are the "sum" (with respect to some binary associative operator) of all previous elements in the source vector, within each segment. The operators available are +, **max**, **min**, **AND** and **OR**.
- Permute instructions: In their simplest form, these rearrange the elements of a vector according to another vector of indices. A segment descriptor is also needed to describe the partitioning of the data and index vectors. There are also more complex instructions in this class that allow indices to be selectively masked, and a default vector to be supplied for the result. The unmasked indices within each segment must form a permutation.
- Segment descriptor manipulation instructions: These instructions create and manipulate segment descriptors and change between various concrete representations.
- I/O instructions: These instructions read and write vectors.

Figure 3 illustrates the effects of several of the instructions.

VCODE provides a function declaration and call facility. The stack is used to pass parameters and return results. Program control flow is directed by an if-then-else form and recursive function calls. The if-then-else form has the restriction that both branches must leave the stack in the same state with respect to depth and data types. All instructions except the if-then-else form are strict, *i.e.*, all the inputs must be available before the instruction can be executed.

We consider VCODE to be an intermediate language rather than a high-level one because it lacks many features associated with high-level languages: user-defined types, record types, overloaded operators, automatic coercion, richer control structures, block structure, and so on. It does, however, have the mechanisms to implement such features; this is essential for mapping high-level languages into VCODE.

We now give two examples of VCODE. The first example computes a (segmented) dot product of two vectors. The arguments on the stack are the integer vectors `V1` and `V2`, and the segment descriptor `S`. The function is as follows:

```
FUNC DOTPRODUCT
COPY 2 1       % copy V1 and V2
* INT          % elementwise *
COPY 1 1       % copy S
CALL +_REDUCE  % segmented sum
POP 3 1        % pop arguments
RET
```

The second example is slightly more complicated. It represents a `pack` operation that given a vector `V` of values, another vector `F` of flags, and a segment descriptor `S` (that describes both `V` and `F`), returns a vector containing only the values from the positions where the flags vector is 1.

Memory/Control Instructions:

- Memory Instructions:
  - `copy, pop, load, store, const`
- Control Instructions:
  - `if-then-else, call, ret`

Vector Instructions:

- Elementwise Instructions:
  - `negate, +, *, =, >, and, not, select`
- Permute Instructions:
  - `permute, spermute, bpermute, dist`
- Scan Instructions:
  - `+-scan, max-scan, or-scan`
- Segment Descriptor Manipulation Instructions:
  - `length, segdes`

Figure 2: A partial list of VCODE instructions. All the vector instructions take values off the top of the stack and return their result to the top of the stack.

```
          [5   1   3   4   3   9   2   6]
          [2   5   3   8   1   3   6   2]
   +
          [7   6   6  12   4  12   8   8]
--------------------------------------------
                [l   i   f   e   p   o   t]
                [2   1   0   3   2   1   0]
                [4   3]
   permute
                [f   i   l   e   t   o   p]
--------------------------------------------
             [1   3   5   7   11   13   15]
             [4   3]
  +-scan
             [0   1   4   9    0   11   24]
```

Figure 3: Some vector instructions. Vectors are shown between square brackets. The + instruction takes the top two vectors off the stack, adds them elementwise and returns the result to the top of the stack. The two source vectors must be of the same length. The `permute` instruction rearranges the elements of a vector according to a second vector of indices. The instruction requires a segment descriptor that partitions the vectors—in the example the vectors are partitioned into the first four elements "life" and the remaining elements "pot". The `+-scan` instruction takes a source vector and a segment descriptor and returns a vector whose elements are the sum of all previous elements in the source vector, within each segment.

V	[l	i	f	e	p	o	t]
F	[0	1	1	0	0	0	1]
S	[4	3]					
+_SCAN	[0	0	1	2	0	0	0]
+_REDUCE	[2	1]					
SPERMUTE	[i	f	t]				

Figure 4: An example of the PACK operation on two segments. The +_REDUCE sums the number of 1 flags in each segment to determine the length of the result segments. The SPERMUTE instruction takes five arguments from the top of stack: a vector of values (V), a vector of indices (the result of the +_SCAN), a vector of flags (F) that masks out the values, a vector of lengths of the result segments (the result of the +_REDUCE), and the segment descriptor (S). The result of the +_REDUCE can also be used as a segment descriptor for the final result.

```
FUNC PACK
COPY 1 2       % copy V
COPY 2 1       % copy F and S
+_SCAN INT     % make indices
COPY 1 3       % copy F
COPY 2 3       % copy F and S
CALL +_REDUCE  % find lengths
COPY 1 4       % copy S
               % permute V to indices
SPERMUTE INT   % generated by the +_SCAN
POP 3 1        % pop arguments
RET
```

This works by generating sequential indices for positions where the flags vector is 1, and then permuting the values in V to those indices (see Figure 4).

In addition to the stack code representation, VCODE can be represented as a program graph, or in a LISP-like syntax. For example, the LISP-like syntax for the PACK operation is:

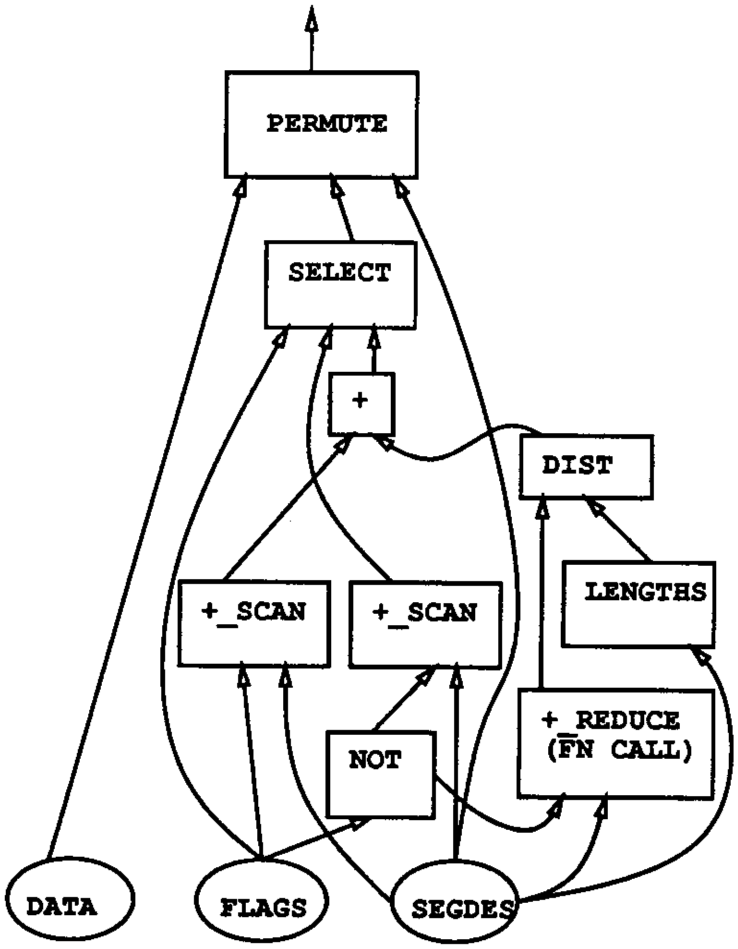
```
(def pack (v f s)
  (spermute v (+_scan f s)
    f (+_reduce f s) s))
```

Each representation is useful in a different context. The stack code is used in the interpreter; the compiler uses the graph representation internally; and the LISP-like syntax is convenient for purposes of readability. We will use the LISP-like syntax for the remainder of this paper. All three representations are shown for a simple piece of VCODE in Figure 5.

3 Design Decisions

This section discusses many of the decisions that went into the design of VCODE. We discuss design alternatives and explain our choices wherever relevant. We divide the discussion into two parts: the basic

```
FUNC SPLIT_25
COPY 1 1
NOT BOOL
COPY 1 0
COPY 1 2
+_SCAN INT
COPY 1 1
COPY 1 3
CALL PLUS_REDUCE
COPY 1 3
LENGTHS
DIST INT
COPY 2 3
+_SCAN INT
+ INT
COPY 1 4
COPY 1 1
COPY 1 3
SELECT INT
COPY 1 6
COPY 1 1
COPY 1 6
PERMUTE INT
POP 7 1
RET
```



```
(def split_25 (data flags segdes)
  (let* ((notflags (not flags))
         (IDown (+_scan notflags segdes))
         (IUp (+ (+_scan flags)
                 (dist (+_reduce notflags segdes)
                       (lengths segdes)))))
    (permute data (select flags IDown IUp) segdes)))
```

Figure 5: Multiple representations for the split operation.

model and the primitives. The basic model includes decisions concerning the primitive data types, the memory model, the control structure, and support for nested parallelism. The discussion of the primitives explains why particular instructions were selected.

As mentioned in Section 1, the main goals in the design of VCODE were simplicity, efficiency and expressiveness. There are many features that appear in existing languages that are absent in VCODE, such as processor context, heterogeneous vectors and arrays. In all these cases, the features can be implemented on top of VCODE without much loss of efficiency.

3.1 Basic Model

The basic model of VCODE includes the primitive data type (vectors), the memory model (a stack), the control mechanism (recursion), and support for nested parallelism (segmented operations). This section covers all these aspects.

Vectors: the primitive data type

The most basic aspect of VCODE is the primitive data type, the vector. Vectors are arbitrarily long homogeneous sequences of atomic values. We discuss several issues concerning vectors: shape, processor context, scalar operations, structures, and arrays.

Many data-parallel languages include the notion of *shape*. In these languages, each shape has a fixed size, every parallel variable belongs to a shape, and most of the operations are only valid between variables of the same shape. In PARALATION LISP this notion is called a *paralation*, in *Lisp it is called a *virtual processor set*, and in C* it is called a *domain*. VCODE does not include the notion of shape—each vector is associated with its own size. CM-Lisp is similar to VCODE in this respect. The omission of shape makes the definition of VCODE smaller, since it is not necessary to introduce a shape type and instructions for allocating, deallocating, activating and deactivating shapes. One of the main arguments for including the notion of shape is that it informs the compiler when parallel variables should be laid out in the same pattern, thereby leading to more efficient code. It turns out, however, that it is relatively easy for a compiler to infer that vectors are of the same shape; our current compiler performs this analysis. We therefore did not feel it necessary to add this feature to the language.

Many data-parallel languages also include the notion of a *processor context*. In these languages, each location of a parallel variable has a context flag associated with it, and operations are only performed in locations where the flag is set to true. Examples of such languages are C*, *Lisp [15] and Fortran 8x (inside a `WHERE` statement). VCODE does not include the notion of processor context. This is largely because the notion of processor context causes many subtle problems when combined with instructions that rearrange the elements of a parallel variable. In the spirit of keeping things simple, we decided to avoid these problems and simulate the processor context above the level of VCODE. The `WHERE` statement of Fortran 8x and C*, and the `*if` statement of *Lisp, can be simulated in VCODE by keeping a vector of context flags explicitly and then either packing the active elements into a smaller vector and unpacking them when leaving the statement, or executing expressions everywhere but using a select instruction to mask out the result on assignments. Section 4.1 shows an example of this for C*.

VCODE has no scalar data type; any scalar code must be implemented with vectors of unit length using the elementwise instructions. This decision was made after we analyzed the performance loss that the use of unit-length vectors would incur, and realized that it would be small: in a compiler, compile-time analysis would locate and optimize the scalar code, and in an interpreter, the instruction overhead is great enough that the extra cost is not significant. As well as saving data types, the decision saves 20 or so scalar instructions.

All the elements of a vector in VCODE must be of the same type—the vectors are *homogeneous*. Not allowing a vector to have a mix of types greatly simplifies the implementation of the vector primitives. Some of the existing data-parallel languages, such as *Lisp, CM-Lisp and PARALATION LISP, allow heterogeneous vectors. Elsewhere we show how heterogeneous vectors can be simulated with homogeneous vectors [2]. A translator from CM-Lisp to VCODE, for example, could implement this mapping. VCODE also only allows atomic values within a vector. It is not possible, for example, to generate a vector of points each of which has two integers. Again this restriction is imposed because vectors of structures are difficult to implement, complicate the language, and can be simulated at a higher level—a vector of points can be implemented with two vectors of integers.

VCODE does not include an array data type. Once again, this is because arrays would complicate VCODE and can easily be mapped onto vectors. It is possible that we will lose some performance by not carrying the array information into VCODE, since array operations on parallel machines often take advantage of locality. Part of our analysis of VCODE will be to determine how well a compiler can derive this information. It may be necessary to add directives to VCODE to gain full performance.

Memory management

Memory management is a major problem in designing a portable parallel language. In the context of VCODE, this refers to mechanisms for allocation, layout, and reclamation of vector storage. It is here that parallel machines show the greatest diversity: allocation may have to be done statically, or it may be done at runtime; data may have to be explicitly partitioned among processors' private memory modules, or it may reside in shared memory; and the length of the data may have to be padded to be a multiple of the number of processors.

Dynamic allocation of vectors was necessary because most data-parallel languages require it. The mechanism for data layout, being machine-specific, was relegated to the runtime system or the interpreter. The remaining issue was reclamation of vector storage, which could either be explicit (freed by the program) or implicit (freed by the runtime system). These possibilities led to three alternatives for memory management: Lisp-style dynamic allocation with garbage collection (implicit storage reclamation), C-style malloc/free operations (explicit storage reclamation), and a vector stack model (implicit storage reclamation).

The garbage collection alternative was rejected because of the complexity of implementing garbage collectors, especially on machines without virtual memory, and because the presence of a garbage collector makes accurate timing measurements very difficult. C-style malloc/free operations were rejected as they were considered to be more low-level than the abstractions provided by VCODE. We did not wish to burden the programmer with the task of memory management when it can be better handled by the interpreter or compiler. The stack-based discipline was chosen as it simplifies memory management and makes it very easy to reclaim memory taken by vectors. At first sight, it appears to introduce severe inefficiencies due to the inherent COPY and POP operations. However, a two level stack implementation—a first-level stack of pointers and a second-level vector store—reduces stack operations to pointer manipulations (as opposed to data copying). Reference counting can be used to perform safe overwrite operations. Further, the VCODE compiler optimizes allocation and reclamation of vector storage. Along with loop optimizations it performs, the code it generates is largely register-based.

Single-assignment semantics

VCODE has a single-assignment semantics: a vector is only written once. This fits well with the stack storage model, and simplifies the work of the compiler by reducing data dependence problems. This does not cause problems with updating vectors or arrays (as in traditional dataflow languages) since computations update entire vectors, not single elements of the vector. Both the interpreter and the compiler can optimize the implementation by doing in-place operations when it is safe to do so.

Recursion as the iteration construct

Recursion is the only iteration construct in VCODE. This fits well with the single-assignment semantics and the vector nature of the language. Explicit iterative constructs typically involve side effects, and this causes problems in conjunction with single assignment. The vector primitives remove the innermost level of iteration (that would occur, for instance, if the vector operation was written as a loop). The compiler can recognize tail-recursive calls and compile them into loops in the final code. It is straightforward to build certain iterative constructs on top of VCODE, such as the `while` and `forall` constructs of SISAL [16].

Nested Parallelism

As mentioned in Section 1, we want VCODE to support nested parallelism. Nested parallelism is the ability to define a parallel routine, and then nest it inside another parallel routine such that it is called multiple times in parallel. For example, to implement a figure drawing algorithm based on drawing multiple lines, a user could define a parallel routine that draws a single line given its endpoints, and then apply it to multiple lines:

```
for each endpoint-pair in endpoint-pairs
    draw-line(endpoint-pair);
```

One way to support nested parallelism would be to supply direct nested parallel constructs by including nested vectors and forms for mapping user-defined functions over the elements of a vector. Unfortunately, such constructs greatly complicate the language and make it significantly more difficult to implement. Instead we decided to supply a data type and instructions that indirectly allow implementation of nested parallelism. The data type is the segment descriptor and the instructions are the segmented instructions. These are quite easy to implement, and, on the machines we have looked at, are almost as efficient as the unsegmented versions. Segments can be used to implement nested parallelism with a technique called flattening nested parallelism [3]. This technique involves placing each nested parallel call in its own segment, therefore allowing them to operate independently but in parallel.

Matrix instructions can also be implemented in terms of segmented vector instructions by mapping rows (or columns) of the matrix into segments. Segmented vectors are more general than matrices, as they can also be used to represent irregular data structures such as tree and graphs [2].

In VCODE we do not supply unsegmented primitives for the same reason we do not supply scalars: a compiler can either determine when a routine is unsegmented or can compile two versions, one segmented and one unsegmented.

3.2 Primitives

A large number of instructions are candidates for inclusion in a data-parallel language. As well as traditional arithmetic and logical operations applied elementwise over vectors, instructions could append vectors, subselect vectors, shift vectors, and reverse vectors. The instruction set of the Connection Machine has over 400 instructions.

In VCODE, rather than including all possible instructions, we chose to go with a greatly reduced primitive instruction set, and to supply libraries of functions built out of the primitive instructions. The translator of a high-level language can use the library functions as well as the primitives. The implementor of VCODE on a new machine only needs to implement the small set of primitive instructions, and, optionally, optimized versions of the library functions for efficiency. Commonly used operations such as reductions and structured permutations (*e.g.*, reverse, rotate) are likely candidates for such optimization. The VCODE compiler also recognizes such operations and generates specialized code for them.

While we stress minimality, we do not carry it to impractical extremes. For instance, it is possible to supply fully general permute instructions (taking both mask and default vectors) and treat all other permutes as special cases of these. However, the overhead for this in an interpreter would be considerable, because of the need to generate arguments that go unused. We therefore supply a rich set of permute instructions so that the user can select the one best suited to his problem.

Communication Instructions

Existing data-parallel languages allow a wide variety of constructs for rearranging the elements of a vector; we will call these communication instructions. Communication instructions range from simple shifts and rotates in Fortran 8x, to the very powerful `match` and `move` primitives in PARALATION LISP. Deciding on a good set of communication instructions is difficult since the relative costs of many of the instructions are machine-dependent.

In choosing a set of instructions, we required that they be efficiently implementable on all the machines of interest, and that they be able to implement all the communication constructs in existing data-parallel languages without serializing (taking time proportional to the length of the vector). For example, VCODE does not supply a "combine" instruction that does not require the index vector to be a permutation, but instead uses some function to combine multiple values arriving at a single location in the destination vector. This is similar to the CM-Lisp β instruction. The combine instruction cannot be implemented efficiently on the CRAY Y-MP, Encore Multimax, or Intel hypercube since the hardware does not support it. It can, however, be simulated with $O(\lg n)$ calls to the other instructions. VCODE also does not supply a shift or rotate directly because they can be implemented using the permute instruction and since it is not possible to build the permute instruction out of the shift instruction without serializing.

The Scan Instructions

The scan instructions were included in VCODE because they are very convenient for building many other operations [2] and can be implemented efficiently if the scan operator is restricted to be associative. The PACK and UNPACK operation in Fortran 8x (see Section 4.2) can be implemented with a scan and a permute. The `index` instruction of PARALATION LISP (similar to ι in APL) can be implemented with a scan. The SUM in Fortran 8x, β+ in CM-Lisp, `xref` in PARALATION LISP, and += in C* can all be implemented with a scan, two extracts, and an add.

4 Mapping other languages onto VCODE

In this section, we illustrate how three data-parallel languages—C*, Fortran 8x and PARALATION LISP—can be mapped onto VCODE. These are illustrative examples rather than complete translation schemes for these languages. We have taken a few liberties with VCODE syntax in order to facilitate the exposition, chiefly in eliding segment descriptors where scans and permutes are obviously being performed on a single segment.

4.1 C*

C* is a data-parallel extension of the C programming language, originally developed in the context of the Connection Machine. Efforts have been made to port it to other machines [18]. Figure 6 shows an example of some C* code. In this example, the salaries of all employees whose salaries are less than 40000 are summed into the `mono` variable `total_salary`. Two ways of translating this into VCODE are shown in the code fragments `salary-sum-1` and `salary-sum-2`. In the first method, salaries that are less than 40000 are gathered into a smaller vector using a permute with index masking, and a `+_reduce` is used

```
total_salary = 0;
[domain employee].{
  if (salary < 40000)
    total_salary += salary;
}

------------------------------------

(def salary-sum-1 (sal len)
  (let ((fl (< (dist 40000 len) sal)))
    (+_reduce
      (spermute
        sal
        (+_scan fl)
        fl
        (+_reduce fl)))))

(def salary-sum-2 (sal len)
  (+_reduce
    (select
      (< (dist 40000 len) sal)
      (dist 0 len)
      sal)))
```

Figure 6: Two different translations of C* code into VCODE.

to sum this vector. The second (and more efficient) method is to mask out salaries greater than or equal to 40000 by placing a 0 in their positions, and then use a +_reduce to sum all the salaries. Although the second method potentially sums a longer vector, it is better because of the additional +_scan and spermute in the first method. The +_reduce operation is not a primitive VCODE instruction, but a function (possibly optimized by the implementation). C* does not allow nested parallelism, so all scan and permute operations are unsegmented.

4.2 Fortran 8x

The proposed standard for Fortran, Fortran 8x, includes a set of vector extensions. These include elementwise operations such as `A[1:100] = B[1:100] + C[1:100]` that add the values in corresponding locations from 1 to 100 in the vectors `B` and `C` into the vector `A`. Such elementwise operations correspond directly to VCODE primitives. More complex elementwise operations such as `A[1:100:2] = B[1:100:2] + C[1:100:2]`, where elements are added with a stride of 2, can be implemented by a short sequence of instructions. The language also includes a set of array instrinsics (see Figure 7). Some of the array intrinsics map directly onto VCODE instructions for one-dimensional arrays. For multidimensional arrays and for other intrinsics, we have implemented VCODE routines, none of which requires more than 10 instructions (typically 2 or 3). Figure 7 shows the translation of the UNPACK intrinsic. We expect that the VCODE compiler should be able to get performance within 20% of optimized library versions.

Reduction Functions: sum, product, maxval, minval
Construction Functions: pack, merge, spread, unpack
Manipulation Functions: transpose, eoshift, cshift
Location Functions: maxloc, minloc

```
(def unpack (result flags values)
  (select
    flags
    (bpermute
      values (+_scan flags) flags)
    result))
```

Figure 7: Some Fortran 8x array instrinsics, and translation of the UNPACK primitive into VCODE.

4.3 PARALATION LISP

PARALATION LISP is a data-parallel language based on LISP. It is different from the two previous languages in that it allows nested parallelism—code with an inner parallel routine nested within an outer parallel routine. The segmented instructions in VCODE are needed for implementing this kind of parallelism efficiently. To demonstrate this, Figure 8 shows the same routine invoked twice—at the top level, and within an `elwise` form. The VCODE is the same in either case, but the segment descriptors are different (with a single segment in one case, and multiple segments in the other).

5 Implementing VCODE

This section discusses the issues we have encountered in implementing VCODE on different kinds of parallel machines, and specific techniques for implementing it on the Connection Machine.

5.1 Implementation issues

Two issues arise in implementing VCODE: how to implement the primitives efficiently, and how to implement the control efficiently. This section is chiefly concerned with the first issue, and assumes that the code is being executed instruction by instruction, with all processors synchronizing after each instruction. This is adequate for an interpreter and is desirable for debugging. This is also the mode of execution on massively parallel machines such as the Connection Machine. However, this strict synchronization regimen is a major bottleneck to speed in MIMD machines (where synchronization is expensive), and can be relaxed without affecting the semantics of the program. A discussion of these issues is beyond the scope of this paper.

Three issues are important for an efficient implementation of VCODE primitives on a parallel machine:

- **Matching grain size to machine characteristics:** Machines vary widely in the grain size they can support efficiently. Parallelism may also be available at multiple levels in a machine: pipelined functional units, multiple functional units, and multiple processors.

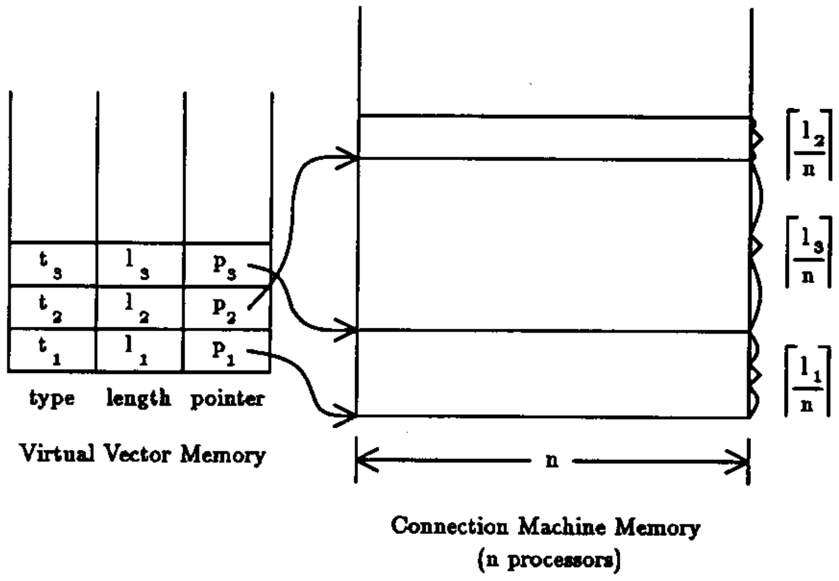
```
(<- a :by (choose b))

(elwise ((aa a)
         (bb b))
  (<- aa :by (choose bb)))
```

a	=	[5	6	3	1	8	3	7]
b	=	[1	0	1	1	0	0	1]
result	=	[5	3	1	7]			

```
(def plisp-example (aflat bflat seg)
  (spermute
    aflat
    (+_scan bflat seg)
    bflat
    (+_reduce bflat seg)
    seg))
```

aflat	=	[5	6	3	1	8	3	7]
bflat	=	[1	0	1	1	0	0	1]
+_scan bflat	=	[0	1	1	2	3	3	3]
+_reduce bflat	=	4						
spermute	=	[5	3	1	7	9]		

Figure 8: Translation of PARALATION LISP into VCODE.



t_3
l_3
p_3
t_2
l_2
p_2
t_1
l_1
p_1
type
length
pointer
Virtual Vector Memory
n
Connection Machine Memory
(n processors)
$\lceil \frac{l_2}{n} \rceil$
$\lceil \frac{l_3}{n} \rceil$
$\lceil \frac{l_1}{n} \rceil$

Figure 9: An illustration of how the vector memory is simulated on the Connection Machine. The *virtual vector memory* is stored on the front-end computer and contains pointers to slices of the actual Connection Machine memory.

- **Synchronization:** Scan and permute operations require synchronization for correct multiprocessor implementation. On machines such as the Connection Machine, such mechanisms are provided by the lock-step nature of the computation. On other kinds of machines (*e.g.*, hypercubes), fast mechanisms must be designed for them.
- **Load balancing:** Load balancing is crucial in parallel programming for achieving short execution times. In the context of VCODE, this becomes important in the case of segmented instructions. The nature of VCODE primitives allows prescheduling of loops in general, allowing each processor to compute its partition of the input [27]. However, segmented instructions present a problem since there is no guarantee that segments will be of equal length. Different techniques are appropriate depending on the number and lengths of the segments, the tradeoff being between synchronization and load balancing.

5.2 Implementing vector memory on the Connection Machine

The Connection Machine is the most straightforward machine on which to implement VCODE. Many of the VCODE instructions are similar to the Connection Machine parallel instruction set (PARIS) [24], but the underlying model is somewhat different since each vector in VCODE has its own length.

To support the vector memory on the Connection Machine, arbitrarily long vectors must be mapped onto the fixed number of processor memories of the Connection Machine. This is implemented using a level of indirection. Each location of a *virtual vector memory* is a structure that contains a pointer to a physical location in the Connection Machine memory where the vector is actually stored (see Figure 9). It also contains other information, including the length and the type of the vector. The virtual vector memory is stored on the front end computer.

Since the vectors can be longer than the number of processors in the Connection Machine, each vector might take up many slices of the Connection Machine memory. A vector of length m on an n processor machine, will require $\lceil m/n \rceil$ slices of memory. Since vector lengths in general are not a multiple of n, $n \times \lceil m/n \rceil - m$ elements are left unused. The elements within a vector are numbered so that elements within the same processor have contiguous indices. For a vector of length m on a n processor machine, elements $0, \ldots, (\lceil m/n \rceil - 1)$ are in processor 0, elements $\lceil m/n \rceil, \ldots, (2\lceil m/n \rceil - 1)$ are in processor 1, and so on.

To implement the vector memory we need some sort of memory management, because it might be necessary to allocate space in the Connection Machine memory when we write a vector into the virtual vector memory. If the vector we write is the same length as the previous vector stored at that location in the virtual vector memory, then we can simply write over the old version. If, however, the vector is longer, the vector might require more slices of the Connection Machine memory and not fit in the previous position. We therefore require some sort of memory management to allocate new space in the Connection Machine memory when writing into a location of the virtual vector memory, and deallocate the space when another vector is overwritten. When using the vector stack, the allocation and deallocation is straightforward since we can keep an analogous stack in the Connection Machine memory, and slices of memory can always be allocated and deleted from the top of the stack. The only somewhat complicated instruction is the POP, since this requires compacting the stack. When using the heap, a more complicated memory management scheme is necessary. At present, we do not reclaim memory that is allocated on the heap.

This general technique of mapping arbitrarily long vectors using a level of indirection can be used for a much broader class of machines.

5.3 Implementing the instructions on the Connection Machine

We now describe how the VCODE instructions are implemented on the Connection Machine. All instructions with vector arguments take indices into the virtual vector memory rather than directly into the Connection Machine memory. When executed, each instruction calculates how many slices of memory each vector occupies and loops over these slices. We now describe the different classes of instructions.

Elementwise Instructions

The elementwise vector instructions take as input some set of vectors each of the same length. Based on the length of the vectors and the size of the machine, they calculate how many slices of CM memory are taken by each vector. They then loop over each slice, loading the values into each processor from its local memory, executing the particular elementwise operation and storing the result back into local memory. The elementwise instructions never require communication among processors.

Permutation Instructions

The permutation instructions are supported by the routing hardware of the Connection Machine. The router uses a packet-switched message routing scheme that directs messages along the hypercube wires to their destinations (see [10] for more details).

The router requires a processor address, and a location within each processor to deliver each value. Before executing the route, we must therefore translate the destination indices from the *index* argument of the permutation instruction into two parts: the processor address, and the slice number within that processor. This can be executed by dividing each index by the number of slices, using the quotient as the processor address, and the remainder as the slice number within that processor.

It is important for the permutation instruction that each processor has independent addressing: different processors can access different locations simultaneously. This is because when a value arrives at the destination processor, the processor must place it in the correct slice. The values arriving at different processors at the same time might be destined for different slices. The CM-2 has such independent addressing.

Another trick used to improve the performance of the permutation instruction is for each processor to randomly select the elements to be sent from its memory rather than to serially loop over them. This reduces the problem of having many elements in one slice going to a single processor and congesting the routing hardware.

Scan Instructions

The scan instructions are implemented on the Connection Machine using the PARIS instructions which use a mix of a binary tree and hypercube algorithm. The implementation uses the same hypercube wires as the router but does not use the routing hardware. When there are multiple elements per processor, each processor first sums an n/p section of the vector to generate a processor sum; the tree technique is then be used to scan the processor sums; and the results are used as an offset for each processor to scan within its n/p section (see [2] for more details).

6 Conclusions

In this early stage of development of parallel programming, it is particularly desirable to have a common platform spanning various machine classes that could be used for experimenting with languages, algorithms, compilers, and architectures. This paper proposes a data-parallel intermediate language, VCODE, as such a platform, and discusses the design of the language and its implementation, and shows how various existing data-parallel languages may be mapped into it.

VCODE was designed to be capable of expressing the features of a wide variety of data-parallel languages and to be capable of being implemented on a wide variety of machines, spanning massively parallel SIMD machines, shared-memory multiprocessors, distributed-memory machines, and vector machines. Initial results suggest that this can be achieved without a large performance loss over machine-specific programming languages.

Acknowledgments

We would like to thank Allan Fisher, Thomas Gross, Peter Lee, Thomas Stricker and Skef Wholey for many hours of fruitful discussions.

Multimax is a trademark of Encore Computer Corporation. CRAY Y-MP is a trademark of Cray Research, Inc. Connection Machine is a registered trade mark of Thinking Machines Corporation. C*, *Lisp and CM-2 are trademarks of Thinking Machiens Corporation. WARP is a service mark of Carnegie Mellon University.

A VCODE Definition

A.1 Primitive types

There are four primitive data types:

- Integer (int): vector of (twos-complement) integers.
- Boolean (bool): vector of booleans.
- Floating point (float): vector of floating point numbers.
- Segment descriptor (segdes): representation up to implementation.

To keep the number of VCODE primitives small, we retain only the most general versions of instructions and data types. Elementwise operations take either *int*, *float*, or *bool* arguments, and scalar operations fall out as the special case when the length of the vector is 1. Similarly, the vector instructions all take a *segdes* type as one of the arguments; the unsegmented versions of the operations is the special case when the segment descriptor contains just one segment.

A.2 Instructions

Instructions are classified into five types: elementwise operations, vector instructions, segment descriptor instructions, control instructions, and I/O instructions. For each instruction, we indicate the input and output types, any constraints that must be satisfied by its arguments, and the effect of the instruction. The appropriate response to an error condition is unspecified and left to the implementation. All instructions take their operands from the stack and leave their results on the stack.

The *segdes* type will be represented as a lengths vector delimited by square brackets for illustrative purposes in the following text.

A.2.1 Elementwise operations

For any operation in this class that takes more than one operand, the operands must have equal lengths.

1. {*int* | *float*} `+ {INT, FLOAT}` {*int int* | *float float*}

   Returns the sum of the top two vectors on the stack.

```
| (1 2 3 0 4) |               | (4 6 3 2 7) |
| (3 4 0 2 3) |     + INT     |      :      |
|      :      |               |_____________|
|_____________|
```

2. {*int* | *float*} `- {INT, FLOAT}` {*int int* | *float float*}

   Returns the difference of the top two vectors on the stack.

```
| (4 5 1 3 2) |               | (3 1 7 6 3) |
| (7 6 8 9 5) |     - INT     |      :      |
|      :      |               |_____________|
|_____________|
```

3. {*int* | *float*} `*` {`INT,FLOAT`} {*int int* | *float float*}

Returns the product of the top two vectors on the stack.

```
| (3 1 5 2 0) |                | (0 1 20 6 0) |
| (0 1 4 3 2) |     * INT      |      :       |
|      :      |                |______________|
|_____________|
```

4. {*int* | *float*} `/` {`INT,FLOAT`} {*int int* | *float float*}

Returns the quotient of the top two vectors on the stack. The divisor vector must not have any zero elements.

```
| (2 4 3 8 1) |                | (2 1 3 0 3) |
| (5 7 9 7 3) |     / INT      |      :      |
|      :      |                |_____________|
|_____________|
```

5. {*int* | *float*} `%` {`INT,FLOAT`} {*int int* | *float float*}

Returns the remainder of the top two vectors on the stack. The divisor vector must not have any zero elements.

```
|   (2 4 3 8 4)   |                | (1 3 0 7 -1) |
| (5 7 9 7 -13)   |     % INT      |      :       |
|        :        |                |______________|
|_________________|
```

6. {*bool*} `<` {`INT,FLOAT`} {*int int* | *float float*}

Returns the result of the "less than" test applied to the top two vectors on the stack.

```
| (7 4 2 1 0) |                | (T F F F F) |
| (3 4 5 2 1) |     < INT      |      :      |
|      :      |                |_____________|
|_____________|
```

7. {*bool*} `>` {`INT,FLOAT`} {*int int* | *float float*}

Returns the result of the "greater than" test applied to the top two vectors on the stack.

```
| (7 4 2 1 0) |                | (F F T T T) |
| (3 4 5 2 1) |     > INT      |      :      |
|      :      |                |_____________|
|_____________|
```

8. {*bool*} `=` {`INT,FLOAT`} {*int int* | *float float*}

Returns the result of the equality test applied to the top two vectors on the stack.

```
| (3 4 5 2 1) |                | (F T F F F) |
| (7 4 2 1 0) |     = INT      |      :      |
|      :      |                |_____________|
|_____________|
```

9. {*int*} `LSHIFT` {*int int*}

Returns the result of shifting each element of the data vector left by the amounts in the shift vector. The elements of the shift vector must be non-negative. The vacated bits are zero-filled.

```
| (1 1 2 2 0) |                 |                      |
| (6 8 3 7 8) |     LSHIFT      | (12 16 12 28 8)      |
|      ⋮      |                 |          ⋮           |
```

10. {*int*} `RSHIFT` {*int int*}

Returns the result of shifting each element of the data vector right by the amounts in the shift vector. The elements of the shift vector must be non-negative. The vacated bits are filled with the sign bit.

```
| (1 1 2 2 0) |                 |               |
| (6 8 3 7 8) |     RSHIFT      | (3 4 0 1 8)   |
|      ⋮      |                 |       ⋮       |
```

11. {*bool* | *int*} `NOT {BOOL, INT}` {*bool* | *int*}

Returns the negation (boolean for `bool`s, bitwise for `int`s) of the vector.

```
| (T F F T T) |    NOT BOOL     | (F T T F F) |
|      ⋮      |                 |      ⋮      |
```

12. {*bool* | *int*} `AND {BOOL, INT}` {*bool bool* | *int int*}

Returns the AND function (boolean or bitwise, as appropriate) applied to the top two vectors on the stack.

```
| (T T F F) |                 |           |
| (T F T F) |    AND BOOL     | (T F F F) |
|     ⋮     |                 |     ⋮     |
```

13. {*bool* | *int*} `OR {BOOL, INT}` {*bool bool* | *int int*}

Returns the OR function (boolean or bitwise, as appropriate) applied to the top two vectors on the stack.

```
| (T T F F) |                 |           |
| (T F T F) |    OR BOOL      | (T T T F) |
|     ⋮     |                 |     ⋮     |
```

14. {*int* | *bool* | *float*} `SELECT {INT, BOOL, FLOAT}` {*bool int int* | *bool bool bool* | *bool float float*}

Returns the result of selecting from the two data vectors based on the value of the selection vector.

```
| (4 1 9 7) |                 |           |
| (2 0 1 6) |                 |           |
| (T F F T) |   SELECT INT    | (2 1 9 6) |
|     ⋮     |                 |     ⋮     |
```

15. {*int*} `RAND` {*int*}

Returns an integer vector of random numbers based on the ranges specified in the data vector. If the range specification is n, the number will be in the range 0–$(n-1)$. The elements of the data vector must be positive.

(2 5 8 8) ⋮	RAND	(1 4 3 7) ⋮

16. {*int*} `FLOOR` {*float*}

Returns the vector truncated towards negative infinity.

(2.6 2.5 0.3 -0.7) ⋮	`FLOOR`	(2 2 0 -1) ⋮

17. {*int*} `CEIL` {*float*}

Returns the vector truncated towards positive infinity.

(2.6 2.5 0.3 -0.7) ⋮	`CEIL`	(3 3 1 0) ⋮

18. {*int*} `TRUNC` {*float*}

Returns the vector truncated towards zero.

(2.6 2.5 0.3 -0.7) ⋮	`TRUNC`	(2 2 0 0) ⋮

19. {*int*} `ROUND` {*float*}

Returns the vector rounded to the nearest integer. If the fractional part is 0.5, the number is rounded to the nearest even integer.

(2.6 2.5 0.3 -0.7) ⋮	ROUND	(3 2 0 -1) ⋮

20. {*float*} `I_TO_F` {*int*}

Changes the `int` into a `float`.

(1 2 3 -1) ⋮	`I_TO_F`	(1.0 2.0 3.0 -1.0) ⋮

21. {*float*} `LOG` {*float*}

Returns the natural logarithm of the vector. All elements must be positive.

(1.0 3.2 4) ⋮	`LOG`	(0.0 1.163 1.386) ⋮

22. *{float}* SQRT *{float}*

Returns the square root of the vector. All elements must be non-negative.

```
| (2.0 4.0 6.0 9.0) |          | (1.414 2.0 2.449 3.0) |
|        ⋮          |   SQRT   |          ⋮            |
```

23. *{float}* EXP *{float}*

Returns the exponential function of the vector.

```
| (1.0 2.0 -1.0) |         | (2.718 7.389 0.367) |
|       ⋮        |   EXP   |          ⋮          |
```

A.2.2 Vector instructions

A vector *V* of type `int`, `bool` or `float` and a vector *S* of type `segdes` are said to be **compatible** iff the length of *V* is equal to the sum of the lengths of the segments described by *S*. We will use this notion of compatibility to define constraints in this class of instructions.

1. *{int |float}* +_SCAN {INT,FLOAT} *{int segdes | float segdes}*

   Returns the segmented plus-scan of the vector. The vectors must be compatible. The identity for the operation is 0.

```
|     [3 3]     |                |               |
| (1 3 2 3 5 1) |   +_SCAN INT   | (0 1 4 0 3 8) |
|       ⋮       |                |       ⋮       |
```

2. *{int | float}* MAX_SCAN {INT,FLOAT} *{int segdes | float segdes}*

   Returns the segmented max-scan of the vector. The vectors must be compatible. The identity for the operation is $-\infty$ (the minimum value of the data type for a given implementation).

```
|     [3 3]     |                  |                 |
| (1 3 2 3 5 1) |   MAX_SCAN INT   | (−∞ 1 3 0 3 5)  |
|       ⋮       |                  |        ⋮        |
```

3. *{int | float}* MIN_SCAN {INT,FLOAT} *{int segdes | float segdes}*

   Returns the segmented min-scan of the vector. The vectors must be compatible. The indentity for the operation is ∞ (the maximum value of the data type for a given implementation).

```
|     [3 3]     |                  |                |
| (1 3 2 3 5 1) |   MIN_SCAN INT   | (∞ 1 1 0 3 3)  |
|       ⋮       |                  |       ⋮        |
```

4. *{bool}* AND_SCAN *{bool segdes}*

Returns the segmented AND-scan of the vector. The vectors must be compatible. The identity element for the operation is T.

```
 | [3 3]         |                | (T T F T F F) |
 | (T F T F T T) |   AND_SCAN     |       ⋮       |
 |      ⋮        |                |_______________|
 |_______________|
```

5. {*bool*} `OR_SCAN` {*bool segdes*}

   Returns the segmented OR-scan of the vector. The vectors must be compatible. The identity for the operation is F.

```
 | [3 3]         |                | (F T T F F T) |
 | (T F T F T T) |   OR_SCAN      |       ⋮       |
 |      ⋮        |                |_______________|
 |_______________|
```

6. {*int* | *bool* | *float*} `PERMUTE {INT,BOOL,FLOAT}` {*int int segdes* | *bool int segdes* | *float int segdes*}

   Simple segmented permute operation. Calling the arguments (**data index segdes**), the following constraints must be satisfied:

   - **data** and **segdes** must be compatible.
   - **index** and **segdes** must be compatible.
   - The elements of **index** must form a permutation within each segment of **segdes**.

   The elements of **data** are permuted to the locations specified by **index**, within each segment. The result vector is compatible with **segdes**.

```
 | [4 2]         |                |               |
 | (3 1 0 2 1 0) |  PERMUTE INT   | (2 1 3 0 5 4) |
 | (0 1 2 3 4 5) |                |       ⋮       |
 |      ⋮        |                |_______________|
 |_______________|
```

7. {*int* | *bool* | *float*} `DPERMUTE {INT,BOOL,FLOAT}` {*int int int segdes segdes* | *bool int bool segdes segdes* | *float int float segdes segdes*}

   Segmented permute operation with default. Calling the arguments (**data index default src-seg dst-seg**), the following constraints must be satisfied:

   - **data** and **src-seg** must be compatible.
   - **index** and **src-seg** must be compatible.
   - **default** and **dst-seg** must be compatible.
   - The elements of **index** must be distinct and properly bounded within each segment of **dst-seg**.
   - **src-seg** and **dst-seg** must have the same number of segments.

The elements of **data** are permuted to the positions indicated by the elements of **index**. Any position that does not receive a value from **data** is filled with the value at the corresponding position of **default**. The result has the same length as **default** and is compatible with **dst-seg**.

```
| [5 3]           |                 |                   |
| [4 2]           |                 |                   |
| (5 5 5 5 5 6 6) |                 |                   |
| (3 4 0 2 2 1)   | DPERMUTE  INT   |                   |
| (0 1 2 3 4 5)   |                 | (2 5 3 0 1 6 5 4) |
|       ⋮         |                 |        ⋮          |
```

8. {*int* | *bool* | *float*} `SPERMUTE` {`INT,BOOL,FLOAT`} {*int int bool segdes segdes* | *bool int bool segdes segdes* | *float int bool segdes segdes*}

Segmented select-permute operation with flags. Calling the arguments (**data index flags src-seg dst-seg**), the following constraints must be satisfied:

- **data** and **src-seg** must be compatible.
- **index** and **src-seg** must be compatible.
- **flags** and **src-seg** must be compatible.
- The elements of **index** not masked out by **flags** must be distinct and properly bounded within the segments defined in **src-seg**.
- **src-seg** and **dst-seg** must have the same number of segments.

The elements of **data** not masked out by **flags** are permuted to the positions indicated by the corresponding elements of **flags**. The result is compatible with **dst-seg**.

```
| [3 3]         |                 |           |
| [4 2]         |                 |           |
| (T F T T T F) |                 |           |
| (2 1 0 1 0 0) |  SPERMUTE  INT  |           |
| (0 1 2 3 4 5) |                 | (2 3 0 4) |
|      ⋮        |                 |     ⋮     |
```

9. {*int* | *bool* | *float*} `BPERMUTE` {`INT,BOOL,FLOAT`} {*int int segdes segdes* | *bool int segdes segdes* | *float int segdes segdes*}

Segmented inverse-permute instruction. The unfilled positions in the result vector are set to 0 for `ints` and `floats`, and F for `bools`. Calling the arguments (**data index src-seg dst-seg**), the following constraints must be satisfied:

- **data** and **src-seg** must be compatible.
- **index** and **dst-seg** must be compatible.
- The elements of **index** must be properly bounded within the segments defined in **src-seg**.
- **src-seg** and **dst-seg** must have the same number of segments.

```
    [4 2]
    [3 3]
 (2 0 1 1 0 2)     BPERMUTE INT
 (1 2 3 4 5 6)                      (3 1 2 0 5 4 6)
       ⋮                                  ⋮
```

10. *{int | bool | float}* `BFPERMUTE {INT,BOOL,FLOAT}` *{int int bool segdes segdes | bool int bool segdes segdes | float int bool segdes segdes}*

Segmented inverse-permute instruction with flags. The unfilled positions in the vector are set to 0 for `ints` and `floats`, and F for `bools`. Calling the arguments (**data index flag src-seg dst-seg**), the following constraints must be satisfied:

- **data** and **src-seg** must be compatible.
- **index** and **flag** must have the same length.
- **index** and **dst-seg** must be compatible.
- The elements of **index** not masked out by **flag** must be properly bounded within the segments defined in **src-seg**.
- **src-seg** and **dst-seg** must have the same number of segments.

```
      [4 3]
 (F T T F T T T)
  (1 0 1 1 0 2)
      [3 3]        BFPERMUTE INT
  (1 2 3 4 5 6)                     (0 1 2 0 4 6 5)
        ⋮                                 ⋮
```

11. *{int | bool | float}* `EXTRACT {INT,BOOL,FLOAT}` *{int int segdes | bool int segdes | float int segdes}*

Segmented extract operation. Calling the arguments **src index segdes**, the following constraints must be satisfied:

- **src** and **segdes** must be compatible.
- **index** must have as many elements as **segdes** has segments.
- The elements of **index** must be non-negative and bounded above by the segment lengths.

The elements specified by **index** are extracted from **src**, one per segment, and returned. The result has the same size as **index**.

```
     [4 2]
     (2 1)
 (0 1 2 3 4 5)        EXTRACT INT        (2 5)
       ⋮                                   ⋮
```

12. $\{int \mid bool \mid float\}$ `REPLACE {INT,BOOL,FLOAT}` $\{int\ int\ int\ segdes \mid bool\ int\ bool\ segdes \mid float\ int\ float\ segdes\}$

    Segmented replace operation. Calling the arguments **src index values segdes**, the following constraints must be satisfied:

    - **src** and **segdes** must be compatible.
    - **index** and **values** must have the same length.
    - **index** must have as many elements as **segdes** has segments.
    - The elements of **index** must be non-negative and bounded above by the segment lengths.

    The result is a vector that has the values of **src** in all locations except those specified in **index** (one per segment), where it has the values of **values**. The result is compatible with **segdes**.

```
 [4 2]
 (3 -1)
 (2 1)                                
 (0 1 2 3 4 5)     EXTRACT INT     (0 1 3 3 4 -1)
 ⋮                                  ⋮
```

13. $\{int \mid bool \mid float\}$ `DIST {INT,BOOL,FLOAT}` $\{int\ segdes \mid bool\ segdes \mid float\ segdes\}$

    Segmented distribute. The second argument is a segment descriptor describing the destination vector.

```
 [3 3]
 (2 4)      DIST INT      (2 2 2 4 4)
 ⋮                         ⋮
```

14. $\{int \mid bool \mid float\}$ `LENGTH {INT,BOOL,FLOAT}` $\{int \mid bool \mid float\}$

    Returns the length of the vector.

```
 (4 3 2 1 0 7 6)     LENGTH INT     (7)
 ⋮                                   ⋮
```

A.2.3 Segment descriptor instructions

1. $\{segdes\}$ `MAKE_SEGDES` $\{int\}$

   Returns a segment descriptor given the lengths of the segments. All the elements of the input must be non-negative.

```
 (3 2)      MAKE_SEGDES      [3 2]
 ⋮                            ⋮
```

2. $\{int\}$ `LENGTHS` $\{segdes\}$

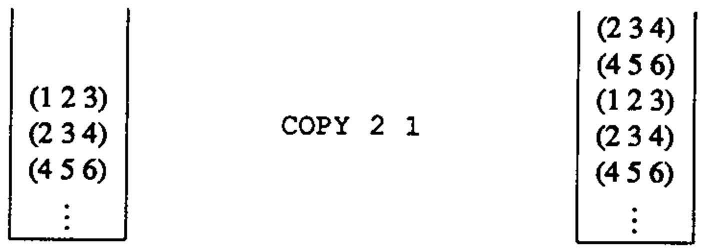
   Returns the lengths of the segments given the segment descriptor.

```
 [3 3]      LENGTHS      (3 3)
 ⋮                        ⋮
```

A.2.4 Control instructions

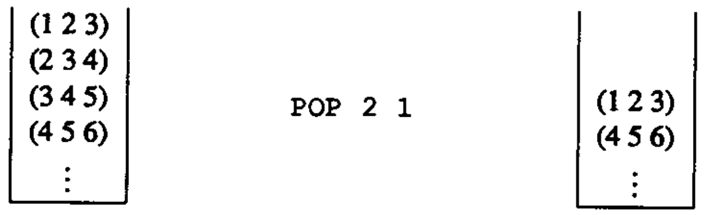
1. *{vector+}* `COPY I J` *{null}*

   Copies `I` elements starting from position `J` within the stack to the top of the stack. The stack must be at least `I+J` deep. `I` and `J` must be non-negative.



2. *{null}* `POP I J` *{null}*

   Deletes `I` elements starting from position `J` within the stack and compresses the stack. The familiar stack pop operation is the special case `POP 1 0`. The stack must be at least `I+J` deep. `I` and `J` must be non-negative.



3. *{null}* `CALL LABEL` *{null}*

   Calls the function named `LABEL`.

4. *{null}* `RET` *{null}*

   Returns from function call.

5. *{null}* `FUNC LABEL` *{null}*

   Defines a function with the name `LABEL`.

6. *{null}* `IF` *{bool}*

   Beginning of a conditional statement. The test is on top of the stack and must be have a length of 1. If the test has a value `T`, the statements following the `IF` are executed until the matching `ELSE` statement is reached, and then control goes to the statement following the matching `ENDIF`. Both branches must leave the stack in the same state with respect to depth and data types. Nested conditionals are matched in the conventional way: to the closest enclosing scope.

7. *{null}* `ELSE` *{null}*

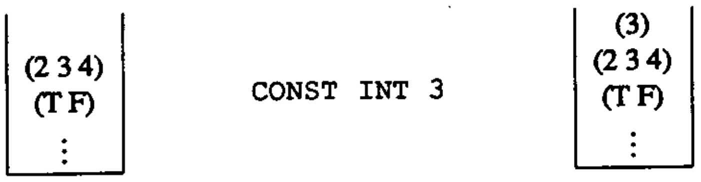
   See description of `IF`.

8. *{null}* `ENDIF` *{null}*

   See description of `IF`.

9. {*int* | *bool* | *float*} `CONST {INT,BOOL,FLOAT} VAL` {*null*}

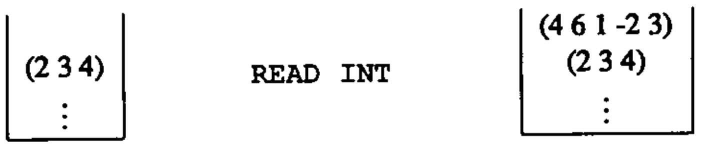
   Puts the constant VAL on top of the stack.



A.2.5 I/O instructions

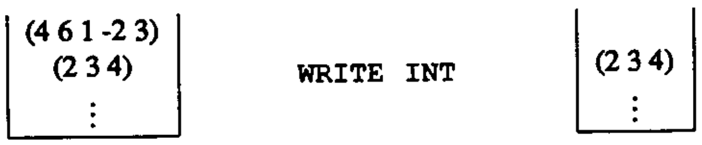
1. {*int* | *bool* | *float*} `READ {INT,BOOL,FLOAT}` {*null*}

   Reads a vector of the appropriate type (terminated by a newline) from the standard input stream.



2. {*null*} `WRITE {INT,BOOL,FLOAT}` {*int* | *bool* | *float*}

   Writes a vector of integers to the standard output stream.



A.3 Miscellaneous features

Comments are delimited by `{` and `}`. Multiline comments are permitted.

References

[1] American National Standards Institute. *American National Standard for Information Systems Programming Language Fortran: S8(X3.9-198x)*, March 1989.

[2] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, Cambridge, MA, 1990.

[3] Guy E. Blelloch and Gary W. Sabot. Compiling Collection-Oriented Languages onto Massively Parallel Computers. *Journal of Parallel and Distributed Computing*, 8(2):119–134, Feb 1990.

[4] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. S. Tseng, J. Sutton, J. Urbanski, and J. Webb. iWarp: An Integrated Solution to High-Speed Parallel Computing. In *Proceedings of Supercomputing '88, IEEE Computer Society and ACM SIGARCH*, November 1988.

[5] F. Crow, G. Demos, J. Hardy, J. Mclaughlin, and K. Sims. 3D Image Synthesis on the Connection Machine. *International Journal of High Speed Computing*, 1(2):329–347, June 1989.

[6] M. D. Durand. Parallel Simulated Annealing: Accuracy vs. Speed in Placement. *IEEE Design and Test of Computers*, 6(3):8–34, June 1989.

[7] E. W. Edmiston, N. G. Core, J. H. Saltz, and R. M. Smith. Parallel Processing of Biological Sequence Comparison Algorithms. *International Journal of Parallel Programming*, 17(3):259–275, June 1988.

[8] R.-D. Fiebrich. Data Parallel Algorithms for Engineering Applications. In *Proceedings Second International Conference on Supercomputing, Vol. 2*, pages 17–22, San Francisco, CA, May 1987.

[9] Leonard G. C. Hamey, Jon A. Webb, and I-Chen Wu. An Architecture Independent Programming Language for Low-Level Vision. *Computer Vision, Graphics, and Image Processing*, 48:246–264, 1989.

[10] W. Daniel Hillis. *The Connection Machine*. MIT Press, Cambridge, MA, 1985.

[11] Kenneth E. Iverson. *A Programming Language*. Wiley, New York, 1962.

[12] Kathleen Knobe, Joan D. Lukas, and Guy L. Steele Jr. Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, Feb 1990.

[13] S. A. Kravitz, R. E. Bryant, and R. A. Rutenbar. Logic Simulation on Massively Parallel Architectures. In *Proceedings 16th Annual International Symposium on Computer Architecture*, pages 336–343, Jerusalem, Israel, 1989.

[14] E. Lander, J. P. Mesirov, and W. Taylor IV. Study of Protein Sequence Comparison Metrics on the Connection Machine CM-2. *Journal of Supercomputing*, 3(4):255–269, Dec 1989.

[15] Clifford Lasser. The essential *Lisp manual. Technical report, Thinking Machines Corporation, Cambridge, MA, July 1986.

[16] James McGraw, Stephen Skedzielewski, Stephen Allan, Rod Oldehoeft, John Glauert, Chris Kirkham, Bill Noyce, and Robert Thomas. *SISAL: Streams and Iteration in a Single Assignment Language, Language Reference Manual Version 1.2*. Lawrence Livermore National Laboratory, Mar 1985.

[17] K. V. Nori, U. Ammann, K. Jensen, H. H. Nageli, and Ch. Jacobi. The PASCAL P compiler: Implementation notes (revised edition). Technical Report 10, ETH, Switzerland, October 1976.

[18] Michael J. Quinn, Philip J. Hatcher, and Karen C. Jourdenais. Compiling C* Programs for a Hypercube Multicomputer. In *Proceedings ACM/SIGPLAN PPEALS 1988—Parallel Programming: Experience with Applications, Languages and Systems*, pages 57–65, New Haven, CT, July 1988.

[19] J. R. Rose and G. L. Steele Jr. C*: An Extended C Language for Data Parallel Programming. In *Proceedings Second International Conference on Supercomputing, Vol. 2*, pages 2–16, San Francisco, CA, May 1987.

[20] Gary W. Sabot. *The Paralation Model: Architecture-Independent Parallel Programming*. MIT Press, Cambridge, MA, 1988.

[21] J. T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, New York, 1986.

[22] D. B. Skillicorn and D. T. Barnard. Parallel Parsing on the Connection Machine. *Information Processing Letters*, 31(3):111–117, May 1989.

[23] Guy L. Steele Jr. CM-Lisp. Technical report, Thinking Machines Corporation, 1986.

[24] Thinking Machines Corporation. Connection Machine parallel instruction set (PARIS), July 1986.

[25] Ping-Sheng Tseng. *A Parallelizing Compiler for Distributed Memory Parallel Computers*. PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, May 1989.

[26] L. W. Tucker. Data Parallelism and Computer Vision Using the Connection Machine. In L. P. Kartashev and S. I. Kartashev, editors, *Proceedings Third International Conference on Supercomputing, Vol. 3*, pages 35–41, Boston, MA, May 1988.

[27] Michael Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge, MA, 1989.