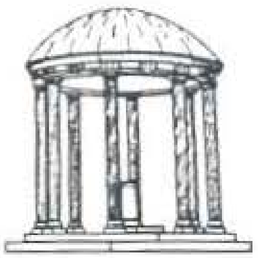
Porta-SIMD User's Manual

TR89-006

January 1989 (revised March 1989)

Russ Tuck

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175



UNC is an Equal Opportunity/Affirmative Action Institution.

Porta-SIMD User's Manual

Russ Tuck
Computer Science Department, Duke University, and
Computer Science Department, University of North Carolina at Chapel Hill*
March 6, 1989

1 Introduction

Porta-SIMD (pronounced PORTA-simm'd) is a portable high-level language for programming massively parallel SIMD (Single-Instruction Multiple-Data) computers. It is the first such language actually implemented in the same form on more than one SIMD computer architecture. Porta-SIMD has been implemented on UNC's Pixel-Planes 4 [FuchsG*85,EylesA*87] and on a Thinking Machines Connection Machine [Hillis85,Thinkin87]; there is also a sequential implementation that runs on ordinary computers.

The Porta-SIMD language is a natural extension of C++ to SIMD architectures, and is relatively easy to learn for programmers familiar with C or C++. This manual is intended for such programmers. Other readers are encouraged to consult [Stroust86,KernigR78] for an introduction to C++ or C, respectively. While familiarity with one or more SIMD architectures is helpful, it is not necessary.

An introduction to SIMD architectures is presented in the next section. It provides the background necessary to understand the rest of this manual. Section 3 provides an overview of Porta-SIMD and a summary of its extensions to C++. The Porta-SIMD language as currently implemented is described in detail in section 4. Section 5 presents example Porta-SIMD programs, and is followed by a section containing instructions for compiling Porta-SIMD programs. The final sections, 7 and 8, discuss the strategy used to implement Porta-SIMD, and additional features planned for Porta-SIMD's future.

2 SIMD Computers

An understanding of Porta-SIMD's extensions to C++ must begin with some knowledge of SIMD computers as they appear to a programmer. A SIMD computer consists of two parts: an ordinary sequential computer, called the "host", and a large number of identical PEs (Processing Elements). Besides evaluating scalar expressions and executing flow-control statements, the host sends commands to the PEs. All the PEs, in simultaneous lock-step fashion, execute each command sent by the host. In response to host commands, each PE can evaluate basic arithmetic and logical expressions, storing data in its local memory as needed. In addition, the host can command each PE to use the result of a logical expression to decide whether to ignore commands until further notice. Those PEs executing host commands are said to be "enabled", while those currently ignoring host commands are "disabled". Finally, the PEs are numbered consecutively from 0 and each is able to compute its own PE number.

This organization allows an entire collection of data values to be stored one per PE and operated on at once. For example, the following statement adds one set of data to another.

```
    a += b;
```

This is much faster than doing the same operation on the host, one value at a time.

```
    for (i=0; i < num_values; i++) {
      a[i] += b[i];
    }
```

However, taking full advantage of this capability generally requires different algorithms than those commonly used on sequential computers.

The above description is true of essentially all SIMD computers. However, some feature additional capabilities. The most important of these is inter-PE communication. While some SIMD computers, including Pixel-Planes, provide no means of communication between PEs, most do. The most common form is a 2D grid interconnection network; the PEs are arranged in a 2D array and adjacent PEs can exchange data. The most general form of communication, supported by the Connection Machine, is characterized by a complete graph.

*Author's electronic address and phone: tuck@cs.unc.edu or rrt@cs.duke.edu; (919) 962-1755, (919) 962-1932, or (919) 684-5110.

It allows each PE to compute the number of another PE and send data to or retrieve data from that PE. All PEs may do this in parallel.

Another optional feature of SIMD computers is the ability of each PE to individually compute a local address into its own memory. In architectures without this feature, PE memory can be accessed only by a single global address computed by the host.

These and other optional architectural features of SIMD computers have significant implications for algorithm and program portability. Taking advantage of the optional features provided by a particular architecture may allow a problem to be solved more rapidly than would be possible without those features. But, it also prevents the algorithm (and program) from being used on architectures lacking one or more of those features. The programmer must evaluate this tradeoff and decide which features to use. This in turn dictates the potential portability of the resulting program.

Most SIMD programming languages do not allow programmers to decide which architectural features to use. They simply assume that a particular set of optional features will be available and will be used. Therefore, these languages are useful only on architectures providing those features, and there is no way to take advantage of any additional features provided by an architecture. Porta-SIMD does not have this limitation. It is "optimally portable", which means it lets the programmer decide exactly which architectural features to use. The programmer thereby determines the potential portability of the program. See [Tuck88] for a full discussion of optimal portability.

3 Language Overview

Porta-SIMD extends C++ by adding SIMD parallel data types. Most C++ operators, and a few additional operations, are defined for these new types. C++, in turn, extends C by adding object-oriented features. Familiarity with C is sufficient to begin using Porta-SIMD productively, though knowledge of C++ allows the programmer to take fuller advantage of the features C++ lends to Porta-SIMD.

Porta-SIMD's current and future extensions to C++ can be summarized as follows.

Parallel data types— For each integer and floating-point data type in C, there is a corresponding SIMD parallel type consisting of a data element of that type in each PE. This type is named by prepending `simd_` to the name of the original type. For instance, `simd_int` is the parallel version of the type `int`.

Flexible data type sizes— Each of the `simd_` integer data types may be declared to contain any number of bits from 2 to some implementation-defined maximum. In addition, a boolean `simd_bool` type is provided which requires only a single bit in each PE. These extensions allow better use of the limited local PE memory.

Extended type conversion rules— The C and C++ rules governing type conversions within an expression containing multiple data types have been extended to encompass the new types.

Multiple SIMD architectures— The `simd_` types each have a variety of forms corresponding to the variety of possible SIMD architectures. These types are named by inserting the name of the architecture after `simd`. Variables in statements using optional architectural features must be declared with a type supporting those features.

Flow control— A parallel if-then-else conditional execution mechanism is provided. For architectures which support them, there are functions which allow loops to execute as long as any processor satisfies some logical expression.

The extended features which have already been implemented are described in detail in section 4, and illustrated there with program fragments. Those features remaining to be added are covered in section 8. Of the parallel data types, the integer and boolean types are implemented, but the floating point and character types are not. Only a single optional architectural feature is currently provided: a 2D arrangement of PEs, without inter-PE communication. The most important parallel flow control structure has been implemented, a parallel analog of the `if` statement. However, functions to determine whether any or all PEs satisfy some expression must wait until additional architectural features are supported.

4 Language Definition

Porta-SIMD's extensions to C++ are described below in detail. These extensions have been made as much in the spirit of C++ (and C) as possible, and in most cases existing language rules apply to the extensions as well. These cases will generally not be mentioned, in order to focus on the extensions themselves and on the few exceptions. All the language features described in this section are currently implemented. Future additional extensions are described in section 8. For authoritative references on C++ and C, see [Stroust86] and [HarbisS84], respectively.

4.1 Reserved Words

The identifier `simd` and all identifiers beginning with the prefix `simd_` are reserved by Porta-SIMD and may not be used as program identifiers.

4.2 Architecture Identifiers

Architecture identifiers are short strings that identify a particular set of optional architectural features and thereby define a SIMD architecture. The null architecture identifier, an empty string, identifies the base SIMD architecture with no optional features.

Porta-SIMD currently implements only one optional feature, named **2d**. It specifies that the PEs are arranged in a 2D array and assigned contiguous non-negative integer (x, y) coordinates; it does not provide communication between PEs. (This feature is useful, for example, in Pixel-Planes, which uses these coordinates heavily in its graphics computations.) The architecture identifier **2d** specifies the use of feature **2d**.

One architecture identifier is said to be a subset of another if it specifies a subset of the features by the second architecture identifier. A program may, through its data declarations, use any number of architecture identifiers, provided they are all subsets of a single architecture identifier. This unique identifier is called the program's architecture identifier.

4.3 Types

Porta-SIMD provides parallel signed and unsigned integer data types, and a parallel boolean data type. A parallel object consists of a set of identical elements, one per PE in the SIMD computer. All parallel data type names consist of the string `simd_`, followed by an architecture identifier, followed by a basic type name. If the architecture identifier is non-null, an underscore (`_`) is appended to its name to separate it from the basic type name. Because their names begin with `simd_`, parallel types are also called `simd_` types.

`simd_` types containing the null architecture identifier are called "base" `simd_` types, and are available on all SIMD computers. Therefore, operations are provided for base `simd_` types only if they are efficiently implementable on all SIMD computers. The remaining `simd_` types are called "derived" `simd_` types. Each derived `simd_` type provides all the operations of its corresponding base type, plus additional operations which take advantage of the optional architectural features specified by its architecture identifier.

The type of elements in a `simd_` object is determined by the basic type name contained in the `simd_` type name. The basic types implemented are `int` (signed integer), `unsigned` (unsigned integer), and `bool` (boolean).

4.3.1 Declaration

`simd_unsigned` is an unsigned integer `simd_` base type; `simd_int` is a signed integer `simd_` base type. Objects of both integer `simd_` base types, as well as the corresponding derived types, may be declared to contain any number of bits from 2 to some implementation-defined maximum (which is at least 32). The "length" of an integer `simd_` type is the number of bits it is declared to contain in each element. The length of an integer `simd_` type can be specified as an argument to its constructor, and defaults to at least 16. Here are some example declarations.

```
simd_unsigned a, b(8);
simd_int c(2), d(32);
simd_2d_unsigned e(24), f;
simd_2d_int g;
```

`simd_bool` is an unsigned boolean `simd_` base type. Its length of `simd_bool` and its derived types is always 1.

```
simd_bool a, b;
simd_2d_bool c;
```

4.3.2 Allocation

A `simd_` value is not a single value stored in the host computer, but a set of values stored one per PE. For this reason, `simd_` objects may not be stored in `malloc()`'d memory. Instead, dynamic allocation is performed using the C++ `new` operator, and such allocation is freed with the C++ `delete` operator.

```
simd_int *sipa, *sipb;
simd_2d_bool *s2bp;
sipa = new simd_int;
sipb = new simd_int(17);
delete sipa;
s2bp = new simd_2d_bool;
delete s2bp;
delete sipb;
```

4.4 Storage

The exact amount of storage per PE used for data elements of a `simd_` object is implementation-dependent, subject to the following constraints. The individual values of any particular `simd_` object must be stored identically in each PE. At least as many bits of PE memory must be used to store every `simd_` object as specified in the declaration of that object, though more may be used. All `simd_` objects declared with a particular length must be stored in the same amount of PE memory.

The C operator **sizeof** does not give information about the amount of PE storage used to store **simd_** objects. Rather, it reports the size of the implementation-dependent host object used to store information about the **simd_** object. The **bits** method may be used to find the declared length of any **simd_** object. The following fragment prints 16.

```
simd_int a(16);
printf("%d", a.bits());
```

There is no provision for finding the number of bits actually used to store a **simd_** object.

4.5 Conversions

The type conversion rules governing ordinary C types are extended in Porta-SIMD to incorporate the **simd_** types as naturally as possible. This section's subsections describe each new or extended conversion rule in Porta-SIMD. The first treats the actual representation changes involved in each type conversion. The rest define which conversions are performed under which circumstances.

4.5.1 Representation Changes

The basic rules governing representation changes for scalar integer types apply to conversions among **simd_** integer types as well.

The most basic conversion is between objects of the same type but different lengths. An unsigned **simd_** object is converted to a shorter unsigned **simd_** object by discarding the excess high-order bits. It is converted to a longer unsigned **simd_** object by filling the additional high-order bits with zeros. If signed integer **simd_** objects are represented in two's-complement form (as in all current implementations), the rules are very similar. On conversion to a shorter object, excess high-order bits are discarded. When converting to a longer object, the additional high-order bits are filled with copies of the sign bit.

Conversion between a base **simd_** type and one of its derived types, or between two derived types of the same base types, requires no change of representation.

Conversion between signed and unsigned integer **simd_** types always begins by converting it, still in its own type, to the same length as the destination object. If two's-complement form is used for signed integers, no further conversion is needed between integer **simd_** types.

An integer **simd_** type is converted to a boolean **simd_** type very simply. Every non-zero element takes the boolean value one (true), and every zero element takes the value zero (false). A boolean **simd_** type is converted to an integer **simd_** type by treating it as a very short unsigned object; it is simply zero-extended to the desired length. These rules are consistent with C's treatment of boolean values (e.g., the test expressions of **if**, **for**, and **while** statements, and the value of relational operators such as <).

A scalar integer value can be converted to a **simd_** value of the same type by replication; every element of the resulting **simd_** object has the scalar's value. Conversion of a scalar integer value to a **simd_** value of a different basic type proceeds in two steps: conversion to a **simd_** value of the same type, followed by conversion to the desired **simd_** type. A scalar floating-point value cannot be converted directly to a **simd_** type. (Of course, it can be converted to an integer scalar type first, then converted in a separate step to a **simd_** type.) A **simd_** type cannot be converted to a scalar.

4.5.2 Casting

C allows all legal conversions to be invoked explicitly using a type cast. Unfortunately, Porta-SIMD does not currently provide quite this flexibility. However, any conversion not possible with a cast can be performed with a simple assignment statement (section 4.5.3).

Porta-SIMD allows casts of **simd_** types from boolean and signed integer to unsigned integer. It does not allow the opposite casts, from unsigned integer to signed integer or boolean. It also does not allow casts between signed integer and boolean. These casts must be between **simd_** types with the same architecture identifier, or from a derived to a base **simd_** type.

A **simd_** type may always be cast to its base type or any derived type of its base type.

A scalar value cannot be cast to a **simd_** type.

The casts below are acceptable. Legal casts that change only the architecture identifier are not shown.

```
simd_unsigned      su;
simd_2d_unsigned s2u;
simd_int           si;
simd_2d_int       s2i;
simd_bool          sb;
simd_2d_bool      s2b;

(simd_unsigned) si;
(simd_unsigned) s2i;
(simd_unsigned) sb;
(simd_unsigned) s2b;

(simd_2d_unsigned) s2i;
(simd_2d_unsigned) s2b;
```

However, the casts below are *not* allowed.

```
(simd_2d_unsigned) si;
(simd_2d_unsigned) sb;
```

```
(simd_int) su;
(simd_int) s2u;
(simd_int) sb;
(simd_int) s2b;

(simd_2d_int) su;
(simd_2d_int) s2u;
(simd_2d_int) sb;
(simd_2d_int) s2b;

(simd_bool) su;
(simd_bool) s2u;
(simd_bool) si;
(simd_bool) s2i;

(simd_2d_bool) su;
(simd_2d_bool) s2u;
(simd_2d_bool) si;
(simd_2d_bool) s2i;

(simd_unsigned)    1;
(simd_2d_unsigned) 1;
(simd_int)         1;
(simd_2d_int)      1;
(simd_bool)        1;
(simd_2d_bool)     1;
```

4.5.3 Assignment

All legal conversions can be accomplished through the simple assignment statement. (The simple assigment statement uses the = operator.) Simply write an object of the desired type on the left side of the =, and the value to be converted on the right side.

4.5.4 Usual Unary Conversions

C and C++ define certain "usual conversions" that are performed implicitly during expression evaluation. Their purpose is to convert all the operands of an operator to a common type before performing the operation. Porta-SIMD also converts the operands to a common base simd_ type and common length before performing an operation. However, the usual conversions also ensure that the common type will be one of a very small set of types. Porta-SIMD extends this set of types to include all the simd_ types. As a result, there is no implicit conversion of simd_ types during evaluation of unary operators.

4.5.5 Usual Binary Conversions

The following rules are added to the set of usual binary conversions, and are applied, in order, whenever an op-

1. If one operand is scalar and the other is parallel, convert the scalar operand to its corresponding simd_ type. (Note that only integer scalar types may be mixed with simd_ types as operands of a single operator, since other scalar types cannot be converted directly to a simd_ type.)

2. If the operands have different lengths, lengthen the shorter to the length of the longer. If one of the operands is boolean, this implicitly converts it to an unsigned integer.

3. If one operand is signed and the other is unsigned, convert the signed operand to its unsigned equivalent. (This is what C does with scalars.)

4. If the operand types have different architecture identifiers, convert both to the same type. This may be their common base type, or any derived simd_ type which has an architecture identifier specifying only architectural features present in both operand types. The exact type selected is implementation dependent.

These ensure that the operands have the same base type, length, and architecture identifier.

Of course, while a Porta-SIMD implementation is required to evaluate expressions as if these rules had been used, it is not required to literally perform the specified conversions if the same result can be achieved by a more efficient method.

4.5.6 Function Arguments and Return Values

If an expression appearing as an argument in a function call does not match the type of that argument as declared in the function declaration, it is cast to the declared type of the argument. Similarly, if the expression appearing in a return statement does not match the return type in the function declaration, it is cast to the declared return value type. In both cases, it is an error if the resulting cast is not allowed by Porta-SIMD.

4.6 Expressions

simd_ values may be accessed only by mechanisms explicitly provided by Porta-SIMD. With a few exceptions discussed in this section, all C operators may be applied to simd_ objects. The use of each operator with simd_ types is discussed in detail below, including the precise operand types allowed, whether to apply the usual binary conversions, and the type of the result.

All the C operators implemented by Porta-SIMD perform the same computation as in C and C++, but do

They have the same precedence and associativity as in C and C++.

Unlike C values, all `simd_` values as currently implemented are "lvalues", meaning that they may appear on the left side of an assignment operator. This is an implementation flaw, not a language feature. Do not use as an lvalue any `simd_` expression which would not be a legal lvalue under C's rules. Programs that ignore this rule may not work under future versions of Porta-SIMD.

4.6.1 Enabled and Disabled PEs

Operations and methods that modify `simd_` objects (including temporary objects created implicitly during expression evaluation) are performed only in enabled PEs. Operations without side effects are performed in all PEs, enabled and disabled. Section 4.8 defines when PEs are enabled.

4.6.2 Primary and Postfix Operators

`simd_` objects are not arrays, and cannot be subscripted with the `[]` (subscript) operator to gain access to individual data elements. (This restriction may be relaxed in future versions of Porta-SIMD.) However, an array of `simd_` objects may be declared and subscripted like any other basic type.

Similarly, there are no parallel function calls, so the `()` (function call) operator cannot be applied to `simd_` objects. Of course, scalar functions can return `simd_` objects.

There are also no parallel structures or pointers, so the `.` (direct selection) and `->` (indirect selection) operators cannot be applied to `simd_` values. Scalar structures may contain `simd_` objects, though. The entire `simd_` object, represented by its host-resident "handle" or "descriptor", is contained in the structure.

The postfix operators `++` (post-increment) and `--` (post-decrement) may be applied to integer `simd_` objects only. Unfortunately, Porta-SIMD is limited by its implementation within C++ here, and is unable to provide the proper access-then-operate semantics. Therefore, these postfix operators are currently exactly equivalent to their prefix forms. A future implementation of Porta-SIMD may correctly implement the postfix forms of these operators. Until then, programs should avoid these postfix operators and use only their prefix forms.

4.6.3 Unary Operators

The prefix `++` (pre-increment) and `--` (pre-decrement) operators and unary `-` (negate) operator may be applied to integer `simd_` types, and produce a value of the same type. They may not be applied to boolean `simd_` types.

The `~` (bitwise not) operator may be applied to any `simd_` type, producing a value of the same type.

The `!` (logical not) operator may be applied to any `simd_` type, and produces a boolean `simd_` value with the same architecture identifier as the operand type.

The `sizeof` and cast operators were discussed in sections 4.4 and 4.5.2, respectively.

The `&` (address of) operator may be applied to a `simd_` object, and produces a scalar pointer to the object. (The pointer points to the object's host "handle".) The unary `*` (indirection) operator may be applied to a pointer to a `simd_` object (initialized with the `&` operator); it produces a `simd_` object which may be used like any other `simd_` object. There are no pointers to individual elements of `simd_` objects.

4.6.4 Arithmetic Operators

The binary arithmetic operators include `*` (multiplication), `/` (division), `%` (remainder), `+` (addition), and `-` (subtraction). These operators may be applied to any combination of scalar and parallel integer operands. The usual binary conversions apply to the operands, and the result is the type to which the operands were converted. The corresponding binary arithmetic assignment operators are `*=`, `/=`, `%=`, `+=`, and `-=`. These may be applied to any combination of scalar and parallel integer operands, provided the left operand is parallel. The usual binary conversions apply to the operands, and the result is converted to the type of the left operand.

Boolean `simd_` types may appear as operands to these operators under the following conditions. Only one operand may be boolean, and the other operand must have a parallel integer type. A boolean object may not be the left operand of an arithmetic assignment operator. The conversion rules are not affected by the presence of a boolean operand.

4.6.5 Shift Operators

The shift operators include `<<` (left shift) and `>>` (right shift). These operators may be applied to any combination of scalar and parallel integer operands. The usual unary conversions apply to each operand, but the usual binary conversions do not apply. However, if one operand is scalar and the other is parallel, the scalar operand is converted to a `simd_` operand of its own basic type and the same architecture identifier as the parallel operand. The operands are also converted to a common architecture identifier, using the same rule used by the usual binary conversions. The result has the type, after all conversions, of the left operand. The corresponding shift assignment operators are `<<=` and `>>=`. These operators may be applied to any combination of scalar and parallel integer operands, provided the left operand is parallel. The same conversion rules apply as for the non-assignment forms.

Boolean simd_ types may appear as operands to these operators under the following conditions. Only one operand may be boolean, and the other operand must have a parallel integer type. A boolean object may not be the left operand of a shift assignment operator. The conversion rules are the same as in the absence of a boolean operand, except that a boolean left operand is converted to an unsigned simd_ type the same length as the right operand.

4.6.6 Bitwise Operators

The bitwise operators include & (and), ^ (xor), and | (or). These operators may be applied to any combination of scalar and parallel integer operands. The usual binary conversions apply to the operands, and the result is the same type that the operands are converted to. The corresponding bitwise assignment operators are &=, ^=, and |=. These may be applied to any combination of scalar and parallel integer operands, provided the left operand is parallel. The usual binary conversions apply to the operands, and the result is converted to the type of the left operand.

Boolean simd_ types may appear as operands to these operators under the following conditions. Only one operand may be boolean, and the other operand must have a parallel integer type. A boolean object may not be the left operand of a bitwise assignment operator. The conversion rules are not affected by the presence of a boolean operand.

4.6.7 Relational Operators

The relational operators include < (less than), > (greater than), <= (less than or equal), >= (greater than or equal), == (equal), and != (not equal). These operators may be applied to any combination of scalar integer, parallel integer, and parallel boolean operands. The usual binary conversions apply to the operands. The result is boolean, with the same architecture identifier as the type to which the operands were converted.

4.6.8 Logical Operators

The logical operators include && (logical and), and || (logical or). These operators may be applied to any combination of scalar and parallel integer operands and parallel boolean operands. The conversion rules which apply are the same as for the shift operators. The usual unary conversions apply, but not the usual binary conversions. If one operand is scalar and the other is parallel, the scalar operand is converted to a simd_ operand of its own basic type and the same architecture identifier as the parallel operand. The operands are also converted to a common architecture identifier, using the same rule used by the usual binary conversions. The result is boolean, with the same architecture identifier as the converted operands.

4.6.9 Other C Operators

The ternary ?: (conditional) operator may not have a simd_ first operand. The second and third operands may be simd_ expressions, provided the expressions evaluate to exactly the same type. (Note that this identical-type requirement implies that either both or neither of the second and third operands have a simd_ type.)

The comma operator is the same in Porta-SIMD as in C and C++. It may separate any pair of expressions.

4.7 Special Operations

Porta-SIMD provides a variety of operations related to its unique capabilities. One is the method pe_number, which may be applied to any integer simd_ type object. The result is to place in each element the unique number of the PE holding it.

```
simd_unsigned a;
a.pe_number();
```

Several more methods may be applied to simd_2d_ objects. Recall that the 2d feature allows each PE to compute its (x, y) coordinate. Methods coord_x and coord_y place the x and y coordinates, respectively, into the object to which they are applied. They may be applied only to integer objects (not boolean). The following example computes a multiplication table.

```
simd_2d_unsigned x, y, mult_table;
x.coord_x();
y.coord_y();
mult_table = x * y;
```

The bilinear method computes and stores $ax+by+c$ in each element of the object to which it is applied. The coefficients a, b, and c may be floating-point scalars, and the floating-point result is truncated before being stored. (This is the only floating-point computation currently implemented in Porta-SIMD.) bilinear may be applied only to integer objects. This code fragment computes $x + .5y - 3$.

```
simd_2d_int a;
a.bilinear(1, .5, -3);
```

A display function is also provided for simd_2d_ objects. It displays its arguments, one value per pixel, on a frame buffer if an appropriate one is available. If given one argument, the display is monochrome. If given three, they are treated as the red, green, and blue components of a color image.

```
simd_2d_int gray, r, g, b;
simd_2d_bool black_and_white;
/* initialize variables. */
display(gray);
display(r, g, b);
display(black_and_white);
```

4.8 Flow Control

Porta-SIMD provides a parallel conditional IF statement similar to the scalar if statement. The syntax is

```
IF ( <simd expr> )
  <statements>
ELSE
  <statements>
ENDIF
```

<simd expr> represents any expression which evaluates to a value of a simd_ type. <statements> represents any statements, including nested IF statements properly paired with their nearest following ENDIFs. The ELSE and its following <statements> may be left out.

The <simd expr> is evaluated once, and effectively assigned into a simd_bool temporary variable. During execution of the statements between IF and ELSE (or ENDIF if there is no ELSE), only PEs where the temporary variable is true are enabled. During execution of the statements between the ELSE and ENDIF, the only enabled PEs are those where the temporary variable is false. When a nested IF statement is executed, PEs already disabled before the nested IF statement was encountered remain disabled throughout its execution. Comments in the following example show which PEs are enabled, by giving the expression which must be true in every enabled PE.

```
simd_bool a, b;
/* Initialize a and b. */
IF (a)
  /* a */
  IF (b)
    /* a && b */
  ELSE
    /* a && !b */
  ENDIF
  /* a */
ELSE
  /* !a */
  IF (b)
    /* !a && b */
  ENDIF
  /* !a */
ENDIF
```

It is important to recognize that every statement between IF and ENDIF is always executed, though only in the appropriate PEs. This includes scalar expressions, which are executed in the host as usual. The following code fragment illustrates the potentially surprising results.

```
simd_bool a;
/* Initialize a. */
int x = 0;
IF (a)
  x++;                  /* bad style */
ELSE
  x++;                  /* bad style */
ENDIF
printf("x=%d", x);
```

It prints x=2. It is usually wise not to modify scalar variables within a parallel IF statement. An exception is a scalar variable local to only one part of the IF statement (the statements following either IF or ELSE, but not both).

4.9 Architecture Parameters

Porta-SIMD programs are executed on a SIMD computer with an architecture specified by the program's architecture identifier. By defining UNIX environment variables before executing the program, the user may request specific values for certain architecture parameters. The program will execute on the available machine most closely matching the requested parameters. In some cases the SIMD hardware will constrain the possible parameter values, causing the environment variables to be partially or completely ignored. Default values are used for any parameters not specified by environment variables.

The environment variable SIMD_WORDS specifies the number of words of PE memory. Word size is implementation dependent.

The environment variable SIMD_PES specifies the number of PEs. This variable is ignored in architectures where it is computed from other parameters.

The number of PEs in the x and y dimensions of a 2d architecture may be specified by the SIMD_PES_X and SIMD_PES_Y environment variables, respectively. The total number of PEs is the product of these values. For example, to request 1K PEs in a 32 by 32 square array, use these csh (UNIX C-shell) commands before executing the Porta-SIMD program.

```
setenv SIMD_PES_X 32
setenv SIMD_PES_Y 32
```

Some of these architecture parameters are available within Porta-SIMD programs. The pes method, provided by all simd_ types, returns the number of PEs

```
/* Perform very simple Sieve of Eratosthenes computation to find all
 * prime-numbered PEs.  Return 1 in prime-numbered PEs, 0 in others.
 */
#include <simd_2d.h>           /* Use 2d in order to get display(). */
simd_bool primes()
{
  simd_unsigned number(24);  /* Number to test for prime-ness. */
  simd_bool     is_prime;    /* Flag is true if number is prime */
  number.pe_number();        /* Place PE number in each element. */
  is_prime = 1;              /* Initialize all numbers to true. */
  IF (number < 1)            /* Zero is not prime. */
    is_prime = 0;
  ENDIF

  /* Check against 2..1000.  This is fine for <= 1,000,000 PEs,
   * though not wonderfully efficient. */
  for (int i=2; i <= 1000; i++) {
    IF (((number % i) == 0) && (number > i)) {
      is_prime = 0;          /* Set false: number divides evenly. */
    } ENDIF
  }
  return(is_prime);
}

/* Display prime numbers as bright pixels and non-primes as dark. */
main()
{ display(primes()); }
```

Figure 1: Example program sieve.c. Despite its over-simplifications, this routine will test up to a million numbers for primality in 1000 operations. A similar sequential algorithm would take millions of operations (up to a billion) to perform the same task.

in the SIMD computer. All types with an architecture identifier specifying the `2d` feature provide the `pes_x` and `pes_y` methods, which return the number of PEs in the x and y dimensions, respectively. This fragment prints the size of the SIMD computer's PE array.

```
simd_2d_bool a;
printf("PE array is %u by %u",
       a.pes_x(), a.pes_y());
```

5 Example Programs

It is difficult to find real programs short and clear enough to make good examples. Although the problems solved by the example programs in this section may seem over-simplified, they are still useful to illustrate the use of Porta-SIMD. The example programs are described in their comments. The first example, `sieve.c`, is contained in figure 1. The second, `square.c`, is shown in figure 2.

6 Using Porta-SIMD

The mechanics of writing, compiling, and linking a Porta-SIMD program involve only a minor addition in each of these steps, compared to an ordinary C++ program.

Every Porta-SIMD program source file must begin by `#include`'ing a Porta-SIMD definition file. If the file uses no optional architectural features, the file to include is `<simd.h>`; otherwise it is `<simd_`*archid*`.h>`, where *archid* is an architecture identifier specifying the optional features required. (Architecture identifiers are discussed in section 4.2.) This was shown in the previous section's example programs.

When compiling and linking a Porta-SIMD program by hand, as opposed to using `make`, begin by defining the shell variable `PORTA_SIMD`. At UNC, it should have the value `/home/common/tuck/simd`. Using the `csh`, this is done with the following command.

```
setenv PORTA_SIMD \
    /home/common/tuck/simd
```

```
#include <simd_2d.h>

/* square accepts the upper left and lower right corners of a square aligned
 * with the coordinate axes.  Returns 1 in each PE inside the square, 0 in
 * each PE outside. */
simd_2d_bool square(int x_ul, int y_ul, int x_lr, int y_lr)
{
  simd_2d_bool inside;
  simd_2d_unsigned x, y;
  inside = 1;
  x.coord_x();
  y.coord_y();
  inside &= (x > x_ul);
  inside &= (y > y_ul);
  inside &= (x < x_lr);
  inside &= (y < y_lr);
  return(inside);
}

/* main displays a white square on a black background. */
main()
{ display(square(2,6,24,57)); }
```

Figure 2: Example program square.c.

(To improve legibility, this and other commands are shown broken into multiple lines with a \ (backslash). They may be typed as shown, but it is more natural to type them as a single line without the \.)

The command which compiles each program source file into an object file must predefine two symbols using the `-D` compiler option. The first symbol identifies the type of SIMD hardware on which the program will run. This may be `seq` (to simulate parallel hardware using a sequential computer), `pxpl4` (to use Pixel-Planes 4), or `cm` (to use the Connection Machine). The second symbol identifies the host computer type. This may be `sun2`, `sun3`, `sun4`, `vax`, or `pxpl4gp` (for the Pixel-Planes 4 Graphics Processor front-end). Also, include this compiler option: `-I${PORTA_SIMD}/include`. For example, the following command compiles the file `square.c`, an example program in the previous section, for sequential execution on a Sun-4 host.

```
CC +e0 -g -I${PORTA_SIMD} \
    -Dseq -Dsun4 -c square.c
```

When linking the object files to generate an executable, specify linkage with a Porta-SIMD library using the `-l` option. If the program uses no optional architectural features, use the `simd` library. Otherwise use library

`-L${PORTA_SIMD}/lib/`*simd_hw*`/`*host*. Here, *simd_hw* and *host* are the symbols defined with `-D` in the compile command discussed above. The command below links `square.o`, compiled above, creating an executable program `square`.

```
CC +e1 -g -L${PORTA_SIMD}/lib/seq/sun4 \
    -o square square.o -lsimd_2d
```

Figure 3 is a `Makefile` to compile and link the example programs presented in section 5. The directory `${PORTA_SIMD}/ex` contains copies of the example programs and `Makefile`.

7 Implementation

Porta-SIMD is currently implemented strictly within the C++ language. Its extensions to C++ are provided by include files and libraries, and Porta-SIMD programs are compiled by `CC` (the C++ compiler) rather than by a Porta-SIMD compiler. This has both advantages and disadvantages.

One advantage is that it was not necessary to write a compiler. This has sped the implementation process, and will continue to do so. Another is the guarantee it provides that the scalar portion of Porta-SIMD is exactly

```
# SIMD_HW is name of a real SIMD hardware machine which will execute the
# parallel part of Porta-SIMD programs.  Choices are:
#       pxpl4     -- Pixel Planes 4
#       cm        -- Connection Machine
#       seq       -- ordinary sequential machine (simulates parallel computer)
SIMD_HW = seq

# TARGET_CPU is the name of a real sequential computer which will execute
# the sequential part of Porta-SIMD programs.  Choices are:
#       sun[234] -- Sun-[234]   (SIMD_HW supported: seq)
#       vax      -- DEC VAX     (SIMD_HW supported: seq, pxpl4, cm)
#       pxpl4gp  -- PxPl4 GP    (SIMD_HW supported: pxpl4)
TARGET_CPU = sun4

# Define home of Porta-SIMD, and its include and library directories.
PORTA_SIMD = /home/common/tuck/simd
INC_DIR = ${PORTA_SIMD}/include
LIB_DIR = ${PORTA_SIMD}/lib/${SIMD_HW}/${TARGET_CPU}
DEF = -D${SIMD_HW} -D${TARGET_CPU} #Porta-SIMD include files need these symbols
CC  = CC                  # Use AT&T cfront C++ compiler
OPT = -g                  # -g to include debugging info in compiled program
# Use +e0 to compile, and +e1 to link, to reduce size of object files.
CFLAGS      = +e0 ${OPT} -I${INC_DIR} ${DEF}
CFLAGS_LINK = +e1 ${OPT} -L${LIB_DIR}            # Use -Bstatic w/SunOS 4.0 only

PROGRAMS = sieve square
all:    ${PROGRAMS}
sieve:  CHECK sieve.o
        ${CC} ${CFLAGS_LINK} -o $@ sieve.o -lsimd_2d
square: CHECK square.o
        ${CC} ${CFLAGS_LINK} -o $@ square.o -lsimd_2d
```

Figure 3: Makefile to compile and link the example programs presented in section 5 for sequential execution on a Sun-4.

immediately be available in Porta-SIMD. Another advantage is enhanced implementation portability—the C++ compiler has been widely ported.

There are two primary disadvantages to not writing a separate Porta-SIMD compiler. First, it places strict limits on lexical and semantic extensions. For example, the keyword `simd` would ideally be added as a storage class modifier, instead of adding `simd_` as a prefix to the type name. The architecture identifier used to name variant `simd_` classes would also be separate from the type name if there were a dedicated compiler. Similarly, parallel conditional flow would be provided by extending the semantics of the `if` statement instead of adding the `IF` statement. The second disadvantage is in execution performance. Porta-SIMD currently must implement its `simd_` types as C++ classes. As a result, expression evaluation generally requires more operations and more memory than if Porta-SIMD were implemented by its own compiler.

8 Planned Features

The Porta-SIMD implementation is just reaching sufficient maturity to support real users. However, many important features are not yet implemented. Some of these are described below.

8.1 Integer and Character Types

Some fixed-length base types may be implemented, for example `simd_int8`, `simd_int16`, and `simd_int32`. It would be possible to evaluate expressions involving only one of these types faster than expressions of the current types.

The `simd_char` base type is not currently implemented. However, a `simd_unsigned` or `simd_int` object declared with a length of 8 bits is an acceptable substitute. `simd_char` may be implemented as a typedef to `simd_unsigned8`, if the latter is implemented.

8.2 Floating-point Types

The design of Porta-SIMD includes SIMD extensions of the C floating-point types, analogous to the extensions implemented for integer types. However, they are not scheduled for implementation in the near future.

8.3 Architecture Identifiers

Porta-SIMD will eventually support all the SIMD architectures in the taxonomy presented in [Tuck88]. The order in which the optional architectural features are added will depend on user needs. When C++ 2.0 is available with multiple inheritance, it will be possible to select any combination of optional features.

8.4 Expressions

A mechanism will be provided very soon to access individual data elements of a `simd_` object on architectures which support this. This will be followed by a mechanism for specifying inter-PE communication.

8.5 Flow Control

In some architectures, it is possible to determine whether all PEs are enabled, or whether all are disabled. These scalar logical values can be useful in scalar flow control structures. Functions providing them will be implemented soon.

9 Acknowledgements

User input has been and will continue to be critical to the successful evolution of Porta-SIMD. Greg Turk has been an especially valuable user in this regard.

The development and documentation of Porta-SIMD were supported by the Pixel-Planes Project, Henry Fuchs and John Poulton, P.I.s, and its grants: National Science Foundation grant #MIP-8601552, Defense Advanced Research Projects Agency order #6090, Office of Naval Research contract #N0014-86-K-0680; and by the GRIP Project, Frederick Brooks, P.I., under National Institutes of Health grant #RR 02170. Access to a Connection Machine was provided by the Advanced Computing Research Facility at Argonne National Laboratories, under grants NSF-ASC-8808327 and DOE-W-31-109-ENG-38.

References

[EylesA*87] John Eyles, John Austin, Henry Fuchs, Trey Greer, and John Poulton. Pixel-Planes 4: a summary. In *Proceedings of Eurographics '87 Second Workshop on Graphics Hardware*, 1987.

[FuchsG*85] Henry Fuchs, Jack Goldfeather, Jeff P. Hultquist, Susan Spach, John D. Austin, Frederick P. Brooks, Jr., John G. Eyles, and John Poulton. Fast spheres, shadows, textures, transparencies, and image enhancements in Pixel-Planes. *Computer Graphics*, 19(3):111–120, July 1985. (Proceedings of SIGGRAPH '85).

[HarbisS84] Samuel P. Harbison and Guy L. Steele Jr. *C: A Reference Manual.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1984.

[Hillis85] W. Daniel Hillis. *The Connection Machine. MIT Press Series in Artificial Intelligence*, The MIT Press, Cambridge, MA, 1985.

[KernigR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978.

[Stroust86] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Publishing Company, Reading, MA, 1986.

[Thinkin87] *Connection Machine Model CM-2 Technical Summary.* Technical Report HA87-4, Thinking Machines Corporation, April 1987.

[Tuck88] Russ Tuck. An optimally portable SIMD programming language. In *Frontiers '88: Second Symposium on the Frontiers of Massively Parallel Computation*, October 1988. Also available as University of North Carolina at Chapel Hill, Department of Computer Science, Technical Report TR88-048.