# Processing in Memory: The Terasys Massively Parallel PIM Array

**Maya Gokhale**
***David Sarnoff Research Center****

**Bill Holmes and Ken Iobst**
***Supercomputing Research Center***

**The work reported here was done while the author was at the Supercomputing Research Center, Bowie, Maryland.*

**The PIM prototype in a workstation environment delivers supercomputer performance at a fraction of the cost. The next step is to incorporate PIM chips into Cray-3 memory.**

SIMD processor arrays provide superior performance on fine-grained massively parallel problems in which all parallel threads do the same operations most of the time. However, this fine-grained synchrony limits the application space of SIMD (single instruction, multiple data) machines. If there are many alternative data-dependent actions among the parallel threads, the total execution time is the *sum* of the alternatives rather than the maximum single-thread execution time. Additionally, if the application is not inherently load-balanced, performance can degrade seriously: Most of the processors finish their work quickly and become idle, while a few processors end up with the lion's share of the work. Thus, the economics of purchasing a high-performance computer often dictate giving up peak performance on a small application set (massively parallel SIMD) in favor of more modest improvement over a larger range of applications (general-purpose MIMD).

Ideally, one platform would provide the advantages of a SIMD array for applications well suited to SIMD processing without penalizing less well-structured applications. That is, a SIMD array would be integrated into the architecture of a high-performance computer, so some applications could benefit from the SIMD array while others could still run on the more flexible high-performance "carrier" (host). Researchers at the Supercomputing Research Center (SRC) have carried this notion one step further. We have integrated the SIMD array so closely into the architecture of the high-performance host that the hardware comprising the SIMD array can be used either as a SIMD processor array or as additional conventional memory.[1]

SRC researchers have designed and fabricated a processor-in-memory (PIM) chip, a standard 4-bit memory augmented with a single-bit ALU controlling each column of memory. In principle, PIM chips can replace the memory of any processor, including a supercomputer. To validate the notion of integrating SIMD computing into conventional processors on a more modest scale, we have built a half dozen Terasys workstations, which are Sun Microsystems' Sparcstation-2 workstations in which 8 megabytes of address space consist of PIM memory holding 32K single-bit ALUs. We have designed and implemented a high-level parallel language, called data-parallel bit C (dbC), for Terasys and demonstrated that dbC applications using the PIM memory as a SIMD array run at the speed of multiple Cray-YMP processors. Thus, we can deliver supercomputer performance for a small fraction of supercomputer cost.

Since the successful creation of the Terasys research prototype, we have begun work on processing in memory in a supercomputer setting. In a collaborative research project, we are working with Cray Computer to incorporate a new Cray-designed implementation of the PIM chips into two octants of Cray-3 memory.

0018-9162/95/$4.00 © 1995 IEEE

## TERASYS WORKSTATION

A Terasys workstation (see Figure 1) consists of

- a Sun Sparc-2 processor,
- an SBus interface card residing in the Sparc cabinet,
- a Terasys interface board, and
- one or more PIM array units.

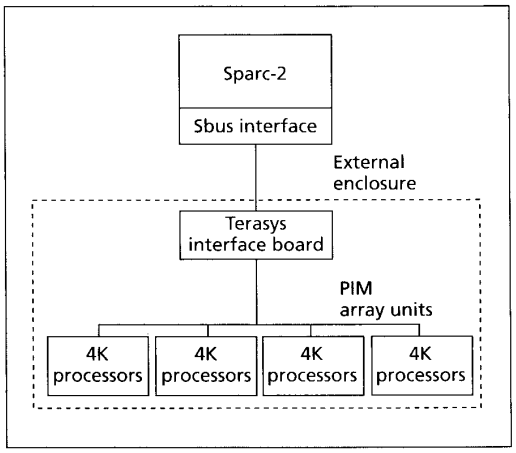


**Figure 1. A 16K processor Terasys workstation.**

The notion of computing in memory has been with us for several decades. For example, Stone[1] proposed a logic-in-memory computer consisting of an enhanced cache memory array that serves as a high-speed buffer between CPU and conventional memory.

More recently, a group at the University of Toronto has designed a computational RAM—conventional RAM with SIMD processors added to the send amplifiers in a 4-Mbit DRAM process.[2]

In the commercial realm, Hitachi markets a video DRAM chip with limited processing in the memory. This chip, the HM53642 series, is a 65K × 4-bit multiport CMOS video DRAM with simple logic operators on each of the 4 bit-planes.

Our data-parallel C is similar to MasPar's MPL[3] and Thinking Machines' C*.[4] Wavetracer's MultiC[5] had user-defined bit lengths, but did not have the bit extraction/insertion or generic bit-length features. Our generic SIMD interface is patterned after the CM-2 Paris instruction set.[5]

### References


1. H.S. Stone, "A Logic-in-Memory Computer," *IEEE Trans. Computers*, Jan. 1970, pp. 73-78.
2. D.G. Elliott, W.M. Snelgrove, and M. Stunn, "A Memory SIMD Hybrid and its Application," *Proc. Custom IC Conf.*, IEEE, Piscateway, N.J., 1992.
3. *MasPar Application Language (MPL) Reference Manual*, MasPar, Pub-9302-0000 01/90, 1990.
4. *The MultiC Programming Language*, Wavetracer, Pub-00001-001-1.00, 1994.
5. *Paris Reference Manual*, Thinking Machines, Cambridge, Mass., Tech. Rep. Version 5.0, 1989.


The Terasys interface board and PIM array units fit into an external enclosure. The system can accommodate up to eight PIM array units, with 4K processors per array unit, giving a maximum configuration of 32,768 PIM processors.

A data parallel program resides on the Sparc. Conventional sequential instructions are executed by the Sparc. Operations on data parallel operands are conveyed as memory writes through the Terasys interface board to PIM memory and cause the single bit ALUs to perform the specified operations at the specified row address across all the columns of the memory.

An instruction written to the PIM array is actually a 12-bit address in a 4K entry table of horizontal microcode, which is generated on a per-program basis. The selected table entry is the real instruction, which is sent from the Terasys interface board to the PIM array units.

For global communications among the processors there are three networks: global OR, partitioned OR, and parallel prefix network. Level 0 of the PPN serves as a bidirectional linear nearest-neighbor network.

In the following sections, we describe the PIM chip and single-bit processor, the three interprocessor communication networks, the PIM array unit, and the Terasys interface board.

### Processor-in-memory chip

The PIM integrated circuit, with slightly over one million transistors on 1-micron technology, contains 2K × 64 bits of SRAM, 64 custom-designed single-bit processors, plus control and error detection circuitry.[2] Figure 2 shows the PIM chip. Solid lines outline conventional memory components, while dotted lines delimit the added PIM circuitry.

In conventional memory mode, the chip is configured as a 32K × 4-bit SRAM with an 11-bit row address and a 4-bit column address. In response to the row address signals, it loads the selected row into a 64-bit register. The column address selects a 4-bit nibble (one of 16 nibbles composing the 64-bit register).

To operate in PIM mode, the chip is activated by one of 25 command bits from the Terasys interface board. Command mode initiates an internal row operation with 64 processors operating on one of 2K rows of memory. The processors can perform a local operation and optionally read/write the global OR, partitioned OR, and parallel prefix lines.

### PIM processor

The PIM processor is a bit-serial processor that accesses and processes bits to/from a 2-Kbit column of attached memory. Functionally, the processor is divided into upper and lower halves. The upper half performs the actual computations on the data, while the lower half performs routing and masking operations. Data is brought in from the processors' attached memory through the load line, circulates through the logic as specified by the program, and is written back to memory via the store line. Figure 3 is a simplified diagram of the processor.

Three primary registers, denoted *A*, *B*, and *C*, supply data to the ALU. The registers have three primary input lines for receiving data, each of which also can be inverted to receive the logical NOT of that input.

At each clock cycle, the pipelined ALU can either load data from memory or store data to memory, but not both at the same time. Also, on each clock cycle, the ALU produces three outputs that can be either selected for storage (under mask control) or selected for recirculation. Additionally, data can be sent to other processors via the routing network.

The processor can input data through a multiplexer (MUX) from either the parallel prefix network, the global OR network, the partitioned OR network, or the internal mask/register control (see the "Network and mask" line at lower right in Figure 3).

### Indirect addressing

Indirect addressing signifies an operation $A[i]$, where each processor has its own instances of $A$ and $i$. To perform this operation, the processors must access different memory locations simultaneously.

We decided against hardware support for indirect addressing in the current PIM implementation for two major reasons. First, introducing indirect addressing registers on a per-processor basis would reduce the number of processors per chip. Second, the error detection and correction circuitry, which operates on a row basis, would not operate correctly in the presence of per-processor indirect addressing registers. Thus, we opted for indirect addressing through microcode. To make this operation efficient, we have the compiler emit array-bound information as part of an indexing operation. This allows the microcoded indexing subroutine to limit the number of locations to query to the number of array elements rather than the entire 2K bits.

In indirect addressing, each processor holds its private instance of an array. The application can also use a single array $A$ in Sparc conventional memory, which is shared by all PIM processors. Each processor can index this shared table with its unique index $i$ that is in PIM memory. An optimized microcode routine, which is a factor of 8 faster than the more general per-processor table lookup, is provided for this operation in a Fortran-based library called Password (Parallel SIMD Workbench).[3]

### Interprocessor communication

A simple linear interconnect, augmented by global OR, partitioned OR, and parallel prefix networks, has been incorporated into the PIM design. This allows the PIM processor to

- compute a global OR or partitioned OR signal in one tick,
- send one bit of data to a left or right neighbor in one tick, and
- perform parallel prefix operations in log (number-of-processors) ticks per bit.

**GLOBAL OR.** The logical OR network combines signals from all processors and returns a single bit result to the host. GOR is performed across all 32K processors in hardware. This signal is used to conditionally control instruction execution: One of two instructions can be selected based on the value of the GOR signal. The microcode programmer has access to the GOR signal at the host processor through a GOR register containing the last 32 GORs generated by the system.

**PARTITIONED OR NETWORK.** In contrast to the GOR, which performs only many-to-one communication, the POR network can be used for many-to-one or one-to-many communication among groups of processors. The 32K processors in the Terasys workstation can be partitioned, starting with 2 processors per partition and increasing in

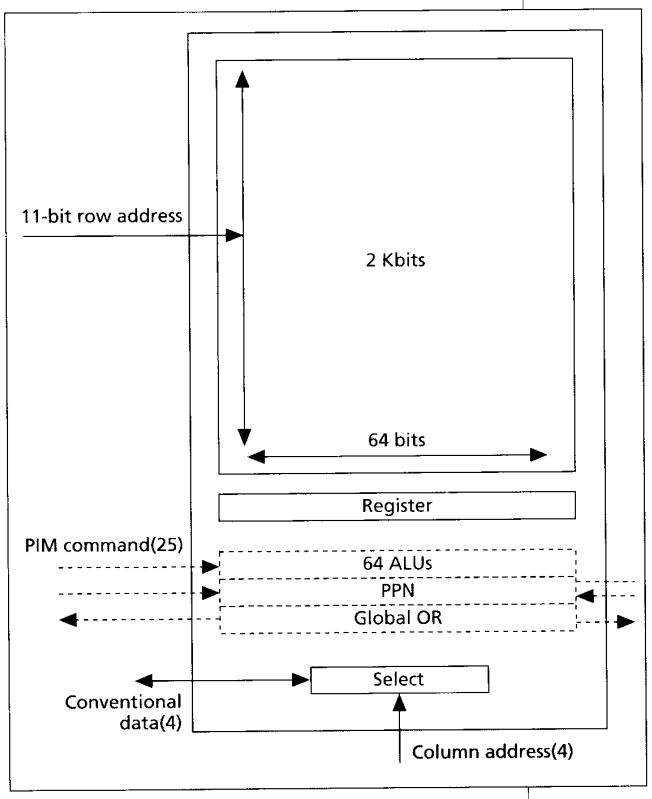


**Figure 2. A processor-in-memory chip.**

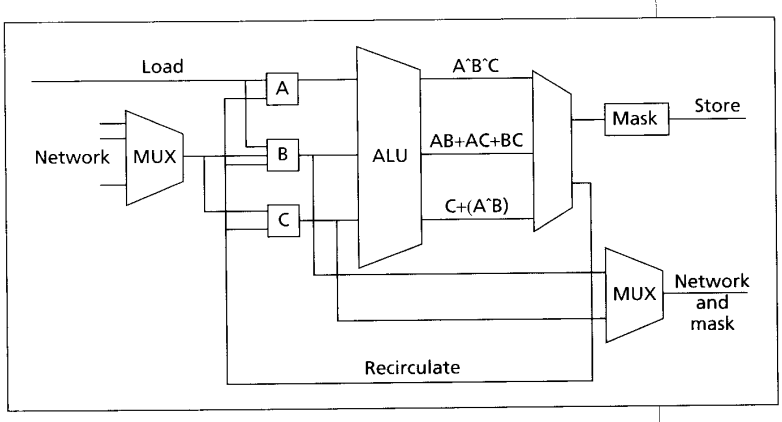


**Figure 3. A processor-in-memory processor.**

powers of 2 up to 32K processors per partition. Within a partition, an OR tree is formed from all the processors in the partition, and the result is fed back to them. POR hardware support is provided for groups of 64, 128, 256, and 512 processors. Other groupings are implemented in software.

The POR network is useful in matrix-pivoting operations. Disjoint matrices in different POR partitions can have different pivot points. The pivot information can be broadcast from one processor in each partition to all processors in that same partition. This is accomplished by using the internal mask to select a particular processor in each partition and then performing the POR operation, where only the signal from the selected processor is moved up the OR tree and fed back to the other processors in the partition.

**Each of eight banks of PIM memory contains eight PIM chips—a total of 64 chips or 4,096 PIM processors per array unit.**

**PARALLEL PREFIX NETWORK (PPN).** The PPN consists of 15 levels, settable by the programmer. It can be used for nearest-neighbor communication and for linear scan operations, which are useful for accumulating partial results such as sums or other associative operations.

At level 0, the PPN is a one-dimensional communications path used to transfer data among the processors, so that all processors can send to either the left or right neighbor in one tick. At higher levels, the PPN is a few-to-many communication path.

At level 0, each selected processor $i$ sends data to its immediate neighbor to the left ($i-1$) or to the right ($i+1$). Levels 1-15 send data to the left only. At level 1, the PPN enables even-numbered processors $i$ to send data to two left neighbors $i-1$ and $i-2$. At PPN level 2, processors 0, 4, 8, ... can send to the four left neighbors. Succeeding PPN levels expand at a power of 2, until at level 15, one data source broadcasts to all 32K processors.

The PPN network can be set to either the toroidally wrapped mode, where processor 0 gets its data from processor 32,767, or to "fill" mode, where processor 0 input data is fixed at 0 or 1. Three other levels of toroidally wrapped data are also supported at the 8-bank, bank, and chip levels.

Levels 0-9 of the PPN, which correspond to the PPN up to the memory bank (see "PIM array unit" below), are implemented in hardware. Levels above the memory bank are implemented in software.

**GENERAL DATA MOVEMENT.** Unstructured interprocessor data communication is through the host processor. For general any-to-any communication, PIM data values are sent to the host, transposed from bit-serial to bit-parallel representation, permuted in conventional memory, transposed again, and sent back to the PIM memory. This operation is time-consuming on the Terasys workstation. In the new vector supercomputer PIM array, where this operation will use the gather-scatter hardware, performance will be comparable to general communication networks in massively parallel processors such as SP-2 or MasPar.[4]

### PIM array unit

A PIM array unit contains eight banks of PIM memory spread over two boards. Each bank contains eight PIM chips—a total of 64 chips or 4,096 PIM processors per array unit. In addition to housing the 64 PIM chips, the array unit performs extended global OR and parallel prefix functions and provides a three-level partitioned OR tree.

### Terasys interface board

Read and write operations to PIM memory are sent to the Terasys interface board. High-order bits of the address field tell the interface logic whether the operation is a normal memory operation or a PIM command pair. If they indicate PIM command mode, the interface board splits the combined commands into two individual commands. This command pairing allows a doubling of the command transfer rate. The Sparc-2 can transfer one command pair every 200 nanoseconds. The command issue rate to the PIM chips is 100 ns per command.

Each command is an index into a 4K lookup table. The lookup table contains 25-bit microcode instructions. Thus, out of a possible $2^{25}$ instructions, each program can use up to 4,096. The Twist (Terasys Workstation Interface Software Tools) microcode assembler generates the lookup table automatically. The table is loaded into the interface board when a Terasys program is initiated.

This board also provides PIM timers, selects one of two PIM commands based on the global OR signal's current value, and registers 31 bits of global OR history.

## DATA-PARALLEL BIT C

The Terasys programming language dbC is an ANSI C superset based on a tightly synchronous (variously called data parallel, "single thread of control," or SIMD) parallel programming model. In dbC, arbitrary-length bit streams are treated as first-class objects in the parallel domain.[5,6] Our goals in designing dbC were to support

- a simple, easy-to-use programming model so that programmers can rapidly become productive on Terasys and other PIM-based systems;
- efficient data-parallel computation on SIMD machines; and
- computation on arbitrary bit-length integers in the parallel domain.

### Data-parallel extensions

dbC data parallel extensions follow language conventions set by C* and MPL.

**PARALLEL DATA.** dbC adds a new memory attribute to C data declarations, the keyword "poly." A poly variable is instantiated on each processor, so that a reference to a poly variable in an expression is an aggregate reference that affects each active processor. dbC supports all the standard C data types in the parallel domain. It also supports parallel pointers. Pointers can point between serial and parallel domains and within the parallel domain, providing language support for indirect addressing. Pointers may not point from the memory of one processor to the memory of another; interprocessor communication intrinsics must be called to access data from another processor.

**PARALLEL EXPRESSIONS.** Polys can be used in C expressions just as normal C (mono) variables. An expression *a* op *b* is parallel if either *a* or *b* is parallel. If the other operand is serial, it is promoted to parallel, and the operation is performed in the parallel domain.

dbC has infix reduction operators such as |= (reduce with OR operator) that, when applied to a parallel expression, yield a serial value. Reductions are computed using the global OR and parallel prefix networks.

```
poly unsigned x:4;  /* 4-bit logical variable x */
typedef unsigned poly int33:33;  /* 33-bit logical, user-defined type */
poly int y: 1000;  /* arbitrarily large variables are supported */
typedef poly struct
  { int33 A[50];
    int33 B;
    char c;
  } S;  /* poly structure */
```

**Figure 4. Parallel variables in dbC (data-parallel bit C), the Terasys high-level programming language.**

**PARALLEL CONTROL CONSTRUCTS.** dbC extends the standard C guarded control constructs (if, while, do, and for) to the parallel domain. The <guard> controlling the statement determines whether the statement is serial or parallel. If the <guard> is parallel, the statement is parallel. The escape statements *break*, *continue*, and *return* have parallel versions. A break or continue in a parallel loop is parallel. A return from a function returning a parallel result (even *poly void*) has parallel semantics. Gotos are not allowed in parallel constructs.

In addition to the C control constructs, dbC provides a masked *where* construct and an *all* block. Parallel code in an all block is executed by all processors regardless of previous processor activity. An all block may contain any serial or parallel code, including parallel if/where and parallel loops.

```
poly x:10, y:11
x[4:8] = y[0:4];      /* start/end index */
x[4+:5] = y[0+:5];    /* bit length notation */
```

**Figure 5. Bit insertion and extraction in dbC.**

```
void func(poly x:?)
{ ... }

  ...
  poly y_5:5, z_7:7;
  func(y_5);          /* pass 5 bits to func */
  func(z_7);          /* pass 7 bits to func */
  ...
```

**Figure 6. Generic bit-length parameters.**

## Bit extensions

The generality of dbC's bit constructs facilitates computation over arbitrarily sized data.

**BIT-LENGTH SPECIFIERS.** dbC extends C's bit-field feature to allow any parallel integer variable (or integer component of a structured variable) to specify a length. Although the syntax of the bit-length specifier is identical to the C bit field, the semantics are quite different. The C bit field represents a compromise between the desire to access bits and the difficulty of supporting efficient bit access in the word-oriented domain. A bit field declared to be 6 bits long in C might really occupy a full memory word (32 or 64 bits). A variable or structure component declared as "poly int:6" in dbC is guaranteed to use exactly 6 bits of parallel memory.

When two parallel operands are used in a binary operation, their bit lengths determine the number of bit operations required to compute the result. Thus, the user controls both

- storage allocation at the bit level (particularly useful on Terasys, where each processor has only 2,048 bits of memory) and
- the bit-serial complexity of operations, since the bit lengths of the operands determine the number of suboperations required to perform an operation.

Figure 4 illustrates poly declarations with bit lengths.

**BIT EXTRACTION AND INSERTION.** dbC also has two bit-oriented operators, bit insertion and bit extraction, illustrated in Figure 5. A parallel variable $x$ may be indexed $x[a:b]$, where $a$ indicates the starting bit position and $b$ the ending bit position, inclusive. On the right-hand side of an assignment, this notation means that $b - a + 1$ bits are extracted from $x$ starting at bit offset $a$. When this "slice" notation is used to index a variable on the left-hand side, bit insertion is performed. As a shorthand, $x[a:]$ is equivalent to $x[a:a]$, meaning one bit at offset $a$ is accessed. An alternative notation is also illustrated in the figure. The +: infix operator in an index expression $a+:b$ means that the starting index is $a$ and the bit length is $b$.

**GENERIC BIT LENGTHS.** To make it easier to write subroutines that can operate on parameters of any bit length, dbC provides a generic bit-length construct. The bit length of a parameter to a function may be "?," indicating that the length is to be determined at runtime at each invocation of the function. Figure 6 illustrates the use of generic bit length in the parameter $x$ to function func. The function is called twice, first with a parameter of bit length 5, and the second time with a parameter of bit length 7.

**COMMUNICATION.** The interprocessor communication intrinsic DBC_net_send transfers data between nearest neighbors in a linear topology at level 0 in the parallel prefix network. In addition, the DBC_send intrinsic can be used for any-to-any communication; this general data movement between processors is done through the Sparc conventional memory.

Intrinsics are also provided to transfer data between serial and parallel domains, both one processor at a time and

with block transfer. The intrinsics are overloaded, so the same name is used (for example, DBC_read_from_proc(x, pe_number)) regardless of the data type of $x$.

### Example

Figure 7 illustrates both the data-parallel and the bit-oriented extensions to dbC. This function computes the greatest common divisor of two 68-bit parallel variables. The algorithm uses a parallel while loop to iteratively divide int2 by int1. As int1 and int2 become smaller at each iteration, the number of bits required for time-consuming bit-serial division is also reduced.

The outer while loop is a parallel loop. This loop iterates until int1 is 0 on every processor; processors fall out of the loop, becoming inactive, as their int1 values become 0. The inner for loop is serial because the test expression bits_per_divide > 0 is serial. The inner loop determines the minimum number of bits required for the next divide. As the algorithm proceeds, fewer and fewer bits are used for each divide operation.

Thus, as soon as a processor finds a nonzero bit in either int1 or int2 (the OR-reduce expressions), the for loop terminates. The intrinsic DBC_Divmod computes the quotient and remainder of int2 divided by int1. In this call, only the minimum number of bits is used for the division (real_isize + 1 bit).

The assignments into int1 and int2 illustrate both bit extraction and bit insertion. Real_isize number of bits are extracted from int1 and rem and inserted into int2 and int1.

```
#define ISIZE 68

typedef poly unsigned wide_int:ISIZE;

wide_int GCD(wide_int int1, wide_int int2)
  {
    wide_int result, rem;
    int bits_per_divide, real_isize;

    while (int1 !=0)
      {
        for (bits_per_divide = ISIZE-1; bits_per_divide>=0; bits_per_divide—)
          {
            /* position past leading zeros of int1 and int2 to
                 reduce the complexity of divide. */

            real_isize = bits_per_divide;
            if (|=(int1[bits_per_divide:]) != 0) break;     /* OR-reduce across ... */
            if (|=(int2[bits_per_divide:]) != 0) break;     /* ... active processors */
          }
        DBC_Divmod(&result, &rem, int2[0:real_isize-1], int1[0:real_isize-1]);

/* clear top bits of int2 and int1 */
        if (real_isize < ISIZE)
          {
            int2[real_isize:ISIZE-1] = 0;
            int1[real_isize:ISIZE-1] = 0;
          }

        int2[0:real_isize-1] = int1[0:real_isize-1];
        int1[0:real_isize-1] = rem[0:real_isize-1];
      }

    return result;
  }
```

**Figure 7. A greatest-common-divisor function in dbC.**

### Twist microcode library

dbC translates all code involving parallel operands into a generic three-address memory-to-memory SIMD assembly code. The generic SIMD is then mapped to the various platforms on which dbC programs run, including the CM-2, Splash-2,[7,8] and workstation clusters, in addition to Terasys. The Terasys Workstation Inter-face Software Tools (Twist) microcode library maps the generic SIMD instruction set to Terasys.

This subroutine library[9] performs basic operations such as

- parallel memory allocation and release,
- basic arithmetic and logical operations on poly operands of arbitrary bit length,
- random parallel-number generation,
- indexed address calculation and indirect addressing,
- global combining and broadcasts,
- parallel prefix operations such as scans and segmented scans,
- nearest-neighbor communication and generalized communication, and
- data transfer between host memory and PIM memory.

So that knowledge available to the compiler about operands can be passed to the runtime system, there are often several forms of an instruction. For example, there are eight variations to the add instruction for each of the basic data types (unsigned integer, signed integer, and floating point). This allows the microcode library to optimize for such properties as

- number of operands, because if the destination is the same as one source, only two operands are required in a binary operation;
- the number of bit lengths being passed, either 1, 2, or 3;
- whether or not an operand is a constant being broadcast as part of the instruction; and
- whether the instruction is to be masked or performed unconditionally, since unmasked instructions are faster. (If a store instruction is conditional, the microcode must load and save the original value at the store address, so that when a value is ready to be stored, inactive processors will store the original value back. This is a slower sequence than an unconditional store, in which the original value does not have to be read.)

The Twist microcode assembler extracts those microinstructions required by the program and builds the instruction lookup

table. The lookup table is downloaded to the Terasys interface board upon program initiation. In addition to the microcode library, we provide a

- dbC simulator
- microcode emulator, and
- Fortran-based development system with the same functionality as Twist, the Parallel SIMD Simulation Workbench (Passwork).[3,10]

## APPLICATIONS AND PERFORMANCE

The Terasys workstation—fully populated with 32K processors and assuming an instruction issue rate of 100 ns—delivers $3.2 \times 10^{11}$ peak bit operations per second. Microcoded applications have achieved and, in one case, surpassed this theoretical peak. (The program surpassing the peak uses superscalar techniques that do two bit operations in one tick by using both the upper and lower parts of the processor.)

### Applications profile

Applications such as DNA sequence match, low-level image processing, and tridiagonal solvers and other matrix algorithms are well suited to the Terasys workstation. In general, problems having

- largely independent computation,
- primarily linear nearest-neighbor communication,
- global data-parallel operations, and
- data of nonstandard integer sizes

are especially well suited to the Terasys architecture.

At SRC, we've written approximately 20 applications for Terasys, and we've observed a performance range of five to 50 Cray-YMP single processor equivalents on these applications. A representative tree-search application written in the high-level dbC language ran at eight YMP equivalents with no performance tuning. A pseudorandom number generator written in Passwork produced $2 \times 10^{10}$ pseudorandom bits per second or about 20 Cray-YMP equivalents.[11] For these applications, performance of the YMP version is either an actual measured figure or a best-case estimate of bit operations, assuming no memory access delays.

In addition to running SRC applications, Terasys has been used for image-processing applications by researchers at the University of Massachusetts at Lowell. We report on one of their applications below.

### Real-time color imaging and visualization

Researchers at the Institute for Visualization and Perception Research and the Computer Science Department at the University of Massachusetts at Lowell have used Terasys to investigate applications of real-time color imaging and visualization.[12] A particularly interesting application is an interactive environment that provides near real-time manipulation of medical imaging data.

Typically, radiologists view pairs of magnetic resonance images (MRIs) taken at different frequencies side by side to find diagnostic information. One image usually has better contrast, while the other has better shape and fine detail information. Both images contribute to the diagnostic process. The goal of the medical imaging application written for Terasys is to integrate the images into a single picture, in which all diagnostic information is clearly and immediately visible. Further, the system lets the user modify viewing parameters dynamically and watch the display change in response.

**COLOR ICON.** In geometric coding, each datum is represented by a small graphical element or icon, whose visual features are controlled by the data. For this project, the group developed a general color icon to integrate data from multiple images. Information in multiple images can be combined into a single display by having the pixel values in the separate images control the three color coordinates of the corresponding pixel in the integrated display.

The color icon is represented by a square box of pixels. In the RGB color model, each corner of the box may have up to three associated parameters, one for each of the red, green, and blue components. The three parameters determine the color of that corner. Intermediate pixels have values interpolated from the corner points. This allows merging up to 12 data-set parameters. The single image is represented, then, by a square display of icons, where each icon is typically a $5 \times 5$ pixel matrix.

For the MRI application, the input data consists of a pair of gray-scale images with values at each pixel position ranging from 0 (black) to 255 (white). The values are normalized to the range (0, 1). The images are combined by associating each image with one of the three parameters of the Generalized Lightness, Hue, and Saturation (GLHS) color model. Since there are only two images in this appli-

### Cost estimates

The Terasys prototype, including hardware and system software, was assembled by a team of about 10 people over two years. The Terasys interface board was designed and built in-house, as were the array unit boards. The chips were fabricated at a silicon foundry.

Since Terasys is a research prototype, it's difficult to estimate costs. The cost of the PIM chip is particularly hard to estimate, since SRAM prices are so sensitive to production volume. A 16-Kbit SRAM, configured as $4K \times 4$ bits with 25-ns access time, can be purchased in quantities of 1,000 for $3.67. A 256-Kbit device, configured as $32K \times 8$ bits at 25 ns, costs $7.00. The PIM SRAM, with 128 Kbits ($64 \times 2,048$) and a 30-ns access time, falls between these two parts in density. However, the PIM chip package is more expensive than the DIP packaging of standard SRAMs. Factoring in all these variables, and especially the fact that we cannot expect chip volume to be anything close to that of standard SRAMs, a conservative per-chip cost estimate would be around $100 or $32,000 for 32K processors. We price the host workstation and PIM cabinet at approximately $10,000.

cation, the third parameter is set to an arbitrary value chosen by the user. The GLHS parameters are transformed into (red, green, blue) triples required by the display hardware.

**A PIM array within a vector supercomputer would create a vector/parallel/SIMD hybrid, where the SIMD array could also function as additional memory.**

**INTERACTIVE DISPLAY.** A unique feature is that the user can modify display parameters (controls) interactively. The simplest control is the "region of interest." Since the entire image cannot be displayed at once, only a portion is displayed. The user may drag the region selector over a small representation of the image to view other areas of interest.

The size of the icon box—that is, the number of pixels in the icon's height and width—can also be changed interactively. The color model used to calculate display colors can also be modified. The user can switch between the RGB and GLHS color models. While using the GLHS color model, the user can set values for each of the three variables (which determine how much of that component will appear in the displayed color) by dragging a point around a triangle, with one variable being maximized at each corner. The display is updated as the point moves within the triangle. Sliding bars can be used to dynamically change the input data's minimum and maximum thresholds. Once again, the display changes as the sliders change, letting the user adjust the parameters to extract the maximum information from the display.

**MAPPING TO TERASYS PROCESSORS.** A straightforward mapping of icon rendering to processors would associate an $M \times N$ icon with each processor, letting the processors compute independently without interprocessor communication. However, this approach was not used because with this mapping, the data is not arranged correctly for display and must be reshuffled in the Sparc memory before it can go to the frame buffer. Since the display time rather than the calculation time is the dominant factor, the method used is to load the data in the order required by the display hardware, and to use the linear nearest-neighbor network to exchange values needed to render an icon.

Using the latter method, a 4K Terasys can render $64 \times 64$ icons (a $320 \times 320$ pixel window) at a frame rate of 20 frames/second. The current implementation—which copies data from PIM memory into Sparc memory, from Sparc memory into a frame buffer; and then uses X-based display software to display the image—can display data at five frames per second. Alternatives to more closely match the display rate to the generation rate include bypassing the X display code to write directly into the frame buffer, and perhaps incorporating the frame buffer directly onto the Terasys hardware, thus eliminating the need to transfer displays over the Sbus.

THE PIM CHIP EMBODIES THE CONCEPT OF SIMD PROCESSING within the memory subsystem of conventional computers. We have built a system at one data point within that space, the Terasys workstation, which incorporates PIM processors in a Sparc-2. Programmed in the high-level dbC language, the Terasys workstation can deliver supercomputer performance at a very reasonable cost.

We are now exploring with Cray Computer the design of a PIM array within a vector supercomputer, the Cray-3/ SuperScalable System. This configuration would create a vector/parallel/SIMD hybrid, where the SIMD array could also function as additional memory. With the faster CPU cycle time of the vector supercomputer, issuing instructions to the PIM chips would not be a dominant factor, as it has been with the Sparc-2 and SBus on Terasys. This hybrid machine uses the supercomputer's gather-scatter hardware for efficient communication among the PIM processors. In the current design, the PIM memory will replace two octants of a Cray-3 memory system, serving as 512K processors, or 16 megawords of memory. Future generations of such a machine with 1K processors per PIM chip will be capable of $10^{15}$ bit operations per second, that is, a peta bitop supercomputer. ■


## Acknowledgments

The PIM chip is the invention of Ken Iobst, Dave Resnick, and Ken Wallgren. The following people have contributed to the Terasys project: Don Becker, Charlie Bostick, Harold Conn, Maya Gokhale, Bill Gromen, Howard Gordon, David Hickey, Bill Holmes, Jason Kassoff, Bob Kimble, Dan Kopetzky, Jennifer Marsh, Fred More, Alan Murray, Mark Norder, Phil Pfeiffer, Lou Podrazik, Judith Schlesinger, Paul Schneck, Al Schwartz, Doug Sweely, Tom Turnbull, and Lucak Womack.

We gratefully acknowledge the assistance of Rob Erbacher, Stu Smith, and Pat Mullins in preparing the section on applications and performance. We are indebted to the anonymous referees for their reviews of an earlier draft of this article.


## References


1. P. Schneck, "Wire Problems and Parallel Computation," *Comm. ACM*, Vol. 36, No. 1, Jan. 1993, p. 20.
2. J. Marsh and M. Norder, "PIM Chip Specifications," Tech. Report TR-93-088, Supercomputing Research Center, Bowie, Md., 1993.
3. K. Iobst and T. Turnbull, "Passwork User's Guide," Tech. Report TR-90-014 (third edition), Supercomputing Research Center, Bowie, Md., 1993.
4. D. Smitley and K. Iobst, "Bit-Serial SIMD on the CM-s and the Cray-s," *J. Parallel and Distributed Computing*, Vol. 11, No. 2, Feb. 1991, pp. 135-145.
5. J. Schlesinger and M. Gokhale, "dbC Reference Manual," Tech. Report TR-93-109, Supercomputing Research Center, Bowie, Md., 1993.
6. M. Gokhale and J. Schlesinger, "A Data-Parallel C and its Platforms," *Proc. Fifth Symp. Frontiers of Massively Parallel Computation*, IEEE CS Press, Los Alamitos, Calif., 1995, pp 194-202.
7. M. Gokhale and R. Minnich, "FPGA Programming in a Data-Parallel C," *Proc. IEEE Workshop FPGAs for Custom Computing Machines*, 1993, pp. 94-101.

8. M. Gokhale and B. Schott, "Data-Parallel C on a Reconfigurable Logic Array," Tech. Report TR-94-121, Supercomputing Research Center, Bowie, Md., 1994.
9. J. Schlesinger and D. Kopetsky, "Terasys, Microcode, and TWIST," Tech. Report TR-94-119, Supercomputing Research Center, Bowie, Md., 1994.
10. K. Iobst and T. Turnbull, "Terasys Reference Manual," Tech. Report TR-90-103 (third edition), Supercomputing Research Center, Bowie, Md., 1993.
11. S. Arno and K. Iobst, "Petasys Supercomputer and a Class of Pseudorandom Number Generators," Tech. Report TR-92-069, Supercomputing Research Center, Bowie, Md., 1991.
12. R. Erbacher, G. Grinstein, and S. Smith, "Implementing an Interactive Visualization System on a SIMD Architecture," Tech. Report, University of Massachusetts, Lowell, 1994.

***Maya Gokhale*** *is the group head of Systems Software for High Performance Computing at the David Sarnoff Research Center, Princeton, New Jersey. Previously, she was a research staff member at the Supercomputing Research Center, Bowie, Maryland, working in languages and compilers for high-performance computers. Prior to joining SRC in 1988, she was an assistant professor at the University of Delaware. She also has seven years of industry experience with Burroughs and Hewlett-Packard as a design engineer. Gokhale received a BS in mathematics from Wake Forest University in 1972 and the MSE and PhD degrees in computer science from the University of Pennsylvania in 1977 and 1983, respectively.*

***Bill Holmes*** *has been with the Supercomputing Research Center since 1987 as manager of the Horizon, Splash, and Terasys projects. Before that, he worked for NASA/Goddard, providing computer support to spacecraft missions. Holmes received a BS degree in mathematics from LaSalle University in 1966 and an MS in mathematics from Georgetown University in 1971.*

***Ken Iobst*** *is the chief architect of processing-in-memory machines at the Supercomputing Research Center, Bowie, Maryland. He is the inventor of the Bit-Serial Orthogonal Transformation Instruction used in Cray Research vector machines and of the processing-in-memory chip used in the Cray-3/Super Scalable System. He received his BS degree in electrical engineering from Drexel University in 1971 and his MS and PhD degrees in electrical engineering/computer science from the University of Maryland in 1974 and 1981.*

*Readers can contact Maya Gokhale at the David Sarnoff Research Center, CN5300, Princeton, NJ 08543. Her e-mail address is maya@earth.sarnoff.com.*