THE B-SYS PROGRAMMABLE SYSTOLIC ARRAY

Daniel P. Lopresti
Richard Hughey

Department of Computer Science
Brown University
Providence, Rhode Island 02912

Technical Report No. CS-89-32
June 1989

THE B-SYS PROGRAMMABLE SYSTOLIC ARRAY†

Daniel P. Lopresti
Richard Hughey

Department of Computer Science
Brown University
Providence, RI 02912

June 1989

Abstract

We introduce a general architecture for programmable systolic arrays which incorporates the following features: regular topology with nearest-neighbor connections, synchronous SIMD control, interprocessor communication using shared registers, and stream-based I/O. In our model, neighboring processors are granted direct access to one another's working storage; as a result, computation and communication are tightly intertwined, the latter resulting as a natural consequence of the former. We have found that many systolic algorithms can be expressed in such a fashion.

The Brown Systolic Array (B-SYS) is an embodiment of this philosophy. B-SYS is a highly parallel array of simple processing elements tuned for solving combinatorial problems, including sequence comparison. We are currently in the midst of implementing a B-SYS prototype in 2μ-CMOS. Although working hardware is not yet available, we have programmed a number of algorithms using B-SIM, a software emulator that has helped us refine the architecture. Preliminary estimates indicate that B-SYS will provide supercomputer performance for the applications of interest.

† This research is supported in part by the National Science Foundation under grant MIP 87-10745.

Contents

1 Introduction

The ongoing revolution in semiconductor fabrication technology has spurred numerous advances in computer architecture. Of these, the systolic array has proven particularly noteworthy. Ideally suited to the strengths of VLSI, systolic arrays offer tremendous speed-up over uniprocessor solutions for a broad variety of applications characterized by regular data flow. Arrays have been proposed for problems from such diverse areas as combinatorial optimization, signal and image processing, and mathematical modelling, an indication of their versatility.

Much of the beauty of the systolic paradigm lies in its simplicity. A small number of basic building blocks are replicated in a regular fashion to yield a powerful computational pipeline. The structure of the hardware is dictated by the structure of the algorithm, which can often be expressed succinctly. For example, a linear array of processors can sort integers using the following scheme: on each clock "tick" every processor compares a value previously written by its left neighbor to an internally stored value, saves the larger, and passes the smaller to its right neighbor. Figure 1 demonstrates this graphically. While few systolic algorithms are this straightforward, all combine computation and communication in a similar way.

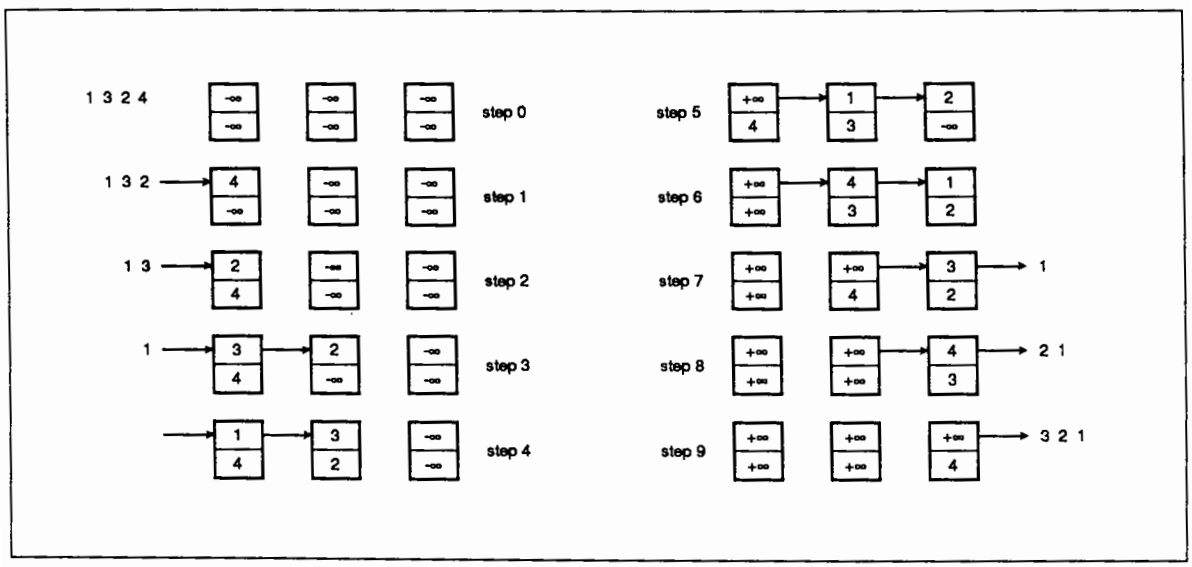

Figure 1. *A simple systolic sorting algorithm.*

Despite the considerable interest in this field, surprisingly few systolic algorithms have actually been implemented. The reasons for this are no doubt varied and complex, but a significant obstacle could be that systolic arrays are traditionally viewed as if they were special-purpose hardware devices. That is, the algorithm and the architecture are regarded as inseparable. This attitude is consistent with the intent originally expressed by Kung and Leiserson [Kung79], and although several special-purpose systolic arrays have been successfully developed, such an approach is simply not cost-effective. It rarely makes sense to construct a complicated system if that system has just a single use. The high design cost and long turn-

around time of fabricating custom silicon makes it impractical to consider building a new array for each variant of an algorithm. As a result, a significant percentage of the systolic algorithms described in the literature have never been run (and hence have never really been tested). This large body of research remains, for the most part, untapped.

The rather obvious solution to this problem is to build a multi-purpose (i.e., programmable) systolic array. Unhappily, the very notion of "programmability" violates a fundamental tenet of systolic design: that the basic cells be functional (and hence "simple") and not procedural (and hence "complicated"). The term "programmable systolic array" is, therefore, self-contradictory. We can sidestep the semantic issue by defining "programmable systolic array" to mean any highly parallel architecture capable of efficiently executing a number of different systolic algorithms. This interpretation is broad enough to include several machines which probably have never been called "systolic" by their own designers.

A more troubling point is the undeniably solid reasoning behind the original definition: programmability cannot be had for free. Many of the applications in question could use thousands of processors effectively; in current systems it is almost always the case that too few processors are available (and hence the problem must be partitioned). Making the processors more general will only make them larger, forcing us to forsake some inherent parallelism. Hence, the designer of a programmable systolic array is confronted by a trade-off between the degree of flexibility and the potential for parallelism. A variety of opinions have been expressed so far. Still, there is no consensus.

In the next section we present an abstract architecture for programmable systolic arrays. The driving philosophy here is that computation and communication should be tightly intertwined, the latter resulting as a natural consequence of the former. Section 3 provides specific details about the Brown Systolic Array (B-SYS), a programmable linear systolic array tuned for executing combinatorial algorithms. Preliminary estimates indicate that with B-SYS we will be able to fit 60 processors on a single 2μ-CMOS chip. The system we are proposing has an aggregate instruction processing rate of 4,000 MIPS, a figure competitive with current generation supercomputers. In Section 4 we compare B-SYS to several other highly parallel machines. Section 5 discusses research in progress. Finally, we summarize our main points in Section 6.

2 An Architecture for Programmable Systolic Arrays

To date, the architectures of most highly parallel computers have been application-driven. MPP, for example, is designed for processing satellite image data [Pott85]. The Connection Machine was originally proposed for artificial intelligence problems [Hill85]. For WARP, the primary area of interest is low-level vision [Anna87], and for NON-VON it is database operations [Shaw84]. Each of these machines can be programmed to handle tasks other than those indicated, with varying degrees of success. None, however, is a general-purpose multiprocessor in the same sense that a VAX 11/780[1] is a general-purpose uniprocessor.

Likewise, it is unrealistically optimistic to hope that a single machine could efficiently execute all systolic algorithms. As a consequence, it may seem pointless to cultivate any notion of a "universal" programmable systolic array. Fortunately, though, most problem-specific details affect the architecture of the individual processing elements, and not the architecture of the array as a whole. The class of target applications determines the functionality required of an ALU, for example: signal processing requires fast

[1] VAX is a trademark of Digital Equipment Corporation.

floating-point multiplication, combinatorial algorithms do not. Whether the basic cells contain hardware multipliers is irrelevant, however, if we concentrate on the high-level architecture of the array.

From that vantage point, one concern is overriding: whatever the structure of the constituent processors, we must have thousands of them. We consider this goal more important than minimizing the array's cycle time. Slow but small processors are better for our purposes than fast but large ones.[2] Only when it becomes possible to have too many processors will it make sense to trade additional chip real estate for increased performance of the individual processors. We further support this position by observing that, in practice, it is often impossible to feed systolic arrays at full-speed anyway. In the case of P-NAC, a linear systolic array for comparing nucleic acid sequences, we were able, at best, to pump data through the hardware at one-tenth of its maximum clock rate [Lopr87b].

With this in mind, we define our architecture for programmable systolic arrays. The following four features form the foundation:

- regular topology with nearest-neighbor connections
- synchronous SIMD control
- interprocessor communication using shared registers
- stream-based I/O

The first two points are relatively straightforward, the last two are comparatively novel. All are illustrated in the linear array of Figure 2(a) and the mesh array of Figure 2(b). We now explain the significance of each in turn.

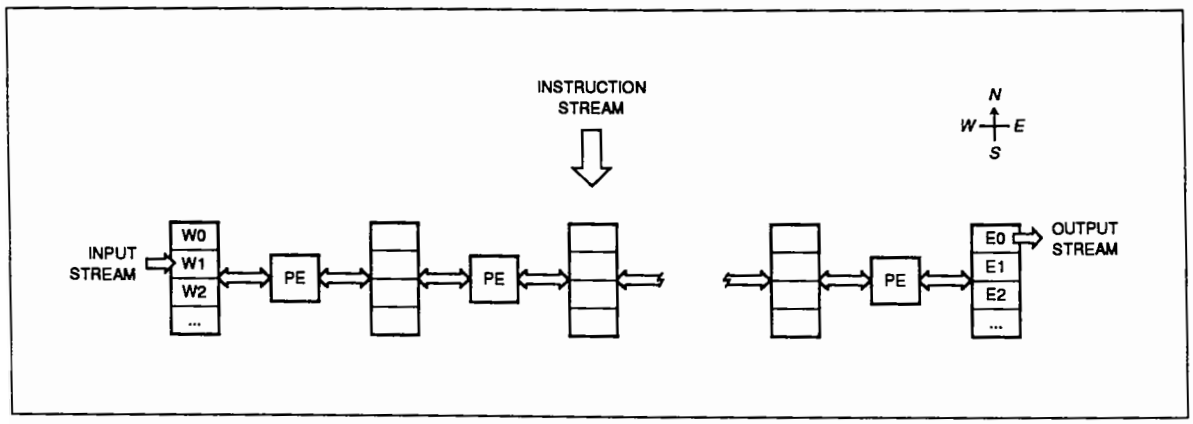

Figure 2(a). *Linear programmable systolic array.*

Regular Topology with Nearest-Neighbor Connections

It is hardly radical to suggest that programmable systolic arrays should use the same topologies as the special-purpose arrays they are designed to emulate. After all, systolic algorithms are optimized for such

[2] Small and fast processors would be best, of course.

architectures. If these algorithms are to be implemented efficiently, the hardware we provide must closely match the programming model. Most current research presumes linear, mesh, or hexagonal interprocessor connections, so it is natural to assume that programmable arrays will fall into the same categories.

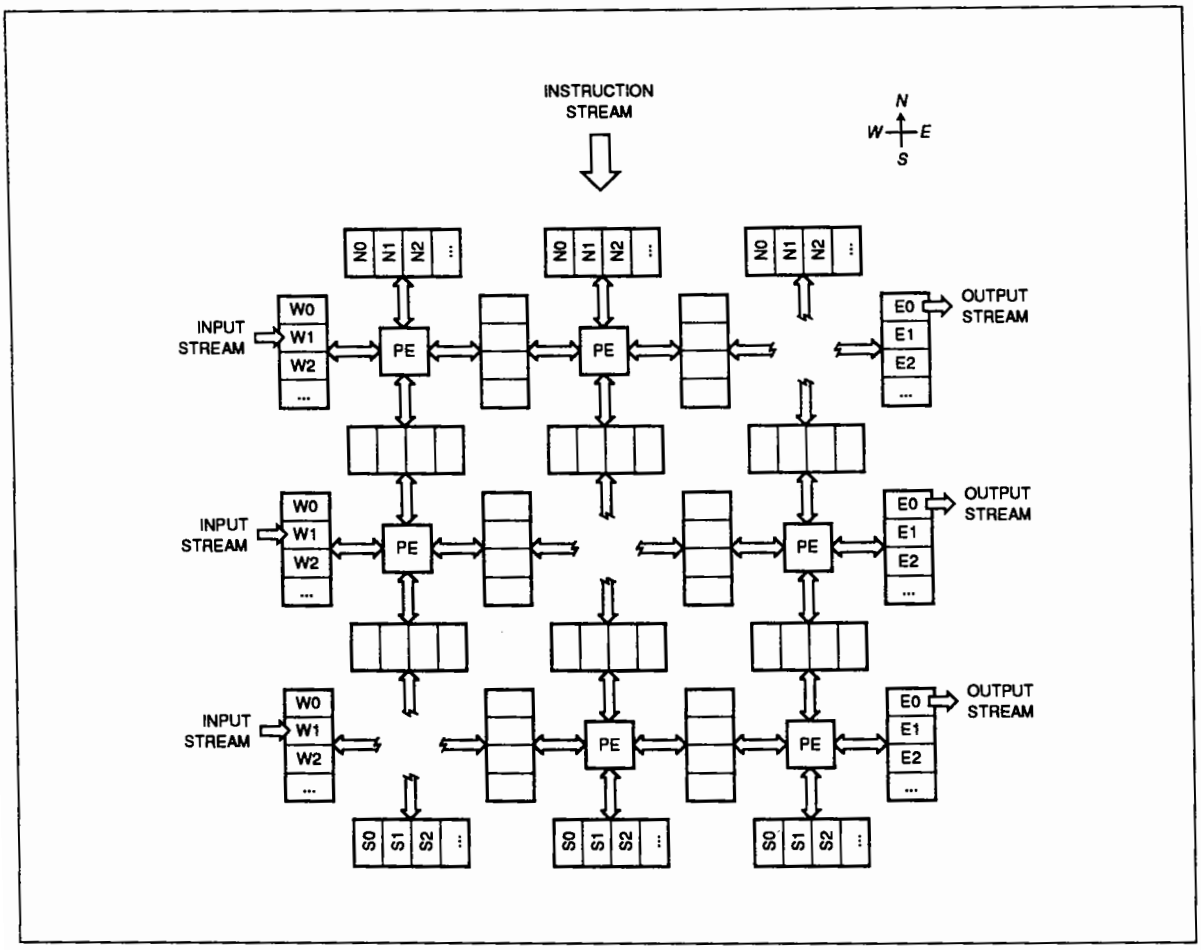

Figure 2(b). *Mesh programmable systolic array.*

Whether the user should be able to reconfigure the topology of a given system at run-time is a matter for some debate. This flexibility could be achieved through circuit switching (as in the CHiP computer [Snyd82]) or packet switching (as in the Connection Machine [Hill85]). In practice, though, we regard such schemes as overkill since almost all systolic algorithms map cleanly onto one of three models mentioned previously. (Certain "semi-systolic" algorithms employ non-local connections; these can be re-timed, however, so that they are purely systolic [Leis83]). Switches and routers consume valuable silicon that we would rather allocate to processing elements.

A secondary concern is that, while planar VLSI technology admits both one- and two-dimensional arrays, the former offer significant advantages when it comes to I/O requirements, extensibility, and clock

distribution. The number of pins available per chip-carrier, although steadily increasing, is not keeping up with the rapid decrease in device sizes. This trend seems likely to continue. As a result, highly parallel two-dimensional arrays are often limited to bit-serial communication, a constraint we find cumbersome.

Synchronous SIMD Control

MIMD architectures are admittedly more versatile than SIMD architectures. Nevertheless, if we take the popular view that the purpose of a systolic array is to provide a "hardware subroutine" for its host computer, it becomes evident that the resources we would be willing to commit to the array are rather modest, say a single printed-circuit board. Hence, to construct a system with thousands of processors, we must fit tens or hundreds of processors on each custom VLSI chip. This goal is not realizable using today's technology if we insist that every processor have its own program memory and instruction sequencer. Fortunately, systolic algorithms can be expressed naturally in SIMD fashion; we pay no penalty for not being able to afford the full-generality of a MIMD architecture.

This conclusion will almost certainly remain valid even when it becomes possible to implement highly parallel machines with fully-independent processors. The price that MIMD computers pay for their versatility is a need for costly synchronization techniques. These are wasted in the case of systolic algorithms, which are intrinsically synchronous. For a given technology, SIMD processors will always be smaller and faster.

Interprocessor Communication Using Shared Registers

Most existing parallel architectures are based on relatively general communication paradigms. Machines with tens of processors commonly use shared memory, while those with higher degrees of parallelism often employ message passing. Both of these approaches are too generic (and hence too expensive) for the class of programmable systolic architectures we envision, however. Systolic arrays have very specific communication requirements. Taking these into account leads to a simpler, more efficient programming model than either shared memory or message passing.

We start by observing that communication and computation are closely related in systolic algorithms. This synergism is exploited if we grant adjacent processing elements direct access to one another's working storage. Processors can communicate with their neighbors via these shared registers, as indicated in Figure 2. For example, the instruction

```
E0 ← min(W1,W2)
```

when broadcast, causes each processor to read two values from its west register set and then store the smaller in its east register set. This one instruction has two distinct effects: the operation (a minimization) and the propagation of data from west to east.

Although registers for interprocessor communication are included in other parallel machines, their use is typically limited to that one purpose. Our registers are general-purpose. This distinction may seem slight, but what we propose is more in accordance with the notion of systolic communication recently espoused by Kung [Kung88]. As a consequence, we have a right to claim the attendant benefits.

Interestingly, the shared register concept resembles shared memory in some ways, and message passing in others. Like shared memory, neighboring processors have a consistent view of the storage locations they

share and no explicit data movement instructions are necessary. Like message passing, we can view each register as a message queue (of length one) and we do not pay a high price to maintain a large uniform address space.

This scheme is not without its problems, however. Whenever a resource is shared, the possibility of contention exists. The instruction

```
E0 ← W0
```

for example, is ambiguous as stated, since one processor's `E0` is another's `W0`. Without further explanation it is unclear which values are saved and which are lost.

We avoid this uncertainty by insisting that all operands be read before any results are written. In effect, we assume the instruction processing cycle shown in Figure 3. On ϕ_a, every processor latches its first source operand. The second source operands are latched on ϕ_b. During ϕ_c the registers sit idle while the ALU's perform the indicated operation. Finally, every processor stores its result in the destination register during ϕ_d. Since the array is synchronous, and addressing is relative, no two processors will attempt to access the same register simultaneously. The above instruction, then, passes data from west to east unchanged, just as expected.

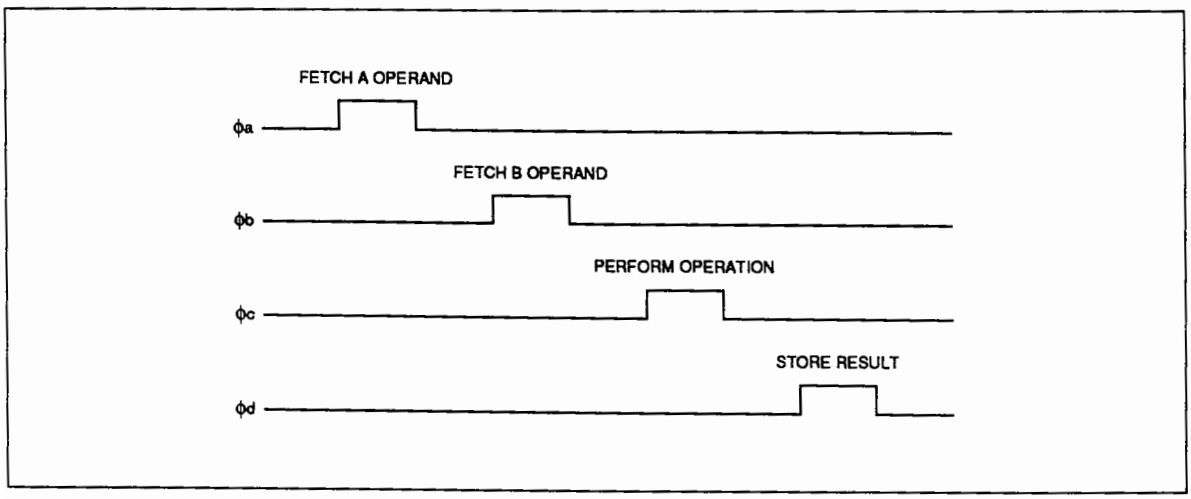

Figure 3. *The instruction processing cycle.*

Register sharing poses an additional complication when implementations must cross chip boundaries: the amount of time needed for a processor to access its local storage could become prohibitively large. We discuss this problem, and a possible solution, in a later section.

Stream-Based I/O

The file-as-a-stream abstraction popularized by UNIX[3] is both powerful and elegant. Central to this thesis is the *pipeline*, a formal mechanism for transforming an input data stream into an output data stream one character at a time. The basic UNIX pipe is somewhat limited, though, in that the data flow is unidirectional (i.e., from left to right as specified by the command line).

We treat our systolic arrays as a generalization of the UNIX pipeline. Input and output streams are bound to registers on the periphery of the array. A read from an input register returns the next item in the associated stream; a write functions analogously. For the linear array shown in Figure 2(a), the instruction

```
E0 ← min(W1,W2)
```

inputs a value to the west side of the array (via `W1`) and outputs a value from the east side of the array (via `E0`). In the case of the mesh array, this same instruction inputs a set of values to the west side and outputs a set of values from the east side.

Programming a systolic array that conforms to our model is straightforward. Consider the sorting algorithm described in Section 1. This scheme can be coded as a single loop of four instructions, as expressed in Figure 4.

```
while (more input) {                    /* until we run out of values to sort ... */
                                            /* even phase */
    E0 ← min(W1,W2);                            /* pass smaller value */
    W2 ← max(W1,W2);                            /* save larger value */
                                            /* odd phase */
    E1 ← min(W0,W2);                            /* pass smaller value */
    W2 ← max(W0,W2);                            /* save larger value */
}
```

Figure 4. *Pseudo-code for systolic sorting.*

The single thread of control (i.e., the `while` construct) resides on the host computer, while the `min` and `max` operations are broadcast for execution by the systolic array. Processors alternate their outputs between `E0` and `E1` to avoid prematurely overwriting previous outputs. The retained value is always held in `W2`. An input stream must be bound to registers `W0` and `W1` on the west edge of the array and an output stream to register `E2` on the east edge. Note that this code will run on either architecture depicted in Figure 2; it causes the mesh array to behave like a collection of independent linear arrays. Also observe that the program segment contains no instruction whose sole purpose is to transfer data between processors.

[3] UNIX is a trademark of AT&T.

The preceding constitutes our abstract model for programmable systolic arrays. Such architectures satisfy the demands of most systolic algorithms in a natural way, and yet should be cost-effective to implement. We now present a machine which builds on this framework.

3 The Brown Systolic Array

The Brown Systolic Array (B-SYS) is a highly parallel architecture tuned for solving combinatorial problems. Although originally conceived as a sequence comparison engine [Hugh88], it has evolved to handle a variety of other applications including sorting, searching, VLSI channel routing, and certain graph problems. These seemingly disparate tasks share a number of characteristics that distinguish B-SYS from parallel machines targeted at numeric or symbolic computations. Before presenting the architecture, we first describe its principal application.

3.1 The Sequence Comparison Problem

A *sequence* is a finite succession of symbols chosen from a finite alphabet. The ability to recognize that two information-bearing sequences are related, even though they may not be identical, is essential to a number of fields, including speech recognition, spelling correction, bird song classification, database retrieval, and genetics [Sank83]. In this last case, for example, the sequences denote deoxyribonucleic acid (DNA) molecules, and the alphabet $\{A, C, G, T\}$ represents DNA's four constituent nucleotides. When attempting to characterize an unfamiliar sequence, it often helps to compare it to a collection of known sequences with the goal of finding a close match.

There exist a number of techniques for judging the similarity of two sequences. One with considerable intuitive appeal is the *edit* (or *evolutionary*) *distance*, which is based on the premise that differences arise as a result of three fundamental phenomena: the deletion of a symbol, the insertion of a symbol, and the substitution of one symbol for another. The proximity of two sequences S and T can be quantified by assessing a charge for each of these steps and then finding the least expensive transformation of S into T. Unlike many other measures, edit distance is a metric (and hence obeys the triangle inequality).

This approach to sequence comparison seems to have been proposed simultaneously and independently in at least two disciplines: computer science [Wagn74] (where it is used in error-correcting compilers) and biology [Sell74] (where it is applied to evolutionary analysis). The calculation of edit distance can be formulated as a dynamic programming recurrence. If $S = s_1 s_2 ... s_m$, $T = t_1 t_2 ... t_n$, and $d_{i,j}$ is the distance between the subsequences $s_1 s_2 ... s_i$ and $t_1 t_2 ... t_j$, then

$$
\begin{aligned}
d_{0,0} &= 0 \\
d_{i,0} &= d_{i-1,0} + c_{del}(s_i) \qquad 1 \le i \le m \\
d_{0,j} &= d_{0,j-1} + c_{ins}(t_j) \qquad 1 \le j \le n
\end{aligned}
$$

and

$$d_{i,j} = \min\begin{cases} d_{i-1,j} + c_{del}(s_i) \\ d_{i,j-1} + c_{ins}(t_j) \\ d_{i-1,j-1} + c_{sub}(s_i,t_j) \end{cases} \qquad 1 \le i \le m,\ 1 \le j \le n \tag{1}$$

Here $c_{del}(s_i)$ is the cost of deleting s_i, $c_{ins}(t_j)$ is the cost of inserting t_j, and $c_{sub}(s_i,t_j)$ is the cost of substituting t_j for s_i.

The obvious implementation of this algorithm on a sequential machine requires time $O(mn)$ and space $O(\min(m,n))$. Since the sequences in question can be long (e.g., thousands or tens of thousands of nucleotides in the case of DNA), searching a large database consumes substantial computer resources.

Fortunately, there is tremendous potential for parallelism in the construction of an edit distance table; all values $d_{i,j}$ can be calculated simultaneously for a given $k = (j - i)$. Mapping the recurrence onto a linear systolic array is a straightforward procedure, although there are several possible data flows that can yield alternate architectures. The Princeton Nucleic Acid Comparator (P-NAC) is one such array, designed and built for the sole purpose of comparing DNA sequences [Lopr87b]. Our implementation assumed that $c_{del}(s_i) = c_{ins}(t_j) = 1$ for all s_i and t_j, and that $c_{sub}(s_i,t_j) = 0$ if s_i matches t_j and $c_{sub}(s_i,t_j) = 2$ otherwise. Benchmarks established that P-NAC was several hundred times faster than existing minicomputers.

Despite P-NAC's speed, the prototype has not gained wide acceptance because it lacks the flexibility of a software solution; its notion of similarity is rigidly fixed. Discussions with biologists have established that, at the very least, the user should be allowed to choose the edit costs. Even better would be to admit variants of the original dynamic programming algorithm. For example, two sequences need not be similar in their entirety for their comparison to be meaningful. One sequence might resemble a subsequence of another, or two sequences might share a related subsequence. While these generalizations have systolic solutions [Lopr87a], none can make use of P-NAC.

Another processor array proposed for sequence comparison suffers from the same limitations [Mukh89]. In this case, the hardware returns the length of the *longest common subsequence*, a criterion essentially equivalent to the edit distance as calculated by P-NAC (if $l_{i,j}$ is the length of the longest common subsequence, then $l_{i,j} = (i + j - d_{i,j}) / 2$ assuming P-NAC's cost assignment). As defined by Equation 1, however, edit distance is a more general measure of similarity than longest common subsequence.

B-SYS is our attempt to address these flaws. Its architecture is based on the guidelines presented in the preceding section, further refined by details specific to sequence comparison. In particular, B-SYS features:

- linear data flow
- a simple but flexible instruction set
- modest amounts of processor memory
- an 8-bit word size[4]

[4] This point deserves further explanation. While eight bits are sufficient to represent any ASCII character, it is not obvious that such a small word can hold edit distances, which grow linearly with the length of the sequences being compared. In a previous work, we introduced a technique that makes it possible to encode edit distances in a constant number of bits, independent of the problem size [Lopr87a].

3.2 B-SYS Processor Architecture

A block diagram depicting the major functional modules and data paths of a B-SYS processing element is shown in Figure 5. Contained within each PE are an 8-bit ALU and a register holding eight 1-bit flags. Interspersed between every pair of adjacent processors are 16 8-bit registers, as discussed in Section 2. These shared registers are used for both computation and communication. At the hardware level none of the memory is private (other than the flags), but at the software level we may treat certain registers as holding local values. In the sorting program of Figure 4, for example, every processor uses `W2` to save the result of its `max` operation. Since no processor reads or writes `E2`, the value stored in `W2` is effectively local.

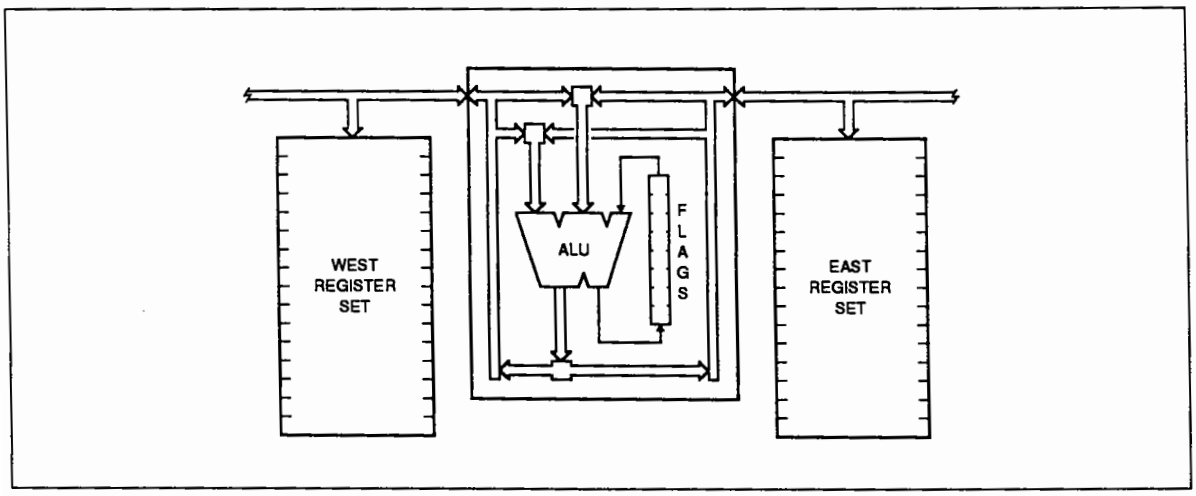

Figure 5. *Architecture of a B-SYS processing element.*

The B-SYS ALU, illustrated in Figure 6, is a bit-slice design reminiscent of the OM-1 [Mead80]. It accepts two 8-bit operands (A and B) and a flag (C) as input, and produces an 8-bit result (Y) and a flag (Z) as output. Each stage i calculates Y_i and Z_i independently using table lookup. Any three-to-one function of A_i, B_i, and C_i can be specified for Y_i. To minimize the ALU's cycle time, the evaluation of Z_i employs a Manchester carry chain. In this case, the truth table specifies two two-to-one functions of A_i and B_i that define the necessary generate (G_i) and propagate (P_i) signals which combine with C_i to produce Z_i.

This approach yields a simple yet flexible ALU capable of performing addition, subtraction, and arbitrary boolean functions, everything required for sequence comparison and the other applications mentioned earlier. B-SYS provides no hardware support for multiplication or for floating-point arithmetic; these were deemed luxuries. Note, though, that this is a limitation imposed by one particular implementation, not by the underlying architecture.

A processor's eight flag bits represent the entirety of its internal state. Seven of these are general-purpose, to be used for holding intermediate results (e.g., the outcome of a register-to-register comparison). The eighth flag serves as a mask. When this bit is asserted, the processor involved is effectively disabled; the Y and Z outputs of its ALU are not latched into their respective destinations. This technique is frequently

used in SIMD machines to give the individual processors a slight degree of autonomy. Of course, we also need a way of clearing the mask bit once it is set. For this reason, every B-SYS instruction is furnished in two forms: one that obeys the mask and one that ignores it.

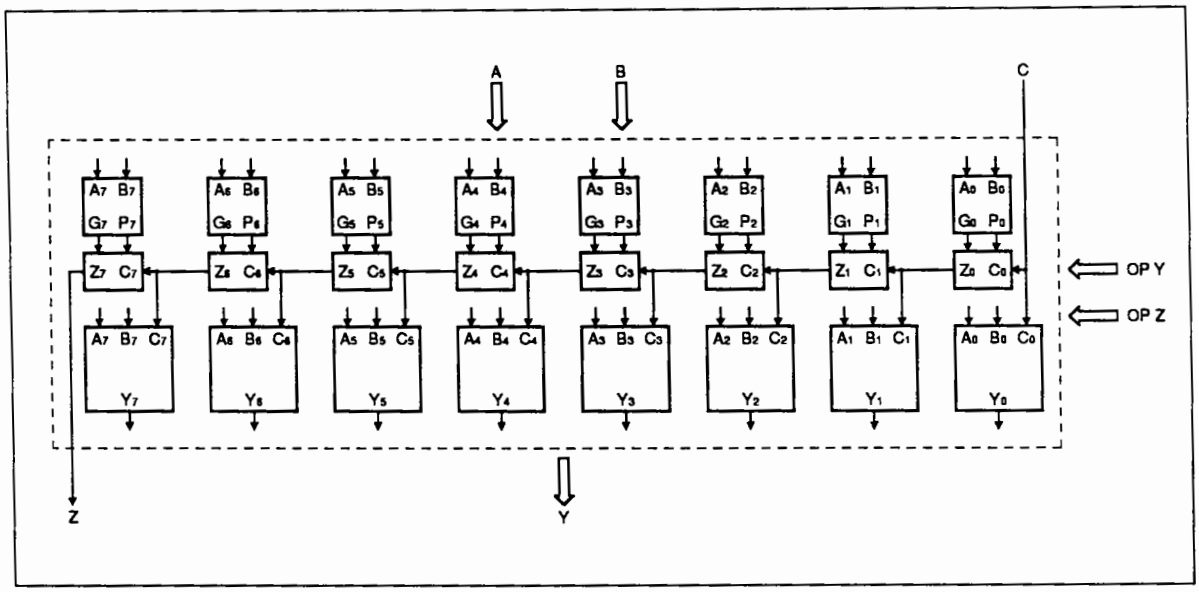

Figure 6. *The B-SYS ALU.*

A scheme that can sometimes take the place of masking is the selective loading of a register based on a previously evaluated predicate. Constructs like

```
if W1 < W2 then
    E0 ← W1
else
    E0 ← W2
```

can be implemented on B-SYS without disabling processors. First the comparison is performed and the result is stored in a temporary flag, say `F1`. Then `W1` and `W2` are input to the ALU and `F1` is used to select which of these is output to the destination register `E0`. We are treating the ALU as if it were a two-to-one multiplexer.[5] This approach entails less overhead than traditional masking since there is no saving and restoring of mask bits.

In keeping with our "simple is better" philosophy, B-SYS is horizontally microcoded.[6] A machine instruction is 38 bits long and can be broken into three distinct fields, as shown in Figure 7. The first

[5] Note that this code fragment expresses the `min` function in terms of more basic operations (comparison, conditional execution, and assignment.)

[6] Actually, "microcode" is a slight misnomer since B-SYS is hard-wired, but we find it convenient to use the term in the same way the designers of RISC microprocessors do (i.e., the microcode and the assembly language for B-SYS are essentially equivalent.)

field, a single bit, indicates whether or not the mask flag is to be overridden. The second field, 23 bits long, specifies the word operation to be performed. Eight of these bits determine the function (i.e., truth table column), while five bits each select the *A* source register, the *B* source register, and the *Y* destination register (one bit designates the east or west register set, the other four bits choose one of the 16 registers therein). The third field, 14 bits long, specifies the flag operation to be performed. Four bits each determine the truth table columns for G_i and P_i, while three bits each select the source and destination flags.

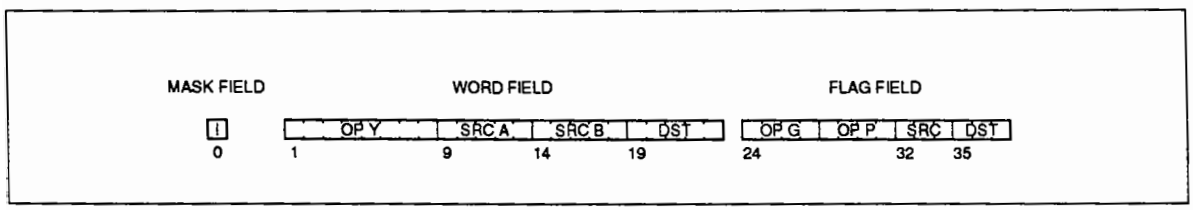

Figure 7. *The B-SYS micro-instruction format.*

Table 1 lists several op-codes for each of the three fields. Note that the mnemonics for flag operations are prefaced by `Z`. The translation of high-level language statements into B-SYS microcode is a relatively mechanical process. For example, the maskable 8-bit addition:

```
E0 ← W1 + W2
```

corresponds to the B-SYS instruction:

```
conditional  xorABC W1 W2 E0  Zadd F7 F1
```

assuming that `F7` has been initialized to provide a carry-in of 0. `F1` receives the carry-out and can be designated as the input flag for a subsequent addition to implement extended precision arithmetic.

mnemonic	*hex*
`conditional`	0
`always`	1

(*a*) mask field

mnemonic	*function*	*hex*
`zero`	0	00
`fnA`	A_i	aa
`nandAB`	$\overline{A_i B_i}$	77
`xorABC`	$A_i \oplus B_i \oplus C_i$	96
`selectABonC`	$A_i C_i + B_i \overline{C_i}$	ac

(*b*) word field

mnemonic	*function*	*hex*
`Zzero`	0	00
`ZA`	A_i	50
`Zconst`	C_i	f0
`Zadd`	$A_i B_i + A_i C_i + B_i C_i$	68
`Zsub`	$A_i \overline{B_i} + A_i \overline{C_i} + B_i C_i$	94

(*c*) flag field

Table 1. *Common B-SYS micro-instructions.*

We are currently in the midst of building a B-SYS prototype in 2μ-CMOS. The layout of a basic cell consisting of one processing element and one register set is complete; a plot is shown in Figure 8. Note that approximately equal amounts of real estate are devoted to the ALU and memory. This balance was unintended, but seems nevertheless appropriate. The entire cell contains 1,753 transistors and measures 583μ × 969μ. Assuming a die size of 6.8mm × 6.9mm, we can expect to fit 60 processing elements and register sets on a chip.

Preliminary estimates indicate that the array will have a major cycle time of 300 ns. This interval is divided into three 100 ns minor cycles, each with a 25 ns precharge phase and a 75 ns evaluate phase. B-SYS is based on very conservative design principles (so that it will likely work on first silicon), a more aggressive implementation would no doubt result in a higher clock rate.

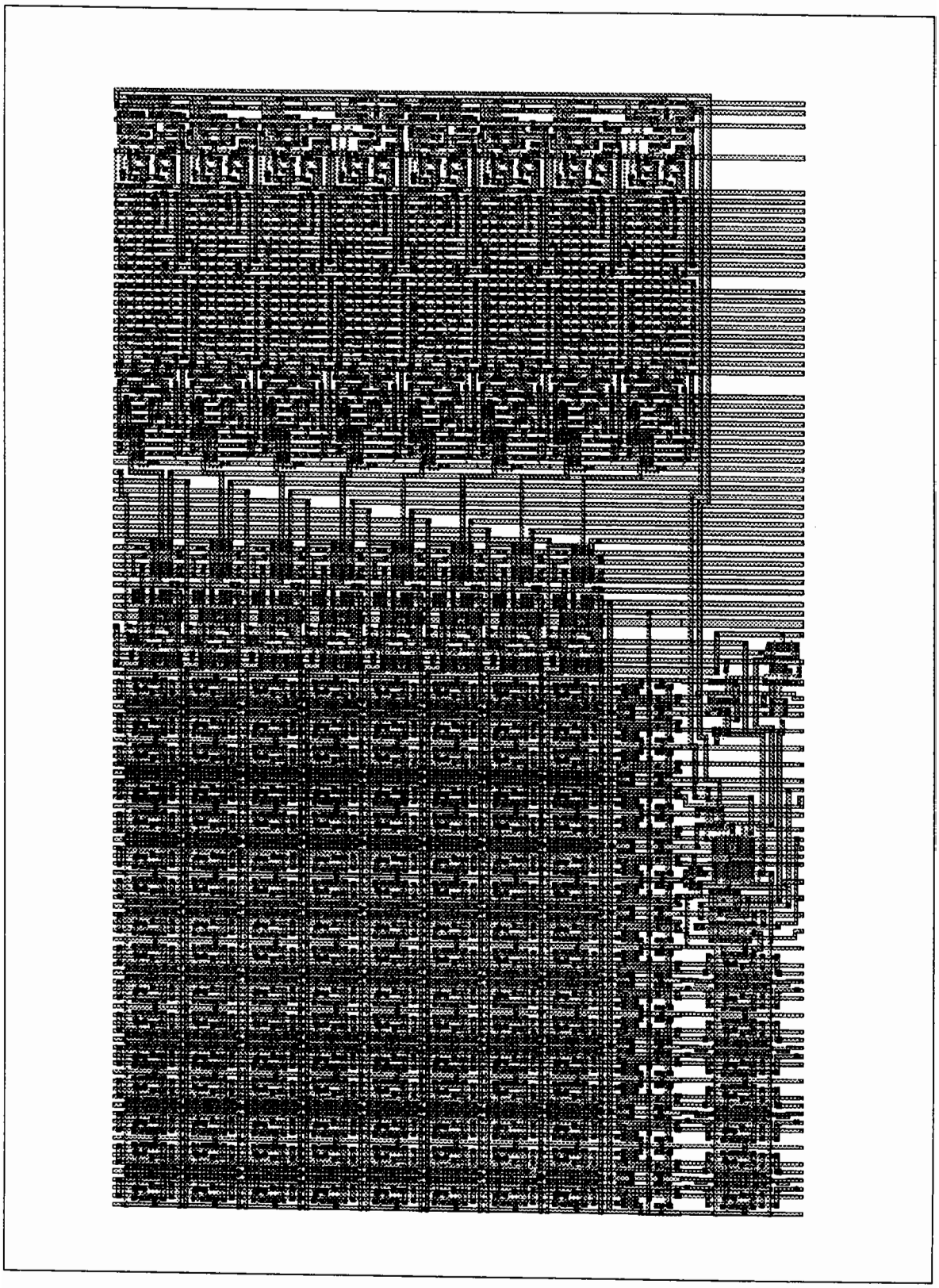

Figure 8. *Plot of a B-SYS processing element and register set.*

3.3 The B-SIM Simulator

Although B-SYS hardware is not yet available, the Brown Systolic Simulator (B-SIM) provides software emulation of the architecture and runs B-SYS code unmodified; a number of applications have already been programmed. B-SIM both facilitates algorithm development and supplies a tool for rapidly weighing design alternatives. We expect that the simulator will remain useful as a debugger even after the hardware becomes functional.

At this point, all programming for B-SYS (and hence B-SIM) is done in microcode. Many low-level details can be hidden, however, through judicious use of the C macro-preprocessor.[7] The inner loop of the sorting algorithm described earlier is shown in Figure 9. To compare the two values, we subtract one from the other and save the resulting borrow in `F1`. We then use this bit as a selector to transmit the smaller value (via either `E0` or `E1`) and retain the larger (in `W2`). Register `W3` is used for scratch storage, and flag `F7` holds a constant 0. While this program segment contains all the information needed for it to be directly executable, it is still not far removed from the abstract pseudo-code presented in Figure 4.

```
#include "../src/opc2.h"

                                                    /* even phase */
always   xorABC W1 W2 W3          Zsub F7 F1           /* compare values */
always   selectABonC W1 W2 E0     Zconst F1 F1         /* pass smaller value */
always   selectABonC W2 W1 W2     Zconst F1 F1         /* save larger value */
                                                    /* odd phase */
always   xorABC W0 W2 W3          Zsub F7 F1           /* compare values */
always   selectABonC W0 W2 E1     Zconst F1 F1         /* pass smaller value */
always   selectABonC W2 W0 W2     Zconst F1 F1         /* save larger value */
```

Figure 9. *B-SYS sorting loop.*

This loop file is one of several we must specify; others define the hardware configuration, initialize the array, and designate the input and output data streams.

The present version of B-SIM is batch-oriented (as opposed to interactive) and produces output for the TEX typesetting system. *Snap-shots* of the array state are taken at regular intervals determined by the user. Figure 10 displays several iterations of the sorting loop of Figure 9. Here we have indicated that we want to view the program's progress after every third instruction (i.e., between the even and odd phases). The simulator depicts not only the state of the array at each time-step, but also the word and flag operations that are executing.

[7] Eventually, we envision embedding the constructs needed to support B-SYS programming in the C language itself. Execution of such code will generate a stream of instructions to be broadcast to the systolic array. Frequently used functions could be maintained in a parameterized subroutine library, to be bound and loaded on demand.

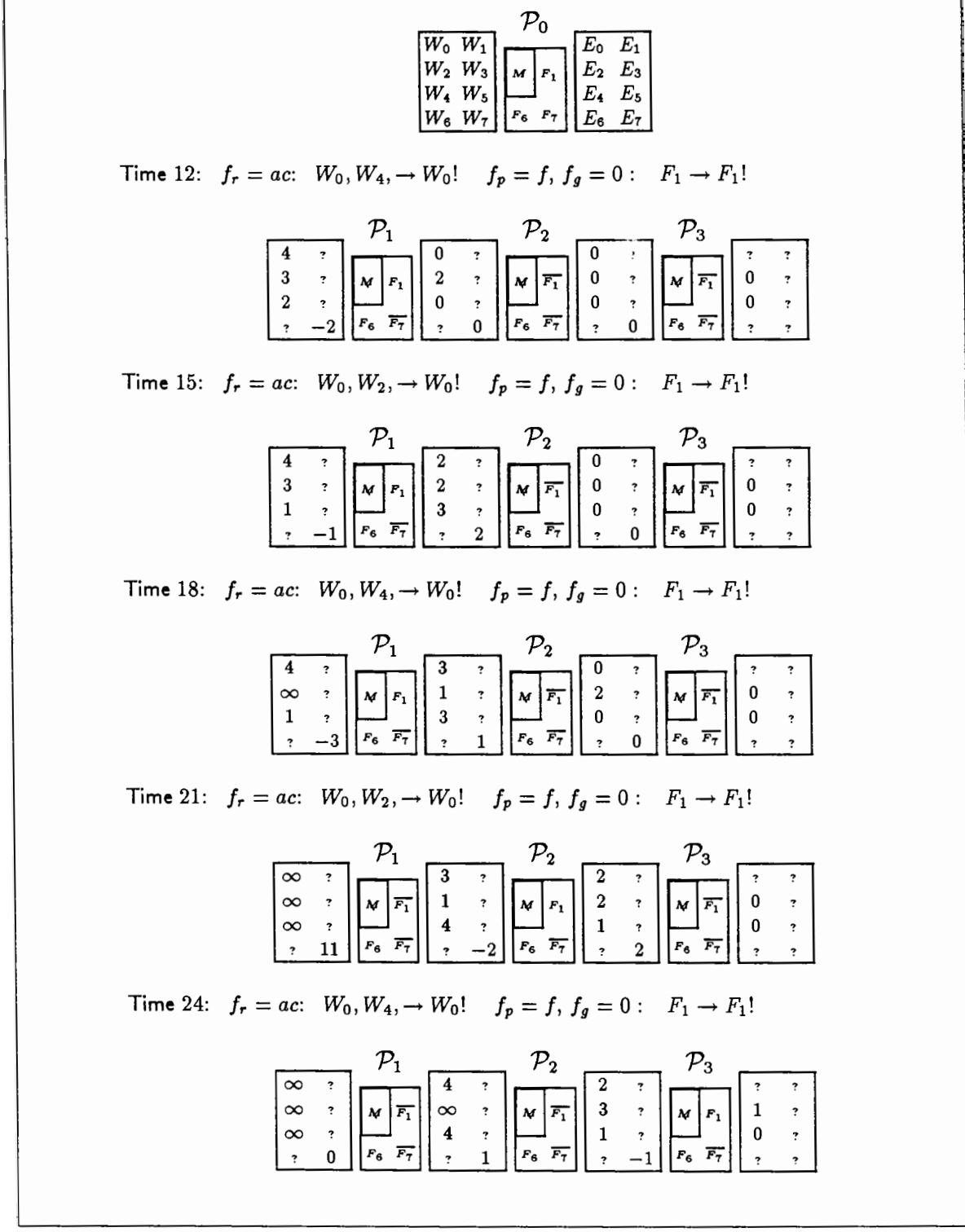

Figure 10. *B-SIM snap-shots showing sorting loop in progress.*

3.4 B-SYS Performance Evaluation

B-SYS is intended to be an attached co-processor. Although we have not yet chosen a host machine, 20 of our custom CMOS chips, along with the necessary support logic, should fit easily on most printed circuit boards. Assuming 60 processors per chip, the complete array will total 1,200 processors. Aggregate system characteristics, based on a 3.33 MHz clock rate, are shown in Table 2. Note that for addition and boolean operations, B-SYS compares favorably with current generation supercomputers. Also interesting is that even though B-SYS possesses no multiplication hardware, it can still perform a respectable number of 8-bit multiplies per second (using the standard shift-and-add scheme). Perhaps most importantly, the data and instruction I/O bandwidths needed to achieve these levels of performance are relatively modest.

characteristic	*measure*
instruction processing rate	4,000 MIPS
8-bit additions / booleans	4,000 MOPS
16-bit additions / booleans	2,000 MOPS
32-bit additions / booleans	1,000 MOPS
8×8 multiplications (8-bit result)	118 MOPS
8×8 multiplications (16-bit result)	80 MOPS
array memory bandwidth	12 gigabytes / second
maximum data I/O bandwidth	10 megabytes / second
instruction issue rate	3,333,333 / second
array memory size	20,400 bytes
total array transistors	2,103,600

Table 2. *B-SYS system characteristics.*

The figures cited in Table 2 provide one indication of the speed of B-SYS. To be thorough, we should also consider its handling of a "real" problem, sequence comparison being the obvious choice. It is natural to wonder whether B-SYS would be faster than, say, a Connection Machine. As we observed earlier, however, highly parallel computers are usually application-driven; benchmarking them against one another is a risky venture. B-SYS has the advantage of being designed with sequence comparison in mind, but it is much simpler than commercial parallel systems, some of which cost millions of dollars. Since programming styles vary widely, an algorithm that is a good choice for one architecture may be a poor choice for another. As a result, it would be unfair to quote absolute timings. Still, we have conducted a large number of informal experiments and can state that B-SYS is the fastest sequence comparison engine encountered to date.

We are more willing to match B-SYS against its predecessor, P-NAC, and general-purpose sequential machines. These figures appear in Table 3. The first set of columns are for performing 100 comparisons

of sequences 100 characters long, the second set for performing 100 comparisons of sequences 1,000 characters long. The B-SYS timings are, of course, estimates. All other timings are actual. The complete B-SYS code for sequence comparison appears in Appendix A.

	100 × 100		100 × 1,000	
system	*time*	*speed-up*	*time*	*speed-up*
B-SYS	66 ms	970	660 ms	9,700
P-NAC	910 ms	70.3	9.1 s	703
DEC VAX 8600	31 s	2.06	3,100 s	2.06
Sun 3/140	48 s	1.33	4,800 s	1.33
DEC VAX 11/785	64 s	1	6,400 s	1

Table 3. *Results of sequence comparison benchmarks (B-SYS times estimated).*

4 Comparison to Other Parallel Architectures

Over the past several years a number of parallel computers have been proposed that satisfy our definition of a programmable systolic array. In terms of architecture, there seem to be two prevailing philosophies. One holds that a machine should have as many bit-serial processors as possible (tens of thousands, with today's technology). The other believes that the individual processors should be as powerful as general-purpose microprocessors, even if that means we can afford relatively few (tens or hundreds). B-SYS falls squarely between these two extremes.

We now briefly survey some of the more important architectures in each category, first reviewing the bit-serial machines, then the bit-parallel. Many of these have influenced the development of B-SYS to one degree or another. We conclude the section by describing three new approaches that may prove fruitful in the future.

The Massively Parallel Processor (MPP)

MPP was built jointly by Goodyear and NASA for solving problems in image processing and fluid dynamics [Batch80], [Pott85]. It features a 128 × 128 square (toroidal) mesh of bit-serial processing elements. Each PE has 36 bits of on-chip memory and can access another 1,024 bits off-chip. A processor communicates with its four neighbors via a dedicated single-bit register. The processing elements are realized in CMOS, eight per chip; the complete array fills 96 circuit boards.

The Connection Machine (CM-1, CM-2)

The Connection Machine, developed at MIT and now sold by Thinking Machines Incorporated, is a highly parallel architecture originally intended for AI applications [Hill85]. In its most powerful configuration, it

incorporates over 65,000 simple one-bit processing elements arranged in two distinct topologies: a hypercube routing network and a multidimensional nearest-neighbor grid. The processors, packed 16 to a chip, each possess 64 kilobits of local memory. The CM-2 can perform 8-bit integer addition at a rate of 4,000 million operations per second (MOPS).[8]

The Geometric Arithmetic Parallel Processor (GAPP)

GAPP, designed by NCR, is not a complete system per se but a building block [NCR85]. The GAPP chip places 72 processing elements in a 6×12 square mesh. Each of these contains a bit-serial ALU, 128 bits of private memory, four flags, and two communication registers (one for north-south data streams and one for east-west data streams). GAPP was used at Washington University to build a 48×48 grid for medical imaging [Morl88].

The Programmable Systolic Chip (PSC)

PSC was an early attempt to build a flexible cell that could form the basis for a variety of arrays [Fish83]. Each PE is essentially an independent 8-bit microprocessor and includes an on-chip writeable control store, making the architecture MIMD. A processor also contains 64 9-bit registers, six I/O ports (three input and three output), and a multiplier-accumulator. Owing to this hardware complexity, and VLSI technology at the time, one PSC processor takes up an entire chip.

Warp

The Warp computer, developed by Carnegie-Mellon in association with General Electric, is the successor of PSC [Anna87]. Warp processors are so complicated that, in the current version, each requires its own printed circuit board.[9] The complete system consists of 10 PE's connected in a linear array, a number that hardly qualifies as "highly" parallel. Instead, Warp achieves its impressive performance through internal parallelism, including extensive pipelining. Processors are horizontally microcoded, with program memory for 8k instructions. Their data paths incorporate separate 32-bit floating point multipliers and adders, 32 kilowords of local storage, and three hardware-managed communication queues, each capable of buffering 512 words. Warp has a peak speed of 100 megaflops.

Like B-SYS, the first three architectures employ SIMD instruction broadcasting. The MPP and CM-2 are distinguished by the magnitude of their implementations; both are expensive large-scale computers. B-SYS is closer in spirit to GAPP, but still differs significantly in topology (linear versus two-dimensional) and ALU complexity (bit-parallel versus bit-serial).

Surprisingly, B-SYS bears less resemblance to PSC and Warp, the two systems put forth specifically as "programmable systolic arrays." These are MIMD architectures with elaborate processors and dedicated communication hardware. B-SYS is considerably simpler (and admittedly more specialized) than either.

---

[8] By coincidence, this is exactly the same figure we project for B-SYS.

[9] Intel and Carnegie-Mellon are collaborating on a single-chip implementation of the Warp processor. This machine will be known as "iWarp."

Figure 11 contrasts B-SYS and these other machines based on four criteria. The three axes represent parallelism (as measured by the number of processors), local storage (as measured by the amount of memory available to each processor), and estimated cost. The size of each data point indicates PE complexity (as measured by the width of the ALU).

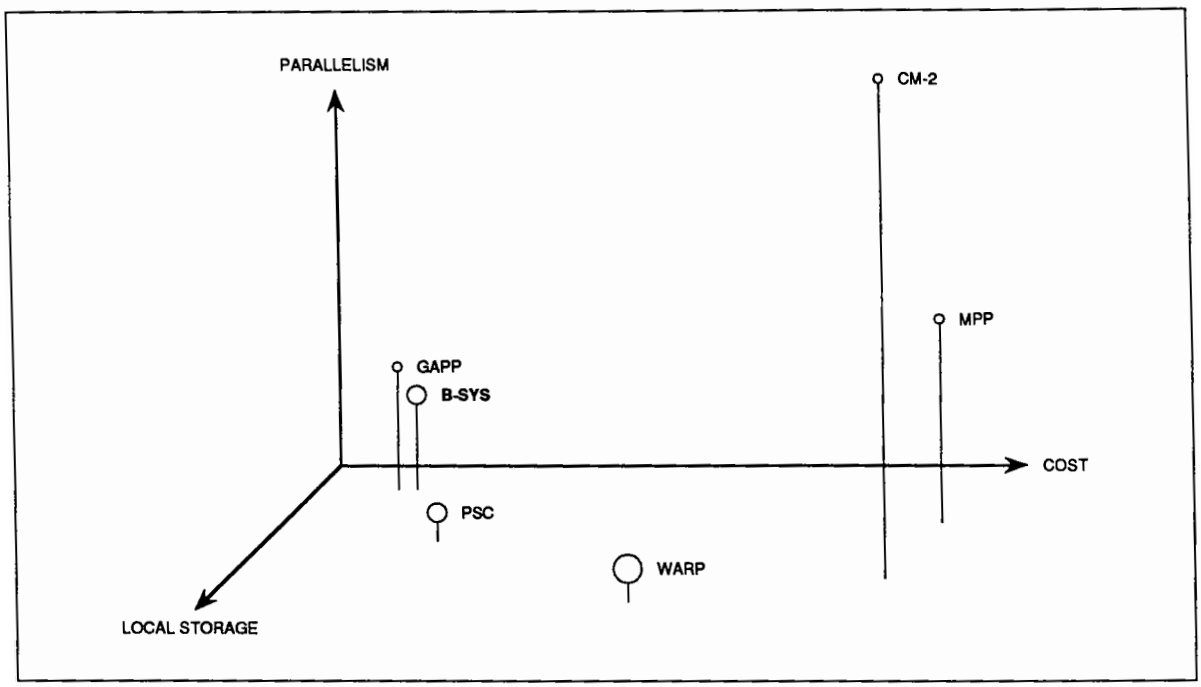

Figure 11. *Comparison of several parallel architectures.*

We now describe three promising, but as of yet unproven, approaches to the problem of building a programmable systolic array.

The Configurable Highly Parallel Computer (CHiP)

The Configurable Highly Parallel Computer is actually a family of "paper" architectures; as such, certain key details are left unresolved [Snyd82]. In particular, no mention is made of the functional modules to be included in a PE.[10] Instead, CHiP is notable because the processors do not communicate directly with each other, but are embedded in a two-dimensional switch lattice. The switches are programmable in the same way the processors are, so that any of a number of different topologies can be realized.

[10] In this way, CHiP resembles the abstract architecture we presented in Section 2.

The Instruction Systolic Array (ISA)

The Instruction Systolic Array is mesh of simple processing elements through which both data and control information flow simultaneously [Lang86], [Schm86]. Instructions pass from north to south while *selectors* pass from west to east. A processor executes a given instruction if and only if the corresponding selector enables it.[11] This is a clever technique for achieving MIMD-like operation without the expense of local program memory. To our knowledge, the ISA has not been implemented. (B-SYS originally was to have used a scheme like this [Hugh88], but it was later determined that for a linear array the SIMD broadcast model is just as powerful and more intuitive).

Configurable Logic

An exciting development in computer engineering has been the emergence of high-density programmable logic (i.e., hardware that can be reconfigured without removing it from the circuit). While manufacturers market these devices primarily as a replacement for random logic, they can also be used to implement certain systolic algorithms [Gray89].

This category is typified by the Xilinx 3090 programmable gate array, which contains 320 *configurable logic blocks* (CLB's) arranged in a 20×16 grid [Xili88]. Each CLB incorporates two bits of storage and an ALU that can perform a single arbitrary function of five inputs, or two arbitrary functions of four inputs. A flexible interconnect network (somewhat like the switch matrix in CHiP) occupies the channels between logic blocks. Customizing a 3090 is more akin to building hardware than writing software; because so many low-level details must be resolved, good design tools are a necessity.

There are, of course, other parallel architectures capable of running systolic algorithms. As we observed earlier, however, most of these use message passing or shared memory and make no special provisions for systolic communication. Hence, they are not as effective as the machines we have just described.

5 Continuing Research

B-SYS has evolved considerably since its conception, although there are a number of issues we have not yet addressed. Our foremost goal has always been to build a working system and make it available to interested users. Accordingly, we are now concentrating much of our effort on finishing the CMOS prototype. The remainder of our time is directed towards topics in system integration, programming support, and application development. While we are concerned primarily with the impact on B-SYS, it should be understood that many of these are general problems arising from the architectural model described in Section 2.

System Integration

As we suggested earlier, the shared register paradigm can be a double-edged sword: communication is just as fast as computation, but, conversely, computation is only as fast as communication. This becomes a

[11] Selectors can be viewed as mobile mask bits.

pressing concern when an implementation must cross chip boundaries; certain processors are necessarily separated from half of their registers, as depicted in Figure 12(a). Since the delay in driving a signal from one chip to another is relatively large, slowing the global clock in response could severely impair system performance.

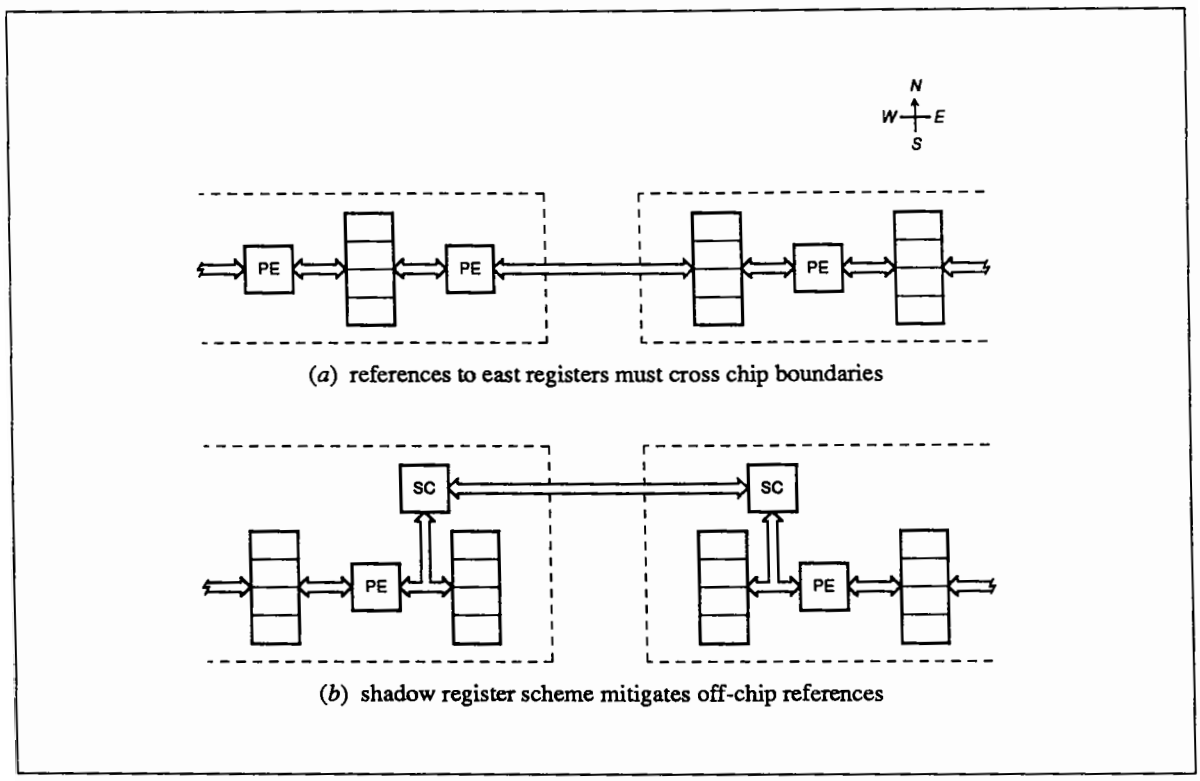

Figure 12. *The off-chip reference problem and a solution.*

An obvious solution would be to acknowledge the asymmetry and allow additional time for such references to complete. This can be achieved by increasing the duration of ϕ_a, ϕ_b, and ϕ_d as necessary (recall Figure 3). For the array in Figure 12(a), storing a value in `E0` takes longer than storing a value in `W2`; hence, the first instruction in the sorting loop of Figure 4 is slower than the second. Under these circumstances, we should write our programs so that they perform local computation in the west register set, using the east set only when communication is required (the code in Figures 4 and 9 follows this guideline).

This scheme, while workable, penalizes all references in the affected direction. A more elegant approach would be to duplicate the register sets in question so that each chip has its own local copy. The difficulty then becomes one of keeping the copies consistent.[12] This could be accomplished by adding two

[12] This is much like the problem of maintaining cache coherence in multiprocessors.

controllers to monitor the peripheral registers, as shown in Figure 12(b). A read is processed as before. On a write, the controllers oversee an independent, pipelined transfer of the new value from the original destination register to its *shadow*. The result is latched on ϕ_c, the only clock cycle where the registers are otherwise inactive. Since this operation will complete after the next instruction has already fetched its operands, there is no guarantee that every computation will have the effect intended. For example, the fragment

```
E0 ← E1 + E2
W2 ← min(W0,W1)
E3 ← W1 + E1
```

will not execute correctly because the sum stored in `E0` by the first instruction does not appear in `W0` until after the second instruction has retrieved the old value. It is not clear how often this read-after-write (RAW) hazard will occur in "normal" programs (the sorting loops of Figures 4 and 9 are hazard-free). If it were to arise, we could perform code rearrangement like that used for certain RISC machines [Henn83]. The above fragment does work if the second and third instructions are interchanged. In the worst case, it will always be possible to insert a NOP (e.g., `RX ← RX`, where `RX` is not the delayed register) to flush the pipeline.

The final decision to use either of these strategies, or perhaps one altogether different, can best be made when the B-SYS layout is nearer to completion.

Another system issue not yet resolved is how to keep the array running at full speed. We know from P-NAC that this can be a formidable task. Clearly, provisions must be made for the data to be stored on the same circuit board as B-SYS. In addition, instructions must also be provided at a high rate. Every B-SYS program written to date contains a main loop that is repeated many times, a fact that suggests we might profit from including an on-board writeable control store. Once the host computer has pre-loaded the loop body, instructions can be broadcast rapidly by a local micro-sequencer.

Programming Support

We have no desire to add yet another parallel programing language to an already crowded field. Instead, we feel that the necessary environment can be provided by embedding a few key constructs in an existing language, most likely C [Kern78]. The benefits of such an approach are already well documented. The general-purpose language furnishes a familiar foundation, upon which we can build a library of subroutines to manage a parallel data structure (i.e., B-SYS). As a consequence, major software tools (e.g., compiler, linker, semantic verifier) need not be created from scratch.

Nevertheless, there is one aspect of B-SYS programming that could be improved by adding another, higher level of abstraction: register allocation. Recognizing, for example, that the sorting code of Figure 4 requires two phases is a tedious detail we should not have to concern ourselves with. We would much rather write the program in Figure 13 and have a preprocessor unwind the loop and perform the mapping.

We have only recently begun to explore this problem. It appears to differ in important ways from previously studied cases because the B-SYS architecture is distinctive. Within the body of a loop we have a classic instance of register allocation. While the general form of this is NP-complete [Seth75], it can be solved efficiently if we are willing to forgo a reordering of the instructions to minimize register usage.

However, the shared register communication paradigm dictates constraints across iterations: certain registers store values, others transport them.

```
while (more input) {                    /* until we run out of values to sort ... */
    out_val ← min(in_val,save_val);          /* pass smaller value */
    save_val ← max(in_val,save_val);         /* save larger value */
}
```

Figure 13. *Improved pseudo-code for systolic sorting.*

Formalizing the above requires that we impose some structure on the use of registers and their symbolic names (i.e., variables). We suspect that many B-SYS applications can be supported with the introduction of two new data types: *state*, which corresponds to values retained locally, and *stream*, which corresponds to values moved through the array. The latter is a composite type containing an *input* and an *output* component. In the sorting code of Figure 13, the variable `save_val` is a state, while `in_val` and `out_val` are members of the same stream. When allocating registers across iterations, the following conventions seem appropriate:

- a state variable always remains in the same register
- input and output variables from the same stream alternate between two complementary registers

Figure 14 illustrates the allocation of registers for the program in Figure 13 as a modified interval-graph coloring problem. The dependencies within an iteration are depicted by arcs, the constraints across iterations by ovals. In this case, the nodes representing `in_val`, `save_val`, and `out_val` must all be assigned different colors. Note that `save_val` keeps the same color across iterations, while in any given iteration `in_val` assumes the color that `out_val` had in the previous iteration. After two iterations the coloring repeats. The allocation used in Figure 4 is obtained if we equate colors to registers as shown.

This formulation is only tentative; it conforms to B-SYS applications we have examined thus far, but probably will have to be extended to apply more generally. One obvious complication is systolic algorithms employing data streams that move at different rates. It may be that allocations using a small number of registers will need a large number of iterations to repeat, and vice versa. The question then becomes one of determining the "best" assignment for a particular system configuration (i.e., number of registers, control store size).

We now turn to the issue of a debugger. B-SIM is, at best, serviceable. While we do not yet know how difficult it will be to test B-SYS programs *in situ*, it is clear that the task would be greatly facilitated by an interactive debugger. At this stage we are beginning to consider several possible approaches. There is reason for optimism; as a parallel architecture, B-SYS is relatively simple, so this aspect of the project should require less effort than some others.

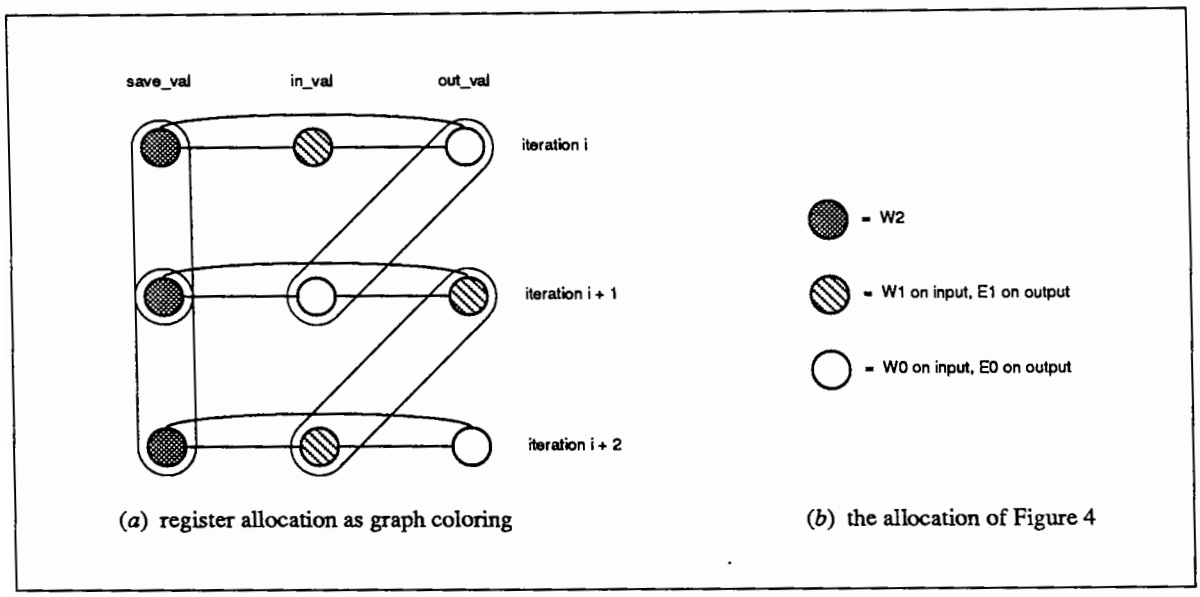

Figure 14. *An example of the register allocation problem.*

Application Development

Although sequence comparison provided the original motivation for B-SYS, the resulting array is remarkably general-purpose. We have already programmed a number of algorithms and are constantly seeking new applications. This exercise will take on added importance when the hardware becomes functional and we can contrast the performance of B-SYS to that of other machines.

Our analysis to date has focused on the linear architecture of Figure 2(a); we have not yet written a program specifically for the mesh architecture of Figure 2(b). Even though we have no plans to build such a machine in the near future, it would behoove us to investigate two-dimensional applications since one of our intentions is to promote the shared register communication paradigm.

In concluding this section, we note that a large body of engineering research has been directed towards the classification, design, and implementation of systolic algorithms for numeric applications (primarily image and signal processing). We plan to treat systolic algorithms for combinatorial applications from the same perspective.

6 Conclusions

This document was prepared with three goals in mind: first, to present an abstract architecture for programmable systolic arrays, second, to review the current status of the B-SYS project, and third, to indicate in which direction our work is headed. We believe that the shared register model described herein is a natural way of implementing systolic communication from the point of view of both the programmer and the hardware designer. The only disadvantage we can see is a need for special precautions to ensure that computation remains efficient. Through B-SYS we hope to test the validity of these ideas.

It seems appropriate to end this discussion by posing a hypothetical question: if it became possible to have too many B-SYS processors, which aspects of the architecture would we change? We can state, without hesitation, that we would prefer to increase the size of the shared register sets and the power of the ALU's. Incorporating a right shift instruction, for example, would give B-SYS the ability to divide by two and hence support asynchronous iteration, a technique to solve the two-dimensional partial differential equations arising in the study of fluid flows [Mano86]. It is unlikely we would modify the topology (linear) or the control methodology (SIMD); we do not consider these to be serious limitations in the present machine, given the broad class of applications we have already targeted.

7 Acknowledgements

Many people have influenced the development of B-SYS in one way or another. Students in CS 236 (VLSI Design) wrote programs using the B-SIM simulator and made a number of helpful comments. In particular, Glenn Carroll, Paul Howard, Marian Nodine, Gail Shaw, and Elizabeth Shriver all provided valuable feedback. We also thank John Savage for his careful critique of the first draft of this paper.

8 References

[Anna87] M. Annaratone, et. al., "The Warp Computer: Architecture, Implementation, and Performance," CMU-RI-TR-87-18, July 1987.

[Batch80] K. E. Batcher, "Design of a Massively Parallel Processor," *IEEE Transactions on Computers*, September 1980, pp. 836-840.

[Fish83] A. L. Fisher, et. al., "Design of the PSC: A Programmable Systolic Chip," *Proceedings of the Third Caltech Conference on VLSI*, R. Bryant, ed., Rockville, MD: Computer Science Press, 1983, pp. 287-302.

[Gray89] J. P. Gray and T. A. Kean, "Configurable Hardware: A New Paradigm for Computation," *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, C. L. Seitz, ed., Cambridge, MA: The MIT Press, 1989, pp. 279-295.

[Henn83] J. Hennessy and T. Gross, "Postpass Code Optimization of Pipeline Constraints," *ACM Transactions on Programming Languages and Systems*, vol. 5, no. 3, July 1983, pp. 422-448.

[Hill85] W. D. Hillis, *The Connection Machine*, Cambridge, MA: The MIT Press, 1985.

[Hugh88] R. Hughey and D. P. Lopresti, "Architecture of a Programmable Systolic Array," *Proceedings of the International Conference on Systolic Arrays*, K. Bromley, et. al., eds., Washington, DC: Computer Society Press, 1988, pp. 41-49.

[Kern78] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs, NJ: Prentice-Hall, 1978.

[Kung79] H. T. Kung and C. E. Leiserson, "Systolic Arrays (for VLSI)," *Sparse Matrix Proceedings 1978*, Society for Industrial and Applied Mathematics, 1979, pp. 256-282.

[Kung88] H. T. Kung, "Systolic Communication," *Proceedings of the International Conference on Systolic Arrays*, K. Bromley, et. al., eds., Washington, DC: Computer Society Press, 1988, pp. 695-703.

[Lang86] H. W. Lang, "The Instruction Systolic Array – A Parallel Architecture for VLSI," *Integration, the VLSI Journal*, vol. 4, 1986, pp. 65-74.

[Leis83] C. E. Leiserson, *Area-Efficient VLSI Computation*, Cambridge, MA: The MIT Press, 1983.

[Lopr87a] D. P. Lopresti, *Discounts for Dynamic Programming with Applications in VLSI Processor Arrays*, Ph.D. Dissertation, Princeton University, January 1987.

[Lopr87b] D. P. Lopresti, "P-NAC: A Systolic Array for Comparing Nucleic Acid Sequences," *Computer*, vol. 20, July 1987, pp. 98-99.

[Mano86] S. Manohar and G. Baudet, "Supercomputing with VLSI," *Proceedings of the 24th Annual Allerton Conference on Communication, Control, and Computing*, October 1986, pp. 585-594.

[Mead80] C. A. Mead and L. A. Conway, *Introduction to VLSI Systems*, Reading, MA: Addison-Wesley, 1980.

[Morl88] R. E. Morley, T. J. Sullivan, "A Massively Parallel Systolic Array Processor System," *Proceedings of the International Conference on Systolic Arrays*, K. Bromley, et. al., eds., Washington, DC: Computer Society Press, 1988, pp. 217-225.

[Mukh89] A. Mukherjee, "Hardware Algorithms for Determining Similarity Between Two Strings," *IEEE Transactions on Computers*, vol. 38, no. 4, April 1989, pp. 600-603.

[NCR85] *NCR GAPP Application Notes*, 1985.

[Pott85] J. L. Potter, ed., *The Massively Parallel Processor*, Cambridge, MA: The MIT Press, 1985.

[Sank83] D. Sankoff and J. B. Kruskal, editors, *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Reading, MA: Addison-Wesley, 1983.

[Schm86] H. Schmeck, "A Comparison-Based Instruction Systolic Array," *Parallel Algorithms and Architectures*, M. Cosnard, et. al., eds., Elsevier Science Publishers B. V. (North Holland), 1986, pp. 281-292.

[Seth75] R. Sethi, "Complete Register Allocation Problems," *SIAM Journal on Computing*, vol. 4, no. 3, September 1975, pp. 226-248.

[Shaw84] D. E. Shaw, "SIMD and MSIMD Variants of the NON-VON Supercomputer," *IEEE COMPCON '84 Proceedings*, March 1984, pp. 360-363.

[Snyd82] L. Snyder, "Introduction to the Configurable, Highly Parallel Computer," *Computer*, vol. 15, January 1982, pp. 47-56.

[Xili88] *XC3000 Logic Cell Array Family*, Xilinx, Inc., 1988.

Appendix A

```
#include "../src/opc2.h"
#define clear(x) zero x x x  nread file0 Zconst F7 F7 \n
#define Sfile       1                          /* source sequence (s[i]) */
#define Tfile       2                          /* target sequence (t[j]) */
#define Iweights    3                          /* initial distances (d[-1,j]) */
#define file255     4                          /* dummy input 255 */
#define file0       5                          /* dummy input 0 */

/*****************************************************************************
*   B-SYS program to compare two five-character sequences using five         *
*   processing elements.  Assumes that the cost of an insertion/deletion     *
*   is 1, and the cost of a substitution is 2.  Only this initialization     *
*   code need be changed for longer sequences.                               *
*                                                                            *
*   The algorithm performs the following steps:                              *
*     1. calculate min(d[i-1,j],d[i,j-1]) + 1                                *
*     2. check s[i] and t[j] for match                                       *
*     3. if they match, then d[i,j] is d[i-1,j-1],                           *
*        otherwise d[i,j] is the min from step 1                             *
*     4. make d[i,j] available to neighboring PE's                           *
*     5. move s[i]                                                           *
*                                                                            *
*   Registers usage:                                                         *
*     W1:  source character (s[i])                                           *
*     E0:  target character (t[j]) (stationary)                              *
*     E2:  previous distance (d[i,j-1])                                      *
*     E4:  previous previous distance (d[i,j-2])                             *
*     WF:  scratch                                                           *
*                                                                            *
*   Note that E2 and E4 alternate meaning every time-step to save a swap.    *
*   d[i,j] is calculated in processor j-1 at step 5(i+j)+11 and is placed    *
*   in E2 if 2|(i+j), otherwise d[i,j] is placed in E4.                      *
*                                                                            *
*   The final column of the distance matrix is output.                       *
*****************************************************************************/

/* clear communication register and read first character */
always zero E0 E0 E0 nread nwrite Zzero F7 F7
always clear(W2)
always clear(W4)
always clear(WF)
always clear(W1)

/* move target (stationary) sequence into B-SYS */
always fnA E0 E0 W0 nread Tfile Zconst F0 F0
always fnA E0 E0 W0 nread Tfile Zconst F0 F0
always fnA E0 E0 W0 nread Tfile Zconst F0 F0
always fnA E0 E0 W0 nread Tfile Zconst F0 F0
always fnA E0 E0 W0 nread Tfile Zconst F0 F0

/* move first character of source sequence into B-SYS and set F6 */
always zero E1 E1 E1 nread Sfile Zone F6 F6
```

Figure A1. *B-SYS sequence comparison initialization.*

```
#include "../src/opc2.h"
#define Sfile       1                   /* source sequence (s[i]) */
#define Tfile       2                   /* target sequence (t[j]) */
#define Iweights    3                   /* initial distances (d[-1,j]) */
#define file255     4                   /* dummy input 255 */
#define file0       5                   /* dummy input 0 */

/*****************************************************************************
*   B-SYS program to compare two five-character sequences using five          *
*   processing elements.  Assumes that the cost of an insertion/deletion      *
*   is 1, and the cost of a substitution is 2.  See initialization file       *
*   for further details.                                                      *
*****************************************************************************/

/* even phase */
always xorABC W2 E2 WF nread nwrite Zsub F7 F1
always selectABonC W2 E2 WF read file255 Zconst F1 F1
always xorAC WF WF WF nread nwrite Zadda F6 F1
always fnA W1 W0 E1 nread Sfile matchAB F7 F2
always selectABonC W4 WF E4 nread Iweights Zconst F2 F2

/* odd phase */
always xorABC W4 E4 WF nread nwrite Zsub  F7 F1
always selectABonC W4 E4 WF read file255 Zconst F1 F1
always xorAC WF WF WF nread nwrite Zadda F6 F2
always fnA W1 W0 E1 nread Sfile matchAB F7 F2
always selectABonC W2 WF E2 nread Iweights Zconst F2 F2
```

Figure A2. *B-SYS sequence comparison loop.*