An Architecture for Programmable Systolic Arrays

Richard Hughey Daniel Lopresti
Brown University Department of Computer Science
Providence, RI 02912, U.S.A.

Technical Report No. CS-89-08
February 1989

Abstract

Systolic algorithms are a systematic and powerful methodology for realizing the potential of very large scale integration. After an initial wave of special-purpose systolic processors, research has turned to the implementation of programmable systolic machines. These machines are often too large and complicated to be able to support massive parallelism: many of the benefits to be reaped from systolic algorithms are lost. Few of the current architectures provide the simple systolic communication of the early special-purpose processors.

The primary contribution of this research is the systolic shared register paradigm. Unlike previous research efforts, the architecture provides a natural means of systolic communication and uses simple processing elements; the architecture can be both easily programmed and easily formed into very large systolic arrays. The main features of the architecture are broadcast instructions, a regular topology, and register sets that are shared between neighboring processing elements. The Brown Systolic Array, an implementation of the concept, is also described. Using the Brown Systolic Array as an example, the programming of these machines is investigated.

An Architecture for Programmable Systolic Arrays[1]

Richard Hughey and Daniel Lopresti

February, 1989

[1]This paper is a slightly modified version of the first author's thesis proposal. This research was supported in part by the National Science Foundation under grant MIP-8710745.

Contents

1 Introduction

The systolic array is perhaps the most significant architectural development inspired by the recent revolution in VLSI fabrication technology. Since the introduction of the systolic paradigm, arrays have been proposed to solve a wide variety of problems including speech recognition, sorting and searching, image processing, and various dynamic programming algorithms. Unfortunately, the majority of these proposals assume special or single-purpose processors. Each new VLSI design requires several months to design and test. Because of this great effort to construct new arrays, research has turned to programmable systolic arrays. This paper proposes a general, programmable systolic array architecture: the systolic shared register (SSR) design. The systolic shared register design preserves the simple communication of special-purpose systolic arrays while providing the user with fully programmable processing elements.

The remainder of this introduction will review the systolic concept. Following this, several existing parallel machines will be examined. The body of the proposal will stress four areas: the general concept of an SSR architecture, the architecture of the Brown Systolic Array and its use of the SSR paradigm, the low-level programming of SSR machines, and finally several theoretical aspects concerning the control of systolic arrays.

1.1 Systolic Arrays

Systolic arrays were first proposed as a class of parallel architectures by H. T. Kung and C. E. Leiserson in 1978 [1, 2]. The utility of systolic arrays rests in their formalism for mapping algorithms to processor arrays. Instead of a labyrinth of individually programmed processing elements, the typical special purpose systolic array consists of one or two types of cells connected in a regular network, such as a line, ring, hexagonal mesh, or square mesh.

Systolic algorithms are designed with the knowledge that individual processing elements are unable, without great cost, to send information to arbitrary or distant locations. In a planar VLSI process, multiprocessors requiring arbitrary communication will use the vast majority of chip area for routing. The point of a systolic implementation is to put as much processing power as possible on a chip. Thus, a systolic algorithm will have the following properties [3]:

- The algorithm can be implemented by only a few different types of simple cells.
- The algorithm's data and control flow is simple and regular, so that cells can be connected by a network with local and regular interconnections. Long distance and irregular communication is thus minimized.

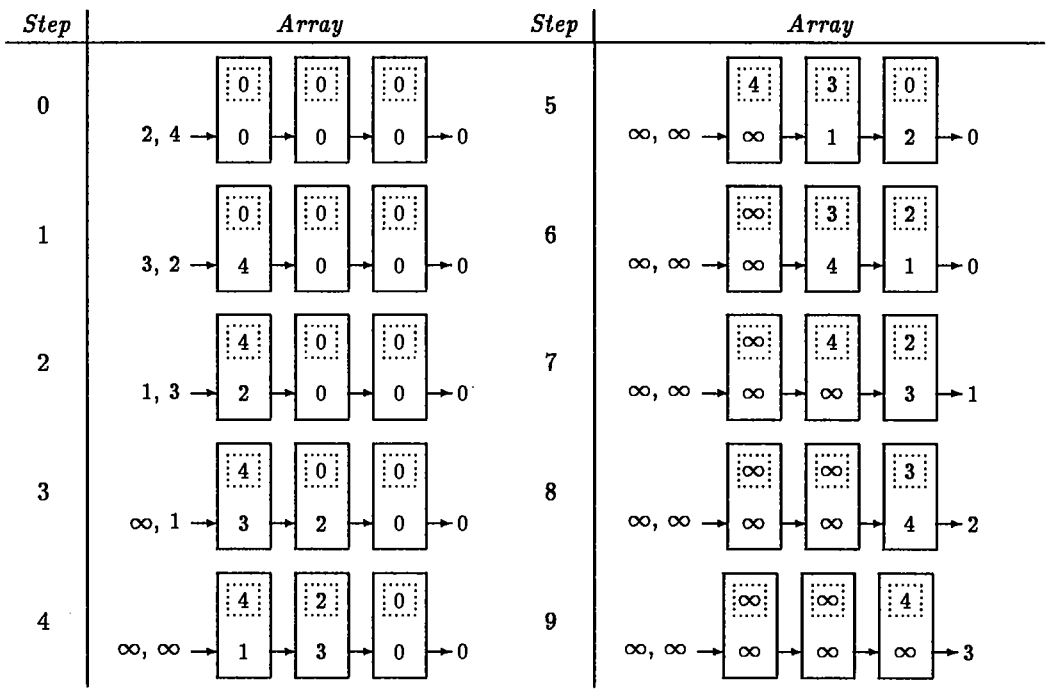
- The algorithm uses extensive pipelining and multiprocessing. Typically, several data streams move at constant velocity over fixed paths in the network, interacting at cells where they meet. In this way a large number of cells are active at one time so that the computation speed can keep up with the data rate.

As a result of these principles, data is written to and read from the processor array only at array boundaries, while computation takes place throughout the array. For example, data might enter on one side and results might exit on the other. The description of a systolic algorithm, therefore, must specify the operation of the systolic cells in addition to the data movement between the cells. Systolic arrays can efficiently manage large quantities of data without the potential for bus contention or network congestion.

The systolic concept has received much attention since its discovery. Special purpose systolic and semi-systolic processors have been built or proposed for speech recognition [4], sequence comparison [5], sorting and searching [6], signal processing [7, 8], the solution of partial differential equations [9], matrix operations [10], and many other problems [11, 12]. In the past several years, however, research has focused on the more general problem of constructing programmable systems capable of executing a variety of systolic algorithms. Before describing these designs and the place of the systolic shared register concept, it would be instructive to consider a short example.

Consider a systolic sorting algorithm on a linear array of processing elements. The basic function of each processor is to read a number from the left, compare it with a stored value, and pass the smaller to the right. In this way, the smaller values will exit the array (from the rightmost cell) in sorted order. Using this algorithm, a type of parallel bubble sort, $n+1$ numbers may be sorted on n processing elements in $2n+1$ steps. Each processor is assumed to have a local storage register and a scratch register. During each step, the processors will compare the number in the scratch register with the stored number and then keep the larger of the two numbers and pass the smaller to the right. For the moment, the problem of how processors send data to adjacent processing elements is ignored. Several steps of this sorting algorithm are shown in Figure 1, where the top register is the local storage and the bottom is the scratch register. The array is shown after the completion of each comparison and movement step. For example, during step 3 the values 4 and 2 are compared in the first processor. The value 2 is passed to the second while 4 is retained. The input value 3 is passed to the first processor at the same time it passed 2 to the second.

This example illustrates all the important principles of a systolic algorithm. Data moves through the array in a regular, synchronous fashion. All data movement is between adjacent processing elements. Finally, the basic operation of each cell is quite simple. For this reason, the replication of the basic cell to form a very large systolic array is easy using current technologies, such as VLSI or WSI (wafer scale integration).



The local storage register is enclosed in a dotted box. The lower scratch register is used for communcation. The array is shown after each complete step, comprised of a comparison phase and a data movement phase.

Figure 1: Systolic insertion sort of $4, 2, 3, 1$.

Since the sorting example exhibits the basic qualities of a systolic algorithm, it will be used both to compare the architectures of the next section and to illustrate some of the issues that arise when programming a systolic shared register machine (Section 5).

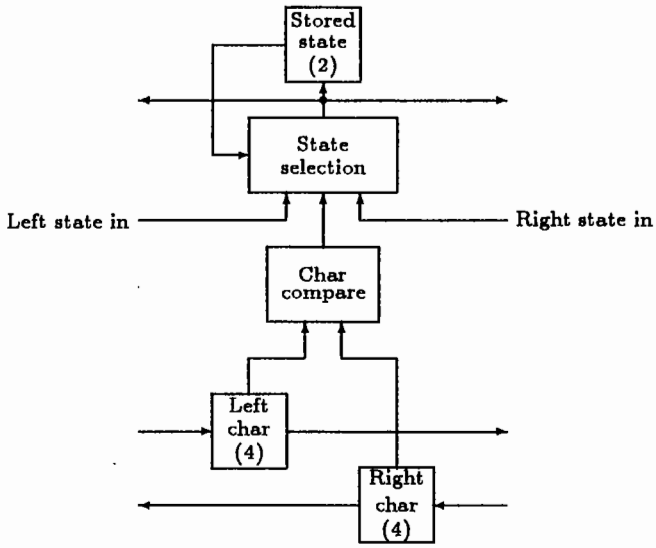


Figure 2: P-NAC architecture.

2 Related Research

This section considers several parallel architectures which have influenced both the systolic shared register concept and the Brown Systolic Array (B-SYS) implementation. First, the direct predecessor of B-SYS will be examined, followed by two bit-serial machines (the Massively Parallel Processor and the Connection Machine) and the Warp computer. Finally, several less publicized designs are summarized.

2.1 The Princeton Nucleic Acid Comparator

The Princeton Nucleic Acid Comparator (P-NAC) was developed by Daniel Lopresti [5, 13] and is the direct predecessor of the Brown Systolic Array. P-NAC is designed to solve the sequence comparison problem over a four character alphabet. Using the simple dynamic programming method of calculating edit distances [14], P-NAC provides optimal speedup by performing $O(n^2)$ computations in $O(n)$ time using $O(n)$ processing elements. The processing elements are connected in a linear array, and data is fed in from both ends of the array. Each processor is very small (the 1985 nMOS implementation placed 30 processors on a 4.6mm $\times$ 6.8mm chip with a feature size of 4μm), and the resulting chip can compare nucleotide sequences quickly. For example, a 1999 processor

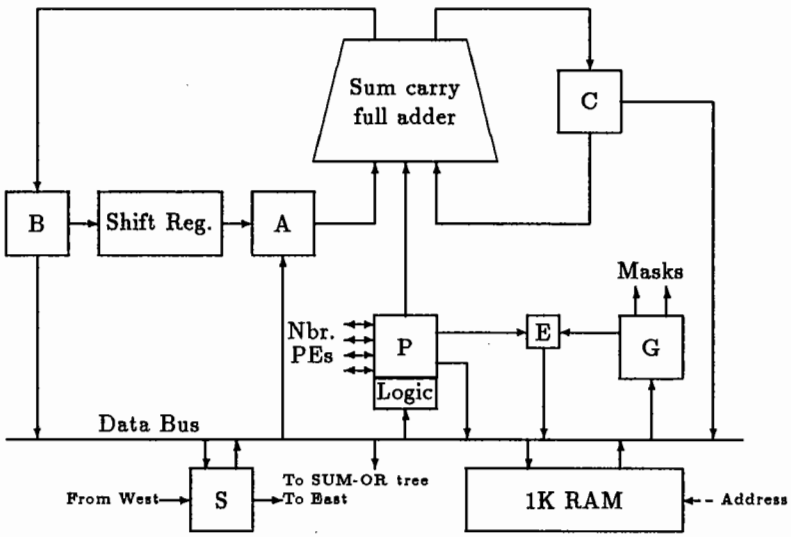


Figure 3: Architecture of the Massively Parallel Processor [15].

array of P-NAC processors can compare two 1000 element sequences in 130mS. The architecture of the P-NAC basic cell is shown in Figure 2.

Unfortunately, P-NAC can solve only a single problem. Even slight variations to the sequence comparison problem, such as using different costs in the calculation of the edit distance, can not be solved with P-NAC. Since building a new chip for each application is clearly inefficient, a programmable successor to P-NAC is needed.

Because only one algorithm is ever executed on P-NAC, systolic communication is achieved with latches and fixed connections. A special-purpose implementation of the sorting algorithm of Figure 1 would use a similar method of communication. A programmable systolic array should, if possible, include a method of systolic communication which is as simple as the intrinsic communication of these special-purpose architectures.

A programmable systolic array should preserve the vast parallelism easily attainable with P-NAC. Since nucleic acids such as DNA can be thousands of nucleotides long, thousands of processing elements can be used effectively to accelerate database searches for similar pairs of sequences. Without small and simple processors, such as those of the P-NAC system, the cost of such huge arrays would be prohibitive.

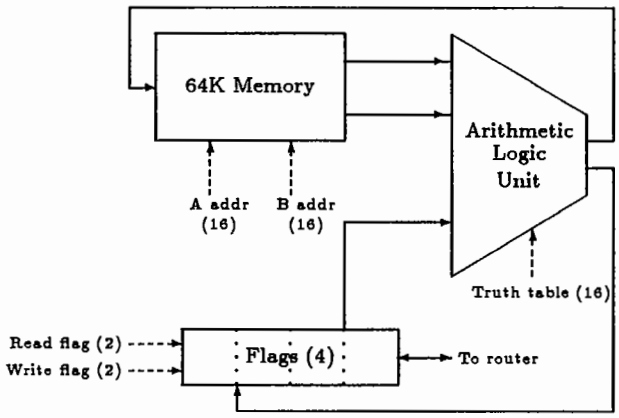


Figure 4: Connection Machine processor architecture [18, 19].

2.2 The Massively Parallel Processor

The Massively Parallel Processor (MPP) was developed jointly by Goodyear and NASA from 1978 to 1983 [16, 15]. The MPP features a 128×128 square (toroidal) mesh of bit-serial processing elements. Each processor has 36 bits of on-chip memory and can access another 1024 bits of off-chip memory, as can be seen in Figure 3. Processors can communicate by a direct read of a neighboring P register. Also, the S register may be used to shift data into the array in parallel with computation. The host machine has access to the logical 'or' of each processor's data bus. The processing elements are fabricated in CMOS, with eight processors per chip.

Since a single program sequencer broadcasts instructions to the processor array, the MPP is an SIMD (single instruction and multiple data stream) architecture according to Flynn's classification [17]. The more complicated MIMD (multiple instruction and multiple data steam) machines provide a control program for each processing element. The entire broadcast SIMD array uses 96 boards.

The MPP has been used for image processing applications, differential equations, fluid dynamics, and matrix operations. The architecture is similar to that of the Connection Machine (next section) without the extensive routing capabilities or the large amount of memory accessible by each processing element.

2.3 The Connection Machine

The Connection Machine features 64K (65,536) single-bit processors and a hypercube routing scheme [19]. Each Connection Machine processor (CM-2) can access 64K bits of memory and a floating point unit shared between several processing elements. The underlying mesh connection and the hypercube routing provide for the construction of arbitrary paths between processing elements. The four flag registers (see Figure 4) are available to neighboring and distant processing elements via the chip's router. The Connection Machine features "data level parallelism", which is to say that it is an SIMD machine.

Although the Connection Machine supports non-systolic algorithms, it can also efficiently execute systolic algorithms intended for a square mesh or linear array of processing elements. For example, Figure 5 displays a program implementing the sorting algorithm of Figure 1 shown using calls to the Connection Machine's low-level Paris language. The estimated number of basic bit operations for each Paris call is shown to the right of each statement: a total of 64 are needed for each basic operation of the sort program. A register (`WORK`) must be used to save the minimum, an issue which will be addressed in Section 5. The program will be similar for any bit-serial machine, such as the MPP or the GAPP system discussed later. Communication in this algorithm takes place much less intuitively than the synchronous communication of single-purpose systolic arrays: the data must be explicitly moved through the array with `send` commands which in turn use the communications registers.

It might also be pointed out that bit-serial operations are slower than bit-parallel. In the sequence comparison case, approximately ten times as many instructions are required on a bit-serial machine as on a bit-parallel machine with a word-size of eight bits. The potentially faster clock speed of a bit-serial machine does not completely overcome this problem. The enormous number of processors in CM-2 partially compensates for this by providing enhanced parallelism.

2.4 Warp

The Warp computer was developed at Carnegie Mellon University under the guidance of H. T. Kung, and is a very powerful systolic array processor [20]. The present model, PC-Warp, consists of ten 10-MFLOP (million floating point operations per second) processors connected in a linear array. The array cells are independently programmable, each containing memory for 8K instructions. Each Warp processing element features a 32-bit multiplier, a 32-bit adder, and 32K words of local memory. Between neighboring processors there are three communications queues able to store 512 32-bit words each. The components of the processor are connected with a 32-bit crossbar

```
#define LOCAL 0                                                    < steps >
#define SCRATCH 8
#define WORK 16
#define INFTY 255
#define BYTE 8

sort_list (num, n)
int *num, n;

{
   /* Set masks, clear registers, etc. */
   initialize();
   for (i=0; i < n ; i++) {
      /* Copy min to a working register */
      CM_u_min_3_1L (WORK, LOCAL, SCRATCH, BYTE);                  < 16 >
      /* Copy max to the local register */
      CM_u_max_2_1L (LOCAL, SCRATCH, BYTE);                        < 16 >
      /* Move the min to the next processor */
      CM_send_to_news_1L (SCRATCH, WORK, 1, CM_upward, BYTE);       < 8 >
      /* copy in the new input value to processor 0 */
      CM_u_write_to_processor_1L (0, SCRATCH, num[i], BYTE);        < 8 >
   }
   /* The numbers in the array: now take them out */
   for (i=0; i < n ; i++) {
      /* Copy min to a working register */
      CM_u_min_3_1L (WORK, LOCAL, SCRATCH, BYTE);                  < 16 >
      /* Copy max to the local register */
      CM_u_max_2_1L (LOCAL, SCRATCH, BYTE);                        < 16 >
      /* Move the min to the next processor */
      CM_send_to_news_1L (SCRATCH, WORK, 1, CM_upward, BYTE);       < 8 >
      /* copy infinity value to processor 0 */
      CM_u_write_to_processor_1L (0, SCRATCH, INFTY, BYTE);         < 8 >
      /* Read output value from processor n */
      num[i] = CM_u_read_from_processor_1L (n, SCRATCH, BYTE);      < 8 >
   }
}
```

Figure 5: Sorting using Paris 5.0 (unverified).

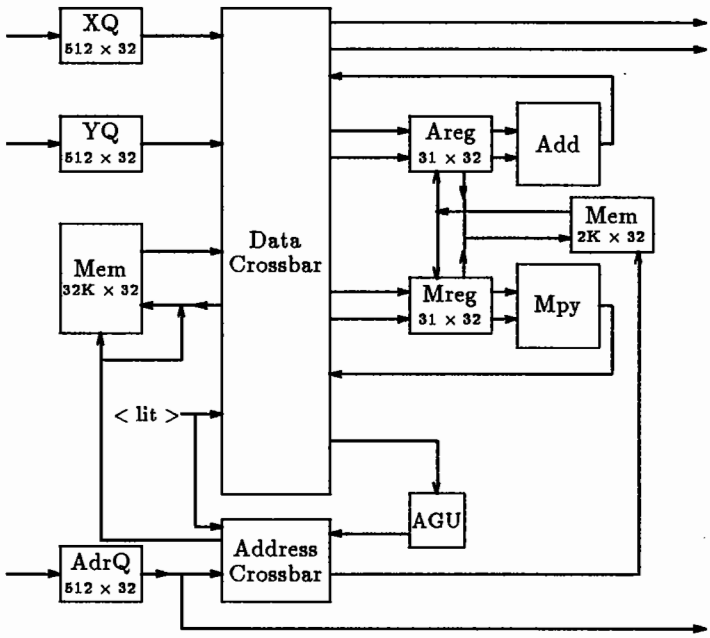


Figure 6: Warp data path [20].

switch. Thus, data may be placed on the queues directly from the units of CPU or from the processor's local memory. Warp provides hardware control of the queues: a full or empty queue will cause the appropriate processor to wait until this condition changes. Figure 6 shows the data path of an individual Warp cell. A new version of Warp, called iWarp, is being developed jointly with the Intel Corporation and is expected to have 72 processors and more memory per processor.

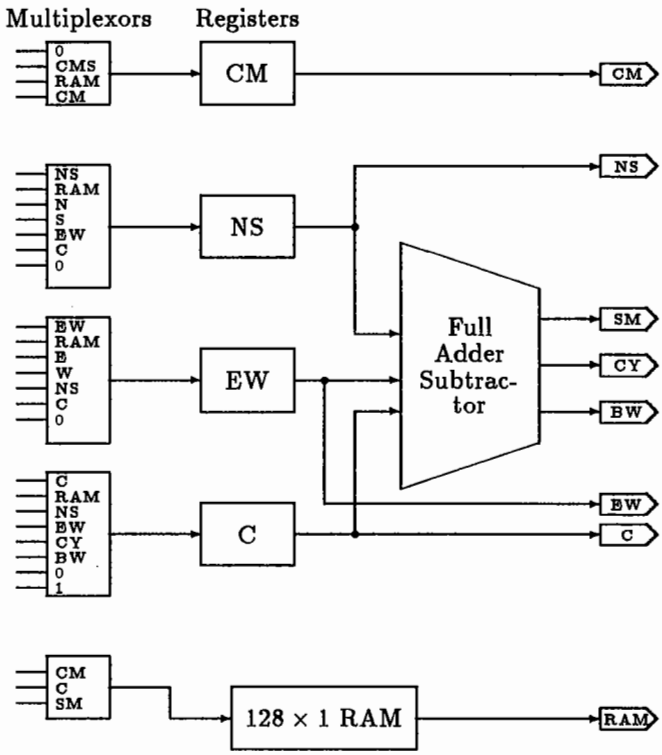
A high-level language, W2, has been developed for use with the Warp system [21]. W2 features the usual language constructs in addition to `send` and `receive` instructions for data movement. Possibly the most basic structure in the W2 language is the `cell program` which specifies a program for some or all of the array cells. Using these constructs, the sorting program of Figure 1 would look something like that shown in Figure 7. The `send` and `receive` statements complicate the simple systolic motion of the sorting algorithm. These two communications statements do, however, provide a method of relating input data (`num[n]`) and output data (`results[n]`) to specific queues. Support for such a notion makes systolic programming much more intuitive in W2 than in Paris (Figure 5), where data is explicitly loaded into or read from the end processing elements.

```
module sort_list (num in, n in, results out)
integer num[n], n, results[n];

   /* The cell program, for use in all 10 cells */
   cellprogram (cid : 0 : 9)
   begin
      function sort
      begin
          integer local, scratch;
          integer i;

          local := 0;
          for i := 1 to n do begin
             /* Read a number from the left on the X queue. */
             /* Use num[i] as an input for processor 0 */
             receive (L, X, scratch, num[i]);
             /* Send the minimum to the right on the X queue */
             send (R, X, min (scratch, local));
             /* Compute and save the maximum */
             local = max (scratch, local);
          end;
          for i := 1 to n do begin
             /* Read a number from the left on the X queue. */
             /* Use INFTY as an input for processor 0 */
             receive (L, X, scratch, INFTY);
             /* Send the minimum to the right on the X queue */
             /* Store the rightmost value in the results array */
             send (R, X, min (scratch, local), results[i]);
             /* Compute and save the maximum */
             local = max (scratch, local);
          end;
      end;
      call  sort;
   end;
```

Figure 7: Sorting on Warp using W2 (unverified).



Inputs: E, W, N, S (from neighbors) Outputs: NS, EW (to neighbors)
Input: CMS (from South neighbor) Output: CM (to North neighbor)

Figure 8: Geometric Arithmetic Parallel Processor [23].

The Warp system can perform many signal and image processing applications quickly [22]. However, it is not especially suited for symbolic or simple computations on large amounts of data. Since the size of the array is limited by the size of the processing elements (one board each), Warp is unable to make full use of the vast parallelism inherent in many problems.

2.5 Other Systems and Building Blocks

This section describes several additional systems as well as a few processors suited for systolic machines.

T. J. Fountain has surveyed several SIMD bit-serial image processing systems [8]. Each of the eight machines surveyed have between 8 and 64 processors per chip. The processors are connected in square mesh arrays, much like the MPP. All these systems are "variations on a theme", and the Massively Parallel Processor represents them well.

The NCR Geometric Arithmetic Parallel Processor (GAPP) is an interesting bit-serial SIMD array [23]. Each GAPP chip has a 6×12 square mesh of processing elements. The processing elements (Figure 8) feature 128 bits of memory, a full adder/subtractor, and 4 flag registers. The NS and EW registers form the mesh connection, each being readable by two adjacent processing elements. Much like the MPP, GAPP allows the user to shift data into the array in parallel with the operation of the ALU via the CM register. The GAPP chip has been used at Washington University to form a 48×48 toroidal mesh which can perform 900 million 8-bit additions per second [24].

Among the more powerful systolic processors is the Programmable Systolic Chip (PSC) developed at CMU [25]. The PSC featured simpler processors than those used with Warp (an 8-bit word, but still including a multiplier), and was designed for use in linear arrays as well as various other topologies. Other high-power programmable systolic arrays have been constructed using the INMOS Transputer and other digital signal processing (DSP) chips [26, 27], as well as custom designed processors [28, 29].

Finally, the proposed Instruction Systolic Array (ISA) is a mesh of simple processing elements in which *both* data and control flow through the array [30, 31]. Instructions flow in one direction (north to south) and selectors (or mask bits) flow in the other (west to east). The ISA thus uses a clever solution to the problem of controlling n^2 processing elements with only n control bit inputs each time step. The first proposal of the Brown Systolic Array considered the use of systolic instructions in a linear systolic array [32]. However, the problems with this concept probably outweigh the benefits in a linear systolic array. The use of systolic instructions is considered in Section 7.

The examples of this section, from P-NAC to Warp, represent three schools of systolic thought: the simple special-purpose array, the SIMD array of bit-serial processors, and the MIMD linear array of powerful processors. The special-purpose architectures lack versatility, while the MIMD machines lack massive parallelism. All three types of architecture present a specialized and not entirely satisfactory method of implementing systolic communication.

3 The Systolic Shared Register Paradigm

This section proposes the systolic shared register (SSR) architecture. The architecture builds on the problems and merits of the systems discussed in Section 2. First, the simple systolic communication of P-NAC and the sorting example of Figure 1 must be maintained. In contrast to special-purpose systolic arrays, however, the architecture must be programmable. Also, simple processing elements, such as those in the Connection Machine, GAPP, and MPP are needed to be able to form very large systolic arrays. However, the vast amounts of memory and complicated communications schemes of these machines are not needed. The three main features of the systolic shared register architecture are:

- broadcast instructions
- regular topology
- shared systolic registers.

The SSR idea results from two new ways of looking at systolic computation. First, the SSR concept considers the systolic system as a whole. Systolic systems are traditionally thought of as an array of processing elements, with memory, connected by several nearest-neighbor data paths. More efficient systolic machines can be designed if the processors and the memory are logically split into separate units. Second, there is no distinction made between data memory and communications memory. This is somewhat reminiscent of early computer engineers discovering that data and program memories do not need to be distinct in hardware.

3.1 Broadcast Instructions

Per-processor program stores require too much area for inclusion in a high-density systolic array. This penalty comes both from the size of the required program memory and the complexity of the instruction sequencer. To design systems with large-scale parallelism, both of these should be on the same chip as the processor. MIMD processing elements are too complicated to fit more than a few on a single chip using current technology.

When large numbers of MIMD processors (hundreds or thousands) are used for systolic algorithms, the programs stored in each processor tend to be similar. Even with a relatively small number of processors, published Warp programs, for example, typically only differentiate one of the end processors while all other processors execute the same cell program. This lack of exploitation of the inherent power of MIMD machines is a strong argument for only a single instruction stream. While a systolic instruction

stream, as discussed in Section 7, is an interesting concept, broadcast instructions offer a much simpler and more efficient method of control. For these reasons, broadcast instructions, which play an important part part in the shared register design of an SSR architecture, are the method of choice.

3.2 Regular Topology

To support systolic algorithms, all of which use a regular topology by the definition of Section 1.1, the SSR architecture itself must have a regular topology. A regular network (such as linear, square mesh, or hexagonal mesh) provides for the orderly and extensible mapping of algorithms to the systolic array. Of these choices, a linear network is perhaps the best choice for a general purpose processor array. A linear network extends easily and requires only two I/O (input/output) streams, one for each end, no matter how large the array is.

The use of regular topologies enforces the systolic principle of locality. To each processing element, possibly even those at the boundary, the world must look the same: there are some number of neighbors, each with a *relative* name or direction. In a mesh, these relative names might be north, south, east, and west. This is very important for two reasons. First, a regular connection network is easily extensible to larger numbers of processors. Second, relative processor names, which are commonly used, greatly facilitate the use of broadcast instructions.

3.3 Shared Memory

Before turning to the inter-processor communication scheme of SSR machines, consider the distinction between the systolic data movement of special (or single) purpose arrays and that of programmable systolic arrays. In the special purpose array, data movement is implicit in the hardware. For example, a processor designed specifically for a sorting problem (such as that of Figure 1) might compare a value with an internally stored one and make the smaller value available to the next processor. However, the execution of these operations is regarded as a single step. Thus, data is pictured as moving through the array at a speed of one processing element per step. Since the processors perform only one function, the timing of the data movement can be built into the hardware.

In a programmable systolic array, data can not be moved automatically since different applications require different types of data movement. Thus, in a programmable systolic array, data movement must be explicitly specified by the user. For Warp, this entails placing data on special queues. In an SSR architecture, data moves through the array as a natural result of the user's program.

Figure 9 shows a segment of a linear SSR processor array and a detail of a single

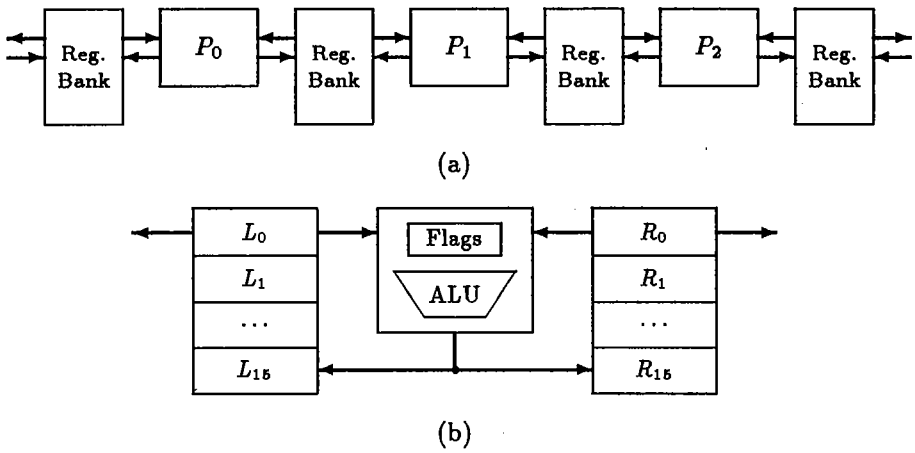


Figure 9: Linear SSR processor array (a) and processor detail (b).

processor. Note that each processor (denoted P_i) is adjacent to two register banks. Each processing element can access data values from the register banks directly to its left and right for both input and output. Likewise, each register bank can send and receive data from two adjacent processing elements. Systolic communication in an SSR architecture is through the use of these shared register banks. As can be seen from the processor detail, there are no internal registers in the processing elements except for a small number of flag bits. This separation of processing elements and memory is the main distinction of an SSR architecture.

Each instruction broadcast to the SSR array specifies the relative addresses of memory locations: specific registers in either the left or the right register bank. During the execution of an instruction, each processing element will read operands from relatively addressed registers and, after the computation, write the result to a relatively addressed register. Because relative addresses are being used, there is no potential for two neighboring processing elements to attempt to read or write the same register during the same part of the instruction execution cycle. While there is no potential for hardware conflicts, software conflicts using the shared register banks must be avoided (discussed in Section 5). The lack of hardware conflicts is a benefit of the regular topology required by the SSR design.

The implementation of systolic data movement via shared registers is quite intuitive.

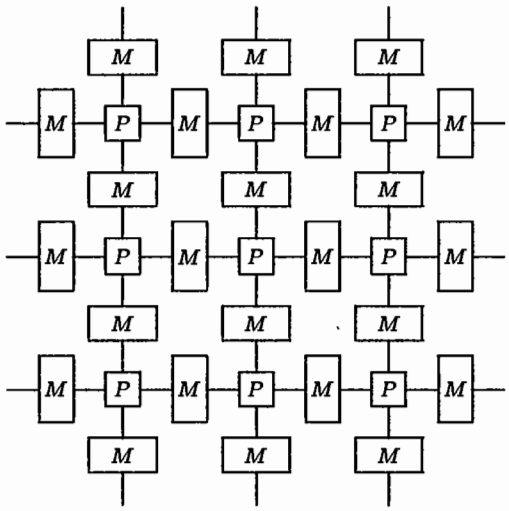
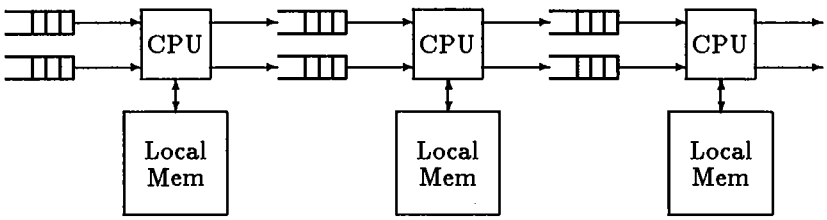


Figure 10: SSR mesh of processors (P) and memory (M).

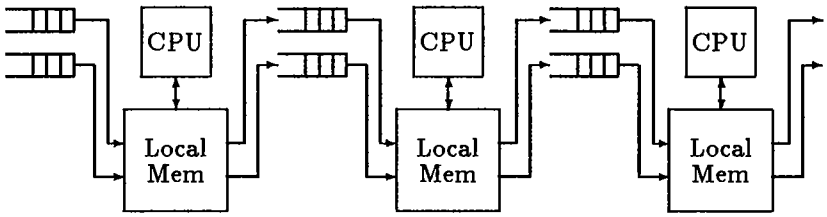
For example, if a processor reads two items from the left register bank, performs some computation on those items, and places the result in the right register bank, information has systolically moved through the array and is now available to the next processing element. No instructions are required to initiate specific types of data movement, such as the `send` and `receive` statements of W2 (Figure 7). On the bit-serial machines discussed in Section 2, data must be placed in special communication registers before it may be accessed by adjacent processing elements. In the case of the Connection Machine (Figure 5), this is done by the Paris `send` instruction, in which the router accesses the processor's flag registers (Figure 4). The MPP and GAPP machines (Figures 3 and 8) have registers dedicated to communication. No extra synchronization statements or special registers are needed in an SSR machine.

The shared register method does not require the use of special queues between processors. The shared register paradigm also does not limit the number of communication channels between processors. The user may decide to use the first few registers of each register bank for local storage, the next for transmission of data from left to right, the next for another data track, and so on. In summary, shared registers provide a small, fast, and elegent implementation of systolic communication.

The use of shared registers may be naturally extended to other topologies, such as the mesh (see Figure 10). As with the linear case, there is a single memory bank between



(a) Systolic communication



(b) Memory-to-memory communication

Figure 11: Methods of inter-processor communication [33].

every pair of adjacent processors. No matter what type of network is used, each SSR memory bank is only connected to two processing elements.

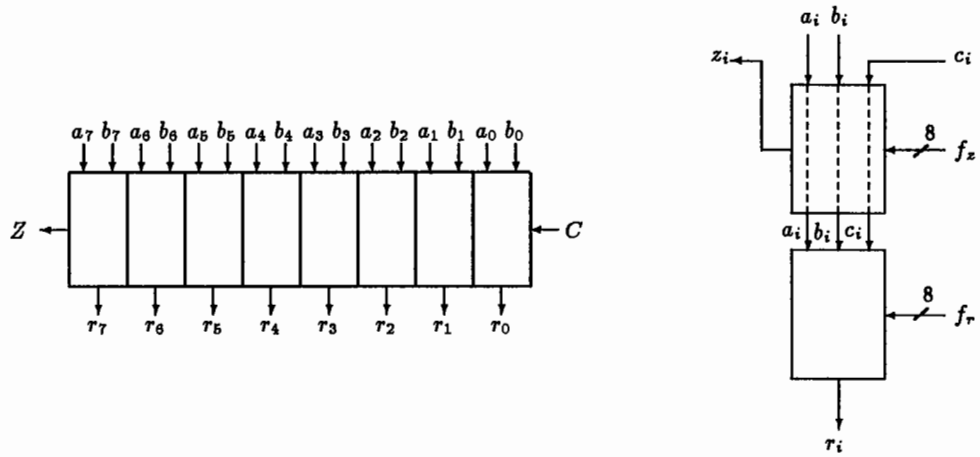
3.3.1 Shared memory and systolic communication

It is instructive to examine the SSR design in light of the definition of systolic communication proposed by H. T. Kung [33, 34]. Kung separates memory-to-memory communication and systolic communication. In the former, information is passed from the memory of one processing element to the memory of another via communication queues. In systolic communication, however, information may be both read from and written to the communication queues directly by a processor's central processing unit (CPU). In both cases, there is a distinction between the queue memory used for communication and each processor's own local memory. Figure 11 shows these two methods for providing communication between processing elements in a linear array.

The distinction between the two types of memory is important. The communications memory defined by Kung only allows sequential access (as with Warp), while the local memory allows random access. Thus, a systolic array machine is one which can make

direct use of data on its input queues. Since the register banks of an SSR architecture are essentially a set of one-element, bi-directional queues, the SSR architecture is systolic by this definition. However, the SSR idea has taken the idea of systolic communication one step further: the input communications register banks allow random access, making local memories superfluous. Since SSR systems do not distinguish between memory types, the CPUs not only communicate directly with their communications registers, but *only* communicate with those registers (Figure 10).

Another difference between communication queues and registers is that the SSR paradigm's use of communication registers forces synchronous execution on the system. If an item is to be read from a specific register, it must be written to that register by an adjacent processor before that time. This is a simple task with synchronous, broadcast instructions and closely resembles the data movement of special purpose systolic arrays.



(a) B-SYS arithmetic logic unit.

(b) Bit-slice detail.

Figure 12: B-SYS arithmetic logic unit.

4 The Brown Systolic Array

In this section, the Brown Systolic Array architecture is described. B-SYS is a linear implementation of the general SSR architecture. It features shared registers, instruction broadcasting, and simple processing elements. It is expected that at least fifty of these processing elements and register banks will fit on a single chip. An overview of the simple arithmetic logic unit (ALU) and its microprogramming is given in this section, along with a description of the proposed system interface.

4.1 Arithmetic Logic Unit

Each B-SYS processing element has an ALU in addition to several one-bit registers, used for both the storage of carry flags and the processing of conditionals. The processors communicate via shared register banks, each with 16 registers. The B-SYS implementation has 8-bit words and 8 flags per processing element. Figure 9b shows an individual B-SYS processor. Simplicity has been an overriding design consideration for the rapid production of this SSR prototype.

The ALU, shown in Figure 12a, has two single word inputs, A and B, and a single bit input C. The outputs are a result word R and a bit Z. The ALU has a bit-slice architecture: each bit of the output is computed in a manner identical to all other bits in the ALU, and indeed the whole chip. The i-th bit slice is detailed in Figure 12b. Observe

that the bit-slice has two parts: each is an arbitrary function logic block able to evaluate any three-to-one function. This design is similar to that of the OM-1 processor from Mead and Conway [35]. The inputs to each of these blocks are identical: a_i, b_i, and c_i, the former being the i-th bits from A and B, while c_i is the flag output z_{i-1} of the previous stage (and is equal to C in the least significant bit-slice). The outputs of the bit-slice are a result bit r_i and a flag bit z_i. Note the distinction between the words A and B, and the individual bits a and b. Similarly, C is the input to the entire ALU while c refers to the propagated flag input of an individual bit slice.

Each logic block is controlled by an eight-bit lookup table. The top logic block, controlled by the function f_z, produces the flag output that is sent to the next processing element. The bottom logic block uses f_r to determine the result bit. For example, the carry generated for integer addition may be logically written as $ab + ac + bc$, generating the function table:

c	b	a	z
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Thus, the hexadecimal control word for the flag block, f_z, would be $E8_{16}$. Similarly, $f_r = 96_{16}$ will generate the sum bit for an addition. In the generation of control words, a is the least significant bit while c is the most significant bit.

4.2 B-SYS Microcode

As has been seen in the previous sections, each B-SYS instruction must specify several fields. The 37 bit horizontal microcode format of an instruction is shown in Figure 13. The instruction word may conveniently be divided into two parts: the register inputs and outputs, along with f_r, and the flag inputs and outputs, along with f_z. Each broadcast instruction has the following fields:

- f_r: an eight bit function for the calculation of R.
- A: one bit specifying which bank (left or right) and four bits specifying which register in that bank is the A input to the ALU.
- B: five bits, as with A.

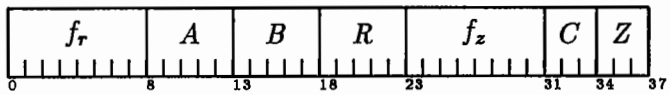


Figure 13: B-SYS instruction word.

Name	*Function*	*hex*	*Name*	*Function*	*hex*
zero	0	00	one	1	FF
fnA	a	AA	fnC	c	F0
xorAC	$a \oplus c$	5A	andAC	ac	C0
borrowAC	$\overline{a}c$	50	carryAC	ac	A0
notAxorC	$\overline{a} \oplus c$	A5	xorAB	$a \oplus b$	66
xnorAB	$a \odot b$	99	andAB	ab	88
nandAB	$\overline{a \cdot b}$	77	orAB	$a + b$	EE
norAB	$\overline{a + b}$	11	xorABC	$a \oplus b \oplus c$	96
carryABC	$ab + ac + bc$	E8	borrowABC	$\overline{a}b + \overline{a}c + bc$	D4
selectABonC	$ac + b\overline{c}$	AC	selectABonC_	$a\overline{c} + bc$	CA
notzeroA	$c + a$	FA	zeroA	$c\overline{a}$	50
equalAB	$c \cdot (a \odot b)$	90	matchAB	$c + (a \odot b)$	F8

Table 1: Some B-SYS control words.

- R: five bits, as with A, specifying the register in which to store the result.
- f_z: an eight bit function for the calculation of Z.
- C: three bits specifying which local flag is the flag input to the ALU.
- Z: three bits specifying the flag register in which to store the flag output.

All fields are required, and there are no immediate operands. The f_r and f_z control words are broadcast to each bit-slice of the ALU, while the remaining fields are used by the processor to access the correct memory elements. If the user has need for neither the R nor the Z output of an operation, the result must be placed in a scratch register. If an algorithm needs certain constants, they must be loaded into one of the registers beforehand. Often, the first step of a program will be to place constants in several flag registers and clear several locations in the register banks. Constants needed in all processing elements can be constructed in a small number of B-SYS statements without having to shift them through the array.

The first flag in each processor (Figure 9b) is a mask register. When the mask flag is not asserted, the processing element will not write to any registers or flags. Instructions

will be provided, of course, to force the writing of data in all processors regardless of the mask bit. The use of this flag will be described in Section 5.2.1.

Several control words for the B-SYS ALU are listed in Table 1. Each of these represents a single hexadecimal control word, usable either for f_r or f_z. Obviously, some operations make more sense in one case than in the other. For example, use of a select instruction as f_r tends to force f_z to be the identity function on the carry, fnC. The last four operations are test instructions: with appropriate values of C (0 or 1, depending on the instruction), Z will be the result of the test. The carry chain is used to pass the result of the test. For example, when testing A for zero (f_z = zeroA), an input of $C = 1$ should be used. On receiving a $c = 0$ input, each bit-slice will pass $z = 0$. If $c = 1$, then there are no previous non-zero bits, and the bit a is checked. Depending on a, either $z = 0$ or $z = 1$ is passed to the next bit-slice.

4.3 System Interface

The previous two sections have considered the processing elements and their programming. The reader may have noticed that the method of getting data to and from B-SYS has been completely ignored. The B-SYS chip will need hardware support to communicate with its host computer. Because of the linear network connecting processors, the design of the board will effectively be the same for any number of processors. Indeed, the only constraint is how much of the board is left for B-SYS chips after the support logic is in place. This section will propose the requirements for that interface.

Consider the operation of a single B-SYS instruction on the registers: two registers are read, a computation is performed, and one register is written. Looking at the register banks of Figure 9a, if the write is to register L_1, register 1 of every register bank except one receives a new value. Since no processor is writing to the right-most register bank of the array, it seems logical that the board write an input value to that location. This is indeed the approach that B-SYS will use. Similarly, if R_2 is read as the A input to the ALU, no processor will be reading register 2 of the left-most register bank. The board will read this value and send it to the host machine if it is of interest. To limit each instruction to at most one output, only the A input register will be read by the board. Each instruction passed to the board will thus have two additional bits of information specifying whether a value should be passed form the board to the host (the unread A register) and whether a value should be passed from the host to the board (the unwritten R register).

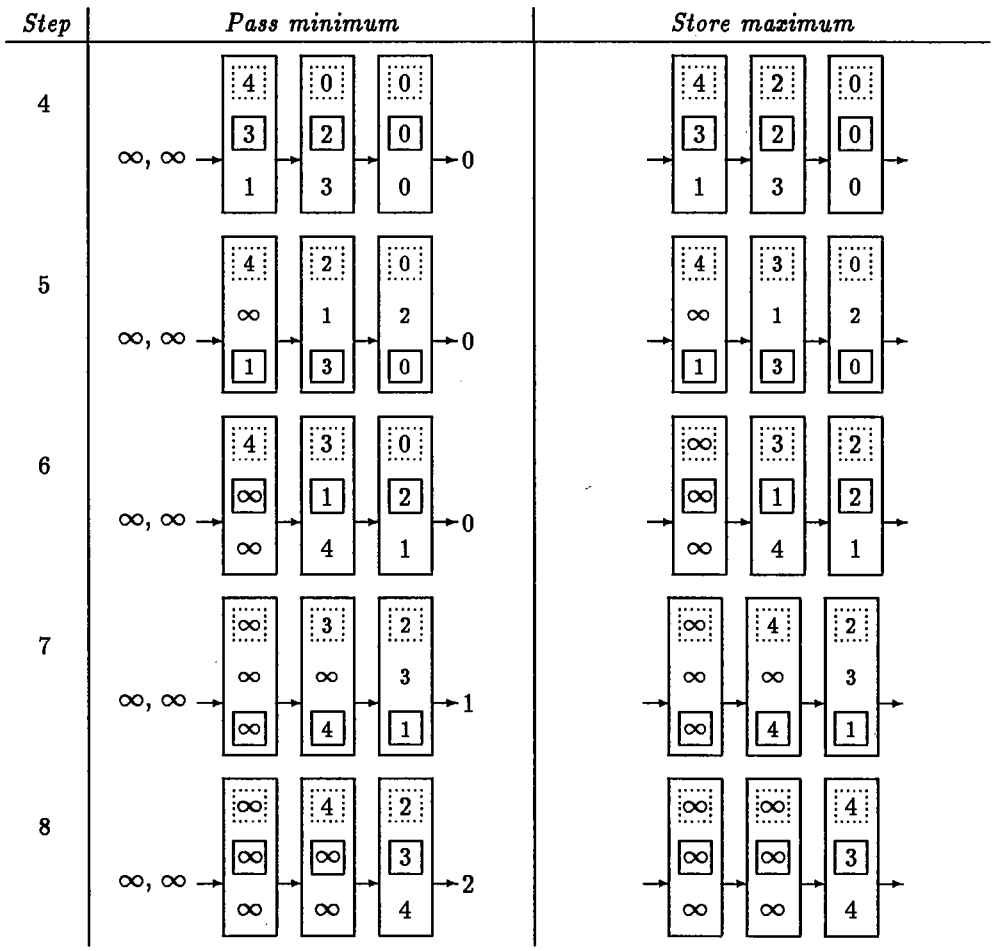
One potential bottleneck in the system is the need for an instruction each time step, while data might be transferred between the host and the board at a much lower frequency. For example, 8-bit multiplication requires around 40 B-SYS instructions with no external I/O. The solution to this inefficiency is to place a small program cache

on the B-SYS board. Instructions from the host can execute specific subroutines stored in the cache. If the board has both an input and output data queue, the host can send and receive data while instructions from the program cache are being executed. Suitable logic will enable B-SYS to continue executing instructions until either the output queue is full or the input queue is empty. The GAPP system referred to in Section 2.5 uses a similar instruction cache to speed up the operation of the co-processor [24].

4.4 Additional Design Options

This section describes several other features which may, time permitting, be included in the B-SYS implementation. It would be advantageous to place more registers in each register bank, say 32, to facilitate multiple-precision and floating-point arithmetic. A study should be made on the advantages of faster RAM circuitry using sense amplifiers [36]. Also, the arithmetic logic unit might perform much faster by adding a Manchester carry chain [35]. The right-shift operation could be a worthwhile addition to the processing elements. A more powerful board can, of course, only speed up the interface between the host and the systolic array.

Finally, a future design goal is to experiment with other SSR topologies such as the mesh shown in Figure 10.



The local storage register is enclosed in a dotted box. The scratch register is shown in a solid box. The remaining register is used to receive data from the adjacent processing element.

Figure 14: Phased systolic sort.

5 Programming the Brown Systolic Array

This section examines several programming issues that arise with any SSR architecture, and specifically with the B-SYS architecture. The discussion assumes that a specific mapping of the computations to the linear array processors has already been made. Solutions to this difficult problem may be found in the literature [37, 38, 39, 40, 41].

The two programming issues which will be examined involve register allocation and the execution of conditional statements. In the next section, these principles will be applied to several problems.

5.1 Register Allocation and Data Movement

Moving data through an SSR machine is more complicated than simply reading two operands from the right register bank (R) and storing the result in the left register bank (L). Since L is the right register bank of an adjacent processor, writing to it may invalidate data that the adjacent processor needs. Taking another look at the sorting algorithm of Figure 1, the operation of storing the maximum and passing the minimum has been conveniently lumped into one instruction. Since instructions typically only have one output value, this is somewhat unrealistic. Suppose the program is:

$$\begin{aligned} R_1 &\leftarrow \min(L_0, L_1) \\ L_0 &\leftarrow \max(L_0, L_1). \end{aligned}$$

Provided that register 0 is used for local storage and register 1 for scratch storage, this program appears to have the same effect as the single instruction version. However, R_1 for the i-th processor is L_1 of the $(i+1)$-st processor. Thus, writing to R_1 invalidates the maximum computation of the $(i+1)$-st processor.

There are two solutions to this problem: the use of scratch registers and the use of phased programs. Scratch registers may be used in the normal way, for example:

$$\begin{aligned} L_2 &\leftarrow \min(L_0, L_1) \\ L_0 &\leftarrow \max(L_0, L_1) \\ R_1 &\leftarrow L_2. \end{aligned}$$

At the cost of one register and one instruction, the problem has been solved.

The other alternative involves the use of a *phased* program. A phased program has several distinct phases in which different registers are used. For example, the scratch register might be R_2 in phase 1 and R_1 in phase 2. Switching back and forth between these two phases will result in a correct program which, though an extra register is still

used, does not use the extra instruction. Two steps of the sorting program would then be:

$$\begin{aligned} R_2 &\leftarrow \min(L_0, L_1) \\ L_0 &\leftarrow \max(L_0, L_1) \\ R_1 &\leftarrow \min(L_0, L_2) \\ L_0 &\leftarrow \max(L_0, L_2). \end{aligned}$$

Sorting with this technique is shown in Figure 14. The current scratch register is boxed, while the top register is still used for local storage. During the first part of each phase, the minimum of the boxed scratch value and the top stored value is passed to the unused register of the adjacent processing element. During the second part, the maximum of the two values is locally stored. This technique can easily be applied to programs of more than two phases.

5.2 Conditionals

This section considers two methods for processing conditionals: the mask bit and the select statement.

5.2.1 Use of the mask bit

The B-SYS mask bit is similar to that of the Connection Machine or the Massively Parallel Processor (GAPP does not have a mask bit). Whenever the mask bit is set, the processor executes each instruction broadcast to the array. Otherwise, the processor will remain idle. A special set of instructions will be provided to force all processing elements to write data, regardless of the state of the mask bit.

The classical method of using a mask bit for a nested if/then statement involves four steps: preserving the old mask bit in a flag register, copying the desired flag register to the mask bit, executing the conditional code, and finally restoring the mask bit to its previous value. The operations to do this, presuming that F_1 is the flag upon which execution is to depend, are:

$$\begin{array}{ll} f_r : \mathsf{fnA}(L_F, L_F) \rightarrow L_F & f_z : \mathsf{fnC}(M) \rightarrow F_2 \\ f_r : \mathsf{fnA}(L_F, L_F) \rightarrow L_F & f_z : \mathsf{fnC}(F_1) \rightarrow M \\ \multicolumn{2}{c}{\vdots} \\ \multicolumn{2}{c}{\langle \text{conditional execution code} \rangle} \\ \multicolumn{2}{c}{\vdots} \\ f_r : \mathsf{fnA}(L_F, L_F) \rightarrow L_F & f_z : \mathsf{fnC}(F_2) \rightarrow M. \end{array}$$

Thus, an overhead of three statements exists for each conditional. For this reason, use of the mask bit is often too cumbersome.

5.2.2 Register selection

An alternative to the extra statements required for mask bits is the use of selection statements. A selection statement simply chooses one value or another depending on a logical condition. Due to the architecture of B-SYS, this condition is always the value of a flag.

Programming a conditional with selection is straightforward: all possible values are computed and then the correct ones are chosen depending on the flags. Selection thus requires at least one scratch register, possibly more depending on the complexity of the program. Selection does not necessarily require extra program statements. For example, the statement:

$$\mathsf{If}\ L_0 < L_1\ \mathsf{then}\ R_2 \leftarrow L_0\ \mathsf{and}\ L_1\ \mathsf{else}\ R_2 \leftarrow L_0$$

may be programmed as:

$$\begin{array}{ll} f_r : \mathsf{xorABC}(L_0, L_1) \rightarrow L_F & f_z : \mathsf{borrowABC}(F_7) \rightarrow F_6 \\ f_r : \mathsf{andAB}(L_0, L_1) \rightarrow L_3 & f_z : \mathsf{zero}(F_7) \rightarrow F_7 \\ f_r : \mathsf{selectABonC}(L_0, L_3) \rightarrow R_2 & f_z : \mathsf{fnC}(F_6) \rightarrow F_6, \end{array}$$

Assuming that $F_7 = 0$, the first line compares L_0 and L_1, storing the borrow flag output in F_6 and the result in L_F, a scratch register. The second line computes the value of $L_0 L_1$, storing it in the scratch register L_3. Finally, a selection between L_3 and L_0 is made based on F_6.

Taking a comparative look at the C programming language, this is similar to the C construct:

$$(\ \langle \text{condition} \rangle\ ?\ \langle \text{statement1} \rangle\ :\ \langle \text{statement2} \rangle\ ).$$

In the bit-parallel world, register selection is often more efficient than using the mask bit. However, if several different results are calculated based on the value of one condition, use of the mask bit could yield a shorter program. On a bit-serial machine, calculations depending on a conditional usually involve an entire word, whatever the size of a word may be. Because of this, selection statements are not efficient on such machines since each result bit would have to be selected.

6 Applications

Many example programs have been written using B-SYM, the Brown Systolic Array Simulator. This section first considers a sequence comparison program After this example, several algorithmic and programming environment issues are discussed.

6.1 Sequence comparison

As was pointed out in Section 2.1, sequence comparison was the initial driving force behind the B-SYS project. The goal is to calculate the simple edit cost of transforming one sequence (s) into another (t) using the elementary operations of character insertion, deletion, and substitution. The result computed provides a measure of the "closeness" of the two sequences. For example, the string CAT may be transformed to $CART$ with one insertion of the letter R. Of course, more letters could be inserted and deleted ($CAT \rightarrow CATS \rightarrow CARTS \rightarrow CART$), but the goal is to find the minimum cost of a transformation. More complicated string comparison problems have been described [13, 42], and most can be solved on the B-SYS architecture.

The method used with B-SYS is a standard dynamic programming algorithm. Included in the algorithm is a character deletion or insertion cost of 1 and a character substitution cost of 2. The CAT to $CART$ transformation thus has a minimum cost of 1 due to the single insertion. The recurrence for the algorithm is:

$$d_{i,j} = \begin{cases} \min(d_{i-1,j}, d_{i,j-1}) + 1 & \text{if } t_i \neq s_j \\ d_{i-1,j-1} & \text{otherwise.} \end{cases}$$

Here, $d_{i,j}$ represents the cost of the partial matching of $s_1 \ldots s_j$ to $t_1 \ldots t_i$. With the given costs, it is always best to take the second case if the characters match and the first if they do not. Since the substitution cost is consistent with the cost for a deletion followed by an insertion, it need never be used.

A look at the dynamic programming table for the comparison of $CAABA$ and $ACCAA$ should clarify the algorithm. Note the initial values included in the table.

		t_0	t_1	t_2	t_3	t_4	t_5
		ϕ	C	A	A	B	A
s_0	ϕ	0	1	2	3	4	5
s_1	A	1	2	1	2	3	4
s_2	C	2	1	2	3	4	5
s_3	C	3	2	3	4	5	6
s_4	A	4	3	2	3	4	5
s_5	A	5	4	3	2	3	4

```
#define clear(x) zero x x x nread file0 zero F7 F7
#define Sfile 1
#define Tfile 2
#define file0 5
one L0 L0 L0 nread nwrite one M M
      print
zero L0 L0 L0 nread file0 zero F7 F7
zero L1 L1 L0 nread nwrite one F6 F6
clear(L2)
clear(L4)
clear(L5)
clear(L1)
fnA R0 R0 L0 nread Tfile fnC F0 F0
fnA R0 R0 L0 nread Tfile fnC F0 F0
fnA R0 R0 L0 nread Tfile fnC F0 F0
fnA R0 R0 L0 nread Tfile fnC F0 F0
fnA R0 R0 L0 nread Tfile fnC F0 F0
      printc Initialization complete: L0 has ti, the fixed string.
```

Figure 15: Sequence comparison initialization program.

From the above table, the cost of matching the two strings is $d_{5,5} = 4$. This corresponds to deleting the initial AC from s and adding BA to the end of the string. The systolic algorithm implemented for this example stores t in the processor array while s moves through the array. Thus, the processing elements may be called by their i index, and each one will calculate one column of the above table. The j index increments with time. The program for the simulator has two logical parts: an initialization program and a loop program.[1] The initialization program is displayed in Figure 15 and the loop program in Figure 16. Both of these programs make use of the C macro preprocessor. Appendix A includes the simulator outputs: snapshots of the systolic array as the computation progresses (several registers and flags are not shown).

The initialization program first sets the mask bit and clears several registers. Next, the string t is loaded into the systolic array. During the computation, t_i will always be stored in the R_0 register of P_i. The characters of s move through the array in the L_1 register. Each processing element must store two cost values of the above table in the right register bank, since both $d_{i-1,j}$ and $d_{i-1,j-1}$ are needed from P_{i-1} for the

[1] P-NAC uses a systolic mapping in which one sequence travels from right to left and the other from left to right. In a single purpose machine, this eliminates the need for an initialization phase. This in turn simplifies the processing elements.

calculation of $d_{i,j}$ by P_i. These are stored respectively in R_2 and R_4. Since when j is incremented, the old $d_{i-1,j}$ becomes $d_{i-1,j-1}$, a two phase program is used. In one phase, R_2 and R_4 hold values in the order previously stated, and in the next the meanings of the two registers are swapped.

The loop program is continuously executed after initialization until the end of the simulation. The first line of the program performs the comparison of $d_{i-1,j}$ and $d_{i,j-1}$ using subtraction. Next, the minimum of the two is chosen and stored in L_F, a temporary register which is then incremented. As the register is incremented, a large input value is put in the leftmost register bank so that the input will never be selected as the minimum weight. The fourth instruction checks to see if t_i matches s_j and shifts the s string through the array. Here, an s_j value is read into the leftmost register bank. Finally, a selection is made depending on the result of the previous matching instruction. The correct value for $d_{i,j}$ is stored in R_4. During this instruction, a $d_{0,j}$ value is written to the leftmost register bank. The next five instructions perform the exact same operations for the second phase, with the meanings of R_2 and R_4 swapped.

Characters in this simulation are represented by specific bit positions. Thus, $A = 1$, $B = 2$, and $C = 4$. This is to accommodate the use of wild cards in sequence comparison. For example, the number 5 will match either A or C. The character '$*$' represents the wild card, while 'ϕ' is the null character.

Referring to the B-SYM output (Appendix A), the computation of $d_{i,j}$ takes place in processor j and is completed at time $12 + 5(i + j)$. In this computation, the values $d_{i-1,j-1}$ and $d_{i,j-1}$ have been stored in the left register bank of P_j by P_{j-1}. The value $d_{i-1,j}$ was computed and stored by P_j during the previous computation cycle. Thus, $d_{5,5}$ is computed at time step 62 and placed in R_2.

6.1.1 Estimated performance

A systolic co-processor can be handicapped by its host machine. If, for example, the interface to the host machine is relatively slow, the true power of the systolic array is not being utilized. P-NAC (Section 2.1) performance suffered by a factor of about seventy because of a slow board interface. For this reason, the *throughput* speed of a systolic array can be used when comparing similar systolic architectures. The throughput speed of a machine is simply the maximum rate at which the systolic array itself can process data with no concern for delay due to the host. Table 2 summarizes the throughput speeds of P-NAC, B-SYS, and GAPP. The GAPP performance is estimated from the system built at Washington University [24], while the B-SYS estimate makes use of the simulated performance of initial chip designs. In this chart, the speed is represented as $T(c^*, M, n)$, where c^* is the throughput clock speed, M is the number of processing elements, and n is the length of the sequences being compared. Since these three linear

arrays use simple and easily replicated cells, the number of processors M is assumed to be the optimal for the sequence length (n in the case of B-SYS and GAPP, $2n-1$ for P-NAC). Using this many processing elements, the computation speed is directly proportional to the time it takes to pass the data through the array.

B-SYS is expected to perform better than many supercomputers on problems such as sequence comparison. This results from the power of the SSR concept and the systolic paradigm in general. Daniel Lopresti has programmed the sequence comparison problem on several general-purpose computers and supercomputers. The results, summarized in Table 3, are mostly extrapolated from a single run with 100 sets of 100 length sequences. The Connection Machine program simply did 100 comparisons at once using 100 processors. A potentially faster algorithm would move the sequences through the underlying mesh connection in a systolic manner. The values for B-SYS and GAPP are once again estimated, but this time including the cost of board to host communication. The P-NAC values are from the actual system running on a Sun-2. Note that in the case of the general-purpose computers, the speed of the algorithm decreases proportionately to the square of the length of the sequence.

6.2 Other Algorithms

Although it might not seem intuitively possible, matrix multiplication and all its extensions [44] may be programmed on a linear systolic array. The version that has been used on the B-SYS simulator was proposed by Lee and Kedem [43]. As with many systolic algorithms, such as sequence comparison, the most obvious sequential algorithm ($O(n^3)$) is best suited for systolic implementation. The algorithm provides linear speed-up (to $O(n^2)$) for this problem. Since it is a fairly involved algorithm, the specific steps of the algorithm will not be discussed here.

There are several other algorithmic issues that need to be addressed when building a large array of simple processing elements. Suppose a 1000 element string comparison needs to be done on an array of only 500 processors? This specific problem has been solved by Lipton and Lopresti [45], while Moldovan and Fortes have examined the general partitioning and mapping problem [41].

The mirror image of the previous problem is comparing 500 length strings on a 1000 element array: what should be done when there are too many processing elements in the array for the problem at hand. This is very easy to solve with an MIMD architecture, since the superfluous processors can simply relay data from left to right, performing no other operations. With an SSR architecture, one should be able to easily construct programs to run on large arrays. Software tools should be developed that, given an algorithm, array size, and data length, can correctly modify the data or algorithm to make use of the large array. Another interesting problem in this area is how to

compare many short sequences against many other short sequences without padding each sequence to 1000 characters. All these questions can be expanded to many other symbolic and numeric algorithms.

At the moment, all B-SYS and B-SYM programming is done in an assembly language mediated by the C macro preprocessor. Some work has been done on specifying what is needed for complete programming environment [46]. Implementation of the system would be the next step. In a similar vein, subroutine libraries for common functions usable by C programs would be very helpful. Finally, the question of what routines should be stored at any given time in the on-board instruction cache needs to be addressed.

```
#define Sfile 1
#define Tfile 2
#define Iweights 3
#define file255 4
xorABC L2 R2 L5 nread nwrite borrowABC F7 F1
selectABonC L2 R2 L5 read file255 fnC F1 F1
      printc L_F now has min(d_{i-1,j}, d_{i,j-1}). Output is d_{n,j-1}.
xorAC L5 L5 L5 nread nwrite andAC F6 F1
      printc L_F incremented.
fnA L1 L0 R1 nread Sfile matchAB F7 F2
      printc Test match of s_j(L_1) and t_i(L_0) and shift s_j to right.
selectABonC L4 L5 R4 nread Iweights fnC F2 F2
      printc Select between min(d_{i-1,j}, d_{i,j-1})(L_F) and d_{i-1,j-1}(L_4)
             depending on match, storing result in R_4 which is now
             d_{i,j}.
xorABC L4 R4 L5 nread nwrite borrowABC F7 F1
selectABonC L4 R4 L5 read file255 fnC F1 F1
      printc L_F now has min(d_{i-1,j}, d_{i,j-1}). Output is d_{n,j-1}.
xorAC L5 L5 L5 nread nwrite andAC F6 F2
      printc L_F incremented.
fnA L1 L0 R1 nread Sfile matchAB F7 F2
      printc Test match of s_j(L_1) and t_i(R_0) and shift s_j to right.
selectABonC L2 L5 R2 nread Iweights fnC F2 F2
      printc Select between min(d_{i-1,j}, d_{i,j-1}(L_F) and d_{i-1,j-1}(L_2)
             depending on match, storing result in R_2 which is now
             d_{i,j}.
```

Figure 16: Sequence comparison program loop.

System	$T(c^*, M, n)$	Speed (100 comparisons) $T(c^*, M, 100)$	$T(c^*, M, 1000)$
P-NAC	$4c^*n$	13 mS	130 mS
B-SYS	$13c^*n$	32 mS	320 mS
GAPP	$217c^*n$	217 mS	2.2 S

Table 2: Sequence comparison throughput speeds.

System	M	$T(c,M,n)$	Speed (100 comparisons) $T(c,M,100)$	$T(c,M,1000)$
B-SYS	n	$13cn$	55 mS	0.6 S
iWarp[a]	72	$5c\frac{n^2}{M}$	8.6 mS	0.9 S
GAPP	n	$217cn$	700 mS	7.0 S
P-NAC	$2n-1$	$4cn$	910 mS	9.1 S
PC-Warp	10	$5c\frac{n^2}{M}$	100 mS	10 S
CM-2	$64K$	cn	4.7 S	47 S
Trace[b]	48	$c\frac{n^2}{M}$	3.7 S	370 S
Cray-2[c]	64	$c\frac{n^2}{M}$	6.5 S	650 S
VAX-8600	1	cn^2	31 S	3100 S
Sun-3/140	1	cn^2	48 S	4800 S
VAX-11/785	1	cn^2	64 S	6400 S

[a]PC-Warp and iWarp times are estimated and are based on what are probably throughput timings.

[b]The Multiflow Trace computer used in this example had 48 processing units.

[c]Only one processor is used. The vector processor operated on 64 element vectors.

Table 3: Sequence comparison speed.

7 Formal Program Transformations

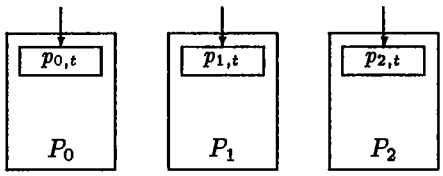
The final research goal of this project is to determine the abstract relationships between various types of systolic architectures. Given a fixed topology, there are several different methods of providing instructions to systolic arrays. The most powerful design allows direct access to each processor's instruction register (an MIMD machine). Methods appropriate to simpler processors are broadcast instructions, as used by an SSR architecture, and systolic instructions which pass through the array, as with the ISA system.

Another important part of any systolic program is the method of control: when and how individual processing elements may be turned on and off. The most powerful method here is to have direct external control of each processing element, but the requirement of one input for each processor from the host computer would be prohibitive for large arrays. One alternative method is the internal calculation of mask bits, such as B-SYS will use. Also, mask bits may be systolically moved through the array, such as the selectors in the ISA design. Figure 17 shows the various combinations of these possibilities that will be described in this section.

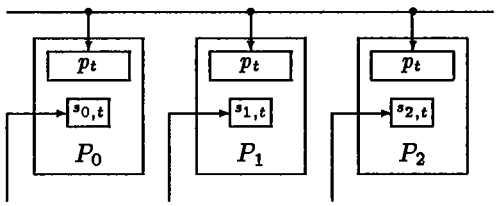
Two types of program can be considered on these machines: general programs $\mathcal{P}$ and systolic programs $\mathcal{P}_m$. A systolic program $p \in \mathcal{P}_m$ is a program in which all processing elements may be grouped into m equivalence classes such that all processors in the same equivalence class perform the same function. This is to say that m distinct cell programs are required. Note that m is independent both of the number of processors and the size of the problem. Many systolic algorithms, such as those discussed previously, are members of $\mathcal{P}_1$: each processing element executes exactly the same program. P-NAC uses a systolic algorithm in which processors alternate between comparing characters and calculating the next state. Since adjacent processors are executing a different operation during each time step, the P-NAC sequence comparison algorithm is a member of $\mathcal{P}_2$. The study of this class of programs may lead to a proof that systolic instruction streams in linear systolic arrays are no better than broadcast instructions.

The remainder of this section describes each of the models of Figure 17. Following this, several results are stated without proof.

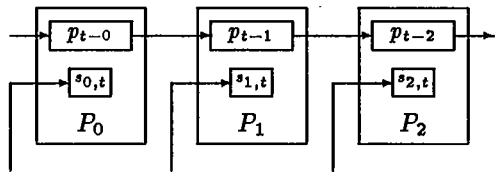
DI Direct instructions provide the most powerful method of control for a systolic array. Since arbitrary instructions may be passed to individual processing elements, there is no need for selector bits. Typically, the MIMD processor array is pre-loaded with individual programs, as with Warp, but such details do not affect this theoretical discussion. If the program is data dependent, some synchronization problems could exits between processing elements This potential problem can be solved with communication queues implemented in the hardware of the machine, as with



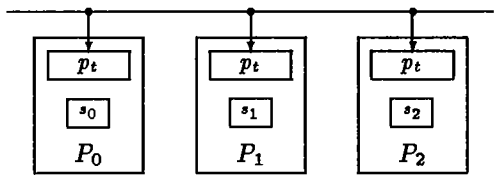
DI
Direct Instructions



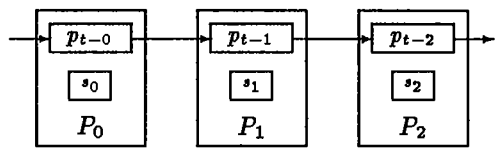
BD
Broadcast instructions, direct selectors



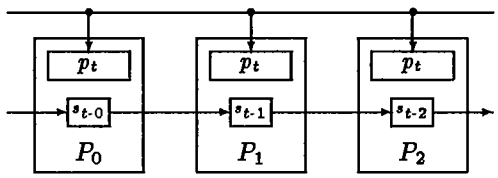
SD
Systolic instructions, direct selectors



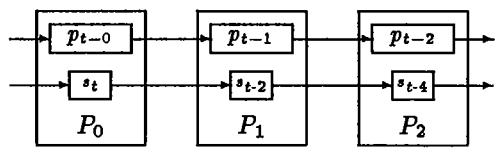
BI
Broadcast instructions, internal selectors



SI
Systolic instructions, internal selectors



BS
Broadcast instructions, systolic selectors



SS
Systolic instructions, systolic selectors

Each processing element has an instruction register (top) and a mask or selector register (bottom). The data paths between neighboring processing elements are not shown. A host computer provides all external inputs.

Figure 17: Systolic array control models.

Warp, or with software tools to synchronize the data streams. Here, it is assumed that software tools have synchronized the programs so that straightline code may be used.

A program $p \in \mathcal{P}^{\mathrm{DI}}$ consists of a sequence of statements labeled $p_{i,t}$ for $1 \leq i \leq n$ and $1 \leq t \leq l$, where n is the number of processors in the array and l is the length of the programs, which are assumed to be the same. Processor i (P_i) executes $p_{i,t}$ at time t.

BD A linear broadcast array with direct selectors provides arbitrary and fast selection of processing elements in the array. The use of direct selectors is excessive since they are not needed for systolic algorithms. This is, however, still a useful model to consider since it represents the best that can be done with broadcast instructions. A program $p \in \mathcal{P}^{\mathrm{BD}}$ has an associated selector matrix S, such that $s_{i,t}$ is the one bit selector for P_i at time t. If $s_{i,t}$ is asserted, then P_i will execute p_t. Otherwise, the processor will idle.

BI This is a SIMD linear broadcast array with systolic communication, such as B-SYS. The data moves through the array systolically while the program is broadcast to the processors. Instruction control is achieved using internal selector flags: during any time cycle, either the broadcast instruction is executed or it is ignored, depending on the flag. A program $p \in \mathcal{P}^{\mathrm{BI}}$ is a sequence of instructions p_t with $1 \leq t \leq m$, where each instruction may be tagged for a specific set of processors, depending on their flag values. At time t, P_i will execute p_t if its internally calculated selector is true.

BS Systolic selectors, a bit stream moving through the array turning processing elements on and off, is an attractive method for controlling a broadcast array. There is, of course, some lag between starting the bit stream and having all processors assigned to an equivalence class. Despite this delay, it is simple to place alternate processors in different classes. For example, a bit stream of $\{01\}^*$ (01010...) will alternatively turn on and off the odd and the even numbered processors ($\mathcal{P}_2^{\mathrm{BS}}$). Similarly, $\{001\}^*$ will divide the processors into three groups ($\mathcal{P}_3^{\mathrm{BS}}$). Each program $p \in \mathcal{P}^{\mathrm{BS}}$ has an associated selector bit stream S such that s_t is input to P_0 at time t. Processor P_i will execute p_t at time t if s_{t-i} is asserted ($t \geq i$). Correctly synchronized instruction broadcasting will yield correct execution of systolic programs.

SD As with the BD machine, this is not a practical model but serves to illustrate the most powerful use of systolic instructions. A program $p \in \mathcal{P}^{\mathrm{SD}}$ has an associated selector bit array S such that $s_{i,t}$ is the selector for processor i at time t. If $s_{i,t}$

Model	Simulated On						
	PA	BI	BD	BS	SI	SD	SS
PA	1	$n+1$	n	n	$2n$	n	$2n$
BI	1	1	1	n			
BD	1	$n+1$	1	n			
BS	1	2	1	1			
SI	1				1	1	n
SD	1				$n+1$	1	n
SS	1				2	1	1

Table 4: Simulation time costs for arbitrary programs.

is asserted, then P_i will execute instruction p_{t-i} ($t \geq i$), otherwise the instruction will only be passed to P_{i+1} for use during the next time cycle.

SI The program and initial configuration of the SI machine are very similar to those of the BI model. Given a program $p \in \mathcal{P}^{\mathrm{SI}}$ and a time $t \geq i$, processor P_i will examine instruction p_{t-i}. At times $t < i$, P_i will not execute any instructions. If the tag of the instruction and the processor match, P_i will execute p_{t-i} and pass p_{t-i} along to P_{i+1}. If the tags do not match, the instruction is only passed along.

The first B-SYS paper considered the use of a systolic instruction stream with internal selectors [32]. However, broadcast machines require less circuitry (no instruction latches) and execute several systolic algorithms more efficiently.

SS An array with both systolic selectors and systolic instructions poses several potential timing problems. Obviously, the selectors and the instructions cannot move through the array in the same direction at the same speed. If that were the case, each instruction would either be always executed or never executed. There are three solutions: moving the instructions and the selectors in the same direction but at different speeds, moving them in different directions, and finally using two streams of selectors which travel either in different directions or at different speeds, or both. Using the first method, S is a bit stream of selectors such that at time t, P_i will use selector s_{t-2i} on instruction p_{t-i}.

Given these seven types of machines, results can be proven about the translation of programs from one model to another. For arbitrary programs, some results are shown without proof in Table 4. In the translations, each processing element is forced to execute the same instructions in the same order on the same set of data as the original program. It is hoped that future research will lead to the completion of Table 4 and the quantification of the relationship between broadcast and systolic instructions. In addition, the class of systolic programs will be examined.

8 Conclusions

This proposal has presented a new architecture for systolic processing: the systolic shared register model. The SSR architecture combines the features of special purpose systolic arrays (many processors, synchronous data movement) with the power of programmable systolic machines. The idea of sharing registers between processors ties the processing elements tightly together. The shared register concept provides a simple and practical method of implementing systolic communication.

The Brown Systolic Array will be built to demonstrate these principles. It is expected that approximately 1000 systolic processors will fit on a single board, making B-SYS a programmable co-processor for systolic algorithms. Several applications have already been programmed on a simulator to demonstrate its usefulness.

The SSR paradigm may be useful for all types of parallel processors. Work beyond the proposed thesis could include implementing a square mesh machine using the principles of systolic shared registers.

9 Acknowledgments

Much of the direction of this inquiry can be attributed to the author's advisor, Daniel Lopresti. John Savage and Steve Reiss have agreed to be on the author's thesis committee. Thanks are also due to Glenn Carroll and Gail Shaw for interest in writing algorithms for B-SYS, and to Marian Nodine and Elizabeth Shriver for work on improving the programming environment of the B-SYS simulator. Paul Howard provided many useful suggestions concerning the simulator. Scott Meyers gave this proposal a thorough reading. Finally, the entire 1988 VLSI class deserves great credit for putting up with the B-SYS simulator for one of its assignments.

10 References

[1] H. T. Kung and C. E. Leiserson, "Systolic arrays (for VLSI)," in *Sparse Matrix Proceedings* (I. S. Duff and G. W. Stewart, eds.), pp. 256–282, SIAM, 1978.

[2] H. T. Kung, "Why systolic architectures?," *Computer*, pp. 37–46, January 1982.

[3] M. J. Foster and H. T. Kung, "The design of special-purpose VLSI chips," *Computer*, pp. 26–40, January 1980.

[4] B. D. Ackland, "Dynamic time warp processor," in *Principles of CMOS VLSI Design* (N. H. E. Weste and K. Eshraghian, eds.), pp. 384–407, Reading, MA: Addison Wesley Publishing Co., 1985.

[5] D. P. Lopresti, "P-NAC: A systolic array for comparing nucleic acid sequences," in *Computer* [11], pp. 98–99.

[6] M. J. Carey, P. M. Hansen, and C. D. Thompson, "Sorting records in VLSI," in *Algorithmically Specialized Parallel Computers*, pp. 27–36, New York: Academic Press, 1985.

[7] A. Aliphas and J. A. Feldman, "The versatility of digital signal processing chips," *IEEE Trans. Computers*, vol. 24, pp. 40–45, June 1987.

[8] T. J. Fountain, "A survey of bit-serial array processor circuits," in *Computing Structures of Image Processing* (M. J. B. Duff, ed.), ch. 1, pp. 1–14, New York: Academic Press, 1983.

[9] S. Manohar and G. Baudet, "Supercomputing with VLSI," in *Proceedings of the 23rd Allerton Conference on Communication, Control, and Computing*, October 1986.

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

[11] "Special issue on systolic arrays," *Computer*, vol. 20, July 1987.

[12] S. Manohar, "Systolic architectures: A critical survey," Tech. Rep. CS-86-05, Brown University, Providence, RI, March 1986.

[13] D. P. Lopresti, *Discounts for Dynamic Programming with Applications in VLSI Processor Arrays.* PhD thesis, Princeton University, Princeton, N.J., 1987.

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

[15] K. E. Batcher, "Design of a massively parallel processor," *IEEE Trans. Computers*, pp. 836–840, September 1980.

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

[17] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing.* New York: McGraw-Hill, 1984.

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

[19] L. W. Tucker and G. G. Robertson, "Architecture and applications of the Connection Machine," *Computer*, vol. 21, pp. 26–38, August 1988.

[20] M. Annaratone *et al.*, "The Warp computer: Architecture, implementation and performance," *IEEE Trans. Computers*, vol. C-36, pp. 1523–1537, December 1987.

[21] T. Gross and M. S. Lam, "Compilation for a high-performance systolic array," in *Proceedings of the 1986 Symposium on Compiler Construction*, pp. 27–38, ACM SIGPLAN, 1986.

[22] J. Deutch *et al.*, "Performance of Warp on the DARPA architecture benchmarks," Tech. Rep. CMU-CS-87-148, Carnegie-Mellon University, Pittsburgh, PA, September 1987.

[23] NCR Corp., *Geometric Arithmetic Parallel Processor Data Sheet, part number NCR45CG72*, 1987.

[24] R. E. Morley and T. J. Sullivan, "A massively parallel systolic array processor system," in Bromley *et al.* [48], pp. 217–226.

[25] A. L. Fisher, H. T. Kung, *et al.*, "Design of the PSC: A programmable systolic chip," in *Third CALTECH Conference on VLSI* (R. Bryant, ed.), pp. 287–302, Rockville, MD: Computer Science Press, 1983.

[26] M. Homewood, D. May, D. Shepherd, and R. Shepherd, "The IMS T800 Transputer," *IEEE Micro*, pp. 10–26, October 1987.

[27] S. Y. Kung, "VLSI array processors," in Moore *et al.* [47], pp. 7–25.

[28] J. G. Nash and C. Petrozolin, "VLSI implementation of a linear systolic array," in *International Conference on Acoustics, Speech, and Signal Processing (ICASSP), proceedings*, pp. 1392–1395, IEEE, 1985.

[29] D. E. Foulser and R. Schreiber, "The Saxpy Matrix-1: A general purpose systolic computer," in *Computer* [11], pp. 35–43.

[30] M. Kunde *et al.*, "The instruction systolic array and its relation to other models of parallel computers," in *Parallel Computing '85* (M. Feilmeier, G. Joubert, and U. Schendel, eds.), pp. 491–497, Amsterdam: North-Holland, 1986.

[31] H.-W. Lang, "The instruction systolic array — a parallel architecture for VLSI," *Integration, the VLSI Journal*, vol. 4, no. 1, pp. 65–74, 1986.

[32] R. Hughey and D. P. Lopresti, "Architecure of a programmable systolic array," in Bromley *et al.* [48], pp. 41–50.

[33] H. T. Kung, "Deadlock avoidance for systolic communication," *Journal of Complexity*, vol. 4, pp. 87–105, 1988.

[34] H. T. Kung, "Systolic communication," in Bromley *et al.* [48], pp. 695–703.

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

[36] M. Annaratone, *Digital CMOS Circuit Design.* Boston: Kluwer Academic Publishers, 1986.

[37] P. R. Capello and K. Steiglitz, "Unifying VLSI array design with linear transformations of space-time," in *Advances in Computing Research*, vol. 2, pp. 23–65, JAI Press Inc., 1984.

[38] B. R. Engstrom and P. R. Cappello, "The SDEF systolic programming environment," tech. rep., University of California, Santa Barbara, CA, August 1987.

[39] P. Gachet, B. Joinnault, and P. Quinton, "Synthesizing systolic arrays using Diastol," in Moore *et al.* [47], pp. 25–37.

[40] D. Moldovan, "On the analysis and synthesis of VLSI algorithms," *IEEE Trans. Computers*, vol. c-31, pp. 1121–1126, November 1982.

[41] D. I. Moldovan and J. A. B. Fortes, "Partitioning and mapping algorithms into fixed size systolic arrays," *IEEE Trans. Computers*, vol. c-35, pp. 1–12, January 1986.

[42] R. Hughey and D. P. Lopresti, "Location of protein encodings in nucleic acids," Tech. Rep. CS-87-09, Brown University, Providence, RI, May 1987.

[43] P. Lee and Z. Kedem, "Systhesizing linear array algorithms from nested FOR loop algorithms," *IEEE Trans. Computers*, vol. 37, pp. 1578–1598, December 1988.

[44] A. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley Publishing Co., 1974.

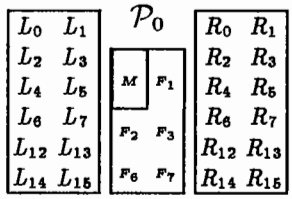
[45] R. Lipton and D. Lopresti, "Comparing long strings on a short systolic array," in Moore *et al.* [47], pp. 363–376.

[46] M. H. Nodine and E. A. Shriver, "A programming environment for the Brown Systolic Array." CS236 Final Report, December 1988.

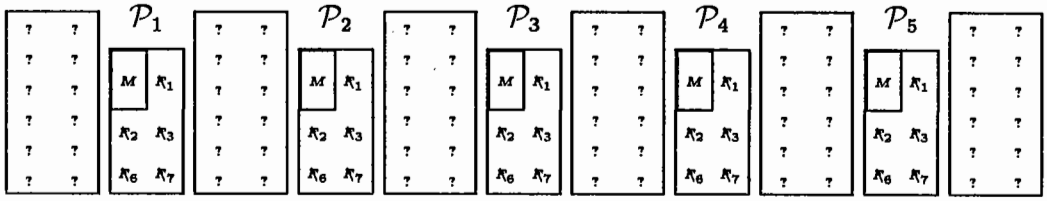
[47] W. Moore, A. McCabe, and R. Urquhart, eds., *Systolic Arrays*. Boston: Adam Hilger, 1987.

[48] K. Bromley, S. Y. Kung, and E. Swartzlander, eds., *Proceedings of the First International Conference on Systolic Arrays*, IEEE Computer Society, May 1988.

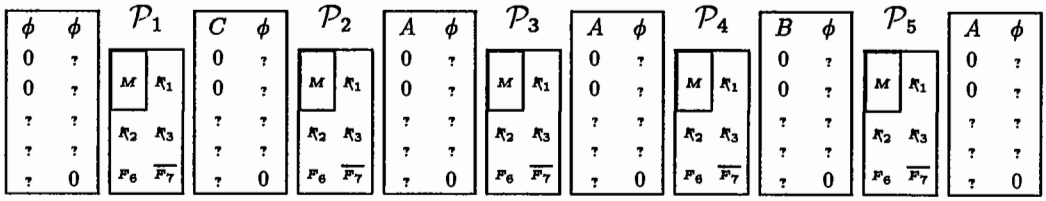
A Sequence comparison on B-SYS



Time: 1 $f_r = ff$: $L_0, L_0, \rightarrow L_0$ $f_z = ff : M \rightarrow M$

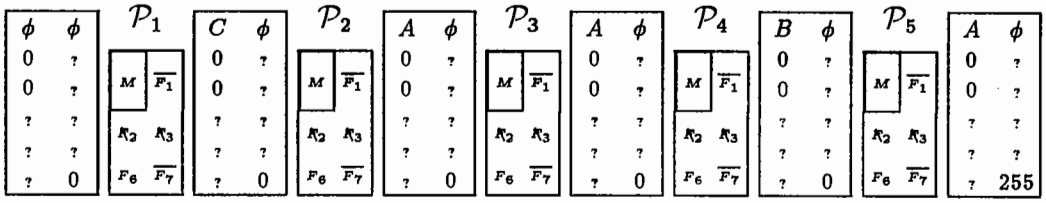


Time: 12 $f_r = aa$: $R_0, R_0, \rightarrow L_0$ $f_z = f0 : M \rightarrow M$ input: 1



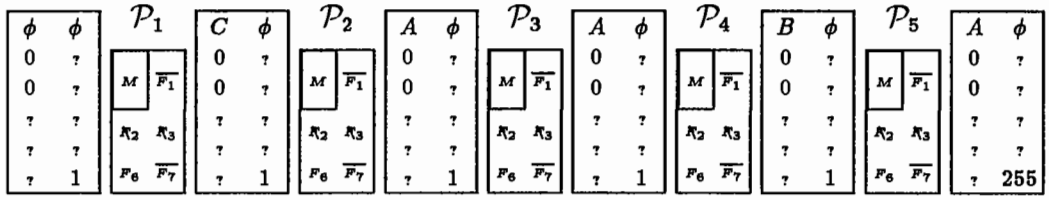
Initialization complete: L_0 has t_i, the fixed string.

Time: 14 $f_r = ac$: $L_2, R_2, \rightarrow L_F$ $f_z = f0 : F_1 \rightarrow F_1$ output: 0 input: 255



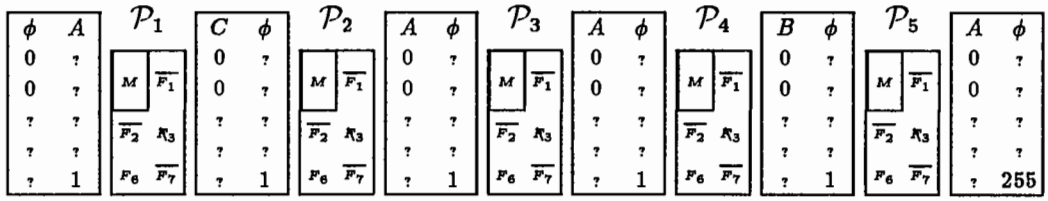
L_F now has $\min(d_{i-1,j}, d_{i,j-1})$. Output is $d_{n,j-1}$.

Time: 15 $f_r = 5a$: $L_F, L_F, \rightarrow L_F$ $f_z = c0 : F_6 \rightarrow F_1$



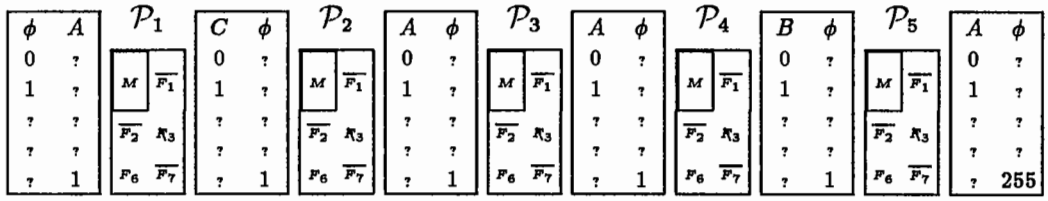
L_F incremented.

Time: 16 $f_r = aa$: $L_1, L_0, \rightarrow R_1$ $f_z = f8 : F_7 \rightarrow F_2$ input: 1



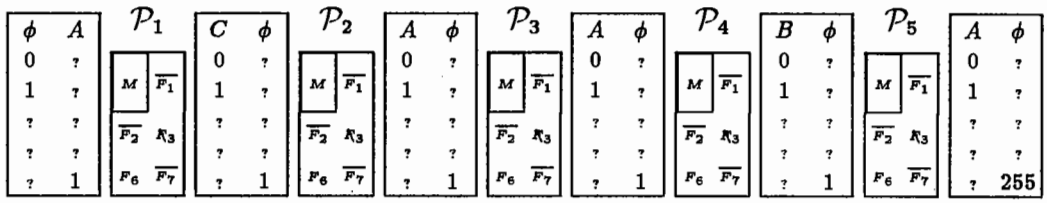
Test match of $s_j(L_1)$ and $t_i(L_0)$ and shift s_j to right.

Time: 17 $f_r = ac$: $L_4, L_F, \rightarrow R_4$ $f_z = f0 : F_2 \rightarrow F_2$ input: 1



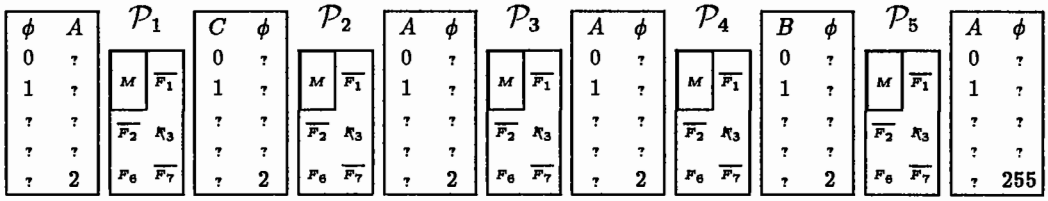
Select between $\min(d_{i-1,j}, d_{i,j-1}(L_F)$ and $d_{i-1,j-1}(L_4)$ depending on match, storing result in R_4 which is now $d_{i,j}$.

Time: 19 $f_r = ac$: $L_4, R_4, \rightarrow L_F$ $f_z = f0 : F_1 \rightarrow F_1$ output: 1 input: 255



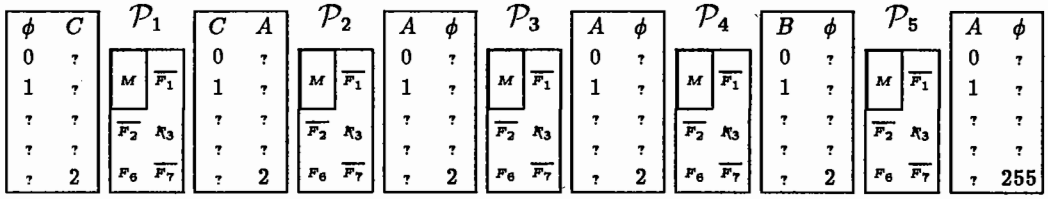
L_F now has $\min(d_{i-1,j}, d_{i,j-1})$. Output is $d_{n,j-1}$.

Time: 20 $f_r = 5a$: $L_F, L_F, \rightarrow L_F$ $f_z = c0$: $F_6 \rightarrow F_2$



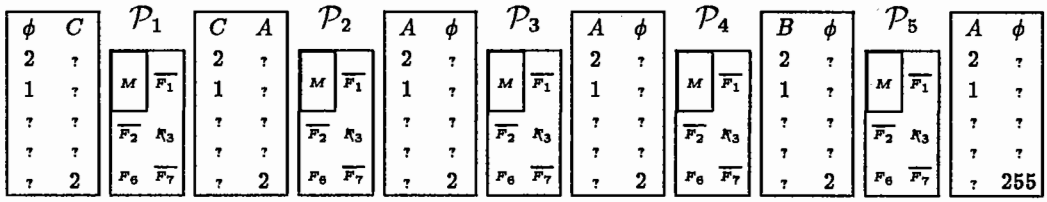
L_F incremented.

Time: 21 $f_r = aa$: $L_1, L_0, \rightarrow R_1$ $f_z = f8$: $F_7 \rightarrow F_2$ input: 4



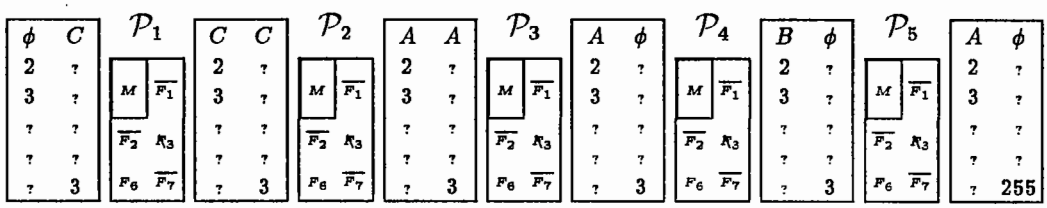
Test match of $s_j(L_1)$ and $t_i(R_0)$ and shift s_j to right.

Time: 24 $f_r = ac$: $L_2, R_2, \rightarrow L_F$ $f_z = f0$: $F_1 \rightarrow F_1$ output: 2 input: 255



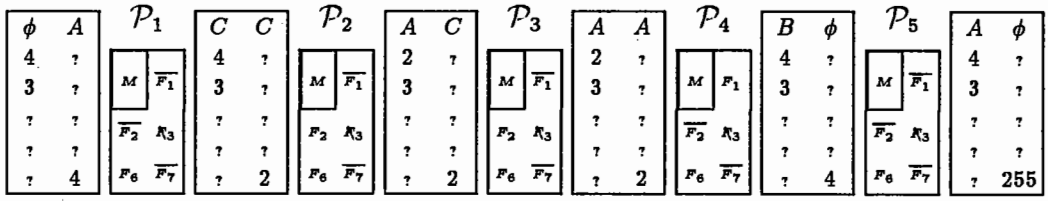
L_F now has $\min(d_{i-1,j}, d_{i,j-1})$. Output is $d_{n,j-1}$.

Time: 29 $f_r = ac$: $L_4, R_4, \rightarrow L_F$ $f_z = f0$: $F_1 \rightarrow F_1$ output: 3 input: 255



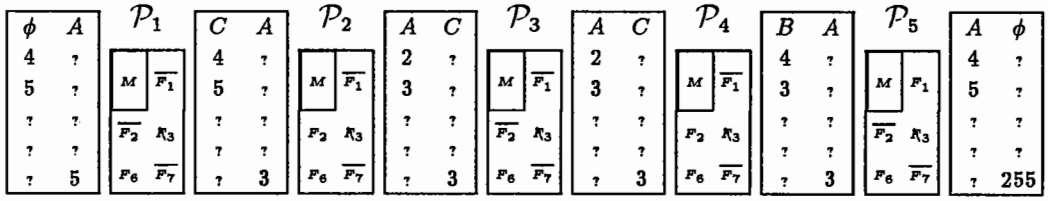
L_F now has $\min(d_{i-1,j}, d_{i,j-1})$. Output is $d_{n,j-1}$.

Time: 34 $f_r = ac$: $L_2, R_2, \rightarrow L_F$ $f_z = f0 : F_1 \rightarrow F_1$ output: 4 input: 255



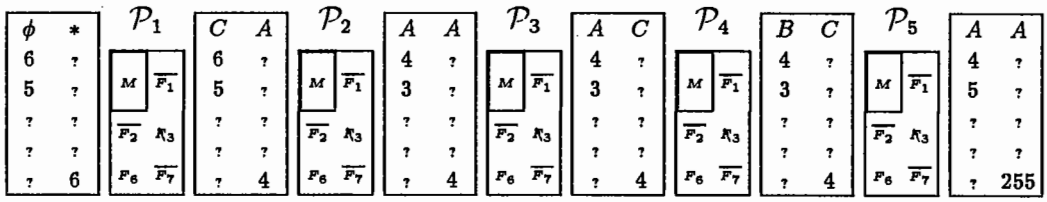
L_F now has $\min(d_{i-1,j}, d_{i,j-1})$. Output is $d_{n,j-1}$.

Time: 39 $f_r = ac$: $L_4, R_4, \rightarrow L_F$ $f_z = f0 : F_1 \rightarrow F_1$ output: 5 input: 255



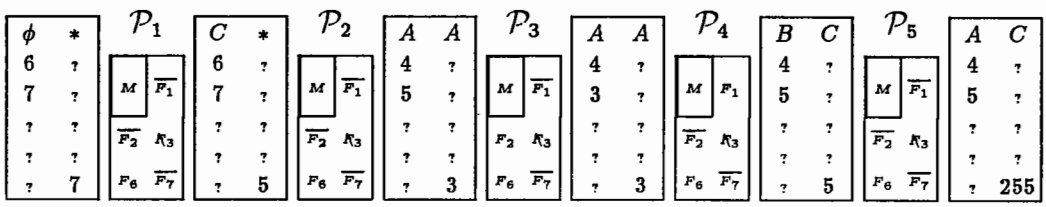
L_F now has $\min(d_{i-1,j}, d_{i,j-1})$. Output is $d_{n,j-1}$.

Time: 44 $f_r = ac$: $L_2, R_2, \rightarrow L_F$ $f_z = f0 : F_1 \rightarrow F_1$ output: 4 input: 255



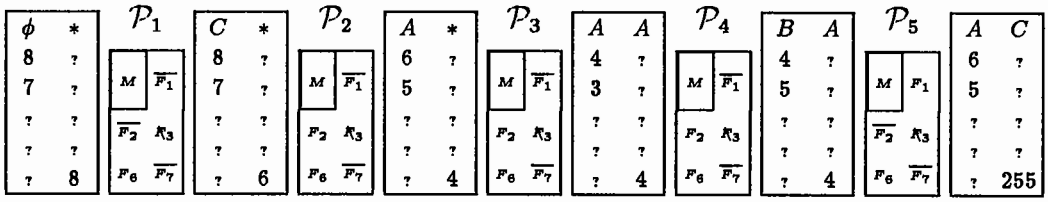
L_F now has $\min(d_{i-1,j}, d_{i,j-1})$. Output is $d_{n,j-1}$.

Time: 49 $f_r = ac$: $L_4, R_4, \rightarrow L_F$ $f_z = f0 : F_1 \rightarrow F_1$ output: 5 input: 255



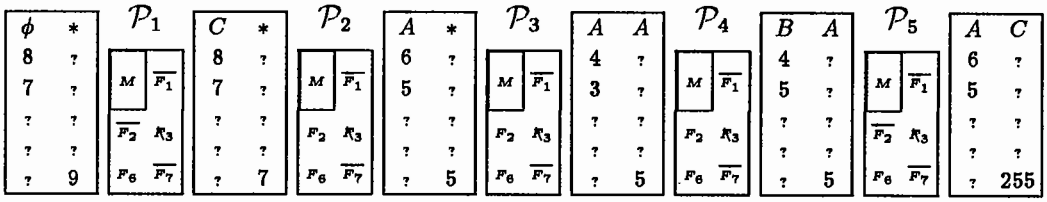
L_F now has $\min(d_{i-1,j}, d_{i,j-1})$. Output is $d_{n,j-1}$.

Time: 54 $f_r = ac$: $L_2, R_2, \rightarrow L_F$ $f_z = f0 : F_1 \rightarrow F_1$ output: 6 input: 255



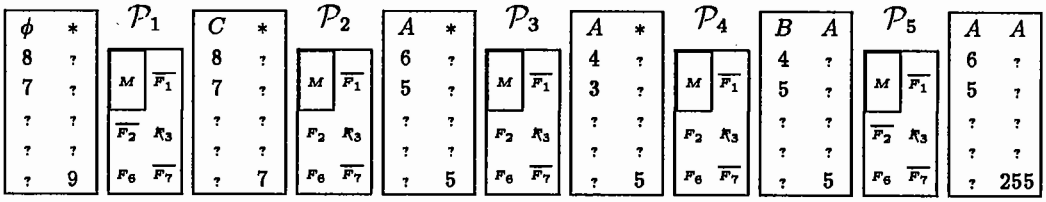
L_F now has $\min(d_{i-1,j}, d_{i,j-1})$. Output is $d_{n,j-1}$.

Time: 55 $f_r = 5a$: $L_F, L_F, \rightarrow L_F$ $f_z = c0 : F_6 \rightarrow F_1$



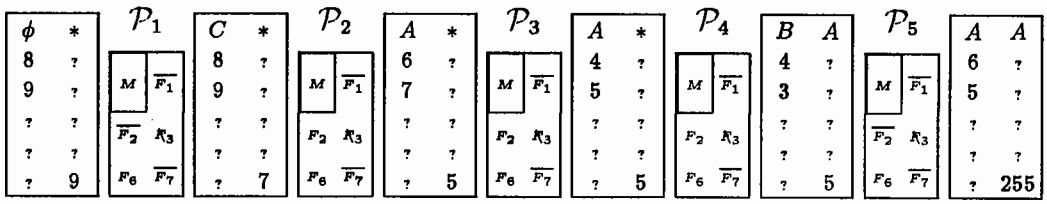
L_F incremented.

Time: 56 $f_r = aa$: $L_1, L_0, \rightarrow R_1$ $f_z = f8 : F_7 \rightarrow F_2$ input: 15



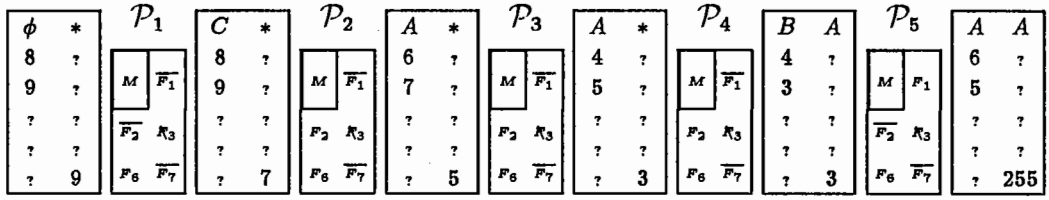
Test match of $s_j(L_1)$ and $t_i(L_0)$ and shift s_j to right.

Time: 57 $f_r = ac$: $L_4, L_F, \rightarrow R_4$ $f_z = f0 : F_2 \rightarrow F_2$ input: 9



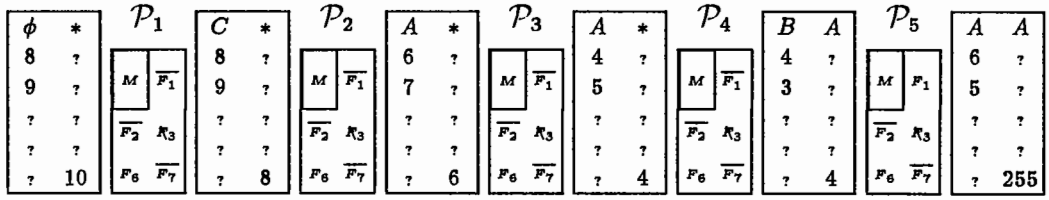
Select between $\min(d_{i-1,j}, d_{i,j-1}(L_F)$ and $d_{i-1,j-1}(L_4)$ depending on match, storing result in R_4 which is now $d_{i,j}$.

Time: 59 $f_r = ac$: $L_4, R_4, \rightarrow L_F$ $f_z = f0 : F_1 \rightarrow F_1$ output: 5 input: 255



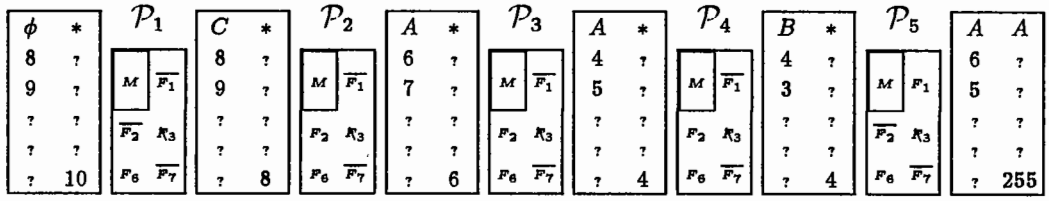
L_F now has $\min(d_{i-1,j}, d_{i,j-1})$. Output is $d_{n,j-1}$.

Time: 60 $f_r = 5a$: $L_F, L_F, \rightarrow L_F$ $f_z = c0 : F_6 \rightarrow F_2$



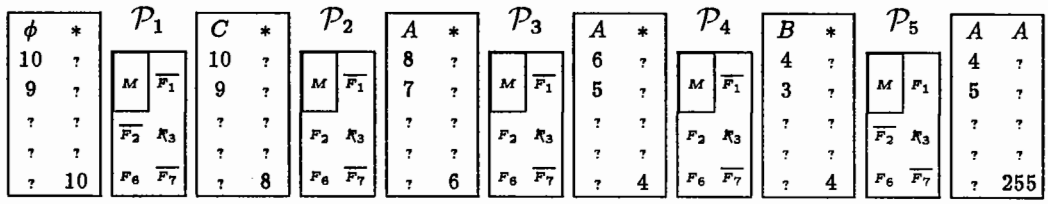
L_F incremented.

Time: 61 $f_r = aa$: $L_1, L_0, \rightarrow R_1$ $f_z = f8 : F_7 \rightarrow F_2$ input: 15



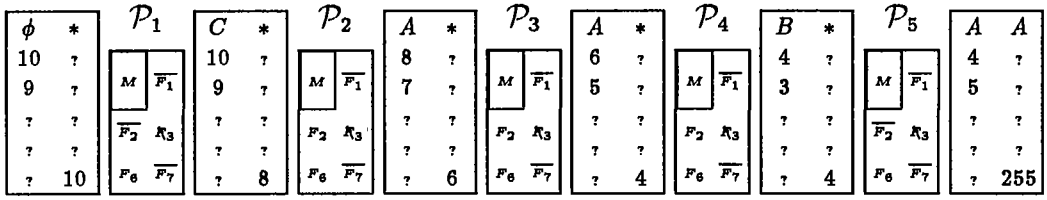
Test match of $s_j(L_1)$ and $t_i(R_0)$ and shift s_j to right.

Time: 62 $f_r = ac$: $L_2, L_F, \rightarrow R_2$ $f_z = f0 : F_2 \rightarrow F_2$ input: 10



Select between $\min(d_{i-1,j}, d_{i,j-1}(L_F)$ and $d_{i-1,j-1}(L_2)$ depending on match, storing result in R_2 which is now $d_{i,j}$.

Time: 64 $f_r = ac$: $L_2, R_2, \rightarrow L_F$ $f_z = f0 : F_1 \rightarrow F_1$ output: 4 input: 255



L_F now has $\min(d_{i-1,j}, d_{i,j-1})$. Output is $d_{n,j-1}$.