Advanced Research in VLSI

Proceedings of the 1991 University of California Santa Cruz Conference

VLSI

edited by Carlo H. Séquin

Rapid Implementation of a Genetic Sequence Comparator Using Field-Programmable Logic Arrays

Daniel P. Lopresti
Department of Computer Science
Brown University
Providence, RI 02912, USA

Abstract

This paper describes the implementation of a parallel algorithm for sequence comparison on the SPLASH programmable logic array. The algorithm, originally developed for a custom VLSI chip, has applications in molecular genetics and runs faster on SPLASH than it does on supercomputers. I discuss details of the problem and its systolic solution, the SPLASH architecture and design environment, and the implementations currently running on SPLASH.

1 History

The research described in this paper grew out of a confluence of efforts in academia, a government research lab, and a commercial chip manufacturer. The end result is a system, called SPLASH, that fits in a Sun workstation and for $13,000 provides supercomputer performance for some important problems of interest. One of these, genetic sequence comparison, was an inspiration for the architecture and is now one of the more notable applications running on the hardware.

In the taxonomy of parallel architectures, SPLASH occupies a new niche. As with single-purpose systolic arrays, the functionality of the chips comprising SPLASH is specified by the user at a very low level. In essence, a hardware description of the desired algorithm must be created. Unlike single-purpose arrays, SPLASH can be reconfigured in less than a second to run a new algorithm. This flexibility is made possible through the use of a field-programmable gate array (FPGA) as the primary compute engine. SPLASH also differs from existing programmable systolic arrays in that its stages do not employ an instruction set architecture. Rather, each contains several hundred configurable logic blocks (CLB's) which are defined at the gate level to realize a range of boolean functions. As Gray and Kean have observed, this yields a new computational paradigm midway between the traditional hardware and software levels [3].

1.1 The Sequence Comparison Problem

Deoxyribonucleic acid (DNA) is a macromolecular chain consisting of from thousands to millions of nucleotides, each identified by which one of four possible bases it contains: Adenine, Cytosine, Guanine, or Thymine. Fundamental genetic information is encoded in the linear ordering of these nucleotides. Similarity (or *homology*) between two sequences can be interesting both because it suggests an evolutionary relationship, and

because structure is a predictor of function. When attempting to characterize an unfamiliar sequence, a biologist will often compare it to collections of known DNA in the hopes of finding close matches. Such databases are large (GenBank currently contains 30 million bases) and growing rapidly (like VLSI densities, they double every year or so). The onset of the 10-year effort to sequence the entire Human Genome [12], coupled with the recent development of automated sequencing equipment, will only accelerate this growth. Hence there is a pressing need for algorithms and architectures for fast sequence comparison.

Biologists employ a number of criteria for judging the similarity of two sequences. One with considerable intuitive appeal is *evolutionary distance*, which is based on the premise that differences arise as a result of three elemental phenomena: the deletion of a single nucleotide, the insertion of a single nucleotide, and the substitution of one nucleotide for another. The proximity of two sequences S and T can be quantified by assessing a charge for each of these steps and then finding the least expensive transformation of S into T.

The calculation of evolutionary distance can be formulated as a well-known dynamic programming recurrence. If $S = s_1 s_2 ... s_m$, $T = t_1 t_2 ... t_n$, and $d_{i,j}$ is the distance between the subsequences $s_1 s_2 ... s_i$ and $t_1 t_2 ... t_j$, then

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

and

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

Here $c_{del}(s_i)$ is the cost of deleting s_i, $c_{ins}(t_j)$ is the cost of inserting t_j, and $c_{sub}(s_i,t_j)$ is the cost of substituting t_j for s_i. Figure 1, for example, shows the distance table that results when the sequences $TGCTAAGC$ and $AGACTAGG$ are compared with a deletion/insertion cost of 1 and a substitution cost of 2. The final distance, in this case 6, is the value found in the lower right-hand corner. Sankoff and Kruskal provide a detailed survey of this and other sequence problems in [10].

In late 1984 Dick Lipton and I, working at Princeton with biologist Doug Welsh, noted that the dynamic programming algorithm could be "systolicized." Mapping the recurrence onto a linear systolic array is a straightforward procedure, although there are several possible data flows that can yield alternate architectures. In the one I chose, the source and target sequences form two data streams moving in opposite directions as depicted in Figure 2. Associated with the characters are initializing values from the first row and column of the distance table. As these pass through the array they are transformed into the last row and column, and hence the answer. When two characters

enter a processor they are compared, and the outcome is used along with the two traveling distances to update the processor's stored distance as specified in Equation 1.

		A	G	A	C	T	A	G	G
	0	1	2	3	4	5	6	7	8
T	1	2	3	4	5	4	5	6	7
G	2	3	2	3	4	5	6	5	6
C	3	4	3	4	3	4	5	6	7
T	4	5	4	5	4	3	4	5	6
A	5	4	5	4	5	4	3	4	5
A	6	5	6	5	6	5	4	5	6
G	7	6	5	6	7	6	5	4	5
C	8	7	6	7	6	7	6	5	6

Figure 1: An evolutionary distance table

As I set out to build a custom VLSI chip to implement this architcture, I discovered that it is not necessary to maintain the full evolutionary distances inside the array. In fact, the entire computation can be performed using just remainders mod 4 (i.e., the low-order two bits). The true distance is reconstructed outside the array, as the intermediate results shift by. This "trick," more fully described in [5], hinges on the fact that adjacent values in the distance table are close in magnitude. For us, this was key since it made it possible to fit 30 processors on a single chip (an impressive number for 1985).

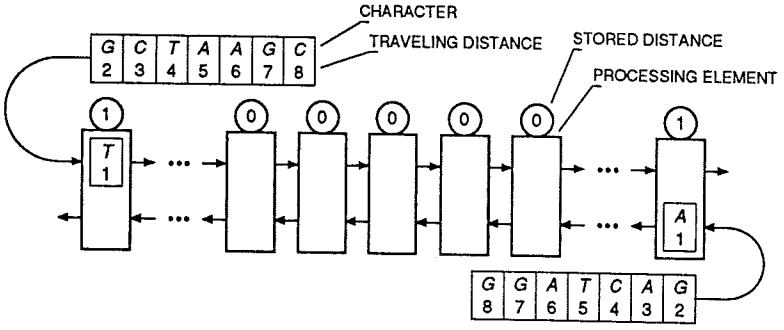

Figure 2: Sequence comparison on a linear systolic array

Later to be known as the Princeton Nucleic Acid Comparator (P-NAC), the 4μ-nMOS chip was designed using the Berkeley CAD tool suite (`caesar`, `crystal`, and `esim`) [8], and `allende`, a procedural layout language developed at Princeton [7]. I made heavy use of programmable logic arrays (PLA's) which, although somewhat

wasteful of area, speed, and power, allowed me to complete the design and ship it to the MOS Implementation Service (MOSIS) for fabrication in two months time. A paper describing the work was prepared and presented at the 1985 Chapel Hill Conference [4].

It took us another two months after we received the packaged chips to build and debug a Multibus board and write the driver software for our Sun 2 host. Because we looked at the board design as more of a chore than research, we choose a "quick and dirty" approach that was simple to implement but could only drive the custom parts at one-tenth their maximum speed. Even so, the resulting system with 10 P-NAC chips (a total of 300 processors) benchmarked several hundred times faster than minicomputers of the day.

Unfortunately, P-NAC never progressed to the point where biologists could actually use it. The wire-wrapped board was not particularly reliable, and had been developed for a Sun, whereas biologists were using almost exclusively IBM-compatible PC's. In addition, there was concern that a single, hard-wired algorithm was too inflexible when compared to software approaches. For instance, some biologists quibbled with my choice of cost assignments. I did not consider the latter an insurmountable obstacle since I always viewed P-NAC as a filter. As such, it would not be necessary to run precisely the same algorithm(s) as are run on a sequential machine, just a good enough approximation that most uninteresting candidates are eliminated and all close matches are flagged for further analysis. There were, however, other problems with the architecture. Important variations on the same basic theme (e.g., sequence-subsequence and subsequence-subsequence matching) could be solved systolically, but not using P-NAC. Another proposed sequence comparison array suffers from similar limitations [9].

1.2 The SPLASH Architecture

About the time I was finishing my dissertation at Princeton, Paul Schneck, director of the then newly-established Supercomputing Research Center (SRC), became aware of our work. He was also intrigued by a new technology introduced by a Silicon Valley start-up named Xilinx: the field-programmable gate array. The merging of these two ideas, one a systolic algorithm and the other a hardware building block, lead to the development of a radically new machine, the SPLASH programmable logic array. The design of a prototype was begun in September of 1988. In June of 1989 SPLASH was released internally for use during the annual SRC Summer Workshop. Five copies of the hardware, a programming language, and a symbolic debugger were operational at that time. Currently there are 16 SPLASH arrays in use at SRC and elsewhere.

The SPLASH hardware consists of two VME boards (triple-high Eurocard format) that plug into Sun workstation slots sharing a VSB backplane connection. One is an "off-the-shelf" 8-megabyte staging memory with ports on both the VME and VSB buses. The other board is the SPLASH engine, containing the logic array, a VME interface to the Sun, and a VSB interface to the staging memory. SPLASH has been run in Sun 3/160's, 3/280's, and 4/280's.

The logic array consists of 32 Xilinx 3090 field-programmable gate array chips connected linearly, as shown in Figure 3. Interspersed between each pair of adjacent 3090's is a 128K × 8 50ns static RAM chip. This memory is accessible to the FPGA on either side and can be used as a scratchpad, or for preloading initial data (e.g., a look-up table). Successive array stages are joined by 68-bit data paths. The first stage (X0) is connected to the input FIFO through 32-bit data and 4-bit control paths. The last (X31) is connected to the output FIFO similarly. Stages 0 and 31 are joined in wrap-around fashion by a 35-bit data path. All connections (except those to the FIFO's) are bidirectional, so data can flow through the array in either direction. The array board is attached to the VME bus for control and the VSB bus for data I/O.

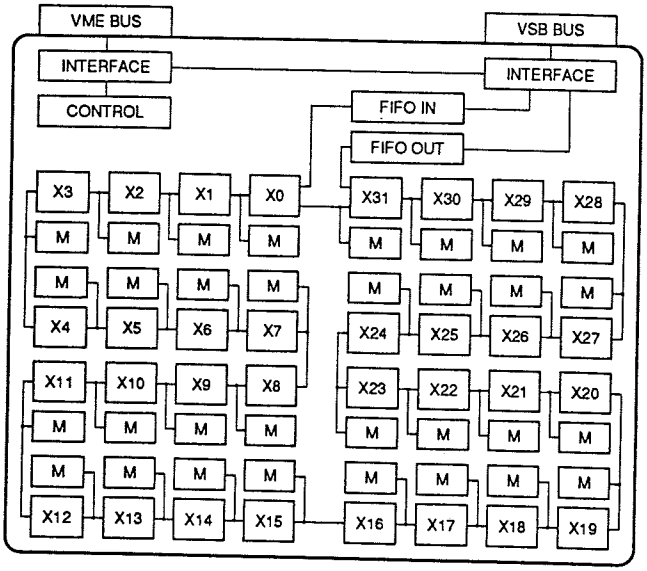

Figure 3: SPLASH

Xilinx 3090 FPGA's are the "heart" of the SPLASH systolic engine [13]. While they currently must be programmed at a very low level, they offer a great deal of flexibility. Each contains 320 configurable logic blocks in a 20 × 16 grid, and 144 input/output blocks (IOB's) around the perimeter of the chip. A CLB contains a combinatorial logic section, two D flip-flops, and an internal control section. The logic section can be configured to perform any boolean function of five inputs, or any two functions of four inputs. The latter option is illustrated in Figure 4.

The SPLASH clock is set by the user up to a maximum rate of 32 MHz. Measured skew across the board is at most 10ns. Most existing designs seem to run at speeds between 1 and 8 MHz. Like P-NAC, SPLASH is I/O limited; data can be transferred at

50 Mbits/second with the current memory card, so the fastest a design that uses the FIFO's will run is 1 MHz.

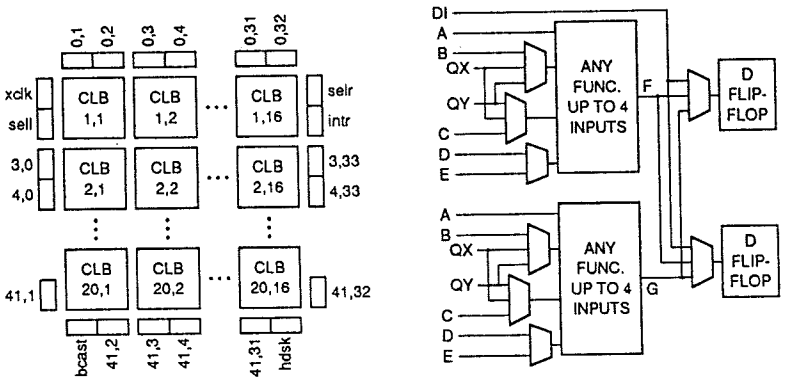

Figure 4: The Xilinx 3090 FPGA – overview (left), CLB internals (right)

1.3 The SPLASH Design Environment

It is important to realize that Xilinx markets their FPGA's as a replacement for random "jelly-bean" logic; they probably never imagined someone would want to use them to build a highly parallel computer. Their design tools reflect this orientation, of course, so much original software had to be developed.

SPLASH is programmed by specifying CLB/IOB configurations at the register-transfer level for each of the 32 chips in the array, and then writing corresponding supervisory code to run on the Sun host. Presently there are three alternate routes to defining the logic: XACT (the Xilinx graphical editor), standard schematic capture packages that support FPGA's, and the Logic Description Generator (LDG) language created at SRC. This last approach is the one most often used for SPLASH.

LDG is a Common Lisp program for manipulating *templates* (function units with designated inputs and outputs) describing the kinds of logic functions realizable on a Xilinx chip. LDG is designed specifically to take advantage of the simplicity and regularity inherent in many systolic algorithms. It allows users to decompose their designs into hierarchical modules, specify the modules in a natural way, and instantiate the modules (and variations) at selected locations on the FPGA chip. In particular, LDG makes it possible to:

- configure CLB's and IOB's,
- define templates composed of CLB's, IOB's, and previously-defined templates,
- create multiple copies of CLB's, IOB's, and templates, and specify their interconnection,
- fix the placement of CLB's, IOB's, and templates on the chip,

- build and use libraries of CLB, IOB, and template *schemas*.

Templates consist of *parts*, which may be either Xilinx primitives (AND, OR, XOR, etc.) or invocations of other CLB, IOB, or template definitions. A template's inputs and outputs are *signals*, which can be single bits or arrays of bits. Within a template, local signals can be defined; LDG will ensure that these receive unique names each time the template is invoked. Figure 5 gives the LDG description of the circuit shown in Figure 9 and discussed later.

```
(clb EvnSttClb1
  (input A C D E ~K)  (output Y)
  (location row col)  (config F)
  (F ((~D * ~E * QY) + (C * E) + (A * D * ~E)))
  (DY F)  (Y QY))

(clb EvnSttClb2
  (input C D E ~K)  (output Y)
  (location row col)  (config F)
  (F ((E * QY) + (C * ~E) + D))
  (DY F)  (Y QY))

(clb EvnSttClb3
  (input A B C D E)  (output X)
  (location row col)  (config F)
  (F ((A @ B) * ~(A @ C) * ~D * ~E))
  (X F))

(clb EvnSttClb4
  (input A B C D E)  (output Y)
  (location row col)  (config F)
  (F (((A @ B) * C * ~D * ~E) + (B * D * ~E) + (A * E)))
  (Y F))

(template EvnStt
  (input (array SrcDstIn 1 0) (array TgtDstIn 1 0) SrcNull Match TgtNull clk)
  (output (array DstOut 1 0)  (location row col)

  (part-list
    ((name EvnStt1)  (part EvnSttClb1)
    (input (index TgtDstIn 0) (index SrcDstIn 0) SrcNull TgtNull clk)
    (output (index DstOut 0))  (location row col))

    ((name EvnStt2)  (part EvnSttClb2)
    (input EvnSttT1 EvnSttT2 Match clk)
    (output (index DstOut 1))  (location row (+ 1 col)))

    ((name EvnStt3)  (part EvnSttClb3)
    (input (array SrcDstIn 1 0) (index TgtDstIn 1) SrcNull TgtNull)
    (output EvnSttT1)  (location (+ 1 row) col))

    ((name EvnStt4)  (part EvnSttClb4)
    (input (index SrcDstIn 1) (index TgtDstIn 1) (index DstOut 1) SrcNull TgtNull)
    (output EvnSttT2)  (location (+ 1 row) (+ 1 col)))))
```

Figure 5: An example of LDG code

In addition, users can create libraries of parameterized templates. A SPLASH designer might, for example, define a schema for a generalized shift register and then generate a register of a specific length by invoking the schema with the appropriate parameters. The final output of LDG is a low-level description in standard Xilinx Netlist Format (XNF). This must be passed successively through a filter (`xnf2lca`), the

Xilinx router (`apr`), and another filter (`makebits`) before the design is ready to download to SPLASH.

An interpretive language called Trigger serves as the primary SPLASH run-time environment. Trigger allows users to debug designs interactively and symbolically, just as might be done with a C program under dbx. SPLASH is treated as a collection of 32 stages, each with its own program and symbol table. Designs can be stepped an arbitrary number of clock ticks, after which the complete state of the array is read back and the values of any variables of interest are displayed. (Due to limitations in the current Xilinx hardware, Trigger cannot be used to initialize on-chip variables.)

Trigger's commands are similar to those of the C-shell. It has *if* and *while* statements, variables containing strings or numeric values, *for* loops, and other features found in most command language interpreters. SPLASH variables may be user-defined during a single session, or programmer-defined across all executions of a particular design. Trigger can modify its own control flow based on the values of these variables.

Although the textual paradigm adopted by Trigger is quite powerful, some SPLASH programmers (myself included) are more comfortable with the graphical view provided by the SRC Chipdraw package. This program reads in a bitmap representing the states of all flip-flops on a given Xilinx chip (dumped previously from within Trigger) and animates the computation cycle-by-cycle, permitting the user to single-step or run to completion. While this hands-on approach has its limits, it is intuitive and more than sufficient for debugging simple systolic algorithms like sequence comparison.

When a design has been thoroughly tested and is ready to move to production, there is available a C library allowing direct access to SPLASH via subroutine calls.

A complete iteration of the SPLASH design loop is shown in Figure 6. I have also indicated the amount of time one would spend waiting for each tool, assuming an average sized design. For the most part the procedure is fast and relatively painless, the notable exception being the `apr` routing utility which can require on the order of half an hour (on a lightly loaded Sun 3/280). To make matters worse, `apr` is almost guaranteed to fail on the first try for anything other than a sparse design. It is not unusual to have to spend several days repacking logic, reordering pins, and shuffling CLB's to try to get the routing to complete. This is by far the most time consuming part of the current design process.

More detailed descriptions of the SPLASH hardware and design environment can be found in [1], [2], and [11].

2 Sequence Comparison on SPLASH

I began work on implementing the logic I used for P-NAC on SPLASH (with a few minor changes) during the 1989 SRC Summer Workshop and completed it the following summer. Altogether, I estimate the design took two months from start to finish (including the time needed to learn the architecture and CAD tools). At this point we have four distinct but related versions up and running: short and long DNA (248 and 746

processors, respectively), and short and long ASCII (256 and 498 processors, respectively). The basic data flow in all cases is identical, using the end-around connections between chips X0 and X31 to mimic the bidirectional P-NAC. Source and target sequences are packed together, one character from each per 32-bit word, and loaded into the input FIFO. The final evolutionary distance is calculated in an up/down counter on the last chip in the array (X31) and returned to the Sun host through the output FIFO.

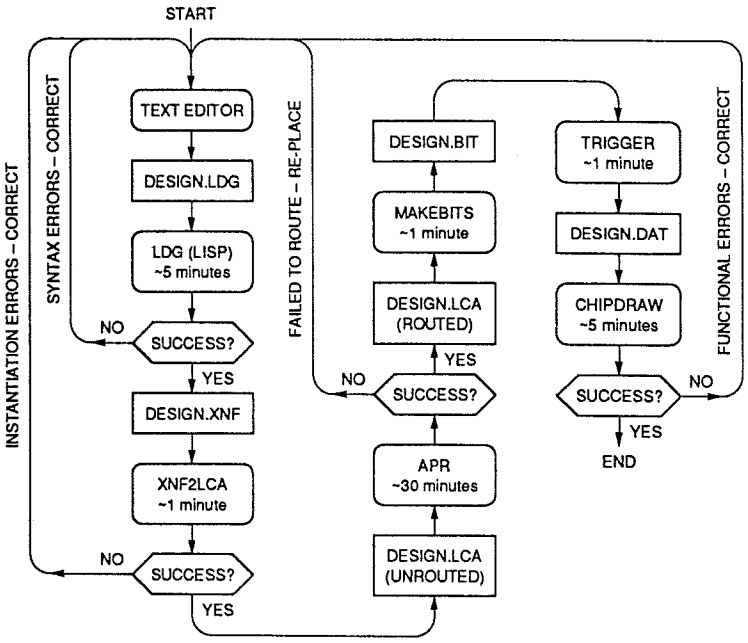

Figure 6: One iteration of the SPLASH design loop

Figure 7 shows a block diagram of a processing element. The source and target characters (four bits for DNA, seven bits for ASCII) are compared and latched on the rising edge of the clock. The finite state machine computes a new distance based on the result of the comparison and the previous source and target distances on the falling edge of the clock, using the mod 4 version of Equation 1. This basic cell is encapsulated as an LDG template and replicated in the obvious way to build longer and longer arrays (with adjacent PE's clocked on alternating edges).

My SPLASH design for the DNA character comparator appears in Figure 8. The bits of the source character are latched in the left column of CLB's, the bits of the target character in the right. The comparator passes on three local signals to the finite state machine: SrcNull and TgtNull indicate that the source and target characters are nulls (used as filler), respectively, and Match indicates that the two characters match under some

criterion. (In the DNA case the nucleotides are given the encodings $A = 0001$, $C = 0010$, $G = 0100$, $T = 1000$. In addition, we have wildcards with the encodings $R = A$ or $G = 0101$, $Y = C$ or $T = 1010$, and $N = \text{any} = 1111$).

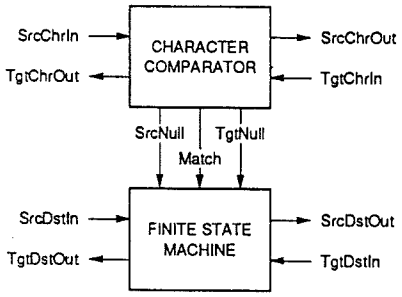

Figure 7: Block diagram of a processing element

The finite state machine logic is shown in Figure 9. The two distance bits are latched in the top two CLB's, the low-order bit on the left and the high-order on the right. Unlike the character comparator which has a fairly obvious partitioning and mapping, this particular arrangement was achieved only after a good deal of experimentation (numerous iterations of the loop in Figure 6, usually ending with `apr` failing to finish the routing). An underlying concern throughout the process was the Xilinx 3090's 20×16 topology.

In the various implementations, a SPLASH chip contains between four and 24 processing elements and the entire array totals from 248 to 746 PE's. The current designs have been tested at up to 4 MHz, but as remarked earlier the FIFO's can only sustain a 1 MHz rate. At this point we are not making effective use of the available I/O bandwidth since our 32-bit input words are not packed full; in anticipation of this future enhancement I left the first and last chips in the array mostly empty.

One particularly annoying problem with the Xilinx chips is that taking a module that routes at a certain location and sliding it up or down a row can sometimes cause it not to route. Evidently the real hardware is not as regular as the high-level descriptions would have one believe. Also, a judicious (and occasionally counter-intuitive) choice of CLB pins can make the difference between a design routing or not, even though the internal function units usually do not care what pin a signal comes in on. Dealing with these issues requires a fair amount of expertise, thus contributing to SPLASH's steep learning curve. (Software for some of these problems does exist, but its quality is unacceptable.)

3 Evaluation – Performance and Other Measures

Because the FPGA is still in its infancy, it is important to strive for high utilization of on-chip resources so that performance figures will compare favorably to older, proven technologies. Fortunately, the kinds of systolic pattern matching applications that were

an original motivation for SPLASH have in fact turned out to map well onto the actual hardware. Figure 10 presents CLB usage statistics for the sequence comparison arrays (the numbers for the long arrays are most telling, since the short arrays are intentionally paired down). It seems unlikely that in the near future a fully-automated approach could achieve the close to 100% utilization I obtained in the long ASCII case.

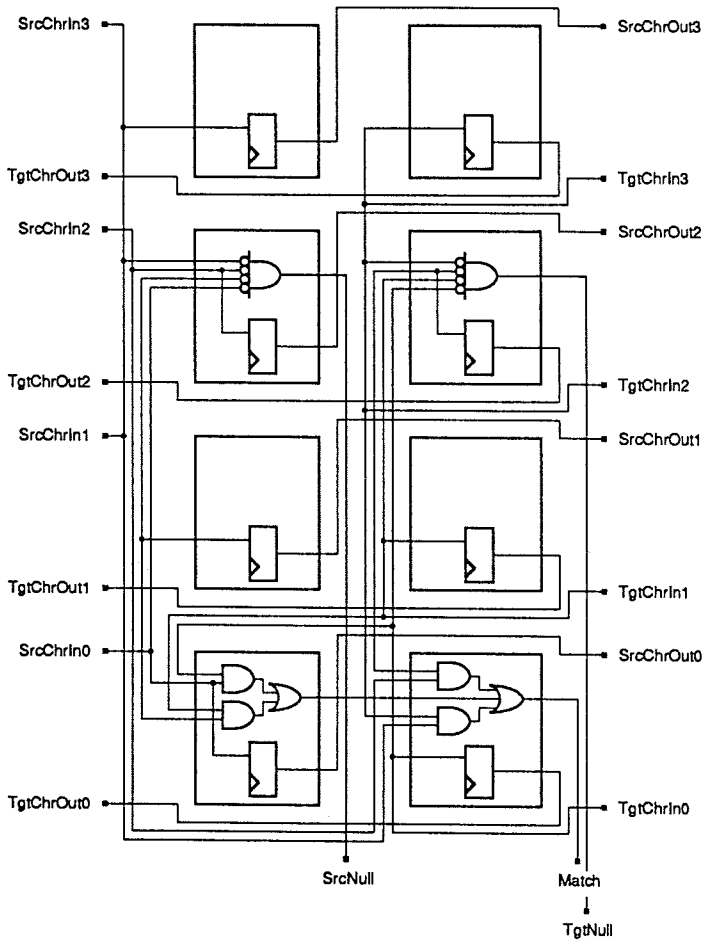

Figure 8: Character comparator logic

Processor counts for each of the chips and arrays are shown in Figure 11. The *DNA long* version is more than twice as long as the P-NAC prototype. In addition, I present throughputs for the implementations in terms of *cell updates per second* (CUPS), one way of normalizing performance figures across problem size (comparing two sequences

each 100 characters long involves $100 \times 100 = 10{,}000$ cell updates). As a rough rule of thumb, a typical sequential machine (VAX, Sun, etc.) will require approximately 10 machine instructions to perform a cell update, so in some sense these numbers represent an effective rate for SPLASH of nearly 2 billion instructions per second.

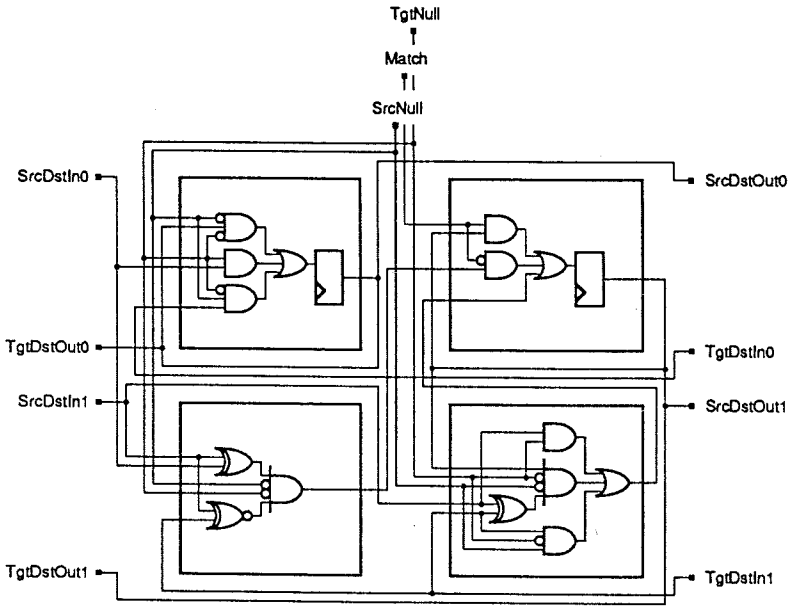

Figure 9: Finite state machine logic

Figure 12 gives the results of a more traditional benchmark comparing SPLASH to a variety of machines including P-NAC and several commercial supercomputers [6]. The test was to perform 100 comparisons of 100-long DNA sequences. (I take responsibility for having written all the code in question. While I attempted to show each machine at its best, it is quite possible that an experienced "hacker" could improve these times somewhat. Even so, it is clear SPLASH will remain orders of magnitude faster than the others.)

	Chips			
Version	*X0*	*X1–X30*	*X31*	*Array Total*
DNA short	48 (15%)	96 (30%)	60 (19%)	2,988 (29%)
DNA long	156 (49%)	288 (90%)	168 (52%)	8,964 (87%)
ASCII short	160 (50%)	150 (50%)	170 (53%)	5,130 (50%)
ASCII long	181 (56%)	320 (100%)	192 (60%)	9,973 (97%)

Figure 10: CLB usage statistics

4 Conclusions and Ongoing Research

At present we have the four different versions of sequence comparison up and running on SPLASH. Researchers at the National Institutes of Health (NIH) have seen the work and were sufficiently impressed with its potential that they placed an order for a copy of the hardware. So, with a little luck, SPLASH will not suffer the same fate as befell P-NAC. (Actually, there are other applications running on SPLASH that alone would justify the research.) We are now looking at other problems of interest to NIH that seem like they would work well on SPLASH.

	Chips			*Array*	
Version	*X0*	*X1–X30*	*X31*	*Total*	*CUPS (max)*
DNA short	4	8	4	248	61,200,000
DNA long	13	24	13	746	186,000,000
ASCII short	8	8	8	256	63,200,000
ASCII long	9	16	9	498	124,000,000

Figure 11: Processor counts and maximum throughputs

The biggest obstacle to the widespread acceptance of SPLASH now is the design environment. The logic paradigm is very low level and there are myriad details that must be mastered. It had been hoped that developers coming from a straight software background could learn to program SPLASH, but this has not been the case. On the user side, work needs to be done on higher-level models that can be mapped efficiently onto the hardware. Perhaps some of the results derived for custom VLSI synthesis can be brought to bear. On the machine side, the situation seems less promising. Any attempts to improve on the status quo would be impeded by the proprietary nature of the Xilinx FPGA. For example, we still do not have enough information to consider writing our own router even after having used the parts for two years. Xilinx is understandably reluctant to divulge such details.

System	*Time*	*Speed-Up*	*Note*
SPLASH	0.020 s	2,700	1 MHz, Sun 3/260 host
P-NAC	0.91 s	60	Sun 2 host
Multiflow Trace	3.7 s	14	cc -O5, 14 functional units
Sun SPARCstation 1	5.8 s	9.3	cc
Cray 2	6.5 s	8.3	vp, one head
Convex C1	8.9 s	6.0	vc -O2
DEC VAX 8600	31 s	1.7	cc
Sun 3/140	48 s	1.1	cc
DEC VAX 11/785	54 s	1.0	cc

Figure 12: Comparative benchmark results

Turning to the title of this paper, there may be some question as to whether anything that takes two months can be called "rapid." Certainly when compared to standard software development this is a long time (considering the simplicity of the algorithm). It is, however, half the time it took to build the P-NAC prototype. Furthermore, the quality of the SPLASH implementations are unquestionably higher. The LDG code is well documented and can be modified as needed (after the first version, *DNA short*, it took me only a day or two to finish each of the others). But perhaps "rapid" speaks more for the promise this technology holds. There is no real reason an iteration of the design cycle should take half a day as it does now; the support software can only get better. Even today, though, low-cost FPGA machines like SPLASH are a practical alternative to custom VLSI or traditional supercomputers for many systolic pattern matching applications.

5 Acknowledgements

The work described in this paper was performed by the author during the 1989 and 1990 Supercomputing Research Center Summer Workshops. A large number of people at SRC and elsewhere played a role in the development of SPLASH, including Neil Coletti, Elaine Davis, Maya Gokhale, Randy Hammel, Bill Holmes, Andy Kopser, Dick Kunze, Dick Lipton, Sara Lucas, Ron Minnich, Fred More, Mark Norder, Peter Olsen, Paul Schneck, Burton Smith, and Doug Sweely.

At Princeton, Dick Lipton, Rich Schaefer, and Doug Welsh all assisted me with the development of P-NAC. Finally, the National Science Foundation deserves my thanks for supporting my research at Brown on parallel VLSI architectures for sequence comparison (under grant MIP 87-10745).

References

[1] M. Gokhale, et. al., "SPLASH: A Reconfigurable Linear Logic Array," *Proceedings of the 1990 International Conference on Parallel Processing*, August 1990, pp. 526-532.

[2] M. Gokhale, et. al., "Experience Building and Using a Highly Parallel Programmable Logic Array," to appear in *Computer*.

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

[4] R. J. Lipton and D. P. Lopresti, "A Systolic Array for Rapid String Comparison," *1985 Chapel Hill Conference on VLSI*, H. Fuchs, ed., Rockville, MD: Computer Science Press, 1985, pp. 363-376.

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

[6] D. P. Lopresti, "Sequence Comparison on Commercial Supercomputers," SRC-TR-89-010, Supercomputing Research Center, Bowie, MD, October 1989.

[7] J. M. da Mata, "The ALLENDE Layout System User's Manual," Princeton University VLSI Memo, no. 9, 1984.

[8] R. N. Mayo, J. K. Ousterhout, and W. S. Scott, eds., *1983 VLSI Tools*, University of California at Berkeley, UCB/CSD 83/115, March 1983.

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

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

[11] Supercomputing Research Center, *SPLASH User's Manual*, ver. 2, May 1990.

[12] J. D. Watson, "The Human Genome Project: Past, Present, and Future," *Science*, vol. 248, April 6, 1990, pp. 44-48.

[13] Xilinx, Inc. *The Programmable Gate Array Data Book*, 1989.