A Systolic Array for the Sequence Alignment Problem

Dzung T. Hoang

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-92-22
April 1992

A Systolic Array for the Sequence Alignment Problem

Dzung T. Hoang*

Department of Computer Science
Brown University
Providence, RI 02912

April 13, 1992

Abstract

This report introduces a new systolic algorithm for the sequence alignment problem. This work builds upon an existing systolic array for computing the edit distance between two sequences. The alignment array is meant to be used as the second phase in a two-phase design with a modified edit distance array serving as the first phase. An implementation on the SPLASH programmable logic array is described. Because of the extensive pipelining in the systolic array, computing an alignment on the array takes that same amount of time as computing just the edit distance. Compared to conventional computers, SPLASH implementation performs several orders of magnitude faster.

1 Introduction

The work presented in this report was begun during the author's summer internship at the National Cancer Institute's Laboratory of Mathematical Biology in Fredrick, Maryland. The goal was to develop sequence comparison algorithms for the SPLASH programmable logic array. A systolic sequence comparison algorithm that computes the edit distance between a pair of sequences had already been implemented on SPLASH [1]. Certain applications of interest to biologists at the laboratory, such as multiple alignment of genetic (DNA and RNA) sequences, however, require more than just the edit distance: a more informative analysis of the similarity, or homology, of the sequences in the form of an alignment is required. This paper introduces a novel systolic array that produces an alignment of two sequences in the same amount of time it takes to compute the edit distance, which is linear in the lengths of the sequences.

*Supported during Summer 1991 by an NIH Summer Internship and afterwards by an NSF Graduate Fellowship.

The next few sections of this paper introduces the sequence comparison problem, outlines a simple dynamic programming solution, and introduces alignments. An existing systolic array that computes the edit distance is then described. A new systolic array which builds upon the previous one to generate an alignment is presented. Next, we provide an overview of the SPLASH programmable logic array and describe an implementation of the alignment algorithm on SPLASH. The performance of the new algorithm on SPLASH is compared to standard implementations on conventional computers.

2 Sequence Comparison

The need for a quantitative measure of similarity between two or more sequences of possibly different lengths arises in several disciplines, such as molecular biology, speech processing, handwriting analysis, and text processing [2]. We discuss sequence comparison as it is applied to genetic sequences, but the methods presented here are adaptable to other applications.

2.1 DNA Sequences

Deoxyribonucleic acid (DNA) is a macromolecular chain of nucleotides that serves as a carrier of genetic information. DNA can be represented as a sequence of nucleotide bases, A, C, G, and T. Biologists have developed methods for determining the sequence of bases from a strand of DNA. DNA sequences are typically from thousands to millions of bases long, and new sequences are determined at an increasing rate. With the onset of the Human Genome Initiative [3] and constant improvement of sequencing technology, the size of GenBank and similar databases have grown and will undoubtedly continue to grow at a steady rate[1]. The analysis of a newly generated sequence typically involves searching the database for similar sequences. With the enormous size of the database, fast methods are needed for comparing sequences.

2.2 Edit Distance

In comparing two sequences, it is useful to quantify their similarity in terms of a distance measure. In general, the correspondence between individual elements of the sequences to be compared is not known *a priori*. Therefore common distance measures such as Euclidean distance and Hamming distance, in which elements correspond in position and only corresponding elements are compared, may not be appropriate. Instead, the correspondence between individual elements of the sequences to be compared needs to be determined during the distance computation.

Biologists have developed several means to characterize the similarity between two DNA sequences. One intuitively appealing measure is *edit distance*, which is defined to be the minimum cost of transforming one sequence into another through a series of the following operations: deletion of a nucleotide, insertion of a nucleotide, and substitution of one nu-

[1] Release 70 of GenBank, a database of DNA sequences, contains 58,952 entries with a total of 77,337,678 bases as of December 1991. It is estimated that by 1999, 1.6 billion base pairs will be sequenced each year [4].

cleotide for another[2]. Each operation has an associated cost, which is a function of the nucleotides involved in the operation. The cost of the transformation is then the sum of the costs of the individual operations. We refer to the problem of computing the edit distance between two sequences as the *Sequence Comparison Problem.*

2.3 Dynamic Programming

The calculation of the edit distance can be done with a well-known dynamic programming algorithm [5, 6]. Let $S = s_1 s_2 s_3 \cdots s_m$ be the source sequence, $T = t_1 t_2 t_3 \cdots t_n$ be the target sequence, and $d_{i,j}$ be the edit distance between the subsequences $s_1 s_2 \cdots s_i$ and $t_1 t_2 \cdots t_j$. Then

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

and

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

Here $c_{del(s_i)}$ is the cost of deleting s_i, $c_{ins(t_j)}$ is the cost of inserting t_j, and $c_{sub(s_i,t_j)}$ is the cost of substituting t_j for s_i. For example, Figure 1 shows the distance table generated when comparing the sequences $AGACTAGG$ and $TGCTAAGC$ with the following cost functions:

$$
\begin{aligned}
c_{del(a)} &= 1, \\
c_{ins(a)} &= 1, \\
c_{sub(a,b)} &= \begin{cases} 0, & \text{if } a = b, \\ 2, & \text{if } a \ne b. \end{cases}
\end{aligned} \tag{2}
$$

The resulting edit distance is 6, which is found at the lower right-hand corner of the table.

2.4 Alignment

The edit distance provides a measure of the similarity between two sequences, but alone it does not indicate how the sequences are related. One straightforward way to qualify differences is to list the series of operations required to transform the source sequence into the target sequence. Such a list is called a *listing.* Figure 2 shows one possible listing for the sequences compared in Figure 1. Listing is an intuitive and informative form of presentation for human readers, but is cumbersome and poorly suited for algorithmic representation. An alternative form form of presentation better suited for computation is the *alignment.* In an alignment, the characters of the source and target sequences are arranged in a matrix with two rows. The source sequence, possibly with intervening null characters, '$-$', is placed in the first row. Similarly, the characters of the target sequence are placed in the second row. The matrix is analyzed column-wise. A column containing $\left[\begin{smallmatrix} x \\ - \end{smallmatrix}\right]$ indicates deletion of the character x; a column containing $\left[\begin{smallmatrix} - \\ y \end{smallmatrix}\right]$ indicates insertion of the character y; and a column

[2]In other areas, a different set of operations may be defined. For example, in text processing, a swap of two adjacent characters may be considered an edit operation.

		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: Dynamic programming table of distances. The lower right-hand entry gives the edit distance, 6 in this case.

$\begin{bmatrix} x \\ y \end{bmatrix}$ indicates substitution of y for x. A column consisting of two nulls is not allowed. Four possible alignments of the sequences in Figure 1 are shown in Figure 3. We call the problem of finding an alignment between two sequences the *Sequence Alignment Problem*.

Alignments can be computed with a minor addition to the dynamic programming algorithm. By storing a set of pointers in the table to indicate the minimization paths, the alignments can be constructed by following the pointers from the lower-right corner to the upper-left corner. The pointers can be created as the dynamic programming table is constructed: when evaluating Equation 1 for $d_{i,j}$, a pointer is added from location (i, j) to one or more of the locations $(i-1, j-1)$, $(i-1, j)$, or $(i, j-1)$ from which the minimum value is computed. For example, if the minimum value in Equation 1 is $d_{i-1,j} + c_{del(s_i)}$, then a pointer is added from (i, j) to $(i-1, j)$. Because more than one argument of the *min* function in Equation 1 can have the minimum value, there may be more than one pointer originating from (i, j). The dynamic programming table in Figure 1 augmented with pointers is shown in Figure 4.

Alternatively, the pointers can be constructed after the DP table has been computed. The pointers originating from (i, j) can be determined from $d_{i-1,j-1}$, $d_{i-1,j}$, $d_{i,j-1}$, s_i, and t_j by reevaluating Equation 1.

Given a table with pointers, an alignment can be constructed (in reverse, rightmost column first) by following the pointers from the lower-right corner of the table to the upper-left corner. Following a diagonal pointer corresponds to a substitution, a horizontal pointer to an insertion, and a vertical pointer to a deletion. For example, following the bold pointers in Figure 4 would result in the first alignment in Figure 3.

3 Systolic Sequence Comparison

One notable property of the dynamic programming recurrence for computing edit distances is that each entry in the DP table depends on adjacent entries, as shown in Figure 5. This

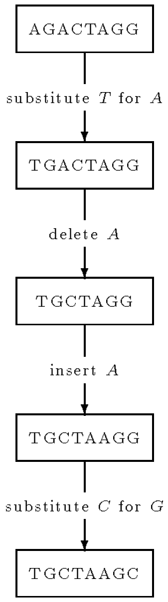

Figure 2: Listing of operations to change $AGACTAGG$ to $TGCTAAGC$

$$\begin{bmatrix} A & G & A & C & T & A & - & G & G \\ T & G & - & C & T & A & A & G & C \end{bmatrix}$$

$$\begin{bmatrix} A & G & A & C & T & A & - & G & - & G \\ T & G & - & C & T & A & A & G & C & - \end{bmatrix}$$

$$\begin{bmatrix} A & - & G & A & C & T & - & A & G & - & G \\ - & T & G & - & C & T & A & A & G & C & - \end{bmatrix}$$

$$\begin{bmatrix} - & A & G & A & C & T & A & - & G & G & - \\ T & - & G & - & C & T & A & A & G & - & C \end{bmatrix}$$

Figure 3: Four possible alignments of $AGACTAGG$ and $TGCTAAGC$

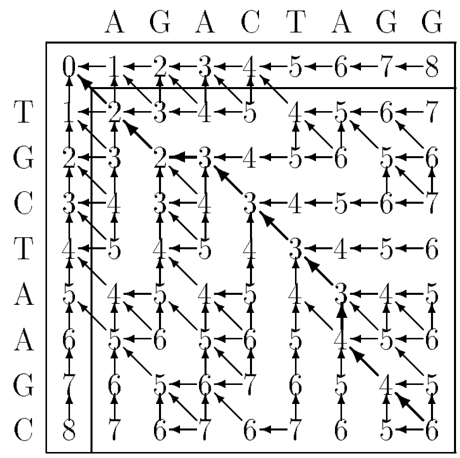

Figure 4: Dynamic programming table with minimization pointers

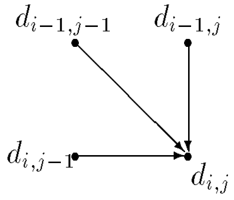

Figure 5: Locality of computation

local dependence can be exploited to produce systolic algorithms in which communication is limited to adjacent processors. The diagonal mapping, shown in Figure 6, was used in the design of the Princeton Nucleic Acid Comparator (P-NAC) [7]. In the figure, the antidiagonal lines denote the linear connection of processing elements (PE's), and the dots denote the active PE's at each step. At each step, all the entries along one antidiagonal of the DP table are computed. Figure 7 depicts the data flow through the systolic array. The input sequences flow systolically through the array in opposite directions. In addition, there is one distance stream associated with each character stream. The distance streams are used to input boundary values in the DP table and to transport the edit distance out. At any instance in time, the contents of the streams contained inside a given PE denote the characters to be compared and the distances along one of the antidiagonals of the DP table. In order to insure that every character in one stream is compared to every other character in the other stream, the characters are entered into the array with a null character inserted between consecutive characters. At the end of the computation, the resulting edit distance emerges at the ends of the distance streams.

Comparing sequences of lengths m and n requires at least $2 \cdot \max(m+1, n+1)$ PE's. The number of steps required to compute the edit distance is of $O(m+n)$. There is also a pipeline delay proportional to the length of the array.

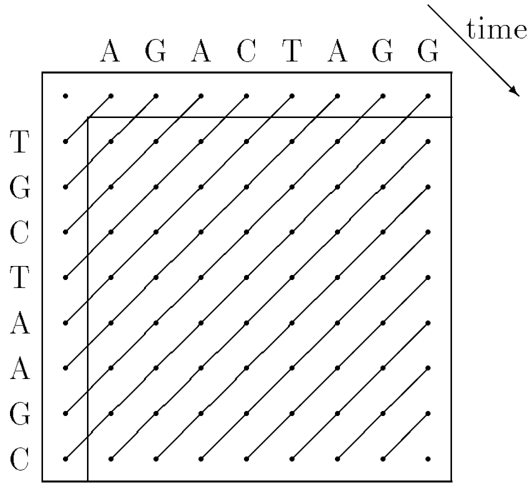

Figure 6: Diagonal mapping of DP table to systolic array

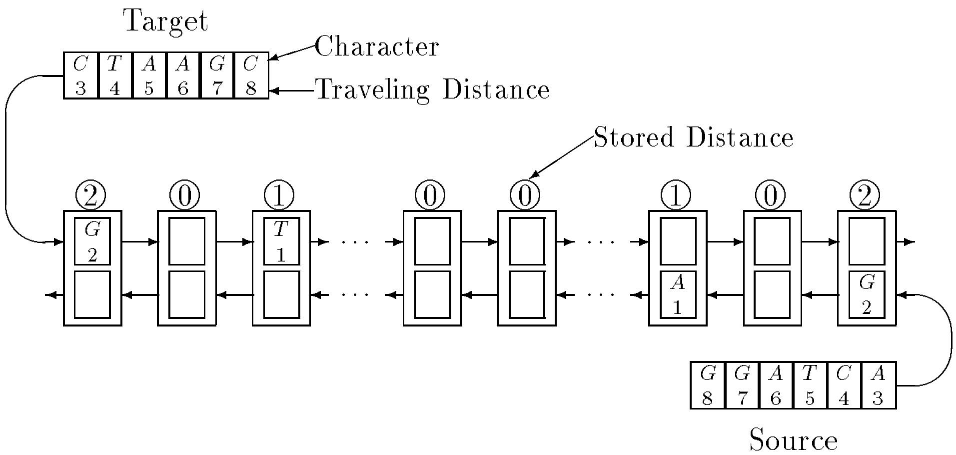

Figure 7: Data flow through linear systolic array

While working on P-NAC, Lopresti discovered that the distances can be represented in a fixed number of bits [7]. Using the cost functions (2), only two bits are required to encode the distances, regardless of the length of the array. This results in a systolic array that is easily scalable to any length without change to the basic processing element. With the new encoding, the PE's compute and store distances modulo 4. Using the cost functions (2), Lopresti proved that adjacent entries in the DP table differ by ± 1. With this result, an encoded distance stream can be decoded as it exits the systolic array with a simple finite state machine controlling an up/down counter. The FSM looks at consecutive exiting distances and increments or decrements the counter accordingly.

4 Systolic Alignment

In this section, we introduce a systolic alignment array that operates in linear time and requires quadratic space. This array is designed to be used as the second in a two-phase algorithm, with the sequence comparison array in Figure 7 serving as the first phase.

4.1 Phase One: Dynamic Programming

In the first phase, the dynamic programming systolic array discussed earlier is used to compute the edit distance and generate an intermediate table which is used in the second phase to generate an alignment.

In Section 2.4, it is shown that an alignment can be constructed by following the minimization pointers from $d_{m,n}$ to $d_{0,0}$. The pointers can be computed at the same time as the minimization function in Equation (1) and stored for later processing. Alternatively, the DP distances can be saved and used later to compute the pointers. Unfortunately, in either case, the amount of data that needs to be saved is proportional to $m \cdot n$, where m and n are the lengths of the input sequences. The table data, therefore, cannot be carried on systolic streams, as the input sequences and traveling distances are. Thus another mechanism must be provided to store the data.

Depending on whether pointers or distances are to be stored, two versions of the sequence alignment array would result. In the sequel, we develop the array using stored distances. The version for stored pointers would be similar, but simpler.

In terms of memory requirements, storing the pointer originating from a particular position in the DP table would require three bits, one for each of the three directions, independent of the set of cost functions used. For the specific cost functions (2), however, only one bit is required to store each encoded distance[3].

The distances can be saved by adding data streams perpendicular to the systolic streams at each processor (Figures 8 and 9). These data streams carry the computed distances from the processor array to be stored either externally in the host, or locally in RAM. In the SPLASH implementation of the alignment algorithm, the latter approach was chosen, taking advantage of the onboard RAM. In the following discussion, we refer to to RAM for convenience.

[3] For a different set of cost functions, more bits may be needed.

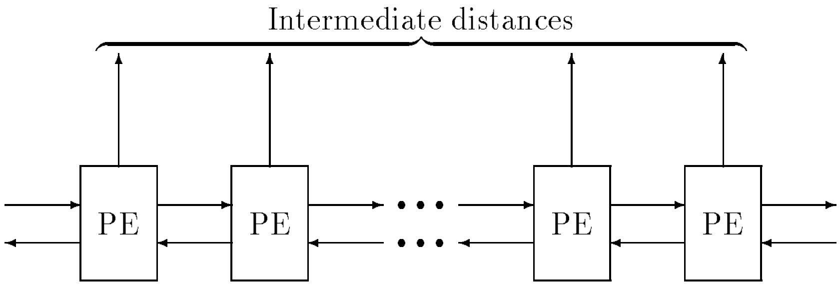

Figure 8: Saving distance table

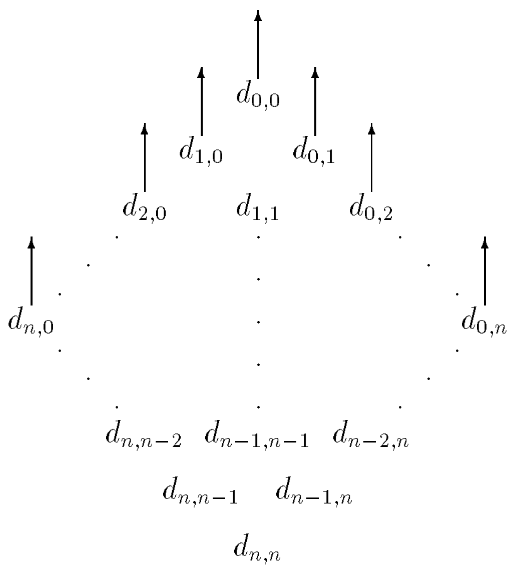

Figure 9: Flow of distances from PE's (Shown here for $m = n$.)

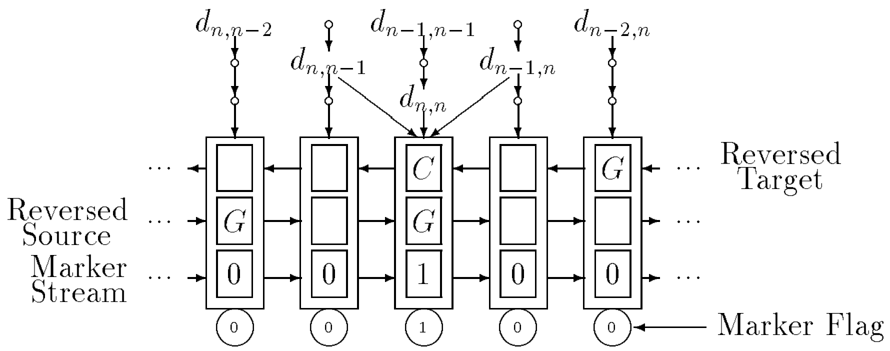

Figure 10: Systolic array for generating alignment

4.2 Phase Two: Alignment Construction

The PE for alignment construction is diagrammed in Figure 10. If stored pointers were used, the character streams would be omitted, resulting in a simpler array. The alignment PE's are assigned the same antidiagonal mapping as with the sequence-comparison PE's (see Figure 6). The distances saved in local memory are fed to the PE's in reverse order, with the memory operating as a stack. There are three systolic streams, two input streams that carry the source and target sequences in reverse order and an output stream that indicates the movement of a *marker* used to follow the minimization pointers. Note that the directions for the source and target sequence streams are the reverse from their directions in the phase one array. Initially, the PE that is to receive $d_{m,n}$ has its marker flag set and the rest have theirs cleared. At each step, the PE with its marker flag set compares its distance with the neighboring distances and the source character with the target character to determine whether to keep the marker or to send the marker to one of its neighbors. In essense, this step reevaluates Equation 1 to determine the minimization pointers. The PE keeps the marker if there is a diagonal minimization pointer, indicating a substitution. The marker is passed to the left PE if there is a horizontal pointer, indicating an insertion. The marker goes to the right PE if there is a vertical pointer, indicating a deletion. There may be more than one pointer present, in which case the PE makes an arbitrary choice[4].

In order to construct an alignment, we need to know the position of the marker at each step in the pointer-traversal process. We can take advantage of the locality of the pointers (see Figure 5) to simplify the tracking of the marker. With the diagonal mapping of PE's, we observe that the marker can move a maximum distance of one PE away from its current position. Instead of reporting the absolute position of the marker, we can report the movement of the marker: left, right, or stationary. Since the marker may only move a maximum distance of one PE, the marker's position can be registered on a systolic stream that moves across two PE's at each step. The registration can be done by having the PE with its marker flag set to copy the flag to the corresponding position in the marker stream. This step can be made uniform across all PE's by having each copy its marker flag to the

[4]The deterministic nature of the algorithm limits it to providing only one minimum-cost alignment out of possibly many.

stream using the logical OR operator. The character streams are then transferred one PE downstream, while the marker stream is transferred two PE's downstream. Figure 11 shows a partial trace of the alignment array.

The data that exits the marker stream encodes the movement of the marker flag, which is used to construct the alignment in reverse. From an intuitive examination of the marker registration scheme and marker stream flow described above, one can observe that the number of 0's between two successive 1's corresponds to direction in which the marker moves in one step. One intervening 0 indicates that the marker did not move in one step, therefore implying a substitution or match; two intervening 0's indicate that the marker moved to the left, implying an insertion; and no intervening 0's indicates that the marker moved to the right, implying a deletion. A closer study of the array timing reveals that for a substitution, the marker remains in the same PE for two steps, as contrast to a deletion or insertion, in which the marker moves at each step. Therefore, for a substitution, the marker pattern of one 0 between two successive 1's would be repeated twice, resulting in a marker stream encoding of 10101. The marker decoding process is more formally described by the FSM in Figure 12. As an example, the alignment

$$\begin{bmatrix} A & G & A & C & T & A & - & G & G \\ T & G & - & C & T & A & A & G & C \end{bmatrix}$$

would be constructed from the marker sequence 10101010100101010101010101101010101, read left to right.

5 SPLASH

The systolic alignment array has been implemented on the SPLASH programmable logic array [8, 9]. Before describing the implementation, we include a brief introduction to SPLASH; `LDG`, a logic design entry language; and `trigger`, an interactive symbolic debugger for SPLASH.

5.1 The SPLASH Architecture

SPLASH is a programmable linear logic array developed by the Supercomputer Research Center (SRC) as a an coprocessor card for the Sun VME bus. The SPLASH board contains 32 Xilinx XC3090 field-programmable gate arrays (FPGA) [10] with local connections to 32 1M-bit (128K by 8) static RAM chips, as shown in Figure 13. The FPGA's are connected linearly in a ring with input coming from a 32-bit FIFO queue connected to chip 0 and output going to a 32-bit FIFO queue connected to chip 31. The FIFO queues have a limited access speed and can be clocked up to a maximum of 1MHz, resulting in a maximum I/O bandwidth of 32Mbits/sec. A RAM chip is connected between each pair of adjacent FPGA chips and can be accessed by either of the adjacent FPGA's. The data path connecting the FIFO's to the array consists of 36 unidirectional lines, 32 for data and 4 for control signals. Adjacent FPGA's, except for chips 0 and 31, are joined by a 68-bit programmable bidirectional bus, which shares connections to the local RAM. Chips 0 and 31 are connected

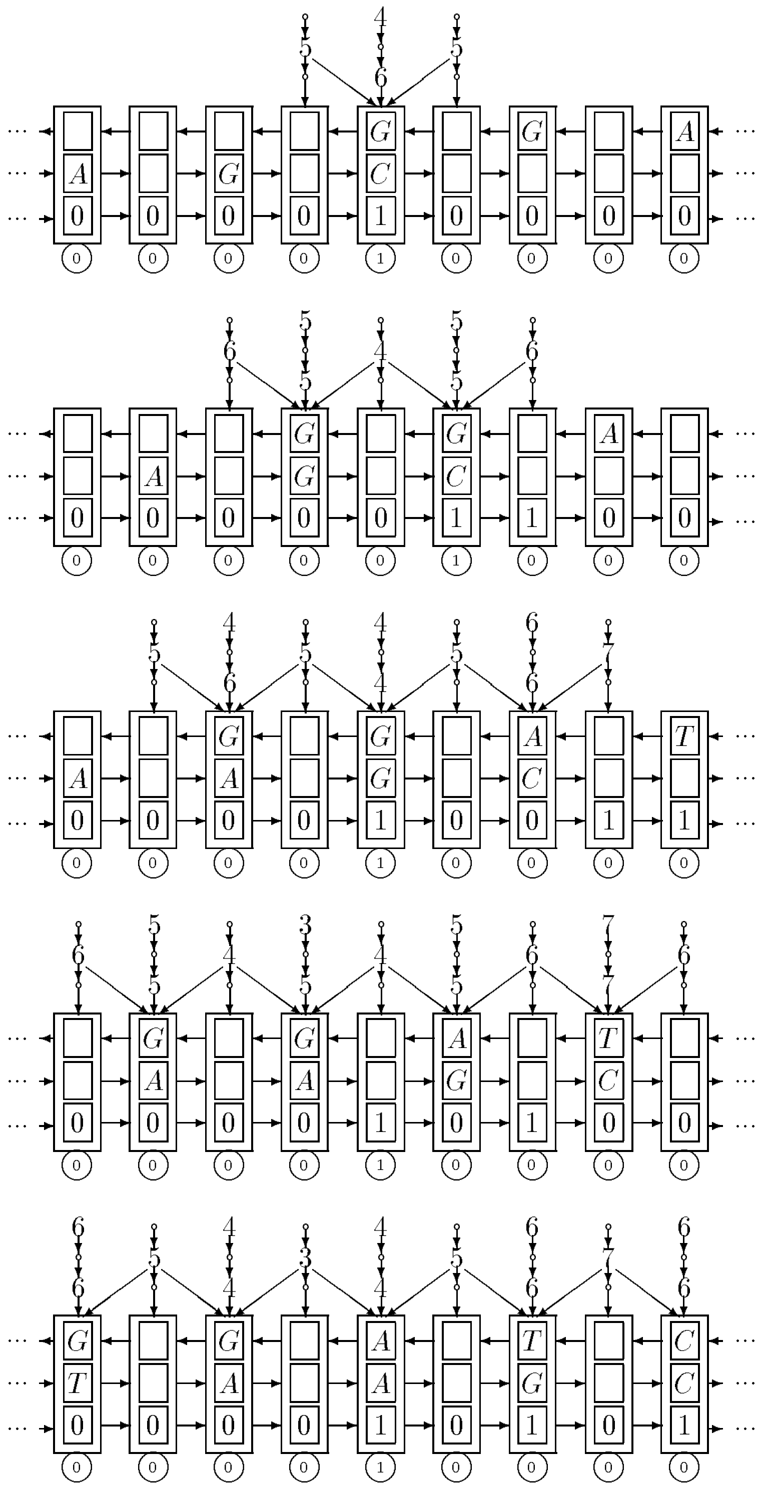

Figure 11: Partial trace of alignment array

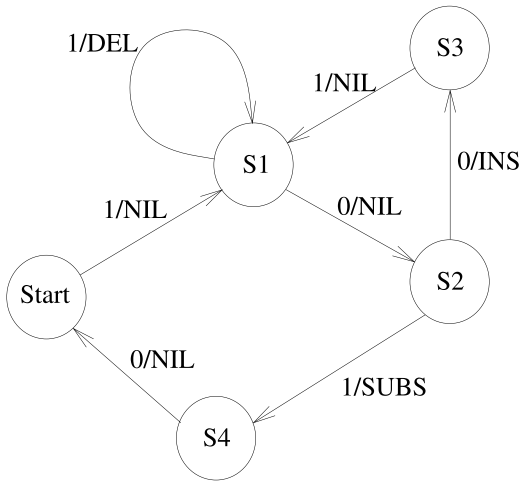

Figure 12: FSM for generating alignment from marker stream

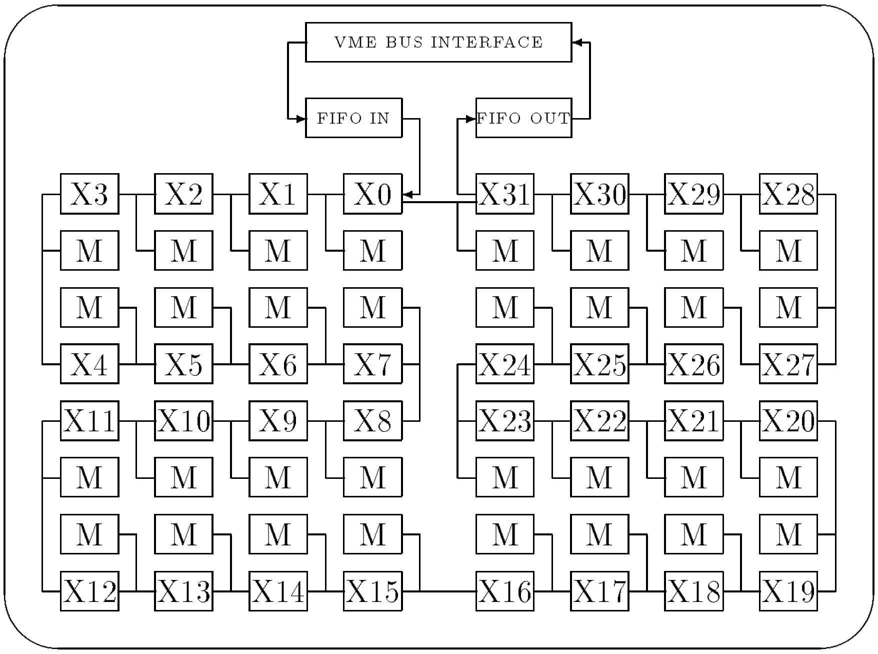

Figure 13: SPLASH

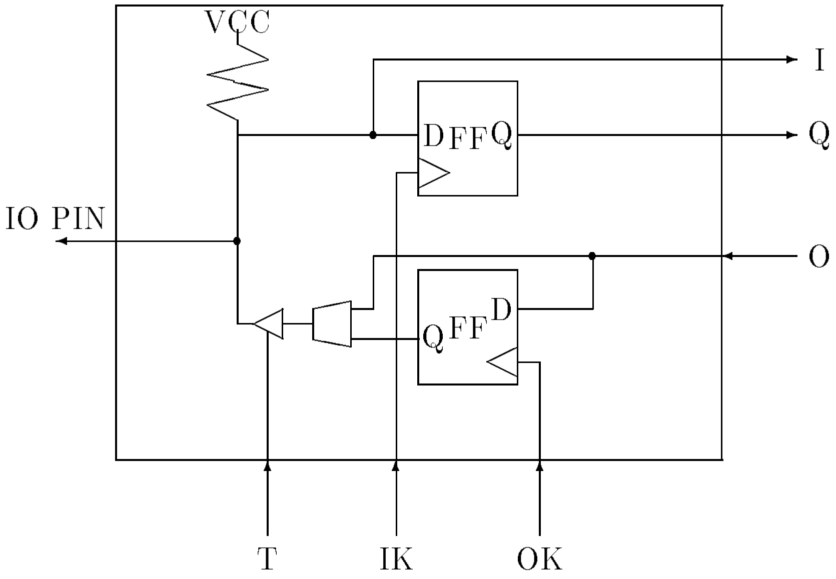

Figure 14: Diagram of IOB configuration

with a 35-bit data path. This "wrap-around" connection allows data flow through the array in either direction.

At the heart of the SPLASH board are the Xilinx XC3090 FPGA's. Each FPGA contains 320 configurable logic blocks (CLB's) arranged in a 20×16 grid and surrounded by 144 input/output blocks (IOB's). Each CLB contains a combinational logic section, two D flip-flops, and an internal control section. The logic section can be configured to provide three forms of outputs: a single output of a boolean function of five inputs (F), two outputs of two individual boolean functions of four inputs (FG), and a output which can be switched between two boolean functions of four inputs (FGM). The F configuration is shown in Figure 15. The outputs of the logic section can either be output directly or registered through D flip-flops.

The 144 IOB's surrounding each XC3090 FPGA provides connections to the control bus and programmable interconnections between adjacent chips and local RAM. Each IOB can be configured as either an input port, an output port, or a bidirectional input/output port, with optional latch or flip-flop operation, as shown in Figure 14. The programmability of the IOB's allows for flexibility in the interchip connections. For example, when the local RAM is not needed, it can be disabled and the IOB's connected to the RAM's address and data lines can be used for communication between adjacent FPGA's.

In summary, the structure of the CLB's are ideal for implementing finite state machines, combinational logic, and registered data streams, which are present in typical systolic processor designs. The numerous programmable IOB's allow flexible connections between FPGA's and local RAM. The combination of flexible CLB's, numerous programmable input/output ports, large local RAM's, and a linear connection of FPGA's makes SPLASH a powerful programmable systolic engine.

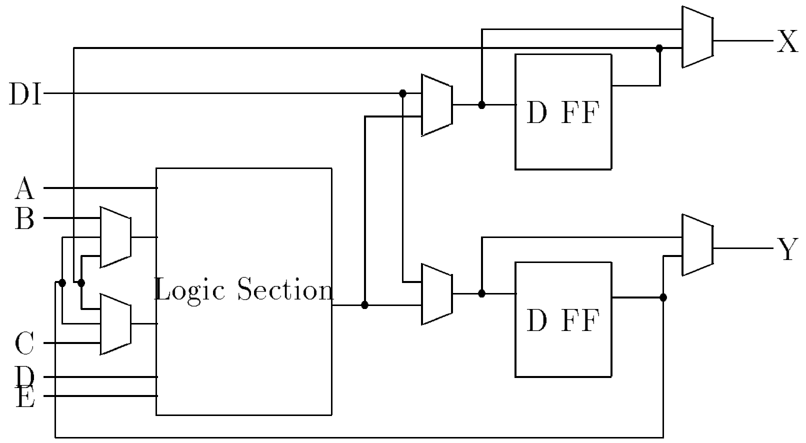

Figure 15: F configuration of CLB

5.2 The Logic Description Generator

The Logic Description Generator (`LDG`) is the "programming-language" for SPLASH developed at the SRC. `LDG` is programmed in and runs on top of Common Lisp. As such, its syntax is based on Lisp. In addition, the programmer has access to Common Lisp functions, such as the arithmetic functions.

As the name implies, one programs in `LDG` at the logic design level. `LDG` allows the programmer to specify the configurations and placement of individual CLB's and IOB's. Configured CLB's and IOB's can then be invoked and interconnected inside a *template*, which functions like procedures in `Pascal` or `C`. A template has an input parameter list, and output parameter list, and a location parameter. The inputs and outputs are signals and can be either a simple variable or an array variable. Arrays are specified by giving a name and two bounds, which may be increasing or decreasing. In addition, array variables may be indexed. Templates can be invoked in other templates in a hierarchical fashion, allowing replication of basic blocks.

`LDG` has looping constructs to allow for replication and interconnection of basic units. `LDG` also provides a library of parameterized parts called *schemas*, which can be invoked to instantiate commonly used logic components. One such schema is a binary counter whose length and counting direction are two parameters. Another example is a shift-register whose size and layout (horizontal or vertical) are parameterized.

Figure 16 contains `LDG` code for an array of four 2-bit multiplexers. This example illustrates CLB configuration, template definition, and looping constructs. First, a CLB containing two 2-bit multiplexers is defined. The CLB is configured in FG mode. The logic equation for a 2-bit multiplexer is specified for the logic functions, F and G. The outputs of the logic section are connected directed to the output pins. A template is defined to replicate the multiplexer CLB to form a unit consisting of four 2-bit multiplexers. The **range** clause acts like a **for** loop with **!** as the loop variable. The **shape**, **start-!row**, and **start-!col** clauses specify a vertical positioning of the CLB's through the variables **!row** and **!col**.

For a more complete discussion of `LDG`, see [11].

```
;;; two mux's in one clb
(clb mux2
    (input B C D E A)
    (output X Y)
    (location row col)

    (config FG)
    (F ((~A * B) + (A * D)))
    (G ((~A * C) + (A * E)))
    (X F)
    (Y G)
)

;;; 8-bit data multiplexer
;;; inputs: mux_sel mux_clk (array A 7 0) (array B 7 0)
;;; outputs: (array C 7 0)
;;; mux_sel = L --> C = A, mux_sel = H --> C = B
(template data_mux8
    (input sel (array dataA 7 0) (array dataB 7 0))
    (output (array dataC 7 0))
    (location row col)

    (part-list
        (
        (name MUX)
        (range! 0 to 6 step 2)
        (shape row-major 4 by 1)
        (start-!row row step 1)
        (start-!col col step 1)
        (part mux2)
        (input (index dataA !) (index dataA (1+ !))
               (index dataB !) (index dataB (1+ !))
               sel)
        (output (index dataC !) (index dataC (1+ !)))
        (location !row !col)
        )
    )
)
```

Figure 16: LDG code for a data multiplexer

5.3 Trigger

`LDG` programs are tested and debugged using `trigger`, an interactive symbolic debugger written in `C`. `trigger` works like `dbx` in allowing users to load and debug programs. With `trigger`, a programmer can load programs onto SPLASH, step through the program one or more clock cycles at a time, readback and display the values of variables contained in flip-flops on the FPGA's, and perform a variety of control functions, such as reseting the whole array. In addition, the user can define procedures to automate repetitive tasks, display formatted debugging information, or perform other algorithmic tasks such as data formatting and interrupt handling. `trigger` also provides global variables and supports common programming constructs such as `if` statements, `while` loops, and `for` loops.

Once a program is successfully debugged with `trigger`, the programmer can write a `C` driver program to control SPLASH using `trigger`'s run-time library.

6 SPLASH Implementation

In this section, we describe the implementation of the DNA sequence alignment algorithm on SPLASH. We use the cost functions (2) and optimize the array for this particular set of cost functions.

As described in Section 4, the alignment algorithm uses two systolic arrays acting in two-phases. In the first phase, the DP table is computed and stored. In the second phase, the DP table is read to create an alignment. The systolic arrays for both phases exist in SPLASH concurrently and their operations are controlled by a FSM controller. We discuss both arrays and the controller below in separate sections.

6.1 Phase One: Sequence Comparison

The phase one array is basically the sequence comparison design described in [1] modified to store the computed DP distances in local RAM. Figure 17 shows a block diagram of the sequence comparison PE. The SPLASH implementation uses 13 CLB's per PE, eight for the character comparator and five for the finite state machine. The character comparator contains registers to store a source character and a target character and combinational logic to compare them. The finite state machine holds the stored distance and contains logic to update the distance.

Lemma *For the specific cost functions (2), the values of the horizontal and vertical neighbors in the DP table differ by ± 1. That is, a 2×2 table fragment must look like:*

$$\begin{array}{cc} a & b = a \pm 1, d \pm 1 \\ c = a \pm 1, d \pm 1 & d \end{array} .$$

Using the results of the above lemma due to [7], we are able to store each modulo-4-encoded distance using only one bit[5]. We take the most significant bit of the modulo-4 distance and store that in RAM. Since the RAM chips attached to each FPGA chip has a

[5]In essence, we are storing only in the change information, ± 1, which requires one bit to encode.

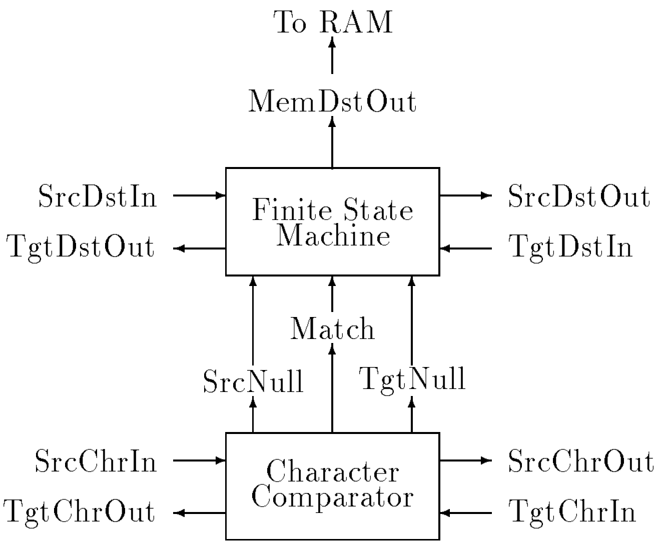

Figure 17: Block diagram of the sequence comparison PE

data path that is 8-bits wide, for convenience, we placed eight PE's on each chip, except for chips X0 and X31, where only placed four PE's per chip to leave room for I/O logic. To address the onboard RAM, each chip contains an `UP/DOWN` address counter which counts in unison with the other address counters. In this phase, the direction of the counters is set to `UP`, and the RAM control logic set to `WRITE` mode. From Figure 6, we observe that at any time step, every other PE is active. Therefore, after each time step, only four bits are to be stored in each chip. Instead of storing four bits and incrementing the address counter at every step, we halve the number of memory write operations by waiting a step and storing eight bits every other step. The sequence comparison array with memory-save logic is diagrammed in Figure 18.

6.2 Phase Two: Sequence Alignment

After the last distance entry in the DP table, $d_{m,n}$, has been computed by the sequence comparison array, the driver program executing on the host computer sends a control signal to each chip to start the alignment array. In the initialization steps, the marker flag in the PE that computed $d_{m,n}$ is set[6]. In addition, the address counter is set to count `DOWN`, and the RAM control logic set to `READ` mode.

In Figure 10, the source and target streams are shown carrying the respective sequence in reverse and in opposite direction from the corresponding streams in Figure 7. Instead of implementing these streams using additional flip-flops, we can reuse the streams in the sequence comparison PE's to carry the reversed sequences. We do this by appending the reversed source stream onto the target stream and the reversed target stream onto the source

[6]In this phase, we blur the distinction between the sequence comparison PE and the sequence alignment PE since the sequence alignment PE uses a portion of the logic in the sequence alignment PE, mostly the character comparison logic.

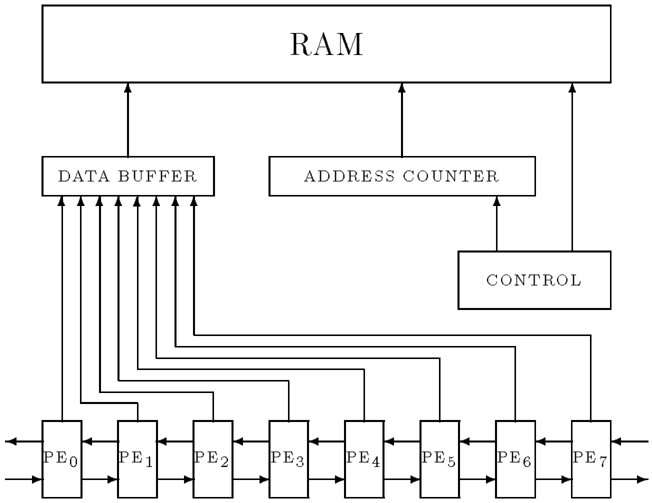

Figure 18: Block diagram of sequence comparison PE's with memory dump logic

stream, yielding the direction reversal we want. We can also reuse the character comparator logic in the alignment phase. This allows us to implement the rest of the alignment PE in only eight CLB's. The alignment PE is diagrammed in Figure 19.

The 1-bit distances stored in RAM encodes only the change between adjacent entries in the DP table. In order to decode the table correctly, we need a correct reference, or seed, value. Since $d_{m,n}$ has just been computed, we can use its value as a seed. Figure 20 illustrates why a seed is needed. The 2×2 table fragment (a) can be decoded to yield either (b) or (c), depending on the least-significant bit of the seed value in the bottom right entry. With the initial seed value given, we can keep an updated seed value at each step by *flipping* the least-significant bit every step, since the adjacent distances change by ± 1. Since we are only concerned with the distances that determine the movement of the marker, the seed needs to be maintained only for the PE that has possession of the marker.

Since the edit distance, $d_{m,n}$ is still in the middle of the array at the beginning of phase two, the distance stream in the sequence comparison PE needs to remain operational in the alignment phase in order to transport the edit distance out of the array. It is interesting to note that since the marker stream travels twice as fast as the distance stream, the alignment data may exit the array before the edit distance does. Therefore, by generating the alignment systolically, we are able to make use of the time that was previously wasted in clearing the distance stream pipeline. In essense, through pipelining we are able to compute an alignment in the same amount of time it takes to compute just the edit distance.

6.3 Control Logic

The operation of the sequence comparison and alignment PE's is controlled by a finite-state controller. The controller generates overlapping two-phase clock signals for the PE's, memory control signals, and clocks for the FIFO queues. The controller is replicated on each chip to reduce clock skews and free I/O pins.

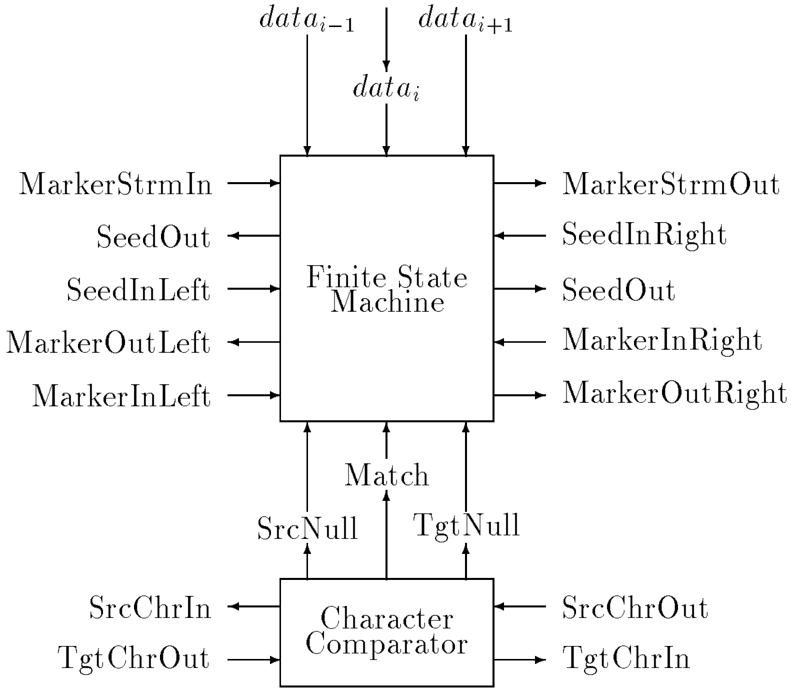

Figure 19: Block diagram of alignment PE

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 11 & 10 \\ 10 & 11 \end{bmatrix} \quad \begin{bmatrix} 10 & 11 \\ 11 & 10 \end{bmatrix}$$

(a) (b) (c)

Figure 20: Possible decodings of a 1-bit table

STATE	CODE	HDSK	NEXT	clka	clkb	addrclk	finrd	foutwt	memcs	memwt
0	0000	X	1	0	0	0	1	1	0	1
1	0001	X	2	0	1	0	1	1	1	1
2	0010	X	3	1	1	0	0	1	1	1
3	0011	X	4	1	0	0	1	1	0	1
4	0100	X	5	0	0	1	1	1	0	1
5	0101	X	6	0	1	0	1	1	1	1
6	0110	X	7	1	1	0	0	1	1	1
7	0111	0	4	1	0	0	1	0	0	1
7	0111	1	8	1	0	0	1	0	0	1
8	1000	X	9	0	0	0	1	1	1	0
9	1001	X	10	0	1	0	1	1	1	0
10	1010	X	11	1	1	0	0	1	1	0
11	1011	X	12	1	0	0	1	0	1	0
12	1100	X	13	0	0	1	1	1	1	0
13	1101	X	14	0	1	0	1	1	1	0
14	1110	X	15	1	1	0	0	1	1	0
15	1111	X	12	1	0	0	1	0	1	0

Figure 21: Truth table for finite state controller

The original sequence comparison design uses a single clock signal. The additional logic for the alignment array complicated the routing of the FPGA and introduced significant clock skews and race conditions. Therefore we redesigned the array to use an two-phase overlapping clocking scheme. As a result, the system clock needs to be four times as fast to produce the same processor clock speed. Since SPLASH has a maximum system clock of 32MHz, and the processor clock is limited to 1MHz in order to communicate with the FIFO's, the increase in system clock frequency presented no problems.

The truth table for the finite state control is presented in Figure 21. States 0 to 7 control the sequence comparison phase. When the FSM receives the `HDSK` signal from the host computer, it moves to state 8 and starts the sequence alignment phase.

7 Results and Analysis

A driver program for the sequence comparison and alignment array has been written in `C` using the `trigger` library. The driver acts as an input/output server, providing the array with input sequence and initialization distance, switching the array from comparison mode to alignment mode, and reading, decoding, and formatting the alignment output.

With 248 PE's onboard, SPLASH can compare sequences up to 123 characters long. For timing purposes, 100 alignments of 123-long sequences were executed on SPLASH. SPLASH took less than 16.7 ms, the resolution of the timing clock. To improve the precision, we repeated timing for 10,000 alignments with the resulting time of 0.30 s. Scaled down, the time for 100 alignments is then 3.0 ms. The time required to initialize SPLASH is 0.47 seconds. We compare our results with the benchmarks from [1] for 100 comparisons of 100-long sequences shown in Figure 22. These benchmarks are for computing the edit distance only. Therefore, the speed-up for sequence alignment should be greater. Even including the initialization time, SPLASH outperforms conventional computers by an order of magnitude.

[7] Timing clock has resolution of 16.7 ms.

System	Time	Speed-Up	Notes
SPLASH	0.020 s[7]	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 22: Benchmark of 100 comparisons of 100-long sequences [1]

8 Acknowledgements

The author would like to thank Daniel Lopresti for his support and guidance; Mort Schultz, Bruce Shapiro, and Anne Barber for the many enlightening and insightful discussions during the summer internship at the NCI Frederick Research Facility; and the National Science Foundation for supporting this work at Brown University through an NSF Graduate Fellowship.

References

[1] D. P. Lopresti, "Rapid Implementation of a Genetic Sequence Comparator Using Field-Programmable Logic Arrays," Invited paper, *Advanced Research in VLSI Conference*, March 1991, Santa Cruz, CA.

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

[3] K. A. Frenkel, "The Human Genome Project and Informatics," *CACM*, November 1991, vol. 34. no. 11, pp. 41-51.

[4] E. S. Lander, R. Langridge, and D. M. Saccocio, "Computing in Molecular Biology: Mapping and Interpreting Biological Information," *Computer*, November 1991, vol. 24, no. 11, pp. 6-13.

[5] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *J. Assn. Comput. Mach.*, vol. 1, pp. 168–173, 1974.

[6] S. B. Needleman and C. D. Wunsch, "A General Method Applicable to the Search for Similarities in the Amino-Acid Sequence of Two Proteins," *Journal of Molecular Biology*, vol. 48, pp. 443-453, 1970.

[7] 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.

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

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

[10] Xilinx, Inc., *The Programmable Gate Array Data Book*, 1991.

[11] M. Gokhale, et. al., "The Logic Description Generator," Supercomputing Research Center Technical Report SRC TR89-011, Supercomputing Research Center, 17100 Science Driver, Bowie, MD 20715.