Journal of VLSI Signal Processing 19, 115–126 (1998)

Kestrel: A Programmable Array for Sequence Analysis

JEFFREY D. HIRSCHBERG, DAVID M. DAHLE, KEVIN KARPLUS,
DON SPECK AND RICHARD HUGHEY
Department of Computer Engineering, Baskin School of Engineering, University of California, Santa Cruz, CA 95064

Received March 31, 1997; Revised July 31, 1997

Abstract. Kestrel is a programmable linear array processor designed for sequence analysis. Among other features, Kestrel includes an 8-bit word, a single-cycle add-and-minimize instruction, a multiplier and efficient communication using shared registers. This paper describes Kestrel's functional units in detail, and examines each of their effects on system performance. With functional prototype chips completed, we will assemble a full single-board Kestrel array, with 512 processing elements on eight chips, in early 1998.

1. Introduction

Kestrel, named after the small, fast falcon found on the Santa Cruz campus of the University of California, is a project to develop a programmable linear array for sequence analysis. The system development is nearly complete. After fabricating three subunits, the multiplier, SRAM, and register bank, we fabricated a 0.8 μm CMOS chip with two processing elements. After our successes with these test chips, we completed the final 64-PE, 0.5 μm chip that is currently in fabrication. A complete system, with 512 PEs in 8 chips, is scheduled for early 1998. Kestrel will be an affordable system; a partially populated Kestrel board could be priced in the low thousands of dollars, while a fully loaded Kestrel board could cost in the low tens of thousands. This paper discusses the motivation for the Kestrel project and the PE architecture, including descriptions of the major architectural components that help Kestrel achieve its goals.

There are three main goals for the Kestrel project. The first is to develop a platform that is well-suited to biological sequence analysis. In projects such as the Human Genome Project, scientists need to analyze large databases containing billions of characters from DNA, RNA, and proteins. Many of the algorithms used in this analysis require scanning large segments of a database. Kestrel has been designed with these algorithms in mind.

The second goal is to provide a programmable architecture. In one sense, this is related to the goal of developing an efficient platform for sequence analysis. There are a great number of sequence analysis algorithms in computational biology, and programmability is required to accommodate these different algorithms within a single system. As new algorithms are developed, Kestrel will be able to execute many of them without the need for redesigning the architecture. Additionally, Kestrel will be able to execute unrelated algorithms suitable for linear arrays.

A third goal is to build a balanced system. The speed of data input and output (I/O) is a primary consideration in the design of any system, especially a massively parallel one. The speed at which the array operates needs to be balanced with the speed of I/O [1]. In the case of sequence analysis, the large databases involved are typically stored on disk. Most disks available today have a maximum sustained transfer rate of 1–5 MB/s. Therefore, we have structured the Kestrel PE architecture and cycle time considering these transfer rates.

The next two sections will briefly review sequence analysis algorithms and architectures. Following this,

we describe and justify the design choices that led to the Kestrel architecture. Finally, we evaluate the effects of these choices on performance.

2. Sequence Analysis Algorithms

Many sequence analysis techniques rely on aligning sequences in the database to a model or to other sequences, often with a variant of the following edit-distance computation. Given two sequences of characters, a and b, this algorithm determines a total cost to transform one sequence into the other through three basic operations: deletion of a character, insertion of a character and mutation of a character (same as deleting a character and inserting a new character). The following dynamic programming equations are used to compute edit distance:

$$c_{i,j} = \min \begin{cases} c_{i-1,j-1} + \text{dist}(a_i, b_j) & \text{match} \\ c_{i-1,j} + \text{dist}(a_i, \phi) & \text{insert} \\ c_{i,j-1} + \text{dist}(\phi, b_j) & \text{delete,} \end{cases}$$

where $\text{dist}(a_i, b_j)$ is the cost of matching a_i to b_j, $\text{dist}(a_i, \phi)$ is the gap cost of not matching a_i to any character in b, and $\text{dist}(\phi, b_j)$ is the gap cost of not matching b_j to any character in a. Edit distance, the number of insertions or deletions required to change one sequence to another, can be calculated by setting $\text{dist}(a_i, \phi) = \text{dist}(\phi, b_j) = 1$, and $\text{dist}(a_i, b_j) = 0$ if $a_i = b_j$ and 2 otherwise. Though not often used in biology, edit distance is a common performance benchmark.

Sequence comparison using affine gap penalties, greatly preferred by biologists, involves three interconnected recurrences of a similar form:

$$\begin{aligned} c^M_{i,j} &= \min\big(c^M_{i-1,j-1} + g^{M\to M}, c^I_{i-1,j-1} + g^{I\to M}, \\ &\qquad c^D_{i-1,j-1} + g^{D\to M}\big) + \text{dist}(a_i, b_j), \\ c^I_{i,j} &= \min\big(c^M_{i-1,j} + g^{M\to I}, c^I_{i-1,j} + g^{I\to I}, \\ &\qquad c^D_{i-1,j} + g^{D\to I}\big) + \text{dist}(a_i, \phi), \\ c^D_{i,j} &= \min\big(c^M_{i,j-1} + g^{M\to D}, c^I_{i,j-1} + g^{I\to D}, \\ &\qquad c^D_{i,j-1} + g^{D\to D}\big) + \text{dist}(\phi, b_j). \end{aligned}$$

Figure 1 shows a dataflow graph for this computation. Variations and restricted forms of this recurrence are used in the classic sequence comparison methods [2–4]. In the most general form (a generalized profile or linear hidden Markov model [5–7]), all transition costs between the three states (in a run of matches, insertions, or deletions) and character costs are position-dependent. Using modulo minimization and the features of the Kestrel architecture allows us to execute the general form represented in Fig. 1 in only 10 single-cycle instructions, without sacrificing programmability.

The dynamic programming calculation easily maps to a linear array of processing elements. A common mapping is to assign one PE to each character of the

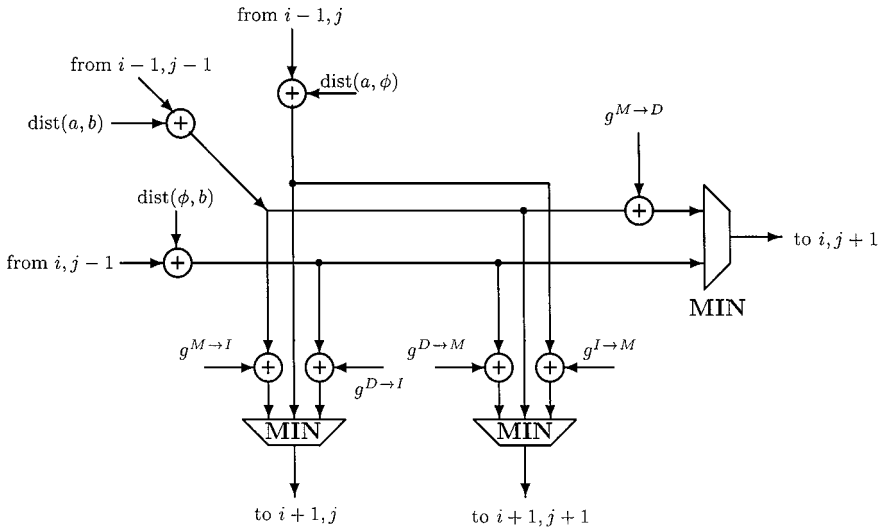

Figure 1. Dataflow for affine gap cost sequence comparison.

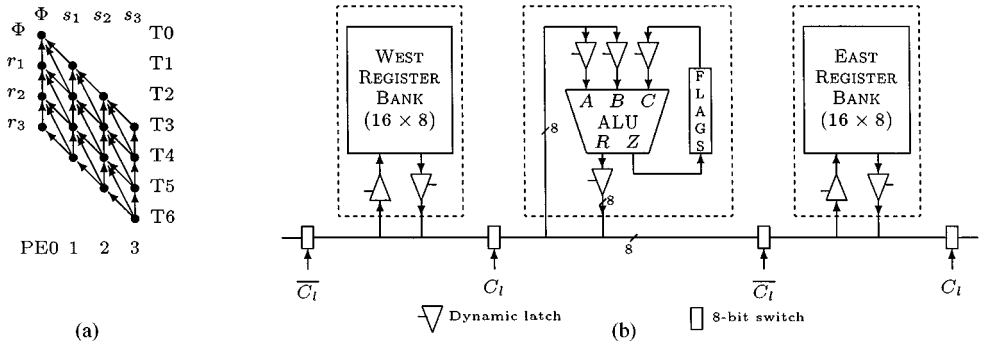

Figure 2. (a) Parallel mapping used by most fine-grain co-processors, and (b) B-SYS processing element [15].

query string, and then to shift the database through the linear chain of PEs (Fig. 2(a)), performing the computation in $2n - 1$ steps on n PEs. Alternate mappings have been used by BioSCAN and by P-NAC [8, 9].

The alignment of two sequences, or a mapping of the characters in one sequence to the characters in the other, is also important. The simplest means of generating a sequence alignment is, for each of the $O(n^2)$ $c_{i,j}$ values, to store the choices made during minimization. The space requirements can be reduced with a divide-and-conquer strategy [10, 11] or our new checkpoint algorithm [12].

In addition to sequence analysis applications, we have also programmed the Kestrel simulator for floating-point division [13], the discrete cosine transform and a neural network application [14].

3. Sequence Analysis Architectures

A number of parallel architectures have been developed for sequence analysis. In addition to architectures specifically designed for sequence analysis, existing programmable sequential and parallel architectures have been used for solving sequence problems.

Workstations are widely available and can be programmed to run all variations of sequence analysis algorithms. Since they are programmable, workstations are suited to other tasks besides sequence analysis. Unfortunately, workstations are slow compared to parallel architectures for sequence analysis, although they can be connected via a network to provide some parallelism.

General-purpose supercomputers are fast and flexible. Like workstations, they can be programmed to run any variation of sequence analysis algorithm, but they are much more efficient than workstations. The limiting factor with supercomputers is cost. They are too expensive to be widely available.

Single-purpose VLSI processors can provide the fastest means of running a particular algorithm. However, they are limited to a single algorithm, and thus cannot supply the flexibility necessary to run the variety of algorithms required for analyzing DNA, RNA, and proteins. P-NAC was the first such machine and computed edit distance over a four-character alphabet [8]. More recent examples, better tuned to the needs of computational biology, some able to perform several variations of Fig. 1 with appropriate parameter setting, include BioSCAN, BISP, FDF, Mercury, and Samba [9, 16–19].

Reconfigurable systems are based on programmable logic such as field-programmable gate arrays (FPGAs) or custom-designed arrays. They are generally slower and have far lower PE densities than single-purpose VLSI but can be faster than supercomputers. They are flexible, but the configuration must be changed for each algorithm, which is generally more complicated than writing new code for a programmable architecture. PAM, Splash, Biocellerator, and Decypher-II are based on FPGAs, while MGAP has its own reconfigurable designs [20–25].

Kestrel is an example of a programmable co-processor. The goal of programmable co-processors is to achieve flexibility while providing performance on a level with single-purpose VLSI. The PIM processor also falls in this category [26]. Kestrel is based on B-SYS, a programmable co-processor developed for sequence analysis (Fig. 2(b)) [15, 27].

In spite of its programmability, B-SYS has several shortcomings, in particular when applied to algorithms other than simple edit distance: (i) its single-port shared registers require three clock cycles to perform an instruction; (ii) there is not enough local storage for

any but the simplest algorithms; (iii) there is no local addressing, required for efficient cost table lookup; (iv) the B-SYS board did not include an instruction sequencer, reducing performance to the time required to transfer each instruction from the host.

For the design of Kestrel, we examined both the shortcomings of B-SYS and the needs of a variety of sequence analysis and other algorithms. We incorporated hardware within the PE to provide for fast execution of these algorithms while maintaining flexibility.

4. Kestrel Architecture

Kestrel is a linear array of 8-bit, single-instruction stream, multiple-data stream (SIMD) processing elements. Linear arrays have modest I/O considerations compared to other types of arrays because data can only enter and exit the array at the ends. The linear design provides a relatively simple method for scaling the array size up or down: chips can be removed from or added to one end of the array as required.

Kestrel's 8-bit datapath provides good balance between PE density and operand size. All 8-bit operations can be done in a single cycle that will be 30 ns or less. Since each PE contains its own SRAM, limiting the width of the datapath is necessary if we want to put 64 PEs on a die of a size that will have a reasonable yield and have enough SRAM locations to handle algorithms such as sequence alignment. Fortunately, 8-bit data is sufficient for most sequence algorithms. In cases where larger data sizes are needed, Kestrel is designed to perform multi-precision operations efficiently. The 8-bit datapath also allows for easy interfacing of the array to external queues, which will hold data to be processed and results generated from running programs.

A block diagram of the Kestrel PE is shown in Fig. 3. The main components of the PE are the arithmetic-logic unit (ALU), static random-access memory (SRAM), comparator, bit shifter and multiplier, which are described in the following sections. Masking logic is also present to enable conditional execution. There are three operands for each instruction: Operand A, Operand B, and Operand C. Operand A and Operand C are two independently selected registers. Operand B can come from the multiplier, the bit shifter, memory data register (MDR), the same register as Operand C or a globally issued immediate value. In addition to these sources, Operand B can be the sign extension of Operand C, the MDR or the high byte of the multiplier result. This is useful for signed multiprecision operations where the two operands differ in length—in sequence comparison, the dist( ) values are commonly of lower precision than the $c_{i,j}$ values.

Depending on the instruction and flags generated in the PE, one of three values is selected as the result. The three possible results are the ALU output, Operand C or MultLo, the low-order byte from the multiplier. The result is always written to a register and can also be written to SRAM and the bit shifter. Instruction execution takes one clock cycle.

4.1. *Systolic Shared Registers*

In systolic algorithms, data moves between PEs as partial results are calculated. Sometimes, values are required by multiple PEs to compute a result. Both of

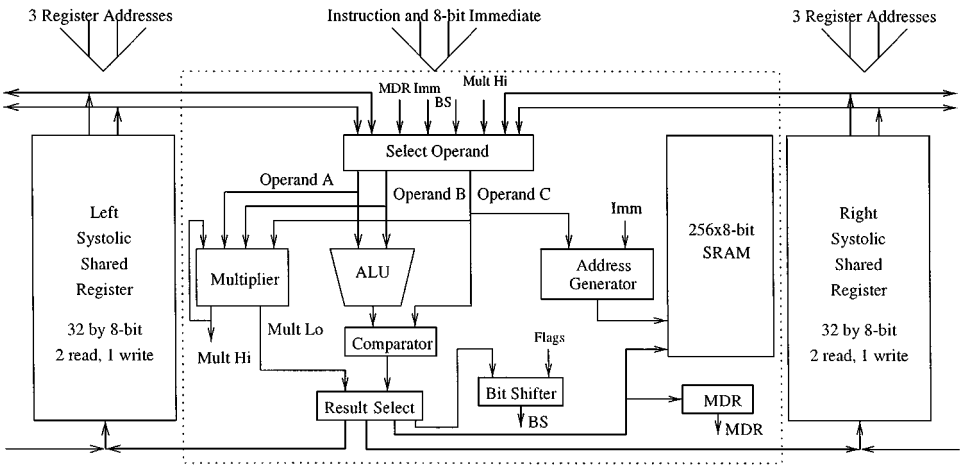

Figure 3. A Kestrel processing element with Shared Systolic Registers.

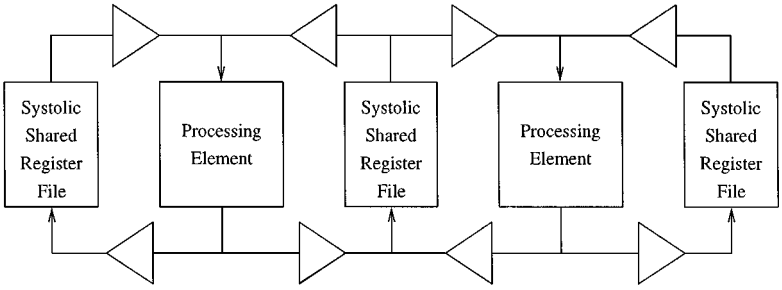

Figure 4. Structure of the Kestrel array.

these are true of the dynamic programming solutions for sequence analysis. The ability to partition problems in this manner is a key factor in making systolic arrays effective parallel solutions for problems. Thus, a good inter-PE communication scheme is essential for any array processor.

Kestrel employs Systolic Shared Registers (SSRs) for inter-PE communication (Fig. 4). The SSRs are composed of 32 8-bit registers. Each SSR has two read ports and one write port. Rather than being local to each PE, as register files usually are, SSRs reside between PEs. This allows neighboring PEs to share a register file. SSRs were invented for the linear B-SYS co-processor and are well-suited to any planar topology [15]. They have some similarity to shared memories [28, 29] or inter-PE queues [30] but provide the tighter coupling between processing elements appropriate for SIMD computation.

The key idea of SSRs is that communication and computation do not require distinct instructions but occur concurrently. When a result is stored after an instruction is executed, communication automatically takes place. The programmer can naturally think about data streaming through the array as values are computed. To pass a value from one PE to the next using SSRs, all that is necessary is to store the value in a register file. For example, if each PE is calculating a value to be used by the PE to its right, then each PE stores its result into the register file on its right. During a subsequent instruction, each PE can read from its left register file to get the previously stored value. Because all addresses for these register files are issued globally, adjacent PEs will never write to the same register bank. SSRs make for an elegant programming model and are a low-overhead solution to the inter-PE communication problem—the cost is one bit (left or right) per register address.

To avoid three operand busses between chips, every chip in a linear SSR machine contains n PEs and $n+1$ SSRs. Thus, at chip boundaries, there are two register banks without an intervening PE. These register banks are coherent: whenever a write occurs, for example, to the left, the value is written both to the leftmost SSR of a chip and off-chip. The rightmost SSR of the adjacent chip then stores the value entering the chip. Inter-chip communication does not involve register reading.

4.2. ALU

Kestrel PEs have an 8-bit arithmetic-logic unit composed of eight identical bit slices, each with a carry-in, a carry-out and a result bit. The carry-in to the least significant slice can come from a bit in the instruction or a latch containing a previous carry-out for multi-precision operations. The carry-out and result from each slice are determined according to the following equations:

$$\text{carry-out} = (\text{propagate AND carry-in}) \text{ OR generate}$$
$$\text{result} = \text{propagate XOR carry-in}$$

Propagate and generate are 4-bit control fields (the 8 bits are compressed to 5 bits in the instruction). One bit from each field is selected for propagate and generate for each slice according to the corresponding bits from Operand A and Operand B. This design is derived from the OM2 [31].

We chose this type of ALU for two reasons. First, it is programmable. The function of the ALU is determined by the propagate and generate fields. Even with the 5-bit encoding, this ALU is capable of all common logic functions and the standard arithmetic functions of addition and subtraction.

Second, the ALU is compact. In the trade-off between functionality and area, this ALU achieves its flexibility without requiring a large amount of area. For sequence analysis, operations will typically be on

8- to 24-bit values. Increasing the datapath to 24 bits would speed up higher precision operations but would also significantly impact PE density. We could have chosen to make a bit-serial design, as well. A bit-serial ALU would be compact, but would have speed limitations. The 8-bit design allows us to achieve good PE density within a chip and a reasonably fast clock rate.

4.3. SRAM

Each PE contains 256 bytes of static random-access memory. Each has an address generator that supports both local and global addressing. Unlike the SSRs, the SRAM contains a single bidirectional port to conserve area. A value can either be read from the SRAM or written to the SRAM once per cycle. Values read from the SRAM are stored in a memory data register (MDR) that can be accessed in subsequent instructions. The MDR can help to alleviate the potential bottleneck of a single port because a value can be read and stored in the MDR prior to use, and used several times without recomputing the address. Thus, the MDR allows us to schedule the use of the single port.

The need for large-capacity storage arises in sequence alignment, in particular. As characters from a database stream through the array to be compared to a target sequence, the costs of the necessary operations (insertion, deletion, mutation) to align the database sequence with the target sequence are calculated. In more sophisticated models, these costs are character dependent, and local addressing allows cost tables to be indexed using the character currently being examined by a PE. The minimum of these three costs is selected as $c_{i,j}$, and flags indicating which was selected are stored in SRAM. When the cost table is complete, the information in the SRAM can be used to determine the minimum-cost set of operations to align the sequences. Storage-efficient variations of this algorithm based on checkpoints enable alignment of sequences up to length $n = 30000$ in 256 memory locations [12].

Although the SSRs could be expanded to handle larger storage requirements, the cost in area and instruction bits would be high. The triple-port registers used in Kestrel take twice the area of the SRAM per bit, and since we specify three register addresses in each instruction, at least 3 bits would have to be added to address larger register files. Since the SRAM is dense and compact, we felt confident making the bulk of the storage a clocked static RAM. The combination of the SSRs and the SRAM provides a memory hierarchy that has high bandwidth (SSRs), large capacity (SRAM) and flexibility.

4.4. Comparator

The comparator compares the output of the ALU with Operand C, and the minimum or maximum can be selected as the result of instruction execution. This is done by subtracting Operand C from the output of the ALU. The subtraction produces three flags, the borrow-out from the subtraction, the most significant bit (msb) of the subtraction, and the true sign of the subtraction. The true sign is the sign that would be produced had the operands been sign-extended to 9 bits, eliminating the possibility of overflow in signed comparisons. Since the machine is SIMD, an overflow bit would be of little use.

The three flags allow for three types of comparison: unsigned comparison (borrow-out), modulo 256 comparison (msb) and signed comparison (true sign). As the cost table is generated when comparing two sequences, it is possible for $c_{i,j}$ values to exceed 8 bits, which would lead to erroneous results. In certain situations, modulo comparison can be used to avoid this problem [32]. The key in modulo comparison is that local differences in costs are small. With m bits of precision for table entries, if delta costs are restricted to being less than 2^{m-1}, then the msb of subtraction determines the minimum value. Use of modulo comparison enables execution of the affine-gap loop in only 10 instructions. This technique cannot be applied to all cost functions; such cases require multiprecision.

In designing Kestrel, we have paid particular attention to multiprecision operations. The comparator is an example of this. Multiprecision comparisons are done top-down by byte. Top-down comparison requires fewer cycles to compare two multiprecision values than the standard method of subtracting one value from the other and then selecting the result based on the borrow-out of the subtraction. When comparing two values of length n bytes, $n - 1$ cycles are saved.

It might appear that the comparator cannot start calculating its result until the ALU output has been determined. If true, this would significantly increase the cycle time for an instruction and be a tremendous penalty for instructions that do not use the comparator. This is not the case, however. Each stage of the borrow chain only depends on the corresponding bits from the ALU and Operand C. So, as soon as the ALU produces each bit of output, that bit is immediately used by the

borrow chain. Thus, the comparator propagates concurrently with the ALU. We estimate the increase to the cycle time compared to a PE without the comparator to be around 15%.

4.5. *Bit Shifter*

The bit shifter is a loadable shift register that can shift left or right. Various flags from the ALU and the comparator can be shifted into either end of the shift register. Also, the result of instruction execution can be loaded into the register, and the bit shifter contents are available as an operand.

The bit shifter serves two purposes within the PE. First, it can be used for data manipulation. Flags can be shifted into either end, and the least-significant bit (lsb) and msb can be used to select the result of instruction execution. Both these operations are useful in the storage and retrieval of minimization costs in sequence alignment.

The second function is as a one-bit-wide stack machine for evaluating conditionals. Because Kestrel is SIMD, PEs must be turned off to perform conditional computation. The bit shifter aids in the fast evaluation of nested conditionals. Figure 5 shows the available condition stack operations for evaluation of conditionals. Each stack operation causes a mask local to each PE to be set based on the contents of the PEs stack. This mask indicates whether or not a PE is active. All of the stack operations can be overlapped with instruction execution, making conditional processing invisible to execution.

A pseudo-code example of conditional execution and the corresponding stack operations (shown in parentheses) appears in Fig. 5. Initially, the stack is cleared, which makes all PEs active. The two *if* statements can be evaluated with the ALU or comparator and a flag, indicating whether the condition was true or not, can be pushed into the bit shifter. The *else* statements can be evaluated by manipulating the conditions placed on the stack during the evaluation of the *ifs*. Note that an *else* operation must be done before the code in the *else* block can be executed. Thus, stack operation (5) is combined with inst0 and stack operation (4) with inst1. Up to eight conditions can be evaluated before the stack becomes full. The contents of the stack can then be stored in SRAM or an SSR and more conditions put on the stack. Even though the generality of this condition mechanism is unnecessary for current sequence comparison methods, it costs only 4 microcode bits and will allow us to easily adapt to more sophisticated algorithms as they are developed.

Pseudo Code	Kestrel Instructions
	(1)
if((X==Y)	ALU== and (2)
&& Z){	ALUtest Z and (6)
if((M==N)	ALU== and (2)
|| P)	ALUtest P and (7)
{inst0}	inst0 and (5)
else{inst1}}	inst1 and (4)
else{inst2}	inst2 and (1)

Operation Key	
Clear stack	(1)
Push flag	(2)
Pop	(3)
Pop, invert msb	(4)
Invert msb	(5)
AND into msb	(6)
OR into msb	(7)
Replace msb	(8)

Figure 5. Kestrel condition stack operations.

4.6. *Multiply-add*

The multiplier performs a variety of functions on Operand A and Operand B to produce a 16-bit result in one cycle. The Operand A and Operand B values can be independently treated as signed or unsigned, enabling signed and unsigned single and multiprecision multiplies and shifts. Adders in the multiplier can be used to add a previously computed high-order byte, Operand C, or both to a newly computed low-order byte for multiprecision multiplies.

Although the multiplier is not needed for standard sequence analysis applications, it is needed for one type of hidden Markov model training. The multiplier also provides an easy way of doing multibit and multiword shifts and aids multiplicative division methods [13]. The multiplier provides significant speed up over the ALU for these functions with a relatively low design cost and increase in PE area of about 18%. It also allowed us to remove less-intuitive hardware for performing shifts, multiply-add and division steps.

4.7. *System Architecture*

The Kestrel system consists of two main elements: the PE array and the array controller. The basic system design is shown in Fig. 6. The array controller will be responsible for issuing instructions to the PE array and for controlling communication with the host system. Since most systolic programs consist mainly of loops, simple loop control will be provided in the controller. The controller will issue instructions from a program memory of 32 K instructions. Two queues of 8 KB will be used to move data in and out of the array. Since

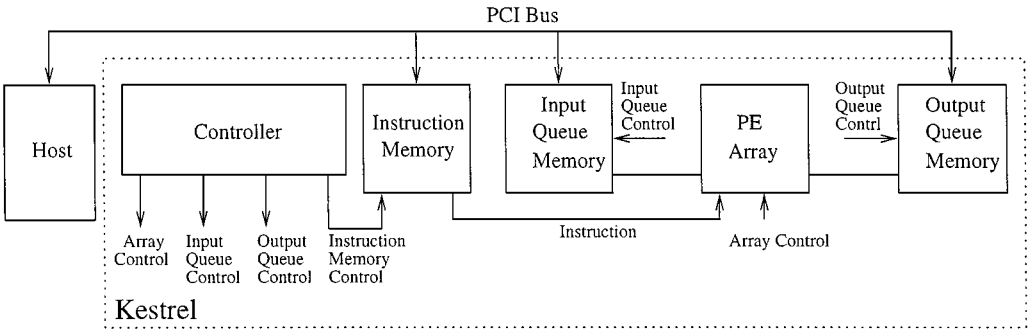

Figure 6. Kestrel system design.

most data will be coming from a disk, the queues should be large enough to accommodate one to two tracks of the disk to make accesses efficient. These queues will connect with the host interface and be managed by the controller. As the queues fill up or empty the controller will stop the array and request service from the host. To achieve the necessary speed to match the array, the controller will make use of the most recent (fastest) FPGA chips.

A Kestrel processing element requires a global 54-bit instruction each clock cycle. Approximately half of the PE's instruction is dedicated to specifying operands; the three registers and immediate require 26 bits. The rest of the instruction is dedicated to controlling the Kestrel PE's many functions. The instructions are well encoded but are easy to decode because they have a fixed length and use a fixed-field encoding. The Kestrel board uses 44 bits (for a combined instruction size of 96 bits) to control input to and output from the array, as well as to provide 0-cycle branches and jumps, loop counter and return stacks, and other control functions.

5. Performance

We have designed a simulator for Kestrel that runs on a MasPar computer. The simulator consists of a simple controller model for issuing instructions to the simulated Kestrel array. The size of the array being simulated can be controlled through a command-line switch. The simulator has proved useful for evaluating our design decisions.

Our timing estimates are based on a 33 MHz clock. The 2-PE test chip was fabricated in a 0.8 μm CMOS process. It was received April, 1997, and was fully functional. The 22 chips without fabrication defects (from a lot of 25 chips) all work correctly with a 35 ns cycle time. Because the 64-PE chips are being fabricated in a faster 0.5 μm process, a 33 MHz chip cycle should be easily attained. Current simulations of the system controller indicate that our 33 MHz clock estimates may be conservative.

Table 1 shows several of the applications we have examined and the effects of architectural components on running times for the applications. The toy edit distance calculation can be performed at 11 million iterations per second (with a 33 MHz clock), well above disk rates. Affine cost functions able to use modulo costs can be evaluated at disk speeds. The 16- and 24-bit affine cost instruction counts are based on a common variant that detects similarities between subsequences of two sequences [4]. These two methods require about 1 MB/s disk rates. We have also examined integer arithmetic and single precision (SP) and double precision (DP) floating-point arithmetic [13]. The multiplier-less numbers for multiplication and division are rough estimates only: if Kestrel did not have a multiplier, the processor would be enhanced to reduce these penalties.

The last column of the table evaluates an alternative choice of word size: 16 bits rather than 8. The values in this column (for selected applications) indicate the length cell programs would be if Kestrel were 16 bits wide instead of 8. The program lengths are multiplied by two to reflect the cost of lower PE density—although multiplier area quadruples, local control area would only increase slightly. There are hidden costs to the 16-bit design: 16 additional data pins to each chip, a slower clock speed, a higher I/O rate, and possibly a wider instruction field to enable 16-bit immediates. On all but the multiplication and division algorithms, a bit-serial machine would perform similarly to the 8-bit Kestrel without a comparator (with an 8 times slowdown but a somewhat less than 8 times increase in PE density, and possibly a higher clock rate if chip-to-chip communication could be sped up).

Table 1. Kestrel cell program times and extra time required with loss of each feature.

Application	Rate (M/s)		Kestrel	Instruction penalty without				16-bit
	Cell	Array	instr.	Compare	SRAM	Bit shift	Mult.	cell × 2
Edit distance	11.0	5600	3	66%	—	—	—	6
Modulo affine	3.0	1500	11	54%	180%	—	—	22
16-bit affine	1.7	870	19	95%	100%	—	—	12
24-bit affine	1.1	560	31	58%	64%	—	—	38
24-bit alignment	0.8	410	40	45%	∞	26%	—	56
16-bit multiplication	6.6	3400	5	—	—	—	1900%	2
16-bit division	0.6	300	57	12%	—	9%	125%	—
SP multiplication	0.9	460	36	—	—	—	730%	—
DP multiplication	0.3	150	102	—	—	—	1000%	—
SP division	0.5	260	64	11%	—	8%	500%	—
DP division	0.2	100	150	7%	—	5%	830%	—
Portion of area				4%	48%	6%	18%	

Figure 7 shows real or estimated protein database search performance using an affine costs model on several machines. The Kestrel curve is revised from an earlier version of this diagram [33] that assumed 1024 processing elements—with the decision to use a PCI board, we do not expect to fit 16 Kestrel chips on a single board. The comparison machines include BioSCAN (single purpose, variant method) [9], BISP (single purpose with variations, not completed) [16], a 15-board Decypher-II system (FPGA) [24], a 5-board FDF system (single purpose) [17], the 3-board SAMBA system (single purpose with variations) [19], 1- and 2-board Mercury systems (single purpose with variations) [18], Biocellerator (FPGA) [23], a 1-board

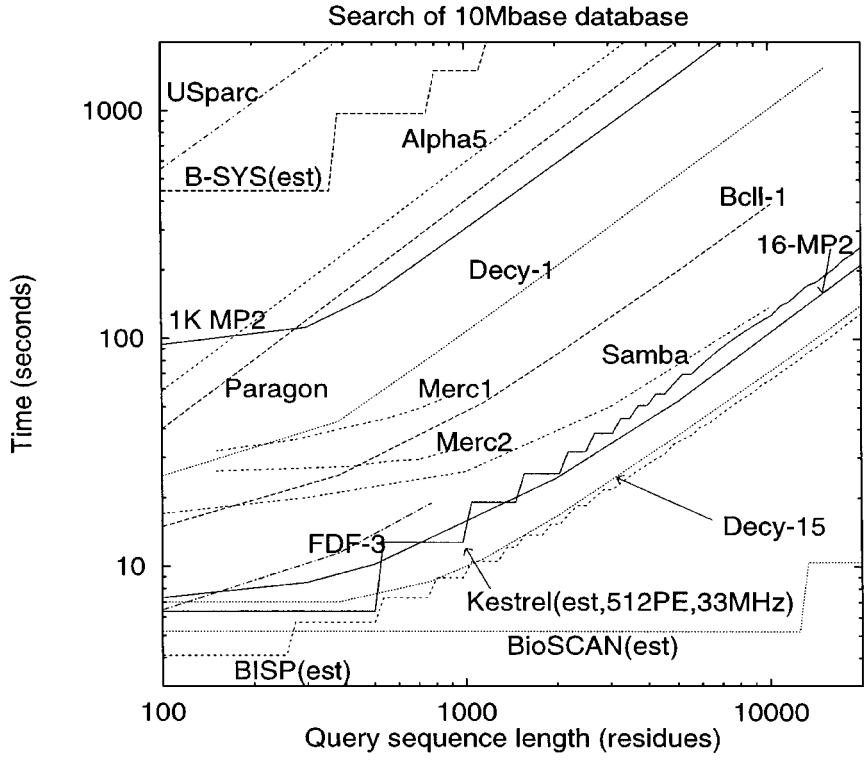

Figure 7. Database search time (real or estimated) as a function of protein query length.

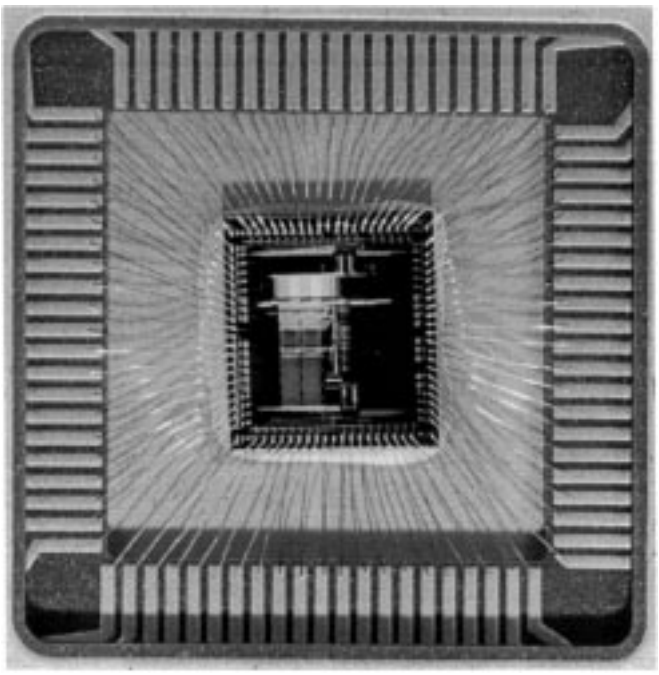

Figure 8. The Kestrel test chip.

Decypher-II, a 32-node Paragon [34], a 1024-PE MasPar, 5 DEC Alpha 3000 workstations [34], B-SYS [15], and a Sun Ultrasparc 140. The Kestrel system is expected to be unique in its ability to pack high performance and programmability into a single-board system.

6. Conclusions

The Kestrel design is motivated by the high throughput requirements of modern computational biology. The processing elements are designed with an eye toward the most common sequence analysis methods: the simple 8-bit PE can add two values, compare the sum to a third value, select the minimum while saving the selector bit, and perform a local memory read all in the same cycle. Special instructions such as this one enable Kestrel to keep reasonable pace with both disk speeds and single-purpose sequence analysis co-processors, yet still be programmable. For example, we expect Kestrel to perform 16-bit affine cost sequence at a similar speed to a 16 384-PE MasPar MP-2 (4 years old, but still a common sequence comparison workhorse) [35], as well as to 16-board FPGA and 3-board custom sequence analysis engines. The simplicity of BioSCAN's differing algorithm (summing diagonals rather than performing dynamic programming) enables high density chips and impressive performance, but the class of algorithm is not as sensitive as the more common Smith and Waterman method [36].

In spite of this tuning, however, the processing elements remain general-purpose. Many algorithms beyond sequence comparison can make use of top-down minimization, bit manipulation, and most any of the features included in Kestrel. Thus, the processing element and array is suited for new sequence analysis methods as well as other applications such as image compression and neural network computation.

A photograph of one of the working 2-PE chips is shown in Fig. 8. In the chip, which is pin-bound, the two processing elements are side by side. The SRAM is at the bottom, followed by the multiplier, the ALU, and the half-width register bank. The extra shadow register bank is to the right, while global instruction decoders and register address decoders are to the left and right of the two processing elements. To aid system development, the 0.8 μm 2-PE chips and the 0.5 μm, 1.4 million transistor 64-PE chips (due back from fabrication in August, 1997) have the same pin assignments.

Acknowledgments

We gratefully acknowledge the contributions of Elizabeth Avila (FP algorithms), Leslie Grate (architecture discussions), J Alicia Grice (sequence alignment), Hansjörg Keller (DCT and neural networks), Jennifer

Leech (runtime environment), Justin Meyer (runtime environment), Salem Osama (simulation), Eric Rice (division), and Doug Williams (SRAM design) to the Kestrel project. Inclusion of the bit-shifter was aided by work with Don Roberts on the MISC architecture [37]. Kestrel was initially funded by the UCSC Division of Natural Sciences, and is currently funded by NSF grant MIP-9423985 and its REU supplement.

More information on Kestrel can be found at `http://www.cse.ucsc.edu/research/kestrel/kestrel.html`.

References

1. H.T. Kung, "Why systolic architectures?" *Computer*, pp. 37–46, 1982.
2. S.B. Needleman and C.D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequences of two proteins," *Journal of Mol. Biol.*, Vol. 48, pp. 443–453, 1970.
3. P.H. Sellers, "On the theory and computation of evolutionary distances," *SIAM J. Appl. Math.*, Vol. 26, pp. 787–793, 1974.
4. T.F. Smith and M.S. Waterman, "Identification of common molecular subsequences," *Journal of Mol. Biol.*, Vol. 147, pp. 195–197, 1981.
5. M. Gribskov, R. Lüthy, and D. Eisenberg, "Profile analysis," *Methods in Enzymology*, Vol. 183, pp. 146–159, 1990.
6. A. Krogh, M. Brown, I.S. Mian, K. Sjölander, and D. Haussler, "Hidden Markov models in computational biology: Applications to protein modeling," *Journal of Mol. Biol.*, Vol. 235, pp. 1501–1531, 1994.
7. R. Hughey and A. Krogh, "Hidden Markov models for sequence analysis: Extension and analysis of the basic method," *CABIOS*, Vol. 12, No. 2, pp. 95–107, 1996.
8. D.P. Lopresti, "P-NAC: A systolic array for comparing nucleic acid sequences," *Computer*, Vol. 20, pp. 98–99, 1987.
9. R.K. Singh, S.G. Tell, C.T. White, D. Hoffman, V.L. Chi, and B.W. Erickson, "A scalable systolic multiprocessor system for biosequence similarity analysis," in *Symp. Integrated Systems*, L. Snyder (Ed.), MIT Press, Cambridge, MA, pp. 169–181, 1993.
10. D.S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Communications ACM*, Vol. 18, pp. 341–343, 1975.
11. E.W. Myers and W. Miller, "Optimal alignments in linear space," *CABIOS*, Vol. 4, No. 1, pp. 11–17, 1988.
12. J.A. Grice, R. Hughey, and D. Speck, "Reduced space sequence alignment," *CABIOS*, Vol. 13, No. 1, pp. 45–53, 1997.
13. E. Rice and R. Hughey, "Multiprecision division on an 8-bit processor," in *Proc. of the 13th IEEE Symp. Comp. Arith.* T. Lang, J.-M. Muller, and N. Takagi (Eds.), IEEE Computer Society, pp. 74–81, 1997.
14. D.M. Dahle, J.D. Hirschberg, Kevin Karplus, H. Keller, E. Rice, D. Speck, D.H. Williams, and R. Hughey, "Kestrel: Design of an 8-bit SIMD parallel processor," in *Proc. 17th Conf. on Adv. Research in VLSI*, IEEE Computer Society, 1997.
15. R. Hughey and D.P. Lopresti, "B-SYS: A 470-processor programmable systolic array," in *Proc. of the Int. Conf. on Parallel Process.*, C. Wu (Ed.), CRC Press, Boca Raton, FL, Vol. 1, pp. 580–583, 1991.
16. E. Chow, T. Hunkapiller, J. Peterson, and M.S. Waterman, "Biological information signal processor," in *Proc. of the Int. Conf. on Application Specific Array Process.*, M. Valero et al. (Eds.), IEEE Computer Society, Los Alamitos, CA, pp. 144–160, 1991.
17. L. Roberts, "New chip may speed genome analysis," *Science*, Vol. 244, pp. 655–656, 1989.
18. D. Brutlag, J.-P. Deautricourt, and J. Griffin, Personal communication, 1995.
19. P. Guerdoux-Jamet and D. Lavenier, "SAMBA: Hardware accelerator for biological sequence comparison," *CABIOS*, Vol. 13, No. 6, pp. 609–615, 1997.
20. J.E. Vuillemin, P. Bertin, D. Roncin, M. Shand, H.H. Touati, and P. Boucard, "Programmable active memories: Reconfigurable systems come of age," *IEEE Trans. VLSI Systems*, Vol. 4, No. 1, pp. 56–69, 1996.
21. M. Gokhale, W. Holmes, A. Kosper, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti, "Building and using a highly parallel programmable logic array," *Computer*, Vol. 24, pp. 81–89, 1991.
22. D.T. Hoang, "Searching genetic databases on Splash 2," in *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, D.A. Buell and K.L. Pocek (Eds.), IEEE Computer Society, Los Alamitos, CA, pp. 185–191, 1993.
23. Compugen Ltd., "Biocellerator information package." Obtained from `compugen@datasrv.co.il`, 1994.
24. Time Logic Inc., "Decypher II product literature." Incline Village, NV, `http://www.timelogic.com`, 1996.
25. M. Borah, R.S. Bajwa, S. Hannenhalli, and M.J. Irwin, "A SIMD solution to the sequence comparison problem on the MGAP," in *Proc. of the Int. Conf. on Application Specific Array Process.*, P. Capello et al. (Eds.), IEEE Computer Society, Los Alamitos, CA, pp. 336–345, 1994.
26. M. Gokhale et al., "Processing in memory: The Terasys massively parallel PIM array," *Computer*, Vol. 28, pp. 23–31, 1995.
27. R. Hughey, "Programming systolic arrays," in *Proc. of the Int. Conf. on Application Specific Array Process.*, E. Lee and T. Meng (Eds.), IEEE Computer Society, Los Alamitos, CA, pp. 604–618, 1992.
28. N.H. Christ and A.E. Terrano, "A very fast parallel processor," *IEEE Trans. Computers*, Vol. C-33, pp. 344–350, 1984.
29. N. Jagadish, J.M. Kumar, and L.M. Patnaik, "An efficient scheme for interprocessor communication using dual-ported RAMs," *IEEE Micro*, pp. 10–19, 1989.
30. M. Annaratone et al., "The Warp computer: Architecture, implementation and performance," *IEEE Trans. Computers*, Vol. 36, pp. 1523–1537, 1987.
31. C.A. Mead and L.A. Conway, *Introduction to VLSI Systems*. Addison-Wesley Publishing Co., Reading, MA, 1980.
32. R.J. Lipton and D.P. Lopresti, "Delta transformations to simplify VLSI processor arrays for serial dynamic programming," in *Proc. of the Int. Conf. on Parallel Process.*, K. Hwang et al. (Eds.), CRC Press, pp. 917–920, 1986.
33. R. Hughey, "Parallel sequence comparison and alignment," *CABIOS*, Vol. 12, No. 6, pp. 473–479, 1996.
34. W.R. Pearson, Personal communication, 1995.
35. J.R. Nickolls, "The design of the Maspar MP-1: A cost effective massively parallel computer," in *Proc. of COMPCON Spring 1990*, IEEE Computer Society Press, Los Alamitos, CA, pp. 25–28, 1990.
36. W.R. Pearson, "Comparison of methods for searching protein sequence databases," *Protein Science*, Vol. 4, No. 6, pp. 1145–1160, 1995.

37. J.D. Roberts, "MISC: A parallel architecture for AI," Ph.D. thesis, University of California, Santa Cruz, CA, 1995.

Jeffrey D. Hirschberg received the BS degree in Computer Engineering from the University of Washington and the MS degree in Computer Engineering from the University of California, Santa Cruz. After obtaining his degree, he continued with the Kestrel project as a full-time design engineer.

On completion of the prototype Kestrel system, he will find an industry position in hardware design.

David M. Dahle expects to receive the BS degree in the majors of Computer Engineering and Physics from the University of California, Santa Cruz, in early 1998, and to continue on to receive an MS degree in Computer Engineering.

His research interests include VLSI design, semiconductor diodes, and VLSI testing.

Kevin Karplus received the BS degree in Mathematics from Michigan State University and the MS degree in Mathematics and Ph.D. degree in Computer Science from Stanford University. He taught at Cornell University before joining the University of California, Santa Cruz, where he is currently an Associate Professor of Computer Engineering.

His research interests have recently changed from VLSI CAD tools to bioinformatics, in which field he uses statistical methods to predict protein structure.

Don Speck received the BS degree in Engineering and Applied Science from the California Institute of Technology and the MS degree in Computer Engineering from the University of California, Santa Cruz. He spent six years at the California Institute of Technology working on the Mosaic multicomputer project, for which he designed the DRAM, and has been pursuing a Ph.D. at the University of California, Santa Cruz.

His research interests include VLSI circuits, computer architecture, algorithms, and image compression.

Richard Hughey received the BS degree in Mathematics and BA degree in Engineering from Swarthmore College, Pennsylvania, and the Sc.M. and Ph.D. degrees in Computer Science from Brown University. After a brief period at the National Institutes of Health, he joined the faculty of the University of California, Santa Cruz, where he is currently an Associate Professor of Computer Engineering.

His research interests include computational biology, parallel processing, and especially the combination of the two, as embodied in the Kestrel project.

rph@cse.ucsc.edu