A Recursive MISD Architecture for Pattern Matching

Arne Halaas, Børge Svingen, Magnar Nedland, Pål Sætrom, Ola Snøve, Jr., and Olaf René Birkeland

***Abstract*—Many applications require searching for multiple patterns in large data streams for which there is no preprocessed index to rely on for efficient lookups. An multiple instruction stream–single data stream (MISD) VLSI architecture that is based on a recursive divide and conquer approach to pattern matching is proposed. This architecture allows searching for multiple patterns simultaneously. The patterns can be constructed much like regular expressions, and add features such as requiring subpatterns to match in a specific order with some fuzzy distance between them, and the ability to allow errors according to prescribed thresholds, or ranges of such. The current implementation permits up to 127 simultaneous patterns at a clock frequency of 100 MHz, and does 1.024 $\times 10^{11}$ character comparisons per second.**

***Index Terms*—Approximate search, multiple instruction stream–single data stream (MISD), online pattern matching, parallel architecture, VLSI.**

I. Introduction

ONLINE multipattern approximate searching applies to situations where several patterns are to be matched concurrently in data where no persistent index can be built to accommodate efficient lookups. Few algorithms exist for searching multiple complex approximate patterns simultaneously, and most of them are filters that lose efficiency for increasing error levels [1].

Consider a string, $S = s_1 s_2, \ldots, s_m$, where s_i is a character from a finite alphabet Σ of size $|\Sigma| = \sigma$. The problem is to simultaneously search S for the occurrence of a set of predefined patterns, $P = \{p_1, p_2, \ldots, p_n\}$, of varying complexity. The patterns p_j considered in this work are constructed much like regular expressions [2], although with some differences. Notably, these include Hamming distance [3] operations on substrings and arbitrary complex boolean functions on subpatterns. Furthermore, hit lingering is allowed to specify coarser approximations, i.e., "match several patterns if they occur within some distance of each other." Order conditions combined with latency is also an additional feature compared with regular expressions; that is, the user may specify that several patterns are to match in an ordered manner and with a (possibly unspecified) distance separating them. The architecture does not yet support the full language of regular expressions; there are limitations when it comes to nested repeating patterns, that is, repeated patterns that consist of subpatterns that are themselves repeating.

Manuscript received April 11, 2003; revised November 6, 2003.

A. Halaas and B. Svingen are with the Department of Computer and Information Science, Norwegian University of Science and Technology, No-7491 Trondheim, Norway (e-mail: Arne.Halaas@idi.ntnu.no).

M. Nedland, P. Sætrom, O. Snøve, Jr., and O. R. Birkeland, are with the Interagon AS, Medisinsk Teknisk Senter, NO-7489 Trondheim, Norway (e-mail: Olaf.Birkeland@interagon.com).

Digital Object Identifier 10.1109/TVLSI.2004.830918

The VLSI implementation uses 0.20-μm CMOS technology that yields a 114 mm^2 die size, and a total of 11.7 million transistors. The architecture's design clock frequency is 100 MHz. Current applications for the Peripheral Component Interconnect (PCI) card on which the chips are mounted include, but are not limited to, optimal selection of reverse transcriptase polymerase chain reaction (RT-PCR) primers [4], DNA sequence assembly [5], and building libraries of siRNA oligonucleotides for mRNA knock-down experiments [6]. Furthermore, the chip has been used to enhance the practicability of genetic programming in pattern mining [7], [8]: the programs that evolve are patterns that can be evaluated in hardware, thus allowing larger datasets, bigger populations, and longer runs than would otherwise be possible.

A brief review of online multipattern approximate searching, as well as previous attempts to implement flexible pattern matching in hardware, is given in Section II. The required functionality is formally described in Section III, while Section IV presents the proposed MISD architecture. The motivation behind this architecture comes from the recursive definitions in Section III. Section V outlines practical design considerations that have been made, and Section VI compares the proposed architecture to other architectures. Finally, in Section VII, we discuss the architecture and makes suggestions for future work.

II. Previous Work

Several excellent surveys exist on the subject of flexible pattern matching in strings [1], [9]–[11]. Among numerous applications are filters for intrusion detection [12], information retrieval [13], and sequence similarity in biology [14]. Multiple approximate pattern matching is used in machine learning, where the concepts of ensembles [15], and mixtures of experts [16] use collections of patterns to increase the performance of the overall model.

Several hardware approaches for exact and approximate string matching have been proposed [17]–[28]. In [10], parallel comparators, associative memories, cellular arrays, and finite-state automata are listed as the architectures that have been used in hardware string matching solutions. Recently, a lot of attention has been given to the problem of edit distance computing and matching [29]–[34] in the context of sequence retrieval and matching from DNA and protein sequence databases. In [35], a review of relevant parallel hardware for sequence comparison and alignment is given. Note, however, that benchmarking is a difficult task due to significant algorithm discrepancies between the systems.

III. Functionality

The architecture design aims to support a set of query semantics:

1) concatenation of basic strings and patterns;
2) alphanumerical comparisons of simple strings;
3) boolean operations on subexpressions;
4) hamming distance filtering;
5) hit lingering (latency);
6) regular expressions.

A formal query language [36], [37] has been constructed to support the given functionality (details omitted). The outlined query modes will be described in the following.

Def. 1: A string $S = s_1 s_2, \ldots, s_m$ is an ordered set of m characters from a finite alphabet Σ of size $|\Sigma| = \sigma$. The ith prefix of S is the string $S_i = s_1, \ldots, s_i$.

Def. 2: A pattern $P = p_1 p_2, \ldots, p_n$ is the concatenation of n subpatterns that are expressions in some query language. The length of the pattern is given by $|P| = \sum_{j=1}^{n} \pi_j$, where π_j is the maximum number of characters in the string that is matched by the jth subpattern. Multiple patterns are separated by commas. For example, $p_1, p_2, \ldots, p_n$ denotes n separate patterns.

Def. 3: The hit function $H(S_i, p_j)\colon S \times P \rightarrow \{0,1\}$ is a function returning 1 if a pattern p_j matches some suffix of S_i, and 0, otherwise. For instance, if $S_i =$ "regular" and $p_j =$ "ar", then $H(S_i, p_j) = 1$.

Items 1)–6) can now be formalized using Defs. 1, 2, and 3. The concatenation of two patterns requires a hit function

$$H(S_i, p_{j-1}p_j) = H(S_{i-\pi_j}, p_{j-1}) \wedge H(S_i, p_j). \qquad (1)$$

When the subpatterns are the smallest possible constructs (single characters) the hit function represents the exact matching of strings. To illustrate, consider the pattern $P =$ "tg" and the string $S =$ "atg". Matching P on the third prefix of S can then be formalized as $H(\mathrm{atg}, \mathrm{tg}) = H(\mathrm{at}, \mathrm{t}) \wedge H(\mathrm{atg}, \mathrm{g})$.

An alphanumerical comparison is the operation of matching substrings alphabetically or numerically related to a given pattern. Given an alphanumeric operator, $\alpha \in \{\leq, <, >, \geq\}$, the hit function becomes

$$H(S_i, \alpha(p_{j-1}p_j)) = H(S_{i-\pi_j}, \alpha(p_{j-1})) \vee (H(S_{i-\pi_j}, p_{j-1}) \wedge H(S_i, \alpha(p_j))). \qquad (2)$$

Thus, given the pattern $P = \leq$ "tg" and the string $S =$ "agg", the hit operation can be formalized as $H(\mathrm{agg}, \leq \mathrm{tg}) = H(\mathrm{ag}, \leq \mathrm{t}) \vee (H(\mathrm{ag}, \mathrm{t}) \wedge H(\mathrm{agg}, \leq \mathrm{g}))$.

Using this formalism on boolean operators is straightforward, and the hit function yields

$$H(S_i, f(p_1, p_2)) = f(H(S_i, p_1), H(S_i, p_2)) \qquad (3)$$

where f denotes the boolean function. Given the pattern $P =$ "t" | "g", where | denotes the boolean *or* function, and the string $S =$ "g", the hit function becomes $H(\mathrm{g}, \mathrm{t} \mid \mathrm{g}) = (H(\mathrm{g}, \mathrm{t}) \mid H(\mathrm{g}, \mathrm{g}))$.

Let $\eta(p_0, \ldots, p_m, n)$ denote a pattern that requires n of m different patterns to match simultaneously. The hit function can be written

$$H(S_i, \eta(p_1, \ldots, p_m, n)) = \sum_{k=1}^{m} H(S_i, p_k) \geq n.$$

A concatenated pattern version, which requires the matching of n of m subexpressions, becomes

$$H(S_i, \eta(p_1, \ldots, p_m, n)) = \sum_{k=1}^{m} H(S_{i - \sum_{\ell=k+1}^{m} \pi_\ell}, p_k) \geq n. \qquad (4)$$

Note that (4) becomes the familiar Hamming distance threshold when the subpatterns are single characters. As a result, matching the pattern $P = \eta(\text{"tg"}, 1)$ in the substring $S =$ "agg" then becomes $H(\mathrm{agg}, \eta(\mathrm{tg}, 1)) = (H(\mathrm{ag}, \mathrm{t}) + H(\mathrm{agg}, \mathrm{g})) \geq 1$.

Hit lingering allows a match to have a prolonged effect in that it can continue to be reported for some (possibly unlimited) time. That is, the construct requires a pattern, P, to match a prefix S_j within a distance, d, before the current prefix S_i, i.e.,

$$H(S_i, \delta(P, d)) = H(S_j, P) \wedge 0 \leq d \leq i - j. \qquad (5)$$

Thus, when the pattern is $P = \delta(\mathrm{t}, 1)$ and the string is $S = \mathrm{atg}$, then $H(S_3, P) = 1$, because $H(\mathrm{atg}, \delta(\mathrm{t}, 1)) = H(\mathrm{at}, \mathrm{t}) \wedge 0 \leq 1 \leq 3 - 2$.

Furthermore, using the hit function formalism, it is possible to define constructs like *near* (two patterns should occur within some distance of each other) and *before* (one pattern should occur before the other within some distance). The former construct can be written

$$H(S_i, \delta(p_1, d)) \wedge H(S_i, \delta(p_2, d))$$

while the latter becomes

$$H(S_{i-\pi_2}, \delta(p_1, d)) \wedge H(S_i, p_2).$$

The defined hit functions suffice to match regular expressions without Kleene closures since both unions and concatenations are supported. Also, every expression containing a skip $P?$, which is shorthand for $(P|\epsilon)$, can be rewritten into the union of an expression containing P and an expression with P removed.

Repeated patterns, $P^+ = PP^*$, can also be described by the framework introduced in this section. The matching of the concatenation between a repeating pattern, p_2, and a preceding pattern, p_1, is formally expressed by the hit function

$$H(S_i, p_1 p_2^+) = H(S_{i-n\cdot\pi_2}, p_1) \wedge_{k=0}^{n-1} H(S_{i-k\cdot\pi_2}, p_2). \qquad (6)$$

Thus, when $P = \mathrm{tg}^+$ and $S = \mathrm{tgg}$, the hit function is $H(\mathrm{tgg}, \mathrm{tg}^+) = H(\mathrm{t}, \mathrm{t}) \wedge H(\mathrm{tg}, \mathrm{g}) \wedge H(\mathrm{tgg}, \mathrm{g}) = 1$.

The following section describes an architecture for implementing these hit functions in hardware.

IV. MISD Architecture

Recall that (1)–(4) in Section III are recursive. That is, the evaluation of $H(S_i, P)$ requires partitioning P into its atomic components, and combining the individual results using the appropriate functions. The recursion can be visualized by drawing a recursion tree where the result is found by propagating the results from the leaf nodes, i.e., the pattern's atomic components, to the root of the tree. Some of the hit function definitions require that S_i is evaluated in parallel, i.e., (3), while others are strictly sequential, i.e., (1), (2), and (4).

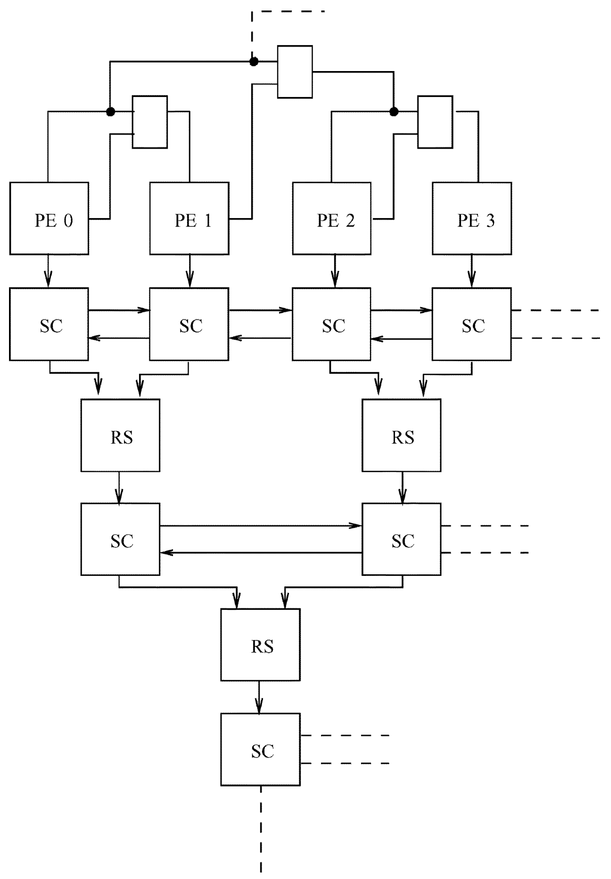


Fig. 1. Search core with data distribution tree (top), processing elements, and result processing nodes (bottom). Distribution nodes receive data in sequence or in parallel. Sequence control (SC) is applied to determine the left and right neighbor's current hit status, while result selection nodes (RS) propagate their results according to their configured hit function.

These observations motivate an architecture with two complete binary trees that are responsible for data distribution and result processing, respectively. The nodes of the former tree should be able to distribute either the data from its parent or from the rightmost leaf node of its left subtree. That is, all subtrees on all levels can receive the data in sequence or in parallel. The leaf nodes performing single character comparisons are processing elements (PEs) that are shared between the trees. These compare their current character, s_i, to a preconfigured value, p_j. (The characters are shifted in by the data distribution tree.) The operations $p_i = s_j$, $p_i \geq s_j$, $p_i \leq s_j$, and $p_i \neq s_j$ are sufficient for the proposed functionality, including alphanumerical comparisons. In addition, they facilitate additional features, such as matching characters that belong to alphanumerical ranges.

The internal nodes of the result processing tree receive their input from their children. Denote the results from a node's left and right child H_ℓ and H_r, respectively. The node's result is given by a dynamically configurable function $H\colon H_\ell \times H_r \to H$. Note that the result is reported as the final hit if the node is the root of the pattern's recursion tree, or propagated to its parent node otherwise. The hit functions $H_\ell = H_r$, $H_\ell > H_r$, $H_\ell \geq H_r$, $H_\ell + H_r$, $H_\ell + H_r \geq c$, $H_\ell + H_r \leq c$, and $H_\ell + H_r = c$ are sufficient to support the outlined functionality of Section III. Here, c is some positive integer value. In addition, any node should have the possibility of being disabled, meaning that it should continuously report 0. To enable alphanumerical comparisons as specified in (2), the equality signal from the leaf nodes is also propagated up the result processing tree. That is, the PEs output $p_i = s_j$, and the internal result processing nodes would propagate $\mathrm{eq}_\ell \wedge \mathrm{eq}_r$ where eq_ℓ and eq_r denotes the equality signal from a node's left and right child, respectively. A conceptual view of the architecture is given in Fig. 1.

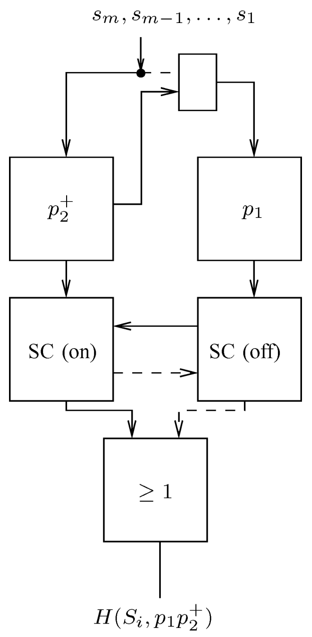


Fig. 2. The tree illustrates how the pattern $p_1p_2^+$ would be implemented in the proposed architecture, using sequence control for the left subtree responsible for matching occurrences of p_2. Note that the bottom node reports the final hit disregarding any input from its right child (dotted line).

Recall from Section III that watching concatenations containing repeating patterns, i.e., $p_1p_2^+$, amounts to knowing if there was a hit for the first part of the expression at a position $n \cdot \pi_2$ characters earlier, as well as $n-1$ consecutive matches for the second part. Consider the construct illustrated in Fig. 2, which exemplifies how the problem may be solved in the proposed double binary tree architecture. The first part of the pattern is matched by the right subtree, which corresponds to the box labeled p_1. This subtree receives data sequentially from its left sibling responsible for matching the second part of the pattern, namely p_2. However, the p_2 part cannot propagate a positive result to its parent node unless either 1) its right sibling responsible for matching p_1 also has a hit or 2) the node itself reported a hit π_2 characters ago. Note that the former condition must be met before entering (possibly) multiple occurrences of the latter situation. Hence, all nodes must receive information from their neighboring node on all levels of the result processing tree. This kind of *sequence control* is actually facilitated in both directions as this, in some cases, allows cheaper implementation of skipping patterns (details omitted). The sequence control signals are illustrated with horizontal arrows between neighboring result processing nodes in Figs. 1 and 2. In addition, a flip-flop chain must be associated with each node. Thus, by feeding the chain with a bit according to its current hit state, the kth bit of the chain represents the hit state k clock cycles earlier.

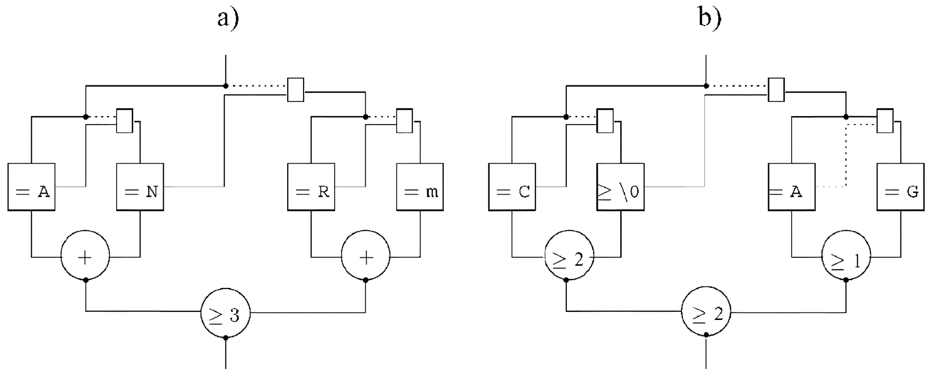


Fig. 3. Illustration of PMC configuration for the queries a) {mRNA:p $>=$ 3}, that is match all strings of length four that match at least three out of four characters in mRNA, and b) (G | A).C, i.e., a G or an A followed by a wild card and a C.

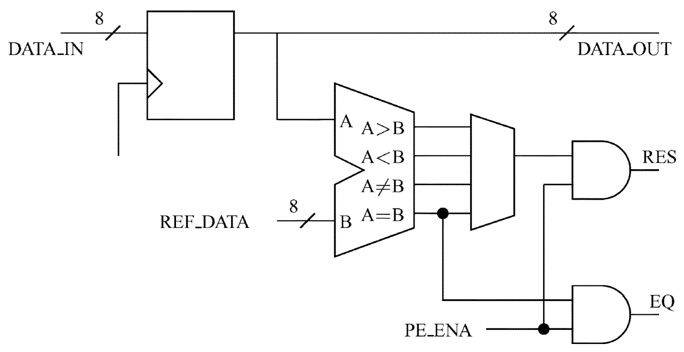


Fig. 4. The PE implementation.

Fig. 3 shows two architecture configurations. The left tree matches all strings of length four with up to one mismatch with the pattern mRNA. The solid lines represent the data path, which is serial between all nodes in this case. The first level result processing nodes transmit $H_\ell + H_r$, while their parent outputs the boolean $H_\ell + H_r \geq 3$. Since the PEs are configured to match the characters of the pattern, the top result processing node will produce the desired result.

Note that the datapath is not sequential for both subtrees in the second example: the rightmost subtree has parallel data distribution, which means that the PEs of this tree receive the same data. As the result processing node gives $H_\ell + H_r \geq 1$ this translates to a hit if either A or a G is currently shifted in. The left subtree has sequential data distribution as in the previous example, and is responsible for matching the subpattern .C : the wildcard matching PE is simply configured to match every character with a byte value greater than or equal to zero.

V. Implementation

The architecture may be implemented with as many PEs as are allowed by practical implementation constraints. More PEs makes it possible to evaluate more expressions, each representing more complex patterns, in parallel. Area constraints dictate the maximum number of levels in the architecture to 10. Hence, there are 1024 PEs on the level that is shared between the trees (level 0). The implementation of the PE's is as illustrated in Fig. 4. Setting PE_ENA low amounts to disabling the PE, that is both the RES and EQ signals are 0 no matter the result of the current comparison. Data is shifted in from the distribution tree, and the ALU performs the four comparisons between the currently held character and the preconfigured reference character. Note that the bus width is 8 bits, which limits the size of applicable alphabets to σ =256 if the patterns are to use a single PE per character comparison (larger alphabets can be used at the expense of more PEs).

The parallel and sequential data distribution of the upper binary tree can easily be accomplished using 2:1 multiplexers on all levels as illustrated in Section IV. However, continuously dividing into eight subtrees until all subtrees on the lowest level contain two PEs, results in an implementation using a series of $2:1$, $3:1$, and $4:1$ multiplexers. The longest chain will be for the rightmost PE whose data path consists of tree $4:1$, and one $2:1$ multiplexer. The data from the stream is thus fed fed to the root node of the data distribution tree, and simultaneously distributed to several collections of PEs belonging to separate queries. The implementation is shown in Fig. 5.

Future implementation involving more PEs will require wider or more levels of multiplexers. For example, doubling the current number of PEs could be implemented with only one additional multiplexer level. However, by using the implementation in Fig. 5, another level of multiplexers is only needed when the number of PEs is equal to 2^{3n+1}, $n \in \{0, 1, \ldots\}$. This means that the current data distribution tree supports up to 4096 PEs without the addition of a new multiplexer level.

For increased clock rates, the data distribution can be pipelined at the expense of pipelining registers as well as delay chains to align data properly. The data distribution is not the timing critical path in the current design. The parallel data distribution can run at the design speed without pipelining. The ability for both serial and parallel data distribution, as well as combinations of these, is crucial for efficient use of all PEs.

Patterns that assume complexities beyond the atomic components and the simplest of combinations, are usually more significant in applications. Hence, the number of bits available for specifying latency increases with tree level. However, all levels are able to set all bits, which yields infinite latency. The number of bits available are $2, 4, 8, 8, 16, 16, \ldots, 16$ for levels $0, 1, \ldots, 10$, respectively. Due to strict area constraints in the implementation it was decided to allow sequence control at levels 0, 2, 4, 6, and 8 only. The length of the flip-flop chains are set to the maximum length of the data path for any given subtree, thus restricted to 1, 4, 16, 64, and 256 at the

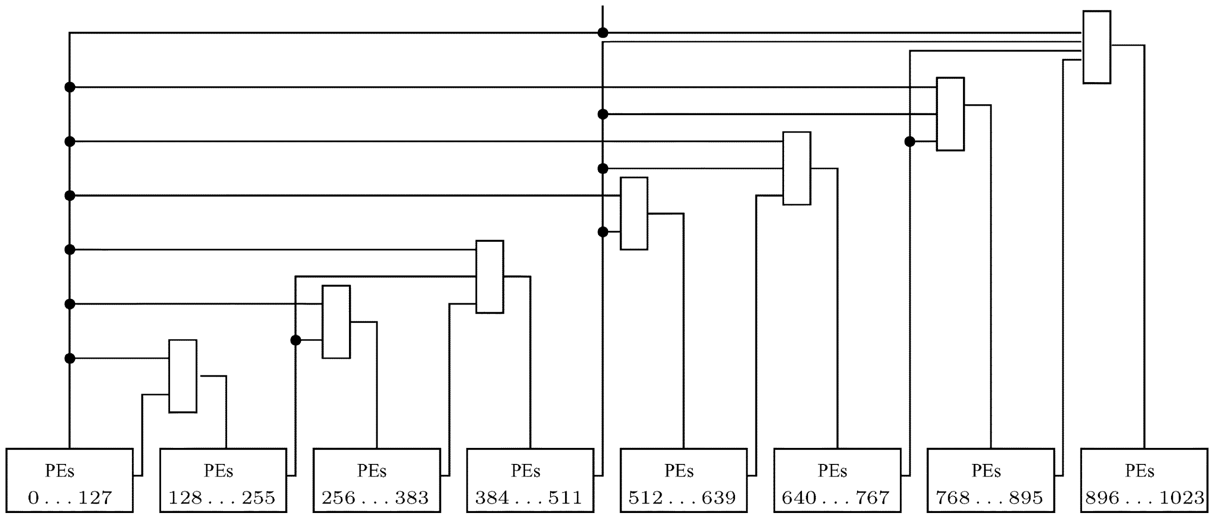


Fig. 5. Data distribution tree implementation using 2 : 1, 3 : 1, and 4 : 1 multiplexers.

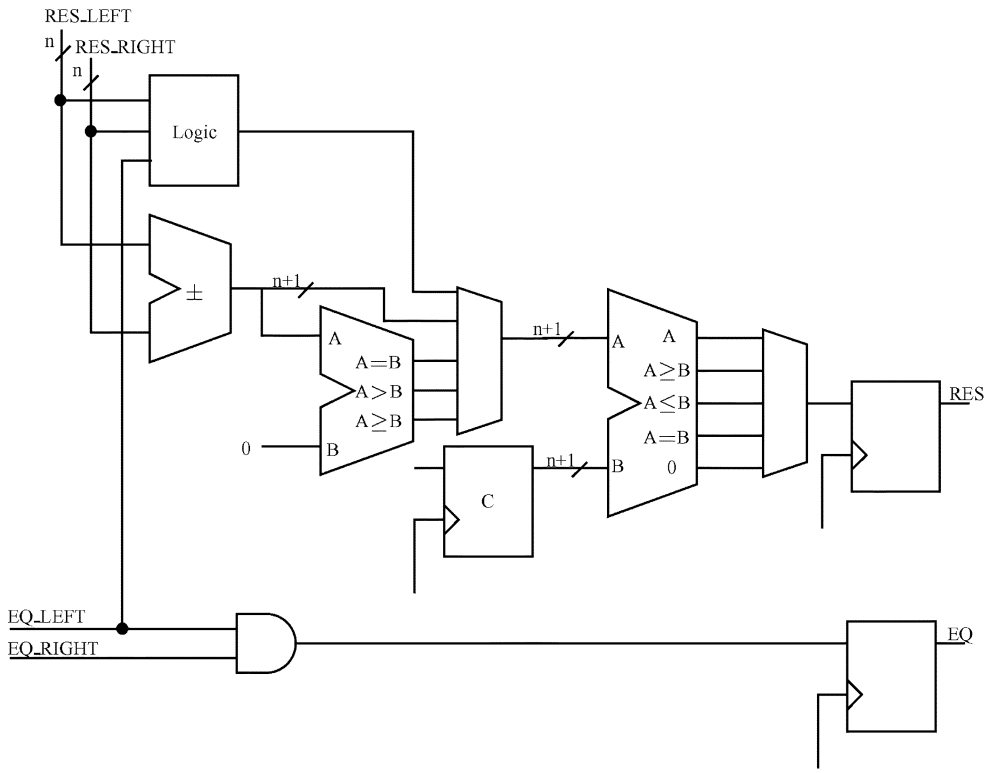


Fig. 6. Core functionality of the result processing nodes, excluding the advanced functionality of sequence control and latency.

respective levels. Note, however, that this does not restrict the functionality, as there is no need to extend the knowledge of a hit beyond the maximum datapath of a given subtree. The core functionality of the result processing nodes, excluding the advanced sequence control and latency features, is implemented as shown in Fig. 6. The first ALU calculates the sum or difference of the input from its children's results. If taking the difference, an optional comparison with zero identifies the subtree with most hits. Alternatively, an alphanumeric comparison is performed at this stage, involving the equality signal from the most significant (left) subtree. The first multiplexer thus receives either a scalar value (when summarizing hits) from the left and right subtrees, or a boolean value encoded in the least significant bit (LSB). This is optionally compared to a configurable constant, c. Furthermore, the combination of the two ALUs can implement any binary function when receiving boolean inputs.

Fig. 7 shows the implementation of sequence control with a flip-flop chain for matching repeating patterns, accompanied by a latency unit that enables holding a result for a predefined number of clock cycles. Results are found as a combination of the output from the connected result processing node (see Fig. 6) and its left and right neighbors. The final result is fed back to the same neighbors. Note that results are either flowing left- or rightward thus avoiding asynchronous loops.

On a practical note, the architecture needs to report hits to the host system for postprocessing. This is accomplished by assigning local *hit managers* to the result processing nodes. The

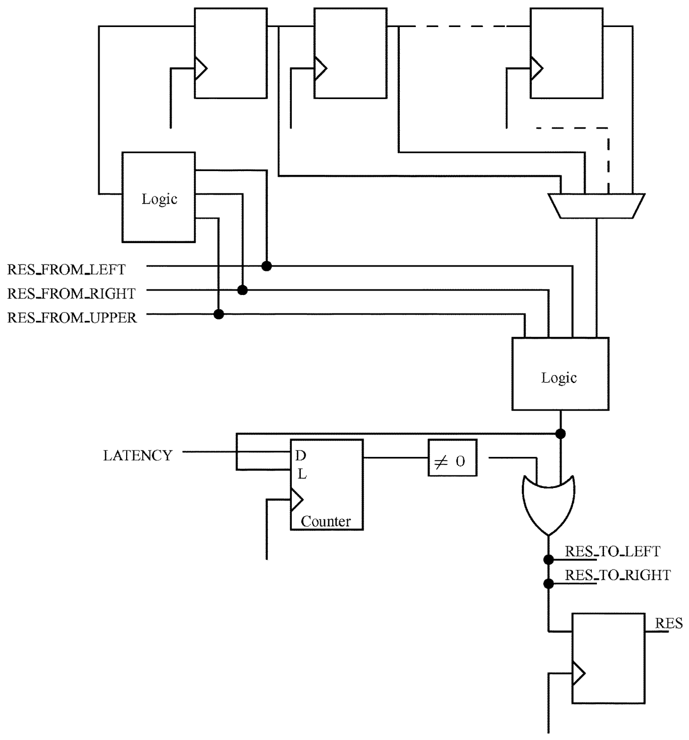


Fig. 7. Sequence control and latency unit implementation.

hit managers report the address of hits back to host memory using direct memory access (DMA). Alternatively, the hit manager can report the document number where a hit occurred. This is accomplished with the help of a separate unit called the *document manager*, which scans the datastream before it reaches the data distribution tree. The document manager counts the number of documents in the stream by looking for a preconfigured character sequence. The current implementation supports a document separator sequence of up to four characters. Additionally, to prevent the chip from flooding the host system with results, the hit managers can be configured to only report one result from each document or within an address range.

It is assumed that queries requiring very few PEs occur infrequently, thus, limiting the need for area consuming hit managers at lower levels. Hence, hit managers are only assigned to the result processing nodes on level four or higher in the result processing tree. Consequently, the architecture is limited to handling a maximum of 127 parallel queries, where each query has a minimum of 16 PEs available.

For easy support of features such as case insensitive searches, a byte remapping table has been included in the chip. Before the search data is passed to the document manager and data distribution tree, each byte in the stream is remapped by reading its value from the table. Thus case insensitive searches can be performed by mapping "a"-"z" to "A"-"Z", and ensuring that the PEs only matches upper case characters.

The chip is programmed through a set of 32 bit registers, which can be divided into configuration and control registers. The configuration registers consist of one register for each PE, two registers for each node in the twin binary trees (the nodes in the data distribution tree are configured using a single bit for each node, where 0 means serial distribution and 1 means

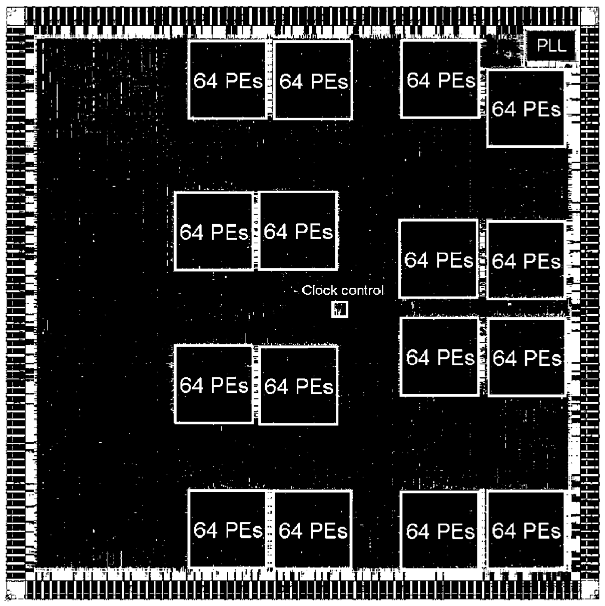


Fig. 8. Chip layout.

parallel), and two registers for each hit manager. The document manager is programmed using two registers, while the remapping table uses 64 registers. The control registers are used for instance to start or stop searches, or to read general status information. As a frontend to the chip, we have developed a compiler for translating queries in the query language [36], [37] to register configurations. This compiler is part of a larger API, which can be used for integrating the chip in different search applications.

Fig. 8 shows a plot of the physical layout on the chip, implemented in standard cell technology. Groups of 64 PEs were created, and separately routed. Sixteen such macros are included in the design, and can be seen as square regions in the layout. The remaining logic was placed and routed as a flat design.

VI. Comparisons

The architecture proposed in this article is primarily designed for search problems where one would rapidly find the occurrences of one or more patterns in a large database. This means that once a query is supplied by a user, both the delay before the query is evaluated and the time needed to search the database should be as small as possible. An important factor for reaching the former goal, is that the architecture is able to evaluate queries in parallel whenever possible. In other words, as few PEs as possible should remain idle during a search. Our architecture accomplishes this through the use of a flexible data distribution tree and by associating hit managers with the nodes in the result gathering tree. This gives us the possibility to pack queries for parallel evaluation using a simple Huffman procedure. To the best of our knowledge, no other proposed hardware design can achieve this ease and flexibility in parallel query evaluation.

The proposed architecture also supports a wider range of functionality than many other recently proposed architectures, whose main features are edit distance computations [29]–[34]. On the other hand, general edit distance computations are not supported by our architecture. Instead, the more limited

TABLE I
TECHNICAL DATA FOR DIFFERENT ASICs (ADAPTED FROM [33])

System	Number of PEs	Number of Transistors	Silicon Area	Maximum Clock Rate	Max. Character Comparisons	Technology
Proposed	1024	1.17×10^7	114 mm^2	100 MHz	1.024×10^{11} ch/s	0.20 μm CMOS
[33]	64	3.4×10^5	53 mm^2	132 MHz	8.448×10^9 ch/s	0.6 μm CMOS
Kestrel [30], [31]	64	1.4×10^6	60 mm^2	33 MHz	7.04×10^8 ch/s	0.5 μm CMOS
CASM [29]	7	2.047×10^4	30.7 mm^2	40 MHz	2.8×10^8 ch/s	2 μm CMOS
SSE [20], [21]	512	2.176×10^5	110 mm^2	10 MHz	5.12×10^9 ch/s	1.6 μm CMOS

Hamming distance filtering functionality can be used, which is identical to using edit distances without insertions and deletions. This lack of functionality may be offset by the fact that our architecture makes it possible to create expressions where mismatches at different positions in the string have different weights. This functionality is important in several potential applications (see for instance, [38]).

To summarize, Table I compares the technical data of our solution to other recently published text processors. The comparison also includes an older architecture [20], [21], which, like our architecture, does not provide the full edit distance comparisons of the other architectures in Table I. The string–search engine (SSE) of [20], [21] combines an associative memory with finite state automata and is mainly used for (near)[1] exact string searches with variable length wildcards.

Initial tests using our ASIC implementation mounted on PCI cards with SDRAM for holding the dataset, have shown that the chip can achieve its desired search rate of 100 MB/s (data not shown).

VII. DISCUSSION

A functionality for performing approximate pattern matching has been outlined and formally described in terms of hit functions. An MISD VLSI architecture has been proposed. The hardware is suitable for searching an online data stream using queries expressed in languages that support the described functionality.

The current implementation runs at a clock frequency of 100 MHz, and is able to search for up to 127 queries, depending on complexity, at the rate of 100 MB/s. The implementation has 1024 processing elements, so a single chip is able to perform up to 1.024×10^{11} character comparisons per second.

Work has been undertaken to implement a data processing system based on the VLSI chip presented here. Several possible solutions are being investigated, with the primary focus being integration of multiple chips on PCI cards with onboard SDRAM as the primary data source. An implementation with 16 chips on a single PCI card, each chip having 128 MB of dedicated SDRAM, is now in the process of being tested. A standard PC fitted with four of these cards is theoretically able to perform about 7×10^{12} character comparisons per second.

REFERENCES

[1] G. Navarro and M. Raffinot, *Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences*. Cambridge, U.K.: Cambridge Univ. Press, 2002.

[2] J. Friedl, *Mastering Regular Expressions*, 2nd ed. Cambridge, MA: O'Reilly, 2002.

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

[4] W. Freeman, S. Walker, and K. Vrana, "Quantitative RT-PCR: pitfalls and potential," *Biotechniques*, vol. 26, no. 1, pp. 112–122, 1999.

[5] S. Smith, W. Welch, A. Jakimcius, T. Dahlberg, E. Preston, and D. Dyke, "High throughput DNA sequencing using an automated electrophoresis analysis system and a novel sequence assembly program," *Biotechniques*, vol. 14, no. 6, pp. 1014–1018, 1993.

[6] M. McManus and P. Sharp, "Gene silencing in mammals by small interfering RNAs," *Nature Rev. Genetics*, vol. 3, no. 10, pp. 737–747, 2002.

[7] M. L. Hetland and P. Sætrom, "Temporal rule discovery using genetic programming and specialized hardware," in *Proc. 4th Int. Conf. Recent Advances in Soft Computing*, 2002.

[8] P. Sætrom and M. L. Hetland, "Unsupervised temporal rule mining with genetic programming and specialized hardware," in *Proc. 2003 Int. Conf. Machine Learning and Applications (ICMLA'03)*, 2003, pp. 145–151.

[9] G. Navarro, "A guided tour to approximate string matching," *ACM Comput. Surveys*, vol. 33, no. 1, pp. 31–88, 2001.

[10] G. Stephen, *String Searching Algorithms*, Singapore: World Scientific, 1994.

[11] P. Michailidis and K. Margaritis, "On-line approximate string searching algorithms: Survey and experimental results," *Int. J. Comput. Math.*, vol. 79, no. 8, pp. 867–888, 2002.

[12] J. Kuri, G. Navarro, L. Mé, and L. Heye, "A pattern matching based filter for audit reduction and fast detection of potential intrusions," in *Proc. 3rd Int. Workshop Recent Advances in Intrusion Detection*, LNCS, 2000, pp. 17–27.

[13] N. Belkin and W. Croft, "Information filtering and information retrieval – 2 sides of the same coin?," *Commun. ACM*, vol. 35, no. 12, pp. 29–38, 1992.

[14] S. Altschul, T. Madden, A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. Lipman, "Gapped blast and psi-blast: A new generation of protein database search programs," *Nucleic Acids Res.*, vol. 25, pp. 3389–3402, 1997.

[15] M. Perrone and L. Cooper, "When networks disagree: Ensemble methods for hybrid neural networks," in *Neural Networks for Speech and Image Processing*, R. Mammone, Ed. London, U.K.: Chapman & Hall, 1993, pp. 126–142.

[16] M. Jordan and R. Jacobs, "Hierarchical Mixtures of Experts and the EM Algorithm," Univ. Calif. Berkeley, Berkeley, CA, Tech. Rep. AIM-1440, 1993.

[17] M. J. Foster and H. T. Kung, "The design of special-purpose VLSI chips," *Computer*, vol. 13, no. 1, pp. 26–40, 1980.

[18] A. Halaas, "A systolic VLSI matrix for a family of fundamental searching problems," *Integration, VLSI J.*, vol. 1, no. 3, 1983.

[19] H. Cheng and K. Fu, "VLSI architectures for string matching and pattern matching," *PR*, vol. 20, pp. 125–141, 1987.

[20] H. Yamada, M. Hirata, H. Nagai, and K. Takahashi, "A high-speed string-search engine," *IEEE J. Solid-State Circuits*, vol. 22, pp. 829–834, Oct. 1987.

[21] M. Hirata, H. Yamada, H. Nagai, and K. Takahashi, "A versatile data string-search VLSI," *IEEE J. Solid-State Circuits*, vol. 23, pp. 329–335, Apr. 1988.

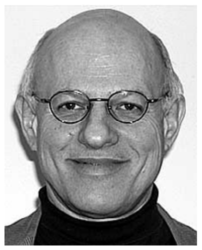
[22] A. Mukherjee, "Hardware algorithms for determining similarity between two strings," *T-COMP*, vol. 38, pp. 600–603, 1989.

[23] A. Halaas, "Arguments and architectures for ASIC's in future information retrieval systems," in *In Proc, IFIP Workshop Design & Test of ASICs*, 1990.

[24] M. E. Isenman and D. E. Shasha, "Performance and architectural issues for string matching," *IEEE Trans. Computers*, vol. 39, pp. 238–250, 1990.

[1]The finite state automaton used can match strings with an edit distance of one.

[25] V. W.-K. Mak, K. C. Lee, and O. Frieder, "Exploiting parallelism in pattern matching: An information retrieval application," *ACM Trans Information Systems*, vol. 9, pp. 52–74, Jan. 1991.
[26] J. Park and K. George, "Efficient parallel hardware algorithms for string matching," *Microprocessors and Microsystems*, vol. 23, no. 3, pp. 155–168, 1999.
[27] Y. Lin and J. Yeh, "A scalable and efficient systolic algorithm for the longest common subsequence problem," *J. Inform. Sci. Eng.*, vol. 18, no. 4, pp. 519–532, 2002.
[28] K.-I. Yu, S.-P. Hsu, and P. Otsubu, "The fast data finder – An architecture for very high speed data search and dissemination," in *Proc. 1st. Int. Conf. Data Engineering*, 1984, pp. 167–174.
[29] R. Sastry, N. Ranganathan, and K. Remedios, "Casm: A VLSI chip for approximate string-matching," *PAMI*, vol. 17, no. 8, pp. 824–830, Aug. 1995.
[30] D. M. Dahle, J. D. Hirschberg, K. 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. Advanced Research in VLSI*, 1997, pp. 145–162.
[31] J. Hirschberg, D. Dahle, K. Karplus, D. Speck, and R. Hughey, "Kestrel: A programmable array for sequence analysis," *J. VLSI Signal Processing Systems for Signal Image and Video Technol.*, vol. 19, no. 2, pp. 115–126, 1998.
[32] J. Lindelien. "The Value of Accelerated Computing in Bioinformatics," Whitepaper. TimeLogic, Crystal Bay, NV. [Online]. Available: http://www.timelogic.com/whitepapers/decypher_benefits_e.pdf
[33] H.-M. Blüthgen and T. G. Noll, "A programmable processor for approximate string matching with high throughput rate," in *Proc. Int. Conf. Application-Specific Systems, Architectures, and Processors (ASAP'00)*, 2000, pp. 309–316.
[34] B. Schmidt, H. Schroder, and M. Schimmler, "Protein sequence comparison on the instruction systolic array," in *Proc. 6th Int. Conf., Parallel Computing Technologies*, vol. 2127, Lecture Notes in Computer Science, V. E. Malyshkin, Ed., Novosibirsk, Russia, Sept. 2001, pp. 498–509.
[35] R. Hughey, "Parallel hardware for sequence comparison and alignment," *CABIOS*, vol. 12, no. 6, pp. 473–479, 1996.
[36] M. Nedland, B. Svingen, and M. Hetland. (2002) "The Interagon Query Language – A User's Guide," Whitepaper. Interagon AS, Trondheim, Norway. [Online]. Available: http://www.interagon.com/pub/whitepapers/IQL.overview-latest.pdf
[37] ——, (2002) "The Interagon Query Language – A Reference Guide," Whitepaper. Interagon AS, Trondheim, Norway. [Online]. Available: http://www.interagon.com/pub/whitepapers/IQL.reference-latest.pdf
[38] M. Amarzguioui, T. Holen, E. Babaie, and H. Prydz, "Tolerance for mutations and chemical modifications in a siRNA," *Nucleic Acids Res.*, vol. 31, no. 2, pp. 589–595, 2003.



Arne Halaas is a Professor in algorithm construction at the Norwegian University of Science and Technology, Trondheim, Norway, where he was Head of the Computer and Information Department from 1982 to 1984 and from 1993 to 1994. From 1981 to 1982, he was a Visiting Professor at the University of Kaiserslautern, Kaiserslautern, Germany. From 1994 to 1995, he was also a Visiting Professor at Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM), University of Montpellier, Montpellier, France. Since 1981, he has worked with special purpose search engines and cofounded three companies operating in this field: Turbit AS (1987), Fast Search & Transfer ASA (1997), and Interagon AS (2002). Fast Search & Transfer ASA is now a major global actor on search engines, while Interagon AS has specialized on complex search tasks, primarily in bioinformatics, requiring high-performance solutions for imprecise, small-fragment searching, and pattern discovery.

Prof. Halaas has served on the Editorial Board of the VLSI Journal *Integration*, since it was established in 1983. He took an active part in establishing the International Federation for Information Processing (IFIP) working group 10.5 on VLSI systems in 1981, and has been involved for over a decade in the arrangements of its biannual series of VLSI conferences.

Børge Svingen received the M.Sc. degree in computer science from the Norwegian University of Science and Technology, Trondheim, Norway, in 1996.

In 1997, he cofounded Fast Search & Transfer ASA, Oslo, Norway, and then in 2002 he cofounded Interagon AS, Trondheim, Norway. He has worked with machine learning, genetic programming, pattern matching and discovery, special-purpose hardware, and architecture and algorithm design. He is now the Chief Technology Officer of IMP Technology AS, Kråkerø, Norway, a database search engine company.

Magnar Nedland received the M.Sc. degree in computer science from the Norwegian University of Science and Technology, Trondheim, Norway, in 2000.

From 2000 to 2001, he was a Systems Engineer with Fast Search & Transfer ASA, Oslo Norway. In 2002, he cofounded Interagon AS, where he is currently a Research Scientist and specializes in C++ software development and architecture design for pattern-matching integrated circuits. He has been involved with the hardware design and development since 1998.

Pål Sætrom received the M.Sc. degree in Computer Science from the Norwegian University of Science and Technology, Trondheim, Norway, in 2000, where he is currently pursuing the Ph.D. degree using machine learning in biological discovery.

He was a Systems Engineer with Fast Search & Transfer ASA, Oslo, Norway, from 2000 to 2001. In 2002, he cofounded Interagon AS, Trondheim, Norway, where he is currently a Research Scientist. He has been involved with hardware design and development since 1998, and has used the chip as a hardware accelerator for challenging genetic programming experiments in his research.

Ola Snøve, Jr. received the M.Sc. degree in mechanical engineering from the Norwegian University of Science and Technology, Trondheim, Norway, in 2000.

From 2000 to 2001, he was a Systems Engineer with Fast Search & Transfer ASA, Oslo, Norway. In 2002, he cofounded Interagon AS, Trondheim, Norway, where he is currently a Research Scientist working with biology applications and business development for the chip architecture.

Olaf René Birkeland received the M.Sc. degree in computer science from the Norwegian University of Science and Technology, Trondheim, Norway, in 1993.

From 1993 to 1998, he was the Director of Engineering with MRT Micro ASA, Oslo Norway/Intelens Inc., Boca Raton, FL. From 1998 to 2001, he was Senior Research and Development Manager with Fast Search & Transfer, Oslo, Norway, ASA. In 2002, he cofounded Interagon AS, Trondheim, Norway, where he is currently Chief Technology Officer. He is a specialist in the development of high-performance hardware for search systems and video processing, and core competencies include ASIC design and development, electronics design, hardware architecture, software design, algorithm design, and system architecture.