Microelectronics Journal 37 (2006) 828–836

Microelectronics Journal

www.elsevier.com/locate/mejo

A CAM-based keyword match processor architecture

Long Bu, John A. Chandy *

Department of Electrical and Computer Engineering, University of Connecticut, 371 Fairfield Road, U-2157, Storrs, CT 06269-2157 USA

Received 10 January 2005; received in revised form 17 October 2005; accepted 25 October 2005
Available online 12 December 2005

Abstract

This paper demonstrates a keyword match processor capable of performing fast dictionary search with approximate match capability. Using a content addressable memory with processor element cells, the processor can process arbitrary sized keywords and match input text streams in a single clock cycle. We present an architecture that allows priority detection of multiple keyword matches on single input strings. The processor is capable of determining approximate match and providing distance information as well. A 64-word design has been developed using 19,000 transistors and it could be expanded to larger sizes easily. Using a modest 0.5 μm process, we are achieving cycle times of 10 ns and the design will scale to smaller feature sizes.

Keywords: VLSI design; Keyword match; Cellular automata

1. Introduction

Searching data for specific keywords or signatures is an important operation in a wide variety of applications including full-text-search, database queries, search engines, and natural language processing. Network protocol operations such as IP routing table lookups, authentication directory lookups, and intrusion detection also require the ability to match input keys on some form of data dictionary. Because of the importance of string matching, there has been a long history of research into efficient software algorithms to improve search times [1–6].

However, in applications that require very fast search times, software approaches may not be adequate, and in such cases, application specific hardware implementations have been introduced. In the context of keyword matching, there has been early work in hardware acceleration particularly for database operations and AI natural language processing [7]. More recently, there has been significant work in the use of FPGAs for pattern matching for network intrusion detection [8–12]. For the most part, these FPGA-based designs must be reconfigured every time the dictionary changes. This is a significant detriment in applications where the dictionary is dynamic. Moreover, in non FPGA designs such as with ASICs, reconfiguration is not an option.

Thus, the use of content-addressable memories (CAM) are particularly attractive for search since they allow the data dictionary to be changed dynamically. Thus, many pattern matching hardware implementations have included the use of content addressable memories (CAM) [13–16]. CAMs allow keyword matches to be done in a single cycle making search a constant time operation rather than a linear (or worse) time operation when done with RAM based software algorithms. With data streaming operations, the CAMs can thus be used to perform search in linear time relative to the length of the input stream rather than the size of the dictionary. The obvious benefits of CAMs have led to their use in a wide variety of networking applications including routing lookups, packet classification and address translation [13].

One of the drawbacks of CAMs, however, is its reliance on fixed size keys. When searching for matches, the input keys must all be the same size. For applications like routing table lookups, where the IP addresses are all the same size, this limitation is not critical. However, for other applications such as directory lookups or signature matching for intrusion detection, the key sizes vary and thus the fixed key size restriction for CAMs can prove to be problematic.

The use of content addressable memories for text search has centered around two strategies: cellular automata [14,17,18] and finite state machines [16,19]. The cellular automata methods cascade CAMs temporally and then use pointers to propagate matches from one CAM set to another [20]. Motomura et al. have

* Corresponding author.
E-mail addresses: long.bu@uconn.edu (L. Bu), long.bu@uconn.edu (J.A. Chandy).

0026-2692/$ - see front matter © 2005 Elsevier Ltd. All rights reserved.
doi:10.1016/j.mejo.2005.10.015

Table 1
Keyword match distance given an input string T = 'blue whale'

Keyword	Distance	
Whales	1	Deletion of 'S'
Shale	1	Substitution of 'S' with 'W'
CUE	2	Substitution of 'C' with 'B' Insertion of 'L'
Blue	0	Exact match

described an cellular automata based architecture to do dictionary search in VLSI [18]. The keywords are grouped in four character segments and they allow for concatenating groups to create larger keywords. However, the grouping by four means that keywords that are not multiples of the group size will result in wasted CAM storage space. Hirata, et al. have designed a FSM-based CAM search architecture that also accommodate variable length keywords by grouping keywords into segments.

This paper presents a cellular automata based design that is more space efficient and can easily handle arbitrarily sized keywords. The design is based on temporally cascaded CAMs and is an extension of Motomura's cellular automata structure. As with Motomura's design, processor element cells, external to the CAM array, will process character match signals from the CAM and output keyword match signals. The architecture is flexible enough to allow for 'approximate word' matches as well. We present the architecture in the following section and then discuss the implementation and performance results in subsequent sections.

2. Keyword match processor architecture

To begin the discussion of the architecture, a definition of the keyword match problem is warranted. Assume a string, T, with $|T|$ characters and a set, K, of keywords of arbitrary lengths. Each character has m bits. The goal is to determine if T matches any keyword $k \in K$. T is said to exactly match a keyword, k, if $|k|$ is equal to $|T|$ and $T_i = k_i$ for all $i \leq |T|$. T can approximately match a keyword, k, if the distance, d (T, k), is less than some criterion. The distance is the sum of insertions, substitutions, and deletions occurring between T and k. A distance of 0 indicates an exact match. Examples of match distance are shown in Table 1.

Given this problem definition, a hardware implementation of the problem was designed using a keyword match processor. An architectural overview of the keyword match processor is shown in Fig. 1. The processor is composed of six major

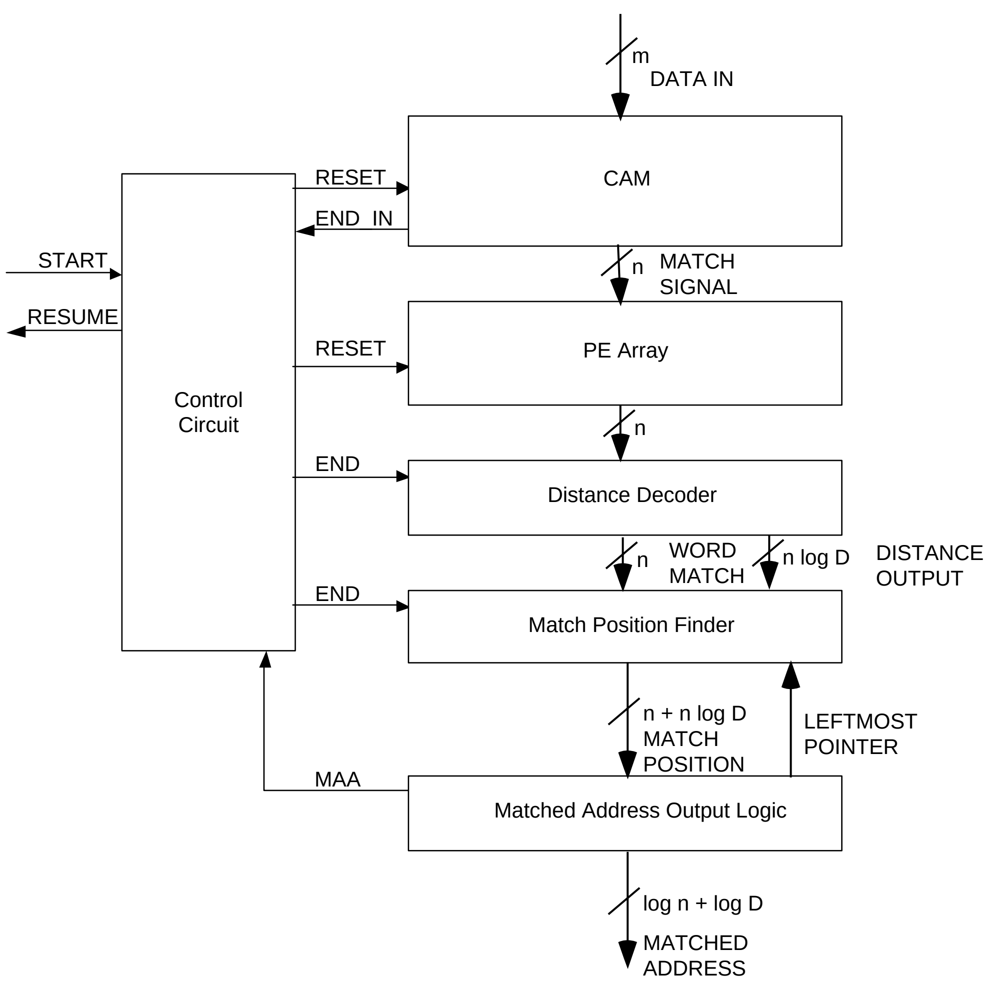

Fig. 1. Keyword match processor architecture.

components: the control circuit, the CAM, the PE array, the distance decoder, the match position finder, and the matched address output logic. These components are described in further detail below.

2.1. Control circuit

The Control Circuit is a simple state machine that manages the data flow of the input stream and resultant outputs. A START input signal resets the CAM and PE arrays and readies the system to start accepting an input data stream. The resume output signal is asserted when the match processor has finished processing the input data and is ready for new data word to be matched.

2.2. CAM array

Details of the CAM and PE array layout are shown in Fig. 2. The CAM stores the keywords as m-bit characters in an m by n array of CAM cells where n is equal to the number of characters in the CAM, i.e. the sum of the lengths of all keyword in K. For keywords comprised of ASCII characters, as would be the case for text search, the character size, m, would be set to seven. For more general byte-based keywords, m is equal to 8. Since the CAM is a sequence of variable length keywords, we need some way to separate the keywords. To do so, we insert between keywords a delimiter appropriate to the application—such as ‘space’ or 0xFF for text search. Certain applications may not have any appropriate delimiters such as when all possible characters are valid. We will discuss mechanisms to address this case later.

The detail of the CAM array is shown in Fig. 3. Each column in the CAM array corresponds to a character in a keyword. Each cell is a normal CAM cell with a traditional 6-transistor RAM cell with an additional three transistors for circuitry to indicate a match between the stored bit and the incoming bit. If there is a match, the output match line is pulled low. The column match line is shared as a wired-NOR line between all the cells in the column. Thus, when an input m-bit character is applied to a column, the column match signal is active high when all m bits match. The input character is applied simultaneously to all n columns in the array.

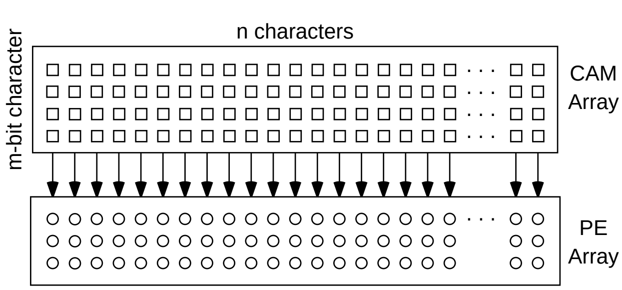

Fig. 2. CAM and PE array layout.

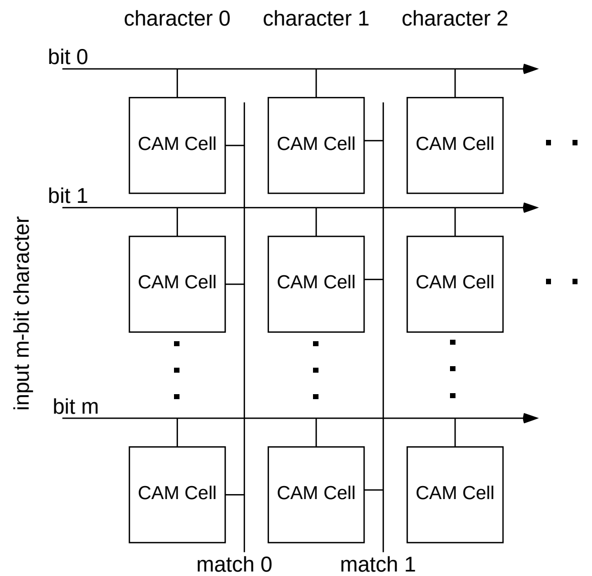

Fig. 3. CAM array.

2.3. PE array

The CAM array described so far is similar to what would be seen in any traditional CAM design. The uniqueness of the keyword match processor is in the processor element (PE) array where each PE is a finite state machine that carries out the approximate match algorithm described later. The PE array is similar in concept to the cellular automaton processor arrays proposed by Motomura et al. [14,18]. The original Motomura design processed 4 match signals from the CAM array in a 5×3 PE array. Thus, a n-column CAM array requires a $5n/4\times3$ PE array. This grouping by four necessitates that words must start on a four-byte boundary, leading to a wastage of space. Moreover, it also introduces an extra fifth column into the array. We have modified the design so that the entire PE array is comprised of just n columns and $D+1$ rows of processing elements (PE) where D is the maximum distance that can be calculated. This simplification of the PE array removes the extra ‘fifth’ column present in the Motomura design, and with modifications to the algorithm as described below, we are also able to remove the restriction that words begin on a four-byte boundary. The maximum distance, D, that can be calculated is equal to one less than the number of rows in the PE array. With a three row array, as implemented, the maximum distance that can be calculated is two.

Let us now describe the keyword algorithm carried out by the PE array. Each PE holds a binary value called a flag, which is used in calculating match distance. We label the PE flags, PE(i, j), and the incoming match signals from the CAM array are labeled $M(i)$, where i is the character number and j is the PE array row number. At the first clock cycle of the keyword match process, we reset both the CAM and PE arrays. The CAM array reset causes a delimiter to be presented to the CAM which activates match signals at every position i where there is a delimiter, i.e. the end of each keyword in the CAM.

When the PE array is reset, PE(i, 0) is set to 1 when $M(i-1)=1$ or $i=0$ and all other PE(i, j) are set to 0.

On each subsequent clock cycle, a character from the input sequence T is presented to the CAM array and is simultaneously compared against all the characters in all the keywords in K. Any character matches are passed to the PE array for match distance processing. The PE array calculates distance by transferring flags from one PE to the next using a set of rules as described below. These rules are similar to the rules derived by Motomura et al. The subscript t indicates the current clock cycle and $t+1$ is the next clock cycle.

PE flag processing rules

(1) If there is a match signal at column i and a flag exists at PE(i, j), the flag is passed to PE($i+1$, j) at the next clock cycle to represent the situation that it is an exact character match. The flag remains on row j to indicate no change in the distance when there is an exact character match. The flag is also passed to PE(i, $j+1$) to cover the possibility of an insertion character that also matches.
if $M_t(j)=1$ and $\mathrm{PE}_t(i, j)=1$ then

$$PE_{t+1}(i+1,j) = 1$$

$$PE_{t+1}(i,j+1) = 1$$

(2) If there is a mismatch signal at column i and a flag exists at PE(i, j), the flag is passed to PE($i+1$, $j+1$) at the next clock cycle to represent the substitution case. The flag is also passed to PE(i, $j+1$) to cover the possibility of an insertion character that does not match.
if $M_t(i)\neq 1$ and $\mathrm{PE}_t(i, j)=1$ then

$$PE_{t+1}(i+1,j+1) = 1$$

$$PE_{t+1}(i,j+1) = 1$$

(3) Whenever there is a flag at PE(i, j), flags are set for the same clock cycle at PE($i+1$, $j+1$), PE($i+2$, $j+2$)… until the boundary of the PE Array. This rule is to handle the deletion case. if $\mathrm{PE}_t(i, j)=1$ then

$$\text{for} \quad (1 \geq d \leq D-j)$$

$$PE_\mathrm{t}(i+d, j+d) = 1$$

To implement these rules, each PE(i, j) is designed to look to the up, upper left, and left for flag signals (PE(i, $j-1$), PE($i-1$, $j-1$), PE($i-1$, j)). For PE(i, j), the match signal to its left ($M(i-1)$) is used to determine how to propagate the flags. The algorithm can be expressed with the following logic equation for each PE to determine its next state (See Fig. 4 for the logic schematic. PE_U, PE_UL, and PE_L correspond to the up, upper left, and left flags, respectively).

$$\begin{aligned}\mathrm{PE}_{t+1}(i,j) = {} & \mathrm{PE}_t(i,j-1) + \mathrm{PE}_t(i-1,j)M(j-1) \\ & + \mathrm{PE}_t(i-1,j-1)\overline{M(j-1)} + \mathrm{PE}_{t+1}(i-1,j-1)\end{aligned} \tag{1}$$

2.4. Distance decoder

The last character of T should be the delimiter. When this delimiter is presented to the CAM, $M(i)$ will be set at all locations of the CAM where there is a delimiter and an END_IN signal is asserted which lets the control circuit know that the input stream has finished. This also causes an END signal to be sent to the Distance Decoder and Match Position Finder. If there is a flag present in a delimiter column in the PE array, we know there is a match. The row position of the flag indicates the distance of the match. The Distance decoder interprets the row information and match signals to output a word match signal and the distance encoding. It is enabled only when it receives the END signal from the Control Circuit. The distance decoder logic to determine a word match is as follows: $\mathrm{WM}(i)=\mathrm{END}\cdot \mathrm{M}(i)\cdot(\mathrm{PE}(i, 0)+\mathrm{PE}(i, 1)+\ldots+\mathrm{PE}(i, \mathrm{D}))$

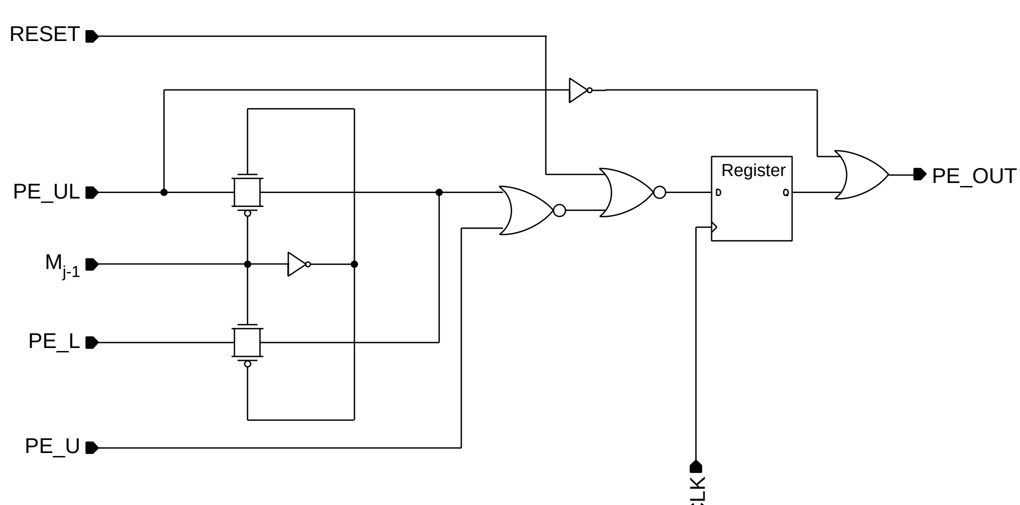

Fig. 4. Keyword match processor PE design.

2.5. Match position finder

The distance decoder can only determine a word match at the end of the keyword, but in order to be useful to any application, the keyword match processor must generate the start of the keyword rather than the end. It is the match position finder (MPF) that accomplishes by taking the word match signal at the end of the keyword and generating a signal corresponding to the start of the keyword. The MPF finds the start by propagating the word match signal to the left until it reaches a match signal from the PE array, in other words, the previous word end. It can then output a match position (MP) signal corresponding to the start of the matched keyword. The distance information for this match is passed along with this matched position.

2.6. Matched address output (MAO) logic

The output of the MPF is an array of *n* registered MP bits. A non-zero entry is an indication that there was a keyword match at that location. This needs to be encoded into a binary address for match processing by an external CPU. Moreover, since there may be multiple matches, these matched addresses must be separated out. The MAO Logic uses a binary tree priority encoder structure to generate the matched address available (MAA) signal which tells the control circuitry that there are matches and to hold off accepting the next input stream. Then, a leftmost pointer (LP) signal propagates back up the tree to determine which is the left most MP signal. At each level of the tree, a bit of the address is generated from the LP signals at that level. The associated distance encoding is output along with the encoded address. When the LP signal reaches the top of the tree, it is sent to the MPF logic where it resets the register of the MP signal that was just encoded. On the next clock cycle, the MAO can then encode the next left-most MP entry. Fig. 5 shows the structure of the four bit tree along with the logic for each block in the tree.

The MPF in addition with the MAO is one of the unique features of the keyword match processor. No previous CAM-based design has described how to encode multiple matches in an efficient manner. The tree-based structure of the MAO makes it very efficient to generate addresses that correspond to keyword matches.

2.7. Keyword match processor example

Fig. 6 shows an example of the keyword match processor in operation. There are two keywords in the CAM: UCONN and HUSKIES. On the first clock cycle, PE(0, 0) and PE(6, 0) are set because they represent the start of the two keywords. PE(1, 1), PE(2, 2), PE(7, 1), and PE(8, 2) are also set because of rule three described above. With an input string of CONE, on the first clock cycle, the character *C* is presented to the CAM. A match on column 1 propagates the flag at PE(1, 1) to PE(1, 2) and PE(2, 1) per rule 1 and then on to PE(3, 2) per rule three. The gray circles indicate the position of the flag in the subsequent cycle. As characters *O*, *N*, and *E* are applied to the CAM, the flags propagate down and across the PE array. When the input string is complete, the delimiter is applied again to identify the keyword end columns and the flags are processed to determine a match. In the example, there is a flag in column five, row two. Since column five is a delimiter column, we declare a match. Because the flag is in row tow, the distance is two as expected (CONE has a deletion, *U*, and a substitution, *E* for *N*).

Since a new input character is presented every clock cycle, the keyword match processor can evaluate an entire input string in linear time relative to the size of the input stream. The processing time is constant relative to the number of keywords. To be exact, the search process takes exactly $p+q+2$ cycles where p is the length of the input stream and q is the number of matches. The two extra cycles are required to initialize the PE array and to process the distance flags from the delimiter column.

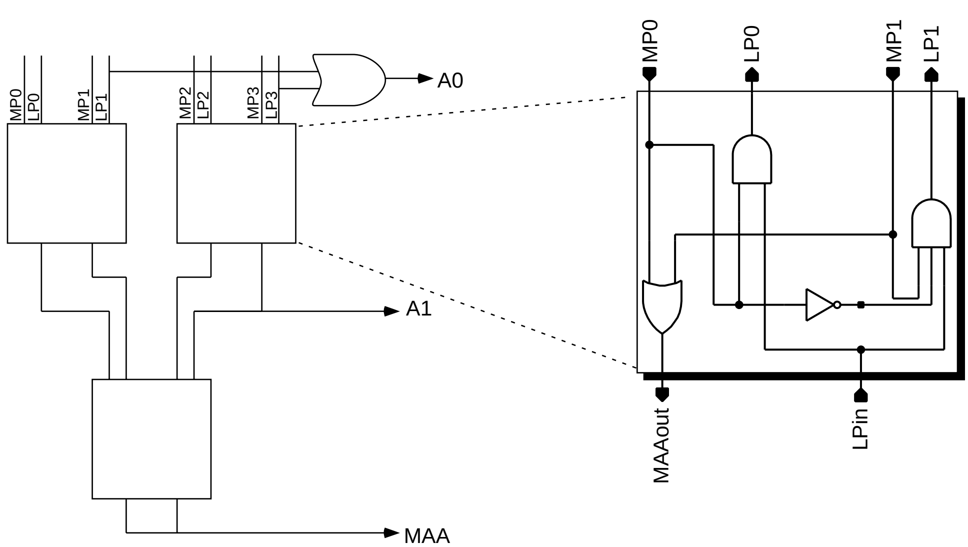

Fig. 5. MAO logic binary tree (four bit example).

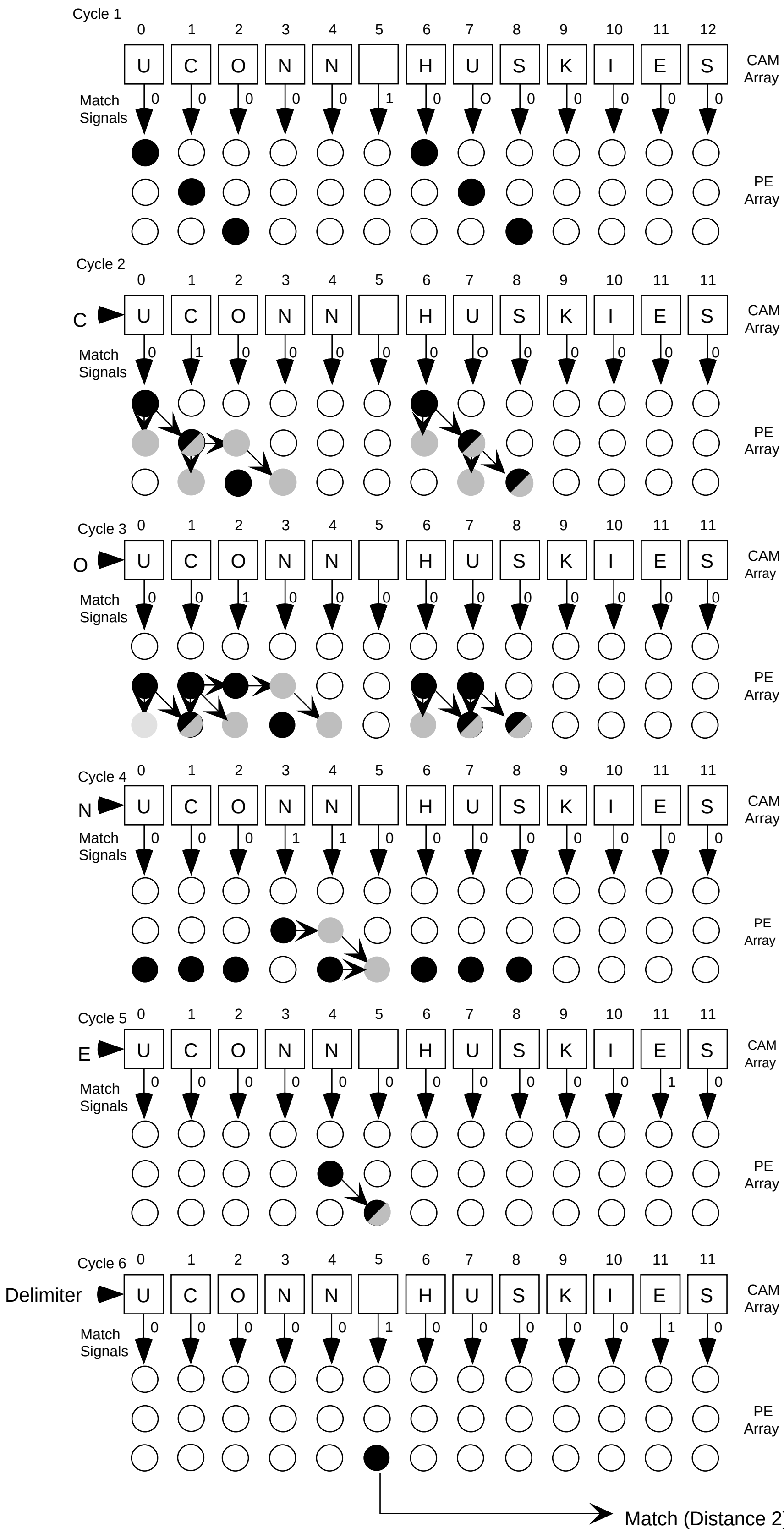

Fig. 6. Keyword match processor example.

2.8. Architecture extensions

2.8.1. Keyword match processor with no delimiters

As discussed above, there may be cases where the input data stream may not have any obvious delimiters. In such a case, we introduce a row of n registers parallel to the CAM array. The register stores a '1' if it is in the column of the last character of a keyword, otherwise it stores a '0'. These registers can be initialized when the CAM is initialized. PE array initialization can thus use the register values to determine keyword ends rather than the match signal from a delimiter column. The same is true during distance match calculation. During flag propagation, we must be careful not to propagate flags past the keyword end into the next keyword.

2.8.2. Subsequence keyword match. The architecture as presented can only compare the entire input stream against the keyword set. However, there may be applications when the input stream is much longer than any keyword, and thus the goal is to search for matches against subsequences of the input stream. The restatement of the keyword match problem is then to determine if any subsequence in T matches any keyword $k \in K$. Thus, T is said to exactly match a keyword, k, if there exists some subsequence $T_{e\ldots e+|k|-1}$ where $T_{e+i}=k_i$ for all $i \leq |k|$ where $|k|$ is the number of characters in k. And likewise, T approximately matches a keyword, k, if there exists a s-character subsequence $T_{e\ldots e+s-1}$ such that the distance, $d(T_{e\ldots e+s-1}, k)$, is less than some criterion. The algorithm to handle this new problem is similar to before except now we can potentially have keywords starting at any point of the input stream. This change simply requires that we introduce a flag at the beginning of each keyword on every clock cycle, not just the first clock cycle.

The modification of the rules are shown in Fig. 7. $E(i)$ is the register value for end of keyword.

(1) `if` $M_t(j)=1$ `and` $PE_t(i,j)=1$ `then`
  $PE_{t+1}(i,j+1)=1$
  `if` $E(i)\neq 1$ `then`
    $PE_{t+1}(i+1,j)=1$

(2) `if` $M_t(i)=1$ `and` $PE_t(i,j)=1$ `then`
  $PE_{t+1}(i,j+1)=1$
  `if` $E(i)\neq 1$ `then`
    $PE_{t+1}(i+1,j+1)=1$

(3) `if` $PE_t(i,j)=1$ `then`
  `for` $(1\geq d\leq D-j)$
    `if` $E(i+d-1)\neq 1$ `then`
      $PE_t(i+d,j+d)=1$

(4) $PE_t(0,0)=1$
  `if` $E(i)=1$ `then`
    $PE_t(i+1,0)=1$

Fig. 7. Keyword match processor rules with support for no delimiters and subsequence matching.

3. Keyword match processor VLSI implementation

The match processor was designed using the AMIS 0.5 μm 3-metal layer process. The layout was completed manually without using any standard cell libraries. A 64 character keyword match processor was implemented on a 4.5 mm square die using roughly 23,000 transistors not including the CAM array. The PE array uses 7680 transistors (three rows with 64 PEs per row and 40 transistors per PE), the distance decoder uses 2432 transistors, and the match position finder uses 7296 transistors. The remainder of the transistors are in the MAO logic and control circuit. The design consumes about 270 transistors per character. Thus, a 32 K character keyword match processor would use roughly 8.8 million transistors.

Using Cadence analog environment, we measured delays for various components (see Table 2) with parasitically extracted resistance and capacitance values. The CAM and PE delays are relatively short compared to the distance decoder delay and match position delay. The distance decoder and match position finder operations must happen within one clock cycle, so t_{DD} and t_{MPF} are the primary factors influencing maximum clock speed. In other words, $t_{CLK} > t_{DD} + t_{DD2MPF} + t_{MPF} \Rightarrow t_{CLK} > 7.66$ ns. A clock period of 7.66 ns leads to a maximum clock frequency of 130.5 MHz. Taking into account the setup times and propagation times of the registers, we have been able to comfortably run the circuit at 100 MHz. That allows the match processor to process data streams at 100 million characters per second, or 800 Mb/s. By scaling to a 0.1 μm process, we believe that we could improve the operating frequency to near 500 MHz providing a data stream operating rate of near 4 Gb/s. This data rate is fast enough to process data streams at gigabit network rates.

The delays presented in Table 2 are for a 64 character keyword match processor. For a practical implementation, the size of the match processor should be much larger. In the following discussion, we develop a delay model for a larger keyword match processor. Because of the parallel nature of the processing, the t_{CAM}, t_{PE}, and t_{DD} delays are not dependent on the size of the match processor. However, because the t_{MPF} and t_{MAO} delays are due to propagation, these delays are dependent on size parameters.

The MPF delay is comprised mostly of the time to propagate the Word Match signal to the beginning of the keyword. This delay is thus dependent on the longest keyword in the set. For an average keyword length of five characters, the delay is

Table 2
Keyword match processor delays

	Description	Delay (ns)
t_{CAM}	Delay from input data to match signal	1.14
t_{PE}	Delay for flag propagation	0.85
t_{DD}	Delay to WM and distance output	3.43
t_{DD2MPF}	Delay from DD to MPF blocks	0.15
t_{MPF}	Delay to propagate match position eight positions	4.08
t_{MP2MAA}	Delay from MP to MAA signal	1.69
t_{MAA2LP}	Delay from MAA to LP signal	1.37

3.43 ns. If we do not allow keywords longer than eight characters, we can guarantee a maximum delay of 4.08 ns. Increasing the maximum length will add roughly.Two nano seconds per character to the delay. Thus, the MPF delay is not dependent on the size of the array but only on the length of the longest keyword.

The MAO logic delay has two components and they are both dependent on the size of the array. The first is the delay from the MP signals to the MAA signal. Our measurements show a maximum delay of .226 ns per level of the binary tree. With t_{MP2MAA} equal to 1.69 ns for a 6 level tree ($n=64$), we can derive that $t_{\mathrm{MP2MAA}}=.226\log_2 n+.335$. The second component of the delay is the LP propagation back up the tree. The delay per level of the tree is 0.228 ns. With t_{MAA2LP} equal to 1.37 ns for a six level tree, we can derive that $t_{\mathrm{MAA2LP}}=.228\log_2 n$. We can generalize this and calculate the delay for the entire MAO tree: $t_{\mathrm{d}}=t_{\mathrm{MP2MAA}}+t_{\mathrm{MAA2LP}}=.454\log_2 n+.335$ ns.

For a 64 character match processor, the DD and MPF delays dominate the MAO delays. However as we increase the size of the array, MAO delay becomes more significant. At $n=64$ K, the MAO delay is approximately 7.6 ns and comparable to the distance decoder and match position finder delays. Thus, as the number of characters gets larger than 64 K, the MAO delay becomes the determining factor in clock speed, but for smaller CAMs, we can still maintain a 100 MHz clock. Though character arrays that large are not envisioned, pipelining the MAO stage is an obvious solution to increase the clock speed. Likewise, the distance decoder and match position finder stages can be pipelined as well to get a significantly faster clock frequency.

In comparison with a pure software implementation, the hardware implementation is significantly faster. Consider a dictionary search application, where we wish to determine if a particular word is in the dictionary. In the VLSI implementation, the dictionary would be stored in the CAM, and the search word would be input serially to the keyword match processor. For example, if we are searching for an eight character keyword in a 32 K character dictionary, the resolution time for a match would be nine clock cycles—eight clock cycles for each character of the keyword plus one extra clock cycle for the match. At a 100 MHz clock, the time would be 80 ns.

A purely software implementation would store the dictionary in normal RAM. Using an algorithm such as Boyer-Moore [3], the search time would require at a worst case roughly one comparison for each character in the dictionary. Thus, for a 32 K character dictionary, the search would take approximately 64 K cycles—32 K comparison cycles and 32 K load from memory cycles (assuming that the dictionary is in single cycle access cache). A 4 GHz processor would take 16 ms to execute those 64 K cycles. Thus, the VLSI implementation is roughly 200 times faster than the software implementation. In high-speed applications, the advantage of the hardware implementation is clear. Note, that in applications where the dictionary is very large and thus does not fit in the CAM, software based approaches would still be required or could be used in cooperation with a CAM-based solution.

4. Related work

Genome sequencing is a pattern matching dependent application that is very compute-intensive and, as a result, hardware implementations have been suggested as a remedy. Several hardware implementations have been proposed to accelerate the Smith-Waterman algorithm for sequence alignment [21–23]. For the most part, these hardware implementations use systolic arrays to attain speedup in the Smith-Waterman algorithm. The architecture is similar to our PE array, in that data progresses systolically through the array. In the case of the Smith-Waterman systolic arrays, each PE is responsible for calculating intermediate values that are propagated through the array. Data dependencies restrict when PEs can operate. The serial algorithm is $O(mn)$ where n is the size of the input string and m is the size of the pattern. When m is very large, there is lots of parallelism available, and thus a hardware implementation is practical. However, for small m, as with text dictionary search, there is not enough parallelism to extract. In our PE array, the parallelism is limited only by the size of the CAM array since all matches are done concurrently across multiple keywords/patterns at once.

In the area of network intrusion detection, there has been significant FPGA work. For example, there have been comparator-based designs that use sliding window comparisons and single character comparators with shift registers to propagate matches across clock cycles [11,9]. The approach is similar to our PE array, with the difference being that the patterns are programmed into the FPGA logic, thus requiring reprogramming whenever the pattern set changes. The use of CAMs as in our design precludes the need for reprogramming. The recent FPGA approaches, however, do support multi-character parallelism, i.e. the ability to match multiple characters in the input stream at once. With a CAM-based approach as we have described, this type of parallelism will require replication of the CAM array—a significant expense in terms of area.

Two FPGA implementations that do stand out for not requiring reprogramming are the KMP approach [24] and Bloom filters [25]. The KMP approach translates the Knuth–Morris–Platt string matching algorithm into hardware and uses on-chip RAM to store the patterns. Bloom filters is another approach using probabilistic methods to perform string matching. While very fast and area-efficient, Bloom filters unfortunately can sometimes produce false matches.

The most closely related work to our design is the work by Motomura [14,18]. The key difference with Motomura's design is the ability to handle varying sized keywords without grouping and also a priority encoder that can output multiple keyword matches. By eliminating grouping, the resulting PE is more compact. Our priority encoder is a novel design that can detect multiple pattern matches more efficiently. The binary tree structure lessens the delay impact of larger arrays.

5. Conclusions and future directions

We have designed and fabricated a VLSI implementation of a keyword match processor. The design extends cascaded CAM designs to do fast keyword matching on input data streams with arbitrarily sized keywords. A design of a priority encoder that can handle multiple keyword matches is also demonstrated. The current implementation allows us to process data at 800 Mb/s. The applications of such a keyword match processor are numerous, but the most promising area is in network processing. We have used a modified keyword match processor design to create a network intrusion detection system that can process incoming networks at 800 Mb/s. Current software based solutions can barely handle 100 Mb/s before dropping packets. Other potential applications include directory lookups in network storage file systems and URL matching for load balancing.

References

[1] A. Aho, M. Corasick, Efficient string matching: an aid to bibliographic search, Communications of ACM 18 (6) (1975) 333–343.

[2] D.E. Knuth, J.H. Morris, V.R. Pratt, Fast pattern matching in strings, SIAM Journal on Computing 6 (2) (1977) 323–350.

[3] R.S. Boyer, J.S. Moore, A fast string searching algorithm, Communications of ACM 20 (10) (1977) 762–772.

[4] C.J. Coit, S. Staniford, J. McAlerney, Towards faster string matching for intrusion detection or exceeding the speed of snort, in: Proceedings of DARPA Information Survivability Conference and Exposition II, 2001, pp. I:367–373.

[5] M. Fisk, G. Varghese, Fast content-based packet handling for intrusion detection, Tech. Rep. CS2001-0670, Department of Computer Science, University of California, San Diego, 2001.

[6] K. Anagnostakis, S. Antonatos, E.P. Markatos, M. Polychronakis, E2XB: a domain-specific string matching algorithm for intrustion detection, in: Proceedings of IFIP International Information Security Conference, 2003, pp. 217–228.

[7] K.E. Grosspietsch, Associative processors and memories: a survey. IEEE Micro 12(3).

[8] B.L. Hutchings, R. Franklin, D. Carver, Assisting network intrusion detection with reconfigurable hardware, in: Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines, 2002, pp. 111–120.

[9] Y.H. Cho, W.H. Mangione-Smith, Deep packet filter with dedicated logic and read only memories, in: Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines, 2004, pp. 125–134.

[10] C. Clark, D. Schimmel, Scalable multi-pattern matching on high-speed networks, in: Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines, 2004.

[11] I. Sourdis, D. Pnevmatikatos, Pre-decoded CAMs for efficient and high-speed NIDS pattern matching, in: Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines, 2004.

[12] Y. Sugawara, M. Inaba, K. Hiraki, Over 10Gbps string matching mechanism for multi-stream packet scanning systems, in: Proceedings of International Conference on Field Programmable Logic and Applications, 2004.

[13] A.J. McAuley, P. Francis, Fast routing table lookup using CAMs, in: Proceedings of IEEE INFOCOM, vol. 3, 1993, pp. 1382–1391.

[14] M. Motomura, J. Toyoura, K. Hirata, H. Ooka, H. Yamada, T. Enomoto, A 1.2-million transistor, 33 MHz, 20-b dictionary search processor (DISP) ULSI with a 160-kb CAM, IEEE Journal of Solid-State Circuits 25 (5) (1990) 1158–1164.

[15] R. Panigrahy, S. Sharma, Sorting and searching using ternary CAMs, IEEE Micro 23 (1) (2003) 44–53.

[16] J.P. Wade, C.G. Sodini, A ternary content addressable search engine, IEEE Journal of Solid-State Circuits 24 (4) (1989) 1003–1013.

[17] A. Mukhopadhyay, Hardware algorithms for string processing, IEEE Computer (1980) 508–511.

[18] M. Motomura, H. Yamada, T. Enomoto, A 2k-word dictionary search processor (DISP) with an approximate word search capability, IEEE Journal of Solid-State Circuits 27 (6) (1992) 883–891.

[19] M. Hirata, H. Yamada, H. Nagai, K. Takahashi, A versatile data string-search VLSI, IEEE Journal of Solid-State Circuits 23 (2) (1988) 329–335.

[20] T. Moors, A. Cantoni, Cascading content addressable memories, IEEE Micro 12 (3) (1992) 56–66.

[21] R.P. Jacobi, M. Ayala-Rincon, L.G.A. Carvalho, C.H. Llanos, R.W. Hartenstein, Reconfigurable systems for sequence alignment and for general dynamic programming, Genetics and Molecular Research 4 (3) (2005) 543–552.

[22] K. Puttegowda, W. Worek, N. Pappas, A. Dandapani, P. Athanas, A. Dickerman, A run-time reconfigurable system for gene-sequence searching, in: Proceedings of the International VLSI Design Conference, 2003, pp. 561–566.

[23] C.W. Yu, K.H. Kwong, K.H. Lee, P.H.W. Leong, A smith-waterman systolic cell, in: Proceedings of International Conference on Field Programmable Logic and Applications, 2003.

[24] Z.K. Baker, V.K. Prasanna, A methodology for synthesis of efficient intrusion detection systems on FPGAs, in: Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines, 2004, pp. 135–144.

[25] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, J. Lockwood, Implementation of a deep packet inspection circuit using parallel bloom filters in reconfigurable hardware, in: Proceedings of HOTi03, 2003.