# A Unified Model of Pattern-Matching Circuit Architectures

Tech Report GIT-CERCS-05-20

Christopher R. Clark and David E. Schimmel
*Center for Experimental Research in Computer Systems*
*School of Electrical and Computer Engineering*
*Georgia Institute of Technology, Atlanta, GA*
*{cclark, schimmel}@ece.gatech.edu*

## Abstract

*There has been a significant volume of recent work on FPGA designs for pattern matching. Although various pattern-matching architectures have been presented, attempts to compare different designs have been inconclusive, or even misleading, due to the lack of a common evaluation framework. In this paper, we present an analytical model of FPGA pattern-matching architectures that quantitatively expresses the relationships between pattern properties, circuit area, and circuit delay. We derive equations that show how the performance of each architecture is dependent on the properties of the pattern set. This model enables many different pattern-matching architectures to be compared in order to determine the optimal design for a given pattern-matching application.*

## 1. Introduction

In recent years, there have been a number of studies on FPGA designs for pattern matching [3-15]. Despite the amount of work done in this area, there is not a clear picture of the relative trade-offs of different approaches. Some authors have tried to make comparisons between implementations by using data from different publications, but variations in the FPGA devices and patterns used have resulted in conflicting results. With the right combination of patterns, circuit parameters, and metrics, a particular architecture can be made to look better than others, but, under different conditions, another architecture might be better. No attempt has been made to evaluate a wide range of architectures under a common framework. In this paper, we develop an analytical model of FPGA pattern-matching circuits and use the model to describe most recently-published designs while illuminating several novel approaches. The unified model of pattern-matching designs presented here allows a system designer to make an informed choice of circuit architecture for a particular application.

First, we define some terminology and notation used in the model. Then, we describe the components that make up a pattern-matching circuit. Next, factors affecting circuit delay are discussed and techniques for trading increased area for reduced delay are presented. Finally, the model is used to derive architecture-specific equations that predict pattern-matching circuit properties based on pattern set properties.

The remainder of this paper covers design approaches for building FPGA pattern-matching circuits that tailor the reconfigurable logic and routing resources to a particular pattern set. Other approaches, such as those that use embedded memory blocks or off-chip memory to store programmable hash tables [9] or state machines [4], are not considered at this time. We are investigating models of these approaches, and they may be included in the future.

## 2. Model Definition

The basic pattern-matching problem considered here is that in which a set of patterns is to be searched for in one or more bodies of text. The pattern set is used to generate a pattern-matching circuit description that is compiled into an FPGA configuration file. This FPGA pattern-matching system is then used to concurrently detect all occurrences of the patterns contained in a given data stream. In this section, we define notation for describing various attributes of patterns and FPGAs.

### 2.1. Pattern Model

The patterns and the text stream are made up of symbols from an alphabet $\Sigma$ that contains $\sigma$ characters. Each character is represented by a binary sequence of $b$ bits, where $b = \lceil \log_2(\sigma) \rceil$. A pattern $x$ of length $|x|$ is a sequence of characters $x[0], x[1], \ldots, x[m\text{-}1]$, where $x[j] \in \Sigma$. The set of all patterns to be considered is denoted by $P$ and contains $p$ patterns. We define $\Theta_k[j]$ as the number of occurrences of a particular character $\theta_k$ at index $j$ across all patterns. The largest index of occurrence of $\theta_k$ in any pattern is $\lambda_k$. The total number of occurrences of $\theta_k$ in $P$, $\Theta_k$, is calculated by adding up the number of occurrences at each index using (1).

$$\Theta_k = \sum_{j=0}^{\lambda_k - 1} \Theta_k[j] \tag{1}$$

The length of the longest pattern in $P$ is $\Lambda$. The number of unique characters used in $P$ is $\Omega$. The total number of characters in $P$ is denoted by $M$ and can be determined either by summing the lengths of each pattern in $P$, or by summing the total occurrences of each character in $\Sigma$ as shown by (2).

$$M = \sum_{k=1}^{p} |x_k| \; = \; \sum_{k=1}^{\sigma} \Theta_k \tag{2}$$

### 2.2. FPGA Model

Our model is based on an FPGA architecture that is commonly used in modern commercial products, such as the Xilinx Virtex [2] and Altera Stratix [1] lines of devices. These FPGAs contain an array of homogeneous logic elements that are connected to a configurable routing network. Each FPGA logic element (LE) contains a 4-input look-up table (LUT) and a one-bit flip-flop (FF), as shown in Figure 1. A LUT can be programmed to implement any logical function with up to four inputs and one output. The output of each LE connects to the routing network and is selectable between either a registered or unregistered version of the LUT output.

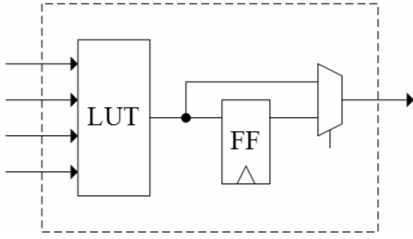


**Figure 1.** An FPGA Logic Element (LE)

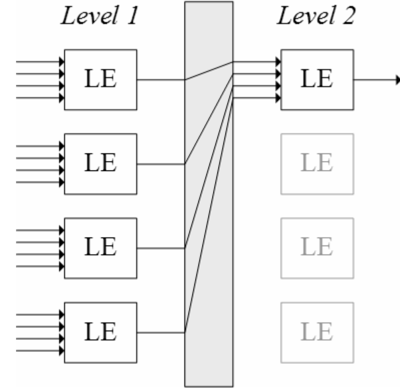


**Figure 2.** A 16-input function computed using LEs with 4-input LUTs

Multiple levels of LEs can be cascaded to implement functions with more than four inputs. For example, Figure 2 shows how a 16-input function can be computed with two levels of LEs. In general, a basic Boolean logic function (i.e. AND, OR, XOR) of $n$ inputs can be implemented in $\lceil \log_4(n) \rceil$ LE levels using a total of $A_f(n)$ LEs, as given by

$$A_f(n) = \sum_{k=1}^{\lceil \log_4(n) \rceil} \left\lceil \frac{n}{4^k} \right\rceil . \tag{3}$$

Because an LE is the fundamental unit of circuit area in an FPGA, we refer to $A_f(n)$ as the area required to implement an $n$-input function. If the registered output of each LE is selected, the latency of the function (the number of clock cycles required to compute the final output) is equal to the number of LE levels and is computed using

$$L_f(n) = \lceil \log_4(n) \rceil . \tag{4}$$

The area required to implement an $n$-bit register using LEs with a single flip-flop can be expressed by

$$A_r(n) = n . \tag{5}$$

## 3. Pattern-Matching Circuit Components

In general, a pattern-matching circuit consists of three main components: a method of decoding input characters, a means of storing the history of inputs, and a mechanism for determining when pattern matches have occurred. The input to the circuit is a stream of text characters and the output is a set of wires indicating which patterns have been matched. The following sections describe the functionality and available design parameters for each aspect of a pattern-matching circuit.

## 3.1. Input Text Stream

The input text stream provides characters that are compared against the stored patterns. In the simplest case, the pattern-matching circuit reads one text character per clock cycle. By trading increased circuit area for more parallelism, it is possible to handle $N$ characters per cycle. When the input is processed in multi-character substrings, a pattern may start at any offset within a substring. Therefore, the circuit must separately check all $N$ offsets in parallel, effectively increasing the pattern count to $p \times N$. The width of the input data bus is $N \times b$ bits.

## 3.2. Character Decoding

We define character decoding as the process of generating a one-bit character-match signal from a $b$-bit encoded input character. Functionally, a character decoder is equivalent to a binary comparator with one input fixed at compile-time. In a pattern-matching circuit, the fixed input has the value of a pattern character. An optimized implementation of a character decoder is a $b$-bit AND function with active-high inputs in positions corresponding to '1' bits and active-low inputs in positions corresponding to '0' bits in the encoded representation of a pattern character.

There are several possible configurations for connecting the inputs and outputs of the character decoders to other pattern-matching circuit components. The decoder inputs can either be directly connected to the input text stream, or they can be connected to the input history registers described in the next section. If the decoder inputs come from the input text stream, then the decoder outputs will connect to the input history component, as shown in Figure 3. Otherwise, the decoder inputs will be taken from the input history, and the outputs of the decoders will be connected to the pattern-match functions associated with one or more patterns, as in Figure 4. If the decoder outputs are used by exactly one pattern, we call this a *per pattern decoding* design. If all of the patterns share the same group of decoders, this is a *global decoding* design. There is also an approach between per pattern and global decoding called *shared decoding* that divides the pattern set into multiple non-intersecting subsets and uses a distinct group of decoders for each subset.

## 3.3. Input History

Most FPGA pattern-matchers are designed to process an input stream by consuming a small number of characters per clock cycle. Therefore, in order to determine if a full pattern has been received from the input, the circuit must store some information about the sequence of characters received in the past. We call this information on previous inputs the input history. The input history also has multiple configuration options. The history can read its input directly from the input stream and store encoded character information (Figure 4), or it can read its input from the

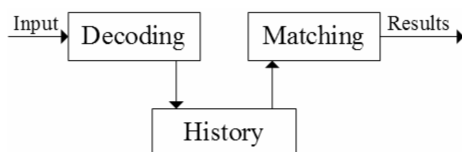


**Figure 3.** Decoded history design: decoding performed before history

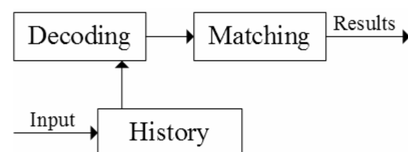


**Figure 4.** Encoded history design: decoding performed after history

character decoders and store decoded character information (Figure 3). In an *encoded history* design, the output of the history registers connects to character decoders, while in a *decoded history* design, the history outputs connect to pattern-match functions. With either design type, the history can be implemented in a *global*, *shared*, or *per pattern* manner.

### 3.4. Matching Functions

Every pattern in the pattern set will have an associated pattern-match function in the circuit. Each pattern-match function uses the information produced by the decoding and history components to determine if a character sequence has occurred in the input text stream that matches its associated pattern. The one-bit outputs of all the pattern-match functions form a result vector that indicates which patterns have been found.

## 4. Area-Delay Trade-Off

In this section, we discuss some delay and latency considerations. A pattern-matching circuit consists of an input bus, three types of functional components, and connections between these elements. The sources of circuit delay are the logic and routing delays within components and the routing delays of communication paths between components. In the following sections, we describe how our model handles these intra-component delay and inter-component delays.

In constructing our model, when faced with a design decision that involves a trade-off between delay and latency, we always opt to reduce delay and increase latency. This approach is in-line with the goals of most FPGA pattern-matching applications, which use an FPGA-based design specifically for its ability to provide higher throughput than other approaches (e.g. software). Adding a few cycles of latency to each pattern-matching operation to achieve a significant increase in throughput is generally preferred over a lower-latency, lower-throughput solution.

### 4.1. Intra-Component Delay

Logic delay within a component can be significant in architectures that use functions with a large number of inputs and multiple levels of LEs. However, since the output of each LE can be registered, the maximum logic delay can be limited to the delay of a single level of logic without any additional area usage. The extra latency introduced is relatively low because the number of levels of logic within a component is a logarithmic function of the number of inputs. Although some FPGA architectures allow unrelated logic to use the LUT and FF of an LE independently, this feature is rarely used by the design tools[1]. Therefore, we consider registering the output of each LE to be a low-cost method of minimizing logic delay, and we assume that this approach is used throughout the designs.

Routing is another source of intra-component delay. Within their components, all of the pattern-matching architectures in our model use only local point-to-point connections with no additional fan-out. In addition to the benefits of short, low fan-out connections, the effects of routing delay are further mitigated by the use of registered LE outputs that allows a larger portion of the clock period to be allocated to routing. Therefore, we assume that, under reasonable

[1] For example, with default settings, the MAP program in the Xilinx ISE package will not place unrelated logic in the same LE until over 99% of the chip's logic resources have been utilized.

logic utilization conditions (e.g. less than 90% total LE utilization), intra-component routing will not be on the critical path.

## 4.2. Inter-Component Delay

Inter-component connections include connections between the decoding, history, and matching components, as well as connections from the input bus to any of the components. The data path in all of the pattern-matching architectures consists of a sequence of connections from the input bus to the matching components. The elements on this path can be divided into two groups: a shared unit that is used by multiple patterns and a distributed set of per-pattern units. The contents of the units vary between architectures. The shared unit always includes the input bus, and each per-pattern unit always includes a match function. History and decoding components can be located in either group.

Since multiple sources in the shared unit connect to multiple destinations in the per-pattern unit, the wires between the shared and per-pattern units can span long distances and have high fan-out. The number of unique wires and the fan-out of each wire depend on the type of components being connected, and also on the properties of the pattern set. However, in all cases, the number, length, and fan-out of these wires increases as the number of patterns is increased. Thus, for most interesting pattern set sizes, the inter-component connection that links the shared and per-pattern units often becomes the critical path of the entire design.

## 4.3. Pipelined Wires

It is desirable to reduce the delay of the wires on the critical path in order to increase the clock frequency and the overall throughput of the circuit. One method of decreasing delay, at the cost of increased latency, is to pipeline the wires. Some FPGA pattern-matching implementations have applied this approach by using a tree of registers [10, 13]. The root of the tree is the signal source and each of the leaf registers drives multiple destination loads. The place-and-route software will automatically spread out the stages of the tree and place the final register stages near destination loads. This technique is effective at reducing delay for two reasons: (1) it decreases the distance a signal must travel in a single clock cycle, and (2) it reduces and localizes the fan-out of wires.

We have conducted experiments to demonstrate the relationship between fan-out and delay in FPGA devices. An FPGA routing network consists of multi-wire routing channels between each row and column of the LE array with programmable switchboxes at each intersection. We constructed simple circuit designs consisting of a single source register (S) connected to multiple destination registers that represent loads (L). Relative location constraints were applied to each register to force the circuits to have a layout similar to that of Figure 5. The use of a square layout of LEs with the source in the center provides the lowest achievable worst-case delay for a given fan-out.

The Xilinx ISE tools were used to compile designs with different numbers of loads for a Virtex II Pro 100 FPGA with a speed grade of -6. An optimistic target clock period goal of 1.0 ns was used to ensure that the tools would find the best possible routing configuration. The maximum path delay is plotted against fan-out load in Figure 6. The log-log plot clearly shows that delay increases as a logarithmic function of fan-out. This is the expected behavior and indi-

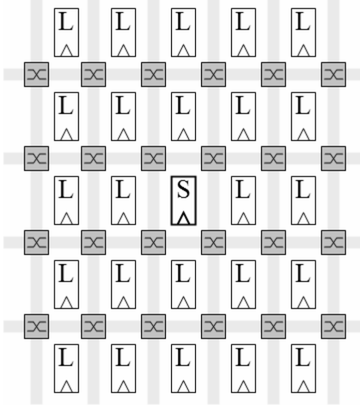


**Figure 5.** Fan-out test circuit using a $5 \times 5$ array of LEs

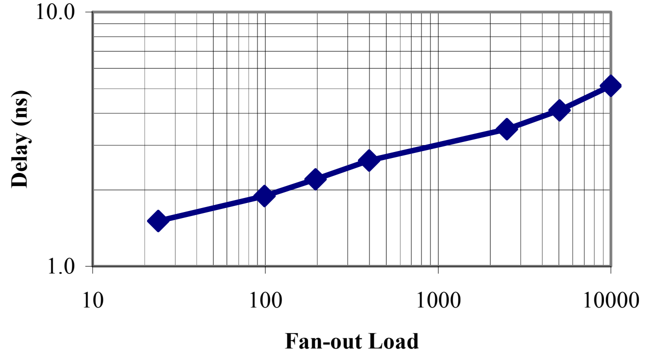


**Figure 6.** Path delay as a function of fan-out in Xilinx Virtex II Pro FPGA

cates that the switchboxes perform optimal buffering and repowering to control the transmission delay of a signal traversing multiple switches.

Given the results above, it is apparent that fan-out can be used as an estimator of delay in FPGA circuits. Thus, using a pipelined fan-out tree to reduce the fan-out of critical signals is an effective strategy for reducing critical path delay. This observation is the basis of an area-delay trade-off used by our model. When building a pattern-matching circuit, we first choose a value, $F_{Limit}$, which is the desired maximum fan-out after pipelining. Then, we determine the signal in the design with the highest fan-out, $F_{Max}$. Next, we calculate the fan-out tree depth required to reduce the fan-out of this signal to below the chosen limit using the formula

$$T_{Depth} = \left\lceil \log_{F_{Limit}} \left( F_{Max} \right) \right\rceil . \tag{6}$$

Then, for each signal in the critical path, we calculate the minimum branching factor that ensures the fan-out limit is not exceeded. The depth of all trees is kept the same to maintain synchronization. The branching factor for a signal with fan-out $F$ is

$$T_{Factor}(F) = \left\lceil \left( \frac{F}{F_{Limit}} \right)^{\frac{1}{T_{Depth}}} \right\rceil . \tag{7}$$

Finally, we calculate the number of nodes in the fan-out tree using

$$T_{Nodes}(T_{Factor}) = \sum_{k=0}^{T_{Depth}} (T_{Factor})^k . \tag{8}$$

Some researchers have proposed another approach to reduce the delay of critical inter-component connections. Both Baker et. al. [3] and Sourdis et. al. [14] partition the pattern set into multiple smaller subsets before generating the pattern-matching circuit. These designs use a separate shared unit for each subset and assign different per-pattern unit to each shared unit. There is some area overhead due to replication among the shared units, but a carefully-chosen pattern partitioning algorithm can minimize the overhead. This approach can reduce the length

and fan-out of the wires on the critical path, but the pipelining technique described above may still need to be applied to some wires to reach the desired delay value.

## 5. Pattern-Matching Architectures

Enumerating all combinations of the pattern-matching circuit component design parameters described above yields a large family of pattern-matching architectures. Actually, the design space is even larger because these parameters can be applied to two different implementation styles of pattern-matching circuits: brute-force designs and finite automata designs. The key distinction between the two design styles is in how the pattern-match functions are implemented. In a brute-force design, pattern-match functions perform comparisons of all pattern characters in each clock cycle. In a finite automata design, pattern-match functions are pipelined and only perform comparisons on the current input characters in each cycle. A finite automata pattern-match function is the next-state logic of a state machine that makes transitions in each cycle based on the current state and the current input characters. The state bits of the state machine serve as implicit history elements that track which characters of the pattern have been matched.

The following sections introduce the pattern-matching architectures that are described by our model. Several of the architectures that collectively cover all the design parameters are described and analyzed in detail. For each of these architectures, equations have been derived that predict circuit area requirements based on the parameters of the pattern set.

The total area ($A$) of any pattern-matching circuit is the sum of the areas used by its four components: character decoders ($A_D$), history registers ($A_H$), match functions ($A_M$), and fan-out trees ($A_F$).

$$A = A_D + A_H + A_M + A_F \tag{9}$$

The latency of a pattern-matching circuit is the number of clock cycles between the time when an input is changed and the time when the outputs reflect the change. The latency of a brute-force design is the sum of the latencies of its components.

$$L = L_D + L_H + L_M + L_F \tag{10}$$

### 5.1. Brute-Force Designs

All the possible configurations of history and decoding components for brute-force designs are listed in Table 1. The first column contains a shorthand notation that will be used to refer to the architectures for the remainder of this paper. There are six configurations (GdS, GdP, SeG, SdP, PeG, PeS) that are not feasible because their history and decoding styles conflict. References are provided for architectures with published implementations. Interestingly, there are several feasible architectures that have not been studied in the literature.

**Table 1.** Brute-force architectures

| Architecture | History | Decoding | Implementations |
|---|---|---|---|
| GeG | Global encoded | Global | *None* |
| GeS | Global encoded | Shared | *None* |
| GeP | Global encoded | per Pattern | *None* |
| GdG | Global decoded | Global | Baker [3], Sourdis [14] |
| *GdS* | *Global decoded* | *Shared* | — |
| *GdP* | *Global decoded* | *per Pattern* | — |
| *SeG* | *Shared encoded* | *Global* | — |
| SeS | Shared encoded | Shared | *None* |
| SeP | Shared encoded | per Pattern | *None* |
| SdG | Shared decoded | Global | *None* |
| SdS | Shared decoded | Shared | Baker [3], Sourdis [14] |
| *SdP* | *Shared decoded* | *per Pattern* | — |
| *PeG* | *per Pattern encoded* | *Global* | — |
| *PeS* | *per Pattern encoded* | *Shared* | — |
| PeP | per Pattern encoded | per Pattern | Sourdis [13] |
| PdG | per Pattern decoded | Global | *None* |
| PdS | per Pattern decoded | Shared | *None* |
| PdP | per Pattern decoded | per Pattern | *None* |

#### 5.1.1. PeP: Per Pattern Encoded History, Per Pattern Decoding

The PeP architecture is perhaps the most straightforward design. In this design, dedicated history, decoding, and matching components are allocated for each pattern, as depicted in Figure 7. A pipelined fan-out tree is used to distribute the input bus to each of the per-pattern units. Figure 8 shows the internal details of a per-pattern unit that processes two input characters per clock cycle and detects the three-character pattern "abc".

The encoded input history is formed by connecting each character from the input bus to a separate serial-in parallel-out (SIPO) shift register. A wire labeled $\alpha^c(t-s)$ carries a $b$-bit encoded character read from character position $c$ of the input word $s$ cycles ago. The length of each history shift register is a function of the pattern length, the input character position, and the number of input characters, and is found by

$$h_k^c = \left\lceil \frac{|x_k| + c}{N} \right\rceil . \tag{11}$$

The latency of a history component is equal to the length of its longest shift register. The total area used by the input history for all patterns is

$$A_H = \sum_{k=1}^{p} \sum_{c=0}^{N-1} \left[ h_k^c \times N \times A_r(b) \right] . \tag{12}$$

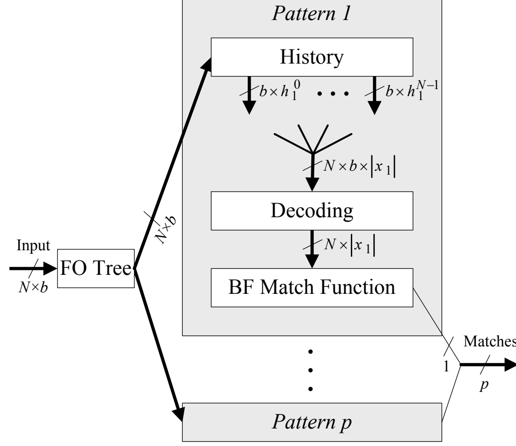


**Figure 7.** PeP architecture

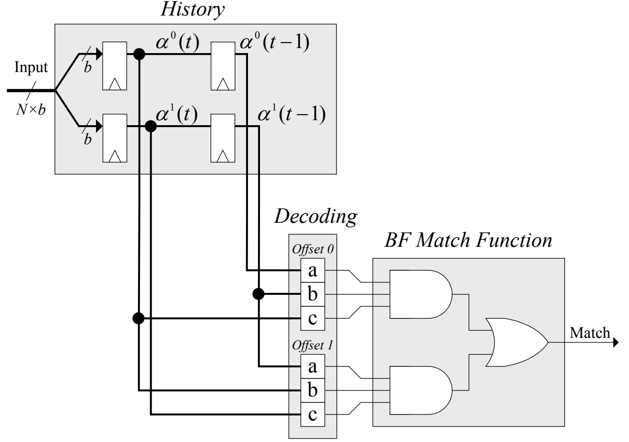


**Figure 8.** PeP circuit for pattern "abc" with $N = 2$

A per-pattern decoding component contains a character decoder for each character in the pattern for each of the $N$ starting position offsets. Thus, the total area used by character decoders is the sum of the area used by the decoding components associated with each pattern, and is equal to

$$A_D = \sum_{k=1}^{p} \left[ |x_k| \times N \times A_f(b) \right] . \tag{13}$$

A pattern-match function is a sum-of-products expression that determines if all the characters at any offset match the pattern. The area used by match functions for all patterns is

$$A_M = \sum_{k=1}^{p} \left[ A_f\left(|x_k|\right) \times N + A_f\left(N\right) \right] . \tag{14}$$

It is possible to reduce the total area used by the decoding and matching components by consolidating them. For each offset, the $|x_k|$ character decoders and the product term of the match function can be combined and computed using one AND operation of $|x_k| \times b$ bits. The sum term of the match function remains the same but now takes its inputs from the combined AND functions. With this design, the area used for decoding and matching for all patterns is

$$A_{D\&M} = \sum_{k=1}^{p} \left[ A_f\left(|x_k| \times b\right) \times N + A_f\left(N\right) \right] . \tag{15}$$

In the PeP architecture, the input bus is connected to every per-pattern unit, so its fan-out is equal to $p$. If $p$, which is $F_{Max}$, is greater than $F_{Limit}$, a pipelined fan-out tree is required, and its area usage is

$$A_F = T_{Nodes}\left(T_{Factor}\left(p\right)\right) \times A_r(N \times b) . \tag{16}$$

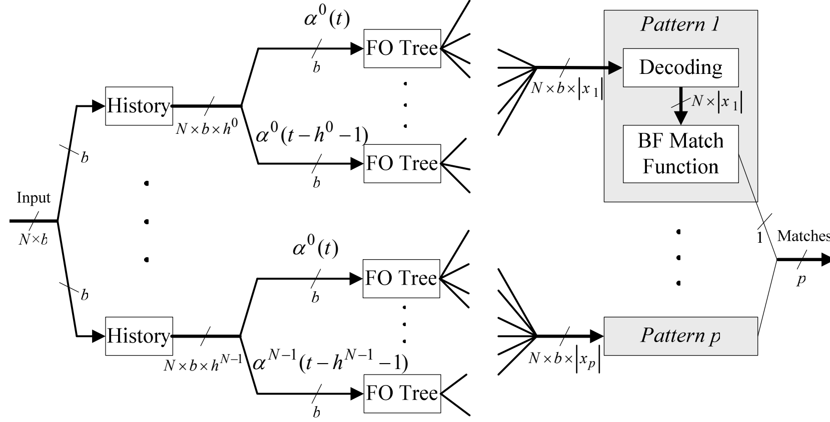


**Figure 9.** GeP architecture

### 5.1.2. GeP: Global Encoded History, Per Pattern Decoding

The GeP architecture is very similar to the PeP architecture, as shown in Figure 9. The main difference is that the per-pattern units do not have history components; instead, they all share $N$ global history components. The length of the SIPO history register for each input character position $c$ depends on the length of the longest pattern in $P$ and the number of input characters, and is computed by

$$h^c = \left\lceil \frac{\Lambda + c}{N} \right\rceil . \tag{17}$$

The total area used by the global history is

$$A_H = \sum_{c=0}^{N-1} \left[ h^c \times N \times A_r(b) \right] . \tag{18}$$

The area used by the decoding and matching components is the same as in the PeP architecture, and is found using (13) and (14), respectively. It is also possible to save area by using the combined decode and match functions, whose area is given by (15).

In this design, the shared unit includes a global encoded history and the critical delay component is due to connections from the history to the per-pattern decoding components. Each of the history registers may be connected to a different number of per-pattern units, so the fan-out of each will vary. For a history signal $\alpha^{c}(t-s)$, which represents the character that appeared on the input bus $s$ cycles ago in position $c$, the fan-out is determined by counting the number of its usages according to (19).

$$F^c(s) = \begin{cases} \sum_{j=0}^{N-1-c} \ell(N-c-j) & \text{, if } s = 0 \\ \sum_{j=0}^{N-1} \ell((N \times s) + N - c - j) & \text{, if } s > 0 \end{cases} \tag{19}$$

This equation uses a function, $\ell(n)$, which returns the number of patterns in the pattern set with length equal to or greater than *n*. Since the inputs to all of the per-pattern decoding components are aligned to the beginning of the history, the outputs of the history registers closer to the beginning will have higher fan-out than those towards the end of the shift registers. The maximum fan-out, $F_{Max}$, of any register is *p*, and this will determine the depth of the fan-out trees associated with all history registers. The total area used by all fan-out trees in the design is calculated by summing the sizes of each using (20).

$$A_F = \sum_{c=0}^{N-1} \sum_{s=0}^{h-1} \left[ T_{Nodes}\left(T_{Factor}\left(F^c(s)\right)\right) \times A_r(b) \right] \tag{20}$$

### 5.1.3. GdG: Global Decoded History, Global Decoding

The GdG architecture is depicted in Figure 10. In global decoding designs, such as GdG, each of the *N* characters from the input bus is connected to a different group of Ω character decoders with one *b*-bit decoder for each of the unique characters used in the pattern set. The total area used by character decoders in the GdG design is

$$A_D = \Omega \times N \times A_f(b) \ . \tag{21}$$

The input history is implemented as a one-bit SIPO shift register attached to each decoder output. The lengths of the history registers vary based on the positions within the patterns where each character occurs. The length of the longest register is determined by the longest pattern in the set as follows,

$$h_{Max} = \left\lceil \frac{\Lambda + N - 1}{N} \right\rceil \ . \tag{22}$$

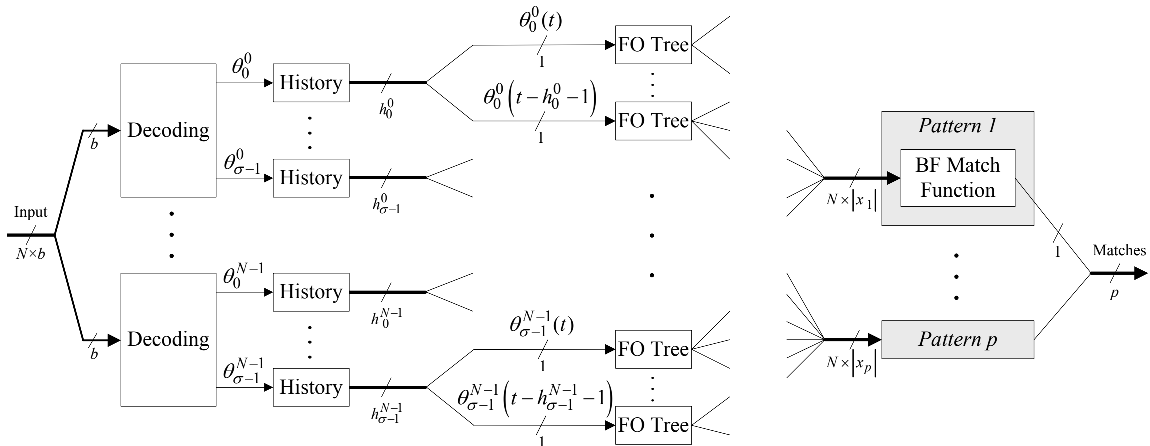


**Figure 10.** GdG architecture

The length of the history register for a decoder output $\theta_k^c$ is the maximum history position with non-zero fan-out and is found using equations (23), (24), and (26).

$$h_k^c = \max_{s=0}^{h_{Max}-1} \Phi_k^c(s) \tag{23}$$

$$\Phi_k^c(s) = \begin{cases} 0 & \text{, if } F_k^c(s) = 0 \\ s+1 & \text{, if } F_k^c(s) > 0 \end{cases} \tag{24}$$

The total area used by the input history is determined by summing the area of the registers for each decoder output using

$$A_H = \sum_{c=0}^{N-1} \sum_{k=0}^{\sigma-1} \left[ h_k^c \times A_r(1) \right]. \tag{25}$$

A history output wire labeled $\theta_k^c(t-s)$ carries a signal that indicates whether or not the character read from the input text stream $s$ cycles ago in character position $c$ of the input word was the character $\theta_k$.

In the GdG design, the critical path arises from the non-local wires connecting the decoded history signals to the per-pattern match functions distributed throughout the logic array. For a history signal $\theta_k^c(t-s)$ the fan-out is given by (26).

$$F_k^c(s) = \begin{cases} \sum_{j=0}^{N-1-c} \Theta_k(N-c-j) & \text{, if } s = 0 \\ \sum_{j=0}^{N-1} \Theta_k((N \times s) + N - c - j) & \text{, if } s > 0 \end{cases} \tag{26}$$

The signal with the maximum fan-out is determined by evaluating (26) for each combination of offset, character value, and history position and finding the maximum value using (27).

$$F_{Max} = \max_{c=0}^{N-1} \max_{k=0}^{\sigma-1} \max_{s=0}^{h_{Max}-1} F_k^c(s) \tag{27}$$

The total area used by all fan-out trees in the design is calculated by summing the sizes of each according to (28).

$$A_F = \sum_{c=0}^{N-1} \sum_{k=0}^{\sigma-1} \sum_{s=0}^{h_k-1} \left[ T_{Nodes}\left(T_{Factor}\left(F_k^c(s)\right)\right) \times A_r(1) \right] \tag{28}$$

The inputs to each pattern-match function are taken from the appropriate history register fan-out tree output. The number of inputs to the function for a pattern $x$ is equal to the length of the pattern times $N$. The brute-force pattern-match function is the same as that used in the previously described brute-force designs, and thus, the total area used by the pattern-match functions is given by (14).

**Table 2.** Finite automata pattern-matching architectures

| Architecture | History | Decoding | Implementations |
|---|---|---|---|
| GfG | Global FA | Global | *None* |
| *GfS* | *Global FA* | *Shared* | — |
| GfP | Global FA | per Pattern | Sidhu [12], Franklin [10] |
| SfG | Shared FA | Global | Clark [7] *(in prefix tree)* |
| SfS | Shared FA | Shared | *None* |
| SfP | Shared FA | per Pattern | Franklin [10] *(in prefix tree)* |
| PfG | per Pattern FA | Global | Cho [5], Clark [7, 8] |
| PfS | per Pattern FA | Shared | *None* |
| PfP | per Pattern FA | per Pattern | Cho [6], Clark [7], Moscola [11] |

## 5.2. Finite Automata Designs

All the possible configurations of history and decoding components for finite automata (FA) designs are listed in Table 2. There is one design (GfS) that is not feasible because its history and decoding styles conflict (i.e. shared decoding means that the patterns are divided into separate groups, while a global FA requires a single group of patterns). In FA designs, there is no notion of encoded or decoded history because the state bits of the state machine serve as implicit history elements. A history based on a global FA implies that all of the patterns are compared against the input text stream using a single FA. A design using a global FA will detect a match of any pattern, but it will not be able to indicate which particular pattern has been matched, making it undesirable for some applications. Similarly, a design that groups patterns into sets and uses an FA to compare each set will only be able to indicate to which set a matching pattern belongs. A common use of a shared FA history is as part of a prefix tree. In a prefix tree design, patterns with common prefixes are grouped together, and a shared FA is used to match each prefix, reducing redundant logic. The match output of the FA for a prefix is connected to the input of multiple per-pattern FA that match the suffixes of each pattern.

There are two types of finite automata: non-deterministic finite automata (NFA) and deterministic finite automata (DFA). NFA-based pattern-matching circuits have been implemented in [5-8, 10, 12, 15], and DFA-based circuits have been implemented in [11]. In this paper, only the more common NFA approach is discussed. The analysis of DFA circuits would be similar, but the implementation of the match function would be different.

### 5.2.1. PfP: Per Pattern FA History, Per Pattern Decoding

In the PfP design, as depicted in Figure 11, there is no logic in the shared unit. Each input character $\alpha^{c}(t)$ must be broadcast to a character decoder associated with every pattern character, so its fan-out is equal to *M*. There are *N* identical fan-out trees used to distribute the input characters to the per-pattern units. The area used by these trees of registers is

$$A_F = T_{Nodes}\left(T_{Factor}\left(M\right)\right) \times A_r(b) \times N \quad (29)$$

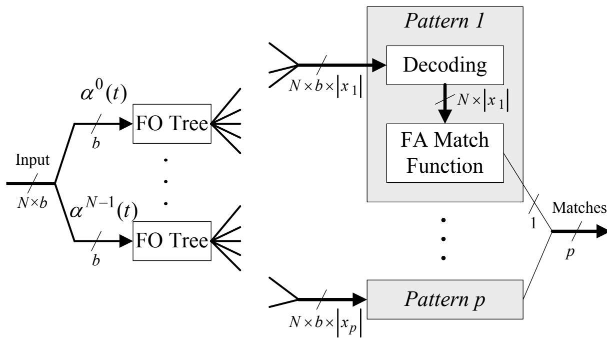


**Figure 11.** PfP architecture

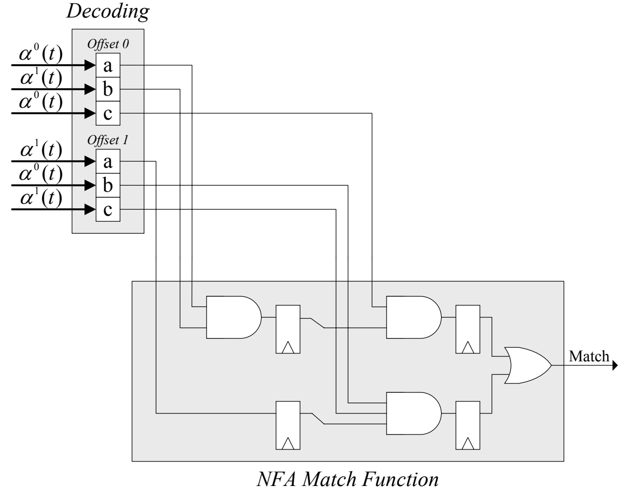


**Figure 12.** PfP circuit for pattern "abc" with $N = 2$

The PfP design uses per-pattern decoding and includes a character decoder for each pattern character at each offset, so the total area used by decoders is

$$A_D = M \times N \times A_f\left(b\right) \ . \tag{30}$$

The finite automata pattern-match functions used in this design consist of state bits and next-state logic. The NFA state machines are constructed as pipelines with one state bit per state and allow multiple active states. The number of states, or pipeline stages, in an NFA match function for a pattern of length $m$ is

$$S\left(m\right) = \left\lceil \frac{m + N - 1}{N} \right\rceil \ . \tag{31}$$

A next-state AND function is associated with each pipeline stage as shown in Figure 12. The inputs to this function are the output of the state register from the previous stage (if any) and the outputs of up to $N$ character decoders. The number of inputs to the next-state function for a given offset and stage is calculated using (32).

$$\eta(offset, stage) = \begin{cases} \min\left(N - offset,\ \left|x_k\right|\right) & \text{, if } stage = 0 \\ 1 + \min\left(N,\ \left|x_k\right| + offset - \left(N \times stage\right)\right) & \text{, if } 0 < stage \le \dfrac{\left|x_k\right| + offset}{N} \\ 0 & \text{, if } stage > \dfrac{\left|x_k\right| + offset}{N} \end{cases} \tag{32}$$

The output of the last level of logic in each next-state function is registered and serves as a state bit. For every pattern, an NFA state machine is instantiated for each of $N$ offsets, and an $N$-input

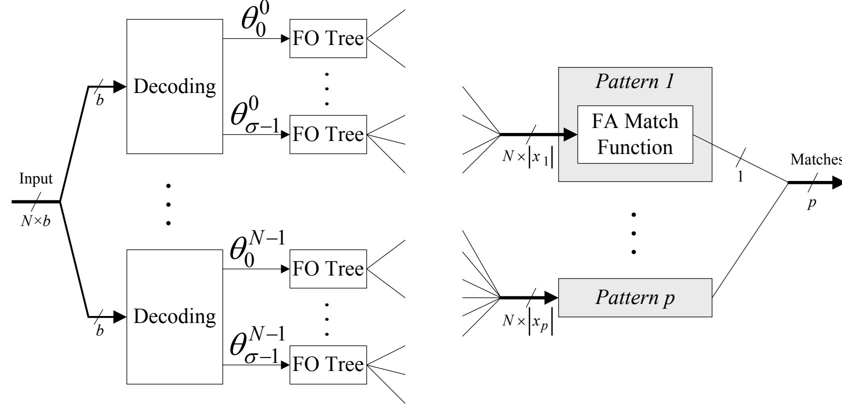


**Figure 13.** PfG architecture

OR function determines if any of the NFA reach their final state, indicating a pattern match. The total area used by NFA match functions for all patterns is

$$A_M = \sum_{k=1}^{p} \left[ \left( \sum_{c=0}^{N-1} \sum_{s=0}^{S(|x_k|)-1} A_f\left(\eta(c,s)\right) \right) + A_f(N) \right] . \tag{33}$$

### 5.2.2. PfG: Per Pattern FA History, Global Decoding

The PfG design (Figure 13) uses the same NFA pattern match functions as the PfP design, and their area usage is given by (33). Unlike the PfP design, the PfG design uses global decoders shared by all patterns. Therefore, the total area used by decoders in this design is independent of the number of characters in the pattern and is a function of the number of unique alphabet characters used as indicated by

$$A_D = \Omega \times N \times A_f(b) . \tag{34}$$

The output of the decoders is distributed to the appropriate stages of the per-pattern NFA match functions. For a decoder output $\theta_k^c$, the fan-out is equal to $\Theta_k$, and fan-out trees are used on these signals. The depth of each tree is determined by the maximum $\Theta_k$ for all *k*. The total area used by all fan-out trees is

$$A_F = \sum_{k=0}^{\sigma-1} \left[ T_{Nodes}\left(T_{Factor}(\Theta_k)\right) \times A_r(1) \times N \right] . \tag{35}$$

## References


[1] "Stratix Devices," Altera Corporation, http://www.altera.com/products/devices/stratix/stx-index.jsp

[2] "Xilinx: Virtex Series," Xilinx, Inc., http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/index.htm

[3] Z. K. Baker and V. K. Prasanna, "A Methodology for Synthesis of Efficient Intrusion Detection Systems on FPGAs," In Proceedings of *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 135-144, 2004.

[4] Z. K. Baker and V. K. Prasanna, "Time and Area Efficient Pattern Matching on FPGAs," In Proceedings of *ACM International Symposium on Field Programmable Gate Arrays (FPGA)*, pp. 223-232, 2004.

[5] Y. H. Cho and W. H. Mangione-Smith, "Deep Packet Filter with Dedicated Logic and Read Only Memories," In Proceedings of *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 125-134, 2004.

[6] Y. H. Cho, S. Navab, and W. H. Mangione-Smith, "Specialized Hardware for Deep Network Packet Filtering," In Proceedings of *International Conference on Field Programmable Logic and Applications (FPL)*, pp. 452-461, 2002.

[7] C. R. Clark and D. E. Schimmel, "Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns," In Proceedings of *International Conference on Field Programmable Logic and Applications (FPL)*, pp. 956-959, 2003.

[8] C. R. Clark and D. E. Schimmel, "Scalable Pattern Matching for High-Speed Networks," In Proceedings of *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 249-257, 2004.

[9] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. W. Lockwood, "Deep Packet Inspection Using Parallel Bloom Filters," In Proceedings of *Symposium on High Performance Interconnects (HotI)*, pp. 44-51, 2003.

[10] R. Franklin, D. Carver, and B. L. Hutchings, "Assisting Network Intrusion Detection with Reconfigurable Hardware," In Proceedings of *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 111-120, 2002.

[11] J. Moscola, J. W. Lockwood, R. P. Loui, and M. Pachos, "Implementation of a Content-Scanning Module for an Internet Firewall," In Proceedings of *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 31-38, 2003.

[12] R. Sidhu and V. K. Prasanna, "Fast Regular Expression Matching using FPGAs," In Proceedings of *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2001.

[13] I. Sourdis and D. Pnevmatikatos, "Fast, Large-Scale String Match for a 10 Gbps FPGA-based Network Intrusion Detection System," In Proceedings of *International Conference on Field Programmable Logic and Applications (FPL)*, pp. 880-889, 2003.

[14] I. Sourdis and D. Pnevmatikatos, "Pre-Decoded CAMs for Efficient and High-Speed NIDS Pattern Matching," In Proceedings of *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 258-267, 2004.

[15] P. Sutton, "Partial Character Decoding for Improved Regular Expression Matching in FPGAs," In Proceedings of *IEEE International Conference on Field-Programmable Technology (FPT)*, pp. 25-32, 2004.