Signature Match Processor Network Processing Applications

by

Janardhan Singaraju

B.Tech., Nagarjuna University, 2003

A Thesis

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

at the

University of Connecticut

2006

APPROVAL PAGE

Master of Science Thesis

Signature Match Processor Network Processing Applications

Presented by
Janardhan Singaraju, B.Tech.

Major Advisor _____
John A. Chandy

Associate Advisor _____
Yunsi Fei

Associate Advisor _____
Jun-Hong Cui

University of Connecticut
2006

Contents

List of Tables

List of Figures

Abstract

Signature matching is a key problem in many network processing applications. Current implementations of this process using software are time consuming and can not meet gigabit bandwidth requirements. Implementing this process in hardware improves the search time drastically and has several other advantages. This thesis aims at a CAM based hardware implementation of this time consuming process for network intrusion detection and directory lookup applications using reconfigurable hardware. These designs are coded in VHDL targeting a Xilinx Virtex- II-P FPGA board with embedded Power PC processor and are evaluated in terms of the speed and resource utilization.

Chapter 1

Introduction

Many network processing applications require frequent use of string matching to find the presence of a keyword. This signature matching process is currently implemented in software. There are several software algorithms for string matching, but the slowness of these implementations creates a bottleneck for growing gigabit network configurations. A hardware implementation can speed up the signature matching process considerably. This thesis gives two similar architectures for the hardware based implementation of data matching for a network intrusion detector and for generic lookup cache applications.

These architectures are flexible enough to handle variable sized keys as well as provide a mechanism to do mapping in addition to string matching. The architecture 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. The key difference with Motomura's design is the ability to handle varying sized keywords without grouping and the ability to do key-to-value mapping. This architecture is also compatible with further optimizations like processing characters in parallel, prefix sharing, pattern partitioning etc to improve

the performance. The compactness of each cellular automata element leads to a highly efficient design in terms of both area and speed.

The rest of the thesis is organized as follows. The following chapter presents background on string matching in software and hardware and a brief description of the basics of FPGA technology. Details of the architecture and working of signature match processor for network intrusion detection and generic lookup cache are explained in the following chapters.

Chapter 2

Related Work

2.1 String Matching

String matching is the process of finding out if a given string or pattern is present in the data we have. String matching is a very common problem we encounter in the computing world. We often search our computer for a specific file or we search some text document for a specific keyword. There are different types of string matching like exact string matching which as the name suggests looks for the presence of the exact pattern in the search area. String matching with errors or approximate string matching looks for a closest match for the input string in the search area.

String matching has many applications. The use of string matching in databases and search engines is common. If we need to retrieve any data from the database, we often will query the database using a string as the key. String matching is also the underlying principle in the working of search engines. String matching is also a key component in many network processing applications like network intrusion detection and cache-lookup applications. Network intrusion detection is performed by inspecting all the incoming data from the network. We look for a specific keyword in form of a computer command or content in the data packet or an IP addresses. Lookup cache applications for network processing also involve string matching. Both string matching for network intrusion detection and generic look-up cache applications

involves exact pattern matching.

2.1.1 String Matching Software Algorithms

There are many algorithms that deal with string matching including the Rabin Karp algorithm [**?**], Knuth-Morris-Pratt [**?**] algorithm and the Boyer-Moore algorithm [**?**]. Given an input string and a long data string in which we need to find the presence of input string, the Rabin-Karp string searching algorithm uses hashing functions to find the string match. This algorithm tries to find the match by comparing the hashes of the input string and substrings of the data string thus avoiding character by character matching. The success of this algorithm however depends on the efficient hashing function. The hashes should be efficiently calculative and easily calculative from identical strings. For example, let the input string be 135 and let the data string be 131356. Let the hash be calculated by modulo 13. The hash for 135 is 5, and the hashes calculated for 131, 313, 135, and 357 are 1, 1, 5, and 5 respectively. Hashes for 313, 135, and 357 can be easily calculated from hash for 131 by simple mathematical operations. The hash of the input string is compared to the corresponding hashes of the substrings of the data string. Once there is a match an additional character by character match reveals the exact string match. In the pre processing stage, the hash values are calculated taking time complexity of $O(m)$, where m is the length of the input string. Finding the hashes of all the other substrings of the data string of length n is a constant time operation. In the search phase the hash value of the input string is compared to $n - m$ substrings of the data string. Once a match is found an additional character by character comparison is to be done. The search phase takes worst case complexity of $O(nm)$ time.

The Knuth-Morris-Pratt(KMP) algorithm matches the input string with data string to be searched character by character. Once there is a mismatch the algorithm slides the data string by the mismatch length and continues searching. For example let

us suppose we are looking for a string "*bat*" in the search space "*babat*". Initially the algorithm checks the first character "*b*" for which there is a match. Then it looks for next character "*a*" for which there is a match. Now it checks for the third character "*b*" which is a mismatch to the character "*t*" in the input string. So the algorithm slides the data string by two characters and start searching from the third character "*b*" in the data string. KMP algorithm compares input string and data string character by character and omits some comparison when ever there is a mismatch as opposed to the brute force string search algorithm which compares all characters in the input and data string. In KMP the overall time to find a match is $O(m+n)$ where m is the length of input string or the keyword and n is the length of data string or the search space.

Boyer-Moore is typically faster than the Knuth-Morris-Pratt method. It differs from the latter in the way the input to the FSM is given. The algorithm scans characters from the right to left. In case of a mismatch, the algorithm shifts the data string to the right based on two pre-computed functions good suffix shift and bad character shift. If there is a mismatch the data string is shifted in such a way that the mismatched character aligns with one of the character in the input string if present. Otherwise, the data string is shifted by the entire length of the input string and the algorithm again starts looking for a match. The advantage of this method is that we are eliminating the unnecessary character by character comparisons. By starting comparison from right to left, if the right most characters give a mismatch checking the characters to the left of mismatched character is useless, so we shift the data string in such a way that it aligns with one of the character in the input string if present or we shift the data string by the length of key word. Again we check the character match for the right most character, thus eliminating unnecessary matches. The time to do the string match varies from a best time of $O(m/n)$ to a worst case time of $O(mn)$ with an average match time of $O(n)$. Other algorithms on string

matching represent the data string in a tree like structure and look in this tree for the presence of input string. The search time is drastically reduced in these algorithms but these algorithms require the preprocessing of the search space in to a tree like structure.

2.1.2 Hardware vs. Software

Signature match processor can be implemented both as software program or hardware design in FPGA. Software comes in the form of programming code that is used to manipulate registers and gates. Hardware is a micro-controller or microprocessor on-chip peripheral, off-chip device, or component. To look at some of the differences between hardware and software implementation of signature match processor, Signature matching is the most computationally intensive process of the intrusion detection process. NIDS spend more than 30 % of the computational time in this process. Software implementation of signature matching is very slow. With larger number of packets to be processed due to heavier network traffic, the situation is worsened.

However, hardware implementation is much faster. To illustrate this say for example we want to implement a simple operation $C = A + B$. In a software implementation, a higher language program is written to implement the above operation. This program would then be converted into a series of micro operations to be performed on microprocessor shown below.

Load R1, A

ADD R3, R1,B

Store R3, C

This implementation takes three cycles to complete. If we implement the same

operation in an *ASIC*, a binary adder would implement A+B in a single cycle.

Software implementation of signature matching puts a lot of burden on the host computer. The host computer can use these resources elsewhere. With hardware implementation using FPGA, all the signature match process is done in the FPGA and there will not be any burden on the systems hardware resources.

Software implementation has some advantages over hardware approach. In software design it is more flexible to change the design at any stage. It is easy to modify the code and obtain a new functionality from the code. The time compile the new design is little. It is easier to design and implement in software, there are no restrictions in implementing the design. Any behavioral process can be implemented in software. In a hardware design process, changing the design is not easy. The time to synthesize the new design can vary from few hours to few days. There are also some restrictions in the design process, not every behavioral model can be synthesized in to a physical circuit. Implementing these models in hardware will be a challenge. Debugging a software program is easier due to availability of lot of debugging tools and ease of insertion of debugging statements in to the program. The overall time to design and implement a software project is little compared to a hardware project.

2.1.3 Hardware String Matching

Hardware based string matching is preferred over a software implementation for data-intensive applications. It is used in applications like data mining that require full text searching. For example, the Mercury System [**?**] is a prototype data mining engine that uses a *shift and add* [**?**, **?**] algorithm for exact string matching extended for handling of mismatches. Hardware based string matching is commonly used for IP lookup in routers.

Lot of research on hardware based text search or string matching has been made by the FPGA community particularly with respect to network intrusion detection.

Previous approaches to string matching in FPGAs have included finite automata methods [?, ?, ?, ?]. Finite automata methods are often used to represent regular expression signatures into hardware implementations. In these methods the regular expressions are represented as finite automata graphs and then these graphs are translated to FPGA circuitry. These translations are very complex, leading to large amounts of logic circuitry consumption and more area on an FPGA. The main disadvantage of these type of methods are that whenever there are changes in the keyword set, the regular expressions have to be calculated again and the FPGA has to be reprogrammed .

The more practical approach to string matching is the use of CAMs [?, ?, ?]. Content addressable memories have long been used for fast string matching against multiple keywords. The use of content addressable memories for text search has centered around two strategies: cellular automata [?, ?, ?] and finite state machines [?, ?]. The cellular automata methods cascade CAMs temporally and then use pointers to propagate matches from one CAM set to another [?]. Motomura et al. have described a cellular automata based architecture to do dictionary search in VLSI [?]. The keywords are grouped in 4 character segments and they allow for concatenating groups to create larger keywords. Hirata et al. have designed an FSM-based CAM search architecture that also accommodate variable length keywords by grouping keywords into segments. These CAM-based designs are limited to string matching applications where the only requirement is to detect the presence of the string in a dictionary. This is the case with network intrusion detection, where the task is to determine if an incoming packet contains a signature from a dictionary. As such, several FPGA based network intrusion detection systems have used variations of these CAM-based designs [?, ?, ?, ?, ?]. However, in mapping tasks such as LDAP or DNS querying, apart from word match a translation to an ID or data structure is required.

2.2 FPGA (Field Programmable Gate Array)

FPGAs have emerged as an attractive means of implementing logic circuits. FPGAs also provide a cheap and fast way for implementing VLSI circuits over a wide variety of applications. They also have opened up new possibilities in designing reconfigurable digital systems. Chips with more than million transistors can be realized in a VLSI circuit using full custom approach. In a full custom design all parts of the VLSI circuit are carefully tailored to meet a set of specific requirements. The approach however requires extensive manufacturing efforts taking several months. The result is high cost unless manufactured in large volumes. Certain products that need the shortest time to enter the market and are sensitive to financial considerations in development cannot afford the full custom approach. FPGAs have emerged as the ultimate solution to market and risk problems.

FPGAs can be used in almost all digital applications. They are used as a replacement for ASICS for implementing digital logic; for example, FIFO controllers, DRAM controller and other telecommunication applications. FPGAs can also be used as replacements to several SSI chips resulting in substantial area reduction on circuit boards. FPGAs are ideally useful in prototyping applications. The quick and low cost implementation of logic circuits and the reconfigurable nature of FPGAs makes them useful in prototyping complex ASICS. A whole new class of computers can be constructed using in-circuit reprogrammable FPGAs. The present FPGAs with embedded processors make them possible.

FPGAs arrived as an alternative to PLDs in 1984. They consist of a two dimensional array of programmable blocks called configurable logic blocks (CLB). There are horizontal routing channels between rows of blocks and vertical routing channels between columns connecting all the CLBs. Figure **??** [**?**] shows the diagram of FPGA. It shows a two dimensional array of logic blocks that are connected through programmable interconnect. Some FPGAs can be programmed only once. These

devices employ anti-fuse technology. Other FPGAs use flash based technologies that allow these FPGAs to be programmed and reprogrammed. Modern FPGAs have become more advanced with more features supported internally. These features include clock management for decreasing the clock delays, embedded memory to configure the FPGA faster using stripes of memory distributed throughout the FPGA, Embedded processors. Most modern FPGAs have processors embedded in the chip to implement any user code in C or C++ for software hardware co design. Other features include high speed I/O for high speed data communication.

2.2.1 Programming Technologies

2.2.1.1 Anti-fuse Technology

The basic principle of operation of FPGAs using **antifuse** is that they use an antifuse to program the connection between different CLBs and routing channels. The material used in different FPGAs differs. Antifuse normally is in high impedance state but when high voltage is applied, the high impedance state can be turned to a fused state thereby establishing an electrical conductive path.

PLICE, a type of antifuse employed in some Actel FPGAs. It consists of three layers. The bottom layer composed of positively doped Silicon ($n+$ diffusion), the middle layer is a dielectric (oxygen-nitrogen-oxygen insulator) and the top layer is poly-silicon. It is shown in Figure **??** [**?**]

PLICE is programmed by placing a voltage of 18V across anti-fuse terminals and driving a current of about 5ma through the device. This generates enough heat in the dielectric to cause a melt down and form a conductive link between the poly-silicon and $n+$ diffusion as shown in the figure. Both the bottom layer and top layer of the antifuse are connected to the metal wires so that when programmed, a conductive link between the metal layers is established as shown in Figure **??** [**?**].

The PLICE antifuse is manufactured by adding three specialized masks to normal

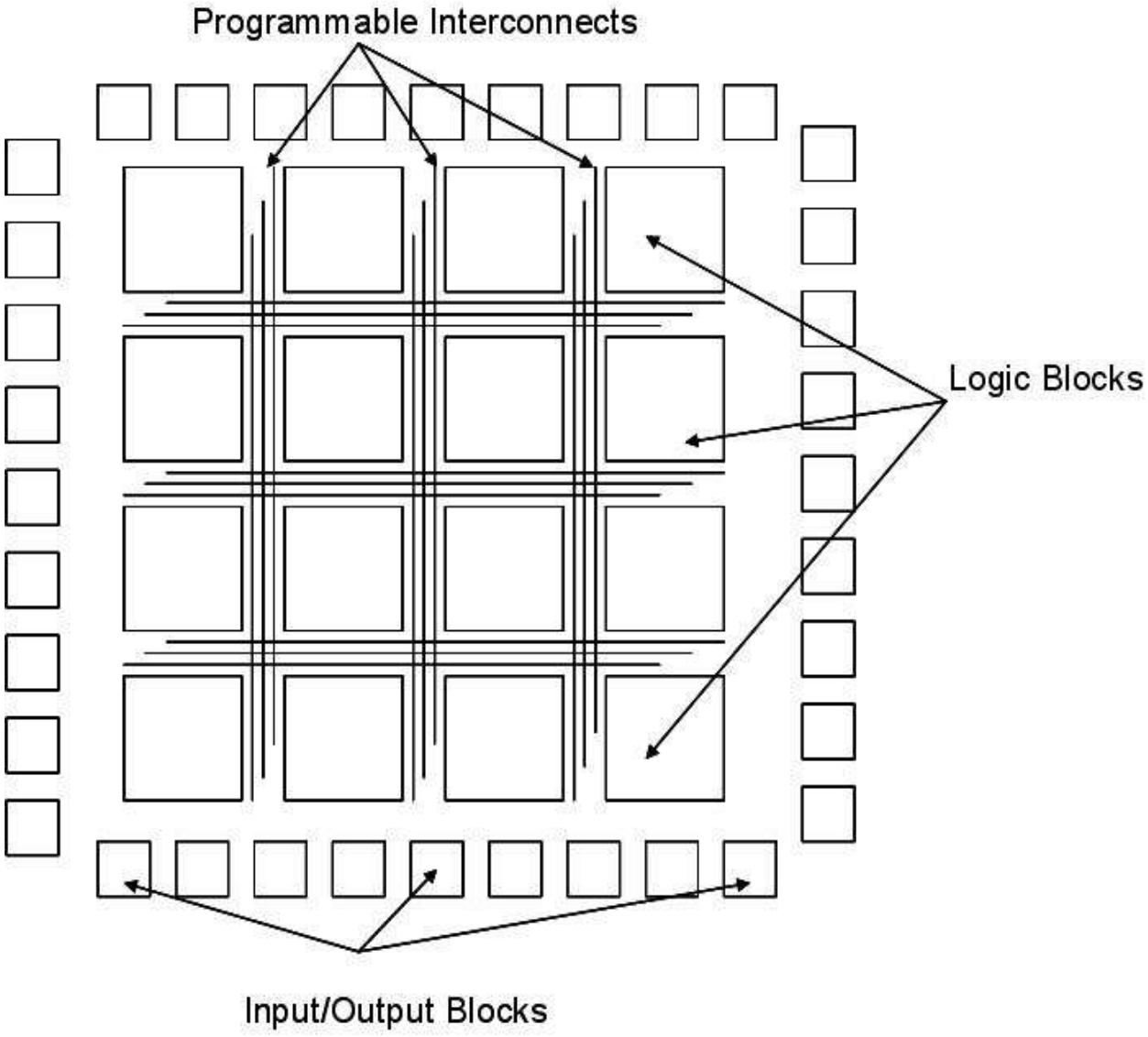

Figure 2.1: FPGA Block [?]

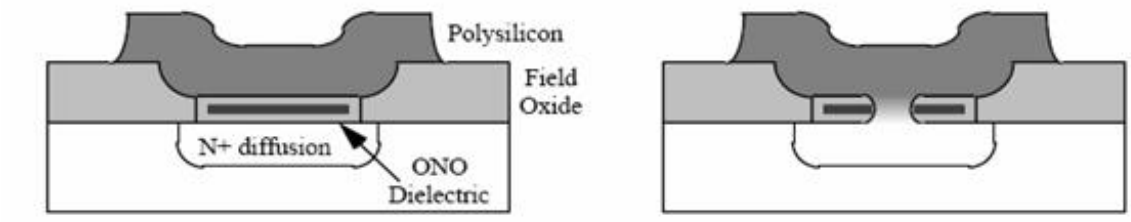

Figure 2.2: PLICE: Programmable Low Impedance Circuit Element [?]

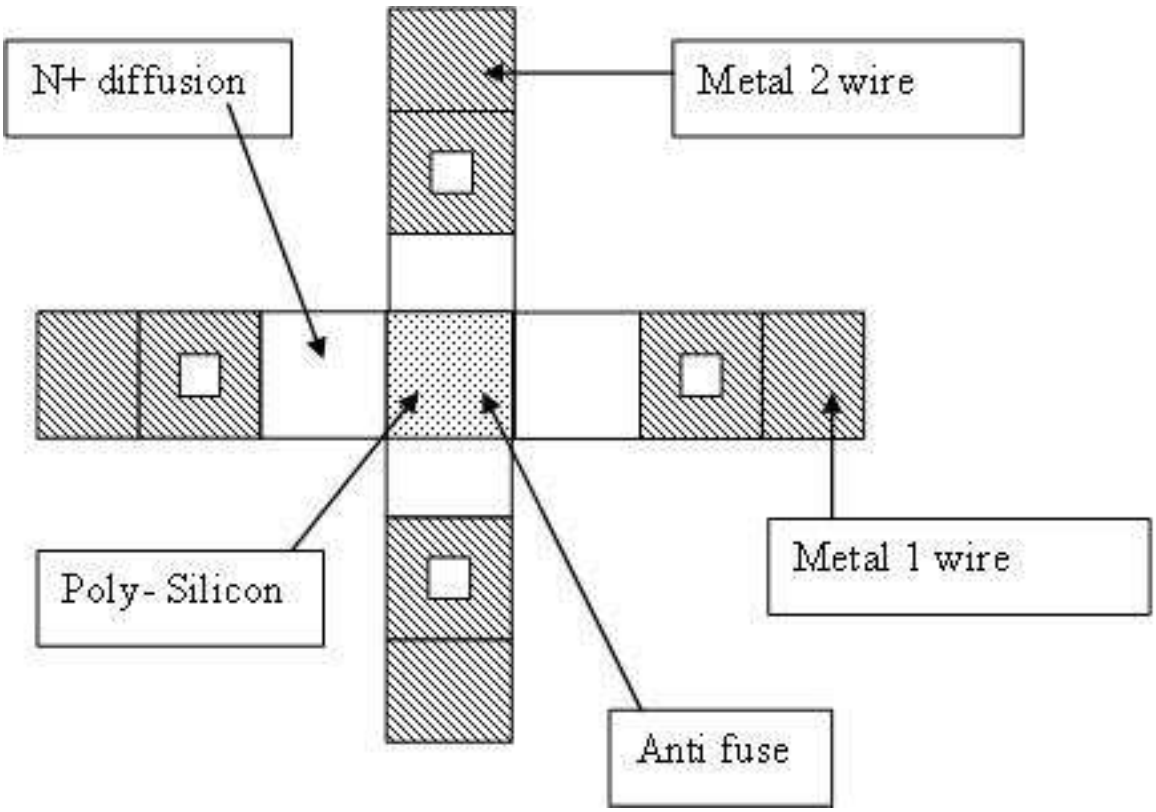

Figure 2.3: PLICE - Structure [?]

CMOS process. The chips required by the antifuse element are very small compared to other technologies. The main disadvantages of antifuse based FPGAs are that they require modifications to the basic CMOS transistors, e.g. EP-ROM transistors.

An EP-ROM transistor consists of two gates, a regular gate or select gate and a floating gate. The floating gate is placed between the select gate and transistors channel. This gate is electrically not connected to any circuitry. In the normal state no charge exists on the floating gate and transistor can be turned on as a regular CMOS gate using the select gate. However when the transistor is programmed, a large current flows between the source and drain trapping some charge under the floating gate. This trapped charge has effect of turning off the transistor, thus EP-ROM transistor functions as programmable element. EP-ROM transistor can be reprogrammed by removing the trapped charge under the floating gate. The main disadvantage of the antifuse technology based FPGAs are that they are not in-system programmable. When the FPGA is configured to a design, it can not be modified without reconfiguring the entire circuit. Antifuse based FPGAs are programmed offline using a device programmer. Once the chip is configured it cannot be altered. However in antifuse technology device configuration is non volatile and needs no external memory. This results in a very small chip area when fabricated.

2.2.1.2 Static RAM Programming Technology

SRAM programming technology is popular in several FPGAs manufactured by XILINX. These transistors are good for prototyping. Most modern FPGAs are based on SRAM configuration cells. These FPGAs offer unlimited re-programmability. In these FPGAs programmable connections are made using pass transistors or multiplexers or LUTs that are controlled by external memory in SRAM cells. This control process is illustrated in Figure **??**.

A RAM cell controls the connection between the routing wires using pass transis-

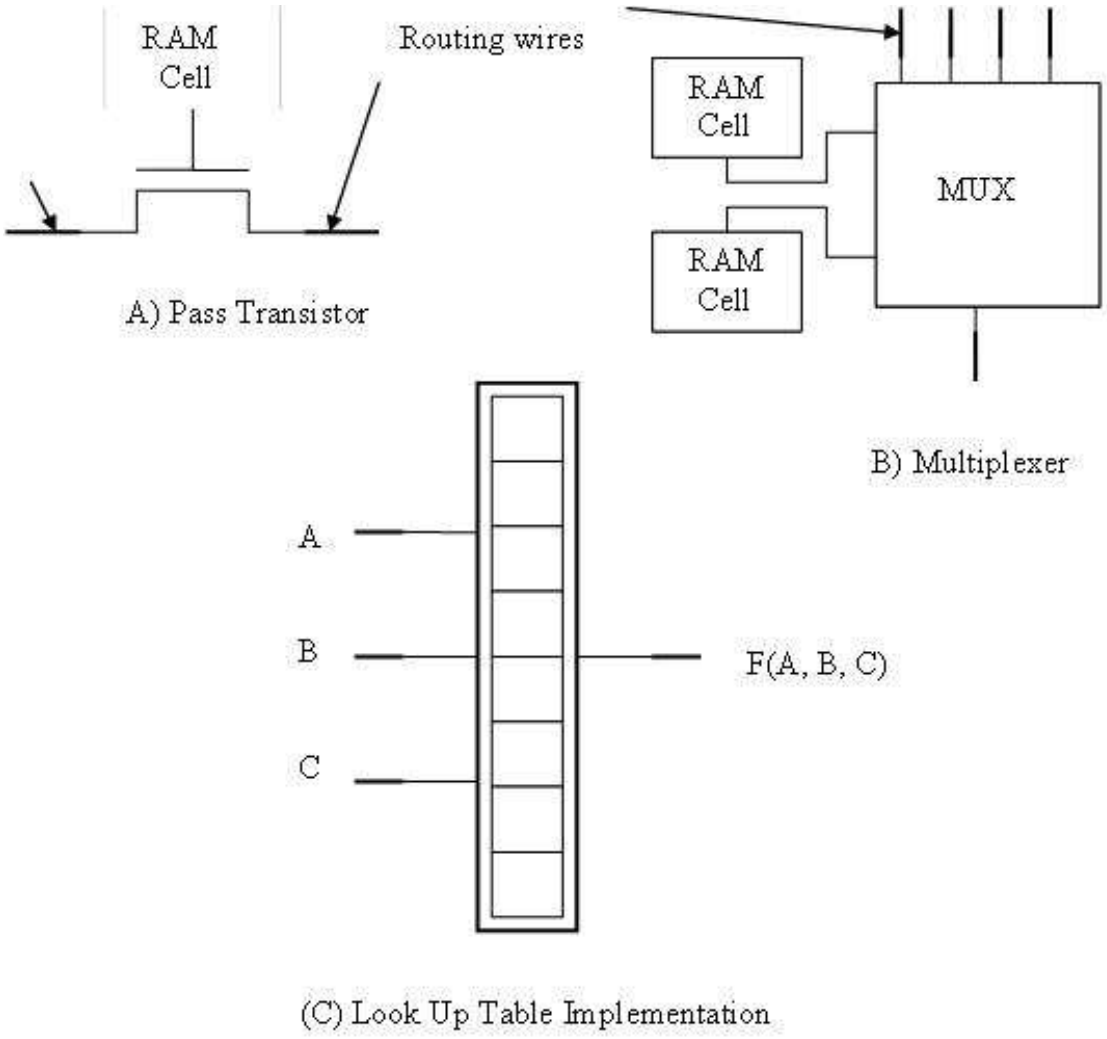

Figure 2.4: SRAM Programming Technology [**?**]

tors **??**(a). A different configuration using transmission gate or multiplexer is possible as shown in fig **??**(b,c). Since, SRAM is volatile; the FPGAs have to be reconfigured every time the power is turned off. Alternatively a permanent memory cell like ROM cell can be used to configure the FPGA to store the configuration even when the power is turned off. SRAM based FPGA consumes a lot of chip space because the SRAM cell consists at least five transistors. The advantage of these FPGAs is that they can be reconfigured in circuit very quickly and they use standard CMOS transistors.

2.2.1.3 Configurable Logic Blocks

CLBs are the main function blocks of FPGA. They implement the user logic. CLB structure varies from one FPGA to the other. But the main components of FPGA are the LUTs (Look Up Table), multiplexers and flip-flops.

A CLB of a Xilinx Virtex 2P FPGA consists of four slices and two tri-state buffers. Each slice contains two function generators, two storage elements, arithmetic logic gates and multiplexers. The function generators here are four-input LUTs. A four-input LUT has 16 bits of distributed SRAM memory cells that store all the possible combinations of a 4-bit logic. A 16-bit shift register controls the select of SRAM memory cell. The storage elements are either in form of edge triggered flip-flops or level sensitive latches.

Figure **??** [**?**] depicts a single slice of a CLB block in a Xilinx Virtex2P XC2VP30 FPGA. There are four such slices in a single CLB and there are 80x46 CLBs distributed in an array, a total of 14720 slices in a Virtex2P FPGA.

2.2.2 FPGA Design Flow

FPGA design flow describes how user logic can be translated to FPGA circuitry. Figure **??** depicts the basic design flow for FPGAs.

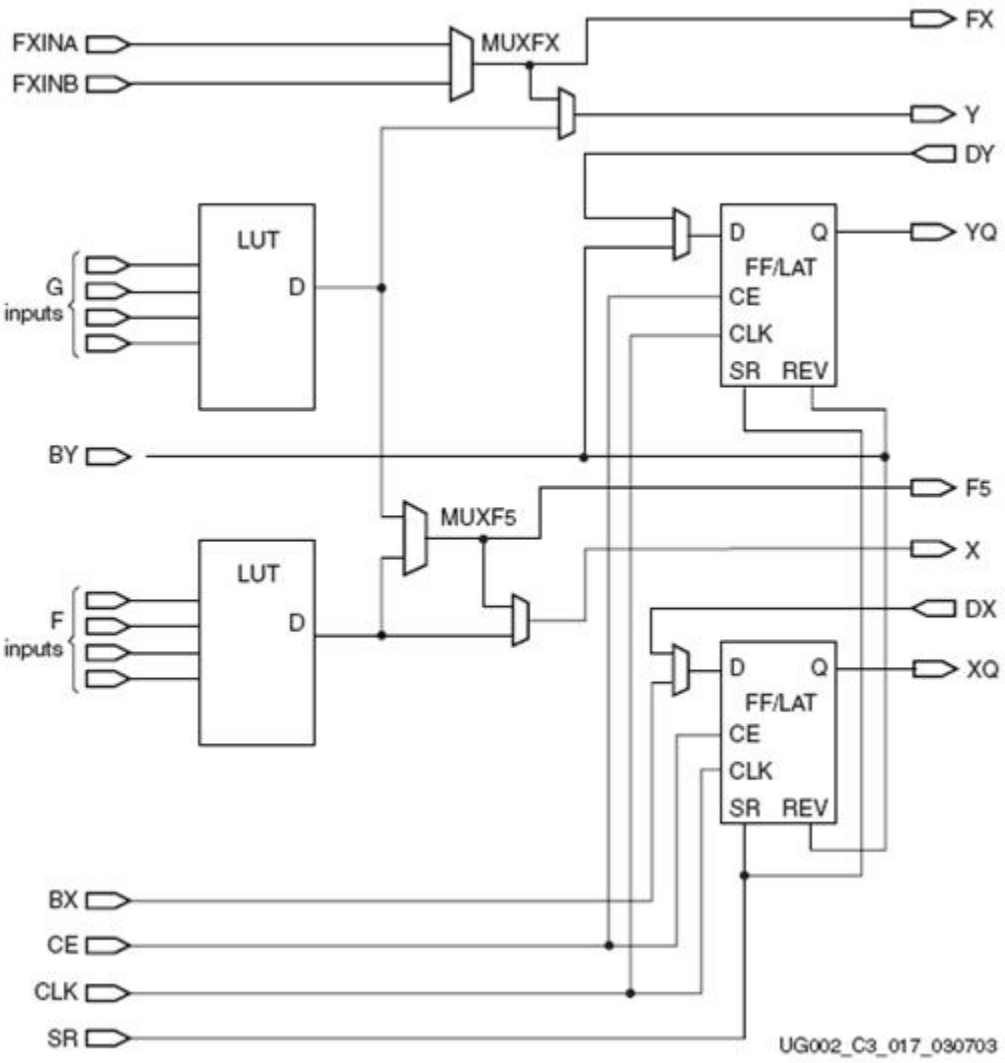

Figure 2.5: XILINX Combinational Logic Block [**?**]

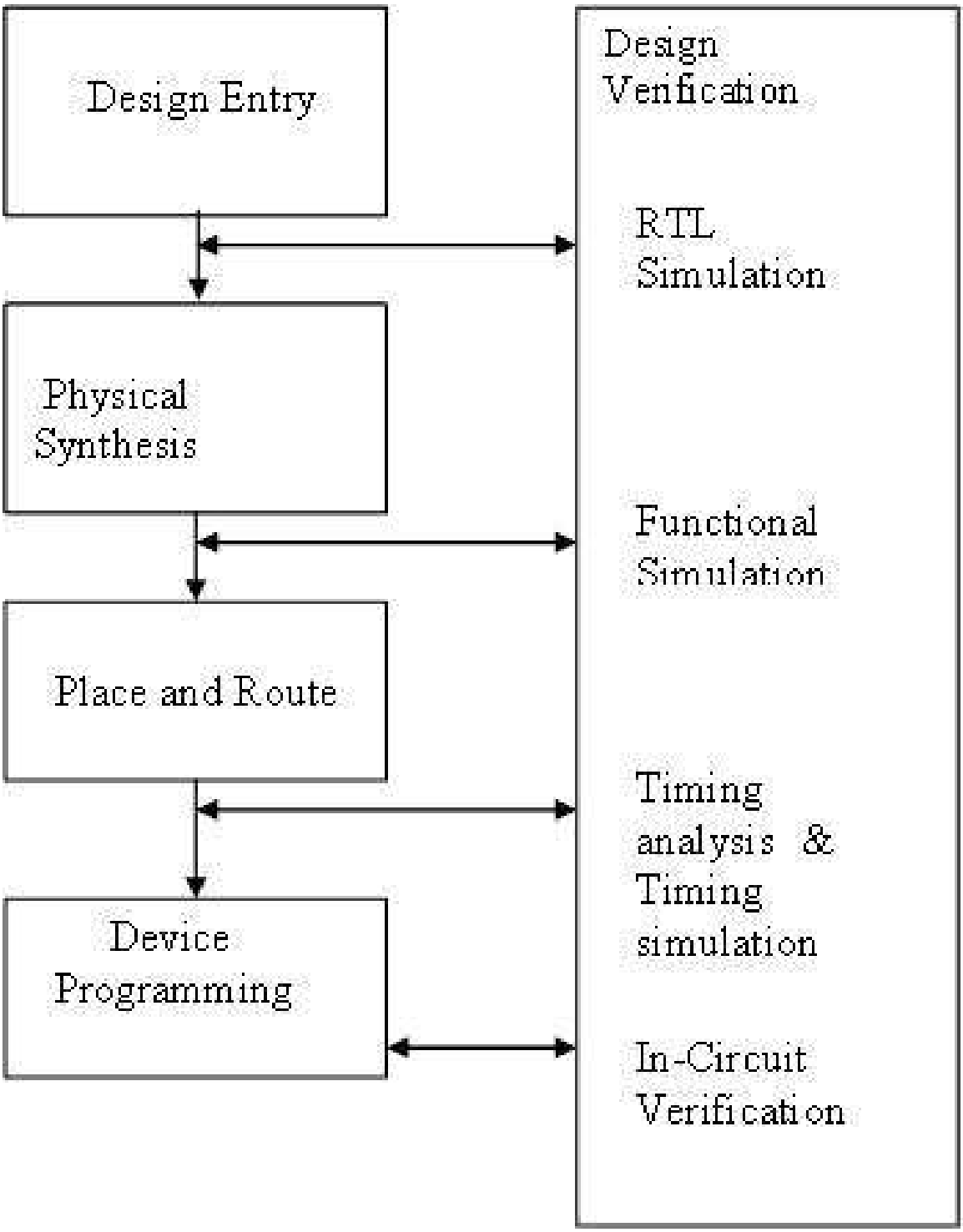

Figure 2.6: FPGA Design flow

2.2.2.1 Design Entry

The first step in the design flow is design entry. Schematic capture and HDL based design entry are two ways in which a designer can enter his design concept. Schematic capture gives designers much more visibility into hardware. Schematic capture involves a graphical interface to interconnect circuit blocks. These circuit blocks are taken from a component library provided by the vendor. The disadvantages of schematic capture are that it is difficult to enter complex designs in schematic entry and it is very difficult to part the design to another FPGA.

An alternative way to specify user logic is to use hardware descriptive language (HDL) like System C, VHDL and Verilog HDL. This approach is the best choice for designers who conceptualize their logic in a software or algorithmic model. HDL languages give the designer freedom to design a digital system at different levels of abstractions ranging from algorithmic level to the gate level, to the switch level. Using t this approach designer could model digital systems whose complexity varies from that of a simple gate to a complete digital system. In this method, the digital system can be described hierarchically. The main disadvantage of designing using HDL method is that the user is isolated from the actual details of the hardware. The software implements the entire design in the FPGA.

Other options for Schematic entry include State machine entry. State machine entry works well for simple designs like system control that can be represented in a visual format. State machine based entry method can be realized as logic function that steps through a series of states. The disadvantages of this method are that it is not ideal for implementing complex designs and tool support fir state machine entry is minimal.

2.2.2.2 Functional Verification

Verification is an important step in the design flow of logic design using FPGAs. Verification ensures that the design works as expected. Verification of design is done at different levels of abstraction from the resistor transfer level (RTL) to the actual implementation of the design in the FPGA. The main purpose of verification is to eliminate errors in the design functionality when implemented on FPGA. Verification in HDL based design methods are done through the use of test-benches. Test benches give stimuli to the user design. Functional verification is done through a series of simulations. They are

RTL Simulation After design entry, the design conceptualized through schematic capture or HDL entry is simulated. These simulations are done at different levels. The top most level is the register transfer level simulation. This is also called the behavioral simulation. Register Transfer level considers the circuit as a flow of data between different registers. It ignores the logic details between the registers. It ignores all the timing details involved. This simulation results vary to a large extent compared to the actual results

Functional Simulations functional simulations are run after the synthesis process. Functional simulations are more accurate than RTL simulations but they are more complex taking more time to complete the simulations. The signals modeled in functional simulations have values of '0', '1' and 'U' and 'X' for logic value 0, 1, un-initialized and unknown state. Functional simulations ignore timing details but include unit delay, i.e. the delay is set to a fixed value.

Timing Analysis and Simulations Timing analysis analyzes the circuit in detail calculating all the delay paths and their timing delays. Timing simulations verify the functionality of the design with timing delays in all the data paths in the circuit. The timing analysis can be performed after mapping and place and route showing the design delays at these levels.

Post map timing and route simulation takes into consideration the effect of logic delays on the timing constrains, routing delays are not considered. Post place and route simulations incorporate both the gate delays and also the routing delays to give the final simulation result. Routing delays account for more than 50% of the total delay. Post place and route simulations which are close to the circuit behavior when implemented on an FPGA.

2.2.2.3 Physical Synthesis

Synthesis is the process of converting the design entered in the form of schematic entry or HDL entry to Net-list. Net-list is a representation of inter-connections between various gates. Logic synthesis consists of two separate phases called logic optimization and technology mapping. Logic optimization produces an optimized network that is equivalent to the original Boolean network and technology mapping transforms this optimized network in to a circuit that consists of minimum restricted sets of FPGA blocks. The user can apply some constraints as input to the synthesis process. Constraints include area constraints, timing constraints etc, Constraints can be applied globally or to specific portions of design. This constraints help further in optimizing the final net-list.

Logic Optimization

Logic optimization is the initial phase of logic synthesis. Here the original network is restructured to reduce the cost of function calculated from the main Boolean network. The purpose of logic optimization is to reduce the complexity of the network. Optimization is achieved by removing the redundant and common sub expressions. For example, consider a 4-input, 2-output network

$$F = abc + abd$$

$$G = ab + c + d$$

The complexity of this network can be reduced by using new nodes $e=ab$ and $h=c+d$ using this intermediate node the two outputs can be expressed as

$$F=eh$$

$$G=e+h$$

The complexity of original network can be reduced, since the optimization phase does not consider the type of elements that will be used in final circuit, it is called technology independent logic optimization.

Technology Mapping Technology mapping transforms the optimized Boolean network to the final circuit. Logic optimization process creates an optimized network. Pieces of this network can be implemented by any of the available circuit elements. Technology mapping specifies how this optimized network can be implemented with the circuit elements like LUTs and how these elements can be connected to get the final net-list. This net-list is further optimized to reduce the area and delay in the circuit. Conventional mapping techniques use simple gates like NAND, NOR and other gates used in standard cell library to implement the optimized network. The complex logic blocks used in the FPGA present difficulties for the library based approaches. These logic blocks implement a large number of functions. Technology mapping deals with implementing the original network with available CLBs in the FPGA

LUT Technology Mapping

Look-up-Tables (LUTs) are the basic blocks in the FPGA. A k input LUT corresponds to a digital memory which can implement any Boolean function of k variables. The k variables select one of the 2^k by 1-bit memory that stores the truth table of the Boolean function. For technology mapping one can think of a library cell based mapping for LUTs but the difficulty is that the LUT implement a large number of functions. A k input LUT implement 2^k functions. For values of k greater than 3, finding a standard cell library representing a k input look-up-table becomes imprac-

tically large. There are different LUT technology mappers like $chortle - crf$ [**?**, **?**, **?**] technology mappers, $mis - pga$ [**?**, **?**, **?**, **?**] mapping etc. All these programs map Boolean network to k input LUTs while trying to minimize the total number of LUTs or improve the timing performance. For example consider a Boolean network as shown in the Figure **??** [**?**], this network is being mapped to a LUT having four inputs. Here the nodes a to m are the inputs to the network and the node z is the output node. The function specified by the network is

$$Z= (abc + def) (g + h + i) (jk + lm)$$

The original network is partitioned in to a set of trees and each tree is separately mapped to a k input LUT. This creates the final net list which can be implemented on the FPGA. The main challenge in mapping is to decompose the Boolean network to appropriate trees such that the total LUT count reduces or the timing improves.

2.2.2.4 Place and Route

Place and route ensures that the logic blocks generated after technology mapping are appropriately placed on the FPGA such that the total area occupied by the design on the FPGA is as small as possible. Placement also has to consider the ease of aiding subsequent routing. It is an optimization problem. Most of the algorithms that implement placement problems are based on heuristic approach. These methods are based on assuming an initial placement model and after some number of iterations a better placement model is achieved. The quality metrics for placement problem are the area and the delay. One more factor to consider while using these algorithms is the time to complete the iteration process. The iteration time should be practical. Place and route takes about 90 percent of the total time taken to compile the entire design flow using the EDA tools.

Routing is the process of identifying appropriate routing segments and switches that connect all the modules that are placed on a FPGA. Good routing is critical for

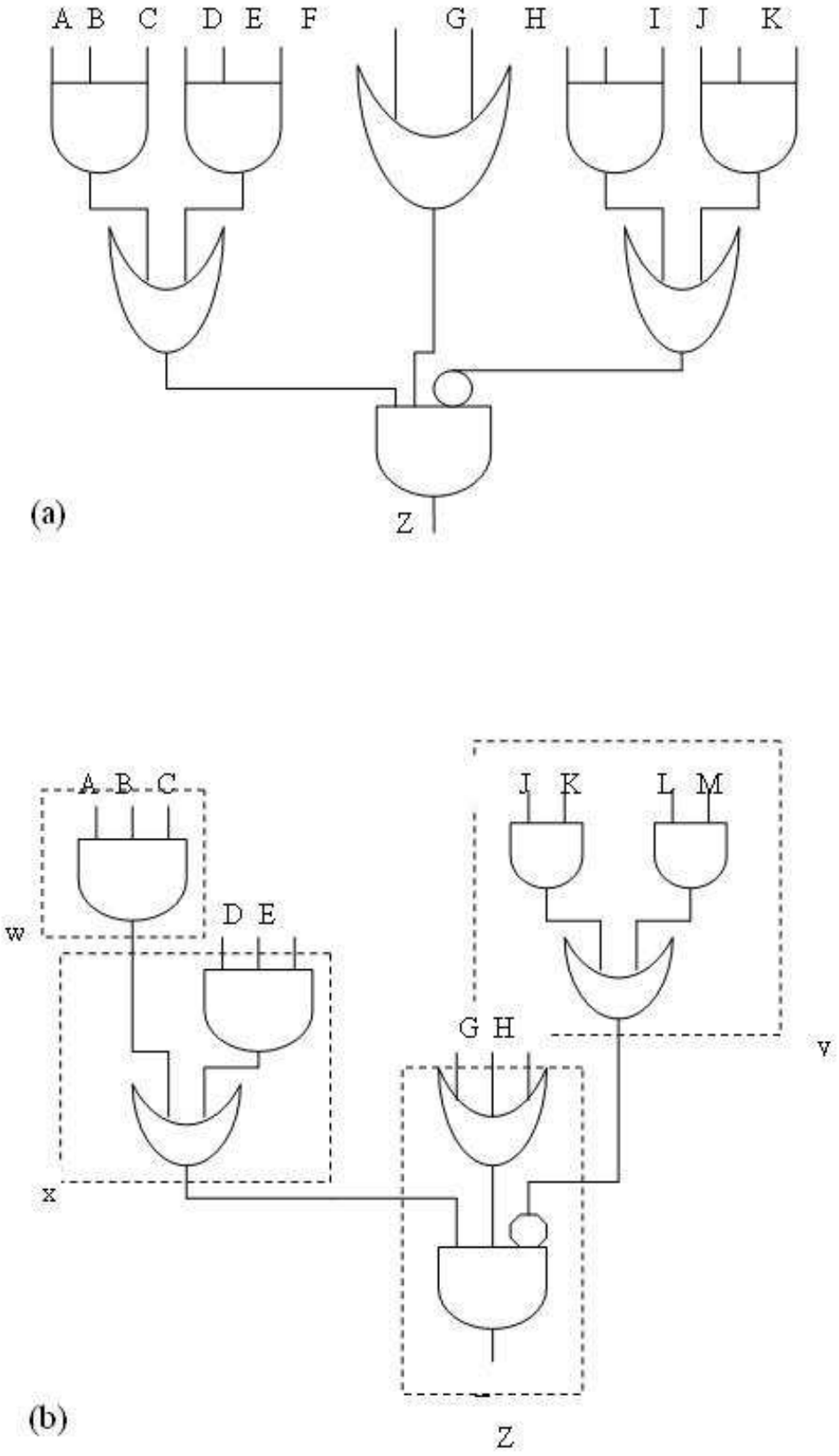

Figure 2.7: Illustrates the implementation of the network shown in (a) using a 5-input LUTs as shown in (b). [**?**]

a good design. The routing delay accounts for 45-65 % of the total delay time in the circuit. Software that performs automatic routing has existed for years. The main strategy used here is to divide the routing problem in to global routing and detailed routing problems. A global router divides the routing resource to small regions and assigns each net in the network to a subset of these regions. The detailed routing decides on how to connect within each of the small regions. The maze algorithm which is based on the Dijkstra's algorithm is used for most of the routing problems.

Chapter 3

Network Intrusion Detection.

Network intrusion detection is the process of identifying and analyzing packets that may signify an impending threat to organizations network. The Internet has grown enormously in recent years. The amount of data being transmitted through the network is rising exponentially. The amount of corporate and other individual networks is also rising sharply. Many corporate networks have been the target by hackers. With the number of attacks on the rising, people began to focus on network security. Intrusion can be in various forms like unauthorized access to databases,stealing confidential data and modifying confidential data. Apart from these we are all familiar with email viruses and spywares. When the network security is compromised, the cost to repair or recover data lost is huge. This explains the need to prevent intrusion.

Network intrusion detection systems (NIDS) are one of the primary tools available to help in creating a secure network infrastructure. These NIDS tools are deployed in a network in various types of configurations like passive and host based configurations. Both of the configurations accomplish the same goals in detecting and alerting when there is an attack. The underlying technical approach, however, is vastly different. In a host based system, NIDS monitor a single system. This type of configuration has limited range protecting only the host computer. Such type of configuration is useful in gateways, switches or routers. To the contrary passive NIDS entail using

secondary node to analyze all the packets on the network thus protecting all the computers connected to the network. Host based configuration is more complex to install and maintain, all the systems connected to the network should have the NIDS tools installed and frequent updates on all the machines are required to maintain solid defense from intrusion.

NIDS inspect all the incoming packets for any suspicious data. The suspicious data can be in various forms like the source and destination ip address, destination port numbers or some header data. The definitions of suspicious data are given as rules in *SNORT*, *Hogwash*, and several other Intrusion detection softwares. *SNORT* is a popular open source network intrusion detection software suit. It has several thousand rules that describe potential network security threats. A typical *SNORT* rule is shown below

Alert udp $EXTERNAL_NET any − > HOME_NET 31335 (msg:"DDOSTrin00 Daemon to Master message detected"; content:"l44"; reference: arachnids, 186; classtype: attempted-dos; Sid: 231; rev: 3 ;)

This rule describes one of the threats posed by the data *l44* in the incoming packet. NIDS prevents the packets containing suspicious data passing through the network. Inspecting all the incoming packets for spurious data is a time consuming process. With the number of rules given in snort ranging in few thousands, network intrusion detection demands high system performance to process the data at full network speed. A hardware based NIDS will be an ideal resource for applications demanding high performance. This model is a CAM (content addressable memory) based model CAMs have long been used for fast string matching against multiple keywords. Use of CAMs in caches and IP address look-up table in routers is well known.

A CAM based NIDS has an architecture where a set of signatures with a fixed key size of k bits are stored in the CAM. The datagram packets coming in from the

network are parsed and given as input to the signature match processor As packets arrive from the network, each k bits could be matched against the CAM to see if there was a match. If a match is found, It indicates the presence of malicious content in the incoming packet. We can flag the packet for further analysis. Since the signature set is stored in a writable CAM, CAM-based NIDs systems do not need to reprogram the FPGA every time there is a change in the rule set. However, unlike finite automata, CAM-based designs can not easily handle regular expressions, though the use of TCAMs (ternary CAMs), does allow limited wildcard matching. The primary problem with such an architecture is that NIDS signatures are not of a fixed size. For example, the Snort rule-set has rules that match on content strings that can be anywhere from 1 to 100 characters long. One solution could be to select the value of k to accommodate the largest possible signature, though that will lead to unused space within the CAM. The *Granidt* system uses such an approach with a 160-bit wide CAM, thereby limiting signatures to 20 characters length. Using a fixed value of k can also cause misses in the incoming stream because of signature overlaps. For example, consider the case where the CAM contains two signatures FOO and BAR, and the input stream is *AFOOBARCD*. The CAM will be presented with $k = 3$ characters at once - *AFO, OBA,* and *RCD*. Because of overlap, none of these will cause a match, even though the input stream obviously contains the two signatures. Recent CAM-based designs have recognized this problem and taken a sliding window approach that use single character comparators with shift registers to propagate matches across clock cycles. Optimizations to these approaches include processing characters in parallel, prefix sharing, and pattern partitioning.

3.1 Signature Match Processor Architecture for Network Intrusion Detection

An architectural overview of the SMP is shown in Figure **??**. The SMP consists of a control unit, a CAM character match array, a Signature Match array, a Signature match buffer and address output logic. The SMP receives packets from the network in a stream and outputs the address of any signature that is found in the packet. A managing network processor or CPU can use this information to raise a network alert or attempt to terminate the offending connection.

3.2 Character Match Array

The core of the SMP is in the character match array and the signature match array. Cam is usually used in high speed pattern matching. In a Signature match processor, the function of the CAM array is to generate a series of character matches based on the input characters. In the signature match processor that has been designed two versions of cam can be used. In one version of the architecture, the character match array was implemented as an array of CAM cells into which the desired signatures were loaded [**?**]. The CAM cells we have used are slightly different from the commercially available CAMs. Each of the CAM cell contains 9 registers which store the content of a single character in 8 registers and one extra cell stores a word end vector which stores the one bit flag indicating the end of a keyword. A CAM cell is shown in the Figure **??**.

The advantage of the structure was that it was easy to update signatures simply by writing into the CAM. The disadvantage, however, was that the CAM array was very large in area. Moreover, the use of a CAM array prohibited the use of optimizations such as parallelism and prefix sharing. Parallelism deals with processing character matches in parallel and prefix sharing deals with optimizing the search space by

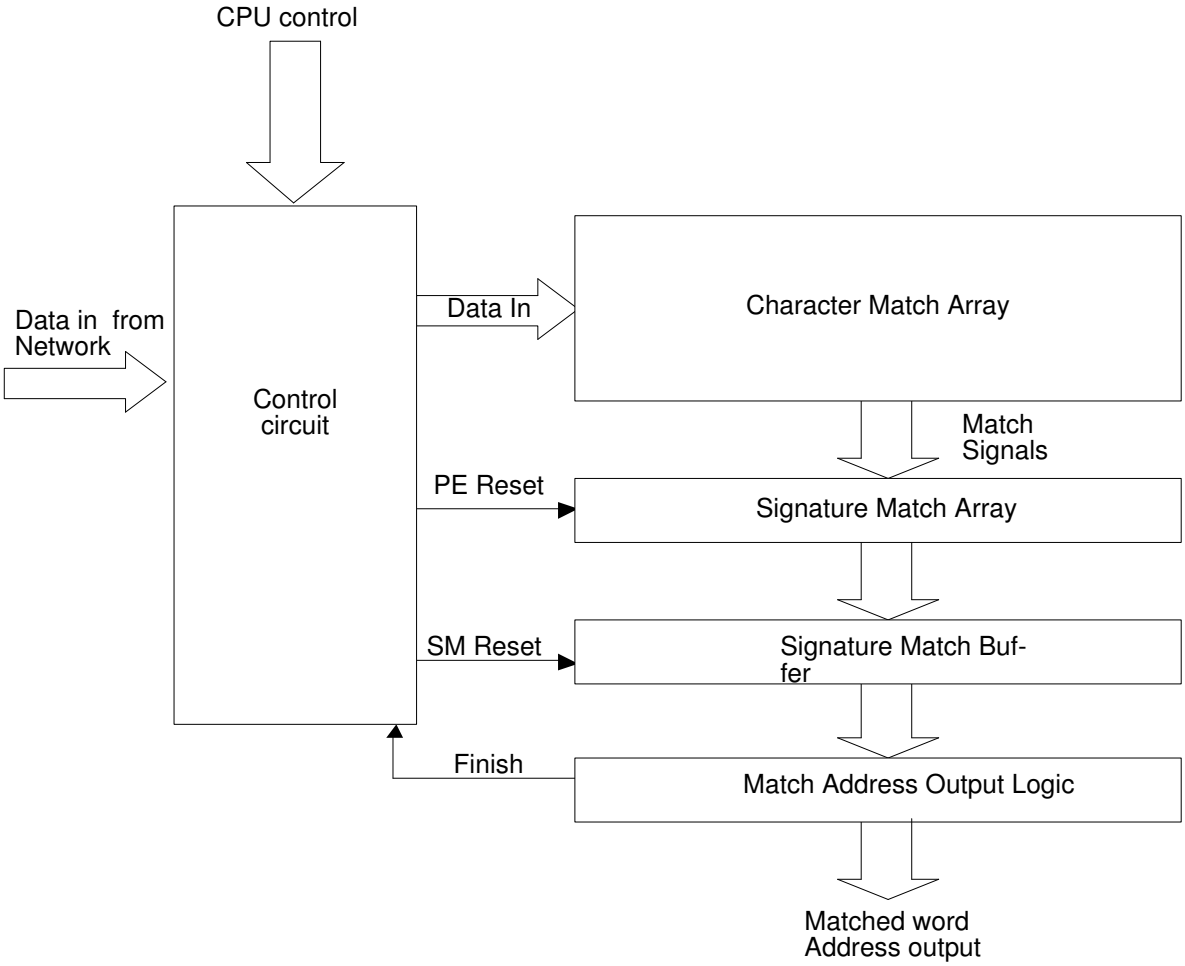

Figure 3.1: Signature Match Processor Architecture.

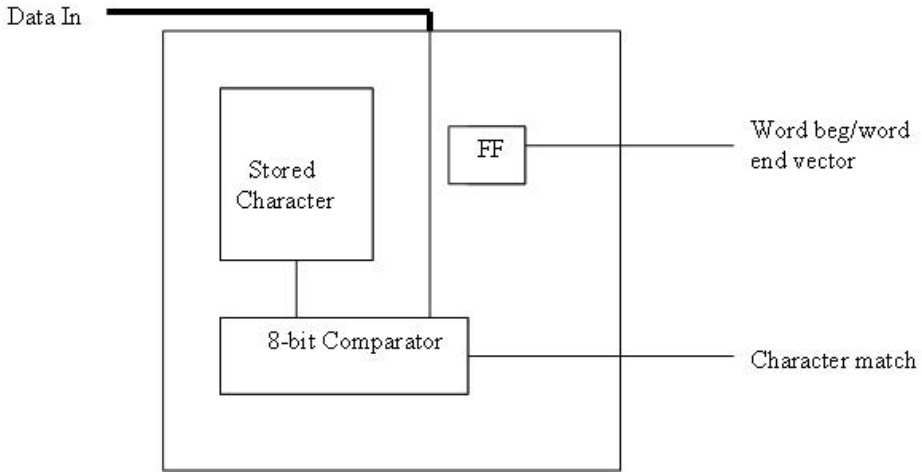

Figure 3.2: CAM Cell

grouping different keywords with common prefixes and reducing the total number of CAM cells used storing all the keywords. Because of these limitations, we decided to abandon the use of the CAM array and instead employ discrete comparators as was used in [**?**, **?**]. It has a similar structure to above cam cell but the cam cell can not be updated with new data. Sacrificing the ability to update the signatures without reconfiguring the FPGA is not a serious loss since signatures in a NIDS context change relatively infrequently. The character match array is comprised of a series of comparators, each of which matches on one of the possible incoming bytes. This, of course, implies that there are 256 comparators to match all possible 8-bit characters. In order to improve performance, it is desirable to be able to match several characters within one clock cycle. Therefore, we use p rows of comparators, where p denotes the degree of parallelism. As in Figure **??**, there are p match signals per comparator column. Also, for every clock cycle, exactly p of the $256p$ match signals from the character match array will be asserted.

3.3 Signature Match Array

The signature match array is the distinguishing feature of the design. The Signature Match Array processes the character matches from the CAM array to find the signature match. The signature match array is similar in concept to systolic processing arrays that have been proposed for approximate word search matching in CAM-based dictionaries [**?**, **?**]. The array is comprised of an n x 1 array of processing elements (PE) where n is the number of characters in the signature set to be matched against. Thus, each PE represents one of the characters in the signature set, and likewise, the PE is connected to the character match array column that corresponds to its character.

Figure **??** shows how the array would be connected for the string ABB. In general, each PE has $2p$ inputs and $p + 1$ output. P of the inputs, represented by $MX[1 : p]$,

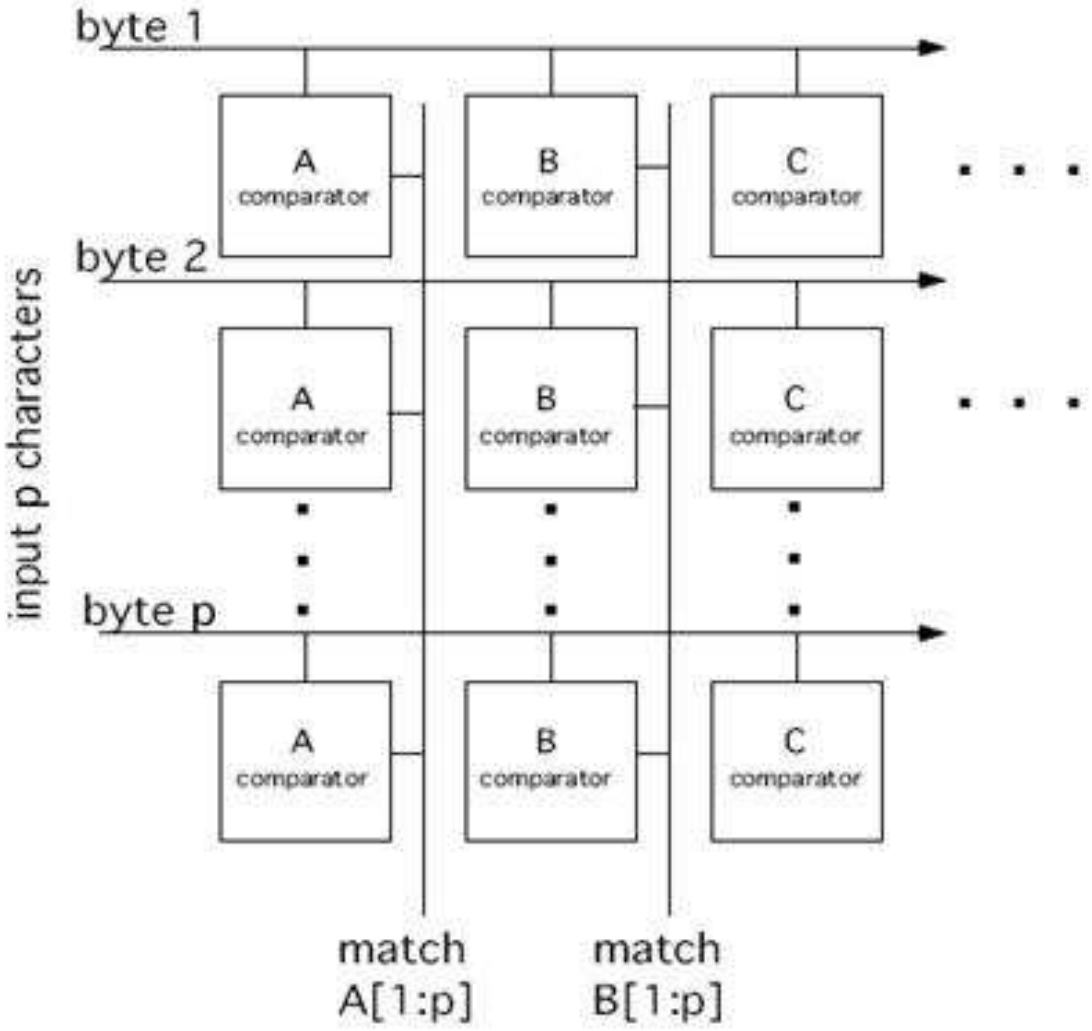

Figure 3.3: Character Match Array

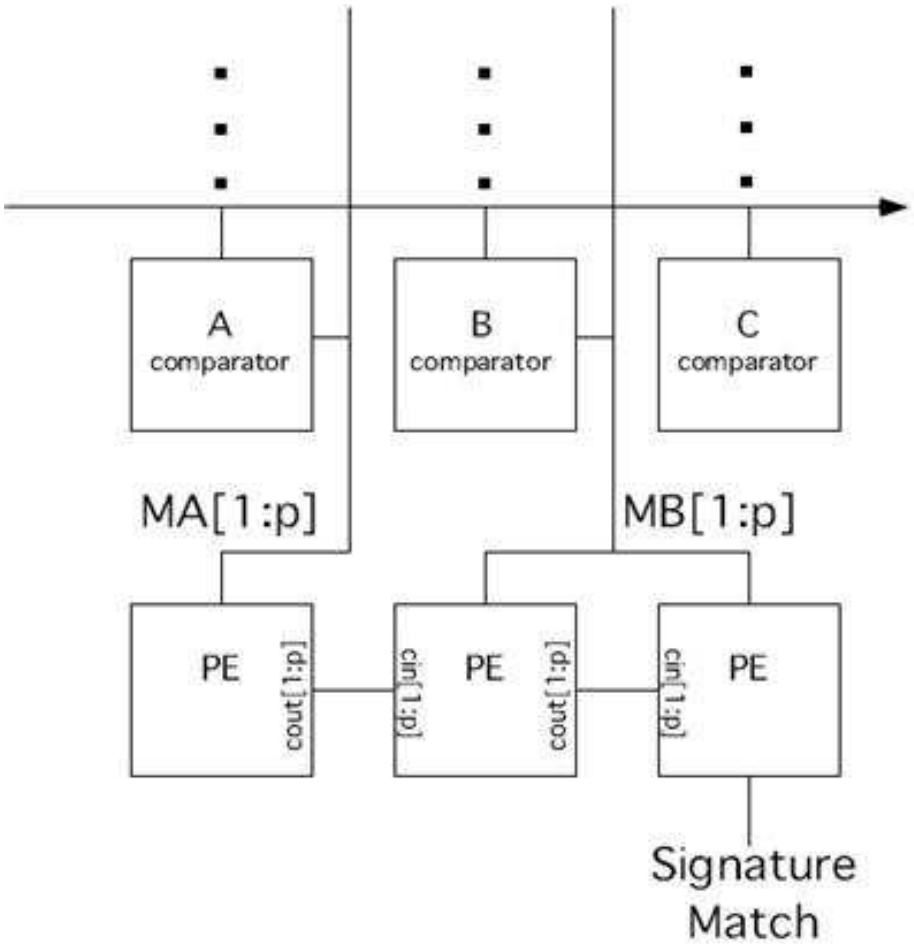

Figure 3.4: Signature Match Array.

correspond to the p match signals from the character match array. X refers to the character for which the PE is responsible. The remaining p inputs are carry signals that forward match information from the previous PE, and likewise, p of the outputs are carry signals to a subsequent character in a signature. The final output is a signal that indicates a signature match was found. The signature output match signal is only valid on PEs that signifies the end of a signature. In addition to knowing whether the PE is at the end of a signature, the PE also knows if it is at the beginning of a signature. A summary of the algorithm executed by each PE is shown below for $p = 4$ in pseudo-*VHDL*.

```
cout1 <= MX1 and (cin4 or sig_beg);
cout2 <= MX2 and (cin1 or sig_beg);
cout3 <= MX3 and (cin2 or sig_beg);
cout4_temp <= MX4 and (cin3 or sig_beg);
sig_match <= sig_end and cout1 or cout2 or cout3 or cout4_temp;
if ( clk'event and clk='1') then
cout4 <= cout4_temp;
end if;
```

The first part of the pseudo-VHDL generates the carry signals to propagate to the next PE. A carry signal indicates that there is a signature match up to that character. The basic idea is to check each of the *MX* signals for a match and then see if the previous PE also found a signature match up to the previous byte. You must also check if the PE is at the beginning of a signature, i.e. *sig_beg* is '1'. If the previous PE has forwarded a signature match or we are at the beginning of a signature, the PE can forward the signature match to the next PE. For example, if cin1 is '1', it indicates that the previous PE has determined that the signature has matched up to

itself, and the last matched character of the signature is in byte 1. Therefore, if the current PE sees a match on byte 2, i.e. MX2 is '1', then you propagate a signature match on cout2. If the PE is at the end of a signature, *sig_end* is '1' and we are going to forward a character match, then we know that we have matched on an entire signature. This will allow us to flag the *sig_ match* signal.

From the pseudo-VHDL it can be seen that the p^{th} carry out is registered in each PE. This is because the only way a match can occur on MX1 is if it is at the beginning of the signature or the last character on the previous clock cycle matched also. The last character on the previous clock cycle corresponds to a registered version of the p^{th} carry out. The implications of this are that only one register is required per character in the signature set and moreover, the number of registers does not increase as we increase parallelism. This is a significant savings compared to other comparator based techniques which are $O(L^2)$ relative to the length of the signature [**?**]. As an example of the PE algorithm, consider the following rule from Snort.

```
alert udp $EXTERNAL_NET
 any -> $HOME_NET 31335
  (msg:"DDOS Trin00 Daemon to Master
    message detected"; content:"l44";
    reference:arachnids,186;
    classtype:attempted-dos;
    sid:231; rev:3;)
```

The signature match arrays are shown in Figure **??** and the PEs are configured to detect the signatures "*l*44 "and "*adsl*". Let us assume that the degree of parallelism k= 2, i.e. the input string "*fl*44" is presented to the SMP in two clock cycles. In the first clock cycle the data "fl" is presented and in the next clock cycle data "44" is presented. In the first clock cycle, the character "*l*" matches in the second row and that sets the register in the first PE corresponding to "*l*". In the second clock cycle, the character "4" matches in both rows. The first '4' PE will have *cin*2 set from the

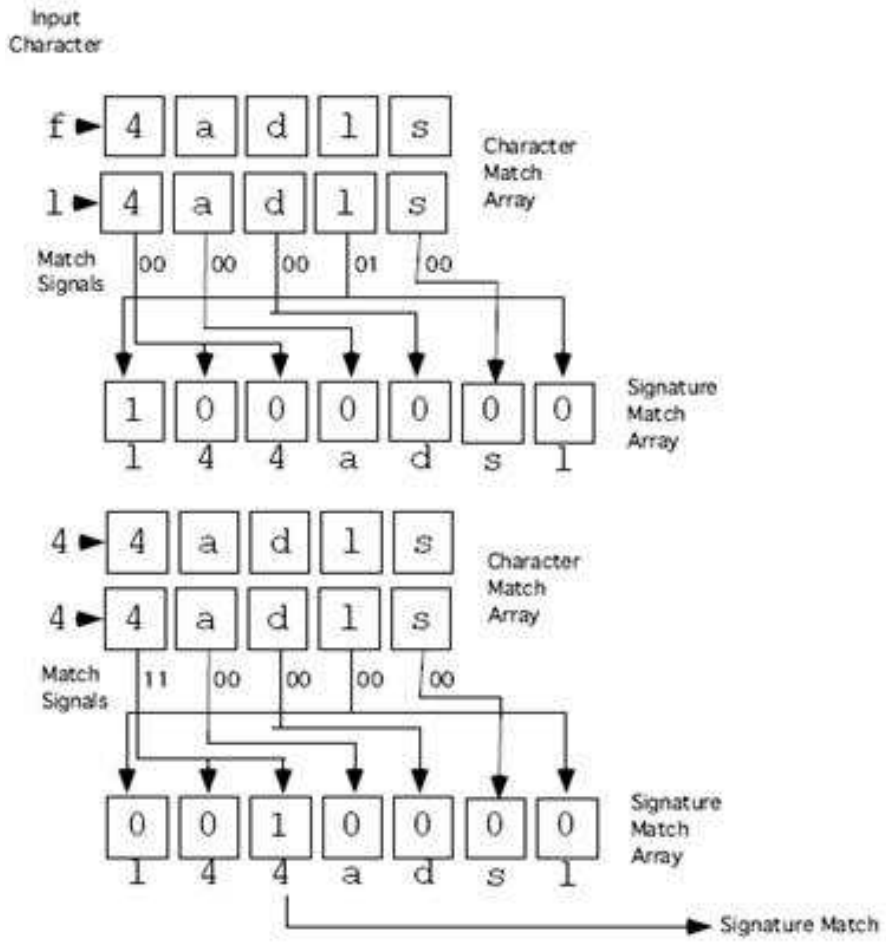

Figure 3.5: Signature Match Processor Example.

'l' PE. That allows it to forward a match to the second '4' PE on $cout1$. The second '4' PE will see $cin1$ set as well as $MX2$ and it can then determine a signature match because it has sig_end set as well.

3.4 Address Output Logic

The signature match array outputs a word match signal that indicates the presence of a match. There can be more than one word match and word matches can be detected at different cycles. For the SMP to be useful in the context of NIDS, we also need to know the signature which caused the match. It is the address output logic that finds which signature or signatures caused a match. In order to generate these addresses, we use the signature match signals generated by the signature match array. As the input string propagates through the CAM array, any time there is a signature match, there will be a signature match signal at the end position of the signature. Since there may be multiple signature matches in a single string and the signature match signals can appear on different clock cycles, it is necessary to latch each word match signal in the word match buffer. The word match buffer is a regular array of flip-flops that stores all the signature matches. The buffer maintains the position of the signature match. When the last character of the input string has passed through, the word match buffer will then have values set at each location corresponding to the end of a matched signature. When the last character has been reached, the match address output logic can begin processing the word match buffer entries. In order to find out the signature which caused a word match, we need to get the beginning of the signature address from end of the signature address returned by the word match buffer.

Previous work by Bu and Chandy [**?**] used a start address RAM that stores the beginning address of each signature. The address output logic simply reads the RAM entries starting from end to beginning. For each start address matches. The buffer

maintains the position of the signature match. When the last character of the input string has passed through, the word match buffer will then have values set at each location corresponding to the end of a matched signature. When the last character has been reached, the match address output logic can begin processing the word match buffer entries. In order to find out the signature which caused a word match, we need to get the beginning of the signature address from end of the signature address returned by the word match buffer. A start address RAM that stores the beginning address of each signature can be used to get to the start address of each signature. The address output logic simply reads the RAM entries starting from end to beginning. For each start address A read from RAM address i, the position $A\text{-}1$ of the word match buffer is verified. A positive match indicates a signature match corresponding to the start address located at the $i\text{-}1^{th}$ position of the RAM. This method takes S cycles to perform the logic, where S is the number of signatures. Larger values of S would result in a longer time delay in performing the logic.

In this thesis, I present match address output logic (MAO logic) which has the structure of a binary tree. The goal here is to get to the start position of signature from the match positions stored in word match buffer. This is accomplished by propagating the word match signal at the end position of the signature to the start position of the signature. This gives the matched position location of the signature (MP) in the CAM. Whenever the CAM character match array is updated these connections must be updated as well. The MP is given as input to MAO (matched address output) logic block which has the structure of a binary tree.

Figure **??** shows the binary tree and the logic for each of the block in the tree. MAO logic separates multiple matches or signature and decodes the start address of each matched signature. This logic generates a MAA (matched address available) signal which tells the control circuitry that there are matches. Then a LP (Leftmost Pointer) signal propagates back up the tree which is the leftmost MP signal. At each

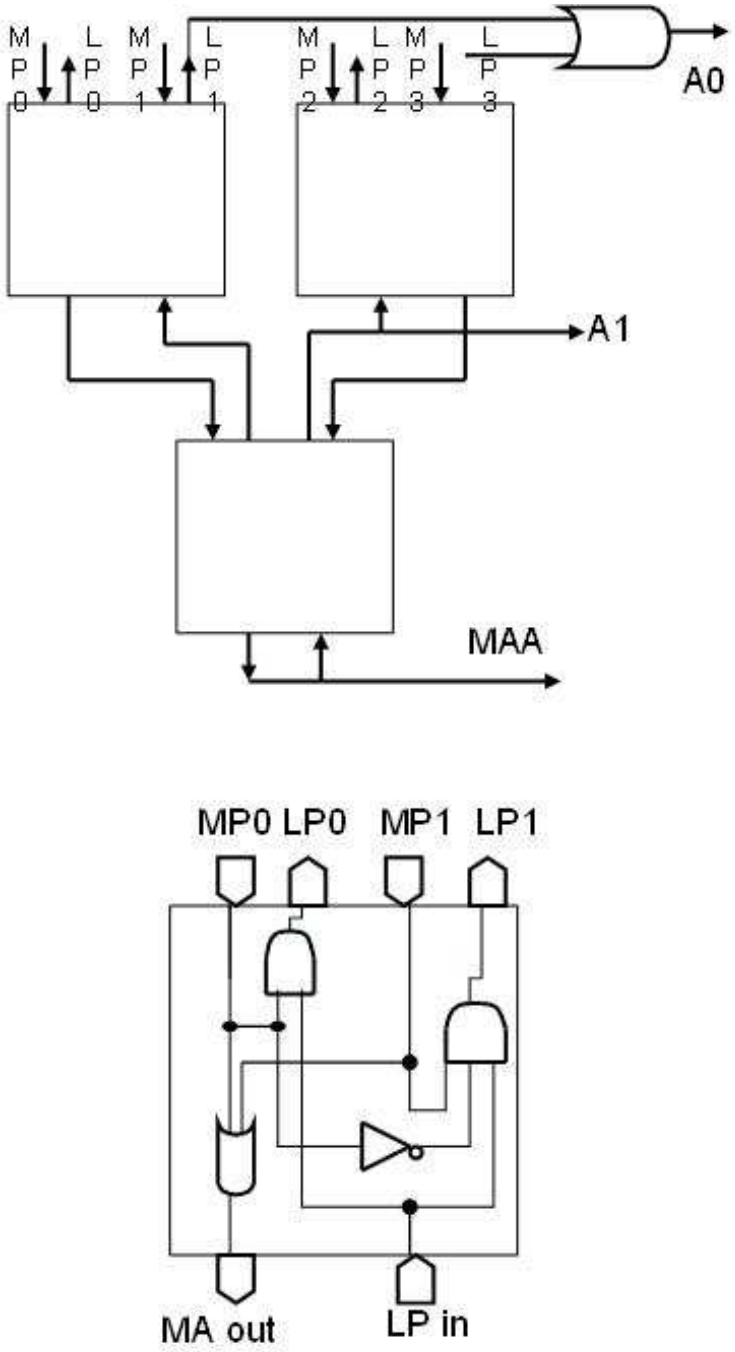

Figure 3.6: Match Address Output Logic.

level of the tree, a bit of the address is generated from the LP signal at that level. When the LP signal reaches the top of the tree, it is used to reset 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. For M matches found for a particular input string, the MAO logic takes M cycles to perform the logic. This is better than the S cycles taken by the previous RAM-based implementation of the MAO logic. The MAO logic is pipelined to increase the clock frequency.

3.5 Control Circuit

The control circuit manages the data flow through the SMP and also manages flow control of the incoming packet. When an incoming packet is ready to be delivered, the control circuit first resets the signature match array and also resets the word match buffer. The control circuit then takes each byte in the incoming packet and presents it to the character match array on every clock cycle. When the last character in the packet has arrived, the address output logic is enabled. Because the address output logic process is independent of the signature matching process, the control circuit can start processing the next packet immediately.

3.6 Performance Analysis

Overall, the time to process a b byte packet is $b/p+M+1$ cycles where M is the number of matches found in the packet. b/p corresponds to the time for the packet to stream through the SMP signature matches and $M + 1$ is the time to do the matched address output. Since the matched address output phase could be completed in parallel with the signature match, we are left with a per-packet cycle time of $max(b/p,M + 1)$. If $b/p > M + 1$, which is the general case, the per-packet cycle time is b/p, and the per-byte run-time is $1/p$ cycles.

3.7 FPGA Implementation and Results

The signature match processor for network intrusion detection was designed using the Xilinx Virtex- II Pro FPGA (XC2VP230-7). The design was written in VHDL and Xilinxs ISE 6.3i design environment. The design was implemented on the Xilinx XUP Virtex II Pro Development board using Xilinx's Embedded Development Kit. The Xilinx Virtex-II Pro FPGA contains two embedded 32-bit PowerPC 405 RISC microprocessors and numerous soft ip cores The Xilinx XUP Virtex II Pro Development board consists of the Virtex- II Pro FPGA with a number of peripherals which can be used to build a complex system. The board runs with a clock signal of 100Mhz frequency. The ISE was used for synthesis, place and route of the design written in VHDL. Mentor Graphics ModelSim was used for functional verification of the behavioral and place and route model of the design.

A Snort rule database consisting of small signature set with 95 rules comprising 1024 characters. Table **??** shows the resource utilization for different levels of parallelism. As can be seen, the circuit using binary tree structured MAO logic uses roughly 1.5 flip-flops and 1.5 LUTs per CAM character compared to 2 flip-flops and 3 LUTs per CAM character of SMP using a start address RAM to implement the MAO logic. The flip-flop is used to implement the PE register and the word match buffer. The LUTs correspond roughly to the CAM, the PE logic, and the matched address output logic. The flip-flops and LUTs can be mapped to almost 1 slice per character for binary tree implementation of MAO logic compared to the other version which uses roughly 2 slices per character.

Table

labeltbl:SMPcomp shows a comparison of our work with other recent related work in FPGA implementations of signature matching. The performance metric is the ratio between throughput and logic cells/char and is similar to that introduced in [**?**, **?**] to evaluate the trade offs between area and performance. This design is comparable in

	XC2VP30	
Level of parallelism	1	2
Slices	423	697
Frequency(MHz)	245.1	230.6
Throughput(Gb/s)	2.0	3.7

Table 3.1: SMP Resource Utilization

performance to other work in the area, particularly other comparator based designs [?, ?, ?]. What is particularly notable is that the number of logic cells/character is significantly smaller than any other comparator based design. With increased parallelism, the throughput should increase significantly. Larger signature sets may, however, decrease the operating frequency because of increased fanout from the character match array.

In this chapter, an architecture for a hardware based network intrusion detection system (NIDS) using an innovative CAM-based signature match processor is explained. Based on the current implementation of the SMP, incoming streams of data can be processed at rates of nearly 4 Gbps. This is more than sufficient to handle intrusion detection on current gigabit networks. Also in this chapter a unique design of a priority address encoder is explained. This encoder will generate start addresses for each of the multiple matches within a packet.

Design	Device	Throughput (Gb/s)	No. characters	Logic cells/char	Performance (Mb/s/cell)
Singaraju et al.	Virtex 2VP30-7	3.7	1024	1.4	2711
Baker-Prasanna (8 byte) [?, ?]	Virtex 2VP100-7	10.3	19584	2.0	5150
Bakaer-Prasanna (Tree) [?]	Virtex 2VP100-7	2.0	19584	0.4	4848
Cho-Mangione-Smith (ROM) [?]	Spartan3-2000	3.2	6805	0.9	3556
Sourdis-Pnevmatikatos [?]	Virtex2-6000	9.7	18000	3.6	2694
Clark-Schimmel et al. [?]	Virtex2-8000	7.0	17537	3.1	2245
Cho-Mangione-Smith (RDL) [?]	Spartan3-2000	3.2	19021	1.6	2000
Hutchings et al. [?]	VirtexE-2000	0.4	16028	2.5	160
Gokhale et al. [?]	VirtexE-1000	2.2	640	15.2	145

Table 3.2: Comparision of NIDS FPGA Designs

Chapter 4

Generic Look-up Cache.

4.1 Introduction.

Apart from network intrusion detection, the signature match process can be used in cache applications. Network processing often requires the ability to cache certain frequently used values. Many network applications such as IP to Ethernet address mappings and routing table lookups require mapping from one known value to another value require the use of caches. It is common to use an Address Resolution Protocol (ARP) cache that serves as a translation table to map layer 3 IP address to corresponding layer 2 hardware addresses. If an entry is found in the ARP cache, the network processor can avoid sending a "who-is" message to determine the mapping. Routing tables that map IP destination addresses to network ports are another application for caching as well.

A major function that network processors perform is packet routing. The packet routing determines the incoming packets next hop by performing the routing table lookup. This problem is equivalent to longest prefix matching. The routing table consists of a set of entries, each containing a destination network address, a network mask and an output port identifier. Given a destination IP address, routing lookup can logically be thought of as follows. The network mask of an entry selects the most significant bits from the destination address. If the result matches the destination

network address of the entry, the output port identifier in the entry is a potential lookup result. Among such matches, the entry with the longest mask is the final lookup result.

Memory based caches or translation tables are easily implemented in software using a variety of data structures including hash tables, binary trees, tries, etc. However, for high performance applications, such as multi-gigabit networks, it becomes more and more difficult for general purpose processors or even specialized network processors to keep up. As a result, hardware based techniques have come into use, and for the most part, these techniques have been based on associative or content-addressable memories (CAM). [**?**, **?**, **?**].

The main advantage of using CAM is that the search time depends on the size of input string rather than the routing table size as it is the case with most of the RAM based techniques [**?**, **?**]. CAM relies on fixed size keys. This is its main disadvantage. When using Cams for searching, the input keys given to CAM must all be of the same size. This limitation does not apply for search operations performed on fixed size keys. Examples include routing table lookup and ARP caches, where IP addresses are all of same size. However, for other applications that require pattern matching for keywords of different sizes, fixed key size restriction on CAM can be problematic. Examples include DNS look-up and signature matching for Intrusion detection. Tasks such as DNS look-up and LDAP searches, or directory lookup require mapping a text string to an ID or object data structure. Other tasks such as intrusion detection simply require knowing whether a particular keyword is present in the search space or not. These dictionary based tasks simply query the presence or absence of a particular string.

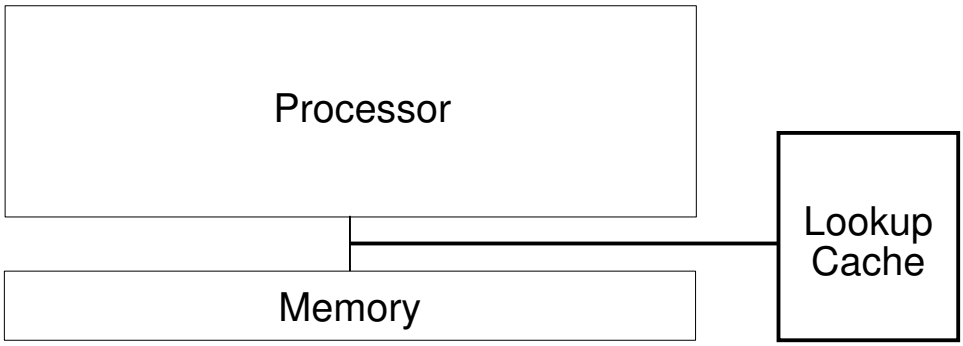

Figure 4.1: System Architecture with Network Processor and Lookup Cache.

4.2 Architecture

Generic lookup cache can be taught as a coprocessor type component to a general purpose or network processor design. The cache could be integrated into the processor core or perhaps off-chip. A possible architecture is shown in Figure **??**.

In a network processor (NP) setting, packets from the network can flow into the NP which will do any necessary processing. In a higher level application such as DNS queries, the network processor can query the lookup cache to do the actual domain name resolution. If the hostname is present in the cache, the IP addressed will be returned. The lookup cache can also sit on the memory bus to provide high performance connectivity. As such, communications with the NP will be in memory bus chunks. Thus, for a 32-bit bus, the IP address can easily be communicated in one memory bus cycle. For a query where the return value is more than 32-bits as with the results of an LDAP query, the lookup cache will return a memory pointer to the LDAP object. An architectural overview of the lookup cache itself is shown in Figure **??**. The cache consists of a control unit, a CAM character match array and a PE array. The control unit manages the lookup cache. The character match array is an array of CAM cells that matches against incoming characters. The PE array is an array of processing element cells - one for each character - and it is responsible for performing word matches as opposed to the character matches done by the CAM

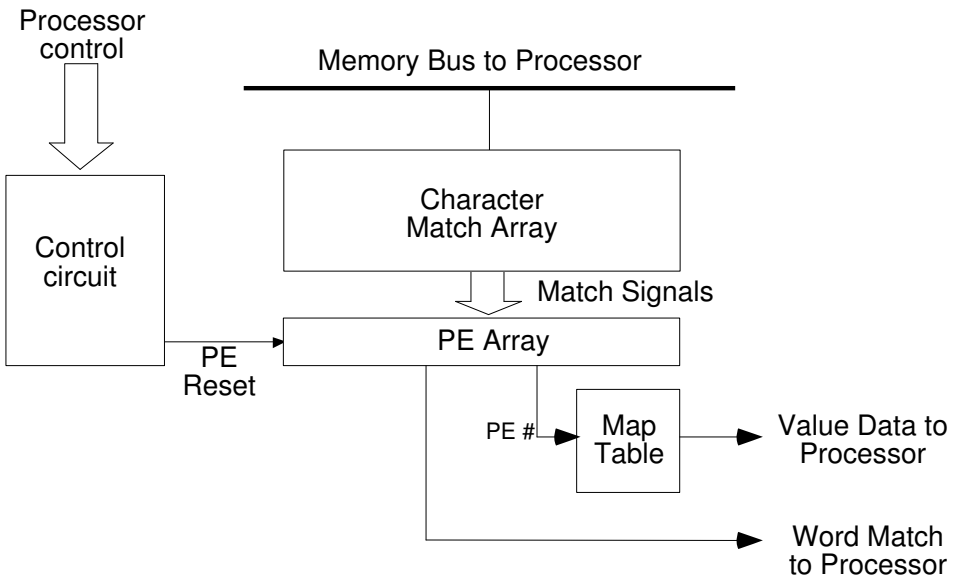

Figure 4.2: Lookup Cache Architecture.

array. The map table contains memory indexed by the PE number to retrieve values corresponding to matched keys.

4.3 Character Match Array

The core of the lookup cache is in the character match array and PE array. The structure of CAM is similar to the CAM discussed earlier. The CAM stores the keywords as 8-bit characters in an 8 by n array of CAM cells where n is equal to the number of characters in the CAM. The detail of the CAM array is shown in Figure **??**. Each column in the CAM array corresponds to a character in a keyword. Each CAM cell consists of a storage cell along with a bit comparator. In a VLSI implementation, the storage cell would be a traditional 6-transistor SRAM cell and the comparison function can be implemented with an additional three transistors. In an FPGA, the storage cell would be a register and the comparator would be an XOR gate. 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 character

is applied to a column, the column match signal is active high when all 8 bits match. The input character is applied simultaneously to all n columns in the array. The layout of the CAM array is straightforward with each key immediately following the other laid out in linear fashion. There is no hierarchy or precedence implied by the ordering of the keys. Since one of the goals of the cache is the ability to match against variable length keywords, we need someway to separate the keywords. To do so, we add a storage cell in each column to indicate the end of a keyword. If the column represents a character that is the end of a keyword, the cell would contain a '1', if not, a '0'.

4.4 PE Array

The uniqueness of the lookup cache is in the processor element (PE) array where each PE is a finite state machine that carries out the approximate match algorithm. The PE array is similar in concept to the cellular automaton processor arrays proposed by Motomura et al. [**?**, **?**]. The original Motomura design processed 4 match signals from the CAM array in a 5x3 PE array. Thus, a n-column CAM array requires a $5n/4$ x 3 PE array. This approach introduces an extra column into the PE array for every 4 CAM array columns. We have modified the design so that the entire PE array is comprised of just n columns of processing elements (PE). This simplification of the PE array removes the extra "fifth" column present in the Motomura design.

To perform a search on the lookup cache, the network processor will on successive clock cycles apply each character of the key to the CAM array. Remember, that the character is applied to all columns of the CAM array simultaneously. If an input character matches a column in the CAM array, the match signal is set and passed onto the corresponding PE for that column. Each PE holds a binary value called a flag which indicates that the input keyword matches a keyword in the CAM up to this point. The PE flags are labeled $PE(i)$, the incoming match signals from the CAM

array are labeled $M(i)$, and the keyword end signals are labeled $E(i)$, where i is the character or column number. At the first clock cycle of the lookup process, we reset each PE flag in the array to 0. As each character from the input sequence is presented to the CAM array and subsequent matches are generated, it is the PE's responsibility to forward its match flag on to the subsequent PEs. The basic algorithm carried out by $PE(i)$ to find a match is shown below in Figure **??**.

The word match signal, WM, is asserted when a match is found. The procedure entailed in the algorithm basically pushes the flag across the array from the beginning of a keyword, and as long as the incoming characters continue to match, the flag will move to the end of the keyword. The if statement on line 1 checks if the current column has a character match and if the character is the first character in a keyword. We check if the character is at the beginning of the keyword by checking if either i=0 or the previous character is the end of a keyword. The second if on line 3 statement checks if the previous PE has the flag and we have a character match in the current column. This basically ensures that all previous characters in the keyword have matched. The final if statement on line 6 checks if we have reached the end of the keyword and sets the word match signal, WM.

As an example, consider a DNS cache with the following two entries.

```
10.0.0.1
foo 10.0.0.9
fubar 10.0.0.3    bar
```

Figure **??**a shows the initial contents of the character match array and the PE array. Before starting the match process, all the PEs have been reset to zero. Note that the second `o` in `foo` and the `r` in `bar` are marked with an asterisk to indicate that the $m+1$st bit is 1 since each is the last character in the keyword. Figure **??**b shows the arrays when when looking for a match on `foo`. The network processor presents the

lookup cache with one character of the input string on every clock cycle. With an input string of `foo`, on the first clock cycle, the character `f` is presented to the CAM. There is a match on the character 0 and character 3 of the character match array. Since both are the first character of a keyword (`foo` and `fubar` respectively), $PE(0)$ and $PE(3)$ are set to 1. On the next clock cycle, the second character, `o`, causes a a match at two locations in the character match array, columns 1 and 2. The PE array evaluates and sets $PE(1)$ to 1 but does not set $PE(2)$ to 1 because the PE in the previous column, i.e. column 1, did not have its flag set. Note that the flag in $PE(3)$ drops out because the incoming character did not match the `u` in position 4. On the next clock cycle, character `o` is presented to the CAM, moving the PE match bit along as well. At this point, the PE flag has reached the end of a keyword and we know that a word match has been found.

Since a new input character is presented every clock cycle, the keyword match array 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. Thus, compared to a normal associative memory, there is no loss in performance to add variable sized key capability. However, there is an additional cost in circuit complexity to add PEs to maintain keyword matching state.

Overall, with the design as shown, the time to search for a m byte keyword is m cycles. Since the communication path between the lookup cache and the processor is most likely larger than a byte, it seems reasonable that the lookup cache should be able to process multiple bytes per cycle rather than a single byte per cycle. In order to do so, we must be able to match multiple bytes against the CAM in each cycle. One approach would be to use multiple copies of the CAM array and change the PE to look at multiple match signals at once. The disadvantage of such an approach is that it would greatly increase the size of the CAM array.

Instead, the approach that we have taken is to use aligned keywords in the CAM.

In other words, each keyword is aligned on a multi-byte boundary. In our current implementation, we have decided to align keywords on 4-byte or 32-bit boundaries. This allows the network processor to present 4 bytes at a time to the CAM array. By forcing keywords to be word aligned, the input bytes no longer have to be applied to all columns of the CAM array. Instead byte 0 is applied to only columns j where $j \ mod \ 4 = 0$, byte 1 to columns j where $j \ mod \ 4 = 1$ and so on. Aligning keywords also allows the design to reduce the number of PEs to just one every 4 columns at the cost of slightly more complex PE functionality. The new PE algorithm is shown in Figure **??**. Line 1 generates the $M4$ signal which represents whether all the attached characters created matches. Line 2 generates a $E4$ signal which indicates that the keyword ends in this 4-character grouping. Line 3 is a check if all characters match and if this PE is at the beginning of a word. Line 5 is a check if all characters match and the previous PE indicated a match as well. Finally line 10 generates a word match (WM) signal if the PE is at the end of a keyword. Figure **??** shows the previous example using word-aligned keywords.

4.4.1 Value Output

The PE array outputs a word match signal that indicates that a match was found. However, for the lookup cache to be useful, we also need the value that corresponds to the lookup match. The PE number is used as an index into a map table memory which stores either the 32-bit value itself or an indirect pointer to the value object. The map table has as many entries as there are PEs or $n/4$. If keywords are very long, there is the potential for holes in the map table, as shown in Figure **??**.

4.4.2 Cache Entry Replacement

Replacement in a cache with variable sized keys provides challenges that do not exist in a normal cache. If keys and values are equal sized, one can simply mark an

entry as invalid and then reuse that entry without loss of capacity. With variable sized keys, however, when we mark an entry as invalid, the next key to be inserted may or may not fit into that space, thus leading to holes in the CAM array space. In order to avoid this problem, whenever an entry in the cache is invalidated, all entries that are to the right of the invalidated entry are shifted an appropriate number of elements left. For example, in the array shown in Figure **??**, if the *"fubar"* entry is deleted, we simply shift the *"bar"* entry and all subsequent keywords 8 elements to the left. Shifting in a CAM array is easily accomplished by wiring connections from each CAM cell to the CAM cell in the previous column. The shift process can be done in a number of cycles equal to the number of shifts. Deletion time is thus proportional to m. We have not, as yet, addressed replacement policy, i.e. the decision as to which entries to delete. The lookup cache always maintains track of the last valid entry in the CAM array, so that it knows where to add a new entry. If there is not enough space to store a new entry, the cache must then invoke a replacement policy. Though currently not implemented, the lookup cache can provide mechanisms for simple replacement policies such as LRU by providing counters for each entry. A LIFO or FIFO replacement policy is trivially achieved by always shifting out the leftmost or rightmost entry respectively. More complicated replacement policies can be handled by the network processor.

4.5 Results

The signature match processor for generic cache lookup was designed using the Xilinx Virtex- II Pro FPGA (XC2VP230-7). The design was written in VHDL and Xilinxs ISE 6.3i design environment. The design was implemented on the Xilinx XUP Virtex II Pro Development board using Xilinx's Embedded Development Kit. The Xilinx Virtex-II Pro FPGA contains two embedded 32-bit PowerPC 405 RISC microprocessors and numerous soft ip cores The Xilinx XUP Virtex II Pro Development

Number of characters	256	512	1024
Slices	2403	4812	9880
Frequency (MHz)	380.1	476.9	460.2
Throughput (MB/s)	1520	1908	1841
Searches per second	254M	318M	307M

Table 4.1: FPGA Implementation Characteristics

board consists of the Virtex- II Pro FPGA with a number of peripherals which can be used to build a complex system. The board runs with a clock signal of 100Mhz frequency. The ISE was used for synthesis, place and route of the design written in VHDL. Mentor Graphics ModelSim was used for functional verification of the behavioral and place and route model of the design. The EDK tools were used to integrate the hardware design with an embedded processor based system. 'C' programming was used to program the embedded processor. EDK tools provide a 'C' compiler and are used to generate the updated bitstream to directly download on to the FPGA.

A lookup cache with different sized **hosts** files was implemented. The FPGA implementation characteristics are shown in Table **??**. The results show that the cache can get very high throughput using an FPGA implementation. Assuming an average keyword size of 6 characters, the cache can perform over 300 million searches per second.

For comparison, a simple software string search test program was implemented using the GNU C library's **hsearch** function. Random strings were inserted into a hash, and then the hash was continuously queried for those strings. The test program

Number of characters	256	512	1024
Time per search (ns)	1128	1305	1582
Searches per second	887K	766K	632K
Throughput (MB/s)	5.32	4.60	5.06

Table 4.2: Software Implementation

was run on a 1GHz PowerPC computer and timed for just the search process, and the results are shown in Table **??**. As would be expected, the hardware implementation performs the search much faster - in fact nearly three orders of magnitude faster. Moreover, note that the speed of the hardware implementation is not dependent on the size of the cache. In a software cache, the size of the cache does impact performance even in a presumably constant order hash-based algorithm. This is because as the size of the cache gets larger, the hash will tend to incur more collisions leading to non constant-time behavior. Tree based algorithms are at best $\log n$ performance, so offer no better performance than a best-case hash implementation.

The FPGA implementation provides a proof-of-concept, but a custom VLSI implementation should get significantly higher throughput because of higher operating frequencies. More importantly, a fully custom design can provide higher densities and thus larger cache sizes. We expect the CAM cost per keyword byte to be roughly 78 transistors, comprising of 72 transistors for 8 9-transistor CAM bit cells and 6 transistors for the end of keyword cell. Each PE consumes roughly 64 transistors and each map table entry consumes 384 transistors. Noting that each PE and map table entry corresponds to 4 characters, the overall cost per keyword byte is 190 transistors. Including associated control logic, a 64KB cache would probably consume roughly 13M transistors, which is a small fraction of the transistor budget on modern microprocessors. As a point of comparison, Intel's upcoming Montecito CPU uses over 85% of the available 1.7 billion transistors on its L3 cache [**?**].

In this chapter, a new architecture for a hardware based lookup cache using an innovative CAM-based design is explained. The cache is unique in that it can accommodate variable string based searches making it amenable to a wide variety of applications. Applications that are heavily dependent on table lookups can benefit greatly from the use of such a cache. This design can process incoming streams at rates of over 10 Gb/s. This is more than sufficient to handle network applications that

require lookup capabilities such as DNS resolution, LDAP queries, and network intrusion detection. Non-networking applications including database queries, dictionary search, linking are amongst others.

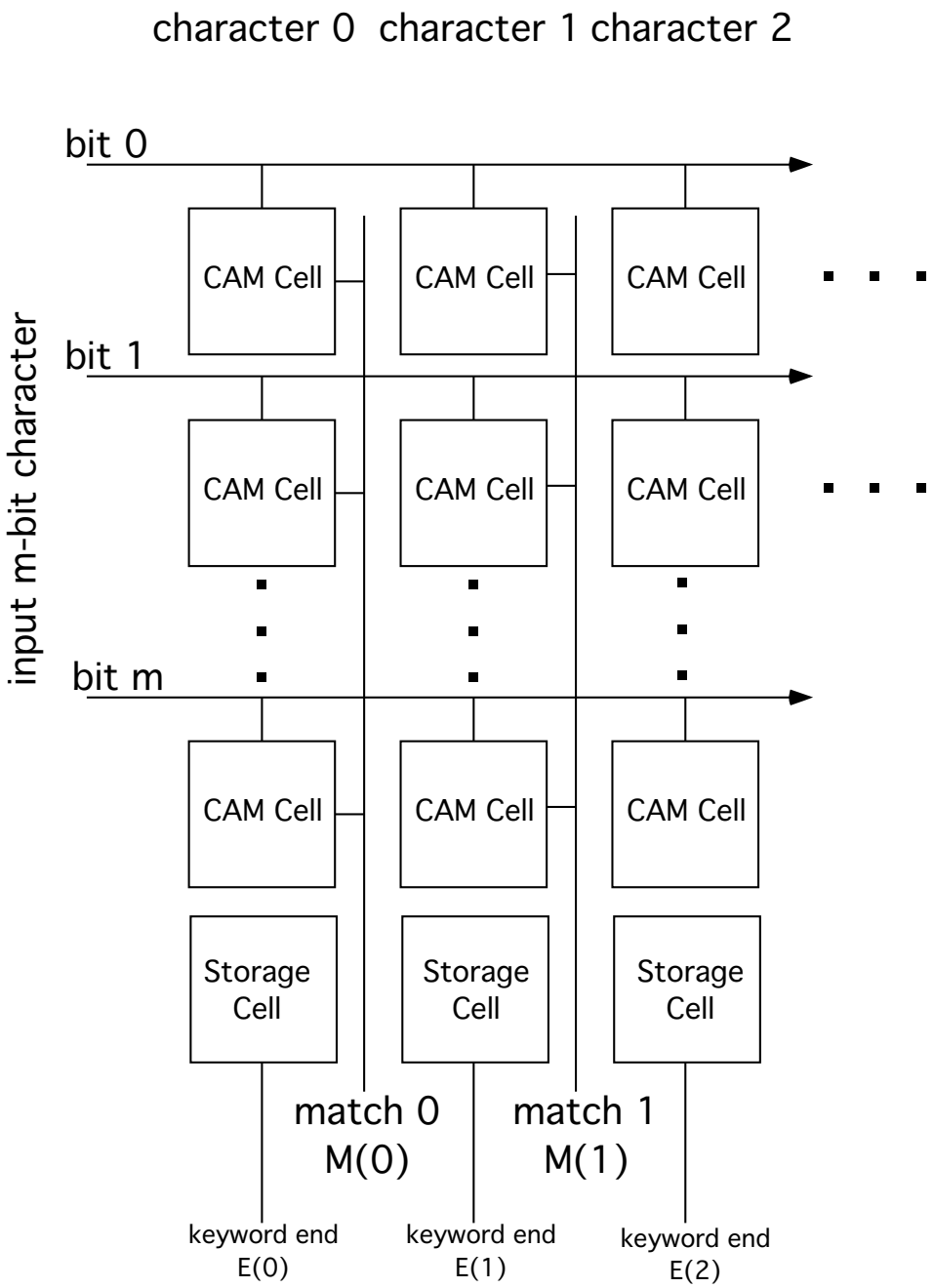

Figure 4.3: Character Match Array

```
1: if (M(i) = 1 and (E(i-1)=1 or i=0)) then
2:   PE(i)=1
3: else if (PE(i-1)=1 and M(i)=1) then
4:    PE(i)=1
5:    PE(i-1)=0
6: else if PE(i)=1 and E(i)=1 then
7:    PE(i)=0
8:    WM=1
9: end if
```

Figure 4.4: PE Algorithm.

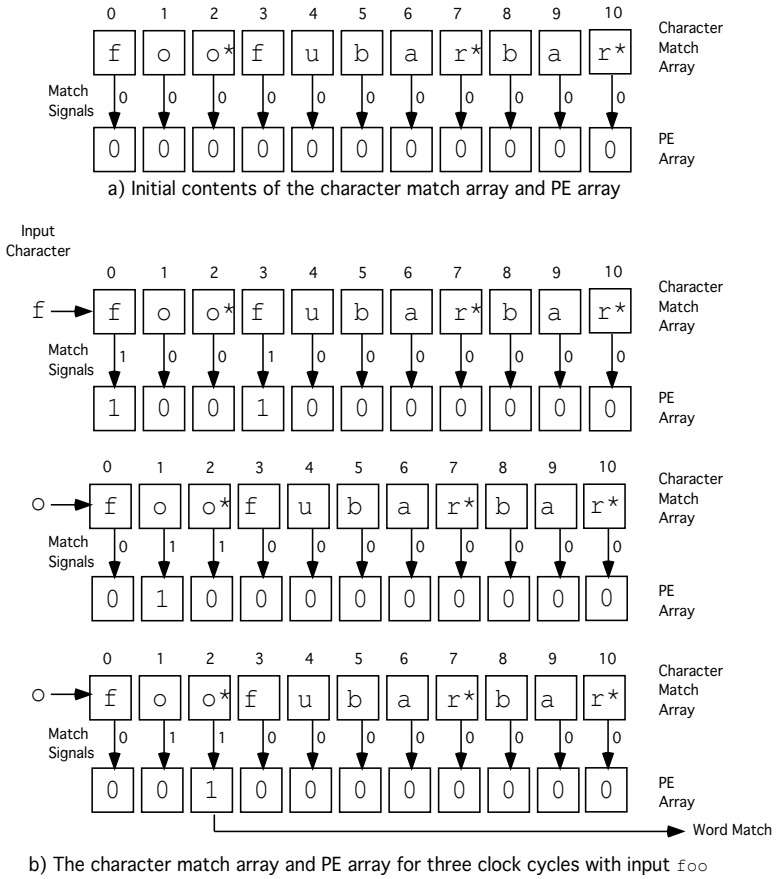

Figure 4.5: Lookup Cache Example.

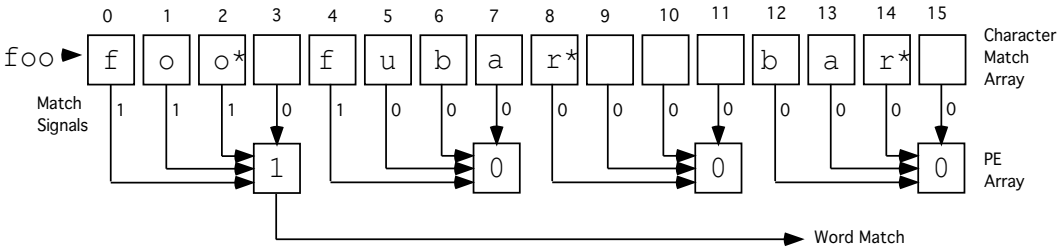

Figure 4.6: Lookup Cache Example with Word Aligned Keywords.

```
1:  M4(i) = M(4*i) and E(4*i) or
            M(4*i) and M(4*i+1) and E(4*i+1) or
            M(4*i) and M(4*i+1) and M(4*i+2) and E(4*i+2) or
            M(4*i) and M(4*i+1) and M(4*i+2) and M(4*i+3)
2:  E4(i) = E(4*i) or E(4*i+1) or E(4*i+2) or E(4*i+3)

3:  if (M4(i) = 1 and (E4(i-1)=1 or i=0)) then
4:     PE(i)=1
5:  else if (PE(i-1)=1 and M4(i)=1) then
6:     PE(i)=1
7:  else
8:     PE(i)=0
9:  end if

10: WM = PE(i) and E4(i)
```

Figure 4.7: PE Algorithm with Word-Aligned Keywords.

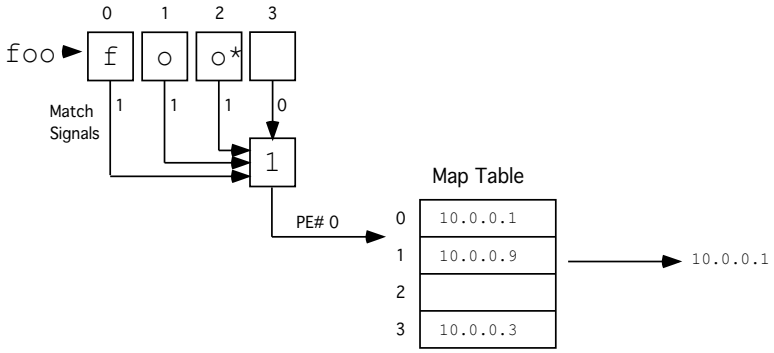

Figure 4.8: Word Match Value Out

Chapter 5

Conclusions

In this thesis, a novel architecture for a hardware based network intrusion detection system (NIDS) using an innovative CAM-based signature match processor is introduced. Based on the current implementation of the SMP, we can process incoming streams at rates of nearly 4 Gbps. This is more than sufficient to handle intrusion detection on current gigabit networks. This thesis also presents a unique design of a priority address encoder that will generate addresses even in cases when there are multiple matches within a packet. The SMP design also opens opportunities in other applications besides NIDS. Any lookup that is based on nonfixed- size keys seems to be an ideal candidate to take advantage of these SMPs. Some examples include directory lookup in network storage applications, DNS lookup, and LDAP processing. These are all applications that require large amounts of computational power to perform string matching lookup based operations. We are investigating the use of SMPs in these applications as well as developing extensions of the SMPs to support wildcards and approximate word matching capabilities as well. Other research directions include improving the power characteristics of the SMPs.

This thesis also presents a new architecture for a hardware based lookup cache using an innovative CAM-based design. The cache is unique in that it can accommodate variable string based searches making it amenable to a wide variety of applications.

Applications that are heavily dependent on table lookups can benefit greatly from the use of such a cache. We envision that there are several network processing applications that could take advantage of the lookup cache, and based on the prototype implementation of the lookup cache, we can process incoming streams at rates of over 10 Gb/s. This is more than sufficient to handle network applications that require lookup capabilities such as DNS resolution, LDAP queries, and network intrusion detection. Non-networking applications include database queries, dictionary search, linking amongst others.

Bibliography

[BC04] Long Bu and John A. Chandy. Fpga based network intrusion detection using content addressable memories. In **Proceedings of IEEE Symposium on Field-Programable Custom Computing Machines**, April 2004.

[BFRV92] Stephen D. Brown, Robert J. Francis, Jonathan Rose, and Zvonko G. Vranesic. **Field-Programmable Gate Arrays**. VLSI, Computer Architecture and Digital Signal Processing. Kluwer Academic Publishers, Boston, MA, 1992.

[BM77] R. S. Boyer and J. S. Moore. A fast string searching algorithm. **Communications of the ACM**, 20(10):762–772, October 1977.

[BP04a] Z. K. Baker and V. K. Prasanna. Automatic synthesis of efficient intrusion detection systems on FPGAs. In **Proceedings of International Conference on Field Programmable Logic and Applications**, 2004.

[BP04b] Z. K. Baker and 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**, pages 135–144, APR 2004.

[BYG92] R. Baeza-Yates and G. H. Gonnet. A new approach to text searching. **Communications of the ACM**, 35(10):7482, October 1992.

[CMS04] Young H. Cho and William 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**, pages 125–134, April 2004.

[CS04] C. Clark and D. Schimmel. Scalable multi-pattern matching on high-speed networks. In **Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines**, pages 249–257, April 2004.

[FRV90] R. J. Francis, J. Rose, and Z. Vranesic. Chortle-crf: A technology mapping program for LUT based FPGAs. In **Proceedings *27th* *DAC***, June 1990.

[FRV91a] R. J. Francis, J. Rose, and Z. Vranesic. Chortle-crf: Fast technology mapping for LUT based FPGAs. In **Proceedings *28th* *DAC***, November 1991.

[FRV91b] R. J. Francis, J. Rose, and Z. Vranesic. Technology mapping of LUT based FPGAs for performance. In **Proceedings *ICCAD-91***, June 1991.

[GDD+02] Maya Gokhale, Dave Dubois, Andy Dubois, Mike Boorman, Steve Poole, and Vic Hogsett. Granidt: Towards gigabit rate network intrusion detection technology. In **Proceedings of International Conference on Field Programmable Logic and Applications**, pages 404–413, 2002.

[GPK03] Bruce Gamache, Zachary Pfeffer, and Sunil P. Khatri. A fast ternary CAM design for IP networking applications. In **Proceedings of International Conference on Computer Communications and Networks**, October 2003.

[Hau95] Scott Hauck. Multi-fpga systems. Master's thesis, University of Washington, 1995.

[HFC02] B. L. Hutchings, R. Franklin, and D. Carver. Assisting network intrusion detection with reconfigurable hardware. In **Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines**, pages 111–120, 2002.

[HYNT88] M. Hirata, H. Yamada, H. Nagai, and K. Takahashi. A versatile data string-search VLSI. **IEEE Journal for Solid State Circuits**, 23(2):329–335, April 1988.

[Jud03] Gregg Judge. FPGA architecture ups intrusion detection performance. **CommsDesign.com**, September 2003.

[KMP77] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. **SIAM Journal on Computing**, 6(2):323–350, 1977.

[KR87] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. **IBM Journal of Research and Development**, 31(2):249–260, March 1987.

[Mal] David Maliniak. Basics of fpgas design. In **Mentor Graphics Technical Publication**.

[MC92] Tim Moors and Antonio Cantoni. Cascading content addressable memories. **IEEE Micro**, 12(3):56–66, May/June 1992.

[MF93] A. J. McAuley and P. Francis. Fast routing table lookup using CAMs. In **Proceedings of IEEE INFOCOM**, volume 3, pages 1382–1391, March 1993.

[MLLP03] James Moscola, John Lockwood, Ronald P. Loui, and Michael Pachos. Implementation of a content-scanning module for an internet firewall. In **Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines**, 2003.

[MNS+91] R. Murgai, Y. Nishizaki, N. Shenoy, R.K. Brayton, and A. Sangiovanni-Vincentelli. Logic synthesis for programable gate arrays. In **Proceedings *ICCAD-91***, pages 620–625, November 1991.

[MNS+92] R. Murgai, Y. Nishizaki, N. Shenoy, R.K. Brayton, and A. Sangiovanni-Vincentelli. An improved synthesis algorithm for muliplexor-based pgas. In **Proceedings *ACM/SIGDA First International Workshop on Field Programable Gate Arrays***, pages 97–102, 1992.

[MSB91] R. Murgai, N. Shenoy, and R.K. Brayton. Performance directed synthesis for table look-up programable gate arrays. In **Proceedings *ICCAD-91***, 1991.

[MSBSV91] R. Murgai, N. Shenoy, R.K. Brayton, and A. Sangiovanni-Vincentelli. Improved logic synthesis algorithms for table look up architectures. In **Proceedings *ICCAD-91***, 1991.

[MTH+90] Masato Motomura, Jun Toyoura, Kazumi Hirata, Hideyuki Ooka, Hachiro Yamada, and Tadayoshi Enomoto. A 1.2-million transistor, 33 MHz, 20-b dictionary search processor (DISP) ULSI with a 160-kb CAM. **IEEE Journal for Solid State Circuits**, 25(5):1158–1164, October 1990.

[Muk80] A. Mukhopadhyay. Hardware algorithms for string processing. **IEEE Computer**, pages 508–511, 1980.

[Mus98] Music Semiconductors. Implementing an ARP cache using the MUAA CAM family. Technical Report Application Note AN-N23, Princeton, NJ, February 1998.

[MYE92] Masato Motomura, Hachiro Yamada, and Tadayoshi Enomoto. A 2k-word dictionary search processor (DISP) with an approximate word search capability. **IEEE Journal for Solid State Circuits**, 27(6):883–891, June 1992.

[SBC05] Janardhan Singaraju, Long Bu, and John A. Chandy. A signature match processor architecture for network intrusion detection. In **Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines**, pages 235–242, April 2005.

[SP01] R. Sidhu and V. K. Prasanna. Fast regular expression matching using FPGAs. In **Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines**, pages 227–238, April 2001.

[SP04] Ioannis Sourdis and Dionisios Pnevmatikatos. Pre-decoded CAMs for efficient and high-speed NIDS pattern matching. In **Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines**, pages 258–267, April 2004.

[WM92] S. Wu and U. Manber. Fast text searching allowing errors. **Communications of the ACM**, 35(10):83–91, October 1992.

[WS89] J. P. Wade and C. G. Sodini. A ternary content addressable search engine. **IEEE Journal for Solid State Circuits**, 24(4):1003–1013, August 1989.

[WWMD05] J. Wuu, D. Weiss, C. Morganti, and M. Dreesen. The asynchronous 24MB on-chip level-3 cache for a dual-core Itanium®-family processor. In **Proceedings of IEEE International Solid-State Circuits Conference**, pages 488–490, February 2005.

[www] www.Xilinx.com. In **Xilinx manual**.

[ZCI+04] Qiong Zhang, Roger D. Chamberlain, Ronald S. Indeck, Benjamin M. West, and Jason White. A massively parallel data mining using reconfigurable hardware: Approximate string matching. In **"18th International Parallel and Distributed Processing Symposium (IPDPS'04) - Workshop 15 "**, April 2004.