VLSI DESIGN
2000, Vol. 11, No. 4, pp. 405–415
Reprints available directly from the publisher
Photocopying permitted by license only

A Chip for a Routing Table Based on a Novel Modified Trie Algorithm

D. TORRES*, A. LARIOS and M. GUZMÁN

CINVESTAV del IPN, Research and Advanced Studies Center of National Politechnic Institute, Apartado Postal 31-438, Guadalajara, C.P. 44550, Jalisco, México

(Received 5 June 1999; In final form 10 February 2000)

The design for a routing table circuit for Ethernet-, IP- and ATM-applications is presented. Starting point for the design was an object-oriented general behavior of the routing table. The selected data structure for the routing table is based on a *modification* of the structure denominated *trie*, *saving one search level and memory space*. The architecture for searching and sorting of data, implemented in hardware, is explained. This *modified trie* stores 64 K addresses and the associated data, achieving a high performance too. The circuit, which can support a flow of 500000 frames/s, is connected to the PCI Bus. For the implementation a FLEX10K100 from Altera Company was used.

Keywords: Routing; Broadband; Hard-software codesign; Algorithm; Implementation; Testing

1. INTRODUCTION

There is a growing interest in high speed data transfer by adopting the Asynchronous Transfer Mode (ATM) technique on a local basis and/or by using high speed LANs, *e.g.*, Fast Ethernet. At the Research and Advanced Studies Center (CINVESTAV) of the National Polytechnic Institute (Mexico) a small high speed switch as a prototype for research and academic purposes is being designed and built. The prototype (see Fig. 1) can be functionally broken down into [1] two main parts: the *switch fabric*, and the *line cards*. The switch fabric works with 56-bytes cells, denominated internal cells [1], of which 53 bytes correspond to ATM cells and 3 bytes are dedicated internally to the routing of the incoming cells to the output port. The line cards, which interface the Fast Ethernet LANs [2] and the PSTN network [3], realize the segmentation of frames into cells and *viceversa* for data networks. Consequently, for routing purpose they need a *routing table*.

Packets are sent, using the routing principles. *Routing is an important function* for data and emerging ATM networks. Conceptually, functions associated with the routing can be divided into two

*Corresponding author. Tel.: 52-3-684-1580, Fax: 52-3-684-1708, e-mail: dtorres@gdl.cinvestav.mx

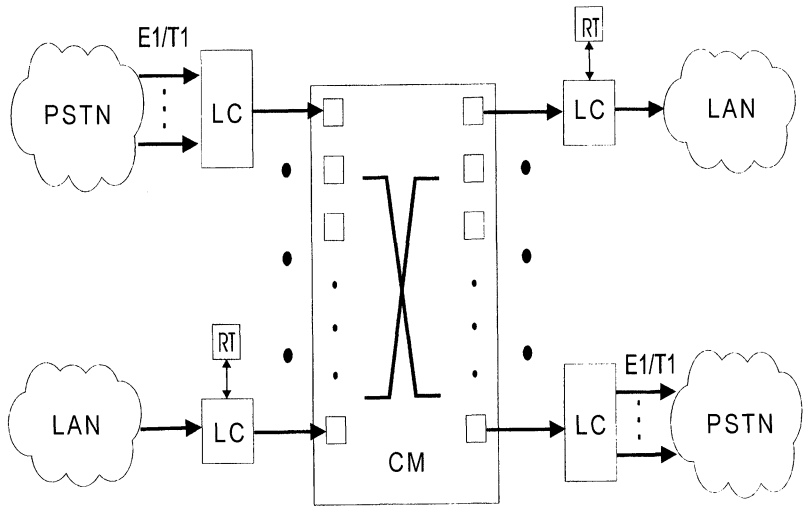

FIGURE 1 A general switch.

groups: the first one is used *to determine the routing tag for a data packet, i.e., searching*; while the second one is used to *add, change*, or *delete routes*, and these are related with *sorting* and *storing of data items*. Actions of the second group occur at much slower rates than routing tag searches. Therefore, *search* is the most dominant function for *a routing table* and others *abstract data types. Main objectives of this paper are*:

1. *to describe and to define the general behavior of a routing table,*
2. *to explain the need of an algorithm for the routing table and the selected solution: a modified trie algorithm,*
3. *to describe the design and the implementation of the selected algorithm on a CPLD circuit.*

2. ALGORITHMS FOR THE ROUTING TABLES

Taking into account the routing function, a functional structure of a **Routing Table Circuit** (**RTC**) is shown in Figure 2. It can be seen as a *specialized processor* or *multiprocessor*, **Proc**, with a *capacity to store* the orders or *commands* (strings of variable or fixed length) in a **Queue**. This queue is a buffer that works under the *producer–consumer* principle. The processor fetches the commands from the queue, and then decode and process them. This process is performed accessing an *external memory*, **Mem**, where address and tag data are stored and updated, and the requested data item is returned through an output port called **Port**.

2.1. Routing Table Behavior

In data networks, every packet has one or more fields building the packet header. The header of internal packets contains an identifier associated with the *packet address*, which helps to the routing of packets from one node to the next one; for this reason it is denominated *routing tag*. In the cell switch fabric, the routing tag is used to map the logical channel number in each packet into another number representing the new channel or a path through a switch to an output port. Also, the routing function f maps a set of natural numbers in another one. Therefore, a routing table must contain an *appropriate number of addresses* and *their associated tags*, executing the sent of packets to the next node. In our case, the routing table

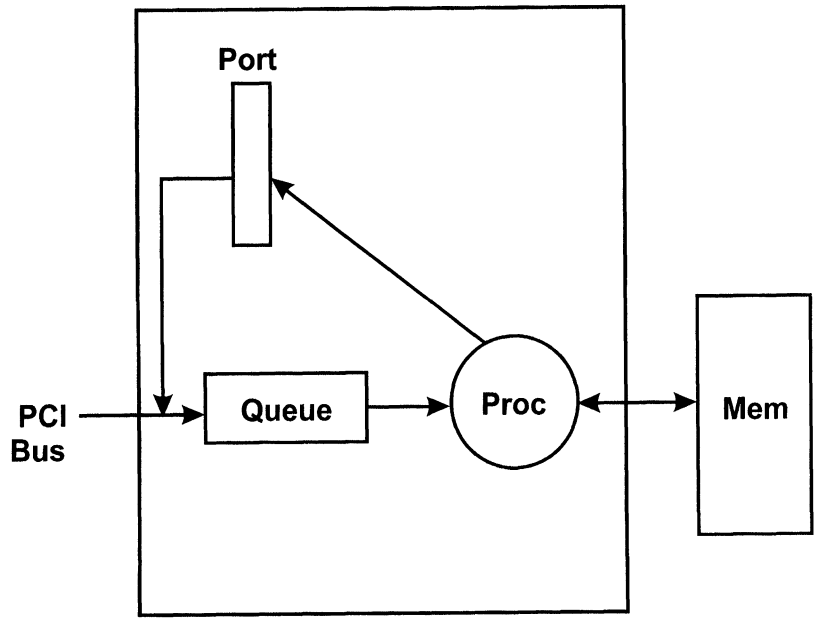

FIGURE 2 Functional structure of the routing table.

must fulfill the following requirements:

- *to store a minimum number of 2048 addresses* and the *associated routing tags*,
- to *search* the routing tag for each input address,
- to support the traffic of several networks (at least 200 Mbps),
- to work with Ethernet (48 bits), IP (32 bits) and ATM (24 bits) addresses.

In order to fulfill these requirements a set of basic and additional functions are defined:

Basic functions are

- to *configure* the RTC for Ethernet, IP or ATM
- to *update* the newest addresses,
- to *delete* the oldest addresses,
- to *insert* a new address,
- to *search* the routing tag for an input address, and

Additional functions are

- to *read* the number of busy *node clusters* (a chain of m addresses) and
- to *read* the number of unsuccessful searches, and
- to *test* a memory block.

A software implementation is possible formalizing a *routing table* as an *abstract data type* **adt rt**, beginning, for example, with a class derived from a base class Queue, *i.e.*, **class rt**: **public** Queue, a set of **basic** and **additional** operations and a set of equations, where the types **tag** and **address** are essentially **adt** too. Here the basic idea is to separate: what the data structure should do from any particular implementation. Therefore, it is possible to define a routing table in the following way.

DEFINITION 1 A *routing table* can be defined as a *data structure* or an abstract data type S with a *storing* and *sorting* capacity of data items d_i ⟨address⟩ and their associated identifiers i ⟨tag⟩; where are implemented a *search algorithm* g, a *set of basic operations* such as: Search(S, i); Insert(S, i); Delete(S, i); Update(S, i); Configure$(S, \text{mod}\, e)$, and *additionally others operations* for tests and supervision of its behavior.

2.1.1. *The Need of Algorithms*

Searching quantities of data for a particular item, and *data sorting* are two of the most frequent

applications of computer science. Scientists have created several data structures, such as *dictionaries*, that allow to store data in the form of *records* d_i matching with a given *key* i, *i.e.* $\langle i, d_i \rangle$, and easy to update as well. Besides, they have developed *methods* and *searching algorithms* to retrieve data efficiently.

Many data structures have been developed for storing data for rapid retrieval, namely, arrays, trees, binary trees and so on. The *simplest method of searching* is to store the records in an array. When a new record is to be inserted, the system puts it at the end of the array; when a search is to be performed, the array is analyzed *sequentially*. This sequential search is too slow for some applications. As a result several *elementary* and *advanced searching methods* as well as a variety of data structures have been developed.

In the development of algorithms, it is important to take into account their major properties, specifically: *correctness*, *performance or efficiency* and *ease of implementation*. In practice however, it is difficult to achieve simultaneously all these requirements.

Most routing tables [4] have been implemented in software with a certain number of different approaches. *The simplest implementation* consists of *a data table or dictionary* stored in a conventional memory using the address of each memory location as key i; that is the *most time efficient*, because it needs only one access to get the data or record, *i.e.*, *complexity* $O(1)$. This approach wastes many memory locations. The reader can calculate how many memory locations are necessary to store in a simple table all possible addresses of 48/32 or 24 bits. Therefore, it is convenient to develop other data structures. We can formulate the question as follows.

Problem 1 The *real problem* is that the *space of all possible addresses* S_p is too large to fit in a small table and, the *space of the real addresses* $S_r \subset S_p$ is also large as compared with the *space of the target addresses* $S_t \subset S_r$. Then, a good *routing function* f_r must map $f_r : S_p \rightarrow S_t$ with the help of a search algorithm g of complexity $O(n) < \tau$, where n is the number of accesses to find the tag associated with a given address, and τ a time interval (time constraint).

In principle, there is a *compromise between the search algorithm complexity* and *the memory size*. Simple search algorithms allow to obtain high performances but require large memories whereas, complex search algorithms using small memories produce low performances. Consequently, there is a trade off between the following elements: *performance*, *memory size* and *system costs*.

2.1.2. *The Selected Solution: a m-trie*

Due to the applications of the routing table, it is very important to have, an algorithm with *high performance* or *efficiency*, and *data structure* with a memory as little as possible. Foremost among the possible algorithms that satisfy the compromise between memory size and search time, outshines the basic trie. A *basic trie* is, [5–7] essentially, a compromise between a *simple table* and a *linked list*. This algorithm attempts to reduce the address-space, which contains all the possible addresses, to a space, with the target addresses (see the above mentioned problem). The basic idea is to not store data items in tree nodes at all, but rather to put all the data items (*routing tags in our case*) in external nodes of the tree. For that purpose, it divides a key or given address of n bits into segments of k bits each one and it builds l levels, given by $l = n/k$. That is a *regular partition* of n-bit words. A trie has two types of nodes. The first one, called *branch node*, contains pointers only, while the second one includes an *element* or *data node*, for the corresponding *data item*. The pointers are used to access the routing tags.

DEFINITION 2 A *basic trie* is a tree, in which data have special nodes called data nodes and where the branching at any level is determined not by the entire key value (n bits) but by a regular partition of the n bits words in l segments of k bits; the nodes containing pointers are called branch nodes.

In this data structure, the search function is straightforward with a $O(l = n/k)$ as the worst case search time.

Several refinements to the *basic trie* have been proposed and studied, *e.g.*, **Patricia** (**P**ractical **A**lgorithm **T**o **R**etrieve **I**nformation **C**oded **I**n **A**lphanumeric) [6]. From the literature, we can conclude that: (1) there are very few real-time dedicated-hardware routing tables with severe time constraints, (2) tries permit to realize the same operations as a *dynamical set* and additionally other functions as *RangeSearch*( ), (3) a *dynamical set* can be defined as a set, where each element has associated an identifier called *key i* and where exists a *total order relation* [6] on this set of keys, (4) among the *operations on a dynamical set S*, given the *key i* are: *Search* (S, i); *Insert* (S, i); *Delete* (S, i); *Minimum* (S, i); *Maximum* (S, i); *Predecessor* (S, i); *Successor* (S, i); *RangeSearch* (S, i_1, i_2) (to find all records such that $i_1 < i < i_2$). Consequently, tries are appropriated data structures for our purpose.

Due to the above mentioned considerations, the selected algorithm is a *modified trie*, denoted here as *m-tries*. It consists of an adaptation of a basic trie with the following modification: instead of using l levels for pointers (branch nodes) $l-1$ levels for branch nodes are used, the level $l-1$ contains pointers to *node clusters*, and each node cluster has a grouping of 2^k *data nodes*. The *address associated with the node cluster* is hereafter referred to as "*base address*", which is built from the first $l-1$ segments of k bits of the given word. Then *two movements* are necessary in the trie: the first one *through the branch nodes*, and the second one *through the node cluster*.

DEFINITION 3 A *m-trie* is a *k-way basic trie* with $l-1$ levels (instead of the l levels of the basic trie), which has in the last level a grouping of node cluster of 2^k data nodes (instead of the simple data nodes); and where branch nodes contains only pointers (see Fig. 3).

PROPOSITION 2 *If the space of all possible addresses S_p is of size 2^n using n-bit words and the space of target addresses S_t is of size 2^m, where $m \leq n-k$; in order to store the same number of routing tags $N = 2^m$, the number of pointers to data clusters in the level $l-2$ of a m-trie is given by 2^{m-k}, where $l = (n/k)$ is the number of levels in the partition of the words, and where the search function is straightforward with a constant search time given by $O(l-1)+\varepsilon$.*

Proof As each data cluster has 2^k data nodes, in order to keep in memory 2^m data nodes (for the routing tags) are necessary 2^{m-k} *pointers* or *base addresses*, consequently the level $l-2$ of the *m*-trie must have 2^{m-k} *pointers*. The number of branch nodes at each level is given by 2^{ik}, where $i \in \{0, 1, 2, \ldots, l = n/k\}$, therefore at the level $l-2$ of the *m*-trie there are $2^{(l-2)k}$ branch nodes. As $2^{(l-2)k} = 2^{n-2k}$ and $m \leq n-k$ then $2^{n-2k} \geq 2^{m-k}$ and *the target space* S_t can be stored in *the m-trie*.

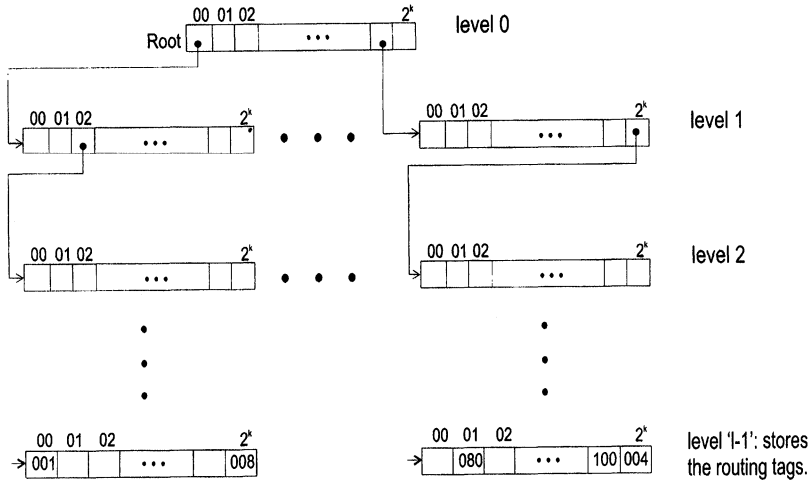

FIGURE 3 Modified trie with $l-1$ levels.

As a *m-trie* has only $l-1$ levels and because all data nodes are in the same level, the number of accesses to search a *base address* is $O(l-1)$, and where ε is the time to access a specific address of the *node cluster -data nodes-* and *to get the corresponding tag*. ■

Besides, it is possible an *irregular partition of* n-bit word*s* with $l-1$ of k bits and the last segment of r bits. In this case, in order to store the same target space S_t of size 2^m are necessary 2^{m-r} pointers at the level $l-2$.

Main advantages of the *m-trie* are that it decreases by one the number of levels and has a constant search time, comparing with the basic trie, saves memory space.

2.1.3. Routing Table for IP and ATM

The implemented routing table can be configurated to work with IP and ATM. For each case a different number of levels is generated, due to change of the word length.

3. RTC IMPLEMENTATION

Figure 4 shows a general architecture for the routing table. It is composed of three main parts: *the PCI Bus Interface*, *the Memory* and *the Routing Table Control*. The *PCI Bus Interface* converts the PCI signals into appropriate signals for the RT Control and *viceversa* [8, 9]. The memory must be large enough in order to store, at least: (1) the number of required addresses, (2) the associated routing tags and, (3) complementary data for control and management of this circuit. The *RT Control* is the kernel of the circuit and the most complex part, it contains the selected algorithm.

The specific and implemented architecture of the routing table is shown in Figure 5. The *processor* can be seen composed of two processors driving five **FSM** (**F**inite **S**tate **M**achines). These FSMs were developed for this circuit [10] and they execute the operations mentioned in Subsection 2.1. A *first processor*, called **DSAC** (**D**ata **S**tructure **A**lgorithm **C**ontrol), controls the main functions of the routing table: *insertion*, *search* and

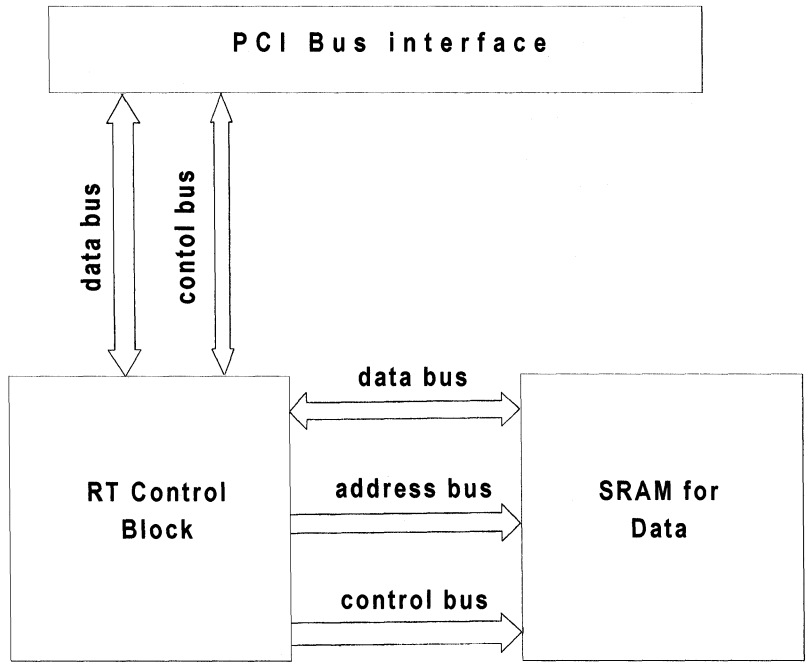

FIGURE 4 Routing table structure.

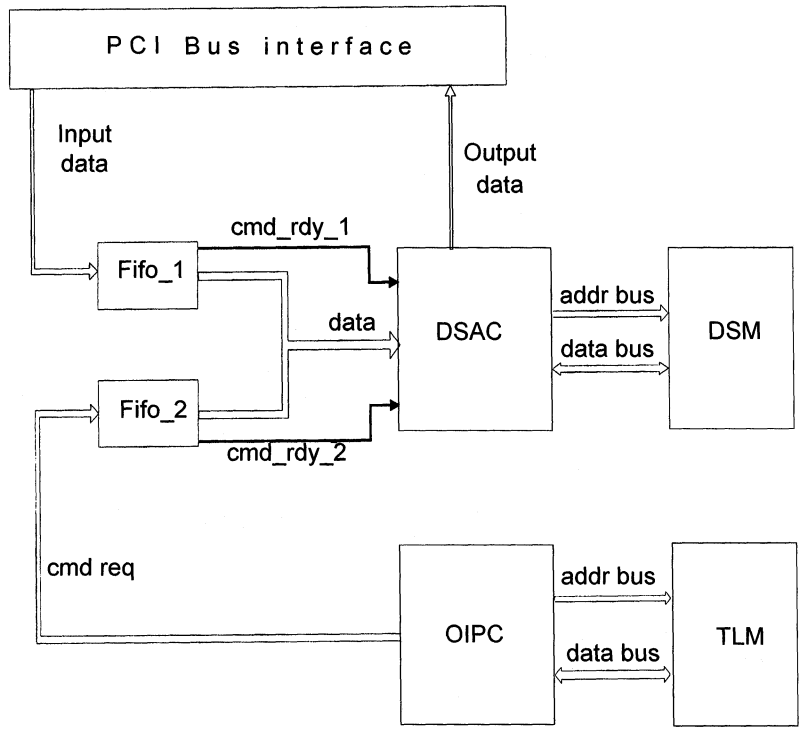

FIGURE 5 Specific routing table architecture.

deletion functions and *manages* the memory **DSM** (**D**ata **S**tructure **M**emory), which contains the *records* including the data for the movement through the branch nodes of the trie. This processor fetches, in an internal queue, the commands indicating the function that must be performed. Commands arrive to the internal queue by two ways: from Line Cards through the PCI Interface and from the second processor. A *second processor*, denominated **OIPC** (**O**ldest **I**nformation **P**olicy **C**ontrol), has the task to detect old addresses and to request its deletion by a command, which is sent to the internal queue. The OIPC is applied to keep free space for the insertion of new addresses. For this purpose, this processor has a memory called **TLM** (**T**ime **L**abel **M**emory), containing data about the age of each record stored in the DSM, and checks the memory periodically to detect old addresses. It says that *an address is old*, if it is the oldest one or it belongs to those whose life time has expired. Period and life time can be programmed by the user. Each one of the functions mentioned before was implemented with one major FSM that includes some combinational logic [11–13].

For the chip design was used AHDL as programming language. In the design flow, a functional test of the blocks that are part of the system architecture [11, 14] is carried out. Starting from the algorithm description, the design was realized at *a logical level*. When this functional part was finished, subsystems and the whole system were tested. Finally for the design were taken into account all the *timing details*. Consequently, it is necessary to test that the implemented logic satisfies the *timing for the system requirements*.

3.1. Functions Implementation

In what follows, two majors of the basic functions of a routing table (see Section 2.1) will

be explained, namely, search and insert functions. Note that from this description, computational algorithms can be developed in a relative short time, but that is not the case for a hardware implementation under severe time constraints.

3.1.1. The Search Function

The *search function* searches and returns *a tag* associated with *a given address* (a *word of n-bits* is associated with this address); but rather, when the search is *unsuccessful*, this function return a null. In the following, the numbers between parenthesis correspond to those shown in Figure 6. In order to execute the search function, the next steps are performed, by the first processor DSAC, according with Figure 6:

1. obtain the *first k bits* (*first segment*) of the *given word* associated with the *searched tag* (1),
2. add these k bits or first segment to the root address of the trie to obtain the *address for the first level*, **addr_level_1**, in the *m-trie* (2),
3. get the data from memory (3), this data is a pointer ***p*** to a branch node,
4. *if the pointer **p** is equal to NULL*, the word or address isn't in the trie structure, and the search was unsuccessful and ends, *else* continue;
5. add the next segment of k bits of the given word to the *pointer **p***, to get the next level l in the trie, **addr_level_L**, if l is not the last level go to the step 3 (4), *else* continue,
6. build first the *base address* of the node cluster and with the last segment of k bits of the given word *the specific data node address*,
7. get the *routing tag* associated with the *searched address* (5).

Note from the algorithm for the search function and Figure 6, that in order to get a base address and the associated tag, multiple accesses to the same memory are necessary. Due to this it is difficult to implement techniques as pipe-line in order to achieve a better performance. Although the used RAM memory, in our circuit, has a 10 ns access time, this implementation could generate a

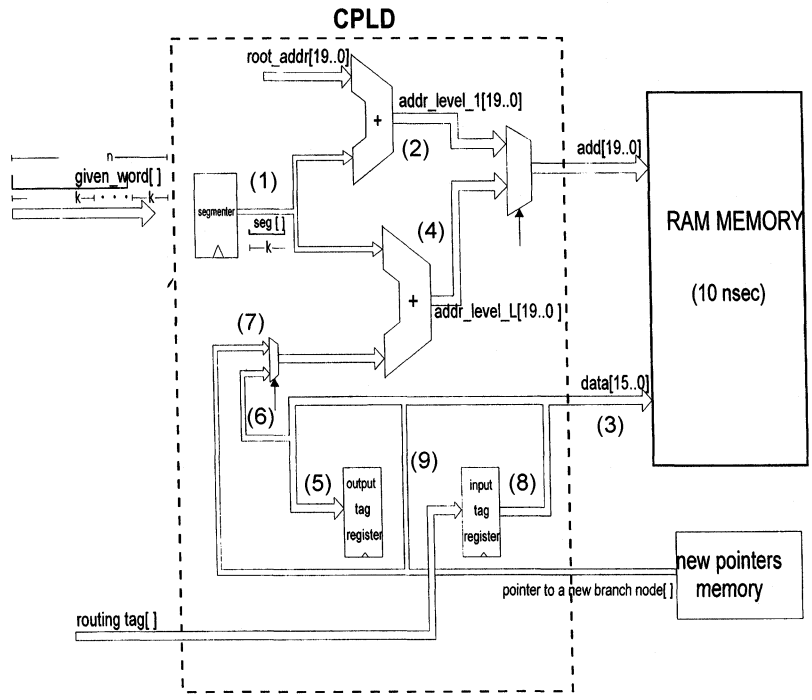

FIGURE 6 Simplified diagram of the **m-trie** implemented in hardware (without the second processor).

very significant delay, and has a *high circuiting complexity*.

3.1.2. The Insertion Function

The *insert function* adds or inserts a tag associated with an address in a specific memory location. Note that the complexity of this function is greater than the search function. For this process the following steps are executed:

1. obtain the *first k bits* (*first segment*) of the *given word* associated with the tag to insert (1),
2. add these k bits or first segment to the root address of the trie to obtain the *address for the first level*, **addr_level_1**, in the *m-trie* (2),
3. get the data from memory (3), this data is a pointer ***p*** to a branch node,
4. *if* the pointer ***p*** is different from NULL, *then* go to step 10, *else* the corresponding *base address* was not stored in the memory, and it must be built in several steps,
5. a *new pointer to a free branch node* ***bp*** is obtained from a memory dedicated to store these pointers, and it is inserted in the corresponding memory (9),
6. add the next segment of k bits to the pointer ***bp*** to get the next level l in the trie, **addr_level_L**;
7. *if* l is not the last level, *then* go to step 5, *else* the *base address* has been built,
8. a *new address location* (*data node address*) is built from the *obtained base address* and the last k bits segment of the given word (7),
9. insert the *routing tag in the data node address* (8) (end),
10. add the next segment of k bits to the pointer ***p*** to get the next level l in the trie, **addr_level_L**; *if* l is not the last level, *then* go to the step 3 (4), *else* go to step 8, because the base address has been found,

Performance The designed RTC can support tag searches in the order of 500 000 frames/s; therefore the *real throughput* depends on the *frame length*. Due to the introduction of voice over data networks, short frames become more frequently than before. For Ethernet networks, when 500 000 frames/s are processed, a real minimum data flow of 256 Mbps and a theoretical maximum of 6 Gbps can be reached. A linear relationship between the *throughput* (Mbps) and the *frame mean length* (Bytes) is given by $y = 4x$.

4. TEST PLAN

When designing a system, it is necessary to prove that it works according to the specifications. For that purpose, we must follow a test methodology. An example of them is the Built-In Self Test (BIST), in which test functions are embedded into the circuit itself [15]. Each block was tested separately and then some blocks were put together and tested as a subsystem. A similar procedure was used for debugging and system integration. The routing table system has the following blocks: PCI bus interface, two internal queues, external memory and two processors containing the mentioned five FSMs, which execute the functions associated with the *m-trie* structure. As the functionality of the RTC relies on the *correctness of the commands* stored in the internal command queue and of the *records* stored in the external memories, special tests were taken into account. Memories are very critical components, because their fault occurrence probability is higher than in other types of components, therefore memory tests are very important tasks, which in most cases should be realized on-line to detect some types of failures. Tests were realized on the following blocks:

- *internal command queue*, by write and read of commands,
- *external memories*, by write and read of data,
- *registers* and *ports*, by write and read of data,
- *processors*, for these elements were tested theirs associated FSMs, and
- the *corresponding functions* (*basic* and *additional*).

For the implemented circuit RTC were needed the following resources: a CPLD FLEX10K100 [16, 17], 64% of the logical cell and 230 pines.

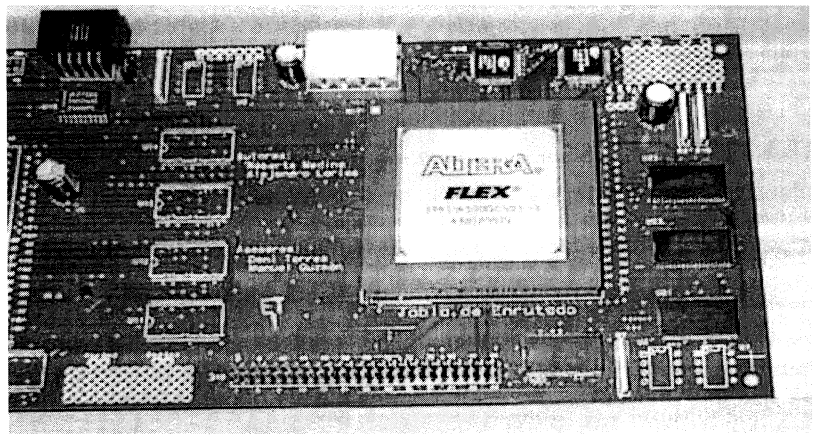

FIGURE 7 Implemented routing table.

Additionally, 2.5 Mbyte static RAM memories and 2 FIFOs 16×32 KB were used. The RTC is connected to the PCI Bus. Special considerations were taken into account [18, 19, 13, 20] and are given for the chip partitioning, clock signals distribution, and PCB design. The RTC occupies, without PCI-Interface, a surface of $4'' \times 4''$ of an 8-layer printed circuit board, see Figure 7.

5. CONCLUSIONS

Although a software implementation of a routing table can be solved in a short time, a hardware implementation under severe time constraints is, yet today, a very hard work. In this paper, we can conclude:

- The general behavior of a routing table is described and defined.
- The implemented architecture of the routing table circuit consists of the following blocks: PCI bus interface, two internal queues, external memory and two processors containing five FSMs. These blocks carried out the *basic and additional functions* described in this work.
- The selected and in hardware implemented algorithm for the RTC Control was a modification of the basic trie, called here *modified-trie*, *with* $l-1$ levels storing an address space of size 2^m and *with a constant search time given by* $O(l-1)+\varepsilon$. This *m-trie*, compared with *the basic trie*, saves *one level* and *memory space*.
- The implemented circuit can support an incoming traffic of more than 500 000 frames/s, in the worst case the time to search a tag was $\tau < 2\,\mu$s. For Ethernet networks, when 500 000 frames/s are processed, a real minimum data flow of 256 Mbps and a theoretical maximum of 6 Gbps can be reached.
- The implemented Routing Table Circuit can be configurated to support Ethernet, IP or ATM traffic.
- The RTC can store 64 K addresses or 1 K node clusters of 64 addresses each one.
- A high performance RTC was implemented on a CPLD FLEX10K100 from Altera Company, using additionally to the internal memory, an external one with access time of 10 ns.

Acknowledgement

The authors wish to express their gratitude to the Semiconductor Technology Center for the technical support given to this work.

References

[1] Torres, D., González, J., Guzmán, M. and Nuñez, L., A New Bus Assignment in a Designed Shared Bus Switch Fabric, *Proc. of IEEE ISCASS'99*, **I**, 423–426, Orlando, USA.

[2] Medina, R., Torres, D. and Guzmán, M., An Interface for a Fast Ethernet LAN Network, *Proc. of the International Symposium on Information Theory and Its Applications 1998*, pp. 564–567.

[3] Lopez, G., Torres, D. and Silva, J., *A Line Interface Circuit for 63 E1/T1*, in the development.

[4] Newman, P., Minshall, G., Lyon, T. and Huston, L., IP Switching and Gigabit Routers, *IEEE Communications Magazine*, pp. 64–69, January, 1997.

[5] Horowitz, E., Sahni, S. and Anderson-Freed, S., *Fundamentals of Data Structures in C*, Computer Science Press, 1993.

[6] Heilemann, G. L., *Data structures, Algorithms, and Object-Oriented Programming*. McGraw Hill, Inc., 1997.

[7] Pei, T. B. and Zukowski, C., Putting Routing Tables in Silicon. *IEEE Network Magazine*, pp. 42–50, January, 1992.

[8] PCI Local Bus Specification. Revision 2.1 PCI Special Interest Group, 1995.

[9] Solari, E. and Willse, G., *PCI Hardware and Software Architecture and Design*. Editorial Annabooks, 3rd edition, 1996.

[10] Leach, T. and Hackett, B., "Modern Synchronous Design", *ASIC & EDA*, January, 1993, pp. 44–52.

[11] Stephen, W., "Design Tips for High-Performance FPGA Design", *ASIC & EDA*, October, 1994, pp. 41–52.

[12] Adham, S. M. I. *et al.*, Linking Diagnostic Software to Hardware Self-Test in Telecom Systems, *IEEE Communications Magazine*, **37**(6), June, 99, pp. 79–83.

[13] De Micheli, G. and Sami, M., *Hardware/Software Co-Design*, Kluwer Academic Publishers, First edn., 1996.

[14] Abramovici, M., Breuer, M. A. and Friedman, A., "*Digital Systems Testing and Testable Design*", Computer Science Press, 1990.

[15] Mukherjee, N. and Chakraborty, J., Built-In Self-Test: A Complete Test Solution for Telecommunication Systems, *IEEE Communications Magazine*, **37**(6), June, 99, pp. 72–78.

[16] Altera Corporation, Data Book, 1998.

[17] FLEX 10K Embedded programable Logic Family Data Sheet. htpp://www.altera.com/document/ds/dsf10k.pdf.

[18] Staunstrup, J. and Wolf, W., Hardware/Software Co-Design, *Principles and Practice*, Kluwer Academic Press, 1997.

[19] De Micheli, G. and Gupta, R. K., Hardware/Software Codesign, *Proc. of the IEEE*, March, 1997, pp. 349–365.

[20] Bolsens, I. *et al.*, Hardware/Software Codesign of Digital Telecommunication Systems, *Proceedings of the IEEE*, March, 1997, pp. 391–418.

Authors' Biographies

Deni Torres-Roman (dtorres@gdl.cinvestav.mx) received a Ph.D. degree in telecommunication from Technical University Dresden, Germany in 1986. He was professor at the University of Oriente, Cuba. Coauthor of a book about Data Transmission. He received the Telecommunication Research Prize in 1993 from AHCIET Association and was recipient of the 1995 Best Paper Award from AHCIET Review, Spain. Since 1996 he is associate professor at Research and Advanced Studies Center of IPN (Cinvestav–Mexico). His research interests include VLSI, hard- and software designs for telecommunication area. He is a member of the IEEE.

Alejandro Larios (larios@marte.cinv.iteso.mx) received a M.Sc. degree in telecommunication from CINVESTAV-GDL Mexico in 1999. Currently he is an engineer in TDCOM, Mexico. His research interests include VLSI designs.

Manuel Guzman (manuelg@gdl.cinvestav.mx) received the Ph.D. degree in Electrical Engineering from the Royal Institute of Technology, Stockolm, Sweden. His main research interests are in the hard/software codesign for digital circuits and algorithm implementation on VLSI. He is currently an Associate Professor at the Center for Research and Advanced Studies (Cinvestav-GDL) of IPN, Guadalajara, Mexico and Head of the Semiconductor Technology Center. He is a member of the IEEE.