A Dictionary Machine Emulation on a VLSI Computing Tree System

A. E. Harvin, III and J. G. Delgado-Frias

Department of Electrical Engineering
State University of New York
Binghamton, NY 13902-6000

Abstract

In this paper, we propose a dictionary machine emulation using a novel VLSI tree structure that operates on the dictionary using a blocking technique. We show that dictionary machine operations can be performed through the implementation of a number of processing and communication tasks overlapped on a simple structure. By manipulating the key-records bit serially, and storing them in an external memory rather than within the layers of the structure, we show that the size of the dictionary is limited only by the capacity of the external memory. This structure, which consists of multiple units, can be implemented in VLSI onto a single-chip. The key advantage of our structure is that it provides a means of implementing a high speed and low cost dictionary machine with virtually unlimited capacity; thus, eliminating the need for multiple chips should the dictionary expand. We have that an exhaustive search on a 2048 key-record dictionary can be performed in 29.78 μs.

1 Introduction

The dictionary is an important data structure used in applications such as sorting, searching, symbol-table and index-table implementations. It can be seen as a special purpose system capable of storing records and performing update and retrieval operations on these records. Usually, the records are stored in increasing order to effectively facilitate the operations. Any applications requiring data acquisition can take advantage of such a dictionary machine with an efficient input/output. This means that the machine has to be very simple, scalable, and adaptable to be implemented in such systems used in robotics or image recognition [1].

One of the main issues in previous research on dictionary machines has been the selection of an appropriate host topology. The host topology determines how the operations are processed and also affects the hardware and time complexity of each computational element required to ensure correct operation. Several parallel architectures have been proposed for the implementation of dictionary machines in VLSI. Some report on novel interconnection networks, while others use well studied topologies, like hypercubes. On the other hand, some dictionary machines have been implemented on existing parallel architectures [1]. One such architecture is the binary tree. Tree architectures have been extensively studied as a host topology of dictionary machines [2]. An approach realizing a dictionary machine on a VLSI chip with mesh structure has also been proposed [3]. There, the importance and necessity of two different networks for update and query operations were identified, and an abstract design method on various kinds of meshes and hypercubes were presented. However, embedding two different networks on one host topology caused resource contention and interference among different operations. These problems were resolved at the price of considerable hardware and time overhead in each node.

In this paper, we present a design for implementing a dictionary machine on a VLSI system architecture of a tree-based pipelined computing structure. This paper is organized as follows: In Section 2, we present the definition of a dictionary machine and some important design issues. The system architecture is presented in Section 3 along with the unit descriptions. In Section 4, we describe the implementation of operations onto the system architecture, that are required to perform the dictionary algorithms. Finally, we present our experimental results, and draw conclusions in Sections 5 and 6, respectively.

2 Background

A formal definition of a system specifies its behavior without the implementation details. When implementing the dictionary machine, the following definitions are considered.

0-8186-8409-7/98 $10.00 © 1998 IEEE

2.1 Definition of a Dictionary Machine

The dictionary task can loosely be defined as the problem of maintaining a set of data elements each of which is composed of a key-record pair (k,r), where k is the search key and r the associated record or pointer to a record. For simplicity, the record whose associated key is k will be denoted as *record k*. The dictionary supports a set of commands on its entries [4], such as:

SEARCH(k): Finds and retrieves *record k* if currently stored;
INSERT(k,r): Inserts *record k* into the dictionary;
DELETE(k): Deletes *record k* from the dictionary;

The performance of dictionary machines can be measured in terms of the following parameters [4]:
Capacity: The maximum number of records that may be stored in a dictionary machine.
Response time: The elapsed time between initiation and completion of an operation.
Initiation interval: The number of time units that must be put between the initiations of any two subsequent operations.

2.2 Existing VLSI Solutions

As reported in the introduction, many papers deal with the implementation of a dictionary machine on special purpose architectures. Dictionary machines based on tree architectures have a limitation in the VLSI layout irrespective of its planarity. Even though an optimal layout for tree architectures in terms of area efficiency has already been developed [5], it still allows a nominal maximum edge length. For dictionary machines in which the processing elements (PEs) store the records, the size of the tree is expected to be very larger. More importantly, the degree of a binary tree architecture is only three. This degree, which is smaller than that of hexagonal arrays, consequently results in a lower performance. Therefore, using a binary tree architecture as the host topology can be very costly if not implemented properly. Also, the positions of the input and output nodes of a dictionary machine are very important for correct and fast operations, because it affects the instructions initiation interval.

3 Dictionary Machine Architecture

The proposed architecture is called the *Computing Tree System* (CTS) [7]. The CTS consists of six main units: System-Controller Unit(SCU), Leaf input/output Buffer Unit (LBU), Root input/output Buffer Unit (RBU), Counter Unit (CU), Register Unit (RU), and a Processing Tree Unit (PTU) consisting of N elementary processors. A block diagram of the CTS is shown in Figure 1 and each unit is briefly described below.

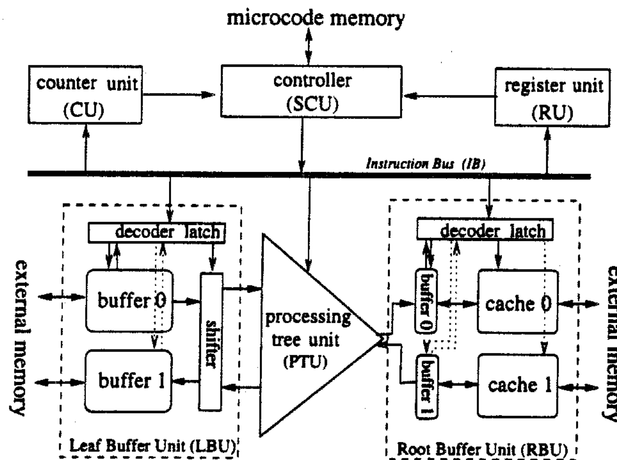

Figure 1: Computing Tree System (CTS)

System-Controller Unit. The SCU has the task of controlling all other units, as well as providing communication to and from an external host. The SCU's design follows that of a pipelined processor. Its internal structure is comprised of five pipelined stages operating in a lockstep fashion. These five stages are: *Instruction Fetch* (IF), *Instruction Decode* (ID), *Execute* (EX), *Memory Access* (MEM), and *Write Back* (WB). Internally there is an arithmetic and logic unit (ALU), a program counter, and several data registers, adders, and staged latches. The SCU provides handshaking with the external host and allows all CTS units to remain synchronous.

Counter and Register Units. The Counter Unit and Register Unit are small in comparison with the others, but quite necessary in maintaining the correct system control flow. The CU is used to count the number of clock cycles a task uses during execution. Data values are provided by the SCU, and are used as initial count values. The counter decrements its count at a rate of once for every. The CU has one output signal that is monitored by the SCU to see if a zero count has been reached. The RU is used to monitor various data values throughout the CTS system. The RU contains a set of shift registers and a data latch that temporarily hold a system state value and a 'compare value', respectively. The RU receives its compare value from the SCU, while the system state value is shifted in from a pre-selected source within the CTS. This source can either be the LBU unit or the PTU unit. A comparison is performed and the result sent back to the SCU.

Leaf Buffer Unit. The LBU acts as a temporary storage unit for input and output data that is processed by the leaf nodes of the PTU unit. It consists of a decoder latch, two buffers, several selection multiplexers, and a bit shifter. Due to an inherently high latency associated with off-chip data transfers, the LBU is specifically designed to minimize this latency by allowing the capability of storing multiple bytes of prefetched data in the two buffers: *buffer0* and *buffer1*. Each buffer is comprised of multiple latches that are staged one below another. These buffers act independently from one another and can simultaneously perform different data transfers. These buffers allow the processors of the PTU to operate at maximum speed. Data is read from the top of the buffers, and when written to, loaded into the first available slot from the top down. After data is read from the top position, all remaining buffer locations shift their data up one location. The LBU is capable of concurrently handling both: double-word bidirectional data transfers to and from the external memory sources, and single-word bi-directional data traffic to and from the PTU. The shifter is used to copy data from one buffer to the other, while intermixing data from another source (PTU unit). The shifter has three modes of operation: *shift-up*, *shift-down*, and *normal*. In determining which shifter locations actually perform a shift, each location initially receives a pre-determined value that is passed in and locked into to its internal control bit. This internal control bit remains locked until the shifter is reset.

Root Buffer Unit. The RBU is quite similar to the LBU with the exception of some additional hardware. Since the RBU is required only to handle single-bit bi-directional data traffic to and from the PTU, smaller bit-sized buffers (*buffer0* and *buffer1*) are used. This smaller buffer design is a result of the PTU's root node outputting single bit data streams. To minimize the number of external data transfers, the RBU utilizes two internal cache memories: *cache0* and *cache1*. Data within these internal caches can be accessed either by row or column. When performing external transfers, a cache's data is accessed by row. When transferring to or from a buffer, a cache's data is accessed by column. These caches each handle word-sized data and collectively, can allow double-word bi-directional data transfers to and from the external memory sources. These caches also allow the RBU to maintain a uniform external data-path to external memory as that of the LBU.

Processing Tree Unit. The PTU structure is primarily comprised of simple processing elements (PEs) [6]. The communication links between the PEs connect them in a fully bi-directional binary tree structure. This tree structure accommodates pipelined computing using the tree's levels as pipeline stages. This structure has I/O channels only at the leaves ($L_0 ... L_N$) and root (R) of the tree. Communication within the inner stages of the tree is hidden from all external units. The total leaf/root communication delay is given by $\log_2 N$, where N is the number of leaves. In a continuous computation, this delay is seen only once. The PTU unit is illustrated in Figure 2. All PEs are identical with the exception of those at the leaf end of the tree. The leaf PEs have additional hardware allowing special operations to be performed without incurring the delay of the tree's structure.

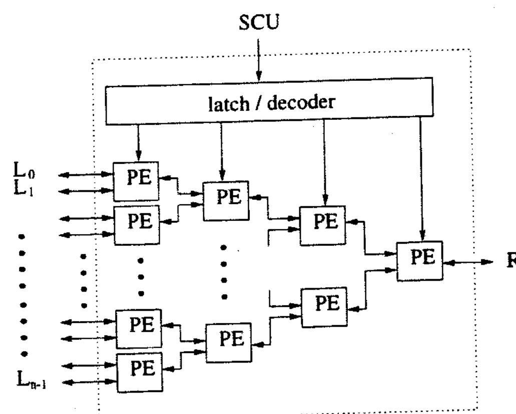

Figure 2: Processing Tree Unit (PTU)

Prior to operation, each PTU stage must be individually initialized. These extra setup steps are necessary due to the PTU's ability to allow each stage to operate in a different mode [6]. During PTU operation, all stages perform their respective operations simultaneously. Rather than storing data in local PE memories, the data moves from stage to stage until all operations are performed.

The PTU has two distinct modes of operation: *computation* and *communication*. In computation mode, the tree performs as a synchronous SIMD parallel computer under the supervision of a single control unit. When in communication mode, there are several schemes in which a stage can perform. However, only the *broadcast* and *root-leaf* schemes are required in emulating the dictionary machine [5]. The necessity for the alternative communication schemes is to allow different communication possibilities among the leaves and the root of the tree structure. When a stage operates in the *broadcast* scheme, the PE's root input is simultaneously broadcasted to both of its leaf out-

puts. This provides an equal propagation delay from the root to each leaf. This communication scheme can be individually implemented at different stages to allow the desired communication pathway through the tree structure. When the *leaf-root* scheme is used, the leaf PEs perform a subtraction using their leaf inputs and root input, and send their results back towards the leaf outputs. Only the leaf PEs can perform in this scheme. A 16-leaf binary tree structure has been successfully built using MOSIS $2\mu m$ CMOS technology [6].

4 Dictionary Machine Emulation

In implementing the dictionary algorithm onto the CTS system architecture, key-record pairs are stored in a memory array connected to the CTS. This memory array of key-record pairs (KRs), known as the *dictionary*, is stored from top to bottom in increasing order. In mapping the dictionary algorithm, the concept of a low-cost, high clock rate, and bit-serial implementation was the key criteria. With this in mind, it was not cost effective to have the entire dictionary accessible every clock cycle. Therefore the dictionary is divided into groups of KR pairs. These groups, called *dictionary blocks*, contain as many KR pairs as there are PTU leaf inputs. The dictionary blocks are presented to the CTS through the LBU. When a record is to be inserted or deleted from the dictionary, it enters the CTS through the RBU. This record is known as the Insert/Delete Record (IDR). When the dictionary receives an IDR to be inserted, the IDR is inserted at the proper place in the memory array, and all KRs with a key value greater than that of the IDR are shifted one position down. When a record is to be deleted, the IDR must match one of the dictionary KRs. Once the IDR is broadcasted through the PTU, the appropriated KR is removed and all KRs with keys greater than that of the IDR's shift one position up. These CTS sequences are described below.

4.1 Dictionary Operations

The basic operations that the CTS uses to perform the dictionary machine algorithms are: *block search*, *record search*, *record insert*, and *record delete*. These operations are defined below in terms of the actions taken be the machine.

Block Search(k): This operation is used to determine if the current dictionary block contains a specific KR. Dictionary blocks are searched in sequential order such that the first searched block where the IDR's key value is less than that of the last KR in the block is determined to be the correct block. This is determined by subtracting the IDR's key value from all KR's within the current block, and using the result of the subtraction procedure to determine the outcome of the block search operation. This outcome is determined by:

- If $(k_{B_l} - k_{IDR}) < 0$, where k_{B_l} represents the key value of the last KR in the current block, and k_{IDR} represents the IDR's key value, then return 0; incorrect block
- If $0 \leq (k_{B_l} - k_{IDR})$, then return 1; correct block

The hardware implementation of a *Block Search(k)* operation on the CTS is as follows. The lowest available dictionary block is loaded into a buffer of the LBU, while the IDR key value is loaded into a cache of the RBU. The IDR key is transfered from the cache into the adjoining buffer in the RBU. The PTU is setup such that all stages, with the exception of the leaf-stage, will operate in broadcast mode. The leaf stage PEs, containing special hardware, are setup to perform in 'root-leaf' mode. The LBU empties the dictionary block from the buffer, through the shifter, and into the leaf inputs of the PTU. The PTU performs the subtraction operation and sends the results back towards the LBU. This is shown in Figure 3a. During this process, the dictionary block is not sent to the PTU leaf inputs until the IDR's key value has had time to propagate from the PTU's root input all the way to the PTU's leaf stage.

As the results of the 'root-leaf' mode pass back to the LBU, they are not read into a buffer, but rather, the MSB of each result is locked into the LBU's shifter at the respective shifter location. This sets up the shifter for any subsequent insert or delete operations to follow. The MSB of the bottom shifter location is then sent to the RU for evaluation. This is shown in Figure 3b.

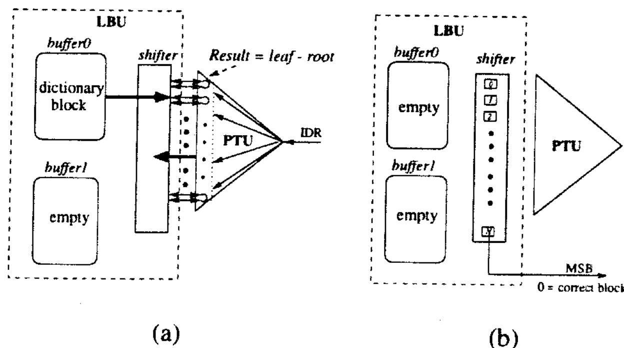

Figure 3: Block Search Operation

The RU matches this shifted-in value against a zero (which has been pre-locked in during initialization)

and determines if a valid match was found. The SCU monitors the RU's output and pending the result, takes the appropriate steps. If the this MSB is a logic 'zero', then the LBU currently has the correct dictionary block. Otherwise, the record does not belong (or exist) in this particular dictionary block.

Record Search(k) - This operation is used to determine if the current block contains the IDR record. In performing this operation, the results of the *Block Search* operation are used. If the IDR does exist within the current block, one of the results acquired during the *block search*'s subtraction operation will contain zeros. This will indicate that the IDR's key value matched one of KR key values in the current block. Following the same implementation as that used in the *Block Search* operation, the results are stored in an LBU buffer. The LBU sends the results through the PTU, which has been initialized to perform a 'tree-min' operation [6] on the data entering its leaf inputs. As the minimum result is outputted from the root stage of the PTU, it is shifted into the RU and matched against a zero (which has been pre-locked during initialization). The RU outputs a one if a valid match was found. The SCU then takes the appropriate steps.

Record Insert(k,r) - This operation updates the current dictionary block to include the IDR record. To perform the *Record Insert* operation, the MSBs of the results obtained during the *Block Search* operation are used. If there exists a logic-one/logic-zero contrast between two consecutive result MSBs, then the IDR is inserted between the two associated KRs. This is performed by re-broadcasting the IDR to the LBU's shifter and inserting it into the proper location as the entire dictionary block is copied from one leaf buffer to the other. As the IDR key is presented to the KRs of the current dictionary block, the dictionary block is shifted as follows:

- If $k_i < k_{IDR}$, then do nothing
- If $k_{i-1} < k_{IDR} < k_i$, then *Insert*; shift $(k_{i+1}, r_{i+1}) \leftarrow (k_i, r_i)$, $(k_i, r_i) \leftarrow (k_{IDR}, r_{IDR})$
- If $k_{IDR} < k_i$, then shift $(k_{i+1}, r_{i+1}) \leftarrow (k_i, r_i)$
- If $k_i = k_{IDR}$, then abort *Insert* operation

These shift operations are only performed on certain KRs. The entire PTU is initialized to perform the broadcast scheme, and the IDR is sent from the RBU, through the PTU, and into the LBU shifter. The dictionary block (buffer) is sent to the shifter at the same time the PTU's leaf nodes output the IDR to the shifter. The shifter is initialized to assess its locked-bits and perform a shift-down function at all locations where the lock value is equal to a logic zero. The shift-down function is performed on all data entering the shifter from the LBU buffer, and the results are outputted into the remaining LBU buffer. This output is the 'newly' shifted dictionary block. Since all KRs will not shift, there will be a location in the shifter where the MSBs of two consecutive locations are not the same. This is where the IDR is inserted. Only this shifter location will read the input from the PTU rather than the LBU's buffer side. With respect to the input line in which each KR pair was read into the shifter, that shifted KR will appear on the next consecutive output line. During this shift down operation, a KR will effectively shift out of the bottom of the dictionary block. The RBU is initialized to receive this KR and temporarily stores it in an RBU buffer location. This KR will be used for insertion into the top of the next sequential dictionary block. Zeros are then locked into the shifter control bits, indicating that the shifter will perform simple data shifts on all subsequent dictionary blocks following the dictionary block in which the IDR was inserted.

Record Delete(k) - This operation updates the current dictionary block by removing the KR that matches the IDR record. The initialization process is similar to that of the *Record Insert* operation, where the MSBs of the results performed during a *Block Search* operation are used to determine which shifter location is to remove the record. The difference is during operation. If there is a logic-one/logic-zero contrast between two consecutive results, then the KR associated with the logic-zero is removed from the dictionary. During this process, all KRs in the dictionary with a key value greater than that of the IDR must shift up one. All records with a key-record value less than that of the IDR remain in their present location.

5 System Performance

In implementing the dictionary algorithms defined in Section 2, the operations described in the above section are used in a repetitive manner such that the algorithm achieved. To demonstrate the full capabilities and performance of the CTS performing the dictionary machine algorithms, a full simulation of the system was performed. In this simulation, a dictionary consisting of 256 key-records was used. External memory banks were used to store the entire dictionary

as well as the suggested IDRs. Although each dictionary algorithm varied in length, the *Delete(k)* and *Insert(k,r)* algorithms required the instruction code of the *Search(k)* algorithm to be included within their code [7]. The 256 KR dictionary was separated into 16 blocks, where each block consisted of 16 key-record pairs. A total of 8 key-records were inserted and deleted. Each *operation* was evaluated and from our simulation we have obtained the results shown in Table 1. To achieve these results the key length and record length were set to match the buffer size of 8.

Table 1: CTS's Dictionary Operation Performance

Capacity	*limited by external memory*
Pipeline Interval	8 clock cycles
Response time:	
- $T_{Block\,Search}$	20 clock cycles
- $T_{Record\,Search}$	28 clock cycles
-$T_{Record\,Insert}$	35 clock cycles
- $T_{Record\,Delete}$	35 clock cycles

Using the above response times for the individual operations, the maximum and minimum time required to perform each of the three dictionary algorithms could be calculated [7]. Applying calculations to different dictionary sizes yields varying results. These results are shown in Table 2.

Table 2: CTS's Dictionary Algorithm Performance in μs (@ 100 MHz clock)

Algorithm		Dictionary Size			
		256	512	1024	2048
Search(k)	*min*	0.48	0.51	0.54	0.57
	max	3.48	7.02	14.40	29.78
Delete(k)	*min*	3.83	7.39	14.81	30.19
	max	6.08	11.71	25.50	53.03
Insert(k,r)	*min*	3.83	7.39	14.81	30.19
	max	6.08	11.71	25.50	53.03

6 Concluding Remarks

In this paper we have presented a novel VLSI pipelined tree system capable of performing several dictionary machine algorithms. The Computing Tree System is able to store the data structure and maintain it. The main features of the dictionary machine implementation proposed in this paper are its adaptability and performance capacity. We have shown that the algorithms can be performed through the implementation of a number of processing and communication tasks overlapped on a simple structure. Using the CTS structure to perform the dictionary algorithms provides several important advantages including high-performance, a large dictionary size (limited only by external memory), and low cost implementation. It is shown that by manipulating the key-records bit serially and storing them in an external memory, the size of the dictionary is limited only by the capacity of the external memory. Since key-records are not stored within the stages of the tree as in other implementations, the PEs are kept quite simple; thus, providing for a low-cost implementation. Using modern CMOS technology, embedding the entire system onto a single chip is quite possible.

Although the first prototype of this system has yet to be built, the PTU component has been successfully designed and fabricated in VLSI [6]. The intent is to develop an entire CTS system prototype as a single-chip system. This single-chip system, coupled with high speed I/O devices can provide for real time information processing with a constant pipeline interval.

References

[1] T. Duboux *et al*, "A Scalable Design for Dictionary Machines," *Algorithms and Parallel VLSI Architectures III.*, pp. 143-154, Elsevier Science B.V., 1995.

[2] T.S. Narayanan, "A Class of Semi-X Tree-Based Dictionary Machines" *The Computer Journal.*, Vol. 39, No. 1, 1996.

[3] F. Dehne and N. Santoro, "An Improved New Embedding for VLSI Dictionary Machines on Meshes," *Proc. Int'l Symp. Computer Applications in Design, Simulation, and Analysis*, pp. 113-116, 1989.

[4] H. Y. Youn and J. Y. Lee, "An Efficient Dictionary Machine Using Hexagonal Processor Arrays," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7, No. 3, Mar. 1996.

[5] H. Y. Youn and J. Y. Lee, "On Implementing Large Binary Tree Architectures in VLSI and WSI," *IEEE Transactions on Computers*, Vol.38, No. 4, Apr. 1989.

[6] A. E. Harvin, J. G. Delgado-Frias, "A VLSI Highly Pipelined Computing/Communicating Tree Structure," *Journal of Microelectronic Systems Integration*, Vol. 4, No. 4, pp. 233-246, December 1996.

[7] A. E. Harvin, *A Pipelined VLSI Tree System with I/O Buffers for Data Parallel Computing*, Dissertation, State University of New York, Binghamton, New York, December 1997.