Portland State University

PDXScholar

Dissertations and Theses Dissertations and Theses

7-26-1991

Associative Processing Implemented with Content-Addressable Memories

Luis Sergio Kida
Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Part of the Electrical and Computer Engineering Commons

Let us know how access to this document benefits you.

Recommended Citation

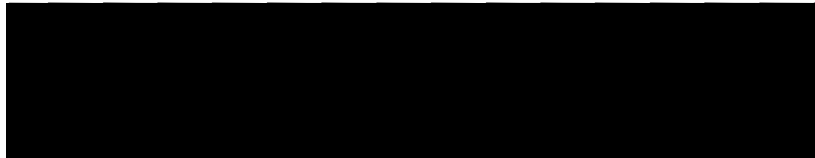
Kida, Luis Sergio, "Associative Processing Implemented with Content-Addressable Memories" (1991). *Dissertations and Theses.* Paper 4176.
https://doi.org/10.15760/etd.6060

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

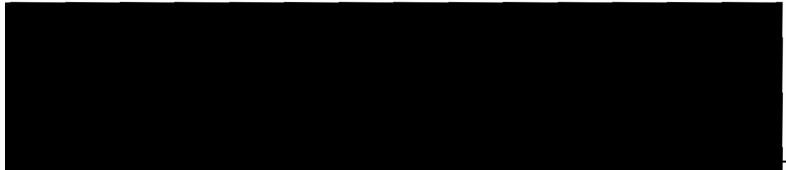
AN ABSTRACT OF THE THESIS OF Luis Sergio Kida for the Master of Science in Electrical and Computer Engineering presented July 26, 1991.

Title: Associative Processing Implemented with Content-Addressable Memories.

APPROVED BY THE MEMBERS OF THE THESIS COMMITTEE:

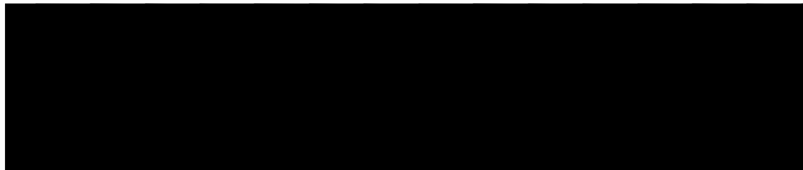


W. Robert Daasch, Chair



Marek A. Perkowski

Michael A. Driscoll



Richard G. Hamlet

The associative processing model provides an alternative solution to the von Neumann bottleneck. The memory of an associative computer takes some of the responsibility for processing. Only intermediate results are exchanged between memory and processor. This greatly reduces the amount of communication between them. Content-addressable memories are one implementation of memory for this computational model. Associative computers implemented with CAMs have

reported performance improvements of three orders of magnitude, which is equivalent to the performance of the same application running in a conventional computer with clock frequencies of the order of GHz. Among the benefits of content-addressable memories to the computer system are: 1) it is simpler to parallelize algorithms and implement concurrency; 2) the synchronization cost for parallel processing is lower, which enables the use of small grain parallelism; 3) it can improve the performance in non-numeric applications that are known to have low performance in conventional computers; 4) it provides a trade off between integration density and clock frequencies to achieve the same performance that is not available in RAM. 5) matches well to current and future technologies due to the trade off between integration and clock frequency; 6) it attacks the von Neumann bottleneck by reducing the requirements on the communication bandwidth between processor and memory.

In this thesis, the role of CAMs in associative processing is analyzed, reaching the conclusion that to implement these characteristics the CAM must be able to filter the data transferred to the processor, provide explicit support for parallelism and data structures, support non-numeric applications, and execute logical operations. The characteristics and architecture of a content-addressable memory integrated circuit are presented along with an application with estimated performance improvement of over three orders of magnitude .

ASSOCIATIVE PROCESSING IMPLEMENTED WITH CONTENT-ADDRESSABLE MEMORIES

by

LUIS SERGIO KIDA

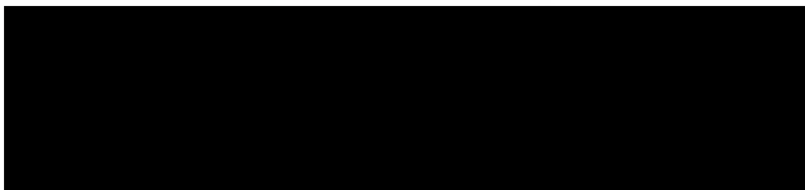
A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
ELECTRICAL AND COMPUTER ENGINEERING

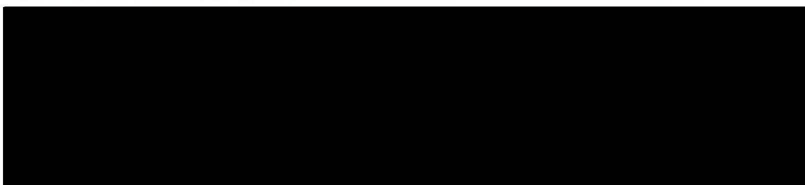
Portland State University
1991

TO THE OFFICE OF GRADUATE STUDIES:

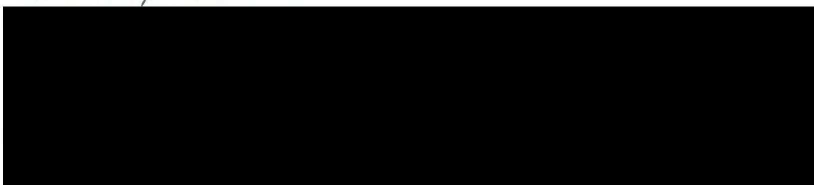
The members of the Committee approve the thesis of Luis Sergio Kida presented July 26, 1991.



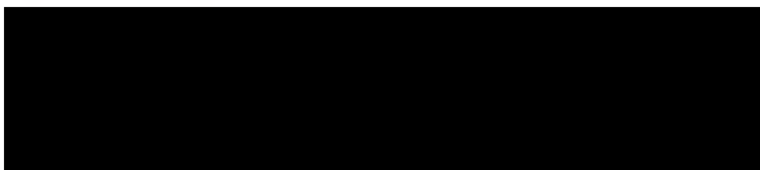
W. Robert Daasch, Chair



Marek A. Perkowski

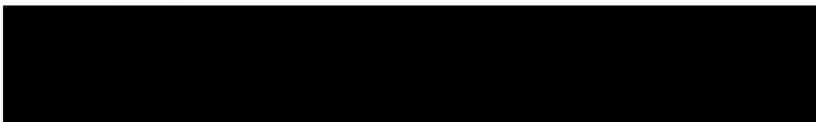


Michael A. Driscoll

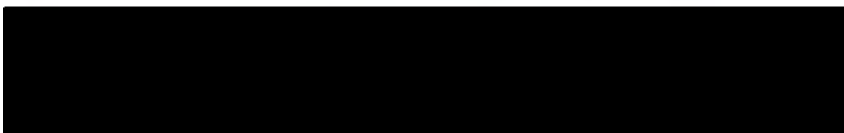


Richard G. Hamlet

APPROVED:



Rolf Schaumann, Chair, Department of Electrical Engineering



C. William Savery, Vice Provost for Graduate Studies and Research

To my parents

TABLE OF CONTENTS

LIST OF TABLES iv

LIST OF FIGURES v

CHAPTER PAGE

I INTRODUCTION 1

Motivation 1

The von Neumann Model of Computation and Parallelism 3

An Alternative Computer Memory Model 6

II CONTENT-ADDRESSABLE MEMORY (CAM) 11

CAM Building Blocks 13

General Issues in the Integration of CAMs 24

III BASIC COMPUTATIONAL TASKS AND CAMS 32

Data Storage 32

IV CUBE CALCULUS 39

Basic Concepts 39

Operations with Cubes 43

Using CAMs for Cube Calculus 49

Logic Minimization of Synthesis of Boolean Functions 52

Cube Calculus and Resolution 64

Image Processing and cube Calculus 67

V CONCLUSIONS AND FUTURE WORK 69

REFERENCES 72

LIST OF TABLES

TABLE		PAGE
I	Execution Time of the Sharp Product on a RAM-Based Computer	57
II	Execution Profile of $F_1 \# F_2$	58
III	Execution Profile of $F_2 \# F_1$	58
IV	Execution Profile of $F_1 \# F_3$	59
V	Cummulative Cost of Sorting Cubes	59

LIST OF FIGURES

FIGURE		PAGE
1	The von Neumann computer architecture.	2
2	Block diagram of a CAM.	12
3	Static CAM cell.	14
4	Functional memory cell.	16
5	Content addressable ROM (CAROM)	23
6	CAM bank built with minimum additional logic	27
7	CAM bank with hierarchy	28
8	Set data structure implemented with CAMs	36
9	Examples of Karnaugh maps and hyperspace	43
10	Examples of cube union	45
11	Inversion of functions	46
12	Sharp operation	47
13	Two examples of cube consensus	48
14	Determining the covering relation among cubes with CAMs	50
15	Testing if cubes overlap using CAMs	50
16	Memory before and after intersection	51
17	State of the memory before processing	53
18	Patterns used for the generation of resultant cubes	53
19	Memory after the sharp	54
20	Results after the removal of empty cubes	54
21	Karnaugh map of F and g used in the example of sharp	54
22	Sharp of two cubes of a multiple output function	56

23	Algorithm to sharp a cube out of a multiple output function	57
24	Estimate of the execution time of sharp in a CAM	60
25	Example of resolution and unification	65
26	Graphical example of the resolution principle	66
27	Sample image and CAM entries	67
28	Update image and CAM entries	68
29	Final image and CAM entries	68

CHAPTER I

INTRODUCTION

Content-addressable memories have been known for over 30 years and there is an extensive list of proposed applications for them in the literature. Nevertheless, content-addressable memories are devices almost unknown to the majority of the engineers in the electronics industry. This master's thesis proposes to organize the existing information on content-addressable memories and show that the lack of systems using content-addressable memories is not due to an inherent fault in the content-addressable memory model.

To achieve this goal, I will outline an application niche that is not well served by current computer systems and present the characteristics of the associative processing model that can execute these applications effectively. The content-addressable memory is one implementation of this model. The characteristics of content-addressable memories and the circuits required to implement content-addressable memories are also described. I will also investigate the data structures that are improved by the addition of associativity and identify the applications that benefit from associativity.

MOTIVATION

The demand for more processing power and more memory continues to increase. In the last 25 years, advances in computer performance have been tied to advances in microelectronics through faster technologies and the integration of system's bottlenecks. Since the introduction of the *Dynamic Random Access Memories (DRAM)* 20 years ago, the density of integrated DRAMs has quadrupled every 3 years [ITOH90] and the number of devices in commercially available integrated circuits has doubled every year [SEITZ84]. During the same period, the computer architecture has remained close to the *von Neumann paradigm* of a single sequential processor, a storage device and an input/output channel. Figure 1 shows the block diagram of the von Neumann computer architecture.

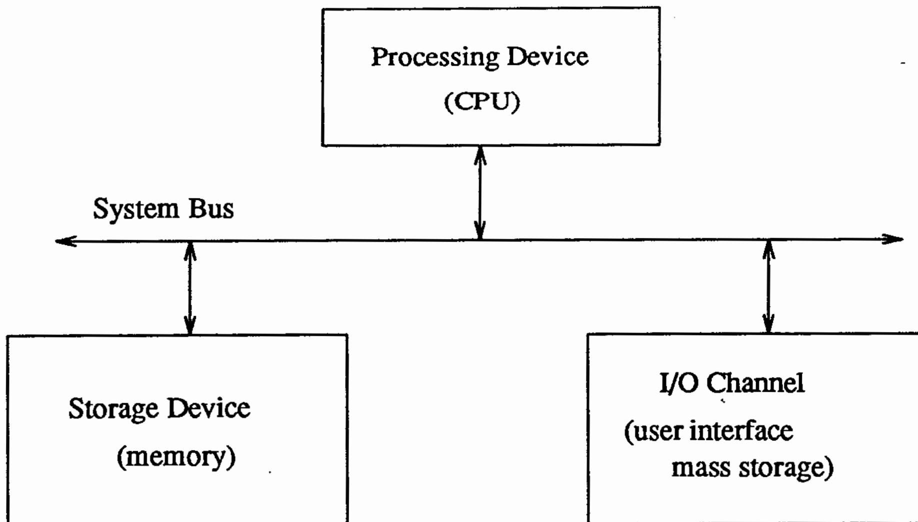


Figure 1. The von Neumann computer architecture.

The simplicity and elegance of the von Neumann paradigm allied with the fast development of microelectronics are responsible for the past developments in computation. To obtain further performance increase and take advantage of new technologies, modifications have been made to the von Neumann architecture. These modifications include higher integration and the increased use of parallelism. Three key reasons for changing the von Neumann computer architecture style have been identified [MOTO-OKA83, SEITZ84]:

(1) Computer system architectures have to match current and future technologies. Device speeds are approaching fundamental technological limits. And much before that, the circuit dimensions and speed will approach the limits where Kirchoff's laws no longer apply. The von Neumann single processor cannot continue indefinitely to increase in complexity and performance. VLSI substantially reduces hardware costs suggesting the use of parallel processing architectures.

(2) There are more problems to be solved than computer scientists to solve them. Programming and computer problem solving have to be simpler in order to enable more people to effectively use computers. One way to improve productivity is through more sophisticated human interfaces, natural programming languages, or more unconventional approaches. The von Neumann architecture was developed to answer computer implementation needs, not human needs.

(3) A general purpose computer system has to provide acceptable performance in a large class of applications. The structure and the processing principles of general-purpose computers used today mainly take into account the demands of numerical algorithms [ZEIDLER89]. Current computers have poor performance in applications for processing speech, text, graphs, images, non-numerical data and for artificial intelligence. These applications are important for the implementation of a better human interface.

This thesis reports my investigation into computer architectures that use content-addressable memories as an alternative to improve performance.

THE VON NEUMANN MODEL OF COMPUTATION AND PARALLELISM

The principle of multi-processing is to have more than one processor cooperating to execute the same job. Systems with more than one processing device depart from the well accepted von Neumann architecture in a manner that change computation in fundamental ways:

(1) Many processors cooperating in the same task have an attendant requirement for synchronization and communication between the processors non-existent in sequential processors. That requirement results in often substantial overhead for multi-processor systems, either in time (performance) or in hardware.

(2) Multi-processor systems also require the development of parallel algorithms to deliver their potential performance. However, the techniques to develop parallel algorithms are not as developed as those for sequential algorithms.

(3) To make full use of the processors, the parallel programmer or the compiler has to find enough parallelism in the problem to keep all processors busy. The task of finding this partitioning can be expensive.

It is also known that the gain in performance with multi-processing has its limits. For a given parallel algorithm there is an optimal number of processors to solve a problem and therefore, increasing the number of processors beyond that number does not lead to linear increase in performance. Partitioning the job into many pieces can even degrade the performance because of the overhead in syn-

chronization. The simple application of a higher degree of multi-processing alone is not the final solution to the quest for improved computer performance. However, parallel processing continues to improve its position because there are technological advantages in multi-processing. The lower costs of the individual processor and *Very Large Scale Integration (VLSI)* makes multi-processor systems a cost effective solution. The proposed memory model has to consider parallel processing. Many existing multi-processor systems use multiple von Neumann processors. Because of the sequential nature of the von Neumann processors, this style of parallel computation can have a large overhead to implement synchronization and communication. To limit the overhead, von Neumann style multi-processor systems have the tendency to sub-divide the job into few large pieces in what is called *coarse grained parallelism*, as opposed to *fine grained parallelism* that sub-divides the job into many smaller tasks enhancing the potential concurrency of processing.

Another main component of the von Neumann computer model that should be considered is the storage device. To understand how the change of memory model affects the computer system, the von Neumann memory model and the effect of multi-processing on memory organization are reviewed.

The single storage device in the von Neumann architecture can be viewed as a black box that takes an address and uses it to select a storage location. This storage location is used to store data via the write operation or to retrieve data previously stored in that location via the read operation. Although all the storage devices studied in this document can access the pieces of memory stored in any order, historically this device is called *random access memory (RAM)* as opposed to magnetic tape mass storage that accessed data in a fixed order. RAMs store data associated with an artificial code, the address. The address refers to the physical location, not with the datum it stores. A physical location is selected based on the address associated to it.

In the von Neumann model, data and instructions are stored in RAM and have to be transferred to and from memory and processor. The characteristic of sequential access to memory, one datum at a time, limits the system performance and is known as the *von Neumann bottleneck*. For maximum efficiency, the memory has to feed data at the same rate the processor consumes it. This sequential nature of RAM creates serious disadvantages when multiple processing units are introduced.

There are two main parameters to evaluate the storage device implemented with RAM in the von Neumann architecture. Those parameters are the maximum rate that data can be accessed and the maximum amount of data that the memory device can store. Larger memories are needed to solve the increasingly larger problems found in current applications. It takes longer to process the massive amount of information stored in this larger memory. To reduce the execution time, the processor and the memory must run at a higher speed.

Multiple processors in the system provide the processing power to execute these larger problems but also stress the fact that the path between the memory and the processor is the key limiting factor for system performance. For example, one scheme of connecting multi-processors and memory is *shared memory* where each memory word has a global address. Multi-processor systems have higher peak demand and potentially consume data at higher average rates and because there is only a single data bus to access memory, each processor has to "wait for its turn" to access memory. A second scheme is *message passing* or *local memory.* Local memories try to widen the system memory bandwidth by giving some of the memory to each processor with the scope of the address of each word localized. To share data, messages are passed through a communication network between the processor requesting the information and the processor that has the data stored in its local memory. Distributed and shared memories are like two ends of a possible continuum of architectures for storage using the same memory model. One notable enhancement of memory systems is the introduction of hierarchical memory levels like paging store and cache. The effectiveness of cache memories relies on the prediction of which data will be accessed in the near future. The need for a good prediction of the next access to memory explains in part the loss of performance in applications that dynamically disorder data in memory. Caches can be used with either shared or local memories.

The traffic between processor and memory is not used solely to exchange data. Communication of the address also consumes some of the communication bandwidth between memory and processor. Addressing by location is particularly inefficient when:

(1) data is associated with several sets of reference properties (e.g. address pointers).

(2) the size of data elements is small when compared to the reference properties that have to be stored with them.

(3) during processing data becomes dynamically disordered in memory.

There is a large overlap between the applications for which von Neumann architectures are considered inefficient and those where addressing by location creates a large overhead. That is one of the main reasons this research is focused on a different computer memory model.

Regardless of these modifications, three observations can be made:

(1) The modifications to the memory architecture have not changed the concept of memory significantly. A better memory still means a larger and faster one.

(2) Parallel processing is becoming increasingly popular. RAM was developed to fit the von Neumann architecture and does not couple well with parallel processing demands.

(3) Many applications for which von Neumann type architectures have poor performance are also applications in which address calculation overhead can be substantial.

AN ALTERNATIVE COMPUTER MEMORY MODEL

A different memory model that comes to mind is the biological model. Human memory is known for its ability to process non-numerical data, image, speech and for (natural) intelligence. Computer memories that model the human memory may capture some of these qualities.

In Aristotle's observations on the human memory [SORABJI72], he makes a distinction between the simple storage of information and storage and processing of information. Aristotle called each action remembering and recollecting, respectively. *Remembering,* according to him, retrieves data exactly as it was presented. And *recollection* returns the data massaged by reasoning.

Aristotle observed that recollections seem to be a *synthesis* of memorized information. The recollection process is a sort of reasoning that would remember an image not necessarily identical to the original occurrence. Human recollections do not have to be exactly what was originally presented; they can be modified by the interaction with other knowledge. The human memory functions are quite different from the von Neumann computer memory. There is no clear distinction between processing and storage in the human memory suggesting that computer memory structures should perform part of the processing.

The following features were selected from human memory as parameters to evaluate memory models based on their apparent importance in the human thought process and the potential improvement they could bring if introduced to computer systems [KOHONEN80].

(1) *Direct association or auto-association:* recollect a data structure from a fragment of the datum large enough to enable recognition.

(2) *Indirect association or hetero-association or association by inference:* recollect data from pieces of data that are not similar to, nor part of, the data to be recalled. Uses reasoning, a sequence of many direct associations to make associations by meaning.

(3) *Sequential recollections, or temporal association, or temporal recall:* the memory also stores the sequence in which data should be recalled. There is a sense of time and/or order.

(4) *Robustness:* Recollections using a key contaminated by noise will recall data that is most similar by some measure to what should have been recalled by the perfect key in an optimal way.

(5) *Graceful degradation and fault-tolerance:* damage to memory cells degrades the results gracefully instead of impairing the whole process.

Associativity is possibly the most desirable characteristic of human memory. The term associative memory is used to refer to a memory that is capable of some kind of data access through association. Kohonen [KOHONEN80] named the type of processing that uses associative memories *associative processing*. In associative processing, address calculations are eliminated, eliminating one intermediate stage between human conception and computer implementation and reducing the traffic between the processing device and the memory device. The computational work load is shared by the memory and by the processor. Non-numerical applications can also use the support of the associative memory to execute basic logic operations. The next sections outline two extremes in the range of implementations of associative memories and discuss the implementation issues of associative memories and how well they answer to the basic requirements of:

(1) Improving the performance of basic functions for processing speech, text, graphs, images and other non-numerical data, artificial intelligence type processing such as inference, association and learning.

(2) Simpler programming.

(3) Matching the implementation to current and future technologies.

Associative Memory Emulation Using Random Access Memories

Unlike RAM that stores data by address, associative memories store data based on the data contents and on associations to other data stored in memory. Associative memories implemented with RAM have to emulate associations using addresses. This requires an overhead of memory usage and processing and further stresses the constraint imposed by the path between memory and processor, the von Neumann bottleneck. Furthermore, the desired characteristics of associative memories are implemented through software, through schemes such as hash coding and indirect association with the inference process and artificial intelligence programs. Although RAM can emulate associative memories with the use of more storage and the participation of the processing device, associative memory emulation is one application where RAMs are inefficient.

RAM store data in a single physical location. The data stored in RAM is completely lost in case of malfunction or damage to the location that stores the data unless error correction, fault tolerance and graceful degradation are explicitly provided through proper storage with redundancy and error correcting codes. Error detecting codes require the addition of at least n bits to the data word to detect any error of length $n-1$ bits or less [HAMMOND86]. Error correcting codes require that an even larger portion of the memory be reserved for redundancy. Storage of information with fault tolerance in RAM demands more memory and processing than only storing data.

There is an additional cost in processing and memory associated to the implementation of each of the features of associative memories analyzed. Although the equivalent of the von Neumann processing device could emulate these features in software, this implementation most likely would not enhance the performance of the computer system.

Associative Memory Implemented with Artificial Neural Networks (ANN)

The *artificial neural network (ANN)*, is a massively parallel array of highly interconnected simple computational units, called the artificial *neurons*. The interconnections between neurons are called

synapses. Synapses are the way neurons communicate and in this manner cooperate to perform collective computations [MURRAY89].

The behavior of an ANN is determined by its synapses. A neural network has to be *trained* to perform a desired behavior like direct association (pattern recognition). The *training* of a neural network consists of adjusting the interconnection weights until the neural network produces the desired outputs. ANN "programming" is conceptually simple and uniform.

Associative memories implemented with artificial neural networks are the ones that best captures the features of human memory. Associative memories implemented with ANN have natural fault tolerance because the information is stored in a distributed manner in the neural network. The contribution of any single element is small and therefore the failure of one element has a small impact on the storage of any individual datum.

Memories implemented with ANNs have a very small ratio of storage capacity to hardware used to implement it. An ANN associative memory interconnected as the well known Hoppfield net, has a storage capacity of patterns proportional to N/logN where N is the number of neurons. Error correcting capabilities are added with even larger use of neurons. For the ANN to be able to correct up to n bits, that is to say that each stable state of the ANN has a radius of attraction 'n' [NIJHUIS89], the storage capacity has to be corrected by the factor $0.5(1-2n/N)$. If the patterns to be stored are not favorable for error correction, that is the differences in the patterns are small, the storage capacity of the ANN is even smaller [NIJHUIS89].

ANNs are robust, degrade gracefully and can be trained to associate. But, to achieve these remarkable characteristics, ANNs depend on massive use of hardware. Large numbers of neurons and synapses are necessary to execute relevant work. The major problem of ANN computer memories is its implementation. A neuron is an element more complex to implement than a simple storage cell and the increase in storage capacity is less than linear with the increase in the number of neurons. So, ANN computer memory implementations would consume significantly more silicon real estate than traditional implementations. Current integrated ANN ICs have on the order of hundreds of neurons. For more information on ANN implementations please refer to [HOLLIS90, HOWARD87, SAGE86, BORGSTROM90, RUECKERT87, VITTOZ89, WEGMANN90, HAMMERSTROM90, GRAF87,

MURRAY89, BRUCE88]

Summary

A limit to the performance of present computer architectures is the communication between the processing device and the memory device. Also, the performance of present computers in non-numeric applications is unsatisfactory. Computation with associative memories is one alternative to improving the performance of computers and, specifically, to improve the performance on non-numeric applications.

The RAM and the ANN are two extremes of memory device implementation. RAM is capable of storing large amounts of information in relatively small area. But, data manipulation is inefficient because it can only associate information to the location where it is stored. ANN, on the other hand, is extremely powerful. But the hardware cost is prohibitive. Furthermore, the computational model of ANNs is radically different from the present one and is incompatible with current algorithms and programs.

Aristotle had already observed that the people who are slow are better at remembering, while those who are quick and learn well are better at recollecting [SORABJI72]. The same occurs with artificial memory. The comparison between RAM and ANN shows that for the same amount of hardware, it is necessary to sacrifice storage capacity for convenience of handling.

In the next chapter the *content-addressable memory (CAM)* is presented. The content-addressable memory balances high level functions and storage densities to achieve high performance by distributing logic circuits within the storage devices to incorporate features of associative memories at the circuit level.

CHAPTER II

CONTENT-ADDRESSABLE MEMORY (CAM)

The content-addressable memory is the proposed device to implement associative memory and to improve the performance in non-numeric applications. This chapter describes the basic features of the CAM at the behavioral and functional level and the expected characteristics of an integrated content-addressable memory circuit.

There are many memory devices with very different characteristics in the literature under the name content-addressable memory. For the purpose of this discussion a CAM is a memory device that uses a technology similar to the one used in RAMs to store information but, contrary to RAM, the CAM selects the physical location based on the data contents. While the RAM requires the address of the location in which the information is stored, the processor has only to describe the data it wants to access and the CAM selects stored data matching the description. Valid descriptions of memory words usually include combinations of the following properties: matching a binary pattern, being smaller or larger than a value, being in a range of values, being the largest value stored, and being the smallest value stored within a CAM. These comparisons are performed in parallel with the time required to execute the operation essentially independent from the number of words stored in memory. After the set of matching words is determined, another important function of the CAM takes place. Generally, each word, or element, of this set has to be accessed sequentially. The CAM sub-divides the set of matching words into many single element sets that can be sequentially accessed.

Some CAMs are also capable of limited data manipulation on the set of matching words such as bitwise inversion, logic operations such as OR and AND, and arithmetic operations such as the addition or subtraction of constants. The following section presents how each of these characteristics are translated into building blocks of a CAM architecture. The architecture will be described in a bottom-up approach.

The CAM architectures discussed in this work implement equality comparisons at hardware level and implement more complex comparisons as a sequence of equality comparisons, usually controlled internally by the CAM. CAM designs such as [RAMAMOORTHY78] and [LEE85] which implement more complex comparisons in hardware by implementing other types of logic, such as magnitude comparators and arithmetic functions, will not be discussed. The major building blocks of a CAM architecture are the data, search key and mask registers, array of CAM cells (CAM storage), registers for the search responses (HREGs), processors to operate on the search responses (HPE), and an arbiter to decide which matching word gains access to the data bus (MRR). They are shown without their control in Figure 2.

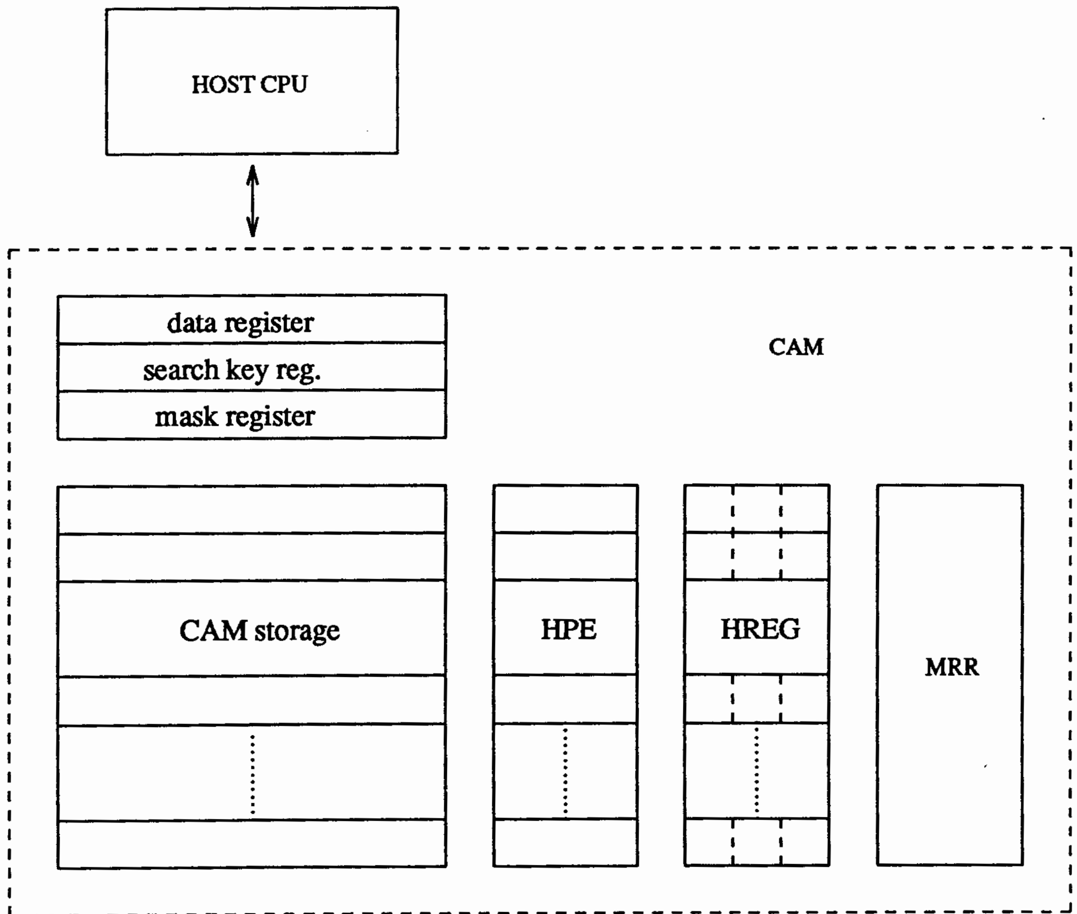


Figure 2. Block diagram of a CAM.

CAM BUILDING BLOCKS

CAM Storage

The distinguishing feature of a CAM is its ability to compare search data and stored data. The equality comparison is the product of the bit by bit logical equivalence between key bits (key_i) and bits of the stored data ($memory_{ji}$) for each of the b bits of the word. The result of the comparison of word j with search word, key, is given by the logical function HIT_j shown in equation (1).

$$HIT_j = \prod_{i=1}^{i=b} \overline{(memory_{ji} \oplus key_i)} \qquad (1)$$

CAMs that only perform an exact or perfect match have limited application because only a test of the presence of a copy of the search key stored in memory is possible. For example, the unmasked search for the relation (John, father, Mary) in a database that stores family relations can only answer whether the relation is true or false. To increase their functionality, CAMs will generally include an additional control for each bit of the word to select the bit columns that participate in the search ($mask_i$). The control signal $mask_i$ prevents the bit stored in column i from affecting the result of the search. Equation (1) is modified to reflect this feature in equation (2).

$$HIT_j = \prod_{i=1}^{i=b} (\overline{(memory_{ji} \oplus key_i)} + mask_i) \qquad (2)$$

Many terms are used in the literature to refer to this kind of comparison. Among them are: *masked exact match, masked perfect match, exact match with mask, perfect match with mask, masked search, etc*. In this work I will use the term masked search. Wherever a search is mentioned without specifying if it is a masked or unmasked search it can be assumed that it is to a masked search. The operation of selecting bits will be called *masking* and the circuitry to implement masking, *masking circuitry*. Masked search is the basic operation of CAMs. By selecting the bits which will participate in the comparison, the CAM is capable of selecting words by partial information, in effect, direct association. The masked search for relations matching (*, father, Mary) returns the identity of the father of Mary. The asterisk (*) is used in this thesis as a "wild card" that will make a search match to a

sequence of symbols. And the question mark (?) is used as a wild card that matches to any single symbol.

The circuit in Figure 3 implements equation (2) for one bit. The bit is stored in a static memory cell that is selected by the signal word select (WS_j) high, the comparison is implemented with a pass-transistor XNOR gate, and the results of the comparisons are accumulated in a wired-NOR gate. HIT_j is evaluated by wire-anding the HIT_{ji} of each of the CAM cells of word j. External masking is achieved by driving both key_j and $\overline{key_j}$ to low at the same time.

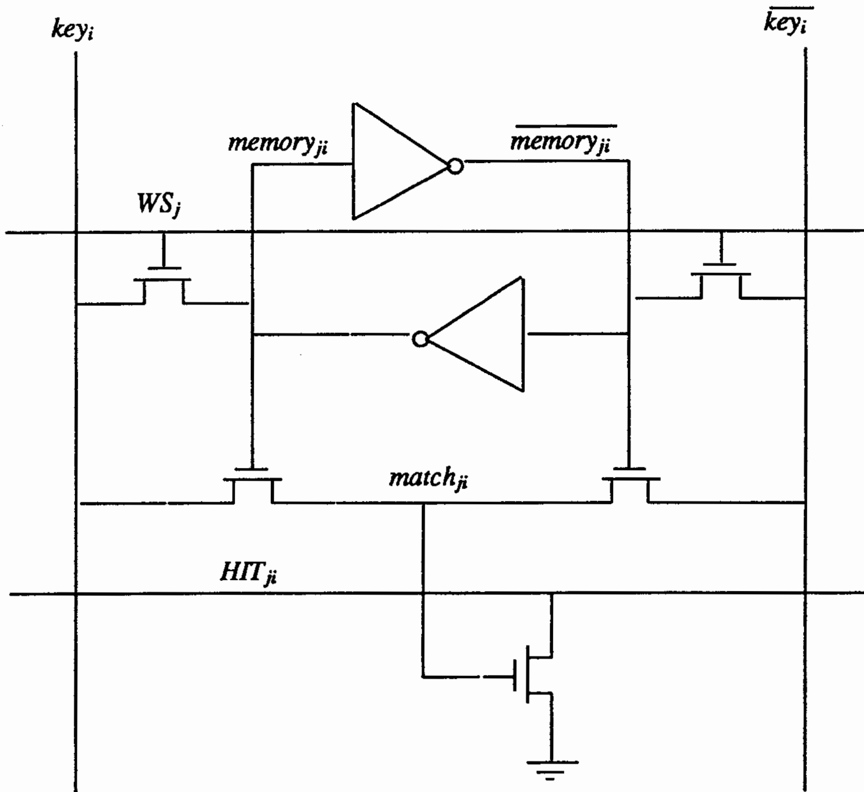


Figure 3. Static CAM cell.

In *external masking*, external information is used to generalize the search. By selecting which bits of the memory word will participate in the comparison, the CAM with external masking is capable of selecting words by partial memory information. For applications that require that each word masks different bits, another model of CAM stores an individual mask for each word. This kind of bit masking for individual words is called *internal masking*. CAMs that have internal masking are sometimes

called *functional memories*. Functional memories can select words by partial search key information. Memory words with internal masks use local information to generalize the search and match a larger number of binary patterns of search keys. For example, a database that stores qualities of John and Mary might store:

individual	quality
John	handsome
Mary	pretty
*	sophisticated

The quality sophisticated is shared by John and Mary and is internally masked to match searches on either (John, *) or (Mary, *). The seach with external mask (John, *) matches the qualities, or words, handsome and sophisticated.

Equation (3) gives the functional description of the match on CAMs with internal and external masking, where $Imask_i^j$ is the internal mask of bit i of word j.

$$HIT_j = \prod_{i=1}^{i=k} \left(\overline{(memory_{ji} \oplus key_i)} + mask_i + Imask_i^j \right) \qquad (3)$$

CAMs implement internal masking either through functional memories or by using more CAM cells with external masking. Functional memories cells have an extra storage cell to store the internal mask for each word in memory. Figure 4 shows the schematic of a functional memory cell. The storage cell in the lower portion of the figure stores the internal mask. Internal masking can be emulated in CAM that have external masking by reserving two memory bits to store each bit of data. For example, the logical 1 can be stored as the pattern 01, logical 0 as 10, don't care as 00 and contradiction as 11. A search for a 1 is converted to the masked search for the pattern 0? and the search for logical 0 into the masked search for the pattern ?0. Chapter IV will show one application of CAM for logic minimization where unmasked data is used to represent a minterm and masked data is used to represent a cube.

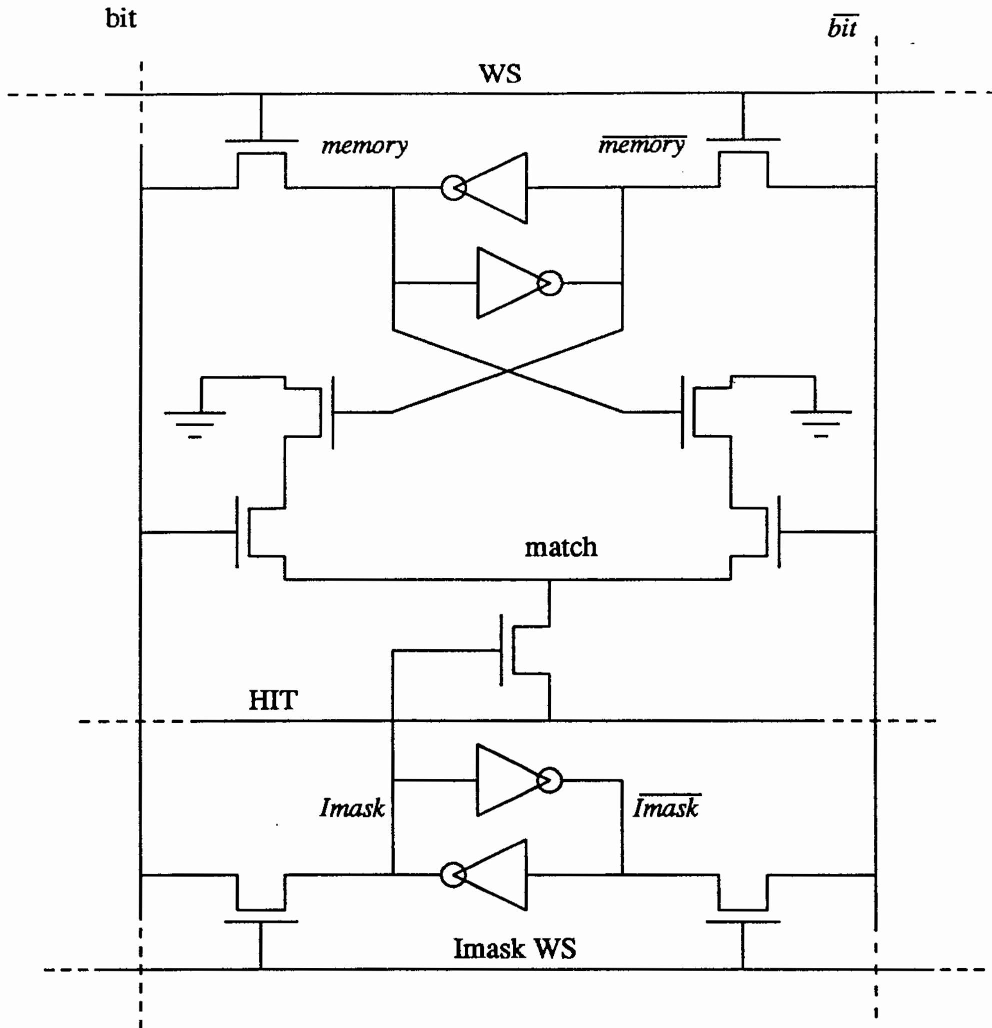


Figure 4. Functional memory cell.

Multiple and Partial Access

This section discusses two important features of CAMs. The first one is partial access, which is used for bit manipulation of memory words. It will be shown that bit manipulation adds processing to the CAM. The second feature is multiple access which adds a fine-grained parallelism to the bit manipulations.

The *partial read* uses the concept of a "mask" for a read operation. Only selected or unmasked bits of a word are read, preserving the previous value of the masked bits in the data register. The partial read operation combines bits from one word in memory with bits in the data register. The infor-

mation in two or more memory words can be combined inside the CAM without transferring data to the host processor. Combining bits of two different words in a RAM-based computer system requires processing and two bus transactions between memory and processor. The same operation in a CAM with partial read capability with different masks reads bits of both words into the CAM data register, combining the bits of the two (or more) internal read operations into a single word. The processing is done in the memory reducing the processor workload and the data traffic between the memory and the processor.

The *partial write* feature of CAMs executes the dual operation of a masked read storing the final result in the memory cells instead of in the data register. The masked bits of the matching word are not written, or modified, preserving their previous values. At the end of the partial write operation, the word in memory is altered.

The importance of multiple and partial access is illustrated with the very common situation where data words are divided into fields, artificially delimited in the examples by commas (,), and the memory and the processor have to combine one field of word A with another field of word B.

(apple,red); (red,sweet) => (apple,sweet)

(1010,1010); (1100,1100) => (1010,1100)

In the above example, apple, red, and sweet are binary fields much like the binary patterns of the second example. The joining of two fields is achieved with two partial access operations with the appropriate masks. To achieve the same results with RAMs and a processor, both words A and B are transferred to the processor that combines them with logical operations.

((apple,red) AND (1...1,0...0)) OR ((red,sweet)) AND (0...0,1...1)) = (apple,sweet).

((1010,1010) AND (1111,0000)) OR ((1100,1100)) AND (0000,1111)) = (1010,1100).

Multiple write stores in parallel, the information in the input data register into more than one word in parallel, creating multiple copies of the data. Multiple identical copies in the CAM are many times useless because they all match the same keys and one copy of the data suffices. There are situations, however, when it is desirable to store identical memory words. Among them are the cases where the requested information is only the number or pattern of matching words (HIT) [HIRATA88,

TAVANGARIAN89].

The *multiple read* operation permits access to the data bus by more than one word at the same time. During the multiple read, the data bus executes an analog sum of the contents in the accessed memory words which, then, is sampled by the data register.

The multiple read and multiple write features have limited applications but, the multiple write combined with partial access will be shown to be a powerful feature. While the partial access operations enable bit manipulation, they have to do so sequentially for each matching word. Multiple access increases the parallelism of the CAM architecture for data manipulation. *Multiple partial writing* can execute in parallel the modifications in the same field of all matching words instead of having to modify each word sequentially in multiple read-modify-write cycles or in many partial write cycles.

The partial read is easy to implement. The read cycle is performed normally for all columns but only data registers of unmasked bits will sample the data bus. The multiple write feature is harder to implement because it requires strong data drivers. The capacitive and resistive load of the memory cells selected for the multiple write are added, limiting the speed and the practical maximum number of words that can be written at the same time.

Search Output

The results of a search must be available outside the CAM IC. One important feature present in most CAM architectures is a signal to flag matching words after a search. SOME/NONE is the binary function that collects the responses of each of the w words in the memory to determine the existence of matching words. Equation (4) describes SOME/NONE as a logical OR of the HITs of each one of the words in memory.

$$SOME/NONE = \sum_{j=1}^{w} HIT_j \tag{4}$$

Similar output signals from a search are the number of matching words, mismatching words, unused and used words. Many applications require the calculation of SOME/NONE or an equivalent function. This value is provided in a CAM through hardware. The function SOME/NONE is usually implemented by a wired-NOR of the HIT_j of each word. In associative memories implemented with

RAM and search algorithms, SOME/NONE is calculated during the sequential search.

Multiple Response Resolution (MRR)

The comparison logic circuits in each of the words work in parallel and independently. Therefore, more than one word may match the same search key. If the result of the search is directly used to read the matching words, the voltage in the data lines during the read operation will be the analog sum of the contents of all matching words (see multiple read above). A means to select one word at a time has to be provided to successfully read each value individually. The circuit that performs this task is often called *multiple response resolver (MRR)* because it solves the conflict between the multiple matching words that want to access the data bus, or *priority resolver* because it prioritizes the matching words to determine which one will have access to the data bus first.

The MRR is needed because there is only one data bus shared by all words. For example, if we have stored (John, father, Mary) and (John, father, JohnJr) and we search with the key (John, father, *) both relations are found. The MRR is used to select which one will be accessed first. Multiple response resolution is unnecessary with search algorithms because of the sequential nature of the von Neumann architecture. Likewise, the MRR can be left out of CAM architectures dedicated to applications for which it can be guaranteed that multiple matches will not happen.

The performance of the MRR directly affects the performance of the CAM memory because it has a key role in determining the maximum rate of access to data. The access to the bus between memory (CAM) and processor is again the limiting factor for performance. But this time the requirements on the bus bandwidth are smaller because only qualified data (matching words) are competing for the resource.

The implementation of MRRs, or the problem of converting a pattern with many scattered ones such as the result of a search into a selection pattern with a single logic one is well known. Lee [LEE85] divided the MRR circuits used in CAMs into two classes. One class of MRR prioritizes the responding words based on the data contents of the word. This scheme is consistent with the *addressless* model of CAMs. He called this scheme of MRR *value-ordered* retrieval. The value-ordered MRR resembles a machine that starts with the set of words matching the specifications given by the

user and continues to "trim" the set of matching words using a sequence of searches determined by the MRR until it reduces the initial set to a set simple enough that it can access the data bus without conflicts. The other class of MRR prioritizes the responding words according to their physical location. It was called *address-ordered* retrieval. The address-ordered MRR resembles the combinatorial logic found in a daisy chain. The priority of a word is defined by its position in the daisy chain. Address-ordered retrieval introduces the association of a physical location to a memory word from the RAM architecture to the CAM architecture.

The simplest approach to implement the priority logic for an address-ordered scheme is to build a chain of simple iterative circuits that can inhibit the output of cells that are lower in the chain. The delay and the circuitry of the MRR grow linearly with the number of words. Since response resolution is essential to determine the memory access time, a slow MRR circuit will negatively affect the performance of the CAM. Foster [FOSTER76] and Anderson [ANDERSON74] proposed to generate the inhibit signal combinatorially to speed up the MRR process similar to a carry look-ahead of an adder. These proposed schemes use a tree-like structure to generate the inhibit signals to the words with lower priority. The tree structure provides logarithmic settling time in exchange for the exponential growth in the number of gates.

Ogura [OGURA85] also proposed an area/speed compromise with a more "flattened" tree structure. The number of levels and words grouped in each look-ahead block is defined by the optimization of delay and the amount of hardware used. The calculation of the critical path delay is similar to the calculation of the critical path delay of the Manchester carry look-ahead in adders. Because of layout and speed considerations it is probable that the capacitance of the wired-NOR SOME/NONE will be broken into many partial SOME/NONEs and more levels of wired-NOR will be added to generate the global SOME/NONE signal. Notice that the split SOME/NONE is functionally equivalent to the carry look-ahead used in the priority resolver. The hardware of the SOME/NONE can be shared with the address-ordered MRR.

The delay and circuitry of address-ordered schemes are strongly correlated to the number of words in the CAM memory. The alternative scheme of value-ordered prioritizes responders according to their contents [RAMAMOORTHY78, LEE85]. All responding words must have different values in

order to have unique responses resulting from ordered retrieval. To guarantee this, Ramamoorthy [RAMAMOORTHY78] proposed that each word should have a tag with a distinguishable value. That tag has to be at least $\log_2(w)$ long for a memory with w words. One disadvantage of using tags, especially if they are hardwired in the design, is that the regularity of design would be smaller because each word is designed with a different tag. Without the individual tags, memory words that store identical data will access the bus together, but without contention. It is impossible to identify the location or the number of words that store the same data accessing the bus at one time. This precludes the application of this kind of addressless CAM for applications based on the pattern of matching words.

The time needed to select a responding word in the ordered retrieval scheme of MRR is the time used to sort the matching words. In CAMs, this time is proportional to $\log(b)$, when the word itself is used for sorting, where b is the number of bits of the memory word, or proportional to $\log(\log(w))$ when a tag is appended to each word. In either case, the response resolution is much faster than the address-ordered MRR schemes. Also, it will be seen later that the tag used for priority resolution will find applications in testing and address encoding and address decoding.

Address-ordered MRR that achieve a logarithmic memory size dependence of the speed of response resolution use priority trees with extensive use of hardware [FOSTER76, ANDERSON74]. Value-ordered retrieval achieves logarithmic dependence using the hardware that already exists in the CAM design. Additional hardware can be added to further improve the performance of value-ordered MRR [LEE85].

Address Encoder and Address Decoder

The "addressing" for a content-addressable read and write access uses the accumulation of one or more consecutive searches in the HREG (see the description of the HREG ahead) to access all matching words in parallel or select them sequentially with the MRR through a feed-back circuit that drives the word select lines (WS) with the output of the HREGs or with the output of the MRR. In the sequential access, the MRR selects the matching word with the highest priority. Words with lower priority are accessed without having to repeat the searches by using another feed-back circuit to reset

the HIT register bit of the word selected by the very output of the MRR. With the highest priority word reset, the priority resolver selects the word of next highest priority. This process can continue until all words have been accessed.

This model of memory access is as powerful and complete as the RAM memory model. But, until microprocessors and application programs under this model are developed, the CAM should also provide access by address to emulate RAMs. RAM emulation utilizing the procedure used for CAM addressing is slow. The conventional solution to this problem is to include an address decoder and an address encoder in the CAM architecture. The address decoder provides the compatibility with RAM-based computers and the association of the memory words to a physical location provides an alternate output form to the result of the searches. Then, the compatibility with RAMs can be achieved by including the RAM circuitry with the same mechanics of the RAM model. CAM architectures that use this alternative pay a high price for compatibility because they must implement the hardware of both models. The commercial CAM IC Am99C10 [AMD88] and the DBA [WADE89] follow this approach. The scheme found in [YASUURA88] optimizes the emulation of RAMs while trying to minimize the departure from the CAM model. In this alternative, a *content addressable read-only memory (CAROM)* field with a different binary pattern is tagged to each word. Figure 5 shows the schematic of a CAROM cell. When emulating a RAM, the addressing of a word is converted to a search on the CAROM field and the MRR is by-passed. The uniqueness of the address searched in the RAM model guarantees that the MRR can be eliminated from the critical path to memory access.

Independent control for the CAROM and CAM cell drivers enable the CAROM to be read during any content-addressed read and write cycle since the CAROM field stores the address of the word during reads and writes. With the bits of the CAM cells masked, the CAROM acts as a substitute for the address decoder in read and write operations. During search operations, the CAROM field acts as an address encoder. The CAROM field can also be used in the implementation of a value-ordered MRR.

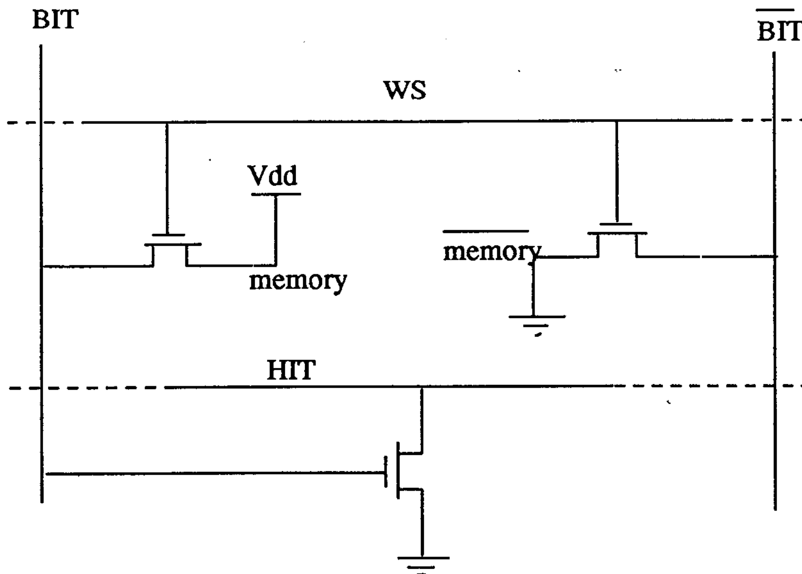


Figure 5. Content addressable ROM (CAROM).

HIT Processing Element (HPE) and HIT Registers (HREG)

Many CAM architectures include registers and arithmetic and logic units between the HIT lines and the MRR to enhance the CAM processing power [WADE89, LEA86b, DULLER89, YASUURA88]. In general, each HPE is a bit-slice processor that has a registers set, the *HIT registers (HREG)*, and the memory bits of the CAM word to work on. The size and complexity of the HIT processor element varies depending on the intended application of the CAM design.

A CAM with HPE is a parallel computer with *w* bit-slice processors, one for each word. The processors work on their own set of registers and memory. Because of pin limitations, generally, all the HPEs share the same instruction bus, and execute the same instruction synchronously as in a *Single Instruction stream Multiple Data stream (SIMD)* computer. Additionally, the execution or the decoding of the instruction in each HPE can be conditioned by values stored in the local HIT registers [FINNILA77].

With the addition of the HIT processing elements and the HIT registers, the CAM can execute more complex searches. The result of individual searches are stored in the HIT registers and are used by the HPE to compose the result of many searches into a complex selection.

Communication Between Words

The class of problems that the processing elements in the CAM HPE can solve is still limited because the HIT register composes the results of different masks and search keys within the same word. In order to use different stored data (*memory*$_i$), there must be a write operation between the searches. To increase the class of problems that the CAM can solve, the HPEs have to be able to cooperate and share the data stored in different CAM words. An *interword communication network (WNET)* for the HPEs is added in some CAM architectures for this purpose [JONES88a, FINNILA77, WADE89]. With the interword communication, the HPEs can solve problems that require information stored in many words. By adding communication, data structures larger than the CAM word can be stored and searched [ADAMS86, ASP88].

GENERAL ISSUES IN THE INTEGRATION OF CAMS

The many general issues of integrating the many building blocks discussed and the use of the CAM integrated circuit as a building block of higher level systems are presented. A comparative analysis of the integration of CAMs and RAMs is provided.

Selection of the Architecture and Features

There must be a coherence between the intended use of the CAM IC and its architecture. Basically, there are two major roles for CAM ICs in computer systems. The first type of computer system uses the CAM as an "intelligent memory". The function of the CAM in the computer system is to store information and retrieve it organized by associations. The CAM used for this purpose is strong in comparison logic and in structured data support. The CAM IC in the second type of computer system modifies the data stored in it in addition to re-organizing the data. These CAMs are virtually indistinguishable from processing devices. There is a thin line, if any, dividing CAMs with high support for data manipulation and content-addressable or associative processors. A characteristic of these CAM architectures is the enhanced capability to access and modify data stored in them.

An important architectural decision is the number of HIT lines per word. Looking to the CAM as a processor, the HIT line is the von Neumann bottleneck between the memory word (memory

device) and the HPE (processing device). CAM architectures designed for data manipulation like GLiTCH [DULLER89] and SCAPE [LEA86b] use two HIT lines to increase the access of the HPE to data. The architectures in [DULLER89] and [AMD88] have one extra HIT line exclusively for the almost unavoidable "tag bits" to support memory management.

Multiple and partial word accesses are important for CAM architectures in which data processing plays an important role. Internal masking, on the other hand, is most often found in architectures dedicated for searching. Mundy [MUNDY72] noticed that both features are rarely needed in the same application.

Other architectural decisions include the degree of complexity of the HPE, WNET and the number of HREG. All of these decisions depend on the role intended for the CAM in the computer system.

Fabrication Technology and Lay-out Scalability

CMOS is the fabrication technology used for most logic circuits. However, memories are usually designed in NMOS. The CAM ICs described in [JONES88a] and in [WADE89] were fabricated in CMOS but the memory cells used n-type transistors only because the well distance necessary for the CMOS design would make the memory cells excessively large. One disadvantage of using only n-transistors for the memory cell is that the HIT evaluation is slow. Sensing devices, like the ones used to read the bit lines, have to be added to speed-up the HIT evaluation.

The fact that CAM has both logic circuits and memory in the same design adds complexity to the fabrication process and to the incorporation of the latest DRAM fabrication technological advances. The technology used for CAM fabrication has to provide a reasonable yield for both memory and logic and the scaling of the CAM integrated circuit will be paced by the slowest scaling rate of logic and memory. This technology constraint is not unique to CAMs. Microprocessors that integrate cache memory also require a technology optimal for memory and logic. While the cache memory cells and logic cells in the microprocessors share the same technology, the design of the memory cells and logic cells are developed independently. However, in the CAM, memory and logic cells are closely knit together. The horizontal and vertical dimensions of the content-addressable

memory cell design are constrained by the size of the logic cells. Scaling down of any dimension of the memory cell is constrained by scaling down the corresponding logic circuitry by the same factor and vice-versa. This consideration is found in [HIRATA88] and in [JONES88a] where memory cells match the pitch of logic cells to prevent the waste of area in connections and to minimize the capacitance of the HIT lines.

Another difference between RAM and CAM requirements for process fabrication is that CAMs have the HIT lines running orthogonal to the data lines. The HIT lines also have to have low resistance and capacitance (RC) to keep delays acceptable. Therefore, the CAM fabrication process must have, at least, two high quality interconnects [KADOTA85, WADE87].

Due to these characteristics, the integration density of static CAMs is half of those of SRAMs [ADAMS86], and the density of dynamic CAMs is comparable to the density of SRAMs [WADE88, HERRMANN91]

CAM ICs also require more expensive packaging because, for the present word size and integration, a CAM requires more pins per package than the RAM with the same storage capacity. For example, a 32 Kword x 32 bit, or 1 Mbit CAM IC would have 32 data bits plus the instruction bits for the additional features of CAMs. A 1 Mword x 1 bit RAM IC has a single data pin and 20 address pins. This difference comes mainly because the logic of the CAM requires the memory cells to be organized in longer words which may require more data pins.

Modularity of the CAM IC

Modularity in CAM designs is that property of design that enables the assembly of larger memory systems with minimum design effort. Modularity is important in two levels of the system. In the IC design level, a modular design is important so that larger CAM chips can be developed with minimal additional design cost, at the board level, the goal is to minimize external "glue circuitry" and therefore design effort, to build larger CAM memory banks.

At chip level, the CAM design is very modular. It can be extended by increasing the number of bits in a word or by increasing the number of words in an IC. Analyzed as processing devices, the CAM ICs have a high degree of modularity. Wafer scale parallel processors were proposed using the

CAM architecture in part because of that modularity [ASP88, FINNILA77]. But as memory devices, providing modularity to CAM ICs is more complex due to the larger functionality of CAMs.

The provisions necessary to build longer RAM memory banks is minimal because the only functionality that has to be preserved is addressing. Integrated "chip select" logic is enough to provide modularity at the board level. CAMs have much more functionality that must be preserved from the IC level to the board level. RAM ICs do not require extra circuitry nor "glue logic" to integrate thirty-two 1 M x 1 bit RAM ICs to build 1 Mword of 32 bits memory. For the same amount of memory (1M 32-bit words), patching together thirty-two 32 Kwords x 32 bit CAM ICs requires much more effort than patching together thirty-two 1 M x 1 bit RAM ICs. The CAM IC must integrate support circuit or there must be "glue logic" on the board to implement all the features at the CAM IC level to the CAM memory bank. Among the common features to CAM ICs that should be supported at board level are signals of the class of SOME/NONE and the number of used or unused words, MRR, and a interword communication network.

The structures of the circuits to implement features at the board level have the tendency to replicate the structure of the circuits used to implement the same features at the IC level. For example, address-ordered MRR that use a linear chain will have the CAM ICs connected in a linear chain to implement the MRR at board level (see Figure 6).

The discontinuity between chip and board level lends itself to the creation of hierarchical schemes to manage the global outputs of the CAM ICs [OGURA85, RAMAMOORTHY78]. In the hierarchical scheme, the outputs of the CAM IC such as SOME/NONE and the matching word of

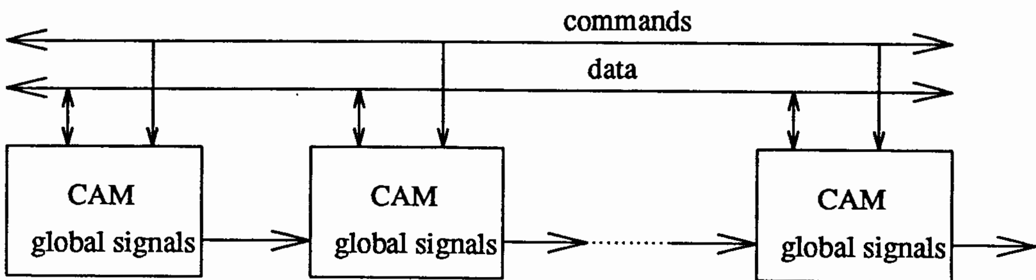


Figure 6. CAM bank built with minimum additional logic.

highest priority are collected by external circuitry and treated as responses of individual words to generate a SOME/NONE output for the data bank and to perform the MRR between the matching words of all the CAM ICs at the board level (see Figure 7).

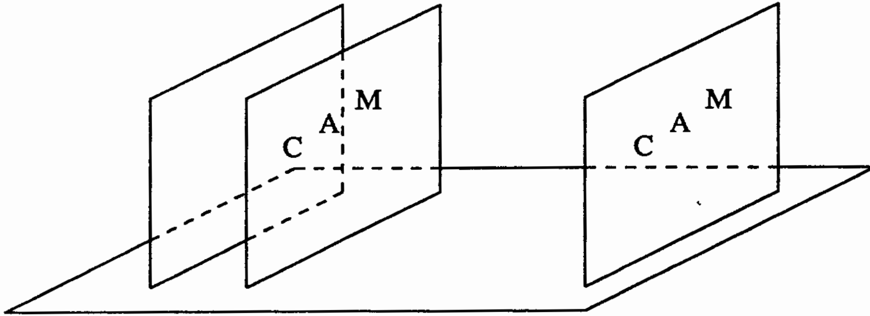


Figure 7. CAM bank with hierarchy.

Reliability, Testability and Fault-Tolerance

This section analyzes three inter-related issues, reliability, testability and fault-tolerance. The link between these issues are the *circuit failures*. Circuit failures reduce the confidence in the information provided by any circuit. As a consequence, the results of the computations of the overall system are less reliable. This section discusses the design techniques to increase the confidence in the output of the CAM. Using only circuits that are not faulty is one way to increase the reliability of the system. Testing finds whether a circuit is faulty and, if possible, diagnoses the fault. The importance of testing in the overall cost of ICs and the importance of design for testability have been continuously increasing. CAM ICs are no exception to this trend. With the faults identified, special circuitry tries to increase the yield of CAM ICs by "salvaging" CAM ICs with a small number of faults. Fault-tolerance increases the yield and enables the design of larger CAM ICs.

CAM ICs have two distinct sections to test, the memory and the logic circuits. Grosspietsch [GROSSPIETSCH89, GROSSPIETSCH87, GROSSPIETSCH86] proposed a CAM architecture that divides the CAM architecture into subcircuits for testability purposes. The major building blocks of the CAM ICs (see Figure 2) are divided into three groups for testing. The first group consists of the data, key, and mask registers. The second group consists of the circuitry used to process the HITs, the HPE logic, HREG, MRR, WNET, etc. The CAM cell fabric is the third group. Observation and

control points are placed on the interface of the subcircuits to facilitate testing.

The first two groups have only logic circuits and can be tested with conventional techniques. Grosspietsch [GROSSPIETSCH89] proposed that the key and mask registers be also readable to enhance the observability of the first subcircuit. Access to the HIT register and to the priority resolver is proposed to solve both lack of observability and controllability of the HPE logic and CAM cells.

The *scan path* testing scheme is very popular in CAM ICs designed for testability because CAM architectures that have the HIT registers configured as shift registers to allow storage of longer data words can be converted to *Built-In Logic Block Observation elements (BILBO)* with minimal modifications. BILBOs are elements used in scan-path and signature analysis styles of design for test. [BENNETTS84].

The testing of the CAM cell array is also simpler than it may appear initially. Even with more functionality and circuitry to be tested than a RAM cell array, the testing of CAMs can be significantly shorter [MAZUMDER88]. The testing is aided by the fact that the subcircuits can be used to help test each other. For example, the HPE logic can be used to help test the CAM cells by analyzing the patterns of HITs as a signature analyzer and the HITs of the CAM cell array provides a binary pattern to test the HPEs. The signal SOME/NONE also contributes to the testing of the CAM IC.

Content-addressable memory cells have both storage and logic to be tested. The testing of the storage part is similar to the testing of RAM cells. The types of faults that affect RAMs are basically of two types:

(1) hard-errors: stuck-at and bridge types of faults and

(2) soft-errors: pattern sensitive, coupling faults, and radiation.

The logic circuit in the CAM cell that has to be tested is a XNOR that compares the data in the storage part and the binary pattern in the bit lines. That is exactly the kind of circuit that is used to test the storage part. The conventional test of storage cells is to write a pattern and read the pattern back to determine if the patterns match. Rather than testing the logic and the storage individually, both parts of the CAM cell can be tested together. The CAM cell integrates the comparator with the storage cell as if the CAM cell were a RAM with built-in testing circuits.

As a final remark on testing of the CAM cell, pattern-sensitive and coupling faults are tested with patterns based on the physical location of the stored data. Therefore, for testing, we need to access the CAM by physical location instead of by contents.

With the faults detected, the next step is to improve the yield with special circuits that enable the use of ICs that have a limited number of faults. The tests for fault repair have to be more complex than the tests for fault detection to diagnose and locate the defects to enable the reconfiguration and repair.

The options proposed for repairability are redundancy and graceful degradation. Redundant schemes have spare words or spare bit columns to substitute faulty words or bits, respectively. Graceful degradation allows the deactivation of faulty words so that the IC is still functional, although with a smaller number of words. For large CAM ICs the strategy of graceful degradation appears to be superior than redundancy [GROSSPIETSH89]. CAMs designed for graceful degradation are naturally fault tolerant because there is no minimal memory size to have a working component. As long as defective circuits can be rendered harmless, the CAM IC will remain functional with the remaining logic [BLAIR87].

Also, because of the characteristics of the "built-in" CAM testing, the graceful degradation of CAM ICs can be managed with a minimum of test equipment and in a uniform way. Blair [BLAIR87] presented a CAM design with graceful degradation in which the CAM IC disables faulty circuitry by itself. Each word in this CAM has a latch that can be accessed only in testing mode that disables the match of its word. Testing is performed with the help of the comparison logic using the following test-and-repair sequence:

(1) write the test pattern to all memory words;

(2) perform a comparison with respect to the test pattern;

(3) latch the result of the comparison.

Defective words will mismatch the test pattern and be disabled in future searches. For more complex architectures, the value stored in the repair latch is also used to direct internal signals past the faulty word, like the signals of the address-ordered MRR and the WNET, rendering the word completely

harmless.

In conclusion, the ideal CAM IC must support enough processing to solve non-numerical and logic processing problems because these are the applications for which the current RAM-based systems are deficient. A balance between processing power and storage capacity is obtained with the following characteristics:

(1) provide multiple partial access;

(2) support data types longer than the word length;

(3) have linear communication between words;

(4) have a HREG register bank; and provide logical operations on the HITs;

(5) have on-chip circuits to allow expansion to larger memories.

(6) have built-in testing schemes such as connecting the HREGs to form a scan-path;

(7) have low-cost repair circuits with the scheme similar to the ones used in [BLAIR87] and [MCAULEY90];

(8) have direct memory access based on data content.

Based on this specifications an integrated circuit designer can select the circuits to implement the CAM IC. If resources are allocated to this end, the described CAM IC with high densities can be built in the near future. The question that arises is: *Is it worth while to implement such a circuit?* The next two chapters will analyze the implications of using CAMs to implement computer memories. It will be shown that the answer is yes. The content-addressable memory described fulfills the goals set out in Chapter I.

CHAPTER III

BASIC COMPUTATIONAL TASKS AND CAMS

This chapter describes the computational environment that uses associative memories in the form of CAMs and the changes in the execution of common computational tasks which result from CAMs. It will be shown that computers that use CAMs could be a fundamental addition to computer design and to non-numeric data processing. Most of the chapter is devoted to the data structures ideally supported by CAMs.

DATA STORAGE

Probably the most important function of the memory device is to store information. The structure of the memory device affects the efficiency of implementation of the data structures used in computation and, consequently, the performance of programs.

The structure of the memory words in RAM is the linear array with an implicit order given by the addresses. This structure maps well to linear and multi-dimensional arrays (matrices) that are used in most numeric computations, and to other data structures in which *order* plays an important role. The set is identified as the natural data structure for CAMs. The mapping of other data structures to CAMs are also analyzed. An illustrative example is analyzed at the end of the chapter to gauge the effect of storing data in a CAM on basic computational tasks.

Sets

Sets are the foundation on which virtually all of mathematics is constructed and many mathematicians believe that it is possible to express all of mathematics in the language of set theory [STANAT77]. CAMs are the ideal medium to implement sets in the same way RAMs are ideal to implement arrays. Elements are naturally stored without order in CAMs. I present here one scheme to

store multiple sets in CAM fully supporting the basic operations and relations of sets.

Sets are defined as collection of objects called *elements* or *members* without duplication nor order [AMSBURY85]. Some definitions of sets allow repeated elements in the set [STANAT77, KOLMAN84]. A set is described by its elements. Therefore, a finite set can be described by listing its elements (e.g. set A = {1, 2, 4, 5}; set B = {a, i, u, e, o}). Sometimes it is inconvenient or impossible to list all elements of a set. Other useful ways to describe a set are through specifying properties that uniquely identify the elements of the set using mathematical or English statements, or by induction (e.g. set C = {x | x is an even number and x is smaller than 12}; set D = {1, 2, 4, ..., 2^i, ... }). It is assumed that there is an *universe of discourse* or *universal set (U)* that is a set that contains all elements for which the discussions and descriptions are meaningful [KOLMAN84] (e.g. "all natural numbers"; "the letters of the English alphabet").

There are several operations that can be performed on sets. The most important ones are: *union* ($\cup$), *intersection* ($\cap$), *relative complement* (-) also known as *sharp* or *difference*, and *symmetric difference* ($\oplus$).

The union of set A and B, denoted $A \cup B$ is the set with all elements of the set A and all elements of set B:

$$C = A \cup B = \{ x \mid x \in A \vee x \in B \}$$

The intersection of the set A and B, denoted $A \cap B$ is the set with the elements that belong to both sets:

$$C = A \cap B = \{ x \mid x \in A \wedge x \in B \}$$

The difference of A and B, also denoted relative complement of B with respect to A, or A sharp B, is the set of all elements of A that do not belong to the set B.

$$C = A - B = \{ x \mid x \in A \wedge x \notin B \}$$

The symmetric difference of sets A and B, denoted $A \oplus B$, is the set consisting of all elements that belong to either set A or to set B but not to both. It is easy to verify that $A \oplus B = (A - B) \cup (B - A)$.

$$C = A \oplus B = \{ x \mid (x \in A \wedge x \notin B) \vee (x \notin A \wedge x \in B)\}$$

There are two fundamental relations between two sets, equality and containment [STANAT77]. Two sets are equal if they have the same elements. Set A contains set B if all elements of B are also elements of A. The set B is then called a *subset* of A, and A is a *superset* of B. It is also said that B is contained in A, and we write:

$$A \supseteq B \text{ or } B \subseteq A$$

A very useful concept for sets is the *characteristic function* [KOLMAN84]. The characteristic function of a subset A of the universal set U, f_A is defined as follows:

$$f_A(x) = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{if } x \notin A \end{cases}$$

Since the characteristic functions are defined over numbers or logic values, logic or arithmetic operations can be performed with characteristic functions. As an illustration, some important properties, valid for characteristic functions defined as logic functions, are listed without proof:

$$f_{A \cup B} = f_A \vee f_B$$

$$f_{A \cap B} = f_A \wedge f_B$$

$$f_{A \oplus B} = f_A \oplus f_B$$

Similar properties can be derived to evaluate the basic relations of set theory. For example, if A = B, then $f_A - f_B$ is false (or zero) over the universe.

One alternative is presented to represent sets in a CAM in a way that provides an efficient implementation of the fundamental operations and relations of sets. An extensive review of the CAM literature indicates that generalizing CAM functions usings sets has not been done.

Implementation of the Set Data Structure with CAMs. In this proposed model, each word of the CAM stores one object of the universal set and a *tag*. The function of the tag will be explained later. We have seen that each set or subset can be described by the properties of its elements. Simple observation shows that the HITs are in fact the evaluation of the characteristic function for each element stored in the CAM. Therefore, the basic functions of the CAM support the relations and operations of set theory. The HIT output is the product of negated and non-negated sets represented as bits in the tag and is equivalent to logical ANDs and NANDs that are universal building blocks for more complex

operations. The HREGs and HPEs accumulate the intermediate results to build these complex operations. The function SOME/NONE is a building block to evaluate the basic relations of set theory. With RAMs, the set operations and relations are evaluated sequentially and the time required to execute most operations is proportional to the size of the set(s) involved. In the CAM, the equivalent set operations and relations are executed in parallel over all elements of the set, possibly over many sets, because attached to each word of a CAM is hardware powerful enough to execute the key set operations and relations.

The sets in a CAM are described by descriptions that can be built with logic and arithmetic operations on the HITs supported by the HPE such that after a sequence of searches describing a set, only the elements that are members of the set will still match the search. Sets that are hard to describe in terms of HITs because they would require long and complex series of searches can have their descriptions simplified by appending to each element of the set an identifier in the "tag". In effect, this identifier in the tag creates a property that helps to describe the set in terms of HITs. In addition to storing identifiers to support the selection of the members of a set, the tag is also used to delimit different universes of discourse, simplifying the memory management in a multi-task environment.

An example of set storage in CAM is shown in Figure 8. Notice that the database in the example stores two universes of discourses, fruits and colors. The color orange will not be affected by operations on sets of the universe fruits even though orange is also an element of the universe fruit. A search of the set:

$$A = \{ x \mid x \text{ is a color of the rainbow and } x \text{ is warm} \}.$$

can be calculated with the logical AND of the searchs on colors of the rainbow and warm colors. The search "color" AND "warm" AND "rainbow" correctly retrieves the elements red, orange and yellow of the set of colors without affecting the HREGs of the element orange of the set fruits during processing.

In the model presented, each object is stored with associations to all of its sets. This requirement enables the evaluation of operations among sets using the HITs as characteristic functions without having to use the communication between words and enhances the parallelism of the execution of the set operations. At the same time, it requires that the tag encodes the complete information regarding the

element	: tag
red	: color; warm; rainbow
orange	: color; warm; rainbow
yellow	: color; warm; rainbow
green	: color; cold; rainbow
blue	: color; cold; rainbow
indigo	: color; cold; rainbow
violet	: color; cold; rainbow
white	: color
black	: color
apple	: fruit
pear	: fruit
orange	: fruit
banana	: fruit
grape	: fruit

Figure 8. Set data structure implemented with CAMs.

element stored in that word. Consequently, this model requires a longer tag than the one used with a model that allows the associations to be distributed among multiple copies of the same element. It also requires that before storing any new object in memory, the presence of the element in memory is checked for duplication to maintain the information in the tag about the element complete. The check for duplication is fully supported by the signal SOME/NONE. The signal SOME/NONE also supports the relation of "membership" between an object and a set. The query to check if an object belongs to a set using SOME/NONE is executed in parallel, using the comparison logic with response time independent from the size of the set.

For sets, it is the RAM that has to emulate the set structure using structures more suited for RAM storage like trees and lists. Applications that use sets would benefit from the direct hardware support provided by the CAM. Analyzing the implementation of algorithms utilizing RAMs, it is easy to find examples where more complex structures like lists were used instead of the simpler set structure. Chapter IV presents in detail the applications of set theory and CAMs to logic minimization and other important applications.

Other more complex data structures can be implemented with the simpler set structure. Records are collections, or sets of data, each element of a record is also a sub-set that has to be uniquely identifiable to be accessed. Lists are sets with their elements or sub-sets ordered. Arrays are sets, too. To implement these more complex data structures, the sets and sub-sets are managed and organized

through the "tag bits". The implementation of data structures with CAMs is more flexible because the tags are programmable as opposed to addresses that are hardwired.

When compared to RAM, the degree of efficiency in which these other data structures can be implemented with sets varies. The applications that require more complex data structures have their performance conditioned by how well the other structures can be implemented with sets or how well the CAM architecture can implement those structures directly. The burning question is: *Is the performance of computer systems that use CAMs satisfactory for all kinds of applications? Or, at least, do they have a satisfactory performance over a wider range of applications than RAM-based computer systems?* This is one of the major questions this work expects to answer.

One of the effects of storing data in CAM is that the execution of basic computation is changed. Considerable effort has been dedicated to searching and sorting algorithms. Foster [FOSTER76] claimed that "at any given instant, half of the university computers in the world are compiling *(table look-up)* and half of the business computers are sorting." (page 125)

In all likelihood, searching and sorting will continue to be one of the most important computational tasks. Any proposed computer system should be efficient in searching and sorting. *Searching* with CAM is a breeze. In the literature, searching algorithms abound for CAMs [KOHONEN80, FOSTER76]: minimum, maximum, next above, next under, magnitude comparison, five way split, interval search, etc. The comparator added to the storage devices in a CAM directly supports searching tasks in hardware. Masked search is also implemented directly in the CAM hardware; the other types of searches can be implemented as sequences of masked searches.

Since data are not sorted with addresses, *sorting* cannot be executed by changing the physical location of data. Data are sorted by changing their positions in an ordered structure such as a list. But sorting can also be done dynamically as access to the data is required. Data are stored in CAM unsorted and can be read out of the CAM sorted using the search algorithm used to select the data (see the discussion of value-ordered MRRs in Chapter II). To sort from largest to smallest dynamically, the program searches for maximum and reads the selected word. The process continues with a search for the "next below" until all words are read (and hence sorted).

Sorting is improved in systems using CAM but the important question is: *Why are we sorting the data?* If the answer is: *to speed up searching*, sorting the data does not improve the speed of searching in CAM. Therefore, we probably could live without sorting [FOSTER76]. The processing and time delay for sorting is eliminated from the process of appending new data to the set and transferred to the time when the information is retrieved. The hardware support of the CAM enables the CAM to sort in parallel and retrieve data in the same order of time needed to retrieve data from a sorted set in RAM.

One non-numeric application that uses both searching and sorting, and is gaining in importance, is database search. There are two main kinds of database search. One is the search by keyword that can be speeded up in RAM with hash-coding. In CAMs, this kind of search is implemented directly at the transistor level. The other kind of search is magnitude search, which is speeded up in RAM by including a structure and sorting the data. Each expected magnitude search query must have its correspondent structure to implement the sorting. A magnitude search of income on a employee database sorted by social security number does not experience any performance improvement. RAMs utilizes additional storage cells and CPU time to solve a problem on a case by case basis. CAMs use their additional hardware in the form of distributed logic in the memory cell to accomplish a fast database search. If the importance of this type of application grows as predicted in Chapter I, it makes sense to think of a computer memory model (CAM) that goes to the root of the problem instead of fighting the side effects of RAM.

This chapter analyzed the effects of CAM-based computers in basic structures of computation. The next chapter will complete the examination of the effects of CAM-based computers on applications.

CHAPTER IV

CUBE CALCULUS

This chapter uses cube calculus to illustrate the support CAMs provide to the set data structure. The use of CAMs is not limited to applications that are based on cube calculus. The applications are presented using the cube calculus mathematical model to take advantage of previous work that used this model. It is the contention of this thesis that any application that maps well to set theory is well supported by CAM-based computers. It will be shown that cube calculus is a powerful mathematical model with many applications. The major cube calculus application treated in this chapter is logic minimization and synthesis of Boolean functions. Cube calculus and all applications in this chapter will be described using the logic minimization language. The basic elements and operations of cube calculus will be introduced informally, through examples and illustrations.

BASIC CONCEPTS

The first concept to be defined is that of the *multi-valued variable (MVV)*. The v-valued variable can assume any of the 2^v subsets that can be built from v elements. For example, a ten-valued variable has $2^{10} = 1024$ possible combinations, or possible sub-sets, of its ten values. Likewise, a two-valued variable can be instantiated by the 2^2 subsets that can be built with its 2 values.

The usual representation of a multi-valued variable is the *positional notation*. In this representation, each value, that can be understood as an element of a set, is represented by a binary digit (bit) being true (1) when the value is a member of the set, and false (0), when it is not. The values of the variable, or elements of the set, are ordered and represented in sequence. The first bit represents the first value, the second bit represents the second value and so on. MVVs and sets are equivalent mathematical elements. The positional notation is equivalent to the characteristic function described in Chapter III. The positional notation simplifies the execution of operations on MVVs as the charac-

teristic function does for sets. For example, a two-valued variable is represented by *two* bits in the positional notation because it has two values, namely 0 and 1 and can represent any of the four subsets that can be built with the two values ($\varnothing$, {0}, {1}, and {0, 1}).

The examples of MVVs that will be presented in this section will use a four-valued variable *v* with the elements V = {0, 1, 2, 3} and is represented in positional notation by a four bit bit-vector. For example, if *v* assumes the value v = {0, 2, 3}, it would be represented by the bit-vector v = 1011.

When a variable includes all elements, it is said that the variable is *full*. This is equivalent to the universal set. In positional notation, the variable is a string of 1's (v = 1111). When the variable does not contain a single value, the variable represents the empty set. It is called a *contradiction*. In positional notation, it is represented by a string of zeros (v = 0000).

Basically, the same operations defined for sets are defined for multi-valued variables. The logic operations on the positional notation are equivalent to operations on characteristic functions of sets. The power of the positional notation can be appreciated in the execution of logic operations. Logic operations over MVVs are transformed into logical operations over the positional representations. The *inversion* or *complement* of a variable *v* ($\neg$v or $\bar{v}$), is the set of values such as all the values that are not in *v* are present in $\bar{v}$ and vice-versa. The inversion of a multi-valued variable in positional notation can be executed with the bitwise logic inversion of the representation (one's complement). For example,

$$\begin{array}{ll} v = \{0, 2, 3\} & v = 1011 \\ \bar{v} = \{1\} & \bar{v} = 0100 \end{array}$$

In positional representation, the *union* ($\cup$) can be implemented by the bit logical OR of the bit-vectors that represent the values. For example,

$$\begin{array}{rlrl} v1 = & \{0, 1\} & v1 = & 1100 \\ v2 = & \underline{\{1, 3\}} & v2 = & \underline{0101} \\ v1 \cup v2 = & \{0, 1, 3\} & v1 \cup v2 = & 1101 \end{array}$$

Similarly, the *intersection* ($\cap$) is implemented in positional notation, as the logical AND of the representations of the variables.

$$\begin{array}{rl} v1 = & \{0, 1\} \\ v2 = & \underline{\{1, 3\}} \\ v1 \cap v2 = & \{1\} \end{array} \qquad \begin{array}{rl} v1 = & 1100 \\ v2 = & \underline{0101} \\ v1 \cap v2 = & 0100 \end{array}$$

The *exclusive-or* ($\oplus$) of two variables is equivalent to the symmetric difference of set theory and can be performed by a logical XOR of the representations.

$$\begin{array}{rl} v1 = & \{0, 1\} \\ v2 = & \underline{\{1, 3\}} \\ v1 \oplus v2 = & \{0, 3\} \end{array} \qquad \begin{array}{rl} v1 = & 1100 \\ v2 = & \underline{0101} \\ v1 \oplus v2 = & 1001 \end{array}$$

Two other concepts of cube calculus to be defined are the cube and the array of cubes. Cubes and arrays of cubes also represent sets. The elements of the universal set for this representation are the elements built by the cartesian product of the values of two or more multi-valued variables. The cube is a *cartesian product of MVVs*. An array of cubes is a set of cubes and represent the union of the elements of each cube in the array. In this section the cube represents a cartesian product of variables and an array of cubes is the sum of the cubes. Cubes and arrays of cubes can also represent other normalized forms of representation of functions such as product of sums, or exclusive sum of products or exclusive sum of sums, etc.

The term "cube" comes from a geometric idealization in which the variables are dimensions of a hyperspace. The vertices of the hyperspace that can be represented by a single cube are vertices of a hyper-cube in that hyperspace. One popular representation of the hyperspace is the Karnaugh map where the hyperspace is flattened to two dimensions and each element of the cartesian product, or vertex has a reticule to represent it.

To illustrate the use of cubes to represent sets we begin with an example with two-valued variables. The cartesian product of the two-valued variables $v1 = \{0, 1\}$ and $v2 = \{0, 1\}$, results in the universal set $U = \{00, 01, 10, 11\}$. The representation of a cube of this universe is the concatenation of the representations of both variables in positional notation. For two-value variables, $\varnothing = 00$, $\{0\} = 10$, $\{1\} = 01$, and $\{0, 1\} = 11$. The cube $c1 = 10.11$ represents the set of all vertices that can have the first coordinate 0 (10) and the second coordinate either 0 or 1 (11), this is the sub-set $c1 = \{00, 01\}$ of U. The cube with all variables replaced by a full variable represents all the elements of the cartesian

product and is equivalent to a "logical one" for logical functions. Any cube with an empty variable or contradiction does not contain a single element because the cartesian product is empty. These are *empty cubes*. The empty cube in which all MVVs are contradictions is equivalent to a "logical zero".

Using this format, not all sub-sets are representable with a single cube. For example, the empty set, can be represented by any one of the cube notations 00.00, 00.01, 00.10, 00.11, 01.00, 10.00, and 11.00. The single cube sub-sets are: 01.01, 01.10, 01.11, 10.01, 10.10, 10.11, 11.01, 11.10, and 11.11. It is necessary to use an array of cubes to represent the remaining sub-sets. For example, to represent the sub-set {00, 01, 11} we can use any of one of the following arrays of cubes: {10.11, 01.01}, {10.10, 10.01, 01.01}, {11.01, 10.10}, or {10.11, 11.01}.

However, a set with the same maximum number of elements as U, the universal set, could be represented with a single four-valued variable and is capable of representing each of the 16 sub-sets. That is, the four valued variable V = {0, 1, 2, 3}, represented by a four bit bit-vector, can represent any of the 16 sub-sets. For example, the same subset v = {0, 1, 3} that was impossible to represent in a single cube, is represented by the bit-vector v = 1101.

The Figure 9 shows a hyperspace built with the cartesian product of four two-valued variables along with the Karnaugh maps of hyperspaces of four two-valued variables, five two-valued variables and two four-valued variables. The higher expressivity of multi-valued variables of larger number of elements is illustrated by the cube representation of the same function in a hyperspace of four two-valued variables and in a hyperspace of two four-valued variables. Notice how the use of MVVs of larger number of values allowed a more compact representation of the set of reticules in the Karnaugh maps marked with a "1".

Although the cube and array of cubes representation of sets is less expressive than a large MVV, it is preferred over the pure set model for some applications because it maps better to those applications. For example, in logic synthesis, the cube representation provide insights into the implementation of the logic functions. The minimization of the number of cubes of a logic function also minimizes the number of gates in the implementation of that logic function. Notice that the representation of sub-sets of cube calculus matches the CAM model well because the sub-sets that can be represented by a single cube map directly to the "sets that can be described with a single masked search" discussed in

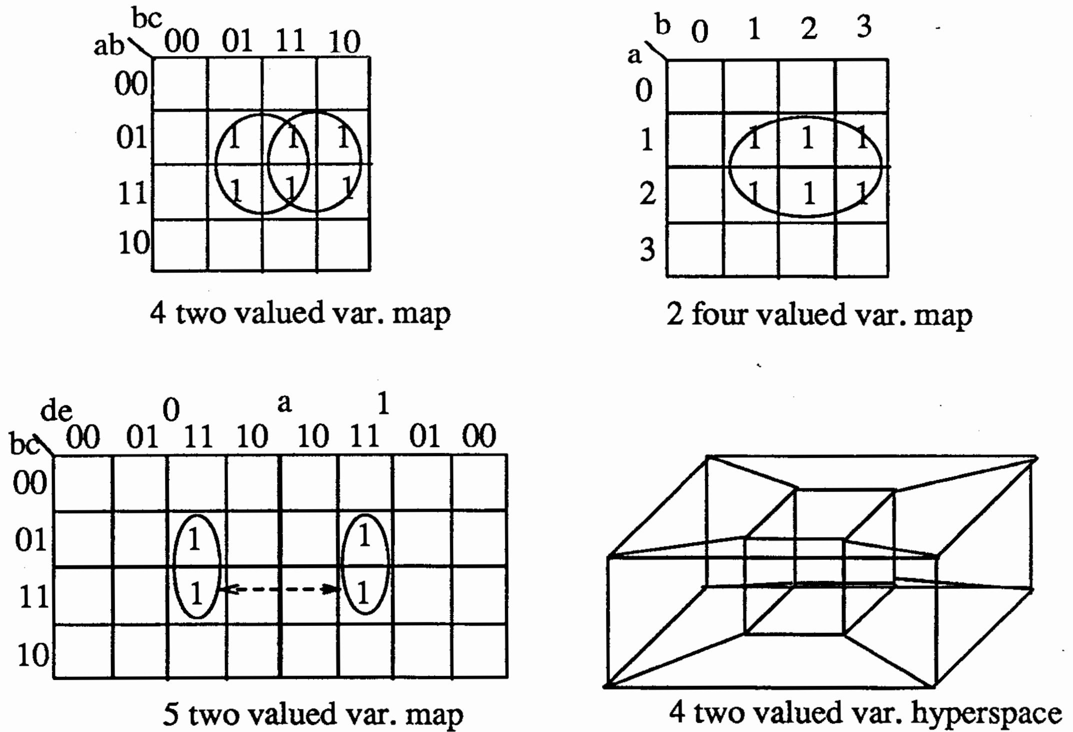


Figure 9. Examples of Karnaugh maps and hyperspace.

Chapter III.

Any sub-set of the universe is a *function* and is a set of vertices that can be represented by an array of cubes. To be more formal, a completely specified, single output function is a mapping of the points of the hyperspace to a binary value. An incompletely specified function maps onto the values {0, 1, X} where X is free to assume either the value 0 or 1. A completely specified function with multiple outputs maps into many binary values, a binary code. For simplicity, the examples presented are restricted to single output, completely specified functions. As a consequence, such function can be represented by the vertices that assume the value 1 because the function assumes the value 0 on all other vertices.

OPERATIONS WITH CUBES

The same essential set operations presented for multi-valued variables are adapted for the cube representation according to the works of Dietmeyer [DIETMEYER78] that present the cube operations

for two-valued variables and the works on MVVs by Sasao [SASAO84], Kuo [KUO87], Su [SU72] and Hong [HONG74]. When appropriate, it will be introduced how to extend the operations to arrays of cubes since some sub-sets require more than one cube for their representation. The union (OR), intersection (AND), sharp (NOT) and consensus are the most important operations in cube calculus. The notation introduced in Chapter III for characteristic functions will be adapted to represent the positional notation of cubes.

Cube Union

The union of two cubes or functions is the function that covers the minterms, or product of literals, or vertices contained in at least one of the functions. *Covering* is the term used in cube calculus for the containment relation. The most straightforward way to perform a union of two cubes, or cube arrays, is to build a result array of cubes with all cubes of the operands in it. The union of two functions F and G is the concatenation of the arrays of cubes of F and of G.

$$F = \{ f_1, f_2, ..., f_n \}$$

$$G = \{ g_1, g_2, ..., g_m \}$$

$$F \cup G = \{ f_1, f_2, ..., f_n, g_1, g_2, ..., g_m \}$$

The representation of the result will probably have more cubes than the minimum necessary to represent the union. Finding the minimum number of cubes necessary to represent a function is a very important minimization problem. To reduce the number of cubes used in the representation of the covering we try to remove redundant cubes and to replace two or more cubes by a larger one that cover the same vertices. *Redundant cubes* are cubes that only cover vertices that are also covered by other cubes in the array. *Absorption* is the process that finds and removes redundant cubes. Absorption decides if a cube or function covers another and removes the covered cube or function.

The Figure 10 illustrates the power of multi-valued variables for minimization. The operand cubes are represented by rectangles of broken lines and the resultant cubes are represented by circles or ellipses surrounding the elements of the cube. Arrows indicate that a cube is represented by two or more rectangles or circles for convenience of drawing. The union of the same operand two cubes can be represented by a single cube in a 2 four-valued variables hyperspace due to its larger expressivity.

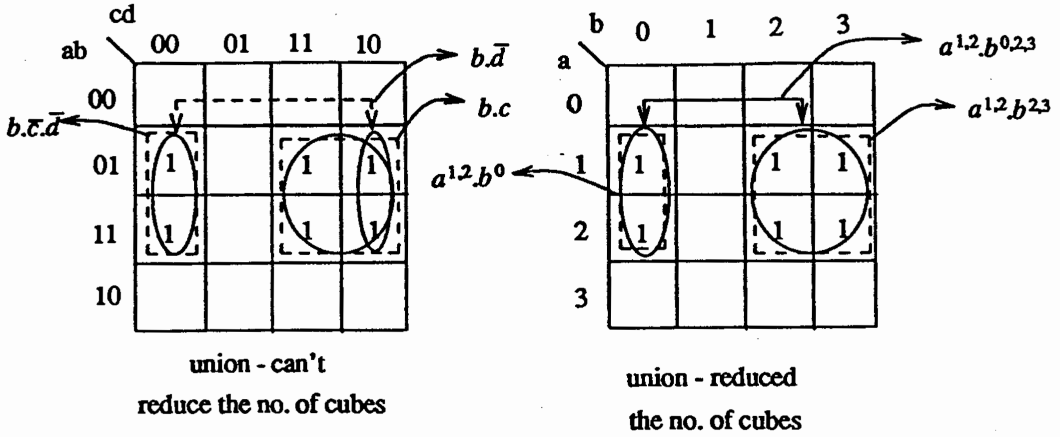


Figure 10. Examples of cube union.

Cube Intersection

The *cube intersection* is the set of all vertices covered by both cubes. As in sets, the intersection of two cubes can be calculated by the bitwise logical AND of the cube notation of the two cubes. For functions represented by multiple cubes, the intersection can be obtained by applying the distributive law of the intersection over the union of the cubes in the cover. The intersection of a function F = { $f_1, f_2, ..., f_n$ } and a cube g_j is:

$$F \cap g_j = \bigcup_{i=1}^{n} f_i \cap g_j$$

and the intersection of two functions F and G = { $g_1, g_2, ..., g_m$ } is:

$$F \cap G = \bigcup_{j=1}^{m} (\bigcup_{i=1}^{n} f_i \cap g_j)$$

Cube Inversion or Complement

The same definition used in set theory and multi-valued variables applies to inversion of cubes. The inversion of a cube array is the set of all vertices in the hyperspace that are not covered by the original array of cubes. The inversion can be considered a special case of the sharp operation that will be studied next. Figure 11 illustrates the inversion of a cube. The array of the cubes represented by the rectangles is the inversion of the covering composed of the reticules marked by "1" and representable by the array of cubes represented by circles.

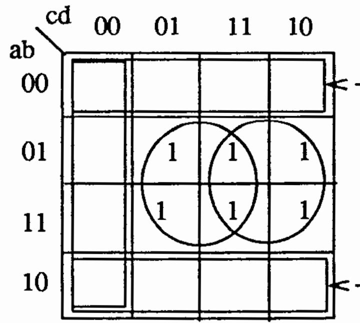


Figure 11. Inversion of functions.

Sharp or Relative Complement

The result of the sharp operation between two functions $F_1 \# F_2$ is another function F_3 that is true only when a vertex is true for F_1 but is not true for F_2. The inversion is a particular case of sharp when F_1 is the universe. The sharp operation is equivalent to a subtraction of minterms.

The sharp of two cubes f and g, $f \# g$ is the array of cubes, F, that is the covering of all the minterms of the first cube that do not intersect with the second cube. Depending on the cubes used in F to represent the results, sharp and inversion are classified into two flavors. In *Disjoint sharp,* the function is expressed as an array of disjoint cubes, meaning the two by two (cube against cube) intersection of the cubes of the result is empty. In the "normal" sharp the result is represented with the minimum number of cubes that cover the function and that are not covered by other cube that is also covered by the function. These cubes are called *prime implicants.* A prime implicant is the largest cube that implies the function and covers the same vertices. Equation (5) generates a list of cubes that cover the vertices covered by the cube f but not covered by the cube g, executing the "normal" sharp of the two cubes. Equation (5) also shows how much harder it is to execute the set difference with the requirement that the resultant set be described with an array of cubes.

$$f_i \# g_j = \bigcup_{k=1}^{l} x_1^f x_2^f \cdots x_k^{f \cap \bar{g}} \cdots x_l^f \tag{5}$$

Figure 12 shows one example of a disjoint sharp and a sharp. The cube represented by the rectangle drawn with a solid line is the cube being sharped, the rectangle drawn with the broken line is the cube being "subtracted" and the ellipses are the resultant cubes.

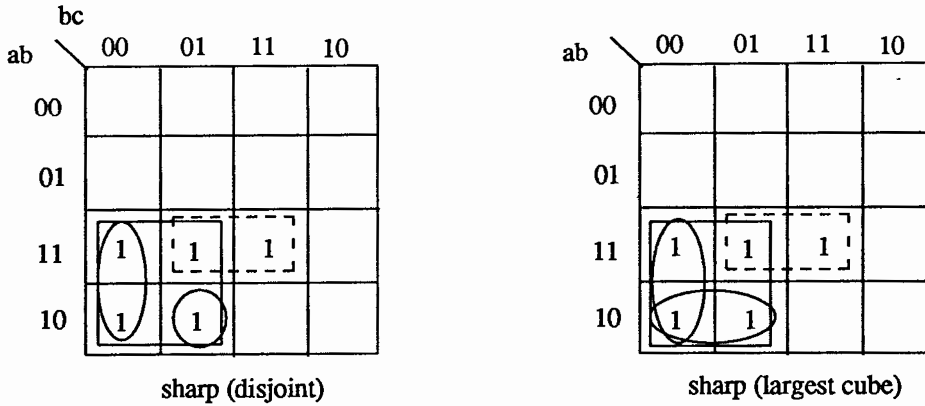


Figure 12. Sharp operation.

Two properties are used in sequential computers to reduce the calculations necessary to execute the sharp of cube against cube: 1) the result of the sharp product of two cubes will be the empty cube if g covers f; 2) the result will be $F = f$ if the cubes are disjoint (do not overlap). A test of these conditions can branch around the execution of a sharp algorithm to avoid the calculation of equation (5).

The sharp product of one array of cubes against one cube, F # g_j, where F = $\{f_1, f_2, ..., f_n\}$, is the union of the results of the sharp product of $f_i \# g_j$

$$F \# g_j = \bigcup_{i=1}^{n} f_i \# g_j$$

The sharp product of one array of cubes against another array of cubes is the recursive application of the sharp product of each cube of the second array to the result of the sharp product of the first array with one cube.

$$F \# G = F - G = F \cap \overline{G} = ((..(F \# g_1) \# g_2) \cdots \# g_m)$$

This is equivalent to the intersection of the sharp products of the first array and each cube of the second array.

$$F \# G = F - G = F \cap \overline{G} = \bigcap_{j=1}^{m} F \# g_j$$

Consensus or Star Product

Consensus is not an essential operation for cube calculus, but life would be more difficult without it. The consensus is used to generate new cubes from an existing array of cubes that still imply the same vertices implied by the array of cubes. The cubes generated by consensus can be used

to represent the same function with a different array of cubes in the process of minimization of the representation of a function and to reduce the number of cubes in the array and still cover the same vertices.

The *consensus* (asymmetric consensus) of the cubes f and g is the array with the prime implicants of f $\cup$ g that cover at least one vertex covered only by f, one vertex covered only by g, and the intersection of f and g. The formula used to calculate the consensus or the star product of two cubes is:

$$f*g = \bigcup_{i=1}^{i=l} x_1^{f_1 \cap g_1} \,.\, x_2^{f_2 \cap g_2} \cdots x_i^{f_i \cup g_i} \ldots x_l^{f_l \cap g_l}$$

Where x_i are the MVVs that compose the cubes.

Figure 13 shows two examples of consensus. In this figure, the stars (*) indicate vertices covered by the first cube and ampersands (@) the vertices covered by the second cube. The resultant cubes are represented by rectangles or ellipses.

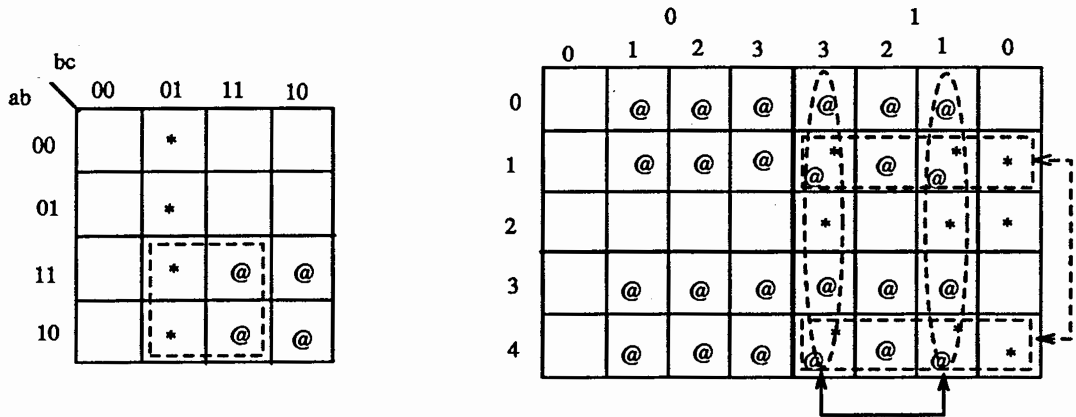


Figure 13. Two examples of cube consensus.

If the cubes f and g overlap as in the examples in the Figure 13, consensus generates a list of cubes with minterms of f and g that are candidates to cover or be covered by other cubes. If the intersection of the MVVs that compose the cubes is empty for more than one MVV, the result of the consensus is an array of empty cubes. The number of intersections of MVVs that are empty is used as a measure of the relative position between two cubes and is called the *distance* between the two cubes. Cubes of distance zero overlap, and cubes of distances one or larger do not overlap.

USING CAMS FOR CUBE CALCULUS

The key reasons to use CAM-based computers are to:

(1) Filter the data that is transferred between memory and the CPU.

(2) Increase the parallelism of execution of programs.

(3) Provide additional operations that improve the performance of the computer in non-numeric applications.

(4) Provide means to rethink algorithms and improve the execution of existing ones.

Beginning with filtering, all these points are illustrated with cube calculus applications.

Processing of cube calculus operations can be greatly reduced by the application of properties of cubes. For example, if the distance between the cube f and the cube g is greater than one, the result of consensus f*g is the empty cube. The intersection of cubes of distance one or larger is also the empty cube. Likewise, the result of the sharp product $f_i \# g_j$ is the empty cube if the cube g_j covers the cube f_i. And, $f_i \# g_j$ is the cube f_i if f_i and g_j are of distance one or larger. With all cubes stored in CAM using positional representation, complex searches can split the cubes by distance and transfer to the host processor only those cubes that will generate resultant cubes, reducing the data traffic between the memory and the host CPU. The use of CAMs reduces the number of the pairwise operations between cubes to the execution of only the pairs of cubes that generate non-empty cubes. This eases the requirements on communication between the host CPU and memory and on the processing of the host CPU.

For the sharp operation, the cubes must be split into overlapping cubes (cubes of distance 0), disjoint cubes (distance 1 or larger), and cubes covered by another cube. These relations can be tested by comparing the representations of the cubes in positional notation. A cube c_i covers another cube c_j if the values of all variables of c_i cover the variables of the cube c_j. This is equivalent to the representation of c_i not having a 0 in a position that the representation of c_j has a 1. A single search for 0's in the positions c_i has 0's will mismatch all cubes stored in the CAM that are not covered by c_i. Figure 14 shows an example of how to use CAMs to determine the covering relationship among cubes. A

search for the pattern ??.0?.??.?0.?? matches only cubes that are covered by the cube c_1 and is performed in parallel to The cubes c_2 through c_5.

		HIT	
c_1	11.01.11.10.11		
	??.0?.??.?0.??		search pattern
c_2	10.01.11.10.10	1	matches, c_1 covers c_2
c_3	11.01.01.10.11	1	matches, c_1 covers c_3
c_4	11.10.01.10.11	0	mismatches, c_1 does not cover c_4
c_5	11.01.01.11.01	0	mismatches, c_1 does not cover c_5

Figure 14. Determining the covering relation among cubes with CAMs.

Two cubes c_i and c_j overlap if no variables are disjoint. In positional notation this can be identified when, for each variable, there is at least one '1' in the same position for both cubes. A search, then, for 0's where the variable is 1 will match only on cubes that are disjoint for this variable. The product of the search for all variables tests whether two cubes overlap. Figure 15 illustrates how to determine whether the cubes c_2 through c_6 overlap the cube c_1. If two cubes are disjoint in any variable, they are disjoint cubes. Each search pattern tests if the cubes in memory and c_1 are disjoint in one MVV of the cube. The first, third and fifth patterns do not need to be applied because c_1 is full for these variables and a match (HIT) in at least one position is guaranteed. Only c_2, c_3 and c_5 mismatch all searches and, therefore, overlap with c_1.

11.01.11.10.11	c_1
??.??.??.??.??	search pattern for the first variable (no need to search)
??.?0.??.??.??	search pattern for the second variable
??.??.??.??.??	search pattern for the third variable (no need to search)
??.??.??.0?.??	search pattern for the fourth variable
??.??.??.??.??	search pattern for the fifth variable (no need to search)
10.01.11.10.10	c_2 mismatches both searches, overlaps
11.01.01.10.11	c_3 mismatches both searches, overlaps
11.10.01.10.11	c_4 matches second variable search, do not overlap
11.01.01.11.01	c_5 mismatches both searches, overlaps
10.11.01.01.11	c_6 matches fourth variable search, do not overlap

Figure 15. Testing if cubes overlap using CAMs.

Besides reducing the number of cubes that have to be transferred to the host CPU for processing, the CAM can also control the order in which the cubes will be passed to the host CPU. Algorithms that use heuristics to reduce the amount of processing can use this sorting property of CAMs to reduce

the overhead of the heuristics. For example, the cubes can be sorted by the inner product of the cube representation weighted by the number of cubes in the cover that have 1's for that value. CAMs can select all cubes with 1's in any position and count them to find out the weights, add the weights to an accumulator field of all matching words, and retrieve the cubes sorted by the largest accumulator field. Each of these operations would be performed in parallel.

In the tasks described above, the CAM was used in the "traditional" applications of filtering and sorting data. CAMs that have partial access, preferably multiple partial access, can execute some of the simple logic operations that would be executed by the host CPU. Notice that the CAM is executing "real" processing, the data retrieved from the CAM is different from the data originally stored.

Simple logic operations like bitwise AND and OR of cube and array can be executed by the CAM without a data transfer to the CPU. The following example of the intersection of a function and a cube illustrates these cube calculus operations executed by a CAM.

The operations necessary to execute the intersection between the cube $c_1 = \bar{a}.\bar{b}.c$ and the function F = $\{\bar{b}.\bar{c}.d, a.\bar{c}, \bar{a}, a.b.\bar{c}.d\}$ in positional notation are shown in Figure 16 that shows the state of the memory before and after the multiple partial write of the pattern 0?.0?.?0.??.

	before
$\bar{a}.\bar{b}.c$	01.01.10.11
$\bar{b}.\bar{c}.d$	11.01.01.10
$a.\bar{c}$	10.11.01.11
$\bar{a}$	01.11.11.11
$a.b.\bar{c}.d$	10.10.01.10
	after
empty	01.01.00.10
empty	00.01.00.11
$\bar{a}.\bar{b}.c$	01.01.10.11
empty	00.00.00.10

Figure 16. Memory before and after intersection.

Representing each array of cubes as a set using the set data structure presented in Chapter III, the union of two arrays is performed with the union operation over sets. The modification of the tags of the set data structure of the set operands with a partial write creates a new set that is the union of the two sets.

More complex cube calculus operations can also be executed in the CAM. Among these is the sharp operation. A naive algorithm for sharp is presented but, even with this simple algorithm the power of CAMs for cube calculus can be perceived. The algorithm presented begins with one copy of the function for each variable of the cube stored in memory. The intersection of each variable is executed with a multiple partial write and is followed by the removal of cubes with contradictions in any of its MVVs. This algorithm could be improved to store only copies of the function for literals that generate non-empty cubes. But, as an illustration, the naive algorithm is used. (Observe that F#g is equivalent to F-g which is also equivalent to $F \cap \bar{g}$)

As an example, the execution of sharp F#g, where $F = \{\bar{c}.d\,,\ \bar{a}.\bar{b}.c\,,\ b.c.\bar{d}\,,\ a.b.c.d\,\}$ and g = a.b is shown in the Figures 17 through 20 and Figure 21 gives the visualization of the example with Karnaugh maps. The memory is initially loaded with the cubes of F with one letter appended (tagged) to indicate which variable is going to be consumed (Figure 17). Notice that because the cube g does not contain the literals *c* and *d*, they generate only empty cubes. Each copy of F is selected sequentially according to the variable being processed and the corresponding intersection is executed. The sequence of searches and multiple partial write patterns is shown in Figure 18. The state of the memory after the processing is shown in Figure 19. Figure 20 shows the state of the memory after the garbage collection of the empty cubes and the removal of redundant cubes.

The union, intersection and sharp operations form a basis that can be used to build any cube calculus operation. The next sections will show applications that put to good use cube calculus and the efficiency of implementation of cube calculus operations with CAMs.

LOGIC MINIMIZATION AND SYNTHESIS OF BOOLEAN FUNCTIONS

The goal for this section is the minimization and synthesis of a Boolean function so that the mapping of its input to its output uses a minimum of hardware. Practical minimization problems work with multiple output and incompletely specified functions of rarely more than 100 binary inputs and up to 100 binary outputs. The cube calculus operations described so far address only completely specified single output functions. With sufficient memory size, the cube calculus model can be easily extended to represent and manipulate the kind of Boolean functions found in practical problems.

$\bar{c}.d$	11.11.01.10-a
$\bar{a}.\bar{b}.c$	01.01.10.11-a
$b.c.\bar{d}$	11.10.10.01-a
$a.b.c.d$	10.10.10.10-a
	11.11.01.10-b
	01.01.10.11-b
	11.10.10.01-b
	10.10.10.10-b
	11.11.01.10-c
	01.01.10.11-c
	11.10.10.01-c
	10.10.10.10-c
	11.11.01.10-d
	01.01.10.11-d
	11.10.10.01-d
	10.10.10.10-d

Figure 17. State of the memory before processing.

search	??.??.??.??-a
intersection	0?.??.??.??-*
search	??.??.??.??-b
intersection	??.0?.??.??-*
search	??.??.??.??-c
intersection	??.??.00.??-*
search	??.??.??.??-d
intersection	??.??.??.00-*

Figure 18. Patterns used for the generation of resultant cubes.

01.11.01.10-a
01.01.10.11-a
01.10.10.01-a
00.10.10.10-a

11.01.01.10-b
01.01.10.11-b
11.00.10.01-b
10.00.10.10-b

11.11.00.10-c
01.01.00.11-c
11.10.00.01-c
10.10.00.10-c

11.11.01.00-d
01.01.10.00-d
11.10.10.00-d
10.10.10.00-d

Figure 19. Memory after the sharp.

01.11.01.10-a
01.01.10.11-a
01.10.10.01-a
11.01.01.10-b

Figure 20. Results after the removal of empty cubes.

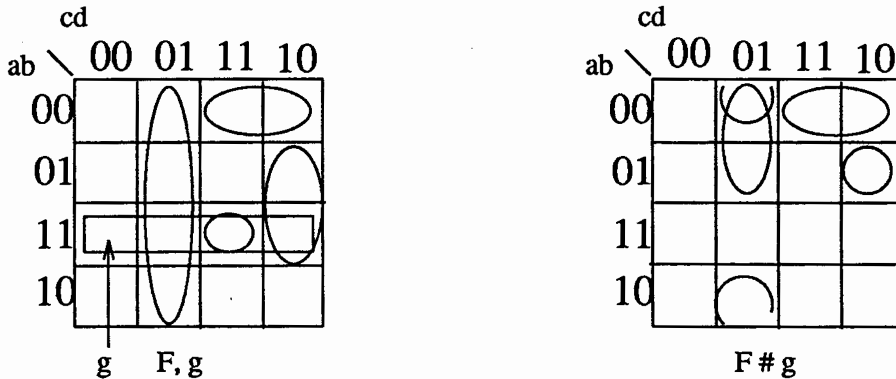


Figure 21. Karnaugh map of F and g used in the example of sharp.

Cubes are still used to represent logic functions but, the vertices of a cube in the representation of an *incompletely specified multiple output function (ISMO)* can evaluate to true (1) for one output, the value false (0) for another, or be either one (X) for yet another output. The execution of the presented cube calculus operations using CAMs can be extended to ISMO functions and preserve the

execution parallelism.

To support ISMO functions the cube notation has to be extended with an *output tag* that flags the value of a cube for each output. This output tag is identical to the tag field presented in Chapter III for set manipulations. The cube calculus operations over ISMO functions manipulate the output tag of the operands and generate resultant cubes with possibly modified output tags.

The complexity of extending the cube calculus operations to ISMO functions is because a common set of cubes is used to describe all outputs. A product or minterm can belong to one of three sets for each output. The set of minterms that evaluate to 0 (OFF-set), the set of minterms that evaluate to 1 (ON-set), and the set of minterms that can evaluate to either 0 or 1 (DC-set). In the representation used for the examples presented, the functions are described by their ON-sets and DC-sets. The tag field of the set data structure must be extended to support the "don't care (X)" state of the output. By treating the DC-set and the ON-set of each output as a bit of the output tag, and the minterms as elements of these sets, the scheme developed in Chapter III for the set data structure using tag fields can be directly applied to the output tag. The number of elements in the representation of an ISMO function is minimized by grouping minterms that have the same output tag into the same cube.

The representation of an ISMO function as an array of cubes maps directly to its PLA implementation. Each cube of the representation can be implemented as an AND gate. The array of cubes is equivalent to an OR gate of the cubes. The binary inputs of the function are mapped into MVVs with the use of decoders, or inverters for the simpler two-valued variables. The minimization of the cube representation of the function minimizes the PLA implementation of the function.

Extension of Cube Calculus Operations for ISMO Functions

Of the basic operations of cube calculus, union, intersection, and sharp, the union operation is the simplest to extend for ISMO functions. The result of the union of two arrays of cubes is a new array with the cubes of both operands. The sharp product is the most complex to extend. The scheme to extend the cube calculus operations with the output tag will be illustrated by an example of the intersection of two cubes of ISMO functions of 2 two-valued inputs and 9 binary outputs. The example is shown in Figure 22. The bits of the output tag are independent from each other. Each output can be

visualized as a function of a single output that will be processed in parallel. Similarly, when generating the result of an intersection, each output is generated independently using a parallel bit-vector logical operation.

cube	inputs	outputs		
c_1	01.11	000	111	XXX
c_2	11.10	01X	01X	01X
c_r	01.10	000	01X	01X

Figure 22. Sharp of two cubes of a multiple output function.

The intersection operation can be parallelized to execute the intersection of c_1 against the cubes c_2 to c_n by generating the resultant cubes of the intersection as presented for the single output case and processing the output tags with a sequence of searches and partial writes. This scheme is presented using the patterns of the cube c_1 in Figure 22. The operation executed is a multiple partial write with the pattern "0?.??000??????, where the first four bits are the inputs and the last nine bits are the outputs of the function. The cube c_2 is overwritten by the intersection. The multiple partial write transforms the operand cubes stored in the CAM into the cube that is the intersection (input field), and sets the bits of output tag of the resultant cubes to the correct values that enable the resultant cube to be shared by all outputs. The outputs that are 0 in c_1 must also be 0 in the intersection cube (first 3 bits of the output), the tag of the intersection of outputs that are 1 in c_1 are determined by the outputs of the cubes stored in memory (the fourth, fifth and sixth bits of the output tag), and the intersection of outputs that are don't cares in c_1 are also determined by the outputs of the operand cube stored in memory (seventh, eight and ninth bits of the output tag).

In software, the sharp of a cube on a multiple output function requires three nested loops (see Figure 23) The outer loop sharps the cube against the arrays representing each of the outputs. The second loop sharps the cube against the cubes in each array and the inner loop runs over the variables to generate the resultant cubes.

In CAM-based and RAM-based computers, the outer loop is executed in parallel by the representation of the outputs with the output tag. However, CAM-based computers also parallelize the intermediate loop, number of cubes, which is by far the longest one. For example, the representation

of an ISMO function can have thousands of cubes. The performance improvement of the sharp product executed with the support of CAMs is proportional to this parallelization.

$F \# c_k$

FOR all output arrays F_i in function F
DO
    FOR all cubes f_j in F_i
    DO
        FOR all variables w
        DO $/f_j \# c_k/$

$$x_1^{f} . x_2^{f} \cdots x_w^{f \cap \bar{g}} \cdots x_m^{f}$$

        END
    END
END

Figure 23. Algorithm to sharp a cube out of a multiple output function.

Table I lists the execution times of the sharp product measured with the UNIX utility gprof [GRAHAM82] of the three functions labeled F_1, F_2 and F_3 using the program ESPRESSO [BAKER88] on a SUN Sparc station 1 iPC with 8 Mbytes of RAM. Each of these functions have 26 binary inputs and 46 binary outputs, represented by an array of 301 cubes. F_1 is the benchmark *bca* provided in the OCT distribution and F_2 and F_3 were created by modifying the cubes of F_1 at random.

TABLE I

EXECUTION TIME OF THE SHARP PRODUCT
ON A RAM-BASED COMPUTER

	CPU [s]
$F_1 \# F_2$	434.58
$F_2 \# F_1$	440.90
$F_1 \# F_3$	10094.35
$F_3 \# F_1$	25.38
$F_3 \# F_2$	32.46
$F_2 \# F_3$	33.35

Tables II, III and IV profile, again using gprof, the execution of the sharp products that are not dominated by reading the operand arrays of cubes in Table I. The function cv_sharp() forms the sharp product of two covers and calls cb_sharp() that forms the sharp product of a cube and a cover. The

function cb_sharp() calls a recursive formulation cb_recur_sharp(). The function cb_recur_sharp() calls the function sharp() that forms the sharp product of two cubes. Cb_recur_sharp() also calls cv_intersect() to find the intersection of the results of the function sharp(). The function sf_union() forms the union of the sharp results and also deletes repeated cubes and cubes that are covered by a single cube. The functions rm2_contain() rm2_equal() that are called by sf_union() and removes the redundant cubes from their already sorted input arrays of cubes.

TABLE II

EXECUTION PROFILE OF $F_1 \# F_2$

function	number of calls	self CPU time	
		[s]	[%]
cv_sharp	1	0.01	0.0
cb_sharp	301	0.04	0.0
cb_recur_sharp	180901	1.17	0.4
sharp	90601	0.44	0.1
cv_intersect	93300	2.53	0.5
sf_union	733	0.02	0.0
rm2_contain	1466	273.99	54.4

TABLE III

EXECUTION PROFILE OF $F_2 \# F_1$

function	number of calls	self CPU time	
		[s]	[%]
cv_sharp	1	0.00	0.0
cb_sharp	301	0.02	0.0
cb_recur_sharp	90732	1.54	0.3
sharp	90601	0.37	0.1
cv_intersect	90300	2.55	0.5
sf_union	733	0.05	0.0
rm2_contain	1466	272.91	53.4

TABLE IV

EXECUTION PROFILE OF F_1 # F_3

function	number of calls	self CPU time	
		[s]	[%]
cv_sharp	1	0.00	0.0
cb_sharp	301	0.00	0.0
cb_recur_sharp	301	1.31	0.0
sharp	90601	0.50	0.0
cv_intersect	90300	21.63	0.2
sf_union	20150	0.22	0.0
rm2_contain	40300	8342.87	78.2

The sharp product of two covers (ISMO functions) executed with CAMs execute the equivalent to the function sharp and the intersection of cube and covering(s) in parallel. The equivalent CAM functions used to substitute the functions cv_intersect(), cb_recur_sharp() and sharp() are called the same number of times cb_sharp() is called. The function rm2_contain() that accounts for over 50% of the processing time is also executed in parallel. Furthermore, rm2_contain() requires sorted coverings as inputs. The cumulative cost of the function sf_sort() that sorts the coverings for the sharp product of Tables II, III and IV are shown in Table V. Due to the efficiency of sorting, and in many cases the lack of need of sorting in CAMs, the cost in time of sorting would be greatly reduced.

TABLE V

CUMMULATIVE COST OF SORTING CUBES

operation	CPU time [s]	CPU time [%]
$F_1 \# F_2$	58.50	13.9
$F_2 \# F_1$	61.60	14.3
$F_1 \# F_3$	267.41	2.8

Another point in favor of CAMs is that they provide a trade-off between execution time and hardware that RAMs cannot provide. The removal of redundant cubes is essential to the performance of the sharp product on RAM-based computers because the processing time is proportional to the number of cubes. Subject to memory constraints, the sharp algorithm for CAM-based computers is

almost independent from the number of cubes in the intermediate representation of the covers. Therefore, the removal of redundant cubes can be executed once, just prior to presenting the final results with small penalty in performance. More RAM only adds storage space , which is essential to execute larger problems, but without performance gain. More CAMs in the system adds storage space and parallel processors that can take advantage of under used parallelism.

Perhaps the most impressive point in favor of CAM-based computers is that the RAM-based performance is measured in CPU time and the performance of the CAM-based sharp is measured in CAM cycles. The CAM has transferred the processing load of the CPU.

To evaluate the performance of the sharp product in a CAM-based computer, the execution time of the sharp product $F_1 \# F_3$ in the recently reported CAM-based computer IXM2 [HIGUCHI91] will be estimated. The sharp product will require the cube operations listed in Figure 24. The cube against cube sharp products are executed in parallel for all 301 cubes of F_1 and sequentially for the 301 cubes of F_3. Each sharp call requires 26 multiple write cycles, one for each input. The intersections are executed in parallel over the sharp results and require a worst case number of calls of 301 * 26 in the case of each sharp product creates a maximum number of resultant cubes. The results of the intersections are composed into a single array. A pessimistic number of cycles is estimated for the removal of redundant cubes based on the results of the profiling. Since each cube is stored in 4 words, pessimistically, the operations have to be repeated for each of the words.

301 sharp calls * 26 MVVs cycles	7826
301 cv_intersect calls * 26 MVVs cycles	7826
301 sf_union calls cycles	301
40300 rm2_contain cycles	40300
total number of CAM cycles	4 * 56253

Figure 24. Estimate of the execution cycles for sharp in a CAM.

The IXM2 executes each the equivalent to the CAM cycle listed in Figure 24 in 18 μs. The total execution time for the sharp product $F_1 \# F_3$ is 18 μs x 4 x 56253 ≈ 4.0 s. This impressive performance improvement ofi *over three orders of magnitude* is achieved if the IXM2 can store the largest intermediate representation of the operand functions. The IXM2 has a storage capacity of 256 Kwords of 40 bits. For the examples used, each cube requires 4 words and the memory required would be 4

words x 310 cubes x 26 MVVs = 31,304 words. Operations that require more than 256 Kwords will demonstrate a smaller gain in performance.

The estimated improvement, of course, will be smaller because only the performance of the calculations is improved. The time required to read the operand covers and for the output of the results is nearly constant. Reading the operand covers took approximately 5% of CPU time of the longest example, and little less than 15% on the other two. The percentage of time spent on setting up the operation of examples with very short sharp execution times may be as high as 50% for a RAM-based computer. In these cases, the performance improvement of the CAM-based computer is limited by the operations not directly related to the calculations.

Another Representation of Logic Functions and CAMS

The representation of functions with cubes presented is suitable for PLA implementations. The factored form is a representation that is more suitable for multi-level minimization of logic functions. Factored forms and multi-level minimization are based on the papers by Brayton [BRAYTON82, BRAYTON87]

The *factored form* is defined recursively: 1) a literal is a factored form; 2) a sum of factored forms is a factored form; 3) a product of factored forms is a factored form. This definition can be generalized to include other operations such as the exclusive-or, arithmetic sum, arithmetic product, etc. For simplicity, the discussion will be restricted to the logical sum and product of factored forms.

Typically, the data structured used to represent factored forms is the list. For example, the cubes of binary literals are represented as lists of literals. The cube, or product of literals, a.b.d, is represented by the list {a, b, d}. When one literal appears in a list in its negated and non-negated form, the list represents an empty cube. This representation again works with symbols that are suitable for algebraic manipulations.

With two lists representing the product of their elements, appending the two lists is equivalent to the AND operation or intersection of algebraic representations. For example, appending the lists that represent the cubes "a.b.d" and "c.d", append({a,b,d},{c,d}), results in the list {a,b,c,d} which represents the cube "a.b.c.d". Analogously, when the list represents the sum of its elements, appending

two lists represents the union or OR operation

Although originally the data structure used for factored forms was the list, the ideal data structure for this representation is the set data structure because the list {a, b, c} represents the same cube as the list {c, a, b}. This representation of functions with lists is one example of the use of a more complex data structure because of the low support of the set data structure and the sequential nature of RAM-based computers. Because of the sequential nature of RAM, sorting of the elements is almost a must to achieve acceptable performance.

The logic minimization with factored forms performs algebraic manipulation to identify common sub-expressions in different outputs, and functions that are by themselves subexpressions of other functions. The cost of the implementation can then be shared by the functions that share the common sub-expression. Brayton [BRAYTON87] presented the concept of kernels, co-kernels and support of a function to help find sub-expressions that can be shared by two functions. An informal description of these concepts is given below.

A *kernel* of a function is a divisor of the function, or sub-expression of a function that cannot be represented by a single cube (product of literals) and cannot be divided evenly by a single cube. A *co-kernel, C, of a kernel k* is a single cube divisor of a function f such as f/C = k.f whose quotient is a kernel. Kernels and co-kernels are good candidates for sub-expressions. For example, the product c.d is a co-kernel of the function f = a.b.c.d + c.d.e = (a.b + c.d).(d.e) that corresponds to the kernel (a.b + c.d).

The *support* of a function is the set of all literals used to describe the function. If the function is the list representing the sum of its elements $\{a\bar{b}, c\bar{d}, \bar{a}d\}$, the support of the function is $\{a, \bar{a}, \bar{b}, c, d, \bar{d}\}$.

The support of the function is used to construct cubes that divide the function evenly and is used to find the kernels and co-kernels of a function. Two functions, *f* and *g*, have a common multiple-cube divisor (a common kernel) if the intersection of the set of kernels of f and g is more than one cube. The algorithm of the multi-level minimization program MIS [OCT] searches for common divisors by generating the set of kernels of each function and forming the intersection of the sets. If the result of

the intersections is either the empty cube or a single cube, the program looks only for single cube divisors. Brayton [BRAYTON87] observed that since most of the possible 2^f intersections of sub-sets of f are empty, the generation of the kernels of an expression by the scheme presented can be extremely inefficient. To avoid the calculation of the intersections, MIS transforms the calculation of intersections into finding the kernels of another function with the overhead of the transformation and the reverse transformation.

Again, the different performance relation of the CAM significantly changes the complexity of execution and development of existing algorithms. In this specific case, it simultaneously simplifies and improves the performance of the algorithm. CAMs can calculate the bitwise intersection in parallel and without the CPU. The transformation and anti-transformation of the problem is then unnecessary, simplifying the algorithm.

The set of literals of a cube and the sub-expressions in a function can be represented with the set data structure described in Chapter III, with each literal as an element of the cube, and each cube as an element of a sub-expression, and so on.

The CAM can be used to find out which literals of the support divide the function evenly by searching, one literal at a time, and finding those that match all cubes in the function. The literals that do not match all lists, match a partial list of the cubes in the function and can be used to generate the kernels of the function. The kernels can also be found by dividing the function by the co-kernels.

An important process of algebraic manipulation of functions is the factorization of the function. This task is performed with the algebraic division of the functions. Due to the high cost of the division and the large number of functions found in minimization problems, some guidelines to filter the divisions were developed. Devising heuristics and discovering properties to reduce the amount of computation is a very important technique in logic minimization and synthesis since it enhances the performance and enables the solution of problems that would be too big to solve otherwise.

Filtering is one of the strong points of CAMs. Brayton [BRAYTON87] showed that g_j is not an algebraic divisor of f_i (i.e., $f_i / g_j = 0$) if:

(1) g_j contains a literal not in f_i

(2) g_j has more terms than f_i

(3) for any literal, the number of appearances in g_j exceeds that in f_i.

(4) if for any literal, the number of appearances in g_j equals that in f_i then (f_i/g_j) is at most a single cube.

These properties are easily mapped into a sequence of searches in the CAM, enhancing the performance of logic minimization algorithms at low cost.

The logic minimization and synthesis of Boolean functions is typical of the kind of application that would be improved by the use of CAM-based computers. The multi-level minimization scheme presented does not use its ideal data structure, the set, and a more complex algorithm had to be developed due to the shortcomings of RAM. Some of the processing can be transferred to CAM and executed in parallel. By executing searches on the data stored in the memory the amount of processing and data transferred from memory to CPU is reduced.

CUBE CALCULUS AND RESOLUTION

Artificial intelligence is one of the non-numeric applications with low performance in RAM-based computers that is expected to continue to grow in importance in the near future. A key factor that makes artificial intelligence programs so computational intensive and suitable for associative processing is that they are often non-deterministic. The amount of computation for AI poses requirements in performance that a single processor system cannot handle in reasonable time. Higher parallelism and concurrency are required to execute these programs at a rate that will make them find realistic applications. A hardware implementation of fine-grained parallelism at a lower level as the content-addressable memory is then suggested.

Of the representations of knowledge used in AI, predicate calculus is a more general representation but it is also less succinct than specialized languages. The retrieval of information from databases that use predicate calculus generally takes longer because the information is not always directly stored in the database. The information has to be deduced, inferred, from the database. The applications that

require predicate calculus or clausal forms to describe knowledge because of its expressivity, have an even higher processing requirements than specialized languages.

The manipulation of declarative knowledge for artificial intelligence in *clausal form* is done with the *resolution* principle. A *clause* is equivalent to a set of literals representing their disjunction. Simply stated, the resolution principle is: If a literal of one clause is false the remainder of the clause has to be true. Finding two literals in different clauses that cannot be true at the same time means that at least one of the remainder of those two clauses is true and we can join both remainders in a new clause assured that the new clause is true.

An illustration of resolution with unification is shown in the Figure 25. No conclusion can be drawn from the clauses marked 1 and 2 until the unifier γ is applied to clause 2 to substitute the free variables by constants, creating the clause 3. The application of resolution on the clauses 2 and 3 concludes that the clause 4 is also true.

1) $\{P(x), Q(x, y)\}$
2) $\{\neg P(A), R(B, z)\}$
$\gamma=(x/A, y/z)$
3) $\{P(A), Q(A, y)\}$
4) $\{R(B, z), Q(A, y)\}$

Figure 25. Example of resolution and unification.

Previously, a cube was interpreted as a product. The formulation of the consensus of two cubes is repeated with the notation used in logic. The consensus of two cubes will have a single cube result that covers both cubes if their distance is one:

$$p \wedge q \wedge r \ * \ \bar{p} \wedge r = q \wedge r$$

If the distance is larger than one, the result cube is the empty cube:

$$p \wedge q \wedge \bar{r} \ * \ \bar{p} \wedge r = q \wedge r \wedge \bar{r} = \emptyset$$

If the cube is interpreted as a sum of literals, each cube is equivalent to a clause, and a function is equivalent to a database. The consensus operation is the dual to the resolution principle.

$$p \vee q \vee r \ * \ \bar{p} \vee r = q \vee r$$

$$p \vee q \vee \bar{r} \ * \ \bar{p} \vee r = q \vee r \vee \bar{r} = 1$$

The Figure 26 shows a simple example of resolution from [ULUG87] and its graphic representation. The example shows that resolution just states:

If (p OR q) is true and ((NOT q) or r) is also true, it is correct to infer that (p OR r) is true also.

In the example:

If an object A is "not round or large" and at the same time it is "not large or do not have a hole", it is safe to assume that the object is also "not round or does not have a hole". It is also safe to assume that an object is "large or not large", but this is a trivial conclusion.

All "mathematics" of cube calculus and CAM-based algorithms shown in this chapter can be applied to artificial intelligence. Testing whether a sentence is true is equivalent to testing the covering of the array of cubes that represents the sentence by the knowledge database. Finding all objects that match some relation is equivalent to the intersection between the database and the sentence that

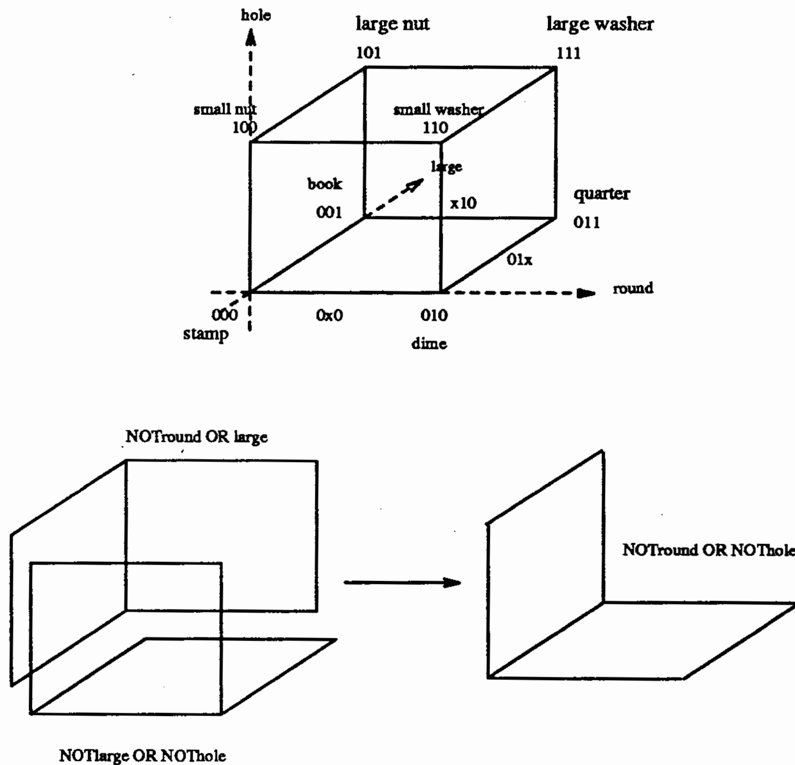


Figure 26. Graphical example of the resolution principle.

describes the relation, and so on.

IMAGE PROCESSING AND CUBE CALCULUS

Oldfield [OLDFIELD87] proposed the use of CAMs to process images subdivided in homogeneous quadrants. In that paper, an update of the image does not require the visiting of all quadrants because CAMs can search and identify which quadrants have to be modified. Using functional memories (trits) it is easy to clip an image or count the quadrants that have a certain color (property). The example from [OLDFIELD87] is repeated here as an illustration. The Figure 29 shows the result of overwriting the non-white quadrants of the update image shown in Figure 28 over the complex image shown in Figure 27. The images are stored in CAM as non-overlaping cubes that represent each quadrant. The entries in the memory are show alongside with each figure.

Only the non-white quadrants of the update data have to be processed. In this example, only the entries l, n and q of Figure 28 have to be processed. The entry l of the update image matches only the quadrant b of the image. The quadrant b and the quadrant l are of the same size and a simple update of the color of b updates the image. The quadrant specified by the entry n covers the quadrants represented by the entries d, e, f and g. These four quadrants can be absorbed , or sharped, from the image and substituted by a new entry d with the color of entry n. The quadrant of the update image q matches the quadrant j of the image. The quadrant j is sharped by the quadrant q generating the new

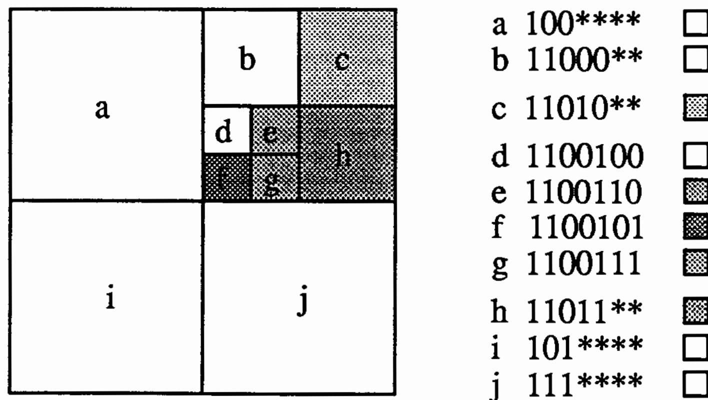


Figure 27. Sample image and CAM entries.

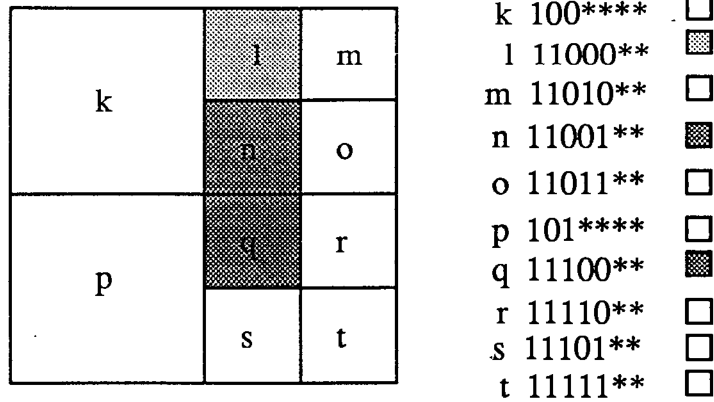


Figure 28. Update image and CAM entries.

quadrants represented by the new entries f, g and j.

The basic theory used is the disjoint sharp of cubes using a quad-tree data structure and using cubes to represent the quadrants. Clipping of one image A from one image B is the equivalent to the operation (A # B) ∪ B. This operation is executed efficiently using CAMs.

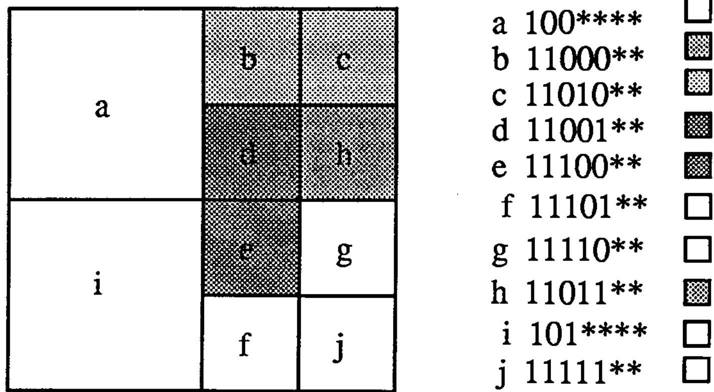


Figure 29. Final image and CAM entries.

CHAPTER V

CONCLUSIONS AND FUTURE WORK

This work analyzed the performance improvement that results from the addition of associativity to the set data structure. The analysis showed that the content-addressable memory implements associativity and enhances the parallelism of algorithms. The communication bandwidth requirement between the processor and the memory is smaller in a CAM-based computer. Therefore, a CAM-based computer has higher performance than RAM-based computer for the same system bus communication bandwidth.

No inherent fault was discovered in the associative processing model implemented with content-addressable memories, but only recently has the integration density permitted the integration of CAM ICs large enough to find realistic applications, and the understanding of content-addressable memories matured to defining the role of the CAM in the system beyond that of a search engine. It is the extended capability of execution of logic operations and the support of data structures that enabled the CAM to assume a more participative role in processing. Probably the strongest reason for the small number of CAM-based systems is the RAM monopoly of the implementations of computer memories. The processors and software are optimized to work in conjunction with RAMs.

Associative processing is such a powerful model that special architectures were developed despite the high cost of the development of CAM ICs and dedicated software. The number and variety of applications of proposed application specific CAM-based systems demands the rethinking of the decision to concentrate research and development efforts to RAM-based computers. The prevailing single mindedness for memory integrated circuits was important in the development of computation but it is time to re-evaluate that decision in face of the wider range of applications found for computers. The technology to develop CAM-based computers is already available, as proved by the IXM2 [HIGUCHI91] which is a CAM-based computer that presents high performance in applications for

which RAM-based computers have poor performance.

It is expected that CAM-based computers will have worse performance than RAM-based computers for some applications. The natural research topic that arises from this fact is the study of alternative architectures that blend the characteristics of RAM and CAM-based computers into an architecture that is more powerful than either one isolated. To decide how to blend both computational models, it is necessary to know the strengths of each model. Compared to the experience accumulated on RAM-based computers, the CAM-based computer is almost unknown.

The effects and applications of other data structures enhanced by associativity have to be studied. And the programming environment of the CAM-based computer must be better understood, the substitution of RAM with CAM changes the execution speed of essential tasks. For example, searching a table was always avoided by experienced programmers, The naive algorithm to sort a set of N values is of complexity of the order of N^2. By the development of sophisticated algorithms sorting can be executed in $N.log\ N$ time. Algorithms developed for systems that use CAM taking the naive approach execute the same task in O(N) time, by using the processing power of N comparator circuits in parallel. It is conceivable that there would be a larger use of interpreters in CAM based systems because compilers take longer to develop and are more difficult to modify than interpreters. Compilers are used because they transfer the table look-up, which implies searching and sorting, to compile time, speeding-up the execution of interpreted programs. Table look-ups are too expensive to be done while executing the program on-the-fly in RAM-based systems but not in CAM based computer systems. In the same way, programs that use more dynamic memory allocation should became more popular. Memory management with CAM is much simpler than memory management of dynamic memory allocation with RAM. Programs that use dynamic memory allocation and disorganize data will not be significantly slower than those that do not, because, in CAM-based computers, the overhead of memory management and allocation is small.

Parallel programs are not as sequential as the programs for sequential computers. CAM-based computer systems can take advantage of the fact that the sequence of execution of instructions in CAMs has to be explicitly stored to increase the parallelism. The *Linda* language for parallel programming is one example of a high-level programming language that uses the associative memory model.

For more information on the Linda language please refer to: [CARRIERO90, AHUJA88, CARRIERO88, LELER90, GELERNTER85, AHUJA86]. Ahuja [AHUJA88] showed that the implementation of the primitives of Linda with an associative memory implemented in hardware using hash coding yields significant gain in performance. Since CAMs have been shown to outperform hash coding, it is reasonable to expect similar or better gains with CAMs.

CAMs also support relational calculus which finds applications in artificial intelligence. Many architectures were proposed for relational databases that use content-addressable memories to improve performance [BLAIR89, RIBEIRO89, NG87, ROBINSON85, SHANKAR88, STORMON88, OLDFIELD86, OLDFIELD87]. And the literature on the use of CAMs for logic programming is extensive [BLAIR89, RIBEIRO89, NG87, ROBINSON85, SHANKAR88, STORMON88, OLDFIELD86, OLDFIELD87]. The papers of Yokota, [YOKOTA86] Woo [WOO85, O'KEEFE86] and Shobatake [SHOBATAKE86] do not use CAMs but could be extended to do so.

The research of the associative model should also include alternative CAM-based computer architectures. The dynamic dataflow computer (and the data-driven computer) uses the data dependencies to enforce the sequence of execution of instructions. This scheme of fine-grained parallelism yields high parallelism and the CAM is the ideal memory device to implement the matching unit of dataflow computers. For more information on dataflow computers please refer to: [YUBA90, AMAMIYA86, DAVIS82, IANNUCCI88, BUEHRER87, ARVIND82, DENNIS80, GAUDIOT86, GAJSKI82, GOSTELOW80, NIKHIL89, ARVIND83, TRELEAVEN82, GAUDIOT89]. The dataflow computer was proposed for the execution of logic programs and the implementation of expert systems to make use of the inherent parallelism of these applications [BIC84, ROKEY85, MURAKAMI83]. High parallelism and artificial intelligence are two of the characteristics of the "fifth generation computer" [MOTO-OKA83, TRELEAVEN83] which was proposed to address the issues raised in Chapter I. This research leads to the conclusion that CAMs will be a building block of this computer.

This thesis has not completely answered all the questions that it set out to answer. And the answers found raised more questions. But we believe that the questions raised are more important and in a higher level than the original ones.

REFERENCES

Adams, Stuart J., Mary Jane Irwin, and Robert M. Owens, "A Parallel, General Purpose CAM Architecture," in *Advanced Research in VLSI*, pp. 51-71, MIT Press, 1986.

Advanced Micro Devices, "Am99C10 256x48 Content Addressable Memory," *Commercial Release*, November 16, 1988.

Ahuja, Sudhir, Nicholas Carriero, and David Gelernter, "Linda and Friends," *Computer*, vol. 19, no. 8, pp. 26-34, August 1986.

Ahuja, Sudhir, Nicholas J Carriero, David H Gelernter, and Venkatesh Krishnaswamy, "Matching Language and Hardware for Parallel Computation in the Linda Machine," *IEEE Transactions on Computers*, vol. 37, no. 8, pp. 921-929, August 1988.

Amamiya, Makoto, Masaru Takesue, Ryuzo Hasegawa, and Hirohide Mikami, "Implementation and Evaluation of a List-Processing-Oriented Data Flow Machine," in *13th International Symposium on Computer Architecture*, pp. 10-19, 1986.

Amsbury, Wayne, *Data Structures, from arrays to priority queues.*, Wadsworth Publishing Co., Belmont, CA, 1985.

Anderson, George A, "Multiple Match Resolvers: A New Design Method," *IEEE Transactions on Computers*, vol. C-23, no. 12, pp. 1317-1320, December 1974.

Arvind, and Kim P Gostelow, "The U-Interpreter," *Computer*, pp. 42-49, February 1982.

Arvind, and Robert A Iannucci, "A Critique of Multiprocessing von Neumann Style," in *10th International Symposium on Computer Architecture*, pp. 426-436, 1983.

Ashenhurst, Robert L, *The Decomposition of Switching Functions*, pp. 74-116, 1959.

Baker, Wendell, Jeff Burns, Wayne Chritopher, Shau-Lim Chow, David Harrison, Chuck Kring, Tom Laidig, Bill Lin, Rick McGeer, Peter Moore, Kurt Pires, Tom Quarles, Jim Reed, Richard Rudell, Carl Sechen, Russel Segal, Rick Spickelmier, Albert Wang, Robert K Brayton, A Richard Newton, and Alberto Sangiovanni-Vincentelli, "OCT Distribution 2.1," *Berkley University*, March 25, 1988.

Bennetts, R. G., *Design of testable logic circuits.*, Addison-Wesley Publishers Limited, 1984.

Bergh, Harald, Johan Eneland, and Lars-Erik Lundstrom, "A Fault-Tolerant Associative Memory with High-Speed Operation," *IEEE Journal of Solid-State Circuits*, vol. 25, no. 4, pp. 912-919, August 1990.

Bic, Lubomir, "Execution of Logic Programs on a Dataflow Architecture," *SIGARCH Newsleller*, vol. 12, no. 3, pp. 290-296, June 1984.

Blair, G. M. and P. B. Denyer, "Content addressability: an exercise in the semantic matching of hardware and software design," *IEE Proceedings, Pt. E*, vol. 136, no. 1, pp. 41-47, January 1989.

Blair, Gerard Miles, "A Content Addressable Memory with a Fault-Tolerant Mechanism," *IEEE Journal of Solid-State Circuits*, vol. sc-22, no. 4, pp. 614-616, August 1987.

Borgstrom, T H, M Ismail, and S B Bibyk, "Programmable current-mode neural network for implementation in analogue MOS VLSI," *IEE Proceedings*, vol. 137, Pt G, no. 2, pp. 175-183, April 1990.

Brayton, R K and Curt McMullen, "The Decomposition and Factorization of Boolean Expressions," in *Proceedings of ISCAS*, ed. IEEE, pp. 49-54, Rome, 1982.

Brayton, Robert K, Richard Rudell, Alberto Sangiovanni-Vincentelli, and Albert R Wang, "MIS: A Multiple-Level Logic Optimization System," *IEEE Transactions on Computer-Aided Design*, vol. CAD-6, no. 6, pp. 1062-1081, November 1987.

Browne, James C., "Parallel architectures for computer systems," *Physics Today*, pp. 28-35, May 1984.

Bruce D Shriver, *Computer*, vol. 21, no. 3, March 1988.

Buehrer, Richard and Kattamuri Ekanadham, "Incorporating Data Flow Ideas into von Neumann Processors for Parallel Execution," *IEEE Transactions on Computers*, vol. C-36, no. 12, pp. 1515-1521, December 1987.

Carpenter, Gail A and Stephen Grossberg, "A Massive Parallel Architecture for a Self-Organizing Neural Pattern Recognition Machine," in *Computer Vision, Graphics and Image Processing*, vol. 37, pp. 54-117, 1987.

Carriero, Nicholas and David Gelernter, *Applications Experience with Linda*, January 1988.

Carriero, Nicholas and David Gelernter, *How to Write Parallel Programs, a First Course*, The MIT Press, 1990.

Chae, Soo-ik, James T. Walker, Chong-cheng Fu, and R. Fabian Pease, "Content-Addressable Memory for VLSI Pattern Inspection," *IEEE Journal of Solid-State Circuits*, vol. 23, no. 1, pp. 74-78, February 1988.

Chisvin, Lawrence and R. James Duckworth, "Content-Addressable and Associative Memory: Alternatives to the Ubiquitous RAM," *Computer*, pp. 51-64, July 1989.

Clark, L. T. and R. O. Grondin, "Comparison of a Pipelined "Best Match" Content Addressable Memory with Neural Networks," in *IEEE International Symposium on Neural Networks*, ed. IEEE, vol. III, pp. 411-418, 1988.

Clark, Lawrence T. and Robert O. Grondin, "A Pipelined Associative Memory Implemented in VLSI," *IEEE Journal of Solid-State Circuits*, vol. 24, no. 1, pp. 28-34, February 1989.

Damarla, T. and M. Karpovsky, "Detection of stuck-at and bridging faults in Reed-Muller canonical (RMC) networks," *IEE Proceedings*, vol. 136, Pt. E, no. 5, pp. 430-433, September 1989.

Dasgupta, Subrata, *Computer architecture. A modern synthesis.*, 2: Advanced topics, John Wiley & Sons, 1989.

Daunicht, W J, "Control of manipulators by neural networks," *IEE Proceedings Pt.E*, vol. 136, no. 5, pp. 395-399, September 1989.

Davis, Alan L and Robert M Keller, "Data Flow Program Graphs," *Computer*, pp. 26-41, February 1982.

Dennis, Jack B, "Data Flow Supercomputers," *Computer*, pp. 48-56, November 1980.

Dietmeyer, D. L., *Logic Design of Digital Systems*, Allyn and Bacon, Boston, 1978. 2nd ed.

Duller, A. W. G., R. H. Storer, A. R. Thomson, E. L. Dagless, M. R. Pout, A. P. Marriot, and J. Goldfinch, "Design of an associative processor array," *IEE Proceedings*, vol. 136, Pt. E, no. 5, pp. 374-382, September 1989.

Eder, Elmar, *Properties of Substitutions and Unifications*, pp. 31-46, Academic Press, Inc., 1985.

Finnila, Charles A. and Hubert H. Love, Jr., "The Associative Linear Array Processor," *IEEE Transactions on Computers*, vol. C-26, no. 2, pp. 112-125, February 1977.

Foster, Caxton C., *Content Addressable Parallel Processors*, Van Nostrand Reinhold Company, New York, 1976.

Gajski, D D, D A Padua, and D J Kuck, "A Second Opinion on Data Flow Machines and Languages," *Computer*, pp. 58-69, February 1982.

Gaudiot, Jean-Luc, "Structure Handling in Data-Flow Systems," *IEEE Transactions on Computers*, vol. C-35, no. 6, pp. 489-502, June 1986.

Gaudiot, Jean-Luc and Yi-Hsiu Wei, "Token Relabeling in a Tagged Token Data-flow Architecture," *IEEE Transactions on Computers*, vol. 38, no. 9, pp. 1225-1239, September 1989.

Gehringer, Edward F and J Leslie Keedy, "Tagged Architecture: How Compelling Are its Advantages?," *SIGARCH Newsletter*, vol. 13, no. 3, pp. 162-170, June 1985.

Gelernter, David, "Generative Communication in Linda," *ACM Transaction on Programming Languages and Systems*, vol. 7, no. 1, pp. 80-112, January 1985.

Gelsinger, Patrick P, Paolo A Gargini, Gerhard H Parker, and Albert Y C Yu, "Microprocessors circa 2000," *IEEE Spectrum*, vol. 26, no. 10, pp. 43-47, October 1989.

Genesereth, Michael R and Nils J Nilsson, *Logical Foundations of Artificial Intelligence*, Morgan Kaufmann Publishers, Los Altos, California, 1987.

Gostelow, Kim P and Robert E Thomas, "Performance of a Simulated Dataflow Computer," *IEEE Transactions on Computers*, vol. C-29, no. 10, pp. 905-919, October 1980.

Graf, Hans P and Paul de Vegvar, "A CMOS Associative Memory Chip Based on Neural Networks," in *1987 IEEE International Solid-State Circuits Conference ISSCC 87*, ed. IEEE, pp. 304-305,437, February 1987.

Graham, Susan L, Peter B Kessler, and Marshall K McKusick, "gprof: a Call Graph Execution Profiler," *SIGPLAN Notices*, vol. 17, no. 6, pp. 120-126, June 1982.

Griffiths, Michael and Carol Palissier, *Algorithmic Methods for Artificial Intelligence*, Chapman and Hall, 1987.

Grosspietsch, K. E., H. Huber, and A. Muller, "The concept of a fault-tolerant and easily-testable associative memory," in *Proc. 16th Fault-tolerant Computing Symp.*, ed. IEEE, pp. 34-39, July 1986.

Grosspietsch, K. E., "Architectures for testability and fault tolerance in content-addressable systems," *IEE Proceedings*, vol. 136, Pt. E, no. 5, pp. 366-373, September 1989.

Grosspietsch, K E, H huber, and A Muller, "The VLSI Implementation of a Fault-tolerant and Easily-testable Associative Memory," *Proceedings of VLSI and Computers First International Conference on Computer Technology, Systems and Applications*, pp. 47-50, IEEE, Hamburg, May 11-15, 1987.

Hammerstrom, Dan, "A VLSI Architecture for High-Performance, Low-Cost, On-chip Learning," in *IEEE Int. Conf. on Neural Networks*, vol. 1, pp. 1-8, 1990.

Hammond, Joseph L and Peter J P O/Reilly, "Error Control," in *Performance Analysis of Local Computer Networks*, ed. Addison-Wesley Publishing Co., pp. 42-66, 1986.

Herrmann, Frederick P, Craig L Keast, Keisuke Ishio, Jon P Wade, and Charles G Sodini, "A Dynamic Three-State Memory Cell for High-density Associative Processors," *IEEE Journal of Solid-State Circuits*, vol. 26, no. 4, pp. 537-541, April 1991.

Higuchi, Tetsuya, Tatsumi Furuya, Kenichi Handa, Naoto Takahashi, Hiroyasu Nishiyama, and Akio Kokubo, "IXM2: A Parallel Associative Processor," *Computer Architecture News*, vol. 19, no. 3, pp. 22-31, Toronto, Canada, May 1991.

Hillberg, W, "Neural networks and conditional association networks: common properties and differences," *IEE Proceedings Pt.E*, vol. 136, no. 5, pp. 343-350, September 1989.

Hillis, W Daniel and Guy L Steele Jr, "Data Parallel Algorithms," *Communications of the ACM*, vol. 29, no. 12, pp. 1170-1183, December 1986.

Hirata, Masaki, Hachiro Yamada, Hajime Nagai, and Kousuke Takashi, "A Versatile Data String-Search VLSI," *IEEE Journal of Solid-State Circuits*, vol. 23, no. 2, pp. 329-335, April 1988.

Hollis, Paul W and John J Paulos, "Artificial Neural Networks Using MOS Analog Multipliers," *IEEE Journal of Solid-State Circuits*, vol. 25, no. 3, pp. 849-855, June 1990.

Hong, S J, R G Cain, and D L Ostapko, "MINI: A Heuristic Approach to Logic Minimization," *IBM Journal of research and development*, vol. 18, no. 5, pp. 443-458, September 1974.

Howard, Richard E., Daniel B. Schwartz, John S. Denker, Roger W. Epworth, Hans Peter Graf, Wayne E. Hubbard, Lawrence D. Jackel, Brian L. Straughn, and D. M. Tennant, "An Associative Memory Based on an Electronic Neural Network Architecture," *IEEE Transactions on Electron Devices*, vol. ED-34, no. 7, pp. 1553-1556, July 1987.

Hwang, Kai and Faye A Briggs, *Computer Architecture and Parallel Processing*, pp. 56-57, 1984.

Iannucci, Robert A, *Toward a Dataflow / Von Neumann Hybrid Architecture*, pp. 131-139, 1988.

IBM Corp., "System for Efficiently Using Spare Memory Components for Defect Corrections Employing Content-Addressable Memory," *IBM Technical Disclosure Bulletin*, vol. 28, no. 6, pp. 2562-2567, November 1985.

Isenman, Merrill E and Dennis E Shasha, "Performance and Architectural Issues for String Matching," *IEEE JSSC*, vol. 39, no. 2, pp. 238-250, February 1990.

Itoh, Kiyo, "Trends in Megabit DRAM Circuit Design," *IEEE Journal of Solid-State Circuits*, vol. 25, no. 3, pp. 779-789, June 1990.

Jones, Simon, "Design, selection and implementation of a content-addressable memory for a VLSI CMOS chip architecture," *IEE Proceedings*, vol. 135, Pt. E, no. 3, pp. 165-172, May 1988.

Jones, Simon R., Ian P. Jalowiecki, Stephen J. Hedge, and R. M. Lea, "A 9-kbit Associative Memory for High-Speed Parallel Processing Applications," *IEEE Journal of Solid-State Circuits*, vol. 23, no. 2, pp. 543-548, April 1988.

Kadota, Hiroshi, Jiro Miyake, Yoshito Nishimichi, Hitoshi Kudoh, and Keiichi Kagawa, "An 8-kbit Content-Addressable and Reentrant Memory," *IEEE Journal of Solid-State Circuits*, vol. SC-20, no. 5, pp. 951-956, October 1985.

Kam, Moshe, Roger Cheng, and Allon Guez, "On the Design of a Content-Addressable Memory via Binary Neural Networks," in *IEEE Int. Conf. on Neural Networks*, pp. II-513, II-522, 1988.

Kohonen, Teuvo, *Associative Memory, a System-Theoretical Approach*, Springer-Verlag, Berlin, 1977.

Kohonen, Teuvo, *Content-Addressable Memories*, Berlin, 1980.

Kolman, Bernard and Robert C Busby, *Discrete Mathematical Structures for Computer Science*, p. chapter 1, Englewood Cliffs, NJ, 1984.

Kuo, Y S, "Generating Essential Primes for a Boolean Function with Multiple-Valued Inputs," *IEEE Transactions on Computers*, vol. c-36, no. 3, p. 356, March 1987.

Lea, R. M., "SCAPE: a single-chip array processing element for signal and image processing," *IEE Proceedings, Pt. E*, vol. 133, no. 3, pp. 145-151, May 1986.

Lea, R M, "VLSI and WSI associative string processor for structured data processing," *IEE Proceedings Pt.E*, vol. 133, no. 3, pp. 153-162, May 1986.

Lea, R M, "ASP: A Cost-effective Parallel Microcomputer," *IEEE Micro*, vol. 8, no. 5, pp. 10-29, October 1988.

Lee, Dik Lun, "A Distributed Multiple-Response Resolver for Value-ordered Retrieval," *SIGARCH Newsletter*, vol. 13, no. 3, pp. 258-265, June 1985.

Lee, Dik Lun and Frederick H Lochovsky, "HYTREM - A Hybrid Text-Retrieval Machine for Large Databases," *IEEE JSSC*, vol. 39, no. 1, pp. 111-123, January 1990.

Leler, Wm, "Linda Meets Unix," *Computer*, pp. 43-54, February 1990.

Lin, Yow-Jian and Vipin Kumar, "AND-parallel Execution of Logic Programs on a Shared Memory Multiprocessor: A Summary of Results," in *Logic Programming: Proceedings of the 5th International Conference and Symposium*, ed. Kenneth A Bowen, vol. 2, pp. 1123-1139, MIT Press, 1988.

Lucente, Michael A, Clifford H Harris, and Robert M Muir, "Memory System Reliability Improvement Through Associative Cache Redundancy," *IEEE JSSC*, vol. 26, no. 3, pp. 404-409, March 1991.

Martin, Ursula and Tobias Nipkow, "Boolean Unification - The Story So Far," *Journal of Symbolic Computation*, vol. 7, pp. 275-292, 1989.

Mazumder, Pinaki, Janak H. Patel, and W. Kent Fuchs, "Methodologies for Testing Embedded Content Addressable Memories," *IEEE Transactions on Computer-aided Design*, vol. 7, no. 1, pp. 11-20, January 1988.

McAuley, Anthony J and Charles J Cotton, "A Self-Testing Reconfigurable CAM," *IEEE Journal of Solid-State Circuits*, vol. 26, no. 3, pp. 257-261, March 1991.

McEliece, R J, E C Posner, E R Rodemich, and S S Venkatesh, "The capacity of the Hopfield associative memory," *IEEE Transaction on Information Theory*, vol. IT-33, no. 4, pp. 461-482, 1987.

Mead, Carver A, *Analog VLSI and Neural Systems*, Reading, MA, 1989.

Milutinovic, Veljko M, *High-level language computer architecture*, Computer Science Press, 1988.

Moskowitz, J P and C Jousselin, "An algebraic memory model," *Computer Architecture News*, vol. 17, no. 1, pp. 55-61, march 1989.

Moto-oka, Tohru, "Overview to the fifth generation computer system project," in *10th Int Symp on Computer Architecture*, pp. 417-422, 1983.

Motomura, Masato, 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 of Solid-State Circuits*, vol. 25, no. 5, pp. 1158-1165, October 1990.

Mundy, Joseph L, James F Burgess, Reuben E Joyson, and Constantine Neugebauer, "Low-Cost Associative Memory," *IEEE Journal of Solid-State Circuits*, vol. SC-7, no. 5, pp. 364-369, October 1972.

Murakami, Kunio, Takeo Kakuta, Nobuyoshi Miyazaki, Shigeki Shibayama, and Haruo Yokota, "A Relational Data Base Machine: First Step to Knowledge Base Machine," in *10th Int. Symposium on Computer Architecture*, pp. 423-425, 1983.

Murray, Alan F. and Anthony V. W. Smith, "Asynchronous VLSI Neural Networks Using Pulse-Stream Arithmetic," *IEE Journal of Solid-State Circuits*, vol. 23, no. 3, pp. 688-697, June 1989.

Naganuma, Jiro, Takeshi Ogura, Shin-Ichiro Yamada, and Takashi Kimura, "High-Speed CAM-Based Architecture for a Prolog Machine (ASCA)," *IEEE Transactions on Computers*, vol. 37, no. 11, pp. 1375-1383, November 1988.

Ng, Yan H and Raymond J Glover, "The Basic Memory Support for Functional Languages," in *Proceedings of VLSI and Computers First International Conference on Computer Technology, Systems and Applications*, ed. IEEE, pp. 35-40, IEEE, Hamburg, May 11-15, 1987.

Nijhuis, J A G and L Spaanenburg, "Fault tolerance of neural associative memories," *IEE Proceedings Pt.E*, vol. 136, no. 5, pp. 389-394, September 1989.

Nikhil, Rishiyur S and Arvind, "Can dataflow subsume von Neumann computing?," in *International Symposium on Computer Architecture*, ed. ACM, pp. 262-272, 1989.

Nodes, T. A., J. L. Smith, and R. Hecht-Nielsen, "A Fuzzy Associative Memory Module and its Application to Signal Processing," in *Proceedings of the International Conference on Acoustics Speech and Signal Processing, ICASSP*, ed. IEEE, pp. 1511-1514, New York, 1985.

Nogami, Kazutaka, Takayasu Sakurai, Kazuhiro Sawada, Kenji Sakaue, Yuichi Miyazawa, Shigeru Tanaka, Yoichi Hiruta, Katsuto Katoh, Toshinari Takayanagi, Tsukasa Shirotori, Yukiko Itoh, Masanori Uchida, and Tetsuya Iizuka, "A 9-ns HIT-Delay 32-Kbyte Cache Macro for High-Speed RISC," *IEEE JSSC*, vol. 25, no. 1, pp. 100-106, February 1990.

O'Keefe, Richard A, "A Comment on " A Hardware Unification Unit: Design and Analysis"," *Computer Architecture News*, vol. 14, no. 1, pp. 2-3, January 1986.

Ogura, Takeshi, Shin-Ichiro Yamada, and Tadanabu Nikaido, "A 4-kbit Associative Memory LSI," *IEEE Journal of Solid-State Circuits*, vol. SC-20, no. 6, pp. 1277-1282, December 1985.

Ogura, Takeshi, Junzo Yamada, Shin-ichiro Yamada, and Masa-aki Tan-no, "A 20-kbit Associative Memory LSI for Artificial Intelligence Machines," *Journal of Solid-State Circuits*, vol. 24, no. 4, pp. 1014-1020, August 1989.

Oldfield, John V, Charles D Stormon, and Mark Brule, "The Application of VLSI Content-addressable Memories to the Acceleration of Logic Programming Systems," *Proceedings of VLSI and Computers First International Conference on Computer Technology, Systems and Applications*, pp. 27-30, IEEE, Hamburg, May 11-15, 1987.

Oldfield, J V, "Logic programms and an experimental architecture for their execution," *IEE Proceedings pt. E*, vol. 133, no. 3, pp. 163-167, May 1986.

Oldfield, J V, R D Williams, and N E Wiseman, "Content-Addressable Memories for Storing and Processing Recursively Subdivided Images and Trees," *Electronic Letters*, vol. 23, no. 6, pp. 262-263, 12 March 1987.

Papachristou, Christos A, "Content-Addressable Memory Requirements for Multivalued Logic," in *11th Int. Symposium on Multi-Valued Logic*, pp. 62-72, 1981.

Peterson, Craig, "iWarp," in *HOT Chips Symposium Record*, Santa Clara, CA, August 20-21, 1990.

Ramamoorthy, C V, James L Turner, and Benjamin W Wah, "A Design of a Fast Cellular Associative Memory for Ordered Retrieval," *IEEEE Transactions on Computers*, vol. C-27, no. 9, pp. 800-815, September 1978.

Ribeiro, J. C. D. F., C. D. Stormon, J. V. Oldfield, and M. R. Brule, "Content-addressable memories applied to execution of logic programs," *IEE Proceedings*, vol. 136, Pt. E, no. 5, pp. 383-388, September 1989.

Robinson, Phillip, "The SUM: an AI coprocessor," *BYTE*, pp. 169-180, June 1985.

Rokey, Mark, "The Dataflow Architecture: A Suitable Base for the Implementation of Expert Systems," *Computer Architecture News*, vol. 13, no. 4, pp. 8-14, September 1985.

Ross, G R T, *Aristotle de Sensu and de Memoria Text and Translation with Introduction and Commentary*, Arno Press, New York, 1973.

Rudell, R and A Sangiovanni-Vincentelli, *Multiple-Valued Minimization for PLA Optimization*, pp. 198-207, IEEE, 1987.

Rueckert, U, I Kreuzer, and K Goser, "A VLSI Concept for an Adaptative Associative Matrix Based on Neural Networks," *Proceedings of VLSI and Computers First International Conference on Computer Technology, Systems and Applications*, pp. 31-34, IEEE, Hamburg, May 11-15, 1987.

Sage, J P, K Thompson, and R S Withers, "An artificial neural network integrated circuit based on MNOS/CCD principles," in *Proc AIP Conf. Neural Networks for Computing*, pp. 381-385, Snowbird, 1986.

Saluja, K K and S M Reddy, "Fault Detecting Test Sets for Reed-Muller Canonic Networks," *IEEE Transactions on Computers*, pp. 995-998, October 1975.

Sasao, Tsutomo, "Input Variable Assignment and Output Phase Optimization of PLA's," *IEEE Transactions on Computers*, vol. c-33, no. 10, p. 892, October 1984.

Sasao, Tsutomo, "An Algorithm to Derive the Complement of a Binary Function with Multiple-Valued Inputs," *IEEE Transactions on Computers*, vol. C-34, no. 2, pp. 131-140, IEEE, February 1985.

Schuster, S. E., "Dynamic Content Addressable Memory with Refresh Feature," *IBM Technical Disclosure Bulletin*, vol. 26, no. 10B, pp. 5364-5366, March 1984.

Seitz, Charles L. and Juri Matisoo, "Engineering limits on computer performance," *Physics Today*, pp. 38-45, May 1984.

Shankar, Subash, "A Hierarchical Associative Memory Architecture for Logic Programming Unification," in *Logic Programming: Proceedings of the 5th International Conference and Symposium*, ed. Kenneth A Bowen, vol. 2, pp. 1428-1447, MIT Press, 1988.

Shobatake, Yasuro and Hideo Aiso, "A unification processor based on uniformly structured cellular hardware," in *13th International Symposium on Computer Architecture*, pp. 140-148, 1986.

Siekmann, Jorg H, "Unification Theory," *Journal of Symbolic Computation*, vol. 7, pp. 207-274, 1989.

Sorabji, Richard, *Aristotle on Memory*, Brown University Press, Providence, 1972.

Stanat, Donald F and David F McAllister, *Discrete Mathematics in Computer Science*, p. chapter 2, Englewood Cliffs, NJ, 1977.

Stormon, C D, M R Brule, J V Oldfield, and J C D F Ribeiro, "An Architecture Based on Content-Addressable Memory for the Rapid Execution of Prolog," in *Logic Programming: Proceedings of the 5th International Conference and Symposium*, ed. Kenneth A Bowen, vol. 2, pp. 1449-1473, MIT Press, 1988.

Strugala, M, D Tavangarian, K. Waldschmidt, and G. Roll, "An Associative Processor as a Design Rule Check Accelerator," *Proceedings of VLSI and Computers First International Conference on Computer Technology, Systems and Applications*, pp. 426-431, IEEE, Hamburg, May 11-15, 1987.

Stubberud, Allen R., "Failure Isolation Using an Associative Memory Algorithm," in *Proceedings of 25th Conference on Decision and Control*, ed. IEEE, pp. 1113-1115, Athens, Greece, December 1986.

Su, S Y H and P T Cheung, "Computer Minimization of Multivalued Switching Functions," *IEEE Transactions on Computers*, vol. c-21, no. 9, p. 995, September 1972.

Tavangarian, D., "Flag-algebra: a new concept for the realisation of fully parallel associative architectures," *IEE Proceedings*, vol. 136, Pt. E, no. 5, pp. 357-365, September 1989.

Treleaven, Philip C, David R Brownbridge, and Richard P Hopkins, "Data-Driven and Demand-Driven Computer Architecture," *Computing Surveys*, vol. 14, no. 1, pp. 93-142, March 1982.

Treleaven, Philip C, "The New Generation of Computer Architecture," *SIGARCH Newsletter*, vol. 11, no. 3, pp. 402-409, June 1983.

Uchida, Shunichi, "Inference Machine: From Sequential to Parallel," in *10th Int. Symposium on Computer Architecture*, pp. 410-416, 1983.

Ullman, Jeffrey D, *Principles of Database Systems*, 1983.

Ulug, M E, "VLSI Knowledge Representation Using Predicate Logic and Cubical Algebra," in *Phoenix Conference on Computers and Communication*, pp. 292-299, 1987.

Venta, Olli and Teuvo Kohonen, "A Content-Addressing Software Method for the Emulation of Neural Networks," in *IEEE Int. Conf. on Neural Networks*, pp. I-191, I-198, July 1988.

Vittoz, Eric A and Xavier Arreguit, "CMOS Integration of Herault-Jutten Cells for Separation of Sources," in *Workshop on "Analog VLSI and Neural Systems"*, ed. to be published by Kluwer Academic Pu., Portland, May 8, 1989.

Wade, Jon P. and Charles G. Sodini, "Dynamic Cross-Coupled Bit-Line Content Addressable Memory Cell for High-Density Arrays," *IEEE Journal of Solid-State Circuits*, vol. SC-22, no. 1, pp. 119-121, February 1987.

Wade, Jon P. and Charles G. Sodini, "A Ternary Content Addressable Search Engine," *Journal of Solid-State Circuits*, vol. 24, no. 4, pp. 1003-1013, August 1989.

Waldschmidt, Klaus, "Associative Processors and Memories Overview and Current Status," *Proceedings of VLSI and Computers First International Conference on Computer Technology, Systems and Applications*, pp. 19-26, IEEE, Hamburg, May 11-15, 1987.

Wegmann, George and Eric A Vittoz, "Analysis and Improvements of Accurate Dynamic Current Mirrors," *IEEE Journal of Solid-State Circuits*, vol. 25, no. 3, pp. 699-706, June 1990.

Welch, Terry A, "An Investigation of Descriptor Oriented Architecture," *Computer Architecture News*, vol. 4, no. 4, pp. 141-146, January 1976.

Woo, Nam Sung, "A Hardware Unification Unit: Design and Analysis," in *12th International Symposium on Computer Architecture*, pp. 198-205, 1985.

Yasuura, Hiroto, Taizo Tsujimoto, and Keikichi Tamaru, "Parallel Exhaustive Search for Several NP-Complete Problems Using Content Addressable Memories," in *Proceeding ISCAS 88*, ed. IEEE, pp. 333-336, 1988.

Yokota, Haruo and Hidenori Itoh, "A Model and an Architecture for a Relational Knowledge Base," *Computer Architecture News*, vol. 14, no. 2, pp. 2-9, June 1986.

Yuba, Toshitsugu, Toshio Shimada, Yoshinori Yamaguchi, Kei Hiraki, and Shuichi Sakai, "Dataflow Computer Development in Japan," *ACM SIGARCH NEWS*, vol. 18, no. 3, pp. 140-147, September 1990.

Zeidler, H. Ch., "Content-addressable mass memories," *IEE Proceedings*, vol. 136, Pt. E, no. 5, pp. 351-356, September 1989.