A Universal Logic Machine

Marek A. Perkowski
Department of Electrical Engineering
Portland State University
P.0. Box 751, Portland, Oregon 97207, tel. (503) 725-5411

Abstract

Although machines which manipulate logic functions have been investigated since the 13th century, much more research has gone into arithmetic machines because of their obvious computational applications. Many newly developing applications such as logic optimization, logic programming, simulation, data-bases, graph theory, computer image processing and image recognition, spectral transforms and signal processing can be better implemented with a logic machine. After a brief tour through the history of logic machines, this paper describes our Cube Calculus Machine (CCM2) [1]. This machine is based on a new architecture in which the data path has been designed to execute operations of "cube calculus", an algebraic model popularly used to process and minimize Boolean functions. CCM2 realizes efficiently all cube calculus operations such as sharp and consensus. The "positional cube representation" used by CCM2 can also represent multiple-valued input binary output cube calculus (MVCC) operands as is commonly done for applications in Logic Synthesis, and other computational methods based on logic. CCM2 can also work with Generalized Multiple-valued Cube calculus (GMVCC), which I developed as an extension of MVCC. This extension allows the machine to operate on set logic, associative tuples, several multiple-valued input multiple-valued output logics, multi-output relations, and symbolic relations.

1 Introduction

A *formal system* is a set of symbols and a set of operations on those symbols. All the formal systems of mathematics are build on the foundation of *set theory* (and equivalently - *logic*). One such system is arithmetic. Most existing computers were designed to perform a subset of arithmetic operations. ALU logic operations such as bit-wise AND, OR, NOT and EXOR are just simple extensions of arithmetic operations, so these computers can be characterized as *"arithmetic"*.

Since all formal systems can be expressed in terms of arithmetic and logic, any of these systems can be implemented on current arithmetic computers by using the appropriate "model" and software which manipulates this model. The price that is often paid for a software based approach is severe speed degradation. This degradation has made the implementation of several high-level formal systems impractical.

Two methods can be used to improve speed at which computers can implement specialized formal systems: **(1)** To design special hardware which implements the specific operations required by the systems. Examples of this method are math, DSP, and image processing coprocessors. **(2)** To execute operations in parallel on multiple processing units.

Current approaches are based on reducing any problem to sequences of arithmetic operations and executing these operations efficiently. We postulate that for many applications, it is much more effective to use a methodology that reduces a wide class of problems to some elementary formal system other than arithmetic, and then design an efficient hardware realization of the basic operations of this system.

Multiple-valued Cube Calculus (MVCC) seems to be one of the most general internal representations of data in propositional logic, logic synthesis, logic programming, logic simulation and sequential evaluation of combinational logic, data-bases, and several areas of AI and problem-solving. Generalized Multiple-valued Cube Calculus(GMVCC), our extension of MVCC, is even more powerful than MVCC because it can represent multiple-valued input, multiple-valued-output logic (called *truly mv logic*), multi-output relations, predicates and other data. This means that it can be used for real-time AI applications, image processing, genetic algorithms, fuzzy logic and logic programming. As with other formalisms such as fuzzy logic, and morphological image algebras, which were not fully appreciated until special hardware was built for them, it may be reasonably expected that computers which operate in GMVCC will find their "applications niche" among computer architectures.

Our group has designed, simulated, and is currently building a Cube Calculus Machine (CCM2) which operates in GMVCC. As it will be described, the heart of the CCM2 is a bit-slice data path constructed as an iterative network of programmable state machines.

To give a perspective of how our machine fits in the long history of logic machines, in the next section we will briefly identify some of the major steps in this history. In section 3 we will discuss binary and multiple-valued cube calculus and show examples of common cube calculus operations. Section 4 introduces the new Generalized Multiple-valued Cube Calculus(GMVCC). Section 5 discusses main ideas

[1] The author is currently seeking a patent for the concept, architecture and hardware described here.

of Cube Calculus Machines: operations on symbols, and hardware realization of the lowest level algorithm loops by a ring of cellular automata. In section 6 we briefly discuss the structure of the CCM2 processor.

2 Short History of Logic Machines

By a Logic Machine we mean any machine intended to solve problems expressed in logic form. In the long history of computing there were several attempts to build such machines, but most were forgotten and in a sense not successful. Some ideas were reinvented many times [2]. The first logic machine was described and built by Raymon Lullus (1235-1315), a Catalonian mystic, professor, linguist, poet, and missionary [11]. It was designed to solve syllogisms. Dealing with binary variables and binary counting his machine did mechanically the same thing as a Venn diagram does visually - enumerate all possible elements of a Cartesian product of sets of variables values. Also, since variables could have more than two elements each, the machine was able to enumerate in N-ary counting (N = number of variables) all elements (minterms) of some solution spaces as Cartesian products of the sets of values of those multiple-valued input variables. Lullus work can be then in a sense thought of as a precursor of multiple-valued logic and the morphological method of invention later reinvented by Zwicky [14]. Lullus had excellent intuitions about his machine, its generalizations and possible uses, and he is regarded by some authors as the father of the Science of Heuristics and Mechanical Theorem Proving. "Tabulating combinations of terms was certainly a familiar process to mathematicians as far back as the Greeks" [11], but nobody before Lullus was able to create a machine and to intuitively develop the concepts of symbolic logic and universal methods to solve problems. His ideas influenced Leibnitz, Bruno, Pierce, other logicians, and the developers of early arithmetic computers.

"The inventor of the world's first logic machine, in a stricter sense of the term, was a colorful eighteenth century British statesman and scientist, Charles Stanhope (1753-1816). His curious device, which he called a "demonstrator", could be used for solving traditional syllogisms by a method closely linked to Venn circles. It also solved numerical syllogisms (anticipating De Morgan's analysis of such forms) and elementary problems of probability. Stanhope's machine was based on a system of logical notation which clearly foreshadowed Hamilton's technique of reducing syllogisms to statements of identity using negative terms and quantified predicates." [11].

Development of logic machines up to 1967 is described in [11,23,33], but to give a feeling for the evolution, here is a list of some of the highlights.

[2] Therefore, an effort has been recently made to find and systematize all available information about such machines. Only representative papers and those that influenced us most are mentioned here. The history of logic machines until 1967 is presented in [11]. The reader wishing to find more details about other machines is referred to [31]. Since non-English literature is often not available to us, we would appreciate obtaining any additional information or corrections that will be useful in our monograph book on logic machines.

* William Stanley Jevons, British logician, invented and had a clockmaker build his "logical piano" in 1863-1869 [11,33]. "It was the first such machine with sufficient power to solve a complicated problem faster than the problem could be solved without the machine's aid" [11]. Jevons was one of first who recognized the power of Boolean logic.

* A diagram variant of Jevons' machine was created by English logician Reverend John Venn (1834-1923).

* Another improved version of the Jevons' machine was developed in 1881 by American professor Allan Marquand (1853-1924), the inventor of Marquand Charts. He also designed an electrical version of it, using electromagnets, multiway switches and a rheostat.

* Two Italian professors, Annibale Pastorale and Antonio Garbasso, constructed in 1903 what can be called the first "analog logic machine", precursor in a sense to modern "fuzzy logic machines".

* Englishman Charles Macaulay obtained in 1913 the first patent for a logic machine which combines the best features of Jevons's and Marquand's machines.

* In 1936 American psychologist Benjamin Burack built the first electrical logic machine [5].

* William Burkart and Theodore Kalin built the first electrical machine designed solely for propositional logic in 1947 while taking the undergraduate symbolic logic course at Harvard with Professor Willard Quine.

* Burroughs Research Center in Pennsylvania designed a ten-term "truth-function evaluator", which used a logical notation proposed by Polish logician Jan Lukasiewicz [6].

While the above machines can be roughly characterized as based on the scanning method of solving Boolean Equations, another line of development, oriented towards designing circuits and minimizing Boolean functions, started in 1952. Perhaps the world's first minimizing machine was build by Daniel Bobrow, then a high school student [3]. Closely related was the machine of Shannon and Moore [37], who in 1953 constructed an analyzer of four-variable Boolean functions for the Bell Labs. In the fifties and sixties many Logic Machines were constructed in the USA, Great Britain [11], and especially in the Soviet Union, often by large groups and at great expense. They were built to aid in analysis and synthesis of contact and relay-contact circuits, but some authors also foresaw wider applications. For instance, Rodin [35] developed a machine for six variables that was based on the Shannon's concept. A large group of researchers including Arkhangelskyaya, Roginskii, Lazarev, Sagalovich, Parkhomenko and Oganov [2] developed several machines for design of multiterminal cascade networks using a theory they developed. One version could handle up to 10 variables. Zakrevskii and Gavrilov [12,47] developed what was perhaps the first "hardware accelerator" for logic synthesis. It was intended to work as a co-processor in a general purpose computer. They developed also the first computer language for solving problems in logic.

Antonin Svoboda worked on logic machines in Czechoslovakia and in 1968 he presented his Boolean

Analyzer (BA) at IFIP [41]. Later he moved to the U.S. where he continued his research at UCLA [42,43]. The work of his group was perhaps the first on general purpose binary logic machines. Several Master Theses were written [13,22,38], but I do not know if a machine was ever actually built.

Not knowing of all these efforts, I designed my first logic machine at the Technical University of Warsaw in Poland in 1973. The machine was build to solve the set covering (Petrick function minimization) and satisfiability/tautology problems and was based again on a binary/ternary counter. Using "universal digital modules" and mechanical switches for input, as a teaching aid I actually built a prototype of this machine to solve 8 by 8 covering tables. In 1976-1982 our group created a general theoretical framework and computer programs to solve a class of consistent labeling tree search problems that occur in logic, games and design [25,26,29], [E37] (to save space the references starting from E are from [32]). It included not only binary but also multiple-valued input logic, which I called "non-univocal logic" [26,20]. This system was perhaps the first one since Lullus' to solve problems in multiple-valued logic [3].

The new era of logic machines started in 1985, when Tsutomu Sasao [36] presented a Tautology Machine, called HART, designed to speed-up ESPRESSO and other two-level PLA minimizers. His machine was actually built in hardware as a PAL-based co-processor for a PC-compatible computer and gave considerable speed improvements. To my knowledge this is the only machine actually implemented in hardware after 1977 and the first machine to use programmable devices. However, like many previous machines it was not commercialized.

While the machine from [36] was intended for just one task, a design we did at PSU [27] was for a general-purpose parallel hardware logic accelerator. Our machine was optimized for satisfiability and related problems that can be reduced to the manipulation of lists of binary and multiple-valued cubes. The proposed machine included several computers. One of them, called the Monte-Carlo Constraints Computer (MCCC) was the precursor of our Cube Calculus Machine(CCM). It used a ring of simple processors which implemented a pipelined, data flow architecture. The MCCC computer was essentially a very-long-word RISC processor with enhanced logic and bit operations. The processor executed vector-wise all two-variable Boolean functions including such cube operations as inclusion, supercube and intersection; comparison with binary patterns; shifts; and such operations as selection of the first "one" from the left, or the random "one" from the vector. It used stochastic and exhaustive search methods and was optimized for solving problems described by constraints. The main idea was to have one processor for a subset of constraints (equations). The MCCC design was simulated and an NMOS chip layout done, but the chip was not fabricated.

[3] Until 1985 I was not aware of positional notation of mv logic as developed by Su and Sasao [40] [E46-E49], or any previous efforts to build logic machines.

At this time two other approaches to logic processors were being pursued: *Content Addressable Memory (CAM) machines* and *Equation Solving machines.* **(1)** In 1986 we proposed a CAM-based machine, Constraints Optimizer [28], that was a generalization of the machines from [27,36]. Yasuura [46] proposed a CAM-based architecture that uses exhaustive search to solve the "Traveling Salesman", "Set Covering", and several other combinatorial problems close to those of logic synthesis. Kida [18] presented a CAM-based machine which is the successor of machines from [28,45], but is more general. **(2)** The second approach was represented by a machine to solve the *partial satisfiability problem* [21], designed in [19]. It was based on reduction of a combinatorial problem to the partial satisfiability problem. This problem was next reduced to integer programming, which was solved using the Karmarkar's algorithm. In turn, the *Karmarkar algorithm* used a general-purpose parallel architecture for solving equations, which, among other matrix operations, realized the *Extended Faddev Algorithm.*

As I learned about previous unsuccessful efforts to built logic machines, I realized that because of the Amdahl Law, to be really useful, a machine must be much more general than any of the machines proposed before. I also realized that logic synthesis operations are strongly related to those used in automated theorem-proving and computer vision, which influenced our new architectures. For example, Ulug [44,45] proposed an extended cube calculus machine for resolution-based theorem-proving with applications in real time AI and data-bases.

Our research on CCM has concentrated on achieving the following goals: **(1)** To determine the most basic idea that all those architectures have in common and built on this idea. (Based on simulation results, the scanning principle that dominated the previous machines was excluded as the basis of our CCM). **(2)** The machine should be a *general-purpose computer*, and must include the most common, and all the necessary instructions of general-purpose computers. **(3)** The complete machine must be a hardware accelerator board for a general purpose computer, and the component chips must be building blocks for various massively parallel architectures such as those based on GAPP and Transputer devices, or pipelined and systolic DSP architectures.

These were the design goals of the CCM-1 [17], designed using OCT tools. This machine introduced for the first time: **(1)** a flexible number of values in literals; **(2)** the concept of two-bit IT cell to represent a part of an mv-literal of an arbitrary number of values; **(3)** a data-path based on an iterative circuit of Finite State Machines (FSMs) with information flowing between FSMs from left to right and from right to left. This was also the first machine which included a more complete set of cube calculus operations, and the first which did not use exhaustive scanning in any way. Algorithms such as tautology or satisfiability, which previously used scanning, were replaced with algorithms based on the cube calculus operations sharp, intersection, and crosslink. A tree architecture designed from CCM-1 and sorting chips for *Generalized Satisfiability*

Problem was described in [16].

Since 1989 our group developed a much improved machine called CCM2 [30]. CCM2 introduced the following new concepts to logic machines: **(1)** Relations and operators can be arbitrary (programmable) functions of input variables. This allows us to create an extremely large number of logic operators by combining basic operators. **(2)** While CCM-1 executed only set-theoretical operations on literals, CCM2 introduces simple and complex symbols, as an intermediate level between bits and variables. This allows to realize arbitrary truly mv logic and to deal with the literals being various kinds of numbers. CCM2 can also deal with data such as: number intervals, symbolic predicates, associative tuples, and multi-valued multi-output relations. These capabilities greatly expand its semantics and the range of potential applications. **(3)** The CCM2 machine is microprogrammed, to make the best use of the above property. **(4)** CCM2 is a general-purpose computer. It belongs to a superset of standard arithmetic computers, string matching computers, associative processors and Long-Word Computers. **(5)** CCM architecture includes a Host Procesor that controls a massively parallel structure from CCM2 processors. The CCM2 processor chip is designed to work in various topological configurations, with various Controller and Content Addressable Memory (CAM) chips. A hardware implementation of CCM2 is currently under development.

3 Binary and Multiple-Valued Input Cube Calculus

There are two basic representation methods for switching functions used in most of the state-of-the-art logic synthesis programs: *decision diagrams*, and *cube calculus*. The main concepts of the cube calculus are those of a *cube* and an *array of cubes* (list of cubes). In this paper an array of cubes will be called *a clist*. In binary input, binary output (Boolean) cube calculus and in multiple-valued (mv) binary output cube calculus a cube usually represents one of the following: **(1)** a *product* of literals, **(2)** a *sum* of literals, **(3)** an *exclusive sum* of literals. It can also represent the *EQUIVALENCE* of literals, the *polarity* of a unate function or a Generalized Reed-Muller (GRM) form, an *index of a spectral coefficient*, or other data equivalent to an *ordered set of ordered sets*.

3.1 Basic Cube Notation

In *binary logic* a *literal* is a binary variable or its negation. A variable x can be represented as $x = x^1$ and its negation can be represented as $\overline{x} = x^0$. In multi-valued input, binary output logic a literal is denoted as a variable with superscripts that represent the values of the variable for which the expression is satisfied. (For basic definitions related to mv functions see [32]). N here denotes the number of variables (literals, positions) in a cube. A product of literals, ${X_1}^{S_1}$ ${X_2}^{S_2}$ ... ${X_n}^{S_N}$, is referred to as a *product term*, *term*, or *product* and can be represented as a cube. A product term that includes literals for all function variables X_1, X_2, ..., X_N is called a *full term.* A sum of products is denoted as a *SOPE*, a product of sums is called a *POSE*, and an EXOR of products is called an *Exclusive Sum of Products Form* or *ESOP*. Also, a product of EXORs is called a *Product of Exclusive Sums expression* or *POES*. SOPE, POSE, ESOP and POES can all be represented as *clists of cubes.*

Example 3.1. Let us assume the order of variables in a cube: x_1, x_2, x_3, x_4. Cube $x_1\ x_2\ \overline{x_3}$ representing a product of literals is denoted in *symbol notation* as 110X. SOPE $x_1\ x_2\ \overline{x_3} + \overline{x_1}\ \overline{x_3}\ \overline{x_4}$ is represented as a clist { 110X, 0X00 }. In the symbol cube notation used in the clist entries a 1 means that variable represented by that position is present in the cube, a 0 means that the negation of the variable is present in that position, and an X means that the variable represented by that position is not present in the cube. Note that this same clist also represents the ESOP $x_1\ x_2\ \overline{x_3}\ \oplus\ \overline{x_1}\ \overline{x_3}\ \overline{x_4}$ and the POS $(x_1 + x_2 + \overline{x_3}) \cdot (\overline{x_1} + \overline{x_3} + \overline{x_4})$. Products of SOPEs (PSOPEs) are also used for the *Generalized Propositional Formulas* from [16]. They are represented as *clists of clists of cubes* called *cclists* (special separator cubes are used to separate lower level clists in the higher order cclist). In summary: a cube is used to represent one level of logic (a set of literals and the linking operators), a clist is used to represent two level logic expressions, and a cclist is used to represent three level logic expressions. The individual chips in CCM2 operate on cubes and short clists. Operations on long clists and cclists are executed by setting up pipelines between CCM chips and/or memories and/or the Host.

Cube calculus is a set of operations on cubes, clists, and cclists. *Cube Operations* are of several kinds: *Cube Operators, Cube Predicates,* and *Counting Operations.* The *Cube Predicates* take cube(s) and return logic values 0/1 which are used in the following operations. The *Cube Operators* take one or two *operand cubes* at a time, create from 0 to N *resultant cubes* and insert these in the output clist. The *Counting Operations* take cube(s) and return integers which are the result of counting of the selected satisfied relations between variables of the cube(s). An example of a counting operation is determining the Hamming distance of two cubes. Operations on clists and cclists are called *Clist Operations.* Each Clist Operation has two kinds of components: **(1)** *list operation*, or the kinds of generic transformations that can be done to a list of arbitrary elements or to several such lists. **(2)** *cube operation*, or the kinds of transformations that can be done to a cube or to several cubes. Clist operations are created from all possible combinations of list operations and cube operations.

For later examples of cube calculus operations SOPE expressions will be used where the literals of a cube are linked with an AND operator and the cubes in a clist are linked with an OR operator. However, these operators can be also applied to clists representing POS expressions. As a quick example, binary consensus for SOPE is the same as the "resolution" operation for POS in automatic theorem-proving. In fact, *every* cube calculus operation can have many different interpretations, depending on whether it is applied to a SOPE, POSE, or ESOP. The number of possible operations is very large and we have not yet found practical applications for many of those we have discovered.

By the *base of logic machine* we mean the number of bits used to represent a *simple symbol* in that machine. CCM2 has a base of two (K=2), which allows us to realize matrices of all logic operators in logics with not more than $2^2 = 4$ values. The four simple symbols for base 2 are: 0 *(negated variable)*, 1 *(positive variable)*, X *(don't care)*, and ϵ *(contradiction)*. They are encoded in *positional notation* as 10, 01, 11, and 00, respectively. By *a W-input K-base universal cell* we understand a logic function with W inputs and one output, each input or output being a base K signal. It means that when multiple-valued logic is realized using binary signals, one has K wires to represent each simple symbol from a set of 2^K symbols. A *Universal Cell* of base K realizes all matrices of *base* 2^K *universal logic*, or base *K Set Logic* from [1]. In CCM each simple symbol is processed by an iterative cell IT. K-base symbol requires K-base IT cell. As an example, with our encoding the cube $bc\overline{d}$ $= [11-01-01-10]$. One advantage of this notation is that the intersection of two cubes representing products of literals corresponds simply to a bit-by-bit product of the respective words. For instance, assuming 4 binary variables, (a, b, c, d), the product of $ab \cdot bc\overline{d} = [01-01-11-11] \cdot [11-01-01-10] = [01-01-01-10] = abc\overline{d}$. When the opposite literals are multiplied, the pair 00 is created from the bit-by-bit product and is detected in the next stages: $ab \cdot a\overline{b} = [01-01-11-11] \cdot [01-10-11-11] = [01-00-11-11] = 1\epsilon XX = contradiction$. The contradiction is detected and signaled.

Binary "cube calculus" [7] was extended for a logic with multiple-valued inputs by Su and Sasao [40], [E46-EE49,E44]. For multi-valued input logic, the positional notation uses for each variable as many bits as that variable can have values. For instance, 4 bits are used to represent a 4-valued variable. Assuming one variable with 4 values and a second variable with 6 values, the product ${X_1}^{A_1}\ {X_2}^{A_2} = {X_1}^{\{0,1,2\}}\ {X_2}^{\{1,3\}}$ is represented as a cube A = $[A_1,\ A_2]$ = [1110-010100]. In a CCM of base 2 each variable can have an arbitrary, but even number of values; 2,4,6,8,... In a CCM of base 3 each variable can have an arbitrary, but divisible by three number of values; 3,6,9,12,...

3.2 Examples of Operations on Cubes

In this section we will show formulas for cube operations on multiple-valued input variables. We have found that operations on cubes fall into three groups. *Intersection* and *supercube* belong to the first group, which we call *simple combinational operations.*

The definition of the *intersection operation* for cubes A and B in positional notation is:

$A \cap B =$ *(if there exists such i that* $A_i \cap B_i = \emptyset$ *then* ϕ *else* $[A_1 \cap B_1, ..., A_N \cap B_N]$ *)*,

where: A_i is the i-th literal (*position*) of cube A, $A_i \cap B_i$ is a set intersection of sets A_i and B_i in positional notation; $\emptyset$ denotes a vector of zeros: 00...0 with as many bits as variable i has values (i.e. as may ϵ's as variable i has ITs). ϕ is an empty set (which is signaled by the machine in some way other than by creating an empty cube).

Example 3.2. Intersection of cubes A = $X^{\{0,1,2\}}$ $Y^{\{1,2,3\}}$ = [1110000-0111000] and B = $X^{\{0,1,3,6\}}$ $Y^{\{0,2,5\}}$ = [1101001-1010010] is cube C = $X^{\{0,1\}}\ Y^2$ = [1100000-0010000].

The second group of operations, the *complex combinational operations* produce one cube whose literals are calculated by conditional operations on the literals of the operands. For instance, the *prime operation* used in ESOP synthesis is defined as follows:

$A\ prime\ B = {X_1}^{A_1} \cdots {X_j}^{A_j \cup B_j}\ {X_{i-1}}^{A_{i-1}}\ {X_i}^{A_i \cup B_i}\ {X_{i+1}}^{A_{i+1}} \cdots {X_k}^{A_k \cup B_k}\ {X_N}^{A_N}$

where $A_k \cup B_k$ is calculated for all those variables X_k for which the relation $A_k \cap B_k \neq \emptyset$ is satisfied.

Example 3.3. Prime of binary cubes A = $X^0\ Y^1\ Z^1\ V^1$ = [10-01-01-01] and B = $X^1\ Y^0\ Z^0\ V^{0,1}$ = [01-10-10-11] is cube C = $X^0\ Y^1\ Z^1\ V^{0,1}$ = [10-01-01-11].

The third group of operations includes the so-called *sequential operations.* Examples of such operations are crosslink, non-disjoint sharp, disjoint sharp, symmetric consensus, asymmetric consensus, negation, disjoint negation, neighborhood, extension, primary crosslink. All these operations have in common that they can produce more than one resultant cube.

Nondisjoint sharp (sharp) on cubes A and B is defined as follows: $A \# B = ($ if $A \cap B = \phi$ then A else if $B \supseteq A$ then ϕ else $A \#_{basic} B)$, where A $\#_{basic}$ B is defined as follows: $A \#_{basic} B =$

$\{ X_1^{A_1} \cdots {X_{i-1}}^{A_{i-1}} {X_i}^{\neg B_i \cap A_i} {X_{i+1}}^{A_{i+1}} \cdots {X_N}^{A_N}$
$\mid for\ such\ i = 1, ..., N,\ that\ \neg(B_i \supseteq A_i)\}$

By $B_i \supseteq A_i$ we denote the *relation of set inclusion,* i.e. that set B_i includes set A_i, in positional notation: $\forall\ j = 0, ..., p_{i-1},\ \ {B_i}^j \geq {A_i}^j$. Formula $\neg\ (B_i \supseteq A_i)$ is the predicate that is *true* when the relation $B_i \supseteq A_i$ is not satisfied. By $B \supseteq A$ we denote positional cube inclusion, i.e. $B \supseteq A \Leftrightarrow \forall\ i = 1, ..., N\ B_i \supseteq A_i$.

Example 3.4. XXX1 # 111X = { 0XX1, X0X1, XX01}. Let us observe that simple symbol 0, shifted from left to right in the above cubes, corresponds to all values of i for which relation $(\neg(B_i \supseteq A_i)) = 1$ is satisfied.

Each resultant cube has exactly one such literal, which is called an *active literal* of this cube. All active literals from the resultant cubes are called *specific literals.* There are as many specific literals as positions of i for which relation *rel* satisfied, which is, from 0 to N. Let us also observe that while all specific literals are calculated in parallel, the active literals are created sequentially, with creating the resultant cubes.

Sharp is a basic operation used in generation of prime implicants, minimization of two-level and three-level logic, tautology verification, complementation of switching functions, and many other fundamental algorithms of logic synthesis, combinational problems-solving and theorem-proving.

Comparing formulas for operations such as sharp, disjoint sharp, crosslink, symmetric consensus and asymmetric consensus one can easily observe that all sequential operations have the same basic structure. Each operand cube has one active literal which is the literal i. The operations to be performed on other literals depend on their position with respect to this *active literal*. A resultant cube is created *only* when some relation *rel* $(A_i,\ B_i)$ is satisfied for the active literal values $A_i,\ \ B_i$. For example, the relation of intersection being empty, $A_i\ \cap\ B_i\ =\ \emptyset$, is required for crosslink to produce a cube, and negation of inclusion relation, $\neg(B_i\ \supseteq\ A_i)$, is required for the case of asymmetric consensus.

In the operations described above the four considerations which relate to the active literal are: **(1)** the *relation* $rel(A_i,\ B_i)$ that must be satisfied in literal i for this literal to become active and a resultant cube to be created. **(2)** the *operation* executed *on* the active literal i, for example, $\neg B_i\ \cap\ A_i$ for sharp and $A_i\ \cup\ B_i$ for crosslink; **(3)** the operation executed on literals *before* (or right of) the active literal i, for example copying the literals from A_{i+1} through A_N to the resultant cube; **(4)** the operation executed on literals *after* (or left of) the active literal i, for example, copying the literals from A_1 through A_{i-1} for sharp, and from B_1 through B_{i-1} for crosslink, to the respective literals from 1 to $i-1$ in the resultant cube.

One can observe that all those cube operators use a loop for all variables and for various CC operations apply different set relations and operations in *before, active* and *after* positions. This observation is a base of two crucial ideas of CCM: **(1)** executing the lowest level loop of algorithms (*the variable loop*) in hardware, using linear iterative array of cellular automata, **(2)** programming logic functions in those automata using electrically programmable logic, similar to those used in Field Programmable Logic Arrays (FPGAs) [10]. The idea of programmable logic applied to dynamic architectures is that of a processor reconfigurable at the lowest level. These two concepts are responsible for the amazing flexibility of CCM and the large number of operations that can be programmed by simply changing logic functions of predicates, operators and cellular FSMs. The very fast massively parallel architectures from [10] use Xilinx and similar general-purpose programmable devices and are then totally programmable, however, at the expense of the number of chips used. In contrast, our approach is to have part of a circuit programmable, and part fixed.

One can observe and use similar structures for other kinds of operations for which our machine was designed, for instance, morphological image processing. While the above observation allowed to map "variable loop" to one dimension of the CCM (horizontal communication inside a CCM processor), other operation patterns are mapped to pipelined data movements (shifts) between several CCMs connected in a linear array - as in WARP computer of Kung; two-dimensional movements: horizontal (inside CCM) and vertical (between CCMs) - such as in SIMD meshes; or three-dimensional (using as the third dimension RAM memories connected to CCM processors) (as in GAPP and similar image processing architectures). Additional advantages are created by using *Content Addresable Memories (CAMs)* for storing the results of processors. The interested reader can find descriptions of other CC operations, or ones reducible to them and used in CCM2, in [4,15,20,24,30,40], and [E4,E6,E14-E19,E36,E37,E51,E54].

4 Generalized MV Cube Calculus

As said earlier, the Generalized Multiple-Valued Cube Calculus (GMVCC), an extension to MVCC has been developed, which allows to operate on many data types including: Set Logic (Bio-Logic) [1], several multiple-valued input multiple-valued output logics, and symbolic relations.

Each cube of GMVCC is a vector of variables. Each variable X has its *type* $T(X)$ and *length,* denoted by $L(X)$. $L(X)$ = R * K(IT), where K(IT) is the base of the IT cells, and R is a natural number. The types of variables are: *a symbol, a number,* and *a set.*

(1) *Symbols* are used in several discrete combinatorial problems, such as consistent labeling, so the ability to work with symbols is an important innovation of CCM2. In CCM2 there are two kinds of symbols: *simple symbols* and *complex symbols.* Simple symbols are implemented as binary strings and operations on them are efficiently realized as Boolean functions described by matrices. For practical reasons the number of different simple symbols is usually limited. Such systems will be called *Limited Finite Systems (LFS).* Complex symbols are ordered sets of simple symbols, For instance a binary number is a complex symbol of simple symbols 0 and 1. A ternary number is a complex symbol of simple symbols 0, 1 and 2.

Restricting the number of logic values allows efficient operations on symbols. Operations on complex symbols are done by iterative operations on ordered sets of simple symbols using multi-bit *internal variables (iterative variables).* This is realized as *linear iterative circuits* (Unger) of ITs with two kinds of internal variables: *carry signals* going from left to right, and *confirm signals* going from right to left. Operations on two simple symbols can be described by two-dimensional matrices called *operator tables.* Matrices describing sharp, consensus, and other binary operators can be found in [7]. Examples of matrices for various multiple-valued logics can be found for instance in [39] and [34].

(2) *Numbers* in CCM2 are represented as complex or simple symbols. Operations on numbers are realized as LFS. For instance, when K = 2, one is able to realize binary, ternary and quaternary counting. Binary arithmetic operations for arbitrary v K (such as *Addition* and *Subtraction modulo v, Minus, Max and Min,* as well as *Equality* and *Order Relations*) are not decomposable to any groups of bits. They are, however, decomposable to groups of K bits assuming one or more *carry signals,* going from left to right. The same is true about *ternary* or *quaternary arithmetic. Multiplication, Division,* and other operations can be only described by matrices, and only for numbers $\leq 2^K$. For complex numbers these operations are executed sequentially.

(3) *Set types* represent ordered sets, in particular, sets of logic values. These types are used in MVCC. As in the positional cube calculus notation or in Set

Logic: 1 is for an existing element, 0 for a non-existing one. For the values of tuples (relations) we use the set values as follows: 10 = false, 01 = true, 11- value irrelevant, 00 = contradiction. (Let us observe that the Set Operations and Binary Number Operations are the most efficiently realized since they are decomposable to single bits, which means that the same Boolean function can be executed on all pairs of bits of literals).

Each cube can have two parts: *input variables* and *output variables.* We call these the *input-cube* and the *output-cube*. The value of each output variable can be a set (in a set logic) or a number (in a truly mv logic). For instance, in the case of a truly mv logic, each value of the output variable in the output-cube represents the value of that variable for the given input-cube. Thus, [01-0110-1011 – 3,4] means the input cube [01-0110-1011] has value 3 in the first output variable and value 4 in the second output variable.

Additionally, any number of adjacent variables can be combined as a *Variable Group.* [4] *Examples:* Two numbers are combined as a *number interval group.* A *tuple* can be created from four symbols; first (simple) symbol stands for tuple value, second (complex) symbol for name of relation, next two (complex) symbols for elements of this relation [44]. For instance the group 10, $`PARENT, `Mary, `Robert$, denoted as 0 PARENT(Mary, Robert) states that it is not true that Mary is a parent of Robert. Variables in linked groups can be of various types. For instance, assuming set variable PEOPLE = $|||[Mary, Carol, Robert, John, Adam, Julie]|||$ ($L(PEOPLE) = 6$) the set of Mary and John is represented by literal $PEOPLE^{\{Mary,\ John\}}$, or in positional notation |||[100100]|||. Assuming the relations: 1 PARENT(Mary, Robert), 1 PARENT(Mary, Carol), 1 PARENT(John, Robert), 1 PARENT(John, Carol), one can create the relation group: PARENT(100100, 011000) which means that Mary and John are Parents of Robert and Carol. This notation compacts four PARENT relations to a tuple with mv variables.

GMVCC Operations include the following components. The *GMVCC Predicates* check if some relation of operands is satisfied. This relation is set-theoretical in the above examples, but can be arbitrary in general. The most common relations are , , =, ≠, ≥, ≤, ⊂, ⊃, ⊇, and ⊇. They can be either local (in a variable), or global (in a group or a cube). The *MVCC Cube Operators* use set-theoretical operations on pairs of variables of operand cubes to calculate the resultant cubes. These *Symbol Operations* are set-theoretical in the examples from Section 3, but in the GMVCC Cube Operators they can be arbitrary. The set theoretical operations can be realized by Boolean functions of two binary signals (they are simple symbol operations for K=1). The results of Symbol Operations are complex symbols, i.e. compositions of simple symbols. Numerical operations on complex symbols include: +, -, *Max, Min, Truncated_max, Truncated_min.* Numerical operations on simple symbols include: *High_radix_plus, High_radix_minus, Modulo_plus, Modulo_minus, High_radix_multiply, High_radix_divide.* They can produce generalized carry signals to create complex symbols as for instance in *Radix-3 Addition.* Many other symbol operations exist for K 1. The *Counting Operations* count the number of the *occurrences of satisfied relations on variables.* For instance, a numerical value of the *Hamming Distance* of binary cubes A and B is calculated by counting the number of symbols ϵ in cube C = A ∩ B. The distance of mv cubes is calculated by counting the number of symbols ∅ in variables of cube C = A ∩ B. In general, counting serves to evaluate the quality of perfect/imperfect matching of the operand cubes. This allows for the realization of *approximate string matching* and *fuzzy logic* algorithms.

Similarly to MVCC, operations in GMVCC are on cubes, clists and cclists. Operations on cubes are *simple combinational, complex combinational* and *sequential.* However, the relations and operators can be different for each variable, and can be treated differently in different variable groups and in different parts of a cube. It is this flexibility which makes it possible to use GMVCC for the wide range of formal systems listed in the Introduction with our first mention of GMVCC.

5 The Main Idea of the CCM

General purpose processors have simple combinational instructions such as bit-wise OR, so they can efficiently execute some operations, for example, *supercube.* However, these computers require many instructions to perform other simple combinational operations, complex combinational operations and sequential logic operations. On a standard microcomputer even the cube intersection operation is slow, because shifted masks are used to detect contradictory cubes. CCM2 was designed to accelerate combinational logic operations and, more importantly, the *sequential* cube calculus operations such as *sharp* or *crosslink.* Below we will present the main principles that are applicable *to logic machines of any base, K.* Here they are described for K = 2, but the reader can easily modify them for any K.

Each sequential operation can be described by a pattern: $A\ OP_{sm}\ B =$

$$\{\ X_1{}^{aft(A_1,B_1)} \ldots X_{i-1}{}^{aft(A_{i-1},B_{i-1})} X_i{}^{act(A_i,B_i)} X_{i+1}{}^{bef(A_{i+1},B_{i+1})} \ldots X_N{}^{bef(A_N,B_N)} \mid \textit{for all such } i \textit{ that } rel(A_i, B_i) = 1\ \}$$

An important property of functions *before (for short - bef), active* (act), *after* (aft), and *relation* (rel) is that they are K-wise functions, e.g., for K=2, bits C^s, C^{s+1} of the resultant cube of each of the functions $bef^{s,s+1}$, $act^{s,s+1}$, $aft^{s,s+1}$, and $rel^{s,s+1}$ of a simple symbol are dependent only on bits A^s, A^{s+1} and B^s, B^{s+1} of the arguments. A complex symbol which represents the value of variable C_i of length = R * K(IT) is a composition of R simple symbols which are the results calculated in each of ITs representing this variable. The value returned by the rel_i of variable C_i is determined by the function *relation_type*. Relation_type, here is *OR, AND,* or one of many other *Boolean functions* of signals corresponding to *partial*

[4] denotes a group of variables and |||[ ]||| a set in positional representation.

relations $rel^{s,\ s+1}$ for simple symbols. This function is selected for the desired type of CCM operation. An iterative circuit similar to *multi-rail Maitra cascades* is used to realize the *relation_type* function. A similar mechanism is used to create more global relations for groups of variables or cubes. In general, *rel* signals are arbitrary base K signals, so Boolean predicates are not treated in a distinct way.

The first resultant cube for a sequential CC operation is produced for the first *specific* literal selected as the *active* one starting from the left. Later, the next *specific* literal to the right is selected as the *active* one, and the next resultant cube is produced. This procedure is repeated until the last *specific* literal has been selected as the *active* one. When producing a particular resultant cube, all the literals with numbers less than the number of the *active* literal are of the *after* type, all the literals with numbers greater than the number of the *active* literal are of the *before* type. All these operations are *totally executed in hardware* by the iterative network of state machines. Similar general patterns were found for other sequential, simple combinational, and complex combinational operations.

The simple combinational operations of CCM are vector-like extensions of standard computer instructions (logical, numerical, jumps, etc) expanded in a SIMD pattern to separate fields (variables), each of arbitrary length. The complex combinational operations are similar extensions, but they allow us to execute different operations in the fields. They are useful for tagged operations, flag conditional setting, etc. Sequential operations generalize shifts and string matching operations. For instance, various types of shifts can be executed in parallel for all variables, as well as shifts between variables. In addition to standard arithmetical/logical shifts, shifts with unary K-wise operation in the shift loop are possible, that are the generalizations of the K=1 case of the Johnson counter with the controlled EXOR in the shift loop.

The tasks of the Controller Unit (CU) in the Cube Calculus Machine are to: **(1)** generate signals for CC operations, **(2)** generate signals for the CCM chip's pipelined communication with memories, other CCM chips and the host processor. In CCM processor the Iterative Logic Unit (ILU) is a counterpart of ALU. It is a linear structure of ITs. To perform CC operations, the CU has to provide the ILU with the correct control signals to calculate the solution cubes. CCM processor includes also register file and interface circuitry. Some basic sequential CC operations are shown in Table 1. A similar table can be created for all kinds of GMVCC operations. Each table has columns corresponding to the fields of a microprogrammed control unit. The first four operations use the OR-type relation. This is an OR-type relation because if the relation is satisfied in at least one bit h $(B_i{}^h = 0,\ A_i{}^h = 1)$, it is satisfied for the entire variable. The last operator, crosslink, uses AND-type relation. It is global within variable, because for all bits of this variable the resultant bits must be 0.

In conclusion, each CC operation can be described as a combination of programmable logic functions and signals created by the CU. A Microprogrammed CU, combined with distributed control based on communicating state machines (cellular automata) in the ILU, and overall function programmability creates an astronomical number of possible logic/arithmetic/set-theoretic/symbol operations. The overall control of the CCM is hierarchical: the highest control level is software in the Host, the intermediate level is the microprogram of the CCM Controller Unit, and the lowest level is the programmability of logic cells and state machines in every CCM processor chip.

6 Hardware Architecture of CCM2 Processor

All the known software subroutines process the literals sequentially, but for most of the literals the resultant cubes generated will have contradictions and that will have to be removed later. CCM2 implements a completely new architecture to take advantage of the peculiarities of sequential cube calculus operations. The architecture is an iterative logic array (ILU) with "carry" signals running from left to right *and* from right to left through the iterative circuit of *Position State Machines (PSMs).* The fundamental advantage of this approach is that only cubes without contradictions are generated. The CCM2 Processor consists of a set of bit-slice ILU processing units, an interface controller, and a control unit. The processing unit is implemented as an iterative logic array (ILU) of basic *buiding blocks.* A single cell (block) from an ILU is called a *ITerative Cell (IT).*

Basically, CCM2 has a base of two (K(IT) = 2), which allows us to realize matrices of all logic operators in logics with not more than $2^2 = 4$ values. Since it is important to realize resolution/unification operations of theorem-proving [44] and other truly mv logics, base 3 operations can be also realized in CCM2. The CUBEX concept of theorem-proving [44] has three variants: *propositional logic, predicate logic* and *hybrid.* This concept uses new "cross" cube calculus operations (cross-intersection, cross-sharp, cross-consensus, cross-subsume), which require four additional symbols. The total set of symbols for base 3 CCM is then 0, 1, X, ϵ, R, L, Y, Z, which leads to K=3. They are coded: 010, 001, 011, 000, 111, 100, 101, 110, respectively (which is different from [44]).

By *a W-input K-base universal cell* one means a logic function with W inputs and one output, each input or output being a base K signal. It means that when multiple-valued logic is realized using binary signals, one has K wires to represent each simple symbol of a set of 2^K symbols. A universal cell of base K realizes all matrices of *base* 2^K *universal logic,* or base K set logic. In CCM2 each simple symbol is processed by an iterative cell IT. A K-base symbol requires a K-wise IT cell. The number of ITs is denoted by n, so that the number of bits is 2n and we can process n binary variables. Besides combinational logic each IT[i] includes a *Position State Machine (PSM)* that influences the local interpretation of the micro-instructions. In this sense each IT is a small processing unit that processes a part of a cube in parallel and communicates with other processors that are connected in a linear organization. A processor with two ITs of K=2 is shown in Fig. 6.1. For explanation purposes we will divide each IT[i] into four blocks according to the function that it performs: *RELATION[i], STATE[i]* (name of the PSM block), *MATCH_COUNTER[i],* and *OPERATION[i].* We also discuss simplified signals for MVCC operations from section 5 with K=2. **(1)** Block *RELATION[i]* has the task of **identifying** the position of the IT within the literal and generating a Boolean signal *VARIABLE[i]* that is *true* when the IT[i] is a part of a literal that satisfies the selected relation *"rel".* To calculate the value of *VARIABLE[i],* the *RELATION[i]* block uses two iterative signals. *CARRY[i]* is an iterative signal that runs from left to right and is *true* when all ITs of the same literal to the left of the IT satisfy the AND-type relation encoded in *rel. CONF[i]* (confirm) says that at this position all ITs have satisfied the relation. As we see, the signal CARRY goes from left to right, up to the end of variable, and next returns as signal CONF, back to all ITs of this variable. This explanation is only for AND-type relations. Similar explanation for OR-type and other relations can be given. **(2)** Block *STATE[i]* is an FSM essential to executing sequential operations. The state of the *STATE[i]* block represents the position of the IT[i] in relation to the *active* literal. The *STATE[i]* is in state *active* if the IT[i] is a part of an active literal; it is in the *before* state if the IT[i] is to the right of active literal; and in *after* state if the IT[i] is to the left of the active literal. All *STATE[i]* are initialized to the state *before* with the global signal *from CU.* **(3)** MATCH_COUNTER[i] counts the number of satisfied predicates in IT[i]. **(4)** OPERATION[i] creates bits of resultant cubes by performing the operation on bits of the operand cubes. This is where the operator matrix is programmed in.

CCM2 has also support for data-flow communication on clists and two-dimensional data with operations such as *Global Reduction, Mapping, Inversion, All Pairs, Associative Selection, Dot Product, Vector Product, Simple Cartesian Product, Sequential Cartesian Product, Convolution. All Permutations, All Subsets, All Combinations* and other. Various structures of parallel processors can be configured from CCM processor chips, controllers and CAMs: Long Word Processors, Pipelined Processors, Trees, and Pyramides. CAMs allow for fast realization of several associative operations, as well as operations such as sorting and absorbing, which are very common in our class of applications.

In various simulation experiments and program measurements we found that for many logic synthesis algorithms such as tautology, satisfiability, primes generation, set covering, ESOP minimization, up to 98% of time is spend on CC operations and sorting. For instance, a Host operating at 10 MHz was assumed for a tree architecture [16] to minimize Generalized Propositional Formulas. Assuming a tree with three levels (7 processors), in order to fetch all four leaf node processors in every execution cycle, the processors' clock rate was set at 2.5 MHz. With this setting, the architecture can solve a 1000-clause Generalized Propositional Formula in approximately 0.7 milliseconds (i.e. 1698 cycles * 0.4 milliseconds/cycle). On the other hand, a two-level tree, with 2 leaf node processors operating at 5 MHz, can solve the same problem instance in 1.4. milliseconds (i.e. 3470 cycles * 0.4 milliseconds/cycle). To solve this problem instance by simulating the same scheme a conventional PC-compatible computer, operating at 10 MHz, requires 70.8 and 76.7 seconds, respectively. Such results are encouraging and give hope that even with a small number of processors the machine will give significant speedup for several applications.

7 Conclusion

The paper briefly introduces new Generalized Multiple-Valued Cube Calculus and the basic features of our Cube Calculus Machine CCM2, the first universal multiple-valued logic machine. CCM2 architecture has been designed for pipelining, local electrical programmability, K-wise serial processing, two-dimensional systolic processing, associative processing, and microprogramming. The innovative property of CCM is a data path based on communicating cellular automata that executes GMVCC operations. The CCM2 bit-slice chip and CCM2 processor board are currently under development at PSU.

We predict that logic machines, which have a history longer than their arithmetic counterparts, also have a bright future in a world that is increasingly dependent on real-time intelligent processing of symbolic information.

Acknowledgments

I would like to thank Professors Andrzej Goralski, Tsutomu Sasao, Ryszard Michalski, and Robert Brayton for their help in finding background information; Professor Malgorzata Marek-Sadowska for her constructive criticisms, and Mr. Douglas V. Hall for his help in organizing and editing this paper. I would also like to thank my many students who have worked and continue to work on the CCM.

References

[1] T. Aoki, M. Kameyama, and T. Higuchi, *Proc. 1989 ISMVL,* pp. 36-367, May 1989. **[2]** A.A. Arkhangelskyaya, et al., *Prob. Pered. Infor.,* No. 6., pp. 5-23, 1960. **[3]** D. Bobrow, "A Symbolic Logic Machine to Minimize Boolean Functions of Four Variables, and Application to Switching Circuits", 12 pp., privately printed manuscript, Bronx 1952. **[4]** F.M. Brown, "Boolean Reasoning. The Logic of Boolean Equations", *Kluwer Academic Publishers,* Boston/Dordrecht/London, 1990. **[5]** B. Burack, *Science,* Vol. 109, June 17, 1949, p. 610. **[6]** A.W. Burks, D.W. Warren, and J.B. Wright, *Mathematical Tables and other Aids to Computation,* Vol. 8., April, 1954, p. 53. **[7]** D.L. Dietmeyer, "Logic Design of Digital Systems", Allyn and Bacon, 1971. **[8]** P. Dysko, "Implementation of MULTICOMP System in FORTRAN", M.S. Thesis, Techn. Univ. of Warsaw, 1978 (in Polish). **[9]** M. Fujita, Y. Tamija, Y. Kukimoto, and K-Ch. Chen, *Proc. ICCAD'91,* Nov. 1991, pp.510-513. **[10]** Proceedings of the FPGA'92

Workshop, Berkeley, 16-18 Febr. 1992. **[11]** M. Gardner, "Logic Machines and Diagrams", *McGraw-Hill,* 1958. Harvester, Brighton, 1968. **[12]** M.A. Gavrilov, and A.D. Zakrevskii (eds.), "LYaPAS: A Programming Language for Logic and Coding Algorithms", *Academic Press,* New York, 1969. **[13]** G.B. Gerace, et al, *IEEE TC,* Vol. C-20, pp. 837-842, Aug. 1971. **[14]** A. Goralski, "Heuristic Methods to Solve Problems", *Scientific-Technical Publishers,* Warsaw, 1978 (in Polish). **[15]** P.L. Hammer, and S. Rudeanu, "Boolean Methods in Operations Research", *Springer-Verlag,* New York, 1968. **[16]** P.M. Ho, and M.A. Perkowski, *Proc. ISCAS'89, pp. 1170-1173, 1989.* **[17]** L. Kida, M.A. Perkowski, "The Cube Calculus Machine: A Ring of Asynchronous Automata to Process Multiple-Valued Switching Functions", *Proc. ISCAS'92,* San Diego, May 1992. **[18]** L. Kida, "Associative Processing Implemented with Content-Addressable Memories", *M.S. Thesis,* PSU, 1991. **[19]** H. V. D. Le, and M. A. Perkowski, *Proc. of ISCAS'90,* New Orleans, 1-3 May 1990, pp. 2312-2315. **[20]** E.B. Lee, and M. Perkowski, *Proc. of the 1984 Intern. Conf. on Syst. Man, and Cybern.,* Halifax, CA, Oct. 9-12, 1984. **[21]** K. Lieberherr, and E. Specker, "Complexity of Partial Satisfaction", Manuscript, 1978. **[22]** M.A. Marin, "Investigation of the Field of Problems for the Boolean Analyzer", *Ph.D. Dissertation,* UCLA, 1971. **[23]** W. Mays, and D. Henry, *Mind,* LXII, 1953, pp.484-505. **[24]** R. S. Michalski, "A Planar Geometrical Model for Representing Multi-Dimensional Discrete Spaces and Multiple-Valued Logic Functions", *Report No. 897, Department of Computer Science,* University of Illinois, Urbana, January 1978. See also in [34]. **[25]** M.A. Perkowski, *in "Artif. Intell. and Patt. Recogn. in CAD",* J. C. Latombe (ed.), North Holland, Amsterdam, pp. 124-140, 1978. **[26]** M.A. Perkowski, "General methods for solving combinational problems". Chapter 4 in A. Goralski (ed.) *"Problem, method, solution", Vol. 4,* Scientific-Technical Publishers, Poland, Warszawa 1982 (in Polish). **[27]** M.A. Perkowski, *Proc. ICCAD'85,* Santa Clara, 19-21 Nov. 1985, pp. 133-135. **[28]** M.A. Perkowski, "Logic Design Machine", *Carnegie Mellon University, Department of Electrical Engineering,* Invited Lecture, 1986. Also NSF proposal. **[29]** M.A. Perkowski, J. Liu, J. Brown, *In G. Zobrist (ed) "Progress in Computer Aided VLSI Design",* Vol. 1., Ablex Publishing Corp., 1989, pp. 353-401. **[30]** M.A. Perkowski, "Multiple-Valued Universal Logic Machine", *U.S. Patent Application.* **[31]** M.A. Perkowski, and A. Sarabi, "History of Logic Machines. Part 1: from Lullus to 1985", *PSU Report,* 1991. **[32]** M.A. Perkowski, "The Generalized Orthonormal Expansion of Functions With Multiple-Valued Inputs and Some of its Applications", *this Proceedings.* **[33]** V. Pratt, "Thinking Machines. The Evolution of Artificial Intelligence", *Basil Blackwell, Inc.,* Oxford, U.K., 1987. **[34]** D. Rine, "Computer Science and Multiple-Valued Logic. Theory and Applications", *North-Holland,* 1984. **[35]** V.I. Rodin, *Avt. i Telem.,* Vol. 18, No. 5., pp. 437-443, 1957. **[36]** T. Sasao, *Proc. ICCD'85,* pp. 713-718, Oct. 7-10, 1985. **[37]** C.E. Shannon, and E.F. Moore, *Proc. IRE,* Vol. 41., pp. 1348-1351, Oct. 1953. **[38]** D. Shumake, "The MOS Boolean Analyzer", *M.Sc. Thesis,* UCLA, 1971. **[39]** A. Stern, "Matrix Logic", *North-Holland,* 1988. **[40]** S.Y.H. Su, and P.T. Cheung, "Computer Minimization of Multivalued Switching Functions", *IEEE Trans. on Comp.,* Vol. C-21, No. 9, p. 995, September 1972. **[41]** A. Svoboda, *Proc. Inform. Proc. 68,* Amsterdam, North-Holland, 1969. pp. 824-830. **[42]** A. Svoboda, *IEEE TC,* Vol. C-22, No. 9, pp. 848-851, Sept 1973. **[43]** A. Svoboda, and D.E. White, "Advanced Logical Circuit Design Techniques", *Garland STPM Press*, New York, 1979. **[44]** M.E. Ulug, *Proc. IEEE Intern. Conf. on Comp. and Comm.*, Arizona, 1985, pp. 292-297. **[45]** M.E. Ulug, "A Real-Time AI System for Military Communications", *Proc. of the Third IEEE Conference on Artificial Intelligence Applications*, February 1987, Orlando, Florida. **[46]** H. Yasuura, T. Tsujimoto, and K. Tamaru, Using Content Addressable Memories", *Proc. ISCAS'88*, pp. 333- 336. **[47]** A.D. Zakrevskii, *Vyc. Tekh. Avt. Teorii. Inf.,* No. 42., pp. 9-37, 1963.