THE CUBE CALCULUS MACHINE: A RING OF ASYNCHRONOUS AUTOMATA TO PROCESS MULTIPLE-VALUED BOOLEAN FUNCTIONS

Luis S. Kida, Marek A. Perkowski

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

ABSTRACT

The paper proposes a Cube Calculus Machine (CCM-1), a new architecture in which the data path has been designed to execute operations of "cube calculus", an algebraical model popularly used to process and minimize Boolean functions. The machine uses a "positional cube representation" which can also represent the multiple-valued input algebra that finds recently many applications in logic synthesis. Another aspect of this architecture is the implementation of the processing unit as an iterative network of asynchronous FSMs. This is a new concept in computer architecture that can find applications wider than the CCM alone. CCM-1 realizes basic micro-instructions that support the microcode implementation of all useful cube calculus operations including sharp, consensus, supercube and crosslink.

1. INTRODUCTION

There is recently an intensive and growing interest in logic synthesis, both in the theory and in the realization of practical design programs and systems of CAD tools. Several electronic design companies use commercially available logic synthesis tools (like those from Synoposis or Mentor). Logic synthesis will become even more important with the proliferation of FPGA-based dynamically reconfigurable multi-chip architectures, configurable from high-level specifications [2]. However, good quality logic optimization programs are very slow. To speed up the logic synthesis and combinatorial algorithms, it was proposed to design special computers and hardware accelerators [6,9,12,15,16,18,26,28,30-38]. What all those machines have in common is that they use special hardware to do some kind of processing of Boolean functions: evaluation, Boolean operations such as intersection or complementation, checking for tautology or satisfiability, verification, resolution, etc. Since all NP-complete problems [5,10,1,3] can be polynomially reduced to one of the above problems, and particularly to the 3-SAT Problem, the proposed by us *general problem-solving methodology* is to reduce any problem to some *consistent labeling problem* (such as graph coloring or set covering) and next reducing it to some manipulations of multiple-valued (mv) logic functions (such as sharp or resolution). For instance, PLA minimization problem is reduced to primes generation and set covering. The second problem is reduced to Petrick function minimization, which is a particular case of satisfiability and can be solved in many ways on CCM-1, for instance, by using sharp operation.

Instead of numbers, the proposed by us co-processor, CCM-1, processes *cubes*, the basic units of the *"cube calculus" (CC)* which is popularly used to represent and process Boolean functions. CC is used in most of the efficient modern logic synthesis programs, including the well-known Espresso and MIS II [1,4,20,21]. It has been extended for a logic with multiple-valued inputs [23-27] by Sasao, and is called a *positional cube notation*. This calculus has been used for many two-, three- and many level Boolean minimizers, verifiers, programs for complementation of Boolean functions, synthesis of mixed and fixed generalized Reed-Muller forms, generation of prime, minimal and disjoint implicants, spectral transforms (Walsh, Reed-Muller, Arithmetic), automatic theorem-proving [32-35], image processing, and many other [1-5,7,8,10,11,17-22].

2. THE IDEA BEHIND THE CCM-1 ARCHITECTURE.

Let us denote by 2n the number of bits of a word (register) that contains a cube in positional notation. To focus our considerations, we assume 2n = 32 bits. We can have as many as n = 16 binary variables in a cube. The encoding is as follows: x - 01, $\bar{x}$ - 10, don't care (often denoted by X) - 11, *contradiction* - 00. In this notation, the intersection of two cubes representing products of literals simply corresponds to a bit-by-bit product of the respective words. For instance, assuming 4 binary variables, (a, b, c, d), the product is $a\bar{b} \cdot bc\bar{d}$ = [01–01–11–11] · [11–01–01–10] = [01–01–01–10] = $abc\bar{d}$. When the opposite literals are multiplied, the pair 00 is created from the bit-by-bit product and is detected in the next stages: $ab \cdot a\bar{b}$ = [01–01–11–11] · [01–10–11–11] = [01–00–11–11] = *contradiction*. The contradiction is detected and signalized. For multi-valued input logic, the *positional notation* takes for a variable as many bits as this variable can take values. For instance, a 4-valued variable takes 4 bits. Assuming the first variable of 4 values and the second variable of 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]. Assuming 4-valued variables and 2n = 32, one has 8 variables in a cube. It is assumed in CCM-1 that each variable can have an arbitrary even number of values, possibly different from other variables.

Careful analysis of all cube calculus operations revealed that they can be divided into three categories according to the type of result [11,18]. (1) In the *first category* there is a single cube output, so-called *resultant cube*, for one or two *operand cubes*. The resultant cube can be combinationally generated with bitwise logical operations like AND, OR, XOR, copying the variable, etc. independently of other bits. This category includes the operations of supercube and intersection [1,19,20,21], and we call them *simple combinational operations*. In this category the operation on two cubes is a concatenation of operations on all positions (respective pairs of literals of the operands). Contradiction is also detected and signalized so that cube with contradiction is not generated at all. In this category each variable C_i from the resultant cube C is: *(2.1)* $C_i = f(A_i, B_i)$, where f is some *set function*, and A_i, B_i are variables of input cubes A, B. (2) The *second category* of operations includes the *complex combinational operations*. For each pair of *operand cubes* there is a single resultant cube. This cube is a bitwise logic operation which is, however, **conditioned for each literal by some relation/pattern of the input cubes.** This includes operations prime, double prime and binary consensus [11,18]. In this category each variable C_i from the resultant cube C is: *(2.2)* C_i = *if rel* (A_i, B_i) *then* $f_1(A_i, B_i)$ *else* $f_2(A_i, B_i)$, where *rel* is some set-relation and f_1, f_2 are some set functions. (3) The *third category, sequential logic operations,* has multiple resultant cubes for one or two operand cubes. Examples of these instructions are crosslink [7,17], consensus [25,11], sharp and other [7,8,17,19,21-27]. Specific position j is one for which $rel(A_j, B_j)$ is satisfied. A literal of the resultant cube is conditioned by the position of this literal relative to other literals and the current specific position. The current specific position in the cube will be called the *active position*. In this category the number of the resultant cubes is equal to the number of times that a certain relation *rel* is satisfied for a pair of respective literals of the operand cubes, which means, to the number of specific positions. Let us observe that the active position during the generation of the i-th resultant cube is the i-th from left among the specific positions of the cube. The general pattern of third category operations is:

(2.3) $C = \{C_j \mid C_j = X_1^{aft\_act(A_1,B_1)} \cdots X_{j-1}^{aft\_act(A_{j-1},B_{j-1})} X_j^{act(A_j,B_j)} X_{j+1}^{bef\_act(A_{j+1},B_{j+1})} \cdots X_N^{bef\_act(A_N,B_N)}$ for *each such j that* $rel(A_j, B_j) = 1\}$,

where: C is the set of resultant cubes, aft_act, act, and bef_act are set functions corresponding to *before active, active*, and *after active* positions in the cube, respectively. N is the number of variables in cubes. The architecture of CCM-1 results from an attempt to optimize the execution of the three above types of operations; it directly implements equations (2.1) - (2.3).

3. THE ARCHITECTURE OF THE CCM-1.

All the known software subroutines process the literals sequentially, but for most of the literals the resultant cubes generated will have contradictions and will have to be removed later. In the beginning we considered implementing a RISC processor specialized in sequential logic operations with all control hard-wired that would process a literal of the cube in each pulse of a fast clock. However, after some thought, we took a completely different approach. More than just doing a direct translation of the subrou-

0-7803-0593-0/92 $3.00 1992 IEEE

tines that implement those operations to micro-instructions, we created a completely new architecture to take advantage of the peculiarities of the sequential cube calculus operations. The architecture is an *Iterative Logic Array (ILU)* with iterative signals running from left to right **and** from right to left of the iterative circuit of *Asynchronous State Machines (AFSMs). The fundamental advantage of this approach is that only cubes without contradictions are generated.* This is a general speed-up method by realizing the lowest level loop of the algorithm in hardware.

The ILU recognizes the next specific (active) position and generates a resultant cube in each cycle. It realizes, using internal distributed control, the lowest level iterative loop, as described by formulas (2.1), (2.2), and (2.3). Therefore, ILU does not need the control unit to execute the basic cube operations: while generating the resultant cube the role of CU is limited to generating signals *ACTIVATE* and *REQUEST*. ILU is controlled by two types of signals, *iterative signals* and *global signals*. Two of those signals, global *REQUEST* and iterative *ACTIVATE*, work in an "interlock mechanism" that substitutes the "clock" of synchronous machines with a "two-phase non overlapping rippling waveform" of asynchronous automata. An analogy that helps to understand the advantages of this architecture over the sequential processing of all literals is to imagine each literal as a domino tile. The linear iterative array has all dominos lined up in a way that if the first one falls, all next will fall in sequence. For the specific positions the correspondent domino tile is removed. This way, when the control unit pushes the first domino, the domino tiles will fall in a "domino effect" until they reach the gap left by the specific position, where the domino effect will stop. At that point the literal is processed and an output cube is generated with each of its literals being a function of its position in the array. If its domino has already fallen down it corresponds to state *aft_act*, if the literal is being processed, it corresponds to state *act* (active), when its domino is still standing, it corresponds to state *bef_act*. The control unit begins the cycle again by pushing the first remaining domino until there are no more standing dominos left. In fact the iterative circuit has a ring configuration and the control unit serves as the first and last domino. This way, it is simple for the control unit to observe the fact that all literals have been processed, without the need to keep track of which literal is being processed and how many remain to be processed.

A co-processor that processes a literal at a time would push a single domino for each cycle and try to improve performance by increasing the rate at which it processes each literal. Among the disadvantages of such a method is that it generates cubes with contradiction that have to be removed. If a circuit to recognize and remove cubes with contradiction were integrated to the architecture, then the rate of generation of resultant cubes would be irregular. The rate of generation of resultant cubes in CCM is regular, making it suitable for pipelining and systolic processing, which allows to build large parallel structures from them and was one of the main objectives of our approach. A great part of the control task is distributed in the asynchronous machines.

4. HARDWARE MAPPING OF CCM-1 ARCHITECTURE.

The CCM chip consists of a processing unit, an interface controller, a register file and a control unit (Fig. 1). The processing unit is implemented as an iterative logic array of basic *buiding blocks IT* creating an *ILU*. Each IT includes besides combinational logic an *AFSM* that modifies the 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. Each IT processes two bits, i.e. a binary variable or part of an mv variable.

The micro-programmed *Control Unit (CU)* of the CCM receives a code of a high level cube calculus operation *(CCM instruction)* in the *Instruction Register (I)* and translates it into simpler basic operations implemented in the processing unit. In the execution of the sequential instructions the CU behaves as the first IT of the line, IT[0], and as the last IT, IT[n+1] (n is the number of IT cells). *Bus Interface Unit (BIU)* handles the communication between the system and the CCM. The communication between the BIU and the ILU is done through shared registers/memory, as is the communication between the BIU and the CU. The interfaces between the BIU and the ILU and the CU were made independent, asynchronous and through a protocol to let the design of the ILU and CU be independent of the BIU. The same CU and ILU can be integrated to different systems by redesigning the interface (BIU) only, and the basic design of the interface for one particular system can be used over and over for different versions of ILU and CU. The shared data register file accounts for storing the input cube, the output cubes, and the intermediate results. Usage of this file prevents also a loss of performance due to the differences in the processing rates of the system and the ILU to feed the input operand cubes, generate the resultant cubes, and transfer them out of the CCM.

There are other registers used for communication and control, among them: *Multi-value (M), Status (S), Instruction (I)*. The Status register stores information useful to support programs in the host processor: the distance of cubes transmitted to the iterative logic unit, the number of resultant cubes, flags and semaphores to signalize when the resultant cube is ready or when there is no result. The register I controls additionally the data transfer and the iterative signals of the first and the last IT cells of the ILU. The Multi-value register specifies the number of ITs used by each literal of the cube. There is one bit in the M for each IT of the ILU plus two for the control unit that emulate IT[0] and IT[n+1]. For example, when the content of M is 10011010, the first bit is for IT[0], the following 2 bits represent a 4-valued variable and will use the first 2 ITs, the second 2 bits are used by another 4-valued variable and will take another two ITs, the third and the fourth variables are 2-valued (binary) and will use only one IT cell each, the last bit is for IT[n+1].

In section 4.1 we will describe the basic timing of the active position propagation in ILU. Section 4.2 illustrates the creation of basic internal and output signals on a simple example.

4.1. Description of the ILU and CU.

We are now ready to understand the execution of a sequential operation by the ILU and CU. Fig. 2 illustrates the execution of a sequential operation, with the stable state of the ITs of the literals (represented as boxes) after the propagation of the signals showed on the left column. Sequential operation begins by loading the operand cubes to let blocks *IDENTIFY[i]* to recognize all the specific positions. The CU keeps the generated by it global signals *REQUEST* and *INITIALIZE*, as well as the initial signal *ACTIVATE[0] false* and lets all signals *VARIABLE[i]* in ITs reach their final values. This is to ensure that only a single *AFSM* input can change, which is necessary to avoid all combinational hazards that might be dangerous in the *AFSM*, being an asynchronous machine. The interval of time before CU is ready to do something next has to be long enough for the IT[n] to have the *VARIABLE[n]* signal stabilized (delay of 1 IT for a binary variable, 2 IT for a 4-valued variable,...). This is represented in Fig. 2a. At the end of this phase all specific positions are marked as values 1 of *VARIABLE* in corresponding ITs.

The CU resets the ILU to its initial condition by asserting *INITIALIZE*, Fig. 2b). By *bef* we denote the state *bef_act* (which stands for before active) of *AFSM*. The control unit does not have the control over signal *VARIABLE* but since A, B and M remain stable, *VARIABLE* will remain stable as well. Again, it is ensured that there is a single transition in the inputs of the *AFSM*. After waiting for all IT cells to reset, the CU deasserts signal *INITIALIZE*, Fig. 2c).

The execution of the instruction really begins by the assertion of *ACTIVATE* for the leftmost IT, *ACTIVATE[0]*, to *true*. The first literal that has *VARIABLE = true* will become active. (If many ITs are used to represent a literal, all of them will have *VARIABLE* = 1, and all will become active). That can be the first literal(Fig. 2d.1) or *ACTIVATE* may ripple through one or more literals (Fig. 2d.2). By *act* we denote the state *active* of *AFSM*. By *aft* we denote the state *aft_act* (after active) of the *AFSM*.

After waiting for the signal *ACTIVATE* to propagate to reach the end of the string of IT cells and allow the *AFSM* to reach its new stable state, CU samples *ACTIVATE[n]*, if it is *true* there are no resultant cubes to generate and the operation is finished, if it is *false*, it means that the signal *ACTIVATE* has been stopped by a specific position and the CU has to store the resultant cube in the data register file. In the latter case the CU deasserts the signal *ACTIVATE* to prepare for the next cycle(s), (Fig. 2e).

After waiting for the output to stabilize and latch the result, the CU makes the *REQUEST* signal *true* to prepare for the next cycle. This will make the ITs in the active position to transit to the *aft_act* state and let the next *ACTIVATE* signal pass for the IT cells after it (Fig. 2f).

The CU deasserts *REQUEST* to prepare for the next cycle (Fig. 2g).

A new cycle can begin for those literals in state *bef_act* after the IT cell that changed from active state to *aft_act* in d.1 and d.2. Such process is iterated until signal *ACTIVATE = true* passes the entire ILU and, as *ACTIVATE[n]*, reaches the CU; the operation is finished.

4.2. Example of execution of one sequential instruction.

Now, when we understand the basic timing, we will show a complete example to illustrate the creation of signals in ILU. For this example the CCM has 6 ITs organized in three variables (M=10001001). The first variable, U, has 6 values, the second variable, V, is binary and the last variable, Z, is 4-valued (Fig. 3b). The inversion of operand cube A will be executed, which will generate 2 resultant cubes, s and S. This cube complementation is

illustrated in a map from Fig. 3b: $\overline{U^{1,2,3}Z^{12}} = \overline{U^{1,2,3}} + \overline{Z^{1,2}} = U^{0,4,5} + Z^{0,3} = s + S$. Table 1 includes the signals of ITs for the example from Fig. 3b. The header row includes the names of the signals. The columns for i=1,2,3 correspond to variable U, the columns for i=4 to variable V, the columns for i=5,6 to variable Z. The value of index *i* is in the row # 1. Its corresponding IT symbol is in row # 2. The respective bits of the input cube A are shown in row # 3. Since the relation checked is A=X (where X is denoted by 11), then signals *RELATION[i]* are satisfied in IT[2] and IT[4]. This is shown in the row # 4. Signals *LEFT* and *RIGHT* together with *RELATION* are used to create signal *VARIABLE* in each IT. If there is a signal *RELATION* = 0 within a literal, then in all ITs of this literal *VARIABLE* is 1 (see row # 7). Signal *COUNT*, initially set to 0 by CU, counts the number of literals in which signal *VARIABLE* is 1, i.e. the number of specific positions. This is propagated from left to right and value "1" is added at the frontier of the previous and next literal (see row # 8). Rows # 9 and # 10 present the resultant cubes generated.

The Fig. 3a shows a location of three signals: *VARIABLE*, *state* and *C* within IT cells from Fig. 3c. Fig. 3c presents six snapshots of the ITs states during calculation of the resultant cubes s and S. The first three ITs correspond to variable U, the next IT to variable V, and the last two ITs to variable Z. In each of the six snapshots, on the left we have the state of signals signals coming from CU: *ACTIVATE[0]* and *REQUEST*. States of signals *VARIABLE*, state and C are shown inside boxes. Let us observe that signals *VARIABLE* remain stable during propagation of *ACTIVATE* signals. The state of signals *ACTIVATE[i]* are shown between IT boxes, and the final signal *ACTIVATE[n]* to CU is shown at the right.

The chain of ILUs will pass the "token" *ACTIVATE* to generate the resultant cubes. In the second snapshot the resultant cube *s* has been generated, which can be seen as the contents of registers C. In snapshot four the second resultant cube, *S*, has been generated. Notice, that the second variable of cube A passes the token without becoming active and without generating a resultant cube, as it was already explained in section 4.1. The last snapshot illustrates signal *ACTIVATE[n] = 1* reaching the CU, which ends the execution of the operation.

Iterative structure of three ITs is shown in Fig. 4. For a more detailed description of the IT and the VLSI realization see [11].

5. DETAILS OF THE ARCHITECTURE OF THE CCM-1.

Some of the CCM features will be now discussed that have been skipped in section 4 in order not to overshadow the general idea in details.

To guarantee its correct operation, the *AFSM* cannot have hazards. An extra state *no_race* was added to guarantee state transitions without hazards. The state *no_race* and register W are also used for increasing the testability and fault tolerance of the CCM.

Water (W) register stores one bit of *enable* for each IT. When a bit of W is *true* the corresponding IT repeats the received iterative signals and has C=11, the *AFSM* is stuck in state *no_race*. Having all bits of W *true* except for W[k] makes the iterative signals of IT[k], that otherwise would be "buried" in the ILU, **observable and controllable** because all IT[i], i ≠k are *"transparent as water"* for the signals of IT[k]. Register *Water* is also used when the number of ITs in the ILU is larger than what is required by the size of the cube. In this case the extra ITs are made inocuos or transparent for the operation of the ILU. In case a fault is found in the IT that does not prevent the IT to go to the transparent state this IT can be eliminated, thus preventing the loss of the chip because of a fault in one of the ITs.

Another important control register is the *Mode(D)* register. *Mode* stores a variable that modifies the interface of the CCM and the operation of CU. At the moment, there are two modes of CCM operation:
(1) *stand_alone* mode, when the CCM works independently, the major characteristics is that all "initial" iterative signals like *LEFT[0]* and *RIGHT[n+1]* come from a register in the CU and the "final" iterative signals like *LEFT[n]* and *RIGHT[1]* are latched by CU in a control register.
(2) *chain* mode, when CCMs can be chained, replicating in a higher level the architecture of the ILU where each CCM replaces a part of the ILU, using the ILUs of the chained CCM to emulate a "longer ILU". According to the data stored in register D, the CCM is configured to be the *last, first or internal* CCM. For the internal CCM the initial iterative signals come from pins and the final iterative signals are output to pins as well.

6. DESCRIPTION OF SYSTEM ENVIRONMENT OF CCM-1.

The CCM is a processing element specialized in cube calculus logic operations. It can be compared to DSP modules from which large data-flow architectures can be configured. It was designed to work as a *co-processor* to accelerate cube calculus operations. A *host processor* is required for control and initialization.

There are two options to execute sequential operations on operand cubes which are larger than the capacity of the ILU. One option is to cascade CCMs in the chain mode and emulate a larger ILU. The other is to split the cubes into *pieces*. The trick to split the cubes and keep the performance is not to split literals. In the way the signals *RIGHT* and *LEFT* were designed, the ILU is broken into small iterative circuits for each literal and there is no propagation of iterative signals in the block *IDENTIFY* beyond the boundary of the literal.

Breaking signals *ACTIVATE* has no performance penalty, as long as no literal is split, because only one literal is processed at a time and the active variable is the breaking point for block *SIGNALIZE*. Pieces of the cube that are "before" and pieces that are "after" have to generate an output that is the result of a simple logic operation and the host processor can "fill in the blanks". To process the literals with more values than the capacity of the ILU, the CCM has to be in the *stand_alone* mode. To process a literal, all the pieces must be loaded in order and the iterative signals from one piece of the literal are stored to be used in the following piece. There is a small speed penalty because all pieces have to be run once, either from left to right or from right to left, to find the values of the iterative signals *LEFT/RIGHT*. In this case we are sub-utilizing the architecture to just "identify" the variable and the *AFSM* is not used.

7. CONCLUSIONS AND EXTENSIONS TO CCM-1.

A prototype of IT was designed in scalable CMOS with the OCT tools from U.C. Berkeley [29]. The area of the prototype as described is 496x997 lambdas. We found that CCM is an excellent research field. Not only the idea of making a co-processor for multiple-valued cube calculus operations opens a new and broad research area, but also the peculiarities of cube calculus led to the development of a new architecture which implementation generated many new research ideas in itself. We found many new unsolved problems in the implementation of the circuit. Design of the CCM-1 served to stimulate new research approaches and formulation of problems of general nature, which are the subject of our current research [18]. The ultimate goal of this project is to design a complete add-on board to PC-compatibles that will serve as a hardware accelerator for logic synthesis and other programs that use cube calculus subroutines.

The above architecture can be used for solving combinatorial problems in many areas. In the new variant of CCM [18] several other operations and relations for other multiple-valued cube calculus operations have been defined in a similar way, which even further extends its applications.

We divided the architecture into loosely coupled building blocks that were formalized and generated from a behavioral description and implemented with automatic design tool. Now one can create several variants of CCM tailoring the architecture to an application with less effort.

The scaling of the chip is simplified because the modularity of the processing unit allows to modify its size by changing the number of iterative cells in the iterative circuit. Also the fact that the processing element is asynchronous eases the scaling problem, by not having to distribute a tightly synchronized clock to large chip areas, even for an ILU with a very large number of ITs. It seems to be a natural extension of this research to have the control unit also asynchronous. Following the design methodology to design asynchronous circuits in [13] the CCM could be the second asynchronous processor [14].

1. LITERATURE

[1] Brayton, R.K., Hachtel, G.D., McMullen, C.T., Sangiovanni-Vincentelli, A.L., "Logic Minimization Algorithms for VLSI Synthesis", *Kluwer Academic Publishers, 1984.* [2] Butts, M.R., and J.A. Batcheller, "Method of Using Electronically Reconfigurable Logic Circuits", *U.S. Patent #5,036,473, Jul. 30, 1991.* Chan, A.H.: "Using Decision Trees to Derive the Complement of a Binary Function With Multiple-Valued Inputs," *IEEE Trans. on Comp.*, Vol. C-36, pp. 212 - 214, Febr. 1987. [3] Davio, M., Deschamps, J.P., and A. Thayse: *"Discrete and Switching Functions"*, McGraw-Hill Book Co., Inc., New York, 1978. [4] DeMicheli, G., Brayton, R.K. and A. Sangiovanni-Vincentelli: "Optimal State Assignment for Finite State Machines", *IEEE Trans. on CAD*, Vol. CAD-4, No. 3, July 1985, pp. 268-284. [5] Garey, M.R. Johnson, D.S., "Computers and Intractability. A Guide to the Theory of NP-Completeness", *W.H. Freeman and Company*, San Francisco 1979. [6] Gerace, G.B. et al, "TOPI-A Special-Purpose Computer for Boolean Analysis and Synthesis", *IEEE TC*, Vol. C-20, pp. 837-842, Aug. 1971. [7] Helliwell, M., and M.A. Perkowski: "A Fast Algorithm to Minimize Multi-Output Mixed-Polarity Generalized Reed-Muller Forms", *Proc. 25-th ACM/IEEE DAC*, paper 28.2, pp. 427-432, June 12-June 15, 1988. [8] Hong, S.J., Cain, R.G., and D.L. Ostapko: "MINI: A Heuristic Approach for Logic Minimization", *IBM J. Res. Develop.*, Vol. 18, pp. 443 - 458, Sept. 1974. [9] Ho, P.M., Perkowski, M., "Systolic Architecture for Solving NP-Hard Combinational Problems of Logic Design and Related Areas", *Proc. ISCAS'89, pp. 1170-1173, 1989.* [10] Johnson, D., "The NP-Completeness Column: An Ongoing Guide". *Journal of Algorithms*, Academic Press, each issue. [11] Kida, L., and M.A. Perkowski, "Cube Calculus Machine, Version One", Technical Report, PSU 1991. [12] Marin, M.A., "Investigation of the Field of Problems for the Boolean

Analyzer", *Ph.D. Dissertation*, Univ. of California, Los Angeles, 1971. [13] Martin, A.J., "A Synthesis Method for Self-Timed VLSI Circuits", *Proc. 15th Annual International Symposium of Computer Architecture*, May 30-June 2, 1988, Honolulu, Hawaii, pp. 224-229. [14] Martin, A.J., Burns, S.M., Lee, T.K., Borkovic, D., Hazewindus, P.J., "The First Asynchronous Microprocessor: The Test Results", *Proc. 16th Ann. Intern. Symp. on Comp. Arch.*, May 28-June 1, 1989, Jerusalem, Israel, pp. 95-110. [15] McCall, J.T., Tront, J.G., Gray, F.G., Haralick, R.T., McCormack, W.M., "Parallel Computer Architectures and Problem Solving Strategies for the Consistent Labeling Problem", *IEEE TC.*, Vol. C-34, No. 11, Nov. 1985. [16] Perkowski, M.A., "Systolic Architecture for the Logic Design Machine", *Proc. ICCAD'85*, pp. 133-135. [17] Perkowski, M., Helliwell, M., Wu, P., "Minimization of Multiple-Valued Input Multi-Output Mixed-Radix Exclusive Sums of Products for Incompletely Specified Boolean Functions", *Proc. ISMVL-89*, [18] Perkowski, M., "The Universal Logic Machine", *Submitted to ISMVL '92*. [19] Roth, P.: *"Computer Logic, Testing and Verification"*, Rockville, MD: Computer Science, 1980. [20] Rudell, R.L., and A.L. Sangiovanni-Vincentelli: "ESPRESSO-MV: Algorithms for Multiple-Valued Logic Minimization, *Proc. IEEE Custom Integrated Circuits Conf.*, 1985. [21] Rudell, R.: *"Multiple-Valued Logic Minimization for PLA Synthesis"*, M.S. Report, June 5, 1986. University of California, Berkeley California 94720. [22] R.L. Rudell, and A.L. Sangiovanni-Vincentelli, "Multiple-Valued Minimization for PLA Optimization," *Proc. ISMVL*, pp. 198-208, May 26-28, Boston, MA, 1987. [23] Sasao, T.: "An Application of Multiple-Valued Logic to a Design of Programmable Logic Arrays", *Proc. 8th ISMVL*, 1978. [24] Sasao, T.: "Multiple-Valued Decomposition of Generalized Boolean Functions and the Complexity of Programmable Logic Arrays," *IEEE Trans. Comput.*, Vol. C-30, pp. 635-643, Sept. 1981. [25] Sasao, T.: "Input Variable Assignment and Output Phase Optimization of PLA's," *IEEE Trans. Comput.*, Vol. C-33, pp. 879 - 984, Oct. 1984. [26] Sasao, T, "HART: A Hardware for Logic Minimization and Verification", *Proc. ICCD'85*, Oct. 7-10, 1985. [27] Sasao, T.: "MACDAS: Multi-level AND-OR Circuit Synthesis Using Two-Variable Function Generators", *Proc. 23-rd Design Automation Conference*, Las Vegas, pp. 86-93, June 1986. [28] Shumake, D., "The MOS Boolean Analyzer", *M.Sc. Thesis*, UCLA, 1971. [29] Spickelmier, R. (ed.), "OCT Tools Distribution 2.1.", *Electronics Research Laboratory*, University of California, Berkeley, March 25, 1988. [30] Svoboda, A., "Boolean Analyzer", *Proc. Information Processing 68*, Amsterdam, North-Holland, 1969. [31] Svoboda A., "Parallel Processing in Boolean Algebra", *IEEE TC*, Vol. C-22, No. 9, pp. 848-851, Sept 1973. [32] Ulug, M.E., Bowen, B.A., "A Unified Theory of the Algebraic Topological Methods for the Synthesis of Switching Systems", *IEEE TC*, pp. 255-267, March 1974. [33] Ulug, M.E, "VLSI Knowledge Representation Using Predicate Logic and Cubical Algebra", *Proc. IEEE Intern. Conf. on Comp. and* Comm. Arizona, 1985, pp. 292-297. [34] Ulug, M.E., "Application of Cubical Array Operators to a Relational Database", *Proc. of the Minnowbrook Workshop*, July 23-26, 1985, Blue Mountain Lake, New York. [35] Ulug, M.E., "A Real-Time AI System for Military Communications", *Proc. of the 3-rd IEEE Conf. on Artif. Intell. Applic.*, Febr. 1987, Orlando, Florida. [36] Wah, B.W., Ma, Y.W.E., "MANIP - A Multicomputer Architecture for Solving Combinatorial Extremum-Search Problems", *IEEE TC*, Vol. C-33, No. 5, pp. 377 - 390, May 1984. [37] Yasura, H., Tsujimoto, T., Tamaru, K., "Parallel Exhaustive Search For Several NP-Complete Problems Using Content Addressable Memories", *Proc. IEEE ISCAS'88*, pp. 333- 336. [38] Pratt, V., "Thinking Machines. The Evolution of Artificial Intelligence", *Basil Blackwell, Inc.*, Oxford, U.K., 1987.

#		CU=IT[0]	U[01]	U[23]	U[45]	V[01]	Z[01]	Z[23]	CU=IT[N+1]	
1	i		1	2	3	4	5	6		
2	IT[i]		IT[1]	IT[2]	IT[3]	IT[4]	IT[5]	IT[6]		
3	A[i]		01	11	00	11	01	10		
4	RELATION[i]		0	1	0	1	0	0		(A=X)
5	LEFT[i]	1=>	0	0	0	1	0	0	=>	
6	RIGHT[i]	<=	0	0	0	1	0	0	<=1	
7	VARIABLE[i]		1	1	1	0	1	1		(A≠X)
8	COUNT[i]	0=>	0	0	1	1	1	2	=>	(2 resultant cubes)
9	s[i]		10	00	11	11	11	11		
10	S[i]		11	11	11	11	10	01		

Table 1.

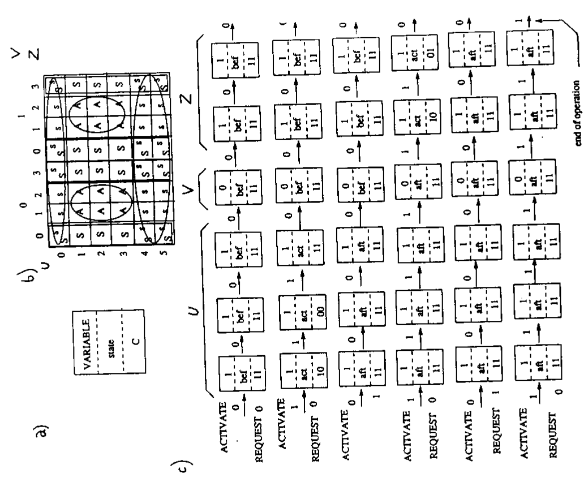

Fig.3.

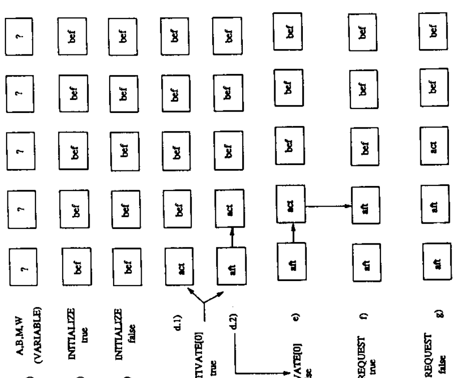

Fig. 2.
An example of a sequential operation.

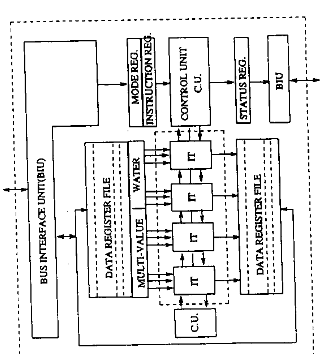

Fig. 1.
The block diagram of CCM architecture.

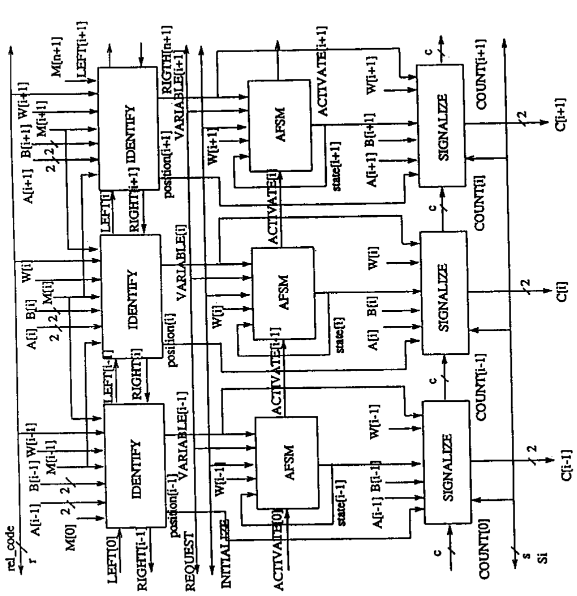

Fig.4.
Details of IT and interconnections of ITs in the ILU.