Massively Parallel Structures of Specialized Reconfigurable Cellular Processors for Fast Symbolic Computations

Lech Jóźwiak, Marek A. Perkowski* and David Foote*

Eindhoven University of Technology Faculty of Electrical Engineering
P.O. Box 513, EH 10.25 5600 MB Eindhoven, The Netherlands

Portland State University* Dept. of Electrical Engineering
Portland, OR 97207 – 0751 USA

Abstract

A large class of problems of modern engineering and scientific computations can be expressed and solved by operations of the Multiple-Valued Cube Calculus (MVCC). However, the standard processors are inefficient for the MVCC operations. We developed therefore a special reconfigurable processor that efficiently implements the MVCC operations. For execution of a certain algorithm, an appropriate massively parallel structure of such processors is used and the processors are programmed only for the MVCC operations actually used in the algorithm. The processor's versatile and scalable design enables a lot of various massively parallel structures. To our knowledge it is the first processor specialized in fast symbolic computations.

1. Introduction.

Most of the existing computers were designed to perform operations on numbers, and therefore, the structure of the standard processor's ALUs is tuned to the arithmetic operations. The standard ALUs implement efficiently the formal system of arithmetic. On the other hand, a lot of other formal systems, as set theory, binary and multi-valued logic, fuzzy logic, morphological algebra etc., are used in many modern engineering areas and for scientific computations. However, the standard ALUs do not implement operations of almost all of these systems in hardware. Even the binary-logic operations performed by the standard ALU's are limited to only very simple operations, such as AND, OR, NOT and EXOR. More complex binary logic operations, as sharp, consensus, crosslink, complement and others, as well as multi-valued logic operations can be implemented on the standard processors (in software) only as long sequences of the simple logic operations. The price paid for the software-based implementation is however speed degradation. Moreover, in most of the commonly used processor architectures, the parallel processing is very limited (even in modern RISC or Pentium processors). It is also a very hard problem for compilers.

To improve the computation speed for a certain formal system, the two following approaches can be used in combination: a special hardware processor can be developed that efficiently implements (in hardware) the specific operations of the considered formal system, and a massively parallel structure of such specialized hardware processors can be used for computations. For many formal systems, just specialized hardware implementations seem to be most appropriate (e.g. fuzzy logic, morphological algebra, DSP, and image processors).

In many modern areas of engineering and scientific computations related to discrete problem solving and artificial intelligence, such as: design automation of digital systems, machine learning, pattern recognition, image processing, data base design and information retrieval from data bases etc., various discrete optimization problems must be solved repeatedly. The discrete optimization problems are complex from the computational viewpoint (typically NP-hard). The effectiveness and efficiency of solving this kind problems heavily depends of the data representations, algorithms and heuristics used, and of the speed of the repeatedly performed typical operations on the data.

It is of course possible to develop a specific hardware processor for each such algorithm, but this would require synthesis of a specific hardware from the behavioral specification of a certain algorithm. The automatic behavioral compilation is however not yet in such an advanced stage as to perform this task effectively and

efficiently for a large class of algorithms. On the other hand human-made hardware designs would be very expensive. Fortunately, it is a well known fact that the combinatorial optimization problems can be polynomially transformed to another combinatorial problems, and can be expressed and solved by few (multiple-valued) logic operations performed repeatedly [3]. Potential applications of the (multiple-valued) logic are of course not limited to the discrete optimization problems.

2. The concept of a specialized massively parallel reconfigurable processor for fast symbolic computations.

Our approach is to express a large class of problems of modern engineering and scientific computations by operations of the **Multiple-Valued Cube Calculus (MVCC)**, to efficiently implement the MVCC operations in specialized hardware, and to compute solutions for these problems with a massively parallel structure of such specialized hardware processors.

The Multiple-Valued input, binary-output, Cube Calculus (MVCC) [10][11] is the most general known data representation and calculus in: propositional logic, logic synthesis, logic programming, logic simulation, machine learning, image processing, data-bases, set logic, and several other areas of problem solving and artificial intelligence [8][9][10][11].

It is possible to develop a general-purpose symbolic processor for efficient computation in all these fields, with hardware-implemented large subset of MV-logic operations, but such processor would be very complex and expensive, because the number of possible MV-logic operations grows fast with the number of logic values. Moreover, only a small specific subset of all implemented operations would actually be used for each particular algorithm.

Fortunately, modern FPGA technology allows for:

- fast and relatively inexpensive realization of specialized programmable processors for arbitrary operations, and
- reuse of the same FPGA hardware to implement various specialized application specific processors (re-programmability).

Moreover, FPGA implementations are especially efficient for regular and scalable architectures.

Therefore, instead of developing a general-purpose symbolic processor with hardware-implemented large subset of MV-logic operations or an application-specific processor for a single optimization algorithm, we designed and implemented a broad-spectrum reconfigurable hardware accelerator for MV-logic operations: a Multiple Valued Cube Calculus (MVCC) processor, called also a **Cube Calculus Machine (CCM)**. CCM can be used as a co-processor of the standard arithmetic processor. It is a hardware accelerator reconfigurable for a specific algorithm that is required at a certain time. When executing a certain algorithm, it is programmed only for the MVCC operations actually used in the algorithm. The reprogrammability feature is crucial for effective and efficient implementation of the MVCC, because the number of different MVCC operations grows fast with the number of logic values, but only a small specific subset of all operations is used for a particular algorithm. Moreover, different MVCC operations have the same general computation pattern, what enabled to develop general processor architecture for all MVCC operations, but each of them involves different

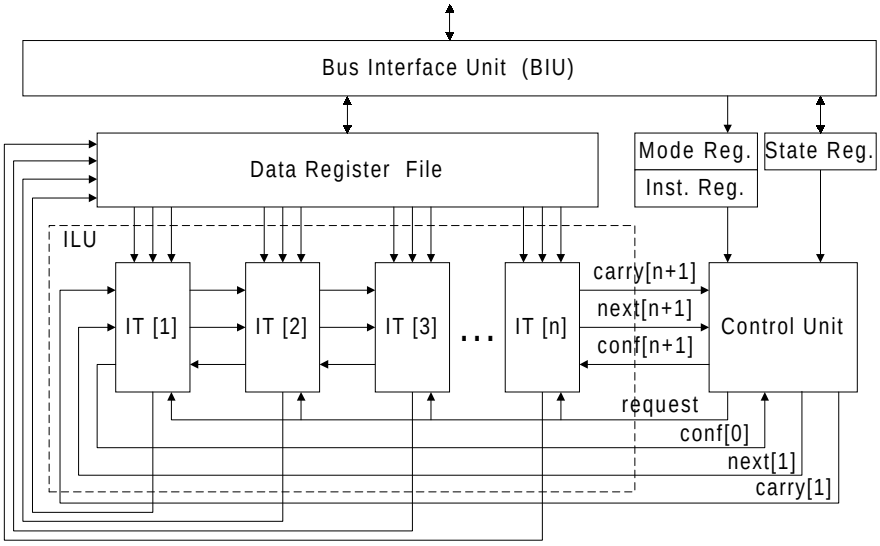

Figure 1 The block diagram of the CCM processor architecture.

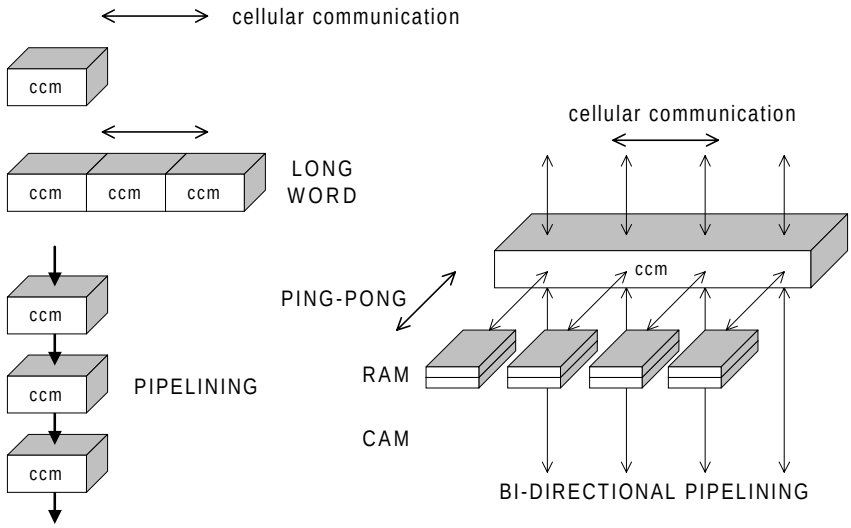

Figure 2 Data flow in structures built of CCM building blocks.

combination of elementary logic operations. These elementary logic operations will be reprogrammed each time the operation sort or number of logic values is changed.

The heart of the CCM is a bit-sliced data path representing a one-dimensional interactive network of electrically programmable cellular automata. The data path is thus not combinational but sequential. This follows from the fact that many of the typically used MVCC operations are sequential (not combinational). Each of them generates as its result not a single word, but a number of words.

Further speedup can be achieved when hardware implementation of typical repeatedly performed operations will be combined with various appropriate sorts of parallelism. The operations are in most cases performed on a large amount of data. In particular, the lowest level loop of the algorithms - the variable loop - may involve a lot of binary or multi-valued variables. From the computation speed viewpoint, it would be the best to implement the computations for all variables of the loop in parallel, by making the word of a single CCM processor at least as long as required by all the variables. However, different word-length may be required for each particular algorithm, and therefore, the CCM architecture has been developed as fully scalable (Figure 1). Moreover, to overcome the limitations on physical dimensions of FPGAs and boards with FPGAs, the CCM's design, in particular the iterative cell's (IT) and control unit's design (Figure 1), enables the horizontal connection of any required number of CCM processors to implement long words exceeding the word-length of a single CCM (Figure 2). For each pair of CCM processors in the horizontal chain, the control unit enables direct connection of the rightmost IT of the left processor with the leftmost IT of the right processor of the pair, without using the bus interface unit. These two features of the CCM's architecture implement the micro parallelism at the sub-instruction level.

The computation speed can be further enhanced by using a vertical linear array (pipeline) of the CCM processors (Figure 2) to implement the second lowest level loop of the algorithms - usually the cube loop or instruction loop. The design of the CCM's bus interface unit (Figure 1) enables vertical communication between the CCM processors as well as communication of the CCM processors with the host processor. It also enables implementation of the two-directional pipelining of CCMs and ping-pong between the CCM processors and RAM or CAM memories (Figure 2), which are very useful for some computations involving cubes. For example, the architecture involving both the two-directional pipelining and ping-pong can be used to compute the morphological Hough transform. These massively parallel structures of CCM's are not the only structures possible. Among others, the tree of pipelined CCM's is possible and it is very useful for many computations, e.g. for computing the generalized Petrick function. Various structures can also be combined to form more complex structures. In this way three-dimensional massively parallel processing architectures with three-dimensional data movements can be realized with CCM processors.

A versatile design of CCM and implementation of the massively parallel CCMs structures with reprogrammable FPGAs allow for an extremely high degree of flexibility.

The number of possible to program CCM operations is extremely high (see Section 5). These operations include: all MVCC operations, operations on multi-valued multi-output relations, on numbers and number intervals, on symbolic predicates, etc. Moreover, in various CCM processors or various iterative logic units of a single CCM (Figure 1) different MVCC operations for MV-variables with different number of values can be programmed at the same time. Furthermore, the same reprogrammable FPGA hardware can be used to implement various massively parallel structures of differently programmed CCM processors, and also various other co-processors or accelerators.

We have designed the CCM and simulated, implemented, and tested its prototype that implements computations for the word of sixteen binary, eight 4-valued, four 8-valued variables, or any combination of binary, 4-valued or 8-valued variables for a total of 32 bits. For the prototype implementation we used two Xilinx FPGA. XC-3090-50 PP175C chips.

3. Introduction to cube calculus.

Let's consider a **set of discrete variables (attributes)** $X_1, X_2, \ldots, X_n$, such that each variable X_i can take values from a certain finite discrete set V_i (V_i can be any finite set of symbols). A **literal** X_i^{Si} of a certain variable X_i represents a characteristic function of a certain subset S_i of V_i, i.e. the literal's value is 1 for symbols from this subset.

Example:

- for binary logic: $X^1 = X$, and $X^0 = X'$ are two literals,
- for four-valued logic $V_i = \{0,1,2,3\}$:

$$X^{\{0,2\}} = \begin{cases} 1 & \text{if } X = \{0,2\} \\ 0 & \text{if } X = \{1,3\} \end{cases} \text{ is a literal.}$$

A **cube** on $X_1, X_2, \ldots, X_n$ is an ordered set of literals on $X_1, X_2, \ldots, X_n$, i.e.: $X_1^{S1} X_2^{S2} X_n^{Sn}$. A cube represents a sub-space (a cube) in the n-dimensional discrete multi-valued space (see Figure 3). Usually (e.g. in the traditional switching algebra) a cube is interpreted as a product of literals, but it can also be interpreted as a **sum of literals** or an **exor of literals**. In general, a cube can represent any ordered set of subsets of certain discrete sets V_i, i = 1, ... , n (i.e. Cartesian product of the subsets). Each such subset can be represented in different ways: as a (binary) vector of a given length, as a number or as a number interval. For the purpose of this paper, the first representation will be used.

Cube calculus is a system of:

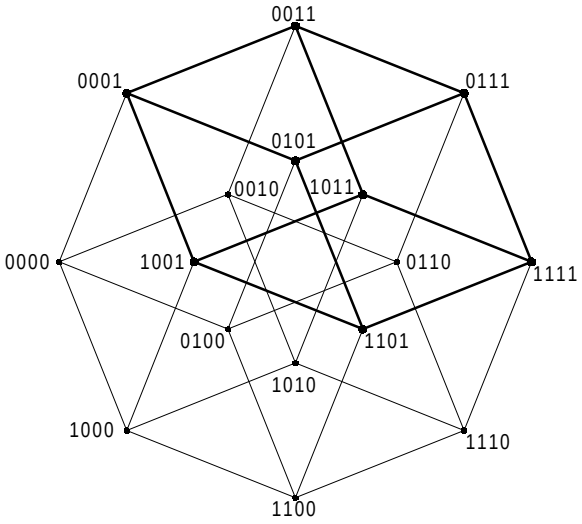

Figure 3 A 3-dimensional cube (---1) in a 4-dimensional space (----).

- **a set of all cubes** on a certain ordered set of discrete variables $X_1, X_2, \ldots, X_n$, that contains also an empty cube and a full cube (full discrete space defined by $X_1, X_2, \ldots, X_n$),
- **a set of operations** on sets of cubes.

Cube operations are of several **kinds**:

- cube **operators**: result is a list of 0 to n cubes,
- cube **predicates**: result is logic value 0/1, and
- **counting operations**: result is a number (e.g. Hamming distance of two cubes).

In CCM literals are represented in positional notation. **Positional notation** uses a separate bit to represent each possible value of each literal. If the literal is true for a specific value, the corresponding bit is set to "1". For example, assuming that each of the variables X_1, X_2 and X_3 is a 3-valued variable and the values are {0,1,2}, the cube:$X_1^{\{0,2\}} X_2^{\{1,2\}} X_3^{\{2\}}$ will be denoted by [101-011- 001].

The **base K** of a logic machine is **the number of bits required to represent a simple symbol** in this machine.

Example:

K = 2 allows us to realize all logic operations (matrices) in multiple-valued logic with no more than 2^2 = 4 values. The four simple symbols are: 0 - negated variable, 1 - positive variable, X - don't care, and ε-contradiction. They are encoded in positional notation as follows: 0-10, 1-01, X-11, and ε-00. With this encoding, cube bcd' (X110) on the ordered set of discrete variables (a, b, c, d) is represented by: [11-01-01-10].

When multiple-valued logic is realized using binary signals, K binary signals are used to represent each simple symbol from the set of 2^K symbols. A simple symbol

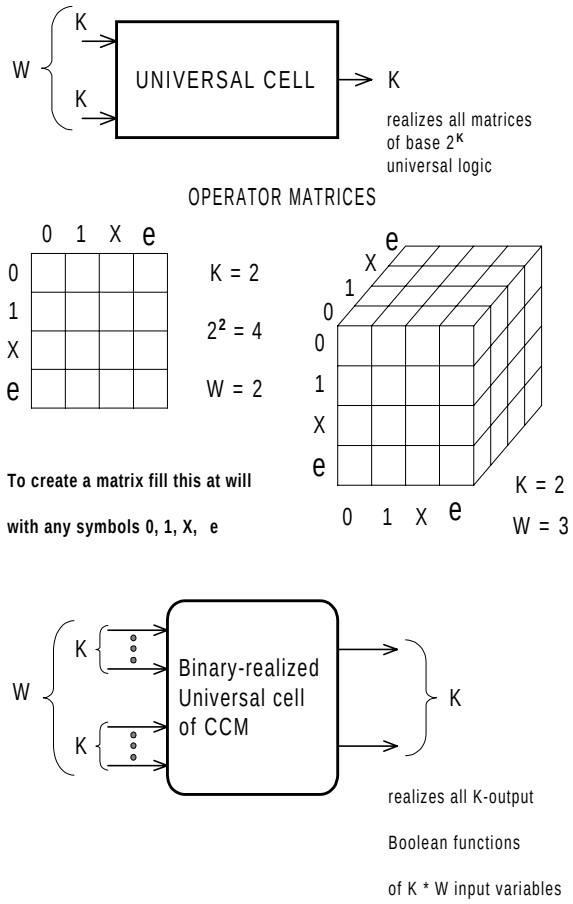

Figure 4 A W-input K-base universal cell.

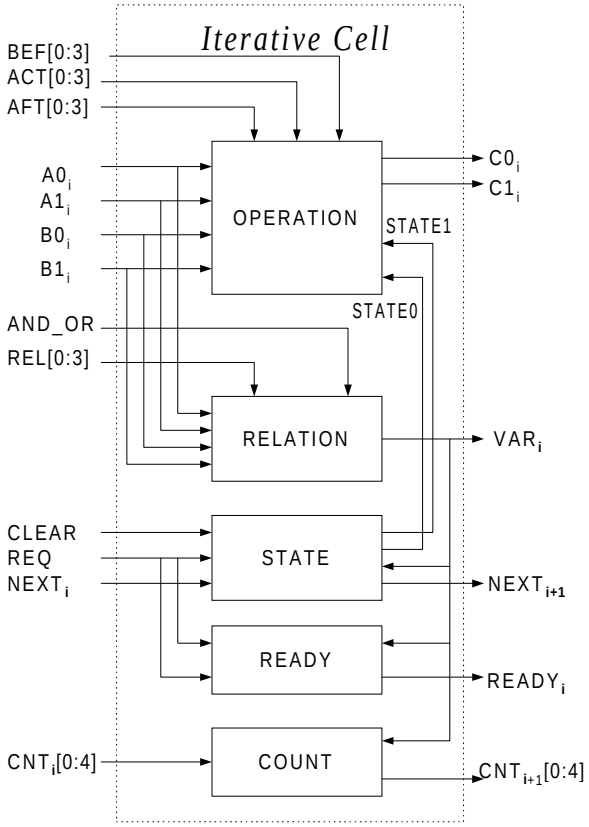

Figure 5 Iterative cell (IT).

expressed in base K will be called a **Kit** (analogously to bit which is a Kit with K=1). K=1 base symbol is sufficient to realize the binary logic, set theory, and binary arithmetic. K>1 are necessary for multiple-valued systems. Kit is a basic information chunk of logic machines analogous to bit in arithmetic machines.

A **W-input K-base universal cell** is a logic block with W inputs and one output, each input and output being a base K signal (see Figure 4).

In CCM each simple symbol is processed by the **ITerative Cell (IT)**, being an elementary processor. K-base symbol requires K-base IT cell (see Figure 5). In a CCM of base K each variable can have arbitrary, but divisible by K, number of values. A complex variable with **R·K** values (a complex symbol) is processed by **R** cooperating ITs. Thus, CCM introduces simple and complex symbols, as an intermediate level between bits and variables, enables a flexible number of values in literals, and usage of a (for example) two-bit IT cell to represent a part of an MV-literal of an arbitrary number of values.

The main concepts of hardware implementation of the cube calculus data are:

- a cube,
- a list (array) of cubes - **clist**, and
- a list of lists of cubes - **cclist**.

Summing up, the Multiple-Valued Cube Calculus is a set of cubes of a certain discrete space and a set of operations on the cubes, clists, and cclists.

4. Computation patterns of cube calculus operations.

There are a lot of (two-argument) operators on cubes, but fortunately they expose certain common computation patterns, and they can be sub-divided into the three following **classes**:

- **simple combinational operations**,
- **complex (conditional) combinational operations**,
- **sequential operations**.

Operations of each class have the same computation pattern, and the pattern of a simpler class can be

considered as a special case of the pattern of a more complex class. Existence of the common computation pattern enabled design of certain dedicated hardware architecture for cube operations. Implementation of each particular operation is then achieved by a particular instantiation of the general architecture that is realized by appropriately programming the FPGA structures.

Simple combinational operations:

- produce a **single cube as a result**, and
- are **position-by-position** (bit-by-bit) operations, the same operation for each position.

Complex (conditional) combinational operations:

- produce a **single cube as a result**,
- are **position-by-position** (bit-by-bit) operations, and
- the literals of the resulting cube are calculated from the corresponding literals of the argument (operand) cubes by **conditional operations on the literals of the argument cubes**.

Sequential operations:

- can produce **more than one resulting cube**: **one cube per active literal** (**active position**),
- can have **from 0 to n potentially active positions** i for which a certain **relation** rel (A_i, B_i) is satisfied,
- the **operation** executed **on the active position** is a certain simple combinational operation,
- **operations** executed **before and after the active position** are in general certain simple combinational operations, but they often consist just of copying A_i or $B_{i,}$,
- **potentially active positions are computed simultaneously**,
- **the resulting cubes are created sequentially** starting from the leftmost potentially active position.

Example (**simple combinational operation**): inter-section (product) operation

$$A \cap B = \begin{cases} \varnothing & \text{if } A_i \cap B_i = \varnothing \text{ for some } i \\ A_1 \cap B_1, A_2 \cap B_2, \ldots, A_n \cap B_n & \text{otherwise} \end{cases}$$

Example (**complex combinational operation**): prime operation. A prime B = X_1^{A1},..., X_{i-1}^{Ai-1}, $X_i^{Ai \cup Bi}$, $_{i+1}^{Ai+1}$,..., X_n^{An}, where $A_i \cup B_i$ are computed only for those variables X_i for which $A_i \cap B_i \neq \varnothing$ is satisfied (such positions are called active positions).

Example (**sequential operation**): nondisjoint sharp

$$A \# B = \begin{cases} A & \text{when } A \cap B = \varnothing \\ \varnothing & \text{when } A \subseteq B \\ A \#_{basic} B & \text{otherwise} \end{cases}$$

$A \#_{basic} B = \{X_1^{A1}, \ldots, X_{i-1}^{Ai-1}, X_i^{\neg Ai \cap Bi}, X_{i+1}^{Ai+1}, \ldots, X_n^{An} |$ for such i = 1,..., n that $\neg(A_i \subseteq B_i)$ } e.g. xxx1 # 111x = {0xx1, x0x1, xx01}.

Each sequential operation can be described by the pattern: A operation B = $\{X_1^{aft(A1,\ B1)}, \ldots, X_{i-1}^{aft(Ai-1,\ Bi-1)}, X_i^{act(Ai,Bi)}, X_{i+1}^{bef(Ai+1,\ Bi+1)}, \ldots, X_n^{bef(An,\ Bn)} |$ for such i = 1,..., n that rel(A_i,B_i) = 1}.

Patterns of combinational operations are special cases of this pattern. Thus, simple combinational operations can be considered as a special case of the complex combinational operations, and all combinational operations can be considered as a special case of the sequential operations (see Table 1).

For different operations, different functions for **rel**, **bef**, **act**, and **aft** are selected. Functions **bef**, **act**, **aft**, and **rel** are K-wise functions. If, for example, K=2, then each two bits of the resulting cube C that represent a simple symbol depend only on corresponding two bits of the argument cubes A and B, i.e.: C^iC^{i+1} depend only on

Table 1 Cube calculus operations.

function	**relation (*rel*)**		**output operation**		
	rel	*and/or*	before*(bef)*	active *(act)*	after *(aft)*
Intersection	*1*	*and*	$Ai \cap Bi$	-	-
Supercube	*1*	*and*	$Ai \cup Bi$	-	-
Prime	$Ai \cap Bi \neq \varnothing$	*and*	*Ai*	$Ai \cup Bi$	-
Crosslink	$Ai \cap Bi = \varnothing$	*and*	*Ai*	$Ai \cup Bi$	*Bi*
Sharp	$(Bi \supseteq Ai)$	*or*	*Ai*	$Bi \cap Ai$	*Ai*
Disjoint Sharp	$(Bi \supseteq Ai)$	*or*	*Ai*	$Bi \cap Ai$	$Ai \cap Bi$
Symmetric Cons.	*1*	*and*	$Ai \cap Bi$	$Ai \cup Bi$	$Ai \cap Bi$
Asymm. Cons.	$(Bi \supseteq Ai)$	*or*	$Ai \cap Bi$	$Ai \cup Bi$	$Ai \cap Bi$

A^iA^{i+1} and B^iB^{i+1}.

A complex symbol that represents a value of a variable C of the length R•K(IT) is a composition of R simple symbols being the results computed in all ITs representing this variable (the ITs form the R K(IT)-sliced CCM chips).

5. The CCM's architecture.

Software implementation of each cube operation uses a single loop that runs through all cube variables.

The following two **crucial ideas** formed a basis for invention **of the CCM**:

- execution of the **lowest level loop** of the cube operation algorithms - the variable loop - **in hardware**, by using a **linear iterative array of cellular automata** (FSMs) with information flowing between the FSMs from the left to the right and from the right to the left,
- **reconfiguration** of logic functions of the cellular automata by their **implementation with FPGAs**.

The **CCM System Architecture** involves:

- a **host processor**, being a traditional general purpose computer, and
- a **massively parallel structure of the CCM processors**, which forms a co-processor (application specific reconfigurable hardware accelerator) of the host processor.

A single **CCM processor** (see Figure 1) consists of:

- an **iterative logic unit** (**ILU**), being a horizontal linear array of R K(IT)-sliced CCM chips, each composed of R ITs,
- a **control unit**,
- a **register file**,
- a **bus interface unit**.

The **lowest level loop** - usually the **variable loop** - is implemented inside the CCM processors, by the **horizontal communication** between the ITs, or with few CCM processors connected horizontally. The s**econd lowest level loop** - usually the **cube loop** - is implemented by the **vertical** linear array (pipeline) of the CCM processors. This enables two-dimensional data movements: horizontal (inside or between the CCM processors) and vertical (between the CCM processors). The third dimension of data movements is realized with RAM memories connected to ITs. Some possible data flows in structures built from the CCM building blocks are given in Figure 2.

The implementation of predicates, counting operations and combinational operators in CCM is very similar to their implementation in a traditional arithmetic processor, but the implementation of the sequential operators requires some more comments.

First, a certain relation rel (A_i, B_i) must be satisfied by literals at a position i for this position to become active and a resultant cube to be created. While all potentially active positions are computed in parallel by evaluating the relation rel (A_i, B_i) for all i in all ITs at the same time, the corresponding literals (positions) are activated sequentially, and exactly one resultant cube is created for each literal (position) active at a certain time, by executing a certain operation act (A_i, B_i) at the active literal (e.g. $\neg A_i \cap B_i$ for sharp), a certain operation bef (A_i, B_i) at all positions before (or right of) the active literal (e.g. copying the literals for sharp), and a certain operation aft (A_i, B_i) at all positions after (or left of) the active literal (e.g. copying the literals for sharp). The literals are activated starting from the leftmost potentially active position. After performing the described above computations for a certain active literal, the literal corresponding to the next to the right potentially active position is activated, and the corresponding resultant cube is created. When producing a particular resultant cube, all the literals on positions to the left from the active literal are of the after type, and all literals on positions to the right from the active literal are of the before type. All these operations are totally executed in hardware by the iterative network of small fast asynchronous FSMs (ITs), when using a fast asynchronous communication between the FSMs.

A CCM operation possible to program is a point in a multi-dimensional space created by a Cartesian product of basic programmable features, such as: rel, bef, act, aft, composition, pipelining, etc. The number of possible to program operations is extremely large. The **possible to program operations** include:

- all known MVCC operations,
- operations on various kinds of numbers and number intervals,
- operations on symbolic predicates,
- operations on multi-valued multi-output relations, etc.

CCM can be considered as a prototype of a **symbol-processing computer** with a sort of data path microprogramming (in contrast to control path microprogramming of the traditional arithmetic computers).

6. Advantages of CCM application.

There are several reasons why the CCM can greatly speed up many applications. The main reason is of course the fact that the ALU of a conventional computer does not efficiently implement the (multi-level) logic operations, while CCM does it. For instance, to calculate the

consensus of two cubes, the ALU must execute a long series of shifts and ANDs. Also, some of the resultant cubes are empty and must be removed. This means that the generation of the resultant cubes is irregular and inefficient. Since the cube calculus machine was specifically designed for MVCC operations, each of these operations can be executed in just a single clock pulse or a few clock pulses and requires only one CCM instruction. The CCM does not generate empty resultant cubes, so the generation of the resultant cubes is completely regular. The time needed to generate the cubes solely depends on the number of non-empty resultant cubes.

Analogously to the traditional processor which is a general-purpose processor, but tuned to the arithmetic computations, the CCM is also a general-purpose processor, but tuned to the symbolic computations. Being the result of a single development process, it can be used efficiently for a broad range of applications. It is reprogrammable, what enables to implement only the operations that are actually required for execution of a certain algorithm at a certain time slot. It allows a customized instruction set, which can be optimized for a certain application. Moreover, the SRAM based FPGAs can be reconfigured while the host computer, or even the host program that uses CCM, is in full operation.

Furthermore, in a conventional computer the control is realized by a program stored in RAM. This results in a considerable control overhead, since the instructions have to be fetched from RAM. If an algorithm contains loops, the same instructions will be read many times. This makes the memory interface the bottleneck of the conventional computer architectures, especially when the memory bus is not fast as the internal processor bus. In the CCM architecture, most of the control is implemented in the CCM's data path itself. Once a complex MVCC instruction is loaded into the CCM, the host computer only needs to write the data cubes to the CCM and read the resultant cubes from the CCM. The host processor can process the resultant values from the CCM while loading them from the CCM, as the CCM awaits the next clock pulse to send another cube.

Additionally, in most of the commonly used computer architectures, the parallel processing is very limited (even in modern RISC or Pentium processors). Parallel processing has also proven to be very hard material for compilers. In the CCM architecture, parts of an existing program for a traditional computer can be replaced by a single CCM instruction. This CCM instruction is then executed in hardware that was specifically designed for this particular instruction, allowing micro parallelism in the CCM.

Another limiting factor of conventional computer architectures is the bandwidth of the ALU. As a result of the FPGA implementation, the CCM suffers to a much lesser extent from this problem, because ALU bandwidth can be flexibly adopted for each application. The only limiting factor is the capacity and speed of the FPGAs in the hardware that is used.

Furthermore, the CCM architecture is regular, scalable and allows massively parallel computers to be built from a large number of CCM processors, which are controlled by other CCMs and ultimately, by the host computer. Thus, true massively parallel processing can be realized. The mapping of these architectures onto the FPGAs requires considerable time, but once they are compiled, the host computer can instantly load new architectures into the FPGA board.

The question arises whether the speedup of certain application justifies the purchase of a costly hardware accelerator board. However, the cost of the FPGA hardware can be shared among various applications, not only those involving symbolic computations. Moreover, the essence of the CCM is not the FPGA board, but the architecture that is programmed into the FPGAs. The CCM architecture can be programmed on existing FPGA-architecture hardware boards, or a 'specific' CCM hardware board may be used for other applications. For example, the CCM can be programmed into the Anyboard or Xputer [2].

Since the essence of the CCM is its architecture that involves reprogrammable basic logic operations of ILU for rel, bef, act, and aft, it is even not necessary to implement the CCM on a classical FPGA board. Faster hardware for the CCM's implementation can be achieved by limiting reprogrammability to only rel, bef, act and aft, and by implementing most of the CCM processor in the classical hardware. In this way, the CCM processor can be implemented as very fast classical hardware chip with few small reprogrammable look-up tables in its ILU.

The fact that the CCM architecture can be programmed into the Anyboard and Xputer is not the only reason that they were mentioned. They both use repetitive compiling of new designs onto multiple FPGAs. This requires extensive and time consuming logic minimization and mapping algorithms. It just happens that logic synthesis is one of the key application areas where the CCM processor can make a great difference in execution speed. Since the SRAM-based FPGAs can be reprogrammed while the host computer is in full operation, it is quite easy to load the FPGA board with CCM processor and use the CCM to speed up the calculations of the new design of the board. When that design is finished, it is simply loaded into the FPGAs and immediately ready for testing. Thus, the CCM can serve as a hardware accelerator for itself – a computational beauty of "bootstrapping", reminding of a Lisp compiler that is compiled by itself. This illustrates the advantages of an FPGA-based add-on board. One single FPGA board can be used to implement several different add-on boards and

speed up many different nonconcurrent tasks. This not only reduces the costs of hardware, but also the power consumption, the number of slots that are needed, and even the size of the computer case.

7. Prototype implementation and experimental results.

We have designed, simulated, and implemented a prototype of the CCM for the word of sixteen binary, eight 4-valued, or four 8-valued variables, or any combination of binary, 4-valued or 8-valued variables for a total of 32 bits. For the prototype implementation we used two Xilinx FPGA XC-3090-50 PP175C chips (175 pin, 50 MHZ). It was designed with FutureNet DASH. 8 iterative cells of the prototype consumed approximately 48% of the available CLBs. The prototype implementation was simulated and tested on many examples. Timing analysis has been also performed: the greatest delay was of 145.8 nanoseconds.

The speedup of sharp operation was app. 25 times on 6 variable terms comparing to its software implementation. The speedup of Satisfiability Problem algorithm was app. 14 times comparing to its software implementation. These speedups were achieved for a single CCM processor having an ILU with a short word composed of 8 IT cells. Since the speedups grow with the number of IT cells in a single CCM processor and number of the CCM processors in various appropriate massively parallel architectures built of the CCM processors, computation speed enhancement in the full scale massively parallel implementation is expected to be much higher.

To have an impression of possible speed enhancement, we performed a number of simulation experiments with the tree of pipelined CCM processors used for computation of the generalized Petrick function. Among others, we considered a quite small tree with 3 levels and 7 CCM processors. We assumed a host processor clock rate of 10 MHz. Since the host processor must fetch all four leaf nodes of the CCM processor's tree in every execution cycle it limits the clock rate of the CCM processor's tree to a slow rate of 2.5 MHz. With these assumptions, the considered (small) parallel processing structure is able to solve a 1000-clause of generalized Petrick function within 0.7 milliseconds. To solve this problem instance with the same algorithm implemented in software, the traditional host computer (PC) operating at 10 MHz requires 70.8 seconds. Thus, application of an appropriate parallel structure of the CCM processors, even with a small number of processors, resulted in the speedup of app. 100000 times.

The experimental results are very encouraging and indicate that an appropriate parallel structure of the specialized CCM processors will give significant speedups for several applications, even a structure with a small number of processors.

8. Conclusion

The algorithms for discrete optimization problems as well as many other algorithms for problem solving in modern engineering areas can be expressed by few MV-logic operations on multiple-valued cubes. However, these operations must be repeated many times during a single run of a certain program that implements such an algorithm. Moreover, for solving practical problems some combinations of algorithms must often be executed repeatedly. Solving such problems with traditional arithmetic computers is inefficient.

We designed therefore a specialized computer architecture - CCM - that implements the MV-logic operations in hardware. To our knowledge, CCM is the first logic machine for multiple-valued logic, universal logic, and cube representation. CCM is specialized in fast symbolic computations. Its versatile, regular and scalable design allows for an extremely high degree of flexibility and enables a lot of various massively parallel structures built of CCM processors.

We implemented a CCM prototype composed of two CCM chips: each CCM chip performs computations for an equivalent of eight binary variables. We performed some experiments and measurements with this prototype. Even with the small scale prototype implementation quite big speedups were achieved, but speedups in the full-scale massively parallel implementation are expected to be much higher.

The full-scale version of the machine is being developed on DEC PERLE 1 board with Xilinx FPGAs, a coprocessor to DEC 5000 that we recently acquired. The CCM architecture is geometrically regular and scalable, what is an important advantage for its implementation with FPGAs, and in particular with the DEC PERLE 1 board. Ultimately, the CCM processor can be implemented as a very fast hardware chip with few small reprogrammable look-up tables. The CCM processor is a horizontal linear array of cellular automata, and more computation power can be achieved by horizontal connection and vertical connection (pipelining) of the CCM chips. One of our ultimate goals is to build a computer for fast functional decomposition of (multi-valued) functions, relations and sequential machines [4][7][13][14]. One of many possible applications of such a machine can be logic synthesis, another data mining in order to learn in real-time from WWW environment [6][14].

References

[1] S. Devadas: *Optimal Layout Via Boolean Satisfiability*, Proc. ICCAD, Santa Clara, Nov. 1989, pp. 294-297.

[2] R. Hartenstein: *A Novel Paradigm of Parallel Computation and its Use to Implement Simple High Performance Hardware*, Proc. Int. Conference on Information Technology, Tokyu, Japan, Oct. 1-5, 1990, pp. 1-12.

[3] G. deMicheli: *Synthesis and Optimization of Digital Circuits*, McGraw Hill, 1994.

[4] L. Jozwiak: *General Decomposition and Its Use in Digital Circuit Synthesis*, VLSI: An International Journal of Custom-Chip Design, Simulation, and Testing, Vol. 3, Nos. 3-4, 1995, pp.225-248

[5] L. Kida, M.A. Perkowski: *The Cube Calculus Machine: A Ring of Asynchronous Automata to Process Multiple-Valued Boolean Functions*, Proc. IEEE ISCAS'92, May 10-13, 1992, pp. 807-810

[6] M.A. Perkowski, L. Jozwiak: *Hardware Accelerator for Real Time Learning from WWW Data*, PSU Report, 1997

[7] M.A. Perkowski, M. Marek-Sadowska, L. Jozwiak, T.Luba, S. Grygiel, M. Nowicka, R. Malawi, Z. Wang and J.S. Zhang: *Decomposition of Multiple-Valued Relations*, Proc. ISMVL'97, 1997

[8] D. Rine: *Computer Science and Multiple Valued Logic*. Theory and Applications, North-Holland, 1984.

[9] T.D. Ross, at al: *Pattern Theory: An Engineering Paradigm for Algorithm Design*, Technical Report WL-TR-91-1060, Wright Laboratories, USAF, WL/AART/WPAFB, OH 45433-6543, August 1991.

[10] T. Sasao: HART: *A Hardware for Logic Minimization and Verification*, Proc. IEEE ICCD'85, Oct. 7-10, 1985, pp.713-718.

[11] S.Y.H. Su and P.T. Cheung: *Computer Minimization of Multi-Valued Switching Functions*, IEEE Trans. on Computers, vol.C-21, No. 9, p. 995, 1972.

[12] N. Song and M. Perkowski: *Minimization of Exclusive Sum-of-Products Expressions for Multiple-Valued Input, Incompletely Specified Functions*, IEEE Trans. On CAD, vol. 15, No 4, April, 1996, pp. 385-395.

[13] F. Volf, L. Jozwiak and M. Stevens: *Division-Based Versus General Decomposition-Based Multiple-Level Logic Synthesis*, VLSI Design: An International Journal of Custom-Chip Design, Simulation, and Testing, Vol. 3, Nos. 3-4, 1995, pp. 267-287.

[14] B. Zupan and M. Bohanec: *Learning Concept Hierarchies from Examples by Function Decomposition*, Technical Report, Department of Intelligent Systems, Josef Stefan Institute, Ljubljana, Slovenia, September, 1996.