CONSTRUCTIVE INDUCTION MACHINES FOR DATA MINING.

Marek Perkowski, Stanislaw Grygiel, Qihong Chen, and Dave Mattson

Portland State University, Dept. of Electr. Engn., Portland, Oregon 97207,
Tel: 503-725-5411, Fax: 503-725-4882, *mperkows@ee.pdx.edu*

Abstract

"Learning Hardware" approach involves creating a computational network based on feedback from the environment (for instance, positive and negative examples from the trainer), and realizing this network in an array of Field Programmable Gate Arrays (FPGAs). Computational networks can be built based on incremental supervised learning (Neural Net training) or global construction (Decision Tree design). Here we advocate the approach to Learning Hardware based on Constructive Induction methods of Machine Learning (ML) using multi-valued functions. This is contrasted with the Evolvable Hardware (EHW) approach in which learning/evolution is based on the genetic algorithm only.

Various approaches to supervised inductive learning for Data Mining and Machine Learning applications require fast operations on complex logic expressions and solving some NP-complete problems such as graph-coloring or set covering. They should be realized therefore in hardware to obtain the necessary speed-ups. Using a fast prototyping tool; the DEC-PERLE-1 board based on an array of Xilinx FPGAs, we are developing virtual processors that accelerate the design and optimization of decomposed networks of arbitrary logic blocks.

I. INTRODUCTION. EVOLVING IN HARDWARE VERSUS LEARNING IN HARDWARE

In recent years the scientific community is witnessing very fast developments in the area of Soft Computing. Thus, Artificial Neural Nets (ANNs), Cellular Neural Nets (CNN), Fuzzy Logic, Rough Sets, Genetic Algorithms (GA), Genetic and Evolutionary Programming and other approaches have been developed and they are concerned with the notions of learning, adapting, modifying, evolving or emerging.

Several mixed approaches are being developed that in many different ways combine elements of these areas with the goal of solving very complex and poorly defined problems that could not be tackled by previous, analytic models. What is common to all these approaches is that they propose a way of **automatic learning** by the system; the computer is taught on examples rather completely programmed (instructed) what to do. This philosophy dominates also the areas of Artificial Life, solving problems by analogy to Nature, decision making, knowledge acquisition, new approaches to intelligent robotics, and many other. **Machine Learning** becomes then now a new and most general system design paradigm unifying many previously disconnected research areas. It starts to become a **new hardware construction paradigm** as well.

Recently, a term **Evolvable Hardware** (EHW) has been coined [17, 16, 18, 20] which is, realization of genetic algorithm (GA) in reconfigurable hardware. Our approach of Universal Logic Machine [31, 33, 38, 21] proposes to build a learning machine based on logic principles, especially the Constructive Induction [26, 27, 44] and Rough Set Theory [30] approaches. While the Genetic Algorithm is a very simple and practically blind mechanism of Nature, it can be easily realizable in hardware. We are afraid, however, that this mechanism alone cannot produce good results. In contrast, the logic algorithms that use previous human knowledge are optimal and mathematically sophisticated, and their software realizations use so complex data structures and controls that it is very difficult to realize them in hardware, but they lead to high quality learning results. Since software/hardware realizations may suffer from the consequences of the Amdahl's Law, interesting software-hardware design trade-offs must be then resolved to realize optimally the learning algorithms based on logic.

When we will talk about **"Learning Hardware"**, we will understand the term "learning" very broadly, as any mechanism that leads to the improvement of operation, evolution-based learning is thus included. Although specific learning concepts and their formalisms differ from one learning approach to other, what is common is that in the process of learning some kind of network is constructed/evolved/adapted/grown that stores the knowledge acquired in the learning phase (the network can become equivalent to a state machine or fuzzy automaton by adding some discrete or continuous memory elements). The learned network is next run (executed, evaluated, etc.) for old or new data given to it, thus producing its responses - expected behaviors(decisions, controls) in unfamiliar situations (new data sets). The responses may be correct or erroneous, the network's behavior is then evaluated by some fitness (cost) functions and the learning and running phases are interspersed. The process of solving problems is thus always reduced to two phases: **the phase of learning**, which is, constructing and tuning the network, and **the phase of using knowledge**, that is, running the network for data sets. Comparing to the process of developing and using a computer, the first stage could be compared to the entire process of conceptualizing, designing and optimizing a computer on all its system, behavioral, architectural, logic design, and physical design levels (partitioning, placement,routing); and the second stage to using this computer to perform calculations. You cannot redesign the standard computer hardware, however, when it cannot solve the problem correctly, while the Learning Hardware will redesign itself automatically based on new learning examples given to it.

Let us observe, that from the operational point of view, from the entire system it is irrelevant what kind of network is being taught. It can be a combinational network, in which the outputs are some functions of the states of input signals, or it can be a network with a memory. It can be digital or analog, with discretization in signal value or in time, synchronous or asynchronous (for simplification, in this paper we will restrict ourselves to combinational digital circuits). It is only important that we have some way of designing this network by positive and negative examples and next some way of evaluating network's behavior on data sets (similar clustering methods have been also designed to acquire knowledge in a feedback from the environment and without direct intelligent supervision). Thus, the **structure** of the network must be created, and also its **elements** must be designed, adapted, selected from a menu, or tuned in the learning process. The network can be realized in software, in hardware or as **software-hardware codesign** (Amdahl' Law can be used as an argument against software/hardware approaches as opposed to a purely hardware approach).

Observe also that once the network has been found, it can be transformed to another network, either completely equivalent to it or being its generalization. For instance, an integer-based neural net or a multi-valued (MV) decision tree can be both compiled to binary logic gates. The net can be designed using constructive methods all at once from the complete set of examples (an approach used for diagnostic trees), or it can be built incrementally (like done for neural nets).

It should be clear for the reader from the above remarks, that there are **close links between various learning approaches**, thus ideas developed in one area, say ANNs, can be next mapped to other area, say Fuzzy Logic. Therefore, many new approaches can be created and investigated based on combining basic learning models and methods in various ways. For instance, the ANN built in Brain Builder's [18] approach can be directly compiled to binary hardware without the intermediate medium of cellular automata used in [18], or a learning algorithm different than the genetic algorithm from [18] can be used to construct the ANN.

It is also irrelevant from the point of view of the entire system when it has been already taught, whether its "black box", the learning module realized as, say, an array of programmed FPGAs, has been taught in an incremental learning process, or constructed as a fully specified system, or constructed by a learning algorithm. Different construction methods will only differ in their convergence speeds, sizes of networks, their learning errors, networks' speeds, testabilities, power consumption, etc. It is in the network model selection and network construction methods where the different philosophies of designing Learning Hardware and Evolvable Hardware essentially disagree.

The plan of this paper is the following. Section II will briefly compare logic versus ANN and GA approaches to learning. In section III we will introduce the concept of Learning Hardware, and in section IV we explain methods of knowledge representation in the Universal Logic Machine (ULM), our realization of the Learning Hardware concept. Section V introduces briefly the DEC-PERLE-1 board to which the virtual processors of the ULM are mapped [51]. Section VI clarifies programming/designing environment for DEC-PERLE/XILINX. Next we illustrate our approach with two different concepts of designing Learning Hardware using the DEC-PERLE-1 board. While the first method is to design a **general-purpose computer with instructions specialized to operate on logic data**, the second method is to design a **processor for only one application**.

In section VII, we present a virtual general-purpose computer that operates on multi-valued cube calculus, an algebra to solve combinatorial problems in multiple-valued logic. The data path of this computer is entirely specialized for efficient realization of cube calculus operations and its control unit implements basic algorithms that use these operations; for instance, two-level AND/OR logic minimization. The virtual processor from section VIII realizes one algorithm only: the generalized Ashenhurst/Curtis decomposition of functions [2, 7, 52] and relations [36]. The unifying concept of both these architectures is the use of **cellular automata** and **regular logic structures**, because they can be easily specified in VHDL or schematic capture tools, and nicely mapped to regular chip-level and board-level "FPGA Array" resources of DEC-PERLE-1. We point to some difficulties, evaluate the design and discuss future improvements in conclusion section.

II. LOGIC METHODS FOR LEARNING

In this paper we will present a new approach to design a learning machine, based on FPGA technology and associated logic development methods (called **logic synthesis** by the design automation community and **constructive induction** by the Machine Learning community) rather than on neural or genetic algorithms. Michie [28] makes distinction between black-box and knowledge-oriented concept learning systems in terms of **weak** and **strong** criteria. The system satisfies a weak criterium when it uses sample data to generate an updated basis for improved performance on subsequent data. Strong criterion is satisfied if the system moreover can communicate its learned concepts in symbolic form [27].

Let us observe that ANNs, CNNs and similar approaches satisfy only the weak criterium while our approach satisfies the strong criterium. For instance, a medical doctor who uses the aid of a knowledge-based system has to understand the explanation of the system to undertake his decision, for which only he will be responsible. The doctor cannot than rely on a "black box"-type of decision from the ANN. We believe that the results of the learning process, and even the process itself, should be understood by humans. The processes should be then similar to those in humans, thus based on symbolic logic and not on the methods of Nature. Human thinking is perceived by other humans as the abstract use of symbols, and not tuning of numeric weights of neurons. Our approach to learning still does allow for fuzziness, imprecision of formulation, and random search or probabilistic solving mechanisms. It operates, however, on higher and more natural symbolic representation levels. Also, the built-in mathematical optimization techniques allow to satisfy the Occam's Razor Principle, thus finding solutions that are provably good in the sense of Computational Learning Theory (COLT) [1, 44]. This is the first main point of our philosophy.

In our past research we have been using and comparing, in software, various network structures for learning: two-level AND/OR (Sum-of-Products (SOP), or Disjunctive-Normal-Forms (DNF)) [29], Exclusive-Or-Sum-of-Products (ESOP) [49, 35], Three-Level NAND/AND/OR networks [39], Three-Level AND/NOT (TANT) networks [34], decision trees (C4.5), and multi-level decomposition structures [52, 36, 13, 14], as well as various logic, non-logic and mixed optimization methods: search [37], rule-based, set-covering, graph-coloring, genetic algorithm [11, 9] (including mixtures of logic and GA approaches), genetic programming [10], artificial neural nets, and simulated annealing. We compared our networks' results on their complexity (Occam's Razor), as well as on various ways of calculating the error of learning [13, 14, 44, 24]. The Decomposed Function Cardinality (DFC) and its extensions for MV logic [1, 44, 13, 14] were used as common measures of complexity, because of its theoretically proven properties [1, 44]. Based on these investigations, we can definitely state that logic approaches and especially the MV decomposition techniques, combined with smart heuristic strategies and good data representations, are usually superior to other approaches with respect to smaller net complexity and learning error.

Based on small complexity and error, especially poor results were obtained using the genetic algorithms [11, 9, 10]. May be GA performs well in other applications, but both in our experience and based on literature we were simply not able to find a single problem domain that a GA-based algorithm would be superior to a hand-crafted human-designed algorithm to design a binary or multi-valued logic network of any kind. This is perhaps because humans have long experience in creating efficient logic minimization algorithms (for instance, more papers have been written on SOP minimization than perhaps on any other engineering topic). We want to make use of this accumulated human experience in our approach, rather than to "reinvent" algorithms using GA. On the other hand, for large data the logic algorithms are relatively slow, hence must be speed-up in hardware.

III. FROM LEARNING HARDARE TO DATA MINING MACHINES

There is one general agreement among various developers of evolvable and learning systems: that realized with current software or even parallel programming technologies, the learning phase and/or the execution phase are too slow for real-life problems, especially real-time problems, regardless whether the exhaustive combinatorial search, simulated annealing, or evolutionary algorithms that involve millions of populations are used. Thus, the researchers proposed to speed-up some phases by **migrating them from software to hardware.**

In general, five approaches to implementing the learning algorithms are possible:

A1. Both learning and execution are done in software (this standard approach still dominates the Artificial Neural Nets, Constructive Induction, Data Mining, and the so-called "extrinsic" Evolvable Hardware [16]).

A2. The learning phase is performed in software and the network is next downloaded to hardware for execution (this approach has been used with fuzzy logic controllers, FPGAs realizing binary and multi-valued networks, etc.).

A3. The learning phase is performed in hardware and the execution phase in software. This approach is thus a hardware-accelerated design of a knowledge-based expert system. (This is an approach that can be used for Data Mining (DM) and Knowledge Discovery in Databases (KDD) of extremely large data. So far, we do not know researches other than ours based on this principle).

A4. Both the learning and the execution phase are performed in hardware. This is the area of classical ANNs, CNNs and "intrinsic" evolvable hardware [16]. The approach of evolving ANNs realized with cellular automata evolved using genetic algorithms [18] requires hardware (the Intrinsic Evolvable Hardware) because the slow evaluation process must be repeated on millions population members to give the sufficiently good results. This is an extreme and purist, but very innovative and ambitious approach. Only time will show if it will be successful; but this philosophy is supported by Amdahl's Law. Satisfying Amdahl's Law would require, however, to remove software decision making from the process, which in our opinion will prove impossible.

A5. Software-hardware co-design in one or in both phases. This approach is the most prospective in our view.

Many ambitious projects based on ANNs, cellular logic, DNA, simulated evolution and biologically motivated hardware have been proposed that will be perhaps some day realized on molecular or quantum levels. However, many of them are quite impractical in **current** technologies. The following general observations related to **practical hardware realization of the Learning Hardware concept** can be made:

1. Most of the approaches to learning and evolutionary hardware use **binary Field Programmable Gate Arrays**, because simply there are no other mass-scale hardware reconfigurable (reprogrammable) and relatively inexpensive technologies widely available now. Other potential realization technologies are either too primitive and do not allow for large networks, or are in their very early development stages. For instance, the Electronically Programmable Logic Devices are too small, and Field Programmable Analog Arrays are in their current state not flexible enough but have a high potential in a longer run [42]. Chips for Cellular Neural Nets [43] and Artificial Neural Nets [25] have high potentials, but their small markets did not make them commercially successful yet. Finally, Multi-Valued FPGAs, Fuzzy Logic Programmable Arrays, and Mixed FPAAs are still in very early stages. On the other hand, binary FPGAs allow now to realize in hardware various conceptual networks, including neural, fuzzy, and decision trees. A practical task should be then to **compare FPGA-based realizations of various machine learning paradigms.**

2. Thus, because in binary FPGAs everything is realized on the level of **binary logic gates**, in our opinion, the learning process should be also performed on the level of logic gates. This level is more natural than the level of arithmetic operations of ANNs or Fuzzy Logic functions, or that of the switching transistor sequences responsible for routing connection paths. This is the second main point of our philosophy.

3. Once we decide to realize the network using logic gates in FPGA, we should **re-use all powerful EDA (Electronic Design Automation) tools that engineers have already developed** in many years in the area of digital design automation, especially for reconfigurable computers: state machines, logic synthesis, technology mapping, placement and routing, partitioning, timing analysis, etc. "Airplanes are not evolved and they do not fly like birds, they are constructed based on accumulated human knowledge and sophisticated mathematical algorithms." It seems like a nonsense to try to teach the evolvable system to do everything that was done by hundreds of thousands scientists and engineers in VLSI and microprocessor industry: Do we believe that the

Intel's Pentium chip can be evolved by examples as an ANN? Neural net uses multiplications; can even the 16-bit Booth multiplier be evolved? Cellular automata-based realizations need connection-routing algorithms, do we believe that practically useful "physical design" algorithms should be tried to evolve, while excellent algorithms for placement, routing and partitioning already exist in commercial EDA tools and can be used? De Garis' group evolves ANN realized as cellular automata for pattern recognition - while this approach demonstrates the power of evolutionary algorithms, is this the best approach to build Pattern Recognition hardware? Why to realize the genetic algorithm in hardware rather than the Tabu Search, Simulated Annealing or any other general problem-solving mechanism?

4. Occam Razor principle should be used whenever possible because only it can lead to meaningful discoveries.

Concluding, we do not believe that the "purist strategies" to evolutionary hardware, DeGaris and Brains Builder Group [17, 3, 16, 18], will be practically acceptable for most commercial applications of Learning Hardware.

Therefore, we propose here the principles of Learning Hardware that will **use previous human problem-solving experience** and apply many mathematical algorithms and problem-solving strategies rather than rely on only two generic methods of Evolvable Hardware: ANNs and GA. We believe also that all methods that exist in VLSI design, and especially, the powerful CAD and EDA tools, should be re-used in their entirety, rather than duplicated by naive low-level evolutionary algorithms. Learning/evolution should still remain as the main principle of building new generation hardware, but it should be restricted to high abstract levels. The variants evaluation/selection should be also performed at abstract levels, before mapping to low-level field-programmable resources, such as switches, for which chromosomes are long and the operation of GA is inefficient.

Our Learning Hardware approach is thus oriented towards modern FPGA technologies and similar technologies that can be predicted in a short time horizon, and is not necessarily best for future realization technologies of learning networks. The proposed by us **Learning Hardware** methodology can be summarized as follows:

1. Based on sets of examples classified to several (at least two) categories, and various network requirements (background knowledge), the hardware processors, using logic/mathematical algorithms, create the logic network description. This network can have two-, three-, or arbitrary number of levels and either **binary** or **multi-valued variables** (attributes, signals). It can use **simple gates** such as ANDs, ORs and EXORs, or **complex gates** such as arbitrary 4-input, 2-output Lookup-Tables. In ULM, to realize (construct, design, learn, evolve) the network we use hardware realization of well-known logic synthesis algorithms such as: two-level AND/OR minimizers [29], two-level AND/EXOR minimizers [49], three-level OR/AND/OR minimizers [34], and functional (Ashenhurst-Curtis) decomposers [36].

2. The (quasi)optimally constructed network is mapped to standard FPGAs and realized using standard partitioning, placement and routing and other EDA tools from Xilinx and commercial EDA software companies.

3. The knowledge of the machine is stored in memory patterns representing logic nets. Under supervision of the software program in the main processor, the hardware multiplexes between various learned nets, depending on rules that also can be acquired automatically. This phase is therefore similar to the CBM approach [18].

4. As the network solves new problems, the new data sets and training decisions are accumulated and the network is repeatedly automatically redesigned. The old network can serve as a redesign plan for the new network, or the net is "redesigned from scratch" to avoid any bias.

Thus, we replace the process of evolving on all design levels used in EHW, with the **ULM model of learning** at high level and next compiling to low level using standard FPGA-based tools. This can be used in approaches A3 - A5 above.

Observe also that the same physical FPGA resources are multiplexed to implement the virtual human-designed **"learning hardware"** and the automatically learned **"data hardware"**. While the "learning hardware" is designed once by humans and cannot be changed, the "data hardware can be permanently modified.

We presently model our algorithms in software or implement them for a prototype reconfigurable platform from DEC, the DEC-PERLE-1 board. We consider the ULM to be an early prototype of **Data Mining machines**, that some day will be able to collect data from on-line data bases, for instance from WWW Pages and the Internet. Other variants of

	Brain Builder	Universal Logic Machine
Model of Learning	Artificial Neural Net	Multi-Valued Logic Language
How the net is constructed	Genetic Algorithm, ANN Training	Multi-Valued Logic Synthesis, Constructive Induction, Rough Set Theory
Virtual intermediate representation	Cellular Automata	MV logic networks and state machines with arbitrary structures and arbitrary operators realized as look-up tables
What is learned:	automata tables	language expressions
Net construction realized in:	hardware	hardware and software
Mapped to:	array of binary FPGAs	array of binary FPGAs
Hardware platform:	Xilinx 6000 series CBM	Xilinx 3090 + on-board memory DEC-PERLE-1 board + DEC workstation

Table 1: Brain Builder versus Universal Logic Machine.

such machines will acquire data from industrial agricultural, military, or other application areas in real-time, using pre-processing techniques of Image Processing and Digital Signal Processing through sensors, microphones and TV cameras. In contrast to similar projects, our goal is not to build the Artificial Brain [3, 17], a superintelligent robot-pet, or a model of instinctual animal behavior, but rather to develop a system being able to perform **meaningful discoveries** in narrowly defined areas, thus speeding-up both the learning and execution phases of application software programs that are now being used in Machine Learning, Knowledge Discovery from Databases, Data Mining, and robotics.

With this respect, as an accomplishment of our project we would treat the system being able to solve in few seconds and with error as small as the learning error of our current software, the following problems:

1. Every data set from the U.C. Irvine Repository of Data Mining benchmarks [50], as well our benchmarks [41]. Including all versions of large examples such as Breast Cancer, and Michalski's Trains with 30 trains [19]. They take now up to 30 minutes in software.

2. The recognition of cervical mucus fearning microscope images for ovulation prediction [32]. It takes now minutes in software.

3. The recognition of 3-dimensional images of rooms and corridors for mobile robot orientation [47]. This task takes now up to 17 minutes in software.

Table 1 shows the comparison of Brain Builders' CAM-Brain Machine (CBM) and Universal Logic Machine approaches.

IV. MULTI-VALUED LOGIC LANGUAGE TO REPRESENT THE LEARNING DATA IN HARDWARE

Because we want the system to learn on some higher level than that of elementary gates and their connections, we need first to develop certain higher-level language, in which expressions, the virtual nets, will be automatically created, evaluated, selected and optimized, to be next realized as hardware FPGA nets by top-down automatic design methods. Since in the learning phase we want to operate on elements of this language in hardware, our choices are limited because of the necessity of operations that are easily realizable in hardware.

Several such languages have been created in the past, mostly for applications in Logic Synthesis, Automatic Theorem-Proving, Data Base Theory, and Information Engineering, and we adopted some of them for hardware representation. They include: binary and Multi-Valued Cube Calculus [8, 33], Decision Tables, Rough Sets [30], Rough Partitions [22, 23], Labeled Rough Partitions [19], Binary [5] and Multi-Valued Decision Diagrams [46]. Observe that in ML, DM and KDD these functions are very strongly unspecified (99% of don't cares, or more). Although Decision Diagrams seem to be nowadays the most successful representation of discrete data and their applications span the whole spectrum of modern Computer Science, we (and other researchers) were not able so far to find good hardware architectures to process them efficiently. Therefore we restricted our attention to the **tabular representation of data** [6] that found their applications in Logic Synthesis, state machines, automatic theorem proving, data base theory, Rough Sets, and pattern recognition. An example of such**two-dimensional tabular representation** is shown in Table 2.

	x_1	x_2	y_1	y_2
a	0,2	1	-	2
b	0,1	0	0,2	1
c	2	0	1,2	0
d	1	1	1,2	2

Table 2: MV multi-output relation.

Rows correspond to objects a, b, c and d and columns to input variables (attributes) x_1 and x_2, and output variables y_1 and y_2. Symbol y_1 denotes a relation output. Inputs x_1 and x_2 together with output y_1 specify an (oriented) relation. Relation can be used to express such facts as: this color is *red* or *white* but not *yellow* or *black*. Symbol y_2 denotes a function output, $y_1(x_1, x_2)$. Rows c and d have only one value for each attribute, so they are *minterms*. Rows a and b have more than one value for attributes, so they are *cubes*. Eeach row can be thought of as a record from a data base, or their set, or a collection of image features after image preprocessing. Dash in y_1 is a *standard don't care*, it corresponds to any possible value of this variable (or to all possible values of this variable in another interpretation of cubes). All other entries in column y_1 are called *generalized don't cares*; they corresponds to some subsets of possible values of this (ternary) variable. Comparing rows and columns of such table can be done partially in parallel and can serve to find certain patterns in data. Such patterns can be used to: generate prime implicants, decompose function, find a "bound set" or "free set" of variables for decompositin [36], remove redundant variables, find essential variables, find essential implicants, etc. Finding and analyzing patterns in such tables is a subject of Rough Sets Theory [30], Logic Synthesis [8, 46], and Data Base Theory [6]. Many similar or competing algorithms for the same task have been developed in these areas independently for the tabular data model. Basic operations of algorithms remove rows or columns, copy and modify them, merge rows or columns, etc.

The main advantage of two-dimensional model is that it can be partitioned regularly to smaller parts. For instance, smaller tables can be extracted as scannable windows in the big table, similarly as it is done in convolution-based algorithms of Image Processing and Digital Signal Processing. Most often, however, the two-dimensional representations are partitioned to one-dimensional representations. This can be done vertically or horizontally.

The main advantage of **one-dimensional representations** is that they can be efficiently processed in one-dimensional cellular automata, systolic, ping-pong, SIMD and pipelined hardware architectures. **Regularity** is the key to success in "Array of FPGAs" environment where routing long connections is the main design bottleneck. This environment is similar to Cellular Automata (CA), but does not require all automata to be the same or to be entirely regularly connected. More flexibility exists thus in this model than in the CAs, which are a **very restricted design environment.**

In addition, these one-dimensional representations resemble **chromosomes** in Genetic Algorithms, which allows to use them in evolutionary computations.

The two-dimensional representations composed from one-dimensional strings include the following.

Standard Binary Cube Calculus (CC) of Roth, Karp, and Dietmeyer [8, 33]. It represents product terms as cubes where the state of each input variable is specified by a symbol: positive (1), negative (0), non-existing (a don't care) (X), or contradictory (ϵ). Each of these symbols is encoded in **positional notation** with two bits as follows: 1 = 01, 0 = 10, X = 11, ϵ = 00. For instance, the positional notation for cube 0X1 is 10-11-01. Dashes have no hardware meaning, they only help the reader to separate visually the variables. Thus, each position represents a state of the variable by the presence of "one" in it: left bit - value 0, right bit - value 1. This encoding presents therefore a simple reduction to set-theoretical representations. A cube can represent a product, a sum, a set of symmetry coefficients of a symmetric function, a spectrum of the function, or another piece of data on which some symbol-manipulation (usually set-theoretical) [8, 33] operations are executed. Usually the cube corresponds to a product term of literals. For instance, assume the following order of binary variables: *age*, *sex* and *color_of_hair*. Assume also that the discretization of variable *age* is: $age = 0$ for person's $age < 18$ and $age = 1$ otherwise. Men are encoded by value 0 of attribute *sex* and women by value 1. *color_of_hair* is 0 for black and 1 for blond. Then a blond woman of age 19 is denoted by 110 and a black-hair seven-years old person of unknown sex is described by cube 0X1. Cube XXX is the set of all possible people for the selected set of attribute variables and their discretized values. Two-dimensional representation is just a set of cubes where the connecting operator is implicitly understood as: OR for SOP; EXOR for ESOP; concatenation for a spectrum, or other. For instance, assuming each cube corresponding to AND operator and the OR being the connecting operator; the list {0X1,110} is the SOP which represents the above mentioned two people (or a set of all people with these properties). Multi-valued and integer data can be encoded with binary strings in this representation, so that next all operations are executed in binary (we use this model in the decomposition machine). For instance, if there were three age categories, *young*, *medium* and *old*, they can be encoded as values 0, 1 and 2 of the ternary variable *age*, respectively. Variable *age* could be next represented in hardware as pair of variables age_1 and age_2, where $0 = 00$, $1 = 01$, $2 = 10$, thus encoding: $young = \overline{age_1}\ \overline{age_2}$, $medium = \overline{age_1}\ age_2$, $old = age_1\ age_2$,. Recall that the minimal hardware operations in CC are executed only on binary variables.

Multi-Valued Cube Calculus (MVCC) [48, 33]. This is a superset of CC. It represents product terms as cubes where each input variable can have a **subset** of a finite set of all possible values that this variable can take. Each element of the set is represented by a single bit, which makes this representation not efficient for large sets of values. This system is the superset of the Standard Binary Cube Calculus. In the above example we could have for instance a 5-valued variable *age* for five age categories, and a quaternary variable $color\_of\_hair$. Each position of a variable corresponds to its possible value. For instance, 10000-10-0100 describes a 7-year old boy with black hair. This is an example of a minterm cube, i.e. with single values in each variable. 01100-11-1100 describes group G_1 of people, men and women, that are either in second or in third age category and have either blond or black hair. This is an example of a cube that is not a minterm. 100000-00-1000 describes a first-category-of-age person with blond hair who has some conflicting information in *sex* attribute, for instance a missing value (this is also how contradictions are signalized during cube calculus calculations [33]). The hardware operations in MVCC are done directly on such MV variable cubes so that the separate encoding to binary variables is not necessary.

Generalized MV Cube Calculus (GMVCC) [33]. This is a superset of MVCC. It has cubes where each **output variable** can be also a subset of values. Such cubes can be directly used to represent MV relations, as in Table 2. Its operations are more general than MVCC, because more interpretations can be given to cubes. This calculus has more descriptive power, but the respective hardware processors are much more complicated.

Simplified Binary Cube Calculus (SBCC). This is a subset of CC. It operates only on **minterms**. It has application in decomposition of functions. The hardware of this machine is much simplified: operations are only set-theoretical. This is the simpliest virtual machine realized by us, so larger data can be processed by it because more of a machine can be fit to the limited FPGA Array resources of DEC-PERLE-1.

Simplified MV Cube Calculus. It has cubes where for every input variable either **only a single value** of its possible values is selected (which is denoted by a binary code (such as a byte) of a symbol corresponding to this value), the variable is missing (which is denoted by a selected symbol, X), or the variable is contradictory (another symbol, $\emptyset$). This representation is used for Rough Sets [30] and variable-valued logic [26]. For instance, assuming 10 age categories, 0 = 0 - 10 years, 1 = 10 - 19 years, 2 = 20 - 29 years, etc, and 3 hair categories: 0 = blond, 1 = black, 2 = red, the 7-year old boy with black hair is described as 0-0-1, the 18-year old girl with black hair is described as 1-1-1, the 28-year old woman with red hair is described as 2-1-2, and a set of all people with red hair is X-X-2. There is no way now to describe in one cube people below 19 with red or black hair, which was possible in MVCC or GMVCC. This simplification of the language brings however big speedup of algorithms and storage reduction when applied for data with many values of attributes. Also, the control of algorithms becomes more complicated, while the data path is simplified.

Spectral Representations. Examples: Reed-Muller FPRM and GRM spectra [11, 9], Walsh spectrum [12], various orthogonal spectra. These representations represent function as a sequence of spectral coefficients or selected coefficient values with their numbers. Some spectral representations are useful to represent data for genetic algorithms: the sequence of spectral coefficients is a chromosome. For instance, in the Fixed-Polarity Reed-Muller (FPRM) canonical AND/EXOR forms for n variables, every variable can have two polarities, 0 and 1. Thus there are 2^n different polarities for a function and the GA algorithm has to search for the polarity that has the minimum number of ones in the chromosome. This way, every solution is correct, and the fitness function is used only to evaluate the cost of the design (100% correctness of the circuit is in general very difficult to achieve in GA [11, 9, 10]. Therefore our approaches to logic synthesis based on GA are to **have a representation that provides you with 100% correctness** and have the GA search only for net minimization. This approach involves however a more difficult fitness function to be calculated in hardware than the pure GA or Genetic Programming approaches [11, 9, 10]. Let us observe that evaluating the fitness function is much more difficult to realize in hardware than all other operations of the GA combined, and logic transformations are necessary for this to achieve. Similarly, the other AND/EXOR canonical form called the Generalized Reed-Muller form (GRM) has $n \cdot 2^{n-1}$ binary coefficients, so there are $2^{n \cdot 2^{n-1}}$ various GRM forms. Because there are more GRM forms, it is more probable to find a shorter form among them than among the FPRM forms [11, 9]. But the chromosomes are much longer and the evaluation is more difficult. This kind of trade-offs is quite common in spectral representations. Spectral methods allow for high degree of parallelism.

Rough Partitions (RP) represented as Bit Sets [22, 23]. This representation stores the two-dimensional table column-wise, and not row-wise as MVCC does. In r-partition every variable (a column of a table) induces a partition of the set of rows (cubes) to blocks, one block for each value the variable can take (there are two blocks for a binary variable, and k blocks for a k-valued variable). Rough Partitions are an interesting and novel idea but they don't really form a representation of a function. Since the values of a variable are not stored together with partition blocks, the essential information on the function is lost and the original data can not be recovered from it. This is kind of an abstraction of a function, useful for instance in various decomposition algorithms.

Labeled Rough Partitions (LRP) represented as Bit Sets [19]. This is a new representation (a generalization of RS) which has very interesting properties and allows to find different kind of patterns in data. It is useful for decomposition of MV relations and it preserves all information about the relation or function. It can be also made canonical, when created for special cubes. Most of its operations are reduced to set-theoretical operations, so hardware realization is relatively easy. Relations happen in tables created from real data-base and features from images, for instance, MV relations are benchmarks `hayes`, `flare1`, `flare2` from [50]. An example of application of relation in logic synthesis area is a modulo-3 counter (a non-deterministic state machine is a special case of multiple-valued, multi-output, relation) that counts in sequence $s0 \rightarrow s1 \rightarrow s2 \rightarrow s0$ and if the state $s3$ happens to be the initial state of the counter, counter should transit to any of the states $s0, s1, s2$, but not to the state $s3$ itself.

Generalized values for input variables are already known from cube calculus but generalized values for output variables are a new concept which allows for representation and manipulation of relations in LRP.

Definition 1. Separation of the elements of a nonempty set S into nonempty subsets S_i, $\bigcup S_i = S$, is called a *rough partition* (r-partition) of S.

Notice that the definition of a rough partition allows subsets S_i to overlap.

Definition 2. [relation]. Let S_1 and S_2 be sets. A *relation* R from S_1 to S_2 is a subset of Cartesian product $S_1 \times S_2$. A *relation* R *on* S_1 is a subset of $S_1 \times S_1$.

Function is a special case of relation from S_1 to S_2 where every element $s_1 \in S_1$ is the first member of precisely one ordered pair $(s_1, s_2) \in S_1 \times S_2$.

Definition 3. [labeled partition block] Let $C(X)$ be a set of MV cubes, and relation R_k be defined by a cube $c_k(X_1), X_1 \subseteq X$, as follows: $c_i(X) R_k c_j(X)$ iff $c_k(X_1) \subseteq c_i(X_1)$ and $c_k(X_1) \subseteq c_j(X_1)$, where $c_k(X_1)$ is given and $c_i(X), c_j(X) \in C(X)$. The set of all cubes $c_i(X)$ being in relation R_k to each other and labeled by the cube $c_k(X_1)$ will be called *labeled partition block* and denoted by $B_{c_k(X_1)}$. The cube $c_k(X_1)$ will be called a block label.

Every cube in $C(X)$ can be enumerated with a different symbol (an integer number in particular) and consequently, a partition block represented by a set of symbols. Label added to the partition block allows for establishing a correspondence between the set of symbols in the partition block and cubes in $C(X)$.

In Table 2 we have: $X = \{x_1, x_2\}, Y = \{y_1, y_2\}$, and $C(X \cup Y) = \{a, b, c, d\}$, where a, b, c, d are symbols denoting cubes $c_1(X \cup Y), c_2(X \cup Y), c_3(X \cup Y)$,and $c_4(X \cup Y)$ respectively. Let $X_1 = \{x_1\}$. Then $c_1(X_1) = \{\{0\}\}$ defines relation $R_1 = \{a, b\}$, cube $c_2(X_1) = \{\{1\}\}$ defines relation $R_2 = \{b, d\}$, and cube $c_3(X_1) = \{\{2\}\}$ defines relation $R_3 = \{a, c\}$. Corresponding labeled partition blocks are $\{a, b\}_0$, $\{b, d\}_1$, and $\{a, c\}_2$.

Definition 4. [labeled rough partition]. The collection of nonempty labeled partition blocks $B_{c_k(X_1)}$ forming a rough partition of the set $C(X)$, $X_1 \subseteq X$, will be called *labeled rough partition* (lr-partition) and denoted by $P(X_1) = \{B_{c_k(X_1)}\}$.

Notice that every lr-partition $P(X_i)$, $X_i \subseteq X$, forms a cover of the set of symbols enumerating the cubes in $C(X)$.

Given the example from Table 2 we have $P(X_1) = P(x_1) = \{\{a, b\}_0, \{b, d\}_1, \{a, c\}_2\}_{x_1}$.

Definition 5. [labeled partition block product]. Product of two labeled partition blocks $B_{c_i(X_1)}$ and $B_{c_j(X_2)}$ is the labeled partition block $B_{c_k(X_3)}$, which is an intersection of partition blocks $B_{c_i(X_1)}$ and $B_{c_j(X_2)}$ ($B_{c_k(X_3)} = B_{c_i(X_1)} \cap B_{c_j(X_2)}$) and label $c_k(X_3)$ is equal to $c_i(X_1) \$ c_j(X_2)$.

Definition 6. [lr-partition product]. The product $P(X_1)P(X_2)$ of lr-partitions $P(X_1)$ and $P(X_2)$ of a set of cubes $C(X)$, $X_1, X_2 \subseteq X$, is lr-partition

$P(X_3), X_3 = X_1 \cup X_2$, the blocks of which are non empty products of the blocks of $P(X_1)$ and $P(X_2)$.

Theorem. For any set of cubes $C(X)$, and any set of subsets X_i of X, $P(\bigcup_i X_i) = \prod_i P(X_i)$.

Characteristics of lr-partition representation can be summarized as follows:

1. Multiple values of both input and output variables can be easily represented. This is especially important in ML and complex Finite State Machine (FSM) controller optimization applications to express uncertainty of choice of variable's value.

2. It can easily handle situations where a variable is not present in a given cube (Michalski's train benchmark [26] and '~' in Espresso format).

3. By selection of sets X_i and Y_j lr-partitions can be dynamically adjusted to a given type of data (completely vs. incompletely specified, many cubes vs. few cubes) to minimize memory requirements (see Theorem 1).

4. lr-partitions can be used for decomposition of large functions and relations, we implemented a decomposer of MV relations which can decompose large functions and relations from ML and controller domains. It was shown in [36] that this representation is not only compact but also allows for a fast processing.

All these representations have certain advantages and disadvantages, depending on type of data processed and the algorithm realized in hardware. For instance, among our virtual machines, the **Cube Calculus Machine** realizes Multi-Valued Cube Calculus, the **Decomposition Machine** realizes the Simplified Binary Cube Calculus, and the **Rough Set Machine** realizes the Simplified MV Cube Calculus. This way, we will be able to compare hardware realization of various representations and operations for the same or similar tasks to understand better the trade-offs between generality and efficiency. We will illustrate also the concept of ULM with two kinds of approaches: CCM will represent a **general-purpose computer with a special list of instructions for MVCC** [33]. It is microprogrammed in a special language called **CCM Assembly**. Decomposition Machine is a **processor for onle application only**: functional decomposition. It is thus completely hardwired and optimized just for this task to make it as efficient as possible.

V. DEC-PERLE-1 BOARD FOR FAST PROTOTYPING

In this section we will present the minimum description of our board, to give the reader some feeling about fast prototyping environment based on arrays of FGPAs and show difficulties that exist in creating Learning Hardware. Digital's Paris Research Laboratory developed its third generation board, **DEC-PERLE-1** in 1992. The overall structure of the DEC-PERLE-1 is shown in Figure 1. The board is organized around a central computational matrix made up of 16 Xilinx XC3090 LCAs[1] (M00 to M15 in the figure), surrounded by a four 1MB RAM banks, and 7 other LCAs to implement switching and controlling functions. The data buses and their width are also shown in the figure. The user has to understand well all programmable resources of the board, otherwise the logic design becomes non-mappable to FGPGA wiring resources. Moreover, the designer needs to take into account this architecture from the very beginning of designing hardware rather than to design first and next try to map. Regularity is the key issue.

Computational matrix. The central computational matrix is a 4×4 matrix of Xilinx 3090 LCAs. These LCAs are interconnected with each other. The LCAs are named LCA_M00 to LCA_M15. This matrix can be used to develop any kind of digital circuitry: data path, control unit and others. But it is typically used to develop the data path of the application. The interconnection resource between them can be classified into the following three categories: direct connections, buses, and rings. **Direct Connections.** These wires connect the adjacent sides of adjacent LCAs. The main purpose of direct connections is to extent the internal regularity of the LCA to the matrix level. The matrix can be seen as a large FPGA with 64×80 Configurable Logic Blocks (CLBs) (one XC3090 FPGA has 16×20 CLBs). Each LCA has 16 such wires on each side.

The FPGA matrix is shown in Figure 2: this figure shows the regularity and the local and global connections. It allows the reader to evaluate the complexity of designs that can be practically implemented in DEC-PERLE-1. The direct connections at the edges of the FPGA matrix four 64-bit-wide connections connected to external connectors, which can be used to connect other devices, for example, another DEC-PERLE-1 board. **Buses.** The horizontal or vertical wires connect the corresponding side of all 4 LCAs in the same row or column. They can thus efficiently distribute global data in one direction, and are comparable to the **longline** interconnections resources in Xilinx internal architecture. Each LCA has 16 such wires on each side. According to their

[1] LCA stands for Logic Cell Arrays

directions, these buses are named **matrix North, East, South** and **West bus**, respectively, and represented by MBusN, MBusE, MBusS and MBusW for short. Each bus has 64 wires which are connected to switches on the corresponding side of matrix FPGAs.

Rings. These wires connect all the matrix LCAs and two control LCAs. These connections are very useful for **global control signals** distribution since they connect to all the matrix LCAs. There are 10 such wires. Note that because of their electrical loading (they are used to connect 18 LCAs, 16 matrix LCAs and two control LCAs), these wires are slower than the buses and should be used with care in high performance designs.

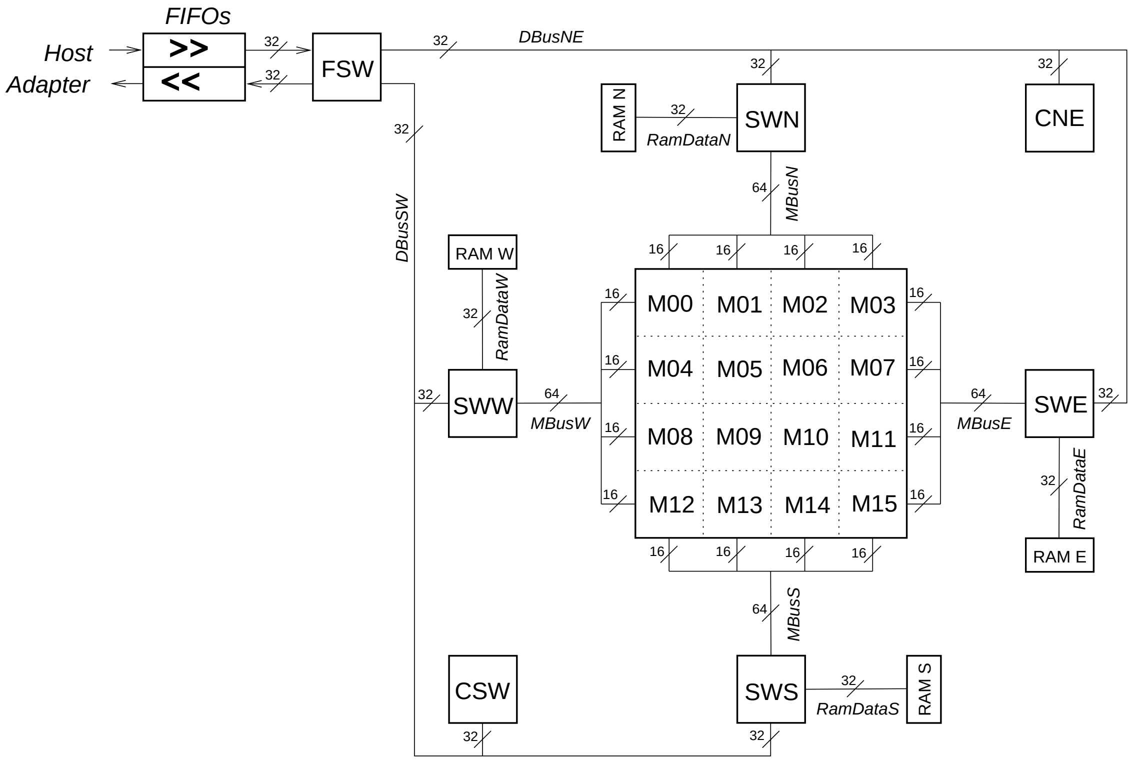
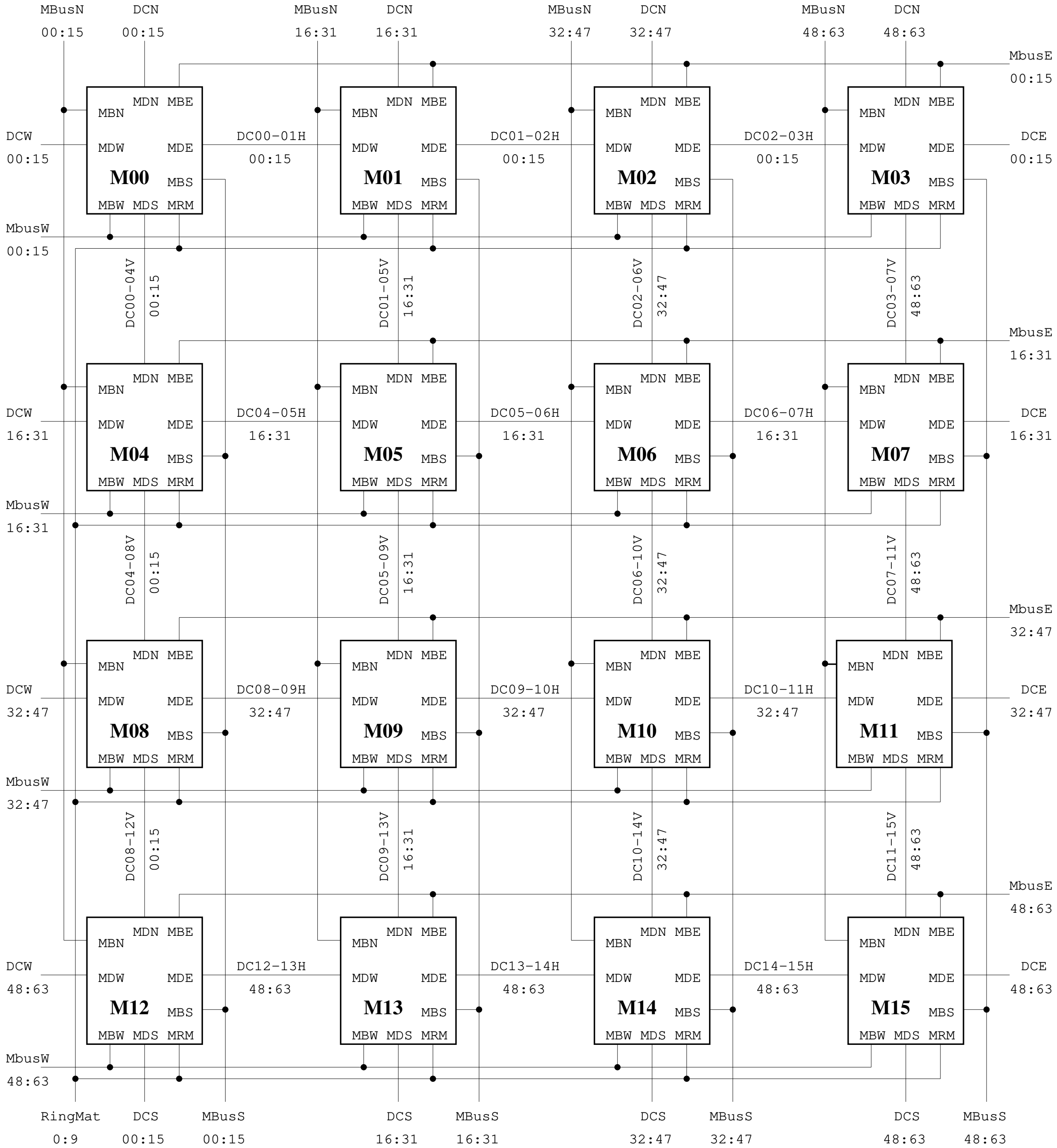


Figure 1: DEC-PERLE-1 architecture

Switches and I/O buses. FIFOs, RAM banks and the central matrix are connected through two 32-bit data buses and five programmable switches (FPGAs). There is one matrix switch on each side of the matrix, respectively called **North Switch (SWN)**, **East Switch (SWE)**, **South Switch (SWW)** and **West Switch (SWW)**, which connect the corresponding matrix data buses and corresponding RAM banks. These 4 switches (SWN, SWE, SWS, SWW) also connected to two 32 bits I/O buses, called **North-East Bus** (DBusNE) and **South-West Bus** (DBusSW) after the names of the switches they respectively connect. Two I/O buses (DBusNE, DBusSW) connect to the input and output FIFOs through the fifth switch called **Fifo Switch** (FSW), and also connect to corresponding controllers, called **North-East Controller** (CNE) and **South-West Controller** (CSW). As their names imply, the FPGAs CNE and CSW are typically used to develop the controller of the application because they connect to all other parts of the DEC-PERLE-1, FIFOs, Memory banks and other FPGAs.

Control resource. The control resource is the programmable resource that can be used to develop the control part of the application other than data path part. The data path resource (matrix, RAM banks, FIFOs and switches) needs the following set of control wires: **MATRIX RINGS**: There are 10 matrix global wires. **RAM ADDRESS**: Each RAM bank has a 18-bit-wide address, that specify the word address of the current read or write operation. Since our CCM design uses two of four memory banks, two addresses are used in our CCM design. **RAM CONTROLS**: Each RAM bank has 4 control signals. **SWITCH CONTROLS**: Each pair of matrix switches (**North** and **East** / **South** and **West**)



DCN, DCE, DCS and DCW: North/East/South/West matrix side to connectors
MDN, MDE, MDS and MDW: Matrix North/East/South/West direct connections
MBN, MBE, MBS and MBW: North/East/South/West matrix buses

Figure 2: DEC-PERLE-1 matrix

has 10 control wires that are the equivalent of the matrix rings, and are called *switch ring*. The **Fifo Switch** has 6 control wires. In addition, each of the matrix switches has 2 dedicated control wires. **FIFO CONTROLS**: Each of the two FIFOs has one status wire: empty flag for input FIFO / full flag for output FIFO; and one control wire: write for output FIFO / read for input FIFO. **TAGS**: Four “tag” wires along the input data wires on the input FIFO. **CLOCK CONTROL**: The clock generator has two control wires that can be driven by the application design. **LCBus**: There is a 24-bit-wide communication path between the board and the host, called **LCBus**. All these control wires are connected to one of two controller LCAs (CNE, CSW) or both of them. These two controllers are identical except that each of them controls two of the four switches and memory banks. These two controllers also connect to corresponding I/O bus in order for it to be able to communicate with the main datapath.

Memory subsystem. DEC-PERLE-1 contains 4MB of high-speed static RAM organized in 4 banks of 256K 32-bit words (4 bytes a word). These banks are named **North, South, East** and **West** after the matrix switch to which they are connected. Each bank is completely independent of the others and has its own data, address and control signals: **DATA BUS**: 32 data wires connect to the corresponding matrix switch. They are represented by *RamdataX*, where *X* is one of **N,S,E,W** to respectively specify the North, South, East or West RAM bank. **ADDRESS BUS**: 18 address wires ($1MB = 2^{20} = 2^{18} \times 2^2$) connect to the corresponding controller. They are represented by *RamAddrX*. **CONTROL BUS**: 4 active-low control signals to specify the read/write operation, connect to the corresponding controller. $\overline{RamReadX}$: read command. $\overline{RamWriteX}$: write command. $\overline{RamDisLowX}$: disable lower half-word (bits 0 to 15). $\overline{RamDisHighX}$: disable upper half-word (bits 16 to 31). In our CCM design, we always read/write memory by a 32-bit word a time. Therefore, the signal $\overline{RamDisLowX}$ and $\overline{RamDisHighX}$ are always set to 1 (not actived).

The basic read and write transactions both last one clock cycle, and either may be performed at every cycle.

Read memory. To read a particular word of memory, the word address (*RamAddrX*) must be presented and the read command ($\overline{RamReadX}$) must be asserted at the beginning of a cycle; the data word read from memory will be available on the data wires at the end of the same cycle and may be latched on the next clock tick. A RAM bank can be seen as a combinational device when read.

Write memory. To write a particular word of memory, the word address (*RamAddrX*), the data (*RamDataX*) and the write command ($\overline{RamWriteX}$) must be asserted during the same cycle; the word will have been written by the end of the same cycle, and the address and the data may be removed after the next clock tick. The reading or writing of either half of the data word may be independently disabled by asserting the corresponding disabled command ($\overline{RamDisLowX}$ or $\overline{RamDisHighX}$) during the transaction cycle. The memory system is clocked by *clock1* signal.

Clock subsystem. Two global synchronous clock signals, **clock0** and **clock1**, are available to **all** DEC-PERLE-1 LCAs for proper synchronous operation. These clock signals are generated by a **phase-locked-loop oscillator** synchronized to the host bus master clock. When DEC-PERLE-1 is connected to a DEC 5000/24 workstation (25MHz TURBOchannel), its frequency can be programmed under software control to be any value from 360 KHz to 120 MHz, with an average resolution of 0.01%.

Clock modes. Under software (the program running on the host) control, the clock generator may be put in the following operation modes: **STOP MODE**: No clock is generated in this mode. **FREE-RUN MODE**: This is the normal operating mode, where the clock continuously runs at the prescribed frequency. **BURST MODE**: This is a mode where, under software control, the clock generator will generate a burst of 1 to 31 clock ticks at the prescribed frequency, then stop. This is useful to implement step and double-step debugging modes. **AUTOSTOP MODE**: There are two autostop modes: *FifoIn-Autostop* and **FifoOut-Autostop**. In the *FifoIn-Autostop* mode, *clock0* will automatically stop whenever the design attempts to read an empty input FIFO. Similarly, in the *FifoOut-Autostop* mode, *clock0* will automatically stop whenever the design attempts to write a full output FIFO. These two modes can be enabled at the same time. For instance, the CCM design runs in this mode. **CLOCK1-DIV2**: This mode is useful for very high performance designs. *clock1* runs at half the speed of *clcok0*. This allows the RAM and FIFOs to be operated on half the speed of the matrix. **clock0 stop.** The *clock0* may stop under control of the application on the board. This is usually used to implement **flow-control**, where the entire datapath is stopped waiting for input data (when the input FIFO is empty) or output space (when the output FIFO is full). It is much more efficiently and easily implemented this way than through the global distribution of a clock enable signal. In effect, when application runs entirely on *clock0* and both autostop modes are enabled, the application can be seen as a perfect synchronous system without flow-control concern. The *clock0* signal will stop under one or more of the following conditions: (1)The active-low $\overline{ClkStop}$ signal is asserted from one of the controllers. (2) In the *FifoIn-autostop* mode, the input FIFO is empty and the active-low $\overline{FifoInRead}$ signal is asserted from one of the controllers. (3) In the *FifoOut-autostop* mode, the output FIFO is full and the active-low $\overline{FifoOutWrite}$ signal is asserted from one of the controllers. The memory subsystem and the FIFOs are clocked by *clock1*. This means that it is still possible to perform memory and/or FIFO operations even when *clock0* is stopped.

Slow mode. Under control of an application on the board, it is possible to slow down the clock (divide its frequency by 4) by asserting the active-low $\overline{ClkSlow}$ signal from one of the controllers. This is useful when an application can run at a very high speed, but must infrequently perform an operation that is impossible to be performed at the high speed (like stopping the clock, or accessing the FIFOs). The $\overline{ClkSlow}$ can be asserted at any speed, but its operation is asynchronous, that is, it will take an unpredictable number of cycles for it to be effective. If the operation frequency is less than 80 MHz, this number of cycles is however guaranteed to be less than or equal to 6.

Host interface. The DEC-PERLE-1 application is running under the control of the software program executed on the host computer. The communication between DEC-PERLE-1 application and its driving software program can be done through FIFOs or LCBus.

FIFOs. There is a 32-bit-wide, 512-word-deep FIFO in each direction These FIFOs are called **input FIFO** for the Host-to-PAM direction and **output FIFO** for the PAM-to-Host direction, respectively. On the application side, their data wires are connected to the **Fifo Switch** LCA and their control wires to the two Controller LCAs. Both FIFOs are purely synchronous devices when operated from the application side. They appear to be always available for reading or writing in *autostop* mode. The input FIFO and output FIFO are synchronous devices that offer two active-low status signals $\overline{FifoInEmpty}$ and $\overline{FifoOutFull}$ and two active-low command signals $\overline{FifoInRead}$ and $\overline{FifoOutWrite}$. These four signals are connected to the two Controller LCAs CNE and CSW.

The input FIFO can be written and the output FIFO can be read by the driving software through the runtime library.

LCBus. The LCBus is a 24-bit-wide general purpose register that can be read and written by both the software and the application design. The LCBus can be used for asynchronous communication between the Controller LCAs and the software program. Under the software control, the direction of each bit can be set independently of the others. Initially (after download), all bits are set for PAM-to-Host communication.

Tags. Every word that the software (the program running on the host) pushes into the input FIFO is **"tagged"** with 4-bit value. These tag bits are read from the input FIFO at the same time as the data word, and are available on both Controller LCAs and on the **Fifo Switch**.

The user of the board has to know the delays of different kinds of connections, so that he can make reasonable trade-off decisions for his designs. For instance, the delay of matrix rings is 43ns, and the delay of matrix direct connection is 24ns. For a given signal, if the designer can use either the matrix rings or the matrix direct connection, then the matrix direct connection should be a better choice. It would be very difficult to have a GA make good timing decisions.

The above described hardware resources have been created for a class of applications, so they are not necessarily optimal for any particular application. The very useful features in designs are: large memories, vertical and horizontal buses and direct connections, global connections, clock control modes and debugging modes. However, the designer is often confronted with too few connections in FPGA resources to map his virtual architecture. This requires frequent modifications, or may require a total redesign. The most difficult are architectures as CCM, which have many buses and many global signals between control units and data paths.

Concluding, DEC-PERLE-1 board, similarly to other FPGA boards, advocates very regular design styles without long and many control signals. It is then good for small SIMD processors, pipelining, systolic processors, cellular automata or complex Boolean functions. The basic design principle is: **"map two-dimensional tables to two-dimensional logic resource arrays"**. The design can be developed incrementally thanks to its easy memory access, host interface with FIFOs, and the clock debugging modes and tags.

VI. PROGRAMMING THE DEC-PERLE-1 BOARD

For using DEC-PERLE-1 board, we must run an application-specific program on the host computer which connects to the DEC-PERLE-1 board. On the other hand, the 23 FPGA chips of the DEC-PERLE-1 must be programmed to realize an application-specific hardware. Therefore, A DEC-PERLE-1 program consists of two parts:

- the **driving program** which runs on the host and controls the DEC-PERLE-1 hardware.
- A 1.5 MB **bitstream** which programs the 23 XC3090 FPGAs of the DEC-PERLE-1 to realize an **application-specific hardware**.

The driving program is written in C or C++ and is linked to the runtime library encapsulating a device driver. The requirement for developing the driving program is the C or C++ programming environment and the DEC-PERLE-1 runtime library.

The runtime library. The runtime library of DEC-PERLE-1 is essential to the developer who develops the **driving program** which runs on the host computer and controls the DEC-PERLE-1 hardware for the application. The runtime library is the only way to access DEC-PERLE-1 hardware for the driving program. The runtime library developed by DEC's Paris Research Laboratory provided a few essential controls to the application driving program: (1) A UNIX I/O interface, with open, close, read and write. (2) Download the configuration bitstreams from host to DEC-PERLE-1, and/or read back the values of all the flip-flops of all the LCAs. (3) Read/write static RAM on DEC-PERLE-1 by the software program. (4) Control the mode and speed of DEC-PERLE-1 clock by the software program.

For generating 1.5MB bitstream that programs the XC3090 FPGAs to realize the application-specific hardware, the following steps are involved:

1. **Design Partition.** In this step the design is mapped onto 23 FPGA chips according to the logic design and the constraints of the DEC-PERLE-1 board. Some of the FPGA chips may be not used. For example, the CCM design uses only 17 FGPA chips of all 23 chips, because we were not able to find better mapping despite many efforts. The steps 2 and 3 should be carried out separately for each FPGA chip that is used in the design.

2. **Design Entry.** In this step, the design is created for each FPGA used in the design separately. This step produces a **Xilinx netlist file (XNF file)** for the next step. There are three kinds of design entry methods: **(1) Schematic editor** to create the XNF file. **(2) Hardware description language**: the designer can use VHDL (or other hardware description language) to create the design, then the synthesis software is used to synthesize and optimize the design and produce the XNF file. **(3) PerleDC library**. Another possible way is to use a C++ program and the PerleDC library to describe the design. Individual configuration of each FPGAs involved in your design are described by this C++ program. Compiling and running this C++ program generates the XNF file of the design.

   There are many tools that can be used. For instance, there are four sets of tools available at EE of PSU as of this writing: Xilinx Foundation Series, OrCAD Express 7.0, Mentor's Leonardo, Summit. Both Xilinx Foundation Series and OrCAD Express 7.0 support schematic editor and hardware description language.

3. **Design Implementation. Map, place** and **route** your design, and finally **generate the bitstream file** by using Xilinx development tools. Since all FPGAs used on DEC DEC-PERLE-1 board are XC3090 FPGAs, the user needs Xilinx development tools that support XC3090 FPGA.

4. **Design Verification.** At this step, the bitstream generated at the previous steps is downloaded into the DEC-PERLE-1 board and the design is tested. If something goes wrong, you may need to modify your design at design entry step, then regenerate the bitstream file, download it to DEC-PERLE-1 board and test your design again.

VII. CUBE CALCULUS MACHINE

In our design, the Cube Calculus Machine is a co-processor to the host computer and is realized as a virtual processor in DEC-PERLE-1. The simplified block diagram of the CCM is shown in Figure 3; the thick arrow stands for data buses, and the thin arrow stand for control buses. As shown in the figure, the CCM communicates with the host computer through the input and the output FIFO. The Iterative Logic Unit (ILU) is realized using a one-dimensional iterative network of combinational modules and cellular automata. Its partial descriptions are included in [21, 38] and a complete description can be found in [40]. ILU is composed from ITs, each of them processes a single binary variable or two values of a multi-valued variable. Any even number of variables can be processed, and only size of the board as well as bus limitations are the limits (it is the total of 32 values now, which is at most 16 binary variables, 8 quaternary variables, or 4 8-valued variables, or any mixture of even-valued variables). It realizes several operations of MVCC [33].

The ILU can take the input from register file and memory, and can write output to the register file, the memory, and the output FIFO. The ILU executes the cube operation under the control of **Operation Control Unit** (OCU). The **Global Control Unit** (GCU) controls all parts of the CCM and let them work together.

The machine realizes the set of operations [33] from Table 3. The Table shows also their programming information. Each row of Table 3 describes one cube operation. Each operation is specified in terms of: *rel* - the elementary relation type between input values, *and/or* - the global relation type, and the internal **state** of the elementary cellular automaton - *before*, *active* and *after*. The operation name, notation, the output value of *rel* (partial relation) function in every IT, *and_or* (relation type), the output values of *before*, *active* and *after* functions are listed from left to right. Partial relation *rel* is an elementary relation on elementary piece of data (pair of bits). These set theoretical relations such as inclusion, equality, etc. The value of and_or equals to 1 means that the relation type is of AND type; otherwise, the relation type is of OR type. This relation is created by composing elementary relations from ITs and variables.

The machine is microprogrammable both in its OCU control unit part (by use of CCM Assembly Language) and in Data Path, as achieved by ILU operations programmability. For instance, each operation is described by the binary pattern corresponding to it in the respective row of Table 3. By creating other binary patterns in the fields of Table 3, new operations can be programmed to be executed by ILU. As the reader can appreciate, there are very many such combinations, and thus CCM micro-operations. We call this **horizontal data-path microprogramming**. Higher order CCM operations are created by sequencing low-level operations. This is called **vertical control microprogramming** and is executed by OCU (within ILU) and GCU (for operations with memories and I/O). Thus, the user has many ways to (micro) program sequences of elementary instructions. This is done in CCM Assembly language [40].

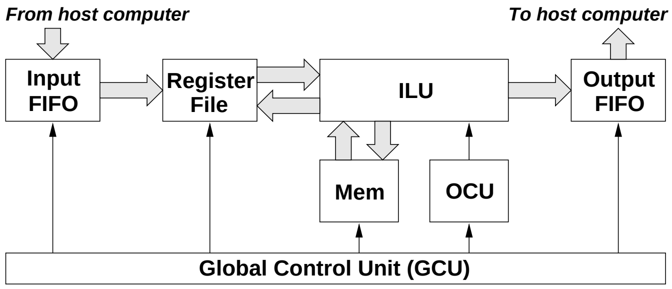


Figure 3: The simplified block diagram of the CCM

Evaluation.

For comparing the performance of the CCM and that of the software approach, a program to execute the disjoint sharp operation on two arrays of cubes was created using C language. Then this program and the CCM are used to solve the following problems: (1) Three variables problem: 1# (all minterm with 3 binary variables). (2) Four variables problem: 1# (all minterm with 4 binary variables). (3) Five variables problem: 1# (all minterm with 5 binary variables). The C program is compiled by GNU C compiler version 2.7.2, and is run on Sun Ultra5 workstation with 64MB real memory. A clock of 1.33 MHz (clock period: 750 ns) is used as the clock of the CCM. The result is shown in Table 4.

It can be seen from Table 4 that our CCM is about 4 times slower than the software approach. But, the clock of the CPU of Sun Ultra5 workstation is 270 MHz, which is 206 times faster than the clock of the CCM. Therefore, we still can say that the design of the CCM is very efficient for cube calculus operations.

It also can be seen from Table 4 that the more variables the input cubes have, the more efficient the CCM is. This is due to the software approach need to iterate through one loop for each variable that is presented in the input cubes.

However, the clock period of 750ns is too slow. From the state diagram of the GCU, it can be found that the delays of *empty carry path* and *counter carry path* only occur in a few states. Thus, if we can just give more time to these states, then we can speedup the clock of the whole CCM. This is very easy to achieve: for example, the state P2 of GCU needs more time for the delay of counter carry path, so add two more states in series between states P2 and P3. These two extra states do nothing but give the CCM two more clock periods to evaluate the signal *prel_res*, which means that the CCM has 3 clock periods to evaluate signal *prel_res* in state P2 after adding two more "delay" states. After making similar modifications to all these kind of states, the CCM can run against a clock of 4 Mhz (clock period of 250 ns). The result is shown in Table 5.

It is very hard to increase the clock frequency again with this mapping because some other paths like *memory path* have delays greater than 150 ns.

From the above comparison result, we can conclude that a design like CCM with a complex control unit and complex data path is not good for the architecture of the DEC-PERLE-1 board. It can be seen from our CCM mapping that since a lot of signals

Operation	**Notation**	**Relation**		**Output Function**		
		rel	*and/or*	*before*	*active*	*after*
crosslink	$A \uplus B$	1110	1	0011	0111	0101
sharp	$A \#_{basic} B$	0010	0	0011	0010	0011
disjoint sharp	$A \#d_{basic} B$	0010	0	0011	0010	0001
consensus	$A *_{basic} B$	1111	1	0001	0111	0001
intersection	$A \cap B$	–	–	0001	–	–
super cube	$A \cup B$	–	–	0111	–	–
prime	$A' B$	0001	0	0011	0111	–
cofactor	$A \|_{basic} B$	1011	1	0001	1111	–

Table 3: The Output Values of Bitwise Functions Used in Cube Operations

Problem	3 variables	4 variables	5 variables
Ultra5	111 usec	268 usec	812 usec
CCM	546×0.75 $= 409$ usec	1285×0.75 $= 963.75$ usec	3405×0.75 $= 2553.75$ usec
speedup	0.27	0.28	0.32

Table 4: Compare CCM (1.33 MHz) with software approach

must go through multiple FPGA chips, this leads to greater signal delays. For instance, if we can connect the memory banks and the registers directly, then the memory path has a delay of only 35 ns. But our current memory path has a delay of 160 ns. Another issue is that XC3090 FPGA is kind of "old" now (6 to 8 years old technology). The latest FPGAs from Xilinx or other vendors have more powerful CLBs and more routing resource, and they are made using deep sub-micron process technology.

If we can map the entire CCM inside one FPGA chip, then we can speedup the CCM from the following aspects:

- If we map entire CCM into one FPGA chip, the signals do not need to go through multiple chips again, which means the routing delay is reduced.
- Since the new FPGA chip has more powerful CLBs and routing resource, we can map the CCM denser. This also reduces the routing delays.
- Since new FPGA chips are made using deep sub-micron technology, the delay of CLB and routing wires are both reduced. For example, the delay of the CLB of XC3090A is 4.5 ns while the delay of CLB of XC4085XL (0.35 micron technology) is only 1.2 ns. This means that it is very easy to achieve 3 times faster mapping.

XC4085XL FPGA, a new FPGA from Xilinx, has a CLB matrix of 56×56 and up to 448 user I/O pins. The CCM should be able to map into one XC4085XL FPGA. With this new chip, it should not be difficult to run the CCM against a clock of 20 MHz (clock period: 50 ns). This means that our CCM will be about 4 times faster than the software approach while the system clock of the CCM is still 5 times slower than that of the workstation.

As stated by the designers of the DEC-PERLE-1 board: *PAM technology is currently best applied to low-level, massively repetitive task such as image or signal processing.* The example applications are a long integer multiplier, RSA cryptography and Fast Hough transform. All these applications have no or very simple control units, and their data paths can be easily pipelined. The CCM has a complex control unit, and a complex data path. It is difficult to pipeline the data path of the CCM. Therefore, the DEC-PERLE-1 board is not the best choice for the CCM.

VIII. LEARNING BY FUNCTIONAL DECOMPOSITION MACHINE

While the previous section presented a complete general-purpose memory-programmable processor for cube calculus, in this section we will show design philosophy: the FPGA implementation of a point algorithm. Phases of the algorithm are executed sequentially, they are then loaded from the host memory, while the intermediate data are stored

Problem	3 variables	4 variables	5 variables
Ultra5	111 usec	268 usec	812 usec
CCM	611×0.25	1486×0.25	4078×0.25
	= 152.75 usec	= 371.5 usec	= 1019.5 usec
speedup	0.72	0.72	0.80

Table 5: Compare CCM (4MHz) with software approach

in DEC-PERLE-1 memories between stages. We will show also how generic combinatorial problems are used in logic learning algorithms. Here the ideas of graph coloring [52, 45] will be used for decomposing functions, and thus in Machine Learning.

The decision table represents a data set, with labeled instances, each relating a set of attribute values to a class (the **output concept**). Decomposition of the table is to decompose the initial table into a hierarchy of decision tables, each of them no more decomposable. Thus, each of these new tables, as well as the entire network are less complex and easier to interpret than the original table. Some regularities not seen in the original table can be found, and the intermediate functions correspond to some features (concepts) of the data set. Ashenhurst/Curtis Decomposition has been adopted to multiple-valued logic, [23, 36]. It applies iteratively the **single decomposition step**, whose goal is to decompose a function $y = F(X)$ into $y = G(A, H(B))$, where X is a set of input attributes $x_1, x_2, ..., x_n$, and y is the class. F, G, and H are functions represented as **decision tables**, i.e. possibly incomplete sets of attribute-value vectors with assigned classes. A and B are subsets of input attributes, called **free** and **bound** set, respectively, such that $A \cup B = X$. Functions G and H are developed in the decomposition process and not predefined in any way. New **concept** $c_1 = H(B)$ has been found. The goal is to find the decomposition of the smallest complexity (DFC [1]).

Let us illustrate the decomposition on a simple example. Consider the decision table in Table 6. It relates the input attributes x_1, x_2, and x_3 to the class y, such that $y = F(x_1, x_2, x_3)$.

There are three possible non-trivial partitions of attributes that yield three different decompositions $y = G_1(x_1, H_1(x_2, x_3))$, $y = G_2(x_2, H_2(x_1, x_3))$, $y = G_3(x_3, H_3(x_1, x_2))$. The first two are given in Figure 4e and Figure 4f, respectively.
The comparison shows that:

(1) decision tables in the decomposition $y = G_1(x_1, H_1(x_2, x_3))$ are smaller than those for $y = G_2(x_2, H_2(x_1, x_3))$,

(2) the new concept $c_i = H_1(x_2, x_3)$ uses only three values, whereas that for $H_2(x_1, x_3)$ uses four,

(3) we found it hard to interpret decision tables G_2 and H_2, whereas by inspecting H_1 and G_1 it can be easy to see that $c_1 = MIN(x_2, x_3)$ and $y = MAX(x_1, c_1)$. This can be even more evident with the assignment of values 0,1, and 2 of a multi-valued variable X_i: ${X_i}^0 = lo$, ${X_i}^1 = me$, ${X_i}^2 = hi$.

The following problems must be solved by an efficient decomposition algorithm:

(1) how to select sets A and B?

(2) how to evaluate the quality of decompositions?

Unfortunately, all known methods require nearly exhaustive searches that involve huge repetitions of basic operations.

A. Simple Decomposition Algorithm for Functions

The decomposition algorithm constructs a partition matrix with attributes of bound set in columns and of free set in rows. Each column in the partition matrix denotes the behavior of F when the attributes in the bound set are constant. Some columns can then be represented with the same value of c, and the number of different columns is equal to the minimal number of values for c to be used for decomposition. In this way, every column is then assigned a value of c, and G and H are straightforwardly derived from such annotated partition matrix. For each of three partitions for our sample decision table F, the partition tables with the corresponding values of c are given in Figure 4b,c, and d respectively.

The assignment of values of c is trivial in case of a completely specified function, which is, when decision table instances completely cover the attribute space. Otherwise, when the function is incompletely specified, the relation of compatibility of columns is no longer transitive, and the **graph coloring approach** is used. Column functions are calculated by a cofactor operation on the original function f. The *cofactor* f_{PROD} of function f with respect to the literals from $PROD$ is this function with all literals from $PROD$ substituted to maximum constant value (constant value 1 in case of binary logic). All functions are represented by arrays of cubes.

x_1	x_2	x_3	y
lo	lo	lo	lo
lo	lo	hi	lo
lo	me	lo	lo
lo	me	hi	me
lo	hi	lo	lo
lo	hi	hi	hi
lo	hi	hi	hi
me	lo	lo	me
me	lo	hi	me
me	me	lo	me
me	me	hi	me
me	hi	lo	me
me	hi	hi	hi
hi	lo	lo	hi
hi	lo	hi	hi
hi	me	lo	hi
hi	me	hi	hi
hi	hi	lo	hi
hi	hi	hi	hi

Table 6: An example decision table $y = F(x_1, x_2, x_3)$

For a completely specified binary function, two columns n_1 and n_2 are compatible if the Boolean functions corresponding to them are a Boolean Tautology:

n_1 *compatible* n_2 iff $f_{Prod_1} = f_{Prod_2}$

which is equivalent to:

n_1 *compatible* n_2 iff $(ON(n_1) \# ON(n_2) = \emptyset)$ and $(ON(n_2) \# ON(n_1) = \emptyset)$

where # denotes the sharp (difference) operation on arrays of cubes, and ON is the set of true cubes in SOP form.

For an incompletely specified binary function, two nodes of the graph for coloring are incompatible if the corresponding columns are not compatible (cannot be merged into one column):

n_1 *incompatible* n_2 iff $(ON(n_1) \cap OFF(n_2) \neq \emptyset)$ or $(ON(n_2) \cap OFF(n_1) \neq \emptyset)$

As we see, only two basic operations, **cofactor** and **sharp** are used for complete functions. Similarly, only cofactor and **intersection** are used for incomplete functions. But, in both cases, these operations are repeated many times on cubes from the cube arrays. Similarly, the basic (mv) logic operators can be used for checking compatibility of columns of multiple-valued functions while creating the graph for coloring. After creation, the graph is colored in such a way that every two nodes linked by an edge obtain different colors, and the minimum number of colors is used. Graph coloring can be reduced to sequences of basic logic operators as well.

Concluding, in addition to cofactoring, the partial combinatorial problems that are solved by our hardware decomposition processor DP are the following: set covering, graph coloring, and maximum clique. They are all NP-hard, and they all have many other applications in ML.

Function Decomposition is an NP complete problem. Moreover, every stage of Function Decomposition, except the stages of G and H function blocks creation, are NP complete problems. Hence, even if more efficient algorithms can be found for exact calculations, they will be either too slow or of inferior quality. One approach to find solutions to NP-hard problem is not to attempt at the exact solution, but be satisfied with one which is near exact but obtainable in a reasonable time. This type of algorithm is based on heuristics, or rules which can be applied which are likely to improve the solution. Such algorithms, when implemented in hardware, can bring orders of magnitude speed-up [24]. We have chosen algorithms that are simple, easy, fast and can be relatively easy implemented in hardware. In addition, in paper [4] we showed that decomposition of fuzzy functions and relations can be reduced to decomposition of multi-valued functions and relations. So, assuming that the fast stages of converting fuzzy relation to MV relation and next converting of MV relations(functions) back to fuzzy logic for each decomposed block are executed in software, our hardware machine will still perform the most complex part of

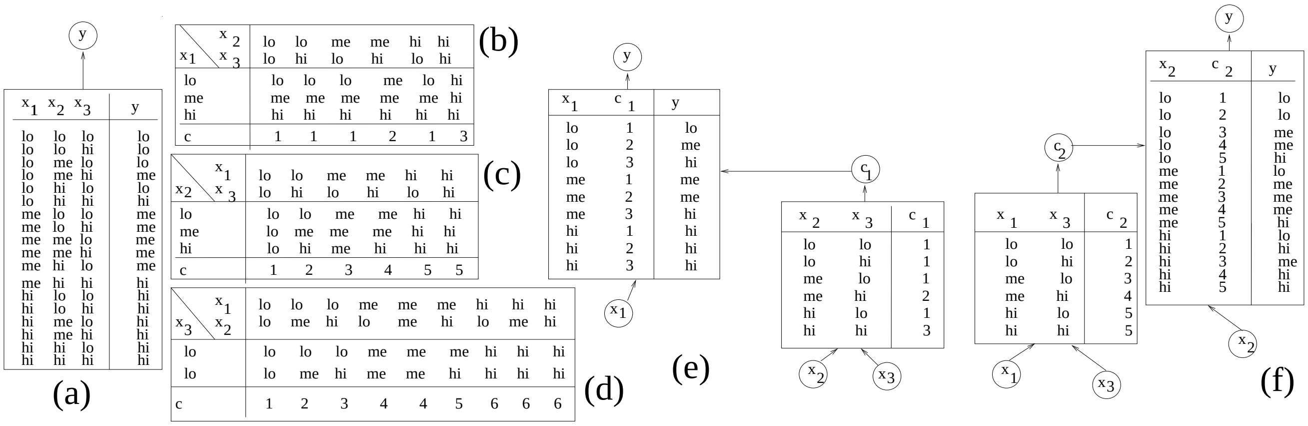


Figure 4: Two one-step decompositions of the decision table from Table 2

the synthesis proces.

In addition to two virtual processors from this paper, we developed and simulated the Rough Set Machine (RSM) [30] and the Satisfiability Machine (SM) [31]. RSM is a SIMD processor that realizes the basic operations of Rough Sets theory of Zdzislaw Pawlak. SM is a systolic processor to solve satisfiability and related problems that occur in many combinatorial optimization problems.

IX. CONCLUSIONS.

We presented principles of Learning Hardware as a competing approach to Evolvable Hardware, and also as its generalization. The concept of Data Mining machines has been outlined and the Universal Logic Machine with several virtual processors was briefly sketched, as just one possible realization of such machines. Although DEC-PERLE-1 is a good medium to prototype such machines, its XC3090A chip is now obsolete. This can be much improved by using XC4085XL FPGA and redesigning the board. Massively parallel architectures such as CBM based on new Xilinx series 6000 chips will allow even higher speedups. We hope that such machines will be used to implement improved models of ULM.

References

[1] Y. Abu-Mostafa (ed.), " Complexity in Information Theory," *Springer Verlag,* New York, 1988, p. 184.

[2] R.L. Ashenhurst, "The Decomposition of Switching Functions", *Proc. Int. Symp. of Th. of Switching,* 1957.

[3] A. Buller, "Artificial Brain. Phantasies no more," *Proszynski i Ska,* Warsaw, 1998, (in Polish).

[4] P. Burkey, M. Perkowski, and A. Wielgus, "Ashenhurst/Curtis Decomposition of Fuzzy Functions and Relations," submitted to *Multiple-Valued Logic. An International Journal,* Gordon and Breach Science Publishers, 1999.

[5] R.E. Bryant, "Graph-based algorithms for boolean function manipulation, *IEEE Transactions on Computers*, C-35, No. 8, pp. 667-691, 1986.

[6] E.F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Comm. ACM,* 13, pp. 377-387.

[7] H.A. Curtis, "A New Approach to the Design of Switching Circuits," *Princeton,* N.J., Van Nostrand, 1962.

[8] D.L. Dietmeyer, "Logic Design of Digital Systems," *Allyn and Bacon, Boston, MA,* 1971.

[9] K. Dill, and M. Perkowski, "Minimization of Generalized Reed-Muller Forms with Genetic Operators," *Proc. Genetic Programming '97 Conf.,* July 1997, Stanford Univ., CA.

[10] K. Dill, J. Herzog, and M. Perkowski, "Genetic Programming and its Application to the Synthesis of Digital Logic," *Proc. PACRIM '97*, Canada, August 20-22, 1997.

[11] K. Dill, and M. Perkowski, "Evolutionary Minimization of Generalized Reed-Muller Forms," *Proc. ICCIMA'98 Conference*, pp. 727-733, February 1998, Australia, published by World Scientific.

[12] B. Falkowski, I. Schaefer, M. Perkowski, "Effective Computer Methods for the Calculation of Rademacher-Walsh Spectrum for Completely and Incompletely Specified Boolean Functions," *IEEE Trans. on Computer-Aided Design,* pp. 1207 - 1226, October 1992.

[13] C. Files, M. Perkowski, "An Error Reducing Approach to Machine Learning Using Multi-Valued Functional Decomposition," *Proc. ISMVL'98*, pp. 167 - 172, May 1998.

[14] C. Files, M. Perkowski, "Multi-Valued Functional Decomposition as a Machine Learning Method," *Proc. ISMVL'98*, pp. 173 - 178, May 1998.

[15] J.M. Francioni, and A. Kandel, "Decomposable Fuzzy-valued Switching Functions," *Fuzzy Sets and Systems,* Vol. 9, No. 1, pp. 41-68, 1983.

[16] H. DeGaris, "Evolvable Hardware: Genetic Programming of a Darwin Machine," In "Artificial Nets and Genetic Algorithms," R.F. Albrecht, C.R. Reeves and N.C. Steele (eds), *Springer Verlag*, pp. 441-449, 1993.

[17] H. DeGaris, "Evolvable Hardware: Principles and Practice," *CACM Journal*, August 1997.

[18] *http : //www.hip.atr.co.jp/ ~ degaris*

[19] S. Grygiel, and M. Perkowski, "New Compact Representation of Multiple-Valued Functions, Relations, and Non-deterministic State Machines," *Proc. ICCD'98*, October 1998.

[20] T. Higuchi, M. Iwata, and W. Liu (eds), "Evolvable Systems: From Biology to Hardware," *Lecture Notes in Computer Science,* No. 1259, *Proc. First Intern. Conf. ICES'96,* Tsukuba, Japan, October 1996, Springer Verlag, 1997.

[21] L. Jozwiak, M.A. Perkowski, D. Foote, "Massively Parallel Structures of Specialized Reconfigurable Cellular Processors for Fast Symbolic Computations," *Proc. MPCS'98 - The Third International Conference on Massively Parallel Computing Systems*, Colorado Springs, Colorado - USA, April 6-9, 1998.

[22] T. Luba, J. Rybnik, "Algorithmic Approach to Discernibility Function with Respect to Attributes and Object Reduction," *Int. Workshop on Rough Sets,* Poznan 1992.

[23] T. Luba, "Decomposition of multiple-valued functions", *Proc. 25th ISMVL,* 1995, pp. 256-261.

[24] R. Malvi, M. Perkowski, and L. Jozwiak, "Exact Graph Coloring for Functional Decomposition: Do we Need it?," pp. 1-10, *Proceedings of 3rd International Workshop on Boolean Problems,* Freiberg University of Mining and Technology, Institute of Computer Science, September 17-18, 1998.

[25] C. Mead, "Analog VLSI And Neural Systems," *Addison Wesley Pub.*, April 1989.

[26] R.S. Michalski and J.B. Larson, "Inductive inference of vl decision rules," in *Workshop in Pattern-Directed Inference Systems,* Hawaii, May 1977.

[27] R.S. Michalski, I. Bratko, and M. Kubat, "Machine Learning and Data Mining: Methods and Applications," *Wiley and Sons*, 1998.

[28] D. Michie, "Machine Learning in the next five years," *Proc. EWSL'88, 3rd European Working Session on Learning,* Glasgow, *Pitman*, London, 1988.

[29] L. Nguyen, M. Perkowski, N. Goldstein, "PALMINI - Fast Boolean Minimizer for Personal Computers," *Proc. of the IEEE/ACM 24th Design Automation Conference,* pp. 615 - 621, Miami, Florida, June 28 - July 1, 1987.

[30] Z. Pawlak, "Rough Sets. Theoretical Aspects of Reasoning about Data," *Kluwer Academic Publishers,* 1991.

[31] M. Perkowski, "Systolic Architecture for the Logic Design Machine," *Proc. of the IEEE and ACM International Conference on Computer Aided Design - ICCAD'85,* pp. 133 - 135, Santa Clara, 19 - 21 November 1985.

[32] M. Perkowski, S. Wang, W.K. Spiller, A. Legate, E. Pierzchala, "Ovulo-Computer: Application of Image Processing and Recognition to Mucus Ferning Patterns," *Proc. of the Third IEEE Symposium on Computer-Based Medical Systems,* pp. 52 - 59, Chapel Hill, North Carolina, June 3-6, 1990.

[33] M.A. Perkowski, "A Universal Logic Machine," invited address, *Proc. of the 22nd IEEE International Symposium on Multiple Valued Logic, ISMVL'92,* pp. 262 - 271, Sendai, Japan, May 27-29, 1992.

[34] M. A. Perkowski, M. Chrzanowska-Jeske, "Multiple-Valued-Input TANT Networks," *Proc. ISMVL'94*, pp. 334-341, Boston, MA, May 25-27, 1994.

[35] M. A. Perkowski, T. Ross, D. Gadd, J.A. Goldman, and N. Song, "Application of ESOP Minimization in Machine Learning and Knowledge Discovery," *Proc. of the Second Workshop on Applications of Reed-Muller Expansion in Circuit Design,* Chiba City, Japan, 27-29 August 1995, pp. 102-109.

[36] M. Perkowski, M. Marek-Sadowska, L. Jozwiak, T. Luba, S. Grygiel, M. Nowicka, R. Malvi, Z. Wang, and J. S. Zhang, "Decomposition of Multiple-Valued Relations," *Proc. ISMVL'97,* Halifax, Nova Scotia, Canada, May 1997, pp. 13 - 18.

[37] M. Perkowski, P. Lech, Y. Khateeb, R. Yazdi, and K. Regupathy, "Software-Hardware Codesign Approach to Generalized Zakrevskij Staircase Method for Exact Solutions of Arbitrary Canonical and Non-Canonical Expressions in Galois Logic," *Booklet of 6th Intern. Workshop on Post-Binary ULSI Systems,* Nova Scotia, Canada, May 27, 1997, pp. 41 - 44.

[38] M. A. Perkowski, L. Jozwiak, and D. Foote, "Architecture of a Programmable FPGA Coprocessor for Constructive Induction Approach to Machine Learning and other Discrete Optimization Problems", in Reiner W. Hartenstein and Victor K. Prasanna (ed) *"Reconfigurable Architectures. High Performance by Configware,"* IT Press Verlag, Bruchsal, Germany, 1997, pp. 33 - 40.

[39] M. Perkowski, L. Jozwiak, and S. Mohamed, "New Approach to Learning Noisy Boolean Functions," *Proc. ICCIMA'98 Conference,* February 1998, Australia, published by World Scientific, pp. 693 - 706. Australia, published by World Scientific.

[40] M. Perkowski, "Do It Yourself Reconfigurable Supercomputer that Learns," *book preprint*, Portland, Oregon, 1999.

[41] PSU POLO Directory with DM/ML Benchmarks, software and papers: *http : //www.ee.pdx.edu/polo/*

[42] E. Pierzchala and M. Perkowski, "A High-Frequency Field-Programmable Analog Array (FPAA), Part 1: Design, Part 2: Applications," *Field-Programmable Analog Arrays,* (E. Pierzchala, ed.), *Kluwer Academic Publishers,* 1998.

[43] L.O. Chua and T. Roska, "The CNN paradigm," *IEEE Trans. on Circuits and Systems-I,* Vol. 40, No. 3, pp. 148-156, March 1993.

[44] T. D. Ross, M.J. Noviskey, T.N. Taylor, and D.A. Gadd, "Pattern Theory: An Engineering Paradigm for Algorithm Design," *Final Technical Report WL-TR-91-1060,* Wright Laboratories, USAF, WL/AART/WPAFB, OH 45433-6543, August 1991.

[45] P. Sapiecha, M. A. Perkowski, and T. Luba, "Decomposition of Information Systems Based on Graph Coloring Heuristics," *Symposium on Modelling, Analysis and Simulation,* CESA'96 IMACS Multiconference, Lille, France, July 9-12, 1996.

[46] T. Sasao (editor), "Representation of Boolean Functions," *Kluwer Academic Publishers,* 1996.

[47] K.B. Stanton, P.R. Sherman, M.L. Rohwedder, Ch.P. Fleskes, D. Gray, D.T. Minh, C. Espinosa, D. Mayi, M. Ishaque, M.A. Perkowski, "PSUBOT - A Voice-Controlled Wheelchair for the Handicapped," *Proc. of the 33rd Midwest Symp. on Circuits and Systems,* pp. 669 - 672, Alberta, Canada, August 1990.

[48] Y.H. Su and P.T. Cheung, "Computer minimization of multiple-valued switching functions," *IEEE Transactions on Computers,* Vol. C-21, pp. 995-1003, 1972.

[49] N. Song, M. Perkowski, "Minimization of Exclusive Sum of Products Expressions for Multi-Output Multiple-Valued Input, Incompletely Specified Functions," *IEEE Transactions on Computer Aided Design,* Vol. 15, No. 4, April 1996, pp. 385-395.

[50] U.C. Irvine, "Repository of Machine Learning Databases and Domain Theories," $ftp://ftp.ics.uci.edu/pub/machine-learning-databases/$

[51] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and Ph. Boucard, "Programmable Active Memories: Reconfigurable Systems Come of Age," *IEEE Trans. on VLSI Systems,* Vol. 4, No. 1., pp. 56-69, March 1996

[52] W. Wan, and M. Perkowski, "A New Approach to the Decomposition of Incompletely Specified Multi-Output Function Based on Graph Coloring and Local Transformations and Its Application to FPGA Mapping," *Proc. Euro-DAC*, pp. 230 - 235, 1992.