LEARNING IN HARDWARE: ARCHITECTURE AND IMPLEMENTATION OF AN FPGA-BASED ROUGH SET MACHINE

Torrey Lewis, Marek Perkowski, and Lech Jozwiak+

Portland State University, Dept. of Electr. Engn., Portland, Oregon 97207,
Tel: 503-725-5411, Fax: 503-725-4882, *mperkows@ee.pdx.edu*
+ Faculty of Electrical Engineering, Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Abstract— The "Learning Hardware" approach proposed here 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). We advocate the approaches based on a "strong AI criterion"; for instance, the computational networks can be built based on Sum-of-Products logic minimization, functional logic decomposition, or Decision Tree construction. Here we propose the constructive induction approach to Learning Hardware based on Rough Sets Theory (RST). This approach allows the use of logical analysis to develop efficient hardware-realizable algorithms, and is contrasted with the popular Evolvable Hardware (EHW) approach in which learning/evolution is based on the genetic algorithm only. The RST algorithms have a natural high parallelism and high possible speed-ups. Using a fast prototyping tool, the DEC-PERLE-1 board based on an array of Xilinx FPGAs, we are developing a virtual SIMD processor that accelerates the learning (design) of optimized multi-valued logic nets.

1. INTRODUCTION

Recently, the concept of **Evolvable Hardware** (EHW) has been invented [7, 8, 9, 11] which is the **realization of a genetic algorithm (GA) in reconfigurable hardware.** In contrast, our approach, the **Universal Logic Machine** (ULM) [12, 20, 21, 24], we propose to build a learning machine based on the **logic principles**, especially on Constructive Induction [15, 16] and Rough Set Theory (RST) [17, 19]. While the Genetic Algorithm of the EHW is a very simple and practically blind mechanism of nature, and as such it can be easily realized in hardware, the **logic algorithms** that use previous human knowledge are mathematically sophisticated, very effective and efficient, but their software realizations use complex data structures and controls that make it very difficult to realize them in hardware. Moreover, their hardware realization may suffer from the consequences of Amdahl's Law. Thus, some nontrivial and often very complex software-hardware design trade-offs must be resolved to effectively and efficiently realize in hardware the logic-based learning methods.

The process of solving problems can be sub-divided into two phases: **the phase of learning**, which is, constructing and tuning a (knowledge) network, and **the phase of using the knowledge**, that is, evaluating 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); the second stage to running this computer on data to perform pre-programmed calculations. However, you cannot redesign computer hardware **automatically** when it cannot solve the problem correctly, while you can do this with the Evolvable or Learning Hardware. Many new approaches can be created and investigated by combining some basic learning models and methods. For instance, the Artificial Neural Nets (ANN) used in the Brain Builder's [9] approach can be directly compiled to binary hardware without using the intermediate medium of cellular automata used there, or an algorithm different than genetic can be used to construct the ANN. It is thus in the network model selection and network construction methods where the different philosophies of designing the Learning Hardware and Evolvable Hardware essentially differ. The ULM model investigates various such combined approaches to learning realized in hardware. In this paper, we will propose a new methodology to the design of a learning machine, based on the Field Programmable Gate Arrays (FPGA) technology and the logic methods of Rough Set Theory. In particular, this paper presents preliminary work on the design and implementation of a SIMD Computer to implement Rough Set Theory operations, and for illustration it examines the RST as it applies to logic minimization.

RST is a mathematical model of data analysis proposed by Zdzislaw Pawlak [19]. The advantage of RST is its high expressive power and inherent parallelism. It also has strong relations to Codd's Data Base model [3]. RST can be used to minimize truth tables in much the same way as Karnaugh Maps. However, logic minimization is only one of many possible applications of Rough Set Theory. Another application of RST is Data Mining [14]. It is important to note that some subsets of RST are isomorphic with some subsets of logic synthesis and decomposition theories, and thus their mutual relationships can be investigated, leading to synergies of concepts. For instance, powerful logic concepts of RST can be linked with efficient algorithms and data structures developed in logic synthesis for EDA [4, 12, 20, 21, 22, 23, 24, 25, 27]. A Parallel Rough Set Computer, PRSComp, has been proposed by Muraszkiewicz and Rybinski as a possible hardware implementation of the Rough Set Theory [17]. Here we extend these concepts and implement them in a practical reconfigurable FPGA architecture, thus using all advantages of this technology.

The remainder of this paper is structured as follows. Section 2 introduces the basic concepts of the Learning Hardware approach and contrasts it with the Evolvable Hardware research. Section 3 presents basic RST operations. For simplification, only a subset of operations is given, and linked to classical logic minimization methods for better explanation. Section 4 describes a SIMD Parallel RST computer realized on DEC-PERLE-1 FPGA board; one implementation of our general "Learning Hardware" methodology. This computer realizes and extends the ideas from [17]. The algorithms realized on this computer are illustrated in section 5. Section 6 concludes the paper.

2. "EVOLVING IN HARDWARE" OR "LEARNING IN HARDWARE"?

The learning system satisfies a **weak criterium** when it uses sample data to generate an updated basis for improved performance on subsequent data. A **strong criterion** is satisfied if the system can also communicate its learned concepts in a symbolic form [16]. For instance, a medical doctor who uses the aid of a knowledge-based system cannot rely on a "black box"-type of decision from the system. He has to understand the explanation of the system to undertake his decision, for which only he will be responsible. Let us observe that ANNs and similar EHW approaches satisfy only the weak criterium. Our approach satisfies the strong criterium. In our opinion, the results of the learning process, and even the process itself, should be understood by humans. 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]. Occam Razor should be used whenever possible because only applying this principle can lead to **meaningful discoveries**.

In the past we developed several logic [22], GA-based [4, 5, 6] and mixed [13] approaches to combinational learning algorithms (such logic is **highly unspecified**). Based on these investigations, we developed the opinion that for our class of problems the logic approaches combined with smart heuristic strategies and good data representations, are superior to other approaches with respect to smaller net complexity and learning error. Especially poor results were obtained using pure genetic algorithms [4, 5, 6]. Relatively good results were achieved when the logic and GA approaches were combined together [13]. Maybe "pure GA" performs well in some other application areas. However, both in our experience and in literature we were not able to find any single case designing a binary or multi-valued network of any kind in which a GA-based algorithm would be superior to a good human-designed algorithm.

Thus, the following general observations related to the **practical hardware realization of Learning Hardware** can be made:

1. Most of the current approaches to learning and evolutionary hardware use binary FPGAs, because there are simply no other large-scale reconfigurable (programmable) hardware technologies commercially available. Other potential realization technologies are either too primitive and do not allow for large networks or are in too early development stages. A practical approach will be then to compare various learning paradigms assuming the binary FPGA implementation model.
2. In our opinion, the learning process should be performed on the level of logic gates rather than that of switching transistor sequences responsible for routing connection paths, or that of arithmetic operations (as in ANNs or Fuzzy Logic functions), because in binary FPGAs everything is realized on the level of **binary logic gates**.
3. Once we decide to realize the network using logic gates in FPGAs, we should **re-use all powerful Electronic Design Automation (EDA) tools that engineers have already developed** in many years in the area of digital design automation; especially the tools for: reconfigurable computers, state machines, logic synthesis, technology mapping, placement and routing, partitioning, timing analysis, etc. The EDA tools should be re-used in their entirety, rather than duplicated by naive low-level evolutionary algorithms. To enhance efficiency, some of them, such as logic minimizers, should be realized in hardware.

Concluding, we believe that the "purist strategies" to evolutionary hardware, DeGaris and Brains Builders [8], will not 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 mathematical algorithms and problem-solving strategies rather than rely on only two generic methods of Evolvable Hardware: ANNs and GA. Learning/evolution should still remain as the main principle, but it should be restricted to high abstract levels, and the variants evaluation should be also performed there. Learning should be performed **before** mapping to low-level field-programmable resources because at such low level the chromosomes are **extremely long** and the operation of GA becomes **totally inefficient**.

Our Learning Hardware approach is thus directed towards the state-of-the-art FPGA (and ASIC) technologies. It 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** (such as PRSComp from sections 4,5), create the logic network description (optimized rule sets), using logic/mathematical algorithms.
2. The (quasi)optimally constructed network is mapped to standard FPGAs and realized using partitioning, placement, routing and other **EDA tools** from Xilinx and EDA software companies (in the context of learning, mapping is much more rarely executed than optimization, and therefore software can be used).
3. The knowledge of the machine is stored in memory patterns representing the logic nets. While solving new problems 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 similar to the CBM approach [9].
4. Since a network solves new problems, new data sets and training decisions are accumulated and the network is **repetitively 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** of EHW with the ULM model of **learning at a high level** and next compiling to the low level using standard EDA tools for FPGA-based synthesis. Moreover, the same physical FPGA **resources are multiplexed** to realize 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. Thus the **growing virtual hardware** has the "learning hardware" as its base of update and growth.

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 the Internet. Other variants of such machines will acquire data from industrial, agricultural, military, or other application areas in real-time, using sensors, microphones and TV cameras use the and pre-processing techniques of Image Processing and Digital Signal Processing. In contrast to similar projects, our ultimate goal is not to build the Artificial Brain [8], 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 learning and execution phases** of the application software programs that are now being used in Machine Learning, Knowledge Discovery from Data Bases, Data Mining, and robotics.

Presently, we model our algorithms in software or we implement them for a prototype reconfigurable platform from DEC, the DEC-PERLE-1 board [18, 28]. It is essentially an array of FPGAs that can be programmed by a host computer to implement any desired function (machine). Typically there are two possibilities on how to implement a desired function. The function can be committed to application specific hardware, or it can be realized in software. Both approaches have advantages and drawbacks. Hardware is fast, but is dedicated and can only perform the specific function. Software has the flexibility to perform many different functions, through multiple programs, but is generally much slower than hardware. The appropriate combination of an adequately programmed DEC-PERLE-1 board and the software run on a general purpose computer attempts to capture the advantages of both worlds and provide the speed of dedicated hardware, yet allowing the flexibility of application software. The DEC-PERLE-1 board has been used to implement several different functions and compares favorably with other similar boards [26, 28]. Below we illustrate the learning phase of ULM using the virtual RST computer.

3. Basic Operations of Rough Sets

Following is a logic minimization example to explain how the Rough Set Theory can be applied to logic minimization in Data Mining. The example is taken from [19]. The same data will be used later in demonstrating Muraszkiewicz and Rybinski's PRSComp algorithms [17] that we emulate on our machine. This will give us the opportunity to compare the RST method with the PRSComp algorithms.

Table 1 shows the definition of a desired net (which can represent a circuit, concept, algorithm, set of rules, etc.) in the form of an incompletely specified truth table (also known as a decision table). Variables a, b, c, d, and e are input variables (attributes) and f is the output variable (concept, decision). The truth table is incompletely specified, because of the total number of possible 2^5 input minterms only 15 are present in the table. The goal is to minimize (Occam's Razor) the function described by Table 1 using the RST method. (Obviously, this problem can also be solved using Karnaugh maps or Espresso, but our interest here lies in finding a **general** software/hardware realizable algorithm, especially for strongly unspecified multi-valued (MV) functions). The procedure is to first eliminate redundant input variables (so-called **vacuous variables**) from the table and then eliminate **unnecessary values of variables** from each decision rule. The result will be a minimum solution to the problem. The formula obtained will assign specific values to input combinations that were don't cares in the training data from Table 1. This means **generalization** and **learning**. This procedure will be explained below through more examples.

Definition 1 *A* **decision rule** *is the relation of the set of input variables to the set of output variables.*

In our example a decision rule corresponds to a row in the truth table.

The first step of the minimization procedure is to find the vacuous variables and remove them. This is performed by removing each input variable one at a time and then determining whether the resulting truth table is still consistent.

Definition 2 Consistency of a table. *For every combination of input variables presented in the table there is a unique value for the output.*

In this example, the initial truth table, Table 1 is known to be **consistent**, because for every input combination of a, b, c, d, and e, there is a unique value for the output variable f. If there happened to be a row 16, that were identical to row 1 except that the output variable f were defined to be 0, the table would then be **inconsistent**.

Another important concept of Rough Sets Theory is whether or not an input variable is **dispensable**.

Definition 3 *An input variable is* **dispensable** *if it is possible to remove the variable and result in a consistent table. A dispensable variable is* **redundant** *in the final table and is also referred to as a* **vacuous variable***.*

Since a dispensable input variable does not effect the value of the output, it can be removed. If a variable is not dispensable then it is indispensable.

Definition 4 *An input variable is* **indispensable** *if removing the variable results in an inconsistent table.*

Hence, f-indispensable variables effect the output value and therefore must remain in the table. By removing one variable (i.e. column) at a time from the table we can find which variables can be removed and still result in a consistent table. Tables 2 through 4 show the results of the removal of columns a, b, and c.

From Table 2, we can see that input variable a is f-indispensable, because removing it results in an inconsistent table. The table is inconsistent because decision rules 6 and 12 yield different output values for identical input values. Both decision rule 6 and 12 have input values of 1, 1, 1 and 0 for b, c, d and e respectively. The determining factor is that the output values differ: 1 for decision rule 6 and 0 for decision rule 12. Similarly, with decision rules 9 and 11. From Table 3, we can see that input variable b is f-indispensable, because rows 2 and 10 are inconsistent. Table 4 shows that input variable c is f-dispensable because removing it from the table results in a consistent table. This means that input variable c does not need to be present in the final solution. By similarly creating a table (table not shown) for removal of d we find that input variable d is f-indispensable because rows 3 and 12, and rows 8 and 15 disagree with each other. Also, we find that input variable e is f-indispensable because decision rules 1 and 11, 3 and 13, and 6 and 14 are inconsistent. We do not test variable f, because it is the output variable and it cannot be removed and result in the desired functionality. From the above discussion we see that input variable c is the only f-dispensable variable in the original table. Table 5 is Table 4 after combining identical decision rules. Further analysis will be performed on Table 5.

When used for referencing decision rules from Table 5, the first value listed in the U column will be used in the case of (2,3), (4,7) and (5,6).

Now that redundant input variables have been removed we can move on to the next step: removing redundant values of input variables. This is known as finding the **core values**. To do this, one input variable is dropped at a time for each decision rule and then it is determined if the intersection of values of the remaining input variables is included in the set of the input variable combinations having the same value of output variables. This will be explained by continuing the example. This step is done for each decision rule independently.

Definition 5 Core Values *are those values that must be kept, rather than dropped, because dropping them would result in realizing a different function.*

This step begins with decision rule 1. First, using rough set notation, we must prove that the combination of the

Table 1: Initial Truth Table

U	a	b	c	d	e	f
1	0	0	0	1	1	1
2	0	1	0	0	0	1
3	0	1	1	0	0	1
4	1	1	0	0	0	1
5	1	1	0	1	0	1
6	1	1	1	1	0	1
7	1	1	1	0	0	1
8	1	0	0	1	1	1
9	1	0	0	1	0	1
10	0	0	0	0	0	0
11	0	0	0	1	0	0
12	0	1	1	1	0	0
13	0	1	1	0	1	0
14	1	1	1	1	1	0
15	1	0	0	0	1	0

Table 2: Removal of a

U	b	c	d	e	f
1	0	0	1	1	1
2	1	0	0	0	1
3	1	1	0	0	1
4	1	0	0	0	1
5	1	0	1	0	1
6	1	1	1	0	1
7	1	1	0	0	1
8	0	0	1	1	1
9	0	0	1	0	1
10	0	0	0	0	0
11	0	0	1	0	0
12	1	1	1	0	0
13	1	1	0	1	0
14	1	1	1	1	0
15	0	0	0	1	0

Table 3: Removal of b

U	a	c	d	e	f
1	0	0	1	1	1
2	0	0	0	0	1
3	0	1	0	0	1
4	1	0	0	0	1
5	1	0	1	0	1
6	1	1	1	0	1
7	1	1	0	0	1
8	1	0	1	1	1
9	1	0	1	0	1
10	0	0	0	0	0
11	0	0	1	0	0
12	0	1	1	0	0
13	0	1	0	1	0
14	1	1	1	1	0
15	1	0	0	1	0

Table 4: Removal of c

U	a	b	d	e	f
1	0	0	1	1	1
2	0	1	0	0	1
3	0	1	0	0	1
4	1	1	0	0	1
5	1	1	1	0	1
6	1	1	1	0	1
7	1	1	0	0	1
8	1	0	1	1	1
9	1	0	1	0	1
10	0	0	0	0	0
11	0	0	1	0	0
12	0	1	1	0	0
13	0	1	0	1	0
14	1	1	1	1	0
15	1	0	0	1	0

Table 5: Combining rules of Table 4

U	a	b	d	e	f
1	0	0	1	1	1
2,3	0	1	0	0	1
4,7	1	1	0	0	1
5,6	1	1	1	0	1
8	1	0	1	1	1
9	1	0	1	0	1
10	0	0	0	0	0
11	0	0	1	0	0
12	0	1	1	0	0
13	0	1	0	1	0
14	1	1	1	1	0
15	1	0	0	1	0

input values for decision rule 1 is unique (i.e. no other decision rules have the same combination of input values).

$F = [1]_{\{a,b,d,e\}} = [1]_a \cap [1]_b \cap [1]_d \cap [1]_e = \{1,2,10,11,12,13\} \cap \{1,8,9,10,11,15\} \cap \{1,5,8,9,11,12,14\} \cap \{1,8,13,14,15\} = \{1\}$

In the rough set notation used above, the number appearing within the square brackets is the decision rule number. The set $[1]_a$ is the set of all decision rules that have the same value for input variable a as decision rule 1. Decision rule 1 has a value of 0 for input variable a. From Table 5 we can see that decision rules 2, 10, 11, 12 and 13 also have a value of 0 for input variable a, thus they all belong to $[1]_a$. Similarly for the sets $[1]_b$, $[1]_d$ and $[1]_e$.

Next to determine the set, $[1]_f$, that contains all decision rules that have the same value for the output variable f as decision rule 1: $[1]_f = \{1,2,4,5,8,9\}$.

To determine which input values are essential to decision rule 1 we must find the core values. To perform this, each input variable is dropped one at a time and then the resulting intersection is compared to $[1]_f$. If the resulting intersection is a subset of $[1]_f$ then the variable can be dropped, meaning it is not a core value, i.e. it is not essential to the decision rule. The following equations show the dropping of one variable at a time and obtaining the intersection.

$\bigcap(F - [1]_a) = [1]_b \cap [1]_d \cap [1]_e = \{1,8,9,10,11,15\} \cap \{1,5,8,9,11,12,14\} \cap \{1,8,13,14,15\} = \{1,8\}$.

$\bigcap(F - [1]_b) = [1]_a \cap [1]_d \cap [1]_e = \{1,2,10,11,12,13\} \cap \{1,5,8,9,11,12,14\} \cap \{1,8,13,14,15\} = \{1\}$.

$\bigcap(F - [1]_d) = [1]_a \cap [1]_b \cap [1]_e = \{1,2,10,11,12,13\} \cap \{1,8,9,10,11,15\} \cap \{1,8,13,14,15\} = \{1\}$.

$\bigcap(F - [1]_e) = [1]_a \cap [1]_b \cap [1]_d = \{1,2,10,11,12,13\} \cap \{1,8,9,10,11,15\} \cap \{1,5,8,9,11,12,14\} = \{1,11\}$.

The first equation yields a result of $\{1,8\}$. $\{1,8\} \subseteq \{1,2,4,5,8,9\} = [1]_f$. This means that dropping input variable a from decision rule 1 has no adverse effect on the value of the output variable with regards to the other decision rules. Likewise for the second and third equation. The fourth equation gives a result of $\{1,11\}$. $\{1,11\} \not\subseteq \{1,2,4,5,8,9\} = [1]_f$. This means that dropping input variable e from decision rule 1 does change the correct value for the output variable in another decision rule. Specifically, if we were to drop input variable e, decision rule 11 would not retain the proper value for output variable f. For this reason we must keep the value of input variable e for decision rule 1.

The same procedure is followed for decision rule 2:

$F = [2]_{\{a,b,d,e\}} = [2]_a \cap [2]_b \cap [2]_d \cap [2]_e = \{1,2,10,11,12,13\} \cap \{2,4,5,12,13,14\} \cap \{2,4,10,13,15\} \cap \{2,4,5,9,10,11,12\} = \{2\}$.

$[2]_f = \{1,2,4,5,8,9\}$.

$\bigcap(F - [2]_a) = [2]_b \cap [2]_d \cap [2]_e = \{2,4,5,12,13,14\} \cap \{2,4,10,13,15\} \cap \{2,4,5,9,10,11,12\} = \{2,4\} \subseteq [2]_f$.

$\bigcap(F - [2]_b) = [2]_a \cap [2]_d \cap [2]_e = \{1,2,10,11,12,13\} \cap \{2,4,10,13,15\} \cap \{2,4,5,9,10,11,12\} = \{2,10\} \not\subseteq [2]_f$.

$\bigcap(F - [2]_d) = [2]_a \cap [2]_b \cap [2]_e = \{1,2,10,11,12,13\} \cap \{2,4,5,12,13,14\} \cap \{2,4,5,9,10,11,12\} = \{2,12\} \not\subseteq [2]_f$.

$\bigcap(F - [2]_e) = [2]_a \cap [2]_b \cap [2]_d = \{1,2,10,11,12,13\} \cap \{2,4,5,12,13,14\} \cap \{2,4,10,13,15\} = \{2,12\} \not\subseteq [2]_f$.

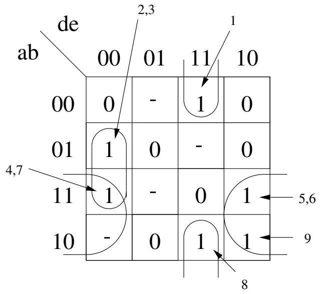
Since removing input variables b, d and e result in extra decision rules appearing in the resulting intersection, these input variables cannot be removed from decision rule 2. Table 6 shows the results of computing the **core values for all decision rules**. It should be noted that Table 6 does not directly correspond to the Karnaugh map of Figure 1, because finding the core values is performed for each decision rule independently. From manipulation of Table 6 (described in detail in [19]), the minimal solution can be found: $bd'e' \vee ae' \vee b'de \rightarrow f$. Observe that the above algorithm is especially efficient for very strongly unspecified MV functions, typical for Data Mining, and this is when its high parallelism proves advantageous. Didactically, it should be noted that the same solution can be obtained through the use of Karnaugh maps. This method may be more familiar to circuit designers and therefore is presented below in Figure 1 (For simplification of the Karnaugh map, input variable c has already been dropped). In general, Kmaps should help the reader with an engineering background to analyze the minimal solution found above using RST and also in understanding all subsequent concepts and algorithms here.

When defining the truth table for a logic circuit there is never a case where the output value can take on different values for the same set of input values. However, in **Data Mining**, it is possible that the data collected could be

Table 6: Results of computing cores

U	a	b	d	e	f
1	-	-	-	1	1
2,3	-	1	0	0	1
4,7	-	-	-	-	1
5,6	1	-	-	0	1
8	-	0	1	-	1
9	1	-	-	-	1
10	-	0	-	-	0
11	0	-	-	0	0
12	0	-	-	0	0
13	-	-	-	1	0
14	-	1	-	1	0
15	-	-	0	-	0

Table 7: Karnaugh Map



erroneous or **inaccurate**, so called **noisy data**. In this case it is indeed possible to have different output values for the same input values in the data set. Table 8 shows an example data set with inaccurate data. Variables a, b, c and d are input variables while f is the output variable. Fig. 9 presents the erroneous data in a Karnaugh map form.

Three Rough Set Theory operations take the possibility of erroneous data into account. They are the **Lower** and **Upper Approximations**, and the **Boundary**. Pawlak [19] defines the lower and upper approximations and boundary in set notation as follows:

$\underline{R}X = \{x \in U : [x]_R \subseteq X\}$
$\overline{R}X = \{x \in U : [x]_R \cap X \neq \emptyset\}$
$BN_R(X) = \overline{R}X - \underline{R}X$

$\underline{R}X$ is the **lower approximation.** x is a decision rule of the universe. U is the universe of knowledge as shown by Table 8. R is an equivalence relation over U and $[x]_R$ is a category in R containing an element $x \in U$. By saying that $[x]_R$ is a category in R it is meant that $x \subseteq R$.

The lower approximation is the set of all elements in U that can be said with certainty that they are elements of X in the knowledge R.

$\overline{R}X$ is the **upper approximation.** Elements in the upper approximation can possibly be classified as elements of X.

$BN_R(X)$ is the **boundary**. Elements in the boundary cannot be classified as being elements of X or not being elements of X.

In addition, Pawlak defines the following:

$POS_R(X) = \underline{R}X$, **R-positive region** of X.
$NEG_R(X) = U - \overline{R}X$, **R-negative region** of X.
$BN_R(X) = \overline{R}X - \underline{R}X$, **R-borderline region** of X.

The **positive region** of X are elements that can be classified with certainty whether that they belong to X. The **negative region** of X are those elements that can be classified with certainty that they do not belong to X. The **borderline region** include those elements that can not be classified with certainty whether they belong to X or that they do not belong to X.

The following example from [19] is presented here to clarify the definitions given above. We are given a universe of knowledge $U = \{x_0, ..., x_{10}\}$ with equivalence classes:

$E_1 = \{x_0, x_1\}$
$E_2 = \{x_2, x_6, x_9\}$
$E_3 = \{x_4, x_5\}$
$E_4 = \{x_4, x_8\}$
$E_5 = \{x_7, x_{10}\}$

The example gives the following roughly R-definable sets:

$X_2 = \{x_0, x_3, x_4, x_5, x_8, x_{10}\}$
$Y_2 = \{x_1, x_7, x_8, x_{10}\}$
$Z_2 = \{x_2, x_3, x_4, x_8\}$

Using these sets, the example defines the following approximations and boundaries:

$\underline{R}X_2 = E_3 \cup E_4 = \{x_3, x_4, x_5, x_8\}$.
$\overline{R}X_2 = E_1 \cup E_3 \cup E_4 \cup E_5 = \{x_0, x_1, x_3, x_4, x_5, x_7, x_8, x_{10}\}$.
$BN_R(X_2) = E_1 \cup E_5 = \{x_0, x_1, x_7, x_{10}\}$.
$\underline{R}Y_2 = E_5 = \{x_7, x_{10}\}$
$\overline{R}Y_2 = E_1 \cup E_4 \cup E_5 = \{x_0, x_1, x_4, x_7, x_8, x_{10}\}$
$BN_R(Y_2) = E_1 \cup E_2$
$\underline{R}Z_2 = E_4 = \{x_4, x_8\}$
$\overline{R}Z_2 = E_2 \cup E_3 \cup E_4 = \{x_2, x_3, x_4, x_5, x_6, x_8, x_9\}$
$BN_R(Z_2) = E_2 \cup E_3 = \{x_2, x_3, x_5, x_6, x_9\}$

The reader is directed to [19], pp. 18-19, for more information regarding this example. The concepts of positive, negative and the borderline regions will be explained using Table 8 and Table 9, the Karnaugh map representation of the dataset. For this example let us define X to be the set of all decision rules that have an output value of 1: $X = \{1, 3, 5, 6, 7, 10, 11, 12\}$. The following Rough Set classifications can be made.

$POS_R(X) = \underline{R}X = \{1, 6, 7, 10, 11, 12\}$.
$NEG_R(X) = U - \overline{R}X = \{8, 9, 13, 15, 14\}$.
$BN_R(X) = \{2, 3, 4, 5\}$.

The presented RST operations are not limited to only binary logic, but can also be applied to MV logic.

Table 8: Dataset with inaccurate data

U	a	b	c	d	f
1	0	0	0	0	1
2	0	0	1	0	0
3	0	0	1	0	1
4	0	0	1	1	0
5	0	0	1	1	1
6	0	1	0	0	1
7	0	1	0	1	1
8	0	1	1	0	0
9	0	1	1	1	0
10	1	0	0	1	1
11	1	0	1	0	1
12	1	0	1	1	1
13	1	1	0	1	0
14	1	1	1	0	0
15	1	1	1	1	0

Table 9: Karnaugh map showing **Lower and Upper Approximation and Boundary**

ab \ cd	00	01	11	10
00	1^{1}	-	$0,1^{4\ 5}$	$0,1^{2\ 3}$
01	1^{6}	1^{7}	0^{9}	0^{8}
11	-	0^{13}	0^{15}	0^{14}
10	-	1^{10}	1^{12}	1^{11}

4. A SIMD Parallel Rough Sets Computer.

Muraszkiewicz and Rybinski [17] present algorithms and a possible implementation of a machine to perform the fundamental Rough Set Theory calculations of upper approximation, lower approximation and indispensibility. The name of their machine is the Parallel Rough Sets Computer or PRSComp. Figure 1 shows the overall architecture of PRSComp. They proposed a machine consisting of $m * n$ primitive processors. An individual processor is connected to its four neighbors (north, east, south and west), as well as to global control signals. The processors execute in lock-step with one another under coordination of the global control signals. In a traditional von Neumann computer a single instruction operates on a single piece of data at any point in time. This is SISD

(Single Instruction Single Data). PRSComp operates as a SIMD (Single Instruction, Multiple Data) computer. When designing a parallel computer the question arises of the origin of instructions executed by the processors. Should each processor run its own programs or should the instructions come from a central source? In a SIMD parallel computer the instructions come from a single source (an FSM controller) and therefore each processor is executing the same instruction at a same time. Because of its rectangular shape and simple cells, this is an ideal architecture for DEC-PERLE-1, assuming a small number of instruction types. A small number of instructions leads to narrow control bus, which is a requirement of this technology [26] (the central controller is realized in FPGAs outside the main FPGA array, and a large number of connections to it results in the design bottleneck). FPGA realization allows for fast prototyping and also allows us to implement only those operations that are actually needed for any given particular application. Thus, the FPGA approach decreases the cell size and increases the array size that can fit in the given physical board resources, in comparison to a hypothetical "universal" ASIC machine that would realize in hardware all potential operations of the adopted calculus (in our case RST). In PRSComp the instructions are provided by a central resource: the global control signals to each processor (not shown in Figure 1). Each processor is connected to the global control signals and therefore each processor performs the same operation, defined by the instruction at a particular time. This explains the Single Instruction part of the SIMD classification of PRSComp. The input data, on which calculations are to be performed, is mapped into the $m * n$ processors as a binary matrix $A_{m,n}$, each processor taking on one element of the matrix. Note, that this means that a single processor operates on only a single bit at a time. Each processor operates on its own data that is independent of the other processors. This explains the Multiple Data part of the SIMD term. Three registers are utilized in the Rough Set Theory calculations: the **column mask register** (CM) (horizontal), the **comparand register** (C) (horizontal) and the **word selection register** (E) (vertical). These registers, along with the global control signals direct the operation of each processor. The column mask register is used to inhibit the processing of matrix cells. The comparand register is used to transfer words to and from the processor array, as well as taking part in comparison operations. The word selection register contains the result of a comparison.

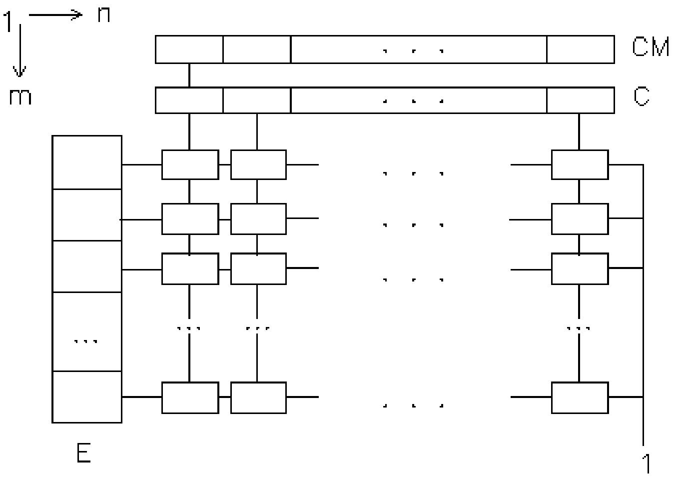


Figure 1: PRSComp Architecture

5. PRSComp Algorithms and Examples

Six routines to compute the fundamental operations of Rough Set Theory are presented in [17]. The routines are as follows: **BasicCAT** (Basic Category), **UpperAPPROX** (Upper Approximation), **LowerAPPROX** (Lower Approximation), **Def** (Definable), **Indispensable** and **EXTCOMP** (External Comparison). These algorithms are presented in Figure 2. The logical OR operation is represented by "+". The logical AND operation is represented by "*". The algorithms will be illustrated by the example from Table 1.

```
Routine EXTCOMP ( A[m, n], C[n], E[m] )
/* C[n] is the comparand */
    MASK
    E[m] := 0
    for j = 0 to n if c[j] = a[i, j] then a couple is transparent
    labeled by "t", otherwise it is opaque labeled by "o"
    logic value of 1 is propagated through rows
    of the array, if all cells in the row are transparent
    then 1 is injected into that position in the E register

Routine BasicCAT ( A[m,n], i, E[m] ) /* Basic category */
/* The E register contains the characteristic vector that
indicates the words belonging to the basic category
generated by a_i */
    MASK
    C[n] := a_i
    EXTCOMP( A[m, n], i, E[m] )

Routine UpperAPPROX ( A[m, n], B[m], C[m] )
/* B[m] stores the characteristic vector of the set X, input */
/* C[m] the characteristic vector of the upper approximation of X,
output */
    for i = 1 to m BasicCAT( A[m, n], i, E_i[m] )
    for i = 1 to m S_i[m] := E_i[m] * B[m]; C[m] := 0
    for i = 1 to m if E_i[m] ≠ 0 then C[m] := C[m] + E_i[m]

Routine LowerAPPROX ( A[m, n], B[m], C[m] )
/* B[m] stores the characteristic vector of the set X, input */
/* C[m] the characteristic vector of the lower approximation of X,
output */
   for i = 1 to m BasicCAT( A[m, n], i, E_i[m] ).
   for i = 1 to mS_i[m] := E_i[m] * B[m]; C[m] := 0.
    for i = 1 to m if E_i[m] = S_i[m] then C[m] := C[m] + E_i[m]

Routine Def ( A[m, n], B[m] ) /* Definability */
/* B[m] stores the characteristic vector of the set X */
    UpperAPPROX ( A[m, n], B[m], C[m] )
    LowerAPPROX ( A[m, n], B[m], C'[m] )
    if C[m] = C'[m] then the set X is definable

Routine Indispensable ( A[m, n], j ) /* Indispensability */
/* j indicates the column number to be checked out */
    for i = 1 to m
    begin
            MASK
            BasicCAT ( A[m, n], i, B[m] )
            (MASK - {j})
            BasicCAT ( A[m, n], i, C[m] ) ???
            if B[m] ≠ C[m] then {j} is indispensable
     end
```

Figure 2: Algorithms presented by Muraszkiewicz and Rybinski.

Another notion used in the PRSComp is the **characteristic vector** of a subset. If $X \subseteq U$ then we can say that the characteristic vector of X against $A(m, n)$ is:

Table 10: Initial State of PRSComp

	CM						
	C						
E	U	a	b	c	d	e	f
	1	0	0	0	1	1	1
	2	0	1	0	0	0	1
	3	0	1	1	0	0	1
	4	1	1	0	0	0	1
	5	1	1	0	1	0	1
	6	1	1	1	1	0	1
	7	1	1	1	0	0	1
	8	1	0	0	1	1	1
	9	1	0	0	1	0	1
	10	0	0	0	0	0	0
	11	0	0	0	1	0	0
	12	0	1	1	1	0	0
	13	0	1	1	0	1	0
	14	1	1	1	1	1	0
	15	1	0	0	0	1	0

Table 11: Intermediate State of PRSComp

	CM	1	1	1	1	1	1
	C	0	0	0	1	1	1
E	U	a	b	c	d	e	f
0	1	0	0	0	1	1	1
0	2	0	1	0	0	0	1
0	3	0	1	1	0	0	1
0	4	1	1	0	0	0	1
0	5	1	1	0	1	0	1
0	6	1	1	1	1	0	1
0	7	1	1	1	0	0	1
0	8	1	0	0	1	1	1
0	9	1	0	0	1	0	1
0	10	0	0	0	0	0	0
0	11	0	0	0	1	0	0
0	12	0	1	1	1	0	0
0	13	0	1	1	0	1	0
0	14	1	1	1	1	1	0
0	15	1	0	0	0	1	0

Table 12: After first execution of EXTCOMP routine

	CM	1	1	1	1	1	1
	C	0	0	0	1	1	1
E	U	a	b	c	d	e	f
1	1	0 t	0 t	0 t	1 t	1 t	1 t
0	2	0 t	1 o	0 t	0 o	0 o	1 t
0	3	0 t	1 o	1 o	0 o	0 o	1 t
0	4	1 o	1 o	0 t	0 o	0 o	1 t
0	5	1 o	1 o	0 t	1 t	0 o	1 t
0	6	1 o	1 o	1 o	1 t	0 o	1 t
0	7	1 o	1 o	1 o	0 o	0 o	1 t
0	8	1 o	0 t	0 t	1 t	1 t	1 t
0	9	1 o	0 t	0 t	1 t	0 o	1 t
0	10	0 t	0 t	0 t	0 o	0 o	0 o
0	11	0 t	0 t	0 t	1 t	0 o	0 o
0	12	0 t	1 o	1 o	1 t	0 o	0 o
0	13	0 t	1 o	1 o	0 o	1 t	0 o
0	14	1 o	1 o	1 o	1 t	1 t	0 o
0	15	1 o	0 t	0 t	0 o	1 t	0 o

Table 13: Result after second call to EXTCOMP

	CM	0	1	1	1	1	1
	C	0	0	0	1	1	1
E	U	a	b	c	d	e	f
1	1	0 t	0 t	0 t	1 t	1 t	1 t
0	2	0 t	1 o	0 t	0 o	0 o	1 t
0	3	0 t	1 o	1 o	0 o	0 o	1 t
0	4	1 t	1 o	0 t	0 o	0 o	1 t
0	5	1 t	1 o	0 t	1 t	0 o	1 t
0	6	1 t	1 o	1 o	1 t	0 o	1 t
0	7	1 t	1 o	1 o	0 o	0 o	1 t
1	8	1 t	0 t	0 t	1 t	1 t	1 t
0	9	1 t	0 t	0 t	1 t	0 o	1 t
0	10	0 t	0 t	0 t	0 o	0 o	0 o
0	11	0 t	0 t	0 t	1 t	0 o	0 o
0	12	0 t	1 o	1 o	1 t	0 o	0 o
0	13	0 t	1 o	1 o	0 o	1 t	0 o
0	14	1 t	1 o	1 o	1 t	1 t	0 o
0	15	1 t	0 t	0 t	0 o	1 t	0 o

$b_i(X) = 1$ for i such that $A(i,*) \subseteq X$
0 otherwise
$i = 1$ to m

m is equivalent to the number of decision rules (rows) in the input dataset. The characteristic vector, $B[m]$, is a vertical vector of height m. If b_i is equal to 1 then the ith decision rule belongs to the set X.

The first step in Section 3 was to determine which input variables were redundant also known as finding which input variables are dispensable. In Section 3, this was determined by removing one input variable at a time and determining whether or not the resulting table was consistent. This was shown in Table 2 through Table 5. Here we investigate how this operation can be performed in PRSComp. First note that each element of the original truth table, Table 1, is mapped to a single processor. The initial state of the PRSComp is shown in Table 10.

Note, that the U column is not stored within PRSComp, but is merely shown in the table to aid discussion. Initially, the contents of the CM (column mask), C (comparand) and E (word selection) register values are undefined. Each box in Table 10 corresponds to one primitive processor within PRSComp.

The Indispensable routine is used to determine which input variables can be removed without adversely affecting the results. The **for** loop within the Indispensable routine executes its contents once for every row in Table 10. The first step in the Indispensable routine is to set up the CM register, shown in the routine Indispensable as $\overline{MASK}$. In this case, the CM register is initialized to all 1s, meaning that no processors are masked. Next, the BasicCAT routine is called. The BasicCAT routine sets up the CM register, again with all 1s. The statement C[n] := $\underline{a}_i$ in the BasicCAT routine means that the current row is copied to the **comparand register** (C). The EXTCOMP routine is then called and sets up the CM register again. The E register is then initialized to all 0s, E[m] := 0. The intermediate state of the machine is shown in Table 11.

The remainder of the EXTCOMP routine is executed and the resulting state of the PRSComp is shown in Table 12.

It is interesting to note that for a given row the set of all cells in a particular column that are labeled as transparent is identical to what was used above with the Rough Set example as the set of decision rules having the same value for a particular variable. More specifically, in Table 12, under the column for input variable a the row numbers that are labeled with "t" are 1, 2, 3, 10, 11, 12 and 13. From Section 3 we found that: $[1]_a = \{1, 2, 10, 11, 12, 13\}$. This is the case for every column. The current value of the E register is then saved and the position in the CM register corresponding to the column being checked, let's say the column for variable a is set to 0 . The EXTCOMP routine is called again and Table 13 shows the results after complete execution of the EXTCOMP routine.

The new value of the E register is compared with the previously stored value of the E register and they are found to be **not equivalent**. This means that input variable a is indispensable (i.e. it must be kept in the final solution). It is also interesting to note that masking off input variable a is analogous to removing input variable a and checking the intersection.

$\bigcap(F - [1]_a) = [1]_b \cap [1]_d \cap [1]_e = \{1, 8, 9, 10, 11, 15\} \cap$

$\{1,5,8,9,11,12,14\} \cap \{1,8,13,14,15\} = \{1,8\}$

The result of the intersection is 1 propagated through the processors that are transparent to the final E register. If a processor is marked as transparent then the 1 is propagated to the left. If a processor is marked as opaque then the 1 is not propagated. In hardware, all comparisons are done in parallel for every row. The value of the E register is stored. Then, a column is masked and the comparisons occur once again, generating a new E register. This new value of the E register is compared to the old value of E register and if they are equivalent the current input variable (column) is dispensable and can therefore be removed.

6. Conclusions

We presented principles of the Learning Hardware as a competing approach to the Evolvable Hardware, and also as its generalization. The concept of a Data Mining machine based on Rough Sets has been outlined and some basic operations and algorithms discussed. A high degree of low-level parallelism of these algorithms calls for an FPGA, ASIC or Contents Addressable Memory (CAM) realization. Although the DEC-PERLE-1 is a good medium to prototype such machines, massively parallel architectures such as CBM based on new Xilinx series 6000 chips will allow larger arrays and internal memories, and thus much higher speedups.

References

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

[2] G. Almasi and A. Gottlieb, "Highly Parallel Computing," *The Benjamin/Cummings Publ. Co.*, 1994.

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

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

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

[6] 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.

[7] 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.

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

[9] $http://www.hip.atr.co.jp/\tilde{d}egaris$

[10] M. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE TC*, Vol C-21, pp. 948-960, 1972.

[11] T. Higuchi, M. Iwata, and W. Liu (eds), "Evolvable Systems: From Biology to Hardware," *Lecture Notes in Computer Science,* No. 1259, Springer Verlag, 1997.

[12] L. Jozwiak, M.A. Perkowski, D. Foote, "Massively Parallel Structures of Specialized Reconfigurable Cellular Processors for Fast Symbolic Computations," *Proc. MPCS'98,* Colorado Spgs, CO, Apr. 6-9, 1998.

[13] L. Jozwiak, N. Enderveen, A. Postula, "Solving Synthesis Problems with Genetic Algorithms," *Proc. Euro-Micro'98*, Vasteras, Sweden, August 25-27, 1998, pp. 1 - 7.

[14] P.J. Lingras and Y.Y. Yao, "Data Mining Using Extensions of the Rough Set Model," *J. Amer. Soc. Inform. Sci.,* 49(5):415-422, 1998.

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

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

[17] M. Muraszkiewicz and H. Rybinski, "Towards a Parallel Rough Sets Computer", In "Rough Sets, Fuzzy Sets and Knowledge Discovery" Wojciech P. Ziarko (ed), Springer Verlag, pp. 434-443.

[18] L. Moll, J. Vuillemin and P. Boucard, "High-Energy Physics on DECPeRLe-1 Programmable Active Memory", *Proc. 1995 ACM Intern. Symp. FPGAs*, Monterey, CA, Feb. 1995.

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

[20] M. Perkowski, "Systolic Architecture for the Logic Design Machine," *Proc. ICCAD'85,* pp. 133 - 135, Santa Clara, 19 - 21 Nov. 1985.

[21] M.A. Perkowski, "A Universal Logic Machine," invited address, *Proc. ISMVL'92,* pp. 262 - 271, Sendai, Japan, May 27-29, 1992.

[22] 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, CA, May 1997, pp. 13 - 18.

[23] 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," *6th Intern. Work. Post-Binary ULSI Systems,* CA, May 27, 1997, pp. 41 - 44.

[24] 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 "Reconfigurable Architectures. High Performance by Configware," R.W. Hartenstein, V.K. Prasanna (ed), IT Press Verlag, Bruchsal, Germany, pp. 33 - 40, 1997.

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

[26] M. Perkowski, "Do It Yourself Supercomputer that Learns," *Book preprint*, 1999.

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

[28] 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