SYSTOLIC ARCHITECTURE FOR SOLVING NP-HARD COMBINATORIAL PROBLEMS OF LOGIC DESIGN AND RELATED AREAS

Phuong Minh Ho, Marek A. Perkowski,

Department of Electrical Engineering, Portland State University
P.O. Box 751, Portland, OR 97207, tel. (503) 464-3806

ABSTRACT

This paper presents a new approach to solving various NP-hard problems in logic synthesis, logic programming, graph theory, and related areas. A problem to be solved is reduced to solving one or several generic combinatorial problems, called Generalized Propositional Formula (GPF) minimization. The special massively parallel computer architecture for the GPF minimization is discussed. The architecture is composed of a Host computer and a data-flow tree (DFT) of processors (Boolean Product Processor - BPP). Each BPP consists of a Product Management Unit (PMU) and a Sorting and Absorbing Architecture (SAPA).

1. INTRODUCTION

It is not well known that logic computers, i.e. devices to aid in verifying sylogisms and solving other logic problems are older than the digital arithmetic computers and date back to the Middle Age "computers" by Lullus. A variety of such computers were built in the XIX and XX centuries before Von Neumann [12]. The first, to our knowledge, special computer to help in logic synthesis was proposed by Antonin Svoboda in Czechoslovakia and then in the United States [20, 21]. Svoboda's group was actively working on them in the early 70's [8, 19, 3]. Some ideas on such computers are also in [10]. We did not find other references on this subject until 1985 [18, 11], but there was related research on the tree-searching computers [23] and consistant labeling computers [9]. Since the hardware accelerators for simulation, design rule checking, routing, placement, and other layout tasks are now available or proposed, we believe that hardware accelerators for logic design will also be incorporated into future CAE workstations for VLSI design because there is an obvious and growing need for them. Since 1984, we have been designing, analyzing, and simulating various architectures related to logic synthesis [11, 5, 4]. We also work on the implementation of the respective algorithms on parallel computers: on iPSC hypercube from Intel [13, 14], Sequent Balance [15], and XTM from Cogent.

As it is shown in [10-14], many problems of logic, system, high-level and physical VLSI design, as well as in graph theory, logic programming and pattern recognition, can be reduced to one of the following two problems.

Given is a Boolean function of the form:

$$\prod_{t} \sum_{P_r \in T_t} \prod_{j \in P_r} x_j^i \quad i \in \{0, 1\}\ ,\ (x^0 = \bar{x}\ , x^1 = x).$$

T_t is the set of *literals* in the *term* t. This means that *the function is a product of sums of products of literals.* The term (or sum) is a sum of products of literals. We will use this kind of function in this paper since it is more general than the decision functions from the literature (Product of Sums Form - PSF of *Petrick* [16] or *satisfiability formula* [2]) and it is also more suitable for our method of data coding in systolic flows. Because of the important role of this function in many problems and since no name has been assigned to it, **we propose to call it a Generalized Propositional Formula (GPF).**

Problem 1 (Satisfiability): Answer *Yes* if there exist a product of literals that satisfies all terms or *No* if such a product does not exist.

Problem 2 (Optimization): Find a product with a minimum number of literals that satisfies all of the terms or prove that such a product does not exist. Or, find the sum of all products (SPF) that satisfy the GPF.

The GPF also finds applications in Petri net analysis, expert systems, pseudo-Boolean programming, data base consistency, CAD, operations research, graph theory, search theory, and criptography. Many other reductions to the problems formulated above, as well as new problems of this class, can be found in [2, 10-14]. [6] is a continuous source of interesting problems. **Needless to say, every NP-hard combinatorial problem can be theoretically reduced to GPF minimization, and such reductions for many problems are reasonable, also from the practical point of view.** The algorithms to solve the above problems can basically be divided into three categories: *tree searching algorithms, array algorithms, and transformational algorithms.* In this paper, it is assumed that the first two types are executed by the Host and the last one by the architecture discussed here.

Basically, we can distinguish *two types* of practical GPF problems. The problems related to Boolean minimization, for instance, require a relatively *small* (≤ 64) number of variables. Then, 128 literals (bits) are sufficient to represent a product of a sum, assuming, like in computer realizations of cube calculus [22], two bits of the word for a variable. Such an approach is discussed in this paper, where we propose a GPF Solver (GPFS) architecture for the two above and other similiar problems. When reducing other problems of this class, however, like, for instance, in the Petrick function minimization approach to set covering for Boolean function minimization [16], the number of the variables (rows of the covering table) can be *in the thousands.* We have developed another architecture [5], similar to the one presented here, that can effectively deal with such problems. The computer word is used in it for a literal, not for a product of literals, as assumed here.

2. THE GPFS ARCHITECTURE

The product of literals is represented as a computer word, two bits per variable. To focus our considerations, we assume 61 variables plus 6 bits (LSB) for product cost. Then, 128 bits in a word. The encoding is as follows: x - 10, $\bar{x}$ - 01, don't care - 11, *contradiction* - 00. In this notation, the product of two products of literals simply corresponds to a bit-by-bit product of the respective words. For instance, assuming 5 variables, (a, b, c, d, e), the product is $be \cdot bc\bar{d}$ = [11 10 11 11 10] · [11 10 10 01 11] = [11 10 10 01 10] = $bc\bar{d}e$. When the opposite literals are multiplied, the pair 00 is created from the bit-by-bit product and is detected in the next stages: $ab \cdot a\bar{b}$ = [10 10 11 11 11] · [10 01 11 11 11] = [10 00 11 11 11] = *contradiction*.

The GPFS architecture is shown in Fig. 1. It consists of the Host and the, tightly coupled with it, convergent Data Flow Tree (DFT) of Boolean Product Processors (BPP). Each BPP is composed of the Product Management Unit (PMU) and the Sorting and Absorbing Architecture (SAPA). The number of levels in DFT can be arbitrary, and the algorithms will not change with the number of levels. Each PMU sub-system contains individual control unit and local memory. They are operated asynchronously, and may coordinate their processes by communicating with each other directly or through the host computer by means of software semaphore/arbiter.

The Host does the partitioning of the larger problem to fit it to the constraints of the DFT, such as the length of the word (number of literals), the sizes of memories in BPPs (the numbers of the terms and products), and others. The partitioning can be done during the tree searching. In such a case, the variables are systematically reduced and the formulas from the leafs of the tree, with less than 61 variables are transmitted to the DFT. In another variant, partitioning is done with respect to smaller products of terms from the GPF formula. Several well known algorithms can be adopted for these partitionings and new algorithms have been written [1, 11, 12].

CH2692-2/89/0000-1170 $1.00 © 1989 IEEE

The host also loads the memories of the leaf-processors of the DFT (level 1 BPPs #1 and #2 in Fig. 1), whenever it gets the "load me" signal from the respective processor. It also receives the partial resultant SPF expression from the root of the DFT (level 2 BPP #3 in Fig. 1). SPFs are multiplied and locally simplified as they traverse from the Host to the root of the DFT. From the external point of view, the whole process is analogous to the parallel realization of the naive Boolean multiplication/simplification algorithm for a product of terms. The most general and important aspect of this architecture is the general formulation of the problem as asynchronous data flow reduction architecture where both the Host and the DFT, as well as each BPP in the DFT, take some part of the GPF, being again a smaller GPF, multiply it, and replace it with a corresponding SPF form which becomes a part of a new GPF for the next phase of multiplication. All of these multiplications and replacements are done asynchronously, in parallel, and in a pipelined mode inside the DFT. Each processor in the tree serves for Boolean multiplication of the two input streams of data (Cartesian Product Generator CPG). Each such stream of data represents a Boolean SPF from the partial multiplications. Additionally, there is massive parallelism inside each BPP processor, both for Cartesian Product and Sorting/Absorbing as well.

Each BPP processor simplifies the intermediate SPF terms, created by the Cartesian Product, by removing the products with contradiction (called *empty products*) and the repeated products, and sorting them (SAPA). Whenever, after the simplification, the term is empty, this means that the total GPF has no solution. For instance, if at anytime, the left and right SPF terms given to a BPP are ab and $(\bar{a} + \bar{b})$, respectively, the multiplication result in this BPP is an empty term. This is communicated to the Host and means that the entire GPF that includes $ab(\bar{a} + \bar{b})$ has no solution. All BPP processors are cleared and a new problem can be started in the DFT.

Because the products with contradictions are removed from SPFs, the sizes of the terms do not grow too quickly when both negated and positive literals exist for the same variables. When all literals are positive, like in the set covering problem, or when the GPF is an unate function [1], as in [15] for prime implicant generation, the length of the terms grows. In such a case, if we are not interested in all products from the final SPF, it can be assumed that the term length size constraints of each processor are exceeded by the created in it SPFs. Thanks to sorting, however, only the best (shortest) products are accumulated in the sums and transmitted to the higher level BPPs. This approach can mean losing the optimal cover or not generating all implicants but this disadvantage is recompensated by the gain in speed and smaller memories. In addition, the simulation results prove that, with the resonable memory size reductions in the BPP processors, the optimum solution is generated for the covering problems in most cases. Since more processing is executed in the processors at higher levels, and the SPFs are longer there, the memory sizes on higher levels are larger and there is more parallelism in each PMU to calculate the Cartesian Products.

In order to balance the processor loads and the communication among them better, the result of each multiplication of two terms is either accumulated in the internal memory and then multiplied by the next term coming to the second input, or it is transmitted through memory to the successor processor. The ALU in CPG is simple and is optimized for the operation of word product and Sorting/Absorbing only.

3. PRODUCT MANAGEMENT UNIT (PMU).

Architectural description:

The overall internal structure of the PMU - as illustrated in Fig. 2 - includes:

a) A Micro-Controller Unit (CU).

b) A Cartesian Product Generator (CPG).

c) A local memory storage.

Functional description:

a) The Micro_Controller Unit:

* does I/O interfacing with the host computer and the associated SAPAs.
* Dynamically allocates/de-allocates memory storages in the local memory unit to accomodate new input data and remove all existing data items not further in use.
* Controlls the Cartesian Product Generator operations, handling its input/output streams.

b) The Cartesian Product Generator:

* Receives data from the host computer into its array of shift registers through the micro-controller I/O Interface.
* Performes 128-bit logical AND on every two given products to generate a new Cartesian Product to be processed by the SAPA.

c) A local memory storage must be large enough to store the sum of products accummulated by iteratively multiplying the given terms.

Operational description:

At system initialization, the Host computer processes the given function, performs the table reduction process, and sends 2r (2r = 4 for clarity here) terms to each PMU. The CUs will allocate local memory storage to accomodate the first three input terms in forms of linked-list structures. The fourth term will be transferred to the CPG to be loaded into its array of registers, AR. The data transmitted from the host computer must be in the format of 128-bit products. The terms are separated by an empty product.

Upon completion of receiving initial data from the host, the CU will retrieve a product from the first term in its local memory, load it into register R in the CPG, and de-allocate the memory storage preserved for this data item. As the CPG completely multiply (ie. logical AND) the contents of R by every item in AR, the CU fetches another data item (of the same term) into R from the local memory, and the corresponding product node of this term's linked-list is removed from the local memory. As all possible products of the first term (from local memory) and the second term (in AR) have been generated, the CU requests the Host or the other BPP to transmit a new term into AR. The process continues with the elements of the second term retrieved/removed sequentially from the local memory to R, to be bit-wise ANDed to every element of the new term in AR. The results are sequentially outputted to the accompanying SAPA. During this process, the PMU may receive the output from the SAPA. The CU must allocate memory to store these accummulated results in a new linked-list of product nodes in its local memory. In one architecture variant [5] there are k registers R and k parts of the Cartesian Product are calculated in parallel.

4. SORTING AND ABSORBING PARALLEL ARCHITECTURE

Architectural and Functional description. The overall internal structure of the SAPA - as illustrated in Fig. 3 consists of:

a) *A Product Absorption Unit (AU),*

b) *A Product Cost Evaluation Unit (PCEU),*

c) *An Empty Product Detection Unit (EPDU),*

d) *A Pipelined Parallel Quad-Tree Sorter (QTS).*

A Product Absorption Unit (AU) detects and eliminates all *dominating products* by applying the Absorption Law of Boolean Algebra *($ab + a = a$, i.e. ab dominates a* and ab is removed from the term). The AU consists of an array of N Product Domination Detectors (PDD), and two arrays of 128-bit-shift-registers, AR1 and AR2, of dimension N. The contents of the i-th registers in AR1 and AR2 (ie. AR1[i], and AR2[i]) are parallely loaded into the i-th PDD (ie. PDD[i]) to be checked for the domination relation. Data in AR1[i] can be copied into AR2[i]. Initially, all registers in AR1 and AR2 are initialized to Empty Product with all bits equal to '0'. As the 128-bit-products serially outputted from the PMU are fetched into the left end of AR1, this array shifts all of its contents one location to the right and the data in each register of AR1 and AR2 are pairwise checked by the accompanying PDD for the domination property. According to the results of the PDD execution, the contents of AR1 may be transfered into AR2 to be evaluated before entering the SAPA. The procedure executed by the AU is as follows.

```
For every register i-th in the range of (1 .. N-1) parallely do
 Begin
  step 1: AR1[i+1] = AR1[i]
  step 2: Receive a 128-bit product into AR1[1]
  step 3: Load AR1[i] and AR2[i] into PDD[i]
  If AR2[i] dominates AR1[i] then
    Begin
     AR2[i] := AR1[i]
     AR1[i] := Empty Product (all bits = '0')
    End
    Else if AR1[i] dominates AR2[i] then
```

(217) 333-4789 Further information can be obtained from Dr. W. Kenneth Jenkins.

```
    Begin
     AR1[i] := Empty Product
    End
   Else if NO DOMINATION DETECTED then
    Begin
     Skip, no operation
    End
End.
```

As the AU processes all 128-bit-product of the current SPF received from the PMU, it will parallely transfer the entire contents of the AR2 array into another buffer. Each product in this buffer will be sequentially examined to have its cost evaluated and be discarded if empty. During that interval, the PMU is generating another SPF by forming the Cartesian product of another terms (in its local memory) and a new term is inputted into the CPG. The result products are serially outputted to the AR1 and AR2 arrays. The complete process of the AU is performed while the QTS sorts the previous SPF.

A Product Cost Evaluation Unit (PCEU) computes the number of literals existing in the given 128-bit product. The product cost is stored in the Cost-field (the least 6 significant bits) of the input 128-bit storage.

An Empty Product Detection Unit (EPDU) detects the empty products, i.e. those that include at least one empty field (ie. 00). The EPDU receives, and examines the 128-bit products one by one, prior to entering them into the two-level input buffer of the Quad-Tree-Sorter. All empty products are removed.

A Pipelined Parallel Quad-Tree Sorter (QTS), as illustrated in Fig. 4, can be classified as an I/O Overlapped Parallel Input/Sequential Output sorting system. It is created in the form of a multilevel, divergent, balanced quad-tree. Given N as the number of data items to be sorted, the system consists of:

i) A two-level register array which buffers the input data items, ie. products from the PMUs, to be sorted. The product costs of the items in different levels are pairwise compared, and swapped (if necessary) to obtain the lower cost products at the level nearest to the processing elements at the leaf-level of the QTS.

ii) A TL level quad-tree which parallely receives input lists into the nodes at the leaves and sequentially outputs every data item of the sorted lists at the root node.

iii) TP Processing Elements (PE) which are located at the nodes of the tree. As depicted in Fig. 4, each PE includes three comparators (C1, C2, and C3), and a combinational control circuit. Data are fetched from four PIPEs at the lower level (child nodes) of the tree into the cells of comparators C1 and C2. The Status signals (S1 and S2) generated by C1 and C2 are fed to the control circuit together with Enable (E) and Forward (F) input signals to determine whether the data item in the left or right cells of the comparators C1 or C2 will be transferred to the left or right cell of the third comparator C3. Signals S1 and S2 of the (j+1)-th level are also fed back to the j-th level of the tree to control the entire data flow in the QTS. At the same time, C3 performs its comparison with the prefetched inputs and decides to push one of its cell contents into its associative intermediate output PIPE. In this application, the costs of any two products are compared and the product with smaller cost is advanced toward the root of the quad-tree structure to be outputted first. If two products have the same cost, the one on the left has the higher priority. Some basic design specifications are pointed out for the sake of clarity:

   * The comparator issues the Status signal, S[i], which is set or reset, depending on the comparison of the comparator Ci. An '1' if the left value is greated than the right one. '0', otherwise.
   * The contents in the left (or right) cell of a comparator is transferred into the adjacent available register in a comparator or a PIPE, if the Enable signal, E[i], is set to '1' (or '0', respectively).
   * The left-most values contained in the C1 comparators (of the PEs on a tree level) are used for determining the status of the Forward signal. F[i] is set to '0' if the i-th level of the QTS is active, '1', otherwise.

iv) TQ FIFO queues are associated with each PE (PIPE). These *First In First Out* queues, located at the output of each PE, have different length based on their locations at different levels of the quad-tree, e.g. every PIPE that exists at the j-th level of the tree must have ($N/4^j$ -1) cells. These PIPEs provide a means to accummulate and transfer partially sorted lists of data from one to the next level in the tree hierarchy.

The formulas for computing the TL, TP, TQ, and TE parameters are: N = Number of products in the unordered input list, TL = Number of tree levels in the QTS, TP = Number of Processing Elements, TQ = Number of FIFO queues required, TE = Number of execution steps required to complete the sort, $\mathrm{TL} = \log_4(N)$,

$$\mathrm{TP} = \sum_{i=0}^{TL-1} 4^i, \quad \mathrm{TQ} = \mathrm{TP}, \quad \mathrm{TE} = \mathrm{N} + \mathrm{TL} - 1 + \sum_{i=1}^{TL}(\frac{N}{4^i} + 1).$$

Operational description. The unsorted data lists generated by the PMU are parallely fetched into the nodes at the leaves of the tree through the m level input buffer. Every four adjacent 128-bit products in the input list are loaded in the left and right cells of the comparators C1 and C2 of a PE. At every execution step, all the comparators will compare their product costs and select the one which has the smaller cost, to be passed to the comparators of the next level, or to the pipeline of products in the associated PIPE. As the data flow through a level of the tree, they are partially sorted and stored as partitioned sorted lists in the intermediate level PIPEs. The products with smallest costs will traverse toward the root of the tree first, in a sorted order. They will be serially transferred to the indicated PMU to be further processed, or passed back to the Host computer.

5. CONCLUSION AND EXTENSIONS TO GPFS

The architecture presented above as well as several of its variants and derivatives were simulated using programs on an IBM PC AT computer [5, 7]. The correctness of the architecture has been verified. It was also compared to other architectures and the issues of communication/load balancing, time/area trade-offs, memories and word sizes were analyzed.

The above architecture can also be used for very fast sorting, sorting/absorbing or absorbing of Boolean functions or other vectors of binary vectors which find applications to many problems. Replacing parallel operations on words to serial decreases essentially the cost of the architecture with only linear decrease of its speed. The control becomes somewhat more complicated. We have not discussed, therefore, the bit serial variant here. It is, however, better suited for VLSI implementation.

The extension of the generality of this architecture can be done by extending the number of bit-by-bit operations in ALUs of CPGs, from bit-by-bit AND to include all 2-variable Boolean functions (which has applications to other cube calculus operations [22, 1]. For instance, the PMU can easily be extended to solve the multiplication of type $\sum o \prod x^i \cdot \sum o \prod x^i$ where $\sum o$ denotes EXOR function [4, 5]. Other extensions are obtained by checking other relations on single arguments and pairs in the SAPA. Only the detection of contradiction is executed above on single elements, and the relations of >, bit-by-bit inclusion and comparison of number of ones are checked on two arguments. All problems discussed above can also easily be extended to logic with multiple-valued inputs [17]. In such a case, the positional notation is used for products. Assuming 3 values and 3 variables, (a, b, c), we have $a^0 b^{1,2} \cdot a^{0,2} b^{0,1} c^1 = [100-011-111] \cdot [101-110-010] = [100-010-010] = a^0 b^1 c^1$. A triple of zeros would mean a contradiction.

6. LITERATURE

[1] Brayton, R.K., Hachtel, G.D., McMullen, C.T., Sangiovanni-Vincentelli, A.L., "Logic Minimization Algorithms for VLSI Synthesis", Kluwer Academic Publishers, 1984. [2] Garey, M.R. Johnson, D.S., "Computers and Intractability. A Guide to the Theory of NP-Completeness", W.H. Freeman and Company, San Francisco 1979. [3] Gerace, G.B. et al, "TOPI-A Special-Purpose Computer for Boolean Analysis and Synthesis", IEEE TC, Vol. C-20, pp. 837-842, Aug. 1971. [4] Helliwell, M., Perkowski, M.A., "GAL-based Hardware Accelerator to Find Exact Minimum Solutions for Mixed-Polarity Generalized Reed Muller Forms", will be submitted, 1989. [5] Ho, P.M., "Massively Parallel Processors for Solving Combinational and Sorting Problems", M.Sc. Thesis, Portland State University, Portland, OR, 1989. [6] Johnson, D., "The NP-Completeness Column: An Ongoing Guide". Journal of Algorithms, Academic Press, each issue. [7] Le, H.V.D., Perkowski, M.A., This proceedings. [8] Marin, M.A., "Investigation of the Field of Problems for the Boolean Analyzer", Ph.D. Dissertation, Univ. of California, Los Angeles, 1971. [9] McCall, J.T., Tront, J.G., Gray, F.G., Haralick, R.T., McCormack, W.M., "Parallel Computer Architectures and Problem Solving Strategies for the Consistent Labeling Problem", IEEE TC. Vol. C-34, No. 11, Nov. 1985. [10] Perkowski, M.A., "General Methods of Solving

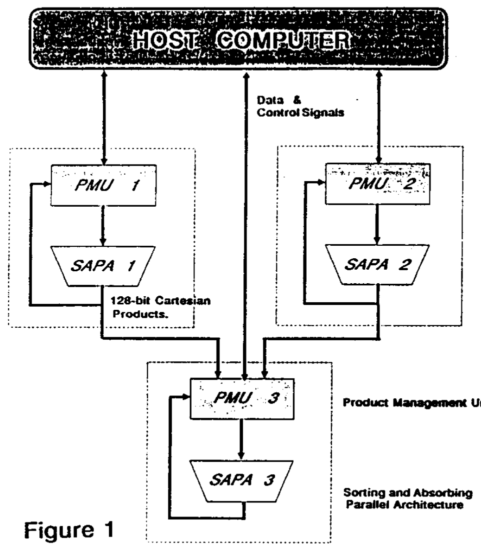


Figure 1

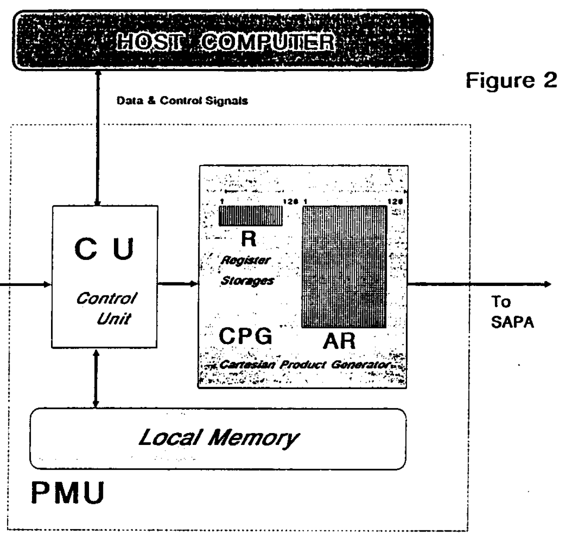
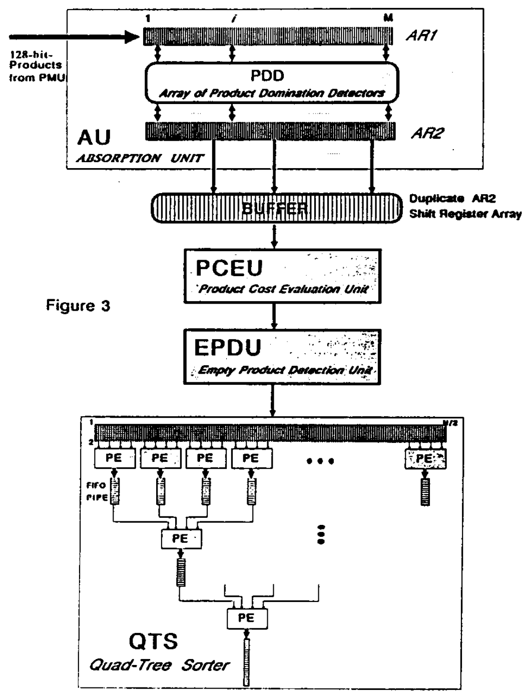
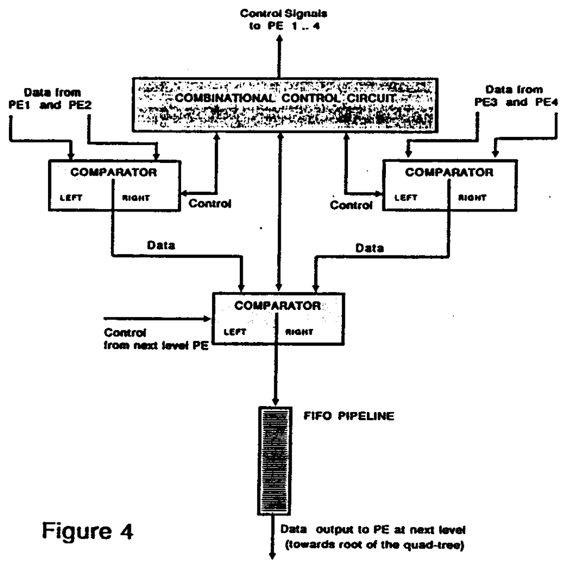


Figure 2



128-bit-Products from PMU
AR1
PDD
Array of Product Domination Detectors
AR2
AU
ABSORPTION UNIT
BUFFER
Duplicate AR2 Shift Register Array
PCEU
Product Cost Evaluation Unit
EPDU
Empty Product Detection Unit
PE
FIFO PIPE
QTS
Quad-Tree Sorter

Figure 3



Control Signals to PE 1 .. 4
Data from PE1 and PE2
COMBINATIONAL CONTROL CIRCUIT
Data from PE3 and PE4
COMPARATOR
LEFT
RIGHT
Control
Data
Control from next level PE
FIFO PIPELINE
Data output to PE at next level (towards root of the quad-tree)

Figure 4

Combinatorial Problems", Publ. of WTU, 1980. [11] Perkowski, M.A., "Systolic Architecture for the Logic Design Machine", Proc. ICCAD'85, pp. 133-135. [12] Perkowski, M.A., Brown, J., "Logic Computers. History and Future", Diades Group Report, PSU, 1989. [13] Perkowski, M.A., Brandenberg, J., "Solving Basic Boolean algebra Problems on a Hypercube Computer", Report PSU, 1989. [14] Perkowski, M.A., Gokul, P., "An Unified Parallel Approach to Basic Computational Algorithms of Multi-Valued Boolean Algebra", Report PSU, 1989. [15] Perkowski, M.A., Wu, P., and K. Pirkl, this proceedings. [16] Petrick, S.R., "On the Minimization of Boolean Functions", Proc. Symp. on Switch. Th., IFIP, Paris, June 1959. [17] Sasao. T., "Input Variable Assignment and Output Phase Optimization of PLA", IEEE TC, Vol. C-33, pp. 879-894, Oct. 1984. [18] Sasao, T, "HART: A Hardware for Logic Minimization and Verification", Proc. ICCD'85, Oct. 7-10, 1985. [19] Shumake, D., "The MOS Boolean Analyzer", M.Sc. Thesis, UCLA, 1971. [20] Svoboda, A., "Boolean Analyzer", Information Processing 68, Amsterdam, North-Holland, 1969. [21] Svoboda A., "Parallel Processing in Boolean Algebra", IEEE TC, Vol. C-22, No. 9, pp. 848-851, Sept 1973. [22] Ulug, M.E., Bowen, B.A., "A Unified Theory of the Algebraic Topological Methods for the Synthesis of Switching Systems", IEEE TC, pp. 255-267, March 1974. [23] [Wah 84] Wah, B.W., Ma, Y.W.E., "MANIP - A Multicomputer Architecture for Solving Combinatorial Extremum-Search Problems", IEEE TC, Vol. C-33, No. 5, pp. 377 - 390, May 1984.

(217) 333-4789 Further information can be obtained from Dr. W. Kenneth Jenkins.