Implementation of Adaptive Logic Networks on an FPGA board

Juan P. Oliver [a], André Fonseca de Oliveira [a], Julio Pérez Acle [a],
Roberto J. de la Vega [b], Rafael Canetti [a]

[a] Instituto de Ingeniería Eléctrica, Facultad de Ingeniería, Universidad de la República, Montevideo - Uruguay
[b] Grupo Adqdat, Facultad de Ingeniería, Universidad Nacional del Centro, Olavarría - Bs. As. - Argentina

ABSTRACT

This work is part of a project that studies the implementation of neural network algorithms in reconfigurable hardware as a way to obtain a high performance neural processor. The results for Adaptive Logic Network (ALN) type binary networks with and without learning in hardware are presented.

The designs were made on a hardware platform consisting of a PC compatible as the host computer and an ALTERA RIPP10 reconfigurable board with nine FLEX8K FPGAs and 512KB RAM. The different designs were run on the same hardware platform, taking advantage of its configurability.

A software tool was developed to automatically convert the ALN network description resulting from the training process with the ATREE 2.7 for Windows software package into a hardware description file. This approach enables the easy generation of the hardware necessary to evaluate the very large combinatorial functions that results in an ALN.

In an on-board learning version, an ALN basic node was designed optimizing it in the amount of cells per node used. Several nodes connected in a binary tree structure for each output bit, together with a control block, form the ALN network. The total amount of logic available on-board in the used platform limits the maximum size of the networks from a small to medium range.

The performance was studied in pattern recognition applications. The results are compared with the software simulation of ALN networks.

Keywords: FPGAs, reconfigurable hardware, artificial neural networks, Adaptive Logical Networks (ALN).

1. INTRODUCTION

The use of programmable or reconfigurable logic to accelerate algorithms of diverse type has been used with success for some years already [1, 2]. The approach in this work is to use a reconfigurable logic board working together with the CPU of a personal computer. The reconfigurable logic carries out the heaviest part in a certain algorithm, thus speeding up its execution.

Further author information -
J.P.O.: E-mail: jpo@iie.edu.uy; WWW: http://www.iie.edu.uy/~jpo
A.F.O.: E-mail: andre@iie.edu.uy; WWW: http://www.iie.edu.uy/~andre
J.P.A.: E-mail: julio@iie.edu.uy; WWW: http://www.iie.edu.uy/~julio
R.J.V.: E-mail: rjdlv@fio.unicen.edu.ar
R.C.: E-mail: canetti@iie.edu.uy; WWW: http://www.iie.edu.uy/~canetti

To be able to increase the use of this technology to potential users' applications it is necessary to narrow the gap between development tools designed for specialists in the area of digital design and the final applications.

In this sense two big tendencies appear: the development of high level languages and compilers able to translate and to transfer automatically certain algorithms to reconfigurable hardware, and the construction of reusable module libraries that could be invoked by the applications. The last approach was the one followed in this work, trying to simplify the use of programmable logic by neural network users.

The focus of this work is the development of hardware libraries of Adaptive Logic Networks (ALN) from the final user's point of view. In a library of C functions some are substituted by their hardware version and the results obtained are compared.

This paper is organized as follows: some comments about neural networks and their hardware implementation are given in Section 2. Section 3 gives an introduction to Adaptive Logic Network (ALN). The hardware platform used is described in Section 4. Section 5 shows an example with a previously trained ALN for the inverted pendulum control problem; and the on-board training ALN implementation is presented in Section 6 with an example of pattern recognition. Finally, conclusions are presented in Section 7.

2. ARTIFICIAL NEURONAL NETWORKS

The Artificial Neural Networks (ANNs) have multiple applications in the fields of control and pattern recognition [3, 4, 5, 6, 7]. In these fields they have had a growing interest as an alternative and feasible tool to solve an enormous quantity of engineering problems, especially those where the conventional procedures have severe limitations. Such is the case, when making decisions in situations difficult to quantify, and therefore not very well defined analytically, is required. This fact is due to the generalization property of neural networks. That is, the ability to give well behaved answers based on previous learning with a representative pattern set, even when the current input pattern was not previously presented.

The software realization of ANN demands a heavy process load to the CPU, because it is implemented in sequential form (inherent of the general purpose computer architecture where the program is running). This fact limits its use to off line applications or real time applications of very slow processes. Another limitation is that certain neural network adaptation algorithms (or training) require extremely high computation time when the dimension of the network to be adapted is relatively big. Although the trained network could be used in a fast way, the necessary time for its training is so high that it makes its use impractical.

Although the use of neural networks as a tool for the solution of real problems of engineering (and other areas) is an interesting alternative, the reasons exposed previously show a clear limitation for their use by means of sequential programs. This makes it attractive to have a system that allows to implement neural networks in hardware, thus allowing to exploit the potentiality of parallel calculation.

In the last years, the academy as well as the industry have worked uninterruptedly on implementing nets in hardware. Good references on the existent works can be found in C. Lindsey's articles and T. Lindblad [7], J. N. H. Heemserk [9] , J. A. Hegt [10] and I. Paolo and G. Kuhn [11].

This work is part of a project that studies the implementation of neural network algorithms in reconfigurable hardware as a way to obtain a high performance neural processor. The Adaptive Logic Network described in the next section was one of the studied networks. It is a digital net that after training can be represented as a collection of logical gates.

3. ADAPTIVE LOGIC NETWORK

An ALN (Adaptive Logic Network) can be described as a set of binary trees with digital inputs (bits). The bits of the input pattern are complemented forming a new pattern with double size. Every input bit has random connections with tree inputs, as many times as possible until the complete connection of the input layer (leaves). This type of trees can carry out any type

of boolean function, being robust for small variations of the input patterns [12, 13, 14]. The probability of having a successful adaptation increases with the ratio between the quantity of leaves and the quantity of bits of the input pattern.

When the input patterns are represented in floating point (real number), these should be coded to binary values for their use with the ALN. Every node of each tree receives two logical inputs and it produces a logical output that can be one of the following functions: AND, OR, LEFT or RIGHT. Each tree gives an output that corresponds to the output of its root node. All output bits form the coded output, that need to be processed by the decoder in order to produce the final output of the network. The adaptation mechanism of these networks change the function carried out by each node: they can adapt to produce a tree that verifies the logical function corresponding to the pattern's input/output relationship. Figure 1 shows a basic outline of this network.

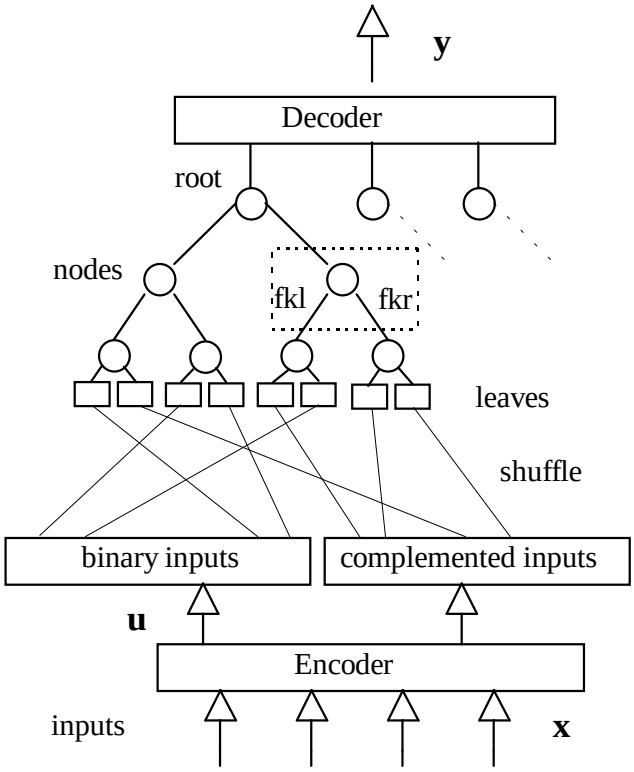

Fig. 1: Adaptive Logic Network

Each node is a sequential circuit that maintains its internal state in two binary counters with saturation. The function performed by the node (AND, OR, LEFT, RIGHT) depends on the value of these counters.

The adaptation is based on the notion of responsibility, that is, the dependence of the network's output on the current function in each node [15]. Each node, besides the inputs *fkl* and *fkr* incoming from the lower layer, receives from the parent node (higher layer) a responsibility input *s*.

The learning is carried out presenting an input pattern and comparing the output obtained with the desired output. Every node computes the responsibility of its children (*skl* and *skr*) as a function of its inputs *fkl* and *fkr*, of the desired output *h*, of its current internal state and of its own responsibility input *sk*. The root node is always responsible. If the responsibility input of a node is a logical one (true), this node modifies its internal state (counters).

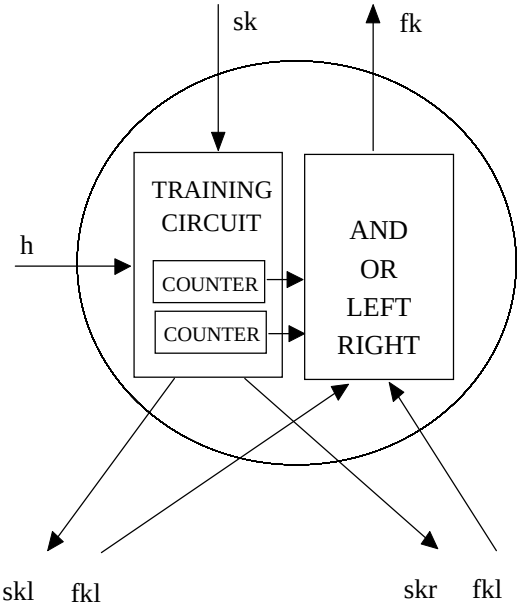

Fig. 2 ALN Node

Adaptation begins once the evaluation of the output corresponding to the pattern presented at the input has been completed. In the first step the root of each tree change its counters and evaluates the responsibilities of his two children nodes (second layer). The process continues for every layer until the base of the tree is reached. In some cases, a final comparison between the output of the tree with the desired output is performed, and a second adaptation is carried out.

Several algorithms exist for the determination of the responsibilities and for how to move on the counters [15]. The algorithm used in this work is the same used in the library Atree version 2.7 for Windows. In this application, Atree, a C library developed at the University of Alberta, Canada that simulates this type of nets was used (the versions used were Atree 2.0 for Unix and Atree 2.7 for Windows [16]).

From the point of view of their implementation it is important to consider that if the learning is carried out off-line, the final size of the trees can significantly decrease, pruning the unused descendent sub-trees in each Left or Right node.

4. HARDWARE PLATFORM

The designs were made on a hardware platform consisting of a PC compatible as the host computer and an ALTERA RIPP10 reconfigurable board with nine FLEX8K FPGAs and 512KB RAM.

The Reconfigurable Interconnect Peripheral Processor (RIPP10) board is provided by the Altera's Programmable Hardware Development Program [17].

The RIPP10 board has a highly interconnected array of 8 FLEX81188 FPGA chips with a total of 8064 Logic Cells. Each chip is connected with four of its neighbors through 32 bit private buses (the "colored buses"). A 36 bit global bus provides broadcast access from a FLEX8452 chip that makes the ISA bus interface. There is also a 4 bit fast bus reserved for clocking.

There are four sockets for RAM chips. Each RAM is accessed from a pair of 81188 chips. The global bus can also be accessed from a buffered 40 pin external interface controlled by the 8452 interface chip. The RAM and the external interface were not used in the work presented here.

Both the array and the interface chips are programmable through the ISA bus. The "bootstrap" process starts loading the interface chip with a circuit that allows loading the array chips. Following this, the array chips are programmed one at a time through the global bus. Finally, if needed, a new design is loaded on the interface chip for the application.

The RIPP10 board presents several advantages and disadvantages from this project's point of view. The RIPP10 provides a very flexible platform that allowed us to easily try different solutions for a given problem. The amount of available logic cells is very large and the board has a high number of interconnection paths between chips. The simplicity of the ISA bus is an advantage while prototyping.

On the other hand this platform has some drawbacks if compared with newer boards. There is no way to partially reconfigure the array. The only way is to clear all the chips at once and then load the eight new circuits. This is a slow process that severely limits the ability to time multiplexing different circuits on the RIPP10. The speed and bandwidth of the ISA bus nowadays has been improved by the PCI interface.

5. BUILDING HARDWARE FROM THE PREVIOUSLY ALN TRAINED SYSTEM

As it has been seen previously a trained ALN is a binary tree where each node carries out one of the following logical functions: AND, OR, LEFT or RIGHT. That is, the necessary circuit to implement a trained ALN tree is a combinatorial function. For real problems this combinatorial function has a great quantity of nodes. When using ALN for real applications the employment of trees of several thousands of nodes is common, although as it was mentioned (Section 3) the number of inputs to the system is always much smaller that the number of leaves. In the application chosen as example the number of leaves is 2048, which implies a total of 2047 nodes organized in 11 layers, and the amount of inputs of the system is 60.

The network product of the training process is stored in text files with the format used by Atree that consists of the functions of each tree represented in postfix notation. The characters ' &', '—', 'L', 'R' represent the node functions AND, OR, LEFT, RIGHT respectively. Leaves are stored as a number, representing the bit index, preceded by a '!' to denote negation when needed.

A tool called "AtreeToHardware" (A2H) was developed. It automatically converts the files of the trees generated by Atree to a hardware description language. A2H, besides carrying out the process of translation, simplifies the network since the

logical functions that it generates don't contain LEFT or RIGHT type nodes, that is, the tree is only made up of AND and OR nodes.

The language used was Altera Hardware Description Language (AHDL), but A2H can be adapted easily to other hardware description languages. The AHDL sources generated with A2H are compiled later with Max+Plus II. This generates a logical function for each tree of the network. The Max+Plus II carries out a minimization of these logical functions, therefore it is possible to map large networks in the chips of the RIPP10.

The developed software is generic, and it allows to automate the translation to hardware of this type of trained networks. This facilitates the final design of applications, reducing the time needed for the hardware system development of a specific application.

In this phase of the project the design is not completely automatic, therefore there are some relatively simple manual tasks, that consists of adapting sizes of parametrized functions previously developed, and then making the final compilations for each array chip.

A C function which is the user's interface to the whole hardware network implementation was developed. This function carries out the evaluation of the group of trees, and its interface is identical to the one included in the Atree library (atree_eval).

5.1 A control application example

An ALN trained to control an inverted pendulum, result of a previous work already carried out in the Electrical Department of the Universidad de la República [18, 19] was chosen to be implemented in hardware. The inputs of the control system are x, θ, dx/dt, $d\theta/dt$, and the output is the current of the DC motor that moves the cart of the pendulum.

The 4 input variables are real, coded in 15 bits each one, conforming a total of 60 input bits.

The control output is made up of a real variable coded in 6 bits. Each one of these 6 output bits is calculated with a redundancy factor of three (3 votes, majority decision), thus needing 18 binary trees.

Each one of these trees has 2048 leaves, being their connections with the input bits and their inverted values at random.

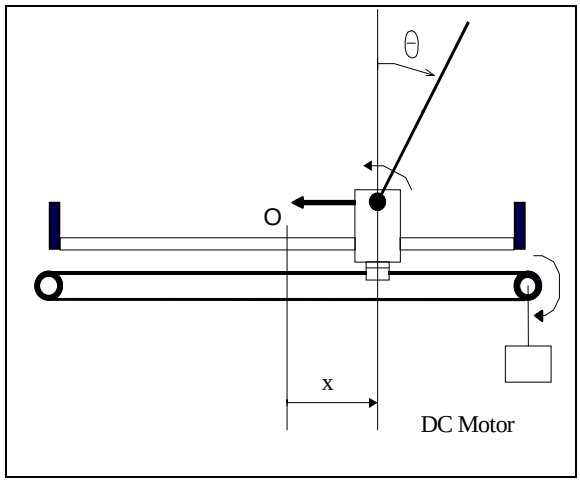

Fig. 3 Inverted Pendulum

This network was previously trained using the Atree 2.7 library for Windows. The input patterns used were generated by a feedback state controller designed earlier for the given system.

The files, product of this training, were processed with A2H and later compiled with Max+Plus II. The three trees corresponding to the majority decision of an output were located in the same chip of the array. In this way, 6 chips of the array were used to carry out the 18 trees.

To complete the design some more modules are added:

- A logical function that takes the majority decision between the 3 votes was built to generate one output bit per chip.
- The shift registers to store the inputs are adapted to the 60 input bits.

Finally for the elected application, in which 18 trees are programmed with 2048 leaves each one, the chips usage is shown in the following table:

CHIP	LCELLS	% used
u1	541	53 %
u2	576	57 %
u3	539	53 %
u4	542	53 %
u5	479	47 %
u6	509	50 %
u7	not used	0 %
u8	not used	0 %

Table 1: Number of Logic Cells

As it can be appreciated in the table, the chips of the array (EPF81188) have about 45% of free cells.

System inputs are written in successive 16 bit transfers to the ISA port. The 6 output bits, generated by the trees, are read simultaneously in a single input transfer. In both cases data passes through the board's global bus and the interface chip.

5.2 Results

To perform a speed comparison, three versions of a program to evaluate 4096 patterns were compiled with different evaluation functions. All functions have the same C interface. One of them makes the evaluation in hardware and the other two are from the original Atree code.

The execution times listed in Table 2 are from a final application point of view, that is, they include the whole program loop to evaluate all the input patterns. They do not include initialization overheads like the configuration of the FPGA chips in the hardware version and the data structure creation and initialization in the software only versions.

Computer	atree_eval() Time[ms]	atree_fast_eval() Time[ms]	atree_ripp_eval() Time[ms]
Pentium 200 MHz	10934	3626	27
Pentium 90 MHz	23077	7692	
486DX2 66 MHz	59286	17473	
386SX 25MHz	298407	86264	192

Table 2: Speed Comparison

atree_eval() - software evaluation of one tree stored in the standard tree structure in RAM.
atree_fast_eval() - same as atree_eval, except that works over a preprocessed tree that accelerates software execution
atree_ripp_eval() - tree evaluation using hardware (RIPP10)

As can be observed in Table 2, the hardware version evaluation time decreases significantly between a 386SX 25MHz and a Pentium 200MHz, although the RIPP10 clock frequency is the same (ISA bus clock). This indicates that the main part of the evaluation time is spent by the software overhead that speeds up in the faster PC.

In the hardware evaluation version, a lower bound to the execution time can be derived from the I/O interface limitations. A total of five 16 bit ISA bus cycles (4 WR and 1 RD cycle) are needed for each pattern evaluation. Each 16 bit ISA I/O cycles takes three 8.33MHz bus clock periods making a total of 1800 ns per pattern. The lower limit for the evaluation of 4096 patterns is then about 7.4ms. As can be seen, although the hardware evaluation version running on a Pentium 200 PC with the RIPP10 board still has a considerable overhead, it is getting closer to the theoretical limit.

Simulating the circuit with Max+Plus Timing Analysis tool, a minimum cycle time of 300ns to evaluate a pattern is derived.

This cycle includes the load of the 4 input words through the global bus, the combinatorial circuit delay and the output transfer through the global bus. This shows that the throughput is bounded by the ISA bus interface.

6. ON-BOARD LEARNING ALN

A library of parameterized modules was developed to construct Adaptive Logic Networks with all the logic necessary for on-board learning.

The basic module of this library is a single ALN node (see Fig. 2), this node includes the training circuit, two counters that give the state of the node, and the logic combinatorial function of the node: AND, OR, LEFT, RIGHT (selected according to the state). The register width in the counters can be determined during compilation by a parameter value. The node is a synchronous circuit with an input that enables the training in one clock period.

The tree module interconnects nodes in a binary tree structure. The number of layers in the tree, (and as a result the number of leaves and binary inputs as well) is parameterized. To load the initial value of each node's counters and to obtain the final values after training, the counters are arranged as a shift register connecting all the nodes in the tree "in-order" (left subtree first, then the node and right subtree last). This allows to load and inspect the binary tree in a natural way with recursive functions, greatly simplifying the software.

System inputs are broadcasted to the array chips one word at a time. Each input at a tree leaf is driven by one system input, either directly or inverted. A configurable shuffle module was developed so that the connection order of the system inputs to the tree inputs can be configured during initialization from the host. The configurable shuffle consists of a multiplexer for each leaf, to select which system input is connected to the tree input, and to invert the signal if it is needed. When the number of system inputs is high the configurable shuffle can become too expensive in logic cells. So if the design is limited by the amount of logic cells available, the shuffle module must be implemented as a fixed interconnection module determined at compilation time.

The control machine module generates control pulses to trigger the training actions in each layer, one at a time starting with the upper layer as needed by the ALN training algorithm (see Section 3). The module is parameterized in the number of layers.

The logic cell count for the modules is summarized in the table below for a counter of 6 bits width, this counter width is used by the software package. One array chip can comfortably hold a 4 layer tree with the necessary I/O registers.

A 7 layer / 128 leaves tree was implemented using all the chips in the array to test the library. This tree was constructed with one 4 layer tree in each array chip for the lower layers and additional nodes manually mapped, distributed into the array chips to form the upper layers.

The control machine was mapped in the interface chip. The control pulses generated to trigger training in each layer, are broadcasted through the global bus to the nodes in the array chips.

To train the network, each pattern is loaded in the array with 16 bit transfers from the ISA bus, passing through the interface chip and the global bus. Subsequently the desired output *h* is loaded. This transfer starts the control machine in the interface chip. The training process for this pattern takes 1 or 2 passes generating one clock period pulses to enable one layer at a time. This process consumes at most 2 x (number of layers) clock periods during which each node state is adjusted according to the ALN training algorithm.

The output of the tree can then be read from the host to compare it with the desired output in order to collect statistics to decide when the network is trained enough and stop the whole training.

The 7 layer tree was used in a sample pattern recognition application. The network was trained to recognize capital letters represented in a 5x5 black and white pixel matrix. One network can be trained to evaluate if a given 5x5 matrix represents or not a given letter. Five networks were trained for each of the vowel capital letters.

The logic cell count for several designs is resumed in Table 3. Cell count is highly dependent on the logic synthesis style chosen in Max+Plus. In the basic ALN tree node, for example, cell count varies between 33 to 56 LCs. Even though the FAST synthesis style generates smaller circuits, the resulting designs are harder to route and a four layer tree could not be fitted in a chip of the array.

Logic Synthesis Style	Module	# Logic Cells
FAST	node	33
NORMAL	node	56
NORMAL	4 layer tree (15 nodes)	817
NORMAL	5 layer tree (31 nodes)	1539

Table 3: Number of Logic Cells

As in the already trained ALN case seen in the previous section, a new version of some functions in the Atree library was written preserving the function interfaces. A program that trains the network with 120 patterns, presenting them 1000 times to the network (1000 epochs in Atree jargon) was compiled in the two versions. The resulting execution times are presented in Table 4.

Computer	atree_train() Time[s]	atree_ripp_train() Time[s]
Pentium 200 MHz	26.5	0.93
Pentium 90 MHz	58.5	
386SX 25MHz	817.0	10.0

Table 4: Training Speed Comparison

A total of four ISA I/O cycles are needed for each pattern trained: two WR cycles for the 25 bit inputs, one WR cycle for the desired output and one RD cycle to read the tree output. So, the lower bound for a pattern training time is 12 ISA bus clock periods or about 1.5μs, and the bound for the 1000 epochs example is 0.18 seconds.

7. CONCLUSIONS

A pre-trained and a learning capable hardware version of an Adaptive Logic Network were implemented in a reconfigurable logic board attached to a standard PC host. Both versions were tested with standard control and pattern recognition objectives.

From the point of view of the final application, we can conclude that the algorithms implemented in hardware exactly substitute C functions. Therefore, for a user accustomed to working with the Atree software, the usage of the hardware library developed is almost transparent, being this one of the main objectives of this work.

Analyzing the speed up results, it is concluded that the hardware implementation is highly convenient, achieving orders of magnitude of improvement.

The RIPP10 board showed to be adequate for the hardware implementation of big size trained ALN, which are applicable to real time control or more generically to evaluate a great amount of patterns for different kinds of applications. In this case the training of the network must be done offline but it can be applied to the control of processes in which slow adaptability is required as well.

As for the implementation of ALN with on-board adaptation capacity, the chosen architecture even though adequate from the

speed point of view, is very limited by the size of the trees that can be implemented. Is possible to adapt networks with a very small number of inputs, that is, only in very simple applications.

8. ACKNOWLEDGEMENTS

This work was supported by the Uruguayan Scientific and Technical Research National Bureau (CONICYT) under the IDB-CONICYT Project Nº 355 “Neural processor based on Programmable Logic Devices”.

The RIPP10 boards were provided at no cost by the Programmable Hardware Development Program at Altera (special thanks to Stephen Smith).

9. REFERENCES

1. S. Guccione, *Programming Fine-grained Reconfigurable Architectures*, Ph. D. Thesis, University of Texas at Austin, 1995.
2. J. Cloutier, E. Cosatto, S. Pigeon, F. R. Boyer and P. Y. Simard , “VIP: An FPGA-based Processor for Image Processing and Neural Networks”, MicroNeuro ‘96, Lausanne, Switzerland, 1996.
3. Several authors, *Artificial Neural Networks: Paradigms, Aplications, and Hardware Implementations*, edited by Edgar Sánchez-Sinencio and Clifford Lau, IEEE Press, New York, 1992.
4. R. Hecht-Nielsen, *Neurocomputing*, Addison-Wesley, Reading, Massachusetts, 1991.
5. S. Haykin, *Neural Networks*, Macmillan, Englewood Cliffs, NJ, 1994.
6. A. G. Supynuk and W. W. Armstrong, “Adaptive Logic Networks and Robot Control”, Proc. Vision Interface Conference '92, also called AI/VI/GI '92, pp. 181 - 186, Vancouver B. C., 1992.
7. W. Armstrong and J. Gecsei, "Architecture of a Tree-based Image Processor", *12th Asilomar Conf. on Circuits, Systems and Computers*, pp. 345-349, Pacific Grove, 1978.
8. C. S. Lindsey and T. Lindblad, “Review of Hardware Neural Networks: A User’s Perspective”, Third Workshop on Neural Networks: From Biology to High Energy Physics, Isola d’Elba, Italy, 1994.
9. J. N. H. Heemskerk, “Overview of Neural Hardware”, Chapter 3, Ph. D. Thesis *Neurocomputers for Brain-Style Processing. Design, Implementation and Application*, Leiden University, The Netherlands, 1995. (ftp://ftp.mrc-apu.cam.ac.uk/pub/nn).
10. J.A. Hegt, “Hardware Implementations of Neural Networks”, *Measurement and Artificial Neural Networks*, Proceedings 'Themadag van de Werkgemeenschap Meten', Utrecht, 1993, (ftp://ftp.tue.nl/pub/neural/hardware_general.ps.gz).
11. I. Paolo. and G. Kuhn, “Digital Systems for Neural Networks”, *Digital Signal Processing Technology*, P. Papamichalis and R. Kerwin, editors, Vol. CR57 of *Critical Reviews Series*, pp. 314-345, SPIE Optical Engineering, 1995.
12. W. W. Armstrong and G. Godbout, "Properties of Binary Trees of Flexible Elements Useful in Pattern Recognition", *IEEE 1975 International Conf. on Cybernetics and Society*, pp. 447-449, San Francisco, 1975.
13. W. W. Armstrong, A. Dwelly, J.Liang, D. Lin and Scott Reynolds, “Learning and Generalization in Adaptive Logic Networks”, *Artificial Neural Networks, Proc. of the 1991 Int. Conf. on Artificial Neural Networks (ICANN-91)*, T. Kohonen, K. Makisara, O. Simula, J. Kangas, eds, pp.1173-1176, North Holland,.Espoo, Finland, 1991.
14. G. V. Bochmann and W. Armstrong, “Properties of Boolean Functions with a Tree Decomposition”, BIT, Vol. 13, pp. 1-13, 1974.
15. W. Armstrong and J. Gecsei, "Adaptation Algorithms for Binary Tree Networks", IEEE Trans. on Systems, Man and Cybernetics, Vol. 9, pp. 276-285, 1979.

16. William Armstrong Home Page. http://www.cs.ualberta.ca/~arms
17. Altera's Programmable Hardware Development Program Web Page. http://www.altera.com/html/programs/phd.html
18. E. Ferreira, E. Planchón and C. Trochón , "Neural Network Controller for the Inverted Pendulum". *XIII Simposio Nacional de Control Automático*, AADECA'92, Buenos Aires, 1992.
19. Ferreira, and A. Fonseca de Oliveira, "Una Aplicación de Redes Binarias Adaptivas (ALN) en Control", *XIV Simposio Nacional de Control Automático*, AADECA'94, Buenos Aires, 1994.