Chapter 1

FROM CONFIGURABLE CIRCUITS TO BIO-INSPIRED SYSTEMS

Moshe Sipper, Eduardo Sanchez,
Jacques-Olivier Haenni, Jean-Luc Beuchat,
André Stauffer, and Andrés Pérez-Uribe
Logic Systems Laboratory, Swiss Federal Institute of Technology,
IN-Ecublens, CH-1015 Lausanne, Switzerland.[*]
e-mail: {name.surname}@epfl.ch, web: lslwww.epfl.ch.

Abstract Field-programmable gate arrays (FPGAs) are large, fast integrated circuits—that can be modified, or configured, almost at any point by the end user. Within the domain of configurable computing we distinguish between two modes of configurability: *static*—where the configurable processor's configuration string is loaded once at the outset, after which it does not change during execution of the task at hand, and *dynamic*—where the processor's configuration may change at any moment. This chapter describes six applications in the domain of configurable computing, considering both static and dynamic systems, including: SPYDER (a reconfigurable processor development system), RENCO (a reconfigurable network computer), an FPGA-based backpropagation neural network, Firefly (an evolving machine), BioWatch (a self-repairing watch), and FAST (a neural network with a flexible, adaptable-size topology). Moreover, we argue that the rise of configurable computing requires a fundamental change in the engineering curriculum, toward which end we present the LABOMAT board, developed for use by students in hardware design courses. While static configurability mainly aims at attaining the classical computing goal of improving performance, dynamic configurability might bring about an entirely new breed of hardware devices—ones that are able to adapt within dynamic environments.

[*]This work was supported in part by Grant 2000-049349.96 from the Swiss National Science Foundation and by a grant from the Werner Steiger Foundation.

1. INTRODUCTION

When one sets about to implement a certain computational task then obtaining the highest performance (speed) is unarguably achieved by constructing a specialized machine, i.e., hardware. Indeed, this possibility exists, e.g., in the form of application-specific integrated circuits (ASICs) Smith, 1997; however, the price per application as well as the turnaround time (from design to actual operation) are both quite prohibitive. Except for a small number of specialized niches, the computing industry has, by and large, converged onto the so-called general-purpose architecture, trading off the best possible performance in favor of a much lower cost per application and shorter delivery time. The gap between these two paradigms has been narrowing over the past few years with the coming of age of configurable computing.

Field-programmable gate arrays (FPGAs) are large, fast integrated circuits—that can be modified, or configured, almost at any point by the end user Trimberger, 1994; Villasenor and Mangione-Smith, 1997. A primary distinction that this novel technology brings about is that between *programmable* processors and *configurable* ones. The programmable paradigm involves a (general-purpose) processor, able to execute a limited set of operations, known as the *instruction set*. The user's (programmer's) task is that of providing a description of the algorithm to be carried out, using only operations from this limited set. This algorithm need not necessarily be written in the target language (i.e., that of the given processor), since compilation tools may be used; however, ultimately one must be in possession of an assembly-language program, which can be directly executed on the processor in question. The prime advantage of programmability is the relatively short turnaround time, as well as the low cost per application, resulting from the fact that one can (potentially swiftly) reprogram the processor to carry out any other programmable task.

The configurable-computing paradigm can also be regarded as one involving a processor that is able to execute but a given set of operations—however, these are at a much *lower level*. One controls the actual types of the logic devices (such as AND, OR, registers, and flip-flops), the input signals, and the output signals (Figure 1.1). The level at which the end user can control the system's operation, i.e., the design level, is perhaps the fundamental difference between programmable processors and configurable ones.

In both a programmable and a configurable processor the algorithm is ultimately expressed as a string of bits that is stored in memory, with the difference being the manner in which these bits are *interpreted*.

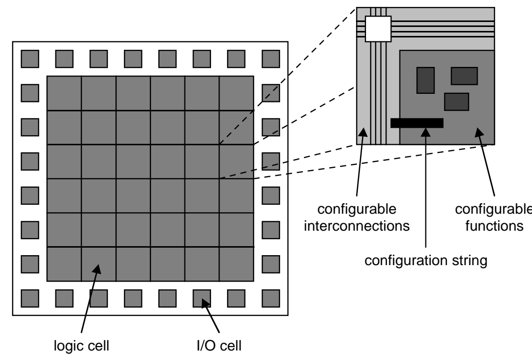

Figure 1.1 A schematic diagram of a field-programmable gate array (FPGA). An FPGA is an array of logic cells placed in an infrastructure of interconnections, which can be programmed at three distinct levels: (1) the function of the logic cells, (2) the interconnections between cells, and (3) the inputs and outputs. All three levels are configured via a configuration bit string that is loaded from an external source, either once or several times.

A programmable processor ceaselessly iterates through a three-phase loop, where an instruction is first *fetched* from memory, after which it is *decoded*, then to be passed on to the final *execute* phase—this latter of which may require several clock cycles; this process is then repeated for the next instruction, and so on. A configurable processor, on the other hand, can be regarded as having but a single, non-iterative fetch phase: the configuration string, fetched from memory, requires no further interpretation, and is directly used to configure the hardware. No further phases nor iterations are needed, as the processor is now configured for the task at hand. The ability to control the hardware in such a direct manner, using a low-level "instruction set," is a double-edged sword: the user is able to access a much wider range of functionality, with the price to be paid being that of a more arduous design task.

So as to avoid any confusion, we shall speak of a *program* when referring to a design (algorithm) within the programmable paradigm, and to a *configuration* or *configuration string* (usually a simple bit sequence) when considering the description of a configurable processor. (In analogy to the term "programmer"—and again so as to avoid any confusion—one might refer to the user of a configurable processor as a *configurer*.)

Within the domain of configurable computing one can distinguish between two types of configuration strings: *static* and *dynamic* Sanchez et al., 1999. A static configuration string, aimed at configuring the processor so as to perform a given function, is loaded (once) at the outset, af-

ter which it does not change during execution of the task at hand. Static configurability has two main objectives: (1) improving performance (i.e., execution speed) for a given function, which essentially results in a rapid coprocessor for the task at hand (e.g., an MPEG coprocessor)—thus, one can consider this an extension of the coprocessor concept; and (2) optimizing the utilization of resources (gates and power consumption) so as to use as much of the chip surface as possible, at each clock cycle. For example, one might divide the task at hand into several sub-tasks, each of which is implemented as a separate configuration. Task execution is achieved by successively loading the sub-task configurations, thus ensuring that at each point the processor is optimized to perform the part of the computation in question

Dynamic configurability involves a configuration string that can change during execution of the task at hand, with the two main objectives being: (1) to adapt to changing (dynamic) specifications (e.g., as with an autonomous robot that is placed in a new environment) as well as to be able to handle incomplete specifications; and (2) to eliminate human design altogether. The first objective involves *partial* design, namely, the configurer designs the system to exhibit a certain general functionality, which is not necessarily the ultimate task to be accomplished—this latter is attained when the system dynamically changes its configuration string, during its operation (rather than at the design phase, as with static systems). Partial design can ultimately lead to the removal of the human configurer from the design cycle, whereupon the system's configuration is carried out dynamically, online. (We note in passing that with the advancement of configurable-computing technology one may eventually be able to configure the processor anew at each clock cycle, producing, in effect, a rapid succession of new machines.)

This chapter describes six projects in the domain of configurable computing, carried out in our lab over the past five years. As can be seen in Table 1.1, we shall consider both static and dynamic systems that exhibit the wide range of characteristics discussed above (the only class that is not described herein is that of dynamic systems that aim at eliminating human design; the work of Thompson Thompson, 1996; Thompson, 1997; Thompson et al., 1996 provides an example belonging to this class). We begin in Section 2. with the description of three static systems: SPYDER (a reconfigurable processor development system), RENCO (a reconfigurable network computer), and an FPGA-based backpropagation neural network. Section 3. presents three dynamic systems: Firefly (an evolving machine), BioWatch (a self-repairing watch), and FAST (a neural network with a flexible, adaptable-size topology). Each system is described by four articles: type, functional description, hardware

Table 1.1 Systems described in this chapter.

Configuration	Objective	System	Section
Static	Improve performance	SPYDER	2.1
		RENCO	2.2
	Optimize resource usage	Backpropagation Neural Network	2.3
	Both of the above	LABOMAT	4.
Dynamic	Adapt to changing or partial specifications	Firefly	3.1
		BioWatch	3.2
		FAST	3.3
	(Eliminate human design)	(Thompson's work)	

description, and performance gains (and—where relevant—a software description as well). In Section 4. we discuss the issue of codesign—basically, the decision of which parts of the application are to be designed as software and which shall be designed directly as hardware. We argue that this requires a fundamental change in the engineering curriculum, and present the LABOMAT board, developed for use by students in hardware design courses. Finally, we present our concluding remarks in Section 5..

2. STATIC SYSTEMS

2.1 SPYDER: A RECONFIGURABLE PROCESSOR DEVELOPMENT SYSTEM

Type (Objective). Static (Improve performance).

Functional description. The main advantage of specialized coprocessors is also one of their weaknesses: they can execute only their intended application. SPYDER (an anagram of the letters of REconfigurable Processor Development SYstem) is a reconfigurable coprocessor that self-adapts to a given application, in a manner which is transparent to the user: the application is written in a high-level language (rather than an assembly program), and the compiler generates the best-adapted hardware description Iseli and Sanchez, 1995.

A processor consists of two parts: a *control unit*—a finite state machine that handles the sequencing of operations of the algorithm being executed, and a *processing unit* or *data path*—the set of memory elements and operators needed to store and process the variables of the algorithm.

The control unit has little influence on the degree of adaptation of the processor to a given algorithm. Indeed, if it is implemented as a microprogrammed machine, its structure is almost fixed: a micromemory (to store the microinstructions) linked to a sequencer (to generate the address of the next microinstruction to be executed) Habib, 1988. On the other hand, the processing unit's architecture is of vital import where the performance of the processor is concerned: the number and the size of the memory elements, the type of available operators, and their interconnection with the memory elements, determine the number of clock cycles needed to realize a certain operation.

Most reconfigurable processors enable the implementation of the two parts of the processor—indeed, they are organized as an array of FPGA circuits, possibly connected to other resources (memories, for example); the configurer (or a compiler) generates the full processor configuration for a given application Arnold et al., 1992; Bertin et al., 1989; DeHon, 1996. Given the minor influence on performance of the control unit's architecture, we took a simpler approach with SPYDER, using a fixed control unit, equivalent to a microprogrammed control unit composed of a sequencer and a very large memory. The microprogram, however, does not interpret a given assembly language, rather, it is the program to be executed (SPYDER can thus be considered a VLIW processor Rau and Fisher, 1993).

The reconfiguration of SPYDER thus takes place in the processing unit, which consists of three FPGA circuits connected to two banks of registers. Each FPGA maintains an independent access to the registers in order to permit parallel processing of the data, and hence the implementation of superscalar architectures. This reconfigurability presents two major limitations: the size of the FPGAs and the number of registers.

The initial objective of the project was to provide transparent hardware reconfiguration: the user would write his program in a high-level language and the compiler would generate both the code to be executed (the contents of the memory of the control unit) and the configuration of the three FPGAs. Given a certain application, the compiler had to automatically determine the optimal set of operators, and their possible concurrent utilization.

Given the complexity of such a compiler, we implemented an intermediate solution: the user determines the operators, and describes them in a high-level language (C++). The compiler then generates the corresponding configuration of the FPGAs (an example program is shown in Figure 1.2). Finally, the user writes the application using the predetermined set of operators. The compiler generates the corresponding

```
#include "x4000.h"
void
count6a(bitvector &out(3))
{
  static bitvector state(3, 0);
  state++;
  if (state == 6)
    state = 0;
  out = state;
}
```

Figure 1.2 Example of a SPYDER operator described in C++: A modulo-6 counter.

code and schedules the operations so as to attain a maximal degree of parallelism.

As with standard coprocessors, SPYDER is connected to a host computer, which handles input/output and runs the development software.

Hardware description. SPYDER was implemented to function as a SPARCstation coprocessor, using a double-Europe board, connected to a SPARC processor by means of a VME bus (Figure 1.3).

The sequencer of the control unit is implemented by means of a Xilinx XC4003 circuit. Its configuration is fixed: 16 different sequences divided into four categories (jump, call subroutine, return of subroutine, and return) are possible. The execution of each instruction takes four clock cycles, but a four-phase pipeline permits the sequencer to generate a new instruction address every clock cycle.

The program memory is separated from the data memory as in Harvard architectures: the instruction memory is 128 bits wide, while the data memory is 16 bits wide. To fully exploit the parallel-processing capabilities, the 128 bits of an instruction directly control all resources of the processor without any intermediate decoding.

The three processing units are implemented using the Xilinx XC4008 circuits. They are fully configurable and are organized in a `load/store` fashion: the data is loaded from the registers and the data memory is accessed only by means of `load` and `store` operations. At every clock cycle, each of the processing units can read two 16-bit data words, and generate two 16-bit results, one per register block. They can also generate one condition bit, which is used by the sequencer, and up to 4-bit addresses to the registers. Finally, the four processing units are

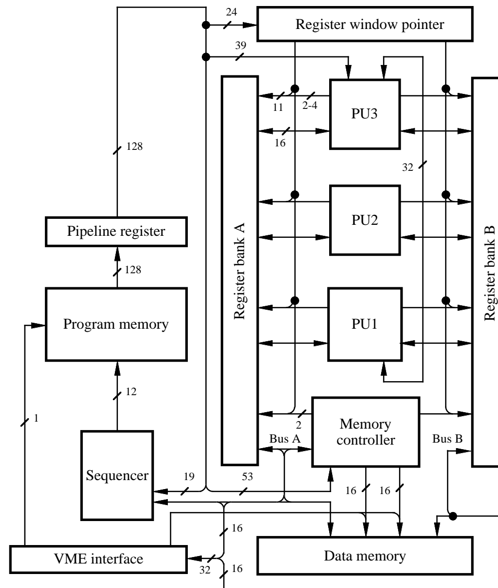

Figure 1.3 The SPYDER architecture.

connected as a ring by means of two 16-bit buses, in order to facilitate pipeline operations.

The operations of the processing units are completely configurable and controlled by 21 bits of the instruction word. The distribution and function of these 21 bits are defined by the user and depend on the configuration of the units.

To facilitate a change of context, the system uses a register window mechanism, similar to that of the SPARC processors Weaver and Germond, 1994. The number of registers per window is also configurable: windows of 4, 8, or 16 registers are possible.

Performance gains. SPYDER runs at only 8 MHz, due to the technology used when the project began, and due to economical reasons. However, the resulting performance of SPYDER on three different applications—a simulation of the Game of Life, and two different image processing algorithms (skeletonization and edge detection)—surpasses several classical architectures.

A SPYDER implementation of Conway's Game of Life Berlekamp et al., 1982 was compared with `xlife`, the most popular software version of this well-known cellular automaton. This application involves

a grid of cells, each one of which can be in a given state at a given moment, which are updated simultaneously in discrete time steps (for details see Berlekamp et al., 1982). Our interest here was to study how fast the grid could be modified, i.e., how many cell states could be updated per second. For a 608x608 matrix of cells, SPYDER—running at 8 MHz—computes the future state of 115 million cells per second, while a microSPARC machine—running at 85 MHz—is only capable of computing the future state of 6.5 million cells per second. The results of the other two applications are delineated in Iseli, 1996; Iseli and Sanchez, 1997.

The performance of SPYDER can be improved by using current-day devices. The communication with the host computer can also be improved: the VME bus was chosen due to its simple implementation and disregarding its low access speed. Nevertheless, the most important enhancements must be done in the software, using new developments in compilation techniques. We hope to see one day a compiler sufficiently powerful to accomplish our initial specifications: a system that automatically determines the optimal hardware implementation and maximal degree of parallelism.

2.2 RENCO: A RECONFIGURABLE NETWORK COMPUTER

Type (Objective). Static (Improve performance).

Functional description. The ability to store an application in various (physical) locations, recently highlighted by the introduction of the *network computer* Slater, 1996, presents many advantages: the vital resources of the computer (mass memories, applications, software libraries, etc.) are exclusively accessible through the network, thus reducing maintenance costs while adding flexibility to the system.

RENCO (REconfigurable Network COmputer) adds the power of reconfiguration to the network computer Villasenor and Mangione-Smith, 1997: a reconfigurable surface is associated with a standard network computer, in such a manner that the user can download from the network not only his or her application, but also the processor configuration able to optimally execute it.

Although for the moment no software manufacturer offers hardware configurations along with the software, we are quite certain of the viability of this approach. As mentioned above, there will soon be compilers able to generate a hardware description given a standard program, and one processor manufacturer (Motorola Report, 1998) has already an-

nounced a processor with a reconfigurable on-chip surface. Currently, RENCO is used for testing new codesign methodologies along with their associated CAD tools, and as a prototyping platform for dedicated processors.

Hardware description. RENCO is composed of two parts (Figure 1.4): a conventional network computer, based on a Motorola MC68EN360 processor Motorola, 1993, and a reconfigurable surface (a cluster of FPGAs connected to their own memories and to the processor bus). The user can design dedicated coprocessors for the 68360, or select them from a specialized library, and dynamically download them through the network when necessary.

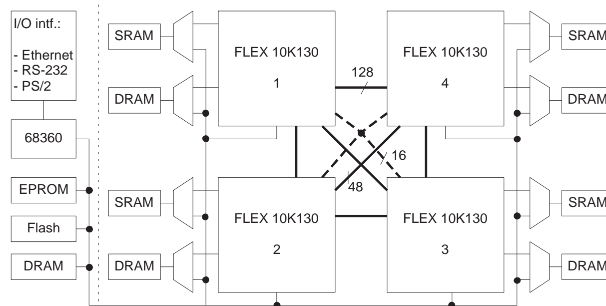

Figure 1.4 RENCO block diagram.

The network computer we have implemented is quite conventional: a microprocessor connected to three types of memory (256k×16b of boot EPROM, 512k×32b of Flash RAM, and up to 16M×32b of DRAM). The 68360 has been chosen for its communication capabilities, for its integrated memory controller, and for the availability of many software tools.

The computer is connected to the network through an Ethernet 10Base-T interface. This communication interface is used at boot time for downloading the operating system, the applications, and the hardware configurations. An RS-232 interface is also available and is used to connect a console to the computer. Several extension connectors allow the user to expand the board features by adding specific extension boards.

The reconfigurable part contains four Altera Flex 10K FPGAs (10K130 or 10K250) Altera, 1997; these large FPGAs contain up to one million programmable logic gates. Since they are connected together, it is possible to split very large designs into up to four parts. The processor bus

is connected to the four FPGAs, which can therefore be accessed as peripherals by the processor and act, e.g., as coprocessors. Each FPGA is connected to its own memories: 512k×8b of SRAM and up to 8M×32b of DRAM. The processor can also access these memories. RENCO is implemented on a 14-layer PCB (Figure 1.5).

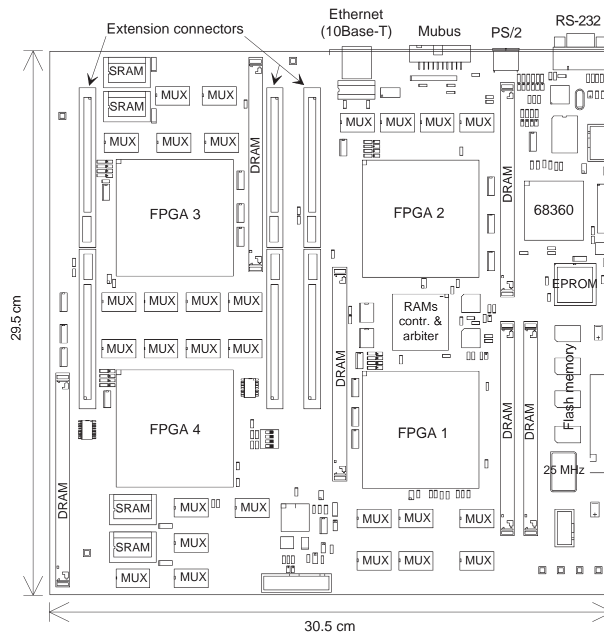

Figure 1.5 RENCO printed circuit board.

Software description. The two parts of RENCO (network computer and reconfigurable area) each require a specific software:

- The network computer requires an operating system, with complete management of the network operations. After examining many possibilities, we chose RTEMS[1] (Real-Time Executive for Multiprocessor Systems). It is a preemptive multi-tasking operating system with rather modest memory requirements. It also contains the drivers for Ethernet and RS-232 and has already been adapted for the 68360 processor. Furthermore, its source code is free and a TCP/IP stack is available.

[1] http://lancelot.gcs.redstone.army.mil/rtems.html

- Many software tools are necessary for the reconfigurable part: a synthesizer, a monitor allowing access to the resources and the configuration loading, a debugger, a user interface, etc. The implementation of all these tools is beyond our reach and we decided to use commercial tools when available (the synthesizer for example) and to concentrate only on the tools specific to our system.

The basic idea is to use Java to develop some of these tools, a choice emanating from our desire to access RENCO from many different platforms through the network. The first step was to implement a Java virtual machine: we chose Kaffe[2] as the source code, since it is freely available and because it has already been ported to the 68000 processor, therefore reducing our development work. In addition to the standard Java application programming interface (API), the user has at his or her disposal a board-specific API that provides classes and methods for accessing the board resources (Figure 1.6). Finally, board-specific code has been written and collected into the Custom Hardware Library (CHL), which includes utility functions for accessing the board resources. As with most reconfigurable systems, the complexity of RENCO's software is much higher than that of the hardware—it is still work in progress.

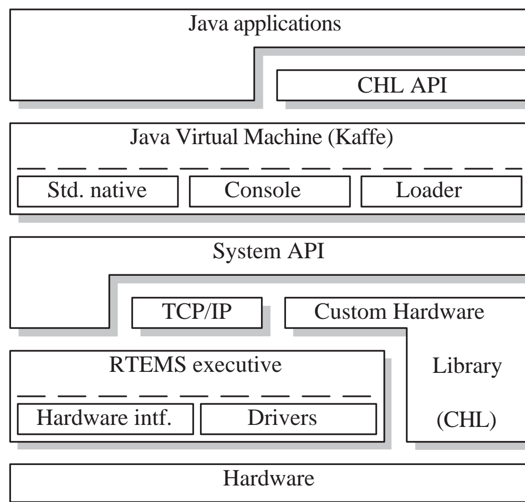

Figure 1.6 RENCO software layers.

[2]http://www.kaffe.org

Performance gains. Our first goal is to test a novel but—in our opinion—very promising idea: considering the hardware architecture as a downloadable resource (in addition to the software). As hardware architecture libraries are currently unavailable, we could not make full-blown evaluations to date and we have not yet proceeded further than the concept validation. Meanwhile, RENCO can also be used for complex logic design prototyping Salcic and Smailagic, 1997. In this context, its large amount of reconfigurable logic and the large memories attached to it represent an important advantage.

2.3 AN FPGA-BASED BACKPROPAGATION NEURAL NETWORK

Type (Objective). Static (Optimize resource usage).

Functional description. Artificial neural networks have been widely used over the past decade for designing adaptive, robust systems, their major advantage stemming from their ability to learn to solve problems from examples (rather than being pre-programmed). These networks are massively parallel, and are composed of non-linear computational elements, often referred to as units or neurons. A so-called activation value is associated with each neuron, and a so-called weight (or synaptic weight) is associated with each connection between two neurons. A neuron's activation depends on the activations of the neurons connected to it and on the interconnection weights. Neurons are often arranged in layers, with input-layer neurons having their activations externally set (i.e., they receive the input, e.g., the image to be recognized).

It is very difficult to precisely define learning; nevertheless, in the artificial neural-network context it is closely related to the adjustment of interconnection weights and topology adaptation Pérez-Uribe, 1998. A *learning algorithm* refers to a procedure in which learning rules are used for adjusting the weights of an artificial neural network, and possibly its topology. Such learning algorithms come in three main flavors: supervised, reinforcement, and unsupervised Rojas, 1996.

Artificial neural networks can solve complex problems such as handwritten pattern recognition and time series prediction. Though software simulations are essential when one sets about to study a new algorithm, they cannot always satisfy real-time demands required by many real-word applications. In order to exploit the inherent parallelism of artificial neural networks, hardware implementation is essential.

Among the many neuroprocessors described in the literature, one can distinguish between two main design philosophies. The first approach

involves the design of a highly parallel computer and a programming language dedicated to neural networks. Many algorithms can be implemented on the same system. Nevertheless, programming such a machine is often arduous Ienne, 1997. The second approach involves the design of a specialized chip for a given algorithm Köllmann et al., 1996, thus avoiding the tedious programming step. The main drawback lies in the need for a different chip for each algorithm.

FPGA circuits offer new paths for implementing neuroprocessors. A learning algorithm can be split into several sequentially executed steps, which are associated with particular FPGA configurations. Such an approach leads to an optimal use of hardware resources. The reconfiguration paradigm also allows the implementation of multiple algorithms on the same hardware. In this subsection, we present the design of a reconfigurable neuroprocessor implementing multilayer perceptrons Hertz et al., 1991 with on-chip training and pruning.

Figure 1.7 depicts a system designed for handwritten digit recognition. We shall use this example to demonstrate some principles of supervised learning algorithms. The network is trained with a set of characters written by different people. After a pre-processing phase (including, e.g., normalization and smoothing) we obtain a gray-level image (M x N pixels) of each character. The M x N gray-level values are stored in an input vector $\underline{\xi}^{\rho}$ to which a class attribute is associated. $\underline{\xi}^{\rho}$ is presented to the neural network which thereupon determines an output. This latter is then compared with the class attribute in order to determine an error signal, which is used by the learning algorithm to adapt the network parameters. This error measure, embodying the difference between the desired (correct) output and the actual output computed by the network, is at the heart of supervised learning techniques. The goal is to minimize this error.

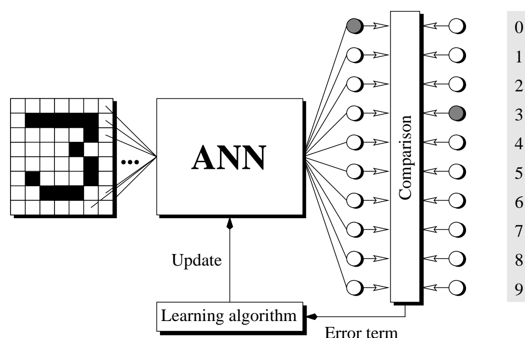

Figure 1.7 Supervised learning principles.

When we train a system by providing examples, it is usually impossible to present it with every possible input pattern (due to the huge size of the pattern space). Therefore, an important issue of training is the capability of the network to generalize, that is, to cope with previously unseen patterns. It has been found that generalization is quite dependent on the network topology. A rule of thumb for obtaining a good generalization is to use the smallest network able to learn the training data Reed, 1993. Training successively smaller networks is, however, a time-consuming approach. Among the efficient processes to determine a good topology, one can cite genetic algorithms, growing methods, and pruning algorithms, the latter of which are used herein.

The pruning algorithms approach consists of training a network that is larger than necessary and deleting superfluous elements (units or connections). These algorithms can be classified into two general categories: sensitivity estimation and penalty-term methods. Algorithms within the first category measure the sensibility of the error to the removal of a connection or a unit, after which elements with the lowest sensibilities are pruned. Methods belonging to the second category suggest new error functions that drive weights to zero during training.

Pruning connections leads sometimes to the situation depicted in Figure 1.8, where some neurons have no more inputs or outputs; such neurons, called *dead units*, can be deleted.

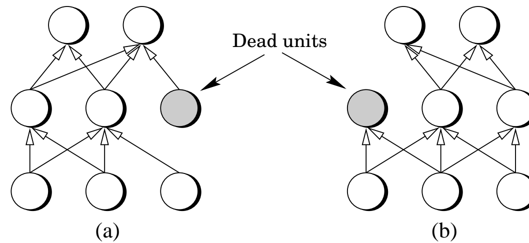

Figure 1.8 Networks with dead units.

Before discussing the hardware architecture, we introduce some notations:

- $w_{n_i m_j}$ is the weight between neuron i in layer n and neuron j in layer m.
- The net input for neuron j in layer m is denoted by $h^{\rho}_{m,j}$.
- $a^{\rho}_{m_j} = \varphi(h^{\rho}_{m,j})$ is the activity (or activation value) of neuron j in layer m; φ denotes the activation function.

Hardware description. A learning algorithm consists of several steps, including: network initialization, forward propagation, error computation, backward propagation, weight update, and pruning. Each step requires specific hardware resources, e.g., network initialization makes use of a random number generator, which is unused in the following steps.

FPGA circuits offer new possibilities for designing hardware neural networks. The learning algorithm is divided into several sequentially executed stages, each of which is associated with an FPGA configuration. A possible decomposition scheme is depicted in Figure 1.9. Note that reconfiguration time is of crucial import—if this process needs more time than computation, such an approach is not appropriate. In order to realize an efficient system one must carefully choose the FPGA family.

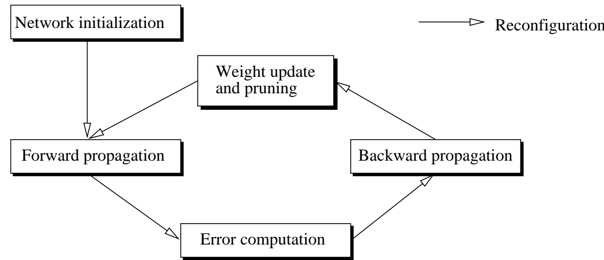

Figure 1.9 A possible decomposition of the backpropagation algorithm.

When designing the hardware architecture of our neural network we first observed that a time-multiplexed interconnection scheme provides a good trade-off between speed and scalability (Figure 1.10). The main idea is to connect all outputs of hidden layer m and all inputs of hidden (or output) layer $m+1$ to a common bus; the same hardware is reused for all layers of the network. The multiplexor allows to provide the network with an input signal or an activation value of a hidden unit. All synaptic weights are stored in a memory associated with the FPGA(s). We will focus herein on forward propagation of a signal (backward propagation obeys the same principles). The first neuron in layer m places its activity $a^{\rho}_{m,1}$ on the bus. All neurons in layer $m+1$ read and multiply it by the appropriate synaptic weight $w_{m_1 m+1_j}$ and finally store the result. Simultaneously, we load the weights from the next layer-m neuron to layer $m+1$. This process is sequentially repeated for every neuron in layer m. Each processing element in layer $m+1$ accumulates the results of the successive multiplications.

Thanks to this interconnection scheme, each neuron is a very simple processing element. Figure 1.11a depicts the architecture used during

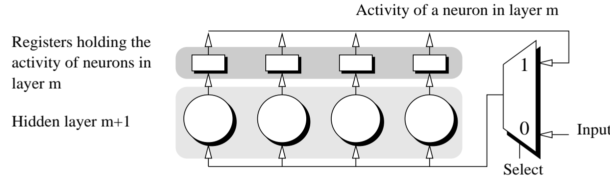

Figure 1.10 The time-multiplexed interconnection scheme.

the forward propagation step. A register stores the weight value involved in the next multiplication. We have associated with each weight a special bit, called $\overline{\text{Pruning}}$, which indicates whether a connection has been pruned (in which case, this bit is set to 0) or not. Combined with a load signal, it enables the accumulation of a multiplication result.

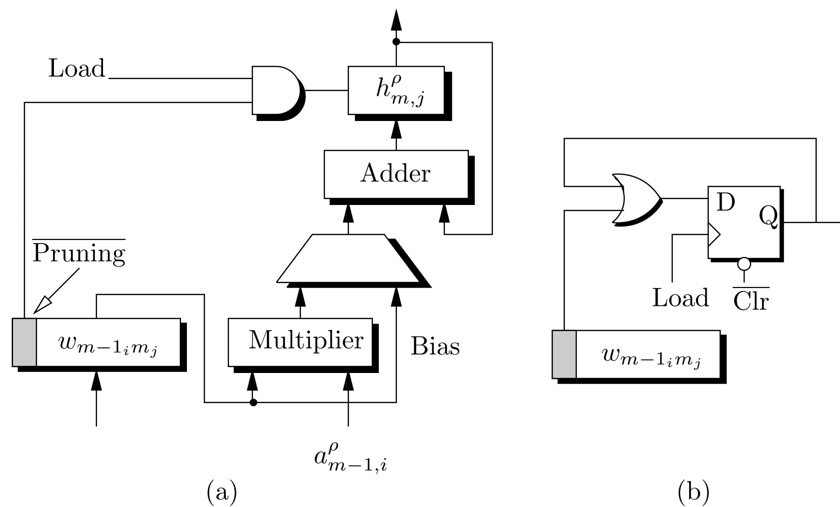

Figure 1.11 (a) Architecture of a neuron. (b) Dead unit detection mechanism.

We now have to provide our neuroprocessor with a means for detecting dead units. The mechanism illustrated in Figure 1.11b solves this problem. Assume that a neuron j in layer m has no more inputs (Figure 1.8a). All $w_{m-1_i m_j}$ coefficients are loaded when a signal is forward-propagated through the network. As the $\overline{\text{Pruning}}$ bits associated with the $w_{m-1_i m_j}$ are set to zero, the flip-flop output remains zero as well. Consider a neuron with no outputs (Figure 1.8b). The backward-propagation process involves all weights $w_{m_j m+1_k}$ whose $\overline{\text{Pruning}}$ bits are equal to zero. Consequently, the detection of such dead units occurs during this step. Once a dead unit has been detected, a signal is sent to a global con-

troller that manages the network topology. As the activities of neurons are sequentially placed on the bus, the deletion of dead units increases the learning speed.

Since FPGAs are not well suited for floating-point computation Villasenor and Mangione-Smith, 1997, we use fixed-point numbers (two's complement) to carry out additions, subtractions, and multiplications. We use a piecewise linear function, whose coefficients are stored in a look-up table, in order to implement the activation function $\varphi(x)$. Relations between $\varphi(x)$ and its derivative permit an easy computation of $\varphi'(x)$ (for example, $\tanh'(x) = 1 - (\tanh(x))^2$).

Performance gains. We have performed a series of experiments to evaluate the efficiency of our limited-precision system. Four learning rules (Backpropagation Widrow and Lehr, 1990, Non-Linear Backpropagation Hertz et al., 1997, Resilient Backpropagation Riedmiller, 1994, and Weighted Error Function Sakaue et al., 1993) and two pruning algorithms (Autoprune Prechelt, 1995 and a penalty-term method by Ishikawa Ishikawa, 1996) have been implemented using the SNNS neural network simulator Zell et al., 1995. A first version of these algorithms executes all arithmetic operations with floating-point numbers, while a second one uses fixed-point numbers.

The training and pruning algorithms were applied to eight problems from the Proben1 data set Prechelt, 1994 and the generalization capabilities were studied. Our experiments have demonstrated that limited-precision algorithms (usually, 3 bits for the integral part and 12 bits for the fractional part) have the same performance as floating-point algorithms.[3]

We now have to complete the hardware implementation on RENCO (Section 2.2) and to evaluate the performance of our neuroprocessor. However, RENCO only allows for the design of a first prototype—in order to attain further improvements specialized hardware will be required. One possible enhancement lies with the partial reconfiguration paradigm: some successive configurations are quite similar—thus, increasing the system's speed might be accomplished by modifying only small parts of the FPGA.

Finally, reconfigurable systems offer some other interesting prospects. The architecture depicted in Figure 1.10 is well suited for the learning process. It would be interesting, though, to increase the degree of parallelism when training is over. Therefore, we plan to design special FPGA

[3]Note that weights sometimes become huge when the network is trained with Resilient Backpropagation Riedmiller, 1994. More bits are then needed for the integral part of numbers.

configurations for the recall process (when training is over, i.e., synaptic weights no longer change, and the network is used to perform its task, e.g., recognizing handwritten characters).

3. DYNAMIC SYSTEMS

3.1 THE FIREFLY MACHINE

Type (Objective). Dynamic (Handle changing and/or incomplete specifications).

Functional description. The idea of applying the biological principle of natural evolution to artificial systems, introduced more than four decades ago, has seen impressive growth in the past few years. Usually grouped under the term *evolutionary algorithms* or *evolutionary computation*, we find the domains of genetic algorithms, evolution strategies, evolutionary programming, and genetic programming Fogel, 1995; Koza, 1992; Michalewicz, 1996. As a generic example of artificial evolution, we consider genetic algorithms.

A genetic algorithm is an iterative procedure that involves a constant-size population of individuals, each one represented by a finite string of symbols, known as the *genome*, encoding a possible solution in a given problem space. This space, referred to as the *search space*, comprises all possible solutions to the problem at hand. The algorithm sets out with an initial population of individuals that is generated at random or heuristically. Every evolutionary step, known as a *generation*, the individuals in the current population are *decoded* and *evaluated* according to some predefined quality criterion, referred to as the *fitness*, or *fitness function*. To form a new population (the next generation), individuals are *selected* according to their fitness, and then transformed via genetically inspired operators, of which the most well known are *crossover* ("mixing" two or more genomes to form novel offspring) and *mutation* (randomly flipping bits in the genomes). Iterating this procedure, the genetic algorithm may eventually find an acceptable solution, i.e., one with high fitness.

Evolutionary algorithms are common nowadays, having been successfully applied to numerous problems from different domains, including optimization, automatic programming, circuit design, machine learning, economics, immune systems, ecology, and population genetics, to mention but a few.

One of the recent uses of evolutionary algorithms is in the burgeoning field of *evolvable hardware* Sanchez and Tomassini, 1996; Sipper et al., 1998; Sipper et al., 1997b, which involves, among others, the use of FP-

GAs as a platform on which evolution takes place. The Firefly machine is one such example; our goal in constructing it was to demonstrate a system in which *all* evolutionary operations (selection, crossover, mutation, and fitness evaluation), are carried out *online*, that is, in hardware Sipper et al., 1997a; Sipper et al., 1997b.

Firefly is based on the cellular automata model (which we briefly encountered in Section 2.1 when describing the Game of Life application)—a discrete dynamical system that performs computations in a distributed fashion on a spatially extended grid. A cellular automaton consists of an array of cells, each of which can be in one of a finite number of possible states, updated synchronously in discrete time steps according to a *local*, *identical* interaction rule Sipper, 1997a; Wolfram, 1994. The *state* of a cell at the next time step is determined by the current states of a surrounding neighborhood of cells. This transition is usually specified in the form of a *rule table*, delineating the cell's next state for each possible neighborhood configuration. The cellular array (grid) is n-dimensional, where $n = 1, 2, 3$ is used in practice. Herein, we consider one-dimensional grids, where each cell can be in one of two states (0 or 1), and has three neighbors (itself, and the cells to its immediate left and right); the rule table thus comprises eight bits since there are eight possible neighborhood configurations. Non-uniform cellular automata have also been considered, where the local update rule need not be identical for all grid cells Sipper, 1997a.

Based on the *cellular programming* evolutionary algorithm of Sipper Sipper, 1997a we implemented an evolving, one-dimensional, non-uniform cellular automaton. Each of the system's 56 binary-state cells contains a genome that represents its rule table. These genomes are initialized at random, thereupon to be subjected to evolution. The system must evolve to resolve a global synchronization task: upon presentation of a random initial configuration of cellular states, the cellular automaton must reach, after a bounded number of time steps, a configuration whereupon the states of the cells oscillate between all 0s and all 1s on successive time steps (this may be compared to a swarm of fireflies that evolves over time to flash on and off in unison). Due to the local connectivity of the system, this global behavior—involving the entire grid—comprises a difficult task. Nonetheless, applying the evolutionary process of Sipper, 1997a, the system evolves (i.e., the genomes change) such that the task is solved Goeke et al., 1997; Sipper et al., 1997a. The machine is depicted in Figure 1.12.

Hardware description. Firefly comprises 56 cells, the architecture of which is shown in Figure 1.13. The binary state of a cell is stored

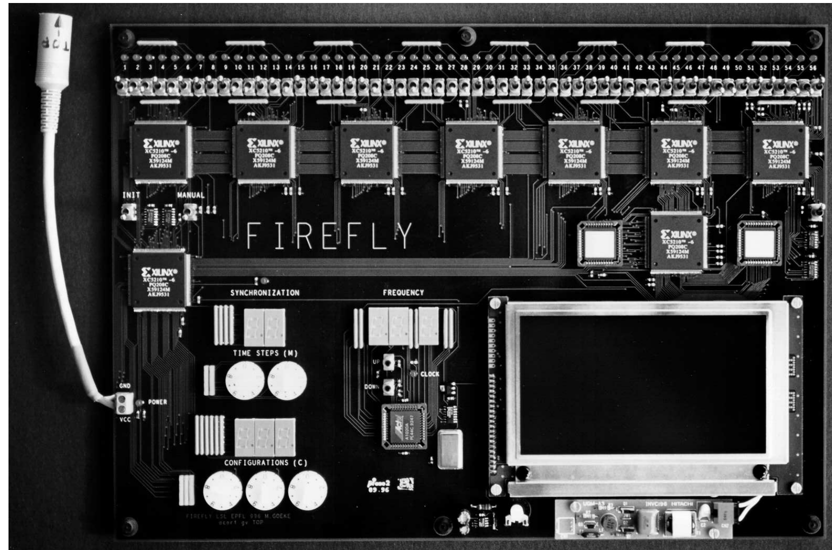

Figure 1.12 The Firefly evolware board. The system is an evolving, one-dimensional, non-uniform cellular automaton. Each of the 56 cells contains a genome that represents its rule table; these genomes are randomly initialized, thereupon to be subjected to evolution. The board contains the following components: (1) LED indicators of cell states (top), (2) switches for manually setting the initial states of cells (top, below LEDs), (3) Xilinx FPGA chips (below switches), (4) display and knobs for controlling two parameters ('time steps' and 'configurations') of the cellular programming algorithm (bottom left), (5) a synchronization indicator (middle left), (6) a clock pulse generator with a manually adjustable frequency from 0.1 Hz to 1 MHz (bottom middle), (7) an LCD display of evolved rule tables and fitness values obtained during evolution (bottom right), and (8) a power-supply cable (extreme left). (Note that this latter is the system's sole external connection.)

in a D-type flip-flop whose next state is determined either randomly, enabling the presentation of random initial configurations, or by the cell's rule table, in accordance with the current neighborhood of states. Each bit of the rule's bit string is stored in a D-type flip-flop whose inputs are channeled through a set of multiplexors according to the current operational phase of the system:

1. During the initialization phase of the evolutionary algorithm, the (eight) rule bits are loaded with random values; this is carried out once per evolutionary run.

2. During the execution phase of the cellular automaton, the rule bits remain unchanged. In this phase several random configurations are run by the system so as to be able to calculate a fitness value.

3. During the evolutionary phase, the cell's genome (which represents its rule table) may evolve via the application of genetic operators. This is done in a completely local manner—only the genomes of the neighboring cells may be consulted.

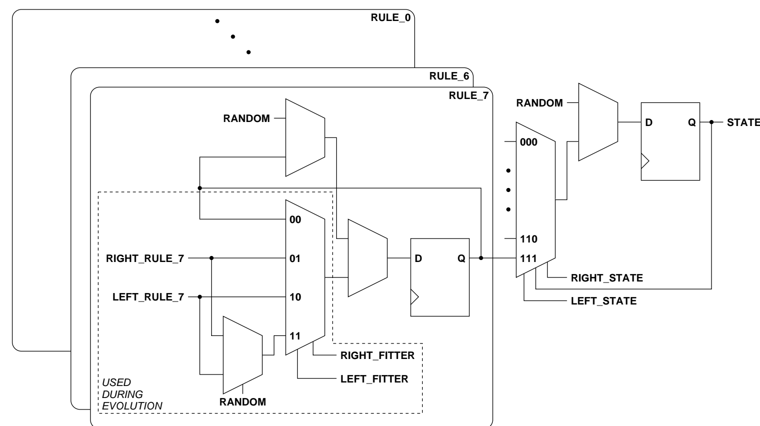

Figure 1.13 Circuit design of a Firefly cell.

Performance gains. The Firefly machine exhibits complete online evolution, all operations being carried out in hardware with no reference to an external computer. This demonstrates that evolving ware, *evolware*, can be constructed Sipper, 1997b; Sipper et al., 1997a. Such evolware systems enable enormous gains in execution speed to be had. The cellular programming algorithm, when run on a high-performance workstation, executes 60 initial configurations per second (as noted, random configurations are constantly presented to the cellular automaton during evolution—these are used to compute the fitness value). In comparison, the Firefly machine executes 13,000 initial configurations per second (this is achieved when the machine operates at the current maximal frequency of 1 MHz; in fact, this can easily be increased to 6 MHz, thereby attaining 78,000 configurations per second).

While the synchronization task is not a real-world application, and was selected to act as a benchmark problem for our evolware demonstration, Firefly does open up interesting avenues for future research.

Evolware machines that operate in an autonomous manner can be used in the field of autonomous mobile robots, as well as for the construction, in general, of controllers for noisy, changing environments Sipper et al., 1997b.

3.2 THE BIOWATCH

Type (Objective). Dynamic (Handle changing and/or incomplete specifications).

Functional description. The BioWatch is one of the applications designed as part of the Embryonics (embryonic electronics) project, whose final objective is the development of very large scale integrated circuits, capable of self-repair and self-replication Mange et al., 1998; Mange and Tomassini, 1998. These two bio-inspired properties, characteristic of the living world, are achieved by transposing certain features of cellular organization onto the two-dimensional world of integrated circuits on silicon.

The BioWatch is an artificial "organism" designed to count minutes (from 00 to 59) and seconds (from 00 to 59); it is thus a modulo-3600 counter. This organism is one-dimensional and comprises four cells with identical physical connections and an identical set of resources. The organization is multicellular (as with living beings), with each cell realizing a unique function, described by a sub-program called the gene of the cell (Figure 1.14).

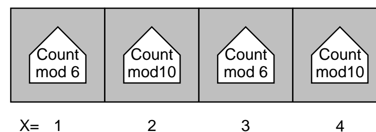

Figure 1.14 Multicellular organization of the BioWatch.

The genome is the set of all the genes of the BioWatch, where each gene is a sub-program, characterized by a set of instructions and by its horizontal coordinate X (Figure 1.14). Storing the whole genome in each cell renders the cell universal, i.e., capable of realizing any gene of the genome. This is another bio-inspired property: each of our (human) cells also contains the entire genome, though only part of it is used (e.g., liver cells do not use the same genes as muscle cells). Depending on its position in the organism, each cell interprets the genome and extracts

and executes the gene which configures it. The BioWatch thus performs what is known in biology as cellular differentiation (Figure 1.15).

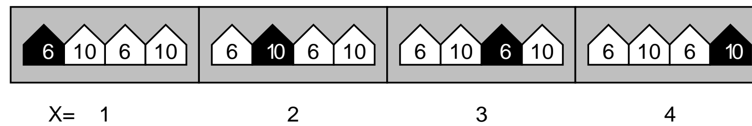

Figure 1.15 Cellular differentiation of the BioWatch.

In the BioWatch, each cell performs one of two specific tasks: a modulo-6 or a modulo-10 count (Figure 1.14). We shall show below that a dynamic reconfiguration of the task executed by some of the cells occurs during the self-repair process of this artificial organism.

Hardware description. The BioWatch is a four-cell, one-dimensional application of the two-dimensional cellular automaton defined in the Embryonics project Mange et al., 1998; Mange and Tomassini, 1998. Each cell of the automaton is a binary decision machine whose microprogram represents the genome, and each part of the microprogram is a gene whose execution depends on the physical position of the cell in the array, i.e., on its coordinates. Ultimately, we plan to implement the automaton using a novel kind of coarse-grained, field-programmable gate array, where each basic cell, called MICTREE (for tree of micro-instructions) has four neighbors (to the south, west, north, and east). The MICTREE cell holds a 4-bit state register $REG3:0$ (Figure 1.16a). Four 4-bit busses enter the cell from its neighbors ($SI3:0$ from the south, $WI3:0$ from the west, $NI3:0$ from the north, and $EI3:0$ from the east), and, correspondingly, four output busses go out in the four cardinal directions ($SO3:0$ to the south, $WO3:0$ to the west, $NO3:0$ to the north, and $EO3:0$ to the east).

Each MICTREE cell thus has 16 outputs $SO3...EO0$. Each of these outputs can be programmed to take on a value from four possible sources (Figure 1.16b). For example, output $NO3$ can take on one of 16 values from the following sources: the four bits $REG3:0$ of register REG, the four bits $SI3:0$ of the south input bus SI, the four bits $WI3:0$ of the west input bus WI, and the four bits $EI3:0$ of the east input bus EI.

The binary decision machine of the MICTREE cell executes microprograms written using a set of six instructions: (1) **if** VAR **else** $LABEL$, (2) **goto** $LABEL$, (3) **do REG** $= DATA$, (4) **do X** $= DATA$, (5) **do Y** $= DATA$, and (6) **do** $VAROUT = VARIN$. The first three instructions are used to compute the modulo-6 and modulo-10 counts of

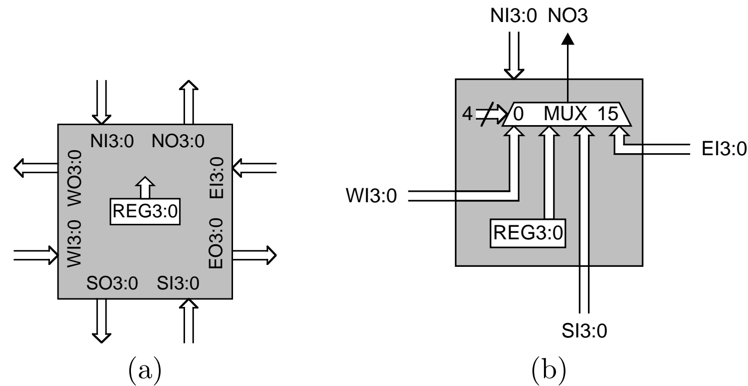

Figure 1.16 MICTREE cell. (a) Diagram of connections to the four neighboring cells. (b) Programmable output connections.

the BioWatch application. The next two are used when computing the $X3:0$ and $Y3:0$ coordinates of the cell. The last instruction is used to program the input/output connections.

While our long-term objective is the design of very large scale integrated circuits, each MICTREE cell is currently implemented in an Actel 1020 FPGA circuit and embedded within a small plastic box intended as a demonstration module.

Performance gains. Self-repair of an artificial organism allows partial reconstruction of the original device in case of a minor fault. In order to implement a self-repair process in the BioWatch, as many spare cells are required to the right of the array as there are faulty cells to repair (four spare cells in the example of Figure 1.17). This process is achieved by bypassing the faulty cell and shifting to the right all or part of the original cellular array. The new coordinates, thus defined, lead to the dynamic reconfiguration of the task performed by the cell (modulo-6 or modulo-10 count) .

Self-replication of an artificial organism allows for the complete reconstruction of the original device in case of a major fault. In the BioWatch, the self-replication process rests on two assumptions: (1) there exists a sufficient number of spare cells to the right of the array (four in our example), and (2) the calculation of the coordinates produces a cycle ($X = 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 1$ in Figure 1.18). As the same pattern of coordinates produces the same pattern of genes, self-replication can be easily accomplished if the microprogram of the genome, associated with

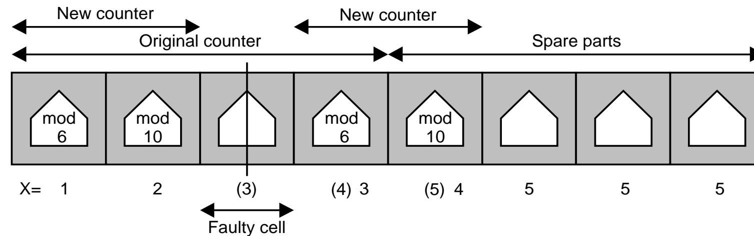

Figure 1.17 Self-repair of the BioWatch. Old coordinates are shown in parentheses.

the homogeneous network of cells, produces several instances of the basic pattern of coordinates.

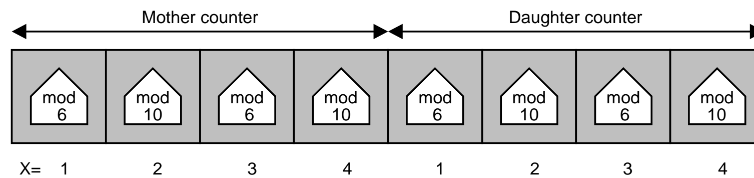

Figure 1.18 Self-replication of the BioWatch.

With a larger number of cells it becomes possible to add the extensions needed for a practical use of the BioWatch: preserving the current time while self-repair is being effected, and setting and resetting the time. It is also quite easy to introduce additional functions other than the counting of seconds, minutes, and hours; for example, computing the date, keeping track of the day of the week, or handling leap years.

3.3 THE FAST NEURAL NETWORK

Type (Objective). Dynamic (Handle changing and/or incomplete specifications).

Functional description. As seen in Section 2.3, an artificial neural network is specified by its topology, the neuron characteristics, and the training or learning algorithm. Most neural network models base their ability to adapt to problems on changing the strengths of their interconnections, according to a given learning algorithm. However, the difficulty in determining the appropriate topology, including the number of layers,

the number of neurons per layer, and the interconnection scheme, often sets an a-priori limit on performance.

FPGAs enable the implementation of neural networks with dynamic struc-tures—the system is able to learn, online, as well as to adapt its topology. While artificial neural networks have been implemented using field programmable devices Bade and Hutchings, 1994; Eldredge and Hutchings, 1994, reconfigurability has not been exploited for dynamic structure optimization.

Herein, we describe a "hardware-friendly" artificial neural network architecture, dubbed *FAST* (Flexible, Adaptable-Size Topology), that implements an unsupervised clustering algorithm. The network incrementally activates neurons, dynamically adapts their weights, and probabilistically deletes neurons Pérez-Uribe and Sanchez, 1996a; Pérez-Uribe and Sanchez, 1996b.

The aim of unsupervised neural networks is to *cluster*, *code*, or *categorize* the input data. Similar inputs are classified as being in the same category, and should activate the same output unit, which corresponds to a prototype of the category. Clusters are determined by the network itself, based on correlations in the input Hertz et al., 1991. A special class of unsupervised learning networks called *ontogenic neural networks* offers the possibility of dynamically modifying the network's topology Fiesler, 1994; Fritzke, 1997. Among the few hardware implementations of this latter approach, one can cite Moreno's work on VLSI architectures for evolutive neural networks Moreno, 1994, and our own FPGA-implemented FAST network. This latter combines three neural network algorithms: the ART neural network Carpenter and Grossberg, 1988, Alpaydin's extension of ART Alpaydin, 1990, and Fritzke's Growing Cell Structures Fritzke, 1994.

The size of the FAST network increases by adding a new neuron to the network when a *sufficiently distinct* input vector is encountered, and decreases by deleting an operational neuron through the application of probabilistic deactivation. Each neuron j maintains an n-dimensional reference vector, W_j, and a threshold, T_j, both of which determine its *sensitivity region*, i.e., the input vectors to which it is "sensitive." At the outset, the network consists of a maximum number of neurons, but none of them is active. Input patterns are then presented and the network adapts through application of the FAST algorithm (see Pérez-Uribe, 1998; Pérez-Uribe and Sanchez, 1996b), which is driven by three processes: learning, incremental growth, and pruning.

The learning mechanism adapts the neuronal reference vectors (W_j and T_j), as each input vector P is presented to the network. Incremental growth dynamically activates new neurons when an input vector P

lies outside the sensitivity regions of all currently operational neurons. Finally, the pruning mechanism probabilistically decreases the size of the network. The probability of an operational neuron being deleted, Pr_j, increases in direct proportion to the overlap between its sensitivity region and the regions of its neighbors. The overlap between the sensitivity regions of several neurons is estimated by computing the frequency of activation of the overlapping neurons with the same input vector.

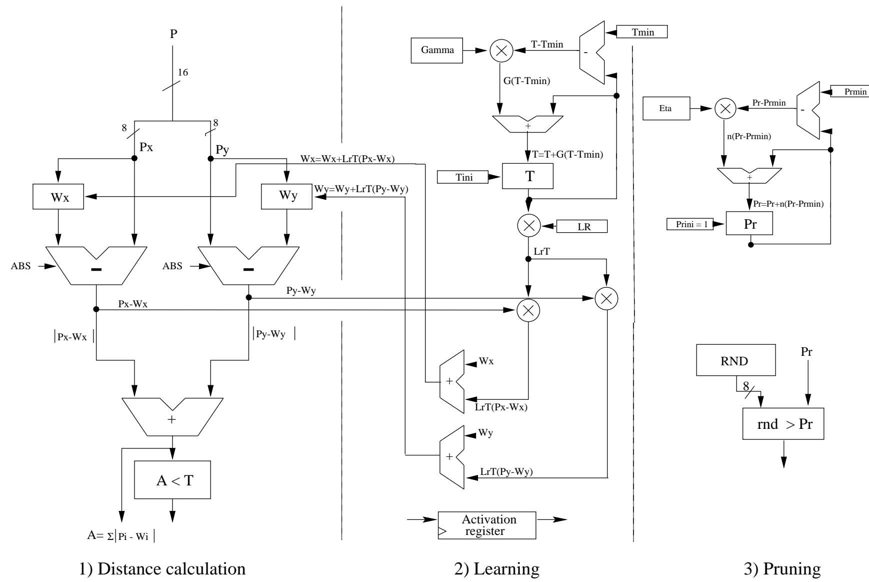

Figure 1.19 Architecture of a FAST neuron. Each neuron contains a distance calculation block (Manhattan distance), a reference vector and threshold adaptation block (learning), a deletion block (pruning), and an activation register.

Hardware description. The FAST neural network architecture was implemented on a custom machine called LOPIOM, which was designed in our laboratory, and is based on programmable logic Mosanya et al., 1996. It is composed of a 68331 microcontroller, four Xilinx XC4013-6 FPGA chips (each with approximately 13,000 equivalent gates Xilinx, 1991), and four field-programmable interconnection devices (FPIDs, in our case Crossbar switches I-Cube 320, each with 320 programmable pads I-CUBE, 1994). We used two of the Xilinx XC4013 FPGAs and two of the I-Cube 320 FPIDs to implement four FAST neurons. The sequencer of the network is implemented in one of the Xilinx XC4013 chips along with two of the FAST neurons. The other two FAST neurons

are implemented in the second XC4013 chip along with a bank of I/O mapping registers.

To date, two such FAST networks have been implemented: one for two-dimensional input vectors and another for three-dimensional input vectors. In both implementations we used 96% to 98% of the Configurable Logic Blocks (CLBs) of the chip. However, only 66% to 79% of the CLBs were actually used to implement logic circuits, while the rest were used for routing. In the 2D-input network we used 30% of the flip-flops, while the 3D-input network required up to 35% of the memory elements.

The sequencer, the FAST neuron architecture, and the I/O mapping register bank are briefly described below.

- *The FAST neuron.* A FAST neuron is composed of three blocks: Manhattan distance computation, learning (i.e., modification of reference vectors and thresholds), and pruning (see Figure 1.19). The system currently supports 8-bit computation. Each neuron includes nine 8-bit adders in the 2D case and twelve 8-bit adders in the 3D case, and a single 8-bit shift-add multiplier, so that additions occur in parallel but the multiplications involved in the learning phase are executed sequentially. The maximal number of multiplications during the learning phase is five for the 2D case and six for the 3D case, and each requires 8 clock cycles. To implement the pruning process, each neuron includes a random number generator, consisting of an 8-cell heterogeneous, one-dimensional cellular automaton Hortensius et al., 1989a; Hortensius et al., 1989b. It has a period of 255 and is implemented using a small number of components: 8 D flip-flops, 5 two-input XOR gates, and 3 three-input XOR gates. Finally, each neuron maintains an *activation register* that indicates whether it is active or inactive.

- *The sequencer.* The sequencer is a finite state machine that handles the addition and deletion of neuronal units in the network (by writing in the *activation register*). In addition it synchronizes the neuron's operation during the Manhattan distance calculation, weight and threshold updating (learning), and probabilistic unit deactivation (pruning).

- *I/O mapping register bank.* The hardware device is connected to a host computer that is used to generate input vectors for the system and to display its outputs. The host computer communicates serially with the 68331 microcontroller that reads and writes signals from and to the I/O mapping registers. The I/O mapping

registers bank consists of ten 8-bit registers used to map every input and output of the neural network. A polling subroutine runs on the 68331 microcontroller in order to generate the read/write control signals and to receive or send data when the host computer requests it. A VME bus connection is also available on the board, but it has not been tested to date.

Performance gains. To evaluate FAST we chose a color image segmentation problem. Given a digital image from a real scene, the problem is to cluster image pixels by chromatic similarity properties. Pixels of similar colors belong to a so-called *chromatic class*. By identifying chromatic classes in an image, and assigning a code to each class, we can do color image segmentation and recognition. For example, an orange sphere with small bumps on its surface can be quickly recognized as an orange fruit. In our experiments we consider the RGB (red, green, blue) components of a color image. Thus, at the outset every pixel is represented in a 3D space. Then, a color-space transformation is applied so as to represent the pixels in a 2D space called I2-I3 Diaz and Quesada, 1995; Ohta, 1985.

These two-dimensional pixel coordinates (I2,I3) are randomly presented to the FAST neural network, which determines clusters in the I2-I3 space, corresponding to color clusters in the RGB space. Then, the resulting color clusters can be used to construct a segmented image that is used in subsequent image analysis phases. In Pérez-Uribe and Sanchez, 1996c we compared the resulting color clusters using FAST with a neural network learning algorithm that combines histogram thresholding techniques Ohta et al., 1990 and fuzzy Kohonen clustering Diaz and Quesada, 1995. This latter network ends up with eight ellipsoidal chromatic clusters in I2-I3 space. Our algorithm dynamically derived eight diamond-shaped (due to the Manhattan distance calculation) chromatic clusters that corresponded to a good estimation of those obtained by the non-dynamic model.

The maximal XC4013-6's pin-to-pin delay of the 2D-input FAST network is 120.2 ns, and the I-CUBE's pin-to-pin delay is $12ns$. The processing time per input vector is up to 57 clock cycles, depending on the number of multiplications during learning, thus enabling the introduction of a new input vector approximately every $7.5\mu s$. For the 3D-input case, the maximal XC4013-6's pin-to-pin delay is $172.7ns$ and a maximum of 66 clock cycles is needed for a learning step, thus enabling the introduction of a new input vector approximately every $8.2\mu s$.

In Littmann and Ritter, 1997, neural and statistical methods for adaptive color segmentation were compared. In these tests, a set of 400x400

pixel images were used, and a maximum of 30 classes were allowed. The processing of one such image running on an IBM RISC 350 workstation took 30 seconds. If an extension of our 3D-input system were used (for clustering directly in RGB space) with these images, it would take 2x(8.2μsx400x400) = 2.62 seconds (one phase is needed for clustering and a second phase to generate the segmented image); thus, our system is about 11 times faster. One possible extension is to use denser Xilinx devices, such as the new XC4036, rather than the older XC4013 devices we have been using until now.

4. CODESIGN AND THE NEED FOR A CHANGE IN THE ENGINEERING CURRICULUM

Before concluding we would like to discuss in this section an issue which usually remains outside the spotlight—teaching—but which is nonetheless highly important, and, what's more, it bears directly on the use of configurable systems. In the classical university curriculum there is a hard distinction between programmable computing and configurable computing; the former, considered as a software practice, is under the responsibility of the computer science department, whereas the latter is regarded as a hardware practice, thereby to be taught at the electrical engineering (or computer engineering) department. However, we believe that this clear-cut frontier is dissolving, with these dichotomous domains slowly merging into a continuum. One prominent aspect of this fusion is the increasing importance attached to the *codesign* issue—basically, the decision of which parts of the application are to be designed as software and which shall be designed directly as hardware de Micheli and Gupta, 1997.

As our laboratory is responsible for teaching several hardware courses, both at the elementary and advanced levels, we have become acutely aware of the codesign issue in the past few years. For the advanced courses we make use of the RENCO processor (Section 2.2), whereas for the elementary-level courses we have developed a special teaching platform—the LABOMAT board—described below.

Type (Objective). Static (Improve performance + Optimize resource usage).

Functional description. LABOMAT is a "lightweight" platform aimed at studying digital design, microcontroller programming, and—above all—the interface between the two: codesign. We followed the "small

is beautiful" approach in order to obtain a low-cost system that could be installed by the dozens, furnishing each student (or small team of students) with their own setup. It has been used with success in our courses and has become a highly regarded prototyping tool that is used not only by students but for our own research as well.

Hardware description. The LABOMAT board (Figure 1.20) consists of a Motorola MC68331 microcontroller, a X4010 Xilinx FPGA, 128KB of ROM, 256KB of Flash memory, and 256KB of RAM. It communicates with a workstation (the host) through a RS-232 interface.

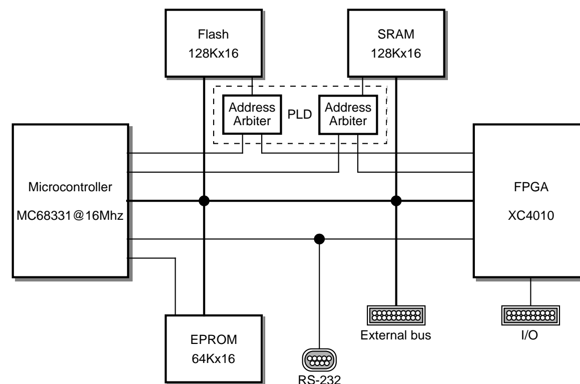

Figure 1.20 The LABOMAT block diagram.

All the microcontroller pins are connected to the FPGA, leaving some FPGA pins free to be used as a general extension port. The memory is shared between the CPU and the FPGA through a shared bus (a simple arbitration scheme prevents access conflicts). All the components are set on a double-sided PCB.

Software description. The board is controlled by a monitor stored in the ROM. The monitor initializes the CPU and communicates with the host station through an interactive, character-based interface which allows the user to execute basic commands, including: uploading a program into the processor, executing it, reading from and writing to the RAM and Flash memories. The monitor is also used to easily configure the FPGA using a binary file generated by the placement and routing tools. This configuration file can be stored in the host system or in the board memory (i.e., in the ROM, the RAM, or the Flash memory).

The ROM also contains a library of low-level functions that simplifies the access to all board resources, namely, those handling the communications with the RS-232 port, the transfer of memory blocks, the programming of the Flash memory, the configuration of the FPGA, and the conversion between different data formats. This library can be used by the monitor and by user application programs.

A user developing an application can test the executable file using the RAM, and—after debugging—transfer it to the Flash memory. The same process can be used by the configurer for the FPGA configuration data. The LABOMAT board is then able to run stand-alone without requiring the presence of a host.

Applications. From the early design stages it was recognized that LABOMAT could become a very useful platform for reconfiguration and codesign research. In fact, with its microcontroller, its programmable logic, and the complete interconnection between the two, the board provides a system that can be used to experiment with the design of dedicated systems comprising both hardware and software, as well as to explore different interfaces between the two parts.

LABOMAT can be used to develop and validate automatic codesign tools, and—since its architecture is conceptually simple—the designer can concentrate his or her effort on the partitioning and synthesis without having to deal with the complexity of the target platform.

The board has been successfully used for student exercises. It has proven itself to be a very simple and flexible tool for teaching basic digital design and microprocessor interfacing. Due to the reconfigurability of the FPGA, it is possible for students to go beyond simulation, testing their designs with real hardware, thus allowing them to confront problems that do not often appear in a simulation. Furthermore, they are able to experience the satisfaction of obtaining a *bona fide* hardware system.

The flexibility and the ease of programming and configuration allow projects to be completed faster and enable students to focus their attention on the design. For example, the design and test of a simplified floating point unit was achieved by a three-student team in five three-hour sessions.

Based on our experience with RENCO (Section 2.2), a new board—LABO-MAT II—is currently under development. The microprocessor part is identical to RENCO, thus adding with respect to LABOMAT I an Ethernet interface and the high-level software (real-time operating system and Java interface). The reconfigurable part, on the other hand, is simpler and cheaper, comprising but two FPGA circuits, one

Xilinx XC4000, and one Xilinx XC6200 (this latter allowing for dynamic reconfiguration).

5. CONCLUDING REMARKS

Our aim herein has been to demonstrate a number of FPGA applications that cover a wide range of characteristics, exemplifying the use of configurable circuits in general, and within the domain of bio-inspired systems in particular. First and foremost we made a distinction—which we believe to be of prime import—between static and dynamic configuration strings. The former, aimed at configuring the processor so as to perform a given function, is loaded once at the outset, after which it does not change during execution of the task at hand. A dynamic configuration string, on the other hand, can continually change.

Static FPGA applications, such as SPYDER, RENCO, and the back-propagation neural network are mainly aimed at attaining the classical goal in computing: that of improving performance—be it in terms of speed, resource utilization, or area usage. With configurable processors slowly but surely inching their way toward the mainstream of the computing industry, we will probably be seeing more such static applications in the near future. Thus, the future may see a merging of the classical processor industry with the configurable computing industry.

Dynamic devices, such as Firefly, BioWatch, and FAST represent a less conventional approach that may in fact be quite revolutionary (though perhaps not in the immediate future). With the rise of bio-inspired computing, we expect to see more hardware devices imbued with properties usually associated up until now only with living beings: learning, evolution, self-repair, self-replication, and so forth. In general, this will result in systems that are more *adaptive*—able to undergo modifications according to changing circumstances, thus continuing to function within their dynamic environments. The applications of such systems are bounded only by our imagination.

Acknowledgments

We are grateful to Daniel Mange for helpful discussions.

References

Alpaydin, A. E. (1990). *Neural Models of Incremental Supervised and Unsupervised Learning.* PhD thesis, Swiss Federal Institute of Technology, Lausanne. Thesis no. 863.

Altera (1997). *EPF10K130 Embedded Programmable Logic Device*. Altera Corporation.

Arnold, J. M., Buell, D. A., and Davis, E. G. (1992). Splash 2. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 316–322.

Bade, S. L. and Hutchings, B. L. (1994). FPGA-based stochastic neural networks – implementation. In Buell, D. A. and Pocek, K. L., editors, *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 189–198, Los Alamitos, CA. IEEE Computer Society Press.

Berlekamp, E. R., Conway, J. H., and Guy, R. K. (1982). *Winning Ways for your Mathematical Plays*, volume 2, chapter 25, pages 817–850. Academic Press, New York.

Bertin, R., Roncin, D., and Vuillemin, J. (1989). Introduction to programmable active memories. Technical Report 3, DEC Paris Research Laboratory.

Carpenter, G. and Grossberg, S. (1988). The ART of Adaptive Pattern Recognition by a self-organizing neural network. *IEEE Computer*, pages 77–88.

de Micheli, G. and Gupta, R. K. (1997). Hardware/Software co-design. *Proceedings of the IEEE*, 85(3):349–365.

DeHon, A. (1996). Architectures for general-purpose computing. Technical Report A.I. Technical Report No. 1586, Artificial Intelligence Laboratory, MIT.

Diaz, D. B. and Quesada, J. G. (1995). Learning algorithm with Gaussian membership function for fuzzy RBF neural networks. In *From Natural to Artificial Neural Computation*, pages 527–534. Springer Verlag.

Eldredge, J. G. and Hutchings, B. L. (1994). Density enhancement of a neural network using FPGAs and run-time reconfiguration. In *IEEE Workshop on FPGAs for Custom Computing Machines*.

Fiesler, E. (1994). Comparative bibliography of ontogenic neural networks. In Marinaro, M. and Morasso, P. G., editors, *Proceedings of the International Conference on Artificial Neural Networks (ICANN'94)*, volume 1, pages 793–796, London, U.K. Springer-Verlag.

Fogel, D. B. (1995). *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. IEEE Press, Piscataway, NJ.

Fritzke, B. (1994). Growing cell structures – a self-organizing network for unsupervised and supervised learning. *Neural Networks*, 7(9):1441–1460.

Fritzke, B. (1997). Unsupervised ontogenic networks. In *Handbook of Neural Computation*, pages C2.4:1–C2.4:16. Institute of Physics Publishing and Oxford University Press.

Goeke, M., Sipper, M., Mange, D., Stauffer, A., Sanchez, E., and Tomassini, M. (1997). Online autonomous evolware. In Higuchi, T., Iwata, M., and Liu, W., editors, *Proceedings of the First International Conference on Evolvable Systems: From Biology to Hardware (ICES96)*, volume 1259 of *Lecture Notes in Computer Science*, pages 96–106. Springer-Verlag, Heidelberg.

Habib, S., editor (1988). *Microprogramming and Firmware Engineering Methods*. Van Nostrand, New York.

Hertz, J., Krogh, A., Lautrup, B., and Lehmann, T. (1997). Non-linear back-propagation: Doing back-propagation without derivatives of the activation function. *IEEE Transactions on Neural Networks*, 8(6):1321–1327.

Hertz, J., Krogh, A., and Palmer, R. G. (1991). *Introduction to the Theory of Neural Computation*. Addison-Wesley Publishing Company, Redwood City, CA.

Hortensius, P. D., McLeod, R. D., and Card, H. C. (1989a). Parallel random number generation for VLSI systems using cellular automata. *IEEE Transactions on Computers*, 38(10):1466–1473.

Hortensius, P. D., McLeod, R. D., Pries, W., Miller, D. M., and Card, H. C. (1989b). Cellular automata-based pseudorandom number generators for built-in self-test. *IEEE Transactions on Computer-Aided Design*, 8(8):842–859.

I-CUBE (1994). *I-CUBE. The FPID Family Data Sheet*. I-CUBE Inc, 2.0 edition.

Ienne, P. (1997). Digital connectionist hardware: Current problems and future challenges. In Mira, J., Moreno-Díaz, R., and Cabestany, J., editors, *Biological and Artificial Computation: From Neuroscience to Technology*, pages 688–713. Springer.

Iseli, C. (1996). *Spyder: A Reconfigurable Processor Development System*. PhD thesis, Computer Science Department, Swiss Federal Institute of Technology, Lausanne. Thesis no. 1476.

Iseli, C. and Sanchez, E. (1995). Spyder: A SURE (SUperscalar and REconfigurable) processor. *The Journal of Supercomputing*, 9(3):231–252.

Iseli, C. and Sanchez, E. (1997). Spyder: Un processeur reconfigurable réalisé à l'aide de circuits FPGA. *Calculateurs Parallèles*, 9(1):29–43.

Ishikawa, M. (1996). Structural learning with forgetting. *Neural Networks*, 9(3):509–521.

Köllmann, K., Riemschneider, K.-R., and Zeidler, H. C. (1996). On-chip backpropagation training using parallel stochastic bit streams. In *Proceedings of MicroNeuro '96*, pages 149–156. IEEE Computer Society Press.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, Cambridge, Massachusetts.

Littmann, E. and Ritter, H. (1997). Adaptive Color Segmentation – A comparison of neural and statistical methods. In *IEEE Transactions on Neural Networks*, pages 175–185.

Mange, D., Sanchez, E., Stauffer, A., Tempesti, G., Marchal, P., and Piguet, C. (1998). Embryonics: A new methodology for designing field-programmable gate arrays with self-repair and self-replicating properties. *IEEE Transactions on VLSI Systems*, 6(3):387–399.

Mange, D. and Tomassini, M., editors (1998). *Bio-Inspired Computing Machines: Toward Novel Computational Architectures*. Presses Polytechniques et Universitaires Romandes, Lausanne, Switzerland.

Michalewicz, Z. (1996). *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, Heidelberg, third edition.

Moreno, J. M. (1994). *VLSI Architectures for Evolutive Neural Models*. PhD thesis, Universitat Politecnica de Catalunya, Barcelona.

Mosanya, E., Goeke, M., Linder, J., Perrier, J.-Y., Rampogna, F., and Sanchez, E. (1996). A platform for co-design and co-synthesis based on FPGA. In *Proceedings of the Seventh IEEE International Workshop on Rapid System Prototyping*, pages 11–16.

Motorola (1993). *Quad Integrated Communications Controller*. Motorola.

Ohta, Y.-I. (1985). *Knowledge-Based Interpretation of Outdoor Natural Color Scenes*. Pitman Advanced Publishing Program.

Ohta, Y.-I., Kanade, T., and Sakai, T. (1990). Color information for region segmentation. In *Comp. Graphics Image Processing*, volume 13, pages 224–241.

Pérez-Uribe, A. (1998). Artificial neural networks: Algorithms and hardware implementation. In Mange, D. and Tomassini, M., editors, *Bio-Inspired Computing Machines: Toward Novel Computational Architectures*, pages 289–316. Presses Polytechniques et Universitaires Romandes, Lausanne, Switzerland.

Pérez-Uribe, A. and Sanchez, E. (1996a). The FAST architecture: A neural network with flexible adaptable-size topology. In *Proceedings of the Fifth International Conference on Microelectronics for Neural Networks and Fuzzy Systems*, pages 337–340, Lausanne, Switzerland.

Pérez-Uribe, A. and Sanchez, E. (1996b). FPGA implementation of an adaptable-size neural network. In von der Malsburg, C., von Seelen, W., Vorbrüggen, J. C., and Sendhoff, B., editors, *Proceedings of the International Conference on Artificial Neural Networks (ICANN96)*, volume 1112 of *Lecture Notes in Computer Science*, pages 383–388. Springer-Verlag, Heidelberg.

Pérez-Uribe, A. and Sanchez, E. (1996c). Implementation of neural constructivism with programmable hardware. In *Proceedings of the International Symposium on Neuro-Fuzzy Systems, AT'96*, pages 47–54. IEEE Press.

Prechelt, L. (1994). Proben1 - a set of neural network benchmark problems and benchmarking rules. Technical Report 21/94, Fakultät für Informatik, University of Karlsruhe, 76128 Karlsruhe, Germany.

Prechelt, L. (1995). Adaptative parameter pruning in neural networks. Technical Report 95-009, International Computer Science Institute, Berkeley, CA.

Rau, B. R. and Fisher, J. A. (1993). Instruction-level parallelism: History, overview, and perspectives. In Rau, B. R. and Fisher, J. A., editors, *Intruction-Level Parallelism*, pages 9–50. Kluwer Academic Publishers, Boston.

Reed, R. (1993). Pruning algorithms - a survey. *IEEE Transactions on Neural Networks*, 4(5):740–747.

Report, M. (1998). Motorola core+ chip merges CPU with FPGA. *Microprocessor Report*, 12(2):10.

Riedmiller, M. (1994). Rprop - description and implementation details. Technical report, Institut für Logik, Komplexität und Deduktionssysteme, University of Karlsruhe, 76128 Karlsruhe, Germany.

Rojas, R. (1996). *Neural Networks: A Systematic Introduction.* Springer-Verlag, Berlin.

Sakaue, S., Kodha, T., Yamamoto, H., Maruno, S., and Shimeki, Y. (1993). Reduction of required precision bits for back-propagation applied to pattern recognition. *IEEE Transactions on Neural Networks*, 4(2):270–275.

Salcic, Z. and Smailagic, A. (1997). *Digital System Design and Prototyping Using Field Programmable Logic.* Kluwer Academic Publishers, Boston.

Sanchez, E., Sipper, M., Haenni, J.-O., Beuchat, J.-L., Stauffer, A., and Pérez-Uribe, A. (1999). Static and dynamic configurable systems. *IEEE Transaction on Computers.* (to appear).

Sanchez, E. and Tomassini, M., editors (1996). *Towards Evolvable Hardware*, volume 1062 of *Lecture Notes in Computer Science.* Springer-Verlag, Heidelberg.

Sipper, M. (1997a). *Evolution of Parallel Cellular Machines: The Cellular Programming Approach.* Springer-Verlag, Heidelberg.

Sipper, M. (1997b). The evolution of parallel cellular machines: Toward evolware. *BioSystems*, 42:29–43.

Sipper, M., Goeke, M., Mange, D., Stauffer, A., Sanchez, E., and Tomassini, M. (1997a). The firefly machine: Online evolware. In *Proceedings of*

1997 IEEE International Conference on Evolutionary Computation (ICEC'97), pages 181–186.

Sipper, M., Mange, D., and Pérez-Uribe, A., editors (1998). *Proceedings of The Second International Conference on Evolvable Systems: From Biology to Hardware (ICES98)*, volume 1478 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg.

Sipper, M., Sanchez, E., Mange, D., Tomassini, M., Pérez-Uribe, A., and Stauffer, A. (1997b). A phylogenetic, ontogenetic, and epigenetic view of bio-inspired hardware systems. *IEEE Transactions on Evolutionary Computation*, 1(1):83–97.

Slater, M. (1996). The many faces of network computers. *Microprocessor Report*, 10(16):3.

Smith, M. J. S. (1997). *Application-Specific Integrated Circuits*. Addison Wesley, Reading, MA.

Thompson, A. (1996). Silicon evolution. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 444–452, Cambridge, MA. The MIT Press.

Thompson, A. (1997). An evolved circuit, intrinsic in silicon, entwined with physics. In Higuchi, T., Iwata, M., and Liu, W., editors, *Proceedings of the First International Conference on Evolvable Systems: From Biology to Hardware (ICES96)*, volume 1259 of *Lecture Notes in Computer Science*, pages 390–405. Springer-Verlag, Heidelberg.

Thompson, A., Harvey, I., and Husbands, P. (1996). Unconstrained evolution and hard consequences. In Sanchez, E. and Tomassini, M., editors, *Towards Evolvable Hardware*, volume 1062 of *Lecture Notes in Computer Science*, pages 136–165. Springer-Verlag, Heidelberg.

Trimberger, S. M., editor (1994). *Field-Programmable Gate Array Technology*. Kluwer Academic Publishers, Boston.

Villasenor, J. and Mangione-Smith, W. H. (1997). Configurable computing. *Scientific American*, 276(6):54–59.

Weaver, D. L. and Germond, T., editors (1994). *The SPARC Architecture Manual*. Prentice Hall, Englewood Cliffs.

Widrow, B. and Lehr, M. A. (1990). 30 years of adaptative neural networks: Perceptron, madaline and backpropagation. *Proc. IEEE*, 78(9):1415–1442.

Wolfram, S. (1994). *Cellular Automata and Complexity*. Addison-Wesley, Reading, MA.

Xilinx (1991). *The XC4000 Data Book*. Xilinx, San Jose.

Zell, A., Mamier, G., Vogt, M., Mache, N., Hübner, R., Dring, S., Herrmann, K.-U., Soyez, T., Schmalzl, M., Sommer, T., Hatzigeorgiou, A., Posselt, D., Schreiner, T., Kett, B., Clemente, G., and Wieland, J.

(1995). SNNS, user manual, version 4.1. Technical Report 6/95, Institute for Parallel and Distributed High Performance Systems, University of Stuttgart. ftp://ftp.informatik.uni-stuttgart.de/pub/SNNS.

Moshe Sipper is a Senior Researcher in the Logic Systems Laboratory at the Swiss Federal Institute of Technology, Lausanne, Switzerland. He received a B.A. in Computer Science from the Technion – Israel Institute of Technology, an M.Sc. and a Ph.D. from Tel Aviv University. His chief interests involve the application of biological principles to artificial systems, including evolutionary computation, cellular automata (with an emphasis on evolving cellular machines), bio-inspired systems, evolving hardware, complex adaptive systems, artificial life, and neural networks. Dr. Sipper has authored and coauthored several scientific papers in these areas, as well as the book *Evolution of Parallel Cellular Machines: The Cellular Programming Approach* (Heidelberg: Springer-Verlag, 1997). He was Program Chairman of the Second International Conference on Evolvable Systems: From Biology to Hardware (ICES98), held in Lausanne in September 1998.

Eduardo Sanchez is Professor of Computer Science in the Logic Systems Laboratory at the Swiss Federal Institute of Technology, Lausanne, Switzerland. He received a diploma in Electrical Engineering from the Universidad del Valle, Cali, Colombia, in 1975, and a Ph.D. from the Swiss Federal Institute of Technology in 1985. Since 1977 he has been with the Department of Computer Science, Swiss Federal Institute of Technology in Lausanne, where he is engaged in teaching and research. His chief interests include computer architecture, VLIW processors, reconfigurable logic, and evolvable hardware. Dr. Sanchez was co-organizer of the inaugural workshop in the field of bio-inspired hardware systems, the proceedings of which are entitled *Towards Evolvable Hardware* (Heidelberg: Springer-Verlag, 1996).

Jacques-Olivier Haenni received a diploma in Computer Engineering from the Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland, in 1997. He is currently a Ph.D. candidate at the Logic Systems Laboratory, EPFL. His research interests include computer architecture, reconfigurable computing, and co-design.

Jean-Luc Beuchat received a diploma in Computer Engineering from the Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland, in 1997. He is currently a Ph.D. candidate in the Department of Computer Science at the EPFL. Since 1997 he has been with the Logic Systems Laboratory, working on the digital implementation of reconfigurable neuroprocessors. His research interests include neural networks, field-programmable devices, reconfigurable systems, and on-line arithmetic.

André Stauffer is a Senior Lecturer in the Department of Computer Science at the Swiss Federal Institute of Technology in Lausanne, Switzerland. In addition to digital design, his research interests include circuit reconfiguration and bio-inspired systems. He received a diploma in Electrical Engineering and a Ph.D. degree from the Swiss Federal Institute of Technology. He spent one year as a Visiting Scientist at the IBM T. J. Watson Research Center in Yorktown Heights,

NY. Dr. Stauffer also collaborates with the Centre Suisse d'Electronique et de Microtechnique SA in Neuchâtel, Switzerland. He was co-organizer of a special session entitled "Toward Evolware," held as part of the IEEE International Conference on Evolutionary Computation (ICEC'97).

Andrés Pérez-Uribe received a diploma in Electrical Engineering from the Universidad del Valle, Cali, Colombia, in 1993. From 1994 to 1996 he held a Swiss government fellowship, and is currently a Ph.D. candidate in the Department of Computer Science at the Swiss Federal Institute of Technology in Lausanne. Since 1994 he has been with the Logic Systems Laboratory, working on the digital implementation of neural networks with adaptable topologies, in collaboration with the Centre Suisse d'Electronique et de Microtechnique SA (CSEM). His research interests include artificial neural networks, field-programmable devices, evolutionary techniques, and complex and bio-inspired systems. He was a member of the steering committee and secretary of the Second International Conference on Evolvable Systems: From Biology to Hardware (ICES98), held in Lausanne in September 1998.