76. Evolving Embedded Fuzzy Controllers

Oscar H. Montiel Ross, Roberto Sepúlveda Cruz

The interest in research and implementations of type-2 fuzzy controllers (T2FCs) is increasing. It has been demonstrated that these controllers provide more advantages in handling uncertainties than type-1 FCs (T1FCs). This characteristic is very appealing because real-world problems are full of inaccurate information from diverse sources. Nowadays, it is no problem to implement an intelligent controller (IC) for microcomputers since they offer powerful operating systems, high-level languages, microprocessors with several cores, and co-processing capacities on graphic processing units (GPUs), which are interesting characteristics for the implementation of fast type-2 ICs (T2ICs). However, the above benefits are not directly available for the design of embedded ICs for consumer electronics that need to be implemented in devices such as an application-specific integrated circuit (ASIC), a field-programmable gate array (FPGAs), etc. Fortunately, for T1FCs there are platforms that generate code in VHSIC hardware description language (VHDL; VHSIC: very high speed integrated circuit), C++, and Java. This is not true for the design of T2ICs, since there are no specialized tools to develop the inference system as well as to optimize it.

The aim of this chapter is to present different ways of achieving high-performance computing for evolving T1 and T2 ICs embedded into FPGAs. Therefore, we provide a compiled introduction to T1 and T2 FCs, with emphasis on the well-known bottle neck of the interval T2FC (IT2FC), and software and hardware proposals to minimize its effect regarding computational cost. An overview of learning systems and hosting technology for their implementation is given. We explain different ways to achieve such implementations: at the circuit level using a hardware description language, using a multiprocessor system and a high-level language, and combining both methods. We explain how to use the IT2FC developed in VHDL as a standalone system, and as a coprocessor for the FPGA Fusion of Actel, Spartan 6, and Virtex 5. We present the methodology and two new proposals to achieve evolution of the IT2FC for FPGA, one for the static region of the FPGA, and the other one for the reconfigurable region using the dynamic partial reconfiguration methodology.

76.1 Overview

An intelligent system and evolution are intrinsically related since it is difficult to conceive intelligence without evolution because intelligence cannot be static. Human beings create, adapt, and replace their own rules throughout their whole lives. The idea to apply evolution to a fuzzy system is an attempt to construct a mathematical assembly that can approximate human-like reasoning and learning mechanisms [76.1]. A mathematical tool that has been successfully applied to better represent different forms of knowledge is fuzzy logic (FL); also if-then rules are a good way to express human knowledge, so the application of FL to a rule-based system leads to a Fuzzy Rule-Based System (FRBS). Unfortunately, an FRBS is not able to learn by itself, the knowledge needs to be derived from the expert or generated automatically with an evolutionary algorithm (EA) such as a genetic algorithm (GA) [76.2].

The use of GAs to design machine learning systems constitutes the soft computing paradigm known as the genetic fuzzy system where the goal is to incorporate learning to the system or tuning different components of the FRBS. Other proposals in the same line of work are: genetic fuzzy neural networks, genetic fuzzy clustering, and fuzzy decision trees. A system with the capacity to evolve can be defined as a self-developing, self-learning, fuzzy rule-based or neuro-fuzzy system with the ability to self-adapt its parameters and structure online [76.3].

Figure 76.1 shows the general structure of an evolutionary FRBS (EFRBS) that can be used for tuning or learning purposes. Although, it is difficult to make a clear distinction between tuning and learning, the particular aspect of each process can be summarized as follows. The tuning process is assumed to work on a predefined rule base having the target to find the optimal set of parameters for the membership functions and/or scaling functions. On the other hand, the learning process requires that a more elaborated search in the space of possible rule bases, or in the whole knowledge base be achieved, as well as for the scaling functions. Since the learning approach does not depend on a predefined set of rules and knowledge, the system can change its fundamental structure with the aim of improving its performance according to some criteria. The idea of using scaling functions for input and output variables is to normalize the universe of discourse in which membership functions were defined.

According to *De Jong* [76.4]:

the common denominator in most learning systems is their capability of making structural changes to themselves over time with the intent of improving performance on tasks defined by the environment, discovering and subsequently exploiting interesting concepts, or improving the consistency and generality of internal knowledge structures.

Hence, it is important to have a clear understanding of the strengths and limitations of a particular learning system, to achieve a precise characterization of all the

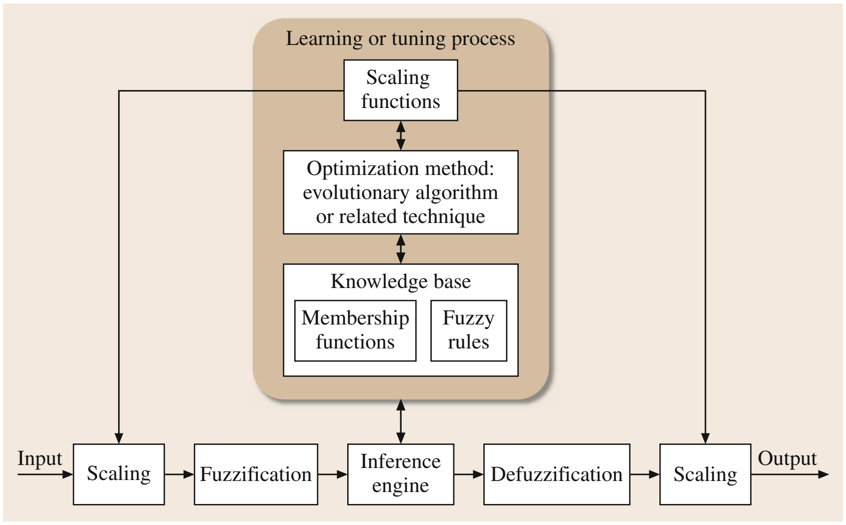

Fig. 76.1 General structure of an evolutionary fuzzy rule-based system

permitted structural changes and how they are going to be made.

De Jong sets three different levels of complexity where the GA can perform legal structural changes in following a goal, these are [76.4]:

1. By changing critical parameters' values
2. By changing key data structures
3. By changing the program itself with the idea of achieving effective behavioral changes in a task subsystem where a prominent representative of this branch is the *learning production-systems program*.

A good reason behind the success of production systems in machine learning is due to the fact that they have a representation of knowledge that can simultaneously support two kinds of activities: (1) the knowledge can be treated as data that can be manipulated according to some criteria; (2) for a particular task, the knowledge can be used as an executable entity.

The two classical approaches for working with evolutionary FRBS (EFRBS) for a learning system are the Pittsburgh and Michigan approaches. Historically, in 1975 *Holland* [76.5] affirmed that a natural way to represent an entire rule set is to use a string, i. e., an individual; so, the population is formed by candidate rule sets, and to achieve evolution it is necessary to use selection and genetic operators to produce new generations of rule sets. This was the approach taken by De Jong at the University of Pittsburgh, hence the name of *Pittsburgh approach*. During the same period, Holland developed a model of cognition in which the members of population are individual rules, and the entire population is conformed with the rule set; this quickly became the *Michigan approach* [76.6, 7].

There are extensive pioneering and recent work about tuning and learning using FRBS most of them fall in some way in the Michigan or in the Pittsburgh approaches, for example, the supervised inductive algorithm [76.8, 9], the iterative rule learning approach [76.10], coverage-based genetic induction (COGIN) [76.11, 12], the relational genetic algorithm learner (REGAL) system [76.13], the compact fuzzy classification system [76.14], with applications to fuzzy control [76.15, 16], and about tuning type-2 fuzzy controllers [76.17–20].

The focus of this chapter is on evolving embedded fuzzy controllers; this subclassification reduces the number of related works; however, they are still a big quantity, since by an embedding system (ES), we can understand a combination of computer hardware (HW) and software (SW) devoted to a specific control function within a larger system. Typically, the HW of an ES can be a dedicated computer system, a microcontroller, a digital signal processor, or a FPGA-based system. If the SW of the ES is fixed, it is called firmware; because there are no strict boundaries between firmware and software, and the ES has the capability of being reprogrammed, the firmware can be low level and high level. Low-level firmware tells the hardware how to work and typically resides in a read only memory (ROM) or in a programmable logic array (PLA); high-level firmware can be updated, hence is usually set in a flash memory, and it is often considered software.

In the literature, there is extensive work on successful applications of type-1 and type-2 fuzzy systems; with regards to evolving embedded fuzzy systems, they were applied in a control mechanism for autonomous mobile robot navigation in real environments in [76.21]. For the sake of limiting more the content of this chapter, we have focused on EFRBSs to be implemented in an FPGA HW platform, with special emphasis on type-2 FRBSs. In this last category, with respect to type-1 FRBS took our attention to the following proposals: The development of an FPGA-based proportional-differential (PD) fuzzy look-up table controller [76.22], FPGA implementation of embedded fuzzy controllers for robotic applications [76.23], a non-fixed structure fuzzy logic controller is presented in [76.24], a flexible architecture to implement a fuzzy controller into an FPGA [76.25], a very simple method for tuning the input membership function (MF) for modifying the implemented FPGA controller response [76.26]; how to test and simulate the different stages of a FRBS for future implementation into an FPGA are explained in [76.27–29]. On type-1 EFRBS there are some works like: A reconfigurable hardware platform for evolving a fuzzy system by using a cooperative coevolutionary methodology [76.30], the tuning of input MFs for an incremental fuzzy PD controller using a GA [76.31]. In the type-2 FRBS category, the amount of reported work is less; representative work can be listed as follows: an architectural proposal of hardware-based interval type-2 fuzzy inference engine for FPGA is presented in [76.32], the use of parallel HW implementation using bespoke coprocessors handled by a soft-core processor of an interval type-2 fuzzy logic controller is explored in [76.33], a high-performance interval type-2 fuzzy inference system (IT2-FIS) that can achieve the four stages fuzzification, inference, KM-type reduction, and defuzzification in four clock cycles is shown in [76.34]; the same system is suitable for implementation in pipelines providing the complete IT2-FIS process in just one clock cycle.

This work deals with the development of evolving embedded type-1 and type-2 fuzzy controllers. In the chapter, a broad exploration of several ways to implement evolving embedded fuzzy controllers are presented. We choose to work with the Mamdani fuzzy controller proposal since it provides a highly flexible means to formulate knowledge.

The organization of this chapter is as follows. In Sect. 76.2 we present the basis of T1 and T2 FL to explain how to achieve the HW implementation of an FRBS. In Sect. 76.3 a brief description of the state of the art in hosting technology for high-performance embedded systems is given.

76.2 Type-1 and Type-2 Fuzzy Controllers

The type-2 fuzzy sets (T2FS) were developed with the aim of handling uncertainty in a better way than T1 FS does, since a T1FS has crisp grades of membership, whereas a T2FS has fuzzy grades of membership. An important point to note is that if all uncertainty disappears, a T2 FS can be reduced to a T1FS. A type-2 membership function (T2MF) is an FS that has primary and secondary membership values; the primary MF is a representation of an FS, and serves to create a linguistic representation of some concept with linguistic and random uncertainties with limited capabilities; the secondary MF allows capturing more about linguistic uncertainty than a T1MF.

There are two common ways to use a T2FS, the generalized T2FS (GT2), and the interval T2FS (IT2FS). The former has secondary membership grades of different values to represent more accurately the existing uncertainty; on the other hand, in an IT2FS the secondary membership value always takes the value of *1*. Unfortunately, to date for GT2 no one knows yet how to choose their best secondary MFs; moreover, this method introduces a lot of computations, making it inappropriate for current application in real-time (RT) systems, even those with small time constraints; in contrast, the calculations are easy to perform in an IT2FS.

A T2MF can be represented using a 3-D figure that is not as easy to sketch as a T1MF. A more common way to visualize a T2MF is to sketch its footprint of uncertainty (FOU) on the 2-D domain of the T2FS. We illustrate this concept in Fig. 76.2, where we show a vertical slice sketch of the FOU at the primary MF value x'; in the case of a GT2, in the right upper part of the figure, the secondary MF shows different height values of the GT2; in the case of an IT2F2, just below is the secondary MF with uniform values for the IT2FS. Note that the secondary values sit on top of its FOU.

Figure 76.3 shows the main components of a fuzzy logic system showing the differences between the T1 and T2 FC. For T1 systems, there are three components: fuzzifier, inference engine, and the defuzzifier which is

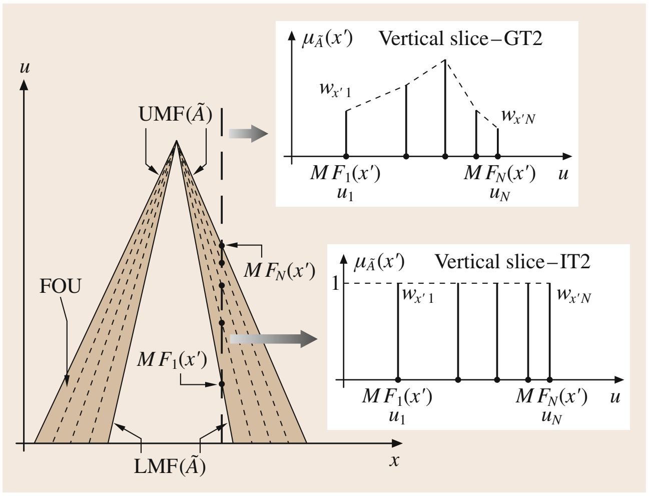

Fig. 76.2 Type-2 membership function. For the triangular MF the FOU is shown. The FOU is bounded by the upper part UMF($\tilde{A}$) and the lower part LMF($\tilde{A}$). A vertical slice at x' is illustrated. *Right, top*: secondary MF values for a generalized T2MF; *bottom*: secondary MF values of an IT2MF

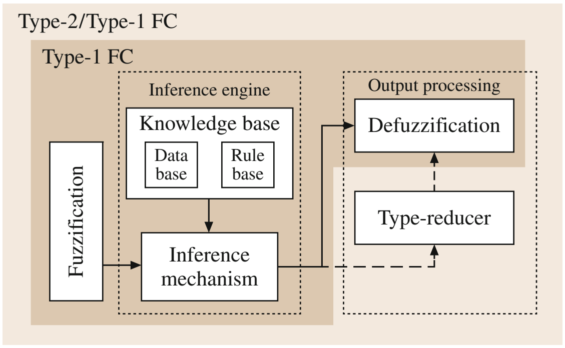

Fig. 76.3 Type-1 and type-2 FC. The T2FC at the output processing has the type reducer block

the only output processing unit; whereas for a T2 system there are four components, since the output processing has interconnected the type reducer (TR) block and the defuzzifier to form the output processing unit.

Ordinary fuzzy sets were developed by *Zadeh* in 1965 [76.35]; they are an extension of classical set theory where the concept of membership was extended to have various grades of membership on the real continuous interval $[0,1]$. The original idea was to use a fuzzy set (FS); i. e., a linguistic term to model a word; however, after almost 10 years, *Zadeh* introduced the concept of type-n FS as an extension of an ordinary FS (T1FS) with the idea of blurring the degrees of membership values [76.36].

T1FSs have been demonstrated to work efficiently in many applications; most of them use the mathematics of fuzzy sets but lose the focus on words that are mainly used in the context to represent a function which is more mathematical than linguistic [76.37].

A T1FS is a set of ordered pairs represented by (76.1) [76.38],

$$A = \{(x, \mu_A(x)) | x \in X\} , \tag{76.1}$$

where each element is mapped to $[0,1]$ by its MF μ_A, where $[0,1]$ means real numbers between 0 and 1, including the values 0 and 1,

$$\mu_A(x) : X \to [0,1] . \tag{76.2}$$

A pointwise definition of a T2FS is given as follows, $\widetilde{A}$ is characterized by a T2MF $\mu_{\widetilde{A}}(x,u)$, where $x \in X$ and $u \in J_x \subseteq [0,1]$, i. e. [76.39],

$$\widetilde{A} = \left\{(x,u), \mu_{\widetilde{A}}(x,u) | \forall x \in X, \forall u \in J_x \subseteq [0,1]\right\} , \tag{76.3}$$

where $0 \le \mu_{\widetilde{A}}(x,u) \le 1$.

Another way to express $\widetilde{A}$ is

$$\widetilde{A} = \int\limits_{x \in X} \int\limits_{u \in J_x} \mu_{\widetilde{A}}(x,u) / (x,u) \quad J_x \subseteq [0,1] , \tag{76.4}$$

where $\int\int$ denote the union over all admissible input variables x' and u'. For discrete universes of discourse $\int$ is replaced by $\sum$ [76.39]. In fact, $J_x \subseteq [0,1]$ represents the primary membership of $x \in X$ and $\mu_{\widetilde{A}}(x',u)$ is a T1FS known as the secondary set. Hence, a T2MF can be any subset in [0,1], the primary membership, and corresponding to each primary membership, there is a secondary membership (which can also be in [0,1]) that defines the uncertainty for the primary membership.

When $\mu_{\widetilde{A}}(x,u) = 1$, where $x \in X$ and $u \in J_x \subseteq [0,1]$, we have the IT2MF shown in Fig. 76.2. The uniform shading for the FOU represents the entire IT2FS and it can be described in terms of an upper membership function and a lower membership function

$$\bar{\mu}_{\widetilde{A}}(x) = \overline{\text{FOU}(\widetilde{A})} \ \forall x \in X , \tag{76.5}$$

$$\underline{\mu}_{\widetilde{A}}(x) = \underline{\text{FOU}(\widetilde{A})} \ \forall x \in X . \tag{76.6}$$

Figure 76.2 shows an IT2MF, the shadow region is the FOU. At the points x_1 and x_2 are the primary MFs J_{x_1} and J_{x_2}, and the corresponding secondary MFs $\mu_{\widetilde{A}}(x_1)$ and $\mu_{\widetilde{A}}(x_2)$ are also shown.

The basics and principles of fuzzy logic do not change from T1FSs to T2FSs [76.37, 40, 41], they are independent of the nature of the membership functions, and in general, will not change for any type-n. When a FIS uses at least one type-2 fuzzy set, it is a type-2 FIS.

In this chapter we based our study on IT2FSs, so the IT2 FIS can be seen as a mapping from the inputs to the output and it can be interpreted quantitatively as $Y = f(X)$, where $X = \{x_1, x_2, \ldots, x_n\}$ are the inputs to the IT2 FIS f, and $Y = \{y_1, y_2, \ldots, y_n\}$ are the defuzzified outputs. These concepts can be represented by rules of the form

$$\text{If } x_1 \text{ is } \widetilde{F}_1 \text{ and } \ldots \text{ and } x_p \text{ is } \widetilde{F}_p, \text{ then } y \text{ is } \widetilde{G} . \tag{76.7}$$

In a T1FC, where the output sets are T1FS, the defuzzification produces a number, which is in some sense a crisp representation of the combined output sets. In the T2 case, the output sets are T2, so the extended defuzzification operation is necessary to get T1FS at the output. Since this operation converts T2 output sets to a T1FS, it is called type reduction, and the T1FS is called a type-reduced set, which may then be defuzzified to obtain a single crisp number.

The TR stage is the most computationally expensive stage of the T2FC; therefore, several proposals to improve this stage have been developed. One of the first proposals was the iterative procedure known as the Karnik–Mendel (KM) algorithm.

In general, all the proposals can be classified into two big groups. Group I embraces all the algorithmic improvements and Group II all the hardware improvements, as follows [76.42]:

1. Improvements to software algorithms, where the dominant idea is to reduce computational cost of IT2-FIS based on algorithmic improvements. This group can be subdivided into three subgroups.

(a) Enhancements to the KM TR algorithm. As the classification's name claims, the aim is to improve the original KM TR algorithm directly, to speed it up. The best known algorithms in this classification are:

i. Enhanced KM (EKM) algorithms. They have three improvements over the original KM algorithm. First, a better initialization is used to reduce the number of iterations. Second, the termination condition of the iterations is changed to remove unnecessary iterations (one). Finally, a subtle computing technique is used to reduce the computational cost of each iteration.

ii. The enhanced Karnik–Mendel algorithm with new initialization (EKMANI) [76.43]. It computes the generalized centroid of general T2FS. It is based on the observation that for two alpha-planes close to each other, the centroids of the two resulting IT2FSs are also closed to each other. So, it may be advantageous to use the switch points obtained from the previous alpha-plane to initialize the switch points in the current alpha-plane. Although EKMANI was primarily intended for computing the generalized centroid, it may also be used in the TR of IT2-FIS, because usually the output of an IT2-FIS changes only a small amount at each step.

iii. The iterative algorithm with stop condition (IASC). This was proposed by *Melgarejo* et al. [76.44] and is based on the analysis of behavior of the firing strengths.

iv. The enhaced IASC [76.45] is an improvement of the IASC.

v. Enhanced opposite directions searching (EODS), which is a proposal to speed up KM algorithms. The aim is to search in both directions simultaneously, and in each iteration the points L and R are the switching points.

(b) Alternative TR algorithms. Unlike iterative KM algorithms, most alternative TR algorithms have a closed-form representation. Usually, they are faster than KM algorithms. Two representative examples are:

i. The Gorzalczany method. A polygon using the firing strengths $[\underline{f}^n, \overline{f}^n]$ and $[(y^1, y^n)$, which can be viewed as an IT2FS. It computes an approximate membership value for each point. Here, $\underline{y}^n = \overline{y}^n = y^n$, for $n = 1, 2, 3 \ldots, N$.

$$\mu(y) = \frac{\underline{f} + \overline{f}}{2} \cdot [1 - (\overline{f} - \underline{f})] \,, \qquad (76.8)$$

where $\overline{f} - \underline{f}$ is called the bandwidth. Then the defuzzified output can be computed as

$$y_G = \arg\max_y \mu(y) \,. \qquad (76.9)$$

ii. The Wu–Tan (WT) method. It searches an equivalent T1FS. The centroid method is applied to obtain the defuzzification. This is the faster method in this category.

2. Hardware implementation. The main idea is to take advantage of the intrinsic parallelism of the hardware and/or combinations of hardware and parallel programming. Here, we divided this group into four main approaches that embrace the existing proposals of reducing the computational time of the type reduction stage by the use of parallelism at different levels.

(a) The use of multiprocessor systems, including multicore systems that enable the same benefits at a reduced cost. In this category are personal and industrial computers with processors such as the Intel Pentium Core Processor family, which includes the Intel Core i3, i5 and i7; the AMD Quad-Core Optetron, the AMD Phenom X4 Quad-Core processors, multicore microcontrollers such as the Propeller P8X32A from Parallax, or the F28M35Hx of the Concerto Microcontrollers family of Texas Instruments. Multicore processors also can be implemented into FPGAs.

(b) The use of a general-purpose GPU (GPGPU), and compute unified device architecture (CUDA). In general, GPU provides a new way to perform high performance computing on hardware. In particular IT2FCs can take the most advantage of this technology because their complexity. Traditionally, before the development of the CUDA technology, the programming was achieved by translating a computational procedure into a graphic format with the idea to execute it using the standard graphic pipeline; a process known as encoding data into a texture format. The CUDA technology of NVIDIA offers a parallel programming model for GPUs that does not require the use of a graphic application programming interface (API), such as OpenGL [76.46].

(c) The use of FPGAs. This approach offers the best processing speed and flexibility. One of the main advantages is that the developer can determine the desired parallelism grade by a trade-off analysis. Moreover, this technology allows us to use the strength of all platforms in tight integration to provide the large performance available at the present time. It is possible to have a standalone T1/IT2FC, or to integrate the same T1/T2FC as a coprocessor as part of a high performance computing system.
(d) The use of ASICs. The T1/T2FC is factory integrated using complementary metal-oxide-semiconductor (CMOS) technology. The main advantages are that they are cheaper than FPGAs. Differently to FPGA technology, ASIC solutions are not field reprogrammable.

A system based on an FPGA platform allows us to program all the Group I algorithms since modern FPGAs have embedded hard and/or soft processors; this kind of system can be programmed using high-level languages such as C/C++ and also they can incorporate operating systems such as Linux. On the other hand, T1/T2 FC hardware implementations have the advantage of providing competitive faster systems in comparison to ASIC systems and the in field reconfigurability.

76.3 Host Technology

Until the beginnings of this century, general-purpose computers with a single-core processor were the systems of choice for high-performance computing (HPC) for many applications; they replaced existing big and expensive computer architectures [76.47]. In 2001, IBM introduced a reduced intstruction set computer (RISC) microarchitecture named POEWER4 (performance optimization with enhanced RISC) [76.48]. This was the first dual core processor embedded into a single die, and subsequently other companies introduced different multicore microprocessor architectures to the market, such as the Arm Cortex A9 [76.49], Sparc64 [76.50], Intel and AMD Quad Core processors, Intel i7 processors, and others [76.51]. These developments, together with the rapid development of GPUs that offer massively parallel architectures to develop high-performance software, are an attractive choice for professionals, scientists, and researchers interested in speeding up applications. Undoubtedly, the use of a generic computer with GPU technology has many advantages for implementing an embedded learning fuzzy system [76.46], and disadvantages are mainly related to size and power consumption. A solution to the aforementioned problems is the use of application specific integrated circuits (ASICs) fuzzy processors [76.52–54], or reprogrammable hardware based on microcontrollers and/or FPGAs.

The orientation of this paper is towards tuning and learning using FRBS for embedded applications; for now, we are going to focus on FPGAs and ASIC technology [76.55], since they provide the best level of parallelization. Both families of devices provide characteristics for HPC that the other options cannot. Each technology has its own advantages and disadvantages, which are narrowing down due to recent developments. In general, ASICs are integrated circuits that are designed to implement a single application directly in fixed hardware; therefore, they are very specialized for solving a particular problem. The costs of ASIC implementations are reduced for high volumes; they are faster and consume less power; it is possible to implement analog circuitry, as well as mixed signal design, but the time to market can take a year or more. There are several design issues that need to be carried out that do not need to be achieved using FPGAs, the tools for development are very expensive. On the other hand, FPGAs can be introduced to the market very fast since the user only needs a personal computer and low-cost hardware to burn the HDL (HDL) code to the FPGA before it is ready to work. They can be remotely updated with new software since they are field reprogrammable. They have specific dedicated hardware such as blocks of random access memory (RAM); they also provide high-speed programmable I/O, hardware multipliers for digital signal processing (DSP), intellectual property (IP) cores, microprocessors in the form of hard cores (factory implemented) such as PowerPC and ARM for Xilinx, or Microblaze and Nios softcore (user implemented) for Xilinx and Altera, respectively. They can have built-in analog digital converters (ADCs). The synthesis process is easier. A significant point is that the HDL tested code developed for FPGAs may be used in the design process of an ASIC.

There are three main disadvantages of the FPGAs versus ASICs, they are: FPGA devices consume more power than ASICs, it is necessary to use the resources available in the FPGA which can limit the design, and they are good for low-quantity production. To overcome these disadvantages it is very important to achieve optimized designs, which can only be attained by coding efficient algorithms.

During the last decade, there has been an increasing interest in evolving hardware by the use of evolutionary computations applied to an embedded digital system. Although different custom chips have been proposed for this plan, the most popular device is the FPGA because its architecture is designed for general-purpose commercial applications. New FGAs allow modification of part of the programmed logic, or add new logic at the running time. This feature is known as dynamic or active reconfiguration, and because in an FPGA we can combine a multiprocessor system and coprocessors, FPGAs are very attractive for implementing evolvable hardware algorithms. Therefore, in the next sections, we shall put special emphasis on multiprocessor systems and FPGAs.

76.4 Hardware Implementation Approaches

In this section, an overview of the three main lines of attack to do a hardware implementation of an intelligent system is given.

76.4.1 Multiprocessor Systems

Multiprocessor systems consist of multiple processors residing within one system; they have been available for many years. Multicore processors have equivalent benefits to multiprocessors at a lower cost; they are integrated in the same electronic component. At the present time, most modern computer systems have many processors that can be single core or multicore processors; therefore, we can have three different layouts for multiprocessing; a multicore system, a multiprocessor system, and a multiprocessor/multicore system.

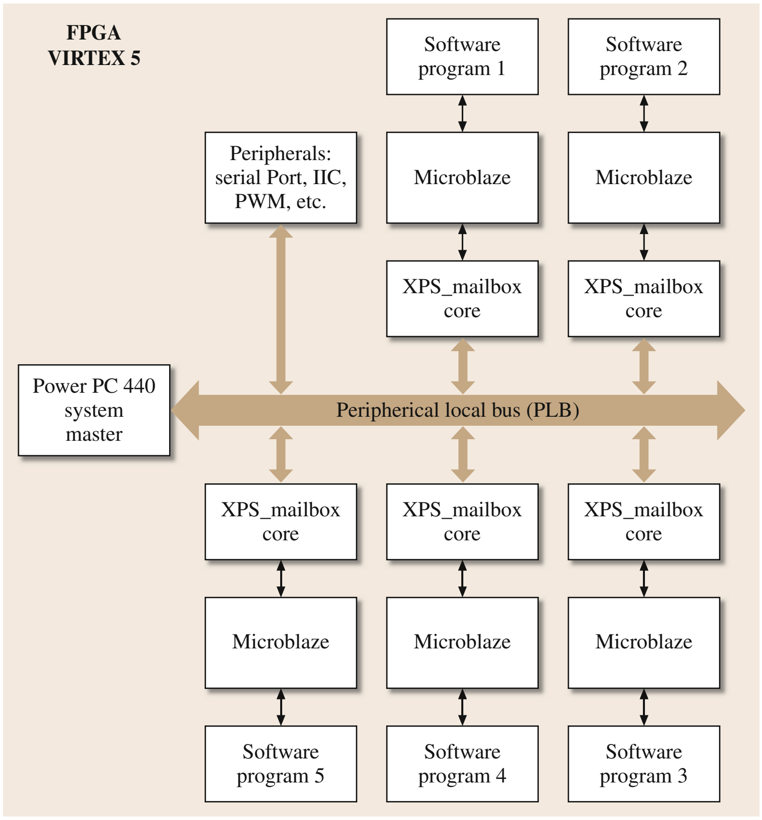

Fig. 76.4 Multicore system embedded into an FPGA. Embedded is a hard-processor PowerPC440 and five MicroBlaze soft-processors. In this system we can process an EA using the island model

Fig. 76.5 The whole embedded evolutionary IT2FC implemented in the program memory of the multiprocessor system, similarly as in a desktop computer ▶

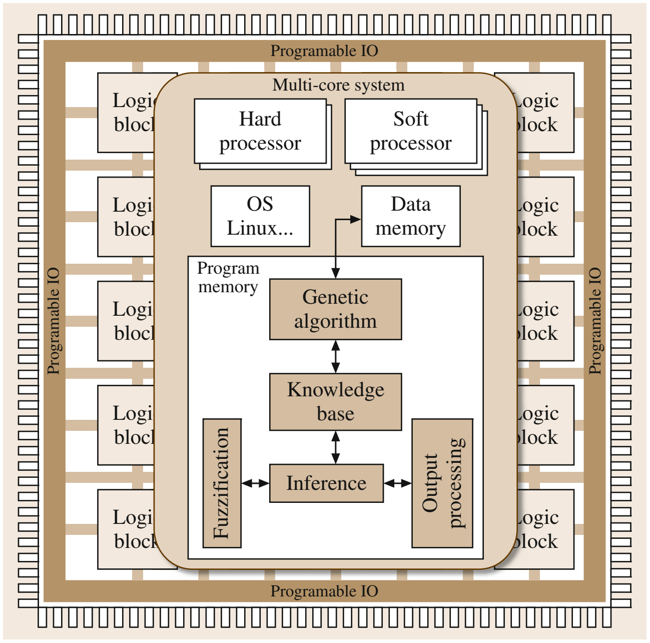

Figure 76.4 shows a multicore system embedded into a Virtex 5 FPGA XC5VFX70; it has the capacity to integrate a distributed multicore system with a hard-processor PowerPC 440 as the master, five Microblaze 32-bit soft-processor slaves, coprocessors, and peripherals. The FPGA capacity to integrate devices is, of course, limited by the size of the FPGA. Figure 76.5 shows the full implementation in the program memory of the multiprocessor system.

76.4.2 Implementations into FPGAs

The architecture of FPGAs offers massive parallelism because they are composed of a large array of configurable logic blocks (CLBs), digital signal processing blocks (DSPs), block RAM, and input/output blocks (IOBs). Similarly, to a processor's arithmetic unit (ALU), CLBs and DSPs can be programmed to perform arithmetic and logic operations like compare, add/subtract, multiply, divide, etc. In a processor, ALU architectures are fixed because they have been designed in a general-purpose manner to execute various operations. CLBs can be programmed using just the operations that are needed by the application, which results in increased computation efficiency. Therefore, an FPGA consists of a set of programmable logic cells manufactured into the device according to a connection paradigm to build an array of computing resources; the resulting arrangement can be classified into four categories: symmetrical array, row-based, hierarchy-based, and sets of gates [76.56]. Figure 76.6 shows a symmetrical array-based FPGA that consists of a two-dimensional array of logic blocks immersed in a set of vertical and horizontal lines; examples of FPGAs in this category are Spartan and Virtex from Xilinx, and Atmel AT40K. In Fig. 76.6 three main parts can be identified: a set of programmable logic cells also called logic blocks (LBs) or configurable logic blocks (CLBs), a programmable interconnection network, and a set of input and output cells around the device.

Embedded programmable logic devices usually integrate one or several processor cores, programmable logic and memory on the same chip (an FPGA) [76.56]. Developments in the field of FPGA have been very amazing in the last two decades, and for this reason, FPGAs have moved from tiny devices with a few thousand gates that were used in small applications such as

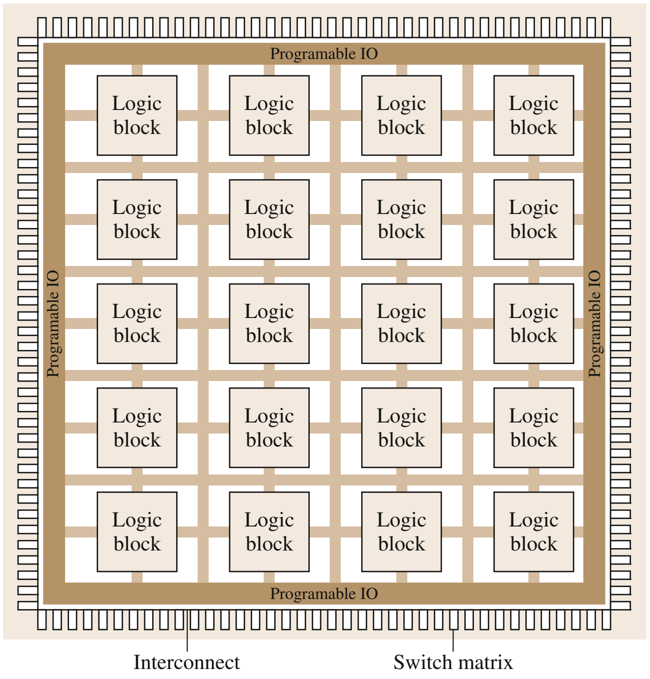

Fig. 76.6 Symmetric array-based FPGA architecture *island style*

finite state machines, glue-logic for complex devices, and very limited CPUs. In a 10-year period of time, a 200% growth rate in the capacity of Xilinx FPGAs

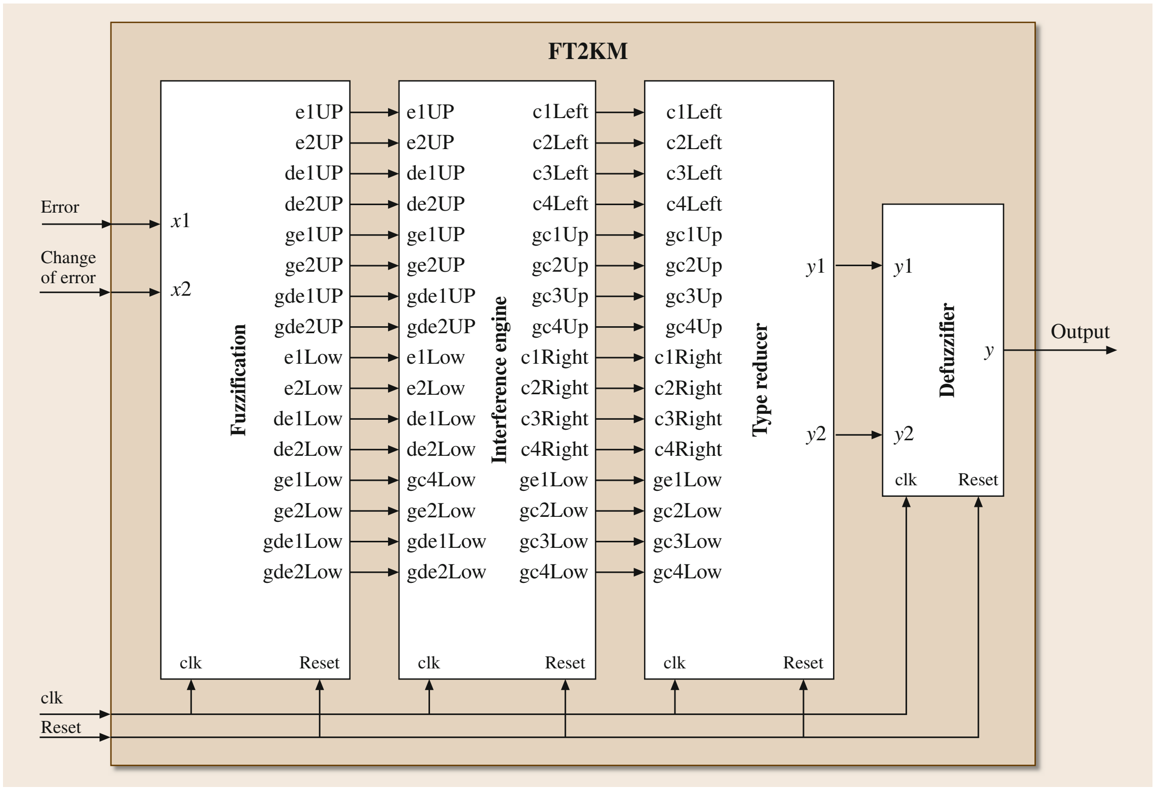

Fig. 76.7 IT2FC design entity (FT2KM). This top-level module contains instances of the four fuzzy controller submodules

devices was observed, a 50% reduction rate in power consumption, and prices also show a significant decrease rate. Other FPGA vendors, such as ACTEL, and ALTERA show similar developments, and this trend still continues. These developments, together with the progress in development tools that include software and low-cost evaluation boards, have boosted the acceptance of FPGAs for different technological applications.

Development Flow

The development flow of an FPGA-based system consists of the following major steps:

1. Write in VHDL the code that describes the systems' logic; usually a top-down and bottom-up methodology is used. For example, to design an IT2FC, we need to achieve the following procedure:
   (a) Describe the design entity where the designer defines the input and output of the top VHDL module. The idea is to present the complex object in different hierarchical levels of abstraction. For our example, the top design entity is *FT2KM*.
   (b) Once the design entity has been defined, it is required to define its architecture, where the description of the design entity is given; in this step, we define its behavior, its structure, or a mixture of both. For the case of the IT2 FLS, we define the system's internal behavior, so we determined the necessity to achieve a logic design formed by four interconnected modules: fuzzification, inference engine, type reduction, and defuzzification. The VHDL circuits (submodules) are described using a register transfer logic (RTL) sequence, since we can divide the functionality in a sequence of steps. At each step, the circuit achieves a task consisting in data transference between registers and evaluation of some conditions in order to go to the next step; in other words, each VHDL module (design entity) can be divided into two areas: data and control. Each of the four modules needs

to be conceptualized, so we need to define its own design entity and, therefore, its particular architecture as well interconnections with internal modules. This process is achieved when we have reached the last system component.

(c) Integrate the system. It is necessary to create a main design entity (top level) that integrates the submodules defining their interconnections. In Fig. 76.7 the integration of the four modules is shown.

2. Develop the test bench in VHDL and perform RTL simulations for each submodule of the main design entity. It is necessary to achieve timing and functional simulations to create reliable internal design entities.
3. Perform synthesis and implementation. In the synthesis process, the software transforms the VHDL constructs to generic gate-level components, such as simple logic gates and flip-flops. The implementation process is composed of three small subprocesses: translate, map, and place, and route. In the translate process the multiple design files of a project are merged into a single netlist. The map process maps the generic gates in the netlist to the FPGA's logic cells and IOBs, this process is also known as technology mapping. In the place and route process, using the physical layout inside the FPGA chip, the process places the cells in physical locations and determines the routes to connect diverse signals. In the Xilinx flow, the static timing analysis performed at the end of the implantation process determines various timing parameters such as maximal clock frequency and maximal propagation delay [76.57].
4. Generate the programming file and download it to the FPGA. According to the final netlist a configuration file is generated, which is downloaded to the FPGA serially.
5. Test the design entity using a simulation program such as Simulink of Matlab and the Xilinx system generator (XSG) for Xilinx devices. The idea here is first to plot the surface control in order to analyze the general behavior of the design (a controller in our example), and second to integrate the design entity as a block of the desired system to be controlled. Although, this fifth step, is not in the current literature of logic design for FPGA implementation, it is the authors's recommendation since we have experienced good results following this practice.

Using the design entity FT2KM.vhd, which was created and tested using the aforementioned development flow, we can integrate it an FPGA in two ways:

1. As a standalone system. Here, we mean an independent system that does not require the support of any microprocessor to work, the system itself is a specialized circuit that can produce the desired output. The IT2FC is implemented using the FPGA flow design; therefore, it is programmed using the complete development flow for a specific application.
2. As a coprocessor. The coprocessor performs specialized functions in such a way that the main system processor cannot perform as well and faster. For IT2FCs, given an input, the time to produce an output is big enough to achieve an adequate control of many plants when the IT2FC is programmed using high-level language, even we have used a parallel programming paradigm. Since a coprocessor is a dedicated circuit designed to offload the main processor, and the FPGA can offer parallelism on the circuit level, the designer of the IT2FC coprocessor can have control of the controller performance. The coprocessor can be physically separated, i. e., in a different FPGA circuit (or module), or it can be part of the system, in the same FPGA circuit. In this work, we show two methods to develop a system with an IT2FC as a coprocessor. In both methods, we consider that we have a tested IT2FC design entity. In the first case, we shall use the *FT2KM* design entity to incorporate the fuzzy controller as a coprocessor of an ARM processor into an FPGA Fusion. In the second case, we are going to create the IT2FC IP core using the Xilinx Platform Studio; the core will serve as a coprocessor of the MicroBlaze processor embedded into a Spartan 6 FPGA.

76.5 Development of a Standalone IT2FC

Figure 76.7 shows the top-level design entity (FT2KM) of the IT2FC and its components (submodules) for FPGA implementation. The entity codification of the top-level entity and its components are given in Sect. 76.5.1. All stages include the clock (clk) and reset (rst) signals. In the defuzzifier, we have included these two signals to illustrate that a full process takes only four clock cycles, one for each stage. In practice, we did not add these two signals, since when we used it as a coprocessor, in order to incorporate it to the system, one 8-bit data latch is added at the output. For a detailed description of the IT2FC stages consult [76.34].

The fuzzification stage has two input variables, x_1 and x_2. This module contains a fuzzifier for the upper MFs, and another for the lower MFs of the IT2FC. For the upper part, for the first input x_1, considering that a crisp value can be fuzzified by two MFs because it may have membership values in two contiguous T2MFs, the linguistic terms are assigned to the VHDL variables $e_{1\text{up}}$ and $e_{2\text{up}}$, and their upper membership values are $ge_{1\text{up}}$ and $ge_{2\text{up}}$. For the second input x_2, the linguistic terms are assigned to the VHDL variables $de_{1\text{up}}$ and $de_{2\text{up}}$, and $gde_{1\text{up}}$ and $gde_{2\text{up}}$ are the upper membership values. The lower part of the fuzzifier is similar; for example, for the input variable x_1 the VHDL assigned variables are $e_{1\text{low}}$ and $e_{2\text{low}}$, and their lower MF values are $ge_{1\text{low}}$ and $ge_{2\text{low}}$, etc. The fuzzification stage entity *only needs one clock cycle* to perform the fuzzification. These eight variables are the inputs of the inference engine stage [76.58].

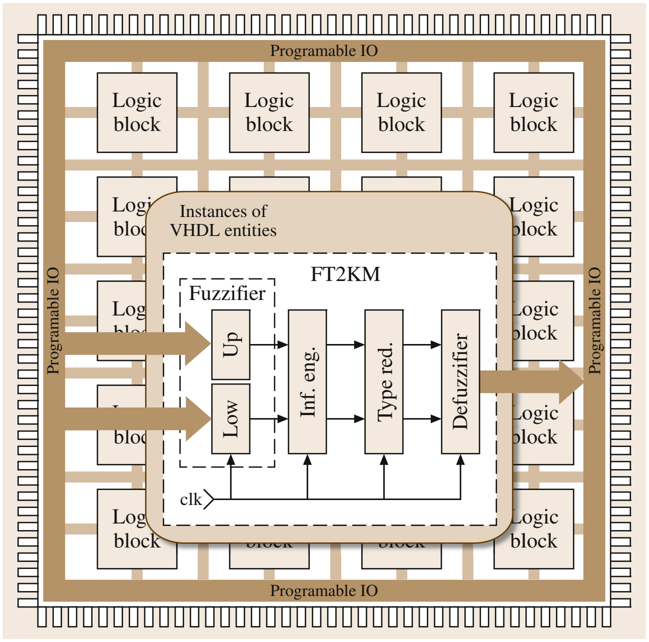

Fig. 76.8 A standalone IT2FC is embedded into an FPGA. The fuzzifier reads the inputs directly from the FPGA terminals. The defuzzifier sends the crisp output to the FPGA terminals. The system may be embedded in the static region or in the reprogrammable region

The inference engine is divided into two parallel inference engine entities IEEup is used to manage the upper bound of the IT2FC, and IEElow for the lower bound of the IT2FCs. Each entity has eight inputs from the corresponding fuzzifier stage, and eight outputs; four belong to the output linguistic terms, the rest correspond to their firing strengths. All the inputs enter into a parallel selection VHDL process, the circuits into the process are placed in parallel; the degree of parallelism can be tailored by an adequate codification style. In our case, all the rules are processed in parallel and the eight outputs of each inference engine section (upper bound and lower bound) are obtained at the same time because the *clk* signal synchronizes the process, hence *this stage needs only one clock cycle to perform a whole inference and provide the output to the next stage*. In the upper bound, the four antecedents are formed at the same time, for example, for the first rule, the antecedent is formed using the concatenation operator &, so it looks like *ante* $:= e1$ & $de1$. Each antecedent can address up to four rules and depending on the combination, one of the four rules is chosen; the upper inference engine output provides the active consequents and its firing strengths. The lower bound of the inference engine is treated in the same way [76.59].

At the input of the TR, we have the equivalent values of the pre-computed y_l^i, i. e., the linguistic terms of the active consequents ($C_{1\text{left}}$, $C_{2\text{left}}$, $C_{3\text{left}}$, and $C_{4\text{left}}$), the upper firing strength ($gc_{1\text{up}}$, $gc_{2\text{up}}$, $gc_{3\text{up}}$, and $gc_{4\text{up}}$), in addition to the equivalent values of the pre-computed y_r^i ($C_{1\text{right}}$, $C_{2\text{right}}$, $C_{3\text{right}}$, and $C_{4\text{right}}$), the lower firing strength ($gc_{1\text{low}}$, $gc_{2\text{low}}$, $gc_{3\text{low}}$, and $gc_{4\text{low}}$) [76.60]. All the above-mentioned signals go to a parallel selection process to perform the KM algorithm [76.39]. There are parallel blocks to obtain the average of the upper and lower firing strength for the active consequents, required to obtain the average of the y_r and y_l; a block to obtain the different defuzzified values of y_r and y_l; parallel comparator blocks to obtain the final result of y_r and y_l [76.61].

The final result of the IT2FC is obtained using the defuzzification block, which computes the average of the y_r and y_l, and produces the only output y.

76.5.1 Development of the IT2 FT2KM Design Entity

Figure 76.8 shows the implementation of a static IT2FC that can work as a standalone system. By static, we mean that the only way to reconfigure (modify) the FC is to stop the application and uploading the whole configuration bit file (bitstream). In this system, the inputs of the fuzzifier and the defuzzifier output are connected directly to the FPGA terminals. The assignment of the terminals is achieved in accordance with the internal architecture of the chosen FPGA. Hence, it is necessary to provide to the Xilinx Integrated Synthesis Environment (ISE) program, special instructions (constraints) to carry through the synthesis process. They are generally placed in the user constraint file (UCF), although they may exist in the HDL code. In general, constraints are instructions that are given to the FPGA implementation tools with the purpose of directing the mapping, placement, timing, or other guidelines for the implementation tools to follow while processing an FPGA design. In Fig. 76.7 the overall entity of design of the IT2FC (FTK2M) was defined as follows,

```
entity FT2KM is
  Port(clk, reset    : in std_logic;
        x1, x2       : in std_logic_vector(8 downto 1);
        y            : out std_logic_vector (8 downto 1)
    );
end FT2KM;
```

The architecture of FT2KM has four components, and all of them have two common input ports: clock (clk), and reset (rst). All ports in an entity are signals by default. This is important since a signal serves to pass values in and out of the circuit; a signal represents circuit interconnects (wires). A component is a simple piece of customized code formed by entities as corresponding architectures, as well as library declarations. To allow a hierarchical design, each component must be declared before been used by another circuit, and to use a component it is neccesary to instatiate it first. In this approach the components are:

1. The component labeled as *fuzzyUpLw*. It is the T2 fuzzifier that consists of one fuzzifier for the upper MF of the FOU and one for the lower MF. It has two input ports *x1* and *x2*; these are 16: *e1Up* to *de2Low*.

   ```
   component fuzzyUpLw is
      port(clk, reset : in std_logic;
           x1, x2, ge1Up, ge2Up, gde1Up, gde2Up :
               in std_logic_vector(n downto 1);
           e1Up, e2Up, de1Up, de2Up, e1Low,
               e2Low, de1Low,
           de2Low : out std_logic_vector(3 downto 1);
           ge1Up, ge2Up, gde1Up, gde2Up, ge1Low,
               ge2Low, gde1Low,
           gde2Low : out std_logic_vector(n downto 1);
      );
   end component;
   ```

The instantiation of this component is achieved using nominal mapping and the name of this instance is *fuzt2*. Note that ports *clk*, *reset*, and *x1* and *x2* are mapped (connected) directly to the entity of design FT2KM, since as we explained before, all ports are signals by default, which represent wires. The piece of code that defines the instantiation of the *fuzzyUpLw* component is as follows,

```
fuzt2 : fuzzyUpLw port map(
          clk => clk, reset=> reset, x1 => x1, x2 => x2,
          e1Up => e1upsig, e2Up => e2upsig, de1Up => de1upsig,
          de2Up => de2upsig, ge1Up => ge1upsig, ge2Up => ge2upsig,
          gde1Up => gde1upsig, gde2Up => gde2upsig, e1Low  => e1lowsig,
          e2Low  => e2lowsig, de1Low => de1lowsig, de2Low => de2lowsig,
          ge1Low => ge1lowsig, ge2Low => ge2lowsig, gde1Low => gde1lowsig,
          gde2Low => gde2lowsig
        );
```

2. The component *Infer_type_2* corresponds to the T2 inference the controller. It has 16 inputs that match to the 16 outputs of the fuzzification stage. This component has 16 outputs to be connected to the type reduction stage. The piece of code to include this component is:

```
component Infer_type_2 is
   port(rst, clk : in std_logic;
        e1, e2, de1, de2, e1_2, e2_2, de1_2, de2_2 : in  STD_LOGIC_VECTOR (m downto 1);
        g_e1, g_e2, g_de1, g_de2, g_e1_2, g_e2_2,
        g_de1_2, g_de2_2 : in  STD_LOGIC_VECTOR (n downto 1);
        c1, c2, c3, c4, c1_2, c2_2, c3_2, c4_2 : out  STD_LOGIC_VECTOR (m downto 1);
        gc1_2, gc2_2, gc3_2, gc4_2, gc1, gc2, gc3, gc4 : out  STD_LOGIC_VECTOR (n downto 1);
    );
end component;
```

This component is instantiated with the name *Infer_type_2* as follows,

```
inft2: Infer_type_2 port map(
         rst => reset, clk => clk, e1 => e1upsig, e2 => e2upsig, de1 => de1upsig,
         de2 => de2upsig, g_e1 => ge1upsig, g_e2 => ge2upsig, g_de1 => gde1upsig,
         g_de2 => gde2upsig, e1_2 => e1lowsig, e2_2 => e2lowsig, de1_2 => de1lowsig,
         de2_2 => de2lowsig, g_e1_2 => ge1lowsig, g_e2_2 => ge2lowsig, g_de1_2 => gde1lowsig,
         g_de2_2 => gde2lowsig, c1 => c1sig, c2 => c2sig, c3  => c3sig, c4  => c4sig,
         gc1 => gc1sig, gc2 => gc2sig, gc3 => gc3sig, gc4 => gc4sig, c1_2 => c12sig,
          c2_2 => c22sig, c3_2 => c32sig, c4_2 => c42sig, gc1_2 => gc12sig,
          gc2_2 => gc22sig, gc3_2 => gc32sig, gc4_2 => gc42sig
       );
```

To connect the instances *fuzt2* and *Infer_type_2* it is necessary to define some signals (wires),

```
    signal e1upsig, e2upsig, de1upsig, de2upsig : std_logic_vector (m-1 downto 0);
    signal ge1upsig, ge2upsig, gde1upsig, gde2upsig :std_logic_vector (7 downto 0);
    signal e1lowsig, e2lowsig, de1lowsig, de2lowsig :std_logic_vector (m-1 downto 0);
    signal ge1lowsig, ge2lowsig, gde1lowsig, gde2lowsig : std_logic_vector (7 downto 0);
```

3. The component *TypeRed* corresponds to the type reduction stage of the T2FC. It has 16 inputs that should connect the inference engine's outputs and it has two outputs *yr* and *yl* that should be connected to the deffuzifier through signals, once both have been instantiated. The piece of code to include this component is:

```
component TypeRed is
    Port (clk, rst : in std_logic;
          c1,  c2,  c3, c4, c1_2, c2_2, c3_2,  c4_2 : in  STD_LOGIC_VECTOR (3 downto 1);
          gc1, gc2, gc3, gc4, gc1_2, gc2_2,  gc3_2, gc4_2 : in  STD_LOGIC_VECTOR (7 downto 0);
          yl, yr : out std_logic_vector (8 downto 1));
end component;
```

   This component is instantiated with the name *trkm* as follows,

```
inft2: Infer_type_2 port map(
         rst => reset, clk => clk, e1 => e1upsig, e2 => e2upsig, de1 => de1upsig,
         de2 => de2upsig, g_e1 => ge1upsig, g_e2 => ge2upsig, g_de1 => gde1upsig,
         g_de2 => gde2upsig, e1_2 => e1lowsig, e2_2 => e2lowsig, de1_2 => de1lowsig,
         de2_2 => de2lowsig, g_e1_2 => ge1lowsig, g_e2_2 => ge2lowsig, g_de1_2 => gde1lowsig,
         g_de2_2 => gde2lowsig, c1 => c1sig, c2 => c2sig, c3  => c3sig, c4  => c4sig,
         gc1 => gc1sig, gc2 => gc2sig, gc3 => gc3sig, gc4 => gc4sig, c1_2 => c12sig,
          c2_2 => c22sig, c3_2 => c32sig, c4_2 => c42sig, gc1_2 => gc12sig,
          gc2_2 => gc22sig, gc3_2 => gc32sig, gc4_2 => gc42sig
       );
```

   The signals that connect the instance *Infer_type_2* to the instance *trkm* are

```
    signal c1sig, c2sig, c3sig, c4sig : std_logic_vector (m-1 downto 0);
    signal gc1sig, gc2sig, gc3sig, gc4sig : std_logic_vector (7 downto 0);
    signal c12sig, c22sig, c32sig, c42sig : std_logic_vector (m-1 downto 0);
    signal gc12sig, gc22sig, gc32sig, gc42sig :std_logic_vector (7 downto 0);
```

4. The last component *defit2* corresponds to the defuzzifier stage of the T2FLC. It has two inputs and one output.

```
component defit2 is
    Port ( yl, yr : in  std_logic_vector (n-1 downto 0);
           y  : out std_logic_vector (n-1 downto 0));
end component;
```

   This component is instantiated with the name *dfit2* as follows,

```
dfit2 : defit2 port map(yl => ylsig, yr => yrsig, y => y);
```

   We did not define any signal for the port *y* since it can be connected directly to the entity of design *FT2KM*. The instances *trkm* and *dfit2* are connected using the following signals,

```
signal ylsig, yrsig : std_logic_vector (n-1 downto 0);
```

This approach of implementing an IT2FC provides the faster response. The whole process consisting of fuzzification, inference, type reduction, and defuzzification is achieved in four clock cycles, which for a Spartan family implementation using a 50 MHz clock represents 80×10^{-9} s, and for a Virtex 5 FPGA-based system is 40×10^{-9} s.

76.6 Developing of IT2FC Coprocessors

The use of IT2FC embedded into an FPGA can certainly be the option that offers the best performance and flexibility. As we shall see, the best performance can be obtained when the embedded FC is used as standalone system. Unfortunately, this gain in performance can present some drawbacks; for example, for people who were not involved in the design process of the controller or who are not familiar with VHDL codification, or the code owners simply want to keep the codification secret. All these obstacles can be overcome by the use of IP cores. Next, we shall explain two methods of implementing IT2FC as coprocessors.

76.6.1 Integrating the IT2FC Through Internal Ports

In Fig. 76.9, we show a control system that integrates the FT2KM design entity embedded into the Actel Fusion FPGA [76.62] as a coprocessor of an ARM processor. This FPGA allows incorporating the soft processor ARM Cortex, as well as other IP cores to make a custom configuration. The embedded system contains the ARM processor, two memory blocks, timers, interrupt controller (IC), a Universal Asynchronous Receiver/Transmitter (UART) serial port, IIC, pulse width modulator/tachometer block, and a general-purpose input/output interface (GPIO) interfacing the FT2KM block. All the factory embedded components are soft IP cores. The FT2KM is a VHDL module that together with the GPIO form the Ft2km_core soft coprocessor, handled as an IP core; however, in this case, it is necessary to have the VHDL code. In the system, the IT2 coprocessor is composed of the GPIO and the FT2KM modules, forming the Ft2km_core. In the system, moreover, are a DC motor with a high-resolution quadrature optical encoder, the system's power supply, an H-bridge for power direction, a personal computer, and a digital display.

The Ft2km_core has six inputs and two outputs. The inputs are *error*, *c.error*, *ce*, *rst*, *w*, and *clk*. The 8-bit inputs *error* and *c.errror* are the controller input for the error and change of error values. *ce* input is used to enable/disable the fuzzy controller, the input *rst* restores all the internal registers of the IT2FC, and the input *w* allows starting a fuzzy inference cycle. The outputs are *out*, and *IRQ/RDY*; the first one is the crisp output value, which is 8-bit wide. *IRQ/RDY* is produced when the output data corresponding to the respective input is ready to be read. *IRQ* is a pulse used to request an interrupt, whereas, *RDY* is a signal that can be programmed to be active in high or low binary logic level, indicating that valid output was produced; this last signal can be used in a polling mode. In Fig. 76.9 we used only 1 bit for the *IRQ/RDY* signal, at the moment of designing the system the designer will have to decide on one method. It is possible to use both, modifying the logic or separating the signal and adding an extra 1-bit output.

The GPIO IP has two 32 bit wide ports, one for input (reading bus) and one for output (write bus). The output bus connects the GPIO IP to the ARM cortex using the 32 bit bus APB. The input bus connects the IT2FC IP to the GPIO IP. The ARM cortex uses the Ft2km_core as a coprocessor.

76.6.2 Development of IP Cores

In Sect. 76.6.1, we showed how to integrate the fuzzy coprocessor through an input/output port, i. e., the IP GPIO. We also commented on the existence of IP cores such as the UART and the timers that are connected directly to the system bus as in any microcontroller system with integrated peripherals. In this section, we shall show how to implement an IT2FC connected to the system bus to obtain an IT2FC IP core integrated to the system architecture. The procedure is basically the same for any FPGA of the Xilinx family. We worked with the Spartan 6 and Virtex 5, so the Xilinx ISE Design Suite was used.

The whole process to start an application that includes a microprocessor and a coprocessor can be broadly divided into three steps:

1. Design and implement the design entity that will be integrated as an IP core in further steps, then follows

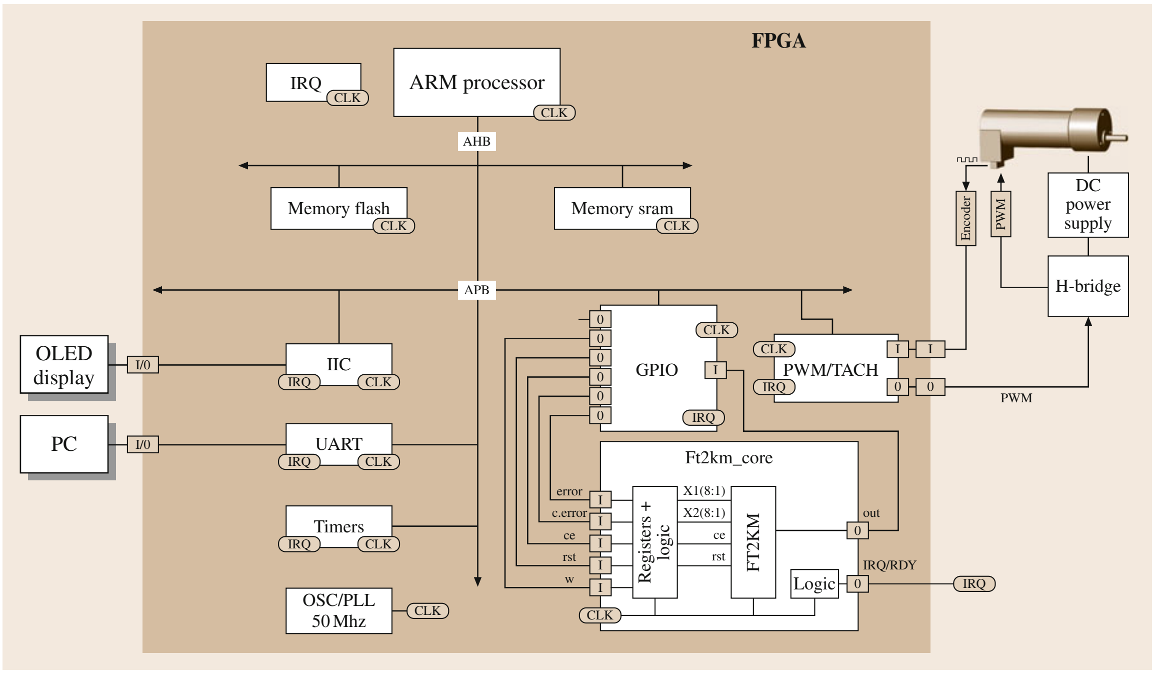

Fig. 76.9 A coprocessor implemented into the Actel Fusion FPGA. The system has an ARM processor, the IT2FC coprocessor implemented through the general-purpose input/output port, and some peripherals

the development flow explained in Sect. 76.4.2. In our case, the design entity is FT2KM.

2. Create the basic embedded microcontroller system tailored for our application. We already know the kind and amount of memory that we will need, as well as the peripherals. This step is achieved as follows: we create the microprocessor system using the base system builder (BSB) of the Xilinx Platform Studio (XPS) software. The system contains a Microblaze softcore, 16 KB of local memory, the data controller bus (dlmb_cntlr), and the instruction controller bus (ilmb_cntlr).
3. Create the IP core, which should contain the desired design entity, in our case the FT2KM. This step is achieved using the *Import Peripheral Wizard* found in the Hardware option in the XPS. The idea is to connect the FTKM design entity to the processor local bus (PLB V4.6) through three registers, one for each input (two registers) and one for the output. Upon the completion, this tool will create synthesizable HDL file (ft2km_core) that implements the intellectual property interface (IPIF)

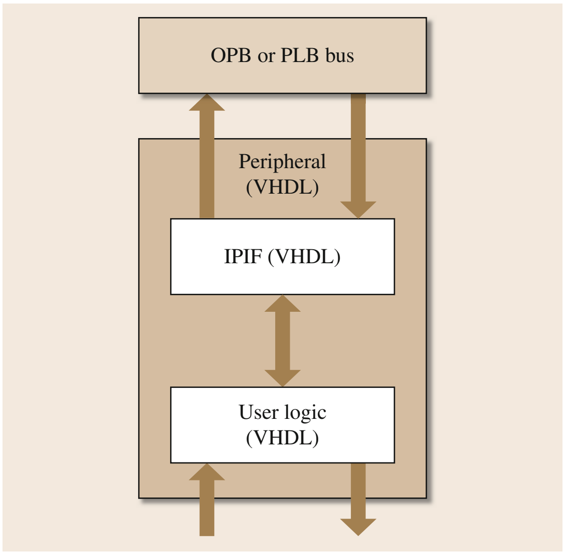

Fig. 76.10 IP Core implementation of a user defined peripheral. The IT2FC coprocessor is implemented into the user logic module. This module achieves communication with the rest of the system through the PLB or the on-chip peripheral bus OPB. For a static coprocessor, use the PLB. For an implementation in the reconfigurable region, use the OPB ◀

services required and a stub *user_logic_module*. These two modules are shown in Fig. 76.10. The IPIF connects the user logic module to the system bus using the OPB or the PLB bus or to the on-chip peripheral bus (OPB). At this stage, we will need to use the *ISE Project Navigator* (ISE) software to integrate to the *user_logic_module* all the required files that implement the FT2KM design entity. Edit the *User_Logic_I.vhd* file to define the FT2KM component and signals. Open the *ftk2_core.vhd* file and create the *ftk2_core* entity and user logic. Synthesize the HDL code and exit ISE. Return to the XSP and add the FTK2_core IP to the embedded system, connect the new IP core to the *mb_plb* bus system and generate address. Figure 76.10 shows the IT2FC IP core; the IPIF consists of the PLB V4.6 bus controller that provides the necessary signals to interface the IP core to the embedded soft core bus system.

4. Design the drivers (software) to handle this design entity as a peripheral.
5. Design the application software to use the design entity.

76.7 Implementing a GA in an FPGA

In essence, evolution is a two-step process of random variation and selection of a population of individuals that responds with a collection of behaviors to the environment. Selection tends to eliminate those individuals that do not demonstrate an appropriate behavior. The survivors reproduce and combine their features to obtain better offspring. In replication random mutation always occurs, which introduces novel behavioral characteristics. The evolution process optimizes behavior and this is a desirable characteristic for a learning system. Although the term *evolutionary computation* dates back to 1991, the field has decades of history, *genetic algorithms* being one avenue of investigation in simulated evolution [76.63]. GAs are family of computational models, which imitates the principles of natural evolution. For consistency they adopt biological terminology to describe operations. There are six main steps of a GA: population initialization, evaluation of candidates using a fitness function, selection, crossover, and termination judgment, as is shown in Algorithm 76.1. The first step is to decide how to code a solution to the problem that we want to optimize; hence, each individual is represented using a chromosome that contains the parameters. Common encoding of solutions are binary, integer, and real value. In binary encoding, every chromosome is a string of bits. In real-value encoding, every chromosome is a string than can contain one or several parameters encoded as real numbers. Algorithm 76.1 starts initializing a population with random solutions, and then each individual of the population is evaluated using a fitness function, which is selected according to the optimization goals. For example, for tuning a controller it may be enough to check if the actual output controller is minimizing errors between the target and the reference. However, one or more complex fitness functions can be designed in order to carry out the control goal. In steps 3 to 5 the genetic operations are applied, i. e., selection, crossover (recombination), and mutation. In step 6, the termination criteria are checked, stopping the procedure if such criteria have been fulfilled.

Algorithm 76.1 General scheme of a GA

```
initialize population with random candidate solutions
evaluate each candidate
repeat
    select parents
    recombine pairs of parents
    mutate the resulting offspring
    evaluate new candidates
    select individuals for the next generation
until termination condition is satisfied
```

In this work, we have chosen work a GA to evolve the IT2FC. However, the ideas exposed here are valid for most evolutionary and natural computing methods. So, there are two methods to implement any evolutionary algorithm. One is based on executing software written using a computer language such as C/C++, similarly as with a desktop computer. The second method is based on designing specialized hardware using a HDL. Both have advantages and disadvantages; the first method is the easier method since there is much information about coding using a high level language for different EAs. However, this solution may have similar limitations for real-time systems since they are slower than hardware implementations by at least

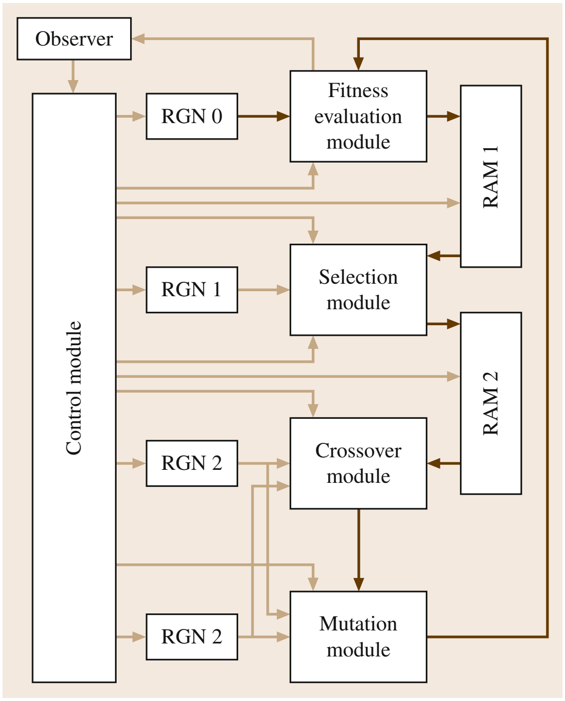

Fig. 76.11 High-level view of the structure of a GA for FPGA implementation

a factor of magnitude of five. On the other hand, state machine hardware-based designs are more complex to implement and use. In this section we shall present a small overview of both methods.

76.7.1 GA Software Based Implementations

It is well known that a GA can run in parallel, taking advantage of the two types of known parallelism: data and control parallelism. Data parallelism refers to executing one process over several instances of the EA, while control parallelism works with separate instances.

Coarse-grained parallelism and fine-grained parallelism are two methods often associated with the use of EA in parallel. The use of both methods is called a hybrid approach. Coarse-grained parallelism entails the EA cores to work in conjunction to solve a problem. The nodes swap individuals of their population with another node running the same problem. The cores can exchange individuals with each other to improve diversity. The amount of information, frequency of exchange, direction, data pattern, etc., are factors that can affect the efficiency of this approach.

In fine-grained parallelism, the approach is to share mating partners instead of populations. The members of populations across the parallel cores select to mate their fittest members with the fittest found in a neighboring node's population. Then, the offspring of the selected individuals are distribuited. The distribution of this next generation can go to one of the parents' populations, both parents' population, or all cores' populations, based on the means of distribution.

Figure 76.4 shows a six-core architecture design for the Virtex 5. Here, we can make fine or coarse-grained implementations of an EA. For example, for coarse-grained implementation, the island model with one processor per island can be used.

76.7.2 GA Hardware Implementations

Figure 76.11 shows a high-level view of the architecture of a GA for hardware implementation. The system has eight basic modules: selection module, crossover module, mutation module, fitness evaluation module, control module, observer module, four random generation number (RGN) modules, and two random access memory modules.

The control module is a Mealy state machine designed to feed all other modules with the necessary control signals to synchronize the algorithm execution. The selection module can have any existing method of selection, for example the Roulette Wheel Selection Algorithm. This method picks the genes of the parents of the current population, and the parents are processed to create new individuals. At the current generation, the crossover and genetic modules achieve the corresponding genetic operation on the selected parents. The fitness evaluation module computes the fitness of each offspring and applies elitism to the population. The observer module determines the stopping criterion and observes its fulfilment. RNGs are indispensable to provide the randomness that EAs require. Additionally, RAM 1 is necessary to store the current population and RAM 2 to store the selected parents of each generation.

76.8 Evolving Fuzzy Controllers

In Sect. 76.1 the general structure of an EFRBS was presented. It was mentioned that the common denominator in most learning systems is their capability of making structural changes to themselves over time to improve their performance for defined tasks. It also was mentioned that the two classical approaches for fuzzy learning systems are the Michigan and Pittsburgh approaches, and there exist newer proposals with the same target. Although to programm a learning system in a computer using high-level language, such as C/C++, requires some skill, system knowledge, and experimentation, there are no technical problems with achieving a system with such characteristics. This can be also true for hardware implementation, if the EFRBS was developed in C/C++ and executed by a hard or soft processor such as PowerPC or Microblaze, it is similarly as it is done in a computer. How to develop a coprocessor was explained in Sect. 76.6. The coprocessor was developed in the FPGA's static (base) region, which cannot be changed during a partial reconfiguration process. Therefore, such coprocessors cannot suffer any structural change. Achieving an EFRBS in hardware is quite different to achieving it using high-level language, because it is more difficult to change the circuitry than to modify programming lines.

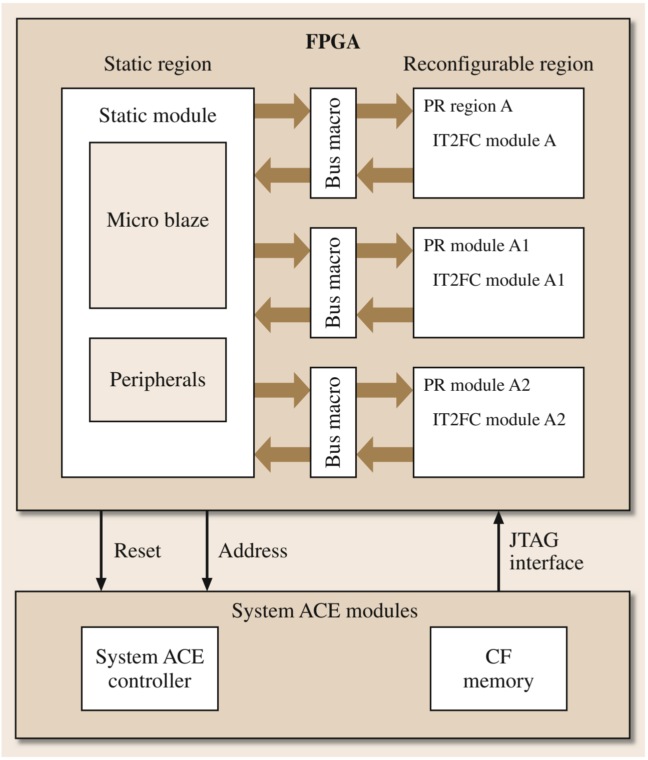

Fig. 76.12 The FPGA is divided into two regions: static and reconfigurable. The soft processor and peripherals are in the static region. Different fuzzy controller architectures are in the reconfigurable region. The bus macro are fixed data paths for signals going between a reconfigurable module and another module

FPGAs are reprogrammable devices that need a design methodology to be successfully used as reconfigurable devices. Since there are several vendors with different architectures, the methodology usually change from vendor to vendor and devices. For the Xilinx FPGAs the configuration memory is volatile, so, it needs to be configured every time that it is powered by uploading the configuration data known as *bitstream*. Configuring FPGA this way is not useful for many applications that need to change its behavior while they still working online. A solution to overcome such a limitation is to use partial reconfiguration, which splits the FPGA into two kinds of regions. The static (base) region is the portion of the design that does not change during partial reconfiguration, it may include logic that controls the partial reconfiguration process. In other words, partial reconfiguration (PR) is the ability to reconfigure select areas of an FPGA any time after its initial configuration [76.64]. It can be divided into two groups: dynamic partial reconfiguration (DPR) and static partial reconfiguration (SPR). DPR is also known as active partial reconfiguration. It allows changing a part of the device while the rest of the FPGA is still running. DPR is accomplished to allow the FPGA to adapt to changing algorithms and enhance performance, or for critical missions that cannot be disrupted while some subsystems are being defined. On the other hand, in SPR the static section of the FPGA needs to be stopped, so autoreconfiguration is impossible (Fig. 76.12).

For Xilinx FPGAs, there are basically three ways to achieve DPR for devices that support this feature. The two basic styles are difference-based partial reconfiguration and module-based partial reconfiguration. The first one can be used to achieve small changes to the design, the partial bitstream only contains information about differences between the current design structure that resides in the FPGA and the new content of the FPGA. Since the bitstream differences are usually small, the changes can be made very quickly. Module-based partial reconfiguration is useful for reconfiguring large blocks of logic using modular design concepts. The third style is also based on modular design but is more flexible and less restrictive. This new style was introduced by Xilinx in 2006 and it is known as early access partial reconfiguration (EAPR) [76.65, 66]. There are two key differences between the design flow EAPR and the module-based one. (1) In the EAPR flow the shape and size of partially reconfigurable regions (PRRs) can be defined by the user. Each PRR has at least one, and usually multiple, partially reconfigurable modules (PRMs) that can be loaded into the PRR. (2) For modules that communicate with each other, a special bus macro allows signals to cross over a partial reconfiguration boundary. This is an important consideration, since without this feature intermodule communication would not be feasible, as it is impossible to guarantee routing between modules. The bus macro provides a fixed *bus* of inter-design communication. Each time partial reconfiguration is performed, the bus macro is used to establish unchanging routing channels between modules, guaranteeing correct connections [76.65].

An important core that enables embedded microprocessors such as MicroBlaze and PowerPC to achieve reconfiguration at run time is HWICAP (hardware internal configuration access point) for the OPB. The HWICAP allows the processors to read and write the FPGA configuration memory through the ICAP (internal configuration access point). Basically it allows writing and reading the configurable logic block (CLB) look-up table (LUT) of the FPGA.

The process to achieve reconfigurable computing with application to IT2FC will be explained with more detail in Sect. 76.8.2. Moreover, how to evolve an IT2FC embedded into an FPGA, whether it resides in the static or in the reconfigurable region, will be also explained in therein.

76.8.1 EAPR Flow for Changing the Controller Structure

Figure 76.12 shows the basic idea of using EAPR flow for reconfigurable computing to change from one IT2FC structure to a different one. In this figure the Microblaze soft processor can evaluate each controller structure according to single or multiobjective criteria. The processor communicates with a PR region using the bus macro, which provides a means of locking the routing between the PRM and the base design. The system can achieve fast reconfiguration operations since partial bitstream are transferred between the FPGA and the compact flash memory (CF) where bitstreams are stored.

In general, the EAPR design flow is as follows [76.64, 67, 68]:

1. Hardware description language design and synthesis. The first steps in the EAPR design flow are very similar to the standard modular design flow. We can summarize this in three steps:
   (a) Top-level design. In this step, the design description must only contain black-box instantiations of lower-level modules. Top-level design must contain: I/O instantiations, clock primitives instantiations, static module instantiations, PR module instantiations, signal declarations, and bus macro instantiations, since all non-global signals between the static design and the PRMs must pass through a bus macro.
   (b) Base design. Here, the static modules of the system contain logic that will remain constant during reconfiguration. This step is very similar to the design flow explained in Sect. 76.4.2. However, the designer must consider input and output assignment rules for PR.
   (c) PRM design. Similarly to static modules, PR modules must not include global clock signals either, but may use those from top-level modules. When designing multiple PRMs to take advantage of the same reconfigurable area, for each module, the component name and port configuration must match the reconfigurable module instantiation of the top-level module.
2. Set design constraints. In this step, we need to place constraints in the design for place and route (PAR). The constraints included are: area group, reconfiguration mode, timing constraint, and location constraints. The area group constraint specifies which modules in the top-level module are static and which are reconfigurable. Each module instantiated by the top-level module is assigned to a group. The reconfiguration mode constraint is only applied to the reconfigurable group, which specifies that the group is reconfigurable. Location constraints must be set for all pins, clocking primitives, and bus macros in top-level design. Bus macros must be located so that they straddle the boundary between the PR region and the base design.
3. Implement base design. Before the implementation of the static modules, the top level is translated to ensure that the constraints file has been created properly. The information generated by implementing the base design is used for the PRM implemen-

tation step. Base design implementation follows three steps: translate, map, and PAR.
4. Implement PRMs. Each of the PRMs must be implemented separately within its own directory, and follows base design implementation steps: i. e., translate, map and PAR.
5. Merge. The final step in the partial reconfiguration flow is to merge the top level, base, and PRMs. During the merge step, a complete design is built from the base design and each PRM. In this step, many partial bitstreams for each PRM and initial full bitstreams are created to configure the FPGA.

Partial dynamic reconfigurable computing allows us to achieve online reconfiguration. By selecting a certain bitstream is possible to change the full controller structure, or any of the stages (fuzzification, inference engine, type reduction, and defuzzification), as well as any individual section of each stage, for example, different membership functions for the fuzzification stage, etc. However, we need to have all the reconfigurable modules previously synthesized because they are loaded using partial bitstreams. Therefore, to have the capability to evolve reconfigurable modules we need to provide them with a control register (CR) to change the desired parameters.

Next, a flexible coprocessor (FlexCo) prototype of an IT2FC (FlexCo IT2FC) that can be implemented either in the static region as well as in the PR is presented.

76.8.2 Flexible Coprocessor Prototype of an IT2FC

Figure 76.13 illustrates the FlexCo IT2FC, which contains the four stages (fuzzification, inference engine, type reduction, and defuzzification). They are connected depending on the target region, to the PLB or to the OPB through a 32 bits command register (CR), which is formed by four 8 bit registers named R1 to R4 (Fig. 76.14). The parameters of each stage can be changed by the programmer since they are not static as they were defined previously for the FT2KM (Sect. 76.5). Now, they are volatile registers connected through signals to save parameter values. The processor (MicroBlaze) can send through the PLB or the OPB, two kinds of commands to the CR: control words (CWs) and data words (DWs). The state machine of the FlexCo IT2FC interprets the command.

Figure 76.14 illustrates the CR coding for static and reconfigurable FC. This register is used to perform

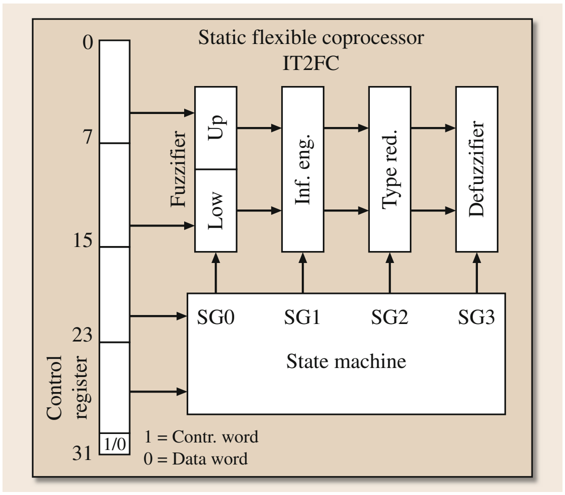

Fig. 76.13 Flexible coprocessor proposal of an IT2FC for the static region

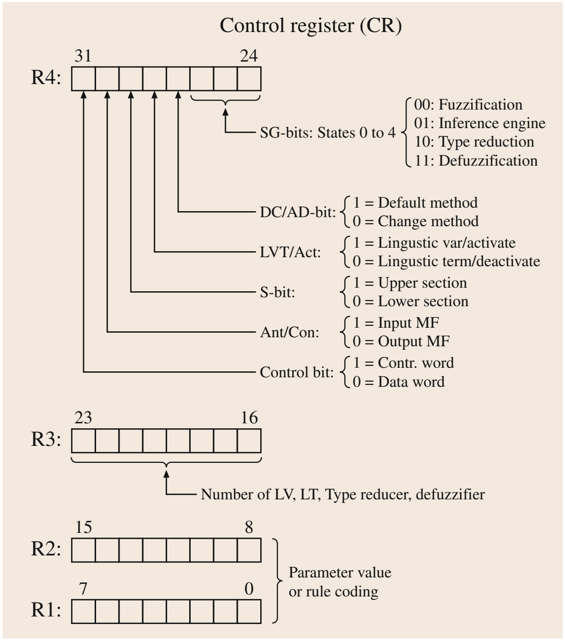

Fig. 76.14 The control register is used for both styles of implementation, in the static region or in the reconfigurable region

parameter modification in both modes, static and reconfigurable. In general, bit 7 of R4 is used to differentiate between a CW or a DW, *1* means a CW, whereas *0* means a DW. The StaGe bits (SG-bits) serves to identify the IT2FC stage that is to be modified.

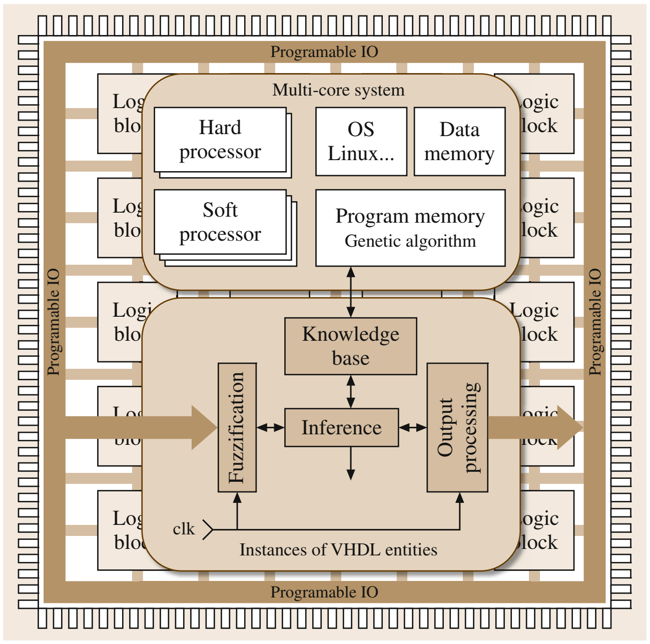

Fig. 76.15 In the static region of the FPGA a multiprocessor system (MPS) with operating system. The GA resides in the program memory, it is executed by the MPS. The IT2FC may be implemented in the reconfigurable region, Fig. 76.16, or in the static region, Fig. 76.13 ►

- *SG-bits* = *00*: The fuzzification stage has been chosen, then it is necessary to set the bit Ant/Con to *1* to indicate that the antecedent MFs are going to be modified. With the section-bit (S-bit) we indicate which part of the FOU (upper or lower) will be modified. The bit linguistic-variable-term/active (LVT/Active) is to indicate whether we want to modify a linguistic variable (LV) or the linguistic term (LT), the *Act* option is for the inference engine (IE). In accordance to the LV/LT bit value, in the register R3 we set the number of the LV or the LT that will be changed. Finally, with registers R1 and R2, the parameter value of the LV or the LT is given, R1 is the least significant byte.
- *SG-bits* = *01*: With this setting, the state machine identifies that the IE will be modified. It works in conjunction with Ant/Con, S-bit, and the registers R1, R2, and R3. Set a *0* value in the Ant/Con bit to change the consequent parameters of a Mamdani inference system, in S-bit choose the upper or lower MF, using R3 indicate the number of MF, and with R1 and R2 set the corresponding value or static implementation. It is possible to activate and deactivate rules using the bit LVT/Active. With bit dynamic change/activate-deactivate (DC/AD), it is possible to change the combination of antecedents and consequents of a specific rule provided that we have made this part flexible by using registers. For an implementation in the reconfigurable region, it is possible to add or remove rules. These two features need to work in conjunction with registers R1, R2, and R3.
- *SG-bits* = *10*: This selection is to modify the type reduction stage. It is possible to have more than one type reducer. By setting the DC/AD-bit to *1*, we indicate that we wish to change the method at running time without the necessity of achieving a reconfiguration process that implies uploading partial bitstreams. The methods can be selected using register R3. By using a DC/AD-bit equal to *0* and LVT/Act equal to *0*, in combination with registers R1 to R3 we can indicate that we wish to change the preloaded values that the KM-algorithm needs to achieve the TR.

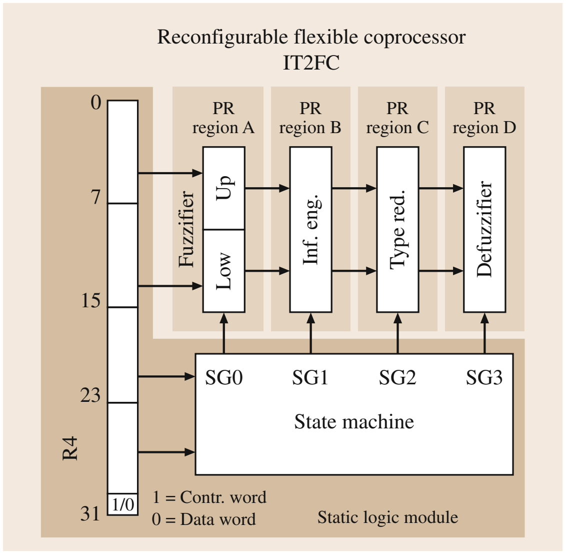

Fig. 76.16 Flexible coprocessor proposal of an IT2FC for the reconfigurable region

- *SG-bits* = *11*: Similarly to the type reduction stage, we can change the defuzzifier at running time.

With respect to the type reducer and defuzzification stages, we give the option to have more than one module, which has the advantage of making the process

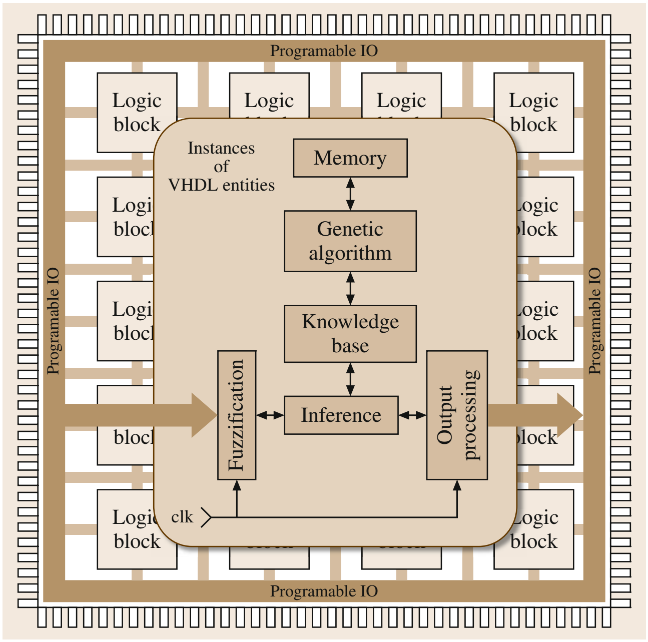

Fig. 76.17 This design may be implemented in both regions to have a dynamic reconfigurable system. For a static implementation, the system must have registers for all the variable parameters to make possible to change their values, Fig. 76.13

easier and possible for static designs, but the disadvantage is that the design will consume more macrocells, increasing the cost of the required FPGAs, boards, and power consumption. Next, we will explain the implementation of the FlexCo IT2FC for the static region and the reconfigurable region.

Implementing the FlexCo IT2FC on the Static Region

The IT2FC of is connected to the PLB. Although the controller structure is static, this system can be evolved for tuning and learning because it is possible to achieve parametric modifications to all the IT2FC stages. Figure 76.13 shows the architecture of this system and Fig. 76.15 a conceptual model of the possible implementation.

Implementing the FlexCo IT2FC on the PR

Figure 76.16 illustrates a more flexible architecture for FlexCo IT2FC. The IT2FC is implemented in the reconfigurable region, using a partially reconfigurable region (PRR) for each stage. This is convenient since each region can have multiple modules that can be swapped in and out of the device on the fly. This is the most recommended method to achieve the evolving IT2FC since it is more flexible. One disadvantage is that at running time it is slower than the static implementation because more logic circuits are incorporated.

Figure 76.17 is an evolutive standalone system; as it was mentioned, the IT2FC and the GA can be in the static or in the reconfigurable region.

76.8.3 Conclusion and Further Reading

FPGAs combine the best parts of ASICs and processor-based systems, since they do not require high volumes to justify making a custom design. Moreover, they also provide the flexibility of software, running on a processor-based system, without being limited by the number of cores available. They are one of the best options to parallelize a system since they are parallel in nature. In an IT2FC, a typical whole T2-inference, computed using an industrial computer equipped with a quad-core processor, lasts about 18×10^{-3} s. A whole IT2FC (fuzzification, inference, KM-type reducer, and defuzzification) lasts only four clock cycles, which for a Spartan implementation using a 50 MHz clock represents 80×10^{-9} s, and for a Virtex 5 FPGA-based system represents 40×10^{-9} s. For the Spartan family the typical implementation speedup is 225 000, whereas for the Virtex 5 it is 450 000. Using a pipeline architecture, the speedup of the whole IT2 process can be obtained in just one clock cycle, so using the same criteria to compare, the speedup for Spartan is 90 000 and 2 400 000 for Virtex. Reported speedups of GAs implemented into an FPGA, are at least 5 times higher than in a computer system. For all these reasons, FPGAs are suitable devices for embedding evolving fuzzy logic controllers, especially the IT2FC, since they are computationally expensive. There are some drawbacks with the use of this technology, mostly with respect to the need to have a highly experienced development team because its implementation complexity. Achieving an evolving intelligent system using reconfigurable computing is not as direct as it is using a computer system. It requires the knowledge of FPGA architectures, VHDL coding, soft processor implementation, the development of coprocessors, high-level languages, and reconfigurable computing bases. Therefore, people interested in achieving such implementations require expertise in the above fields, and further reading must focus on these topics, FPGA vendor manuals and white papers, as well as papers and books on reconfigurable computing.

References

76.1 P.P. Angelov, X. Zhou: Evolving fuzzy-rule-based classifiers from data streams, IEEE Trans. Fuzzy Syst. **16**(6), 1462–1475 (2008)

76.2 O. Cordón, F. Herrera, F. Hoffman, L. Magdalena: *Genetic Fuzzy Systems: Evolutionary Tuning and Learning of Fuzzy Knowledge Bases* (World Scientific, Singapore 2001)

76.3 P. Angelov, R. Buswell: Evolving rule-based models: A tool for intelligent adaptation, IFSA World Congr. 20th NAFIPS Int. Conf. 2001. Jt. 9th, Vancouver, Vol. 2 (2001) pp. 1062–1067

76.4 K. De Jong: Learning with genetic algorithms: An overview, Mach. Learn. **3**(2), 121–138 (1988)

76.5 J.H. Holland: *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence* (MIT Press/Bradford Books, Cambridge 1998)

76.6 K.A. De Jong: *Evolutionary Computation: A Unified Approach* (MIT Press, Cambridge 2006)

76.7 O. Cordón, F. Gomide, F. Herrera, F. Hoffmann, L. Magdalena: Ten years of genetic fuzzy systems: Current framework and new trends, Fuzzy Sets Syst. **141**(1), 5–31 (2004)

76.8 V. Gilles: SIA: A supervised inductive algorithm with genetic search for learning attributes based concepts, Lect. Notes Comput. Sci. **667**, 280–296 (1993)

76.9 J. Juan Liu, J. Tin-Yau Kwok: An extended genetic rule induction algorithm, Proc. 2000 Congr. Evol. Comput., Vol. 1 (2000) pp. 458–463

76.10 O. Cordón, M.J. del Jesus, F. Herrera, M. Lozano: MOGUL A methodology to obtain genetic fuzzy rule based systems under the iterative rule learning approach, Int. J. Intell. Syst. **14**(11), 1123–1153 (1999)

76.11 G.D. Perry, F.S. Stephen: Competition-based induction of decision models from examples, Mach. Learn. **13**, 229–257 (1993)

76.12 G.D. Perry, F.S. Stephen: Using coverage as a model building constraint in learning classifier systems, Evol. Comput. **2**, 67–91 (1994)

76.13 A. Giordana, F. Neri: Searc-intensive concept induction, Evol. Comput. **3**, 375–416 (1995)

76.14 H. Ishibuchi, K. Nozaki, N. Yamamoto, H. Tanaka: Selecting fuzzy if-then rules for classification problems using genetic algorithms, IEEE Trans. Fuzzy Syst. **3**(3), 260–270 (1995)

76.15 A. Homaifar, E. McCormick: Simultaneous design of membership functions and rule sets for fuzzy controllers using genetic algorithms, IEEE Trans. Fuzzy Syst. **3**(2), 129–139 (1995)

76.16 D. Park, A. Kandel, G. Langholz: Genetic-based new fuzzy reasoning models with application to fuzzy control, IEEE Trans. Syst. Man Cybern. **24**(1), 39–47 (1994)

76.17 O. Castillo, R. Sepúlveda, P. Melin, O. Montiel: Evolutionary optimization of interval type-2 membership functions, Proc. 2006 Int. Conf. Artif. Intell. ICAI 2006, Las Vegas (2006) pp. 558–564

76.18 R. Sepúlveda, O. Castillo, P. Melin, O. Montiel, L.T. Aguilar: Evolutionary optimization of interval type-2 membership functions using the human evolutionary model, FUZZ-IEEE (2007) pp. 1–6

76.19 R. Sepúlveda, O. Montiel-Ross, O. Castillo, P. Melin: Optimizing the MFs in type-2 fuzzy logic controllers, using the human evolutionary model, Int. Rev. Autom. Control **3**(1), 1–10 (2010)

76.20 O. Castillo, P. Melin, A.A. Garza, O. Montiel, R. Sepúlveda: Optimization of interval type-2 fuzzy logic controllers using evolutionary algorithms, Soft Comput. **15**(6), 1145–1160 (2011)

76.21 C. Wagner, H. Hagras: A genetic algorithm based architecture for evolving type-2 fuzzy logic controllers for real world autonomous mobile robots, Fuzzy Syst. Conf, Proc. 2007. FUZZ-IEEE 2007, London (2007) pp. 1–6

76.22 J.E. Bonilla, V.H. Grisales, M.A. Melgarejo: Genetic tuned FPGA based PD fuzzy LUT controller, 10th IEEE Int. Conf. Fuzzy Syst. (2001) pp. 1084–1087

76.23 S. Sánchez-Solano, A.J. Cabrera, I. Baturone: FPGA implementation of embedded fuzzy controllers for robotic applications, IEEE Trans. Ind. Electron. **54**(4), 1937–1945 (2007)

76.24 J.L. González, O. Castillo, L.T. Aguilar: FPGA as a tool for implementing non-fixed structure fuzzy logic controllers, IEEE Symp. Found. Comput. Intell. 2007. FOCI 2007 (2007) pp. 523–530

76.25 O. Montiel, Y. Maldonado, R. Sepúlveda, O. Castillo: Simple tuned fuzzy controller embedded into an FPGA, Fuzzy Inf. Proc. Soc. 2008. NAFIPS 2008. Annu. Meet. North Am. (2008) pp. 1–6

76.26 O. Montiel, J. Olivas, R. Sepúlveda, O. Castillo: Development of an embedded simple tuned fuzzy controller, IEEE Int. Conf. Fuzzy Syst., FUZZ-IEEE 2008, IEEE World Congr. Comput. Intell. (2008) pp. 555–561

76.27 Y. Maldonado, O. Montiel, R. Sepúlveda, O. Castillo: Design and simulation of the fuzzification stage through the Xilinx system generator. In: *Soft Computing for Hybrid Intelligent Systems*, Studies in Computational Intelligence, Vol. 154, ed. by O. Castillo, P. Melin, J. Kacprzyk, W. Pedrycz (Springer, Berlin, Heidelberg 2008) pp. 297–305

76.28 J.A. Olivas, R. Sepúlveda, O. Montiel, O. Castillo: Methodology to test and validate a VHDL inference engine through the Xilinx system generator. In: *Soft Computing for Hybrid Intelligent Systems*, Studies in Computational Intelligence, Vol. 154, ed. by O. Castillo, P. Melin, J. Kacprzyk, W. Pedrycz (Springer, Berlin, Heidelberg 2008) pp. 325–331

76.29 G. Lizárraga, R. Sepúlveda, O. Montiel, O. Castillo: Modeling and simulation of the defuzzification stage using Xilinx system generator and simulink.

In: *Soft Computing for Hybrid Intelligent Systems*, Studies in Computational Intelligence, Vol. 154, ed. by O. Castillo, P. Melin, J. Kacprzyk, W. Pedrycz (Springer, Berlin, Heidelberg 2008) pp. 333–343

76.30 M. Grégory, U. Andres, P. Carlos-Andres, S. Eduardo: A dynamically-reconfigurable FPGA platform for evolving fuzzy systems, Lect. Notes Comput. Sci. **3512**, 296–359 (2005)

76.31 Y. Maldonado, O. Castillo, P. Melin: Optimization of membership functions for an incremental fuzzy PD control based on genetic algorithms. In: *Soft Computing for Intelligent Control and Mobile Robotics*, Studies in Computational Intelligence, ed. by O. Castillo, J. Kacprzyk, W. Pedrycz (Springer, Berlin Heidelberg 2011) pp. 195–211

76.32 R.M.A. Melgarejo, C.A. Peña-Reyes: Hardware architecture and FPGA implementation of a type-2 fuzzy system, Proc. 14th ACM Great Lakes Symp. VLSI (2004) pp. 458–461

76.33 C. Lynch, H. Hagras, V. Callaghan: Parallel type-2 fuzzy logic co-processors for engine management, IEEE Int. Conf. Fuzzy Syst., FUZZ-IEEE (2007) pp. 1–6

76.34 R. Sepúlveda, O. Montiel, O. Castillo, P. Melin: Embedding a high speed interval type-2 fuzzy controller for a real plant into an FPGA, Appl. Soft Comput. **12**(3), 988–998 (2012)

76.35 L.A. Zadeh: Fuzzy sets, Inf. Control **8**(3), 338–353 (1965)

76.36 L.A. Zadeh: The concept of a linguistic variable and its application to approximate reasoning -I, Inf. Sci. **8**(3), 199–249 (1975)

76.37 J.M. Mendel: Type-2 fuzzy sets: Some questions and answers, IEEE Connect. Newsl. IEEE Neural Netw. Soc. **1**, 10–13 (2003)

76.38 J.S.R. Jang, C.T. Sun, E. Mizutani: *Neuro-Fuzzy and Soft Computing: A Computational Approach to Learning and Machine Intelligence* (Prentice Hall, Upper Saddle River 1997)

76.39 J.M. Mendel: *Uncertainty Rule-Based Fuzzy Logic Systems: Introduction and New Directions* (Prentice Hall, Upper Saddle River 2001)

76.40 J.M. Mendel: Type-2 fuzzy sets and systems: An overview, IEEE Comput. Intell. Mag. **2**(2), 20–29 (2007)

76.41 J.M. Mendel, R.I.B. John: Type-2 fuzzy sets made simple, IEEE Trans. Fuzzy Syst. **10**(2), 117–127 (2002)

76.42 D. Wu: Approaches for reducing the computational cost of interval type-2 fuzzy logic controllers: overview and comparison, IEEE Trans. Fuzzy Syst. **21**(1), 80–99 (2013)

76.43 D. Wu: Approaches for Reducing the Computational Cost of Interval Type-2 Fuzzy Logic Systems: Overview and Comparisons, IEEE Trans. Fuzzy Syst. **21**(1), 80–99 (2013)

76.44 K. Duran, H. Bernal, M. Melgarejo: Improved iterative algorithm for computing the generalized centroid of an interval type-2 fuzzy set, Fuzzy Inf. Proc. Soc. 2008. NAFIPS 2008. Annu. Meet. North Am. (2008) pp. 1–6

76.45 D. Wu, M. Nie: Comparison and practical implementation of type reduction algorithms for type-2 fuzzy sets and systems, Proc. IEEE Int. Conf. Fuzzy Syst. (2008) pp. 2131–2138

76.46 L.T. Ngo, D.D. Nguyen, L.T. Pham, C.M. Luong: Speed up of interval type-2 fuzzy logic systems based on GPU for robot navigation, Adv. Fuzzy Syst. **2012**, 475894 (2012), doi: 10.1155/2012/698062

76.47 P. Sundararajan: High Performance Computing Using FPGAs, Xilinx. White Paper: FPGA. WP375 (v1.0), 1–15 (2010)

76.48 IBM Redbooks: *The Power4 Processor Introduction and Tuning Guide*, 1st edn. (IBM, Austin 2001)

76.49 J. Yiu: *The Definitive Guide To The ARM CORTEX-M0* (Newnes, Oxford 2011)

76.50 S.P. Dandamudi: *Fundamentals of Computer Organization and Design* (Springer, Berlin, Heidelberg 2003)

76.51 T. Tauber, G. Runger: *Parallel Programming: For Multicore and Cluster Systems* (Springer, Berlin, Heidelberg 2010)

76.52 K.P. Abdulla, M.F. Azeem: A novel programmable CMOS fuzzifiers using voltage-to-current converter circuit, Adv. Fuzzy Syst. **2012**, 419370 (2012), doi: 10.1155/2012/419370

76.53 D. Fikret, G.Z. Sezgin, P. Banu, C. Ugur: ASIC implementation of fuzzy controllers: A sampled-analog approach, 21st Eur. Solid-State Circuits Conf. 1995 ESSCIRC '95. (1995) pp. 450–453

76.54 L. Kourra, Y. Tanaka: Dedicated silicon solutions for fuzzy logic systems, IEE Colloquium on 2 Decades Fuzzy Contr. Part 1 **3**, 311–312 (1993)

76.55 M. Khosla, R.K. Sarin, M. Uddin: Design of an analog CMOS based interval type-2 fuzzy logic controller chip, Int. J. Artif. Intell, Expert Syst. **2**(4), 167–183 (2011)

76.56 C. Bobda: *Introduction to Reconfigurable Computing. Architectures, Algorithms, and Applications* (Springer, Berlin, Heidelberg 2007)

76.57 P.C. Pong: *FPGA Prototyping by VHDL Examples* (Wiley, Hoboken 2008)

76.58 M. Oscar, S. Roberto, M. Yazmin, C. Oscar: Design and simulation of the type-2 fuzzification stage: Using active membership functions. In: *Evolutionary Design of Intelligent Systems in Modeling, Simulation and Control*, Studies in Computational Intelligence, Vol. 257, ed. by O. Castillo, W. Pedrycz, J. Kacprzyk (Springer, Berlin, Heidelberg 2009) pp. 273–293

76.59 S. Roberto, M. Oscar, O. José, C. Oscar: Methodology to test and validate a VHDL inference engine of a type-2 FIS, through the Xilinx system generator. In: *Evolutionary Design of Intelligent Systems in Modeling, Simulation and Control*, Studies in Computational Intelligence, Vol. 257, ed. by O. Castillo, W. Pedrycz, J. Kacprzyk (Springer, Berlin/Heidelberg 2009) pp. 295–308

76.60 S. Roberto, M.-R. Oscar, C. Oscar, M. Patricia: Embedding a KM type reducer for high speed fuzzy controller into an FPGA. In: *Soft Computing in Industrial Applications*, Advances in Intelligent and Soft Computing, Vol. 75, ed. by X.-Z. Gao, A. Gaspar-Cunha, M. Köppen, G. Schaefer, J. Wang (Springer, Berlin/Heidelberg 2010) pp. 217–228

76.61 S. Roberto, M. Oscar, L. Gabriel, C. Oscar: Modeling and simulation of the defuzzification stage of a type-2 fuzzy controller using the Xilinx system generator and simulink. In: *Evolutionary Design of Intelligent Systems in Modeling*, *Simulation and Control*, Studies in Computational Intelligence, Vol. 257, ed. by O. Castillo, W. Pedrycz, J. Kacprzyk (Springer, Berlin/Heidelberg 2009) pp. 309–325

76.62 O. Montiel-Ross, J. Quiñones, R. Sepúlveda: Designing high-performance fuzzy controllers combining ip cores and soft processors, Adv. Fuzzy Syst. **2012**, 1–11 (2012)

76.63 D.B. Fogel, T. Back: An introduction to evolutionary computation. In: *Evolutionary Computation*. *The Fossile Record*, ed. by D.F. Fogel (IEEE, New York 1998)

76.64 W. Lie, W. Feng-yan: Dynamic partial reconfiguration in FPGAs, 3rd Int. Symp. Intell. Inf. Technol. Appl. (2009) pp. 445–448

76.65 D. Lim, M. Peattie: Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations. XAPP290. May 17, 2007

76.66 Emil Eto: Difference-Based Partial Reconfiguration. XAPP290. December 3, 2007

76.67 Xilinx: Early Access Partial Reconfiguration User Guide For ISE 8.1.01i, UG208 (v1.1). May 6, 2012

76.68 C.-S. Choi, H. Lee: A self-reconfigurable adaptive FIR filter system on partial reconfiguration platform, IEICE Trans. **90-D**(12), 1932–1938 (2007)