Real-Time Full-Programmable Fuzzy Processor on an Intel Pentium III

Enrique Frías-Martínez[1] **Julio Gutiérrez-Ríos**[2] **Felipe Fernández-Hernández**[2]

[1] Dpto. de Tecnología Fotónica
ETSI Telecomunicación, Universidad Politécnica de Madrid
28040 Ciudad Universitaria s/n, Madrid, Spain
e-mail: efrias@tfo.upm.es

[2] Dpto. de Tecnología Fotónica
Facultad de Informática,
Universidad Politécnica de Madrid
28660 Boadilla del Monte, Madrid, Spain

Abstract

This paper presents a real-time full-programmable fuzzy processor using piecewise-linear interpolation techniques and implements it using the SSE (Streaming SIMD Extensions) set of instructions of an Intel Pentium III. A full-programmable fuzzy processor is defined as a system where the set of rules, the membership functions, the t-norm, the t-conorm, the aggregation operator, the propagation operator, and the defuzzification can be defined by any valid algorithm. Real-time fuzzy processing is defined as processing the knowledge base in a constant time and with a minimum speed of 10 MFLIPS. The full-programmable fuzzy processor presented processes four 2 input-1 output fuzzy systems at 100 MFLIPS on an Intel Pentium III 450 MHz.

Key Words: fuzzy processing, linear interpolation

1. Introduction

The need to process fuzzy knowledge base systems with high speed resulted in the development of fuzzy hardware architectures. This first developments were done in the mid-80's by Togai [6] using a digital architecture, and by Yamakawa [7] using analog techniques. Before that, other ASICS designed to process fuzzy knowledge bases with high speed where developed [8][2].
The reasons for using a standard architecture for high-speed fuzzy processing are:

- Standard processors have reached a point where their speed is fast enough to process fuzzy systems with high speed.
- The use of a compiler that adapts a fuzzy system to a standard architecture allows to obtain high-speed.

The key concept is given by the second point. The fuzzy syntax of the rules is a useful way of representing knowledge, and the fuzzy algorithm is a useful way of processing it, but, it does not defines the best way of processing for a standard architecture. In order to process fuzzy systems with high speed in standard architectures it is needed a compilation from the fuzzy system syntax to a syntax suitable for a standard architecture.

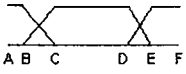
There are compilers developed for TSK systems mainly, and usually are based on interpolation [4][8].
A characteristic common to all fuzzy coprocessors is that only the set of rules and the membership functions of the system can be defined, because the fuzzy algorithm is implemented by hardware, and can not be programmed. This problem is also presented in fuzzy compilers, they are designed for a specific fuzzy system (usually TSK), and again, only rules and membership functions can be defined.
In this paper a full-programmable fuzzy system is defined as a system where the rules, membership functions, the T-norm, the T-conorm, the propagation operator, the aggregation operator, and the defuzzification algorithm can be defined. A full-programmable fuzzy compiler will allow to execute any kind of application using standard hardware.
In the rest of the paper, first a full-programmable fuzzy compiler is introduced. Then, the high level architecture of the controller is presented, and the following points present the speed achieved for different systems.

1. Full-Programmable Fuzzy Model

The model proposed is divided in an off-line processing and an on-line processing.

1.1 Off-line processing

FS is a vector $FS=(T_n,T_c,PO,AO,D,R,MF,I,O)$ that describes a fuzzy system with T_n the T-norm, T_c the T-conorm, PO the propagation operator, AO the aggregation operator, D the defuzification algorithm, R the set of Rules , MF the membership functions, I the inputs and O the outputs of the system.
For each $I_i \in I$, the Activation Intervals of I_i, AI_i, are defined as the union of the set of intervals, left-closed and right-open, except the last one which is also right-closed, given by the extreme points of the kernel and the support of each one of the membership functions defined in I_i. AI_i is obtained from $(MF_{i,1},...,MF_{i,p})$ with $p=Card(MF_i)$. Fig. 1 presents an example of Activation Intervals.



$AI=\{[A,B),[B,C),[C,D),[D,E),[E,F]\}$

Fig. 1 *Activation Intervals AI.*

The Activation Intervals can be obtained from the Activation Points. The Activation Points are defined as the set of points of each dimension of the system that define the kernel and support of each linguistic label. Formally, the Activation Point, AP_i can be obtained as:

$$AP_i = F_e(F_o(\cup(Supp_e(MF_{i,m}) \cup Kernel_e(MF_{i,m})))) \text{ with}$$

$$m=1,...,p$$

From that, the Activation Intervals, are obtained as:

$$AI_i = F_{in}(AP_i)$$

where $Supp_e$ and $Kernel_e$ are functions that obtain the extreme points that define the kernel and the support of a label, F_o is a function that orders a set of points, F_e is a function that eliminates the repeated points of a set, and F_{in} obtains the intervals, right-closed and left-open, that are defined by a set of points.
Given AI_i, $i=1,...,Card(I)$, and S the n-dimensional input space given by $(I_1,...,I_n)$, P is defined as a partition of S given by the cartesian product ($\times$) of AI_i, $i=1,...,Card(I)$:

$$P = \times AI_i,\ i=1,...,Card(I)$$

The partition P is defined in a way that divides the input space of the system in the zones where the designer has a complete certainty of the output from the zones where the certainty is only partial. This can be done supposing that the certainty of the designer is shown in the kernel of each one of the membership functions designed.

The cell P_i of the partition P is defined in a N-dimensional system as:

$$P_i=\{(A_1,B_1),(A_2,B_2),...(A_N,B_N)\}$$

From the definition of P_i, it can be obtained the set of vertex that define a cell:

$$P_i=\{(A_1,A_2,..,A_N),(A_1,A_2,...,B_N),...,(B_1,B_2,...,B_N)\}$$

The model is based on the value of the fuzzy system FS in the set of vertex that define P. For that, the matrix V is defined as a matrix that contains the set of vertex of partition P. V can be obtained from the Activation Points as:

$$V = \times AP_i,\ i=1,...,Card(I)$$

The Characteristic Matrix CM contains the value of the fuzzy system FS in each vertex of the partition P. CM is defined as:

$$CM=FS(V)$$

where $FS(V)$ represents the value of each element of matrix V in the fuzzy system given by $FS=(T_n,T_c,PO,AO,D,R,MF,I,O)$.

Example:

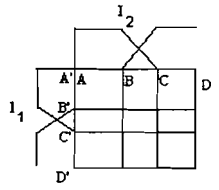


Fig. 3 Partition P of the fuzzy system FS

Given FS a bidimensional fuzzy system (Fig. 3), and P the partition of the input space, V and CM can be obtained as:

$$V = \begin{pmatrix} (A',A) & (A',B) & (A',C) & (A',D) \\ (B',A) & (B',B) & (B',C) & (B',D) \\ (C',A) & (C',B) & (C',C) & (C',D) \\ (D',A) & (D',B) & (D',C) & (D',D) \end{pmatrix}$$

$$MC = \begin{pmatrix} SB(A',A) & SB(A',B) & SB(A'.C) & SB(A',D) \\ SB(B',A) & SB(B',B) & SB(B',C) & SB(B',D) \\ SB(C',A) & SB(C',B) & SB(C',C) & SB(C',D) \\ SB(D',A) & SB(D',B) & SB(D',C) & SB(D',D) \end{pmatrix}$$

■

1.1.1 Equalization and Normalization of the inputs

The compiler also defines a set of equalization and normalization functions.
One equalization function, $E_k(I_k)$, is defined for each input of the system. $E_k(I_k)$ is defined as:

$$E_k(I_k) = \begin{cases} 0 \cdot si \cdot I_k \in IA_{k,0} \\ 1 \cdot si \cdot I_k \in IA_{k,1} \\ ... \\ p \cdot si \cdot I_k \in IA_{k,p} \end{cases}$$

The equalization function is defined to identify in which *Activation Interval* is the input included. The set of values of the equalization functions of a system allows to identify the active cell, the cell in which the input is included.

One normalization function, $N_{k,p}(I_k)$, is defined for each Activation Interval of each dimension k. The normalization function normalizes between 0 and 1 the actual input in the active cell. The normalization function $N_{k,p}(I_k)$ is defined as:

$$N_{k,p} = \frac{1}{(B-A)}(I_k - A)$$

Fig. 4 shows the normalization done by $N_{k,p}(I_k)$.

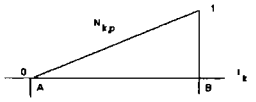


Fig.4 Normalization done by $N_{k,p}(I_k)$.

The proposed model defines a compilation of the programmable fuzzy system *FS* to a specification of the same system (Characteristic Matrix, Equalization functions and Normalization functions) suitable to be executed in a standard architecture.

1.2 On-line processing

The on-line processing is divided in two steps:

• The first step obtains the relevant information to obtain the output of the system. First, the equalization of each dimension of the system is obtained:

$$a_1 = E_1(I_1), \ldots, a_n = E_n(I_n)$$

The vector $(a_1,\ldots,a_n)$ identifies the cell of P_k in which the input is included. The Characteristic Vector of an input, *CV(I)*, is defined as the set of 2^n values of the original fuzzy systems in the set if vertex that define the cell in which the input is included. *CV(I)* can be obtained from the Characteristic Matrix, *CM*, as:

$$VC(I') = \{MC_{a_1,a_2,\ldots,a_n}, MC_{a_1+1,a_2,\ldots,a_n}, \ldots, MC_{a_1,a_2,\ldots,a_n+1}\}$$

In this step, the normalization of the input in the active cell, *N(I)*, is also obtained:

$$N(I') = (N_{1,a_1}(I'_1),\ldots,N_{n,a_n}(I'_n))$$

• The second step, using the information given by the Characteristic Vector, *CV(I)*, and *N(I)*, calculates the output of the system *O*. The output of the fuzzy model *O*, will be calculated with a function F_o of *CV(I)* and *N(I)*:

$$O = F_o(VC(I),N(I))$$

F_o has to produce a value similar to the output produced by the fuzzy system *FS*, and it has to be evaluated with high speed. The F_o proposed for the model is a multilinear interpolation among the values of *VC(I)* using *N(I)*.

3. Architecture of the Controller

The controller implemented is divided in four modules, the sensor, the interpolation cache, the model memory and the inference engine, interconnected as seen in Fig. 5.

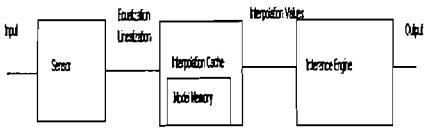


Fig. 5. Interconetion of the modules of the Controller

Sensor
The sensor implements the set of Equalization and Normalization functions, $E_k(I_k)$, $N_{k,p}(I_k)$. From the input of the system *I*, the sensor obtains $(a_1,\ldots,a_n)$ and *N(I)*.

Memory Model
The Memory Model stores the Characteristic Matrix of the system, *CM*.

Interpolation Cache
The Interpolation Cache receives the equalization and the normalization of the input. From the equalization of the input, it obtains the Characteristic Vector *CV* accessing the Memory Model.
Due to the locality of the inputs, if an input is in a cell P_k, the most possible situation is that the next input of the system *I*, will be in the same cell P_k. This means that the Inference Engine will have to work with the same Characteristic Vector as in the previous inference, so the Interpolation Cache does not need to access the Memory Model to obtain it.
The output of the interpolation cache is the Characteristic Vector *CV* and the normalization of the input.

Inference Engine
The inference engine receives the Characteristic Vector *CV* and the normalization of the input *N(I)* and obtains the output *O* applying multilinear interpolation.
The SSE set of instructions [3] of an Intel Pentium III is an excellent environment to implement a multilinear interpolation. This can be done using the SIMD [5] architecture of the processor and the set of intrinsics provided to access the hardware.
The Pentium III processor has a 128 bit register which can be accessed as a __mm128 data type. This data type can be also viewed as four 32 bit registers. Using this structure, one operation can be done in parallel to four different data, as can be seen in Fig. 6.

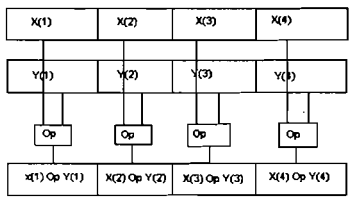


Fig. 6 SIMD data structure of a Pentium III

To access and operate with __m128 data the Intel Compiler has a set of intrinsics [3]. An intrinsic is a C function that directly translates into a assembler operation that can not be obtained using standard C code. Table 1 has examples of some intrinsics.

Table 1. Example of Intrinsics.

Intrinsic	Description
__m128 _mm_mul_ps (__m128 x, __m128 y)	Multiplies two _m128 data and returns a _m128 data
__m128 __mm_add_ps(__m128 x, __m128 y)	Adds two _m128 data and returns a _m128 data.

The code presented in Fig. 7 implements four inference engines (one in each element of the SIMD vector) of two inputs and one output, using the set of SSE instructions.

```
OutputG.OutputI=_mm_add_ps(_mm_add_ps(CV_G.
CV_I[0],_mm_mul_ps(N_G.N_I[1],_mm_sub_ps(CV
_G.CV_I[1],CV_G.CV_I[0]))),_mm_mul_ps(N_G.N
_I[0],_mm_sub_ps(_mm_add_ps(CV_G.CV_I[2],_m
m_mul_ps(N_G.N_I[1],_mm_sub_ps(CV_G.CV_I[3]
,CV_G.CV_I[2]))),_mm_add_ps(CV_G.CV_I[0],_m
m_mul_ps(N_G.N_I[1],_mm_sub_ps(CV_G.CV_I[1]
,CV_G.CV_I[0])))));
```

Fig. 7 Code of the inference engines

Where *OutputG* is the output of the system, *CV_G* is the Characteristic Vector of the input and *N_G* the normalization. All of them have been defined as union, in order to be accessed either as __m128 or as an array of four floats.

The feedback given to the controller will be given by the architecture of the implemented fuzzy system. This way, with the code given in Fig. 7 different controllers can be implemented. Some examples can be: four controllers of 2 inputs-1 output, 1 controller of 8 inputs-4 outputs, 1 controller of 5 inputs- 1 output, etc.

4. Features of the High-Speed Full-Programmable Fuzzy Model

Fig. 8 shows the original surface of a 2 Input-1 Output fuzzy system with Max-Min as inference system and COG as defuzzification method, and the approximation obtained applying the high-speed full-programmable model.

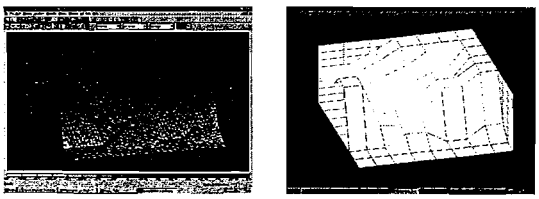


Fig. 8 Surfaces of the original fuzzy system (left) and the approximation (right)

Both surfaces are really similar because the surface obtained with the approximation is based on keeping the certainty of the designer. Comparing 48682 values of the two systems it has been found that 35% of the points have no error, that 65% of the points have an error smaller that 10 units of the output (in a range of 1900 units), and that the average error is 0.77% on the range of the ouput.

Each inference engine implemented in Fig. 7 has a speed of 25 MFLIPS, this gives a total throughput of 100 MFLIPS given by a Pentium III 450 MHz. (25 MFLIPS each inference system multiplied by 4 inference engines implemented). Table 2 presents the speed obtained for four inference engines of 2 inputs-1 output and four inference engines of 4 inputs-1 output.

Table 2. Speed obtained with the propossed model.

	Inference Engine	Throughput
4 Systems 2 I / 1 O	*25 MFLIPS*	*100 MFLIPS*
4 Systems 4 I / 1 O	*3.2 MFLIPS*	*12.8 MFLIPS*

The processing speed obtained is enough to process any kind of fuzzy application [1], and is faster that the best part of implementations and ASICS developed up to now.

5. Conclusions and Future Work

We have developed a full-programmable fuzzy model able to process fuzzy systems with high speed on standard architectures. A Full-Programmable model allows to use the same hardware and the same model with any kind of application. The speed of the system can be easily improved using the new Intel family processors.

References

[1] A. Costa, A. Gloria, "Hardware Solutions for Fuzzy Control", in Proc. of the IEEE, Vol. 83, No 3:422-434, 1995

[2] H. Eichfeld, "A 12b General-Purpose Fuzzy Logic Controller Chip", in IEEE Transactions on Fuzzy Systems,Vol 4, No. 4:460-475 ,1996

[3] Intel Corporation, "Intel C/C++ Compiler User's Guide, with support for the Streaming SIMD Extensions", 1999

[4] R. Rovatti, "Linear and Fuzzy Piecewise-Linear Signal Processing with an Extended DSP Architecturein FUZZ-IEEE 98:1082-1087

[5] S. Thakkar, "Internet Streaming SIMD Extension", in IEEE Computer, Dec. 1999:26-34

[6] M. Togai, "Expert System on A Chip", IEEE Expert, Vol. 1, No. 3:55-62, 1986

[7] T. Yamakawa, "A simple fuzzy computer hardware system employing MIN&MAX operations", Proc. of the 2nd IFSA Congress:122-130, 1987

[8] N. Yubazaki, "Fuzzy inference chip ZP-0401 based on interpolation", in Fuzzy Sets and Systems, Vol.98(3):299-310