CHAPTER 2

The Parallel Pipeline Digital VLSI 0.7 μm High Rate Fuzzy Processor

2.1. Introduction

So far, the commercial fuzzy chips which may be found on the market at very low prices are mostly designed and oriented to very general purpose customers for application fields such as control systems [3], [4]. The most commonly used fuzzy processors have been implemented in the past years in washing machine controls, auto focus imaging, speed and brake control for electric engines and so on. The main reasons that may justify these applications rely on the fuzzy logic intrinsic vagueness, on the robust output response and on the easy implementation at very low cost. In fact, where control systems do not require many input variables, very fast responses, and where the input-output mathematical relationship is not strictly defined, fuzzy logic applies and fit very well [5]. Thus, fuzzy processors are generally smaller in size and, consequently, more efficient than traditional processors.

On the other hand, there are several fields in which fuzzy processors are not directly oriented to. For example where high computation performances are needed, neither traditional nor fuzzy processors can be implemented since the output result of a generic or a fuzzy algorithm may not be met within a very short time. In other words, for some dedicated application fields, traditional processors require a well-defined input-output algorithm that could not be easily defined while commercial fuzzy processors may not be sufficiently fast. For this reason, our research group has been dealing with high speed fuzzy processors in the latest years, for HEPE applications [6], [7], [8]. Within this application fields in fact, the electronic devices used for detecting, recognizing and storing physics events such as particle trajectories have absolutely to be as fast as possible, have to be designed with a high noise immunity, low power consumption and high performances in terms of reliability, robustness and flexibility. This is why our research group has investigated the possibility of designing a high speed Fuzzy Processor for HEPE applications. This goal has been met both by applying parallel-pipeline architecture and by implementing no-time consumption rule identification. For this reason we have particularly designed an *Active_Rule_Selector* for selecting just a subset of the fuzzy rules, here after called *active fuzzy rules*, and the architecture has been divided both with parallel and pipeline stages (see Figure 2.1.). This goal was pretty hard at the beginning but, step by step, it turned to more feasible solutions. We received the chip back from the foundry last year and, so far, it works properly.

2.2. Fuzzy Inference Scheme

Fuzzy inference is the process of formulating the mapping from a given input set to an output using fuzzy logic. The basic elements of fuzzy logic are linguistic variables, fuzzy sets, and fuzzy rules. The linguistic variable's values are words,

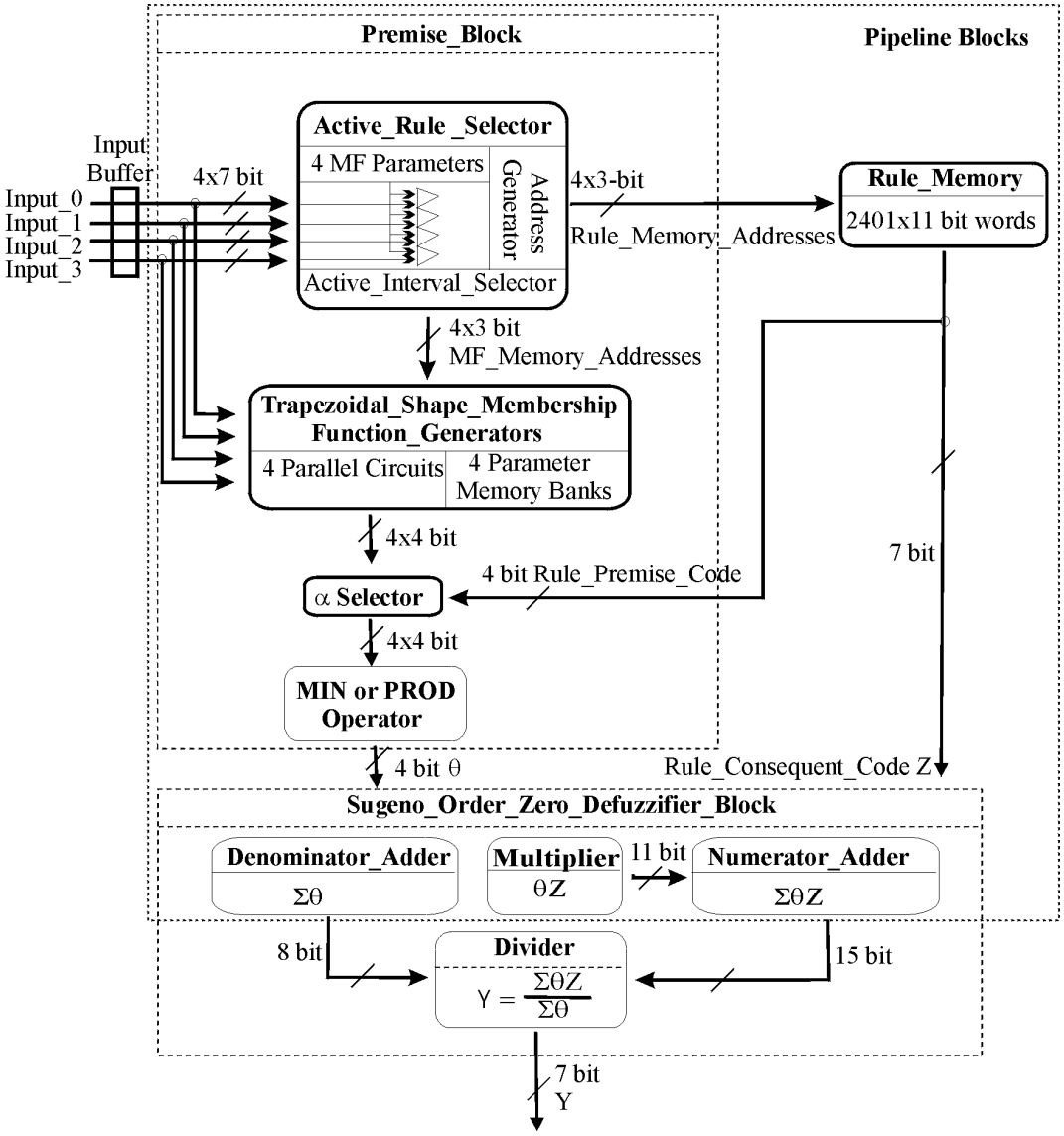

Figure 2.1. The Fuzzy Processor Architecture

specifically adjectives like “small”, “little”, “medium”, “high” and so on. A fuzzy set is a collection of couples of elements. It generalizes the concept of classical set, allowing its elements to have a partial membership. The degree to which the generic element *x* belongs to the fuzzy set *A* (expressed by the linguistic statement *x* is *A*) is characterized by a membership function (MF), $f_A(x)$. In other words, a fuzzy set *A* is defined within a universe of discourse *U* as follows:

$A = \{(x, f_A(x)), f_A(x):\rightarrow[0,1]\}$

U is the whole input range allowed for a given fuzzy linguistic variable.
All fuzzy sets related to a given variable make up the term set, the set of labels within the linguistic variable described or, more properly, granulated.

Fuzzy rules form the basis of fuzzy reasoning. They describe relationships among, imprecise, qualitative, linguistic expressions of the system's input and output. Generally, these rules are natural language representations of human or expert knowledge and provide an easily understood knowledge representation scheme. A typical conditional fuzzy rule assumes a form such as

IF Speed is low AND Race is Dry THEN Braking is Soft

Speed is low AND Race is Dry is the rule's premise, while Braking is Soft is the consequent. The premise predicate may not be completely true or false, and its degree of truth ranges from 0 to 1 (as explained later). We compute this value by applying the membership functions of the fuzzy sets labeled Low and Dry to the actual value of the input variables Speed and Race. (We will explain this fuzzification process in subsequent sessions). After that, fuzzification is applied to the conclusion; the way in which this happens depends on the inference process. As explained later, we used the Sugeno order zero defuzzification method [9].

A Sugeno order zero system uses a weighted average of data points. These points are output membership functions' constant values (singletons), as if they were pre-defuzzified fuzzy sets. The fuzzy sets representing the outputs of each fuzzy rule are weighted by the rule premise degree of truth. Our fuzzy processor computes the output by adding all the contributions $\sum(\theta i * Zi)$ just prior to the final step – the division of $\sum(\theta i * Zi)$ by $\sum(Zi)$ – to give a weighted average result. In this way each fuzzy rule contributes to the final result as long as its $\theta i * Zi$ is non–null, meaning its θi is non-null, since generally the output membership functions are non-null.

2.3. Fuzzy Processor Main Features

Firstly the processing rate is independent from the fuzzy system. In fact, since in this solution the fuzzy system is always composed of all the possible combinations of the input fuzzy sets, the number of fuzzy rules loaded into the *Rule_Memory* is fixed and depends on the number of input variables (see below). Of course the Fuzzy Processor has been provided with a software tool for generating all the fuzzy rules starting from just a few of them. In other words, if one has just a few rule fuzzy system, the software tool may expand the system to an equivalent one which contains all the possible fuzzy rules in a format to be ready for the Fuzzy Processor.

To select the rules an *Active_Rule_Selector* identifies the *fuzzy active rules* related to each input data set, which are going to be processed, with a no-time consuming operation. These *fuzzy active rules* are just the fuzzy rules among all the possible ones that give a non null contribution to the output result. Nevertheless this point is explained in detail in section 2.6. The chip architecture has been divided into several pipeline stages where each of the 12 pipeline stages takes 20 ns for a clock signal of 50 MHz that is the frequency the chip has been supposed to work.

2.4. Block Subdivision

The main features of the Fuzzy Processor are below summarized:

- up to four 7 bit inputs, one 7 bit output;

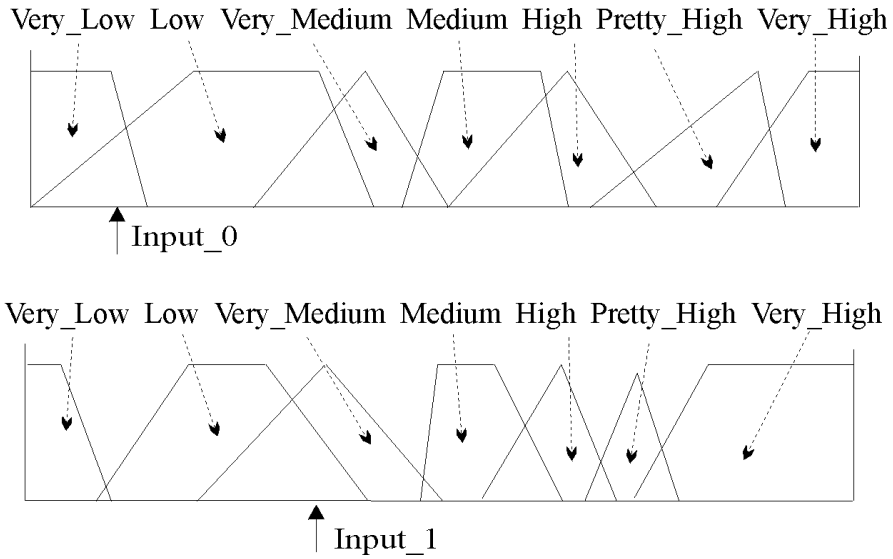

Figure 2.2. Involved Fuzzy Sets

- up to 7 trapezoidal membership functions (MFs) for each of the input variable fuzzy sets;
- up to 2401 fuzzy rules for 4 input variables (in general the number of fuzzy rules is given by 7 fuzzy sets raised to the number of input variables);
- overlapping of the input fuzzy sets at most two at a time;
- 128 crisp MFs named *Zi* for the output variable *Y*;
- 4 bits both for the antecedent and the premise degree of matching (truth), called respectively α and θ values. This is explained in details in section 4.1.2;
- Sugeno order zero [9] defuzzification method;
- T-norm conjunction implemented by a minimum (here after called MIN) or product (here after called PROD) operator to obtain the θ value here after called premise degree of truth;
- 50 Mega Fuzzy Inference per Second for a clock frequency of 50 MHz.

On the other hand the chip architecture, for readability purposes, can be mainly divided into two blocks as shown in Figure 2.1.:

- the *Premise_Block* ;
- the *Sugeno_Order_Zero_Defuzzifier_Block.*

2.4.1 Fuzzification Process

The fuzzification process consists on associating a fuzzy set to a crisp value. In the case of a fuzzy system applied to a physical problem, at any time of observation the input variables have a unique numerical value. In this Fuzzy Processor the numerical values are singleton crisp ones. This process applies for the four input variables; the intersection of the crisp input variable values and the fuzzy sets corresponding to the linguistic terms of the premise give the four input degrees of matching here after called α. Actually we prefer calling these values degrees of truth even if it is well known that

the value of physical variable is intrinsically true. These α indicate how much each input variable belongs to a given fuzzy set, with a scale factor, which represents a grade of membership from 0 to 1.

The *Premise_Block* generate the trapezoidal MF shape and compute the α values by means of four parallel circuits called *Trapezoidal_Shape_Membership_Function_Generators.* Then the *MIN_or_PROD_Operator* carries out the θ value. Nevertheless, before extracting this value, an α_Selector circuit selects the right α among which the θ value is to be considered. This is done by rejecting the α values, or better, by putting them to 1111 so that they do not affect the θ value. This applies when the corresponding input variables are not present into the premise of the fuzzy rule. In fact any fuzzy rule may involve just a subset of the 4 input variables; see section 4.1.2. Then the θ value goes directly to the *Sugeno_Order_Zero_Defuzzifier_Block* that is described in section 5. To carry out this entire premise block 10 pipeline stages are necessary.

2.4.1.1 Fuzzification Process: Active Interval Selector

The four input variables *Input_0*, *Input_1*, *Input_2* and *Input_3* are coded as 7-bit numbers and enter at the same time into the Fuzzy Processor. We use 7-bit numbers instead of a byte because it is enough for the precision required in HEPE applications. At this time a circuit named *Active_Interval_Selector* (see Figure 2.1.) selects, for each input variable, the involved trapezoidal fuzzy sets that have been previously defined by means of four parameter each. In more details, under the hypothesis that for each input variable the fuzzy sets overlap each other just two at a time, we have a situation like that one illustrated in Figure 2.2. where we suppose to have seven fuzzy sets for each of the four input variables. In the Figure 2.2. is shown that only the fuzzy rules where *Input_0* is related to *Very_Low* and *Low* fuzzy sets and *Input_1* is related to *Low* and *Very_Medium* fuzzy sets give a non null contribution to the final result: this is the definition of *active fuzzy rule*. Otherwise the degree of truth of the input variable is zero. Therefore, the problem turns to process only the fuzzy rules that involve these fuzzy sets. All the rest of the rules would not give any contribution so that their computation is not worthwhile. From this point on the problem has been faced by designing a circuit able to find out the involved fuzzy sets and, consequently, the involved fuzzy rules. The operation of extracting the desired intervals can be easily done by means of successive comparisons between each input variable value and the starting end ending points of the trapezoidal shape membership functions. The four parameters that identify each fuzzy set are the starting and ending points of the oblique lines and their slopes. These parameter-points are hold into flip-flop buffers that may be loaded during the loading phase of the Fuzzy Processor. Furthermore, once the involved fuzzy sets have been identified, the architecture is ready to carry out the input variable degrees of truth.
All the previously described job is made by four parallel *Trapezoidal_Shape_Membership_Function_Generator*.

2.4.1.2. Fuzzification Process: Active Rule Selector

In Figure 2.1. the output of the *Active_Interval_Selector* circuit, that is a part of the *Active_Rule_Selector*, consists of a couple of 3-bit codes, one for each variable, that define the involved fuzzy sets explained in Figure 2.2. These 3-bit codes identify the

two adjoining fuzzy sets involved by any given input variable. After that these 3-bit codes are used to generate both the *Rule_Memory_Addresses* and the *MF_Memory_Addresses* which store respectively the fuzzy rules and the four parameters needed for generating the trapezoidal shape membership functions. Particularly the *Address_Generator* performs all the possible combinations of the generated 3-bit codes and generates the addresses needed for the *fuzzy active rules*. In more detail, since the *Active_Interval_Selector* identify all the possible involved fuzzy sets at once, the *Address_Generator* can create, clock period by clock period, all the possible addresses relative to the *fuzzy active rules*. This is why we say that this is a no-time consumption operation.

Besides that, since only 7 fuzzy sets are allowed for each variable and the overlapping of any adjoining fuzzy sets is up to 2, the total number of possible rules is 7^4=2401, but the number of the *fuzzy active rules*, which can give a non-null contribution, is much smaller. In fact the *fuzzy active rules* are only 2^4=16. In this way using the *Active_Rule_Selector* to select just the *fuzzy active rules* the number of rules to be processed is strongly reduced and, consequently, is reduced the processing time.

Once the addresses have been carried out the *Rule_Memory* is read and the minimum or product operation is done by selecting the four α values related to the *active fuzzy rule* under process. Thus, the rule memory output, that contains a *Rule_Premise_Code* as shown in Figure 2.1., selects the right degrees of truth. In fact, a given rule can anyway involve or not all the input variables. The *Rule_Memory* is dimensioned to contain all the possible 7^4 combinations of the input variables and fuzzy sets. In this way the fuzzy rules are loaded in the *Rule_Memory* starting, for example, from the one that involves all the lowest FSs for the input variables up to the one that involves all the highest corresponding FSs. So that for a given address (coming from the *Address_Generator*) it is known in advance which fuzzy rule is considered. Thus the *Rule_Memory* can be organized as 2401 words of 11 bits; each word of this memory contains both *Zi* that is a 7-bit code representing the *Rule_Consequent_Code* (zero order Sugeno crisp value) and the *Rule_Premise_Code* that is a 4-bit code for selecting which variables are present and, consequently, are to be taken, and which ones are to be rejected. For example when the rule premise code is 1111 all the input variables are present, when 1000 only *Input_3* is present, when 0000 neither are present and so on. It should be noted that this 4-bit code has anything to do with the previously mentioned 3-bit one. In fact, the previous 3-bit code is generated by the *Active_Rule_Selector* since it identify the involved fuzzy sets among seven ones.

The output of the *Address_Generator* is also used as addresses for the *4_Parameter_Memory_Banks* memories: in these memories 4 parameters are stored for defining each of the 7 MFs of the 4 input variables.

The following step is to calculate the premise degree of truth θ by performing the minimum or product operations on α. First the four α values have to be selected by the α*_Selector* depending on the value assumed by the *Rule_Premise_Code* and then the *MIN_or_PROD_Operator* performs the minimum or product operations on demand. If for example one variable is not present the corresponding α value has to be changed to 1111 in order to avoid affecting minimum or product operations; this is exactly what the α*_Selector* does.

It is to be noted here that 1111 does not affect the minimum operation since it is the highest 4-bit value and does not also affect the product operation since a normalization process has been implemented so that each 4-bit value multiplied by 1111

returns itself; in fact 1111 represents the highest degree of truth. So the *MIN_or_PROD_Operator* block receives as input the corrected α and is able to extract the final premise degree of truth θ among them. Then a fast 7x4 multiplier, implemented using the Wallace algorithm [10], performs the multiplication between θi and *Zi*. Two parallel adders carry out the additions between θi and the products $\theta i * Zi$ and a final divider performs the division between the 15-bit $\sum(\theta i * Zi)$ and the 8-bit $\sum \theta i$ giving in output the desired value *Y*. The previous additions are respectively represented by 15 and 8 bits since θi is a 4-bit word that, once added up to 16 times give a 8 bit addition (denominator of formula 2.2.), while $\theta i * Zi$ is a 4-bit word multiplied by a 7-bit word that is 11-bit word; once again added 16 times gives a 15-bit addition (numerator of formula 2.2.).

2.4.1.3. Fuzzification Process: Trapezoidal Shape Membership Function Generator

The proposed solution reduces dramatically the layout area in comparison to the look-up table solutions, by a factor that depends on the size of the look-up table and, of course on the definition in terms of number of bit [11], [12]. The circuit approximates a generic trapezoidal shape function by two straight lines and three strictly fixed zones for high and low levels. More precisely, in digital electronics the straight lines are digitized into 15 steps for 4-bit values. Of course, to define a trapezoidal MF, four parameters are anyway required as previously mentioned. Here, as parameters, we have used the two starting points of the *Rising* and the *Falling Straight Lines* and two coefficients related to the two desired slopes. To give a generic example on how the trapezoidal shape is generated suppose to have a 7-bit fuzzy variable, and a 4-bit degree of membership α (16 values). As shown in Figure 2.3., it is easy to understand that the two *Low Zones* and the *High Zone* can be generated by implementing digital comparators. For each zone the related comparators check whether the input value is included or not; this means that the degree of membership has to be carried out by choosing it between low and high logic level or, in other words, the degree of membership α is put either to 0 or 15. The rest of the MF, corresponding to the straight lines, may be carried out in another manner.
Let us now consider the *Falling Straight Line*: the circuit must generate a straight line from the *Falling Edge* to the beginning of the *Low Zone*.
This straight line must fall down linearly by giving a digital output result from 15 to 0. In short terms, the architecture related to this job executes equation 1 without performing any division computation that, as well known, is a big time consuming operation. In the following equations we give some more details about this fast solution. In particular, as shown in Figure 2.3., ΔX_Rise and ΔX_Fall are the intervals under which respectively the rising and the falling straight lines are defined. For example let $\Delta X\_rise(fall)$ be lower than 128 (taking into account that the universe of discourse of *X* is 128 [0,127]). Let us define in the below equations some parameters related to the input variables, such as *Rise*, *Fall*, *A* and *B*. The parameters *Rise* and *Fall* are allowed to vary wherever into the universe of discourse while *A* and *B* can assume integer values always belonging to the interval [0,127].

$$Rise = X - Rising\_Edge ; \quad Fall = X - Falling\_Edge ; \quad A = 128/\Delta X\_rise ; \quad B = 128/\Delta X\_fall$$

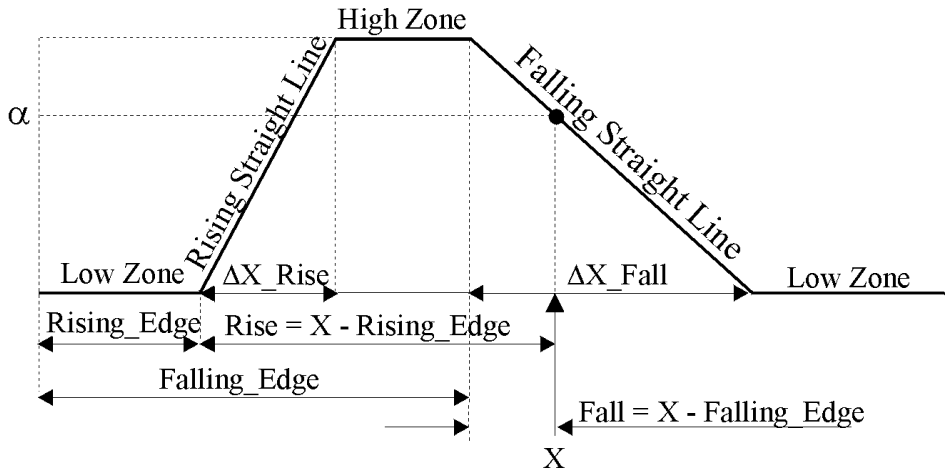

Figure 2.3. Trapezodal Shape Membership Function

where *Rise*, *Fall*, *A* and *B* belong to [0,127].
Using the previous terms the straight-line equation becomes:

$\alpha = XOR[(\Delta Y/\Delta X\_fall)*Fall;1111]$; $8*\alpha = XOR[\ 8*(\Delta Y/\Delta X\_fall)*Fall;1111]$;
$8*\alpha = XOR[\ (128/\Delta X\_fall)*Fall;1111]$; $8*\alpha = XOR[A*Fall\ ;1111]$;
$\alpha = XOR[(A*Fall)/8;1111]$; (2.1)

Here ΔY is put to 16 and is directly the degree of membership α. Nevertheless, the right ΔY should be 15 but, since $15/\Delta X\_fall$ is a non-integer number and must be rounded anyway, the approximation is really reasonable. The same reasoning can be extended to the number 127 that is replaced with 128.

So far, we have not been dealing with the hardware implementation of formula 2.1. Nevertheless, this operation just needs a multiplication between the operands *A* and *Fall*. The operand *A* stands for an 8-bit slope parameter while the *Rise* one is the shifted-input variable. The division by 8 does not take effect at all since it is a division by a power of 2 (truncation operation). Thus, the XOR operation with the code 1111 just makes the complement to 1 that transposes a rising straight line into a falling one with the opposite slope.

Finally, the rising and falling straight-line generation solutions are very similar since just the output αs are different. Obviously, the α related to the rising straight line are not complement to by means of XOR operation. This feature allows generating the two straight lines by sharing most of the applied hardware reducing again the global layout silicon area. For example, for a 7-bit fuzzy variable, 4-bit degree of membership α and 1 bit of precision, we have obtained a standard-cell layout area of about 2 mm^2 and it works properly within 20 ns with a pipeline structure.

2.4.2. Rule Memory

The *Rule_Memory* stores all the fuzzy rules for describing any given problem. As just mentioned this memory is composed of 2401 7-bit words for taking into account the largest case in which 4 input variables are used. In the other cases where the Fuzzy Processor is used for 2 or 3 input variables just a subset of this memory cells will be loaded. In these cases only the first 7^2=49 or 7^3=343 will be used.
A description of how the fuzzy rules are stored into the *Rule_Memory* follows below.

2.4.2.1. Rule Memory: Data Organization

As well known, the fuzzy reasoning is made of fuzzy rules that involve several input and one-output variables. For example, one of the typical way to code the fuzzy reasoning is storing into the *Rule_Memory* the fuzzy set code of the input and output

$$Y = \frac{\sum_{1}^{\#Act} Zi * \vartheta i}{\sum_{1}^{\#Act} \vartheta i} \qquad (2.2)$$

variables involved by each rule. Let us give an example by means of a general fuzzy rule like

if (Input_0 is Very_Low) and (Input_1 is Medium) and .... then (Output is High)

This fuzzy rule is a generic one while the Fuzzy Processor here presented deals only with *fuzzy active rules* to reduce the processing time. As previously mentioned *fuzzy active rules* mean the fuzzy rules that give a non null contribution to the output result. For example, let us have N input variables, K fuzzy sets for each input variable, only t-norm operator for the rules and at most an overlap of 2 (at most only two different consecutive fuzzy sets can overlap each other for any given input variable value). With these conditions we would have K^N possible fuzzy rules that, especially for large K and N numbers, are too many to deal with. On the other hand, the active fuzzy rules, under the constraint of a fuzzy sets overlap of 2, are just 2^N. This is the main point usually adopted to find a way to reduce the number of processed fuzzy rules by selecting the active ones. The basic idea of this solution is to dimension the *Rule_Memory* by means of the number of all possible K^N combinations of input variables and fuzzy sets. This reasonably applies for K^N smaller than few thousands that means for example 7^4, 13^3, etc. In addition the fuzzy rules are to be loaded into the *Rule_Memory* in a sorted way starting, for example, from the one that involves all the lowest fuzzy sets *Very_Low* for all the input variables up to the one that involves all the highest corresponding fuzzy sets *Very_High*. So for example, the first fuzzy rule corresponds to the input fuzzy sets *Very_Low, Very_Low, Very_Low, Very_Low*, the second fuzzy rule corresponds to the input fuzzy sets *Very_Low*, *Very_Low*, *Very_Low*, *Low* and so on. Then for any given address it is identified in advance, apart from the consequent, which fuzzy rule is considered.

In order to match the high-speed constraints previously described, the fuzzy sets related to the input and output variables are to be identified within the fuzzy rules before the global inference process takes place. To match this purpose a specific code has been

developed and stored into the rule memory. In other words the *Rule_Premise_Code* allows identifying if the rule is present in the fuzzy system and which input variables and output fuzzy sets are involved. If for example in Figure 2.1. the *Rule_Premise_Code* was 0110 the related *active fuzzy rule* in the fuzzy system would involve only *Input_1* and *Input_2* while would not consider the *Input_0* and *Input_3* by means of the (1111) 4-bit codes described in sections 4.1. and 4.1.1. Moreover the *Rule_Consequent_Code* identifies the crisp value *Zi* of the output fuzzy set. If we had 0000, as *Rule_Premise_Code*, it would mean that the fuzzy system do not need its contribution and, if involved by the input variable values, the contribution must be zero.

2.5. Sugeno Order Zero Defuzzifier Block: Numerator/Denominator Adder

The *Sugeno_Order_Zero_Defuzzifier_Block* performs the two additions $\Sigma Zi\theta i$ and $\Sigma\theta i$ by two parallel pipeline stages and, once all the rules have been processed, the data stored into the two adders of the defuzzifier go to the divider circuit to compute the crisp output value by means of the Sugeno order zero formula 2.2. that here follows:
In the formula 2.2. #Act stands for the number of *fuzzy active rules* that is $2^{\#\text{ Input Variables}}$.
In case of 4 input variables *#Act* is 16, in case of 3 is 8 and in case of 2 is 4.

The above division operation is computed in parallel to the pipeline stages while the system begins a new data set processing. The division of the two above sums is performed in a combinatorial circuit in less than 90 ns. Eventually, each rule is processed in one clock period and, for a 50 MHz clock signal and 4 input variables (16 active rules as explained below) we obtain that the total processing time is given by adding altogether the:

- number of active rules times the clock period: 16 x 20 ns = 320 ns;
- the delay due to the number of pipeline stages which is 12 x 20 ns = 240 ns;
- the delay due to the time required by the division process, which is less than 90 ns.

2.6. Pipeline stages

The overall architecture of the Fuzzy Processor is pipeline as shown in Figure 2.4., where it is displayed the data flow for every pipeline stage. Besides that, two different pipeline architectures work in parallel since, for many pipeline stages, more than one computation has to be carried out. This, for example, applies for the two final adders *Numerator_Adder* and *Denominator_Adder*, it also applies for the memories read cycles since, while the 4_Parameter_Memory_Banks are read, the *Rule_Memory* is just addressed and so on. Thus the whole pipeline structure has to be considered as a double branch parallel pipeline one. It is to be noted that the 20 pipeline stages shown in the Figure 2.4. are composed of 12 actual pipeline stages into which the fuzzy architecture has been divided and 8 pipeline stages due to the number of *fuzzy active rules*. In fact, in the Figure 2.4. is shown the case of just three input variables where only 2^3=8 *fuzzy active rules* are present. Nevertheless this time is considered as a pipeline time; it also should be noted that if four input variables would be used, the *fuzzy active rules* would rise to 16 and, consequently 16 pipeline stages would be required.

From the moment a new data set enters the processor 12 pipeline stages are required for the fuzzification and inference processes. In the first clock period the input data have to be synchronized with the internal clock signal. Then the two addresses for

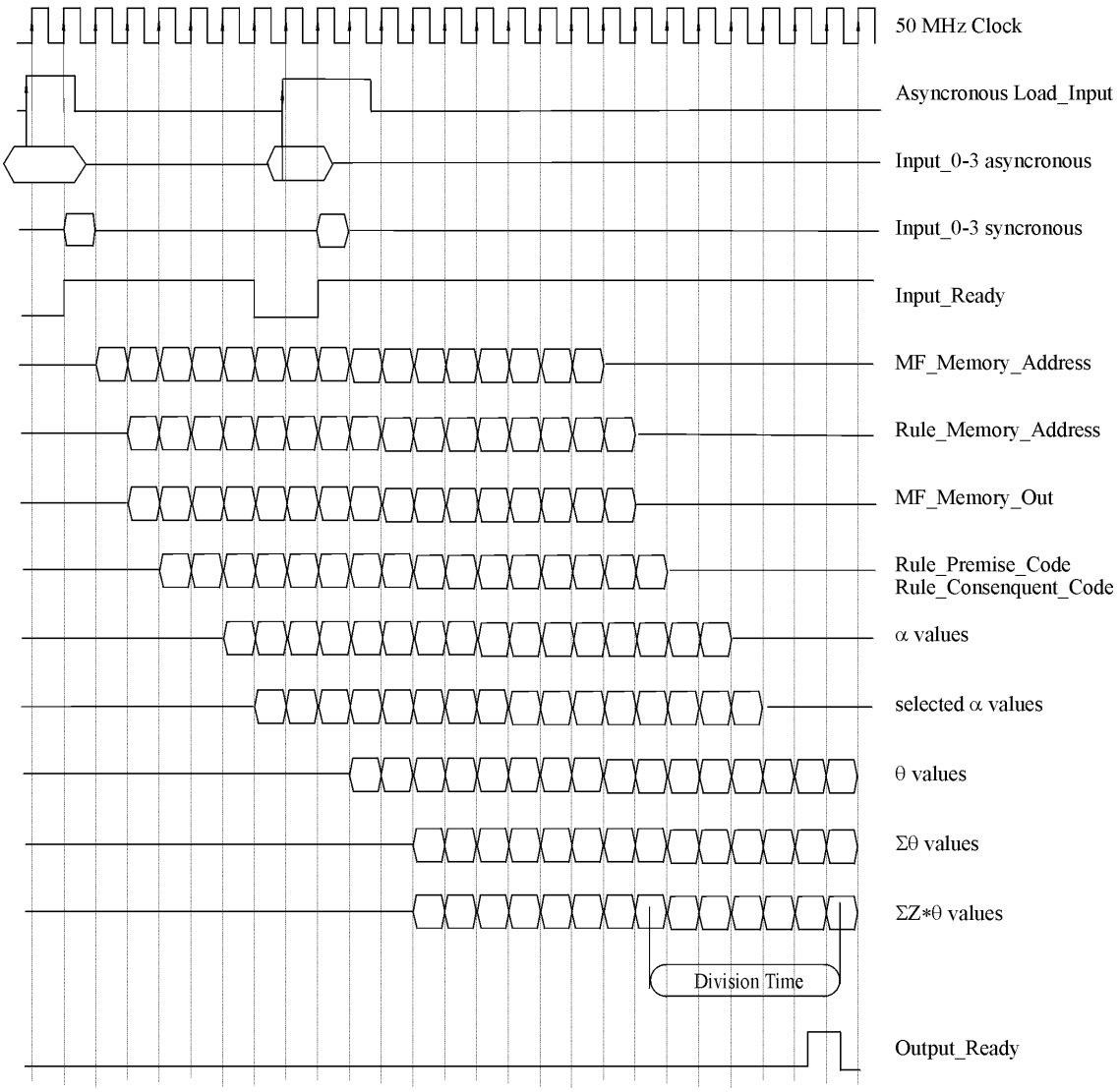

Figure 2.4. Pipeline Stages

the *4_Parameter_Memory_Banks* and the *Rule_Memory* are computed. This is done by selecting the involved fuzzy sets for each input variable. This processes take place into the *Active_Rule_Selector*. First, during the second pipeline stage, the *MF_Memory_Address* is produced and a period later, during the third one, the four MF parameters are available for the *Trapezoidal_Shape_Membership_Function_Generator*. This circuit takes three pipeline stages to compute the input variable degree of truth, from the fourth to the sixth pipeline stages. This process is computed while the *Rule_Memory* is also addressed and read. Once the three α are ready they have to be selected depending on the *Rule_Premise_Code* and this is done in the seventh pipeline stage. Then the four α (three selected plus one put to the highest value ”1111” for not affecting the minimum or product operation) are processed two at a time for giving the premise degree of truth θ. The first θ is produced in the tenth pipeline stage since during

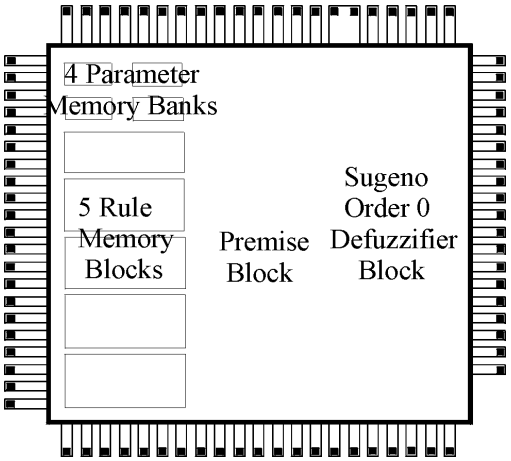

Figure 2.5. Layout Representation

the eight and nine pipeline stages the minimum and product operation are computed and in the ninth one of the two is selected. The first θ^*Z is valid two period later during the twelfth pipeline stage. Thus, after 12 periods both sums $\sum\theta$ and $\sum(\theta^*Z)$ are carried out, so that the final process of division, which requires about 90 ns, can start. What is really remarkable in this pipeline structure is that a new input data set can enter the system after only eight clock periods since at this stage all the eight memory addresses have already been generated and the first logic blocks can accept new data. Figure 2.4. shows a data flow shaded representation of a first input data set and a normal representation of a new data set that immediately follows.
Altogether in the Figure 2.4. the delays give rise to a global processing time of 490 ns if a 50 MHz clock rate is used. Nevertheless it has to be noted that this time does not have anything to do with the input data set rate which depends on the number of *fuzzy active rules*: in the previous case would have been 160 ns.

2.7. Layout Representations

Below is shown a view of the whole layout. Particularly, during the layout design, firstly the main fuzzy *Rule_Memory* has been divided into five smaller blocks for reducing both the access time (read cycle) and the power consumption. In fact, being able to predict which part of the *Rule_Memory* will be read, it is possible to enable just the correspondent memory block instead of enabling all the *Rule_Memory*. This solution reduces the memories power consumption to one fifth of their global value. In other words, enabling just one *Rule_Memory* block at a time and leaving the others in a stand-by mode, the power consumption is greatly reduced if compared to the global memory consumption that would be required for one bigger memory block. In addition, generally the smaller is the memory block, the lower is the access time. Nevertheless, all these considerations give rise to a larger chip area but this can be afforded and accepted for high-speed constraints.

In addition, all the standard cells that have been implemented for the rest of the Fuzzy Processor, from the membership function generators to the *Active_Rule_Selector*, from the inference circuits to the defuzzifier, have also been divided into four main

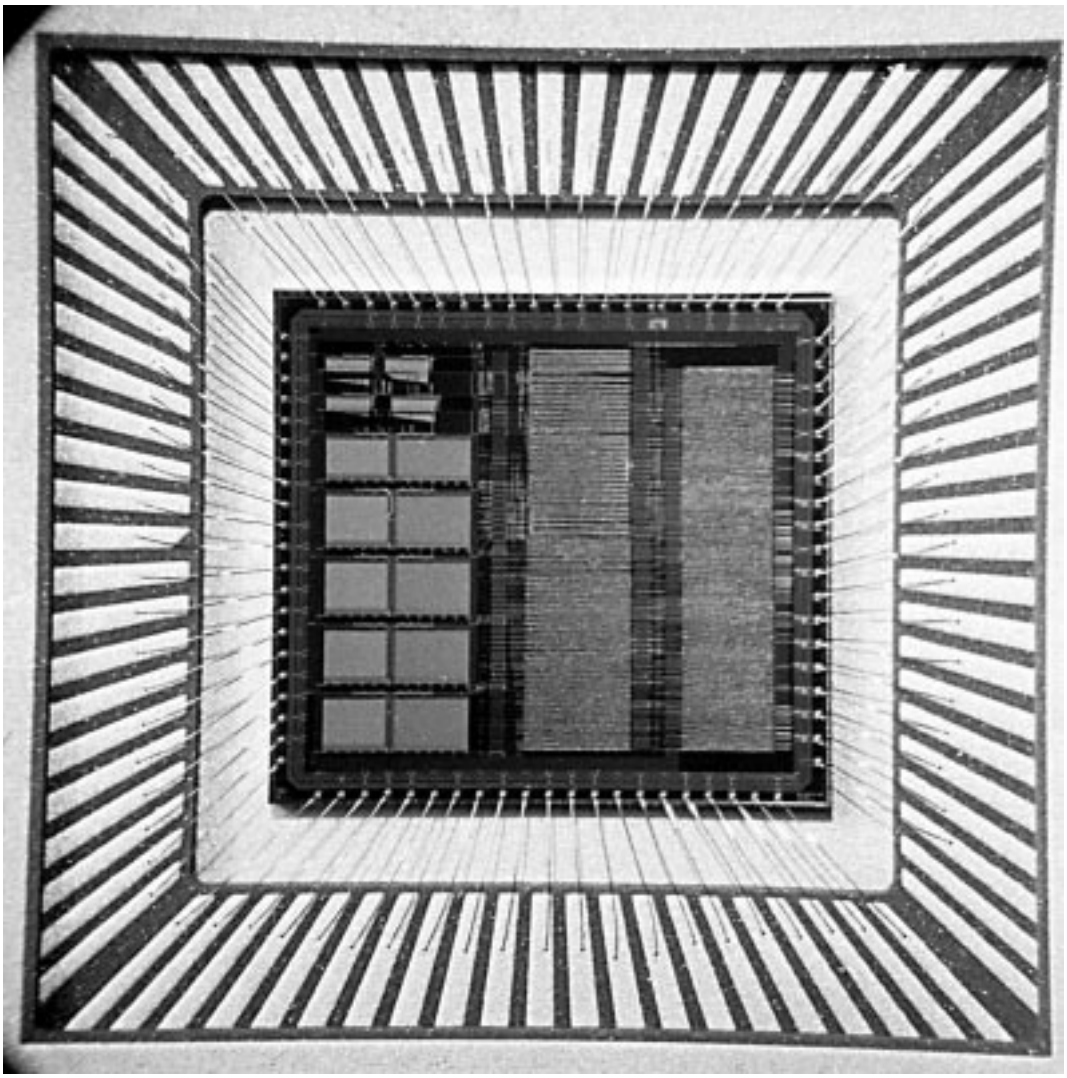

Figure 2.6. Microphotograph of the Fuzzy Processor

blocks, according to the logical function they were designed to. In more details, the defuzzification circuits have been grouped together in one *Sugeno_Order_Zero_Defuzzifier_Block*; all the circuits related to the memories address selection and to the four membership function generators have been put together with the circuits dedicated to the input variable interval identification, into the *Premise_Block* (see Figure 2.5.).

This chip organization allows a pretty simple layout design from several points of view. For example the clock net distribution can be faced easily by a tree structure routed within the main blocks; the standard cells related to the same circuits are forced to stay close to each other; the power and ground nets can be interdigitized for the power supply distribution and so on. In other words the clock wire has been routed by means of a main vertical 25 μm wide trunk and several 3 μm wide branches among all the standard cell lines. The choice of this net routing style has been justified by the fact

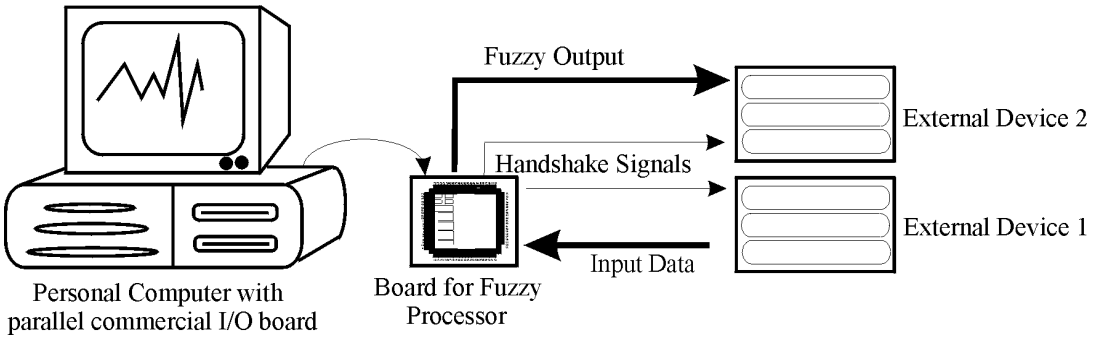

Figure 2.7. Fuzzy Processor Implementation

that, in this case, the global net capacitance is mainly due to the logic gates and does not depend much on the parasitic effects; otherwise a H clock tree structure would have been more appropriated. Finally the microphotograph of the Fuzzy Processor is shown in Figure 2.6.

2.8. Input-Output Implemented Features

As already explained above, for HEPE applications the speed in terms of computation time is a strong constraint and is absolutely to be met and, for making the Fuzzy Chip flexible, it has to be easy-to-use as far as the input-output handshake signals. The Fuzzy Processor is to be used with a printed board and synchronized with an on-board clock signal. So, it provides itself with all the synchronization phases between itself and the external device write and load cycles. The Fuzzy Processor, in fact, does not delegate the input-output handshake synchronization signals to external devices such as controllers or dedicated processors, but a simple handshake signal configuration has been designed.
In more details an *Input_Ready* signal, synchronous with the on-board clock signal, is used for enabling the external device write cycle (External Input Device in Figure 2.7.). In other words the external device can write its data into the Fuzzy Processor, by means of an external driven *Load_Input* signal, just when this *Input_Ready* signal is activated. In addition, the external device must hold the input data set and the *Load_Input* signal valid for at least two on-board clock periods. In this way the Fuzzy Processor can both recognize the external device write cycle and synchronize the device data with the on-board clock signal. Moreover, an output signal named *Output_Ready* has been implemented to enable the external device (External Output Device in Figure 2.7.) for loading the output datum of the Fuzzy Processor. Since the Fuzzy Processor may be synchronized with a up to 50 MHz (20 ns) clock rate, and since the division process can take up to 90 ns, this output handshake signal is to be synchronized five clock periods after the division process starts and lasts one clock period. In addition, this output handshake signal may be considered both during rising and falling edge since it is low when non active while goes high for one on-board clock cycle when is activated. This is for a flexible output handshake configuration.

2.9. Conclusions

The Fuzzy Processor has met the constraints for which it has been designed in terms of speed, flexibility and feasible implementation on a printed board. We are going to apply it to physics experiments where high computation speed are required for detecting, selecting and recognizing particle trajectories. Nevertheless, due to the implemented features for making it configurable in different ways it may be applied as a general purpose Fuzzy Processor. In more details the Fuzzy Processor has an architecture configurable in different ways in terms of number of input variables, shape of input membership functions, minimum or product inference operation. The estimated power consumption is about 1300 mW for a 50 MHz clock frequency while the global silicon area is 60 mm^2 [13]. It has been implemented with a 0.7 μm digital technology. Moreover, since the Fuzzy Processor has been mostly designed by means of VHDL language apart from the memory blocks, it may be adapted for future, more dedicated applications.