uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Vishal Thareja

AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.A.Sc. (Electrical and Computer Engineering)

GRADE / DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

A Configurable Coprocessor Architecture for Fuzzy Logic Algorithms Implemented on FPGA

TITRE DE LA THÈSE / TITLE OF THESIS

Miodrag Bolic

DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

Voicu Groza

Rafik Goubran

Gary W. Slater

Le Doyen de la Faculté des études superieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

A Configurable Coprocessor Architecture for Fuzzy Logic Algorithms Implemented on a FPGA

VISHAL THAREJA

A thesis submitted to the Faculty of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements for the degree of

MASTER OF APPLIED SCIENCE
IN ELECTRICAL AND COMPUTER ENGINEERING

Ottawa-Carleton Institute for Electrical and Computer Engineering
School of Information Technology and Engineering
University of Ottawa

Your file Votre référence
ISBN: 978-0-494-73850-4
Our file Notre référence
ISBN: 978-0-494-73850-4

Abstract

Fuzzy Logic (FL) has grown to be a popular algorithm for designing embedded control systems. FL simplifies the design of a control system, but there are many challenges in implementing FL in embedded control systems. Deep knowledge of hardware design as well as processor integration is required to embed FL in a control system. The proposed design methodology takes a FL algorithm modelled in Matlab and configures the coprocessor architecture to the control behaviours of the model. The coprocessor is then automatically integrated with the Leon3 processor and the entire design is compiled and synthesized to be implemented on an FPGA. The model parameters configure behavioural VHDL generics without changing the coprocessor architecture and automated software scripts help generate the design from model to VHDL easily and quickly. The coprocessor architecture achieves high throughput with minimal area utilization by incorporating FPGA on-chip resources in the coprocessor architecture. FL implementation with the configurable coprocessor architecture with processor offers better flexibility over pure hardware implementations and better performance over pure software implementations. In comparison with other hardware FL implementations, this thesis presents a design flow from model to hardware such that the number of design iterations is reduced. This thesis reviews existing hardware FL architectures, presents a proof of concept design to illustrate the challenges, details the coprocessor architecture and software configuration, and finally presents experimentation results using FL case studies.

Acknowledgements

I would like to express my sincere thanks to my thesis supervisor Dr. Miodrag Bolic for his help, guidance, and patience throughout my graduate studies. It has been his support and tutelage which enabled me to pursue my studies farther than I had ever imagined. It has been a pleasure to learn from him and I am honoured to have him as my supervisor, a mentor, and a friend.

I would also like to thank my colleagues at the Computer Architecture Research Group, University of Ottawa, for their support and companionship throughout the many years.

Exceptional and particular thanks go to my parents and brother. Their endless love and support allowed me to pursue all my endeavours in life. I would also like to thank Lindsy for her incredible support and love for all our years together.

Last but not least, a warm thanks to my very close friends for their encouragement that lead me to finish my graduate studies in a timely manner.

Vishal Thareja

Ottawa, September 2010

Table of Contents

Table of Figures

Table of Tables

Table of Algorithms

List of Acronyms and Definitions

ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
FL	Fuzzy Logic
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GPP	General Purpose Processor
GUI	Graphical User Interface
HDL	Hardware Description Language
ISA	Instruction Set Architecture
LPM	Library of Parameterized Modules
PID	Proportional, Integral, and Derivative
PLD	Programmable Logic Device
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RTL	Register Transfer Level
SOC	System on-chip
SPARC	Scalable Processor Architecture

TIE	Tensilica Instruction Extension
VHDL	VHISC Hardware Description Language
VHSIC	Very-High-Speed Integrated Chip
VLSI	Very Large Scale Integration

Chapter 1 - Introduction

1.1 Control Systems and Fuzzy Logic

Control systems have become a part of everyday life. Control systems have been integrated in automobiles we drive, vending machines we use satisfy our sweet tooth, the door entry system where we swipe our access badge to get into our office, even the coffee maker we use to enjoy the morning brew. Control systems have become an industry standard for maintaining and automating control to make our lives more comfortable and enjoyable.

Although control systems are everywhere, the challenge is to design and incorporate control algorithms in embedded systems. As users of control systems, we do not understand the difficulties, challenges, and technical complexity involved in ensuring the coffee maker does not burn the brewed coffee. Many control systems require inputs and feedback from the external world in order to make the correct control decisions.

Since the late 1930's, Proportional, Integral, and Derivative (PID) controllers are the most widely used in process control. A survey done in 1989 determined that in Japan more than 90% of the controllers used in process industries are simple and advanced PID controllers [1]. PID controllers use feedback control to make future control decisions based upon the current control state and the previous control state. This methodology ensures that the PID controller makes correct decisions.

The conventional approach in designing PID controllers involves a deep understanding and knowledge of control theory and mathematical modelling. The first step is to define the physical system and its control requirements. The second step is to model the control system including sensors, actuators, etc. This model is a mathematical description defining all elements as math functions. After defining the model, linear control theory principles are applied to the model to simplify the control system. Next the development of an algorithm for the controller and the last step is to simulate the design for effects of noise

and variation in the inputs and outputs. The performance of the controller is also analyzed and if the performance does not meet design requirements such as performance, area, or power then the model is modified, the controller re-designed, and the algorithm re-written [2].

Although PID controllers have become an industry norm for embedded control systems, there are alternatives. Some alternatives have been driven due to the fact that PID controllers do not always work well and offer no flexibility. Fuzzy Logic (FL) offers a different approach of representing control system behaviour.

Fuzzy Logic is a problem solving methodology for control and information processing by drawing definite conclusions from vague, ambiguous or imprecise information. FL resembles human decision making using approximate data to find precise results. FL theory was developed by Dr. Lofti A. Zadeh in 1964. The goal was to develop a model that could more closely describe the natural language process. The concept of linguistic variables was proposed that equates to a variable defined as a fuzzy set. Although FL was faced with hesitation in North America, FL was accepted in Japan as a new methodology for the design of control systems and many experiments soon followed [3] [4] [5].

To illustrate how FL works, consider the following example: a hot water tank in which water is heated to a desired temperature by using natural gas. The input variable (temperature) is divided into a range of states such as cold, cool, nominal, warm, and hot. Figure 1 illustrates a FL membership function that describes the states of temperature.

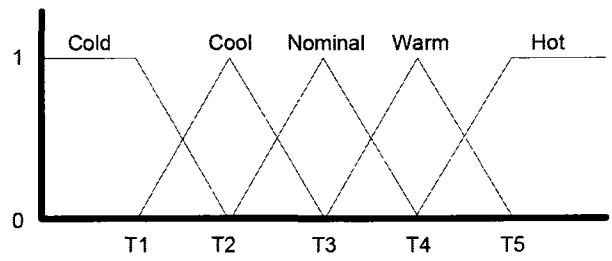

Figure 1: Temperature Input Membership Function

The vertical axis represents the degree for each state [0,...,1] and the horizontal axis represents the input temperatures [T1,...,T5]. The bounds between states can describe the

expected behaviour of the control system. However, by overlapping the shapes allows the input to change without a discontinuous change between states. As the temperature changes, it loses value in one membership function shape while gaining value in the next shape. At any one time, the input temperature is always represented by two membership function states: 0.4 warm and 0.6 hot, etc. Similarly, another input value (error) can be described using a similar membership function.

The inputs are mapped to membership function shapes to derive fuzzy sets. The process of mapping a crisp input value to a fuzzy set value is called fuzzification. After mapping the input to a fuzzy set, the controller needs to make a decision on what action to take based on a set of rules. These rules are known as fuzzy inference rules and are represented as follows:

```
IF temperature IS warm AND error IS not large
THEN flow of natural gas IS slightly increased.
```

In this example, fuzzy sets describe the input temperature and input error. The output, flow of natural gas, is defined by the output fuzzy set. Once the decision is made based on the input fuzzy sets, the output fuzzy set is then mapped back to a crisp output value, the flow of natural gas.

The approach to design a FL controller is relatively simple and straightforward. The first step is to define the physical system and its control requirements. The second step is to design the control algorithm using fuzzy inference rules and membership functions. These rules and functions describe the relationship between inputs and outputs. The last step is to simulate the design for effects of noise and variation on the inputs and outputs. The performance of the controller is also analyzed here and if the performance does not meet design requirements such as performance, area, or power the FL algorithm is modified by changing the fuzzy inference rules or membership function shapes.

FL controllers offer many advantages over PID controllers. First and foremost, Fl reduces and simplifies the design development cycle. FL reduces the number of steps required to design and simulate a control system. It is also easier to modify the controller properties by

adjusting language like fuzzy inference rules or membership function shapes rather than changing the complex mathematical models and then reapplying linear control theory principles. FL design methodology also improves time to market. With the reduction of development time and reduced hardware costs, FL controllers can easily be designed, simulated, and deployed more quickly than PID controllers. Moreover, since FL is based on vague and imprecise inputs it is well suited for low cost implementations using cheap sensors, analog-to-digital converters, etc [6] [7] [8].

1.2 Motivation

Fuzzy Logic design methodology provides more flexibility for designing embedded control systems. Although it is easy to design the FL controller, the challenge lies in the implementation in an embedded system. Implementing FL algorithms on microcontrollers and simple RISC processors is relatively easy because of the general instruction set architecture (ISA). However, the complex FL operations are difficult to handle and require many additional cycles to compute. This in turn can cause FL implementations on microcontrollers and simple RISC processors fail performance constraints.

In academia, dedicated hardware FL controllers have been proposed in [9] [10] [11] [12]. These controllers offer dedicated hardware to target specific FL algorithms implemented on a FPGA. However, these hardware FL controllers are difficult to integrate in larger embedded systems. Many others have also suggested dedicated FL processors [13] [14]. These processors have FL specific instructions to perform FL tasks but the processors are not configurable for different types of FL algorithms. The difficulty in integrating both hardware controllers and processors in an embedded system is due to the fact that these architectures are designed to be independent. The FL controllers do not provide an integration methodology. Furthermore, FL processors do not provide a broad instruction set to tackle other types of tasks.

There are many commercially available tools that help with the design and integration of FL controllers with a processor targeting an embedded control system. Tensilica offers a processor design flow where their proprietary processor can be extended to support custom hardware. As a leader in the configurable and standard processor cores for SoC

design, Tensilica offers a set of configurable processor cores, branded as Xtensa. The two essential features that Xtensa configurable processors have are: configurability and extensibility. The configurability feature allows the processor core to be customized through checkboxes and drop-down menus to select the features developers desire. The extensibility feature gives developers the ability to add multi-cycle units, registers, register files, and other units using Tensilica's proprietary Tensilica Instruction Extension (TIE) language. Using TIE, developers can design and verify the functional behaviour of the custom datapath. Register Transfer Level (RTL) is automatically generated and integrated with the Xtensa processor [15] [16] [17] [18] [19].

First and foremost, using Tensilica's Xtensa processor to support application specific functions in hardware requires the understanding of the TIE language. TIE is similar to Verilog in terms of syntax and coding style. However, the techniques used to describe certain functions are different. Moreover, there are specific Verilog functions that are not supported by TIE. Once the Xtensa processor is configured, Tensilica hardens the core, freezing the configuration. This prevents any alteration once the processor has been generated.

MathWorks provides a toolbox in their Matlab environment that assists in modelling FL algorithms. The Fuzzy Logic Toolbox offers developers the ability to design and model FL algorithms using membership functions and fuzzy inference rules. The model can then be simulated using Matlab's Simulink tool. Matlab provides an excellent environment to design and simulate FL algorithms; however challenges arise in porting the design model into an embedded control system. Matlab offers processor independent FL controllers and allows developers to export their FL algorithm model into VHDL using Simulink. Simulink's HDL Coder function generates cycle accurate synthesizable Verilog and VHDL code. The drawback of using HDL Coder is its inefficient generation of Verilog or VHDL code from Matlab models. The code generated is non-optimized HDL code that is difficult to understand, modify, and integrate in an embedded control system [20] [21] [22] [23].

1.3 Research Problem & Objective

There are many different solutions and implementations of FL algorithms in embedded control systems. These implementations are dedicated and meet the specified design requirements, but offer no flexibility or configurability. The general trend in FL controllers in hardware is application specific. Each application consists of a different FL controller implementation. In industry, commercial tools are available to design FL controllers integrated within a processor to achieve greater flexibility. This is an attractive solution, but developers are required to design their own FL hardware in a black box design methodology. There are also tools that assist in modelling FL algorithms and generating HDL code. Again attractive, but difficulties arise in modification and integration of the HDL code within an embedded control system.

The focus of this work lies in proposing a different design methodology for FL algorithms. The goal here is to provide a flexible and configurable FL controller that can execute different types of FL algorithms. Moreover, to simplify integration within an embedded control system, the FL controller is automatically coupled with a general purpose processor. The key here is to provide automation in the design generation, integration, and provide an open source approach where design modifications can be made at any step in the design flow.

The Leon3 soft processor [24] is selected to be extended to support a configurable hardware controller for FL algorithms. The Leon3 is an open source processor based on the SPARC V8 architecture [25]. The Leon3 processor supports coprocessor instructions that can facilitate FL controller implemented as a coprocessor. The design flow will begin by modelling the FL algorithm using Matlab's Fuzzy Logic Toolbox. The model is then imported into an automated HDL generator that generates the FL controller. The communication between the Leon3 and the FL controller is done through a control interface where coprocessor instructions are decoded and executed by the FL controller. This interface couples the FL controller with the Leon3. The hardware integration and synthesis are performed automatically using software algorithms and synthesis tool scripts.

1.4 Research Contributions

This thesis presents a design methodology where Fuzzy Logic algorithms can be executed in hardware instead of software. The hardware datapath is coupled with the Leon3 soft processor that can be configured automatically using models described by Matlab's Fuzzy Logic Toolbox. This is achieved by five objectives and is presented in this thesis. The contributions are:

1. A hardware datapath for FL algorithms that is synthesizable for a FPGA, while configurable for any type of two input and one output FL algorithm supporting triangular and trapezoid shapes and to be integrated with a host processor.

2. Extend the Leon3 processor to support coprocessor instructions.

3. Software algorithms that are able to extract model properties from MATLAB's Fuzzy Logic Toolbox that configures the hardware datapath for FL algorithms.

4. Software scripts that are able to build, package, and compile the hardware datapath and the Leon3 processor using Altera Quartus II synthesis scripts.

5. Software scripts that generate a test environment for the processor with hardware datapath using ModelSim verification environment.

1.5 Research Publications

The peer reviewed publications related to the field of fuzzy logic architectures are given below.

Conference Publications:

Thareja, V.; Montcalm, M.; Bolic, M.; , "Configurable Fuzzy Logic coprocessor for small scale food preparation," *Soft Computing Applications, 2009. SOFA '09. 3rd International Workshop on* , vol., no., pp.229-234, July 29 2009-Aug. 1 2009

Thareja, V.; Bolic, M.; Groza, V.; , "Design of a Fuzzy Logic Coprocessor using Handel-C," *Soft Computing Applications, 2007. SOFA 2007. 2nd International Workshop on* , vol., no., pp.83-88, 21-23 Aug. 2007.

Competitions

Thareja, V., Shapiro, D., Ratti, S., Vijayakumar, S., Yang, M., Bolic, M., "Fuzzy Logic Coprocessor for Leon3," Second place, Innovate Canada FPGA Competition, 2008.

1.6 Thesis Outline

The majority of this thesis presents the design methodology and architecture of the Fuzzy Logic hardware datapath. Accordingly, the remainder of this thesis is structured as follows:

Chapter 2 – Background provides background information on Fuzzy Logic algorithms, Algorithmic Transformations, Leon3 soft processor, and Matlab's Fuzzy Logic Toolbox.

Chapter 3 – Related Work discusses existing approaches taken by a variety of Fuzzy Logic algorithms implemented in hardware using dedicated and general purpose architectures.

Chapter 4 – Proof of Concept presents implementation of Fuzzy Logic algorithms in hardware, evaluating the architecture, identifying design limitations, and proposing improvements.

Chapter 5 – Architecture presents the hardware architecture for implementing Fuzzy Logic algorithms.

Chapter 6 – Software Configuration describes the software configuration tool used to configure, package, and compile the Leon3 processor with Fuzzy Logic hardware datapath.

Chapter 7 – Results presents the evaluation results of simulations and implementation are carried out to determine the performance of the hardware datapath.

Chapter 8 – Conclusion summarizes and concludes the thesis while outlining venues for future research.

Chapter 2 - Background

2.1 Fuzzy Logic

A FL algorithm consists of three stages where imprecise inputs are interpreted and a definite conclusion is produced: Fuzzification, Inference Engine, and Defuzzification as illustrated in Figure 2. The Fuzzification stage maps a crisp input value to a fuzzy set. A fuzzy set is a representation of a crisp value using membership functions. The Inference Engine stage compares fuzzy sets from the Fuzzification step against the fuzzy inference rule database. The fuzzy inference rules describe the behaviour of the FL algorithm. The Defuzzification stage maps the output fuzzy set back to a crisp output value. There are many methods for mapping the fuzzy set to a crisp value. The variety of methods allows the developers a choice of how precise the output of the FL algorithm should be [26].

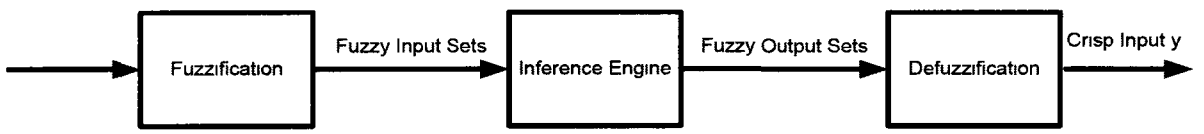

Figure 2: Fuzzy Logic System

The conventional approach to design a controller involves mathematical models and characterization of system parameters. The first step is to define the physical system and control constraints. Next, a mathematical model is developed to include components such as sensors and actuators. The third step is to use linear control theory to simplify the mathematical model. After the simplification, an algorithm is implemented to run on the controller. Simulation helps identify effects such as noise and parameter variations and any constraint violations. If the performance of the controller does not meet the performance requirement, the developer would go back to the mathematical model to redefine the system parameters.

To design a controller using FL, the first step is to define the inputs and outputs of the system and the required system behaviour. Then the behaviour of the system is described using membership function shapes and fuzzy inference rules. Since FL works on the principle of vagueness, the system behaviour can be easily described using fuzzy inference rules. The final step is to simulate the design and identify any errors or undesirable system behaviours. If the simulation does not meet system requirements and constraints, then the behaviour of the system can easily be changed by modifying the membership function shapes or fuzzy inference rules [27] [28].

The comparison between the traditional approach and the FL approach is illustrated in Figure 3.

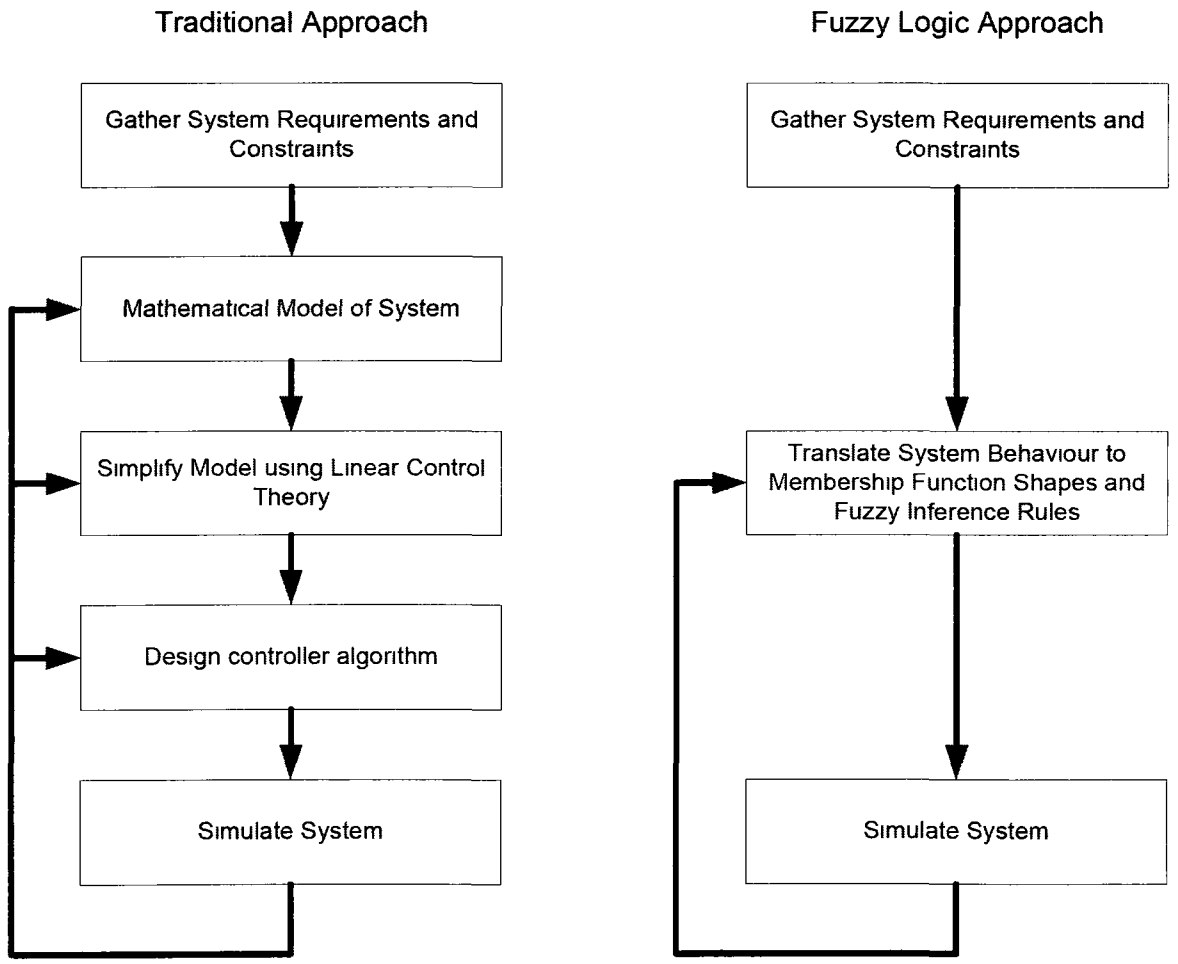

Figure 3: Traditional Approach vs. Fuzzy Logic Approach

The FL approach reduces the design complexity by using membership function shapes and fuzzy inference rules to describe the system behaviour. No mathematical models or linear control theory principles are needed, just simple human reasoning. The design can easily be modified by manipulating the simple "if-then" fuzzy inference rules. Most importantly, the FL approach reduces the time-to-market ratio. Since physical processes are difficult to model using mathematical models, geometric shapes describing membership functions and "if-then" fuzzy inference rules can quickly describe the behaviour of the system and be verified to meet system requirements and constraints [29].

2.1.1 Fuzzification

Fuzzification is the process where a crisp value is mapped to a fuzzy set. A fuzzy set is a generalization of a crisp value. A fuzzy set [*F*] allows a crisp element [*x*] to belong to a fuzzy set with a membership degree. The function to map a crisp element [*x*] to a fuzzy set [0, 1] is defined as a membership function $u_F(x)$. The membership function allows the measure of the degree of similarly of the crisp element [*x*] in the fuzzy set. The fuzzy set [*F*] can be represented as a set of pairs [*x*] and its membership function $u_F(x)$ as:

$$F = \{(x, u_F(x)) \mid x \in X\} \tag{1}$$

Some common membership function shapes are triangular, trapezoidal, S-shaped, Gaussian, and bell-shaped as shown in Figure 4. Membership function shapes can be arbitrary such that the shape can be optimized for the particular algorithm. Moreover, the number of membership functions for FL is not defined. The number of membership functions depends on the resolution of fuzzy sets required for the algorithm. To increase the resolution, more membership function shapes can be used at the price of higher computational complexity. Membership function shapes do not have to overlap, but having shapes overlap distributes the fuzzy set over many shapes making the FL algorithm robust [30].

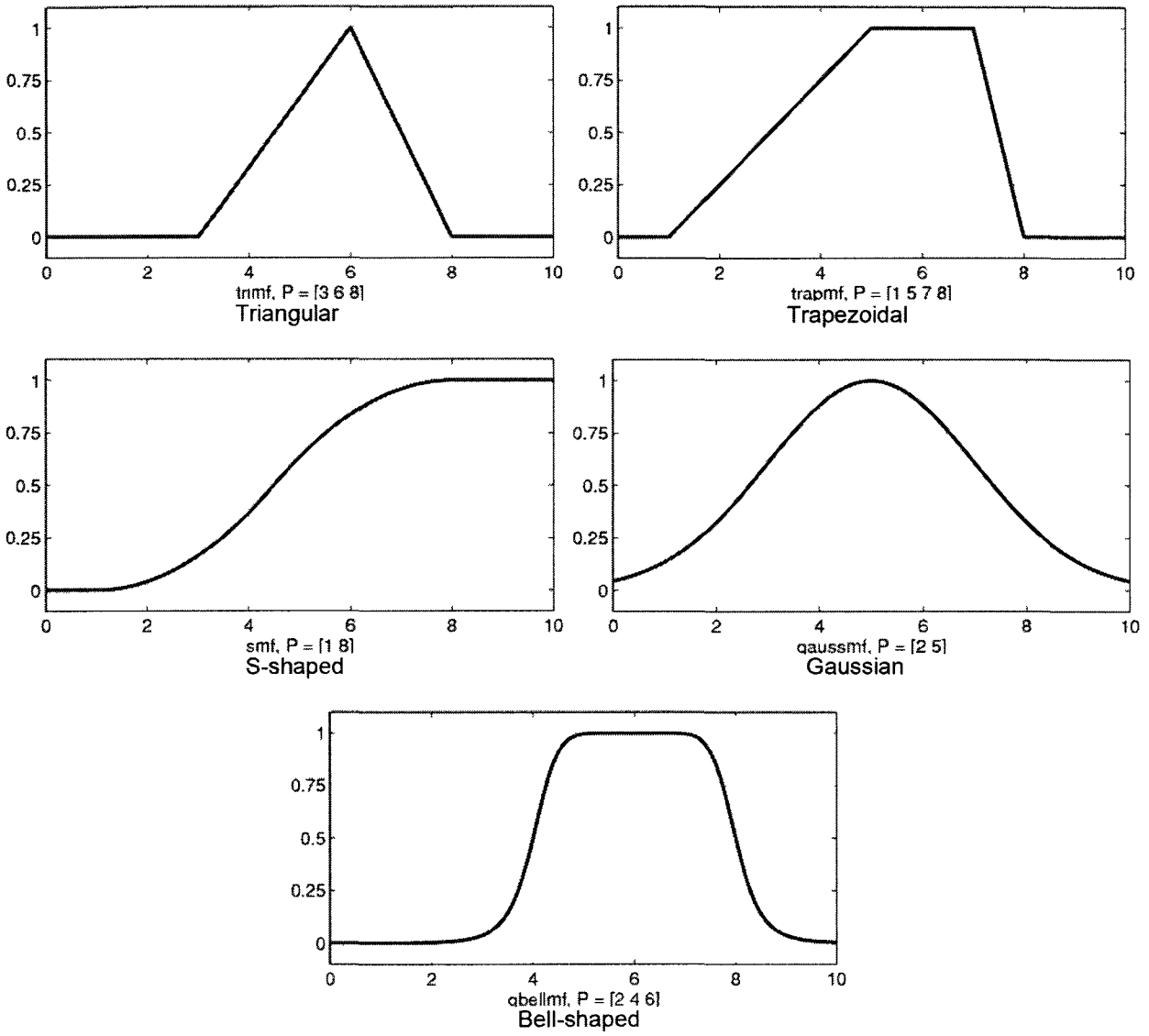

Figure 4: Types of Membership Function Shapes [21]

With many membership function shapes overlapping, a fuzzy set can be described using some elementary fuzzy-set operations. These operations include union, intersection, and complement as shown in Figure 5. Let A and B represent their membership functions $u_A(x)$ and $u_B(x)$ and x represent an element in the fuzzy set. The union of A and B, denoted as $A \cup B$, is represented as:

$$u_{A\cup B}(x)=\left\{\begin{array}{ll} 1 & \text{if } x\in A \text{ or } x\in B \\ 0 & \text{if } x\notin A \text{ and } x\notin B \end{array}\right\} \tag{2}$$

$$u_{A\cup B}(x)=\max[u_A(x),u_B(x)] \tag{3}$$

The union of two fuzzy sets is defined as the maximum of the two membership function shapes, called the maximum criterion. The fuzzy set union operation is the equivalent of the *OR* operation in Boolean algebra [30].

The intersection of A and B, denoted as $A\cap B$ is represented as:

$$u_{A\cap B}(x)=\left\{\begin{array}{ll} 1 & \text{if } x\in A \text{ and } x\in B \\ 0 & \text{if } x\notin A \text{ or } x\notin B \end{array}\right\} \tag{4}$$

$$u_{A\cap B}(x)=\min[u_A(x),u_B(x)] \tag{5}$$

The intersection of two fuzzy sets is defined as the minimum of the two membership function shapes, called the minimum criterion. The fuzzy set intersection operation is the equivalent of the *AND* operation in Boolean algebra [30].

The complement operation, denoted as $\overline{A}$ is represented as:

$$u_{\overline{A}}(x)=\left\{\begin{array}{ll} 1 & \text{if } x\notin A \\ 0 & \text{if } x\in A \end{array}\right\} \tag{6}$$

$$u_{\overline{A}}(x)=1-u_A(x) \tag{7}$$

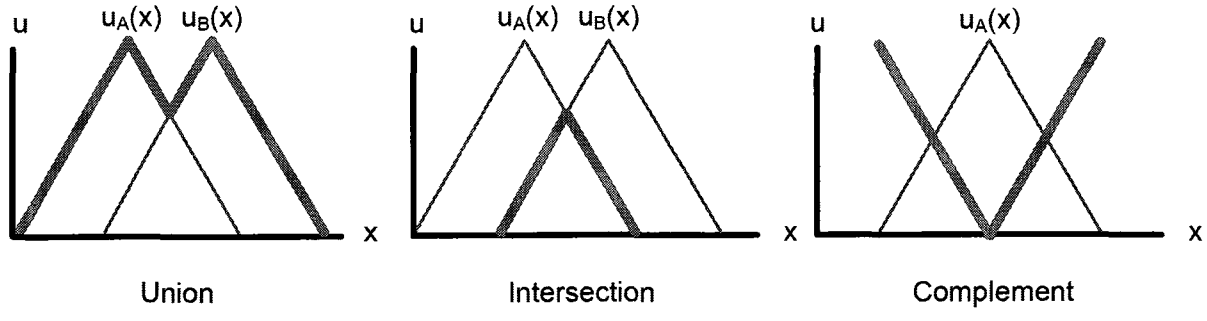

Figure 5: Fuzzy Set Operations

2.1.2 Inference Engine

The inference engine is based on fuzzy logic principles to evaluate fuzzy inference rules with fuzzy input sets to produce fuzzy output sets. The inference engine is comprised of simple "if-then" rules. The condition evaluated is commonly referred as the consequent and the output of the rule is known as the antecedent. Consider the following:

$$R^n \Rightarrow \text{IF } x_1 \text{ is } F_1^n \text{ and} \ldots \text{and } x_m \text{ is } F_m^n \text{ THEN } y \text{ is } G^n \tag{8}$$

The above definition describes a fuzzy inference rule that can be found in the inference engine. For each rule n, there can be multiple-antecedents and a single consequent. Any membership function that describes the inputs, F or output, G is defined by m. The consequent of each rule is described by a singleton y [30].

A fuzzy inference rule antecedent can be defined by the input membership functions and fuzzy operators. The consequent can be defined by the output membership functions.

$$R^n \Rightarrow \text{IF } x_1 \text{ is } F_1^n \text{ AND } x_2 \text{ is } F_2^n \text{ THEN } y \text{ is } G^n \tag{9}$$

$$R^n \Rightarrow \text{IF } x_1 \text{ is } F_1^n \text{ OR } x_2 \text{ is } F_2^n \text{ THEN } y \text{ is } G^n \tag{10}$$

$$R^n \Rightarrow \text{IF } x_1 \text{ is NOT } F_1^n \text{ THEN } y \text{ is } G^n \tag{11}$$

A fuzzy inference rule can contain multiple antecedents that can be a combination of union, intersection, and complement fuzzy operators.

2.1.3 Defuzzification

Defuzzification is the process of mapping the output fuzzy set from the Inference Engine to a crisp or real value. There are many defuzzification techniques that vary in computational complexity. Some examples of defuzzification techniques are maximum, centroid, center-of-sums, height, etc.

The centroid defuzzification technique combines the output fuzzy sets from the Inference Engine using the Union fuzzy operator and then determines the centroid of the fuzzy set. The centroid of the output membership function is given as:

$$Y_{centroid}(x) = \frac{\sum_{i=1}^{N} y_i \cdot u_i(x)}{\sum_{i=1}^{N} u_i(x)} \tag{12}$$

The output of each fuzzy set *y* is multiplied by the union defined as *u(x)* for each fuzzy set output *i* up to *N* fuzzy sets. The union for each fuzzy set needs to be computed before determining the centroid of the output membership function [30].

The bisector defuzzification technique divides the area under the output membership function shape defined by the fuzzy output sets into two regions of equal area. It is but not always coincident with the centroid defuzzification technique.

$$Y_{bisector} \rightarrow \int_{\alpha}^{Y_{\text{bisector}}} u(x)dx = \int_{Y_{\text{bisector}}}^{\beta} u(x)dx \tag{13}$$

The minimum and maximum points of the area of the output membership function shape are defined by α and β.

Some simpler defuzzification techniques are Middle, Smallest, and Largest of Maximum. These techniques determine the maximum value of the output membership function shape. There are scenarios where if there is only one maximum point, then the Middle, Smallest, and Largest of Maximum (MOM, SOM, LOM) are equal. Another scenario is where there is a plateau where the maximum spans multiple values. In this case, MOM, SOM, and LOM would be middle, smallest, and largest of the plateau. Examples of the different defuzzifcation technique are shown in Figure 6 [30].

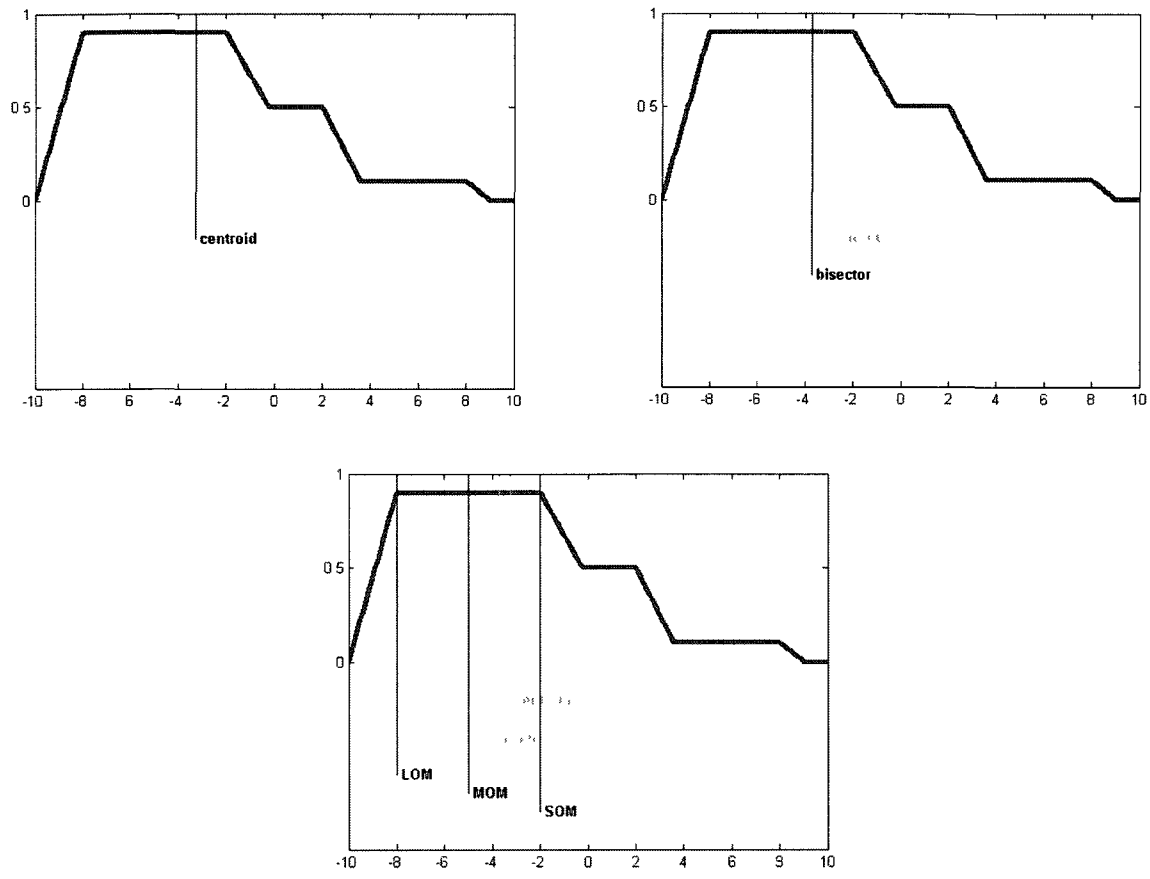

Figure 6: Defuzzification Techniques [21]

2.2 MATLAB's Fuzzy Logic Toolbox

MATLAB's Fuzzy Logic Toolbox is an extension to the MATLAB environment that provides a graphical user interface (GUI) for describing a FL system. The toolbox assists in developing a model of a control system with simple logic rules and membership function shapes. The toolbox can design a stand-alone FL inference engine or can be integrated within a model of a dynamic system using MATLAB's Simulink tool. The toolbox allows for modeling membership function shapes using a graphical method. The toolbox supports intersection, union, and complement fuzzy operators and Mamdani or Sugeno-type inference engine [20] [21].

For illustration purposes, a two input and one output inverted pendulum controller is designed using Matlab's Fuzzy Logic Toolbox. The two inputs are *angle* and *velocity* and the one output is *force*. The control system behaviour will be described using 25 fuzzy inference rules.

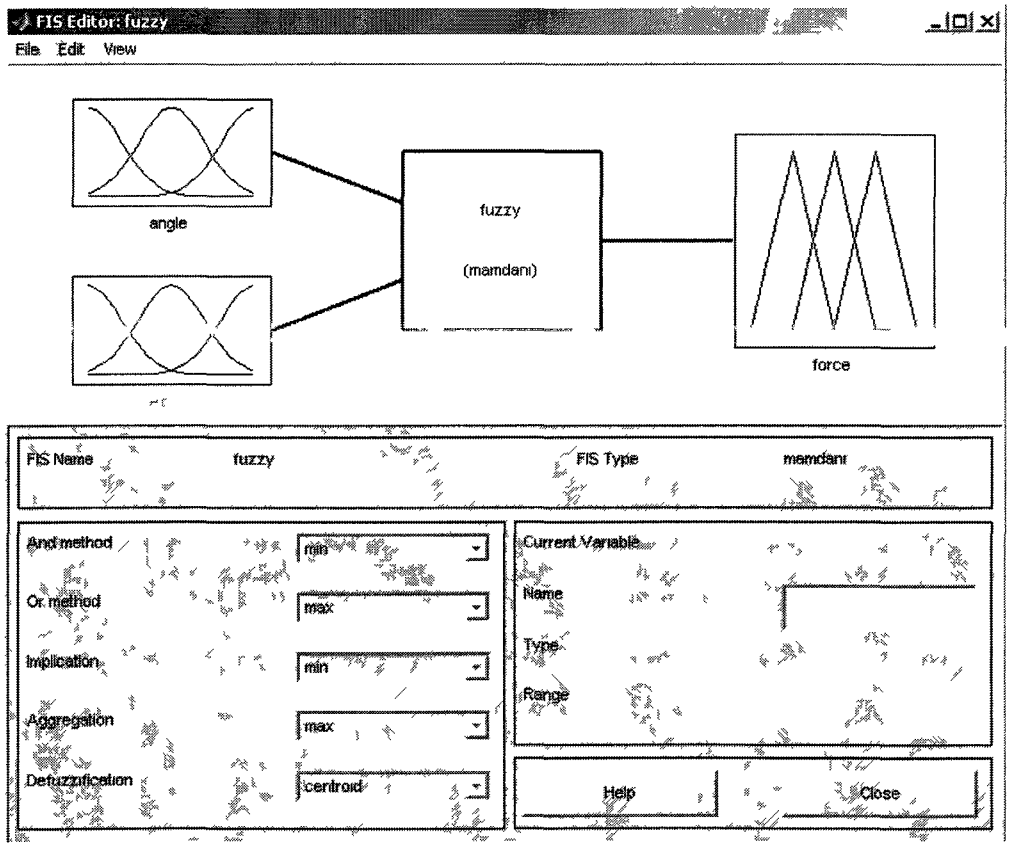

Figure 7: MATLAB's Fuzzy Logic Toolbox

To use the toolbox, the first task the assignment of how many inputs and outputs are there in the FL system. The fuzzy operators and defuzzification method are also chosen before defining the membership function shapes and fuzzy inference rules. The high-level system description is illustrated in Figure 7.

For each input in the FL system, membership function shapes can be defined. The toolbox supports triangular, trapezoidal, Gaussian, S-shaped, Bell shaped, and user defined membership function shapes. The shapes can be described graphically as shown in Figure 7. Membership function shapes can be overlapped and the range for the input variable can be altered. Similarly, the outputs can also be described using the membership function editor.

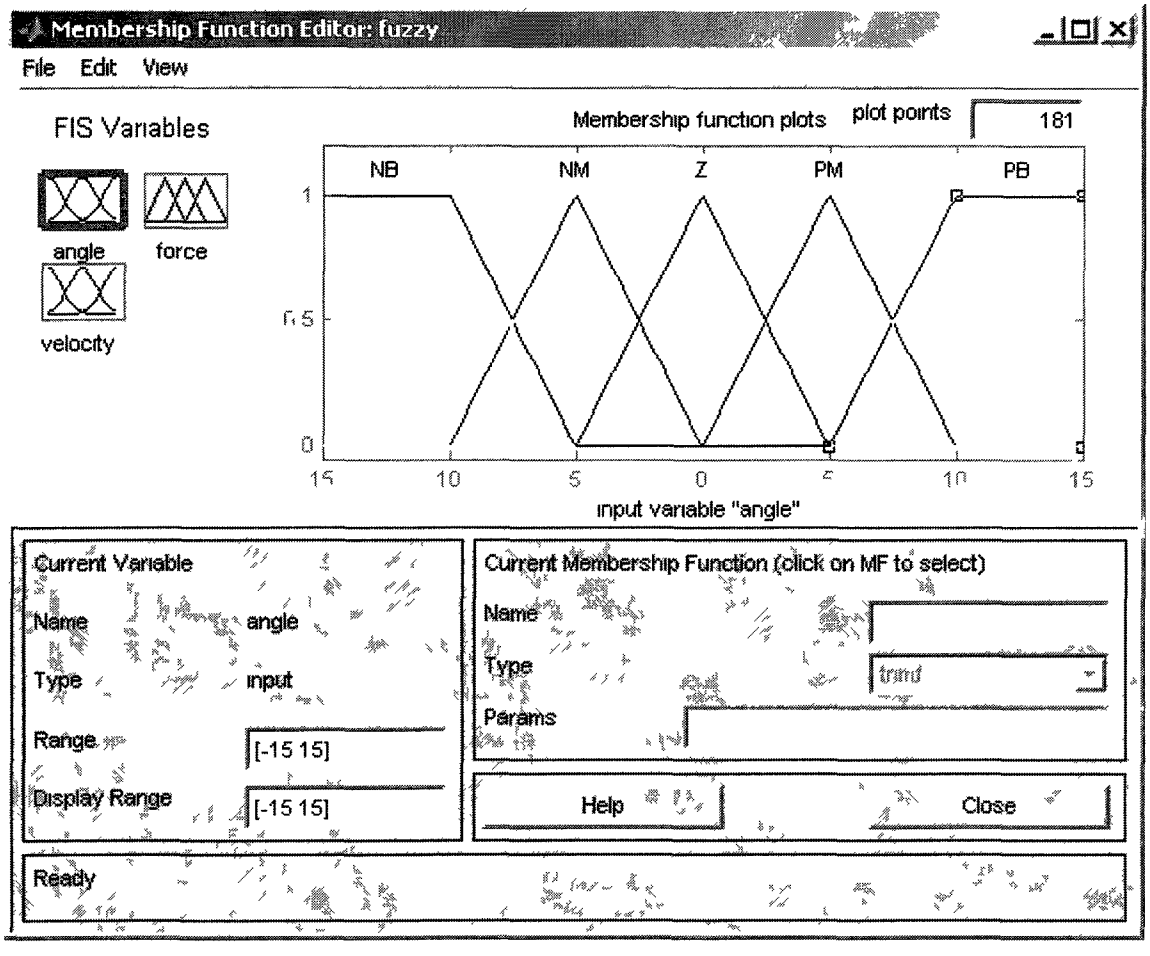

Figure 8: Fuzzy Logic Toolbox Membership Function Editor

The rule editor allows the addition of fuzzy inference rules to the FL system. Each fuzzy inference rule can be added, changed, and removed from the FL system. To add a fuzzy inference rule to the FL system, a membership function shape is selected for each input to create the implication. The connections between all inputs are interchangeable and complements are shown in Figure 9. The consequent is selected by selecting a membership function shape from the output.

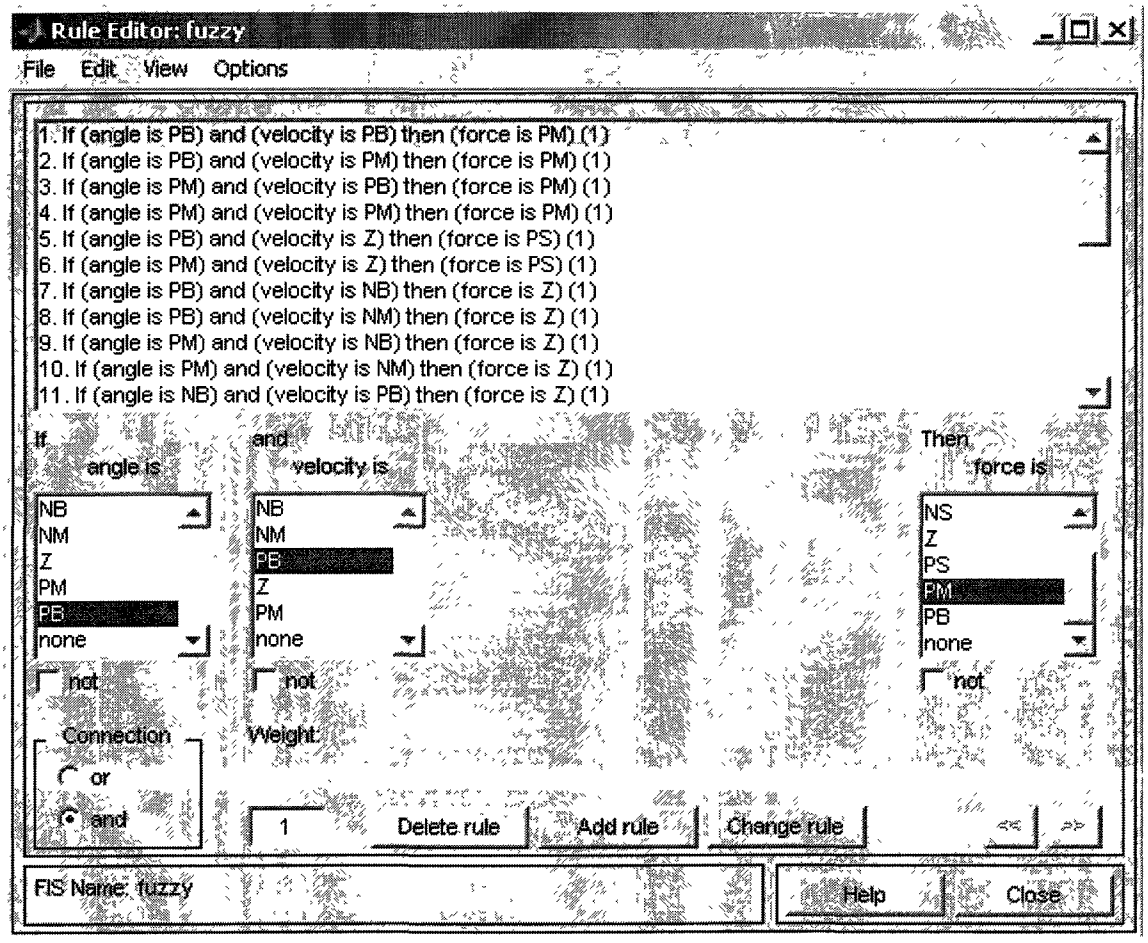

Figure 9: Fuzzy Logic Toolbox Rule Editor

With the input and output membership function shapes defined and the fuzzy inference rules added to the FL system, the model is now ready to simulate. The toolbox offers a graphical method to simulate the FL system, where the inputs can be altered in real-time and the output of the FL system can be observed immediately. The rule viewer as seen in

Figure 10 can alter the input values to the FL system and show what membership functions are covered.

To supplement the rule viewer, the surface viewer of the FL system illustrates the surface coverage of the FL system based on inputs and outputs. The 3D plot view, shown in Figure 11, can be altered by changing the axis orientation.

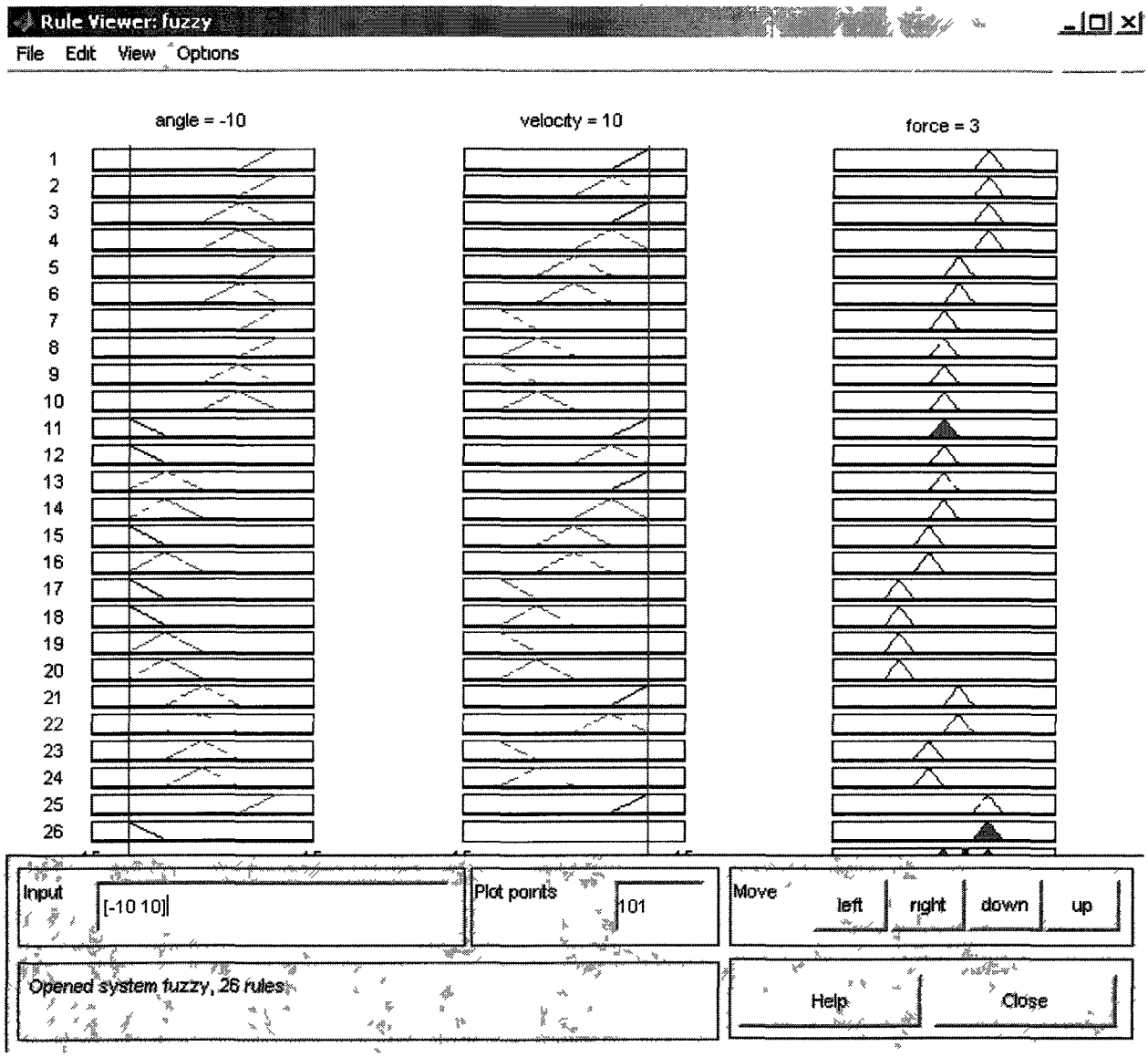

Figure 10: Fuzzy Logic Toolbox Rule Viewer

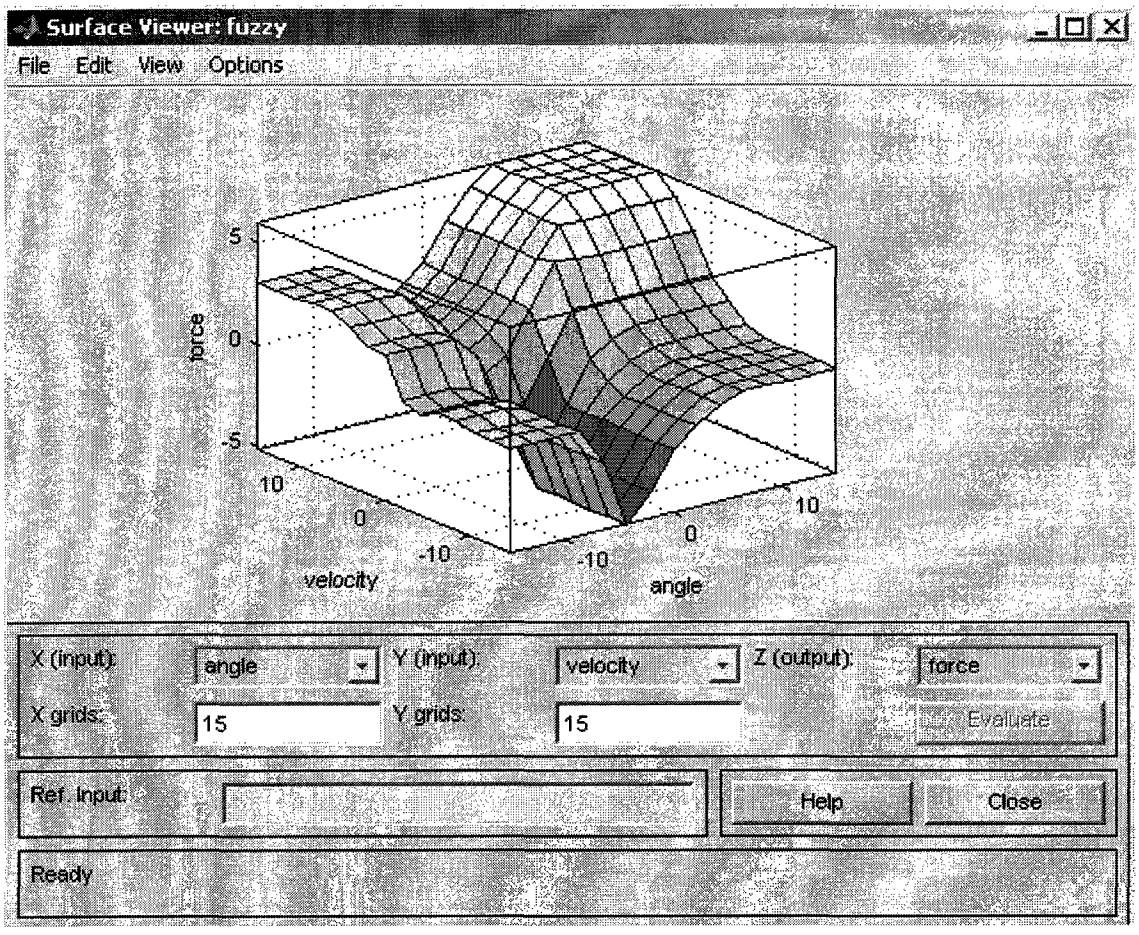

Figure 11: Fuzzy Logic Toolbox Surface Viewer

2.3 Algorithmic Transformations

Algorithmic transformations are important techniques used by developers to help efficiently run algorithms in hardware. The goal here is to obtain the most suitable transformation for a particular implementation. Algorithmic transformations do not change the functionality of the algorithm, but changes the hardware architecture.

2.3.1 Definitions

To describe the different algorithmic transformation techniques, some basic definitions will be presented. To help understand the definitions, a three tap FIR (Finite Impulse Response) filter example will be used.

$$y[n] = b_0 x[n] + b_1 x[n-1] + b_2 x[n-2] \tag{14}$$

2.3.1.1 *Iteration*

Iteration is the single execution of all components in an algorithm. Looking at the above example, the filter takes in one input signal, and then performs three multiplications and two additions. Therefore, in one iteration, three multiplications and two additions are executed [31].

2.3.1.2 *Critical Path*

Critical path is the longest path between serial elements, or the longest combinational path. You can compute the critical path anywhere in the circuit and it is not necessarily between the input and output. The critical path can be found in combinational logic between registers [31].

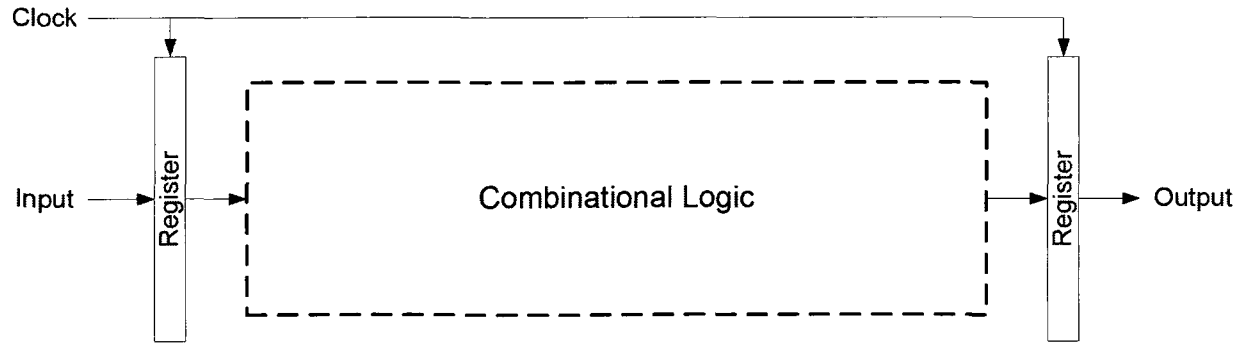

Figure 12: Critical Path between input and output

2.3.1.3 *Iteration Period*

Iteration period is time to execute a single iteration. To compute the iteration period, simply add up all the hardware delays of all the components in the critical path. For instance, if the delay of the multiplier is 2ns and the adder 1ns, then the iteration period will be 8ns when three multiplications and two additions are preformed [31].

2.3.1.4 *Latency*

Latency is defined to be the delay between the input and output. Latency of a circuit can equal to its critical path, however this is not always the case. For example, within the combinational circuit, delays are assigned to each component and then the latency can be computed with the summation of all the delays. With a sequential circuit, the latency can be computed as the number of clock cycles for the circuit to execute [31].

2.3.2 Pipelining

Pipelining is a technique often used to improve efficiency and throughput of a hardware circuit. Pipelining is performed by placing registers or latches at intermediate points in the hardware circuit. This leads to the reduction of the critical path and improvement in clock speed. However, pipelining increases data dependencies between instructions [31].

Synchronous pipelining is a traditional way of pipelining combinational logic. Synchronous pipelining is achieved by placing registers or latches at the beginning and the end of specific combinational logic that are synchronous on the same clock. The benefits of this is that as the combinational logic is running, the next input can be taken and stored, until the next clock edge [32].

Reducing the critical path, increasing throughput, and improving clock speed are potential approaches to meet design constraints such as performance, area, and power, but at what cost? With the addition of new registers, the number of hardware components increases causing the hardware circuit to be more bulky in area [33].

There is a limitation on when and how often pipelining can be applied to a hardware circuit. When the circuit path in the hardware circuit is less than the input/output delay (I/O) it is known that the hardware circuit is communication bounded because the I/O

delay is fixed. Once the critical path delay is equal to the I/O delay, pipelining can no longer increase the speed of the hardware circuit [31].

2.3.3 Parallel Processing

Parallel processing is a technique where one large serial task is broken up into smaller manageable tasks. Multiple outputs are computed in parallel in a clock period. Parallel processing ensures that there will be less interprocessor communication and no data dependencies, but additional hardware is required to achieve parallel processing.

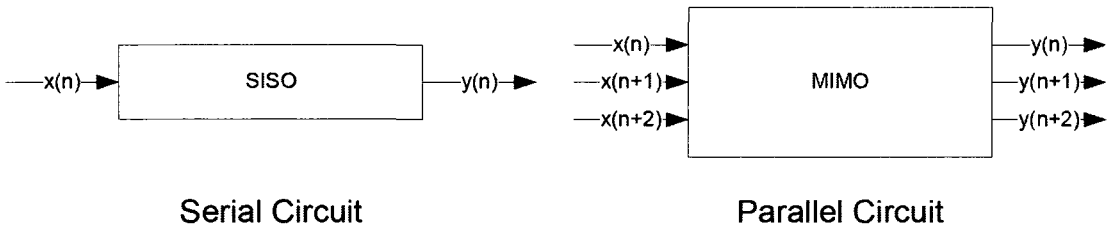

Figure 13: SISO and MIMO systems

As shown in Figure 13, parallel processing takes a traditional Single-Input-Single-Output (SISO) system and converts it to a Multiple-Input-Multiple-Output (MIMO) system. The parallel circuit contains multiple smaller sequential circuits rather than one large serial circuit. Note here that the critical path of the hardware circuit remains unchanged. This is so because in the parallel circuit, three samples are processed in one clock cycle, not three clock cycles when using sequential processing. It is important to note that when applying parallel processing to a hardware circuit, the sampling rate is not equal to the clock rate, whereas when applying pipelining, the sampling rate is equal to the clock rate [31].

Applying both pipelining and parallel processing to a hardware circuit allows a developer to meet design requirements such as performance or area utilization. Combining parallel processing and pipelining, the sample period of the hardware circuit can be quantified as:

$$T_{iteration} = T_{sampling\_period} = \frac{T_{clock\_period}}{L \bullet M} \tag{15}$$

where, L is the block size for parallel processing and M is the number of pipeline stages [31].

2.4 LEON3 Processor

The Leon3 processor is a soft IP processor compliant with the SPARC V8 architecture. The processor designed by Areoflex Gaisler is a synthesizable VHDL model of the SPARC architecture that is configurable and suitable for SoC designs. The Leon3 processor contains a seven stage pipelined datapath that supports hardware multiplication and division units, high performance floating-point unit (FPU), and coprocessor instructions. The processor supports an AMBA 2.0 AHB bus interface with on-chip debug support. The Leon3 can be synthesized for a FPGA with a clock rate of 125MHz or can be synthesized for 130nm ASIC technologies with a clock rate of up to 400MHz [24] [34] [35].

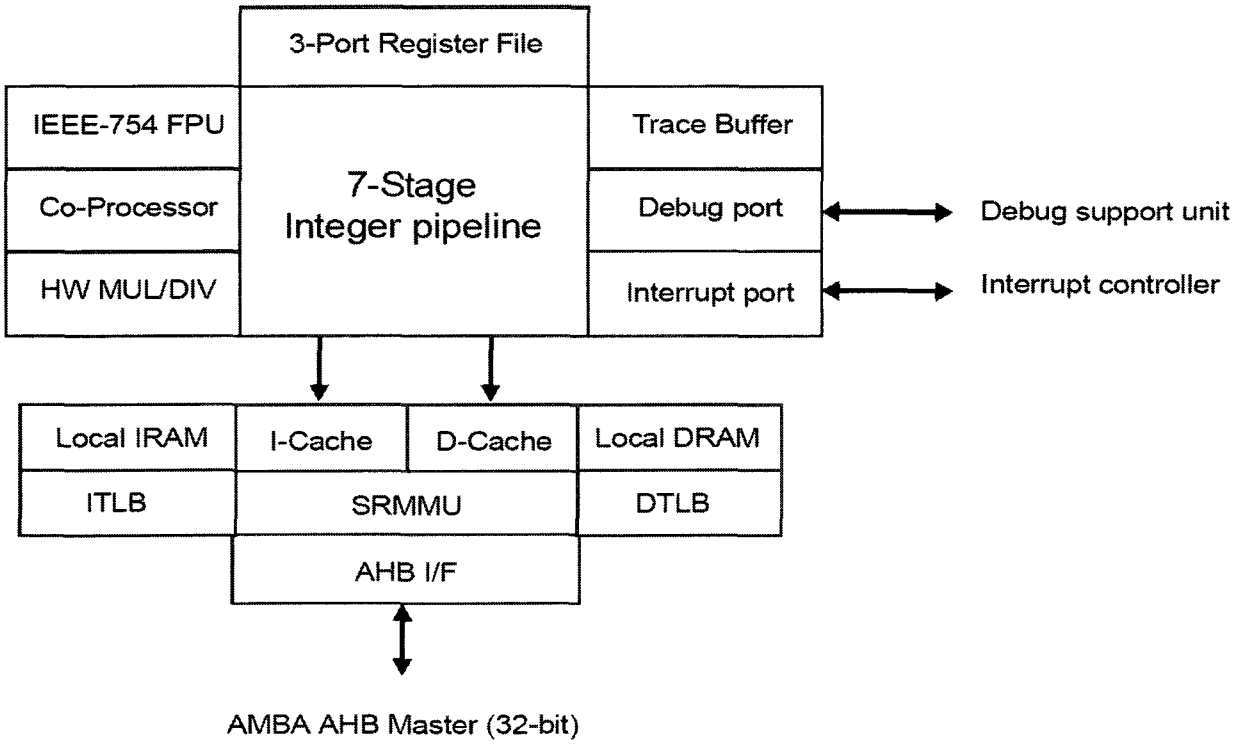

Figure 14: Leon3 Processor Block Diagram [24]

2.4.1 Integer Unit

The Leon3 processor's integer unit contains seven stages; Instruction Fetch (FE), Decode (DE), Register access (RA), Execute (EX), Memory (ME), Exception (XC), and Write (WR). The Leon3 Integer Unit (IU) first fetches the instruction from the instruction cache or from the memory controller if there is a cache miss. Next, the instruction is decoded and branch

addresses are generated. General purpose registers are accessed in the RA stage. The EX stage performs ALU, logic, and shift operations. Results from the EX stage are written in the data cache. If any exceptions occur in the previous stages, the XC stage catches the exception. Finally, the result from the EX stage is written back to the register file in the RA stage.

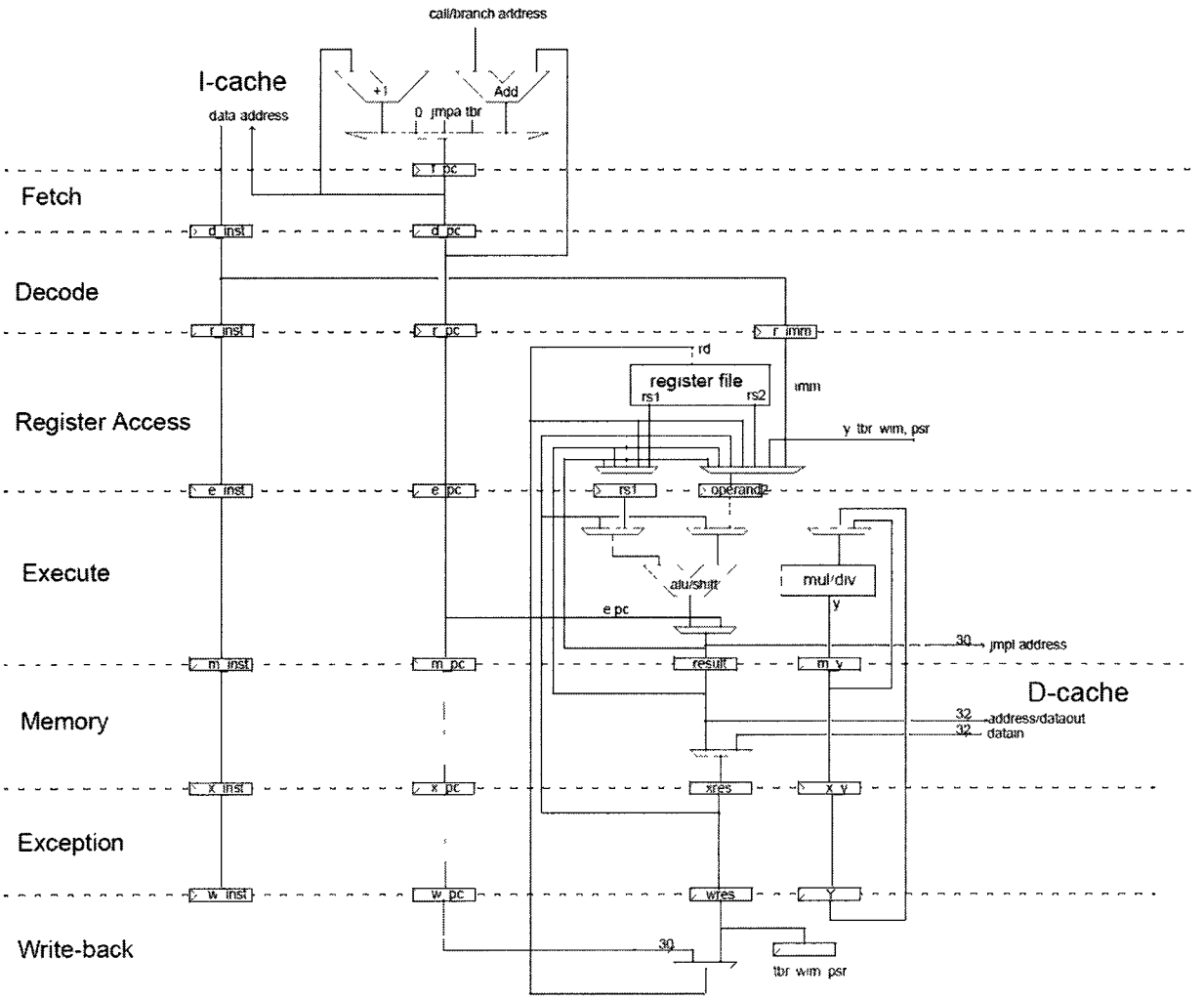

Figure 15: Leon3 Integer Unit datapath [24]

2.4.2 3-Port Register File

The 3-Port Register File is the Leon3's register file where operands are loaded and stored during execution of processor instructions. The register file has two read ports and one write port. Each port has a separate address bus that is eight bits wide and a separate data bus that is 32 bits wide. The register file can support up to 256 words. On the rising edge of

clock, all inputs are latched and the read data on the output ports appears after the clock rising edge. The register file is configurable by using VHDL generics.

2.4.3 IEEE-754 FPU

The Leon3 soft processor includes a high performance floating-point unit (GRFPU) that supports floating-point operations as defined by IEEE-754 standard. The FPU supports single and double precision floating-point numbers and the hardware design combines two execution units for most common floating-point operations and for non-blocking operations such as divide and square-root.

The GRFPU supports arithmetic, compare, convert and move floating-point operations. Arithmetic floating-point operations include addition, subtraction, multiplication, division and square-root. The conversion operations provide conversion between different floating-point numbers and between floating-point numbers and integers.

The GRFPU is interfaced with the Leon3's IU using the GRFPU Control Unit (GRFPC). The GRFPC performs scheduling, decoding and dispatching of floating-point operations to the GRFPU. The GRFPC contains a floating-point register file that is used for floating-point operations [36]

2.4.4 Co-processor

The Leon3 IU provides an interface for a custom co-processor that can support SPARC based coprocessor instructions. The co-processor executes in parallel with the integer unit and does not block the operation unless there is a data dependency.

2.4.5 Hardware Multiplier and Divider

The hardware multiplier is a configurable module that performs 32 bit multiplication. The multiplier supports two signed or unsigned inputs and produces a 64 bit output. The latency and complexity varies depending on the configuration. The hardware multiplier can be configured to perform a multiply & accumulate (MAC) operation usually found in DSP applications. For a MAC operation, a 16 bit multiplication is performed and then the 32 bit result is added to a 40 bit accumulator.

The hardware divider performs 32 bit division using a non-restoring iterative division algorithm. The hardware divider is not configurable and takes 36 clock cycles to execute. The divider supports two signed or unsigned inputs and produces a 32 bit output.

2.4.6 Trace Buffer

The Leon3 supports an instruction trace buffer that stores executed processor instructions in a circular buffer. The trace buffer is accessed through the debug support interface and does not affect the processor operations. The trace buffer is configurable using VHDL generics and stores the following information: instruction address and opcode, instruction result, load/store data and address, trap information and a time tag.

2.4.7 Debug Port and Interrupt Port

The Leon3 debug port provides an interface to the Debug Support Unit (DSU) that during debug mode the processor is controlled through this debug port interface. The DSU accesses the Leon3 through the AHB slave interface where the DSU acts as an AHB master and can access processor registers and instruction trace buffer. In debug mode, the processor pipeline is held and the processor can move into debug mode on the following events: breakpoint instructions, hardware breakpoints, software trap and many other processor debug events.

The interrupt port allows the Leon3 to support 15 asynchronous interrupts Along with interrupts; SPARC processor exceptions are also handled.

2.4.8 SPARC V8 Reference Memory Management Unit

The SRMMU is an optional memory management unit that provides mapping between multiple 32 bit virtual address spaces and the 36 bit physical memory address. When the SRMMU is enabled, the cache tags store the virtual address. The SRMMU implements a Translation look-aside buffer (TLB) for instructions and data used for virtual address spaces.

2.4.9 Local Instruction ROM and Local Data RAM

The Leon3 can support a local instruction ROM and local data RAM for instruction and data without using an external memory on the AHB bus interface. The local instruction ROM is implemented as a 32 bit on-chip ROM with an AHB slave interface. The local instruction

ROM supports byte and half-word access and can be loaded with an initial translated program.

The local data RAM is implemented as a 32 bit on-chip RAM with AHB slave interface. The memory size of the local data RAM is configurable through VHDL generics. The local data RAM supports byte and half-word access and can be loaded with an initial translated program.

2.4.10 Instruction and Data Cache

The Leon3 instruction and data cache can be configured as a direct-mapped or a multi-set cache. The size of the caches is configurable and is divided into cache lines of 16 or 32 bytes. Each cache tag consists of a tag field, valid field with one valid bit.

The instruction cache uses streaming during line refill to minimize latency and the data cache snoops on the AHB bus. On an instruction cache miss, the instruction is fetched and the corresponding tag and data line is updated. On a data cache miss, four bytes of data are loaded into the cache from main memory.

2.5 FPGA development using VHDL

For FPGA development, VHDL or VHSIC Hardware Description Language is among the first choice for design entry. VHDL offers a standardized modelling language for documentation and simulation of circuits. VHDL supports components and packages that allow generic components to be defined and reused within other designs. There are three types of abstraction levels available in VHDL; Behavioural level, Register Transfer Level (RTL), and Gate level.

Behavioural level modelling is the highest level of abstraction. Algorithms can be modelled without specifying clock information such as frequency or period. This allows for quick development of algorithms, but the implementation can be inefficient in terms of area and performance [37].

RTL modelling allows clock information to be specified when modelling algorithms. All operations are modelled in terms of clock cycles and defined in terms of registers and the flow of signals between them.

Gate level modelling is the lowest level of abstraction. This type of modelling consists of gates and registers instantiated from a specific technology library. The technology library specifies timing and gate delay properties for components instantiated using gate level modelling [38].

2.6 DK Design Suite

DK Design Suite offers an integrated development environment (IDE) for C-based designs. DK allows design entry, simulation and architecture exploration through synthesis. DK uses C-based languages such as Handel-C for high-level system design that is directly optimized for FPGA logic. For simulations, DK supports Transaction Level Modeling (TLM), fast cycle accurate simulations and co-simulations with hardware and software models. DK can import designs from EDIF, HDL, and Matlab-Simulink.

The features of DK Design Suite allows developers to explore algorithms and design alternatives in the high level specification code using Handel-C, rapidly partition hardware and software for optimized performance, and drive test benches from system specifications, reduces verification time and design cost with an easy to use design environment [39].

2.7 Altera Quartus II

Altera offers design entry and synthesis tools for designs targeting Altera FPGA devices. The Quartus II design entry tool enables developers to compile their designs, perform timing analysis, and program target FPGA devices. The tool provides short compile times with time preservation techniques, advanced place and route algorithms, and incremental compile stages for large designs. Behavioural VHDL modeling is supported where design functionality can be described using process statements. Altera provides many FPGA devices with different densities and features. Altera's FPGA families provide low cost FPGA architectures while maintaining performance.

Quartus II also supports the use of Library of Parameterized Modules (LPM). LPM is an industry standard that enables efficient mapping of design components to architectures such as PLDs, FPGAs, and standard cells. LPM modules are architecture independent such that the target architecture needs not be specified during design entry and verification. LPM modules are efficiently designed since IC vendors are responsible for mapping LPM functions. LPM modules are also tool independent and designs with LPM modules can migrate between synthesis and verifications tools. Altera offers many LPM design modules that are specifically designed for Altera FPGA device families. LPM modules provided include gate functions, arithmetic functions, storage function and few custom functions such as FIFOs and RAMs [40] [41].

2.8 Altera Cyclone III FPGA Devices

Altera offers low power, high functionality, and low cost FPGA devices in the Cyclone III FPGA family. The architecture for the Cyclone III devices allows for high volume, low power, and cost sensitive applications. The Cyclone III family offers FPGA devices with densities between 5K and 200K logic elements and 0.5 Mbits to 8 Mbits of memory. Lower power consumption is achieved by using TSMC low power process technology and power aware design flow. Cyclone III FPGA devices also offer on-chip embedded multipliers, PLLs, and global clock networks. These devices also support a large density of I/O banks with single-ended and differential I/O standards [42].

Chapter 3 - Related Work

There has been much research done in the area of FL algorithm hardware architecture. Hardware implementations of FL are presented as controllers or dedicated processors. These works contain various approaches for implementing FL algorithms in hardware.

3.1 Standalone Controllers

The authors in [9] describe a dedicated hardware architecture for executing FL algorithms. The datapath has six pipeline stages and consists of basic arithmetic units and lookup tables. The datapath uses a simplified defuzzification method known as Truth Value Flow Inference (TVFI). The method assumes that all output membership function shapes are symmetric, thus only requiring the centre of the membership function shape to be known. The datapath relies on a division lookup table that is not on-chip. The table provides division values when performing defuzzification. The datapath can also be configured by the host processor. The lookup tables for the membership functions and fuzzy inferences rules can be re-configured with new data from the host processor.

Although the hardware architecture is flexible such that it can be configured via host processor, but there are a few limitations. The first limitation is the resolution of the input and output and the number of fuzzy inference rules supported. The resolution of inputs is small, not allowing the input and output to be as sensitive if the resolution was 32 bits. Moreover, the number of fuzzy inference rules supported is very small. This limits the control system behaviour. The method of using TVFI for defuzzification simplifies the conversion from fuzzy set back to crisp value; however the assumption that the output membership function shapes are symmetric restricts the complexity and robustness of the hardware implementation of the FL algorithm.

The authors in [10] propose a hardware FL controller that is independent and requires no host processor. The datapath accepts two or three inputs and produces one output that drives the system. The datapath consists of ROMs containing data which represent the

input and output membership functions and fuzzy inference rules. Instead of using a divider, a shift register is used to perform the division in the defuzzification stage. The controller was implemented as a two input and one output and three input and one output controller and significant speed up was achieved.

The controller supports algorithms with two or three inputs; however, the controller is lacking configurability of the datapath and integration. The controller utilizes ROMs and lookup tables for storing membership function shape and fuzzy inference rules, but the configuration can be a challenge. There is no automated flow or defined format on how to decode the membership function shapes and fuzzy inference rules in ROMs using lookup tables. The developer must decode the shapes and rules manually. Furthermore, the FL controller is an independent datapath that cannot be integrated with a host processor or other peripherals easily. This limits what applications the controller can be used for.

A high speed Digital FL controller is proposed by [11] where a high internal clock frequency is achieved by using Odd-Even rule evaluation methodology. Typical FL controllers process one active rule per clock cycle. By using the Odd-Even method, two active rules can be evaluated per clock cycle, doubling the processing time. The performance significantly increases the input data processing rate of the FL controller. The controller utilizes a Sugeno-type inference engine and does not require a host processor. The datapath of the controller is designed using VHDL that is fully parameterized that allows the generation of different FL algorithms.

The Digital FL controller has the highest internal clock frequency of 200MHz making it an attractive architecture; however there are a few drawbacks. The Digital FL controller datapath contains multiple clock domains, deep pipelines, and complex custom logic blocks. Although the design is parameterized, the configuration of the datapath is cumbersome. The datapath only supports a small set of fuzzy inference rules and cannot be integrated with a host processor.

A VHDL based FL controller is presented in [12] targeted for Xilinx FPGAs. The VHDL based FL controller is a three stage hardware datapath that provides flexibility in number of inputs, bit width, and number of fuzzy sets. The fuzzy inference rules can also be configured

by updating the contents of the rule memory. The FL controller only supports triangular membership function shapes with a limited number of fuzzy inference rules. The FL controller utilizes local on-chip memories to store data for the fuzzification and inference rules stage. For defuzzification, the controller implements the centroid method using hardware multipliers and a pipelined hardware divider.

The VHDL based FL controller is flexible in terms of configuring the controller for specific algorithms. In the fuzzification stage, adders, subtractors, and a multiplier are needed to map the input values to fuzzy set values. This approach unnecessarily increases the latency of the fuzzification stage. Secondly, the datapath can support up to eight bits per input. This significantly reduces the resolution of the input values. Inside the FL controller, the control unit manages the flow of data inside the datapath. There are few redundant control signals for the datapath unnecessary for algorithms.

3.2 Dedicated Processors

Along with many dedicated FPGA based hardware architectures datapaths for executing FL algorithms, there have been many proposals for a dedicated FL processor. This FL processor will have instructions to execute fuzzification, fuzzy inference rule comparison, and defuzzification operations. The FL processor executes the complete FL algorithm over multiple instructions and multiple cycles. Many have stated that a FL processor would yield higher throughput when compared against a GPP executing a FL algorithm.

The authors in [13] proposed a VLSI fuzzy processor that exploits parallelism to achieve maximum performance and throughput. Along with exploiting parallelism, the way fuzzy inference rules are managed is also simplified. Consider a situation where a single antecedent is repeated in several fuzzy inference rules. Instead of repeating the antecedent for several fuzzy inference rules, a single antecedent with a lookup table is required. The lookup table would contain the truth values for the antecedent. This exploitation would decrease the number of fuzzy inference rules required and increase performance. The last optimization is a preliminary phase to determine what fuzzy inference rules needed to be active. This significantly reduces the time required to compare the fuzzy sets against the fuzzy inference rule database.

Although the VLSI FL processor applied parallelism to improve performance, the FL processor is still very restrictive. The FL processor is implemented as an ASIC where many attributes of the FL processor are fixed after synthesis. The membership function shapes and fuzzy inference rules are fixed and cannot be modified. Furthermore, the FL processor is not pipelined. The FL processor is designed as a multi-cycle datapath. The latency of the datapath is 24 clock cycles. Thus the datapath will only process one instruction every 24 clock cycles. This has a significant impact on the throughput of the processor.

Another VLSI FL processor was proposed in [14]. The authors present a fast parallel-pipeline architecture using RAM blocks for storing membership function shapes and fuzzy inference rules. The processor datapath exploits both pipelining and parallelism and uses RAM blocks as lookup tables reduced the overall processor latency. The inference engine uses the methodology of performing fuzzy implications with active rules. This reduces the time required to evaluate the fuzzy sets against the fuzzy inference rule database. The defuzzification block is heavily pipelined and parallelized to reduce the latency. To increase the throughput of the processor, the processor frequency, and decrease the latency, the layout of the processor was crucial. Significant time and effort was required to lay out all the blocks such that the processor met performance and area requirements.

This FL processor boasts the highest throughput, managing up to 50 million fuzzy inferences per second. However significant design effort and time was required to achieve that. First and foremost, the layout of the FL processor was critical. The blocks of the datapath were placed in a specific orientation and in order to achieve performance and area criteria. This task is not necessarily easy or trivial. The architecture of the datapath exploited all the necessary techniques to reduce latency, but the exploitations resulted in a datapath that cannot be easily configured. The datapath only supports trapezoidal membership function shapes and union operators for the inference rule engine.

A FPGA based FL processor was proposed by [43] for medical diagnosis. The architecture provides high performance low power processor based on single chip architecture. The FL processor supports up to 256 fuzzy inference rules with 16 inputs 16 bits wide. FL processor attempts to optimize the computational process of performing fuzzy inference.

This is done by implementing several fuzzy inference rules in parallel. The FL processor supports complex membership function shapes such as Gaussian curves, where the translation of mapping the input to a fuzzy set is performed in hardware using arithmetic units. The FL processor was designed in VHDL and targeted for the Altera Cyclone FPGA family and was able to achieve a maximum frequency of 40MHz.

The FPGA based FL processor presents many new ideas in implementing FL controllers in hardware, but still lacks in areas of configurability and integration in an embedded control system. The FL processor only supports FL based operations and cannot be used to perform other operations such as multiplication or branch. Since the FL processor is targeting Altera Cycle FPGA family, the FL processor design did not take advantage of on-chip resources for arithmetic and memory operations.

3.3 Hardware Comparison Parameters

To evaluate the proposed architecture in this thesis, comparison criteria must be defined. Since there is no standard comparison criterion for hardware implementation of FL controllers, most common and important criterion will be selected as a comparison point. The comparison will focus on core FL principles as well as hardware implementation.

3.3.1 Inputs and Outputs

The inputs and outputs of a FL controller is defined as the number of inputs and outputs of the FL algorithm for a control system. Typical FL algorithms configurations include three inputs / one output, four inputs / two outputs, and the most common being two inputs and one output.

3.3.2 Input and Output Resolution

The input and output resolution describes the bit width of the inputs and outputs. This parameter defines how coarse or fine the supported inputs and outputs are.

3.3.3 Membership Function Shapes

This parameter defines how many membership function shapes are supported per input and output. The shapes define how the inputs and outputs are translated into fuzzy sets. The less membership function shapes available will provide a coarse fuzzy set whereas more membership function shapes will provide a much finer fuzzy set.

3.3.4 Membership Function Shape Types

This parameter defines the types of membership function shapes. From Chapter 2, there are many membership function shapes such as triangular, trapezoidal, S-shaped, Gaussian, Bell-shaped, etc.

3.3.5 Fuzzy Operators

This parameter defines the types of operators applied when translating input values to their respective fuzzy sets. These operators help define the fuzzy sets used for when evaluating against fuzzy inference rules.

3.3.6 Inference Rules Supported

This parameter defines how many fuzzy inference rules are supported by the FL controller. The fuzzy inference rules describe the behaviour of the FL controller. The more fuzzy inference rules defined leads to accurate and precise results generated by the FL controller.

3.3.7 Implication Operator

This parameter defines the type of implication operator supported when evaluating fuzzy sets against fuzzy inference rules. Typically intersection and union implication operators are used in describing fuzzy inference rules.

3.3.8 Defuzzification Method

This parameter defines the type of defuzzification method is used by the FL controller to translate the output fuzzy back to a crisp output value. The type of defuzzification method defines how precise the output value is from the FL controller. The most popular defuzzification method used by FL algorithms is the centroid method described in Chapter 2.

3.3.9 Fuzzy Inference System

This parameter defines the type of fuzzy inference system used by the FL controller. There are only two types of fuzzy inference systems; Mamdani-type and Sugeno-type. Mamdani-type fuzzy inference system uses different defuzzification methods to translate the output fuzzy set back to a crisp value whereas Sugeno-type fuzzy inference system uses weighted average to compute the crisp output.

Using a Sugeno-type fuzzy inference system requires no output membership function, no defuzzification method to translate output fuzzy set to a crisp value, but there is a loss of interpretability.

3.3.10 Configurability

This parameter defines if the FL controller or processor can be configured for different types of FL algorithms. Configuration should be relatively automated where it can be derived from a model or a configuration standard. The configuration should occur before compilation and synthesis of the hardware datapath.

3.3.11 Processor Support

This parameter defines if the FL controller can be integrated with a processor in an embedded control system. The integration involves defining an interface to the processor where FL controller operations can be coupled with the processor's instruction set.

3.3.12 Modeling

This parameter defines if a model of the FL algorithm can be integrated with the design flow of the FL controller or processor. For typical designs, extensive modeling is done to ensure proper functionality and that the design meets requirements. However, the difficulty lies in translating the model to hardware.

3.3.13 Target Platform

This parameter defines which target platform the FL controller or processor designed for. Types of target platforms include ASIC technologies or FPGA devices. This parameter helps determine the maximum operating frequency parameters and the throughput of the FL controllers and processors.

3.3.14 Maximum Operating Frequency

This parameter defines the maximum operating frequency of the FL controller or processor after synthesis.

3.3.15 Throughput

This parameter defines the maximum throughput of the FL controller or processor. The throughput is measured in millions of fuzzy inferences per second. The throughput is dependent on the maximum operating frequency and the latency of the FL controller or processor.

Table 1: Standalone Controllers and Dedicated Processors Comparisons

Features	Hung [9]	Vasantha Rani et al. [10]	Deliparaschos et al. [11]	Singh et al. [12]	Ascia et al. [13]	Gabrielli et al. [14]	Chowdhury et al. [43]
Inputs	2	2 or 3	2	2	8	4	16
Outputs	1	1	1	1	1	1	
Input Resolution	4 bits	4 bits	8 bits	8 bits	8 bits	7 bits	16 bits
Output Resolution	8 bits	8 bits	12 bits	8 bits	8 bits	7 bits	
Membership Function Shapes	3	7	7	7	7	128	64
Membership Function Shape Types	Triangular	Trapezoid / Triangular	Trapezoid / Triangular	Triangular	N/A	Trapezoid	Gaussian
Fuzzy Operators	Intersection	Intersection	Intersection	Intersection	N/A	N/A	Intersection
Inference Rules Supported	9	49 - 343	49	36	256	2401	256
Implication Operator	Intersection	Intersection	Intersection	Union	Intersection	Intersection	Intersection
Defuzzification Method	TVFI	Centroid	Centroid	Centroid	Yager	Centroid	Centroid
Fuzzy Inference System	Mamdani	Mamdani	Sugeno	Mamdani	Mamdani	Sugeno	Mamdani
Configurability	Yes	No	No	No	No	No	No
Processor Support	Yes	No	No	No	No	No	No
Modeling	No	No	No	No	No	No	No
Target Platform	Xilinx FPGA	Xilinx Spartan FPGA	Xilinx Spartan 3 FPGA	Xilinx Virtex FPGA	ASIC HCMOS 0.5um	ASIC CMOS 0.7um	Altera Cyclone FPGA
Maximum Operating Frequency	10MHz	43.75MHz	200MHz	21.86MHz	60MHz	50MHz	40MHz
Throughput (million/second)	3.30	1.40	7 69	1.45	2.50	50.00	5.00

The reviewed architectures will be compared against the architecture presented in this thesis as described in Chapter 7.

Chapter 4 - Proof of Concept

The design and implementation of a proof of concept hardware datapath for executing FL algorithms is discussed in this chapter. To understand the benefits of implementing FL algorithms in hardware, a datapath is implemented on a FPGA to evaluate the performance and area requirement. Determining the performance and area requirement will help us understand the trade offs of executing FL in hardware rather than software and recommendations that can be proposed to further improve the performance and area utilization of the datapath.

4.1 Case Study: Inverted Pendulum Problem

To design a proof of concept hardware FL controller and understand the advantages and disadvantages of implementing FL algorithms in hardware, the classic inverted pendulum problem is chosen. The FL controller for the inverted pendulum problem consists of two inputs; pole angle and pole angular velocity and one output force. There are 25 fuzzy inference rules that describe the behaviour of the inverted pendulum control system [44].

The input and output membership functions are described using triangular and trapezoidal shapes as illustrated in Figure 16, Figure 17, and Figure 18. The input membership functions are described using the fuzzy sets: NB (Negative Big), NM (Negative Medium), Z (Zero), PM (Positive Medium) and PB (Positive Big). The output membership function is described using the fuzzy sets: NB (Negative Big), NM (Negative Medium), NS (Negative Small), Z (Zero), PS (Positive Small), PM (Positive Medium) and PB (Positive Big).

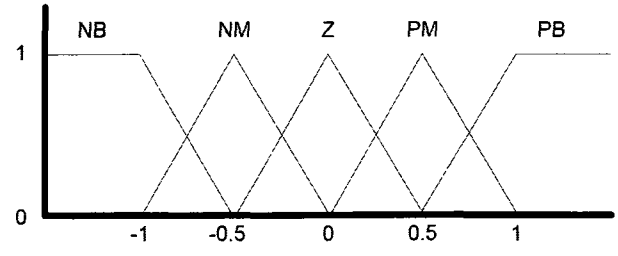

Figure 16: Pole Angle Input Membership Function

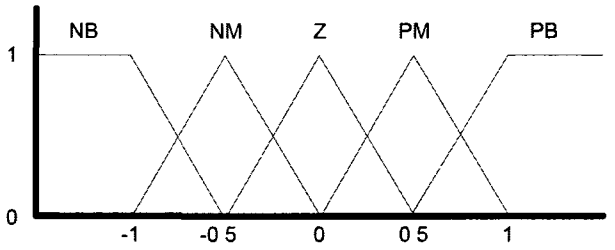

Figure 17: Pole Angular Velocity Input Membership Function

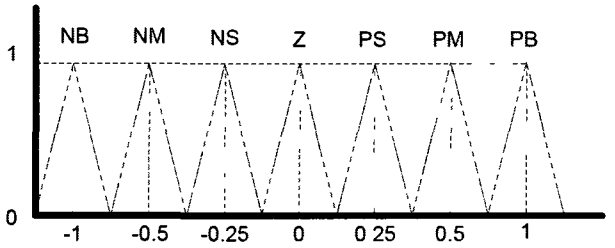

Figure 18: Force Output Membership Function

Table 2: Inverted Pendulum Fuzzy Inference Rules

Pole Angle	Pole Angular Velocity				
	NB	**NM**	**Z**	**PM**	**PB**
NB	NM	NM	NS	Z	Z
NM	NM	NM	NS	Z	Z
Z	NS	NS	Z	PS	PS
PM	Z	Z	PS	PM	PM
PB	Z	Z	PS	PM	PM

4.2 FL Controller

The proof of concept hardware datapath will establish a baseline of implementing a FL controller in hardware. Using this baseline the advantages and disadvantages can be understood.

The hardware datapath for FL algorithms was initially conceived as a standalone FL controller. The FL controller would execute a single controller instruction where two input

values would be evaluated in the FL domain and one output value is produced. To minimize the development time of the FL controller, the DK Design Suite design tool is selected.

For the inverted pendulum problem, few design considerations are made. Firstly, the FL controller must have two inputs and one output, target Mamdani-type inference engine and use pipelining and parallel processing algorithmic transformations to achieve maximum performance with minimal area utilization.

The pipelined datapath consists of three stages; fuzzification, inference engine, and defuzzification.

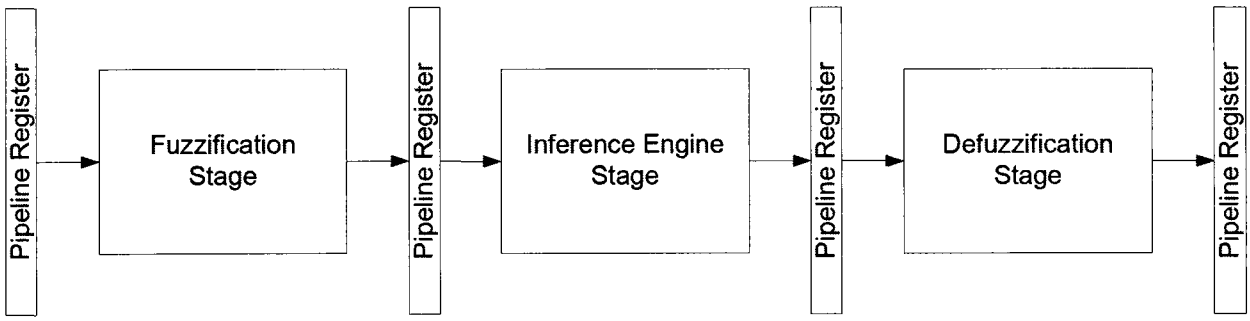

Figure 19: Proof of Concept FL Controller Datapath

4.2.1 Fuzzification

The fuzzification stage maps the input value a fuzzy set using input membership functions. To achieve minimal latency, lookup tables implemented as ROMs are used to store the input membership function shapes. Each input of the FL algorithm is mapped to a memory mapper component where the input is decoded into an address that is used to lookup its fuzzy set value stored in the membership function memory component.

In Figure 20, the internal signals inside the fuzzification stage are illustrated. Signals ***input1*** and ***input2*** represent the input variables to the FL algorithm. These inputs are then translated to a memory address using the ***memory_mapper1*** and ***memory_mapper2*** components. The ***address1*** and ***address2*** signals are used to lookup the input fuzzy sets from the ***membership_function1*** and ***membership_function2*** memories. From the membership memories, ***in_fuzzyset1*** and ***in_fuzzyset2*** represent the input fuzzy sets.

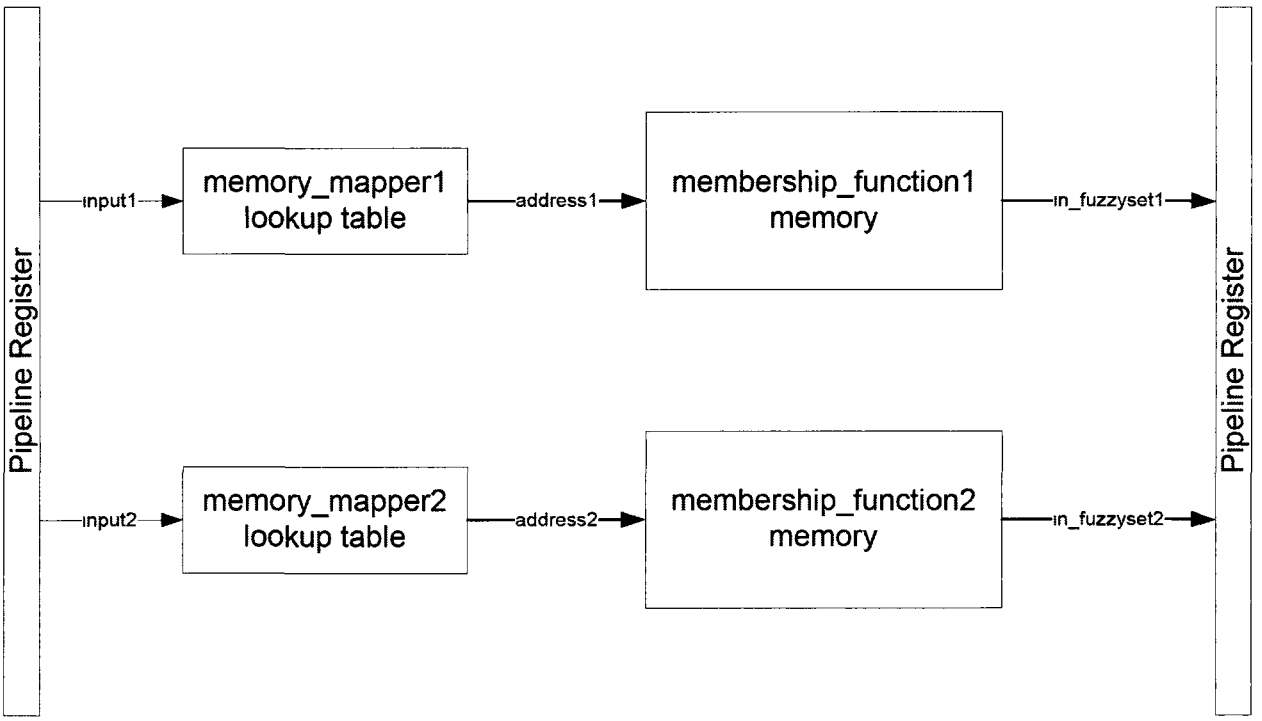

Figure 20: Fuzzification Stage

The size of the memory mapper component is dependent on the resolution of the input value. If the resolution is very small, the size of the lookup table will be large to accommodate all the possible input values. Therefore, to minimize the size of the memory

mapper lookup table; a resolution of 0.25 was selected. The input values will be rounded to the nearest resolution of 0.25.

Using the case study as an example, if the pole angle input is [-0.75] and the pole angular velocity input is [0.25], then the pole angle fuzzy set is [NB 0.5, NM 0.5] and the pole angular velocity fuzzy set is [Z 0.5, PM 0.5]. The fuzzy sets are determined by mapping the pole angle input to the pole angle membership function and the pole angle to the pole angle membership function illustrated in Figure 16 and Figure 17.

4.2.2 Inference Engine

The inference engine stage evaluates the fuzzy set values determined from the fuzzification stage against fuzzy inference rules defined by the FL algorithm. For a FL algorithm with two inputs, two different fuzzy value sets are generated. The fuzzy value sets need to be combined and then evaluated against the fuzzy inference rules. The ***combination_generator*** combines the fuzzy set values before the rule evaluation occurs. Each combination is evaluated against the ***inference_rules*** lookup tables in parallel. The fuzzy inference rules are stored in lookup tables to achieve minimum latency.

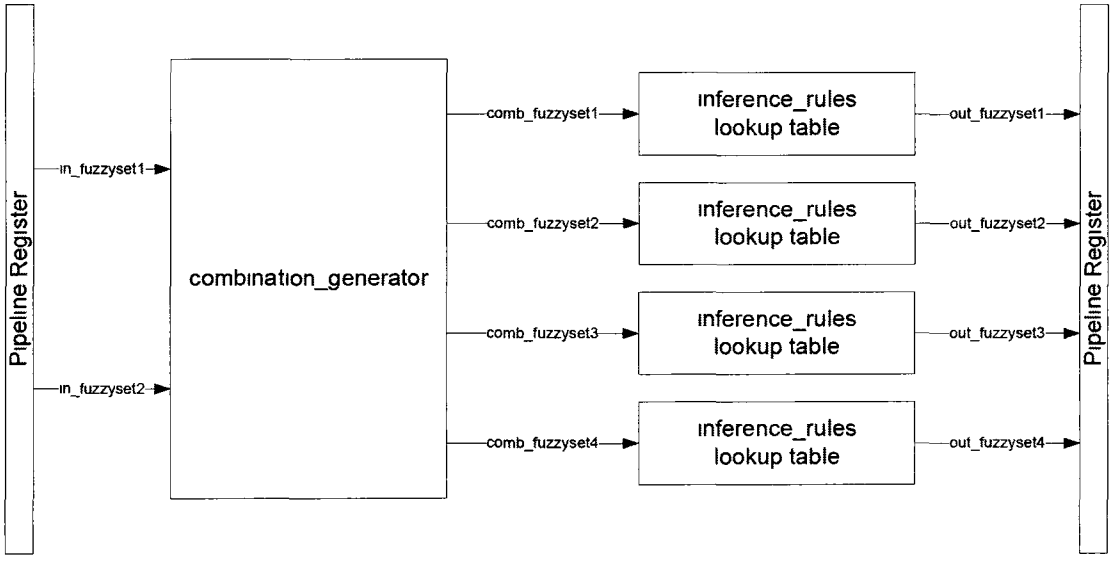

Figure 21: Inference Engine Stage

In the inference engine stage, the input fuzzy sets determined by the fuzzification stage are passed through the ***combination_generator*** component as seen in Figure 21. This

component merges the two input fuzzy sets and derives four fuzzy sets; ***comb_fuzzyset1***, ***comb_fuzzyset2***, ***comb_fuzzyset3***, and ***comb_fuzzyset4***. All four combination fuzzy sets are evaluated against fuzzy inference rules in parallel using four instantiations of the ***inference_rules*** lookup table. Evaluating each fuzzy set against the fuzzy inference rules determines an output fuzzy set ***out_fuzzyset1***, ***out_fuzzyset2***, ***out_fuzzyset3***, ***out_fuzzyset4***.

Continuing with the case study, the pole angle and pole angular velocity input fuzzy sets are merged to evaluate against the fuzzy inference rules. Combining the pole angle fuzzy set [NB 0.5, NM 0.5] and the pole angular velocity [Z 0.5, PM 0.5] generates four fuzzy sets [NB 0.5, Z 0.5], [NB 0.5, PM 0.5], [NM 0.5, Z 0.5], and [NM 0.5, PM 0.5].

Using the fuzzy inference rules shown in Table 2, the four merged fuzzy sets derive four output fuzzy sets [NS 0.5], [Z 0.5], [NS 0.5], and [Z 0.5]. It is important to note that during rule evaluation, the minimization of fuzzy sets values is also performed.

4.2.3 Defuzzification

The defuzzification stage translates the output fuzzy set from the inference engine to a crisp output value. The defuzzification process depends on the type of defuzzification technique used. For this datapath, the centroid method is applied as the defuzzification technique.

As seen in Figure 22, the internal components of the defuzzification stage require many arithmetic units, lookup tables, and buffer registers. The design of the stage is a mixture of both parallel and serial circuits. The output fuzzy sets from the inference engine are first sign extended and mapped to the output membership function. The ***weight_mapper*** lookup table maps the output fuzzy sets to the output membership function. In parallel, the output fuzzy set values are sign extended before the defuzzification process by the ***sign_extender*** component. Buffer registers help synchronize the timing between the ***weight_mapper*** and ***sign_extender*** components.

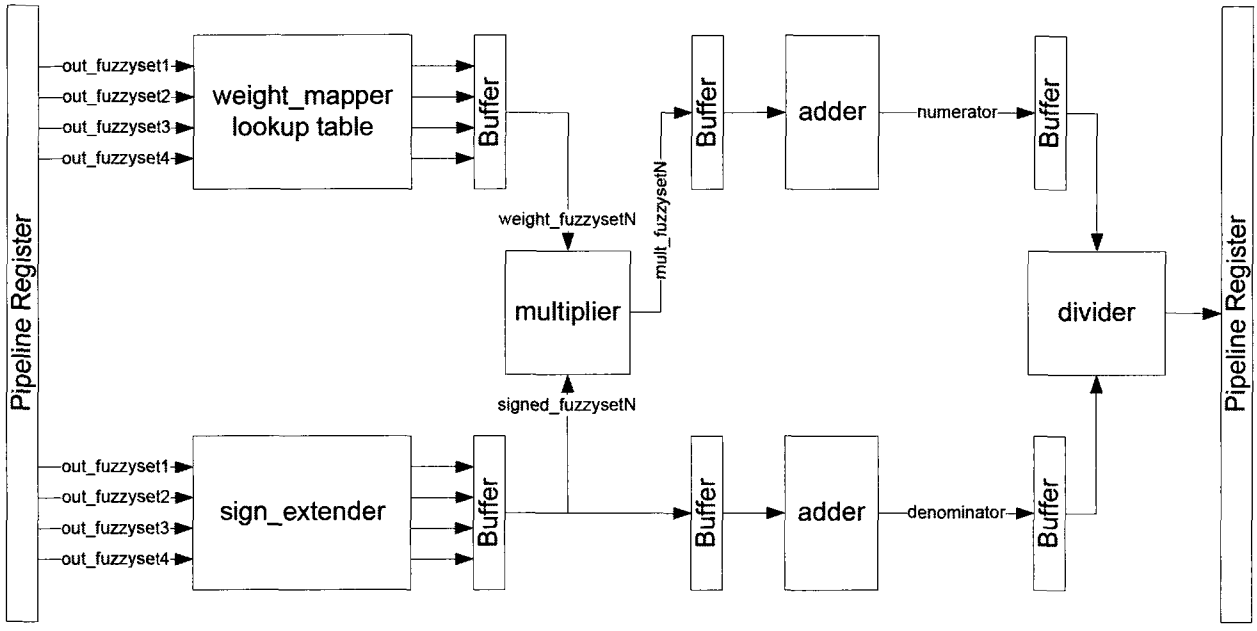

Figure 22: Defuzzification Stage

Based on the centroid defuzzification technique, the output fuzzy sets weight mapped to the output membership function needs to be multiplied with the respective output fuzzy set values. The multiplication between the output fuzzy sets weight with the output fuzzy set values is performed serially. Since this serial circuit requires multiple clock cycles to complete, the multiplier is mapped to an on-chip hardware multiplier to minimize the latency. Each ***weight_fuzzysetN*** is multiplied with the ***signed_fuzzysetN*** using the ***multiplier*** component and then the result ***mult_fuzzysetN*** is stored in a buffer register.

Using the centroid defuzzification technique, division occurs between the summation of products between the fuzzy set output values and its corresponding weight based on the output membership function andk the summation of all the output fuzzy set values. To synchronize timing between the numerator and denominator, buffer registers are used. The summations are performed sequentially and the ***adder*** components are mapped to dedicated on-chip resources to reduce the latency of this operation.

The last step in the defuzzification stage is to perform the division between the numerator and denominator. The ***numerator*** and ***denominator*** values are buffered to synchronize

timing. Then the division is performed using the ***divider*** component. The result of the divider is then passed to the last pipeline register.

To finish the case study example, the output fuzzy sets [NS 0.5], [Z 0.5], [NS 0.5], and [Z 0.5] are mapped to the output membership function to determine the weight strength. For the [NS] fuzzy set, the weight strength is [-0.25] when mapped to the output membership function shown in Figure 18. The weight strength is determined where the fuzzy set is at its maximum. Continuing with this reasoning and walking through the rest of the defuzzification stage datapath, the output force can be determined as follows:

$$Force = \frac{(0.5)\cdot(-0.25)+(0.5)\cdot(0)+(0.5)\cdot(-0.25)+(0.5)\cdot(0)}{(0.5)+(0.5)+(0.5)+(0.5)}$$

$$Force = -0.125$$

4.3 Implementation

To evaluate the FL controller, a FPGA development board was utilized to obtain the performance, area, and throughput metrics. With DK Design Suite, Celoxica offers FPGA development boards that were easy to target and program. The FPGA development targeted for the evaluation had a Xilinx Spartan 3 3SI1500L-4 FPGA with on-chip resources. The development board contains many peripherals such as USB, external VGA, etc.

4.4 Evaluation

To evaluate the performance of the FL controller, the critical path is determined to obtain the maximum frequency the FL controller on the FPGA. The DK Design Suite analysis algorithms determine the critical path and recommend suggestions on how to improve the performance of the FL controller. DK Design Suite presents a summary of the maximum logic and routing delay from Flip flop to Flip flop and from Flip flop to Pin.

Table 3: Combinatorial path analysis

Iteration	Maximum logic and routing delay from Flip flop to Flip flop	Maximum logic and routing delay from Flip flop to Pin
1	68.49ns	2.12ns
2	62.07ns	2.12ns

The first design iteration compiled and synthesized the FL controller with no optimizations. This gives a baseline of the maximum performance of the FL controller. The critical path in the datapath is determined to be between Flip flops. The logic and routing delay is 68.49ns that translates to a maximum operating frequency of 14.6MHz. This is a cause of concern since the goal here is to integrate the FL controller with a processor. Next, design optimizations are considered to help reduce the critical path in the datapath.

The second design iteration applied optimization techniques provided by DK Design Suite tool in hopes of reducing the critical path. The first optimization forces use of on-chip resources. Using the dedicated on-chip resources will help improved performance by reducing the latency and area of the design. The target FPGA contains embedded multipliers inside the FPGA that are used to improve the performance and area utilization of the FL controller. The second optimization technique applied is pipelining. Additional pipeline stages are introduced inside each of the stages to reduce the critical path. After applying these optimization techniques, the FL controller is then re-compiled and synthesized. The logic and routing delay from Flip flop to Flip flop reduces from 68.49ns to 62.07ns, increasing the maximum operating frequency from 14.6MHz to 16.1MHz. Although the frequency has increased, 16.1MHz is still not enough to be an attractive solution of implementing FL algorithms in hardware. Each stage in the FL controller is now evaluated to determine where additional design optimizations can be applied to further reduce the critical path in the design.

Using DK Design Suite, optimization techniques were applied to the datapath without any improvement. High-level, rewriting, common subexpression, and conditional rewriting optimizations were applied to further optimize the datapath. Another technique to improve

the latency is attempted where the divider was implemented as a lookup table. This reduces the latency between flip flops from 62.07ns to 19.88ns and improves the maximum operating frequency to 50.3MHz. The drawback of this technique is that a true divider is not implemented having a significant impact on precision. Although the maximum operating frequency is improved, this optimization technique cannot be applied to the FL controller, leaving the maximum operating frequency to be 16.1MHz.

4.5 Drawbacks

The evaluation of the FL controller highlights advantages of using a hardware datapath to execute FL algorithms, but there are also many drawbacks that need to be addressed in this proof of concept architecture. First and foremost, the divider implemented in the FL controller greatly increased the latency of the FL controller and reduced the maximum operating frequency. The FL controller consists of many serial circuits that require additional clock cycles to execute. The FL controller also lacks the ability to be configured for different types of FL algorithms and does not address the issue for processor integration. Again, there is no defined interface for the FL controller to any given processor.

Chapter 5 - Architecture

The proof of concept FL controller established a baseline of performance and area utilization, identified the strengths and weaknesses of the datapath, and highlighted the marginalities. The critical design marginality is the implementation of the divider inside the defuzzification stage of the FL controller. The divider increased the latency of the design and decreased the maximum operating frequency. The datapath consisted of many serial circuits that also increased the overall latency of the FL controller and did not support configurability or processor integration.

This chapter addresses the drawbacks determined from the proof of concept architecture and presents a configurable architecture that supports processor integration and configurability. The proof of concept FL controller is used as a template in the design of the configurable architecture.

The design is divided into two parts; datapath and control unit. The Fuzzy Logic Datapath (FLDP) outlines the RTL components used in the design. The Fuzzy Logic Control Unit (FLCU) outlines the state machines and control structures used in managing control signals through the FLDP and the interface between the FLDP and the Leon3 processor.

5.1 Fuzzy Logic Datapath

The FLDP supports 32 bit inputs and outputs and contains three pipeline stages. Similar to the proof of concept architecture, the three pipeline stages are: fuzzification, inference engine, and defuzzifcation. The datapath is designed using behavioural VHDL modelling that allows the instantiation of process and generic statements. and for defining configuration parameters before compilation.

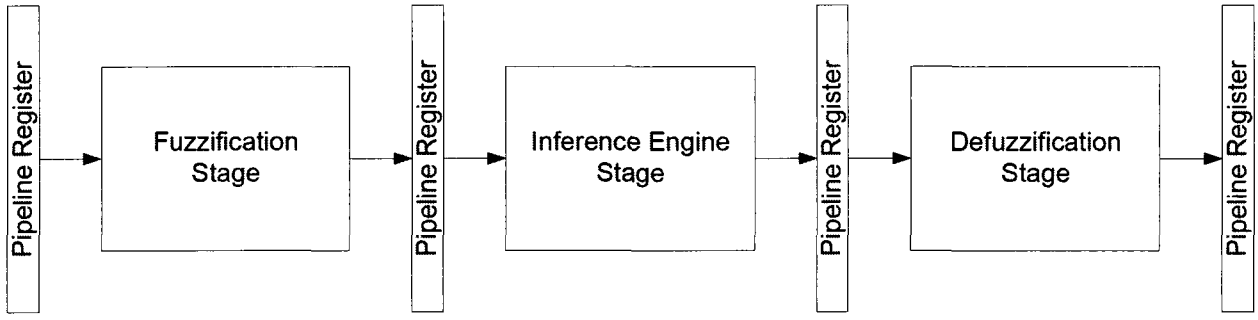

Figure 23: High-level Datapath of FLDP

The FLDP contains the same number of pipeline stages as the proof of concept datapath since the high level description of implementing FL algorithm in hardware remains unchanged. However, each stage is optimized to reduce latency and uses FPGA on-chip resources to minimize area utilization and increase performance.

5.1.1 Fuzzification

The first stage maps the input crisp values to its respective fuzzy set determined by the membership function shapes. Inside the ***Fuzzification*** component, lookup tables are used to translate the input values to fuzzy set values. The lookup tables are described using if-then-else statements inside a VHDL process statement. Before the input values are evaluated, sign extension is performed.

From Figure 24, signals ***Input0*** and ***Input1*** are input values from the pipeline and ***InFuzzySet0***, ***InFuzzySet1***, ***InFuzzySet2***, and ***InFuzzySet3*** describe the fuzzy set values for each input value that are routed to the next pipeline stage.

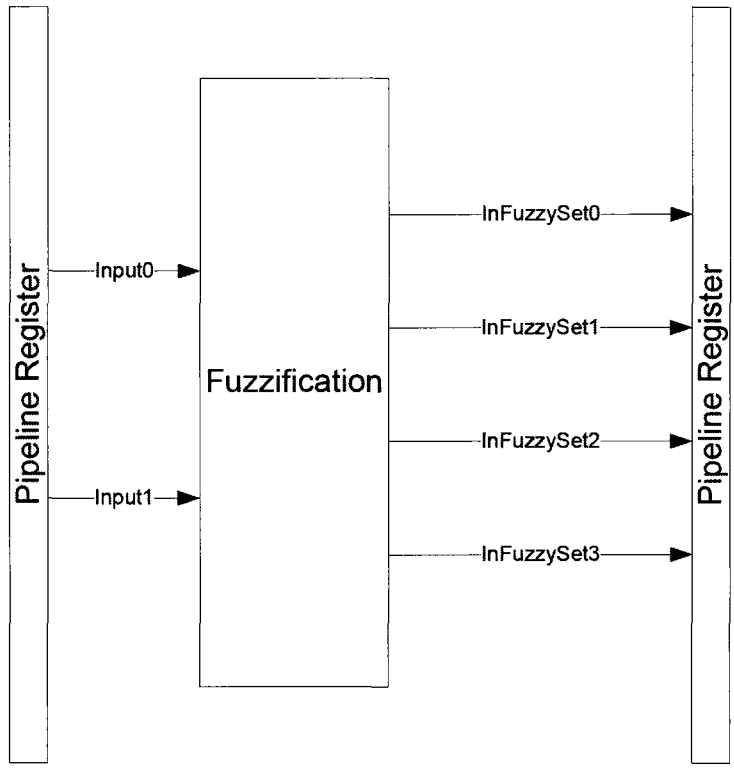

Figure 24: Fuzzification Stage

Comparing against the proof of concept datapath, the fuzzification stage is similar. Instead of using the memory mapper component to generate a lookup address for the membership ROM, the ***Fuzzification*** component directly uses the input values as an address and outputs the input fuzzy sets.

5.1.2 Inference Engine

The inference engine utilizes four ***Inference Rules*** components in parallel to evaluate the fuzzy sets against the fuzzy inference rules of the FL algorithm. All the components are identical and the fuzzy inference rules are stored in an if-then-else structure in a VHDL process statement. The evaluation and minimization of the fuzzy set against the fuzzy inference rules is performed in parallel to reduce latency.

Figure 25 illustrates the input and output signals of the Inference Engine stage. Signals ***InFuzzySet0***, ***InFuzzySet1***, ***InFuzzySet2***, and ***InFuzzySet3*** are the fuzzy sets to be evaluated against the fuzzy inference rules. Signals ***OutFuzzySet0***, ***OutFuzzySet1***, ***OutFuzzySet2***, and ***OutFuzzySet3*** are the output fuzzy sets after the rule inference.

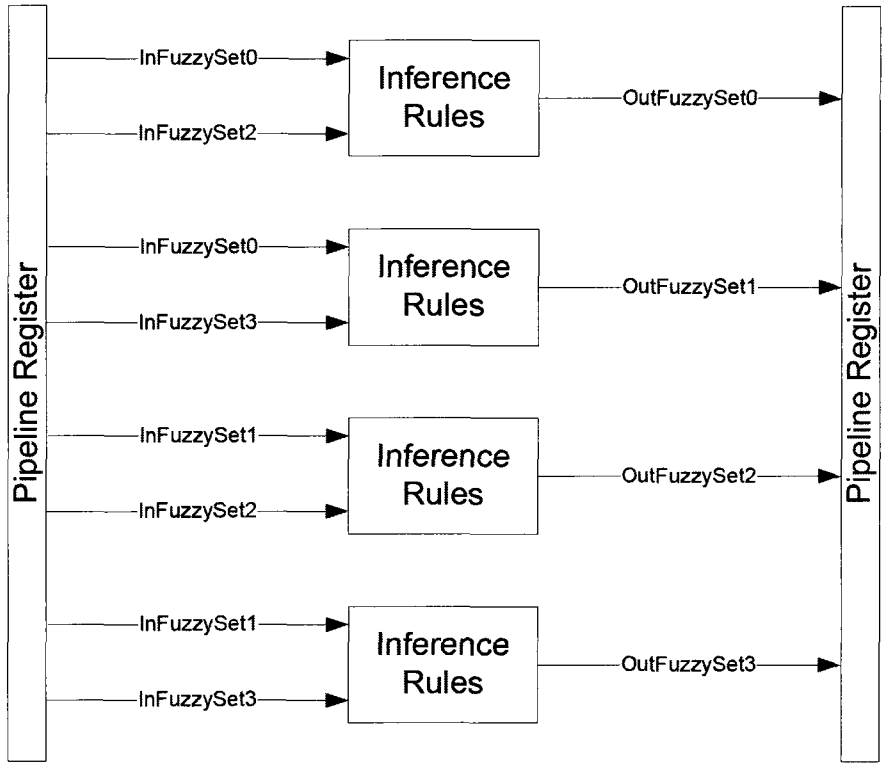

Figure 25: Inference Engine Stage

Comparing this stage to the proof of concept datapath, the inference engine stage is also similar. Instead of using the combination generator, the combined input fuzzy sets are directly mapped from the pipeline register. This reduces the number of components instantiated in this stage.

5.1.3 Defuzzification

The defuzzification stage is significantly different from the defuzzification stage in the proof of concept datapath. Most importantly, the serial operation of computing the numerator and denominator for the divider is now parallelized.

To harness the features of the Altera Cyclone III FPGA device family, LPM modules are used to reduce the area and improve the performance of this stage. Instead of using one multiplier component and perform the multiplication serially, four ***Multiplier*** components are instantiated to perform four multiplications in parallel. Using LPM modules to implementing the four ***Multiplier*** components offers the inputs to be registered internally. This allows the inputs to be latched for one clock cycle before executing the multiplication

operation and not requiring any buffer registers between the ***WeightMapper***, ***SignExtender,*** and the ***Multiplier*** components.

To synchronize the outputs of the ***Multiplier*** components an internal pipeline ***Numerator Pipeline*** register is used. The internal pipeline ***Denominator Pipeline*** register also latches in the outputs from the ***SignExtender*** component to line up with the ***Numerator Pipeline*** register as shown in Figure 26. The next step is to perform addition to determine the numerator and denominator needed for the centroid defuzzification method. Instead of performing summation of numerators and denominators serially, the task is broken up into parallel processing using six instantiations of the ***Adder*** component.

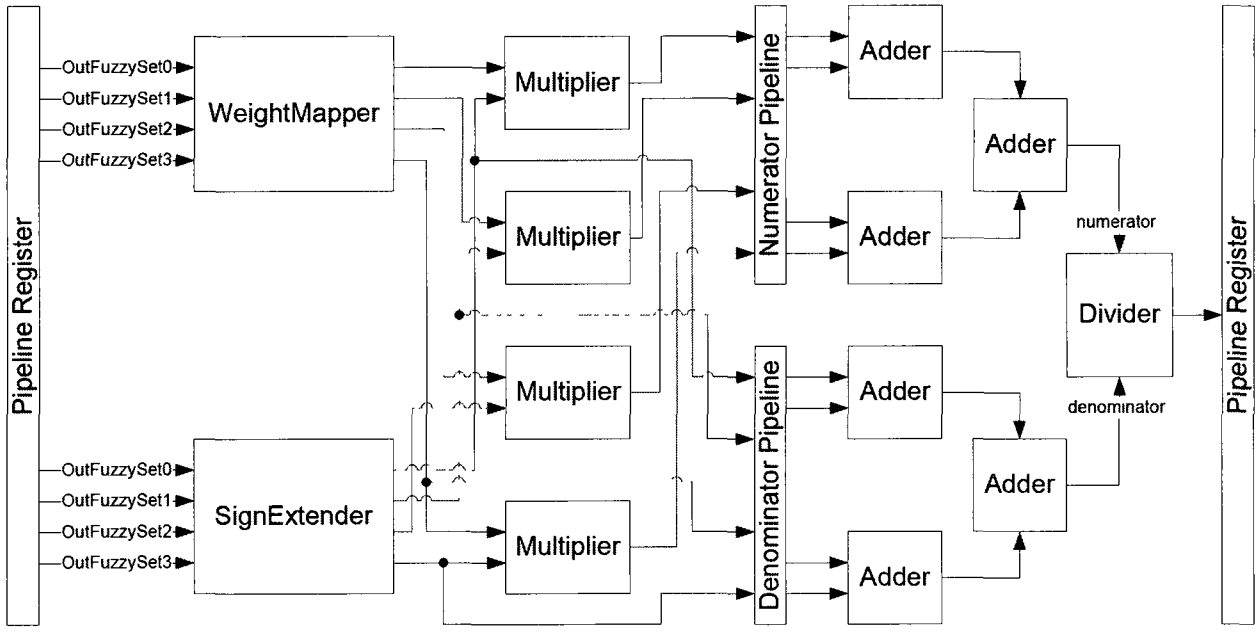

Figure 26: Defuzzification Stage

The operation determining the numerator and denominator uses an adder tree to perform the summation in parallel. For a two input and one output FL algorithm only six ***Adder*** components are necessary. Altera LPM modules are used to implement the ***Adder*** components and the inputs to the components are latched internally, not requiring buffers registers.

Finally, the last component is a ***Divider*** that is also implemented as an Altera LPM module. Using the LPM divider module gives the best performance by using FPGA on-chip

resources. The output of the ***Divider*** component is then latched in the last pipeline stage of the datapath.

5.1.4 FLDP LPM Components

The table below outlines all LPM components used in the FLDP datapath.

Table 4: FLDP LPM Components

Component	Configuration Parameters	
Parameterized Adder/Subtractor (lpm_add_sub)	LPM_DIRECTION:	ADD
	LPM_PIPELINE:	0
	LPM_REPRESENTATION:	SIGNED
	LPM_WIDTH:	32
Parameterized Multiplier (lpm_mult)	LPM_PIPELINE:	0
	LPM_REPRESENTATION:	SIGNED
	LPM_WIDTHA:	32
	LPM_WIDTHB:	32
	LPM_WIDTHP:	64
Parameterized Divider (lpm_divide)	LPM_WIDTHN:	32
	LPM_WIDTHD:	32
	LPM_NREPRESENTATION:	SIGNED
	LPM_DREPRESENTATION:	SIGNED
	LPM_PIPELINE:	2
Parameterized D FlipFlop (lpm_ff)	LPM_AVALUE:	NO
	LPM_FFTYPE:	DFF
	LPM_SVALUE:	NO
	LPM_WIDTH:	32
Parameterized Multiplexer (lpm_mux)	LPM_PIPELINE:	NO
	LPM_SIZE:	4
	LPM_WIDTH:	32

5.2 Fuzzy Logic Control Unit

The FLCU provides an interface to the Leon 3 Integer Unit (IU). FLCU decodes Leon3 coprocessor instructions, dispatches instructions to FLDP, and manages the FLCU register file. The FLCU can support many FLDP components in the FLCU Execution Stage. This allows the system to support many different FL algorithms using the same processor interface. Coprocessor instructions are executed in parallel with other Leon3 integer instructions; the Leon3 integer pipeline is only stalled in the case of operand conflicts detected by the IU.

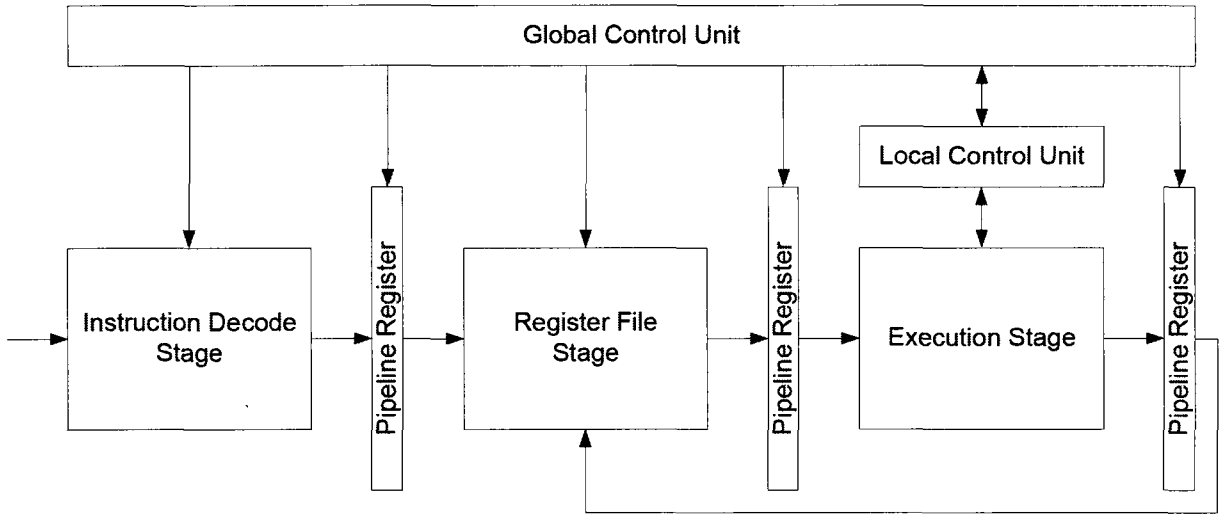

Figure 27: Fuzzy Logic Control Unit Datapath

The FLCU is a four stage pipelined datapath. The datapath consists of Instruction Decode (ID) stage, Register File (RF) stage, Execution (EX) stage, and Write Back (WB) stage. The FLCU control is managed by the Global Control Unit (GCU) and the FLDP control is managed by the Local Control Unit (LCU).

5.2.1 Instruction Decode

The ID stage decodes SPARC coprocessor instructions. There are three base instructions supported; load, store, and operation. The base instructions are 32-bit SPARC instructions shown in Figure 28 and Figure 29. For the load and store instruction format, bits 29 down to 25 are used for the destination register. The (rs1) and (simm13) fields are used to

determine the logical address. For the coprocessor operation instruction, the bits 31 and 30 are used to identify as an operation instruction and bits 24 down to 19 are used to identify as a coprocessor instruction. The (rs1) and (rs2) represents the indices of source registers and (rd) represents the index of the destination register. The (opc) specifies which custom instruction to execute. Since the (opc) field is 9 bits wide, the FLCU can support up to 512 different custom instructions. As shown in the format, a custom instruction can read at most two registers and write one register from/to the register file. The custom instruction is represented by the following assembly format: `cpop1 opc, rs1, rs2, rd` [25].

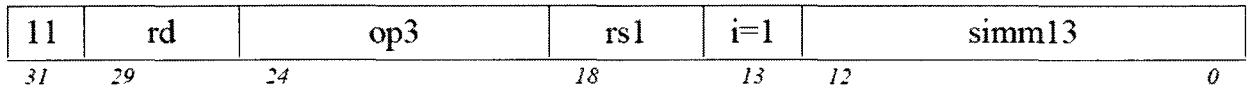

Figure 28: Coprocessor Load and Store Instruction Format [25]

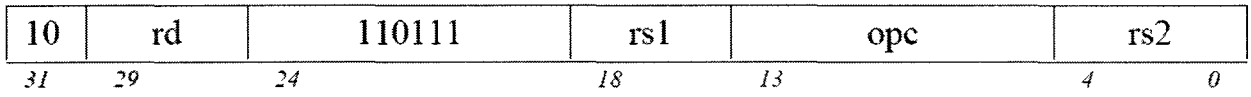

Figure 29: Coprocessor Operation Instruction Format [25]

Inside the instruction decode stage the ***DecodeUnit*** extracts field parameters from the 32 bit instruction as shown in Figure 30. Along with the ***Instruction***, the ***LoadData*** is also passed through the ***DecodeUnit***. The ***LoadData*** input is only used for load coprocessor instructions. Once decoded, the instruction parameters are passed to the register file stage through a pipeline register.

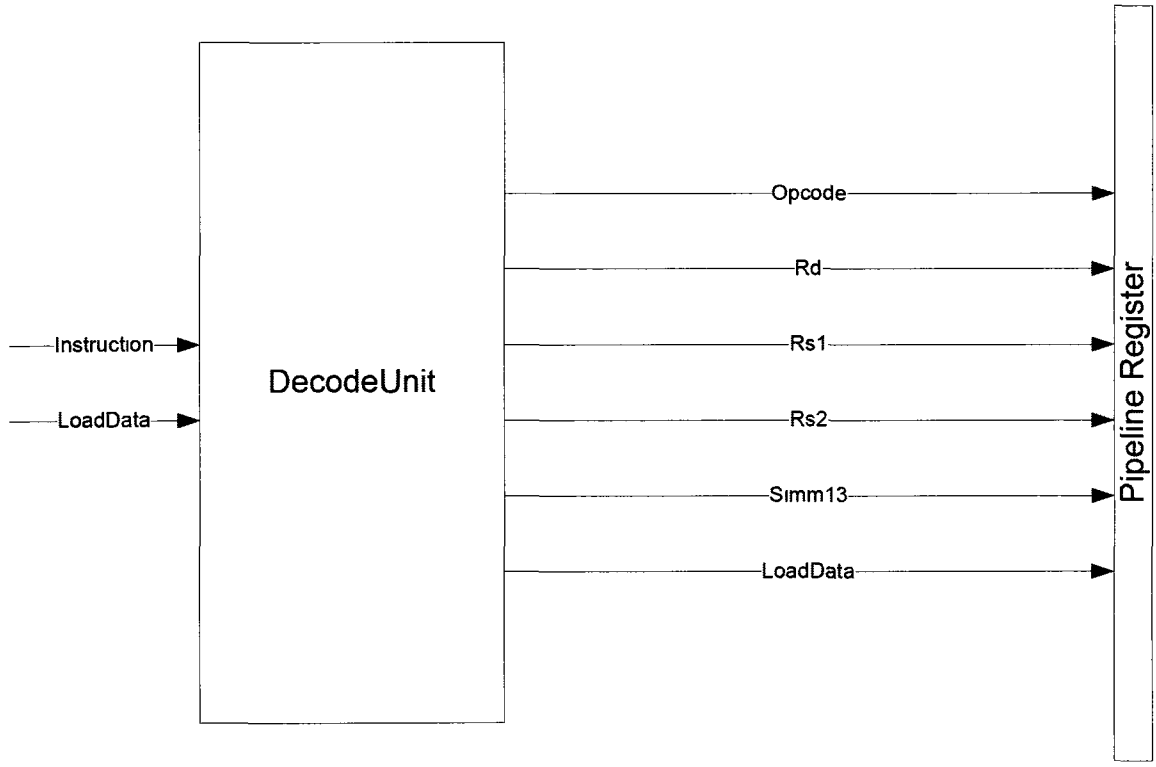

Figure 30: Instruction Decode Stage

5.2.2 Register File

The FLCU register file contains 32 32-bit registers and is accessed by coprocessor load, store, and coprocessor operation instructions. The register file is configurable and can support a minimum of two registers and a maximum of 32 registers. The register file is a dual port RAM with two read address ports and one write address port.

In Figure 31, components and internal signals are shown inside the register file stage. Inside the register file stage there are many control signals that alter what the datapath executes and there are many inputs from another stage in the FLCU. There are three ***Mux*** components that select which outputs from the pipeline register are mapped to inputs of the ***RegisterFile*** component. The ***Rs1*** output is mapped to the first read input of the ***RegisterFile*** component. The second read input of the ***RegisterFile*** component is from ***Mux2*** where either the ***Rd*** output or ***Rs2*** output is mapped to the second read input. This multiplexer is necessary because for operation instructions, ***Rs1*** and ***Rs2*** outputs are used. For load and store instructions, ***Rs1*** and ***Rd*** outputs are used. The ***regsel*** control signal for this multiplexer is driven from the ***GCU*** component.

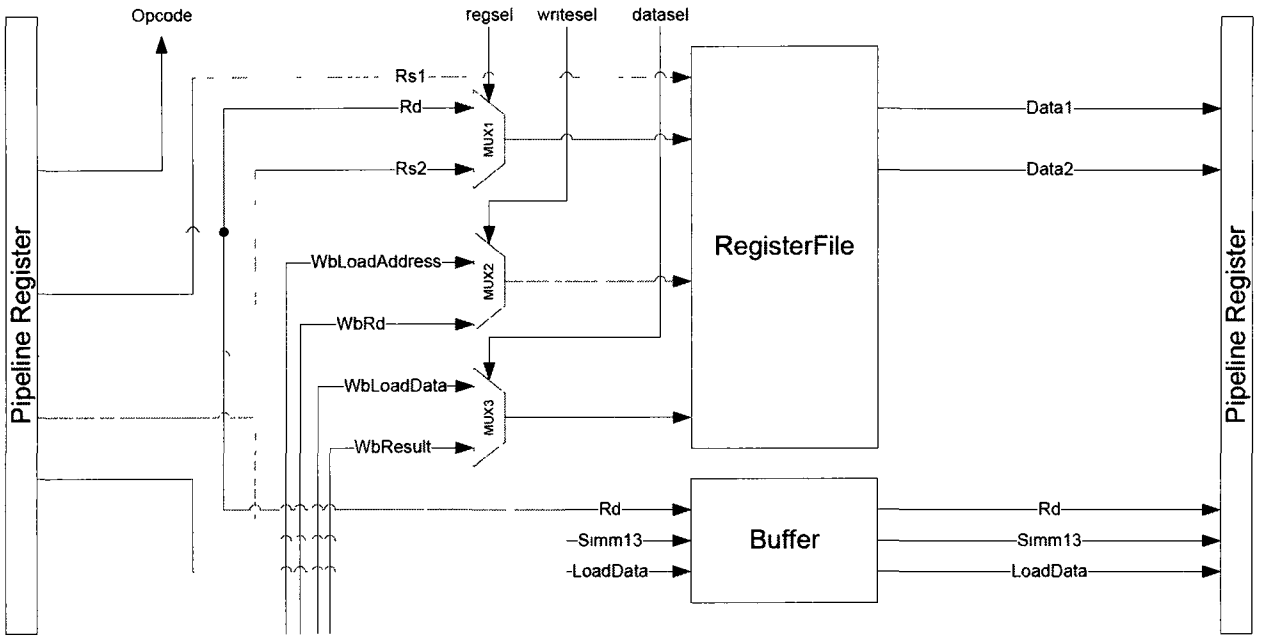

Figure 31: Register File Stage

For the ***RegisterFile*** component, the write address input is selected from ***Mux2*** where either the ***WbLoadAddress*** input or ***WbRd*** input is mapped to the write address port. These two inputs are from the write back stage where either the load instruction address or the operation instruction address is used to write data to the ***RegisterFile*** component. The ***writesel*** control signal for this multiplexer is driven from the ***GCU*** component.

The ***Mux3*** component selects what data is mapped to the write data input of the ***RegisterFile*** component. Similar to the write address input, either ***WbLoadData*** or ***WbResult*** inputs are mapped to the write data port. These two inputs are from the write back stage where either the load instruction data or the operation instruction result is written back to the ***RegisterFile*** component. The ***datasel*** control signal for this multiplexer is driven from the ***GCU*** component.

A multi-input buffer register is used to synchronize the outputs from the ***RegisterFile*** component to the ***Rd***, ***Simm13***, and ***LoadData*** outputs from the pipeline register. These signals are used in the execution stage for either load/store instructions or operation instructions.

The ***RegisterFile*** component is implemented as a LPM module where on-chip resources and memory bits are used. This implementation reduces the area of the component and achieves the maximum performance.

5.2.3 Execution

The execution stage contains the FLDPs where FL algorithms are executed in hardware. The execution stage also contains components for load and store instructions. The ***LoadUnit*** handles coprocessor load and store instructions and the ***FLDPUnit*** handles coprocessor operation instructions.

The ***LoadUnit*** computes the address for load operations. The address is a 32 bit value that is determined by the equation below.

$$address = r[rs1] + sign\_ext(simm13) \quad (16)$$

The ***address*** is determined by the content of ***Rs1*** and the function ***sign_ext*** performed on ***Simm13***. The content of ***Rs1*** is 32 bits thus ***Simm13*** is also extended to 32 bits to determine the ***address***.

For load operations, the contents of ***Rs1*** and ***Simm13*** move through the pipeline stages with the load data in parallel. After the ***LoadUnit*** generates the address, the load address and load data move to the next pipeline stage where the load data is stored to the generated address.

For store operations, ***Data2*** is simply passed through the ***LoadUnit*** and is mapped to the ***LoadData*** output.

The ***FLDPUnit*** is where the FLDP components are instantiated. The FLCU can support multiple FLDP components. Although unrealistic, the FLCU can support up to a maximum of 512 FLDP components. Operands ***rs1*** and ***rs2*** are buffered internally before being routed to all instantiated FLDP components.

During execution of a particular FLDP instruction, all other FLDP components are disabled. After execution, the ***results*** output from all FLDP components are routed to a multiplexer

where the LCU selects which output from the FLDP component is sent to the next stage of the FLCU datapath.

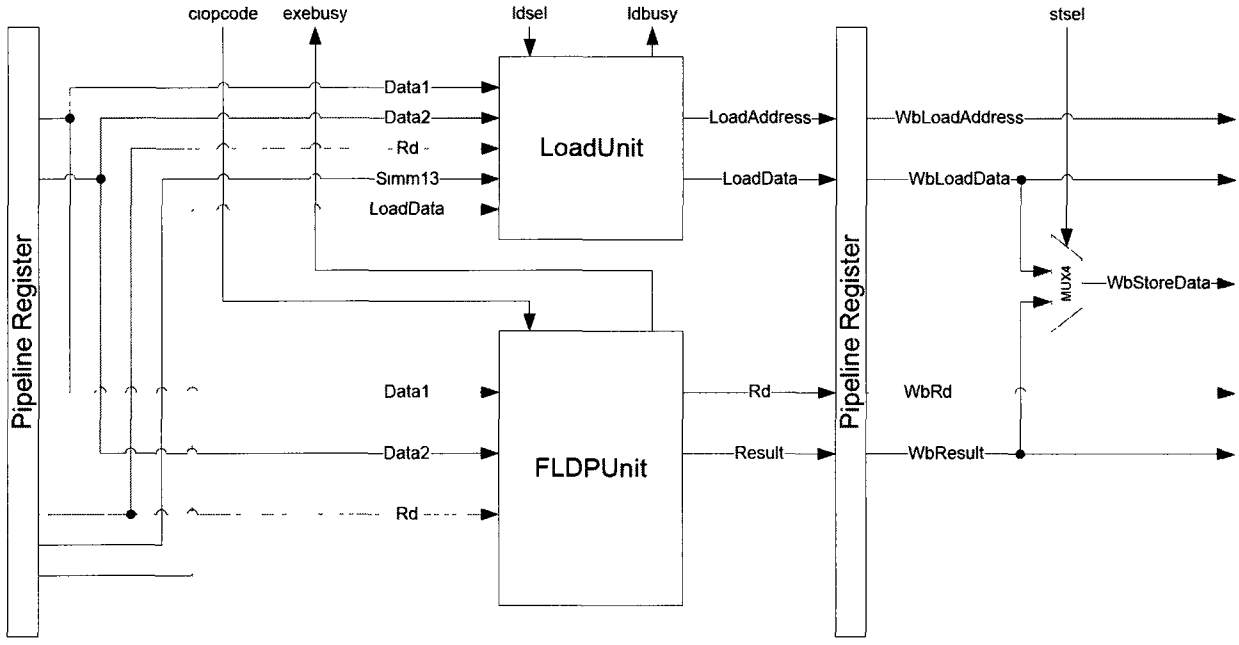

Figure 32: Execution Stage

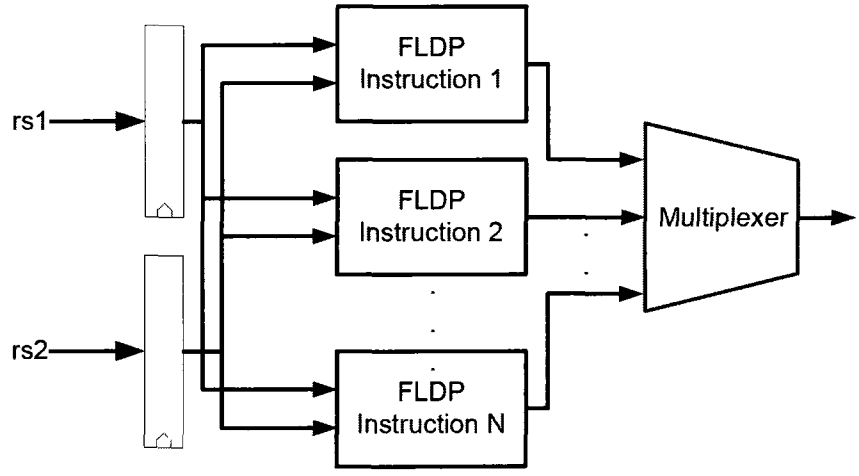

Figure 33: Execution Stage Internal Datapath

5.2.4 Write Back

The WB stage is the last stage in the FLCU datapath where load data or operation data is written back the register file in the RF stage. Signals ***WbLoadAddress*** and ***WbLoadData***

are mapped back to the RF stage's ***Mux4*** component where the load data is written into the register file.

For store operations, ***WbLoadData*** and ***WbResult*** are mapped to ***Mux4*** where the ***stsel*** signal from the GCU will select the data to write back into the Leon3 register file.

5.2.5 Global and Local Control Unit

The control path for the FLCU consists of two interdependent finite state machines (FSMs) that handle control for both the Leon3 coprocessor interface and the FLDP interface. The GCU maintains control the FLCU datapath and the LCU controls the FLDP datapath. The interdependency between the GCU and the LCU is based on the principles of distributed control.

Distributed control principles presented in [45] describes the methodology where each elementary operation has its own local control through the system. Each elementary operation has its own start and done signals that inform other elementary operations in the system. This creates a daisy-chain between all elementary operations where the next operation will wait until the previous operation is finished.

The GCU handles the control for the ID, RF, EX, and WB stages. The LCU handles the control inside the EX stage. When an instruction reaches the EX stage, the GCU passes control over to the LCU and the GCU waits for the LCU to finish. Once the LCU is finished, the LCU passes control back to the GCU and moves to the WB stage.

The LCU consists of a FSM counter. The counter is set to the latency of the FLDP. Once the counter has reached the latency, the LCU passes control back to the GCU. The GCU and LCU state machines are Mealy based and can be seen in Figure 34.

The advantage of applying distributed control principles to the FLCU allows the execution of the FLDP while the FLCU maintains the interface to the Leon3. Proper utilization of the FLDP is also achieved by the FLCU processing future FLDP operations while the FLDP is executing the current operation. The modular design of the control path allows addition of new FLDP components with minimal modification of the GCU.

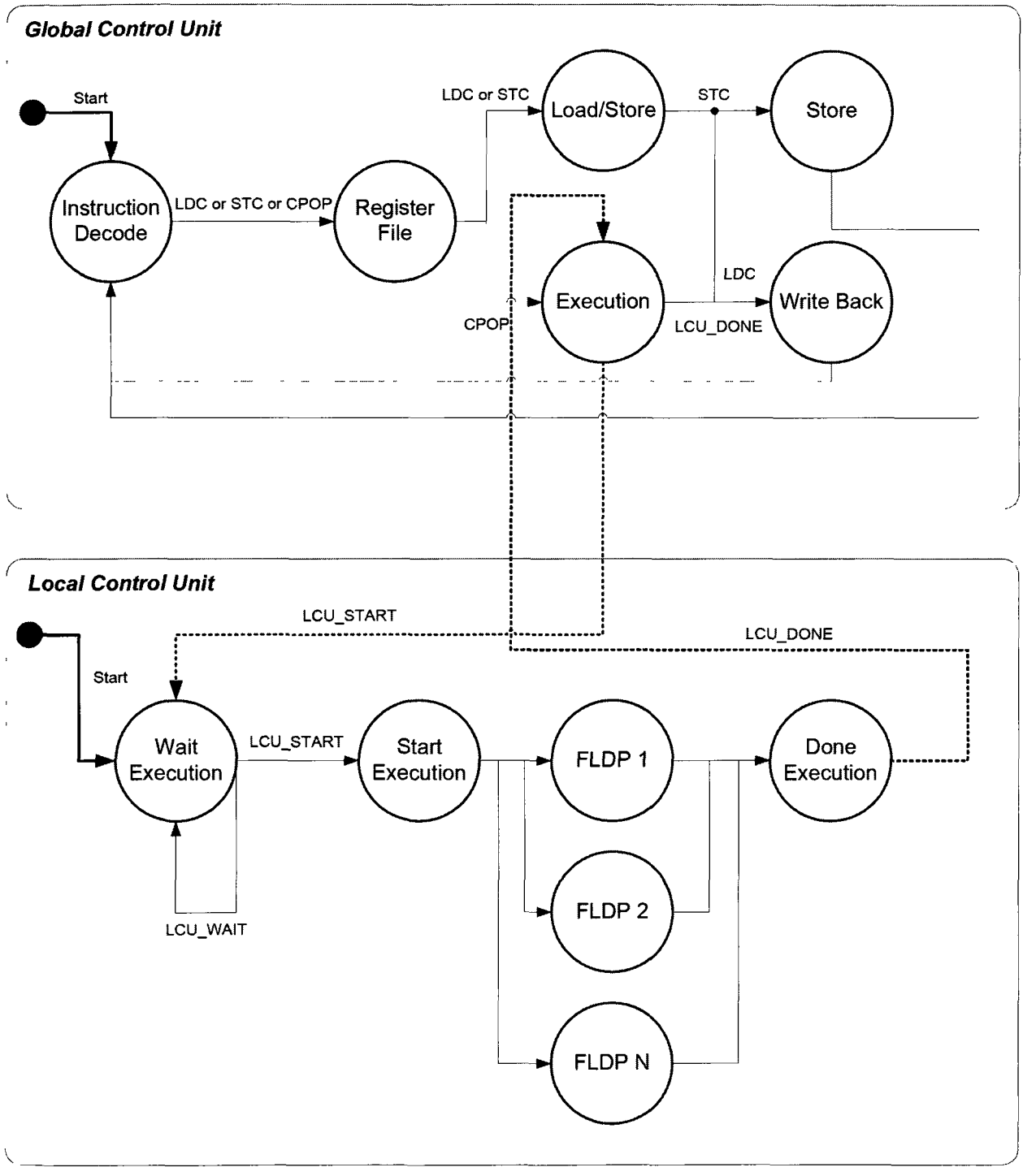

Figure 34: Global Control Unit and Local Control Unit State Machines

5.3 Integration with Leon3

The FLDP can be attached to the Leon3 soft processor through the FLCU. The FLCU receives coprocessor instructions from the Leon3 Integer Unit (IU), decodes coprocessor load, store, or operation instructions. The coprocessor instructions are executed in parallel with other Leon3 integer instructions. The FLCU maintains the coprocessor register file where register values are loaded and stored before and after running FLDP operations.

The FLCU provides a non blocking interface such that the overall performance of the system remains unchanged. Coprocessor instructions do not block the Leon3 IU pipeline and coprocessor instructions complete even if the Leon3 IU pipeline is blocked. The Leon3 IU pipeline is only stalled by the FLCU when there are operand or resource conflicts. Operand and resource conflicts are detected and managed by the IU.

Coprocessor instructions are dispatched from the Leon3 IU to the FLCU. The FLCU receives the coprocessor operation instruction and decodes it. The operand register addresses are passed to the register file where the operand values are retrieved and sent to the execution stage. The FLDP latches the operand values and executes the instruction. After the result is found, the result value is written back to the register file. By default, the Leon3 processor has a coprocessor port where an external coprocessor can be integrated. The floating-point unit and coprocessor port use the same port type.

Input Port to Leon3	Output Port to FLCU

`lddata : std_logic_vector(31 downto 0);`	`data : std_logic_vector(31 downto 0);`
	`ldlock : std_logic;`
	`holdn : std_ulogic;`

Figure 35: Leon3 Input and Output Port Definition

The input port to the Leon3's IU is from the FLCU. Signal ***lddata*** is used for store coprocessor instructions where data from the FLCU is moved into the Leon3's register file. The output port to the FLCU is from the Leon3's IU. Signal ***data*** is used to send coprocessor instructions to the FLCU. Signals ***ldlock*** and ***holdn*** are used for debug and testing purposes during simulation.

Chapter 6 - Software Configuration

This chapter presents how the FLDP and FLCU datapaths are configured from a Matlab model describing a FL algorithm. The FLDP and FLCU datapaths are designed using VHDL generics that allow the datapaths to be easily configured before compilation and synthesis. The FLDP configuration is based upon the FL algorithm model using Matlab's Fuzzy Logic Toolbox and the FLCU configuration is based upon the design requirements of how many FL algorithms are implemented as FLDPs. The FLDP configuration is performed first and then followed by the FLCU configuration. Both these configurations occur in a process where parameter setting, compilation, synthesis, and test bench generation are performed through software and TCL scripts.

6.1 FLDP Configuration

The FLDP configuration is based upon a FL algorithm model using Matlab's Fuzzy Logic Toolbox. The toolbox allows the design and simulation of FL algorithms by defining membership functions and fuzzy inference rules. The objective is to use the FL algorithm model designed using the Fuzzy Logic Toolbox to configure the FLDP datapath. The typical model parameters that will configure the FLDP include: input membership function shapes, output membership function shapes, fuzzy inference rules, etc.

The FLDP configuration flow shown in Figure 36 begins by analyzing the Fuzzy Inference System (FIS) model. Input membership functions, output membership functions, fuzzy inference rules are extracted as configuration parameters for the FLDP instantiation. The input and output value resolution is defined before the membership functions and fuzzy inference rules are translated. After the translation, the FLDP Fuzzification Stage, Inference Engine Stage, and Defuzzification Stage VHDL models are written and packaged together with a FLDP Top Level VHDL model.

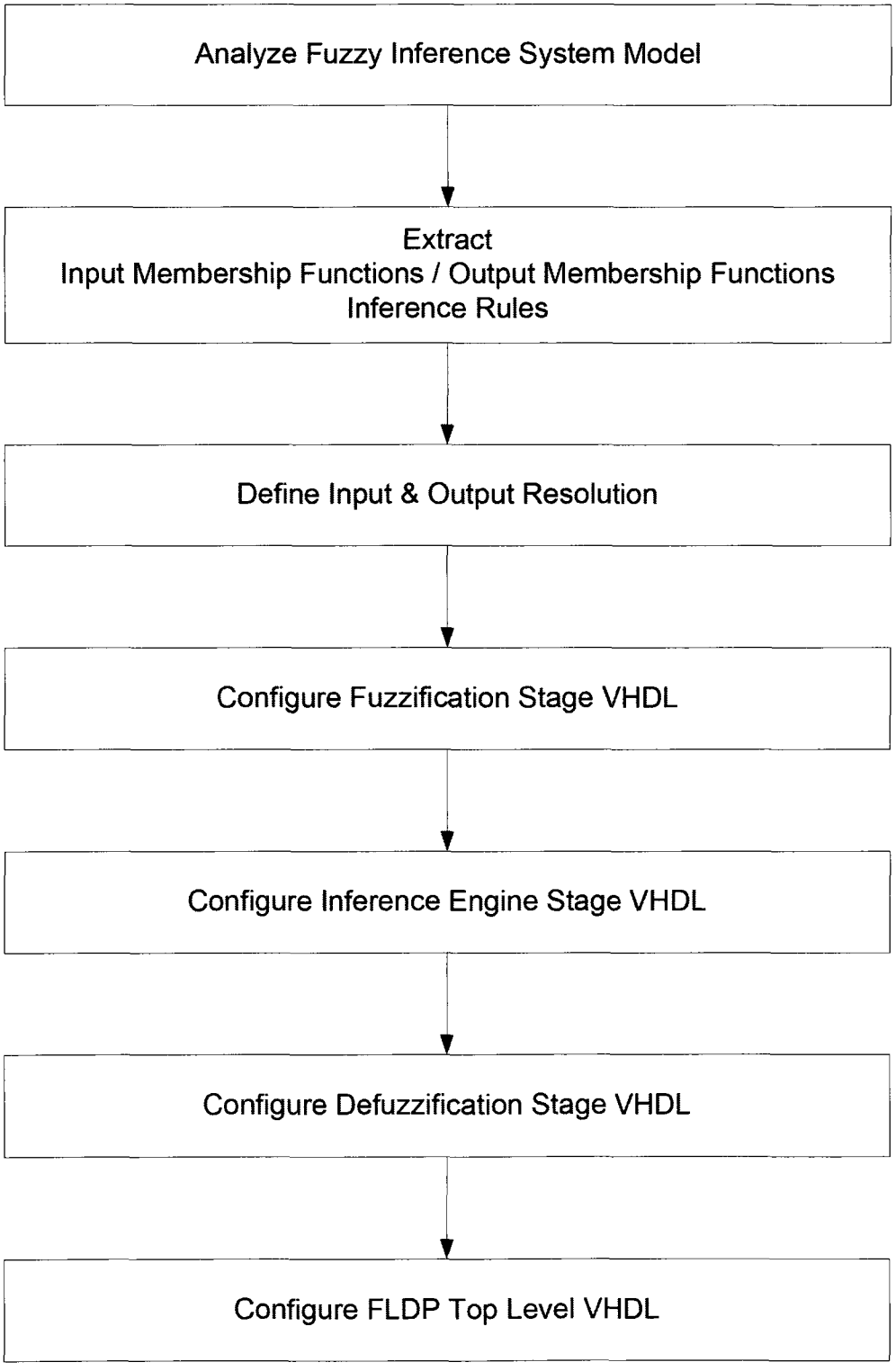

Figure 36: FLDP Configuration Flow

6.1.1 Fuzzy Inference System Model

Using the Fuzzy Logic Toolbox to model a FL algorithm produces a ***Fuzzy Inference System*** or ***FIS*** model that describes the FL algorithm. This model file contains many parameters that describe the inputs, outputs, fuzzy inference rules, membership function shapes, etc. Appendix A describes the model file of the FL algorithm for the Inverted Pendulum Control System. The FIS model file is divided into four sections: ***System, Inputs, Outputs,*** and ***Rules***.

The ***System*** section describes the model at a high level where the number of inputs, output, and fuzzy inference rules are defined. The type of inference engine, implication operators, and defuzzification methods are also defined in the ***System*** section.

The FIS model describes the membership function shapes by using scalar parameters that define the size of the shape. These parameters are used to define the shape such that the input or output can be translated to and from a fuzzy set using the membership function shapes. Using the Inverted Pendulum Control System as an example, triangular and trapezoidal shapes are used to describe the input and output membership functions.

For triangular membership function shapes, the curve is defined by three scalar parameters in the FIS model. The triangular curve is defined as follows:

$$f(x;a,b,c)=\begin{Bmatrix} 0, & x\le a \\ \dfrac{x-a}{b-a}, & a\le x\le b \\ \dfrac{c-x}{c-b}, & b\le x\le c \\ 0, & c\le x \end{Bmatrix} \tag{16}$$

The triangular membership function shape is defined as the function ***f(x,a,b,c)*** where parameters ***a*** and ***c*** describe the minimum and maximum bounds of the base of the shape and parameter ***b*** describes the peak of the triangle shape.

Trapezoidal membership function shapes are also described in a similar fashion as triangular membership function shapes in the FIS model. The trapezoidal membership function is defined by a curve vector with four scalar parameters. The trapezoidal curve is defined as follows:

$$f(x;a,b,c,d)=\begin{cases}0, & x\le a\\ \dfrac{x-a}{b-a}, & a\le x\le b\\ 1, & b\le x\le c\\ \dfrac{d-x}{d-c}, & c\le x\le d\\ 0, & d\le x\end{cases} \tag{17}$$

The trapezoidal membership function shape is defined as the function ***f(x,a,b,c,d)*** where parameters ***a*** and ***d*** describe the minimum and maximum bounds of the base of the shape and parameters ***b*** and c describes the peak minimum and maximum bounds of the trapezoidal shape.

The ***Inputs*** section of the FIS model describes the input membership functions. The types of shapes and size in terms of coordinates are described. Each input is described by a set of membership function shapes and a range for the set. Each shape is described by a label, the type of shape, and a set of scalar parameters that define the shape.

The ***Outputs*** section of the FIS model describes the output membership functions. Similar to the ***Inputs*** section, the ***Outputs*** section describes the range of the output and the number of membership functions. Each membership function shape is defined by a label, the type of shape, and a set of scalar parameters that define the shape.

The last section of the FIS model describes the fuzzy inference rules in the ***Rules*** section. The rules are defined in a simple format. Each rule is described using an if-then statement. For example, if X and Y then Z. In the FIS model, the if-then statement would be described as [X Y, Z (1):1].

6.1.2 Membership function shapes

The input membership functions are translated into hardware lookup tables where each valid input value is mapped to its corresponding fuzzy set value. Before this lookup table is populated with data, the resolution of the input needs to be determined.

The input resolution defines what input values will be translated into corresponding fuzzy set values. For example, for the input range is [-1,1] and an input resolution of 0.25, the accepted input vales that will be translated to fuzzy set values are [-1, -0.75, -0.50, -.025, 0, 0.25, 0.50, 0.75, 1]. A fuzzy set consists of two elements; fuzzy region and fuzzy value. For

each input membership function, the fuzzy region is assigned a binary code. For each input value ***x***, its corresponding fuzzy value is determined by computing the area under the triangular and trapezoidal shapes. Figure 37 illustrates how each input value [***x1,x2,x3,...,xN***] is translated to a fuzzy value.

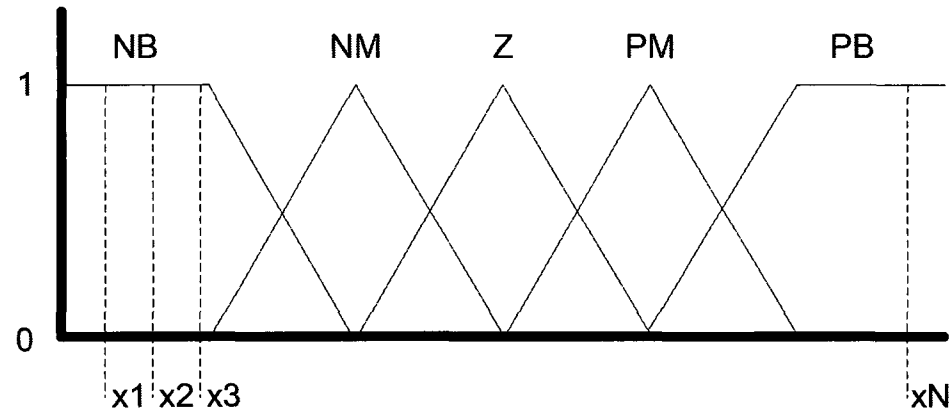

Figure 37: Input Membership Function Translation

For each increment based on the input resolution, the fuzzy set translation is determined and the Fuzzification Stage VHDL model is written to handle the incremented input.

Using translation algorithms to map membership function shapes in lookup table simplifies the decoding of membership function shapes in hardware. The translation algorithms use membership function shapes described in the FIS model and the desired input resolution to determine the equivalent fuzzy sets for the real input.

Table 5: Definitions for Membership Function Shape Translation Algorithms

Variable	Definition
resolutionStep	Resolution step in the membership function shape range
shapeRange	Minimum and maximum bounds of the membership function shape
lowerBound	Minimum bound of membership function shape
midBound	Midpoint of membership function shape
upperBound	Maximum bound of membership function shape
fuzzySetValue	Fuzzy set describing fuzzy region and value
shapeArea	Area under membership function shape

Algorithm 1: Trapezoid Membership Function Shape Translation

```
1:  for each resolutionStep in shapeRange
2:      if (resolutionStep >=lowerBound & resolutionStep<=midBound) then
3:          fuzzySetValue[resolutionStep]=1
4:      elseif (resolutionStep>=midBound & resolutionStep<=upperBound) then
5:          shapeArea=(0.5) * (lowerBound - midBound) * (1)
6:          fuzzySetValue[resolutionStep]=shapeArea + (midBound - resolutionStep)
7:                                                    upperBound - midBound
8:      endif
9:  end for
```

Algorithm 2: Triangular Membership Function Shape Translation

```
1:  for each resolutionStep in shapeRange
2:      if (resolutionStep >=lowerBound & resolutionStep<=midBound) then
3:          shapeArea=(0.5) * (lowerBound - midBound) * (1)
4:          fuzzySetValue[resolutionStep]=shapeArea + (resolutionStep - midBound)
5:                                                    lowerBound - midBound
6:      elseif (resolutionStep>=midBound & resolutionStep<=upperBound) then
7:          shapeArea=(0.5) * (lowerBound - midBound) * (1)
8:          fuzzySetValue=shapeArea + (midBound - resolutionStep)
9:                                                    midBound - upperBound
10:     endif
11: end for
```

Algorithm 1 describes the translation of trapezoid membership function shapes. This translation is an intermediate step where all fuzzy sets and determined before the configuration the fuzzification stage in the FLDP.

Lines 1-9 describe the increment steps through the range of the trapezoid membership function shape. Lines 2-3 describe the translation when the resolution step is between the minimum bound and the midpoint of the trapezoid membership function shape. The translation here is simple since the area between the minimum bound and the midpoint is always one. Lines 4-8 describe the translation when the resolution step is between the midpoint and the maximum bound of the trapezoid membership function shape. The translation determines the area under the trapezoid membership function shape using the current resolution step is in the range with respect to the minimum and maximum bounds.

Algorithm 2 similarly describes the translation of triangular membership function shapes. Lines 1-11 describe the increment steps through the range of the triangular membership function shape. The resolution step between the minimum bound and midpoint is translated by determining the area under the triangular membership function shape. Similarly, the resolution step between the midpoint and the maximum bound is translated by determining the area under the triangular membership function shape.

Once all resolution steps are translated to a fuzzy set, the next step is to configure the fuzzification stage of the FLDP. The fuzzification stage is configured by writing the fuzzy sets as lookup tables. This is achieved by writing to the VHDL model of the fuzzification stage where simple if-then statements are used. Inside the process statement, the if-then statements are then synthesized as hardware lookup table.

6.1.3 Inference rules

The fuzzy inference rules in the FIS model describe the output behaviour of the control system based on the input values. The fuzzy inference rules only evaluate the fuzzy regions and the evaluation of the fuzzy values occurs in the Defuzzification Stage of the algorithm. Similar to the membership functions, each fuzzy region in each input and output membership function is given a binary code based on the fuzzy region's integer code in the FIS model. Each fuzzy inference rule is described in the following format:

$$[X\ Y,\ Z]\ (n) : m$$

In the above format, ***X*** is the first argument in the conditional statement. Variable ***Y*** is the second argument and the consequent ***Z*** is based upon the implication connection between ***X*** and ***Y***. The implication is set by parameter ***m***, where the implication factor can either be intersection connection (and) or union connection (or). If the implication factor is using a intersection connection then both ***X*** and ***Y*** needs to be true. If the implication factor is using a union connection then either ***X*** or ***Y*** needs to be true. The parameter ***n*** defines the weight factor of the consequent ***Z***.

Table 6: Definitions for Fuzzy Inference Rule Extraction Algorithm

Variable	Definition
Rule	Fuzzy inference rule described in FIS model
fuzzyRules	Set of fuzzy inference rules described in FIS model
ruleInference	Fuzzy inference operator defined in FIS model
fuzzyAndArg1, fuzzyAndArg2	Fuzzy sets for AND inference comparison
fuzzyAndArg3	Fuzzy set consequent from AND inference comparison
fuzzyOrArg1, fuzzyOrArg2	Fuzzy sets for OR inference comparison
fuzzyOrArg3	Fuzzy set consequent from OR inference comparison
fuzzyInference	Fuzzy inference operator

Algorithm 3: Fuzzy Inference Rule Extraction

```
1:    for each rule in fuzzyRules
2:        if (ruleInference is AND) then
3:            fuzzyAndArg1←ruleArg1
4:            fuzzyAndArg2←ruleArg2
5:            fuzzyAndArg3←ruleArg3
6:            fuzzyInference←AND
7:        elseif (ruleInference is OR)then
8:            fuzzyOrArg1←ruleArg1
9:            fuzzyOrArg2←ruleArg2
10:           fuzzyOrArg3←ruleArg3
11:           fuzzyInference←OR
12:       endif
13:   end for
```

Algorithm 3 describes how fuzzy inference rules are extracted from the FIS model. This extraction is an intermediate step where all the rules are extracted and then the inference stage of the FLDP is configured.

Lines 1-13 describe the loop where all possible fuzzy inference rules are evaluated in the FIS model. Lines 2-6 describe how the fuzzy inference rule is evaluated if the fuzzy inference operator is an AND. The fuzzy sets used for comparison and the fuzzy set consequent are extracted from the FIS model. Similarly, lines 7-11 extract fuzzy inference rules if the fuzzy inference operator is an OR. Rules for fuzzy inference AND operator and OR operator are separated here for easier configuration of the inference rule stage in the FLDP.

Once all the rules are evaluated from the FIS model, the fuzzy inference rules in the intermediate format are then configured in the inference rule stage of the FLDP. The inference rule stage is configured by writing the fuzzy inference rules as lookup tables. This is achieved by writing to the VHDL model of the inference rule stage with simple if-then statements. Inside the process statement, the if-then statements are then synthesized as a lookup table.

6.2 FLCU Configuration

The FLCU configuration is based on how many instances of the FLDP are need to be interfaced with the Leon3 processor. The FLCU can support multiple instances of the FLDP using different Leon3 coprocessor operation codes. Each FLDP instance is integrated in the FLDP unit inside the execution stage of the FLCU. A top level VHDL wrapper model is used to integrate the FLDP into the FLDP Unit in the FLCU Execution Stage.

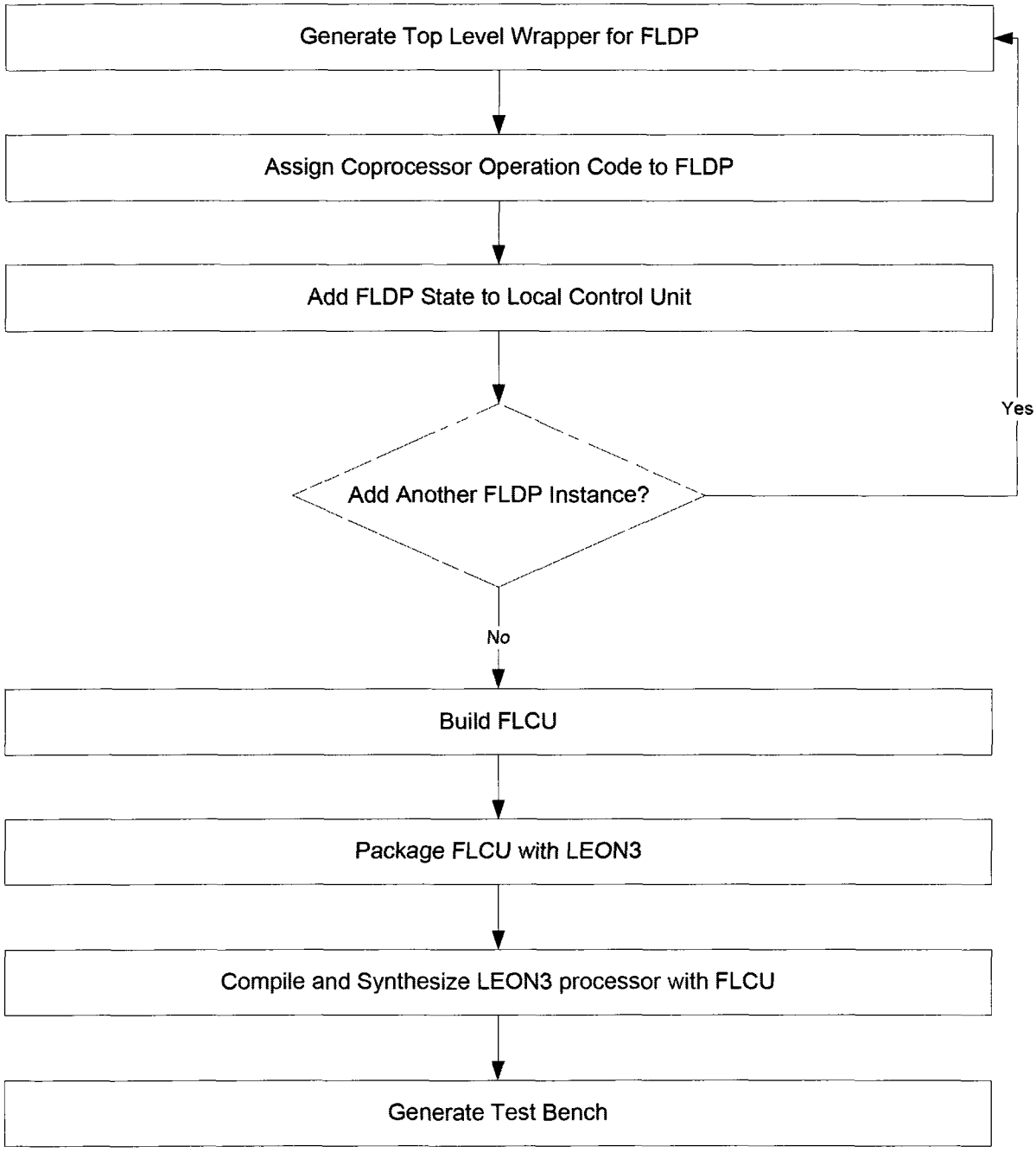

Figure 38: FLCU Configuration Flow

To add a FLDP instance to the FLCU, a top level wrapper is generated. The wrapper consists of input and output signals for (rs1), (rs2), (rd), (clock), (reset), and (enable). The (rs1), (rs2), and (rd) signals are all 32 bits wide. These are the base signals required for the FLDP, but additional signals can be added if required.

The next step is to assign a coprocessor operation code for the FLDP top level wrapper. The coprocessor operation is nine bits and can support up to 512 unique instructions. Along with assigning a coprocessor operation code, the execution stage in the FLCU needs to be configured to support the new top level wrapper. Algorithm 4 describes the configuration of the execution stage of the FLCU.

Table 7: Definitions for FLCU Execute Stage Configuration Algorithm

Variable	Definition
FLDPcomponent	FLDP component
allFLDPcomponents	All FLDP components to be implemented
assignedInstructions	Count of how many coprocessor instructions implemented
totalInstructions	Maximum number of coprocessor instructions supported
fldpInstruction	Top level VHDL model for FLDP component
fldpMux	Top level VHDL model of FLDP component select

Algorithm 4: FLCU Execution Stage Configuration

```
1:    for each FLDPcomponent in allFLDPcomponents
2:        if(assignedInstructions<totalInstructions) then
3:            fldpInstruction←FLDPcomponent
4:            fldpMux←FLDPcomponent
5:        endif
6:    end for
```

Algorithm 4 generates the top level wrapper and configures the output multiplexer to support the new FLDP component. The execution stage consists of a multiplexer that selects what output from the FLDP component is mapped to the output of the execution stage. The multiplexer is altered by adding an additional port and an additional select bit for the new FLDP component.

Lines 1-6 describe the loop where all FLDP components to be integrated in the FLCU execution stage are evaluated. Lines 2-5 describe how a new top level VHDL model is generated for the new FLDP component and the top level VHDL model of the output

multiplexer is modified to support the new FLDP component. These lines only get executed if there are available coprocessor operation codes. If no coprocessor operation codes are available, the FLDP component is not integrated in the FLCU execution stage.

The next step is to configure the Local Control Unit to support the new FLDP component. A new state is added to the LCU state machine. Algorithm 5 describes the configuration of the LCU. For each FLDP component added, a new state is added to the state machine. With the new state, a counter is seeded with the latency of the FLDP component.

Table 8: Definitions for FLCU Local Control Unit Configuration Algorithm

Variable	Definition
FLDPcomponent	FLDP component
allFLDPcomponents	All FLDP components to be implemented
assignedInstructions	Count of how many coprocessor instructions implemented
totalInstructions	Maximum number of coprocessor instructions supported
lcuNewState	New state added to LCU FSM
lcuLatencyCounter	New latency counter for FLDP component added to LCU FSM

Algorithm 5: FLCU Local Control Unit Configuration

```
1:    for each FLDPcomponent in allFLDPcomponents
2:        if(assignedInstructions<totalInstructions) then
3:            lcuNewState←FLDPcomponent
4:            lcuLatencyCounter←FLDPcomponentLatency
5:        endif
6:    end for
```

Algorithm 5 configures the LCU FSM to support the newly added FLCP component. Lines 1-6 describe the loop where all FLDP components to be added to the LCU FSM are evaluated. Lines 2-5 describe how a state is added to the LCU FSM. Along with the new state, a new latency counter is also seeded with the latency of the FLDP component. These lines only get executed if there are available coprocessor operation codes. If no coprocessor operation codes are available, the FLDP component is not added to the LCU FSM.

The last step is to package all the FLCU VHDL models together before integrating the FLCU with the Leon3 processor VHDL models. After all the FLCU VHDL models are packaged together, the FLCU design is compiled before the design is integrated with the Leon3 processor. After compilation, the FLCU VHDL model is integrated with the Leon3 processor VHDL models. The processor is already pre-configured and synthesized. To use another configuration of the Leon 3 processor, the new configuration must support coprocessor instructions.

The final step is to compile the Leon 3 with the coprocessor interface. A new Quartus II project is generated and the designs are compiled and synthesized using Altera Quartus II TCL scripts. Once the design is synthesized, a test bench for the coprocessor interface is generated. The test bench is used for simulating the FLCU interface to observe the behaviour of the FLDP components added to the FLCU.

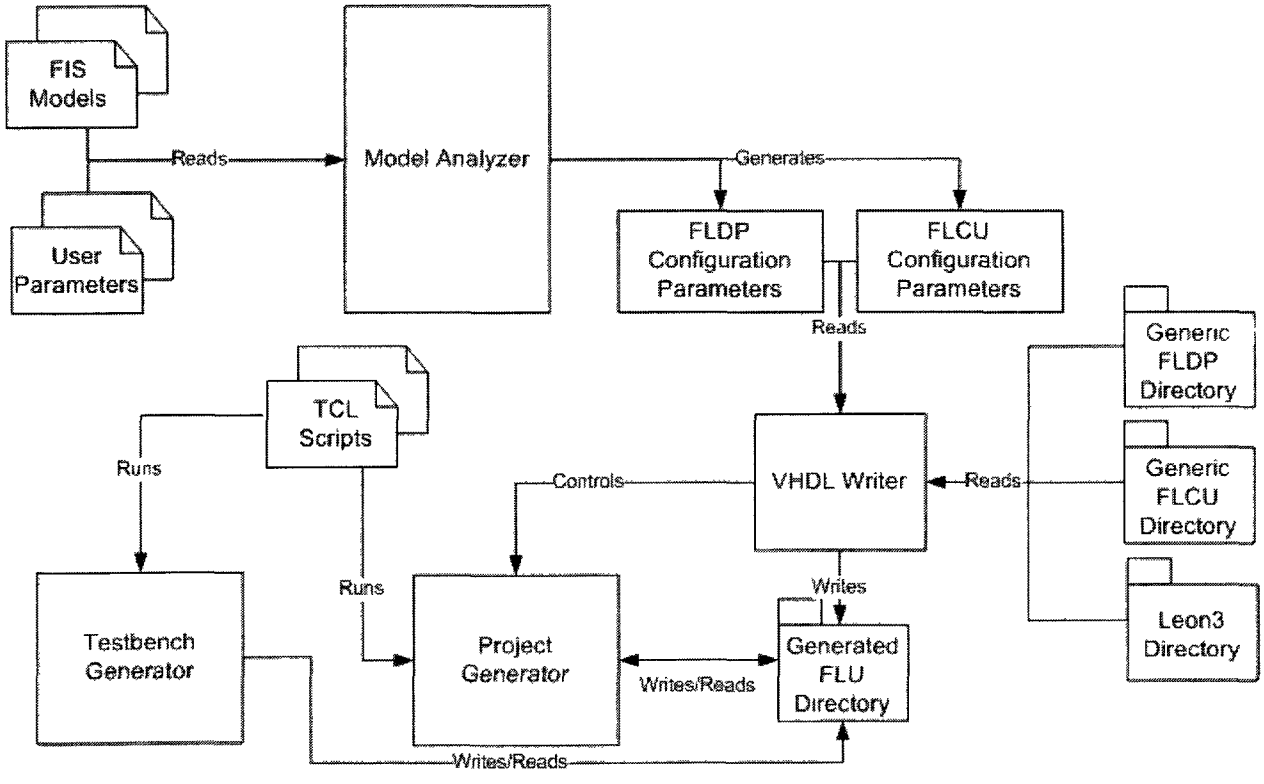

Figure 39: Software Configuration Flow

Figure 39 illustrates the software configuration flow for the FLU. The configuration begins with Matlab FIS models that describe the FL algorithm and user parameters that describe the processor configuration, number of FL algorithms, and resolution of inputs and outputs. The model analyzer uses Algorithm 1, Algorithm 2, and Algorithm 3 to extract FL configuration parameters from the FIS model. The model analyzer separates the configuration parameters for both FLDP and FLCU components.

The VHDL writer then uses the configuration parameters for both FLDP and FLCU components to generate VHDL models. Generic VHDL models are used as templates and are configured using the FLDP and FLCU configuration parameters. Algorithm 4 and Algorithm 5 are used to configure the FLCU and the processor configuration parameters configure the Leon3 processor.

TCL scripts are used to generate the Altera Quartus II design project and ModelSim simulation project. Project settings such as VHDL models, compilation parameters, and target FPGA device are defined. Compilation and synthesis is then executed using TCL scripts.

Chapter 7 - Results

This chapter presents performance and area analysis of the FLU with the Leon3 processor. Two case studies are presented and implemented across four different platforms to understand the advantages and disadvantages of FL algorithms implemented as software and hardware. The implementation platforms are divided between open source and commercially available tools. The first two implementation platforms use the Leon3 processor whereas the other two use Tensilica's Xtensa 7 processor. Both processor platforms allow processor configuration and software development.

7.1 Implementation Platforms

The first implementation platform uses only the Leon3 processor to execute the FL case studies. This implementation will establish a baseline where the entire FL algorithm is implemented in software. Aerolfex Gaisler provides a Base-C Cross Compilation (BCC) System to allow software development on the Leon3 processor. The software environment allows programs to be written in C/C++ and are compiled and translated into assembly instructions for the Leon3. This implementation platform is referred to as (LEON3).

The second implementation platform will use the FLU integrated with the Leon3 processor. The FL case studies will be modelled in Matlab using the Fuzzy Logic Toolbox and then the model will configure the FLCU and FLDP components. Once configured, the FLU is integrated with the Leon3 processor. The same software code is used from the LEON3 implementation, however all assembly instructions referring to the FL in software will be replaced by a single coprocessor instruction that calls the FLU. This implementation platform is referred to as (LEON3FLU)

The third and fourth implementation platforms will use a commercially available tool provided by Tensilica. Tensilica's Xtensa Xplorer IDE supports software development for the Xtensa processors. With Xplorer, applications can be designed, profiled, and configured to run on the Xtensa processor. The Xtensa processor can also be extended by using TIE to

design specialized hardware that can be integrated with the processor. Xplorer allows multiple iterations of analysis and optimization before the processor is generated.

The third implementation platform will use Tensilica's Xtensa LX2 processor where again the FL case studies will be implemented entirely in software. The Xtensa LX2 processor is a configurable and extensible processor with many configuration options. This implementation platform will establish a baseline of implementing FL case studies on a commercially available processor. This implementation platform is referred to as (TXP).

The fourth implementation platform is the Xtensa LX2 processor with hardware extensions. Tensilica Instruction Extensions (TIE) will be used to implement FL in hardware. TIE offers flexibility by adding multi-cycle, pipelined execution units, register files, arithmetic and other logic units to the Xtensa processor. With TIE, developers can create their own custom instructions with the Xtensa processor. For both case studies the FL TIE is designed manually. Deep knowledge of the Xtensa LX2 and TIE language assisted in designing efficient TIE operations for FL. This implementation platform is referred to as (TXPTIE).

7.2 Characterization Definitions

To evaluate implementation platforms, a set of performance and area metrics need to be defined. These metrics will evaluate each implementation platform such that a proper evaluation can take place.

7.2.1 Performance

Performance is measured in terms of execution time or response time. For the evaluation in this thesis, performance is defined as follows:

$$Performance_x = \frac{1}{Execution\,Time_x} \tag{18}$$

To evaluate the performance of any two implementations, if the performance of X is greater than the performance of Y, we then have the following:

$$Performance_X > Performance_Y$$

$$\frac{1}{Execution\,Time_X} > \frac{1}{Execution\,Time_Y} \tag{19}$$

$$Execution\,Time_Y > Execution\,Time_X$$

That is, when the execution time of *Y* is longer than that of *X*, then *X* is faster than *Y*. To relate the performance of two different implementations quantitatively, the phrase often used is "*X* is *n* times faster than *Y*". This translates to:

$$\frac{Performance_X}{Performance_Y} = n \tag{20}$$

7.2.2 Area

For designs implemented in the ASIC design flow, area of the design is referred to as gate count. Gate count refers to the number of transistors used to implement a design. The number of gates required to implement a design is dependent on the type of process technology used [46].

For designs implemented in the FPGA design flow, area of the design can be referred in terms of logic elements or logic cells. FPGA vendor Altera refers the smallest unit of logic as a Logic Element (LE) and Xilinx refers to it as a Logic Cell (LC). Using Altera as an example, one LE configuration consists of a four input lookup table, programmable flip-flop, and a interconnect interface. Many LEs can be connected together and programmed to perform a specific function on the FPGA. There are many other area metrics used in the FPGA design flow such as Logic Utilization, Registers and Memory Bits. Logic Utilization refers to the total percentage of FPGA logic used by the design and is expressed by the total summation of lookup tables, registers, and logic. Memory Bits refers to the total number of memory bits used on the FPGA based upon the total number of memory bits available [42].

7.3 Inverted Pendulum Control System Case Study

The Inverted Pendulum Control System describes a system where there is a pole of mass m, on a cart M that is moving in a straight line along the axis x. The motion of M is caused by an impulse force F. The inertia of the pole causes it to deviate from the normal by angle θ when force is applied to the cart [44].

The problem is to control the magnitude of force F such that the pole does not swing too much causing a horizontal fall. To determine how much force is required, the FL algorithm uses the pole angle and the pole angular velocity as input parameters. The pole angular velocity is determined by using the pole angle and the velocity of the cart is determined by the rotation of the angle.

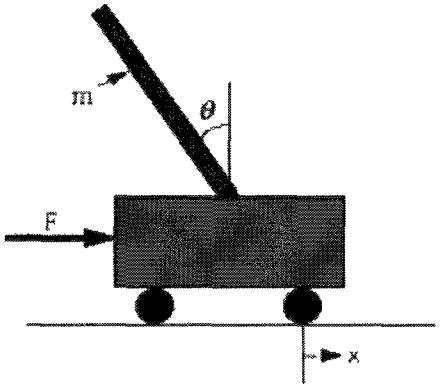

Figure 40: Inverted Pendulum Control System

The Inverted Pendulum Control System is designed using a FL algorithm with two inputs and one output with trapezoidal and triangular membership functions with 25 fuzzy inference rules. The pole angle and angular velocity input membership functions are presented in Figure 41 and Figure 42. The fuzzy regions are: NB(Negative Big), NM(Negative Medium), Z(Zero), PM(Positive Medium), and PB(Positive Big). The force output membership function is shown in Figure 43 with fuzzy regions NB(Negative Big), NM(Negative Medium), NS (Negative Small), Z(Zero), PS(Positive Small) PM(Positive Medium), and PB(Positive Big). The fuzzy inference rules for the inverted pendulum control system are shown in Table 9.

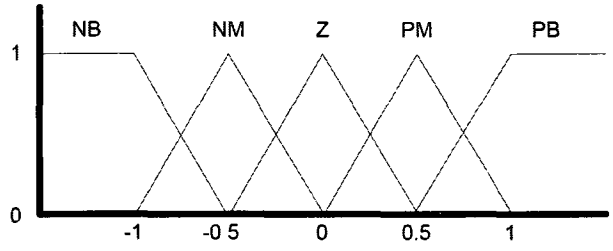

Figure 41: Pole Angle Input Membership Function

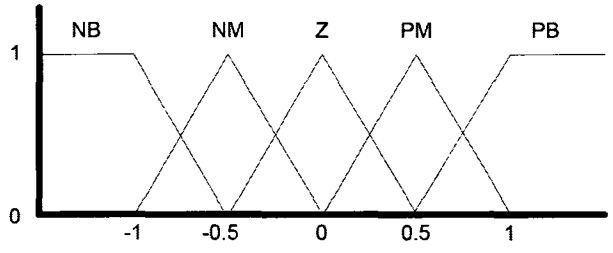

Figure 42: Pole Angular Velocity Input Membership Function

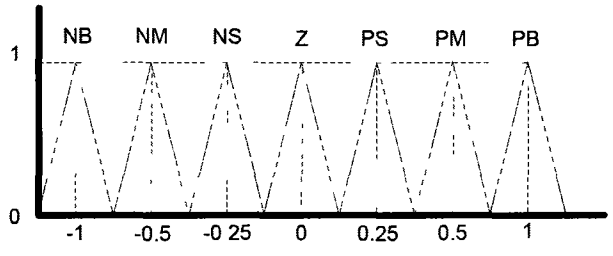

Figure 43: Force Output Membership Function

Table 9: Inverted Pendulum Fuzzy Inference Rules

Pole Angle	Pole Angular Velocity				
	NB	**NM**	**Z**	**PM**	**PB**
NB	NM	NM	NS	Z	Z
NM	NM	NM	NS	Z	Z
Z	NS	NS	Z	PS	PS
PM	Z	Z	PS	PM	PM
PB	Z	Z	PS	PM	PM

7.4 Small Scale Food Preparation Control System Case Study

The Small Scale Food Preparation Control System is designed to produce the exact amount of doneness requested by the user, taking into account the varying thicknesses of the steaks being used. The control system will utilise FL for approximating the best cooking times for products with varying size and required doneness. The FL algorithm uses trapezoidal and triangular functions and the inputs and outputs of the system. Due to the fact that the current implementation deals with two input one output designs, the inputs to the cooking FLC are the thickness and requested doneness with a range of one to five and one to three, respectively [47].

The thickness input membership function consists of trapezoidal and triangular shapes. The thickness fuzzy regions are defined in terms of inches and include: ¾ in, 1 in, 1 ¼ in, 1 ½ in, and 1 ¾ in. The doneness input membership function fuzzy regions include: rare, medium, and well. The cooking time output membership function consists of triangular shapes and the fuzzy regions are defined in terms of time in seconds and the fuzzy regions range from 270 to 700 seconds with 30 second intervals.

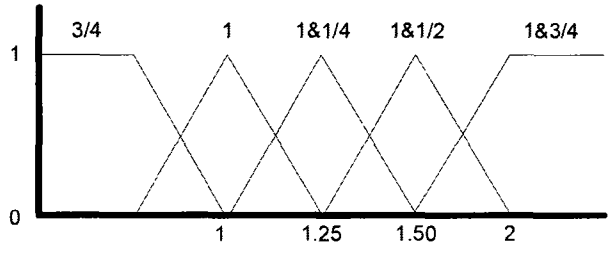

Figure 44: Thickness Input Membership Function

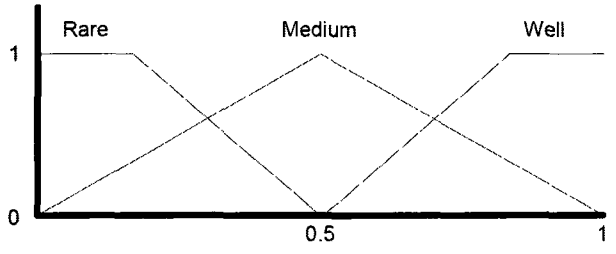

Figure 45: Doneness Input Membership Function

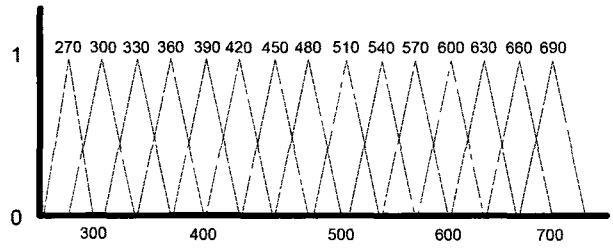

Figure 46: Cooking Time Output Membership Function

The FL algorithm consists of 15 fuzzy inference rules that control the output cooking time depending on the thickness and doneness inputs.

Table 10: Cooker Fuzzy Inference Rules

Doneness	Thickness				
	¾"	1"	1 & ¼"	1 & ½"	1 & ¾"
Rare	270	330	360	390	450
Medium	360	420	450	480	510
Well	540	600	660	720	780

In the event that the temperature strays from the optimum, the controller must be able to adjust the oven temperature back towards the optimum as quickly as possible. Should the temperature stray too far from the optimum such that the cooking time of the product will be affected, the controller must also be able to adjust the amount of time required to cook the product.

The second FL algorithm maintains the temperature using two inputs and one output with 25 fuzzy inference rules. The temperature input membership function consists of trapezoidal and triangular shapes. The fuzzy regions include: Low Large (LL), Low Small (LS), Exact (EX), High Small (HS), and High Large (HL). The temperature change input membership function consists of fuzzy regions: Decrease Fast (DF), Decrease Slow (DS), Zero (Z), Increase Slow (IS) and Increase Fast (IF). The oven control output membership function consists of trapezoidal and triangular shapes and the fuzzy regions include: Off (O), On Minimum (OM), Stable (S), On High (OH) and On Full (OF).

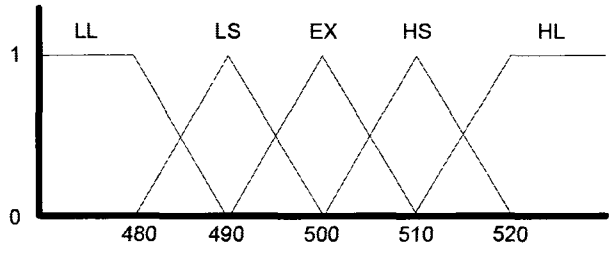

Figure 47: Temperature Input Membership Function

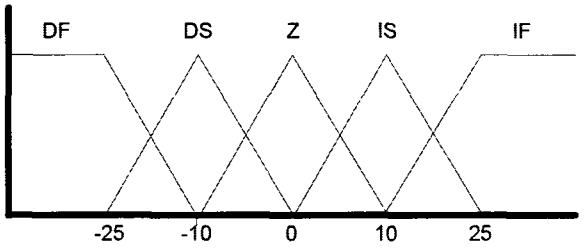

Figure 48: Temperature Change Input Membership Function

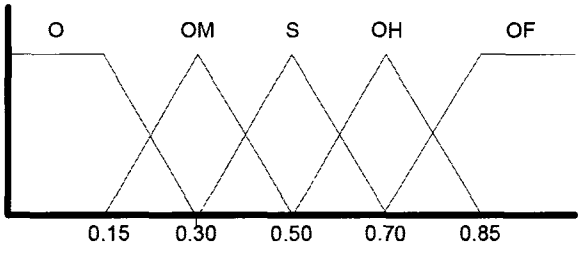

Figure 49: Oven Control Output Membership Function

Table 11: Temperature Fuzzy Inference Rules

Current Temperature	Temperature Change				
	DF	**DS**	**Z**	**IS**	**IF**
LL	OF	OF	OF	OH	OH
LS	OF	OH	OH	S	S
EX	OH	S	S	OM	OM
HS	S	OM	OM	O	O
HL	OM	O	O	O	O

7.5 Leon3 Implementation

For this implementation platform, both case studies are entirely written in software. There is no defined methodology of developing FL algorithms in software, thus the FL algorithm is written in C using basic language functions. The source code mainly consisted of if-then-else statements and multidimensional arrays. Arithmetic operations such as addition, multiplication, and division are also used. Standard C libraries stdio.h and stdlib.h are used in the software that provide macro definitions, constants, declarations of functions and types, memory allocation, process control and conversions. These are common libraries used in most C based applications. The Leon3 processor configuration used is shown in Appendix B. The software is compiled using BCC and the Leon3 processor is compiled and synthesized using Altera's Quartus II targeting Altera Cyclone III EP3C25 FPGA device.

Table 12: Case Study 1 LEON3 Implementation Summary

Execution Time in Clock Cycles	1500
Maximum Operating Frequency (MHz)	125
Total logic elements	7472
Total combinational functions	7112
Dedicated logic registers	2195
Total memory bits	49152
Embedded Multipliers	2
Total PLLs	1

Table 13: Case Study 2 Leon3 Implementation Summary

Execution Time in Clock Cycles	3158
Maximum Operating Frequency (MHz)	125
Total logic elements	7472
Total combinational functions	7112
Dedicated logic registers	2195
Total memory bits	49152
Embedded Multipliers	2
Total PLLs	1

7.6 Leon3 with Fuzzy Logic Unit Implementation

For this implementation platform, both case studies are implemented using the FLDP and FLCU datapaths and software presented in Chapter 5 and 6 from this thesis. Each FL algorithm is modelled using Matlab's Fuzzy Logic Toolbox. The model is then analyzed by the FLU generator software that extracts FL parameters to configure the FLDP. The FLDP is then integrated with the FLCU and Leon3 processor. A new Altera Quartus II design project is created, all VHDL models compiled, and then the entire design is synthesized. The Leon3 processor is at its default configuration shown in Appendix B. The FL software from the Leon3 implementation is then modified where the entire FL code is replaced by coprocessor instruction calls to the FLU. The design is compiled and synthesized using Altera's Quartus II targeting Altera Cyclone III EP3C25 FPGA device.

The software from the baseline Leon3 implementation is used as a template to develop the software for the Leon3 with FLU implementation. The assembly instructions from the Leon3 implementation executing FL in software are replaced by coprocessor assembly instructions. Coprocessor load and store assembly instructions are used to move data to and from the FLU. Before executing a coprocessor operation instruction, a coprocessor load instruction moves data to the FLCU register file. Then the coprocessor operation instruction is executed. After the operation, a coprocessor store instruction moves data from the FLCU register file to the Leon3's register file. These load and store coprocessor instructions are considered overhead when using the FLU.

For a coprocessor operation instruction, the worse case overhead is three coprocessor instructions. Two coprocessor load instructions move two operands from the Leon3 to the FLU. After the coprocessor operation instruction, one coprocessor store instruction moves the result from the FLU to the Leon3. The best case overhead is one coprocessor instruction. Assuming the register operands are already in the coprocessor register file, after the coprocessor operation instruction, one coprocessor store instruction moves the result from the FLU to the Leon3.

Table 14: Case Study 1 Leon3 with FLU Implementation Summary

Execution Time in Clock Cycles	318
Maximum Operating Frequency (MHz)	125
Total logic elements	8152
Total combinational functions	7759
Dedicated logic registers	2646
Total memory bits	51200
Embedded Multipliers	6
Total PLLs	1

Table 15: Case Study 2 Leon3 with FLU Implementation Summary

Execution Time in Clock Cycles	576
Maximum Operating Frequency (MHz)	125
Total logic elements	9332
Total combinational functions	8759
Dedicated logic registers	3154
Total memory bits	51200
Embedded Multipliers	10
Total PLLs	1

7.7 Tensilica Xtensa LX2 Implementation

For this implementation platform, both case studies are developed in software for the Xtensa LX2 processor using Tensilica's Xplorer. Xplorer offers a custom C/C++ compiler for the Xtensa processor. The source code mainly consisted of if-then-else statements as well as multidimensional arrays. The Xtensa LX2 processor is at its default configuration shown in Appendix B. The source code was simulated on the Xtensa LX2 baseline processor.

Table 16: Case Study 1 Tensilica Xtensa LX2 Implementation Summary

Execution Time in Clock Cycles	1991
Maximum Operating Frequency (MHz)	330
Processor gate count	48000
TIE gate count	0

Table 17: Case Study 2 Tensilica Xtensa LX2 Implementation Summary

Execution Time in Clock Cycles	6765
Maximum Operating Frequency (MHz)	330
Processor gate count	48000
TIE gate count	0

7.8 Tensilica Xtensa LX2 with TIE Implementation

For this implementation platform, both case studies are designed using Tensilica Instruction Extension (TIE). TIE allows the Xtensa LX2 processor to be extended with additional custom hardware. In both case studies, TIE functions are developed that describe FL operations. Within a TIE operation, register and immediate operands define the inputs and outputs. The TIE operation can also support states, memory interfaces, and port/queue interfaces. TIE operations were manually designed for the fuzzification, inference rule, and defuzzification stages. In the fuzzification stage, simple if-then statements describe the membership function shapes. In the inference rule stage, again simple if-then statements describe the fuzzy inference rules. In the defuzzification stage, few arithmetic operations are developed in TIE and others were developed using specific TIE modules. These TIE modules are specified for easy use and provide a more efficient timing and area implementation. The Xtensa LX2 processor is at its default configuration shown in Appendix B.

Table 18: Case Study 1 Xtensa with TIE Implementation Summary

Execution Time in Clock Cycles	387
Maximum Operating Frequency (MHz)	330
Processor gate count	48000
TIE gate count	16442

Table 19: Case Study 2 Xtensa with TIE Implementation Summary

Execution Time in Clock Cycles	1108
Maximum Operating Frequency (MHz)	330
Processor gate count	48000
TIE gate count	51806

7.9 Discussion

The LEON3 implementation of the Inverted Pendulum Control System and Small Scale Food Preparation Control System case studies established a baseline of performance and area utilization for executing FL algorithms in software. The configuration of the Leon3 is the same for both case studies, but the execution time for each case study is different.

For case study 1, the FL algorithm required 1500 cycles to complete. For case study 2, the FL algorithm required 3158 cycles to complete. Case study 2 requires more cycles to complete since 2 FL algorithms are implemented. The Leon3 executed the FL algorithms sequentially which increased the execution time.

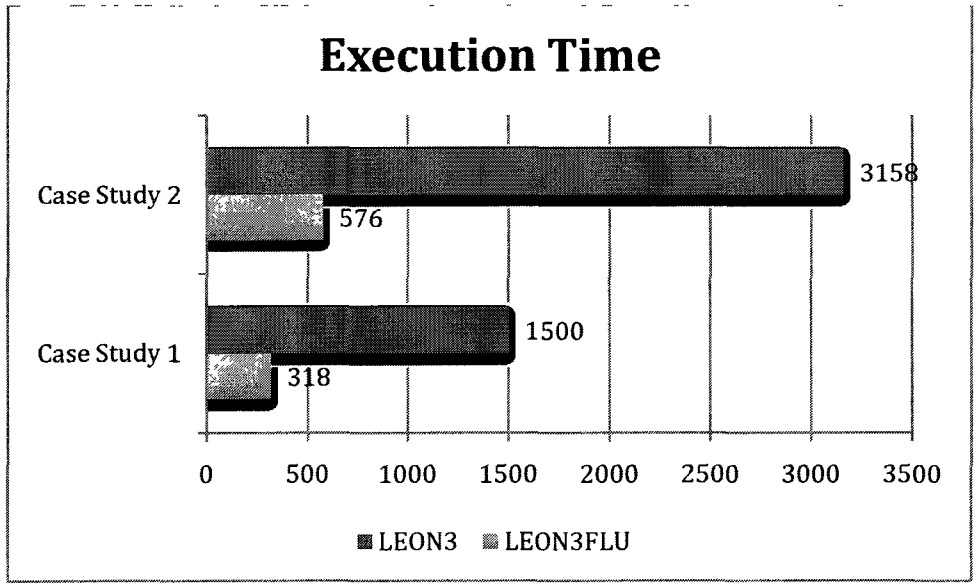

Figure 50: Comparison of Execution Time of Leon3 and Leon3 FLU implementations

The LEON3FLU implementation for both case studies provided a significant performance increase and a minor area increase by using the FLCU with FLDP. For case study 1, the execution time decreased from 1500 cycles to 318 cycles, a performance increase of 4.72 times greater than the Leon3 implementation. For case study 2, the execution decreased from 3158 cycles to 576 cycles, a performance increase of 5.48 times greater than the Leon3 implementation. The FLU addition to the Leon3 greatly improved performance by executing FL algorithms in hardware rather than software. For case study 2, both FL

algorithms run in parallel to further improve the performance over the Leon3 implementation.

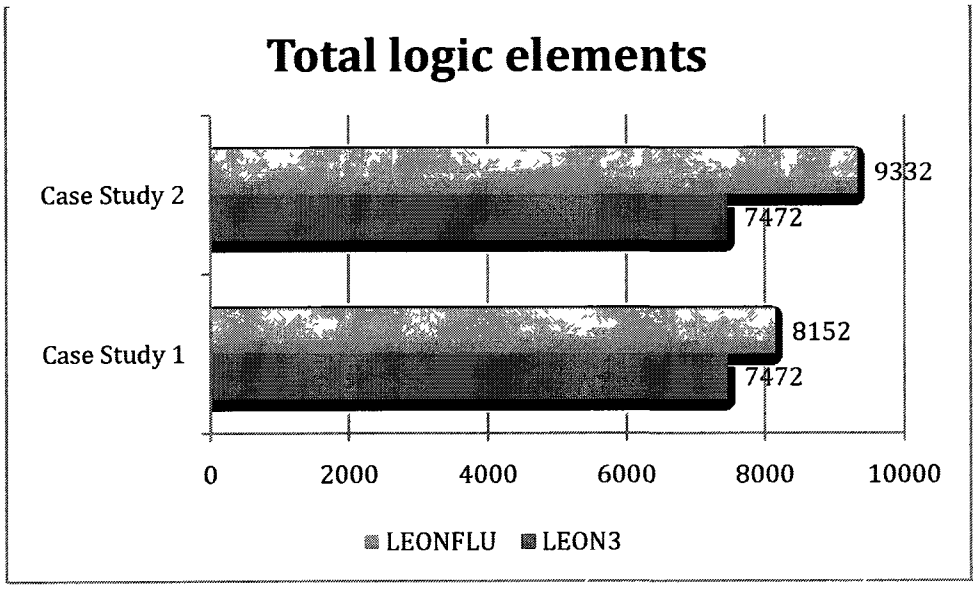

Figure 51: *Comparison of the Total Logic Elements of Leon3 and Leon3 with FLU implementations*

For both case studies, the total logic elements, total combinational elements, and dedicated logic register area metrics all increased when using the FLU with the Leon3 processor. For case study 1, while the FLU there is an increase of total logic elements of 1.12 times in comparison with the Leon3 implementation. Similarly for case study 2, there is an increase of 1.3 times in comparison with the Leon3 implementation.

Analyzing the Tensilica implementations, there is a similar trend when implementing the FL algorithm in hardware. For case study 1, using TIE produced a performance increase 5.1 times greater than the baseline Xtensa LX2 processor. For case study 2, TIE produced a performance increase 4.3 times greater than the baseline Xtensa LX2 processor. The use of TIE operations with TIE modules designed specifically for efficient area and timing yielded a significant performance increase.

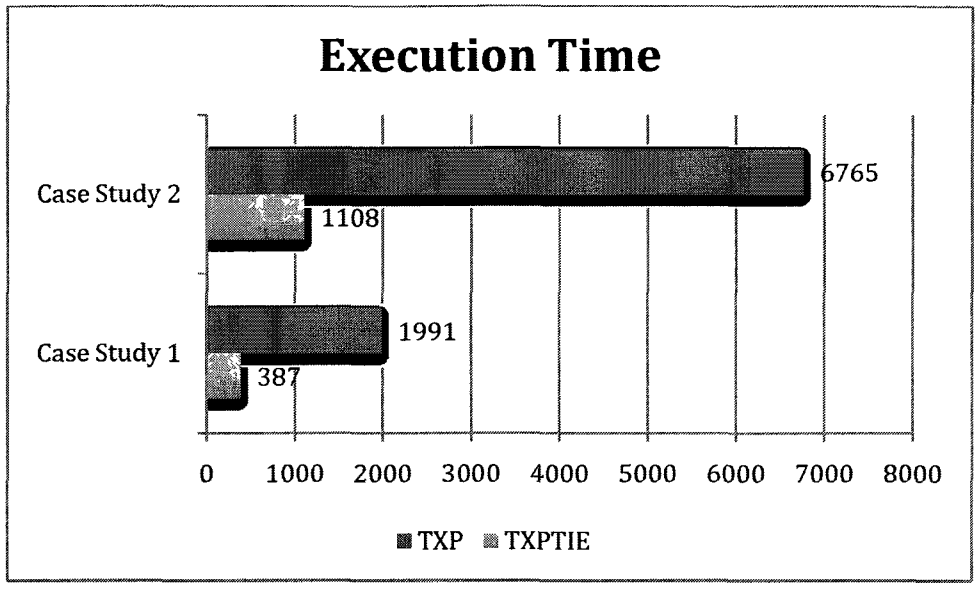

Figure 52: Comparison of the Execution Time of Xtensa and Xtensa with TIE implementations

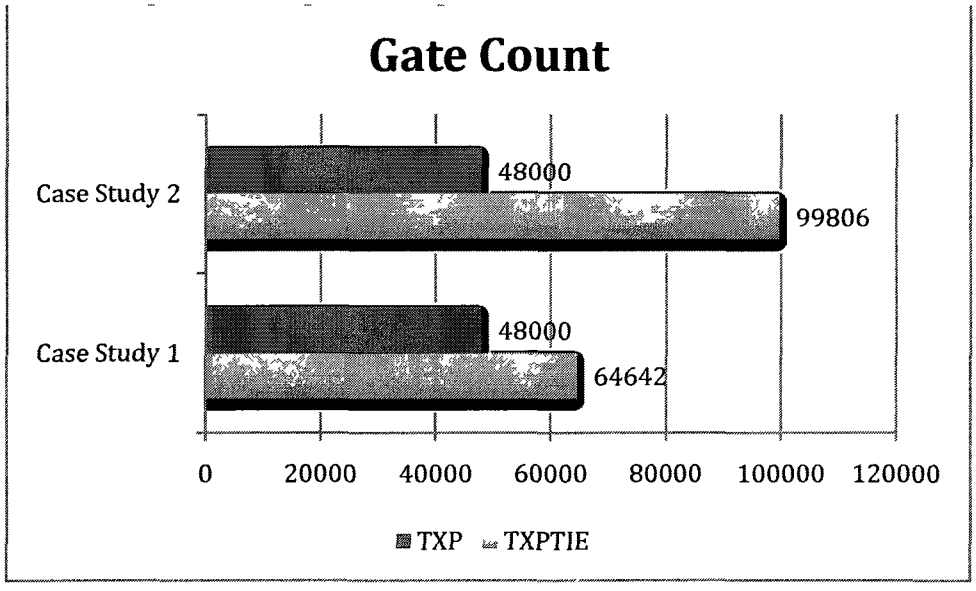

Figure 53: Comparison of Gate Count of Xtensa with TIE implementation

For both case studies, there was a significant increase in gate count. The baseline Xtensa LX2 processor has a gate count of 48000 gates. For case study 1, the FL TIE added 16442 gates to the design. For case study 2, the FL TIE added 51806 gates to the design. For case study 2 there is a greater gate count increase since two FL algorithms are implemented using TIE.

Leon3 and Xtensa based implementations both present a significant performance increase and area increase when implementing FL in hardware. However, there are key differences between the two implementations. Using the FLU, the datapaths are easily configured from the FIS model of the FL algorithms and then integrated with the Leon3 processor. Using TIE for the Xtensa LX2 processor required developing FL specific TIE operations. The TIE operations are developed independently of the FIS models for the FL algorithms. This can introduce inconsistencies between the FIS model and the FL hardware.

7.10 Hardware Comparison

Using the comparison criteria defined in chapter 3, the FLU can be compared against other reviewed architectures from Chapter 3.

Table 20: FLU Comparison with other reviewed architectures

Features	Leon3 with FLU	Hung [9]	Vasantha Rani et al. [10]	Deliparaschos et al. [11]	Singh et al. [12]	Ascia et al. [13]	Gabrielli et al. [14]	Chowdhury et al [43]
Inputs	2	2	2 / 3	2	2	8	4	16
Outputs	1	1	1	1	1	1	1	16
Input Resolution	32 bits	4 bits	4 bits	8 bits	8 bits	8 bits	7 bits	16 bits
Output Resolution	32 bits	8 bits	8 bits	12 bits	8 bits	8 bits	7 bits	16 bits
Membership Function Shapes	7	3	7	7	7	7	128	64
Membership Function Shape Types	Trapezoid / Triangular	Triangular	Trapezoid / Triangular	Trapezoid / Triangular	Triangular	N/A	Trapezoid	Gaussian
Fuzzy Operators	Intersection, Union, Complement	Intersection	Intersection	Intersection	Intersection	N/A	N/A	Intersection
Inference Rules Supported	Unlimited	9	49 - 343	49	36	256	2401	256
Implication Operator	Intersection, Union	Intersection	Intersection	Intersection	Union	Intersection	Intersection	Intersection
Defuzzification Method	Centroid	TVFI	Centroid	Centroid	Centroid	Yager	Centroid	Centroid
Fuzzy Inference System	Mamdani	Mamdani	Mamdani	Sugeno	Mamdani	Mamdani	Sugeno	Mamdani
Configurability	Yes	Yes	No	No	No	No	No	No
Processor Support	Yes	Yes	No	No	No	No	No	No
Modeling	Yes	No	No	No	No	No	No	No
Target Platform	Altera Cyclone 3 FPGA	Xilinx FPGA	Xilinx Spartan FPGA	Xilinx Spartan 3 FPGA	Xilinx Virtex FPGA	ASIC HCMOS 0.5um	ASIC CMOS 0 7um	Altera Cyclone FPGA
Maximum Operating Frequency	125MHz	10MHz	43 75MHz	200MHz	21.86MHz	60MHz	50MHz	40MHz
Throughput (million/second)	6.94	3.30	1 40	7.69	1.45	2 50	50.00	5.00

Overall, the FLU offers a configurable architecture that can support basic membership functions, configurability from Matlab FIS model, and integration with the Leon3 processor. It has a comparable throughput to compute fuzzy inferences as dedicated hardware controllers and processors.

The FLU design offers greater flexibility and configurability and comparable throughput against the other reviewed FPGA based FL architectures. The FLU configuration is based upon FL modelling using Matlab's Fuzzy Logic Toolbox. This allows developers to model the FL algorithm before implementing it in hardware. Modelling FL algorithms also maintains consistency when the FL algorithm is implemented in hardware. The FLU throughput is comparable but dependent on the FPGA architecture and maximum operating frequency. The advantage of the FLU is direct FL modelling and FLU configuration from the FL model. The FLU design also offers greater flexibility and configurability against other reviewed ASIC based FL architectures. Traditionally, ASIC architectures are fixed designs that offer no configurability. However, before synthesis ASIC architecture can be configured.

Chapter 8 - Conclusion

Taking into account the increasing popularity of FL for embedded control systems, it would be advantageous to have a design methodology where a FL algorithm model can generate the hardware model equivalent. Then the hardware model is integrated with a processor and ready to be integrated in an embedded system. FPGAs are emerging to be excellent platforms for prototyping designs and implementing control based applications. The FPGA design flow also allows for greater flexibility and configurability of hardware designs.

This thesis presents a configurable coprocessor architecture customized for FL algorithms implemented in hardware. The hardware is configured using FL models from Matlab's Fuzzy Logic Toolbox using translation and extractions algorithms. Then software scripts package and integrate the FLU with the Leon3 processor. The entire design is then compiled and synthesized using automatic synthesis scripts targeted for Altera FPGA devices. The novel FLDP and FLCU datapath design allows easy configuration for different FL algorithms from a Matlab FIS model. The translation and extraction algorithms extract FL parameters from the FIS model that can easily configure lookup tables in the FLCU and FLDP components.

The proof of concept architecture helped identify the challenges in implementing FL in hardware. The challenges of meeting a high frequency, configurability, and integration with a host processor highlighted the drawbacks of implementing FL in hardware. Others overcame these challenges by providing no configuration of the datapath and no processor support. However, the configurable coprocessor architecture addressed all those challenges as well as reducing the design time by the use of modelling and automation. The architecture defined a datapath for implementing FL in hardware with configuration. The architecture also simplified the integration with a host processor by using coprocessor instructions already used by the Leon3 processor.

The performance of the configurable FLU architecture is compared against traditional general purpose implementations in order to demonstrate the advantages of implementing FL in hardware rather than software. The final results shows promise that this architecture can be pursued further in order to provide easier control system design and integration within an embedded system.

Evaluating the different implementation platforms highlighted the advantages of implementing FL in hardware. Using the FLU, the datapaths are easily configured from the FIS model of the FL algorithms and then integrated with the Leon3 processor. However, to use the Xtensa LX2 processor requires developing FL specific TIE operations manually. This can lead to inconsistencies between the FIS model and the FL hardware.

While the presented architecture and design flow accomplishes easier control system design with FL and integration within an embedded system, other avenues can be explored by future research in adapting other types of FL algorithms. Other types of membership function shapes can be considered such as S-shaped or Gaussian. Linear and rigid shapes like Triangular and Trapezoidal are easier to translate, but S-shaped and Gaussian shapes have nonlinear curves that are difficult to translate.

Handling data forwarding and hazards may also prove to be fruitful, as more FLDP units can be instantiated which can yield more complex control system behaviours. For instance, implementing many FL algorithms with dependences without data forwarding or handling data hazards can lead to waiting for one FL algorithm to finish before another FL algorithm starts. This impacts system performance and throughput. With data forwarding and hazards handled, performance and throughput can be increased.

References

[1] M Araki. PID Control - Encyclopedia of Life Support Systems. [Online]. http://www.eolss.net/ebooks/Sample%20Chapters/C18/E6-43-03-03.pdf

[2] B. Singh, R. Goyal, R. Kumar, and R.P.P. Singh, "Design and VLSI implementation of Fuzzy Logic," *International Journal of Computer and Network Security*, vol. 1, no. 3, pp. 77-81, December 2009.

[3] L.A. Zadeh, "Fuzzy logic: issues, contentions and perspectives," in *IEEE International Conference on Acoustics, Speech, and Signal Processing*, 1994.

[4] L.A. Zadeh, "Is there a need for fuzzy logic?," in *Annual Meeting of the North American Fuzzy Information Processing Society*, 2008, pp. 1-3.

[5] L.A. Zadeh, "From fuzzy logic to extended fuzzy logic - A first step," in *Annual Meeting of the North American Fuzzy Information Processing Society*, 2009, pp. 1-2.

[6] L. Hongli, D. Peiyong, C. Wenjian, and J. Lei, "Direct conversion of PID controller to fuzzy controller method for robustness," in *IEEE Conference on Industrial Electronics and Applications*, 2008, pp. 790-794.

[7] M.I.H. Nour, J. Ooi, and K.Y. Chan, "Fuzzy logic control vs. conventional PID control of an inverted pendulum robot," in *International Conference on Intelligent and Advanced Systems*, 2007, pp. 209-214.

[8] M.F. Rahmat and M.M. Ghazaly, "Performance comparison between PID and fuzzy logic controller in position control system of dc servomotor," Universiti Teknologi Malaysia, Technical Report 2006.

[9] D.L. Hung, "Dedicated digital fuzzy hardware," *IEEE Micro*, vol. 15, no. 4, pp. 31-39, August 1995.

[10] S.P.J. Vasantha Rani, P. Kanagasabapathy, and A. Sathish Kumar, "Digital Fuzzy Logic Controller using VHDL," in *Annual IEEE INDICON*, 2005, pp. 463-466.

[11] K.M. Deliparaschos, F.I. Nenedakis, and S.G. Tzafestas, "A fast digital fuzzy logic controller: FPGA design and implementation," in *IEEE Conference on Emerging Technologies and Factory Automation*, 2005, pp. 259-262.

[12] S. Singh and K.S. Rattan, "Implementation of a fuzzy logic controller on an FPGA using VHDL," in *International Conference of the North American Fuzzy Information Processing Society*, 2003, pp. 110-115.

[13] G. Ascia and V. Catania, "A dedicated parallel processor for fuzzy computation," in *EEE International Conference on Fuzzy Systems*, 1997, pp. 787-792.

[14] A. Gabrielli and E. Gandolfi, "A fast digital fuzzy processor," *IEEE Micro*, vol. 19, no. 1, pp. 68-79, January/February 1999.

[15] R.E. Gonzalez, "Xtensa: a configurable and extensible processor," *IEEE Micro*, vol. 20, no. 2, pp. 60-70, March/April 2000.

[16] J. Sanghavi and A. Wang, "Estimation of speed, area, and power of parameterizable, soft IP," in *Design Automation Conference*, 2001, pp. 31-34.

[17] G. Ezer, "Xtensa with user defined DSP coprocessor microarchitectures," in *International Conference on Computer Design*, 2000, pp. 335-342.

[18] S. Xu and H. Pollitt-Smith, "Optimization of HW/SW Co-Design: Relevance to Configurable Processor and FPGA Technology," in *Canadian Conference on Electrical and Computer Engineering*, 2007, pp. 1691-1696.

[19] S. Leibson, *Designing SOCs with Configured Cores*, Wayne Wolf, Ed. San Francisco, United States of America: Morgan Kaufmann, 2006.

[20] L.O. Nall and R.J. Hathaway, "Fuzzy Systems Toolbox, Fuzzy Logic Toolbox [Software Review]," *IEEE Transactions on Fuzzy Systems*, vol. 4, no. 1, pp. 82-85, February 1996.

[21] The MathWorks Inc. (2010) Fuzzy Logic Toolbox. [Online]. http://www.mathworks.com/access/helpdesk/help/pdf_doc/fuzzy/fuzzy.pdf

[22] I. A. Grout and K. Keane, "A Matlab to VHDL conversion toolbox for digital control," in *IFAC Symposium on Computer Aided Control Systems Design*, Salford, UK, 2000.

[23] M.A. Shanblatt and B. Foulds, "A Simulink-to-FPGA implementation tool for enhanced design flow," in *IEEE International Conference on Microelectronic Systems Education*, 2005, pp. 89-90.

[24] J. Gaisler, E. Catovic, M. Isomaki, K. Glembo, and S. Habinc. (2008, September) GRLIB IP Core User's Manual. [Online]. http://www.gaisler.com/products/grlib/grlib.pdf

[25] SPARC International Inc. (1992) The SPARC Architecture Manual Version 8. [Online]. www.sparc.org/standards/V8.pdf

[26] G.J. Klir, "Fuzzy logic," *IEEE Potentials*, vol. 14, no. 4, pp. 10-15, October/November 1995.

[27] B.H. Wang and G. Vachtsevanos, "Fuzzy logic control: a systematic design methodology," in *IEEE Conference on Decision and Control*, 1991, pp. 1219-1220.

[28] S.J. Chalmers, "A design methodology for fuzzy logic controllers," in *IEE Colloquium on Synthesis and Optimisation of Logic Systems*, 1994, pp. 10/1-10/4.

[29] H.X. Li and H.B. Gatland, "A new methodology for designing a fuzzy logic controller," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 25, no. 3, pp. 505-512, March 1995.

[30] J.M. Mendel, *Uncertain Rule-Based Fuzzy Logic Systems*, Wil Mara, Ed. Upper Saddle

River, United States of America: Prentice Hall, 2001.

[31] K.K. Parhi, *VLSI Signal Processing Systems, Design and Implementation*. New York: John Wiley & Sons, Inc., 1999.

[32] Lih-Gwo Jeng and Liang-Gee Chen, "Rate-optimal DSP synthesis by pipeline and minimum unfolding," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 2, no. 1, pp. 81-88, March 1994.

[33] L.E. Lucke and K.K. Parhi, "Data-flow transformations for critical path time reduction in high-level DSP synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, no. 7, pp. 1063-1068, July 1993.

[34] S. Penolazzi, L. Bolognino, and A. Hemani, "Energy and Performance Model of a SPARC Leon3 Processor," in *Euromicro Conference on Digital System Design, Architectures, Methods and Tools*, 2009, pp. 651-656.

[35] Z. Zhonghua, W. Wuchen, H. Ming, and H. Ligang, "A SoPC design based on LEON3 SoC platform," in *Asia Pacific Conference on Postgraduate Research in Microelectronics & Electronics*, 2009, pp. 400-403.

[36] E. Catovic. (2004) GRFPU - High Performance IEEE-754 Floating-Point Unit. [Online]. http://www.gaisler.com/doc/grfpu_wp.pdf

[37] S., Vranesic, Z. Brown, *Fundamentals of Digital Logic with VHDL Design*, 2nd ed. New York, United States of America: McGraw-Hill, 2004.

[38] K., Nissbrandt, K. Lindberg, "An evaluation of methods for FPGA implementation from a Matlab description," KTH Electrical Engineering, Stockholm, Master's Degree Project 2008.

[39] Celoxica, "DK Design Suite," Datasheet 2005.

[40] Altera Corporation. (2010, July) Quartus II Handbook. [Online]. http://www.altera.com/literature/hb/qts/quartusii_handbook.pdf

[41] M. White, M. Waller, G. Dunnett, and P. Lister, "FPGA design with Mentor and Altera CAD software," in *IEEE Colloquium on Software Support and CAD Techniques for FPGAs*, 1994, pp. 1-5.

[42] Altera Corporation. (2010, July) Cyclone III Device Handbook. [Online]. http://www.altera.com/literature/hb/cyc3/cyclone3_handbook.pdf

[43] S.R. Chowdhury and H. Saha, "A High-Performance FPGA-Based Fuzzy Processor Architecture for Medical Diagnosis," *IEEE Micro*, vol. 28, no. 5, pp. 38-52, September/October 2008.

[44] V. Thareja, M. Bolic, and V. Groza, "Design of a Fuzzy Logic Coprocessor using Handel-C," in *International Workshop on Soft Computing Applications*, 2007, pp. 83-88.

[45] P. Arato, T. Visegrady, and I. Jankovits, *High Level Synthesis of Pipelined Datapaths*. West Sussex, England: John Wiley & Sons, 2001.

[46] M. Hutton et al., "A Methodology for FPGA to Structured-ASIC Synthesis and Verification," in *Design, Automation and Test in Europe*, 2006, pp. 1-6.

[47] V. Thareja, M. Montcalm, and M. Bolic, "Configurable Fuzzy Logic coprocessor for small scale food preparation," in *International Workshop on Soft Computing Applications*, 2009, pp. 229-234.

[48] D.A. Patterson and J.L. Hennessy, *Computer Organization and Design*, 3rd ed. San Francisco, United States of America: Morgan Kaufmann, 2005.

[49] A. Mehrotra, L. van Ginneken, and Y. Trivedi, "Design flow and methodology for 50M gate ASIC," in *Design Automation Conference*, 2003, pp. 640- 647.

[50] W.J. Daily and A. Chang, "The role of custom design in ASIC chips," in *Design Automation Conference*, 2000, pp. 643-647.

Appendix A – Matlab FIS Models

Inverted Pendulum Control System FIS Model

```
[System]
Name='inverted_pendulum_model'
Type='mamdani'
NumInputs=2
NumOutputs=1
NumRules=26
AndMethod='min'
OrMethod='max'
ImpMethod='min'
AggMethod='max'
DefuzzMethod='centroid'
[Input1]
Name='pole_angle'
Range=[-15 15]
NumMFs=5
MF1='NB':'trapmf',[-15 -15 -10 -5]
MF2='NM':'trimf',[-10 -5 0]
MF3='Z':'trimf',[-5 0 5]
MF4='PM':'trimf',[0 5 10]
MF5='PB':'trapmf',[5 10 15 15]
[Input2]
Name='pole_angular_velocity'
Range=[-15 15]
NumMFs=5
MF1='NB':'trapmf',[-15 -15 -10 -5]
MF2='NM':'trimf',[-10 -5 0]
MF3='PB':'trapmf',[5 10 16.2 25.8]
MF4='Z':'trimf',[-5 0 5]
MF5='PM':'trimf',[0 5 10]
[Output1]
Name='force'
Range=[-15 15]
NumMFs=7
MF1='NB':'trimf',[-12 -10 -8]
MF2='NM':'trimf',[-8 -6 -4]
MF3='NS':'trimf',[-4 -2 0]
MF4='Z':'trimf',[-2 0 2]
MF5='PS':'trimf',[0 2 4]
MF6='PM':'trimf',[4 6 8]
MF7='PB':'trimf',[8 10 12]
[Rules]
5 3, 6 (1) : 1
5 5, 6 (1) : 1
4 3, 6 (1) : 1
4 5, 6 (1) : 1
5 4, 5 (1) : 1
4 4, 5 (1) : 1
5 1, 4 (1) : 1
5 2, 4 (1) : 1
4 1, 4 (1) : 1
4 2, 4 (1) : 1
1 3, 4 (1) : 1
1 5, 4 (1) : 1
2 3, 4 (1) : 1
2 5, 4 (1) : 1
1 4, 3 (1) : 1
2 4, 3 (1) : 1
1 1, 2 (1) : 1
1 2, 2 (1) : 1
2 1, 2 (1) : 1
2 2, 2 (1) : 1
3 3, 5 (1) : 1
3 5, 5 (1) : 1
3 1, 3 (1) : 1
3 2, 3 (1) : 1
5 3, 6 (1) : 1
1 0, 6 (1) : 1
```

Small Scale Food Preparation Control System FIS Model

```
[System]
Name='temponly_integer'
Type='mamdani'
NumInputs=2
NumOutputs=1
NumRules=25
AndMethod='min'
OrMethod='max'
ImpMethod='min'
AggMethod='max'
DefuzzMethod='centroid'
[Input1]
Name='Temperature'
Range=[425 575]
NumMFs=5
MF1='LL':'trapmf',[425 425 450 475]
MF2='HL':'trapmf',[525 550 575 575]
MF3='LS':'trimf',[450 475 500]
MF4='HS':'trimf',[500 525 550]
MF5='EX':'trimf',[475 500 525]
[Input2]
Name='TempDelta'
Range=[0 100]
NumMFs=5
MF1='DF':'trapmf',[0 0 10 33]
MF2='Z':'trimf',[33 50 66]
MF3='IF':'trapmf',[66 90 100 100]
MF4='DS':'trimf',[16 33 50]
MF5='IS':'trimf',[50 66 83]
[Output1]
Name='output1'
Range=[0 100]
NumMFs=5
MF1='O':'trimf',[0 10 20]
MF2='OM':'trimf',[15 30 45]
MF3='S':'trimf',[35 50 65]
MF4='OH':'trimf',[55 70 85]
MF5='OF':'trimf',[80 90 100]
[Rules]
1 1, 5 (1) : 1
1 4, 5 (1) : 1
1 2, 5 (1) : 1
1 5, 3 (1) : 1
1 3, 3 (1) : 1
3 1, 5 (1) : 1
3 4, 3 (1) : 1
3 2, 3 (1) : 1
3 5, 2 (1) : 1
3 3, 2 (1) : 1
5 1, 3 (1) : 1
5 4, 2 (1) : 1
5 2, 2 (1) : 1
5 5, 1 (1) : 1
5 3, 1 (1) : 1
4 1, 2 (1) : 1
4 4, 1 (1) : 1
4 2, 1 (1) : 1
4 5, 4 (1) : 1
4 3, 4 (1) : 1
2 1, 1 (1) : 1
2 4, 4 (1) : 1
2 2, 4 (1) : 1
2 5, 4 (1) : 1
2 3, 4 (1) : 1

[System]
Name='steakonly_integer'
Type='mamdani'
NumInputs=2
NumOutputs=1
NumRules=15
AndMethod='min'
OrMethod='max'
ImpMethod='min'
AggMethod='max'
DefuzzMethod='centroid'
[Input1]
```

```
Name='Thickness'
Range=[50 200]
NumMFs=5
MF1='3/4':'trapmf',[50 50 75 100]
MF2='1&3/4':'trapmf',[150 175 200 200]
MF3='1':'trimf',[75 100 125]
MF4='1&1/2':'trimf',[125 150 175]
MF5='1&1/4':'trimf',[100 125 150]
[Input2]
Name='Doneness'
Range=[0 100]
NumMFs=3
MF1='rare':'trapmf',[0 0 10 50]
MF2='medium':'trimf',[0 50 100]
MF3='well':'trapmf',[50 90 100 100]
[Output1]
Name='output1'
Range=[240 810]
NumMFs=20
MF1='240':'trimf',[210 240 270]
MF2='270':'trimf',[240 270 300]
MF3='300':'trimf',[270 300 330]
MF4='330':'trimf',[300 330 360]
MF5='360':'trimf',[330 360 390]
MF6='390':'trimf',[360 390 420]
MF7='420':'trimf',[390 420 450]
MF8='450':'trimf',[420 450 480]
MF9='480':'trimf',[450 480 510]
MF10='510':'trimf',[480 510 540]
MF11='540':'trimf',[510 540 570]
MF12='570':'trimf',[540 570 600]
MF13='600':'trimf',[570 600 630]
MF14='630':'trimf',[600 630 660]
MF15='660':'trimf',[630 660 690]
MF16='690':'trimf',[660 690 720]
MF17='720':'trimf',[690 720 750]
MF18='750':'trimf',[720 750 780]
MF19='780':'trimf',[750 780 810]
MF20='810':'trimf',[780 810 840]
[Rules]
1 1, 2 (1) : 1
1 2, 5 (1) : 1
1 3, 11 (1) : 1
3 3, 13 (1) : 1
3 2, 7 (1) : 1
3 1, 4 (1) : 1
5 1, 5 (1) : 1
5 2, 8 (1) : 1
5 3, 15 (1) : 1
4 1, 6 (1) : 1
4 2, 9 (1) : 1
4 3, 17 (1) : 1
2 1, 8 (1) : 1
2 2, 10 (1) : 1
2 3, 19 (1) : 1
```

Appendix B – Processor Configurations

Leon3 Processor Configuration

Configuration Parameter	Default
AHB master index	0
Target technology	0
Vendor library for regfile and cache RAMs	0
Enable Debug Support Unit interface	Disabled
Floating-point Unit	Disabled
Generate SPARC V8 MUL and DIV instructions	Disabled
Generator co-processor interface	Disabled
Generate SPARC V8e SMAC/UMAC instruction	Disabled
Number of watchpoints	0
Enable instruction cache	Enabled
Instruction cache replacement policy	Least recently used (LRU)
Number of instruction cache sets	1
Instruction cache line size in number of words	4
Size of each instruction cache set in kByte	1
Enable instruction cache line locking	Disabled
Data cache enable	Enabled
Data cache replacement policy	Least recently used (LRU)
Number of data cache sets	1
Data cache line size in number of words	4
Size of each data cache set in kByte	1
Enable data cache line locking	Disabled
Enable data cache snooping	Disabled
Enable local instruction RAM	Disabled
Local instruction RAM size in kB	1
8 MSB bits used to decode local instruction RAM	0x8E

area	
Enable local data RAM	Disabled
Local data RAM size in kB	1
8 MSB bits used to decode local data RAM area	0x8F
Enable memory management unit	Disabled
Number of instruction TLB entries	8
Number of data TLB entries	8
TLB type	Shared TLB with slow write
TLB replacement policy	Least recently used (LRU)
Load delay	2
Size of instruction trace buffer in kB	0
Power Down	Disabled
Enable single-vector trapping	Disabled
Default reset start address	0
Enable multi-processor support	Disabled
Enable scan test support	Disabled

Xtensa LX2 Processor Configuration

Xtensa ISA version	LX2.1
16-bit MAC with 40 bit Accumulator	no
MUL16	no
MUL32	no
32 bit integer divider	no
Floating Point (coprocessor id 0)	no
CLAMPS	no
NSA/NSAU	yes
MIN/MAX and MINU/MAXU	yes
SEXT	yes
Boolean Registers	no
Number of Coprocessors (NCP)	0
Enable Density Instructions	yes

Enable Processor ID	no
Zero-overhead loop instructions	yes
Synchronize instruction	no
Conditional store synchronize instruction	no
TIE arbitrary byte enables	no
Count of Load/Store units	1
Max instruction width (bytes)	3
L32R hardware support	Extended L32R
Pipeline length	5
Vectra LX DSP coprocessor instruction family	no
HiFi2 Audio Engine DSP coprocessor instruction family	no
Thread Pointer	no
Interrupts enabled ?	yes
Interrupt count	1
Int 0 type / priority level	ExtLevel / 1
High Priority Interrupts	yes
Interrupt Level count	2
Medium Level Interrupts	no
Timer count	no
Byte ordering (endianness)	Little Endian
Number of physical AR registers. 16 forces Call0 ABI	32
Miscellaneous Special Register count	0
Generate exception on unaligned load/store address	align
Enable Processor Interface (PIF)	yes
Write buffer entries	4
Enable PIF Write Responses	no
Prioritize Load Before Store	no
Width of Instruction Fetch Interface	32
Width of Data Memory/Cache interface	32

Width of PIF interface	32
Width of Interface to instruction cache	32
Instruction Cache size (Bytes) / Line size (Bytes)	1024 / 16
Associativity	Direct
Line Locking	no
Instruction Cache memory error	none
Data Cache (Bytes) / Line size (Bytes)	1024 / 16
Associativity	Direct
Write Back	no
Line Locking	no
Data Cache memory error	none
Debug	yes
Data address breakpoint registers	0
Instruction address breakpoint registers	0
Debug interrupt level	2
Trace port (address trace and pipeline status)	no
Add data trace	no
On-chip Debug(OCD)	yes
Use array of 4 Debug Instruction Registers (DIRs)	no
External Debug Interrupt	no
Full scan	no
Make latches transparent	no
Xtensa Exception Architecture	XEA2
MemoryProtection/MMU	Region Protection
System RAM start address / size	0x60000000 / 4M
System ROM start address / size	0x40000000 / 128K
Reset Vector start address / size	0x40000000 / 0x2e0
Kernel (Stacked) Exception Vector start address / size	0x6000019c / 0x1c
User (Program) Exception Vector start	0x600001bc / 0x1c

address / size	
Double Exception Vector start address / size	0x600001dc / 0x1c
Window Register Overflow Vector start address / size	0x60000000 / 0x178
Level 2 Interrupt Vector start address / size	0x6000017c / 0x1c
Relocatable Vectors	no
RTL description	verilog
Synthesis / P&R flow	Physical Synthesis, Route
Geometry / Process	130lv / Worst
Core Speed	332 MHz
Functional Unit Clock Gating	yes
Global Clock Gating	yes
Register file implementation block	Flip-flops
Reset All Flops	yes
Software Target Options	
Xtensa Tools should use Extended L32R	no
Software ABI	windowed
C Libraries	newlib
Wind River VxWorks compatibility (deprecated)	no
Generic RTOS compatibility	no
Xtensa Model for Seamless compatibility	no
Target Linux compatibility	no