IMPLEMENTATION OF GENETIC ALGORITHM BASED FUZZY LOGIC CONTROLLER WITH AUTOMATIC RULE EXTRACTION IN FPGA

Under The Guidance of

Prof. S.K. Patra

Department of Electronics & Communication Engineering

National Institute of Technology, Rourkela

Pushpak Pati

109EI0088

Electronics & Instrumentation Engineering

Jyotiprakash Sahoo

109EC0236

Electronics & Communication Engineering

National Institute of Technology, Rourkela

CERTIFICATE

This is to certify that the thesis entitled, "**IMPLEMENTATION OF GENETIC ALGORITHM BASED FUZZY LOGIC CONTROLLER WITH AUTOMATIC RULE EXTRACTION IN FPGA**" submitted by **PUSHPAK PATI** (109EI0088) and **JYOTIPRAKASH SAHOO** (109EC0236) in partial fulfillment of the requirements for the award of Bachelor of Technology degree in **Electronics and Instrumentation Engineering** and **Electronics and Communication Engineering** respectively during the session 2009-2013 at National Institute of Technology, Rourkela and is an authentic work done by them under my supervision and guidance.

To the best of my knowledge, the matter embodied in the thesis has not been submitted to any other university/institute for the award of any Degree/Diploma.

Date:

Professor Sarat Kumar Patra

Department of Electronics and Communication

National Institute of Technology, Rourkela-769008

Acknowledgement

We are grateful to our respected supervisor Prof. Sarat Kumar Patra for his ubiquitous support and meticulous guidance throughout the duration of the project. We would also like to express our heartfelt appreciation and unfathomable note of thankfulness to Mr. J.Bhaskar Rao and Mr. Pallav Majhi for their altruism and guidance that helped us realize our goals. Lastly we would convey our gratitude to all the faculty members and staff of the Department of Electronics and Communication, National Institute of Technology, Rourkela.

Pushpak Pati

109EI0088

Jyotiprakash Sahoo

109EC0236

Abstract

A number of fuzzy logic controllers are being designed till now to replace complex, non-linear and huge controlling equipment in numerous industrial sectors. But the designing of these controllers requires thorough knowledge about the controlled process. For this purpose a highly experienced experts are required, which is not feasible all the time. Most of these processes are non-linear and depend on large number of parameters. Thus mathematical representation of these systems is an arduous line of work. Even in the fuzzy design methodologies popular today the number of rules increases non-linearly with number of features adding to the computational burden.

This project addresses these problems by proposing using of genetic algorithm based Fuzzy Logic systems as controllers. The system includes algorithms which are run on a capable computing platform, to read an experimental data sheet obtained from experimental observations of the system and generate a fine tuned rule base that is to be used in the fuzzy logic controller hardware. The hardware is implemented in an FPGA. Transfer of synthesized rule base from the computer to the FPGA implementation and crisp output value back to the computer is done by UART. A graphical user interface is provided that runs on the computer.

List of figures

Contents

1. Introduction

Systems are designed to take certain inputs from the operator and do some operations on the taken inputs to provide desired output. It constitutes of detailed methods, procedures, routines that carry out a specific activity that are represented by the combination of different variables related through several mathematical relations and operations. The entire representation may constitute of simple or critical or both types of mathematical computations which are difficult to synthesize practically and may give numerous errors even with best design implementations. Classical theory fails to design the systems completely and efficiently because of the increased number of variables and conditions and has become less powerful with the requirement of multiple-input-multiple-output system. To avoid such kind of problems and to design the system with easy user interfaces when input-output data is given, different mechanisms can be followed like neural networks, regression, evolutionary algorithms, fuzzy logic etc. Systems are first trained with some known input-output results using any of the above methods or a combination of them and then tested on new input values. In this project the approach of fuzzy logic based system design is discussed.

Fuzzy logic can be used to solve problems like input-output problems, classification problem, mapping problems etc. This project has been designed to solve the input-output problem, i.e. the fuzzy system will be modeled approximating a system which takes some input from the user, does complex mathematical computations with it and provides some output, e.g. PID controllers. The actual system may have a number of problems and high level of complexity

during computation. The designed fuzzy system reduces the level of complexity with a number of advantages over the original system.

A fuzzy system consists of four basic blocks which are the fuzzifier, the inference engine, the knowledge base or rule base and the defuzzifier. Almost all the signals available in real world are analog in nature. To process these signals first they need to be sampled in time domain. The processing of these samples by a fuzzy logic based system involves feeding of these samples to the fuzzifier. It converts these analog samples in to appropriate fuzzy values and feeds them to the inference engine. Inference engine gives a conclusion (output) from the facts (input) and knowledge (control rules). The knowledge base contains all the rules to take the necessary decisions. The fuzzy outputs provided by the inference engine are defuzzified by the defuzzifier section in order to give final crisp output.

In this project a novel algorithm has been proposed for the fuzzy system designing and it has been implemented in an FPGA. Direct interaction between user interface (PC) and the FPGA have been done over serial communication through UART.

2. FLC versus traditional controllers

2.1. Shortcomings of PID controller

Most of the controllers used today in automation and control are PID based. PID controllers have been serving since a long time. PID controllers are applicable to many control problems, and often perform satisfactorily without any improvements or even tuning. However, they exhibit certain shortcomings.

1. Reduction in loop gains to avoid overshooting gives poor performance.
2. PID controllers owing to their linearity and symmetric behavior show variable performance in case of non-linear systems.
3. The derivative noise is affected greatly by noise. A slight change in measurement or process noise can imply large change in outputs.
4. The fundamental difficulty with PID control is that it is a feedback system, with constant parameters.
5. Designing methods of conventional PID-like controller have been using more and more advanced mathematical tools. This is needed in order to solve difficult problems in fashion. However this also results in fewer and fewer practical engineers who can understand these design tools.

2.2. Advantages of FLC

PID controllers can be approximated by Fuzzy logic controllers (FLC). Though FLCs are designed on the basis of approximation, but they can provide the necessary amount of accuracy just like a PID controller along with a number of other advantages, which are given below:

1. Fuzzy controllers have a very convenient user interface. Users without knowledge of control engineering can interpret the system as well.
2. Fuzzy logic provides a certain level of artificial intelligence to the conventional controllers, leading to the effective fuzzy controllers.
3. Process loops that can benefit from a non-linear control response are excellent candidates for fuzzy control.
4. Fuzzy logic provides fast response times with virtually no overshoot, loops with noisy process signals have better stability and tighter control when fuzzy logic control is applied.
5. Input and output ranges can be subdivided as per the performance requirements and can be given different treatment.
6. Control rules can be added to cover important interactions among variables.

Every control system can be incorporated by a FLC and fine-tuned to plant non-linearity due to universal approximation capabilities.

2.3. Advantages of GA based FLC over regular FLC

Generally the Fuzzy controllers are designed based on expert's knowledge. But experts may not be available at all circumstances and their results may not be the optimum. Also if the number of input features of each data point increases and if more rules are required then it is not possible on part of any expert to design a completely logical and accurate system.

In such a situation we need to design a system using another means, i.e. using the datasheet for the given situation. System or rule base designing using datasheet has been found to be more reliable in the circumstances given below,

- Where humans are relied to make the decision making process.
- Where enough prior knowledge is required to do the decision making.
- Where many conditions are to be checked to design the system which is not possible in the part of a person to remember and utilize properly.
- Where solution to the system must be made understandable to non-experts.

So a novel approach is used in this project in order to extract rules from datasheet and the rule base is tuned to meet the required accuracy level using genetic algorithm.

3. System architecture

The basic architecture of the project is given as follows:

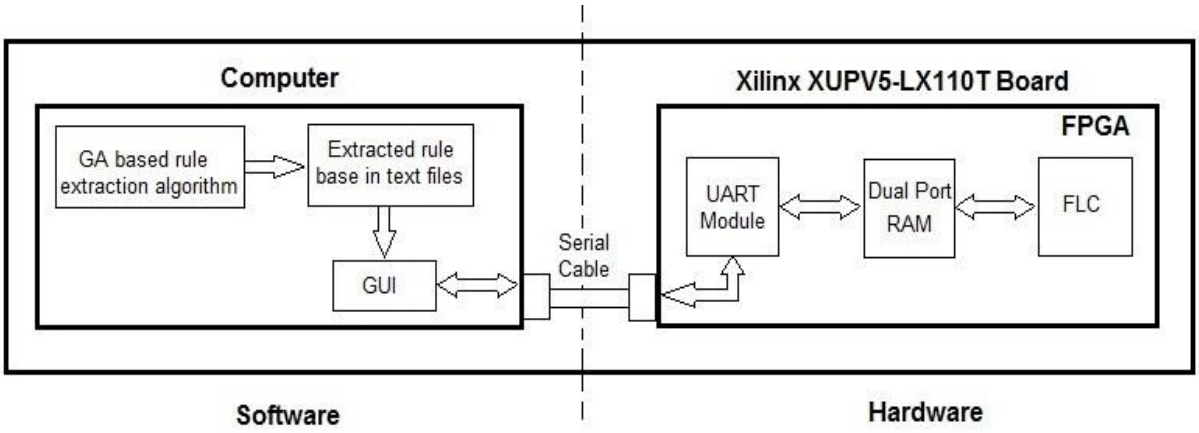


Figure 1 Architecture of the Fuzzy Logic Controller

The data sheet prepared on the experimental observations of the system is provided to the computer. It contains a list of parameters which are to be taken into consideration in the designing of the system. Corresponding output values for a given set of input parameters are provided. The program is made to read the data sheet and extract the initial rule base as per the algorithms. The rule base is then optimized using genetic algorithm. Accuracy of training depends on the distribution of data points in the data sheet over the range of application. The algorithms are implemented in C language and the rules are saved in text files.

Once the rule base is designed the parameters of the rules are sent through serial communication protocol like RS-232 from UART of the computer to the UART of the Xilinx board and saved in the dual port block RAM on the FPGA through its port A. This completes the update of the rule base at hardware level.

Then for testing purpose input parameters of a data point are taken through a GUI on the computer and sent serially to the Xilinx board. All the rules are read from port B of the dual port RAM, the data points are fired on the rule base and the output value is computed. This value is again transferred serially back to the computer for display on the GUI. The design of the GUI is done using MATLAB R2010a and the hardware is designed at RTL level using Verilog for HDL coding, iSim for simulation and Xilinx ISE for synthesis and implementation.

4. Design and implementation approach

As shown in the basic architecture section the project is divided in two sections, i.e. software section and hardware section. The software part is done in the PC for the designing and tuning of the rule base and the hardware section includes real time implementation of the FLC on the XUPV5LX110T board. FPGAs allow easy and effective testing of digital circuits. Verilog is used to model and implement the FLC in the FPGA. Using digital circuitry reduces the size of the system as well as reduces power consumption as compared to analog counterparts.

The designed rule base and required input values are transferred from the PC to the board is done via serial communication between the two using COM port of the PC and UART of the board.

4.1. Rule base extraction

The most important step in designing of a FLC involves the rule base. The designing can be done either using expert's knowledge or by using available experimental data sheet. The benefits of use of data sheet and the constraints of relying on expert's knowledge have been explained before. Thus this project is based on designing of rule base using data sheet. It is used for the training of the system.

Fuzzy rule bases can be designed from dataset using various algorithms. Fuzzy C-Means (FCM), Hard C-Means (HCM), K-Means algorithms etc. are most widely used algorithms for this purpose. In this paper we have used K-Means clustering algorithm for rule base designing. The number of clusters equals the number of rules in the rule base. After fixing the number of

rules as per our requirement the clustering is done to generate rules randomly out of the datasheet at each run.

Genetic algorithm is used to optimize the rule base. The parameters of genetic algorithm are defined as per the required accuracy level of the output. It also depends mostly on the capabilities and constraints of the hardware, doing the computations. Time efficiency, memory efficiency and hardware requirement for the computation purpose play the most vital role in defining the parameters.

The rule base extraction can be explained under two headings,

1. Initial designing
2. Optimization using Genetic Algorithm

4.1.1. Initial designing

Fuzzy rule bases can be designed from dataset using various algorithms. Fuzzy C-Means (FCM), Hard C-Means (HCM), K-Means algorithms etc. are the most widely used algorithms for this purpose. K-Means clustering algorithm was used for rule base designing in this project. The number of clusters equals the number of rules in the rule base. After fixing the number of rules as per the requirement the clustering was done to generate the rules randomly out of the datasheet at each run.

To increase the flexibility and degree of randomness in the design of the rule base, randomly 80 per cent of the data were used for the rule base designing. Out of ‘n’ data points in the datasheet 80 per cent of the data were chosen to form the new datasheet. For designing of ‘c’ number of rules ‘c’ number of data points were randomly chosen and they were modified to design the initial rule base.

Rules were designed for the antecedent and consequent data. After the selection of 'c' number of data points clustering was done. The Euclidian distance of all the data points in the actual datasheet were calculated from each of the 'c' rules. A data points belongs to i^{th} cluster if its distance from the i^{th} rule is minimum as compared to all other distances calculated from other rules in the rule base. Thus all the data points in the data sheet were distributed in to 'c' number of clusters.

Suppose, initially there are 'n' number of data points. 80 percent of the data points are chosen to form the new data sheet. Let there be 'm' number of each data point and a consequent depending upon the antecedent. We design 'c' numbers of clusters to form 'c' number of rules. Initially 'c' number of points was chosen randomly. Then the Euclidian distance of a data point from each of the rules is calculated by the formula given below,

$$d_i = \sqrt{\sum_{k=1}^{m} (n_k - v_{ik})^2}$$

where, d_i =distance of a point 'n' from the i^{th} rule

k = feature number, $k = 1,2,....m$

i = rule number, $i = 1,2,.....c$

The data point belonged to the cluster 'i' for which 'd_i' was minimum among all distances. After clustering was done, the initial cluster centers were modified. The feature wise mean of all data points were calculated for each cluster. These values became the new cluster centers. If the newly formed cluster centers differed from the old cluster centers by a value greater than some threshold, then again clustering was done with the new cluster centers. This process continues till the difference between the new cluster centers and old

cluster centers came below the threshold value. This process was used to generate the cluster centers.

After final clustering, in each cluster the minimum and maximum values of the data points were found feature wise. These values were used to design the initial rule base. In this project triangular rules were designed for each feature. For i^{th} triangle in j^{th} feature the center equaled the corresponding i^{th} cluster center for j^{th} feature and the lower and upper bounds of the triangle equaled the minimum and maximum values of the data points in i^{th} cluster and j^{th} feature.

4.1.2. Genetic Algorithm for Modification of Rule Base

Based on the initial designed rule base, the output values were calculated for all the data points in the training data sheet and sum of all squared errors was calculated. The total error was observed to be very high. Hence the initial designed rule base has to be modified. This modification can be done using any of the evolutionary algorithms. In this paper real coded genetic algorithm has been used for the modification. A new algorithm has been designed to deal with this situation where, the antecedents and the consequent all were member functions. The proposed algorithm has been used for triangular membership functions but it can also be extended to any other type of membership function. This algorithm contains basic steps of genetic algorithm (GA), i.e. Selection, Crossover, Mutation and Elitism.

I. Selection

For selection population size has to be chosen as per the design requirements. As per the

algorithm that has been used in this paper the population size has to be even. The clustering algorithm and initial rule base designing method described in section 1 has to be used to generate initial rule bases for each population. Thus each population corresponds to one rule base for which total squared error can be calculated for all the data points in the training data sheet. Here the objective function was to reduce the total squared error.

II. Crossover

Total squared errors were then sorted in ascending order; hence the 1st one corresponds to the best rule base, i.e. rule base with minimum squared error. Roulette Wheel technique was used to select the populations for crossover purpose. Let all the squared errors be stored in an array error [population]. The following algorithm was used to design the roulette wheel.

$$total\_error = \sum_{i=1}^{population} error[i]$$

$$error\_new[i] = total\_error - error[i]$$

$$error\_new[i] = \frac{error\_new[i]}{total\_error}$$

$error\_new[population]$ contains the values for roulette wheel designing. The rule base with minimum squared error takes maximum area in the roulette wheel.

The purpose of crossover was to generate new children from their parents. In this case both the parents were antecedent s of rule base and the crossed-over children were also rule bases. A crossover probability was chosen as a design parameter. A number was chosen

randomly between 0 and 1, if its value was less than crossover probability then crossover will be done, else no crossover will take place. The concept of crossover was best parent give best children. Total number of crossovers will be population size/2. During each crossover one parent was from first half of the populations. The corresponding parent with whom crossover will happen was chosen randomly from roulette wheel.

After each crossover two new children were generated. If no crossover was done between the two parents then parents were transferred to their children. Thus the total number of children generated after crossover equals the population size. The technique used for crossover was given below.

Crossover was done between two populations i.e. between two rule bases. For triangular membership functions only the peaks of the triangles which construct the rule base, take part in the crossover process. Suppose the two triangles which take part in crossover have the base points given by $[a_1, m_1, b_1]$ and $[a_2, m_2, b_2]$. Then

$$d_1 = \frac{m_1 - a_1}{b_1 - a_1} \text{ and } d_2 = \frac{m_2 - a_2}{b_2 - a_2}$$

These d_1 and d_2 values were then interchanged between the 2 triangles, i.e. the new value of

$$m_1 = a_1 + \frac{m_2 - a_2}{b_2 - a_2} \times (b_1 - a_1) \text{ and } m_2 = a_2 + \frac{m_1 - a_1}{b_1 - a_1} \times (b_2 - a_2)$$

The end points of the triangle remain unchanged.

III. Mutation

The newly generated triangles were again modified slightly expecting for a better rule base. A

mutation probability and a mutation fraction were set by the designer as a design parameter. Maximum number of points that can be mutated was equal to mutation probability times the total number of points. In our case the triangle centers and the boundary points may get mutated to give a better rule base. The above algorithm includes that if the center point of a triangle gets mutated then the boundary points of the triangle will also get mutated to give a new triangle.

A number was chosen randomly in between 0 to 1. If that number was below the mutation probability then mutation will take place, otherwise no mutation will occur. The mutation of a triangle described by the points $[a, m, b]$ was done as given below.

Minimum distance was calculated, i.e. $d = \min(m-a, b-m)$. Then a new value of 'm' was assumed in the neighborhood of 'm' within a region given,

$[m - d \times mutation\_factor, m + d \times mutation\_factor]$.

Similarly the boundary points of the triangle also get mutated. 'a' was assumed within a region given by, $[a - (m-a) \times mutation\_factor, a + (m-a) \times mutation\_factor]$ and 'b' was assumed within a region given by

$[b - (m-b) \times mutation\_factor, b + (m-b) \times mutation\_factor]$. The value of '$m$' used in the above boundary conditions was actually the old value of 'm' before getting mutated. Thus a complete new triangle was generated after mutation.

IV. Elitism

The total squared errors for all the populations, i.e. for all the rule bases generated before crossover, after crossover and after mutation were calculated. All the rule base triangles

in a rule base and corresponding total squared errors were stored in a mating pool. Now new population was designed for next generation. For this purpose a number populations were chosen out of the mating pool, equal to the total population size, based upon minimum total squared error. These new populations were again modified through the above procedure in number of generations in order to give an optimum rule base with minimum total squared error.

This generation process terminates if,

- The number of generations set by the designer were over or
- The total squared error comes down below the required error level or
- The total squared error does not get modified over a number of generations, i.e. No improvement was seen in the rule base over a number of generations.

Final rule base designing

The triangular rules were designed using the above GA method. The centers of the triangles were the finally obtained values. Thus the entire rule base designing was completed.

4.1.3. Flow diagram of GA based optimization

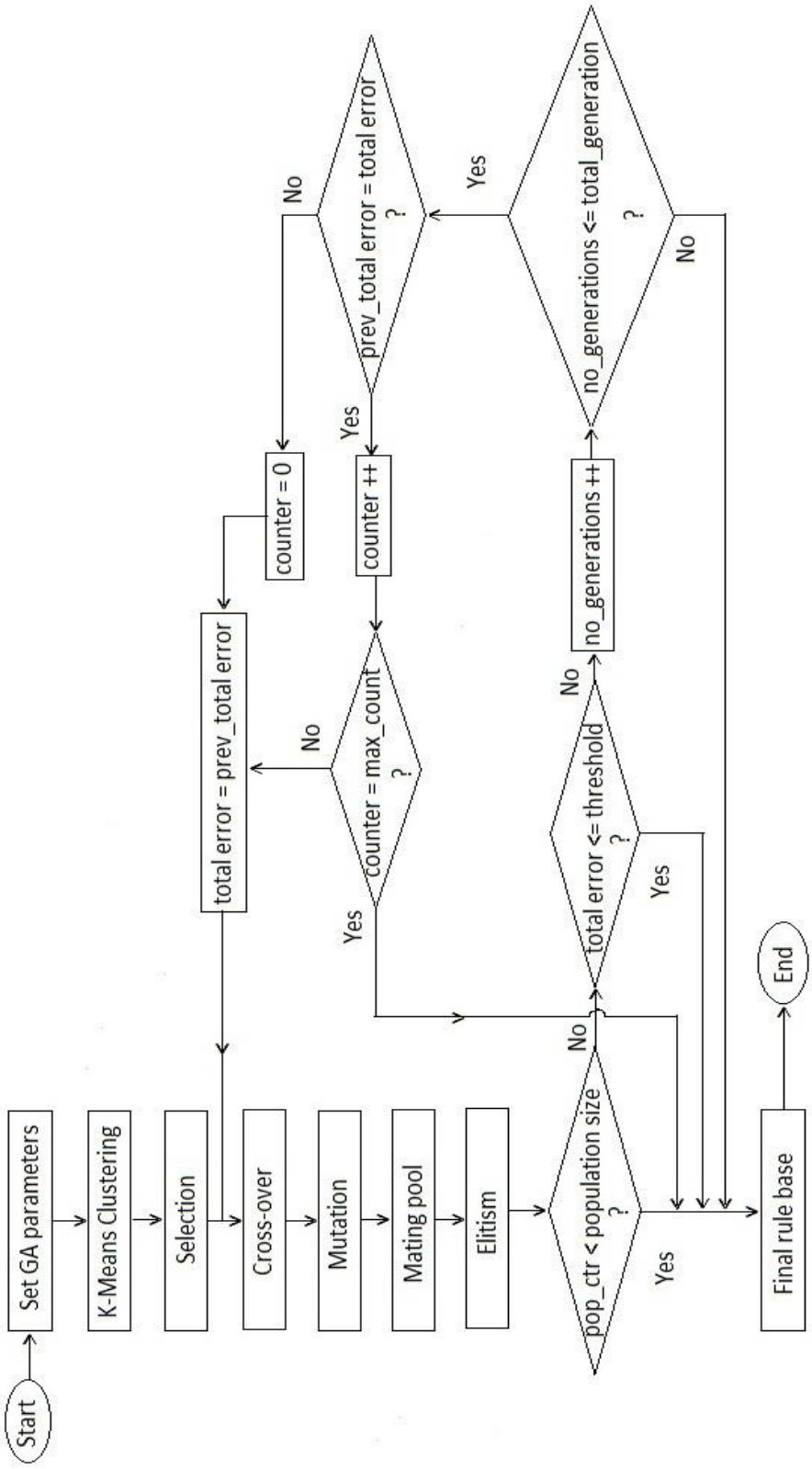


Figure 2 Flow diagram showing the GA based optimization of the rule base

4.2. Rule Transmission

After the optimized rule base was designed, all the rules are saved in text files. A GUI was designed in MATLAB which reads the rule base from the text files and sends it through serial communication over the UART to the FPGA.

In the GUI the rule files were browsed and uploaded, the crisp input values were given in the provided space. Then the 'Compute' button in the GUI was clicked to begin the transmission.

To send the floating point values some formatting was used, as explained below.

1. The data to be sent was multiplied by 256 i.e. 8 bit shift in binary value.
2. The obtained result was rounded to nearest integer value.
3. This value was represented in 16 bit format and sent over the UART in chunks of bytes.
4. 1st high byte was sent then low byte was sent afterwards.

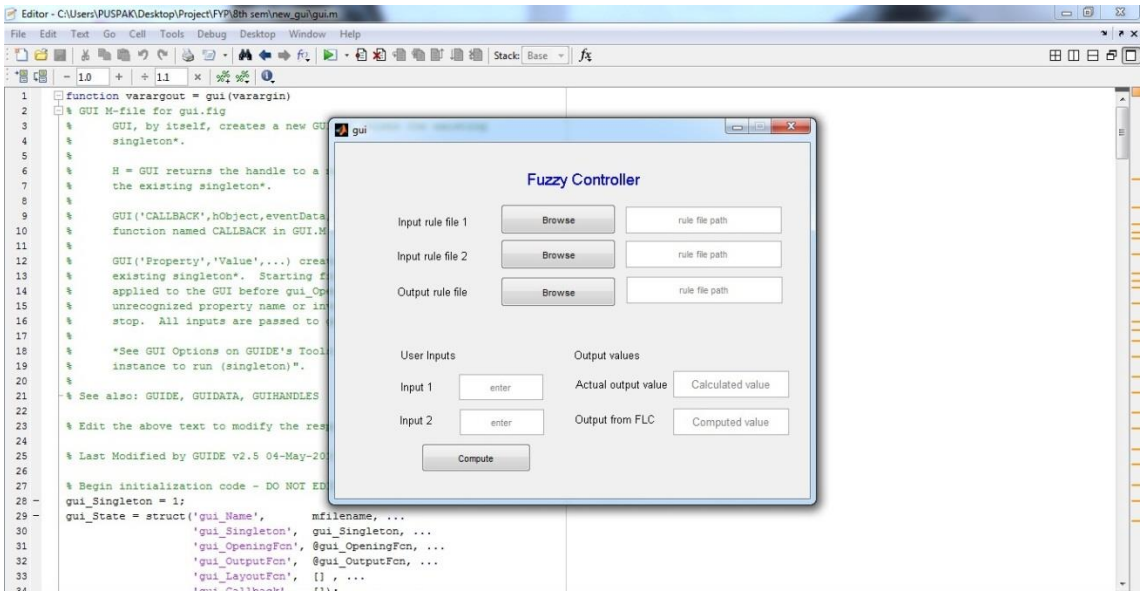


Figure 3 Designed GUI using MATLAB

The transmission of data points begins from the reverse direction, i.e. 1^{st} the upper value of the triangle of the *final* rule for the output was sent. Then center value of the final rule followed by lower value of the corresponding triangle was sent. Once the final rule for the output is sent over the UART, then the final rule for the nth input was sent followed by the final rule for the $(n-1)^{th}$ input. This sequence was followed till the final rule for input1 was sent.

Then $(final-1)^{th}$ rule was sent in the same fashion. This process continues till all the rule points were sent. After rule base was sent successfully, high byte of n^{th} crisp input was sent, then its low byte. All the crisp input values were sent one by one after this in decreasing manner.

Finally the GUI sent control word '0x03' on the UART, indicating the availability of the crisp input values. The UART saved all the received bytes in registers as explained in the next section.

$$\begin{aligned} &total\ number\ of\ points \\ &\quad = no.of\ rules \\ &\quad \times\ no.of\ points\ required\ to\ represent\ each\ membership\ function \\ &\quad \times (no.of\ inputs + no.of\ outputs) \end{aligned}$$

$$total\ number\ of\ bytes\ to\ be\ transfered = total\ no.of\ points \times 2$$

For examples, let us design a system for 2 inputs and 1 output. Triangular membership functions are used to design 8 rules in the rule base both for input and output side. 3 points are required to represent each triangle.

Total number of data points to be transferred=8×3× (2+1) = 72.

Total number of bytes to be transferred=72×2= 144

After 144 bytes have been captured, the FLC treats the next pair of bytes as inputs. After the membership function points are saved in the RAM, a rule control word is sent with value 0x03 to indicate the hardware that building process is completed.

4.3. Hardware architecture

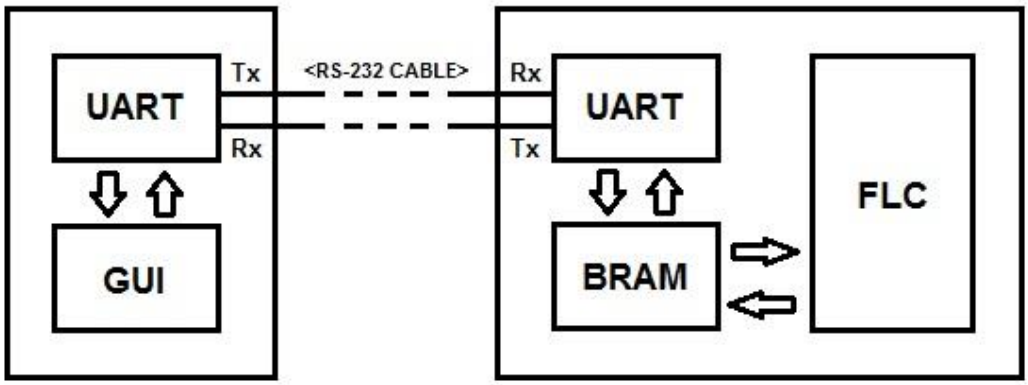


Figure 4 Architecture of the system

The design goal was to enable extraction of synthesized rule base from a more capable hardware platform onto a dedicated implementation platform for the fuzzy system. The interface was chosen as UART due to low speed and simplicity which shall reduce debugging time. The choice of components for the architecture was dictated by terms imposed by the HDL as well as the synthesis software. A word size of 16 bits was considered as the numbers were represented in Q8.8 format. Rule base representation was addressed following the triangular nature of membership functions. As triangular functions are used, a set of three points is required to represent each function. The software transmits these points as a set of two bytes

over UART. The membership function points of the rules are stored in RAM locations after the UART on FPGA has captured the bytes. Following the rule base extraction, the crisp input values are sent to hardware and are also stored in RAM locations. Once new crisp inputs are transferred the software writes '0x03' to RAM location 0x94, which is the control word. The FLC keeps polling this location for the value '0x03'. Upon reading '0x03' the FLC starts with writing '0x01' back to the control word location to acknowledge. It proceeds with reading the crisp inputs, reads membership function points rule wise and computes the crisp output value. This value requires 35 bits for storage and hence is stored at five byte wide RAM locations. The control then signals the UART to transmit these bytes to the software. The hardware then waits for another cycle of operation.

A dual-port RAM is used with its ports A and B interfaced to the UART and the FLC modules respectively. The Xilinx Core Generator tool was used to generate the core for the block RAM module. The functionality of dual ports purges bus sharing issues. The address on port A is set to 0x00 and increments twice for each membership point or crisp input (2 bytes). Arrival of data over UART triggers address increment and memory write operation.

CODE :

```
always @ (posedge clk or posedge rst)
begin
    if (rst)
       UartWea <= 'h0;
    else
    begin
       if (new_rx_data_bd)
            UartWea <= 'h1;
        else
            UartWea <= 'h0;
    end
```

```
end

always @ (posedge clk or posedge rst)
begin
    if (rst)
    begin
       UartAddr <= 'h0;
    end
    else
    begin
       if (UartWea & (UartAddr <= 'h93))
            UartAddr <= UartAddr + 1;
       .
       .
       .
    end
end
always @ (posedge clk or posedge rst)
begin
    if (rst)
    begin
       UartDataIn <= 'h0;
    end
    else
    begin
       if (new_rx_data_bd)
            UartDataIn <= rx_data_bd;
    end
end
```

The value in the UART receiving buffer is written to the current RAM location over the data bus of port A. All the membership function points (144 in number) occupy RAM locations 0x00 to 0x8f. The crisp inputs are stored at locations 0x90, 0x91 and 0x92, 0x93. The address stops incrementing after 0x94 where the control word (1 byte) is stored.

The software writes ‘0x03’ to the control word location after transmitting the membership function points and crisp inputs. This denotes that the rule base is updated and new set of crisp inputs are available. This location is polled by hardware to check for the 0^{th} and 1^{st} bits.

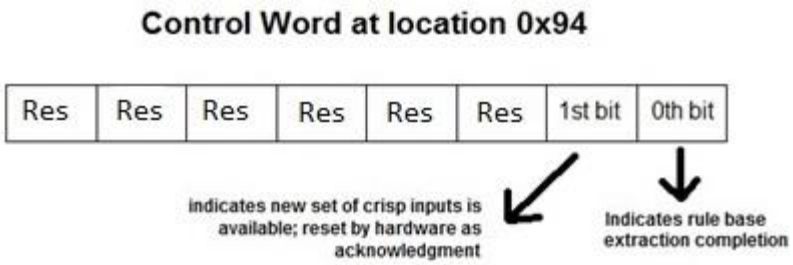


Figure 5 Control word at location 0x94

If both these bits are high the FLC operation is triggered. The hardware uses port B for data transfer with FLC as well as the update the control word. It writes ‘0x01’ at the location of control word. Control word value of ‘0x01’ denotes that rule base extraction is done and the hardware has read the crisp inputs. It serves as an acknowledge action from the hardware.

CODE :

```
always @ (posedge clk or posedge rst)
begin
    if (rst)
    begin
        RuleEn <= 1'b0;
       InputEn <= 1'b0;
    end
    else
    begin
        if ((FuzAddrReg == 'h94 ) && (|FuzDataOut))
         begin
              RuleEn <= FuzDataOut[0];
             InputEn <= FuzDataOut[1];
        end
         else
        begin
             RuleEn <= 1'b0;
            InputEn <= 1'b0;
        end
```

```
    end
end
always @ (posedge clk or posedge rst)
begin
    if (rst)
    begin
        EnReg <= 1'b0;
    end
    else
    begin
       if (InputEn & RuleEn)
            EnReg <= 1'b1;
       else if (Rdy == 1)
            EnReg <= 'h0;
    end
end
```

CODE :

```
always @ (negedge clk or posedge rst) // set FuzDataIn
begin
    if (rst)
       FuzDataIn <= 0;
    else
    begin
       .
       .
       .
    else if ((FuzAddr == 'h94) & (InputEn & RuleEn))
           FuzDataIn <= 'h01;
       else
           FuzDataIn <= 0;
    end
```

The rule base is then read by the FLC. The address is set to 0x93 and decremented till it bottoms out at 0x00. Evidently crisp inputs are to be read first. Then the membership function points are read rule wise. The software transmits the data in a sequence such that the last membership point of the output membership function for the last rule is stored at location ‘0x00’. The location ‘0x8f’ is used for the first point of the first input membership function for the first rule.

The crisp inputs are stored with the values for second input going first at locations ‘0x90’ and ‘0x91’.

The FLC is designed to evaluate the outputs rule wise and accumulate them. This is due to the nature of the method of defuzzification chosen. A multiply and accumulate operation is to be performed. Thus the FLC reads the points rule wise. It reads the points for the one rule only at a time and computes the output before reading the next set of points.

CODE :

```
.
.
.
else if (EnReg & !((FuzAddr == 'h7e) || (FuzAddr == 'h6c) || (FuzAddr == 'h5a)
                || (FuzAddr == 'h48) || (FuzAddr == 'h36) || (FuzAddr == 'h24)
                || (FuzAddr == 'h12) || (FuzAddr == 'h00)))
begin
    FuzAddr <= FuzAddr - 1;
end
.
.
.
```

After FLC has read the last set of points (for the last rule) and computed their output, the crisp output value is computed. A ready strobe (*CrispOutDataRdy*) signals the termination of computation and that the output can be saved in the RAM from the output bus of the FLC. The most significant byte of the five byte output is stored at location 0x95 and the least significant bit is stored at location 0x99. The memory address is set to 0x95 and incremented till 0x99. A one bit counter is used (as a toggling signal) to control the write enable of the RAM. Port B is used by the hardware for this transfer as it is interfaced to the FLC.

CODE :

```
.
.
.
else if (!EnReg & (FuzAddr == 'h99))
begin
    if (!FuzWrCount)
        begin
       FuzAddr <= 'h94;
       UartTransmit <= 1;
        end
    else
       FuzWrCount <=0;
end

else if (!EnReg & (FuzAddr == 'h98))
begin
    if (!FuzWrCount)
    begin
        FuzAddr <= 'h99;
        FuzWrCount <= 1;
    end
    else
        FuzWrCount <= 0;
end
else if (!EnReg & (FuzAddr == 'h97))
begin
    if (!FuzWrCount)
    begin
        FuzAddr <= 'h98;
        FuzWrCount <= 1;
    end
    else
        FuzWrCount <= 0;
end
else if (!EnReg & (FuzAddr == 'h96))
begin
    if (!FuzWrCount)
    begin
        FuzAddr <= 'h97;
        FuzWrCount <= 1;
    end
```

```
    else
        FuzWrCount <= 0;
end
else if (!EnReg & (FuzAddr == 'h95))
begin
    if (!FuzWrCount)
    begin
        FuzAddr <= 'h96;
        FuzWrCount <= 1;
    end
    else
        FuzWrCount <= 0;
end

else if (!EnReg & (FuzAddr == 'h00) & CrispOutDataRdy)
begin
    FuzWrCount <= 1;
     FuzAddr <= 'h95;
end
.
.
.
```

On successful storage of the last byte at RAM locations, a strobe (*UartTransmit*) is set. The output value bytes are copied to five byte wide registers. The address is set to 0x95 and incremented till 0x99 the locations are read through port A. These bytes are to be transmitted to the software. The UART baud clock runs much slowly than the system clock. There would be problems synchronizing movement of data out of the RAM and into the UART buffer as much more time is needed to transmit a byte than reading it from the RAM. Therefore five registers are used to copy the bytes in a single read operation.

The UART then starts transmitting these five bytes to the software. A counter is used to count the number bytes transmitted. The most significant byte of the output is sent out first. When all

bytes have been sent, the UART stops transmitting. The system idles and waits for the software to send new set of crisp inputs and write ‘0x03’ to the control word location.

CODE :

```
always @ (posedge clk or posedge rst)
begin
    if (rst)
    begin
       UartAddr <= 'h0;
    end
    else
    begin
       if (UartWea & (UartAddr <= 'h93))
            UartAddr <= UartAddr + 1;
       else if (UartTransmit & (UartAddr == 'h94))
            UartAddr <= 'h95;
        else if (UartTransmit & (UartAddr == 'h95))
            UartAddr <= 'h96;
        else if (UartTransmit & (UartAddr == 'h96))
            UartAddr <= 'h97;
        else if (UartTransmit & (UartAddr == 'h97))
            UartAddr <= 'h98;
       else if (UartTransmit & (UartAddr == 'h98))
            UartAddr <= 'h99;
        else if (UartAddr == 'h99)
            UartAddr <= 'h0;
    end
end
```

CODE :

```
always @ (posedge clk or posedge rst) // to save CrispOutput from RAM
begin
    if (rst)
    begin
       UartCrispBuffer1 <= 0;
       UartCrispBuffer2 <= 0;
       UartCrispBuffer3 <= 0;
       UartCrispBuffer4 <= 0;
       UartCrispBuffer5 <= 0;
    end
    else
```

```
    begin
       if (UartTransmitRegR & (UartAddrReg == 'h95))
       begin
           UartCrispBuffer5 <= UartDataOut;
       end
       else if (UartTransmitRegR & (UartAddrReg == 'h96))
       begin
           UartCrispBuffer4 <= UartDataOut;
       end
       else if (UartTransmitRegR & (UartAddrReg == 'h97))
       begin
           UartCrispBuffer3 <= UartDataOut;
       end
       else if (UartTransmitRegR & (UartAddrReg == 'h98))
       begin
           UartCrispBuffer2 <= UartDataOut;
       end
       else if (UartTransmitRegR & (UartAddrReg == 'h99))
       begin
           UartCrispBuffer1 <= UartDataOut;
       end
    end
end
```

CODE:

```
always @ (posedge baud_clk_bd or posedge rst)
begin
    if (rst)
        begin
            tx_data_bd <= 'h0;
            new_tx_data_bd <= 'h0;
            UartByteCount <= 'b000;
        end
        else
        begin
             if (!tx_busy_bd & UartTransmitRegRR & (UartByteCount == 'b000))
             begin
                tx_data_bd <= UartCrispBuffer5;
                new_tx_data_bd <= 1;
                UartByteCount <= UartByteCount + 1;
             end
             else if (!tx_busy_bd & UartTransmitRegRR & (UartByteCount == 'b001))
             begin
```

```
            tx_data_bd <= UartCrispBuffer4;
            new_tx_data_bd <= 1;
            UartByteCount <= UartByteCount + 1;
         end
         else if (!tx_busy_bd & UartTransmitRegRR & (UartByteCount == 'b010))
         begin
            tx_data_bd <= UartCrispBuffer3;
            new_tx_data_bd <= 1;
            UartByteCount <= UartByteCount + 1;
         end
         else if (!tx_busy_bd & UartTransmitRegRR & (UartByteCount == 'b011))
         begin
            tx_data_bd <= UartCrispBuffer2;
            new_tx_data_bd <= 1;
            UartByteCount <= UartByteCount + 1;
         end
         else if (!tx_busy_bd & UartTransmitRegRR & (UartByteCount == 'b100))
         begin
            tx_data_bd <= UartCrispBuffer1;
            new_tx_data_bd <= 1;
            UartByteCount <= UartByteCount + 1;
         end
         else if (!tx_busy_bd)
         begin
               UartByteCount <= 'b000;
         end
         else
         begin
               tx_data_bd <= 'h0;
               new_tx_data_bd <= 'h0;
         end
      end
end
```

4.4. Fuzzy Logic Controller Architecture

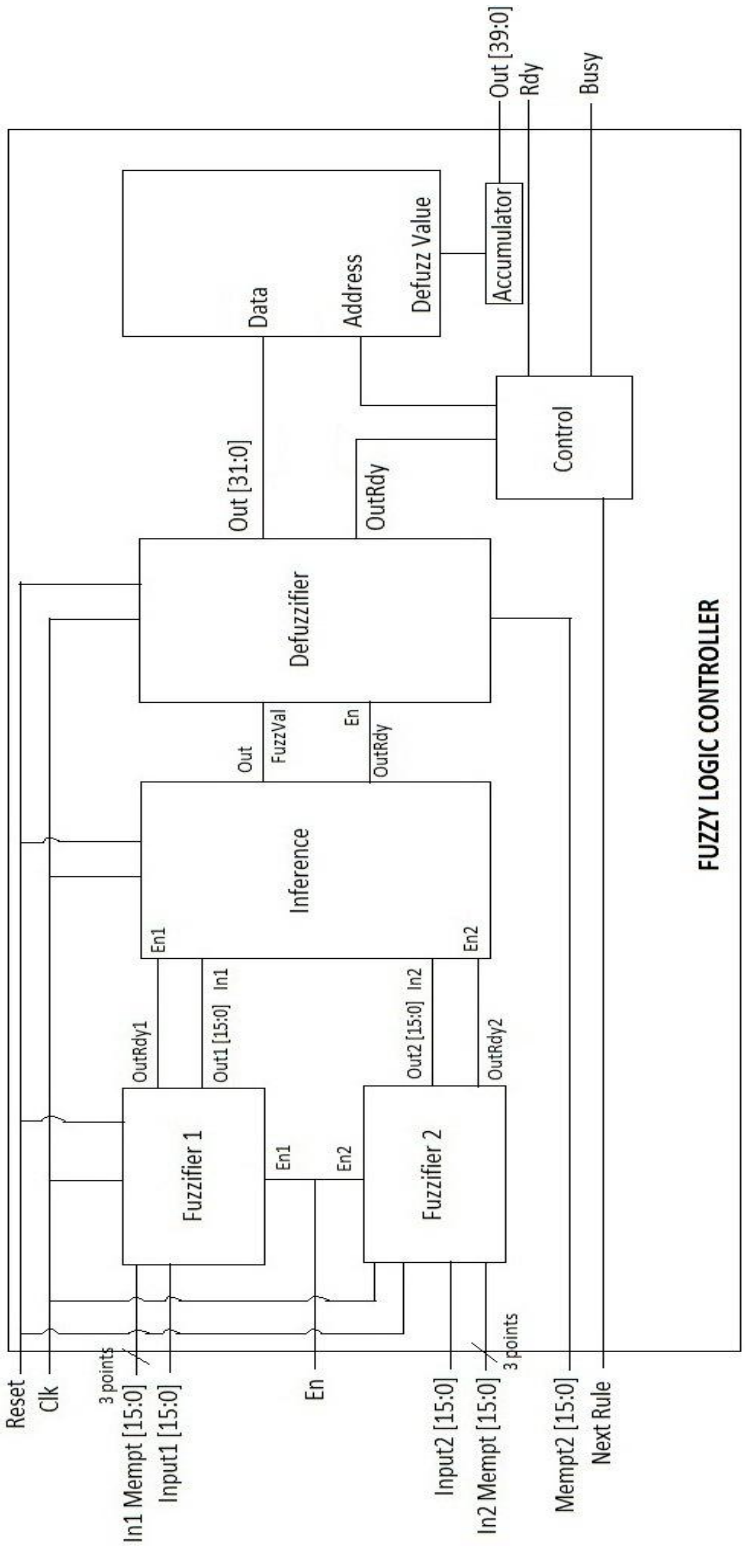


Figure 6 Architecture of the Fuzzy Logic

The FLC module consists of three sub modules and control logic. There is a 2-byte register file consisting of 8 registers for storing each rule's output. A 5 byte accumulator is present for multiplication and accumulation. The sub modules are,

1. Fuzzifiers (two, one for each input)
2. Inferenece
3. Defuzzifier

4.4.1. Fuzzifier

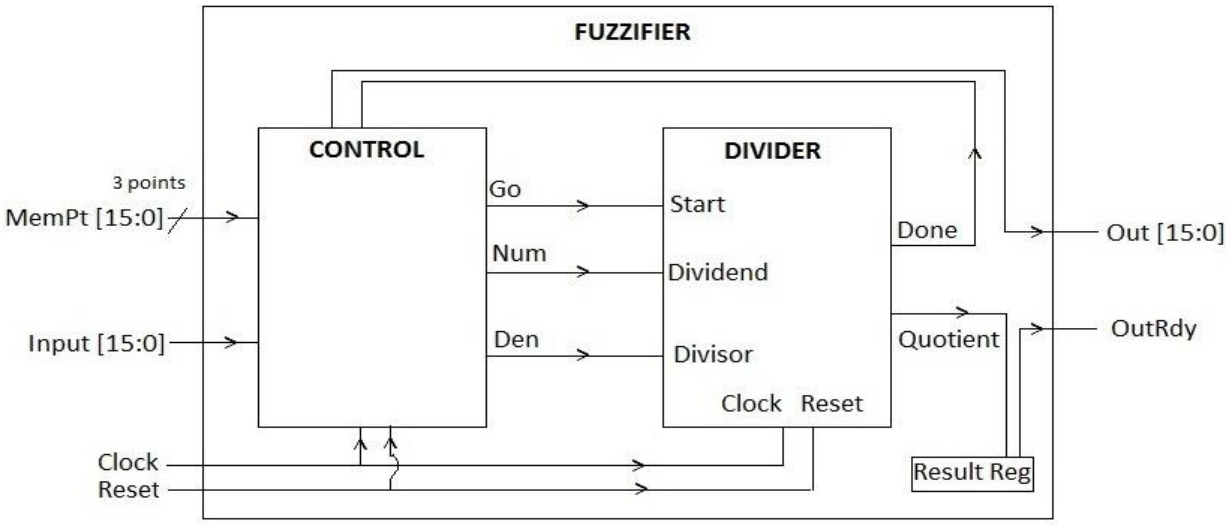


Figure 7 Fuzzifier architecture

The fuzzifier module's functionality is to compute a membership value for given input and membership function. Hence these values are supplied to the fuzzifier in Q8.8 format through 16 bit lines.

The fuzzification operation involves calculating the abscissa for a given line and ordinate. It returns membership values lying between 0 and 1 for crisp values given in any range. Therefore

a divider was designed and implemented. The divider uses a faster version of the non-restoring division algorithm involving shifting the divisor to the right while checking the difference with the dividend.

The fuzzifier checks if the inputs are in range for calculation. If it lies beyond the extremes of the triangle then the membership value would be zero. Else the output would be calculated using the divider as per the following equation.

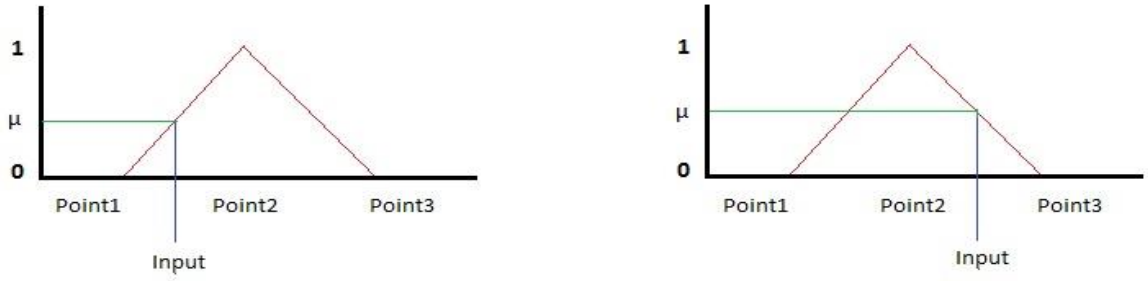


Figure 8 Calculation of membership values

If Input > Point1 and Input < Point2

$$\mu = Input - Point_1 / Point_2 - Point_1$$

If Input > Point2 and Input < Point3

$$\mu = Point_3 - Input / Point_3 - Point_2$$

The divider gives its output serially. There is no enable signal. A start (*Go*) signal triggers the calculation and the quotient is ready on the output bus after certain clock cycles along with a strobe. The number of clock cycles required to generate the output would be equal to the number of bits in the quotient. The divider was designed to carry out integer divisions. However

fixed point numbers can be divided as long as the scaling factors are accounted for. Sometimes when fractional outputs are expected, it is obvious that some of the dividend bits result in zeroes in the quotient. Thus it would be intelligent if those bits were skipped. This s facilitated by setting the number of bits to be truncated in the quotient. Thus the output arrives in number of cycles equal to the number of bits in the quotient minus the number of bits to be truncated.

CODE :

```
always @ (posedge Clock)
begin
		if (Reset)
		begin
			//reset the divider
			grand_dividend <= 'b0;
			grand_divisor <= 'b0;
			count <= 'b0;
			int_quotient <= 'b0;
			int_done <= 1'b0;
			div_in_prog <= 1'b0;
		end
		else
		begin
			int_done <= 1'b0;
			if ((Start == 1) && (div_in_prog == 0)) // start a division

			begin
				int_quotient <= 'b0;
				count <= 'b0;
				grand_dividend <= 'b0;
				grand_divisor <= 'b0;
				grand_dividend[M+R-1:R] <= Dividend; //load dividend
				grand_divisor[M+N+R-S-2:M+R-S-1] <= Divisor; //load divisor
				div_in_prog <= 1'b1;
			//flag to signal that a division has been started and is ongoing
			end
			else if (count == 15 && div_in_prog == 1) //check if division has
ended
			begin
```

```
                        if (int_done == 0) //check if ready output has been
signalled
                        begin
                           int_quotient <= int_quotient << 1;  //final shift
                               int_quotient[0] <= !diff[M+N+R-S-1]; //final shift
                               div_in_prog <= 1'b0;
                               int_done <= 1'b1;
                        end
                end
                else if (div_in_prog == 1) //check if division is in progress
                begin
                        if (diff[M+N+R-S-1] == 0) grand_dividend <= diff;
                        //difference is new dividend if result is positive
                        int_quotient <= int_quotient << 1;
                        //shift quotient and add bits in place of LSB
                        int_quotient[0] <= !diff[M+N+R-S-1];
                        grand_divisor <= grand_divisor >> 1; //shift divisor right
                        count <= count + 1; //increment count register
                end
        end
end
```

CODE:

```
begin
        if (Reset)
        begin
                OutRange <= 1'b0;
                Num <= 16'b0;
                Den <= 16'b0;
                Busy <= 1'b0;
                Out <= 16'b0;
                OutRdy <= 1'b0;
                Go <= 1'b0;
        end
        else
        begin
        case (NextState)
                1'b0:
                begin
                        if (OutRange)
                        begin
                                if (Busy)
                                begin
```

```
                        Out <= 16'b0;
                        OutRdy <= 1'b1;
                end
                Busy <= 1'b0;
        end
        else
        begin
                if (Busy)
                begin
                        Out <= RsltReg;
                        OutRdy <= 1'b1;
                end
                Busy <= 1'b0;
        end
end
1'b1:
begin
        Go <= 1'b0;
        OutRdy <= 1'b0;
        if (!Busy)
        begin
                if ((Input > MemPoint1) && (Input < MemPoint2))
                begin
                        Num <= Input - MemPoint1;
                        Den <= MemPoint2 - MemPoint1;
                        OutRange <= 1'b0;
                        Busy <= 1'b1;
                        Go <= 1'b1;
                end
                else if (Input > MemPoint2 && Input < MemPoint3)
                begin
                        Num <= MemPoint3 - Input;
                        Den <= MemPoint3 - MemPoint2;
                        OutRange <= 1'b0;
                        Busy <= 1'b1;
                        Go <= 1'b1;
                end
                else
                begin
                        OutRange <= 1'b1;
                        Busy <= 1'b1;
                end
```

```
                end
        end
        default :
        begin
                OutRdy <= 1'b0;
        end
    endcase
    end
end
```

4.4.2. Inference

This system uses Mamdani implication, in which the inference is carried out by comparison between the firing strengths or the membership values of each input for all the rules. Therefore the inference module has two 16 bit inputs and two separate enables which are connected to the outputs of the fuzzifiers and the ready strobe respectively, in a cascade. Only when both the enable signals are high the inference module is enables and performs a comparison between its two inputs. As per the implication method provided in the inference engine, the inference module gives the output accordingly. In this case a 'minimum' implication method was chosen. Thus the minimum of the two inputs is selected as the output by a comparator. There is a ready strobe that goes high as soon as data is valid on the output bus.

CODE :

```
begin
    case (NextState)
    1'b0:
    begin
        if (Done)
        begin
            Out <= OutReg;
            OutRdy <= 1'b1;
        end
    end
    1'b1:
    begin
```

```
                OutRdy <= 1'b0;
                Done <= 1'b0;
                if (Input1 > Input2)
                begin
                        OutReg <= Input2;
                        Done <= 1'b1;
                end
                else
                begin
                        OutReg <= Input1;
                        Done <= 1'b1;
                end
        end
        default:
        begin
                OutRdy <= 1'b0;
        end
        endcase
end
```

4.4.3 Defuzzifier

Weighted average defuzzification method is used. This calculates the output of each rule as product of a membership function point and rule strength (inference output) and averages them for all rules. There is multiplication and summation involved. The defuzzifier module only calculates the output of each rule. For this it requires the output of the inference module as input and the output membership point corresponding to unity membership. It multiplies these 16 bit numbers in Q8.8 format to obtain a 32 bit result in Q16.16 format.

This output data is latched on to a register and put on the output bus. The ready strobe is pulsed in this case indicating availability of valid data on the bus.

CODE :

```
begin
                case (NextState)
                1'b0:
```

```
        begin
                if (Done)
                begin
                        Done <= 1'b0;
                        Out <= OutReg;
                        OutRdy <= 1'b1;
                end
        end
        1'b1:
        begin
                OutRdy <= 1'b0;
                OutReg <= MemPoint2 * FuzzyValue;
                Done <= 1'b1;
        end
        default:
                OutRdy <= 1'b0;
        endcase
end
```

4.4.4 State Machine Description

The fuzzifier resides in *IDLE* state after reset. In this state the contents of the accumulator register are output on to the bus. The accumulator register serves the purpose of storing the sum of the output of the rules. As a maximum of eight rules can be accommodated by the system and output of each rule is of 32 bits in size, the accumulator should be at least 35 bits wide. Thus a 40 bit accumulator is used. When the enable signal for FLC goes high it enters the *ACCUMULATE* state.

In this state the system waits for new set of crisp inputs and the membership points for each rule to be available on the input bus. The *NextRule* signal is an input to signal this, when it goes high, the system transitions to *COMPUTE* state.

In the *COMPUTE* state the inputs and membership points are provided to the respective modules. Then the cascade operation is started with setting the enable signal for the fuzzifier high. After certain delay the output of each rule is stored in the register file. Then a strobe is set which causes the system to transition back to *COMPUTE* state.

Now, in the *ACCUMULATE* state the accumulator is updated with summation of all the outputs that have arrived till now. The system repeats this until outputs for all the rules have been computed. Then it averages the sum for all the rules (8 in this case; thus averaging is division by 8 or 3 left bit wise shifts). This is followed by signaling a strobe which causes the system to transition back to *IDLE* state where it would output the contents of accumulator.

The entire time the FLC is involved in calculation, storage and accumulation of the output of one rule, a busy signal is set high. This output signal conveys if the FLC is free to receive new membership points for next rule.

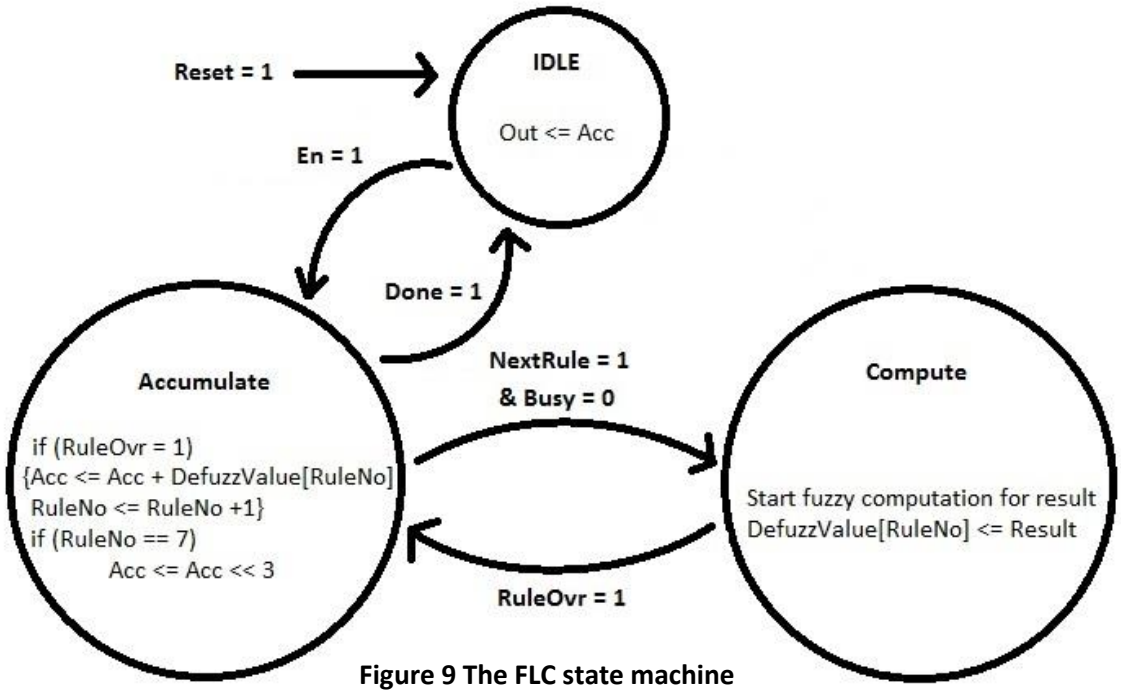


Figure 9 The FLC state machine

FLC state diagram

Connecting the sub-modules in a cascade eases the transfer of data from module to module. As the data flow is simply one directional, assigning the output and ready strobes of one module as the input and enables respectively, saves the design from using additional buses, control signals, and the logic to control these signals. Feeding data to the cascade and storing the resulting data for one rule leads to a simplified architecture.

There are provisions in the design to accommodate or be modified if required. A variety of membership functions, implication methods and defuzzification algorithms can be implemented without major changes in the RTL source.

CODE :

```
begin
	case (NextState)
	2'b00:
	begin
		Out <= Acc;
		if (Busy)
			Rdy <= 1'b1;
		Busy <= 1'b0;
		Done <= 1'b0;
	end
	2'b01:
	begin
		if (Rdy)
		begin
			DefuzzVal [0] <= 'h0;
			DefuzzVal [1] <= 'h0;
			DefuzzVal [2] <= 'h0;
			DefuzzVal [3] <= 'h0;
			DefuzzVal [4] <= 'h0;
			DefuzzVal [5] <= 'h0;
			DefuzzVal [6] <= 'h0;
			DefuzzVal [7] <= 'h0;
			Acc <= 0;
			RuleNo <= 0;
		AllValues <= 0;
		end
```

```
            RuleOvr <= 1'b0;
            Rdy <= 1'b0;
            if (Busy)
            begin
                  Acc <= Acc + DefuzzVal [RuleNo];
                  if (AllValues)
                  begin
                        Acc <= Acc >> 3;
                        Done <= 1'b1;
                  end
                  else if (RuleNo == 7)
                        AllValues <= 1'b1;
                  else
                  begin
                        RuleNo <= RuleNo + 1;
                              Busy <= 1'b0;
                  end
            end
      end
      2'b10:
      begin
            if (Busy)
                  Go <= 1'b0;
            else
                  Go <= 1'b1;
            Busy <= 1'b1;
            if (ValueRdy)
            begin
                        DefuzzVal [RuleNo] <= Value;
                        RuleOvr <= 1'b1;
            end
      end
   endcase
end
```

5. Results

The GA based FLC was tested on Hang data function. The mathematical representation of the function is given by,

$$y = (1 + x_1^{-1.5} + x_2^{-2})^2$$

where, $1 \leq x_1 \leq 5$ and $1 \leq x_2 \leq 5$. It is a 2 input 1 output system. Rule base had been generated for this test case and it was optimized using GA.

The GA parameters are:

1. Population size = 6
2. Cross-over probability = 0.8
3. Mutation probability = 0.2
4. Number of generations = 600
5. Number of rules in the rule base = 8

Total mean square error for initial rule base = 386.213348

Total mean square error for final rule base = 53.802254

5.1. ANFIS vs. GA-FLC

Number of rules used in ANFIS = 7×7 = 49

Number of epochs = 10

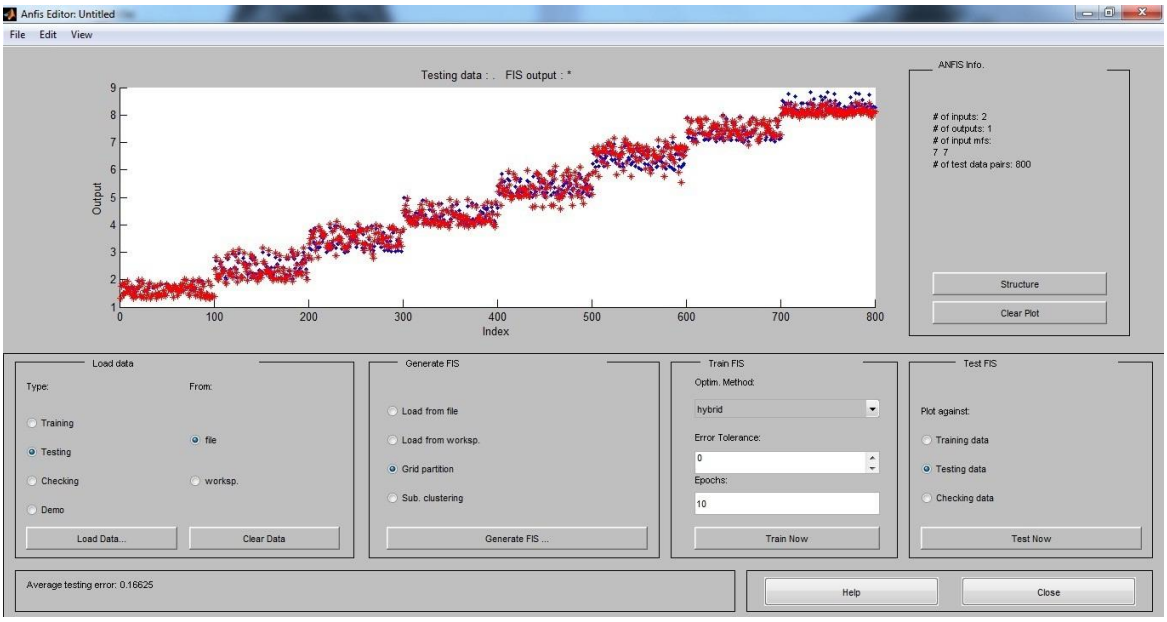


Figure 10 ANFIS plot (dots are desired outputs; asterisks are obtained outputs)

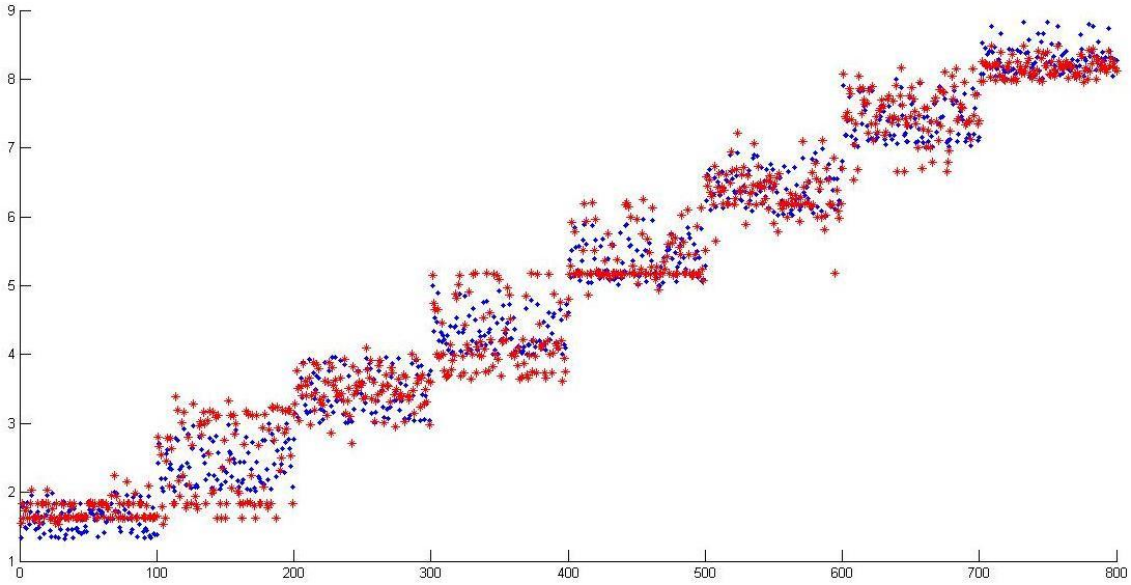


Figure 11 GA-FLC output plot (dots are desired outputs; asterisks are obtained outputs)

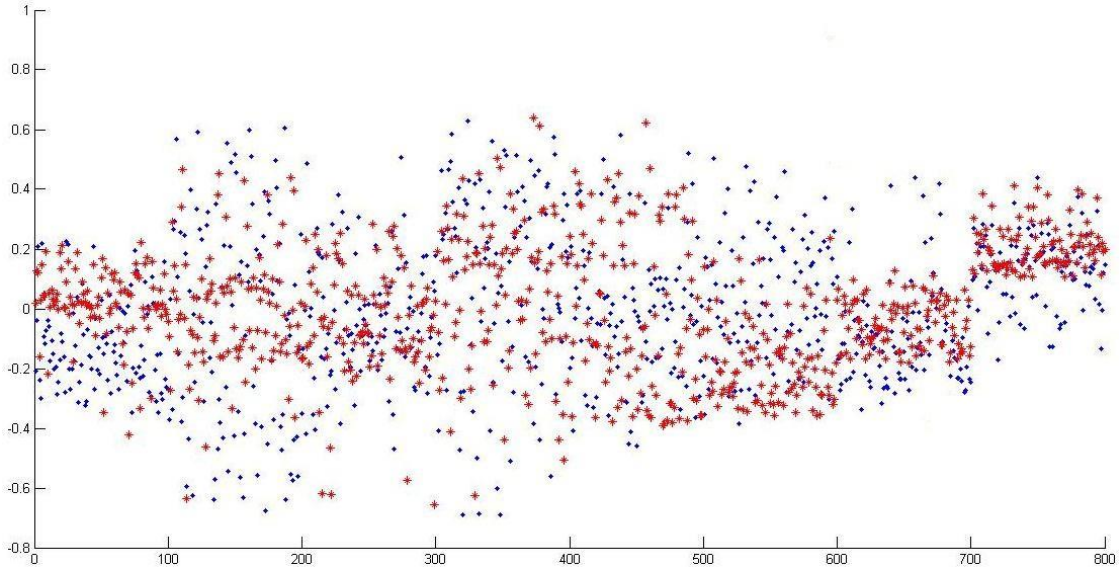
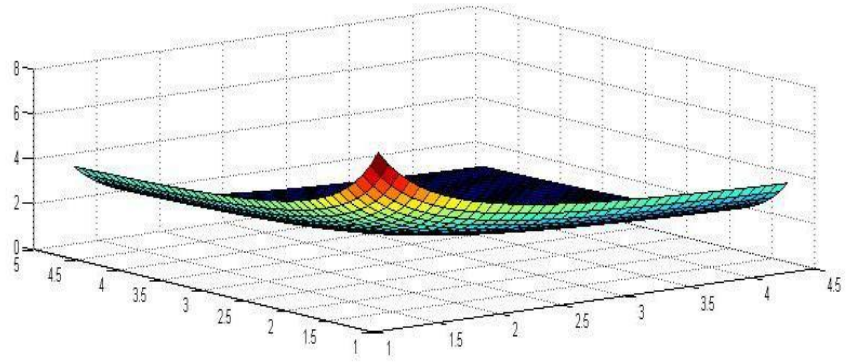
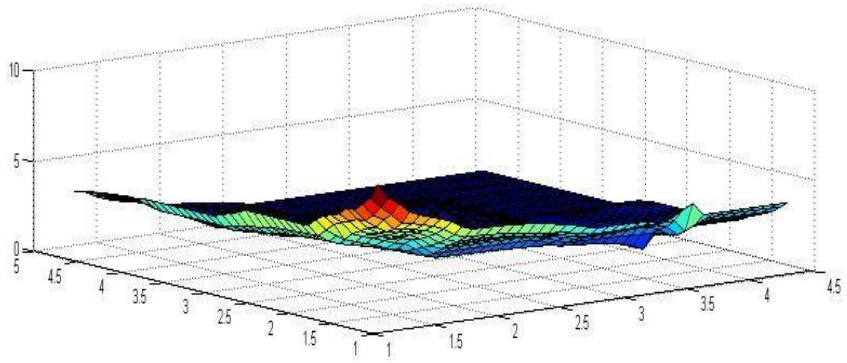


Figure 12 GA-FLC vs. ANFIS error plot (dots are errors from designed fuzzy system; asterisks are errors from ANFIS)



(a)



(b)

Figure 13 Hang data function plot. (a) Actual surface plot, (b) Surface plot with rule base obtained using GA

5.2. Data transmission form GUI to FPGA over UART

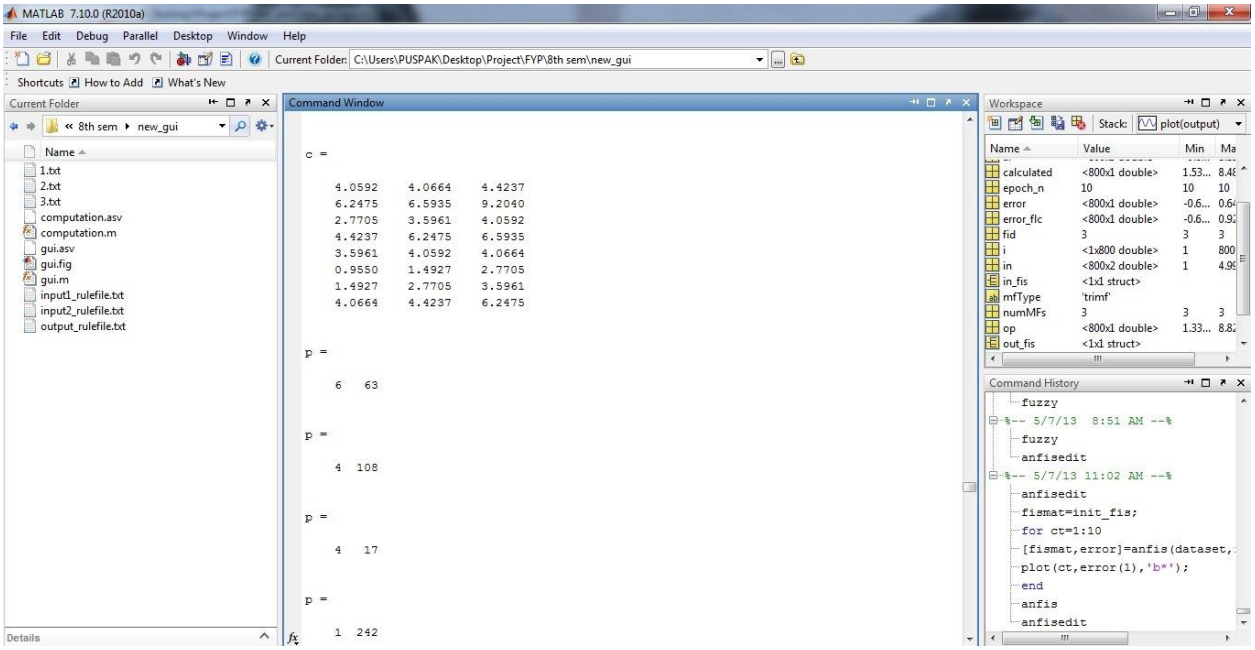


Figure 14 Data point transmission over GUI

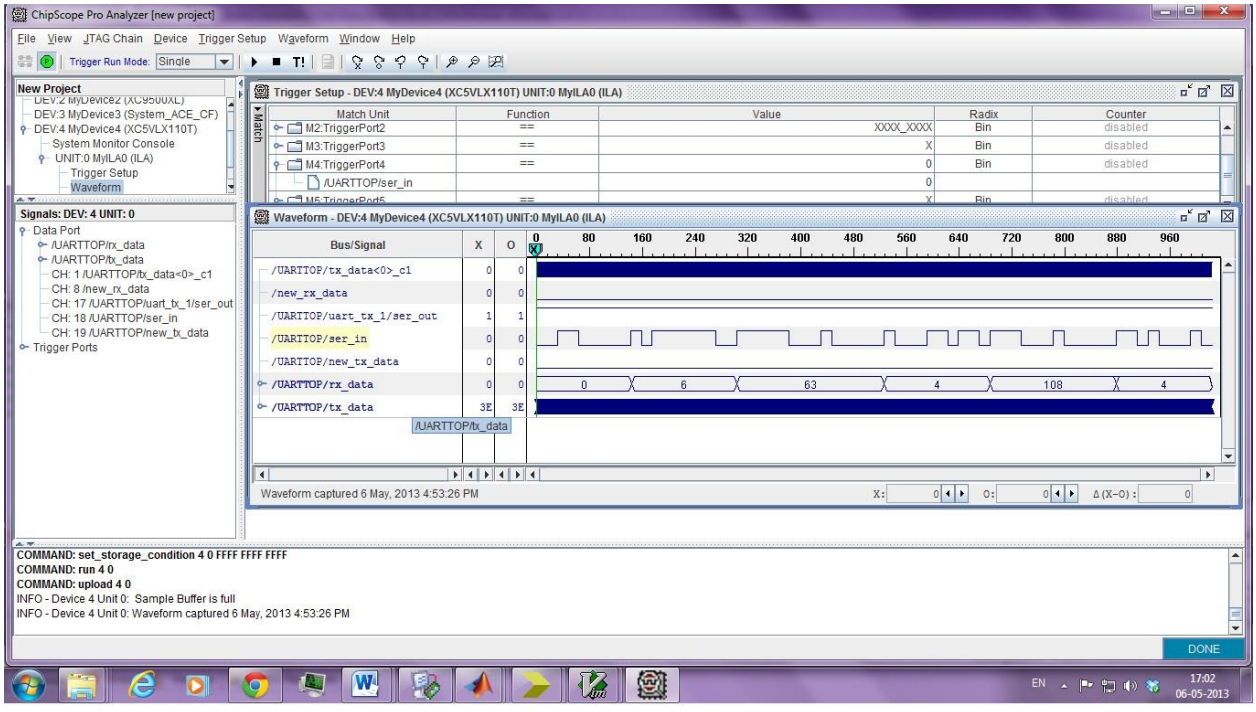
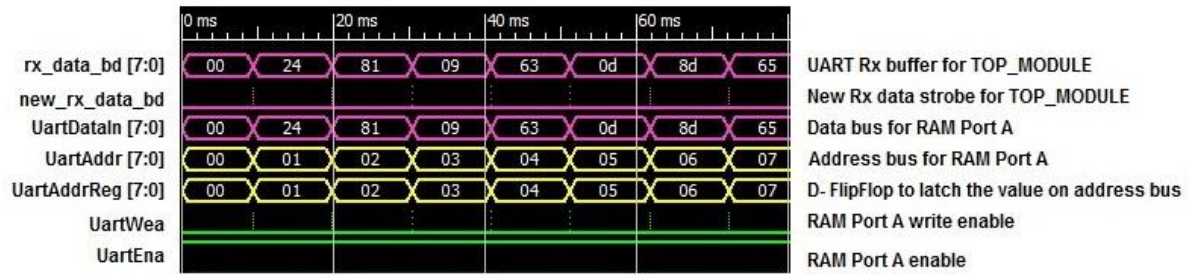


Figure 15 Data reception by the UART on FPGA

5.3. Software Simulation (with iSim)

5.3.1. TOP_MODULE simulation results:

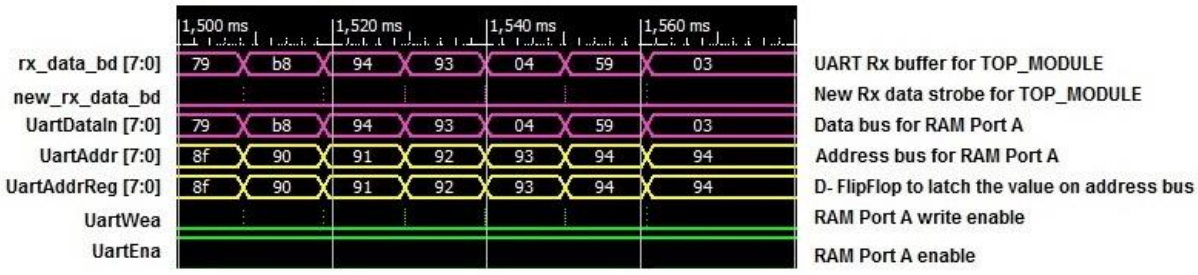


NOTE: The clock pulses are small enough for the FlipFlop to appear as if it is updated instantly with values on address bus though in reality it updates after one clock cycle.

The new_rx_data_bd signal triggers UartWea and address increment. The data in UART Rx buffer is placed on the data bus and stored at corresponding RAM locations.

Transfer of data from UART to BRAM

Figure 16 Waveforms depicting signals involved in transfer of data from UART to BRAM



NOTE: The clock pulses are small enough for the FlipFlop to appear as if it is updated instantly with values on address bus though in reality it updates after one clock cycle.

The new_rx_data_bd signal triggers UartWea and address increment. The data in UART Rx buffer is placed on the data bus and stored at corresponding RAM locations.

Transfer of data from UART to BRAM

Figure 17 Waveforms depicting signals involved in transfer of data from UART to BRAM

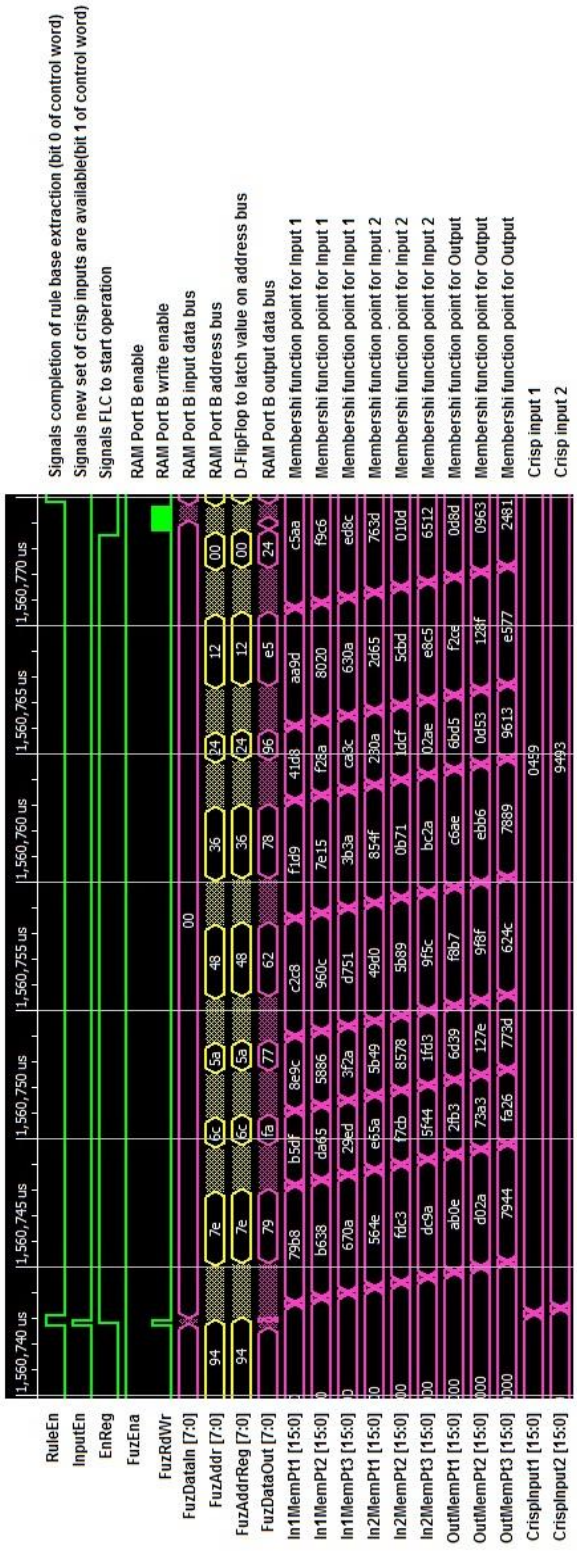


Figure 18 Waveforms depicting signals involved in transfer of data from BRAM to FLC (Overview of reading points for all rules)

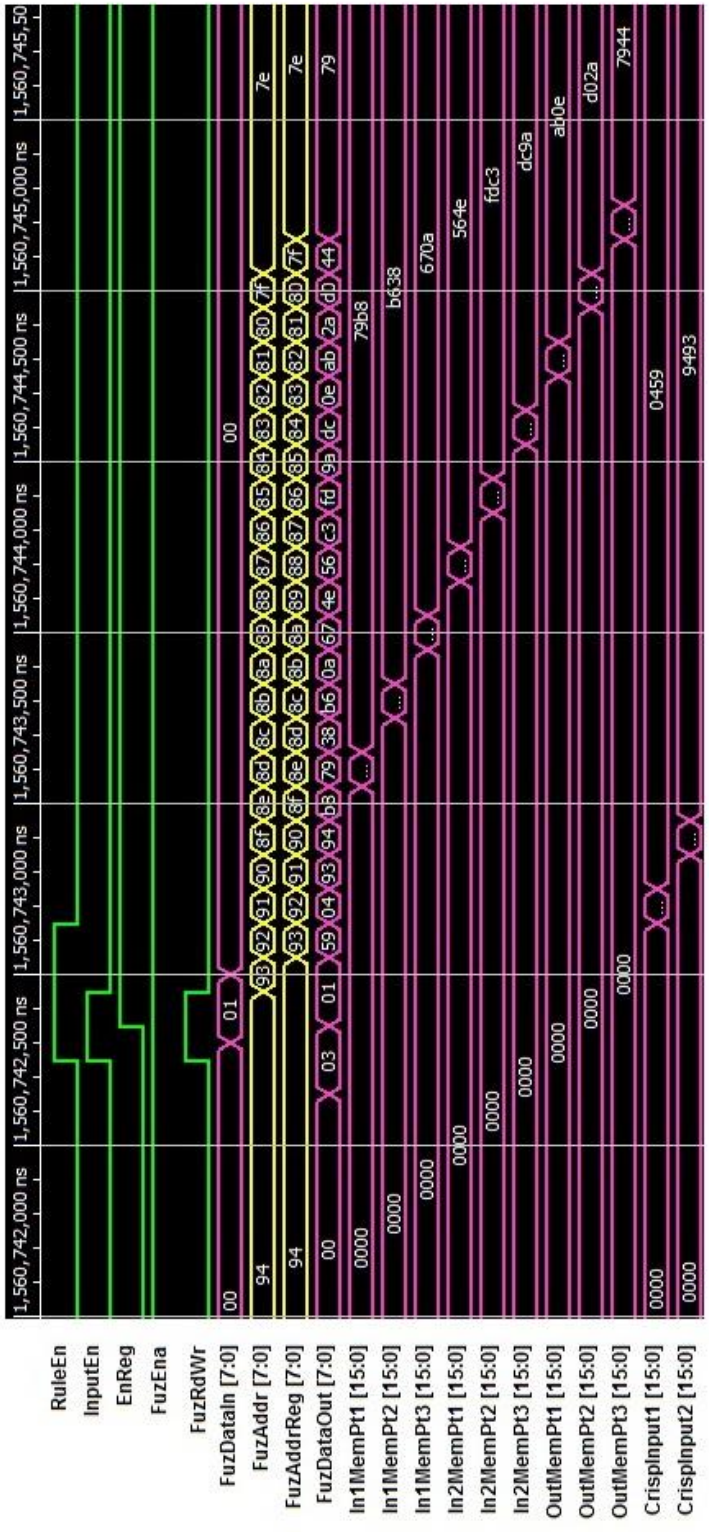


Figure 19 Waveforms depicting signals involved in transfer of data from BRAM to FLC (Reading of the points for first rule)

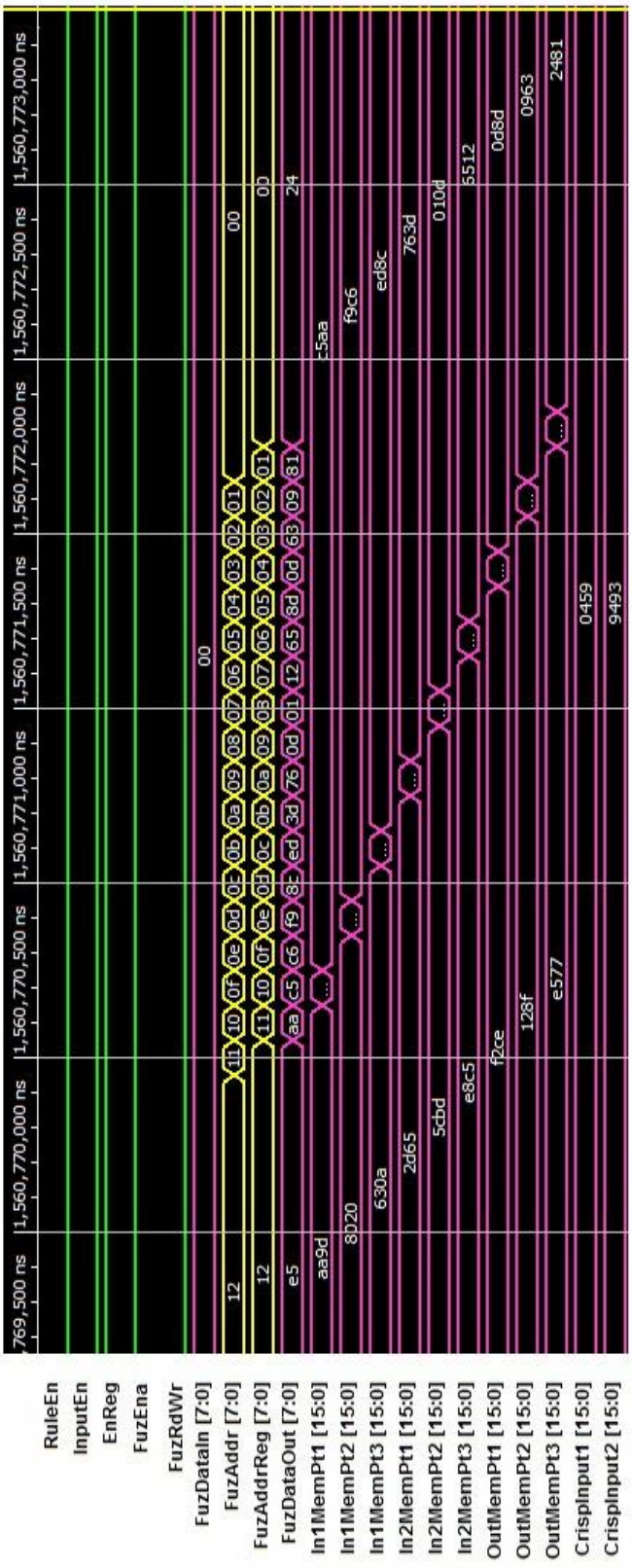
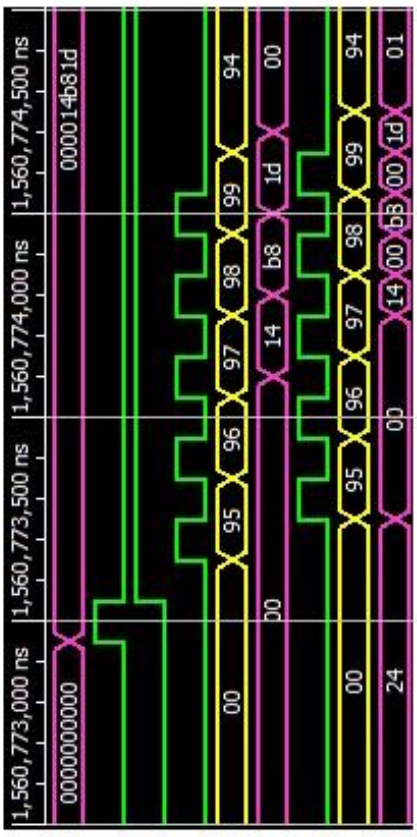


Figure 20 Waveforms depicting signals involved in transfer of data from BRAM to FLC (Reading of the points for the last rule)



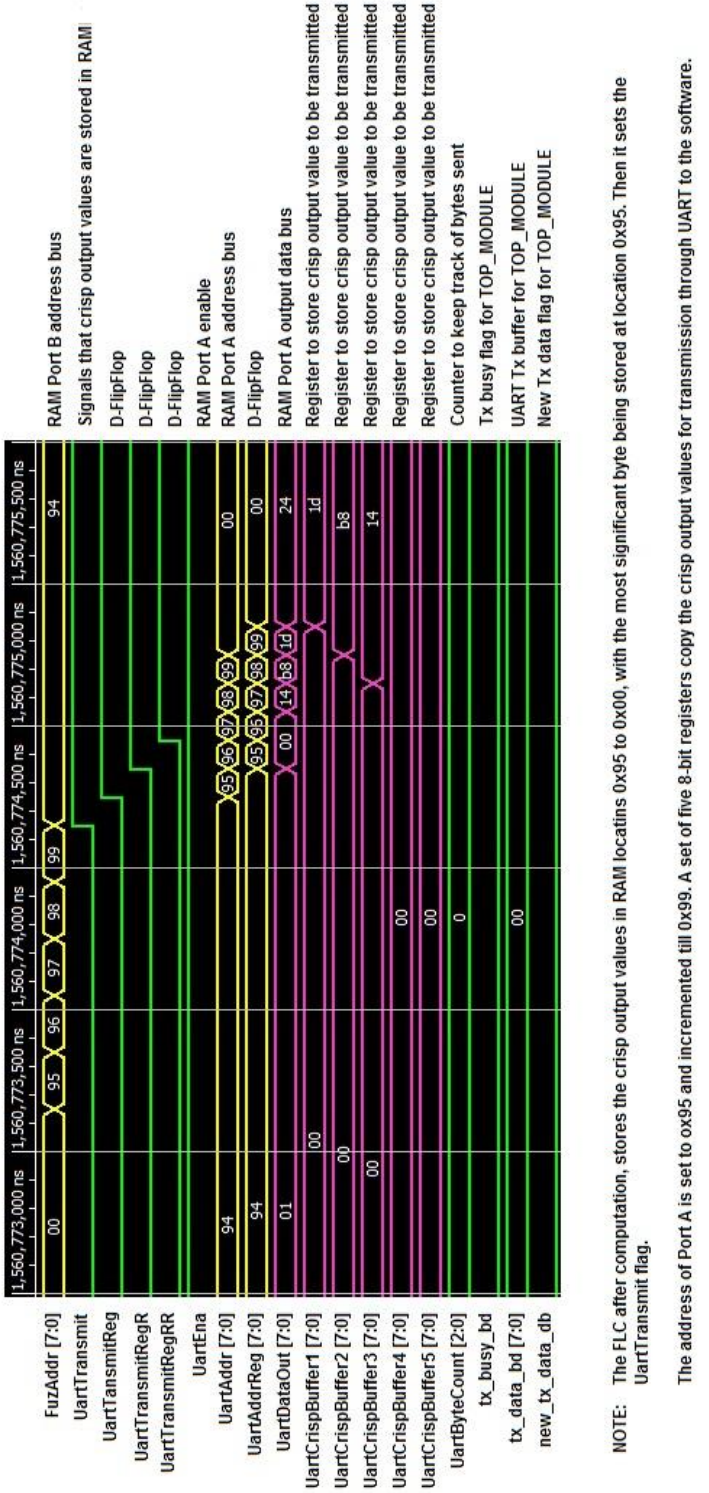
NOTE: The FLC signals that the crisp output has been computed by the Rdy strobe. Following this event address on Port B is set to 0x95 and incremented till 0x99. The 40-bit output value is written byte wise with the most significant byte stored at location 0x95.

An 1-bit counter is used to keep track of data transfer and toggle the write enable signal for RAM Port B.

After all the five bytes are written, the address is set to 0x94 and the FLC starts polling the Control Word for further operation.

Saving computed crisp output value in the RAM

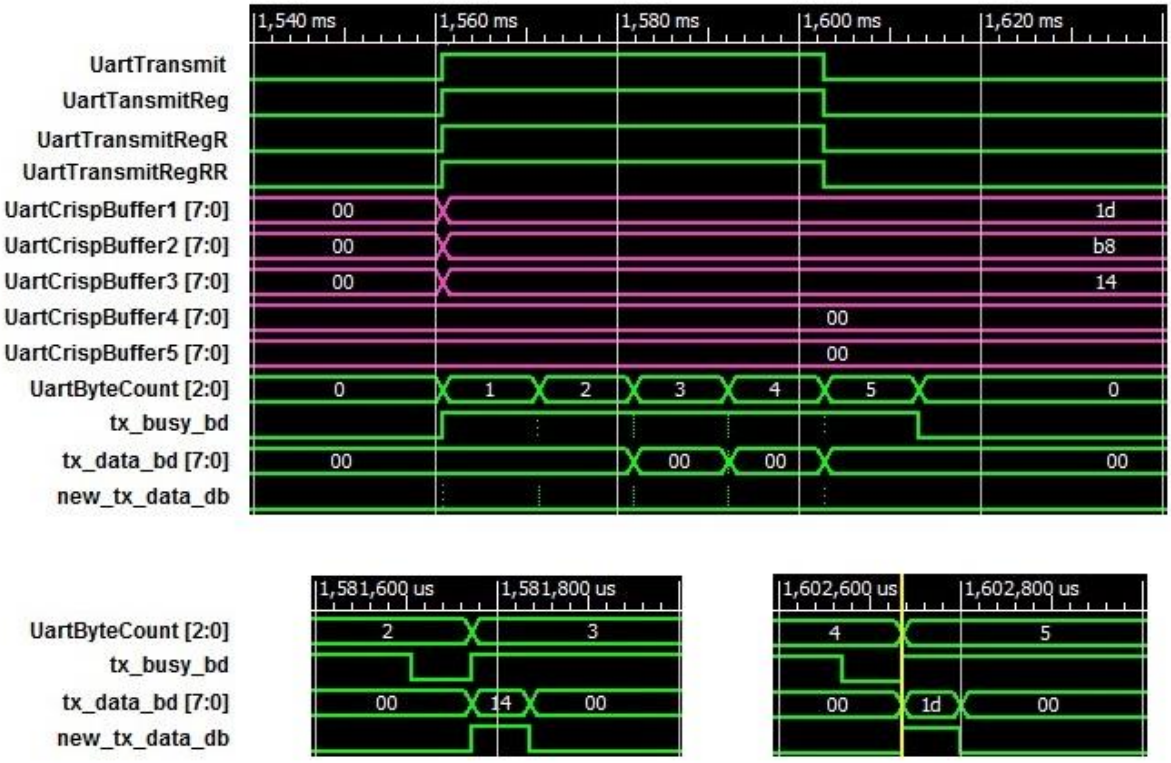
Figure 21 Waveforms depicting signals involved in transfer of data from FLC to BRAM (Storing of computed crisp output at RAM locations)



NOTE: The FLC after computation, stores the crisp output values in RAM locatins 0x95 to 0x00, with the most significant byte being stored at location 0x95. Then it sets the UartTransmit flag.

The address of Port A is set to ox95 and incremented till 0x99. A set of five 8-bit registers copy the crisp output values for transmission through UART to the software.

Figure 22 Waveforms depicting signals involved in transfer of data from BRAM to registers (Copying of five output bytes for transmission over UART)



NOTE: When UartTransmit is set and the registers have been loaded with valid data, the crisp output values are transmitted through UART to software.

The most significant byte is sent first. A counter is used to track the number of bytes transmitted successfully

Transmission of crisp output to software

Figure 23 Waveforms depicting signals involved in transfer of data from registers to UART (Transmission of five output bytes to software with most significant byte sent first)

5.3.2. FLC simulation results:

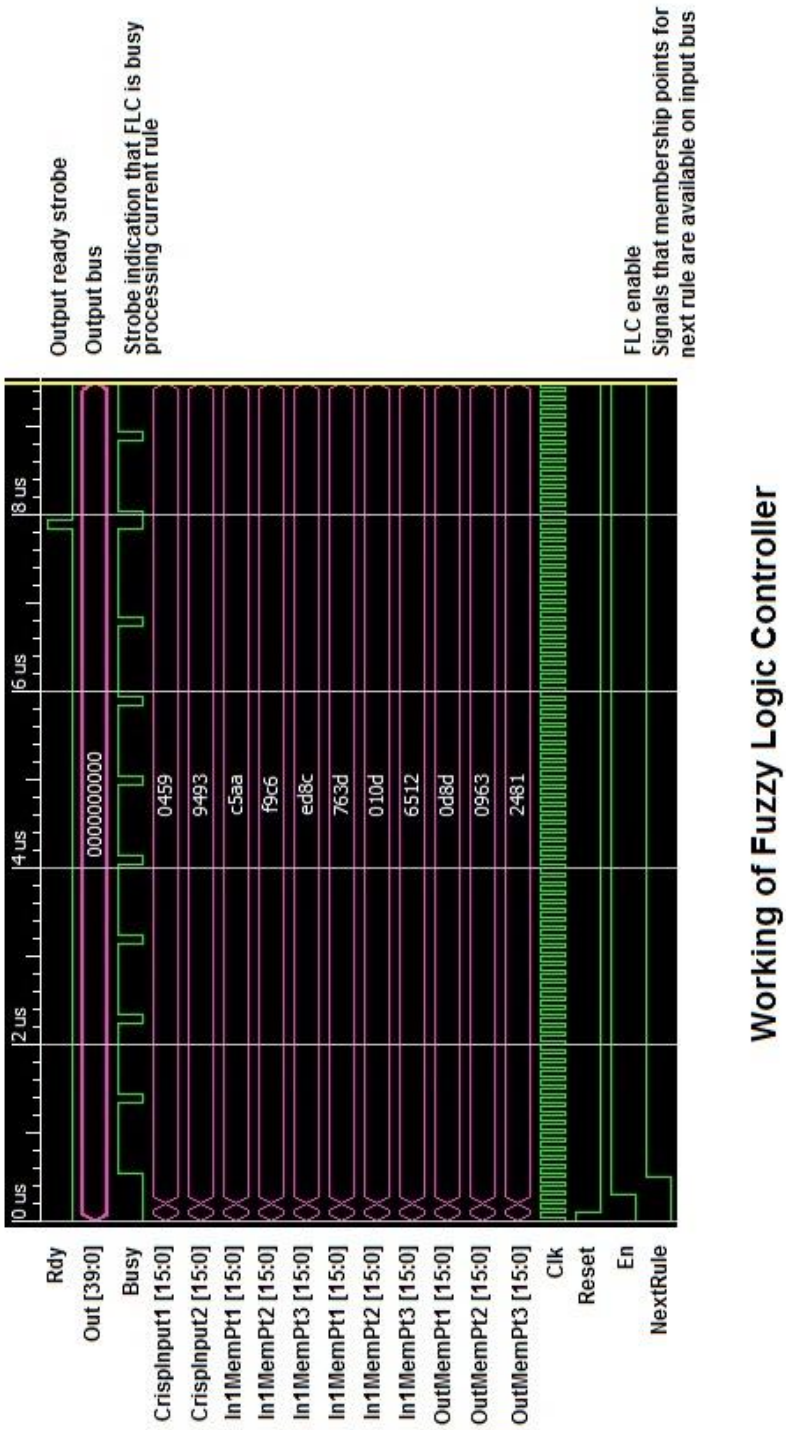


Figure 24 Waveforms depicting signals involved in computation of crisp output value for a set of crisp inputs (FLC operation)

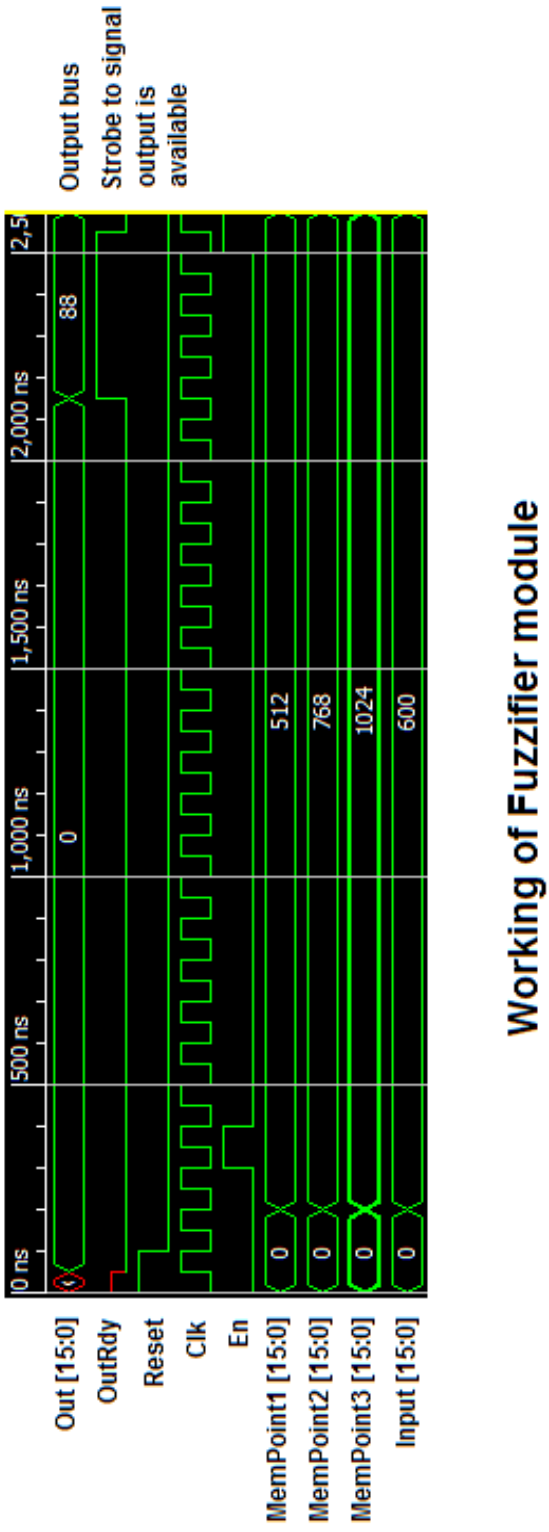


Figure 25 Waveforms depicting signals involved in computation of membership values (Fuzzifier operation)

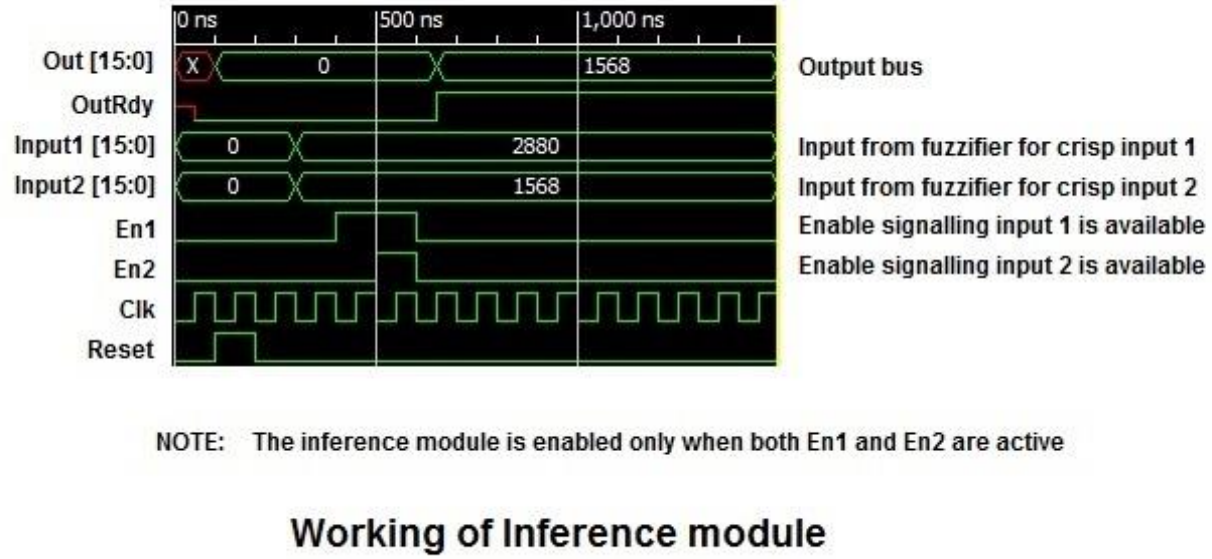


Figure 26 Waveforms depicting signals involved in implication procedure (Inference operation)

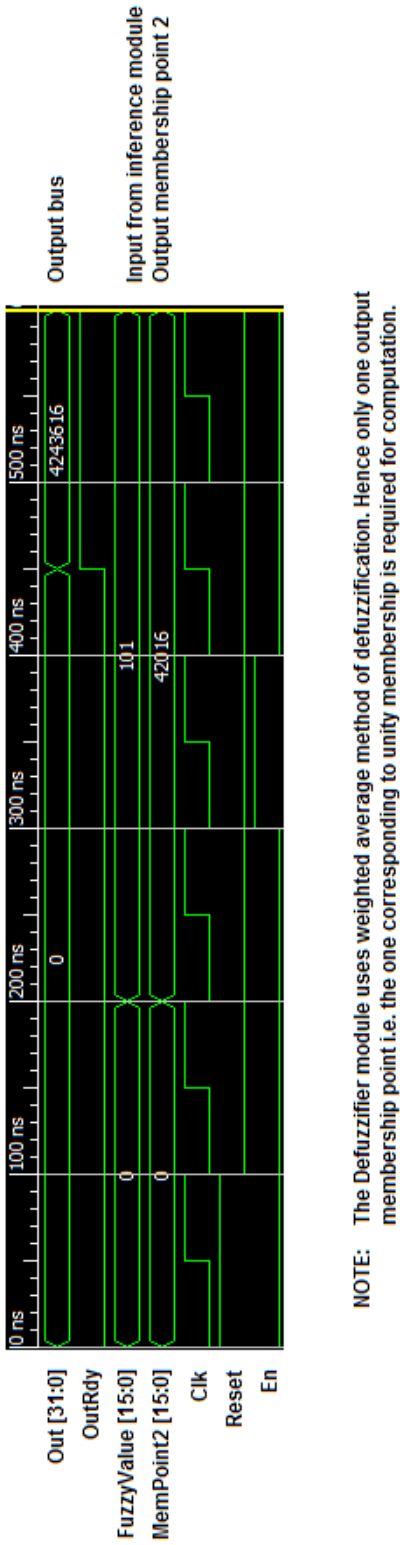


Figure 27 Waveforms depicting signals involved in calculation of output for each rule only

6. Conclusion

A novel approach towards rule base synthesis for fuzzy systems was taken. Using an evolutionary algorithm like Genetic Algorithm for convergence of the rule base proved to provide better results. Comparison with an ANFIS model in MATLAB which used 49 rules and ran for 100 iterations show that this novel method provide better results with lesser rules. Further there is no need for an expert to design the system. An accurate data set of the system/plant under discussion is required only for generating the rule base.

The hardware extracts the rule base once it is synthesized by the software. This is done over UART. Re-programmability of the rule base makes the system more flexible. The design has provisions to accommodate the use of other membership functions (Gaussian, sigmoid, etc), implication methods, and defuzzification methods.

Rigorous software simulation was carried out with iSim tool. The RTL code is fully synthesizable. Upon synthesis it was observed that the FLC module was implemented using DSP slices in the FPGA. This is consequence to the multiply and accumulate operations in the algorithm which are common amongst DSP routines.

The complete hardware testing could not be completed due to unresolved issues with the implementation hardware. Even though the UART module functioned fine which was analyzed using ChipScope tool; the hardware is unable to receive data from the software when the TOP MODULE is implemented on the FPGA.

The design has some weaknesses. The distribution of data points over the input range determines the accuracy and efficiency of the rule base formed. Due to use of evolutionary algorithms, rule base synthesis programs require huge hardware resources to run. Further neural networks provide better improvement of objective function than genetic algorithm. UART is a restrictive and slow interface. There no choice of remote rule base extraction by the hardware. The current implementation lacks ADC and DAC modules for interfacing the FLC to the real world. The computation delay introduced by the hardware should be optimized for application in plants.

Further, other evolutionary algorithms maybe used for better results from the rule base. If rule-base redundancy and conflict among rules are considered while designing, the performance of the rule-base can be more consistent. The system can be modified for real world interfacing. Optimization of division algorithms would result in faster computations. Optimizing the state machine and logic utilization can be done. This implementation may also be modified into a general purpose fuzzy processor. Replacing the low speed UART with a faster and more versatile interface such as Ethernet or Zigbee would facilitate faster and remote rule base extraction.

7. References

1. Timothy J Ross, *Fuzzy Logic with Engineering Applications.* 3rd ed., Chennai, Wiley, 2010.
2. J S R Jang, C T Sun, E Mizutani, *Neuro-Fuzzy and Soft Computing*. Upper Saddle River, NJ, Prentice-Hall, 1997.
3. Kalyanmoy Deb, *Optimization for Engineering Design: Algorithms and Examples*. New Delhi, Prentice-Hall India, 2005.
4. Pong P Chu, *FPGA prototyping by Verilog examples*. Hoboken, New Jersey, Wiley, 2008.
5. John Clayton, "Unsigned serial divider." Internet: http://www.opencores.org/project,serial_devide_uu, Aug. 19, 2009 [Sept. 14, 2012] .
6. M McKenna, B M Wilamowski, "Implementing a Fuzzy System on a Field Programmable Gate Array." In *Proc. IJCNN*, 2001, pp. 189-194.
7. Dr. Kasim M. Al-Aubidy, "FPGA Implementation of Fuzzy Inference System for Embedded Applications." Philadelphia University, Jordan.
8. Philip T. Vuong, Asad M. Madni, Jim B. Vuong, "VHDL Implementation for a Fuzzy Logic Controller." In *World Automation Congress*, 2006, pp. 1-8.