4-2016

A Parameterized Implementation of a Hybrid Fuzzy Boolean Finite State Machine using an FPGA

Sean T. Fuller

A PARAMETERIZED IMPLEMENTATION OF A HYBRID FUZZY BOOLEAN FINITE STATE MACHINE USING AN FPGA

by

Sean T Fuller

A thesis submitted to the Graduate College
in partial fulfillment of the requirements
For the degree of Master of Science in
Computer Engineering
Western Michigan University
April 2016

Thesis Committee:

Janos L Grantner, Ph.D., Chair
Bradley J. Bazuin, Ph.D., Advisor
Lina Sawalha, Ph.D.

A PARAMETERIZED IMPLEMENTATION OF A HYBRID FUZZY BOOLEAN FINITE STATE MACHINE USING AN FPGA

Sean T Fuller, M.S.E.

Western Michigan University, 2016

A fuzzy system based on the extended Hybrid Fuzzy Boolean Finite State Machine (HFB-FSM) model is designed and implemented in a Xilinx Zynq® all programmable System on Chip (SoC). The system allows for a single dominant crisp state and multiple non-dominant states to be activated simultaneously. A method is proposed to calculate the degree of membership of non-dominant states, β, by the distance of defuzzified fuzzy inputs to state transition trigger mechanisms.

The HFB-FSM is was realized by the author using custom logic written in VHDL (Very High Speed Integrated Circuit (VHSIC) Hardware Description Language) in the Programmable Logic (PL) portion of the SoC while one of the application processors provides system control and an Ethernet network interface. The design is flexible and parameterized to simplify and accelerate deployment to new applications.

The thesis first presents the theoretical background followed by a discussion of the detailed design of the custom logic and processor based systems. Detailed simulations are included to support the design material along with discussion of a relevant application.

ACKNOWLEDGEMENTS

I would first like to thank the members of my graduate committee, Dr. Janos Grantner, Dr. Bradley Bazuin, and Dr. Lina Sawalha, for taking the time to review my work and, hopefully, learning something from it. Particularly, I would like to thank Dr. Grantner for providing me guidance in the theory of fuzzy logic and for believing in me over the last few years.

Secondly, I would like to thank my current and former co-workers for helping throughout the years to improve my VHDL design skills and to reach this goal – one annoying question at a time. I would also like the thank Jon Cory of Xilinx for being supportive and helping obtain licensing, tutorials, and support for this and other graduate projects.

This wouldn't be complete without acknowledging the influence my mother and father have imparted. My mother gave me her tenacity and taught me the value of hard work. My father gave me his patience and taught me the value of education. Thank you for everything.

Finally, I would like to thank my wife, Thammala, and my three sons, Elliott, Hunter, and Maximus, for being as patient and understanding as a wife with three young and screaming children could be throughout this process.

Sean T Fuller

TABLE OF CONTENTS

Table of Contents - Continued

LIST OF TABLES

LIST OF FIGURES

List of Figures - Continued

1. Introduction to Fuzzy Logic

Fuzzy logic as we know it today was first introduced by Zadeh [1] as a means of addressing the inherent vagueness in nature and how this is handled by traditional binary or crisp logic. Rather than simply applying a 0 or 1 to indicate whether an element belongs to a class or set, the author proposed a membership function which can take on values in the range of [0, 1] to indicate a "grade of membership" for a given element in a class. By doing so, a class can now be used to not only make the distinction between members and non-members of a class but also those members which partially meet the membership criteria.

Fast-forward 50 years and the theory of fuzzy logic has gained wide acceptance in several application areas – largely in Asia and Europe rather than the United States. Some of the important applications of fuzzy logic include an automatic braking system utilized on the Sendai subway trains [2], auto-focusing of camera systems [3], and Antilock Braking Systems (ABS) in cars [4]. One area that remains a common research topic and has many current applications is the use of fuzzy logic in control systems.

1.1 Fuzzy Logic in Control Systems

Significant research exists on the applications of fuzzy logic to control systems and fine-tuning of these systems for various applications. However, one of the hurdles to implementing a fuzzy controller is the time required to model all aspects of the system and implement a suitable fuzzy controller when the control algorithms are complex or even unknown [5].

The problem of modeling and mimicking human control of a system remains a particularly complex challenge. In systems where the main control source is a human operator with many years of training or experience with the machinery or system, models for control may be vague or unknown. These applications would require significant and costly research to first develop the mathematical models that govern control through data collection and analysis then develop a system capable of implementing said models followed by significant fine-tuning to work out any issues. The cost of such a process and the resulting system may be prohibitive for many applications.

In applications where an electronic control system already exists the implementation logic may be flawed or incomplete or an external disturbance may result in an unknown state or control action. In such a system there may be a need for an ontological controller capable of

determining when the system has reached an unknown state or the control assumptions have been violated. The ontological controller must be capable of recognizing these conditions and brining the control system back to a known state.

1.2 Introduction to the Hybrid Fuzzy Boolean Finite State Machine

One proposed system for addressing these issues is the Hybrid Fuzzy Boolean Finite State Machine (HFB-FSM) which was first presented in [6] then extended in [7] to address the modeling requirements of complex hybrid systems. In this system the concept of a crisp state is modified with the ability to stay in multiple crisp states at one time. A single dominant state can be active along with multiple non-dominant or β states [8]. The underlying state machine is comparable to a typical Boolean FSM where the transition points are based on crisp binary logic. The key difference is the use of β to assign a degree of membership to non-dominant states such that the output calculated in these states is combined through fuzzy composition with the dominant state output to form the final system fuzzy output. A block diagram of the HFB-FSM system is shown in Figure 1.

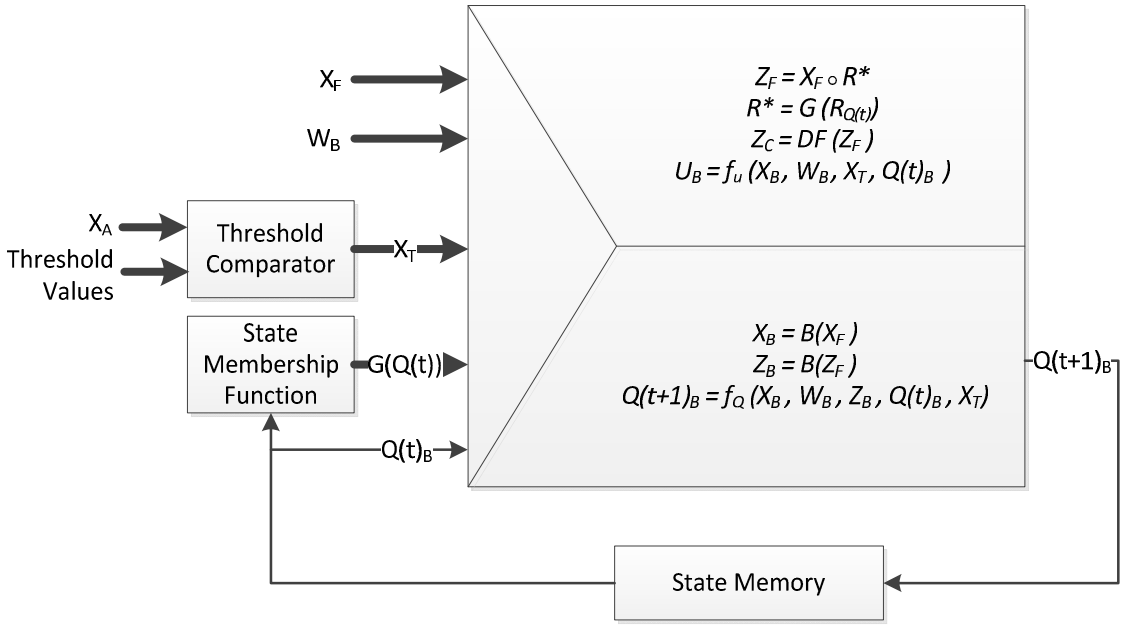

Figure 1. Extended HFB-FSM Model Block Diagram

$$Z_F = X_F \circ R^* \quad (1)$$

$$R^* = G(R_{Q(t)}) \quad (2)$$

$$Z_C = DF(Z_F) \quad (3)$$
$$U_B = f_U(X_B, W_B, X_T, Q(t)_B) \quad (4)$$
$$X_B = B(X_F) \quad (5)$$
$$Z_B = B(Z_F) \quad (6)$$
$$Q(t+1)_B = f_Q(X_B, W_B, Z_B, Q(t)_B, X_T,) \quad (7)$$

Z_F is the fuzzy output of the system and is generated through composition of the fuzzy input, X_F, and the R^*, the composite linguistic model composed of $G(R_{Q(t)})$ which is a matrix representing state membership functions. Z_C is the crisp form of the output generated by performing defuzzification (*DF*) on Z_F. U_B represents the crisp output of the system and is a function of the Boolean transformed fuzzy inputs, X_B, the crisp inputs, W_B, the present state, $Q(t)_B$, and an analog input X_A which is passed through a threshold comparator to make X_T. X_B and Z_B are based on a Fuzzy-to-Boolean transformation algorithm, the B algorithm [6], to map a change in X_F and Z_F, respectively, into state changes in a finite set of corresponding Boolean variables. Finally, the next state of the system, $Q(t+1)_B$, is a function of X_B, W_B, Z_B, $Q(t)_B$, and X_T, which represent either directly or indirectly all system inputs as well as the current system state.

1.2.1 The B Algorithm

The Fuzzy-to-Boolean transformation algorithm, or B algorithm, uses the Mean of Maxima (MoM) defuzzification method to find the average location of the maxima in a given universal space for a fuzzy set. The universal space represents all possible elements for a fuzzy input or output. For example, a temperature measurement in an automotive system may apply to a universal space of -40°C to 105°C. If the granularity of this universal space was 1°C then a fuzzy input for temperature would have a membership value for every integer element in the range of [-40, 105]. The MoM algorithm walks through each value in the universal space, locates the maximum value, and then determines the average offset of all maxima. For example, if the only maxima of a normalized universal set were located at 78°C, 79°C and 80°C then the resulting MoM would be 79°C. This is illustrated conceptually in Figure 2 as applied to a convex inferred output set.

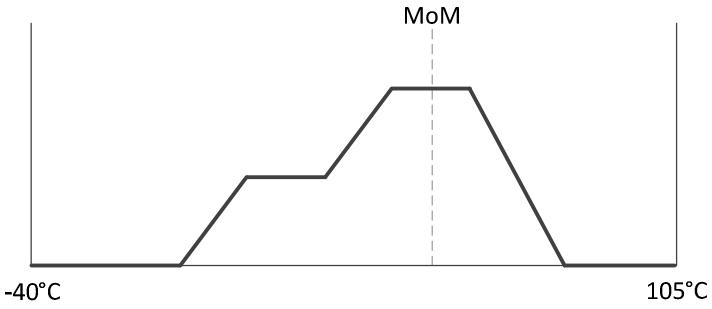

Figure 2. Conceptual MoM

The B algorithm continues by using the defuzzified input to determine which linguistic label applies to the input value. For example, consider again the temperature input for which a membership function with a set of linguistic labels is shown in Figure 3.

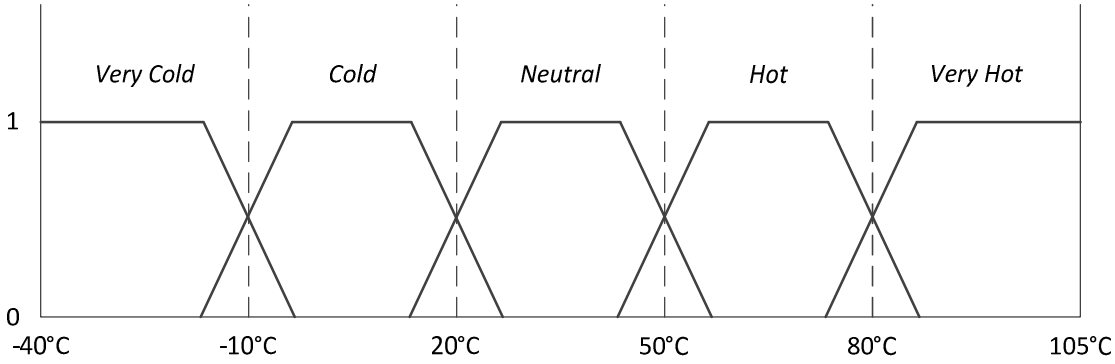

Figure 3. Example Linguistic Labels

A defuzzified temperature reading of 79°C would fall into the *Hot* category which can be represented multiple ways in binary logic. For example, if one bit is used for each linguistic label then the example reading could result in "00010". This value can then be used in conjunction with the current state and the result of the B algorithm from any other fuzzy inputs to determine the next state from a look-up table.

1.2.2 β Calculations

Another important concept of the HFB-FSM system is that of the β state and the β degree of membership. In a crisp FSM only a single state is allowed at any given time and the output of the FSM is dependent upon the mathematical relationships hard-coded in the state logic. The HFB-FSM allows for only one dominant state, similar to the crisp approach, but also allows for

multiple non-dominant states to be active to a varying degree specified by β. Each state is represented by an independent R_{Qi} rule memory that holds the relationship between X_F and Z_F and can be individually enabled or disabled. Rule memories can either be pre-defined if the system is known or populated through model building operations for complex or vague systems.

In addition to having β states, the HFB-FSM also includes a β degree of membership value that defines to what degree the non-dominant states are active. Consider again the previously introduced temperature example. A temperature reading of 65°C would clearly fall into the *Hot* linguistic label. A temperature reading of 79°C would still fall into the *Hot* linguistic label; however, being so close to the transition to *Very Hot* it may fall into the latter label to some degree. Considering this, it may be beneficial to consider any control actions that would have been executed had the temperature fallen in the *Very Hot* linguistic label. For the first example of 65°C, the β value would likely be zero since the temperature clearly falls in the middle of a linguistic variable. For the latter example the answer becomes fuzzier and β would likely take on a non-zero value since the temperature is approaching the next linguistic label. This means that as the input approaches a defined label boundary the β value increases to include the control actions resulting from β states to an increasing degree.

In the current state multiple fuzzy inputs may contribute to the conditions to move to the next state. For each of these fuzzy inputs individual β values are calculated. The overall β value for inference in the current state will be the min of these β values.

The remainder of this paper is concerned with implementing the HFB-FSM in a parameterized and flexible manner to allow for adapting the system to diverse applications. The implementation provides the system designer the flexibility to define and build a system using minimal parameters. The system is designed to support both Single Input Single Output (SISO) and Multiple Input Multiple Output (MISO) systems but, conceivably, the system could be modified to support a Multiple Input Multiple Output (MIMO) application.

2. Overview of the Implemented HFB-FSM System

The HFB-FSM is implemented in a Xilinx Zynq® XC7Z020 SoC utilizing the programmable logic (PL) portion of the device and one of the two available ARM® Cortex™-A9 MPCore™ application processors [9]. Each processor can operate at up to 667MHz and includes an L1 cache consisting of 32KB of instruction and 32KB of data per core in addition to 512KB of shared L2 cache. Additionally, each processor core includes a Floating Point Unit (FPU) and a NEON™ Single

Instruction Multiple Data (SIMD) co-processor. The final implementation is targeting an Avnet ZedBoard development kit which includes 512MB of DDR3 Synchronous Dynamic Random Access Memory (SDRAM) and Secure Digital (SD) card interface, 32MB of serial Flash memory, and multiple high-speed peripherals including a 1G Ethernet port [10].

The PL fabric is based on that of the Artix-7 series. The Artix family is the low-end of Xilinx's 7-series line of FPGAs; however, given the flexibility of having two processors and a sizeable PL the XC7Z020 still proves to be a powerful device.

2.1 System Overview

A block diagram of the implemented HFB-FSM system is shown in Figure 4.

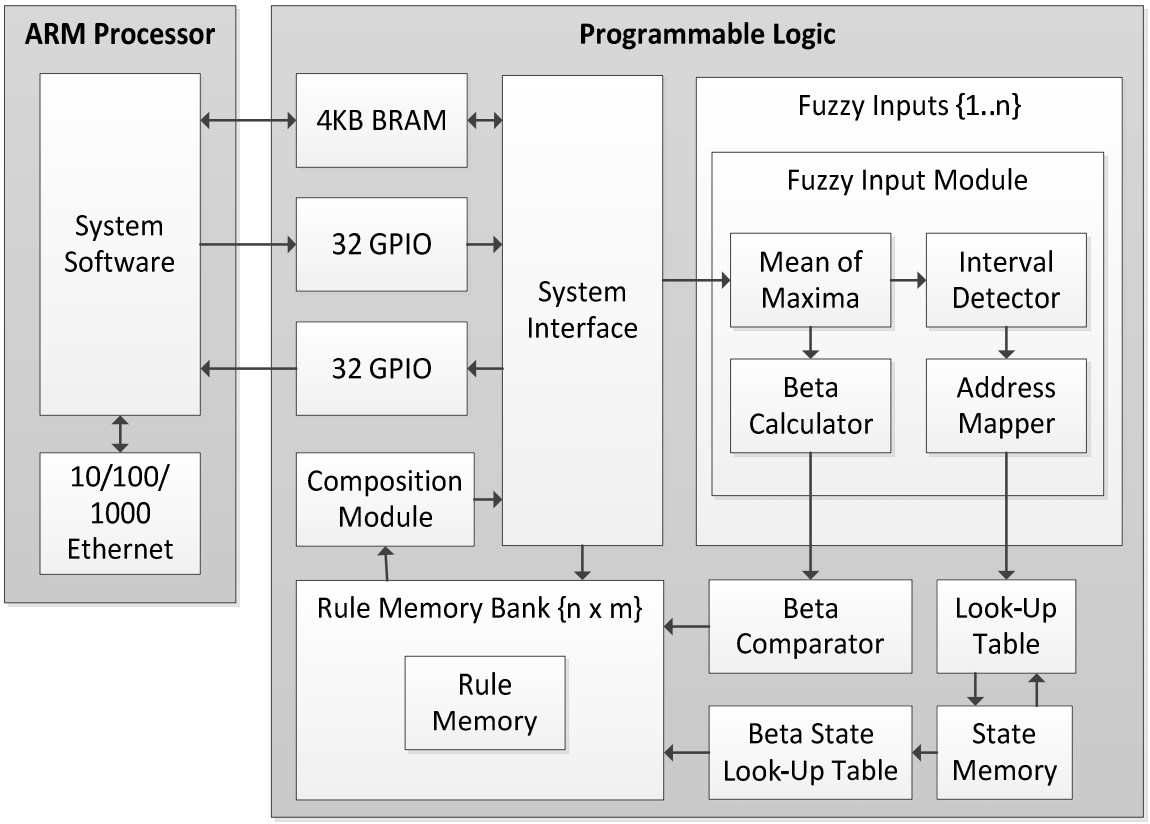

Figure 4. HFM-FSM Implementation Block Diagram

Commands and fuzzified data are received over a 10/100/1000 Ethernet interface using Transmission Control Protocol (TCP) and a custom command/response payload. The application processor receives and processes Ethernet packets and passes fuzzy data to a 4KB Block Random Access Memory (BRAM) shared with the PL. Commands are passed to the PL using a 32-bit

General Purpose Input/Output (GPIO) interface while PL status signals are sent back to the application processor through a separate 32-bit GPIO interface.

All PL interfacing to BRAM and GPIO ports is done through the System Interface module. When fuzzy input data for model building or inference is ready in BRAM the application processor signals with a command and the data is then read from BRAM and written to Rule Memory and/or the Mean of Maxima. The Mean of Maxima module is part of the Fuzzy Input module which is instantiated once for each fuzzy input. This module defuzzifies the inputs, determines the sub-interval where the defuzzified input lies, calculates a partial address for the Look-Up Table module, and determines the β value for that input. This data is calculated independently for each fuzzy input at the beginning of an inference operation and passed to subsequent modules for further processing.

Calculated β values from each fuzzy input inference operation are passed to the Beta Comparator in order to find the minimum β value for use by the Rule Memory. Partial address vectors are sent to the Look-Up Table where they are combined with the current state from the State Memory module to create the full address that is used to determine the next state from fixed initialization data. The next state is then passed to State Memory where it is stored and sent to the Beta State Look-Up Table in order to determine which rules will be activated for the inference operation.

Rule Memory stores the result of min/max composition of fuzzy inputs and outputs calculated through model building operations for use in fuzzy output inference. An array of Rule Memory modules with dimensions [*i*, *s*], where *i* is the number of fuzzy inputs and *s* is the number of states, is automatically generated at synthesis using parameters provided by the system designer. During model building the System Interface loads a fuzzy input and fuzzy output into one or more Rule Memories and controls all aspects of the operation. During inference a fuzzy input is loaded by the System Interface module then the output of the Beta State Look-Up Table is used to determine which R_Q models will be activated. Once the inference operation is complete the resulting fuzzy outputs are sent to the Composition Module for final min/max composition resulting in a single fuzzy output that is then read back into BRAM through the System Interface.

The unique aspect of this particular fuzzy processing system is the concept of the β state and value. β states are those states that share a transition to or from the current, or dominant, state

– in other terms, these states are near the current state. A β value is calculated to indicate the degree to which the outputs of β states will be considered during the final min/max composition operation. In the simplest sense, β values indicate how close the current input value is to a crisp state transition point. The result of this is a fuzzy system with an FSM that can activate multiple states simultaneously.

The PL system is parameterized to allow for scaling from a SISO system to a MISO system with up to 8 fuzzy and 8 crisp inputs. The size of the FSM is also scalable to up to 32 states. By changing only a handful of parameters and initializing the Look-Up Table and Beta State Look-Up Table modules the system can be completely reconfigured for different SISO or MISO applications.

Furthermore, a MIMO system with two fuzzy outputs could be realized provided the system designer implements the second application processor in the block design tied to a second instance of the HFB-FSM RTL.

3. Detailed Design Description

The following sections detail the implementation of each module utilized in the HFB-FSM RTL design as well as the software co-design.

3.1 Mean of Maxima Module Design Description

The Mean of Maxima module defuzzifies fuzzy inputs by finding the mean sample offset of all peaks within a sampled universal set. A running sum of the sample offset of all peak values is maintained along with the quantity of peak values located. Once an entire universal set has been sampled the running sum is divided by the peak count to obtain the result.

The flowchart shown in Figure 5 illustrates this modules behavior.

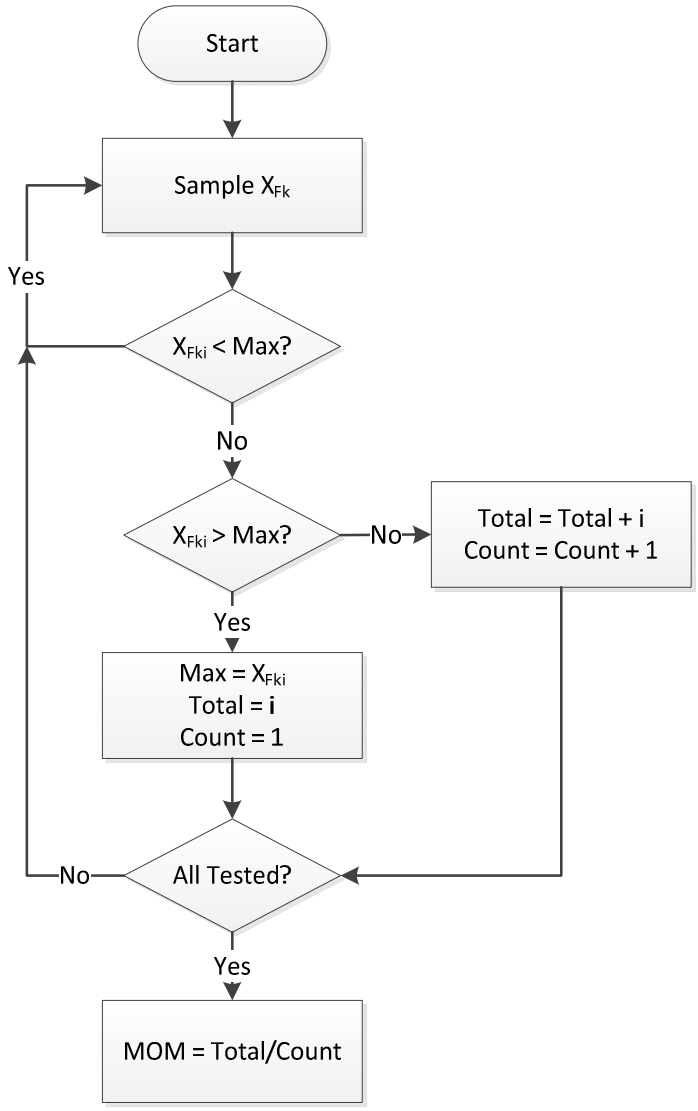

Figure 5. Mean of Maxima Flowchart

The process starts by sampling the fuzzy input X_{Fk}. The sample, X_{Fkj} is received along with the sample offset, *j*, which represents its location in the universal set. Once a sample is received it is compared to the running maximum, *Max*, which is set to zero by default. If the sample is less than *Max* then no further action is required. If it is equal to *Max* then the running maxima count, *Count*, is incremented and *j* is added to the running sum, *Total*. If the sample is greater than *Max* then the new value overwrites *Max*, *Total* is set to *j*, and *Count* is reset to 1.

The Mean of Maxima module is designed to continue searching for peaks until all samples in the universal set for a given fuzzy input are received. The universal set size is a configurable

parameter provided by the system designer prior to design synthesis. Once the entire universal set has been received the output is calculated as the quotient of *Total* and the dividend *Count*. The input universal space must be normalized to the range of (0, *Granularity*-1).

As an example, consider the universal set with a *Granularity* of 8 shown in Table 1.

Table 1. Example Mean of Maxima Input Data

Sample Offset (*i*)	-	0	1	2	3	4	5	6	7
Input Value	-	10	17	29	75	75	75	60	15
Max	0	10	17	29	75	75	75	75	75
Total	0	0	1	2	3	7	12	12	12
Count	1	1	1	1	1	2	3	3	3

At the start of the sequence *Max* and *Total* are set to 0 and *Count* is set to 1. At sample offset 0 the input value is greater than *Max*; therefore, *Max* is set to the input value, *Total* is set to the sample offset, and *Count* is set to 1. The same is true for samples 1 through 3. At samples 4 and 5 the input value is equal to *Max* so the sample offsets are added to *Total* and Count is incremented by one at each sample. For samples 6 and 7 the input value is less than *Max* so no values are updated. Once the entire universal set has been sampled the output is calculated as *Total*/*Count*. In this example, the output would be 4 indicating that the mean location of the peak input value is at sample offset 4.

The Mean of Maxima is capable of receiving one sample per clock period. Each sample is first verified to be within the pre-defined boundaries of the universal set, otherwise the sample is ignored. This is implemented to ensure that a system designer does not incorrectly define the fuzzy inputs or provide out of range values to the system.

The divide operation is implemented using a Xilinx Divider Generator Intellectual Property (IP) block. Division operations utilize a large amount of FPGA resources and result in long combinational delays. Implementing a division operation using IP simplifies design constraints and results in a more predictable design because the delay is fixed.

Once the output is calculated an output strobe signal is sent to the Interval Detector along with the calculated MoM. The total time taken to perform this calculation is dependent on the size of the universal set, referred to as *Granularity*, and how quickly the samples can be received. If

the module continually receives one sample per clock cycle and the total number of samples received per calculation is equal to the universal set then the propagation delay is (G*ranularity* + 10) clock cycles. This includes eight clock cycles for the divide operation to complete using the instantiated IP.

3.2 Interval Detector Module Design Description

The Interval Detector module receives the calculated MoM and determines the interval into which the input falls where each interval represents a linguistic label. Consider a fuzzy input for speed that is represented by the linguistic labels shown in Figure 6. These non-overlapping sub-intervals are used by the B algorithm to determine which linguistic label applies to a defuzzified input value. The actual membership functions for the linguistic labels overlap in the universal space.

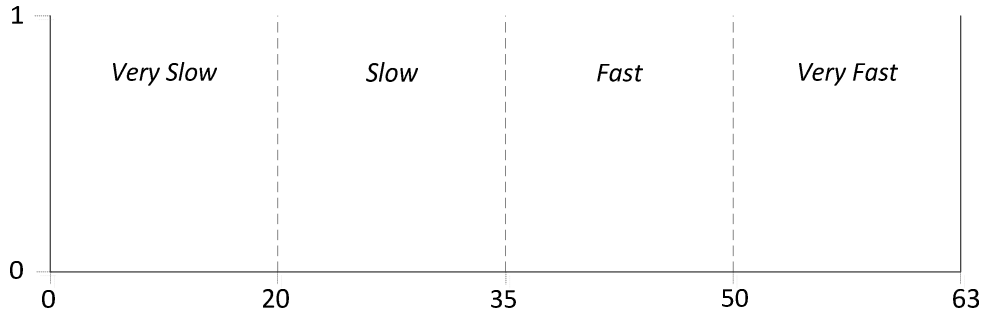

Figure 6. Example Linguistic Variables for Speed

A threshold comparator is instantiated for *c* = 3 boundaries at 20, 35, and 50. There is no comparator at 63 because any speed over 50 falls into the *Very Fast* linguistic label. Conversely, a comparator at 0 is not necessary because any speed under 20 is considered *Very Slow*.

Threshold comparators output a logic one when the input value is greater than or equal to the threshold it represents. By implementing a threshold comparator for each boundary and then concatenating the comparator outputs a single vector of size *c* that represents the highest interval into which the input falls.

Using the speed input as an example, the output of the system is a 3-bit binary vector. If the input value falls into the *Very Slow* interval then the output would be “000” because the value is less than 20, 35, and 50. If the input value falls into the *Fast* category then the output would be “110” because the value is greater than 20 and 35 but less than 50. The output of all Interval

Detectors can be used in conjunction with the current system state to determine the next system state.

It is noteworthy that this implementation differs from that originally proposed for the B algorithm. The proposed algorithm will output a '1' for a comparator only if the MoM value falls within that non overlapping sub-interval whereas this algorithm outputs a '1' when the value falls within or is greater than the non-overlapping sub-interval. This deviation was chosen to simplify the implementation.

The number of boundaries in the system, *Boundary Count*, can be uniquely set by the system designer for each fuzzy input but is limited to the respective fuzzy input's *Granularity*. Regardless of the number of boundaries, the output is calculated in a single clock cycle due to the simple and parallel nature of this module.

3.3 Beta Calculator Module Design Description

The Beta Calculator module calls a look-up table to determine the β value based on the calculated MoM. The β value represents how close an input value is to a linguistic label boundary and is used to limit the outputs of β state rule memories.

The β value reflects the overlapping nature of linguistic labels. As the MoM input value converges to a defined boundary the corresponding β value will increase to indicate such. Using the example from Figure 6, defuzzified input value of 19 would fall strictly into the *Very Slow* interval; however, with the value being so close to the boundary it can be conceived that it may fall into the *Slow* interval to some degree. In this case, *Very Slow* can be referred to as the dominant interval and *Slow* the β interval. The opposite would hold true if the value were 21. Figure 7 shows how β calculations are performed.

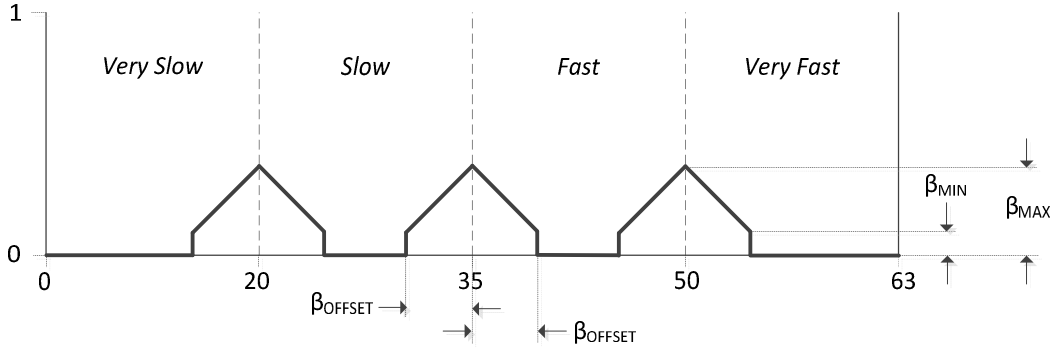

Figure 7. Graphical Depiction of β

The β value for the currently assigned dominant state, e.g. β_0, is always set to 1. Using the previous example, the state resulting from falling into the *Very Slow* interval would use β_0. However, a second β value, e.g. β_1, would be calculated for non-dominant states using the ramp function in Figure 7 because the MoM value falls near the boundary of the *Slow* interval. In the end, an array of unique β values are calculated, $\{B_1, \ldots, B_i\}$, using a separate ramp function for each input.

The system designer provides three factors for a fuzzy input which control how β is calculated, β_{MAX}, β_{MIN}, and β_{OFFSET}. These values represent the maximum β value, minimum β value, and the offset from each boundary where β is applied, respectively. Graphically, β_{MAX} represents the height of the triangle surrounding each boundary and β_{OFFSET} represents the width of the triangle on either side of each boundary. β_{MIN} represents the height of the triangle at β_{OFFSET} from each boundary.

Again using the example from Figure 6, although a defuzzified input value of 20 would fall into the *Very Slow* interval it would result in a β_i value of β_{MAX} for interval *Slow*. A value of 21 would likely result in a β_i value somewhere between β_{MIN} and β_{MAX} and a value of 10 would likely result in a β_i value of zero because it is far from any of the linguistic label boundaries.

A unique β_i value is calculated for each fuzzy input each time the Mean of Maxima processes a universal set. The resulting β_i values from all fuzzy inputs are later aggregated into a single value to be applied to all non-dominant rule memory outputs. The dominant rule memory will always use a β_0 value of 1 while the non-dominant rule memories will always use a value less than one – even if the system designer sets β_{MAX} to one.

β values are implemented as an 8-bit number – the same resolution as the fuzzy inputs and output. Using the three parameters described above as well as the input *Granularity*, which are provided by the system designer, a static look-up table is generated at synthesis that gives a β value as a function of sample offset. The equations for calculating Beta as a function of sample offset are given below. The equation used depends on whether the sample offset is below (8) or above (9) the *Boundary_Limit*. Note that $Boundary\_Limit_{ik}$ refers to the k^{th} value of the defined limits for input *i*.

$$\beta_i = \left\lfloor \frac{255*[\text{Sample\_Offset} - (Boundary\_Limit_{ik} - \beta_{OFFSET})]*(\beta_{MAX} - \beta_{MIN})}{\beta_{OFFSET}} \right\rfloor \quad (8)$$

$$\beta_i = \left\lfloor \frac{255*[(Boundary\_Limit_{ik}+\beta_{OFFSET})-\text{Sample\_Offset}]*(\beta_{MAX}-\beta_{MIN})}{\beta_{OFFSET}} \right\rfloor \quad (9)$$

A simplified algorithm for generating the β_i look-up table values for a single fuzzy input is shown below.

```
for i in (Granularity-1) downto 0 loop
   for j in Boundary_Count downto 1 loop
      boundary_v = Boundary_Limits (j);
      if (i >= (boundary_v-βOFFSET)) and (I < boundary_v) then
         delta_v <= i - (boundary_v - βOFFSET);
         βi      <= βMIN + (255 * (βMAX – βMIN) * delta_v) / βOFFSET;
      elsif (i >= boundary_v) and (I <= (boundary_v + βOFFSET))
         delta_v <= (boundary_v + βOFFSET) - i;
         βi      <= βMIN + (255 * (βMAX – βMIN) * delta_v) / βOFFSET;
      end if;
   end loop;
end loop;
```

The algorithm loops through all sample offsets in the universal set and checks if that value is within β_{OFFSET} of a defined *Boundary_Limit*. If so then the distance from the *Boundary_Limit* is calculated and used to scale the resulting value based on the offset of β_{MIN} and the slope of (β_{MAX} – β_{MIN}) / β_{OFFSET}. The process is repeated until the entire universal set has been traversed. If there are any overlaps in the boundaries then the *Boundary_Limit* with the lowest value will override any previously calculated values since the values are calculated from *Boundary_Count* down to 1.

The Beta Calculator module performs its operation in parallel with the Interval Detector module and takes two clock cycles to complete.

3.4 Address Mapper Module Design Description

The Address Mapper module receives the output of the Interval Detector module and compresses values into the smallest number of bits possible in order to efficiently use BRAM resources since not all values within the Interval Detector output vector can be realized. For instance, if there are three defined boundaries then there can only be four unique output values for the 3-bit output vector: “000”, “100”, 110”, and “111”. These four values can be compressed into a 2-bit vector to be used Look-Up Table module to determine the next state. The importance of this module becomes more apparent as the number of boundaries increases, as shown in Table 2.

Table 2. Mapped Address Bits vs Maximum Boundary Count

Mapped Address Bits	2	3	4	k
Maximum Boundary Count	3	7	15	$2^k - 1$

This operation is performed using a trivial state machine that, upon receiving an input, loops through the value from left to right counting the non-zero bits. Once a zero is found an output valid strobe is generated and the state machine returns to the idle state waiting for the next input.

The loop is implemented such that only one bit of the input is checked for each clock cycle. Although this method increases the propagation time of the AM module, this is done to preserve the maximum clock speed of the system. Implementing the loops with combinational logic could have a significant effect on the system clock speed or result in a limitation as to the number of boundaries. Because of this implementation, the Address Mapper takes *Boundary_Count* + 1 clock cycles to complete the mapping.

3.5 Fuzzy Input & Fuzzy Inputs Modules Design Description

The Fuzzy Input module instantiates and connects the Mean of Maxima, Interval Detector, Beta Calculator and Address Mapper modules to create a processing path for a single fuzzy input. The Fuzzy Inputs module then instantiates the Fuzzy Input module for each fuzzy input defined by the system designer. These modules only instantiate sub-modules and contain only minimal combinational logic; therefore, they do not add processing delays to the system.

The Fuzzy Inputs module uses a single generate statement to automatically instantiate a Fuzzy Input module for implementing the B algorithm for each fuzzy input. The code for this statement is shown below.

```
fuzzy_input_array : for i in NUM_FUZZY_INPUTS downto 1 generate
begin
       fim : fuzzy_input_module
       generic map (
           FIN                     => i
       )
       port map (
           reset_p                 => reset_p,
           mom_reset_p             => mom_reset_p,
           clk_p                   => clk_p,
           mom_input_data_p        => mom_input_data_p (i),
           mom_input_valid_p       => mom_input_valid_p (i),
           am_output_valid_p       => am_output_valid_p (i),
           am_output_p             => am_output_p (AMUB(i) downto AMLB(i)),
           bc_beta_value_p         => bc_beta_value_p (i)
       );
   end generate;
```

Input and output data are generally passed as arrays of values to make indexing simple. The exception is the output of the Address Mappers. The Address Mapper outputs are passed as a single vector that represents the concatenated output partial address values from each Fuzzy Input module. This vector will later include the values of all crisp inputs and the current state to form the full address for the Look-Up Table module. The arrays that hold the indices for each partial address, AMUB and AMLB, are calculated automatically based on the parameters provided by the system designer.

Two levels of modules are used in order to simplify the process of automatically generating a processing module for each fuzzy input. By creating a lower level module that instantiates the modules necessary to process a fuzzy input, it is trivial to create a statement in a higher-level module which generates the lower level module.

3.6 Look-Up Table Module Design Description

The Look-Up Table module determines the next state of the system given the current state and the outputs of the Address Mappers. The module essentially acts as an interface to a BRAM for storing the state machine transition table.

The construction of the look-up table (LUT) used is shown in Figure 8.

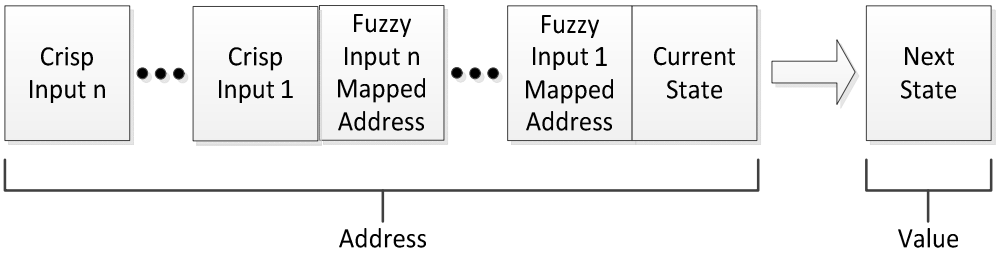

Figure 8. Look-Up Table Module Addressing Scheme

Once all fuzzy inputs have been defuzzified in the Mean of Maxima then passed through the Interval Detector and Address Mapper modules, the outputs are passed to the Look-Up Table where the next state is determined. An inferred BRAM LUT is used where the address is the concatenation of the Address Mapper outputs, the crisp system inputs, and the current state while the next state is the data held in BRAM. The values held in the LUT are static and must be provided by the system designer.

The system designer may not provide a state transition for every possible combination of inputs when initializing the LUT so the Look-Up Table contains logic to check for incomplete state transition information. These are denoted by the next state being all zeroes which is considered a reserved state. If a look-up operation results in an output of all zeroes then it is assumed that no information has been provided for this combination of input values and the system should remain in the current state.

Additionally, future work may add the ability to create new transitions in cases where an incomplete transition table is provided or where adaptive learning is needed. This implementation allows for a method to distinguish unassigned state transition conditions. The ability to add transitions during normal operation would, however, require adding write capability to the Look-Up Table module.

The Look-Up Table module can process a single request per clock cycle. Each request is read from BRAM on the first clock cycle where an input valid signal is detected. The data is then delayed one clock cycle to allow the LUT module to check for a reserved state entry. The output strobe signal is set two clock cycles after the input valid signal is received, regardless of whether a reserved state was found.

3.7 Beta Comparator Module Design Description

The Beta Comparator module takes the calculated β values from all fuzzy inputs and finds the minimum value. This value is then used in min/max composition for all non-dominant state rule memory outputs. Since β values are held at the output of the Beta Calculator module until a new value is available, this process is carried on continuously to simplify the module.

β values could be combined in a different manner if deemed necessary by the system designer. The minimum is chosen here in order to take a pessimistic approach and ensure that all fuzzy inputs agree there should be a non-zero β value.

The Beta Comparator cycles through the calculated β values from each fuzzy input at a rate of one value per clock cycle. When it reaches fuzzy input 1 the locally stored value is reset and the module output is updated with the value calculated from the prior loop. This means the output of the module will be valid somewhere between (i + 1) and 2i clock cycles after the input values are valid where i represents the number of fuzzy inputs.

3.8 State Memory Module Design Description

The State Memory module holds the current dominant state of the fuzzy state machine. When all fuzzy inputs have been processed and the next state has been determined by the Look-Up Table module, the new state is stored in State Memory. This state is then used by the Beta State Look-Up Table to determine which rule memories to activate and is also used to determine the dominant rule memory.

This module does not make a determination as to which state is the current state, it only holds the value. Upon receiving a valid signal at the input, the new current state is valid in one clock cycle.

3.9 Beta State Look-Up Table Module Design Description

The Beta State Look-Up Table module uses a small LUT to determine which rule memories are to be active based on the current system state. Contents of the LUT are statically set by the system designer and represent the possible state transitions to or from the current state. The current state is used as the address and the data is a vector containing one bit for each state in the system.

Consider the simple state transition diagram shown in Figure 9.

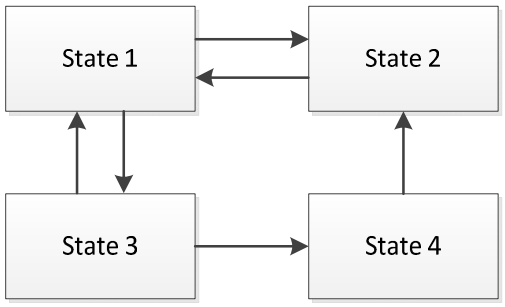

Figure 9. Example State Transition Diagram

This state machine would result in the look-up table shown in Table 3.

Table 3. Example Look-Up Table Entries

Current State (Address)	Active Rule Memories (Data)
1	0111
2	1011
3	1101
4	1110

The LUT contents are stored as a vector containing *Num_Fuzzy_States* bits arranged from (*Num_Fuzzy_States*:1). In state 1, the transitions exist to or from states 2 or 3 but not state 4. Therefore, the bit that represents the rule memory enable for state 4 is set to zero and the bits for states 1, 2, and 3 are set to one resulting in 0111. Similarly, when in state 4 the system can transition from state 3 and to state 2; therefore, the LUT value is 1110. An enable bit is set to one regardless of the direction of the state transition.

The output of this module is used to enable the dominant and all non-dominant Rule Memory modules during inference operations based on the current system state. The Beta State Look-Up Table waits for a rising edge on the valid signal coming from the State Memory module before performing a look-up. Upon receiving a rising edge signal a look-up is performed and the output is valid two clock cycles after a valid input is received.

3.10 Rule Memory Module Design Description

Model building and inference operations are implemented in the Rule Memory module. A 4Kx8 BRAM is instantiated and the relationship between input and output are built up through model building operations. Inference operations then make use of the stored rules to infer the output when provided only with input data. Rule memory is initially empty at start-up but is designed such that a base rule set could be provided. If no base rule set is provided then the system has no knowledge of the relationship between input and output at startup.

In a typical system, one or more inputs are provided and a well-defined state machine along with mathematical relationships is used to generate one or more outputs. By monitoring the inputs and outputs of this expert system and storing their relationship rule memory is capable of learning this relationship through model building without implementing complex mathematical relationships.

The algorithm implemented for model building is illustrated in Figure 10. The algorithm shown is for a SISO system. It has been shown that a MISO system can be decomposed into multiple SISO models and MIMO systems can likewise be decomposed into multiple MISO models [14]. Therefore, this SISO model building algorithm is the basis of more complex systems.

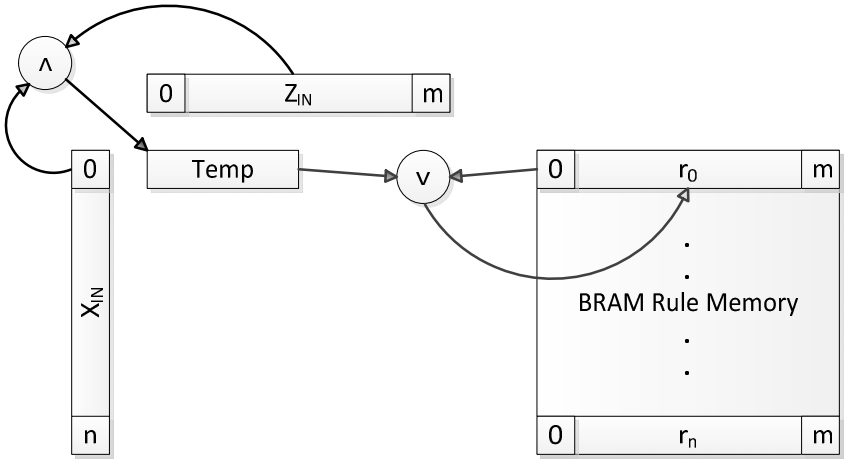

Figure 10. Model Building Algorithm

An input, X_{IN}, with (n + 1) 8-bit elements is provided along with an expected output, Z_{IN}, with (m + 1) 8-bit elements. The first elements of X_{IN} and Z_{IN} are sampled and compared to find the minimum value which is then stored in a temporary register. Then the first element in Rule

Memory is sampled and compared to the temporary register so that the maximum can be written back into Rule Memory. The comparison process then repeats for each remaining element of Z_{IN}. Once all elements of Z_{IN} have been sampled the next element of X_{IN} is sampled and the process is repeated. Model building completes when all (n + 1) elements of X_{IN} have been compared against all (m + 1) elements of Z_{IN}.

BRAM is shown as an (m + 1) x (n + 1) array of 8-bit elements. In practice, however, it may be challenging to implement such an array as there are limited BRAM resources within an FPGA and limits to how they may be utilized. In order to ensure efficient use of resources, BRAM is instantiated at a fixed size and elements are stored as shown in Figure 11.

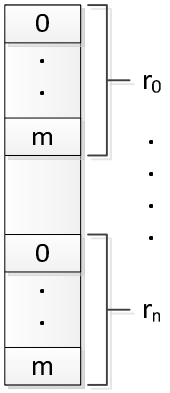

Figure 11. Element Storage in BRAM

Elements of rule memory are referenced by pointer registers that hold the offsets for the current input and output elements being computed. An element is addressed by concatenating the two pointer registers. Sample pointer registers are indexed from 0 to (*Granularity* – 1) meaning the registers can take on values between 0 and 63 since *Granularity* and *Output Granularity* can both take on a maximum value of 64. The address of the each relationship, r_{nm}, is given by $n * 64 + m$, regardless of the actual *Granularity* or *Output Granularity* settings.

Once the rule base has been sufficiently populated through model building it can be used for inference operations. There is no limit to the number of model building operations that can be completed, nor any minimum number of operations that must be completed, prior to performing inference. The system designer or user is expected to ensure that inference operations are not performed prior to establishing an adequate rule base.

Inference operations utilize the rule base held in BRAM to generate a fuzzy output when given only a fuzzy input. The algorithm for SISO inference is illustrated in Figure 12.

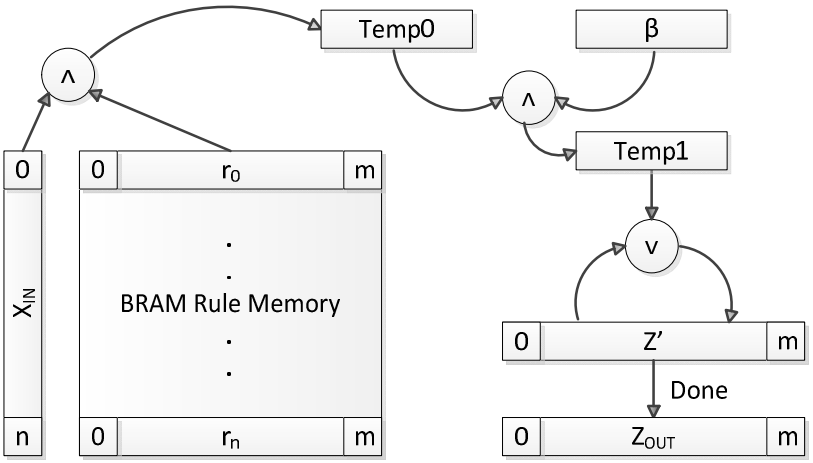

Figure 12. Inference Algorithm

An input, X_{IN}, with (n + 1) elements is provided to begin the operation. The first element of X_{IN} is sampled along with the first corresponding element of rule memory and the minimum value of these samples is placed in temporary register. Next, the minimum value of the first temporary register and the current β is found and placed in the next temporary register – this limits the maximum value a non-dominant state can obtain to β. For dominant states the β value will be set to 256 (equivalent to 1.0) such that it will have no effect.

Finally, the maximum value of the second temporary register, Temp1, and the Z' temporary register is placed back in Z', which is initialized to all zeroes at the beginning of inference. The next element of rule memory is then sampled and compared to the same element of X_{IN} through the same process. Once each element of the r_0 has been sampled the next element of X_{IN} is sampled and the process is repeated until all (m + 1) elements of all (n + 1) rule memories have been compared to an element of X_{IN}.

Once all elements have been traversed, the Z' register is transferred to the Z_{OUT} register and an output valid signal is set. The result of inference can then be read one sample at a time by the composition module which determines the final output of the system.

Processing in the Rule Memory module is the lengthiest operation of the HFB-FSM PL design due to the nature of fuzzy model building and inference. The processing time required for either a

model building or inference operation depends on the universal set size of the X_{IN} and Z arrays. If *n* and *m* represent the highest indexed element of X_{IN} and Z, respectively, then the total number of clock cycles required for model building is approximately $[(n + 6) * (m + 2) + 2]$. For inference, the total number of clock cycles needed is approximately $[(n + 6) * (m + 3) + 3]$.

3.11 Composition Module Design Description

The Composition module uses min/max composition to combine the result of inference from all active rule memories resulting in a single Z_{OUT} array of (*m* + 1) elements. Each element from each rule memory is combined one at a time until all elements have been sampled. This implementation results in a higher latency than could be achieved through other methods; however, it reduces the module's resource requirements and makes it flexible to allow for implementing more complex fuzzy operations without adding significant PL resources.

Figure 13 illustrates how SISO inference results are combined to form a MISO system. This algorithm is implemented by the simple finite state machine shown in Figure 14.

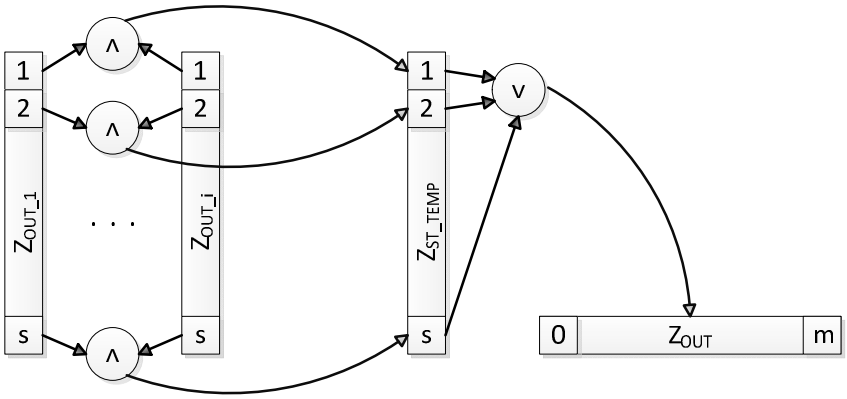

Figure 13. Composition Algorithm

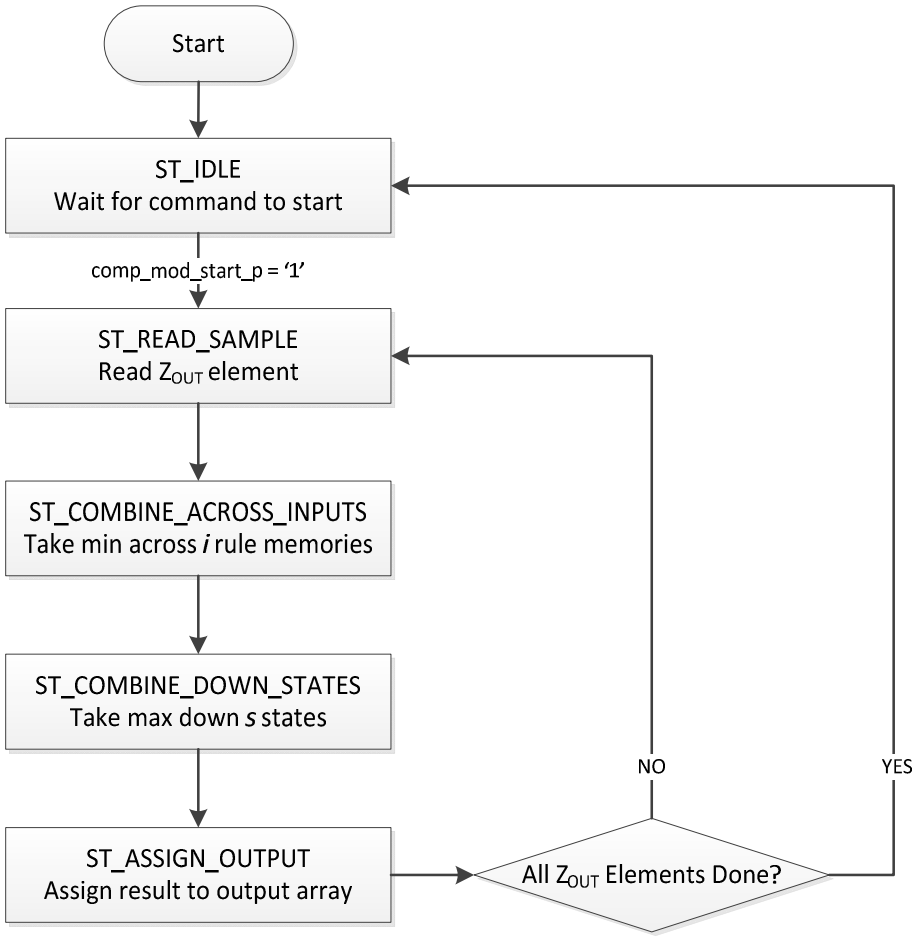

Figure 14. Composition Module State Diagram

When the rule memories complete an inference operation they assert an output valid signal that is registered by the Composition module. The System Interface module then signals for composition to start by setting the comp_mod_start_p signal high.

The process starts by reading in the first of (m + 1) samples of Z_{OUT} from each active rule memory in ST_READ_SAMPLE. Only those rule memories that were active during the inference operation are read; however, all rule memories are still traversed during composition with the inactive rule memory samples being ignored.

Recall that there are $i * s$ rule memories where i is the number of fuzzy inputs and s is the number of states. The samples can be viewed as being arranged in a two-dimensional array with s rows and i columns. In state ST_COMBINE_ACROSS_INPUTS, the minimum is taken along each row and placed in a temporary register, Z_{ST_TEMP}. In state ST_COMBINE_DOWN_STATES,

the maximum of all elements in the Z_{ST_TEMP} register is found and then placed in the output array in following state. This process repeats until all (m + 1) elements of Z_{OUT} have been combined.

To further illustrate this algorithm, consider the data shown in Table 4 which represents the result of inference from all rule memories for the 0^{th} element of Z_{OUT}. In this example the system has been designed with two fuzzy inputs and 5 states.

Table 4. Example of Composition

		Fuzzy Input No.	
Active?	**State No.**	1	2
0	1	-	-
1	2	3	7
1	3	15	14
1	4	27	4
0	5	-	-

Z_{ST_TEMP}
0
3
14
4
0

Z_{OUT}	14

The first column shows which rule memories were active for the inference operation and indicates that rule memories corresponding to states 1 and 5 were inactive. Values from these rule memories will be ignored and the associated elements of Z_{ST_TEMP} will be zeroed. The rule memories associated with state 2 were active; therefore, the minimum of 3 and 7 is placed in Z_{ST_TEMP} element associated with state 2. Once Z_{ST_TEMP} is full then the maximum of all its elements is taken and placed in the Z_{OUT} array. The process then repeats for elements {1, 2, … ,m} of Z_{OUT}.

The composition process latency depends on the number of states, *s*, the number of fuzzy inputs, *i*, and the *Output_Granularity*, (m + 1). The number of clock cycles required to complete composition is estimated by [(m + 1) * ((s * i) + s + 2) + 1].

3.12 Top Module Design Description

The Top module ties together all aforementioned modules and instantiates a rule memory for each fuzzy input and each fuzzy state. Where necessary, parameterized arrays are used to

connect sub-modules such that no modifications need to be made if the number of fuzzy inputs or states changes.

The design of the Top module is trivial with the exception of instantiating rule memories and applying the current β value to each rule memory. Rule memories are instantiated using two nested VHDL generate loops. The outer loop runs from *Num_Fuzzy_Inputs* down to 1 and the inner loop runs from *Num_Fuzzy_States* down to 1.

A simplified code snippet below shows how rule memories are instantiated in the two-level *generate* loop. By passing the *Fuzzy_Input_Number* (FIN) the module reads the *Granularity* and other pertinent parameters for that fuzzy input from the package file.

```
input_rule_array : for i in NUM_FUZZY_INPUTS downto 1 generate
begin
    state_rule_array : for j in NUM_FUZZY_STATES downto 1 generate
        signal curr_beta1_s    : unsigned (8 downto 0);
    begin

        curr_beta1_s <= "100000000" when (curr_state_s = j) else ('0' & curr_beta_s);

    rm_arr : rule_memory
            generic map (
                FIN             => i,
                STATE_NUM       => j
            )
            port map (
                ...
                rm_enable_p     => rm_enable_p (i)(j),
                ...
                rm_ready_p      => rm_ready_s (((i-1)*NUM_FUZZY_STATES)+j),
                ...
            );

    end generate state_rule_array;
end generate input_rule_array;
```

Note that there are two methods used for array indexing in the port map. One uses two-dimensional arrays that are easier to follow and make the code more readable. The other method uses one-dimensional arrays that are more difficult to index but make the numbers easier to work with in code.

This generate statement also assigns the β value to all rule memories except those that represent the current state. A signal is instantiated for each rule memory which multiplexes either the current β value expanded to 9-bits or 0x100. Recall that β values are encoded as 8-bits. Here they are expanded to 9-bits to distinguish a dominant state Rule Memory. Only the current state will receive a β value of 0x100 and all others will receive the current β value calculated from a ramp function that can take on a maximum value of 0x0FF. Within the Rule

Memory module β values for non-dominant states are further limited to 0x0FE (254 or 0.996) to prevent the system designer from unintentionally creating multiple dominant states.

3.13 System Interface Module Design Description

The System Interface module is the top-level of the PL design and interfaces to the processor while controlling the processes of model building and inference. Control signals and crisp inputs are received from the processor over a 32-bit GPI port while status signals are returned to the processor through a separate 32-bit GPO port. A 1Kx32 BRAM is used to transfer the input and output data of model building and inference between the processor and PL.

Nearly all functions of this module are implemented in a single FSM. However, because of the size of the state machine it is split into two logical blocks shown in two figures. The first of these in Figure 15 shows the portion of the FSM which loads data from BRAM into rule memory or the Mean of Maxima modules.

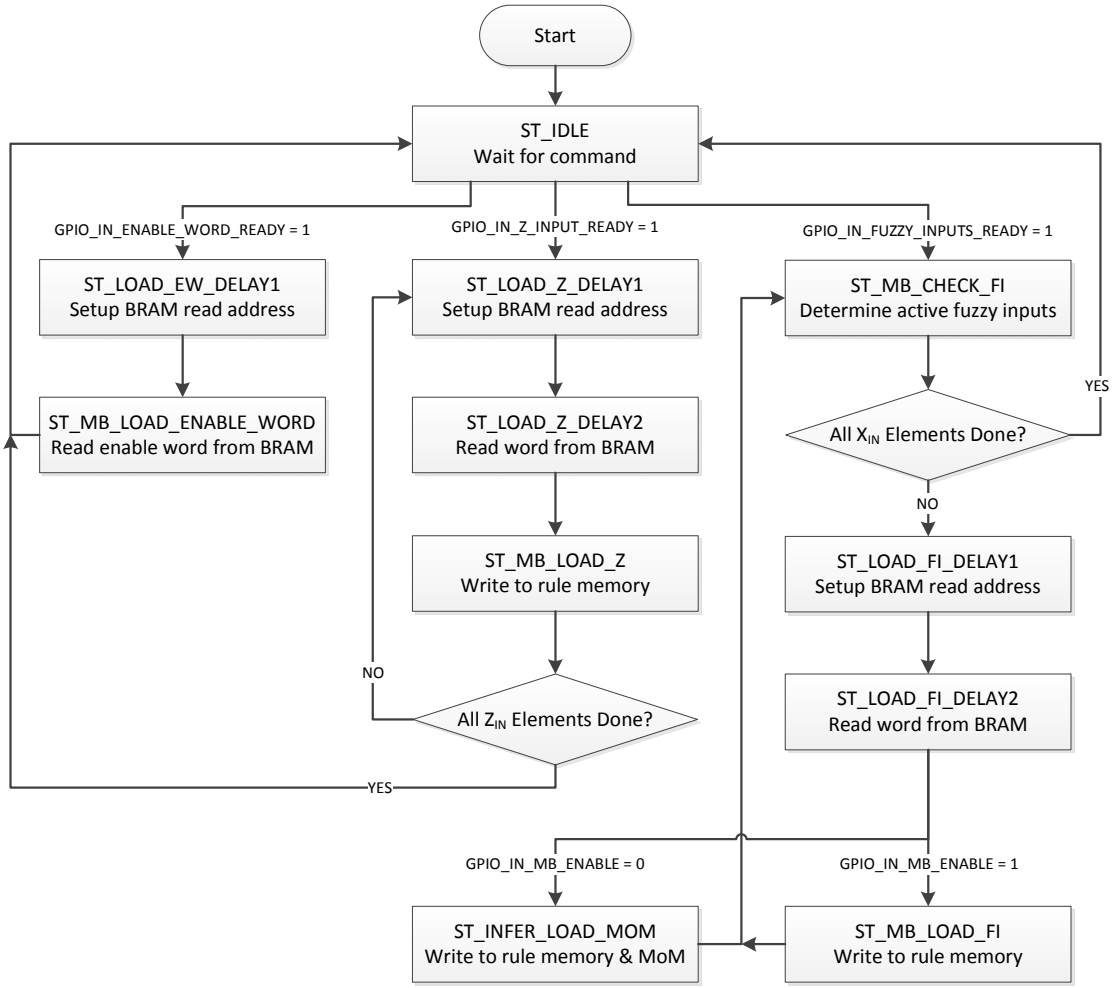

Figure 15. Data Loader Portion of System Interface FSM

From the ST_IDLE state, there are three possible paths for loading data depending on the type of data. The left-most is the path to load the enable word. This 32-bit word is used only for model building to define which state-associated rule memories will be activated. This word must be read prior to starting model building to ensure that only the Rule Memories intended for this operation are activated. For example, if the enable word is set to 0x00000001 then only Rule Memories associated with state 1 will be activated for model building. Since there may be multiple fuzzy inputs in a given system, exactly which modules will be enabled also depends on what X_{IN} data is loaded. If there are two fuzzy inputs and X_{IN} is loaded for fuzzy input 2 but not 1 then only the rule memory associated with fuzzy input 2, state 1 will be enabled for model building and all others will be disabled.

The middle path in Figure 15 shows the process for loading a Z_{IN} array for model building. The process begins by setting the BRAM address to the base Z_{IN} address and reading in the first 32-bit word. This word is then read out one byte at a time, representing one element of Z_{IN}, into all Rule Memories. The process repeats until *Output_Granularity* bytes have been written. The data is written to all Rule Memories in the system; however, only those enabled by the enable word are activated for model building. All others will have their input data reset as part of the model building process.

Loading the fuzzy input data is shown in the right-most path of Figure 15. The process is similar to loading Z_{IN} with a few exceptions. For model building, fuzzy input data is loaded into only the Rule Memories. For inference operations fuzzy input data is loaded into the Mean of Maxima and all Rule Memories. Additionally, there may be more than one fuzzy input to the system and it is likely that the inputs will not share the same data. Therefore, loading fuzzy input data requires tracking which inputs are being loaded and ensuring that model building and inference are only performed for those active inputs. This is done automatically by setting a local signal when a fuzzy input has been loaded and clearing that signal after model building has completed.

The remainder of the system interface FSM is shown in Figure 16. This portion of the FSM controls model building and inference operations and writes the result of inference to BRAM.

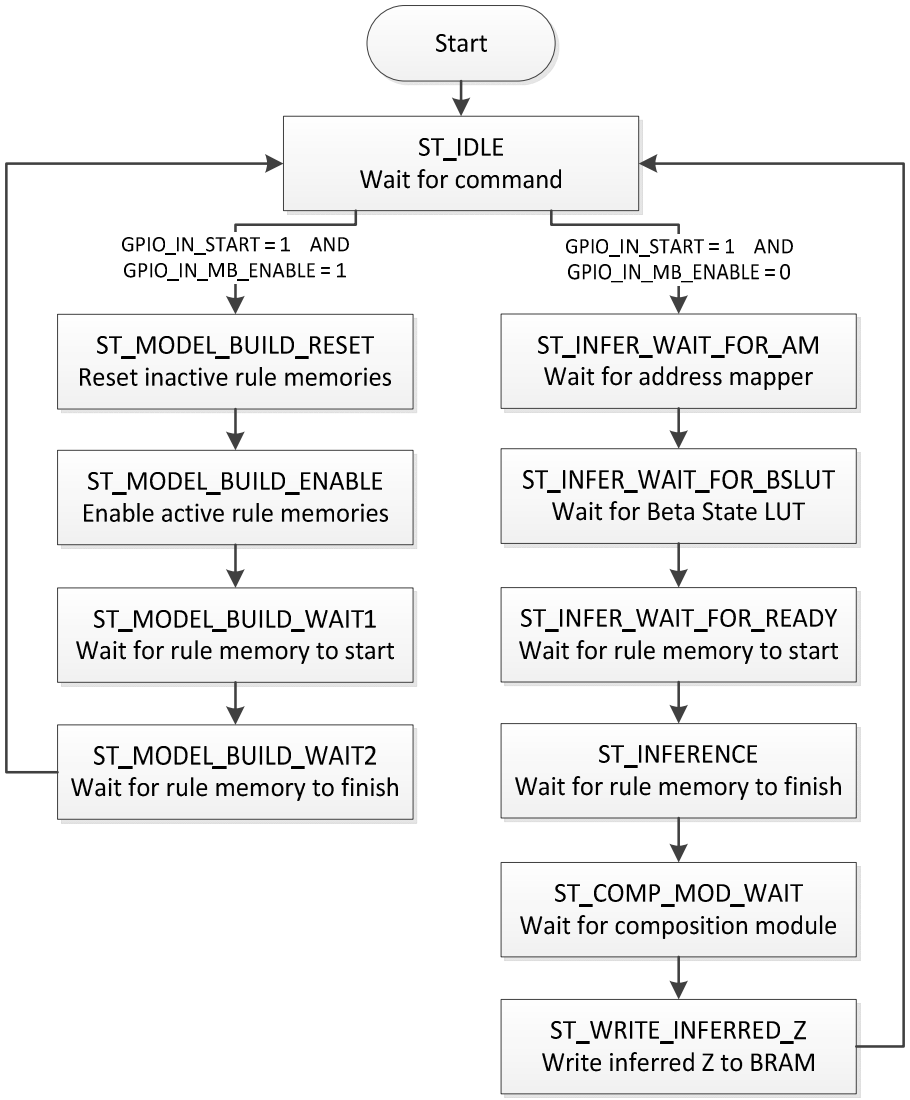

Figure 16. Control Portion of System Interface FSM

Model building and inference operations are enabled by the processor system through two GPI signals - Start and Model Building Enable. The Model Building Enable signal is sampled when the Start signal is set high. If it is sampled high then the system checks that a valid Z_{IN}, X_{IN}, and Enable Word have been received before starting a model building operation. If any of these inputs has not been received then the Start signal is simply ignored. If Model Building Enable is sampled low then the system checks that at least one fuzzy input has been received before starting inference. Again, if no fuzzy inputs are received then the request is ignored.

Model building operations start by building the reset signal used to clear Z_{IN} and X_{IN} data from Rule Memories not enabled by the Enable Word. The next step is to start the operation by setting Model Build Enable high and enabling the active Rule Memories.

Rule Memory provides a status signal to indicate that a model building or inference operation is underway. This signal is sampled by the system to determine whether the desired operation has started. Once this signal indicates it has started the system waits for the signal to indicate that the operation is complete before returning to an idle state and clearing internal status signals.

Inference operations start when the fuzzy inputs are loaded into the Mean of Maxima and Rule Memory modules simultaneously. The Mean of Maxima calculates the output and passes the value to the Interval Detector and subsequently the Address Mapper once a full universal set has been received. Upon receiving the Start signal, the System Interface waits for valid signals for all active Address Mapper modules before enabling the Look-Up Table module to determine the next system state. The Look-Up Table module transfers the next state into State Memory then enables the Beta State Look-Up Table. Once completed, its output will determine which Rule Memories should be enabled for inference. The System Interface module samples the Beta State Look-Up Table output and simultaneously resets all inactive rule memories while enabling all active rule memories.

Similar to model building, the system must wait for the inference operation to start and then wait again for operation to complete. Once completed, the Composition Module is enabled to combine the outputs of all active Rule Memories into a single Z_{OUT} array. The result of composition is then read into BRAM to be transferred to the processor and the system returns to ST_IDLE.

A total of 64 signals are provided for control and status signals between the processor system and the System Interface module. Many of these signals are currently unused but can easily be used for system debugging or growth purposes. The usage of these signals is summarized in Table 5 and Table 6.

Table 5. GPI from Processor System

Bit	Name	Description
0	GPIO_IN_MB_ENABLE	'1' = Model building, '0' = Inference
1	GPIO_IN_START	'1' = Start model building or inference operation
2	GPIO_IN_Z_INPUT_READY	'1' = Z_{IN} ready to load
3	GPIO_IN_FUZZY_INPUTS_READY	'1' = X_{IN} ready to load
4	GPIO_IN_ENABLE_WORD_READY	'1' = Enable word ready to load
7:5	Reserved	Reserved
15:8	GPIO_IN_FUZZY_IN_READY_BASE	'1' = X_{INn} to X_{IN1}, respectively, ready to load
23:16	GPIO_IN_CRISP_INPUTS_BASE	Crisp input n to 1, respectively
31:24	Reserved	Reserved

Table 6. GPO to Processor System

Bit	Name	Description
0	GPIO_OUT_RULE_MEM_READY	'1' = All rule memories are ready
1	GPIO_OUT_XIN_FULL	'1' = Rule memory X_{IN} loaded
2	GPIO_OUT_ZIN_FULL	'1' = Rule memory Z_{IN} loaded
3	GPIO_OUT_Z_READY	'1' = System Interface Z_{IN} loaded
4	GPIO_OUT_FUZZY_INPUTS_READY	'1' = System Interface X_{IN} loaded
5	GPIO_OUT_INFER_DONE	'1' = Inference operation complete
6	GPIO_OUT_MODEL_BUILD_DONE	'1' = Model building operation complete
7	GPIO_OUT_EN_WORD_READY	'1' = Enable word loaded
23:8	Reserved	Reserved
29:24	Debug System Interface State	Debug output of current System Interface state
31:30	Reserved	Reserved

The workings of the System Interface module are complex and require interaction with every module in the system as well as the application processor and, ultimately, the Ethernet interface. This makes it difficult to provide an exact processing delay for this module; however, ignoring down-time due to processor interaction the delays can be estimated as follows.

Loading of X_{IN} and Z_{IN} depends on the *Granularity* or *Output*_Granularity of the system, respectively. Loading X_{IN} takes approximately [*Num_Fuzzy_inputs* * (4 + 1.75 * *Granularity*)]

clock cycles. Loading Z_{IN} takes approximately [3 + 1.75 * *Output_Granularity*] clock cycles. Loading the enable word takes 3 clock cycles. Performing model building or inference operations is largely dependent on the delays in the underlying modules – most notably the Rule Memory module. However, the added delay for model building and inference is 5 and 6 clock cycles, respectively. Finally, writing Z_{OUT} to BRAM results in a delay of *Granularity* clock cycles.

3.14 Software Design Description

The Zynq EPP provides two 800MHz capable ARM® Cortex-A9™ application processors connected to the PL fabric through individual 32-bit AXI slave ports. A block design created in Vivado Design Suite instantiates one of the processor interfaces as well as the AXI peripherals it will communicate with. The block design for this system is shown in Figure 17.

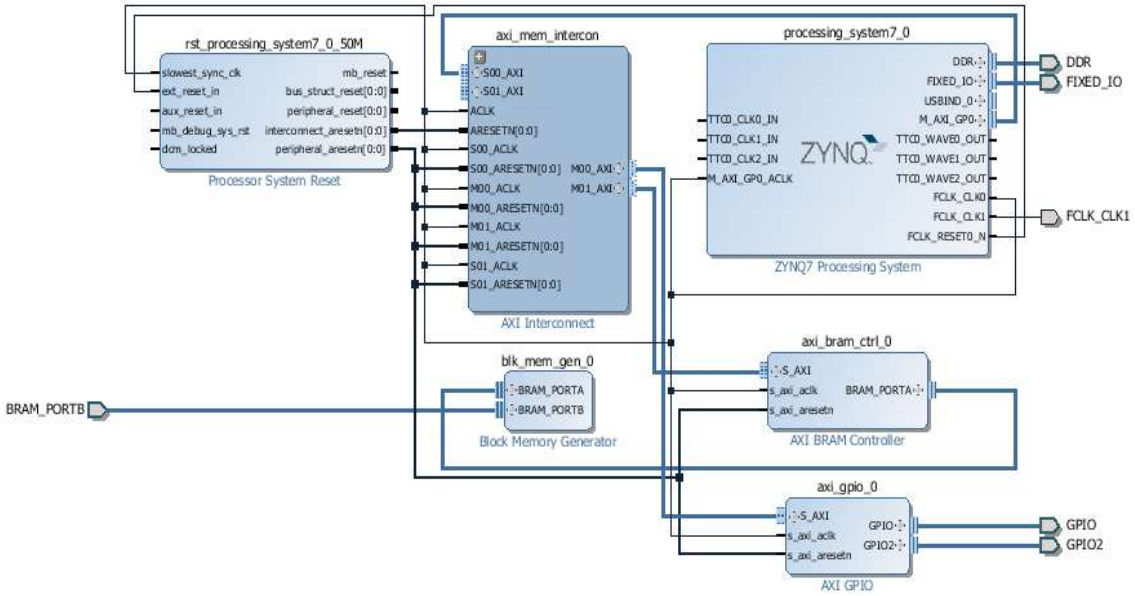

Figure 17. HFB-FSM Vivado Block Design

The processing_system7 block is the application processor and is shown connected to multiple IP blocks through an AXI bus. The rst_processing_system7 is somewhat standard in Zynq designs as it implements processor and peripheral resets. The axi_mem_intercon IP block is also common and acts as a bus bridge between the processor's master AXI port to multiple slave AXI ports.

The remaining IP blocks are instantiated specifically to support the HFB-FSM system. The axi_bram_ctrl block converts AXI transactions into BRAM control signals for reading and writing

BRAM. Connected to this module is a blk_mem_gen IP that instantiates the BRAM used for passing data between the application processor and the PL fabric. Finally, the axi_gpio block is an AXI to GPIO converter used as a two channel control and status signal interface to PL fabric.

A single clock is provided to the PL fabric through FCLK_CLK1. This clock is derived from the same Phase Locked Loop (PLL) used to clock the AXI bus. The design currently meets timing with a PL fabric clock of 100MHz and the AXI bus running at 50MHz.

3.14.1 Ethernet Command/Response Interface Design Description

A custom command/response protocol using TCP has been implemented to allow for sending commands with data for model building and inference operations. All system commands require a minimum of three payload bytes. The first byte is the magic word and is fixed at 0x87. The second byte is a command as defined commands in Table 7. The final required byte is the data count which can range from 0x00 to 0x40. The number of specified data bytes follows immediately.

Table 7. Ethernet Command Bytes

Command	Name	Description
0x01	Write Enable Word	Write 32-bit enable word – Includes 4 bytes of data
0x02	Write Z_{IN}	Write Z_{IN} – Includes *Output_Granularity* bytes of data
0x03	Write X_{IN1}	Write X_{IN1} – Includes *Granularity* bytes of data
0x04	Write X_{IN2}	Write X_{IN2} – Includes *Granularity* bytes of data
0x10	Read Z_{OUT}	Read Z_{OUT} – Include *Output_Granularity* bytes of dummy data
0x20	Start Model Building	Start model building operation
0x40	Start Inference	Start inference operation
0xAC	Dump BRAM	Dump BRAM to standard IO port
0xAE	Fill BRAM	Fill BRAM with random data

The application processor first checks that the magic word is present at the first data byte of any received packet. If the magic word is present then the payload is checked to verify it contains at least three bytes and the command word is compared against all existing commands. If a match is found then the payload is processed and the command executed.

The Ethernet interface operates at a fixed Internet Protocol (IP) address and the settings shown in Table 7.

Table 8. Ethernet Port Settings

Parameter	Value
IP Address	192.168.1.10
Net Mask	255.255.255.0
Gateway	192.168.1.1
Port	7

3.14.2 Software System Control Design Description

3.15 Configurable System Parameters

Several parameters must be provided in order to adequately describe a system. These parameters define the β function, the universal set size for both input and output, the number of fuzzy and crisp inputs, etc… The configurable parameters that must be provided by the system designer prior to synthesis are listed below.

Parameter Name	Num_Fuzzy_Inputs
Parameter Type	Natural
Parameter Range	1 to 8
Parameter Description	Number of fuzzy inputs to the system

Parameter Name	Num_Crisp_Inputs
Parameter Type	Natural
Parameter Range	0 to 8
Parameter Description	Number of crisp binary inputs to the system

Parameter Name	Num_Fuzzy_States
Parameter Type	Natural
Parameter Range	2 to 64
Parameter Description	Number of states in the fuzzy FSM

Parameter Name	Granularity
Parameter Type	Natural Array (*Num_Fuzzy_Inputs* downto 1)
Parameter Range	2 to 64
Parameter Description	Size of the universal set for each fuzzy input

Parameter Name	Boundary_Count
Parameter Type	Natural Array (*Num_Fuzzy_Inputs* downto 1)
Parameter Range	1 to 32
Parameter Description	Number of boundaries for each fuzzy input.

Parameter Name	Beta_Offset (β_{OFFSET})
Parameter Type	Natural Array (*Num_Fuzzy_Inputs* downto 1)
Parameter Range	0 to 32
Parameter Description	Point where β ramp function equals *Beta_Min*

Parameter Name	Beta_Max (β_{MAX})
Parameter Type	Real Array (*Num_Fuzzy_Inputs* downto 1)
Parameter Range	0.0 to 1.0
Parameter Description	Maximum of β ramp function for each fuzzy input

Parameter Name	Beta_Min (β_{MIN})
Parameter Type	Real Array (*Num_Fuzzy_Inputs* downto 1)
Parameter Range	0.0 to Beta_Max
Parameter Description	Minimum of β ramp function for each fuzzy input

Parameter Name	Boundary_Limits
Parameter Type	Natural Array (*Boundary_Count* downto 1)
Parameter Range	0 to GRANULARITY-1
Parameter Description	Division points between linguistic variables for each fuzzy input

Parameter Name	Output_Granularity
Parameter Type	Natural
Parameter Range	2 to 64
Parameter Description	Size of the universal set for the fuzzy output

4. Simulation Results

The following simulation images were captured for a system with two fuzzy inputs and 12 states. The system steps through 24 separate model building operations followed by a single inference operation. A testbench for the System Interface module is included with the project source code and includes a stimulus file name system_interface_data_full_sim.txt that contains X_{IN} data for model building and inference, Z_{IN} data for model building, and commands for performing these operations.

The testbench is run in a standalone project without the block design or application processor. Only the HFB-FSM RTL portion of the design is tested. The testbench can be run in the full project but requires commenting out the block design within the System Interface module.

4.1 Mean of Maxima Simulation Results

Figure 18 shows the first simulation results from the Mean of Maxima module.

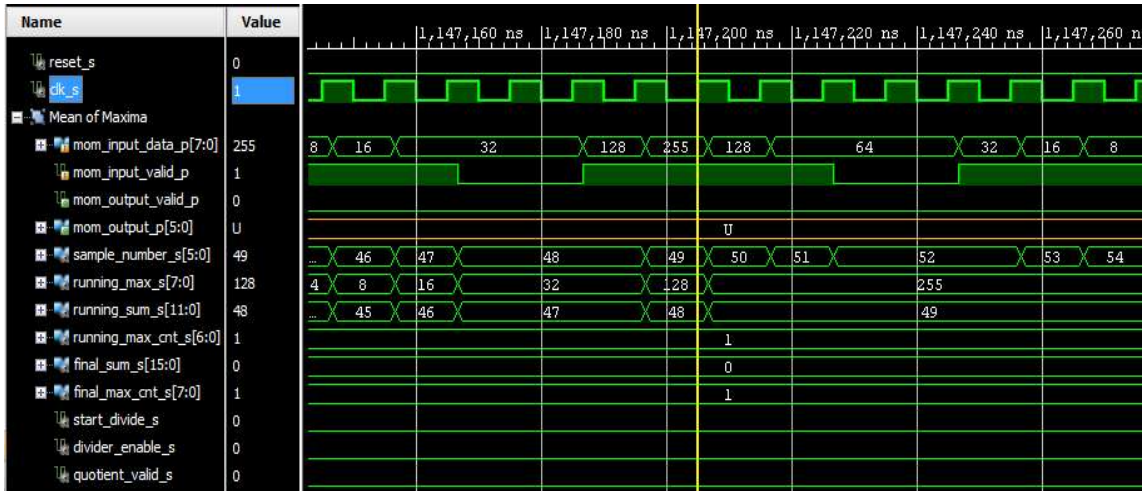

Figure 18. Mean of Maxima Local Maxima Found

Samples are being passed into the module each clock cycle that *mom_input_valid_p* is set high. An internal counter, *sample_number_s*, increments for each received sample to ensure only *Granularity* samples are received for a given calculation. When a value of 255 is received it becomes the new maximum causing *running_max_s* to be set to that value, 255, *running_sum_s* to be set to 1, and *running_max_cnt_s* to be set to the sample offset corresponding to the new maximum, 48.

Once all samples from the universal set have been received the signals *final_sum_s* and *final_max_cnt_s* are updated with the running values and the divider is started by setting *start_divide_s* high which consequently sets *divider_enable_s* high, as shown in Figure 19.

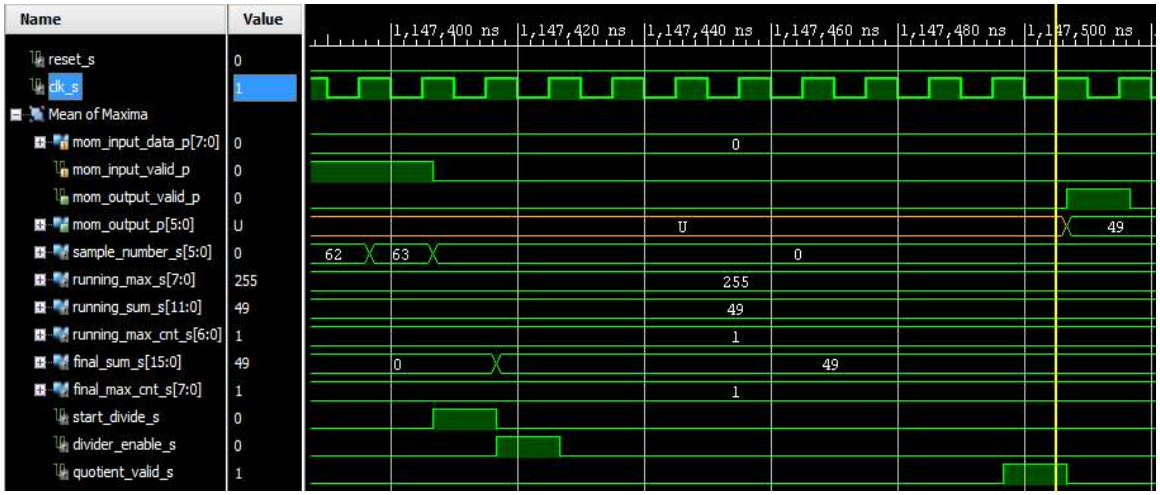

Figure 19. Mean of Maxima Divide Operation

The final output is available when *quotient_valid_s* is sampled high eight clock cycles after enabling the divider. This is a fixed delay in the divider IP block. The final output is signaled by setting *mom_output_valid_p* high and matches the expected value of $49/1 = 49$.

4.2 Beta Calculator, Interval Detector & Address Mapper Simulation Results

The output of the Mean of Maxima module is passed directly to the Beta Calculator as well as the Interval Detector followed by the Address Mapper. The simulation in Figure 20 shows an input value of 48 reaching the Beta Calculator and Interval detector simultaneously and the output of both modules updating and being ready on the next clock cycle. Using the output of Mean of Maxima and the system parameters outlined above, the expected β value can be calculated using (9) as shown below.

$$\left\lfloor \frac{255 * [(48 + 6) - 49] * (0.9 - 0.1)}{6} \right\rfloor = 170$$

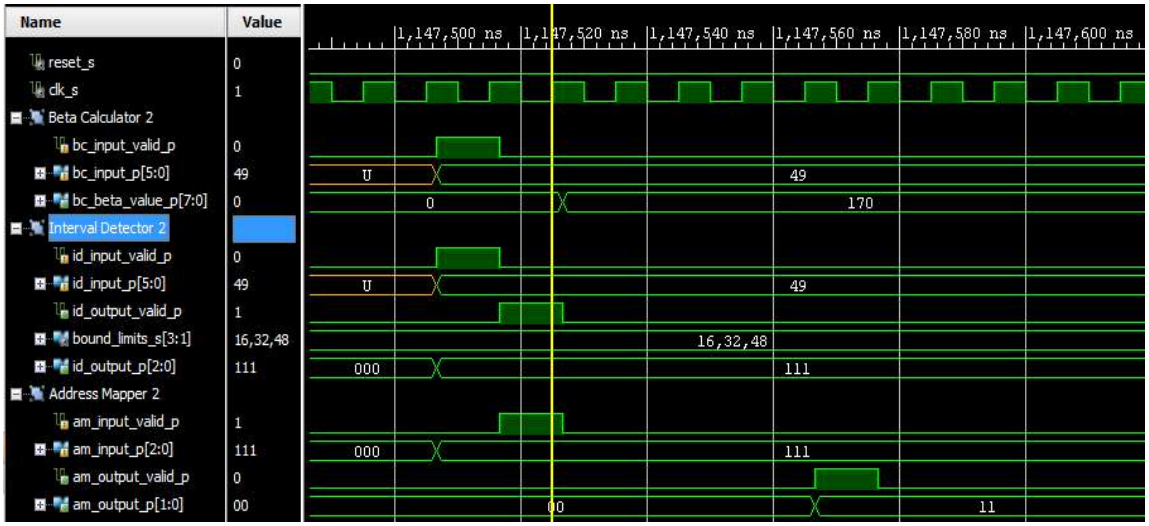

Figure 20. Beta Calculator, Interval Detector, & Address Mapper Outputs

The output of the Interval Detector is calculated as “111” because the input value, 49, is above all three defined *Boundary_Limits* of 16, 32, and 48. After this value is calculated the Address Mapper correctly outputs the two-bit address as “11” because there are three ‘1’s in the calculated interval.

4.3 Look-Up Table & State Memory Simulation Results

After the outputs of all Address Mappers are signaled as valid the Look-Up Table module is enabled to determine the next state, as shown in Figure 21.

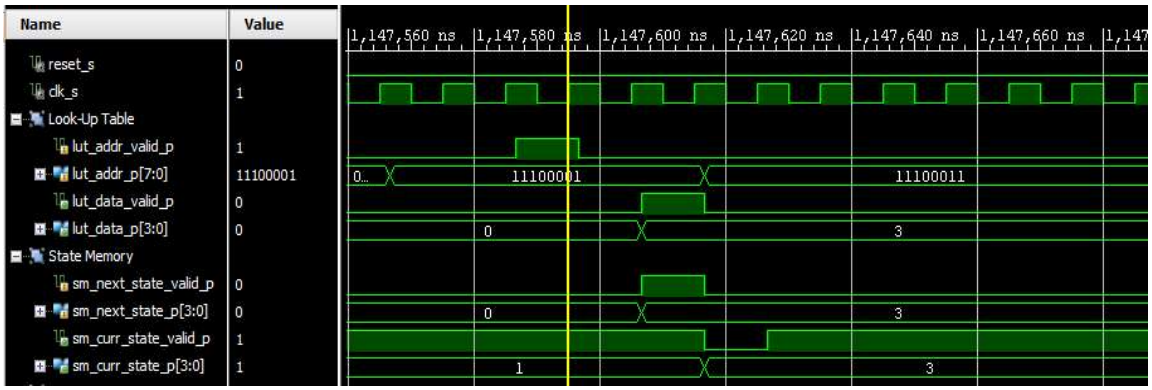

Figure 21. Look-Up Table & State Memory Outputs

The system starts in the *Default_State* which is set to state 1 in this example. The inputs from each Fuzzy Input module are concatenated along with the current state to form the address for the Look-Up Table. As shown in the previous simulation images, fuzzy input 2 gives a two-bit partial address of "11". Fuzzy input 1 gives a two-bit partial address of "10". When concatenated with the current state of "0001" the final address is "11100001" which results in the next state being set to state 3.

Once the Look-Up Table outputs the next state it is registered by the State Memory module which sets its output valid signal low for one clock cycle to indicate that a new state is available.

4.4 Beta State Look-Up Table & Beta Comparator Simulation Results

As previously described, the Beta Comparator continuously walks through β values from each fuzzy input and outputs the minimum value. Figure 22 shows a simulation of the Beta Comparator and Beta State Look-Up Table modules.

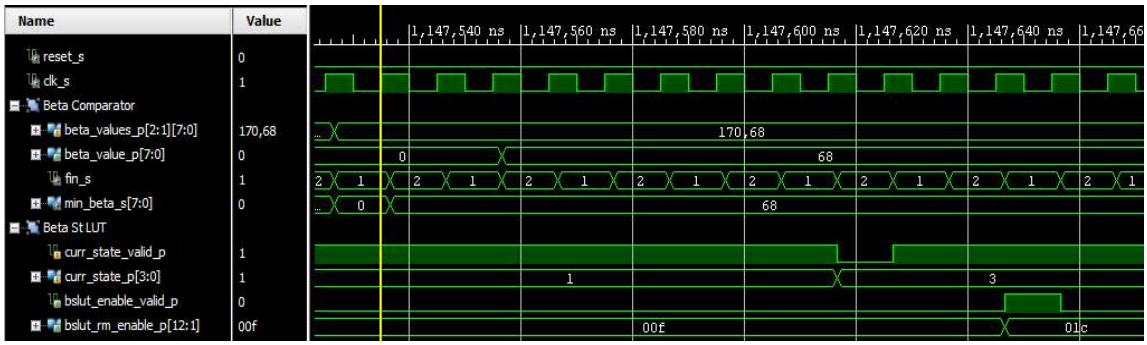

Figure 22. Beta State-Look-Up Table & Beta Comparator Output

Once the calculated β values from each fuzzy input are ready the output is calculated as the minimum of the two input values, 170 and 68 and is valid three clock cycles later.

The Beta State Look-Up Table takes the current state as the address to a LUT and outputs data that represents the enable signals for each state-associated rule memory. The output is a vector with *Num_Fuzzy_States* bits and ordered from (*Num_Fuzzy_States* : 1). For this example the states that share a transition with state 3, the dominant state, are states 4 and 5, the β states. Since the dominant state must be enabled in addition to any β states the enable vector should have bits (5:3) set to '1' and all others '0', resulting in 0x01C = "000000011100", which is correctly shown in the simulation.

4.5 Rule Memory Simulation Results

Figure 23 shows the first simulation image from the Rule Memory module.

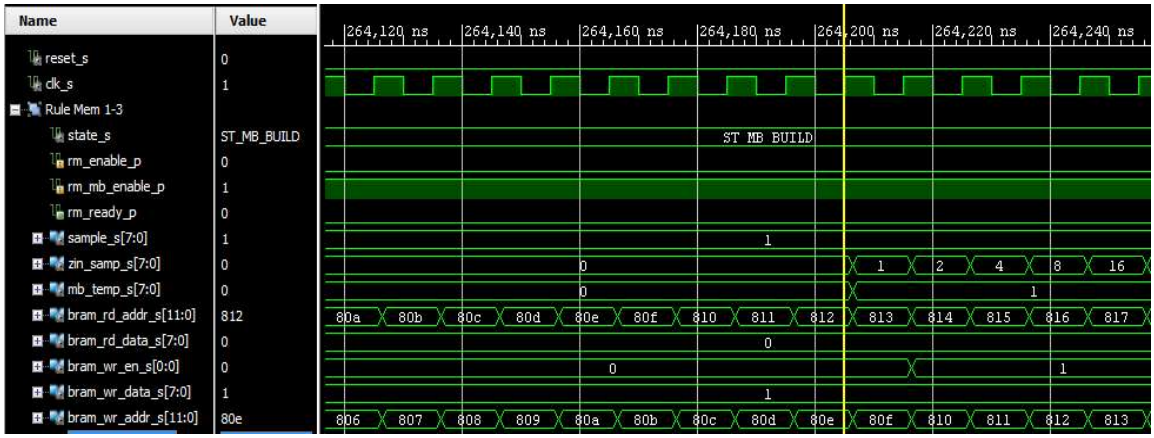

Figure 23. Rule Memory Model Building

The cursor marks a model building operation that did not result in a write to BRAM. First, the minimum of the current sample of Z_{IN}, shown as *zin_samp_s*, and X_{IN}, shown as *sample_s*, is found and stored in *mb_temp_s*. Next, *mb_temp_s* is compared to the corresponding value read from BRAM and written back to BRAM only if the value is greater. In this case the value of *mb_temp_s* was zero which matches what had been read from BRAM.

Shown at the next rising edge after the cursor in Figure 23 is a model building operation that does result in a write to BRAM. The sampled Z_{IN} is non-zero as is the sampled X_{IN} resulting in a non-zero value in *mb_temp_s*. The value read from the corresponding location in BRAM is zero; therefore, the value of *mb_temp_s* is greater and is written to BRAM.

The next simulation in Figure 24 shows a model building operation that overwrites previous model building data in BRAM.

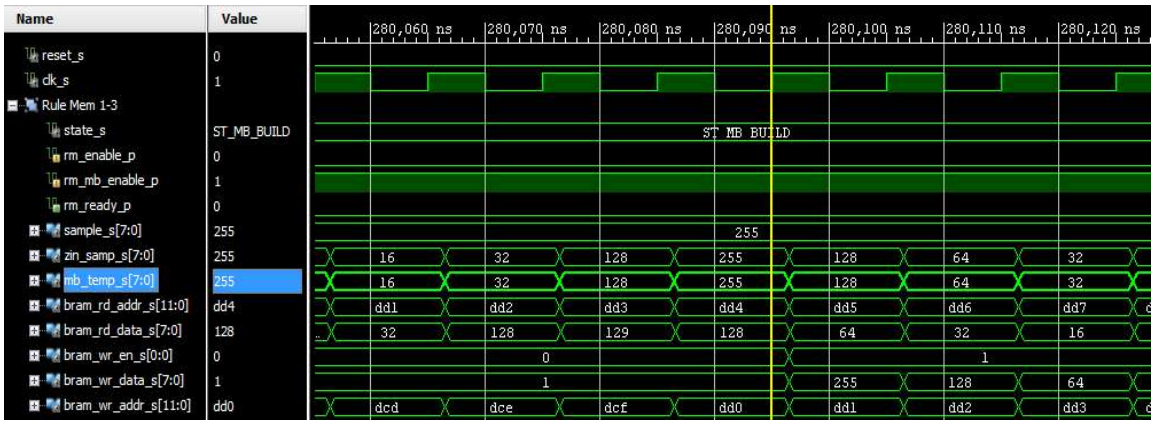

Figure 24. Rule Memory Model Building Overwriting

Both Z_{IN} and X_{IN} are sampled at a value of 255 resulting in *mb_temp_s* taking on a value of 255. This value is then compared with *bram_rd_data_s*, 129, and the maximum is written back to the corresponding location in BRAM on the next clock edge. It is important to note that the value of *zin_samp_s* shown in the simulations above is delayed by one clock from the actual value used by the system. Z_{IN} is sampled with a value of 255 during the same clock edge that *bram_rd_data_s* reads 129. The signal *zin_samp_s* is for simulation purposes only and will be optimized out during synthesis.

The next simulation image, shown in Figure 25, shows an inference operation in Rule Memory.

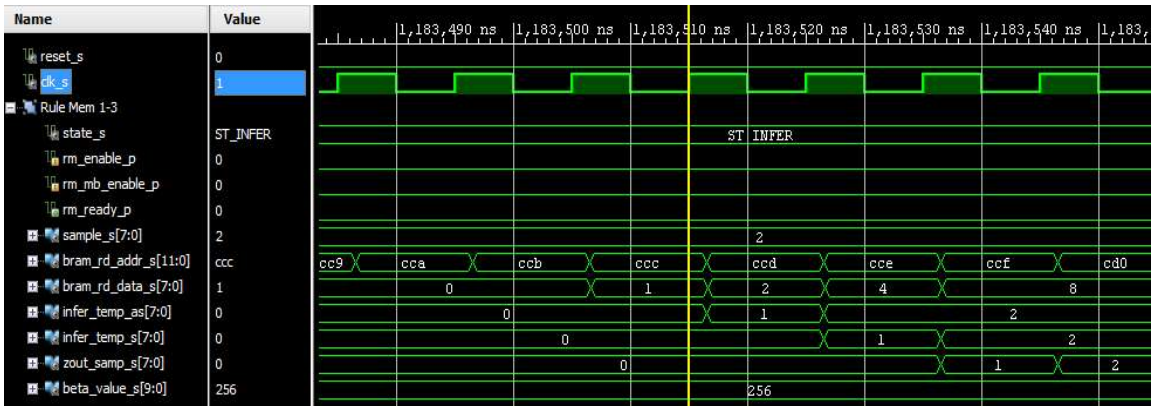

Figure 25. Rule Memory Dominant Inference Operation

First, at the shown cursor a sample of X_{IN}, *sample_s*, is compared with a corresponding value from BRAM, *bram_rd_data_s*, which has been built up through multiple model building operations. The maximum of these two values is stored in the signal *infer_temp_as*. On the next clock edge the sample of *infer_temp_as* is compared to the β value for this Rule Memory, *beta_value_s*, and the minimum is written to *infer_temp_s*. Finally, two clock cycles after the cursor the maximum of *infer_temp_s* and the current sample of Z_{OUT} is found and written back to Z_{OUT}. Note that in this case the Rule Memory in question represents the dominant state because it uses a β value of $256/256 = 1.0$.

The next simulation shown in Figure 26 captures the same inference operation inside a non-dominant Rule Memory. This simulation demonstrates the limiting of output values for non-dominant state rule memories through the β value.

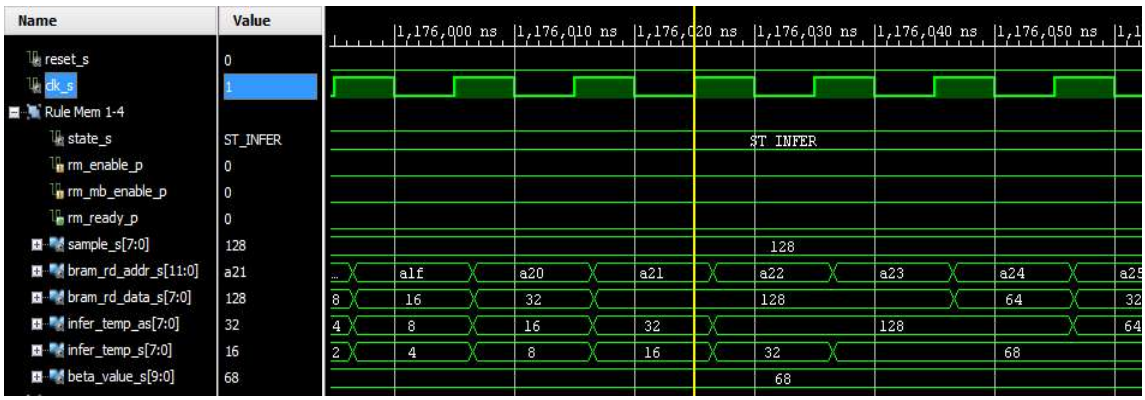

Figure 26. Rule Memory β Inference Operation

At the cursor, both the value of *sample_s* and *bram_rd_data_s* are 128 which results in *infer_temp_as* being set to 128. At the following clock edge *infer_temp_as* is compared to the current β value held in *beta_value_s*, 68, and the minimum is written to the corresponding element of Z_{OUT}. It is noteworthy that the β value of 68 matches the output of the Beta Comparator module from Figure 22.

4.6 Composition Module Simulation Results

Figure 27 shows the sequence of combining the outputs of all Rule Memory modules through min/max composition for a single element of Z_{OUT} in the Composition module.

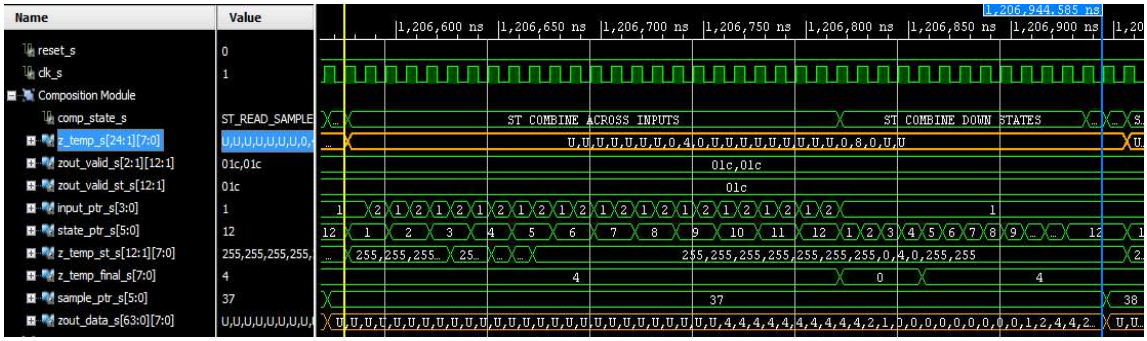

Figure 27. Composition Module Simulation

The yellow cursor to the left shows the start of composition for element 37 of Z_{OUT}. The process starts by requesting an element of the inferred Z_{OUT} from each rule memory. Note that "U" values are ignored because they come from inactive Rule Memory modules. The *zout_valid_s* signal shows that only memories that represent states 3, 4, and 5 are active for this inference operation.

While new samples are being received the module resets each element of *z_temp_st_s* to 255 and *z_temp_final_s* to 0. These signals are used to combine inferred Z_{OUT} across inputs through min composition and then down states through max composition, respectively. Once the samples are received the process begins by stepping through each Rule Memory and combining the Z_{OUT} for each state into the *z_temp_st_s* array through min composition. After all inputs have been processed through all states, the state machine transitions to using max composition to combine all elements of *z_temp_st_s* that were enabled for this inference operation into *z_temp_final_s*. After going through all of the states the value in *z_temp_final_s* is written to the output array and the module transitions to the next element of Z_{OUT}.

4.7 System Interface Module Simulation Results

Figure 28 captures a simulation of the System Interface module loading X_{IN} to the Mean of Maxima and Rule Memory modules.

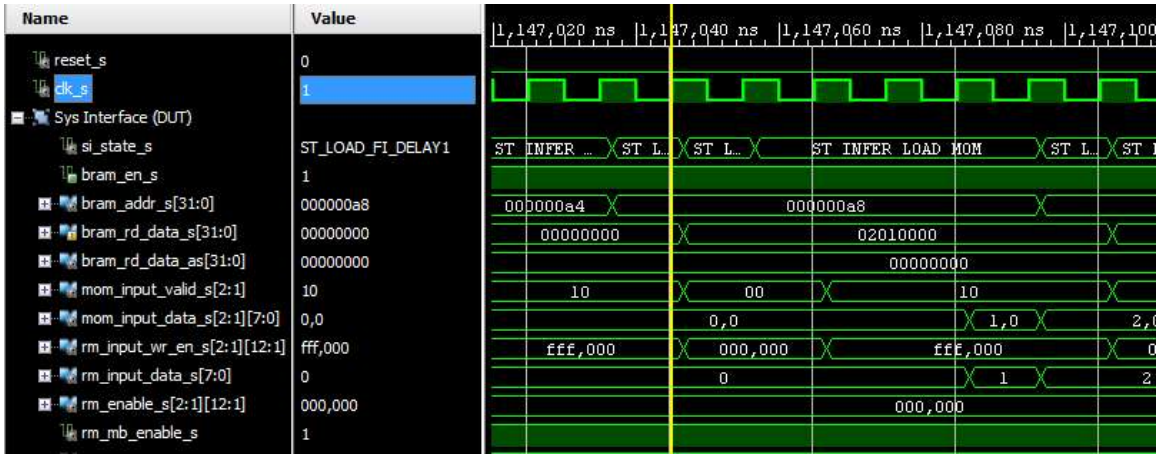

Figure 28. System Interface Simulation Showing X_{IN} Loading

The simulation shows a word which includes four elements of X_{IN} being read for BRAM and stored in *bram_rd_data_as*. This data is then written to both the Mean of Maxima and Rule Memory modules one byte at a time. The elements are written to only the modules associated with fuzzy input 2. Because the data is being written to the Mean of Maxima, this must be an inference operation as this module is not involved in model building. The data is written to all Rule Memories associated with fuzzy input 2 because at the time when the system is writing this input data the type of operation is not known. Furthermore, if the operation is inference then the active Rule Memory modules are not known until after the Beta State Look-Up Table has completed processing. Therefore, the simplest implementation is to write the data to all Rule Memories and clear the data from the inactive states prior to completing inference.

The process of loading Z_{IN} for model building is similar to loading X_{IN}; therefore, no simulation images are shown for loading Z_{IN}.

Reading X_{OUT} and loading the data into BRAM is shown in Figure 29.

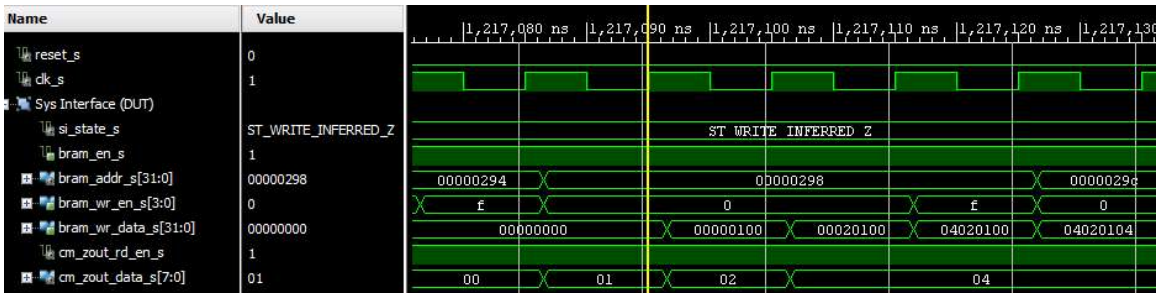

Figure 29. System Interface Simulation Showing Z_{OUT} Data Storage

Z_{OUT} samples are read from the Composition Module one sample at a time while a word is built up to write into BRAM. Once four bytes have been received the data is written to BRAM and the counter signal that points to the active *bram_wr_data_s* byte is reset. This process repeats until the entire Z_{OUT} array has been written.

4.8 Full System Simulation Results

The final simulation image in Figure 30 captures the high-level view of 24 model building operations followed by a single inference operation. Using parameters *Granularity* and *Output_Granularity* at their maximum values of 64 and simulated clock rate of 100MHz, which matches the actual implementation but is not reflected in the included simulation images, a model building operation takes approximately 48us and an inference operation takes approximately 73us. These delays do not include Ethernet transfer and processing time or delays in the application processor. Assuming software processing and the Ethernet interface are capable of keeping up with the PL, the HFB-FSM implementation is theoretically capable of performing approximately 13,700 inference operations per second.

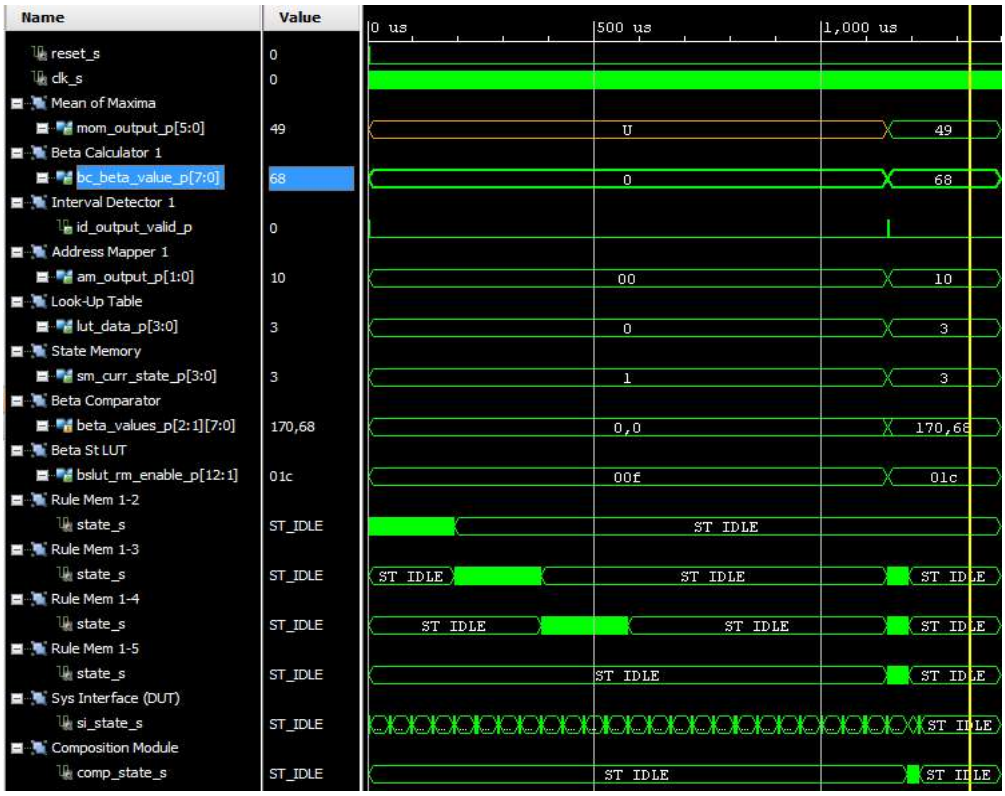

Figure 30. Full System Simulation

5. Application to Eye-Hand Coordination Intelligent Decision Support System

In order to demonstrate some key features of the HFB-FSM an example application was implemented based on a previously developed fuzzy system designed to assess children with eye-hand coordination disabilities [11]. In this system an individual is given the task to draw or trace labyrinths or letter patterns through the use of a haptic-feedback robotic device connected to a computer. After each task has been completed the time taken to complete the test and the accuracy of the drawing are assessed and used to determine the next task and ultimately the level and type of haptic feedback.

A simplified version of this system utilizes two fuzzy inputs, time-taken, Time, to complete a test and accuracy of the drawing, Accuracy, to generate a single fuzzy output that represents the helping force exerted by the robotic device. The FSM of the system consists of 12 states starting with state 1 and pausing the test when state 12 is reached. The state diagram for this system is shown in Figure 31.

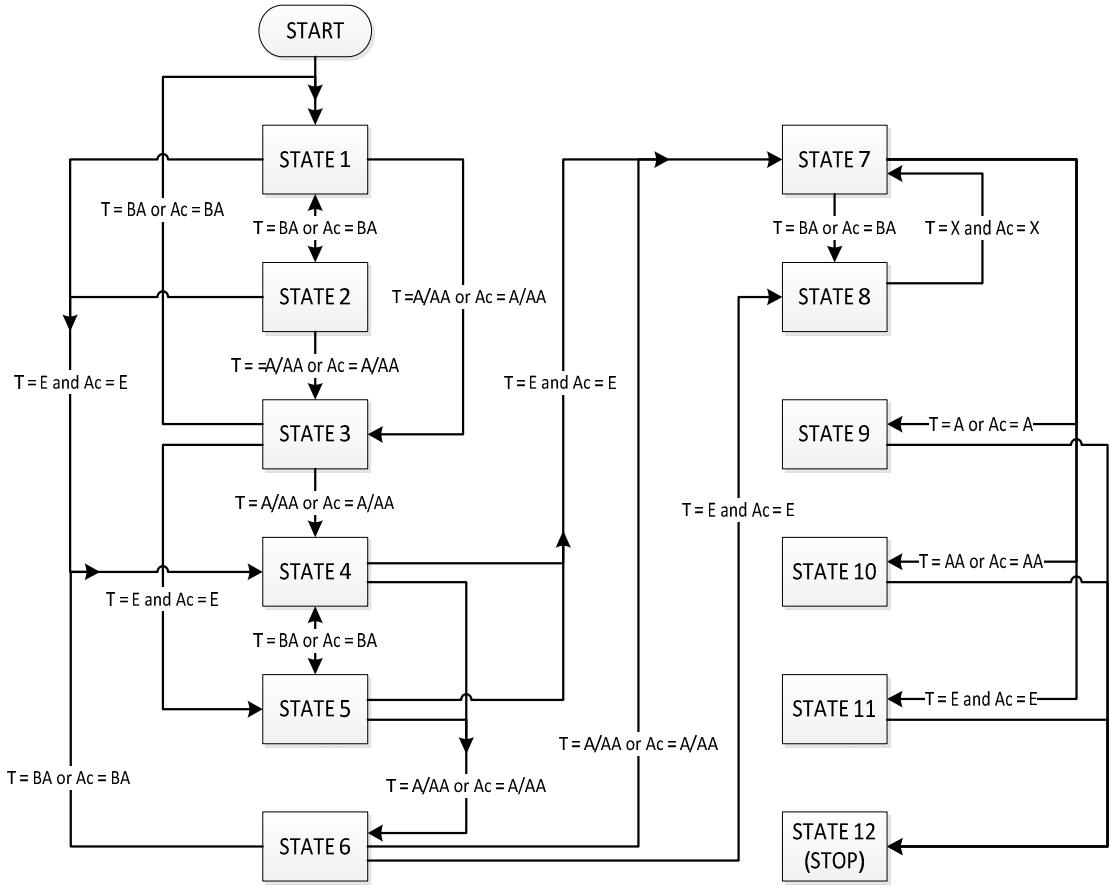

Figure 31. IDSS State Machine

In state 1 the patient is given a specified test to determine which state to take next based on the inputs. Time and Accuracy can take on one of four linguistic labels: Below Average (BA), Average (A), Above Average (AA), or Excellent (E). The range of values assigned for each linguistic label are shown in Table 9.

Table 9. Eye-Hand Coordination Assessment Linguistic Labels

	Accuracy (%)	Time Taken (sec)	Accuracy Normalized	Time Taken Normalized
Below Average (BA)	0 – 40	75 – 100	0 – 25	48 – 63
Average (A)	40 – 55	55 – 75	25 – 35	32 – 48
Above Average (AA)	55 – 70	25 – 55	35 – 44	16 – 32
Excellent (E)	70 – 100	0 – 25	44 – 63	0 – 16

In order to implement this system the HFB-FSM was first setup to accept two fuzzy inputs with 8-bit resolution and a *Granularity* of 64. An *Output_*Granularity of 64 was chosen to represent a single output, Force, that represents the level of haptic feedback provided through the robotic device.

The linguistic label thresholds based on Table 9 were normalized to the range of 0 to 63, giving the values shown in the two right-most columns. An initialization array was implemented for the Look-Up Table module to define the system state transitions using the current state and the defuzzified inputs. Input 1 is Time while input 2 is Accuracy. Two bits are needed to store the partial addresses of each defuzzified input since each has four linguistic labels. Four bits are needed to store the 12 FSM states giving a total of 8-bits for the address vector.

There are three additional input parameters to define the β values for each fuzzy input: $\beta_{ACCURACY}$ was simulated using $\beta_{MIN} = 0.9$, $\beta_{MAX} = 0.1$, and $\beta_{OFFSET} = 6$ while β_{TIME} was calculated using $\beta_{MIN} = 0.7$, $\beta_{MAX} = 0.3$, and $\beta_{OFFSET} = 5$. These values are automatically used in conjunction with the *Granularity* parameters to create a look-up table for each fuzzy input. The values in the LUT are plotted in Figure 32 as a function of sample offset.

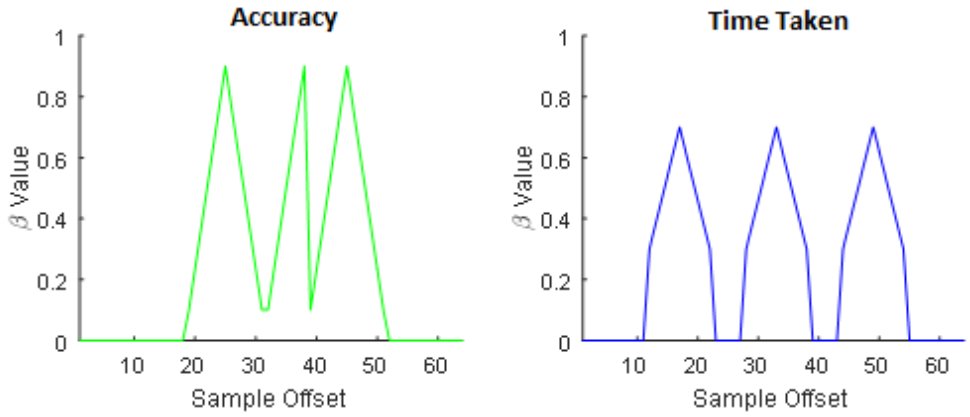

Figure 32. β Functions for Time and Accuracy Inputs

The final parameter required to generate the system is an initialization array for the Beta State Look-Up Table. This array is generated by analyzing the FSM transitions from Figure 31 and generating 12 total vectors, one for each state, which represent neighboring states – those states that have a transition to or from the current state. The contents of the initialization array are shown in Table 10.

Table 10. Beta State Look-Up Table Contents

Current State	Beta State Look-Up Table Value (12:1)
1	000000001111
2	000000001111
3	000000011111
4	000001111111
5	000001111100
6	000011111000
7	011111111000
8	000011100000
9	100101000000
10	100101000000
11	100101000000
12	111100000000

In state 1, the FSM shares transitions with states 2, 3, or 4; therefore, bits 1, 2, 3, and 4 of the vector for current state 1 are set to '1' and all others to '0'. In state 4, the FSM shares transitions with states 1 through 3 and 5 through 7 meaning bits (7:1) of the corresponding vector are set to '1'. Recall that a neighboring state becomes a β state regardless of the direction of the state transition. This means that although the FSM cannot transition from state 4 to state 3, it can transition from state 3 to state 4 and, therefore, the bit representing state 3 must be set in the entry for current state 4.

Once all parameters and LUTs were defined a set of model building and inference data was generated. Because no direct relationship between the inputs and output of the eye-hand coordination fuzzy system were given and no example data was provided, fictional data was generated simply to demonstrate the system's operation.

Figure 33 shows the contents of rule memory through four model building operations for fuzzy input 1, Time, and state 4. The fuzzified data used for these operations was generated using a narrow fuzzification algorithm resulting in sharp and defined peaks. From left to right and top to bottom the plots represent a snapshot of the contents of rule memory after each of four

model building operations. Axes labeled “n” represent the sample offset of the fuzzy input, X_{IN}, while “m” represents the sample offset of the fuzzy output, Z_{IN}.

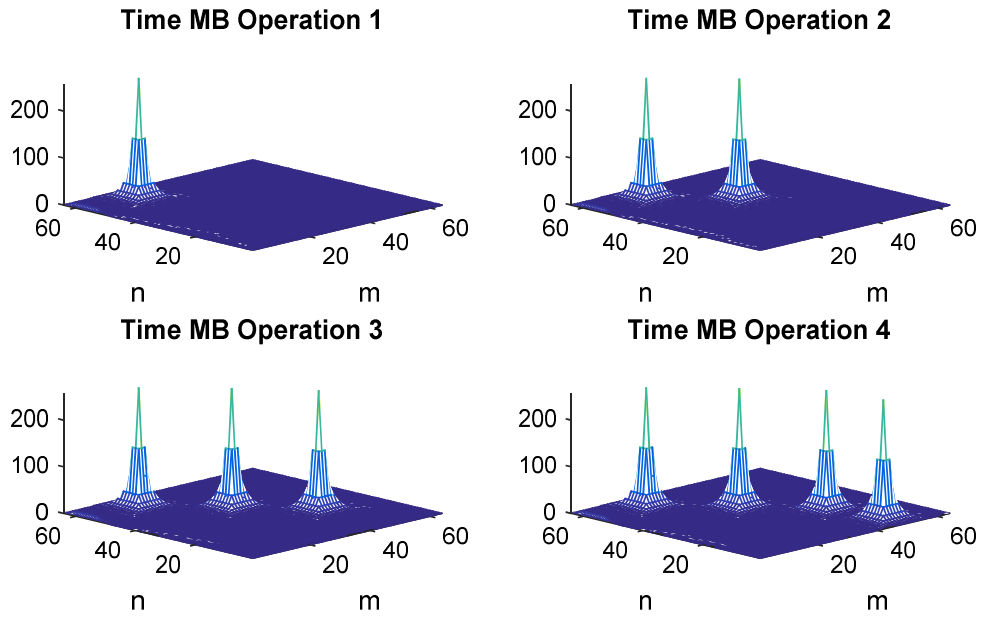

Figure 33. Rule Memory Through Model Building Operations

For these simulations, the fuzzified data X_{IN} and Z_{IN} are given a single peak value and a sharp slope to emphasize the operations resulting in the noticeable peaks where both X_{IN} and Z_{IN} reach a peak value of 255 which correlates to a membership value of 1.0.

Using the rule-base generated from the model building operations in Figure 33, an inference operation is simulated and shown in Figure 34 for a non-dominant state. Recall that an inference operation starts by taking the 0^{th} element of X_{IN}, “n”, and performing min/max composition with rule memory to determine the fuzzy output, Z_{OUT}, with “m” elements. The plot actually shows this process by plotting the inferred output after each new element of X_{IN} is completed starting with the 0^{th} element and ending with the 63^{rd}.

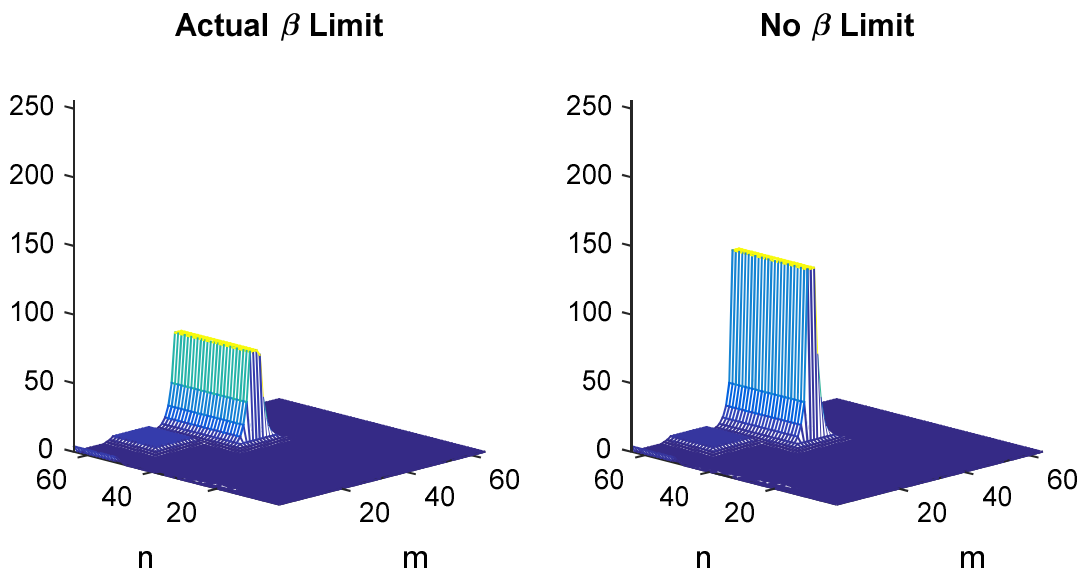

Figure 34. Z_{OUT} Building Through Inference

The plot on the left of Figure 34 shows the result of applying the calculated β value to this β state while the plot on the right shows the inferred output without applying β.

In order to thoroughly exercise the FSM and fully demonstrate the example, a set of model building and inference data was generated to step the system through several states of the FSM to demonstrate the remaining features of the system. Figure 35 shows the system stepping through 10 simulation trials. At trial 1, the system is in state 1, the initial state, with an initial β value of 0. Trial 1 is the initial state of the system and the first trial where inference is completed is trial 2 where the system moves to state 2 and calculates a non-zero β value, as indicated by the shorter bars associated with states 1, 3 and 4. The dominant state for each trial always uses an effective β value of 1.0 while all β states will use a value less than 1.0. In trial 3 the dominant state is determined to be state 3 and the β states are states 4 and 5 with a β value of approximately 0.3. This continues through trial 11 where the system reaches state 12 and stops.

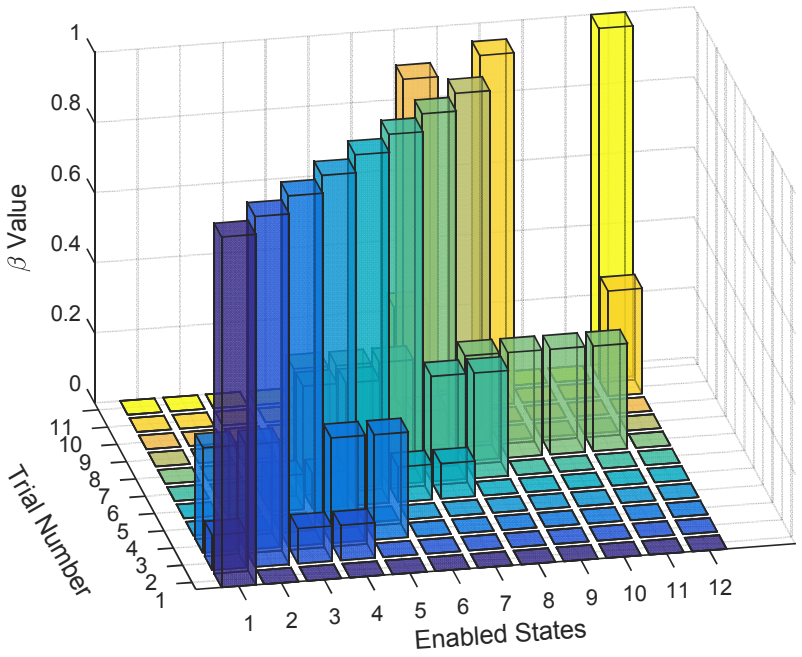

Figure 35. Dominant and β States at Each Simulation Trial

Figure 36 shows the dominant states chosen for each trial of this simulation as well as the β values calculated for each fuzzy input and the final value used in inference. The upper plot shows the crisp state of the HFB-FSM starting in state 1 then transitioning through states over the course of 10 simulation trials. The lower graph shows calculated β values through these same simulation trials. A β value is independently calculated for each fuzzy input giving β_{TIME} and β_{ACC} for inputs Time and Accuracy, respectively. The minimum of these values, shown simply as β, is then found and applied during fuzzy composition. β values are calculated offline based on parameters provided by the system designer and then found from a LUT based on the defuzzified MoM.

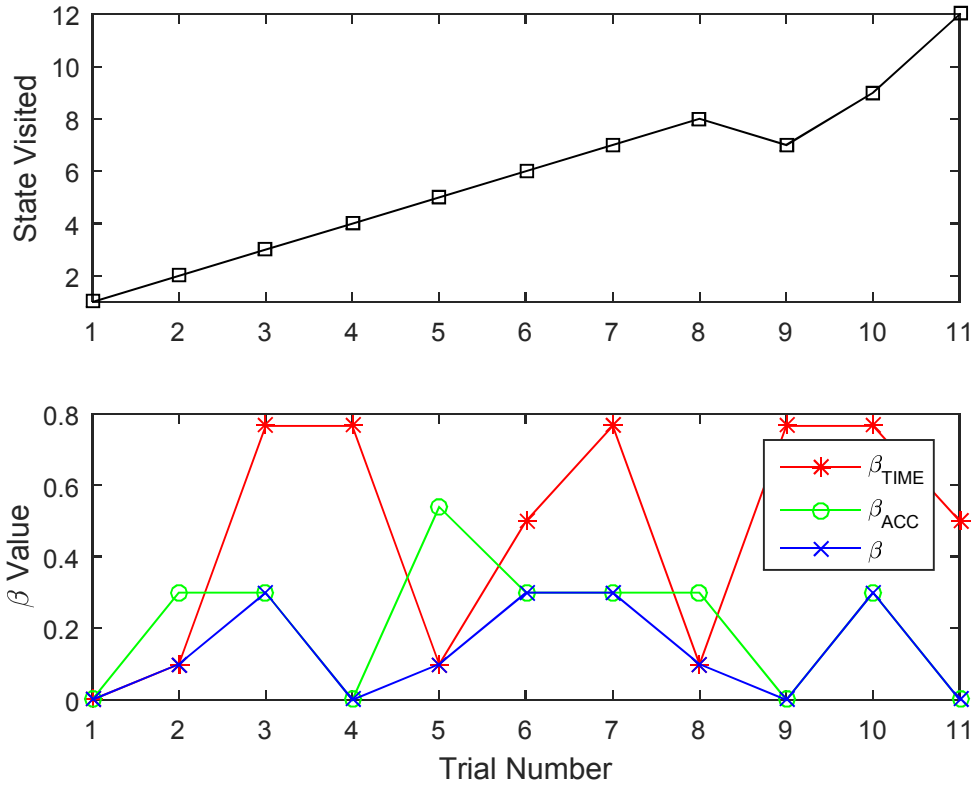

Figure 36. State Transitions and β Values at Each Simulation Trial

6. Conclusions

An implementation of the extended HFB-FSM system was simulated, designed, tested, and demonstrated based on an existing fuzzy logic application. The system was designed in VHDL and simulated using Xilinx ISIM. The system was also implemented in a Xilinx Zynq-7000 EPP on a Zedboard development board with a software co-design using an Ethernet interface to interact with the HFB-FSM. A Matlab model of the system was created to correlate the outputs of simulated and empirical data. The system is parameterized to allow for fast adaptation to new applications.

Future work may look at extending the capabilities of the system to MIMO systems. This can be done for a two-output system by instantiating the second processor connected to a second instance of the PL design.

Future work may also look to replace the min/max composition operations in Rule Memory and the Composition module with more advanced operators, such as those presented by Dombi [12].

Additionally, it may be beneficial to implement partial reconfiguration in the system to extend its capabilities [13]. However, because the application processor implements the host interface and due to the complexity of implementing a partial reconfiguration system, it may be advantageous to simply implement multiple full configurations to extend the capabilities.

Appendix A

Resource Usage and Performance Statistics

Table 11 summarizes resource utilization of the eye-hand coordination example as implemented on the Xilinx ZedBoard.

Table 11. Post-Implementation Utilzation Statistics

Resource	Utilization	Available	Utilization Percentage
FF	18768	106400	18 %
LUT	16472	53200	31 %
Memory LUT	666	17400	4 %
I/O	9	200	5 %
BRAM	33	140	24 %
DSP48	2	220	1 %
BUFG	2	32	6 %

Table 12 summarizes the timing report for the example system when constrained to a PL clock rate of 100MHz and an AXI bus clock rate of 50MHz.

Table 12. Post-Implementation Timing Summary

	Setup	Hold	Pulse Width
Worst Slack	0.396 ns	0.022 ns	3.75 ns
Total Slack	0 ns	0 ns	0 ns
Number of Failing Endpoints	0	0	0
Total Number of Endpoints	44138	44138	19521

Appendix B

List of Design Files

The following source files are listed as used in the design hierarchy for the PL portion of the system. For those files that are part of an instantiated IP block, only the name and version of the block is shown.

hfb_fsm_pkg.vhd
lut_data_pkg.vhd
beta_state_lut_data_pkg.vhd
system_interface.vhd
    top_module.vhd
        fuzzy_inputs.vhd
            fuzzy_input_module.vhd
                mean_of_maxima.vhd
                    div_gen_0 (Divider Generator v5.1)
                interval_detect.vhd
                    comparator.vhd
                beta_calculator.vhd
                addr_mapper.vhd
            lookup_table.vhd
            state_memory.vhd
            beta_comparator.vhd
            beta_state_lookup_table.vhd
            rule_memory.vhd
                blk_mem_gen_0 (Block Memory Generator v8.1)
            composition_module.vhd
    zynq_1_wrapper.vhd
        zyng_1.bd (Block Design)

The following files are used only for the purposes of simulating the PL system, excluding the block design.

system_interface_sim_module_tb.vhd
    system_interface_data_full_sim.txt

The following source files are used in the software co-design. This list excludes those files that are included in the board support package.

echo.c
hfb_fsm.c
hfb_fsm.h
hfb_fsm_bram.c
hfb_fsm_bram.h
hfb_fsm_gpio.c
hfb_fsm_gpio.h
i2c_access.c
main.c
platform.c
platform.h
platform_config.h
platform_zynq.c
sfp.c
si5324.c

REFERENCES

[1] Zadeh, Lotfi, “Fuzzy sets.” *Information and Control 8*, 338–353, 1965.

[2] Kosko, Bart, and Satoru Isaka. "Fuzzy logic." *Scientific American* 269.1 (1993): 62-7.

[3] Elamvazuthi, I., P. Vasant, and J. Webb. "The Application of Mamdani Fuzzy Model for Auto Zoom Function of a Digital Camera.” 2010

[4] Aly, Ayman A., et al. "An antilock-braking systems (ABS) control: A technical review." *Intelligent Control and Automation* 2.03 (2011): 186.

[5] Quiroga-Aranibay, Luis Alfonso. “Learning fuzzy logic from examples.” Diss. Ohio University, 1994.

[6] Grantner, J.L., Fodor, G., Driankov, D., “Hybrid Fuzzy-Boolean Automata for Ontological Controllers”**,** Fuzzy Systems Proceedings, 1998*. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on*, Volume: 1 , 4-9 May 1998 Pages:400 - 404 vol.1

[7] Grantner, J.L., Fodor, G.A., “Fuzzy Automaton for Intelligent Hybrid Control Systems”, FUZZ-IEEE'02. *Proceedings of the 2002 IEEE International Conference on* ,Volume: 2 , 12-17 May 2002 Pages:1027 – 1032

[8] Grantner, Janos L., et al. "Design of a reconfigurable state transition algorithm for fuzzy automata." *International Journal of General Systems* 37.1 (2008): 103-126.

[9] http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf

[10] http://zedboard.org/sites/default/files/ZedBoard_HW_UG_v1_1.pdf

[11] Grantner, Janos L., et al. "Intelligent Decision Support System for Eye-Hand Coordination Assessment." *Fuzzy Systems, 2005. FUZZ'05. The 14th IEEE International Conference on*. IEEE, 2005.

[12] J. Dombi, “A general class of fuzzy operators, the De Morgan class of fuzzy operators and fuzziness measures induced by fuzzy operators,” Fuzzy Sets and Systems, vol. 8, pp. 149–163, 1982

[13] D. Kim, "An Implementation of Fuzzy Logic Controller on the Reconfigurable FPGA System," *IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS,* p. 1, June 2000.

[14] Grantner, J. "Design of event-driven real-time linguistic models based on fuzzy logic finite state machines for high-speed intelligent fuzzy logic controllers." *Candidate of Science Dissertation (Hungarian Academy of Sciences)* (1994).