Making use of CORDICs and Distributed Arithmetic to produce a Field-Programmable Fuzzy Logic Controller in an FPGA

Trevor Lund, Miguel Aguirre and Antonio Torralba

Electronic Engineering Department, Seville University Engineering School, Seville, Spain.
[Trevor Lund was on study leave from the School of Computing, University of Canberra, ACT 2601, Australia.]
tlund@ise.canberra.edu.au

Abstract – Techniques from digital signal processing (DSP) have been used to perform some of the problematic operations needed in a fuzzy logic controller. These techniques have enabled the development of circuits which are not only compact but also scalable, so that we do not suffer as high a rate of increase of the use of resources as for conventional circuits when the accuracy of the controller or the number of inputs or membership functions is increased. Circuits have been developed so that a complete, reprogrammable fuzzy logic controller can be fitted into a medium-sized FPGA.

I. INTRODUCTION

We have been working on the design of a chip for use as a field-programmable fuzzy logic controller [1-3]. This chip uses analogue circuits for calculating membership values, evaluating rules, and defuzzifying the result, but uses FPGA-type interconnection techniques to allow it to be programmed in the field to suit a particular application. Analogue circuits are used in that design because they were seen to be more efficient in the use of chip area than digital circuits for the multiplication, division and hyperbolic function operations needed at various stages of the fuzzy control process. Other groups have worked on the design of circuits for fuzzy control without producing a complete programmable chip design, for example [4-6]. A completely digital design has been implemented and tested [7], and a discussion of how this compares with what we are trying to achieve is given in [3].

Recently we have realised that some of the techniques used in digital signal processing can be applied to the problem areas of fuzzy logic circuits, to enable the design of compact circuits which will perform the mathematical operations needed in fuzzification and defuzzification. An additional advantage of these techniques is that increasing the accuracy comes with an approximately linear increase in demand for the use of area rather than the approximately quadratic increase in demand produced by, for example, a conventional multiplier, so that they remain compact. As well as this, multiple copies of the same circuit can share certain components whilst retaining their individual function, so that a doubling of the scope of a subsystem will not lead to the doubling of the need for resources. The use of such circuits, with their limited demand on hardware resources, should enable us to fit a complete fuzzy logic controller inside a medium-sized FPGA.

The first of the techniques which has attracted our attention is the CORDIC [8-10]. The CORDIC (COordinate Rotation Digital Computer) is an iterative arithmetic computing algorithm capable of evaluating various functions using a shift-and-add technique. In this algorithm, all of the evaluation tasks are formulated as the rotation of a vector in various coordinate systems. By varying a few parameters, the same CORDIC circuit is capable of evaluating any of those functions that can be evaluated by the vector rotation technique. Being able to use the same hardware, and the same amount of time, to evaluate any of a variety of functions has made CORDIC based architecture appealing for implementation in pipelined VLSI array processors, and those same properties make the CORDIC attractive in this application.

The second of the techniques which has attracted our attention is "distributed arithmetic"[11], for vector-vector multiplications. In this technique, each word in the vectors is represented as a binary number, and the multiplications are re-ordered such that the arithmetic becomes distributed through the structure. When one of the vectors is known in advance, many of the components of this arithmetic can be computed in advance, so that the vector multiplication is reduced to a small number of shift-and-add operations.

Our intention is to produce an easy-to-use tool, which will accept the designer's specification for the fuzzy system - number of inputs, membership functions for each input, rule base, etc. – and generate the VHDL code ready for input to one of the commercially available FPGA programming systems. This scheme should facilitate the production of low cost reprogrammable fuzzy logic controller chips, since the FPGAs are readily available and inexpensive, as are the programming tools.
In this paper we outline the circuits we have developed to perform the various operations needed in a fuzzy logic controller, and how we are fitting them into an FPGA to produce a complete, multiple-input single-output (MISO) controller.

II. TOP LEVEL PLAN

Fig. 1 shows the basic configuration of a fuzzy logic controller. The idea is to use the CORDIC algorithm [8-10] to generate the hyperbolic functions needed in the fuzzification block to produce the membership function values. In the defuzzification block, "distributed arithmetic" [11] can be used to calculate the inner product of two vectors to calculate the weighted sum of the consequents, and a CORDIC circuit configured as a divider can be used to normalise that sum. For the rule evaluation block we simply need multiple-input comparators to perform the MIN/MAX operations.

Since the CORDIC algorithms and the distributed arithmetic algorithm each need several clock cycles to produce their result, we will make use of pipelining so that fresh samples can be fuzzified whilst earlier samples are further along the chain. This will allow the circuit to produce updates to the controlled variable at a few MHz, although the latency may be as much as five times the update period.

III. DEVELOPMENT OF THE CIRCUITS

A. Defuzzifier

We have chosen to use singletons for the membership functions for the output variable because we believe that this leads to no loss in generality nor performance, yet leads to easier understanding by the designer and hence easier tuning as well as having reduced computational complexity compared with the alternatives. In particular, the output is calculated from:

\[
\text{Controlled Variable} = \frac{\Sigma w_i C_i}{\Sigma w_i} \quad (1)
\]

Here the \( w_i \) are the outputs from the rule evaluation block specifying the degree to which each consequent is true, and the \( C_i \) are the numerical values of those consequents. In practical cases, the rule base may be incomplete, or not every rule will include mention of every input variable, so we cannot assume that \( \Sigma w_i \) will be unity, and the division operation cannot be omitted.

The numerator of (1) can be considered an inner product between two vectors, and the distributed arithmetic algorithm can be used to implement it. The division operation can be done by an appropriately configured CORDIC circuit.

Fig. 2 shows how the firing strengths of the rules (the \( w_i \)) are combined with the output fuzzy sets (the \( C_i \)) to which they refer in order to produce the value required for the controlled variable. The ROM contents are dependent only on the values \( C_i \). Hence, although the ROM is specific to an application, its contents can be calculated at the same time as the rest of the system is configured, and this is what enables the vector multiplication to be performed so efficiently in terms of both time and space. We have written a program in C to write the VHDL code for the ROM, given the values \( C_i \).

B. Fuzzifier

A popular form for membership functions is:

\[
\mu(x) = \exp[-(x-x_0)^2/2\sigma^2] \quad (2)
\]

Here \( x \) is the crisp value of the input variable in question, \( x_0 \) is the centre of the membership function and \( \sigma \) is a measure of its width. \( \mu \) is the degree of membership.

A CORDIC circuit can be configured to calculate hyperbolic functions efficiently, and so we were inspired to plan our fuzzifier around this form of membership function. However, calculating the argument for the exponential function proves to be more of a problem, perhaps, than evaluating the exponential in this situation.

The steps required in calculating (2) are shown in Fig. 3. It becomes apparent that it would be more sensible, on computational grounds, to choose triangular membership functions, and there seems to be no fundamental reason why triangular membership functions will work any less well than hyperbolic membership functions in practical applications, although hyperbolic functions may be preferred in learning or self-adapting systems.
We decided, for the moment, to use triangular membership functions, which means that only the first three stages of Fig. 3 will be needed for each membership function. However, comparators will be needed to ensure that a value of zero is generated when the input is out of the range of the set, rather than a negative value (or some other invalid value depending on the choice of representation of the numbers). Nevertheless, the extra expense of the comparators pays off, as we no longer need one membership function generator for each fuzzy input set, but only enough membership function generators to cover the number of sets which will have non-zero membership values at any one time.

Further, if we choose a structure for fuzzy sets for the input such that there is only overlap of two sets at a time, and the end of one set is directly below the peak of the next, as shown in Fig. 4, we need just two membership function generators for each input variable. In addition, the input sets are completely specified by specifying the peaks of the fuzzy sets (or the corners, for the sets at each end of the range), yet it is still possible to have asymmetric sets and sets of varied widths. Hence, for each input variable we need a fuzzifier of the structure shown in Fig. 5. This circuit is proceeded by a chain of comparators which compares the current value of the input variable, $x$, with the defined peak positions for the fuzzy sets, and sets one of the signals $a(i)$ true. $a(1)$ will be true if $x$ is above the lowest corner but below the first peak. $a(M)$ will be true if $x$ is above the highest corner, where $M$ is the number of fuzzy sets in use for that input.] The $a(i)$ signals are use to select from a ROM the values to be used for the lower peak, the upper peak and the width. Here, for the sake of efficiency at run-time, as well as the peak positions, the ROM contains the distances between adjacent peaks (the “width” values needed), calculated in advance from the specified peak positions.

C. Rule Evaluation Circuit
This circuit will simply be a collection of multiple-input digital comparators. Most of these will be configured to take the minimum of the truth values (the appropriate membership value) of the antecedents of each active rule. Given the design chosen for the fuzzifier circuit, we need just $2^r$ of these comparators, each with $n$ inputs, where $n$ is the number of input variables. Where more than one rule has the same consequent, another comparator will be needed, configured to take the maximum truth value (rule strength) for that consequent.

IV. IMPLEMENTATION OF THE CIRCUITS
The circuits needed to implement this system have been designed and tested using VHDL and Xilinx Foundation software, targeting initially the XC4010. Targeting a specific processor, such as this one, enables comparisons to be more easily made between our proposed methods and those better established.

A. Cordic
A general purpose CORDIC has been made, following the schematic given in Figure 1 of [8]. This CORDIC can be configured to operate in any of the three modes possible, viz. circular, linear, or hyperbolic, by setting the value of $r$ and providing the appropriate ROM and a finite state machine to drive it. When configured as a divider, 16 bits wide, it consumes the resources shown in Table I, and can operate with a clock rate up to 20 MHz. If a division should require the maximum number of steps (16 for a 16-bit wide operation), this implies that up to 800ns are needed.

<table>
<thead>
<tr>
<th>Number of CLBs</th>
<th>101 out of 400</th>
<th>25%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total CLB Flops</td>
<td>72 out of 800</td>
<td>9%</td>
</tr>
<tr>
<td>4 input LUTs</td>
<td>187 out of 800</td>
<td>23%</td>
</tr>
<tr>
<td>3 input LUTs</td>
<td>24 out of 400</td>
<td>6%</td>
</tr>
</tbody>
</table>

B. Distributed Arithmetic Circuit
Figure 13.36 of [7] was used as the basis for the distributed arithmetic circuit. It was constructed with a data path 8 bits wide, and with the assumption that up to 8 singletons values would be used to describe the output variable of the fuzzy controller. We chose a data structure such that all of the values involved (the rule strengths and the singleton positions) would be represented as positive fractions, and this led to a simplification of the overall design, not having to deal with negative numbers.
A C program was written to calculate the ROM contents and write the VHDL code to implement the ROM, given the values of the singletons describing the output variable. For $N$ singletons, we need a ROM of $2^N$ entries. For the choices specified in the paragraph above, this implies a ROM of 256 bytes.

The complete “inner vector product” circuit, including the special multiplexer circuit to generate the ROM addresses and a finite state machine to drive it, consumes the resources shown in Table II, and can operate with a clock speed up to 20 MHz. For 8-bit data width – implying that 8 cycles are needed for this product operation – that means that 400 ns are needed.

### Table II

<table>
<thead>
<tr>
<th>RESOURCES NEEDED FOR DISTRIBUTED ARITHMETIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>(configured as 8x8, 8 bits wide, and including the ROM)</td>
</tr>
<tr>
<td>Number of CLBs</td>
</tr>
<tr>
<td>Total CLB Flops</td>
</tr>
<tr>
<td>4 input LUTs</td>
</tr>
<tr>
<td>3 input LUTs</td>
</tr>
</tbody>
</table>

**C. Defuzzifier**

The defuzzifier, as indicated in section III, is simply the distributed arithmetic circuit followed by a Cordic configured as a divider (subsections B and A above), and consumes just the sum of the resources. Although only one of these defuzzifiers is needed in a complete fuzzy logic controller, it can be seen that by itself it consumes almost all of the resources in an XC4010, so a larger device will be needed for a complete design.

**D. Fuzzifier**

The structure shown in Fig. 5 has been implemented and tested, together with the chain of comparators and ROM needed to provide its inputs, for the case where the number of membership functions ($M$) is 5. Table III shows the resources consumed by this circuit.

### Table III

<table>
<thead>
<tr>
<th>RESOURCES NEEDED FOR FUZZIFIER</th>
</tr>
</thead>
<tbody>
<tr>
<td>(configured for 5 membership functions)</td>
</tr>
<tr>
<td>Number of CLBs</td>
</tr>
<tr>
<td>Total CLB Flops</td>
</tr>
<tr>
<td>4 input LUTs</td>
</tr>
<tr>
<td>3 input LUTs</td>
</tr>
<tr>
<td>Number of TBUs</td>
</tr>
</tbody>
</table>

Given that we need one fuzzifier for each input variable, it can be seen that, at first sight, the fuzzifiers for just three input variables would consume all of the resources of the device if an XC4010 is to be used. Whilst certain savings are possible – for example, all of the Cordics could share a single ROM and be controlled by the same counter – it is clear that the XC4010 will be inadequate for the implementation of the complete controller.

**E. Rule Evaluation Circuit**

At the time of writing, this circuit had not been implemented.

**V. CONCLUSION**

We have designed and implemented most of the circuits needed to construct a field programmable fuzzy logic controller, using some techniques not previously applied to this field. We have identified the resources needed for the implementation of the controller, and estimated the speed of operation of such a device.

**VI. REFERENCES**


