論文97-34C-7-5

고정밀 저비용 퍼지 제어기(I) - VHDL 설계 및 시뮬레이션

(An Accurate and Cost-Effective Fuzzy Logic Controller(I) - A VHDL Design and Simulation)

金 大 鎭 *, 趙 仁 顯 *

(Daijin Kim and Inhyun Cho)

요 약

본 논문은 저비용이면서 정확한 제어를 수행하는 새로운 퍼지 제어기의 VHDL 설계 및 시뮬레이션을 다룬다. 제안한 퍼지 제어기 (Fuzzy Logic Controller: FLC)의 정확성은 비퍼지화 연산시 소속값뿐 아니라 소속 함수의 폭을 고려함으로써 얻어진다. 제안한 퍼지 제어기 저비용성은 기존의 FLC를 다음과 같이 개조함으로써 이루어진다. 먼저, MAX-MIN 추론이 레지스터 파일의 형태로 쉽게 구현 가능한 read-modify-write 연산에 의해 대치된다. 두 번째, COG 비퍼지화기에서 요구하는 제산 연산을 모멘트 균형점의 탐색에 의해 피할 수 있다. 제안한 COG 비퍼지화기는 곱셈기가 부가적으로 요구되며 모멘트 균형점의 탐색 시간이 오래 걸리는 단점이 있다. 부가적 곱셈기 요구에 의한 하드웨어 복잡도 증가 문제는 곱셈기를 확률론적 AND 연산에 의해 해결할 수 있고, 오랜 탐색 시간 문제는 coarse-to-fine 탐색 알고리즘에 의해 크게 경감될 수 있다. 제안한 퍼지 제어기의 각 모듈은 VHDL에 의해 구조적 수준 및 행위적 수준에서 기술되고, 이들이 제대로 동작하는지 여부를 SYNOPSYS사의 VHDL 시뮬레이터 상에서 트럭 후진 주차 문제에 적용하여 검증하였다.

Abstract

This paper concerns a VHDL design and simulation of an accurate and cost-effective fuzzy logic controller (FLC). The accuracy of the proposed FLC is obtained by using the center of gravity (COG) defuzzifier that considers both membership values and spans of membership functions in calculating a crisp value. The cost-effectiveness of the proposed FLC is obtained by restructuring the conventional FLC in the following ways: Firstly, the MAX-MIN inference is replaced by a read-modify-write operation that can be implemented economically in the structure of register files. Secondly, the division in the COG defuzzifier is avoided by finding the moment equilibrium point. The proposed COG defuzzifier has two disadvantages that it requires additional multipliers and it takes a lot of computation time to find the moment equilibrium point. The first disadvantage is overcome by replacing the multipliers with stochastic AND operations and the second disadvantage is alleviated by using a coarse-to-fine searching algorithm. The proposed FLC is described in VHDL structurally and behaviorally and whether it is working well or not is checked on SYNOPSYS VHDL simulator by using the truck backer-upper control problem.

* 正會員, 東亞大學校 컴퓨터工學科

(Dept. of Computer Eng., DongA University)

※ 본 논문은 정보통신연구 관리단의 대학 기초 연구 지원 (96039-BT-12)에 의해 수행되었음.

接受日字:1997年1月27日, 수정완료일:1997年7月3日

I. 서 론

퍼지 논리 제어기는 가전 및 산업 분야의 공정 제어에 폭넓게 응용되고 있다. 특히 시스템의 특성이 복잡하여 기존의 정량적인 방법으로는 해석할 수 없거나, 얻어지는 정보가 정성적이며, 부정확하고 불확실한 경

우에 있어서 기존의 제어기보다 우수한 제어결과를 나타낸다[1]. 그림 1은 퍼지 제어기의 일반적 구성도를 나타낸 것으로 크게 4가지 구성 요소 - 퍼지화부, 추론 엔진부, 퍼지 규칙 베이스부, 그리고 비퍼지화부로 나뉜다. 각 부분의 동작 설명은 다음과 같다.

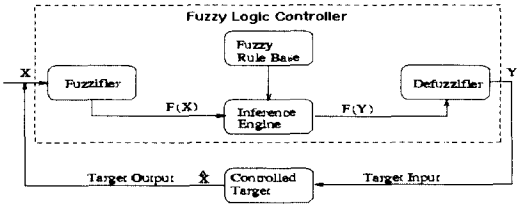

그림 1. 퍼지 제어기 일반적 구성도
Fig. 1. A typical organization of fuzzy logic controller.

퍼지화부에서는 입력 변수의 값을 측정하고, 입력 변수의 영역을 전체 집합범위에 맞게끔 크기 변환한 뒤 입력값을 적절한 언어적인 값으로 변환시키고, 추론부에서는 퍼지 관계와 퍼지 논리의 추론 규칙을 사용하여 퍼지 제어 출력을 결정하며, 퍼지 규칙 베이스부에서는 퍼지 논리 제어에서의 퍼지 자료를 조작하고 언어적 제어 규칙을 정의하는데 필요한 사항들을 정의한 데이타 베이스와 제어 전문가가 수행하는 일련의 제어 과정을 언어적 제어 규칙들로 나타낸 제어 규칙부로 구성되고, 그리고 비퍼지화부에서는 퍼지 출력값을 실제 제어 입력에 맞게끔 변환시켜 실제 제어 입력으로 사용할 수 있는 확정된 출력값으로 변환시켜 준다.

흔히 사용되는 비퍼지화 방법에는 최대값 방법(Max), 최대 평균법(Mean Of Maximum:MOM), 그리고 무게 중심법(Center Of Gravity: COG)이 있다[2]. COG 비퍼지화기를 하드웨어로 구성하는 경우 흔히 하드웨어의 복잡도를 줄이기 위해 출력 변수의 소속 함수 중심에서의 단일 출력값(Singleton)을 사용하여 비퍼지화를 행하는데, 이는 비퍼지화 출력값에 많은 오차를 유발하여 정확한 제어를 어렵게 만들 수 있다.

이를 극복하기 위해 출력 변수가 갖는 소속 함수 중심에서의 단일 출력값뿐 아니라 소속 함수의 면적도 동시에 고려하여 비퍼지화값을 연산할 것을 제안하였는데[3], 면적을 적분기에 의해 직접 계산하면 연산시간이 많이 걸리므로 적분기를 사용할 때에 비해 제어결과가 크게 뒤떨어지지 않으면서 비퍼지화 연산시 하드웨어 복잡도를 크게 증가시키지 않는 방안으로 소속함수의 면적을 소속 함수가 갖는 폭으로 근사화시키는 방안을 사용하였다.

기존의 COG 비퍼지화기가 갖는 또다른 단점은 비퍼지화값 연산시 구현 비용이 높고 연산시간이 많이 걸리는 나눗셈을 수행해야 한다. Ruitz 등[4]은 나눗셈을 출력 변수상의 좌·우측 모멘트의 균형점을 찾는 것으로 대신할 수 있음을 보였다. 그러나 이들이 제안한 모멘트 균형점 탐색은 매 탐색마다 출력 변수상에 매우 작은 크기의 동일 간격만큼 좌측 또는 우측으로 탐색점을 이동시킴으로서 탐색 시간이 많이 걸리는 단점이 있다. 이러한 단점을 극복하기 위해 coarse-to-fine 탐색 방법[3]을 제안하였는데 이 방법에서는 coarse 탐색시 출력 변수상 탐색점의 이동을 퍼지항 단위로 수행하며, 인접하는 두 퍼지항에 도달한 다음 Ruitz 등이 사용한 Fine 탐색을 수행함으로서 탐색시간을 크게 줄일 수 있다.

그러나, 제안된 모멘트 균형점의 coarse-to-fine 탐색 알고리즘은 곱셈기를 추가적으로 요구하는데 본 논문에서는 이 문제점을 확률론적 연산 이론[5]의 도움으로 해결하고자 한다. 즉, 확률론적 연산에 따르면 한 수치값이 확률적으로 비트열내의 "1"의 개수와 비례하도록 표현되며, 이 경우 약간의 연산 오차가 발생되지만 두 수의 곱이 두 비트열간의 AND 연산에 의해 가능하다. 따라서 곱셈기를 하나의 AND 게이트에 의해 구현함으로서 구현 비용을 줄일 수 있다.

본 논문의 구성은 다음과 같다. II장에서는 제안한 비퍼지화기의 하드웨어 복잡도를 줄이는 두 가지 방안을 제시한다. III장에서는 고정밀 저비용 특성을 갖는 새로운 FLC의 아키텍쳐어를 보이고 이의 동작 과정을 설명한다. VI장에서는 제안한 FLC를 재구성 가능한 FPGA 시스템 상에 구현하는 과정을 설명한다. V장에서는 FPGA상에 구현한 FLC와 VHDL 시뮬레이터 상에 구현한 FLC를 제어 수행 속도면에서 서로 비교한다. 마지막으로, VI장은 결론과 앞으로의 연구 방향을 언급한다.

II. 제안한 비퍼지화기의 하드웨어 복잡도 감소 방안

기존의 COG 비퍼지화기는 비퍼지화값을 다음 식에 의해 얻는다.

$$y_c = \frac{\sum_{i=1}^{n} y_i \cdot \mu_Y(y_i)}{\sum_{i=1}^{n} \mu_Y(y_i)} \quad (1)$$

여기서 n은 이산 퍼지항의 개수, y_i는 i번째 퍼지항의 단일 지지값(singleton), $\mu_Y(y_i)$는 단일 지지값 y_i에서의 소속 함수값을 나타낸다.

기존의 COG 비퍼지화기가 비퍼지화값 연산시 소속함수의 폭이 다름에도 불구하고 같은 비퍼지화값을 나타내는 잘못을 바로잡기 위해, 비퍼지화값을 계산하는데 소속 함수의 단일 지지값과 절단된 소속 함수 아래의 면적을 고려하기 위해 아래 식에 의해 계산할 것을 제안한다.

$$y_c = \frac{\sum_{i=1}^{n} A_Y(y_i) \cdot y_i}{\sum_{i=1}^{n} A_Y(y_i)} \quad (2)$$

여기서 $A_Y(y_i)$는 i번째 추론된 소속함수의 절단된 소속 함수값의 아래 영역, n은 퍼지항의 갯수, y_i는 i번째 퍼지항의 단일 지지값을 나타낸다.

그러나, 절단된 소속 함수값의 아래 영역의 연산은 적분기를 요구하므로 이를 하드웨어로 구현하면 구현 비용이 증가하고 연산시간이 오래 걸리는 단점이 있다. 따라서 이 문제는 정확한 연산을 위해서는 면적 적분을 위한 적분기가 요구되는 점과 이를 하드웨어적으로 구현하면 너무 하드웨어가 복잡해지는 성능 대비 비용이라는 공학적 trade-off 문제임을 알 수 있다. 이러한 타협 문제에 대한 한 해결책으로 적분기를 사용할 때에 비해 제어 결과가 크게 뒤떨어지지 않으면서 비퍼지화 연산시 하드웨어 복잡도를 크게 증가시키지 않는 방안으로 아래 식과 같이 소속 함수의 면적을 소속 함수가 갖는 폭으로 근사화시키는 방안을 고려한다.

$$y_c^a = \frac{\sum_{i=0}^{n} A_Y(y_i) \cdot y_i}{\sum_{i=0}^{n} A_Y(y_i)} = \frac{\sum_{i=0}^{n} \mu_Y(y_i) \cdot s_i \cdot y_i}{\sum_{i=0}^{n} \mu_Y(y_i) \cdot s_i} \quad (3)$$

여기서, $A_Y(y_i)$는 i번째 추론된 소속함수의 절단된 소속 함수값의 아래 영역, n은 퍼지항의 개수, s_i는 i번째 소속 함수가 갖는 폭, y_i는 i번째 퍼지항의 단일 지지값을 나타낸다. 기존의 COG 비퍼지화기와 비교해 보면, 비퍼지화값을 계산하기 위해서는 부가적인 곱셈기를 요구한다. 그러나, 뒷장의 시뮬레이션 결과에서 볼 수 있듯이, 이러한 추가적인 하드웨어의 요구는 제어 성능의 향상에 의해서 보상되어진다.

1. 승산 연산을 확률론적 AND 연산으로 대치

식 (2)에서 알 수 있듯이 제안한 COG 비퍼지화기는 비퍼지화값을 계산하기 위하여 소속값과 소속함수 폭의 값을 동시에 곱셈 처리해야 하므로 추가적인 곱셈기를 요구한다. 본 논문에서 추가적 곱셈기 요구 문제를 확률론적 (stochastic) 연산 이론에 기초한 간단한 AND 연산으로 대치함으로써 추가적인 곱셈기 요구에 따른 하드웨어 복잡도 증가 문제를 해결하고자 한다.

확률론적 연산은 하나의 수치값에 비례하는 "1"의 개수를 갖는 상대적으로 긴 확률론적인 랜덤 펄스열을 사용한다. 한 수치값을 확률론적인 펄스열로 부호화하는 코딩 회로는 아래 그림 2-(a)처럼 하나의 의사 난수 발생기(PRNG; Pseudo-random number generator)와 하나의 디지털 비교기를 사용한다. 본 논문에서는 의사 난수 발생기로서 통상적으로 사용되는 선형 궤환 쉬프트 레지스터(Linear feedback shift register) 대신에 셀룰러 오토마타(CA) 이론에 기반한 PRNG[6]를 사용하였는데, 이는 후자를 택하면 제일 처음과 마지막 셀 사이의 긴 궤환 루프가 필요하지 않기 때문이다. 최대 순환 길이가 2^n-1을 나타내는 CA 기반 PRNG는 아래 식과 같은 규칙 90과 규칙 150에 따라 그들의 상태를 바꾸는 n개의 셀로 구성된다[7].

$$\begin{aligned} Rule90: \quad & s_i(t+1) = s_{i-1}(t) \oplus s_{i+1}(t) \\ Rule150: \quad & s_i(t+1) = s_{i-1}(t) \oplus s_i(t) \oplus s_{i+1}(t). \end{aligned} \quad (4)$$

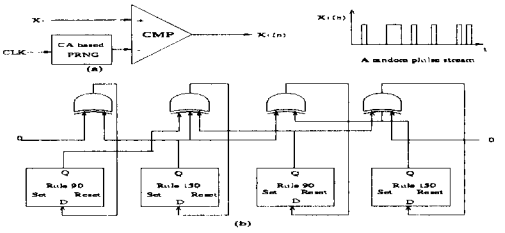

그림 2. 확률론적 연산을 위한 CA 기반 PRNG를 갖는 부호화 회로

Fig. 2. A coding circuit with CA-based PRNG for stochastic computing.

여기서, $s_i(t)$와 $s_i(t+1)$ 각각 i번째 셀의 현재 및 다음 상태를 나타내고, $\oplus$는 배타적 OR 연산을 의미한

다. 규칙 90은 이웃하는 셀들의 상태값에 의해 자신의 다음 상태를 결정하지만, 규칙 150은 다음 상태로 갱신하는데 자신의 현재 상태 및 이웃하는 셀들의 현재 상태를 동시에 고려한다. 그림 2-(b)는 4개의 셀을 갖는 CA 기반 PRNG의 한가지 가능한 형태를 보인 것이다.

비교기 출력에서 발생되는 각 펄스는 firing 확률 f_x를 갖는 Bernoulli 사건으로 모델링된다. Firing 확률 f_x는 $\hat{f}_x=\frac{n}{N}$에 의해서 근사화되는데 여기서 n은 N개의 클럭 주기동안에 발생되는 "1"의 개수를 나타낸다. Firing 확률을 $\hat{f}_x$에 근사화시키는 것은 평균이 f_x이고 분산 $\sigma^2_{f_x}=\frac{f_x(1-f_x)}{N}$인 이항 분포를 갖는다[7]. 따라서, 펄스열 $x_{(n)}$은 부호화 잡음 $\sigma^2_{f_x}$에 의한 오차를 피할 수 없다.

확률론적 연산의 한 장점은 두 개의 확률론적 펄스열의 곱셈이 한 AND 게이트에 의해 수행 가능하다는 점이다. 세 실수 x,y 및 z ($0\le x,y\le N$, $x\cdot y=z$)가 주어지면 x,y에 대응하는 두 확률론적 펄스열 f_x 및 f_y의 AND 연산은 아래 식에 의해 $\frac{z}{N}$을 나타낸다.

$$f_x \wedge f_y = \frac{n_x}{N}\cdot\frac{n_y}{N}=\frac{1}{N}\cdot\frac{n_x\cdot n_y}{N}=\frac{1}{N}\frac{n_z}{N}=\frac{f_z}{N}. \quad (5)$$

위식에서, 곱셈 결과가 $\frac{1}{N}$에 의해 크기가 감소됨을 알 수 있는데 이러한 scaling 감소는 오히려 FLC 내의 수치들의 범위를 동일하게 유지 시켜주는 역할을 한다.

2. 제산 연산을 모멘트 균형점 탐색으로 대치

COG 비퍼지화기는 다른 ALU 연산보다 훨씬 복잡한 나눗셈 연산을 요구한다. 본 논문에서는 나눗셈 연산을 모멘트 균형점 탐색으로 대신하고, coarse-to-fine 탐색 알고리즘에 의해 탐색 속도를 훨씬 빠르게 개선한다.

먼저, 출력 변수의 양끝에서부터 마주 보는 방향으로 출발하여 $y=y_c$까지의 좌측 및 우측 모멘트 M_l과 M_r는 각각 아래 식과 같이 정의된다.

$$\begin{aligned} M_l &= \sum_{i=0}^{c}\mu_Y(y_i)\cdot s_i\cdot y_i \\ M_r &= \sum_{i=c}^{M-1}\mu_Y(y_i)\cdot s_i\cdot y_i \end{aligned} \quad (6)$$

여기서 $\mu_Y(y_i)$, s_i, y_i와 M은 각각 i번째 소속 함수의 절단된 소속값, i번째 소속 함수의 폭, i번째 소속 함수의 단일 지지값과 출력 변수 y가 갖는 퍼지항의 갯수를 나타낸다. 이때, 찾고자 하는 비퍼지화값 $y=y_c$에서 좌측 우측 모멘트가 아래 식과 같이 같은 값을 갖는다.

$$\begin{aligned} M_l(c) &= M_r(c) \\ \sum_{i=0}^{c}\mu_Y(y_i)\cdot s_i\cdot y_i &= \sum_{i=c}^{M-1}\mu_Y(y_i)\cdot s_i\cdot y_i. \end{aligned} \quad (7)$$

위의 좌측 모멘트 M_l은 아래 식과 같은 반복식에 의해 쉽게 계산된다[3].

$$\begin{aligned} M_l(n) &= M_l(n-1)+A_l(n-1)\cdot(y_n-y_{n-1}) \\ A_l(n) &= A_l(n-1)+\mu_n\cdot s_n \end{aligned} \quad (8)$$

여기서 μ_n, s_n 과 y_n은 각각 n번째 반복에서 사용되는 소속함수의 절단된 소속값, 소속 함수의 폭, 그리고 소속함수의 단일 지지값을 나타내며, $A_l(n)$과 $M_l(n)$은 각각 n번째 반복에서 얻어지는 좌측 면적과 우측 모멘트를 나타낸다. 비슷하게 우측 면적 $A_r(n)$과 우측 모멘트 $M_r(n)$의 반복식은 아래와 같이 표현된다[3].

$$\begin{aligned} M_r(n) &= M_r(n-1)+A_r(n-1)\cdot(y_{M-n}-y_{M-n-1}) \\ A_r(n) &= A_r(n-1)+\mu_{M-n-1}\cdot s_{M-n-1} \end{aligned} \quad (9)$$

여기서 M은 출력 변수가 갖는 퍼지항의 갯수이다.

Ruitz등[4]은 M_l과 M_r을 계산하는데 출력 변수가 가지는 전체 정의역을 2^p개로 균등 분할하여 탐색점을 한번에 한 간격씩 옮기면서 탐색을 수행하는데 모멘트 균형점까지 최소 2^{p-1}번에서 최대 2^p번을 옮겨야한다. 이러한 과도한 탐색점 이동 문제를 해결하기 위해 본 논문에서는 coarse-to-fine 탐색 방법을 제안한다. (여기서 대문자로 나타낸 양들은 coarse 탐색과 관련된 것이고, 소문자로 나타낸 양들은 fine 탐색과 관련된 것이다.)

Coarse-to-fine 탐색 방법은 다음과 같이 수행된다. 탐색을 수행하기에 앞서 A_l과 A_r, 그리고 M_l과 M_r을 0으로, $I_l=0$, $I_r=M+1$ (단, M은 퍼지항의 갯수)으로 초기화시킨다. coarse 탐색은 앞의 식 (8)과 (9)를 이용하여 M_l과 M_r을 구한 뒤 이들을 서로 비교하여 작은 값을 갖는 쪽이 한 퍼지항 단위로 이동하는데 (즉, M_l이 작으면 $I_l=I_l+1$이고 M_r이 작으면 $I_r=I_r-1$) 이 과정을 $|I_r-I_l|\le 1$이 될 때까지 반복한다. $M_l=M_r$인 경우에는 양쪽 모두 이동(즉, $I_l=I_l+1$과 $I_r=I_r-1$) 한다.

Coarse 탐색의 종료 조건을 만족하는 순간의 I_l과

I_l을 각각 I_l^c, I_r^c라고 하면, 왼쪽 및 오른쪽 모멘트의 값은 각각 $M_l(I_l^c)$과 $M_r(I_r^c)$가 된다. 이들 값을 Fine 탐색을 위한 초기값으로 사용하기 위해 $i_l = I_l^c$, $i_r = I_r^c$ 그리고 $m_l(i_l) = M_l(I_l^c)$, $m_r(i_r) = M_r(I_r^c)$, $a_l = A_l(I_l^c)$, $a_r = A_r(I_r^c)$로 둔다. Fine 탐색 과정에서 왼쪽 및 오른쪽 모멘트의 반복식은 아래와 같다.

$$\begin{aligned} m_l(n+1) &= m_l(n) + a_l \qquad n = i_l, i_l+1, \cdots\cdots \\ m_r(n+1) &= m_r(n) + a_r \qquad n = i_r, i_r-1, \cdots\cdots \end{aligned} \tag{10}$$

실제 구현에서는 위 식의 덧셈을 AND 게이트의 한 입력을 항상 "1"로 둔 채 확률론적 곱셈에 의해서 수행하였다. 위 식을 이용하여 계산된 m_l과 m_r을 서로 비교하여 작은 값의 모멘트를 갖는 쪽을 한 단위만큼 (여기서 한 단위는 $\frac{\text{출력 변수 범위}}{2^p}$의 크기를 갖는다.) 이동하는데, (즉, m_l이 작으면 $i_l = i_l + 1$이고 m_r이 작으면 $i_r = i_r - 1$) 이 과정을 $|i_R - i_l| \le 1$이 될 때까지 반복한다. $m_l = m_r$인 경우에는 양쪽 모두 이동 (즉, $i_l = i_l + 1$ 과 $i_r = i_r - 1$) 한다. Fine 탐색이 끝난 뒤, 비퍼지화값 y_c는 다음 식 (10)에 의해 결정된다.

$$y_c = \begin{cases} i_l & \text{if } m_l \ge m_r \\ i_r & \text{if } m_l < m_r \end{cases} \tag{11}$$

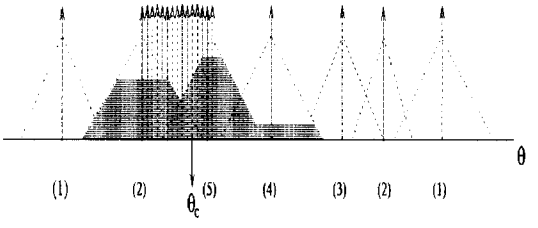

그림 3. 모멘트 균형점 탐색 과정 예시
Fig. 3. An illustration for finding a moment e-quilibrium point.

그림 3은 제안한 coarse-to-fine 탐색 알고리즘에서 모멘트 균형점을 찾는 과정을 나타낸 것이다. 앞에서 설명한 것처럼 coarse 탐색에서는 탐색점이 그림 3의 아래 부분에 나타낸 인덱스의 순서에 따라 퍼지항 단위로 움직인다. 일단 좌측 및 우측 모멘트가 서로 인접하는 두항까지 도달한 다음에는 미리 정의한 간격 ($\frac{|\theta_{\max} - \theta_{\min}|}{2^p}$)으로 움직이는 fine 탐색이 시작된다. Coarse 탐색동안 모멘트 탐색점의 이동은 $\frac{M}{2} \sim M$ (여기서 M은 퍼지항의 개수)번이고 fine 탐색동안 모멘트 탐색점의 이동은, 인접한 소속 함수의 중심간 거리가 평균적으로 $\frac{2^p}{M}$일 때, ($\frac{2^p}{2 \cdot M} \sim \frac{2^p}{M}$)번이다. 따라서 제안한 coarse-to-fine 탐색 알고리즘의 전체 모멘트 탐색점의 이동은 최소 $\frac{M}{2} + \frac{2^p}{2 \cdot M}$번, 최대 $M + \frac{2^p}{M}$번이다. 따라서 $2^p \gg M$ 이라면 Ruiz 알고리즘에 비해 최대 $O(M)$의 탐색 속도 향상을 얻을 수 있다.

Ⅲ. 제안한 FLC의 구조

FLC를 하드웨어적으로 구현하는 경우, 하드웨어의 복잡도를 줄이고 제어 성능을 높이기 위해 다음과 같은 제약을 가한다.

1. FLC의 입력은 비퍼지형(crisp) 값을 갖고 유한개의 값으로 양자화되어 있다.
2. 각 입출력 변수의 소속 함수 중첩도는 최대 2이다.
3. 각 출력 변수의 소속 함수 모양은 대칭형의 삼각형이다.
4. 정확한 비퍼지화값을 얻기 위해 각 소속 함수의 소속값과 폭을 동시에 고려한다.

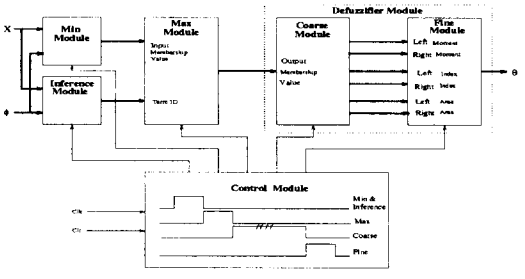

그림 4. FLC 하드웨어의 구조
Fig. 4. A architecture of FLC hardware.

첫 번째 제약은 MAX-MIN 추론을 간단한 lookup 테이블 연산에 의해서 가능하게 한다. 소속 함수값과 입력 소속 함수의 인덱스를 lookup 테이블에 미리 저장하며, 얻어진 입력 비퍼지형 값은 소속 함수값과 입력 소속 함수의 인덱스를 읽기 위한 주소로 사용된다. 두 번째 제약은 n차원 입력의 경우 최대 2^n개의 제어 규칙을 가질 수 있으며, 퍼지 규칙 베이스는 동시에 동작 가능한 2^n개의 배타적 부-규칙 베이스로 분할 가능하다. 세 번째 제약은 각 출력 소속 함수는 소속함수 중심에서의 단일 퍼지값으로 대신할 수 있게 됨

으로서 COG 비퍼지화값의 계산 복잡도를 크게 줄일 수 있다. 네 번째 제약은 COG 비퍼지화값 계산시 소속 함수값과 폭이 동시에 고려됨으로서 제어 성능이 개선된다. 아래 그림 4는 제안한 FLC의 하드웨어 구조를 나타낸 것으로, MIN 모듈, 추론 모듈, MAX 모듈, 비퍼지화 모듈, 및 제어 모듈로 구성되어 있다.

입력 변수 x_i가 $[l_{x_i}, u_{x_i}]$의 범위를 갖고, 이들이 2^p 등분으로 균일하게 양자화되어 있다고 가정한다. 비슷하게, 입력 변수 x_i의 각 소속 함수값이 2^q등분되어 각 양자화 간격이 $\frac{1}{2^q}$이다. 입력 변수 x_i가 갖는 N_{x_i}개의 퍼지항을 $\{T^1_{x_i}, T^2_{x_i}, \cdots, T^{N_{x_i}}_{x_i}\}$라고 하자. 여기서, 각 퍼지항 $T^j_{x_i}$ $(j=1,2,\cdots,N_{x_i})$는 소속 함수와 인덱스의 한쌍 $(M^j_{x_i}, I^j_{x_i})$에 의해서 나타내어진다. 입력 변수의 중첩도가 최대 2이므로, 입력 변수 x_i가 갖는 N_{x_i}개의 퍼지항은 아래 식과 같이 두 개의 배타적 퍼지 집합 $T^o_{x_i}$와 $T^e_{x_i}$로 나뉘어질 수 있다.

$$T^o_{x_i} = \{(M^j_{x_i}, I^j_{x_i}) \mid j=2\cdot k-1,\ k=1,\cdots,\frac{N_{x_i}}{2}\} \quad (12)$$
$$T^o_{x_i} = \{(M^j_{x_i}, I^j_{x_i}) \mid j=2\cdot k,\ k=1,\cdots,\frac{N_{x_i}}{2}\}$$

따라서, n개의 입력 변수를 갖는 FLC는 $2\cdot n$ 입력 lookup 테이블이 필요하며 각 lookup 테이블은 두 개의 MF 테이블과 ID 테이블로 구성된다. 각 MF 테이블에 소속 함수값을 저장하는데 요구되는 비트수는 $2^p\cdot q$이고, 각 ID 테이블에 인덱스 값을 저장하는데 요구되는 비트수는 퍼지항 집합내 각 퍼지항은 $\log_2(\frac{N_{x_i}}{2})$비트로 구분되므로 전체 $2^p\cdot\log_2(\frac{N_{x_i}}{2})$비트가 요구된다. 아래 그림 6은 $n=2$, $p=8$, $q=8$, $N_1=N_2=4$인 경우, 입력 lookup 테이블의 구성을 나타낸 것 이다.

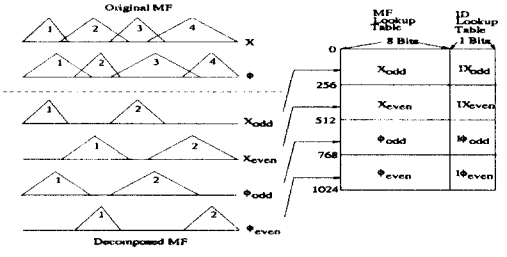

그림 5. 입력 lookup 테이블의 구성 예시
Fig. 5. An illustrative organization of input lo-okup table.

1. MIN 모듈

입력 변수 x_i의 한 특정 단일값이 두 개의 대응하는 lookup 테이블 $\{T^o_{x_i}, T^e_{x_i}\}$에 가해지면, 두 입력 lookup 테이블은 각각 소속 함수값과 인덱스로 구성되는 두 순서쌍 $(M^o_{x_i}, I^o_{x_i})$와 $(M^e_{x_i}, I^e_{x_i})$를 내보낸다. 따라서, n개의 입력 변수를 갖는 FLC는 2^n개의 전단부를 나타내는데 인덱스 $\{(I^k_{x_1} I^k_{x_2}, \cdots, I^k_{x_n}) \mid k=o \text{ or } e\}$를 갖는 전단부는 한 확신도 $\{MIN(M^k_{x_1} M^k_{x_2}, \cdots, M^k_{x_n}) \mid k=o \text{ or } e\}$를 갖는다. 만약 MIN 연산이 비교기 트리에 의해 구현된다면, 2^n개의 전단부를 동시에 수행한다면, 전체 $2^n\cdot(n-1)$개의 비교기가 요구된다.

본 논문에서는 이러한 하드웨어 복잡도를 줄이기 위해 *read-modify-write* 연산[8](여기서 *modify*는 MIN을 의미함.)에 의해 MIN 연산을 수행할 것을 제안한다. *Read-modify-write* 연산을 수행하는데는 단지 2^n개의 레지스터가 요구된다. 이는 하드웨어 복잡도가 $O(n)$만큼 감소됨을 의미한다. MIN 연산을 위한 *read-modify-write* 연산은 다음과 같이 수행된다. 초기에, 2^n개의 레지스터를 소속 함수가 가질 수 있는 최대값(FFH)으로 로드한다. MIN 연산을 마치는데 n싸이클 걸리는데, i번째 싸이클에서는 입력 변수 x_i가 *read-modify-write* 연산을 수행한다. 또한 모든 싸이클에서 동일한 과정이 수행되므로 여기서는 특정 i번째 싸이클에 대해서만 설명하기로 한다. 첫 번째 레지스터의 값을 읽어서 입력 변수 x_i가 갖는 소속 함수값과 비교한다. 두 값중 작은 값이 첫 번째 레지스터에 다시 쓰여진다. 이러한 연산이 2^n개의 레지스터에 걸쳐 반복 수행된다. 한가지 주목할 점은 우수번째 및 기수번째 MF lookup 테이블이 읽혀지는 주기는 입력 변수 x_i가 동작되는 i번째 싸이클에서는 2^{n-1}로 교대된다.

예를 들어 $n=2$일때를 고려해보면, 입력 변수 x_1는 첫 번째 싸이클 동안 x^o_1, x^o_1, x^e_1, 그리고 x^e_1의 순서로 참조되며, 입력 변수 x_2는 두 번째 싸이클 동안 x^o_2, x^e_2, x^o_2, 그리고 x^e_2의 순서로 참조된다. 따라서, MIN 연산을 완전히 끝마치는데 $n\cdot 2^n$번의 *read- modify-write* 연산이 필요하다. 그림 6은 입력 변수의 개수가 2일 때, 제안한 MIN 연산을 수행하는 회로의 블록도를 나타낸 것이다.

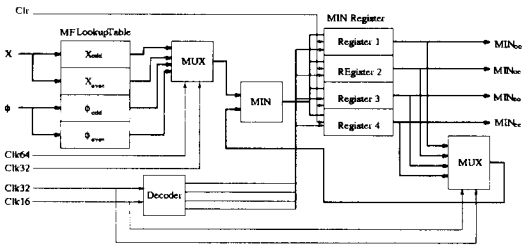

그림 6. Read-modify-write 연산을 수행하는 M-IN 모듈 회로 블록도

Fig. 6. An circuit diagram of MIN module executing read-modify-write operation.

2. 추론 모듈

n-입력 단일 출력을 갖는 FLC의 경우, 퍼지 규칙 베이스는 n차원 셀 어레이로 취급할 수 있는데[9], 각 셀 $C(x_1, x_2, \cdots, x_n)$은 입력 변수의 인덱스에 의해 정해진 한 제어 규칙 $R(x_1, x_2, \cdots, x_n)$의 후단부를 나타내는 출력 인덱스 $I_y(x_1, x_2, \cdots, x_n)$를 갖는다. 입력 변수 내 퍼지항끼리의 중첩되는 것은 각 퍼지항을 우수 및 기수 인덱스를 갖는 두 개의 배타적인 퍼지항 집합으로 분리할 수 있으므로, 퍼지 규칙 베이스는 2^n개의 부규칙 베이스 $\{R_{k_1,k_2,\cdots,k_n} | k_i = o \text{ or } e, i = 1,2,\cdots,n\}$로 분할될 수 있다. 각 부규칙 베이스 $R_{k_1,k_2,\cdots,k_n}$는 셀집합 $\{C(I^{x_1}_{k_1}, I^{x_2}_{k_2}, \cdots, I^{x_n}_{k_n}) | k_i = e \text{ or } o,\ i = 1,2,\cdots,n\}$으로 구성되며, 여기서 $I^{k_i}_{x_i}$는 k_i가 우수냐 기수냐에 따라 인덱스 lookup 테이블 $T^e_{x_i}$ 또는 $T^o_{x_i}$로부터 얻어진 인덱스를 나타낸다. 예를 들어 $n=2$이고 각 입력 변수가 각각 5개와 7개의 퍼지항으로 되어 있는 경우를 생각해보면, 퍼지 규칙 베이스는 최대 $5 \cdot 7 = 35$개의 제어 규칙을 갖는다. 이 퍼지 규칙 베이스는 다음과 같이 2^2개의 부규칙 베이스로 나뉘어 질 수 있다.

$$
\begin{aligned}
R_{o,o} &= \{C(i,j) |\ i = 2\cdot k-1, k = 1,\cdots,\frac{5}{2}+1,\\
&\qquad j = 2\cdot l-1, l = 1,\cdots,\frac{7}{2}+1\}\\
R_{o,e} &= \{C(i,j) |\ i = 2\cdot k-1, k = 1,\cdots,\frac{5}{2}+1,\\
&\qquad j = 2\cdot l, l = 1,\cdots,\frac{7}{2}\}\\
R_{e,o} &= \{C(i,j) |\ i = 2\cdot k, k = 1,\cdots,\frac{5}{2},\\
&\qquad j = 2\cdot l-1, l = 1,\cdots,\frac{7}{2}+1\}\\
R_{e,e} &= \{C(i,j) |\ i = 2\cdot k, k = 1,\cdots,\frac{5}{2},\\
&\qquad j = 2\cdot l, l = 1,\cdots,\frac{7}{2}\}
\end{aligned}
\qquad (13)
$$

그림 7은 입력 변수의 개수가 2인 경우 제안한 추론 모듈의 회로 블록도를 나타낸 것이다. 추론 모듈은 후단부를 나타내는 출력 인덱스를 저장하기 위해 $(\prod_{i=1}^{n} N_{x_i}) \cdot \log_2 N_y$ 비트가 요구되는데, 여기서 N_{x_i}와 N_y는 각각 입력 변수 x_i와 출력 변수 y의 퍼지항의 개수이다. 퍼지 규칙 베이스의 분할이 단순히 퍼지 규칙 베이스를 출력 인덱스를 저장하는데 각각 $(\prod_{i=1}^{n} \frac{N_{x_i}}{2}) \cdot \log_2 N_y$ 비트를 요구하는 2^n개의 배타적인 부규칙 베이스로 나누므로, 출력 인덱스를 저장하기 위해 요구되는 메모리 용량을 증가시키지는 않는다.

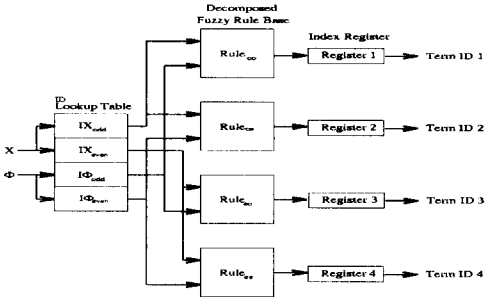

그림 7. 제안한 추론 모듈의 회로 블록도 ($n=2$)

Fig. 7. An circuit diagram of the proposed inference module ($n=2$).

3. MAX 모듈

2^n개의 각 부규칙 베이스는 관측된 하나의 n차원 입력에 관해 하나씩의 출력 인덱스를 나타내므로 최대 2^n개의 출력 인덱스를 나타낸다. 어떤 부규칙 베이스들은 같은 출력 인덱스를 나타낼 수 있으므로, MAX 연산은 이러한 충돌을 해결하는 방안이 있어야만 한다. 2^n개의 부규칙 베이스에서 추론된 후단부의 출력 인덱스와 소속 함수값을 $\{(I^1_y, M^1_y), (I^2_y, M^2_y), \cdots, (I^{2^n}_y, M^{2^n}_y)\}$라고 둔다. 이 경우, 두 개의 서로 다른 출력 인덱스 (I^i_y, I^j_y)를 비교하기 위해 $C(2^n, 2)$개의 등호 비교기가 필요하며, 두 개의 서로 다른 소속 함수값 (M^i_y, M^j_y)를 비교하기 위해 $C(2^n, 2)$개의 크기 비교기가 필요하다. 따라서, MAX 연산시 같은 출력을 나타내는 것을 해결하기 위해 요구되는 비교기 개수는 입력 변수의 수가 늘어날수록 급증하게 된다.

이러한 급증하는 하드웨어 복잡도를 줄이기 위해, 본 논문에서는 또다른 *read-modify-write* 연산(여기서, *modify* 연산은 MAX 연산을 의미함.)에 의해서 MAX 연산을 수행할 것을 제안한다. 하나의 *read-*

modify-write 연산을 수행하는데, 출력 변수가 갖는 퍼지항의 개수가 M인 경우, M개의 레지스터가 필요한데 이것은 하드웨어 요구가 $O(\frac{2^n}{M})$배만큼 줄었음을 의미한다. 그림 8는 입력 변수의 개수가 2인 경우 제안한 MAX 모듈의 회로 블록도를 나타낸 것이다. 여기서 좌측 상단에 있는 2^n to 1 멀티플렉서는 부규칙 베이스에서 얻은 2^n개의 최소 소속 함수값 $M_y^{1*}M_y^{2*}\cdots$, $M_y^{2^n*}$중 하나를 선택하며, 좌측 하단에 있는 2^n to 1 멀티플렉서는 부규칙 베이스에서 얻은 2^n개의 출력 인덱스 $I_y^{1*}I_y^{2*}\cdots, I_y^{2^n*}$중 하나를 선택한다.

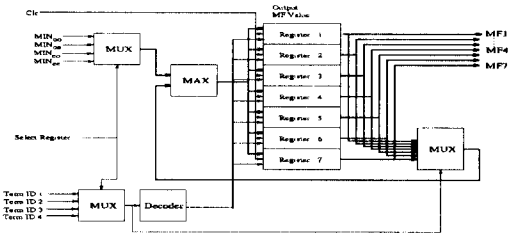

그림 8. 제안한 MAX모듈의 회로 블록도 ($n=2$)
Fig. 8. An circuit diagram of the proposed M-AX module ($n=2$).

MAX 연산을 위한 *read-modify-write* 연산은 다음과 같이 수행된다. 초기에, M개의 레지스터를 소속 함수가 가질 수 있는 최소값(00H)으로 로드한다. 첫 번째로, 출력 인덱스 I_y^1가 가리키는 레지스터의 값을 읽어 이를 첫 번째 레지스터가 갖고 있는 소속 함수값 M_y^1고 비교하여 이들값중 큰 값이 첫 번째 레지스터에 다시 쓰여진다. 이와 같은 *read-modify-write* 연산을 마지막 인덱스 $I_y^{2^n*}$에 대해 수행할 때까지 계속한다. 따라서, 하나의 MAX 연산을 끝마치는데 2^n싸이클이 걸린다. 이와 같은 순차적 처리는 보통의 MAX 연산시간보다 2^n배만큼 느리게 한다. 그러나, 이러한 수행 속도 저하는 FLC가 차지하는 수행 시간의 대부분이 다음에 설명하는 비퍼지화 단계에서 소모되므로 그리 큰 문제는 아니다.

4. 비퍼지화 모듈

비퍼지화기 모듈은 두 개의 기능 블록 (coarse 탐색 블록(B_c)과 fine 탐색 블록(B_f)) 으로 구성된다. 다시 coarse 탐색 블록(B_c)는 각각 좌·우측 모멘트를 계산하는 두개의 부블록 B_c^l과 B_c^r로 구성된다. 부블록 B_c^l는 다시 두 개의 기능 유닛 $B_c^{l,m}$ 과 $B_c^{l,a}$로 구성되고, 부블록 B_c^r는 다시 두 개의 기능 유닛 $B_c^{r,m}$ 과 $B_c^{r,a}$로 구성된다. 두 개의 부블록 B_c^l과 B_c^r는 같은 구조를 갖고 동작 과정이 비슷하므로 여기서는 좌측 부블록 B_c^l에 대해서만 설명한다. 첫 번째 기능 유닛 $B_c^{l,m}$는 좌측 모멘트를 연산하는 역할을 하는데 이는 $A_l(i)$와 $y_c(i)-y_c(i-1)$와의 확률론적 AND 연산 결과를 카운터에 누적 가산을 함으로서 이루어진다. 두 번째 기능 유닛 $B_c^{l,a}$는 좌측 면적을 연산하는 역할을 하는데 이는 $\mu_Y(y_i)$와 s_i와의 확률론적 AND 연산 결과를 카운터에 누적 가산을 함으로서 이루어진다. 여기서, AND 게이트의 두 입력은 CA 기반 PRNG와 디지털 비교기에 의해 발생된 256비트 비트열들이다. 두 기능성 유닛의 동작 속도 T_c는 $256\cdot CLK$이며 여기서 CLK는 확률론적 곱셈 연산시 사용되는 한 펄스당 시간을 의미한다.

T_c의 마지막 클럭 주기에서, 두 카운터 M_l과 A_l에 누산된 값들은 $Latch_c^m$과 $Latch_c^a$에 각각 옮겨진다. 두 기능 유닛사이에는 더 작은 모멘트 값을 갖는 부블록의 인덱스 포인터를 모멘트 균형점으로 이동하도록 하는 제어 회로가 존재한다. 두 부블록의 모멘트 값이 동일하면 양 인덱스 포인터가 모멘트 균형점으로 이동한다. 원하는 대로 동작하기 위해 인덱스 포인터가 이동하지 않는 부블록은 비활성화(deactivation) 시키는 것이 요구된다. 이를 위해 인덱스 포인터가 이동하지 않는 부블록은 입력으로 $M_Y(y_i)$와 s_i 대신 각각 "0"을 선택하도록 하였다. 이러한 제어 전략은 비활성화를 위해 추가적인 제어용 하드웨어가 필요하지 않으므로 매우 효과적이다.

제안한 coarse 탐색을 위해 각 소속 함수의 폭 s_i와 두 소속 함수간의 중심값의 간격 y_i-y_{i-1}, 그리고 소속 함수값 $M_Y(y_i)$를 저장하기 위해 각각 퍼지항 개수 M만큼의 크기를 갖는 세 개의 레지스터 파일이 필요하다. 각 레지스터 파일 다음에는 두 개의 멀티플렉서가 뒤따르는데 하나는 좌측 모멘트 부블럭을 위한 것이고 다른 하나는 우측 모멘트 부블럭을 위한 것이다. 두 개의 인덱스 포인터 I_l과 I_r은 레지스터 파일 중 하나의 레지스터 값을 선택하기 위해 사용하는데 I_l은 up 카운터로 I_r은 down 카운터로 작용한다. 그림 9는 출력 변수의 퍼지항이 7인 경우 제안한 COG

비퍼지화기의 모멘트 균형점 탐색의 coarse 탐색을 수행하는 회로의 블록도를 나타낸 것이다.

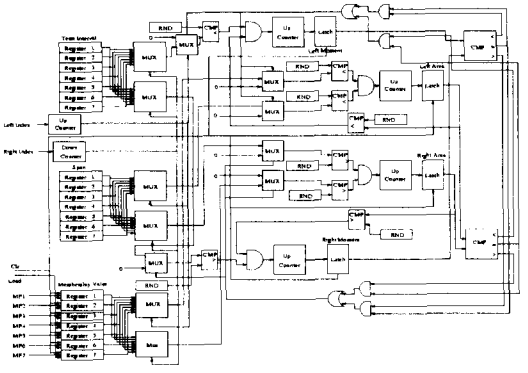

그림 9. Coarse 탐색을 수행하는 회로 블록도
Fig. 9. A circuit diagram for executing the coarse searching.

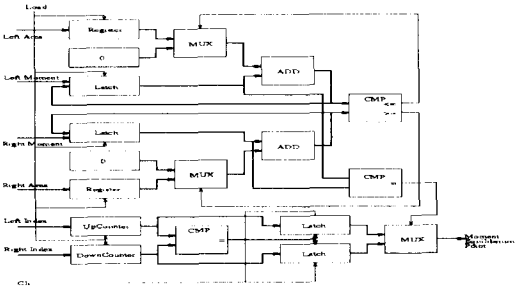

그림 10. Fine 탐색을 수행하는 회로 블록도
Fig. 10. A circuit diagram for executing the f-ine searching.

Fine 탐색 블록 B_f 역시 두 개의 부블록 B_f^l과 B_f^r로 구성된다. 두 부블록 B_f^l과 B_f^r는 각각 좌·우측 모멘트를 계산하는 두 개의 가산기 ADD_l과 ADD_r로 구성된다. 두 개의 부블록이 같은 구조를 갖고 같은 방식으로 동작하므로 본 논문에서는 좌측 부블럭 B_f^l에 대해서만 설명한다. Fine 탐색을 위한 두 개의 인덱스 포인터 i_l과 i_r는 초기값으로 각각 $y_c(I_c^l)$과 $y_c(I_c^r)$를 갖는다. 가산기 ADD_l의 두 입력은 a_l과 m_l로 각각 coarse 탐색 종료 후 면적과 모멘트 값을 나타낸다. Fine 탐색의 동작 주파수는 클럭 주파수 CLK와 같다. 매 탐색마다, 두 가산기 출력이 서로 비교되어 작은 모멘트 값을 갖는 부블럭이 모멘트 균형점으로 나아간다. 두 모멘트 값이 동일하면, 양 부블럭의 두 인덱스 포인터가 모멘트 균형점 쪽으로 나아간다.

Fine 탐색을 원하는 대로 동작시키기 위해서도 coarse 탐색과 같은 제어 전략을 사용한다. 그림 10은 제안한 COG 비퍼지화기의 모멘트 균형점 탐색의 fine 탐색을 수행하는 회로의 블록도를 나타낸 것이다.

5. 제어 모듈

전체 FLC는 각 모듈이 언제 시작하고 언제 끝나는지를 제어하는 여러 제어 신호들에 따라서 동작한다. 기본 클럭 주기(CLK)는 확률론적 곱셈을 하는데 쓰이는 한 펄스당 시간이다.

따라서, 한 변수값이 256개의 이진 펄스열에 표현되므로 확률론적 곱셈을 마치는데는 256 · CLK이 소요되는데 본 논문에서는 256 · CLK을 $CLK256$으로 정의한다. 그러면, 제안한 FLC에서 하나의 제어 연산을 수행하는데 걸리는 연산시간은 다음과 같다. MIN 연산과 추론 연산은 하나의 $CLK256$이 걸리며 이들은 동시에 수행 가능하다.

MAX 연산도 한 $CLK256$이 걸리며, 비퍼지화 모듈내의 coarse 탐색은 $N_c \cdot CLK256$이 걸리는 데, 여기서 N_c는 coarse 탐색에서 발생된 모멘트 탐색점의 이동 횟수이다. 끝으로 비퍼지화 모듈내의 fine 탐색은 한 $CLK256$이 걸린다.

따라서 하나의 퍼지 연산을 수행하는데 걸리는 연산시간은 최소 $(3+\frac{M}{2}) \cdot CLK256$에서 최대 $(3+M) \cdot CLK256$이 걸린다. FLC는 제어 대상물로부터 보내어 오는 $STARTUP$ 신호에 의해 제어 동작을 시작한다. 제어 동작을 시작하기 앞서, $RESET$ 신호가 모멘트 함수값을 메모리로부터 레지스터 파일에 로드하고 카운터, 가산기, 레지스터 파일들을 초기화한다. 다음 $START$ 신호에 의해 퍼지 연산이 시작된다. 각 모듈은 $CLK256$ 클럭의 마지막 클럭이 시작하면서 해당 연산의 끝을 알리는 END 신호를 발생한다.

$CLK256$ 클럭의 마지막 클럭동안 다음 상태를 위한 모든 초기화 동작이 이루어지고 다음 단계는 새로운 $CLK256$의 첫 번째 클럭과 함께 시작된다. 이러한 ripple 형태의 동작은 비퍼지화 모듈의 fine 탐색의 끝날 때까지 계속되며, fine 탐색이 끝나면 새로운 START 신호를 발생한다. 이러한 퍼지 제어 동작은 제어 대상물로부터 오는 $STOP$ 신호에 의해 종료된다.

Ⅳ. VHDL 시뮬레이션 및 결과 분석

제안한 FLC가 제대로 동작하는지를 트럭 후진 주차 문제에 테스트해 보았다. 트럭 주차 문제의 목표는 가능한 빨리, 그리고 정확하게 트럭을 주차시키는 것이며, 이 문제는 기존 제어 기술로는 풀기 힘든 전형적인 비선형 제어 문제이다. 그림 12는 트럭 주차 제어 문제에서 사용된 트럭과 주차대의 위치를 보여준다. 트럭의 위치는 (x, y, ϕ)에 의해 결정된다. 단, 여기에서 ϕ는 트럭 진행 방향과 x축간의 각도이며, 트럭의 후진 주행 제어는 ϕ와 핸들의 축 간의 각도인 θ에 의해 결정된다. 트럭이 움직이는 운동 방정식은 아래와 같이 나타내어진다.[10]

$$\begin{aligned} x(t+1) &= x(t) + \cos[\phi(t)+\theta(t)] \\ &\quad + \sin[\theta(t)]\cdot\sin[\phi(t)] \\ y(t+1) &= y(t) + \sin[\phi(t)+\theta(t)] \\ &\quad - \sin[\theta(t)]\cdot\sin[\phi(t)] \\ \phi(t+1) &= \phi(t) - \sin^{-1}\left[\frac{2\sin(\theta(t))}{b}\right] \end{aligned} \tag{14}$$

여기서 b는 트럭의 길이이며, 본 논문에서는 b = 4로 하였다.

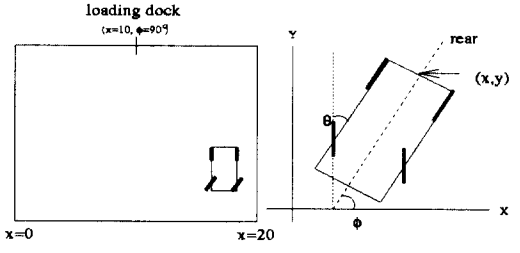

그림 11. 모형 트럭과 주차대 위치
Fig. 11. A model truck and loading dock.

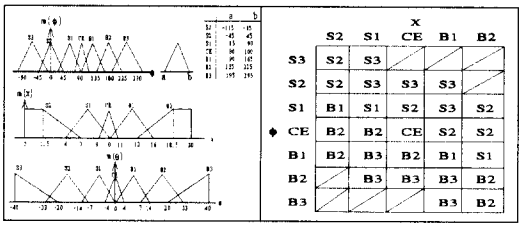

그림 12. 트럭 후진 주차 제어에 사용된 소속 함수와 퍼지 규칙 베이스
Fig. 12. Membership function and fuzzy rule base for the truck backer-upper con-trol.

만약 트럭과 주차대까지의 거리가 충분하다면 트럭이 $x=10$, $\phi=90^{\circ}$ 가까이 오면 트럭을 곧장 후진하기만 하면 되기 때문에 변수 y를 퍼지 입력 변수 (x, y, ϕ)에서 뺄 수 있다. 그러므로 트럭 주차 제어 문제는 주어진 공간내 $\{0\le x\le 20,\ -90^{\circ}\le\phi\le 270^{\circ}\}$ 임의의 초기 위치 (x_0, ϕ_0)에서 가능하면 신속·정확하게 주차대($x=10$, $\phi=90^{\circ}$) 쪽으로 후진하도록 바퀴 각도 $\theta(-40\le\theta\le 40)$ 를 제어하는 것이 요구된다. 그림 12는 Wang과 Mendel[11]이 사용한 퍼지 제어기의 입·출력 변수의 소속함수와 퍼지 제어를 위한 퍼지 규칙 베이스를 나타낸 것이다. 본 실험을 위해 앞에서 기술한 각 모듈을 하드웨어 기술 언어인 VHDL[12]을 이용하여 구조적 수준 및 행위적 수준에서 기술하여 이들 각 모듈을 SYNOPSYS사의 VHDL 시뮬레이터[13] 상에서 제대로 동작하는지를 각각 시뮬레이션 하였다. 나아가, 모델 트럭의 운동 방정식을 SYNOPSYS사가 제공하는 기본적인 IEEE 라이브러리 이외에도 Math-real package를 사용하여 행위적 수준에서 기술하여 제대로 동작하는지를 역시 SYNOPSYS사의 VHDL 시뮬레이터 상에서 제어 동작 과정을 시뮬레이션 하였다. 원하는 주차대 위치와 현재의 위치 사이의 오차를 계속 모니터링하는 경로 추적 모니터링 프로세스가 모델 트럭 프로세스와 연결되어 있는데, 이는 시뮬레이션 계속 여부를 조사하는 역할을 한다. VHDL 시뮬레이션 환경 관점에서 보면 설계된 FLC는 테스트중인 하나의 컴파일된 VHDL 컴포넌트 유닛 (Unit under test; UUT)으로 생각할 수 있으며, 모델 트럭은 testbed인 설계된 FLC에 stimulus 입력을 제공하는 하나의 프로세스로 볼 수 있다. 그림 13은 본 연구에서 사용한 FLC의 VHDL 시뮬레이션 환경을 나타낸 것이다.

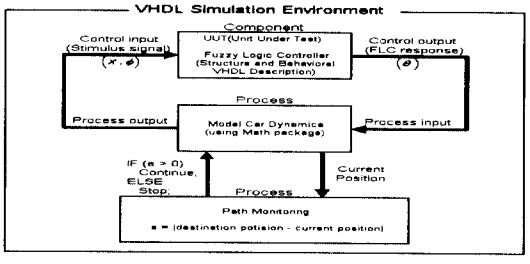

그림 13. VHDL 시뮬레이션 환경
Fig. 13. VHDL simulation environment.

FLC 시뮬레이션을 위한 VHDL 코드는 2개의 프로세스(CLK_GEN, PATH_MONITORING)과 2개의 컴포넌트(FLC_1, CAR_1)로 구성되어 있다.

```
ENTITY    Trace IS
END    Trace;
.
ARCHITECTURE Behaviour OF Trace IS
   COMPONENT  FLC
      PORT( CLK,CLR   :   IN  Std_Logic;
            PI,X      :   IN  TData;
            THETA:    OUT   TData;
            Valid     :   OUT   Std_Logic);
   END COMPONENT;

   COMPONENT  CAR
      PORT( CLK,ENABLE,  Init :IN Std_Logic;
        InitPI,InitX,InitY,THETA:IN Std_Logic_Vector;
        PI,X,Y  :   OUT   Std_Logic_Vector);
   END COMPONENT;

   FOR All : FLC USE ENTITY WORK.FLC(Structure);
   FOR All : CAR USE ENTITY WORK.CAR(Behaviour);
   SIGNAL
CLK,CLK2,START_FLC,END_FLC:Std_Logic;
   SIGNAL    THETA,PI,X:   TData;
   SIGNAL    Init,ENABLE,END_Tracing:Std_Logic;
   SIGNAL    Init_PI,Init_X,Init_Y: TData;
   SIGNAL    CAR_PI,CAR_X,Y   :   TData;
Begin
   CLK_GEN : PROCESS
   BEGIN
      CLK <= '0';
      WAIT FOR 10 NS;
      CLK <= '1';
      WAIT FOR 10 NS;
   END PROCESS;

   PATH_MONITORING   : PROCESS
      FILE      DataFile     : TEXT IS OUT "trace.dat";
      VARIABLE  L      : TEXT;
      VARIABLE state    : Natural := 0;
      VARIABLE String   : Bit_Vector(7 downto 0);
      VARIABLE INT : Integer;
   BEGIN
      WAIT UNTIL CLK'EVENT AND CLK = '1';
      CASE State Of
         WHEN 0   =>
            Init_PI          <= "11101111";
            Init_X           <= "10101111";
            Init_Y           <= "01111111";
            Init             <= '1';
            START_FLC        <= '0';
            ENABLE           <= '0';
            END_Tracing      <= '0';
         WHEN 1   =>
            ENABLE           <= '1';
         WHEN 3   =>
            ENABLE           <= '0';
            Init          <= '0';
            State         := 7;
         WHEN 5   =>
            ENABLE           <= '1';
         WHEN 7   =>
            ENABLE           <= '0';
         WHEN 8   =>
            START_FLC <= '1';
         WHEN 9   =>
            START_FLC <= '0';
         WHEN 10  =>
            PI       <= CAR_PI;
            X        <= CAR_X;
         WHEN 11  =>
            IF (END_FLC = '1') Then
               Write(l,NOW,Left,20);
               INt := StdTOI(CAR_PI);
               Write(l,int,left,10);
               int := StdToI(CAR_X);
               Write(l,int,left,10);
               int := StdToI(Y);
               Write(l,int,left,10);
               WriteLine(DataFile,l);
               IF (("01111000" < CAR_X)
                  And (CAR_X  < "10000111")
                    and ("01111000" < CAR_PI)
                    and (CAR_PI < "10000111"))Then
                    State := 12;
               END IF;
            END IF;
         WHEN 12  =>
            State        := 4;
         WHEN 13  =>
            END_Tracing    <= '1';
         WHEN 14  =>
            ASSERT(END_Tracing = '0')
            REPORT " END of tracing!"
            SEVERITY FAILURE;
         WHEN OTHERS=>
      END CASE;
      State := State + 1;
   END PROCESS;
   FLC_1 : FLC PORT MAP (CLK, START_FLC, PI,X,
                         THETA, END_FLC);
   CAR_1  :  CAR  Port  MAP(CLK,ENABLE,INIT,Init_PI,
             Init_X,Init_Y,THETA,CAR_PI,CAR_X,Y);
END Behaviour;
```

그림 14. FLC 시뮬레이션을 위한 VHDL 코드
Fig. 14. A VHDL code for FLC simulation.

CLK_GEN 프로세스는 시스템 동작은 위한 기본적인 클럭을 생성한다. PATH_MONITORING 프로세스는 시뮬레이션 전체의 초기화 및 모델 트럭의 위치를 파악하여 시뮬레이션 종료 여부를 결정한다. FLC_1 컴포넌트는 제안된 퍼지 제어기로서, 제어 입력 (ϕ, x)를 받아서 제어 출력 θ를 내보낸다. CAR_1 컴포넌트는 트럭 후진 주차 문제를 위해 설계된 모델 트럭으로 초기 위치 설정하고, 제어 출력 θ를 받아 새로운 위치 (ϕ, x)를 내보낸다.

그림 15과 16은 각각 식 (3)을 이용하여 비퍼지화 연산을 수행하는 제안한 COG 비퍼지화기와 식 (1)을 이용하여 비퍼지화 연산을 수행하는 기존의 COG 비퍼지화기를 포함하는 FLC가 사용될 때, 모델 트럭이 임의의 한 상태 $(x, y, \phi) = (13.5, 0.0, 225.0^\circ)$에서 출

발하여 경우의 VHDL 시뮬레이션 결과를 보인 것이다. 여기서 모든 변수값은 256 레벨로 양자화되어 있다. 예를 들면, 입력 변수 ϕ가 갖는 범위 [−90°, 270°]는 양자화 레벨 [00*H*, *FFH*]에 대응한다. 따라서, 시작 상태 (13.5, 0.0, 225.0°)는 한 16진수 (*AFH*, *7FH*, *EFH*)에 대응한다. 제안한 비퍼지화기를 포함하는 FLC가 목적지에 9 스텝만에 도달한 반면, 기존의 COG비퍼지화기를 포함하는 FLC는 목적지까지 15 스텝이 걸렸다.

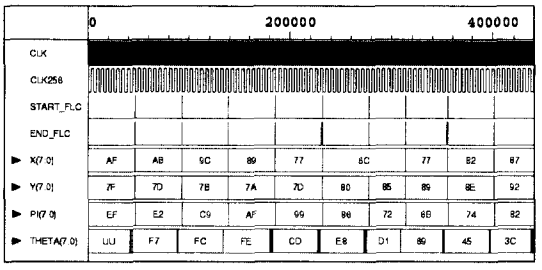

그림 15. 제안한 COG 비퍼지화기를 갖는 FLC의 VHDL 시뮬레이션 결과
Fig. 15. A VHDL simulation result of the FLC with the proposed COG defuzzifier.

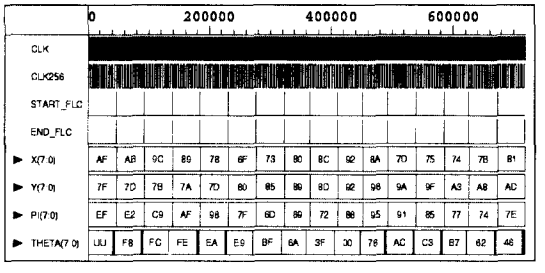

그림 16. 기존의 COG 비퍼지화기를 갖는 FLC의 VHDL 시뮬레이션 결과
Fig. 16. A VHDL simulation result of the FLC with the conventional COG defuzzifier.

그림 17은 하나의 퍼지 연산 동안 일어나는 MAX-MIN 추론과 비퍼지화 연산을 시간적으로 나타낸 것이다. 이 경우, 하나의 퍼지 연산을 수행하는데 *CLK*256의 주기로 8 사이클이 걸림을 알 수 있는데 각각 MAX-MIN 추론에 1 사이클, coarse 탐색에 6 사이클, find 탐색에 1 사이클 이 걸린다. 이 그림에서 좌측 인덱스 포인터(IDL)가 계속해서 6번 오른쪽으로 이동하는 것으로 보아서, 이 퍼지 연산의 경우는 출력 변수 상에 오직 하나의 절단된 소속 함수만이 존재하고 이는 가장 오른쪽 (7번째) 퍼지항에 놓여 있음을 알 수 있다.

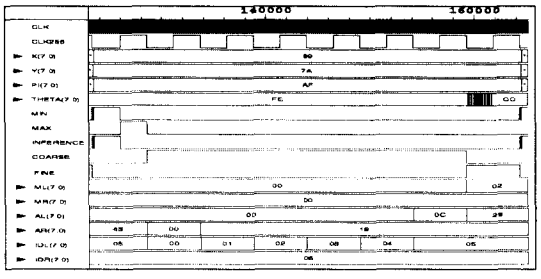

그림 17. 한 퍼지 연산의 수행 과정
Fig. 17. A snapshot of one fuzzy operation.

V. 결 론

본 논문은 시스템의 복잡도를 줄여 구현 비용이 적게 들며, 기존의 COG 비퍼지화기보다 정확한 제어를 수행하는 새로운 COG 비퍼지화기를 갖는 퍼지 제어기를 제안하여 이를 VHDL에 의해 구조적 또는 행위적 수준으로 기술한 뒤, 이들이 제대로 동작하는지 여부를 SYNOPSYS사의 VHDL 시뮬레이터 상에서 트럭 후진 주차 문제에 적용하여 검증하였다.

제안한 퍼지 제어기의 정확성은 비퍼지화 연산시 소속값뿐 아니라 소속 함수의 폭을 고려함으로서 얻어진다. 이 경우, 부가적인 곱셈기 요구에 의한 하드웨어 복잡도 증가 문제는 곱셈기를 확률론적 AND 연산에 의해 해결하였다. 트럭 후진 주차 문제에 적용한 결과, 목적지 주차대에 도달하는 데 걸리는 평균 주행 거리를 24.5% 이상 줄이는 성능 개선 효과를 얻을 수 있었다.

제안한 퍼지 제어기는 하드웨어 복잡도를 줄이기 위해 기존의 FLC내 MIN 및 MAX 연산을 레지스터 파일 구조상의 *read-modify-write* 연산에 의해 대치하였으며, 퍼지 규칙 베이스를 여러 개의 배타적 부규칙 베이스로 분할한 뒤, 이들 부규칙 lookup 테이블로부터 제어 규칙의 후단부를 독립적으로 동시에 얻어내었다. 기존의 COG 비퍼지화 연산은 나눗셈이 요구되는데 본 논문에서는 COG 비퍼지화 연산을 모멘트 균형점을 찾는 것으로 대신하여 하드웨어 복잡도를 크게 줄였다. 나아가, 보다 신속하게 모멘트 균형점을 찾기 위해 coarse 탐색시 모멘트 계산점의 이동은 퍼지항 단위로 이동시키고, fine 탐색시 coarse 탐색 결과 얻어진 두 인접항 사이에 모멘트 계산점을 단위 구간씩 이동시키는 coarse-to-fine 탐색법을 제시하고, 이 탐색 알고리즘 구현을 위한 회로의 블록도를 제시하였다.

제안한 퍼지 제어기가 실질적인 제어 문제에 사용 가능함을 확인하기 위해 재구성이 가능한 FPGA 시스템 상에 직접 구현하였으며, 그 자세한 구현 과정과 실험 결과는 다른 논문[14]에 뒤따른다. 구현 과정을 간략히 설명하면 다음과 같다. 각 모듈은 VHDL 언어에 의해서 기술된 뒤, SYNOPSYS사의 FPGA 컴파일러에 의해 합성된다. 합성된 각 모듈은 Xilinx 사의 XactStep 6.0에 의해 최적화 및 배치 배선이 이루어진다. 얻어진 Xilinx rawbit 파일은 VCC사의 r2h에 의해 C 언어의 header 파일 형태의 하드웨어 object로 변환된다. 원하는 목적을 수행하기 위해 이들 하드웨어 object를 포함하는 응용 프로그램이 컴파일된다. 이 실행 파일은 재구성 가능한 FPGA 시스템인 EVC1 보드를 원하는 목적을 수행 가능하도록 구성하기 위하여 하드웨어 object를 다운로드한다.

참 고 문 헌

[1] E. H. Mamdani, "Application of fuzzy algorithms for control of simple dynamicplant," *IEEE Proc. Control & Science,* vol. 121, no. 12, pp. 1585-1588, Dec. 1974.

[2] C. C. Lee, "Fuzzy Logic in Control Systems: Fuzzy Logic Controller," *IEEE Transactions on Systems, Man, and Cybernetics,* vol. 20, no. 2, pp. 404-435, Feb. 1990.

[3] 김대진, 조인현, "모멘트 균형점의 효율적 탐색을 갖는 비제산기 COA 비퍼지화기", 대한 전자 공학회 논문지, 33권 10호, pp. 138-151, 1996년 10월

[4] A. Ruitz, J. Gutiérrez, and J. Fernández, "A Fuzzy Controller with an Optimized Defuzzification Algorithm," *IEEE Micro,* pp. 1-10, Dec. 1995.

[5] Y. Kondo and Y. Sawada, "Functional Abilities of a Stochastic Logic Neural Network," *IEEE Transactions on Nerual Networks,* vol. 3, no. 3, pp. 434-443, May 1992.

[6] C. Gloster and F. Brglez, "Boundary scan with cellular-based built-inself-test," *IEEE International Test Conference,* pp. 138-145, 1988.

[7] Richard L. Scheaffer and James T. McClave, "Probability and Statistics for Engineers," *PWS-KENT Publishing Company,* Boston, USA, 1990.

[8] K. Hwang and F. A. Briggs, "Computer Architecture and Parallel Processing," *McGraw-Hill Book Company,* New York, USA, 1984.

[9] Daijin Kim, "Improving the Fuzzy System Performance by Fuzzy System Ensemble," *Fuzzy Set and System,* Accepted for publication.

[10] Li-Xin Wang And Jerry M. Mendel, "Generating Fuzzy Rules from Numerical Data with Applications," *USC-SIPI Report,* no.169, 1991.

[11] Li-Xin Wang and Jerry M. Mendel, "Generating Fuzzy Rules by Learning from Examples," *IEEE Transactions on System, Man, and Cybernetics,* vol. 22, no. 6, pp. 1414-1427, Nov. 1992.

[12] S. Mazor and P. Langstraat, A Guide to VHDL, Kluwer Academic Publishers, 1993.

[13] SYNOPSYS, "VSS Family Tutorial v3.4," *Synopsys Corp.*, 1994.

[14] 김대진, 조인현, "고정밀 저비용 퍼지 제어기(II) - 재구성 가능한 FPGA 시스템 상에 구현," 대한 전자 공학회 제출중

저 자 소 개

金 大 鎭(正會員) 第 33卷 B編 5號 參照
현재 동아대학교 컴퓨터공학과 조교수

趙 仁 顯(正會員) 第 33卷 B編 10號 參照
현재 동아대학교 컴퓨터공학과 석사 과정