퍼지 제어기의 설계 및 구현 자동화를 위한 통합 개발 환경
(An Integrated Development Environment for Automatic Design and Implementation of FLC)

조 인 현, 김 대 진
(Inhyun Cho, Daijin Kim)

요 약

본 논문은 저비용이면서 정확한 제어를 수행하는 새로운 퍼지 제어기의 VHDL 설계 및 FPGA 구현을 자동적으로 수행하는 통합 개발 환경(IDE; Integrated Development Environment)을 다룬다. 이를 위해 FLC의 자동 설계 및 구현의 전 과정을 하나의 환경 내에서 개발 가능하게 하는 퍼지 제어기 자동 설계 및 구현 시스템 (FLC Automatic Design and Implementation Station : FADIS)을 개발하였는데 이 시스템은 다음 기능을 포함한다. (1) 원하는 퍼지 제어기의 설계 파라메터를 입력받아 이로부터 FLC를 구성하는 각 모듈의 VHDL 코드를 자동적으로 생성한다. (2) 생성된 각 모듈의 VHDL 코드가 원하는 동작을 수행하는지를 Synopsys사의 VHDL Simulator상에서 시뮬레이션을 수행한다. (3) Synopsys사의 FPGA Compiler에 의해 VHDL 코드를 합성하여 FLC의 각 구성 모듈을 얻는다. (4) 합성된 모듈은 Xilinx사의 XactStep 6.0에 의해 최적화 및 배치, 배선이 이루어진다. (5) 얻어진 Xilinx rawbit 파일은 VCC사의 r2h에 의해 C 언어의 header 파일 형태의 하드웨어 object로 변환된다. (6) 하드웨어 object를 포함하는 응용 제어 프로그램의 실행 파일을 재구성 가능한 FPGA 시스템 상에 다운로드한다. (7) 구현된 FLC의 동작 과정은 구현된 FLC와 제어 target 사이의 상호 통신에 의해 모니터링한다. 트럭 후진 주차 제어에 사용하는 퍼지 제어기 설계 및 구현의 전 과정을 FADIS상에서 수행하여 FADIS가 완전하게 동작하는지를 확인하였다.

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 경우에 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.

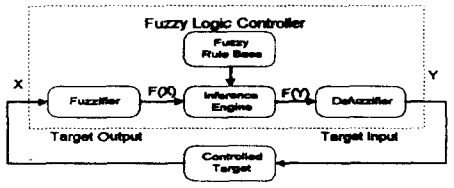


그림 1. 퍼지 제어기 일반적 구성도

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 데이터 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.

본 논문에서 사용된 고정밀 저비용 퍼지 제어기의 구조 및 특성은 다른 논문[2]에 잘 나타나 있다. 원하는 FLC를 FPGA 시스템 상에 구현할 경우, FPGA가 갖는 제사용 능력에 의해 여러 가지 다양한 형태를 갖는 퍼지 제어기의 구현이 가능하게 된다. 그림 1에서 보는 바와 같이 퍼지 제어기의 일반적인 구조는 불변이며, 응용에 따라 입출력 변수의 개수, 퍼지항의 개수, 소속 함수값, 제어 규칙 등이 달라지게 된다. 따라서, FLC 구현시 이들 설계 파라메터가 변할 때마다 이에 상응하는 FLC의 VHDL 코드를 자동적으로 생성해주는 기능은 매우 필요하다. 나아가, 생성된 VHDL 코드로부터 FLC를 FPGA 시스템 상에 구현하는 과정에 여러 가지 상이한 CAD 장비(Synopsys VHDL Simulator, Synopsys FPGA Compiler, Xilinx's XactStep 6.0, 및 VCC's H.O.T)들의 지원이 요구되며, 구현된 FLC의 동작 상태를 확인하기 위해 GUI 환경의 모니터링 윈도우의 구현도 필요하다. 이상의 여러 기능, 장비, 및 인터페이스 등을 하나의 통합된 환경에서 처리, 지원해줄 수 있는 통합 개발 환경(IDE)의 구축이 요구된다. 본 연구에서는 이를 위해 설계 파라메터 입력에서 구현된 퍼지 제어기의 동작 모니터링까지의 전 과정에서 일어나는 코드 발생, 시뮬레이션, 합성, 최적화, 배치 및 배선, 하드웨어 object 변환, 다운로드등을 하나의 통합 환경에서 수행 가능하게 해주는 퍼지 제어기 자동 설계 및 구현 시스템(FLC Automatic Design and Implementation Station)을 개발하고자 한다.

본 논문의 구성은 다음과 같다. II장에서는 고정밀 저비용 특성을 갖는 새로운 FLC의 아키텍쳐어를 설명한다. III장에서는 제안한 FLC 구현의 Front-End 과정으로 설계 파라메터로 VHDL 코드의 자동 생성, FPGA 합성 및 최적화, 배치, 배선을 설명한다. VI장에서는 제안한 FLC 구현의 Back-End 과정으로 하드웨어 object 기술과 재구성 가능한 FPGA 시스템 구현을 설명한다. V장에서는 위의 전 과정을 통합하여 설계 및 구현을 자동화한 FADIS에 대해 설명한다. VI장에서는 트럭 후진 주차 문제에 적용한 FLC를 FADIS상에서 구현한 결과를 보인다. 마지막으로 결론과 앞으로의 연구 방향을 언급한다.

II. 제안한 FLC의 아키텍쳐어

FLC를 하드웨어적으로 구현하는 경우, 하드웨어의 복잡도를 줄이고 제어 성능을 높이기 위해 다음과 같은 제약을 가한다[6].

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.

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 소속 함수 중심에서의 단일 퍼지값으로 대신할 수 있게 됨으로서 COG 비퍼지화값의 계산 복잡도를 크게 줄일 수 있다. 네 번째 제약은 COG 비퍼지화값 계산시 소속 함수값과 폭이 동시에 고려됨으로서 제어 성능이 개선된다. 아래 그림 2는 제안한 FLC의 하드웨어 구조를 나타낸 것으로, MIN 모듈, 추론 모듈, MAX 모듈, 비퍼지화 모듈, 및 제어 모듈로 구성되어 있다. 각 모듈의 구조 및 동작 설명은 다른 논문[2]에 잘 나타나 있다. 제안한 퍼지 제어기의 설계 파라메터는 다음과 같다.

- 입력 변수 개수 = 2 : (x, ϕ)
- 출력 변수 개수 = 1 : (θ)
- 입력 변수의 소속함수 개수 = (5, 7)
- 출력 변수의 소속함수 개수 = (7)
- 입 · 출력 변수의 크기 해상도 = 8 비트 = 256 레벨
- 소속 함수의 크기 해상도 = 8 비트 = 256 레벨
- 소속 함수의 범위 = [0-255]
- 소속 함수간 최대 중첩도 = 2
- coarse-to-fine 모멘트 탐색에 의한 COG 비퍼지화

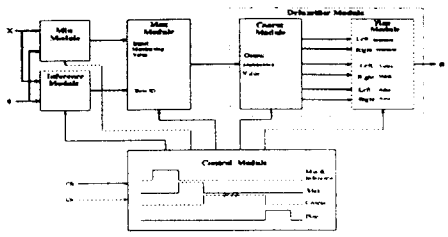


그림 2. 제안한 FLC의 하드웨어 구조

III. 제안한 FLC 구현의 전단부(Front-End) 처리.

FLC 구현의 전단부 처리 과정은 (1) 원하는 FLC의 Configuration 파라메터로부터 VHDL코드를 자동적으로 생성하고, (2) 생성된 VHDL 코드가 원하는 대로 동작하는지를 행위 및 구조 수준에서 시뮬레이션을 수행하고, (3) FPGA 컴파일러를 사용하여 시뮬레이션을 끝낸 VHDL 코드로부터 FPGA 구성요소로 구성된 netlist를 자동적으로 얻는 과정으로 이루어진다. 다음은 각 단계를 자세히 설명하는 것이다.

1. VHDL 코드 자동 생성

FLC의 configuration 파라메터로부터 VHDL 컴포넌트를 자동 생성하기 위해서는 제어기가 갖는 하드웨어 구조의 형태가 일정해야 한다. 그림 4에서 알 수 있듯이 입출력 변수의 개수가 변하는 경우에는 요구하는 레지스터의 개수가 변하고, 입출력 변수의 해상도가 변하면, 사용되는 레지스터나 수치 연산기의 크기만 변할 뿐, 전체적인 하드웨어의 구조 형태는 항상 동일하게 유지된다. 이러한 성질을 설계 파라메터로부터 퍼지제어기를 기술하는 VHDL 코드를 자동적으로 생성 가능하도록 한다.

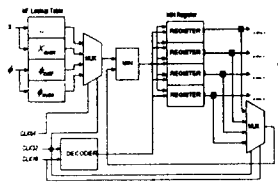


그림 3-(a). 8비트 2입력 FLC의 MIN 블록

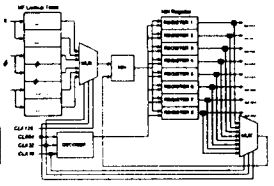


그림 3-(b). 4비트 3입력 FLC의 MIN 블록

그림 3은 서로 다른 설계 파라메터에 대해 설계된 FLC를 보인 것이다. 그림 3-(a)는 입출력 변수의 크기 해상도가 8 비트이고, 입력 변수가 2개인 경우로 두입력에 대해 4개의 MF lookup 테이블이 필요하고, MIN 연산의 수행 결과를 저장할 4개의 레지스터가 필요하다. 이 경우, lookup 테이블, MUX, MIN 레지스터의 해상도는 각각 8 비트이다. 그림 3-(b)는 입출력 변수의 크기 해상도가 4비트이고, 입력 변수가 3개인 경우로 세입력에 대해 6개의 MF lookup 테이블이 필요하다. 이 경우, lookup 테이블, MUX, MIN 레지스터의 해상도는 각각 4비트이다. 두 모듈은 서로 다른 입출력 파라메터를 가짐으로 인해, MF lookup 테이블, MIN 레지스터, MUX의 갯수와 해상도만 다를 뿐, 전체적인 구조는 동일하다. 이것은, 기본적인 원형판 모듈에 입출력 파라미터에 맞도록 테이블이나 레지스터를 생성하고, 입출력 변수의 크기 해상도를 맞춤으로서 원하는 VHDL 컴포넌트의 생성이 가능하다. 그림 4는 MIN 블록의 원형판을 나타낸 것이다.

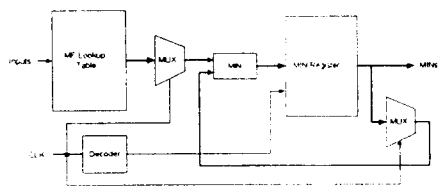


그림 4. MIN 블록의 원형판

주어진 설계 파라메터로부터 VHDL 코드를 생성하는 과정은 다음과 같은 계층적 설계 기법을(Design Hierarchy) 이용한다. (1) 각 모듈의 기본 구조를 나타내는 원형판과 VHDL 코드를 준비한다. (2) 각 입출력 파라메터 값에 맞도록 레지스터나 MUX등에 대응하는 VHDL 코드를 생성하여 원형판의 VHDL 코드와 상호 연결시킨다. 여기서, 두 번째 과정의 예로서 MF Lookup 테이블과 레지스터 파일의 경우를 설명한다. 두 블록은 다른 각기 기능을 수행하지만 동일한 형태를 지니고 있다. MF lookup 테이블은 입력을 주소로 사용하여 테이블 내에서 입력주소에 대한 소속값을 출력한다. 레지스터 파일은 MIN 회로에서 얻어진 결과를 저장해 두고, MIN 레지스터의 연산이 완료되면, 저장해 두었던 값을 결과로 출력한다. 여기서, 두 블록은 인덱스에 의해 참조되는 배열 형태로 표현 가능하다. 배열은 초기에 원소들의 크기 해상도와 전체 원소들의 개수만 결정되면 항상 동일한 기능을 수행한다. 따라서, MF lookup 테이블과 레지스터 파일은 입출력 파라메터에 의해 입력 변수의 개수와 크기 해상도만 결정되면 배열의 크기가 결정되므로 자동 생성할 수 있다. 또다른 예로서, MF lookup 테이블의 출력과 MIN 레지스터의 값을 순차적으로 참조하여 새로운 MIN 결과를 만들 수 있도록 제어해 주는 MUX와 디코더를 든다. MUX와 디코더는 expression문과 when문으로 이루어진 case문에 의해 기술되는데, expression에 들어오는 변수값에 따라 when 문 중 하나가 선택되어 선택된 when문 다음에 오는 문장을 수행한다. 그리고, case문 내의 when문의 개수는 expression문에 대한 표현 가능한 값의 범위만큼 확장할 수 있다. 이러한 두 성질은 입출력 변수 개수의 변수에 대한 다양한 형태의 MUX와 디코더의 VHDL 코드 생성을 가능하게 한다. 그림 5-(a)는 8비트 4입력의 MUX를 출력 파라메터에 의해 VHDL 컴포넌트를 자동 생성하기 위한 C++ 코드이며, 그림 5-(b)는 이에 의해 생성된 VHDL 컴포넌트를 보인 것이다.

```
Void MakeMUX(int numberOfInput, int resolution)
{
  cout <<"ENTITY MUX_"<<numberOfInput << " IS" <<endl;
  cout << "\tPort (" << endl;
  int   i;
  for( i=0;  i<numberOfInput; i++)
    cout<<"\t\tDataIn"<<i<<" : IN BIT_VECTOR("
        << resolution-1 <<"downto 0);"<<endl;
  cout<<"\t\tSEL : INTEGER;"<<endl;
  cout<<"\t\tDataOut"<< " : OUT BIT_VECTOR("
    << resolution-1 << "downto 0));"<<endl;
  cout << "END MUX_" << numberOfInput << ";" << endl;
  cout << "ARCHITECTURE BEHAVIOUR OF MUX_"
    <<numberOfInput<<"IS"<< endl;
  cout << "BEGIN" <<endl;
  cout << "\tPROCESS(";
  for( i=0; i<numberOfInput ; i++)
    cout << "DataIn"<< i <<",";
  cout << "Sel)" <<endl;
  cout << "\tBEGIN" <<endl;
  cout << "\t\tCASE sel IS"<< endl;
  for(i=0; i < numberOfInput ; i++){
    cout << "\t\t\tWHEN " << i << " =>" << endl;
    cout << "\t\t\t\tDataOut <= DataIn" << i << ";" << endl;
  }
  cout <<"\t\tEND CASE;"<<endl<< "\tEND PROCESS;" << endl;
  cout << "END BEHAVIOUR;" << endl;
};
```

그림 5-(a). MUX를 위한 VHDL code 생성 함수

```
ENTITY MUX_4 IS
  PORT(
    DataIn0, DataIn1, DIn2,  DataIn3 : IN BIT_VECTOR(7 Downto 0);
    SEL   : IN INTEGER;
    DataOut  : OUT BIT_VECTOR(7 Downto 0));
END MUX_4;
ARCHITECTURE BEHAVIOUR OF MUX_4 IS
BEGIN
  PROCESS(DataIn0,DataIn1,DataIn2,DataIn3,SEL)
  BEGIN
    CASE SEL IS
      WHEN  0 =>
        DataOut <= DataIn0;
      WHEN  1 =>
        DataOut <= DataIn1;
      WHEN  2 =>
        DataOut <= DataIn2;
      WHEN  3 =>
        DataOut <= DataIn3;
    END CASE;
  END PROCESS;
END BEHAVIOUR;
```

그림 5-(b). VHDL code 생성 함수에 의해 얻어진 MUX의 VHDL 코드

제안된 FLC의 [illegible] 모듈과 MAX 모듈을 위한 VHDL 코드로

앞에서 설명한 MIN 모듈의 경우와 같은 방식에 의해 자동으로 생성된다. 비퍼지화 모듈의 경우는 앞서의 모듈들과 달리 출력 변수에만 의존한다. 따라서, 비퍼지화 모듈의 coarse 모듈과 fine 모듈을 위한 VHDL 코드는 출력 변수 파라메터들에 영향을 받는다. Coarse 모듈은 출력 변수가 가질 수 있는 퍼지항들의 개수에 따라 사용되는 레지스터 수가 결정되고 출력 변수의 크기 해상도에 따라 이들 레지스터의 해상도가 결정된다. Fine 모듈은 단지 출력 변수가 갖는 해상도에 따라 레지스터의 해상도를 결정하기만 하면 된다. 그림 6은 fine 모듈의 경우 레지스터 해상도를 결정하는데 출력 변수의 해상도인 8비트를 입력받고, 이를 package내에 constant 데이터형으로 정의한 다음 이를 fine 모듈의 VHDL 코드 상에서 사용된 예를 보인 것이다.

```
PACKAGE Types IS
    CONSTANT  RESOLUTION : INTEGER := 8;
END Types;
ENTITY FINE IS
    PORT( LeftMoment : BIT_VECTOR(RESOLUTION-1 DOWNTO 0);
        :
    );
END FILE;
```

그림 6. Constant 형을 이용한 입출력 변수의 크기 해상도 조정

2. VHDL 시뮬레이션

앞에서 얻어진 각 모듈의 구조적 또는 행위적 수준에서 기술된 VHDL 코드가 제대로 동작하는지를 확보하기 위하여 미리 VHDL 시뮬레이션을 하는 것이 요구된다. 이를 위하여 본 연구에서는 Synopsys사의 VHDL 시뮬레이터를 사용하였다. 시뮬레이션을 위해 얻어진 제어기의 VHDL 코드를 트럭 후진 주차 제어에 적용하여 보았다. 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 7은 본 연구에서 사용한 FLC의 VHDL 시뮬레이션 환경을 나타낸 것이다[7].

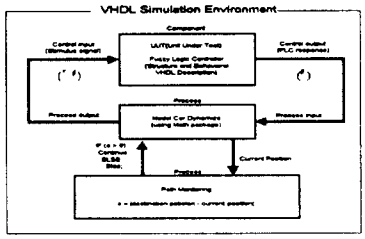


그림 7. VHDL 시뮬레이션 환경

3. FPGA 합성

제안한 FLC를 재구성 가능한 FPGA 시스템 상에 구현하기 위한 첫 단계는 VHDL 언어로 기술된 각 모듈을 게이트 수준으로 netlist로 합성하는 것이다. 이를 위해 본 연구에서는 Synopsys사의 FPGA 컴파일러를 사용하였다.

제안한 FLC의 각 모듈을 합성하기 과정은 모든 모듈에 대해서 동일하므로 본 논문에서는 가장 복잡한 COG 비퍼지화기의 coarse 모듈을 중심으로 설명하고자 한다. 아래 그림 8은 Synopsys사의 FPGA 컴파일러[3] 상에서 coarse 모듈을 합성하기 위해 사용된 명령어 스크립트 파일을 나타낸 것이다.

```
read -f vhdl {synopsys.vhd}
read -f vhdl {address.vhd}
read -f vhdl {sbus_if.vhd}
read -f vhdl {mem_if.vhd}
read -f vhdl {coarse.vhd}
read -f vhdl {top.vhd}
compile -ungroup_all
replace_fpga
write  format xnf -output coarse.sxnf
```

그림 8. 합성을 위한 FPGA 컴파일러의 컴파일 스크립트

위의 명령어 파일의 내용을 간단하게 설명하면 다음과 같다. FPGA 컴파일러가 VHDL 파일을 받아들여(read 명령) 원하는 제약 조건하에서 합성한 후(compile 명령) 합성 결과를 게이트 레벨의 netlist 파일 (이 경우 synopsys xnf 파일)로 저장한다(write 명령). 여기서 6개의 읽혀지는 VHDL 파일은 각각 합성시 사용되는 연산자 package, 입출력 포트 어드레스를 정의한 package, workstation의 S버스 인터페이스 프로그램, 메모리 인터페이스 프로그램, coarse 모듈 프로그램, 그리고 앞의 5개 VHDL 프로그램을 결합한 최상위 프로그램을 의미한다. 그리고, replace_fpga 명령[9]은 컴파일 결과 얻어진 파일내 존재하는 CLB셀이나 IOB셀 등을 Xilinx FPGA 칩내에서 실제 사용 가능한 기본 셀(and 또는 or 등 게이트 수준)로 바꾸어 주는 역할을 한다. 아래 그림 9는 Synopsys상의 FPGA 컴파일러에 의해 위의 명령어 파일에 의해 합성된 coarse 모듈의 스키메틱 view로서 S버스 인터페이스, 메모리 인터페이스 및 COG 비퍼지화기를 포함한다.

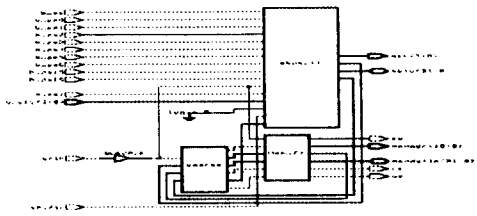


그림 9. 합성된 coarse 모듈 스키메틱 표시

위와 같은 과정을 제안한 FLC의 각 모듈에 적용하여 게이트 수준으로 합성된 각 모듈의 netlist 파일(min.sxnf, max.sxnf, inference.sxnf, coarse.sxnf, fine.sxnf, control.sxnf)을 얻음으로서 FLC 구현의 첫단계가 끝난다.

IV. 제안한 FLC 구현의 후단부(Back-End) 처리

FLC 구현의 후단부 처리 과정은 (1) 얻어진 netlist 들의 최적화, 배치 및 배선을 하여 bitstream 파일을 얻고 (2) 각 모듈의 하드웨어 object로 변환하고 재구성 가능한 FPGA 시스템으로 다운로드시킨다.다음은 각 단계를 자세히 설명한 것이다.

1. 최적화, 배치 및 배선

전단부에서 얻어진 netlist 파일(*.sxnf 파일)을 Xilinx FPGA상에 적합하도록 배치 및 배선을 하여 논리 셀 어레이 파일(*.lca 파일)을 얻기 위해 본 연구에서는 Xilinx사의 XactStep 6.0 Xilinx FPGA 개발 시스템[4]을 사용하였는데 그림 10은 이 시스템 상에서 FPGA 구현시 일어나는 과정을 나타낸 것이다.

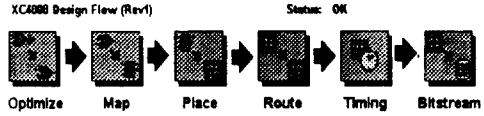


그림 10. XactStep 6.0상에서 일어나는 최적화, 배치 및 배선 과정

아래 그림 11는 Xilinx사의 XactStep 6.0상에서 coarse 모듈을 최적화, 배선 및 배치를 하기 위해 사용된 명령어 파일을 나타낸 것이다. 다른 모듈들은 이 그림에서 coarse 대신 해당 모듈명을 대신하면 된다.

```
flc.xnf : flc.sxnf   # synopsys XNF file to xilinx XNF
          syn2xnf -p 4013PQ208 flc.sxnf
flc.xff : flc.xnf # merge
          xnfmerge -A -D xnf -P 4013PQ208-3 flc.xnf flc.xff
flc.xtf : flc.xff # DRC & optimization
          xnfprep flc.xff flc.xtf parttype=4013PQ208-3
flc.lca : flc.cst flc.xtf # placement & routing
          ppr flc.xtf parttype=4013PQ208-3
          xdelay -D -W flc.lca
flc.rbt : flc.lca # make raw bit file
          makebits -b flc.lca
```

그림 11. 최적화, 배치 및 배선을 위한 XactStep 6.0의 스크립트

위 명령어 프로시쥬어에 대한 간단한 설명은 다음과 같다. 먼저 syn2xnf 명령어는 Synopsys사의 FPGA 컴파일러가 만든 netlist를(*. sxnf 파일) Xilinx사의 XactStep 6.0에서 사용 가능한 netlist로(*.xnf 파일) 변환시키는 역할을 한다. 다음 xnfmerge 명령어는

여러 개의 xnf 파일을 다음에 오는 배치 및 배선을 쉽게 하도록 하기 위하여 하나의 평처진(flat) netlist 파일 (*.xff)로 변환시키는 역할을 한다. 다음 xnfprep 명령어는 설계 규칙에 맞는가를 검증하고(DRC) 디바이스에 독립적으로 얻어진 netlist 형태를 특정 target인 FPGA의 형태(*.xtf)에 적합하도록 변환시키는 역할을 한다. 다음 ppr 명령어는 변환된 netlist를 Xilinx의 기본 구성 요소인 CLB와 IOB로 매핑하고, 매핑된 CLB와 IOB를 연결하는 선의 길이가 최소가 되도록 배치시키며, 마지막으로 위치가 정해진 CLB와 IOB간을 최소 지연 시간을 갖도록 연결시킨다. ppr 명령어 수행시 입력으로 사용되는 *.cst 파일은 FPGA가 갖는 실제 핀에 FPGA가 원하는 동작을 하도록 하는데 요구되는 기능적 핀을 매핑시킨 정보를 갖고 있어 원하는 대로 회로를 쉽게 변경 가능하게 한다. 다음 makebits 명령어는 Xilinx FPGA에 원하는 기능을 하도록 하는 configuration 파일에 대응하는 비트열을 자동적으로 발생시킨다.

FPGA가 갖는 CLB 및 IOB 개수가 유한하므로 제안한 FLC를 하나의 FPGA에 구현하는 것을 불가능하므로 FLC를 어떻게 분할하는 것이 효과적인가 하는 분할 문제에 부딪히게 된다. 본 연구에서는 제안한 FLC의 각 모듈을 하나의 FPGA에 대응시킴으로서 이 문제를 간편하게 해결하였다. 그림 12는 각 모듈의 배치 및 배선이 끝난 결과 사용한 CLB, flipflop, 및 I/O 핀의 백분율을 나타낸 것이다. 이 그림으로부터 하나의 FPGA에 하나의 모듈을 매핑시키는 분할법이 아주 효과적이지는 않지만 각 모듈내의 interface 부분이 서로 동일하므로 실행 시간에 부분적으로 재구성이 가능한 FPGA 시스템을 사용하는 경우 재구성 시간을 줄일 수 있는 장점이 있다.

	Min	Inference	Max	Coarse	Fine
CLBs	308(53%)	196(34%)	353(61%)	510(88%)	362(62%)
F/Fs	249(21%)	149(12%)	253(21%)	294(25%)	185(16%)
I/O pins	102(63%)	102(63%)	102(63%)	102(63%)	102(63%)

그림 12. 각 모듈의 FPGA 요소 사용율

2. 재구성 가능한 FPGA 시스템 상에 구현

재구성 가능한(reconfigurable) 컴퓨팅 시스템은 FPGA가 가지는 재구성 능력을 이용하여 구현하고자 하는 응용 알고리즘내 연산시간이 많이 걸리는 부분을 hardwiring에 의해 구현함으로서 알고리즘 수행시간을 크게 줄여줄 수 있는 능력을 나타낸다[10]. 이 시스템의 가장 중요한 구성 요소는 SRAM 기반 FPGA로서 기능이 미지정된(uncommitted) 많은 프로그래머블 논리 게이트와 프로그래머블 연결선으로 구성된 집적 회로의 일종으로 최종 사용자에 의해 원하는 기능을 나타내도록 이들 게이트와 연결선이 구성(또는 재구성)된다. 현재 나오고 있는 재구성 가능한 컴퓨팅 시스템은 대부분 호스트 컴퓨터에 대한 co-processing 디바이스 역할을 하며 호스트 컴퓨터의 슬롯에 직접 꽂을 수 있는 형태로 개발되고 있다. 본 연구에서 사용한 재구성 가능한 FPGA 시스템은 VCC사가 개발한 EVC1 보드로 SUN 워크스테이션의 S버스 슬롯에 직접 꽂아서 사용할 수 있다.

알고리즘을 재구성 가능한 FPGA 시스템 상에서 hardwiring에 의해 구현하는 방법에는 크게 두 가지가 있다[11]. 하나는 컴파일 시점(compile-time) 재구성법이고 다른 하나는 실행 시점(run-time) 재구성법이다. 전자는 원하는 알고리즘을 수행하는 동안 하나의 FPGA가 하나의 동일한 configuration을 계속 유지하는 것을 말하므로 기존의 EDA 설계 장비로서도 원하는 알고리즘을 충분히 구현 가능하다. 후자는 알고리즘이 시간적으로 서로 무관한 여러 개의 부분으로 분할될 수 있는 경우, 하나의 FPGA가 각 분할에 대응하는 configuration을 여러 단계에 걸쳐 동적으로 가질 수 있는 것을 말하므로 기존의 EDA 설계 장비로서는 실행 시점 재구성에 의해 알고리즘을 구현하기는 불편한 점이 있다.

후자는 다시 각 단계 수행마다 FPGA의 configuration이 변하는 형태에 따라 전체적 run-time 재구성과 국부적 run-time 재구성으로 나뉜다. 전자는 각 단계 수행시 FPGA의 configuration이 전체적으로 바뀌어지는데 반해, 후자는 각 단계 수행시 FPGA의 configuration내 일정 부분만 원하는 기능을 갖도록 재구성되고 나머지는 변하지 않고 그대로 유지되는 것을 말한다. 그림 19는 앞에서 설명한 여러 가지 재구성 방법을 예시한 것이다. EVC1보드내 장착된 Xilinx FPGA XC4013PQ208[8]은 국부적 변경이 불가능하므로 본 연구에서는 전체적 run-time 재구성법에 의해 알고리즘을 구현하였다. 현재 출시되고 있는 XC6020은 run-time시 국부적 재구성이 가능하므로 재구성 시간이 크게 단축되고 응용분야가 더욱 넓어질 것으로 전망된다. 나아가, 실행 시점 재구성에 의한 알고리즘 구현 시에는 앞선 configuration의 중간 결과를 뒤따르는 configuration이 입력 데이터로 사용하는 경우가 흔하므로 이를 위해 configuration간의 데이터를 주고받을 수 있도록 메모리 보드가 요구된다. 이를 위해 본 연구에서는 2M SRAM 보드를 사용하였다.

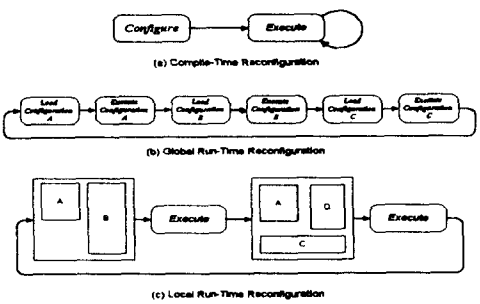


그림 13 여러 가지 재구성 방법

앞에서 설명한대로 run-time 재구성에 의해 원하는 알고리즘을 구현하는데 기존의 EDA 설계 장비를 이용하는데는 불편한 점이 따른다. 이를 해소하기 위해 본 연구에서는 VCC사가 개발한 하드웨어 object 기술(Hardware Object Technology, H.O.T)[5]을 사용하였다. 이 기술에 의하면 EVC1에 원하는 알고리즘을 구현하는 과정은 다음과 같다[11].

1) 앞에서 얻어진 FLC의 각 모듈의 raw 비트 파일 (*.rbt)을 r2h 명령어에 의해 동적으로 다운로딩 가능하도록 하드웨어 object인 header 파일로 (*.h) 변환시킨다. 얻어진 하드웨어 object는 virtual processing 라이브러리에 저장되어 재사용이 가능하다.
2) 하드웨어 object는 원하는 제어 목적에 맞는 응용 프로그램을 C언어로 작성할 때 필요하면 header 파일로 불려와 EVCdownload 명령어에 의해 EVC1 보드에 다운로드되어 FPGA를 configuration한다.
3) 하나의 configuration이 수행을 종료하면 다음 configuration을 위해 EVC1보드를 EVC reset 명령어에 의해 초기화시킨다.

EVC1 보드 상에 하드웨어 object를 다운로드시켜 원하는 기능을 수행하기 위해서는 Host (SUN 워크스테이션)와 EVC1 보드내의 FPGA 상에 다운로드된 하드웨어 object (사용자 정의 모듈) 사이에 데이터를 주고받기 위한 S버스 인터페이스와 EVC1 보드내의 FPGA 상에 다운로드된 하드웨어 object와 2M SRAM 보드 상에 데이터를 읽고 쓰기 위한 메모리 인터페이스를 FPGA상에 구현하는 것이 요구된다. 이들 두 인터페이스 로직은 FPGA내 CLB의 약 1.5%에서 11%정도를 사용하므로 실제 하드웨어 object 구현하는데 사용되는 CLB수는 그만큼 줄어든다. 아래 그림 14는 EVC1 보드내 FPGA상에 구현된 두 개의 인터페이스와 주변 Host 컴퓨터와 메모리 보드사이의 연결을 나타낸 것이다.

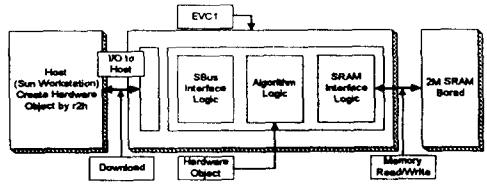


그림 14. 두 인터페이스와 주변 시스템사이의 관계

메모리 인터페이스는 입출력 버퍼들과 읽기/쓰기 제어 신호들로 구성되어 있다. 메모리 인터페이스와 메모리 모듈의 연결은 입출력이 동일한 양방향 포트에 의해 이루어지고, 사용자 정의 모듈과의 연결은 단방향 포트 2개로 이루어진다. 메모리 모듈은 read, write, CS 신호에 의해 제어되고, 메모리 모듈과 연결된 양방향 포트는 read, write 신호에 의해 제어된다.

V. 퍼지 제어기 설계 및 구현 자동화를 위한 통합 개발 환경

제안한 퍼지 제어기 설계 및 구현 방법은 여러 가지 입출력 파라메터들로부터 VHDL 코드를 자동 생성하고, 시뮬레이션, 합성, 배치 및 배선 과정을 거쳐 재구성 가능한 FPGA 시스템 상에 구현할 수 있음을 보았다. 이러한 과정을 거치면서 여러 종류의 변환 프로그램과 CAD 장비가 사용됨을 알 수 있다. 따라서 설계에서 구현까지 전 과정에 걸리는 개발 시간을 단축시킬 수 있고, 각

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 10는 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]

$$\begin{aligned} x(t+1) &= x(t) + \cos[\phi(t) + \theta(t)] + \sin[\theta(t)] \cdot \sin[\phi(t)] \\ y(t+1) &= y(t) + \sin[\phi(t) + \theta(t)] - \sin[\theta(t)] \cdot \sin[\phi(t)] \\ \phi(t+1) &= \phi(t) - \sin^{-1}\left[\frac{2\sin(\theta(t))}{b}\right] \end{aligned} \quad (12)$$

여기서 b는 트럭의 길이이며, 본 논문에서는 b = 4로 하였다.

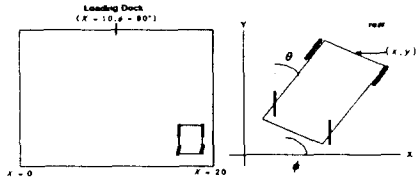
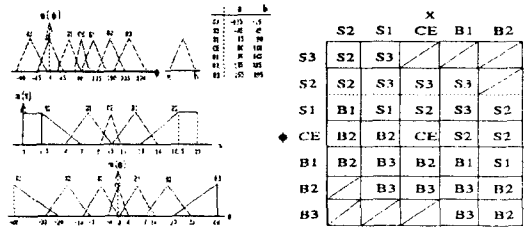


그림 10. 모형 트럭과 주차대 위치

만약 트럭과 주차대까지의 거리가 충분하다면 트럭이 $x=10$, $\phi=90°$ 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 $\{0 \le x \le 20,\ -90° \le \phi \le 270°\}$ 임의의 초기 위치 (x_0, ϕ_0)에서 가능하면 신속·정확하게 주차대($x=10$, $\phi=90°$) 쪽으로 후진하도록 바퀴 각도 $\theta(-40 \le \theta \le 40)$를 제어하는 것이 요구된다. 그림 11는 Wang과 Mendel[11]이 사용한 퍼지 제어기의 입·출력 변수의 소속함수와 퍼지 제어를 위한 퍼지 규칙 베이스를 나타낸 것이다.



	S2	S1	CE	B1	B2
S3	S2	S3			
S2	S2	S3	S3	S3	
S1	B1	S1	S2	S3	S2
CE	B2	B2	CE	S2	S2
B1	B2	B3	B2	B1	S1
B2		B3	B3	B3	B2
B3				B3	B2

그림 11. 트럭 후진 주차 제어에 사용된 소속 함수와 퍼지 규칙 베이스

본 실험을 위해 앞에서 기술한 각 모듈을 하드웨어 기술 언어인 VHDL[12]로 기술하고, 모델 트럭의 운동 방정식을 SYNOPSYS사가 제공하는 기본적인 IEEE 라이브러리 이외에도 Math-real package를 사용해 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 12은 본 연구에서 사용한 FLC의 VHDL 시뮬레이션 환경을 나타낸 것이다.

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

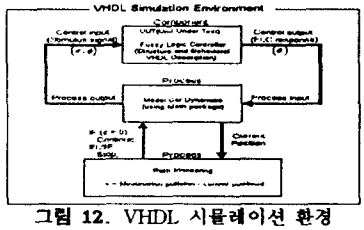


그림 12. VHDL 시뮬레이션 환경

그림 13과 14은 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, ϕ)= (13.5, 0.0, 225.0°)에서 출발하여 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 [00H, FFH]에 대응한다. 제안한 비퍼지화기를 포함하는 FLC가 목적지에 9 스텝만에 도달한 반면, 기존의 COG비퍼지화기를 포함하는 FLC는 목적지까지 15 스텝이 걸렸다.

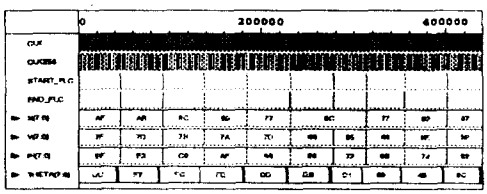


그림 13. 제안한 COG 비퍼지화기를 갖는 FLC의 VHDL 시뮬레이션 결과

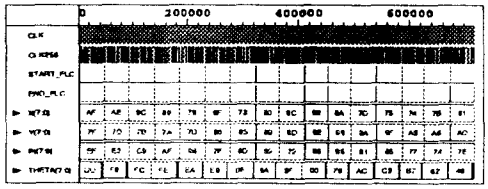


그림 14. 기존의 COG 비퍼지화기를 갖는 FLC의 VHDL 시뮬레이션 결과

그림 15은 하나의 퍼지 연산 동안 일어나는 MAX-MIN 추론과 비퍼지화 연산을 시간적으로 나타낸 것이다. 이 경우, 하나의 퍼지 연산을 수행하는데 의 주기로 8 사이클이 걸림을 알 수 있는데 각각 MAX-MIN *CLK*256 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.

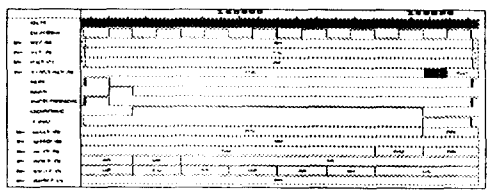


그림 15. 한 퍼지 연산의 수행 과정

V. 결론

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 Xilinx사의 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.

참고 문헌

[1] E. H. Mandani, "Application of fuzzy algorithms for control of simple dynamic plant," *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 비계산기 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.