FADIS: 퍼지 제어기의 설계 및 구현 자동화를 위한 통합 개발 환경

FADIS: An Integrated Development Environment for Automatic Design and Implementation of FLC

김대진 · 조인현*
Daijin Kim and In-Hyun Cho*

동아대학교 컴퓨터공학과
*퓨처시스템 정보통신연구소

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 확인하였으며, FLC를 FPGA상에 구현함에 따른 제어 시간의 단축을 다른 구현의 경우와 비교하였다.

ABSTRACT

This paper developes an integrated environment CAD system that can design and implement an accurate and cost-effective FLC automatically. For doing this, an integrated development environment (IDE) (called FADIS; FLC Automatic Design and Implementation Station) is built by the seemless coupling of many existing. CAD tools in an attempt to the FADIS performs various functions such that (1) automatically generate the VHDL components appropriate for the proposed FLC architecture from the various design parameters (2) simulate the generated VHDL code on the Synopsys's VHDL Simulator, (3) automatically compiler, (4) generate the optimized, placed, and routed rawbit files from the synthesized modules by Xilinx's XactStep 6.0, (5) translate the rawbit files into the downloadable execution reconfigurable FPGA board (VCC's EVC1), and (7) continuously monitor the control status graphically by communicating the FLC with the controlled target via S-bus. The developed FADIS is tested for its validity by carrying out the overall procedures of designing and implementing the FLC required for the truck-backer upper control, the reduction of control execution time due to the controller's FPGA implementation is verified by comparing with other implementations.

1. 서 론

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

이 논문은 1997년도 학술진흥재단연구비(1997-001-E00365)와 1998년도 동아대학교 특정목적과제 연구비에 의해 연구되었음.

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.

본 논문에서 사용된 고정밀 저비용 퍼지 제어기의 구조 및 특성은 다른 논문[2]에 잘 나타나 있다. 사용된 FLC의 정확성은 비퍼지화 연산시 소속값뿐 아니라 소속 함수의 폭을 고려함으로서 얻어진다. 이 경우, 부가적인 곱셈기 요구에 의한 하드웨어 복잡도 증가 문제는 곱셈기를 확률론적 AND 연산[3]에 의해 해결하였다. 사용된 FLC의 저비용성은 기존의 FLC를 다음과 같이 개조함으로서 이루어진다. 먼저, MAX-MIN 추론이 레지스터 파일의 형태로 쉽게 구현 가능한 read-modify-write 연산[4]에 의해 대치된다. 두 번째, COG 비퍼지화기에서 요구하는 제산 연산을 모멘트 균형점의 탐색에 의해 피할 수 있다. 제안한 COG 비퍼지화기는 곱셈기가 부가적으로 요구되며 모멘트 균형점의 탐색시간이 오래 걸리는 단점이 있다. 부가적 곱셈기 요구에 의한 하드웨어 복잡도 증가 문제는 곱셈기를 확률론적 AND 연산에 의해 해결할 수 있고, 오랜 탐색시간 문제는 coarse-to-fine 탐색 알고리즘[5]에 의해 크게 경감될 수 있다.

제안한 퍼지 제어기가 실질적인 제어 문제에 사용 가능함을 확인하기 위해 재구성이 가능한 FPGA 시스템 상에 직접 구현하고자 한다. 구현 과정을 간략히 설명하면 다음과 같다. 각 모듈은 VHDL언어에 의해서 기술된 뒤, Synopsys사의 FPGA 컴파일러[6]에 의해 합성된다. 합성된 각 모듈은 Xilinx사의 XactStep 6.0[7]에 의해 최적화 및 배치 배선이 이루어진다. 얻어진 Xilinx rawbit 파일은 VCC사의 r2h[8]에 의해 C 언어 프로그램의 header 파일 형태의 하드웨어 object로 변환된다. 원하는 목적(본 논문의 경우 트럭 후진 주차 제어)을 수행하기 위해 이들 하드웨어 object를 포함하는 응용 프로그램을 C 언어로 작성한 후 이를 C 컴파일러에 의해 컴파일한다. 이 실행 파일은 재구성 가능한 FPGA 시스템인 EVC1 보드[9]로 다운로드되어 원하는 목적을 수행하는지를 점검하게 된다.

원하는 FLC를 FPGA 시스템 상에 구현할 경우, FPGA가 갖는 재사용 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, 지원해 줄 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 아키텍츄어를 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 보이며, 수행 속도 및 제어 능력 개선을 보인다. 마지막으로 결론과 앞으로의 연구 방향을 언급한다.

2. 제안한 FLC의 아키텍츄어

FLC를 하드웨어적으로 구현하는 경우, 하드웨어의 복잡도를 줄이고 제어 성능을 높이기 위해 다음과 같은 제약을 가한다[10].

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

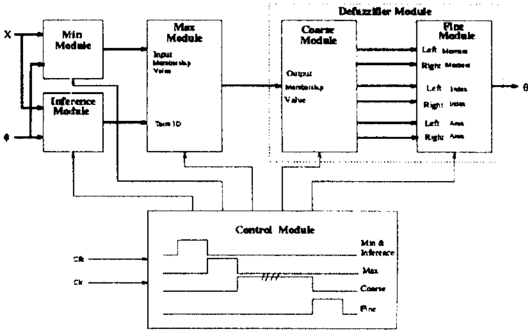


그림 1. 제안한 FLC의 하드웨어 구조.

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 1은 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.

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

3. 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.

3.1 VHDL 코드 자동 생성

FLC의 configuration 파라메터로 VHDL 컴포넌트를 자동 생성하기 위해서는 제어기가 갖는 하드웨어 구조의 형태가 일정해야 한다. 그림 2에서 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 2은 서로 다른 설계 파라메터에 대해 설계된 FLC를 보인 것이다. 그림 2-(a)은 입출력 변수의 크기 해상도가 8 비트이고, 입력 변수가 2개인 경우로 두 입력에 대해 4개의 MF lookup 테이블이 필요하고, MIN 연산의 수행 결과를 저장할 4개의 레지스터가 필요하다. 이 경우, lookup 테이블, MUX, MIN 레지스터의 해상도는 각각 8비트이다. 그림 2-(b)는 입출력 변수의 크기 해상도가 4비트이고, 입력 변수가 3개인 경우로 세입력에 대해 6개의 MF lookup 테이

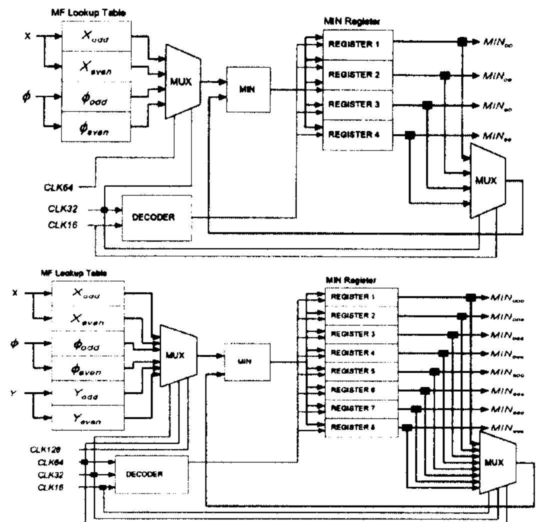


그림 2. (a) 8비트 2입력 FLC의 MIN 블록(위), (b) 4비트 3입력 FLC의 MIN 블록(아래).

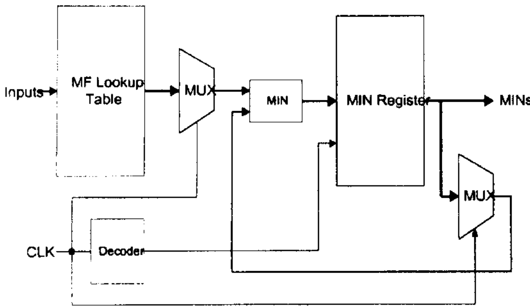


그림 3. MIN 블록의 원형판.

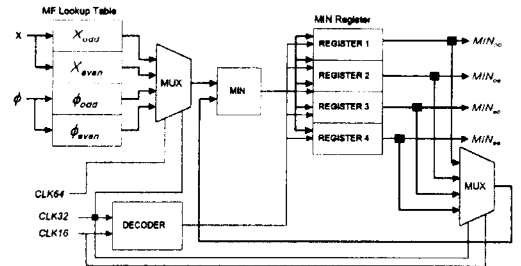
블이 필요하다. 이 경우, lookup 테이블, MUX, MIN 레지스터의 해상도는 각각 4비트이다. 두 모듈은 서로 다른 입출력 파라메터를 가짐으로 인해, MF lookup 테이블, MIN 레지스터, MUX의 갯수와 해상도만 다를 뿐, 전체적인 구조는 동일하다. 이것은, 기본적인 원형판 모듈에 입출력 파라메터에 맞도록 테이블이나 레지스터를 생성하고, 입출력 변수의 크기 해상도를 맞춤으로서 원하는 VHDL 컴포넌트의 생성이 가능하다. 그림 3은 MIN 블록의 원형판을 나타낸 것이다.

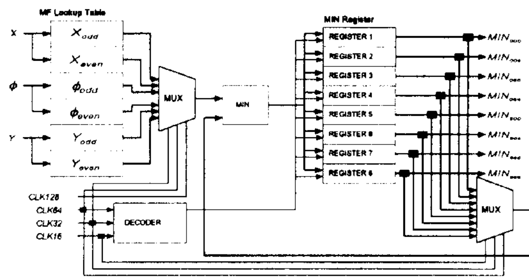
주어진 설계 파라메터로부터 VHDL 코드를 생성하는 과정은 다음과 같은 계층적 설계 기법을(Design Hierarchy) 이용한다. (1) 각 모듈의 기본 구조를 나타내는 원형판과 VHDL 코드를 준비한다. (2) 각 입출력 파라메터 값에 맞도록 레지스터나 MUX등에 대응하는 VHDL 코드를 생성하여 원형판의 VHDL 코드와 상호 연결시킨다. 여기서, 두 번째 과정의 예로서 MF Lookup 테이블과 레지스터 파일의 경우를 설명한다. 두 블록은 다른 각기 기능을 수행하지만 동일한 형태를 지니고 있다. MF lookup 테이블은 입력을 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 문중 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 변

```
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;
};
```

(a)





(b)

그림 4. (a) MUX를 위한 VHDL code 생성 함수, (b) VHDL code 생성 함수에 의해 얻어진 MUX의 VHDL 코드.

```
PACKAGE Types IS
 CONSTANT RESOLUTION : INTEGER := 8;
END Types;

ENTITY FINE IS
   PORT(LeftMoment :
   BIT_VECTOR(RESOLUTION-1 DOWNTO 0);
END FILE;
```

그림 5. Constant형을 이용한 입출력 변수의 크기 해상도 조정.

수 개수의 변수에 대한 다양한 형태의 MUX와 디코더의 VHDL 코드 생성을 가능하게 한다. 그림 4-(a)는 8비트 4입력의 MUX를 출력 파라메터에 의해 VHDL 컴포넌트를 자동 생성하기 위한 C++ 코드이며, 그림 4-(b)는 이에 의해 생성된 VHDL 컴포넌트를 보인 것이다.

제안된 FLC의 추론 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 5는 fine 모듈의 경우 레지스터 해상도를 결정하는데 출력 변수의 해상도인 8비트를 입력받고, 이를 package내에 constant 데이터형으로 정의한 다음 이를 fine 모듈의 VHDL 코드상에서 사용된 예를 보인 것이다.

3.2 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. 트럭 후진 주차 문제에서 사용되는 모델 트럭의 운동 방정식을 Synopsys사가 제공하는 기본적인 IEEE 라이브러리 이 외에도 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 testb-

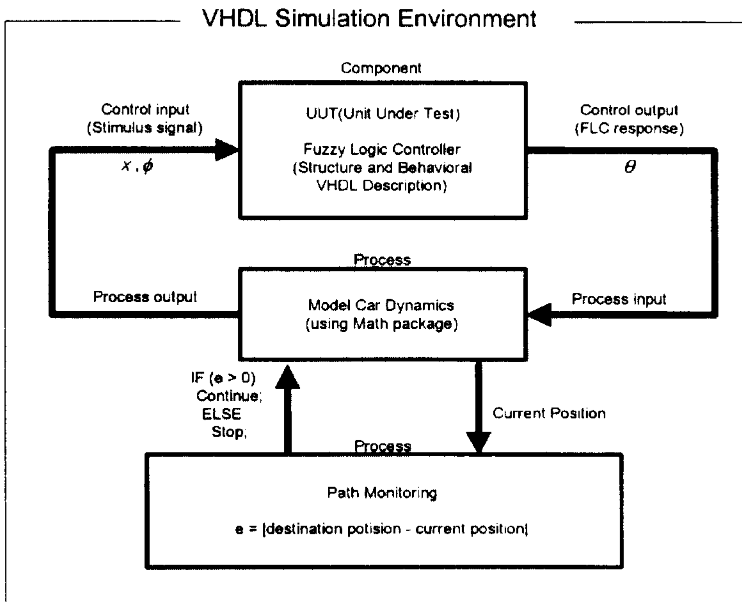


그림 6. VHDL 시뮬레이션 환경.

ed인 설계된 FLC에 stimulus 입력을 제공하는 하나의 프로세스로 볼 수 있다. 그림 6은 본 연구에서 사용한 FLC의 VHDL 시뮬레이션 환경을 나타낸 것이다[11].

3.3 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. FPGA 컴파일러를 사용하기 위해서는 먼저 사용되는 FPGA의 모델에 따라 요구되는 link 라이브러리와 target 라이브러리를 설정하는 것이 요구된다. 본 연구의 경우 재구성 가능한 FPGA 시스템 상에 사용되는 FPGA 모델명이 XC 4013PQ208[12]로 이를 위한 link 라이브러리와 target 라이브러리를 설정 예가 그림 7에 나타나 있다. 그림 7의 시작 부분에 사용하는 라이브러리가 위치하는 경로를 나타내었고, 사용하는 여러 가지 라이브러리를 종류별로 정의하였다. 두 번째 부분은 FPGA 컴파일러가 동작하는 동작 조건과 만족시켜야하는 시간 및 면적에 대한 제약 조건들이 명시되어 있다[13].

제안한 FLC의 각 모듈을 합성하기 위한 과정은 모든 모듈에 대해서 동일하므로 본 논문에서는 가장 복잡한 COG 비퍼지화기의 coarse 모듈을 중심으로 설명하고자 한다. 아래 그림 8은 Synopsys사의 FPGA 컴파일러 상에서 coarse 모듈을 합성하기 위해 사용된 명령어 파일을 나타낸 것이다. 다른 모듈들은 이 그림에서 coarse.vhd 대신 해당 모듈명.vhd로 출력 이름 coarse.sxnf 대신 해당 모듈명.sxnf로 대신하면 된다.

```
search_path={"." "/xilinx/synopsys/libraries/syn"}
link_library={xprim_4013e-3.db xprim_4000e-3.db
              xio_4000e-3.db xdc_4000e-3.db
              xgen_4000e.db}
target_library={xprim_4013e-3.db
                xprim_4000e-3.db xio_4000e-3.db
                xdc_4000e-3.db xgen_4000e.db}
synthetic_library = {standard.sldb}
symbol_library = xc4000.sdb

set_operating_conditions WCCOM
create_clock -name "sclk" -period 100
-waveform {"0" "50"} {"sclk" }
set_wire_load "4013e-3_avg"
compile_fix_multiple_port_nets = true
```

그림 7. 게이트 수준 합성을 위한 FPGA 컴파일러의 configuration 명령어.

```
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. 게이트 수준 합성을 위한 FPGA 컴파일러의 컴파일 명령어.

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 합성 시 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[14]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.

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. 아래 그림 10은 제안한 FLC의 최상위층 스키메틱 view를 나타낸 것이다.

4. 제안한 FLC 구현의 후단부(Back-End) 처리

FLC 구현의 후단부 처리 과정은 (1) 얻어진 netlist들의 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.

4.1 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 시

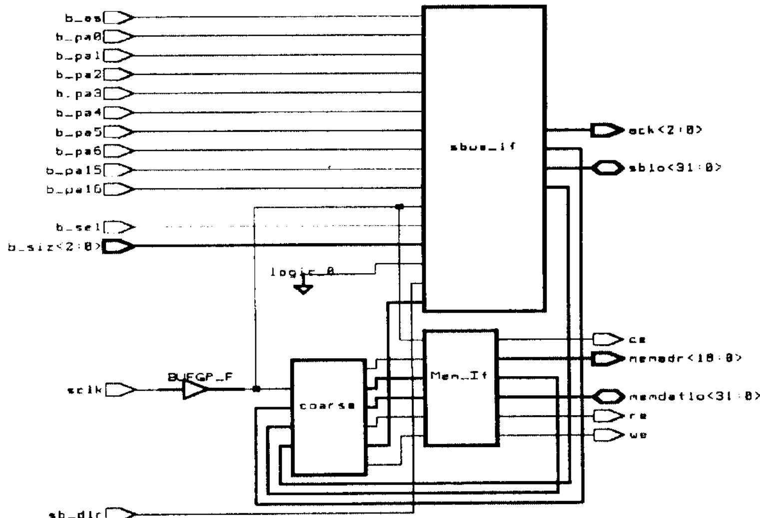


그림 9. 합성된 coarse 모듈 스키메틱 표시.

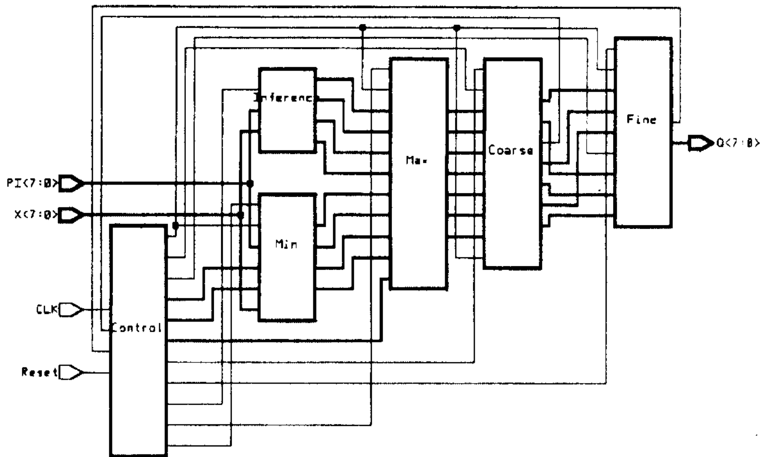


그림 10. 합성된 FLC의 최상위층 스키메틱 표시.

스템을 사용하였는데 그림 11은 이 시스템 상에서 FPGA 구현시 일어나는 과정을 나타낸 것이다.

제안한 FLC의 각 모듈을 배치 및 배선하는 과정은 모든 모듈에 대해서 동일하므로 본 논문에서는 가장 복잡한 COG 비퍼지화기의 coarse 모듈을 중심으로 설명하고자 한다. 아래 그림 12는 Xilinx사의 XactStep 6.0상에서 coarse 모듈을 최적화, 배선 및 배치를 하기 위해 사용된 명령어 파일을 나타낸 것이다. 다른 모듈들은 이 그림에서 coarse 대신 해당 모듈 명을 대신하면 된다.

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 펼쳐진(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

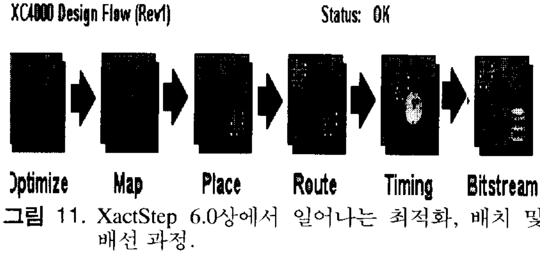


그림 11. XactStep 6.0상에서 일어나는 최적화, 배치 및 배선 과정.

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. 그림 13은 제안한 COG 비퍼지화기내 fine 모듈에 대해 위의 과정을 거쳐 얻어진 논리 셀 어레이 파일(fine.lca)의 내부 스키메틱을 나타낸 것이다. 현재 사용중인 PC가 갖는 메모리 용량이 32M인 관계로 내부 구조가 매우 복잡한 coarse 모듈 내용 전체를 화면상에 보여주지 못한 관계로 편의상 본 논문에서는 fine 모듈의 내부 레이아웃을 보여준다.

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 14는 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.

4.2 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[15]. 이 시스템의 가장 중요한 구성 요소는 SRAM 기반 FPGA로서 기능이 미지정된(uncommitted) 많은 프로그래머블 논리 게이트와 프로그래머블 연결선으로

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

그림 12. 최적화, 배치 및 배선을 위한 XactStep 6.0의 명령어 프로시쥬어.

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. 그림 15-(a)는 사용된 EVC1보드의 외형을 나타낸 것으로 아래쪽의 조그만 IC가 프로그래머블 클럭 발생기이고 가운데 IC가 13,000게이트까지 구현 가능한 Xilinx FPGA XC4013PQ208이며, 위쪽에 있는 PLA는 EVC1보드와 workstation간의 S버스를 통한 상호 통신을 제어하는 목적으로 사용된다

알고리즘을 재구성 가능한 FPGA 시스템 상에서 hardwiring에 의해 구현하는 방법에는 크게 두 가지가 있다[16]. 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

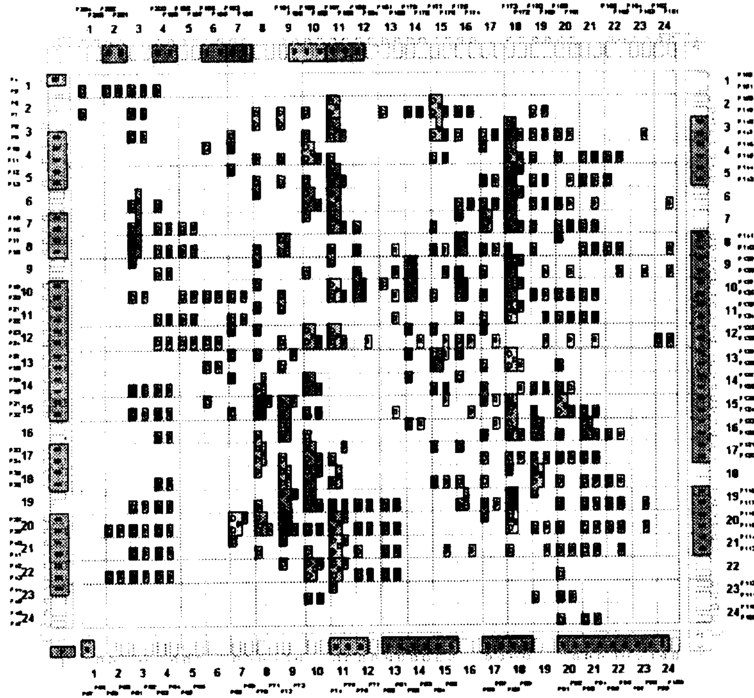


그림 13. Fine 모듈의 FPGA 내부 레이아웃.

	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 14. 각 모듈의 FPGA 요소 사용율.

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 16은 앞에서 설명한 여러 가지 재구성 방법을 예시한 것이다. EVC1보드내 장착된 Xilinx FPGA XC4013PQ208은 국부적 변경이 불가능하므로 본 연구에서는 전체적 run-time 재구성법에 의해 알고리즘을 구현하였다. 현재 출시되고 있는 XC 6020은 run-time시 국부적 재구성이 가능하므로 재구성 시간이 크게 단축되고 응용분야가 더욱 넓어질 것으로 전망된다. 나아가, 실행 시점 재구성에 의한 알

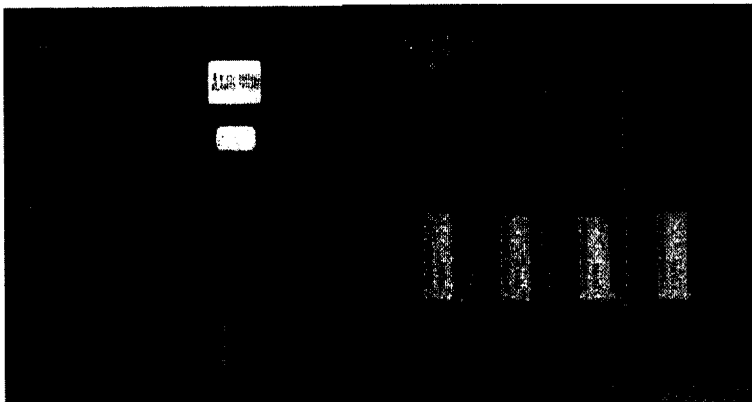


그림 15. (a) EVC1보드 외양, (b) 2M SRAM보드 외양.

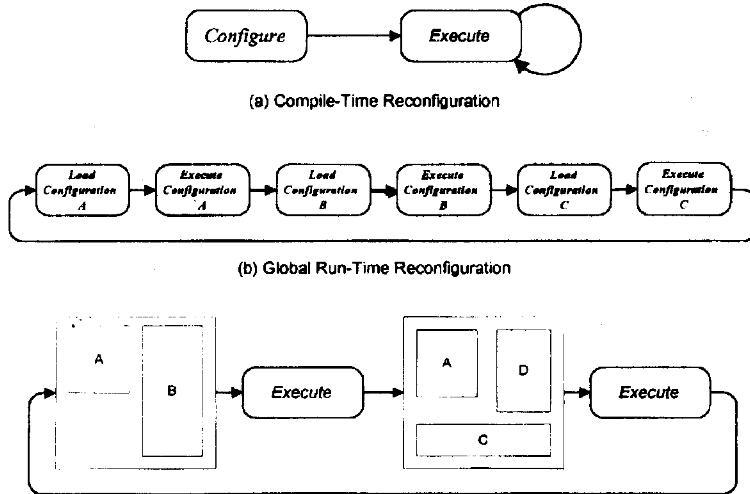


그림 16. 여러 가지 재구성 방법.

고리즘 구현시에는 앞선 configuration의 중간 결과를 뒤따르는 configuration이 입력 데이터로 사용하는 경우가 흔하므로 이를 위해 configuration간의 데이터를 주고 받을 수 있도록 메모리 보드가 요구된다. 이를 위해 본 연구에서는 그림 15-(b)에 보인 2M SRAM 보드를 사용하였다.

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)을 사용하였다. 이 기술에 의하면 EVC1에 원하는 알고리즘을 구현하는 과정은 다음과 같다[16].

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.

그림 17은 제안한 FLC를 EVC1 보드 상에 하드웨어 object 기술을 사용하여 구현하는 경우의 C 응용 프로그램의 일부를 보인 것이다. 프로그램의 while 루프내 WriteMemory 함수가 현재 위치의 (x, ϕ)값을 메모리에 쓰면 이 값을 이용하여 FLC내 각 모듈이 순차적으로 연산을 하여 θ값을 결정한 후 이를 메모리에 저장하면 ReadMemory 함수에 의해 이 값을 읽어와서 모형 트럭으로 보낸다.

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 주고 받기 위한 S버스 인터페이스와 EVC1 보드내의 FPGA 상에 다운로드된 하드웨어 object와 2M SRAM 보드상에 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 18은 EVC1 보드네 FPGA상에 구현된 두 개의 인터페이스와 주변 Host 컴퓨터와 메모리 보드사이의 연결을 나타낸 것이다.

VCC사에 의해 제공된 위의 두 인터페이스는 Viewlogic 시스템에서 만든 스키메틱 마크로 상태여서 Viewlogic 설계 장비가 없는 관계로 이를 그대로 사용할 수가 없어서 본 연구에서는 앞서의 FLC내 여러 모듈을 얻는 과정과 마찬가지로 이들 interface 로직을 VHDL을 사용하여 기술한 후 이를 Synopsys에 의해 합성하여 사용하였다.

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 단

```
#include <math.h>
#include <sys/times.h>
#include <unistd.h>
#include "evc.h"
#include "memory_module.h"
#include "min_module.h"
#include "inference_module.h"
#include "max_module.h"
#include "coarse_module.h"
#include "fine_module.h"
int      *ports;
main(int argc,char *argv[])
{
        int      x,y,phi,theta;
        ports = EVCdownload(memory_module);
                        /* Memory Initialization */
        InitMemory();
        EVCreset(1);
        while(!goal_position(x,y,phi)){
           ports = EVCdownload(memory_module);
                        /* set current position */
           WriteMemory(INPUT, (phi << 8) | x);
              EVCreset(1);
              ports = EVCdownload(min_module);
                        /* configure min module */
              EVCreset(1);
              ports = EVCdownload(inference_module);
                        /* configure inference module */
              EVCreset(1);
              ports = EVCdownload(max_module);
                        /* configure max module */
              EVCreset(1);
              ports = EVCdownload(coarse_module);
                        /* configure coarse module */
              EVCreset(1);
              ports = EVCdownload(fine_module);
                        /* configure fine module */
              EVCreset(1);
              ports = EVCdownload(memory_module);
                        /* get control data */
              ReadMemory(41,&theta);
              EVCreset(1);
             Car(x,y,phi,theta,&x,&y,&phi);  /* run car */
        }
}
void      ReadMemory(int address,int *x)
{         ports[ADDRESS_PORT] = address;
          ports[READ_PORT] = 1;
          *x = ports[READ_DATA_PORT];
}
void      WriteMemory(int address,int x)
{         ports[ADDRESS_PORT] = address;
          ports[DATA_PORT] = x;
```

그림 17. 제안한 FLC 구현을 위한 C 응용 프로그램.

방향 포트 2개로 이루어진다. 메모리 모듈은 read, write, CS 신호에 의해 제어되고, 메모리 모듈과 연결된 양방향 포트는 read, write 신호에 의해 제어된다. 모든 신호는 low 상태에서 동작하므로, read 신호가 low 상태이고, write 신호가 high 상태이면 OE가 high 상태가 되고, 메모리 모듈로부터 데이터를 읽어 들이는 상태가 되므로, 양방향 포트는 메모리 모듈로부터 데이터를 읽어 들이는 기능만을 수행한다. 반대로, 경우도 이와 동일하게 메모리 모듈로 쓰고자 하는 데이터를 출력하는 기능을 수행한다.

5. 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 시스템상에 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, 각종 파일을 효과적으로 관리할 수 있으며, 얻어진 라이브러리들의 재사용이 편리하도록 여러 가지 프로그램과 CAD장비를 하나의 개발 환경 하에서 통합하는 것이 요구된다. 이러한 요구를 만족시키고자 본 연구에서는 VHDL 코드 자동 생성, 시뮬레이션, 합성, FPGA 구현, 제어 동작 모니터링 기능을 하나의 환경 하에 통합한 FADIS(FLC Automatic Design and Implementation Station)을 개발하였다. 그림 19는 FADIS 개발과정에서 고려할 자동 설계 및 구현 과정의 구성도를 나타낸 것이다. FADIS는 그래픽 사용자 인터페이스(GUI: Graphic User Interface)를 제공하기 위하여 X Window 시스템[17]상에서 개발되었다. 따라서, 자동 설계 및 구현에 필요한 모든 설계 파라메터들은 GUI를 통하여 직접 입력되어 설정되어진다.

그림 20은 구현된 FADIS의 구성 요소를 나타낸 것이다. FADIS는 전체 흐름을 제어하고, 퍼지 제어기의 입출력 변수들의 소속함수 내용을 보여주는 Main Window, 새로운 제어기 설계를 위한 입출력 파라메

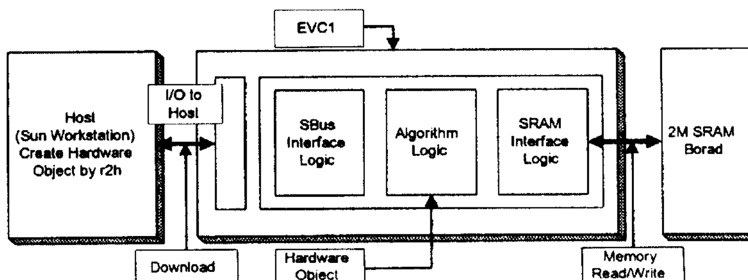


그림 18. 두 인터페이스와 주변 시스템사이의 관계.

터를 설정하는 Project Dialog, 입출력 변수의 속성(퍼지항의 갯수, 변수 해상도, 범위, 등)을 정의하는 I/O Configuration Dialog, 각 퍼지항들의 속성(퍼지항의 중심, 폭, 형태, 등)을 정의하는 Fuzzy Term Configuration Dialog, 퍼지 제어기의 제어 규칙을 정의하는 Rulebase Configuration Dialog, 마지막으로 구현된 퍼지 제어기의 동작을 모니터링하는 Trajectory Monitoring Dialog로 구성되어 있다. 주요 구성 요소들을 설명하면 다음과 같다.

5.1 Main Window

FADIS은 1) configuration dialog에서 입력받은 설계하고자 하는 퍼지 제어기의 여러 설계 파라메터를 사용하여 제안한 새로운 형태의 퍼지 제어기 아키텍츄어에 맞는 VHDL 컴포넌트를 자동적으로 생성시키며, (2) 얻어진 각 컴포넌트에 대응하는 netlist를 Synopsys사의 FPGA 컴파일러에 의해 자동적으로 합성시키고, (3) 합성된 각 모듈의 netlist들을 Xilinx사의 XactStep 6.0상에서 최적화 및 배치, 배선 과정을 수행시켜 Xilinx rawbit 파일을 만들어내며, (4) 이들 모듈의 rawbit 파일을 VCC사의 하드웨어 object 기술에 의해 실행 가능한 파일로 변환시키며, (5) C 언어로 프로그래밍한 원하는 응용 제어 프로그램에 의해 이들 실행 가능한 파일을 재구성 가능한 FPGA 시스템(EVC1 보드)상에 다운로드시키며, (6) FADIS 툴과 FPGA 시스템 사이에 S버스를 통해 상호 정보를 교환하여, FPGA 시스템으로부터 읽혀진(readback) FLC의 제어 상태 정보는 그래픽 환경을 통해 연속적으로 모니터링하는 전 과정을 수행한다.

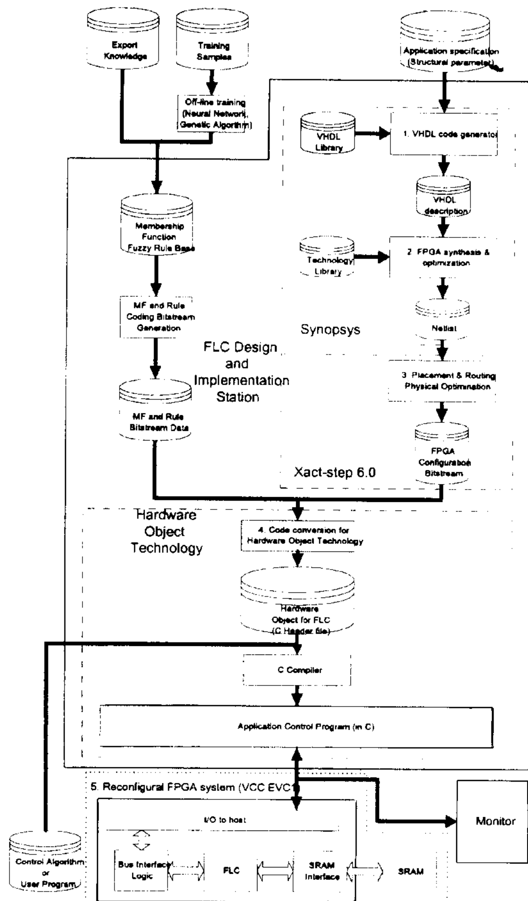


그림 19. FADIS의 자동 설계 및 구현 흐름도.

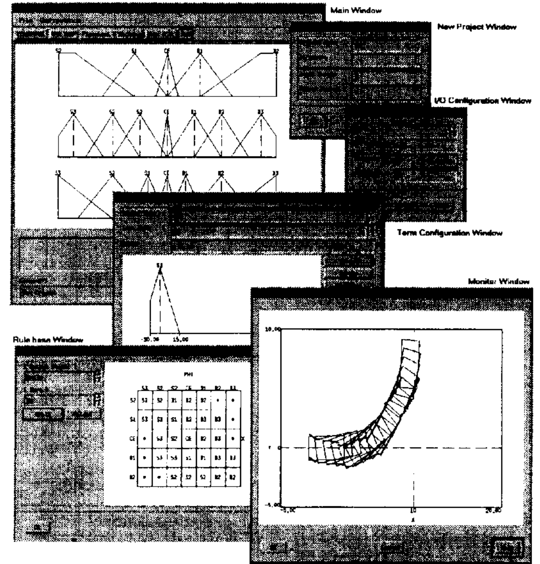


그림 20. FADIS의 구성 요소.

5.2 Configuration Dialog

FADIS는 설계하고자 하는 퍼지 제어기의 여러 파라메터를 입력 받기 위한 3종류의 configuration dialog로 구성되어 있다. 설계 파라메터는 크게 2종류로 나눠어지는데 퍼지 제어기의 입출력에 대한 파라메터와 퍼지 제어기의 추론에 사용되는 제어 규칙이다. 입출력 변수에 대한 파라메터로는 입출력 변수가 가지고 있는 퍼지항의 갯수, 입출력 변수의 범위, 입출력 변수의 해상도가 있다. I/O configuration dialog는 입출력 변수에 대한 파라메터를 입력받아 설계하고자 하는 퍼지 제어기의 입출력에 대한 기본적인 구성을 한다. 각 입출력 변수들은 여러 개의 퍼지항으로 구성되어 있다. 초기 퍼지항들은 균일 간격으로 구성되며, Fuzzy term configuration dialog는 퍼지항들을

원하는 목적에 맞도록 퍼지항의 폭, 중심 및 형태를 설정하기 위한 환경을 제공해 준다. Rulebase configuration dialog는 그림 20의 하단에 보이는 것으로서 제어 규칙을 나타내는 일반적인 형태인 테이블 형태로 받아들인다. 각각의 테이블에 원하는 목적에 맞도록 제어 규칙을 삽입 또는 수정하면 된다.

5.3 모니터링 Dialog

FADIS 툴과 FPGA 시스템 사이에 S버스를 통해 상호 정보를 교환한다. FPGA 시스템으로부터 읽혀진 FLC의 제어 상태 정보는 그래픽 환경을 통해 연속적으로 모니터링된다. 이러한 모니터링은 실시간으로 처리되므로 정확한 제어 상태를 알 수 있다.

6. 구현결과 및 분석

구현한 FADIS가 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 21은 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 축간의 각도인 θ에 의해 결정된다. 트럭이 움직이는 운동 방정식은 아래와 같이 나타내어진다[18].

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

여기서 b는 트럭의 길이이며, 본 논문에서는 b=4로 하였다.

만약 트럭과 주차대까지의 거리가 충분하다면 트럭이 x=10, ϕ=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, ϕ=90°)쪽으로 후진하도록 바퀴 각도 θ($-40 \le \theta \le 40$)를 제어하는 것이 요구된다. 그림 22는 Wang과 Mendel[19]이 사용한 퍼지 제어기의 입 · 출력 변수의 소속함수와 퍼지 제어를 위한 퍼지 규칙 베이스를 나타낸 것이다.

제안한 FLC를 재구성 가능한 FPGA 시스템 상에

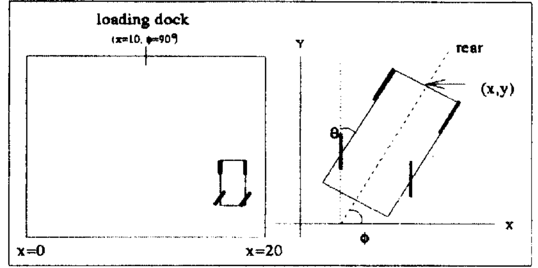


그림 21. 모형 트럭과 주차대 위치.

구현한 경우 이들이 시뮬레이션에서와 똑같이 제대로 동작하는지를 알아보기 위해 임의의 두점(왼쪽 그림 시작점: (0.0, 0.0, 0°), 가운데 시작점(13.5, 0.0, 241°))에서의 목적지까지 어떻게 주행하는가를 조사하였다. 그림 23으로부터 두 경우 모두 목적지에 정확히 도달하는 것을 알 수 있다. 이로 미루어보아 재구성 가능한 FPGA 시스템 상에 구현한 FLC는 정확히 원하는 데로 제어 동작을 수행하고 있음을 확인할 수 있다.

제안한 FLC를 트럭 후진 주차 문제에 적용하여 재구성 가능한 FPGA 시스템의 일종인 EVC1 보드상에 구현한 경우와 Synopsys사의 VHDL 시뮬레이터 상에 구현한 경우, 워크스테이션 상에 C언어에 의해 구현한 경우 각각의 제어 수행시 연산 속도를 서로 비교하기 위해 목적지 도달에 관계없이 임의로 선택한 1,000개의(x, ϕ)점에 대한 퍼지 제어기의 출력 θ가 얻어질 때까지 걸리는 시간을 평균하여 얻은 평균 퍼지 연산시간을 비교하였다(실제 Synopsys사의 VHDL 시뮬레이터 상에서 퍼지 연산을 수행하는데는 너무 많은 연산시간이 걸려 10개의 10스텝에 대해서만 수행하여 평균 시간을 얻었다). 이 실험에서 세 가지 경우에 대한 수행시간 비교시 실험 조건을 될 수 있는 데로 공평하게 비교하기 위해 Synopsys VHDL 시뮬레이션의 경우 그래픽 환경에서 수행하지 않고 Shell

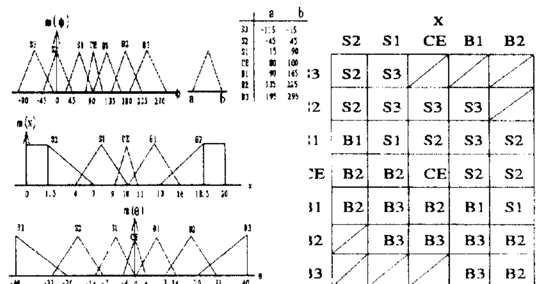


그림 22. 트럭 후진 주차 제어에 사용된 소속 함수와 퍼지 규칙 베이스.

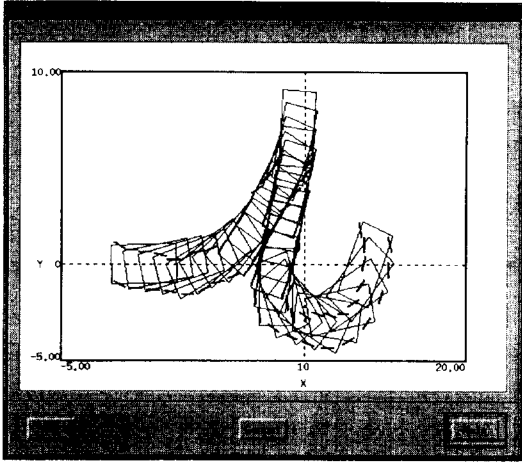


그림 23. 두 점으로부터의 모형트럭의 주행곡선.

표 1. 세 가지 경우 평균 퍼지 연산시간

	Synopsys VHDL simulation	C language simulation	FPGA implementation
퍼지연산 수행 수	10	1000	1000
전체 걸린시간	320.4	88.36	5.14
평균 퍼지 연산시간	32.04	0.08830	0.00514

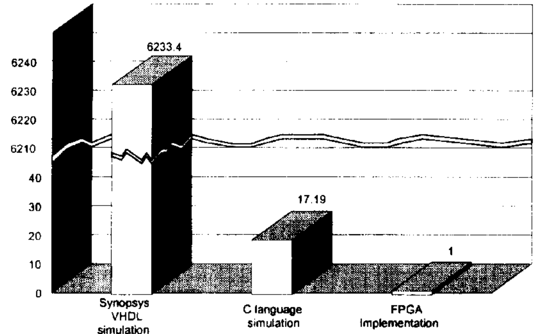


그림 24. 세 가지 경우 속도 향상비 비교.

환경에서 Vhdlsim을 이용하였다. 표 1은 위의 세 가지 경우의 퍼지 연산에 거리는 시간을 계산하는 과정을 나타낸 것이다.

그림 24는 FPGA 구현시 걸리는 시간을 기준으로 하였을 때 다른 두 가지 시뮬레이션에 비해 얻어진 속도 향상비를 나타낸 것이다. 이 그림으로부터 제안한 FLC를 FPGA에 직접 구현하는 경우가 C 언어에 의한 시뮬레이션보다는 O(10)배, Synopsys VHDL 시뮬레이션보다는 O(10^3)배의 수행 속도 개선 효과를 얻을 수 있었다.

7. 결 론

본 논문은 시스템의 복잡도를 줄여 구현 비용이 적게 들며, 기존의 COG 비퍼지화기보다 정확한 제어를 수행하는 새로운 COG 비퍼지화기를 갖는 퍼지 제어기를 제안하였다. 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 도달하는데 걸리는 평균 주행 거리를 24.5% 이상 줄이는 성능 개선 효과를 얻을 수 있었다.

제안한 퍼지 제어기가 실질적인 제어 문제에 사용 가능함을 확인하기 위해 재구성이 가능한 FPGA 시스템 상에 직접 구현하였다. 제안한 FLC의 재구성 가능한 FPGA 시스템상의 구현 과정은 다음과 같다. 각 모듈은 VHDL 언어에 의해서 기술한 후, Synopsys사의 FPGA 컴파일러에 의해 합성하였다. 합성된 각 모듈은 Xilinx사의 XactStep 6.0에 의해 최적화 및 배치 배선의 back-end 처리를 하였다. 얻어진 Xilinx rawbit 파일은 VCC사의 r2h에 의해 C 언어 프로그램의 header 파일 형태의 하드웨어 object로 변환시킨 후, 이들 하드웨어 object를 포함하는 FLC 제어용 C 응용 프로그램을 작성한 후 이를 C 컴파일러에 의해 컴파일하였다. 이 실행 파일을 재구성 가능한 FPGA 시스템인 EVC1 보드에 다운로드하여 원하는 제어 목적을 수행하는지 확인하였다.

제안한 퍼지 제어기를 자동으로 설계 및 구현하기 위해 설계 파라메터로부터 VHDL 코드 자동생성, 시뮬레이션, 합성, 배치 및 배선, 다운로드 및 모니터링 등의 기능을 하나의 환경하에 통합된 통합 개발 환경 FADIS를 구현하였다. FADIS는 X Window 시스템의 GUI 환경하에서 구현되었으며 설계 파라메터 입력에서부터 여러 단계를 거쳐 최종의 동작 모니터링까지의 기능을 연속적으로 수행 가능하게 한다.

재구성 가능한 FPGA 시스템 상에 구현한 FLC를 트럭 후진 주차 문제에 적용하여 그 연산 속도를 Synopsys사의 VHDL 시뮬레이터 및 워크스테이션 상에 C 언어로 구현한 경우의 연산 속도와 비교하였는데, FPGA상에 구현한 경우가 VHDL 시뮬레이터 보다는 O(10^3)배, C언어에 의한 시뮬레이션보다는 O(10)배 정도 빠름을 확인하였다. 아울러, 재구성 가능한 FPGA 시스템 상에 구현한 FLC가 제대로 동작하는지를 확인하고자 임의의 세점에 대한 모형 트럭

의 주행 경로를 추적하여 보았는데 세 경우 모두 목적지에 제대로 도달하는 것을 확인할 수 있었다.

현재는 구현한 FLC를 구성하는 5가지 모듈을 독립적으로 하나씩 구현하여 매 퍼지 연산시 5번의 다운로드가 필요하다. 그러나, MIN 모듈과 MAX 모듈의 레지스터 파일을 서로 공유하고 Inference 모듈의 구조를 최적화시키면, MIN, Inference 및 MAX 세 모듈을 하나의 XC4013에 집어 넣을 수 있다. 이 경우, 3번의 다운로드로 퍼지 연산이 가능하므로 연산시간을 더욱 단축할 수 있을 것으로 판단되어 MIN, Inference, MAX 모듈 통합 작업을 수행할 계획이다. 나아가, 퍼지항의 소속 함수와 규칙 베이스가 미리 알려져 있지 않는 경우 학습 샘플로부터 이들을 유전학적 진화에 의해 결정할 수 있는데[20], 이들 유전 알고리즘 부분을 FPGA상에 하드웨어로 직접 구현하여, 유전 학습 기능을 포함하는 지능적 퍼지 제어기를 구현할 예정이다.

감사의 글

본 연구를 수행하는데 사용된 Synopsys FPGA 컴파일러 및 Xilinx사의 XactStep 6.0은 KAIST 부설 반도체설계교육센터(IDEC)의 CAD 장비 지원에 의해 이루어졌으며 본 연구자들은 이러한 지원에 큰 감사의 뜻을 밝힙니다.

참고문헌

[1] E. H. Mandani, "Application of fuzzy algorithms for control of simple dynamic plant", *IEEE Proc. Control & Science*, **121**(12), 1585-1588, Dec. 1974.

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

[3] Y. Kondo and Y. Sawada, "Functional abilities of a stochastic logic neural network", *IEEE Transactions on Nerual Networks*, **3**(3), 434-443, May 1992.

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

[5] Daijin Kim and In-Hyun Cho, "A Dividerless COG Defuzzifier with an Efficient Searching of Moment Equilbrium Point", Seventh IFSA World Congress, Accepted for presentation, 1997.

[6] SYNOPSYS, "Design compiler reference manual v3.4", Synopsys Corp., 1996.

[7] Xilinx, "XactStep development system user guide", Xilinx., 1996.

[8] VCC, "EVC1-virtual computer programming tutorial", VCC Corp., 1994.

[9] VCC, "EVC1 technical reference", VCC Corp., 1995.

[10] A. Costa, A. D. Gloria, P. Faraboschi, A. Pagni and G. Rizzotto, "Hardware solutions for fuzzy control", *Proceedings of the IEEE*, **83**(3), 422-434, March 1995.

[11] S. Mazor and P. Langstaraat, "A guide to VHDL", Synopsys Corp., 1995.

[12] Xilinx, "The programmable logic data book", Xilinx., 1996.

[13] SYNOPSYS, "VSS family tutorial v3.4", Synopsys Corp., 1996.

[14] Xilinx, "Xilinx synopsys interface FPGA user guide", Xilinx., 1994.

[15] B. Hutchings and M. Wirhkin, "Implementation approaches for reconfigurable logic applications", *Field Programmable Logic and Applications*, pp. 419-428, 1995.

[16] S. Casselman, M. Thornburg and J. Schewel, "H.O.T (hardware object technology) programming tutorial", VCC Corp., 1995.

[17] D. A. Young, "The X window system programming and applications with Xt", Prentice Hall Inc., 1994.

[18] Li-Xin Wang and Jerry M. Mendel, "Generating fuzzy rules from numerical data with applications", USC-SIPI Report, No. 169, 1991.

[19] Li-Xin Wang and Jerry M. Mendel, "Generating fuzzy rules by learning from examples", *IEEE Transactions on System, Man and Cybernetics*, **22**(6), 1414-1427, Nov. 1992.

[20] Daijin Kim, "Improving the fuzzy system performance by fuzzy system ensemble", Fuzzy Sets and Systems, **98**(1), 43-56, 1998.

김 대 진(Daijin Kim) 정회원
제 6 권 제 4 호 참조

조 인 현(In-Hyun Cho) 정회원
제 6 권 제 4 호 참조