고정밀 저비용 퍼지 제어기의 재구성 가능한 FPGA 시스템 상에 구현
An Implementation on the Reconfigurable FPGA System of Accurate and Cost-effective Fuzzy Logic Controller

조 인 현, 김 대 진
(Inhyun Cho, Daijin Kim)

요 약

본 논문은 저비용이면서 정확한 제어를 수행하는 새로운 퍼지 제어기의 재구성 가능한 FPGA 시스템상의 구현을 다룬다. 제안한 퍼지 제어기 (Fuzzy Logic Controller: FLC)의 시스템 구조와 이의 VHDL 설계 및 시뮬레이션은 다른 논문[1]에 나타나 있다. 제안한 퍼지 제어기의 구현 과정은 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. C 언어 형태의 하드웨어 object를 포함하는 응용 제어 프로그램이 C 컴파일러에 의해 컴파일된 후, 이 실행 파일이 재구성 가능한 FPGA 시스템 상에 다운로드된다. 제안한 퍼지 제어기를 EVC1 보드 상에 동적으로 구현하여 트럭 후진 주차 제어에 사용할 때 걸리는 시간을 Synopsys사의 VHDL 시뮬레이터와 워크스테이션상에서 C언어에 의해 구현하여 트럭 후진 주차 제어에 사용할 때 걸리는 시간을 각각 비교하였다.

I. 서 론

제안한 FLC의 시스템 구성 및 아키텍쳐어는 앞서의 논문[1]에 잘 기술되어 있다. 제안한 FLC의 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. 제안한 퍼지 제어기 저비용성은 기존의 FLC를 다음과 같이 개조함으로서 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 퍼지 제어기의 각 모듈은 VHDL에 의해 기술되고, 이들이 제대로 동작하는지 여부를 SYNOPSYS사의 VHDL 시뮬레이터[2]상에서 트럭 후진 주차 문제에 적용하여 검증하였다. 트럭 후진 주차 문제에 적용한 결과, 목적지 주차대에 도달하는 데 걸리는 평균 주행 거리를 24.5% 이상 줄이는 성능 개선 효과를 얻을 수 있었다. 아래 그림 1은 제안한 FLC를 트럭 후진 주차 문제에 적용한 VHDL 시뮬레이션 환경을 도식적으로 나타낸 것이다.

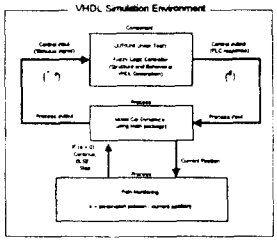


그림 1. FLC의 VHDL 시뮬레이션 환경

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 VHDL 언어에 의해서 기술된 뒤, Synopsys사의 FPGA 컴파일러[4]에 의해 합성된다. 합성된 각 모듈은 Xilinx사의 XactStep 6.0[5]에 의해 최적화 및 배치 배선이 이루어진다. 얻어진 Xilinx rawbit 파일은 VCC사의 r2h[6]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[7]로 다운로드되어 원하는 목적을 수행하는지를 점검하게 된다. 그림 2는 본 연구에서 수행된 FLC의 설계 과정의 흐름도를 나타낸 것이다.

본 논문의 구성은 다음과 같다. II장에서는 제안한 FLC를 VHDL 언어에 의해 기술한 뒤, Synopsys사의 FPGA 컴파일러에 의해 Xilinx사의 FPGA 기술적 라이브러리를 사용하여 게이트 레벨의 netlist를 얻는 과정을 설명한다. III장에서는 Xilinx사의 XactStep 6.0에 의해 최적화 및 배치, 배선되는 과정을 설명한다. VI장에서는 재구성 가능한 FPGA 시스템인 VCC사의 EVC1 보드와 하드웨어 object 기술에 대해 설명한다. V장에서는 EVC1 보드 상에 구현된 FLC, VHDL 시뮬레이터 상에 구현된 FLC, 워크스테이션상에 C언어에 의해 구현된 FLC간의 제어 수행시 연산 속도를 서로 비교한다. 마지막으로, 결론과 앞으로의 연구 방향을 언급한다.

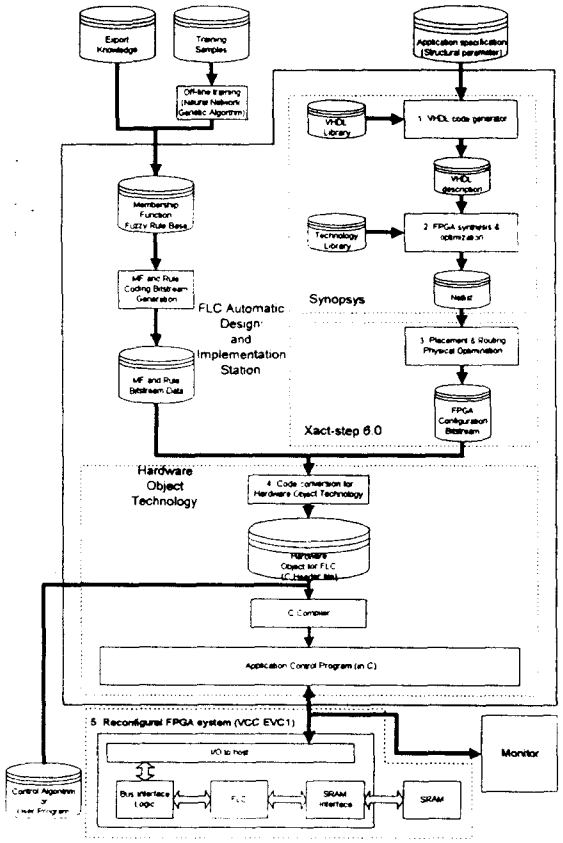


그림 2. 제안한 FLC의 설계 흐름도

Ⅱ. 제안한 FLC의 FPGA 합성

FLC를 하드웨어적으로 구현하는 경우, 하드웨어의 복잡도를 줄이고 제어 성능을 높이기 위해 다음과 같은 제약을 가한다[8].

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

아래 그림 3은 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[1]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)
- 입출력 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 균형점 탐색에 의한 COG 비퍼지화[1]

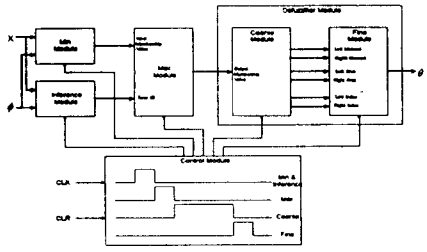


그림 3. 제안한 FLC의 하드웨어 구조

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 모델명이XC4013PQ208[9]로 이를 위한 link 라이브러리와 target 라이브러리를 설정예가 그림 4에 나타나 있다. 그림 4의 시작 부분에 사용하는 라이브러리가 위치하는 경로를 나타내었고, 사용하는 여러 가지 라이브러리를 정의하였다. 두 번째 부분은 FPGA 컴파일러가 동작하는 동작 조건과 만족시켜야하는 시간 및 면적에 대한 제약 조건들이 명시되어 있다[4].

제안한 FLC의 각 모듈을 합성하기 위한 과정은 모든 모듈에 대해서 동일하므로 본 논문에서는 가장 복잡한 COG 비퍼지화기의 coarse 모듈을 중심으로 설명하고자 한다. 아래 그림 5는 Synopsys사의 FPGA 컴파일러상에서 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=("." "/usr3/xilinx/synopsys/libraries/syn")
link_library = {xprim_4013e-3.db xprim_4000e-3.db
                xio_4000e-3.db xdc_4000e-3.db}
target_library = {xprim_4013e-3.db xprim_4000e-3.db
                xio_4000e-3.db  xdc_4000e-3.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 4. 게이트 수준 합성을 위한 FPGA 컴파일러의 configuration 명령어

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

그림 5. 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 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[10]은 컴파일 결과 얻어진 파일내 존재하는 CLB셀이나 IOB셀 등을 Xilinx FPGA 칩내에서 실제 사용가능한 기본 셀(and 또는 or등 게이트 수준)로 바꾸어 주는 역할을 한다. 아래 그림 6은 Synopsys상의 FPGA 컴파일러에 의해 위의 명령어 파일에 의해 합성된 S버스 인터페이스, 메모리 인터페이스 및 COG 비퍼지화기의 coarse 모듈을 포함하는 스키메틱 view를 나타낸 것이다.

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

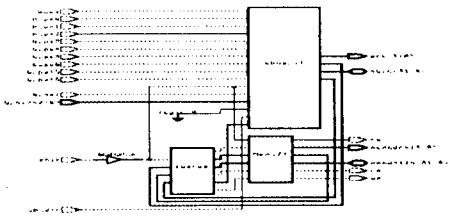


그림 6. 합성된 coarse 모듈의 전체의 스키메틱 표시

Ⅲ. 제안한 FLC의 배선 및 배치

제안한 FLC를 재구성 가능한 FPGA 시스템 상에 구현하기 위한 두 번째 단계는 앞단계에서 얻어진 netlist 파일(*.sxnf 파일)을 Xilinx FPGA상에 적합하도록 배치 및 배선을 하여 논리셀 어레이 파일(*.lca 파일)을 얻는 과정이다. 위를 위해 본 연구에서는 Xilinx사의 XactStep 6.0 Xilinx FPGA 개발 시스템을 사용하였는데 그림 7은 이 시스템 상에서 일어나는 과정을 나

타낸 것이다.

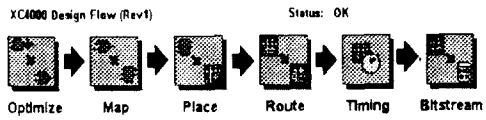


그림 7. XactStep 6.0상에서 일어난 후단부 처리 과정

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 8은 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.

```
# 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
                flc.xnf flc.xff
# DRC & optimization
flc.xtf : flc.xff
      xnfprep flc.xff flc.xtf parttype=4013PQ208
# placement & routing
flc.lca : flc.cst flc.xtf
        ppr flc.xtf parttype=4013PQ208
        xdelay -D -W flc.lca
# make raw bit file
flc.rbt : flc.lca
        makebits -b flc.lca
```

그림 8. 후단부 처리를 위한 XactStep6.0의 명령어 프로시쥬어

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. 그림 9는 제안한 COG 비퍼지화기내 fine 모듈에 대해 위의 과정을 거쳐 얻어진 논리 셀 어레이 파일(fine.lca)의 내부 스키메틱을 나타낸 것이다.

FPGA가 갖는 CLB 및 IOB 개수가 유한하므로 제안한 FLC를 하나의 FPGA에 구현하는 것은 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 10는 각 모듈의 배치 및 배선이 끝난 결과 사용한 CLB, flipflop, 및 I/O 핀의 백분율을 나타낸 것이다. 이 그림으로부터 하나의 FPGA에 하나의 모듈을 대응시키는 분할법이 아주 효과적이지는 않지만 각 모듈내의 [illegible] 부분이 서로 동일하므로 실행 시간에 부분적으로 재구성이 가능한 FPGA 시스템을 사용하는 경우 재구성 시간을 줄일 수 있는 장점이 있다.

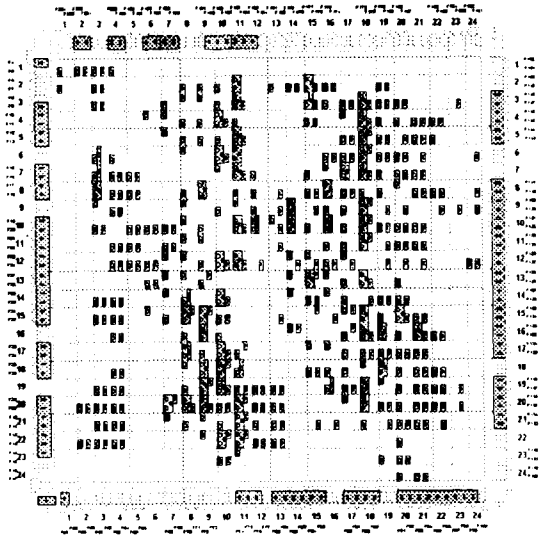


그림 9. Fine 모듈의 FPGA 내부 레이아웃

	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 10. 각 모듈의 FPGA 요소 사용율

Ⅳ. 재구성 가능한 FPGA 시스템 상에 구현

재구성 가능한(reconfigurable) 컴퓨팅 시스템은 FPGA가 가지는 재구성 능력을 이용하여 구현하고자 하는 응용 알고리즘 내 연산시간이 많이 걸리는 부분을 hardwiring에 의해 구현함으로서 알고리즘 수행시간을 크게 줄여줄 수 있는 능력을 나타낸다[11]. 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[12]. 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 11는 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 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.

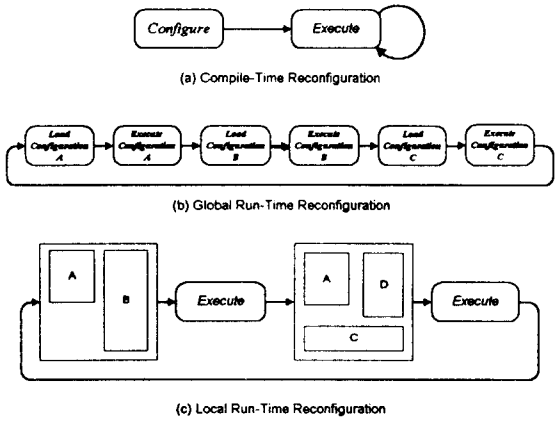


그림 11. 여러 가지 재구성 방법

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)을 사용하였다. 이 기술에 의하면 EVC1에 원하는 알고리즘을 구현하는 과정은 다음과 같다[13].

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 12은 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.

```
#include "evc.h"
#include "module.h"
int *ports;
main(int argc,char *argv[])
{
    int x,y,phi,theta;
    ports = EVCdownload(memory_module);
    InitMemory();/* Memory Initialization */
    EVCreset(1);
    while(!goal_position(x,y,phi)){
        /* set current position */
        ports = EVCdownload(memory_module);
        WriteMemory(INPUT, (phi << 8) | x);
        EVCreset(1);
        ports = EVCdownload(min_module);
        EVCreset(1);
        ports = EVCdownload(inference_module);
        EVCreset(1);
        ports = EVCdownload(max_module);
        EVCreset(1);
        ports = EVCdownload(coarse_module);
        EVCreset(1);
        ports = EVCdownload(fine_module);
        EVCreset(1);
        ports = EVCdownload(memory_module);
        ReadMemory(41,&theta);
        EVCreset(1);
        Car(x,y,phi,theta,&x,&y,&phi); /* run car */
    }
}
```

그림 12. 제안한 FLC 구현을 위한 C 응용 프로그램

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 13은 EVC1 보드네 FPGA상에 구현된 두 개의 인터페이스와 주변 Host 컴퓨터와 메모리 보드사이의 연결을 나타낸 것이다.

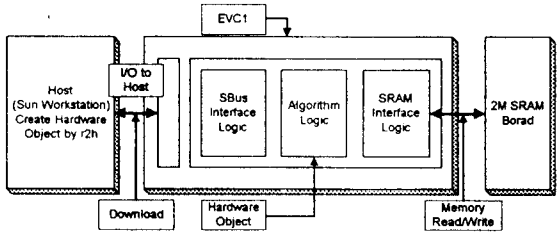


그림 13. 두 인터페이스와 주변 시스템사이의 관계

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 단방향 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.

V. 연산 속도 성능 평가

제안한 FLC를 트럭 후진 주차 문제에 적용하여 재구성 가능한 FPGA 시스템의 일종인 EVC1 보드 상에 구현한 경우와 VHDL 시뮬레이터 상에 구현한 경우, 워크스테이션상에서 C 언어에 의해 구현한 경우 각각의 제어 수행시 연산 속도를 서로 비교한다. 트럭 주차 문제의 목표는 가능한 빨리, 그리고 정확하게 트럭을 주차시키는 것이며, 이 문제는 기존 제어 기술로는 풀기 힘든 전형적인 비선형 제어 문제이다. 그림 14은 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.[14]

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

여기서 b는 트럭의 길이이며,본 논문에서는 b=4로 하다.

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 15은 Wang과 Mendel[15]이 사용한 퍼지 제어기의 입·출력 변수의 소속함수와 퍼지 제어를 위한 퍼지 규칙 베이스를 나타낸 것이다.

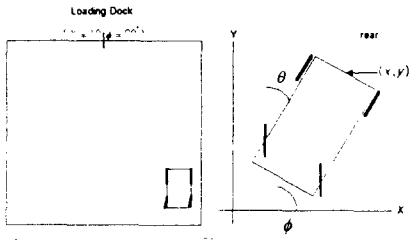
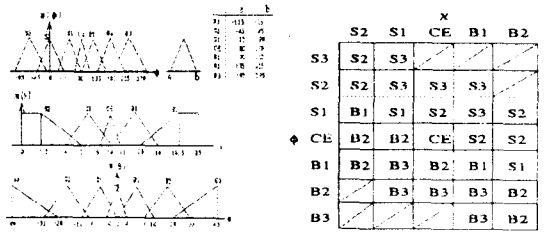


그림 14. 모형 트럭과 주차대 위치



Δ	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 15. 트럭 후진 주차 제어에 사용된 소속 함수 및 제어 규칙 베이스

제안한 FLC를 트럭 후진 주차 문제에 적용하여 재구성 가능한 FPGA 시스템의 일종인 EVC1 보드 상에 구현한 경우, 워크스테이션상에 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 환경에서 Vhdlsim을 이용하였다. 표 1은 위의 세 가지 경우의 퍼지 연산에 거리는 시간을 계산하는 과정을 나타낸 것이다.

표 1. 세 가지 경우 평균 퍼지 연산시간

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

그림 16는 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.

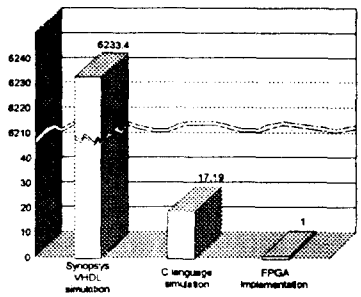


그림 16. 세 가지 경우 속도 향상비 비교

제안한 FLC를 재구성 가능한 FPGA 시스템 상에 구현한 경우 이들이 시뮬레이션에서와 똑같이 제대로 동작하는지를 알아보기 위해 임의의 세점(왼쪽 그림 시작점 : (0.0, 0.0, 0°), 가운데 시작점 (13.5., 0.0, 241°), 오른쪽 그림 시작점 (20.0, 0.0, 180°))에서의 목적지까지 이떻게 주행하는가를 조사하였다. 그림 17으로부터 세 경우 각각 16 스텝, 18 스텝, 15스텝 걸려서 목적지에 도달하는 것을 알 수 있다. 이로 미루어보아 재구성 가능한 FPGA 시스템 상에 구현한 FLC는 정확히 원하는 데로 동작하며 앞의 VHDL 시뮬레이터 상에서 보다 훨씬 빠르게 제어를 수행하고 있음을 확인할 수 있다.

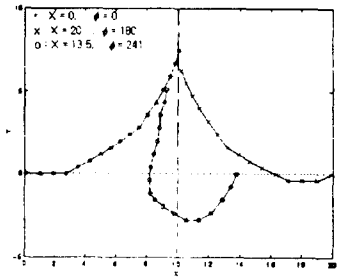


그림 20. 임의의 세점으로부터의 모형 트럭의 주행곡선

VI. 결론

본 논문은 앞서의 논문[1]에서 제안한 FLC를 재구성 가능한 FPGA 시스템 상에 직접 hardwiring에 의해 구현하였다. 본 연구에서 구현한 FLC는 기존의 FLC 구조와 크게 달라, 1) 기존의 MIN-MAX 추론을 레지스터 파일 상에 read-modify-write 연산에 의해 대치했으며, 2) 비퍼지화값을 계산하는데 있어 소속 함수의 중심 출력 값과 소속함수의 폭을 동시에 고려하였고, 나눗셈기를 사용하지 않고 coarse-to-fine 탐색법에 의해 효과적으로 모멘트 균형점을 찾는 새로운 형태의 COG 비퍼지화기를 사용하였다.

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.

재구성 가능한 FPGA 시스템 상에 구현한 FLC를 트럭 후진 주차 문제에 적용하여 그 연산 속도를 Synopsys사의 VHDL 시뮬레이터 및 워크스테이션상에 C 언어로 구현한 경우의 연산 속도와 비교하였는데, FPGA상에 구현한 경우가 VHDL 시뮬레이터보다는 $O(10^3)$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 트럭의 주행 경로를 추적하여 보았는데 세 경우 모두 14-18 스텝 내에 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 집어넣을 수 있다. 이 경우, 3번의 다운로드로 퍼지 연산이 가능하므로 연산시간을 더욱 단축할 수 있을 것으로 판단되어 MIN, Inference, MAX 모듈 통합 작업을 수행할 계획이다.

참고 문헌

[1] 김대진, 조인현, "고정밀 저비용 퍼지 제어기(I) - VHDL 설계 및 시뮬레이션" 대한 전자 공학회 제출중.

[2] SYNOPSYS, "VSS Family Tutorial v3.4," *Synopsys Corp.*, 1996.

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

[4] SYNOPSYS, "Design Compiler Reference Manual v3.4," *Synopsys Corp.*, 1996.

[5] Xilinx, "XactStep Development System User Guide," *Xilinx*, 1996.

[6] VCC, "EVC1-Virtual Computer Programming Tutorial," *VCC Corp.*, 1994.

[7] VCC, "EVC1 Technical Reference," *VCC Corp.*, 1995.

[8] D. Hung, "Custom Design of A Hardware Fuzzy Logic Controller," *FUZZ-IEEE '94*, pp. 1181-1185, 1994.

[9] Xilinx, "The Programmable Logic Data Book," *Xilinx*, 1996.

[10] Xilinx, "Xilinx Synopsys Interface FPGA User Guide," *Xilinx*, 1994.

[11] S. Casselman, "Virtual Computing and Virtual Computer," *Procrrdings of IEEE Workshop on FPGAs for Custom Computing Machines*, 1993.

[12] B. Hutchings and M. Wirhkin, "Implementation Approaches for Reconfigurable Logic Applications," *Field Programmable Logic and Applications*, pp. 419-428, 1995.

[13] S. Casselman, M. Thornburg, J. Schewel, "H.O.T(Hardware Object Technology) Programming Tutorial," *VCC Corp.*, 1995.

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

[15] 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.