논문 2004-41SD-6-9

CISC micro controller 설계 및 검증 과정에 관한 연구

(Design of CISC Micro Controller and Study on Verification Step)

김 경 수*, 박 주 성*

(Kyoung-Soo Kim and Ju-Sung Park)

요 약

본 논문은 8비트 마이크로 컨트롤러인 8051과의 호환성을 가진 16비트 마이크로 컨트롤러의 설계 및 검증 과정에 대해서 다루고 있다. 설계 디자인의 동작을 확인하기 위해 명령어별 검증과 명령어 조합에 의해 생성된 다양한 형태의 명령어 셋을 검증했다. 또한 다양한 형태의 명령어를 보다 효율적으로 검증하기 위한 방법을 제시한다. IMA-ADPCM, SOLA 등의 응용 프로그램의 검증을 통해서 설계 디자인의 동작을 확인하였다. 최종적으로 Xilinx FPGA(XCV1000-560C)를 이용한 보드 구현을 통해서 명령어 및 응용 프로그램 등의 동작을 검증했다. 타겟 컨트롤러인 8비트 마이크로 컨트롤러, 8051과의 호환성 및 성능 비교를 통해서 널리 사용 중인 8051을 대체 할 수 있고 보다 나은 성능을 발휘할 수 있다는 것을 보인다.

Abstract

In this paper, we study for the design and verification of a 16 bits micro controller, which is compatible with a 8 bits micro controller, 8051, widely used in the industrial fields these days. To confirm our design, we verified our design for all instruction sets and various combinational sets of them. Also we propose a new idea for the verification of various instruction sets. We verified our design through some application programs such as IMA-ADPCM, SOLA. Finally, we verified our design for all instruction sets and application programs through an application board, used Xilinx FPGA(XCV1000-560C). After the comparison our design with a 8051 for various cases, We concluded that we could substitute our design for a 8051 and our design could be operated more powerfully than a 8051.

Keywords : 8051, 컨트롤러, 검증, CISC, Verification

I. 서 론

현재 수 Giga Hz를 상회하는 고속의 프로세스가 상용화되어 사용되고 있지만, 아직도 산업 현장에는 마이크로 컨트롤러의 사용은 빈번하다. 주로 범용 컴퓨터의 중앙 처리 장치로 사용되는 마이크로 프로세서와는 달리 마이크로 컨트롤러는 특정 기계의 제어에 주로 사용된다. 마이크로 컨트롤러는 정해진 특정한 일들을 반복해서 처리하므로 각 응용 분야에 적합한 처리 기능은 강화하고, 필요없는 부분은 과감히 삭제하기도 한다. 반면 사용자가 원하는 어떤 일이라도 수행할 수 있도록 하기위해 다양하고 막강한 기능을 제공하는 마이크로 프로세서와는 달리, 마이크로 컨트롤러는 제어기 설계가 용이하도록 하기 위해 중앙 처리 장치뿐만 아니라 기본적인 주변 장치 즉 메모리, 인트럽트 컨트롤러 등을 포함하고 있다.[1] 여전히 다양한 응용 분야를 가진 마이크로 컨트롤러 설계는 이런 의미에서 가치가 있다.

본 논문에서는 CISC 16비트 마이크로 컨트롤러의 설계 과정과 그 검증 절차를 다룬다. 또한 설계된 마이크로 컨트롤러의 동작을 검증하기 위해 여러 단계의 검증 과정을 거치게 되는데, 본 연구자는 이런 검증 단계 과정에서 좀 더 신빙성 있고, 신속한 검증의 절차를 소개한다.

* 정회원, 부산대학교 전자공학과
(Dept. of Electronic Engineering, Pusan National University)
※본 연구는 IDEC, 부산대학교 기성회 지원을 받음
접수일자: 2003년1월17일, 수정완료일: 2004년5월17일

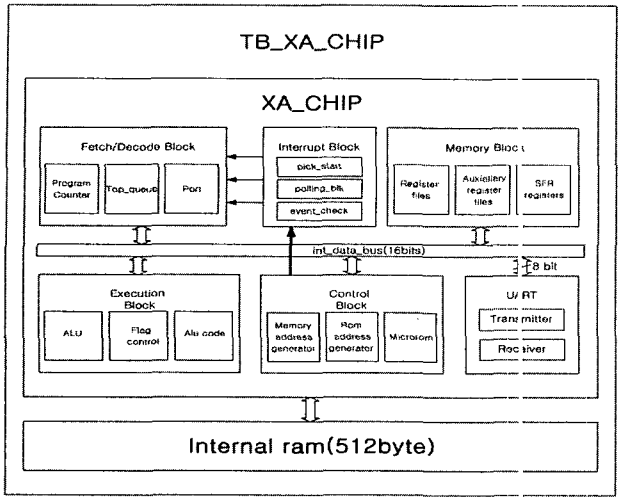


그림 1. 설계 컨트롤러의 전체 구성도
Fig. 1. The configuration of designed controller.

Ⅱ. 설계 컨트롤러의 개요

그림 1.은 설계 컨트롤러에 대한 구성도를 나타낸 블록 다이어그램이다. 그 구성은 크게 16비트의 ALU와 SFR(Special Function Register)과 내부/외부 데이터 메모리 또는 외부 프로그램 메모리와의 인터페이스 회로와 명령어 레지스터와 16비트의 내부 데이터 버스로 이루어져있다. 여기서 주의해서 볼 것은 SFR을 억세스하는 경우는 별도의 버스를 이용하지 않고 내부 버스의 하위 8비트를 사용한다는 것이다. 16비트 버스는 어드레스와 데이터가 공유된다. 이런 이유로 내부 24비트의 버스(내부버스 : 16 + SFR버스 : 8)를 16비트로 버스 사이즈를 줄임으로 불필요한 버스 사이즈의 감소를 얻을 수 있다.

1. 명령어 세트와 어드레싱 모드

설계 컨트롤러는 다양한 명령어와 어드레싱 모드를 지원한다. 설계 컨트롤러에서 지원하는 명령어 세트들의 유형은 데이타 전송, 산술 연산, 논리 연산 쉬프트 및 로테이트, 비트 연산, 조건 점프, 조건 분기 무조건적 분기, 리턴, 기타 명령어 등의 여러 형태로 나눌 수 있다. 사용 가능한 어드레싱 모드는 소스 및 목적지의 유형에 따라 레지스터 어드레싱 모드, 간접 어드레싱 모드, 간접 옵셋 어드레싱 모드, 직접 어드레싱 모드, SFR 어드레싱 모드, 상수 어드레싱 모드, 비트 어드레싱 모드 등의 7가지가 존재한다. 실제 명령어의 사용시에는 다양한 명령어 세트와 어드레싱 모드를 조합하여 다양한 명령어를 사용할 수 있다.

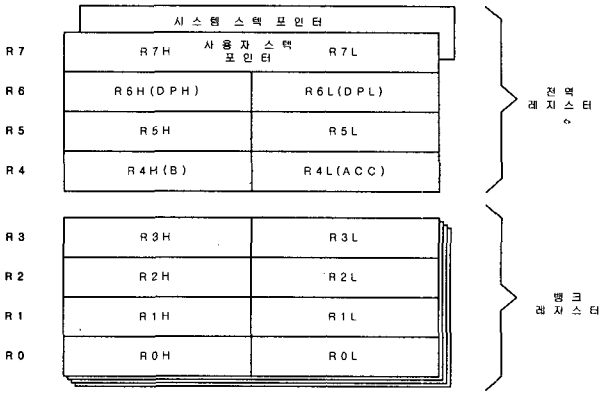


그림 2. 레지스트 파일의 구성
Fig. 2. The configuration of register file.

2. 메모리 구성

설계 컨트롤러의 메모리는 코드 메모리와 데이터 메모리가 분리된 어드레스 영역에 존재하는 하버드 구조(Harvard architecture)로 구성되어 있다.[2-3] 24비트의 어드레스에 16M 바이트의 코드/데이터 메모리 영역을 가질 수 있다. 레지스터 파일은 16개의 워드(16비트) 레지스터로 구성되어 있으며 R0~R7까지는 구현이 되어 있고, R8~R15는 필요에 따라 사용 유무를 정할 수 있도록 되어 있다. R0~R6까지는 아무런 제약 없이 사용될 수 있으며, 특히 R7은 스택 포인터로 사용될 수도 있는 레지스터이다. 그림 2.에서 보면, 8051과의 호환성을 유지하기 위해 R4와 R6은 8051에서 사용되는 레지스터로 매핑 되어 있다. R0~R3는 뱅킹 구조로 되어 있어 실제 사용할 수 있는 레지스터의 수는 4(R0~R3)×4=16개가 된다. CPU 제어나 상태 레지스터, 주변 장치와 I/O 포트를 억세스하기 위한 프로그램 수단을 제공하는 등 특수 목적을 위해 사용되는 레지스터 SFR(Special Function Register)이 존재한다.[4]

Ⅲ. 컨트롤러의 설계

1. 설계시 고려 사항

본 컨트롤러의 설계시에 8비트 마이크로 컨트롤러인 8051과의 호환성에 가장 역점을 두었다. 설계시 몇 가지 고려 사항을 보면, 첫째, 그 사용 빈도에 비해 구현 구조가 너무 복잡한 XCHD라는 한 개의 명령어를 제외하고는 명령어 레벨에서 1:1 매핑이 되도록 설계했다. 즉, 8051에서 사용된 명령어 셋은 설계 컨트롤러의 명령어 셋에 포함된다. 둘째, 8051과의 호환성을 위해서 설계 컨트롤러는 8051의 모든 메모리 구조를 포함한다.

표 1. C 연산자와 설계 컨트롤러 연산자의 매핑
Table 1. Mapping of our controller operations to C operations.

ANSI C Operator(op)	Designed controller Op codes
+=, +=+C()	ADD, ADDC
-=, -=-C()	SUB, SUBB
<, <=, ==, >=, !=	CMP
&=, |=, ^=	AND, OR, XOR

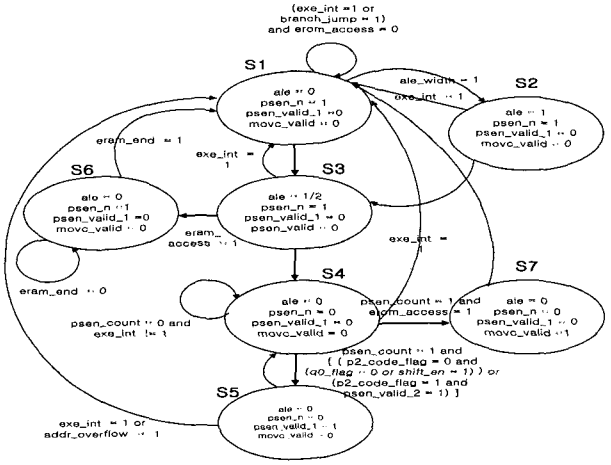


그림 3. 페치 스테이트 머신
Fig. 3. Fetch state machine.

즉, 8051의 레지스터, 코드 메모리, 데이터 메모리와 SFR(Special Function Register)을 포함하면서 그 코드/데이터 메모리의 영역은 최대 16M 바이트까지 확장하였다.[4] 셋째, C언어와 같은 HLL(High Level Langu-age)에서도 잘 적용될 수 있도록 하기 위한 명령어 세트를 가지도록 하였다. 즉, C 언어로 작성된 코드를 최소한의 명령어를 가지는 어셈블리어로 작성할 수 있도록 고려하였다. 그 예로 표 1.에서는 C 언어의 명령어와 유사한 기능의 설계 컨트롤러의 어셈블리어를 보여 준다.

2. 기능 블록의 설계

기능 블록의 설계는 크게 실행 블록, 제어 블록, 페치/디코딩 블록, 메모리 블록, 인터럽트 블록, UART 블록으로 나누어 설계되었다. 주요 블록의 구성 및 시뮬레이션 결과를 위주로 설명한다. 페치/디코딩 블록은 컨트롤러의 외부와 내부를 연결하는 xa_port와 프로그램 카운트를 저장하는 pc_blk, 외부로부터 읽어 들인 명령어 코드를 저장하는 top_queue와 현재의 페치의 상태와

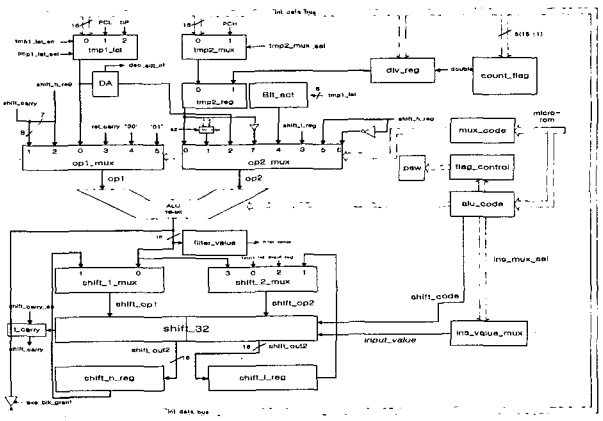


그림 4. 실행 블록의 구조
Fig. 4. The configuration of Execution block.

페치에 관련된 몇 개의 제어 신호를 생성하는 fetch와 현재 실행 중인 명령어 코드를 저장하고 있는 ir 등으로 구성되어 있다. 페치/디코딩 블록은 결국 포트를 통해서 외부의 데이터(프로그램롬, 외부램)를 읽어서 해당 명령어 또는 데이터를 분석하고 그에 해당하는 정보들을 제어 블록으로 옮겨 주는 역할을 한다. 그림 3.에서 보여지는 스테이트 머신은 이런 역할을 하는 페치 블록의 주요 부분인데, 대부분의 경우에는 모든 스테이트를 거치지 않고 S4 -> S5 -> S4 -> S5 … 과정을 반복하는 burst mode로 동작해 명령어를 읽는데 이런한 방법으로 명령어 페치에 소요되는 시간을 줄였다.[5]

명령어 큐에서 페치한 명령어를 48비트의 exe_ir에 저장한 후부터 실행 사이클이 시작된다. 명령어의 실행은 매 사이클마다 제어 블록으로부터 실행에 필요한 제어 신호를 받아 이루어진다. 16비트 ALU와 쉬프트 연산을 위한 32비트 쉬프트 레지스터 및 mux등으로 구성되어 있으며, 아래 그림 4.에 실행 블록의 전체구성을 보여준다.

16비트 ALU는 산술 연산을 위해서 16비트의 덧셈기와 뺄셈기가 구현되어 있으며 논리 연산의 수행을 위해 AND, OR, XOR, NOT 기능을 지원하며 오퍼랜드1 또는 오퍼랜드2를 bypass 할 수 있다. 표 2.에 구현된 ALU의 기능을 나타내었다.[2-3] 곱셈기와 나눗셈기는 구현되어 있지는 않고 ALU와 쉬프트 레지스터를 이용해서 곱셈 및 나눗셈을 구현한다. 8비트의 연산도 지원해야 하기 때문에 ALU의 매 연산마다 데이터 사이즈에 따라 플래그를 2개씩 발생시킨다. 즉 8비트용과 16비트용 플래그가 이에 해당한다.

제어 블록에는 a0_rom~a9_rom까지 10개의 마이크로 롬이 있으며 각 롬별로 제어 신호들이 인코딩 되어

표 2. ALU의 기능 및 동작
Table 2. ALU functions and operations.

산술 기능	ALU 코드	동작
ADD	0000	op1 + op2
ADDC	0001	op1 + op2 + Carry
ADD with tmp_carry	0011	op1 + op2 + tmp_carry
SUBB	0100	op1 - op2 - Carry
SUB	0110	op1 - op2
논리 기능	ALU 코드	동작
AND	1000	op1 AND op2
OR	1001	op1 OR op2
XOR	1010	op1 XOR op2
NOT	1011	NOT op1
MOV	1100	op2
MOV	1101	op1

있다. 제어 신호는 마이크로롬 어드레스 제너레이터(addr_gen)에서 생성된 어드레스에 의해 억세스된 마이크로롬의 데이터로서 존재한다. 마이크로롬 방식으로 제어 신호를 생성하기 때문에 비슷한 유형의 명령어에서는 많은 부분의 롬데이터를 공유할 수 있게 되어 제어 신호를 저장한 롬의 사이즈를 대폭 줄일 수 있다. 제어 코드의 생성 과정을 살펴보면, 16비트의 명령어 레지스터의 2바이트 명령어 코드를 받아서 a?_addr_gen에서 마이크로롬의 시작 주소를 생성하고, 디코딩 블록의 start_reg_en 신호가 발생하여 a?_start_reg의 마이크로롬에 저장하는 타이밍을 제공하고, 실행 시작 신호(exe_start)를 받아 a?_start_reg를 a?_reg 및 a?_rom의 입력(어드레스)으로 전달한다. 실행 시작 신호가 디세이블된 후의 실행 사이클 중에는 이전에 저장되었던 a?_reg의 내용이 a?_rom의 어드레스를 가리키게 된다. a?_rom의 내용 즉 제어 신호는 파이프 레지스터(pipe_reg)에 매 클럭 업데이트 된다. 그림 5. 에서 나타낸 것과 같이 rom_dec_mux에서 현재 선택되어야 할 a?_rom이 결정 된다. 결국 pipe_reg에는 현재의 사이클에 필요한 제어 신호가 저장되어 있다. 그림 5.에서는 rom이 a0_rom -> a1_rom -> a2_rom를 거치는 한 예를 보여준다. 명령어의 사이클별 마이크로 연산 추출 과정을 보면, 먼저 RTL 레벨의 기능 블록을 고려하여 모든 명령어의 마이크로 연산을 추출해 내고 비슷한 형태의 명령어나 같은 어드레싱 모드인 경우 동일한 마이크로 연산을 공유하도록 설계를 하여 마이크로 롬의 사용에 대한 효율성을 극대화했다.[2-3]

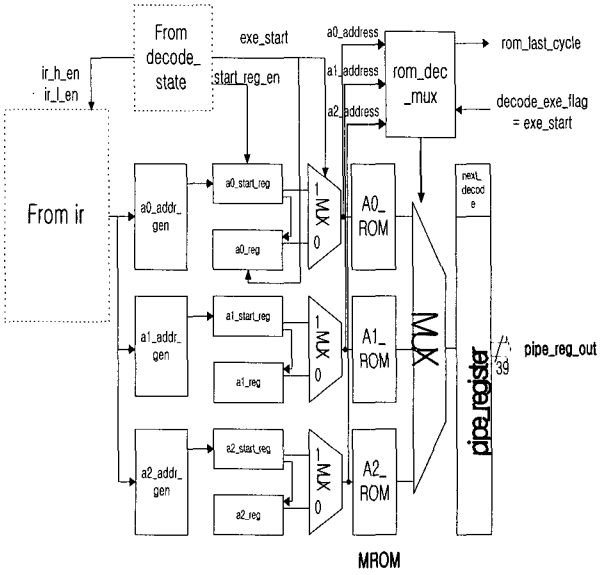


그림 5. 마이크로롬 코드의 생성 예
Fig. 5. An example for the generation of micro rom codes.

설계 컨트롤러에서의 인트럽트는 외부/내부의 요청에 의해서 발생할 수 있으며 크게 4가지로 나누어지는데 현재의 인터럽트 수준(Interrupt level)에 상관없이 항상 요청이 승낙되는 exception interrupt와 현재의 인터럽트 수준보다 높을 경우에만 그 요청을 받아들이는 이벤트 인터럽트와 이벤트 인터럽트와 비슷하지만 내부의 요청-소프트웨어적으로 SFR의 특정 비트를 세팅함으로서 이루어짐-에 의해서 이루어지는 소프트웨어 인트럽트와 trap이라는 특수한 명령어에 의해서 이루어지는 trap 인트럽트가 있다. 인트럽트 요청이 수락되어 서비스 루틴이 수행되는 과정은 크게 두가지 과정으로 볼 수 있는데, 첫째는 현재의 상태를 저장하고 둘째는 해당 인터럽트의 정보를 읽어 오는 과정이다. 좀더 구체적으로 살펴보면, 먼저 현재의 PC(Program counter)와 PSW(Program status word)를 스택에 저장한다. 이는 인터럽트 서비스 루틴이 완료되어 이전 상태로 복귀할 때 사용되는 정보이다. 그 후 요청된 인터럽트 소스에 해당되는 주소의 코드메모리의 데이터 4바이트를 읽어온다. 읽혀진 코드 메모리의 4바이트는 새로운 PSW 정보 2바이트와 인터럽트 서비스 루틴의 시작 주소 2바이트로 구성되어 있다. 4가지 종류의 인터럽트에 대한 인터럽트 서비스 루틴을 수행하는 인터럽트 블록의 구성과 ISR (Interrupt Service Routine)으로 분기시의 결과를 그림 6.과 그림 7.에 나타내었다.

본 논문에서 설계된 컨트롤러는 8xC51FB에서 사용

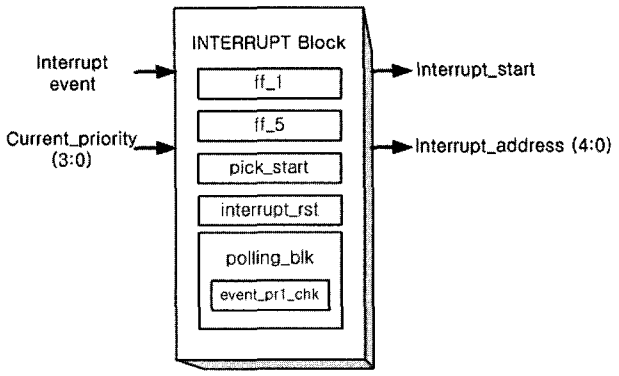


그림 6. 인터럽트 블록의 구성도
Fig. 6. The configuraton of interrupt block.

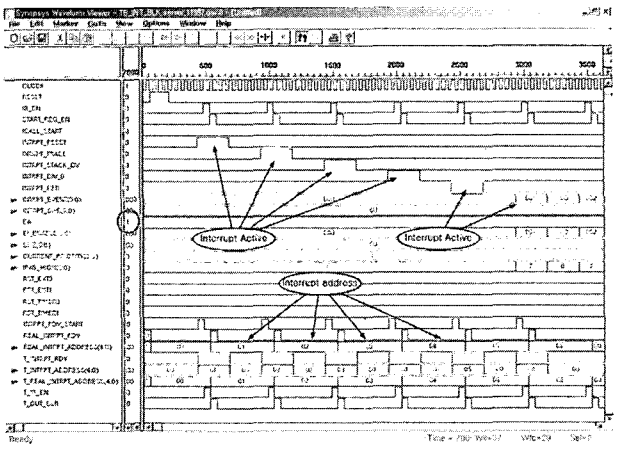


그림 7. Interrupt block 시뮬레이션 결과
Fig. 7. The simulation result for interrupt block.

되는 enhanced UART와 호환 가능한 UART (Univer-sal Asynchronous Receiver & Transmitter)가 장착되어 있다.[4] UART의 모드는 Mode0 (Serial I/O expan-sion mode), Mode1 (Standard 8-bit UART mode), Mode2 (Fixed rate 9-bit UART mode), Mode3 (Stan-dard 9-bit UART mode)의 4가지 경우가 있다. UART의 구성은 크게 Transmitter부와 Receiver부로 이루어지며 아래 그림 8.에 나타내었다.

여러 모드 중 수신부의 모드0에 대한 검증 결과를 그림 9.에 나타내었다. 모드0에서는 ri_ff_out이 '0'이고, ren='1'이면 data가 수신되기 시작한다. 수신이 완료되면 ri가 '1'이 된다.

Ⅳ. 명령어 검증법에 대한 고찰

1. 검증 절차

새로이 제시되거나 기존에 존재하는 마이크로 프로세서나 마이크로 컨트롤러를 설계할 때 가장 많은 작업

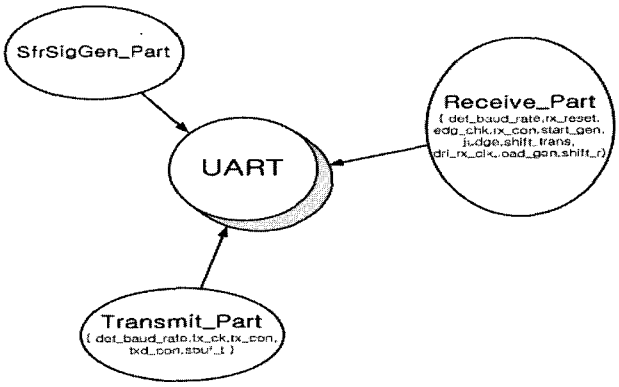


그림 8. UART 전체 구성도
Fig. 8. The configuration of UART.

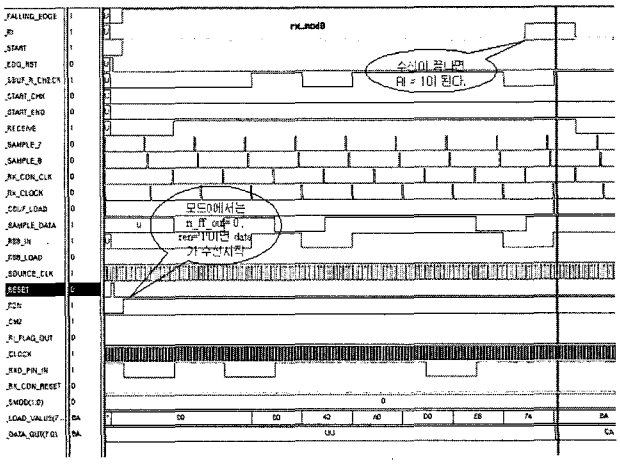


그림 9. UART 수신부 검증 결과 - mode 0
Fig. 9. UART Receiver simulation result - mode 0.

의 양을 갖는 부분은 설계된 디자인에 대한 검증 과정이다. 이 검증에는 개별 명령어의 검증과 다양한 응용 프로그램의 검증이 포함될 수 있다. 개별 명령어의 검증이 완료된 후에도 응용 프로그램의 검증시 새로운 버그가 발견되기도 한다. 기존 명령어 검증시는 발견되지 않았지만 명령어의 순서가 바뀐 경우 발견되는 이러한 에러는 잠재되어 있는 버그라 할 수 있다. 이런 이유로 개별 명령어의 검증 후에 다양한 순서로 명령어를 섞어 검증을 하는 과정을 거쳐야 한다. 이를 위해 본 연구자가 가능한 많은 수의 명령어 검증을 하고, 동일한 시간에 보다 다양한 형태의 명령어 검증을 하기 위해 수행한 절차를 소개한다. 첫번째는 마이크로 프로세서 및 마이크로 컨트롤러의 설계 후 명령어 검증 작업에서의 일련의 작업들을 보다 효율적으로 하기 위해 명령어 리스트 생성하고 이들 명령어 실행 후 결과값들-상용 시뮬레이터와 synopsys 등을 이용한 시뮬레이션의 결과값-의 비교를 텍스트 형태의 파일을 통해 수행하는 방법이다. 이런 형태의 작업을 통해서 수작업에 의한 비

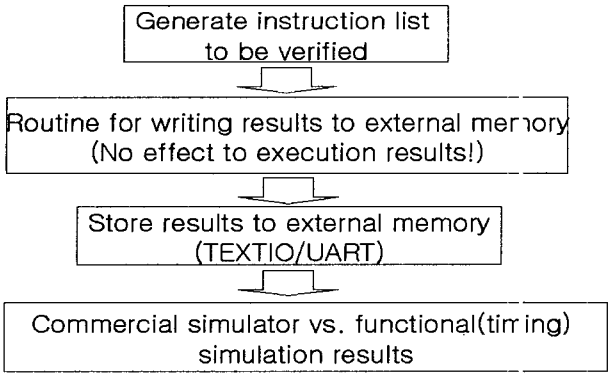


그림 10. 메모리를 쓰기를 이용한 검증 절차
Fig. 10. Instruction verification using momory.

교보다 정확성을 높일 수 있으며, 작업 시간을 현격하게 단축할 수 있다. 이 절차는 2절에서 자세히 소개한다. 두 번째 방법은 모든 명령어의 접근 가능한 모든 영역에서의 검증은 비현실적이라는 생각을 가지고, 제한된 시간 또는 제한된 명령어 개수를 가지고 보다 다양한 형태의 명령어 리스트를 생성하기 위해 임의의 값의 생성을 통해서 명령어를 검증하는 방법인데 이는 3절에서 소개한다.

2. 메모리 쓰기를 통한 검증 방법

메모리 쓰기를 통한 명령어 검증 절차를 그림 10.에 나타내었다. 먼저 검증하고자 하는 개별명령어의 리스트를 생성한다. 둘째 검증하고자 하는 명령어를 외부메모리에 저장할 수 있는 루틴을 생성한다. 이 루틴을 생성할 때 주의해야 하는 점은 외부메모리에 쓰기 루틴의 실행 전/후에 검증 하고자 하는 명령어의 결과 및 여러 상태 레지스터의 값의 변동이 없어야 한다는 점이다. 만약 이 과정에서 상태 레지스터 등의 변화가 생긴다면 명령어 검증에 오류가 발생할 수 있기 때문이다. 셋째, 이렇게 생성한 루틴을 개별 명령어 실행 후에 계속해서 호출한다. 이와 같은 방법으로 그림 11.과 같은 텍스트 파일 형태의 결과를 얻을 수 있는데, 상용 시뮬레이터에서 얻은 결과 파일과 디자인된 코드의 시뮬레이션 결과에서 얻은 결과 파일의 비교를 통해서 해당 코어가 올바른 연산을 수행하는지 여부를 알 수 있고, 차이가 발생했을 때, 명령어 리스트 상에서 어떤 명령어에서 오류가 발생했는지를 쉽게 찾을 수 있다.

3. 임의의 값 생성을 통한 명령어 검증

명령어를 검증하기 위한 일반적인 방법은 설계자가

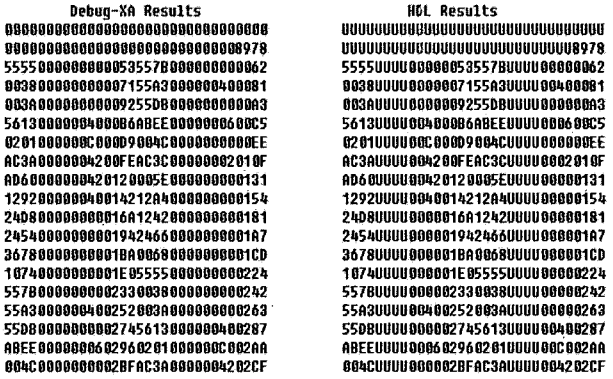


그림 11. 상용 시뮬레이터와 HDL 시뮬레이션 결과를 담은 텍스트 파일
Fig. 11. Text file including results of commercial simulator and HDL simulation.

가능한 많은 명령어 리스트를 만들고 본 장의 2절에서 제시한 방법으로 결과를 비교하는 것이다. 설계자가 가능한 경우들을 고려하여 명령어 리스트를 만드는 경우에 두 가지의 문제점을 생각할 수 있다. 첫째, 모든 명령어 리스트-접근 가능한 모든 메모리 영역과 모든 레지스터를 포함하는-를 만드는 것은 비현실적이다. 이에 대한 대안으로 명령어 리스트를 작성할 때는 모든 명령어 종류에 대해 메모리의 특정 영역, 특정 레지스터만을 포함하는 명령어 목록을 작성하는 방법이 있다. 두 번째의 문제점은 설계자에 의해서 작성되는 명령어는 명령어 실행 순서, 메모리 영역과 레지스터 선택에 있어 설계자의 주관이 포함될 수 밖에 없다. 이런 이유로 잘 실행되지 않는 형태의 명령어-Corner Cases-의 검증은 누락될 가능성이 있다.

위에서 살펴본 이러한 이유를 바탕으로 본 연구자는 일반적인 명령어 검증 후, 보다 신뢰성 있는 검증을 위해 하나의 방법을 제시한다. 본 연구자가 제시하고자하는 검증법의 기본 취지는 보다 많은 명령어를 검증하는 것보다 제한된 시간에 설계자의 주관이 배제된 보다 다양한 명령어 리스트를 생성하는데 있다. 그 방법의 대략적인 흐름을 그림 12.에 나타내었다. 먼저 C 언어를 이용해서 모든 명령어 종류에 대해서 접근 가능한 메모리 영역 및 사용 가능한 레지스터 번호를 임의로 생성한다. 또한 각 명령어 실행 줄은 실행 가능한 경우로 쓴다. 이와 같이 생성된 유사 명령어 리스트를 C에서 어셈블리어로 변환 후 본 장의 2절에서 설명된 방법으로 검증한다. 명령어 리스트는 반복해서 생성할 수 있으며, 이미 검증된 명령어는 제외시킨다. 표 3.에서 임의의 수의 생성 횟수를 증가시키면 중복되지 않는 보다

표 3. 검증된 명령어 리스트
Table 3. Verified instruction list.

	일반 명령어	분기 관련	Bit 연산	기타 명령어	소계	Random Number 생성:10
단일 사이즈, 단일 모드	35	29	5	3	72	720
사이즈 고려 (3)	74	31	5	3	113	1130
모드 고려시 (45)	236	36	8	3	283	2830
사이즈, 모드 고려시	412	39	8	3	462	4620
사이즈, 모드, 램영역 고려시	1080	51	8	3	1142	11420

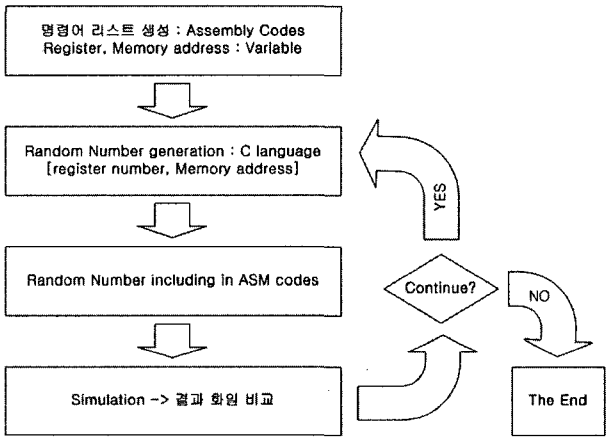


그림 12. Random Number 생성을 통한 명령어 리스트의 작성 과정
Fig. 12. The regeneration of instructions list by random number generation.

다양한 종류의 명령어를 생성할 수 있다. 본 연구자는 이와 같은 검증 방법으로 표3.에 보는 바와 같이 10회의 임의의 수 생성으로 최대 11,420개의 명령어 리스트를 생성하여 검증하였다. 완벽하게 설계된 프로세서 및 컨트롤러라면 어떤 형태로 명령어 리스트를 생성하여도 잘못된 결과가 나오지 않겠지만, 본 검증 방법은 현재 발견되지 못한 잠재적인 버그에 대해 임의의 접근을 통해서 디버깅 가능성을 제공하는데 그 의의가 있다 할 수 있다.

4. 적용 범위

2절과 3절에서 소개된 검증법을 통해서 프로세서 및 컨트롤러 설계 후 명령어와 응용 프로그램 검증시, 동일한 명령어 리스트에 대하여 논리 시뮬레이션, 타이밍 시뮬레이션, 응용 보드 상에서의 검증 작업을 반복하게 된다. 각 단계에서 해당 디버그와 컴파일러를 등을 이용해서 동일한 결과를 생성하는지 조사한다. 이 과정은 많은 시간과 노력을 필요로 하는 과정인데, 본 연구자는 본 논문에서 제시한 검증 방법을 사용하여 검증 단계별로 단일한 비교 수단으로 많은 노력과 시간을 줄일 수 있었다. 또한 수작업에 의한 결과값들의 비교가 아니라, 에디터 등을 사용한 자동화된 비교 방법을 사용하므로 결과 비교시에 발생할 수 있는 오류를 줄일 수 있었으며, 임의의 값 생성을 통한 명령어 검증을 통해서 설계자의 주관이 최대한 배제된 임의의 명령어 생성을 통해서 보다 다양하고 신뢰성 있는 명령어 리스트를 생성할 수 있었다. 본 연구자가 제시한 검증 방법은 프로세서나 컨트롤러의 설계 후 명령어 검증을 위해서 사용될 수 있으며, 별다른 수정없이 동일한 명령어 리스트에 대해 논리 시뮬레이션, 타이밍 시뮬레이션 뿐만 아니라 UART 등을 통해서 응용 보드 레벨에서도 사용 가능하다.

V. 설계 컨트롤러의 검증

1. 명령어 검증

명령어 검증을 위해서 먼저 설계 컨트롤러가 지원하는 모든 명령어에 대해 명령어 리스트를 작성한다. 그 방법은 각각의 명령어 별로 분류하고, 그것을 다시 어드레스 모드로 구분한 뒤 내부램 영역과 외부램 영역으로 구분한 뒤 명령어 리스트를 생성하였다. 외부램의 데이터를 임의의 값으로 초기화하기 위하여 Verilog HDL를 이용하여 램과 롬의 시뮬레이션 모델을 작성하였다.[6] 즉, 설계 컨트롤러는 VHDL를 이용해서 작성되고, 메모리 모델링은 파일 입출력이 용이한 Verilog HDL를 이용해 작성되었다. 결과 파일의 생성은 VHDL과 Verilog HDL를 모두 지원하는 ModelSim을 이용해서 이루어 졌다. 이렇게 작성된 명령어 리스트에 대해 Ⅳ장에서 제시된 방법으로 최대 11,420개의 명령어 리스트에 대해서 검증을 완료했다.

2. 응용 프로그램 검증

개별 명령어의 검증 완료 후 응용 프로그램을 통한 검증 과정을 거쳤다. 검증된 응용 프로그램으로는 IMA-ADPCM(Adaptive Differential Pulse Code Modulation)과 SOLA의 2가지 오디오 관련 응용 프로

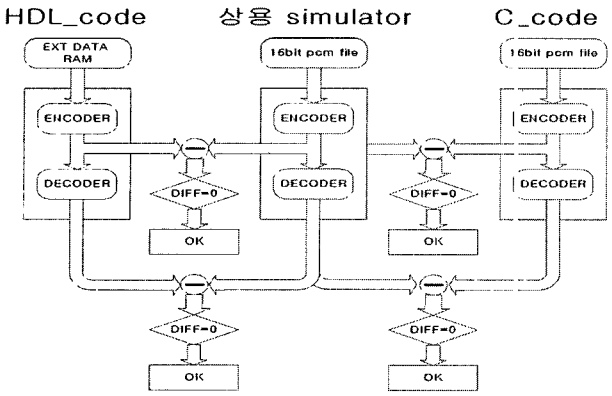


그림 13. IMA-ADPCM 검증 과정
Fig. 13. IMA-ADPCM verification flow.

그램이 있다. IMA-ADPCM에서의 결과값 검증은 C언어 상에서 계산된 엔코딩/디코딩 결과값과 HDL 시뮬레이션 상에서의 결과값의 비교를 통해서 1차적으로 검증하고, PCM 입력 오디오를 엔코딩 후 디코딩된 결과 파일의 청음 테스트를 통해서 동일한 결과를 얻을 수 있었다. 소리의 왜곡을 최소화하면서 원음을 느리게 혹은 빠르게 재생할 수 있는 SOLA 알고리즘의 검증도 IMA-ADPCM과 동일한 방법을 통해서 올바르게 동작함을 확인할 수 있었다. 아래 그림 13.에 IMA-ADPCM의 검증 과정을 보여 주고 있다.

3. FPGA 검증

논리 시뮬레이션과 타이밍 시뮬레이션의 검증 완료 후 응용 보드상에서의 검증이 이루어 졌다. 사용된 FPGA는 XCV1000-560C이며, 합성 후 총 게이트 수는 27,500이며, 동작 주파수는 25MHz(내부 12.5MHz)의 성능을 발휘한다.[5,7,8] 아래 그림 14.는 응용 보드의 구성을 보여 준다. 우측 상단의 FPGA에는 설계 컨트롤러가 들어 있으며, 좌측 하단의 FPGA에는 외부 장치(PC, Audio I/O)와의 인터페이스를 위한 회로가 들어간다. 응용 보드는 UART를 통해서 PC와 인터페이스 할 수 있으며, 코덱을 통해서 마이크 입력과 스피커 출력을 낼 수 있다. 명령어 검증은 IV장에서 제시된 방법으로 이루어 졌다. 즉, UART를 통해서 결과값을 PC에 파일 형태로 저장하고, 이를 이전 결과값들과 비교하여 검증한다. ADPCM의 결과값도 UART를 통해서 PC에 Encoding/Decoding 값을 저장해서 상용 시뮬레이터 및 C language 상에서의 결과값과 비교를 하여 검증하였다. 또한 코덱을 이용한 스피커 출력과 PC 상에서의 PCM 오디오 청취를 통한 청음 테스트도 병행하였다.

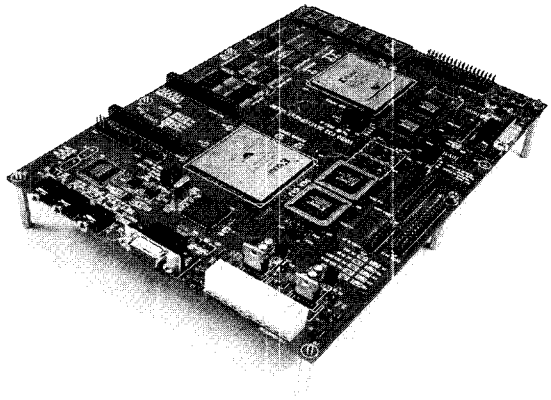


그림 14. 응용 보드의 구현
Fig. 14. Implementation of application board.

VI. 8비트 마이크로 컨트롤러와의 성능 비교

설계된 16비트 마이크로 컨트롤러의 성능을 확인하기 위해, 설계 디자인의 대체 타겟인 8비트 마이크로 컨트롤러인 8051과의 성능 비교를 했다. 동일한 조건에서 하며 인위적인 부분을 제거하기 위하여 동일 회사의 8051 C 컴파일러와 16비트 컨트롤러 C 컴파일러를 이용하였다. 같은 내용의 C 코드를 이용하여 생성된 어셈블리 코드를 이용하여 같은 동작 주파수에서 비교하였으며 HDL 코드에서도 동일한 어셈블리 코드를 이용하여 simulation 하였다. 비교 대상은 IEEE Standard for Binary Floating-Point Arithmetic(IEEE Std 754)에 따르는 32 비트의 부동소수점 덧셈/뺄셈의 연산 과정이다. 먼저 2개의 32비트의 입력(getnum)을 받는다. 계산의 편의성을 위해서, IEEE 부동소수점 포맷을 내부적으로 포맷 변환(funpack)한 후 연산을 수행 한다. 표 4.는 변환된 부동 소수점의 포맷을 보여 준다. 연산 수행 후 다시 IEEE 부동소수점 포맷으로 변환(fpack)하는 과정을 거친다. 표 5.의 비교 결과를 보면, 32비트 부동 소수점 덧셈/뺄셈 연산시 실행 시간은 1/18로 단축되었으며, 코드의 사이즈는 2/3가량으로 줄었음을 알 수 있다. 이 결과는 8비트 버스 사이즈에서 16비트 버스 사이즈로의 확장에 따른 성능 향상은 단순히 2배가 아님을 보여 준다. 이는 버스 사이즈의 확장뿐만 아니라 더욱 다양해진 명령어와 내부 구조의 향상에 기인한 결과이다.

8비트 마이크로 컨트롤러인 8051과 호환되거나 성능을 개선한 컨트롤러에 관한 연구는 많이 이루어져 왔다.[9-10] [9]는 인텔 8051 마이크로 컨트롤러(MCS51)과

명령어 수준에서 완벽하게 호환하면서 독자적인 내부 구조를 가진 마이크로 컨트롤러 설계 과정을 다루고 있다. 메모리 구조에 있어 다소의 개선은 있지만 특이한 개선 사항보다 호환성 그 자체에 초점을 맞추고 있다. 반면 [10]은 마이크로 컨트롤러의 내부 구조를 균등한 데이터패스를 갖는 5 스테이지 파이프라인으로 개선하고, 비동기 회로의 특징을 이용한 단순한 분기 예측 방법을 통해서 분기 명령어로 발생되는 제어 해저드 패널티를 감소시키는 방법 등으로 Intel 80C51에 비해 약 23.6배의 성능 향상을 갖고 Asynch80C51에 비해 약 18.9 배의 성능 향상을 갖는다. [10]에서의 성능 비교는 개별 명령어에 대한 수행 시간을 기준으로 삼고 있으며 각 명령어들은 기본적으로 8비트 사이즈 형태이다. 8비트용 컨트롤러로서의 사용 측면에서 볼 때는 [10]에서 설계한 컨트롤러가 더 나은 성능을 보이지만, 본 논문에서 제안한 컨트롤러는 8비트의 마이크로 컨트롤러와 호환성을 유지하면서 내부적으로 16비트의 구조를 가지고 모든 명령어가 16비트 사이즈를 지원하므로 16비트 이상의 데이터 사이즈를 가지는 프로그램의 실행시 성능면에서 [10]에서 제안한 컨트롤러를 능가할 것으로 기대된다.

사이즈 측면에서 살펴보면, 호환성 중심으로 설계된 [9]에서는 synopsys design complier로 합성한 결과 메모리를 제외하고 9,600개의 게이트를 가지며, 본 논문에서 설계된 컨트롤러는 27,500개의 게이트를 가져 약 2.86배의 증가를 보인다. 이는 본 논문에서 설계된 컨트롤러가 기본적으로 16비트 사이즈의 컨트롤러라는 것에 주된 이유가 있다. 본 연구에서 제안된 컨트롤러는 8비트 컨트롤러와의 호환성을 유지해서 기존의 8비트 기반으로 작성된 코드를 재사용할 수 있으며, 특히 16비트 이상의 데이터 사이즈를 가지는 프로그램의 실행시 표 5.와 같은 성능 향상을 보이므로 이는 사이즈 증가분을 충분히 보상할 수 있을 것으로 기대된다.

Ⅶ. 결 론

본 논문에서는 16비트 마이크로 컨트롤러의 설계 과정과 그 검증 절차에 관한 연구에 관해 살펴보았으며, 특히 프로세서나 컨트롤러 설계에 있어 유용하고 신뢰성 있는 명령어 검증의 한 방법을 제시했다. 제시된 검증 방법을 이용해서 개별 명령어 뿐만 아니라 모드별, 사이즈별, 메모리 영역별로 발생할 수 있는 많은 경우

표 4. 변환된 부동 소수점 포맷
Table 4. Reformatted floating point format.

Fn-Exponent (8-bit biased)	Fn-Sign (extended to 8-bits)
MS 16-bits of Mantissa (implicit 1 is present as MSB)	
LS 8-bits of Mantissa	Eight 0's

표 5. 부동 소수점 연산시 성능 비교
Table 5. Comparison by floating point operation.

	8051	Our controller
getnum	4.116	0.705
funpack	5.352	0.478
add(sub)	35.616	1.501
fpack	9.768	0.270
실행시간(ms)	54.852	2.954
Code size(byte)	3K	2K

의 명령어 조합에 대해서도 고려해 검증 과정을 거쳤다. 다양한 명령어 검증 후에 설계 디자인의 신뢰성을 높이기 위해 다양한 응용 프로그램의 구현을 통한 디자인의 검증을 하였으며, FPGA 구현을 통해 디자인의 오디오 관련 코어로서의 사용 가능성도 확인했다. 설계 디자인의 타겟 코어인 8비트 마이크로 컨트롤러와의 성능 비교를 통해서 동일한 작업량에 대해 월등한 스피드와 더 작은 명령어 코드를 가짐을 확인 할 수 있었다. 이런 결과를 통해서 설계 디자인은 현재 산업 현장이나 프린터, 팩시밀리, 복사기 및 컴퓨터 주변 장치 등의 데이터 프로세싱 분야에서 많이 사용되고 있는 8비트 마이크로 컨트롤러인 8051을 대체할 수 있을 것으로 보며, 보다 나은 성능을 발휘할 것으로 본다.

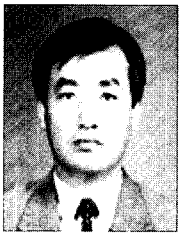
참 고 문 헌

[1] 황기수, "비메모리 산업의 마지막 선택-마이크로 프로세서 & 마이크로 컨트롤러", 전자 공학회지 제22권 제10호, 1995
[2] David A. Patterson, John L. Hennessy, "Computer Organization & Design, Hardware/Software Interface", Morgan Kaufmann, 1994.
[3] John P. Hayes, "Computer Architecture And Organization", McGraw-Hill, 2nd Edition, 1988
[4] Philips "16-bit 80C51XA Microcontrollers (eXtend

ed Architecture)", Philips semiconductors, 1996.
[5] "Digital Design and Modeling with VHDL and Synthesis", K.C.Chang, 1996.
[6] Janick Bergeron, "Writing Testbenches - Functional Verification of HDL Models", Kluwer Academic publishers, 2001.
[7] "Design Analyzer Reference", synopsys, 1996
[8] "Logic Synthesis using Synopsys", Pran Kurup & Taher Abbasi, 1995.
[9] 이용석 외 4명, "8051 호환 마이크로컨트롤러의 설계", 대한전자공학회 추계학종합 학술대회 논문집 제23권 제2호, pp.173 ~ 176, 2000년 11월
[10] 이제훈, 조경록, "CISC 임베디드 컨트롤러를 위한 새로운 비동기 파이프라인 아키텍쳐", 대한전자공학회 논문집 제40권 SD편 제4호, pp.275 ~ 284, 2003년 4월

저 자 소 개

김 경 수(정회원)
1998년 2월 부산대학교 전자공학과 졸업(공학사)
2000년 2월 부산대학교 전자공학과 졸업(공학석사)
2000년 3월 ~ 현재 부산대학교 전자공학과 박사과정
<주관심분야: 반도체, DSP 설계, ASIC 설계, 검증 자동화 기법>



박 주 성(정회원)
1976년 2월 부산대학교 전자공학과 졸업(공학사)
1978년 6월 KAIST 졸업 (공학석사)
1978년 ~ 1985년 ETRI 실장
1989년 6월 Univ. of Florida 전자공학과 박사 취득
1991년 ~ 현재 부산대학교 전자공학과 교수
1998년 ~ 현재 부산대 IDEC 센터장
<주관심분야: DSP 설계, ASIC 설계, 반도체 소자 모델링, 음성/사운드 신호 처리 및 구현, SOC 설계>