기계명령어-레벨 RTOS 시뮬레이터의 개발
(Development of Machine Instruction-level RTOS Simulator)

김 종 현 † 김 방 현 †† 이 광 용 †††
(Jong-Hyun Kim) (Bang-Hyun Kim) (Kwang-yong Lee)

요 약 실시간 운영체제 개발 환경에서 제공하는 도구 중에 하나인 RTOS 시뮬레이터는 타겟 H/W가 호스트에 연결되어 있지 않은 상태에서도 호스트에서 응용 프로그램의 개발과 디버깅을 가능하게 해주는 타겟 시뮬레이션 환경을 제공해 줌으로써, 하드웨어 개발이 완료되기 전에도 응용 프로그램의 개발이 가능하도록 해준다. 그러한 이유로 현재 대부분의 상용 RTOS 개발환경에서는 RTOS 시뮬레이터를 제공하고 있다. 그러나 그들의 대부분은 RTOS의 기능적인 부분들만 호스트에서 시뮬레이션 하도록 구현되어 있어서, RTOS 및 응용 프로그램이 실제 타겟 H/W에서 실행될 때의 실질적인 시간 추정이 불가능하다. 실시간 시스템은 정해진 시간 내에 프로그램 실행을 종료해야 하기 때문에, 실행시간 추정 기능도 가지는 RTOS 시뮬레이터가 필요하다. 본 연구에서는 RTOS 및 응용 프로그램이 실제 타겟 H/W에서 처리될 때의 실행시간 추정이 가능하고 구현도 용이한 기계명령어-레벨(machine instruction-level)의 RTOS 시뮬레이터를 개발하고, 실제 프로그램을 실행하여 기능과 정확도를 검증하였다.

키워드 : 실시간 운영체제, 시뮬레이터, RTOS 개발도구, 기계명령어-레벨

Abstract The real-time operating system(RTOS) simulator, one of the tools provided by RTOS development environment, allows users to develop and debug application programs even before the target hardware is ready. Thus, most of commercial RTOS development environments provide with RTOS simulator for the purpose. But they are implemented to simulate only functional aspects on a host system, so that it is not possible to estimate execution time of application programs on the target hardware. Since the real-time system has to complete program executions in predetermined time, the RTOS simulator that can estimate the execution time is very useful in the development phase. In this study, we develop a machine instruction-level RTOS simulator that is able to estimate execution time of application programs on a target hardware, and prove its functionality and accuracy by using test programs.

Key words : RTOS, simulator, development tool, machine instruction-level

1. 서 론

최근 정보가전 기기의 개발이 급속히 진행됨에 따라 소형의 내장 컴퓨터시스템(embedded computer system)들을 위한 고성능 저가격의 실시간 운영체제(Real-Time Operating System: 이하 RTOS라 함)의 개발이 필수적이다. 또한 그러한 시스템의 개발 과정에서 타겟 H/W(target hardware)가 개발되기 전에 호스트(host) 상에서 시스템 디버깅이나 응용 프로그램의 개발이 가능할 수 있도록 해주는 그림 1과 같은 사용자 개발환경도 필요하다. 그림 1은 RS-232를 통하여 호스트에서 타겟으로 RTOS를 다운로드하고 타겟 시스템을 부팅한 후에, 이더넷을 통하여 호스트와 타겟이 연동되는 사용자 개발환경을 나타낸다. 이러한 RTOS 사용자 개발환경은 RTOS의 선택에 중요한 조건이 되기 때문에 독립성, 개방성, 신뢰성 및 사용자의 편의성을 충족하는 사용자 개발환경에 관한 연구는 필수적이라고 할 수 있다[1].

그러나 대부분의 RTOS 개발도구들은 여전히 초보적이며 도구들의 통합성도 부족하다. 이러한 환경에서는 비효율적인 생산성과 응용프로그램을 위한 기초 구조를 만들고 통합하는데 많은 노력을 필요로 하기 때문에, RTOS 응용프로그램의 개발 기간이 길어진다. 최근에

· 본 연구는 한국전자통신연구원의 위탁연구비와 연세대학교 매지학술연구비의 지원에 의해 수행되었음
† 종신회원 : 연세대학교 컴퓨터공학과 교수
jhkim@dragon.yonsei.ac.kr
†† 학생회원 : 연세대학교 전산학과
legnamai@chol.com
††† 비 회 원 : 한국전자통신연구원 임베디드S/W연구단 연구원
kylee@etri.re.kr
논문접수 : 2004년 8월 13일
심사완료 : 2005년 1월 27일

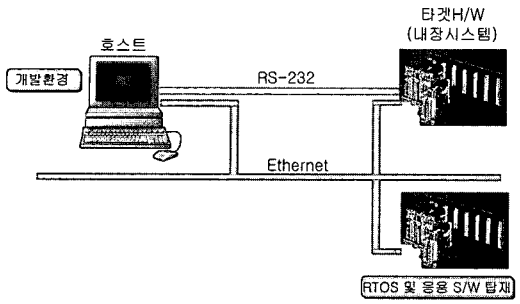


그림 1 내장 소프트웨어 개발환경

들어서 이러한 문제점을 인식하고 RTOS와 더불어 개발 기간을 단축시킬 수 있는 개발 도구들의 개발에 세계 각국의 관련 업체들이 총력을 기울이고 있다.

RTOS 개발환경에서 제공하는 도구 중에 하나인 RTOS 시뮬레이터는 타겟 H/W가 호스트에 연결되어 있지 않아도 호스트에서 응용프로그램의 개발과 디버깅이 가능하도록 해주는 타겟 시뮬레이션 환경을 제공해 줌으로써, 개발자로 하여금 빠른 시간 내에 응용프로그램을 개발할 수 있도록 지원하며 하드웨어 개발이 완료되기 전에도 응용프로그램을 개발할 수 있게 해준다. 그러한 이유로 현재 대부분의 상용 RTOS 개발환경에서는 RTOS 시뮬레이터를 제공하고 있다. 그러나 현재 상용 RTOS 시뮬레이터들은 대부분 RTOS의 기능적인 부분들만 호스트에서 동작하도록 구현되어 있어서, RTOS나 RTOS 응용프로그램이 실제 타겟에서 실행될 때의 실질적인 시간 추정이 불가능하다. 실시간 시스템은 정해진 시간 내에 프로그램 실행을 종료해야 하기 때문에, 실행시간 추정 기능도 가지는 RTOS 시뮬레이터가 필요하다.

현재 상용화되어 보급되고 있는 RTOS 시뮬레이터로는 미국의 VxWorks의 VxSim[2]과 VRTX의 MIPS XRAY Simulator[4], 그리고 RT-Linux의 Carbon-Kernel[3] 등이 있다. VxSim은 VxWorks에 포함된 시뮬레이터로서, 타겟 시스템에서 실행될 VxWorks를 호스트에서 하나의 프로세스로 간주하고 실행함으로서 가상 타겟 환경을 제공하며, VxSim 실행 창은 타겟 보드의 표준 입출력 역할을 한다. 또한 VxSim은 최대 16개까지 동시에 실행함으로서 네트워크로 연결된 타겟 하드웨어들의 시뮬레이션이 가능하며, 이들 간의 통신도 시뮬레이션이 가능하다[2]. 이 시뮬레이터는 현재 구현된 RTOS 중에서 가장 다양한 기능들을 제공하고 있으며, RTOS 기능의 대부분을 시뮬레이션 하는 강력한 기능을 가지고 있다.

VxSim과 대조적인 시뮬레이션 방법으로 구현된 MIPS XRAY Simulator는 VRTX의 시뮬레이터로서 XRAY 디버거의 부분적인 기능으로 동작한다. 이 시뮬레이터는 MIPS32 프로세서 명령어-세트 시뮬레이터로서, 응용프로그램의 소스 단계에서 명령어 자체를 해석하여 시뮬레이션 한다[4]. 타겟 응용프로그램의 개발언어인 C 언어 자체를 명령어 단위로 해석하여 기능성을 시뮬레이션하기 때문에 속도는 빠르지만, 타겟 H/W의 특성이 시뮬레이션에 반영되지 못하며, RTOS의 제한된 기능만이 시뮬레이션에 적용되는 단점이 있다. Carbon-Kernel은 사건-구동(event-driven) 시뮬레이션 기법에 기반을 둔 RT-Linux 시뮬레이터로서, 응용프로그램의 소스 코드를 분석하여 소스 코드가 실행될 때 발생하는 이벤트들을 시간선(time-line)에 스케줄링하고, 가상 RTOS 커널 API를 통하여 RTOS 동작을 수행하는 방식으로 시뮬레이션 한다[3]. 이 방식은 CarbonKernel과 RT-Linux가 같은 기반으로 구성되어 있기 때문에 RT-Linux의 기능을 거의 완벽하게 시뮬레이션 하는 장점이 있다.

국내에서도 네이버웰드의 Symphos[5]나 아로마소프트의 TeaPot[6] 등과 같이 특정 응용분야를 위한 RTOS가 구현된 적은 있으나, 활용 범위가 제한적이고 통합되어 있는 개발도구의 부족으로 인하여 실제 사용되지는 못하고 있는 실정이며, RTOS 시뮬레이터는 없는 상태이다. 따라서 RTOS가 필요한 국내업체들 대부분은 통합개발도구를 제공하는 RTOS들을 외국으로부터 도입하거나 분야별로 개량하여 사용하고 있다. 최근 국내에서는 그로 인한 막대한 기술료 지불로부터 탈피하고자 많은 노력을 기울이고 있으며, 특히 한국전자통신연구원(ETRI)에서는 프로세스 기반의 RTOS인 Q+[7]와 RTOS 개발환경인 Q+Esto[8]를 2000년에 개발하였다.

본 연구에서는 RTOS와 RTOS 응용프로그램이 실제 타겟에서 처리될 때의 실행시간 추정이 가능하고 상용화가 가능한 기계 명령어 기반(machine instruction-based)의 RTOS 시뮬레이터를 구현하는 것을 목표로 하였다. 이를 위하여 기계 명령어 레벨의 시뮬레이션 기법을 이용하여 RTOS의 실행 이미지를 생성하고, 그것이 호스트 상에서 수행되도록 하였다. 결과적으로 타겟 하드웨어에서의 RTOS 수행 동작이 호스트에서 시뮬레이션 되고, RTOS 상에서 처리되는 응용프로그램의 실행 동작도 시뮬레이션 된다. 본 연구에서 사용한 방법은 C와 같은 상위 수준의 명령어를 호스트 기반으로 시뮬레이션 하거나, VHDL과 같은 하드웨어 설계 언어 위에서 시뮬레이션 하는 기존의 방법과는 달리 타겟 시스템의 프로세서 명령어들을 타겟의 하드웨어 특성을 반영한 가상 하드웨어 위에서 시뮬레이션 한다는 점에서 구별된다.

본 논문의 구성은 2장에서 시뮬레이션 환경에 대하여 설명하였고, 3장에서는 시뮬레이터의 설계 방법에 대하

여 자세하게 분석하였다. 그리고 4장에서는 시뮬레이터의 구현에 대하여 설명하였으며, 5장에서는 시뮬레이션을 실제 사용한 예를 통하여 검증한 결과를 보여주고 있다. 마지막으로 6장에서는 연구에 대한 전반적인 결론을 요약하였다.

2. 시뮬레이션 환경

본 연구에서 시뮬레이션 대상으로 한 RTOS는 최근 한국전자통신연구원(ETRI)에서 개발된 Q+[7]이고, Q+가 동작하는 타겟 하드웨어는 ARM 계열의 StrongARM SA-110 마이크로프로세서[9]와 21285 주제어기(core logic controller)가 장착된 DTV 셋톱박스(Digital TV Set Top Box : 이하 DTSB라 함) 보드이다. 그리고 본 연구에서 개발하는 RTOS 시뮬레이터의 동작환경은 윈도우즈 NT/2000이며, Q+의 통합개발환경인 Q+Esto[8]에 도구들 중의 하나로서 포함되도록 하였다.

Q+Esto는 원격지 호스트(remote host)에서 목적 프로그램(object program)을 개발하여 타겟(target) 보드에 적재 및 실행하는 것을 지원하며, 실행 상태와 자원 사용을 추적 관찰하고, 발생하는 오류의 원인을 찾아 제거하기 위한 도구들을 통합적으로 지원하는 환경으로서, 크로스 컴파일러, 유틸리티, 원격 디버거, 대화형 쉘, 자원모니터 등의 도구들과 타겟 서버, 디버그 에이전트 등으로 구성된다.

그림 2는 이러한 Q+Esto의 구성과 타겟과의 관계를 보여주고 있다. 구성 요소들 중에서 크로스 컴파일러는 윈도우 환경에서 작동되는 gcc 컴파일러로 K&R C나 ANSI C 등으로 작성한 프로그램을 ARM 계열의 StrongARM SA-110 마이크로프로세서에서 실행할 수 있는 형태의 ELF(Executable and Linking Format)나 COFF(Common Object File Format) 파일 서식을 만들어 주는 도구이다. 그리고 원격 디버거는 호스트에서 타겟에 적재된 프로그램을 실행시키는 과정에서 오류를 쉽게 발견하고 수정할 수 있게 해주는 기능을 가진 도구이다. 대화형 쉘은 타겟의 메모리에 목적 프로그램을 적재하거나 제거하고, 타겟에 적재된 커널과 라이브러리 응용 API의 어떠한 함수도 호출하고 상호작용을 지원하는 도구이다. 또한 자원 모니터는 조립형 RTOS가 탑재되는 타겟 시스템의 상태와 운영체제 자원에 대한 정보를 원격에서 제공하는 도구이다. 타겟 서버와 디버그 에이전트는 각각 호스트와 타겟에 상주하여 상호 통신을 통해 호스트와 타겟간의 인터페이스를 구성해준다[8].

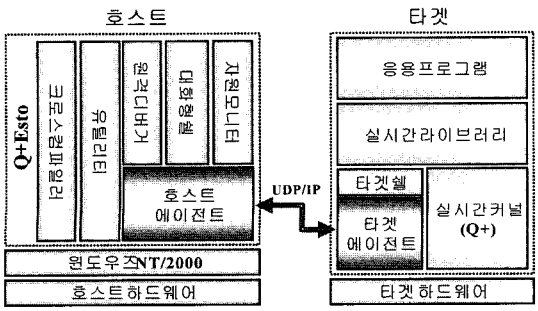


그림 2 Q+Esto와 타겟 하드웨어의 접속도

타겟 하드웨어 플랫폼으로는 ARM 계열의 StrongARM SA-110 마이크로프로세서와 21285 주제어기가 내장되어진 DTVRO사의 DTSB 보드를 사용하였다. 이 보드는 SA-110과 21285 주제어기를 평가하기 위해 제작된 EBSA-285 평가 보드[10]가 가지고 있던 기능을 대폭 향상시킨 제품이다.

SA-110 프로세서는 DEC(Digital Equipment Corporation)과 공동으로 ARM사에서 개발한 StrongARM 계열의 프로세서로, 분리 명령어(separate instruction)와 데이타 캐쉬를 사용하는 수정-하버드(modified Harvard) 구조가 처음으로 적용된 ARM V4 구조의 ARM 프로세서이며 1998년부터는 인텔(Intel)에서 생산되고 있다. SA-110은 휴대용 제품이나 대화형 디지털 비디오와 같은 내장시스템에 사용될 수 있는 저전력, 고성능의 범용 32비트 RISC 마이크로프로세서로서, 레지스터 포워딩(register forwarding)을 통한 5단계 파이프라인(5-stage pipeline) 구조를 갖고 있어 64비트 곱셈 명령어(UMULL, UMLAL, SMULL, SMLAL)와 레지스터간 다중 전송 명령어(LDM, STM), 그리고 메모리와 레지스터간 교체 명령어(SWP)를 제외한 모든 명령어들이 한 사이클 내에 실행될 수 있다[9].

3. 시뮬레이터의 설계

3.1 설계 개념

본 연구에서 개발한 Q+ 시뮬레이터(Q+ Simulator: 이하 Q+Sim이라 함)는 Q+Esto에 포함되는 도구들 중의 하나로서, DTSB 보드와 같은 타겟 하드웨어가 호스트에 연결되어 있지 않더라도 타겟을 연결하여 작업할 때와 동일한 개발 환경을 개발자에게 제공할 수 있게 하는 것을 목표로 설계되었다. 또한 기능 면에서는 기존 상용화된 RTOS 시뮬레이터에서 제공하는 기능들을 Q+Sim에서도 모두 지원하도록 하고, 이와 더불어 기존의 제품들이 가지지 못한 실질적인 수행시간 추정 기능을 추가하도록 함으로써 본 연구가 구현되어 상용화되었을 때에 차별화 된 제품으로 RTOS 시장에서 경쟁력을 가질 수 있게 설계하였다. 이를 위해 본 연구에서는 기계명령어-레벨 시뮬레이션 기법을 이용하여 RTOS의 실행 이미지를 생성하여 RTOS를 시뮬레이션 하도록

하였다. 따라서 타겟 하드웨어에서의 RTOS 동작들이 호스트에서 수행되는 시뮬레이터에 의해 동일하게 시뮬레이션 되고, 응용프로그램의 동작도 그 위에서 실행되도록 하였다.

기계명령어-레벨 시뮬레이션 기법을 이용했을 때 명령어들이 실행되는 과정은 실제 타겟 하드웨어에서 실행되는 상황과 거의 동일하게 이루어지는데, 그 과정을 보면 다음과 같다:

① 호스트 시스템에 타겟 하드웨어의 실제 기억장치와 동일하게 가상 기억장치를 구성한다.

② 직렬통신을 통하여 타겟 하드웨어로 다운로드 할 실행 이미지 파일(운영체제 및 응용프로그램)을 가상 기억장치에 적재한다.

③ 호스트 프로세서는 프로그램 카운터(PC)가 지정하는 순서대로 가상 기억장치로부터 4 바이트의 2진 명령어 코드를 인출하고, 실제 타겟 하드웨어의 프로세서와 동일한 방법으로 명령어 비트 패턴을 분석하여 명령어를 해독하고, 소프트웨어로 구현한 가상 하드웨어에 적용하여 실행한다.

결과적으로 실제 타겟 하드웨어에서와 동일한 명령어 실행 과정이 Q+Sim에서 이루어지게 된다. 또한 이와 같이 타겟 하드웨어의 동작과 동일한 방법으로 기계 명령어 실행을 시뮬레이션 함으로써 어느 정도 정확한 실행 시간 추정이 가능하며, 파이프라인이나 캐쉬와 같은 타겟 하드웨어의 특성들을 적용시키면 추정 실행시간의 정확도는 더욱 높아진다. 그림 3은 이러한 실행 과정에 대한 개념도를 보여주고 있다.

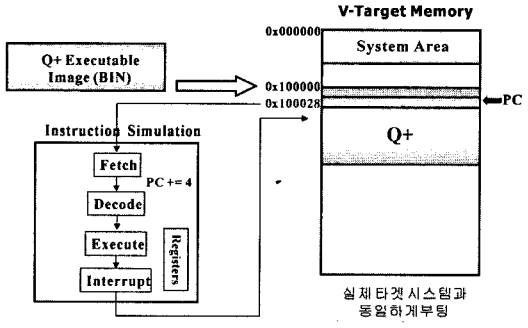


그림 3 Q+Sim의 실행 과정에 대한 개념도

Q+Sim과 Q+Esto와의 연결은 실제 타겟 하드웨어와의 연결과 동일하게 UDP/IP 통신을 통해 이루어진다. 따라서 Q+Esto는 부팅된 Q+Sim을 실제 타겟 하드웨어로 간주하고 동작하게 되므로, 사용자는 실제 타겟 하드웨어가 연결된 것과 동일한 환경에서 작업이 가능하게 된다. 이것은 부팅된 Q+Sim이 실제 타겟 하드웨어와 동일한 서비스를 Q+Esto에게 제공한다는 것을 의미한다.

이러한 기계명령어-레벨 시뮬레이터의 설계와 구현에 있어서 중요하게 고려해야 할 점은 타겟 하드웨어의 속성을 어느 정도까지 반영할 것인가이다. 타겟 하드웨어의 속성을 정밀하게 반영할수록 시뮬레이션의 정확도는 높아지는 반면에 시뮬레이션 시간이 오래 걸리기 때문에, 시뮬레이터의 사용 목적에 따라서 적절한 정밀도를 설정하는 것이 중요하다. 본 연구에서는 캐쉬 시뮬레이션에 대한 부분을 사용자가 선택할 수 있도록 함으로써, 시뮬레이터가 두 가지 정밀도를 제공할 수 있도록 설계되었다.

3.2 Q+Sim의 구조와 기능

Q+Sim은 그림 4와 같이 Q+ 에이전트(Q+ agent), 시뮬레이션 제어기(simulation controller), Q+Sim 쉘(Q+Sim shell), 그리고 Q+Sim 코어(Q+Sim core)로 구성되어 있다. Q+Sim 코어는 ARM 프로세서를 시뮬레이션하기 위한 가상 하드웨어로서, 이식성을 위하여 ANSI C로 구현하였고, 나머지는 윈도우즈 프로그램으로 구현하였다. 여기서 Q+Sim 에이전트는 타겟 서버와 UDP/IP 통신을 통해 Q+Sim과 Q+Esto의 인터페이스 역할을 하며, 실제 타겟에서의 디버거 에이전트와 동일한 역할을 수행한다. 따라서 타겟 서버는 Q+Sim을 실제 타겟으로 간주하고 동작하게 된다.

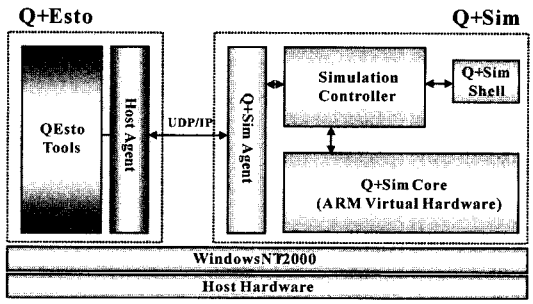


그림 4 Q+Sim의 구조

시뮬레이션 제어기는 Q+ 에이전트로부터 요청된 명령을 해석하여 가상 하드웨어로 전달하고, 처리된 결과를 Q+ 에이전트로 전달한다. 즉, 시뮬레이션 제어기는 Q+에이전트와 가상 하드웨어간의 인터페이스라고 볼 수 있으며, Q+ 에이전트와 시뮬레이션 제어기 사이의 통신 형태와 내용은 실제 타겟에서 타겟 에이전트와 Q+간의 통신과 동일하게 이루어진다. 이와 더불어 시뮬레이션 제어기는 Q+Sim 쉘과 가상 하드웨어간의 인터페이스 역할도 수행하게 되는데, Q+Sim 쉘에서 입력된 명령을 해석하여 가상 하드웨어로 전달하고, 가상 하드웨어로부터 되돌아 온 결과를 Q+Sim 쉘로 되돌려준다. 시뮬레

이션 제어기의 또 다른 역할은 전체 시뮬레이션에 관한 정보를 제공하는 것으로서, 이것은 가상 하드웨어의 주요 부분에서 기록된 자료를 정리하여 개발자에게 보여주기 위한 것이다. 이러한 기능들을 수행하기 위해 시뮬레이션 제어기는 가상 하드웨어의 인터럽트를 사용하여 가상 하드웨어를 제어한다.

Q+Sim 쉘은 가상 타겟 쉘로서 실제 타겟의 표준 입출력으로 동작하며, Q+Sim의 실행창의 일부분으로 나타난다. 예를 들어 응용 프로그램이 실제 타겟의 표준 출력을 통해 출력을 수행하려고 할 때에 가상 하드웨어는 출력 요구를 시뮬레이션 제어기에 전달하고, 시뮬레이션 제어기는 출력될 메시지를 Q+Sim 쉘에 전달하여 출력한다. 반대로, 가상 하드웨어에서 표준 입력 요구가 발생하면, 시뮬레이션 제어기가 요구를 받아서 Q+Sim 쉘로 전달하여 표준 입력을 받아들인다. 또한 Q+Sim 쉘에서는 시뮬레이터 제어를 위해 소수의 콘솔 관리 명령어를 사용할 수 있도록 설계하였다.

DTSB 보드를 모델링하여 가상 하드웨어로 구성한 Q+Sim 코어는 그림 5와 같이 처리 코어(processing core)와 가상 기억장치(virtual memory)로 구성된다. 처리 코어는 마이크로프로세서에 해당하는 부분으로서, 32-비트 레지스터 37 개가 32-비트 변수 배열 형태로 포함되어 있으며, 명령어 실행 시뮬레이션을 처리하는 부분이다. 그리고 가상 기억장치는 타겟 하드웨어의 16K바이트의 명령어 캐쉬(Icache), 16K바이트의 데이타 캐쉬(Dcache), 그리고 32M바이트의 기억장치를 가상적으로 구현한 부분으로서, Q+Sim이 동작하는 시스템 자원의 변화에 적응할 수 있도록 동적 메모리를 이용한다. 이와 같이 동적 기억장치 방식을 사용함으로써, 기억장치 용량에 대한 프로그램의 환경 설정에 따라 가상 기억장치가 유연성 있게 생성될 수 있는 이점을 가질 수 있게 되었다.

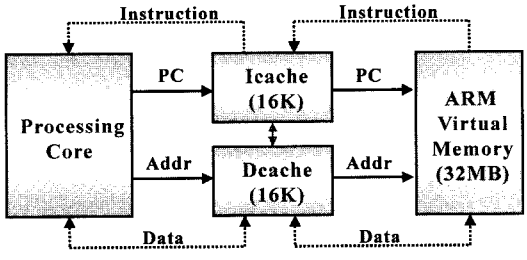


그림 5 Q+Sim 코어

3.3 기계명령어-레벨 시뮬레이션

타겟 하드웨어를 시뮬레이션 하는 방법에 따라 명령어 시뮬레이션의 정밀도는 세 가지로 분류될 수 있다. 첫 번째는 기능적 정밀도(functional accuracy)로서, 타겟 하드웨어에서 동작될 프로그램을 호스트 기반으로 컴파일 하여 호스트에서 실행하는 경우이다. 이 경우에 타겟 하드웨어의 변동에 영향을 거의 받지 않으며, 또한 프로그램의 소스 레벨에서의 기능 확인 및 디버그가 가능하다. 그러나 실제 타겟 하드웨어에서의 실행 시간 분석은 불가능해진다. 두 번째는 명령어 단계의 시간 정밀도를 갖는 소프트웨어-사이클 정밀도(software-cycle accuracy)서, 크로스 컴파일 된 타겟 환경의 목적 프로그램을 명령어 시뮬레이터(instruction set simulator)가 명령어의 기능적인 면을 시뮬레이션 한다. 이 경우에는 명령어 수준의 사이클 측정이 가능하지만, 캐쉬나 버스 중재와 같은 하드웨어적인 요소는 반영이 되지 않는다. 세 번째는 하드웨어-사이클 정밀도(hardware-cycle accuracy)서, 실제 타겟 하드웨어와 같은 역할을 할 수 있게 구성된 가상 하드웨어 위에서 명령어 시뮬레이터가 가상 하드웨어 클록에 따라 크로스 컴파일 된 타겟 환경의 목적 프로그램을 수행한다. 이 경우에는 실제 타겟 하드웨어에서의 기능적 및 시간적 동작을 거의 동일하게 시뮬레이션 할 수 있지만, 복잡성 때문에 구현하기 어렵고 시뮬레이션 시간이 길어진다는 단점이 있다[11].

본 연구는 소프트웨어-사이클 정밀도와 하드웨어-사이클 정밀도의 중간 정도를 지향하여 설계함으로써 양쪽의 장점을 모두 가지게 하였다. 이를 위해 가상 하드웨어를 C 언어로 구현하고, 그 위에서 타겟 프로세서 명령어로 컴파일 된 실행 이미지를 해석하여 시뮬레이션 하는 기계명령어-레벨의 명령어 시뮬레이션 방법을 사용하였다. 명령어 시뮬레이션이 동작될 가상 하드웨어 환경으로는 DTSB 보드를 모델링 하였으며, 시뮬레이션 될 명령어는 SA-110 마이크로프로세서용으로 크로스 컴파일된 ELF 형태의 목적 프로그램을 사용하였다. ELF는 USL(UNIX System Laboratories)에서 처음 개발된 이진 형식으로 기존의 UNIX의 표준 이진 형식에 비하여 향상된 유연성을 가지고 있다[12]. 이러한 ELF는 시뮬레이션에 사용될 명령어에 대한 유연성을 제공할 수 있기 때문에, 본 연구에서는 ELF 중에서 실행 파일(executable file) 형태의 Q+와 재배치 가능 파일(relocatable file) 형태의 응용 프로그램을 사용하여 시뮬레이션을 수행한다.

SA-110 프로세서의 명령어는 기본 명령어 47 가지와 보조프로세서 명령어들로 구성된다. SA-110의 명령어들은 연산코드(opcode)에 접미 코드가 붙거나, 오퍼랜드(operand)에 쉬프트 연산과 같은 부가 기능이 들어갈 수 있기 때문에 실제로는 무수히 많은 종류의 명령어들이 존재할 수 있다. 또한 명령어의 비트 패턴은 복잡하게 구성되어 있으며, 명령어에 따라 비트 패턴이 의미하

는 것이 달라진다[13]. 본 연구에서는 이렇게 복잡한 명령어들의 비트 패턴을 체계적으로 정리하여 명령어 시뮬레이션이 가능하도록 하였다. 전체적인 명령어 시뮬레이션 과정은 SA-110이 명령어를 수행하는 방법과 유사하게 설계되었으며, 복잡성을 감소시키기 위하여 기능에 영향을 받지 않는 부분들은 추상화시켰다. 따라서 가상 기억장치에 있는 명령어를 인출하여, 해독하고, 실행하는 전형적인 명령어 실행 방법으로 명령어 시뮬레이션을 진행하며, 해당 명령어의 부가적인 특성이 필요한 부분은 기본 구조에 기능을 추가하는 방법으로 설계되어 그림 6과 같은 명령어 사이클로 명령어를 시뮬레이션 하였다.

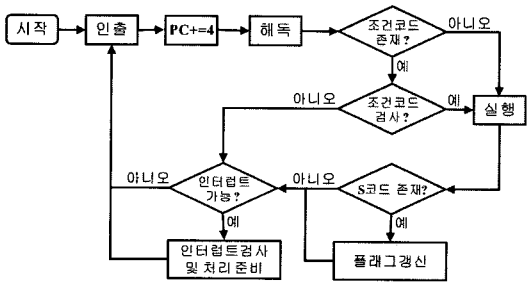


그림 6 명령어 사이클

3.4 실행시간 추정 기능

Q+Sim은 기계명령어-레벨의 시뮬레이션을 수행하기 때문에, 실행된 명령어들의 계수(counting)를 통하여 기본적인 실행시간 추정이 가능하다. 이 경우에 SA-110 프로세서가 하나의 CPU 클록마다 평균 한 개의 명령어를 실행한다고 가정하고 A라는 함수가 수행되는 동안 1000개의 명령어가 수행되었다면, 실행 시간은 1000 클록이며 200 Mhz의 SA-110 경우에 5 μs가 된다. 그러나 실제로는 실행되는 프로그램에 따라 명령어 실행 파이프라인과 캐쉬의 효율이 달라지기 때문에 정확한 실행 시간이 될 수 없다.

따라서 본 연구에서는 파이프라인과 캐쉬의 효율을 실행시간에 반영하기 위하여, Q+Sim 가상 하드웨어에 실제 SA-110의 파이프라인 및 캐쉬의 영향을 위한 부분을 소프트웨어로 구현하였다. 파이프라인은 소프트웨어 복잡도를 고려하여 분기되는 명령어들을 계수하는 방식으로 설계되었고, 캐쉬는 명령어 캐쉬와 데이타 캐쉬를 각각 16K 바이트 동적 메모리로 구성하여 32-way 세트-연관 사상 방식과 라운드 로빈 교체 방식으로 동작하도록 함으로써 실제 물리적인 캐쉬와 동일하게 설계되었다. 그러나 캐쉬와 파이프라인을 시뮬레이션에 포함하는 경우에는 Q+Sim의 수행 시간이 매우 길어지기 때문에, 이 기능들은 실행시간 추정 기능을 수행하는 경우에만 동작하도록 구현하였다.

4. 시뮬레이터의 구현

4.1 프로그램 구조

Q+Sim 프로그램은 크게 제어부와 처리부로 구성된다. 제어부는 Q+Sim 에이전트와 Q+Sim 쉘 및 프로그램의 골격을 이루는 부분으로 이루어지는데, 전체 프로그램의 입출력과 제어 기능을 가지고 있으며, 윈도우즈 프로그램으로 구현하였다. 처리부는 Q+Sim 코어에 해당하는 부분으로서, 명령어 시뮬레이션 기능을 수행하며, ANSI C로 구현되었다. 처리부는 가상 기억장치 생성 및 입출력을 담당하는 기억장치 관리부와 명령어 시뮬레이션을 담당하는 명령어 처리부로 나누어진다. Q+Sim 프로그램의 수행이 시작되면 제어부로부터 입력을 받아서 처리부로 해당 모듈을 호출하는 형태로 프로그램이 수행된다. 제어부와 처리부 사이의 데이타 교환은 그림 7과 같은 포인터형 자료 구조를 통하여 이루어지며, 주로 SA-110 프로세서의 레지스터들을 나타낸다.

```
typedef struct {
    WORD32  r[140];        // registers
    WORD32  ir;            // instruction register
    WORD32  mar;           // memory address register
    BYTE    opcode;        // opcode
    BYTE    Rn;            // first source operand register
    BYTE    Rd;            // destination register
    BYTE    Rs;            // source register
    BYTE    Rm;            // register address
    ...
} ARMT;
ARMT *arm;
```

그림 7 제어부와 처리부간 데이타 교환용 자료 구조

4.2 Q+Sim 에이전트

Q+에서 UDP/IP 통신은 인터럽트 11번을 통하여 이루어진다. 이러한 통신 동작을 시뮬레이션하기 위하여 Q+Sim 에이전트의 통신 모듈은 MS 윈도우즈 NT/2000에서 동작할 수 있도록 MS Visual C++를 이용하여 구현하였으며, UDP/IP 메시지를 수신한 Q+Sim 에이전트의 통신 모듈은 Q+Sim 코어에 인터럽트 11번을 강제로 발생시키도록 구현하였다. Q+Sim 에이전트는 그림 8에서 보는 바와 같이 윈도우즈 통신 모듈을 통하여 Q+Esto로부터의 RPC 요구를 수신하면, Q+Sim 코어의 가상 기억장치에 있는 수신 영역(taXport)에 수신된 데이타를 저장하고, Q+Sim 코어에 인터럽트 11번을 발생시킨다. 이후의 RPC 메시지 처리는 Q+Sim 코어의 동작에 의하여 처리되는데, Q+Sim 코어는 RPC 처리가 끝나고 RPC 응답의 송신 함수가 실행되면 Q+Sim 에이전트에게 RPC 응답의 송신 요구를 보낸다. Q+Sim 코어로부터 RPC 응답의 송신 요구를 받은 Q+Sim 에

이전트는 가상 기억장치에 있는 송신 영역(taXport)에 저장된 데이타를 Q+Esto로 전송한다. 이와 같은 동작은 부팅된 Q+Sim의 가상 타겟이 실제 타겟과 동일한 서비스를 제공할 수 있게 해준다.

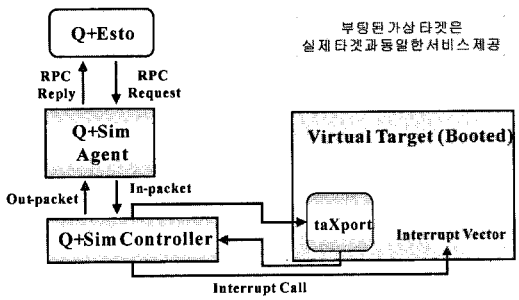


그림 8 Q+Sim 에이전트의 동작 흐름도

4.3 Q+Sim 쉘

Q+Sim 쉘은 Q+Sim의 실행 창을 통해 표준 입출력을 수행하도록 구현되었다. Q+Sim 쉘을 통하여 입력된 내용은 Q+Sim 엔진의 표준 입력으로 전달되거나 시뮬레이션을 위한 관리 명령어로 수행되고, Q+Sim 엔진으로부터 발생한 표준 출력과 시스템 정보 메시지는 Q+Sim 쉘의 출력창을 통하여 출력된다. Q+Sim 쉘에서 구현된 관리 명령어는 표 1과 같으며, 인자 이름 뒤의 숫자는 진법을 의미한다.

4.4 기억장치 관리부

기억장치 관리부는 주기억장치를 관리하는 부분과 캐쉬를 관리하는 부분으로 구성된다. 주기억장치의 크기는 가변적이므로 동적 메모리를 이용하여 구현하였고, 반면에 캐쉬의 크기는 고정적이므로 정적 메모리를 이용하여 구현하였다. 이렇게 구현된 주기억장치와 캐쉬를 위한 자료 구조는 그림 9와 같다.

```
#define DCACHE_SIZE        16*1024
#define ICACHE_SIZE        16*1024
#define CACHE_WAY          32
typedef struct {
   WORD32  tag[CACHE_WAY];
   WORD32  data[CACHE_WAY];
   BYTE    state[CACHE_WAY];       // INVALID|VALID|DIRTY
   BYTE    rr;                     // Round-robin counter
} CST;                             // Cache Set Type
CST        icache[ICACHE_SIZE/(CACHE_WAY*4)];
CST        dcache[DCACHE_SIZE/(CACHE_WAY*4)];
BYTE *qsim_mem;                    // main memory
```

그림 9 가상 기억장치를 위한 자료 구조

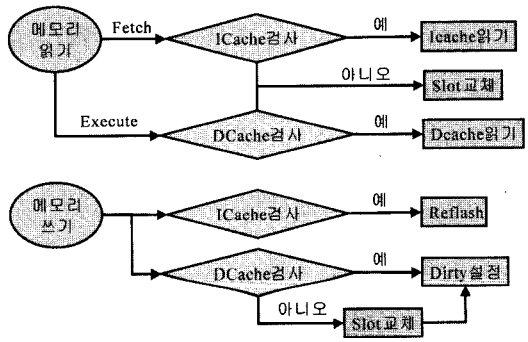


그림 10 캐쉬의 동작 과정

데이타 캐쉬와 명령어 캐쉬에 대한 접근의 구분은 명령어 사이클의 단계를 기준으로 구현하였는데, 인출 단계에서 발생하는 기억장치 요구는 명령어 캐쉬 요구를 발생시키고 그 외의 경우에는 데이타 캐쉬 요구를 발생시켰다. 또한 데이타 캐쉬의 write-back 쓰기 정책과 이에 따른 캐쉬 일관성 유지를 위하여 캐쉬 슬롯 당 상태 필드를 두어 'INVALID', 'VALID', 'DIRTY' 중의 하나로 설정되게 구현하였다. 'INVALID'는 현재 캐쉬 슬롯에 있는 데이타가 무효라는 것을 나타내고, 'VALID'

표 1 Q+Sim 쉘 관리 명령어

관리 명령어 형식	동 작
f [function_name]	[function_name]의 함수를 실행. 인자가 필요한 경우에는 "rw" 명령을 이용하여 r0부터 순서대로 입력.
g [address16]	PC의 값이 [address16]이 될 때까지 Q+Sim을 동작.
h	Q+Sim 쉘 명령어들에 대한 도움말을 출력.
i [interrupt_num10]	인터럽트 [interrupt_num10]을 발생.
n {count10}	{count10}개의 명령어를 실행. {count10}이 생략되면 1로 간주.
m [address16] {count10}	메모리 주소 [address16]으로부터 {count10}개의 연속된 값을 출력. {count10}이 생략되면 1로 간주.
mw [address16] [value16]	메모리 주소 [address16]의 값을 [value16]으로 변경.
r	현재 레지스터들의 내용을 출력.
s	Q+Sim의 동작을 정지.
resume	정지된 Q+Sim의 동작을 재개.
rw [reg_num10] [value16]	레지스터 [reg_num10]의 값을 [value16]으로 변경.
t	호출 함수에 대한 추적(trace)기능의 on/off를 전환.

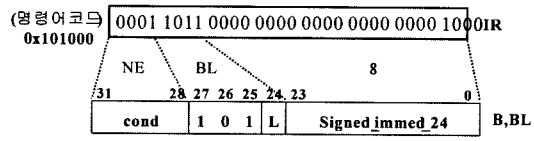
는 데이타가 유효임을 나타내며, 'DIRTY'는 수정되었기 때문에 캐쉬 슬롯 교체 시에 기억장치에 저장되어야 하는 데이타를 나타낸다. 그리고 캐쉬가 교체될 경우에는 각 세트(set)에 있는 'rr' 카운터 필드를 사용하여 교체될 슬롯을 결정하고 교체한다. 그림 10은 이러한 캐쉬의 동작 과정을 보여주고 있다.

4.5 명령어 처리부

명령어 처리부는 그림 6의 명령어 사이클을 수행하며, 명령어를 인출하는 부분, 조건 코드를 검사하는 부분, 명령어를 해독하는 부분, 명령어를 실행하는 부분, 그리고 인터럽트를 처리하는 부분으로 구성된다. 명령어의 인출 동작에서는 현재 주소 레지스터(arm→mar)에 저장된 주소를 사용하여 Q+Sim 코어에 있는 가상 메모리의 해당 영역을 읽어 명령어 레지스터(arm→ir)에 저장하며, PC의 값을 4만큼 증가시키고 이 값을 주소 레지스터에 저장한다. 조건 코드의 검사는 명령어 레지스터에 저장된 명령어의 최상위 4비트의 조건 코드를 해독하여 이루어진다. 조건 코드는 4비트이므로 16가지의 조건 코드가 존재하며, 상태 레지스터의 상태 플래그(Negative|Zero|Carry|oVerflow)를 사용하여 조건을 검사한다. 만약 조건을 만족하지 못하면 인터럽트를 처리하는 부분을 실행하고, 조건을 만족하면 명령어 레지스터에 있는 명령어를 해독한 후에 해독된 명령어를 실행한다[13].

명령어의 해독은 연산코드와 오퍼랜드로 나누어 진행되며, 연산코드에 따라 오퍼랜드의 해독이 달라진다. 즉, 연산코드의 성격에 따라서 오퍼랜드의 형태가 다섯 가지로 분류되는데, 본 연구에서는 보조프로세서 명령어의 연산코드와 오퍼랜드 형태를 제외한 나머지 명령어의 해독을 구현하였다. 연산코드의 구현은 각 명령어의 비트 패턴을 분석한 후에 이들을 분류하여 하나의 비트 패턴이 하나의 명령어에 대응되도록 하였고, 오퍼랜드의 해독은 해독된 연산코드에 해당하는 오퍼랜드의 형태가 대응되도록 하였다.

예를 들어 현재 PC의 내용이 '0x101000'이고, 가상 메모리의 '0x101000' 번지 내용이 '0001 1011 0000 0000 0000 0000 0000 1000'인 경우에 명령어 시뮬레이션은 그림 11과 같이 수행된다. 먼저 인출된 명령어는 명령어 레지스터(arm→ir)에 저장되고, PC의 내용은 '0x101004'로 증가된다. 인출된 명령어 코드의 최상위 비트 '0001'은 조건 코드 'NE'(not equal)를 의미하며, 만약 상태 레지스터의 'Z' 플래그가 '0'이면 명령어를 해독하고 실행한다. 명령어 해독은 'arm→ir'의 27번 비트부터 24번까지의 4비트를 해석하여 이루어지며, 이 경우에는 'BL' 명령어로 해석된다. 'BL' 명령어는 'branch and link'를 나타내는 약자로 현재 PC의 주소를 저장하고 분기하는 명령어이다. 'BL' 명령어 형식에서 'arm→ir'의 23번 비트부터 0번까지의 24비트는 'Signed_immed_24'로서, 분기할 위치에 대한 4바이트 단위 거리의 상대주소를 의미하는 상수 오퍼랜드이다. 따라서 분기할 절대주소는 'PC + (signed_immded_ 24 << 2) + 4'로 계산되어 '0x101028'이 되며, 명령어는 니모닉 어셈블리 문법으로 'BLNE 0x101028'으로 해석된다. 결과적으로 명령어의 실행은 그림 11의 '수행된 연산'과 같이 이루어진다.



(해석된명령) **BLNE 0x101028**

(수행된연산) **arm->r[arm->mode+LR] = arm->r[arm->mode+ PC];**
arm->r[arm->mode+ PC] = arm->r[arm->mode+ PC]
+ (signed_immed_24 << 2) + 4;

그림 11 분기 명령어 시뮬레이션의 예

5. Q+Sim의 사용 예

Q+Sim에서 타겟에 다운로드 될 Q+ 이미지 파일(.bin)을 지정하고 시뮬레이션을 시작하면, 그림 12와 같이 부팅이 시작된다. 부팅 중에는 하단의 상태 바에 "On Booting"이라는 메시지가 나타나고, 정상적으로 부팅이 완료되면 "Running"이라는 메시지가 나타난다. 부팅 후에 Q+Esto의 호스트 에이전트 환경 설정에서 타겟의 네트워크 주소를 Q+Sim의 네트워크 주소로 설정하고, 호스트 에이전트를 실행하면 Q+Esto와 Q+Sim간에 RPC 통신 경로가 설정된다. 이 때부터 Q+Esto는 Q+Sim을 실제 타겟으로 여기고 동작하게 되며, Q+Sim도 Q+Esto의 모든 도구들을 사용할 수 있게 된다.

그림 13은 Q+Sim과 Q+Esto의 도구인 원격 디버거와의 연동을 검증하기 위하여 Q+에 내장된 테스트 프로그램인 "sh_test1()"에서 소스 내에 브레이크 포인트(break point)들을 설정하고, 한 스텝(step) 또는 브레이크 포인트까지 실행을 하며 디버깅을 한 화면이다. Q+Esto의 원격 디버거는 디버깅을 위하여 정의되지 않은 명령어(undefied instruction)에 대한 예외 처리(exception handling)를 이용하는데, Q+Sim은 이 예외 처리를 수행함으로써 원격 디버거와의 연동을 지원한다.

Q+Sim의 표준 입출력 시뮬레이션과 타이머 인터럽트를 이용한 시분할 스케줄링 시뮬레이션, 그리고 실행시간 추정기능을 검증하기 위하여 그림 14와 같은 테스트용 C 프로그램을 사용하였다. 'qsimtest1()' 함수는 문자

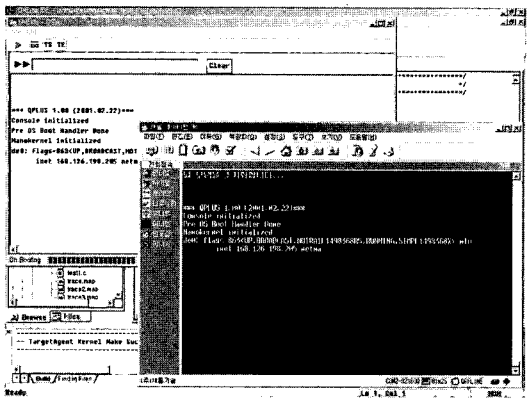


그림 12 부팅 중의 Q+Sim

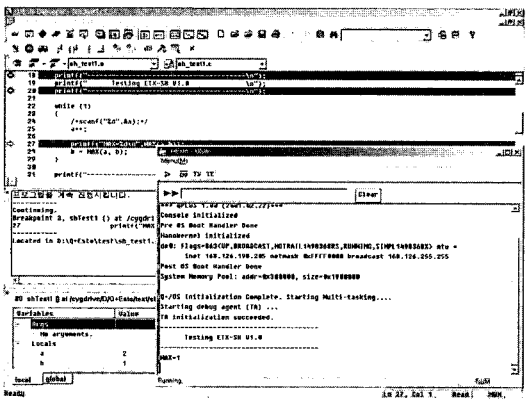


그림 13 Q+Sim과 원격 디버거의 연동

열을 입력받아서, 입력된 문자열과 그 길이를 화면에 출력하는 예제이고, 'qsimtest2()' 함수는 "First Loop 1 COUNT"를 무한루프로 출력하는 예제이며, 'qsimtest3()' 함수는 "Second Loop 2 COUNT"를 무한루프로 출력하는 예제이다.

실제 DTSB 보드를 연결했을 경우에는 DTSB 보드와 호스트 컴퓨터 사이를 COM1 또는 COM2로 연결하여 "새롬 데이타맨"과 같은 직렬통신 프로그램을 통해 표준 입출력을 수행한다. Q+Sim은 호스트 컴퓨터의 COM1과 COM2를 크로스 케이블(cross cable)로 연결한 상태에서는 실제 DTSB 보드와 연결했을 경우와 동일하게 입출력을 수행하는데, 그림 15는 'qsimtest1()' 함수를 사용하여 표준 입출력을 검증한 결과를 보여준다. 또한 Q+Sim은 실행창을 통해서도 이와 동일한 입출력을 지원한다.

실제 시스템에서는 하드웨어 인터럽트인 타이머 인터럽트를 이용하여 시분할 스케줄링을 한다. 시분할 스케줄링을 통하여 멀티태스킹이 이루어지기 때문에, 이것은 RTOS 시뮬레이터에서 반드시 지원되어야 하는 중요한

```
void qsimtest1() {
  char string[30];
  printf("Type a string... \n=> :");
  scanf("%s", string);
  printf("string  - %s and length - %d\n", string, strlen(string));
}
void qsimtest2() {
  int a=0;
  while (1)
    printf("First Loop 1 %d\n", ++a);
}
void qsimtest3() {
  int a=0;
  while (1)
    printf("Second Loop 2 %d\n", ++a);
}
void qsimtest4() {
  int i, sum=0;
  for (i=0; i<1000000; ++i)
    sum += i;
}
void qsimtest5() {
  int i;
  char *tt;
  tt = (char *)malloc(1000000);
  for (i=0; i<1000000; ++i)
    tt[i] = (char) i;
  free(tt);
}
```

그림 14 테스트용 C 프로그램

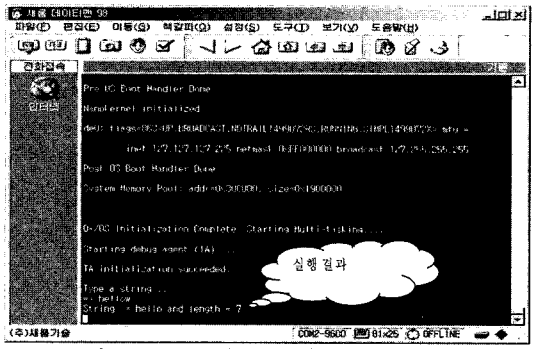


그림 15 Q+Sim의 표준 입출력 시뮬레이션

기능이다. Q+Sim에서는 이 기능을 지원하기 위하여 DTSB 보드와 동일한 간격으로 Q+Sim 코어에 타이머 인터럽트를 발생시킨다. Q+Sim에서 멀티태스킹 기능을 확인하기 위하여 'qsimtest2()' 함수와 'qsimtest3()' 함수를 Q+Esto의 원격 대화형 쉘을 통하여 실행해 보았다. 그림 16의 Q+Sim 실행창은 일정한 간격으로 두 함수가 교대로 실행되는 것을 보여주고 있는데, 이것은 Q+Sim이 타이머 인터럽트를 이용한 멀티태스킹 기능을 지원한다는 것을 나타낸다. 한편 그림 16의 원격 대화형 쉘에서 'ld' 명령은 지정된 실행 모듈을 적재하고, 'i' 명령은 현재 프로세스들을 보여주며, 'sp' 명령은 적재된 실행 모듈에서 지정된 함수를 실행시킨다. 그림 16의 원격 대화형 쉘 창의 실행 결과에서 실행된 테스트용 C 함수

들이 "READY" 상태로 표시되는 것은 원격 대화형 쉘의 명령어를 실행하는 동안에는 이들 프로세서들이 대기상태에 있기 때문이다.

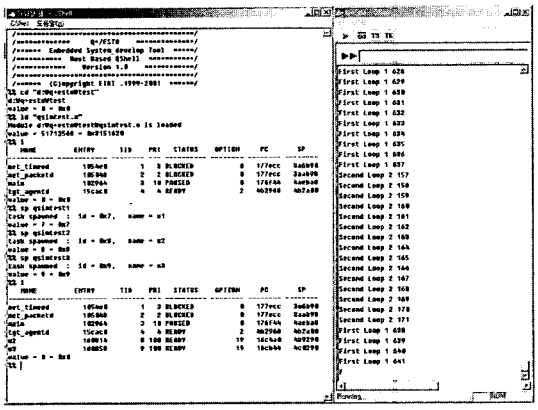


그림 16 타이머 인터럽트를 통한 시분할 스케줄링

표 2는 메모리 액세스가 거의 없는 'qsimtest4()' 함수와 메모리 액세스가 빈번한 'qsimtest5()' 함수의 실행 시간을 Q+Sim을 사용하여 시뮬레이션 한 결과를 보여주고 있다. 실행시간 추정을 위하여 Q+Sim이 수집한 데이타는 실행된 CPU 명령어들의 수, 메모리 액세스 수, 명령어 캐쉬의 적중률, 데이타 캐쉬의 적중률, 그리고 분기 명령어의 수이다. 실제 타겟 하드웨어의 실험 결과를 토대로 하여, 메모리 액세스 및 분기가 발생하지 않는 명령어의 평균 실행시간은 1.2 사이클로 가정하였다. 그리고 메모리 액세스가 발생할 경우에는 6 사이클이 추가되고, 분기가 발생할 경우에는 4 사이클이 추가되는 것으로 가정하였으며, 사이클 시간은 200 ㎒ SA-110 프로세서의 경우에 5 ns이다.

결과에 따르면, 추정 시간과 실제 타겟 하드웨어에서의 실행시간과 추정시간 간의 오차율은 'qsimtest4()' 함수의 경우에는 약 5.5%이고, 'qsimtest5()' 함수의 경우에는 약 16.7%로 나타났다. 표 2에서의 시뮬레이션 시간은 Q+Sim이 2.8 ㎓ 펜티엄-4 PC에서 시뮬레이션을 수행한 시간으로서, 정밀한 기계명령어-레벨의 시뮬레이션임에도 불구하고 비교적 짧은 시뮬레이션 시간이 소요되었다.

6. 결 론

본 연구에서는 현재 상용 RTOS 시뮬레이터들이 가지고 있는 기능적 시뮬레이션의 문제점을 개선하기 위하여, 하드웨어 플랫폼에 제한되지 않고 구현될 수 있으면서 실제 타겟 하드웨어에서의 실질적인 프로그램 실행 시간의 추정도 가능한 RTOS 시뮬레이터를 구현하

표 2 실행시간 추정 결과

	qsimtest4()	qsimtest5()
실행된 명령어 수	4,016,605	20,006,728
메모리 액세스 횟수	984	497,533
명령어 캐쉬 적중률	99%	99%
데이타 캐쉬 적중률	96%	96%
분기 명령어 수	1,002,073	4,001,200
Q+Sim의 추정시간	0.044175 sec	0.214994 sec
타겟 H/W의 실행시간	0.046751 sec	0.258012 sec
시뮬레이션 시간	0.952299 sec	7.546998 sec

는 기법을 개발하였다. 그리고 실제 이 기법을 이용하여 실시간 운영체제인 Q+의 실행 이미지를 시뮬레이션 하는 Q+Sim을 구현하였다.

Q+Sim은 운영체제 및 응용프로그램의 기계명령어들을 해독하고 동작들을 시뮬레이션하기 때문에, 타겟 하드웨어에서의 Q+ 및 응용프로그램의 동작이 호스트에서의 시뮬레이션 모드에서 동일하게 수행될 수 있다. 실제 시뮬레이션을 수행한 결과를 보면, Q+Sim은 실시간 운영체제인 Q+의 각 모듈별 함수들을 정상적으로 수행하였고, Q+Esto의 원격 대화형 쉘과 디버거 등과 같은 도구들과의 연동도 확인할 수 있었다. 그리고 ARM 프로세서의 명령어 실행 파이프라인과 캐쉬의 영향을 고려하여 RTOS의 특정 함수나 응용프로그램의 실행시간을 추정할 수 있었다.

본 연구에서 구현한 Q+Sim은 호스트에서 소프트웨어로 구현된 타겟 하드웨어에 해당하므로, 타겟 하드웨어가 개발되기 전에도 호스트에서 응용프로그램의 개발이 가능하게 해준다. 따라서 Q+Sim은 비교적 고가의 실제 타겟 하드웨어를 구입하기 어려운 일반 사용자들에게 타겟 하드웨어가 없이 호스트 상에서 RTOS를 사용할 수 있게 해주고, 타겟 응용프로그램을 개발할 수 있는 환경을 제공한다. 또한 RTOS를 접하기 어려운 일반 사용자들이나 학생들도 쉽게 RTOS 개발 환경을 경험할 수 있게 해주기 때문에, RTOS 개발 인력의 양성에도 크게 도움이 될 것이다.

참 고 문 헌

[1] 한국전자통신연구원, "조립형 실시간 OS 사용자 요구사항 정의서 1", 1998년 12월.
[2] WindRiver, VxWorks 5.3.1 Programmer's Guide, April, 1997.
[3] Realiant Systems, "Carbon Kernel User Manual 1.2," January, 2000.
[4] http://www.mentor.com/embedded, 2001. 1.
[5] http://www.neiworld.co.kr.
[6] http://www.aromasoft.com.
[7] 한국전자통신연구원, "실시간 OS 커널 상세 설계서

1.0", 1999년 7월.
[8] 한국전자통신연구원, "사용자개발도구 서브시스템 설계서 1.0", 1999년 3월.
[9] Intel, SA-110 Technical Reference Manual, December, 2000.
[10] Intel, EBSA-285 Evaluation Board Reference Manual, October, 1998.
[11] Ron Pluth, Taimur Aslam, "Developing Device Drivers in a Hardware / Software Co-Simulation Environment," The Embedded Systems Conference San Francisco, April, 1998.
[12] ARM, ARM ELF, 2000. 10.
[13] David Seal, ARM Architecture Reference Manual 2th ed., Addison-Wesley, 2000.

김 종 현
1976년 연세대학교 전기공학과(학사) 1981년 연세대학교 전기공학과(석사) 1988년 Arizona State University 전기 및 컴퓨터공학과(박사). 1976년~1982년 국방과학연구소 연구원. 1988년~1990년 한국전자통신연구소 실장 역임. 1990년~현재 연세대학교 문리대학 컴퓨터공학과 교수. 1996년 Oregon State University 전산학과 방문교수. 2003년 Florida State University 전산학과 방문교수. 관심분야는 컴퓨터구조, 병렬처리, 유비쿼터스컴퓨팅, 시뮬레이션 등

김 방 현
1996년 연세대학교 전산학과(학사). 2001년 연세대학교 전산학과(석사). 2001년~현재 연세대학교 전산학과 박사과정. 관심분야는 임베디드시스템, 유비쿼터스컴퓨팅, 병렬처리, 시뮬레이션, 정보보안 등

이 광 용
1991년 숭실대학교 전자계산학과(학사) 1993년 숭실대학교 전자계산학과(공학석사). 1997년 숭실대학교 전자계산학과(공학박사). 1996년~1998년 숭실대학교 생산기술연구소 연구원. 1997년~1998년 ETRI 컴퓨터소프트웨어기술연구소. 소프트웨어공학연구부 박사후연수연구원(Post-Doc.). 1999년~현재 ETRI 임베디드S/W연구단 편재형컴퓨팅미들웨어연구팀 선임연구원. 관심분야는 유비쿼터스컴퓨팅, 나노운영체제, 무선센서네트워크(WSN), 임베디드시스템, 실시간 시뮬레이터 및 디버거, 정형기법, 실시간 객체지향 모델링, 소프트웨어공학