모바일 응용을 위한 자바 하드웨어 가속기의 설계

최병윤* · 박영수**

Design of Java Hardware Accelerator for Mobile Application

Byeong-yoon Choi* · Young-soo Park**

이 논문은 한국전자통신연구원 위탁 과제와 BB21 연구비 지원을 받았음

요 약

자바 가상 기계는 모바일 및 내장형 제어 디바이스와 같은 소형 디바이스에 적합한 간결한 코드, 단순한 수행 동작, 플랫폼 독립성의 특성을 제공하지만, 스택 기반 동작에 기인한 낮은 연산 효율이라는 큰 문제점을 갖고 있다. 본 논문에서는 이러한 낮은 동작 속도 문제를 제거하여, 모바일 및 내장형 제어 분야용 자바 가속기를 설계하였다. 설계된 자바 가속기는 자바 가상머신 명령어 코드 중 81개를 구현하며, 효율적인 보조 프로세서 인터페이스와 명령어 버퍼를 사용하여 기존 32-비트 RISC 프로세서에 자바 보조 프로세서로 활용될 수 있도록 하였다. 자바 가속기는 14,300개의 게이트로 구성되며, 0.35um CMOS 공정 조건에서 약 50 Mhz의 동작 주파수를 갖는다.

ABSTRACT

Java virtual machine provides code compactness, simple execution engines, and platform-independence which are important features for small devices such as mobile or embedded device, but it has a big problem, such as low throughput due to stack-oriented operation. In this paper hardware Java accelerator targeted for mobile or embedded application is designed to eliminate the slow speed problem of Java virtual machine. The designed Java accelerator can execute 81 instructions of Java virtual machine(JVM)'s opcodes and be used as Java coprocessor of conventional 32-bit RISC processor with efficient coprocessor interface and instruction buffer. It consists of about 14,300 gates and its maximum operating frequency is about 50 Mhz under 0.35um CMOS technology.

키워드

자바 프로세서, 자바 가속기, 스택 머신, 모바일 디바이스, 자바 가상 기계, 자바 카드

I. 서 론

스택 컴퓨터는 모바일 또는 내장형 응용, 스마트 카드 분야와 같은 작은 면적을 필요로 응용에 적합한 코드 간결성과 단순한 수행 엔진을 제공하는 것으로 알려져 있다[1]. 인터넷이 보편화됨에 따라 컴퓨터 기종에 무관하게 동작할 수 있는 언어의 필요성이 부각됨에 따라 도입된 자바 언어는, "Write

*동의대학교 컴퓨터 공학과
**한국 전자통신 연구원(ETRI)
접수일자 : 2004. 5. 12

Once, Run Anywhere"라는 모토를 내걸고 있으며, 기계 독립성을 만족하기 위해 인터프리터형 언어로 되어 있으며, 스택 기반의 연산을 활용해서 코드 크기를 최소화하는 특성을 갖고 있다. 자바 언어는 기존 인터프리터 언어의 낮은 동작 속도 문제를 고려하여, 중간 코드 형태인 클래스 파일(class file)을 사용하고, 가상 컴퓨터인 자바 가상 머신(Java Virtual Machine)을 도입하여, 기계 독립성 보장과 속도 문제를 개선하고자 하였다[2]. 그러나 이러한 접근 방식에도 불구하고 기존 C언어와 같은 컴파일 형 언어에 비해 동작 속도가 수십 배 이상 떨어지는 특성을 갖고 있어서, 고속의 동작이 필요한 분야에 자바의 응용을 제약하고 있다. 즉 자바 언어를 컴파일 하여 수행 컴퓨터에 맞게 기계어 코드를 생성하여 수행할 수도 있지만, 이 경우 자바 언어의 기계독립성이 없어지는 문제가 존재한다. 따라서 자바 언어의 기계 독립성 특성을 유지하면서 소프트웨어와 하드웨어 측면에서 자바 클래스 파일을 고속으로 처리하는 방안을 연구하고 있다.

자바 언어를 개발한 선 마이크로 시스템에서는 자바소프트라는 자 회사를 만들어 다양한 응용 분야에 자바가 이용될 수 있도록 지원하고 있다. 즉 응용별로 지원하는 자바 API(Application Programming Interface)를 달리함으로써 개발용도에 알맞은 JAE(Java Application Environment)를 선택할 수 있도록 하였다. 자바 가상 머신은 자바 프로그램의 바이트 코드를 수행하는 '소프트' 컴퓨터로서 JAE 핵심요소로써 존재한다. 이러한 자바 가상 머신은 자바 가상 머신 명세에 의해 정의된 명령어의 집합을 포함하는 클래스 파일을 수행하는 기능을 제공해야 한다. 자바 가상 머신은 자바 가상 머신 명세 범위 안에서 여러 가지 방안으로 구현될 수 있다. 즉 선 마이크로시스템 사는 응용 분야별로 요구하는 자바 코드 및 시스템 사양이 다른 점을 반영하여 자바의 환경을 J2SE, J2EE, J2ME로 3등분하고 있다[3]. 또한 이와 별도로 개발된 자바 카드 표준인 JCVM(Java Card Virtual Machine)은 기존 J2ME보다 면적 조건이 엄격해서, 기존 JVM(Java Virtual Machine)이 32 비트 구조를 근간으로 하고 있는데 비해, JCVM은 16 비트 구조를 기본으로 하고 32 비트 구조를 선택사양으로 갖고 있다[4].

최근 인터넷 환경, 자바 기반 내장형 제어 분야와 자바 카드 분야에서, 기계 독립성을 만족하며 Java byte 코드의 고속 처리를 요구하는 응용 분야가 증가함에 따라, Java byte 코드를 하드웨어로 처리하는 방안이 활발히 연구되고 있다. 특히 JVM을 소프트웨어로 구현하는 가장 최신 방안인 Just In Time (JIT) 컴파일과 Hotspot 기술은 커다란 성능 향상을 보여, 데스크 탑 컴퓨터 환경에서는 하드웨어 방식의 JVM 구현 필요성을 많은 부분 제거하고 있다[5]. 그러나 이러한 방식은 많은 메모리가 필요하기 때문에, 모바일 환경, 혹은 스마트카드 분야에 적용하는데 문제가 있다. 즉, 모바일 또는 스마트카드 분야에서는 적절한 연산 능력을 갖고 있으며 면적이 최소화된 하드웨어 자바 가속기가 필요하다. 선 사의 경우 자바 바이트 코드를 직접적으로 처리하는 IP(Intellectual Property) 기반의 PicoJava 계열의 프로세서를 개발하여 라이센스를 받고 코드를 제공하고 있다[6]. 반면 Nazomi, ARM 등의 회사는 하드웨어 인터프리터 방식으로 Java Byte Code의 일부를 동작 수행과정에 온라인(on-the-fly) 방식으로 내부 CPU의 기계어 코드로 변환하는 방식의 자바 가속기(Java accelerator)를 IP 형태로 개발하고 있다[7-9]. 본 논문에서는 J2ME과 자바 카드 가상 머신 JCVM 2.2의 사양 분석을 바탕으로 32 비트 모바일 환경 과 32 비트 자바 카드 분야에 응용될 수 있는 자바 하드웨어 가속기를 설계하고 성능을 분석하였다.

본 논문의 구성은 다음과 같다. 제 2장에서는 JVM의 동작 특성과 JVM의 하드웨어-소프트웨어 공동 설계 방안을 분석하였다. 제 3 장에서는 모바일 환경에 적합한 자바 가속기의 하드웨어 설계를 기술한다. 제 4장에서는 설계된 회로에 대한 검증과 성능 분석을 기술하였으며, 마지막으로 결론 및 향후 연구 방향을 기술하였다.

II. 자바 가상 머신(JVM)의 하드웨어-소프트웨어 공동 설계 기법

자바 언어는 기존 인터프리터 언어의 낮은 동작 속도 문제를 고려하여, 중간 코드 형태인 클래스 파일(class file)을 사용하고, 가상 컴퓨터인 자바 가상 머신(Java Virtual Machine)을 도입하여, 기계 독립성 보장과 기존 인터프리터형 언어의 속도 문제를 개선하고자 하였다. 그러나 자바 언어의 이러한 특성은 JVM 명령을 하드웨어로 구현하는 것을 어렵게 하고 있다. 특히 JVM 명령중 메소드 호출(method invocation), 상수 풀(constant pool) 참조 명령, 클래스와 객체 처리 명령, 테이블 참조에 의한 분기와 같은 가변 길이 명령 등은 하드웨어로 처리하려면 수십 혹은 수백 클록 사이클이 소요된다. 따라서 기존에 소프트웨어만으로 구현되었던

JVM의 성능을 향상시키는 방안은, JVM을 구성하는 명령 중에서 성능에 큰 영향을 미치거나 발생빈도가 높은 대부분의 명령을 하드웨어로 구현하고, 하드웨어로 구현하기 힘들며 성능에 큰 영향을 미치지 않은 일부 명령은 호스트 프로세서에서 에뮬레이션 기법으로 처리하는 그림 1과 같은 하드웨어-소프트웨어 공동 설계(Hardware-Software Codesign) 기법의 구현 형태가 바람직하다.

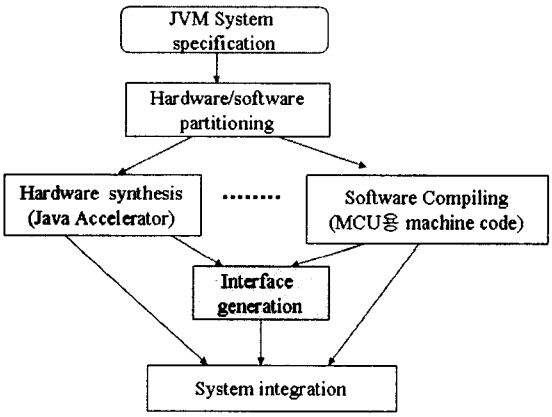

그림 1. JVM의 하드웨어-소프트웨어 공동 설계 기법
Fig. 1. Hardware-software codesign of JVM

III. 자바 가속기 설계

본 장에서는 자바 가속기의 설계 사양, 호스트 프로세서와 자바 가속기간의 인터페이스를 정의하고, 이를 바탕으로 하드웨어로 구현된 자바 가속기를 기술하였다.

1. 설계 사양

본 연구에서 작은 하드웨어 면적을 필요로 하는 모바일 과 스마트카드 응용에 적합한 자바 가속기 내부 구조를 결정하면서 고려한 사항은 다음과 같다.

첫째, 본 연구에서는 공유 메모리 인터페이스 방식을 갖는 자바 가속기 구조를 채택하였다. 이러한 인터페이스를 방식을 채택한 이유는 호스트 프로세서의 형에 관계없이 모든 프로세서에 적용할 수 있다는 장점과 함께, 호스트의 성능에 의존하지 않고 자바 가속기 구조를 최적화할 수 있다는 점을 고려하였다.

둘째, 내부 아키텍처로 32 비트 구조를 선택하였다. 모바일 응용을 위한 KVM의 경우 32 비트 아키텍처를 사용하고, 자바 카드용 JCVM경우는 16 비트 아키텍처를 기본 모델로 사용하고 32 비트 아키텍처는 선택 사양으로 하고 있는데, 2가지 분야의 공통적인 응용을 고려하여 32 비트 내부 구조를 채택하였다.

셋째, 호스트 프로세서와 자바 가속기는 병렬로 동작하지는 않는 방식을 채택하였다. 호스트 프로세서와 자바 명령어 집합이 별도로 존재하고, 호스트 프로세서가 자바 명령어를 직접적으로 인식하여 수행할 수 없기 때문에, 호스트 프로세서와 자바 가속기간의 동작은 상호 배타적(mutual exclusion) 방식으로 동작한다.

넷째, 모바일 및 내장형 제어 응용 용도의 자바 가속기는 성능과 함께 면적이 매우 중요하므로, 본 연구에서는 파이프라인 구조와 스택 캐쉬 기반의 명령어 폴딩 기법을 사용치 않았으며[10-11], 단일 쓰레드(thread)만 지원하는 구조를 채택하였다.

2. 하드웨어로 구현된 자바 명령어

자바 가상 머신에 정의된 명령어 코드는 바이트 길이의 배수로 되어 있으며, 그림 2와 같이 스택 기반의 동작을 수행하므로, 명령 코드 내에 오퍼랜드 주소 필드가 없기 때문에 평균 코드 길이가 1.8 바이트로 기존 RISC나 CISC 프로세서에 비해 훨씬 작다. 그러나 스택 동작은 많은 메모리 접근 동작과 인접한 명령어간의 데이터 의존성, 가변 길이 특성으로 하드웨어로 구현시 성능 향상을 어렵게 만든다.

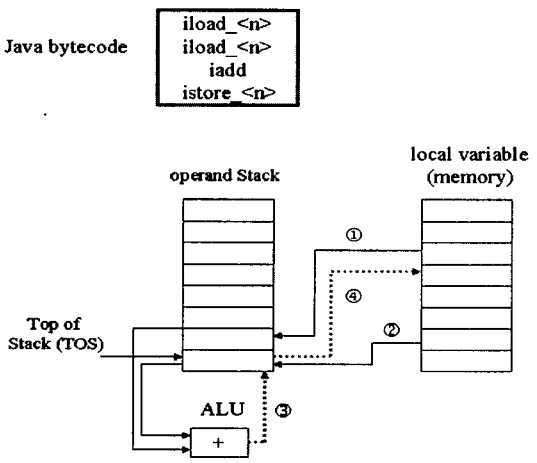

그림 2. 자바 바이트 코드의 스택 기반 동작
Fig. 2. Stack-based operation of Java bytecode

그림 2의 iload_<n> 명령의 경우 지역 변수를

오퍼랜드 스택에 삽입하고(push), iadd 명령은 오퍼랜드 스택의 상위 2개의 오퍼랜드를 인출하여 덧셈 연산을 수행한 후 스택의 최상위 위치에 결과를 저장한다(2 pops, 1 push 동작). 그리고 마지막으로 istore_<n> 명령은 메모리 내 대응하는 인덱스에 저장한다. JVM은 각 쓰레드(thread)와 관련된 스택(stack)을 가지고 있으며, 스택에는 JVM 프레임(frame)을 유지한다. JVM 프레임은 자바 메소드가 호출될 때 생성되고, 메소드가 수행 완료되는 경우 소멸된다. 각 메소드 프레임은 지역 변수(local variables) 영역, 오퍼랜드 스택(operand stack) 영역, 프레임의 상태를 유지하는 스택 프레임(stack frame) 영역으로 구성된다. 본 논문에서 설계한 자바 가속기가 하드웨어로 구현한 자바 가상 머신 명령어는 표 1에 나타낸 81 개이다. 자바 가속기에서 지원되지 않은 명령어를 제어 회로가 감지하는 경우, 마이크로프로그램 메모리 내 Unimp 마이크로프로그램 루틴으로 매핑되어 호스트 프로세서에게 인터럽트를 발생하도록 하였다. 호스트 프로세서는 자바 가속기에서 오는 인터럽트 감지시 공유 메모리를 통해 전달된 예외 야기 명령어 코드를 분석해서 적절한 에뮬레이션으로 해당 명령을 처리하도록 하였다.

표 1. 자바 가속기로 구현된 명령어
Table 1. Instructions implemented with Java accelerator

명령 유형	명 령
자바 가속기 내에서 하드웨어로 구현되는 명령	aaload, iaload, aastore, iastore, aconst_null, iconst_0, iconst_m1, iconst_1, iconst_2, iconst_3, iconst_4, iconst_5, aload, iload, aload_0, aload_1, aload_2, aload_3, iload_0, iload_1, iload_2, iload_3,..., astore, istore, astore_0, astore_1, astore_2, astore_3, istore_0, istore_1, istore_2, istore_3, newarray, arraylength, pop, pop2, dup, dup2, dup_x1, nop, bipush, sipush, iadd, isub, ineg, iand, ixor, ior, ishl, ishr, iushr, ifnull, ifnonnull, ifeq, ifne, iflt, ifge, ifgt, ifle, if_acmpeq, if_acmpne, if_icmpeq, if_icmpne, if_icmplt, if_icmpge, if_icmpgt, if_icmple, goto, iinc, jsr, ret, swap, getstatic, putstatic, getfield, putfield, invokevirtual, invokestatic, return, tableswitch, breakpoint

3. 자바 가속기와 호스트 프로세서 간 인터페이스

본 연구에서 설계 중인 자바 가속기는 공유메모리를 통한 그림 3과 같은 정보 전달 방식을 갖는 보조 프로세서 구조 형태를 갖고 있다. 그리고 외부 메모리 접근을 위해 32 비트 데이터 버스와 32 비트 어드레스 버스를 갖는 구조를 갖는다.

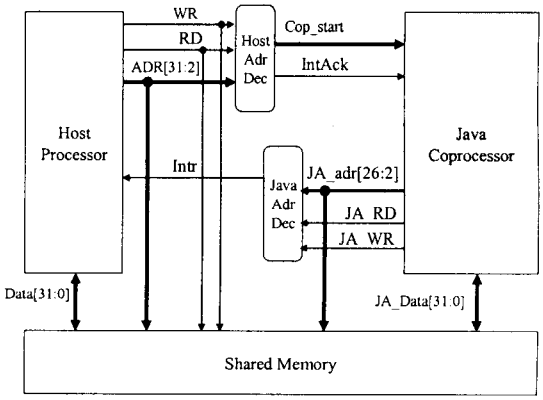

그림 3. 자바 가속기와 32 비트 호스트 프로세서 간 인터페이스
Fig. 3. Interface between 32-bit host processor and Java coprocessor

본 연구의 자바 가속기와 호스트 프로세서는 동시에 활성화 되지 않으며(mutual exclusion), 호스트 프로세서가 동작하는 동안 자바 가속기는 모든 출력 신호를 고 저항(high-impedance) 상태로 둔다. 그리고 자바 가속기와 호스트 프로세서간의 동작 개시(COP_start)와 종료를 지시하는 신호는 메모리-맵된(memory-mapped) 제어 신호 형태로 발생되며, 바이트 코드의 시작 주소와 같이 자바 가속기가 동작하는데 필요한 정보는 공유 메모리을 통해 전달된다. 그림 4와 그림 5는 각각 호스트 프로세서와 자바 가속기 사이의 인터페이스 프로토콜 동작과 프로세서 간에 정보 전달과 인터페이스용 제어 신호를 생성하기 위해 할당된 공유 메모리의 주소 할당을 나타낸다.

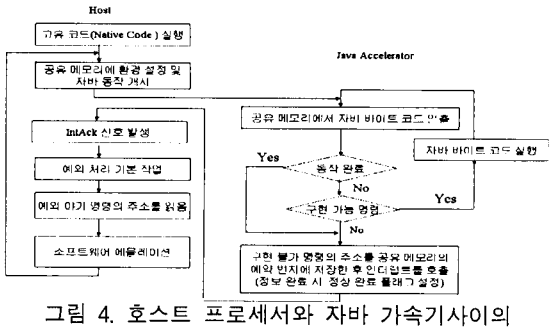

그림 4. 호스트 프로세서와 자바 가속기사이의 인터페이스 프로토콜
Fig. 4. Interface protocol between host processor and Java accelerator

호스트 프로세서는 자신의 기계어 코드(native code) 부분을 수행하다가 자바 바이트 코드 처리가 필요할 경우, 주소 값 (0x00100000)번지에서 시작되는 클래스 파일 초기화 영역에, 자바 가속기가 자바 바이트 코드를 수행하는데 필요한 정보, 즉 메소드 테이블(method table)의 시작 주소(mbase), 스택 포인터 주소(SP), 상수 풀(constant pool)의 시작 주소(Cpool), 힙 영역의 시작 주소(Heap) 등을 저장한 후, 주소 값 0xf0000000번지에 대한 메모리 접근 동작을 수행하여 자바 가속기의 동작을 개시시킨다. 자바 가속기는 대기 상태에서 동작 개시 신호(COP_start)를 감지할 경우, 자바 코드 수행에 필요한 상태 정보(mbase, Cpool, SP, Heap) 주소를 읽어와 내부 레지스터 파일과 스택 포인터에 저장한다. 이러한 상태 정보는 범용 프로세서의 부팅 과정에 설정되는 레지스터 설정과 유사하다. 이러한 자바 가상 머신 관련 주소 포인터는 사용 빈도수가 높기 때문에 외부 공유 메모리에 두는 것 보다는 내부 레지스터 파일에 두는 것이 연산 속도를 향상시킬 수 있으며, 마이크로프로그램의 크기를 감소시킬 수 있다. JVM 사양에서는 레지스터 파일이 존재하지 않지만 본 연구에서는 마이크로프로그램으로 JVM 명령을 처리할 때, 복잡한 JVM 명령의 중간 결과를 저장하기 위한 임시 기억 장소 역할과 사용빈도가 높은 시스템 제어 주소 포인터를 저장하기 위해 16개의 32 비트(16 × 32-bit) 레지스터 파일을 채택하였다. 그리고 자바 가속기가 처리해야 할 명령어의 주소는 메소드 테이블(method table)의 시작 주소(mbase)를 바탕으로 바이트 코드가 놓인 시작 주소를 얻어 자바 가속기의 PC(program counter)를 설정한 후 자바 바이트 코드 수행 동작을 시작한다. Comm_info로 정의된 주소 값 (0x00c00000)번지에는 자바 가속기가 연산 수행 후에 호스트 프로세서에 전달하는 정보가 담긴다. 정상적인 종료인 경우는 0 값이 담기지만, 자바 가속기가 수행할 수 없는 명령어를 감지하는 경우 해당 명령의 PC(Program Counter) 값이 담겨서 호스트 프로세서가 해당 명령을 에뮬레이션을 통해 소프트웨어적으로 등가 구현할 수 있도록 하였다. 자바 가속기가 동작을 마친 경우 호스트 프로세서에게 인터럽트로 이를 알리게 된다. 이러한 인터럽트 신호를 생성하기 위해 자바 가속기는 예약된 주소 값(0x00f00000)번지에 대한 메모리 읽기 동작을 수행하며, 호스트 프로세서는 인터럽트를 인식하여 대기 상태에서 벗어나고, 자바 가속기에게 IntAck 신호(주소 값 =0xd0000000)를 제공한다.

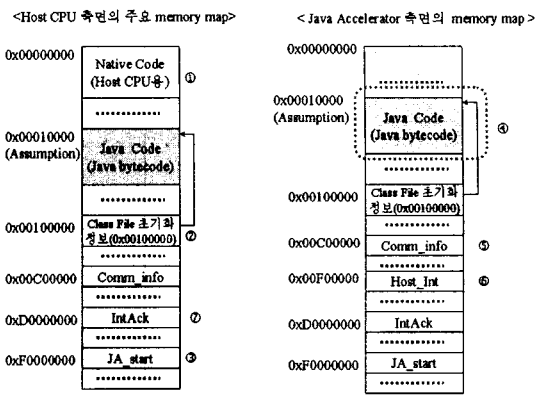

그림 5. 정보 전달과 제어 신호 생성을 위한 메모리 맵
Fig. 5. Memory map for parameter passing and control signals

4. 자바 가속기의 하드웨어 설계

표 1의 81개 명령과 4.3절에서 기술한 호스트 프로세서와의 인터페이스 동작을 만족하도록 설계된 자바 가속기의 전체 블록도는 그림 6과 같다.

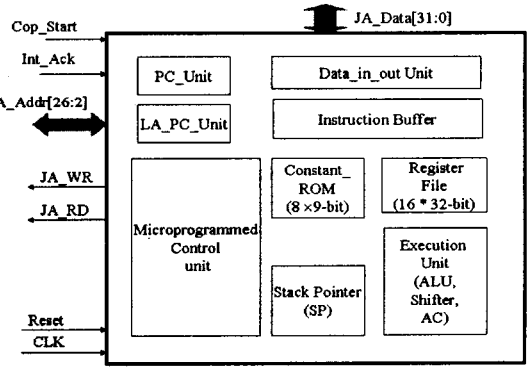

그림 6. 자바 가속기의 블록도
Fig. 6. Block diagram of Java accelerator

자바 가속기는 수행중인 바이트 코드의 주소를 저장하는 프로그램 카운터(PC_unit), 명령어 버퍼부(Instruction Buffer), 외부에서 명령어를 4 바이트 단위로 명령어를 인출할 주소를 담고 있는 LA_PC (Lookahead PC 부) 부, ALU와 배럴 시프터로 구성된 실행부(Execution unit), 데이터 입출력 부, 스택 포인터 레지스터, 상수 ROM부, 연산 중간 결과와 시스템 포인터 정보를 담고 있는 레지스터 파일 부로 구성된다. 단, JVM을 구현하는 마이크로프로그램의 개수를 감소시키기 위해 ALU와 배럴 시프터를 직렬로 연결하는 방식을 사용하였

다. 그리고 본 연구의 자바 가속기는 스택 캐쉬 메모리를 내장하지 않기 때문에 기존 RISC와 같은 이중 읽기(dual read) 포트 구조를 가진 레지스터 파일 대신에 단일 포트를 갖는 레지스터 파일을 사용하였다. 단, 레지스터 파일이 단일 포트인 단점을 개선하기 위해, 최근 사용된 오퍼랜드를 저장하는 AC(accumulator)를 두어, 준 이중(pseudo dual port) 포트의 성능을 낼 수 있도록 하였다. 마지막 블록으로 마이크로프로그램 제어 기법을 갖는 제어 회로가 있다. 본 연구에서 제어 기법으로 고정 배선 제어(hardwired-control) 기법을 사용하지 않은 이유는 invokevirtual과 tableswitch 명령과 같은 일부 명령은 너무 복잡하고, 구현된 명령어 81개의 명령에 대해 차후 새로운 명령어를 하드웨어로 구현할 가능성이 많은데, 이 경우 마이크로프로그램 제어 방식이 마이크로프로그램 수정으로 쉽게 확장이 용이하기 때문에 이 방식을 채택하였다.

그림 7은 자바 가속기 내부 데이터 패스부의 블록도를 나타낸다.

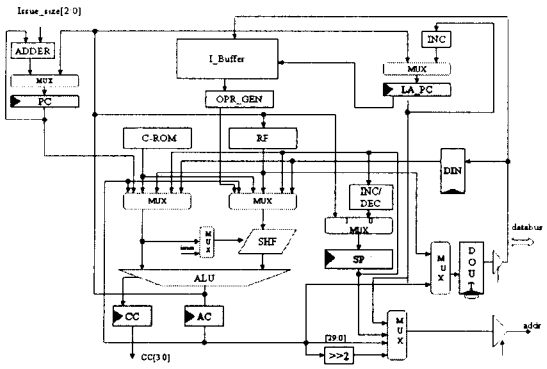

그림 7. 자바 가속기의 datapath 부
Fig. 7. Datapath unit of Java accelerator

JVM 명령어중 오퍼랜드 스택에 저장하는 명령어(iconst_m1, iconst_0, iconst_2, iconst_3, iconst_4, iconst_5)과 마이크로프로그램 작업 중 다양한 주소 계산을 위해 많은 상수 처리 동작이 존재한다. 이러한 동작을 효율적으로 구현하기 위해서는 다양한 상수를 저장하는 ROM(Read-Only Memory)가 필요하다. 그러나 필요한 모든 상수를 저장하는 ROM을 하드웨어로 구현하는 것은 많은 면적이 필요하므로, 본 연구에서는 면적과 연산 효율성을 절충하여 8개의 상수만을 ROM에 저장하는 방식을 채택하였다. 자바 명령어가 바이트 단위로 처리하는 기능이 많은 점을 고려하여, 7개의 기본적인 상수(-1, 0, 1, 2, 3, 4, 5)외에 255(011111111)를 저장하는 8 × 9-bit 구조를 갖는다. 여기서 8 비트 출력이 아닌 9 비트 값을 사용한 이유는, ROM 값의 부호 확장시 바이트 단위의 마스킹을 위한 상수 255와 "-1"을 구별하기 위해 9 비트 구조를 사용하였다. 단, 자바 데이터 패스가 32 비트인 것을 고려하여, 상수 ROM의 출력은 32 비트로 부호 확장한 형태로 출력된다. 상수 롬에 저장되어 있지 않은 상수가 마이크로프로그램 동작 중에 필요할 경우는 기존 상수 값을 시프트시켜 필요한 상수 값을 생성하여 사용하는 기법을 채택하였다. 예를 들면 0x00000c400의 경우 (3<< 14 + 4<<8)와 같은 연산 기법을 통해 구현한다.

OPR_GEN 블록은 명령어 버퍼에 존재하는 자바 가상 머신 명령을 분석하여, 명령어 코드 뒤에 오는 8/16-비트의 부호 및 무부호 즉치(immediate data)를 부호 확장(sign extension), 무부호 확장(zero extension)시켜, ALU에 32 비트 형태로 데이터를 제공하는 역할을 수행한다. 다른 시스템 관련 포인터는 레지스터 파일에 저장하는데 비해, 오퍼랜드 스택에 대한 스택 포인터(SP)는 별도의 독립적인 하드웨어로 구현한 이유는, 다른 시스템 관련 포인터와 달리 대부분의 자바 명령 수행시 오퍼랜드 스택이 영향을 받기 때문에 성능 향상을 위해 별도의 독립된 레지스터로 구현하였다. 스택 포인터는 1씩 증가 및 감소 기능을 담당하는 INC/DEC부와 30비트 레지스터로 구성된다. 단, 1 이상의 증가 및 감소 동작이 필요할 경우는 증가 또는 감소 동작을 반복 실행하여 구현된다. 1보다 큰 증감 동작의 빈도가 크지 않기 때문에 이러한 기법을 사용하였다. DIN과 DOUT 레지스터는 입출력 데이터 값을 저장하는 레지스터이며, AR 레지스터는 외부 메모리 접근시 주소 값을 제공하는 레지스터이다. 그리고 실행부를 구성하는 ALU와 배럴 시프터(SHF)는 ARM RISC 프로세서와 같이 시프터와 ALU를 직렬로 연결하여, 시프트 동작과 산술 연산이 하나의 동작으로 처리가 가능하도록 하여 마이크로프로그램의 코드 길이를 줄일 수 있도록 하였다. ALU는 4비트 단위로 그룹화된 블록 캐리 룩 어헤드 가산기에 기반을 둔 구조를 갖고 있으며, 8가지 산술 및 논리 연산을 지원한다. 배럴 시스터는 퍼널 시스터(funnel shifter) 구조를 취하고 있으며, LSL(logical shift left), ASR(arithmetic shift right), LSR(logical shift right) 동작을 지원한다. 데이타 패스에 사용되는 대부분의 연산 모듈은 기존 RISC 마이크로프로세서에서 사용되는 구조를 갖는다. 상용 RISC 마이크로프로세서와 달리 본 자바 가속기에 특징적인 데이터 패스 블록은 가변 길이의 특성을 갖는 자바 명령어의 인출 동작을 제

어하는 명령어 버퍼와 프로그램 카운터(PC, LA_PC) 블록이다. 자바 가상 머신의 명령어 코드는 가변 길이 특성을 갖는 바이트 코드이므로 바이트 단위의 입출력 구조가 하드웨어 구현시 용이하지만, 본 연구에서 설계된 자바 가속기는 32 비트 범용 프로세서의 보조 프로세서로 동작해야 하므로 32비트 데이터 버스에 맞게 동작하도록 설계되었다.

본 연구에서 명령어 버퍼 설계 시 고려한 사항은 다음과 같다.

첫째, 자바 명령의 평균 길이는 1.8 바이트이지만, 호스트 프로세서가 32 비트 구조이므로, 명령어 인출에 사용하는 데이터 버스의 폭은 32 비트 구조를 사용하였다. 따라서 한번에 4 바이트의 명령어를 인출할 수 있다.

둘째, 분기 동작이 발생할 경우 앞서 인출되어 주소 값이 워드 정렬(주소 값의 하위 2비트, EA[1:0]=00)이 되어 있지 않을 수 있기 때문에, 4 바이트 보다 작은 바이트 수의 명령어가 인출될 수 있다. 이러한 점을 고려하여 인출된 명령어 개수를 고려하여 명령어 버퍼에 존재하는 타당한 명령어 개수를 저장하는 Entry_Num 레지스터가 존재한다.

셋째, 분기(branch) 동작 발생시 명령어 버퍼에 이전에 담긴 명령어는 모두 취소(flush)되고 새로 인출된 명령어로 채워진다. 또한, 명령어 버퍼는 항상 타당한 명령어가 맨 왼쪽에 놓이도록 함에 의해, 제어 회로가 타당한 명령어의 시작 위치를 찾는데 부가의 작업이 필요하지 않도록 하였다.

넷째, 명령어 버퍼에 외부 메모리에서 인출한 명령어가 놓일 수 있는 조건은 명령어 버퍼에 4개 이상의 빈 바이트가 존재하는 경우로 제한하였다.

다섯째, Tableswitch 명령과 같이 명령어 길이가 고정되어 있지 않은 명령은 명령어 디코더에서 1 바이트 길이 명령으로 간주한다. 이러한 명령어의 실제 길이 계산은 마이크로프로그램으로 구현되며, 명령어의 정확한 길이가 계산된 후 "PC + 명령어 길이"를 계산하여 PC와 LA_PC를 새로운 PC 값으로 대체하는 기법을 사용하였다.

여섯째, 명령어 큐(IB)는 버퍼의 활용도를 고려하여 7개의 바이트 엔트리로 구성하였다..

이러한 명령어 버퍼 설계 사양에 따라 명령어 버퍼가 동작하는 조건은 크게 4가지 경우로 나뉘어 진다.

① 분기 동작인 경우
- 외부에서 인출된 명령어가 명령어 버퍼에 맨 왼쪽에 맞추어 저장되며, 기존 버퍼에 담긴 내용은 취소됨.

② 명령어 버퍼에 빈 공간이 3개 이하인 경우
- 외부에서 새로운 명령이 인출되지 못하며, 명령어 버퍼가 이전에 수행된 명령어 길이만큼 왼쪽으로 이동함

③ 명령어 버퍼에 빈 공간이 4개 이상이 되는 경우
- 외부에서 인출된 4 바이트 명령과 명령어 버퍼에 담겨 있던 명령이 함께 통합되어 왼쪽으로 이동됨

④ 명령어 버퍼에 담긴 내용이 전체 명령어의 일부분인 경우
- 명령어는 처리되지 못했기 때문에 명령어 버퍼는 이동되지 않고, 외부에서 인출된 명령어만 명령어 버퍼의 빈 공간에 담김

위에서 기술한 4가지 동작 조건을 구현하기 위한 명령어 버퍼의 블록도는 그림 8과 같다. 명령어 버퍼는 위에서 기술한 ②의 조건을 구현하는 Left_Aligner_1, 조건 ①, ③, ④를 구현하기 위한 Left_Algner_4, Left_Aligner_2, Merger와 명령어를 담고 있는 명령어 큐(IB) 레지스터, 명령어 버퍼 내에 담긴 명령어 개수 정보와 명령어 버퍼 제어 정보를 제공하는 IB_con블록, 그리고 제어 회로로 제공되는 수행 명령어의 바이트 길이 정보를 생성하는 Len_Gen회로로 구성된다. Left_aligner_1 블록은 명령어 버퍼가 새로운 명령어를 메모리에서 받아들이지 않고, 명령어 버퍼 내부적인 좌측 이동 동작을 수행하는 경우에만 사용하는 왼쪽 정렬기이고. Merger는 명령어 버퍼내의 데이터와 외부 메모리에서 인출한 명령어를 통합하는 기능을 수행하며, Left_aligner_2는 Merger 블록에서 통합된 명령어를 명령어 버퍼에 담겨 있다가 수행된 명령어의 길이(issue_size[2:0])를 바탕으로 통합된 명령어를 이동시키는 기능을 수행한다. 명령어 큐(IB[6] ~ IB[0])는 7 바이트의 명령어로 구성된다.

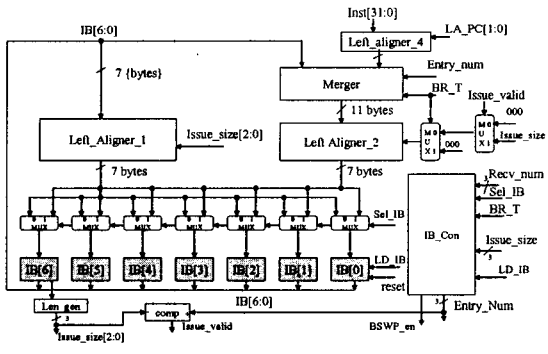

그림 8. 명령어 버퍼 회로
Fig. 8. Instruction buffer

그림 9의 IB_con 블록은 명령어 버퍼를 제어하는 제어 신호를 생성한다. 새로운 명령어를 외부에서 인출할 조건을 나타내는 BSWP_en신호는 식 (1)과 같은 조건으로 판단된다.

$$CurrentEntryNmber[2:0] - issuesize[2:0] <= 3 \quad (1)$$

본 연구의 자바 가속기는 2개의 명령어 주소 레지스터(PC, LA_PC)를 갖고 있다. 이와 같이 2개의 주소 레지스터를 사용한 이유는 자바 가속기가 32비트 데이터 버스를 갖는 명령어 버퍼를 갖고 있기 때문에 채택한 방식이다. 즉 PC(Program Counter) 회로는 현재 자바 가속기의 실행부에서 수행되고 있는 명령어의 주소를 유지하는 역할을 한다. 그리고 명령어 완료 시점에 현재의 PC값과 실행 완료된 명령어의 길이(issue size)를 더해 다음 명령어의 주소를 가리키게 된다. 반면 LA_PC(Lookahead _PC) 회로는 외부 메모리에서 명령어를 인출하기 위한 주소 레지스터로, 명령어를 워드 정렬된 4 바이트 단위로 인출하므로 하위 2 비트는 명령어 인출시 외부 메모리에 제공되지 않는다. 그러나 분기 명령어의 목적지 주소가 워드 정렬(하위 2 비트, EA1:0]=0)이 되어 있지 않을 경우, 주소의 하위 2비트는 인출된 4 바이트 명령어에서 타당한 분기 목적지 명령어가 명령어 큐내 맨 앞으로 배치될 수 있도록 제어하는 정보로 활용된다. 즉 주소의 하위 2 비트 정보는 Recv_Num[2:0]으로 인코딩되어 명령어 버퍼 제어 신호를 생성하는 IB_con에 제공된다.

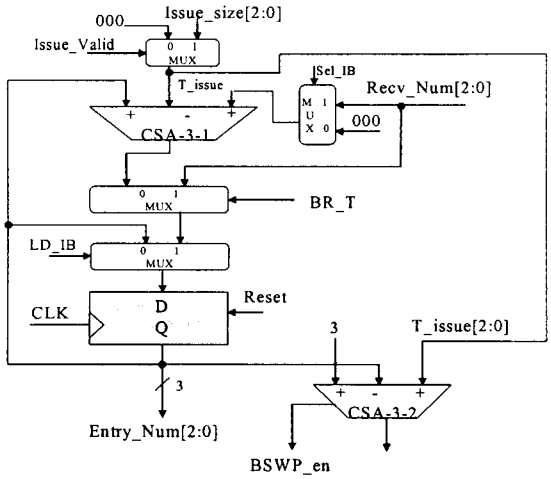

그림 9. IB_Con 회로도
Fig. 9. IB_con circuit

자바 가속기에 대한 제어 회로는 JVM 명령어의 복잡한 명령어의 효율적인 처리와 명령어 추가 용이성, 융통성 등을 고려하여, 기존 RISC 마이크로프로세서에서 널리 사용되고 있는 고정 배선 제어 방식(Hardwired Control) 기법 대신에 마이크로프로그래밍(microprogramming) 기법을 사용했으며, 제어 신호의 빠른 발생을 위해 마이크로 인스트럭션 형식으로 Horizontal Encoding 방식을 사용하였다. 제어 회로는 크게 마이크로프로그램을 담고 있는 제어 ROM 메모리(210 × 73-비트), 제어 메모리에 대한 주소 레지스터 CAR(Control Address Register), JVM 명령어를 대응하는 마이크로프로그램이 담긴 주소로 매핑시키는 매핑 ROM 메모리, 그리고 반복되는 마이크로프로그램을 위한 SBR(Subroutine Register), 분기 결정 회로, Incrementor 회로 등으로 구성된다. 마이크로프로그램의 분기 동작을 제어하기 위한 외부 플래그 정보로는 JA_start신호, Int_Ack신호, 데이터 패스에서 제공하는 조건 코드 CC(condition code), 명령어 버퍼에서 제공되는 issue_valid, BSWP_en 신호로 구성된다.

IV. 설계 검증 및 성능 분석

본 연구에서 설계한 자바 가속기는 먼저 개별 데이터 패스 모듈을 Verilog-HDL로 작성한 후 Cadence사의 NC-Verilog 소프트웨어를 사용하여 올바른 동작을 검증하였다. 데이터 패스에 대한 올바른 동작을 확인한 후에 제어 회로와 통합한 설계 검증 환경을 구축하였다. 완전한 JVM 검증을 하려면 호스트 프로세서를 연결해야 하지만, JVM의 일부 코드를 에뮬레이션(emulation)할 수 있는 호스트 프로세서가 정의되어 있지 않아서 호스트 프로세서 부분은 테스트 벤치 프로그램에서 적절히 필요한 신호를 생성하여 제공하는 기법으로 대신하였다.

그리고 자바 가속기를 테스트하기 위해, 간단한 산술 명령과 정렬 프로그램, 간단한 메소드 호출 명령을 작성하여, 메모리에 필요한 코드를 정의된 클래스 파일 적재형식에 맞도록 적재한 후, JA_start를 발생시켜 올바른 동작이 수행됨을 확인하였다. 기존 클래스 파일을 내부 메모리에 적재할 때 자바 가속기가 수행하기 적합한 형태로 수정되어 적재되어야 하는데, 본 연구에서는 참고 문헌 [12-14]의 클래스 파일 적재 연구 결과를 참고하여, 클래스 파일의 메모리 적재 형태를 정의하였다. 그

림 10은 기본적인 산술 연산을 구현하는 자바 프로그램을 수행하는 프로그램을 자바 가속기에 적재한 후, 자바 가속기에 대한 Cadence사의 NC-Verilog 시뮬레이터로 검증한 파형을 나타낸다. NC-Verilog로 설계 검증된 자바 가속기 회로는 0.35um IDEC 삼성 표준 COMS 셀 라이브러리를 사용하여 Synopsys Tool로 합성한 결과, 전체 자바 가속기 회로는 약 14,300개의 게이트(2-input NAND 기준)로 구성되며, 최악 전달 경로가 18ns로 약 50Mhz까지 동작 가능함을 확인하였다. 표 2는 본 연구에서 자바 가속기의 동작 특성을 나타낸다. 가장 많은 사이클이 소용되는 명령은 가변길이의 테이블 기반 분기 명령(tableswitch)과 메소드 호출 명령(invokestatic)으로 각각 27, 28개의 마이크로프로그램으로 구성되었다. 그리고 명령어 당 평균 마이크로프로그램 수는 약 5개이다. 설계한 자바 가속기의 MIPS(Million Instruction Set Per Second) 기준으로 예상 성능은 식 (2)과 같다.

$$\text{자바 가속기의 성능(MIPS)} = \frac{f}{N} \qquad (2)$$

여기서, f=clock frequency, N= 명령어 당 필요한 클록 수식 (2)에서 f로 50 Mhz, N으로 명령어 당 평균 마이크로프로그램 수 5를 사용할 경우 약 10 MIPS정도의 성능을 얻을 수 있음을 알 수 있다. 현재 설계된 자바 가속기는 적은 면적과 적절한 명령어 처리능력으로 호스트 프로세서와 통합되어 사용될 경우 Hardware-Software Co-design 형태로 모바일 자바에 적용할 수 있을 것으로 판단된다.

표 2. 설계된 자바 가속기의 전기적 특성
Table 2. Electrical characteristics of designed Java accelerator

동작 주파수	50 Mhz
가장 긴 마이크로프로그램을 가진 명령 (마이크로프로그램 수)	Invokestatic(28), Tableswitch(27)
게이트 수	14,300
지원하는 명령어 개수	81
제어 기법	마이크로프로그램 제어
데이터 버스	32 비트
주소 버스	26 비트
평균 성능 (명령어 당 평균 마이크로프로그램 수)	10 MIPS (5)
최장 전달 지연	18 ns
사용한 표준 셀	0.35um CMOS Cell Library

V. 결 론

본 논문에서는 JVM의 하드웨어-소프트웨어 공동 설계 방안을 연구하고 이를 바탕으로 32 비트 호스트프로세서에 보조 프로세서 형태로 동작하는 자바 가속기 구조를 Verilog HDL(Hardware Description Language)언어로 설계한 후 성능을 평가하였다. 설계된 자바 가속기는 호스트 프로세서와 공유 메모리를 통한 효율적인 인터페이스를 갖고 있으며, 다양한 길이 특성을 갖는 JVM 명령어를 처리하기 위해 새로운 구조의 명령어 버퍼를 내장하며, 81개의 명령어를 하드웨어로 처리한다. 그리고 하드웨어로 처리되지 않는 명령은 호스트 프로세서에 서 소프트웨어 에뮬레이션으로 구현하는 하드웨어-소프트웨어 공동 설계기법으로 자바 가상 기계를 구현하였다. 기존 자바 가속기는 대부분 단순한 명령어만 구현하므로 호스트 프로세서와 자바 가속기간의 잦은 정보 전달 발생으로 성능이 저하되는데 비해, 본 논문의 자바 가속기는 invokestatic, tableswitch와 같은 복잡한 명령을 직접 자바 가속기에서 처리하므로 통신 오버헤드를 최소화한다.

자바 가속기는 0.35um 표준 셀 라이브러리로 설계하였으며, 약 14,300개의 게이트로 구성되며 50Mhz의 동작 주파수를 갖는다. 설계된 자바 가속기는 작은 면적과 적절한 연산 성능으로 모바일 환경 및 자바 카드 분야에서 범용 호스트 프로세서와 통합되어 IP 형태로 응용될 수 있을 것으로 판단된다. 그리고 제어 회로 기법으로 마이크로프로그램 제어기법을 사용하였는데, 명령어 추가가 용이하고, 융통성이 크므로 보다 많은 명령어를 추가할 경우 다양한 응용 분야에 적용할 수 있을 것으로 판단된다.

앞으로 설계한 자바 가속기와 호스트 프로세서 간의 통합 작업, 다중 쓰레드 지원, 명령어 폴딩 회로 내장을 통한 고속 하드웨어 개발, 상수 풀에 대한 고속 레졸루션 하드웨어 등의 연구가 필요하다.

감사의 글
본 연구에 반도체 설계 교육센터(IDEC) 지원 CAD 소프트웨어가 사용되었습니다.

참고문헌

[1] Philip Koopman, Jr, Stack Computers-The

New Wave- Ellis Horwood Limited, 1989.
[2] John Meyer and Troy Downing, Java Virtual Machine, O'Reilly, 1997
[3] James P. White and David A. Hemphill, Java 2 Microedition, Manning Publication Co., 2002.
[4] Sun Microsystems Inc, Java CardTM 2.2 Virtual Machine Specification, April, 2002.
[5] Timothy Cramer, et.al, "Compiling Java Just In Time", IEEE Micro, May/June, 1997, pp.36-43.
[6] J. Michael O'Connor and Marc Tremblay, "PicoJava-I : The Java Virtual Machine in Hardware", IEEE Micro, March/April, 1997, pp.45-53,
[7] Markus Levy, "Java to Go : Part I", Microprocessor Report, February 12, 2001. pp.1-4.
[8] Markus Levy, "Java to Go : Part II", Microprocessor Report, Martch 5, 2001. pp.1-3.
[9] Markus Levy, "Java to Go : Part III", Microprocessor Report, Martch 26, 2001. pp.1-3.
[10] Byeong-Yoon Choi, "Design of Instruction Buffer and Folding Circuit for Java Accelerator", North-East IT Symposium '2001, 2002, pp.644-648.
[11] Radhakrishman, Microarchitecture technique to enable efficient Java execution, Ph.D thesis, University of Texas at Austin, 2000.
[12] 강두진, 내장형 자바 시스템을 위한 롬 이미지 제작기법, 연세대학교 전산과학과 석사 논문, 1999.12
[13] 최병윤, "Java USIM 혹은 Mobile Java의 성능 향상 방안 연구", 한국 전자 통신 연구원 위탁 과제 최종 연구 보고서, 2003. 11.
[14] Martin Schoeberl, "JOP-Java Optimized Processor", http://www.jopdesign.com

저자소개

최병윤(Byeong-Yoon Choi)

1985년 2월 : 연세대학교 전자공학과 졸업
1992년 8월 : 연세대학교 전자공학과 공학 박사
1997년 8월~1998년 8월 : 일리노이 주립대 방문 연구 교수
1993년 3월~현재 : 동의대학교 교수
※관심분야 : RISC/Java 마이크로프로세서 설계, 통신, 네트워크 및 암호 알고리즘의 VLSI 설계

박영수(Young-Soo Park)

1990년~현재 : 한국 전자 통신 연구원 선임 연구원
※관심분야 : CAD 및 VLSI 설계, 암호 프로세서 설계, IC 카드 설계