DOI: 10.3745/KIPSTA.2009.16A.6.443

임베디드 프로세서의 성능 향상을 위한 DIAM의 진보한 아키텍처

윤 종 희[†] · 신 세 철[††] · 백 윤 흥[†††] · 조 정 훈[††††]

요 약

32비트 아키텍처가 현대 마이크로프로세서의 표준이 되어가고 있음에도 불구하고 작은 사이즈와 적은 파워 소모량을 우선시 하는 저가의 프로세서에서는 여전히 16비트 아키텍처가 사용되고 있다. 그러나 16비트 아키텍처는 특정 애플리케이션을 위한 특별한 명령어들을 추가할 만한 충분한 인코딩 공간이 제공되지 않는 결정적인 단점을 가지고 있다. 이것을 극복하기 위해 기존의 많은 아키텍처에서 일반적이지 않은 다양한 어드레싱 모드들을 수용하기 위한 직교적이지 않으면서(non-orthogonal) 불규칙한 명령어 셋이 사용되었다. 일반적으로 직교적이지 않은 아키텍처들은 최적의 코드를 생성하기 위해서 매우 정교한 컴파일러 기술을 요구하는 경향이 있기 때문에 컴파일러에 지향적이지 않는 것으로 간주된다. 이전에 우리는 이런 문제를 해결하기 위해 새로운 어드레싱 모드인 DIAM (dynamic implied addressing mode)을 사용하는 컴파일러 지향적 프로세서를 제안하였다. 이 논문에서는16비트 프로세서에서 우리의 애플리케이션들을 위해 더 많은 인코딩 공간을 제공하였던 DIAM을 사용하는 아키텍처를 설명하고, 그것을 보완하여 성능이 더욱 개선된 아키텍처에 대하여 설명할 것이다. 우리의 실험에서 제안된 아키텍처는 기존의 아키텍처에 비해 평균적인 성능을 11.6% 증가시켰다.

키워드 : 다이엠, 묵시적어드레싱 모드, 임베디드 프로세서

Advanced Architecture using DIAM for Improved Performance of Embedded Processor

Jonghee M. Youn[†] · Sechul Shin[††] · Youheung Paek[†††] · Jeonghun Cho[††††]

ABSTRACT

Although 32-bit architectures are becoming the norm for modern microprocessors, 16-bit ones are still employed by many low-end processors, for which small size and low power consumption are of high priority. However, 16-bit architectures have a critical disadvantage for embedded processors that they do not provide enough encoding space to add special instructions coined for certain applications. To overcome this, many existing architectures adopt non-orthogonal, irregular instruction sets to accommodate a variety of unusual addressing modes. In general, these non-orthogonal architectures are regarded *compiler-unfriendly* as they tend to requires extremely sophisticated compiler techniques for optimal code generation. To address this issue, we proposed a compiler-friendly processor with a new addressing mode, called the *dynamic implied addressing mode*(DIAM). In this paper, we will demonstrate that the DIAM provides more encoding space for our 16-bit processor so that we are able to support more instructions specially customized for our applications. And we will explain the advanced architecture which has improved performance. In our experiment, the proposed architecture shows 11.6% performance increase on average, as compared to the basic architecture.

Keywords : DIAM, Dynamic Implied Addressing Mode, Embedded Processor

※ 본 연구는 2006년 정부(교육인적자원부)의 재원으로 한국학술진흥재단의 신진교수연구지원사업 (KRF-2006-331-D00449), 교육과학기술부/한국과학재단 우수연구센터육성사업 (R11-2008-007-01001-0), 지식경제부 출연금으로ETRI, SoC 산업진흥센터에서 수행한 ITSoC 핵심설계인력양성사업, 서울시 산학연 협력사업, 2009 년도 정부(교육과학기술부)의 재원으로 한국과학재단의 국가지정연구실 사업(2009-0083190), 지식경제부 및 정보통신연구진흥원의 대학 IT 연구센터 지원사업(IITA-2009-C1090-0902-0020), 지식경제부 및 정보통신 연구진흥원의IT원천기술개발사업[과제관리번호:2006-S-006-02, 과제명: 유비쿼터스 단말용 부품/모듈]의 지원을 받아 수행 되었습니다.
† 준 회 원 : 서울대학교 전기컴퓨터공학부 박사과정
†† 준 회 원 : 경북대학교 전자전기컴퓨터공학부 석사과정
††† 종신회원 : 서울대학교 전기컴퓨터공학부 교수
†††† 종신회원 : 경북대학교 전자전기컴퓨터학부 조교수(교신저자)
논문접수 : 2009년 6월 4일
수 정 일 : 1차 2009년 7월 20일
심사완료 : 2009년 8월 21일

1. 서 론

임베디드 프로세서들은 보통 더 적은 에너지 소비량과 코드 사이즈를 이루기 위해서 매우 엄격한 디자인 조건들을 만족시켜야 한다. 그렇기 때문에 많은 임베디드 프로세서를 위하여 최근 32비트 아키텍처가 대부분의 마이크로프로세서의 표준이 되고 있음에도 불구하고 16비트 아키텍처가 여전히 사용되고 있다. 하지만 16비트 아키텍처는 인코딩 공간이 충분하게 제공되지 않는다는 제약이 있다. 이 제약은 디지털신호처리와 네트워크 처리와 같은 특별한 애플리케이션 분야에 대한 임베디드 프로세서에 대해서는 결정적인 문제가 될 수 있다. 왜냐하면 임베디드 프로세서는 일반적으로 타겟 애플리케이션의 수행 성능을 향상시키기 위해 제작된 다양한 명령어를 포함할 수 있는 CISC 방식으로 디자인하기 때문이다. 이 제약을 극복하기 위해 종종 특정 임베디드 프로세서의 명령어는 전용 어드레싱 모드(dedicated addressing mode)를 사용한다. 이 어드레싱 모드를 사용하면 각 명령어의 오퍼랜드들은 프로세서에서 사용 가능한 하드웨어 레지스터 중에서 일부분만 사용할 수 있게 된다.

전용 어드레싱 모드는 오퍼랜드 필드의 폭을 줄일 수 있는 장점이 있기 때문에 제한된 명령어 공간 안에 더 많은 명령코드와 오퍼랜드 필드를 위한 인코딩 공간을 확보할 수 있다. 이것을 설명하기 위해서 총 6개(입력 레지스터 4개, 누산기 2개)의 데이터 레지스터를 갖고 있는 Freescale DSP566xx [9]를 살펴보자. 입력 레지스터들은 곱셈이나 덧셈의 입력 오퍼랜드로도 사용될 수 있는 반면에 누산기는 오직 덧셈의 출력에만 연결되어 있다. 따라서 DSP566xx의 덧셈 명령어는 목적지 오퍼랜드의 인코딩을 위하여 오직 한 비트만 요구한다. (그림 1)은 DSP566xx의 ADD 명령어 포맷을 나타낸다. (그림 1)을 통해서 3개의 비트(JJJ) 가 소스 오퍼랜드로 사용되었고, 목적지 오퍼랜드를 위해서는 단지 한 개의 비트(d)만 사용되는 것을 확인할 수 있다.

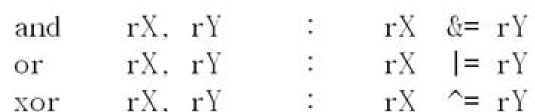
만약에 전용 어드레싱 모드의 특별한 형태라 생각하는 묵시적 어드레싱 모드(implied addressing mode)를 사용한다면 소스 오퍼랜드를 기술하기 위해서는 몇 비트가 요구되지만 목적지 오퍼랜드를 위해서는 어떤 비트도 할당할 필요가 없음을 알 수 있다. 연산의 결과 값은 오직 정해진 레지스터에 존재하기 때문에 별도로 지정할 필요가 없다.

일반적으로 전용 어드레싱 모드를 지원하기 위해서는 하드웨어 레지스터 파일들이 명령어에 따라 물리적으로 분리되어 있는 HRA (heterogeneous register architecture)를 사용한다. 이것은 각각의 레지스터들의 접근성이 HRA를 사용하는 프로세서의 데이터 패스 토폴로지에 의해서 제한된다는 것을 의미한다. 명령어에 대한 레지스터 아키텍처의 이질성은 보통 프로세서가 직교적이지 않은 명령어 집합구조(ISA, Instruction Set Architecture)를 가지도록 한다. 왜냐하면 각각의 레지스터 파일의 사용은 코드 안에 있는 명령어 마다 다르기 때문이다. 컴파일러 입장에서는 유감스럽게도 직교적이지 않은 ISA를 위한 코드 생성은 RISC 형식의 프로세스에서 발견되는 직교적 ISA보다 훨씬 더 어려운 알고리즘을 요구한다. 컴파일러가 명령어를 선택함과 동시에 명령어 각각의 오퍼랜드를 위해 할당되는 레지스터를 결정해야 하기 때문이다. 이것은 컴파일러가 같은 시간에 코드 생성의 두 가지 단계(즉, 명령어 선택과 레지스터 할당)를 모두 다뤄야 한다는 것을 의미한다. *Phase coupling* 이라고 불리는 이 문제는 NP-hard 로 알려져 있다. 이전 연구 [3, 8]에서 *phase coupling* 에 대한 정교한 알고리즘이 없으면 HRA는 각각의 명령어의 오퍼랜드를 위해 분산되어 있는 레지스터 파일들 안에 적절하지 않은 레지스터의 할당에 의해서 주로 발생하는 더욱 분산된 코드를 생성하는 경향이 있다고 보고했다. 전용 어드레싱 모드의 개발 방법은 오퍼랜드들의 인코딩 폭을 줄일 수는 있겠지만 코드 생성 문제를 대단히 복잡하게 만들기 때문에 컴파일러 지향적이지 못하다고 할 수 있다.

연산에 대하여 오퍼랜드의 수를 줄여서 인코딩하는 것은 인코딩 공간의 부족을 극복하기 위한 대체 방법으로 간주 될 수 있다. 이에 관련된 전형적인 예는 두 개의 오퍼랜드를 사용하는 명령어들이다. 비록 바이너리 연산에 3개의 오퍼랜드가 필요하지만 한 개의 소스 오퍼랜드와 목적지 오퍼랜드를 같은 위치에 공유함으로써 두 개의 오퍼랜드로 바이너리 연산을 인코드 할 수 있다. 실례로 (그림 2)는 ZSP400 [11]에서rX 가 왼쪽 소스와 목적지 오퍼랜드로 사용된다.

이 방법의 장점은 프로세서가 더욱 큰 인코딩 공간을 얻기 위해 전용 어드레싱 모드에 의존하는 것을 피할 수 있다는 것이다. 이로 인해 코드생성의 복잡성을 매우 줄일 수 있는 직교적인 ISA 를 유지할 수 있다. 그러나 두 개의 오퍼랜드를 가지는 명령어로 구성된 2-address 코드는 보통 많은 move 명령어를 포함한다는 단점을 가지고 있다. 이것은 기존의 내용을 보호하기 위해 추가적인 move 연산을 수반하기 때문이다. 우리의 연구에 따르면 2-address 코드는 일반적으로 3-address 코드보다 거의 두 배정도 많은 move 명령어를 가지고 있다. (그림 3)은 벤치마크 IDCT (inverse discrete cosine transform) 의 코드를 보여준다. 2-address 코드가 3-address 코드에서는 불필요한 2

(그림 1) Freescale DSP566xx 의 ADD 명령어



(그림 2) ZSP 400 의 명령어[11]

```
// x5=x8-(W1+W7)*x5;
    mov r10 r11
    add r10 r11
    mul r10 r5
    sub r8 r20
    mov r5 r8
```

(a) 2-어드레스 코드

```
// x5=x8-(W1+W7)*x5;
    add r10 r11 r7
    mul r10 r10 r5
    sub r5 r8 r10
```

(b) 3-어드레스 코드

(그림 3) IDCT를 위한 2-어드레스 코드 vs 3-어드레스 코드

개의 move 연산이 추가된 것을 알 수 있다.

본 논문에서는 대부분의 프로세서 아키텍트가 많은 명령어와 함께 16비트 임베디드 프로세서를 디자인 할 때 직면하는 인코딩 공간 문제를 경감시키기 위한 시도를 하였다. 이러한 목적을 달성하기 위해 DIAM (dynamic implied addressing mode) 이라고 불리는 새로운 어드레싱 모드를 제안했다 [21]. DIAM을 사용하면 각각의 명령어는 오직 두 개의 오퍼랜드로 바이너리 연산을 인코드한다. 그러나 그것은 세 번째 오퍼랜드를 암시적으로 기술하기 때문에 일반적으로 두 개의 오퍼랜드를 사용하는 것과는 다르다. 만약 세 번째 오퍼랜드가 필요하다면 컴파일 시간에 DIAM이 사용 될 것이다. 결국 세 번째 오퍼랜드는 명령어 워드로부터 숨겨지고 프로세서 안에 분리된 위치에 저장된다. 실행 시간에 이 오퍼랜드는 하드웨어에 의해서 그 위치로부터 자동적으로 검색된다. 사실 DIAM은 명령어에서 오퍼랜드가 나타나지 않는다는 의미에서 보통의 묵시적 어드레싱 모드[예. (그림 1)의 목적지 오퍼랜드]와 약간 유사하다. 그러나 DIAM을 사용하는 오퍼랜드가 보통의 것과 다른 것은 고정된 레지스터에 정적으로 바운드 되지 않는 것이다. (그림 1)을 예로 들어, 묵시적 어드레싱 모드를 사용하는 목적지 오퍼랜드는 반드시 정해진 레지스터를 사용해야 한다. 하지만 DIAM을 사용하면 오퍼랜드는 다른 사용 가능한 레지스터를 동적으로 가리킬 수 있다. 이런 의미에서 보통의 방법을 SIAM(static implied addressing mode)이라 하고, 그 반대는 DIAM으로 명명한다.

본 논문은 다음과 같이 구성된다. 2장에서는 다른 프로세서들이 어떻게 인코딩 공간 문제를 다루는지에 대하여 논할 것이다. 3장은 우리의 16 비트 프로세서의 기본적인 아키텍처를 소개하고, 4장에서는 DIAM을 사용하는 아키텍처에 대하여 설명한다. 5장에서는 DIAM을 보완한 새로운 아키텍처에 대하여 설명하고, 6, 7장은 각각 실험결과와 결론을 나타낸다.

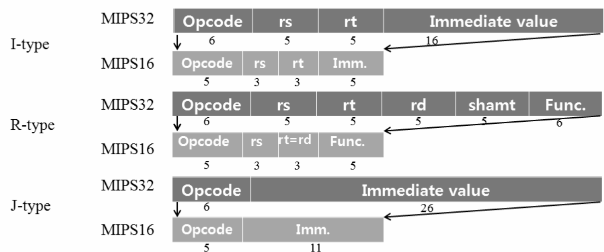
2. 관련 연구

앞에서 말한 것과 같이 현재 존재하는 프로세서들은 SIAM 또는 두 개의 오퍼랜드를 가지는 명령어를 포함하는 전용 어드레싱 모드와 함께HRA를 채택함으로써 인코딩 공간 문제를 극복한다. 예를 들어 디지털 신호처리 (digital signal processing)를 위한 LSI Logic 16비트 코어인 ZSP400은 DSP에 특화된 명령어를 지원하기 위한 명령어 인코딩을 구성한다. ZSP400은 두 개의 가드 레지스터(guard register - g1, g0)와 16개의 레지스터를 가지고 있다. 범용 레지스터의 쌍은 한 개의 32비트 레지스터로 사용될 수 있고, (r1, r0) 와 (r2, r3) 는 누산기로 사용될 수 있다. 그리고 ZSP400은 기본적인 연산(그림 2 참조)뿐만 아니라 더욱 복잡한 DSP에 특화된 연산도 두 개의 오퍼랜드 명령어로 인코드한다. 예를 들면 ZSP400은 조합 명령어인 MAC(multiply and accumulate) 을 지원한다. 실제로 MAC은 4개의 오퍼랜드를 요구하기 때문에 2-어드레스 형식을 적용하기 위해서는 (그림 4)에서 보여지는 것과 같이 두 개의 오퍼랜드에 대하여 SIAM 을 사용한다. 더욱 복잡한 연산이 두 개의 오퍼랜드 명령어로 인코드되기 때문에 전용 혹은 묵시적 어드레싱 방법은 더욱 공격적으로 지원되어야한다. 따라서 소스 오퍼랜드의 내용을 저장하기 위한 move명령어의 추가적인 증가가 발생하게 된다.

Thumb ISA [4] 는 32비트 ARM 프로세서에서 16비트 인코딩의 장점을 가지고 있다고 소개되었다. 이것은 32비트 프로세서에서 16비트 ISA를 포함하여 유지하기 위해서는 런타임 때에 32비트 명령어와 동등하게 확장시켜야 한다. 인코딩 공간의 부족 때문에 16비트 명령어는 그것들의 오퍼랜드에 대하여 16개의 레지스터 중에서 오직 절반만 할당하는 전용 어드레싱을 사용한다. 이것은 분명히 spill code를 증가시킬 것이다. 그리고 분산된 코드를 줄이기 위한 더욱 정교한 코드 생성 알고리즘이 요구될 것이다. Thumb 또한 16비트 명령어를 위해 2-어드레스 포맷을 사용하기 때문에 ZSP400과 유사한 단점을 가지고 있다. (그림 5)에서 보여지는 것과 같이 MIPS16 [5]도 ARM-Thumb과 유사한 접근 방식을 사용했기 때문에 Thumb과

Name	Syntax	Behavior
MAC.A	mac.a rX, rY	{g0 r1 r0} += rX * rY
MAC.B	mac.b rX, rY	{g1 r3 r2} += rX * rY
MACN.A	macn.a rX, rY	{g0 r1 r0} -= rX * rY
MACN.B	macn.b rX, rY	{g1 r3 r2} -= rX * rY
MAC2.A	mac2.a rX.e, rY.e	{g0 r1 r0} += rX *rY +r(X+1)*r(Y+1)
MAC2.B	mac2.b rX.e, rY.e	{g1 r3 r2} += rX *rY +r(X+1)*r(Y+1)

(그림 4) ZSP400의 MAC 명령어들



(그림 5) MIPS32 VS MIP16

같은 단점을 가진다.

본 논문의 접근 방식에서 우리는 두 개의 오퍼랜드로 바이너리 연산을 인코드하는 DIAM을 사용한다. 이것은 전용 어드레싱을 위해서 HRA에 의지하지 않고 인코딩 공간을 증가시킬 수 있다. DIAM을 사용하는 프로세서는 더욱 효율적인 코드 생성을 위한 직교적인 ISA를 유지할 수 있기 때문에 컴파일러의 복잡성을 줄일 수 있다. 전용 어드레싱을 사용하는 기존의 HRA들은 분산 디코딩 기법(dispersed decoding mechanism)을 사용하는 경향이 있다. 분산 디코딩 장치는 종종 데이터 포워딩 장치를 복잡하게 만들고 정적으로 예측할 수 없게 만든다. 따라서 컴파일 시에 효율적인 명령어 스케줄링을 하는 것을 제한하게 된다. 그러나 DIAM은 중앙으로 집중하는 명령어 디코딩 장치를 유지할 수 있으므로 하드웨어가 컴파일러 명령어 스케줄링에 대한 조건들을 단순화 시킬 수 있다. 또한 우리의 프로세서는 사실상DIAM을 사용하는 명령어들이 3개의 오퍼랜드를 가지기 때문에 기본적으로3-어드레스 코드 포맷을 가진다. 그렇기 때문에 우리의 기계 코드는 2-어드레스 코드에서 사용된 추가 move 명령어가 불필요하다.

3. 기본적인 아키텍처

우리들은 기본적으로 전형적인 RISC 형식의 16비트 프로세서를 디자인 하였고, 미디어 애플리케이션을 위한 명령어의 확장을 시도하였다. 이러한 시도에서 특정 애플리케이션에 대하여 특별한 명령어를 추가하기에는 인코딩 공간이 부족하다는 것을 알게 되었다. 이 문제를 해결하기 위해 기본적인 아키텍처에 DIAM을 추가하였다. 이 장에서는 기본적인 아키텍처로 사용된 16비트 프로세서의 구조를 설명한다. 이어지는 장에서는 DIAM을 지원하기 위해 기존의 아키텍처가 어떻게 수정되었는지에 대하여 논할 것이다. 미디어 처리를 위한 애플리케이션은 이미지 압축과 선형 순환, 그리고 웨이블릿 필터를 포함한다. 이 애플리케이션은 대부분의 실행 시간을 커널 코드에서 소비한다. 그렇기 때문에 미디어 애플리케이션의 성능과 비용에 관한 조건을 만족하기 위해서 커널 부분에 초점을 맞춰서 기본적인 아키텍처를 디자인하였다.

3.1 아키텍처 개발과 수행을 위한 툴들

기본적인 아키텍처는 Coware사 [12, 18]의 디자인 툴인 Processor Designer를 사용하여 개발되었다. ADL(architecture description language) 인 LISA2.0 [16]을 기반으로 하는 Processor Designer는 모든 디자인 단계의 통합 개발 환경을 제공한다 [17]. Processor Designer는 자세한 기술서와 추가적인 조건들을 이용하여 관계된 아키텍처의 HDL (hardware description language) 코드를 생성한다. 생성된 코드는 Synopsys사 [20]의 상업적인 합성 툴인 Design Compiler(DC) 를 사용하여 게이트 레벨(gate level) 에서 합성 가능하다. 다음은 소프트웨어 개발 도구를 보인다.

- **프로그래밍 언어 디버거**: 그래픽 환경의 디버거로 명령어들을 디버그 한다.
- **어셈블리어(lasm) 와 디스어셈블러(ldsm)**: 각각의 아키텍처에 대하여 어셈블리 명령어 파일을 오브젝트 코드로(또는 오브젝트코드를 명령어 파일로) 변환해 준다.
- **링커(link)**: 전용 링커 명령파일에 의해서 라이브러리 링킹이나 메모리 정보를 제어하거나 기술 한다.
- **디버거를 사용한 ISA 시뮬레이터(ldb)**: 명령어와 데이터 파이프라인들의 지원을 포함하는 사이클 단계의 시뮬레이션을 제공한다.

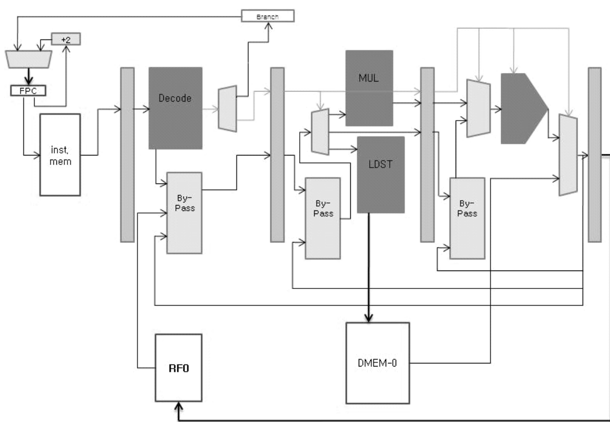
3.2 기본적인 아키텍처의 특징

기본적인 아키텍처의 특징들은 아래에 나열되어있다.

- **디지털 신호처리를 위한 아키텍처**: 미디어 애플리케이션의 커널 부분을 효율적으로 처리하기 위한 코프로세서로써 제공된다.
- **16비트 데이터와 명령어 패스**: 코프로세서로써 코드 사이즈, wrapper die 그리고 파워 소모를 줄이기 위하여 디자인 되었다.
- **하버드 아키텍처(Harvard Architecture) (1-IMEN, 1-DMEM)**: 기본적인 아키텍처는 명령어 메모리와 데이터 메모리가 분리되어있다. 이것은 명령어 메모리와 데이터 메모리에 동시 접근이 가능하여 메모리 대역폭을 확장시키기 위함이다. 또한 DMEM은 읽기와 쓰기를 위하여 하나의 결합된 포트를 사용하는 동기화 메모리 접근을 가진다. 반면에 IMEM은 비동기적인 메모리 접근을 가진다.
- **다섯 단계 파이프라인(FE, DC, EX1, EX2, WB)**: 기본적인 아키텍처는 Fetch, Decode, Execute1, Execute2 그리고 Writeback의 다섯 단계로 구성되어 있다. Execute단계는 EX1과 EX2 두 단계로 분리되어 있는데 곱셈과 메모리 연산들이 다른 연산들 보다 더 긴 지연시간을 가지기 때문이다.
- **싱글 레지스터 파일(16개의 슬롯, 두 개의 읽기 포트와 한 개의 쓰기 포트)**: 싱글 레지스터 파일은 16개의 레지스터 슬롯과 함께 하나의 범용 레지스터를 가지고 있다. 이 레지스터 파일은 한번에 두 개의 읽기와 하나의 쓰기 연산을 허용한다.
- **내부 메모리 구조**: 기본적인 아키텍처는 모든 메모리가 프로세서 내부에 위치하기 때문에 버스 구조가 존재하지 않는다. 또한 직접 메모리 전송 제어 신호나 OS에 의한 인터럽트 신호와 같은 외부 시스템 온 칩 장치들에 의한 충돌이 없다.

3.3 기본적인 아키텍처의 수행

(그림 6)의 블록 다이어그램은 파이프라인 단계들, 메모리 블록들, 레지스터 파일, 데이터 패스 그리고 기능적인



(그림 6) 기본적인 아키텍처의 블록 다이어그램[21]

장치들을 포함한다. 기본적인아키텍처는 FE, DC, EX1, EX2 그리고 WB의 5단계 파이프라인으로 구성되고, 각 명령어들은 FE 단계에서 메모리로부터 패치되고 DC 단계에서 명령어가 해석되어 제어 신호를 각 모듈에 전달한다. Branch, jump 그리고 call 명령어들은 파이프라인의 효율성을 위해 DC 단계에서 끝나도록 디자인 하였다. 메모리 연산들은 EX1 에서 요구되고 그 결과는 EX2로 이동된다. 곱셈과 ALU 그리고 메모리 연산들은 임계 경로 지연을 가지고 있기 때문에 EX1과 EX2로 구별된다. 이것은 결정적인 지연을 줄일 수 있지만 EX2 로 데이터를 전송해야 하는 제약을 야기한다. 레지스터 파일에 쓰는 것은 WB단계에서 수행된다.

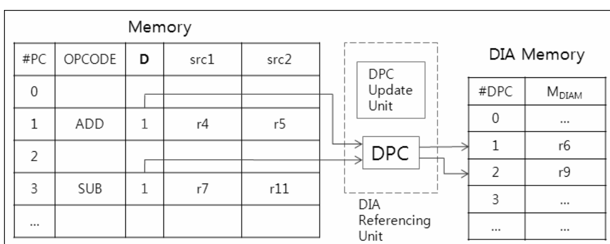
4. 선행 연구

4.1 DIAM(Dynamic Implied Addressing mode)의 개념

DIAM을 사용하는 명령어는 다음과 같다.

- 바이너리 연산을 위해서 3개의 오퍼랜드(명시된 것 두 개와 암시된 것 하나)를 사용한다.
- 암시적인 오프셋을 사용하는 메모리 연산을 위해 변위 어드레싱 모드(displacement addressing mode)를 사용한다.

메모리 명령어의 오프셋 또는 명령어의 목적지 레지스터는 (그림 7)에서 나타내는 것과 같이 명령어 워드 안에



(그림 7) DIAM 의 개념

있는 D 비트에 의해서 암시적으로 사용될 수 있다. 또한 DIAM을 사용하여 3번째 오퍼랜드가 DIA라는 분리된 메모리 M_{DIAM} 안에 암시적으로 저장된다. M_{DIAM} 은 오직 읽기만 가능한 메모리로 DPC(dynamic program counter)에 의해서 포인트 된다. M_{DIAM}안에 저장된 DIAM 값 V_{DIA}는 ALU 연산의 목적지 레지스터의 주소, 메모리 연산의 오프셋 값 또는 branch와 같은 명령어를 위한 DPC의 증가 값이 될 수 있다. 명령어 안에 있는 D 비트는 M_{DIAM}가 DPC 에 의해서 포인트 되는 위치에 묵시적인 오퍼랜드 값 V_{DIA} 을 가지고 있는지에 대하여 나타낸다. D 비트가 셋 되어 있으면 DIAM 참조 장치는 DPC를 조사한다. DPC는 M_{DIAM} 안에 있는 묵시적인 오퍼랜드의 현재 위치를 나타낸다. 묵시적 오퍼랜드의 값이 패치 될 때, DPC는 DPC 업데이트 장치에 의해서 증가된다. 또한 두 개의 소스 오퍼랜드를 위하여 8비트가 요구되는데, 이는 16개의 레지스터 모두가 두 개의 소스 오퍼랜드를 위해서 사용되기 때문이다.

4.2 DIAM을 사용하는 명령어의 분류

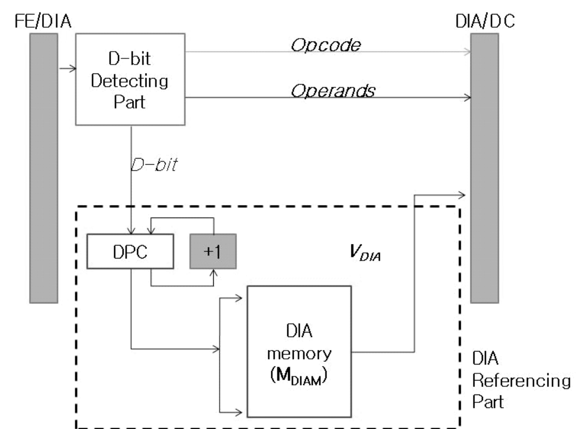
후보 명령어(Candidate instruction) I_{DIAM} 는 DIAM을 사용하는 명령어이다. 이것은 다음과 같이 두 개의 그룹으로 구성된다.

- 흐름 제어가 불가능한 명령어들(Non-control-flow instructions, I_{DIAM}^{NC}): ALU 또는 메모리 명령어들과 같이 소스 코드의 제어 흐름을 바꿀 수 없는 명령어들이다.
- 흐름 제어가 가능한 명령어들(Control-flow instructions, I_{DIAM}^{C}): jump, call과 같이 소스 코드의 제어 흐름을 바꿀 수 있는 명령어들이다.

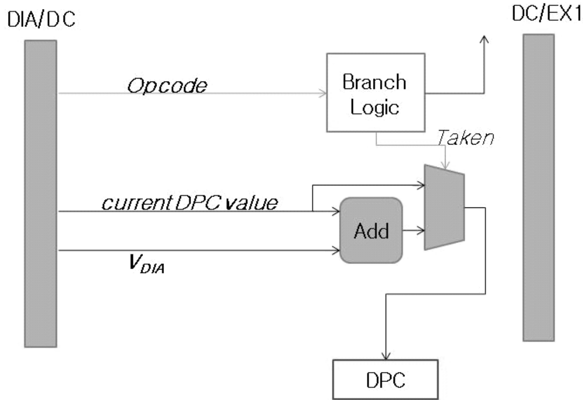
4.3 아키텍처 구성

4.3.1 DIAM을 사용하는 아키텍처의 구성

(그림 8)에서 보여지는 것과 같이 DIA 참조 장치를 수행하기 위해 FE 와 DC 단계 사이에 DIA 라고 불려지는



(그림 8) DIA 단계의 DIA 참조 장치[21]

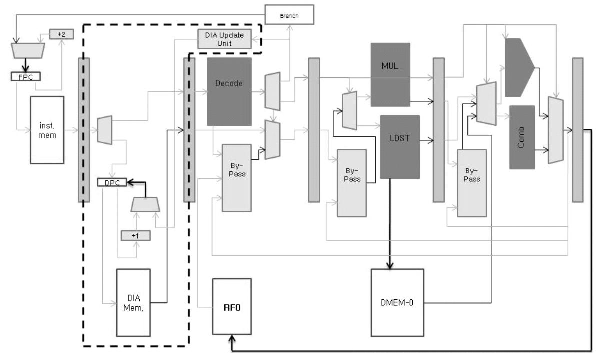


(그림 9) branch에 대한 DPC 업데이트[21]

새로운 파이프라인 단계를 추가한다. DIA 참조 장치는 DIA 단계에 위치하고 DIA 검출 부분과 DIA 참조 부분으로 구성된다. DIA 검출 부분은 D 비트가 셋인지 아닌지를 확인하고 DC 단계로 명령코드와 오퍼랜드를 전달한다. DIA 참조 부분은 DIA 검출 부분에서 검출된 D비트가 셋일 경우 DPC를 업데이트하고 M_{DIAM}을 참조하여 V_{DIA} 를 가져온다. DIA 단계에서의 연산은 명령어의 종류(I_{DIAM}^{NC} 또는 I_{DIAM}^{C})에 따라서 달라진다. I_{DIAM}^{NC} 에서 D 비트가 셋일 때, DIA 참조 부분은 M_{DIAM} 의 주소를 가져오기 위해 DPC를 참조한 뒤, M_{DIAM}를 읽고 V_{DIA}를 가져온다. V_{DIA} 는 목적지 레지스터의 주소, load 와 store 명령어의 오프셋 값 또는 DPC 증가 값이 될 수 있다. DIA 참조 장치로부터 가져온 V_{DIA} 는 다음단계(DC)로 이동될 것이다. 그러나 D 비트가 리셋일 때는 DIAM을 사용하지 않는 명령어처럼 실행되고 DPC 또한 증가하지 않는다. I_{DIAM}^{C} 에서는 DPC가control flow 변화에 따라서 1이나 그 이상의 값만큼 증가 될 수 있다. (그림 9)에서는 branch 명령어에 대해서 DPC가 업데이트되는 예를 보여준다. V_{DIA} 와 DPC 값은 DIA 단계에 있는 DIA 참조 장치로부터 DC로 이동된다. branch 가 테이큰 될 때 DPC는 현재 DPC의 값과 V_{DIA} 의 합으로 업데이트 된다. 이 경우에 발생하는 데이터 의존 문제를 해결하기 위해 branch 명령어 뒤에 반드시 NOP 명령어가 삽입되어야 한다. branch 가 테이큰 되지 않았을 때는 DPC 가 I_{DIAM}^{NC}와 처럼 1만큼 증가된다. I_{DIAM}^{C} 의 모든 다른 명령어를 위한 DPC의 갱신은 이것과 비슷하게 적용된다.

4.3.2 DIAM을 사용하는 아키텍처의 블록 다이어그램

DIAM을 사용하는 아키텍처는 기본적인 아키텍처에DIA 단계를 추가 한 뒤, DIAM과 같이 수행된다. DIAM을 사용하는 아키텍처가 기본적인 아키텍처와 다른 점은 다음과 같다. 첫 번째, 파이프라인 DIA 단계와 DPC 그리고 M_{DIAM} 가 추가되었다. 두 번째, DC, EX1 그리고EX2 단계들이 목적지 오퍼랜드, 오프셋 값, 또는 DPC 업데이트 값에 대한 추가적인 정보를 전달하기 위해 약간 수정되었다.



(그림 10) DIAM을 사용하는 아키텍처의 블록다이어그램

특히, DPC 값을 업데이트하기 위한 DIA 업데이트 장치가 DC 에 추가되었다.

(그림 10)은 DIAM이 추가된 아키텍처의 블록 다이어그램을 보여준다.

4.3.3 DIAM을 사용하는 아키텍처의 ISA

(그림 11)은 DIAM을 사용하는 아키텍처에 대한 ISA의 한 부분을 보여준다. 첫 번째 열은 기능으로 구분되는 명령어 그룹을 나타내고, 두 번째 열은 각각의 명령어를 나타낸다. 또한 (그림 11)에서 보여주고 있는 명령어들은 I_{DIAM} 에 속한다. 굵은 선으로 둘러싸여 있는 1비트 열의 D는 DIAM을 위한 D 비트를 의미한다.

ALUR	add sub and or xor sll slr sra	OpCode	D	dest	src
ALUI	addi andi ori xori slli srli srai subi		D	dest	imm5
LDST	ldw ldb ldbu stw stb		D	base	imm5
DSP	mul mac		D	dest	src

(그림 11) DIAM을 사용하는 ISA

5. DIAC (DIA+DC) 사용하는 확장된 아키텍처

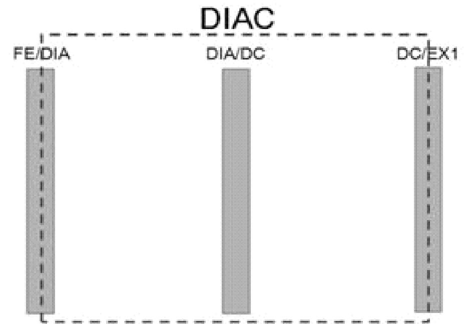
이미 우리는 5단계 파이프라인에서 DIA 라는 새로운 단계를 추가하는 아키텍처인 DIAM을 사용하는 아키텍처를 개발하였다. DIAM을 사용함으로써 코드 사이즈를 줄였고 성능을 향상시켰다. 이 섹션에서는 이미 개발된 DIAM

을 사용하는 아키텍처를 보완하여 좀 더 성능을 향상시킬 수 있는 확장된 아키텍처에 대하여 논할 것이다. 여기서 용어의 혼동을 예방하기 위하여DIA단계를 추가하여 DIAM을 사용하는 아키텍처를 $DIAM_O$, 그 아키텍처를 보완하여 만든 새로운 아키텍처를 $DIAM_N$ 이라고 명명한다.

5.1 DIAC(DIA + DC)

$DIAM_O$ 에서는 새로운 파이프라인 단계인 DIA를 추가하였기 때문에 클럭 타임이 늘어나는 문제가 발생하였다. 물론 애플리케이션 실행 시 기본적인 아키텍처에 비해서 실행 시간(명령어 카운트 X 클럭 타임)의 평균값은 줄어들게 된다. 이것은 줄어든 move 명령어의 개수에 의해서 전체적인 명령어의 동적 카운터 수가 줄어들었기 때문이다. 하지만 클럭 타임이 늘어나는 문제는 여전히 존재하고 있다. 그렇기에 클럭 타임을 줄여서 프로세서의 성능을 더욱 향상시키는 방법을 고안하였다. (그림 12)에서 나타내는 것과 같이 $DIAM_O$ 에서 두 개로 나눠져 있는 파이프라인 단계(DIA, DC)를 합쳐서 하나의 단계(DIAC)로 만드는 것이다.

$DIAM_N$ 의 DIAC 단계는 실제적으로는 $DIAM_O$ 의 DC 단계에 해당한다. $DIAM_O$ 에서 따로 DIA 단계를 두어 DIAM을 참조하고 DPC를 업데이트 했던 작업을 DC 단계로 이동을 시킨 것이다. (그림 13)은 $DIAM_N$ 의 블록다이어그램을 나타내었다. 두 개의 단계가 하나로 합쳐진 것을 확인 할 수 있다.

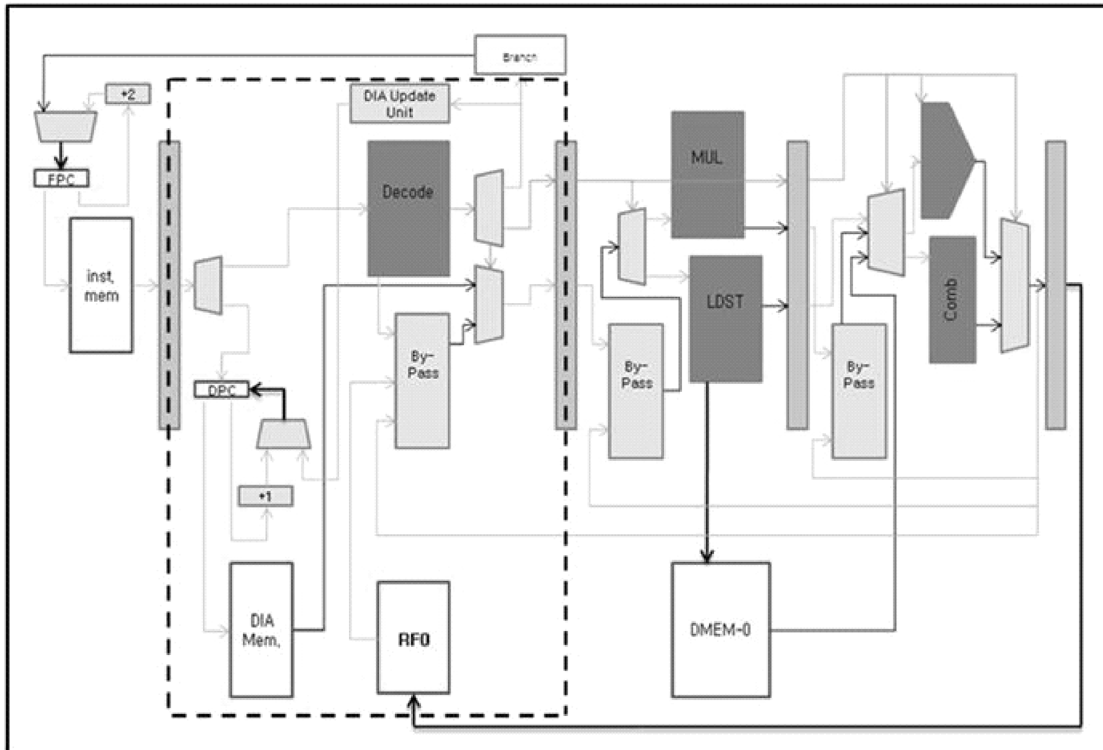


(그림 12) DIAC (DIA + DC)

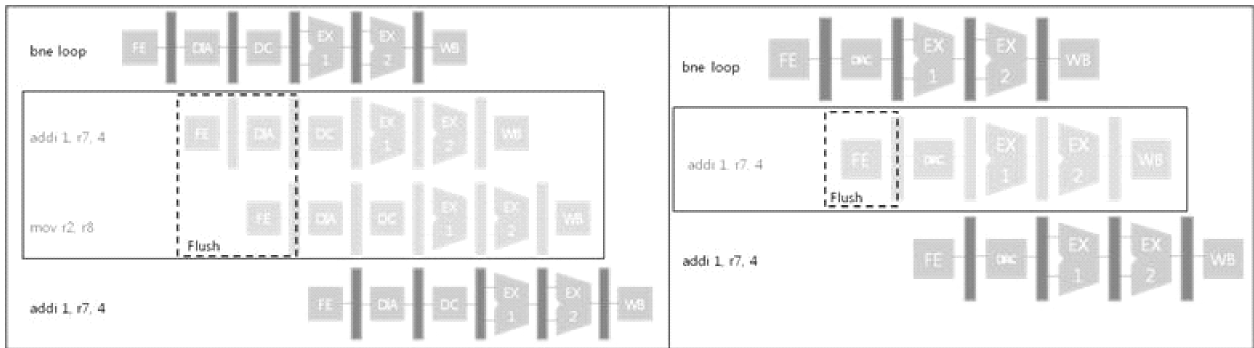
5.2 명령어 카운트의 최적화

새로운 아키텍처인 $DIAM_N$ 는 기본적인 아키텍처에서 DIAM만을 추가한 $DIAM_O$ 보다 클럭 타임이 줄어들 뿐만 아니라, 동적 명령어 카운트 또한 줄여준다. 동적 명령어가 줄어들게 되는 이유는 근본적으로 파이프라인이 한 단계 줄어들었기 때문이고, 구체적으로는 branch 와 jump 그리고call 과 같은 분기명령어와 관련 있다. (그림 14)는 분기명령어에 따른 두 가지 아키텍처($DIAM_O$, $DIAM_N$)의 수행을 비교하여 나타내었다.

분기명령어들은 분기가 될 때 파이프라인에서 실행되고 있는 연산들을 Flush 시키게 된다. (그림 14)에서 나타내는 것과 같이 $DIAM_O$ 에서는 FE 단계와 DIA 단계가 Flush 가 되는 반면에 $DIAM_N$는 FE 단계만 Flush 된다.



(그림 13) $DIAM_N$ 의 확장된 아키텍처



(그림 14) 분기명령어에 따른 파이프라인 비교

그렇기 때문에 분기 명령어에 의해서 분기가 되었을 때 $DIAM_N$ 가 $DIAM_O$보다 한 클럭 적게 카운트 되는 것이다. 이는 모든 분기 명령어에 적용이 되므로 총 명령어 카운트에 크게 영향을 미치게 된다.

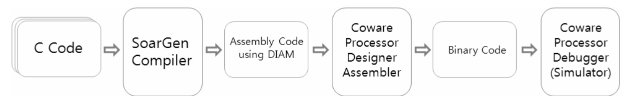
6. 실 험

이 장에서는 2-어드레스 아키텍처와 DIAM을 사용하는 아키텍처 ($DIAM_O$), 그리고 새로 제안된 아키텍처($DIAM_N$)의 성능을 비교할 것이다. 2-어드레스 아키텍처는 전용 어드레싱을 사용하지 않은2-어드레싱 모드 ISA 셋을 사용하고, $DIAM_N$ 과 $DIAM_O$ 는 DIAM과 함께 3-어드레싱 모드 ISA 셋을 사용한다. 우리들은 임베디드 시스템으로 널리 알려진 멀티미디어 벤치마크와 수의 연산 커널들에 대한 벤치마크의 실험을 수행하였다. <표 1>에서는 실험에 사용된 애플리케이션을 나타내었고, (그림 15)는 벤치마크 애플리케이션의 코드 생성 과정을 나타내었다.

벤치마크 애플리케이션은 (그림 15)에서 보여지는 것과 같이 C 소스코드를 SoarGen [22-23] 컴파일러에서 생성된 DIAM을 적용한 어셈블리 코드를 Coware사의 Processor Designer 어셈블러에 적용시킨다. 어셈블러를 통해 생성된

〈표 1〉 실험에 사용된 벤치마크 애플리케이션

Name	Description	Source
Compress	Image compression scheme by estimating the current cell based on the neighbors values, then stores the difference	Multimedia benchmarks
Linear	A general linear recurrence solver	
Wavelet	The Debaucles 4-Coefficient Wavelet filter to vector a[1..n] containing complex numbers	
LL5	Tri-diagonal elimination, below diagonal	Livermore Loops
LL17	Implicit, conditional computation	



(그림 15) 벤치마크 프로그램의 코드 생성 과정

바이너리 코드는 Coware Processor 디버거(시뮬레이터)를 통하여 시뮬레이션 하게 된다.

6.1 성능 비교

성능은 Processor Debugger와 Synopsys Design Compiler을 사용하여 측정된다. 먼저 Processor Debugger 를 사용하여 동적 명령어의 개수를 센다. 그런 다음 Design Compiler을 사용하여 정적 클럭 타이밍을 얻는다. 실제 실행 시간은 다음 식과 같이 계산된다 [19].

$$(Running\ Time) = (Dynamic\ Count * CPI) * (Clock\ Time)$$
$$CPI \approx 1$$

<표 2>에서는 각각의 명령어의 대한 동적 명령어 개수와 실제 실행시간을 나타내었다.

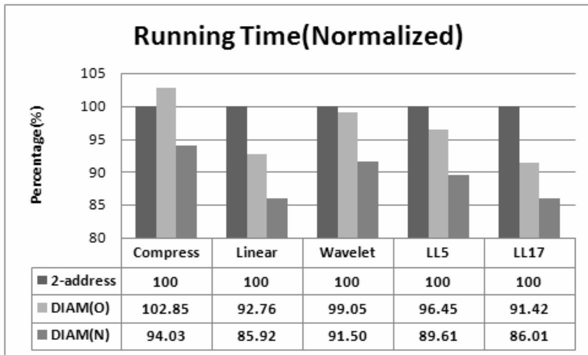
정규 결과는 2-어드레스 아키텍처에 비교되는 확장한 아키텍처들($DIAM_N$, $DIAM_O$)의 실행 시간 비율이다. <표 3>은 각 아키텍처에 대한 정규 결과를 나타낸다. $DIAM_N$에서의 모든 벤치마크의 성능이 좋아진 것을 확인할 수 있다. 심지어$DIAM_O$에서 더욱 나빠졌었던 Compress에서도 성능이 좋아졌다. $DIAM_O$은 평균 3.5%의 성능 향상에 비해 새롭게 고안된 아키텍처인 $DIAM_N$ 는 11.6%로 크게 성능이 향상되었다. (그림 16)에서는 정규 결과를 그래프

〈표 2〉 동적 명령어 개수와 실행시간 비교

Benchmark	Dynamic count			Running Time(ns)		
	2-address	$DIAM_O$	$DIAM_N$	2-address	$DIAM_O$	$DIAM_N$
Compress	1999	2002	1887	5197.4	5345.3	4887.33
Linear	662	598	571	1721.2	1596.6	1478.89
Wavelet	479	462	440	1245.4	1233.5	1139.6
LL5	428	402	385	1112.8	1073.3	997.15
LL17	820	730	708	2132	1949.1	1833.72

〈표 3〉 정규 결과

Benchmark	Normalized Time		
	2-address	DIAM$_O$	DIAM$_N$
Compress	100	102.85	94.03
Linear	100	92.76	85.92
Wavelet	100	99.05	91.50
LL5	100	96.45	89.61
LL17	100	91.42	86.01



(그림 16) 정규적인 성능 결과

〈표 4〉 DIAM 적용 명령어의 비율

Total Instruction Count	Instruction Count with DIAM(Count of 'NOP')	Percentage of Instruction with DIAM
80	15(20)	25%

로 나타내었다. 추가적으로 <표 4>는 DIAM$_N$ 에서 총 명령어 중에서 DIAM이 적용된 명령어의 비율을 나타내었다. 아무런 수행을 하지 않는 NOP 명령어의 개수를 빼면 약 22% 정도이다. 아직 완벽하게 최적화가 되지 않는 SoarGen Compiler가 업그레이드 된다면 30%~40%까지 적용이 될 것으로 예상된다.

6.2 클럭 타이밍

Processor Designer 로부터 생성된 HDL 코드는 Synopsis Design Compiler 를 사용하여 합성된다. <표 5>는 합성 후 얻어진 클럭 타이밍 정보를 나타낸다.

〈표 5〉 합성 후 클럭 타이밍 정보

	2-address		DIAMO		DIAMN	
Point	Incr.	Path	Incr.	Path	Incr.	Path
Clock clk_main(rise_edge)	2.5	2.5	2.5	2.5	2.5	2.5
	0	2.5	0	2.5	0	2.5
library Setup time	-0.05	2.45	-0.05	2.45	-0.07	2.43
data required time		2.45		2.45		2.43
data arrival time		2.61		2.66		2.59
slack(VIOLATED)		-0.16		-0.22		-0.16

합성 스크립트에clock_main 이2.5ns로 설정되어 있다. DIAM$_N$ 의 slack 값이 2-어드레스 아키텍처의 값과 동일하게 0.16ns 가 나오는데 이것은 DIA 와 DC 의 두 개의 파이프라인 단계를 합쳐도 전체 클럭 타이밍에는 영향을 끼치지 않는다는 것을 의미한다.

7. 결 론

이 논문에서 우리는 DSP를 위해 기술된 16 비트 기본적인 아키텍처와 DIAM을 사용하는 아키텍처인 DIAM$_O$ 를 설명하였고, DIAM$_O$ 의 성능을 더욱 향상시키기 위한 아키텍처인 DIAM$_N$ 를 제안하였다. DIAM$_O$ 확장 형태인 DIAM$_N$ 또한DIAM을 사용하므로 전용 어드레싱 없이 3-어드레스 코드를 사용할 수 있다. 우리의 실험에서 DIAM$_N$는 평균적으로 DIAM$_O$ 에 비해서 약 7%, 2-어드레스 아키텍처에 비해서는11.6% 성능 향상을 가져왔다. 추후 연구 과제로는 현재 연구를 기반으로 DIAM과 잘 어울리고 높은 질의 코드를 만들어내는 컴파일러 기술과 DIAM을 위해서 추가되는 메모리에 대한 하드웨어의 복잡성을 해결하는 것이다.

참 고 문 헌

[1] Xiaotong Zhuang, Tao Zhang, Santosh Pande, Hardware-managed register allocation for embedded processors, Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems.

[2] Motorola, Inc., Motorola DSP56300 Family Manual, Revision 3.0, Nov., 2000.

[3] Guido Araujo, Code generation algorithms for digital signal processors, PhD thesis, Princeton University, June 1997.

[4] Advanced RISC Machines Ltd., 'An Introduction to THUMB', March, 1995.

[5] SHRIVASTAVA, A., BISWAS, P., HALAMBI, A., DUTT, N., AND NICOLAU, A. 2006. Compilation framework for code size reduction using reduced bit-width isas. ACM Trans. Des. Autom. Electronic Syst.

[6] Muresan, R. Gebotys, C. A dynamic programming approach to complex allocation in a DSPpipelined processor, Electrical and Computer Engineering, 2001. Canadian Conference on.

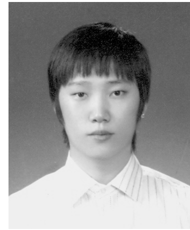
[7] Gary William Gréwal , Charles Thomas Wilson, Mapping reference code to irregular DSPs within the retargetable, optimizing compiler COGEN(T), Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture, December 01-05, 2001, Austin, Texas

[8] M. Ahn, Y. Paek, Fast Code Generation for Embedded Processors with Aliased Heterogeneous Registers. Transactions on High-Performance Embedded Architectures

and Compilers, 2(2):40-59, 2007.
[9] Freescale Semiconductor, Inc., “DSP56600 16-bit Digital Signal Processor Family Manual,” 1996.
[10] T. Wilson, et. al., “An ILP-based Approach to Code Generaton,” in Code Generation for Embedded Processors, ed. By P. Marwedel, G. Goossens, Kluwer Academic Publishers, 1995.
[11] ZSP 400 Digital Signal Processor Technical Manual, http://www.zsp.com
[12] CoWare INC. http://www.coware.com
[13] Infineon Technologies, *http://www.infineon.com*
[14] Motorola, *http://www.motorola.com*
[15] OpenCores org., http://www.opencores.org
[16] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr., LISA - Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures, *In Proc. of the Design Automation Conference (DAC), New Orleans, June, 1999.*
[17] Oliver Schliebusch, Andreas Hoffmann, Achim Nohl, Gunnar Braun and Heinrich Meyr, Architecture Implementation Using the Machine Description Language LISA, Design Automation Conference, 2002. *Proceedings of 7th Asia and South Pacific and the 15th International Conference on VLSI Design. Proceedings. (ASP-DAC 2002)*
[18] A. Hoffmann, A. Nohl, G. Braun, O. Schliebusch, T. Kogel, and H. Meyr., A Novel Methodology for the Design of Application Specific Instruction Set Processors (ASIP) Using a Machine Description Language, *IEEE Transactions on Computers-Aided Design(TCAD), Nov., 2001.*
[19] D. Patterson, J. Hennessy, Computer Organization and Design - The Hardware/Software Interface, Morgan Kaufmann Publishers, 2005.
[20] Synopsys Inc. http://www.synonpsys.com
[21] Y. Youn , D. Kim, M. Ahn, Y. Kim,Y. Paek, Orthogonal Instruction Encoding for a 16-bit Embedded Processor with Dynamic Implied Addressing Mode , *AHPCN2009 IEEE* (accepted)
[22] Soargen framework: Retargetable software toolkit generation, *LCTES, Jun 2005* Soargen & reconfigurable architecture, RWTH Aachen University of Technology, Jul 2006.

윤 종 희

e-mail : jhyoun@optimizer.snu.ac.kr
2003년 경북대학교 전자전기컴퓨터공학부(학사)
2003년～현 재 서울대학교 전기컴퓨터공학부 박사과정
관심분야 : 컴파일러, 임베디드 소프트웨어, 임베디드 시스템 개발도구, MPSoC



신 세 철

e-mail : a1702216@ee.knu.ac.kr
2007년 위덕대학교 컴퓨터공학과(학사)
2007년～현 재 경북대학교 전자전기컴퓨터공학부 석사과정
관심분야 : 임베디드 소프트웨어, 임베디드 시스템 개발도구, 시뮬레이터, MPSoC

백 윤 흥

e-mail : ypaek@snu.ac.kr
1988년 서울대학교 컴퓨터공학과(학사)
1990년 서울대학교 컴퓨터공학과(석사)
1997년 UIUC 전산과학(박사)
1997년～1999년 NJIT 조교수
1999년～2003년 KAIST 전자전산학과 부교수
2003년～현 재 서울대학교 전기컴퓨터공학부 교수
관심분야 : 임베디드 소프트웨어, 임베디드 시스템 개발도구, 컴파일러, MPSoC

조 정 훈

e-mail : jcho@ee.knu.ac.kr
1996년 KAIST 전기및전자공학과(학사)
1998년 KAIST 전기및전자공학과(석사)
2003년 KAIST 전자전산학과 전기 및 전자공학전공(박사)
2003년～2005년 하이닉스반도체 선임연구원
2005년～현 재 경북대학교 전자전기컴퓨터학부 조교수
관심분야 : 임베디드 소프트웨어, 임베디드 시스템 개발도구, 컴파일러, MPSoC