Journal of The Korea Society of Computer and Information
Vol. 17, No. 11, November 2012
www.ksci.re.kr
http://dx.doi.org/10.9708/jksci/2012.17.11.001

AMEX: 16비트 Thumb 명령어 집합 구조의 주소 지정 방식 확장

김 대 환*

AMEX: Extending Addressing Mode of 16-bit Thumb Instruction Set Architecture

Dae-Hwan Kim*

요 약

본 논문에서는 16비트 Thumb 명령어 집합 구조를 개선하기 위하여 주소 지정 방식을 확장하는 기법을 제시한다. 제시된 방법의 핵심 아이디어는 사용 빈도가 낮은 명령어들의 레지스터 필드의 너비를 감소시키고 이를 통해 절약한 비트들을 이용하여 사용 빈도가 높은 명령어들에 새로운 주소 지정 방식을 도입하는 것이다. 제시된 기법은 16비트 Thumb 구조의 상위 집합인 32비트 ARM 구조에서 사용되는 유용한 주조 지정 방식들을 채택한다. 데이터 리스트에 대한 접근 속도를 향상시키기 위하여 크기가 조정된 레지스터 오프셋 주소 지정 방식과 사후 인덱스 주소 지정 방식이 로드와 저장 명령어에 도입된다. 실험 결과, 제시된 방법은 전통적인 방식과 비교하여 평균 8.5%의 성능을 향상시킨다.

▸ Keywords : 명령어 집합 설계, 주소 지정 방식, 임베디드 프로세서, Thumb, ARM

Abstract

In this paper, the extension of the addressing mode in the 16-bit Thumb instruction set architecture is proposed to improve the performance of 16-bit Thumb code. The key idea of the proposed approach is the introduction of new addressing modes for more frequent instructions by using the saved bits from the reduction of the register fields in less frequently used instructions. The proposed approach adopts efficient addressing modes from the 32-bit ARM architecture, which is the superset of the 16-bit Thumb architecture. To speed up access to a data list, scaled register offset addressing mode and post-indexed addressing mode are introduced for load and store instructions. Experiments show that the proposed approach improves performance by an average of 8.5% when compared to the conventional approach.

▸ Keywords : Instruction set design, Addressing mode, Embedded processor, Thumb, ARM

• 제1저자 : 김대환 • 교신저자 : 김대환
• 투 고 일 : 2012. 07. 26, 심사일 : 2012. 09. 05, 게재확정일 : 2012. 10. 15.
* 수원과학대학교 컴퓨터정보과 (Dept. of Computer Information, Suwon Science College)

I. 서 론

임베디드 시스템에서는 종종 메모리의 크기가 제한되기 때문에 프로그램의 코드 크기를 감소하는 것이 중요하다. 압축된 명령어 집합 구조(instruction set architecture)는 이러한 코드 크기 문제의 하나의 해결책이 되었고, ARM/Thumb [1]이나 MIPS/MIPS16 [2]와 같은 이중 명령어 집합 구조(dual instruction set architecture) 프로세서들은 32비트의 정규 명령어 집합과 16비트의 압축된 명령어 집합 모두를 실행할 수 있으며 이때 16비트 명령어 집합은 실제로는 자주 사용되는 32비트 명령어들의 부분집합이다.

ARM 프로세서는 가장 널리 사용되는 32비트 임베디드 프로세서로 2011년에는 약 80억 개의 출시된 칩이 ARM 구조에 기반을 두고 있다[3]. Thumb 구조는 1994년 처음 등장하였으며 ARM7TDMI, ARM926EJ-S, XScale, ARM1176JZ(F)-S 등 많은 프로세서에서 여전히 지원되고 있다. 대표적인 Thumb 모드 지원 프로세서 중의 하나인 ARM7TDMI는 2011년에 35억 개 이상이 출시되는 등 처음 등장 이후 현재까지 100억 개 이상의 칩이 출하되고 있다[3].

압축된 16비트 Thumb 명령어는 32비트 ARM 명령어로 동적으로 확장된다[1]. 이러한 압축 해제는 명령어 디코더 단계에서 수행되기에 실행 단계에서는 추가적인 하드웨어가 불필요하다. 이렇게 함으로써 압축 해제는 간단하게 사이클(cycle) 손실 없이 수행된다. 그림 1은 Thumb 명령어가 하드웨어 명령어 디코더에 의해 해당 32비트 ARM 명령어로 변환되는 예를 보여준다. '00110|001|00101'로 인코딩된 'ADD r1, #5'의 Thumb 명령어가 ARM 'ADD r1, r1, #5' 명령어의 '1110|00101001|0001|0001|00000101' 비트열로 변환된다. 2003년에 Thumb-2 구조[4]가 제시되었는데 이 구조는 전통적인 16비트 Thumb 명령어와 추가적인 32비트 ARM 명령어를 하나의 명령어 집합으로 결합한다.

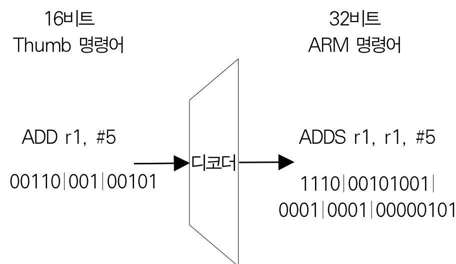


그림 1. Thumb 명령어 디코딩 예
Fig. 1. Thumb instruction decoding example

16비트 공간에 명령어들을 인코딩하기 위하여 16비트 Thumb 명령어 형식에는 여러 가지 제한이 가해진다. 접근 가능한 레지스터(register)의 수, 피연산자(operand)의 개수, 주소 지정 방식(addressing mode), 즉시 필드(immediate field) 너비(width) 등을 저감한다. ARM 구조에서는 16개의 범용 레지스터가 사용 가능한 반면에 Thumb의 대부분의 명령어들은 단지 8개의 범용 레지스터만을 사용할 수 있다. 이로 인해 Thumb 코드에서는 레지스터에 할당되지 못한 변수를 메모리에서 접근하게 된다. 또한 주소 지정 방식이 제한되기에 유효 주소(effective address)를 계산하기 위한 추가적인 명령어가 종종 필요하게 된다. 이러한 제한들은 16비트 Thumb 형식으로 변환될 수 있는 32비트 ARM 명령어의 수를 제한하게 되고 이는 결국 Thumb의 압축 효율과 성능 두 가지 모두를 감소시키게 된다. 32비트 ARM 코드와 비교하여 16비트 Thumb의 압축 효율은 30% 정도 향상되지만 실행 속도는 20~25% 정도 감소된다[1].

16비트 Thumb 명령어 집합 구조를 개선하기 위해 다양한 기법들[5-9]이 제시되어 왔다. 대부분의 방법은 가용 레지스터의 수를 증대시키는 기법에 초점을 맞추거나[5-7], 코드 크기 감소를 위해 즉시 필드(immediate field) 할당 비트 수를 증가시킨다[8]. 하지만 명령어 집합 구조에서 중요한 고려 사항 중 하나인 주소 지정 방식의 확장을 통한 Thumb 명령어 집합 구조 개선에는 관심을 두지 않았다.

16비트 Thumb 명령어 집합 구조의 성능과 압축 효율을 모두 향상시키기 위해, 본 논문에서는 AMEX (Addressing Mode EXtension)라 불리는 새로운 명령어 집합 구조를 제시하는데 이 구조에서는 32비트 ARM 구조로부터 효율적인 주소 지정 방식을 채택함으로써 코드 크기 증가 없이 성능을 향상시키는 방법을 제시한다. 제시된 방식의 핵심 아이디어는 사용 빈도가 낮은 명령어의 레지스터 필드에 사용되는 비트 수를 줄이고 이를 통해 절약된 비트들을 이용하여 사용 빈도가 높은 명령어들에 효율적인 주소 지정 방식을 추가로 도입하는 것이다. 제시된 기법을 적용하면 기존의 16비트 Thumb 명령어 집합을 사용할 때 보다 응용 프로그램의 실행 속도가 평균적으로 8.5% 빨라지는 의미 있는 결과를 가져온다. 기존의 Thumb 명령어 집합 구조에서는 ARM 명령어 집합보다 속도가 20 ~ 25% 정도 저하되는데 그 격차를 1/3가량 감소시킬 수 있게 된다.

본 논문의 구성은 다음과 같다. II장에서는 관련 연구를 기술하고, III장에서는 새로운 주소 지정 방식의 확장 기법들을 제시한다. IV장에서는 제시된 기법의 성능 향상을 평가하며, 마지막으로 V장에서 결론을 맺는다.

II. 관련 연구

1. 16비트 Thumb 명령어 집합 확장

프로세서의 명령어 집합을 확장함으로써 성능을 개선하는 다양한 연구들이 진행되고 왔으며[5-22] 대표적인 임베디드 프로세서인 ARM의 16비트 Thumb 명령어 집합 구조를 개선하기 위해 다양한 기법들이 제시되어 왔다[5-9]. [5], [6], [7]과 같은 기법들은 가용 레지스터의 수를 늘림으로써 성능 향상을 도모한다. 반면에 [8]과 같은 기법에서는 코드 크기를 더욱 감소하고자 한다.

Krishnaswamy와 Gupta[5]는 Thumb 코드에서 보이지 않는 레지스터를 사용하는 방법을 제안한다. 대부분의 Thumb 명령어들은 물리적으로 존재하는 16개의 레지스터 중 단지 8개의 레지스터만을 접근할 수 있다. 모든 명령어들이 모든 레지스터를 사용할 수 있도록 하기 위하여, 16개 레지스터 중 가용 레지스터 8개를 동적으로 지정할 수 있는 특수 명령어가 도입된다. 이를 통해 가용 레지스터의 수가 증가됨으로써 성능이 향상된다. 하지만 레지스터 부분집합을 변경하기 위한 특수 명령어가 많이 발생하게 되어 코드 크기가 증가하는 단점이 있다. [6]에서는 이 기법을 명령어 집합에 확대 적용한다. Thumb 코드에서는 하나의 ARM 명령어로 변환될 수 있는 Thumb 명령어 쌍이 존재한다. 이 정보를 기술해주는 보강(augmented)된 명령어를 추가로 지원하여 해당 명령어 쌍을 보강 명령어가 추가된 명령어 쌍으로 변경하고 명령어 디코더에서는 이 명령어 쌍을 하나의 ARM 명령어로 변환한다. 이 기법에서는 추가되는 보강 코드로 인하여 코드 크기가 증가한다. 또한 보강 코드와 연관되는 Thumb 명령어 쌍을 통합(coalesce)하기 위한 추가적인 하드웨어가 필요하다.

[7]에서는 레지스터 파일을 뱅크(bank) 구조로 재구성하고 뱅크 전환 명령어를 제공함으로써 모든 레지스터 파일을 접근 가능하게 한다. 이 구조를 효과적으로 이용하기 위한 영역 기반 레지스터 뱅크 할당 기법이 추가로 제시된다.

[8]에서는 코드 크기를 더욱 감소하는 데에 주안점을 둔다. 이를 위해 일부 명령어의 목적지 레지스터 (destination register) 필드를 축소하고 이 비트를 즉시 필드에 추가로 할당한다. 감소된 레지스터 파일을 효율적으로 이용하기 위해 명령어 유형에 따라 레지스터 파일을 분할하는 방법과 이 구조에 적합한 레지스터 할당(register allocation) 기법을 제시한다. 하지만, 제시된 기법은 작은 벤치마크 프로그램들에서만 성능 평가가 수행되었기에 다양한 실제적인 프로그램에서의 추가적인 성능 평가가 필요하다.

[9]에서는 Thumb 명령어 집합에 스레드(thread) 전환 명령어가 추가한다. 디코더의 지원을 통해, 제시된 명령어의 처리에 하드웨어 사이클이 추가적으로 필요하지 않게 된다. 제시된 방법은 하드웨어의 추가를 최소화하면서 스레드 처리 성능을 개선한다.

[10]에서는 기존의 Thumb 코드의 압축 효율을 더욱 향상시키기 위해 명령어 집합을 개선하기 보다는 Thumb 코드 데이터에 대해 고속 데이터 비손실 하드웨어 압축 해제기(Decompressor)를 사용한다. 제시된 기법은 커다란 시간적 오버헤드 없이 기존의 Thumb 코드의 압축 효율을 15%가량 개선한다.

주소 지정 방식 확장을 통한 Thumb 명령어 집합 구조를 개선에는 상대적으로 적은 관심이 기울여져왔다. 저자가 알고 있는 바로는 현재까지 알려진 방법은 없다. Thumb 구조와는 별개로 Fiskiran 등[11]은 주소 지정 방식의 중요성을 기술하고 AES 알고리즘을 위한 효율적인 주소 지정 방식을 제시하였으나 상당한 분량의 하드웨어를 요한다.

2. ARM 명령어 집합 확장

일부의 기법들은 ARM 명령어 집합 구조에 초점을 맞춘다[12-20]. [12]는 미디어와 네트워크 처리 등의 임베디드 영역에서 흔히 요구되는 워드 내부에서의 비트 선택 연산들은 효율적으로 처리하기 위해 워드보다 낮은 단계의 데이터를 처리할 수 있는 비트 선택 확장(Bit Selection Extension) 기법을 도입한다. [13-14]는 ARM 구조에서 사용 빈도가 낮은 조건 필드(conditional field) 비트들을 레지스터 기술에 추가로 할당함으로써 가용 레지스터의 수를 두 배로 증가시킨다. 이로 인해 ARM 명령어는 조건부 실행이 불가능해져서 사소하게 성능이 저하되지만 가용 레지스터의 수가 증가되어 전체적인 성능이 향상된다.

[15]에서는 타원 곡선 암호(Elliptic Curve Cryptography) 분야에서 ARM 명령어 집합의 성능을 평가하고 명령어 집합의 병목(bottleneck) 요인을 분석한다. 이를 기반으로 ARM 명령어 집합을 확장하는 기법을 제시한다. 워드 단계에서의 곱하기 연산을 사용함으로써 이 특정 분야의 응용 프로그램의 성능을 33% 개선한다.

ARM DSP 명령어 확장[16]은 신호 처리 알고리즘의 실행 속도를 개선하고자 새로운 DSP 명령어를 ARM 명령어 집합에 추가한다. 이 명령어 집합은 ARM926EJ-S, ARM946E-S, ARM966E-S 등의 프로세서에 포함되며 Audio 응용 프로그램의 경우 70%까지 속도를 개선한다.

2000년에 소개된 ARMv5EJ 구조는 Java를 위한 Jazelle Java 실행 확장 가속기[17]를 포함하여 ARM 프로세서가 Java 바이트 코드를 직접 실행 가능하게 한다. 이러한 명령어 집합을 통해 ARM이 소프트웨어 기반 자바 가상 기계(Java virtual machine)를 실행할 때 보다 8배의 속도가 향상되며 소비전력도 80%가 감소된다. ARM926EJ-S, ARM1026EJ-S, ARM1136J(F)-S, ARM1176J(F)-S 등의 프로세서가 자바 명령어 집합을 포함한다.

2002년에 처음 소개된 ARMv6 구조[18]는 SIMD (Single Instruction Multiple Data) 명령어 집합을 지원하며 이 명령어 집합은 두 개의 16비트 또는 4개의 8비트 산술 연산을 동시에 실행할 수 있게 하는 등 하나의 연산을 여러 개의 데이터에 대해 병렬적으로 수행할 수 지원한다. ARM11 프로세서 등에 포함되었으며 오디오 및 비디오 코덱, 이미지와 음성 처리, 그래픽과 일반전인 신호 처리 등의 프로그램들에 유용하게 적용된다. 개선된 SIMD 집합이 ARMv7 구조에 NEON [19]이라는 이름으로 확장된다. 이 구조는 ARM SIMD을 개선하여 비디오, 오디오 코덱과 같은 응용에서 속도가 4배 정도 개선되며 Cortex-A9 등의 프로세서에 포함된다.

ARM 개발팀은 병렬성(parallelism)을 향상시키기 위해 다양한 기법들을 제시한다[20]. 그 기법들은 변동 실행 시간(variable execution time), 서브워드 병렬성(subword parallelism), DSP유사한 연산(DSP-like operations), 스레드 단위 병렬성(thread-level parallelism), 멀티프로세싱(multiprocessing) 등을 포함한다.

III. 명령어 집합 설계

1. 제시된 주소 지정 방식 확장

프로그램의 실행 속도를 향상시키기 위해서는 메모리 주소 계산을 효율적으로 수행하는 것이 중요하다. 이를 위해 대부분의 프로세서는 다양한 주소 계산 방식을 지원한다. ARM 구조는 오프셋 주소 지정 방식(offset addressing mode)을 지원하는데 이 방식은 유효 주소(effective address)를 기준 레지스터(base register)의 값에 오프셋 값을 더한 값으로 계산한다. 오프셋 값은 즉시(immediate), 레지스터(register), 크기 조정된 레지스터(scaled register)의 세 가지 방식으로 명시될 수 있다. 이 중에서 크기 조정된 레지스터 오프셋 방식은 레지스터의 값을 배열의 요소(element) 크기 단위로 조정할 수 있기에 배열이나 벡터를 접근할 때 유용하다.

또 다른 유용한 주소 지정 방식인 사후 인덱스 주소 지정 방식(post-indexed addressing mode)도 제공되는데 여기에서는 기준 레지스터의 주소가 유효 주소로 사용되며 그리고 나서 그 레지스터 값은 다음 메모리 참조 주소로 갱신된다. 이 방식에서는 메모리 데이터 이동과 메모리 주소 갱신이 하나의 명령어에서 동시에 수행된다. 배열의 각 원소들은 메모리에서 인접한 위치에 저장되기 때문에 루프(loop) 내에서 배열이 참조될 때 그 주소가 루프 반복마다 일정 값 만큼 증감하는 경우가 대부분이다. 따라서 사후 인덱스 주소 지정 방식을 이용하면 메모리 주소 변경을 위한 추가적인 명령어가 불필요해지고 이는 프로그램의 루프를 최적화하기에 유용하다.

그림 2는 사후 인덱스 주소 지정 방식을 이용한 프로그램 코드를 보여준다. 그림 2 (a)는 원본 C 프로그램을, 그림 2 (b)는 사후 인덱스 주소 지정 방식이 적용된 어셈블리 코드를 보여준다. 그림 2 (a)에서 int 타입의 배열 A의 오프셋 값은 루프 반복마다 4만큼 증가하기 때문에, 그림 2 (b)에서와 같이 사후 인덱스 주소 지정 방식인 STR R1, [R2], #4 명령어를 이용하여 주소 계산 연산을 최적화한다. 이 명령어는 레지스터 R2의 값을 주소로 하는 메모리에 레지스터 R1의 값을 저장한 후에 R2의 값을 4만큼 증가시킨다. 그 결과, 레지스터 R2는 참조되는 배열 A의 요소(element) 주소를 계속적으로 가리키게 된다. 그림 2 (a)의 루프는 변수 i의 값이 10일 때 종료된다. 이때 배열 참조 주소는 그림 2 (b)에서와 같이 A + 40이다. 배열 A의 시작 주소에 종료 오프셋 값 40을 더하는 ADD 명령어는 루프에 무관하므로 루프 밖의 두 번째 줄로 이동한다.

```
int A[10];

for (i = 0; i < 10; i ++)
    A[i] = 0;
```

(a) C 코드

```
LDR   R2, #배열 A의 시작 주소
ADD   R3, R2, #40
MOV   R1,#0
|loop1|
STR   R1, [R2], #4
CMP   R2, R3
BLT   |loop1|
```

(b) 어셈블리 코드

그림 2. 사후 인덱스 주소 지정 방식 예
Fig. 2. Post-indexed addressing mode example

Thumb 명령어 집합 구조 설계자들은 명령어 연산코드(opcode), 레지스터와 같은 다양한 명령어 필드와 주소 지정 방식간의 상충 관계(trade-off)에서 크기 조정된 레지스터 오프셋 주소 지정 방식(scaled register offset addressing mode)과 사후 인덱스 주소 지정 방식(post-indexed addressing mode)을 희생하였다. 그 결과, Thumb에서는 주소 값을 증가하거나 오프셋 크기를 조정하기 위하여 추가적인 덧셈 또는 시프트 명령어가 종종 필요하게 된다.

임베디드 프로그램에서는 없어진 두 가지 주소 지정 방식이 일부 명령어들보다 더 빈번하게 사용되기 때문에 이 결정이 최적이 아닐 수 있다. 이 두 방식은 배열이나 데이터 리스트를 접근하는 데에 유용함을 상기하기 바란다. Thumb에는 두 종류의 블록 전송 명령어가 있어서 PUSH (Push registers)와 POP (Pop registers) 명령어들은 함수 진입과 복귀 코드에 주로 사용되고, LDMIA (Load multiple)과 STMIA (Store multiple) 명령어들은 메모리 블록 복사를 위해 많이 사용된다. LDMIA 명령어는 메모리로부터 여러 개의 데이터를 범용 레지스터로 로드하고 STMIA 명령어는 여러 레지스터들의 데이터를 메모리로 저장한다. 두 명령어는 한 번에 최대 8개의 데이터를 메모리와 레지스터 사이에 전송할 수 있는데 실제로 8개의 레지스터를 동시에 전송하는 일은 드물다. 왜냐하면 대부분의 Thumb 명령어들은 단지 8개의 레지스터만을 접근할 수 있어서 레지스터들 중 상당수는 일반적으로 다른 용도로 사용되기 때문이다. 이러한 관찰 결과를 반영하여, 제시된 기법은 LDMIA와 STMIA와 같이 사용 빈도가 낮은 몇 개의 명령어들에서 접근 가능한 레지스터의 수를 줄이고, 절약된 비트들을 활용해 제외된 두 주소 지정 방식을 다시 복구하는 결정을 내린다.

Thumb에서 사용되지 않는 이용 가능한 연산코드(opcode)는 거의 없다. 따라서 새로운 주소 지정 방식에 대한 인코딩 공간을 생성하기 위하여, 제시된 방식은 몇 개의 명령어들에서 레지스터 필드에 사용되는 비트 수를 저감한다. 이러한 변경은 사용 빈도가 가장 낮은 명령어들에 국한되기에 프로그래머에게 부담을 주지는 않으리라 기대된다. 또한, Thumb 명령어 집합 구조는 컴파일러를 대상으로 설계되어서 이미 직교적(orthogonal)이지 않다는 사실에 유의하자[1].

표 1은 AMEX에서 제한되는 Thumb 명령어와 제한 내용을 보여준다. 자주 사용되지 않으면서도 새로운 주소 지정 방식을 내장할 수 있는 충분한 피연산자 비트 수를 가지는 세 명령어가 선택된다. 그 명령어는 ADD immediate to PC (Program Counter), LDMIA (Load multiple), STMIA (Store multiple) 이다. ADD immediate to PC 명령어는 보통 ADR (ADdress to Register)로 표시되는데, 즉시값을 PC 값에 더하고 그 결과를 목적지 레지스터(destination register)에 저장한다. LDMIA 와 STMIA 명령어는 메모리와 범용 레지스터들 사이에 여러 개의 데이터를 로드하고 저장한다. 제시된 방법은 각 명령어에서 목적지 레지스터 필드나 레지스터 리스트 필드를 한 비트 감소시켜 이를 이용한다. ADR 명령어에서는 즉시 필드를 8비트에서 7비트로 저감하며, LDMIA와 STMIA 명령어에서는 레지스터 R7을 가능한 전송 리스트에서 제외하여서 전송 가능 레지스터 리스트를 R0~R7에서 R0~R6로 축소한다. 이 비트는 본 논문에서는 X 비트라 불리는데, 축소된 Thumb 명령어 (X 비트=0)와 새로운 AMEX 명령어(X 비트=1)를 구분하는 데에 사용된다.

표 1. 제한되는 Thumb 명령어
Table 1. Reduced thumb instruction

명령어	설 명	제한사항
ADR Rd,#imm	Rd ← Rd+ imm	imm 필드를 8비트에서 7비트로 제한
LDMIA Rn!, 〈register_list〉	Rn 값을 시작 주소로 메모리 블록을 레지스터로 로드	register list를 R0 ~ R7 에서 R0 ~ R6 으로 제한
STMIA Rn!, 〈register_list〉	레지스터들을 Rn 주소의 메모리에 저장	

이러한 축소에 의한 성능 저하는 해당 명령어들의 사용 빈도가 낮기에 심각하지 않다. 그림 3은 축소된 세 명령어의 실행 빈도를 보여준다. 사용한 벤치마크는 jpeg, mpeg, 164, susan, pegwit, adpcm, blowfish 프로그램이다. 표 2는 사용된 벤치마크 프로그램을 기술한다. 정지 이미지 코덱인 jpeg, 비디오 압축 코덱인 mpeg, 데이터 압축 프로그램인 164, 이미지 노이즈 필터 프로그램인 susan, 음성 코덱인 adpcm과 데이터 암호화 프로그램인 pegwit 및 blowfish를 대상으로 한다. 각 프로그램에 대해, ARM Realview 컴파일러 툴 체인이 Thumb 이진 코드(binary code)를 생성하고 ARM Realview 시뮬레이터가 총 동적 실행 사이클과 세 명령어의 사이클을 각각 측정한다. 측정 결과, 평균적으로 ADR, LDM, STM 명령어는 각각 총 실행 사이클의 단지 0.0007%, 0.0321%, 0.0005%를 차지한다.

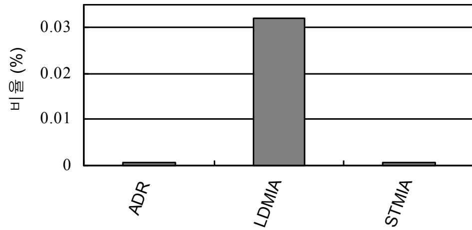


그림 3. ADR, LDMIA, STMIA 명령어의 실행 빈도
Fig. 3. The frequency of the ADR, LDMIA, and STMIA instructions

AMEX의 Thumb 확장은 새로운 세 가지 주소 지정 방식, ADD 명령어를 위한 하나와 로드와 저장 명령어를 위한 두 가지로 구성된다. 하나는 왼쪽으로 시프트된 레지스터 피연산자(left-shifted register operand) 방식으로 주소 계산에 널리 사용되는 ADD 명령어에 도입된다. 나머지 둘은 크기가 조정된 레지스터 오프셋 주소 지정 방식(scaled register offset addressing mode)와 즉시 피연산자 사후 인덱스 주소 지정 방식(immediate post-indexed addressing mode)이다.

표 2. 벤치마크 프로그램
Table 2. Benchmark program

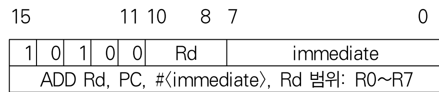
벤치마크	설명
jpeg	이미지 압축 표준
mpeg	동영상 압축 표준
164	GNU gzip 데이터 압축
susan	이미지 노이즈 필터
pegwit	공개키 암호화 및 인증
adpcm	음성 인코딩 및 디코딩
blowfish	대칭적 블록 암호화

표 3은 AMEX에 새로 도입되는 주소 지정 방식을 보여준다. 제한된 인코딩 공간 때문에 AMEX는 각 주소 지정 방식 중 가장 대표적인 유형만을 포함한다. 크기 조정된 레지스터 오프셋 주소 지정 방식에서는 왼쪽, 오른쪽 논리 시프트, 오른쪽 산술 시프트, 오른쪽 회전(rotate) 연산 중에서 왼쪽 시프트를 선정한다. 이 주소 지정 방식은 LDR (Load word)과 STR (Store word) 명령어들에 채택된다. 마찬가지로 즉시, 레지스터, 크기 조정된 레지스터 오프셋의 세 가지 사후 인덱스 주조 지정 방식 중에서는 즉시가 선택된다. 이 주소 지정 방식은 LDRB (Load byte), LDRSB (Load signed byte), LDRH (Load halfword), LDRSH (Load signed halfword), LDR (Load word), STRB (Store byte), STRH (Store halfword), STR (Store word) 명령어들에 채택된다.

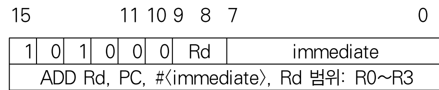
표 3. AMEX의 새로운 주소 지정 방식
Table 3. AMEX new addressing mode

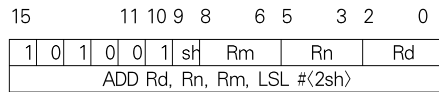
주소 지정 방식	관련 명령어
왼쪽으로 시프트된 레지스터 피연산자	ADD
왼쪽으로 시프트된 레지스터 오프셋 주소 지정 방식	LDR, STR
즉시 사후 인덱스 주소 지정 방식	LDRB, LDRSB, LDRH, LDRSH, LDR, STRB, STRH, STR

2. AMEX 명령어 인코딩



Thumb



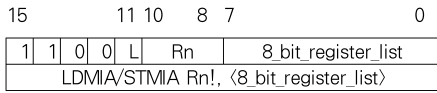


AMEX

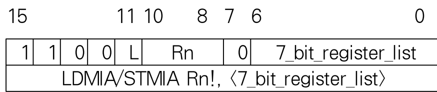
그림 4. AMEX 명령어 집합 설계: Thumb ADR 명령어 형식과 이 형식을 이용한 AMEX 명령어 형식
Fig. 4. AMEX instruction set design: Thumb ADR instruction format and its corresponding AMEX instruction formats

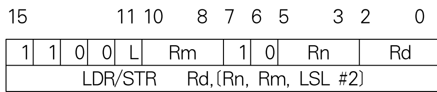
그림 4는 AMEX에서 왼쪽으로 시프트된 레지스터 피연산자 형식의 ADD 명령어를 인코딩하기 위해 Thumb의 ADR (ADD immediate to PC) 명령어 형식을 변경한 것이다. 이러한 형태의 ADD 명령어는 주소 계산에 널리 사용되지만 16비트 Thumb 명령어 집합 구조에 포함되어 있지 않다. 제시된 기법에서는 ADR 명령어 형식에서 목적지 레지스터 필드의 너비를 3에서 2로 감소시켜서 이 필드의 최상위 비트(비트 10)을 절약한다. AMEX의 ADR 명령어에 사용되는 목적지 레지스터의 범위는 기존의 R0~R7에서 R0~R3으로 축소된다. 이 비트가 X 비트로 사용되며 ADR 명령어에 대해서는 0, 새로운 AMEX 명령어에 대해서는 1의 값을 가진다. 다양한 시프트 유형과 양 중에서 16비트와 32비트 배열을 접근하는 데에 각각 유용한 1비트와 2비트 왼쪽 논리 시프트 (LSL: Logical Shift Left) 연산을 지원한다. 이제 새로 도입되는 명령어 형식을 살펴보자. 비트 0부터 2, 비트 3부터 5, 비트 6에서 8은 각

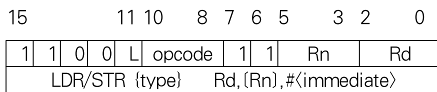
각 목적지 레지스터, 첫 번째 레지스터, 그리고 두 번째 레지스터를 인코딩한다. 비트 9는 시프트 양을 결정한다. 이 비트가 0이면 그 양은 1로, 아닌 경우는 2로 정의된다.



Thumb







연산 코드	데이터 타입		즉시값
	L = 0 (저장)	L = 1 (로드)	
0	바이트	바이트	1
1	-	부호있는 바이트	1
2	하프워드	하프워드	2
3	-	부호있는 하프워드	2
4	워드	워드	4
5-7	-	-	-

AMEX

그림 5. Thumb LDMIA/STMIA 명령어 형식과 이 형식을 이용한 AMEX 명령어 형식
Fig. 5. Thumb LDMIA/STMIA instruction format and its corresponding AMEX instruction formats

그림 5는 LDMIA (L 비트=1)와 STMIA (L 비트=0) 명령어에 대한 변경이다. 두 명령어에서 레지스터 R7은 전송 가능한 레지스터에서 제외되어서, 전송 가능한 레지스터 리스트가 R0~R7에서 R0~R6로 축소된다. R7에 해당하는 비트(비트 7)이 X 비트로 사용된다. 절약된 비트를 이용하여 두 가지 새로운 방식인 시프트된 레지스터 오프셋 주조 지정 방식(shifted register offset addressing mode)와 즉시 사후 인덱스 주소 지정 방식(immediate post-indexed addressing mode)가 로드와 저장 명령어에 도입된다. A 비트(비트 6)는 시프트된 레지스터 오프셋 주소 지정 방식(A 비트=0)과 사후 인덱스 주소 지정 방식(A 비트=1)을 구분한다.

먼저 시프트된 레지스터 오프셋 주소 지정 방식을 살펴보자. AMEX는 시프트 연산을 2비트 왼쪽 시프트로 제한하고 이 주소 지정 방식을 지원하는 명령어로 LDR (Load word)과 STR (Store word)을 채택한다. L 비트는 LDR (L 비트=1)과 STR (L 비트=0) 명령어를 구분한다. 버전 4와 같은 전통적인 ARM 구조에서는 바이트와 워드 데이터 타입에 대하여 이러한 시프트된 레지스터 오프셋 주소 지정 방식을 지원하며 32비트 워드 배열의 요소를 접근하는 데에 유용한 2비트 왼쪽 시프트가 가장 빈번하다. 이 주소 지정 방식은 목적지 레지스터, 베이스 레지스터, 오프셋 레지스터의 세 개의 피연산자를 요구한다. 비트 0에서 2, 비트 3에서 5, 비트 8에서 10은 각각 목적지 레지스터, 베이스 레지스터, 오프셋 레지스터를 인코딩한다.

두 번째는 즉시 사후 인덱스 주소 지정 방식이다. 이 방식과 결합하여 바이트(byte), 하프워드(halfword), 워드(word) 데이터 타입의 로드와 저장 명령어들인 LDRB (Load byte), LDRSB (Load signed byte), LDRH (Load halfword), LDRSH (Load signed halfword), LDR (Load word), STRB (Store byte), STRH (Store halfword), STR (Store word)가 도입된다. 이 주소 지정 방식은 베이스 레지스터의 값을 즉시값(immediate)만큼 증가시키며, 가장 빈번한 값들은 1, 2, 4로 각각 바이트, 하프워드, 워드 타입의 배열을 접근하는 데에 유용한다. 이 주소 접근 방식은 목적지 레지스터, 베이스 레지스터를 필요로 한다. 비트 0부터 2, 비트 3부터 5는 각각 두 개의 레지스터를 인코딩하며 비트 8부터 10의 3비트는 연산부호를 인코딩하는데 이는 연산 데이터 종류와 즉시값 모두를 명시한다.

IV. 성능 평가

제시된 방법의 효율성을 측정하기 위하여 FaCSim ARM 시뮬레이터[23]에서 ARM9TDMI 프로세서를 대상으로 성능을 평가한다. 시뮬레이터는 새로운 AMEX 명령어 집합을 지원하기 위해 확장된다. 성능을 평가하기 위해, 임베디드 시스템에서 널리 사용되는 표 2의 jpeg, mpeg, 164, susan, pegwit, adpcm, blowfish 프로그램을 벤치마크로 사용한다. 각 프로그램에 대해 ARM Realview 컴파일러로 16비트 Thumb 명령어를 생성한다. 생성된 어셈블리 명령어 쌍 중에서 AMEX의 주소 지정 방식으로 변경될 수 있는 코드 패턴을 좁은 지역 범위에서 찾아내어 변경한다. 변경 전 Thumb 코드와 변경 후 AMEX 코드의 성능을 FaCSim ARM 시뮬레이터에서 측정한다.

그림 6은 16비트 Thumb 명령어 집합 구조와 비교하여 제시된 기법인 AMEX의 성능 효율을 보여준다. 7개의 벤치마크 프로그램에 대해 제시된 방식은 Thumb 명령어 집합 구조와 비교하여 평균 8.5%의 성능 향상을 보여준다. blowfish 프로

그램 의 경우 성능(17.8%)이 가장 향상된다. 이는 해당 프로그램이 AMEX에서는 왼쪽으로 시프트된 ADD (left-shifted ADD) 명령어 하나로 결합될 수 있는 LSL (Logical Shift Left)과 ADD 명령어 쌍을 많이 사용하기 때문이다.

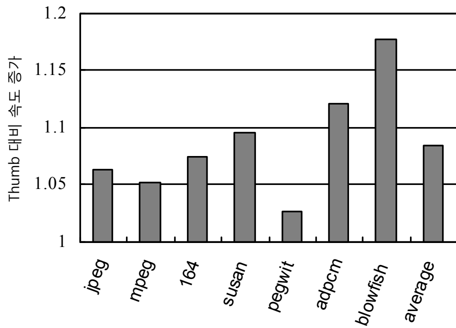


그림 6. Thumb 대비 AMEX의 속도 증가
Fig. 6. Speed-up of AMEX compared to Thumb

16비트 Thumb 모드를 지원하는 모든 프로세서는 32비트 ARM 모드도 함께 지원한다. 이러한 프로세서에서는 소스 파일별로 프로그래머가 성능과 코드 밀도의 상충관계를 고려하여 두 모드 중 하나를 선택한다. 따라서 제시된 방법은 Thumb 모드의 사용 빈도에 낮을수록 성능 개선 정도가 저하된다. 하지만 Thumb 모드는 ARM 모드와 비교하여 코드 크기가 30% 정도 저감한다. 또, 이를 통해 메모리 접근 횟수를 감소시키기에 소비전력도 줄어드는 장점이 있어서[24] 메모리 크기가 제한적이거나 저전력 소모가 중요한 모바일이나 임베디드 시스템에 널리 사용된다. 예를 들어 닌텐도의 게임보이(Gameboy)는 전적으로 Thumb 모드만을 사용하며 애플의 XCode도 디폴트로 Thumb 모드로 컴파일된다.

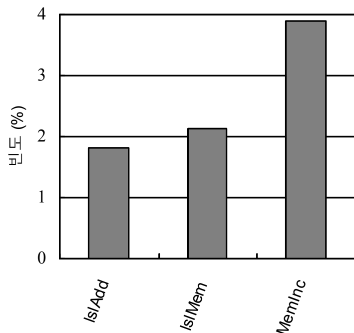


그림 7. 도입된 주소 방식의 등장 빈도
Fig. 7. The frequency of the introduced addressing modes

그림 7은 AMEX에 도입된 새로운 주소 지정 방식의 사용 빈도를 보여준다. lslAdd는 두 번째 레지스터 피연산자가 1비트나 2비트 왼쪽으로 시프트된 ADD 명령어를 표시한다고 하자. lslMem은 2비트 왼쪽으로 시프트된 레지스터 오프셋 주소 지정 방식의 LDR과 STR 명령어, MemInc는 LDRB, LDRSB, LDRH, LDRSH, LDR, STRB, STRH, STR 명령어에 도입된 즉시 사후 인덱스 주소 지정 방식이라고 하자. lslAdd, lslMem, MemInc은 평균적으로 각각 전체 실행 사이클의 1.8%, 2.1%, 3.9%를 차지한다.

그림 8은 16비트 Thumb 명령 집합 대비 AMEX의 압축 효율을 보여준다. 7개의 벤치마크 프로그램에 대해, 평균적으로 2.2% 코드 압축 효율이 향상된다. 이는 AMEX가 Thumb 코드에서 지원되지 않는 주소 지정 방식을 지원하기에 주소의 크기를 조정하거나 갱신할 때 Thumb에서 요구되던 LSL과 ADD 명령어들을 제거하기 때문이다.

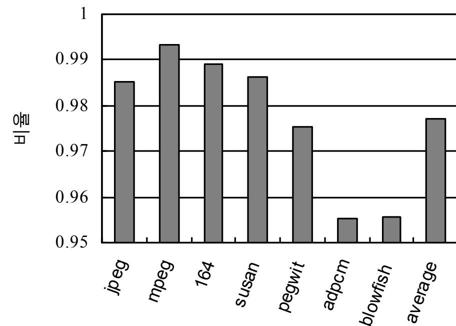


그림 8. Thumb 대비 AMEX의 압축 효율
Fig. 8. The compression efficiency of AMEX compared to Thumb

V. 결 론

본 연구에서는 주소 지정 방식이 성능에 미치는 영향에 주목하여 AMEX (Addressing Mode EXtension)라 불리는 16비트 Thumb 명령어 집합 구조를 개선하는 새로운 명령어 집합 구조를 제시한다. 전통적인 16비트 Thumb 구조와 비교하여 제시된 기법은 평균 8.5%의 성능을 향상시킨다. 제시된 방식은 16비트 Thumb 명령어 집합을 포함하는 전통적인 ARM7TDMI, ARM926EJ-S, ARM1136J-S 등의 ARM 프로세서부터 최신의 Thumb-2 명령어 집합 구조의 Cortex-M3 등의 다양한 프로세서들에 적용 가능하다.

참고문헌

[1] S. Segars, K. Clarke, and L. Goudge, "Embedded control problems, Thumb, and the ARM7TDMI," IEEE Micro, Vol. 15, No. 5, pp. 22-30, Oct. 1995.

[2] K. Kissell, "MIPS16: High-Density MIPS for the Embedded Market", Technical report, Silicon Graphics MIPS Group, 1997.

[3] Advanced RISC Machines Ltd., "ARM Annual Report & Accounts 2011," Advanced RISC Machines Ltd., 2011.

[4] R. Phelan, "Improving ARM Code Density and Performance," Technical report, Advanced RISC Machines Ltd.. June 2003.

[5] A. Krishnaswamy and R. Gupta, "Efficient Use of Invisible Registers in Thumb Code," In Proc. of the 38th IEEE/ACM International Symposium on Microarchitecture, pp. 30-42, Nov 2005.

[6] A. Krishnaswamy, and R. Gupta, "Dynamic coalescing for 16-bit instructions," ACM Transaction on Embedded Computing System, Vol. 4, No. 1, pp. 3-37, Feb. 2005.

[7] J. H. Lee, S. M. Moon, and H.K. Choi, "Comparison of Bank Change Mechanisms for Banked Reduced Encoding Architectures," In Proc. of the International Conference on Computational Science and Engineering, Vancouver, Vol. 2, pp. 334-341, Aug. 2009.

[8] Y. -J. Kwon, X. Ma, and H. J. Lee, "PARE: instruction set architecture for efficient code size reduction," IEE Electronics Letters, Vol. 35, No. 24, pp. 2098-2099, Nov. 1999.

[9] L. Dong, Z. Ji, G. Gui, and M. Hu, "Multithreading extension for Thumb ISA and decoder support," In Proc. of the 5th WSEAS Int. Conf. on Electronics, Hardware, Wireless and Optical Communications, pp. 78-81, 2006.

[10] X. H. Xu, S. R. Jones, and C. T. Clarke, "ARM/THUMB code compression for embedded systems," In Proc. of 15th Int. Conf. on Microelectronics, pp. 32-35, 2003.

[11] A.M. Fiskiran and R.B. Lee, "Performance Impact of Addressing Modes on Encryption Algorithms," In Proc. of the International Conference on Computer Design (ICCD 2001), pp. 542-545, Sep. 2001.

[12] B. Li, and R. Gupta, "Bit Section Instruction Set Extension of ARM for Embedded Applications," In Proc. of International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), pp. 69-78, Grenoble, France, 2002.

[13] H.-J. Cheng, Y.-S. Hwang, R.-G. Chang, and C.-W. Chen, "Trading Conditional Execution for More Registers on ARM Processors," In Proc. of the 8th IEEE/IFIP Int. Conf. on Embedded and Ubiquitous Computing (EUC), pp. 53-59, Dec. 2010.

[14] H.-H. Chiang, H.-J. Cheng, and Y.-S. Hwang, "Doubling the Number of Registers on ARM Processors," In Proc. of the 16th Workshop on Interaction between Compilers and Computer Architectures (INTERACT-16), Feb. 2012.

[15] S. Bartolini, I. Branovic, R. Giorgi, and E. Martinelli, "A Performance Evaluation of ARM ISA Extension for Elliptic Curve Cryptography over Binary Finite Fields," In Proc. of 16th Symposium on Computer Architecture and High Performance Computing, pp.238-245, 2004.

[16] Hedley Francis, "ARM DSP-Enhanced Extensions," ARM Ltd., 2001.

[17] ARM Ltd. "ARM Jazelle Technology", 2001.

[18] J. Rokov, and D. Ing, "ARM Architecture and Multimedia Applications," RIZ-Transmitters Co., 2010.

[19] ARM Ltd. "Introducing NEONTM Development Article," 2009.

[20] J. Goodacre, and A. N. Sloss, "Parallelism and the ARM instruction set architecture," Computer, Vol. 38, No. 7, pp. 42-50, 2005.

[21] S. -M. Kang, and J. -M. Kim, "Multimedia Extension Instructions and Optimal Many-core Processor Architecture Exploration for Portable Ultrasonic Image Processing," Journal of The Korea Society of Computer and Information, Vol. 17, No. 8, pp. 1-10, 2012.

[22] Y. -B. Jung, Y. -M. Kim, C. -H. Kim, and J. -M. Kim, "Performance Evaluation and Verification of MMX-type Instructions on an Embedded Parallel Processor," Journal of The Korea Society of Computer and Information, Vol. 16, No. 10, pp. 11-21, 2011.

[23] J. Lee, J. Kim, C. Jang, S. Kim, B. Egger, K. Kim, and S. Han, "FaCSim: A Fast and Cycle-Accurate Architecture Simulator for Embedded Systems," In Proc. of the Int. Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'08), Tucson, Arizona, USA, pp. 89-100, June 2007.

[24] A. Krishnaswamy, and R. Gupta, "Profile guided selection of arm and thumb instructions," In Proc. of the ACM SIGPLAN Joint Conf. on Languages Compilers and Tools for Embedded Systems & Software and Compilers for Embedded Systems, pp. 55-64, 2002.

저 자 소 개

김 대 환

1993 : 서울대학교 계산통계학과 이학사

1995 : 서울대학교 전산과학전공 이학석사

2010 : 서울대학교 전기컴퓨터공학부 공학박사

현 재 : 수원과학대학교 컴퓨터정보과 조교수

관심분야 : 컴파일러, 컴퓨터 구조, 임베디드 시스템, 멀티미디어, 모바일 컴퓨팅

Email : kimdh@ssc.ac.kr