Journal of The Korea Society of Computer and Information www.ksci.re.kr
Vol. 17, No. 10, October 2012 http://dx.doi.org/10.9708/jksci/2012.17.10.011

OpenRISC 프로세서를 위한 압축 명령어 집합 구조

김 대 환*

The Compressed Instruction Set Architecture for the OpenRISC Processor

Dae-Hwan Kim*

요 약

본 논문에서는 OpenRISC 프로세서의 코드 크기를 저감하는 새로운 압축 명령어 집합 구조를 제시한다. 새로운 명령어와 형식은 기존 명령어들의 사용 빈도와 용법에 대한 프로파일 정보에 의해 결정된다. 제시된 기법에서는 기존의 32비트 명령어들과 연속적인 명령어들을 각각 대체하는 새로운 16비트 명령어와 32비트 명령어를 도입한다. 제시된 명령어는 세 유형으로 분류할 수 있다. 첫 번째는 사용 빈도가 높은 기존의 덧셈, 로드, 저장, 분기 명령어 등의 32비트 명령어들을 대체하는 새로운 16비트 명령어들이다. 두 번째 유형은 사용 빈도가 높은 두 개의 연속적인 로드 명령어, 두 개의 연속적인 저장 명령어, 32비트 데이터 이동 명령어를 압축하는 새로운 32비트 명령어들이다. 마지막으로 함수 프롤로그와 에필로그 명령어들을 각각 하나로 압축하는 두 개의 새로운 32비트 명령어가 제시된다. 추가된 명령어들을 디코딩하기 위해서 OpenRISC 하드웨어 디코더 부분이 확장된다. OpenRISC 1200 프로세서에서 실험을 수행한 결과, 성능 저하 없이 30.4%의 코드 크기를 절감한다.

▸ Keywords : 명령어 집합 설계, 코드 크기, 임베디드 프로세서, OpenRISC, OR1200

Abstract

To achieve efficient code size reduction, this paper proposes a new compressed instruction set architecture for the OpenRISC architecture. The new instructions and their corresponding formats are designed by the profiling information of the existing instruction usage. New 16-bit instructions and 32-bit instructions are proposed to compressed the existing 32-bit instructions and instruction sequences, respectively. The proposed instructions can be classified into three types. The first is the new 16-bit instructions for the frequent normal 32-bit instructions such as add, load, store,

---

• 제1저자 : 김대환 • 교신저자 : 김대환
• 투 고 일 : 2012. 08. 10, 심사일 : 2012. 09. 03, 게재확정일 : 2012. 09. 14.
* 수원과학대학교 컴퓨터정보과 (Dept. of Computer Information, Suwon Science College)

branch, and jump instructions. The second type is the new 32-bit instructions for the consecutive two load instructions, two store instructions, and 32-bit data mov instructions. Finally, two new 32-bit instructions are proposed to compress function prolog and epilog code, respectively. OpenRISC hardware decoder is extended to support the new instructions. Experiments show that the efficiency of code size reduction improves by an average of 30.4% when compared to the OR1200 instruction set architecture without loss of execution performance.

▸ Keywords : Instruction set design, Code size, Embedded processor, OpenRISC, OR1200

I. 서 론

임베디드 시스템에서 프로그램 코드 크기는 중요한 고려 사항이다. 코드 크기를 감소하기 위한 컴퓨터 구조 중 일반적인 것은 '이중(dual) 명령어 집합(instruction set, IS)'을 지원하는 것으로 이 구조는 정규 명령어 집합(일반적으로 32비트)과 가장 많이 사용되는 명령어를 작은 비트수로 부호화한 '축소한 비트-너비 (reduced bit-width) 명령어' 집합(보통 16비트)을 지원한다. 가장 널리 알려진 이중 명령어 집합을 가지는 프로세서는 ARM 프로세서[1-2]로 32비트 IS과 Thumb으로 불리는 16비트 IS을 가진다. 다른 프로세서로는 MIPS/MIPS16[3], TinyRISC[4], STMicro의 ST100[5], ARC Tangent[6] 등이 있다.

이중 명령어 집합의 프로세서는 동적으로 축소 명령어 집합의 명령어를 그 명령어에 해당하는 정규 명령어로 확장한다. 이러한 압축 해제는 보통 디코더 단계에서 수행되고 실행 단계에서의 추가적인 하드웨어가 불필요하다. 이렇게 함으로써 압축 해제는 간단하고 사이클 손실 없이 수행된다.

보통 축소 명령어만으로 같은 태스크를 구현하기 위해서 정규 명령어 집합에 비해 성능 저하가 있다. 축소 명령어들은 비트 너비의 제한 때문에 여러 가지 제한이 가해져서 정규 명령어의 부분 집합만을 부호화하게 된다. 접근 가능한 레지스터의 개수, 피연산자의 개수, 명령어 내의 상수 범위, 분기 주소 값의 범위, 주소 지정 방식(addressing mode)의 수, 메모리 접근 오프셋(offset) 범위 등이 제한된다. 가령, Thumb의 대부분의 명령어는 16개의 범용 레지스터 중 8개만 접근 가능하다. ARM7TDMI 프로세서에서의 실험결과, 16비트 Thumb 구조는 32비트 ARM 구조와 비교하여 30% 정도 코드 크기 감소를 가져오지만 20~25% 정도 실행 속도가 저하된다[1-2].

이중 명령어 집합을 하나로 통합하는 노력이 기울여져 왔다. 2003년에는 Thumb-2 구조[7]가 도입되었는데 여기에서는 전통적인 16비트 Thumb 명령어와 추가적인 32비트 명령어를 하나의 명령어 집합으로 결합한다. Thumb-2 명령어 집합 구조는 32비트 ARM과 16비트 Thumb 사이의 최상의 절충안을 제시함으로써 ARM, Thumb 두 명령어 집합 구조를 개선하여 16비트 Thumb 코드 크기와 32비트 ARM 코드 성능을 제공한다.

코드 크기 감소와 실행 속도 향상의 두 가지 목적을 만족시키기 위해서는 이중 명령어 집합 혹은 Thumb-2와 같은 가변 길이 명령어 집합을 지원하는 것이 효과적이다. 본 논문에서는 코드 크기 감소를 위하여 프로세서의 명령어 집합을 확장하는 기법을 제시한다. 실행 코드의 빈도수와 용법을 분석하여 압축 명령어 집합 구조를 설계한다. 가장 널리 사용되는 공개 프로세서중 하나인 OpenRISC 1200 (OR1200)[8]에 구현한다. 실험 결과, 약 30%의 프로그램 코드 압축 효과를 얻는다.

본 논문의 구성은 다음과 같다. II장에서는 관련 연구를 기술하고, III장에서는 OpenRISC 명령어 확장 기법을 제시한다. VI장에서는 제시된 명령어 집합의 성능을 평가하며, 마지막으로 V장에서는 결론을 맺는다.

II. 관련 연구 및 배경

1. 관련 연구

OpenRISC 성능을 개선하기 위한 다양한 연구가 진행되어 왔다. [9-10]에서는 OpenRISC 코어의 성능 및 전력 소모 개선을 위하여 동적 분기 예측 구조, 4원 집합 연관 캐쉬 구조, 클록 게이팅 기법을 제시한다. [11]에서는 OpenRISC 구조에서 저전력 소모를 위해 RTL (Register Transfer Level) 코드에 회로의 스위칭(switching)이 최소화하게 자동으로 주석(annotation)을 삽입하는 기법을 제시한다. [12]에서는 OR1200 프로세서의 성능을 향상시키기 위해 멀티그레인(multigrain) 병렬 HPRC을 CPU내의 정수 실행

파이프라인에 내부적으로 추가하는 기법을 제시한다.

코드 크기 감소를 위한 OpenRISC 프로세서의 개선된 명령어 집합 구조는 일반에게 공개되지 않고 있으며 OpenRISC이외의 프로세서의 명령어 집합의 성능을 개선하기 위한 많은 연구들이 진행되어 오고 있다[13-16]. 대표적인 것이 가장 널리 사용되는 임베디드 프로세서 중의 하나인 ARM 프로세서의 명령어 집합 구조를 개선하는 기법들이다[13-14]. [13]은 미디어 처리 등 임베디드 시스템에서 자주 요구되는 워드보다 작은 단위에서 의 비트 선택 연산들을 지원함으로써 성능 향상을 도모한다. [14]에서는 명령어 집합 구조의 성능 향상에 중요한 요인 중의 하나인 가용 레지스터의 수를 증가시켜 메모리를 통한 변수 접근을 최소화한다. ARM 명령어 집합 구조에서 사용 가능한 가용 레지스터의 수를 증가시키기 위해 기존에 사용되던 조건 필드(conditional field) 비트들을 레지스터 필드에 추가로 할당한다. 이로 인해 ARM 명령어는 조건부 실행이 불가능해지지만 이용 가능한 레지스터의 수가 증가되어 전체적인 성능은 향상된다.

2. OpenRISC 구조

OpenRISC 1200 (OR1200)[8]은 32비트 OpenRISC 1000 구조[17]를 구현한 합성 가능한 프로세서로 가장 널리 사용되는 공개 소스(open-source) 프로세서 중의 하나며 OpenCores[18] 팀에서 개발되고 유지 보수된다. OR1200은 ARM10 프로세서 구조[2]와 호환될 수 있는 성능을 목표로 개발되었으며 32비트 ORBIS32 기본 명령어 집합(32-bit basic instruction set)을 구현한다. 그림 1는 OR1200 CPU의 구조를 보여준다. 32비트 크기의 32개의 범용 레지스터(General-Purpose Register, GPR)을 가지며 하버드(harvard) 구조, 단일 이슈(single-issue), 5단 파이프라인 구조로 대부분의 명령어를 단일 사이클에 실행할 수 있다. 추가적으로 DSP 응용 프로그램을 효율적으로 지원하기 위해 MAC (Multiply-accumulate) 장치를 포함한다.

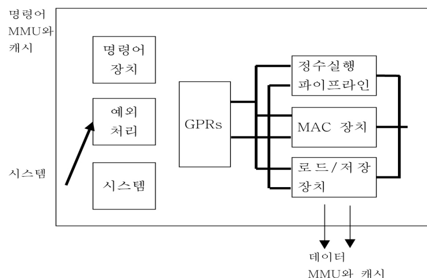

그림 1. OpenRISC 1200 CPU 구조
Fig. 1. OpenRISC 1200 CPU block diagram

1.1 범용 레지스터 용법

OpenRISC 1000 구조는 32개의 범용 레지스터를 지원한다. 각 레지스터의 용도는 표 1과 같이 정의된다. R0는 언제나 0으로 고정된다. R1은 스택 포인터로 현재 함수의 스택 주소 값을 가진다. R2는 프레임 포인터로 직전 스택 프레임의 주소 값을 지닌다. R3-R8은 함수 호출 매개 변수(parameter)로 사용된다. 6개를 넘어서는 매개 변수는 스택에 저장된다. R9는 링크 레지스터로 함수 호출 명령어의 위치를 지정하며 함수 종료 후 복귀 주소를 계산하는데 사용된다. R11은 함수의 복귀 값이다. R12는 64비트 복귀 값을 가지는 함수의 경우 상위 32비트를 지시하고 그 이외에는 임시 레지스터로 사용된다. R13, R15, …, R31은 임시 레지스터로 사용되며 R14, R16, …, R30은 피호출함수(callee)가 보존해야 한다.

표 1. OpenRISC 레지스터 용도
Table 1. OpenRISC register usage

레지스터	용도
R14, R16, R18, R20, R22, R24, R26, R28, R30	피호출함수 보존
R13, R15, R17, R19, R21, R23, R25, R27, R29, R31	임시레지스터
R12	임시레지스터/상위 복귀 값 (64비트 복귀시)
R11	복귀 값
R10	피호출함수 보존
R9	링크 레지스터
R3 - R8	매개변수0 ~ 5
R2	프레임 포인터
R1	스택 포인터
R0	0으로 고정

1.2 OpenRISC 1200 명령어 집합

OpenRISC 1200는 32비트 정수 명령어를 지원한다. 32비트 로드/저장 명령어, ALU 명령어, 흐름 명령어, 특수 명령어 등으로 구성된다. 표 2는 OpenRISC 1200의 명령어 집합을 분류한 것이다. ALU 명령어, 건너뛰기(jump)/분기(branch) 명령어, 로드/저장 명령어 등이 지원된다. 한 가지 주목할 점은 OpenRISC 1200에서의 조건 분기는 SR 레지스터(Supervision Register)의 1비트 플래그(flag) 비트에 의해 결정된다. 이 플래그는 SFcond 명령어에 의해 설정되는데 여기에서 cond는 EQ, NE, GT, LT, GE, LE의 조건을 의미한다. 한 예로 SFEQ 명령어는 두 피연산자의 값이 동일하면 플래그를 1로 하고 아닌 경우는 0으로 지정한다.

표 2. OpenRISC 1200의 주요 명령어 그룹
Table 2. OpenRISC 1200 major instruction group

명령어 그룹	의미
ALU	산술 논리 연산
J/JAL/JAR/JALR	건너뛰기/건너뛰고 링크 (Jump/jump and link)
BNF/BF	플래그 비지정/지정 분기 (Branch if no flag/Branch if flag)
LWZ/LHZ/LBZ, LWS/LHS/LBS	0 확장으로 워드/하프워드/바이트 로드, 부호 확장으로 워드/하프워드/바이트 로드
SW/SH/SB	워드/하프워드/바이트 저장
SFcond (cond: EQ, GE, GT, LE, LT, NE)	조건이 충족되면 플래그 비트 설정
MOVHI	즉시값을 레지스터 상위 16비트로 이동
NOP	무연산(no operation)

표 3은 ALU 명령어들을 보여준다. 산술 연산자로 ADD, SUB, MUL, 부호 없는 정수형의 곱하기인 MULU, 자리 올림 (carry) 덧셈인 ADDC가 지원된다. 논리 연산자로 AND, OR, XOR, 회전(rotate) 및 왼쪽/오른쪽 시프트 연산자로 ROR, SLL, SRA, SRL가 있다. 위의 연산자들은 모두 피연산자로 레지스터를 요구한다. 레지스터와 즉시(immediate)의 연산을 허용하는 명령어가 MULU를 제외한 모든 명령어에 존재한다. 가령 ADDI는 레지스터와 즉시를 더한 값을 목적지 레지스터에 저장한다.

표 3. ALU 명령어
Table 3. ALU instructions

명령어	의미
ADD/ADDC/SUB/ MUL/MULU	레지스터 산술 연산
ADDI/SUBI/MULI	즉시(Immediate) 산술 연산
AND/OR/XOR	논리 연산
ANDI/ORI/XORI	즉시 논리 연산
ROR/SLL/SRA/SRL	산술/논리 시프트/회전 명령
RORI/SLLI/SRAI/SRLI	즉시 시프트/회전 명령

III. 명령어 집합 설계

새로운 명령어 집합을 정의하기 위해 OR1200 GNU 컴파일러에 의해 생성된 이진 코드(binary code)에 대해 프로파일링(profiling)을 수행한다. 프로파일링에서는 빈번하게 발생하는 명령어와 연속적인 명령어들의 패턴을 분석한다. 프로파일링 결과를 지침으로 새로운 압축 명령어와 그 형식을 결정한다. 사용하는 벤치마크 프로그램은 대표적인 상용화 임베디드 프로그램 벤치마크인 MiBench [19], 대표적인 멀티미디어 벤치마크인 MediaBench [20], 대표적인 CPU 벤치마크인 SPEC2000 [21]에서 상호 보완적으로 선정한다. MiBench에서 이미지 코덱인 JPEG, 음성 코덱인 adpcm, 이미지 노이즈 필터 프로그램인 susan, 데이터 암호화 프로그램인 blowfish을 선정한다. MediaBench에서 동영상 압축 코덱인 MPEG과 데이터 암호화 및 인증 프로그램인 pegwit, SPEC200에서 데이터 압축 프로그램인 164(gzip)을 선정한다. JPEG과 adpcm은 MiBench와 MediaBench 모두에 포함된다.

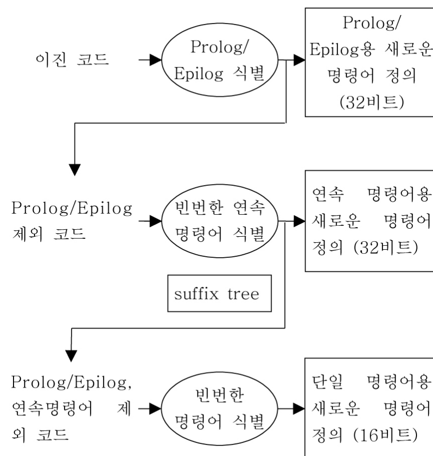

그림 2. 제시된 명령어 집합 확장 단계
Fig. 2. The steps of the proposed instruction set extension

제시된 기법은 세 단계로 구성된다. 그림 2는 제시된 기법을 보여준다. 먼저 입력 이진 코드에서 각 함수의 프롤로그(prolog)/에필로그(epilog) 코드를 찾는다. 이 코드의 발생빈도와 패턴을 분석하여 해당 명령어들을 대체하는 새로운 32비트 명령어를 도입한다. 다음 단계에서는 프롤로그/에필로그가 제외된 이진 코드를 입력으로 한다. 이는 새로운 명령어로 대체될 프롤로그/에필로그 코드의 명령어들이 지속적으로 프로파일링에 영향을 미치는 것을 방지하기 위해서다. 입력 코드에 대해 빈번하게 등장하는 연속 명령어들을 식별한다. 코드 식별을 위해서 각 명령어를 입력으로 한 접미사 트리(suffix tree)[22]를 구축한다. 생성된 접미사 트리로부터 연속적인 명령어 패턴의 명령어와 반복 횟수를 알 수 있다. 둘 이상의 연속 명령어들에 대해 출현 빈도수가 높은 것부터 새로운 명령어를 도입한다. 제시된 기법의 마지막 단계는 프롤로그/에필로그와 새로운 명령어로 대체될 연속 명령어들을

제외된 코드를 입력으로 한다. OR1200의 단일 명령어의 빈도수를 계산하고 빈도수가 높은 명령어들을 압축하는 새로운 16비트 명령어를 정의한다. 본 논문에서는 기존의 프롤로그/에필로그 명령어들을 각각 대체하는 새로운 32비트 명령어는 XPELOG로 표시한다. 연속된 코드 명령을 대체하는 새로운 32비트 명령어는 X32, 기존 OR1200 명령어를 대체하는 새로운 16비트 명령어는 X16으로 표시한다.

1. 프롤로그/에필로그 명령어 정의

그림 3은 프롤로그와 에필로그 코드의 등장 빈도수를 보여준다. 전체 명령어 중 프롤로그에 해당하는 명령어가 8.4%이며 에필로그 코드의 등장 빈도도 이와 동일하다. 프롤로그와 에필로그 코드는 함수의 진입 전과 후의 문맥을 저장 및 복원하기 위하여 생성된다.

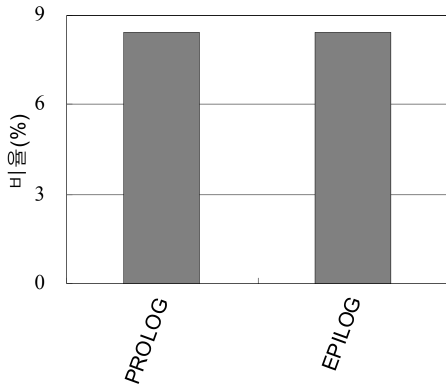

그림 3. 프롤로그/에필로그 명령어 등장 빈도
Fig. 3. The frequency of the prolog/epilog instructions

OR1200의 프롤로그 코드는 표 4의 두 가지 유형 중 하나에 속한다. 첫 번째는 저장할 레지스터의 수가 1개인 경우로 프레임 포인터 레지스터인 R2를 저장하게 된다. 표 4의 왼쪽 코드처럼 프레임 포인터를 저장하기 위해 스택 포인터를 4만큼 감소하고 프레임 포인터 R2를 스택에 저장한 후, 프레임 포인터를 변경 전 스택 포인터 값으로 지정한다. 프롤로그의 두 번째 유형은 여러 개의 레지스터를 저장하는 경우다. 각 레지스터의 크기가 4바이트이므로 스택 포인터 값을 4*저장할 레지스터의 개수만큼 감소한다. 그리고나서 'R2'를 저장한 후 변경 전 스택 포인터 값을 프레임 포인터로 지정한다. 그 후 각 레지스터를 스택에 차례로 저장한다. 이 때 저장되는 레지스터는 표 1의 레지스터 사용 규약에 따라 'R9', 'R10', 'R12', 'R14', 'R16', 'R18', 'R20', 'R22', 'R24', 'R26', 'R28', 'R30'의 순서이며 'R2'포함 최대 13개이다. 표 4의 오른쪽 코드는 저장할 레지스터가 13개인 경우를 보여준다. 저장할 레지스터의 수가 13개 보다 작은 경우는 그에 해당하는 크기만큼 스택 영역을 할당하고 해당 레지스터들을 저장한다.

보다 복잡한 프롤로그 코드 패턴은 레지스터를 저장할 스택 포인터의 값에 1바이트 크기의 오프셋이 더해지는 경우다. 이 값을 사전 오프셋(pre-offset)이라 하자. 레지스터들은 스택 포인터 값의 0번지부터 저장되는 것이 아니라 사전 오프셋만큼 더해진 위치부터 차례로 저장된다. 또 다른 예로는 스택 포인터에 또 다른 1바이트 크기의 값이 나중에 더해지는 경우다. 이 값을 사후 오프셋(post-offset)이라 하자. 이 경우 스택 포인터의 변동 후 값은 레지스터들을 저장한 후에 후기 오프셋이 더해진 값이다. 즉 지역 변수들을 저장하기 위한 영역을 확보하기 위해 스택 포인터의 값에 사전/사후 오프셋이 더해질 수 있다.

표 4. 함수 프롤로그 코드 패턴
Table 4. Function prolog code pattern

패턴 1 (저장 레지스터가 1개)	패턴 2 (저장 레지스터가 13개)

// 스택 포인터 4 감소
l.addi r1,r1,0xfffffffc
// R2 저장
l.sw 0x0(r1),r2
// 프레임 포인터 조정
l.addi r2,r1,0x4
	l.addi r1,r1,0xffffffcc
	l.sw 0x4(r1),r2
	l.addi r2,r1,0x34
	l.sw 0x0(r1),r9
	l.sw 0x8(r1),r10
	l.sw 0xc(r1),r12
	l.sw 0x10(r1),r14
	l.sw 0x14(r1),r16
	l.sw 0x18(r1),r18
	l.sw 0x1c(r1),r20
	l.sw 0x20(r1),r22
	l.sw 0x24(r1),r24
	l.sw 0x28(r1),r26
	l.sw 0x2c(r1),r28
	l.sw 0x30(r1),r30

일반적으로 사용자 코드에서는 프레임 포인터인 레지스터 R2를 저장/복원할 수 있지만 최적화된 라이브러리 코드에서는 저장/복원하지 않는다. 따라서 레지스터 저장/복원 리스트에 R2를 포함할 지, 아니면 제외할지를 추가적으로 지시해 줄 필요가 있다.

에필로그 코드 패턴은 프롤로그 코드 패턴의 반대이다. 저장된 각 레지스터의 값과 스택 포인터의 값을 함수 호출 전으로 복원하며 함수 호출 다음 명령어로 복귀한다.

프롤로그/에필로그 명령어 코드를 하나의 명령어로 대체할 새로운 명령어의 형식을 살펴보자. 새로운 명령어는 프롤로그/에필로그 코드를 대체하기 위하여 스택 크기, 사전/사후 오

프셋, R2 포함 여부를 명시해주어야 한다. 레지스터 저장에 필요한 스택 크기는 최대 52바이트이므로 6비트로 표현 가능하다. 프롤로그/에필로그를 기술하는 연산코드(opcode)는 기존 OR1200의 형식에 따라 연산 코드 6비트와 프롤로그/에필로그를 식별하는 1비트를 포함하여 7비트로 구현 가능하다. 프롤로그/에필로그를 나타내는 연산코드는 OR1200에서 사용되지 않는 연산코드 중 하나인 0x1C를 사용한다. 그림 4는 새롭게 도입되는 프롤로그/에필로그의 명령어 형식을 보여준다. 비트 25는 프롤로그 코드와 에필로그 코드를 식별한다. 이 값이 0이면 프롤로그 코드들, 1이면 에필로그 코드를 표현한다. 비트 17부터 24, 비트 9부터 16은 각각 사전 오프셋 및 사후 오프셋이고 비트 6는 레지스터 R2 저장 여부를 결정한다. 비트 0부터 5은 레지스터 저장에 필요한 바이트 단위의 스택 사이즈로 register list의 개수로 결정된다. 비트 7에서 8은 현재로는 사용되지 않고 0b00의 값으로 지정된다.

31 … 26	25	24 … 17	16 … 9	8 7	6	5 … 0
011100	T	pre-offset	post-offset	00	r2	reg.size

PROLOG (T=0) pre-offset, post-offset, reg_list
EPILOG (T=1) pre-offset, post-offset, reg_list

그림 4. 새로운 32비트 프롤로그/에필로그 명령어 형식
Fig. 4. New 32-bit prolog/epilog instruction format

2. 명령어 쌍을 대체하는 압축 명령어

그림 5는 프롤로그와 에필로그가 제외된 이진 코드에서 등장 확률이 높은 명령어 패턴을 보여준다. 'MOVHI+ORI'는 MOVHI (Mov Immediate High)와 ORI (Or with Immediate Half Word)의 명령어 쌍으로 전체 명령어의 10.4%를 차지하고, 다음으로는 2개의 연속적인 LWZ (Load Single Word and Extend with Zero) 명령어 쌍인 'LWZ+LWZ'와 두 개의 연속적인 SW (Store Single Word) 명령어 쌍인 'SW+SW'로 등장 빈도는 모두 6.2%이다. 프로파일링에 사용된 벤치마크 프로그램은 전역 변수와 구조체의 여러 필드를 종종 참조한다. OpenRISC에서는 전역 변수 주소 지정, 구조체와 배열 등의 집합체 데이터의 로드와 저장을 위해 각각 'MOVHI+ORI' 명령어 쌍, 두 개의 연속적인 LWZ와 SW 명령어 쌍이 생성된다. 세 명령어 이상이 연속하는 패턴은 등장 확률이 낮아서 고려 대상에서 제외된다.

'MOVHI rD, K' 명령어는 즉시값 K를 레지스터 rD의 상위 16비트로 이동한다. 'ORI rD, rA, K'는 원시 피연산자 레지스터 rA의 값과 즉시값 K를 OR 연산한 값을 목적지 레지스터 rD의 값으로 지정한다. 'LWZ rD, I(rA)'는 원시 레지스터 rA의 값에 변위(displacement) 값 I를 더한 값을 유효 주소로 하여 메모리에서 워드 데이터를 읽어서 레지스터 rD로 로드한다. 'SW I(rA), rB'는 레지스터 rB의 값을 rA+I 번지의 메모리에 저장한다. 'ADDI rD, rA, I'는 레지스터 rA의 값과 I를 더한 값을 rD에 지정한다.

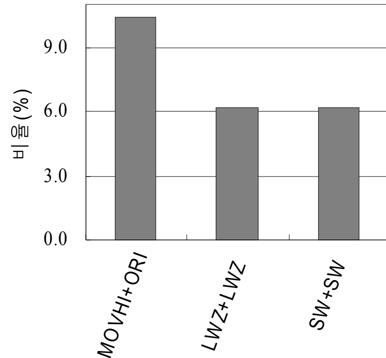

그림 5. 연속 명령어 등장 빈도
Fig. 5. The frequency of the instruction pair

OpenRISC 1200의 두 명령어를 압축하는 새로운 32비트 명령어를 도입하기 위해 연속 명령어의 피연산자를 분석한다. 두 연속 명령어 'MOVHI rD1, K1', 'ORI rD2, rA2, K2'에서 세 레지스터 rD1, rD2, rA2가 모두 동일한 경우가 전체의 91.6%이고 이 두 명령어는 레지스터 rD1의 값의 상위 16비트를 K1으로, 하위 16비트를 K2로 지정한다. 또 83.4%의 이 명령어 쌍에서는 K1값이 동일한 값인 전역 변수 메모리 영역 시작 주소로 지정된다. 이러한 관찰 결과를 반영하여 이 값을 저장하는 특수용 레지스터를 추가로 도입하고 본 논문에서는 SP_GLB_ADDR라 부른다. 그리고 새로운 명령어 X32.XMOV32를 정의하여 상위 16비트의 값을 SP_GLB_ADDR에서 가져오고 하위 16비트의 값은 즉시값으로 지정하게 하면 83.4%의 'MOVHI+ORI'의 연속 명령어는 하나의 새로운 X32.XMOV32 명령어로 표현 가능하게 된다. 해당 레지스터인 SP_GLB_ADDR는 시동 코드(start-up code)에서 전역 변수 영역 시작 주소 값으로 지정할 수 있다.

'LWZ+LWZ'은 연속적인 두 개의 로드 명령어 쌍이다. 이 두 명령어를 차례로 'LWZ rD1, I1(rA1)', 'LWZ rD2, I2(rA2)'라 하자. 그러면 한 명령어의 피연산자 지정에는 각 레지스터에 5비트, 각 변위에 16비트, 총 26비트가 요구되며, 두 명령어의 피연산자를 위해서 총 52비트가 요구된다. 이 둘을 대체하는 새로운 명령어를 X32.2LWZ라 하자. 기존 OR1200 명령어 형식에 따라 연산코드는 6비트를 할당한다. 그러면, 26비트 안에 모든 피연산자를 지시해야 한다. 피연산

자 분석 결과, 두 베이스 레지스터인 A1, A2가 동일한 경우가 71.8%이고 추가로 A1, A2 레지스터가 모두 15이하인 경우가 69.8%로 대부분을 차지한다. 이 경우 A1과 A2는 총 4비트로 표현 가능하다. 유사하게, 목적지 레지스터인 rD1, rD2에도 각각 4비트를 할당한다. 그러면, 이제 변위 I1과 I2에는 총 14비트를 할당할 수 있다. I1과 I2에 할당되는 비트 수를 변경하면서 최적의 조합을 결정한다. 실험결과, I1과 I2에 각각 7비트씩 동일하게 할당할 때 가장 좋은 결과인 전체의 57.7%의 명령어 쌍을 하나의 명령어로 통합할 수 있다. I1과 I2의 비트 할당 수에 따른 상세한 성능 평가는 IV장의 그림 10을 참고하기 바란다. 한 가지 주목할 점은 기존의 OR1200의 LWZ 명령어의 변위 값은 바이트 단위인데 비해 X32.2LWZ의 변위 값은 워드 단위로 표현된다. 이를 통해 접근 가능한 메모리 영역을 4배로 확장 가능하다. LWZ은 메모리 주소로부터 4바이트 값을 가져오는 명령어이기에 거의 대부분의 LWZ의 변위 값은 워드 정렬되어 있으리라 기대되며 실제로 프로파일링 결과, 이진 코드에 등장하는 모든 LWZ 명령어들의 변위 값은 워드 정렬되어 있다. 따라서 X32.2LWZ 명령어 양식에 등장하는 두 변위 값은 모두 워드 주소 값이며 실제 유효 주소 계산에서는 변위 값을 왼쪽으로 두 비트 시프트한 값을 사용한다.

SW+SW′ 명령어는 연속적인 두 저장 명령어이다. 두 개의 저장 명령어를 ′SW I1(rA1), rB1′, ′SW I2(rA2), rB2′라 하고 이에 해당하는 새로운 명령어를 X32.2SW라 하자. 두 개의 연속적인 저장 명령어를 X32.2SW로 변경 가능한 조건은 X32.2LWZ와 유사하다. 프로파일링 결과에 따라서 베이스 레지스터인 rA1, rA2는 동일하며 rA1, rA2, rD1, rD2는 각각 4비트, I1과 I2에 각각 워드 주소 7비트씩 할당한다. I1과 I2의 비트 할당 수에 따른 상세한 성능 평가는 IV장의 그림 10을 참고하기 바란다.

그림 6은 추가된 연속 명령어의 인코딩 형식을 보여준다. 두 개의 연속 로드 명령어에 해당되는 X32.2LWZ, 두 개의 연속 저장 명령어에 해당하는 X32.2SW, 레지스터 값에 32비트 값을 지정하는 X32.XMOV32의 형식이 기술된다. X32.2LWZ, X32.2SW, X32.XMOV32의 연산코드는 각각 OR1200에서 사용하지 않는 연산 코드인 0x1D, 0x1E, 0x1F를 사용한다. X32.2LW는 두 개의 연속적인 LWZ에 해당한다. X32.LWZ rD1, rD2, I1(rA), I2(rA)의 구문으로 표시되며 rA+4*I1 번지에서 rD1으로, rA+4*I2 번지의 메모리에서 rD2으로 각각 데이터를 로드한다. X32.SW 두 개의 연속적인 SW를 의미한다. X32.2SW I1(rA), 2(rA), rD1, rD2의 구문으로 표시되며 rD1과 rD2를 각각 rA+4*I1 번지, rA+4*I2 번지의 메모리로 저장한다. X32.XMOV32는 기존 MOV 명령어를 대체하는 MFSPR (Move From Special-Purpose Register) 명령어와 ORI 명령어의 조합으로 구성된다.

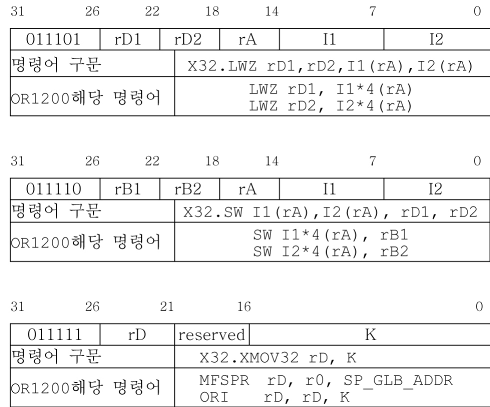

그림 6. X32 명령어 형식
Fig. 6. X32 instruction format

3. OR1200의 단일 명령어 압축

새로운 명령어 집합 설계의 마지막 단계는 등장 빈도수가 높은 OR1200의 32비트 명령어를 대체하는 새로운 16비트 명령어를 도입하는 것이다. 이를 위해 프롤로그/에필로그 코드와 X32 명령어로 변환될 연속 명령어들을 제외한 이진 코드에서 각 명령어의 등장 빈도수를 측정한다. 그림 7은 상위 빈도수의 명령어들을 보여준다. ORI, ADDI, LWZ, SW, JAL, BF, NOP, BNF, J, SFNEI, SFEQI 순이며 등장 확률은 각각 13.7%, 13.6%, 10.2%, 8.4%, 8.0%, 6.4%, 5.0%, 4.0%, 3.3%, 3.0%, 2.3%이다. ADD, LWZ, SW, 분기 및 건너뛰기, 레지스터 간의 데이터 이동 명령어는 일반적으로 가장 널리 사용되는 명령어들이며[23], ORI 명령어는 OpenRISC에서는 레지스터간의 데이터 이동 용도로도 사용되어 등장 빈도가 높다.

그림 8는 제시된 16비트 압축 명령어의 형식을 보여준다. OR1200와 동일하게 새로운 명령어의 연산코드의 너비는 6비트다. 각 명령어의 연산 코드는 OR1200의 사용되지 않는 연산코드를 사용한다. ORI, ADDI, LWZ, SW, NOP, J, JAL, BF, BNF, SFEQI, SFNEI 명령어는 각각 0x3C, 0x3D, 0x3E, 0x3F, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1A, 0x1B를 사용한다. 각 피연산자 비트 할당에 대한 성능 평가는 IV장에서 상세히 기술된다. ORI 명령어는 각 레지

스터에 5비트를 할당한다. ADDI 명령어는 각 레지스터에 4비트를 할당하고 즉시에는 2비트를 할당하고 부호 있는 바이트 값으로 해석한다. LWZ, SW 명령어에서는 레지스터 필드에 총 8비트와 양의 워드 주소 2비트 즉시 값을 지원하여 0, 4, 8, 12 변위의 LWZ, SW를 16비트로 변환가능하게 한다. 분기 및 건너뛰기 명령어에서는 분기 오프셋의 비트 수를 10비트로 결정한다. 플래그 지정 명령어에서는 레지스터와 즉시에 각각 5비트를 할당한다.

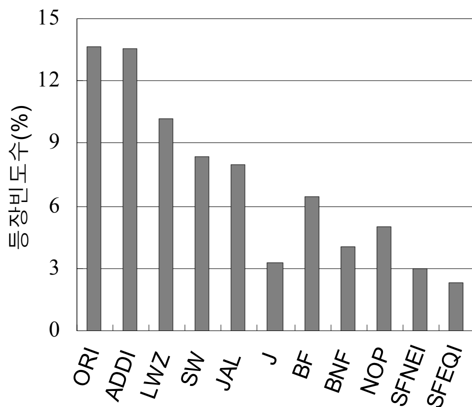

그림 7. OR1200 명령어 등장 횟수
Fig. 7. The frequency of OR1200 instruction

IV. 성능 평가

그림 9는 제시된 기법의 최종적인 성능 평가 결과를 보여준다. 프롤로그/에필로그의 해당 명령어들을 하나의 새로운 압축 명령어로 변경하면 전체 코드 크기가 14.9% 감소한다. 두 번째 단계인 연속된 두 명령어를 하나로 압축하는 X32 최적화를 적용하면 추가로 4.8%, 총 19.7%의 코드 크기가 감소된다. 마지막으로 X16 명령어를 도입하면 추가로 10.7%, 총 30.4%의 코드 크기가 감소한다.

제시된 기법에서는 추가적인 명령어를 처리하기 위해 OpenRISC 1200 하드웨어 디코더를 확장하는 것이 필요하다. 표 5는 명령어 추가에 따른 하드웨어 디코더의 크기 증가를 보여준다. Quartus II 7.2 버전을 통해 Cyclone 디바이스를 대상으로 합성한 결과 8428개의 로직 셀이 8833개로 4.8% 증가한다. 확장된 디코더는 전체 프로세서의 약 4.8% 정도로 절약되는 프로그램 코드 크기를 고려하면 경제적이다. 제시된 기법에서는 디코더를 확장하기 위해 디코딩 단계에서 높은 클록 단계와 낮은 클록 단계 중 사용되지 않는 낮은 단계의 클록을 사용함으로써 디코딩으로 인한 속도 저하가 없이 기존의 디코딩과 실행 단계가 유지된다.

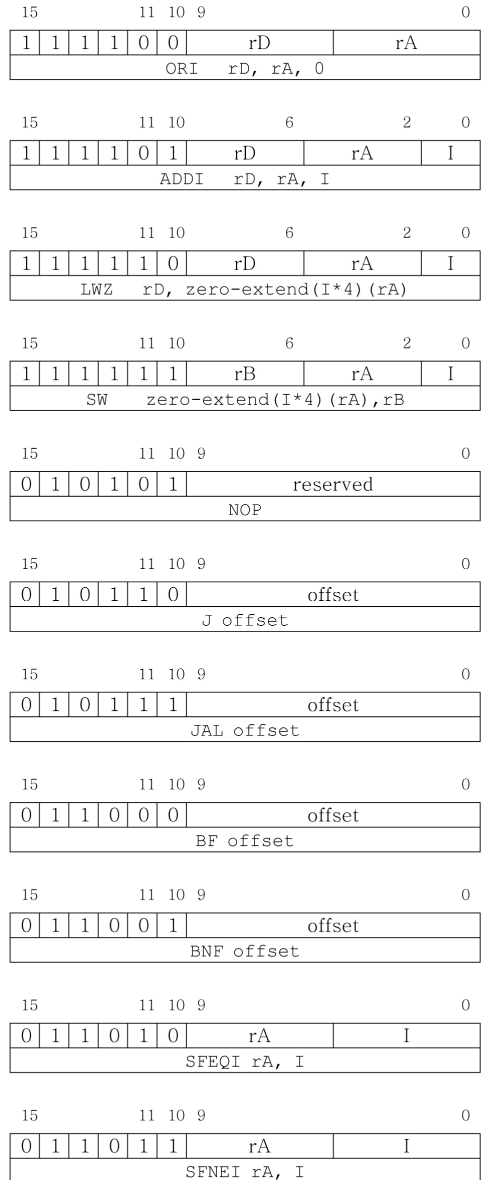

그림 8. X16 명령어 형식
Fig. 8. X16 instruction format

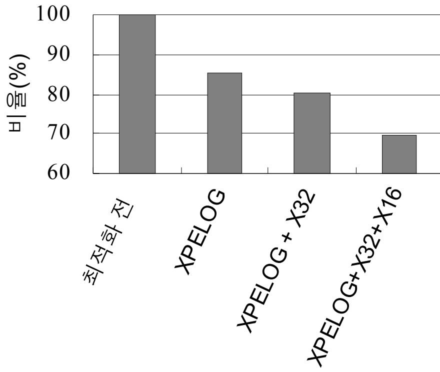

그림 9. 압축 결과
Fig. 9. Compression efficiency

표 5. 하드웨어 크기 증가
Table 5. Hardware size increase

	OR1200	확장된 OR1200
로직 셀	8428 (100%)	8833(104.8%)

1. 제시된 32비트 명령어 형식에 따른 성능 평가

명령어 쌍을 대체하는 새로운 32비트 명령어의 형식에 따른 성능을 평가한다. 먼저 두 개의 연속 로드 명령어의 즉시 필드 비트 할당 수에 따른 성능을 평가한다. 두 로드 명령어 중 첫 번째 명령어의 즉시 오프셋 필드를 I1, 두 번째 명령어의 즉시 필드를 I2라고 하자. 그러면, 그림 6에서와 같이 변위 I1과 I2에는 총 14비트를 할당할 수 있다. I1과 I2에 할당되는 비트 수를 변경하면서 최적의 조합을 결정한다. I1은 3비트부터 11비트까지 변경하며 이 때 I2는 11부터 3으로 변동된다. 그림 10은 각 비트 할당의 성능을 보여준다. X축의 두 숫자 중 앞의 숫자는 첫 번째 LWZ 명령어의 즉시 필드에 할당된 비트수를, 두 번째 숫자는 두 번째 명령어의 할당 비트수를 의미한다. 실험결과, I1과 I2에 각각 7비트씩 동일하게 할당할 때 가장 좋은 결과인 전체의 57.7%의 명령어 쌍을 하나의 새로운 명령어로 통합할 수 있다.

두 변위 값을 표현하는 대체 방안을 살펴보자. 연속적인 로드 명령어의 두 변위 값인 I1과 I2는 인접하리라 기대된다. 따라서 두 번째 로드 명령어의 변위 값 I2는 절대적인 값 대신에 직전 명령어의 변이와의 차이인 I2-I1으로 표현하는 경우를 살펴보자. 이 I2-I1값을 relI1(I2)라 하자. 이 경우 디코더 단계에서 두 번째 명령어의 실제 변위 값인 I2로 변경된다고 가정하자. 성능을 평가하기 위해 I1의 할당 비트 수와 I2-I1의 비트 수를 변경하면서 실험을 수행한다. 그림 11은 그 결과를 보여준다. I1에 9비트, relI1(I2)에 5비트를 할당한 경우가 가장 좋은 결과를 보여주며 58.4%의 명령어를 X32.2LWZ로 변경할 수 있다. 이는 I2의 값을 직접 기술해 준 그림 10의 결과 값인 57.7%보다 사소하게 좋다. 하지만 이 방식에서는 'X32.2LWZ'를 H/W 명령어 디코딩 과정에서 두 번째 LWZ 명령어의 변위 값을 relI1(I2)+I1으로 지정하기 위한 I1의 비트 너비에 해당하는 9비트 덧셈기가 추가적으로 필요하다. 이러한 추가적인 H/W 추가는 바람직하지 않고 직접 I1, I2를 기술하는 것과의 성능 차이가 미미하므로 X32.2LWX의 명령어 양식에서는 직접적으로 두 변위 값을 기술한다.

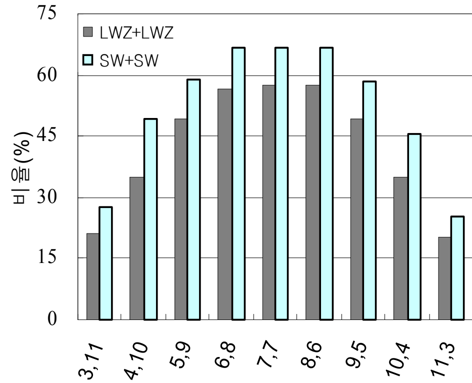

그림 10. 두 변위 할당 비트 수에 따른 연속 메모리 명령어의 압축 명령어 변환 비율
Fig. 10. The ratio of the conversion from sequential memory instructions to a new compressed instruction

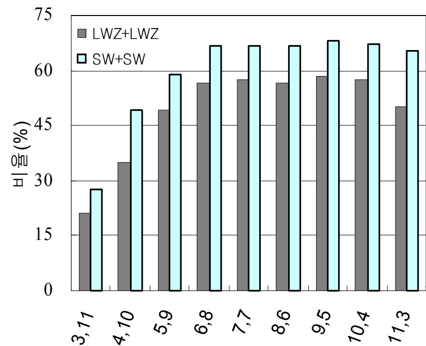

그림 11. 상대 변위 값으로 부호화하는 경우 연속 메모리 명령어의 압축 명령어 변환 비율
Fig. 11. The ratio of the conversion from sequential memory instructions to a new compressed instruction

2. 제시된 16비트 명령어 형식에 따른 성능 평가

OpenRISC의 32비트 단일 명령어를 16비트로 압축하는 제시된 명령어들의 형식에 따른 성능을 평가한다. 32비트 LWZ/SW 명령어에서 즉시 값은 부호화 값이기에 양의 값과 음의 값 모두를 지정할 수 있다. 그림 12는 두 명령어에서 즉시 값의 빈도를 보여준다. 즉시 값이 양수인 경우가 음수인 경우보다 훨씬 많이 등장한다. 따라서 LWZ, SW의 16비트 축약형 명령어에서는 즉시 값을 양으로 해석하면 부호 있는 값 표현보다 더 많은 32비트 LWZ, SW 명령어를 16비트로 변환할 수 있다. 즉시 피연산자를 워드 주소 2비트를 표현하면 0, 4, 8, 12 변위의 LWZ, SW를 변환할 수 있게 된다.

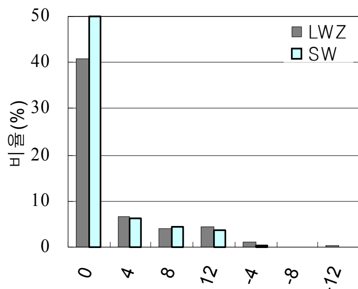

그림 12. LWZ, SW 명령어의 즉시 값 등장 빈도
Fig. 12. The frequency of LZW, SW immediate value

그림 13은 LWZ와 SW 명령어에서 레지스터 필드 할당 비트 수와 즉시 필드 할당 비트 수의 상충 관계를 보여준다. 그림에서 즉시는 워드 단위 값이다. 즉시 필드에 할당된 비트 수가 0인 경우는 즉시 값이 0임을 의미한다. LWZ와 SW 두 명령어에서는 두 개의 레지스터에 각각 5비트씩 총 10비트를 할당하고 즉시 변위 값은 0비트를 할당한 경우에 가장 좋은 결과를 얻는다. 그림 13은 이러한 관찰 결과를 반영한 결과를 보여준다. 즉시 필드 숫자 앞의 +부호는 양의 값을, +/-는 양과 음의 값을 의미한다. 예를 들어 +2는 2비트 부호 없는 값으로, +/-2는 부호 있는 2비트로 간주한다. 양의 2비트 즉시 값을 지원하고 레지스터 필드에 총 8비트를 할당하는 방식이 가장 좋은 결과를 보인다.

그림 14는 ORI 명령어에서 레지스터와 즉시의 비트 할당에 따른 16비트 축약 가능 명령어의 비율을 보여준다. 그림에서 X축의 두 숫자 중 앞의 숫자는 레지스터에 할당된 비트 수를, 두 번째 숫자는 즉시에 할당된 비트수를 의미한다. ORIU는 즉시 값이 부호 없는 정수인 ORI 명령어만 변환한 것을, ORIW는 즉시 값이 4의 배수인 명령어만을, ORIUW는 즉시 값이 부호 없는 4의 배수인 것만을 처리한 것을 나타낸다. 평가 결과, ORI 명령어는 레지스터에 더 많은 비트를 할당하는 것이 좋다.

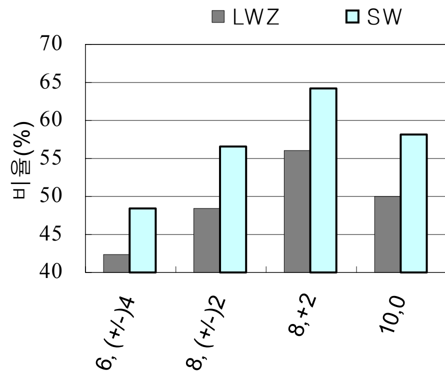

그림 13. 레지스터 필드, 즉시 필드 할당 비트 수에 따른 LWZ와 SW 명령어의 변환 비율
Fig. 13. The conversion ratio for the LZW/SW instruction by the number of bits for register and immediate fields.

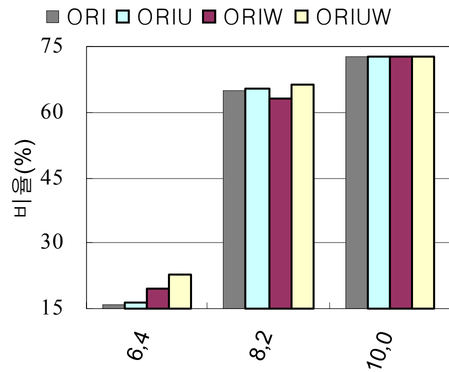

그림 14. ORI 명령어 변환 비율
Fig. 14. The conversion ratio for ORI instruction

그림 15는 ADDI 명령어에서 레지스터와 즉시 필드의 비트 할당에 따른 16비트 명령어로의 변환 가능 비율을 보여준다. 평가 결과, ADDI 명령어는 즉시에 2비트를 할당하여 부호 있는 바이트 단위로 처리하는 것이 가장 좋다. ADDI 명령어의 경우 다양한 범위의 즉시 값을 가지기에 상대적으로 적은 수의 명령어만을 16비트 명령어로 변환할 수 있다.

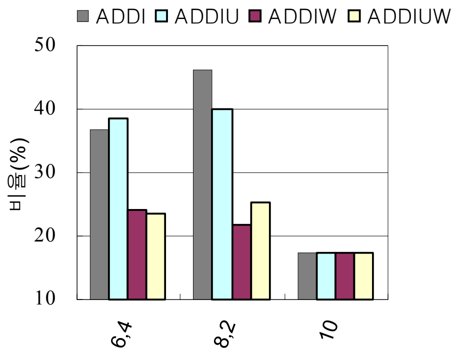

그림 15. ADDI 명령어 변환 비율
Fig. 15. The conversion ratio for ADDI instruction

그림 16은 레지스터 비트 수와 즉시 비트 수 변동에 따른 16비트 연산으로 변경 가능한 플래그 지정 명령어의 비율이다. 플래그 지정 명령어의 피연산자는 하나의 레지스터와 하나의 즉시로 구성된다. 연산 코드 6비트를 제외한 10비트 중 레지스터와 즉시에 동일하게 5비트씩 할당하였을 때에 가장 좋은 결과를 얻는다.

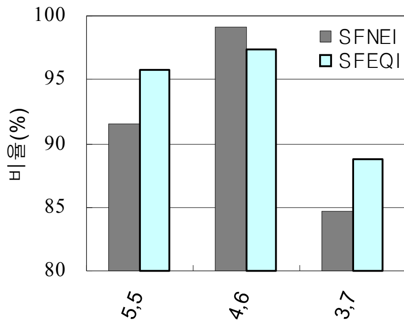

그림 16. SFNEI, SFEQI 명령어의 변환 비율
Fig. 16. The conversion ratio for SFNEI, SFEQI instructions

그림 17은 32비트 OR1200 명령어의 X16 명령어 변환 비율을 보여준다. OR1200의 건너뛰기/분기 유형 명령어의 71.1%가 X16의 10비트 오프셋 범위의 압축 명령어로 변환 가능하다. NOP 명령어는 피연산자를 요구하지 않으므로 기존의 모든 32비트 NOP 명령어를 16비트로 변경 가능하다. LZW, SW는 각각 56.0%, 64.3%가, ORI와 ADDI는 각각 72.6%, 46.1%, SFNEI와 SFEQI는 각각 91.6%와 95.8% 변경 가능하다.

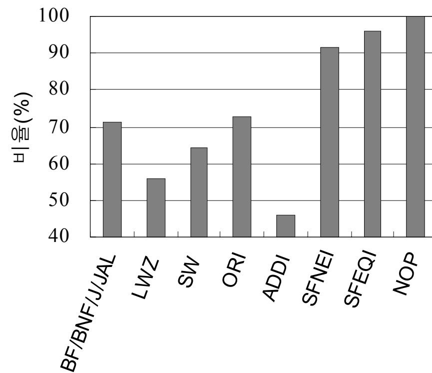

그림 17. OR1200 명령어의 X16 명령어 변환 비율
Fig. 17. The ratio of the conversion from the OR1200 instruction to the X16 instruction

V. 결 론

본 논문에서는 임베디드 시스템에서 중요한 고려 사항인 코드 크기를 감소하기 위해 프로세서의 명령어 집합을 확장하는 방법을 제안한다. 제시된 기법은 자주 등장하는 단일 명령어와 명령어 패턴들을 대체하는 새로운 16비트, 32비트 명령어들을 도입한다. 실험 결과, 제시된 명령어 집합 구조는 성능 저하 없이 30.4%의 코드 크기 효율을 향상시킨다. 새로운 명령어들을 디코딩하기 위해 하드웨어 디코더가 확장된다. 디코더 단에 추가된 H/W는 전체의 4.8% 정도 증가를 가져오지만 코드 압축으로 얻을 수 있는 효과에 비하면 경제적이기에 제시된 명령어 집합 확장 방법이 유용함을 보여준다. 제시된 방식은 기존 명령어들과 피연산자의 사용 빈도에 기반하여 새로운 명령어 집합과 형식을 결정하기에 OpenRISC 뿐 아니라 다양한 마이크로프로세서의 명령어 집합 설계에 효과적으로 적용되리라 기대된다.

향후에 코드 크기 절감 효율 향상을 위하여 명령어의 형식을 더욱 다양하게 검토하여 제시된 16비트, 32비트 형식뿐 아니라 8비트, 24비트, 48비트, 64비트 명령어들의 성능을 평가하고 도입하려고 한다. 또한, 실행 유닛의 작은 변경으로 압축 효율을 향상시킬 수 있는 새로운 명령어들의 가능성을 평가할 계획이다.

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] S. Furber, "ARM system-on-chip architecture," Addison-Wesley, 2000.

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

[4] LSI LOGIC, "TinyRISC LR4102 Microprocessor Technical Manual," LSI LOGIC, Milpitas, CA, 2000.

[5] ST Microelectronics, "ST100 Technical Manual," ST Microelectronics, Geneva, Switzerland, 2004.

[6] ARC Cores, "ARCtangent-A5 Microprocessor Technical Manual," Herts, England, 2005.

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

[8] D. Lampret, "OpenRISC 1200 IP core specification," 2001.

[9] H. Jung, and K. Ryoo, "Performance and Power Consumption Improvement of Embedded RISC Core," Journal of the Korean Institute Of Maritime information & Communication Science, Vol. 14, No. 2, pp. 453-461, 2010.

[10] H. Jung, X. Jin, and K. Ryoo, "Performance Improvement and Power Consumption Reduction of an Embedded RISC Core," Journal of information and communication convergence engineering, Vol. 10, No. 1, pp. 78-84, 2012.

[11] V. Viswanath, J. A. Abraham, and W. A. Hunt, "Automatic insertion of low power annotations in RTL for pipelined microprocessors," In Proceedings of the conference on Design, automation and test in Europe, pp. 496-501, 2006.

[12] R. Maheswari, and V. Pattabiraman, "A new technique of embedding multigrain parallel HPRC in OR1200 a soft-core processor," In Proceedings of SEPADS'12/EDUCATION'12, pp. 92-97, 2012.

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

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

[15] 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.

[16] 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.

[17] D. Lampret, C.-M. Chen, M.Mlinar, et al, "OpenRISC 1000 Architecture Manual," 2003.

[18] OpenCores, http://www.opencores.org

[19] C. Lee, M. Potkonjak and H. Mangione-Smith," MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," Micro-30, pp. 330-335, November 1997.

[20] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "Mibench: A free, commercially representative embedded benchmark suite," In Proceedings of the 4th IEEE International Workshop on the Workload Characterization, pp. 3-14, 2001.

[21] John L. Henning, "SPEC CPU 2000: Measuring CPU performance in the new millennium," IEEE

Computer, Vol. 33, No. 7, pp. 28-35, July 2000.
〔22〕 E.M. McCreight, "A Space-Economical Suffix Tree Construction Algorithm," Journal of the ACM, Vol. 23, No. 2, pp. 262-272, April 1976.
〔23〕 J. L. Hennessy, and D. A. Patterson, "Computer Architecture - A Quantitative Approach (5. ed.)," Morgan Kaufmann, pp. B1-B.47, 2012.

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, 모바일컴퓨팅, 멀티미디어
Email : kimdh@ssc.ac.kr