Instruction Set Reference Manual

for AE32000: An 32bit EISC microprocessor

Processor Team, R&D center,

ADChips Inc.

Revision : 1.7

October 20, 2009

Document revision histroy

```
$Log: ae32000-isa-rm_ko.tex,v $
Revision 1.7  2008/05/06 10:00:34  babyworm
fix type

Revision 1.6  2008/03/12 01:32:38  babyworm
modified hangul emph type

Revision 1.5  2008/03/12 01:28:47  pioneer
Correct typo

Revision 1.4  2007/10/12 02:49:09  babyworm
fix font

Revision 1.3  2007/10/12 02:32:43  babyworm
fix format

Revision 1.2  2007/09/18 07:38:49  babyworm
modify style

Revision 1.1  2007/09/18 05:45:41  babyworm
Initially Added
```


©*Advanced Digital Chips Inc.*

All right reserved.

No part of this document may be reproduced in any form without written permission from Advanced Digital Chips Inc.

Advanced Digital Chips Inc. reserves the right to change in its products or product specification to improve function or design at any time, without notice.

Office

14th Floor, Instopia Bldg., 467-23, Dogok-Dong,

Gangnam-Gu, Seoul, 135-270, Korea.

Tel : +82-2-2107-5800

Fax : +82-2-571-4890

URL : http://www.adc.co.kr

Contents

List of Tables

List of Figures

This page intentionally left blank

TRADEMARKS

EISC®는 Advacned Digital Chips Inc.의 등록상표입니다.
AE32000®은 Advacned Digital Chips Inc.의 등록상표입니다.
AMBA®, AHB®, AXI®, APB®는 ARM Inc.의 등록상표입니다.
GNU toolchain은 GPL/LGPL을 따릅니다.
기타 각사의 등록 상표는 각사의 소유입니다.

Chapter 1

Introduction

본 매뉴얼은 32비트 EISC프로세서인 AE32000 명령어 셋 아키텍쳐에 대한 전반적인 내용을 다루도록 한다. AE32000 명령어 셋 아키텍쳐(ISA)는 EISC(Extendable Instruction Set Architecture)를 기반으로 하며, 16비트의 고정 길이 명령어 형식을 지니고 있다.
본 장에서는 다음과 같은 사항을 다루도록 한다.

- About this Document
- About EISC Architecture
- About AE32000

1.1 About this Document

이 문서는 EISC 아키텍처 기반의 32비트 마이크로 프로세서 명령어 셋 아키텍처인 AE32000에 대하여 설명하려고 한다. 또한, AE32000 명령어 셋에서 정의 된 명령어들에 대하여 설명하고 사용법 및 주의 사항을 제시하기 위하여 작성되었다. 이 문서는 AE32000 프로세서를 이용하는데 알아야 하는 명령어에 대한 프로세서의 동작을 기술하는데 집중하도록 한다.

1.1.1 Cautions

AE32000의 명령어 셋은 프로세서의 구현에 따라 확장/축소 될 수 있으므로, 자신의 프로세서 참조 매뉴얼을 통하여 지원되는 명령어 및 특성을 알고 있어야 한다. 이 매뉴얼에서는 해당 명령어가 공통적으로 적용되지 않는 경우에는 해당 명령이 어느 프로세서(혹은 variant)에서 사용 혹은 사용 불가능한지를 명시적으로 나타내도록 한다.

1.1.2 Feedback

문서에 오류나 정정 사항이 있다면, 홈페이지 www.adc.co.kr에 접속하셔서 알려주시거나, babyworm@adc.co.kr로 해주시면 감사드리겠습니다.

1.2 About EISC Architecture

내장형 프로세서는 일반적인 데이터 처리용 프로세서와 비교하여 보았을때 저가, 저전력이 중요한 부분을 차지한다. 수행 프로그램이 ROM의 형태로 저장되는 내장형 프로세서 분야에서 수행 프로그램의 크기를 효과적으로 줄임으로 다이(die)에 있어서 많은 크기를 차지하는 ROM의 크기를 줄일 수 있으므로 조금 더 저가의 내장형 시스템을 공급할 수 있다.

EISC(Extendable Instruction Set Computer)는 수행 프로그램의 크기 및 메모리 접근의 회수를 매우 효과적으로 줄일 수 있도록 설계된 명령어 셋으로서, 확장 명령어를 이용하여 명령어의 즉치 값 및 변위 등의 값을 자 유롭게 확장할 수 있는 형태를 지닌 아키텍쳐로서 기존의 CISC와 RISC의 장점을 합친 형태를 취하고 있다. EISC는 기본적으로 RISC가 갖고 있는 간단한 구조의 하드웨어를 취하면서도 CISC의 장점을 추가하여 높은 성능을 갖게 하였고, 코드 밀도가 높아 기존의 RISC 프로세서와 비교하여 약 60%, CISC 프로세서와 비교하여 약 80% 정도로 프로그램 크기가 작다는 장점을 지니고 있다. 따라서, 코드 밀도가 중시되는 내장형 응용 분야에 있어서 강점을 지니고 있다. 기존의 CISC (Complex Instruction Set Computer) 기반의 마이크로 프로세서는 코드 밀도에 있어서 장점이 있으나 고속화에 문제점을 지니고 있으며, RISC (Reduce Instruction Set Computer) 기반의 마이크로 프로세서는 대부분 32-bit 크기의 명령어를 지님으로 프로그램 코드의 크기가 지나치게 커진다는 단점을 지니고 있다.

EISC기반의 32-bit 마이크로 프로세서인 AE32000은 16-bit 크기의 명령어를 이용하여 32-bit 데이터를 효과적으로 처리할 수 있도록 설계되었다. 이는 ARM사의 Thumb 계열의 프로세서나 MIPS16에서도 채택하고 있는 방법이나, EISC 아키텍쳐의 경우 16-bit 크기의 명령어 사용으로 인한 즉치 값(immediate value) 공간의부족을 이전의 접근 방식과는 다르게 독립적으로 사용 가능한 LERI(Load Extension Register and set E) 명령을 이용하는 것으로 해결하였다. LERI명령은 2비트 Op-Code와 14비트 즉치 값을 가지는 명령으로서 ER(Extension Register)에 즉치 값을 저장하고, 이후에 즉치 값의 확장이 필요한 명령에서 ER의 값을 인출하여 해당 명령어의 즉치 값과 덧붙이는 구조를 지니고 있다. 이러한 구조의 장점은 앞에서 기술한 바와 같이 짧은 즉치 값의 크기로 인하여 발생 가능한 문제를 효과적으로 해결 할 수 있다는 것이나, LERI의 부가로 인하여 코드의 길이가 길어질 수 있다는 문제와 성능의 저하를 가져 올 수 있다는 문제를 지니고 있다. 따라서, EISC 기반의 프로세서는 이 LERI를 효과적으로 처리하는 것이 중요하다.

AE32000은 EISC기반의 32-bit 마이크로 프로세서로서 Low-End용 32-bit 마이크로 프로세서인 SE3208의 특징에 부가적으로 더 깊은 파이프라인과 OSI(On-Silicon In Circuit Emulator) 지원 기능, 캐쉬 및 가상 메모리의 지원기능, LERI 명령어 폴딩 기능등을 추가하여 성능을 향상시킨 프로세서이다.

본 사용자 매뉴얼은 고성능 EISC(Extendable Instruction Set Computer) Family중 32bit버전인 AE32000에 대하여 다루도록 한다.

1.3 About AE32000

AE32000은 EISC 아키텍쳐를 기반으로 하고 있는 32-bit 프로세서를 위한 명령어 셋 아키텍처이다. 이 명령어 셋은 16비트 명령어를 이용하여 32비트 데이터 패스를 제어하는 축약 코드 RISC(Compressed Code RISC)의 형태적 특징을 지니고 있음으로써, 코드 밀도가 상대적으로 훌륭하며, EISC 아키텍처 특유의 즉치 확장 기능을 통하여 쉽게 32비트 즉치값의 생성이 가능함으로써, 즉치 연산이 용이하고, 32비트 메모리 영역에 대한 주소 생성이 간단하다는 장점을 지니고 있다.

AE32000 아키텍처는 그 구현에 있어서 프로세서의 처리 효율을 향상시키기 위하여 분리된 명령어/데이터 버스를 지니는 Separated-Bus 아키텍처[1]를 채용하며, 즉치 확장을 위한 `leri`[2] 명령어를 좀더 효과적으로 처리하기 위한 LERI instruction folding 기법이 적용된다.

AE32000 명령어 셋 아키텍처는 보조 프로세서를 통한 기능 확장성을 지니고 있으며 4개의 보조 프로세서와 연결 가능한데, 보조 프로세서 0번은 시스템 보조 프로세서로 1번은 부동 소수점 연산 보조 프로세서로 예약 되어 있다.

AE32000 명령어 셋 아키텍처는 다음과 같이 몇 가지 Revision을 지니고 있다. 해당 명령어 셋은 binary 수준에서 하위 호환성을 지니고 있으나, AE32000A와 AE32000B간에는 곱셈의 결과 저장 방식이 다르므로, 호환성이 떨어진다.

1. AE32000A: 초기 AE32000 아키텍처. 현재 사용하지 않음
2. AE32000B: MH/ML 및 기본적인 DSP가 추가된 ISA.
3. AE32000C: 사용되지 않는 `PREFD`, `PREFI` 명령등이 제외 혹은 시스템 보조 프로세서의 기능으로 변경되었으며, DSP 기능 확장, JAVA 기능 확장 및 명령어 추가.
   (a) AE32000C-DSP: SIMD DSP 확장 명령어가 추가 됨
   (b) AE32000C-Tiny: 저전력 프로세서를 위하여 DSP 명령어 중 일부 제한
   (c) AE32000C-JAVA: JAVA 하드웨어 가속기를 위하여 확장 명령어가 추가 됨

[1] Harvard Architecture라고도 불린다
[2] 즉치값 확장을 위하여 사용되는 명령어. 159페이지의 4.53절을 참조하라

Chapter 2

Programmer's Model

본 장에서는 프로그래머 관점에서의 AE32000 프로세서 기반의 SoC에서 프로그램을 작성하는데 있어서 필요한 사항을 기술하도록 한다.

2.1 Data Types

AE32000은 32비트 데이터 처리를 기반으로 하고 있으므로 word의 크기는 32비트가 된다. AE32000은 다음과 같은 형식의 데이터 처리를 지원한다. 또한, 각 데이터는 signed와 unsigned number가 지원된다.

- `byte  :  8 bits, char`
- `short  :  16 bits, short int`
- `word  :  32 bits, int, long int`

32비트간의 곱셈을 수행하는 경우 그 결과는 64비트로 표현 될 수 있다. 64비트 값의 길이는 컴파일러에 따라 그 정의가 다르다.

- `MSC_VER :  signed 64bit int, signed __int64`
- `GNUC  :  signed long long`

2.2 Processor Modes

많은 프로세서들은 동작 모드를 구분함으로서 사용자 프로그램의 잘못된 프로그램으로 인하여 프로세서가 오 동작하는 것을 방지하고 있다. AE32000은 기존의 프로세서들이 가지고 있는 관리자 모드 및 사용자 모드 이외에 디버깅시에 사용할 수 있는 OSI 모드를 지니고 있다. 다음은 AE32000이 지원하는 프로세서 모드들이다.

- 관리자 모드 (`Supervisor mode`)
- 사용자 모드 (`User mode`)
- OSI 모드 (`OSI mode`)

관리자 모드 : 관리자 모드는 모든 자원에 대한 접근이 가능한 모드로서, OS와 같이 자원 관리를 담당하는 프로그램들이 사용하는 모드이다. 프로세서의 모드를 구분함으로서 사용 자원 관리에 있어서 안정성을 높이기 위하여 사용된다. 사용자 모드에서 SWI를 통하여 진입 가능하며, POP SR을 통하여 이전 프로세서 모드로 복귀 가능하다. 관리자 모드에서 사용자 모드, 혹은 OSI모드로 전환할 때 SET, CLR을 통하여 processor mode를 직접 바꿈으로서 모드를 바꾸는 것은 시스템 초기 이외에는 권장되지 않으며, 반드시 사용이 필요한 경우 주의하여야 한다.

사용자 모드 : 사용자 모드는 사용자 프로그램이 수행되는 모드로서, 일반 용도의 프로그램이 구동되는 부분이다. 사용자에서는 사용자에게 할당된 자원(할당된 메모리 영역 및 보조 프로세서)에 대한 접근만이 가능하며, 만일 사용자에게 할당되지 않은 자원에 접근하는 경우 violation이 발생한다.

OSI 모드 : 프로세서의 동작을 디버깅하기 위한 모드로서, 관리자 모드의 권한과 동일한 자원 접근 권한을 지닌다. 단, 관리자 모드와 구분되는 스택 영역을 지니고 있으므로, 관리자 모드에서의 각 레지스터 상태를 보다 정확하고 쉽게 찾아낼 수 있다는 장점이 있다. (물론, 이러한 동작은 관리자 모드에서 세심한 프로그램을 통하여 찾아낼 수 있으나, OSI모드를 사용하는 경우 모니터 프로그램 및 관리가 간단하다는 장점을 지닌다.)

2.3 General Purpose Registers

AE32000은 16개의 범용 레지스터를 지니고 있으므로, 데이터를 위한 메모리의 잦은 접근을 줄일 수 있다는 장점을 지닌다. ae32000-elf-gcc를 이용하여 컴파일 하는 경우 5번과 7번 레지스터는 frame pointer과 index register로서 사용되도록 지정되어 있으므로, 인라인 어셈블러에서 이 레지스터를 사용할 때는 주의를 기울여야 한다.

일반적으로 8번과 9번 레지스터는 argument passing을 위하여 사용되므로, 만일 인자에 대한 처리를 인라인 어셈블러를 통하여 하고자 할때 이 레지스터들의 사용에 주의하여야 한다. 2개 이상의 인자가 사용될 경우에는 부족한 인수들을 stack을 이용하여 passing 한다. 컴파일러에서 레지스터를 할당할 때는 0, 1, 2, 3, 4, 6, 10, 11, 12, 13, 14, 15의 순서로 할당을 선호하게 된다. 자세한 사항은 ae32000-elf-gcc혹은 소프트웨어 레퍼런스 매뉴얼을 참조.

번 호	목 적
5	Frame Pointer
7	Index Register
8	Argument 0
9	Argument 1

Figure 2.1: 사용에 주의하여야 할 범용 레지스터

2.4 Special Purpose Registers

AE32000에는 9개의 특수 목적 레지스터와 3개의 스택 포인터 레지스터들이 존재하며, 이중 각 동작 모드에서 사용되는 스택 포인터(SSP, ISP, USP)를 제외한 모든 특수 목적 레지스터들은 PUSH/POP되는 것이 가능하다.

2.4.1 Status Register

AE32000의 동작 상태 혹은 설정을 나타내는 레지스터로서 SET/CLR 명령에 의하여 15~0번 비트까지 조절 할 수 있다. 단, 15~8번 비트의 경우 관리자 모드에서만 변경 가능하며, 사용자 모드에서 15~8번 비트를 변경하고자 하는 경우 아무런 영향을 미치지 않는다.

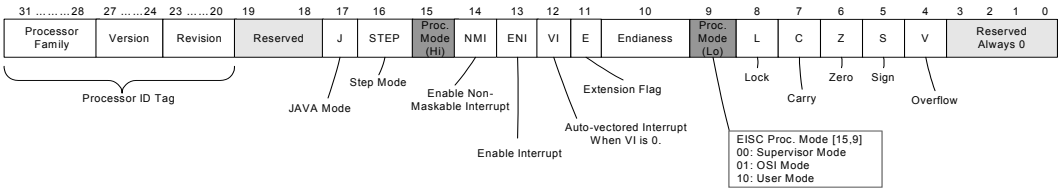


Figure 2.2: Status Register

Table 2.1: 상태 레지스터의 각 비트 정의 및 설정 방법

Tag	Width	Description
Processor Family	4bit	프로세서의 종류를 나타낸다. AE32000C-Lucida 프로세서의 경우 AE32000 Family로서 '4'번을 지닌다.
Version	4bit	프로세서의 버전을 나타낸다. AE32000C-Lucida 프로세서의 경우 AE32000의 세번째 버전인 'C'버전에 해당하므로, '3' 값을 지닌다.
Revision	4bit	프로세서의 **revision**을 나타낸다. Revision은 동일한 명령어 셋 아키텍쳐 및 구성에서 마이크로아키텍쳐적으로 작은 수정이 가해진 경우이다. AE32000C-Lucifer : '0' AE32000C-Lucida : '1'

Table 2.1: 상태 레지스터의 각 비트 정의 및 설정 방법(계속)

Tag	Width	Description
J	1bit	**JAVA Flag**를 의미한다. JAVA 응용 프로그램을 JAVA 하드웨어 가속기를 사용하여 실행 시키는 경우에 사용된다. SET/CLR 명령을 이용하여 프로세서의 모드를 변경시킬 수 없다. Bit Setting 0 : EISC Mode 1 : JAVA Mode
STEP	1bit	**Single Step Mode**를 의미한다. 디버깅 기능 중에 Single Step 기능을 활성화 시킨 경우를 의미한다. Single Step Mode가 적용되는 경우 하나의 명령어(LERI 제외)가 실행된 후 자동적으로 OSI exception이 발생하게 된다. Single Step Mode는 관리자 모드 및 사용자 모드에서만 적용되며, OSI 모드에서는 Single Step Mode로 지정되어 있더라도 Single Step 기능을 수행하지 않는다. Bit Setting 0 : Disable 1 : Enable
Proc Mode	2bit	**Processor Operation Mode**를 의미한다. SR의 15번째 bit와 9번째 bit의 조합으로서 구성되며, 프로세서의 현재 동작을 나타낸다. SET/CLR 명령을 이용하여 프로세서의 상태를 변경시킬 수 있으나, 이는 권장되지 않는다. Bit Setting 00 : Supervisor Mode 01 : OSI Mode 10 : User Mode 11 : Undefined Mode

Table 2.1: 상태 레지스터의 각 비트 정의 및 설정 방법(계속)

Tag	Width	Description
NMI	1bit	**Non Maskable Interrupt Enable**을 의미한다. Non Maskable Interrupt(NMI)를 받아들일 것인지 결정한다. Bit Setting 0 : Disalbe 1 : Enable
INT	1bit	**Interrupt Enable**을 의미한다. 외부 인터럽트를 받아들일 것인지 결정한다. Bit Setting 0 : Disable 1 : Enable
VI	1bit	**Vectored Interrupt Mode**를 의미한다. 외부 인터럽트의 처리 방식을 vectored 형식으로 할 것인지 auto vectored 형식으로 할 것인지 결정한다. Bit Setting 0 : Autovectored Interrupt 1 : Vectored Interrupt
E	1bit	**Extension Flag**를 의미한다. Bit Setting 0 : Normal Instruction 1 : Extenstion Instruction

Table 2.1: 상태 레지스터의 각 비트 정의 및 설정 방법(계속)

Tag	Width	Description
Endian	1bit	**Endianess**를 의미한다. 프로세서의 Endian 형식을 결정한다. 이 비트는 Power-On Configuration으로서 동작 중간에 변경할 수 없다. Bit Setting 0 : Little Endian Mode 1 : Big Endian Mode
L	1bit	**Lock Flag**를 의미한다. Lock은 프로세서의 Automic processing 동작에 설정된다. 이 bit이 설정되어 있는 경우 프로세서는 NMI와 INT가 disable된 것 처럼 동작한다. Bit Setting 0 : Disable 1 : Enable - Locking Mode
C	1bit	**Carry Flag**를 의미한다. 이전 연산의 결과에서 Carry가 발생한 경우 설정된다. Bit Setting 0 : Carry Not Occured 1 : Carry Occured
Z	1bit	**Zero Flag**를 의미한다. 이전 연산의 결과가 '0'인 경우 설정된다. Bit Setting 0 : Result isn't Zero 1 : Result is Zero

Table 2.1: 상태 레지스터의 각 비트 정의 및 설정 방법(계속)

Tag	Width	Description
S	1bit	**Sign Flag**를 의미한다. 이전 연산의 결과가 음수인 경우 설정된다. Bit Setting 0 : Positive result 1 : Negative result
V	1bit	**Overflow Flag**를 의미한다. 이전 연산의 과정에서 Overflow가 발생한 경우 설정된다. Bit Setting 0 : Overflow Not Occured 1 : Overflow Occured

2.4.2 Program Counter(PC)

프로그램의 수행 위치를 가르키는 레지스터로서 다음에 수행되어야 할 명령의 주소를 지니고 있다. 단, 파이프라인 프로세서에서는 여러 개의 명령이 동시에 수행되고 있으므로, ID 단계에서 수행되는 명령을 기준으로 PC값이 설정되며, 분기가 결정된 경우 분기 목적 주소가 PC로 설정된다.

주의) 0bit은 항상 '0' 값을 가진다

2.4.3 Link Register(LR)

Jump and Link 형식의 분기(`JAL` , `JALR` 명령)를 수행하는 경우 복귀할 주소(link address)를 저장한다. 분기에서의 복귀는 JPLR명령을 이용하여 LR에 저장된 주소로 분기할 수 있으며, 만일 중첩된 분기에서 복귀할 때는 PUSH LR, POP PC의 명령어 Sequence를 이용할 수도 있다.

주의) 0bit은 항상 '0'값을 가진다

2.4.4 Extenstion Register(ER)

LERI [1]명령을 통하여 설정되며, 즉치 값이나 오프셋을 확장하는데 이용되며, 이 레지스터의 값은 SR의 E-flag가 '1'인 경우에만 valid하다. AE32000 계열의 프로세서는 LERI Folding기능을 지니고 있으므로, 아키텍쳐/마이크로아키텍쳐 수준에서 실제로 사용하지는 않는다. 따라서, ER을 PUSH하는 경우 '0'의 값이 PUSH되며, POP의 경우에도 프로세서의 동작상에 영향을 주지 않는다. 따라서, 실제적인 아키텍쳐 수준에서 ER은 dummy로 동작한다.

단, UDI(Undefined Instruction Interrupt)의 경우나 UII(Unimplemented Instruction Interrupt)에서는 해당 명령을 인터럽트 핸들러에서 분석하여 동작을 수행하기 위하여 ER의 값을 GAP(General Access Pointer) 형태로 보존한다.

2.4.5 Multiply Result Registers

Multiply Result Register는 곱셈(MUL , MULU / MAC 연산 및 MAC 관련 DSP 동작의 결과를 저장하는 레지스터로서 일반적인 경우에 64 비트의 크기를 지닌다. 이 결과 값의 상위 32비트는 MH로, 하위 32비트는 ML로 지정된다. MAC 연산의 경우 이 레지스터들은 accumulation register로서 사용되며, DSP 확장 옵션[2]을 사용한 경우 accumulator의 정확성을 잃지 않기 위하여 MRE 레지스터가 부가적으로 사용되어 Multiply Result Register의 상위 16비트[3]를 추가적으로 확장 할 수 있다. MRE [4] 레지스터는 일반적인 PUSH/POP에 의하여 저장/복원될 수 없으므로, 값의 보존이 필요한 경우 반드시 범용 레지스터를 경유하여 PUSH/POP을 수행하여야 한다.

2.4.6 CR0, CR1

자동 증가 메모리 지정 모드를 지원하기 위한 레지스터로서, 반복적인 for loop 및 DSP 연산을 효과적으로 지원하기 위하여 사용된다. 다음과 같은 주소 모드가 지원되며, 주소 모드에 따라 각 비트의 용도가 변경된다. 그림. 2.3과 표. 2.2는 각각 Counter Register의 필드 정보와 모드별 동작에 대한 기술이다.

- Auto Increment Mode : 12비트 카운터를 자동으로 incrementor만큼 증가 시킨다. 카운터는 항상 POST Increment된다.
  AE32000C 프로세서군에서 모두 사용할 수 있다.

[1]Load Extension Register with Immediate : 즉치 값으로서 ER을 확장하는 명령
[2]이 옵션에 대한 자세한 사항은 AE32000C-Lucida Processor User Guide를 참조
[3]구현에 따라 다를 수 있으나, 일반적으로 16비트가 추가된다
[4]MUL/MULU 명령어들은 이 레지스터를 사용하지 않는다. MAC과 연관성 있는 명령어들만 사용한다. 단 MAC 명령어와 연관성이 있는 명령어를 이용하는 경우에도 AE32000C-Lucida version에서만 MRE를 이용할 수 있다.

Figure 2.3: Counter Register field

- Auto Increment with End-offset Compare Mode : 12비트 카운터를 자동으로 incrementor만큼 증가 시킨다. 카운터는 항상 POST Increment되며 EOFF과의 비교는 PRE Increment 값이 사용된다. 카운터가 EOFF과 같을 경우 zero-flag가 인가된다. AE32000C 프로세서군에서 모두 사용할 수 있다.

- Wrap-around Auto Increment Mode : 12비트 카운터를 자동으로 incrementor만큼 증가 시킨다. 메모리 주소는 항상 POST Increment된다. 단, 카운터 값이 MASK에 의해 마스킹되므로 cicular 형태로 메모리 접근이 일어난다.
  DSP 확장 옵션을 사용해야 해당 모드를 사용할 수 있다.

- Wrap-around Auto Increment with End Check Mode : 기본 동작은 Wrap-around Auto Increment Mode와 같다. 단, 카운터가 '0'이 되는 경우 zero-flag가 인가 된다. 이때 비교에 사용되는 카운터는 POST Increment된 값을 사용한다.
  DSP 확장 옵션을 사용해야 해당 모드를 사용할 수 있다.

- Bit Reverse Mode : 카운터를 n-bit에 대해 증가 시키되, 이를 n-bit 단위로 bit reverse하여 메모리 주소를 생성한다. 이때, n-bit는 MASK에 의해 결정된다.
  예를들어 카운터가 0x1이고 MASK가 0xF인 경우, 메모리 주소의 오프셋 값은 0x8이 되고, 카운터가 0x3이고 MASK가 0xFF인 경우, 메모리 주소의 오프셋 값은 0xC0가 된다.
  DSP 확장 옵션을 사용해야 해당 모드를 사용할 수 있다.

Table 2.2: Counter Register의 각 필드 정의 및 동작

Tag	Width	Description
Mode	4bit	**Counter Register**의 동작 모드를 결정한다. 주소 모드를 지정하며, 지정된 모드에 따라 Counter Register의 다른 필드의 역할이 결정된다. Bit Setting 0000 : Auto Increment Mode 0001 : Auto Increment with End-offset Compare Mode 0100 : Wrap-around Auto Increment Mode 0101 : Wrap-around Auto Increment with End Check Mode 0110 : Bit Reverse Mode
Incrementor	4bit	**Counter**의 증분을 나타낸다. Counter에 더해질 계수를 지정한다. 현재는 항상 '4'의 값이 사용된다. Bit Setting 0000 : 4
End Offset	12bit	최종 값을 나타낸다. Auto Increment with End-offset Compare Mode에서 최종 값으로 사용된다. Counter 값이 최종 값과 일치하는 경우 SR의 zero-flag가 설정된다.
Mask	12bit	몇 비트 연산을 수행할 것인지 결정한다. Wrap-around Mode 및 Bit Reverse Mode에서 몇 비트 연산을 수행할 것인지 결정한다. Wrap-around Auto Increment with End Check Mode인 경우, 카운터가 최종 값에 이르면 SR의 zero-flag가 설정된다. Bit Reverse Mode에서는 Mask의 크기를 8비트 이하만 지원한다.
Counter	12bit	**Counter**의 값을 나타낸다. 실제적인 카운터의 값을 지니고 있다. 이 값이 index register와 더해져서 메모리 접근 주소로 사용된다.

2.4.7 SSP, ISP, USP

AE32000 프로세서는 각 프로세서 동작 모드에 해당하는 스택 포인터를 지닌다. 스택 포인터는 스택 영역의 최상위 위치(가장 최근에 PUSH된 entry의 주소)를 지정하며, 하위 2비트의 값은 항상 '0' 값을 지닌다. 스택의 성장은 항상 상위 주소에서 하위 주소 방향으로 이루어진다.(PUSH시 주소가 감소한다.)

스택 포인터는 프로세서 동작 모드에 따라 shadowing되며, 현재 프로세서 동작 모드 이외의 스택 포인터는 시스템 보조프로세서의 레지스터(SCPR13, SCPR14)를 통하여 접근이 가능하다.

- **SCPR13 : User Stack Pointer**

  Supervisor Mode, OSI Mode 에서 USP에 접근할 수 있도록 한다.

- **SCRP14 : OSI Stack Pointer / Supervisor Stack Pointer**

  Supervisor Mode 에서 ISP에 접근할 수 있으며, OSI Mode 에서 SSP에 접근할 수 있도록 한다.

각 프로세서 동작 모드를 사용하기 위해서는 프로그래밍 초기에 해당 스택 포인터를 지정해 주어야만 한다. 이때 LEA명령을 이용해 현재 프로세서 동작 모드에 해당하는 스택 포인터 값을 설정할 수 있다. 프로그램에서 하나 이상의 프로세서 동작 모드를 사용할 경우, 각 동작 모드를 변경해 가며 LEA 명령을 이용해 각 동작 모드에 해당하는 스택 포인터 값을 설정할 수 있다. 하지만, 아래의 예와 같이 시스템 보조프로세서 레지스터를 이용하여 동작 모드의 변경 없이 다른 동작 모드의 스택 포인터를 설정할 수도 있다. 이때 시스템 보조프로세서 레지스터를 접근하기 위한 `MVTC` 명령이 사용된다.

2.5 Exceptions

Exception은 프로세서의 정상적인 흐름을 방해하는 모든 상황을 의미한다. AE32000 프로세서는 프로그램 수행중에 exception이 발생하였을 경우, 그림. 2.4와 같은 과정을 통해 exception을 처리한다.

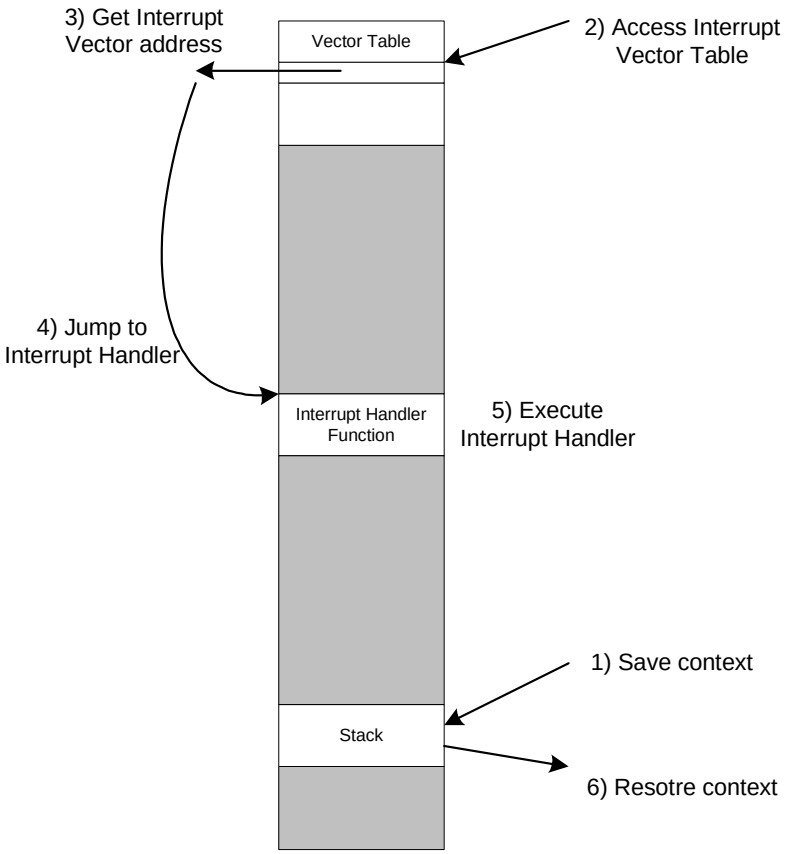


Figure 2.4: Exception 처리 과정

1. **Save context**

   Exception 발생과 관련이 있는 명령이 있는지 확인하여, exception 처리 이후 해당 명령부터 다시 수행하기 위해 현재의 프로세서 상태 및 해당 명령의 주소를 스택 영역에 저장한다.

2. **Access Interrupt Vector Table**

발생된 exception의 종류에 해당하는 인터럽트 벡터 테이블에 접근한다[5]. 인터럽트 벡터 테이블에 대한 자세한 내용은 2.5.2절을 참조하기 바란다.

3. **Get Interrupt Vector address**

   인터럽트 벡터 테이블을 접근하여 exception을 처리하기 위한 인터럽트 핸들러의 함수 포인터를 가지고 온다.

4. **Jump to Interrupt Handler**

   Exception을 처리하기 위한 인터럽트 핸들러 함수로 분기한다.

5. **Execute Interrupt Handler**

   Exception을 처리하기 위한 인터럽트 핸들러 함수를 수행한다. 인터럽트 핸들러에서 사용할 범용 레지스터들은 스택을 통해 보관되어야 하며, exception을 처리한 이후 인터럽트 핸들러 함수를 빠져 나올 때는 저장된 범용 레지스터와 함께 (1)단계에서 저장된 프로세서 상태 및 복구 명령의 주소를 복구시켜야 한다[6].

6. **Restore context**

   인터럽트 핸들러 함수를 수행한 이후 exception 발생 이전의 프로세서 상태로 복구시킨다.

2.5.1 Exception의 종류

1) Reset

코어 프로세서의 초기화 동작을 요구하고자 할 때, 외부에서 코어 프로세서로 입력해 주는 신호를 통하여 발생한다. 이 인터럽트를 받는 경우 프로세서는 초기화를 수행한다. 초기화시 모든 범용 레지스터 및 특수 목적 레지스터는 'unknown'값 또는 '0'의 값으로 변경된다[7]. 리셋 벡터는 Power-on Configration에 따라 달라진다.

2) External Hardware Interrupt

외부 하드웨어 모듈에서 요청하는 인터럽트를 의미한다. 현재 상태의 PC, SR 값을 **관리자 스택**에 PUSH한 후 인터럽트 핸들러를 수행한다. 다른 PUSH가 필요한 레지스터들은 인터럽트 핸들러에 의하여 관리되어야 한다. 외부 인터럽트는 다음 두 가지 방식으로 인터럽트 벡터를 지정할 수 있다.

[5] 인터럽트가 발생하면, 그 인터럽트의 종류에 따라 특정 번호가 주어지며 이를 기반으로 인터럽트 벡터 테이블에 접근한다.
[6] 해당 동작은 인터럽트 핸들러 함수 앞부분에 '#pragma interupt_handler' 를 추가하여 동작되는 부분이다. 자세한 내용은 "Software User Guide"를 참조하기 바란다.
[7] 구현 모델에 따라 변경될 수 있다.

- **Auto-vectored Interrupt**:

  인터럽트 벡터가 지정되어 있는 경우로서, 인터럽트가 발생한 경우 내정되어 있는 인터럽트 벡터로 이동한다.

- **Vectored Interrupt**:

  외부 인터럽트 컨트롤러에서 해당 인터럽트의 번호를 넘겨줌으로서, 인터럽트 벡터의 위치를 판별하여 선택된 인터럽트 벡터로 이동한다.

3) Software Interrupt

소프트웨어적으로 발생시키는 인터럽트를 의미한다. 관리자의 자원을 사용자 모드에서 접근하고자 하는 경우 호출하며, 일반적으로 system call을 위하여 사용된다. 현재 상태의 PC, SR 값을 **관리자 스택**에 PUSH한 후, 지정된 인터럽트 번호의 인터럽트 핸들러를 수행한다.

4) Non Maskable Interrupt

5) System Coprocessor Interrupt

메모리 접근 과정에서 발생하는 인터럽트들을 의미한다. Memory Management Unit에 의해 발생되며, 다음과 같은 경우에 발생된다.

- **Access violation**: 관리자 영역으로 지정된 부분을 사용자가 접근하는 경우
- **Address alignment error**:
- **TLB miss**: TLB를 사용하여 가상 주소를 접근하고자 할 때 TLB에 entry가 존재하지 않는 경우

6) Coprocessor Interrupt

보조 프로세서의 접근 과정에서 발생하는 인터럽트를 의미한다. 또한, 이 인터럽트는 연산을 수행하는 보조 프로세서의 연산 결과를 polling하여 발생시킬 수 있다. 보조 프로세서 접근 과정에서 발생하는 인터럽트는 관리자 전용으로 지정된 보조 프로세서를 사용자 모드에서 접근할 때 발생한다.

7) Breakpoint & Watchpoint Interrupt

OSI에서 제공하는 디버깅 기능을 제공하기 위한 인터럽트이다. OSI 모듈에서 지정된 breakpoint[8]와 watchpoint[9]의 조건을 만족하였을 때 이 인터럽트가 호출되어 프로세서의 동작 모드를 OSI 모드로 변경시킨다.

8) Bus Error

명령어 버스나 데이터 버스의 복구 불가능한 버스의 오류를 의미한다. 목적 주소에 메모리가 존재하지 않는 경우 발생할 수 있다.

- 인터럽트 벡터 테이블 주소가 0x00000010으로 고정되어 있다[10].

9) Double Fault

인터럽트가 발생하여 인터럽트 핸들러로 진입하는 과정에서 오류가 발생한 경우를 의미한다. 이 경우 supervisor stack의 영역에 문제가 있거나, 인터럽트 핸들러의 위치를 변경시키는 Vector base register의 설정이 잘못되어 있는 경우 발생할 수 있다.

- 인터럽트 벡터 테이블 주소가 0x0000000C로 고정되어 있다[11].

10) Undefined Instruction Exception

정의되지 않은 명령이 입력된 경우 발생하는 예외이다. 이 경우 버전에 따라 오류를 발생시키거나, NOP로 처리할 수 있다.

11) Unimplemented Instruction Exception

명령어 셋에는 정의되어 있으나, 해당 버전에서는 구현되어 있지 않은 명령을 의미한다. 이 경우 인터럽트를 통하여 소프트웨어적으로 에뮬레이션함으로서 값을 얻어낸다. 호환성을 높이기 위하여 사용된다.

2.5.2 Interrupt Vector Table

Exception의 처리를 위하여 프로세서는 인터럽트 핸들러 함수를 읽어와야 하는데, 이 인터럽트 핸들러 함수의 주소는 인터럽트 벡터 테이블에 존재한다. AE32000 프로세서의 인터럽트 벡터 테이블은 기본적으로 0x0번지에 존재하며, 뒤에 나올 2.5.3절에서 설명할

[8] 관찰할 명령어 접근 주소를 의미한다.
[9] 관찰할 데이터 접근 주소를 의미한다.
[10] 자세한 내용은 2.5.2절을 참조하기 바란다.
[11] 자세한 내용은 2.5.2절을 참조하기 바란다.

벡터 베이스를 변경하여 위치를 이동시킬 수 있다. 단, 몇 가지 인터럽트들의 벡터 위치는 이동시킬 수 없다. 그림. 2.5는 벡터 테이블의 구성을 나타내며, 그림에서 shade 처리되어 있는 인터럽트들은 벡터 위치를 이동시키는 것이 불가능하다.

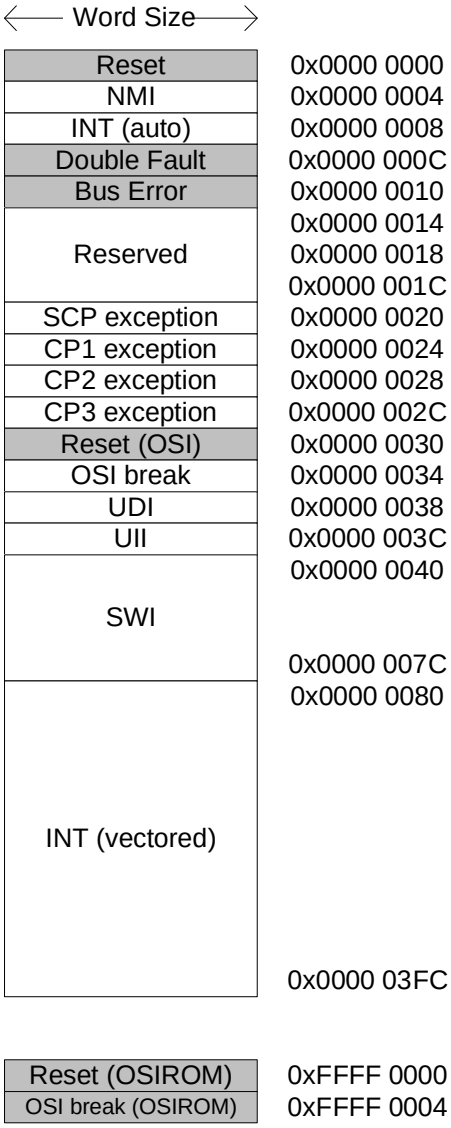


Figure 2.5: 인터럽트 벡터 테이블의 구성

인터럽트 벡터는 앞에서 언급하였듯이 0x0번지에 위치해야 하므로, C 컴파일러를 사용하는 경우에는 loader script에서 반드시 이 위치가 0x0에 올 수 있도록 다음과 같이 memory layout을 지정하여야 한다.

```
1  SECTIONS
2  {
```

```
3     /* Read-only sections, merged into text segment: */
4     . = 0x0;
5     .text      :
6     {
7       *(.vects) // Interrupt vector location
```

위의 '.vects'는 section을 가르키는 symbol로서, 로더에서 이러한 section symbol을 기반으로 메모리를 배치하게 된다. 물론, 이러한 section symbol은 코드 내에 존재해야 의미를 지닌다. 코드 내에 정의된 '.vects'는 다음과 같다.

```
1   void (*vector_table[])(void) __attribute__((section (".vects")))= {
2       reset_vector              , /* V00 : Reset Vector         */
3       nmi_autovector            , /* V01 : NMI Vector           */
4       interrupt_autovector      , /* V02 : Interrupt Auto Vector */
5       double_fault_exception    , /* V03 : Double fault Vector   */
6       bus_error_exception       , /* V04 : Bus Error Exception   */
7       NOTUSEDISR                , /* V05 : Reserved             */
8       NOTUSEDISR                , /* V06 : Reserved             */
9       NOTUSEDISR                , /* V07 : Reserved             */
10      coprocessor0_exception    , /* V08 : Coprocessor0 Exception*/
11      coprocessor1_exception    , /* V09 : Coprocessor1 Exception*/
12      coprocessor2_exception    , /* V0A : Coprocessor2 Exception*/
13      coprocessor3_exception    , /* V0B : Coprocessor3 Exception*/
14      osi_reset_vector          , /* V0C : OSI reset vector     */
15      osi_break_exception       , /* V0D : OSI break exception   */
16        ...
17        ...
18        ...
19  };
```

위에서 볼 수 있듯이 벡터 테이블은 인자를 지니지 않고, 반환값을 지니지 않는 함수 포인터들의 배열로 이루어져 있다. 또한, 테이블의 속성으로서 section symbol을 '.vects'로서 지정하여 로더에서 이를 인식할 수 있도록 해 두었다.

2.5.3 Vector Base

벡터 베이스란 인터럽트 벡터 테이블의 위치를 의미한다. 기본적으로 인터럽트 벡터 테이블은 0x0번지에 위치하며, 벡터 베이스를 변경하여 인터럽트 벡터 테이블의 위치를 이동시킬 수 있다. 단, 앞서 설명했듯이 몇 가지 인터럽트들의 벡터 위치는 이동시킬 수 없으며, 이동될 수 없는 벡터들은 그림. 2.5을(를) 참조하기 바란다. 벡터 베이스는 시스템 보조 프로세서의 12번 레지스터인 벡터 베이스 레지스터에 저장되며, 이 값이 지정된 이후부터 대부분의 인터럽트는 변경된 인터럽트 벡터 테이블로부터 벡터 주소를 읽어온다. 즉, 시스템 보조 프로세서의 12번 레지스터는 프로세서 리셋시 0의 값을 가지며, 레지스터를 설정하게 되면 인터럽트 벡터 테이블의 위치가 변경되는 것이다. **이때, 단순히 인터럽트 벡터 테이블의 위치가 변경된 것이지 벡터 테이블의 데이터(인터럽트 핸들러의 주소)가 옮겨진 것은 아니다. 따라서, 기존의 인터럽트 벡터 테이블의 데이터를 복사한 위치이거나, 별도의 인터럽트 벡터 테이블이 존재하는 위치로 벡터 베이스를 지정하여야 한다.**

벡터 베이스 레지스터는 벡터 테이블의 접근을 ROM 영역에서 RAM 영역으로 변경하는 경우, 동작의 수행 시간을 단축시키고 인터럽트의 hooking을 가능하게 할 수 있다. 벡터 베이스에서 지정된 벡터 테이블 영역은 가상 메모리를 사용하는 경우 반드시 TLB(Traslate Look-aside Buffer) 내에 존재해야 한다. 이는 AE32000 프로세서의 경우 software managed TLB를 채용하고 있으므로, 벡터 처리 과정 중에 벡터 테이블 접근에서 TLB miss exception이 발생하는 경우 불필요한 double fault가 발생할 수 있기 때문이다.

2.5.4 Exception Priority

AE32000 프로세서의 인터럽트는 인터럽트들 간의 우선 순위가 존재한다. 프로세서 내부적으로 지니고 있는 우선 순위는 표. 2.3과 같다. 인터럽트 간의 우선 순위는 **같은 시점에서 발생한 하나 이상의 인터럽트**의 우선 순위를 가리는데 사용되며, 일반적으로는 먼저 요청된 인터럽트가 우선한다.

Table 2.3: 인터럽트 우선 순위

순위	설명	약어
1	Reset	RST
2	Bus Error	BERR
3	Double Fault	DF
4	OSI Exception	OSI
5	Coprocessor Exception	CP

Table 2.3: 인터럽트 우선 순위(계속)

순위	설명	약어
6	System-Coprocessor Exception	CP0
7	Non-Maskable Interrup	NMI
8	Software Interrupt	SWI
9	Interrupt	INT
10	Halt	HALT
11	Undefined Instruction Exception	UDI
12	Unimplemented Instruction Exception	UII

2.6 Memory Interface

2.6.1 Address Space

AE32000은 32bit프로세서이므로 32bit주소 영역을 지니고 있다. 즉, 4GB 까지의 선형 메모리 영역에 대한 접근이 가능하다. 내장형 응용 분야에서 4GB의 메모리를 요구하는 경우는 매우 제한적이므 로, 실제 주소 영역을 제한할 수 있다. 하지만, 이러한 경우에도 시스템 보조 프로세서에서 제공하는 가상 주소를 이용하여 32비트 주소 영역을 사용할 수 있다.

2.6.2 Endianness

Endian이란 한 워드 내에서의 데이터를 어떠한 방향으로 정렬할 것인지를 규정하는 것이다. Endian은 크게 Big Endian과 Little Endian으로 나뉜다. Big Endian은 말 그대로 워드의 큰 부분(상위)이 워 드의 끝에 위치하는 것을 나타내며, Little Endian은 말 그대로 워드의 작 은 부분(하위)이 워드의 끝에 위치하는 것을 의미한다. 아래의 그림은 한 워드를 특정 위치에 저장한 경우의 그림이다.

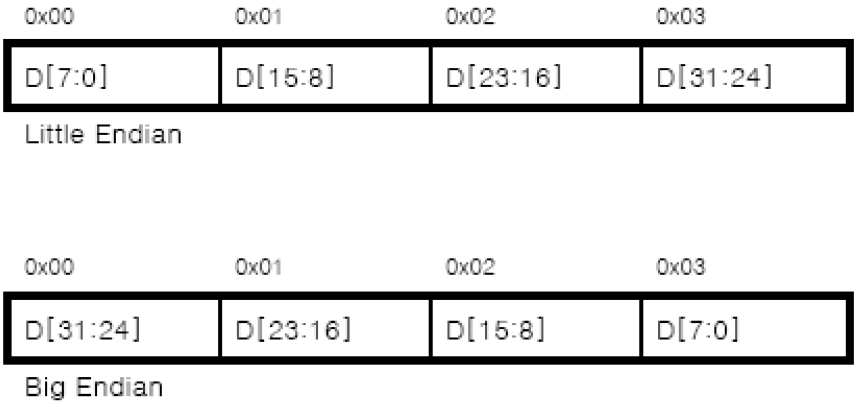


Figure 2.6: Endianess

위의 그림과 같이 Little Endian의 경우 작은 값을 워드의 하위에 위치시 킨다. Little Endian의 장점은 데이터의 크기와 관계없이 동일한 접근 방식 을 사용하면 되므로, 데이터 크기가 커지는 math routine등에서 사용이 쉽다는 점이다. Big Endian의 경우 위의 그림에서와 같이 상위의 값이 워드의 하위에 위 치한다. Big Endian의 장점은 최상위 바이트가 워드의 최하위에 위치하고 있으므로, 부호를 판별하기 용이하다는 장점을 지니고 있다. (즉, 데이터 길이와 관계없이 offset을 0으로 하고 바이트를 읽는 경우 그 데이터의

부호를 판별할 수 있다. 이는 byte단위로 입력되는 스트림에서 연산 종류 를 빠르게 결정할 수 있다는 장점을 지닌다). 단점은 데이터의 크기가 바 뀌는 경우 그 접근 방식이 달라져야 한다는 점이다. AE32000은 구현에 따라 Little-Endian version과 Bi-Endian버전이 있다. 또한 endian이 구분에 따라 다른 컴파일러를 이용해야 하므로 주의를 기 울여야 한다.

2.6.3 Unaligned Menory Access

정렬되지 않은 메모리 접근은 워드 단위로 정렬된 메모리에서 워드 단위에서 나올 수 없는 주소로 워드에 접근을 시 도한 경우이다. 예를 들면 32bit시스템에서 0x2라는 주소로 워드를 접근 하는 경우 메모리 구성상 0x0이라는 주소에서 상위 2바이트를 읽어오고, 0x4라는 주소에서 하위 2바이트를 읽어와서 정렬해야 하는데, 이러한 메 모리 접근을 unaligned memory access라 한다. AE32000은 구현에 따라 unaligned memory access에 대하여 exception 을 발생시키는 버전과 이를 지원하는 버전으로 구분된다. 이를 지원하지 않는 버전의 경우에도 exception handling을 통하여 이를 지원할 수 있으 므로, 큰 문제는 되지 않으나 속도 차이가 크게 날 수 있으므로 항상 염 두에 두고 프로그램을 작성하여야 한다.

2.6.4 Prefetching & Self-Modifying Code

이 절에서는 프로그래밍에 주의해야 할 부분을 설명한다.

- **Prefetching** - Prefetch는 다수의 명령어를 프로세서에서 먼저 읽어 들이 는 동작을 의미한다. AE32000에서는 구현에 따라 틀리지만, 메모리와 프 로세서간의 속도 차이를 극복하기 위하여 캐쉬 메모리를 사용하는 경우가 많다. 이 경우 캐쉬 미스가 발생한 경우 명령어 접근에 시간이 오래 걸리 며, 명령어 fetch가 끝날 때까지 프로세서는 정지한다. Prefetch는 이러한 문제를 줄일 수 있는 방법으로서 사용된다. AE32000에서는 이외에 LERI 명령어의 folding을 위하여 사용된다.

- **Self-Modifying Code** - Self-Modifying Code 는 프로그램 수행 도중에 프 로그램 자체를 변경하는 프로그램을 의미한다. 이러한 프로그램은 네트웍 을 firmware 업그레이드 등의 용도로 유용하게 사용될 수 있으나, 하드웨 어적인 복잡도나 context의 관리가 어려워진다는 단점을 지니고 있으므로 대부분의 프로세서에서는 이를 허용하지 않는다. AE32000는 기본적으로 자체 변경 코드를 지원하지 않으며, 이러한 동작 을 수행하기 위해서는 프로그램에서 변경할 프로그램을 일정 RAM영역에 둔 다음에 RESET interrupt 수행하도록 하고, 리셋 벡터 부분에서 해당 RAM영역의 데이터를 writeable-ROM(eg. flash ROM)으로 옮기는 동작을 수행하면 된다.

2.6.5 Memory Map

AE32000는 기본적으로 최하위 주소에 벡터 테이블을 위치시키도록 되어 있으며, 이 부분에 대해서는 변경 할 수 없다. (벡터 테이블의 위치는 Vector Base를 변경함으로서, 그 위치를 조정할 수 있으나 inexact exception processing을 하는 exception은 항상 0x0번지를 지닌다. 2.5.11 참조) 또한, 스택은 상위 주소 영역에서 하위 주소 영역으로 성장 하므로, 스택 영역은 메모리의 상위 부분에 위치하여야 하며 충분한 공간 이 주어져야 한다. AE32000에서는 이 두 가지 조건을 만족하는 경우 어떠한 형태의 메모리 맵이라도 구성 가능하다.

2.6.6 Memory Mapped I/O

AE32000은 장치 접근을 위한 dedicated instruction이 없으며, 모두 memory mapped방식을 취하고 있다. 이 장치들에 대한 메모리 할당은 버스에서 지정하는 것을 따르도록 한다.

Chapter 3

Instruction Set Highlights

AE32000는 코드 밀도를 높이기 위하여 다양한 명령어를 갖추고 있다. 본 장에서는 AE32000의 명령어에 대한 대략적인 구분과 주의 사항에 대해 설명한다.

3.1 Binary Encoding

AE32000 프로세서의 binary encoding과 간략한 Mnemonic은 그림. 3.1과 같다. Mnemonic에 사용된 지시어들은 다음과 같은 뜻을 가진다.

- **dst**

  목적 레지스터(destination register)로 범용 레지스터 중의 하나를 뜻한다. 연산의 결과를 저장하는 레지스터이다.

- **src, src1, src2**

  원천 레지스터(source register)로 범용 레지스터 중의 하나를 뜻한다. 연산에 사용되는 오퍼랜드를 의미한다.

- **idx**

  인덱스 레지스터(index register)로 범용 레지스터 중의 하나를 뜻한다. [1] index addressing에서 인덱스 값으로 사용된다.

- **offset**

  즉치 값(immediate value)으로 정수 값을 가진다. index addressing, stack addressing, pc reltive program memory addressing에서 변위(offset) 값으로 사용된다.

- **imm**

  즉치 값(immediate value)으로 정수 값을 가진다. 연산에 직접 사용되거나 LERI 명령에 의해 확장되어 연산에 사용된다.

- **list**

  PUSH, POP 명령에 사용되는 범용 레지스터, 특수 목적 레지스터의 집합을 나타낸다.

- **sftamt**

  쉬프트 연산에 사용되는 쉬프트 횟수를 나타낸다. 정수 값으로 0에서 31 사이의 값을 가진다.

- **sftreg**

  쉬프트 연산에 사용되는 쉬프트 횟수를 간직하는 레지스터로 범용 레지스터 중의 하나를 뜻한다. 32비트 레지스터 가운데 하위 5비트가 사용된다.

[1] LDAU, STAU 명령에서는 다른 의미를 가진다.

- **flag pos**

  SET, CLR 명령에 사용되는 정수 값으로 상태 레지스터의 0에서 15 사이의 각 비트를 나타낸다.

- **or**

  Counter Register의 number를 나타낸다.

- **cpno**

  보조프로세서(Coprocessor)의 number를 나타낸다. 0에서 3 사이의 정수 값을 가진다.

- **src_grp**

  UNPACK 연산에 사용되는 연속된 두개의 범용 레지스터를 뜻한다. 하위에 0과 1이 생략된 두 개의 범용 레지스터를 나타낸다.(예. 010 - 4,5번 레지스터)

Figure 3.1: Instruction Set Highlights

Instruction	f	e	d	c	b	a	9	8	7	6	5	4	3	2	1	0	Mnemonic
LD	0	0	0	0	dst				offset				idx				LD (idx, off), dst
ST	0	0	0	1	src				offset				idx				ST src, (idx, off)
LDB	0	0	1	0	dst				0	offset			idx				LDB (idx, off), dst
LDS	0	0	1	0	dst				1	offset			idx				LDS (idx, off), dst
STB	0	0	1	1	dst				0	offset			idx				STB src, (idx, off)
STS	0	0	1	1	dst				1	offset			idx				STS src, (idx, off)
LERI	0	1	imm														LERI imm
LDBU	1	0	0	0	dst				0	offset			idx				LDBU (idx, off), dst
LDSU	1	0	0	0	dst				1	offset			idx				LDSU (idx, off), dst
LD	1	0	0	1	dst				0	offset							LD (SP, off), dst
ST	1	0	0	1	src				1	offset							ST src, (SP, off)
LDI	1	0	1	0	dst				imm								LDI imm, dst
PUSH	1	0	1	1	0	0	0	0	REGLIST(7,6,5,4,3,2,1,0)								PUSH list
PUSH	1	0	1	1	0	0	1	0	(15,14,13,12,11,10,9,8)								PUSH list
PUSH	1	0	1	1	0	1	0	0	(SR,PC,LR,ER,MH,ML,CR1,CR0)								PUSH list
POP	1	0	1	1	0	0	0	1	REGLIST(7,6,5,4,3,2,1,0)								POP list
POP	1	0	1	1	0	0	1	1	(15,14,13,12,11,10,9,8)								POP list
POP	1	0	1	1	0	1	0	1	(SR,PC,LR,ER,MH,ML,CR1,CR0)								POP list
LEA(ASPO)	1	0	1	1	0	1	1	0	offset								LEA (SP, imm), SP
MRS	1	0	1	1	0	1	1	1	0	imm			dst				MRS imm, dst
	1	0	1	1	0	1	1	1									Reserved
ADD	1	0	1	1	1	0	0	0	dst				src/imm				ADD src/imm, dst
ADC	1	0	1	1	1	0	0	1	dst				src/imm				ADC src/imm, dst
SUB	1	0	1	1	1	0	1	0	dst				src/imm				SUB src/imm, dst
SBC	1	0	1	1	1	0	1	1	dst				src/imm				SBC src/imm, dst
AND	1	0	1	1	1	1	0	0	dst				src/imm				AND src/imm, dst
OR	1	0	1	1	1	1	0	1	dst				src/imm				OR src/imm, dst
XOR	1	0	1	1	1	1	1	0	dst				src/imm				XOR src/imm, dst
CMP	1	0	1	1	1	1	1	1	dst				src/imm				CMP src/imm, dst
ASR	1	1	0	0	dst				0	sftamt					0	0	ASR sftamt, dst
LSR	1	1	0	0	dst				0	sftamt					0	1	LSR sftamt, dst
ASL	1	1	0	0	dst				0	sftamt					1	0	ASL sftamt, dst
SSL	1	1	0	0	dst				0	sftamt					1	1	SSL sftamt, dst
ASR	1	1	0	0	dst				1	0	sftreg				0	0	ASR sftreg, dst
LSR	1	1	0	0	dst				1	0	sftreg				0	1	LSR sftamt, dst
ASL	1	1	0	0	dst				1	0	sftreg				1	0	ASL sftreg, dst
SSL	1	1	0	0	dst				1	0	sftreg				1	1	SSL sftreg, dst

Instruction	f	e	d	c	b	a	9	8	7	6	5	4	3	2	1	0	Mnemonic
ADDQ	1	1	0	0	dst				1	1	0	imm					ADDQ imm, dst
CMPQ	1	1	0	0	dst				1	1	1	imm					CMPQ imm, src
JNV	1	1	0	1	0	0	0	0	offset								JNV lable
JV	1	1	0	1	0	0	0	1	offset								JV lable
JP	1	1	0	1	0	0	1	0	offset								JP lable
JM	1	1	0	1	0	0	1	1	offset								JM lable
JNZ	1	1	0	1	0	1	0	0	offset								JNZ lable
JZ	1	1	0	1	0	1	0	1	offset								JZ lable
JNC	1	1	0	1	0	1	1	0	offset								JNC lable
JC	1	1	0	1	0	1	1	1	offset								JC lable
JGT	1	1	0	1	1	0	0	0	offset								JGT lable
JLT	1	1	0	1	1	0	0	1	offset								JLT lable
JGE	1	1	0	1	1	0	1	0	offset								JGE lable
JLE	1	1	0	1	1	0	1	1	offset								JLE lable
JHI	1	1	0	1	1	1	0	0	offset								JHI lable
JLS	1	1	0	1	1	1	0	1	offset								JLS lable
JMP	1	1	0	1	1	1	1	0	offset								JMP lable
JAL	1	1	0	1	1	1	1	1	offset								JAL lable
EXTB	1	1	1	0	0	0	0	0	0	0	0	0	dst				EXTB dst
EXTS	1	1	1	0	0	0	0	0	0	0	0	1	dst				EXTS dst
MFMRE	1	1	1	0	0	0	0	0	0	0	1	0	dst				MFMRE dst
MTMRE	1	1	1	0	0	0	0	0	0	0	1	1	src				MTMRE src
CVB	1	1	1	0	0	0	0	0	0	1	0	0	dst				CVB dst
CVS	1	1	1	0	0	0	0	0	0	1	0	1	dst				CVS dst
NEG	1	1	1	0	0	0	0	0	0	1	1	0	dst				NEG dst
NOT	1	1	1	0	0	0	0	0	0	1	1	1	dst				NOT dst
JR	1	1	1	0	0	0	0	0	1	0	0	0	src				JR src
JALR	1	1	1	0	0	0	0	0	1	0	0	1	src				JALR src
JPLR	1	1	1	0	0	0	0	0	1	0	1	0					JPLR
NOP	1	1	1	0	0	0	0	0	1	0	1	1					NOP
SWI	1	1	1	0	0	0	0	0	1	1	0	0	imm				SWI exception_no
STEP	1	1	1	0	0	0	0	0	1	1	0	1					STEP
HALT	1	1	1	0	0	0	0	0	1	1	1	0	imm				HALT halt_level
BRKPT	1	1	1	0	0	0	0	0	1	1	1	1					BRKPT
CNT0	1	1	1	0	0	0	0	1	0	0	0	0	src				CNT0 src
CNT1	1	1	1	0	0	0	0	1	0	0	0	1	src				CNT1 src
SET	1	1	1	0	0	0	0	1	0	0	1	0	flag pos				SET position
CLR	1	1	1	0	0	0	0	1	0	0	1	1	flag pos				CLR position
MTML	1	1	1	0	0	0	0	1	0	1	0	0	src				MTML src
MTMH	1	1	1	0	0	0	0	1	0	1	0	1	src				MTMH src
MFML	1	1	1	0	0	0	0	1	0	1	1	0	dst				MFML dst
MFMH	1	1	1	0	0	0	0	1	0	1	1	1	dst				MFMH dst
MTCR0	1	1	1	0	0	0	0	1	1	0	0	0	src				MTCR0 src
MTCR1	1	1	1	0	0	0	0	1	1	0	0	1	src				MTCR1 src
SYNC	1	1	1	0	0	0	0	1	1	0	1	0					SYNC
EXJ	1	1	1	0	0	0	0	1	1	0	1	1					EXJ
LEA(LSEA)	1	1	1	0	0	0	0	1	1	1	0	0	idx				LEA (idx, imm), SP
LEA(LESA)	1	1	1	0	0	0	0	1	1	1	0	1	dst				LEA (SP, imm), dst
MFCR0	1	1	1	0	0	0	0	1	1	1	1	0	dst				MFCR0 dst
MFCR1	1	1	1	0	0	0	0	1	1	1	1	1	dst				MFCR1 dst
AVGB	1	1	1	0	0	0	1	0	dst			0	src/imm				AVGB src/imm, dst
AVGS	1	1	1	0	0	0	1	0	dst			1	src/imm				AVGS src/imm, dst

Instruction	f	e	d	c	b	a	9	8	7	6	5	4	3	2	1	0	Mnemonic
TST	1	1	1	0	0	0	1	1	src2				src1				TST src1, src2
LEA	1	1	1	0	0	1	0	0	dst				idx				LEA (idx, imm), dst
ROR	1	1	1	0	0	1	0	1	0	imm			dst				ROR imm, dst
ROL	1	1	1	0	0	1	0	1	1	imm			dst				ROL imm, dst
MUL	1	1	1	0	0	1	1	0	dst			0	src/imm				MUL src/imm, dst
MULU	1	1	1	0	0	1	1	0	dst			1	src/imm				MULU src/imm, dst
MAC	1	1	1	0	0	1	1	1	src2				src1/imm				MAC src1/imm, src2
LDB	1	1	1	0	1	0	0	0	0	offset			dst				LDB (SP, imm), dst
LDS	1	1	1	0	1	0	0	0	1	offset			dst				LDS (SP, imm), dst
LDBU	1	1	1	0	1	0	0	1	0	offset			dst				LDBU (SP, imm), dst
LDSU	1	1	1	0	1	0	0	1	1	offset			dst				LDSU (SP, imm), dst
STB	1	1	1	0	1	0	1	0	0	offset			src				STB src, (SP, imm)
STS	1	1	1	0	1	0	1	0	1	offset			src				STS src, (SP, imm)
LDAU	1	1	1	0	1	0	1	1	0	idx		cr	dst				LDAU CRNO, idx, dst
STAU	1	1	1	0	1	0	1	1	1	idx		cr	src				STAU CRNO, src, idx
CPCMD	1	1	1	0	1	1	cpno		0	0	cpcmd						CPCMD CPNO, cpcmd
GETCn	1	1	1	0	1	1	cpno		0	1	0	0	status				GETC CPNO, bit_position
EXECn	1	1	1	0	1	1	cpno		0	1	0	1	status				EXEC CPNO, bit_position
MVTCn	1	1	1	0	1	1	cpno		0	1	1	0	dst				MVTC CPNO, cpdst
MVFCn	1	1	1	0	1	1	cpno		0	1	1	1	src				MVFC CPNO, cpdst
LDCn	1	1	1	0	1	1	cpno		1	0	0	0	dst				LDC CPNO, (R1, imm), cpdst
LDCn	1	1	1	0	1	1	cpno		1	0	0	1	dst				LDC CPNO, (SP, imm), cpdst
STCn	1	1	1	0	1	1	cpno		1	0	1	0	src				STC CPNO, cpsrc, (R1, imm)
STCn	1	1	1	0	1	1	cpno		1	0	1	1	src				STC CPNO, cpsrc, (SP, imm)
	1	1	1	0	1	1			1	1							Reserved
MACB	1	1	1	1	0	0	0	0	src2				src1/imm				MACB src1/imm, src2
MACS	1	1	1	1	0	0	0	1	src2				src1/imm				MACS src1/imm, src2
MSOPB	1	1	1	1	0	0	1	0	src2				src1/imm				MSOPB src1/imm, src2
MSOPS	1	1	1	1	0	0	1	1	src2				src1/imm				MSOPS src1/imm, src2
SADDB	1	1	1	1	0	1	0	0	dst				src/imm				SADDB src/imm, dst
SADDS	1	1	1	1	0	1	0	1	dst				src/imm				SADDS src/imm, dst
SADUB	1	1	1	1	0	1	1	0	dst				src/imm				SADUB src/imm, dst
SADUS	1	1	1	1	0	1	1	1	dst				src/imm				SADUS src/imm, dst
SADD	1	1	1	1	1	0	0	0	dst				src/imm				SADD src/imm, dst
UPKLS	1	1	1	1	1	0	0	1	src_grp			0	dst				UPKLS src_grp, dst
UPKHS	1	1	1	1	1	0	0	1	src_grp			1	dst				UPKHS src_grp, dst
MIN	1	1	1	1	1	0	1	0	dst			0	src/imm				MIN src/imm, dst
MAX	1	1	1	1	1	0	1	0	dst			1	src/imm				MAX src/imm, dst
ABS	1	1	1	1	1	0	1	1	0	0	0	0	dst				ABS dst
	1	1	1	1	1	0	1	1									Reserved
UPK0LB	1	1	1	1	1	1	0	0	src_grp			0	dst				UPK0LB src_grp, dst
UPK0HB	1	1	1	1	1	1	0	0	src_grp			1	dst				UPK0HB src_grp, dst
UPK1LB	1	1	1	1	1	1	0	1	src_grp			0	dst				UPK1LB src_grp, dst
UPK1HB	1	1	1	1	1	1	0	1	src_grp			1	dst				UPK1HB src_grp, dst
UPK2LB	1	1	1	1	1	1	1	0	src_grp			0	dst				UPK2LB src_grp, dst
UPK2HB	1	1	1	1	1	1	1	0	src_grp			1	dst				UPK2HB src_grp, dst
UPK3LB	1	1	1	1	1	1	1	1	src_grp			0	dst				UPK3LB src_grp, dst
UPK3HB	1	1	1	1	1	1	1	1	src_grp			1	dst				UPK3HB src_grp, dst

3.2 Memory Access

AE32000 프로세서에서 지원되는 메모리 접근 명령들은 표. 3.1에 나열되어 있다.

Table 3.1: Memory Access Instructions

Class	Details	Mnemonic
Load	Word	LD (%Ri/%SP, imm), %Rd
	Byte (unsigned)	LDBU (%Ri/%SP, imm), %Rd
	Byte (signed)	LDB (%Ri/%SP, imm), %Rd
	Short (unsigned)	LDSU (%Ri/%SP, imm), %Rd
	Short (signed)	LDS (%Ri/%SP, imm), %Rd
	Word (auto increment)	LDAU CRNO, %Ri, %Rd
Load Multiple	Pop	POP reg list
Store	Word	ST %Rs, (%Ri/%SP, imm)
	Byte	STB %Rs, (%Ri/%SP, imm)
	Short	STS %Rs, (%Ri/%SP, imm)
	Word (auto increment)	STAU CRNO, %Rs, %Ri
Store Multiple	Push	PUSH reg list

- Byte, Short(Half Word), Word 크기로 접근이 가능하며, Load의 경우 Byte/Short 크기의 데이터를 signed/unsigned extension을 통해 32비트 레지스터에 채울 수 있다.
- PUSH/POP 명령은 하나의 명령을 통해 다수의 레지스터에 대한 Load/Store를 할 수 있다.
- LD/ST auto increment 명령어들은 AE32000C 버젼에서 지원하는 DSP연산(Auto Increment Mode)을 강화하기 위해 추가되었다.

3.3 Move

AE32000에서 지원되는 레지스터간의 데이터 이동 명령들은 표. 3.2에 나열되어 있다.

Table 3.2: Move Instructions

Class	Details	Mnemonic
Move	Move	MOV %s, %d
	Move with add	LEA (%Rs/%SP, imm), %Rd/%SP
	Move from GPR to MH	MTMH %Rs
	Move from GPR to ML	MTML %Rs
	Move from MH to GPR	MFMH %Rd
	Move from ML to GPR	MFML %Rd
	Move from GPR to MRE	MTMRE %Rs
	Move from MRE to GPR	MFMRE %Rd
	Move from GPR to CR0	MTCR0 %Rs
	Move from GPR to CR1	MTCR1 %Rs
	Move from CR0 to GPR	MFCR0 %Rd
	Move from CR1 to GPR	MFCR1 %Rd

- 범용 레지스터간의 데이터 이동은 MOV 명령을 사용한다.
- 범용 레지스터와 스택 포이터간의 데이터 이동은 LEA 명령을 사용하며, 이동시에 즉치값을 같이 더하여 이동할 수 있다.
- 이때 즉치값은 32비트 singed number 이므로, 실제적으로 가/감산이 가능하다.
- 범용 레지스터와 특수 목적 레지스터간의 데이터 이동은 MTxx, MFxx 명령을 사용하며, 즉치값이 더해질 수 없다.
- 단, 음영 부분의 명령어는 AE32000C에서만 지원한다.

3.4 Branch

분기 명령은 프로그램의 흐름을 제어하는 동작을 수행하는 명령으로서, 조건 분기와 무조건 분기로 나뉜다. AE32000 프로세서에서는 코드 밀도를 향상시키기 위하여 다양한 형태의 조건 분기를 지원한다. 분기 명령들은 표. 3.3에 나열되어 있다.

Table 3.3: Branch Instructions

Class	Details	Mnemonic
Branch	Jump	JMP label
	Jump and Link	JAL label
	Jump on overflow clear	JNV label
	Jump on overflow set	JV label
	Jump on sign clear (positive or zero)	JP label
	Jump on sign set (negative)	JN label
	Jump on non-zero (not equal)	JNZ label
	Jump on zero (equal)	JZ label
	Jump carry clear (unsigned higher or equal)	JNC label
	Jump carry set (unsigned lower)	JC label
	Jump signed greater	JGT label
	Jump signed less	JLT label
	Jump signed greater or equal	JGE label
	Jump signed less or equal	JLE label
	Jump unsigned higher	JHI label
	Jump unsigned lower or equal	JLS label
	Jump register indirect	JR %Rs
	Jump register indirect and Link	JALR %Rs
	Jump to LR(Link Register)	JPLR

- 분기의 목적 주소는 다음 두 가지 형태로 지정할 수 있다.
  - PC-relative : 현재 PC값을 기준으로 offset 만큼 분기하는 방식으로 가장 일반적으로 사용되는 형태이다. offset의 값은 32bit singend값이 될 수 있어 가/감산이 가능하다.
  - Register indirect : 범용 레지스터의 값을 이용해 분기하는 방식으로서, dynamic linked library 등의 함수를 연결할 때 사용된다. AE32000 프로세서의

구현에서는 PC-relative 방식보다 더 많은 분기 패널티가 존재한다.

- JAL, JALR 명령은 분기의 시점에서 현재 PC값을 LR(Link Register)에 저장하며, JPLR 명령을 통해 복귀할 수 있다.

3.5 Arithmetic & Logical

AE32000 프로세서에서는 기본적인 산술 연산 이외에 DSP에서 사용할 수 있는 다양한 연산 기능(32비트 배럴 쉬프트, MAC연산, leading 0/1 count)을 제공한다.

Table 3.4: Arithmetic & Logical Instructions

Class	Details	Mnemonic	flag
Arithmetic	Add	ADD %Rs/imm, %Rd	C Z S V
	Add with short immediate	ADDQ imm5, %Rd	C Z S V
	Add with carry	ADC %Rs/imm, %Rd	C Z S V
	Substract	SUB %Rs/imm, %Rd	C Z S V
	Substract with carry	SBC %Rs/imm, %Rd	C Z S V
	Negate	NEG %Rd	
Logical	AND	AND %Rs/imm, %Rd	Z S
	OR	OR %Rs/imm, %Rd	Z S
	XOR	XOR %Rs/imm, %Rd	Z S
	NOT	NOT %Rd	Z S
	Test	TST %Rs1/imm, %Rs2	Z S
Compare	Compare	CMP %Rs1/imm, %Rs2	C Z S V
	Compare with short immediate	CMPQ imm5, %Rs	C Z S V
Shift	Arithmetic shift right	ASR %Rc/imm5, %Rd	C Z S
	Logical shift right	LSR %Rc/imm5, %Rd	C Z S
	Arithmetic shift left	ASL %Rc/imm5, %Rd	C Z S
	Set shift left	SSL %Rc/imm5, %Rd	C Z S
Multiply	Multiply	MUL %Rs/imm, %Rd	
	Multiply unsigned	MULU %Rs/imm, %Rd	
	Multiply and accumulate	MAC %Rs/imm, %Rs2	
Misc	Extension from byte to word	EXTB %Rd	Z S
	Extension from short to word	EXTS %Rd	Z S
	Convert from word to byte	CVB %Rd	Z S
	Convert from word to short	CVS %Rd	Z S
	Count leading zero	CNT0 %Rs	Z
	Count leading one	CNT1 %Rs	Z

- 각 연산의 결과에 의해 변경되는 flag는 표. 3.4에 표시되어 있다.
- 대부분의 연산 기능의 경우 %Rs 대신 즉치값을 이용하는 것이 가능하며, 이때 즉치값은 32bit singed number가 된다.
- MUL, MULU 연산의 %Rd는 짝수 범용 레지스터(R0, R2, R4,...)만 가능하다.

3.6 Coprocessor Access

AE32000 프로세서는 시스템 보조 프로세서 이외에 3개 까지의 보조 프로세서를 연결하는 것이 가능하다. 보조 프로세서를 접근하기 위한 명령들은 표. 3.5에 나열되어 있다.

Table 3.5: Coprocessor Access Instructions

Class	Details	Mnemonic	flag
Coprocessor	Instruction	CPCMD coproc_command	
	Move from Coproc. to GPR	MVFC %Rd@CP	
	Move from GPR to Coproc.	MVTC %Rd@CP	
	Check status bit in Coproc.	GETC imm4	Z
	Load from memory to Coproc.	LDC CPNO, (%Ri, imm), %Rd@CP	
	Store from Coproc. to memory	STC CPNO, %Rs@CP, (%Ri, imm)	
	Exception on Coproc. status	EXEC imm4	Z

- MVTC, MVFC 명령은 범용 레지스터 0번을 이용한 데이터 전달만 가능하다.
- LDC, STC 명령은 시스템 보조 프로세서와 같이 사용될 수 없다.
- LDC, STC 명령의 인덱스 레지스터는 스택 포인터 또는 범용 레지스터 [2] 1번이 이용된다.
- EXEC 명령은 보조 프로세서의 상태 비트를 이용하여 Coprocessor Interrupt를 발생시킨다.

[2] 스택 포인터를 이용한 이유는 스택 기반의 주소 indexing이 빈번하기 때문이며, GPR 1 번의 경우는 명령어 인코딩 용량 관계로 임의로 지정하여 사용하였다.

3.7 DSP Acceleration

AE32000C 프로세서는 신호처리 응용 분야에서의 성능을 향상시키기 위한 DSP 명령어를 지원한다. 아래에 소개될 DSP 명령어는 DSP 확장 옵션을 사용한 경우 사용 가능하다. 이 옵션에 대한 자세한 사항은 AE32000C-Lucida Processor User Guide를 참조하면 된다.

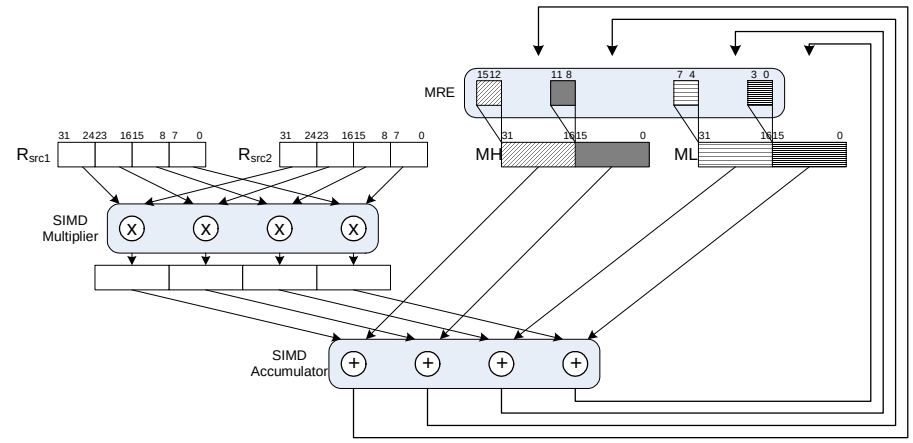
3.7.1 Multiply and Accumulation

Multiply and Accumulation(MAC) 연산은 곱셈의 결과를 누적하여 더해 나가는 연산으로, 신호처리 응용 분야에서 널리 사용되는 연산이다. AE32000C 프로세서에서 지원되는 MAC 연산은 표. 3.6에 나열되어 있다.

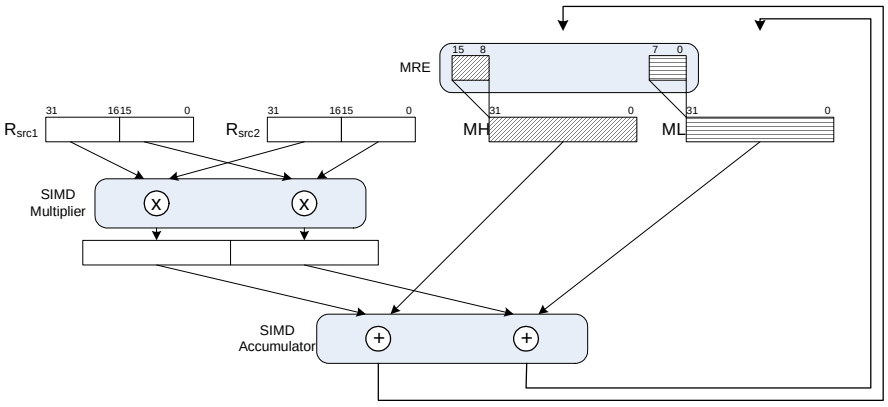
Table 3.6: Multiply and Accumulation Instructions

Class	Details	Mnemonic
MAC	Word	MAC (%Rs1/imm), %Rs2
	Short SIMD	MACS (%Rs1/imm), %Rs2
	Byte SIMD	MACB (%Rs1/imm), %Rs2
	Multiple Sum of Product (Short)	MSOPS (%Rs1/imm), %Rs2
	Multiple Sum of Product (Byte)	MSOPB (%Rs1/imm), %Rs2

- MAC Word 연산은 3.5절에서 소개된 MAC 연산과 마이크로 아키텍쳐적으로 차이를 보이지만 사용자 입장에서는 동일하게 사용 가능하다.
- SIMD MAC 연산에는 Accumulation 과정에서 정확성을 잃지 않기 위하여 Multiply Result Extend(MRE) 레지스터가 추가적으로 사용된다.
- SIMD MAC 연산의 동작은 그림.3.2에 나타나 있다.
- MSOPB, MSOPS 연산은 SIMD 형식으로 연산된 결과들을 하나로 합치는 역할을 수행하며, 각각 48비트(MH[15:0],ML[31:0]), 32비트(ML[31:0])의 결과를 가진다.



(a) Byte SIMD

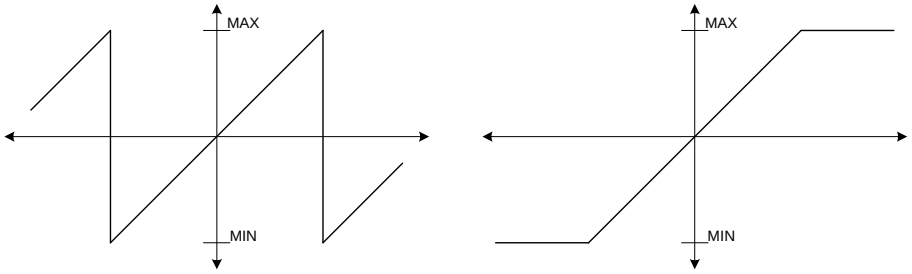


(b) Short SIMD

Figure 3.2: SIMD MAC 연산의 동작

3.7.2 Saturate Arithmetic

Saturate 연산은 그림. 3.3과 같이 일반적인 wrap-around 연산과 다르게 최대값/최소값을 초과한 경우 표현 가능한 최대/최소값을 유지하도록 되어 있는 연산이다. AE32000C 프로세서에서 지원되는 saturate 연산은 표. 3.7에 나열되어 있다.



(a) Wrap-around arithmetic (b) Saturate arithmetic

Figure 3.3: Saturate Arithmetic

Table 3.7: Saturate Instructions

Class	Details	Mnemonic
Saturate Add	Word	SADD (%Rs/imm), %Rd
	Short (signed)	SADDS (%Rs/imm), %Rd
	Short (unsigned)	SADUS (%Rs/imm), %Rd
	Byte (signed)	SADDB (%Rs/imm), %Rd
	Byte (unsigned)	SADUB (%Rs/imm), %Rd

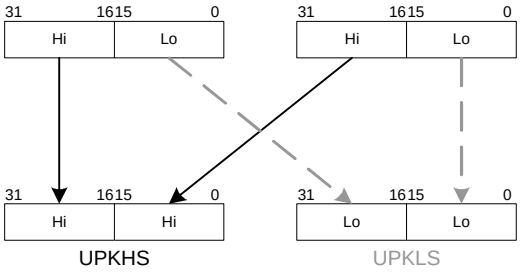
3.7.3 Unpack

Unpack은 SIMD 구조에서 필요한 연산의 형태를 정렬하기 위하여 사용되는 명령으로서, 필요에 따라 Word 전체에서 사용되는 필드들을 재정렬하여 이후에 사용되는 SIMD 연산의 효율을 높이기 위하여 사용된다. AE32000C 프로세서는 Short(16bit) 데이터 타입과 Byte(8bit) 데이터 타입에 대한 SIMD 연산을 지원하므로, 이 두가지 SIMD 형식에 대한 unpack을 지원한다. Unpack 연산은 표. 3.8에 나열되어 있으며, unpack 과정에서 사용되는 연산의 동작은 그림. 3.4에 나타내고 있다.

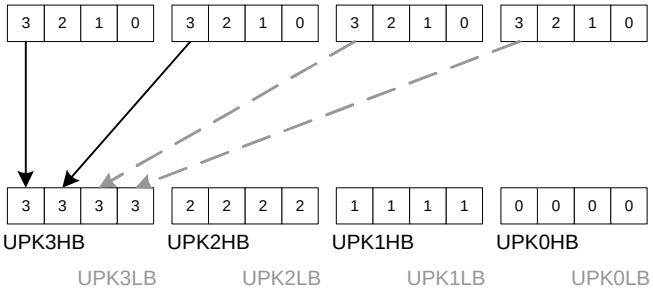
Table 3.8: Unpack Instructions

Class	Details	Mnemonic
Unpack	Short to high	UPKHS %Rsrc_grp, %Rd
	Short to low	UPKLS %Rsrc_grp, %Rd
	Byte from 0 to high	UPK0HB %Rsrc_grp, %Rd
	Byte from 0 to low	UPK0LB %Rsrc_grp, %Rd
	Byte from 1 to high	UPK1HB %Rsrc_grp, %Rd
	Byte from 1 to low	UPK1LB %Rsrc_grp, %Rd
	Byte from 2 to high	UPK2HB %Rsrc_grp, %Rd
	Byte from 2 to low	UPK2LB %Rsrc_grp, %Rd
	Byte from 3 to high	UPK3HB %Rsrc_grp, %Rd
	Byte from 3 to low	UPK3LB %Rsrc_grp, %Rd

- %Rsrc_grp은 짝수 범용 레지스터(R0, R2, R4,...)만 지정할 수 있으며, 연속된 두개의 레지스터가 사용된다.(R0-R1, R2-R3, ...)
- Byte unpack 명령은 목적 레지스터의 상위 16비트 또는 하위 16비트를 채우는 명령이므로, 하나의 Word를 구성하기 위해서는 2개의 unpack 명령이 사용되어야 한다.



(a) Unpack (Short data type)



(b) Unpack (Byte data type)

Figure 3.4: Unpack 연산의 동작

3.7.4 Miscellianeous

AE32000C 프로세서는 위에 기술된 DSP 연산 이외에 다수의 DSP 명령을 내장하고 있으며, 표. 3.9에 나열되어 있다.

Table 3.9: DSP 기타 Instructions

Class	Details	Mnemonic
Average	Average SIMD Short Average SIMD Byte	AVGS (%Rs/imm), %Rd AVGB (%Rs/imm), %Rd
Rotate	Rotate Left Rotate Right	ROL imm, %Rd ROR imm, %Rd
MIN/MAX	Minimum Maximum	MIN (%Rs/imm), %Rd MAX (%Rs/imm), %Rd
Absolute Value	Absolute Value	ABS %Rd
Radix Point adjust	Fixed Point Multiply Result Shift	MRS imm, %Rd

- AVGS, AVGB, MIN, MAX 명령은 목적 레지스터를 0 - 7번의 범용 레지스터만 사용 가능하다.

3.8 JAVA Acceleration

AE32000C 프로세서는 JAVA 응용 프로그램의 성능을 향상시키기 위해 JAVA 하드웨어 가속기를 탑재하였으며 이를 지원하기 위해 JAVA 확장 명령어를 지원한다. 아래에 소개될 JAVA 명령어들은 JAVA 확장 옵션을 사용한 경우 사용이 가능하다. 이 옵션에 대한 자세한 사항은 AE32000C-Empress Processor User Guide를 참조하면 된다.

Table 3.10: JAVA Instructions

Class	Details	Mnemonic	flag
JAVA	Mode Exchange EISC/JAVA	EXJ	J

- EISC 모드와 JAVA 모드간의 전환 시에 사용되며, 스택 캐시와 지역 변수 캐시의 초기화 명령어 시퀀스를 수반한다. (4.20절 참조)
- 현재까지는 하나의 명령어가 추가되었으나 더 많은 명령어들이 추가될 예정이다.

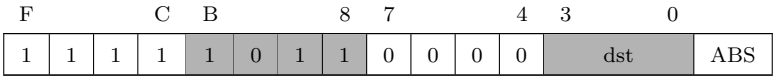
Chapter 4

AE32000 Instructions

AE32000는 코드 밀도를 높이기 위하여 다양한 명령어를 갖추고 있다. 본 장에서는 AE32000 명령어들의 구체적인 동작 및 사용시 주의 사항에 대하여 기술한다.

4.1 ABS - absolute value

ABS 명령은 Signed 입력에 대한 절대값을 출력한다.



Syntax

```
ABS   %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 변화 없음

C	Z	S	V	E
–	–	–	–	–

Operation

```
%Rdst = |%Rdst|
```

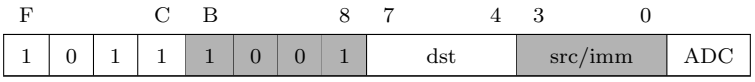
Usage ABS연산은 Signed 입력에 대한 절대값이 필요한 경우에 사용되며 다음과 같이 사용한다.

```
1  ABS %R0
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C-DSP version에서 지원된다.

4.2 ADC - addition with carry

ADC 명령은 Destination Register값과 Source Register값 혹은 즉치 값과 carry_flag를 같이 더하는 명령이다.



Syntax

```
ADC    %Rsrc, %Rdst
ADC    <imm>, %Rdst
```

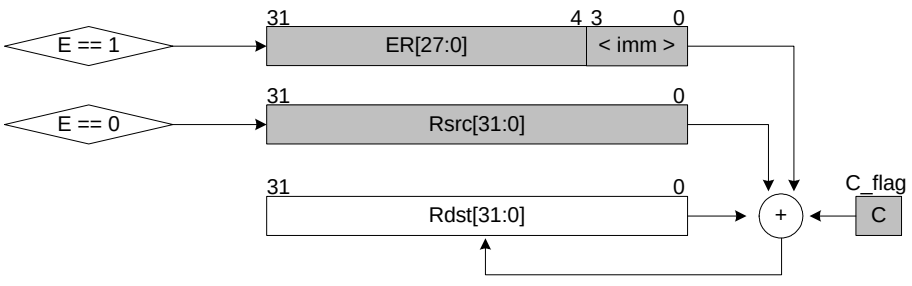
Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 명령어 수행 결과에 따라 C, Z, S, V-flag이 변화하며, E-Flag은 항상 clear된다. C-Flag 값은 연산 결과의 변경된 Carry 값이 그대로 반영된다.

C	Z	S	V	E
*	*	*	*	0

Operation

```
if(E_flag)
     %Rdst = %Rdst + (ER <<4 + imm) + C_flag
else
     %Rdst = %Rdst + %Rsrc + C_flag
```



Usage ADC 명령은 한 워드 이상의 덧셈을 수행하기 위하여 사용된다.
R1:R0에 64비트 값과 R3:R2에 64비트 값이 있다고 할 때, 두 64비트의 덧셈은 다음과 같이 사용된다.

```
1  ADD %R2, %R0
2  ADC %R3, %R1
```

이러한 동작은 R0:R1 = R3:R2 + R1:R0와 같은 동작을 수행하게 된다. 만일 ADC에서 피연산자로서 즉치값을 사용하고자 할 때는 반드시 LERI를 이용해야 한다. 단, 일반적인 assembly programming시에는 `adc <imm>, %Rdst`와 같이 사용하는 경우 assembler가 자동적으로 LERI를 추가한다.

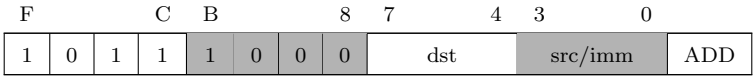
```
1  LERI 0x0
2  ADC 0x1, %R0
```

위의 동작은 즉치 값을 1로 취하기 위하여 LERI를 사용하는 예를 보여준다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.3 ADD - addition

ADD 명령은 Destination Register값과 Source Register값 혹은 즉치 값을 더하는 명령이다.

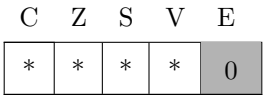


Syntax

```
ADD   %Rsrc, %Rdst
ADD   <imm>, %Rdst
```

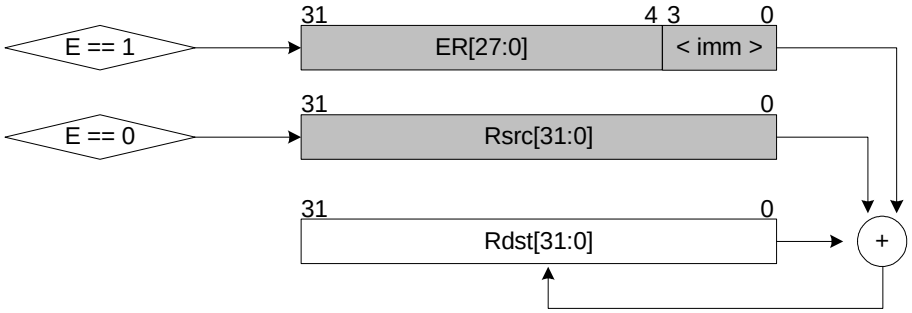
Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 명령어 수행 결과에 따라 C, Z, S, V-flag이 변화하며, E-Flag은 항상 clear된다. C-Flag 값은 연산 결과의 변경된 Carry 값이 그대로 반영된다.



Operation

```
if(E_flag)
     %Rdst = %Rdst + (ER << 4 + imm)
else
     %Rdst = %Rdst + %Rsrc
```



Usage ADD연산은 두 피연산자간의 덧셈을 수행하기 위하여 사용된다.

```
1  ADD %R2, %R0
```

사용 가능한 피연산자로는 레지스터 혹은 즉치값이 가능하다. 만일 ADD에서 피연산자로서 즉치값을 사용하고자 할 때는 어떠한 경우에도 LERI를 이용해야 한다.

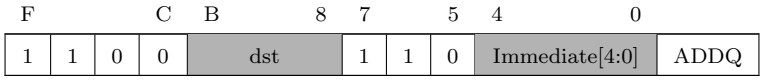
```
1  LERI 0x0
2  ADD 0x1, %R0
```

위의 동작은 즉치 값을 1로 취하기 위하여 LERI를 사용하는 예를 보여준다. 그러나, 위의 예에서 볼 수 있듯이 즉치값을 사용할 경우 LERI명령이 하나 이상 선행해야 한다는 단점을 지닌다. 이를 방지하기 위하여 짧은 즉치값(-16 ~ 15)을 사용하는 경우 ADDQ를 이용할 것을 강력히 권장한다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.4 ADDQ - short immediate addition

ADDQ 명령은 Destination Register값과 즉치 값을 더하는 명령이다.

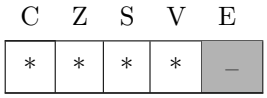


Syntax

```
ADDQ   <imm>,  %Rdst
```

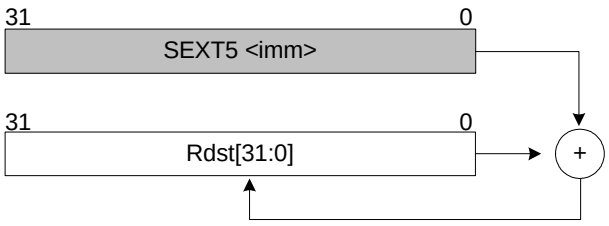
Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 명령어 수행 결과에 따라 C, Z, S, V-flag이 변화한다. C-Flag 값은 연산 결과의 변경된 Carry 값이 그대로 반영된다.



Operation

```
%Rdst = %Rdst + SEXT5<imm>
```



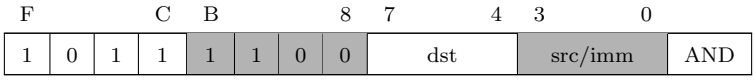
Usage ADD연산은 즉치값을 사용하고자 할 때는 어떠한 경우에도 LERI를 이용해야 한다. 높은 코드 밀도를 위해 짧은 즉치값을 사용하는 경우, LERI 명령어를 사용하지 않고 즉치값을 사용하기위해 ADDQ를 사용한다. (LERI가 선행되지 않는다.)

```
ADDQ 0x1, %R0
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.5 AND - bitwise AND

AND 명령은 Destination Register값과 Source Register값 혹은 즉치 값을 bitwise AND연산을 취하는 명령이다.



Syntax

```
AND   %Rsrc,  %Rdst
AND   <imm>,  %Rdst
```

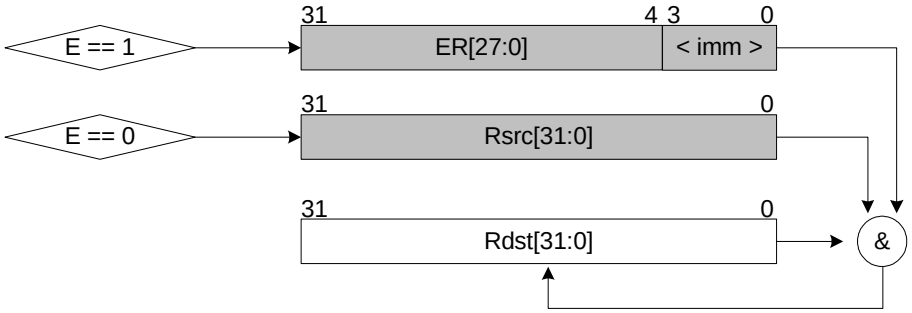
Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 명령어 수행 결과에 따라 Z, S-flag이 변화하며, E-Flag은 항상 clear된다.

C	Z	S	V	E
–	*	*	–	0

Operation

```
if(E_flag)
     %Rdst = %Rdst & (ER << 4 + imm)
else
     %Rdst = %Rdst & %Rsrc
```



Usage AND연산은 두 피연산자간에 bitwise AND연산을 수행하기 위하여 사용된다.

```
AND %R0, %R2
```

즉치값을 사용하기 위해서는 반드시 LERI를 이용해야 한다.

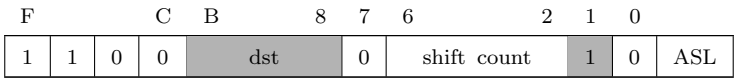
```
LERI 0x0
AND 0x1, %R2
...
LERI 0x3fff
AND 0xe, %R2 // %R2 = %R2 & 0xfffffffe
...
```

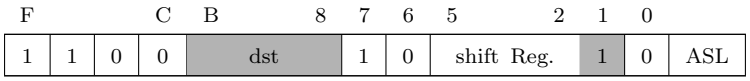
LERI는 sign extension을 통하여 ER을 만들기 때문에 위의 예는 하나의 LERI를 통하여 0xfffffffe라는 마스크를 생성하고, 이를 이용하여 %R2를 마스킹하는 예를 보여준다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.6 ASL - Arithmetic Shift Left

ASL 명령은 Destination Register값을 shift count만큼 좌측으로 쉬프트하는 명령이다. 쉬프트 될 때 LSB값은 '0'이 채워진다. 쉬프트 카운트로는 즉치값과 레지스터의 값이 가능하다. 레지스터 값을 이용할 경우에는 하위 5비트만 사용된다.



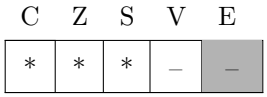


Syntax

```
ASL   <shift_count>,  %Rdst
ASL   %Rsrc,  %Rdst
```

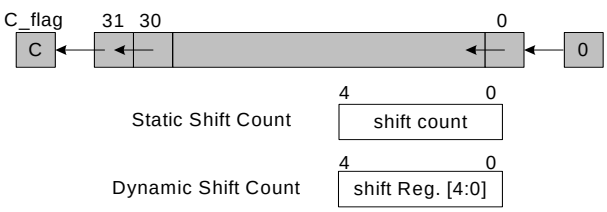
Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 명령어 수행 결과에 따라 C, Z, S-flag이 변화한다. C-Flag 값은 shifted-out된 Carry 값이 반영된다. shift되는 크기가 '0'인 경우 C-Flag는 '0'을 갖는다.



Operation

```
Static Shift
     %Rdst = %Rdst << <shift_count>
Dynamic Shift
     %Rdst = %Rdst << (%Rsft & 0x1f)
```



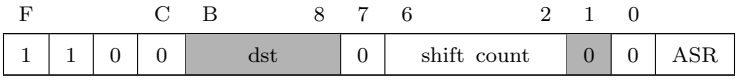
Usage ASL 연산은 signed/unsigned number를 left shift할 때 사용된다.
Arithmetic Shift Right 의 경우 MSB(sign bit)의 값을 그대로 유지하게 된다. 그러나 Arithmetic Shift Left 의 경우는 sign bit 를 무시하고 shift 한다.

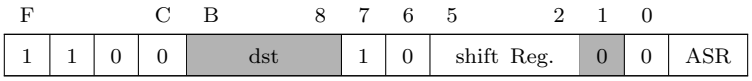
```
1  ASL 0x1, %R2
2  ASL %R0, %R2
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.7 ASR - Arithmetic Shift Right

ASR 명령은 Destination Register값을 shift count만큼 우측으로 쉬프트하는 명령이다. Arithmetic shift이므로 상위에는 sign값이 채워진다. 쉬프트 카운트로는 즉치값과 레지스터의 값이 가능하다. 레지스터 값을 이용할 경우에는 하위 5비트만 사용된다.



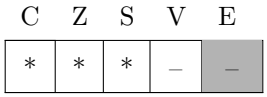


Syntax

```
ASR   <shift_count>,  %Rdst
ASR   %Rsrc,  %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 명령어 수행 결과에 따라 C, Z, S-flag이 변화한다. C-Flag 값은 shifted-out된 Carry 값이 반영된다. shift되는 크기가 '0'인 경우 C-Flag는 '0'을 갖는다.

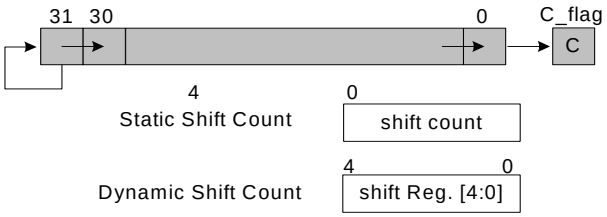


Operation

```
Static Shift
     %Rdst = %Rdst >> <shift_count>
Dynamic Shift
     %Rdst = %Rdst >> (%Rsft & 0x1f)
```

쉬프트되면서 Sign은 유지된다. (상위 비트들은 %Rdst[31]로 채워진다.)

Usage ASR 연산은 signed number를 right shift할 때 사용된다.




```
1  ASR 0x1, %R2
2  ASR %R0, %R2
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.8 AVGB - SIMD average (SIMD_BYTE)

AVGB 명령은 SIMD로 각 부분의 평균을 구한다.

F			C	B			8	7	5	4	3	0	
1	1	1	0	0	0	1	0	dst		0	src/imm		AVGB

Syntax

```
AVGB  %Rsrc, %Rdst
AVGB  <imm>, %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification E-Flag를 항상 clear한다.

C	Z	S	V	E
–	–	–	–	0

Operation

```
if(E_flag)
     OP = ((ER << 4) + imm)
else
     OP = %Rsrc
%Rdst[7:0] = (OP[7:0] + %Rdst[7:0]) >> 1
%Rdst[15:8] = (OP[15:8] + %Rdst[15:8]) >> 1
%Rdst[23:16] = (OP[23:16] + %Rdst[23:16]) >> 1
%Rdst[31:24] = (OP[31:24] + %Rdst[31:24]) >> 1
```

Usage AVGB 연산은 각 8 비트 부분의 평균을 구하는 연산으로써 0 - 7번 레지스터만이 목적 레지스터로 사용 가능하다. 또한 사용 가능한 피연산자로는 레지스터 혹은 즉치값이 가능하며. 피연산자로서 즉치값을 사용하고자 할 때는 어떠한 경우에도 LERI를 이용해야 한다.

```
1  AVGB %R3, %R2
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C-DSP version에서 지원된다.

4.9 AVGS - SIMD average (SIMD_SHORT)

AVGS 명령은 SIMD로 각 부분의 평균을 구한다.

F			C	B			8	7	5	4	3	0
1	1	1	0	0	0	1	0	dst		1	src/imm	AVGS

Syntax

```
AVGS %Rsrc, %Rdst
AVGS <imm>, %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification E-Flag를 항상 clear한다.

C	Z	S	V	E
–	–	–	–	0

Operation

```
if(E_flag)
     OP = ((ER << 4) + imm)
else
     OP = %Rsrc
%Rdst[15:0] = (OP[15:0] + %Rdst[15:0]) >> 1
%Rdst[31:16] = (OP[31:16] + %Rdst[31:16]) >> 1
```

Usage AVGS 연산은 각 16 비트 부분의 평균을 구하는 연산으로써 0 - 7번 레지스터만이 목적 레지스터로 사용 가능하다. 또한 사용 가능한 피연산자로는 레지스터 혹은 즉치값이 가능하며. 피연산자로서 즉치값을 사용하고자 할 때는 어떠한 경우에도 LERI를 이용해야 한다.

```
1  AVGS %R3, %R2
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C-DSP version에서 지원된다.

4.10 BRKPT - instruction breakpoint

BRKPT 명령은 instruction breakpoint를 지정하기 위하여 사용되는 명령으로서, OSI exception을 발생시킨다. OSI exception은 프로세서 모드를 디버깅을 위한 OSI mode로 전환한다.

F			C	B			8	7			4	3			0	
1	1	1	0	0	0	0	0	1	1	1	1					BRKPT

Syntax

```
BRKPT
```

Exceptions – Prefetch 취소

Status Modification OSI 모드로의 bits과 L-flag이 변화하며, E-Flag은 항상 clear된다.

Proc_Mode[1]	Proc_Mode[0]	L	E
0	1	0	0

Operation

1. special cycle을 수행하여 외부 로직에 OSI 인터럽트로 진입함을 알림
   (Special cycle no = 2)
2. SR을 ISP를 이용하여 저장
3. SR 값 변경
4. PC - 2 저장
5. 외부 입력인 OSIROM 값을 이용하여 인터럽트 벡터 테이블 접근
6. 인터럽트 핸들러에 존재하는 OSI 루틴 수행

Usage 이 명령은 instruction breakpoint를 제공하기 위한 목적으로 사용되며, 소프트웨어 디버거에서 OSI유닛을 이용하지 않고, 해당 명령을 BRKPT로 대치함으로서 break를 거는 방식을 이용하고자 할 경우에 사용된다.
일반적으로 이러한 용도로 사용되는 SWI대신 BRKPT를 이용하는 경우 스택을 따로 할당받을 수 있으므로, OSI mode에서의 동작과 관계 없이 supervisor mode의 스택의 내용을 쉽게 찾아낼 수 있다는 장점이 있다.

```
0x030 ADD %R1, %R2
0x032 BRKPT  // 원래 명령어 대신 BRKPT대치
.. interrupt 처리.. 복귀..
0x032 ADD %R2, %R4 // 원래 명령어 복구
```

위의 예는 0x032번지에 대하여 BRKPT 명령을 이용하여 breakpoint를 지정하는 예를 보여주고 있다.

Note

- System Coprocessor내의 OSI 로직을 이용하여 breakpoint를 수행하며, 독립적으로 BRKPT 명령을 이용하는 것이 가능하다. 즉, OSI 로직은 BRKPT 명령에 대하여 아무런 영향을 주지 않는다.
- BRKPT의 경우 인터럽트 처리 후 BRKPT가 존재하던 주소(PC-2)의 값으로 복귀하므로, handler는 처리 과정에서 BRKPT가 존재하던 위치의 명령을 다시 복구시킬 수 있어야 한다.
- BRKPT 명령을 이용하는 방법은 프로그램의 변경이 요구되므로, 프로그램이 ROM으로 실장되는 경우에는 사용할 수 없다. 따라서, 초기 디버깅 과정에서만 이용 가능하며, 실장을 위해서는 OSI 유닛을 이용해야 한다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.11 CLR - clear a bit of status register

CLR 명령은상태 레지스터의 상태 비트 중 한 비트를 clear한다. 지정 가능한 범위는 비트 번호 15~0까지로 제한되며, 이중 사용자 모드에서는 7번째 비트 이하(7~0)까지만 clear 시키는 것이 가능하다.

F			C	B			8	7			4	3	0	
1	1	1	0	0	0	0	1	0	0	1	1	Flag Pos.		CLR

Syntax

```
CLR <bit_pos>
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification Supervisor Mode 해당 비트 clear

User Mode 해당 비트가 8이하인 경우, 해당 비트 clear

Operation

```
SR[bit_pos] = 0
```

상태 레지스터내 특정 비트의 값을 clear한다.

상태 레지스터의 F~8 비트는 user mode에서는 접근 불가능하므로, 사용자 모드에서 위의 비트에 접근한 경우 NOP와 동일하게 처리된다.

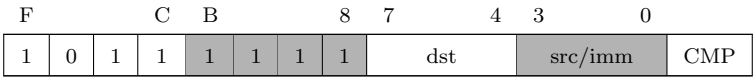
Usage 아래 예는 7번 비트(carry flag) 을 clear하는 명령을 보여준다.

```
1 CLR 0x7
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.12 CMP - compare

CMP 명령은 두 레지스터간의 비교를 위하여 사용되며, 목적 레지스터로부터 소스 레지스터 값을 빼는 동작을 수행한다. 결과는 저장되지 않으며, flag의 변경만 수행된다.

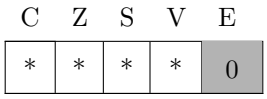


Syntax

```
CMP   %Rsrc,  %Rdst
CMP   <imm>,  %Rdst
```

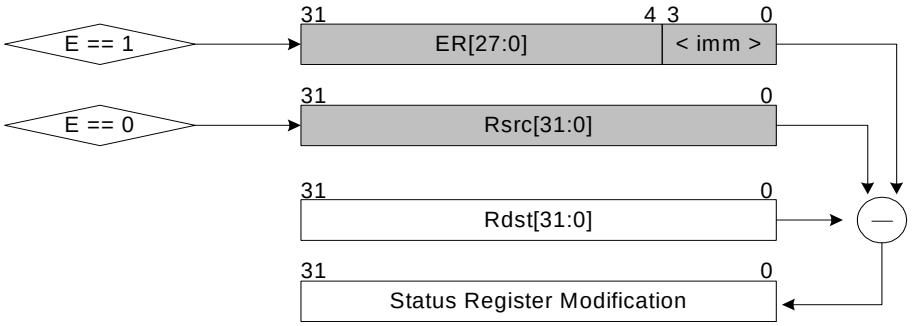
Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 명령어 수행 결과에 따라 C, Z, S, V-flag이 변화하며, E-Flag은 항상 clear된다. 이 명령어에 의해 변경되는 Flag의 성질은 subtract 연산의 결과와 동일하며, 단지 결과 값이 범용 레지스터에 저장되지 않는 점만이 차이가 있다.



Operation

```
if(E_flag)
     %Rdst - (ER << 4 + imm)
else
     %Rdst - %Rsrc
```



Usage CMP연산은 두 피연산자간의 비교를 수행하기 위하여 사용된다.

```
1  CMP %R2, %R0;
```

사용 가능한 피연산자로는 레지스터 혹은 즉치값이 가능하다. 만일 CMP에서 피연산자로서 즉치값을 사용하고자 할 때는 어떠한 경우에도 LERI를 이용해야 한다.

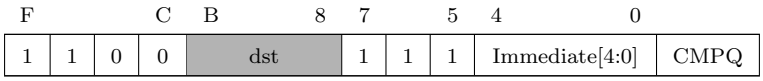
```
1  LERI 0x0;
2  CMP 0x1, %R0;
```

위의 동작은 즉치 값을 1로 취하기 위하여 LERI를 사용하는 예를 보여준다. 그러나, 위의 예에서 볼 수 있듯이 즉치값을 사용할 경우 LERI명령이하나 이상 선행해야 한다는 단점을 지닌다. 이를 방지하기 위하여 짧은 즉치값 (-16 ~ 15)을 사용하는 경우 CMPQ를 이용할 것을 강력히 권장한다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.13 CMPQ - short immediate compare

CMPQ 명령은 작은 상수와 레지스터간의 비교를 위하여 사용되며, 목적 레지스터로부터 상수 값을 빼는 동작을 수행한다. 결과는 저장되지 않으며, flag의 변경만 수행된다.

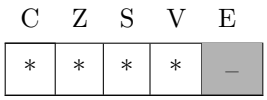


Syntax

```
CMP   <imm>, %Rdst
```

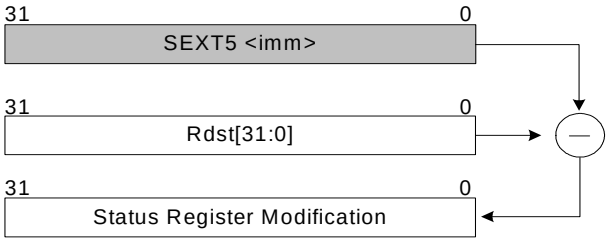
Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 명령어 수행 결과에 따라 C, Z, S, V-flag이 변화한다. 이 명령어에 의해 변경되는 Flag의 성질은 subtract 연산의 결과와 동일하며, 단지 결과 값이 범용 레지스터에 저장되지 않는 점만이 차이가 있다.



Operation

```
%Rdst - SEXT5<imm>
```



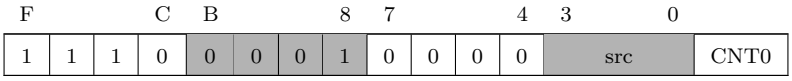
Usage CMPQ연산은 두 피연산자간의 비교를 수행하기 위하여 사용되며, 피연산자는 즉치값이 사용된다. LERI가 선행되지 않는다.

```
1  CMPQ 0x01, %R0
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.14 CNT0 - count leading zeros

CNT0 명령은 소스 레지스터의 leading zero의 개수를 세어서 %R0에 저장한다.



F			C	B			8	7			4	3	0	
1	1	1	0	0	0	0	1	0	0	0	0	src		CNT0

Syntax

```
CNT0 %Rsrc
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 명령어 수행 결과에 따라 Z-flag이 변화한다.

C	Z	S	V	E
–	*	–	–	–

Operation

```
if(%Rsrc == 0)
     %R0 = 32
else
     %R0 = 31 - (bit position of most significant '1' in %Rsrc)
```

Usage CNT0 연산은 단항 연산으로서 지정된 레지스터에서 leading zero의 수를 세어 항상 그 결과를 %R0에 저장한다는 점에 유의하여야 한다.

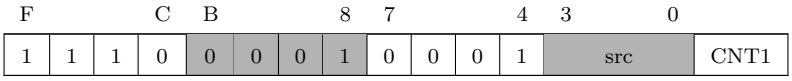
```
1  CNT0 %R1
```

Note 이 연산은 Huffman decoding에서 directory search, finite field inversion에서 extend euclidan algorithms 등에서 유용하게 사용될 수 있다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.15 CNT1 - count leading ones

CNT1 명령은 소스 레지스터의 leading one의 개수를 세어서 %R0에 저장한다.



Syntax

```
CNT1 %Rsrc
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 명령어 수행 결과에 따라 Z-flag이 변화한다.

C	Z	S	V	E
–	*	–	–	–

Operation

```
if(%Rsrc == 0xffffffff)
     %R0 = 32
else
     %R0 = 31 - (bit position of most significant '0' in %Rsrc)
```

Usage CNT0 연산은 단항 연산으로서 지정된 레지스터에서 leading one의 수를 세어 항상 그 결과를 %R0에 저장한다는 점에 유의하여야 한다.

```
1  CNT1 %R1
```

Note 이 연산은 Huffman decoding에서 directory search, finite field inversion에서 extend euclidan algorithms 등에서 유용하게 사용될 수 있다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.16 CPCMD - coprocessor command

CPCMD 명령은 지정한 보조 프로세서로 명령을 전당한다.보조 프로세서로 보내는 명령의 길이는 보조 프로세서에서의 필요에 의하여 LERI를 이용하여 확장 가능하다.

F			C	B		9	8	7	6	5	0	
1	1	1	0	1	1	CP #		0	0	Command Bit[5:0]		CPCMDn

Syntax

```
CPCMD <cp_no>, <command>
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification E-Flag만을 항상 clear된다.

C	Z	S	V	E
–	–	–	–	0

Operation

```
if(E_flag)
     cp_cmd = (ER << 6 + <command>)
else
     cp_cmd = ZEXT(<command>)
```

Usage CPCMD명령은 보조 프로세서로 보낼 명령을 encapsulation하고 있는 명령으로서, 해당 명령과 대상 보조 프로세서를 지정하여 사용한다.

```
1  CPCMD 0x1, 0x4be40
```

위의 명령은 1번 보조 프로세서(일반적으로 fp/dsp coprocessor)로 0x4be40이라는 20비트 명령을 전달하는 것을 보여준다.

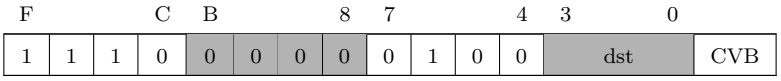
Note 보조 프로세서 명령어의 길이는 앞에서 설명한 바와 같이 필요에 따라 LERI를 이용하여 원하는 만큼 확장하는 것이 가능하다. 그러나, 마이크로 아키텍쳐의 복잡도를

고려할 때 48비트, 34비트, 32비트, 20비트, 16비트, 6비트 정도로 한정하는 것이 적당하다고 판단되며, 48비트 이상의 명령은 이용하지 않을 것을 강력히 권고한다. 또한, 일반적인 경우에서는 명령의 길이가 32비트를 넘어가는 것도 권장되지 않는다.
각 프로세서에서의 보조 프로세서 명령의 길이는 구현에 따라 한정되며, 이는 각 프로세서의 technical reference manual 을 참조하면 된다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.17 CVB - convert to byte

CVB 명령은 레지스터의 값에 0xff값의 마스크와 AND를 취하여 byte값으로 변경시킨다.

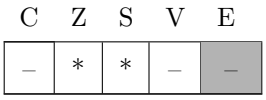


Syntax

```
CVB   %Rdst
```

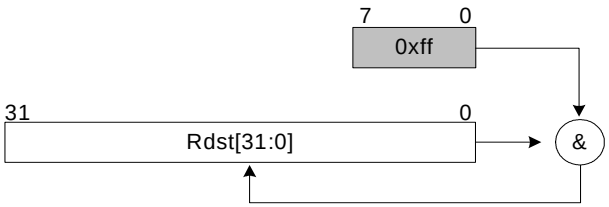
Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 명령어 수행 결과에 따라 Z, S-flag이 변화한다.



Operation

```
%Rdst = %Rdst & 0xff
```



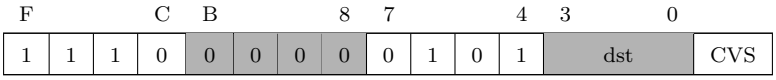
Usage CVB명령은 대상 레지스터의 하위 8bit만을 취하는 명령이다.

```
1  CVB %R0
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.18 CVS - convert to short

CVS 명령은 레지스터의 값에 0xffff값의 마스크와 AND를 취하여 short값으로 변경시킨다.

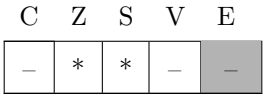


Syntax

```
CVS  %Rdst
```

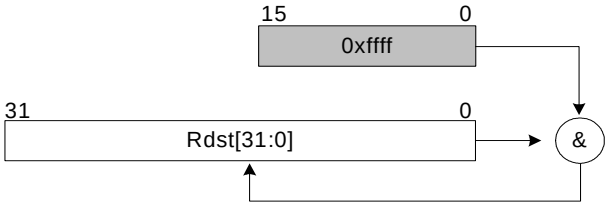
Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 명령어 수행 결과에 따라 Z, S-flag이 변화한다.



Operation

```
%Rdst = %Rdst & 0xffff
```



Usage CVS명령은 대상 레지스터의 하위 16bit만을 취하는 명령이다.

```
1  CVS %R0
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.19 EXEC - exception on coprocessor status

EXEC 명령은 대상 보조 프로세서에 존재하는 상태 레지스터의 한 비트를 살펴보고 보조 프로세서 인터럽트를 발생시킨다.

F			C	B		9	8	7			4	3	0	
1	1	1	0	1	1	CP #		0	1	0	1	Status Bit #		EXECn

Syntax

```
EXECn <cp_no>, <bit_pos>
```

Exceptions – 명령어 fetch취소, 명령어 큐 초기화

Status Modification Supervisor 모드로 전환

Proc_Mode[1]	Proc_Mode[0]	INT_en	L	Z	E
0	0	0	0	*	0

Operation

```
Z_flag = %R15[bit_no]@cp<cp_no>
if(Z_flag)
     makes_cpint
```

보조 프로세서 인터럽트 처리 과정

1. SR을 SSP를 이용하여 저장
2. SR 값 변경
3. PC 저장
4. 접근 CP값에 따른 인터럽트 벡터 테이블 접근
5. 인터럽트 핸들러에 존재하는 OSI 루틴 수행

Usage EXEC명령은 보조 프로세서 상태 레지스터로의 특정 비트의 값을 통하여 인터럽트를 발생시키는 동작을 한다.

```
EXEC 0x1, 0x4
```

이 인터럽트는 연산에 관련된 보조 프로세서의 경우 보조 프로세서의 연산을 구동시킨 상태에서 독립적으로 프로세서를 동작시키다가 결과를 polling하기 위하여 사용할 수 있다.

```
CPCMD 0x1, 0xfdfe0  // 보조 프로세서에 명령

ADD %R3, %R4        // 프로세서 동작
...
EXEC 0x1, 0x4       // 결과 polling
ADD %R2, %R2        // 프로세서 동작
...
EXEC 0x1, 0x4       // 결과 polling
```

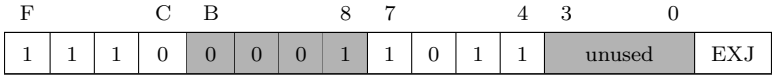
위와 같이 보조 프로세서의 결과를 polling하는 동작으로 사용 가능하다. 단, 보조 프로세서의 동작을 이용하여 인터럽트를 걸 필요가 없는 경우 GETC명령과 JNZ 명령, JAL의 조합을 통하여 분기를 만들 수 있다. 자세 한 것은 GETC에서 설명하도록 한다.

Note 보조 프로세서 명령어의 길이는 앞에서 설명한 바와 같이 필요에 따라 LERI를 이용하여 원하는 만큼 확장하는 것이 가능하다. 그러나, 마이크로 아키텍쳐의 복잡도를 고려할 때 48비트, 34비트, 32비트, 20비트, 16비트, 6비트 정도로 한정하는 것이 적당하다고 판단되며, 48비트 이상의 명령은 이용하지 않을 것을 강력히 권고한다. 또한, 일반적인 경우에서는 명령의 길이가 32비트를 넘어가는 것도 권장되지 않는다.
각 프로세서에서의 보조 프로세서 명령의 길이는 구현에 따라 한정되며, 이는 각 프로세서의 technical reference manual 을 참조하면 된다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.20 EXJ - exchange Java mode

EXJ 명령은 EISC 모드와 JAVA 모드간의 전환을 수행하는 명령어이다.



F			C	B			8	7			4	3	0	
1	1	1	0	0	0	0	1	1	0	1	1	unused		EXJ

Syntax

```
EXJ
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification J-Flag를 변경한다.

J	C	Z	S	V	E
*	–	–	–	–	–

Operation

- SR[17] == 0 인 경우, 현재 EISC 모드이므로
  1. 현재 모드의 스택에 PC를 push
  2. SR[17] = 1, JAVA 모드로 변환
  3. 모든 파이프라인 플러쉬
  4. GPR 15번을 이용하여 바이트코드 패치를 시작
- SR[17] == 1 인 경우, 현재 JAVA 모드이므로
  1. SR[17] = 0, EISC 모드로 변환
  2. pop PC 수행하여 EISC 명령어 패치를 시작

Usage EXJ 명령어는 EISC 모드와 JAVA 모드간의 전환 시에 사용되며, 스택 캐시와 지역 변수 캐시의 초기화 명령어 시퀀스를 수반한다.

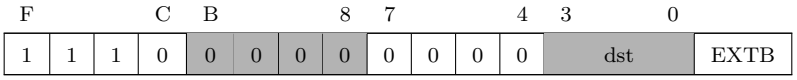
```
1  // Stack cache init
2  lea (%0, 0x0), %%r0"::"r"(stack[-2].intValue)
3  lea (%0, 0x0), %%r1"::"r"(stack[-1].intValue)
```

```
4
5   // Local Variable cache init
6   lea (%0, 0x0), %%r10"::"r"(var[0].intValue)
7   lea (%0, 0x0), %%r11"::"r"(var[1].intValue)
8   lea (%0, 0x0), %%r12"::"r"(var[2].intValue)
9   lea (%0, 0x0), %%r13"::"r"(var[3].intValue)
10
11  // Exchange JAVA mode
12  EXJ
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C, AE32000C-DSP version 중, JAVA 확장 옵션을 지원하는 프로세서에서 지원한다.

4.21 EXTB - extension from byte

EXTB 명령은 대상 레지스터의 내용 중 하위 8bit의 내용을 기반으로 부호 확장을 수행한다.

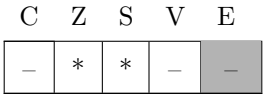


Syntax

```
EXTB %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 명령어 수행 결과에 따라 Z, S-flag이 변화한다.



Operation

```
%Rdst = SEXT8(%Rdst)
```

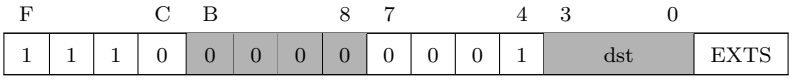
Usage EXTB명령은 대상 레지스터의 하위 8bit에 대한 부호 확장을 수행하는 명령이다.

```
1 EXTB %R0
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.22 EXTS - extension from short

EXTS 명령은 대상 레지스터의 내용 중 하위 16bit의 내용을 기반으로 부호 확장을 수행한다.

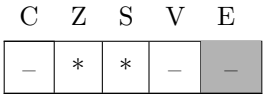


Syntax

```
EXTS   %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 명령어 수행 결과에 따라 Z, S-flag이 변화한다.



Operation

```
%Rdst = SEXT16(%Rdst)
```

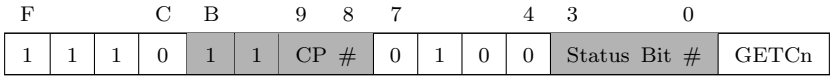
Usage EXTS명령은 대상 레지스터의 하위 16bit에 대한 부호 확장을 수행하는 명령이다.

```
1  EXTS %R0
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.23 GETC - get a status from coprocessor

GETC 명령은 대상 보조 프로세서에 존재하는 상태 레지스터의 한 비트를 가져와서 Z_flag의 값을 변경한다



F			C	B		9	8	7			4	3	0	
1	1	1	0	1	1	CP #		0	1	0	0	Status Bit #		GETCn

Syntax

```
GETC  <cp_no>, <bit_pos>
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 명령어 수행 결과에 따라 Z-flag이 변화한다.

C	Z	S	V	E
–	*	–	–	–

Operation GETC명령은 지정된 보조 프로세서로의 상태 레지스터로부터 해당 비트를 가지고 와서 Z_flag를 변경시킨다.

Usage GETC명령은 보조 프로세서 상태 레지스터로의 특정 비트의 값을 프로세서의 Z_flag에 반영시키는 동작을 한다.

```
1  GETC 0x1, 0x4
```

이 인터럽트는 연산에 관련된 보조 프로세서의 경우 보조 프로세서의 연산을 구동시킨 상태에서 독립적으로 프로세서를 동작시키다가 결과를 polling하기 위하여 사용할 수 있다.

```
1  CPCMD 0x1, 0xfdfe0 // 보조 프로세서에 명령
2  ADD %R3, %R4 // 프로세서 동작
3  ...
4  GETC 0x1, 0x4 // 결과 polling
5  JNZ TARGET
6  ADD %R2, %R2 // 프로세서 동작
```

```
...
GETC 0x1, 0x4 // 결과 polling
JNZ TARGET
```

위와 같은 코드는 보조 프로세서에 명령을 내린 후 보조 프로세서의 결과와 무관한 동작을 수행하면서 보조 프로세서 동작의 결과를 기다리는 것을 보여준다.
만일 결과를 기다려서 결과를 받은 경우 function call을하는 경우는 다음과 같이 한다.

```
CPCMD 0x1, 0xfdfe0 // 보조 프로세서에 명령
P1: ADD %R3, %R4 // 프로세서 동작
...
GETC 0x1, 0x4 // 결과 polling
JNZ P1
JAL TARGET
```

위의 코드는 보조 프로세서에 명령을 내린 후 관계없는 연산을 반복적으로 수행하다가 결과가 나온 이후에 특정 함수(TARGET)를 call하는 동작을 보여준다. 만일 보조 프로세서에 명령을 내린 후 특정 함수로 뛰어서 결과를 기다리고자 하면 다음과 같이 한다.

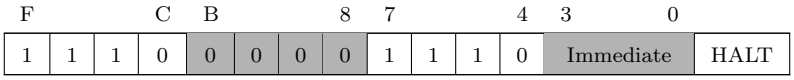
```
CPCMD 0x1, 0xfdfe0 // 보조 프로세서에 명령
JAL DL
...
DL: ADD %R3, %R4 // 프로세서 동작
...
GETC 0x1, 0x4 // 결과 polling
JNZ DL
JPLR
```

위의 예는 보조 프로세서로 명령을 내린 후 특정 함수(DL)로 진입하여 특정한 작업을 수행하다가 결과를 이용하여 원래 함수로 복귀하는 것이다. GETC와 JZ의 조합을 이용하는 경우 task switching에 따른 overhead가 인터럽트를 이용하는 EXEC명령에 비하여 매우 적어진다는 장점이 있다. 그러나, 레지스터의 사용에 대하여 주의를 기울여야 한다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.24 HALT - halt instruction to save power consumption

HALT 명령은 프로세서 혹은 외부 장치에 대한 동작을 정지 시킨다. 전력 소모를 줄이기 위하여 사용할 수 있다.



F			C	B			8	7			4	3	0	
1	1	1	0	0	0	0	0	1	1	1	0	Immediate		HALT

Syntax

```
HALT  <imm>
```

Exceptions – Prefetch 취소. **모든 처리가 중단**

Status Modification 변화 없음

Operation special cycle을 수행하여 외부 장치에 ¡imm¿값을 전달한다. ¡imm¿의 값이 4보다 작은 경우 core는 현재 상태를 유지하고 정지한다. ¡imm¿의 값이 4이상인 경우 프로세서는 정지하지 않는다. 정지된 프로세서는 static device이므로 클럭에 관계없이 현재의 상태를 유지하게 된다. 프로세서의 동작을 다시 시작하기 위해서는 interrupt가 요구된다.

Usage HALT명령은 프로세서를 정지시키는 명령으로서 전원 관리를 위하여 사용 가능하다.

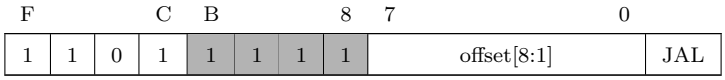
```
1  HALT 0x4   // 이 경우 프로세서는 정지하지 않는다.
```

Note HALT를 이용하여 전원 관리를 하려면 버스와 연결되어 있는 전원 관리 유닛이 있어야 한다. AE32000의 HALT 명령은 전원 관리 유닛에 imm값을 보냄으로서 명령을 내린다고 가정하고 만들어진 것이다. 외부 전원 관리 유닛은 imm의 값에 따라 주변 장치 혹은 코어 프로세 서로 공급되는 클럭을 조정하여 전원을 관리하도록 한다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.25 JAL - jump and link

JAL 명령은 무조건 분기를 수행하며, 현재 PC의 값은 Link Register에 저장하여, 추후에 복귀를 돕는다. 이 명령은 간단한 function call등에 이용할 수 있으며, leaf function의 경우 사용할 것을 권장한다.

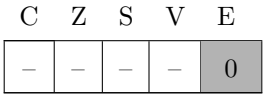


Syntax

```
JAL  <label>
JAL  <imm>
```

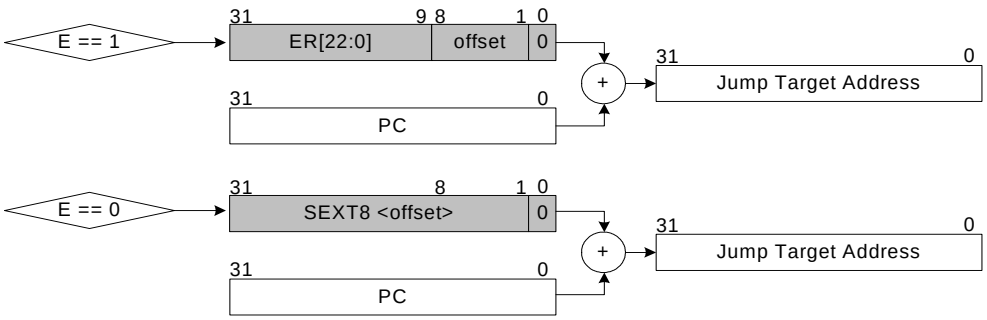
Exceptions – Prefetch 취소

Status Modification E-Flag를 항상 clear한다.



Operation

```
%LR = %PC
if(E_flag)
     %PC = %PC + (ER << 8 + offset) << 1
else
     %PC = %PC + SEXT8(offset) << 1
```



Usage JAL명령은 leaf function으로 jump할 때 사용하는 branch명령이다.

```
        JAL LEAF          // LEAF function으로 jump
        SUB %R2, %R4
        ...
LEAF :  ADD %R1, %R2       // LEAF function main
        ...
        JPLR              // JAL이후로 복귀한다.
```

위의 예는 가장 기본적인 JAL, JPLR의 사용을 보여주고 있다. AE32000는 Link Register(LR)를 이용하고 있으므로 jump된 마지막 주소를 JAL명령으로 LR에 저장할 수 있다. Nested call의 경우 JAL을 이용하여 계속 들어가면 이전의 LR값이 사라질 수 있으므로, 이때는 LR의 값을 Push하여 계속 보존할 수 있다. 만일 leaf function에서 빠져 나올 경우에는 POP PC하여 빠져 나올 수 있다.

```
M : JAL L1              // LR = M+2
    ...
L1: ...
    PUSH LR             // MEM(SP) = M+2
    JAL L2              // LR = L2 + a
L2: ...
    POP PC              // PC = M
```

위의 예는 L2에서 L1을 거치치 않고 바로 M으로 복귀하는 과정을 보여 준다.

```
M : JAL L1              // LR = M+2
    ...
L1: ...
    PUSH LR            // MEM(SP) = M+2
    JAL L2             // LR = L2 + a
L2: ...
    PUSH LR            // MEM(SP) = L1
    JAL L3             // LR = L3 + a
```

```
9       ...
10  L10: ...
11      LD SP + 36, %R0     // SP + (Loop깊이 *4)
12      JR R0               // PC = M+2
```

위의 예는 nested call의 깊이가 깊은 경우 leaf function에서 한번에 빠져나오는 방법을 보여주고 있다. 만일 중간 함수들에서 call을 하고 return이 필요 없다면 call에서 jal 대신 jmp를 사용하면 위의 예를 더 간단히 구현 할 수 있다.

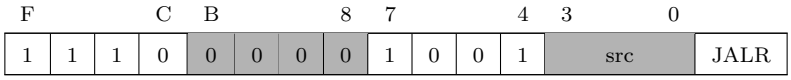
```
1   M  : JAL L1             // LR = M+2
2       ...
3   L1 : ...
4       JMP L2
5   L2 : ...
6       JMP L3
7       ...
8   L10: ...
9       JPLR                // PC = M+2
```

위의 예는 L1~L9까지의 call과정에서 return이 요구되지 않는 경우의 예를 보이고 있다. 이 경우 M에서 JAL을 한번 수행하고 leaf function에서 JPLR을 수행함으로서 M+2로 바로 빠져 나올 수 있다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.26 JALR - register indirect jump and link

JALR 명령은 무조건 분기를 수행하며, 현재 PC의 값은 Link Register에 저장하여, 추후에 복귀를 도운다. 무조건 분기의 목적 주소는 레지스터 값으로서 지정된다.



Syntax

```
JAL  %Rsrc
```

Exceptions – Prefetch 취소

Status Modification 변화 없음

Operation

```
%LR = %PC
%PC = %Rsrc
```

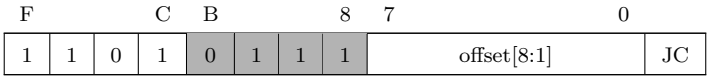
Usage JALR명령은 leaf function으로 jump 할 때 사용하는 branch명령이다. 단, 목적 주소가 레지스터를 이용하려 지정된다는 것만 차이가 있다.

```
1       LEA L1, %R2
2       JALR %R2
3       SUB %R2, %R4
4   L1 : ...
5       JPLR
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.27 JC - jump on carry

JC 명령은 Carry flag이용하여 조건 분기를 수행한다. carry flag의 값이 '1'인 경우에 대상 주소로 분기한다.

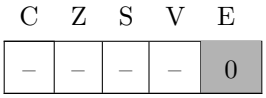


Syntax

```
JC  <label>
JC  <imm>
```

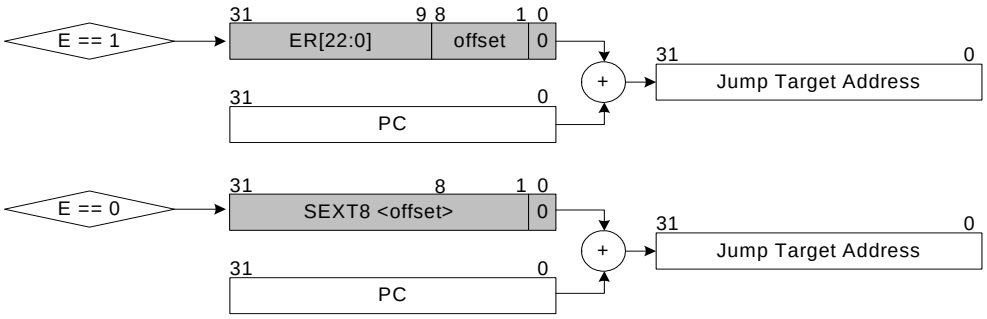
Exceptions – Prefetch 취소

Status Modification E-Flag를 항상 clear한다.



Operation

```
if(E_flag)
     branch_offset = (ER << 8 + offset) << 1
else
     branch_offset = SEXT8(offset) << 1
if(C_flag)
     %PC = %PC + branch_offset
```



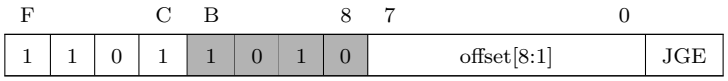
Usage JC는 상태 플래그 중 Carry flag를 이용하여 분기를 수행하는 명령으로서, 연산의 결과 carry비트가 발생하는 경우 이에 대한 처리를 해줄 수 있다.

```
1         ADD %R2, %R3
2         JC C_SAVE
3         ...
4   C_SAVE : ADDQ 0x1, %R4
5         ...
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.28 JGE - jump on signed greater or equal

JGE 명령은 비교의 결과 대상의 값이 부호를 고려하여 크거나 같은 경우 조건 분기를 수행한다. 상태 플래그들중 S_flag^V_flag = 0인 경우 조건 분기가 수행된다.

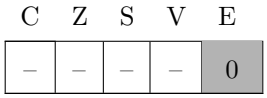


Syntax

```
JGE  <label>
JGE  <imm>
```

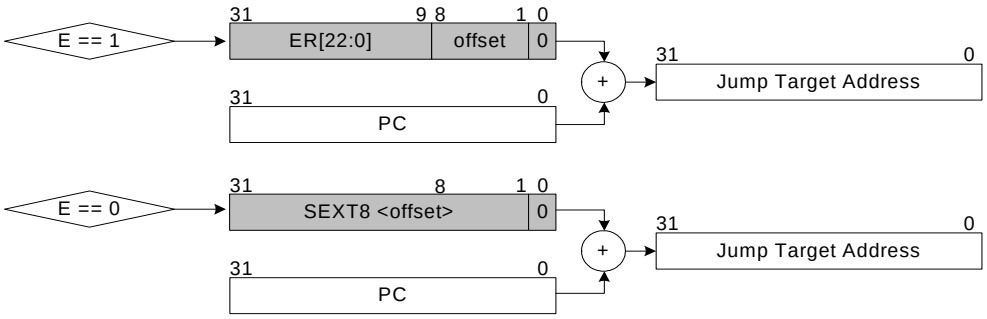
Exceptions – Prefetch 취소

Status Modification E-Flag를 항상 clear한다.



Operation

```
if(E_flag)
     branch_offset = (ER << 8 + offset) << 1
else
     branch_offset = SEXT8(offset) << 1
if(S_flag ^ V_flag == 0)
     %PC = %PC + branch_offset
```



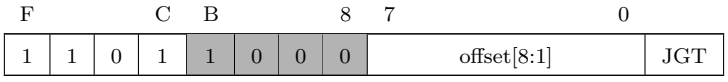
Usage JGE는 상태 플래그 중 sign flag와 overflow flag의 결과를 이용하여 분기를 수행하는 명령으로서, 비교 명령을 이용하는 경우 두 피연산자간의 부호를 고려하여 목적 레지스터의 값이 소스 레지스터의 값보다 크거나 같은 경우에 적용된다.

```
1  CMP %R2, %R3              // if (%R3 >= %R2)
2  JGE LABEL
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.29 JGT - jump on signed greater

JGT 명령은 비교의 결과 대상의 값이 부호를 고려하여 큰 경우 조건 분기를 수행한다. 상태 플래그들 중 Z_flag | (S_flag^V_flag) = 0인 경우 조건 분기가 수행된다.



Syntax

```
JGT  <label>
JGT  <imm>
```

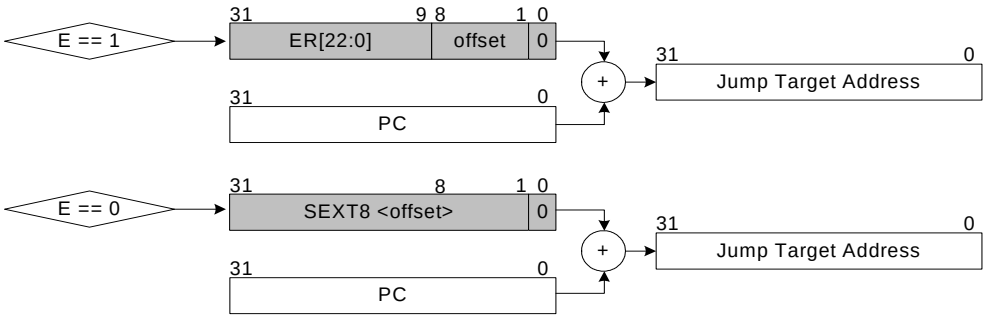
Exceptions – Prefetch 취소

Status Modification E-Flag를 항상 clear한다.

C	Z	S	V	E
–	–	–	–	0

Operation

```
if(E_flag)
     branch_offset = (ER << 8 + offset) << 1
else
     branch_offset = SEXT8(offset) << 1
if(Z_flag | (S_flag ^ V_flag) == 0)
     %PC = %PC + branch_offset
```



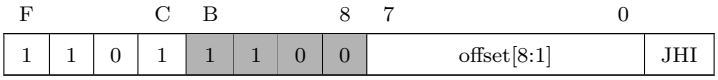
Usage JGT는 상태 플래그 중 zero flag, sign flag, overflow flow의 결과를 이용하여 분기를 수행하는 명령으로서, 비교 명령을 이용하는 경우 두 피연산자간의 부호를 고려하여 목적 레지스터의 값이 소스 레지스터의 값보다 큰 경우에 적용된다.

```
1  CMP %R2, %R3              // if (%R3 > %R2)
2  JGT LABEL
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.30 JHI - jump on unsigned higher

JHI 명령은 대상 레지스터의 값이 소스 레지스터의 값 보다 부호를 고려하지 않고 비교하여 큰 경우 분기를 수행한다. 상태 플래그들 중 C_flag | Z_flag = 0인 경우 조건 분기가 수행된다.

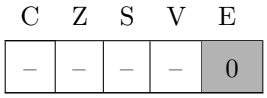


Syntax

```
JHI  <label>
JHI  <imm>
```

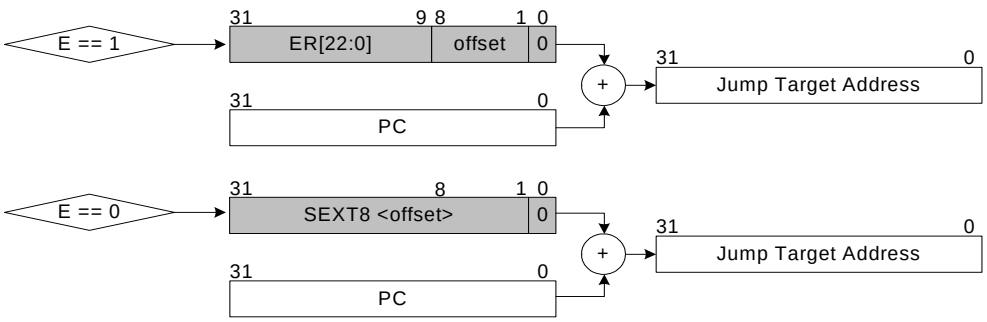
Exceptions – Prefetch 취소

Status Modification E-Flag를 항상 clear한다.



Operation

```
if(E_flag)
     branch_offset = (ER << 8 + offset) << 1
else
     branch_offset = SEXT8(offset) << 1
if(C_flag | Z_flag == 0)
     %PC = %PC + branch_offset
```



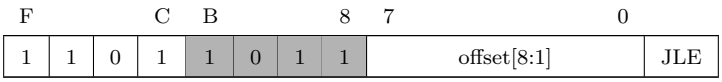
Usage JHI는 상태 플래그 중 carry flag, zero flag의 결과를 이용하여 분기를 수행하는 명령으로서, 비교 명령을 이용하는 경우 두 피연산자간의 부호를 고려하지 않고, 목적 레지스터의 값이 소스 레지스터의 값보다 큰 경우에 적용된다.

```
1  CMP %R2, %R3              // if (%R3 > %R2)
2  JHI LABEL
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.31 JLE - jump on signed less or equal

JLE 명령은 대상 레지스터의 값이 소스 레지스터의 값 보다 부호를 고려하고 비교하여 작거나 같은 경우 분기를 수행한다. 상태 플래그들 중 Z_flag — (S_flagˆV_flag) = 1인 경우 조건 분기가 수행된다.

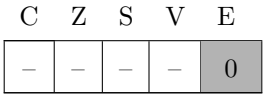


Syntax

```
JLE  <label>
JLE  <imm>
```

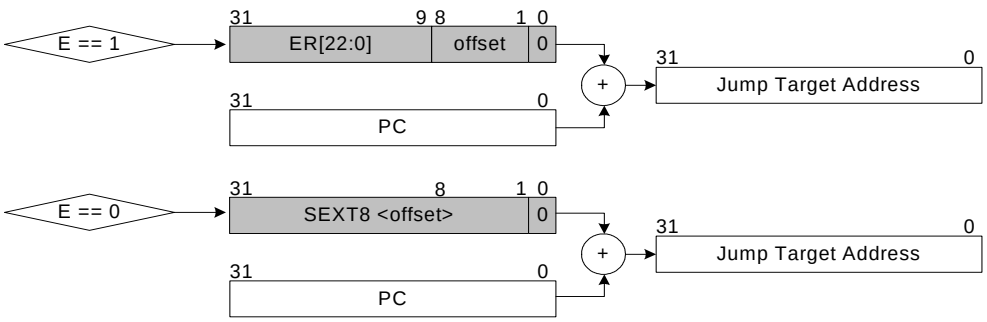
Exceptions – Prefetch 취소

Status Modification E-Flag를 항상 clear한다.



Operation

```
if(E_flag)
     branch_offset = (ER << 8 + offset) << 1
else
     branch_offset = SEXT8(offset) << 1
if(Z_flag | (S_flag ^ V_flag) == 1)
     %PC = %PC + branch_offset
```



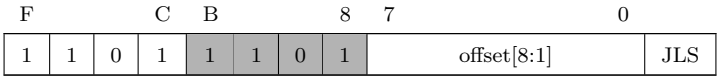
Usage JLE는 상태 플래그 중 zero flag, sign flag, overflow flow의 결과를 이용하여 분기를 수행하는 명령으로서, 비교 명령을 이용하는 경우 두 피연산자 간의 부호를 고려하여, 목적 레지스터의 값이 소스 레지스터의 값보다 작거나 같은 경우에 적용된다.

```
1  CMP %R2, %R3              // if (%R3 <= %R2)
2  JLE LABEL
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.32 JLS - jump on unsigned lower or equal

JLS 명령은 대상 레지스터의 값이 소스 레지스터의 값 보다 부호를 고려하지 않고 비교하여 작거나 같은 경우 분기를 수행한다. 상태 플래그들 중 C_flag | Z_flag = 1인 경우 조건 분기가 수행된다.

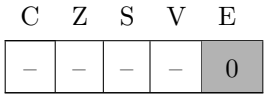


Syntax

```
JLS  <label>
JLS  <imm>
```

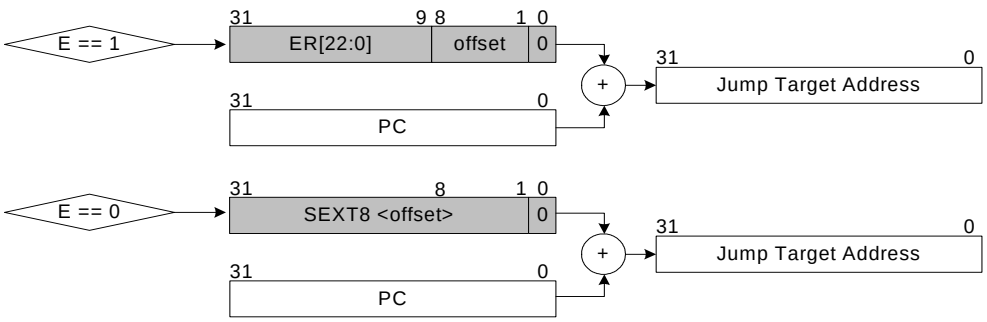
Exceptions – Prefetch 취소

Status Modification E-Flag를 항상 clear한다.



Operation

```
if(E_flag)
     branch_offset = (ER << 8 + offset) << 1
else
     branch_offset = SEXT8(offset) << 1
if(C_flag | Z_flag == 1)
     %PC = %PC + branch_offset
```



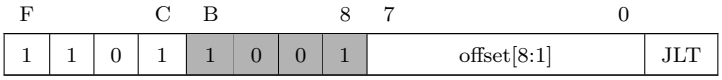
Usage JLS는 상태 플래그 중 carry flag, zero flag 의 결과를 이용하여 분기를 수행하는 명령으로서, 비교 명령을 이용하는 경우 두 피연산자간의 부호를 고려하지 않고 목적 레지스터의 값이 소스 레지스터의 값보다 작거나 같은 경우에 적용된다.

```
1  CMP %R2, %R3              // if (%R3 <= %R2)
2  JLS LABEL
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.33 JLT - jump on signed less than

JLT 명령은 대상 레지스터의 값이 소스 레지스터의 값 보다 부호를 고려하고 비교하여 작은 경우 분기를 수행한다. 상태 플래그들 중 S_flag ^ V_flag = 1인 경우 조건 분기가 수행된다.

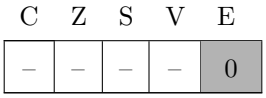


Syntax

```
JLT  <label>
JLT  <imm>
```

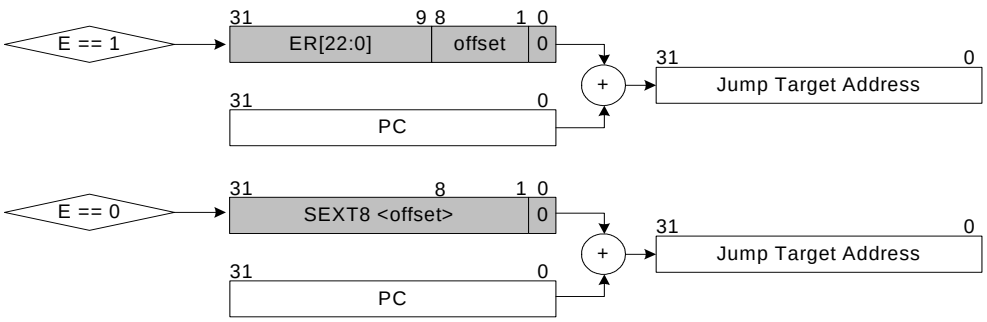
Exceptions – Prefetch 취소

Status Modification E-Flag를 항상 clear한다.



Operation

```
if(E_flag)
     branch_offset = (ER << 8 + offset) << 1
else
     branch_offset = SEXT8(offset) << 1
if(S_flag ^ V_flag == 1)
     %PC = %PC + branch_offset
```



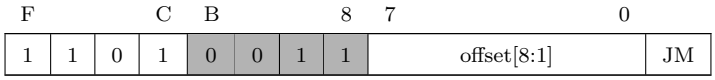
Usage JLT는 상태 플래그 중 sign flag, overflow flag 의 결과를 이용하여 분기를 수행하는 명령으로서, 비교 명령을 이용하는 경우 두 피연산자간의 부호를 고려하고 목적 레지스터의 값이 소스 레지스터의 값보다 작은 경우에 적용된다.

```
1  CMP %R2, %R3              // if (%R3 < %R2)
2  JLT LABEL
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.34 JM - jump on minus

JM 명령은 대상 레지스터의 값이 소스 레지스터의 값 보다 부호를 고려하지 않고 비교하여 작은 경우 분기를 수행한다. 상태 플래그들 중 S_flag = 1인 경우 조건 분기가 수행된다.

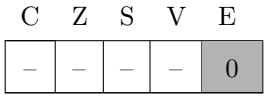


Syntax

```
JM  <label>
JM  <imm>
```

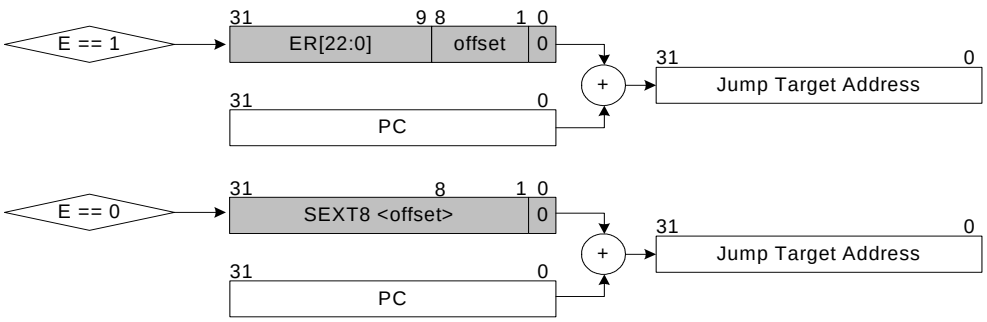
Exceptions – Prefetch 취소

Status Modification E-Flag를 항상 clear한다.



Operation

```
if(E_flag)
     branch_offset = (ER << 8 + offset) << 1
else
     branch_offset = SEXT8(offset) << 1
if(S_flag == 1)
     %PC = %PC + branch_offset
```



Usage JM은 비교 명령어와 같이 사용할 때 %R3가 작은 경우 사용된다. 또한, 비교 명령과 같이 사용되지 않는 경우에는 이전 연산의 결과가 음수인 경우(즉 sign flag가 1인 경우)에 분기를 수행한다.

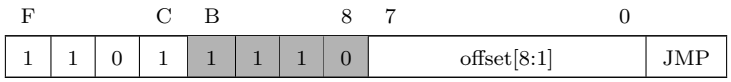
```
1  CMP %R2, %R3              // if (%R3 < %R2)
2  JM LABEL
```

* 이 예제에서 이용한 %R3 레지스터는 임의로 지정된 것이며, 레지스터 사용에 대한 제약은 없다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.35 JMP - jump always

JMP 명령은 무조건 분기를 수행한다.

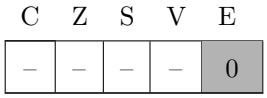


Syntax

```
JMP  <label>
JMP  <imm>
```

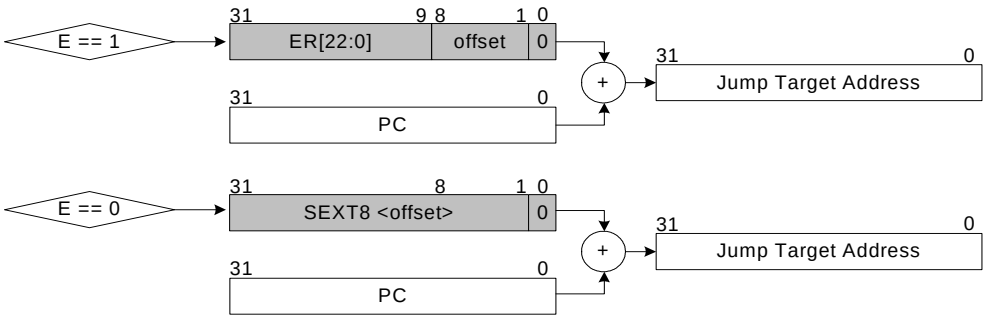
Exceptions – Prefetch 취소

Status Modification E-Flag를 항상 clear한다.



Operation

```
if(E_flag)
     branch_offset = (ER << 8 + offset) << 1
else
     branch_offset = SEXT8(offset) << 1
%PC = %PC + branch_offset
```



Usage JMP는 무조건 분기를 수행하는 명령이다.

```
1   JMP LABEL
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.36 JNC - jump on not carry

JNC 명령은 프로세서의 carry flag가 '0'인 경우 분기를 수행한다.

F			C	B			8	7	0
1	1	0	1	0	1	1	0	offset[8:1]	JNC

Syntax

```
JNC  <label>
JNC  <imm>
```

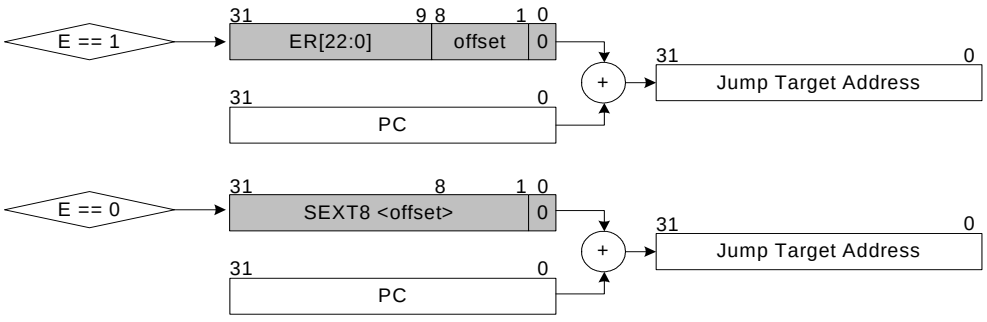
Exceptions – Prefetch 취소

Status Modification E-Flag를 항상 clear한다.

C	Z	S	V	E
–	–	–	–	0

Operation

```
if(E_flag)
     branch_offset = (ER << 8 + offset) << 1
else
     branch_offset = SEXT8(offset) << 1
if(C_flag == 0)
     %PC = %PC + branch_offset
```



Usage JNC는 연산에 결과(혹은 이전의 연산, 동작)에 대하여 캐리가 발생하지 않은 경우 분기를 수행하는 명령으로서, 캐리가 발생하였을 때 추가적인 연산을 해 줄 필요가 있는 경우 사용된다.

```
1         ADD %R2, %R3
2         JNC LABEL
3         ADDQ 0x1, %R4
4   LABEL :
5         ...
```

위의 예는 덧셈의 결과 carry가 발생하는 경우 상위 값을 가지는 레지스터에 carry의 결과를 반영시키는 예이다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.37 JNV - jump on not overflow

JNV 명령은 프로세서의 overflow flag가 '0'인 경우 분기를 수행한다.

F			C	B			8	7	0
1	1	0	1	0	0	0	0	offset[8:1]	JNV

Syntax

```
JNV  <label>
JNV  <imm>
```

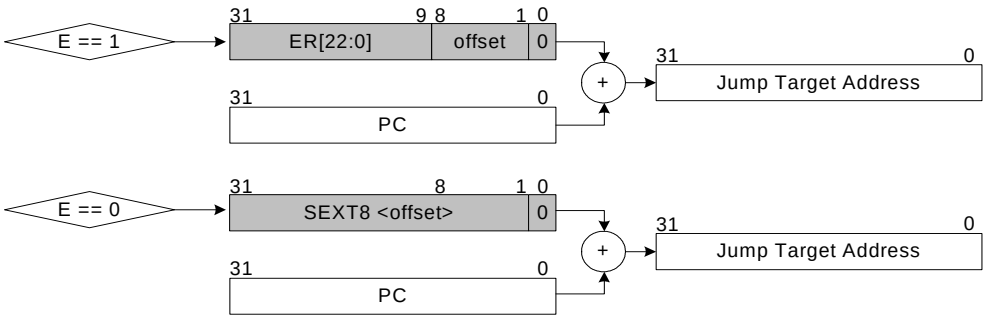
Exceptions – Prefetch 취소

Status Modification E-Flag를 항상 clear한다.

C	Z	S	V	E
–	–	–	–	0

Operation

```
if(E_flag)
     branch_offset = (ER << 8 + offset) << 1
else
     branch_offset = SEXT8(offset) << 1
if(V_flag == 0)
     %PC = %PC + branch_offset
```



Usage JNV는 연산에 결과(혹은 이전의 연산, 동작)에 대하여 오버 플로우가 발생하지 않은 경우 분기를 수행하는 명령으로서, 오버 플로우 발생에 대한 처리가 필요한 경우 사용된다.

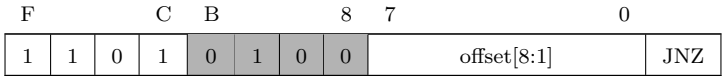
```
1         ADD %R2, %R3
2         JNV LABEL
3         ADDQ 0x1, %R4
4   LABEL :
5         ...
```

위의 예는 덧셈의 결과 overflow를 발생시키는 경우 분기를 발생시키는 예이다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.38 JNZ - jump on not zero

JNZ 명령은 프로세서의 zero flag가 '0'인 경우 분기를 수행한다.



Syntax

```
JNZ  <label>
JNZ  <imm>
```

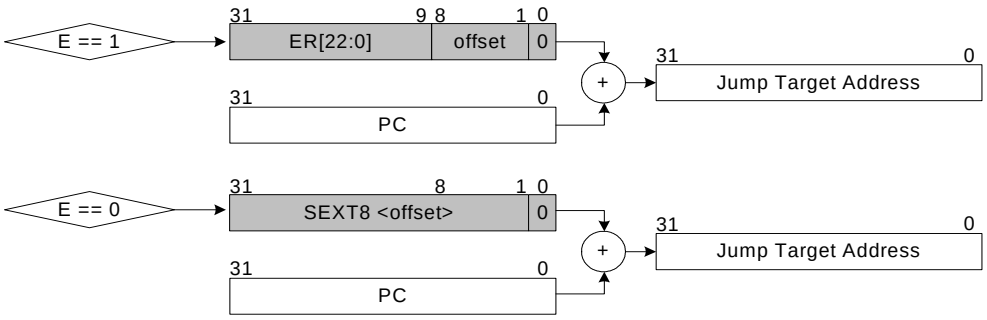
Exceptions – Prefetch 취소

Status Modification E-Flag를 항상 clear한다.

C	Z	S	V	E
–	–	–	–	0

Operation

```
if(E\_flag)
     branch_offset = (ER << 8 + offset) << 1
else
     branch_offset = SEXT8(offset) << 1
if(Z_flag == 0)
     %PC = %PC + branch_offset
```



Usage JNZ는 연산에 결과가 '0'이 아닌 경우, 특히 비교의 결과 두 값이 같지 않은 경우 수행되는 분기이다.

```
1         CMP %R2, %R3
2         JNZ LABEL
3         ...
4   LABEL :
5         ...
```

위의 예는 %R2와 %R3가 같지 않은 경우 LABEL로 분기하는 예를 보여준다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.39 JP - jump on positive

JP 명령은 프로세서의 sign flag가 '0'인 경우 분기를 수행한다.

F			C	B			8	7							0
1	1	0	1	0	0	1	0	offset[8:1]							JP

Syntax

```
JP  <label>
JP  <imm>
```

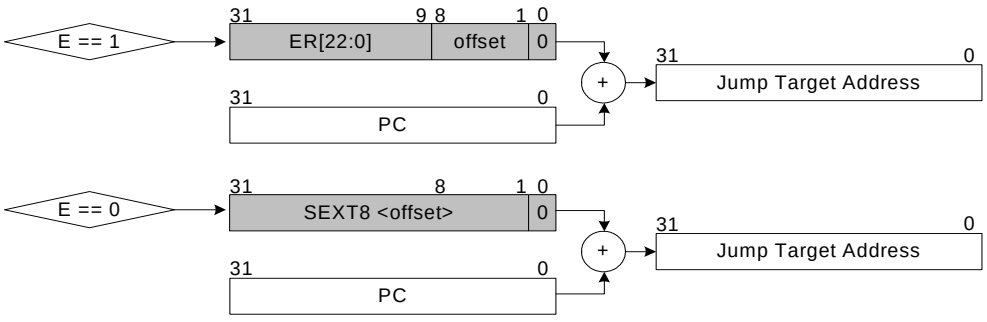
Exceptions – Prefetch 취소

Status Modification E-Flag를 항상 clear한다.

C	Z	S	V	E
–	–	–	–	0

Operation

```
if(E_flag)
     branch_offset = (ER << 8 + offset) << 1
else
     branch_offset = SEXT8(offset) << 1
if(S_flag == 0)
     %PC = %PC + branch_offset
```



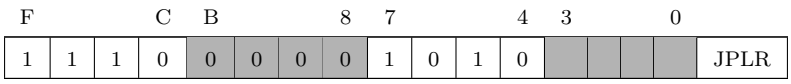
Usage JP는 연산에 결과가 양수가 된 경우 즉, sign flag가 '0'인 경우 수행되는 분기이다.

```
1         ADD %R2, %R3
2         JP LABEL
3         ...
4   LABEL :
5         ...
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.40 JPLR - jump to link register

JPLR 명령은 Link register에 저장된 주소로 분기를 수행한다.

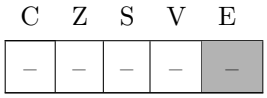


Syntax

```
JPLR
```

Exceptions – Prefetch 취소

Status Modification 변화 없음



Operation

```
%PC = %LR
```

Usage JPLR은 이전의 JAL 혹은 JALR을 통하여 LR에 저장된 주소로 복귀하는 명령으로써, sub function에서 복귀할때 사용한다.

```
1         JAL LABEL
2         ADD
3         ...
4   LABEL :
5         ...
6         JPLR
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.41 JR - register indirect jump

JR 명령은 범용 레지스터에 저장되어 있는 주소로 무조건 분기한다.

F			C	B			8	7			4	3			0
1	1	1	0	0	0	0	0	1	0	0	0	dst reg.			JR

Syntax

```
JR %Rdst
```

Exceptions – Prefetch 취소

Status Modification 변화 없음

C	Z	S	V	E
–	–	–	–	–

Operation

```
%PC = %Rdst
```

Usage JR은 레지스터를 이용한 간접분기로써, 일반적으로는 많이 사용되지 않는다. 그러나, 공유 라이브러리를 사용하는 경우 라이브러리의 함수 포인터를 지정하여 분기하는 경우가 증가하고 있으므로, 레지스터를 이용한 간접 분기 역시 증가한다.

```
1 JR %R3
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.42 JV - jump on overflow

JV 명령은 프로세서의 overflow flag가 '1'인 경우 분기를 수행한다.

F			C	B			8	7	0
1	1	0	1	0	0	0	1	offset[8:1]	JV

Syntax

```
JV  <label>
JV  <imm>
```

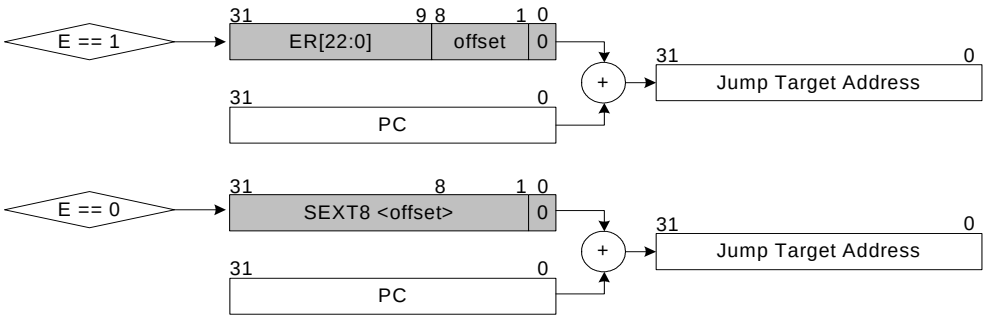
Exceptions – Prefetch 취소

Status Modification E-Flag를 항상 clear한다.

C	Z	S	V	E
–	–	–	–	0

Operation

```
if(E_flag)
     branch_offset = (ER << 8 + offset) << 1
else
     branch_offset = SEXT8(offset) << 1
if(V_flag == 1)
     %PC = %PC + branch_offset
```



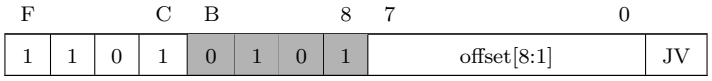
Usage JV는 연산에 결과에 대하여 overflow가 발생한 경우 분기를 수행하는 명령어로써 overflow 발생에 대한 처리가 필요한 경우 사용된다.

```
1         ADD %R2, %R3
2         JV LABEL
3         ...
4   LABEL :
5         ...
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.43 JZ - jump on zero

JZ 명령은 프로세서의 zero flag가 '1'인 경우 분기를 수행한다.



Syntax

```
JZ   <label>
JZ   <imm>
```

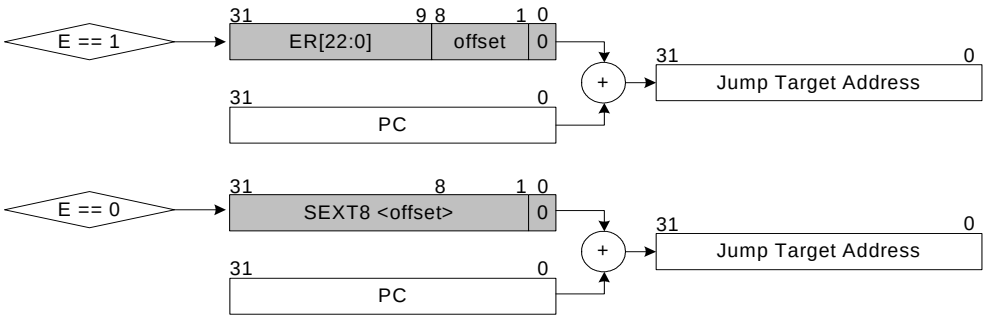
Exceptions – Prefetch 취소

Status Modification E-Flag를 항상 clear한다.

C	Z	S	V	E
–	–	–	–	0

Operation

```
if(E_flag)
     branch_offset = (ER << 8 + offset) << 1
else
     branch_offset = SEXT8(offset) << 1
if(Z_flag == 1)
     %PC = %PC + branch_offset
```



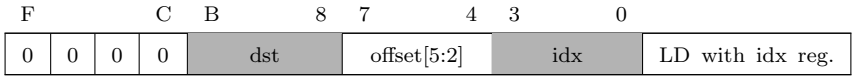
Usage JZ는 연산에 결과가 '0'인 경우 분기를 수행하는 명령어로써 두 값을 비교하였을 때 두 값이 같을 경우 분기하는 명령어이다.

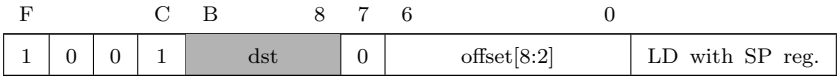
```
1         CMP %R2, %R3
2         JZ LABEL
3         ...
4   LABEL :
5         ...
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.44 LD - load 32-bit

LD 명령은 지정한 메모리 주소로부터 32비트의 데이터를 읽어서 대상 레지스터에 저장한다.





Syntax

```
LD (%Ridx, <imm>), %Rdst
LD <label>, %Rdst
LD <imm>,  %Rdst
```

%Ridx 대신 현재 Stack Pointer를 이용할 수 있다.

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification E-Flag를 항상 clear한다.

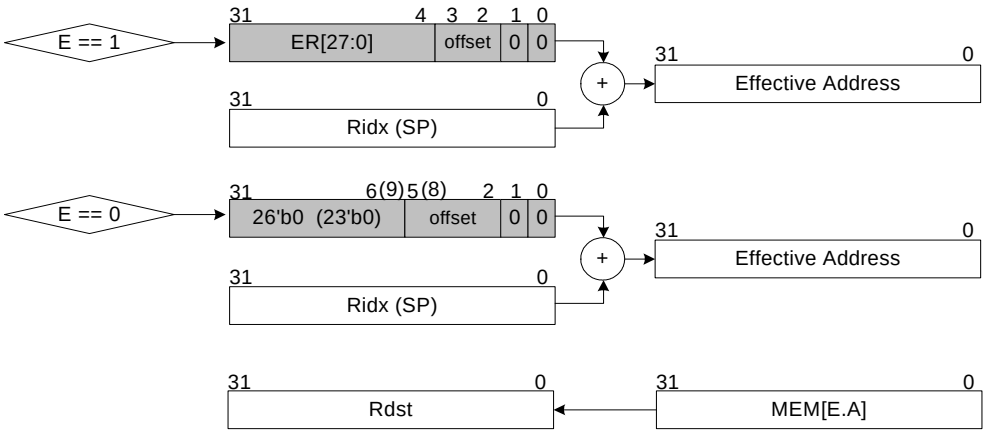
C	Z	S	V	E
–	–	–	–	0

Operation

```
if(E_flag)
     imm = (ER << 4) + ((offset & 0x3) << 2)
else
     imm = ZEXT9(offset << 2)
%Rdst = MEM((%Ridx or %SP) + imm)
```

Usage LD명령은 워드 단위(32비트)로 데이터를 메모리에서 읽어오는 동작을 수행한다.

```
1  LD (%R2, 0x4), %R3
```



* Stack Pointer를 사용하는 경우, 괄호 안의 숫자를 사용

위의 경우 %R2를 인덱스 레지스터로서 사용하고, 4를 offset으로 사용하여 메모리를 접근하여 32비트 데이터를 가지고 와서 %R3에 대입하는 동작을 보여준다. 인덱스 레지스터를 사용할 경우에는 offset(4비트)를 사용한다. E-Flag가 세팅되어 있는 경우, 하위 2비트를 사용({ER[27:0], offset[3:2], 2'b00})하여 imm을 생성하며 세팅되지 않았을 경우에는 4비트를 사용({26'b0, offset[5:2], 2'b00})하여 imm을 생성한다.

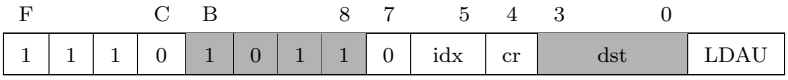
```
1  LD (%SP, 0x4), %R3
```

LD명령에서 인덱스 레지스터 이외에 스택 포인터를 이용할 수 있으며, 이 경우 offset(7비트)를 사용한다. E-Flag가 세팅되어 있는 경우, 하위 2비트를 사용({ER[27:0], offset[3:2], 2'b00})하여 imm을 생성하며 세팅되지 않았을 경우에는 7비트를 사용({23'b0, offset[8:2], 2'b00})하여 imm을 생성한다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.45 LDAU - Auto-Increment load

LDAU 명령은 CR레지스터를 이용하여 자동 증가 되는 주소를 생성하고 이를 바탕으로 load 명령을 수행한다.

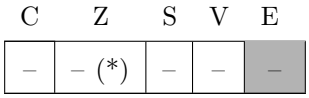


Syntax

```
LDAU CRNO, %Ridx, %Rdst
```

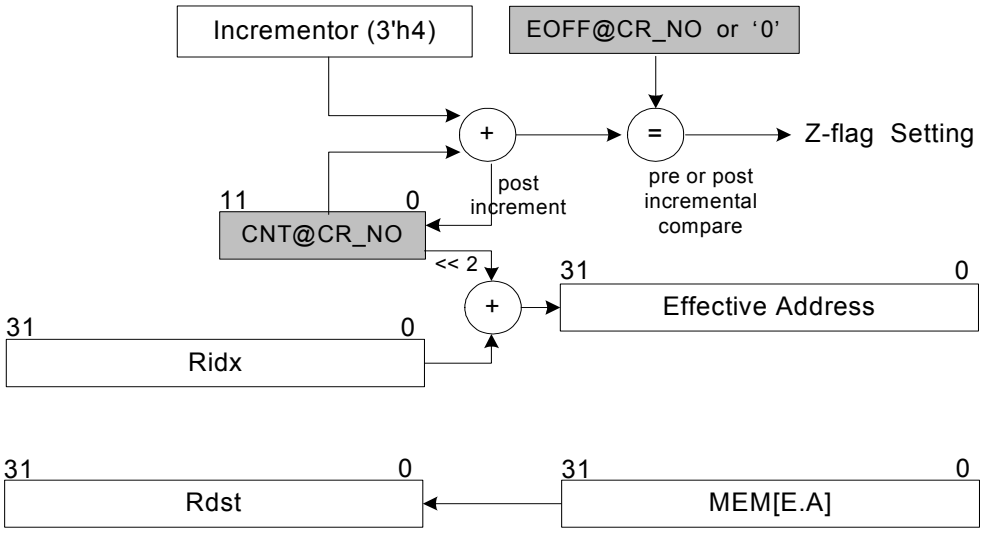
Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 동작 모드에 따라 zero-flag가 변화 됨



Operation

```
%Rdst = MEM(%Ridx + offset@CRNO)   // increment CR and check CR
```



Usage LDAU 연산은 CR레지스터를 이용하여 자동 증가 되는 주소를 생성하고 이를 바탕으로 load 명령을 수행하는 명령으로써 다음의 예는 CR0의 내용을 바탕으로 %R1와

함께 이용하여 메모리에 접근하고 해당 데이터를 %R2로 가져오는 예를 보인 것이다.

```
1  LDAU 0x0, %R1, %R2
```

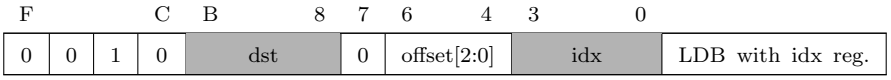
Note

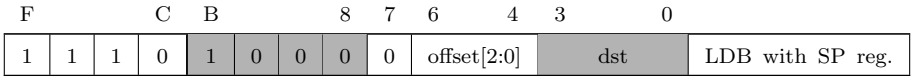
- Index register는 %R0, %R1, %SP를 사용할 수 있다.
- 00: %R0
- 01: %R1
- 1?: %SP
- 동작 모드는 CR0와 CR1의 세팅으로써 가능하며, 이에 따라 CR의 End-offset/Mask 필드를 사용한다. 각 레지스터의 필드 내용은 표. 2.2를 참고한다.
- 동작 모드가 Auto Incremental with End-offset compare mode 또는 Wrap-around Auto Incremental with End Check mode로 지정된 경우, 최종 카운트에 도달하면 SR의 zero-flag가 assert된다.
- Auto Incremental with End-offset compare mode인 경우에는 카운터의 증가되기 직전 값을 사용(pre-incremental compare)하고, Wrap-around Auto Incremental with End Check mode인 경우에는 증가된 이후의 카운터 값을 사용(post-incremental compare)한다.
- Index값은 변경되지 않으며, offset으로 사용되는 CR의 값이 변경된다

Processor Version 이 명령어는 AE32000 계열 중 AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다. 단, AE32000C의 경우, Auto Increment mode와 Auto Incremental with End-offset compare mode만 지원하며 AE32000C-Tiny의 경우에는 모드에 따라 부분적으로 지원되므로 해당 해당 프로세서의 reference manual을 참고한다.

4.46 LDB - load signed byte

LDB 명령은 지정한 메모리 주소로부터 8비트의 데이터를 읽어서 대상 레지스터에 저장한다.





Syntax

```
LDB (%Ridx,  <imm>+), %Rdst
LDB <label>, %Rdst
LDB <imm>,   %Rdst
```

%Ridx 대신 현재 Stack Pointer를 이용할 수 있다.

Exceptions 이 명령에 연관된 예외 사항은 없다.

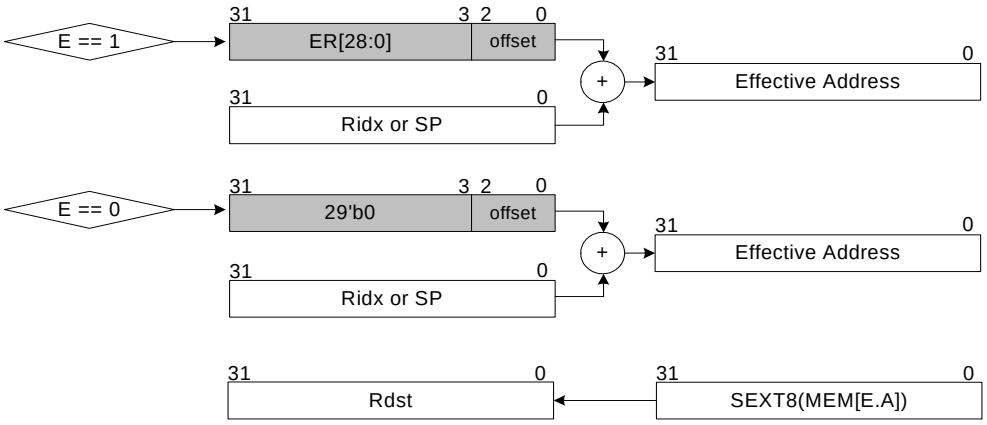
Status Modification E-Flag를 항상 clear한다.

C	Z	S	V	E
–	–	–	–	0

Operation

```
if(E_flag)
     imm = (ER << 3) + offset
else
     imm = ZEXT3(offset)
%Rdst = SEXT8(MEM((%Ridx or %SP) + imm))
```

Usage LDB명령은 바이트 단위(8비트)로 데이터를 메모리에서 읽어오는 동작을 수행한다.



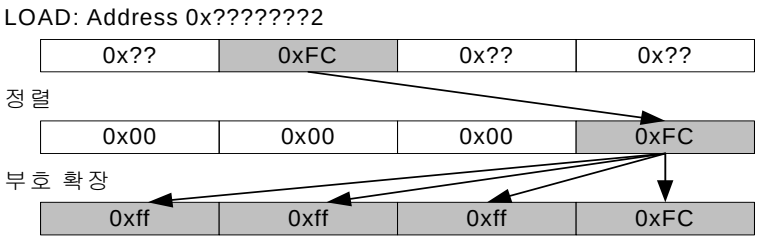

```
1 LDB (%R2, 0x4), %R3
```

위의 경우 %R2를 인덱스 레지스터로서 사용하고, 4를 offset으로 사용하여 메모리를 접근하여 8비트 데이터를 가지고 와서 부호 확장을 취한 후 %R3에 대입하는 동작을 보여준다. LDB명령에서 인덱스 레지스터 이외에 스택 포인터를 이용할 수 있으며, 이 경우 아래와 같이 사용한다.

```
1 LDB (%SP, 0x4), %R3
```

Note

- LDB는 LERI를 이용하여 offset을 확장하는 것이 가능하다. 따라서 32비트까지 offset을 지정하는 것이 가능하다. 단, 사용자가 명시적으로 LERI를 넣을 필요가 없으며, 필요한 만큼 offset의 값을 지정하는 경우 자동적으로 어셈블러에서 LERI를 생성해 주시 때문에 사용자는 어셈블러를 이용할 때 임의로 LERI를 쓰지 않아도 된다. (LDB의 경우 offset의 길이를 31비트 이하로 할 것을 권장한다.)

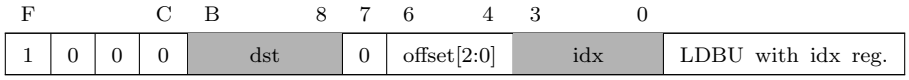


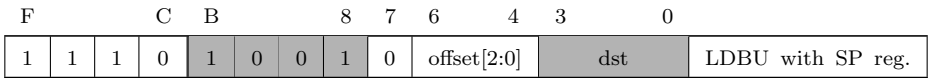
- AE32000은 32비트 데이터 버스 및 32비트 단위로 정렬되어 있는데이터를 이용하므로, 8비트 데이터를 읽어와서 그 위치를 정렬하고 부호 확장을 하여야 한다. 위의 그림은 정렬과 부호 확장 과정을 도시하고 있다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.47 LDBU - load unsigned byte

LDBU 명령은 지정한 메모리 주소로부터 8비트의 데이터를 읽어서 무부호 확장을 수행한 후 대상 레지스터에 저장한다.





Syntax

```
LDBU (%Ridx, <imm>+), %Rdst
LDBU <label>, %Rdst
LDBU <imm>+, %Rdst
```

%Ridx 대신 현재 Stack Pointer를 이용할 수 있다.

Exceptions 이 명령에 연관된 예외 사항은 없다.

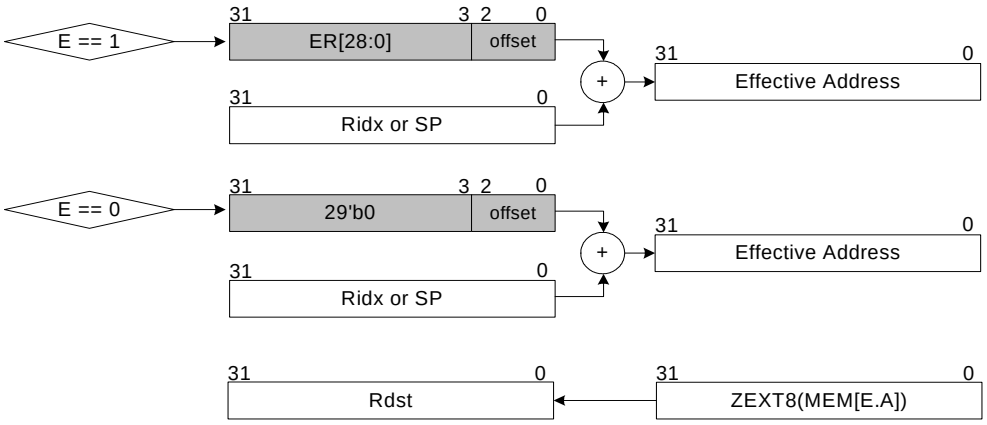
Status Modification E-Flag를 항상 clear한다.

C	Z	S	V	E
–	–	–	–	0

Operation

```
if(E_flag)
     imm = (ER << 3) + offset
else
     imm = ZEXT3(offset)
%Rdst = ZEXT8(MEM((%Ridx or %SP) + imm))
```

Usage LDBU명령은 바이트 단위(8비트)로 데이터를 메모리에서 읽어오는 동작을 수행한다.



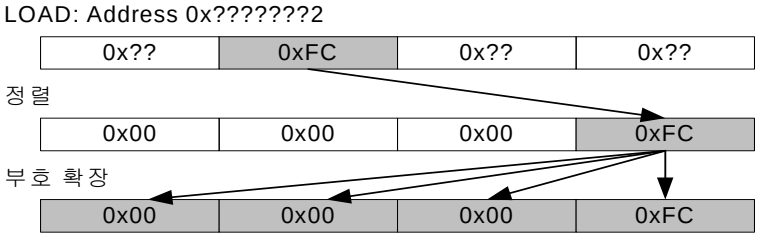

```
1  LDBU (%R2, 0x4), %R3
```

위의 경우 %R2를 인덱스 레지스터로서 사용하고, 4를 offset으로 사용하여 메모리를 접근하여 8비트 데이터를 가지고 와서 무부호 확장을 취한 후 %R3에 대입하는 동작을 보여준다. LDBU 명령에서 인덱스 레지스터 이외에 스택 포인터를 이용할 수 있으며, 이 경우 아래와 같이 사용한다.

```
1  LDBU (%SP, 0x4), %R3
```

Note

- LDBU는 LERI를 이용하여 offset을 확장하는 것이 가능하다. 따라서 32비트까지 offset을 지정하는 것이 가능하다. 단, 사용자가 명시적으로 LERI를 넣을 필요가 없으며, 필요한 만큼 offset의 값을 지정하는 경우 자동적으로 어셈블러에서 LERI를 생성해 주시 때문에 사용자는 어셈블러를 이용할 때 임의로 LERI를 쓰지 않아도 된다. (LDBU의 경우 offset의 길이를 31비트 이하로 할 것을 권장한다.)



- AE32000은 32비트 데이터 버스 및 32비트 단위로 정렬되어 있는데이터를 이용하므로, 8비트 데이터를 읽어와서 그 위치를 정렬하고 부호 확장을 하여야 한다. 위의 그림은 정렬과 부호 확장 과정을 도시하고 있다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.48 LDC - load on coprocessor

LDC 명령은 지정한 메모리 주소로부터 데이터를 읽어서 보조 프로세서의 대상 레지스터에 저장한다. 데이터의 크기는 최소 32비트이며, 보조 프로세서에 추가적인 메모리 버스를 추가한다면 그 이상이 될 수 있다.

F			C	B		9	8	7			4	3	0
1	1	1	0	1	1	CP #		1	0	0	idx	dst	LDCn

Syntax

```
LDC   <cp_no>, (%R1, <imm>), %Rdst
LDC   <cp_no>, (%SP, <imm>), %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification E-Flag를 항상 clear한다.

C	Z	S	V	E
–	–	–	–	0

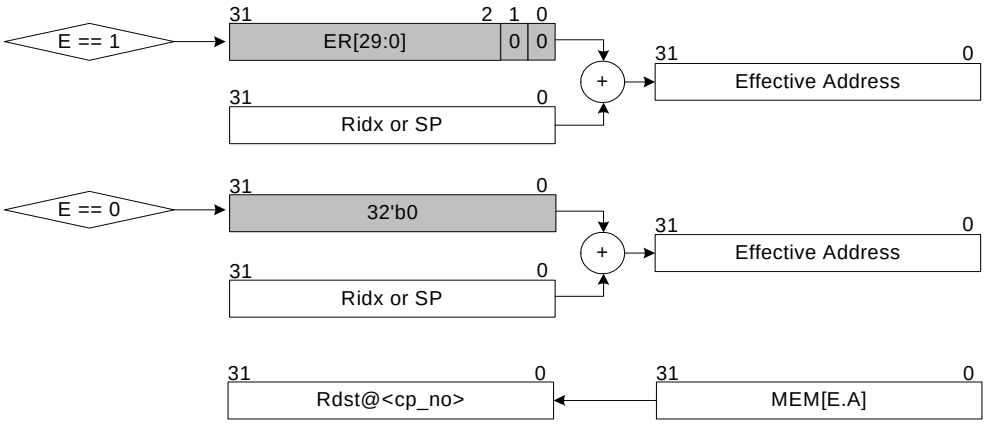
Operation LDC 명령어는 idx 값이 1인 경우 스택포인터 레지스터를 지정하며, idx 값이 0인 경우 범용 레지스터 R1을 지정한다.

```
if(E_flag)  imm = ER << 2
else        imm = 0

if(idx == 1)
  %Rdst@CP<cp_no> = MEM(%SP + imm)
else
  %Rdst@CP<cp_no> = MEM(%R1 + imm)
```

Usage LDC명령은 바이트 단위(32비트)로 데이터를 메모리에서 읽어서 보조 프로세서의 대상 레지스터로 가지고 오는 동작을 수행한다.

```
1  LDC 0x1, (%R1, 0x4), %R3
```



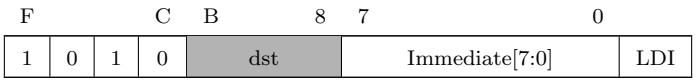
Note

- 이 명령은 명령어 필드 상에는 immediate 필드가 없으나 LERI를 이용하여 offset을 지정하는 것이 가능하다. 만일 LERI를 이용할 경우 하위 2비트는 0으로 지정되므로, 30비트까지 사용자가 주소를 지정하는 것이 가능하다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다. 단, AE32000B에서는 구현에 따라 다를 수 있으므로 해당 해당 프로세서의 reference manual을 참고한다.

4.49 LDI - load immediate

LDI 명령은 즉치 값을 대상 범용 레지스터에 저장하는 동작을 수행한다.

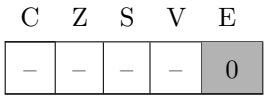


Syntax

```
LDI   <imm>, %Rdst
```

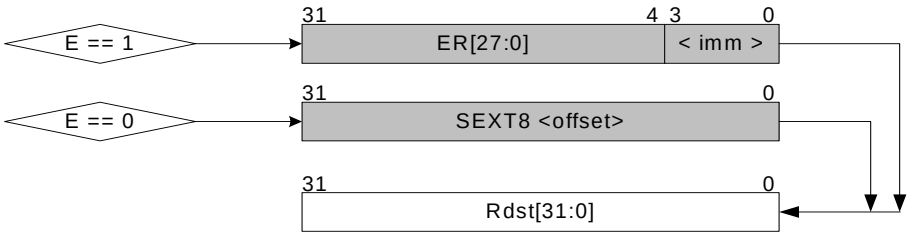
Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification E-Flag를 항상 clear한다.



Operation

```
if(E_flag)
     imm = (ER << 4) + imm[3:0]
else
     imm = SEXT8(imm)
%Rdst = imm
```



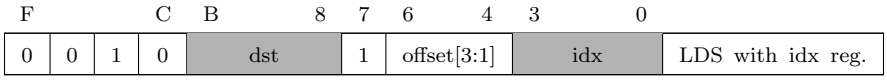
Usage LDI는 즉치 값을 레지스터에 저장할 때 사용된다.

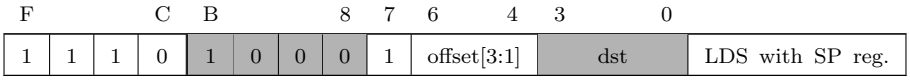
```
1   LDI 0x1, %R2
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.50 LDS - load signed short

LDS 명령은 지정한 메모리 주소로부터 16비트의 데이터를 읽어서 부호 확장을 수행한 후 대상 레지스터에 저장한다.





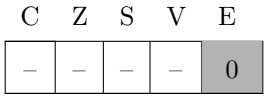
Syntax

```
LDS (%Ridx,  <imm>+), %Rdst
LDS <label>, %Rdst
LDS <imm>,   %Rdst
```

%Ridx 대신 현재 Stack Pointer를 이용할 수 있다.

Exceptions 이 명령에 연관된 예외 사항은 없다.

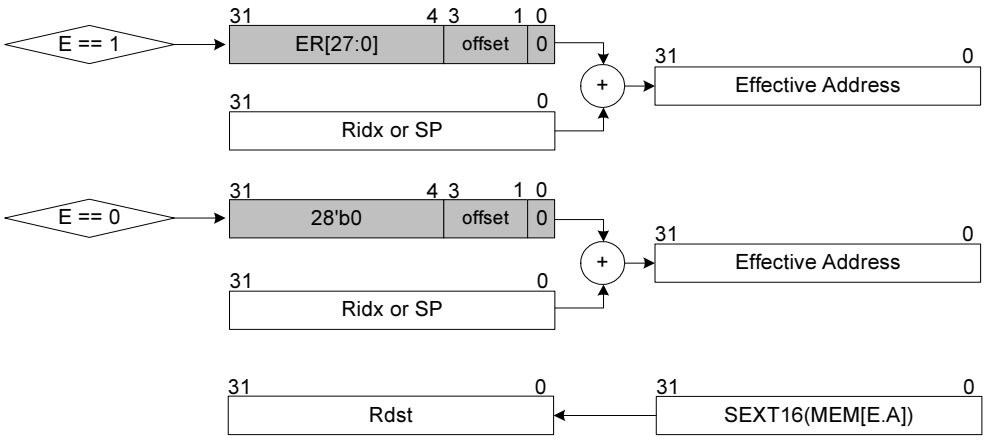
Status Modification E-Flag를 항상 clear한다.



Operation

```
if(E_flag)
     imm = ((ER << 3) + offset) << 1
else
     imm = ZEXT4(offset << 1)
%Rdst = SEXT16(MEM((%Ridx or %SP) + imm))
```

Usage LDS명령은 short 단위(16비트)로 데이터를 메모리에서 읽어오는 동작을 수행한다.



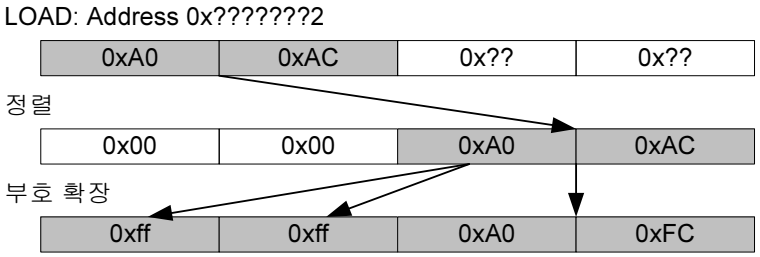

```
1  LDS (%R2, 0x4), %R3
```

위의 경우 %R2를 인덱스 레지스터로서 사용하고, 4를 offset으로 사용하여 메모리를 접근하여 16비트 데이터를 가지고 와서 부호 확장을 취한 후 %R3에 대입하는 동작을 보여준다. LDS 명령에서 인덱스 레지스터 이외에 스택 포인터를 이용할 수 있으며, 이 경우 아래와 같이 사용한다.

```
1  LDS (%SP, 0x4), %R3
```

Note

- LDS는 LERI를 이용하여 offset을 확장하는 것이 가능하다. 따라서 32비트까지 offset을 지정하는 것이 가능하다. 단, 사용자가 명시적으로 LERI를 넣을 필요가 없으며, 필요한 만큼 offset의 값을 지정하는 경우 자동적으로 어셈블러에서 LERI를 생성해 주시 때문에 사용자는 어셈블러를 이용할 때 임의로 LERI를 쓰지 않아도 된다.

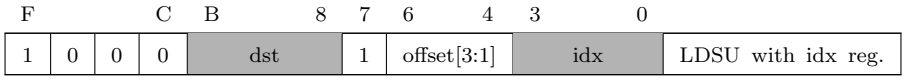


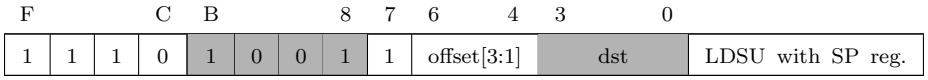
- AE32000은 32비트 데이터 버스 및 32비트 단위로 정렬되어 있는데이터를 이용하므로, 16비트 데이터를 읽어와서 그 위치를 정렬하고 부호 확장을 하여야 한다. 아래 그림은 정렬과 부호 확장 과정을 도시하고 있다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.51 LDSU - load unsigned short

LDSU 명령은 지정한 메모리 주소로부터 16비트의 데이터를 읽어서 무부호 확장을 수행한 후 대상 레지스터에 저장한다.





Syntax

```
LDSU (%Ridx,  <imm>+), %Rdst
LDSU <label>, %Rdst
LDSU <imm>,   %Rdst
```

%Ridx 대신 현재 Stack Pointer를 이용할 수 있다.

Exceptions 이 명령에 연관된 예외 사항은 없다.

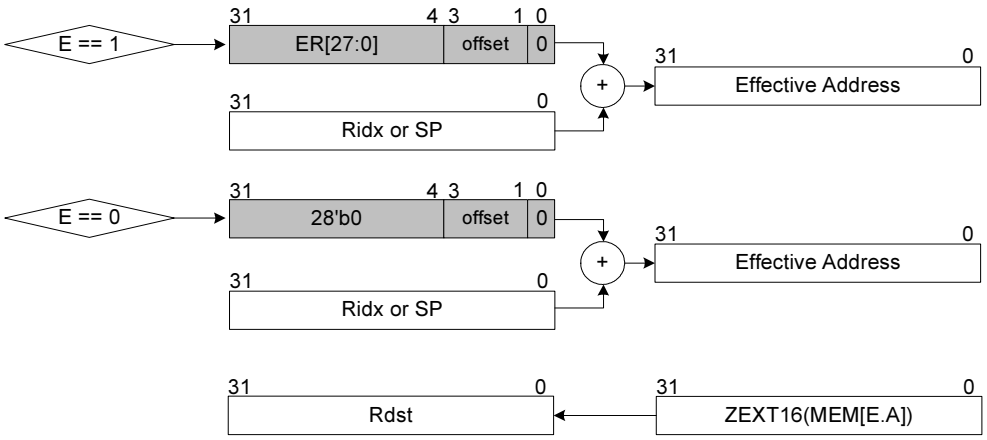
Status Modification E-Flag를 항상 clear한다.

C	Z	S	V	E
–	–	–	–	0

Operation

```
if(E_flag)
     imm = ((ER << 3) + offset) << 1
else
     imm = ZEXT4(offset << 1)
%Rdst = ZEXT16(MEM((%Ridx or %SP) + imm))
```

Usage LDSU명령은 short 단위(16비트)로 데이터를 메모리에서 읽어오는 동작을 수행한다.



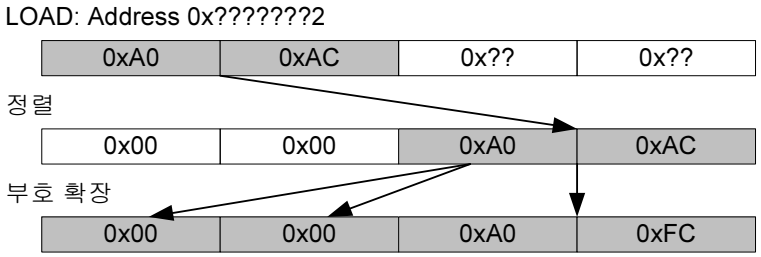

```
1  LDSU (%R2, 0x4), %R3
```

위의 경우 %R2를 인덱스 레지스터로서 사용하고, 4를 offset으로 사용하여 메모리를 접근하여 16비트 데이터를 가지고 와서 무부호 확장을 취한 후 %R3에 대입하는 동작을 보여준다. LDS 명령에서 인덱스 레지스터 이외에 스택 포인터를 이용할 수 있으며, 이 경우 아래와 같이 사용한다.

```
1  LDSU (%SP, 0x4), %R3
```

Note

- LDS는 LERI를 이용하여 offset을 확장하는 것이 가능하다. 따라서 32비트까지 offset을 지정하는 것이 가능하다. 단, 사용자가 명시적으로 LERI를 넣을 필요가 없으며, 필요한 만큼 offset의 값을 지정하는 경우 자동적으로 어셈블러에서 LERI를 생성해 주시 때문에 사용자는 어셈블러를 이용할 때 임의로 LERI를 쓰지 않아도 된다.

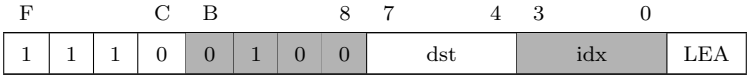


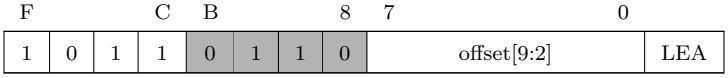
- AE32000은 32비트 데이터 버스 및 32비트 단위로 정렬되어 있는데이터를 이용하므로, 16비트 데이터를 읽어와서 그 위치를 정렬하고 부호 확장을 하여야 한다. 아래 그림은 정렬과 부호 확장 과정을 도시하고 있다.

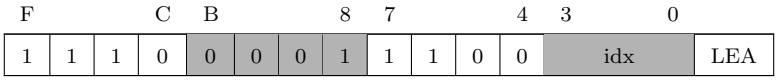
Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

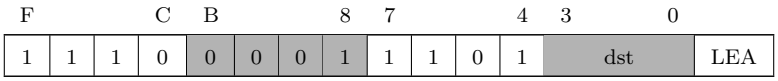
4.52 LEA - load effective address

LEA 명령은 레지스터간의 이동을 수행하며, source 및 destination 레지스터에는 스택포인터 및 범용 레지스터가 사용 될 수 있다. 이러한 속성에 의해 4 가지[1]의 명령어가 파생되며, 데이터 이동 시에 일정 값을 더하여 이동 시킬 수 있다.









Syntax

```
LEA        (%Ridx, <imm>), %Rdst
LEA(ASPO)  (%SP,   <imm>), %SP
LEA(LSEA)  (%Ridx, <imm>), %SP
LEA(LESA)  (%SP,   <imm>), %Rdst
```

%Ridx 혹은 %Rdst 대신 현재 Stack Pointer를 이용할 수 있다.

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification E-Flag를 항상 clear한다.

C	Z	S	V	E
–	–	–	–	0

[1] 내부적으로는 binary encoding 및 동작을 구분하기 위하여 ASPO(Add Stack Pointer with Offset), LESA(Load Effective Stack pointer Address), LEA(Load Effective Address) 혹은 MOV, LSEA(Load Stack pointer witch Effective Address)로 구분하여 표기하고 있으나, Mnemonic은 LEA를 사용한다.

Operation

```
소스, 목적 레지스터가 모두 스택 포인터인 경우.
if(E_flag)    imm = (ER << 4) + {offset[3:2], 2'b00}
else          imm = SEXT10(offset << 2)
그 외의 경우.
if(E_flag)    imm = ER
else          imm = 0
```

Usage LEA는 레지스터간의 데이터를 이동시킨다. 아래의 경우 범용 레지스터간의 데이터 이동을 나타내고 있다.

```
1  LEA (%R2, 0x0), %R3
```

만일 이동과 더불어 즉치 값을 더하고 싶다면 다음과 같이 구성하면 된다.

```
1  LEA (%R2, 0x8), %R3
```

스택 포인터에 대한 조작이 필요한 경우 다음과 같이 사용한다.

```
1  LEA (%sp, 0xfffff010), %sp    # %sp - 0xff0
```

또한, 스택 포인터로 혹은 스택 포인터에서부터 데이터의 이동을 할 수 있다.

```
1  LEA (%sp, 0x0), %R0
2  LEA (%R0, 0x0), %sp
```

위의 경우에도 역시 즉치 값을 이용 할 수 있다.

```
1  LEA (%sp, 0x4), %R0
2  LEA (%R0, 0x4), %sp
```

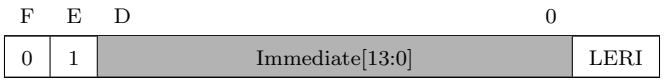
Note

- 이 명령의 즉치 값은 LERI명령을 이용하여 지정 가능하며, 32비트까지 지정 가능하다. 일반적인 어셈블리로 프로그래밍할 경우 어셈블러에서 자동으로 LERI를 삽입하므로, 사용자는 이에 대한 고려하지 않아도 된다.
- ADD와 LEA의 차이는 ADD의 경우 레지스터간의 덧셈을 수행하는 경우 머신의 상태 플래그를 변경하지만, LEA의 경우 머신의 상태 플래그를 변경하지 않는다. 또한, LEA의 경우 스택 포인터를 소스/목적 레지스터로 사용하는 것이 가능하다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.53 LERI - load extension register with immediate

LERI 명령은 즉치 값을 대상 범용 레지스터에 저장하는 동작을 수행한다.

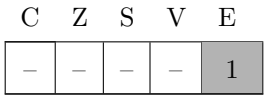


Syntax

```
LERI <imm>
```

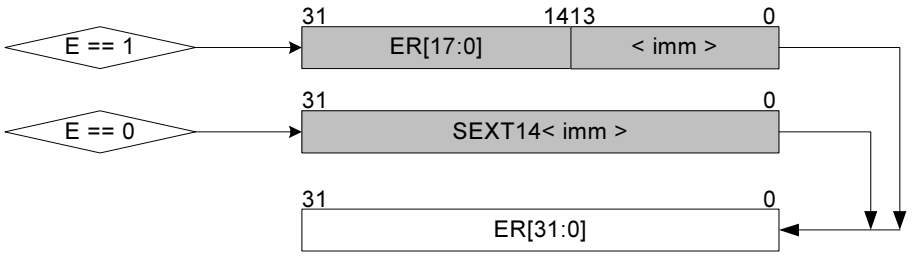
Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification E-Flag를 항상 set한다.



Operation

```
if(E_flag)
     ER = (ER << 14) + imm
else
     ER = SEXT14(imm)
```



Usage LERI는 ER에 즉치값을 대입하는 동작을 수행한다. ER에 대입된 즉치값은 E_flag를 이용하는 명령과 함께 사용되어 즉치값을 확장하는 동작을 수행한다.

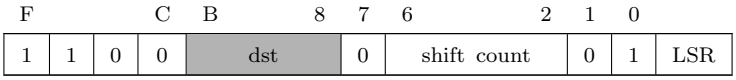
```
1  LERI 0x1ff
2  SUB 0x3, %R4     // %R4 - 0x1ff3
```

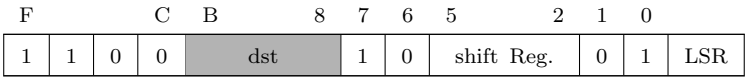
많은 경우 LERI는 1개 이상 발생하지 않으나, 즉치값의 크기가 큰 경우에는 최대 3개까지 발생할 수 있다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.54 LSR - logical shift right

LSR 명령은 Destination Register값을 shift count만큼 우측으로 쉬프트하는 명령이다. 쉬프트 될 때 MSB값은 '0'이 채워진다. 쉬프트 카운트로는 즉치갑과 레지스터의 값이 가능하다. 레지스터 값을 이용할 경우에는 하위 5비트만 사용된다.



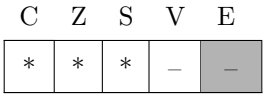


Syntax

```
LSR  <Shift_Count>, %Rdst
LSR  %Rsft, %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 명령어 수행 결과에 따라 C, Z, S-flag이 변화한다. C-Flag 값은 shifted-out되어 LSB로 입력되는 bit가 Carry 값에 반영된다. shift되는 크기가 '0'인 경우 C-Flag는 '0'을 갖는다.

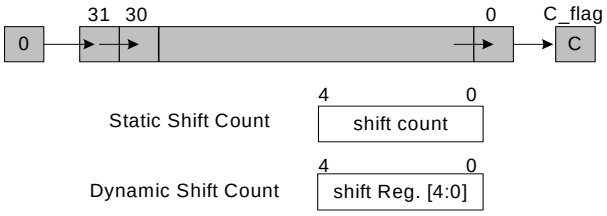


Operation

```
Static Shift
%Rdst = %Rdst >> <shift_count>
Dynamic Shift
%Rdst = %Rdst >> (%Rsft & 0x1f)
```

상위 비트들은 '0'으로 채워진다.

Usage LSR 연산은 unsigned number를 right shift할 때 사용된다.

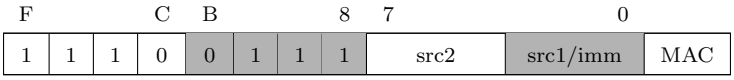



```
1  LSR 0x1, %R2
2  LSR %R0, %R2
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.55 MAC - multiply and add

MAC 명령은 두 소스의 값을 곱한 후 이 값을 누산 레지스터로서 사용되는 %MH:%ML과 더하는 역할을 수행한다.



Syntax

```
MAC  %Rsrc1, %Rsrc2
MAC  <imm>, %Rsrc2
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification E-Flag를 항상 clear한다.

C	Z	S	V	E
–	–	–	–	0

Operation

```
if(E_flag)
     %MH:%ML = (%Rsrc2 * (ER << 4 + imm[3:0])) + %MH:%ML
else
     %MH:%ML = (%Rsrc2 * %Rsrc1)) + %MH:%ML
```

Usage 사용 가능한 피연산자로는 레지스터 혹은 즉치값이 가능하며. 피연산자로서 즉치값을 사용하고자 할 때는 어떠한 경우에도 LERI를 이용해야 한다. 다음 예는 곱셈을 통하여 %MH, %ML을 지정하고, MAC을 이용하는 예를 보여주고 있다.

```
1  MUL %R3, %R4
2  MAC %R6, %R7
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C-DSP version에서 지원된다.

4.56 MACB - SIMD MAC BYTE

MACB 명령은 Byte단위의 SIMD MAC을 수행한다. 이때 수행되는 단위 MAC은 8b x 8b + 24b = 24b를 수행하게 되며, 동시에 4개의 MAC이 수행된다.

F			C	B			8	7	4	3	0
1	1	1	1	0	0	0	0	src2		src1/imm	MACB

Syntax

```
MACB  %Rsrc1, %Rsrc2
MACB  <imm>, %Rsrc2
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification E-Flag를 항상 clear한다.

C	Z	S	V	E
–	–	–	–	0

Operation (3.7.1 참조)

```
ACCB0 = {MRE[7:0], ML[15:0]}
ACCB1 = {MRE[15:8], ML[31:16]}
ACCB2 = {MRE[23:16], MH[15:0]}
ACCB3 = {MRE[31:24], MH[31:16]}
if(E_flag)
     OP = ((ER << 4) + imm)
else
     OP = %Rsrc1
ACCB0 = OP[7:0] * %Rsrc2[7:0] + ACCB0
ACCB1 = OP[15:8] * %Rsrc2[15:8] + ACCB1
ACCB2 = OP[23:16] * %Rsrc2[23:16] + ACCB2
ACCB3 = OP[31:24] * %Rsrc2[31:24] + ACCB3
```

Usage 사용 가능한 피연산자로는 레지스터 혹은 즉치값이 가능하며. 피연산자로서 즉치값을 사용하고자 할 때는 어떠한 경우에도 LERI를 이용해야 한다. 다음 예에서는 MRE

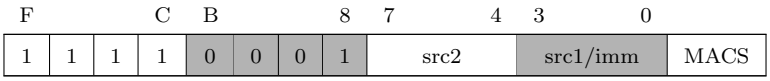
에 특정 값을 저장하고 이를 이용하여 MACB연산을 수행하는 예를 보여준다.

```
1  MTMRE %R3
2  MACB %R1, %R2
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C-DSP version에서 지원된다.

4.57 MACS - SIMD MAC SHORT

MACS 명령은 Short단위의 SIMD MAC을 수행한다. 이때 수행되는 단위 MAC은 16b x 16b + 48b = 48b를 수행하게 되며, 동시에 2개의 MAC이 수행된다.



Syntax

```
MACS  %Rsrc1, %Rsrc2
MACS  <imm>, %Rsrc2
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification E-Flag를 항상 clear한다.

C	Z	S	V	E
–	–	–	–	0

Operation (3.7.1 참조)

```
ACC0 = {MRE[15:0], ML}
ACC1 = {MRE[31:16], MH}
if(E_flag)
     OP = ((ER << 4) + imm)
else
     OP = %Rsrc1
ACC0 = OP[15:0] * %Rsrc2[15:0] + ACC0
ACC1 = OP[31:16] * %Rsrc2[31:16] + ACC1
```

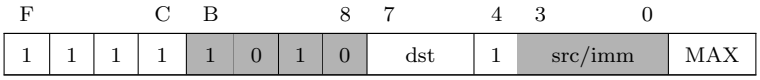
Usage 사용 가능한 피연산자로는 레지스터 혹은 즉치값이 가능하며. 피연산자로서 즉치값을 사용하고자 할 때는 어떠한 경우에도 LERI를 이용해야 한다. 다음 예에서는 MRE에 특정 값을 저장하고 이를 이용하여 MACS연산을 수행하는 예를 보여준다.

```
1  MTMRE %R3
2  MACS %R1, %R2
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C-DSP version에서 지원된다.

4.58 MAX - Maximum

MAX 명령은 두 레지스터값(혹은 즉치값)의 비교를 통하여 둘 중 최대값을 출력한다.

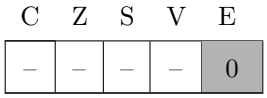


Syntax

```
MAX  %Rsrc, %Rdst
MAX  <imm>, %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification E-Flag를 항상 clear한다.



Operation

```
if(E_flag)
     imm = ((ER << 4) + imm)
else
     imm = %Rsrc1
if(%Rdst > imm)
     %Rdst = %Rdst
else
     %Rdst = imm
```

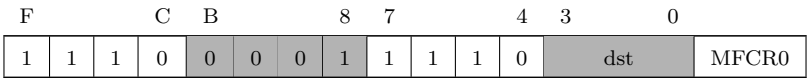
Usage 사용 가능한 피연산자로는 레지스터 혹은 즉치값이 가능하며. 피연산자로서 즉치값을 사용하고자 할 때는 어떠한 경우에도 LERI를 이용해야 한다. 다음은 MAX 명령을 통해 큰 값을 취하는 명령의 예를 보여준다. 목적 레지스터는 0 - 7번 레지스터만을 사용한다.

```
1   MAX %R1, %R2
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C-DSP version에서 지원된다.

4.59 MFCR0 - move from CR0

MFCR0 는 CR0에 저장되어 있는 값을 범용 레지스터로 이동하는 역할을 수행한다.



Syntax

```
MFCR0  %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 변화 없음

C	Z	S	V	E
–	–	–	–	–

Operation

```
%Rdst = %CR0
```

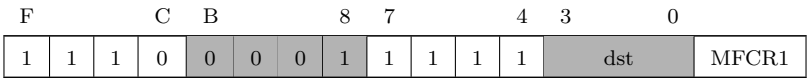
Usage 다음 예는 CR0의 값을 확인하기 위하여 %R3으로 CR0의 값을 가져오는 동작을 보여준다.

```
1  MFCR0 %R3
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.60 MFCR1 - move from CR1

MFCR1 은 CR1에 저장되어 있는 값을 범용 레지스터로 이동하는 역할을 수행한다.



Syntax

```
MFCR1  %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 변화 없음

C	Z	S	V	E
–	–	–	–	–

Operation

```
%Rdst = %CR1
```

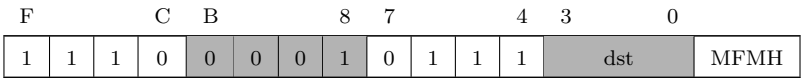
Usage 다음 예는 CR1의 값을 확인하기 위하여 %R3으로 CR1의 값을 가져오는 동작을 보여준다.

```
1  MFCR1 %R3
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.61 MFMH - move from MH

MFMH 는 %MH에 저장되어 있는 값을 범용 레지스터로 이동하는 역할을 수행한다.

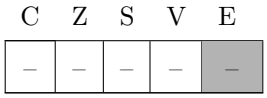


Syntax

```
MFMH  %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 변화 없음



Operation

```
%Rdst = %MH
```

Usage 다음 예는 %R3, %R4을 곱하고 그 결과를 다시 %R3와 %R4에 저장하는 동작을 보여주고 있다. 곱셈 연산의 경우 하위 32비트를 %Rdst로 저장하는 동작을 수행하므로, 전체 결과를 원하는 경우 곱셈을 수행한 후에 반드시 MFMH를 수행하여야 한다.

```
1  MUL %R3, %R4
2  MFMH %R3      // %R3:%R4 = %R3 * %R4
```

Note

- MAC연산에서의 결과는 따로 범용 레지스터로 저장되지 않으므로, 이 결과를 저장하기 위해서는 PUSH %MH,%ML을 이용하거나, MFML, MFMH명령을 이용하여 범용 레지스터로 이동시켜야 한다.
- 곱셈의 경우 하위 결과(ML)가 바로 목적 레지스터로 저장되므로, 모든 결과를 얻기 위해서는 MFMH가 필요하다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.62 MFML - move from ML

MFML 는 %ML에 저장되어 있는 값을 범용 레지스터로 이동하는 역할을 수행한다.

F			C	B			8	7			4	3	0
1	1	1	0	0	0	0	1	0	1	1	1	dst	MFML

Syntax

```
MFML  %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 변화 없음

C	Z	S	V	E
–	–	–	–	–

Operation

```
%Rdst = %ML
```

Usage 다음 예는 %R3, %R4을 곱하고 그 결과를 다시 %R5와 %R6과 MAC을 수행한 후 하위 결과를 %R11에 보내는 동작을 수행한다.

```
1  MUL %R3, %R4
2  MAC %R5, %R6
3  MFML %R11
```

Note

- MAC연산에서의 결과는 따로 범용 레지스터로 저장되지 않으므로, 이 결과를 저장하기 위해서는 PUSH %MH,%ML을 이용하거나, MFML, MFMH명령을 이용하여 범용 레지스터로 이동시켜야 한다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.63 MFMRE - move from MRE

MFMRE 는 MRE에 저장되어 있는 값을 범용 레지스터로 이동하는 역할을 수행한다.

F			C	B			8	7			4	3	0
1	1	1	0	0	0	0	0	0	0	1	0	dst	MFMRE

Syntax

```
MFMRE  %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 변화 없음

C	Z	S	V	E
–	–	–	–	–

Operation

```
%Rdst = %MRE
```

Usage 다음 예는 MRE의 값을 확인하기 위하여 %R3으로 MRE의 값을 가져오는 동작을 보여준다.

```
1  MFMRE %R3
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C, AE32000C-DSP version 에서 지원된다.

4.64 MIN - Minimum

MIN 명령은 두 레지스터값(혹은 즉치값)의 비교를 통하여 둘 중 최소값을 출력한다.

F			C	B			8	7	4	3	0
1	1	1	1	1	0	1	0	dst	0	src/imm	MIN

Syntax

```
MIN %Rsrc, %Rdst
MIN <imm>, %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification E-Flag를 항상 clear한다.

C	Z	S	V	E
–	–	–	–	0

Operation

```
if(E_flag)
     imm = ((ER << 4) + imm)
else
     imm = %Rsrc1
if(%Rdst > imm)
     %Rdst = imm
else
     %Rdst = %Rdst
```

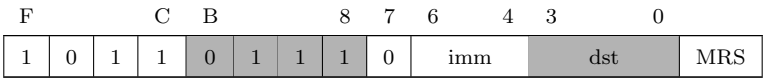
Usage 사용 가능한 피연산자로는 레지스터 혹은 즉치값이 가능하며. 피연산자로서 즉치값을 사용하고자 할 때는 어떠한 경우에도 LERI를 이용해야 한다. 다음은 MIN 명령을 통해 작은 값을 취하는 명령의 예를 보여준다. 목적 레지스터는 0 - 7번의 레지스터만을 사용한다.

```
1  MIN %R1, %R2
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C-DSP version에서 지원된다.

4.65 MRS - multiply result shift and extraction

MRS 는 MR(Multiply Result Register) Shift의 약자로서, fixed point multiplication에서 결과를 Shift하여 보정한 후 하나의 레지스터로 전달하는 명령으로서, fixed point 연산을 지원하는 명령이다.

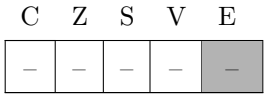


Syntax

```
MRS  <imm>, %Rdst
```

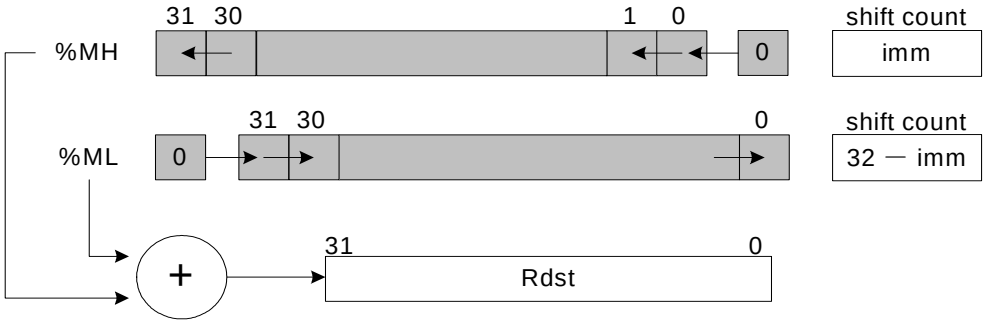
Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 변화 없음



Operation

```
%Rdst = (%MH << imm) + (%ML >> (32 - imm))
```



Usage 다음은 fixed point multiplication에서 결과를 3만큼 Shift하여 보정한 후 %R4로 전달하는 예를 보여준다. MRS 명령의 경우, MTML 혹은 MTMH 직후에 사용하는 것을 금지하고 있다. 또한 LERI 명령어를 사용하지 않기 때문에 사용할 수 있는 즉치값은 0-7로 제한된다.

```
1 MUL %R0, %R2
2 MRS 0x3, %R4
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.66 MSOPB - SIMD MAC BYTE

MSOPB 명령은 Byte단위의 Sum of product 연산(Signed)을 수행한다. 수행되는 연산은 (8b x 8b) + (8b x 8b) + (8b x 8b) + (8b x 8b) + 32b = 32b의 형태이다.

Syntax

```
MSOPB  %Rsrc1, %Rsrc2
MSOPB  <imm>, %Rsrc2
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification E-Flag를 항상 clear한다.

C	Z	S	V	E
–	–	–	–	0

Operation

```
if(E_flag)
     OP = ((ER << 4) + imm)
else
     OP = %Rsrc1
SUM0 = OP[7:0] * %Rsrc2[7:0]
SUM1 = OP[15:8] * %Rsrc2[15:8]
SUM2 = OP[23:16] * %Rsrc2[23:16]
SUM3 = OP[31:24] * %Rsrc2[31:24]

%MH = SEXT(%ML[31])
%ML = SUM0 + SUM1 + SUM2 + SUM3 + %ML
```

Usage 사용 가능한 피연산자로는 레지스터 혹은 즉치값이 가능하며. 피연산자로서 즉치값을 사용하고자 할 때는 어떠한 경우에도 LERI를 이용해야 한다. 다음 예에서는 8비트 단위로 Sum of Product를 구하기위한 명령어의 예를 보여준다.

```
1  MSOPB %R1, %R2
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C-DSP version에서 지원된다.

4.67 MSOPS - SIMD MAC SHORT

MSOPS 명령은 Short단위의 Sum of product 연산(Signed)을 수행한다. 수행되는 연산은 (16b x 16b) + (16b x 16b) + 48b = 48b의 형태이다.

F			C	B			8	7	4	3	0
1	1	1	1	0	0	1	1	src2		src1/imm	MSOPS

Syntax

```
MSOPS  %Rsrc1, %Rsrc2
MSOPS  <imm>, %Rsrc2
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification E-Flag를 항상 clear한다.

C	Z	S	V	E
–	–	–	–	0

Operation

```
if(E_flag)
     OP = ((ER << 4) + imm)
else
     OP = %Rsrc1
SUM0 = OP[15:0] * %Rsrc2[15:0]
SUM1 = OP[31:16] * %Rsrc2[31:16]
%MH[15:0]:%ML = SUM0 + SUM1 + %MH[15:0]:%ML
%MH[31:16] = SEXT(%MH[15:0])
```

Usage 사용 가능한 피연산자로는 레지스터 혹은 즉치값이 가능하며. 피연산자로서 즉치값을 사용하고자 할 때는 어떠한 경우에도 LERI를 이용해야 한다. 다음 예에서는 16비트 단위로 Sum of Product를 구하기위한 명령어의 예를 보여준다.

```
1  MSOPB %R1, %R2
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C-DSP version에서 지원된다.

4.68 MTCR0 - move to CR0

MTCR0 는 범용 레지스터의 값을 CR0로 이동하는 역할을 수행한다.

F			C	B			8	7			4	3	0
1	1	1	0	0	0	0	1	1	0	0	0	src	MTCR0

Syntax

```
MTCR0  %Rsrc
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 변화 없음

C	Z	S	V	E
–	–	–	–	–

Operation

```
%CR0 = %src
```

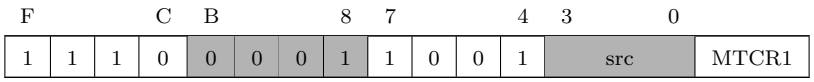
Usage 다음 예는 CR0의 값을 세팅하기 위하여 %R2로부터 CR0의 값을 넣는 동작을 보여준다.

```
1  MTCR0 %R2
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.69 MTCR1 - move to CR1

MTCR1 는 범용 레지스터의 값을 CR1로 이동하는 역할을 수행한다.

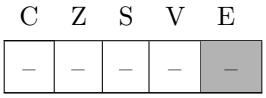


Syntax

```
MTCR1  %Rsrc
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 변화 없음



Operation

```
%CR1 = %Rsrc
```

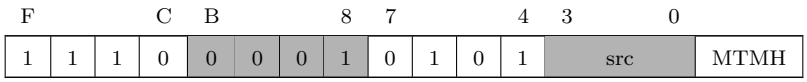
Usage 다음 예는 CR1의 값을 세팅하기 위하여 %R2로부터 CR1의 값을 넣는 동작을 보여준다.

```
1  MTCR1 %R2
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.70 MTMH - move to MH

MTMH 는 소스 레지스터에 저장되어 있는 값을 %MH로 보내는 동작을 수행한다.



Syntax

```
MTMH  %Rsrc
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 변화 없음

C	Z	S	V	E
–	–	–	–	–

Operation

```
%MH = %Rsrc
```

Usage 다음 예는 MTMH, MTML을 통하여 %MH:%ML의 값을 설정하고 MAC을 수행하는 예를 보여준다.

```
1  MTMH %R2
2  MTML %R1
3  MAC  %R5, %R6
```

Note

- MTMH는 MAC연산 이전에 누산 레지스터를 초기화하기 위하여 사용된다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.71 MTML - move to ML

MTML 는 소스 레지스터에 저장되어 있는 값을 %ML로 보내는 동작을 수행한다.

F			C	B			8	7			4	3	0
1	1	1	0	0	0	0	1	0	1	0	0	src	MTML

Syntax

```
MTML  %Rsrc
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 변화 없음

C	Z	S	V	E
–	–	–	–	–

Operation

```
%ML = %Rsrc
```

Usage 다음 예는 MTMH, MTML을 통하여 %MH:%ML의 값을 설정하고 MAC을 수행하는 예를 보여준다.

```
1  MTMH %R2
2  MTML %R1
3  MAC  %R5, %R6
```

Note

- MTML는 MAC연산 이전에 누산 레지스터를 초기화하기 위하여 사용된다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.72 MTMRE - move to MRE

MTMRE 는 범용 레지스터의 값을 MRE로 이동하는 역할을 수행한다.

F			C	B			8	7			4	3	0	
1	1	1	0	0	0	0	0	0	0	1	1	src		MTMRE

Syntax

```
MTMRE  %Rsrc
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 변화 없음

C	Z	S	V	E
–	–	–	–	–

Operation

```
%MRE = %Rsrc
```

Usage 다음 예는 MRE의 값을 세팅하기 위하여 MRE의 값을 %R4로 가져오는 동작을 보여준다.

```
1  MTMRE %R4
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C-DSP version에서 지원된다.

4.73 MUL - signed multiply

MUL 은 두 값을 곱하여 그 결과를 %MH:%ML에 저장하고, 목적 레지스터로 이중 %ML의 값을 전달하는 동작을 수행한다. 단, 두 피 연산자를 signed number로 인식하여 곱셈을 수행한다.

F			C	B			8	7	5	4	3	0
1	1	1	0	0	1	1	0	dst		0	src/imm	MUL

Syntax

```
MUL  %Rsrc, %Rdst
MUL  <imm>, %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification E-Flag를 항상 clear한다.

C	Z	S	V	E
–	–	–	–	0

Operation

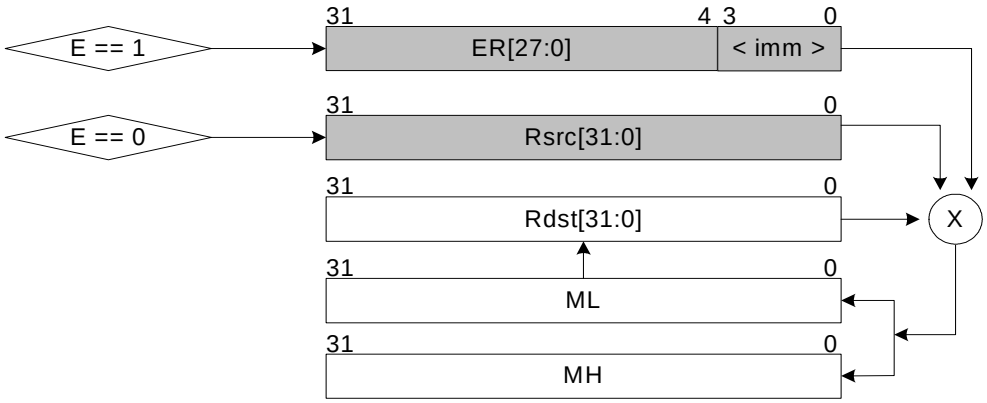
```
if(E_flag)
     %MH:%ML = %Rdst * (ER << 4 + imm[3:0])
else
     %MH:%ML = %Rdst * %Rsrc
%Rdst = %ML
```

Usage 다음 예는 두 값의 곱셈을 보이고 있다.

```
1  MUL  %R5, %R6
```

Note

- 사용 가능한 피연산자로는 레지스터 혹은 즉치값이 가능하며. 피연산자로서 즉치값을 사용하고자 할 때는 어떠한 경우에도 LERI를 이용해야 한다.



- 곱셈 연산에서 목적 레지스터의 주소는 짝수[2] 레지스터만이가능하다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

[2] 짝수 레지스터 아닌 경우 최하위 비트가 0으로 변환되어 처리된다.

4.74 MULU - unsigned multiply

MULU 은 두 값을 곱하여 그 결과를 %MH:%ML에 저장하고, 목적 레지스터로 이중 %ML의 값을 전달하는 동작을 수행한다. 단, 두 피 연산자를 unsigned number로 인식하여 곱셈을 수행한다.

F			C	B			8	7	5	4	3	0
1	1	1	0	0	1	1	0	dst		1	src/imm	MULU

Syntax

```
MULU  %Rsrc, %Rdst
MULU  <imm>+, %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification E-Flag를 항상 clear한다.

C	Z	S	V	E
–	–	–	–	0

Operation

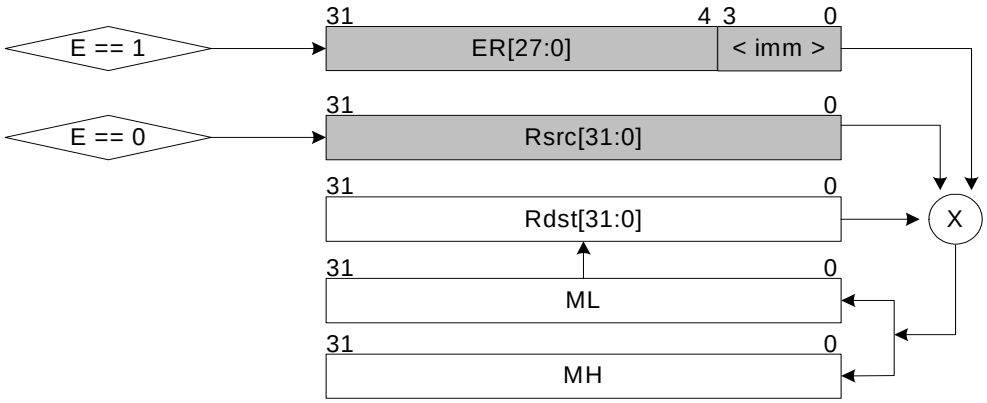
```
if(E_flag)
     %MH:%ML = %Rdst * (ER << 4 + imm[3:0])
else
     %MH:%ML = %Rdst * %Rsrc
%Rdst = %ML
```

Usage 다음 예는 두 값의 곱셈을 보이고 있다.

```
1  MULU  %R5, %R6
```

Note

- 사용 가능한 피연산자로는 레지스터 혹은 즉치값이 가능하며. 피연산자로서 즉치값을 사용하고자 할 때는 어떠한 경우에도 LERI를 이용해야 한다.

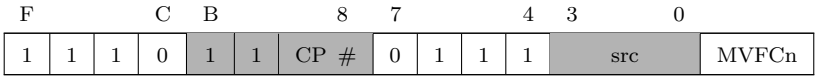


- 곱셈 연산에서 목적 레지스터의 주소는 짝수 레지스터만이 가능하다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.75 MVFC - move from coprocessor

MVFC 는 보조 프로세서의 지정 레지스터로부터 데이터를 읽어와서 %R0에 저장하는 동작을 수행한다.

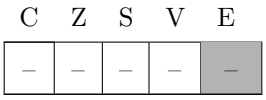


Syntax

```
MVFC  <cp_no>, %Rsrc
```

Exceptions – cpint: 사용자 모드에서 관리자 전용 보조 프로세서에 접근한 경우 인터럽트가 발생한다.

Status Modification 변화 없음



Operation

```
%R0 = %Rsrc@CP<cp_no>
```

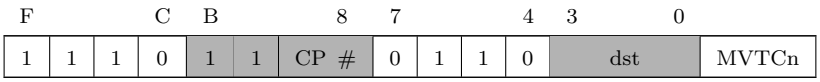
Usage 다음 예는 1번 보조 프로세서의 6번 레지스터로부터 값을 읽어와 AE32000의 %R0로 가지고 오는 예를 보여주고 있다.

```
1  MVFC 0x1, %R6
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.76 MVTC - move to coprocessor

MVTC 는 %R0의 값을 지정 보조 프로세서의 지정 레지스터로 쓰는 동작을 수행한다.

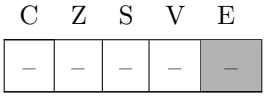


Syntax

```
MVTC  <cp_no>, %Rsrc
```

Exceptions – cpint: 사용자 모드에서 관리자 전용 보조 프로세서에 접근한 경우 인터럽트가 발생한다.

Status Modification 변화 없음



Operation

```
%Rdst@CP<cp_no> = %R0
```

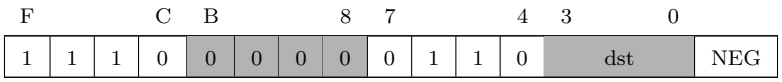
Usage 다음 예는 1번 보조 프로세서의 6번 레지스터로 AE32000의 0번 레지스터의 값을 보낸다.

```
1  MVTC 0x1, %R6
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.77 NEG - negate

NEG 명령은 해당 레지스터를 음수로 만드는 동작을 수행한다.

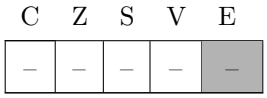


Syntax

```
NEG  %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 변화 없음



Operation

```
%Rdst = -%2Rdst
```

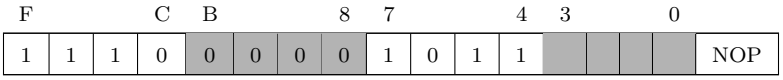
Usage EISC에서는 2's complement의 음수를 취하기 위해서는 subtract를 해야 하며, 이 때 반드시 LERI가 추가되어야 한다. 다음과 같이 NEG를 사용하면 LERI의 사용없이 음수로 만들 수 있다.

```
1  NEG %R2
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.78 NOP - no operation

NOP 는 아무런 동작을 수행하지 않는 명령으로서 프로세서의 파이프라인을 비두거나, 지연 분기에서 적절한 명령을 찾지 못한 경우 filler로서 사용된다

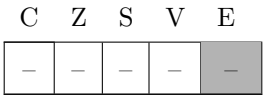


Syntax

```
NOP
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 변화 없음



Operation

```
None
```

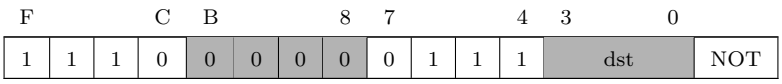
Usage

```
1 ADD %R1, %R0
2 NOP
3 MVTC 0x1, %R1
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.79 NOT - logical inversion

NOT 는 대상 레지스터의 값에 대하여 logical inversion을 취한다.



Syntax

```
NOT  %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 명령어 수행 결과에 따라 Z, S-flag이 변화한다.

C	Z	S	V	E
–	*	*	–	–

Operation

```
%Rdst = !%Rdst
```

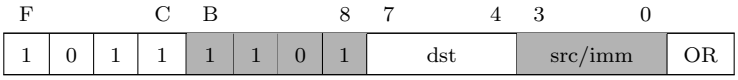
Usage NOT연산은 대상 레지스터를 logical inversion을 취한다.

```
1  NOT %R0
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.80 OR - bitwise OR

OR 는 목적 레지스터 또는 즉치 값과 대상 레지스터의 값을 비트 단위로 OR연산을 취한다.



Syntax

```
OR   %Rsrc,  %Rdst
OR   <imm>,  %Rdst
```

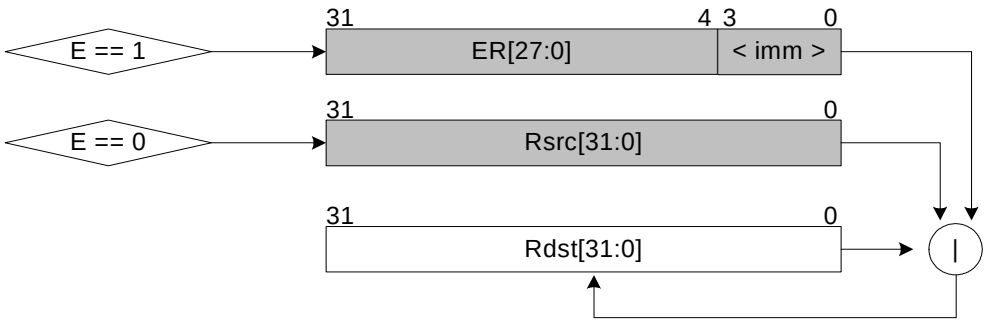
Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 명령어 수행 결과에 따라 Z, S-flag이 변화하며, E-Flag은 항상 clear된다.

C	Z	S	V	E
–	*	*	–	0

Operation

```
if(E_flag)
     %Rdst = %Rdst | (ER << 4 + imm)
else
     %Rdst = %Rdst | %Rsrc
```



Usage OR연산은 두 피연산자간에 bitwise OR연산을 수행하기 위하여 사용된다.

```
OR %R0, %R2
```

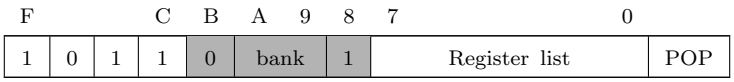
즉치값을 사용하기 위해서는 반드시 LERI를 이용해야 한다.

```
LERI 0x0
OR 0x1, %R2
...
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.81 POP - pop list

POP 명령은 list에 있는 레지스터들로 POP하는 동작을 수행한다.



F			C	B	A	9	8	7 ... 0	
1	0	1	1	0	bank		1	Register list	POP

Syntax

```
POP   <reg_list>
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 변화 없음

C	Z	S	V	E
–	–	–	–	–

Operation

```
for(<reg_list> == empty)
     POP(reg_list_entry)
```

Usage PUSH list연산은 다음의 예에서 보이는 바와 같이 지정할 수 있으며, 두 명령은 같은 동작을 수행한다.

```
1  // 4, 3, 2, 1, 0의 순서로 push된다.
2  PUSH %R0 - %R4
3  // 어셈블리 상에서의 레지스터 순서와 무관하게 4, 3, 2, 1, 0의 순서로 push된다.
4  PUSH %R0, %R4, %R2, %R3, %R1
```

서로 다른 뱅크의 값을 POP하려면 서로 다른 뱅크 만큼 PUSH명령이 요구된다. (최대 3개)

```
1  // 7, 6, 5, 4, 3, 0의 순서로 push된다.
2  PUSH %R0, %R3 - %R7
```

```
3  // 15, 13, 12, 11, 10의 순서로 push된다.
4  PUSH %R10 - %R13, %R15
5  // SR, LR, MH, ML의 순서로 push된다.
6  // register list의 MSB 부터 차례로 push된다.
7  PUSH %SR, %LR, %MH, %ML
```

Usage POP list연산은 다음의 예에서 보이는 바와 같이 지정할 수 있으며, 두 명령은 같은 동작을 수행한다.

```
1  // 0, 1, 2, 3, 4의 순서로 pop된다
2  POP %R0 - %R4
3  // 어셈블리 상에서의 레지스터 순서와 무관하게 0, 1, 2, 3, 4의 순서로 pop된다.
4  POP %R0, %R4, %R2, %R3, %R1
```

서로 다른 뱅크의 값을 push하려면 서로 다른 뱅크 만큼 POP명령이 요구된다. (최대 3개 - 표 4.171 참조)

```
1  // 0, 3, 4, 5, 6, 7의 순서로 pop된다.
2  POP %R0, %R3 - %R7
3  // 10, 11, 12, 13, 15의 순서로 pop된다.
4  POP %R10 - %R13, %R15
5  // ML, MH, LR, SR의 순서로 pop된다.
6  // register list의 LSB 부터 차례로 pop된다.
7  POP %SR, %LR, %MH, %ML
```

Note

- POP PC의 경우는 허용되나 PUSH PC는 소프트웨어적으로 허용되지 않는다.
- POP 되는 순서는 번호가 작은 레지스터부터 stack의 값을 읽어 들인다.

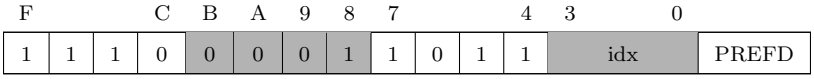
Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

Register list								
bit position	7	6	5	4	3	2	1	0
Bank 1	R7	R6	R5	R4	R3	R2	R1	R0
Bank 2	R15	R14	R13	R12	R11	R10	R9	R8
Bank 3	SR	PC	LR	ER	MH	ML	CR1	CR0

Table 4.171: Bank에 따른 POP list

4.82 PREFD - data cache prefetch

PREFD 는 인덱스 레지스터가 지정하는 부분을 데이터 캐쉬로 미리 가져 올 수 있도록 하는 명령으로서, 데이터 접근 시간을 줄이는데 도움을 준다.



Syntax

```
PREFD   %Ridx
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 변화 없음

C	Z	S	V	E
–	–	–	–	–

Operation

```
None
```

* 캐쉬에 영향을 미치고, 프로세서는 아무런 동작을 수행하지 않는다.

Usage PREFD는 다음과 같이 지정한다.

```
1  PREFD %R0
```

Note

- PREFD는 data prefetch로서, matrix 연산이나 stream데이터를 이용할 경우 도음이 될 수 있다.
- 프로세서는 캐쉬가 있는 경우에 캐쉬 컨트롤러에 해당 정보만을 전해 한 다른 동작은 수행하지 않는다.
- 구현이 되지 않은 경우 NOP처리 되며, Unimplemented interrupt를 발생시키지는 않는다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B version에서만 지원된다.

4.83 PREFI - instruction cache prefetch

PREFI 는 오프셋이 지정하는 곳에 있는 명령어 주소에 존재하는 부분을 명령어 캐쉬로 미리 가져 올 수 있도록 하는 명령으로서, 명령어 접근 시간을 줄이는데 도움을 준다.

F			C	B	A	9	8	7 ... 0	
1	1	1	0	0	0	1	0	offset[8:1]	PREFI

Syntax

```
PREFI   <offset>
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification E-Flag를 항상 clear한다.

C	Z	S	V	E
–	–	–	–	0

Operation

```
None
```

* 캐쉬에 영향을 미치고, 프로세서는 아무런 동작을 수행하지 않는다.

Usage PREFI는 다음과 같이 지정한다.

```
1   PREFI 0x0ff0
```

Note

- PREFI는 instruction prefetch로서, 분기 이전에 분기 목적 주소를 명령어 캐쉬로 보냄으로서 미리 가져 올 수 있도록 하여 분기시 캐쉬 미스에 따른 지연을 줄일 수 있다.
- offset의 값은 LERI에 의하여 확장 가능하나, 어셈블러에 의하여 자동적으로 이루어지므로 사용자가 이에 대하여 고려할 필요는 없다. offset의 길이가 9비트를 넘어가는 경우 어셈블러에 의하여 LERI가 자동으로 추가된다. offset의 생성은 다음과 같다.

```
1  if(E_flag)
2       imm = ER << 9 + {Offset[8:1],1'b0}
3  else
4       imm = SEXT9({Offset[8:1],1'b0))
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B version에서만 지원된다.

4.84 PUSH - push list

PUSH 명령은 list에 있는 레지스터들을 PUSH하는 동작을 수행한다.

F			C	B	A	9	8	7	0	
1	0	1	1	0	bank		0	Register list		PUSH

Syntax

```
PUSH   <reg_list>
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 변화 없음

C	Z	S	V	E
–	–	–	–	–

Operation

```
for(<reg_list> == empty)
     PUSH(reg_list_entry)
```

Usage PUSH list연산은 다음의 예에서 보이는 바와 같이 지정할 수 있으며, 두 명령은 같은 동작을 수행한다.

```
1  // 4, 3, 2, 1, 0의 순서로 push된다.
2  PUSH %R0 - %R4
3  // 어셈블리 상에서의 레지스터 순서와 무관하게 4, 3, 2, 1, 0의 순서로 push된다.
4  PUSH %R0, %R4, %R2, %R3, %R1
```

서로 다른 뱅크의 값을 POP하려면 서로 다른 뱅크 만큼 PUSH명령이 요구된다. (최대 3개)

```
1  // 7, 6, 5, 4, 3, 0의 순서로 push된다.
2  PUSH %R0, %R3 - %R7
```

```
3  // 15, 13, 12, 11, 10의 순서로 push된다.
4  PUSH %R10 - %R13, %R15
5  // SR, LR, MH, ML의 순서로 push된다.
6  // register list의 MSB 부터 차례로 push된다.
7  PUSH %SR, %LR, %MH, %ML
```

Note

- PUSH list의 경우 레지스터 list에 존재하는 레지스터들을 스택 영역으로부터 쓰기 동작을 수행한다.
- PUSH list는 3개의 bank로서 구성되어 있으며, 첫 번째 뱅크는 범용 레지스터 0번에서 7번까지, 두 번째 뱅크는 범용 레지스터 8번에서 15번까지, 세번째 뱅크는 특수 목적 레지스터들을 지정한다. PUSH list를 사용할 때 서로 다른 뱅크의 레지스터를 동시에 PUSH할 수는 없다. 표 4.178 참조.
- PUSH 동작에 있어서 스택 포인터는 현재 동작 상태에 해당하는 스택 포인터가 선택되며, 스택 포인터가 스택의 최상위 주소를 가르키고 있으므로, 스택의 값을 감소시킨 후에 이 주소를 이용하여 메모리로 write를 수행한다.
- PUSH list는 구현에 있어서 복잡하나, 코드 밀도를 높이는데 도움을 준다.
- 특수 목적 레지스터를 PUSH할 경우 ER의 경우 사용자 모드에서 PUSH하여도 단지 스택 포인터의 값을 감소시키기만 하며, SR의 관리자 전용 상태들의 경우 사용자 모드에서 PUSH하여도 해당 비트의 값은 '0'으로 write된다.
- PUSH PC[3]는 허용되지 않는다.
- PUSH가 되는 순서는 register list의 MSB부터 stack에 쌓인다.

Register list								
bit position	7	6	5	4	3	2	1	0
Bank 1	R7	R6	R5	R4	R3	R2	R1	R0
Bank 2	R15	R14	R13	R12	R11	R10	R9	R8
Bank 3	SR	PC	LR	ER	MH	ML	CR1	CR0

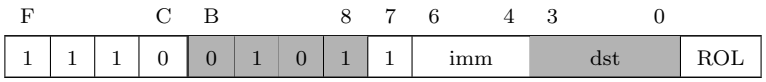
Table 4.178: Bank에 따른 PUSH list

[3]소프트웨어적으로는 PUSH PC를 할 수 없다. 단 POP PC의 경우는 허용 된다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.85 ROL - rotete left

ROL 명령은 Rotate Left 동작을 수행한다.

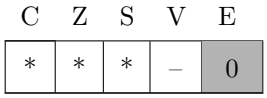


Syntax

```
ROL  <imm>, %Rdst
```

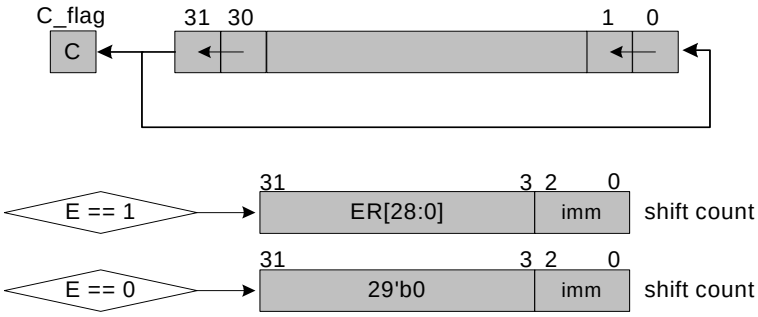
Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 명령어 수행 결과에 따라 C, Z, S-flag이 변화하며, E-Flag은 항상 clear된다. C-Flag 값은 shifted-out되어 LSB로 입력되는 bit가 Carry 값에 반영된다.



Operation

```
if(E_flag)
     imm = (ER << 3) + imm
else
     imm = {29'b0, imm}
%Rdst = Rotate Left(%Rdst << imm)
```



Usage ROL 명령은 다은과 같이 사용하며 어떤 경우에도 하위 5bit만[4] 유효하다.

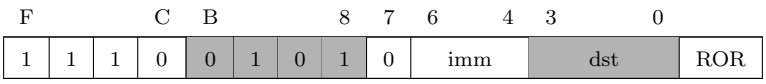
```
1  ROL 0x4, %R2
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

[4]즉 즉치값은 ER 값이 적용되더라도 0 에서 31 까지의 값만이 유효하며, 이보다 큰 값일 경우 modulo 32 연산의 결과 값이 적용

4.86 ROR - rotete right

ROR 명령은 Rotate Right 동작을 수행한다.

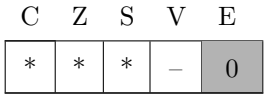


Syntax

```
ROR  <imm>, %Rdst
```

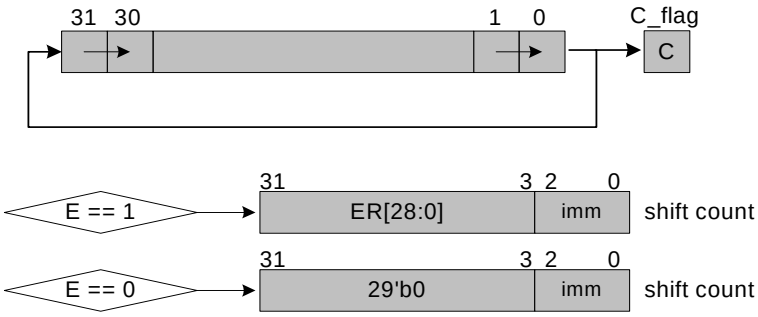
Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 명령어 수행 결과에 따라 C, Z, S-flag이 변화하며, E-Flag은 항상 clear된다. C-Flag 값은 shifted-out되어 LSB로 입력되는 bit가 Carry 값에 반영된다.



Operation

```
if(E_flag)
     imm = (ER << 3) + imm
else
     imm = {29'b0, imm}
%Rdst = Ratate Right(%Rdst >> imm)
```



Usage ROR 명령은 다은과 같이 사용하며 어떤 경우에도 하위 5bit만 유효(즉 즉치값은 ER 값이 적용되더라도 0 31 값만이 유효하며, 이보다 큰 값일 경우 modulo 32 연산의

결과 값이 적용)하다.

```
1   ROR 0x4, %R2
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.87 SADD - saturated addition

SADD 명령은 Destination Register값과 Source Register값 혹은 즉치 값을 더하는 명령이다.

F			C	B			8	7	4	3	0
1	1	1	1	1	0	0	0	dst		src/imm	SADD

Syntax

```
SADD   %Rsrc,  %Rdst
SADD   <imm>,  %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification E-Flag를 항상 clear한다.

C	Z	S	V	E
–	–	–	–	0

Operation

```
if(E_flag)
     %Rdst = %Rdst + (ER << 4 + imm)
else
     %Rdst = %Rdst + %Rsrc
```

Usage SADD연산은 두 피연산자간의 덧셈을 수행하기 위하여 사용된다. 보통은 ADD 연산과 동일하나 표현 범위를 넘어갈때 결과가 saturate되는 점이 다르다. 그 내용은 그림. 3.3을 참조한다.

```
1  SADD %R2, %R0
```

사용 가능한 피연산자로는 레지스터 혹은 즉치값이 가능하다. 만일 SADD에서 피연산자로서 즉치값을 사용하고자 할 때는 어떠한 경우에도 LERI를 이용해야 한다.

```
1 LERI 0x0
2 SADD 0x1, %R0
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C-DSP version에서 지원된다.

4.88 SADDB - signed saturated addition (SIMD_BYTE)

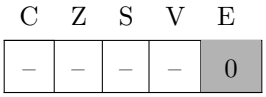
SADDB 명령은 Signed Byte 데이터에 대한 Saturate ADD를 수행하며, 4개의 연산을 동시에 수행한다.

Syntax

```
SADDB   %Rsrc,  %Rdst
SADDB   <imm>,  %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification E-Flag를 항상 clear한다.



Operation

```
if(E_flag)
     OP = (ER << 4) + imm
else
     OP = %Rsrc
%Rdst[7:0] = OP[7:0] + %Rdst[7:0]
%Rdst[15:8] = OP[15:8] + %Rdst[15:8]
%Rdst[23:16] = OP[23:16] + %Rdst[23:16]
%Rdst[31:24] = OP[31:24] + %Rdst[31:24]
```

Usage SADDB연산은 두 피연산자간의 덧셈을 8 비트 단위로 수행하기 위하여 사용되며 그 사용은 다음과 같다. 또한 사용 가능한 피연산자로는 레지스터 혹은 즉치값이 가능하며. 피연산자로서 즉치값을 사용하고자 할 때는 어떠한 경우에도 LERI를 이용해야 한다.

```
1   SADDB %R2, %R0
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C-DSP version에서 지원된다.

4.89 SADDS - signed saturated addition (SIMD_SHORT)

SADDS 명령은 Signed Byte 데이터에 대한 Saturate ADD를 수행하며, 2개의 연산을 동시에 수행한다.

F			C	B			8	7 ... 4	3 ... 0	
1	1	1	1	0	1	0	1	dst	src/imm	SADDS

Syntax

```
SADDS   %Rsrc,  %Rdst
SADDS   <imm>,  %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification E-Flag를 항상 clear한다.

C	Z	S	V	E
–	–	–	–	0

Operation

```
if(E_flag)
     OP = (ER << 4) + imm
else
     OP = %Rsrc
%Rdst[15:0] = OP[15:0] + %Rdst[15:0]
%Rdst[31:16] = OP[31:16] + %Rdst[31:16]
```

Usage SADDS연산은 두 피연산자간의 덧셈을 16 비트 단위로 수행하기 위하여 사용되며 그 사용은 다음과 같다. 또한 사용 가능한 피연산자로는 레지스터 혹은 즉치값이 가능하며. 피연산자로서 즉치값을 사용하고자 할 때는 어떠한 경우에도 LERI를 이용해야 한다.

```
1   SADDS %R2, %R0
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C-DSP version에서 지원된다.

4.90 SADUB - unsigned saturated addition (SIMD_BYTE)

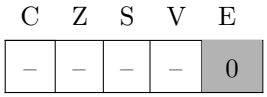
SADUB 명령은 Unsigned Byte 데이터에 대한 Saturate ADD를 수행하며, 4개의 연산을 동시에 수행한다.

Syntax

```
SADUB   %Rsrc,  %Rdst
SADUB   <imm>,  %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification E-Flag를 항상 clear한다.



Operation

```
if(E_flag)
     OP = (ER << 4) + imm
else
     OP = %Rsrc
%Rdst[7:0] = OP[7:0] + %Rdst[7:0]
%Rdst[15:8] = OP[15:8] + %Rdst[15:8]
%Rdst[23:16] = OP[23:16] + %Rdst[23:16]
%Rdst[31:24] = OP[31:24] + %Rdst[31:24]
```

Usage SADUB연산은 두 피연산자간의 덧셈을 8 비트 단위로 수행하기 위하여 사용되며 그 사용은 다음과 같다. 또한 사용 가능한 피연산자로는 레지스터 혹은 즉치값이 가능하며. 피연산자로서 즉치값을 사용하고자 할 때는 어떠한 경우에도 LERI를 이용해야 한다.

```
1   SADUB %R2, %R0
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C-DSP version에서 지원된다.

4.91 SADUS - unsigned saturated addition (SIMD_SHORT)

SADUS 명령은 Unsigned Byte 데이터에 대한 Saturate ADD를 수행하며, 2개의 연산을 동시에 수행한다.

F			C	B		8	7	4	3	0
1	1	1	1	0	1	1	1	dst	src/imm	SADUS

Syntax

```
SADUS   %Rsrc,  %Rdst
SADUS   <imm>,  %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification E-Flag를 항상 clear한다.

C	Z	S	V	E
–	–	–	–	0

Operation

```
if(E_flag)
     OP = (ER << 4) + imm
else
     OP = %Rsrc
%Rdst[15:0] = OP[15:0] + %Rdst[15:0]
%Rdst[31:16] = OP[31:16] + %Rdst[31:16]
```

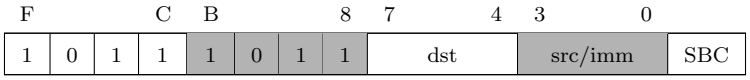
Usage SADUS연산은 두 피연산자간의 덧셈을 16 비트 단위로 수행하기 위하여 사용되며 그 사용은 다음과 같다. 또한 사용 가능한 피연산자로는 레지스터 혹은 즉치값이 가능하며. 피연산자로서 즉치값을 사용하고자 할 때는 어떠한 경우에도 LERI를 이용해야 한다.

```
1   SADUS %R2, %R0
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C-DSP version에서 지원된다.

4.92 SBC - subtract with carry

SBC 명령은 Destination Register값과 Source Register값 혹은 즉치 값과 carry_flag를 같이 더하는 명령이다.

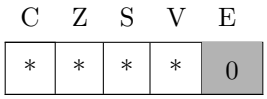


Syntax

```
SBC   %Rsrc,  %Rdst
SBC   <imm>,  %Rdst
```

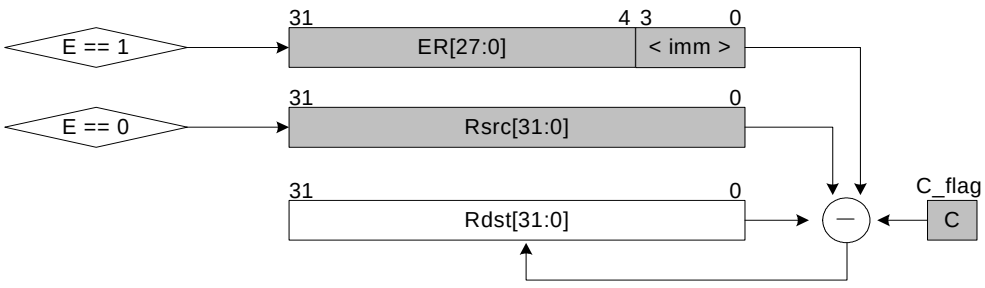
Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 명령어 수행 결과에 따라 C, Z, S, V-flag이 변화하며, E-Flag은 항상 clear된다. C-Flag 값은 연산 결과의 변경된 Carry 값이 그대로 반영된다.



Operation

```
if(E_flag)
     %Rdst = %Rdst - (ER << 4 + imm) - C_flag
else
     %Rdst = %Rdst - %Rsrc - C_flag
```



Usage SBC연산은 한 워드 이상의 뺄셈을 수행하기 위하여 사용된다.

R1:R0에 64비트 값과 R3:R2에 64비트 값이 있다고 할 때, 두 64비트의 뺄셈은 다음과

같이 사용된다.

```
1 SUB %R2, %R0
2 SBC %R3, %R1
```

이러한 동작은 R0:R1 = R3:R2 - R1:R0와 같은 동작을 수행하게 된다. 만일 ADC에서 피연산자로서 즉치값을 사용하고자 할 때는 어떠한 경우에도 LERI를 이용해야 한다.

```
1 LERI 0x0
2 SBC 0x1, %R0
```

위의 동작은 즉치 값을 1로 취하기 위하여 LERI를 사용하는 예를 보여준다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.93 SET - set status register

SET 명령은 상태 레지스터의 상태 비트 중 한 비트를 set한다. 지정 가능한 범위는 비트 번호 15~0까지로 제한되며, 이중 사용자 모드에서는 7번째 비트 이하 (7~0)까지만 set 시키는 것이 가능하다.

F			C	B			8	7			4	3	0	
1	1	1	0	0	0	0	1	0	0	1	0	Flag pos.		SET

Syntax

```
SET   <bit_pos>
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification Supervisor Mode 해당 비트 set
User Mode 해당 비트가 8이하인 경우 set

Operation

```
SR:b<bit_pos> = 1
```

상태 레지스터내 특정 비트의 값을 set한다. 상태 레지스터의 F~8 비트는 user mode에서는 접근 불가능하므로, 사용자 모드에서 위의 비트에 접근한 경우 NOP와 동일하게 처리된다.

Usage

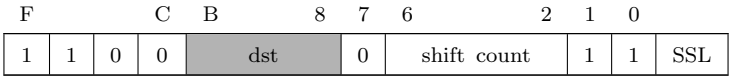
```
1   SET 0x7
```

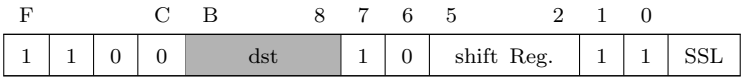
위의 예는 7번 비트(carry flag) 을 set하는 명령을 보여준다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.94 SSL - set shift left

SSL 명령은 Destination Register값을 shift count만큼 좌측으로 쉬프트하는 명령이다. 쉬프트 될 때 LSB값은 '1'이 채워진다. 쉬프트 카운트로는 즉치값과 레지스터의 값이 가능하다. 레지스터 값을 이용할 경우에는 하위 5비트만 사용된다.



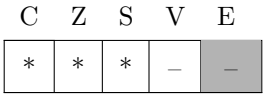


Syntax

```
SSL  <Shift_Count>, %Rdst
SSL  %Rsft, %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

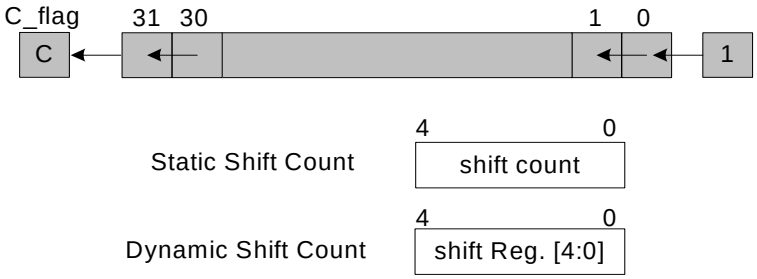
Status Modification 명령어 수행 결과에 따라 C, Z, S-flag이 변화한다. C-Flag 값은 shifted-out된 Carry 값이 반영된다. shift되는 크기가 '0'인 경우 C-Flag는 '0'을 갖는다.



Operation

```
Static Shift
     %Rdst = %Rdst << <shift_count>
Dynamic Shift
%Rdst = %Rdst << (%Rsft & 0x1f)
하위 비트들은 '1'로 채워진다.
```

Usage SSL 연산은 특수 목적에 활용된다.

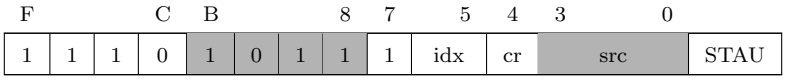



```
1  SSL 0x1, %R2
2  SSL %R0, %R2
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.95 STAU - Auto-Increment store

STAU 명령은 CR레지스터를 이용하여 자동 증가 되는 주소를 생성하고 이를 바탕으로 store 명령을 수행한다.



Syntax

```
STAU CRNO, %Rsrc, %Ridx
```

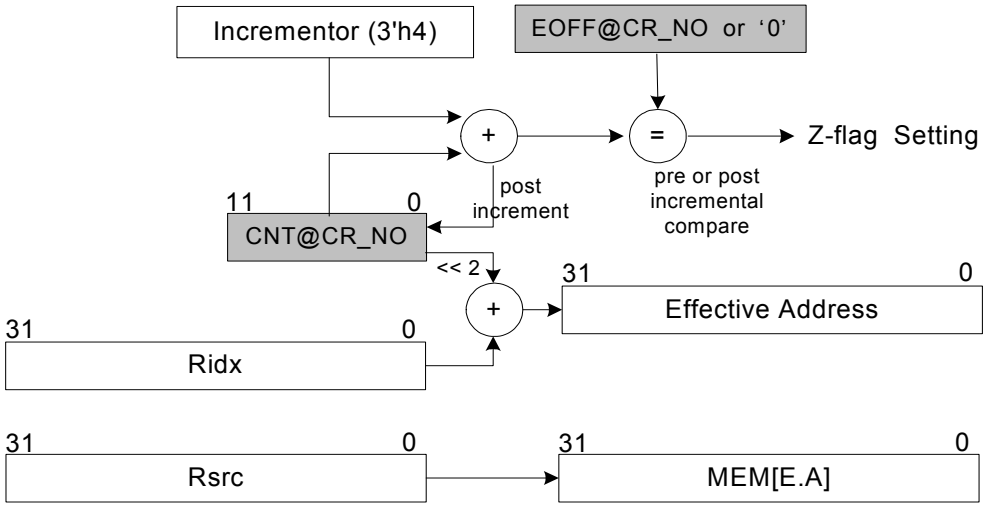
Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 동작 모드에 따라 zero-flag가 변화 됨

C	Z	S	V	E
–	– (*)	–	–	–

Operation

```
MEM(%Ridx + offset@CRNO) = %Rsrc   // increment CR and check CR
```



Usage STAU 연산은 CR레지스터를 이용하여 자동 증가 되는 주소를 생성하고 이를 바탕으로 store 명령을 수행하는 명령으로써 다음의 예는 CR0의 내용을 바탕으로 %R1와

함께 이용하여 메모리에 접근하고 %R2의 데이터를 메모리에 저장하는 예를 보인 것이다.

```
1  STAU 0x0, %R2, %R1
```

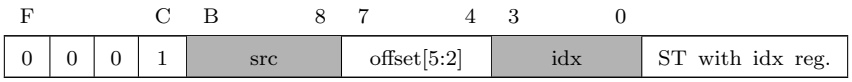
Note

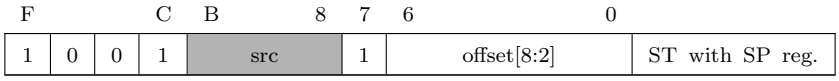
- Index register는 %R0, %R1, %SP를 사용할 수 있다.
- 00: %R0
- 01: %R1
- 1?: %SP
- 동작 모드는 CR0와 CR1의 세팅으로써 가능하며, 이에 따라 CR의 End-offset/Mask 필드를 사용한다. 각 레지스터의 필드 내용은 표. 2.2를 참고한다.
- 동작 모드가 Auto Incremental with End-offset compare mode 또는 Wrap-around Auto Incremental with End Check mode로 지정된 경우, 최종 카운트에 도달하면 SR의 zero-flag가 assert된다.
- Auto Incremental with End-offset compare mode인 경우에는 카운터의 증가되기 직전 값을 사용(pre-incremental compare)하고, Wrap-around Auto Incremental with End Check mode인 경우에는 증가된 이후의 카운터 값을 사용(post-incremental compare)한다.
- Index값은 변경되지 않으며, offset으로 사용되는 CR의 값이 변경된다

Processor Version 이 명령어는 AE32000 계열 중 AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다. 단, AE32000C의 경우, Auto Increment mode와 Auto Incremental with End-offset compare mode만 지원하며 AE32000C-Tiny의 경우에는 모드에 따라 부분적으로 지원되므로 해당 해당 프로세서의 reference manual을 참고한다.

4.96 ST - store 32-bit

ST 명령은 소스 레지스터의 값을 지정한 메모리 주소로 저장한다.





Syntax

```
ST %Rsrc, (%Ridx, <imm>)
```

%Ridx 대신 현재 Stack Pointer를 이용할 수 있다.

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification E-Flag를 항상 clear한다.

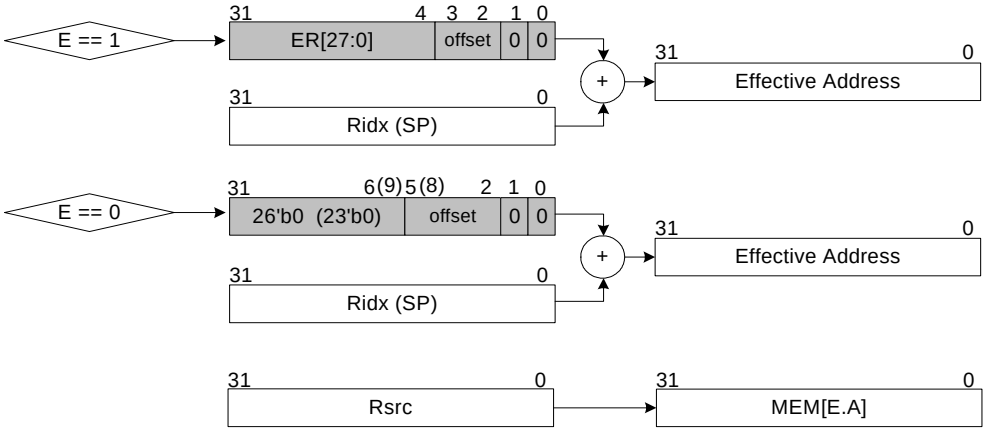
C	Z	S	V	E
–	–	–	–	0

Operation

```
if(E_flag)
     imm = (ER << 4) + ((offset & 0x3) << 2)
else
     imm = ZEXT9(offset << 2)
MEM((%Ridx or %SP) + imm) = %Rsrc
```

Usage ST명령은 레지스터에 존재하는 워드 단위(32비트)의 데이터를 메모리로 쓰는 동작을 수행한다.

```
1  ST %R3, (%R2, 0x4)
```



* Stack Pointer를 사용하는 경우, 괄호 안의 숫자를 사용

위의 경우 %R2를 인덱스 레지스터로서 사용하고, 4를 offset으로 사용하여 메모리를 접근하여 32비트 데이터를 가지고 있는 %R3의 값을 해당 메모리에 쓰는 동작을 보여준다. 인덱스 레지스터를 사용할 경우에는 offset(4비트)를 사용한다. E-Flag가 세팅되어 있는 경우, 하위 2비트를 사용({ER[27:0], offset[3:2], 2'b00})하여 imm을 생성하며 세팅되지 않았을 경우에는 4비트를 사용({26'b0, offset[5:2], 2'b00})하여 imm을 생성한다.

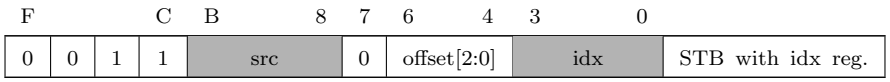
```
1  ST %R3, (%SP, 0x4)
```

ST명령에서 인덱스 레지스터 이외에 스택 포인터를 이용할 수 있으며, 이 경우 offset(7비트)를 사용한다. E-Flag가 세팅되어 있는 경우, 하위 2비트를 사용({ER[27:0], offset[3:2], 2'b00})하여 imm을 생성하며 세팅되지 않았을 경우에는 7비트를 사용({23'b0, offset[8:2], 2'b00})하여 imm을 생성한다.

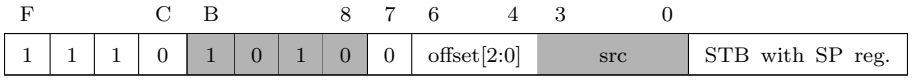
Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.97 STB - store byte

STB 명령은 지정한 레지스터의 하위 8비트 데이터를 메모리 주소에 저장한다.



F			C	B	8	7	6	4	3	0	
0	0	1	1	src		0	offset[2:0]		idx		STB with idx reg.



F			C	B			8	7	6	4	3	0	
1	1	1	0	1	0	1	0	0	offset[2:0]		src		STB with SP reg.

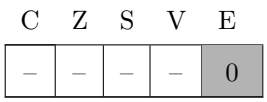
Syntax

```
STB %Rsrc, (%Ridx, <imm>)
```

%Ridx 대신 현재 Stack Pointer를 이용할 수 있다.

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification E-Flag를 항상 clear한다.



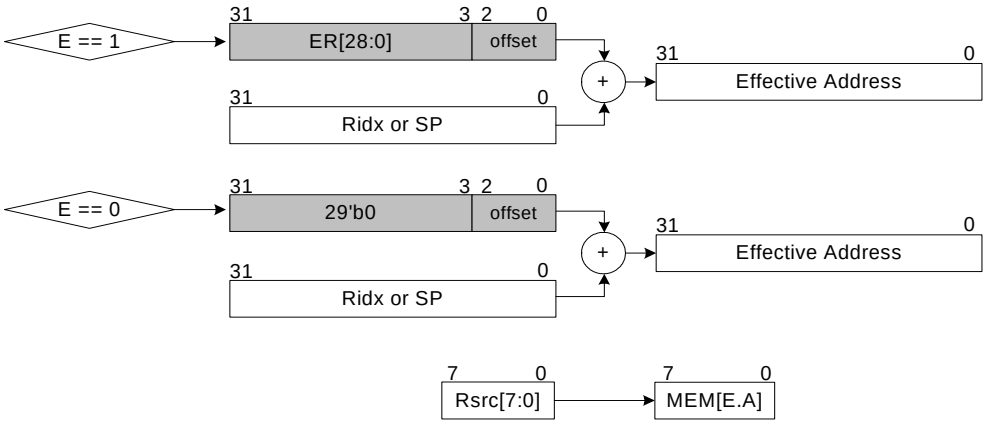
C	Z	S	V	E
–	–	–	–	0

Operation

```
if(E_flag)
     imm = (ER << 3) + offset
else
     imm = ZEXT3(offset)
MEM((%Ridx or %SP) + imm) = %Rsrc
```

Usage STB명령은 바이트 단위(8비트)로 데이터를 메모리에 write하는 동작을 수행한다.

```
1  STB %R3, (%R2, 0x4)
```



위의 경우 %R2를 인덱스 레지스터로서 사용하고, 4를 offset으로 사용하여 메모리를 접근하여 8비트 데이터를 가지고 있는 소스 레지스터의 값을 해당 주소에 write하는 것을 보여준다. STB명령에서 인덱스 레지스터 이외에 스택 포인터를 이용할 수 있으며, 이 경우 아래와 같이 사용한다.

```
1 STB %R3, (%SP, 0x4)
```

Note

- 이 명령은 LERI를 이용하여 Offset을 확장하는 것이 가능하다. 따라서 32비트까지 offset을 지정하는 것이 가능하다. 단, 사용자가 명시적으로 LERI를 넣을 필요가 없으며, 필요한 만큼 offset의 값을 지정하는 경우 자동적으로 어셈블러에서 LERI를 생성해 주시 때문에 사용자는 어셈블러를 이용할 때 임의로 LERI를 쓰지 않아도 된다. (STB의 경우 offset의 길이를 31비트 이하로 할 것을 권장한다)
- AE32000은 32비트 데이터 버스 및 32비트 단위로 정렬되어 있는 데이터를 이용하므로, 적절한 위치에 8비트 데이터를 write 하기 위행서는 데이터 버스에 값을 보내기 전에 align되어 출력된다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.98 STC - store on coprocessor

STC 명령은 보조 프로세서의 대상 레지스터에 데이터를 지정한 메모리 주소로 저장하는 명령어이다. 데이터의 크기는 최소 32비트이며, 보조 프로세서에 추가적인 메모리 버스를 추가한다면 그 이상이 될 수 있다.

F			C	B		9	8	7			4	3	0	
1	1	1	0	1	1	CP #		1	0	1	idx	src		STCn

Syntax

```
STC   <cp_no>, %Rsrc, (%R1, <imm>)
STC   <cp_no>, %Rsrc, (%SP, <imm>)
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification E-Flag를 항상 clear한다.

C	Z	S	V	E
–	–	–	–	0

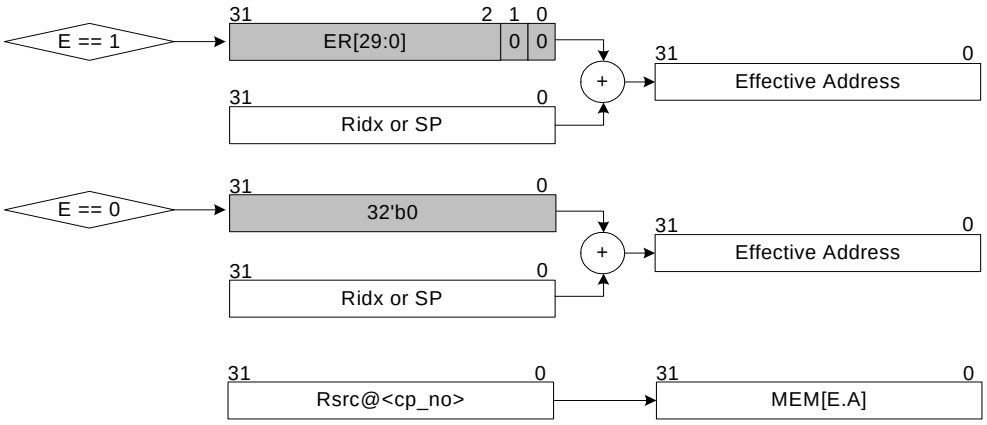
Operation STC 명령어는 idx 값이 1인 경우 스택포인터 레지스터를 지정하며, idx 값이 0인 경우 범용 레지스터 R1을 지정한다.

```
if(E_flag)  imm = ER << 2
else        imm = 0


if(idx == 1)
  MEM(%SP + imm) = %Rsrc@CP<cp_no>
else
  MEM(%R1 + imm) = %Rsrc@CP<cp_no>
```

Usage STC명령은 보조 프로세서의대상 레지스터의 데이터를 메모리에 쓰는 동작을 수행한다.

```
1   STC 0x1, %R8, (%R1, 0x4)
```



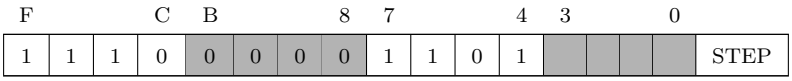
Note

- 이 명령은 명령어 필드 상에는 immediate 필드가 없으나 LERI를 이용하여 offset을 지정하는 것이 가능하다. 만일 LERI를 이용할 경우 하위 2비트는 0으로 지정되므로, 30비트까지 사용자가 주소를 지정하는 것이 가능하다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다. 단, AE32000B에서는 구현에 따라 다를 수 있으므로 해당 해당 프로세서의 reference manual을 참고한다.

4.99 STEP - single step debugging

STEP 명령은 Debugging Mode에서 Single -Step Tracing을 지원하기 위한 모드를 추가하며, 이는 OSI모드에서만 접근 가능하도록 한다.

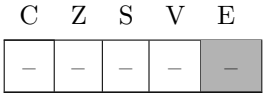


Syntax

```
STEP
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 변화 없음



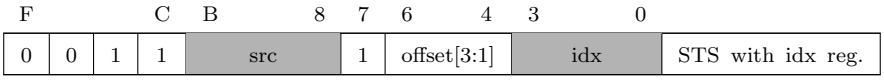
Operation

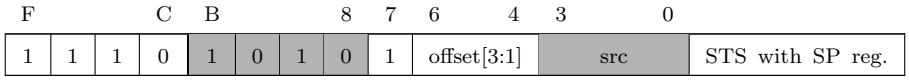
Usage Single Step은 OSI모드가 아닐 때에만 활성화되며, 상태 비트의 16번째 비트에 위치한다. (SR_SINGLE = SR_BIT16)
만일 프로세서의 동작이 OSI Mode인 경우에는 Single Step에 따른 break가 발생한 이후이므로, 이에 대한 처리를 수행하는 부분이므로 이때는 Single Step이 중단된다. Single Step의 On/Off는 오직 OSI모드에서만 가능하며, 이는 특수 명령어를 이용하여 토글 하도록 한다.
단, 관리자 모드에서는 PUSH SR/ Modify Memory Content/ POP SR의 방법을 통하여 해당 비트를 활성화시키는 것이 가능하나, 이를 권장하지는 않는다. Single Step이 활성화되어 있는 경우 현재 RDptr + 1의 패킷에 ibreak를 거는 방법을 사용하도록 한다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.100 STS - store short

STS 명령은 지정한 메모리 주소로부터 16비트의 데이터를 읽어서 부호 확장을 수행한 후 대상 레지스터에 저장한다.





Syntax

```
STS %Rsrc, (%Ridx, <imm>)
```

%Ridx 대신 현재 Stack Pointer를 이용할 수 있다.

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification E-Flag를 항상 clear한다.

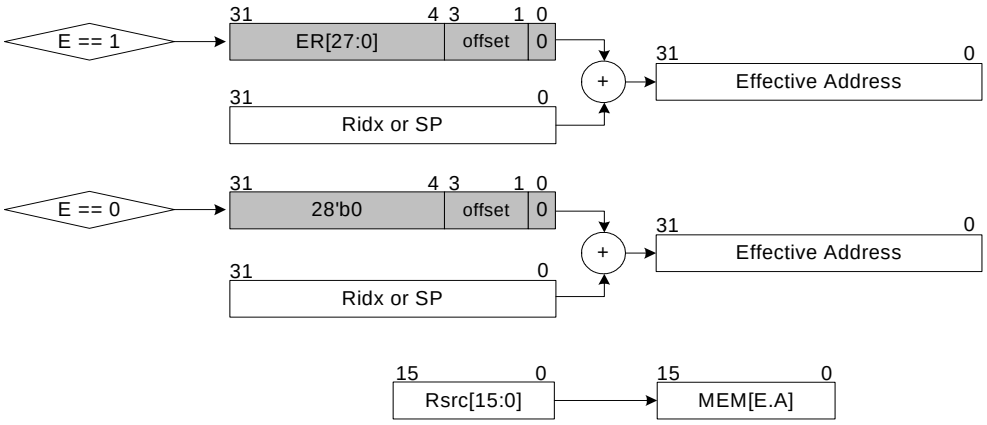
C	Z	S	V	E
–	–	–	–	0

Operation

```
if(E_flag)
     imm = ((ER << 3) + offset) << 1
else
     imm = ZEXT4(offset << 1)
MEM((%Ridx or %SP) + imm) = %Rsrc
```

Usage STS명령은 short 단위(16비트)의 데이터를 메모리에 write 하는 동작을 수행한다.

```
1  STS %R3, (%R2, 0x4)
```



위의 경우 %R2를 인덱스 레지스터로서 사용하고, 4를 offset으로 사용하여 메모리를 접근하여 16비트 데이터를 가지고 있는 소스 레지스터의 값을 해당 주소에 write 하는 것을 보여준다. STS명령에서 인덱스 레지스터 이외에 스택 포인터를 이용할 수 있으며, 이 경우 아래와 같이 사용한다.

```
1 STS %R3, (%SP, 0x4)
```

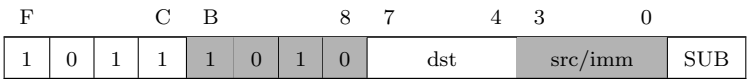
Note

- 이 명령은 LERI를 이용하여 Offset을 확장하는 것이 가능하다. 따라서 32비트까지 offset을 지정하는 것이 가능하다. 단, 사용자가 명시적으로 LERI를 넣을 필요가 없으며, 필요한 만큼 offset의 값을 지정하는 경우 자동적으로 어셈블러에서 LERI를 생성해 주기 때문에 사용자는 어셈블러를 이용할 때 임의로 LERI를 쓰지 않아도 된다.
- AE32000은 32비트 데이터 버스 및 32비트 단위로 정렬되어 있는 데이터를 이용하므로, 16비트 단위의 데이터를 적절한 위치로 정렬한 후 데이터 버스로 출력하여야 한다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.101 SUB - subtract

SUB 명령은 Destination Register값으로부터 Source Register값 혹은 즉치 값을 뺀다.

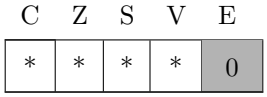


Syntax

```
SUB   %Rsrc,  %Rdst
SUB   <imm>,  %Rdst
```

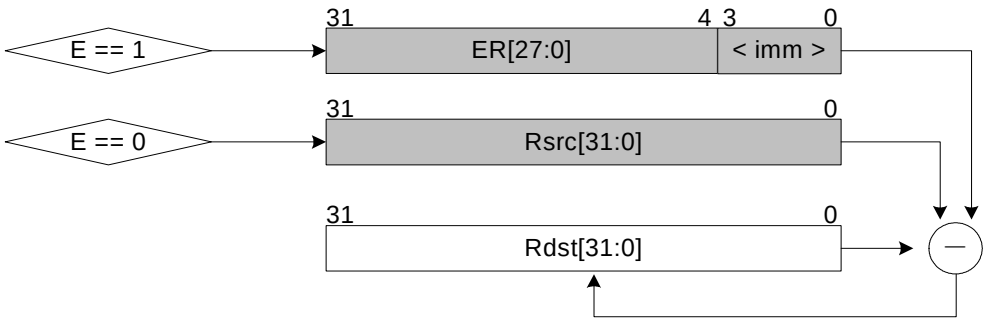
Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 명령어 수행 결과에 따라 C, Z, S, V-flag이 변화하며, E-Flag은 항상 clear된다. C-Flag 값은 연산 결과의 변경된 Carry 값이 그대로 반영된다.



Operation

```
if(E\_flag)
     %Rdst = %Rdst - (ER << 4 + imm)
else
     %Rdst = %Rdst - %Rsrc
```



Usage SUB연산은 두 피연산자간의 덧셈을 수행하기 위하여 사용된다.

```
1  SUB %R2, %R0
```

만일 SUB에서 피연산자로서 즉치값을 사용하고자 할 때는 어떠한 경우에도 LERI를 이용해야 한다.

```
1  LERI 0x0
2  SUB 0x1, %R0
```

위의 동작은 즉치 값을 1로 취하기 위하여 LERI를 사용하는 예를 보여준다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.102 SWI - software interrupt

SWI 명령은 Software적으로 인터럽트를 발생시키는 명령이다.

F			C	B			8	7			4	3	0	
1	1	1	0	0	0	0	0	1	1	0	0	imm		SWI

Syntax

```
SWI   <int_no>
```

<int_no> 호출할 인터럽트 번호

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification

- Proc_mode – Supervisor Mode로 변경 (Proc_mode = 5'b00)
- INT – Disable
- E_flag – Clear
- L_flag – Clear

Operation

1. SR을 SSP를 이용하여 저장
2. SR 값 변경
3. PC 저장
4. <imm>를 이용하여 인터럽트 벡터 테이블 접근
   tbl_addr = {2'b10,<imm>,2'b0}
5. 인터럽트 핸들러에 존재하는 SWI 루틴 수행

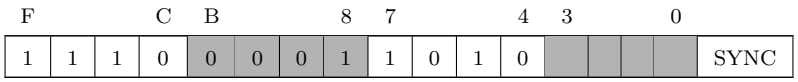
Usage 이 명령은 사용자 모드에서 관리자 모드로 변경하여 해당 동작을 수행하고자 할 때 사용된다. 일반적으로, 시스템 콜을 이용하기 위하여 많이 사용된다.

```
0x030 ADD  %R1, %R2
0x032 SWI  0x5
...interrupt 처리 ... 복귀 ...
0x034 ADD  %R2, %R4
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.103 SYNC - synchronization

SYNC 명령은 파이프 라인을 flush시킨 후 현재 PC값으로 분기하는 명령이다.



Syntax

```
SYNC
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 변화 없음

Operation

```
BRPC = PC
```

* BRPC : Branch PC를 의미한다.
이 명령은 프로세서 자체에는 특별한 동작을 수행하지 않고, 공유 영역에 대한 외부의 load/store가 종결될 때까지 파이프라인을 정지하는 동작을 수행한다.

Usage

```
1 LD (%R4, 0x4), %R3
2 SYNC
```

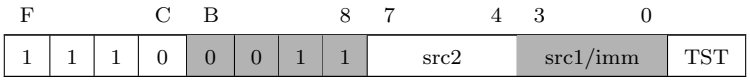
SYNC명령은 원래 shared memory를 이용하는 multiprocessor 에 대한 프로그래밍을 할 때 critical section의 동작을 보장하기 위하여 사용되는 명령으로서, 다수의 프로세서가 sync 까지의 동작이 모두 완결될 때까지 모든 프로세서는 critical section으로 진입하지 않는다.

System Coprocessor의 설정을 변경하는 경우에 뒤따르는 명령이 변경된 설정에 대하여 적절하게 반응하도록 하기 위하여 Sync 명령을 이용한다. 일반적으로 Sync명령은 대부분 Write-Back Mode의 Cache/Write -Buffer를 Memory에 Sync시키는 동작을 수행한다.

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다. 단, AE32000B에서는 구현에 따라 다를 수 있으므로 해당 해당 프로세서의 reference manual을 참고한다.

4.104 TST - test(logical compare)

TST 는 두 소스 레지스터를 비트 단위로 비교하는 명령이다.



Syntax

```
TST   %Rsrc1,  %Rsrc2
TST   <imm>,  %Rsrc2
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 명령어 수행 결과에 따라 Z, S-flag이 변화하며, E-Flag은 항상 clear된다.

C	Z	S	V	E
–	*	*	–	0

Operation

```
if(E_flag)
     %Rsrc2 & (ER << 4 + imm)
else
     %Rsrc2 & %Rsrc1
```

Usage TST연산은 두 피연산자간에 비교를 위하여 bitwise AND연산을 수행하여 사용된다.

```
1   TST %R2, %R0
```

즉치값을 사용하기 위해서는 반드시 LERI를 이용해야 한다.

```
1   LERI 0x0
2   TST 0x1, %R0
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

4.105 UPKHS - unpack short to higher part

UPKHS 명령은 연속된 두개의 소스 레지스터에서 Short단위의 데이터를 추출하여 목적 레지스터에 넣는다.

Syntax

```
UPKHS src_grp, %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 변화 없음

C	Z	S	V	E
–	–	–	–	–

Operation (3.7.3절 참조)

```
%Rdst[15:0] = %R{src_grp, 0}[31:16]
%Rdst[31:16] = %R{src_grp, 1}[31:16]
```

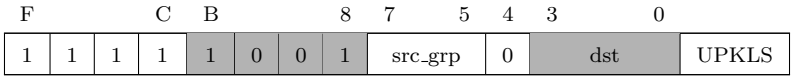
Usage Unpack은 SIMD 구조에서 필요한 연산의 형태를 정렬하기 위하여 사용되는 명령으로서, 필요에 따라 Word 전체에서 사용되는 필드들을 재정렬하여 이후에 사용되는 SIMD 연산의 효율을 높이기 위하여 사용된다.
다음은 %R2와 %R3의 상위 16 비트 씩을 가져와 %R6에 채우는 동작을 수행하는 예이다.

```
1 UPKHS %R2, %R6
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C-DSP version에서 지원된다.

4.106 UPKLS - unpack short to lower part

UPKLS 명령은 연속된 두개의 소스 레지스터에서 Short단위의 데이터를 추출하여 목적 레지스터에 넣는다.



Syntax

```
UPKLS src_grp, %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 변화 없음

C	Z	S	V	E
–	–	–	–	–

Operation (3.7.3절 참조)

```
%Rdst[15:0] = %R{src_grp, 0}[15:0]
%Rdst[31:16] = %R{src_grp, 1}[15:0]
```

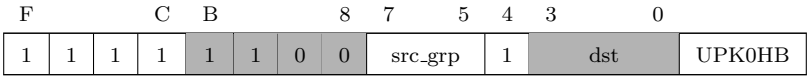
Usage Unpack은 SIMD 구조에서 필요한 연산의 형태를 정렬하기 위하여 사용되는 명령으로서, 필요에 따라 Word 전체에서 사용되는 필드들을 재정렬하여 이후에 사용되는 SIMD 연산의 효율을 높이기 위하여 사용된다.
다음은 %R2와 %R3의 하위 16 비트 씩을 가져와 %R6에 채우는 동작을 수행하는 예이다.

```
1 UPKLS %R2, %R6
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C-DSP version에서 지원된다.

4.107 UPK0HB - unpack 0 byte to higher part

UPK0HB 명령은 연속된 두개의 소스 레지스터에서 Byte단위의 데이터를 추출하여 목적 레지스터에 넣는다.

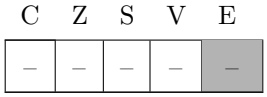


Syntax

```
UPK0HB src_grp, %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 변화 없음



Operation (3.7.3절 참조)

```
%Rdst[23:16] = %R{src\_grp, 0}[7:0]
%Rdst[31:24] = %R{src\_grp, 1}[7:0]
```

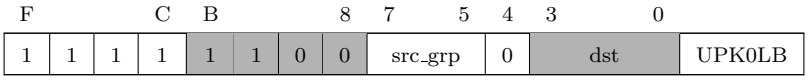
Usage Byte unpack 명령은 목적 레지스터의 상위 16비트 또는 하위 16비트를 채우는 명령이므로, 하나의 Word를 구성하기 위해서는 2개의 unpack 명령이 사용되어야 한다. 다음은 %R2, %R3, %R4, %R5의 최하위 8 비트 씩을 가져와 Word 길이의 %R6를 만드는 동작을 수행하는 예이다.

```
1 UPK0HB %R2, %R6
2 UPK0LB %R4, %R6
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C-DSP version에서 지원된다.

4.108 UPK0LB - unpack 0 byte to lower part

UPK0LB 명령은 연속된 두개의 소스 레지스터에서 Byte단위의 데이터를 추출하여 목적 레지스터에 넣는다.

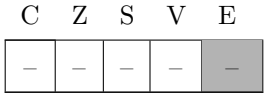


Syntax

```
UPK0LB src_grp, %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 변화 없음



Operation (3.7.3절 참조)

```
%Rdst[7:0] = %R{src_grp, 0}[7:0]
%Rdst[15:8] = %R{src_grp, 1}[7:0]
```

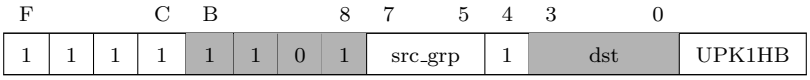
Usage Byte unpack 명령은 목적 레지스터의 상위 16비트 또는 하위 16비트를 채우는 명령이므로, 하나의 Word를 구성하기 위해서는 2개의 unpack 명령이 사용되어야 한다. 다음은 %R2, %R3, %R4, %R5의 최하위 8 비트 씩을 가져와 Word 길이의 %R6를 만드는 동작을 수행하는 예이다.

```
1 UPK0HB %R2, %R6
2 UPK0LB %R4, %R6
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C-DSP version에서 지원된다.

4.109 UPK1HB - unpack 1 byte to higher part

UPK1HB 명령은 연속된 두개의 소스 레지스터에서 Byte단위의 데이터를 추출하여 목적 레지스터에 넣는다.

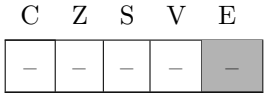


Syntax

```
UPK1HB src_grp, %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 변화 없음



Operation (3.7.3절 참조)

```
%Rdst[23:16] = %R{src_grp, 0}[15:8]
%Rdst[31:24] = %R{src_grp, 1}[15:8]
```

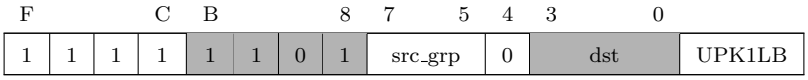
Usage Byte unpack 명령은 목적 레지스터의 상위 16비트 또는 하위 16비트를 채우는 명령이므로, 하나의 Word를 구성하기 위해서는 2개의 unpack 명령이 사용되어야 한다. 다음은 %R2, %R3, %R4, %R5의 하위 두번째 8 비트 씩을 가져와 Word 길이의 %R6를 만드는 동작을 수행하는 예이다.

```
1 UPK1HB %R2, %R6
2 UPK1LB %R4, %R6
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C-DSP version에서 지원된다.

4.110 UPK1LB - unpack 1 byte to lower part

UPK1LB 명령은 연속된 두개의 소스 레지스터에서 Byte단위의 데이터를 추출하여 목적 레지스터에 넣는다.

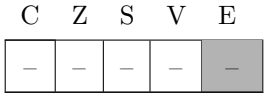


Syntax

```
UPK1LB src_grp, %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 변화 없음



Operation (3.7.3절 참조)

```
%Rdst[7:0] = %R{src_grp, 0}[15:8]
%Rdst[15:8] = %R{src_grp, 1}[15:8]
```

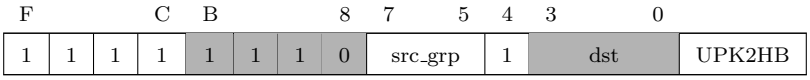
Usage Byte unpack 명령은 목적 레지스터의 상위 16비트 또는 하위 16비트를 채우는 명령이므로, 하나의 Word를 구성하기 위해서는 2개의 unpack 명령이 사용되어야 한다. 다음은 %R2, %R3, %R4, %R5의 하위 두번째 8 비트 씩을 가져와 Word 길이의 %R6를 만드는 동작을 수행하는 예이다.

```
1 UPK1HB %R2, %R6
2 UPK1LB %R4, %R6
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C-DSP version에서 지원된다.

4.111 UPK2HB - unpack 2 byte to higher part

UPK2HB 명령은 연속된 두개의 소스 레지스터에서 Byte단위의 데이터를 추출하여 목적 레지스터에 넣는다.

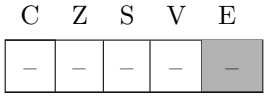


Syntax

```
UPK2HB src_grp, %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 변화 없음



Operation (3.7.3절 참조)

```
%Rdst[23:16] = %R{src_grp, 0}[23:16]
%Rdst[31:24] = %R{src_grp, 1}[23:16]
```

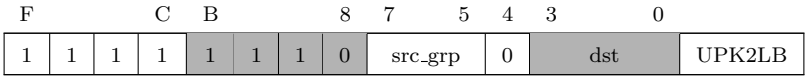
Usage Byte unpack 명령은 목적 레지스터의 상위 16비트 또는 하위 16비트를 채우는 명령이므로, 하나의 Word를 구성하기 위해서는 2개의 unpack 명령이 사용되어야 한다. 다음은 %R2, %R3, %R4, %R5의 하위 세번째 8 비트 씩을 가져와 Word 길이의 %R6를 만드는 동작을 수행하는 예이다.

```
1 UPK2HB %R2, %R6
2 UPK2LB %R4, %R6
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C-DSP version에서 지원된다.

4.112 UPK2LB - unpack 2 byte to lower part

UPK2LB 명령은 연속된 두개의 소스 레지스터에서 Byte단위의 데이터를 추출하여 목적 레지스터에 넣는다.

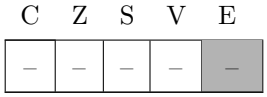


Syntax

```
UPK2LB src_grp, %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 변화 없음



Operation (3.7.3절 참조)

```
%Rdst[7:0] = %R{src_grp, 0}[23:16]
%Rdst[15:8] = %R{src_grp, 1}[23:16]
```

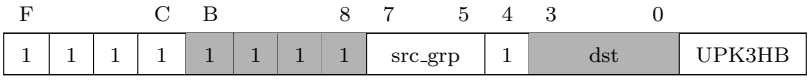
Usage Byte unpack 명령은 목적 레지스터의 상위 16비트 또는 하위 16비트를 채우는 명령이므로, 하나의 Word를 구성하기 위해서는 2개의 unpack 명령이 사용되어야 한다. 다음은 %R2, %R3, %R4, %R5의 하위 세번째 8 비트 씩을 가져와 Word 길이의 %R6를 만드는 동작을 수행하는 예이다.

```
1 UPK2HB %R2, %R6
2 UPK2LB %R4, %R6
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C-DSP version에서 지원된다.

4.113 UPK3HB - unpack 3 byte to higher part

UPK3HB 명령은 연속된 두개의 소스 레지스터에서 Byte단위의 데이터를 추출하여 목적 레지스터에 넣는다.

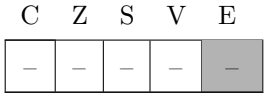


Syntax

```
UPK3HB src_grp, %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 변화 없음



Operation (3.7.3절 참조)

```
%Rdst[23:16] = %R{src_grp, 0}[31:24]
%Rdst[31:24] = %R{src_grp, 1}[31:24]
```

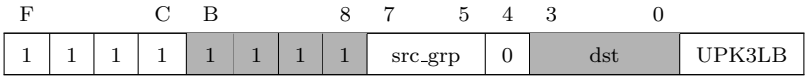
Usage Byte unpack 명령은 목적 레지스터의 상위 16비트 또는 하위 16비트를 채우는 명령이므로, 하나의 Word를 구성하기 위해서는 2개의 unpack 명령이 사용되어야 한다. 다음은 %R2, %R3, %R4, %R5의 최상위 8 비트 씩을 가져와 Word 길이의 %R6를 만드는 동작을 수행하는 예이다.

```
1 UPK3HB %R2, %R6
2 UPK3LB %R4, %R6
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C-DSP version에서 지원된다.

4.114 UPK3LB - unpack 2 byte to lower part

UPK3LB 명령은 연속된 두개의 소스 레지스터에서 Byte단위의 데이터를 추출하여 목적 레지스터에 넣는다.

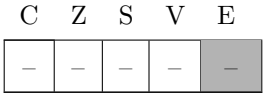


Syntax

```
UPK3LB src_grp, %Rdst
```

Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 변화 없음



Operation (3.7.3절 참조)

```
%Rdst[7:0] = %R{src_grp, 0}[31:24]
%Rdst[15:8] = %R{src_grp, 1}[31:24]
```

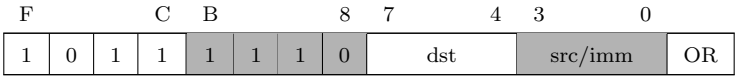
Usage Byte unpack 명령은 목적 레지스터의 상위 16비트 또는 하위 16비트를 채우는 명령이므로, 하나의 Word를 구성하기 위해서는 2개의 unpack 명령이 사용되어야 한다. 다음은 %R2, %R3, %R4, %R5의 최상위 8 비트 씩을 가져와 Word 길이의 %R6를 만드는 동작을 수행하는 예이다.

```
1 UPK3HB %R2, %R6
2 UPK3LB %R4, %R6
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000C-DSP version에서 지원된다.

4.115 XOR - bitwise XOR

XOR 는 목적 레지스터 또는 즉치 값과 대상 레지스터의 값을 비트 단위로 XOR연산을 취한다.

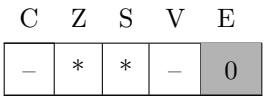


Syntax

```
XOR   %Rsrc,  %Rdst
XOR   <imm>,  %Rdst
```

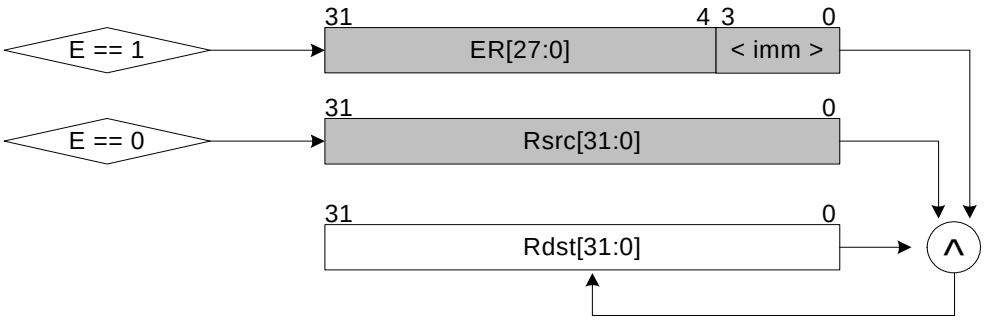
Exceptions 이 명령에 연관된 예외 사항은 없다.

Status Modification 명령어 수행 결과에 따라 Z, S-flag이 변화하며, E-Flag은 항상 clear된다.



Operation

```
if(E_flag)
     %Rdst = %Rdst ^ (ER << 4 + imm)
else
     %Rdst = %Rdst ^ %Rsrc
```



Usage XOR 연산은 두 피연산자간에 bitwise XOR 연산을 수행하기 위하여 사용된다.

```
1 XOR %R0, %R2
```

즉치값을 사용하기 위해서는 반드시 LERI를 이용해야 한다.

```
1 LERI 0x0
2 XOR 0x1, %R2
3 ...
```

Processor Version 이 명령어는 AE32000 계열 중 AE32000B, AE32000C, AE32000C-DSP, AE32000C-Tiny version에서 지원된다.

Appendix A

찾아보기