RealView® Compilation Tools

버전 4.0

어셈블러 설명서

ARM®

ARM DUI 0204IK

RealView Compilation Tools
어셈블러 설명서

릴리스 정보

이 설명서에서 변경된 내용은 다음과 같습니다.

변경 내역

날짜	발행판	기밀 상태	변경 내용
2002년 8월	A	기밀 문서 아님	릴리스 1.2
2003년 1월	B	기밀 문서 아님	릴리스 2.0
2003년 9월	C	기밀 문서 아님	RealView Development Suite 버전 2.0용 릴리스 2.0.1
2004년 1월	D	기밀 문서 아님	RealView Development Suite 버전 2.1용 릴리스 2.1
2004년 12월	E	기밀 문서 아님	RealView Development Suite 버전 2.2용 릴리스 2.2
2005년 5월	F	기밀 문서 아님	RealView Development Suite 버전 2.2 SP1용 릴리스 2.2
2006년 3월	G	기밀 문서 아님	RealView Development Suite 버전 3.0용 릴리스 3.0
2007년 3월	H	기밀 문서 아님	RealView Development Suite 버전 3.1용 릴리스 3.1
2008년 9월	I	기밀 문서 아님	RealView Development Suite 버전 4.0용 릴리스 4.0

소유권 고지 사항

이 설명서에 설명된 제품은 지속적으로 개발 및 개선될 수 있습니다. 이 설명서에 포함된 모든 제품 명세와 해당 사용법은 ARM의 신뢰하에 제공됩니다. 그러나 ARM에서는 상품성 또는 특정 목적에의 적합성을 비롯하여 그 밖의 묵시적이거나 명시적인 모든 보증을 부인합니다.

이 설명서는 제품 사용자를 지원하는 용도로만 만들어졌습니다. ARM은 이 설명서 정보의 사용, 정보의 오류나 누락 또는 제품의 잘못된 사용에 따른 어떠한 손실이나 손상도 책임지지 않습니다.

ARM이라는 단어가 사용되는 경우 "ARM이나 해당하는 자회사"를 의미합니다.

기밀 상태

이 설명서는 기밀 문서가 아닙니다. 이 설명서의 사용, 복사 및 공개 권한은 ARM과 설명서 사용 당사자의 동의하에 라이센스 제한을 받습니다.

액세스 제한 없음은 ARM의 내부 분류입니다.

제품 상태

이 설명서의 정보는 개발이 완료된 제품에 대한 최종 정보입니다.

웹 주소

`http://www.arm.com`

목차
어셈블러 설명서

서문

설명서 정보 x
사용자 의견 xiv

1 장 소개

1.1 RealView Compilation Tools 어셈블러 개요 1-2

2 장 ARM 어셈블리 언어 작성

2.1 소개 2-2
2.2 ARM 아키텍처 개요 2-4
2.3 어셈블리 언어 모듈의 구조 2-13
2.4 조건부 실행 2-20
2.5 레지스터에 상수 로드 2-28
2.6 레지스터에 주소 로드 2-36
2.7 다중 레지스터 로드 및 저장 명령어 2-42
2.8 매크로 사용 2-49
2.9 기호 버전 추가 2-53
2.10 프레임 지시어 사용 2-54
2.11 어셈블리 언어 변경 사항 2-55

3 장 어셈블러 참조
3.1 명령 구문 3-2
3.2 소스 행 형식 3-21
3.3 미리 정의된 레지스터 및 보조 프로세서 이름 3-22
3.4 기본 제공 변수 및 상수 3-24
3.5 기호 3-27
3.6 식 , 리터럴 및 연산자 3-33
3.7 진단 메시지 3-46
3.8 C 사전 처리기 사용 3-48

4 장 ARM 및 Thumb 명령어
4.1 명령어 요약 4-3
4.2 메모리 액세스 명령어 4-10
4.3 일반 데이터 처리 명령어 4-41
4.4 곱하기 명령어 4-71
4.5 포화 명령어 4-92
4.6 병렬 명령어 4-97
4.7 패킹 및 패킹 해제 명령어 4-105
4.8 분기 및 제어 명령어 4-113
4.9 보조 프로세서 명령어 4-123
4.10 기타 제한 4-131
4.11 Thumb 에서 명령어 너비 선택 4-148
4.12 ThumbEE 명령어 4-150
4.13 의사 명령어 4-154

5 장 NEON 및 VFP 프로그래밍
5.1 명령어 요약 5-3
5.2 NEON 및 VFP 에 대한 아키텍처 지원 5-9
5.3 확장 레지스터 뱅크 5-10
5.4 조건 코드 5-13
5.5 일반 정보 5-15
5.6 NEON 및 VFP 공유 명령어 5-23
5.7 NEON 논리 및 비교 연산 5-31
5.8 NEON 일반 데이터 처리 명령어 5-41
5.9 NEON 시프트 명령어 5-53
5.10 NEON 일반 산술 명령어 5-60
5.11 NEON 곱하기 명령어 5-73
5.12 NEON 요소 및 구조체 로드 / 저장 명령어 5-78
5.13 NEON 및 VFP 의사 명령어 5-86
5.14 NEON 및 VFP 시스템 레지스터 5-93
5.15 0 으로 플러시 모드 5-98
5.16 VFP 명령어 5-100
5.17 VFP 벡터 모드 5-110

6 장 Wireless MMX 기술 명령어
6.1 소개 6-2
6.2 Wireless MMX 기술에 대한 ARM 지원 6-3
6.3 Wireless MMX 명령어 6-9

7 장 지시어 참조
7.1 지시어의 사전순 목록 7-2
7.2 기호 정의 지시어 7-4
7.3 데이터 정의 지시어 7-16
7.4 어셈블리 제어 지시어 7-32
7.5 프레임 지시어 7-41
7.6 보고 지시어 7-57
7.7 명령어 세트 및 구문 선택 지시어 7-63
7.8 기타 지시어 7-66

서문

이 서문에서는 *RealView Compilation Tools 어셈블러 설명서*에 대해 소개합니다. 이 서문은 다음 단원으로 구성되어 있습니다.

- x페이지의 *설명서 정보*
- xiv페이지의 *사용자 의견*

설명서 정보

이 설명서에서는 *RealView® Compilation Tools* 어셈블러 (ARM 어셈블러) 에 대한 자습 및 참조 정보를 제공합니다. RVCT 어셈블러에는 독립형 어셈블러인 armasm 과 C 및 C++ 컴파일러의 인라인 어셈블러가 있습니다. 이 설명서에서는 어셈블리 언어 프로그래머가 사용할 수 있는 어셈블러, 어셈블리 언어 니모닉, 의사 명령어, 매크로 및 지시문에 대한 명령 행 옵션에 대해 설명합니다.

대상 독자

이 설명서는 RealView Compilation Tools를 사용하여 응용 프로그램을 만드는 모든 개발자를 위한 것입니다. 여기에서는 사용자가 경험 있는 소프트웨어 개발자이고 RealView Compilation Tools 핵심 설명서에 설명된 ARM 개발 도구에 익숙하다고 가정합니다.

설명서 사용

이 설명서는 다음 장으로 구성되어 있습니다.

1장 *소개*

이 장에서는 ARM 어셈블러 및 어셈블리 언어에 대해 소개합니다.

2장 *ARM 어셈블리 언어 작성*

이 장에서는 ARM 어셈블러 및 어셈블리 언어를 사용하는 데 도움을 주는 자습 정보를 제공합니다.

3장 *어셈블러 참조*

이 장에서는 ARM 어셈블러에서 제공하는 언어의 구문 및 구조에 대한 참조 자료를 제공합니다.

4장 *ARM 및 Thumb 명령어*

이 장에서는 ARM 및 Thumb (Thumb-2, Thumb-2 이전 Thumb 및 Thumb-2EE 포함) 명령어 세트에 대한 참조 자료를 제공합니다.

5장 *NEON 및 VFP 프로그래밍*

이 장에서는 ARM NEON™ 기술 및 VFP 명령어 세트에 대한 참조 자료를 제공하고 타사 VFP 관련 어셈블리 언어에 대해서도 설명합니다.

6장 *Wireless MMX 기술 명령어*

이 장에서는 Wireless MMX™ 기술의 ARM 지원에 대한 참조 자료를 제공합니다.

7장 *지시어 참조*

이 장에서는 ARM 어셈블러인 armasm에서 사용할 수 있는 어셈블러 지시문에 대한 참조 자료를 제공합니다.

이 설명서에서는 ARM 소프트웨어가 기본 위치 (Windows의 경우 *volume*:\Program Files\ARM) 에 설치되어 있다고 가정합니다. 예를 들어 *install_directory*\Documentation\...과 같은 경로 이름을 참조할 때 *install_directory*는 이 위치를 가리키는 것으로 가정합니다. ARM 소프트웨어를 다른 위치에 설치한 경우에는 이 위치를 변경해야 합니다.

표기 규칙

이 설명서에서는 다음과 같은 표기 규칙을 사용합니다.

monospace 명령, 파일 및 프로그램 이름, 소스 코드와 같이 키보드로 입력할 수 있는 텍스트를 나타냅니다.

mono**space** 명령 또는 옵션 대신 사용할 수 있는 약어를 나타냅니다. 밑줄이 그어진 텍스트는 전체 명령이나 옵션 이름 대신 입력할 수 있습니다.

monospace italic

명령 및 함수의 인수를 나타냅니다. 인수는 특정 값으로 대체할 수 있습니다.

고정 폭 굵은 글꼴

외부 예제 코드가 사용될 경우 언어 키워드를 나타냅니다.

기울임 글꼴 중요한 사항을 강조 표시하고, 특수 용어를 소개하며, 내부 상호 참조 및 인용 부분을 나타냅니다.

굵은 글꼴 메뉴 이름과 같은 인터페이스 요소를 강조 표시합니다. 적절한 경우 설명 목록의 내용을 강조할 때와 ARM 프로세서 신호 이름을 표시할 때도 사용됩니다.

추가 정보

이 단원에는 ARM 계열 프로세서용 코드를 개발하는 데 대한 추가 정보를 제공하는 ARM Limited 및 타사 게시물 목록이 나와 있습니다.

ARM Limited는 설명서의 내용을 정기적으로 업데이트하고 수정합니다. `http://infocenter.arm.com/help/index.jsp`에서 정오표, 추가 목록 및 ARM *FAQ (질문과 대답)* 를 참조하십시오.

ARM 게시물

이 설명서에는 RealView Compilation Tools와 함께 제공되는 개발 도구와 관련된 정보가 포함되어 있습니다. 이 제품군에 포함된 다른 게시물은 다음과 같습니다.

- *RVCT 핵심 설명서* (ARM DUI 0202)
- *RVCT 컴파일러 사용 설명서* (ARM DUI 0205)
- *RVCT 컴파일러 참조 설명서* (ARM DUI 0348)
- *RVCT 라이브러리 및 부동 소수점 지원 설명서* (ARM DUI 0349)
- *RVCT 링커 사용 설명서* (ARM DUI 0206)
- *RVCT 링커 참조 설명서* (ARM DUI 0381)
- *RVCT 유틸리티 설명서* (ARM DUI 0382)
- *RVCT 개발자 설명서* (ARM DUI 0203)

ARM에서 지원하는 기본 표준, 소프트웨어 인터페이스 및 기타 표준에 대한 자세한 내용은 *install_directory*`\Documentation\Specifications\...`에서 볼 수 있습니다.

또한 ARM 제품과 관련된 구체적인 내용은 다음 설명서를 참조하십시오.

- *ARM 아키텍처 참조 문서, ARMv7-A 및 ARMv7-R edition* (ARM DDI 0406)
- *ARMv7-M 아키텍처 참조 문서* (ARM DDI 0403)
- *ARMv6-M 아키텍처 참조 문서* (ARM DDI 0419)
- *ARM 참조 주변 기기 사양* (ARM DDI 0062)
- 하드웨어 장치에 대한 ARM 데이터시트 또는 기술 참조 문서

기타 게시물

ARM 아키텍처에 대한 소개는 *ARM system-on-chip 구조* (Steve Furber 저, 나종화 등 역, 홍릉과학출판사, 2005년, ISBN 8972833592) 를 참조하십시오.

Intel® Wireless MMX™ 기술에 대한 자세한 내용은 *Wireless MMX Technology Developer Guide* (2000년 8월, Order Number: 251793-001) 를 참조하십시오. 이 설명서는 `http://www.intel.com`에서 확인할 수 있습니다.

사용자 의견

RealView Compilation Tools와 해당 설명서에 대한 의견이 있으시면 ARM Limited에 알려 주시기 바랍니다.

RealView Compilation Tools에 대한 사용자 의견

RealView Compilation Tools와 관련된 문제가 있으시면 해당 공급업체에 문의하십시오. 문의 시 다음 사항을 함께 알려 주시면 보다 신속하고 유용한 답변을 받으실 수 있습니다.

- 사용자 이름 및 회사
- 제품 일련 번호
- 사용 중인 릴리스 정보
- 실행 중인 플랫폼의 세부 사항 (예: 하드웨어 플랫폼, 운영 체제 종류 및 버전)
- 문제를 재현하는 작은 독립 실행형 코드 샘플
- 의도한 결과와 실제로 발생한 결과에 대한 명확한 설명
- 사용한 명령 (명령 행 옵션 포함)
- 문제를 보여 주는 샘플 출력
- 도구의 버전 문자열 (버전 번호 및 빌드 번호 포함)

설명서에 대한 사용자 의견

이 설명서에 오류나 누락이 있으면 다음 사항을 기재하여 `errata@arm.com`으로 전자 메일을 보내 주시기 바랍니다.

- 설명서 제목
- 설명서 번호
- 문의 내용에 해당하는 페이지 번호
- 문제에 대한 간략한 설명

추가 및 향상되었으면 하는 기능에 대한 일반적인 제안도 환영합니다.

1장
소개

이 장에서는 *RealView® Compilation Tools*와 함께 제공되는 어셈블러에 대해 소개합니다. 이 장에는 다음 단원이 포함되어 있습니다.

- 1-2페이지의 *RealView Compilation Tools 어셈블러 개요*

1.1 RealView Compilation Tools 어셈블러 개요

RVCT (*RealView Compilation Tools*) 에서는 다음 항목이 제공됩니다.

- 독립형 어셈블러인 armasm (이 설명서에서 설명함)
- C 및 C++ 컴파일러에 기본적으로 제공된, 최적화 기능이 있는 인라인 어셈블러와 최적화 기능이 없는 임베디드 어셈블러. 이러한 어셈블러는 동일한 구문을 사용하여 어셈블리 명령어를 처리합니다. 이 설명서에서는 이 두 어셈블러에 대해 설명하지 않습니다. 인라인 및 임베디드 어셈블러에 대한 자세한 내용은 *개발자 설명서*에서 *C, C++ 및 어셈블리 언어 조합* 장을 참조하십시오.

이전 릴리스의 RVCT에서 업그레이드하는 경우에는 핵심 설명서에서 이 릴리스의 새 기능과 향상된 기능에 대한 세부 정보를 읽어 보십시오.

1.1.1 ARM 어셈블리 언어

이전 버전의 ARM 및 Thumb 어셈블러 언어가 최신 ARM 및 Thumb 어셈블러 언어로 대체되었습니다. 이 언어를 UAL (*통합 어셈블러 언어*) 이라고도 합니다.

UAL을 사용하여 작성한 코드는 ARM, Thumb-2 또는 Thumb-2 이전 Thumb에 대해 어셈블할 수 있습니다. 사용할 수 없는 명령을 사용하면 어셈블러에서 오류가 발생합니다.

1.1.2 Wireless MMX 기술 명령어

어셈블러는 PXA270 프로세서에서 실행할 코드를 어셈블할 수 있도록 Intel® Wireless MMX™ 기술 명령어를 지원합니다. 이 프로세서는 MMX 확장을 사용하여 ARMv5TE 아키텍처를 구현합니다. RVCT에는 Wireless MMX 기술 제어 및 SIMD (*Single Instruction Multiple Data*) 데이터 레지스터에 대한 지원뿐 아니라 Wireless MMX 기술 개발을 위한 새로운 지시어가 포함되어 있습니다. 여기에는 로드 및 저장 명령어에 대한 향상된 지원도 포함되어 있습니다. Wireless MMX 기술 지원에 대한 자세한 내용은 6장 *Wireless MMX 기술 명령어*를 참조하십시오.

1.1.3 NEON 기술

ARM NEON™ 기술은 Advanced SIMD 아키텍처 확장의 구현으로, 고급 미디어 및 신호 처리 응용 프로그램과 임베디드 프로세서를 대상으로 하는 64/128비트 복합 SIMD 기술입니다. 이 기술은 ARM 코어의 일부로 구현되지만 자체 실행 파이프라인이 있으며 ARM 코어 레지스터 뱅크와는 별도의 레지스터 뱅크가 있습니다.

NEON 명령어는 ARM과 Thumb-2 코드에서 모두 사용할 수 있습니다. NEON에 대한 자세한 내용은 5장 *NEON 및 VFP 프로그래밍*을 참조하십시오.

1.1.4 예제 사용

이 설명서에서는 RealView Development Suite와 함께 제공되는 예제를 참조합니다. 이러한 예제는 주 예제 디렉토리인 *install_directory*\RVDS\Examples에 있습니다. 제공된 예제에 대한 요약 정보는 *RealView Development Suite 시작 설명서*를 참조하십시오.

2장
ARM 어셈블리 언어 작성

이 장에서는 ARM® *어셈블리 언어*를 작성하는 일반 원칙에 대해 소개합니다. 여기에는 다음 단원이 포함되어 있습니다.

- 2-2페이지의 *소개*
- 2-4페이지의 *ARM 아키텍처 개요*
- 2-13페이지의 *어셈블리 언어 모듈의 구조*
- 2-20페이지의 *조건부 실행*
- 2-28페이지의 *레지스터에 상수 로드*
- 2-36페이지의 *레지스터에 주소 로드*
- 2-42페이지의 *다중 레지스터 로드 및 저장 명령어*
- 2-49페이지의 *매크로 사용*
- 2-53페이지의 *기호 버전 추가*
- 2-54페이지의 *프레임 지시어 사용*
- 2-55페이지의 *어셈블리 언어 변경 사항*

2.1 소개

이 장에서는 ARM 어셈블리 언어 모듈을 작성하는 방법에 대한 기본적이고 실제적인 이해를 제공하며 ARM 어셈블러 (armasm) 에서 제공하는 기능에 대해서도 설명합니다.

이 장에서는 Thumb®, Thumb-2, NEON™, VFP 또는 Wireless MMX 명령어 세트에 대해 자세히 설명하지 않습니다. 이러한 명령어 세트에 대한 자세한 내용은 다음을 참조하십시오.

- 4장 *ARM 및 Thumb 명령어*
- 5장 *NEON 및 VFP 프로그래밍*
- 6장 *Wireless MMX 기술 명령어*

자세한 내용은 *ARM 아키텍처 참조 문서*를 참조하십시오.

RVCT 버전 2.1 이하 버전에서 허용되는 ARM 및 Thumb 어셈블리 언어에 익숙한 프로그래머가 간편하게 사용할 수 있도록 이 장에는 이러한 언어와 최신 버전의 ARM 어셈블리 언어 간 차이점을 설명하는 단원이 포함되어 있습니다. 자세한 내용은 2-55페이지의 *어셈블리 언어 변경 사항*을 참조하십시오.

2.1.1 코드 예제

이 장에서는 여러 가지 코드 예제를 제공합니다. 이러한 코드 예제의 대부분은 *install_directory*\RVDS\Examples\...\asm 디렉토리에 있습니다.

다음 단계에 따라 어셈블리 언어 파일을 빌드하고 링크합니다.

1. 명령 프롬프트에 armasm --debug *filename*.s를 입력하여 파일을 어셈블하고 디버그 테이블을 생성합니다.
2. 다음으로 armlink *filename*.o -o *filename* 을 입력하여 객체 파일을 링크하고 ELF 실행 이미지를 생성합니다.

이미지를 실행하고 디버깅하려면 RealView ISS (*RealView Instruction Set Simulator*) 와 같은 적절한 디버그 타겟을 사용하여 RealView 디버거와 같은 호환 디버거로 이미지를 로드합니다.

어셈블러에서 소스 코드를 변환하는 방법을 보려면 다음을 입력하십시오.

```
fromelf -c filename.o
```

armlink에 대한 자세한 내용은 *링커 사용 설명서*를, fromelf에 대한 자세한 내용은 *유틸리티 설명서*를 참조하십시오.

ELF 및 DWARF에 대한 자세한 내용은 www.infocenter.arm.com에서 ABI (*응용 프로그램 바이너리 인터페이스*) 설명서를 참조하십시오.

2.2 ARM 아키텍처 개요

이 단원에서는 ARM 아키텍처 개요에 대해 간략히 설명합니다.

ARM 프로세서는 로드 및 저장 아키텍처를 구현하므로 일반적인 RISC 프로세서입니다. 로드 및 저장 명령어만 메모리에 액세스할 수 있고 데이터 처리 명령어는 레지스터 내용에 대해서만 작동합니다.

이 단원에서는 다음 내용을 설명합니다.

- *아키텍처 버전*
- *ARM, Thumb, Thumb-2 및 Thumb-2EE 명령어 세트*
- 2-5페이지의 *ARM, Thumb 및 ThumbEE 상태*
- 2-6페이지의 *프로세서 모드*
- 2-7페이지의 *레지스터*
- 2-9페이지의 *명령어 세트 개요*
- 2-11페이지의 *명령어 기능*

2.2.1 아키텍처 버전

이 설명서에 나오는 정보와 예제는 ARMv4 이상을 구현하는 프로세서를 사용하고 있다는 가정 하에 작성되었습니다. 이러한 모든 프로세서에는 32비트 주소 지정 범위가 있습니다. 다양한 아키텍처 버전에 대한 자세한 내용은 *ARM 아키텍처 참조 문서*를 참조하십시오.

2.2.2 ARM, Thumb, Thumb-2 및 Thumb-2EE 명령어 세트

ARM 명령어 세트는 포괄적인 범위의 연산을 제공하는 32비트 명령어 세트입니다.

ARMv4T 이상에서는 Thumb 명령어 세트라는 16비트 명령어 세트를 정의합니다. 32비트 ARM 명령어 세트의 기능 대부분을 사용할 수 있지만 일부 연산에는 추가 명령어가 필요합니다. Thumb 명령어 세트에서는 성능이 저하되는 대신 향상된 코드 밀도를 제공합니다.

ARMv6T2는 Thumb 명령어 세트가 크게 향상된 Thumb-2를 정의합니다. Thumb-2는 ARM 명령어 세트와 거의 동일한 기능을 제공합니다. 또한 16비트 명령어와 32비트 명령어를 모두 포함하며 성능은 ARM 코드와 비슷하지만 코드 밀도는 Thumb 코드와 비슷합니다.

ARMv7에서는 Thumb-2EE (Thumb-2 Execution Environment) 를 정의합니다. Thumb-2EE 명령어 세트는 Thumb-2를 기반으로 하지만, 동적으로 생성되는 코드, 즉 실행 직전이나 실행 중에 장치에서 컴파일되는 코드에 보다 적합한 타겟이 되도록 몇 가지 사항을 변경하고 추가한 것입니다.

자세한 내용은 2-9페이지의 *명령어 세트 개요*를 참조하십시오.

2.2.3 ARM, Thumb 및 ThumbEE 상태

ARM 명령어를 실행하는 프로세서는 *ARM 상태*에서 작동하고 Thumb 명령어를 실행하는 프로세서는 *Thumb 상태*에서 작동합니다. ThumbEE 명령어를 실행하는 프로세서는 *ThumbEE 상태*에서 작동합니다. 프로세서는 *Jazelle 상태*라는 다른 상태에서도 작동할 수 있습니다.

특정 상태의 프로세서는 다른 명령어 세트의 명령어를 실행할 수 없습니다. 예를 들어 ARM 상태의 프로세서는 Thumb 명령어를 실행할 수 없고 Thumb 상태의 프로세서는 ARM 명령어를 실행할 수 없습니다. 따라서 프로세서에서 현재 상태와 맞지 않는 명령어 세트의 명령어를 수신하지 않도록 해야 합니다.

대부분의 ARM 프로세서는 항상 ARM 상태에서 코드 실행을 시작합니다. 그러나 일부 프로세서의 경우 Thumb 코드만 실행할 수 있거나 Thumb 상태에서 시작하도록 구성할 수 있습니다.

상태 변경

각 명령어 세트에는 프로세서 상태를 변경하는 명령어가 포함되어 있습니다.

ARM 상태와 Thumb 상태 간에 전환하려면 `ARM` 또는 `THUMB` 지시어를 통해 올바른 op 코드를 생성하도록 어셈블러 모드를 전환해야 합니다. Thumb-2EE 코드를 생성하려면 `THUMBX`를 사용합니다. `CODE32` 및 `CODE16`을 사용하는 어셈블러 코드는 어셈블러에서 계속 어셈블할 수 있지만 새 코드에 대해서는 `ARM` 및 `THUMB`을 사용하는 것이 좋습니다.

자세한 내용은 7-63페이지의 *명령어 세트 및 구문 선택 지시어*를 참조하십시오.

2.2.4 프로세서 모드

ARM 프로세서에서는 아키텍처 버전에 따라 다양한 프로세서 모드를 지원합니다 (표 2-1 참조).

참고

ARMv6-M 및 ARMv7-M은 다른 ARM 프로세서와 동일한 모드를 지원하지 않습니다. 이 단원은 ARMv6-M 및 ARMv7-M에는 적용되지 않습니다.

표 2-1 ARM 프로세서 모드

프로세서 모드	아키텍처	모드 번호
사용자	모두	0b10000
FIQ - 고속 인터럽트 요청	모두	0b10001
IRQ - 인터럽트 요청	모두	0b10010
관리자	모두	0b10011
중단	모두	0b10111
정의되지 않음	모두	0b11011
시스템	ARMv4 이상	0b11111
모니터	보안 확장에만 해당	0b10110

사용자 모드를 제외한 모든 모드를 *권한* 모드라고 합니다. 이러한 모드에서는 시스템 리소스에 대한 모든 액세스 권한이 제공되고 모드를 자유롭게 변경할 수 있습니다.

작업 보호가 필요한 응용 프로그램은 일반적으로 사용자 모드에서 실행됩니다. 일부 임베디드 응용 프로그램은 전적으로 관리자 또는 시스템 모드에서 실행될 수 있습니다.

사용자 모드 이외의 모드에서는 예외를 처리하거나 권한 있는 리소스에 액세스합니다 (*개발자 설명서*의 6장 *프로세서 예외 처리* 참조).

2.2.5 레지스터

ARM 프로세서에는 37개의 레지스터가 있습니다. 이러한 레지스터는 뱅크의 일부가 서로 겹치는 방식으로 정렬됩니다. 레지스터 뱅크는 프로세서 모드에 따라 서로 다릅니다. 뱅크 레지스터는 신속한 컨텍스트 전환을 통해 프로세서 예외와 권한 있는 연산을 처리할 수 있도록 합니다. 레지스터가 뱅크되는 방법에 대한 자세한 내용은 *ARM 아키텍처 참조 문서*를 참조하십시오.

다음 레지스터를 사용할 수 있습니다.

- *30개의 범용 32비트 레지스터*
- *PC (프로그램 카운터)*
- 2-8페이지의 *APSR (응용 프로그램 상태 레지스터)*
- 2-9페이지의 *SPSR (저장된 프로그램 상태 레지스터)*

30개의 범용 32비트 레지스터

15개의 범용 레지스터 (r0 ~r12, sp, lr) 는 현재 프로세서 모드에 따라 한 번에 하나씩 표시됩니다.

sp (r13) 는 *스택 포인터*입니다. C 및 C++ 컴파일러에서는 항상 sp를 스택 포인터로 사용합니다. Thumb-2에서는 sp가 스택 포인터로 엄격하게 정의되기 때문에 sp가 사용될 경우 스택 조작에 도움이 되지 않는 여러 명령어는 예상할 수 없는 결과를 낼 수 있습니다. sp는 범용 레지스터로 사용하지 않는 것이 좋습니다.

사용자 모드에서 lr (r14) 은 *링크 레지스터*로 사용되어 서브루틴 호출이 수행될 때 복귀 주소를 저장합니다. 또한 반환 주소가 스택에 저장될 경우 범용 레지스터로 사용될 수도 있습니다.

예외 처리 모드에서 lr은 예외 복귀 주소나 서브루틴 복귀 주소 (서브루틴 호출이 예외 내에서 실행될 경우) 를 저장합니다. 또한 복귀 주소가 스택에 저장될 경우 범용 레지스터로 사용될 수도 있습니다.

PC (*프로그램 카운터*)

프로그램 카운터는 pc (r15) 로 액세스되며, ARM 상태에 있는 각 명령어에 대해 1워드 (4바이트) 씩 증가하거나 Thumb 상태에서 실행되는 명령어의 크기만큼 증가합니다. 분기 명령어는 대상 주소를 pc로 로드합니다. 또한 데이터 연산 명령어를 사용하면 PC를 직접 로드할 수도 있습니다. 예를 들어 서브루틴에서 복귀하려면 다음을 사용하여 링크 레지스터를 PC로 복사할 수 있습니다.

```
MOV pc,lr
```

실행하는 동안 pc는 현재 실행되는 명령어의 주소를 포함하지 않습니다. 현재 실행되는 명령어의 주소는 일반적으로 ARM의 경우 pc-8이거나 Thumb의 경우 pc-4입니다.

APSR (*응용 프로그램 상태 레지스터*)

APSR은 ALU (*산술 논리 단위*) 상태 플래그의 복사본을 포함하며, 조건부 명령어의 실행 여부를 결정하는 데 사용됩니다. 자세한 내용은 2-20페이지의 *조건부 실행*을 참조하십시오.

ARMv5TE 및 ARMv6 이상에서는 APSR에 Q 플래그도 포함됩니다 (2-21페이지의 *ALU 상태 플래그* 참조).

ARMv6 이상에서는 APSR에 GE 플래그도 포함됩니다 (4-98페이지의 *병렬 더하기 및 빼기* 참조).

이러한 플래그에는 `MSR` 및 `MRS` 명령어를 사용하여 모든 모드에서 액세스할 수 있습니다. 자세한 내용은 4-134페이지의 *MRS* 및 4-136페이지의 *MSR*을 참조하십시오.

CPSR (*현재 프로그램 상태 레지스터*)

CPSR에는 다음이 포함됩니다.

- APSR 플래그
- 현재 프로세서 모드
- 인터럽트 비활성화 플래그
- 현재 프로세서 상태 (ARM, Thumb, ThumbEE 또는 Jazelle)
- IT 블록의 실행 상태 비트

실행 상태 비트는 IT 블록의 조건부 실행을 제어하며 (4-118페이지의 *IT* 참조) ARMv6TE 이상에서만 사용할 수 있습니다.

APSR 플래그만 모든 모드에서 액세스할 수 있고 CPSR의 나머지 비트에는 `MSR` 및 `MRS` 명령어를 사용하여 권한 모드에서만 액세스할 수 있습니다. 자세한 내용은 4-134페이지의 *MRS* 및 4-136페이지의 *MSR*을 참조하십시오.

SPSR (*저장된 프로그램 상태 레지스터*)

SPSR은 예외가 발생할 때 CPSR을 저장하는 데 사용됩니다. 각 예외 처리 모드에서는 하나의 SPSR에 액세스할 수 있습니다. 사용자 모드와 시스템 모드는 예외 처리 모드가 아니므로 SPSR을 가지고 있지 않습니다. 자세한 내용은 개발자 설명서에서 6장 *프로세서 예외 처리*를 참조하십시오.

2.2.6 명령어 세트 개요

모든 ARM 명령어의 길이는 32비트입니다. 명령어는 워드로 정렬된 채 저장되므로 ARM 상태에서 명령어 주소의 최하위 2비트는 항상 0입니다.

Thumb, Thumb-2 및 Thumb-2EE 명령어의 길이는 16비트 또는 32비트입니다. 명령어는 하프워드로 정렬된 채 저장됩니다. 일부 명령어는 주소의 최하위 비트를 사용하여 분기되는 코드가 Thumb 코드인지 아니면 ARM 코드인지를 확인합니다.

Thumb-2가 도입되기 전에 Thumb 명령어 세트는 ARM 명령어 세트의 제한된 하위 기능 세트로 제한되었으며 거의 모든 Thumb 명령어가 16비트였습니다. Thumb-2 명령어 세트 기능은 ARM 명령어 세트 기능과 거의 같습니다.

ARMv6 이상에서는 모든 ARM 및 Thumb 명령어가 리틀엔디안입니다.

ARM 및 Thumb 명령어 구문에 대한 자세한 내용은 4장 *ARM 및 Thumb 명령어*를 참조하십시오.

ARM 및 Thumb 명령어는 다음과 같은 여러 기능 그룹으로 분류할 수 있습니다.

- *분기 및 제어 명령어*
- 2-10페이지의 *데이터 처리 명령어*
- 2-10페이지의 *레지스터 로드 및 저장 명령어*
- 2-10페이지의 *다중 레지스터 로드 및 저장 명령어*
- 2-10페이지의 *상태 레지스터 액세스 명령어*
- 2-10페이지의 *보조 프로세서 명령어*

분기 및 제어 명령어

이러한 명령어를 사용하면 다음을 수행할 수 있습니다.

- 서브루틴으로 분기
- 루프 형성을 위한 역방향 분기
- 조건부 구조체에서 정방향 분기

- 다음의 최대 네 개 명령어를 분기 없는 조건 명령어로 지정
- ARM과 Thumb 사이에서 프로세서 상태 전환

데이터 처리 명령어

이러한 명령어는 범용 레지스터에서 작동하며, 두 레지스터의 내용에 대해 더하기, 빼기 또는 비트 단위 논리와 같은 연산을 수행하고 결과를 세 번째 레지스터에 배치합니다. 이러한 명령어는 단일 레지스터 내의 값을 연산하거나 레지스터 값과 명령어 내에 제공된 상수 (*즉치값*) 를 연산할 수도 있습니다.

Long 곱하기 명령어는 두 개의 레지스터에 64비트 결과를 제공합니다.

레지스터 로드 및 저장 명령어

이러한 명령어는 단일 레지스터 값을 메모리에서 로드하거나 메모리에 저장하며, 32비트 워드, 16비트 하프워드 또는 8비트 부호 없는 바이트를 로드하거나 저장할 수 있습니다. 바이트 및 하프워드 로드는 32비트 레지스터를 채우도록 부호 확장 또는 0 확장됩니다.

이외에도 64비트 더블워드 값을 두 개의 32비트 레지스터에 로드하거나 저장할 수 있는 몇 가지 명령어가 정의되어 있습니다.

다중 레지스터 로드 및 저장 명령어

이러한 명령어는 범용 레지스터의 하위 세트를 메모리에서 로드하거나 메모리에 저장합니다. 이러한 명령어에 대한 자세한 내용은 2-42페이지의 *다중 레지스터 로드 및 저장 명령어*를 참조하십시오.

상태 레지스터 액세스 명령어

이러한 명령어는 상태 레지스터와 범용 레지스터 간에 내용을 이동합니다.

보조 프로세서 명령어

이러한 명령어는 ARM 아키텍처를 확장하는 일반적인 방법을 지원합니다.

2.2.7 명령어 기능

이 단원에는 다음 소단원이 포함되어 있습니다.

- *조건부 실행*
- *레지스터 액세스*
- 2-12페이지의 *인라인 배럴 시프터에 액세스*

조건부 실행

거의 모든 ARM 명령어는 APSR의 ALU 상태 플래그 값에 대해 조건부로 실행될 수 있습니다. 일련의 명령어에 동일한 조건을 적용할 경우 성능이 향상될 수 있지만 반드시 분기를 사용하여 조건부 명령어를 건너뛰어야 하는 것은 아닙니다.

Thumb-2 이전 프로세서의 Thumb 상태에서 조건부 실행을 위한 유일한 메커니즘은 조건부 분기뿐입니다. 대부분의 데이터 처리 명령어는 ALU 플래그를 업데이트합니다. 일반적으로 명령어가 ALU 플래그 상태를 업데이트할지 여부는 지정할 수 없습니다.

Thumb-2는 IT (If-Then) 명령어와 동일한 ALU 플래그를 사용하여 조건부 실행을 위한 대체 메커니즘을 제공합니다. IT는 최대 네 개까지 다음과 같은 명령어의 조건부 실행을 제공하는 16비트 명령어입니다. 이외에도 조건부 실행을 위한 추가 메커니즘을 제공하는 여러 명령어가 있습니다.

ARM 및 Thumb-2 코드에서는 데이터 처리 명령어가 ALU 플래그를 업데이트할지 여부를 지정할 수 있습니다. 한 명령어로 설정된 플래그를 사용하여 중간에 여러 플래그 비설정 명령어가 있는 경우에도 다른 명령어의 실행을 제어할 수 있습니다.

자세한 내용은 2-20페이지의 *조건부 실행*을 참조하십시오.

레지스터 액세스

ARM 상태에서는 모든 명령어가 r0 ~ r14에 액세스할 수 있으며, 이중 대부분의 명령어가 pc (r15) 에 액세스할 수 있습니다. MRS 및 MSR 명령어는 상태 레지스터의 내용을 범용 레지스터로 이동할 수 있고 이 레지스터에서 일반 데이터 처리 연산을 사용하여 내용을 조작할 수 있습니다. 자세한 내용은 4-134페이지의 *MRS* 및 4-136페이지의 *MSR*을 참조하십시오.

Thumb-2 프로세서의 Thumb 상태에서는 유용하지 않을 경우 sp 및 pc에 대한 일부 액세스가 허용되지 않는다는 점을 제외하고 동일한 기능을 제공합니다.

Thumb-2 이전 프로세서에서는 대부분의 Thumb 명령어가 r0 ~ r7에만 액세스할 수 있고 적은 수의 명령어만 r8 ~ r15에 액세스할 수 있습니다. 레지스터 r0 ~ r7은 Lo 레지스터라고 하고 레지스터 r8 ~ r15는 Hi 레지스터라고 합니다.

인라인 배럴 시프터에 액세스

ARM 산술 논리 단위에는 시프트 및 회전 연산을 수행할 수 있는 32비트 배럴 시프터가 있습니다. 대부분의 ARM 및 Thumb-2 데이터 처리와 단일 레지스터 데이터 전송 명령어에 대한 두 번째 피연산자는 데이터 처리나 데이터 전송이 실행되기 전에 명령어의 일부로 시프트할 수 있습니다. 이 기능은 다음을 지원하며 이에 제한되지 않습니다.

- 스케일된 주소 지정
- 상수로 곱하기
- 상수 생성

배럴 시프터를 사용하여 상수를 생성하는 방법에 대한 자세한 내용은 *2-28페이지의 레지스터에 상수 로드*를 참조하십시오.

Thumb-2 명령어는 ARM 명령어와 거의 동일한 배럴 시프터에 대한 액세스를 제공합니다.

Thumb2 이전 Thumb 명령어 세트는 별도의 명령어를 통해서만 배럴 시프터에 액세스할 수 있습니다.

2.3 어셈블리 언어 모듈의 구조

어셈블리 언어는 ARM 어셈블러 (armasm) 에서 구문 분석하고 어셈블하여 개체 코드를 생성하는 언어입니다. 기본적으로 어셈블러에서는 소스 코드를 ARM 어셈블리 언어로 작성하도록 요구합니다.

armasm에서는 이전 버전의 ARM 어셈블리 언어로 작성된 소스 코드를 실행할 수 있습니다. 이 경우 이를 알리지 않아도 됩니다.

또한 armasm에서는 UAL 이전 Thumb 어셈블리 언어로 작성된 소스 코드를 실행할 수도 있습니다. 이 경우 --16 명령 행 옵션이나 소스 코드의 CODE16 지시어 사용하여 armasm에 이를 알려야 합니다. UAL 이전 Thumb 어셈블리 언어는 Thumb-2 명령어를 지원하지 않습니다.

이 단원에서는 다음 내용을 설명합니다.

- *어셈블리 언어 소스 파일 레이아웃*
- 2-16페이지의 *ARM 어셈블리 언어 모듈 예제*
- 2-18페이지의 *서브루틴 호출*

2.3.1 어셈블리 언어 소스 파일 레이아웃

어셈블리 언어의 일반적인 소스 행 형식은 다음과 같습니다.

{*label*} {*instruction*|*directive*|*pseudo-instruction*} {;*comment*}

참고

레이블이 없는 경우에도 명령어, 의사 명령어 및 지시어 앞에는 공백이나 탭이 한 칸 있어야 합니다.

일부 지시어에서는 레이블을 사용할 수 없습니다.

소스 행의 세 부분은 모두 선택적입니다. 빈 행을 사용하여 코드를 읽기 쉽게 만들 수 있습니다.

대소문자 규칙

명령어 니모닉, 지시어 및 기호 레지스터 이름은 대문자나 소문자 중 하나로만 작성할 수 있고 대소문자를 모두 사용하여 작성할 수는 없습니다.

행 길이

소스 파일을 읽기 쉽게 만들려면 행 끝에 백슬래시 문자 (\) 를 배치하여 긴 소스 행을 여러 행으로 분할할 수 있습니다. 백슬래시 뒤에는 공백과 탭을 포함하여 다른 문자가 오면 안 됩니다. 어셈블러에서는 백슬래시와 행 끝 시퀀스를 공백으로 처리합니다.

참고

백슬래시와 행 끝 시퀀스를 따옴표로 묶인 문자열 내에 사용하면 안 됩니다.

백슬래시를 사용하는 확장을 포함하여 행 길이 제한은 4095자입니다.

레이블

레이블은 주소를 나타내는 심볼입니다. 레이블로 지정된 주소는 어셈블리 동안 계산됩니다.

어셈블러에서는 레이블이 정의된 섹션의 원점을 기준으로 레이블 주소를 계산합니다. 같은 섹션 내에 있는 레이블에 대한 참조는 오프셋을 더하거나 뺀 PC를 사용할 수 있습니다. 이 작업을 *프로그램 상대 주소 지정*이라고 합니다.

다른 섹션의 레이블 주소는 링커가 각 섹션에 대해 특정 메모리 위치를 할당한 경우 링크 타임에 계산됩니다.

지역 레이블

지역 레이블은 레이블의 하위 클래스로, 0 ~ 99 범위에 있는 숫자로 시작합니다. 다른 레이블과 달리 지역 레이블은 여러 번 정의할 수 있습니다. 지역 레이블은 매크로를 통해 레이블을 생성하는 경우 유용합니다. 어셈블러에서는 지역 레이블에 대한 참조를 찾으면 이 참조를 지역 레이블의 주변 인스턴스에 링크합니다.

지역 레이블 범위는 `AREA` 지시어로 제한되지 않습니다. `ROUT` 지시어를 사용하면 범위를 보다 엄격하게 제한할 수 있습니다.

다음에 대한 자세한 내용은 3-31페이지의 *지역 레이블*을 참조하십시오.

- 지역 레이블 선언 구문
- 어셈블러에서 지역 레이블에 대한 참조를 해당 레이블에 연결하는 방법

주석

행의 첫 번째 세미콜론은 문자열 상수 내부에 세미콜론이 나타나는 위치를 제외하고 주석의 시작을 표시합니다. 행 끝은 주석의 끝입니다. 주석만으로 하나의 유효한 행을 구성할 수 있습니다. 어셈블러에서는 모든 주석을 무시합니다.

상수

상수는 다음 중 하나일 수 있습니다.

숫자 다음 형식의 숫자 상수가 허용됩니다.

- 10진수 (예: `123`)
- 16진수 (예: `0x7B`)
- *n_xxx*.다음은 이 요소에 대한 설명입니다.
  - *n* 2와 9 사이의 기수입니다.
  - *xxx* 해당 기수에 있는 숫자입니다.
- 부동 소수점 (예: 0.02, 123.0 또는 3.14159)

부동 소수점 숫자는 시스템에 부동 소수점을 사용하는 VFP 또는 NEON이 있는 경우에만 사용할 수 있습니다.

부울 bool 상수 `TRUE` 및 `FALSE`는 `{TRUE}` 및 `{FALSE}`로 작성되어야 합니다.

문자 문자 상수는 표준 C 이스케이프 문자를 사용하는 이스케이프 문자나 단일 문자를 묶는 열고 닫는 작은따옴표로 구성됩니다.

문자열 문자열은 문자와 공백을 묶는 열고 닫는 큰따옴표로 구성됩니다. 큰따옴표 또는 달러 기호가 문자열 내에서 리터럴 텍스트 문자로 사용될 경우에는 해당 문자를 쌍으로 표시되어야 합니다. 예를 들어 문자열에 하나의 `$`가 필요하면 `$$`를 사용해야 합니다. 표준 C 이스케이프 시퀀스는 문자열 상수 내에서 사용할 수 있습니다.

2.3.2 ARM 어셈블리 언어 모듈 예제

예제 2-1에서는 어셈블리 언어 모듈의 일부 핵심 구성요소를 보여 줍니다. 이 예제는 ARM 어셈블리 언어로 작성되었으며, 주 예제 디렉토리인 *install_directory*\RVDS\Examples에 armex.s로 제공되어 있습니다. 이 예제를 어셈블, 링크 및 실행하는 방법에 대한 자세한 내용은 2-2페이지의 *코드 예제*를 참조하십시오.

이 예제의 구성 부분에 대해서는 다음 단원에서 자세히 설명합니다.

예제 2-1

```
        AREA     ARMex, CODE, READONLY
                                ; Name this block of code ARMex
        ENTRY                   ; Mark first instruction to execute
start
        MOV      r0, #10        ; Set up parameters
        MOV      r1, #3
        ADD      r0, r0, r1     ; r0 = r0 + r1
stop
        MOV      r0, #0x18      ; angel_SWIreason_ReportException
        LDR      r1, =0x20026   ; ADP_Stopped_ApplicationExit
        SVC      #0x123456      ; ARM semihosting (formerly SWI)
        END                     ; Mark end of file
```

ELF 섹션 및 AREA 지시어

ELF *섹션*은 코드 또는 데이터의 독립적이고 명명된 나눌 수 없는 시퀀스입니다. 응용 프로그램을 만들려면 최소한 하나 이상의 코드 섹션이 있어야 합니다.

어셈블리 및 컴파일의 출력에는 다음이 포함될 수 있습니다.

- 하나 이상의 코드 섹션으로, 대개 읽기 전용 섹션입니다.
- 하나 이상의 데이터 섹션으로, 대개 읽기/쓰기 섹션입니다. 이러한 섹션은 *0으로 초기화* (ZI) 될 수 있습니다.

링커는 섹션 배치 규칙에 따라 각 섹션을 프로그램 이미지에 배치합니다. 소스 파일의 인접 섹션이 응용 프로그램 이미지에서도 인접할 필요는 없습니다. 링커가 섹션을 배치하는 방법에 대한 자세한 내용은 *링커 사용 설명서*에서 5장 *스캐터 로딩 설명 파일 사용*을 참조하십시오.

소스 파일에서 AREA 지시어는 섹션의 시작을 표시합니다. 이 지시어는 섹션의 이름을 지정하고 해당 특성을 설정합니다. 특성은 이름 뒤에 배치되고 쉼표로 구분됩니다. AREA 지시어의 구문에 대한 자세한 내용은 7-70페이지의 *AREA*를 참조하십시오.

아무 이름이나 선택할 수 있지만 알파벳 이외의 문자로 시작하는 이름은 막대로 묶어야 합니다. 그렇지 않으면 AREA name missing 오류가 생성됩니다. 예를 들면 |1_DataArea|와 같습니다.

2-16페이지의 예제 2-1에서는 코드를 포함하고 READONLY로 표시되는 ARMex라는 단일 섹션을 정의합니다.

ENTRY 지시어

ENTRY 지시어는 실행되는 첫 번째 명령어를 표시합니다. C 코드를 포함하는 응용 프로그램에서는 진입점이 C 라이브러리 초기화 코드 내에도 포함되고 초기화 코드와 예외 처리기에도 포함됩니다.

응용 프로그램 실행

2-16페이지의 예제 2-1의 응용 프로그램 코드는 10진수 값 10 및 3을 레지스터 r0 및 r1로 로드하는 start 레이블에서 실행을 시작합니다. 이러한 레지스터를 더한 결과는 r0에 배치됩니다.

응용 프로그램 종료

기본 코드가 실행된 후에는 디버거로 제어권이 반환되어 응용 프로그램이 종료됩니다. 이 작업은 ARM 세미호스팅 SVC (기본적으로 0x123456임) 를 다음 매개변수와 함께 사용하여 수행합니다.

- r0은 angel_SWIreason_ReportException (0x18) 과 같습니다.
- r1은 ADP_Stopped_ApplicationExit (0x20026) 와 같습니다.

*RVCT 개발자 설명서*의 8장 *세미호스팅*을 참조하십시오.

END 지시어

이 지시어는 이 소스 파일의 처리를 중지하도록 어셈블러에 지시합니다. 모든 어셈블리 언어 소스 모듈은 별도의 행에서 END로 끝나야 합니다.

2.3.3 서브루틴 호출

서브루틴을 호출하려면 분기 및 링크 명령어를 사용해야 합니다. 구문은 다음과 같습니다.

```
BL  destination
```

여기서 *destination*은 일반적으로 서브루틴의 첫 번째 명령어 레이블입니다.

*destination*은 프로그램 상대 식일 수도 있습니다. 자세한 내용은 4-114페이지의 *B, BL, BX, BLX 및 BXJ*를 참조하십시오.

BL 명령어의 경우

- 링크 레지스터에 반환 주소를 배치합니다.
- PC를 서브루틴의 주소로 설정합니다.

서브루틴 코드가 실행되고 나면 BX lr 명령어를 사용하여 복귀할 수 있습니다. 일반적으로 레지스터 r0 ~ r3은 매개변수를 서브루틴에 전달하는 데 사용되고 r0은 결과를 호출자에게 다시 전달하는 데 사용됩니다.

참고

개별적으로 어셈블되었거나 컴파일된 모듈 간의 호출은 프로시저 호출 표준에서 정의된 제한과 규칙을 준수해야 합니다. 자세한 내용은 *install_directory*\Documentation\Specifications\...에 있는 *Procedure Call Standard for the ARM Architecture* 사양 (aapcs.pdf) 을 참조하십시오.

2-19페이지의 예제 2-2에서는 두 매개변수 값을 더하고 결과를 r0에 반환하는 서브루틴을 보여 줍니다. 이 예제는 주 예제 디렉토리인 *install_directory*\RVDS\Examples에 subrout.s로 제공되어 있습니다. 이 예제를 어셈블, 링크 및 실행하는 방법에 대한 자세한 내용은 2-2페이지의 *코드 예제*를 참조하십시오.

예제 2-2

```
        AREA    subrout, CODE, READONLY     ; Name this block of code
        ENTRY                       ; Mark first instruction to execute
start   MOV     r0, #10             ; Set up parameters
        MOV     r1, #3
        BL      doadd               ; Call subroutine
stop    MOV     r0, #0x18           ; angel_SWIreason_ReportException
        LDR     r1, =0x20026        ; ADP_Stopped_ApplicationExit
        SVC     #0x123456           ; ARM semihosting (formerly SWI)
doadd   ADD     r0, r0, r1          ; Subroutine code
        BX      lr                  ; Return from subroutine
        END                         ; Mark end of file
```

2.4 조건부 실행

ARM 상태 및 Thumb-2가 포함된 프로세서의 Thumb 상태에서는 대부분의 데이터 처리 명령어에 연산 결과에 따라 APSR (*응용 프로그램 상태 레지스터*) 의 ALU 상태 플래그를 업데이트하는 옵션이 있습니다. 일부 명령어는 모든 플래그를 업데이트하고 일부 명령어는 하위 세트만 업데이트합니다. 플래그가 업데이트되지 않으면 원래 값이 저장됩니다. 각 명령어의 설명 부분에는 해당 명령어가 플래그에 주는 영향이 자세히 나와 있습니다. 실행되지 않는 조건부 명령어는 플래그에 영향을 주지 않습니다.

Thumb-2 이전 프로세서의 Thumb 상태에서는 대부분의 데이터 처리 명령어가 ALU 상태 플래그를 자동으로 업데이트합니다. 플래그를 변경 및 업데이트하지 않는 옵션은 없습니다. 이외의 명령어는 플래그를 업데이트할 수 없습니다.

ARM 상태 및 Thumb-2가 포함된 프로세서의 Thumb 상태에서는 다음과 같이 다른 명령어에 설정된 ALU 상태 플래그에 따라 조건부로 명령어를 실행할 수 있습니다.

- 플래그를 업데이트한 명령어 바로 다음에 실행합니다.
- 플래그를 업데이트하지 않은 임의 개수의 간섭 명령어 다음에 실행합니다.

거의 모든 ARM 명령어는 APSR의 ALU 상태 플래그의 상태에 대해 조건부로 실행될 수 있습니다. 명령어를 조건부 명령어로 만들기 위해 명령어에 추가하는 접미사 목록은 2-21페이지의 표 2-2를 참조하십시오.

Thumb 상태에서는 조건부 분기를 통해 조건부 실행을 위한 메커니즘이 제공됩니다.

Thumb-2가 포함된 프로세서의 Thumb 상태에서는 특수 `IT` (If-Then) 명령어를 사용하여 명령어를 조건부 명령어로 만들 수도 있습니다. 또한 `CBZ` (0인 경우 조건부 분기) 및 `CBNZ` 명령어를 사용하여 레지스터의 값을 0과 비교하고 결과에 대해 분기할 수도 있습니다.

이 단원에서는 다음 내용을 설명합니다.

- 2-21페이지의 *ALU 상태 플래그*
- 2-21페이지의 *조건부 실행*
- 2-23페이지의 *조건부 실행 사용*
- 2-23페이지의 *조건부 실행 사용 예제*
- 2-27페이지의 *Q 플래그*

2.4.1 ALU 상태 플래그

APSR에는 다음 ALU 상태 플래그가 포함되어 있습니다.

N 연산 결과가 음수인 경우 설정합니다.

Z 연산 결과가 0인 경우 설정합니다.

C 연산 결과가 carry인 경우 설정합니다.

V 연산 결과가 오버플로인 경우 설정합니다.

carry는 다음과 같은 경우에 발생합니다.

- 더하기의 결과가 2^{32}보다 크거나 같은 경우
- 빼기 결과가 양수이거나 0인 경우
- 이동 또는 논리 명령어의 인라인 배럴 시프터 작업 결과

오버플로는 더하기, 빼기 또는 비교의 결과가 2^{31}보다 크거나 같거나 -2^{31}보다 작은 경우 발생합니다.

2.4.2 조건부 실행

조건부로 실행될 수 있는 명령어에는 구문 설명에 {*cond*}로 표시되는 선택적 조건 코드가 있습니다. 이 조건은 ARM 명령어에 인코딩되며, Thumb-2 명령어의 경우 위의 `IT` 명령어에 인코딩됩니다. 조건 코드가 포함된 명령어는 APSR의 조건 코드 플래그가 지정된 조건을 충족하는 경우에만 실행됩니다. 표 2-2에서는 사용할 수 있는 조건 코드를 보여 줍니다.

Thumb-2 이전 프로세서의 Thumb 상태에서 {*cond*} 필드는 특정 분기 명령어에만 허용됩니다.

표 2-2에서는 조건 코드 접미사와 `N`, `Z`, `C` 및 `V` 플래그 간의 관계도 보여 줍니다.

표 2-2 조건 코드 접미사

접미사	플래그	의미
`EQ`	`Z` 세트	같음
`NE`	`Z` 지우기	같지 않음
`CS or HS`	`C` 세트	높거나 같음 (부호 없는 >=)
`CC or LO`	`C` 지우기	보다 낮음 (부호 없는 <)
`MI`	`N` 세트	음수

표 2-2 조건 코드 접미사 (계속)

접미사	플래그	의미
PL	N 지우기	양수 또는 0
VS	V 세트	오버플로
VC	V 지우기	오버플로 없음
HI	C 설정 및 Z 지우기	보다 높음 (부호 없는 >)
LS	C 지우기 또는 Z 설정	낮거나 같음 (부호 없는 <=)
GE	N 및 V 같음	부호 있는 >=
LT	N 및 V 다름	부호 있는 <
GT	Z 지우기, N 및 V 같음	부호 있는 >
LE	Z 설정, N 및 V 다름	부호 있는 <=
AL	Any	항상. 이 접미사는 대개 생략됩니다.

예제 2-3에서는 조건부 실행 예제를 보여 줍니다.

예제 2-3

```
ADD     r0, r1, r2    ; r0 = r1 + r2, don't update flags
ADDS    r0, r1, r2    ; r0 = r1 + r2, and update flags
ADDSCS  r0, r1, r2    ; If C flag set then r0 = r1 + r2, and update flags
CMP     r0, r1        ; update flags based on r0-r1.
```

2.4.3 조건부 실행 사용

ARM 명령어의 조건부 실행을 사용하여 코드에서 분기 명령어 수를 줄일 수 있습니다. 이렇게 하면 코드 밀도가 향상됩니다. Thumb-2의 IT 명령어도 이와 비슷하게 향상되었습니다.

분기 명령어는 프로세서 사이클을 많이 사용합니다. 분기 예상 하드웨어가 없는 ARM 프로세서에서 분기 명령어는 분기가 생성될 때마다 프로세서 파이프라인을 다시 채우기 위해 세 개의 프로세서 주기를 사용합니다.

일부 ARM 프로세서 (예: ARM10™ 및 StrongARM®) 에는 분기 예상 하드웨어가 있습니다. 이러한 프로세서를 사용하는 시스템에서는 잘못된 분기 예상이 있을 경우 파이프라인을 플러시하고 다시 채워야 합니다.

2.4.4 조건부 실행 사용 예제

이 예제에서는 *최대 공약수* (gcd) 알고리즘 (Euclid) 의 두 가지 구현을 사용합니다. 이 예제에서는 조건부 실행을 사용하여 코드 밀도와 실행 속도를 향상시키는 방법을 보여 줍니다. 실행 속도에 대한 자세한 분석은 ARM7™ 프로세서에만 적용되고 코드 밀도 계산은 모든 ARM 프로세서에 적용됩니다.

C에서는 이 알고리즘을 다음과 같이 표시할 수 있습니다.

```
int gcd (int a, int b)
{
    while (a != b)
      {
        if (a > b)
            a = a - b;
        else
            b = b - a;
      }
    return a;
}
```

조건부 분기 실행만 포함된 gcd 함수는 다음과 같은 방법으로 구현할 수 있습니다.

```
gcd     CMP     r0, r1
        BEQ     end
        BLT     less
        SUBS    r0, r0, r1  ; could be SUB r0, r0, r1 for ARM
        B       gcd
less
```

```
        SUBS     r1, r1, r0  ; could be SUB r1, r1, r0 for ARM
        B        gcd
end
```

분기 수 때문에 코드 길이는 명령어 일곱 개입니다. 분기가 생성될 때마다 프로세서는 파이프라인을 다시 채우고 새 위치에서 계속 실행되어야 합니다. 다른 명령어와 실행되지 않은 분기는 각각 단일 주기를 사용합니다.

ARM 명령어 세트의 조건부 실행 기능을 사용하면 다음과 같이 네 개의 명령어만으로 gcd 함수를 구현할 수 있습니다.

```
gcd
        CMP      r0, r1
        SUBGT    r0, r0, r1
        SUBLE    r1, r1, r0
        BNE      gcd
```

코드 크기를 향상시킬 뿐 아니라 대부분의 경우 이 코드가 빠르게 실행됩니다. 표 2-3 및 2-25페이지의 표 2-4에서는 r0이 1이고 r1이 2인 경우 각 구현에서 사용하는 사이클 수를 보여 줍니다. 이 경우 분기를 모든 명령어의 조건부 실행으로 바꾸면 3사이클이 절약됩니다.

코드의 조건부 버전은 r0이 r1과 같은 경우 동일한 주기 수로 실행됩니다. 다른 모든 경우 코드의 조건부 버전은 더 적은 주기 수로 실행됩니다.

표 2-3 주기 수 (조건부 분기에만 해당)

r0: a	r1: b	명령어	주기 (ARM7)
1	2	`CMP r0, r1`	1
1	2	`BEQ end`	1 (실행되지 않음)
1	2	`BLT less`	3
1	2	`SUB r1, r1, r0`	1
1	2	`B gcd`	3
1	1	`CMP r0, r1`	1
1	1	`BEQ end`	3
			총 13개

표 2-4 주기 수 (모든 조건부 명령어에 해당)

r0: a	r1: b	명령어	주기 (ARM7)
1	2	`CMP r0, r1`	1
1	2	`SUBGT r0,r0,r1`	1 (실행되지 않음)
1	1	`SUBLT r1,r1,r0`	1
1	1	`BNE gcd`	3
1	1	`CMP r0,r1`	1
1	1	`SUBGT r0,r0,r1`	1 (실행되지 않음)
1	1	`SUBLT r1,r1,r0`	1 (실행되지 않음)
1	1	`BNE gcd`	1 (실행되지 않음)
			총 10개

gcd의 Thumb-2 이전 Thumb 버전

`B`는 조건부로 실행될 수 있는 유일한 Thumb-2 이전 Thumb 명령어이므로 gcd 알고리즘은 조건부 분기로 작성되어야 합니다.

ARM 조건부 분기 구현과 같이 Thumb-2 이전 Thumb 코드에는 일곱 개의 명령어가 필요합니다. 이 경우 전체 코드 크기는 16바이트의 ARM 구현에 비해 작은 14바이트입니다.

또한 16비트 메모리를 사용하는 시스템에서 각 16비트 Thumb 명령어에는 하나의 메모리 액세스만 필요하지만 각 ARM 32비트 명령어에는 두 개의 페치가 필요하므로 Thumb 버전이 두 번째 ARM 구현보다 *빠르게* 실행됩니다.

gcd의 Thumb-2 버전

다음과 같이 `IT` 명령어를 사용하여 `SUB` 명령어를 조건부 명령어로 만들어 이 코드의 ARM 버전을 Thumb-2 코드로 변환할 수 있습니다.

```
gcd
        CMP     r0, r1
        ITE     GT
        SUBGT   r0, r0, r1
        SUBLE   r1, r1, r0
        BNE     gcd
```

이 명령어는 ARM 또는 Thumb-2 코드로 똑같이 잘 어셈블됩니다. 어셈블러에서는 IT 명령어를 확인하지만 ARM 코드로 어셈블할 때는 IT 명령어를 생략합니다. IT 명령어를 생략할 수 있습니다. 어셈블러에서 Thumb-2 코드로 어셈블할 때 자동으로 IT 명령어를 삽입합니다.

명령어가 하나 더 필요한 코드는 ARM 코드가 아니라 Thumb-2 코드이지만 전체 코드 크기가 Thumb-2 코드는 10바이트지만 ARM 코드는 16바이트입니다.

실행 속도

실행 속도를 최적화하려면 명령어 타이밍, 분기 예상 논리 및 타겟 시스템의 캐시 동작을 자세히 알고 있어야 합니다. 개별 프로세서에 대한 자세한 내용은 *ARM 아키텍처 참조 문서* 및 기술 참조 설명서를 참조하십시오.

2.4.5 Q 플래그

ARMv5TE 및 ARMv6 이상에는 포화 산술 명령어에서 포화가 발생할 경우 (4-93 페이지의 *QADD, QSUB, QDADD 및 QDSUB* 참조) 또는 특정 곱하기 명령어에서 오버플로가 발생할 경우 (4-76페이지의 *SMULxy 및 SMLAxy*, 4-78페이지의 *SMULWy 및 SMLAWy* 참조) 기록할 Q 플래그가 있습니다.

Q 플래그는 *스티키* 플래그입니다. 이러한 명령어는 플래그를 설정할 수만 있고 지울 수는 없습니다. 각 명령어 다음에 나오는 플래그를 확인하지 않고도 이러한 일련의 명령어를 실행한 다음 플래그를 테스트하여 특정 지점에서 포화나 오버플로가 발생했는지 여부를 확인할 수 있습니다.

Q 플래그를 지우려면 `MSR` 명령어 (4-136페이지의 *MSR* 참조) 를 사용하십시오.

Q 플래그 상태는 조건 코드에서 직접 테스트할 수 없습니다. Q 플래그의 상태를 보려면 `MRS` 명령어 (4-134페이지의 *MRS* 참조) 를 사용하십시오.

2.5 레지스터에 상수 로드

메모리에서 데이터를 로드하지 않으면 단일 ARM 명령어로 임의의 32비트 즉치 상수를 레지스터에 로드할 수 없습니다. 이것은 ARM 및 Thumb-2 명령어의 길이가 32비트이기 때문입니다.

데이터 로드를 통해 32비트 값을 레지스터로 로드할 수 있지만 보다 직접적이고 효율적인 방법으로 공통적으로 사용되는 많은 상수를 로드할 수 있습니다.

별도의 로드 연산 없이 공통적으로 사용되는 많은 상수를 데이터 처리 명령어 내에 피연산자로 직접 포함할 수 있습니다. 16비트 Thumb 명령어에 피연산자로 포함할 수 있는 상수의 범위는 훨씬 더 좁습니다.

ARMv6T2 이상에서는 두 개의 명령어 `MOV`와 `MOVT`를 차례로 사용하여 32비트 값을 레지스터에 로드할 수 있습니다. `MOV32` 의사 명령어를 사용하여 명령어 시퀀스를 생성할 수 있습니다.

다음 단원에서는 다음 내용에 대해 설명합니다.

- `MOV` 및 `MVN` 명령어를 사용하여 즉치값 범위를 로드하는 방법

  자세한 내용은 2-29페이지의 *MOV 및 MVN을 통한 직접 로드*를 참조하십시오.

- `MOV32` 의사 명령어를 사용하여 32비트 상수를 로드하는 방법

  자세한 내용은 2-33페이지의 *MOV32를 사용하여 로드*를 참조하십시오.

- `LDR` 의사 명령어를 사용하여 32비트 상수를 로드하는 방법

  자세한 내용은 2-33페이지의 *LDR Rd, =const를 통한 직접 로드*를 참조하십시오.

- 부동 소수점 상수를 로드하는 방법

  2-35페이지의 *부동 소수점 상수 로드*를 참조하십시오.

2.5.1 MOV 및 MVN을 통한 직접 로드

ARM 및 Thumb-2에서는 32비트 MOV 및 MVN 명령어를 사용하여 광범위한 상수 값을 레지스터에 직접 로드할 수 있습니다.

16비트 Thumb MOV 명령어는 0 ~ 255 범위의 상수를 로드할 수 있습니다. 16비트 MVN 명령어를 사용하여 상수를 로드할 수는 없습니다.

*ARM 상태 즉치 상수*에서는 단일 ARM 명령어로 로드할 수 있는 값 범위를 보여 주고 2-31페이지의 *Thumb-2 즉치 상수*에서는 단일 Thumb-2 명령어로 로드할 수 있는 값 범위를 보여 줍니다.

MOV 또는 MVN 중 어느 것을 사용할지를 지정하지 않아도 어셈블러에서 자동으로 적절한 명령어를 사용합니다. 이 기능은 값이 어셈블리 타임 변수인 경우에 유용합니다.

사용할 수 없는 상수가 포함된 명령어를 작성하면 어셈블러에서 오류를 보고합니다. 이 작업을 수행하는 즉시 n은 범위를 벗어나게 됩니다.

ARM 상태 즉치 상수

ARM 상태의 경우

- MOV로 0x0-0xFF (0 ~ 255) 범위의 8비트 상수 값을 로드할 수 있습니다.

  또한 이러한 값을 짝수로 회전할 수 있습니다.

  또한 이러한 값은 별도의 명령어로 로드하지 않고 대부분의 데이터 처리 연산에서 즉치 피연산자로 사용할 수 있습니다.

- MVN으로 이러한 값의 비트 단위 보수를 로드할 수 있습니다. 숫자 값은 -(*n*+1) 입니다. 여기서 *n*은 MOV에서 사용할 수 있는 값입니다.

- ARMv6T2 이상에서는 MOV로 0x0-0xFFFF (0 ~ 65535) 범위의 16비트 숫자를 로드할 수 있습니다.

2-30페이지의 표 2-5에서는 이 명령어가 제공하는 8비트 값 범위를 보여 줍니다 (데이터 처리 연산의 경우).

2-30페이지의 표 2-6에서는 이 명령어가 제공하는 16비트 값 범위를 보여 줍니다 (MOV 명령어의 경우에만).

표 2-5 ARM 상태 즉치 상수 (8비트)

바이너리	10진수	단계	16진수	MVN 값[a]	메모
00000000000000000000000abcdefgh	0-255	1	0-0xFF	–1 ~ +256	-
00000000000000000000abcdefgh00	0-1020	4	0-0x3FC	–4 ~ +1024	-
00000000000000000000abcdefgh0000	0-4080	16	0-0xFF0	–16 ~ +4096	-
00000000000000000000abcdefgh000000	0-16320	64	0-0x3FC0	–64 ~ +16384	-
	...	...	...	...	-
abcdefgh000000000000000000000000	0-255 x 2^{24}	2^{24}	0-0xFF000000	1-256 x -2^{24}	-
cdefgh000000000000000000000000ab	(비트 패턴)	-	-	(비트 패턴)	*메모*의 b 참조
efgh000000000000000000000000abcd	(비트 패턴)	-	-	(비트 패턴)	*메모*의 b 참조
gh000000000000000000000000abcdef	(비트 패턴)	-	-	(비트 패턴)	*메모*의 b 참조

표 2-6 MOV 명령어의 ARM 상태 즉치 상수

바이너리	10진수	단계	16진수	MVN 값	메모
0000000000000000abcdefghijklmnop	0-65535	1	0-0xFFFF	-	*메모*의 c 참조

메모

이러한 참고는 표 2-5 및 표 2-6에 대한 추가 정보를 제공합니다.

a MVN 값은 MVN 명령어에서만 피연산자로 직접 사용할 수 있습니다.

b 이러한 값은 ARM 상태에서만 사용할 수 있습니다. 이 표에 나와 있는 다른 모든 값은 Thumb-2에서도 사용할 수 있습니다.

c 이러한 값은 ARMv6T2 이상에서만 사용할 수 있고 다른 명령어에서 피연산자로 직접 사용할 수 없습니다.

Thumb-2 즉치 상수

Thumb 상태에 있는 ARMv6T2 이상의 경우

- 32비트 MOV 명령어가 다음을 로드할 수 있습니다.
  - 0x0 ~ 0xFF (0 ~ 255) 범위의 8비트 상수 값
  - 임의의 숫자를 기준으로 왼쪽으로 시프트된 8비트 상수 값
  - 레지스터의 4바이트 모두에 복제된 8비트 패턴
  - 바이트 1과 바이트 3이 0으로 설정된 상태에서 바이트 0과 바이트 2에 복제된 8비트 패턴
  - 바이트 0과 바이트 2가 0으로 설정된 상태에서 바이트 1과 바이트 3에 복제된 8비트 패턴

  또한 이러한 값은 별도의 명령어로 로드하지 않고 대부분의 데이터 처리 연산에서 즉치 피연산자로 사용할 수 있습니다.
- 32비트 MVN 명령어는 이러한 값의 비트 단위 보수를 로드할 수 있습니다. 숫자 값은 - (*n*+1) 입니다. 여기서 *n*은 MOV에서 사용할 수 있는 값입니다.
- 32비트 MOV 명령어는 0x0 ~ 0xFFFF (0 ~ 65535) 범위의 16비트 숫자를 로드할 수 있습니다. 이러한 값은 데이터 처리 연산에서 즉치 피연산자로 사용할 수 없습니다.

표 2-7에서는 이 명령어가 제공하는 값 범위를 보여 줍니다 (데이터 처리 연산의 경우).

2-32페이지의 표 2-8에서는 이 명령어가 제공하는 16비트 값 범위를 보여 줍니다 (MOV 명령어의 경우에만).

표 2-7 Thumb-2 즉치 상수

바이너리	10진수	단계	16진수	MVN 값[a]	메모
000000000000000000000000abcdefgh	0-255	1	0-0xFF	–1 ~ +256	-
000000000000000000000000abcdefgh0	0-510	2	0-0x1FE	–2 ~ +512	-
000000000000000000000000abcdefgh00	0-1020	4	0-0x3FC	–4 ~ +1024	-
	...	...	...	...	-
0abcdefgh000000000000000000000000	0-255 x 2^{23}	2^{23}	0-0x7F800000	1-256 x -2^{23}	-

표 2-7 Thumb-2 즉치 상수 (계속)

바이너리	10진수	단계	16진수	MVN 값[a]	메모
abcdefgh00000000000000000000000	0-255 x 2^{24}	2^{24}	0-0xFF000000	1-256 x -2^{24}	-
abcdefghabcdefghabcdefghabcdefgh	(비트 패턴)	-	0xXYXYXYXY	0xXYXYXYXY	-
00000000abcdefgh00000000abcdefgh	(비트 패턴)	-	0x00XY00XY	0xFFXYFFXY	-
abcdefgh00000000abcdefgh00000000	(비트 패턴)	-	0xXY00XY00	0xXYFFXYFF	-
0000000000000000000abcdefghijkl	0-4095	1	0-0xFFF	-	*메모*의 b 참조

표 2-8 MOV 명령어의 Thumb-2 즉치 상수

바이너리	10진수	단계	16진수	MVN 값	메모
0000000000000000abcdefghijklmnop	0-65535	1	0-0xFFFF	-	*메모*의 c 참조

메모

이러한 참고는 2-31페이지의 표 2-7 및 표 2-8에 대한 추가 정보를 제공합니다.

a MVN 값은 MVN 명령어에서만 피연산자로 직접 사용할 수 있습니다.

b 이러한 값은 ADD, SUB 및 MOV 명령어에서 피연산자로 직접 사용할 수 있지만 MVN 또는 다른 데이터 처리 명령어에서는 사용할 수 없습니다.

c 이러한 값은 MOV 명령어에서만 사용할 수 있습니다.

2.5.2 MOV32를 사용하여 로드

ARMv6T2 이상에서 ARM 및 Thumb-2 명령어 세트에는 다음이 포함됩니다.

- 0x00000000 ~ 0x0000FFFF 범위의 값을 레지스터로 로드할 수 있는 MOV 명령어
- 최하위 반의 내용을 변경하지 않고 0x0000 ~ 0xFFFF 범위의 값을 레지스터의 최상위 반으로 로드할 수 있는 MOVT 명령어

이러한 두 개 명령어를 사용하여 레지스터에서 32비트 상수를 생성할 수 있습니다. 이외에도 MOV32 의사 명령어를 사용할 수 있습니다. 어셈블러는 MOV, MOVT 명령어 쌍을 생성합니다. MOV32 의사 명령어의 구문에 대한 자세한 내용은 4-157페이지의 *MOV32 의사 명령어*를 참조하십시오.

2.5.3 LDR Rd, =const를 통한 직접 로드

의사 LDR Rd,=*const* 명령어는 단일 명령어에서 32비트 숫자 상수를 생성할 수 있습니다. 이 의사 명령어를 사용하여 MOV 및 MVN 명령어 범위를 벗어난 상수를 생성할 수 있습니다.

LDR 의사 명령어는 특정 상수에 대한 가장 효율적인 단일 명령어를 생성합니다.

- 단일 MOV 또는 MVN 명령어를 사용하여 상수를 생성할 수 있으면 어셈블러가 해당 명령어를 생성합니다.
- 단일 MOV 또는 MVN 명령어를 사용하여 상수를 생성할 수 없으면 어셈블러가 다음을 수행합니다.
  - *리터럴 풀*(상수 값을 포함하기 위한 코드에 임베드된 메모리 일부) 에 값 배치
  - 리터럴 풀에서 값을 읽는 프로그램 상대 주소가 포함된 LDR 명령어 생성

  예를 들면 다음과 같습니다.

```
LDR      rn, [pc, #offset to literal pool]
                    ; load register n with one word
                    ; from the address [pc + offset]
```

  어셈블러에서 생성된 LDR 명령어의 범위 내에 리터럴 풀이 있는지 확인해야 합니다. 자세한 내용은 2-34페이지의 *리터럴 풀 배치*를 참조하십시오.

LDR 의사 명령어의 구문에 대한 자세한 내용은 4-159페이지의 *LDR 의사 명령어*를 참조하십시오.

리터럴 풀 배치

어셈블러는 각 섹션의 끝에 리터럴 풀을 배치합니다. 이러한 리터럴 풀은 다음 섹션 시작에 있는 AREA 지시어 또는 어셈블리 끝에 있는 END 지시어를 통해 정의됩니다. 포함 파일 끝에 있는 END 지시어는 섹션의 끝을 나타내지 않습니다.

넓은 섹션에서는 기본 리터럴 풀이 하나 이상의 LDR 명령어 범위를 벗어날 수 있습니다. PC에서 상수까지의 오프셋은 다음과 같아야 합니다.

- ARM 또는 Thumb-2 코드에서는 4KB 미만이지만 두 방향 중 한 방향이 될 수 있음
- Thumb-2 이전 Thumb 코드에서 또는 Thumb-2 코드에서 16비트 명령어를 사용할 경우 1KB 미만 및 정방향

LDR Rd,=const 의사 명령어가 상수를 리터럴 풀에 배치해야 할 경우 어셈블러에서는 다음을 수행합니다.

- 이전 리터럴 풀에서 상수를 사용할 수 있고 주소 지정이 가능한지 확인합니다. 이 경우 기존 상수에 주소를 지정합니다.
- 아직 사용하도록 설정되지 않은 경우 다음 리터럴 풀에 상수를 배치하려고 합니다.

다음 리터럴 풀이 범위를 벗어날 경우 어셈블러는 오류 메시지를 생성합니다. 이 경우 LTORG 지시어를 사용하여 코드에 추가 리터럴 풀을 배치해야 합니다. 실패한 LDR 의사 명령어 뒤에 ±4KB (ARM, 32비트 Thumb-2) 내 또는 0 ~ +1KB 범위 (Thumb-2 이전 Thumb, 16비트 Thumb-2) 에 LTORG 지시어를 배치합니다. 자세한 내용은 7-18페이지의 *LTORG*를 참조하십시오.

프로세서가 지시어를 명령어로 실행하려고 하지 않으면 리터럴 풀을 배치해야 합니다. 조건부가 아닌 분기 명령어 또는 서브루틴 끝의 반환 명령어 다음에 지시어를 배치합니다.

2-35페이지의 예제 2-4에서는 이 작업을 수행하는 방법을 보여 줍니다. 이 예제는 주 예제 디렉토리인 *install_directory*\RVDS\Examples에 loadcon.s로 제공되어 있습니다. 이 예제를 어셈블, 링크 및 실행하는 방법에 대한 자세한 내용은 2-2페이지의 *코드 예제*를 참조하십시오.

주석으로 표시된 명령어는 어셈블러에서 생성된 ARM 명령어입니다.

예제 2-4

```
        AREA    Loadcon, CODE, READONLY
        ENTRY                           ; Mark first instruction to execute
start
        BL      func1                   ; Branch to first subroutine
        BL      func2                   ; Branch to second subroutine
stop
        MOV     r0, #0x18               ; angel_SWIreason_ReportException
        LDR     r1, =0x20026            ; ADP_Stopped_ApplicationExit
        SVC     #0x123456               ; ARM semihosting (formerly SWI)
func1
        LDR     r0, =42                 ; => MOV R0, #42
        LDR     r1, =0x55555555         ; => LDR R1, [PC, #offset to
                                        ; Literal Pool 1]
        LDR     r2, =0xFFFFFFFF         ; => MVN R2, #0
        BX      lr
        LTORG                           ; Literal Pool 1 contains
                                        ; literal Ox55555555
func2
        LDR     r3, =0x55555555         ; => LDR R3, [PC, #offset to
                                        ; Literal Pool 1]
        ; LDR r4, =0x66666666           ; If this is uncommented it
                                        ; fails, because Literal Pool 2
                                        ; is out of reach
        BX      lr
LargeTable
        SPACE   4200                    ; Starting at the current location,
                                        ; clears a 4200 byte area of memory
                                        ; to zero
        END                             ; Literal Pool 2 is empty
```

2.5.4 부동 소수점 상수 로드

NEON 및 VFPv3 명령어 세트에는 제한된 범위의 부동 소수점 상수를 즉치 상수로 로드하는 명령어가 있습니다. 다음을 참조하십시오.

- NEON 명령어에 대한 자세한 내용은 5-46페이지의 *VMOV, VMVN (즉치값)*을 참조하십시오.
- VFPv3 명령어에 대한 자세한 내용은 5-109페이지의 *VMOV*를 참조하십시오.

VLDR 의사 명령어를 사용하여 단일 명령어에서 단정밀도 또는 배정밀도 부동 소수점 값을 로드할 수 있습니다.

자세한 내용은 5-87페이지의 *VLDR 의사 명령어*를 참조하십시오.

2.6 레지스터에 주소 로드

주소를 레지스터로 로드해야 할 수 있습니다. 변수 주소, 문자열 상수, 이동 테이블의 시작 위치를 로드해야 할 수 있습니다.

일반적으로 주소는 현재 PC 또는 기타 레지스터의 오프셋으로 표시됩니다.

이 단원에서는 주소를 레지스터로 로드하는 다음과 같은 방법에 대해 설명합니다.

- 레지스터 직접 로드 (*ADR 및 ADRL을 통한 직접 로드* 참조)
- 리터럴 풀에서 주소 로드 (2-39페이지의 *LDR Rd, =label을 통한 주소 로드* 참조)

2.6.1 ADR 및 ADRL을 통한 직접 로드

ADR 명령어 및 ADRL 의사 명령어를 사용하여 데이터 로드를 수행하지 않고 특정 범위 내에 주소를 생성할 수 있습니다. ADR 및 ADRL은 프로그램 상대 식, 즉 레이블 주소가 현재 PC에 상대적인 선택적 오프셋이 있는 레이블을 허용합니다.

참고

ADR 또는 ADRL에 사용된 레이블은 동일한 코드 섹션 내에 있어야 합니다. 어셈블러에서는 동일한 섹션의 범위를 벗어난 레이블에 대한 참조에 대해 오류를 발생시킵니다.

Thumb 상태에서 16비트 ADR 명령어는 워드로 정렬된 주소만 생성할 수 있습니다.

ADRL은 Thumb-2 이전 프로세서의 Thumb 상태에서 사용할 수 없습니다.

ADR

사용 가능한 범위는 다음과 같은 명령어 세트에 따라 달라집니다.

ARM 바이트 또는 하프워드로 정렬된 주소의 경우, 255바이트
워드로 정렬된 주소의 경우, 1020바이트

32비트 Thumb-2 바이트, 하프워드 또는 워드로 정렬된 주소의 경우, 4095바이트

16비트 Thumb 0 ~ 1020바이트. *label*는 워드로 정렬되어야 합니다. 이렇게 하려면 ALIGN 지시어를 사용하면 됩니다.

자세한 내용은 4-23페이지의 *ADR*을 참조하십시오.

ADRL

어셈블러는 다음을 생성하여 ADRL *rn*,*label* 의사 명령어를 변환합니다.

- 범위 내에 있는 경우 주소를 로드하는 두 개의 데이터 처리 명령어
- 두 개의 명령어로 주소를 생성할 수 없는 경우 오류 메시지

사용 가능한 범위는 다음과 같이 사용 중인 명령어 세트에 따라 달라집니다.

ARM 바이트 또는 하프워드로 정렬된 주소의 경우, 64KB
워드로 정렬된 주소의 경우 256KB

32비트 Thumb-2 바이트, 하프워드 또는 워드로 정렬된 주소의 경우 1MB

16비트 Thumb ADRL을 사용할 수 없음

ADRL 의사 명령어 범위를 벗어난 주소를 로드하는 방법에 대한 자세한 내용은 2-39페이지의 *LDR Rd, =label을 통한 주소 로드*를 참조하십시오.

ADR을 통한 점프 테이블 구현

예제 2-5에서는 점프 테이블을 구현하는 ARM 코드를 보여 줍니다. 이 예제에서 ADR 의사 명령어는 점프 테이블의 주소를 로드합니다. 이 예제는 주 예제 디렉토리인 *install_directory*\RVDS\Examples에 jump.s로 제공되어 있습니다. 이 예제를 어셈블, 링크 및 실행하는 방법에 대한 자세한 내용은 2-2페이지의 *코드 예제*를 참조하십시오.

예제 2-5 이동 테이블 구현 (ARM)

```
        AREA    Jump, CODE, READONLY     ; Name this block of code
        ARM                              ; Following code is ARM code
num     EQU     2                        ; Number of entries in jump table
        ENTRY                            ; Mark first instruction to execute
start                                    ; First instruction to call
        MOV     r0, #0                   ; Set up the three parameters
        MOV     r1, #3
        MOV     r2, #2
        BL      arithfunc                ; Call the function
stop
        MOV     r0, #0x18                ; angel_SWIreason_ReportException
        LDR     r1, =0x20026             ; ADP_Stopped_ApplicationExit
        SVC     #0x123456                ; ARM semihosting (formerly SWI)
arithfunc                                ; Label the function
        CMP     r0, #num                 ; Treat function code as unsigned
integer
```

```
        BXHS    lr                      ; If code is >= num then simply return
        ADR     r3, JumpTable           ; Load address of jump table
        LDR     pc, [r3,r0,LSL#2]       ; Jump to the appropriate routine
JumpTable
        DCD     DoAdd
        DCD     DoSub
DoAdd
        ADD     r0, r1, r2              ; Operation 0
        BX      lr                      ; Return
DoSub
        SUB     r0, r1, r2              ; Operation 1
        BX      lr                      ; Return
        END                             ; Mark the end of this file
```

2-37페이지의 예제 2-5에서 함수 arithfunc는 세 개의 인수를 사용하고 결과를 r0에 반환합니다. 첫 번째 인수는 두 번째 및 세 번째 인수에서 수행되는 연산을 결정합니다.

argument1=0 Result = argument2 + argument3.

argument1=1 Result = argument2 – argument3.

점프 테이블은 다음 명령어와 어셈블러 지시어를 사용하여 구현합니다.

EQU 어셈블러 지시어입니다. 심볼에 값을 제공하는 데 사용됩니다. 2-37페이지의 예제 2-5에서 이 지시어는 값 2를 *num*에 할당합니다. 표준의 요구 사항에 따라 *num*이 코드에 사용되면 값 2를 대체합니다. 이 방법으로 EQU를 사용하는 것은 #define을 사용하여 C에서 상수를 정의하는 것과 비슷합니다.

DCD 저장소에 하나 이상의 워드를 선언합니다. 2-37페이지의 예제 2-5에서 각 DCD는 이동 테이블의 특정 절을 처리하는 루틴의 주소를 저장합니다.

LDR LDR pc,[r3,r0,LSL#2] 명령어는 점프 테이블의 필수 절 주소를 PC로 로드합니다. 이 명령어는 다음을 수행합니다.

- r0의 절 번호에 4를 곱해 워드 오프셋을 제공합니다.
- 결과에 점프 테이블 주소를 더합니다.
- 결합된 주소 내용을 PC로 로드합니다.

2.6.2 LDR Rd, =label을 통한 주소 로드

LDR Rd,= 의사 명령어는 32비트 숫자 상수를 레지스터에 로드할 수 있습니다 (2-33 페이지의 *LDR Rd, =const를 통한 직접 로드* 참조). 또한 레이블 및 오프셋 포함 레이블과 같은 프로그램 상대 식을 허용합니다. 구문 설명을 보려면 4-159페이지의 *LDR 의사 명령어*를 참조하십시오.

어셈블러에서는 다음을 수행하여 LDR r0, =*label* 의사 명령어를 변환합니다.

- 리터럴 풀 (상수 값을 포함하기 위한 코드에 임베드된 메모리 일부) 에 *label* 주소를 배치합니다.
- 리터럴 풀에서 주소를 읽는 프로그램 기준 주소가 포함된 LDR 명령어를 생성합니다. 예를 들면 다음과 같습니다.

```
LDR      rn [pc, #offset to literal pool]
                  ; load register n with one word
                  ; from the address [pc + offset]
```

  범위 내에 리터럴 풀이 있는지 확인해야 합니다. 자세한 내용은 2-34페이지의 *리터럴 풀 배치*를 참조하십시오.

ADR 및 ADRL 의사 명령어와 달리 LDR은 현재 섹션 외부의 레이블에 사용할 수 있습니다. 레이블이 현재 섹션의 외부에 있으면 어셈블러는 소스 파일을 어셈블할 때 재배치 지시어를 개체 코드에 배치합니다. 재배치 지시어는 링크 타임에 링커가 주소를 확인하도록 지정합니다. 링커가 LDR과 리터럴 풀이 포함된 섹션을 배치하는 위치에 관계없이 주소는 유효한 상태로 유지됩니다.

예제 2-6에서는 이 작업을 수행하는 방법을 보여 줍니다. 이 예제는 주 예제 디렉토리인 *install_directory*\RVDS\Examples에 ldrlabel.s로 제공되어 있습니다. 이 예제를 어셈블, 링크 및 실행하는 방법에 대한 자세한 내용은 2-2페이지의 *코드 예제*를 참조하십시오.

주석에 나열된 명령어는 어셈블러에서 생성된 ARM 명령어입니다.

예제 2-6

```
        AREA    LDRlabel, CODE,READONLY
        ENTRY                           ; Mark first instruction to execute
start
        BL      func1                   ; Branch to first subroutine
        BL      func2                   ; Branch to second subroutine
stop
        MOV     r0, #0x18               ; angel_SWIreason_ReportException
        LDR     r1, =0x20026            ; ADP_Stopped_ApplicationExit
```

```
        SVC     #0x123456               ; ARM semihosting (formerly SWI)
func1
        LDR     r0, =start              ; => LDR R0,[PC, #offset into
                                        ; Literal Pool 1]
        LDR     r1, =Darea + 12         ; => LDR R1,[PC, #offset into
                                        ; Literal Pool 1]
        LDR     r2, =Darea + 6000       ; => LDR R2, [PC, #offset into
                                        ; Literal Pool 1]
        BX      lr                      ; Return
        LTORG                           ; Literal Pool 1
func2
        LDR     r3, =Darea + 6000       ; => LDR r3, [PC, #offset into
                                        ; Literal Pool 1]
                                        ; (sharing with previous literal)
        ; LDR   r4, =Darea + 6004       ; If uncommented produces an error
                                        ; as Literal Pool 2 is out of range
        BX      lr                      ; Return
Darea   SPACE   8000                    ; Starting at the current location,
                                        ; clears a 8000 byte area of memory
                                        ; to zero
        END                             ; Literal Pool 2 is out of range of
                                        ; the LDR instructions above
```

LDR Rd, =label 예제: 문자열 복사

예제 2-7에서는 한 문자열에 다른 문자열을 덮어쓰는 ARM 코드 루틴을 보여 줍니다. 이 예제에서는 LDR 의사 명령어를 사용하여 두 문자열의 주소를 데이터 섹션에서 로드합니다. 다음은 특히 중요한 예제입니다.

DCB DCB 지시어는 저장소에 하나 이상의 바이트를 정의합니다. 정수 값과 함께 DCB 는 따옴표로 묶인 문자열을 허용합니다. 문자열의 각 문자는 연속된 바이트로 배치됩니다. 자세한 내용은 7-22페이지의 *DCB* 를 참조하십시오.

LDR, STR LDR 및 STR 명령어는 Post 인덱싱된 주소 지정을 사용하여 주소 레지스터를 업데이트합니다. 다음 명령어를 예로 들 수 있습니다.

```
LDRB    r2,[r1],#1
```

r1에 의해 지정된 주소 내용을 사용하여 r2를 로드한 다음 r1을 1씩 증가시킵니다.

예제 2-7 문자열 복사

```
        AREA    StrCopy, CODE, READONLY
        ENTRY                           ; Mark first instruction to execute
start
        LDR     r1, =srcstr             ; Pointer to first string
        LDR     r0, =dststr             ; Pointer to second string
        BL      strcopy                 ; Call subroutine to do copy
stop
        MOV     r0, #0x18               ; angel_SWIreason_ReportException
        LDR     r1, =0x20026            ; ADP_Stopped_ApplicationExit
        SVC     #0x123456               ; ARM semihosting (formerly SWI)
strcopy
        LDRB    r2, [r1],#1             ; Load byte and update address
        STRB    r2, [r0],#1             ; Store byte and update address
        CMP     r2, #0                  ; Check for zero terminator
        BNE     strcopy                 ; Keep going if not
        MOV     pc,lr                   ; Return
        AREA    Strings, DATA, READWRITE
srcstr  DCB     "First string - source",0
dststr  DCB     "Second string - destination",0
        END
```

2.7 다중 레지스터 로드 및 저장 명령어

ARM, Thumb-2 및 Thumb-2 이전 Thumb 명령어 세트는 메모리에서 여러 레지스터를 로드하고 메모리에 저장하는 명령어를 포함합니다.

다중 레지스터 전송 명령어는 여러 레지스터의 내용을 메모리와 사이에서 이동하는 효율적인 방법을 제공합니다. 이 명령어는 블록 복사 및 서브루틴 진입 및 종료의 스택 연산에 가장 많이 사용됩니다. 일련의 단일 데이터 전송 명령어 대신 다중 레지스터 전송 명령어를 사용하면 다음과 같은 장점이 있습니다.

- 더 작은 코드 크기
- 여러 명령어 페치가 아니라 단일 명령어 페치 오버헤드
- 변경되지 않은 ARM 프로세서에서 다중 로드 또는 다중 저장에 의한 데이터의 첫 워드는 항상 비순차 메모리 주기이지만 전송되는 모든 후속 워드는 순차 메모리 주기가 될 수 있습니다. 대부분의 시스템에서는 순차 메모리 주기가 더 빠릅니다.

참고

번호가 가장 낮은 레지스터는 액세스되는 가장 낮은 메모리 주소에서 또는 이 주소로 전송되고 번호가 가장 높은 레지스터는 액세스되는 가장 높은 메모리 주소에서 또는 이 주소로 전송됩니다. 명령어의 레지스터 목록에 나열된 레지스터 순서에 따른 차이는 없습니다.

`--diag_warning 1206` 어셈블러 명령 행 옵션을 사용하여 레지스터 목록의 레지스터가 오름차순으로 지정되어 있는지 확인할 수 있습니다.

이 단원에서는 다음 내용을 설명합니다.

- 2-43페이지의 *ARM 및 Thumb-2 에서 사용할 수 있는 다중 로드 및 저장 명령어*
- 2-44페이지의 *LDM 및 STM 을 통한 스택 구현*
- 2-47페이지의 *LDM 및 STM 을 통한 블록 복사*

2.7.1 ARM 및 Thumb-2에서 사용할 수 있는 다중 로드 및 저장 명령어

다음 명령어는 ARM 명령어 세트와 Thumb-2 명령어 세트에서 사용할 수 있습니다.

LDM 다중 레지스터를 로드합니다.

STM 다중 레지스터를 저장합니다.

PUSH 다중 레지스터를 스택에 저장하고 스택 포인터를 업데이트합니다.

POP 다중 레지스터를 스택에서 로드하고 스택 포인터를 업데이트합니다.

LDM 및 STM 명령어의 경우

- 로드 또는 저장된 레지스터 목록에는 다음이 포함될 수 있습니다.
  - ARM 명령어의 경우, r0 ~ r15 모두
  - 32비트 Thumb-2 명령어의 경우, r0 ~ r12 모두 및 선택적으로 일부 제한이 있는 r14 또는 r15
  - 16비트 Thumb 및 Thumb-2 명령어의 경우 r0 ~ r7 모두
- 주소는 다음 중 하나일 수 있습니다.
  - 각 전송 후에 증가하는 주소
  - 각 전송 전에 증가하는 주소 (ARM 명령어에만 해당)
  - 각 전송 후에 감소하는 주소 (ARM 명령어에만 해당)
  - 각 전송 전에 감소하는 주소 (16비트 Thumb 제외)
- 기준 레지스터는 다음 중 하나일 수 있습니다.
  - 메모리의 다음 데이터 블록을 가리키도록 업데이트되는 레지스터
  - 명령어 이전 상태로 남아 있는 레지스터

기준 레지스터가 메모리의 다음 블록을 가리키도록 업데이트될 경우 이 작업을 *쓰기 되돌림*이라고 합니다. 즉, 인접 주소가 기준 레지스터에 다시 기록됩니다.

PUSH 및 POP 명령어의 경우

- 스택 포인터 (sp) 가 기준 레지스터이고 항상 업데이트됩니다.
- 주소는 POP 명령어의 각 전송 후에 증가되고 PUSH 명령어의 각 전송 전에 감소됩니다.
- 로드 또는 저장된 레지스터 목록에는 다음이 포함될 수 있습니다.
  - ARM 명령어의 경우, r0 ~ r15 모두

— 32비트 Thumb-2 명령어의 경우, r0 ~ r12 모두 및 선택적으로 일부 제한이 있는 r14 또는 r15

— 16비트 Thumb-2 및 Thumb 명령어의 경우, r0 ~ r7 모두 및 선택적으로 r14 (PUSH에만 해당) 또는 r15 (POP에만 해당)

2.7.2 LDM 및 STM을 통한 스택 구현

다중 로드 및 저장 명령어는 기준 레지스터를 업데이트할 수 있습니다. 스택 연산의 경우 기준 레지스터는 일반적으로 스택 포인트 sp입니다. 즉, 이러한 명령어를 사용하여 단일 명령어에서 임의의 레지스터 수에 대해 푸시 및 팝 연산을 구현할 수 있습니다.

다중 로드 및 저장 명령어는 여러 유형의 스택에 사용할 수 있습니다.

내림차순 또는 오름차순

스택은 상위 주소에서 시작하여 하위 주소로 진행하여 (*내림차순* 스택) 아래쪽으로 증가하거나 하위 주소에서 시작하여 상위 주소로 진행하여 (*오름차순* 스택) 위쪽으로 증가합니다.

전체 또는 비어 있음

스택 포인터는 스택의 마지막 항목 (*전체* 스택) 또는 스택의 다음 빈 공간 (*빈* 스택) 을 가리킬 수 있습니다.

프로그래머가 보다 쉽게 작업할 수 있도록 스택 지향 접미사를 증가 또는 감소 대신 접미사 앞이나 뒤에 사용할 수 있습니다. 표 2-9에서는 스택 지향 접미사 및 로드 및 저장 명령어의 해당 주소 지정 모드 접미사를 보여 줍니다.

표 2-9 스택 지향 접미사 및 해당 주소 지정 모드 접미사

스택 지향 접미사	저장 또는 푸시 명령어의 경우	로드 또는 팝 명령어의 경우
FD (전체 내림차순 스택)	DB (이전 감소)	IA (이후 증가)
FA (전체 오름차순 스택)	ID (이전 증가)	DA (이후 감소)
ED (빈 내림차순 스택)	DA (이후 감소)	ID (이전 증가)
EA (빈 오름차순 스택)	IA (이후 증가)	DB (이전 감소)

표 2-10에서는 여러 유형의 스택에 사용할 수 있는 스택 지향 접미사가 포함된 여러 로드 및 저장 명령어를 보여 줍니다.

표 2-10 다중 로드 및 저장 명령어의 접미사

스택 유형	저장	로드
전체 내림차순	STMFD (STMDB, 이전 감소)	LDMFD (LDM, 이후 증가)
전체 오름차순	STMFA (STMIB, 이전 증가)	LDMFA (LDMDA, 이후 감소)
빈 내림차순	STMED (STMDA, 이후 감소)	LDMED (LDMIB, 이전 증가)
빈 오름차순	STMEA (STM, 이후 증가)	LDMEA (LDMDB, 이전 감소)

예를 들면 다음과 같습니다.

```
STMFD    sp!, {r0-r5}  ; Push onto a Full Descending Stack
LDMFD    sp!, {r0-r5}  ; Pop from a Full Descending Stack
```

참고

Procedure Call Standard for the ARM Architecture (AAPCS) 와 ARM , Thumb C 및 C++ 컴파일러에서는 항상 전체 내림차순 스택을 사용합니다.

PUSH 및 POP 명령어는 전체 내림차순 스택을 가정합니다. 이러한 명령어 쓰기 되돌림이 포함된 STMDB 및 LDM의 동의어입니다.

중첩 서브루틴에 대해 레지스터 스택

스택 연산은 서브루틴 진입과 종료에 유용합니다. 서브루틴 진입 시에 필요한 작업 레지스터는 스택에 저장되고 종료 시에 다시 팝될 수 있습니다.

또한 진입 시 링크 레지스터를 스택으로 푸시하면 복귀 주소가 손실되지 않고 추가 서브루틴 호출을 안전하게 수행할 수 있습니다. 이렇게 하면 종료 시 스택에서 pc를 팝하고 해당 값을 pc로 이동하여 서브루틴에서 복귀할 수 있습니다. 예를 들어 다음과 같습니다.

```
subroutine  PUSH    {r5-r7,lr} ; Push work registers and lr
            ; code
            BL      somewhere_else
            ; code
            POP     {r5-r7,pc} ; Pop work registers and pc
```

참고

혼합 ARM 및 Thumb 시스템에서는 이 명령어를 주의하여 사용해야 합니다. ARMv4T 시스템에서 pc로 직접 팝하여 상태를 변경할 수 없습니다. 이 경우 주소를 임시 레지스터로 팝하고 BX 명령어를 사용해야 합니다.

ARMv5T 이상에서는 이 방법으로 상태를 변경할 수 있습니다.

ARM과 Thumb을 함께 사용하는 데 대한 자세한 내용은 개발자 설명서의 5장 *ARM과 Thumb의 인터워킹*을 참조하십시오.

2.7.3 LDM 및 STM을 통한 블록 복사

예제 2-8은 한 번에 하나의 워드를 복사하여 소스 위치에서 대상으로 워드 세트를 복사하는 ARM 코드 루틴입니다. 이 예제는 주 예제 디렉토리인 *install_directory*\RVDS\Examples에 word.s로 제공되어 있습니다. 이 예제를 어셈블, 링크 및 실행하는 방법에 대한 자세한 내용은 2-2페이지의 *코드 예제*를 참조하십시오.

예제 2-8 LDM 및 STM을 사용하지 않는 블록 복사

```
            AREA    Word, CODE, READONLY     ; name this block of code
num         EQU     20                       ; set number of words to be copied
            ENTRY                            ; mark the first instruction called
start
            LDR     r0, =src                 ; r0 = pointer to source block
            LDR     r1, =dst                 ; r1 = pointer to destination block
            MOV     r2, #num                 ; r2 = number of words to copy
wordcopy
            LDR     r3, [r0], #4             ; load a word from the source and
            STR     r3, [r1], #4             ; store it to the destination
            SUBS    r2, r2, #1               ; decrement the counter
            BNE     wordcopy                 ; ... copy more
stop
            MOV     r0, #0x18                ; angel_SWIreason_ReportException
            LDR     r1, =0x20026             ; ADP_Stopped_ApplicationExit
            SVC     #0x123456                ; ARM semihosting (formerly SWI)
            AREA    BlockData, DATA, READWRITE
src         DCD     1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4
dst         DCD     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
            END
```

이 모듈은 가능한 한 많은 복사본을 만들기 위해 LDM 및 STM을 사용하여 보다 효율적으로 만들 수 있습니다. ARM에 있는 레지스터의 수를 고려하면 한 번에 여덟 개의 워드를 전송하는 것이 좋습니다. 복사할 블록에서 8워드 배수는 다음을 사용하여 찾을 수 있습니다 (r2 = 복사할 워드 수).

```
MOVS   r3, r2, LSR #3    ; number of eight word multiples
```

이 값은 이터레이션당 여덟 개의 워드를 복사하는 루프를 통해 이터레이션 수를 제어하는 데 사용할 수 있습니다. 여덟 개 미만의 워드가 남아 있으면 r2가 손상되지 않았다는 가정 하에 다음을 사용하여 남은 워드 수를 찾을 수 있습니다.

```
ANDS   r2, r2, #7
```

예제 2-9에서는 복사에 LDM 및 STM을 사용하도록 재작성된 블록 복사 모듈을 나열합니다.

예제 2-9 LDM 및 STM을 사용한 블록 복사

```
            AREA    Block, CODE, READONLY    ; name this block of code
num         EQU     20                       ; set number of words to be copied
            ENTRY                            ; mark the first instruction called
start
            LDR     r0, =src                 ; r0 = pointer to source block
            LDR     r1, =dst                 ; r1 = pointer to destination block
            MOV     r2, #num                 ; r2 = number of words to copy
            MOV     sp, #0x400               ; Set up stack pointer (sp)
blockcopy
            MOVS    r3,r2, LSR #3            ; Number of eight word multiples
            BEQ     copywords                ; Less than eight words to move?
            PUSH    {r4-r11}                 ; Save some working registers
octcopy
            LDM     r0!, {r4-r11}            ; Load 8 words from the source
            STM     r1!, {r4-r11}            ; and put them at the destination
            SUBS    r3, r3, #1               ; Decrement the counter
            BNE     octcopy                  ; ... copy more
            POP     {r4-r11}                 ; Don't need these now - restore
                                             ; originals
copywords
            ANDS    r2, r2, #7               ; Number of odd words to copy
            BEQ     stop                     ; No words left to copy?
wordcopy
            LDR     r3, [r0], #4             ; Load a word from the source and
            STR     r3, [r1], #4             ; store it to the destination
            SUBS    r2, r2, #1               ; Decrement the counter
            BNE     wordcopy                 ; ... copy more
stop
            MOV     r0, #0x18                ; angel_SWIreason_ReportException
            LDR     r1, =0x20026             ; ADP_Stopped_ApplicationExit
            SVC     #0x123456                ; ARM semihosting (formerly SWI)
            AREA    BlockData, DATA, READWRITE
src         DCD     1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4
dst         DCD     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
            END
```

2.8 매크로 사용

매크로 정의는 MACRO 및 MEND 지시어로 묶인 코드 블록입니다. 매크로 정의는 전체 코드 블록을 반복하는 대신 사용할 수 있는 이름을 정의합니다. 매크로는 주로 다음과 같은 목적으로 사용됩니다.

- 코드 블록을 하나의 의미 있는 이름으로 바꿔서 더 간편하게 소스 코드의 논리를 따를 수 있도록 합니다.
- 코드 블록을 여러 번 반복하지 않도록 합니다.

자세한 내용은 7-33페이지의 *MACRO 및 MEND*를 참조하십시오.

이 단원에서는 다음 내용을 설명합니다.

- 2-50페이지의 *테스트 및 분기 매크로 예제*
- 2-50페이지의 *부호 없는 정수 나누기 매크로 예제*

2.8.1 테스트 및 분기 매크로 예제

ARM 코드 및 Thumb-2 이전 프로세서의 Thumb에서 테스트 및 분기 연산에는 두 개의 ARM 명령어를 구현해야 합니다.

다음과 같이 매크로 정의를 정의할 수 있습니다.

```
        MACRO
$label  TestAndBranch  $dest, $reg, $cc
$label  CMP     $reg, #0
        B$cc    $dest
        MEND
```

MACRO 지시어 다음의 행은 *매크로 프로토타입 문*입니다. 이 문은 매크로 호출에 사용할 이름 (TestAndBranch) 을 정의합니다. 또한 *매개변수* ($label, $dest, $reg 및 $cc) 를 정의합니다. 지정되지 않은 매개변수는 빈 문자열로 대체됩니다. 이 매크로의 경우 구문 오류를 방지하려면 $dest, $reg 및 $cc에 값을 제공해야 합니다. 어셈블러에서는 제공된 값을 코드로 대체합니다.

이 매크로는 다음과 같이 호출할 수 있습니다.

```
test    TestAndBranch    NonZero, r0, NE
         ...
         ...
NonZero
```

대체 후에는 다음과 같이 됩니다.

```
test    CMP     r0, #0
        BNE     NonZero
         ...
         ...
NonZero
```

2.8.2 부호 없는 정수 나누기 매크로 예제

2-51페이지의 예제 2-10에서는 부호 없는 정수 나누기를 수행하는 매크로를 보여줍니다. 네 개의 매개변수를 사용합니다.

$Bot — 제수가 들어 있는 레지스터

$Top — 명령어가 실행되기 전 피제수가 들어 있는 레지스터로, 명령어가 실행된 후에는 나머지를 포함합니다.

$Div — 나누기의 몫이 배치되는 레지스터로, 나머지만 필요한 경우에는 NULL ("") 일 수 있습니다.

`$Temp` 계산하는 동안 사용되는 임시 레지스터

예제 2-10

```
        MACRO
$Lab    DivMod  $Div,$Top,$Bot,$Temp
        ASSERT  $Top <> $Bot            ; Produce an error message if the
        ASSERT  $Top <> $Temp           ; registers supplied are
        ASSERT  $Bot <> $Temp           ; not all different
        IF      "$Div" <> ""
            ASSERT  $Div <> $Top        ; These three only matter if $Div
            ASSERT  $Div <> $Bot        ; is not null ("")
            ASSERT  $Div <> $Temp       ;
        ENDIF
$Lab
        MOV     $Temp, $Bot             ; Put divisor in $Temp
        CMP     $Temp, $Top, LSR #1     ; double it until
90      MOVLS   $Temp, $Temp, LSL #1    ; 2 * $Temp > $Top
        CMP     $Temp, $Top, LSR #1
        BLS     %b90                    ; The b means search backwards
        IF      "$Div" <> ""            ; Omit next instruction if $Div is null
            MOV     $Div, #0            ; Initialize quotient
        ENDIF
91      CMP     $Top, $Temp             ; Can we subtract $Temp?
        SUBCS   $Top, $Top,$Temp        ; If we can, do so
        IF      "$Div" <> ""            ; Omit next instruction if $Div is null
            ADC     $Div, $Div, $Div    ; Double $Div
        ENDIF
        MOV     $Temp, $Temp, LSR #1    ; Halve $Temp,
        CMP     $Temp, $Bot             ; and loop until
        BHS     %b91                    ; less than divisor
        MEND
```

매크로에서는 두 개의 매개변수가 동일한 레지스터를 사용하는지 확인합니다. 또한 나머지만 필요한 경우 생성된 코드를 최적화합니다.

어셈블러 소스에서 둘 이상의 `DivMod`가 사용되는 경우 여러 개의 레이블 정의를 방지하기 위해 매크로에서는 지역 레이블 (90, 91) 을 사용합니다. 자세한 내용은 2-14페이지의 *지역 레이블*을 참조하십시오.

2-52페이지의 예제 2-11에서는 이 매크로를 다음과 같이 호출할 경우 생성되는 코드를 보여 줍니다.

```
ratio  DivMod  r0,r5,r4,r2
```

예제 2-11

```
        ASSERT  r5 <> r4                ; Produce an error if the
        ASSERT  r5 <> r2                ; registers supplied are
        ASSERT  r4 <> r2                ; not all different
        ASSERT  r0 <> r5                ; These three only matter if $Div
        ASSERT  r0 <> r4                ; is not null ("")
        ASSERT  r0 <> r2                ;
ratio
        MOV     r2, r4                  ; Put divisor in $Temp
        CMP     r2, r5, LSR #1          ; double it until
90      MOVLS   r2, r2, LSL #1          ; 2 * r2 > r5
        CMP     r2, r5, LSR #1
        BLS     %b90                    ; The b means search backwards
        MOV     r0, #0                  ; Initialize quotient
91      CMP     r5, r2                  ; Can we subtract r2?
        SUBCS   r5, r5, r2              ; If we can, do so
        ADC     r0, r0, r0              ; Double r0
        MOV     r2, r2, LSR #1          ; Halve r2,
        CMP     r2, r4                  ; and loop until
        BHS     %b91                    ; less than divisor
```

2.9 기호 버전 추가

ARM 링커는 BPABI (*ARM 아키텍처용 기본 플랫폼 ABI*) 를 준수하고 GNU 확장 기호 버전 관리 모델을 지원합니다.

기존 기호에 기호 버전을 추가하려면 동일한 주소에서 버전 기호를 정의해야 합니다. 버전 심볼의 형식은 다음과 같습니다.

- *name*의 기본 버전이 아닌 *ver*의 경우, *name@ver*
- *name*의 기본 *ver*의 경우, *name@@ver*

버전 심볼은 세로 막대로 묶어야 합니다.

예를 들어 기본 버전을 정의하려면 다음 명령어를 실행합니다.

```
|my_versioned_symbol@@ver2|   ; Default version
my_asm_function PROC
                ...
                BX lr
                ENDP
```

기본 버전 이외의 버전을 정의하려면 다음 명령어를 실행합니다.

```
|my_versioned_symbol@ver1|    ; Non default version
my_old_asm_function     PROC
                        ...
                        BX lr
                        ENDP
```

RVCT의 기호 관리에 대한 자세한 내용은 *링커 사용 설명서*에서 4장 *이미지 기호 액세스*를 참조하십시오.

2.10 프레임 지시어 사용

다음 중 하나를 수행하려는 경우 코드에서 스택을 사용하는 방법을 정의하려면 프레임 지시어를 사용해야 합니다.

- 스택 해제를 사용하여 응용 프로그램 디버그
- 플랫 또는 콜 그래프 프로파일링 사용

이러한 지시어에 대한 자세한 내용은 7-41페이지의 *프레임 지시어*를 참조하십시오.

어셈블러에서는 프레임 지시어를 사용하여 DWARF 디버그 프레임 정보를 개체 파일에 생성되는 ELF 형식으로 삽입합니다. 이 정보는 디버거에서 스택 해제 및 프로파일링을 위해 필요합니다. 스택 검사 한정자에 대한 자세한 내용은 *install_directory*\Documentation\Specifications\...에 있는 *Procedure Call Standard for the ARM Architecture* 사양 (aapcs.pdf) 을 참조하십시오.

다음 사항에 유의하십시오.

- 프레임 지시어는 어셈블러에서 생성한 코드에 영향을 주지 않습니다.
- 어셈블러에서는 프레임 지시어의 정보를 내보낸 명령어에 대해 확인하지 않습니다.

2.11 어셈블리 언어 변경 사항

표 2-11에서는 UAL과 이전의 개별 ARM 및 Thumb 어셈블러 언어 간의 기본 차이점을 보여 줍니다. 어셈블러에서는 UAL 이전 ARM 구문이 허용됩니다.

표 2-11 이전 ARM 어셈블리 언어의 변경 사항

변경된 내용	UAL 이전 ARM 구문	기본 구문
LDM 및 STM의 기본 주소 지정 모드는 IA입니다.	`LDMIA, STMIA`	`LDM, STM`
ARM 및 Thumb에서 전체 내림차순 스택 연산에 대해 PUSH 및 POP 니모닉을 사용할 수 있습니다.	`STMFD sp!, {reglist}` `LDMFD sp!, {reglist}`	`PUSH {reglist}` `POP {reglist}`

ARM 및 Thumb에서는 다른 연산을 제외하고 회전만 포함된 명령어에 대해 LSL, LSR, ASR, ROR 및 RRX 명령어 니모닉을 사용할 수 있습니다.	`MOV Rd, Rn, LSL shift`
	`MOV Rd, Rn, LSR shift`
	`MOV Rd, Rn, ASR shift`
	`MOV Rd, Rn, ROR shift`
	`MOV Rd, Rn, RRX`
		`LSL Rd, Rn, shift`
		`LSR Rd, Rn, shift`
		`ASR Rd, Rn, shift`
		`ROR Rd, Rn, shift`
		`RRX Rd, Rn`

PC 상대 주소 지정에는 *label* 형식을 사용합니다. 새 코드에서 *offset* 형식을 사용하면 안 됩니다.	`LDR Rd, [pc, #offset]`	`LDR Rd, label`
더블워드 메모리 액세스에 대해 두 레지스터를 모두 지정합니다. 사용할 수 있는 레지스터 조합에 대한 규칙을 따라야 합니다.	`LDRD Rd, addr_mode`	`LDRD Rd, Rd2, addr_mode`
{*cond*}, 사용될 경우 항상 모든 명령어의 마지막 요소입니다 .	`ADD{cond}S` `LDR{cond}SB`	`ADDS{cond}` `LDRSB{cond}`

ARM 및 Thumb-2 코드 모두에서 ARM {*cond*} 조건부 형식 및 Thumb-2 IT 명령어를 모두 사용할 수 있습니다. 어셈블러에서는 두 코드의 일관성을 확인하고 해당 코드를 현재 명령어 세트에 따라 어셈블합니다. IT 명령어를 생략하면 어셈블러에서 코드를 Thumb-2 코드에 자동으로 삽입합니다.	`ADDEQ r1, r2, r3`
	`LDRNE r1, [r2, r3]`
		`ITEQ E ; optional`
		`ADDEQ r1, r2, r3`
		`LDRNE r1, [r2, r3]`

또한 이전 어셈블러에서 허용되지 않은 일부 융통성이 허용됩니다 (표 2-12 참조).

표 2-12 필요 조건 완화

완화	기본 구문	허용 구문
대상 레지스터가 첫 번째 연산과 동일할 경우 명령어의 2레지스터 형식을 사용할 수 있습니다.	`ADD r1, r1, r3`	`ADD r1, r3`

UAL을 사용하여 Thumb-2 이전 Thumb 프로세서에 대한 소스 코드를 작성할 수 있습니다.

Thumb-2 이전 프로세서에 대한 Thumb 코드를 작성할 경우 프로세서에서 사용할 수 있는 명령어로 제한해야 합니다. 사용할 수 없는 명령어를 사용하려고 하면 어셈블러에서는 오류 메시지를 생성합니다.

Thumb-2 프로세서에 대한 Thumb 코드를 작성할 경우 가능한 여러 위치에 16비트 명령어를 사용하여 코드 크기를 최소화할 수 있습니다.

표 2-13에서는 UAL 이전 Thumb 어셈블리 언어와 UAL 간의 기본 차이점을 보여줍니다. 어셈블러에서는 CODE16 지시어보다 먼저 사용되거나 소스 파일이 --16 명령 행 옵션으로 어셈블된 경우에만 UAL 이전 Thumb 구문을 허용합니다.

표 2-13 UAL 이전 Thumb 구문과 UAL 구문 간의 차이점

변경된 내용	UAL 이전 Thumb 구문	UAL 구문
LDM 및 STM의 기본 주소 지정 모드는 IA입니다.	`LDMIA, STMIA`	`LDM, STM`

플래그를 업데이트하는 명령어에는 S 접미사를 사용해야 합니다. 이 변경 사항은 32비트 Thumb-2 명령어와의 충돌을 방지하는 데 필수적입니다.	`ADD r1, r2, r3`
	`SUB r4, r5, #6`
	`MOV r0, #1`
	`LSR r1, r2, #1`
		`ADDS r1, r2, r3`
		`SUBS r4, r5, #6`
		`MOVS r0, #1`
		`LSRS r1, r2, #1`

ALU 명령어의 기본 형식은 대상 레지스터가 첫 번째 피연산자와 동일한 경우에도 세 개의 레지스터를 지정합니다.	`ADD r7, r8` `SUB r1, #80`	`ADD r7, r7, r8` `SUBS r1, r1, #80`

Rd 및 *Rn*이 모두 Lo 레지스터이면 MOV *Rd*, *Rn*은 ADDS *Rd*, *Rn*, #0으로 디스어셈블됩니다.	`MOV r2, r3`
	`MOV r8, r9`
	`CPY r0, r1`
	`LSL r2, r3, #0`
		`ADDS r2, r3, #0`
		`MOV r8, r9`
		`MOV r0, r1`
		`MOVS r2, r3`

NEG *Rd*, *Rm*은 RSBS *Rd*, *Rm*, #0으로 디스어셈블됩니다.	`NEG Rd, Rm`	`RSBS Rd, Rm, #0`
NOP 명령어는 가능한 경우 MOV r8, r8을 바꿉니다.	`- NOP`	`NOP MOV r8, r8`

3장
어셈블러 참조

이 장에서는 ARM® 어셈블러에 대한 일반적인 참조 자료를 제공합니다. 여기에는 다음 단원이 포함되어 있습니다.

- 3-2페이지의 *명령 구문*
- 3-21페이지의 *소스 행 형식*
- 3-22페이지의 *미리 정의된 레지스터 및 보조 프로세서 이름*
- 3-24페이지의 *기본 제공 변수 및 상수*
- 3-27페이지의 *기호*
- 3-33페이지의 *식, 리터럴 및 연산자*
- 3-46페이지의 *진단 메시지*
- 3-48페이지의 *C 사전 처리기 사용*

이 장에서는 ARM 어셈블리 언어를 작성하는 방법에 대해 설명하지 않습니다. 자습 정보를 보려면 2장 *ARM 어셈블리 언어 작성*을 참조하십시오.

이 장에서는 명령어, 지시어 또는 의사 명령어에 대해서도 설명하지 않습니다. 이러한 항목에 대한 참조 정보는 개별 장을 참조하십시오.

3.1 명령 구문

이 단원에서는 armasm과 관련된 정보만 다룹니다. 인라인 및 임베디드 어셈블러는 C 및 C++ 컴파일러의 일부로, 고유한 명령 구문을 갖고 있지 않습니다.

armasm 명령 행은 파일 이름에서만 대소문자를 구분하고 달리 지정하지 않는 한 대소문자를 구분하지 않습니다. ARM 어셈블러는 *컴파일러 사용 설명서*의 2-10 페이지의 *명령 행 옵션 순서 지정*에 설명되어 있는 일반 명령 행 정렬 규칙을 사용합니다. 따라서 명령 행에 서로 충돌하는 옵션이 있으면 마지막 옵션이 항상 우선적으로 적용됩니다.

다음 명령을 사용하여 ARM 어셈블러를 호출합니다.

```
armasm {options} {inputfile}
```

여기서 *options* 자리에는 다음과 같은 옵션을 공백으로 구분하여 조합할 수 있습니다.

`--16` UAL 이전 Thumb 구문을 사용하여 명령어를 Thumb® 명령어로 해석하도록 어셈블러에 지시합니다. 이 옵션은 소스 파일 헤드에 있는 CODE16 지시어와 같습니다. `--thumb` 옵션을 사용하면 UAL 구문을 통해 Thumb 또는 Thumb-2 명령어를 지정할 수 있습니다.

`--32` `--arm`의 동의어입니다.

`--apcs` *qualifier*

AAPCS (*ARM 아키텍처용 프로시저 호출 표준*) 를 사용하는지 여부를 지정합니다. 또한 코드 섹션의 일부 속성을 지정할 수 있습니다. 자세한 내용은 3-10페이지의 *AAPCS*를 참조하십시오.

`--arm` 명령어를 ARM 명령어로 해석하도록 어셈블러에 지시합니다. 그러나 객체 파일에 ARM 전용 코드만 사용되는 것은 아닙니다. 기본값입니다.

`--arm_only` ARM 코드만 생성하도록 어셈블러에 지시합니다. 그 결과로 생성되는 객체 파일은 Thumb 코드를 포함하거나 허용하지 않습니다.

`--bi` `--bigend`의 동의어입니다.

`--bigend` 빅엔디안 ARM에 적합한 코드를 어셈블하도록 어셈블러에 지시합니다. 기본값은 `--littleend`입니다.

--brief_diagnostics

자세한 내용은 3-18페이지의 *진단 메시지의 출력 제어*를 참조하십시오.

--checkreglist

RLIST, LDM 및 STM 레지스터 목록을 확인하여 모든 레지스터 번호가 오름차순으로 나열되도록 어셈블러에 지시합니다. 레지스터가 순서대로 나열되지 않으면 경고가 표시됩니다.

이 옵션은 제공되지 않습니다. 대신 --diag_warning 1206을 사용하십시오 (3-18페이지의 *진단 메시지의 출력 제어* 참조).

--cpreproc armcc를 호출하여 입력 파일을 어셈블하기 전에 사전 처리하도록 어셈블러에 지시합니다. 자세한 내용은 3-48페이지의 *C 사전 처리기 사용*을 참조하십시오.

--cpreproc_opts=*options*

C 사전 처리기를 사용할 때 어셈블러가 armcc로 컴파일러 옵션을 전달할 수 있도록 합니다. 자세한 내용은 3-48페이지의 *C 사전 처리기 사용*을 참조하십시오.

*options*는 옵션 및 해당 값이 콤마로 구분되어 있는 목록입니다.

--cpu *name* 타겟 CPU를 설정합니다. 자세한 내용은 3-12페이지의 *CPU 이름*을 참조하십시오.

--debug DWARF 디버그 테이블을 생성하도록 어셈블러에 지시합니다. --debug는 -g의 동의어입니다.

기본값은 DWARF 3입니다.

참고

지역 기호는 --debug를 통해 유지되지 않습니다. 디버깅에 유용하도록 지역 기호를 유지하려면 --keep을 지정해야 합니다.

--depend *dependfile*

소스 파일 종속 목록을 *dependfile*에 저장하도록 어셈블러에 지시합니다. 이러한 목록은 make 유틸리티에 사용하기에 적합합니다.

--depend_format=*string*

출력 종속 파일의 형식을 일부 UNIX make 프로그램과 호환될 수 있도록 UNIX 스타일 형식으로 변경합니다.

*string*의 값은 다음 중 하나일 수 있습니다.

unix UNIX 스타일 경로 구분 기호를 사용하여 종속 파일을 생성합니다.

unix_escaped
unix와 같지만 백슬래시로 공백을 이스케이프합니다.

unix_quoted
unix와 같지만 경로 이름을 큰따옴표로 묶습니다.

--device=*list*

--device=*name* 옵션에 사용할 수 있는 지원되는 장치 이름을 나열합니다.

--device=*name*

특정 장치를 선택하고 관련 프로세서 설정을 지정합니다. 컴파일러 참조 설명서의 2-42페이지의 *--device=name*을 참조하십시오.

--diag_[error | remark | warning | suppress | style]

자세한 내용은 3-18페이지의 *진단 메시지의 출력 제어*를 참조하십시오.

--dllexport_all

소스 지시어에서 재정의하는 경우가 아니면 ELF에서 내보낸 모든 전역 기호에 대해 STV_HIDDEN이 아닌 STV_PROTECTED 표시를 적용합니다 (7-78페이지의 *EXPORT 또는 GLOBAL* 참조).

--dwarf2 --debug와 함께 사용하여 DWARF 2 디버그 테이블을 생성하도록 어셈블러에 지시합니다.

--dwarf3 --debug와 함께 사용하여 DWARF 3 디버그 테이블을 생성하도록 어셈블러에 지시합니다. --debug를 지정한 경우 이 옵션이 기본 옵션입니다.

--errors *errorfile*

오류 메시지를 *errorfile*에 출력하도록 어셈블러에 지시합니다.

--exceptions 자세한 내용은 3-20페이지의 *예외 테이블 생성 제어*를 참조하십시오.

`--exceptions_unwind`

자세한 내용은 3-20페이지의 *예외 테이블 생성 제어*를 참조하십시오.

`--fpmode` *model*

부동 소수점 규칙을 지정하고 라이브러리 속성 및 부동 소수점 최적화를 설정합니다. 자세한 내용은 3-11페이지의 *부동 소수점 모델*을 참조하십시오.

`--fpu` *name* 타겟 FPU (*부동 소수점 단위*) 아키텍처를 선택합니다. 자세한 내용은 3-13페이지의 *FPU 이름*을 참조하십시오.

`-g` `--debug`의 동의어입니다.

`-i`*dir*`{,`*dir*`,…}`

정규화할 소스 파일에 디렉토리를 추가합니다 (7-81페이지의 *GET 또는 INCLUDE* 참조).

`--keep` 디버거에서 사용할 수 있도록 객체 파일의 기호 테이블에 지역 레이블을 유지하도록 어셈블러에 지시합니다 (7-85페이지의 *KEEP* 참조).

`--length` 자세한 내용은 3-16페이지의 *파일에 출력 나열*을 참조하십시오.

`--li` `--littleend`의 동의어입니다.

`--library_type=`*lib*

선택한 관련 라이브러리를 링크 타임에 사용할 수 있도록 합니다.

여기서 *lib*는 다음 중 하나일 수 있습니다.

`standardlib` 링크 타임에 전체 ARM 런타임 라이브러리가 선택되도록 지정합니다. 기본값입니다.

`microlib` 링크 타임에 C 마이크로 라이브러리 (microlib)가 선택되도록 지정합니다.

참고

라이브러리를 사용하는 데 더욱 특수화된 최적화가 필요한 경우 컴파일러, 어셈블러 또는 링커에서 이 옵션을 사용할 수 있습니다.

다른 `--library_type` 옵션을 모두 무시하려면 링커에 이 옵션을 사용합니다.

자세한 내용은 다음 항목을 참조하십시오.

- *라이브러리 설명서*의 3-4페이지의 *microlib로 응용 프로그램 빌드*
- *컴파일러 참조 설명서*의 2-77페이지의 `--library_type=lib`

`--licretry` 어셈블러에 특정 FLEXnet 오류 코드가 발생할 경우 라이센스 체크아웃을 약 10초 간격으로 최대 10번까지 다시 시도하도록 지시합니다.

`--list` *file* 어셈블러에 의해 생성된 자세한 어셈블리 언어 목록을 *file*에 출력하도록 어셈블러에 지시합니다. 자세한 내용은 3-16페이지의 *파일에 출력 나열*을 참조하십시오.

`--littleend` 리틀엔디안 ARM에 적합한 코드를 어셈블하도록 어셈블러에 지시합니다.

`-m` 소스 파일 종속 목록을 `stdout`에 작성하도록 어셈블러에 지시합니다.

`--maxcache` *n* 최대 소스 캐시 크기를 *n*바이트로 설정합니다. 기본값은 8MB입니다. 크기가 8MB보다 작으면 `armasm`에서 경고를 표시합니다.

`--md` 소스 파일 종속 목록을 `inputfile.d`에 작성하도록 어셈블러에 지시합니다.

`--memaccess` *attributes*

타겟 메모리 시스템의 메모리 액세스 속성을 지정합니다. 자세한 내용은 3-15페이지의 *메모리 액세스 특성*을 참조하십시오.

참고

`--memaccess` 옵션은 제공되지 않습니다.

`--no_code_gen`

패스 1 후 종료하도록 어셈블러에 지시합니다. 객체 파일이 생성되지 않습니다.

`--no_esc` `\n` 및 `\t`와 같은 C 스타일 이스케이프 특수 문자를 무시하도록 어셈블러에 지시합니다.

--no_exceptions

자세한 내용은 3-20페이지의 *예외 테이블 생성 제어*를 참조하십시오.

--no_exceptions_unwind

자세한 내용은 3-20페이지의 *예외 테이블 생성 제어*를 참조하십시오.

--no_hide_all

소스 지시어에서 재정의하는 경우가 아니면 ELF에서 내보내고 가져온 모든 전역 기호에 대해 STV_HIDDEN이 아닌 STV_DEFAULT 표시를 적용합니다 (7-78페이지의 *EXPORT 또는 GLOBAL* 및 7-82페이지의 *IMPORT 및 EXTERN* 참조).

--no_regs 레지스터 이름을 미리 정의하지 않도록 어셈블러에 지시합니다. 미리 정의된 레지스터 이름 목록을 보려면 3-22페이지의 *미리 정의된 레지스터 및 보조 프로세서 이름*을 참조하십시오.

이 옵션은 제공되지 않습니다. 대신 --regnames=none을 사용하십시오.

--no_terse 자세한 내용은 3-16페이지의 *파일에 출력 나열*을 참조하십시오.

--no_unaligned_access

정렬되지 않은 액세스가 사용되지 않음을 나타내는 객체 파일의 속성을 설정하도록 어셈블러에 지시합니다.

--no_warn 경고 메시지를 해제합니다.

-o *filename* 출력 객체 파일의 이름을 지정합니다. 이 옵션을 지정하지 않으면 어셈블러에서 *inputfilename*.o 형식의 객체 파일 이름을 만듭니다. 이 옵션은 대소문자를 구분합니다.

--pd --predefine의 동의어입니다.

--predefine "*directive*"

SET 지시어 중 하나를 미리 실행하도록 어셈블러에 지시합니다. 자세한 내용은 3-15페이지의 *SET 지시어 사전 실행*을 참조하십시오. 이 옵션은 조건부 어셈블리에 유용합니다 (7-39페이지의 *정의 중인 변수에 따른 조건부 어셈블리* 참조).

--[no_]project=*filename*

자세한 내용은 3-17페이지의 *프로젝트 템플릿 옵션*을 참조하십시오.

--[no_]reduce_paths

파일 경로에서 중복 경로 이름 정보를 제거하거나 제거할 수 없도록 합니다. 이 옵션은 Windows 시스템에만 유효합니다.

Windows 시스템에서는 파일 경로에 260자 제한이 있습니다. 절대 이름이 260자 넘게 확장되는 상대 경로 이름이 있는 경우 --reduce_paths 옵션을 사용하여 디렉터리를 해당 .. 인스턴스와 일치시키고 directory/.. 시퀀스를 쌍에서 제거하여 절대 경로 이름 길이를 줄일 수 있습니다.

참고

--reduce_paths 옵션을 사용하여 경로 길이를 최소화하는 것보다는 중첩 수준이 높고 긴 파일 경로를 사용하지 않는 것이 좋습니다.

자세한 내용은 *컴파일러 참조 설명서*의 2-103페이지의 *--[no_]reduce_paths*를 참조하십시오.

--regnames=none

레지스터 이름을 미리 정의하지 않도록 어셈블러에 지시합니다. 미리 정의된 레지스터 이름 목록을 보려면 3-22페이지의 *미리 정의된 레지스터 및 보조 프로세서 이름*을 참조하십시오.

--regnames=callstd

--apcs 옵션을 사용하여 지정하는 AAPCS 변형을 기반으로 추가 레지스터 이름을 정의합니다. 자세한 내용은 3-10페이지의 *AAPCS*를 참조하십시오.

--regnames=all

--apcs 값에 관계없이 모든 AAPCS 레지스터를 정의합니다. 자세한 내용은 3-10페이지의 *AAPCS*를 참조하십시오.

--reinitialize_workdir

자세한 내용은 3-17페이지의 *프로젝트 템플릿 옵션*을 참조하십시오.

`--show_cmdline`

어셈블러에서 명령 행이 처리된 방법을 보여 줍니다. 명령은 표준화된 상태로 표시되고 via 파일의 내용은 확장됩니다.

`--split_ldm` 긴 `LDM` 및 `STM` 명령어에 대해 오류를 생성하도록 어셈블러에 지시합니다. 자세한 내용은 3-15페이지의 *긴 LDM 및 STM 분할*을 참조하십시오. 이 옵션은 향후 사용할 수 없습니다.

`--thumb` UAL 구문을 사용하여 명령어를 Thumb 명령어로 해석하도록 어셈블러에 지시합니다. 이 옵션은 소스 파일 헤드에 있는 `THUMB` 지시어와 같습니다.

`--unaligned_access`

정렬되지 않은 액세스의 사용을 나타내는 객체 파일의 속성을 설정하도록 어셈블러에 지시합니다.

`--unsafe` 다양한 아키텍처의 명령어를 오류 없이 어셈블할 수 있습니다. 자세한 내용은 3-18페이지의 *진단 메시지의 출력 제어*를 참조하십시오.

`--untyped_local_labels`

Thumb 코드의 레이블을 참조할 경우 Thumb 비트를 설정하지 않도록 어셈블러에 지시합니다. 자세한 내용은 4-159페이지의 *LDR 의사 명령어*를 참조하십시오.

`--via` *`file`* *`file`*을 열고 어셈블러에 대한 명령 행 인수를 읽도록 어셈블러에 지시합니다. 자세한 내용은 *컴파일러 참조 설명서*에서 부록 A *via 파일 구문*을 참조하십시오.

`--width` 자세한 내용은 3-16페이지의 *파일에 출력 나열*을 참조하십시오.

`--workdir=`*`directory`*

자세한 내용은 3-17페이지의 *프로젝트 템플릿 옵션*을 참조하십시오.

`--xref` 자세한 내용은 3-16페이지의 *파일에 출력 나열*을 참조하십시오.

`inputfile` 어셈블러의 입력 파일을 지정합니다. 입력 파일은 UAL 또는 UAL 이전 Thumb 어셈블리 언어 소스 파일이어야 합니다.

3.1.1 사용 가능한 옵션 목록 보기

사용 가능한 어셈블러 명령 행 옵션에 대한 요약을 보려면 다음 명령을 입력합니다.

```
armasm --help
```

3.1.2 환경 변수를 통한 명령 행 옵션 지정

RVCT40_ASMOPT 환경 변수 값을 설정하여 명령 행 옵션을 지정할 수 있습니다. 이 구문은 명령 행 구문과 동일합니다. 어셈블러에서는 RVCT40_ASMOPT의 값을 읽고 명령 문자열의 맨 앞에 삽입합니다. 즉, RVCT40_ASMOPT에 지정된 옵션을 명령 행의 인수로 재정의할 수 있습니다.

3.1.3 AAPCS

AAPCS (*ARM 아키텍처용 프로시저 호출 표준*) 를 사용하고 있는지 여부를 지정하는 옵션이 있습니다.

```
--apcs qualifier
```

AAPCS는 BSABI (*ARM 아키텍처용 기본 표준 응용 프로그램 바이너리 인터페이스*) 사양 부분을 구성합니다. AAPCS를 준수하는 코드를 작성하여 별도로 컴파일되고 어셈블된 모듈을 함께 사용할 수 있습니다.

또한 --apcs 옵션도 코드 섹션의 일부 속성을 지정할 수 있습니다.

자세한 내용은 *install_directory*\Documentation\Specifications\... 에 있는 *Procedure Call Standard for the ARM Architecture* 사양 (aapcs.pdf) 을 참조하십시오.

참고

AAPCS 한정자는 어셈블러에서 생성한 코드에 영향을 주지 않습니다. 이 한정자는 *inputfile*의 코드가 AAPCS의 특정 변수로 컴파일되는 프로그래머의 어설션입니다. 이 한정자를 통해 어셈블러에서 생성한 객체 파일에서 특성이 설정됩니다. 링커에서는 이러한 특성을 사용하여 파일 확장성을 확인하고 적절한 라이브러리 변형을 선택합니다.

*qualifier*의 값은 다음과 같습니다.

none
: *inputfile*이 AAPCS를 사용하지 않음을 지정합니다. AAPCS 레지스터는 설정되지 않습니다. none을 사용하면 다른 한정자를 사용할 수 없습니다.

/interwork	*inputfile*의 코드가 ARM 및 Thumb 인터워킹에 적합함을 지정합니다. 자세한 내용은 개발자 설명서에서 5장 *ARM과 Thumb의 인터워킹*을 참조하십시오.
/nointerwork	*inputfile*의 코드가 ARM 및 Thumb 인터워킹에 적합하지 않음을 지정합니다. 기본값입니다.
/inter	/interwork의 동의어입니다.
/nointer	/nointerwork의 동의어입니다.
/ropi	*inputfile*의 내용이 읽기 전용 위치 독립적 코드임을 지정합니다.
/noropi	*inputfile*의 내용이 읽기 전용 위치 독립적이지 않은 코드임을 지정합니다. 이것이 기본값입니다.
/pic	/ropi의 동의어입니다.
/nopic	/noropi의 동의어입니다.
/rwpi	*inputfile*의 내용이 읽기/쓰기 위치 독립적 코드임을 지정합니다.
/norwpi	*inputfile*의 내용이 읽기/쓰기 위치 독립적이지 않은 코드임을 지정합니다. 이것이 기본값입니다.
/pid	/rwpi의 동의어입니다.
/nopid	/norwpi의 동의어입니다.
/fpic	*inputfile*의 내용이 FPIC 주소 지정을 필요로 하는 읽기 전용 위치 독립적 코드임을 지정합니다.

참고

*qualifier*를 여러 개 지정하는 경우에는 목록의 개별 한정자 사이에 공백이나 콤마가 없어야 합니다.

3.1.4 부동 소수점 모델

다음은 부동 소수점 모델을 지정하는 옵션입니다.

`--fpmode` *model*

타겟 부동 소수점 모델을 선택하고 링크할 때 가장 적합한 라이브러리를 선택하는 특성을 설정합니다.

참고

이 옵션은 작성한 코드를 변경하지 않습니다.

model 은 다음 중 하나일 수 있습니다.

`ieee_full` IEEE 표준이 보장하는 모든 기능, 작업 및 표현이 단정밀도와 배정밀도로 제공됩니다. 런타임에 작동 모드를 동적으로 선택할 수 있습니다.

`ieee_fixed` 가장 가까운 수로 반올림되며 부정확한 예외가 발생하지 않는 IEEE 표준입니다.

`ieee_no_fenv` 가장 가까운 수로 반올림되며 예외가 발생하지 않는 IEEE 표준입니다. 이 모드는 Java 부동 소수점 산술 모델과 호환됩니다.

`std` 0으로 플러시되는 IEEE 유한 값으로, 가장 가까운 수로 반올림되며 예외가 발생하지 않습니다. 이 값은 C 및 C++와 호환됩니다. 기본 옵션입니다.

유한 값은 IEEE 표준에서 예측한 값입니다. IEEE 모델에서 정의한 일부 환경에서는 NaN과 무한 값이 생성되지 않을 수 있으며, 생성된다고 해도 동일한 부호를 갖지 않을 수 있습니다. 또한 0의 부호가 IEEE 모델에서 예측하는 것과 다를 수 있습니다.

`fast` 속도가 향상되는 대신 정확도가 떨어지도록 최적화를 변경합니다. 이 옵션은 IEEE와 호환되지 않으며 표준 C가 아닙니다.

3.1.5 CPU 이름

다음은 CPU 이름을 지정하는 옵션입니다.

`--cpu` *name* 타겟 CPU를 설정합니다. 일부 명령어는 잘못된 타겟 CPU에 대해 어셈블될 경우 오류 또는 경고를 생성합니다 (3-18페이지의 *진단 메시지의 출력 제어* 참조).

*name*의 유효한 값은 4T, 5TE 또는 6T2와 같은 아키텍처 이름이거나 ARM7TDMI와 같은 부품 번호입니다. 아키텍처에 대한 자세한 내용은 *ARM 아키텍처 참조 문서*를 참조하십시오. 기본값은 ARM7TDMI입니다.

참고

--cpu=7을 사용하면 ARMv7-A, ARMv7-R 및 ARMv7-M 아키텍처에서 지원하는 코드가 생성됩니다. 즉, 어셈블러는 ARMv-7A, ARMv7-R 및 ARMv7-M 아키텍처에서 사용할 수 있는 Thumb 명령어만 생성하도록 제한됩니다.

링크 타임에 소프트웨어 라이브러리 선택이 주는 영향에 대한 자세한 내용은 *링커 사용 설명서*를 참조하십시오.

유효한 CPU 이름 목록 보기

다음 명령을 통해 어셈블러를 호출하면 유효한 CPU 및 아키텍처 이름 목록을 볼 수 있습니다.

```
armasm --cpu list
```

3.1.6 FPU 이름

다음은 FPU 이름을 지정하는 옵션입니다.

--fpu *name* 타겟 FPU (*부동 소수점 단위*) 아키텍처를 선택합니다. 이 옵션을 지정하면 --cpu 옵션으로 설정된 암시적 FPU가 재정의됩니다. 명시적으로 지정하는 FPU가 CPU와 호환되지 않으면 어셈블러에서 오류가 생성됩니다. 또한 부동 소수점 명령어는 잘못된 타겟 FPU에 대해 어셈블될 경우에도 오류 또는 경고를 생성합니다.

어셈블러에서는 객체 파일의 *name*에 해당하는 빌드 특성을 설정합니다. 그러면 링커에서 이러한 특성에 따라 객체 파일 간 호환성을 확인하고 라이브러리를 선택합니다.

*name*에 사용할 수 있는 값은 다음과 같습니다.

none 부동 소수점 아키텍처를 선택하지 않습니다. 그러면 어셈블된 객체 파일이 FPU를 사용하여 빌드한 객체 파일과 호환됩니다.

vfpv3	VFPv3 아키텍처를 준수하는 하드웨어 벡터 부동 소수점 단위를 선택합니다.
vfpv3_fp16	반정밀도 부동 소수점 확장을 포함하는 아키텍처 VFPv3을 준수하는 하드웨어 부동 소수점 단위를 선택합니다.
vfpv3_d16	VFPv3-D16 아키텍처를 준수하는 하드웨어 부동 소수점 단위를 선택합니다.
vfpv3_d16_fp16	반정밀도 부동 소수점 확장을 포함하는 아키텍처 VFPv3-D16을 준수하는 하드웨어 부동 소수점 단위를 선택합니다.
vfpv2	VFPv2 아키텍처를 준수하는 하드웨어 벡터 부동 소수점 단위를 선택합니다.
softvfp	소프트웨어 부동 소수점 연결을 선택합니다. --fpu 옵션을 지정하지 않고 선택된 --cpu 옵션에 특정 FPU가 포함되어 있지 않은 경우 이 옵션이 기본 옵션입니다.
softvfp+vfpv2	VFP 명령어를 사용하는 소프트웨어 부동 소수점 연결이 있는 부동 소수점 라이브러리를 선택합니다. 그렇지 않은 경우 이 옵션은 --fpu vfpv2를 사용하는 것과 같습니다.
softvfp+vfpv3	VFP 명령어를 사용하는 소프트웨어 부동 소수점 연결이 있는 부동 소수점 라이브러리를 선택합니다. 그렇지 않은 경우 이 옵션은 --fpu vfpv3을 사용하는 것과 같습니다.
softvfp+vfpv3_fp16	VFP 명령어를 사용하는 소프트웨어 부동 소수점 연결이 있는 부동 소수점 라이브러리를 선택합니다. 그렇지 않은 경우 이 옵션은 --fpu vfpv3_fp16을 사용하는 것과 같습니다.
softvfp+vfpv3_d16	VFP 명령어를 사용하는 소프트웨어 부동 소수점 연결이 있는 부동 소수점 라이브러리를 선택합니다. 그렇지 않은 경우 이 옵션은 --fpu vfpv3_d16을 사용하는 것과 같습니다.

softvfp+vfpv3_d16_fp16

VFP 명령어를 사용하는 소프트웨어 부동 소수점 연결이 있는 부동 소수점 라이브러리를 선택합니다.

그렇지 않은 경우 이 옵션은 --fpu vfpv3_d16_fp16을 사용하는 것과 같습니다.

링크 타임에 이러한 값이 소프트웨어 라이브러리 선택에 주는 영향에 대한 자세한 내용은 *링커 사용 설명서*를 참조하십시오.

유효한 FPU 이름 목록 보기

다음 명령을 통해 어셈블러를 호출하면 유효한 FPU 이름 목록을 볼 수 있습니다.

```
armasm --fpu list
```

3.1.7 메모리 액세스 특성

타겟 메모리 시스템의 메모리 액세스 특성을 지정하려면 다음을 사용하십시오.

--memaccess *attributes*

기본값은 바이트, 하프워드 및 워드의 정렬된 로드 및 저장을 활성화하는 것입니다. *attributes*는 기본값을 수정하며 다음 중 하나일 수 있습니다.

+L41 정렬되지 않은 LDR을 활성화합니다.

-L22 하프워드 로드를 활성화하지 않습니다.

-S22 하프워드 저장을 활성화하지 않습니다.

-L22-S22 하프워드 로드 및 저장을 활성화하지 않습니다.

참고

--memaccess 옵션은 제공되지 않습니다.

3.1.8 SET 지시어 사전 실행

다음 옵션을 사용하여 SET 지시어 중 하나를 미리 실행하도록 어셈블러에 지시할 수 있습니다.

--predefine "*directive*"

*directive*는 따옴표로 묶어야 합니다. 자세한 내용은 7-8페이지의 *SETA, SETL 및 SETS*를 참조하십시오. 또한 어셈블러에서는 변수 값을 설정하기 전에 해당 GBLL, GBLS 또는 GBLA 지시어를 실행하여 변수를 정의합니다.

변수 이름은 대소문자를 구분합니다.

참고

*directive*에 문자열을 포함하려면 시스템의 명령 행 인터페이스에 \" 같은 특수 문자 조합을 입력해야 합니다. 또는 --via *file*을 사용하여 --predefine 인수를 포함할 수 있습니다. 명령 행 인터페이스는 --via 파일의 인수를 변경하지 않습니다.

3.1.9 긴 LDM 및 STM 분할

다음 옵션을 사용하여 많은 수의 레지스터가 있는 LDM 및 STM 명령어에 대해 오류를 생성하도록 어셈블러에 지시합니다.

--split_ldm

이 옵션은 다음과 같은 전송된 레지스터의 최대 수가 초과된 경우 LDM 및 STM 명령어에 대해 오류를 생성합니다.

- 모든 STM 및 PC를 로드하지 않는 LDM의 경우, 5
- PC를 로드하는 LDM의 경우 4

많은 수의 레지스터를 전송할 수 없도록 하면 다음과 같은 ARM 시스템에서 인터럽트 대기 시간을 줄일 수 있습니다.

- 캐시 또는 작성 버퍼가 없는 시스템 (예: 캐시가 없는 ARM7TDMI)
- 대기 상태가 0인 32비트 메모리를 사용하는 시스템

이외에도 많은 수의 레지스터를 전송할 수 없도록 하면 다음과 같이 됩니다.

- 코드 크기가 항상 증가합니다.
- 캐시된 시스템 또는 작성 버퍼가 있는 프로세서의 경우 중요한 이점이 없습니다.
- 메모리 대기 상태가 0이 아닌 시스템 또는 주변 장치가 느린 시스템의 경우 이점이 없습니다. 이런 시스템의 인터럽트 대기 시간은 가장 느린 메모리 또는 주변 장치 액세스에 필요한 주기 수에 따라 결정됩니다. 일반적으로 이것은 여러 레지스터 전송에 의해 삽입된 대기 시간보다 훨씬 큽니다.

3.1.10 파일에 출력 나열

출력을 파일에 나열하려면 다음 옵션을 사용합니다.

--list *file*

이 옵션은 어셈블러에서 생성한 자세한 어셈블리 언어 목록을 *file*에 출력하도록 해당 어셈블러에 지시합니다.

-을 *file*로 지정하면 목록이 stdout으로 전송됩니다.

*file*을 지정하지 않을 경우 --list=을 사용하여 출력을 *inputfile*.lst로 보냅니다.

참고

파일 이름 없이 --list를 사용하여 출력을 *inputfile*.lst로 보낼 수 있습니다. 그러나 이 구문은 향후 제공되지 않을 예정이므로 이 구문을 사용하면 어셈블러에서 경고를 표시합니다. 이 구문은 이후 릴리스에서 제거됩니다. 대신 --list=를 사용하십시오.

--list 동작을 제어하려면 다음 명령 행 옵션을 사용합니다.

--no_terse
: terse 플래그를 해제합니다. 이 옵션을 설정하면 조건부 어셈블리로 인해 건너뛴 행이 목록에 표시되지 않습니다. terse 옵션을 해제하면 이러한 행이 목록에 표시됩니다. 기본값은 설정입니다.

--width
: 목록 페이지 너비를 설정합니다. 기본값은 79자입니다.

--length
: 목록 페이지 길이를 설정합니다. 길이 0은 페이지 번호가 지정되지 않은 목록을 나타냅니다. 기본값은 66행입니다.

--xref
: 기호에 대한 상호 참조 정보를 나열하도록 어셈블러에 지시합니다. 여기에는 매크로 내부와 외부 모두에서 이러한 기호가 사용되었거나 정의된 위치도 포함됩니다. 기본값은 해제입니다.

3.1.11 프로젝트 템플릿 옵션

프로젝트 템플릿은 특정 구성에 대한 명령 행 옵션과 같은 프로젝트 정보가 포함된 파일입니다. 이러한 파일은 프로젝트 템플릿 작업 디렉토리에 저장됩니다. 다음 옵션을 사용하여 프로젝트 템플릿의 사용을 제어할 수 있습니다.

--[no_]project=*filename*

프로젝트 템플릿 파일을 사용하거나 사용하지 않습니다.

--reinitialize_workdir

프로젝트 템플릿 작업 디렉토리를 다시 초기화할 수 있도록 합니다.

--workdir=*directory*

프로젝트 템플릿의 작업 디렉토리를 제공할 수 있도록 합니다.

이러한 각 옵션에 대한 자세한 내용은 *컴파일러 참조 설명서*에서 다음을 참조하십시오.

- 2-101페이지의 *--[no_]project=filename*
- 2-104페이지의 *--reinitialize_workdir*
- 2-130페이지의 *--workdir=directory*

3.1.12 진단 메시지의 출력 제어

다음은 진단 메시지의 출력을 제어하는 다양한 옵션입니다.

--brief_diagnostics

더 간단한 형식의 진단 출력이 사용되는 모드를 사용하거나 사용하지 않습니다. 이 옵션을 지정하면 원래 소스 행이 표시되지 않고 오류 메시지 텍스트가 너무 길어 한 행에 맞지 않아도 다음 행으로 이어지지 않습니다. 기본값은 --no_brief_diagnostics입니다.

--diag_style {arm|ide|gnu}

진단 메시지를 표시하는 데 사용되는 스타일을 지정합니다.

arm
: ARM 어셈블러 스타일을 사용하여 메시지를 표시합니다. --diag_style을 지정하지 않은 경우 이 옵션이 기본 옵션입니다.

ide
: 오류가 발생한 행의 행 번호 및 문자 수를 포함합니다. 이 값은 괄호 안에 표시됩니다.

gnu
: GNU 스타일을 사용하여 메시지를 표시합니다.

--diag_style=ide 옵션을 선택하면 --brief_diagnostics 옵션이 암시적으로 선택됩니다. 명령 행에서 명시적으로 --no_brief_diagnostics를 선택하면 --diag_style=ide에 의해 암시적으로 선택된 --brief_diagnostics가 무시됩니다.

--diag_style=arm 옵션 또는 --diag_style=gnu 옵션의 선택이 --brief_diagnostics 선택을 의미하지는 않습니다.

`--diag_error` *tag*{,*tag*,…}

지정된 태그가 있는 진단 메시지를 오류 심각도로 설정합니다 (3-19 페이지의 표 3-1 참조).

`--diag_remark` *tag*{,*tag*,…}

지정된 태그가 있는 진단 메시지를 설명 심각도로 설정합니다 (3-19 페이지의 표 3-1 참조).

`--diag_warning` *tag*{,*tag*,…}

지정된 태그가 있는 진단 메시지를 경고 심각도로 설정합니다 (3-19 페이지의 표 3-1 참조).

`--diag_suppress` *tag*{,*tag*,…}

지정된 태그가 있는 진단 메시지를 표시하지 않습니다.

`--unsafe` 다양한 아키텍처의 명령어를 오류 없이 어셈블할 수 있습니다. 해당 오류 메시지를 경고 메시지로 변경합니다. 또한 연산자 우선순위에 대한 경고를 표시하지 않습니다 (3-41페이지의 *바이너리 연산자* 참조).

`--diag_` 옵션 중 네 개는 표시되지 않은 메시지 수를 나타내는 *tag*를 필요로 합니다. 둘 이상의 태그를 지정할 수 있습니다. 예를 들어, 숫자 1293 및 187이 있는 경고 메시지를 표시하지 않으려면 다음 명령을 사용하십시오.

```
armasm --diag_suppress 1293,187 ...
```

어셈블러 접두사 A는 `--diag_error`, `--diag_remark` 및 `--diag_warning`과 함께 사용하거나 메시지를 표시하지 않을 때 사용할 수 있습니다. 예를 들면 다음과 같습니다.

```
armasm --diag_suppress A1293,A187 ...
```

진단 메시지는 잘라내어 명령 행에 직접 붙여 넣을 수 있습니다. 접두사를 사용하는 것은 선택 사항입니다. 그러나 접두사 문자가 포함될 경우 armasm 식별 문자와 일치해야 합니다. 다른 접두사가 있으면 어셈블러에서는 메시지 번호를 무시합니다.

표 3-1에서는 옵션 설명에 사용된 *심각도*의 의미에 대해 설명합니다.

표 3-1 진단 메시지의 심각도

심각도	설명
치명적 오류	치명적 오류는 어셈블리가 중지되는 문제를 나타냅니다. 이 오류에는 명령 행 오류, 내부 오류 및 포함 파일 누락이 포함됩니다.
오류	오류는 어셈블리 언어의 구문 또는 의미 규칙 위반을 나타냅니다. 어셈블리가 계속되지만 객체 코드가 생성되지 않습니다.
경고	경고는 코드에서 문제를 나타낼 수 있는 일반적이지 않은 상황을 나타냅니다. 심각도가 오류인 문제가 더 발견되지 않으면 어셈블리가 계속되고 객체 코드가 생성됩니다.
주의	주의는 일반적이지만 권장되지 않는 어셈블리 언어 사용을 나타냅니다. 이러한 진단은 기본적으로 생성되지 않습니다. 심각도가 오류인 문제가 더 발견되지 않으면 어셈블리가 계속되고 객체 코드가 생성됩니다.

3.1.13 예외 테이블 생성 제어

예외 테이블 생성을 제어하는 옵션에는 다음과 같은 네 가지가 있습니다.

`--exceptions` 발견한 모든 함수에 대해 예외 테이블 생성을 설정하도록 어셈블러에 지시합니다.

`--no_exceptions`

예외 테이블 생성을 해제하도록 어셈블러에 지시합니다. 디버그 테이블은 생성되지 않습니다. 이것이 기본값입니다.

`--exceptions_unwind`

가능한 경우 함수에 대해 *해제* 테이블을 생성하도록 어셈블러에 지시합니다. 이것이 기본값입니다.

`--no_exceptions_unwind`

모든 함수에 대해 *해제 없음* 테이블을 생성하도록 어셈블러에 지시합니다.

보다 세밀하게 제어하려는 경우 `FRAME UNWIND ON` 및 `FRAME UNWIND OFF` 지시어를 사용합니다 (7-54페이지의 *FRAME UNWIND ON* 및 7-54페이지의 *FRAME UNWIND OFF* 참조).

해제 테이블

*function*은 PROC/ENDP 또는 FUNC/ENDFUNC 지시어로 묶인 코드입니다.

예외는 *해제* 테이블이 포함된 함수를 통해 전파될 수 있습니다. 어셈블러에서는 디버그 프레임 정보에서 해제 정보를 생성합니다.

예외는 *해제 없음* 테이블이 포함된 함수를 통해 전파될 수 없습니다. 예외 처리 런타임 환경에서는 예외를 처리하는 동안 *해제 없음* 테이블이 발견되면 프로그램을 종료합니다.

어셈블러에서는 모든 함수 및 비 함수에 대해 *해제 없음* 테이블 항목을 생성할 수 있습니다. 어셈블러에서는 함수 내의 스택 사용을 설명하기에 충분한 FRAME 지시어가 함수에 포함된 경우 함수에 대해 *해제* 테이블을 생성할 수 있습니다. 함수는 EHABI (*ARM 아키텍처용 예외 처리 ABI*), 섹션 9.1 *사용 제한*에 설명된 조건을 준수해야 합니다. 어셈블러에서 *해제* 테이블을 생성할 수 없는 경우에는 *해제 없음* 테이블을 생성합니다.

3.2 소스 행 형식

ARM 어셈블리 언어 모듈에 있는 소스 행의 일반 형식은 다음과 같습니다.

{*symbol*} {*instruction*|*directive*|*pseudo-instruction*} {;*comment*}

소스 행의 세 부분은 모두 선택적입니다.

명령어는 첫 번째 열에서 시작할 수 없습니다. 선행 기호가 없는 경우에도 명령어 앞에는 공백이 있어야 합니다.

명령어, 의사 명령어 또는 지시어는 이 설명서에서와 마찬가지로 모두 대문자로 작성할 수 있으며, 소문자로만 작성할 수도 있습니다. 대소문자를 함께 사용하여 명령어, 의사 명령어 또는 지시어를 작성하면 안 됩니다.

빈 행을 사용하여 코드를 읽기 쉽게 만들 수 있습니다.

*symbol*은 대개 레이블입니다 (3-30페이지의 *레이블* 및 3-31페이지의 *지역 레이블* 참조). 명령어와 의사 명령어에서 기호는 항상 레이블이지만 일부 지시어에서는 변수나 상수의 기호입니다. 지시어 설명을 보면 각 경우를 확실하게 구별할 수 있습니다.

*symbol*은 첫 번째 열에서 시작해야 하고 공백이나 탭과 같은 공백 문자를 포함할 수 없습니다 (3-27페이지의 *기호 명명 규칙* 참조).

3.3 미리 정의된 레지스터 및 보조 프로세서 이름

모든 레지스터 및 보조 프로세서 이름은 대소문자를 구분합니다.

3.3.1 미리 선언된 레지스터 이름

다음 레지스터 이름이 미리 선언됩니다.

- `r0-r15` 및 `R0-R15`
- `a1-a4` (인수, 결과 또는 스크래치 레지스터, r0 ~ r3의 동의어)
- `v1-v8` (변수 레지스터, r4 ~ r11)
- `sb` 및 `SB` (정적 기준, r9)
- `ip` 및 `IP` (내부 프로시저 호출 스크래치 레지스터, r12)
- `sp` 및 `SP` (스택 포인터, r13)
- `lr` 및 `LR` (링크 레지스터, r14)
- `pc` 및 `PC` (프로그램 카운터, r15)

3.3.2 미리 선언된 확장 레지스터 이름

다음 확장 레지스터 이름이 미리 선언됩니다.

- `q0` ~ `q15` 및 `Q0` ~ `Q15` (NEON™ 쿼드워드 레지스터)
- `d0` ~ `d31` 및 `D0` ~ `D31` (NEON 더블워드 레지스터, VFP 배정밀도 레지스터)
- `s0` ~ `s31` 및 `S0` ~ `S31` (VFP 단정밀도 레지스터)

3.3.3 미리 선언된 XScale 레지스터 이름

Intel XScale CPU용으로 어셈블할 때 다음 레지스터 이름이 미리 선언됩니다.

- `acc0 ~ acc7` 및 `ACC0 ~ ACC7` (XScale 누산기)

Wireless MMX가 포함된 Intel XScale CPU용으로 어셈블할 때 다음 레지스터 이름이 미리 선언됩니다.

- `wR0-wR15`, `wr0-wr15` 및 `WR0-WR15`
- `wC0-wC15`, `wc0-wc15` 및 `WC0-WC15`
- `wCID`, `wcid` 및 `WCID`
- `wCon`, `wcon` 및 `WCON`
- `wCSSF`, `wcssf` 및 `WCSSF`
- `wCASF`, `wcasf` 및 `WCASF`

3.3.4 미리 선언된 보조 프로세서 이름

다음 보조 프로세서 이름과 보조 프로세서 레지스터 이름이 미리 선언됩니다.

- `p0 ~ p15` (보조 프로세서 0 ~ 15)
- `c0 ~ c15` (보조 프로세서 레지스터 0 ~ 15)

3.4 기본 제공 변수 및 상수

표 3-2에서는 ARM 어셈블러에서 정의한 기본 제공 변수를 보여 줍니다.

표 3-2 기본 제공 변수

{ARCHITECTURE}	선택된 ARM 아키텍처 이름을 저장합니다.
{AREANAME}	현재 AREA 이름을 저장합니다.
{ARMASM_VERSION}	각 armasm 버전과 함께 증가하는 정수를 저장합니다.
|ads$version|	{ARMASM_VERSION} 값과 동일합니다.
{CODESIZE}	{CONFIG}의 동의어입니다.
{COMMANDLINE}	명령 행의 내용을 저장합니다.
{CONFIG}	어셈블러에서 ARM 코드를 어셈블하면 32 값을 갖고, Thumb 코드를 어셈블하면 16 값을 갖습니다.
{CPU}	선택된 cpu 이름을 저장합니다. 기본값은 "ARM7TDMI"입니다. 명령 행 --cpu 옵션에 아키텍처가 지정되면 {CPU}는 "Generic ARM" 값을 저장합니다.
{ENDIAN}	어셈블러가 빅엔디안 모드에 있으면 "big" 값을 갖고, 리틀엔디안 모드에 있으면 "little" 값을 갖습니다.
{FPIC}	/fpic가 설정되면 bool 값 True를 갖습니다. 기본값은 False입니다.
{FPU}	선택된 fpu 이름을 저장합니다. 기본값은 "SoftVFP"입니다.
{INPUTFILE}	현재 소스 파일의 이름을 저장합니다.
{INTER}	/inter가 설정되면 bool 값 True를 갖습니다. 기본값은 False입니다.
{LINENUM}	현재 소스 파일의 행 번호를 나타내는 정수를 저장합니다.
{OPT}	현재 설정된 목록 옵션의 값입니다. OPT 지시어를 사용하여 현재 목록 옵션을 저장하거나 옵션 내 변경을 지시하거나 원래 값을 복원할 수 있습니다.
{PC} 또는 .	현재 명령어의 주소입니다.
{PCSTOREOFFSET}	STR pc,[...] 또는 STM Rb,{..., pc} 명령어의 주소와 저장된 pc 값 간의 오프셋입니다. 이 값은 지정된 CPU나 아키텍처에 따라 달라집니다.

표 3-2 기본 제공 변수 (계속)

{ROPI}	/ropi가 설정되면 bool 값 True를 갖습니다. 기본값은 False입니다.
{RWPI}	/rwpi가 설정되면 bool 값 True를 갖습니다. 기본값은 False입니다.
{VAR} 또는 @	저장 영역 위치 카운터의 현재 값입니다.

기본 제공 변수는 SETA, SETL 또는 SETS 지시어를 사용하여 설정할 수 없습니다. 기본 제공 변수는 다음과 같은 식 또는 조건에서 사용할 수 있습니다.

```
IF {ARCHITECTURE} = "4T"
```

기본 제공 변수 |ads$version|은 모두 소문자여야 합니다. 기타 기본 제공 변수의 이름은 대문자나 소문자로만 작성하거나 대소문자를 함께 사용하여 작성할 수 있습니다. 예를 들어 다음과 같습니다.

```
IF {CpU} = "Generic ARM"
```

참고

모든 기본 제공 문자열 변수에는 대소문자를 구분하는 값이 포함됩니다. {CPU} 및 {ARCHITECTURE}에 사용할 수 있는 값을 확인하려면 3-12페이지의 *CPU 이름*을 참조하십시오. {FPU}에 사용할 수 있는 값을 확인하려면 3-13페이지의 *FPU 이름*을 참조하십시오. 이러한 기본 제공 변수에 대해 관계형 연산을 수행한 결과는 대소문자가 잘못된 문자열과 일치하지 않습니다.

표 3-3에서는 ARM 어셈블러에서 정의한 기본 제공 bool 상수를 보여 줍니다.

표 3-3 기본 제공 bool 상수

{FALSE}	논리 상수 false입니다.
{TRUE}	논리 상수 true입니다.

3.4.1 armasm 버전 검색

기본 제공 변수 {ARMASM_VERSION}을 사용하여 armasm 버전을 구분할 수 있습니다. 버전 번호의 형식은 *PVtbbb*입니다. 여기서 각 문자는 다음을 의미합니다.

P 주 버전입니다.

V 부 버전입니다.

t 패치 릴리스입니다.

bbb 빌드 번호입니다.

ADS 및 RVCT 이전 ARM 어셈블러에는 기본 제공 변수 |ads$version|이 없습니다. 레거시 개발 도구용 코드 버전을 빌드해야 할 경우 기본 제공 변수 |ads$version|을 테스트할 수 있습니다. 다음과 같은 코드를 사용합니다.

```
IF :DEF: |ads$version|
  ; code for RealView or ADS
ELSE
  ; code for SDT
ENDIF
```

3.5 기호

기호를 사용하여 변수, 주소 및 숫자 상수를 나타낼 수 있습니다. 주소를 나타내는 기호는 *레이블*이라고도 합니다. 다음을 참조하십시오.

- *기호 명명 규칙*
- 3-28페이지의 *변수*
- 3-28페이지의 *숫자 상수*
- 3-29페이지의 *변수의 어셈블리 타임 대체*
- 3-30페이지의 *레이블*
- 3-31페이지의 *지역 레이블*

3.5.1 기호 명명 규칙

기호 이름에는 다음과 같은 일반 규칙이 적용됩니다.

- 기호 이름은 해당 범위 내에서 고유해야 합니다.
- 기호 이름에는 대문자, 소문자, 숫자 상수 또는 밑줄 문자를 사용할 수 있습니다. 기호 이름은 대소문자를 구분하며 기호 이름의 모든 문자가 중요합니다.
- 지역 레이블을 제외하고 기호 이름의 첫 문자로 숫자 문자를 사용하면 안 됩니다 (3-31페이지의 *지역 레이블* 참조).
- 기호는 기본 제공 변수 이름 또는 미리 정의된 기호 이름과 같은 이름을 사용하면 안 됩니다 (3-22페이지의 *미리 정의된 레지스터 및 보조 프로세서 이름* 및 3-24페이지의 *기본 제공 변수 및 상수* 참조).
- 명령어 니모닉이나 지시어와 같은 이름을 사용할 경우 다음과 같이 이중 막대를 사용하여 기호 이름을 구분합니다. 예를 들면 다음과 같습니다.

  ```
  ||ASSERT||
  ```

  막대는 기호의 일부가 아닙니다.
- `|$a|`, `|$t|`, `|$t.x|` 또는 `|$d|` 기호는 프로그램 레이블로 사용하면 안 됩니다. 이러한 기호는 객체 파일 내에서 ARM, Thumb, ThumbEE 및 데이터를 표시하는 데 사용되는 매핑 기호입니다.

컴파일러로 작업하는 경우와 같이 기호에 다양한 문자를 사용해야 할 경우 다음과 같이 단일 막대를 사용하여 기호 이름을 구분합니다. 예를 들면 다음과 같습니다.

```
|.text|
```

막대는 기호의 일부가 아닙니다. 막대 안에서는 막대, 세미콜론 또는 새 행을 사용할 수 없습니다.

3.5.2 변수

변수 값은 어셈블리가 수행될 때 변경될 수 있습니다. 변수 유형에는 다음 세 가지가 있습니다.

- 숫자
- 논리
- 문자열

변수 유형은 변경할 수 없습니다.

숫자 변수의 가능한 값 범위는 숫자 상수나 숫자 식의 가능한 값 범위와 동일합니다 (*숫자 상수* 및 3-34페이지의 *숫자 식* 참조).

논리 변수에 사용할 수 있는 값은 {TRUE} 또는 {FALSE}입니다 (3-37페이지의 *논리 식* 참조).

문자열 변수에 사용할 수 있는 값 범위는 문자열 식의 값 범위와 동일합니다 (3-33페이지의 *문자열 식* 참조).

GBLA, GBLL, GBLS, LCLA, LCLL 및 LCLS 지시어를 사용하여 변수를 나타내는 기호를 선언하고 SETA, SETL 및 SETS 지시어를 사용하여 값을 할당합니다. 다음을 참조하십시오.

- 7-5페이지의 *GBLA, GBLL 및 GBLS*
- 7-7페이지의 *LCLA, LCLL 및 LCLS*
- 7-8페이지의 *SETA, SETL 및 SETS*

3.5.3 숫자 상수

숫자 상수는 32비트 정수입니다. 부호 없는 숫자를 사용하여 0 ~ 2^{32}–1 범위에서 숫자 상수를 설정하거나 부호 있는 숫자를 사용하여 ^{31} ~ 2^3 –1 범위에서 숫자 상수를 설정할 수 있습니다. 그러나 어셈블러에서는 $-n$과 $2^{32}-n$을 구분하지 못합니다. >=와 같은 관계 연산자는 부호 없는 해석을 사용합니다. 즉, 0 > –1은 {FALSE}입니다.

EQU 지시어를 사용하여 상수를 정의합니다 (7-77페이지의 *EQU* 참조). 숫자 상수를 정의한 후에는 해당 값을 변경할 수 없습니다. 숫자 상수와 이항 연산자를 조합하여 숫자 상수 식을 만들 수 있습니다.

3-34페이지의 *숫자 식* 및 3-35페이지의 *숫자 리터럴*도 참조하십시오.

3.5.4 변수의 어셈블리 타임 대체

어셈블리 언어의 전체 행이나 행의 일부에 문자열 변수를 사용할 수 있습니다. 변수를 대체할 값이 있는 위치에 $ 접두사가 포함된 변수를 사용합니다. 달러 문자는 행의 구문을 검사하기 전에 소스 코드 행으로 문자열을 대체하도록 어셈블러에 지시합니다.

숫자 및 논리 변수도 대체할 수 있습니다. 변수의 현재 값은 대체 전에 16진수 문자열 (논리 변수의 경우, T 또는 F) 로 변환됩니다.

기호 이름에 다음 문자를 사용할 수 있는 경우 마침표를 사용하여 변수 이름의 끝을 표시합니다 (3-27페이지의 *기호 명명 규칙* 참조). 변수를 사용하려면 먼저 변수 내용을 설정해야 합니다.

대체하지 않을 $가 필요하면 $$를 사용합니다. 이렇게 하면 단일 $로 변환됩니다.

$ 접두사가 포함된 변수를 문자열에 포함할 수 있습니다. 이 경우 다른 위치와 마찬가지로 대체가 발생합니다.

따옴표로 묶은 세로 막대가 대체에 영향을 주지 않는 것을 제외하고 세로 막대 내에서는 대체가 발생하지 않습니다.

예

```
    ; straightforward substitution
        GBLS    add4ff
        ;
add4ff  SETS    "ADD  r4,r4,#0xFF"    ; set up add4ff
        $add4ff.00                    ; invoke add4ff
        ; this produces
        ADD  r4,r4,#0xFF00
    ; elaborate substitution
            GBLS    s1
            GBLS    s2
            GBLS    fixup
            GBLA    count
            ;
count       SETA    14
s1          SETS    "a$$b$count" ; s1 now has value a$b0000000E
s2          SETS    "abc"
fixup       SETS    "|xy$s2.z|"  ; fixup now has value |xyabcz|
|C$$code|   MOV     r4,#16       ; but the label here is C$$code
```

3.5.5 레이블

레이블은 명령어나 데이터의 메모리에서 주소를 나타내는 기호입니다. 레이블은 프로그램 기준, 레지스터 기준 또는 절대 레이블일 수 있습니다.

프로그램 기준 레이블

이 레이블은 숫자 상수를 더하거나 뺀 PC를 나타냅니다. 이 레이블을 분기 명령어의 타겟으로 사용하거나 코드 섹션에 포함된 데이터의 작은 항목에 액세스합니다. 명령어 또는 데이터 정의 지시어 중 하나에서 레이블을 사용하여 프로그램 기준 레이블을 정의할 수 있습니다. 다음을 참조하십시오.

- 7-22페이지의 *DCB*
- 7-23페이지의 *DCD 및 DCDU*
- 7-25페이지의 *DCFD 및 DCFDU*
- 7-26페이지의 *DCFS 및 DCFSU*
- 7-27페이지의 *DCI*
- 7-29페이지의 *DCQ 및 DCQU*
- 7-30페이지의 *DCW 및 DCWU*

레지스터 기준 레이블

이 레이블은 숫자 상수를 더한 명명된 레지스터를 나타냅니다. 이 레이블은 주로 데이터 섹션의 데이터에 액세스하는 데 사용됩니다. 저장 맵을 사용하여 이 레이블을 정의할 수 있습니다. EQU 지시어를 사용하여 저장소 맵에 정의된 레이블을 기반으로 추가 레지스터 상대 레이블을 정의할 수 있습니다. 다음을 참조하십시오.

- 7-19페이지의 *MAP*
- 7-21페이지의 *SPACE 또는 FILL*
- 7-24페이지의 *DCDO*
- 7-77페이지의 *EQU*

절대 주소

이 레이블은 숫자 상수입니다. 이 레이블은 0 ~ 2^{32}–1 범위의 정수이고 메모리 주소를 직접 지정합니다.

3.5.6 지역 레이블

지역 레이블은 0 ~ 99 범위의 숫자이며, 경우에 따라 숫자 다음에 이름이 올 수 있습니다. 영역에서 둘 이상의 지역 레이블에 동일한 숫자를 사용할 수 있습니다.

지역 레이블은 어셈블리 언어 모듈의 소스 행에서 *symbol* 대신 사용할 수 있습니다 (3-21페이지의 *소스 행 형식* 참조).

- 명령어나 지시어 없이 레이블만 있는 행
- 명령어가 있는 행
- 코드 또는 데이터 생성 지시어가 있는 행

일반적으로 지역 레이블은 프로그램 상대 레이블이 사용되는 위치에서 사용됩니다 (3-30페이지의 *레이블* 참조).

일반적으로 지역 레이블은 루틴 내 루프 및 조건부 코드 또는 지역으로만 사용되는 작은 하위 루틴에 사용됩니다. 지역 레이블은 특히 매크로에서 유용합니다 (7-33페이지의 *MACRO 및 MEND* 참조).

ROUT 지시어를 사용하여 지역 레이블 범위를 제한합니다 (7-88페이지의 *ROUT* 참조). 지역 레이블에 대한 참조는 동일 범위 내 일치 레이블을 참조합니다. 범위 내에서 어느 한 방향으로도 일치하는 레이블이 없으면 오류 메시지가 생성되고 어셈블리가 실패합니다.

같은 범위 내에서도 둘 이상의 지역 레이블에 같은 숫자를 사용할 수 있습니다. 기본적으로 어셈블러에서는 지역 레이블 참조를 다음 항목에 링크합니다.

- 범위 내에 한 숫자가 있는 경우 같은 숫자의 최신 지역 레이블
- 범위 내에 이전 숫자가 있는 경우 같은 숫자의 뒤따르는 지역 레이블

필요한 경우 선택적 매개변수를 사용하여 이 검색 패턴을 수정합니다.

구문

지역 레이블 구문은 다음과 같습니다.

n{*routname*}

지역 레이블에 대한 참조 구문은 다음과 같습니다.

%{F|B}{A|T}*n*{*routname*}

인수 설명:

n	지역 레이블의 수입니다.
routname	현재 범위의 이름입니다.
%	참조를 추가합니다.
F	정방향으로만 검색하도록 어셈블러에 지시합니다.
B	역방향으로만 검색하도록 어셈블러에 지시합니다.
A	모든 매크로 수준을 검색하도록 어셈블러에 지시합니다.
T	현재 매크로 수준만 검색하도록 어셈블러에 지시합니다.

F와 B를 모두 지정하지 않으면 어셈블러에서는 먼저 역방향으로 검색한 다음 정방향으로 검색합니다.

A와 T를 모두 지정하지 않으면 어셈블러에서는 현재 수준에서 최상위까지 모든 매크로를 검색하지만 하위 수준 매크로는 검색하지 않습니다.

레이블이나 레이블 참조에서 *routname*을 지정하면 어셈블러에서는 이 매개변수를 가장 가까운 이전 ROUT 지시어에 대해 확인합니다. 일치하지 않으면 어셈블러에서는 오류 메시지를 생성하고 어셈블리가 실패합니다.

3.6 식, 리터럴 및 연산자

이 단원에는 다음 소단원이 포함되어 있습니다.

- *문자열 식*
- 3-34페이지의 *문자열 리터럴*
- 3-34페이지의 *숫자 식*
- 3-35페이지의 *숫자 리터럴*
- 3-36페이지의 *부동 소수점 리터럴*
- 3-37페이지의 *레지스터 기준 및 프로그램 기준 식*
- 3-37페이지의 *논리 식*
- 3-38페이지의 *논리 리터럴*
- 3-38페이지의 *연산자 우선순위*
- 3-40페이지의 *단항 연산자*
- 3-41페이지의 *바이너리 연산자*

3.6.1 문자열 식

문자열 식은 문자열 리터럴, 문자열 변수, 연산자의 문자열 조작 및 괄호의 조합으로 구성됩니다. 다음을 참조하십시오.

- 3-28페이지의 *변수*
- 3-34페이지의 *문자열 리터럴*
- 3-40페이지의 *단항 연산자*
- 3-42페이지의 *문자열 조작 연산자*
- 7-8페이지의 *SETA, SETL 및 SETS*

문자열 리터럴에 배치할 수 없는 문자는 :CHR: 단항 연산자를 사용하여 문자열 식에 배치할 수 있습니다. 0에서 255 사이의 ASCII 문자를 사용할 수 있습니다.

문자열 식의 값의 길이는 512자를 초과할 수 없으며, 0일 수 있습니다.

예제

```
improb  SETS    "literal":CC: (strvar2:LEFT:4)
            ; sets the variable improb to the value "literal"
            ; with the left-most four characters of the
            ; contents of string variable strvar2 appended
```

3.6.2 문자열 리터럴

문자열 리터럴은 큰 따옴표 문자로 묶인 일련의 문자로 구성됩니다. 문자열 리터럴의 길이는 입력 행의 길이로 제한됩니다 (3-21페이지의 *소스 행 형식* 참조).

문자열에 따옴표나 달러 문자를 포함하려면 다음 두 문자를 사용합니다.

--no_esc를 지정하지 않으면 C 문자열 이스케이프 시퀀스도 사용할 수 있습니다 (3-2페이지의 *명령 구문* 참조).

예

```
abc     SETS    "this string contains only one "" double quote"
def     SETS    "this string contains only one $$ dollar symbol"
```

3.6.3 숫자 식

숫자 식은 숫자 상수, 숫자 변수, 일반 숫자 리터럴, 바이너리 연산자 및 괄호의 조합으로 구성됩니다. 다음을 참조하십시오.

- 3-28페이지의 *숫자 상수*
- 3-28페이지의 *변수*
- 3-35페이지의 *숫자 리터럴*
- 3-41페이지의 *바이너리 연산자*
- 7-8페이지의 *SETA, SETL 및 SETS*

전체 식이 레지스터 또는 PC가 포함되지 않은 값으로 평가될 경우 숫자 식에 레지스터 기준 또는 프로그램 기준 식이 포함될 수 있습니다.

숫자 식은 32비트 정수로 평가됩니다. $0 \sim 2^{32}-1$ 범위의 부호 없는 숫자 또는 $-2^{31} \sim 2^{31}-1$ 범위의 부호 있는 숫자로 숫자 식을 해석할 수 있습니다. 그러나 어셈블러에서는 $-n$과 $2^{32}-n$을 구분하지 못합니다. >=와 같은 관계 연산자는 부호 없는 해석을 사용합니다. 즉, $0 > -1$은 {FALSE}입니다.

예제

```
a   SETA    256*256           ; 256*256 is a numeric expression
    MOV     r1,# (a*22)        ; (a*22) is a numeric expression
```

3.6.4 숫자 리터럴

숫자 리터럴은 다음 형식을 사용할 수 있습니다.

decimal-digits

0xhexadecimal-digits

&hexadecimal-digits

n_base-n-digits

'character'

인수 설명:

decimal-digits 0에서 9 사이의 숫자만 사용하는 문자 시퀀스입니다.

hexadecimal-digits 0에서 9 사이의 숫자와 A에서 F 사이 또는 a에서 f 사이의 문자만 사용하는 문자 시퀀스입니다.

*n_* 2에서 9 사이의 한 자리 숫자로, 뒤에 밑줄 문자가 옵니다.

base-n-digits 0에서 (*n*–1) 사이의 숫자만 사용하는 문자 시퀀스입니다.

character 작은 따옴표를 제외한 단일 문자입니다. 작은 따옴표가 필요하면 \'를 사용합니다. 이 경우 숫자 리터럴 값은 문자의 숫자 코드입니다.

이외의 다른 문자는 사용하면 안 됩니다. 문자 시퀀스는 0에서 2^{32}–1 사이의 정수로 평가되어야 합니다 (0에서 2^{64}–1 사이의 정수로 평가되는 DCQ 및 DCQU 지시어 제외).

예

```
a       SETA    34906
addr    DCD     0xA10E
        LDR     r4,=&1000000F
        DCD     2_11001010
c3      SETA    8_74007
        DCQ     0x0123456789abcdef
        LDR     r1,='A'             ; pseudo-instruction loading 65 into r1
        ADD     r3,r2,#'\''         ; add 39 to contents of r2, result to r3
```

3.6.5 부동 소수점 리터럴

부동 소수점 리터럴에는 다음 형식을 사용할 수 있습니다.

```
{-}digitsE{-}digits
{-}{digits}.digits
{-}{digits}.digitsE{-}digits
0xhexdigits
&hexdigits
0f_hexdigits
0d_hexdigits
```

인수 설명:

digits 0에서 9 사이의 숫자만 사용하는 문자 시퀀스입니다. E는 대문자 또는 소문자일 수 있습니다. 이러한 형식은 일반 부동 소수점 표시에 해당합니다.

hexdigits 0에서 9 사이의 숫자와 A에서 F 사이 또는 a에서 f 사이의 문자만 사용하는 문자 시퀀스입니다. 이러한 형식은 컴퓨터 숫자의 내부 표시에 해당합니다. 무한 값과 NaN을 입력하거나 사용 중인 정확한 비트 패턴을 확인하려는 경우 이러한 형식을 사용합니다.

`0x` 및 `&` 형식을 사용하는 경우 임의의 자릿수의 16진수로 부동 소수점 비트 패턴을 지정할 수 있습니다.

`0f_` 형식을 사용하는 경우 정확히 8자리 8진수로 부동 소수점 비트 패턴을 지정해야 합니다.

`0d_` 형식을 사용하는 경우 정확히 16자리 16진수로 부동 소수점 비트 패턴을 지정해야 합니다.

단정밀도 부동 소수점 값의 범위는 다음과 같습니다.

- 최대 3.40282347e+38
- 최소 1.17549435e–38

배정밀도 부동 소수점 값의 범위는 다음과 같습니다.

- 최대 1.79769313486231571e+308
- 최소 2.22507385850720138e–308

예

```
    DCFD    1E308,-4E-100
    DCFS    1.0
    DCFD    3.725e15
    DCFS    0x7FC00000              ; Quiet NaN
    DCFD    &FFF0000000000000       ; Minus infinity
```

3.6.6 레지스터 기준 및 프로그램 기준 식

레지스터 상대 식은 숫자 상수를 더하거나 뺀 명명된 레지스터로 평가됩니다 (7-19페이지의 *MAP* 참조).

프로그램 상대 주소는 현재 PC (*프로그램 카운터*) 로부터의 오프셋으로 표시되며, 일반적으로 숫자 식과 결합된 레이블입니다.

다음 단계에서는 프로그램 상대 주소가 평가되는 내용을 보여 줍니다.

1. 현재 실행 중인 명령어 *다음*에 나오는 명령어의 주소
2. 비트 OR 0xFFFFFFFC (ARM 코드의 경우에는 차이가 없음)
3. 위의 수에 숫자 상수를 더하거나 뺍니다.

예제

```
        LDR     r4,=data+4*n    ; n is an assembly-time variable
        ; code
        MOV     pc,lr
data    DCD     value_0
        ; n-1 DCD directives
        DCD     value_n         ; data+4*n points here
        ; more DCD directives
```

3.6.7 논리 식

논리 식은 논리 리터럴 ({TRUE} 또는 {FALSE}), 논리 변수, bool 연산자, 관계 및 괄호의 조합으로 구성됩니다 (3-45페이지의 *부울 연산자* 참조).

관계는 변수, 리터럴, 상수 또는 적절한 관계 연산자가 포함된 식의 조합으로 구성됩니다 (3-44페이지의 *관계 연산자* 참조).

3.6.8 논리 리터럴

논리 리터럴은 다음과 같습니다.

- {TRUE}
- {FALSE}

3.6.9 연산자 우선순위

어셈블러에는 식에 사용할 수 있는 다양한 연산자 세트가 포함되어 있습니다. 대부분의 연산자는 C와 같은 상위 언어의 해당 연산자와 비슷합니다 (3-40페이지의 *단항 연산자* 및 3-41페이지의 *바이너리 연산자* 참조).

식을 평가할 때는 다음과 같은 엄격한 우선순위가 적용됩니다.

1. 괄호로 묶은 식이 가장 먼저 평가됩니다.
2. 연산자는 우선순위에 따라 적용됩니다.
3. 인접 단항 연산자는 오른쪽에서 왼쪽으로 평가됩니다.
4. 동일한 우선순위의 바이너리 연산자는 왼쪽에서 오른쪽으로 평가됩니다.

armasm 및 C의 연산자 우선순위

어셈블러 우선순위는 C의 우선순위와 정확히 같지는 않습니다.

예를 들어 armasm에서 `(1 + 2 :SHR: 3)` 은 `(1 + (2 :SHR: 3) ) = 1`로 평가됩니다. C에서 해당 식은 `( (1 + 2) >> 3) = 0`으로 평가됩니다.

괄호를 사용하여 우선순위를 명시적으로 만드는 것이 좋습니다.

코드에 C에서 다르게 구문 분석되는 식이 들어 있고 `--unsafe` 옵션을 사용하지 않으면 armasm에서 대개 경고를 표시합니다.

```
A1466W: Operator precedence means that expression would evaluate differently in C
```

3-39페이지의 표 3-4에서는 armasm의 연산자 우선순위를 보여 주고 C의 우선순위와 비교합니다 (3-39페이지의 표 3-5 참조).

이 테이블에서 우선순위는 다음과 같습니다.

- 최상위 우선순위 연산자는 목록의 맨 위에 있습니다.
- 최상위 우선순위 연산자가 가장 먼저 평가됩니다.

- 같은 우선순위의 연산자는 왼쪽에서 오른쪽으로 평가됩니다.

표 3-4 armasm의 연산자 우선순위

armasm 우선순위	해당 C 연산자
단항 연산자	단항 연산자
* / :MOD:	* / %
문자열 조작	해당 없음
:SHL: :SHR: :ROR: :ROL:	<< >>
+ - :AND: :OR: :EOR:	+ - & | ^
= > >= < <= /= <>	== > >= < <= !=
:LAND: :LOR: :LEOR:	&& ||

표 3-5 C의 연산자 우선순위

C 우선순위
단항 연산자
* / %
+ - (바이너리 연산자)
<< >>
< <= > >=
== !=
&
^
|
&&
||

3.6.10 단항 연산자

단항 연산자의 우선순위가 가장 높고 먼저 평가됩니다. 단항 연산자는 피연산자 앞에 나옵니다. 인접 연산자는 오른쪽에서 왼쪽으로 평가됩니다.

표 3-6에서는 문자열을 반환하는 단항 연산자를 보여 줍니다.

표 3-6 문자열을 반환하는 단항 연산자

연산자	사용법	설명
:CHR:	:CHR:A	ASCII 코드 A가 포함된 문자를 반환합니다.
:LOWERCASE:	:LOWERCASE:string	모든 대문자를 소문자로 변환하여 지정된 문자열을 반환합니다.
:REVERSE_CC:	:REVERSE_CC:cond_code	cond_code에서 조건 코드의 반대 조건을 반환하거나, cond_code에 유효한 조건 코드가 들어 있지 않으면 오류를 반환합니다.
:STR:	:STR:A	논리 식에 사용되는 경우 숫자 식이나 문자열 "T" 또는 "F"에 해당하는 8자리 16진수 문자열을 반환합니다.
:UPPERCASE:	:UPPERCASE:string	모든 소문자를 대문자로 변환하여 지정된 문자열을 반환합니다.

표 3-7에서는 숫자 값을 반환하는 단항 연산자를 보여 줍니다.

표 3-7 숫자 또는 논리 값을 반환하는 단항 연산자

연산자	사용법	설명
?	?A	기호 A를 정의하는 행에서 생성된 실행 가능 코드의 바이트 수입니다.
+ 및 -	+A -A	단항 더하기 및 단항 빼기. + 및 –는 숫자 및 프로그램 기준 식에서 사용할 수 있습니다.
:BASE:	:BASE:A	A가 PC 상대 또는 레지스터 상대 식이면 :BASE:는 레지스터 구성요소의 수를 반환합니다. :BASE:는 매크로에 매우 유용합니다.
:CC_ENCODING:	:CC_ENCODING:cond_code	cond_code에서 조건 코드의 숫자 값을 반환하거나, cond_code에 유효한 조건 코드가 들어 있지 않으면 오류를 반환합니다.
:DEF:	:DEF:A	A가 정의되어 있으면 {TRUE}이고, 그렇지 않으면 {FALSE}입니다.
:INDEX:	:INDEX:A	A가 레지스터 상대 식이면 :INDEX:는 기준 레지스터의 오프셋을 반환합니다. :INDEX:는 매크로에 매우 유용합니다.
:LEN:	:LEN:A	문자열 A의 길이입니다.

표 3-7 숫자 또는 논리 값을 반환하는 단항 연산자 (계속)

연산자	사용법	설명
:LNOT:	:LNOT:A	A의 논리 보수입니다.
:NOT:	:NOT:A	A의 비트 보수입니다 (~은 ~A와 같은 별칭).
:RCONST:	:RCONST:Rn	r0 ~ r15에 해당하는 0 ~ 15의 레지스터 번호입니다.

3.6.11 바이너리 연산자

바이너리 연산자는 해당 하위 식 쌍 사이에서 작성됩니다.

바이너리 연산자의 우선순위는 단항 연산자보다 낮습니다. 바이너리 연산자는 이 섹션에 우선순위에 따라 나타납니다.

참고

이 우선순위는 C의 우선순위와 같지 않습니다 (3-38페이지의 *armasm 및 C의 연산자 우선순위* 참조).

곱하기 연산자

곱하기 연산자는 모든 바이너리 연산자 중 우선순위가 가장 높으며, 숫자 식에만 사용할 수 있습니다.

표 3-8에서는 곱하기 연산자를 보여 줍니다.

표 3-8 곱하기 연산자

연산자	별칭	사용법	설명
*		A*B	곱하기
/		A/B	나누기
:MOD:	%	A:MOD:B	A 모듈로 B

문자열 조작 연산자

표 3-9에서는 문자열 조작 연산자를 보여 줍니다. CC, 에서 A와 B는 둘 다 문자열이어야 합니다. 나누기 연산자에서 LEFT 및 RIGHT의 조건은 다음과 같습니다.

- A는 문자열이어야 합니다.
- B는 숫자 식이어야 합니다.

표 3-9 문자열 조작 연산자

연산자	사용법	설명
:CC:	A:CC:B	B가 A의 끝에 연결됨
:LEFT:	A:LEFT:B	A의 가장 왼쪽 B 문자
:RIGHT:	A:RIGHT:B	A의 가장 오른쪽 B 문자

시프트 연산자

시프트 연산자는 첫 번째 피연산자를 두 번째 피연산자로 지정된 양만큼 시프트 또는 회전하는 방식으로 숫자 식에 사용됩니다.

표 3-10에서는 시프트 연산자를 보여 줍니다.

표 3-10 시프트 연산자

연산자	별칭	사용법	설명
:ROL:		A:ROL:B	A를 B비트만큼 왼쪽으로 회전
:ROR:		A:ROR:B	A를 B비트만큼 오른쪽으로 회전
:SHL:	<<	A:SHL:B	A를 B비트만큼 왼쪽으로 시프트
:SHR:	>>	A:SHR:B	A를 B비트만큼 오른쪽으로 시프트

참고

SHR은 논리 시프트이며 부호 비트를 전파하지 않습니다.

더하기, 빼기 및 논리 연산자

더하기 및 빼기 연산자는 숫자 식에 적용됩니다.

논리 연산자는 숫자 식에 적용됩니다. 연산은 *비트 단위* 즉, 결과를 생성하는 각 피연산자 비트별로 개별적으로 수행됩니다.

표 3-11에서는 더하기, 빼기 및 논리 연산자를 보여 줍니다.

표 3-11 더하기, 빼기 및 논리 연산자

연산자	별칭	사용법	설명
+		A+B	A와 B를 더함
-		A-B	A에서 B를 뺌
:AND:	&	A:AND:B	A와 B의 비트 단위 AND
:EOR:	^	A:EOR:B	A와 B의 비트 단위 배타적 OR
:OR:	|	A:OR:B	A와 B의 비트 단위 OR

관계 연산자

표 3-12에서는 관계 연산자를 보여 줍니다. 이 연산자는 논리 값을 생성하는 동일한 타입의 두 피연산자에 사용됩니다.

피연산자는 다음 중 하나일 수 있습니다.

- 숫자
- 프로그램 기준
- 레지스터 기준
- 문자열

문자열은 ASCII 순서를 사용하여 정렬됩니다. 문자열 A가 문자열 B의 선행 하위 문자열이거나 문자열 A의 가장 왼쪽에 있는 문자가 문자열 B의 가장 왼쪽에 있는 문자보다 작으면 문자열 A가 문자열 B보다 작습니다.

산술 값은 부호가 없으므로 0>-1 값은 {FALSE}입니다.

표 3-12 관계 연산자

연산자	별칭	사용법	설명
=	==	A=B	A와 B가 같음
>		A>B	A가 B보다 큼
>=		A>=B	A가 B보다 크거나 같음
<		A<B	A가 B보다 작음
<=		A<=B	A가 B보다 작거나 같음
/=	<> !=	A/=B	A와 B가 같지 않음

부울 연산자

이 연산자는 우선순위가 가장 낮은 연산자이며 피연산자에 대해 표준 논리 연산을 수행합니다.

세 가지 경우 모두에서 A와 B는 둘 다 {TRUE} 또는 {FALSE}로 평가되는 식이어야 합니다.

표 3-13에서는 부울 연산자를 보여 줍니다.

표 3-13 부울 연산자

연산자	별칭	사용법	설명
:LAND:	&&	A:LAND:B	A와 B의 논리 AND
:LEOR:		A:LEOR:B	A와 B의 논리 배타적 OR
:LOR:	||	A:LOR:B	A와 B의 논리 OR

3.7 진단 메시지

어셈블러는 다양한 추가 진단 메시지를 제공할 수 있습니다. 기본적으로 이러한 진단 메시지는 표시되지 않지만, 명령 행 옵션을 사용하여 어셈블러에서 제공하는 메시지를 제어할 수 있습니다. 자세한 내용은 3-18페이지의 *진단 메시지의 출력 제어*를 참조하십시오.

이 단원에는 다음 소단원이 포함되어 있습니다.

- *인터럭*
- *IT 블록 생성*
- 3-47페이지의 *Thumb 분기 타겟 정렬*

3.7.1 인터럭

--cpu 옵션을 사용하여 선택한 프로세서의 파이프라인으로 인해 코드에서 발생할 수 있는 인터럭에 대한 경고 메시지를 표시할 수 있습니다. 이렇게 하려면 어셈블러를 호출할 때 다음 명령 행 옵션을 사용합니다.

```
armasm --diag_warning 1563
```

참고

--cpu 옵션으로 Cortex-A8과 같은 다중 실행 프로세서를 지정하면 어셈블러 경고를 예상할 수 없습니다.

3.7.2 IT 블록 생성

다음과 같은 코드를 작성하는 경우

```
AREA x,CODE
THUMB
MOVNE   r0,r1 ; (1)
NOP
IT      NE
MOVNE   r0,r1 ; (2)
END
```

어셈블러는 첫 번째 MOVNE 명령어 전에 IT 명령어를 생성합니다.

Thumb 코드를 어셈블할 때 이러한 IT 블록의 자동 생성에 대한 경고 메시지를 얻을 수 있습니다. 이렇게 하려면 어셈블러를 호출할 때 다음 명령 행 옵션을 사용합니다.

```
armasm --diag_warning 1763
```

3.7.3 Thumb 분기 타겟 정렬

일부 프로세서에서 워드로 정렬되지 않은 Thumb 명령어는 루프에서 실행되기 위해 하나 이상의 추가 주기를 사용하는 경우가 있습니다. 따라서 분기 타겟이 워드로 정렬되도록 하는 것이 유용할 수 있습니다. 어셈블러에서는 Thumb 코드의 분기 타겟이 워드로 정렬되지 않은 경우 경고를 생성할 수 있습니다. 이렇게 하려면 어셈블러를 호출할 때 다음 명령 행 옵션을 사용합니다.

```
armasm --diag_warning 1604
```

3.8 C 사전 처리기 사용

어셈블리 언어 소스 파일에서 C 사전 처리기 명령을 사용할 수 있습니다. 이렇게 하는 경우에는 어셈블러를 호출할 때 --cpreproc 명령 행 옵션을 사용해야 합니다. 그러면 armasm이 파일을 어셈블하기 전에 사전 처리하도록 armcc를 호출합니다.

armasm은 armasm 바이너리와 동일한 디렉토리에서 armcc 바이너리를 찾으며, 이 디렉토리에 바이너리가 없으면 PATH에 있다고 간주합니다.

armasm은 명령 행에 있는 경우 armcc에 특정 옵션을 전달합니다. 이러한 옵션이 표 3-14에 나와 있습니다. 이러한 옵션 중 일부는 armcc로 전달하기 전에 동등한 armcc로 변환됩니다. 이러한 옵션이 표 3-15에 나와 있습니다.

표 3-14 명령 행 옵션

--16	--arm_only	--diag_error	--diag_warning	--li
--32	--bi	--diag_remark	--fpu	--library_type
--apcs	--cpu	--diag_style	--fpumode	--thumb
--arm	--device	--diag_suppress	--i	--[no_]unaligned_access

표 3-15 armcc와 동등한 명령 행 옵션

armasm	armcc
--16	--thumb
--32	--arm
--i	--I

사전 처리기 옵션 -D와 같은 다른 간단한 컴파일러 옵션을 전달하려면 --cpreproc_opts 명령 행 옵션을 사용해야 합니다. 자세한 내용은 *컴파일러 사용 설명서*를 참조하십시오. armasm은 사전 처리된 #line 명령을 올바르게 해석합니다. #line 명령의 정보를 사용하여 오류 메시지와 debug_line 테이블을 생성할 수 있습니다.

3-49페이지의 예제 3-1에서는 source.s 파일을 사전 처리하고 어셈블하기 위해 작성하는 명령을 보여 줍니다. 또한 이 예제에서는 컴파일러 옵션을 전달하여 RELEASE라는 매크로를 정의하고 ALPHA라는 매크로의 정의를 취소합니다.

예제 3-1 어셈블리 언어 소스 파일 사전 처리

```
armasm --cpreproc --cpreproc_opts=-D,RELEASE,-U,ALPHA source.s
```

복잡한 사전 처리기 옵션을 사용하려는 경우에는 armasm을 호출하기 전에 armcc를 수동으로 호출하여 파일을 사전 처리해야 합니다. 예제 3-2에서는 source.s 파일을 수동으로 사전 처리 및 어셈블하기 위해 작성하는 명령을 보여 줍니다. 이 예제에서는 사전 처리기에서 preprocessed.s라는 파일을 출력하고 armasm에서 preprocessed.s를 어셈블합니다.

예제 3-2 수동으로 어셈블리 언어 소스 파일 사전 처리

```
armcc -E source.s > preprocessed.s
armasm preprocessed.s
```

4장
ARM 및 Thumb 명령어

이 장에서는 ARM 어셈블러에서 지원하는 ARM®, Thumb® (모든 버전) 및 ThumbEE 명령어에 대해 설명합니다. 여기에는 다음 단원이 포함되어 있습니다.

- 4-3페이지의 *명령어 요약*
- 4-10페이지의 *메모리 액세스 명령어*
- 4-41페이지의 *일반 데이터 처리 명령어*
- 4-71페이지의 *곱하기 명령어*
- 4-92페이지의 *포화 명령어*
- 4-97페이지의 *병렬 명령어*
- 4-105페이지의 *패킹 및 패킹 해제 명령어*
- 4-113페이지의 *분기 및 제어 명령어*
- 4-123페이지의 *보조 프로세서 명령어*
- 4-131페이지의 *기타 제한*
- 4-148페이지의 *Thumb에서 명령어 너비 선택*
- 4-150페이지의 *ThumbEE 명령어*
- 4-154페이지의 *의사 명령어*

일부 명령어 단원에는 아키텍처 소단원이 있습니다. 아키텍처 소단원이 없는 명령어는 모든 버전의 ARM 명령어 세트와 모든 버전의 Thumb 명령어 세트에서 사용할 수 있습니다.

4.1 명령어 요약

표 4-1에서는 ARM, Thumb 및 ThumbEE 명령어 세트에서 사용할 수 있는 명령어에 대해 간략히 설명합니다. 이 표를 사용하면 이 장의 나머지 부분에서 설명하는 개별 명령어와 의사 명령어를 찾을 수 있습니다.

참고

다른 언급이 없는 한 ThumbEE 명령어는 Thumb 명령어와 동일합니다.

표 4-1 명령어 위치

니모닉	간단한 설명	페이지	아키텍처 [a]
ADC, ADD	carry 포함 더하기, 더하기	4-45페이지	모두
ADR	프로그램 또는 레지스터 기준 주소 로드 (짧은 범위)	4-23페이지	모두
ADRL 의사 명령어	프로그램 또는 레지스터 기준 주소 로드 (중간 범위)	4-155페이지	x6M
AND	논리 AND	4-51페이지	모두
ASR	오른쪽으로 산술 시프트	4-67페이지	모두
B	분기	4-114페이지	모두
BFC, BFI	비트 필드 지우기 및 삽입	4-106페이지	T2
BIC	비트 지우기	4-51페이지	모두
BKPT	브레이크포인트	4-132페이지	5
BL	링크 포함 분기	4-114페이지	모두
BLX	링크 포함 분기 및 변경 명령어 세트	4-114페이지	T
BX	분기, 변경 명령어 세트	4-114페이지	T
BXJ	분기, Jazelle로 변경	4-114페이지	J, x7M
CBZ, CBNZ	0인 경우 (0이 아닌 경우) 비교 및 분기	4-120페이지	T2
CDP	보조 프로세서 데이터 처리 연산	4-124페이지	x6M
CDP2	보조 프로세서 데이터 처리 연산	4-124페이지	5, x6M

표 4-1 명령어 위치 (계속)

니모닉	간단한 설명	페이지	아키텍처 [a]
CHKA	배열 검사	4-152페이지	EE
CLREX	단독 지우기	4-39페이지	K, x6M
CLZ	선행 0 수 계산	4-54페이지	5, x6M
CMN, CMP	음수 비교, 비교	4-55페이지	모두
CPS	프로세서 상태 변경	4-138페이지	6
DBG	디버그	4-144페이지	7
DMB, DSB	데이터 메모리 장벽, 데이터 동기화 장벽	4-144페이지	7, 6M
ENTERX, LEAVEX	ThumbEE로 또는 ThumbEE에서 상태 변경	4-151페이지	EE
EOR	배타적 OR	4-51페이지	모두
HB, HBL, HBLP, HBP	처리기 분기, 지정된 처리기로 분기	4-153페이지	EE
ISB	명령어 동기화 장벽	4-144페이지	7, 6M
IT	If-Then	4-118페이지	T2
LDC	보조 프로세서 로드	4-129페이지	x6M
LDC2	보조 프로세서 로드	4-129페이지	5, x6M
LDM	다중 레지스터 로드	4-27페이지	모두
LDR	레지스터 로드 명령어	4-10페이지	모두
LDR 의사 명령어	레지스터 로드 의사 명령어	4-159페이지	모두
LDREX	단독 레지스터 로드	4-36페이지	6, x6M
LDREXB, LDREXH	바이트 및 하프워드 단독 레지스터 로드	4-36페이지	K, x6M
LDREXD	더블워드 단독 레지스터 로드	4-36페이지	K, x7M
LSL, LSR	왼쪽으로 논리 시프트, 오른쪽으로 논리 시프트	4-67페이지	모두
MAR	레지스터에서 40비트 누산기로 이동	4-147페이지	XScale
MCR	레지스터에서 보조 프로세서로 이동	4-125페이지	x6M

표 4-1 명령어 위치 (계속)

니모닉	간단한 설명	페이지	아키텍처 [a]
MCR2	레지스터에서 보조 프로세서로 이동	4-125페이지	5, x6M
MCRR	여러 레지스터에서 보조 프로세서로 이동	4-125페이지	5E, x6M
MCRR2	여러 레지스터에서 보조 프로세서로 이동	4-125페이지	6, x6M
MIA, MIAPH, MIAxy	내부 40비트 누산으로 곱하기	4-90페이지	XScale
MLA	곱하기 누산	4-72페이지	x6M
MLS	곱하기 및 빼기	4-72페이지	T2
MOV	이동	4-57페이지	모두
MOVT	맨 위로 이동	4-60페이지	T2
MOV32 의사 명령어	레지스터로 32비트 상수 이동	4-157페이지	T2
MRA	40비트 누산기에서 레지스터로 이동	4-147페이지	XScale
MRC	보조 프로세서에서 레지스터로 이동	4-127페이지	모두
MRC2	보조 프로세서에서 레지스터로 이동	4-127페이지	5, x6M
MRS	PSR에서 레지스터로 이동	4-134페이지	모두
MSR	레지스터에서 PSR로 이동	4-136페이지	모두
MUL	곱하기	4-72페이지	모두
MVN	이동하지 않음	4-57페이지	모두
NOP	연산 없음	4-142페이지	모두
ORN	논리 OR NOT	4-51페이지	T2
ORR	논리 OR	4-51페이지	모두
PKHBT, PKHTB	하프워드 패킹	4-111페이지	6, x7M
PLD	데이터 사전 로드	4-25페이지	5E, x6M
PLDW	쓰기를 위해 데이터 사전 로드	4-25페이지	7MP
PLI	사전 로드 명령어	4-25페이지	7

표 4-1 명령어 위치 (계속)

니모닉	간단한 설명	페이지	아키텍처 [a]
PUSH, POP	스택으로 레지스터 푸시, 스택에서 레지스터 팝	4-30페이지	모두
QADD, QDADD, QDSUB, QSUB	포화 산술	4-93페이지	5E, x7M
QADD8, QADD16, QASX, QSUB8, QSUB16, QSAX	병렬 부호 있는 포화 산술	4-98페이지	6, x7M
RBIT	비트 반전	4-65페이지	T2
REV, REV16, REVSH	바이트 순서 반전	4-65페이지	6
RFE	예외에서 복귀	4-32페이지	T2, x7M
ROR	레지스터를 오른쪽으로 회전	4-67페이지	모두
RSB	역방향 빼기	4-45페이지	모두
RSC	carry 포함 역방향 빼기	4-45페이지	x6M
SADD8, SADD16, SASX	병렬 부호 있는 산술	4-98페이지	6, x7M
SBC	carry 포함 빼기	4-45페이지	모두
SBFX, UBFX	부호 있는 및 부호 없는 비트 필드 추출	4-107페이지	T2
SDIV	부호 있는 나누기	4-70페이지	7M, 7R
SEL	APSR GE 플래그에 따라 바이트 선택	4-63페이지	6, x7M
SETEND	메모리 액세스에 엔디안 설정	4-141페이지	6, x7M
SEV	이벤트 설정	4-142페이지	K, 6M
SHADD8, SHADD16, SHASX, SHSUB8, SHSUB16, SHSAX	병렬 부호 있는 양분 산술	4-98페이지	6, x7M
SMC	보안 모니터 호출	4-140페이지	Z
SMLAD	이중 부호 있는 곱하기 누산 (32 <= 32 + 16 x 16 + 16 x 16)	4-85페이지	6, x7M
SMLAL	부호 있는 곱하기 누산 (64 <= 64 + 32 x 32)	4-74페이지	x6M

표 4-1 명령어 위치 (계속)

니모닉	간단한 설명	페이지	아키텍처 [a]
SMLALxy	부호 있는 곱하기 누산 (64 <= 64 + 16 x 16)	4-79페이지	5E, x7M
SMLALD	long에 대한 이중 부호 있는 곱하기 누산 (64 <= 64 + 16 x 16 + 16 x 16)	4-87페이지	6, x7M
SMLSD	이중 부호 있는 곱하기 빼기 누산 (32 <= 32 + 16 x 16 – 16 x 16)	4-85페이지	6, x7M
SMLSLD	long에 대한 이중 부호 있는 곱하기 빼기 누산 (64 <= 64 + 16 x 16 – 16 x 16)	4-87페이지	6, x7M
SMMUL	부호 있는 상위 워드 곱하기 (32 <= TopWord (32 x 32))	4-83페이지	6, x7M
SMUAD, SMUSD	이중 부호 있는 곱하기 및 결과 더하기 또는 빼기	4-81페이지	6, x7M
SMULxy	부호 있는 곱하기 (32 <= 16 x 16)	4-76페이지	5E, x7M
SMULL	부호 있는 곱하기 (64 <= 32 x 32)	4-74페이지	x6M
SMULWy	부호 있는 곱하기 (32 <= 32 x 16)	4-78페이지	5E, x7M
SRS	반환 상태 저장	4-34페이지	T2, x7M
SSAT	부호 있는 포화	4-95페이지	6, x6M
SSAT16	부호 있는 포화, 병렬 하프워드	4-103페이지	6, x7M
SSUB8, SSUB16, SSAX	병렬 부호 있는 산술	4-98페이지	6, x7M
STC	보조 프로세서 저장	4-129페이지	x6M
STC2	보조 프로세서 저장	4-129페이지	5, x6M
STM	다중 레지스터 저장	4-27페이지	모두
STR	레지스터 저장 명령어	4-10페이지	모두
STREX	단독 레지스터 저장	4-36페이지	6, x6M
STREXB, STREXH	단독 레지스터 저장 바이트, 하프워드	4-36페이지	K, x6M
STREXD	더블워드 단독 레지스터 저장	4-36페이지	K, x7M
SUB	빼기	4-45페이지	모두

표 4-1 명령어 위치 (계속)

니모닉	간단한 설명	페이지	아키텍처 [a]
SUBS pc, lr	스택 없이 예외에서 반환	4-49페이지	T2, x7M
SVC (이전 SWI)	관리자 호출	4-133페이지	모두
SWP, SWPB	레지스터와 메모리 스왑 (ARM에만 해당)	4-40페이지	모두, x7M
SXTB, SXTB16, SXTH	부호 있는 확장	4-108페이지	6
SXTAB, SXTAB16, SXTAH	더하기 포함 부호 있는 확장	4-108페이지	6, x7M
TBB, TBH	테이블 분기 바이트, 하프워드	4-122페이지	T2
TEQ, TST	동등 테스트, 테스트	4-61페이지	모두
UADD8, UADD16, UASX	병렬 부호 없는 산술	4-98페이지	6, x7M
UDIV	부호 없는 나누기	4-70페이지	7M, 7R
UHADD8, UHADD16, UHASX, UHSUB8, UHSUB16, UHSAX	병렬 부호 없는 양분 산술	4-98페이지	6, x7M
UMAAL	long에 대한 부호 없는 곱하기 누산 (64 <= 32 + 32 + 32 x 32)	4-89페이지	6, x7M
UMLAL, UMULL	부호 없는 곱하기 누산, 곱하기 (64 <= 32 x 32 + 64), (64 <= 32 x 32)	4-74페이지	x6M
UQADD8, UQADD16, UQASX, UQSUB8, UQSUB16, UQSAX	병렬 부호 없는 포화 산술	4-98페이지	6, x7M
USAD8	부호 없는 절대차의 합	4-101페이지	6, x7M
USADA8	부호 없는 절대차의 합 누산	4-101페이지	6, x7M
USAT	부호 없는 포화	4-95페이지	6, x6M
USAT16	부호 없는 포화, 병렬 하프워드	4-103페이지	6, x7M
USUB8, USUB16, USAX	병렬 부호 없는 산술	4-98페이지	6, x7M
UXTB, UXTB16, UXTH	부호 없는 확장	4-108페이지	6

표 4-1 명령어 위치 (계속)

니모닉	간단한 설명	페이지	아키텍처 [a]
UXTAB, UXTAB16, UXTAH	더하기 포함 부호 없는 확장	4-108페이지	6, x7M
V*	자세한 내용은 5장 *NEON 및 VFP 프로그래밍*을 참조하십시오.		
WFE, WFI, YIELD	이벤트 대기, 인터럽트 대기, 양도	4-142페이지	T2, 6M

a. 아키텍처 열의 항목은 다음과 같은 의미를 갖습니다.

모두 해당 명령어는 모든 버전의 ARM 아키텍처에서 사용할 수 있습니다.
5 해당 명령어는 ARMv5T*, ARMv6* 및 ARMv7 아키텍처에서 사용할 수 있습니다.
5E 해당 명령어는 ARMv5TE, ARMv6* 및 ARMv7 아키텍처에서 사용할 수 있습니다.
6 해당 명령어는 ARMv6* 및 ARMv7 아키텍처에서 사용할 수 있습니다.
6M 해당 명령어는 ARMv6-M 및 ARMv7 아키텍처에서 사용할 수 있습니다.
x6M 해당 명령어는 ARMv6-M 프로파일에서 사용할 수 없습니다.
7 해당 명령어는 ARMv7 아키텍처에서 사용할 수 있습니다.
7M 해당 명령어는 ARMv7-M 프로파일에서 사용할 수 있습니다.
x7M 해당 명령어는 ARMv6-M 또는 ARMv7-M 프로파일에서 사용할 수 없습니다.
7R 해당 명령어는 ARMv7-R 프로필에서 사용할 수 있습니다.
7MP 해당 명령어는 다중 처리 확장을 구현하는 ARMv7 아키텍처에서 사용할 수 있습니다.
EE 해당 명령어는 ARM 아키텍처의 ThumbEE 변형에서 사용할 수 있습니다.
J 해당 명령어는 ARMv5TEJ, ARMv6* 및 ARMv7 아키텍처에서 사용할 수 있습니다.
K 해당 명령어는 ARMv6K 및 ARMv7 아키텍처에서 사용할 수 있습니다.
T 해당 명령어는 ARMv4T, ARMv5T*, ARMv6* 및 ARMv7 아키텍처에서 사용할 수 있습니다.
T2 해당 명령어는 ARMv6T2 이상 아키텍처에서 사용할 수 있습니다.
XScale 해당 명령어는 ARM 아키텍처의 XScale 버전에서 사용할 수 있습니다.
Z 해당 명령어는 보안 확장이 구현된 경우 사용할 수 있습니다.

4.2 메모리 액세스 명령어

이 단원에는 다음 소단원이 포함되어 있습니다.

- 4-11페이지의 *주소 정렬*
  모든 메모리 액세스 명령어에 적용되는 정렬에 대한 고려 사항
- 4-13페이지의 *LDR 및 STR (즉치 오프셋)*
  즉치 오프셋, 사전 인덱싱된 즉치 오프셋 또는 사후 인덱싱된 즉치 오프셋을 사용하여 로드 및 저장
- 4-16페이지의 *LDR 및 STR (레지스터 오프셋)*
  레지스터 오프셋, 사전 인덱싱된 레지스터 오프셋 또는 사후 인덱싱된 레지스터 오프셋을 사용하여 로드 및 저장
- 4-18페이지의 *LDR 및 STR (사용자 모드)*
  사용자 모드 권한을 사용하여 로드 및 저장
- 4-20페이지의 *LDR (pc 기준)*
  레지스터 로드. 주소는 pc를 기준으로 한 오프셋입니다.
- 4-23페이지의 *ADR*
  프로그램 기준 또는 레지스터 기준 주소 로드
- 4-25페이지의 *PLD, PLDW 및 PLI*
  나중에 사용할 주소 사전 로드
- 4-27페이지의 *LDM 및 STM*
  다중 레지스터 로드 및 저장
- 4-30페이지의 *PUSH 및 POP*
  스택에 하위 레지스터 및 lr (옵션) 푸시
  스택에서 하위 레지스터 및 pc (옵션) 팝
- 4-32페이지의 *RFE*
  예외에서 복귀
- 4-34페이지의 *SRS*
  반환 상태 저장

- 4-36페이지의 *LDREX 및 STREX*
  단독 레지스터 로드 및 저장
- 4-39페이지의 *CLREX*
  단독 지우기
- 4-40페이지의 *SWP 및 SWPB*
  레지스터와 메모리 간 데이터 스왑

참고

그 외에도 LDR 의사 명령어가 있습니다 (4-159페이지의 *LDR 의사 명령어* 참조). 이 의사 명령어는 LDR 명령어로 어셈블되거나 MOV 또는 MVN 명령어로 어셈블됩니다.

4.2.1 주소 정렬

대부분의 경우 4바이트로 전송할 주소는 4바이트 워드로 정렬해야 하고 2바이트로 전송할 주소는 2바이트 워드로 정렬해야 합니다. ARMv6T2 이상에서는 정렬되지 않은 액세스가 허용됩니다. ARMv7 이상에서는 정렬되지 않은 액세스가 사용 가능하며 기본 액세스 방법입니다.

ARMv6 이하에서는 시스템에 시스템 보조 프로세서 (cp15) 가 있을 경우 정렬 검사를 사용할 수 있습니다. 워드로 정렬되지 않은 32비트 전송은 정렬 검사를 사용할 경우 정렬 예외를 발생시킵니다.

모든 액세스가 정렬되는 경우 --no_unaligned_access 명령 행 옵션을 사용하면, 정렬되지 않는 옵션을 가질 수 있는 라이브러리 함수에서 링크를 방지할 수 있습니다.

시스템에 시스템 보조 프로세서 (cp15) 가 없거나 정렬 검사를 사용하지 않을 경우 다음 사항이 적용됩니다.

- STR의 경우 지정된 주소가 4의 배수로 잘립니다.
- LDR의 경우
  1. 지정된 주소가 4의 배수로 잘립니다.
  2. 4바이트의 데이터가 결과 주소에서 로드됩니다.
  3. 주소의 비트[1:0]에 따라 로드된 데이터가 1, 2 또는 3바이트씩 오른쪽으로 회전합니다.

이 경우 리틀엔디안 메모리 시스템에서는 주소가 지정된 바이트가 레지스터의 최하위 바이트를 차지합니다.

이 경우 빅엔디안 메모리 시스템에서는 주소가 지정된 바이트가 다음 바이트를 차지합니다.

— 주소의 비트[0]이 0일 경우, 비트[31:24]

— 주소의 비트[0]이 1일 경우, 비트[15:8]

4.2.2 LDR 및 STR (즉치 오프셋)

즉치 오프셋, 사전 인덱싱된 즉치 오프셋 또는 사후 인덱싱된 즉치 오프셋을 사용하여 로드 및 저장합니다.

구문

```
op{type}{cond} Rt, [Rn {, #offset}]        ; immediate offset
op{type}{cond} Rt, [Rn, #offset]!          ; pre-indexed
op{type}{cond} Rt, [Rn], #offset           ; post-indexed
opD{cond} Rt, Rt2, [Rn {, #offset}]        ; immediate offset, doubleword
opD{cond} Rt, Rt2, [Rn, #offset]!          ; pre-indexed, doubleword
opD{cond} Rt, Rt2, [Rn], #offset           ; post-indexed, doubleword
```

인수 설명:

op 다음 중 하나일 수 있습니다.

- LDR 레지스터 로드
- STR 레지스터 저장

type 다음 중 하나일 수 있습니다.

- B 부호 없는 바이트 (로드 시 32비트로 0 확장)
- SB 부호 있는 바이트 (LDR에만 해당, 32비트로 부호 확장)
- H 부호 없는 하프워드 (로드 시 32비트로 0 확장)
- SH 부호 있는 하프워드 (LDR에만 해당, 32비트로 부호 확장)
- - 워드용으로, 생략됨

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

Rt 로드 또는 저장할 레지스터입니다.

Rn 메모리 주소의 기준이 되는 레지스터입니다.

offset 오프셋입니다. *offset*이 생략되면 주소는 *Rn*의 내용입니다.

Rt2 더블워드 연산의 경우 로드 또는 저장할 추가 레지스터입니다.

모든 옵션을 모든 명령어 세트와 아키텍처에서 사용할 수 있는 것은 아닙니다. 자세한 내용은 4-14페이지의 *오프셋 범위 및 아키텍처*를 참조하십시오.

오프셋 범위 및 아키텍처

표 4-2에서는 이러한 명령어의 오프셋 범위와 사용 가능성을 보여 줍니다.

표 4-2 오프셋 및 아키텍처, LDR/STR, 워드, 하프워드 및 바이트

명령어	즉치 오프셋	사전 인덱싱된 오프셋	사후 인덱싱된 오프셋	아키텍처
ARM, 워드 또는 바이트 [a]	–4095 ~ 4095	–4095 ~ 4095	–4095 ~ 4095	모두
ARM, 부호 있는 바이트, 하프워드 또는 부호 있는 하프워드	–255 ~ 255	–255 ~ 255	–255 ~ 255	모두
ARM, 더블워드	–255 ~ 255	–255 ~ 255	–255 ~ 255	v5TE +
32비트 Thumb, 워드, 하프워드, 부호 있는 하프워드, 바이트 또는 부호 있는 바이트 [a]	–255 ~ 4095	–255 ~ 255	–255 ~ 255	v6T2, v7
32비트 Thumb, 더블워드	–1020 ~ 1020 [c]	–1020 ~ 1020 [c]	–1020 ~ 1020 [c]	v6T2, v7
16비트 Thumb, 워드 [b]	0 ~ 124 [c]	사용할 수 없음	사용할 수 없음	모든 T
16비트 Thumb, 부호 없는 하프워드 [b]	0 ~ 62 [d]	사용할 수 없음	사용할 수 없음	모든 T
16비트 Thumb, 부호 없는 바이트 [b]	0 ~ 31	사용할 수 없음	사용할 수 없음	모든 T
16비트 Thumb, 워드, Rn은 r13임 [e]	0 ~ 1020 [c]	사용할 수 없음	사용할 수 없음	모든 T
16비트 ThumbEE, 워드 [b]	–28 ~ 124 [c]	사용할 수 없음	사용할 수 없음	T-2EE
16비트 ThumbEE, 워드, Rn은 r9임 [e]	0 ~ 252 [c]	사용할 수 없음	사용할 수 없음	T-2EE
16비트 ThumbEE, 워드, Rn은 r10임 [e]	0 ~ 124 [c]	사용할 수 없음	사용할 수 없음	T-2EE

a. 워드 로드의 경우 Rt는 pc일 수 있습니다. pc로 로드하면 로드된 주소로 분기됩니다. ARMv4에서는 로드된 주소의 비트[1:0]이 0b00이어야 합니다. ARMv5 이상에서는 비트[1:0]이 0b10이면 안 됩니다. 비트[0]이 1이면 실행이 Thumb 상태에서 계속되고, 그렇지 않으면 실행이 ARM 상태에서 계속됩니다.

b. Rt 및 Rn은 r0 ~ r7 범위에 있어야 합니다.

c. 4의 배수여야 합니다.

d. 2의 배수여야 합니다.

e. Rt는 r0 ~ r7 범위에 있어야 합니다.

더블워드 레지스터 제한

Thumb-2 명령어의 경우, *Rt* 또는 *Rt2*에 sp 또는 pc를 지정하면 안 됩니다.

ARM 명령어의 경우

- *Rt*는 짝수의 레지스터여야 합니다.
- *Rt*는 lr이면 안 됩니다.
- *Rt*에는 r12를 사용하지 않는 것이 좋습니다.
- *Rt2*가 R (*t* + 1)이어야 합니다.

예제

```
LDR     r8,[r10]            ; loads r8 from the address in r10.
LDRNE   r2,[r5,#960]!       ; (conditionally) loads r2 from a word
                            ; 960 bytes above the address in r5, and
                            ; increments r5 by 960.
STR     r2,[r9,#consta-struc]   ; consta-struc is an expression evaluating
                                ; to a constant in the range 0-4095.
```

4.2.3 LDR 및 STR (레지스터 오프셋)

레지스터 오프셋, 사전 인덱싱된 레지스터 오프셋 또는 사후 인덱싱된 레지스터 오프셋을 사용하여 로드 및 저장합니다.

구문

```
op{type}{cond} Rt, [Rn, +/-Rm {, shift}]   ; register offset
op{type}{cond} Rt, [Rn, +/-Rm {, shift}]!  ; pre-indexed
op{type}{cond} Rt, [Rn], +/-Rm {, shift}   ; post-indexed
opD{cond} Rt, Rt2, [Rn, +/-Rm {, shift}]   ; register offset, doubleword
opD{cond} Rt, Rt2, [Rn, +/-Rm {, shift}]!  ; pre-indexed, doubleword
opD{cond} Rt, Rt2, [Rn], +/-Rm {, shift}   ; post-indexed, doubleword
```

인수 설명:

op 다음 중 하나일 수 있습니다.

- LDR 레지스터 로드
- STR 레지스터 저장

type 다음 중 하나일 수 있습니다.

- B 부호 없는 바이트 (로드 시 32비트로 0 확장)
- SB 부호 있는 바이트 (LDR에만 해당, 32비트로 부호 확장)
- H 부호 없는 하프워드 (로드 시 32비트로 0 확장)
- SH 부호 있는 하프워드 (LDR에만 해당, 32비트로 부호 확장)
- - 워드용으로, 생략됨

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

Rt 로드 또는 저장할 레지스터입니다.

Rn 메모리 주소의 기준이 되는 레지스터입니다.

Rm 오프셋으로 사용할 값이 포함된 레지스터입니다. *Rm* 은 r15이면 안 됩니다. Thumb 코드에서는 *-Rm*을 사용할 수 없습니다.

shift 선택적 시프트입니다.

Rt2 더블워드 연산의 경우 로드 또는 저장할 추가 레지스터입니다.

모든 옵션을 모든 명령어 세트와 아키텍처에서 사용할 수 있는 것은 아닙니다. 자세한 내용은 *오프셋 레지스터 및 시프트 옵션*을 참조하십시오.

오프셋 레지스터 및 시프트 옵션

표 4-3에서는 이러한 명령어의 오프셋 범위와 사용 가능성을 보여 줍니다.

표 4-3 옵션 및 아키텍처, LDR/STR (레지스터 오프셋)

명령어	*+/–Rm* [a]	시프트	아키텍처
ARM, 워드 또는 바이트 [b]	*+/–Rm*	LSL #0-31 LSR #1-32 ASR #1-32 ROR #1-31 RRX	모두
ARM, 부호 있는 바이트, 하프워드 또는 부호 있는 하프워드	*+/–Rm*	사용할 수 없음	모두
ARM, 더블워드	*+/–Rm*	사용할 수 없음	v5TE +
32비트 Thumb, 워드, 하프워드, 부호 있는 하프워드, 바이트 또는 부호 있는 바이트 [b]	*+Rm*	LSL #0-3	v6T2, v7
32비트 Thumb, 더블워드	*+Rm*	사용할 수 없음	v6T2, v7
16비트 Thumb, 모두 [c]	*+Rm*	사용할 수 없음	모든 T
16비트 ThumbEE, 워드 [b]	*+Rm*	LSL #2 (필수)	T-2EE
16비트 ThumbEE, 하프워드, 부호 있는 하프워드 [b]	*+Rm*	LSL #1 (필수)	T-2EE
16비트 ThumbEE, 바이트, 부호 있는 바이트 [b]	*+Rm*	사용할 수 없음	T-2EE

a. +/–Rm이 표시되면 –Rm, +Rm 또는 Rm을 사용할 수 있습니다. +Rm이 표시되면 –Rm을 사용할 수 없습니다.
b. 워드 로드의 경우 Rt는 pc일 수 있습니다. pc로 로드하면 로드된 주소로 분기됩니다. ARMv4에서는 로드된 주소의 비트[1:0]이 0b00이어야 합니다. ARMv5 이상에서는 비트[1:0]이 0b10이면 안 됩니다. 비트[0]이 1이면 실행이 Thumb 상태에서 계속되고, 그렇지 않으면 실행이 ARM 상태에서 계속됩니다.
c. Rt, Rn 및 Rm은 모두 r0 ~ r7 범위에 있어야 합니다.

더블워드 레지스터 제한

Thumb-2 명령어의 경우, *Rt* 또는 *Rt2*에 sp 또는 pc를 지정하면 안 됩니다.

ARM 명령어의 경우

- *Rt*는 짝수의 레지스터여야 합니다.
- *Rt*는 lr이면 안 됩니다.
- Rt에는 r12를 사용하지 않는 것이 좋습니다.
- *Rt2* 는 R (t + 1)이어야 합니다.

4.2.4 LDR 및 STR (사용자 모드)

사용자 모드 권한을 사용하여 바이트, 하프워드 또는 워드 로드 및 저장합니다.

이러한 명령어가 권한 모드에서 실행되면 사용자 모드에서 실행된 경우와 동일한 제한을 갖고 메모리에 액세스합니다.

사용자 모드에서 이러한 명령어는 일반 메모리 액세스와 똑같은 방식으로 동작합니다.

구문

```
op{type}T{cond} Rt, [Rn {, #offset}]       ; immediate offset (Thumb-2 only)
op{type}T{cond} Rt, [Rn] {, #offset}       ; post-indexed (ARM only)
op{type}T{cond} Rt, [Rn], +/-Rm {, shift}  ; post-indexed (register) (ARM only)
```

인수 설명:

op 다음 중 하나일 수 있습니다.

- LDR 레지스터 로드
- STR 레지스터 저장

type 다음 중 하나일 수 있습니다.

- B 부호 없는 바이트 (로드 시 32비트로 0 확장)
- SB 부호 있는 바이트 (LDR에만 해당, 32비트로 부호 확장)
- H 부호 없는 하프워드 (로드 시 32비트로 0 확장)
- SH 부호 있는 하프워드 (LDR에만 해당, 32비트로 부호 확장)
- \- 워드용으로, 생략됨

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

Rt 로드 또는 저장할 레지스터입니다.

Rn 메모리 주소의 기준이 되는 레지스터입니다.

offset 오프셋입니다. 오프셋이 생략되면 주소는 *Rn*의 값입니다.

Rm 오프셋으로 사용할 값이 포함된 레지스터입니다. *Rm* 은 r15이면 안 됩니다.

shift 선택적 시프트입니다.

오프셋 범위 및 아키텍처

4-14페이지의 표 4-2에서는 이러한 명령어의 오프셋 범위와 사용 가능성을 보여 줍니다.

표 4-4 오프셋 및 아키텍처, LDR/STR (사용자 모드)

명령어	즉치 오프셋	사후 인덱싱된 오프셋	*+/–Rm* [a]	시프트	아키텍처
ARM, 워드 또는 바이트	사용할 수 없음	–4095 ~ 4095	*+/–Rm*	LSL #0-31 LSR #1-32 ASR #1-32 ROR #1-31 RRX	모두
ARM, 부호 있는 바이트, 하프워드 또는 부호 있는 하프워드	사용할 수 없음	–255 ~ 255	*+/–Rm*	사용할수 없음	모두
32비트 Thumb, 워드, 하프워드, 부호 있는 하프워드, 바이트 또는 부호 있는 바이트	0 ~ 255	사용할 수 없음	사용할 수 없음		v6T2, v7

a. –Rm, +Rm 또는 Rm을 사용할 수 있습니다.

4.2.5 LDR (pc 기준)

레지스터 로드. 주소는 pc를 기준으로 한 오프셋입니다.

참고

4-154페이지의 *의사 명령어*도 참조하십시오.

구문

```
LDR{type}{cond}{.W} Rt, label
LDRD{cond} Rt, Rt2, label        ; Doubleword
```

인수 설명:

type 다음 중 하나일 수 있습니다.

- B 부호 없는 바이트 (로드 시 32비트로 0 확장)
- SB 부호 있는 바이트 (LDR에만 해당, 32비트로 부호 확장)
- H 부호 없는 하프워드 (로드 시 32비트로 0 확장)
- SH 부호 있는 하프워드 (LDR에만 해당, 32비트로 부호 확장)
- - 워드용으로, 생략됨

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

.W 선택적 명령어 너비 지정자입니다. 자세한 내용은 4-21페이지의 *Thumb-2의 LDR (pc 기준)*을 참조하십시오.

Rt 로드 또는 저장할 레지스터입니다.

Rt2 로드 또는 저장할 두 번째 레지스터입니다.

label 프로그램 기준 식입니다. 자세한 내용은 3-38페이지의 *레지스터 기준 및 프로그램 기준 식*을 참조하십시오.

*label*은 현재 명령어의 제한된 거리 안에 있어야 합니다. 자세한 내용은 4-21페이지의 *오프셋 범위 및 아키텍처*를 참조하십시오.

오프셋 범위 및 아키텍처

어셈블러는 자동으로 pc를 기준으로 오프셋을 계산하며, *label*이 범위를 벗어날 경우 오류를 생성합니다.

표 4-5에서는 레이블과 현재 명령어 사이의 가능한 오프셋을 보여 줍니다.

표 4-5 pc 기준 오프셋

명령어	오프셋 범위	아키텍처
ARM LDR, LDRB, LDRSB, LDRH, LDRSH [a]	+/– 4095	모두
ARM LDRD	+/– 255	v5TE +
32비트 Thumb LDR, LDRB, LDRSB, LDRH, LDRSH [a]	+/– 4095	v6T2, v7
32비트 Thumb LDRD	+/– 1020 [b]	v6T2, v7
16비트 Thumb LDR [c]	0-1020 [b]	모든 T

a. 워드 로드의 경우 Rt는 pc일 수 있습니다. pc로 로드하면 로드된 주소로 분기됩니다. ARMv4에서는 로드된 주소의 비트[1:0]이 0b00이어야 합니다. ARMv5 이상에서는 비트[1:0]이 0b10이면 안 됩니다. 비트[0]이 1이면 실행이 Thumb 상태에서 계속되고, 그렇지 않으면 실행이 ARM 상태에서 계속됩니다.
b. 4의 배수여야 합니다.
c. Rt는 r0 ~ r7 범위에 있어야 합니다. 바이트, 하프워드 또는 더블워드 16비트 명령어는 없습니다.

Thumb-2의 LDR (pc 기준)

.W 너비 지정자를 사용하여 Thumb-2 코드에서 32비트 명령어를 생성하도록 LDR에 지시할 수 있습니다. LDR.W는 16비트 LDR을 사용하여 타겟에 도달할 수 있는 경우에도 항상 32비트 명령어를 생성합니다.

정방향 참조의 경우, .W가 없는 LDR은 32비트 Thumb-2 LDR 명령어를 사용하여 도달할 수 있는 타겟에서 오류를 발생시키는 경우에도 Thumb 코드에서 항상 16비트 명령어를 생성합니다.

더블워드 레지스터 제한

Thumb-2 명령어의 경우, *Rt* 또는 *Rt2*에 sp 또는 pc를 지정하면 안 됩니다.

ARM 명령어의 경우

- *Rt*는 짝수의 레지스터여야 합니다.
- *Rt*는 lr이면 안 됩니다.
- *Rt*에는 r12를 사용하지 않는 것이 좋습니다.
- *Rt2* 는 R (t + 1)이어야 합니다.

4.2.6 ADR

ADR은 즉치값을 pc 값에 더하고 결과를 대상 레지스터에 기록합니다.

구문

ADR{*cond*}{.W} *Rd*,*label*

인수 설명:

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

.W 선택적 명령어 너비 지정자입니다. 자세한 내용은 4-24페이지의 *Thumb-2의 ADR*을 참조하십시오.

Rd 로드할 레지스터입니다.

label 프로그램 기준 식입니다. 자세한 내용은 3-38페이지의 *레지스터 기준 및 프로그램 기준 식*을 참조하십시오.

*label*은 현재 명령어의 제한된 거리 안에 있어야 합니다. 자세한 내용은 4-24페이지의 *오프셋 범위 및 아키텍처*를 참조하십시오.

사용법

ADR은 주소가 프로그램 기준 또는 레지스터 기준 주소이기 때문에 위치 독립적인 코드를 생성합니다.

ADRL 의사 명령어를 사용하면 더 넓은 범위의 유효 주소를 어셈블할 수 있습니다 (4-155페이지의 *ADRL 의사 명령어* 참조).

*label*이 프로그램 기준 주소이면 ADR 명령어와 동일한 어셈블러 영역에 있는 주소로 평가되어야 합니다 (7-70페이지의 *AREA* 참조).

ADR을 사용하여 BX 또는 BLX 명령어의 타겟을 생성하는 경우 타겟에 Thumb 명령어가 포함되어 있으면 주소의 Thumb 비트 (비트 0) 를 설정하는 것은 사용자의 책임입니다.

오프셋 범위 및 아키텍처

어셈블러는 자동으로 pc를 기준으로 오프셋을 계산하며, *label*이 범위를 벗어날 경우 오류를 생성합니다.

4-21페이지의 표 4-5에서는 레이블과 현재 명령어 사이의 가능한 오프셋을 보여줍니다.

표 4-6 pc 기준 오프셋

명령어	오프셋 범위	아키텍처
ARM ADR	자세한 내용은 4-43페이지의 *Operand2의 상수*를 참조하십시오.	모두
32비트 Thumb ADR	+/– 4095	v6T2, v7
16비트 Thumb ADR [a]	0-1020 [b]	모든 T

a. Rd는 r0 ~ r7 범위에 있어야 합니다.
b. 4의 배수여야 함

Thumb-2의 ADR

.W 너비 지정자를 사용하여 Thumb-2 코드에서 32비트 명령어를 생성하도록 ADR에 지시할 수 있습니다. .W가 있는 ADR 은 주소가 16비트 명령어로 생성될 수 있는 경우에도 항상 32비트 명령어를 생성합니다.

정방향 참조의 경우, .W 가 없는 ADR 은 32비트 Thumb-2 ADD 명령어로 생성할 수 있는 주소에서 오류를 발생시킬 수 있는 경우에도 Thumb 코드에서 항상 16비트 명령어를 생성합니다.

4.2.7 PLD, PLDW 및 PLI

데이터 및 명령어 사전 로드. 프로세서는 주소에서 데이터나 명령어를 곧 로드할 것이라는 신호를 메모리 시스템에 보낼 수 있습니다.

구문

```
PLtype{cond} [Rn {, #offset}]
PLtype{cond} [Rn, +/-Rm {, shift}]
PLtype{cond} label
```

인수 설명:

type 은 다음 중 하나일 수 있습니다.

- D 데이터 주소
- DW 쓰기를 위한 데이터 주소
- I 명령어 주소

구문이 *label*을 지정하는 경우 *type*은 DW일 수 없습니다.

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

참고

*cond*는 위의 IT 명령어를 통해 Thumb-2 코드에서만 사용할 수 있습니다. 이는 ARM의 무조건 명령어이며 *cond*를 사용해서는 안 됩니다.

Rn 메모리 주소의 기준이 되는 레지스터입니다.

offset 즉치 오프셋입니다. 오프셋이 생략되면 주소는 *Rn*의 값입니다.

Rm 오프셋으로 사용할 값이 포함된 레지스터입니다. *Rm* 은 r15이면 안 됩니다. Thumb 명령어의 경우 *Rm*은 r13이어서도 안 됩니다.

shift 선택적 시프트입니다.

label 프로그램 기준 식입니다. 자세한 내용은 3-38페이지의 *레지스터 기준 및 프로그램 기준 식*을 참조하십시오.

오프셋 범위

오프셋은 사전 로드가 발생하기 전에 *Rn*의 값에 적용됩니다. 이 결과는 사전 로드할 메모리 주소로 사용됩니다. 허용되는 오프셋 범위는 다음과 같습니다.

- ARM 명령어의 경우, –4095 ~ +4095
- Thumb-2 명령어의 경우 *Rn*이 r15가 아니면 –255 ~ +4095
- Thumb-2 명령어의 경우 *Rn*이 r15이면 –4095 ~ +4095

어셈블러에서는 자동으로 pc를 기준으로 오프셋을 계산하며, *label*이 범위를 벗어날 경우 오류를 생성합니다.

레지스터 또는 시프트된 레지스터 오프셋

ARM에서는 *Rn*의 값에서 *Rm*의 값을 더하거나 뺍니다. Thumb-2에서는 *Rm*의 값을 *Rn*의 값에 더할 수만 있습니다. 이 결과는 사전 로드할 메모리 주소로 사용됩니다.

허용되는 시프트 범위는 다음과 같습니다.

- Thumb-2 명령어의 경우, LSL #0 ~ #3
- ARM 명령어의 경우 다음 중 하나
  - LSL #0 ~ #31
  - LSR #1 ~ #32
  - ASR #1 ~ #32
  - ROR #1 ~ #31
  - RRX

사전 로드할 주소 정렬

사전 로드 명령어에 대해서는 정렬 검사가 수행되지 않습니다.

아키텍처

ARM PLD는 ARMv5TE 이상에서 사용할 수 있고

32비트 Thumb PLD는 ARMv6T2 이상에서 사용할 수 있습니다.

PLDW는 다중 처리 확장을 구현하는 ARMv7 이상에서만 사용할 수 있습니다.

PLI는 ARMv7 이상에서만 사용할 수 있습니다.

16비트 Thumb PLD, PLDW 또는 PLI 명령어는 없습니다.

힌트 명령어 구현 여부는 옵션입니다. 두 명령어는 구현되어 있지 않은 경우 NOP로 실행됩니다.

4.2.8 LDM 및 STM

다중 레지스터 로드 및 저장. ARM 상태에서는 레지스터 r0 ~ r15의 모든 조합을 전송할 수 있지만 Thumb 상태에서는 일부 조합을 전송하지 못할 수 있습니다.

4-30페이지의 *PUSH 및 POP*도 참조하십시오.

구문

op{*addr_mode*}{*cond*} *Rn*{!}, *reglist*{^}

인수 설명:

op 다음 중 하나일 수 있습니다.

- LDM 다중 레지스터 로드
- STM 다중 레지스터 저장

addr_mode 다음 중 하나입니다.

- IA 각 전송 후에 주소를 증가시킵니다. 기본값이며 생략할 수 있습니다.
- IB 각 전송 전에 주소를 증가시킵니다 (ARM에만 해당).
- DA 각 전송 후에 주소를 감소시킵니다 (ARM에만 해당).
- DB 각 전송 전에 주소를 감소시킵니다.

스택 지향 주소 지정 모드 접미사는 2-44페이지의 표 2-9를 참조하십시오.

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

Rn *기준 레지스터*로, 전송할 초기 주소가 들어 있는 ARM 레지스터입니다. *Rn*은 r15이면 안 됩니다.

! 선택적 접미사입니다. ! 기호가 있을 경우 최종 주소가 *Rn*에 다시 기록됩니다.

reglist 로드 또는 저장할 하나 이상의 레지스터가 포함된 목록으로, 중괄호로 묶입니다. 이 목록에는 레지스터 범위가 포함될 수도 있습니다. 이 목록에 하나 이상의 레지스터나 레지스터 범위가 포함될 경우 콤마로 구분해야 합니다 (4-29페이지의 *예제* 참조).

자세한 내용은 *32비트 Thumb-2 명령어의 reglist에 대한 제한*을 참조하십시오.

^ ARM 상태에서만 사용할 수 있는 선택적 접미사로, 사용자 모드나 시스템 모드에서는 사용할 수 없습니다. 이 접미사의 용도는 다음과 같습니다.

- 명령어가 LDM (임의의 주소 지정 모드 포함) 이고 *reglist*에 일반 다중 레지스터 전송 외에도 pc (r15) 가 포함되어 있는 경우 SPSR이 CPSR로 복사됩니다. 이는 예외 처리기에서 복귀하기 위한 것으로, 예외 모드에서만 사용해야 합니다.
- 또는 현재 모드 레지스터 대신 사용자 모드 레지스터 내부 또는 외부로 데이터를 전송합니다.

32비트 Thumb-2 명령어의 reglist에 대한 제한

32비트 Thumb-2 명령어의 경우

- 목록에 SP를 포함할 수 없습니다.
- STM 명령어의 목록에 pc를 포함할 수 없습니다.
- LDM 명령어의 목록에 pc와 lr을 둘 다 포함할 수 없습니다.
- 이 목록에는 둘 이상의 레지스터가 있어야 합니다.

reglist에서 레지스터를 하나만 사용하여 STM 또는 LDM 명령어를 기록하는 경우 어셈블러에서 해당하는 STR 또는 LDR 명령어를 자동으로 대체합니다. 디스어셈블리 목록을 소스 코드와 비교할 때는 이러한 사항에 주의해야 합니다.

`--diag_warning 1645` 어셈블러 명령 행 옵션을 사용하여 명령어 대체가 발생하는 시기를 확인할 수 있습니다.

16비트 명령어

이러한 명령어의 16비트 버전 하위 세트는 Thumb-2 코드 및 다른 Thumb-2 이전 프로세서의 Thumb 코드에서 사용할 수 있습니다.

16비트 명령어에는 다음 제한이 적용됩니다.

- *reglist*의 모든 레지스터가 Lo 레지스터여야 합니다.
- *Rn* 이 Lo 레지스터여야 합니다.
- *addr_mode*가 생략되어야 하거나 IA여야 합니다. 즉, 각 전송 후에 주소가 증가해야 합니다.
- STM 명령어에 대해 갱신을 지정해야 합니다.

- *Rn*이 *reglist*에 없는 LDM 명령어에 대해 갱신을 지정해야 합니다.

또한 PUSH 및 POP 명령어를 이 형식으로 표시할 수 있습니다. 일부 형식의 PUSH 및 POP은 16비트 명령어이기도 합니다. 자세한 내용은 4-30페이지의 *PUSH 및 POP*을 참조하십시오.

참고

이러한 16비트 명령어는 Thumb-2EE에서 사용할 수 없습니다.

pc로 로드

pc로 로드하면 로드된 주소에 있는 명령어로 분기됩니다.

ARMv4에서는 로드된 주소의 비트[1:0]이 0b00이어야 합니다.

ARMv5T 이상의 경우

- 비트[1:0]은 0b10이면 안 됩니다.
- 비트[0]이 1이면 실행이 Thumb 상태에서 계속됩니다.
- 비트[0]이 0이면 실행이 ARM 상태에서 계속됩니다.

쓰기 되돌림을 사용하여 기준 레지스터 로드 및 저장

ARM 코드 또는 Thumb-2 이전의 Thumb 코드에서 *Rn*이 *reglist*에 있고 ! 접미사로 갱신이 지정될 경우 다음 사항이 적용됩니다.

- 명령어가 STM 또는 STMIA이고 *Rn*이 *reglist*에서 가장 작은 숫자의 레지스터일 경우 *Rn*의 초기값이 저장됩니다.
- *Rn*의 로드되거나 저장된 값은 신뢰할 수 없습니다.

Thumb-2 코드에서 *Rn*이 *reglist*에 있고 ! 접미사로 갱신이 지정될 경우 다음 사항이 적용됩니다.

- 모든 32비트 명령어를 예상할 수 없습니다.
- 16비트 명령어는 Thumb-2 이전의 Thumb 코드에서와 동일한 방식으로 동작하지만 이러한 명령어는 향후 사용할 수 없습니다.

예제

```
LDM     r8,{r0,r2,r9}       ; LDMIA is a synonym for LDM
STMDB   r1!,{r3-r6,r11,r12}
```

올바르지 않은 예제

```
STM     r5!,{r5,r4,r9} ; value stored for r5 unpredictable
LDMDA   r2, {}         ; must be at least one register in list
```

4.2.9 PUSH 및 POP

전체 내림차순 스택에 레지스터 푸시 및 전체 내림차순 스택에서 레지스터 팝

구문

PUSH{*cond*} *reglist*

POP{*cond*} *reglist*

인수 설명:

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

reglist 중괄호로 묶인 비어 있지 않은 레지스터의 목록으로, 레지스터 범위를 포함할 수 있습니다. 이 목록에 하나 이상의 레지스터나 레지스터 범위가 포함될 경우 콤마로 구분해야 합니다.

사용법

PUSH와 POP은 기준 레지스터가 sp (r13) 이고 조정된 주소가 기준 레지스터에 다시 기록된 STMDB와 LDM (또는 LDMIA) 의 동의어입니다. 이러한 경우 권장되는 니모닉은 PUSH와 POP입니다.

레지스터는 스택에 번호 순서대로 저장됩니다. 즉, 가장 낮은 번호의 레지스터가 최하위 주소에 저장됩니다.

pc가 있는 reglist를 사용하는 POP

이 명령어는 스택에서 pc로 팝된 주소로 분기를 생성합니다. 일반적으로 이 분기는 시작 위치에 있는 스택에 lr이 푸시된 하위 루틴에서 되돌아옵니다.

ARMv5T 이상의 경우

- 비트[1:0]은 0b10이면 안 됩니다.
- 비트[0]이 1이면 실행이 Thumb 상태에서 계속됩니다.
- 비트[0]이 0이면 실행이 ARM 상태에서 계속됩니다.

ARMv4에서는 로드된 주소의 비트[1:0]이 0b00이어야 합니다. 상태를 변경하는 데 POP을 사용할 수 없습니다.

Thumb 명령어

Thumb 명령어 세트에서는 이러한 명령어의 하위 세트를 사용할 수 있습니다.

16비트 명령어에는 다음 제한이 적용됩니다.

- PUSH의 경우 *reglist*에는 Lo 레지스터 및 lr만 포함될 수 있습니다.
- For POP의 경우 *reglist*에는 Lo 레지스터 및 pc만 포함될 수 있습니다.

32비트 명령어에는 다음 제한이 적용됩니다.

- *reglist*는 sp를 포함해서는 안 됩니다.
- PUSH의 경우 *reglist*는 pc를 포함해서는 안 됩니다.
- POP의 경우 *reglist*는 lr 또는 pc 중 하나만 포함할 수 있습니다.

예제

```
PUSH    {r0,r4-r7}
PUSH    {r2,lr}
POP     {r0,r10,pc} ; no 16-bit version available
```

4.2.10 RFE

예외에서 복귀

구문

RFE{*addr_mode*}{*cond*} *Rn*{!}

인수 설명:

addr_mode 다음 중 하나입니다.

- IA 각 전송 후에 주소를 증가시킵니다 (전체 내림차순 스택).
- IB 각 전송 전에 주소를 증가시킵니다 (ARM에만 해당).
- DA 각 전송 후에 주소를 감소시킵니다 (ARM에만 해당).
- DB 각 전송 전에 주소를 감소시킵니다.

*addr_mode*가 생략된 경우 기본적으로 각 전송 후에 주소를 증가시킵니다.

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

참고

*cond*는 위의 IT 명령어를 통해 Thumb-2 코드에서만 사용할 수 있습니다. 이는 ARM의 무조건 명령어입니다.

Rn 기준 레지스터를 지정합니다. *Rn*에 r15를 사용하면 안 됩니다.

! 선택적 접미사입니다. ! 기호가 있을 경우 최종 주소가 *Rn*에 다시 기록됩니다.

사용법

이전에 SRS 명령어 (4-34페이지의 *SRS* 참조) 를 사용하여 반환 상태를 저장한 경우 RFE를 사용하여 예외에서 복귀할 수 있습니다. *Rn*은 대개 반환 상태 정보를 저장한 sp입니다.

Thumb-2EE에서는 기준 레지스터의 값이 0일 경우 실행이 HandlerBase - 4에서 NullCheck 처리기로 분기됩니다.

연산

*Rn*에 들어 있는 주소와 그 다음 주소에서 pc와 CPSR을 로드하고, 경우에 따라 *Rn*을 업데이트합니다.

메모

RFE는 pc에 주소를 기록합니다. 이 주소는 다음과 같이 예외 반환 후 사용 중인 명령어 세트를 기준으로 올바로 정렬되어야 합니다.

- ARM으로 복귀할 경우 pc에 기록된 주소가 워드로 정렬되어야 합니다.
- Thumb-2로 복귀할 경우 pc에 기록된 주소가 하프워드로 정렬되어야 합니다.
- Jazelle®로 복귀할 경우 pc에 기록된 주소의 정렬에 대한 제한이 없습니다.

이러한 규칙을 위반하면 예상할 수 없는 결과가 발생합니다. 그러나 유효한 예외 엔트리 메커니즘 후에 이 명령어를 사용하여 복귀한 경우 소프트웨어에서 특별한 예방 조치를 취하지 않아도 됩니다.

주소가 워드로 정렬되지 않은 경우 RFE는 *Rn*의 최하위 2비트를 무시합니다.

RFE가 생성하는 메모리의 개별 워드에 액세스하는 시간 순서는 아키텍처 면에서 정의되지 않습니다. 따라서 액세스 순서가 중요한, 메모리에 매핑된 I/O 위치에 대해서는 이 명령어를 사용하면 안 됩니다.

*mode*에 의해 사용자 모드가 지정될 경우 CPSR에 쓰기에 대해 일반 규칙이 적용됩니다. 자세한 내용은 *ARM 아키텍처 참조 문서*를 참조하십시오.

*mode*에 의해 모니터 모드가 지정될 경우 예상할 수 없는 결과가 발생합니다 (4-140 페이지의 *SMC* 참조).

아키텍처

이 ARM 명령어는 ARMv6 이상에서 사용할 수 있습니다.

이 32비트 Thumb 명령어는 ARMv7-M 프로필을 제외하고 ARMv6T2 이상에서 사용할 수 있습니다.

이 명령어의 16비트 버전은 없습니다.

예제

```
RFE sp!
```

4.2.11 SRS

스택에 반환 상태를 저장합니다.

구문

```
SRS{addr_mode}{cond} sp{!}, #modenum

SRS{addr_mode}{cond} #modenum{!}        ; This is a pre-UAL syntax
```

인수 설명:

addr_mode 다음 중 하나입니다.

- IA 각 전송 후에 주소를 증가시킵니다.
- IB 각 전송 전에 주소를 증가시킵니다 (ARM에만 해당).
- DA 각 전송 후에 주소를 감소시킵니다 (ARM에만 해당).
- DB 각 전송 전에 주소를 증가시킵니다 (전체 내림차순 스택).

*addr_mode*가 생략된 경우 기본적으로 각 전송 후에 주소를 증가시킵니다. 스택 지향 주소 지정 모드 접미사는 2-44페이지의 표 2-9를 참조하십시오.

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

참고

*cond*는 위의 IT 명령어를 통해 Thumb-2 코드에서만 사용할 수 있습니다. 이는 ARM의 무조건 명령어입니다.

! 선택적 접미사입니다. ! 기호가 있을 경우 최종 주소가 *modenum*에 의해 지정된 모드의 sp에 다시 기록됩니다.

modenum 해당 뱅크 sp가 기준 레지스터로 사용되는 모드의 번호를 지정합니다 (2-6페이지의 *프로세서 모드* 참조).

연산

SRS는 현재 모드의 lr과 SPSR을 *modenum*에 의해 지정된 모드의 sp에 들어 있는 주소와 그 다음 워드에 각각 저장하고, 경우에 따라 *modenum*에 의해 지정된 모드의 sp를 업데이트합니다. 이 명령어는 스택 액세스를 위한 일반적인 용도의 STM 명령어와 함께 사용할 수 있습니다 (4-27페이지의 *LDM 및 STM* 참조).

참고

전체 내림차순 스택의 경우에는 SRSFD 또는 SRSDB를 사용해야 합니다.

사용법

SRS를 사용하면 자동으로 선택된 스택이 아닌 다른 스택에 예외 처리기의 반환 상태를 저장할 수 있습니다.

Thumb-2EE에서는 기준 레지스터의 값이 0일 경우 실행이 HandlerBase - 4에서 NullCheck 처리기로 분기됩니다.

메모

주소가 워드로 정렬되지 않은 경우 SRS는 지정된 주소의 최하위 2비트를 무시합니다.

SRS가 생성하는 메모리의 개별 워드에 액세스하는 시간 순서는 아키텍처 면에서 정의되지 않습니다. 따라서 액세스 순서가 중요한, 메모리에 매핑된 I/O 위치에 대해서는 이 명령어를 사용하면 안 됩니다.

사용자 및 시스템 모드에서는 SPSR이 없으므로 SRS를 예상할 수 없습니다.

아키텍처

이 ARM 명령어는 ARMv6 이상에서 사용할 수 있습니다.

이 32비트 Thumb 명령어는 ARMv7-M 프로필을 제외하고 ARMv6T2 이상에서 사용할 수 있습니다.

이 명령어의 16비트 버전은 없습니다.

예제

```
R13_usr  EQU     16
         SRSFD   sp,#R13_usr
```

4.2.12 LDREX 및 STREX

단독 레지스터 로드 및 저장

구문

LDREX{*cond*} *Rt*, [*Rn* {, *#offset*}]

STREX{*cond*} *Rd*, *Rt*, [*Rn* {, *#offset*}]

LDREXB{*cond*} *Rt*, [*Rn*]

STREXB{*cond*} *Rd*, *Rt*, [*Rn*]

LDREXH{*cond*} *Rt*, [*Rn*]

STREXH{*cond*} *Rd*, *Rt*, [*Rn*]

LDREXD{*cond*} *Rt*, *Rt2*, [*Rn*]

STREXD{*cond*} *Rd*, *Rt*, *Rt2*, [*Rn*]

인수 설명:

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

Rd 반환된 상태에 대한 대상 레지스터입니다.

Rt 로드 또는 저장할 레지스터입니다.

Rt2 더블워드를 로드 또는 저장할 두 번째 대상 레지스터입니다.

Rn 메모리 주소의 기준이 되는 레지스터입니다.

offset *Rn*의 값에 적용되는 선택적 오프셋입니다. *offset*은 Thumb-2 명령어에서만 사용할 수 있습니다. *offset*이 생략될 경우 오프셋은 0으로 간주됩니다.

LDREX

LDREX는 메모리에서 데이터를 로드합니다.

- 물리 주소에 공유 TLB 속성이 있을 경우, LDREX는 물리 주소를 현재 프로세서의 단독 액세스로 태그 설정하고 다른 모든 물리 주소에서 이 프로세서의 단독 액세스 태그를 지웁니다.
- 또는 실행 프로세서에 미결정 태그가 설정된 실제 주소가 있다는 사실을 태그 설정합니다.

STREX

STREX는 메모리에 대한 조건부 저장을 수행합니다. 조건은 다음과 같습니다.

- 물리 주소에 공유 TLB 속성이 없고 실행 프로세서에 미결정 태그가 설정된 물리 주소가 있는 경우 저장 작업이 수행되고 태그가 지워지며 *Rd*에서 값 0이 반환됩니다.
- 물리 주소에 공유 TLB 속성이 없고 실행 프로세서에 미결정 태그가 설정된 물리 주소가 없는 경우 저장 작업이 수행되지 않고 *Rd*에서 값 1이 반환됩니다.
- 물리 주소에 공유 TLB 속성이 있고 물리 주소가 실행 프로세서의 단독 액세스로 태그 설정된 경우 저장 작업이 수행되고 태그가 지워지며 *Rd*에서 값 0이 반환됩니다.
- 물리 주소에 공유 TLB 속성이 있고 물리 주소가 실행 프로세서의 단독 액세스로 태그 설정되지 않은 경우 저장 작업이 수행되지 않고 *Rd*에서 값 1이 반환됩니다.

제한

Rd, *Rt*, *Rt2* 또는 *Rn*에 r15를 사용하면 안 됩니다.

STREX의 경우, *Rd*가 *Rt*, *Rt2* 또는 *Rn*과 동일한 레지스터이면 안 됩니다.

ARM 명령어의 경우

- *Rt* 가 짝수의 레지스터여야 하고 r14이면 안 됩니다.
- *Rt2*가 *R (t+1)* 이어야 합니다.
- *offset*은 허용되지 않습니다.

Thumb 명령어의 경우

- *Rd*, *Rt* 또는 *Rt2*에 r13을 사용하면 안 됩니다.
- LDREXD의 경우 *Rt* 및 *Rt2*가 동일한 레지스터이면 안 됩니다.
- *offset*의 값은 0 ~ 1020 범위에 있는 4의 배수일 수 있습니다.

사용법

LDREX 및 STREX를 사용하여 다중 프로세서와 공유 메모리 시스템에서 프로세서 간 통신을 구현합니다.

성능상의 이유로 해당 LDREX 명령어와 STREX 명령어 사이의 명령어 수를 최소로 유지해야 합니다.

참고

STREX 명령어에 사용된 주소는 가장 최근에 실행된 LDREX 명령어에 사용된 주소와 같아야 합니다. 다른 주소에 대해 STREX 명령어를 실행하면 예상할 수 없는 결과가 발생합니다.

아키텍처

ARM LDREX 및 STREX는 ARMv6 이상에서 사용할 수 있습니다.

ARM LDREXB, LDREXH, LDREXD, STREXB, STREXD 및 STREXH는 ARMv6K 이상에서 사용할 수 있습니다.

LDREXD와 STREXD를 ARMv7-M 프로필에서 사용할 수 없다는 점을 제외하고 이러한 32비트 Thumb 명령어는 ARMv6T2 이상에서 사용할 수 있습니다.

이러한 명령어의 16비트 버전은 없습니다.

예제

```
    MOV r1, #0x1               ; load the  'lock taken' value
try
    LDREX r0, [LockAddr]       ; load the lock value
    CMP r0, #0                 ; is the lock free?
    STREXEQ r0, r1, [LockAddr] ; try and claim the lock
    CMPEQ r0, #0               ; did this succeed?
    BNE try                    ; no - try again
    ....                       ; yes - we have the lock
```

4.2.13 CLREX

단독 지우기 주소에 단독 액세스에 대한 요청이 들어 있는 실행 프로세서의 지역 레코드를 지웁니다.

구문

CLREX{*cond*}

인수 설명:

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

참고

*cond*는 위의 IT 명령어를 통해 Thumb-2 코드에서만 사용할 수 있습니다. 이는 ARM의 무조건 명령어입니다.

사용법

CLREX 명령어를 사용하여 밀접하게 결합된 배타적 액세스 모니터를 공개 액세스 상태로 되돌립니다. 이렇게 하면 메모리를 위한 더미 저장소가 필요 없게 됩니다. 동기화 기본 형식 지원에 대한 자세한 내용은 *ARM 아키텍처 참조 문서*를 참조하십시오.

CLREX가 주소에 단독 액세스에 대한 요청이 들어 있는 실행 프로세서의 전역 레코드도 지우는지 여부는 구현 시 정의됩니다.

아키텍처

이 ARM 명령어는 ARMv6K 이상에서 사용할 수 있습니다.

이 32비트 Thumb-2 명령어는 ARMv7 이상에서 사용할 수 있습니다.

16비트 Thumb CLREX 명령어는 없습니다.

4.2.14 SWP 및 SWPB

레지스터와 메모리 간 데이터 스왑

구문

SWP{B}{*cond*} *Rt*, *Rt2*, [*Rn*]

인수 설명:

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

B 선택적 접미사입니다. B가 있으면 바이트가 스왑되고, 그렇지 않으면 32비트 워드가 스왑됩니다.

Rt 대상 레지스터입니다.

Rt2 소스 레지스터입니다. *Rt2*는 *Rt*와 동일한 레지스터일 수 있습니다.

Rn 메모리에 주소를 포함합니다. *Rn*은 *Rt* 및 *Rt2* 둘 다와 다른 레지스터여야 합니다.

사용법

SWP 및 SWPB를 사용하여 세마포를 구현할 수 있습니다.

- 메모리에 있는 데이터가 *Rt*로 로드됩니다.
- *Rt2*의 내용이 메모리에 저장됩니다.
- *Rt2*가 *Rt*와 동일한 레지스터이면 레지스터의 내용과 메모리 위치의 내용이 스왑됩니다.

참고

ARMv6 이상에서는 SWP 및 SWPB를 향후 사용할 수 없습니다. ARMv6 이상에서 더 복잡한 세마포를 구현하는 명령어에 대해서는 4-36페이지의 *LDREX 및 STREX*를 참조하십시오.

아키텍처

이러한 ARM 명령어는 모든 버전의 ARM 아키텍처에서 사용할 수 있습니다.

Thumb SWP 또는 SWPB 명령어는 없습니다.

4.3 일반 데이터 처리 명령어

이 단원에는 다음 소단원이 포함되어 있습니다.

- 4-42페이지의 *유연한 두 번째 피연산자*
- 4-45페이지의 *ADD, SUB, RSB, ADC, SBC 및 RSC*
  각각 carry를 포함하거나 포함하지 않은 더하기, 빼기 및 역방향 빼기
- 4-49페이지의 *SUBS pc, lr*
  스택을 팝하지 않고 예외에서 복귀
- 4-51페이지의 *AND, ORR, EOR, BIC 및 ORN*
  논리 AND, OR, 배타적 OR, OR NOT 및 비트 지우기
- 4-54페이지의 *CLZ*
  선행 0 수 계산
- 4-55페이지의 *CMP 및 CMN*
  비교 및 음수 비교
- 4-57페이지의 *MOV 및 MVN*
  이동 및 이동하지 않음
- 4-60페이지의 *MOVT*
  맨 위로 이동, 광역
- 4-61페이지의 *TST 및 TEQ*
  테스트 및 동등 테스트
- 4-63페이지의 *SEL*
  APSR GE 플래그의 상태에 따라 각 피연산자에서 바이트 선택
- 4-65페이지의 *REV, REV16, REVSH 및 RBIT*
  바이트 또는 비트 반전
- 4-67페이지의 *ASR, LSL, LSR, ROR 및 RRX*
  오른쪽으로 산술 시프트
- 4-70페이지의 *SDIV 및 UDIV*
  부호 있는 나누기 및 부호 없는 나누기

4.3.1 유연한 두 번째 피연산자

대부분의 ARM 및 Thumb-2 일반 데이터 처리 명령어에는 유연한 두 번째 피연산자가 있습니다. 이 피연산자는 각 명령어의 구문에 대한 설명에 *Operand2* 로 표시됩니다. ARM 명령어와 Thumb-2 명령어의 *Operand2* 에 대해 허용되는 옵션에는 몇 가지 차이점이 있습니다.

구문

*Operand2*에는 다음 두 가지 형식을 사용할 수 있습니다.

#constant

Rm{, *shift*}

인수 설명:

constant 숫자 상수로 평가되는 식입니다. ARM과 Thumb-2에서 사용할 수 있는 상수의 범위는 동일하지 않습니다. 자세한 내용은 4-43페이지의 *Operand2의 상수*를 참조하십시오.

Rm 두 번째 피연산자에 대한 데이터가 들어 있는 ARM 레지스터입니다. 레지스터 내의 비트 패턴은 다양한 방법으로 시프트하거나 회전할 수 있습니다.

shift *Rm*에 적용할 선택적 시프트. 다음 중 하나일 수 있습니다.

- ASR *#n* *n*비트만큼 오른쪽으로 산술 시프트. $1 \le n \le 32$
- LSL *#n* *n*비트만큼 왼쪽으로 논리 시프트. $0 \le n \le 31$
- LSR *#n* *n*비트만큼 오른쪽으로 논리 시프트. $1 \le n \le 32$
- ROR *#n* *n*비트만큼 오른쪽으로 회전. $1 \le n \le 31$
- RRX 확장 포함 1비트만큼 오른쪽으로 회전합니다.
- *type Rs* ARM에서만 사용할 수 있습니다. 다음은 각 요소에 대한 설명입니다.
  - *type* ASR, LSL, LSR, ROR 중 하나입니다.
  - *Rs* 시프트 양을 제공하는 ARM 레지스터입니다. 최하위 바이트만 사용됩니다.

참고

시프트 연산의 결과는 명령어의 *Operand2*로 사용되지만 *Rm* 자체는 변경되지 않습니다.

Operand2의 상수

ARM 명령어에서는 32비트 워드 내에서 임의의 짝수 비트 수만큼 8비트 값을 오른쪽으로 회전하여 얻을 수 있는 모든 값을 *constant*에 지정할 수 있습니다.

32비트 Thumb-2 명령어에서 *constant*는 다음 중 하나일 수 있습니다.

- 32비트 워드 내에서 임의의 비트 수만큼 8비트 값을 왼쪽으로 시프트하여 얻을 수 있는 모든 상수
- `0x00XY00XY` 형식의 모든 상수
- `0xXY00XY00` 형식의 모든 상수
- `0xXYXYXYXY` 형식의 모든 상수

이 외에도 일부 명령어에서는 *constant*에 더 넓은 범위의 값을 지정할 수 있습니다. 이러한 상수에 대한 자세한 내용은 각 명령어에 대한 설명을 참조하십시오.

2, 4 또는 6비트만큼 8비트 값을 회전하여 얻은 상수는 ARM 데이터 처리 명령어에서만 사용할 수 있고 Thumb-2에서는 사용할 수 없습니다. 다른 모든 ARM 상수는 Thumb-2에서도 사용할 수 있습니다.

ASR

*n*비트만큼 오른쪽으로 산술 시프트는 내용이 2의 보수 부호 있는 정수로 간주될 경우 *Rm*에 들어 있는 값을 2^n으로 나눕니다. 원래 비트[31]은 레지스터의 *n*비트 왼쪽에 복사됩니다.

LSR 및 LSL

*n*비트만큼 오른쪽으로 논리 시프트는 내용이 부호 없는 정수로 간주될 경우 *Rm*에 들어 있는 값을 2^n으로 나눕니다. 레지스터의 왼쪽 *n*비트는 0으로 설정됩니다.

*n*비트만큼 왼쪽으로 논리 시프트는 내용이 부호 없는 정수로 간주될 경우 *Rm*에 들어 있는 값을 2^n으로 곱합니다. 이 경우 경고 없이 오버플로가 발생할 수 있습니다. 레지스터의 오른쪽 *n*비트는 0으로 설정됩니다.

ROR

*n*비트만큼 오른쪽으로 회전은 레지스터의 오른쪽 n비트를 결과의 왼쪽 *n*비트로 이동합니다. 이때 다른 모든 비트도 *n*비트만큼 오른쪽으로 이동합니다 (4-44페이지의 그림 4-1 참조).

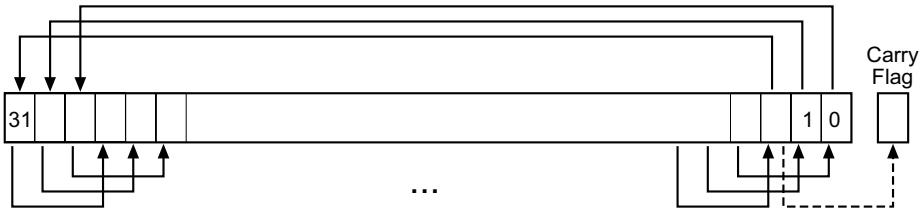

그림 4-1 ROR

RRX

확장 포함 오른쪽으로 회전은 *Rm*의 내용을 1비트만큼 오른쪽으로 회전합니다. carry 플래그는 *Rm*의 비트[31]로 복사됩니다 (그림 4-2 참조).

S 접미사가 지정된 경우 *Rm*의 이전 비트[0] 값이 carry 플래그로 시프트됩니다 (*carry 플래그* 참조).

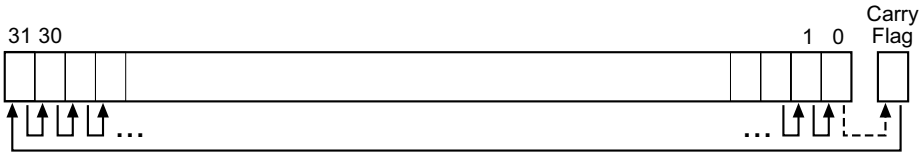

그림 4-2 RRX

carry 플래그

명령어가 다음 중 하나일 경우 carry 플래그는 *Rm*에서 시프트된 마지막 비트로 업데이트됩니다.

- S 접미사를 사용하는 경우, MOV, MVN, AND, ORR, ORN, EOR 또는 BIC
- S 접미사가 필요 없는 경우, TEQ 또는 TST

명령어 대체

일부 명령어 쌍 (ADD와 SUB, ADC와 SBC, AND와 BIC, MOV와 MVN, CMP와 CMN) 은 *constant*의 부정이나 논리 반전을 제외하고 서로 동일합니다.

constant 값을 사용할 수 없지만 해당 논리 반전이나 부정을 사용할 수 있는 경우 어셈블러는 둘 중 한 명령어를 대체하고 *constant*를 반전시키거나 부정합니다.

디스어셈블리 목록을 소스 코드와 비교할 때는 이러한 사항에 주의해야 합니다.

--diag_warning 1645 어셈블러 명령 행 옵션을 사용하여 명령어 대체가 발생하는 시기를 확인할 수 있습니다.

4.3.2 ADD, SUB, RSB, ADC, SBC 및 RSC

각각 carry를 포함하거나 포함하지 않은 더하기, 빼기 및 역방향 빼기

4-98페이지의 *병렬 더하기 및 빼기*도 참조하십시오.

구문

```
op{S}{cond} {Rd}, Rn, Operand2
op{cond} {Rd}, Rn, #imm12              ; Thumb-2 ADD and SUB only
```

인수 설명:

op 다음 중 하나입니다.

- ADD 더하기
- ADC carry 포함 더하기
- SUB 빼기
- RSB 역방향 빼기
- SBC carry 포함 빼기
- RSC carry 포함 역방향 빼기 (ARM에만 해당)

S 선택적 접미사입니다. S를 지정하면 연산 결과의 조건 코드 플래그가 업데이트됩니다 (2-20페이지의 *조건부 실행* 참조).

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

Rd 대상 레지스터입니다.

Rn 첫 번째 피연산자가 들어 있는 레지스터입니다.

Operand2 유연한 두 번째 피연산자입니다. 옵션에 대한 자세한 내용은 4-42페이지의 *유연한 두 번째 피연산자*를 참조하십시오.

imm12 0 ~ 4095 범위에 있는 값입니다.

사용법

ADD 명령어는 *Rn*의 값과 *Operand2*의 값을 더합니다.

SUB 명령어는 *Rn*의 값에서 *Operand2*의 값을 뺍니다.

RSB (역방향 빼기) 명령어는 *Operand2*의 값에서 *Rn*의 값을 뺍니다. 이 명령어는 *Operand2*의 다양한 옵션 때문에 유용합니다.

ADC, SBC 및 RSC를 사용하여 복수 워드 산술을 통합할 수 있습니다 (4-48페이지의 *복수 워드 산술 예제* 참조).

ADC (carry 포함 더하기) 명령어는 carry 플래그와 함께 *Rn*의 값과 *Operand2*의 값을 더합니다.

SBC (carry 포함 빼기) 명령어는 *Rn*의 값에서 *Operand2*의 값을 뺍니다. carry 플래그가 지워지면 결과는 1씩 감소합니다.

RSC (carry 포함 역방향 빼기) 명령어는 *Operand2*의 값에서 *Rn*의 값을 뺍니다. carry 플래그가 지워지면 결과는 1씩 감소합니다.

경우에 따라 어셈블러가 한 명령어를 다른 명령어로 대체할 수 있습니다. 디스어셈블리 목록을 읽을 때는 이러한 사항에 주의해야 합니다. 자세한 내용은 4-44페이지의 *명령어 대체*를 참조하십시오.

Thumb-2 명령어에서 pc 사용

이러한 명령어에서는 대부분의 경우 *Rd* 또는 피연산자에 pc (r15) 를 사용할 수 없습니다.

예외적으로 0 ~ 4095 범위에 있는 상수 *Operand2* 값을 사용하고 S 접미사가 없는 ADD 및 SUB 명령어에서는 *Rn*에 pc를 사용할 수 있습니다. 이러한 명령어는 pc 상대 주소를 생성하는 데 유용합니다. 이 경우 pc 값의 비트[1]이 0으로 읽히므로 계산할 기본 주소는 항상 워드로 정렬됩니다.

4-49페이지의 *SUBS pc, lr*도 참조하십시오.

4-23페이지의 *ADR*도 참조하십시오.

ARM 명령어에서 pc 사용

pc (r15) 를 *Rn*으로 사용하면 명령어 주소에 8을 더한 값이 사용됩니다.

pc를 *Rd*로 사용할 경우 다음 사항이 적용됩니다.

- 실행이 결과에 해당하는 주소로 분기됩니다.
- S 접미사를 사용하면 현재 모드의 SPSR이 CPSR로 복사됩니다. 이 접미사를 사용하여 예외에서 복귀할 수 있습니다 (*개발자 설명서*의 6장 *프로세서 예외 처리* 참조).

4-23페이지의 *ADR*도 참조하십시오.

주의

사용자 모드 또는 시스템 모드에서 pc를 *Rd*로 사용할 경우 S 접미사를 사용하면 안 됩니다. 이러한 명령어를 실행하면 예상할 수 없는 결과가 발생하지만 어셈블러에서 어셈블리 타임에 경고를 표시할 수 없습니다.

레지스터에 의해 제어된 시프트가 있는 데이터 처리 명령어에서는 *Rd* 또는 피연산자에 pc를 사용할 수 없습니다 (4-42페이지의 *유연한 두 번째 피연산자* 참조).

조건 플래그

S를 지정하면 이러한 명령어가 결과에 따라 N, Z, C 및 V 플래그를 업데이트합니다.

16비트 명령어

이러한 명령어의 다음 형식은 Thumb-2 이전 Thumb 코드에서 사용할 수 있으며, Thumb-2 코드에서 사용될 경우 16비트 명령어입니다.

`ADDS Rd, Rn, #imm` *imm* 범위 0 ~ 7. *Rd* 및 *Rn*은 모두 Lo 레지스터여야 합니다.

`ADDS Rd, Rn, Rm` *Rd*, *Rn* 및 *Rm*은 모두 Lo 레지스터여야 합니다.

`ADD Rd, Rd, Rm` ARMv6 이하의 경우 *Rd* 또는 *Rm*이나 둘 다가 Hi 레지스터여야 합니다. ARMv6T2 이상의 경우 이 제한이 적용되지 않습니다.

`ADDS Rd, Rd, #imm` *imm* 범위 0 ~ 255. *Rd*는 Lo 레지스터여야 합니다.

`ADCS Rd, Rd, Rm` *Rd*, *Rn* 및 *Rm*은 모두 Lo 레지스터여야 합니다.

`ADD SP, SP, #imm` *imm* 범위 0 ~ 508 (워드로 정렬됨)

`ADD Rd, SP, #imm` *imm* 범위 0 ~ 1020 (워드로 정렬됨) *Rd*가 Lo 레지스터여야 합니다.

`ADD Rd, pc, #imm` *imm* 범위 0 ~ 1020 (워드로 정렬됨) *Rd*가 Lo 레지스터여야 합니다. 이 명령어에서 pc의 비트[1:0]은 0으로 읽습니다.

`SUBS Rd, Rn, Rm` *Rd*, *Rn* 및 *Rm*은 모두 Lo 레지스터여야 합니다.

`SUBS Rd, Rn, #imm` *imm* 범위 0 ~ 7. *Rd* 및 *Rn*은 모두 Lo 레지스터여야 합니다.

`SUBS Rd, Rd, #imm` *imm* 범위 0 ~ 255. *Rd*는 Lo 레지스터여야 합니다.

`SBCS Rd, Rd, Rm` *Rd*, *Rn* 및 *Rm*은 모두 Lo 레지스터여야 합니다.

`SUB SP, SP, #imm` *imm* 범위 0 ~ 508 (워드로 정렬됨)

`RSBS Rd, Rn, #0` *Rd* 및 *Rn*은 모두 Lo 레지스터여야 합니다.

예제

```
ADD     r2, r1, r3
SUBS    r8, r6, #240        ; sets the flags on the result
RSB     r4, r4, #1280       ; subtracts contents of r4 from 1280
ADCHI   r11, r0, r3         ; only executed if C flag set and Z
                            ; flag clear
RSCSLE  r0,r5,r0,LSL r4     ; conditional, flags set
```

올바르지 않은 예제

```
RSCSLE  r0,pc,r0,LSL r4    ; pc not permitted with register
                            ; controlled shift
```

복수 워드 산술 예제

이러한 두 명령어는 r2 및 r3에 들어 있는 64비트 정수를 r0 및 r1에 포함된 다른 64비트 정수에 더하고 결과를 r4 및 r5에 배치합니다.

```
ADDS    r4, r0, r2    ; adding the least significant words
ADC     r5, r1, r3    ; adding the most significant words
```

이러한 명령어는 다른 정수에서 96비트 정수를 뺍니다.

```
SUBS    r3, r6, r9
SBCS    r4, r7, r10
SBC     r5, r8, r11
```

다시 말하면 위 예제에서는 복수 워드 값에 연속 레지스터를 사용합니다. 이 작업을 수행할 필요는 없습니다. 예를 들어 다음 예제는 완전히 유효합니다.

```
SUBS    r6, r6, r9
SBCS    r9, r2, r1
SBC     r2, r8, r11
```

4.3.3 SUBS pc, lr

스택을 팝하지 않고 예외 반환

참고

이 명령어는 Thumb-2에서 특별한 경우의 명령어입니다. ARM 코드에서 동일한 명령어를 4-45페이지의 *ADD, SUB, RSB, ADC, SBC 및 RSC*에서 설명하는 SUB 명령어의 일반 형식으로 사용할 수 있습니다.

구문

```
SUBS{cond} pc, lr, #imm
```

인수 설명:

imm 즉치 상수입니다. Thumb-2 코드에서는 범위 0 ~ 255로 제한되고 ARM 코드에서는 유연한 두 번째 피연산자입니다. 자세한 내용은 4-42페이지의 *유연한 두 번째 피연산자*를 참조하십시오.

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

사용법

스택에 반환 상태가 없는 경우 SUBS pc, lr을 사용하여 예외에서 복귀할 수 있습니다.

SUBS pc, lr은 링크 레지스터에서 값을 빼고 pc에 결과를 로드한 후 SPSR을 CPSR로 복사합니다.

메모

SUBS pc, lr 은 pc에 주소를 기록합니다. 이 주소는 다음과 같이 예외 반환 후 사용 중인 명령어 세트를 기준으로 올바로 정렬되어야 합니다.

- ARM으로 복귀할 경우 pc에 기록된 주소가 워드로 정렬되어야 합니다.
- Thumb-2로 복귀할 경우 pc에 기록된 주소가 하프워드로 정렬되어야 합니다.
- Jazelle로 복귀할 경우 pc에 기록된 주소의 정렬에 대한 제한이 없습니다.

이러한 규칙을 위반하면 예상할 수 없는 결과가 발생합니다. 그러나 유효한 예외 엔트리 메커니즘 후에 이 명령어를 사용하여 복귀한 경우 소프트웨어에서 특별한 예방 조치를 취하지 않아도 됩니다.

Thumb-2에서 `MOVS pc, lr` 기호는 `SUBS pc, lr, #0`의 동의어입니다.

아키텍처

이 ARM 명령어는 모든 버전의 ARM 아키텍처에서 사용할 수 있습니다.

이 32비트 Thumb 명령어는 ARMv7-M 프로필을 제외하고 ARMv6T2 이상에서 사용할 수 있습니다.

이 명령어의 16비트 Thumb 버전은 없습니다.

4.3.4 AND, ORR, EOR, BIC 및 ORN

논리 AND, OR, 배타적 OR, 비트 지우기 및 OR NOT

구문

op{S}{*cond*} *Rd*, *Rn*, *Operand2*

인수 설명:

op 다음 중 하나입니다.

- AND 논리 AND
- ORR 논리 OR
- EOR 논리 배타적 OR
- BIC 논리 AND NOT
- ORN 논리 OR NOT (Thumb-2에만 해당)

S 선택적 접미사입니다. S를 지정하면 연산 결과의 조건 코드 플래그가 업데이트됩니다 (2-20페이지의 *조건부 실행* 참조).

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

Rd 대상 레지스터입니다.

Rn 첫 번째 피연산자가 들어 있는 레지스터입니다.

Operand2 유연한 두 번째 피연산자입니다. 옵션에 대한 자세한 내용은 4-42페이지의 *유연한 두 번째 피연산자*를 참조하십시오.

사용법

AND, EOR 및 ORR 명령어는 *Rn* 및 *Operand2*의 값에 대해 비트 AND, 배타적 OR 및 OR 연산을 수행합니다.

BIC (비트 지우기) 명령어는 *Operand2* 값의 해당 비트 보수가 포함된 *Rn*의 비트에 대해 AND 연산을 수행합니다.

ORN Thumb-2 명령어는 *Operand2* 값의 해당 비트 보수가 포함된 *Rn*의 비트에 대해 OR 연산을 수행합니다.

경우에 따라 어셈블러가 BIC를 AND로, AND를 BIC로, ORN을 ORR로 또는 ORR을 ORN으로 대체할 수 있습니다. 디스어셈블리 목록을 읽을 때는 이러한 사항에 주의해야 합니다. 자세한 내용은 4-44페이지의 *명령어 대체*를 참조하십시오.

Thumb-2 명령어에서 pc 사용

이러한 명령어의 *Rd* 또는 피연산자에 pc (r15) 를 사용할 수 없습니다.

ARM 명령어에서 pc 사용

참고

이러한 ARM 명령어에서 pc를 향후 사용할 수 없습니다.

pc를 *Rn*으로 사용하면 명령어 주소에 8을 더한 값이 사용됩니다.

pc를 *Rd*로 사용할 경우 다음 사항이 적용됩니다.

- 실행이 결과에 해당하는 주소로 분기됩니다.
- S 접미사를 사용하면 현재 모드의 SPSR이 CPSR로 복사됩니다. 이 접미사를 사용하여 예외에서 복귀할 수 있습니다 (개발자 설명서의 6장 *프로세서 예외 처리* 참조).

주의

사용자 모드 또는 시스템 모드에서 pc를 *Rd*로 사용할 경우 S 접미사를 사용하면 안 됩니다. 이러한 명령어를 실행하면 예상할 수 없는 결과가 발생하지만 어셈블러에서 어셈블리 타임에 경고를 표시할 수 없습니다.

레지스터에 의해 제어된 시프트가 있는 데이터 처리 명령어에서는 피연산자에 pc를 사용할 수 없습니다 (4-42페이지의 *유연한 두 번째 피연산자* 참조).

조건 플래그

S를 지정하면 이러한 명령어는 다음을 수행합니다.

- 결과에 따라 N 및 Z 플래그를 업데이트합니다.
- *Operand2*를 계산하는 동안 C 플래그를 업데이트할 수 있습니다 (4-42페이지의 *유연한 두 번째 피연산자* 참조).
- V 플래그를 변경하지 않습니다.

16비트 명령어

이러한 명령어의 다음 형식은 Thumb-2 이전 Thumb 코드에서 사용할 수 있으며, Thumb-2 코드에서 사용될 경우 16비트 명령어입니다.

`ANDS` *Rd*, *Rd*, *Rm* *Rd* 및 *Rm*은 모두 Lo 레지스터여야 합니다.

`EORS` *Rd*, *Rd*, *Rm* *Rd* 및 *Rm*은 모두 Lo 레지스터여야 합니다.

`ORRS` *Rd*, *Rd*, *Rm* *Rd* 및 *Rm*은 모두 Lo 레지스터여야 합니다.

`BICS` *Rd*, *Rd*, *Rm* *Rd* 및 *Rm*은 모두 Lo 레지스터여야 합니다.

처음 세 경우에서는 *OP*`S` *Rd*, *Rm*, *Rd*를 지정하는지 여부에 관계없이 명령어가 동일합니다.

예제

```
AND     r9,r2,#0xFF00
ORREQ   r2,r0,r5
EORS    r0,r0,r3,ROR r6
ANDS    r9, r8, #0x19
EORS    r7, r11, #0x18181818
BIC     r0, r1, #0xab
ORN     r7, r11, r14, ROR #4
ORNS    r7, r11, r14, ASR #32
```

올바르지 않은 예제

```
EORS    r0,pc,r3,ROR r6     ; pc not permitted with register
                            ; controlled shift
```

4.3.5 CLZ

선행 0 수 계산

구문

CLZ{*cond*} *Rd*, *Rm*

인수 설명:

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

Rd 대상 레지스터입니다. *Rd*는 r15이면 안 됩니다.

Rm 피연산자 레지스터입니다. *Rm*은 r15이면 안 됩니다.

사용법

CLZ 명령어는 *Rm*의 값에서 앞에 오는 0의 수를 계산하여 결과를 *Rd*에 반환합니다. 소스 레지스터에 대해 비트가 설정되어 있지 않으면 결과 값은 32이고, 비트 31이 설정되어 있으면 결과 값은 0입니다.

조건 플래그

이 명령어는 플래그를 변경하지 않습니다.

아키텍처

이 ARM 명령어는 ARMv5 이상에서 사용할 수 있습니다.

이 32비트 Thumb 명령어는 ARMv6T2 이상에서 사용할 수 있습니다.

이 명령어의 16비트 Thumb 버전은 없습니다.

예제

```
CLZ     r4,r9
CLZNE   r2,r3
```

결과 *Rd* 값에 의해 왼쪽으로 시프트된 *Rm*이 뒤에 오는 CLZ Thumb-2 명령어를 사용하여 레지스터 *Rm* 값을 표준화합니다. *Rm*이 0인 경우 MOV 대신 MOVS를 사용하여 플래그를 설정합니다.

```
CLZ r5, r9
MOVS r9, r9, LSL r5
```

4.3.6 CMP 및 CMN

비교 및 음수 비교

구문

CMP{*cond*} *Rn*, *Operand2*

CMN{*cond*} *Rn*, *Operand2*

인수 설명:

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

Rn 첫 번째 피연산자가 들어 있는 ARM 레지스터입니다.

Operand2 유연한 두 번째 피연산자입니다. 옵션에 대한 자세한 내용은 4-42페이지의 *유연한 두 번째 피연산자*를 참조하십시오.

사용법

이러한 명령어는 레지스터 값을 *Operand2*와 비교하며 결과에서 조건 플래그를 업데이트하지만 레지스터에 결과를 배치하지 않습니다.

CMP 명령어는 *Rn*의 값에서 *Operand2*의 값을 뺍니다. 이 명령어는 결과가 버려진다는 점을 제외하고 SUBS 명령어와 동일합니다.

CMN 명령어는 *Rn*의 값에 *Operand2*의 값을 더합니다. 이 명령어는 결과가 버려진다는 점을 제외하고 ADDS 명령어와 동일합니다.

경우에 따라 어셈블러에서 CMN을 CMP로 또는 CMP를 CMN으로 대체할 수 있습니다. 디스어셈블리 목록을 읽을 때는 이러한 사항에 주의해야 합니다. 자세한 내용은 4-44페이지의 *명령어 대체*를 참조하십시오.

ARM 명령어에서 pc 사용

참고

이러한 ARM 명령어에서 pc (r15) 를 향후 사용할 수 없습니다.

pc를 *Rn*으로 사용하면 명령어 주소에 8을 더한 값이 사용됩니다.

레지스터에 의해 제어된 시프트가 있는 데이터 처리 명령어에서는 피연산자에 pc를 사용할 수 없습니다 (4-42페이지의 *유연한 두 번째 피연산자* 참조).

Thumb-2 명령어에서 pc 사용

이러한 명령어의 피연산자에 pc (r15) 를 사용할 수 없습니다.

조건 플래그

이러한 명령어는 결과에 따라 N, Z, C 및 V 플래그를 업데이트합니다.

16비트 명령어

이러한 명령어의 다음 형식은 Thumb-2 이전 Thumb 코드에서 사용할 수 있으며, Thumb-2 코드에서 사용될 경우 16비트 명령어입니다.

`CMP Rn, Rm`

`CMN Rn, Rm` *Rn* 및 *Rm*은 모두 Lo 레지스터여야 합니다.

`CMP Rn, #imm` *Rn*이 Lo 레지스터여야 합니다. *imm* 범위 0 ~ 255

예제

```
CMP     r2, r9
CMN     r0, #6400
CMPGT   r13, r7, LSL #2
```

올바르지 않은 예제

```
CMP     r2, pc, ASR r0 ; pc not permitted with register controlled shift
```

4.3.7 MOV 및 MVN

이동 및 이동하지 않음

구문

`MOV{S}{cond} Rd, Operand2`

`MOV{cond} Rd, #imm16`

`MVN{S}{cond} Rd, Operand2`

인수 설명:

`S` 선택적 접미사입니다. S를 지정하면 연산 결과의 조건 코드 플래그가 업데이트됩니다 (2-20페이지의 *조건부 실행* 참조).

`cond` 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

`Rd` 대상 레지스터입니다.

`Operand2` 유연한 두 번째 피연산자입니다. 옵션에 대한 자세한 내용은 4-42페이지의 *유연한 두 번째 피연산자*를 참조하십시오.

`imm16` 0 ~ 65535 범위에 있는 값입니다.

사용법

`MOV` 명령어는 *`Operand2`*의 값을 *`Rd`*로 복사합니다.

`MVN` 명령어는 *`Operand2`* 값을 가지고 값에 대해 비트 단위 논리 NOT 연산을 수행한 후 결과를 *`Rd`*에 배치합니다.

경우에 따라 어셈블러에서 `MVN`을 `MOV`로 또는 `MOV`를 `MVN`으로 대체할 수 있습니다. 디스어셈블리 목록을 읽을 때는 이러한 사항에 주의해야 합니다. 자세한 내용은 4-44페이지의 *명령어 대체*를 참조하십시오.

Thumb-2 MOV 및 MVN에서 pc 사용

Thumb-2 `MOV` 또는 `MVN` 명령어에서는 *`Rd`* 또는 *`Operand2`*에 pc (r15) 를 사용할 수 없습니다.

ARM MOV 및 MVN에서 pc 사용

참고

MOV Rd,Rm 구문은 Rd 또는 Rn = pc와 함께 사용할 수 있지만 둘 모두와 함께 사용할 수는 없습니다. 다른 모든 경우는 향후 제공되지 않을 예정입니다.

---

pc를 *Rd*로 사용하면 명령어 주소에 8을 더한 값이 사용됩니다.

pc를 *Rd*로 사용할 경우 다음 사항이 적용됩니다.

- 실행이 결과에 해당하는 주소로 분기됩니다.
- S 접미사를 사용하면 현재 모드의 SPSR이 CPSR로 복사됩니다. 이 접미사를 사용하여 예외에서 복귀할 수 있습니다 (개발자 설명서의 6장 *프로세서 예외 처리* 참조).

주의

사용자 모드 또는 시스템 모드에서 pc를 *Rd*로 사용할 경우 S 접미사를 사용하면 안 됩니다. 이러한 명령어를 실행하면 예상할 수 없는 결과가 발생하지만 어셈블러에서 어셈블리 타임에 경고를 표시할 수 없습니다.

레지스터에 의해 제어된 시프트가 있는 데이터 처리 명령어에서는 *Rd* 또는 피연산자에 pc를 사용할 수 없습니다 (4-42페이지의 *유연한 두 번째 피연산자* 참조).

조건 플래그

S를 지정하면 이러한 명령어는 다음을 수행합니다.

- 결과에 따라 N 및 Z 플래그를 업데이트합니다.
- *Operand2*를 계산하는 동안 C 플래그를 업데이트할 수 있습니다 (4-42페이지의 *유연한 두 번째 피연산자* 참조).
- V 플래그를 변경하지 않습니다.

16비트 명령어

이러한 명령어의 다음 형식은 Thumb-2 이전 Thumb 코드에서 사용할 수 있으며, Thumb-2 코드에서 사용될 경우 16비트 명령어입니다.

MOVS *Rd*, *#imm* *Rd*가 Lo 레지스터여야 합니다. *imm* 범위 0 ~ 255

`MOVS Rd, Rm` — *Rd* 및 *Rm*은 모두 Lo 레지스터여야 합니다.

`MOV Rd, Rm` — ARMv6 이하의 경우 *Rd* 또는 *Rm*이나 둘 다가 Hi 레지스터여야 합니다. ARMv6 이상에서는 이 제한이 적용되지 않습니다.

아키텍처

#imm16 형식의 ARM 명령어는 ARMv6T2 이상에서 사용할 수 있습니다. ARM 명령어의 다른 형식은 모든 버전의 ARM 아키텍처에서 사용할 수 있습니다.

이러한 32비트 Thumb 명령어는 ARMv6T2 이상에서 사용할 수 있습니다.

이러한 16비트 Thumb 명령어는 ARM 아키텍처의 모든 T 변형에서 사용할 수 있습니다.

예제

```
MVNNE   r11, #0xF000000B ; ARM only. This constant is not available in T2.
```

올바르지 않은 예제

```
MVN     pc,r3,ASR r0     ; pc not permitted with register controlled shift
```

4.3.8 MOVT

맨 위로 이동. 하위 하프워드에 영향을 주지 않고 레지스터의 상위 하프워드에 16비트 즉치값을 기록합니다.

구문

`MOVT{cond} Rd, #immed_16`

인수 설명:

cond	선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).
Rd	대상 레지스터입니다. *Rd*는 pc이면 안 됩니다.
immed_16	16비트 즉치 상수입니다.

사용법

MOVT는 *Rd*[31:16]에 *immed_16*을 기록합니다. 쓰기는 *Rd*[15:0]에 영향을 주지 않습니다.

MOV, MOVT 명령어 쌍을 사용하여 32비트 상수를 생성할 수 있습니다.

4-157페이지의 *MOV32 의사 명령어*도 참조하십시오.

조건 플래그

이 명령어는 플래그를 변경하지 않습니다.

아키텍처

이 ARM 명령어는 ARMv6T2 이상에서 사용할 수 있습니다.

이 32비트 Thumb 명령어는 ARMv6T2 이상에서 사용할 수 있습니다.

이 명령어의 16비트 Thumb 버전은 없습니다.

4.3.9 TST 및 TEQ

비트 테스트 및 동등 테스트

구문

TST{*cond*} *Rn*, *Operand2*

TEQ{*cond*} *Rn*, *Operand2*

인수 설명:

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

Rn 첫 번째 피연산자가 들어 있는 ARM 레지스터입니다.

Operand2 유연한 두 번째 피연산자입니다. 옵션에 대한 자세한 내용은 4-42페이지의 *유연한 두 번째 피연산자*를 참조하십시오.

사용법

이러한 명령어는 레지스터 값을 *Operand2*에 대해 테스트하며 결과에서 조건 플래그를 업데이트하지만 레지스터에 결과를 배치하지 않습니다.

TST 명령어는 *Rn*의 값과 *Operand2*의 값에 대해 비트 단위 AND 연산을 수행합니다. 이 명령어는 결과가 버려진다는 점을 제외하고 ANDS 명령어와 동일합니다.

TEQ 명령어는 *Rn*의 값과 *Operand2*의 값에 대해 비트 배타적 OR 연산을 수행합니다. 이 명령어는 결과가 버려진다는 점을 제외하고 EORS 명령어와 동일합니다.

TEQ 명령어를 사용하면 CMP와 마찬가지로 V 또는 C 플래그에 영향을 주지 않고 두 값이 동일한지 여부를 테스트할 수 있습니다.

TEQ는 값의 부호를 테스트하는 데도 유용합니다. 비교 후, N 플래그는 두 피연산자 부호 비트의 논리 배타적 OR이 됩니다.

pc 사용

ARM 명령어의 경우

- pc (r15) 를 *Rn*으로 사용하면 명령어 주소에 8을 더한 값이 사용됩니다.
- 레지스터에 의해 제어된 시프트가 있는 데이터 처리 명령어에서는 피연산자에 pc를 사용할 수 없습니다 (4-42페이지의 *유연한 두 번째 피연산자* 참조).

Thumb-2 명령어에서는 *Rn* 또는 *Operand2*.에 pc를 사용할 수 없습니다.

조건 플래그

이러한 명령어는 다음을 수행합니다.

- 결과에 따라 N 및 Z 플래그를 업데이트합니다.
- *Operand2*를 계산하는 동안 C 플래그를 업데이트할 수 있습니다 (4-42페이지의 *유연한 두 번째 피연산자* 참조).
- V 플래그를 변경하지 않습니다.

16비트 명령어

TST 명령어의 다음 형식은 Thumb-2 이전 Thumb 코드에서 사용할 수 있으며, Thumb-2 코드에서 사용될 경우 16비트 명령어입니다.

`TST Rn, Rm` *Rn* 및 *Rm*은 모두 Lo 레지스터여야 합니다.

예제

```
TST     r0, #0x3F8
TEQEQ   r10, r9
TSTNE   r1, r5, ASR r1
```

올바르지 않은 예제

```
TEQ     pc, r1, ROR r0      ; pc not permitted with register
                            ; controlled shift
```

4.3.10 SEL

APSR GE 플래그의 상태에 따라 각 피연산자에서 바이트 선택

구문

SEL{*cond*} {*Rd*}, *Rn*, *Rm*

인수 설명:

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

Rd 대상 레지스터입니다.

Rn 첫 번째 피연산자가 들어 있는 레지스터입니다.

Rm 두 번째 피연산자가 들어 있는 레지스터입니다.

연산

SEL 명령어는 *Rn* 또는 *Rm*에서 APSR GE 플래그에 따라 바이트를 선택합니다.

- GE[0]가 설정되면 Rd[7:0]을 Rn[7:0]에서 가져오고, 그렇지 않으면 Rm[7:0]에서 가져옵니다.
- GE[1]가 설정되면 Rd[15:8]을 Rn[15:8]에서 가져오고, 그렇지 않으면 Rm[15:8]에서 가져옵니다.
- GE[2]가 설정되면 Rd[23:16]을 Rn[23:16]에서 가져오고, 그렇지 않으면 Rm[23:16]에서 가져옵니다.
- GE[3]이 설정되면 Rd[31:24]를 Rn[31:24]에서 가져오고, 그렇지 않으면 Rm[31:24]에서 가져옵니다.

사용법

Rd, *Rn* 또는 *Rm*에 r15를 사용하면 안 됩니다.

부호 있는 병렬 명령어 중 하나 다음에 SEL 명령어를 사용합니다 (4-98페이지의 *병렬 더하기 및 빼기* 참조). 이 명령어를 사용하면 복수 바이트 또는 하프워드 데이터에서 최대값 또는 최소값을 선택할 수 있습니다.

조건 플래그

이 명령어는 플래그를 변경하지 않습니다.

아키텍처

이 ARM 명령어는 ARMv6 이상에서 사용할 수 있습니다.

이러한 32비트 Thumb 명령어는 ARMv7-M 프로필을 제외하고 ARMv6T2 이상에서 사용할 수 있습니다.

이 명령어의 16비트 Thumb 버전은 없습니다.

예제

```
SEL     r0, r4, r5
SELLT   r4, r0, r4
```

다음 명령어 시퀀스는 r4의 각 바이트를 r1 및 r2에 해당하는 바이트의 부호 없는 최소값과 같게 설정합니다.

```
USUB8    r4, r1, r2
SEL      r4, r2, r1
```

4.3.11 REV, REV16, REVSH 및 RBIT

워드 또는 하프워드 내에서 바이트 또는 비트 반전

구문

op{*cond*} *Rd*, *Rn*

인수 설명:

op 다음 중 하나입니다.

- REV 워드에서 바이트 순서를 반전시킵니다.
- REV16 각 하프워드에서 개별 바이트 순서를 반전시킵니다.
- REVSH 하위 하프워드에서 바이트 순서를 반전시키거나 32비트로 부호 확장합니다.
- RBIT 32비트 워드에서 비트 순서를 반전시킵니다.

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

Rd 대상 레지스터입니다. *Rd* 는 r15이면 안 됩니다.

Rn 피연산자가 들어 있는 레지스터입니다. *Rn*은 r15이면 안 됩니다.

사용법

이러한 명령어를 사용하여 엔디안을 변경할 수 있습니다.

REV 32비트 빅엔디안 데이터를 리틀엔디안 데이터로 또는 32비트 리틀엔디안 데이터를 빅엔디안 데이터로 변환합니다.

REV16 16비트 빅엔디안 데이터를 리틀엔디안 데이터로 또는 16비트 리틀엔디안 데이터를 빅엔디안 데이터로 변환합니다.

REVSH 다음과 같이 변환합니다.

- 16비트 부호 있는 빅엔디안 데이터를 32비트 부호 있는 리틀엔디안 데이터로 변환합니다.
- 16비트 부호 있는 리틀엔디안 데이터를 32비트 부호 있는 빅엔디안 데이터로 변환합니다.

조건 플래그

이러한 명령어는 플래그를 변경하지 않습니다.

16비트 명령어

이러한 명령어의 다음 형식은 Thumb-2 이전 Thumb 코드에서 사용할 수 있으며, Thumb-2 코드에서 사용될 경우 16비트 명령어입니다.

`REV` *`Rd`*, *`Rm`* *`Rd`* 및 *`Rm`*은 모두 Lo 레지스터여야 합니다.

`REV16` *`Rd`*, *`Rm`* *`Rd`* 및 *`Rm`*은 모두 Lo 레지스터여야 합니다.

`REVSH` *`Rd`*, *`Rm`* *`Rd`* 및 *`Rm`*은 모두 Lo 레지스터여야 합니다.

아키텍처

`RBIT`를 제외하고 이러한 ARM 명령어는 ARMv6 이상에서 사용할 수 있습니다.

ARM `RBIT` 명령어는 ARMv6T2 이상에서 사용할 수 있습니다.

이러한 32비트 Thumb 명령어는 ARMv6T2 이상에서 사용할 수 있습니다.

이러한 16비트 Thumb 명령어는 ARMv6 이상의 모든 T 변형에서 사용할 수 있습니다.

예제

```
REV     r3, r7
REV16   r0, r0
REVSH   r0, r5       ; Reverse Signed Halfword
REVHS   r3, r7       ; Reverse with Higher or Same condition
RBIT    r7, r8
```

4.3.12 ASR, LSL, LSR, ROR 및 RRX

오른쪽으로 산술 시프트, 왼쪽으로 논리 시프트, 오른쪽으로 논리 시프트, 오른쪽으로 회전 및 확장을 포함하여 오른쪽으로 회전

이러한 명령어는 시프트된 레지스터 두 번째 피연산자가 있는 MOV 명령어와 같습니다.

구문

op{S}{*cond*} *Rd*, *Rm*, *Rs*

op{S}{*cond*} *Rd*, *Rm*, *#sh*

RRX{S}{*cond*} *Rd*, *Rm*

인수 설명:

op ASR, LSL, LSR 또는 ROR 중 하나입니다.

S 선택적 접미사입니다. S를 지정하면 연산 결과의 조건 코드 플래그가 업데이트됩니다 (2-20페이지의 *조건부 실행* 참조).

Rd 대상 레지스터입니다.

Rm 첫 번째 피연산자가 들어 있는 레지스터입니다. 이 피연산자는 오른쪽으로 시프트됩니다.

Rs *Rm*의 값에 적용할 시프트 값이 들어 있는 레지스터입니다. 최하위 바이트만 사용됩니다.

sh 상수 시프트입니다. 허용되는 값의 범위는 다음과 같이 명령어에 따라 달라집니다.

ASR	1 ~ 32
LSL	0 ~ 31
LSR	1 ~ 32
ROR	1 ~ 31

사용법

ASR은 레지스터 내용을 2의 제곱으로 나눈 부호 있는 값을 제공하며 부호 비트를 왼쪽의 비어 있는 비트 위치로 복사합니다.

LSL은 레지스터를 2의 제곱으로 곱한 값을 제공하고, LSR은 레지스터를 2의 가변 제곱으로 나눈 부호 없는 값을 제공합니다. 두 명령어는 모두 비어 있는 비트 위치에 0을 삽입합니다.

ROR은 레지스터 내용을 특정한 값만큼 회전한 값을 제공합니다. 오른쪽 끝으로 회전된 비트는 왼쪽의 비어 있는 비트 위치로 삽입됩니다.

RRX는 레지스터 내용을 1비트 오른쪽으로 시프트한 값을 제공합니다. 이전 carry 플래그는 비트[31]로 시프트됩니다. S 접미사가 있으면 이전 비트[0]이 carry 플래그에 배치됩니다.

제한

*Rs*를 사용하는 ARM 명령어는 r15를 사용하면 안 됩니다. Thumb 명령어는 r15 또는 r13을 사용하면 안 됩니다.

조건 플래그

S를 지정하면 이러한 명령어는 결과에 따라 N 및 Z 플래그를 업데이트합니다.

시프트 값이 0일 경우 C 플래그는 변경되지 않습니다. 시프트 값이 0일 아닐 경우 C 플래그는 시프트된 마지막 비트로 업데이트됩니다.

16비트 명령어

이러한 명령어의 다음 형식은 Thumb-2 이전 Thumb 코드에서 사용할 수 있으며, Thumb-2 코드에서 사용될 경우 16비트 명령어입니다.

`ASRS Rd, Rm, #sh` *Rd* 및 *Rm*은 모두 Lo 레지스터여야 합니다.

`ASRS Rd, Rd, Rs` *Rd* 및 *Rs*는 모두 Lo 레지스터여야 합니다.

`LSLS Rd, Rm, #sh` *Rd* 및 *Rm*은 모두 Lo 레지스터여야 합니다.

`LSLS Rd, Rd, Rs` *Rd* 및 *Rs*는 모두 Lo 레지스터여야 합니다.

`LSRS Rd, Rm, #sh` *Rd* 및 *Rm*은 모두 Lo 레지스터여야 합니다.

`LSRS Rd, Rd, Rs` *Rd* 및 *Rs*는 모두 Lo 레지스터여야 합니다.

`RORS Rd, Rd, Rs` *Rd* 및 *Rs*는 모두 Lo 레지스터여야 합니다.

예제

```
ASR     r7, r8, r9
LSLS    r1, r2, r3
LSR     r4, r5, r6
ROR     r4, r5, r6
```

4.3.13 SDIV 및 UDIV

부호 있는 나누기 및 부호 없는 나누기

구문

`SDIV{cond} {Rd}, Rn, Rm`

`UDIV{cond} {Rd}, Rn, Rm`

인수 설명:

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

Rd 대상 레지스터입니다.

Rn 나눌 값이 들어 있는 레지스터입니다.

Rm 제수가 들어 있는 레지스터입니다.

레지스터 제한

pc 또는 sp는 Rd, Rn 또는 Rm에 사용할 수 없습니다.

아키텍처

이러한 32비트 Thumb 명령어는 ARMv7-R과 ARMv7-M에서만 사용할 수 있습니다.

ARM 또는 16비트 Thumb SDIV 및 UDIV 명령어는 없습니다.

4.4 곱하기 명령어

이 단원에는 다음 소단원이 포함되어 있습니다.

- 4-72페이지의 *MUL, MLA 및 MLS*

  곱하기, 곱하기 누산 및 곱하기 빼기 (32비트 x 32비트, 하위 32비트 결과)

- 4-74페이지의 *UMULL, UMLAL, SMULL 및 SMLAL*

  부호 없는 및 부호 있는 long 곱하기 및 곱하기 누산 (32비트 x 32비트, 64비트 결과 또는 64비트 누산이기)

- 4-76페이지의 *SMULxy 및 SMLAxy*

  부호 있는 곱하기 및 부호 있는 곱하기 누산 (16비트 x 16비트, 32비트 결과)

- 4-78페이지의 *SMULWy 및 SMLAWy*

  부호 있는 곱하기 및 부호 있는 곱하기 누산 (32비트 x 16비트, 상위 32비트 결과)

- 4-79페이지의 *SMLALxy*

  부호 있는 곱하기 누산 (16비트 x 16비트, 64비트 누산)

- 4-81페이지의 *SMUAD{X} 및 SMUSD{X}*

  결과 더하기 또는 빼기 포함 이중 16비트 부호 있는 곱하기

- 4-83페이지의 *SMMUL, SMMLA 및 SMMLS*

  곱하기, 곱하기 누산 및 곱하기 빼기 (32비트 x 32비트, 상위 32비트 결과)

- 4-85페이지의 *SMLAD 및 SMLSD*

  이중 16비트 부호 있는 곱하기, 32비트 결과의 합 또는 차이의 32비트 누산

- 4-87페이지의 *SMLALD 및 SMLSLD*

  이중 16비트 부호 있는 곱하기, 32비트 결과의 합 또는 차이의 64비트 누산

- 4-89페이지의 *UMAAL*

  long에 대한 부호 없는 곱하기 누산

- 4-90페이지의 *MIA, MIAPH 및 MIAxy*

  내부 누산으로 곱하기 (XScale 보조 프로세서 0 명령어)

4.4.1 MUL, MLA 및 MLS

결과의 최하위 32비트를 제공하여 부호 있는 또는 부호 없는 32비트 피연산자 포함 곱하기, 곱하기 누산 및 곱하기 빼기.

구문

`MUL{S}{cond} {Rd}, Rn, Rm`

`MLA{S}{cond} Rd, Rn, Rm, Ra`

`MLS{cond} Rd, Rn, Rm, Ra`

인수 설명:

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

S 선택적 접미사입니다. S를 지정하면 연산 결과의 조건 코드 플래그가 업데이트됩니다 (2-20페이지의 *조건부 실행* 참조).

Rd 대상 레지스터입니다.

Rn, Rm 곱할 값이 들어 있는 레지스터입니다.

Ra 더하거나 뺄 값이 들어 있는 레지스터입니다.

사용법

MUL 명령어는 *Rn*의 값과 *Rm*의 값을 곱하고 결과의 최하위 32비트를 *Rd*에 배치합니다.

MLA 명령어는 *Rn*의 값과 *Rm*의 값을 곱하고 *Ra*의 값을 더한 다음 결과의 최하위 32비트를 *Rd*에 배치합니다.

MLS 명령어는 *Rn*의 값과 *Rm*의 값을 곱하고 *Ra*의 값에서 결과를 뺀 다음 최종 결과의 최하위 32비트를 *Rd*에 배치합니다.

Rd, *Rn*, *Rm* 또는 *Ra*에 r15를 사용하면 안 됩니다.

조건 플래그

S를 지정하면 MUL 및 MLA 명령어는 다음을 수행합니다.

- 결과에 따라 N 및 Z 플래그를 업데이트합니다.
- ARMv4 이하에서는 C 및 V 플래그를 손상시킵니다.
- ARMv5 이상에서는 C 또는 V 플래그에 영향을 주지 않습니다.

Thumb 명령어

MUL 명령어의 다음 형식은 Thumb-2 이전 Thumb 코드에서 사용할 수 있으며, Thumb-2 코드에서 사용될 경우 16비트 명령어입니다.

MULS *Rd*, *Rn*, *Rd* *Rd* 및 *Rn*은 모두 Lo 레지스터여야 합니다.

조건 코드 플래그를 업데이트할 수 있는 다른 Thumb 곱하기 명령어는 없습니다.

아키텍처

MUL 및 MLA ARM 명령어는 모든 버전의 ARM 아키텍처에서 사용할 수 있습니다.

MLS ARM 명령어는 ARMv6T2 이상에서 사용할 수 있습니다.

이러한 32비트 Thumb 명령어는 ARMv6T2 이상에서 사용할 수 있습니다.

MULS 16비트 Thumb 명령어는 ARM 아키텍처의 모든 T 변형에서 사용할 수 있습니다.

예제

```
MUL     r10, r2, r5
MLA     r10, r2, r1, r5
MULS    r0, r2, r2
MULLT   r2, r3, r2
MLS     r4, r5, r6, r7
```

4.4.2 UMULL, UMLAL, SMULL 및 SMLAL

선택적 누산, 32비트 피연산자, 64비트 결과 및 누산기 포함 부호 있는 및 부호 없는 long 곱하기.

구문

Op{S}{*cond*} *RdLo*, *RdHi*, *Rn*, *Rm*

인수 설명:

Op UMULL, UMLAL, SMULL 또는 SMLAL 중 하나입니다.

S ARM 상태에서만 사용할 수 있는 선택적 접미사입니다. S를 지정하면 연산 결과의 조건 코드 플래그가 업데이트됩니다 (2-20페이지의 *조건부 실행* 참조).

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

RdLo, *RdHi* 대상 레지스터입니다. UMLAL 및 SMLAL에는 누산 값도 들어 있습니다. *RdLo*와 *RdHi*는 서로 다른 레지스터여야 합니다.

Rn, *Rm* 피연산자가 들어 있는 ARM 레지스터입니다.

RdHi, *RdLo*, *Rn* 또는 *Rm*에 r15를 사용하면 안 됩니다.

사용법

UMULL 명령어는 *Rn* 및 *Rm*의 값을 부호 없는 정수로 해석하며, 이러한 정수를 곱하고 결과의 최하위 32비트를 *RdLo*에 배치하고 결과의 최상위 32비트를 *RdHi*에 배치합니다.

UMLAL 명령어는 *Rn* 및 *Rm*의 값을 부호 없는 정수로 해석하며, 이러한 정수를 곱하고 64비트 결과를 *RdHi* 및 *RdLo*에 들어 있는 64비트 부호 없는 정수에 더합니다.

SMULL 명령어는 *Rn* 및 *Rm*의 값을 2의 보수 부호 있는 정수로 해석하며, 이러한 정수를 곱하고 결과의 최하위 32비트를 *RdLo*에 배치하고 결과의 최상위 32비트를 *RdHi*에 배치합니다.

SMLAL 명령어는 *Rn* 및 *Rm*의 값을 2의 보수 부호 있는 정수로 해석하며, 이러한 정수를 곱하고 64비트 결과를 *RdHi* 및 *RdLo*에 들어 있는 64비트 부호 있는 정수에 더합니다.

조건 플래그

S를 지정하면 이러한 명령어는 다음을 수행합니다.

- 결과에 따라 N 및 Z 플래그를 업데이트합니다.
- C 또는 V 플래그에 영향을 주지 않습니다.

아키텍처

이러한 ARM 명령어는 모든 버전의 ARM 아키텍처에서 사용할 수 있습니다.

이러한 32비트 Thumb 명령어는 ARMv6T2 이상에서 사용할 수 있습니다.

이러한 명령어의 16비트 Thumb 버전은 없습니다.

예제

```
UMULL       r0, r4, r5, r6
UMLALS      r4, r5, r3, r8
```

4.4.3 SMUL*xy* 및 SMLA*xy*

16비트 피연산자, 32비트 결과 및 누산기 포함 부호 있는 곱하기 및 곱하기 누산

구문

SMUL<*x*><*y*>{*cond*} {*Rd*}, *Rn*, *Rm*

SMLA<*x*><*y*>{*cond*} *Rd*, *Rn*, *Rm*, *Ra*

인수 설명:

<*x*> B 또는 T입니다. B는 *Rn*의 하위 하프 (비트[15:0]) 사용을 의미하고, T는 *Rn*의 상위 하프 (비트[31:16]) 사용을 의미합니다.

<*y*> B 또는 T입니다. B는 *Rm*의 하위 하프 (비트[15:0]) 사용을 의미하고, T는 *Rm*의 상위 하프 (비트[31:16]) 사용을 의미합니다.

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

Rd 대상 레지스터입니다.

Rn, *Rm* 곱할 값이 들어 있는 레지스터입니다.

Ra 더할 값이 들어 있는 레지스터입니다.

사용법

Rd, *Rn*, *Rm* 또는 *Ra*에 r15를 사용하면 안 됩니다.

SMULxy는 *Rn* 및 *Rm*의 선택된 하프에서 16비트 부호 있는 정수를 곱하고 32비트 결과를 *Rd*에 배치합니다.

SMLAxy는 *Rn* 및 *Rm*의 선택된 하프에서 16비트 부호 있는 정수를 곱하고 32비트 결과를 *Ra*의 32비트 값에 더한 다음 결과를 *Rd*에 배치합니다.

조건 플래그

이러한 명령어는 N, Z, C 또는 V 플래그에 영향을 주지 않습니다.

누산에 오버플로가 발생하면 SMLAxy가 Q 플래그를 설정합니다. Q 플래그의 상태를 보려면 MRS 명령어 (4-134페이지의 *MRS* 참조) 를 사용하십시오.

참고

SMLAxy는 Q 플래그를 지울 수 없습니다. Q 플래그를 지우려면 MSR 명령어 (4-136페이지의 *MSR* 참조) 를 사용하십시오.

아키텍처

이러한 ARM 명령어는 ARMv6 이상과 ARMv5의 E 변형에서 사용할 수 있습니다.

이러한 32비트 Thumb 명령어는 ARMv7-M 프로필을 제외하고 ARMv6T2 이상에서 사용할 수 있습니다.

이러한 명령어의 16비트 Thumb 버전은 없습니다.

예제

```
SMULTBEQ    r8, r7, r9
SMLABBNE    r0, r2, r1, r10
SMLABT      r0, r0, r3, r5
```

4.4.4 SMULW*y* 및 SMLAW*y*

상위 32비트 결과를 제공하여 하나의 32비트 피연산자 및 하나의 16비트 피연산자 포함 부호 있는 곱하기 광역 및 부호 있는 곱하기 누산 광역

구문

SMULW<*y*>{*cond*} {*Rd*}, *Rn*, *Rm*

SMLAW<*y*>{*cond*} *Rd*, *Rn*, *Rm*, *Ra*

인수 설명:

<*y*> B 또는 T입니다. B는 *Rm*의 하위 하프 (비트[15:0]) 사용을 의미하고, T는 *Rm*의 상위 하프 (비트[31:16]) 사용을 의미합니다.

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

Rd 대상 레지스터입니다.

Rn, Rm 곱할 값이 들어 있는 레지스터입니다.

Ra 더할 값이 들어 있는 레지스터입니다.

사용법

Rd, *Rn*, *Rm* 또는 *Ra*에 r15를 사용하면 안 됩니다.

SMULWy는 *Rm*의 선택된 하프에서 부호 있는 정수에 *Rn*의 부호 있는 정수를 곱하고 48비트 결과의 상위 32비트를 *Rd*에 배치합니다.

SMLAWy는 *Rm*의 선택된 하프에서 부호 있는 정수에 *Rn*의 부호 있는 정수를 곱하고 32비트 결과를 *Ra*의 32비트 값에 더한 다음 결과를 *Rd*에 배치합니다.

조건 플래그

이러한 명령어는 N, Z, C 또는 V 플래그에 영향을 주지 않습니다.

누산에 오버플로가 발생하면 SMLAWy가 Q 플래그를 설정합니다 (4-134페이지의 *MRS* 참조).

아키텍처

이러한 ARM 명령어는 ARMv6 이상과 ARMv5의 E 변형에서 사용할 수 있습니다.

이러한 32비트 Thumb 명령어는 ARMv7-M 프로필을 제외하고 ARMv6T2 이상에서 사용할 수 있습니다.

이러한 명령어의 16비트 Thumb 버전은 없습니다.

4.4.5 SMLAL*xy*

16비트 피연산자 및 64비트 누산기 포함 부호 있는 곱하기 누산

구문

SMLAL<*x*><*y*>{*cond*} *RdLo*, *RdHi*, *Rn*, *Rm*

인수 설명:

<*x*> B 또는 T입니다. B는 *Rn*의 하위 하프 (비트[15:0]) 사용을 의미하고, T는 *Rn*의 상위 하프 (비트[31:16]) 사용을 의미합니다.

<*y*> B 또는 T입니다. B는 *Rm*의 하위 하프 (비트[15:0]) 사용을 의미하고, T는 *Rm*의 상위 하프 (비트[31:16]) 사용을 의미합니다.

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

RdHi, RdLo 대상 레지스터입니다. 이 대상 레지스터에는 누산 값도 들어 있습니다. *RdHi*와 *RdLo*는 서로 다른 레지스터여야 합니다.

Rn, Rm 곱할 값이 들어 있는 레지스터입니다.

RdHi, *RdLo*, *Rn* 또는 *Rm*에 r15를 사용하면 안 됩니다.

사용법

SMLALxy는 *Rm*의 선택된 하프에서 부호 있는 정수에 *Rn*의 선택된 하프에서 부호 있는 정수를 곱하고 32비트 결과를 *RdHi* 및 *RdLo*의 64비트 값에 더합니다.

조건 플래그

이 명령어는 플래그를 변경하지 않습니다.

참고

SMLALxy는 예외를 발생시킬 수 없습니다. 이 명령어에서 오버플로가 발생하면 결과가 경고 없이 래핑됩니다.

아키텍처

이 ARM 명령어는 ARMv6 이상과 ARMv5의 E 변형에서 사용할 수 있습니다.

이러한 32비트 Thumb 명령어는 ARMv7-M 프로필을 제외하고 ARMv6T2 이상에서 사용할 수 있습니다.

이 명령어의 16비트 Thumb 버전은 없습니다.

예제

```
SMLALTB     r2, r3, r7, r1
SMLALBTVS   r0, r1, r9, r2
```

4.4.6 SMUAD{X} 및 SMUSD{X}

결과 더하기 또는 빼기 포함 이중 16비트 부호 있는 곱하기 및 피연산자 하프의 선택적 교환

구문

op{X}{*cond*} {*Rd*}, *Rn*, *Rm*

인수 설명:

op 다음 중 하나입니다.

- SMUAD 이중 곱하기, 결과 더하기
- SMUSD 이중 곱하기, 결과 빼기

X 선택적 매개변수입니다. X가 있으면 곱하기가 수행되기 전에 두 번째 피연산자의 최상위 및 최하위 하프워드가 교환됩니다.

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

Rd 대상 레지스터입니다.

Rn, *Rm* 피연산자가 들어 있는 레지스터입니다.

Rd, *Rn* 또는 *Rm*에 r15를 사용하면 안 됩니다.

사용법

SMUAD는 *Rn*의 하위 하프워드를 *Rm*의 하위 하프워드와 곱하고 *Rn*의 상위 하프워드를 *Rm*의 상위 하프워드와 곱합니다. 그런 다음 결과를 더하고 그 합을 *Rd*에 저장합니다.

SMUSD는 *Rn*의 하위 하프워드와 *Rm*의 하위 하프워드를 곱하고 *Rn*의 상위 하프워드를 *Rm*의 상위 하프워드와 곱합니다. 그런 다음 첫 번째 결과에서 두 번째 결과를 빼고 그 차이를 *Rd*에 저장합니다.

조건 플래그

더하기가 오버플로될 경우 SMUAD 명령어는 Q 플래그를 설정합니다.

아키텍처

이러한 ARM 명령어는 ARMv6 이상과 ARMv5의 E 변형에서 사용할 수 있습니다.

이러한 32비트 Thumb 명령어는 ARMv7-M 프로필을 제외하고 ARMv6T2 이상에서 사용할 수 있습니다.

이러한 명령어의 16비트 Thumb 버전은 없습니다.

예제

```
SMUAD       r2, r3, r2
SMUSDXNE    r0, r1, r2
```

4.4.7 SMMUL, SMMLA 및 SMMLS

부호 있는 최상위 워드 곱하기, 누산 포함 부호 있는 최상위 워드 곱하기, 빼기 포함 부호 있는 최상위 워드 곱하기. 이러한 명령어는 32비트 피연산자를 사용하고 결과의 최상위 32비트만 생성합니다.

구문

SMMUL{R}{*cond*} {*Rd*}, *Rn*, *Rm*

SMMLA{R}{*cond*} *Rd*, *Rn*, *Rm*, *Ra*

SMMLS{R}{*cond*} *Rd*, *Rn*, *Rm*, *Ra*

인수 설명:

R 선택적 매개변수입니다. R이 있으면 결과를 반올림하고, 그렇지 않으면 결과를 자릅니다.

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

Rd 대상 레지스터입니다.

Rn, Rm 피연산자가 들어 있는 레지스터입니다.

Ra 더하거나 뺄 값이 들어 있는 레지스터입니다.

Rd, *Rn*, *Rm* 또는 *Ra*에 r15를 사용하면 안 됩니다.

연산

SMMUL은 *Rn* 및 *Rm*의 값을 곱하고 64비트 결과의 최상위 32비트를 *Rd*에 저장합니다.

SMMLA는 *Rn* 및 *Rm*의 값을 곱하고 *Ra*의 값을 결과의 최상위 32비트에 더한 다음 결과를 *Rd*에 저장합니다.

SMMLS는 *Rn* 및 *Rm*의 값을 곱하고 32비트 왼쪽으로 시프트된 *Ra*의 값에서 결과를 뺀 다음 결과의 최상위 32비트를 *Rd*에 저장합니다.

선택적 R 매개변수를 지정하면 최상위 32비트를 추출하기 전에 0x80000000이 더해집니다. 이것은 결과를 반올림하는 효과가 있습니다.

조건 플래그

이러한 명령어는 플래그를 변경하지 않습니다.

아키텍처

이러한 ARM 명령어는 ARMv6 이상과 ARMv5의 E 변형에서 사용할 수 있습니다.

이러한 32비트 Thumb 명령어는 ARMv7-M 프로필을 제외하고 ARMv6T2 이상에서 사용할 수 있습니다.

이러한 명령어의 16비트 Thumb 버전은 없습니다.

예제

```
SMMULGE     r6, r4, r3
SMMULR      r2, r2, r2
```

4.4.8 SMLAD 및 SMLSD

결과 더하기 또는 빼기 및 32비트 누산 포함 이중 16비트 부호 있는 곱하기

구문

op{X}{*cond*} *Rd*, *Rn*, *Rm*, *Ra*

인수 설명:

op 다음 중 하나입니다.

- SMLAD 이중 곱하기, 결과의 합 누산
- SMLSD 이중 곱하기, 결과의 차이 누산

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

X 선택적 매개변수입니다. X가 있으면 곱하기가 수행되기 전에 두 번째 피연산자의 최상위 및 최하위 하프워드가 교환됩니다.

Rd 대상 레지스터입니다.

Rn, Rm 피연산자가 들어 있는 레지스터입니다.

Ra 누산 피연산자가 들어 있는 레지스터입니다.

Rd, *Rn*, *Rm* 또는 *Ra*에 r15를 사용하면 안 됩니다.

연산

SMLAD는 *Rn*의 하위 하프워드와 *Rm*의 하위 하프워드를 곱하고 *Rn*의 상위 하프워드와 *Rm*의 상위 하프워드를 곱합니다. 그런 다음 두 결과를 모두 *Ra*의 값에 더하고 그 합을 *Rd*에 저장합니다.

SMLSD는 *Rn*의 하위 하프워드와 *Rm*의 하위 하프워드를 곱하고 *Rn*의 상위 하프워드와 *Rm*의 상위 하프워드를 곱합니다. 그런 다음 첫 번째 결과에서 두 번째 결과를 빼고 그 차이를 *Ra*의 값에 더한 다음 결과를 *Rd*에 저장합니다.

조건 플래그

이러한 명령어는 플래그를 변경하지 않습니다.

아키텍처

이러한 ARM 명령어는 ARMv6 이상과 ARMv5의 E 변형에서 사용할 수 있습니다.

이러한 32비트 Thumb 명령어는 ARMv7-M 프로필을 제외하고 ARMv6T2 이상에서 사용할 수 있습니다.

이러한 명령어의 16비트 Thumb 버전은 없습니다.

예제

```
SMLSD       r1, r2, r0, r7
SMLSDX      r11, r10, r2, r3
SMLADLT     r1, r2, r4, r1
```

4.4.9 SMLALD 및 SMLSLD

결과 더하기 또는 빼기 및 64비트 누산 포함 이중 16비트 부호 있는 곱하기

구문

op{X}{*cond*} *RdLo*, *RdHi*, *Rn*, *Rm*

인수 설명:

op 다음 중 하나입니다.

- SMLALD 이중 곱하기, 결과의 합 누산
- SMLSLD 이중 곱하기, 결과의 차이 누산

X 선택적 매개변수입니다. X가 있으면 곱하기가 수행되기 전에 두 번째 피연산자의 최상위 및 최하위 하프워드가 교환됩니다.

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

RdLo, *RdHi* 64비트 결과의 대상 레지스터입니다. 이 대상 레지스터에는 64비트 누산 피연산자도 들어 있습니다. *RdHi*와 *RdLo*는 서로 다른 레지스터여야 합니다.

Rn, *Rm* 피연산자가 들어 있는 레지스터입니다.

RdLo, *RdHi*, *Rn* 또는 *Rm*에 r15를 사용하면 안 됩니다.

연산

SMLALD는 *Rn*의 하위 하프워드와 *Rm*의 하위 하프워드를 곱하고 *Rn*의 상위 하프워드와 *Rm*의 상위 하프워드를 곱합니다. 그런 다음 두 결과를 모두 *RdLo*, *RdHi*의 값에 더하고 그 합을 *RdLo*, *RdHi*에 저장합니다.

SMLSLD는 *Rn*의 하위 하프워드와 *Rm*의 하위 하프워드를 곱하고 *Rn*의 상위 하프워드와 *Rm*의 상위 하프워드를 곱합니다. 그런 다음 첫 번째 결과에서 두 번째 결과를 빼고 그 차이를 *RdLo*, *RdHi*의 값에 더한 다음 결과를 *RdLo*, *RdHi*에 저장합니다.

조건 플래그

이러한 명령어는 플래그를 변경하지 않습니다.

아키텍처

이러한 ARM 명령어는 ARMv6 이상과 ARMv5의 E 변형에서 사용할 수 있습니다.

이러한 32비트 Thumb 명령어는 ARMv7-M 프로필을 제외하고 ARMv6T2 이상에서 사용할 수 있습니다.

이러한 명령어의 16비트 Thumb 버전은 없습니다.

예제

```
SMLALD      r10, r11, r5, r1
SMLSLD      r3, r0, r5, r1
```

4.4.10 UMAAL

long에 대한 부호 없는 곱하기 누산

구문

UMAAL{*cond*} *RdLo*, *RdHi*, *Rn*, *Rm*

인수 설명:

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

RdLo, *RdHi* 64비트 결과의 대상 레지스터입니다. 이 대상 레지스터에는 두 개의 32비트 누산 피연산자도 들어 있습니다. *RdLo* 및 *RdHi*는 서로 다른 레지스터여야 합니다.

Rn, *Rm* 곱하기 피연산자가 들어 있는 레지스터입니다.

RdLo, *RdHi*, *Rn* 또는 *Rm*에 r15를 사용하면 안 됩니다.

연산

UMAAL 명령어는 *Rn* 및 *Rm*의 값을 곱하고 두 32비트 값을 *RdHi* 및 *RdLo*에 더한 다음 64비트 결과를 *RdLo*, *RdHi*에 저장합니다.

조건 플래그

이 명령어는 플래그를 변경하지 않습니다.

아키텍처

이 ARM 명령어는 ARMv6 이상과 ARMv5의 E 변형에서 사용할 수 있습니다.

이러한 32비트 Thumb 명령어는 ARMv7-M 프로필을 제외하고 ARMv6T2 이상에서 사용할 수 있습니다.

이 명령어의 16비트 Thumb 버전은 없습니다.

예제

```
UMAAL       r8, r9, r2, r3
UMAALGE     r2, r0, r5, r3
```

4.4.11 MIA, MIAPH 및 MIAxy

내부 누산 포함 곱하기 (32비트 x 32비트, 40비트 누산)

내부 누산 포함 곱하기, 패킹된 하프워드 (16비트 x 16비트 x 2, 40비트 누산)

내부 누산 포함 곱하기 (16비트 x 16비트, 40비트 누산)

구문

`MIA{cond} Acc, Rn, Rm`

`MIAPH{cond} Acc, Rn, Rm`

`MIA<x><y>{cond} Acc, Rn, Rm`

인수 설명:

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

Acc 내부 누산기입니다. 표준 이름은 acc*x*입니다. 여기서 *x*는 0에서 *n* 사이의 정수입니다. *n*의 값은 프로세서에 따라 달라집니다. 현재 프로세서에서는 0입니다.

Rn, Rm 곱할 값이 들어 있는 ARM 레지스터입니다.

Rn 또는 *Rm*에 r15를 사용하면 안 됩니다.

<*x*><*y*> 다음 중 하나입니다. BB, BT, TB, TT

사용법

MIA 명령어는 *Rn* 및 *Rm*의 부호 있는 정수를 곱하고 결과를 *Acc*의 40비트 값에 더합니다.

MIAPH 명령어는 *Rn* 및 *Rm*의 하위 하프에 있는 부호 있는 정수를 곱하고 *Rn* 및 *Rm*의 상위 하프에 있는 부호 있는 정수를 곱한 다음 두 개의 32비트 결과를 *Acc*의 40비트 값에 더합니다.

MIAxy 명령어는 *Rs*의 선택된 하프에서 부호 있는 정수와 *Rm*의 선택된 하프에서 부호 있는 정수를 곱한 다음 32비트 결과를 *Acc*의 40비트 값에 더합니다. <*x*> == B는 *Rn*의 하위 하프 (비트[15:0]) 사용을 의미하고, <*x*> == T는 *Rn*의 상위 하프 (비트[31:16]) 사용을 의미합니다. <*y*> == B는 *Rm*의 하위 하프 (비트[15:0]) 사용을 의미하고, <*y*> == T는 *Rm*의 상위 하프 (비트[31:16]) 사용을 의미합니다.

조건 플래그

이러한 명령어는 플래그를 변경하지 않습니다.

참고

이러한 명령어는 예외를 발생시킬 수 없습니다. 이러한 명령어에서 오버플로가 발생하면 경고 없이 결과가 겹쳐집니다.

아키텍처

이러한 ARM 보조 프로세서 0 명령어는 XScale 프로세서에서만 사용할 수 있습니다.

이러한 명령어의 Thumb 버전은 없습니다.

예제

```
MIA     acc0,r5,r0
MIALE   acc0,r1,r9
MIAPH   acc0,r0,r7
MIAPHNE acc0,r11,r10
MIABB   acc0,r8,r9
MIABT   acc0,r8,r8
MIATB   acc0,r5,r3
MIATT   acc0,r0,r6
MIABTGT acc0,r2,r5
```

4.5 포화 명령어

이 단원에는 다음 소단원이 포함되어 있습니다.

- *포화 산술*
- 4-93페이지의 *QADD, QSUB, QDADD 및 QDSUB*
- 4-95페이지의 *SSAT 및 USAT*.

일부 병렬 명령어도 포화됩니다 (4-97페이지의 *병렬 명령어* 참조).

4.5.1 포화 산술

이러한 연산은 *포화*됩니다 (SAT). 즉, 2^n의 일부 값이 명령어에 따라 달라집니다.

- 부호 있는 포화 연산의 경우 전체 결과가 -2^n 미만이면 반환 결과는 -2^n입니다.
- 부호 없는 포화 연산의 경우 전체 결과가 음수이면 반환 결과는 0입니다.
- 전체 결과가 $2^n - 1$을 초과하면 반환 결과는 $2^n - 1$입니다.

이러한 연산이 수행되는 경우를 *포화*라고 합니다. 포화가 발생하면 일부 명령어가 Q 플래그를 설정합니다.

참고

포화가 수행되지 않을 경우 포화 명령어는 Q 플래그를 지우지 않습니다. Q 플래그를 지우려면 MSR 명령어 (4-136페이지의 *MSR* 참조) 를 사용하십시오.

Q 플래그는 두 개의 다른 명령어 (4-76페이지의 *SMULxy 및 SMLAxy* 및 4-78페이지의 *SMULWy 및 SMLAWy* 참조) 로 설정할 수도 있지만 이러한 명령어는 포화되지 않습니다.

4.5.2 QADD, QSUB, QDADD 및 QDSUB

부호 있는 더하기, 빼기, 이중 및 더하기, 이중 및 빼기, 부호 있는 $–2^{31} \le x \le 2^{31}–1$ 범위로 결과 포화

4-98페이지의 *병렬 더하기 및 빼기*도 참조하십시오.

구문

op{*cond*} {*Rd*}, *Rm*, *Rn*

인수 설명:

op QADD, QSUB, QDADD 또는 QDSUB 중 하나입니다.

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

Rd 대상 레지스터입니다.

Rm, Rn 피연산자가 들어 있는 레지스터입니다.

Rd, *Rm* 또는 *Rn*에 r15를 사용하면 안 됩니다.

사용법

QADD 명령어는 *Rm*의 값과 *Rn*의 값을 더합니다.

QSUB 명령어는 *Rm*의 값에서 *Rn*의 값을 뺍니다.

QDADD 명령어는 SAT (*Rm* + SAT (*Rn* * 2)) 를 계산합니다. 이 경우 배수화 연산이나 더하기 중 하나 또는 둘 다에서 포화가 발생할 수 있습니다. 포화가 배수화에서만 발생하고 더하기에서는 발생하지 않으면 Q 플래그가 설정되지만 최종 결과는 포화되지 않습니다.

QDSUB 명령어는 SAT (*Rm* - SAT (*Rn* * 2)) 를 계산합니다. 이 경우 배수화 연산이나 빼기 중 하나 또는 둘 다에서 포화가 발생할 수 있습니다. 포화가 배수화에서만 발생하고 빼기에서는 발생하지 않으면 Q 플래그가 설정되지만 최종 결과는 포화되지 않습니다.

참고

이러한 명령어는 모든 값을 2의 보수 부호 있는 정수로 처리합니다.

ARMv6 이상에서만 사용할 수 있는 유사한 병렬 명령어에 대해서는 4-98페이지의 *병렬 더하기 및 빼기*를 참조하십시오.

조건 플래그

포화가 발생하면 이러한 명령어는 Q 플래그를 설정합니다. Q 플래그의 상태를 보려면 MRS 명령어 (4-134페이지의 *MRS* 참조) 를 사용하십시오.

아키텍처

이러한 ARM 명령어는 ARMv6 이상과 ARMv5의 E 변형에서 사용할 수 있습니다.

이러한 32비트 Thumb 명령어는 ARMv7-M 프로필을 제외하고 ARMv6T2 이상에서 사용할 수 있습니다.

이러한 명령어의 16비트 Thumb 버전은 없습니다.

예제

```
QADD    r0, r1, r9
QDSUBLT r9, r0, r1
```

4.5.3 SSAT 및 USAT

포화 전 선택적 시프트를 사용하여 임의의 비트 위치로 부호 있는 포화 및 부호 없는 포화

SSAT은 부호 있는 값을 부호 있는 범위로 포화시킵니다.

USAT은 부호 있는 값을 부호 없는 범위로 포화시킵니다.

4-103페이지의 *SSAT16 및 USAT16*도 참조하십시오.

구문

op{*cond*} *Rd*, #*sat*, *Rm*{, *shift*}

인수 설명:

op SSAT 또는 USAT입니다.

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

Rd 대상 레지스터입니다. *Rd* 는 r15이면 안 됩니다.

sat 포화시킬 비트 위치를 지정합니다. SSAT의 경우 1에서 32 사이에서 지정하고 USAT의 경우 0에서 31 사이에서 지정합니다.

Rm 피연산자가 들어 있는 레지스터입니다. *Rm* 은 r15이면 안 됩니다.

shift 선택적 시프트입니다. 다음 중 하나여야 합니다.

- ASR #*n* 여기서 *n*은 1에서 32 사이 (ARM) 또는 1에서 31 사이 (Thumb-2) 의 숫자입니다.
- LSL #*n* 여기서 *n*은 0에서 31 사이의 숫자입니다.

연산

SSAT 명령어는 지정된 시프트를 적용한 다음 부호 있는 범위 $-2^{sat-1} \leq x \leq 2^{sat-1}-1$ 로 포화시킵니다.

USAT 명령어는 지정된 시프트를 적용한 다음 부호 없는 범위 $0 \leq x \leq 2^{sat}-1$로 포화시킵니다.

조건 플래그

포화가 발생하면 이러한 명령어는 Q 플래그를 설정합니다. Q 플래그의 상태를 보려면 MRS 명령어 (4-134페이지의 *MRS* 참조) 를 사용하십시오.

아키텍처

이러한 ARM 명령어는 ARMv6 이상에서 사용할 수 있습니다.

이러한 32비트 Thumb 명령어는 ARMv6T2 이상에서 사용할 수 있습니다.

이러한 명령어의 16비트 Thumb 버전은 없습니다.

예제

```
SSAT    r7, #16, r7, LSL #4
USATNE  r0, #7, r5
```

4.6 병렬 명령어

이 단원에는 다음 소단원이 포함되어 있습니다.

- 4-98페이지의 *병렬 더하기 및 빼기*
  다양한 바이트 및 하프워드 더하기 및 빼기
- 4-101페이지의 *USAD8 및 USADA8*
  부호 없는 절대차의 합 및 부호 없는 절대차의 합 누산
- 4-103페이지의 *SSAT16 및 USAT16*
  병렬 하프워드 포화 명령어

이외에도 몇 가지 병렬 패킹 해제 명령어도 있습니다 (4-108페이지의 *SXT, SXTA, UXT 및 UXTA* 참조).

4.6.1 병렬 더하기 및 빼기

다양한 바이트 및 하프워드 더하기 및 빼기

구문

<prefix>op{cond} {Rd}, Rn, Rm

인수 설명:

<prefix> 다음 중 하나입니다.

- S 부호 있는 산술 모듈로 2^8 또는 2^{16}, APSR GE 플래그 설정
- Q 부호 있는 포화 산술
- SH 결과를 양분하는 부호 있는 산술
- U 부호 없는 산술 모듈로 2^8 또는 2^{16}, APSR GE 플래그 설정
- UQ 부호 없는 포화 산술
- UH 결과를 양분하는 부호 없는 산술

op 다음 중 하나입니다.

- ADD8 바이트 더하기
- ADD16 하프워드 더하기
- SUB8 바이트 빼기
- SUB16 하프워드 빼기
- ASX *Rm*의 하프워드를 교환한 다음 상위 하프워드를 더하고 하위 하프워드를 뺍니다.
- SAX *Rm*의 하프워드를 교환한 다음 상위 하프워드를 빼고 하위 하프워드를 더합니다.

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

Rd 대상 레지스터입니다. *Rd*에 r15를 사용하면 안 됩니다.

Rm, Rn 피연산자가 들어 있는 ARM 레지스터입니다. *Rm* 또는 *Rn*에 r15를 사용하면 안 됩니다.

연산

이러한 명령어는 피연산자의 바이트 또는 하프워드에 대한 개별 산술 연산을 수행합니다. 즉, 두 번 또는 네 번의 더하기 또는 빼기를 수행하거나 한 번의 더하기와 한 번의 빼기를 수행합니다.

다음과 같은 다양한 유형의 산술을 선택할 수 있습니다.

- 부호 있는 또는 부호 없는 산술 모듈로 2^8 또는 2^{16}, APSR GE 플래그 설정 (*조건 플래그* 참조)
- 부호 있는 범위 $-2^{15} ?x ?2^{15}-1$ 또는 $-2^7 \leq x \leq 2^7-1$ 중 하나에 대한 부호 있는 포화 산술. 이러한 연산이 포화되는 경우에도 Q 플래그는 영향을 받지 않습니다.
- 부호 없는 범위 $0 \leq x \leq 2^{16}-1$ 또는 $0 \leq x \leq 2^8-1$ 중 하나에 대한 부호 없는 포화 산술. 이러한 연산이 포화되는 경우에도 Q 플래그는 영향을 받지 않습니다.
- 결과를 양분하는 부호 있는 또는 부호 없는 산술. 이 연산은 오버플로를 일으킬 수 없습니다.

조건 플래그

이러한 명령어는 N, Z, C, V 또는 Q 플래그에 영향을 주지 않습니다.

이러한 명령어의 Q, SH, UQ 및 UH 접두사 변형은 플래그를 변경하지 않습니다.

이러한 명령어 세트의 S 및 U 접두사 변형은 다음과 같이 APSR에 GE 플래그를 설정합니다.

- 바이트 연산의 경우, GE 플래그는 다음과 같이 32비트 SUB 및 ADD 명령어의 C (Carry) 플래그와 동일한 방식으로 사용됩니다.
  - GE[0] 결과 비트[7:0]의 경우
  - GE[1] 결과 비트[15:8]의 경우
  - GE[2] 결과 비트[23:16]의 경우
  - GE[3] 결과 비트[31:24]의 경우
- 하프워드 연산의 경우, GE 플래그는 다음과 같이 일반 워드 SUB 및 ADD 명령어의 C (Carry) 플래그와 동일한 방식으로 사용됩니다.
  - GE[1:0] 결과 비트[15:0]의 경우
  - GE[3:2] 결과 비트[31:16]의 경우

이러한 플래그를 사용하면 그 다음 SEL 명령어를 제어할 수 있습니다 (4-63페이지의 *SEL* 참조).

참고

하프워드 연산의 경우, GE[1:0]이 함께 설정되거나 해제되고 GE[3:2]가 함께 설정되거나 해제됩니다.

아키텍처

이러한 ARM 명령어는 ARMv6 이상에서 사용할 수 있습니다.

이러한 32비트 Thumb 명령어는 ARMv7-M 프로필을 제외하고 ARMv6T2 이상에서 사용할 수 있습니다.

이러한 명령어의 16비트 Thumb 버전은 없습니다.

예제

```
SHADD8      r4, r3, r9
USAXNE      r0, r0, r2
```

올바르지 않은 예제

```
QHADD       r2, r9, r3    ; No such instruction, should be QHADD8 or QHADD16
SAX         r10, r8, r5   ; Must have a prefix.
```

4.6.2 USAD8 및 USADA8

부호 없는 절대차의 합 및 부호 없는 절대차의 합 누산

구문

USAD8{*cond*} {*Rd*}, *Rn*, *Rm*

USADA8{*cond*} *Rd*, *Rn*, *Rm*, *Ra*

인수 설명:

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

Rd 대상 레지스터입니다.

Rn 첫 번째 피연산자가 들어 있는 레지스터입니다.

Rm 두 번째 피연산자가 들어 있는 레지스터입니다.

Ra 누산 피연산자가 들어 있는 레지스터입니다.

Rd, *Rn*, *Rn* 또는 *Ra*에 r15를 사용하면 안 됩니다.

연산

USAD8 명령어는 *Rn*의 해당 바이트에 있는 부호 없는 값 사이에서 네 가지 차이를 찾아서 *Rm*. 해당 절대 값을 더하고 결과를 *Rd*에 저장합니다.

USADA8 명령어는 네 가지 차이의 절대 값을 *Ra*의 값에 더하고 결과를 *Rd*에 저장합니다.

조건 플래그

이러한 명령어는 플래그를 변경하지 않습니다.

아키텍처

이러한 ARM 명령어는 ARMv6 이상에서 사용할 수 있습니다.

이러한 32비트 Thumb 명령어는 ARMv7-M 프로필을 제외하고 ARMv6T2 이상에서 사용할 수 있습니다.

이러한 명령어의 16비트 Thumb 버전은 없습니다.

예제

```
USAD8       r2, r4, r6
USADA8      r0, r3, r5, r2
USADA8VS    r0, r4, r0, r1
```

올바르지 않은 예제

```
USADA8      r2, r4, r6      ; USADA8 requires four registers
USADA16     r0, r4, r0, r1  ; no such instruction
```

4.6.3 SSAT16 및 USAT16

병렬 하프워드 포화 명령어

SSAT16은 부호 있는 값을 부호 있는 범위로 포화시킵니다.

USAT16은 부호 있는 값을 부호 없는 범위로 포화시킵니다.

구문

op{*cond*} *Rd*, *#sat*, *Rn*

인수 설명:

op 다음 중 하나입니다.

- SSAT16 부호 있는 포화
- USAT16 부호 없는 포화

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

Rd 대상 레지스터입니다.

sat 포화시킬 비트 위치를 지정합니다. SSAT16의 경우 1에서 16 사이에서 지정하고 USAT16의 경우 0에서 15 사이에서 지정합니다.

Rn 피연산자가 들어 있는 레지스터입니다.

Rd 또는 *Rn*에 r15를 사용하면 안 됩니다.

연산

임의의 비트 위치로 하프워드 부호 있는 포화 및 부호 없는 포화

SSAT16 명령어는 각 부호 있는 하프워드를 부호 있는 범위 $-2^{sat-1} \leq x \leq^{sat-1} -1$로 포화시킵니다.

USAT16 명령어는 각 부호 있는 하프워드를 부호 없는 범위 $0 \leq x \leq^{sat} -1$로 포화시킵니다.

조건 플래그

하프워드에서 포화가 발생하면 이러한 명령어는 Q 플래그를 설정합니다. Q 플래그의 상태를 보려면 MRS 명령어 (4-134페이지의 *MRS* 참조) 를 사용하십시오.

아키텍처

이러한 ARM 명령어는 ARMv6 이상에서 사용할 수 있습니다.

이러한 32비트 Thumb 명령어는 ARMv7-M 프로필을 제외하고 ARMv6T2 이상에서 사용할 수 있습니다.

이러한 명령어의 16비트 Thumb 버전은 없습니다.

예제

```
SSAT16  r7, #12, r7
USAT16  r0, #7, r5
```

올바르지 않은 예제

```
SSAT16  r1, #16, r2, LSL #4 ; shifts not permitted with halfword saturations
```

4.7 패킹 및 패킹 해제 명령어

이 단원에는 다음 소단원이 포함되어 있습니다.

- 4-106페이지의 *BFC 및 BFI*
  비트 필드 지우기 및 비트 필드 삽입
- 4-107페이지의 *SBFX 및 UBFX*
  부호 있는 또는 부호 없는 비트 필드 추출
- 4-108페이지의 *SXT, SXTA, UXT 및 UXTA*
  선택적 더하기 포함 부호 확장 또는 0 확장 명령어
- 4-111페이지의 *PKHBT 및 PKHTB*
  하프워드 패킹 명령어

4.7.1 BFC 및 BFI

비트 필드 지우기 및 비트 필드 삽입. 레지스터에서 인접 비트를 지우거나 한 레지스터에 있는 인접 비트를 다른 레지스터에 삽입합니다.

구문

`BFC{cond} Rd, #lsb, #width`

`BFI{cond} Rd, Rn, #lsb, #width`

인수 설명:

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

Rd 대상 레지스터입니다. *Rd* 는 r15이면 안 됩니다.

Rn 소스 레지스터입니다. *Rn* 은 r15이면 안 됩니다.

lsb 지우거나 복사할 최하위 비트입니다.

width 지우거나 복사할 비트 수입니다. *width*는 0이 아니어야 하며, (*width*+*lsb*) 는 32보다 작아야 합니다.

BFC

*Rd*의 *width* 비트는 *lsb*부터 지워집니다. *Rd*의 다른 비트는 변경되지 않습니다.

BFI

*Rd*에서 *lsb*부터 시작하는 *width* 비트는 *Rn*에서 비트[0]부터 시작하는 *width* 비트로 대체됩니다. *Rd*의 다른 비트는 변경되지 않습니다.

조건 플래그

이러한 명령어는 플래그를 변경하지 않습니다.

아키텍처

이러한 ARM 명령어는 ARMv6T2 이상에서 사용할 수 있습니다.

이러한 32비트 Thumb 명령어는 ARMv6T2 이상에서 사용할 수 있습니다.

이러한 명령어의 16비트 Thumb 버전은 없습니다.

4.7.2 SBFX 및 UBFX

부호 있는 및 부호 없는 비트 필드 추출. 한 레지스터에 있는 인접 비트를 두 번째 레지스터의 최하위 비트에 복사하고 32비트로 부호 확장 또는 0 확장합니다.

구문

op{*cond*} *Rd*, *Rn*, *#lsb*, *#width*

인수 설명:

op SBFX 또는 UBFX입니다.

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

Rd 대상 레지스터입니다.

Rn 소스 레지스터입니다.

lsb 비트 필드에 있는 최하위 비트의 비트 수로, 0에서 31 사이의 숫자입니다.

width 비트 필드의 너비로, 1에서 (32–*lsb*) 사이의 숫자입니다.

Rd 또는 *Rn*에 r15를 사용하면 안 됩니다.

조건 플래그

이러한 명령어는 플래그를 변경하지 않습니다.

아키텍처

이러한 ARM 명령어는 ARMv6T2 이상에서 사용할 수 있습니다.

이러한 32비트 Thumb 명령어는 ARMv6T2 이상에서 사용할 수 있습니다.

이러한 명령어의 16비트 Thumb 버전은 없습니다.

4.7.3 SXT, SXTA, UXT 및 UXTA

부호 확장, 더하기 포함 부호 확장, 0 확장 및 더하기 포함 0 확장

구문

SXT<*extend*>{*cond*} {*Rd*}, *Rm* {,*rotation*}

SXTA<*extend*>{*cond*} {*Rd*}, *Rn*, *Rm* {,*rotation*}

UXT<*extend*>{*cond*} {*Rd*}, *Rm* {,*rotation*}

UXTA<*extend*>{*cond*} {*Rd*}, *Rn*, *Rm* {,*rotation*}

인수 설명:

<extend> 다음 중 하나입니다.

- B16 두 개의 8비트 값을 두 개의 16비트 값으로 확장합니다.
- B 8비트 값을 32비트 값으로 확장합니다.
- H 16비트 값을 32비트 값으로 확장합니다.

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

Rd 대상 레지스터입니다.

Rn 더할 숫자가 들어 있는 레지스터입니다 (SXTA 및 UXTA에만 해당).

Rm 확장할 값이 들어 있는 레지스터입니다.

rotation 다음 중 하나입니다.

- ROR #8 *Rm*의 값이 8비트만큼 오른쪽으로 회전합니다.
- ROR #16 *Rm*의 값이 16비트만큼 오른쪽으로 회전합니다.
- ROR #24 *Rm*의 값이 24비트만큼 오른쪽으로 회전합니다.

*rotation*이 생략될 경우 회전이 수행되지 않습니다.

Rd, *Rn* 또는 *Rm*에 r15를 사용하면 안 됩니다.

연산

이러한 명령어는 다음을 수행합니다.

1. *Rm*의 값이 0, 8, 16 또는 24비트만큼 오른쪽으로 회전합니다.
2. 얻은 값에 대해 다음 중 하나를 수행합니다.
   - 비트[7:0]을 추출하고 32비트로 부호 확장 또는 0 확장합니다. 명령어가 확장 및 더하기이면 *Rn*의 값을 더합니다.
   - 비트[15:0]을 추출하고 32비트로 부호 확장 또는 0 확장합니다. 명령어가 확장 및 더하기이면 *Rn*의 값을 더합니다.
   - 비트[23:16] 및 비트[7:0]을 추출하고 32비트로 부호 확장 또는 0 확장합니다. 명령어가 확장 및 더하기이면 *Rn*의 비트[31:16] 및 비트[15:0]에 각각 더해 결과의 비트[31:16] 및 비트[15:0]을 얻습니다.

조건 플래그

이러한 명령어는 플래그를 변경하지 않습니다.

16비트 명령어

Thumb 코드에서 사용하는 경우 다음 형식만 16비트 명령어이며 Thumb 2 이전 Thumb 코드에서 사용할 수 있습니다.

`SXTB` *Rd*, *Rm* — *Rd* 및 *Rm*은 모두 Lo 레지스터여야 합니다.

`SXTH` *Rd*, *Rm* — *Rd* 및 *Rm*은 모두 Lo 레지스터여야 합니다.

`UXTB` *Rd*, *Rm* — *Rd* 및 *Rm*은 모두 Lo 레지스터여야 합니다.

`UXTH` *Rd*, *Rm* — *Rd* 및 *Rm*은 모두 Lo 레지스터여야 합니다.

아키텍처

이러한 ARM 명령어는 ARMv6 이상에서 사용할 수 있습니다.

`SXTA` 및 `UXTA` Thumb 명령어는 ARMv7-M 프로필을 제외하고 ARMv6T2 이상에서 사용할 수 있습니다.

`SXT` 및 `UXT` 32비트 Thumb 명령어는 ARMv6T2 이상에서 사용할 수 있습니다.

`SXT` 및 `UXT` 16비트 Thumb 명령어는 ARMv6 이상에서 사용할 수 있습니다.

예제

```
SXTH         r3, r9, r4
UXTAB16EQ    r0, r0, r4, ROR #16
```

올바르지 않은 예제

```
SXTH     r9, r3, r2, ROR #12 ; rotation must be by 0, 8, 16, or 24.
```

4.7.4 PKHBT 및 PKHTB

하프워드 패킹 명령어

두 레지스터의 하프워드를 결합합니다. 하프워드를 추출하기 전에 피연산자 중 하나가 시프트될 수 있습니다.

구문

PKHBT{*cond*} {*Rd*}, *Rn*, *Rm*{, LSL #*leftshift*}

PKHTB{*cond*} {*Rd*}, *Rn*, *Rm*{, ASR #*rightshift*}

인수 설명:

PKHBT *Rn*의 비트[15:0]을 *Rm*에서 시프트된 값의 비트[31:16]과 결합합니다.

PKHTB *Rn*의 비트[31:16]을 *Rm*에서 시프트된 값의 비트[15:0]과 결합합니다.

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

Rd 대상 레지스터입니다.

Rn 첫 번째 피연산자가 들어 있는 레지스터입니다.

Rm 첫 번째 피연산자가 들어 있는 레지스터입니다.

leftshift 0 ~ 31 범위에 있어야 합니다.

rightshift 1 ~ 32 범위에 있어야 합니다.

Rd, *Rn* 또는 *Rm*에 r15를 사용하면 안 됩니다.

조건 플래그

이러한 명령어는 플래그를 변경하지 않습니다.

아키텍처

이러한 ARM 명령어는 ARMv6 이상에서 사용할 수 있습니다.

이러한 32비트 Thumb 명령어는 ARMv7-M 프로필을 제외하고 ARMv6T2 이상에서 사용할 수 있습니다.

이러한 명령어의 16비트 Thumb 버전은 없습니다.

예제

```
PKHBT   r0, r3, r5          ; combine the bottom halfword of r3 with
                            ; the top halfword of r5
PKHBT   r0, r3, r5, LSL #16 ; combine the bottom halfword of r3 with
                            ; the bottom halfword of r5
PKHTB   r0, r3, r5, ASR #16 ; combine the top halfword of r3 with
                            ; the top halfword of r5
```

또한 다양한 시프트 값을 사용하여 두 번째 피연산자의 스케일을 지정할 수도 있습니다.

올바르지 않은 예제

```
PKHBTEQ r4, r5, r1, ASR #8  ; ASR not permitted with PKHBT
```

4.8 분기 및 제어 명령어

이 단원에는 다음 소단원이 포함되어 있습니다.

- 4-114페이지의 *B, BL, BX, BLX 및 BXJ*

  분기, 링크 포함 분기, 분기 및 교환 명령어 세트, 링크 포함 분기 및 교환 명령어 세트, 분기 및 Jazelle로 변경 명령어 세트

- 4-118페이지의 *IT*

  If-Then. IT는 다음과 같은 최대 네 개의 명령어를 조건 명령어로 만듭니다. 명령어의 조건은 모두 동일할 수도 있고 일부 명령어의 조건이 다른 명령어의 조건과 반대 개념일 수도 있습니다. IT는 Thumb-2에서만 사용할 수 있습니다.

- 4-120페이지의 *CBZ 및 CBNZ*

  0 및 분기 비교. 이러한 명령어는 Thumb-2에서만 사용할 수 있습니다.

- 4-122페이지의 *TBB 및 TBH*

  테이블 분기 바이트 또는 하프워드. 이러한 명령어는 Thumb-2에서만 사용할 수 있습니다.

4.8.1 B, BL, BX, BLX 및 BXJ

분기, 링크 포함 분기, 분기 및 교환 명령어 세트, 링크 포함 분기 및 교환 명령어 세트, 분기 및 Jazelle 상태로 변경

구문

op1{*cond*}{.W} *label*

op2{*cond*} *Rm*

인수 설명:

op1 다음 중 하나입니다.

- B 분기
- BL 링크 포함 분기
- BLX 링크 포함 분기 및 교환 명령어 세트

op2 다음 중 하나입니다.

- BX 분기 및 교환 명령어 세트
- BLX 링크 포함 분기 및 교환 명령어 세트
- BXJ 분기 및 Jazelle 실행으로 변경

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조). *cond*는 이 명령어의 모든 형식에서 사용할 수 없습니다 (4-115페이지의 *명령어 사용 가능 여부 및 분기 범위* 참조).

.W Thumb-2에서 32비트 B 명령어를 사용하도록 지시하는 선택적 명령어 너비 지정자입니다. 자세한 내용은 4-116페이지의 *Thumb-2의 B*를 참조하십시오

label 프로그램 기준 식입니다. 자세한 내용은 3-38페이지의 *레지스터 기준 및 프로그램 기준 식*을 참조하십시오.

Rm 분기할 주소가 들어 있는 레지스터입니다.

연산

이러한 모든 명령어는 *label* 또는 *Rm*에 들어 있는 주소로 분기를 생성합니다. 이외에도 이러한 명령어는 다음 작업을 수행합니다.

- BL 및 BLX 명령어는 그 다음 명령어의 주소를 lr (r14, 링크 레지스터) 에 복사합니다.

- BX 및 BLX 명령어는 프로세서 상태를 ARM에서 Thumb으로 변경하거나 Thumb에서 ARM으로 변경할 수 있습니다.

  BLX *label* 은 항상 상태를 변경합니다.

  BX *Rm* 및 BLX *Rm*은 다음과 같이 *Rm*의 비트[0]에서 타겟 상태를 파생합니다.
  - *Rm*의 비트[0]이 0일 경우 프로세서가 ARM 상태로 변경되거나 ARM 상태를 유지합니다.
  - *Rm*의 비트[0]이 1일 경우 프로세서가 Thumb 상태로 변경되거나 Thumb 상태를 유지합니다.

- BXJ 명령어는 프로세서를 Jazelle 상태로 변경합니다.

명령어 사용 가능 여부 및 분기 범위

표 4-7에서는 ARM 및 Thumb 상태에서 사용할 수 있는 명령어를 보여 줍니다. 이 표에 나와 있지 않은 명령어는 사용할 수 없습니다. 괄호 안의 참고 정보는 명령어를 사용할 수 있는 첫 번째 아키텍처 버전을 나타냅니다.

표 4-7 분기 명령어 사용 가능 여부 및 범위

명령어	ARM		32비트 Thumb		32비트 Thumb	
B label	±32MB	(모두)	±2KB	(모든 T)	±16MB[a]	(모든 T2)
B{cond} label	±32MB	(모두)	–252 ~ +258	(모든 T)	1MB[a]	(모든 T2)
BL label	±32MB	(모두)	±4MB [b]	(모든 T)	±16MB	(모든 T2)
BL{cond} label	±32MB	(모두)	-		-	-
BX Rm	사용할수 있음	(4T, 5)	사용할 수 있음	(모든 T)	16비트 사용	(모든 T2)
BX{cond} Rm	사용할수 있음	(4T, 5)	-		-	-
BLX label	±32MB	(5)	±4MB [b]	(5T)	±16MB	(ARMv7-M을 제외한 모든 T2)
BLX Rm	사용할수 있음	(5)	사용할 수 있음	(5T)	16비트 사용	(모든 T2)

표 4-7 분기 명령어 사용 가능 여부 및 범위 (계속)

명령어	ARM		32비트 Thumb	32비트 Thumb	
BLX{cond} Rm	사용할수 있음	(5)	-	-	-
BXJ Rm	사용할수 있음	(5J, 6)	-	사용할 수 있음	(ARMv7-M을 제외한 모든 T2)
BXJ{cond} Rm	사용할수 있음	(5J, 6)	-	-	-

a. .W를 사용하여 이 32비트 명령어를 사용하도록 어셈블러에 지시합니다.
b. 명령어 쌍입니다.

분기 범위 확장

시스템 수준 B 및 BL 명령어는 현재 명령어의 주소부터 시작하도록 범위가 제한됩니다. 그러나 *label*이 범위를 벗어나더라도 이러한 명령어를 사용할 수 있습니다. 종종 링커에서 배치하는 *label*의 위치를 모를 수 있습니다. 필요한 경우 링커에서는 더 긴 분기를 활성화하는 코드를 추가합니다. *링커 사용 설명서*에서 3장 *기본 링커 기능 사용*을 참조하십시오. 이러한 코드를 *베니어*라고 합니다.

Thumb-2의 B

.W 너비 지정자를 사용하여 Thumb-2 코드에서 32비트 명령어를 생성하도록 B에 지시할 수 있습니다.

B.W는 16비트 명령어를 사용하여 타겟에 도달할 수 있는 경우에도 항상 32비트 명령어를 생성합니다.

정방향 참조의 경우, 32비트 Thumb 명령어를 사용하여 도달할 수 있는 타겟에서 오류를 발생시킬 수 있더라도 .W가 없는 B는 Thumb 코드에서 항상 16비트 명령어를 생성합니다.

Thumb-2EE의 BX, BLX 및 BXJ

이러한 명령어는 Thumb-2EE 코드에서 분기로 사용할 수 있지만 상태를 변경하는 데는 사용할 수는 없습니다. Thumb-2EE에서는 이러한 명령어의 *op*{*cond*} *label* 형식을 사용할 수 없습니다. 레지스터 형식에서 *Rm*의 비트[0]은 1이어야 하고 실행은 ThumbEE 상태의 타겟 주소에서 계속됩니다.

조건 플래그

이러한 명령어는 플래그를 변경하지 않습니다.

아키텍처

이러한 명령어의 아키텍처별 사용 가능 여부에 대한 자세한 내용은 4-115페이지의 *명령어 사용 가능 여부 및 분기 범위*를 참조하십시오.

예제

```
B       loopA
BLE     ng+8
BL      subC
BLLT    rtX
BEQ     {pc}+4  ; #0x8004
```

4.8.2 IT

IT (If-Then) 명령어는 다음 명령어 (*IT 블록*) 를 최대 네 개까지 조건 명령어로 만듭니다. 조건은 모두 동일할 수도 있고 일부 명령어의 조건이 다른 명령어의 조건과 논리적으로 반대일 수도 있습니다.

IT 블록의 명령어 (분기 포함) 는 구문의 {*cond*} 부분에서 조건도 지정해야 합니다.

어셈블러가 다음 명령어에 지정되어 있는 조건에 따라 IT 명령어를 자동으로 생성하므로, 코드에서 해당 명령어를 작성할 필요가 없습니다. 그러나 IT 명령어를 작성하는 경우에는 어셈블러가 IT 명령어에 지정되어 있는 조건을 다음 명령어에 지정되어 있는 조건에 대해 확인합니다.

IT 명령어를 작성하면 코드 디자인에서 조건 명령어 배치와 조건 선택을 고려할 수 있습니다.

ARM 코드로 어셈블할 때도 어셈블러는 동일한 확인을 수행하지만, 이 경우에는 IT 명령어를 생성하지 않습니다.

구문

IT{*x*{*y*{*z*}}} {*cond*}

인수 설명:

x IT 블록의 두 번째 명령어에 대한 조건 스위치를 지정합니다.

y IT 블록의 세 번째 명령어에 대한 조건 스위치를 지정합니다.

z IT 블록의 네 번째 명령어에 대한 조건 스위치를 지정합니다.

cond IT 블록의 첫 번째 명령어에 대한 조건을 지정합니다.

IT 블록의 두 번째, 세 번째 및 네 번째 명령어에 대한 조건 스위치는 다음 중 하나일 수 있습니다.

T Then. *cond* 조건을 명령어에 적용합니다.

E Else. *cond*의 반대 조건을 명령어에 적용합니다.

사용법

CMP, CMN, 및 TST를 제외하면, 일반적으로는 조건 코드 플래그에 영향을 주는 16비트 명령어는 IT 블록 내에서 사용될 때 해당 플래그에 영향을 주지 않습니다.

IT 블록의 BKPT 명령어는 조건이 실패한 경우에도 항상 실행됩니다.

참고

AL을 사용하여 무조건 명령어에 대해 IT 블록을 사용할 수 있습니다.

IT 블록 내의 조건 분기는 IT 블록 외부의 조건 분기에 비해 분기 범위가 더 깁니다.

제한

다음 명령어는 IT 블록에서 사용할 수 없습니다.

- IT
- CBZ 및 CBNZ
- TBB 및 TBH
- CPS, CPSID 및 CPSIE
- SETEND

IT 블록을 사용할 때 적용되는 기타 제한은 다음과 같습니다.

- pc를 수정하는 분기 또는 명령어는 블록의 마지막 명령어인 경우에만 IT 블록에서 사용할 수 있습니다.
- 예외 처리기에서 복귀하는 경우가 아니면 IT 블록의 명령어로 분기할 수 없습니다.
- IT 블록에는 어셈블러 지시어를 사용할 수 없습니다.

조건 플래그

이 명령어는 플래그를 변경하지 않습니다.

예외

IT 명령어와 해당 IT 블록 사이 또는 IT 블록 내에서는 예외가 발생할 수 있습니다. 이러한 예외가 발생하면 lr 및 SPSR에 적합한 반환 정보가 포함된 적절한 예외 처리기가 시작됩니다.

예외 반환에 사용되도록 설계된 명령어는 예외에서 복귀하고 IT 블록의 실행을 올바로 다시 시작하는 기본 방법으로 사용될 수 있습니다. 이것은 pc 수정 명령어를 IT 블록의 명령어로 분기할 수 있는 유일한 방법입니다.

아키텍처

이 16비트 Thumb 명령어는 ARMv6T2 이상에서 사용할 수 있습니다.

예제

```
ITTE   NE        ; IT can be omitted
ANDNE  r0,r0,r1  ; 16-bit AND, not ANDS
ADDSNE r2,r2,#1  ; 32-bit ADDS (16-bit ADDS does not set flags in IT block)
MOVEQ  r2,r3     ; 16-bit MOV
ITT    AL        ; emit 2 non-flag setting 16-bit instructions
ADDAL  r0,r0,r1  ; 16-bit ADD, not ADDS
SUBAL  r2,r2,#1  ; 16-bit SUB, not SUB
ADD    r0,r0,r1  ; expands into 32-bit ADD
IT     NE
ADD    r0,r0,r1  ; syntax error: no condition code used in IT block
ITT    EQ
MOVEQ  r0,r1
BEQ    dloop
```

메모

IT 블록의 예상할 수 없는 명령어

어셈블러에서는 IT 블록에 예상할 수 없는 명령어 (예: B, BL 및 CPS)에 대해 경고를 표시합니다. 또한 pc를 변경하는 명령어 (예: BX, CBZ 및 RFE) 에 대해서도 경고를 표시합니다.

4.8.3 CBZ 및 CBNZ

0인 경우 비교 및 분기, 0이 아닌 경우 비교 및 분기

구문

CBZ *Rn*, *label*

CBNZ *Rn*, *label*

인수 설명:

Rn 피연산자가 들어 있는 레지스터입니다.

label 분기 대상입니다.

사용법

CBZ 또는 CBNZ 명령어를 사용하면 조건 코드 플래그를 변경하지 않아도 되며 명령어 수를 줄일 수 있습니다.

조건 코드 플래그를 변경하지 않는다는 점을 제외하고 CBZ Rn, label은 다음과 같습니다.

```
CMP     Rn, #0
BEQ     label
```

조건 코드 플래그를 변경하지 않는다는 점을 제외하고 CBNZ Rn, label은 다음과 같습니다.

```
CMP     Rn, #0
BNE     label
```

제한

분기 대상이 명령어 다음에 4 ~ 30바이트 내에 있어야 합니다.

이러한 명령어는 IT 블록 내에서 사용하면 안 됩니다.

조건 플래그

이러한 명령어는 플래그를 변경하지 않습니다.

아키텍처

이러한 16비트 Thumb 명령어는 ARMv6T2 이상에서 사용할 수 있습니다.

이러한 명령어의 ARM 또는 32비트 Thumb 버전은 없습니다.

4.8.4 TBB 및 TBH

테이블 분기 바이트 및 테이블 분기 하프워드

구문

```
TBB [Rn, Rm]
TBH [Rn, Rm, LSL #1]
```

인수 설명:

Rn 기준 레지스터입니다. 이 레지스터에는 분기 길이 테이블의 주소가 포함됩니다. *Rn*은 r13이면 안 됩니다.

*Rn*에 r15를 지정하면 명령어 주소에 4를 더한 값이 사용됩니다.

Rm 인덱스 레지스터입니다. 테이블에 대한 인덱스가 포함되어 있습니다.

*Rm*은 r15 또는 r13이면 안 됩니다.

연산

이러한 명령어는 단일 바이트 오프셋 (TBB) 이나 하프워드 오프셋 (TBH) 테이블을 사용하여 pc 기준 정방향 분기를 생성합니다. *Rn*은 테이블에 대한 포인터를 제공하고 *Rm*은 테이블에 대한 인덱스를 제공합니다. 분기 길이는 테이블에서 반환된 바이트 (TBB) 또는 하프워드 (TBH) 값의 두 배입니다.

메모

Thumb-2EE에서는 기준 레지스터의 값이 0일 경우 실행이 HandlerBase - 4에서 NullCheck 처리기로 분기됩니다.

아키텍처

이러한 32비트 Thumb 명령어는 ARMv6T2 이상에서 사용할 수 있습니다.

이러한 명령어의 ARM 또는 16비트 Thumb 버전은 없습니다.

4.9 보조 프로세서 명령어

이 단원에서는 VFP (5장 *NEON 및 VFP 프로그래밍* 참조) 또는 Wireless MMX™ 기술 명령어 (6장 *Wireless MMX 기술 명령어* 참조) 에 대해 설명하지 않습니다. XScale 관련 명령어는 이 장의 뒷부분에서 설명합니다 (4-131페이지의 *기타 제한* 참조).

이 장에는 다음 단원이 포함되어 있습니다.

- 4-124페이지의 *CDP 및 CDP2*
  보조 프로세서 데이터 연산
- 4-125페이지의 *MCR, MCR2, MCRR 및 MCRR2*
  가능한 한 보조 프로세서 연산을 통해 ARM 레지스터 또는 레지스터에서 보조 프로세서로 이동
- 4-127페이지의 *MRC, MRC2, MRRC 및 MRRC2*
  가능한 한 보조 프로세서 연산을 통해 보조 프로세서에서 ARM 레지스터 또는 레지스터로 이동
- 4-129페이지의 *LDC, LDC2, STC 및 STC2*
  메모리와 보조 프로세서 간 데이터 전송

참고

지정된 보조 프로세서가 없거나 활성화되지 않은 경우 보조 프로세서 명령어가 정의되지 않은 명령어 예외를 발생시킵니다.

4.9.1 CDP 및 CDP2

보조 프로세서 데이터 연산

구문

op{*cond*} *coproc*, *#opcode1*, *CRd*, *CRn*, *CRm*{, *#opcode2*}

인수 설명:

op	CDP 또는 CDP2입니다.
cond	선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조). ARM 코드에서 *cond*는 CDP2에 대해 허용되지 않습니다.
coproc	명령어의 대상 프로세서 이름입니다. 표준 이름은 p*n*입니다. 여기서 *n*은 0에서 15 사이의 정수입니다.
opcode1	4비트 보조 프로세서 관련 op 코드입니다.
opcode2	선택적 3비트 보조 프로세서 관련 op 코드입니다.
CRd, *CRn*, *CRm*	보조 프로세서 레지스터입니다.

사용법

이러한 명령어의 사용 방법은 보조 프로세서에 따라 다릅니다. 자세한 내용은 보조 프로세서 설명서를 참조하십시오.

아키텍처

CDP ARM 명령어는 모든 버전의 ARM 아키텍처에서 사용할 수 있습니다.

CDP2 ARM 명령어는 ARMv5 이상에서 사용할 수 있습니다.

이러한 32비트 Thumb 명령어는 ARMv6T2 이상에서 사용할 수 있습니다.

이러한 명령어의 16비트 Thumb 버전은 없습니다.

4.9.2 MCR, MCR2, MCRR 및 MCRR2

ARM 레지스터 또는 여러 ARM 레지스터에서 보조 프로세서로 이동. 보조 프로세서에 따라 다양한 연산을 추가로 지정할 수도 있습니다.

구문

op1{*cond*} *coproc*, *#opcode1*, *Rt*, *CRn*, *CRm*{, *#opcode2*}

op2{*cond*} *coproc*, *#opcode3*, *Rt*, *Rt2*, *CRm*

인수 설명:

op1 MCR 또는 MCR2입니다.

op2 MCRR 또는 MCRR2입니다.

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조). ARM 코드에서 *cond*는 MCR2 또는 MCRR2에 대해 허용되지 않습니다.

coproc 명령어의 대상 프로세서 이름입니다. 표준 이름은 p*n*입니다. 여기서 *n*은 0에서 15 사이의 정수입니다.

opcode1 3비트 보조 프로세서 관련 op 코드입니다.

opcode2 선택적 3비트 보조 프로세서 관련 op 코드입니다.

opcode3 4비트 보조 프로세서 관련 op 코드입니다.

Rt, *Rt2* ARM 소스 레지스터입니다. MCRR 또는 MCRR2는 r15를 사용하면 안 됩니다.

CRn, *CRm* 보조 프로세서 레지스터입니다.

사용법

이러한 명령어의 사용 방법은 보조 프로세서에 따라 다릅니다. 자세한 내용은 보조 프로세서 설명서를 참조하십시오.

아키텍처

MCR ARM 명령어는 모든 버전의 ARM 아키텍처에서 사용할 수 있습니다.

MCR2 ARM 명령어는 ARMv5 이상에서 사용할 수 있습니다.

MCRR ARM 명령어는 ARMv6 이상과 ARMv5의 E 변형에서 사용할 수 있습니다.

MCRR2 ARM 명령어는 ARMv6 이상에서 사용할 수 있습니다.

이러한 32비트 Thumb 명령어는 ARMv6T2 이상에서 사용할 수 있습니다.

이러한 명령어의 16비트 Thumb 버전은 없습니다.

4.9.3 MRC, MRC2, MRRC 및 MRRC2

보조 프로세서에서 ARM 레지스터 또는 레지스터로 이동

보조 프로세서에 따라 다양한 연산을 추가로 지정할 수도 있습니다.

구문

op1{*cond*} *coproc*, *#opcode1*, *Rt*, *CRn*, *CRm*{, *#opcode2*}

op2{*cond*} *coproc*, *#opcode3*, *Rt*, *Rt2*, *CRm*

인수 설명:

op1 MRC 또는 MRC2입니다.

op2 MRRC 또는 MRRC2입니다.

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조). ARM 코드에서 *cond*는 MRC2 또는 MRRC2에 대해 허용되지 않습니다.

coproc 명령어의 대상 프로세서 이름입니다. 표준 이름은 p*n*입니다. 여기서 *n*은 0에서 15 사이의 정수입니다.

opcode1 3비트 보조 프로세서 관련 op 코드입니다.

opcode2 선택적 3비트 보조 프로세서 관련 op 코드입니다..

opcode3 4비트 보조 프로세서 관련 op 코드입니다.

Rt, *Rt2* ARM 소스 레지스터입니다. r15를 사용하면 안 됩니다.

MRC 및 MRC2에서 *Rt*는 APSR_nzcv일 수 있습니다.

CRn, *CRm* 보조 프로세서 레지스터입니다.

사용법

이러한 명령어의 사용 방법은 보조 프로세서에 따라 다릅니다. 자세한 내용은 보조 프로세서 설명서를 참조하십시오.

아키텍처

MRC ARM 명령어는 모든 버전의 ARM 아키텍처에서 사용할 수 있습니다.

MRC2 ARM 명령어는 ARMv5 이상에서 사용할 수 있습니다.

MRRC ARM 명령어는 ARMv6 이상과 ARMv5의 E 변형에서 사용할 수 있습니다.

MRRC2 ARM 명령어는 ARMv6 이상에서 사용할 수 있습니다.

이러한 32비트 Thumb 명령어는 ARMv6T2 이상에서 사용할 수 있습니다.

이러한 명령어의 16비트 Thumb 버전은 없습니다.

4.9.4 LDC, LDC2, STC 및 STC2

메모리와 보조 프로세서 간 데이터 전송

구문

op{L}{*cond*} *coproc*, *CRd*, [*Rn*]

op{L}{*cond*} *coproc*, *CRd*, [*Rn*, #{-}*offset*]{!}

op{L}{*cond*} *coproc*, *CRd*, [*Rn*], #{-}*offset*

op{L}{*cond*} *coproc*, *CRd*, *label*

인수 설명:

op LDC, LDC2, STC 또는 STC2 중 하나입니다.

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

ARM 코드에서 *cond*는 LDC2 또는 STC2에 대해 허용되지 않습니다.

L long 전송을 지정하는 선택적 접미사입니다.

coproc 명령어의 대상 프로세서 이름입니다. 표준 이름은 p*n*입니다. 여기서 *n*은 0에서 15 사이의 정수입니다.

CRd 로드 또는 저장할 보조 프로세서 레지스터입니다.

Rn 메모리 주소의 기준이 되는 레지스터입니다. r15를 지정하면 현재 명령어의 주소에 8을 더한 값이 사용됩니다.

- 선택적 빼기 기호입니다. -가 있으면 *Rn*에서 오프셋을 빼고, 그렇지 않으면 *Rn*에 오프셋을 더합니다.

offset 0 ~ 1020 범위에 있는 4의 배수로 평가되는 식입니다.

! 선택적 접미사입니다. ! 기호가 있으면 오프셋이 포함된 주소가 *Rn*에 다시 기록됩니다.

label 워드로 정렬된 프로그램 기준 식입니다. 자세한 내용은 3-38페이지의 *레지스터 기준 및 프로그램 기준 식*을 참조하십시오.

*label*은 현재 명령어의 1020바이트 내에 있어야 합니다.

사용법

이러한 명령어의 사용 방법은 보조 프로세서에 따라 다릅니다. 자세한 내용은 보조 프로세서 설명서를 참조하십시오.

Thumb-2EE에서는 기준 레지스터의 값이 0일 경우 실행이 HandlerBase - 4에서 NullCheck 처리기로 분기됩니다.

아키텍처

LDC와 STC는 모든 버전의 ARM 아키텍처에서 사용할 수 있습니다.

LDC2와 STC2는 ARMv5 이상에서 사용할 수 있습니다.

이러한 32비트 Thumb 명령어는 ARMv6T2 이상에서 사용할 수 있습니다.

이러한 명령어의 16비트 Thumb 버전은 없습니다.

메모

STC 및 STC2 명령어에서 프로그램 기준 주소 지정을 향후 사용할 수 없습니다.

4.10 기타 제한

이 단원에는 다음 소단원이 포함되어 있습니다.

- 4-132페이지의 *BKPT*

  브레이크포인트
- 4-133페이지의 *SVC*

  관리자 호출 (이전 SWI)
- 4-134페이지의 *MRS*

  CPSR 또는 SPSR의 내용을 범용 레지스터로 이동
- 4-136페이지의 *MSR*

  즉치 상수를 사용하거나 범용 레지스터의 내용에서 CPSR 또는 SPSR의 지정된 필드 로드
- 4-138페이지의 *CPS*

  프로세서 상태 변경
- 4-140페이지의 *SMC*

  보안 모니터 호출 (이전 SMI)
- 4-141페이지의 *SETEND*

  CPSR에서 엔디안 비트
- 4-142페이지의 *NOP, SEV, WFE, WFI 및 YIELD*

  연산 없음, 이벤트 설정, 이벤트 대기, 인터럽트 대기 및 양도 힌트 명령어
- 4-144페이지의 *DBG, DMB, DSB 및 ISB*

  디버그, 데이터 메모리 장벽, 데이터 동기화 장벽 및 명령어 동기화 장벽 힌트 명령어
- 4-147페이지의 *MAR 및 MRA*

  두 개의 범용 레지스터와 40비트 내부 누산기 (XScale 보조 프로세서 0 명령어) 간에 데이터 전송

4.10.1 BKPT

브레이크포인트

구문

`BKPT #immed`

인수 설명:

immed 다음 범위에 있는 정수로 평가되는 식입니다.

- ARM 명령어의 경우, 0 ~ 65535 (16비트 값)
- 16비트 Thumb 명령어의 경우 0 ~ 255 (8비트 값)

사용법

`BKPT` 명령어는 프로세서를 디버그 상태로 변경합니다. 디버그 도구는 특정 주소의 명령어에 도달하면 이 명령어를 사용하여 시스템 상태를 조사할 수 있습니다.

ARM 상태와 Thumb 상태 모두에서 ARM 하드웨어는 *immed*를 무시합니다. 그러나 디버거는 이 식을 사용하여 브레이크포인트에 대한 추가 정보를 저장할 수 있습니다.

아키텍처

이 ARM 명령어는 ARMv5 이상에서 사용할 수 있습니다.

이 16비트 Thumb 명령어는 ARMv5 이상의 T 변형에서 사용할 수 있습니다.

이 명령어의 32비트 Thumb 버전은 없습니다.

4.10.2 SVC

관리자 호출

구문

SVC{*cond*} #*immed*

인수 설명:

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

immed 다음 범위에 있는 정수로 평가되는 식입니다.

- ARM 명령어의 경우 0 ~ 2^{24}–1 (24비트 값)
- 16비트 Thumb 명령어의 경우 0 ~ 255 (8비트 값)

사용법

SVC 명령어는 예외를 발생시킵니다. 즉, 프로세서 모드가 관리자로 변경되고, CPSR이 관리자 모드 SPSR로 저장되며, 예외가 SVC 벡터로 분기됩니다 (개발자 설명서의 6장 *프로세서 예외 처리* 참조).

프로세서에서는 *immed*를 무시하지만 예외 처리기에서는 요청 중인 서비스를 확인하기 위해 이 식을 검색할 수 있습니다.

참고

ARM 어셈블리 언어를 개발하는 과정에서 SWI 명령어의 이름이 SVC로 변경되었습니다. 이번 RVCT 릴리스에서는 SWI 명령어가 SVC로 디스어셈블되고 해당 명령어가 이전에는 SWI였다는 주석이 추가됩니다.

조건 플래그

이 명령어는 플래그를 변경하지 않습니다.

아키텍처

이 ARM 명령어는 모든 버전의 ARM 아키텍처에서 사용할 수 있습니다.

이 16비트 Thumb 명령어는 ARM 아키텍처의 모든 T 변형에서 사용할 수 있습니다.

4.10.3 MRS

PSR의 내용을 범용 레지스터로 이동

구문

`MRS{cond} Rd, psr`

인수 설명:

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

Rd 대상 레지스터입니다. *Rd*는 r15이면 안 됩니다.

psr 다음 중 하나입니다.

- APSR 모든 프로세서와 모든 모드에서 사용할 수 있습니다.
- CPSR ARMv7-M 및 ARMv6-M을 제외한 모든 프로세서에서 APSR의 제공되지 않는 동의어로 디버그 상태에서 사용됩니다.
- SPSR ARMv7-M 및 ARMv6-M을 제외한 모든 프로세서의 권한 모드에서만 사용할 수 있습니다.
- *Mpsr* ARMv7-M 및 ARMv6-M 프로세서에서만 사용할 수 있습니다.

Mpsr 다음 중 하나일 수 있습니다. IPSR, EPSR, IEPSR, IAPSR, EAPSR, MSP, PSP, XPSR, PRIMASK, BASEPRI, BASEPRI_MAX, FAULTMASK 또는 CONTROL

사용법

PSR을 업데이트하기 위한 읽기- 수정- 쓰기 시퀀스에서 MSR과 함께 MRS를 사용하여 프로세서 모드를 변경하거나 Q 플래그를 지우는 등의 작업을 수행합니다.

프로세스 스왑 코드에서는 스왑 중인 원본 프로세스의 프로그래머 모델 상태를 관련 PSR 내용과 함께 저장해야 합니다. 마찬가지로 스왑 중인 대상 프로세스의 상태도 복원해야 합니다. 이러한 작업에서는 MRS/store 및 load/MSR 명령어 시퀀스를 사용합니다.

SPSR

프로세서가 사용자 또는 시스템 모드에 있을 때는 SPSR에 액세스하려고 시도하면 안 됩니다. 이것은 사용자의 책임입니다. 어셈블러에서는 실행 시간에 프로세서 모드에 대한 정보가 없기 때문에 이에 대한 경고를 표시할 수 없습니다.

프로세서가 사용자 또는 시스템 모드에 있을 때 SPSR에 액세스하려고 하면 예상할 수 없는 결과가 발생합니다.

CPSR

CPSR 실행 상태 비트는 프로세서가 디버그 상태인 중단 디버그 모드에 있을 때만 읽을 수 있습니다. 그렇지 않으면 CPSR 실행 상태 비트가 0으로 표시됩니다.

조건 플래그는 모든 프로세서의 모든 모드에서 읽을 수 있습니다. CPSR 대신 APSR을 사용하십시오.

조건 플래그

이 명령어는 플래그를 변경하지 않습니다.

아키텍처

이 ARM 명령어는 모든 버전의 ARM 아키텍처에서 사용할 수 있습니다.

이러한 32비트 Thumb 명령어는 ARMv6T2 이상에서 사용할 수 있습니다.

이 명령어의 16비트 Thumb 버전은 없습니다.

4.10.4 MSR

즉치 상수나 범용 레지스터의 내용을 PSR (*프로그램 상태 레지스터*) 의 지정된 필드에 로드

구문 (ARMv7-M 및 ARMv6-M 제외)

```
MSR{cond} APSR_flags, #constant
MSR{cond} APSR_flags, Rm
MSR{cond} psr_fields, #constant
MSR{cond} psr_fields, Rm
```

인수 설명:

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

flags 이동할 APSR 플래그를 지정합니다. *flags*는 다음 중 하나 이상일 수 있습니다.

- nzcvq — ALU 플래그 필드 마스크의 경우, PSR[31:27] (사용자 모드)
- g — SIMD GE 플래그 필드 마스크의 경우, PSR[19:16] (사용자 모드)

constant 숫자 상수로 평가되는 식입니다. 상수는 32비트 워드 내에서 짝수 비트 수만큼 회전하는 8비트 패턴에 해당해야 합니다. Thumb에서는 사용할 수 없습니다.

Rm 소스 레지스터입니다.

psr 다음 중 하나입니다.

- CPSR — 디버그 상태에서 사용할 수 있으며 APSR에 대한 향후 제공되지 않을 동의어이기도 합니다.
- SPSR — 모든 프로세서의 권한 모드에서만 사용할 수 있습니다.

fields 이동할 SPSR 또는 CPSR 필드를 지정합니다. *fields*는 다음 중 하나 이상일 수 있습니다.

- c — 제어 필드 마스크 바이트의 경우, PSR[7:0] (권한 모드)
- x — 확장 필드 마스크 바이트의 경우, PSR[15:8] (권한 모드)
- s — 상태 필드 마스크 바이트의 경우, PSR[23:16] (권한 모드)
- f — 플래그 필드 마스크 바이트의 경우, PSR[31:24] (권한 모드)

구문 (ARMv7-M 및 ARMv6-M에만 해당)

MSR{*cond*} *psr*, *Rm*

인수 설명:

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

Rm 소스 레지스터입니다.

psr 다음 중 하나일 수 있습니다. APSR, IPSR, EPSR, IEPSR, IAPSR, EAPSR, XPSR, MSP, PSP, PRIMASK, BASEPRI, BASEPRI_MAX, FAULTMASK 또는 CONTROL

사용법

자세한 내용은 4-134페이지의 *MRS*를 참조하십시오.

사용자 모드

- APSR을 사용하여 조건 플래그, Q 또는 GE 비트에 액세스합니다.
- CPSR의 할당되지 않은 상태 비트, 권한 있는 상태 비트 또는 실행 상태 비트에 쓰기가 무시됩니다. 이것은 사용자 모드 프로그램을 권한 모드로 변경할 수 없도록 합니다.

사용자 또는 시스템 모드에 있을 때 SPSR에 액세스하면 예상할 수 없는 결과가 발생합니다.

조건 플래그

APSR_nzcvq 또는 CPSR_f 필드를 지정하면 이 명령어는 플래그를 명시적으로 업데이트합니다.

아키텍처

이 ARM 명령어는 모든 버전의 ARM 아키텍처에서 사용할 수 있습니다.

이 32비트 Thumb 명령어는 ARMv6T2 이상에서 사용할 수 있습니다.

이 명령어의 16비트 Thumb 버전은 없습니다.

4.10.5 CPS

CPS (프로세서 상태 변경) 는 CPSR에서 하나 이상의 모드와 A, I 및 F 비트를 변경하고 다른 CPSR 비트는 변경하지 않습니다.

CPS는 권한 모드에서만 사용할 수 있고 사용자 모드에서는 아무런 영향을 주지 않습니다.

CPS는 조건 명령어일 수 없으며 IT 블록에서 사용할 수 없습니다.

구문

CPS*effect iflags*{, *#mode*}

CPS *#mode*

인수 설명:

effect 다음 중 하나입니다.
- IE 인터럽트 또는 어보트 사용
- ID 인터럽트 또는 어보트 사용 안 함

iflags 다음 중 하나 이상의 시퀀스입니다.
- a 부정확한 어보트를 사용하거나 사용하지 않습니다.
- i IRQ 인터럽트를 사용하거나 사용하지 않습니다.
- f FIQ 인터럽트를 사용하거나 사용하지 않습니다.

mode 변경할 모드의 번호를 지정합니다.

조건 플래그

이 명령어는 조건 플래그를 변경하지 않습니다.

16비트 명령어

이러한 명령어의 다음 형식은 Thumb-2 이전 Thumb 코드에서 사용할 수 있으며, Thumb-2 코드에서 사용될 경우 16비트 명령어입니다.

CPSIE *iflags*

CPSID *iflags*

16비트 Thumb 명령어에서 모드 변경을 지정할 수 없습니다.

아키텍처

이 ARM 명령어는 ARMv6 이상에서 사용할 수 있습니다.

이 32비트 Thumb 명령어는 ARMv6T2 이상에서 사용할 수 있습니다.

이 16비트 Thumb 명령어는 ARMv6 이상의 T 변형에서 사용할 수 있습니다.

예제

```
CPSIE if      ; enable interrupts and fast interrupts
CPSID A       ; disable imprecise aborts
CPSID ai, #17 ; disable imprecise aborts and interrupts, and enter FIQ mode
CPS #16       ; enter User mode
```

4.10.6 SMC

보안 모니터 호출

자세한 내용은 *ARM Architecture Reference Manual Security Extensions Supplement*를 참조하십시오.

구문

`SMC{cond} #immed_4`

인수 설명:

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

immed_4 4비트 즉치값입니다. ARM 프로세서에서는 이 값을 무시하지만 SMC 예외 처리기에서는 요청 중인 서비스를 확인하기 위해 이 값을 사용할 수 있습니다.

참고

ARM 어셈블리 언어를 개발하는 과정에서 `SMI` 명령어의 이름이 `SMC`로 변경되었습니다. 이번 RVCT 릴리스에서는 `SMI` 명령어가 `SMC`로 디스어셈블되고 해당 명령어가 이전에는 `SMI`였다는 주석이 추가됩니다.

아키텍처

이 ARM 명령어는 보안 확장이 있는 ARMv6 이상의 구현에서 사용할 수 있습니다.

이 32비트 Thumb 명령어는 보안 확장이 있는 ARMv6T2 이상의 구현에서 사용할 수 있습니다.

이 명령어의 16비트 Thumb 버전은 없습니다.

4.10.7 SETEND

CPSR에서 다른 비트에 영향을 주지 않고 엔디안 비트 설정

SETEND는 조건 명령어일 수 없으며 IT 블록에서 사용할 수 없습니다.

구문

SETEND *specifier*

인수 설명:

specifier 다음 중 하나입니다.

- BE 빅엔디안
- LE 리틀엔디안

사용법

SETEND를 사용하여 다양한 엔디안에 액세스합니다. 예를 들어 다른 리틀엔디안 응용 프로그램에서 여러 빅엔디안 DMA 서식 데이터 필드에 액세스합니다.

아키텍처

이 ARM 명령어는 ARMv6 이상에서 사용할 수 있습니다.

이 16비트 Thumb 명령어는 ARMv7-M 프로파일을 제외하고 ARMv6 이상의 T 변형에서 사용할 수 있습니다.

이 명령어의 32비트 Thumb 버전은 없습니다.

예제

```
SETEND  BE          ; Set the CPSR E bit for big-endian accesses
LDR     r0, [r2, #header]
LDR     r1, [r2, #CRC32]
SETEND  le          ; Set the CPSR E bit for little-endian accesses for the
                    ; rest of the application
```

4.10.8 NOP, SEV, WFE, WFI 및 YIELD

연산 없음, 이벤트 설정, 이벤트 대기, 인터럽트 대기 및 양도

구문

NOP{*cond*}

SEV{*cond*}

WFE{*cond*}

WFI{*cond*}

YIELD{*cond*}

인수 설명:

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

사용법

힌트 명령어로, 경우에 따라 구현하거나 구현하지 않을 수 있습니다. 이러한 명령어 중 하나를 구현하지 않으면 해당 명령어는 NOP처럼 동작합니다.

NOP

NOP는 아무 작업도 수행하지 않습니다. 타겟 아키텍처에서 NOP가 특정 명령어로 구현되지 않으면 어셈블러는 NOP를 의사 명령어로 간주하여 MOV r0, r0 (ARM) 또는 MOV r8, r8 (Thumb) 과 같이 아무 작업도 수행하지 않는 대체 명령어를 생성합니다.

NOP는 시간이 걸리는 NOP가 아닐 수도 있습니다. 프로세서는 이 명령어가 실행 단계에 도달하기 전에 파이프라인에서 이 명령어를 제거할 수 있습니다.

NOP를 패딩에 사용할 수 있습니다. 예를 들어 64비트 경계에 그 다음 명령어를 배치할 수 있습니다.

SEV

SEV는 다중 프로세서 시스템 내의 모든 코어에 신호를 보낼 이벤트를 발생시킵니다. SEV가 구현될 경우 WFE도 구현되어야 합니다.

WFE

이벤트 레지스터가 설정되지 않은 경우 WFE는 다음 이벤트 중 하나가 발생할 때까지 실행을 일시 중단합니다.

- CPSR I 비트로 마스킹되지 않은 경우, IRQ 인터럽트
- CPSR F 비트로 마스킹되지 않은 경우, FIQ 인터럽트
- CPSR A 비트로 마스킹되지 않은 경우, 부정확한 데이터 어보트
- 디버그를 사용하는 경우, 디버그 시작 요청
- SEV 명령어를 통해 다른 프로세서의 신호를 받는 이벤트

이벤트 레지스터가 설정될 경우 WFE는 이를 해제하고 즉시 원래 상태로 되돌립니다.

WFE가 구현될 경우 SEV도 구현되어야 합니다.

WFI

WFI는 다음 이벤트 중 하나가 발생할 때까지 실행을 일시 중단합니다.

- CPSR I 비트에 관계없이 IRQ 인터럽트
- CPSR F 비트에 관계없이 FIQ 인터럽트
- CPSR A 비트로 마스킹되지 않은 경우, 부정확한 데이터 어보트
- 디버그를 사용하는지 여부에 관계없이 디버그 시작 요청

YIELD

YIELD는 현재 스레드가 스왑이 가능한 spinlock과 같은 작업을 수행 중임을 하드웨어에 알립니다. 하드웨어는 이 힌트를 사용하여 다중 스레드 시스템에서 스레드를 일시 중단하고 다시 시작할 수 있습니다.

아키텍처

이러한 ARM 명령어는 ARMv6K 이상에서 사용할 수 있습니다.

이러한 32비트 Thumb 명령어는 ARMv6T2 이상에서 사용할 수 있습니다.

이러한 16비트 Thumb 명령어는 ARMv6T2 이상에서 사용할 수 있습니다.

NOP는 다른 모든 ARM 및 Thumb 아키텍처에서 의사 명령어로 사용할 수 있습니다.

4.10.9 DBG, DMB, DSB 및 ISB

디버그, 데이터 메모리 장벽, 데이터 동기화 장벽 및 명령어 동기화 장벽

구문

DBG{*cond*} {*#option*}

DMB{*cond*} {*#option*}

DSB{*cond*} {*#option*}

ISB{*cond*} {*#option*}

인수 설명:

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

option 힌트 작업에 대한 선택적 제한입니다.

사용법

힌트 명령어로, 경우에 따라 구현하거나 구현하지 않을 수 있습니다. 이러한 명령어 중 하나를 구현하지 않으면 해당 명령어는 NOP처럼 동작합니다.

DBG

디버그 힌트는 디버그 및 관련 시스템에 대한 힌트를 제공합니다. 이러한 시스템에서 이 명령어가 어떤 용도로 사용되는지 보려면 해당 시스템 설명서를 참조하십시오.

DMB

메모리 장벽 역할을 하는 데이터 메모리 장벽으로, 프로그램 순서에서 DMB 명령어 앞에 있는 모든 명시적 메모리 액세스가 DMB 명령어 뒤에 있는 명시적 메모리 액세스보다 앞에 표시되도록 합니다. 이때 프로세서에서 실행 중인 다른 명령어의 순서에는 영향을 주지 않습니다.

허용되는 *option*의 값은 다음과 같습니다.

SY 전체 시스템 DMB 작업. 기본값이며 생략할 수 있습니다.

ST 저장이 완료될 때까지만 기다리는 DMB 작업

ISH 공유 가능한 내부 도메인에만 수행되는 DMB 작업

ISHST 저장이 완료될 때까지만 기다리고 공유 가능한 내부 도메인까지만 수행되는 DMB 작업

NSH 통합 지점까지만 수행되는 DMB 작업

NSHST 저장이 완료될 때까지만 기다리고 통합 지점까지만 수행되는 DMB 작업

OSH 공유 가능한 외부 도메인에만 수행되는 DMB 작업

OSHST 저장이 완료될 때까지만 기다리고 공유 가능한 외부 도메인까지만 수행되는 DMB 작업

DSB

특수한 유형의 메모리 장벽 역할을 하는 데이터 동기화 장벽. 이 명령어가 완료될 때까지 프로그램 순서에서 이 명령어 뒤에 있는 명령어는 실행되지 않습니다. 이 명령어는 다음 경우에 완료됩니다.

- 이 명령어 앞에 있는 모든 명시적 메모리 액세스가 완료될 경우 다음 사항이 적용됩니다.
- 이 명령어 앞에 있는 모든 캐시, 분기 예측기 및 TLB 유지관리 작업이 완료될 경우 다음 사항이 적용됩니다.

허용되는 *option*의 값은 다음과 같습니다.

SY 전체 시스템 DSB 작업. 기본값이며 생략할 수 있습니다.

ST 저장이 완료될 때까지만 기다리는 DSB 작업

ISH 공유 가능한 내부 도메인에만 수행되는 DSB 작업

ISHST 저장이 완료될 때까지만 기다리고 공유 가능한 내부 도메인까지만 수행되는 DSB 작업

NSH 통합 지점까지만 수행되는 DSB 작업

NSHST 저장이 완료될 때까지만 기다리고 통합 지점까지만 수행되는 DSB 작업

OSH 공유 가능한 외부 도메인에만 수행되는 DSB 작업

OSHST 저장이 완료될 때까지만 기다리고 공유 가능한 외부 도메인까지만 수행되는 DSB 작업

ISB

명령어 동기화 장벽은 ISB 명령어가 완료된 후 이 명령어 뒤에 있는 모든 명령어를 캐시나 메모리에서 가져오기 위해 프로세서에서 파이프라인을 플러시합니다. 이렇게 하면 ISB 명령어 앞에서 실행된 CP15 레지스터에 대한 모든 변경 작업뿐 아니라 ASID 변경 같은 컨텍스트 변경 작업, 완료된 TLB 유지관리 작업 또는 분기 예측기 작업의 효과가 ISB 명령어 뒤에서 가져온 명령어에 나타납니다.

또한 ISB 명령어는 프로그램 순서에서 자신보다 뒤에 있는 모든 분기가 항상 ISB 명령어 뒤에 있는 컨텍스트를 사용하여 분기 예상 논리에 기록되도록 합니다. 이것은 명령어 스트림이 올바로 실행되도록 하는 데 필요합니다.

허용되는 *option*의 값은 다음과 같습니다.

SY　　전체 시스템 ISB 작업. 기본값이며 생략할 수 있습니다.

별칭

DMB 및 DSB에 대해서는 다음과 같은 *option*의 대체 값이 지원되지만, 이러한 값은 사용하지 않는 것이 좋습니다.

- SH는 ISH의 별칭입니다.
- SHST는 ISHST의 별칭입니다.
- UN은 NSH의 별칭입니다.
- UNST는 NSHST의 별칭입니다.

아키텍처

이러한 ARM 및 32비트 Thumb 명령어는 ARMv7에서 사용할 수 있습니다.

이러한 명령어의 16비트 Thumb 버전은 없습니다.

4.10.10 MAR 및 MRA

두 개의 범용 레지스터와 40비트 내부 누산기 간에 데이터를 전송합니다.

구문

`MAR{cond} Acc, RdLo, RdHi`

`MRA{cond} RdLo, RdHi, Acc`

인수 설명:

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

Acc 내부 누산기입니다. 표준 이름은 acc*x*입니다. 여기서 *x*는 0에서 *n* 사이의 정수입니다. *n*의 값은 프로세서에 따라 달라집니다. 현재 프로세서에서는 0입니다.

RdLo, *RdHi* 범용 레지스터입니다. *RdLo* 및 *RdHi*는 pc이면 안 되며, MRA에 대해 서로 다른 레지스터여야 합니다.

사용법

MAR 명령어는 *RdLo*의 내용을 *Acc*의 비트[31:0]에 복사하고 *RdHi*의 최하위 바이트를 *Acc*의 비트[39:32]에 복사합니다.

MRA 명령어는 다음을 수행합니다.

- *Acc*의 비트[31:0]을 *RdLo*에 복사합니다.
- *Acc*의 비트[39:32]를 *RdHi*의 비트[7:0]에 복사합니다.
- *Acc*의 비트[39]를 *RdHi*의 비트[31:8]에 복사하여 값을 부호 확장합니다.

아키텍처

이러한 ARM 보조 프로세서 0 명령어는 XScale 프로세서에서만 사용할 수 있습니다.

이러한 명령어의 Thumb 버전은 없습니다.

예제

```
MAR     acc0, r0, r1
MRA     r4, r5, acc0
MARNE   acc0, r9, r2
MRAGT   r4, r8, acc0
```

4.11 Thumb에서 명령어 너비 선택

ARMv6T2 이상 프로세서를 위한 Thumb 코드를 작성할 경우 일부 명령어에는 16비트 인코딩이나 32비트 인코딩이 있을 수 있습니다. 일반적으로 어셈블러는 두 인코딩을 모두 사용할 수 있는 경우 16비트 인코딩을 생성합니다. 이 동작의 예외에 대해서는 *일부 명령어의 다른 동작*을 참조하십시오.

4.11.1 명령어 너비 지정자, .W 및 .N

이 동작을 재정의하려면 .W 너비 지정자를 사용할 수 있습니다. 이 너비 지정자를 사용하면 16비트 인코딩을 사용할 수 있는 경우에도 어셈블러에서 32비트 인코딩을 생성합니다.

ARM 또는 Thumb (ARMv6T2 이상) 코드로 어셈블될 수 있는 코드에서 .W 지정자를 사용할 수 있습니다. .W 지정자는 ARM 코드로 어셈블될 때는 아무런 영향을 주지 않습니다.

명령어가 16비트로 인코딩되도록 하려면 .N 너비 지정자를 사용할 수 있습니다. 이 경우 명령어가 16비트로 인코딩될 수 없거나 ARM 코드로 어셈블하면 어셈블러에서 오류를 생성합니다.

명령어 너비 지정자를 사용하는 경우 다음과 같이 명령어 니모닉과 조건 코드 (있는 경우) 바로 뒤에 너비 지정자를 배치해야 합니다.

```
BCS.W   label   ; forces 32-bit instruction even for a short branch
B.N     label   : faults if label out of range for 16-bit instruction
```

4.11.2 일부 명령어의 다른 동작

정방향 참조의 경우, .W가 없는 LDR, ADR 및 B는 32비트 명령어를 사용하여 도달할 수 있는 타겟에서 오류를 발생시키는 경우에도 항상 16비트 명령어를 생성합니다.

외부 참조의 경우, .W가 없는 LDR 및 B는 항상 32비트 명령어를 생성합니다.

4.11.3 진단 경고

진단 경고를 사용하여 분기 명령어가 16비트로 인코딩될 수 있었지만 .W를 지정했기 때문에 32비트로 인코딩된 경우를 감지할 수 있습니다.

```
--diag_warning 1607
```

이 경고는 기본적으로 해제됩니다.

참고

최종 주소가 알려지지 않기 때문에 이 진단에서는 재배치된 분기 명령어에 대한 경고를 생성하지 않습니다. 분기가 32비트 명령어의 범위를 벗어나는 경우 링커에서 베니어를 삽입할 수 있습니다.

4.12 ThumbEE 명령어

ENTERX와 LEAVEX를 제외하고 이러한 ThumbEE 명령어는 --thumbx 명령 행 옵션이나 THUMBX 지시어를 사용하여 어셈블러가 ThumbEE 상태로 전환된 경우에만 허용됩니다.

이 단원에는 다음 소단원이 포함되어 있습니다.

- 4-151페이지의 *ENTERX 및 LEAVEX*
  Thumb 상태와 ThumbEE 상태 간 전환
- 4-152페이지의 *CHKA*
  배열 검사
- 4-153페이지의 *HB, HBL, HBLP 및 HBP*
  처리기 분기, 지정된 처리기로 분기

4.12.1 ENTERX 및 LEAVEX

Thumb 상태와 ThumbEE 상태 간 전환

구문

ENTERX

LEAVEX

사용법

ENTERX는 Thumb 상태를 ThumbEE 상태로 변경하거나 ThumbEE 상태를 유지합니다.

LEAVEX는 ThumbEE 상태를 Thumb 상태로 변경하거나 Thumb 상태를 유지합니다.

IT 블록에서는 ENTERX 또는 LEAVEX를 사용하면 안 됩니다.

아키텍처

이러한 명령어는 ARM 명령어 세트에서 사용할 수 없습니다.

이러한 32비트 Thumb 및 Thumb-2EE 명령어는 Thumb-2EE를 지원하는 ARMv7에서 사용할 수 있습니다.

이러한 명령어의 16비트 Thumb-2 버전은 없습니다.

자세한 내용은 *ARM Architecture Reference Manual Thumb-2 Execution Environment Supplement*를 참조하십시오.

4.12.2 CHKA

CHKA (배열 검사) 는 두 레지스터에 있는 부호 없는 값을 비교합니다.

첫 번째 레지스터의 값이 두 번째 레지스터의 값보다 작거나 같으면 pc를 lr에 복사하고 IndexCheck 처리기로 분기를 생성합니다.

구문

CHKA *Rn*, *Rm*

인수 설명:

Rn 배열 크기가 들어 있습니다. r15를 사용하면 안 됩니다.

Rm 배열 인덱스가 들어 있습니다. r13 또는 r15를 사용하지 마십시오.

아키텍처

이 명령어는 ARM 상태에서 사용할 수 없습니다.

이 16비트 ThumbEE 명령어는 Thumb-2EE를 지원하는 ARMv7에서만 사용할 수 있습니다.

4.12.3 HB, HBL, HBLP 및 HBP

처리기 분기, 지정된 처리기로 분기

이 명령어는 선택적으로 복귀 주소를 lr에 저장하거나 매개변수를 처리기에 전달할 수 있으며 두 작업을 모두 수행할 수도 있습니다.

구문

```
HB{L} #HandlerID
HB{L}P #immed, #HandlerID
```

인수 설명:

`L` 선택적 접미사입니다. `L`이 있을 경우 이 명령어는 복귀 주소를 lr에 저장합니다.

`P` 선택적 접미사입니다. `P`가 있을 경우 이 명령어는 *immed*의 값을 r8에 있는 처리기에 전달합니다.

immed 즉치값입니다. `L`이 있으면 *immed*는 0 ~ 31 범위에 있어야 하고, 그렇지 않으면 *immed*는 0 ~ 7 범위에 있어야 합니다.

HandlerID 호출할 처리기의 인덱스 번호입니다. *P*가 있으면 *HandlerID*는 0 ~ 31 범위에 있어야 하고, 그렇지 않으면 *HandlerID*는 0 ~ 255 범위에 있어야 합니다.

아키텍처

이러한 명령어는 ARM 상태에서 사용할 수 없습니다.

이러한 16비트 ThumbEE 명령어는 ThumbEE 상태와 Thumb-2EE를 지원하는 ARMv7에서만 사용할 수 있습니다.

4.13 의사 명령어

ARM 어셈블러는 어셈블리 타임에 ARM, Thumb-2 또는 Thumb-2 이전 Thumb 명령어의 적절한 조합으로 변환되는 많은 의사 명령어를 지원합니다.

의사 명령어에 대해서는 다음 단원에서 설명합니다.

- 4-155페이지의 *ADRL 의사 명령어*

  프로그램 기준 또는 레지스터 기준 주소를 레지스터로 로드 (중간 범위, 위치 독립적)

- 4-157페이지의 *MOV32 의사 명령어*

  32비트 상수 값 또는 주소를 사용하여 레지스터 로드 (무제한 범위, 위치 독립적이지 않음). ARMv6T2 이상에 대해서만 사용할 수 있습니다.

- 4-159페이지의 *LDR 의사 명령어*

  32비트 상수 값 또는 주소를 사용하여 레지스터 로드 (무제한 범위, 위치 독립적이지 않음). 모든 ARM 아키텍처에 대해 사용할 수 있습니다.

- 4-162페이지의 *UND 의사 명령어*

  아키텍처에서 정의되지 않은 명령어 생성. 모든 ARM 아키텍처에 대해 사용할 수 있습니다.

4.13.1 ADRL 의사 명령어

프로그램 기준 또는 레지스터 기준 주소를 레지스터로 로드합니다. 이 의사 명령어는 ADR 명령어와 유사합니다. ADRL은 두 개의 데이터 처리 명령어를 생성하므로 ADR보다 넓은 범위의 주소를 로드할 수 있습니다.

참고

ADRL은 Thumb-2 이전 프로세서에 대해 Thumb 명령어를 어셈블할 때 사용할 수 없습니다.

구문

ADRL{*cond*} *Rd*,*label*

인수 설명:

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

Rd 로드할 레지스터입니다.

label 프로그램 기준 또는 레지스터 기준 식입니다. 자세한 내용은 3-38페이지의 *레지스터 기준 및 프로그램 기준 식*을 참조하십시오.

사용법

ADRL은 항상 두 개의 32비트 명령어로 어셈블됩니다. 즉, 단일 명령어만으로 주소에 도달할 수 있는 경우에도 두 번째 중복 명령어가 생성됩니다.

어셈블러가 두 개의 명령어로 주소를 생성할 수 없으면 오류가 생성되고 어셈블리가 실패합니다. 넓은 범위의 주소를 로드하는 방법에 대한 자세한 내용은 4-159페이지의 *LDR 의사 명령어*를 참조하십시오. 또한 2-28페이지의 *레지스터에 상수 로드*도 참조하십시오.

ADRL은 주소가 프로그램 기준 또는 레지스터 기준 주소이기 때문에 위치 독립적인 코드를 생성합니다.

*label*이 프로그램 기준 주소이면 ADRL 의사 명령어와 동일한 어셈블러 영역에 있는 주소로 평가되어야 합니다 (7-70페이지의 *AREA* 참조).

ADRL을 사용하여 BX 또는 BLX 명령어의 타겟을 생성하는 경우 타겟에 Thumb 명령어가 포함되어 있으면 주소의 Thumb 비트 (비트 0) 를 설정하는 것은 사용자의 책임입니다.

아키텍처 및 범위

사용 가능한 범위는 다음과 같이 사용 중인 명령어 세트에 따라 달라집니다.

ARM 바이트 또는 하프워드로 정렬된 주소의 경우, ±64KB
워드로 정렬된 주소의 경우, ±256KB

32비트 Thumb 바이트, 하프워드 또는 워드로 정렬된 주소의 경우, ±1MB

16비트 Thumb ADRL을 사용할 수 없음

범위는 현재 명령어의 주소에서 4바이트 (Thumb 코드) 또는 2워드 (ARM 코드) 뒤에 있는 포인트를 기준으로 지정됩니다. 이 포인트를 기준으로 16바이트 이상 정렬될 경우 더욱 먼 주소 범위가 지정될 수 있습니다.

4.13.2 MOV32 의사 명령어

다음 중 하나를 사용하여 레지스터를 로드합니다.

- 32비트 상수 값
- 임의의 주소

MOV32는 항상 두 개의 32비트 명령어, MOV, MOVT 쌍을 생성합니다. 따라서 임의의 32비트 상수를 로드하거나 전체 32비트 주소 공간에 액세스할 수 있습니다.

구문

MOV32{*cond*} *Rd*, *expr*

인수 설명:

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

Rd 로드할 레지스터입니다. *Rd*는 sp 또는 pc이면 안 됩니다.

expr 다음 중 하나일 수 있습니다.

symbol	현재 프로그램 영역이나 다른 프로그램 영역에 있는 레이블
constant	임의의 32비트 상수
symbol + *constant*	레이블에 32비트 상수를 더한 값

사용법

MOV32 의사 명령어의 주 목적은 다음과 같습니다.

- 단일 명령어로 즉치값을 생성할 수 없을 때 리터럴 상수를 생성합니다.
- 프로그램 기준 또는 외부 주소를 레지스터로 로드합니다. 링커에 MOV32이 포함된 ELF 섹션을 배치하는 위치에 관계없이 주소는 유효한 상태로 유지됩니다.

  참고

  이러한 방식으로 로드된 주소는 링크 타임에 고정되므로 코드는 위치 독립적이지 *않*습니다.

MOV32는 참조되는 레이블이 Thumb 코드에 있는 경우 주소의 Thumb 비트 (비트 0)를 설정합니다.

아키텍처

이 의사 명령어는 ARMv6T2 이상의 ARM 및 Thumb 모두에서 사용할 수 있습니다.

4.13.3 LDR 의사 명령어

다음 중 하나를 사용하여 레지스터를 로드합니다.

- 32비트 상수 값
- 주소

참고

이 단원에서는 LDR *의사* 명령어에 대해서만 설명합니다. LDR *명령어*에 대한 자세한 내용은 4-10페이지의 *메모리 액세스 명령어*를 참조하십시오.

구문

LDR{*cond*}{.W} *Rt*, =*expr*

LDR{*cond*}{.W} *Rt*, =*label_expr*

인수 설명:

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

.W 선택적 명령어 너비 지정자입니다.

Rt 로드할 레지스터입니다.

expr 숫자 상수로 평가됩니다 (3-29페이지의 *숫자 상수* 참조).

- *expr*의 값이 범위 내에 있을 경우 어셈블러는 MOV 또는 MVN 명령어를 생성합니다.
- *expr*의 값이 MOV 또는 MVN 명령어의 범위 내에 있지 *않은* 경우 어셈블러는 리터럴 풀에 상수를 배치하고 리터럴 풀에서 상수를 읽는 프로그램 기준 LDR 명령어를 생성합니다.

상수 로드에 대한 자세한 내용은 2-33페이지의 *LDR Rd, =const를 통한 직접 로드*를 참조하십시오.

label_expr 레이블에 숫자 상수를 더하거나 뺀 형식으로 된 주소의 프로그램 기준 또는 외부 식입니다 (3-38페이지의 *레지스터 기준 및 프로그램 기준 식* 참조). 어셈블러는 *label_expr*의 값을 리터럴 풀에 배치하고 리터럴 풀에서 값을 로드하는 프로그램 기준 LDR 명령어를 생성합니다.

*label_expr*이 외부 식이거나 현재 섹션에 포함되어 있지 않으면 어셈블러에서는 객체 파일에 링커 재배치 지시어를 배치합니다. 링커에서는 링크 타임에 주소를 생성합니다.

*label_expr*이 지역 레이블 (3-32페이지의 *지역 레이블* 참조) 이면 어셈블러는 객체 파일에 링커 재배치 지시어를 배치하고 해당 지역 레이블을 나타내는 기호를 생성합니다. 주소는 링크 타임에 생성됩니다. 지역 레이블이 Thumb 코드를 참조하는 경우 주소의 Thumb 비트 (비트 0) 가 설정됩니다.

참고

RVCT 버전 2.2에서는 주소의 Thumb 비트가 설정되지 않습니다. 이 동작에 의존하는 코드가 있으면 명령 행 옵션 `--untyped_local_labels`를 사용하여 Thumb 코드에서 레이블을 참조할 때 Thumb 비트를 설정하지 않도록 어셈블러에 지시합니다.

사용법

LDR 의사 명령어의 주 목적은 다음과 같습니다.

- 즉치값이 MOV 및 MVN 명령어의 범위를 벗어나 레지스터로 이동할 수 없을 때 리터럴 상수를 생성합니다.
- 프로그램 기준 또는 외부 주소를 레지스터로 로드합니다. 링커에 LDR이 포함된 ELF 섹션을 배치하는 위치에 관계없이 주소는 유효한 상태로 유지됩니다.

  참고

  이러한 방식으로 로드된 주소는 링크 타임에 고정되므로 코드는 위치 독립적이지 *않*습니다.

pc부터 리터럴 풀에 있는 값까지의 오프셋은 ±4KB (ARM, 32비트 Thumb-2) 보다 작거나 0 ~ +1KB 범위 (16비트 Thumb-2, Thumb-2 이전 Thumb) 에 있어야 합니다. 범위에 리터럴 풀이 있는지 확인하는 것은 사용자의 책임입니다. 자세한 내용은 7-18페이지의 *LTORG*를 참조하십시오.

참조되는 레이블이 Thumb 코드에 있는 경우 LDR 의사 명령어는 *label_expr*의 Thumb 비트 (비트 0) 를 설정합니다.

LDR을 사용하는 방법과 MOV 및 MVN에 대한 자세한 내용은 2-28페이지의 *레지스터에 상수 로드*를 참조하십시오.

Thumb 코드의 LDR

.W 너비 지정자를 사용하여 ARMv6T2 이상 프로세서의 Thumb 코드에서 32비트 명령어를 생성하도록 LDR에 지시할 수 있습니다. 16비트 MOV에서 상수를 로드할 수 있거나 16비트 pc 기준 로드 범위 내에 리터럴 풀이 있을 경우 LDR.W는 항상 32비트 명령어를 생성합니다.

상수의 값이 어셈블러의 첫 번째 패스에서 알려지지 않는 경우, 32비트 MOV 또는 MVN 명령어에서 생성될 수 있는 상수에 대해 16비트 pc 기준 로드가 수행되더라도 .W가 없는 LDR은 Thumb 코드에서 16비트 명령어를 생성합니다. 그러나 상수가 첫 번째 패스에서 알려지고 32비트 MOV 또는 MVN 명령어를 사용하여 생성될 수 있으면 MOV 또는 MVN 명령어가 사용됩니다.

LDR 의사 명령어는 16비트 플래그 설정 MOV 명령어를 생성하지 않습니다. --diag_warning 1727 어셈블러 명령 행 옵션을 사용하면 16비트 명령어가 사용될 수 있었던 시기를 확인할 수 있습니다.

리터럴 풀에서 로드하지 않고 상수나 주소를 생성하는 데 대한 자세한 내용은 4-157페이지의 *MOV32 의사 명령어*를 참조하십시오.

예제

```
LDR     r3,=0xff0    ; loads 0xff0 into r3
                     ; =>  MOV.W r3,#0xff0
LDR     r1,=0xfff    ; loads 0xfff into r1
                     ; =>  LDR r1,[pc,offset_to_litpool]
                     ;     ...
                     ;     litpool DCD 0xfff
LDR     r2,=place    ; loads the address of
                     ; place into r2
                     ; =>  LDR r2,[pc,offset_to_litpool]
                     ;     ...
                     ;     litpool DCD place
```

4.13.4 UND 의사 명령어

아키텍처에서 정의되지 않은 명령어 생성. 정의되지 않은 명령어를 실행하려고 시도하면 정의되지 않은 명령어 예외가 발생합니다. 아키텍처에서 정의되지 않은 명령어는 정의되지 않은 상태로 남아 있습니다.

구문

`UND{cond}{.W} {#expr}`

인수 설명:

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조). *cond*는 Thumb-2 이전 Thumb 코드에서 이 의사 명령어에 대해 허용되지 않습니다.

`.W` 선택적 명령어 너비 지정자입니다.

expr 숫자 상수로 평가됩니다. 표 4-8에서는 명령어의 *expr* 범위 및 인코딩을 보여 줍니다. 여기서 Y는 *expr*에 대해 인코딩되는 비트 위치를 나타내고, V는 조건 코드에 대해 인코딩되는 4비트입니다.

*expr*이 생략되면 값 0이 사용됩니다.

표 4-8 *expr*의 범위 및 인코딩

명령어	인코딩	*expr*의 비트 수	범위
ARM	0xV7FYYYFY	16	0-65535
32비트 Thumb	0xF7FYAYFY	12	0-4095
16비트 Thumb	0xDEYY	8	0-255

Thumb 코드의 UND

`.W` 너비 지정자를 사용하여 ARMv6T2 이상 프로세서의 Thumb 코드에서 32비트 명령어를 생성하도록 `UND`에 지시할 수 있습니다. `UND.W`는 *expr*이 0 ~ 255 범위에 있는 경우에도 항상 32비트 명령어를 생성합니다.

디스어셈블리

이 의사 명령어가 생성하는 인코딩은 `DCI`로 디스어셈블됩니다.

5장
NEON 및 VFP 프로그래밍

이 장에서는 NEON™ 및 VFP 보조 프로세서의 어셈블리 프로그래밍을 설명합니다.

- 5-3페이지의 *명령어 요약*
- 5-9페이지의 *NEON 및 VFP에 대한 아키텍처 지원*
- 5-10페이지의 *확장 레지스터 뱅크*
- 5-13페이지의 *조건 코드*
- 5-15페이지의 *일반 정보*
- 5-23페이지의 *NEON 및 VFP 공유 명령어*
- 5-31페이지의 *NEON 논리 및 비교 연산*
- 5-41페이지의 *NEON 일반 데이터 처리 명령어*
- 5-53페이지의 *NEON 시프트 명령어*
- 5-60페이지의 *NEON 일반 산술 명령어*
- 5-73페이지의 *NEON 곱하기 명령어*
- 5-78페이지의 *NEON 요소 및 구조체 로드/저장 명령어*
- 5-86페이지의 *NEON 및 VFP 의사 명령어*
- 5-93페이지의 *NEON 및 VFP 시스템 레지스터*

- 5-98페이지의 *0으로 플러시 모드*
- 5-100페이지의 *VFP 명령어*
- 5-110페이지의 *VFP 벡터 모드*

5.1 명령어 요약

이 단원에서는 NEON 및 VFP 명령어에 대해 간략히 설명합니다. 이 단원에 나와 있는 표를 참조하면 나중에 나오는 개별 명령어와 의사 명령어를 쉽게 찾을 수 있습니다. 다음 내용이 포함됩니다.

- *NEON 명령어*
- 5-7페이지의 *공유 NEON 및 VFP 명령어*
- 5-8페이지의 *VFP 명령어*

5.1.1 NEON 명령어

표 5-1에서는 NEON 명령어를 간략하게 보여 줍니다. 이러한 명령어는 VFP에서 사용할 수 없습니다.

표 5-1 NEON 명령어 위치

니모닉	간단한 설명	페이지
VABA, VABD	절대 차이, 절대 차이 및 누산	5-61페이지
VABS	절대값	5-62페이지
VACGE, VACGT	절대 비교 크거나 같음, 보다 큼	5-37페이지
VACLE, VACLT	절대 비교 작거나 같음, 보다 작음 (의사 명령어)	5-91페이지
VADD	더하기	5-63페이지
VADDHN	더하기, 상위 반 선택	5-64페이지
VAND	비트 단위 AND	5-32페이지
VAND	비트 단위 AND (의사 명령어)	5-90페이지
VBIC	비트 단위 비트 지우기 (레지스터)	5-32페이지
VBIC	비트 단위 비트 지우기 (즉치값)	5-33페이지
VBIF, VBIT, VBSL	False인 경우 비트 단위 삽입, True인 경우 삽입 및 선택	5-35페이지
VCEQ, VCLE, VCLT	비교 같음, 작거나 같음, 비교 보다 작음	5-38페이지
VCLE, VCLT	비교 작거나 같음, 비교 보다 작음 (의사 명령어)	5-92페이지
VCLS, VCLZ, VCNT	선행 부호 비트 계산, 선행 0 수 계산 및 세트 비트 계산	5-69페이지

표 5-1 NEON 명령어 위치 (계속)

니모닉	간단한 설명	페이지
VCVT	고정 소수점 또는 정수를 부동 소수점으로 변환, 부동 소수점을 정수 또는 고정 소수점으로 변환	5-42페이지
VCVT	반정밀도 부동 소수점 숫자와 단정밀도 부동 소수점 숫자 간에 변환	5-43페이지
VDUP	스칼라를 벡터의 모든 레인으로 복제	5-44페이지
VEXT	추출	5-45페이지
VCGE, VCGT	비교 크거나 같음, 보다 큼	5-38페이지
VEOR	비트 단위 배타적 OR	5-32페이지
VHADD, VHSUB	양분 더하기, 양분 빼기	5-65페이지
VMAX, VMIN	최대값, 최소값	5-68페이지
VLD	벡터 로드	5-78페이지
VMLA, VMLS	곱하기 누산, 곱하기 빼기 (벡터)	5-74페이지
VMLA, VMLS	곱하기 누산, 곱하기 빼기 (스칼라 기준)	5-75페이지
VMOV	이동 (즉치값)	5-46페이지
VMOV	이동 (레지스터)	5-36페이지
VMOVL, VMOV{U}N	Long 이동, Narrow 이동 (레지스터)	5-47페이지
VMUL	곱하기 (벡터)	5-74페이지
VMUL	곱하기 (스칼라 기준)	5-75페이지
VMVN	음수 이동 (즉치값)	5-46페이지
VNEG	부정	5-62페이지
VORN	비트 단위 OR NOT	5-32페이지
VORN	비트 단위 OR NOT (의사 명령어)	5-90페이지
VORR	비트 단위 OR (레지스터)	5-32페이지
VORR	비트 단위 OR (즉치값)	5-33페이지
VPADD, VPADAL	인접 쌍 더하기, 인접 쌍 더하기 및 누산	5-66페이지

표 5-1 NEON 명령어 위치 (계속)

니모닉	간단한 설명	페이지
VPMAX, VPMIN	인접 쌍 최대값, 인접 쌍 최소값	5-68페이지
VQABS	절대값, 포화	5-62페이지
VQADD	더하기, 포화	5-63페이지
VQDMLAL, VQDMLSL	포화 배수화 곱하기 누산 및 곱하기 빼기	5-76페이지
VQMOV{U}N	포화 이동 (레지스터)	5-47페이지
VQDMUL	포화 배수화 곱하기	5-76페이지
VQDMULH	포화 배수화 곱하기 상위 반 반환	5-77페이지
VQNEG	부정, 포화	5-62페이지
VQRDMULH	포화 배수화 곱하기 상위 반 반환	5-77페이지
VQRSHL	왼쪽으로 시프트, 반올림, 포화 (부호 있는 변수 기준)	5-56페이지
VQRSHR	오른쪽으로 시프트, 반올림, 포화 (즉치값 기준)	5-58페이지
VQSHL	왼쪽으로 시프트, 포화 (즉치값 기준)	5-54페이지
VQSHL	왼쪽으로 시프트, 포화 (부호 있는 변수 기준)	5-56페이지
VQSHR	오른쪽으로 시프트, 포화 (즉치값 기준)	5-58페이지
VQSUB	빼기, 포화	5-63페이지
VRADDH	더하기, 상위 반 선택, 반올림	5-64페이지
VRECPE	역수 추정	5-70페이지
VRECPS	역수 단계	5-71페이지
VREV	요소 반전	5-48페이지
VRHADD	양분 더하기, 반올림	5-65페이지
VRSHR, VRSRA	오른쪽으로 시프트와 반올림, 오른쪽으로 시프트, 반올림 및 누산 (즉치값 기준)	5-57페이지
VRSUBH	빼기, 상위 반 선택, 반올림	5-64페이지
VRSQRTE	역수 제곱근 추정	5-70페이지

표 5-1 NEON 명령어 위치 (계속)

니모닉	간단한 설명	페이지
VRSQRTS	역수 제곱근 단계	5-71페이지
VSHL	왼쪽으로 시프트 (즉치값 기준)	5-54페이지
VSHR	오른쪽으로 시프트 (즉치값 기준)	5-57페이지
VSLI	왼쪽으로 시프트 및 삽입	5-59페이지
VSRA	오른쪽으로 시프트, 누산 (즉치값 기준)	5-57페이지
VSRI	오른쪽으로 시프트 및 삽입	5-59페이지
VST	벡터 저장	5-78페이지
VSUB	빼기	5-63페이지
VSUBH	빼기, 상위 반 선택	5-64페이지
VSWP	벡터 스왑	5-49페이지
VTBL, VTBX	벡터 테이블 조회	5-50페이지
VTST	비트 테스트	5-40페이지
VTRN	벡터 이항	5-51페이지
VUZP, VZIP	벡터 인터리브 및 디인터리브	5-52페이지

5.1.2 공유 NEON 및 VFP 명령어

표 5-2에서는 NEON과 VFP에서 공통적으로 사용되는 명령어를 간략하게 보여줍니다.

표 5-2 공유 NEON 및 VFP 명령어 위치

니모닉	간단한 설명	페이지	피연산자	아키텍처
VLDM	다중 로드	5-25페이지	-	모두
VLDR	로드 (5-87페이지의 *VLDR 의사 명령어* 참조)	5-24페이지	스칼라	모두
	로드 (사후 증가 및 사전 감소)	5-88페이지	스칼라	모두
VMOV	하나의 ARM□ 레지스터에서 더블워드 레지스터의 반으로 전송	5-28페이지	스칼라	모두
	두 개의 ARM 레지스터에서 더블워드 레지스터로 전송	5-27페이지	스칼라	VFPv2
	더블워드 레지스터의 반에서 ARM 레지스터로 전송	5-28페이지	스칼라	모두
	더블워드 레지스터에서 두 개의 ARM 레지스터로 전송	5-27페이지	스칼라	VFPv2
	단정밀도에서 ARM 레지스터로 전송	5-29페이지	스칼라	모두
	ARM 레지스터에서 단정밀도로 전송	5-29페이지	스칼라	모두
VMRS	NEON 및 VFP 시스템 레지스터에서 ARM 레지스터로 전송	5-30페이지	-	모두
VMSR	ARM 레지스터에서 NEON 및 VFP 시스템 레지스터로 전송	5-30페이지	-	모두
VSTM	다중 저장	5-25페이지	-	모두
VSTR	저장	5-24페이지	스칼라	모두
	저장 (사후 증가 및 사전 감소)	5-88페이지	스칼라	모두

5.1.3 VFP 명령어

표 5-3에서는 NEON에서 사용할 수 없는 VFP 명령어를 간략하게 보여 줍니다.

표 5-3 VFP 명령어 위치

니모닉	간단한 설명	페이지	피연산자	아키텍처
VABS	절대값	5-101페이지	벡터	모두
VADD	더하기	5-102페이지	벡터	모두
VCMP	비교	5-104페이지	스칼라	모두
VCVT	단정밀도와 배정밀도 간 변환	5-105페이지	스칼라	모두
	부동 소수점과 정수 간 변환	5-106페이지	스칼라	모두
	부동 소수점과 고정 소수점 간 변환	5-107페이지	스칼라	VFPv3
VCVTB, VCVTT	반정밀도 부동 소수점과 단정밀도 부동 소수점 간에 변환	5-108페이지	스칼라	반 정밀도
VDIV	나누기	5-102페이지	벡터	모두
VMLA	곱하기 누산	5-103페이지	벡터	모두
VMLS	곱하기 빼기	5-103페이지	벡터	모두
VMOV	부동 소수점 상수를 단정밀도 또는 배정밀도 레지스터에 삽입 (5-7페이지의 표 5-2 참조)	5-109페이지	스칼라	VFPv3
VMUL	곱하기	5-103페이지	벡터	모두
VNEG	부정	5-101페이지	벡터	모두
VNMLA	부정 곱하기 누산	5-103페이지	벡터	모두
VNMLS	부정 곱하기 빼기	5-103페이지	벡터	모두
VNMUL	부정 곱하기	5-103페이지	벡터	모두
VSQRT	제곱근	5-101페이지	벡터	모두
VSUB	빼기	5-102페이지	벡터	모두

5.2 NEON 및 VFP에 대한 아키텍처 지원

NEON 확장은 ARMv7-A 및 ARMv7-R 아키텍처에서만 선택적으로 사용할 수 있습니다. 반정밀도 명령어를 제외한 모든 NEON 명령어는 NEON을 지원하는 시스템에서 사용할 수 있습니다. 이러한 명령어 중 일부는 NEON 없이 VFP 확장을 구현하는 시스템에서도 사용할 수 있습니다. 이러한 명령어를 공유 명령어라고 합니다.

반정밀도 명령어는 반정밀도 확장을 구현하는 NEON 또는 VFPv3 시스템에서만 사용할 수 있습니다 (*반정밀도 확장* 참조).

대부분의 VFP 및 공유 명령어는 모든 버전의 VFP 아키텍처에서 사용할 수 있습니다. 그렇지 않은 경우에는 명령어 설명에 따라 적용 가능한 VFP 아키텍처 버전이 지정됩니다.

ARMv7-M에서는 VFP를 지원하지 않습니다. 그 외 모든 ARMv7 아키텍처 프로필은 VFPv4 아키텍처를 지원합니다.

VFPv3에는 모든 VFPv3 레지스터와 부동 소수점 데이터 형식을 지원하지 않는 변형이 있습니다. 구현되는 VFP 아키텍처 및 변형에 대한 자세한 내용을 보려면 항상 해당 제품 설명서를 참조해야 합니다.

NEON 및 VFP 명령어 (반정밀도 명령어 포함) 는 필요한 아키텍처 확장을 지원하지 않는 시스템에서는 정의되지 않은 명령어로 간주됩니다. 그리고 NEON 및 VFP를 지원하는 시스템에서도 보조 프로세서 액세스 제어 레지스터 (CP15 CPACR) 에서 필요한 보조 프로세서가 활성화되어 있지 않으면 명령어가 정의되지 않습니다. 자세한 내용은 해당 프로세서의 기술 참조 문서를 참조하십시오.

5.2.1 반정밀도 확장

반정밀도 확장은 VFPv3 및 NEON 아키텍처를 모두 확장하는 선택적 아키텍처로, 단정밀도 (32비트) 및 반정밀도 (16비트) 부동 소수점 숫자 간에 변환을 수행하는 VFP 및 NEON 명령어를 제공합니다.

5.3 확장 레지스터 뱅크

NEON 및 VFP에서는 동일한 확장 레지스터 뱅크를 사용합니다. 이 레지스터 뱅크는 ARM 레지스터 뱅크와는 다릅니다.

VFP 보조 프로세서에는 32개의 단정밀도 레지스터가 있으며, 각 레지스터에는 단정밀도 부동 소수점 값이나 32비트 정수가 포함될 수 있습니다.

이러한 32개의 레지스터는 16개의 배정밀도 레지스터로 간주되기도 합니다. 그러나 일부 VFPv3 변형의 경우에는 VFP 레지스터 세트에 16개의 배정밀도 레지스터가 더 추가됩니다. 이러한 레지스터는 단정밀도 VFP 레지스터와 겹치지 않습니다.

확장 레지스터 뱅크는 다음 단원에 설명된 대로 명시적으로 별칭이 지정된 세 개의 뷰를 사용하여 참조할 수 있습니다.

5-12페이지의 그림 5-1에서는 세 개의 확장 레지스터 뱅크 뷰를 보여 주고, 워드, 더블워드 및 쿼드워드 레지스터가 겹치는 방식을 보여 줍니다.

참고

프로세서에 NEON과 VFP가 둘 다 있으면 모든 NEON 레지스터가 VFP 레지스터와 겹칩니다.

다음과 같이 사용할 수 있습니다.

- 동시에 단정밀도 값, 배정밀도 값 및 NEON 벡터에 각기 다른 레지스터를 사용합니다.
- 서로 다른 시간에 단정밀도 값, 배정밀도 값 및 NEON 벡터에 동일한 레지스터를 사용합니다.

상응하는 단정밀도와 배정밀도 레지스터를 동시에 사용하면 안 됩니다. 이 경우 손상이 발생하지는 않지만 결과가 무의미해 집니다.

5.3.1 레지스터 뱅크의 NEON 뷰

NEON에서는 확장 레지스터 뱅크를 다음과 같이 표시할 수 있습니다.

- 16개의 128비트 쿼드워드 레지스터, Q0-Q15
- 32개의 64비트 더블워드 레지스터, D0-D31
- 위의 두 뷰에 있는 레지스터의 조합

NEON은 각 레지스터가 크기와 유형이 모두 같은 1, 2, 4, 8 또는 16요소의 *벡터*를 포함하고 있는 것으로 표시합니다. 개별 요소는 *스칼라*로 액세스할 수도 있습니다.

5.3.2 확장 레지스터 뱅크의 VFP 표시

VFPv3 및 VFPv3_fp16에서는 확장 레지스터 뱅크를 다음과 같이 표시할 수 있습니다.

- 32개의 64비트 더블워드 레지스터, D0-D31
- 32개의 32비트 단일 워드 레지스터, S0-S31. 이 뷰에서는 레지스터 뱅크의 반에만 액세스할 수 있습니다.
- 위의 두 뷰에 있는 레지스터의 조합

VFPv2, VFPv3-D16 및 VFPv3-D16_fp16에서는 확장 레지스터 뱅크를 다음과 같이 표시할 수 있습니다.

- 16개의 64비트 더블워드 레지스터, D0-D15
- 32개의 32비트 단일 워드 레지스터, S0-S31
- 위의 두 뷰에 있는 레지스터의 조합

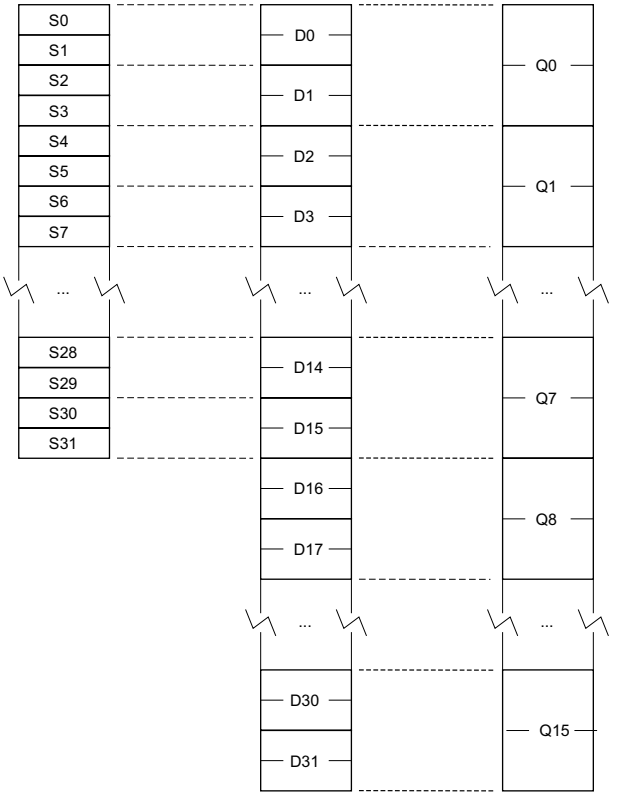

그림 5-1 확장 레지스터 뱅크

레지스터 간 매핑은 다음과 같습니다.

- S<2n>이 D<n>의 최하위 반에 매핑됩니다.
- S<2n+1>이 D<n>의 최상위 반에 매핑됩니다.
- D<2n>이 Q<n>의 최하위 반에 매핑됩니다.
- D<2n+1>이 Q<n>의 최상위 반에 매핑됩니다.

예를 들어 D12를 참조하여 Q6에서 벡터 요소의 최하위 반에 액세스하고 D13을 참조하여 벡터 요소의 최상위 반에 액세스할 수 있습니다.

5.4 조건 코드

ARM 상태에서는 조건 코드를 사용하여 VFP 명령어 실행을 제어할 수 있습니다. 이 명령어는 APSR의 상태 플래그에 따라 대부분의 다른 ARM 명령어와 마찬가지로 조건부로 실행됩니다.

ARM 상태에서는 VFP와 NEON에 공통적인 명령어를 제외하고 조건 코드를 사용하여 NEON 명령어 실행을 제어할 수 없습니다.

Thumb-2 프로세서의 Thumb® 상태에서는 IT 명령어를 사용하여 다음에 나오는 최대 네 개의 NEON 또는 VFP 명령어에 조건 코드를 설정할 수 있습니다. 자세한 내용은 4-118페이지의 *IT*를 참조하십시오.

상태 플래그 업데이트에 사용할 수 있는 유일한 VFP 명령어는 VCMP입니다. 이 명령어는 APSR에서 플래그를 직접 업데이트하지 않지만 FPSCR에서 별도의 플래그 세트를 업데이트합니다 (5-93페이지의 *FPSCR, 부동 소수점 상태 및 제어 레지스터* 참조).

참고

이러한 플래그를 사용하여 조건부 VFP 명령어와 같은 조건부 명령어를 제어하려면 먼저 VMRS 명령어를 사용하여 플래그를 APSR에 복사해야 합니다 (5-30페이지의 *VMRS 및 VMSR* 참조).

플래그의 정확한 의미는 해당 플래그가 VCMP 명령어 뒤에 나오는지, ARM 데이터 처리 명령어 뒤에 나오는지에 따라 달라집니다. 이것은 다음과 같은 이유 때문입니다.

- 부동 소수점 값에 항상 부호가 지정되기 때문에 부호 없는 조건이 필요하지 않습니다.
- *Not-a-Number* (NaN) 값은 숫자 또는 다른 NaN 값과 순서상의 관계가 없기 때문에 *순서가 지정되지 않은* 결과를 나타내기 위해 추가 조건이 필요합니다.

5-14페이지의 표 5-4에는 조건 코드 니모닉의 의미가 나와 있습니다.

표 5-4 조건 코드

니모닉	ARM 데이터 처리 명령어 다음에 나올 경우의 의미	VFP VCMP 명령어 다음에 나올 경우의 의미
EQ	같음	같음
NE	같지 않음	같지 않음 또는 순서가 지정되지 않음
CS or HS	carry 설정 또는 부호 없는 높거나 같음	크거나 같음 또는 순서가 지정되지 않음
CC or LO	carry 지우기 또는 부호 없는 낮음	보다 작음
MI	음수	보다 작음
PL	양수 또는 0	크거나 같음 또는 순서가 지정되지 않음
VS	오버플로	순서가 지정되지 않음 (하나 이상의 NaN 피연산자)
VC	오버플로 없음	순서가 지정됨
HI	부호 없는 높음	보다 큼 또는 순서가 지정되지 않음
LS	부호 없는 낮거나 같음	작거나 같음
GE	부호 있으면서 크거나 같음	크거나 같음
LT	부호 있으면서 보다 작음	보다 작음 또는 순서가 지정되지 않음
GT	부호 있으면서 보다 큼	보다 큼
LE	부호 있으면서 작거나 같음	작거나 같음 또는 순서가 지정되지 않음
AL	항상 (대개 생략됨)	항상 (대개 생략됨)

참고

조건 코드의 의미는 APSR에서 플래그를 마지막으로 업데이트한 명령어의 유형에 따라 달라집니다.

5.5 일반 정보

이 단원에서는 중복을 피하기 위해 여러 명령어에 공통된 정보를 함께 모아 제공합니다. 이 단원에는 다음 소단원이 포함되어 있습니다.

- *부동 소수점 예외*
- 5-16페이지의 *NEON 및 VFP 데이터 형식*
- 5-18페이지의 *NEON의 Normal, Long, Wide, Narrow 및 포화 명령어*
- 5-20페이지의 *NEON 스칼라*
- 5-20페이지의 *확장 표시*
- 5-21페이지의 *{0,1}을 통한 다항식 산술*
- 5-22페이지의 *VFP 보조 프로세서*

5.5.1 부동 소수점 예외

부동 소수점 예외를 발생시킬 수 있는 명령어의 설명 부분에는 예외가 나열된 소단원이 있지만 부동 소수점 예외를 발생시킬 수 없는 명령어의 설명 부분에는 부동 소수점 예외 소단원이 없습니다.

5.5.2 NEON 및 VFP 데이터 형식

NEON 및 VFP 명령어의 데이터 형식 지정자는 데이터 형식을 나타내는 문자로 구성되며, 대개 너비를 나타내는 숫자가 뒤에 옵니다. 이러한 유형 지정자와 명령어 니모닉은 마침표로 구분됩니다. 표 5-5에서는 NEON 명령어에 사용할 수 있는 데이터 형식을 보여 주고, 표 5-6에서는 VFP 명령어에 사용할 수 있는 데이터 형식을 보여 줍니다.

표 5-5 NEON 데이터 유형

	8비트	16비트	32비트	64비트
부호 없는 정수	U8	U16	U32	U64
부호 있는 정수	S8	S16	S32	S64
유형이 지정되지 않은 정수	I8	I16	I32	I64
부동 소수점 숫자	사용할 수 없음	F16	F32 (또는 F)	사용할 수 없음
{0,1}을 통한 다항식	P8	P16	사용할 수 없음	사용할 수 없음

표 5-6 VFP 데이터 유형

	16비트	32비트	64비트
부호 없는 정수	U16	U32	사용할 수 없음
부호 있는 정수	S16	S32	사용할 수 없음
부동 소수점 숫자	F16	F32 (또는 F)	F64 (또는 D)

{0,1}을 통한 다항식 연산에 대한 자세한 내용은 5-21페이지의 *{0,1}을 통한 다항식 산술*을 참조하십시오.

두 번째 (또는 유일한) 피연산자의 데이터 유형은 명령어에서 지정됩니다.

참고

- 대부분의 명령어는 사용할 수 있는 데이터 형식이 제한되어 있습니다. 자세한 내용은 명령어 페이지를 참조하십시오. 그러나 데이터 형식 설명은 다음과 같이 유동적입니다.
  - 설명에서 I를 지정하면 S 또는 U 데이터 형식도 사용할 수 있습니다.
  - 데이터 크기만 지정되어 있으면 형식 (I, S, U, P, F) 을 지정할 수 있습니다.
  - 데이터 형식이 지정되어 있지 않으면 직접 지정할 수 있습니다.
- F16 데이터 형식은 반정밀도 아키텍처 확장을 구현하는 시스템에서만 사용할 수 있습니다.

5.5.3 NEON의 Normal, Long, Wide, Narrow 및 포화 명령어

대부분의 NEON 데이터 처리 명령어는 Normal, Long, Wide, Narrow 및 포화 변형에서 사용할 수 있습니다.

NEON 명령어는 다음에 대해 작동할 수 있습니다.

- 다음과 같이 구성된 더블워드 벡터
  - 여덟 개의 8비트 요소
  - 네 개의 16비트 요소
  - 두 개의 64비트 요소
  - 하나의 64비트 요소
- 다음과 같이 구성된 쿼드워드 벡터
  - 16개의 8비트 요소
  - 여덟 개의 16비트 요소
  - 네 개의 32비트 요소
  - 두 개의 64비트 요소

Normal 명령어

Normal 명령어는 이러한 벡터 유형에 대해 작동하며 크기와 유형이 피연산자 벡터와 동일한 결과 벡터를 생성할 수 있습니다.

명령어 니모닉 끝에 Q를 추가하여 Normal 명령어의 피연산자와 결과가 모두 쿼드워드가 되도록 지정할 수 있습니다. 이렇게 하면 피연산자나 결과가 쿼드워드가 아닐 경우 어셈블러에서 오류가 발생합니다.

Long 명령어

Long 명령어는 더블워드 벡터 피연산자에 대해 작동하며 쿼드워드 벡터 결과를 생성합니다. 일반적으로 결과 요소는 피연산자 요소 너비의 두 배이고 유형이 모두 동일합니다.

Long 명령어는 명령어 니모닉 끝에 추가된 L을 사용하여 지정합니다.

Wide 명령어

Wide 명령어는 하나씩의 더블워드 벡터 피연산자와 쿼드워드 벡터 피연산자에 대해 작동하며 쿼드워드 벡터 결과를 생성합니다. 결과 및 첫 번째 피연산자 요소는 두 번째 피연산자 요소 너비의 두 배입니다.

Wide 명령어는 명령어 니모닉 끝에 추가된 W를 사용하여 지정합니다.

Narrow 명령어

Narrow 명령어는 쿼드워드 벡터 피연산자에 대해 작동하며 더블워드 벡터 결과를 생성합니다. 일반적으로 결과 요소는 피연산자 요소 너비의 반입니다.

Narrow 명령어는 명령어 니모닉 끝에 추가된 N을 사용하여 지정합니다.

포화 명령어

포화 명령어가 수행하는 작업에 대한 일반적인 설명을 보려면 4-92페이지의 *포화 명령어*를 참조하십시오. NEON 포화 명령어가 포화되는 범위를 보려면 표 5-7를 참조하십시오.

포화 명령어는 V와 명령어 니모닉 사이에 Q 접두사를 사용하여 지정합니다.

표 5-7 NEON 포화 범위

데이터 유형	*x*의 포화 범위
S8	$-2^7 <= x < 2^7$
S16	$-2^{15} <= x < 2^{15}$
S32	$-2^{31} <= x < 2^{31}$
S64	$-2^{63} <= x < 2^{63}$
U8	$0 <= x < 2^8$
U16	$0 <= x < 2^{16}$
U32	$0 <= x < 2^{32}$
U64	$0 <= x < 2^{64}$

5.5.4 NEON 스칼라

일부 NEON 명령어는 벡터와 함께 스칼라에 대해 작동합니다. NEON 스칼라는 8비트, 16비트, 32비트 또는 64비트일 수 있습니다. 곱하기 명령어를 제외하고 스칼라에 액세스하는 명령어는 레지스터 뱅크의 요소에 액세스할 수 있습니다. 명령어 구문이 더블워드 벡터에 대한 인덱스를 사용하여 스칼라를 참조하므로 `Dm[x]`는 *Dm*의 *x*번째 요소입니다.

곱하기 명령어는 16비트 또는 32비트 스칼라만 허용하고 레지스터 뱅크에서 처음 32개의 스칼라에만 액세스할 수 있습니다. 즉, 다음 제한이 적용됩니다.

- 16비트 스칼라는 0 ~ 3 범위의 *x*가 있는 D0 ~ D7 레지스터로 제한됩니다.
- 32비트 스칼라는 0 또는 1의 *x*가 있는 D0 ~ D15 레지스터로 제한됩니다.

5.5.5 확장 표시

어셈블러는 *확장 표시*라고 하는 아키텍처 기반 NEON 및 VFP 어셈블리 구문에 대한 확장을 구현합니다. 이 확장을 사용하면 레지스터 이름에 데이터 유형 정보나 스칼라 인덱스를 포함할 수 있습니다. 이 경우 모든 명령어에 데이터 유형이나 스칼라 인덱스 정보를 포함하지 않아도 됩니다.

레지스터 이름은 다음 중 하나일 수 있습니다.

Untyped 레지스터 이름이 레지스터만 지정하고 레지스터에 포함된 데이터 유형이나 레지스터 내의 특정 스칼라에 대한 인덱스는 지정하지 않습니다.

Untyped with scalar index

레지스터 이름이 레지스터와 레지스터 내의 특정 스칼라에 대한 인덱스만 지정하고 레지스터에 포함된 데이터 유형은 지정하지 않습니다.

Typed 레지스터 이름이 레지스터와 레지스터에 포함된 데이터 유형만 지정하고 레지스터 내의 특정 스칼라에 대한 인덱스는 지정하지 않습니다.

Typed with scalar index

레지스터 이름이 레지스터, 레지스터에 포함된 데이터 유형 및 레지스터 내의 특정 스칼라에 대한 인덱스를 지정합니다.

`SN`, `DN` 및 `QN` 지시어를 사용하여 유형화된 레지스터와 스칼라 레지스터를 만들 수 있습니다. 자세한 내용은 7-14페이지의 *QN, DN 및 SN*을 참조하십시오.

5.5.6 {0,1}을 통한 다항식 산술

계수 0 및 1은 다음과 같은 부울 산술 규칙을 사용하여 조작합니다.

- 0 + 0 = 1 + 1 = 0
- 0 + 1 = 1 + 0 = 1
- 0 * 0 = 0 * 1 = 1 * 0 = 0
- 1 * 1 = 1

즉, {0,1}을 통해 두 개의 다항식을 더하는 연산은 비트 단위 배타적 OR과 같고, {0,1}을 통해 두 개의 다항식을 곱하는 연산은 정수 곱하기와 같습니다. 단, 부분 곱은 더하기가 아닌 배타적 OR과 같습니다.

5.5.7 VFP 보조 프로세서

VFP 보조 프로세서는 연관된 지원 코드와 함께 *ANSI/IEEE Std. 754-1985 IEEE Standard for Binary Floating-Point Arithmetic*에 정의된 단정밀도 및 배정밀도 부동 소수점 산술을 제공합니다. 이 장에서는 이 문서를 IEEE 754 표준이라고 합니다. 자세한 내용은 *라이브러리 설명서*에서 4장 *부동 소수점 지원*을 참조하십시오.

최대 여덟 개의 단정밀도 또는 네 개의 배정밀도 숫자의 짧은 벡터를 사용할 수도 있지만 이 옵션은 향후 제공되지 않을 예정입니다. 자세한 내용은 5-110페이지의 *VFP 벡터 모드*를 참조하십시오.

5.6 NEON 및 VFP 공유 명령어

이 단원에는 다음 소단원이 포함되어 있습니다.

- 5-24페이지의 *VLDR 및 VSTR*
  확장 레지스터 로드 및 저장
- 5-25페이지의 *VLDM, VSTM, VPOP 및 VPUSH*
  확장 레지스터 다중 로드 및 저장
- 5-27페이지의 *VMOV (두 개의 ARM 레지스터와 확장 레지스터 간 전송)*
  두 개의 ARM 레지스터와 64비트 확장 레지스터 간에 내용을 전송합니다.
- 5-28페이지의 *VMOV (ARM 레지스터와 NEON 스칼라 간 전송)*
  ARM 레지스터와 4비트 확장 레지스터의 반 간에 내용을 전송합니다.
- 5-29페이지의 *VMOV (한 ARM 레지스터와 단정밀도 VFP 간 전송)*
  32비트 확장 레지스터와 ARM 레지스터 간에 내용을 전송합니다.
- 5-30페이지의 *VMRS 및 VMSR*
  ARM 레지스터와 NEON 및 VFP 시스템 레지스터 간에 내용을 전송합니다.

5.6.1 VLDR 및 VSTR

확장 레지스터 로드 및 저장

구문

VLDR{*cond*}{*.size*} *Fd*, [*Rn*{, *#offset*}]

VSTR{*cond*}{*.size*} *Fd*, [*Rn*{, *#offset*}]

VLDR{*cond*}{*.size*} *Fd*, *label*

VSTR{*cond*}{*.size*} *Fd*, *label*

인수 설명:

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

size 선택적 데이터 크기 지정자입니다. *Fd*가 단정밀도 VFP 레지스터이면 32여야 하고, 그렇지 않으면 64여야 합니다.

Fd 로드 또는 저장할 확장 레지스터로, NEON 명령어의 경우 D 레지스터여야 하고 VFP 명령어의 경우 D 또는 S 레지스터 중 하나일 수 있습니다.

Rn 전송할 기본 주소가 들어 있는 ARM 레지스터입니다.

offset 선택적 숫자 식으로, 어셈블리 타임에 숫자 상수로 평가되어야 합니다. 이 값은 –1020 ~ +1020 범위에 있어야 하며 4의 배수여야 합니다. 기본 주소에 이 값을 더하면 전송에 사용되는 주소가 됩니다.

label 프로그램 기준 식입니다. 자세한 내용은 3-38페이지의 *레지스터 기준 및 프로그램 기준 식*을 참조하십시오.

*label*은 현재 명령어의 1KB 내에 있어야 합니다.

사용법

VLDR 명령어는 메모리에서 확장 레지스터를 로드하고, VSTR 명령어는 확장 레지스터의 내용을 메모리에 저장합니다.

*Fd*가 단정밀도 레지스터이면 하나의 워드가 전송되고 (VFP에만 해당), 그렇지 않으면 두 개의 워드가 전송됩니다.

그 외에도 VLDR 의사 명령어가 있습니다 (5-87페이지의 *VLDR 의사 명령어* 참조).

5.6.2 VLDM, VSTM, VPOP 및 VPUSH

확장 레지스터 다중 로드, 다중 저장, 스택에서 팝 및 스택에 푸시

구문

VLDM*mode*{*cond*} *Rn*{!}, *Registers*

VSTM*mode*{*cond*} *Rn*{!}, *Registers*

VPOP{*cond*} *Registers*

VPUSH{*cond*} *Registers*

인수 설명:

mode 다음 중 하나여야 합니다.

- IA 각 전송 후에 주소를 증가시킵니다. IA는 기본값이며 생략할 수 있습니다.
- DB 각 전송 전에 주소를 감소시킵니다.
- EA 빈 오름차순 스택 연산으로, 로드할 때는 DB와 같고 저장할 때는 IA와 같습니다.
- FD 전체 내림차순 스택 연산으로, 로드할 때는 IA와 같고 저장할 때는 DB와 같습니다.

동등한 주소 지정 모드 접미사는 2-44페이지의 표 2-9를 참조하십시오.

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

Rn 전송할 기본 주소가 들어 있는 ARM 레지스터입니다.

! 선택 사항입니다. ! 기호는 업데이트된 기본 주소를 *Rn*에 다시 기록해야 함을 나타냅니다. ! 기호를 지정하지 않으면 *mode*는 IA여야 합니다.

레지스터 중괄호 ({ 및 }) 로 묶인 연속된 확장 레지스터의 목록입니다. 이 목록은 쉼표로 구분된 형식이거나 범위 형식일 수 있습니다. 이 목록에는 최소한 하나 이상의 레지스터가 있어야 합니다.

S, D 또는 Q 레지스터를 지정할 수 있지만 이들을 함께 지정하면 안 됩니다. 레지스터 수는 D 레지스터의 경우 16개로 제한되고 Q 레지스터의 경우 여덟 개로 제한됩니다. Q 레지스터를 지정하는 경우 이 레지스터는 디스어셈블리에 D 레지스터로 표시됩니다.

참고

VPOP *Registers*는 VLDM sp!, *Registers*와 같습니다.

VPUSH *Registers*는 VSTMDB sp!, *Registers*와 같습니다.

이러한 두 명령어 형식 중 아무 형식이나 사용할 수 있습니다. 이러한 형식은 VPOP 및 VPUSH로 디스어셈블됩니다.

5.6.3 VMOV (두 개의 ARM 레지스터와 확장 레지스터 간 전송)

두 개의 ARM 레지스터와 64비트 확장 레지스터 간 또는 연속하는 두 개의 32비트 VFP 레지스터 간에 내용을 전송합니다.

구문

`VMOV{cond} Dm, Rd, Rn`

`VMOV{cond} Rd, Rn, Dm`

`VMOV{cond} Sm, Sm1, Rd, Rn`

`VMOV{cond} Rd, Rn, Sm, Sm1`

인수 설명:

cond	선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).
Dm	64비트 확장 레지스터입니다.
Sm	VFP 32비트 레지스터입니다.
Sm1	*Sm* 뒤에 나오는 다음 연속 VFP 32비트 레지스터입니다.
Rd, *Rn*	ARM 레지스터입니다. r15를 사용하면 안 됩니다.

사용법

`VMOV Dm, Rd, Rn`은 *Rd*의 내용을 *Dm*의 하위 반으로 전송하고 *Rn*의 내용을 *Dm*의 상위 반으로 전송합니다.

`VMOV Rd, Rn, Dm`은 *Dm*의 하위 반 내용을 *Rd*로 전송하고 *Dm*의 상위 반 내용을 *Rn*으로 전송합니다.

`VMOV Rd, Rn, Sm, Sm1` 은 *Sm*의 내용을 *Rd*로 전송하고 *Sm1*의 내용을 *Rn*으로 전송합니다.

`VMOV Sm, Sm1, Rd, Rn` 은 *Rd*의 내용을 *Sm*으로 전송하고 *Rn*의 내용을 *Sm1*로 전송합니다.

아키텍처

64비트 명령어는 다음 항목에서 사용할 수 있습니다.

- NEON
- VFPv2 이상

2 x 32비트 명령어는 VFPv2 이상에서 사용할 수 있습니다.

5.6.4 VMOV (ARM 레지스터와 NEON 스칼라 간 전송)

ARM 레지스터와 NEON 스칼라 간에 내용을 전송합니다.

구문

VMOV{*cond*}{.*size*} *Dn*[*x*], *Rd*

VMOV{*cond*}{.*datatype*} *Rd*, *Dn*[*x*]

인수 설명:

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

size 데이터 크기입니다. 8, 16 또는 32일 수 있습니다. 생략할 경우 *size*는 32입니다. VFP 명령어의 경우 *size*는 32이거나 생략해야 합니다.

datatype 데이터 형식입니다. U8, S8, U16, S16 또는 32일 수 있습니다. 생략할 경우 *datatype*은 32입니다. VFP 명령어의 경우 *datatype*은 32이거나 생략해야 합니다.

Dn[*x*] NEON 스칼라입니다 (5-20페이지의 *NEON 스칼라* 참조).

Rd ARM 레지스터입니다. *Rd*는 r15이면 안 됩니다.

사용법

VMOV *Rd*, *Dn*[x] 는 *Dn*[x]의 내용을 *Rd*의 최하위 바이트, 하프워드 또는 워드로 전송합니다. 나머지 *Rd* 비트는 0이거나 부호 확장됩니다.

VMOV *Dn*[x], *Rd* 는 *Rd*의 최하위 바이트, 하프워드 또는 워드 내용을 *Sn*으로 전송합니다.

5.6.5 VMOV (한 ARM 레지스터와 단정밀도 VFP 간 전송)

단정밀도 부동 소수점 레지스터와 ARM 레지스터 간에 내용을 전송합니다.

구문

VMOV{*cond*} *Rd*, *Sn*

VMOV{*cond*} *Sn*, *Rd*

인수 설명:

cond	선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).
Sn	VFP 단정밀도 레지스터입니다.
Rd	ARM 레지스터입니다. *Rd*는 r15이면 안 됩니다.

사용법

VMOV *Rd*, *Sn* 은 *Sn*의 내용을 *Rd*로 전송합니다.

VMOV *Sn*, *Rd* 는 *Rd*의 내용을 *Sn*으로 전송합니다.

5.6.6 VMRS 및 VMSR

ARM 레지스터와 NEON 및 VFP 시스템 레지스터 간에 내용을 전송합니다.

구문

VMRS{*cond*} *Rd*, *extsysreg*

VMSR{*cond*} *extsysreg*, *Rd*

인수 설명:

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

extsysreg NEON 및 VFP 시스템 레지스터로, 대개 FPSCR, FPSID 또는 FPEXC입니다 (5-93페이지의 *NEON 및 VFP 시스템 레지스터* 참조).

Rd ARM 레지스터입니다. *Rd* 는 r15이면 안 됩니다.

*extsysreg*가 FPSCR이면 APSR_nzcv일 수 있습니다. 이 경우 부동 소수점 상태 플래그가 ARM CPSR의 해당 플래그로 전송됩니다.

사용법

VMRS 명령어는 *extsysreg*의 내용을 *Rd*로 전송합니다.

VMSR 명령어는 *Rd*의 내용을 *extsysreg*로 전송합니다.

참고

이러한 명령어는 현재 NEON 또는 VFP 연산이 모두 완료될 때까지 ARM을 중단합니다.

예

```
VMRS    r2,FPCID
VMRS    APSR_nzcv, FPSCR     ; transfer FP status register to ARM APSR
VMSR    FPSCR, r4
```

5.7 NEON 논리 및 비교 연산

이 단원에는 다음 소단원이 포함되어 있습니다.

- 5-32페이지의 *VAND, VBIC, VEOR, VORN 및 VORR (레지스터)*
  비트 단위 AND, 비트 지우기, 배타적 OR, OR Not 및 OR (레지스터)
- 5-33페이지의 *VBIC 및 VORR (즉치값)*
  비트 단위 비트 지우기 및 OR (즉치값
- 5-35페이지의 *VBIF, VBIT 및 VBSL*
  False인 경우 비트 단위 삽입, True인 경우 삽입 및 선택
- 5-36페이지의 *VMOV, VMVN (레지스터)*
  이동 및 이동하지 않음
- 5-37페이지의 *VACGE 및 VACGT*
  비교 절대값
- 5-38페이지의 *VCEQ, VCGE, VCGT, VCLE 및 VCLT*
  비교
- 5-40페이지의 *VTST*
  비트 테스트

5.7.1 VAND, VBIC, VEOR, VORN 및 VORR (레지스터)

VAND (비트 단위 AND), VBIC (비트 지우기), VEOR (비트 단위 배타적 OR), VORN (비트 단위 OR NOT) 및 VORR (비트 단위 OR) 명령어는 두 레지스터 간에 비트 단위 논리 연산을 수행하고 결과를 대상 레지스터에 배치합니다.

구문

V*op*{*cond*}.{*datatype*} {*Qd*}, *Qn*, *Qm*

V*op*{*cond*}.{*datatype*} {*Dd*}, *Dn*, *Dm*

인수 설명:

op 다음 중 하나여야 합니다.

- AND 논리 AND
- ORR 논리 OR
- EOR 논리 배타적 OR
- BIC 논리 AND 보수
- ORN 논리 OR 보수

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

datatype 선택적 데이터 유형입니다. 어셈블러에서는 *datatype*을 무시합니다.

Qd, Qn, Qm 쿼드워드 연산에 대한 대상 레지스터, 첫 번째 피연산자 레지스터 및 두 번째 피연산자 레지스터를 지정합니다.

Dd, Dn, Dm 더블워드 연산에 대한 대상 레지스터, 첫 번째 피연산자 레지스터 및 두 번째 피연산자 레지스터를 지정합니다.

참고

두 피연산자 모두에 같은 레지스터를 사용하는 VORR은 VMOV 명령어입니다. 이와 같이 VORR을 사용할 수 있지만 결과 코드를 디스어셈블하면 VMOV 구문이 생성됩니다. 자세한 내용은 5-36페이지의 *VMOV, VMVN (레지스터)*을 참조하십시오.

5.7.2 VBIC 및 VORR (즉치값)

VBIC (비트 지우기 즉치값) 는 대상 벡터의 각 요소를 가져와서 즉치 상수로 비트 단위 AND 보수를 수행하고 결과를 대상 벡터에 반환합니다.

VORR (비트 단위 OR 즉치값) 은 대상 벡터의 각 요소를 가져와서 즉치 상수로 비트 단위 OR을 수행하고 결과를 대상 벡터에 반환합니다.

의사 명령어 5-90페이지의 *VAND 및 VORN (즉치값)*도 참조하십시오.

구문

V*op*{*cond*}.*datatype Qd*, *#imm*

V*op*{*cond*}.*datatype Dd*, *#imm*

인수 설명:

op	BIC 또는 ORR 중 하나여야 합니다.
cond	선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).
datatype	I8, I16, I32 또는 I64 중 하나여야 합니다.
Qd 또는 *Dd*	소스 및 결과의 NEON 레지스터입니다.
imm	즉치 상수입니다.

즉치 상수

*imm*을 어셈블러가 대상 레지스터를 채우기 위해 반복하는 패턴으로 지정할 수도 있고, 전체적으로 패턴을 준수하는 즉치 상수를 직접 지정할 수도 있습니다. *imm* 패턴은 표 5-8에 나와 있는 것처럼 *datatype*에 따라 달라집니다.

표 5-8 즉치 상수 패턴

I16	I32
0x00XY	0x000000XY
0xXY00	0x0000XY00
	0x00XY0000
	0xXY000000

I8 또는 I64 데이터 형식을 사용하는 경우에는 어셈블러가 *imm* 패턴과 일치시키기 위해 해당 데이터 형식을 I16 또는 I32 명령어로 변환합니다. 즉치 상수가 표 5-8의 패턴과 일치하지 않으면 어셈블러에서는 오류를 생성합니다.

5.7.3 VBIF, VBIT 및 VBSL

VBIT (True인 경우 비트 단위 삽입) 는 두 번째 피연산자의 해당 비트가 1이면 첫 번째 피연산자의 각 비트를 대상에 삽입하고, 그렇지 않으면 대상 비트를 변경하지 않습니다.

VBIF (False인 경우 비트 단위 삽입) 는 두 번째 피연산자의 해당 비트가 0이면 첫 번째 피연산자의 각 비트를 대상에 삽입하고, 그렇지 않으면 대상 비트를 변경하지 않습니다.

VBSL (비트 단위 선택) 은 대상의 해당 비트가 1이면 첫 번째 피연산자에서 대상의 각 비트를 선택하고, 대상의 해당 비트가 0이면 두 번째 피연산자에서 선택합니다.

구문

V*op*{*cond*}{.*datatype*} {*Qd*}, *Qn*, *Qm*

V*op*{*cond*}{.*datatype*} {*Dd*}, *Dn*, *Dm*

인수 설명:

op BIT, BIF 또는 BSL 중 하나여야 합니다.

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

datatype 선택적 데이터 유형입니다. 어셈블러에서는 *datatype*을 무시합니다.

Qd, Qn, Qm 쿼드워드 연산에 대한 대상 레지스터, 첫 번째 피연산자 레지스터 및 두 번째 피연산자 레지스터를 지정합니다.

Dd, Dn, Dm 더블워드 연산에 대한 대상 레지스터, 첫 번째 피연산자 레지스터 및 두 번째 피연산자 레지스터를 지정합니다.

5.7.4 VMOV, VMVN (레지스터)

벡터 이동 (레지스터) 은 소스 레지스터에서 대상 레지스터로 값을 복사합니다.

벡터 이동하지 않음 (레지스터) 은 소스 레지스터의 각 비트 값을 반전시키고 결과를 대상 레지스터에 배치합니다.

구문

VMOV{*cond*}{*.datatype*} *Qd*, *Qm*

VMOV{*cond*}{*.datatype*} *Dd*, *Dm*

VMVN{*cond*}{*.datatype*} *Qd*, *Qm*

VMVN{*cond*}{*.datatype*} *Dd*, *Dm*

인수 설명:

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

datatype 선택적 데이터 형식입니다. 어셈블러에서는 *datatype*을 무시합니다.

Qd, Qm 쿼드워드 연산에 대한 대상 벡터 및 소스 벡터를 지정합니다.

Dd, Dm 더블워드 연산에 대한 대상 벡터 및 소스 벡터를 지정합니다.

5.7.5 VACGE 및 VACGT

벡터 절대 비교는 벡터에 있는 각 요소의 절대값을 가져와서 두 번째 벡터의 해당 요소 절대값과 비교합니다. 조건이 true이면 대상 벡터의 해당 요소가 모두 1로 설정되고, 그렇지 않으면 모두 0으로 설정됩니다.

의사 명령어 5-91페이지의 *VACLE 및 VACLT*도 참조하십시오.

구문

VAC*op*{*cond*}.F32 {*Qd*}, *Qn*, *Qm*

VAC*op*{*cond*}.F32 {*Dd*}, *Dn*, *Dm*

인수 설명:

op 다음 중 하나여야 합니다.

- GE 절대 크거나 같음
- GT 절대 보다 큼

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

Qd, Qn, Qm 쿼드워드 연산에 대한 대상 레지스터, 첫 번째 피연산자 레지스터 및 두 번째 피연산자 레지스터를 지정합니다.

Dd, Dn, Dm 더블워드 연산에 대한 대상 레지스터, 첫 번째 피연산자 레지스터 및 두 번째 피연산자 레지스터를 지정합니다.

결과 데이터 유형은 I32입니다.

5.7.6 VCEQ, VCGE, VCGT, VCLE 및 VCLT

벡터 비교는 벡터에 있는 각 요소의 값을 가져와서 두 번째 벡터의 해당 요소 절대값 또는 0과 비교합니다. 조건이 true이면 대상 벡터의 해당 요소가 모두 1로 설정되고, 그렇지 않으면 모두 0으로 설정됩니다.

의사 명령어 5-92페이지의 *VCLE 및 VCLT*도 참조하십시오.

구문

`VC`*op*`{`*cond*`}.`*datatype* `{`*Qd*`}, `*Qn*`, `*Qm*

`VC`*op*`{`*cond*`}.`*datatype* `{`*Dd*`}, `*Dn*`, `*Dm*

`VC`*op*`{`*cond*`}.`*datatype* `{`*Qd*`}, `*Qn*`, #0`

`VC`*op*`{`*cond*`}.`*datatype* `{`*Dd*`}, `*Dn*`, #0`

인수 설명:

op 다음 중 하나여야 합니다.

- `EQ` 같음
- `GE` 크거나 같음
- `GT` 보다 큼
- `LE` 작거나 같음 (두 번째 피연산자가 `#0`인 경우에만 해당)
- `LT` 보다 작음 (두 번째 피연산자가 `#0`인 경우에만 해당)

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

datatype 다음 중 하나여야 합니다.

- `EQ`의 경우 `I8`, `I16`, `I32` 또는 `F32`
- `GE`, `GT`, `LE` 또는 `LT` (`#0` 형식 제외) 의 경우 `S8`, `S16`, `S32`, `U8`, `U16`, `U32` 또는 `F32`
- `GE`, `GT`, `LE` 또는 `LT` (`#0` 형식) 의 경우 `S8`, `S16`, `S32` 또는 `F32`

결과 데이터 유형은 다음과 같습니다.

- 피연산자 데이터 형식 `I32`, `S32`, `U32` 또는 `F32`의 경우 `I32`
- 피연산자 데이터 유형 `I16`, `S16` 또는 `U16`의 경우 `I16`
- `I8`피연산자 데이터 형식 `I8`, `S8` 또는 `U8`의 경우

Qd, Qn, Qm 쿼드워드 연산에 대한 대상 레지스터, 첫 번째 피연산자 레지스터 및 두 번째 피연산자 레지스터를 지정합니다.

Dd, Dn, Dm 더블워드 연산에 대한 대상 레지스터, 첫 번째 피연산자 레지스터 및 두 번째 피연산자 레지스터를 지정합니다.

#0 *Qm* 또는 *Dm*을 비교를 위해 0으로 바꿉니다.

5.7.7 VTST

VTST (벡터 테스트 비트) 는 벡터의 각 요소를 가져와서 두 번째 벡터의 해당 요소로 비트 논리 AND를 수행합니다. 결과가 0이 아니면 대상 벡터의 해당 요소가 모두 1로 설정되고, 그렇지 않으면 모두 0으로 설정됩니다.

구문

VTST{*cond*}.*size* {*Qd*}, *Qn*, *Qm*

VTST{*cond*}.*size* {*Dd*}, *Dn*, *Dm*

인수 설명:

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

size 8, 16 또는 32 중 하나여야 합니다.

Qd, Qn, Qm 쿼드워드 연산에 대한 대상 레지스터, 첫 번째 피연산자 레지스터 및 두 번째 피연산자 레지스터를 지정합니다.

Dd, Dn, Dm 더블워드 연산에 대한 대상 레지스터, 첫 번째 피연산자 레지스터 및 두 번째 피연산자 레지스터를 지정합니다.

5.8 NEON 일반 데이터 처리 명령어

이 단원에는 다음 소단원이 포함되어 있습니다.

- 5-42페이지의 *VCVT (고정 소수점 또는 정수와 부동 소수점 간 변환)*
  고정 소수점 또는 정수와 부동 소수점 간의 벡터 변환
- 5-43페이지의 *VCVT (반정밀도 부동 소수점과 단정밀도 부동 소수점 간에 변환)*
  반정밀도 부동 소수점과 단정밀도 부동 소수점 간의 벡터 변환
- 5-44페이지의 *VDUP*
  스칼라를 벡터의 모든 레인으로 복제
- 5-45페이지의 *VEXT*
  추출
- 5-46페이지의 *VMOV, VMVN (즉치값)*
  이동 및 음수 이동 (즉치값)
- 5-47페이지의 *VMOVL, V{Q}MOVN, VQMOVUN*
  이동 (레지스터)
- 5-48페이지의 *VREV*
  벡터 내 요소 반전
- 5-49페이지의 *VSWP*
  벡터 스왑
- 5-50페이지의 *VTBL, VTBX*
  벡터 테이블 조회
- 5-51페이지의 *VTRN*
  벡터 이항
- 5-52페이지의 *VUZP, VZIP*
  벡터 인터리브 및 디인터리브

5.8.1 VCVT (고정 소수점 또는 정수와 부동 소수점 간 변환)

VCVT (벡터 변환) 는 다음과 같이 벡터의 각 요소를 변환하고 결과를 대상 벡터에 배치합니다.

- 부동 소수점에서 정수로 변환
- 정수에서 부동 소수점으로 변환
- 부동 소수점에서 고정 소수점으로 변환
- 고정 소수점에서 부동 소수점으로 변환

구문

VCVT{*cond*}.*type Qd*, *Qm* {, *#fbits*}

VCVT{*cond*}.*type Dd*, *Dm* {, *#fbits*}

인수 설명:

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

type 벡터 요소의 데이터 형식을 지정합니다. 다음 중 하나여야 합니다.

- S32.F32 부동 소수점을 부호 있는 정수 또는 고정 소수점으로 변환
- U32.F32 부동 소수점을 부호 없는 정수 또는 고정 소수점으로 변환
- F32.S32 부호 있는 정수 또는 고정 소수점을 부동 소수점으로 변환
- F32.U32 부호 없는 정수 또는 고정 소수점을 부동 소수점으로 변환

Qd, Qm 쿼드워드 연산에 대한 대상 벡터 및 피연산자 벡터를 지정합니다.

Dd, Dm 더블워드 연산에 대한 대상 벡터 및 피연산자 벡터를 지정합니다.

fbits 있을 경우 고정 소수점 숫자의 부분 비트 수를 지정합니다. 그렇지 않으면 부동 소수점 숫자와 정수 간에 변환합니다. *fbits*는 0 ~ 32 범위에 있어야 합니다. *fbits*를 생략하면 소수 비트 수는 0입니다.

반올림

정수 또는 고정 소수점을 부동 소수점으로 변환할 때는 가장 가까운 수로 반올림이 사용됩니다.

부동 소수점을 정수 또는 고정 소수점으로 변환할 때는 0으로 반올림이 사용됩니다.

5.8.2 VCVT (반정밀도 부동 소수점과 단정밀도 부동 소수점 간에 변환)

반정밀도 확장을 포함하는 VCVT (벡터 변환) 는 다음과 같이 벡터의 각 요소를 변환하고 결과를 대상 벡터에 배치합니다.

- 반정밀도 부동 소수점에서 단정밀도 부동 소수점으로 변환 (F32.F16)
- 단정밀도 부동 소수점에서 반정밀도 부동 소수점으로 변환 (F16.F32)

구문

VCVT{*cond*}.F32.F16 *Qd*, *Dm*

VCVT{*cond*}.F16.F32 *Dd*, *Qm*

인수 설명:

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

Qd, *Dm* 단정밀도 결과와 반정밀도 피연산자 벡터에 대한 대상 벡터를 지정합니다.

Dd, *Qm* 반정밀도 결과와 단정밀도 피연산자 벡터에 대한 대상 벡터를 지정합니다.

아키텍처

이 명령어는 반정밀도 확장을 포함하는 NEON 시스템에서만 사용할 수 있습니다.

5.8.3 VDUP

VDUP (벡터 복제) 는 스칼라를 대상 벡터의 모든 요소로 복제합니다. 소스는 NEON 스칼라 또는 ARM 레지스터가 될 수 있습니다.

구문

VDUP{*cond*}.*size Qd*, *Dm*[*x*]

VDUP{*cond*}.*size Dd*, *Dm*[*x*]

VDUP{*cond*}.*size Qd*, *Rm*

VDUP{*cond*}.*size Dd*, *Rm*

인수 설명:

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

size 8, 16 또는 32여야 합니다.

Qd 쿼드워드 연산의 대상 레지스터를 지정합니다.

Dd 더블워드 연산의 대상 레지스터를 지정합니다.

Dm[*x*] NEON 스칼라를 지정합니다.

Rm ARM 레지스터를 지정합니다. *Rm*은 pc이면 안 됩니다.

5.8.4 VEXT

VEXT (벡터 추출) 는 두 번째 피연산자 벡터의 아래쪽 끝과 첫 번째 피연산자 벡터의 위쪽 끝에서 8비트 요소를 추출하여 연결하고 결과를 대상 벡터에 배치합니다. 예제를 보려면 그림 5-2를 참조하십시오.

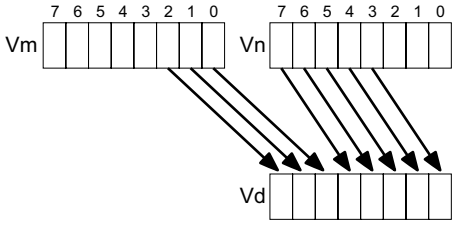

그림 5-2 imm = 3에 대한 더블워드 VEXT 연산

구문

VEXT{*cond*}.8 {*Qd*}, *Qn*, *Qm*, #*imm*

VEXT{*cond*}.8 {*Dd*}, *Dn*, *Dm*, #*imm*

인수 설명:

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

Qd, Qn, Qm 쿼드워드 연산에 대한 대상 레지스터, 첫 번째 피연산자 레지스터 및 두 번째 피연산자 레지스터를 지정합니다.

Dd, Dn, Dm 더블워드 연산에 대한 대상 레지스터, 첫 번째 피연산자 레지스터 및 두 번째 피연산자 레지스터를 지정합니다.

imm 두 번째 피연산자 벡터의 아래쪽 끝에서 추출할 8비트 요소 수로, 더블워드 연산의 경우 0 ~ 7 범위에 있고 쿼드워드 연산의 경우 0 ~ 15 범위에 있습니다.

VEXT 의사 명령어

여덟 개가 아닌 16, 32 또는 64개의 데이터 형식을 지정할 수 있습니다. 이 경우 #imm은 바이트 수가 아니라 하프워드, 워드 또는 더블워드를 나타내고 허용 범위도 이에 따라 줄어듭니다.

5.8.5 VMOV, VMVN (즉치값)

VMOV (벡터 이동) 및 VMVN (벡터 음수 이동) (즉치값) 은 대상 레지스터에 즉치 상수를 생성합니다.

구문

V*op*{*cond*}.*datatype Qd*, #*imm*

V*op*{*cond*}.*datatype Dd*, #*imm*

인수 설명:

op MOV 또는 MVN 중 하나여야 합니다.

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

datatype I8, I16, I32, I64 또는 F32 중 하나여야 합니다.

Qd 또는 *Dd* 결과의 NEON 레지스터입니다.

imm *datatype*에 지정된 유형의 상수입니다. 이 상수는 대상 레지스터를 모두 채울 수 있도록 복제됩니다.

표 5-9 사용할 수 있는 상수

datatype	VMOV	VMVN
I8	0xXY	-
I16	0x00XY, 0xXY00	0xFFXY, 0xXYFF
I32	0x000000XY, 0x0000XY00, 0x00XY0000, 0xXY000000	0xFFFFFFXY, 0xFFFFXYFF, 0xFFXYFFFF, 0xXYFFFFFF
	0x0000XYFF, 0x00XYFFFF	0xFFFFXY00, 0xFFXY0000
I64	바이트 마스크, 0xGGHHJJKKLLMMNNPP [a]	-
F32	부동 소수점 숫자[b]	-

a. 각각의 0xGG, 0xHH, 0xJJ, 0xKK, 0xLL, 0xMM, 0xNN 및 0xPP는 0x00 또는 0xFF 중 하나여야 합니다.

b. +/–n * 2^{-r}로 표시할 수 있는 임의의 숫자입니다. 여기서 n은 16에서 31 사이의 정수이고 r은 0에서 7 사이의 정수입니다.

5.8.6 VMOVL, V{Q}MOVN, VQMOVUN

VMOVL (벡터 이동 Long) 은 더블워드 벡터의 각 요소를 가져와서 원래 길이의 두 배로 부호 또는 0 확장하고 결과를 쿼드워드 벡터에 배치합니다.

VMOVN (벡터 이동 및 Narrow) 은 쿼드워드 벡터의 각 요소 최하위 반을 더블워드 벡터의 해당 요소로 복사합니다.

VQMOVN (벡터 포화 이동 및 Narrow) 은 피연산자 벡터의 각 요소를 대상 벡터의 해당 요소로 복사합니다. 결과 요소는 피연산자 요소 너비의 반이고 값은 결과 너비로 포화됩니다.

VQMOVUN (벡터 포화 이동 및 Narrow, 부호 없는 결과 포함 부호 있는 연산자) 은 피연산자 벡터의 각 요소를 대상 벡터의 해당 요소로 복사합니다. 결과 요소는 피연산자 요소 너비의 반이고 값은 결과 너비로 포화됩니다.

구문

VMOVL{*cond*}.*datatype Qd*, *Dm*

V{Q}MOVN{*cond*}.*datatype Dd*, *Qm*

VQMOVUN{*cond*}.*datatype Dd*, *Qm*

인수 설명:

Q — 있을 경우 결과가 포화됨을 나타냅니다.

cond — 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

datatype — 다음 중 하나여야 합니다.

S8, S16, S32	VMOVL의 경우
U8, U16, U62	VMOVL의 경우
I16, I32, I64	VMOVN의 경우
S16, S32, S64	VQMOVN 또는 VQMOVUN의 경우
U16, U32, U64	VQMOVN의 경우

Qd, Dm — VMOVL에 대한 대상 벡터 및 피연산자 벡터를 지정합니다.

Dd, Qm — V{Q}MOV{U}N에 대한 대상 벡터 및 피연산자 벡터를 지정합니다.

5.8.7 VREV

VREV16 (하프워드 내 벡터 반전) 은 벡터의 각 하프워드 내에서 8비트 요소 순서를 반전시키고 결과를 해당 대상 벡터에 배치합니다.

VREV32 (워드 내 벡터 반전) 는 벡터의 각 워드 내에서 8비트 또는 16비트 요소 순서를 반전시키고 결과를 해당 대상 벡터에 배치합니다.

VREV64 (더블워드 내 벡터 반전) 는 벡터의 각 더블워드 내에서 8비트, 16비트 또는 32비트 요소 순서를 반전시키고 결과를 해당 대상 벡터에 배치합니다.

구문

VREV*n*{*cond*}.*size Qd*, *Qm*

VREV*n*{*cond*}.*size Dd*, *Dm*

인수 설명:

n	16, 32 또는 64 중 하나여야 합니다.
cond	선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).
size	8, 16 또는 32 중 하나여야 하고 *n*보다 작아야 합니다.
Qd, Qm	쿼드워드 연산에 대한 대상 벡터 및 피연산자 벡터를 지정합니다.
Dd, Dm	더블워드 연산에 대한 대상 벡터 및 피연산자 벡터를 지정합니다.

5.8.8 VSWP

VSWP (벡터 스왑) 는 두 벡터의 내용을 교환합니다. 벡터는 더블워드 또는 쿼드워드일 수 있습니다. 데이터 유형에는 차이가 없습니다.

구문

VSWP{*cond*}{*.datatype*} *Qd*, *Qm*

VSWP{*cond*}{*.datatype*} *Dd*, *Dm*

인수 설명:

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

datatype 선택적 데이터 유형입니다. 어셈블러에서는 *datatype*을 무시합니다.

Qd, Qm 쿼드워드 연산의 벡터를 지정합니다.

Dd, Dm 더블워드 연산의 벡터를 지정합니다.

5.8.9 VTBL, VTBX

VTBL (벡터 테이블 조회) 은 제어 벡터에 바이트 인덱스를 사용하여 테이블에서 바이트 값을 조회하고 새 벡터를 생성합니다. 범위를 벗어난 인덱스는 0을 반환합니다.

VTBX (벡터 테이블 확장) 는 범위를 벗어난 인덱스가 대상 요소를 변경하지 않는 것을 제외하고 동일한 방법으로 작동합니다.

구문

```
V{op}{cond}.8 Dd, list, Dm
```

인수 설명:

op TBL 또는 TBX 중 하나여야 합니다.

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

Dd 대상 벡터를 지정합니다.

list 테이블이 포함된 벡터를 지정합니다. 다음 중 하나여야 합니다.

- {*Dn*}
- {*Dn,D (n+1)* }
- {*Dn,D (n+1),D (n+2)* }
- {*Dn,D (n+1),D (n+2),D (n+3)* }

*list*의 모든 레지스터는 D0 ~ D31 범위에 있어야 합니다.

Dm 인덱스 벡터를 지정합니다.

5.8.10 VTRN

VTRN (벡터 이항) 은 피연산자 벡터의 요소를 2 x 2 매트릭스의 요소로 처리하고 매트릭스를 이항합니다. 그림 5-3 및 그림 5-4에서는 VTRN 연산의 예제를 보여 줍니다.

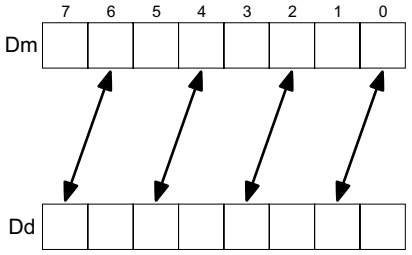

그림 5-3 더블워드 VTRN.8 연산

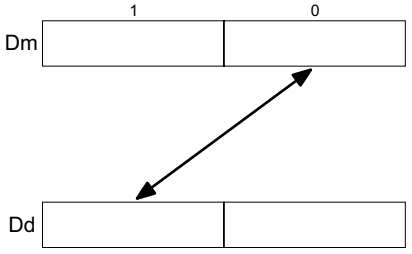

그림 5-4 더블워드 VTRN.32 연산

구문

VTRN{*cond*}.*size Qd*, *Qm*

VTRN{*cond*}.*size Dd*, *Dm*

인수 설명:

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

size 8, 16 또는 32 중 하나여야 합니다.

Qd, *Qm* 쿼드워드 연산의 벡터를 지정합니다.

Dd, *Dm* 더블워드 연산의 벡터를 지정합니다.

5.8.11 VUZP, VZIP

VZIP (벡터 압축) 는 두 벡터의 요소를 인터리브합니다.

VUZP (벡터 압축 해제) 는 두 벡터의 요소를 디인터리브합니다.

디인터리브에 대한 예제를 보려면 5-78페이지의 *3요소 구조체 배열 디인터리브* 를 참조하십시오. 인터리브는 역방향 프로세스입니다.

구문

V*op*{*cond*}.*size Qd*, *Qm*

V*op*{*cond*}.*size Dd*, *Dm*

인수 설명:

op	UZP 또는 ZIP 중 하나여야 합니다.
cond	선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).
size	8, 16 또는 32 중 하나여야 합니다.
Qd, Qm	쿼드워드 연산의 벡터를 지정합니다.
Dd, Dm	더블워드 연산의 벡터를 지정합니다.

참고

다음은 모두 같은 명령어입니다.

- VZIP.32 *Dd*, *Dm*
- VUZP.32 *Dd*, *Dm*
- VTRN.32 *Dd*, *Dm*

명령어는 VTRN.32 *Dd*, *Dm*. 으로 디스어셈블됩니다. 5-51페이지의 *VTRN*도 참조하십시오.

5.9 NEON 시프트 명령어

이 단원에는 다음 소단원이 포함되어 있습니다.

- 5-54페이지의 *VSHL, VQSHL, VQSHLU 및 VSHLL (즉치값 기준)*
  즉치값만큼 왼쪽으로 시프트
- 5-56페이지의 *V{Q}{R}SHL (부호 있는 변수 기준)*
  부호 있는 변수만큼 왼쪽으로 시프트
- 5-57페이지의 *V{R}SHR{N}, V{R}SRA (즉치값 기준)*
  즉치값만큼 오른쪽으로 시프트
- 5-58페이지의 *VQ{R}SHR{U}N (즉치값 기준)*
  즉치값만큼 오른쪽으로 시프트 및 포화
- 5-59페이지의 *VSLI 및 VSRI*
  왼쪽으로 시프트 및 삽입 및 오른쪽으로 시프트 및 삽입

5.9.1 VSHL, VQSHL, VQSHLU 및 VSHLL (즉치값 기준)

벡터 왼쪽으로 시프트 (즉치값 기준) 명령어는 정수 벡터의 각 요소를 가져와서 즉치값만큼 왼쪽으로 시프트하고 결과를 대상 벡터에 배치합니다.

VSHL (벡터 왼쪽으로 시프트) 의 경우 각 요소 왼쪽에서 시프트된 비트는 버려집니다.

VQSHL (벡터 포화 왼쪽으로 시프트) 및 VQSHLU (부호 있는 벡터 포화 왼쪽으로 시프트) 의 경우 포화가 발생하면 스티키 QC 플래그 (FPSCR 비트[27]) 가 설정됩니다.

VSHLL (벡터 왼쪽으로 시프트 Long) 의 경우 값은 부호 확장되거나 0으로 확장됩니다.

구문

V{Q}SHL{U}{*cond*}.*datatype* {*Qd*}, *Qm*, *#imm*

V{Q}SHL{U}{*cond*}.*datatype* {*Dd*}, *Dm*, *#imm*

VSHLL{*cond*}.*datatype Qd*, *Dm*, *#imm*

인수 설명:

Q — 있을 경우 결과가 오버플로되면 포화됨을 나타냅니다.

U — Q도 있는 경우에만 허용됩니다. 피연산자에 부호가 있어도 결과에 부호가 없음을 나타냅니다.

cond — 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

datatype — 다음 중 하나여야 합니다.

I8, I16, I32, I64	VSHL의 경우
S8, S16, S32	VSHLL, VQSHL 또는 VQSHLU의 경우
U8, U16, U32	VSHLL 또는 VQSHL의 경우
S64	VQSHL 또는 VQSHLU의 경우
U64	VQSHL의 경우

Qd, Qm — 쿼드워드 연산에 대한 대상 및 피연산자 벡터입니다.

Dd, Dm — 더블워드 연산에 대한 대상 및 피연산자 벡터입니다.

Qd, Dm — long 연산에 대한 대상 및 피연산자 벡터입니다.

imm 다음 범위에서 시프트 크기를 지정하는 즉치 상수입니다.

- VSHLL의 경우 1 ~ 크기 (*datatype*)
- VSHL, VQSHL 또는 VQSHLU의 경우 1 ~ (크기 (*datatype*) – 1)

0이 허용되지만 결과 코드는 VMOV 또는 VMOVL로 디스어셈블됩니다.

5.9.2 V{Q}{R}SHL (부호 있는 변수 기준)

VSHL (부호 있는 변수 기준 벡터 왼쪽으로 시프트) 은 벡터의 각 요소를 가져와서 두 번째 벡터에 있는 해당 요소의 최하위 바이트 값만큼 시프트하고 결과를 대상 벡터에 배치합니다. 시프트 값이 양수이면 연산이 왼쪽으로 시프트되고, 그렇지 않으면 오른쪽으로 시프트됩니다.

결과는 선택적으로 포화 또는 반올림되거나 포화도 되고 반올림도 될 수 있습니다. 포화가 발생하면 스티키 QC 플래그 (FPSCR bit[27]) 가 설정됩니다.

구문

V{Q}{R}SHL{*cond*}.*datatype* {*Qd*}, *Qm*, *Qn*

V{Q}{R}SHL{*cond*}.*datatype* {*Dd*}, *Dm*, *Dn*

인수 설명:

Q — 있을 경우 결과가 오버플로되면 포화됨을 나타냅니다.

R — 있을 경우 각 결과가 반올림됨을 나타내고, 그렇지 않으면 각 결과가 잘립니다.

cond — 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

datatype — S8, S16, S32, S64, U8, U16, U32 또는 U64 중 하나여야 합니다.

Qd, *Qm*, *Qn* — 쿼드워드 연산에 대한 대상 벡터, 첫 번째 피연산자 벡터 및 두 번째 피연산자 벡터입니다.

Dd, *Dm*, *Dn* — 더블워드 연산에 대한 대상 벡터, 첫 번째 피연산자 벡터 및 두 번째 피연산자 벡터입니다.

5.9.3 V{R}SHR{N}, V{R}SRA (즉치값 기준)

V{R}SHR{N} (즉치값 기준 벡터 오른쪽으로 시프트) 은 벡터의 각 요소를 가져와서 즉치값만큼 오른쪽으로 시프트하고 결과를 대상 벡터에 배치합니다. 결과는 선택적으로 반올림 또는 축소되거나 반올림도 되고 축소도 될 수 있습니다.

V{R}SRA (즉치값 기준 벡터 오른쪽으로 시프트 및 누산) 는 벡터의 각 요소를 가져와서 즉치값만큼 오른쪽으로 시프트하고 결과를 대상 벡터에 더합니다. 결과는 선택적으로 포화되거나 반올림됩니다.

구문

V{R}SHR{*cond*}.*datatype* {*Qd*}, *Qm*, *#imm*

V{R}SHR{*cond*}.*datatype* {*Dd*}, *Dm*, *#imm*

V{R}SRA{*cond*}.*datatype* {*Qd*}, *Qm*, *#imm*

V{R}SRA{*cond*}.*datatype* {*Dd*}, *Dm*, *#imm*

V{R}SHRN{*cond*}.*datatype Dd*, *Qm*, *#imm*

인수 설명:

R — 있을 경우 결과가 반올림됨을 나타내고, 그렇지 않으면 결과가 잘립니다.

cond — 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

datatype — 다음 중 하나여야 합니다.

S8, S16, S32, S64	V{R}SHR 또는 V{R}SRA의 경우
U8, U16, U32, U64	V{R}SHR 또는 V{R}SRA의 경우
I16, I32, I64	V{R}SHRN의 경우

Qd, *Qm* — 쿼드워드 연산에 대한 대상 벡터 및 피연산자 벡터입니다.

Dd, *Dm* — 더블워드 연산에 대한 대상 벡터 및 피연산자 벡터입니다.

Dd, *Qm* — 축소 연산에 대한 대상 벡터 및 피연산자 벡터입니다.

imm — 0 ~ (크기 (*datatype*) – 1) 범위에서 시프트 크기를 지정하는 즉치 상수입니다.

5.9.4 VQ{R}SHR{U}N (즉치값 기준)

VQ{R}SHR{U}N (즉치값 기준 벡터 포화 오른쪽으로 시프트, 축소, 선택적 반올림 포함) 은 정수 쿼드워드 벡터의 각 요소를 가져와서 즉치값만큼 오른쪽으로 시프트하고 결과를 더블워드 벡터에 배치합니다.

포화가 발생하면 스티키 QC 플래그 (FPSCR bit[27]) 가 설정됩니다.

구문

VQ{R}SHR{U}N{*cond*}.*datatype Dd*, *Qm*, #*imm*

인수 설명:

R — 있을 경우 결과가 반올림됨을 나타내고, 그렇지 않으면 결과가 잘립니다.

U — 있을 경우 피연산자에 부호가 있어도 결과에 부호가 없음을 나타내고, 그렇지 않으면 결과는 피연산자와 동일한 유형입니다.

cond — 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

datatype — 다음 중 하나여야 합니다.

S16, S32, S64	VQ{R}SHRN 또는 VQ{R}SHRUN의 경우
U16, U32, U64	VQ{R}SHRN의 경우에만

Dd, *Qm* — 대상 벡터 및 피연산자 벡터입니다.

imm — 0 ~ (크기 (*datatype*) – 1) 범위에서 시프트 크기를 지정하는 즉치 상수입니다.

5.9.5 VSLI 및 VSRI

VSLI (벡터 왼쪽으로 시프트 및 삽입) 는 벡터의 각 요소를 가져와서 즉치값만큼 왼쪽으로 시프트하고 결과를 대상 벡터에 삽입합니다. 각 요소 왼쪽에서 시프트된 비트는 버려집니다.

VSRI (벡터 오른쪽으로 시프트 및 삽입) 는 벡터의 각 요소를 가져와서 즉치값만큼 오른쪽으로 시프트하고 결과를 대상 벡터에 삽입합니다. 각 요소 오른쪽에서 시프트된 비트는 버려집니다.

구문

V*op*{*cond*}.*size* {*Qd*}, *Qm*, #*imm*

V*op*{*cond*}.*size* {*Dd*}, *Dm*, #*imm*

인수 설명:

op SLI 또는 SRI 중 하나여야 합니다.

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

size 8, 16, 32 또는 64 중 하나여야 합니다.

Qd, *Qm* 쿼드워드 연산에 대한 대상 벡터 및 피연산자 벡터입니다.

Dd, *Dm* 더블워드 연산에 대한 대상 벡터 및 피연산자 벡터입니다.

imm 다음 범위에서 시프트 크기를 지정하는 즉치 상수입니다.

- VSLI의 경우 0 ~ (*size* – 1)
- VSRI의 경우 1 ~ *size*

5.10 NEON 일반 산술 명령어

이 단원에는 다음 소단원이 포함되어 있습니다.

- 5-61페이지의 *VABA{L} 및 VABD{L}*
  벡터 절대 차이, 누산 및 절대 차이
- 5-62페이지의 *V{Q}ABS 및 V{Q}NEG*
  벡터 절대값 및 부정
- 5-63페이지의 *V{Q}ADD, VADDL, VADDW, V{Q}SUB, VSUBL 및 VSUBW*
  벡터 더하기 및 빼기
- 5-64페이지의 *V{R}ADDHN 및 V{R}SUBHN*
  벡터 더하기 상위 반 선택 및 빼기 상위 반 선택
- 5-65페이지의 *V{R}HADD 및 VHSUB*
  벡터 양분 더하기 및 빼기
- 5-66페이지의 *VPADD{L}, VPADAL*
  벡터 인접 쌍 더하기, 더하기 및 누산
- 5-68페이지의 *VMAX, VMIN, VPMAX 및 VPMIN*
  벡터 최대값, 최소값, 인접 쌍 최대 값 및 인접 쌍 최소값
- 5-69페이지의 *VCLS, VCLZ 및 VCNT*
  벡터 선행 부호 비트 계산, 선행 0 수 계산 및 세트 비트 계산
- 5-70페이지의 *VRECPE 및 VRSQRTE*
  벡터 역수 추정 및 역수 제곱근 추정
- 5-71페이지의 *VRECPS 및 VRSQRTS*
  벡터 역수 추정 및 역수 제곱근 추정

5.10.1 VABA{L} 및 VABD{L}

VABA (벡터 절대 차이 및 누산) 는 한 벡터 요소를 다른 벡터의 해당 요소에서 빼고 결과의 절대값을 대상 벡터 요소에 더합니다.

VABD (벡터 절대 차이) 는 한 벡터 요소를 다른 벡터의 해당 요소에서 빼고 결과의 절대값을 대상 벡터 요소에 배치합니다.

두 명령어의 Long 버전을 사용할 수 있습니다.

구문

V*op*{*cond*}.*datatype* {*Qd*}, *Qn*, *Qm*

V*op*{*cond*}.*datatype* {*Dd*}, *Dn*, *Dm*

V*op*L{*cond*}.*datatype Qd*, *Dn*, *Dm*

인수 설명:

op ABA 또는 ABD 중 하나여야 합니다.

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

datatype 다음 중 하나여야 합니다.

- VABA, VABAL 또는 VABDL의 경우 S8, S16, S32, U8, U16 또는 U32VABD의 경우
- S8, S16, S32, U8, U16, U32 또는 F32

Qd, *Qn*, *Qm* 쿼드워드 연산에 대한 대상 벡터, 첫 번째 피연산자 벡터 및 두 번째 피연산자 벡터입니다.

Dd, *Dn*, *Dm* 더블워드 연산에 대한 대상 벡터, 첫 번째 피연산자 벡터 및 두 번째 피연산자 벡터입니다.

Qd, *Dn*, *Dm* long 연산에 대한 대상 벡터, 첫 번째 피연산자 벡터 및 두 번째 피연산자 벡터입니다.

5.10.2 V{Q}ABS 및 V{Q}NEG

VABS (벡터 절대값) 는 벡터에 있는 각 요소의 절대값을 구하고 결과를 두 번째 벡터에 배치합니다. 부동 소수점 버전에서만 부호 비트를 지웁니다.

VNEG (벡터 부정) 는 벡터의 각 요소를 부정하고 결과를 두 번째 벡터에 배치합니다. 부동 소수점 버전에서만 부호 비트를 반전시킵니다.

두 명령어의 포화 버전을 사용할 수 있습니다. 포화가 발생하면 스티키 QC 플래그 (FPSCR bit[27]) 가 설정됩니다.

구문

V{Q}*op*{*cond*}.*datatype Qd*, *Qm*

V{Q}*op*{*cond*}.*datatype Dd*, *Dm*

인수 설명:

Q — 있을 경우 결과가 오버플로되면 포화됨을 나타냅니다.

op — ABS 또는 NEG 중 하나여야 합니다.

cond — 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

datatype — 다음 중 하나여야 합니다.

S8, S16, S32	VABS, VNEG, VQABS 또는 VQNEG의 경우
F32	VABS 및 VNEG의 경우에만

Qd, *Qm* — 쿼드워드 연산에 대한 대상 벡터 및 피연산자 벡터입니다.

Dd, *Dm* — 더블워드 연산에 대한 대상 벡터 및 피연산자 벡터입니다.

5.10.3 V{Q}ADD, VADDL, VADDW, V{Q}SUB, VSUBL 및 VSUBW

VADD (벡터 더하기) 는 두 벡터의 해당 요소를 더하고 결과를 대상 벡터에 배치합니다.

VSUB (벡터 빼기) 는 한 벡터 요소를 다른 벡터의 해당 요소에서 빼고 결과를 대상 벡터에 배치합니다.

포화, Long 및 Wide 버전을 사용할 수 있습니다. 포화가 발생하면 스티키 QC 플래그 (FPSCR bit[27]) 가 설정됩니다.

구문

V{Q}*op*{*cond*}.*datatype* {*Qd*}, *Qn*, *Qm*

V{Q}*op*{*cond*}.*datatype* {*Dd*}, *Dn*, *Dm*

V*op*L{*cond*}.*datatype Qd*, *Dn*, *Dm*

V*op*W{*cond*}.*datatype* {*Qd*}, *Qn*, *Dm*

인수 설명:

Q — 있을 경우 결과가 오버플로되면 포화됨을 나타냅니다.

op — ADD 또는 SUB 중 하나여야 합니다.

cond — 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

datatype — 다음 중 하나여야 합니다.

I8, I16, I32, I64, F32	VADD 또는 VSUB의 경우
S8, S16, S32	VQADD, VQSUB, VADDL, VADDW, VSUBL 또는 VSUBW의 경우
U8, U16, U32	VQADD, VQSUB, VADDL, VADDW, VSUBL 또는 VSUBW의 경우
S64, U64	VQADD 또는 VQSUB의 경우

Qd, *Qn*, *Qm* — 쿼드워드 연산에 대한 대상 벡터, 첫 번째 피연산자 벡터 및 두 번째 피연산자 벡터입니다.

Dd, *Dn*, *Dm* — 더블워드 연산에 대한 대상 벡터, 첫 번째 피연산자 벡터 및 두 번째 피연산자 벡터입니다.

Qd, *Dn*, *Dm* — long 연산에 대한 대상 벡터, 첫 번째 피연산자 벡터 및 두 번째 피연산자 벡터입니다.

Qd, *Qn*, *Dm* — Wide 연산에 대한 대상 벡터, 첫 번째 피연산자 벡터 및 두 번째 피연산자 벡터입니다.

5.10.4 V{R}ADDHN 및 V{R}SUBHN

V{R}ADDH (벡터 더하기 및 Narrow, 상위 반 선택) 는 두 벡터의 해당 요소를 더하고 결과의 최상위 반을 선택한 후 최종 결과를 대상 벡터에 배치합니다. 결과는 반올림되거나 잘릴 수 있습니다.

V{R}SUBH (벡터 빼기 및 Narrow, 상위 반 선택) 는 한 벡터 요소를 다른 벡터의 해당 요소에서 빼고 결과의 최상위 반을 선택한 후 최종 결과를 대상 벡터에 배치합니다. 결과는 반올림되거나 잘릴 수 있습니다.

구문

V{R}*op*HN{*cond*}.*datatype Dd*, *Qn*, *Qm*

인수 설명:

R — 있을 경우 각 결과가 반올림됨을 나타내고, 그렇지 않으면 각 결과가 잘립니다.

op — ADD 또는 SUB 중 하나여야 합니다.

cond — 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

datatype — I16, I32 또는 I64 중 하나여야 합니다.

Dd, *Qn*, *Qm* — 대상 벡터, 첫 번째 피연산자 벡터 및 두 번째 피연산자 벡터입니다.

5.10.5 V{R}HADD 및 VHSUB

VHADD (벡터 양분 더하기) 는 두 벡터의 해당 요소를 더하고 각 결과를 1비트 오른쪽으로 시프트한 후 최종 결과를 대상 벡터에 배치합니다. 결과는 반올림되거나 잘릴 수 있습니다.

VHSUB (벡터 양분 빼기) 는 한 벡터 요소를 다른 벡터의 해당 요소에서 빼고 각 결과를 1비트 오른쪽으로 시프트한 후 최종 결과를 대상 벡터에 배치합니다. 결과는 항상 잘립니다.

구문

V{R}HADD{*cond*}.*datatype* {*Qd*}, *Qn*, *Qm*

V{R}HADD{*cond*}.*datatype* {*Dd*}, *Dn*, *Dm*

VHSUB{*cond*}.*datatype* {*Qd*}, *Qn*, *Qm*

VHSUB{*cond*}.*datatype* {*Dd*}, *Dn*, *Dm*

인수 설명:

R 있을 경우 각 결과가 반올림됨을 나타내고, 그렇지 않으면 각 결과가 잘립니다.

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

datatype S8, S16, S32, U8, U16 또는 U32 중 하나여야 합니다.

Qd, *Qn*, *Qm* 쿼드워드 연산에 대한 대상 벡터, 첫 번째 피연산자 벡터 및 두 번째 피연산자 벡터입니다.

Dd, *Dn*, *Dm* 더블워드 연산에 대한 대상 벡터, 첫 번째 피연산자 벡터 및 두 번째 피연산자 벡터입니다.

5.10.6 VPADD{L}, VPADAL

VPADD (벡터 인접 쌍 더하기) 는 두 벡터 요소의 인접 쌍을 더하고 결과를 대상 벡터에 배치합니다.

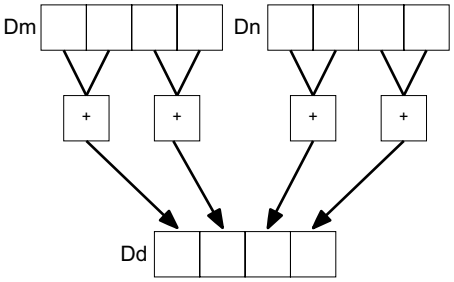

그림 5-5 VPADD 연산 예제 (이 경우 데이터 유형은 I16임)

VPADDL (벡터 인접 쌍 더하기 Long) 은 벡터 요소의 인접 쌍을 더하고 결과를 원래 너비의 두 배로 부호 또는 0 확장한 후 최종 결과를 대상 벡터에 배치합니다.

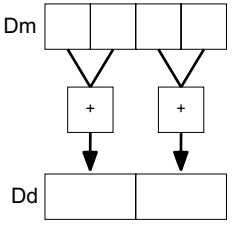

그림 5-6 더블워드 VPADDL 연산 예제 (이 경우 데이터 유형은 S16임)

VPADAL (벡터 인접 쌍 더하기 및 누산 Long) 은 벡터 요소의 인접 쌍을 더하고 결과의 절대 값을 대상 벡터 요소에 더합니다.

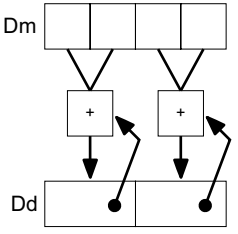

그림 5-7 VPADAL 연산 예제 (이 경우 데이터 유형은 S16임)

구문

VPADD{*cond*}.*datatype* {*Dd*}, *Dn*, *Dm*

VP*op*L{*cond*}.*datatype Qd*, *Qm*

VP*op*L{*cond*}.*datatype Dd*, *Dm*

인수 설명:

op ADD 또는 ADA 중 하나여야 합니다.

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

datatype 다음 중 하나여야 합니다.

I8, I16, I32, F32	VPADD의 경우
S8, S16, S32	VPADDL 또는 VPADAL의 경우
U8, U16, U32	VPADDL 또는 VPADAL의 경우

Dd, *Dn*, *Dm* VPADD 명령어에 대한 대상 벡터, 첫 번째 피연산자 벡터 및 두 번째 피연산자 벡터입니다.

Qd, *Qm* 쿼드워드 VPADDL 또는 VPADAL에 대한 대상 벡터 및 피연산자 벡터입니다.

Dd, *Dm* 더블워드 VPADDL 또는 VPADAL에 대한 대상 벡터 및 피연산자 벡터입니다.

5.10.7 VMAX, VMIN, VPMAX 및 VPMIN

VMAX (벡터 최대값) 는 두 벡터의 해당 요소를 비교하고 각 쌍의 큰 쪽을 대상 벡터의 해당 요소에 복사합니다.

VMIN (벡터 최소값) 은 두 벡터의 해당 요소를 비교하고 각 쌍의 작은 쪽을 대상 벡터의 해당 요소에 복사합니다.

VPMAX (벡터 인접 쌍 최대값) 는 두 벡터 요소의 인접 쌍을 비교하고 각 쌍의 큰 쪽을 대상 벡터의 해당 요소에 복사합니다. 피연산자와 결과는 더블워드 벡터여야 합니다.

VPMIN (벡터 인접 쌍 최소값) 은 두 벡터 요소의 인접 쌍을 비교하고 각 쌍의 작은 쪽을 대상 벡터의 해당 요소에 복사합니다. 피연산자와 결과는 더블워드 벡터여야 합니다.

인접 쌍 연산에 대한 다이어그램을 보려면 5-66페이지의 그림 5-5를 참조하십시오.

구문

V*op*{*cond*}.*datatype Qd*, *Qn*, *Qm*

V*op*{*cond*}.*datatype Dd*, *Dn*, *Dm*

VP*op*{*cond*}.*datatype Dd*, *Dn*, *Dm*

인수 설명:

op MAX 또는 MIN 중 하나여야 합니다.

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

datatype S8, S16, S32, U8, U16, U32 또는 F32 중 하나여야 합니다.

Qd, *Qn*, *Qm* 쿼드워드 연산에 대한 대상 벡터, 첫 번째 피연산자 벡터 및 두 번째 피연산자 벡터입니다.

Dd, *Dn*, *Dm* 더블워드 연산에 대한 대상 벡터, 첫 번째 피연산자 벡터 및 두 번째 피연산자 벡터입니다.

부동 소수점 최대값 및 최소값

max (+0.0, –0.0) = +0.0

min (+0.0, –0.0) = –0.0

입력이 NaN이면 해당 결과 요소는 기본 NaN입니다.

5.10.8 VCLS, VCLZ 및 VCNT

VCLS (벡터 선행 부호 비트 계산) 는 벡터의 각 요소에서 최상위 비트 뒤에 나오는 최상위 비트와 동일한 연속 비트 수를 계산하고 결과를 두 번째 벡터에 배치합니다.

VCLZ (벡터 선행 0 수 계산) 는 벡터의 각 요소에서 상위 비트부터 연속 0 수를 계산하고 결과를 두 번째 벡터에 배치합니다.

VCNT (벡터 세트 비트 계산) 는 벡터의 각 요소 중 하나인 비트 수를 계산하고 결과를 두 번째 벡터에 배치합니다.

구문

V*op*{*cond*}.*datatype Qd*, *Qm*

V*op*{*cond*}.*datatype Dd*, *Dm*

인수 설명:

op CLS, CLZ 또는 CNT 중 하나여야 합니다.

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

datatype 다음 중 하나여야 합니다.

- CLS의 경우 S8, S16 또는 S32
- CLZ의 경우 I8, I16 또는 I32
- CNT의 경우 I8

Qd, *Qm* 쿼드워드 연산에 대한 대상 벡터 및 피연산자 벡터입니다.

Dd, *Dm* 더블워드 연산에 대한 대상 벡터 및 피연산자 벡터입니다.

5.10.9 VRECPE 및 VRSQRTE

VRECPE (벡터 역수 추정) 는 벡터에서 각 요소의 대략적 역수를 찾고 결과를 두 번째 벡터에 배치합니다.

VRSQRTE (벡터 역수 제곱근 추정) 는 벡터에서 각 요소의 대략적 역수 제곱근을 찾고 결과를 두 번째 벡터에 배치합니다.

구문

V*op*{*cond*}.*datatype Qd*, *Qm*

V*op*{*cond*}.*datatype Dd*, *Dm*

인수 설명:

op RECPE 또는 RSQRTE 중 하나여야 합니다.

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

datatype U32 또는 F32 중 하나여야 합니다.

Qd, *Qm* 쿼드워드 연산에 대한 대상 벡터 및 피연산자 벡터입니다.

Dd, *Dm* 더블워드 연산에 대한 대상 벡터 및 피연산자 벡터입니다.

범위를 벗어난 입력 결과

표 5-10에서는 입력 값이 범위를 벗어난 결과를 보여 줍니다.

표 5-10 범위를 벗어난 입력 결과

	피연산자 요소 (VRECPE)	피연산자 요소 (VRSQRTE)	결과 요소
정수	<= 0x7FFFFFFF	<= 0x3FFFFFFF	0xFFFFFFFF
부동 소수점	NaN	NaN, 음의 정규값, 음의 무한대	기본 NaN
	음의 0, 음의 비정규값	음의 0, 음의 비정규값	음의 무한대[a]
	양의 0, 양의 비정규값	양의 0, 양의 비정규값	양의 무한대[a]
	양의 무한대	양의 무한대	양의 0
	음의 무한대		음의 0

a. FPSCR (FPSCR[1]) 에서 0으로 나눔 예외 비트가 설정됩니다.

5.10.10 VRECPS 및 VRSQRTS

VRECPS (벡터 역수 단계) 는 한 벡터 요소에 다른 벡터의 해당 요소를 곱하고 2에서 각 결과를 뺀 후 최종 결과를 대상 벡터의 요소에 배치합니다.

VRSQRTS (벡터 역수 제곱근 단계) 는 한 벡터 요소에 다른 벡터의 해당 요소를 곱하고 3에서 각 결과를 빼고 2로 나눈 후 최종 결과를 대상 벡터의 요소에 배치합니다.

구문

V*op*{*cond*}.F32 {*Qd*}, *Qn*, *Qm*

V*op*{*cond*}.F32 {*Dd*}, *Dn*, *Dm*

인수 설명:

op RECPS 또는 RSQRTS 중 하나여야 합니다.

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

Qd, *Qn*, *Qm* 쿼드워드 연산에 대한 대상 벡터, 첫 번째 피연산자 벡터 및 두 번째 피연산자 벡터입니다.

Dd, *Dn*, *Dm* 더블워드 연산에 대한 대상 벡터, 첫 번째 피연산자 벡터 및 두 번째 피연산자 벡터입니다.

범위를 벗어난 입력 결과

표 5-11에서는 입력 값이 범위를 벗어난 결과를 보여 줍니다.

표 5-11 범위를 벗어난 입력 결과

첫 번째 피연산자 요소	두 번째 피연산자 요소	결과 요소 (VRECPS)	결과 요소 (VRSQRTS)
NaN	-	기본 NaN	기본 NaN
-	NaN	기본 NaN	기본 NaN
+/– 0.0 또는 비정규값	+/– 무한대	2.0	1.5
+/– 무한대	+/– 0.0 또는 비정규값	2.0	1.5

사용법

Newton-Raphson 이터레이션:

$x_{n+1} = x_n\ (2-dx_n)$

x_0이 d에 적용된 VRECPE의 결과이면 $(1/d)$ 로 수렴됩니다.

Newton-Raphson 이터레이션:

$x_{n+1} = x_n\ (3-dx_n^2)\ /2$

x_0이 d에 적용된 VRSQRTE의 결과이면 $(1/\sqrt{d})$ 로 수렴됩니다.

5.11 NEON 곱하기 명령어

이 단원에는 다음 소단원이 포함되어 있습니다.

- 5-74페이지의 *VMUL{L}, VMLA{L} 및 VMLS{L}*
  벡터 곱하기, 곱하기 누산 및 곱하기 빼기
- 5-75페이지의 *VMUL{L}, VMLA{L} 및 VMLS{L} (스칼라 기준)*
  벡터 곱하기, 곱하기 누산 및 곱하기 빼기 (스칼라 기준)
- 5-76페이지의 *VQDMULL, VQDMLAL 및 VQDMLSL (벡터 기준 또는 스칼라 기준)*
  벡터 포화 배수화 곱하기, 곱하기 누산 및 곱하기 빼기 (벡터 또는 스칼라 기준)
- 5-77페이지의 *VQ{R}DMULH (벡터 기준 또는 스칼라 기준)*
  상위 반을 반환하는 벡터 포화 배수화 곱하기 (벡터 또는 스칼라 기준)

5.11.1 VMUL{L}, VMLA{L} 및 VMLS{L}

VMUL (벡터 곱하기) 은 두 벡터의 해당 요소를 곱하고 결과를 대상 벡터에 배치합니다.

VMLA (벡터 곱하기 누산) 는 두 벡터의 해당 요소를 곱하고 결과를 대상 벡터의 요소에 더합니다.

VMLS (벡터 곱하기 빼기) 는 두 벡터의 해당 요소를 곱하고 대상 벡터의 해당 요소에서 결과를 뺀 후 최종 결과를 대상 벡터에 배치합니다.

구문

V*op*{*cond*}.*datatype* {*Qd*}, *Qn*, *Qm*

V*op*{*cond*}.*datatype* {*Dd*}, *Dn*, *Dm*

V*op*L{*cond*}.*datatype Qd*, *Dn*, *Dm*

인수 설명:

op 다음 중 하나여야 합니다.

MUL	곱하기
MLA	곱하기 누산
MLS	곱하기 빼기

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

datatype 다음 중 하나여야 합니다.

I8, I16, I32, F32	MUL, MLA 또는 MLS의 경우
S8, S16, S32	MULL, MLAL 또는 MLSL의 경우
U8, U16, U32	MULL, MLAL 또는 MLSL의 경우
P8	MUL 또는 MULL의 경우

데이터 유형 P8에 대한 자세한 내용은 5-21페이지의 *{0,1}을 통한 다항식 산술*을 참조하십시오.

Qd, *Qn*, *Qm* 쿼드워드 연산에 대한 대상 벡터, 첫 번째 피연산자 벡터 및 두 번째 피연산자 벡터입니다.

Dd, *Dn*, *Dm* 더블워드 연산에 대한 대상 벡터, 첫 번째 피연산자 벡터 및 두 번째 피연산자 벡터입니다.

Qd, *Dn*, *Dm* long 연산에 대한 대상 벡터, 첫 번째 피연산자 벡터 및 두 번째 피연산자 벡터입니다.

5.11.2 VMUL{L}, VMLA{L} 및 VMLS{L} (스칼라 기준)

VMUL (스칼라 기준 벡터 곱하기) 은 벡터의 각 요소에 스칼라를 곱하고 결과를 대상 벡터에 배치합니다.

VMLA (벡터 곱하기 누산) 는 벡터의 각 요소에 스칼라를 곱하고 결과를 대상 벡터의 해당 요소에 누산합니다.

VMLS (벡터 곱하기 빼기) 는 벡터의 각 요소에 스칼라를 곱하고 대상 벡터의 해당 요소에서 결과를 뺀 다음 최종 결과를 대상 벡터에 배치합니다.

구문

V*op*{*cond*}.*datatype* {*Qd*}, *Qn*, *Dm*[*x*]

V*op*{*cond*}.*datatype* {*Dd*}, *Dn*, *Dm*[*x*]

V*op*L{*cond*}.*datatype Qd*, *Dn*, *Dm*[*x*]

인수 설명:

op 다음 중 하나여야 합니다.

MUL	곱하기
MLA	곱하기 누산
MLS	곱하기 빼기

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

datatype 다음 중 하나여야 합니다.

I16, I32, F32	MUL, MLA 또는 MLS의 경우
S16, S32	MULL, MLAL 또는 MLSL의 경우
U16, U32	MULL, MLAL 또는 MLSL의 경우

Qd, *Qn* 쿼드워드 연산에 대한 대상 벡터 및 첫 번째 피연산자 벡터입니다.

Dd, *Dn* 더블워드 연산에 대한 대상 벡터 및 첫 번째 피연산자 벡터입니다.

Qd, *Dn* long 연산에 대한 대상 벡터 및 첫 번째 피연산자 벡터입니다.

Dm[*x*] 두 번째 피연산자가 들어 있는 스칼라입니다.

5.11.3 VQDMULL, VQDMLAL 및 VQDMLSL (벡터 기준 또는 스칼라 기준)

벡터 포화 배수화 곱하기 명령어는 피연산자를 곱하고 결과를 배수화합니다. VQDMULL은 결과를 대상 레지스터에 배치합니다. VQDMLAL은 결과를 대상 레지스터의 값에 더합니다. VQDMLSL은 대상 레지스터의 값에서 결과를 뺍니다.

결과가 오버플로되는 경우 포화됩니다. 포화가 발생하면 스티키 QC 플래그 (FPSCR bit[27]) 가 설정됩니다.

구문

VQD*op*L{*cond*}.*datatype Qd*, *Dn*, *Dm*

VQD*op*L{*cond*}.*datatype Qd*, *Dn*, *Dm*[*x*]

인수 설명:

op 다음 중 하나여야 합니다.

- MUL 곱하기
- MLA 곱하기 누산
- MLS 곱하기 빼기

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

datatype S16 또는 S32 중 하나여야 합니다.

Qd, *Dn* 대상 벡터 및 첫 번째 피연산자 벡터입니다.

Dm *벡터 기준* 연산에 대한 두 번째 피연산자가 들어 있는 벡터입니다.

Dm[*x*] *스칼라 기준* 연산에 대한 두 번째 피연산자가 들어 있는 스칼라입니다.

5.11.4 VQ{R}DMULH (벡터 기준 또는 스칼라 기준)

벡터 포화 배수화 곱하기 명령어는 피연산자를 곱하고 결과를 배수화합니다. 이 명령어는 결과의 상위 반만 반환합니다.

결과가 오버플로되는 경우 포화됩니다. 포화가 발생하면 스티키 QC 플래그 (FPSCR bit[27]) 가 설정됩니다.

구문

VQ{R}DMULH{*cond*}.*datatype* {*Qd*}, *Qn*, *Qm*

VQ{R}DMULH{*cond*}.*datatype* {*Dd*}, *Dn*, *Dm*

VQ{R}DMULH{*cond*}.*datatype* {*Qd*}, *Qn*, *Dm*[*x*]

VQ{R}DMULH{*cond*}.*datatype* {*Dd*}, *Dn*, *Dm*[*x*]

인수 설명:

R — 있을 경우 각 결과가 반올림됨을 나타내고, 그렇지 않으면 각 결과가 잘립니다.

cond — 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

datatype — S16 또는 S32 중 하나여야 합니다.

Qd, *Qn* — 쿼드워드 연산에 대한 대상 벡터 및 첫 번째 피연산자 벡터입니다.

Dd, *Dn* — 더블워드 연산에 대한 대상 벡터 및 첫 번째 피연산자 벡터입니다.

Qm 또는 *Dm* — *벡터 기준* 연산에 대한 두 번째 피연산자가 들어 있는 벡터입니다.

Dm[*x*] — *스칼라 기준* 연산에 대한 두 번째 피연산자가 들어 있는 스칼라입니다.

5.12 NEON 요소 및 구조체 로드/저장 명령어

이 단원에는 다음 소단원이 포함되어 있습니다.

- *인터리브*
- 5-79페이지의 *요소 및 구조체 로드와 저장 명령어의 정렬에 대한 제한*
- 5-80페이지의 *VLDn 및 VSTn (단일 n-요소 구조체를 하나의 레인에 로드)* 대부분의 데이터 액세스에 사용됩니다. 일반 벡터가 로드될 수 있습니다 ($n = 1$)
- 5-82페이지의 *VLDn (단일 n-요소 구조체를 모든 레인에 로드)*
- 5-84페이지의 *VLDn 및 VSTn (여러 n-요소 구조체)*

5.12.1 인터리브

이 그룹에 있는 대부분의 명령어는 구조체를 메모리에 저장할 때 인터리브를 제공하고 메모리에서 구조체를 로드할 때 디인터리브를 제공합니다. 그림 5-8에서는 디인터리브의 예를 보여 줍니다. 인터리브는 역방향 프로세스입니다.

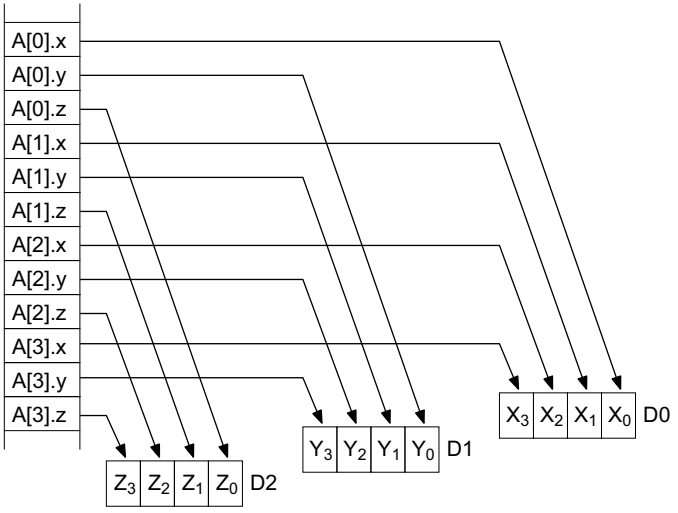

그림 5-8 3요소 구조체 배열 디인터리브

5.12.2 요소 및 구조체 로드와 저장 명령어의 정렬에 대한 제한

이러한 명령어의 대부분에서는 메모리 정렬 제한을 지정할 수 있습니다. 명령어에서 정렬을 지정하지 않으면 정렬 제한은 A비트 (CP15 레지스터 1비트[1]) 에 의해 제어됩니다.

- A비트가 0이면 정렬 제한이 없습니다 (강력한 순서가 지정된 경우나 액세스가 요소로 정렬되어야 하거나 결과를 예상할 수 없는 장치 메모리의 경우 제외).
- A비트가 1이면 액세스가 요소로 정렬되어야 합니다.

주소가 제대로 정렬되지 않으면 정렬 오류가 발생합니다.

5.12.3 VLD*n* 및 VST*n* (단일 *n*-요소 구조체를 하나의 레인에 로드)

단일 *n*-요소 구조체를 하나의 레인에 벡터 로드. 단일 *n*-요소 구조체가 메모리에서 하나 이상의 NEON 레지스터로 로드됩니다. 로드되지 않은 레지스터 요소는 변경되지 않습니다.

단일 *n*-요소 구조체를 하나의 레인에 벡터 저장. 하나의 *n*-요소 구조체가 하나 이상의 NEON 레지스터에서 메모리에 저장됩니다.

구문

```
V{op}n{cond}.datatype list, [Rn{@align}]{!}
V{op}n{cond}.datatype list, [Rn{@align}], Rm
```

인수 설명:

op LD 또는 ST 중 하나여야 합니다.

n 1, 2, 3 또는 4 중 하나여야 합니다.

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

datatype 5-81페이지의 표 5-12를 참조하십시오.

list NEON 레지스터 목록을 지정합니다. 옵션을 보려면 5-81페이지의 표 5-12를 참조하십시오.

Rn 기본 주소가 들어 있는 ARM 레지스터입니다. *Rn*은 r15이면 안 됩니다.

align 선택적 정렬을 지정합니다. 옵션을 보려면 5-81페이지의 표 5-12를 참조하십시오.

! ! 기호가 있으면 *Rn*이 (*Rn* + 명령어로 전송된 바이트 수) 로 업데이트됩니다. 업데이트는 로드 또는 저장이 모두 수행된 후에 수행됩니다.

Rm 기본 주소의 오프셋이 들어 있는 ARM 레지스터입니다. *Rm*이 있으면 메모리 액세스에 주소가 사용된 *후* *Rn*이 (*Rn* + *Rm*) 으로 업데이트됩니다. *Rm*은 r13 또는 r15일 수 없습니다.

표 5-12 허용된 매개변수 조합

n	*datatype*	list [a]	*align* [b]	정렬
1	8	{Dd[x]}	-	표준에만 해당
	16	{Dd[x]}	@16	2바이트
	32	{Dd[x]}	@32	4바이트
2	8	{Dd[x], D (d+1) [x]}	@16	2바이트
	16	{Dd[x], D (d+1) [x]}	@32	4바이트
		{Dd[x], D (d+2) [x]}	@32	4바이트
	32	{Dd[x], D (d+1) [x]}	@64	8바이트
		{Dd[x], D (d+2) [x]}	@64	8바이트
3	8	{Dd[x], D (d+1) [x], D (d+2) [x]}	-	표준에만 해당
	16 또는 32	{Dd[x], D (d+1) [x], D (d+2) [x]}	-	표준에만 해당
		{Dd[x], D (d+2) [x], D (d+4) [x]}	-	표준에만 해당
4	8	{Dd[x], D (d+1) [x], D (d+2) [x], D (d+3) [x]}	@32	4바이트
	16	{Dd[x], D (d+1) [x], D (d+2) [x], D (d+3) [x]}	@64	8바이트
		{Dd[x], D (d+2) [x], D (d+4) [x], D (d+6) [x]}	@64	8바이트
	32	{Dd[x], D (d+1) [x], D (d+2) [x], D (d+3) [x]}	@64 또는 @128	8바이트 또는 16 바이트
		{Dd[x], D (d+2) [x], D (d+4) [x], D (d+6) [x]}	@64 또는 @128	8바이트 또는 16 바이트

a. list의 모든 레지스터는 D0 ~ D31 범위에 있어야 합니다.

b. *Align* 은 생략할 수 있습니다. 이 경우 표준 정렬 규칙이 적용됩니다. 자세한 내용은 5-79페이지의 *요소 및 구조체 로드와 저장 명령어의 정렬에 대한 제한*을 참조하십시오.

5.12.4 VLD*n* (단일 *n*-요소 구조체를 모든 레인에 로드)

단일 *n*-요소 구조체를 모든 레인으로 벡터 로드. *n*-요소 구조체 하나의 여러 복사본을 메모리에서 하나 이상의 NEON 레지스터로 로드합니다.

구문

VLD*n*{*cond*}.*datatype list*, [*Rn*{@*align*}]{!}

VLD*n*{*cond*}.*datatype list*, [*Rn*{@*align*}], *Rm*

인수 설명:

n 1, 2, 3 또는 4 중 하나여야 합니다.

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

datatype 표 5-13을 참조하십시오.

list NEON 레지스터 목록을 지정합니다. 옵션을 보려면 표 5-13을 참조하십시오.

Rn 기본 주소가 들어 있는 ARM 레지스터입니다. *Rn*은 r15이면 안 됩니다.

align 선택적 정렬을 지정합니다. 옵션을 보려면 표 5-13을 참조하십시오.

! ! 기호가 있으면 *Rn*이 (*Rn* + 명령어로 전송된 바이트 수) 로 업데이트됩니다. 업데이트는 로드 또는 저장이 모두 수행된 후에 수행됩니다.

Rm 기본 주소의 오프셋이 들어 있는 ARM 레지스터입니다. *Rm*이 있으면 메모리 액세스에 주소가 사용된 *후* *Rn*이 (*Rn* + *Rm*) 으로 업데이트됩니다. *Rm*은 r13 또는 r15일 수 없습니다.

표 5-13 허용된 매개변수 조합

n	*datatype*	list [a]	*align* [b]	정렬
1	8	{Dd[]}	-	표준에만 해당
		{Dd[],D (d+1) []}	-	표준에만 해당
	16	{Dd[]}	@16	2바이트
		{Dd[],D (d+1) []}	@16	2바이트

표 5-13 허용된 매개변수 조합 (계속)

n	*datatype*	list [a]	*align* [b]	정렬
	32	{Dd[]}	@32	4바이트
		{Dd[],D (d+1) []}	@32	4바이트
2	8	{Dd[], D (d+1) []}	@8	바이트
		{Dd[], D (d+2) []}	@8	바이트
	16	{Dd[], D (d+1) []}	@16	2바이트
		{Dd[], D (d+2) []}	@16	2바이트
	32	{Dd[], D (d+1) []}	@32	4바이트
		{Dd[], D (d+2) []}	@32	4바이트
3	8, 16 또는 32	{Dd[], D (d+1) [], D (d+2) []}	-	표준에만 해당
		{Dd[], D (d+2) [], D (d+4) []}	-	표준에만 해당
4	8	{Dd[], D (d+1) [], D (d+2) [], D (d+3) []}	@32	4바이트
		{Dd[], D (d+2) [], D (d+4) [], D (d+6) []}	@32	4바이트
	16	{Dd[], D (d+1) [], D (d+2) [], D (d+3) []}	@64	8바이트
		{Dd[], D (d+2) [], D (d+4) [], D (d+6) []}	@64	8바이트
	32	{Dd[], D (d+1) [], D (d+2) [], D (d+3) []}	@64 또는 @128	8바이트 또는 16바이트
		{Dd[], D (d+2) [], D (d+4) [], D (d+6) []}	@64 또는 @128	8바이트 또는 16바이트

a. list의 모든 레지스터는 D0 ~ D31 범위에 있어야 합니다.

b. *Align* 은 생략할 수 있습니다. 이 경우 표준 정렬 규칙이 적용됩니다. 자세한 내용은 5-79페이지의 *요소 및 구조체 로드와 저장 명령어의 정렬에 대한 제한*을 참조하십시오.

5.12.5 VLD*n* 및 VST*n* (여러 *n*-요소 구조체)

여러 *n*-요소 구조체를 벡터 로드. *n* == 1이 아닌 경우 디인터리브를 사용하여 여러 *n*-요소 구조체를 메모리에서 하나 이상의 NEON 레지스터로 로드합니다. 이때 각 레지스터의 모든 요소가 로드됩니다.

여러 *n*-요소 구조체를 벡터 저장. *n* == 1이 아닌 경우 인터리브를 사용하여 여러 *n*-요소 구조체를 하나 이상의 NEON 레지스터에서 메모리에 저장합니다. 이때 각 레지스터의 모든 요소가 저장됩니다.

구문

```
Vopn{cond}.datatype list, [Rn{@align}]{!}
Vopn{cond}.datatype list, [Rn{@align}], Rm
```

인수 설명:

op — LD 또는 ST 중 하나여야 합니다.

n — 1, 2, 3 또는 4 중 하나여야 합니다.

cond — 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

datatype — 옵션을 보려면 5-85페이지의 표 5-14를 참조하십시오.

list — NEON 레지스터 목록을 지정합니다. 옵션을 보려면 5-85페이지의 표 5-14를 참조하십시오.

Rn — 기본 주소가 들어 있는 ARM 레지스터입니다. *Rn*은 r15이면 안 됩니다.

align — 선택적 정렬을 지정합니다. 옵션을 보려면 5-85페이지의 표 5-14를 참조하십시오.

! — ! 기호가 있으면 *Rn*이 (*Rn* + 명령어로 전송된 바이트 수) 로 업데이트됩니다. 업데이트는 로드 또는 저장이 모두 수행된 후에 수행됩니다.

Rm — 기본 주소의 오프셋이 들어 있는 ARM 레지스터입니다. *Rm*이 있으면 메모리 액세스에 주소가 사용된 *후* *Rn*이 (*Rn* + *Rm*) 으로 업데이트됩니다. *Rm*은 r13 또는 r15일 수 없습니다.

표 5-14 허용된 매개변수 조합

n	*datatype*	list [a]	*align* [b]	*정렬*
1	8, 16, 32 또는 64	{Dd}	@64	8바이트
		{Dd, D (d+1) }	@64 또는 @128	8바이트 또는 16바이트
		{Dd, D (d+1), D (d+2) }	@64	8바이트
		{Dd, D (d+1), D (d+2), D (d+3) }	@64, @128 또는 @256	8바이트, 16바이트 또는 32바이트
2	8, 16 또는 32	{Dd, D (d+1) }	@64, @128	8바이트 또는 16바이트
		{Dd, D (d+2) }	@64, @128	8바이트 또는 16바이트
		{Dd, D (d+1), D (d+2), D (d+3) }	@64, @128 또는 @256	8바이트, 16바이트 또는 32바이트
3	8, 16 또는 32	{Dd, D (d+1), D (d+2) }	@64	8바이트
		{Dd, D (d+2), D (d+4) }	@64	8바이트
4	8, 16 또는 32	{Dd, D (d+1), D (d+2), D (d+3) }	@64, @128 또는 @256	8바이트, 16바이트 또는 32바이트
		{Dd, D (d+2), D (d+4), D (d+6) }	@64, @128 또는 @256	8바이트, 16바이트 또는 32바이트

a. list의 모든 레지스터는 D0 ~ D31 범위에 있어야 합니다.

b. *Align* 은 생략할 수 있습니다. 이 경우 표준 정렬 규칙이 적용됩니다. 자세한 내용은 5-79페이지의 *요소 및 구조체 로드와 저장 명령어의 정렬에 대한 제한*을 참조하십시오.

5.13 NEON 및 VFP 의사 명령어

이 단원에는 다음 소단원이 포함되어 있습니다.

- 5-87페이지의 *VLDR 의사 명령어* (NEON 및 VFP)
- 5-88페이지의 *VLDR 및 VSTR (사후 증가 및 사전 감소)* (NEON 및 VFP)
- 5-89페이지의 *VMOV2* (NEON만)
- 5-90페이지의 *VAND 및 VORN (즉치값)* (NEON만)
- 5-91페이지의 *VACLE 및 VACLT* (NEON만)
- 5-92페이지의 *VCLE 및 VCLT* (NEON만)

5.13.1 VLDR 의사 명령어

VLDR 의사 명령어는 상수 값을 64비트 NEON 벡터의 모든 요소나 VFP 단정밀도 또는 배정밀도 레지스터로 로드합니다.

참고

이 단원에서는 VLDR *의사* 명령어에 대해서만 설명합니다. VLDR 명령어에 대한 자세한 내용은 5-24페이지의 *VLDR 및 VSTR*을 참조하십시오.

구문

VLDR{*cond*}.*datatype Dd*,=*constant*

VLDR{*cond*}.*datatype Sd*,=*constant*

인수 설명:

datatype 다음 중 하나여야 합니다.

I*n*	NEON만 해당
S*n*	NEON만 해당
U*n*	NEON만 해당
F32	NEON 또는 VFP
F64	VFP만 해당

n 8, 16, 32 또는 64 중 하나여야 합니다.

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

Dd 또는 *Sd* 로드할 확장 레지스터입니다.

constant *datatype*에 적합한 유형의 상수입니다.

사용법

레지스터에 상수를 직접 생성할 수 있는 명령어 (예: VMOV) 를 사용할 수 있으면 어셈블러에서 이 명령어를 사용합니다. 그렇지 않으면 어셈블러에서 상수가 포함된 더블워드 리터럴 풀 엔트리를 생성하고 VLDR 명령어를 사용하여 상수를 로드합니다.

5.13.2 VLDR 및 VSTR (사후 증가 및 사전 감소)

사후 증가 및 사전 감소를 통해 확장 레지스터를 로드하거나 저장하는 의사 명령어입니다.

참고

사후 증가 및 사전 감소 없는 VLDR 및 VSTR명령어에 대한 자세한 내용은 5-24페이지의 *VLDR 및 VSTR*을 참조하십시오.

구문

```
op{cond}{.size} Fd, [Rn], #offset          ; post-increment
op{cond}{.size} Fd, [Rn, #-offset]!        ; pre-decrement
```

인수 설명:

op 다음과 같을 수 있습니다.
- VLDR - 메모리에서 확장 레지스터 로드
- VSTR - 확장 레지스터 내용을 메모리에 저장

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

size 선택적 데이터 크기 지정자입니다. *Fd*가 단정밀도 VFP 레지스터이면 32여야 하고, *Fd*가 배정밀도 레지스터이면 64여야 합니다.

Fd 로드 또는 저장할 확장 레지스터로, NEON 명령어의 경우 배정밀도 (*Dd*) 레지스터여야 합니다. VFP 명령어의 경우 배정밀도 (*Dd*) 또는 단정밀도 (*Sd*) 레지스터일 수 있습니다.

Rn 전송할 기본 주소가 들어 있는 ARM 레지스터입니다.

offset 어셈블리 타임에 숫자 상수로 평가되어야 하는 숫자 식입니다. *Fd*가 단정밀도 VFP 레지스터이면 값은 4여야 하고, *Fd*가 배정밀도 레지스터이면 값은 8이어야 합니다.

사용법

사후 증가 명령어는 전송 후에 오프셋 값만큼 레지스터의 기본 주소를 증가시킵니다. 사전 감소 명령어는 레지스터의 기본 주소를 오프셋 값만큼 감소시킨 후에 레지스터의 새 주소를 사용하여 전송을 수행합니다. 이러한 의사 명령어는 VLDM 또는 VSTM 명령어로 어셈블됩니다 (5-25페이지의 *VLDM, VSTM, VPOP 및 VPUSH* 참조).

5.13.3 VMOV2

VMOV2 의사 명령어는 리터럴 풀에서 값을 로드하지 않고 상수를 생성하여 NEON 벡터의 모든 요소에 배치합니다. 이 명령어는 항상 정확히 두 개의 명령어로 어셈블됩니다.

VMOV2는 모든 16비트 상수와 제한된 범위의 32비트 및 64비트 상수를 생성할 수 있습니다.

구문

VMOV2{*cond*}.*datatype Qd*, *#constant*

VMOV2{*cond*}.*datatype Dd*, *#constant*

인수 설명:

datatype 다음 중 하나여야 합니다.

- I8, I16, I32 또는 I64
- S8, S16, S32 또는 S64
- U8, U16, U32 또는 U64
- F32

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

Qd 또는 *Dd* 로드할 확장 레지스터입니다.

constant *datatype*에 적합한 유형의 상수입니다.

사용법

일반적으로 VMOV2는 VMOV 또는 VMVN 명령어로 어셈블되며, 그 뒤에는 VBIC 또는 VORR 명령어가 옵니다. 자세한 내용은 5-46페이지의 *VMOV, VMVN (즉치값)* 및 5-33페이지의 *VBIC 및 VORR (즉치값)*을 참조하십시오.

5.13.4 VAND 및 VORN (즉치값)

VAND (비트 단위 AND 즉치값) 는 대상 벡터의 각 요소를 가져와서 즉치값 상수로 비트 단위 AND를 수행하고 결과를 대상 벡터에 반환합니다.

VORN (비트 단위 OR NOT 즉치값) 은 대상 벡터의 각 요소를 가져와서 즉치값 상수로 비트 단위 OR 보수를 수행하고 결과를 대상 벡터에 반환합니다.

참고

디스어셈블리에서 이러한 의사 명령어는 보수 즉치 상수를 통해 해당 VBIC 및 VORR 명령어로 디스어셈블됩니다.

구문

V*op*{*cond*}.*datatype Qd*, #*imm*

V*op*{*cond*}.*datatype Dd*, #*imm*

인수 설명:

op	VAND 또는 VORN 중 하나여야 합니다.
cond	선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).
datatype	I8, I16, I32 또는 I64 중 하나여야 합니다.
Qd 또는 *Dd*	결과의 NEON 레지스터입니다.
imm	즉치 상수입니다.

즉치 상수

*datatype*이 I16이면 즉치 상수는 다음 형식 중 하나를 포함해야 합니다.

- 0xFFXY
- 0xXYFF

*datatype*이 I32이면 즉치 상수는 다음 형식 중 하나를 포함해야 합니다.

- 0xFFFFFFXY
- 0xFFFFXYFF
- 0xFFXYFFFF
- 0xXYFFFFFF

자세한 내용은 5-33페이지의 *VBIC 및 VORR (즉치값)*을 참조하십시오.

5.13.5 VACLE 및 VACLT

벡터 절대 비교는 벡터에 있는 각 요소의 절대값을 가져와서 두 번째 벡터의 해당 요소 절대값과 비교합니다. 조건이 true이면 대상 벡터의 해당 요소가 모두 1로 설정되고, 그렇지 않으면 모두 0으로 설정됩니다.

참고

디스어셈블리에서 이러한 의사 명령어는 피연산자가 반전되어 해당 VACGE 및 VACGT 명령어로 디스어셈블됩니다.

구문

VAC*op*{*cond*}.*datatype* {*Qd*}, *Qn*, *Qm*

VAC*op*{*cond*}.*datatype* {*Dd*}, *Dn*, *Dm*

인수 설명:

op 다음 중 하나여야 합니다.
- LE 절대 작거나 같음
- LT 절대 보다 작음

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

datatype F32여야 합니다.

Qd 또는 *Dd* 결과의 NEON 레지스터입니다.
결과 데이터 유형은 I32입니다.

Qn 또는 *Dn* 첫 번째 피연산자가 들어 있는 NEON 레지스터입니다.

Qm 또는 *Dm* 두 번째 피연산자가 들어 있는 NEON 레지스터입니다.

5.13.6 VCLE 및 VCLT

벡터 비교는 벡터에 있는 각 요소의 값을 가져와서 두 번째 벡터의 해당 요소 절대값 또는 0과 비교합니다. 조건이 true이면 대상 벡터의 해당 요소가 모두 1로 설정되고, 그렇지 않으면 모두 0으로 설정됩니다.

참고

디스어셈블리에서 이러한 의사 명령어는 피연산자가 반전되어 해당 VCGE 및 VCGT 명령어로 디스어셈블됩니다.

구문

VC*op*{*cond*}.*datatype* {*Qd*}, *Qn*, *Qm*

VC*op*{*cond*}.*datatype* {*Dd*}, *Dn*, *Dm*

인수 설명:

op 다음 중 하나여야 합니다.
- LE 작거나 같음
- LT 보다 작음

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

datatype must be one of S8, S16, S32, U8, U16, U32 또는 F32

Qd 또는 *Dd* 결과의 NEON 레지스터입니다.

결과 데이터 유형은 다음과 같습니다.
- 피연산자 데이터 형식 I32, S32, U32 또는 F32의 경우 I32
- 피연산자 데이터 유형 I16, S16 또는 U16의 경우 I16
- I8피연산자 데이터 유형 I8, S8 또는 U8.의 경우

Qn 또는 *Dn* 첫 번째 피연산자가 들어 있는 NEON 레지스터입니다.

Qm 또는 *Dm* 두 번째 피연산자가 들어 있는 NEON 레지스터입니다.

5.14 NEON 및 VFP 시스템 레지스터

NEON 및 VFP의 모든 구현에서 세 개의 NEON 및 VFP 시스템 레지스터에 액세스할 수 있습니다.

- *FPSCR, 부동 소수점 상태 및 제어 레지스터*
- 5-96페이지의 *FPEXC, 부동 소수점 예외 레지스터*
- 5-96페이지의 *FPSID, 부동 소수점 시스템 ID 레지스터*
- 5-97페이지의 *NEON 및 VFP 시스템 레지스터의 개별 비트 수정*

NEON 또는 VFP의 특정 구현에는 추가 레지스터가 있을 수 있습니다. 자세한 내용은 사용 중인 VFP 보조 프로세스의 기술 참조 설명서를 참조하십시오.

5.14.1 FPSCR, 부동 소수점 상태 및 제어 레지스터

FPSCR에는 모든 사용자 수준 NEON 및 VFP 상태 및 제어 비트가 포함되어 있습니다. NEON은 비트[31:27]만 사용합니다. 이 비트는 다음과 같이 사용됩니다.

비트[31:28] N, Z, C 및 V 플래그로, NEON 및 VFP의 상태를 나타냅니다. 이러한 플래그는 CPSR의 상태 플래그에 복사될 때까지 조건부 실행을 제어하는 데 사용할 수 없습니다 (5-13페이지의 *조건 코드* 참조).

비트[27] QC, 누적 포화 플래그로, NEON 포화 명령어에서 포화가 발생할 경우 설정됩니다.

비트[25] DN (*기본 NaN*) 모드 제어 비트입니다.

0 비활성화됩니다. NaN 피연산자는 부동 소수점 연산 출력으로 전달됩니다.

1 활성화됩니다. 하나 이상의 NaN이 포함된 모든 연산은 기본 NaN으로 반환됩니다.

참고

NEON은 이 비트에 관계없이 항상 기본 NaN이 활성화된 설정을 사용합니다.

비트[24] 0으로 플러시 모드 제어 비트입니다.

0 0으로 플러시 모드가 해제됩니다.

1 0으로 플러시 모드가 설정됩니다.

0으로 플러시 모드에서는 범위를 줄이면 하드웨어와 소프트웨어에 따라 성능이 향상될 수 있습니다 (5-98페이지의 *0으로 플러시 모드* 참조).

참고

NEON은 이 비트에 관계없이 항상 0으로 플러시 모드를 사용합니다.

IEEE 754와의 호환성이 필요한 경우 0으로 플러시 모드를 사용하면 안 됩니다.

비트[23:22] 다음과 같이 반올림 모드를 제어합니다.

0b00 *가장 가까운 수로 반올림* (RN) 모드

0b01 *양의 무한대로 반올림* (RP) 모드

0b10 *음의 무한대로 반올림* (RM) 모드

0b11 *0으로 반올림* (RZ) 모드

참고

NEON은 이 비트에 관계없이 항상 가장 가까운 수로 반올림 모드를 사용합니다.

비트[21:20] STRIDE는 벡터에 연속해서 있는 값 간의 거리입니다 (5-112페이지의 *벡터* 참조). 스트라이드는 다음과 같이 제어됩니다.

0b00 STRIDE = 1

0b11 STRIDE = 2

비트[18:16] LEN은 각 벡터에서 사용되는 레지스터 수입니다 (5-112페이지의 *벡터* 참조). 1 + 비트[18:16] 값은 다음과 같습니다.

0b000 LEN = 1

...

0b111 LEN = 8

비트[15, 12:8] 예외 트랩 활성화 비트입니다.

IDE 비정규 입력 예외 활성화

IXE 정확하지 않은 예외 활성화

UFE 언더플로 예외 활성화

OFE 오버플로 예외 활성화

DZE 0으로 나누기 예외 활성화

IOE 잘못된 연산 예외 활성화

이 설명서에서는 부동 소수점 예외 트랩의 사용에 대해 설명하지 않습니다. 자세한 내용은 사용 중인 VFP 보조 프로세서의 기술 참조 설명서를 참조하십시오.

비트[7, 4:0] 누적 예외 비트는 다음과 같습니다.

IDC 비정규 입력 예외

IXC 정확하지 않은 예외

UFC 언더플로 예외

OFC 오버플로 예외

DZC 0으로 나누기 예외

IOC 잘못된 연산 예외

해당 예외가 발생하면 누적 예외 비트가 설정됩니다. 누적 예외 비트는 사용자가 FPSCR에 직접 기록하여 지울 때까지 설정된 상태로 있습니다.

다른 모든 비트

기본 NEON 및 VFP 사양에서는 사용되지 않지만 특정 구현에서는 사용될 수 있습니다. 자세한 내용은 사용 중인 VFP 보조 프로세서의 기술 참조 설명서를 참조하십시오. 특정 구현에서 사용될 경우를 제외하고 이러한 비트를 수정하면 안 됩니다.

다른 비트에 영향을 주지 않고 원하는 비트만 변경하려면 읽기- 수정- 쓰기 프로시저를 사용하십시오 (5-97페이지의 *NEON 및 VFP 시스템 레지스터의 개별 비트 수정* 참조).

참고

벡터 모드 사용이 제공되지 않습니다. LEN 및 STRIDE를 1로 설정하십시오.

5.14.2 FPEXC, 부동 소수점 예외 레지스터

FPEXC에는 권한 모드로만 액세스할 수 있습니다. 여기에는 다음 비트가 포함됩니다.

비트[31] EX 비트입니다. 모든 NEON 및 VFP 구현에서 읽을 수 있으며 일부 구현에서는 이 비트에 쓸 수도 있습니다.

값이 0이면 NEON 또는 VFP 시스템의 중요 상태만 범용 레지스터와 FPSCR 및 FPEXC로 구성됩니다.

값이 1이면 상태를 저장하기 위해 구현 관련 정보가 필요합니다. 자세한 내용은 사용 중인 VFP 보조 프로세서의 기술 참조 설명서를 참조하십시오.

비트[30] EN 비트입니다. 모든 NEON 또는 VFP 구현에서 읽고 쓸 수 있습니다.

값이 1이면 NEON (있을 경우) 및 VFP (있을 경우) 가 활성화되고 정상적으로 작동합니다.

값이 0이면 NEON 및 VFP가 비활성화됩니다. 비활성화되면 FPSID 또는 FPEXC 레지스터를 읽거나 쓸 수 있지만 다른 NEON 또는 VFP 명령어는 정의되지 않은 명령어로 간주됩니다.

비트[29:0] VFP의 특정 구현에서 사용될 수 있습니다. 이 장에 설명된 모든 VFP 함수는 이러한 비트에 액세스하지 않고 사용할 수 있습니다.

특정 구현에서 사용되는 경우를 제외하고 이러한 비트는 변경하면 안 됩니다. 자세한 내용은 사용 중인 VFP 보조 프로세서의 기술 참조 설명서를 참조하십시오.

다른 비트에 영향을 주지 않고 원하는 비트만 변경하려면 읽기- 수정- 쓰기 프로시저를 사용하십시오 (5-97페이지의 *NEON 및 VFP 시스템 레지스터의 개별 비트 수정* 참조).

5.14.3 FPSID, 부동 소수점 시스템 ID 레지스터

FPSID는 읽기 전용 레지스터입니다. 이 레지스터를 읽으면 프로그램이 실행 중인 NEON 또는 VFP 아키텍처의 구현을 찾을 수 있습니다.

5.14.4 NEON 및 VFP 시스템 레지스터의 개별 비트 수정

다른 비트에 영향을 주지 않고 원하는 NEON 및 VFP 시스템 레지스터 비트만 변경하려면 다음 예제와 같이 읽기- 수정- 쓰기 프로시저를 사용합니다.

```
VMRS    r10,FPSCR           ; copy FPSCR into r10
BIC     r10,r10,#0x00370000 ; clears STRIDE and LEN
ORR     r10,r10,#0x00030000 ; sets STRIDE = 1, LEN = 4
VMSR    FPSCR,r10           ; copy r10 back into FPSCR
```

자세한 내용은 5-30페이지의 *VMRS 및 VMSR*을 참조하십시오.

5.15 0으로 플러시 모드

VFP의 일부 구현에서는 지원 코드를 사용하여 비정규 숫자를 처리합니다. 시스템의 성능은 정규 계산을 수행할 때보다 비정규 숫자와 관련된 계산을 수행할 때 훨씬 저하됩니다.

0으로 플러시 모드는 비정규 숫자를 0으로 바꿉니다. 이 모드는 IEEE 754 산술을 준수하지 않지만 경우에 따라 성능을 상당히 향상시킬 수 있습니다.

NEON 및 VFPv3의 0으로 플러시는 부호 비트를 유지합니다. VFPv2의 0으로 플러시는 +0으로 플러시합니다.

NEON은 항상 0으로 플러시 모드를 사용합니다.

5.15.1 0으로 플러시 모드를 사용할 경우

다음에 모두 해당하는 경우 0으로 플러시 모드를 선택해야 합니다.

- 시스템에서 IEEE 754를 준수하지 않아도 됩니다.
- 사용 중인 알고리즘이 경우에 따라 비정규 숫자를 생성합니다.
- 시스템에서 지원 코드를 사용하여 비정규 숫자를 처리합니다.
- 사용 중인 알고리즘의 정확성이 비정규 숫자의 보유 여부에 의해 결정되지 않습니다.
- 사용 중인 알고리즘이 비정규 숫자를 0으로 바꾸는 작업의 결과로 자주 발생하는 예외를 생성하지 않습니다.

코드 부분에 따라 요구 사항이 다를 경우 언제든지 0으로 플러시 모드와 및 표준 모드 사이에서 변경할 수 있습니다. 레지스터에 이미 포함되어 있는 숫자는 모드 변경의 영향을 받지 않습니다.

5.15.2 0으로 플러시 모드의 사용에 따른 영향

특정 예외 상황에서 0으로 플러시 모드는 부동 소수점 연산에 다음과 같은 영향을 줍니다 (5-99페이지의 *0으로 플러시 모드의 영향을 받지 않는 연산* 참조).

- 비정규 숫자가 부동 소수점 연산에 대한 입력으로 사용될 경우 0으로 처리됩니다. 소스 레지스터는 변경되지 않습니다.
- 단정밀도 소수점 연산의 결과가 반올림하기 전에 $-2^{-126} \sim +2^{-126}$ 범위에 있으면 결과가 0으로 대체됩니다.

- 배정밀도 소수점 연산의 결과가 반올림하기 전에 -2^{-1022} ~ $+2^{-1022}$ 범위에 있으면 결과가 0으로 대체됩니다.

비정규 숫자가 피연산자로 사용되거나 결과가 0으로 플러시될 때마다 정확하지 않은 예외가 발생합니다. 언더플로 예외는 0으로 플러시 모드에서 발생하지 않습니다.

5.15.3 0으로 플러시 모드의 영향을 받지 않는 연산

다음 NEON 및 VFP 연산은 0으로 플러시 모드에서도 결과를 0으로 플러시하지 않고 비정규 숫자에서 수행할 수 있습니다.

- 복사, 절대값 및 부정 (5-36페이지의 *VMOV, VMVN (레지스터)*, 5-101페이지의 *VABS, VNEG 및 VSQRT* 및 5-62페이지의 *V{Q}ABS 및 V{Q}NEG* 참조)
- 복제 (5-44페이지의 *VDUP* 참조)
- 스왑 (5-49페이지의 *VSWP* 참조)
- 로드 및 저장 (5-24페이지의 *VLDR 및 VSTR* 참조)
- 다중 로드 또는 다중 저장 (5-25페이지의 *VLDM, VSTM, VPOP 및 VPUSH* 참조)
- 확장 레지스터와 ARM 범용 레지스터 간 전송 (5-27페이지의 *VMOV (두 개의 ARM 레지스터와 확장 레지스터 간 전송)*, 5-28페이지의 *VMOV (ARM 레지스터와 NEON 스칼라 간 전송)* 및 5-29페이지의 *VMOV (한 ARM 레지스터와 단정밀도 VFP 간 전송)* 참조)

5.16 VFP 명령어

이 단원에는 다음 소단원이 포함되어 있습니다.

- 5-101페이지의 *VABS, VNEG 및 VSQRT*
  부동 소수점 절대값, 부정 및 제곱근
- 5-102페이지의 *VADD, VSUB 및 VDIV*
  부동 소수점 더하기, 빼기 및 나누기
- 5-103페이지의 *VMUL, VMLA, VMLS, VNMUL, VNMLA 및 VNMLS*
  선택적 부정 포함 부동 소수점 곱하기 및 곱하기 누산
- 5-104페이지의 *VCMP*
  부동 소수점 비교
- 5-105페이지의 *VCVT (단정밀도와 배정밀도 간 변환)*
  단정밀도와 배정밀도 간 변환
- 5-106페이지의 *VCVT (부동 소수점과 정수 간 변환)*
  부동 소수점과 정수 간 변환
- 5-107페이지의 *VCVT (부동 소수점과 고정 소수점 간 변환)*
  부동 소수점과 고정 소수점 간 변환
- 5-108페이지의 *VCVTB, VCVTT (반정밀도 확장)*
  반정밀도 부동 소수점과 단정밀도 부동 소수점 간에 변환
- 5-109페이지의 *VMOV*
  부동 소수점 상수를 단정밀도 또는 배정밀도 레지스터에 삽입

5.16.1 VABS, VNEG 및 VSQRT

부동 소수점 절대값, 부정 및 제곱근

이러한 명령어는 스칼라, 벡터 또는 혼합 연산일 수 있습니다 (5-113페이지의 *VFP 벡터 및 스칼라 연산* 참조).

구문

V*op*{*cond*}.F32 *Sd*, *Sm*

V*op*{*cond*}.F64 *Dd*, *Dm*

인수 설명:

op ABS, NEG 또는 SQRT 중 하나입니다.

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

Sd, *Sm* 결과 및 피연산자의 단정밀도 레지스터입니다.

Dd, *Dm* 결과 및 피연산자의 배정밀도 레지스터입니다.

사용법

VABS 명령어는 *Sm* 또는 *Dm*의 내용을 가져와서 부호 비트를 지우고 결과를 *Sd* 또는 *Dd*에 배치합니다. 이 명령어는 절대값을 제공합니다.

VNEG 명령어는 *Sm* 또는 *Dm*의 내용을 가져와서 부호 비트를 변경하고 결과를 *Sd* 또는 *Dd*에 배치합니다. 이 명령어는 값의 부정을 제공합니다.

VSQRT 명령어는 *Sm* 또는 *Dm* 내용의 제곱근을 가져와서 결과를 *Sd* 또는 *Dd*에 배치합니다.

VABS 및 VNEG 명령어의 경우 피연산자가 NaN이면 부호 비트는 위의 각 경우에 따라 결정되지만 예외가 생성되지 않습니다.

부동 소수점 예외

VABS 및 VNEG 명령어는 예외를 생성할 수 없습니다.

VSQRT 명령어는 잘못된 연산 또는 정확하지 않은 예외를 생성할 수 있습니다.

5.16.2 VADD, VSUB 및 VDIV

부동 소수점 더하기, 빼기 및 나누기

이러한 명령어는 스칼라, 벡터 또는 혼합 연산일 수 있습니다 (5-113페이지의 *VFP 벡터 및 스칼라 연산* 참조).

구문

V*op*{*cond*}.F32 {*Sd*}, *Sn*, *Sm*

V*op*{*cond*}.F64 {*Dd*}, *Dn*, *Dm*

인수 설명:

op ADD, SUB 또는 DIV 중 하나입니다.

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

Sd, *Sn*, *Sm* 결과 및 피연산자의 단정밀도 레지스터입니다.

Dd, *Dn*, *Dm* 결과 및 피연산자의 배정밀도 레지스터입니다.

사용법

VADD 명령어는 피연산자 레지스터의 값을 더하고 결과를 대상 레지스터에 배치합니다.

VSUB 명령어는 첫 번째 피연산자 레지스터의 값에서 두 번째 피연산자 레지스터의 값을 빼고 결과를 대상 레지스터에 배치합니다.

VDIV 명령어는 첫 번째 피연산자 레지스터의 값을 두 번째 피연산자 레지스터의 값으로 나누고 결과를 대상 레지스터에 배치합니다.

부동 소수점 예외

VADD 및 VSUB 명령어는 잘못된 연산, 오버플로 또는 정확하지 않은 예외를 생성할 수 있습니다.

VDIV 연산은 0으로 나누기, 잘못된 연산, 오버플로, 언더플로 또는 정확하지 않은 예외를 생성할 수 있습니다.

5.16.3 VMUL, VMLA, VMLS, VNMUL, VNMLA 및 VNMLS

선택적 부정 포함 부동 소수점 곱하기 및 곱하기 누산

이러한 명령어는 스칼라, 벡터 또는 혼합 연산일 수 있습니다 (5-113페이지의 *VFP 벡터 및 스칼라 연산* 참조).

구문

V{N}MUL{*cond*}.F32 {*Sd*,} *Sn*, *Sm*

V{N}MUL{*cond*}.F64 {*Dd*,} *Dn*, *Dm*

V{N}MLA{*cond*}.F32 *Sd*, *Sn*, *Sm*

V{N}MLA{*cond*}.F64 *Dd*, *Dn*, *Dm*

V{N}MLS{*cond*}.F32 *Sd*, *Sn*, *Sm*

V{N}MLS{*cond*}.F64 *Dd*, *Dn*, *Dm*

인수 설명:

N — 최종 결과를 부정합니다.

cond — 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

Sd, *Sn*, *Sm* — 결과 및 피연산자의 단정밀도 레지스터입니다.

Dd, *Dn*, *Dm* — 결과 및 피연산자의 배정밀도 레지스터입니다.

사용법

MUL 연산은 피연산자 레지스터의 값을 곱하고 결과를 대상 레지스터에 배치합니다.

MLA 연산은 피연산자 레지스터의 값을 곱하고 대상 레지스터의 값을 더한 다음 최종 결과를 대상 레지스터에 배치합니다.

MLS 연산은 피연산자 레지스터의 값을 곱하고 대상 레지스터의 값에서 결과를 뺀 다음 최종 결과를 대상 레지스터에 배치합니다.

각각의 경우 N 옵션이 사용되면 최종 결과는 무효화됩니다.

부동 소수점 예외

이러한 명령어는 잘못된 연산, 오버플로, 언더플로 또는 정확하지 않은 예외 또는 비정규 입력 예외를 생성할 수 있습니다.

5.16.4 VCMP

부동 소수점 비교

VCMP는 항상 스칼라입니다.

구문

VCMP{*cond*}.F32 *Sd*, *Sm*

VCMP{*cond*}.F32 *Sd*, #0

VCMP{*cond*}.F64 *Dd*, *Dm*

VCMP{*cond*}.F64 *Dd*, #0

인수 설명:

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

Sd, *Sm* 피연산자가 들어 있는 단정밀도 레지스터입니다.

Dd, *Dm* 피연산자가 들어 있는 배정밀도 레지스터입니다.

사용법

VCMP 명령어는 첫 번째 피연산자의 값에서 두 번째 피연산자 레지스터의 값 (두 번째 피연산자가 #0일 경우 0) 을 빼고 VFP 조건 플래그를 결과에 설정합니다 (5-13페이지의 *조건 코드* 참조).

부동 소수점 예외

VCMP 명령어는 잘못된 연산 예외를 생성할 수 있습니다.

5.16.5 VCVT (단정밀도와 배정밀도 간 변환)

단정밀도 숫자와 배정밀도 숫자 간에 변환합니다.

VCVT는 항상 스칼라입니다.

구문

VCVT{*cond*}.F64.F32 *Dd*, *Sm*

VCVT{*cond*}.F32.F64 *Sd*, *Dm*

인수 설명:

cond	선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).
Dd	결과의 배정밀도 레지스터입니다.
Sm	피연산자가 들어 있는 단정밀도 레지스터입니다.
Sd	결과의 단정밀도 레지스터입니다.
Dm	피연산자가 들어 있는 배정밀도 레지스터입니다.

사용법

이러한 명령어는 *Sm*의 단정밀도 값을 배정밀도로 변환하고 결과를 *Dd*에 배치하거나 *Dm*의 배정밀도 값을 단정밀도로 변환하고 결과를 *Sd*에 배치합니다.

부동 소수점 예외

이러한 명령어는 잘못된 연산, 표준화되지 않은 입력, 오버플로, 언더플로 또는 정확하지 않은 예외를 생성할 수 있습니다.

5.16.6 VCVT (부동 소수점과 정수 간 변환)

부동 소수점 숫자와 정수 간에 변환합니다.

VCVT는 항상 스칼라입니다.

구문

VCVT{R}{*cond*}.*type*.F64 *Sd*, *Dm*

VCVT{R}{*cond*}.*type*.F32 *Sd*, *Sm*

VCVT{*cond*}.F64.*type Dd*, *Sm*

VCVT{*cond*}.F32.*type Sd*, *Sm*

인수 설명:

R — FPSCR에 지정된 반올림 모드를 사용하도록 합니다. 그렇지 않으면 0으로 반올림됩니다.

cond — 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

type — U32 (부호 없는 32비트 정수) 또는 S32 (부호 있는 32비트 정수) 중 하나일 수 있습니다.

Sd — 결과의 단정밀도 레지스터입니다.

Dd — 결과의 배정밀도 레지스터입니다.

Sm — 피연산자가 들어 있는 단정밀도 레지스터입니다.

Dm — 피연산자가 들어 있는 배정밀도 레지스터입니다.

사용법

이 명령어의 처음 두 형식은 부동 소수점을 정수로 변환합니다.

세 번째와 네 번째 형식은 정수를 부동 소수점으로 변환합니다.

부동 소수점 예외

이러한 명령어는 비정규 입력, 잘못된 연산 또는 정확하지 않은 예외를 생성할 수 있습니다.

5.16.7 VCVT (부동 소수점과 고정 소수점 간 변환)

부동 소수점 숫자와 고정 소수점 숫자 간에 변환합니다.

VCVT는 항상 스칼라입니다.

구문

VCVT{*cond*}.*type*.F64 *Dd, Dd*, *#fbits*

VCVT{*cond*}.*type*.F32 *Sd, Sd*, *#fbits*

VCVT{*cond*}.F64.*type Dd, Dd*, *#fbits*

VCVT{*cond*}.F32.*type Sd, Sd*, *#fbits*

인수 설명:

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

type 다음 중 하나일 수 있습니다.

- S16 16비트 부호 있는 고정 소수점 숫자
- U16 16비트 부호 없는 고정 소수점 숫자
- S32 32비트 부호 있는 고정 소수점 숫자
- U32 32비트 부호 없는 고정 소수점 숫자

Sd 피연산자 및 결과의 단정밀도 레지스터입니다.

Dd 피연산자 및 결과의 배정밀도 레지스터입니다.

fbits 고정 소수점 숫자, 0 ~ 16 범위 (*type*이 S16 또는 U16일 경우) 또는 1 ~ 32 범위 (*type*이 S32 또는 U32일 경우) 의 부분 비트 수입니다.

사용법

이 명령어의 처음 두 형식은 부동 소수점을 고정 소수점으로 변환합니다.

세 번째와 네 번째 형식은 고정 소수점을 부동 소수점으로 변환합니다.

모든 경우에 고정 소수점 숫자는 레지스터의 최하위 16비트 또는 32비트에 포함됩니다.

부동 소수점 예외

이러한 명령어는 비정규 입력, 잘못된 연산 또는 정확하지 않은 예외를 생성할 수 있습니다.

5.16.8 VCVTB, VCVTT (반정밀도 확장)

다음과 같은 방식으로 반정밀도 부동 소수점 숫자와 단정밀도 부동 소수점 숫자 간을 변환합니다.

- VCVTB는 단일 워드 레지스터의 하위 하프 (비트[15:0]) 를 사용하여 반정밀도 값을 가져오거나 저장합니다.
- VCVTT는 단일 워드 레지스터의 상위 하프 (비트[31:16]) 를 사용하여 반정밀도 값을 가져오거나 저장합니다.

VCVTB 및 VCVTT는 항상 스칼라입니다.

구문

VCVTB{*cond*}*.type Sd, Sm*

VCVTT{*cond*}*.type Sd, Sm*

인수 설명:

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

type 다음 중 하나일 수 있습니다.

F32.F16 반정밀도에서 단정밀도로 변환합니다.

F16.F32 단정밀도에서 반정밀도로 변환합니다.

Sd 결과의 단일 워드 레지스터입니다.

Sm 피연산자의 단일 워드 레지스터입니다.

아키텍처

이 명령어는 반정밀도 확장을 포함하는 VFPv3 시스템에서만 사용할 수 있습니다.

부동 소수점 예외

이러한 명령어는 비정규 입력, 잘못된 연산, 오버플로, 언더플로 또는 정확하지 않은 예외를 생성할 수 있습니다.

5.16.9 VMOV

부동 소수점 상수를 단정밀도 또는 배정밀도 레지스터에 삽입하거나 한 레지스터를 다른 레지스터에 복사합니다.

이 명령어는 항상 스칼라입니다.

구문

VMOV{*cond*}.F32 *Sd*, *#imm*

VMOV{*cond*}.F64 *Dd*, *#imm*

VMOV{*cond*}.F32 *Sd*, *Sm*

VMOV{*cond*}.F64 *Dd*, *Dm*

인수 설명:

cond 선택적 조건 코드입니다 (5-13페이지의 *조건 코드* 참조).

Sd 단정밀도 대상 레지스터입니다.

Dd 배정밀도 대상 레지스터입니다.

imm 부동 소수점 상수입니다.

Sm 단정밀도 소스 레지스터입니다.

Dm 배정밀도 소스 레지스터입니다.

상수 값

+/–*n* * 2^{-r}로 표시할 수 있는 임의의 숫자입니다. 여기서 *n*은 16에서 31 사이의 정수이고 *r*은 0에서 7 사이의 정수입니다.

아키텍처

이 명령어는 VFPv3에서 사용할 수 있습니다.

5.17 VFP 벡터 모드

대부분의 산술 명령어는 이러한 벡터에서 사용할 수 있고 이를 통해 SIMD (*Single Instruction Multiple Data*) 병렬화를 사용할 수 있습니다. 또한 부동 소수점 로드 및 저장 명령어에는 여러 개의 레지스터 형식이 있고 이를 통해 메모리와의 사이에서 벡터를 전송할 수 있습니다.

VFP 보조 프로세서에 대한 자세한 내용은 *ARM 아키텍처 참조 문서*를 참조하십시오.

참고

VFP 벡터 모드 사용이 제공되지 않습니다.

5.17.1 레지스터 뱅크

VFP 레지스터가 다음과 같이 정렬됩니다.

- 여덟 개의 단정밀도 레지스터 중 네 개의 뱅크, s0 ~ s7, s8 ~ s15, s16 ~ s23 및 s24 ~ s31
- 네 개의 배정밀도 레지스터 중 여덟 개의 뱅크, d0 ~ d3, d4 ~ d7, d8 ~ d11, d12 ~ d15, d16 ~ d19, d20 ~ d23, d24 ~ d27 및 d28 ~ d31 (VFPv2의 경우 네 개)
- 단정밀도 및 배정밀도 레지스터의 혼합

자세한 내용은 그림 5-9 및 5-111페이지의 그림 5-10을 참조하십시오.

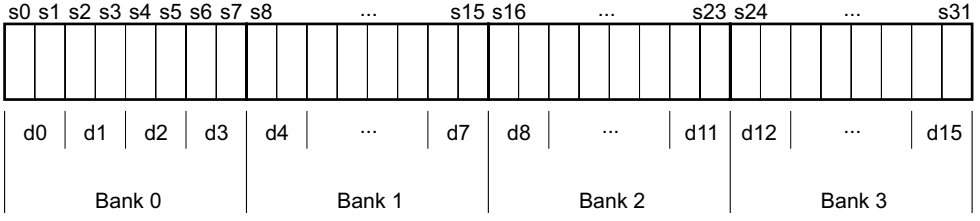

그림 5-9 VFPv2 레지스터 뱅크

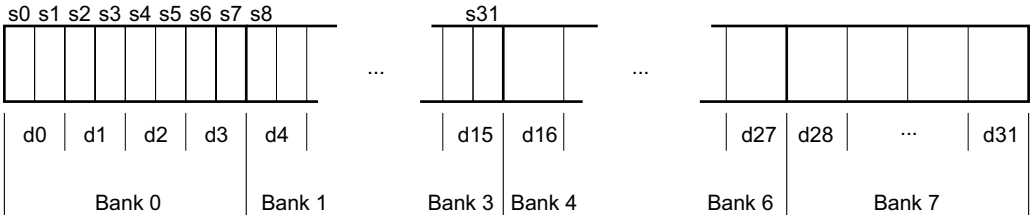

그림 5-10 VFPv3 레지스터 뱅크

5.17.2 벡터

벡터는 동일한 뱅크에서 최대 여덟 개의 단정밀도 레지스터나 네 개의 배정밀도 레지스터를 사용할 수 있습니다. 벡터가 사용하는 레지스터 수는 FPSCR의 LEN 비트로 제어됩니다 (5-93페이지의 *FPSCR, 부동 소수점 상태 및 제어 레지스터* 참조).

벡터는 모든 레지스터에서 시작할 수 있습니다. 벡터가 사용하는 첫 번째 레지스터는 개별 명령어의 레지스터 필드에 지정됩니다.

벡터 래핑

벡터가 뱅크 끝을 넘어서 확장되면 다음과 같이 동일한 뱅크의 시작으로 래핑됩니다.

- s5에서 시작하는 길이가 6인 벡터는 {s5, s6, s7, s0, s1, s2}입니다.
- s15에서 시작하는 길이가 3인 벡터는 {s15, s8, s9}입니다.
- s22에서 시작하는 길이가 4인 벡터는 {s22, s23, s16, s17}입니다.
- d7에서 시작하는 길이 2인 벡터는 {d7, d4}입니다.
- d10에서 시작하는 길이가 3인 벡터는 {d10, d11, d8}입니다.

벡터에는 둘 이상의 뱅크에 속한 레지스터가 포함될 수 없습니다.

벡터 스트라이드

벡터는 위의 예와 같이 연속된 레지스터를 차지하거나 대체 레지스터를 차지할 수 있습니다. 이 동작은 FPSCR의 STRIDE 비트로 제어됩니다 (5-93페이지의 *FPSCR, 부동 소수점 상태 및 제어 레지스터* 참조). 예를 들면 다음과 같습니다.

- s1에서 시작하며 길이와 스트라이드가 각각 3과 2인 벡터는 {s1, s3, s5}입니다.
- s6에서 시작하며 길이와 스트라이드가 각각 4와 2인 벡터는 {s6, s0, s2, s4}입니다.
- d1에서 시작하며 길이와 스트라이드가 둘 다 2인 벡터는 {d1, d3}입니다.

벡터 길이 제한

한 벡터에서 동일한 레지스터를 두 번 사용할 수 없습니다. 즉, 벡터 래핑을 설정하면 다음과 같은 벡터가 생성되지 않습니다.

- 길이가 4보다 길고 스트라이드가 2인 단정밀도 벡터
- 길이가 4보다 길고 스트라이드가 1인 배정밀도 벡터

- 길이가 2보다 길고 스트라이드가 2인 배정밀도 벡터

5.17.3 VFP 벡터 및 스칼라 연산

VFP 산술 명령어를 사용하여 다음에 대해 연산을 수행할 수 있습니다.

- 스칼라
- 벡터
- 스칼라와 벡터 둘 다

FPSCR의 LEN 비트를 사용하여 벡터 길이를 제어할 수 있습니다 (5-93페이지의 *FPSCR, 부동 소수점 상태 및 제어 레지스터* 참조).

LEN이 1이면 모든 연산이 스칼라입니다.

벡터에는 FPSCR의 STRIDE 비트에 따라 한 개 또는 두 개의 *스트라이드*가 있을 수 있습니다. STRIDE가 1이면 벡터 요소는 뱅크에서 연속된 레지스터를 차지합니다. STRIDE가 2이면 벡터 요소는 뱅크에서 대체 레지스터를 차지합니다.

스칼라, 벡터 및 혼합 연산의 제어

LEN이 1보다 크면 산술 연산의 동작은 대상 및 연산 레지스터가 있는 레지스터 뱅크에 따라 결정됩니다 (5-110페이지의 *레지스터 뱅크* 참조).

다음과 같은 형식의 명령어가 제공됩니다.

```
Op   Fd,Fn,Fm
Op   Fd,Fm
```

이러한 명령어는 다음과 같이 동작합니다.

- *Fd*가 s0 ~ s7, d0 ~ d3 또는 d16 ~ d19 레지스터의 첫 번째나 다섯 번째 뱅크에 있으면 스칼라 연산입니다.
- *Fm*이 레지스터의 첫 번째나 다섯 번째 뱅크에 있지만 *Fd*가 해당 뱅크에 없으면 혼합 연산입니다.
- *Fd* 또는 *Fm*이 모두 레지스터의 첫 번째나 다섯 번째 뱅크에 없으면 벡터 연산입니다.

스칼라 연산

*Op*는 *Fm*의 값 및 *Fn*의 값 (있을 경우) 에 대해 작동합니다. 결과는 *Fd*에 배치됩니다.

벡터 연산

*Op*는 *Fm*에서 시작하는 벡터의 값 및 *Fn*에서 시작하는 벡터의 값 (있을 경우) 에 대해 작동합니다. 결과는 *Fd*에서 시작하는 벡터에 배치됩니다.

혼합 스칼라 및 벡터 연산

단일 피연산자 명령어의 경우 *Op*는 *Fm*의 단일 값에 작동합니다. 결과의 LEN 복사본은 *Fd*에서 시작하는 벡터에 배치됩니다.

복수 피연산자 명령어의 경우 *Op*는 *Fm*의 단일 값 및 *Fn*에서 시작하는 벡터의 값에 대해 작동합니다. 결과는 *Fd*에서 시작하는 벡터에 배치됩니다.

5.17.4 VFP 지시어 및 벡터 표시

이 단원에서는 armasm에 대해서만 설명합니다. 이러한 지시어 또는 벡터 표시는 C 및 C++ 컴파일러의 인라인 어셈블러에서 사용할 수 없습니다.

VFP 벡터 모드는 향후 사용할 수 없으며 벡터 표시는 UAL에서 지원되지 않습니다. 벡터 표시를 사용하려면 UAL 이전의 VFP 니모닉을 사용해야 합니다. 자세한 내용은 5-116페이지의 *UAL 이전의 VFP 니모닉*을 참조하십시오. UAL 이전의 VFP 니모닉과 UAL VFP 니모닉을 함께 사용할 수 있습니다.

코드에서 VFP 벡터 길이 및 스트라이드에 대한 어설션을 만들고 어셈블러를 통해 확인할 수 있습니다. 다음을 참조하십시오.

- 5-119페이지의 *VFPASSERT SCALAR*
- 5-120페이지의 *VFPASSERT VECTOR*

VFPASSERT 지시어를 사용할 경우 UAL 이전 니모닉으로 작성된 모든 VFP 데이터 처리 명령어에서 벡터 정보를 지정해야 합니다. 벡터 표시는 5-118페이지의 *벡터 표시*에서 설명합니다. VFPASSERT 지시어를 사용하지 않을 경우 이 표시를 사용하면 안 됩니다.

UAL 이전의 VFP 니모닉

UAL 니모닉에서 .F32를 사용하여 단정밀도 데이터를 지정하는 경우 UAL 이전 니모닉에서는 명령어 니모닉에 추가된 S를 사용합니다. 예를 들어 VABS.32는 FABSS였습니다.

UAL 니모닉에서 .F64를 사용하여 배정밀도 데이터를 지정하는 경우 UAL 이전 니모닉에서는 명령어 니모닉에 추가된 D를 사용합니다. 예를 들어 VCMPE.64는 FCMPED였습니다.

표 5-15에서는 VFP 벡터 모드의 영향을 받은 이러한 명령어의 UAL 이전 니모닉을 보여 줍니다. 다른 모든 VFP 명령어는LEN 및 STRIDE의 설정에 관계없이 항상 스칼라입니다.

표 5-15 UAL 이전의 VFP 니모닉

UAL 니모닉	이에 해당하는 UAL 이전 니모닉
VABS	FABS
VADD	FADD
VDIV	FDIV
VMLA	FMAC
VMLS	FNMAC
VMOV (즉치값)	FCONST [a]
VMOV (레지스터)	FCPY
VMUL	FMUL
VNEG	FNEG
VNMLA	FNMSC
VNMLS	FMSC
VNMUL	FNMUL
VSQRT	FSQRT
VSUB	FSUB

a. VMOV (즉치값) 의 즉치값은 로드할 부동 소수점 숫자이고, FCONST의 즉치값은 로드할 부동 소수점 숫자를 생성하는 명령어에 인코딩된 숫자입니다. 자세한 내용은 5-117페이지의 *FCONST의 즉치값*을 참조하십시오.

FCONST의 즉치값

표 5-16에서는 FCONST를 사용하여 로드할 수 있는 부동 소수점 상수를 보여 줍니다. 뒤에 오는 0은 명확성을 위해 생략되었습니다. FCONST 명령어에 입력해야 하는 즉치값은 이진수 abcdefgh를 십진수로 표현한 것입니다. 다음은 이러한 이진수에 대한 설명입니다.

a 양수일 경우는 0이고 음수일 경우는 1입니다.

bcd 열 제목에 표시됩니다.

efgh 행 제목에 표시됩니다.

또는 앞에 0x가 오는 16진수 표현을 사용할 수도 있습니다.

표 5-16 부동 소수점 상수 값

bcd / efgh	000	001	010	011	100	101	110	111
0000	2.0	4.0	8.0	16.0	0.125	0.25	0.5	1.0
0001	2.125	4.25	8.5	17.0	0.1328125	0.265625	0.53125	1.0625
0010	2.25	4.5	9.0	18.0	0.140625	0.28125	0.5625	1.125
0011	2.375	4.75	9.5	19.0	0.1484375	0.296875	0.59375	1.1875
0100	2.5	5.0	10.0	20.0	0.15625	0.3125	0.625	1.25
0101	2.625	5.25	10.5	21.0	0.1640625	0.328125	0.65625	1.3125
0110	2.75	5.5	11.0	22.0	0.171875	0.34375	0.6875	1.375
0111	2.875	5.75	11.5	23.0	0.1796875	0.359375	0.71875	1.4375
1000	3.0	6.0	12.0	24.0	0.1875	0.375	0.75	1.5
1001	3.125	6.25	12.5	25.0	0.1953125	0.390625	0.78125	1.5625
1010	3.25	6.5	13.0	26.0	0.203125	0.40625	0.8125	1.625
1011	3.375	6.75	13.5	27.0	0.2109375	0.421875	0.84375	1.6875
1100	3.5	7.0	14.0	28.0	0.21875	0.4375	0.875	1.75
1101	3.625	7.25	14.5	29.0	0.2265625	0.453125	0.90625	1.8125
1110	3.75	7.5	15.0	30.0	0.234375	0.46875	0.9375	1.875
1111	3.875	7.75	15.5	31.0	0.2421875	0.484375	0.96875	1.9375

벡터 표시

UAL 이전의 VFP 데이터 처리 명령어에서 각괄호를 사용하여 VFP 레지스터의 벡터를 지정합니다.

- s*n*은 단정밀도 스칼라 레지스터 *n*입니다.
- s*n*<>은 레지스터 *n*에서 시작하며 현재 벡터 길이와 스트라이드를 기준으로 길이와 스트라이드가 지정된 단정밀도 벡터입니다.
- s*n*<*L*>은 레지스터 *n*에서 시작하며 길이와 스트라이드가 각각 L과 1인 단정밀도 벡터입니다.
- s*n*<*L*:*S*>은 레지스터 *n*에서 시작하며 길이와 스트라이드가 각각 *L*과 *S*인 단정밀도 벡터입니다.
- d*n*은 배정밀도 스칼라 레지스터 *n*입니다.
- d*n*<>은 레지스터*n*에서 시작하며 현재 벡터 길이와 스트라이드를 기준으로 길이와 스트라이드가 지정된 배정밀도 벡터입니다.
- d*n*<*L*>은 레지스터 *n*에서 시작하며 길이와 스트라이드가 각각 L과 1인인 배정밀도 벡터입니다.
- d*n*<*L*:*S*>는 레지스터 *n*에서 시작하며 길이와 스트라이드가 각각 *L*과 *S*인 배정밀도 벡터입니다.

DN 및 SN 지시어를 사용하여 정의한 이름에 이 벡터 표시를 사용할 수 있습니다 (7-14페이지의 *QN, DN 및 SN* 참조).

DN 및 SN 지시어 자체에 이 벡터 표시를 사용하면 안 됩니다.

VFPASSERT SCALAR

VFPASSERT SCALAR 지시어는 어셈블러에 다음 VFP 명령어가 스칼라 모드에 있음을 알립니다.

구문

```
VFPASSERT SCALAR
```

사용법

VFPASSERT SCALAR 지시어를 사용하여 VFP 모드가 VECTOR인 코드 블록의 끝을 표시할 수 있습니다.

VFPASSERT SCALAR 지시어를 변경이 발생한 명령어 바로 뒤에 배치합니다. 일반적으로 FMXR 명령어가 사용되지만 BL 명령어가 사용될 수 있습니다.

함수에서 VFP가 종료 시 벡터 모드에 있도록 요구할 경우 VFPASSERT SCALAR 지시어를 마지막 명령어 바로 뒤에 배치합니다. 이러한 함수는 AAPCS를 준수하지 않습니다. 자세한 내용은 *install_directory*\Documentation\Specifications\...에 있는 *Procedure Call Standard for the ARM Architecture* 사양 (aapcs.pdf) 을 참조하십시오.

추가 참고:

- 5-118페이지의 *벡터 표시*
- 5-120페이지의 *VFPASSERT VECTOR*

참고

이 지시어는 코드를 생성하지 않으며 단지 프로그래머에 의한 어설션일 뿐입니다. 어셈블러에서는 이러한 어설션이 서로 일치하지 않거나 VFP 데이터 처리 명령어의 벡터 표시와 일치하지 않을 경우 오류 메시지를 생성합니다.

어셈블러에서는 벡터 길이가 1인 경우에도 VFPASSERT SCALAR 지시어 다음에 나오는 VFP 데이터 처리 명령어의 벡터 표시에 대해 오류를 발생시킵니다.

예제

```
VFPASSERT   SCALAR           ; scalar mode
faddd       d4, d4, d0       ; okay
fadds       s4<3>, s0, s8<3> ; ERROR, vector in scalar mode
fabss       s24<1>, s28<1>   ; ERROR, vector in scalar mode
                             ; (even though length==1)
```

VFPASSERT VECTOR

VFPASSERT VECTOR 지시어는 어셈블러에 다음 VFP 명령어가 벡터 모드에 있음을 알립니다. 또한 벡터의 길이 및 스트라이드를 지정할 수 있습니다.

구문

VFPASSERT VECTOR[<[*n*[:*s*]]>]

인수 설명:

n 1 ~ 8의 벡터 길이입니다.

s 1 ~ 2의 벡터 스트라이드입니다.

사용법

VFPASSERT VECTOR 지시어를 사용하여 VFP 모드가 VECTOR인 명령어 블록의 시작을 표시하고 벡터의 길이 또는 스트라이드에 대한 변경 내용을 표시할 수 있습니다.

VFPASSERT VECTOR 지시어를 변경이 발생한 명령어 바로 뒤에 배치합니다. 일반적으로 FMXR 명령어가 사용되지만 BL 명령어가 사용될 수 있습니다.

함수에서 VFP가 시작 시 벡터 모드에 있도록 요구할 경우 VFPASSERT VECTOR 지시어를 마지막 명령어 바로 뒤에 배치합니다. 이러한 함수는 AAPCS를 준수하지 않습니다. 자세한 내용은 *install_directory*\Documentation\Specifications\...에 있는 *Procedure Call Standard for the ARM Architecture* 사양 (aapcs.pdf) 을 참조하십시오.

참조:

- 5-118페이지의 *벡터 표시*
- 5-119페이지의 *VFPASSERT SCALAR*

참고

이 지시어는 코드를 생성하지 않으며 단지 프로그래머에 의한 어설션일 뿐입니다. 어셈블러에서는 이러한 어설션이 서로 일치하지 않거나 VFP 데이터 처리 명령어의 벡터 표시와 일치하지 않을 경우 오류 메시지를 생성합니다.

예제

```
    VMRS    r10,FPSCR          ; UAL mnemonic - could be FMRX instead.
    BIC     r10,r10,#0x00370000
    ORR     r10,r10,#0x00020000    ; set length = 3, stride = 1
    VMSR    FPSCR,r10
    VFPASSERT VECTOR           ; assert vector mode, unspecified length & stride
    faddd  d4, d4, d0          ; ERROR, scalar in vector mode
```

```
fadds  s16<3>, s0, s8<3>   ; okay
fabss  s24<1>, s28<1>      ; wrong length, but not faulted (unspecified)
VMRS    r10,FPSCR
BIC     r10,r10,#0x00370000
ORR     r10,r10,#0x00030000    ; set length = 4, stride = 1
VMSR    FPSCR,r10
VFPASSERT VECTOR<4>        ; assert vector mode, length 4, stride 1
fadds  s24<4>, s0, s8<4>   ; okay
fabss  s24<2>, s24<2>      ; ERROR, wrong length
VMRS    r10,FPSCR
BIC     r10,r10,#0x00370000
ORR     r10,r10,#0x00130000    ; set length = 4, stride = 2
VMSR    FPSCR,r10
VFPASSERT VECTOR<4:2>      ; assert vector mode, length 4, stride 2
fadds  s8<4>, s0, s16<4>   ; ERROR, wrong stride
fabss  s16<4:2>, s28<4:2>  ; okay
fadds  s8<>, s2, s16<>     ; okay (s8 and s16 both have
                           ; length 4 and stride 2.
                           ; s2 is scalar.)
```

6장
Wireless MMX 기술 명령어

이 장에서는 Wireless MMX™ 기술 명령어 지원을 설명합니다. 이 장은 다음 단원으로 구성되어 있습니다.

- 6-2페이지의 *소개*
- 6-3페이지의 *Wireless MMX 기술에 대한 ARM 지원*
- 6-9페이지의 *Wireless MMX 명령어*

6.1 소개

Wireless MMX 기술은 일부 멀티미디어 응용 프로그램의 성능을 향상시키는 XScale 프로세서에서 사용할 수 있는 SIMD (*Single Instruction Multiple Data*) 명령어 세트입니다. Wireless MMX 기술은 64비트 레지스터를 통해 패킹된 형식의 여러 데이터 요소에 대해 작동합니다.

Wireless MMX 기술은 ARM 보조 프로세서 0 및 1을 사용하여 해당 명령어 세트와 데이터 유형을 지원합니다. Wireless MMX 기술 명령어를 사용하는 소스 코드를 어셈블하여 PXA270 프로세서에서 실행할 수 있습니다.

Wireless MMX 2 기술은 Wireless MMX 기술의 업데이트된 버전입니다.

ARM 어셈블러를 사용할 때는 다음 사항에 유의해야 합니다.

- Wireless MMX 기술 명령어는 지원되는 프로세서 (`armasm --cpu PXA270`) 를 지정한 경우에만 어셈블할 수 있습니다.
- PXA270 프로세서는 ARM 또는 Thumb® 으로 작성된 코드만 지원합니다.
- 대부분의 Wireless MMX 기술 명령어는 ARM 플래그 상태에 따라 조건부로 추출할 수 있습니다. Wireless MMX 기술 조건 코드는 ARM 조건 코드와 동일합니다.

이 장에서는 RealView Compilation Tools의 ARM 어셈블러에서 제공하는 Wireless MMX 기술 지원에 대해 설명하고 Wireless MMX 기술에 대해서는 자세히 설명하지 않습니다. 프로그래머 모델과 Wireless MMX 기술 명령어 세트에 대한 자세한 내용은 *Wireless MMX Technology Developer Guide*를 참조하십시오.

6.2 Wireless MMX 기술에 대한 ARM 지원

이 단원에서는 어셈블러에서 제공하는 Wireless MMX 및 MMX 2 기술 지원에 대해 설명합니다. 이 단원에는 다음 소단원이 포함되어 있습니다.

- *레지스터*
- 6-4페이지의 *WRN 및 WCN 지시문*
- 6-5페이지의 *프레임 지시문*
- 6-6페이지의 *Wireless MMX 로드 및 저장 명령어*
- 6-8페이지의 *Wireless MMX 기술 및 XScale 명령어*

6.2.1 레지스터

Wireless MMX 기술은 다음 두 레지스터 유형을 지원합니다.

상태 및 제어 레지스터

제어 레지스터는 보조 프로세서 1로 매핑되고 범용 레지스터 wCGR0 - wCGR3 및 SIMD 플래그를 포함합니다. 이러한 레지스터에 대한 자세한 내용은 표 6-1을 참조하십시오.

Wireless MMX 기술 명령어 TMCR 및 TMRC를 사용하면 이러한 레지스터에서 읽기 및 쓰기를 수행할 수 있습니다.

표 6-1 상태 및 제어 레지스터

형식	Wireless MMX 기술 레지스터	CP1 레지스터
보조 프로세서 ID	wCID	c0
제어	wCon	c1
포화 SIMD 플래그	wCSSF	c2
산술 SIMD 플래그	wCASF	c3
예약되어 있음	-	c4 - c7
범용	wCGR0 - wCGR3	c8 - c11
예약되어 있음	-	c12 - c15

SIMD 데이터 레지스터

데이터 레지스터 (wR0 - wR15) 는 보조 프로세서 0으로 매핑되고 16 x 64비트로 패킹된 데이터를 포함합니다. Wireless MMX 기술 의사 명령어 TMRRC 및 TMCRR을 사용하면 이러한 레지스터와 ARM 레지스터 간에 데이터를 이동할 수 있습니다.

레지스터에 대한 자세한 내용은 *Wireless MMX Technology Developer Guide*를 참조하십시오.

Wireless MMX 기술 명령어를 어셈블할 때 어셈블러에서는 다음과 같은 레지스터 사양을 사용합니다.

- wR0, wCID, wCon 등과 같이 Wireless MMX 기술 사양과 정확히 일치하도록 대소문자 혼용
- wr0, wcid, wcon 등과 같이 대문자만 사용
- WR0, WCID, WCON 등과 같이 대문자만 사용

어셈블러에서는 사용자 고유의 이름을 지정할 수 있도록 WRN 및 WCN 지시문 (*WRN 및 WCN 지시문* 참조) 을 지원합니다.

6.2.2 WRN 및 WCN 지시문

다음과 같은 지시문을 사용하여 Wireless MMX 기술을 지원할 수 있습니다.

WCN 지정된 제어 레지스터의 이름을 다음과 같이 정의합니다.

```
speed WCN  wcgr0 ; defines speed as a symbol for control reg 0
```

WRN 지정된 SIMD 데이터 레지스터의 이름을 다음과 같이 정의합니다.

```
rate WRN  wr6 ; defines rate as a symbol for data reg 6
```

한 레지스터에 여러 이름을 사용하면 안 됩니다. 3-23페이지의 *미리 정의된 레지스터 및 보조 프로세서 이름*에 나와 있는 미리 정의된 이름이나 6-3페이지의 *레지스터*에 설명된 레지스터 이름을 사용하면 안 됩니다.

6.2.3 프레임 지시문

Wireless MMX 기술 레지스터는 FRAME 지시문과 함께 일반적인 방법으로 개체 파일에 디버그 정보를 추가하는 데 사용할 수 있습니다 (7-41페이지의 *프레임 지시어* 참조). 다음 제한 사항에 유의하십시오.

- Wireless MMX 기술 레지스터 wR0 - wR9 또는 wCGR0 - wCGR3을 스택에 푸시하려고 하면 경고가 표시됩니다 (7-46페이지의 *FRAME PUSH* 참조).
- Wireless MMX 기술 레지스터를 주소 오프셋으로 사용할 수 없습니다 (7-43페이지의 *FRAME ADDRESS* 및 7-50페이지의 *FRAME RETURN ADDRESS* 참조).

6.2.4 Wireless MMX 로드 및 저장 명령어

Wireless MMX 보조 프로세서 레지스터에서 바이트, 하프워드, 워드 또는 더블워드를 로드 및 저장합니다.

구문

```
op<type>{cond} wRd, [Rn, #{-}offset]{!}

op<type>{cond} wRd, [Rn] {, #{-}offset}

opW{cond} wRd, label

opW wCd, [Rn, #{-}offset]{!}

opW wCd, [Rn] {, #{-}offset}

opD{cond} wRd, label

opD wRd, [Rn, {-}Rm {, LSL #imm4}]{!}      ; MMX2 only

opD wRd, [Rn], {-}Rm {, LSL #imm4}         ; MMX2 only
```

인수 설명:

op 다음 중 하나일 수 있습니다.

- WLDR Wireless MMX 레지스터 로드
- WSTR Wireless MMX 레지스터 저장

<type> 다음 중 하나일 수 있습니다.

- B 바이트
- H 하프워드
- W 워드
- D 더블워드

cond 선택적 조건 코드입니다 (2-20페이지의 *조건부 실행* 참조).

wRd 로드 또는 저장할 Wireless MMX SIMD 데이터 레지스터입니다.

wCd 로드 또는 저장할 Wireless MMX 상태 및 제어 레지스터입니다.

Rn 메모리 주소의 기준이 되는 레지스터입니다.

offset 즉치 오프셋입니다. 오프셋이 생략될 경우 명령어는 0 오프셋 명령어입니다.

!　　선택적 접미사입니다. ! 기호가 있을 경우 명령어는 사전 인덱싱된 명령어입니다.

label　　프로그램 기준 식입니다. 자세한 내용은 3-38페이지의 *레지스터 기준 및 프로그램 기준 식*을 참조하십시오.

*label*은 현재 명령어의 +/- 1020바이트 내에 있어야 합니다.

Rm　　오프셋으로 사용할 값이 포함된 레지스터입니다. *Rm* 은 r15이면 안 됩니다.

imm4　　0 ~ 15 범위에서 *Rm*을 왼쪽으로 시프트할 비트 수를 포함합니다.

SIMD 레지스터에 상수 로드

또한 어셈블러는 다음과 같은 WLDRW 및 WLDRD 리터럴 로드 의사 명령어도 지원합니다.

```
WLDRW wr0, =0x114
```

다음 사항에 유의하십시오.

- 어셈블러에서 바이트 및 하프워드 리터럴을 로드할 수 없습니다. 이 경우 다운그레이드할 수 있는 오류가 생성됩니다. 다운그레이드되면 명령어가 WLDRW로 변환되고 32비트 리터럴이 생성됩니다. 이것은 32비트 워드를 사용한다는 점을 제외하면 바이트 리터럴 로드와 같습니다.
- 로드할 리터럴이 0이고 대상이 SIMD 데이터 레지스터이면 어셈블러에서 명령어를 WZERO로 변환합니다.
- 8바이트로 정렬되지 않은 더블워드는 예상할 수 없습니다.

6.2.5 Wireless MMX 기술 및 XScale 명령어

Wireless MMX 기술 명령어와 XScale 명령어가 서로 겹치므로 충돌을 피하기 위해 어셈블러에 다음 제한이 적용됩니다.

- 동일한 어셈블리에서 XScale 명령어와 Wireless MMX 기술 명령어를 함께 사용할 수 없습니다.
- Wireless MMX 기술 `TMIA` 명령어에 XScale `MIA` 명령어와 겹치는 `MIA` 니모닉이 있습니다. 다음 사항에 유의하십시오.
  - `MIA acc0, Rm, Rs` 는 XScale에서만 사용할 수 있고 Wireless MMX 기술에서 사용하면 오류가 발생합니다.
  - `MIA wR0, Rm, Rs` 및 `TMIA wR0, Rm, Rs`는 Wireless MMX 기술에서 사용할 수 있습니다.
  - `TMIA acc0, Rm, Rs`를 XScale에서 사용하면 오류가 발생합니다. XScale에는 `TMIA` 명령어가 없습니다.

XScale 명령어에 대한 자세한 내용은 4-90페이지의 *MIA, MIAPH 및 MIAxy* 및 4-147페이지의 *MAR 및 MRA*를 참조하십시오.

6.3 Wireless MMX 명령어

표 6-2에서는 Wireless MMX 기술 명령어 세트의 목록을 보여 줍니다. 이 표를 사용하면 *Wireless MMX Technology Developer Guide*에서 설명하는 개별 명령어를 찾을 수 있습니다. 의사 명령어 (6-11페이지의 표 6-3) 도 참조하십시오.

이 단원에서는 Wireless MMX 기술 레지스터를 *wRn*, *wRd*라고 하고 ARM 레지스터를 *Rn*, *Rd*라고 합니다.

표 6-2 Wireless MMX 기술 명령어

니모닉	예제
TANDC	TANDCB r15
TBCST	TBCSTB wr15, r1
TEXTRC	TEXTRCB r15, #0
TEXTRM	TEXTRMUBCS r3, wr7, #7
TINSR	TINSRB wr6, r11, #0
TMIA, TMIAPH, TMIAxy	TMIANE wr1, r2, r3 TMIAPH wr4, r5, r6 TMIABB wr4, r5, r6 MIAPHNE wr4, r5, r6
TMOVMSK	TMOVMSKBNE r14, wr15
TORC	TORCB r15
WACC	WACCBGE wr1, wr2
WADD	WADDBGE wr1, wr2, wr13
WALIGNI, WALIGNR	WALIGNI wr7, wr6, wr5,#3 WALIGNR0 wr4, wr8, wr12
WAND, WANDN	WAND wr1, wr2, wr3 WANDN wr5, wr5, wr9
WAVG2	WAVG2B wr3, wr6, wr9 WAVG2BR wr4, wr7, wr10
WCMPEQ	WCMPEQB wr0, wr4, wr2
WCMPGT	WCMPGTUB wr0, wr4, wr2
WLDR	WLDRB wr1, [r2, #0]

표 6-2 Wireless MMX 기술 명령어 (계속)

니모닉	예제
WMAC	WMACU wr3, wr4, wr5
WMADD	WMADDU wr3, wr4, wr5
WMAX, WMIN	WMAXUB wr0, wr4, wr2 WMINSB wr0, wr4, wr2
WMUL	WMULUL wr4, wr2, wr3
WOR	WOR wr3, wr1, wr4
WPACK	WPACKHUS wr2, wr7, wr1
WROR	WRORH wr3, wr1, wr4
WSAD	WSADB wr3, wr5, wr8
WSHUFH	WSHUFH wr8, wr15, #17
WSLL, WSRL	WSLLH wr3, wr1, wr4 WSRLHG wr3, wr1, wcgr0
WSRA	WSRAH wr3, wr1, wr4 WSRAHG wr3, wr1, wcgr0
WSTR	WSTRB wr1, [r2, #0] WSTRW wc1, [r2, #0]
WSUB	WSUBBGE wr1, wr2, wr13
WUNPCKEH, WUNPCKEL	WUNPCKEHUB wr0, wr4 WUNPCKELSB wr0, wr4
WUNPCKIH, WUNPCKIL	WUNPCKIHB wr0, wr4, wr2 WUNPCKILH wr1, wr5, wr3
WXOR	WXOR wr3, wr1, wr4

6.3.1 의사 명령어

표 6-3에서는 Wireless MMX 기술 의사 명령어에 대해 간략히 설명합니다. 이 표를 사용하면 *Wireless MMX Technology Developer Guide*와 4장 *ARM 및 Thumb 명령어*에서 설명하는 개별 명령어를 찾을 수 있습니다.

표 6-3 Wireless MMX 기술 의사 명령어

니모닉	간단한 설명	예제
TMCR	소스 레지스터 *Rn*의 내용을 제어 레지스터*wCn*으로 이동합니다. ARM MCR 보조 프로세서 명령어로 매핑합니다 (4-125페이지 참조).	TMCR wc1, r10
TMCRR	두 소스 레지스터 *RnLo* 및 *RnHi*의 내용을 대상 레지스터 *wRd*로 이동합니다. r15를 *RnLo* 또는 *RnHi*에 사용하면 안 됩니다. ARM MCRR 보조 프로세서 명령어로 매핑합니다 (4-125페이지 참조).	TMCRR wr4, r5, r6
TMRC	제어 레지스터 *wCn*의 내용을 대상 레지스터 *Rd*로 이동합니다. *Rd*에 r15를 사용하면 안 됩니다. ARM MRC 보조 프로세서 명령어로 매핑합니다 (4-127페이지 참조).	TMRC r1, wc2
TMRRC	소스 레지스터 *wRn*의 내용을 두 개의 대상 레지스터 *RdLo* 및 *RdHi*로 이동합니다. 두 대상 레지스터에 모두 r15를 사용하면 안 됩니다. *RdLo* 및 *RdHi*는 고유한 레지스터여야 하며, 그렇지 않으면 결과를 예측할 수 없습니다. ARM MRRC 보조 프로세서 명령어로 매핑합니다 (4-127페이지 참조).	TMRRC r1, r0, wr2
WMOV	소스 레지스터 *wRn*의 내용을 대상 레지스터 *wRd*로 이동합니다. 이 명령어는 WOR 형식입니다 (6-9페이지의 표 6-2 참조).	WMOV wr1, wr8
WZERO	대상 레지스터 *wRd*를 0으로 설정합니다. 이 명령어는 WANDN 형식입니다 (6-9페이지의 표 6-2 참조).	WZERO wr1

7장
지시어 참조

이 장에서는 ARM® 어셈블러인 armasm에서 제공하는 지시어에 대해 설명합니다. 이 장은 다음 단원으로 구성되어 있습니다.

- 7-2페이지의 *지시어의 사전순 목록*
- 7-4페이지의 *기호 정의 지시어*
- 7-16페이지의 *데이터 정의 지시어*
- 7-32페이지의 *어셈블리 제어 지시어*
- 7-41페이지의 *프레임 지시어*
- 7-57페이지의 *보고 지시어*
- 7-63페이지의 *명령어 세트 및 구문 선택 지시어*
- 7-66페이지의 *기타 지시어*

참고

이러한 지시어는 ARM C 및 C++ 컴파일러의 인라인 어셈블러에서 사용할 수 없습니다.

7.1 지시어의 사전순 목록

표 7-1에서는 지시어의 전체 목록을 보여 줍니다. 이 표를 사용하여 개별 지시어를 찾을 수 있습니다.

표 7-1 지시어 위치

지시어	페이지	지시어	페이지	지시어	페이지
ALIGN	7-67페이지	EXPORT 또는 GLOBAL	7-78페이지	MACRO 및 MEND	7-33페이지
ARM 및 CODE32	7-64페이지	EXPORTAS	7-80페이지	MAP	7-19페이지
AREA	7-70페이지	EXTERN	7-82페이지	MEND (MACRO 참조)	7-33페이지
ASSERT	7-57페이지	FIELD	7-20페이지	MEXIT	7-36페이지
ATTR	7-74페이지	FRAME ADDRESS	7-43페이지	NOFP	7-86페이지
CN	7-12페이지	FRAME POP	7-45페이지	OPT	7-60페이지
CODE16	7-64페이지	FRAME PUSH	7-46페이지	PRESERVE8 (REQUIRE8 참조)	7-87페이지
COMMON	7-31페이지	FRAME REGISTER	7-48페이지	PROC (FUNCTION 참조)	7-55페이지
CP	7-13페이지	FRAME RESTORE	7-49페이지	QN	7-14페이지
DATA	7-31페이지	FRAME SAVE	7-51페이지	RELOC	7-9페이지
DCB	7-22페이지	FRAME STATE REMEMBER	7-52페이지	REQUIRE	7-86페이지
DCD 및 DCDU	7-23페이지	FRAME STATE RESTORE	7-53페이지	REQUIRE8 및 PRESERVE8	7-87페이지
DCDO	7-24페이지	FRAME UNWIND ON 또는 OFF	7-54페이지	RLIST	7-11페이지

표 7-1 지시어 위치 (계속)

지시어	페이지	지시어	페이지	지시어	페이지
DCFD 및 DCFDU	7-25페이지	FUNCTION 또는 PROC	7-55페이지	RN	7-10페이지
DCFS 및 DCFSU	7-26페이지	GBLA, GBLL 및 GBLS	7-5페이지	ROUT	7-88페이지
DCI	7-27페이지	GET 또는 INCLUDE	7-81페이지	SETA, SETL 및 SETS	7-8페이지
DCQ 및 DCQU	7-29페이지	GLOBAL (EXPORT 참조)	7-78페이지	SN	7-14페이지
DCW 및 DCWU	7-30페이지	IF, ELSE, ENDIF 및 ELIF	7-37페이지	SPACE 또는 FILL	7-21페이지
DN	7-14페이지	IMPORT	7-82페이지	SUBT	7-62페이지
ELIF 및 ELSE (IF 참조)	7-37페이지	INCBIN	7-84페이지	THUMB	7-64페이지
END	7-76페이지	INCLUDE (GET 참조)	7-81페이지	THUMBX	7-64페이지
ENDFUNC 또는 ENDP	7-56페이지	INFO	7-59페이지	TTL	7-62페이지
ENDIF (IF 참조)	7-37페이지	KEEP	7-85페이지	WHILE 및 WEND	7-40페이지
ENTRY	7-76페이지	LCLA, LCLL 및 LCLS	7-7페이지		
EQU	7-77페이지	LTORG	7-18페이지		

7.2 기호 정의 지시어

이 단원에서는 다음 지시어에 대해 설명합니다.

- 7-5페이지의 *GBLA, GBLL 및 GBLS*
  전역 산술, 논리 또는 문자열 변수를 선언합니다.
- 7-7페이지의 *LCLA, LCLL 및 LCLS*
  논리 산술, 논리 또는 문자열 변수를 선언합니다.
- 7-8페이지의 *SETA, SETL 및 SETS*
  산술, 논리 또는 문자열 변수의 값을 설정합니다.
- 7-9페이지의 *RELOC*
  객체 파일에서 ELF 재배치를 인코딩합니다.
- 7-10페이지의 *RN*
  지정한 레지스터의 이름을 정의합니다.
- 7-11페이지의 *RLIST*
  범용 레지스터 세트의 이름을 정의합니다.
- 7-12페이지의 *CN*
  보조 프로세서 레지스터 이름을 정의합니다.
- 7-13페이지의 *CP*
  보조 프로세서 이름을 정의합니다.
- 7-14페이지의 *QN, DN 및 SN*
  배정밀도 또는 단정밀도 VFP 레지스터 이름을 정의합니다.

7.2.1 GBLA, GBLL 및 GBLS

GBLA 지시어는 전역 산술 변수를 선언하고 해당 값을 0으로 초기화합니다.

GBLL 지시어는 전역 논리 변수를 선언하고 해당 값을 {FALSE}로 초기화합니다.

GBLS 지시어는 전역 문자열 변수를 선언하고 해당 값을 Null 문자열 ("") 로 초기화합니다.

구문

<gblx> variable

인수 설명:

<gblx> GBLA, GBLL 또는 GBLS 중 하나입니다.

variable 변수 이름입니다. *variable*은 소스 파일 내의 기호 간에 고유해야 합니다.

사용법

이미 정의된 변수에 대해 이러한 지시어 중 하나를 사용하면 위에 나온 것과 동일한 값으로 다시 초기화됩니다.

변수 범위는 해당 변수가 들어 있는 소스 파일로 제한됩니다.

SETA, SETL 또는 SETS 지시어 (7-8페이지의 *SETA, SETL 및 SETS* 참조) 을 사용하여 변수 값을 설정합니다.

지역 변수 선언에 대한 자세한 내용은 7-7페이지의 *LCLA, LCLL 및 LCLS*를 참조하십시오.

전역 변수는 --predefine 어셈블러 명령 행 옵션을 사용하여 설정할 수도 있습니다. 자세한 내용은 3-2페이지의 *명령 구문*을 참조하십시오.

예제

예제 7-1에서는 objectsize 변수를 선언하고 objectsize 값을 0xFF로 설정한 다음 나중에 SPACE 지시어에 사용합니다.

예제 7-1

```
            GBLA    objectsize    ; declare the variable name
objectsize  SETA    0xFF          ; set its value
            .
            .                     ; other code
            .
            SPACE   objectsize    ; quote the variable
```

예제 7-2에서는 armasm을 호출할 때 변수를 선언하고 설정하는 방법을 보여 줍니다. 어셈블리 타임에 변수 값을 설정하려는 경우 이 예제 코드를 사용하십시오. --pd는 --predefine의 동의어입니다.

예제 7-2

```
armasm --predefine "objectsize SETA 0xFF" sourcefile -o objectfile
```

7.2.2 LCLA, LCLL 및 LCLS

LCLA 지시어는 지역 산술 변수를 선언하고 해당 값을 0으로 초기화합니다.

LCLL 지시어는 지역 논리 변수를 선언하고 해당 값을 {FALSE}로 초기화합니다.

LCLS 지시어는 지역 문자열 변수를 선언하고 해당 값을 Null 문자열 ("") 로 초기화합니다.

구문

<lclx> variable

인수 설명:

<lclx> LCLA, LCLL 또는 LCLS 중 하나입니다.

variable 변수의 이름입니다. *variable*은 자신이 포함된 매크로 내에서 고유해야 합니다.

사용법

이미 정의된 변수에 대해 이러한 지시어 중 하나를 사용하면 위에 나온 것과 동일한 값으로 다시 초기화됩니다.

변수 범위는 해당 변수가 들어 있는 매크로의 특정 인스턴스화로 제한됩니다 (7-33페이지의 *MACRO 및 MEND* 참조).

SETA, SETL 또는 SETS 지시어 (7-8페이지의 *SETA, SETL 및 SETS* 참조) 을 사용하여 변수 값을 설정합니다.

전역 변수 선언에 대한 자세한 내용은 7-5페이지의 *GBLA, GBLL 및 GBLS*를 참조하십시오.

예제

```
        MACRO                           ; Declare a macro
$label  message $a                      ; Macro prototype line
        LCLS    err                     ; Declare local string
                                        ; variable err.
err     SETS    "error no: "            ; Set value of err
$label  ; code
        INFO    0, "err":CC::STR:$a     ; Use string
        MEND
```

7.2.3 SETA, SETL 및 SETS

SETA 지시어는 지역 또는 전역 산술 변수의 값을 설정합니다.

SETL 지시어는 지역 또는 전역 논리 변수의 값을 설정합니다.

SETS 지시어는 지역 또는 전역 문자열 변수의 값을 설정합니다.

구문

variable <setx> expr

인수 설명:

<setx> SETA, SETL 또는 SETS 중 하나입니다.

variable GBLA, GBLL, GBLS, LCLA, LCLL 또는 LCLS 지시어에 의해 선언된 변수 이름입니다.

expr 다음 중 하나입니다.

- SETA의 경우, 숫자 식 (3-35페이지의 *숫자 식* 참조) 입니다.
- SETL의 경우, 논리 식 (3-38페이지의 *논리 식* 참조) 입니다.
- SETS의 경우, 문자열 식 (3-34페이지의 *문자열 식* 참조) 입니다.

사용법

이러한 지시어 중 하나를 사용하기 전에 전역 또는 지역 선언 지시어를 사용하여 *variable*을 선언해야 합니다. 자세한 내용은 7-5페이지의 *GBLA, GBLL 및 GBLS* 및 7-7페이지의 *LCLA, LCLL 및 LCLS*를 참조하십시오.

명령 행에서 변수 이름을 미리 정의할 수도 있습니다. 자세한 내용은 3-2페이지의 *명령 구문*을 참조하십시오.

예제

```
                GBLA    VersionNumber
VersionNumber   SETA    21
                GBLL    Debug
Debug           SETL    {TRUE}
                GBLS    VersionString
VersionString   SETS    "Version 1.0"
```

7.2.4 RELOC

RELOC 지시어는 객체 파일에서 ELF 재배치를 명시적으로 인코딩합니다.

구문

RELOC *n*, *symbol*

RELOC *n*

인수 설명:

n 0 ~ 255 범위에 있어야 합니다.

symbol 임의의 프로그램 상대 레이블일 수 있습니다.

사용법

RELOC *n*, *symbol*을 사용하여 *symbol*로 레이블이 지정된 주소와 관련된 재배치를 만듭니다.

RELOC를 ARM 또는 Thumb 명령어 바로 다음에 사용할 경우 해당 명령어에 재배치가 만들어집니다. RELOC를 DCB, DCW, DCD 또는 다른 모든 데이터 생성 지시어 바로 다음에 사용할 경우 데이터의 시작 부분에 재배치가 만들어집니다. 적용할 모든 가수는 명령어나 DCI 또는 DCD에 인코딩해야 합니다.

어셈블러가 해당 위치에서 재배치를 이미 만든 경우 이 재배치는 RELOC 지시어의 세부 정보로 업데이트됩니다. 예를 들면 다음과 같습니다.

```
DCD     sym2 ; R_ARM_ABS32 to sym32
RELOC   55   ; ... makes it R_ARM_ABS32_NOI
```

그 밖의 모든 경우에는 RELOC를 사용하면 오류가 발생합니다. 예를 들어 이 지시어를 LTORG 또는 ALIGN 같이 데이터를 생성하지 않는 지시어 다음에 사용하거나 AREA에서 맨 처음 사용하면 오류가 발생합니다.

RELOC *n*을 사용하여 익명 기호 즉, 익명 테이블의 기호 0과 관련된 재배치를 만듭니다. 위의 어셈블러가 만든 재배치 없이 RELOC *n*을 사용하면 익명 기호와 관련된 재배치가 만들어집니다. 자세한 내용은 *ARM 아키텍처용 응용 프로그램 바이너리 인터페이스*를 참조하십시오.

예제

```
IMPORT  impsym
LDR     r0,[pc,#-8]
RELOC   4, impsym
DCD     0
RELOC   2, sym
DCD     0,1,2,3,4  ; the final word is relocated
RELOC   38,sym2    ; R_ARM_TARGET1
```

7.2.5 RN

RN 지시어는 지정한 레지스터의 레지스터 이름을 정의합니다.

구문

name RN *expr*

인수 설명:

name 레지스터에 지정할 이름입니다. *name*은 3-23페이지의 *미리 정의된 레지스터 및 보조 프로세서 이름*에 나와 있는 미리 정의된 이름과 같을 수 없습니다.

expr 0에서 15 사이의 레지스터 번호로 평가됩니다.

사용법

RN을 사용하여 각 레지스터의 용도를 기억하기 쉽도록 레지스터에 적절한 이름을 지정합니다. 한 레지스터에 여러 이름을 사용하지 않도록 주의해야 합니다.

예제

```
regname     RN  11  ; defines regname for register 11
sqr4        RN  r6  ; defines sqr4 for register 6
```

7.2.6 RLIST

RLIST (레지스터 목록) 지시어는 범용 레지스터 세트에 이름을 지정합니다.

구문

name RLIST {*list-of-registers*}

인수 설명:

name 레지스터 세트에 지정할 이름입니다. *name*은 3-23페이지의 *미리 정의된 레지스터 및 보조 프로세서 이름*에 나와 있는 미리 정의된 이름과 같을 수 없습니다.

list-of-registers

레지스터 이름 및/또는 레지스터 범위를 콤마로 구분하여 나열한 목록입니다. 레지스터 목록은 중괄호로 묶어야 합니다.

사용법

RLIST를 사용하여 LDM 또는 STM 명령어를 통해 전송할 레지스터 세트에 이름을 지정합니다.

LDM 및 STM은 해당 LDM 또는 STM 명령어에 레지스터가 제공되는 순서에 관계없이 항상 최하위 메모리 주소에 최하위 물리 레지스터 번호를 배치합니다. 사용자 고유의 기호 레지스터 이름을 정의한 경우 레지스터 목록이 오름차순으로 되어 있지 않으므로 명확성이 떨어질 수 있습니다.

--diag_warning 1206 어셈블러 옵션을 사용하여 레지스터 목록에 레지스터가 오름차순으로 나열되도록 합니다. 레지스터가 오름차순으로 나열되지 않으면 경고가 나타납니다.

예제

```
Context RLIST   {r0-r6,r8,r10-r12,r15}
```

7.2.7 CN

CN 지시어는 보조 프로세서 레지스터의 이름을 정의합니다.

구문

name CN *expr*

인수 설명:

name 보조 프로세서 레지스터에 대해 정의할 이름입니다. *name*은 3-23페이지의 *미리 정의된 레지스터 및 보조 프로세서 이름*에 나와 있는 미리 정의된 이름과 같을 수 없습니다.

expr 0에서 15 사이의 보조 프로세서 레지스터 번호로 평가됩니다.

사용법

CN을 사용하여 각 레지스터의 용도를 기억하기 쉽도록 적절한 이름을 레지스터에 지정합니다.

참고

한 레지스터에 여러 이름을 사용하면 안 됩니다.

c0 ~ c15 이름이 미리 정의되어 있습니다.

예제

```
power    CN  6        ; defines power as a symbol for
                      ; coprocessor register 6
```

7.2.8 CP

CP 지시어는 지정한 보조 프로세서의 이름을 정의합니다. 보조 프로세서 번호는 0 ~ 15 범위에 있어야 합니다.

구문

name CP *expr*

인수 설명:

name 보조 프로세서에 지정할 이름입니다. *name*은 3-23페이지의 *미리 정의된 레지스터 및 보조 프로세서 이름*에 나와 있는 미리 정의된 이름과 같을 수 없습니다.

expr 0에서 15 사이의 보조 프로세서 번호로 평가됩니다.

사용법

CP를 사용하여 각 보조 프로세서의 용도를 기억하기 쉽도록 보조 프로세서에 적절한 이름을 지정합니다.

참고

한 보조 프로세서에 여러 이름을 사용하면 안 됩니다.

보조 프로세서 0 ~ 15에 대해 p0 ~ p15 이름이 미리 정의되어 있습니다.

예제

```
dmu    CP  6       ; defines dmu as a symbol for
                   ; coprocessor 6
```

7.2.9 QN, DN 및 SN

QN 지시어는 지정한 128비트 확장 레지스터의 이름을 정의합니다.

DN 지시어는 지정한 64비트 확장 레지스터의 이름을 정의합니다.

SN 지시어는 지정한 단정밀도 VFP 레지스터의 이름을 정의합니다.

구문

name directive expr{.*type*}{[*x*]}

인수 설명:

지시어 QN, DN 또는 SN입니다.

name 확장 레지스터에 지정할 이름입니다. *name*은 3-23페이지의 *미리 정의된 레지스터 및 보조 프로세서 이름*에 나와 있는 미리 정의된 이름과 같을 수 없습니다.

expr 다음과 같을 수 있습니다.

- 배정밀도 VFP 레지스터나 NEON 128비트 레지스터이면 0 ~ 15 범위의 숫자로 평가되고, 그렇지 않으면 0 ~ 31로 평가되는 식
- 미리 정의된 레지스터 이름이나 이전 지시어에서 이미 정의된 레지스터 이름

type 5-16페이지의 *NEON 및 VFP 데이터 형식*에서 설명하는 데이터 유형 중 하나입니다.

[*x*] NEON 코드에만 사용할 수 있습니다. [*x*]는 레지스터로의 스칼라 인덱스입니다.

type 및 [*x*]는 *확장 표시*입니다. 자세한 내용은 5-20페이지의 *확장 표시*를 참조하고, 사용법 예제는 7-15페이지의 *확장 표시 예제*를 참조하십시오.

사용법

QN, DN 또는 SN을 사용하여 각 확장 레지스터의 용도를 기억하기 쉽도록 확장 레지스터에 적절한 이름을 지정합니다.

참고

한 레지스터에 여러 이름을 사용하면 안 됩니다.

DN 또는 SN 지시어에서는 벡터 길이를 지정할 수 없습니다 (5-115페이지의 *VFP 지시어 및 벡터 표시* 참조).

예제

```
energy  DN  6   ; defines energy as a symbol for
                ; VFP double-precision register 6
mass    SN  16  ; defines mass as a symbol for
                ; VFP single-precision register 16
```

확장 표시 예제

```
varA    DN      d1.U16
varB    DN      d2.U16
varC    DN      d3.U16
        VADD    varA,varB,varC       ; VADD.U16 d1,d2,d3
index   DN      d4.U16[0]
result  QN      q5.I32
        VMULL   result,varA,index    ; VMULL.U16 q5,d1,d3[2]
```

7.3 데이터 정의 지시어

이 단원에서는 메모리를 할당하고, 데이터 구조체를 정의하며, 초기 메모리 내용을 설정하는 다음 지시어에 대해 설명합니다.

- 7-18페이지의 *LTORG*

  리터럴 풀의 원점을 설정합니다.

- 7-19페이지의 *MAP*

  저장 맵의 원점을 설정합니다.

- 7-20페이지의 *FIELD*

  저장 맵 내에서 필드를 정의합니다.

- 7-21페이지의 *SPACE 또는 FILL*

  0으로 채워진 메모리 블록을 할당합니다.

- 7-22페이지의 *DCB*

  메모리의 바이트를 할당하고 초기 내용을 지정합니다.

- 7-23페이지의 *DCD 및 DCDU*

  메모리의 워드를 할당하고 초기 내용을 지정합니다.

- 7-24페이지의 *DCDO*

  메모리의 워드를 할당하고 초기 내용을 정적 기준 레지스터의 오프셋으로 지정합니다.

- 7-25페이지의 *DCFD 및 DCFDU*

  메모리의 더블워드를 할당하고 초기 내용을 배정밀도 부동 소수점 숫자로 지정합니다.

- 7-26페이지의 *DCFS 및 DCFSU*

  메모리의 워드를 할당하고 초기 내용을 단정밀도 부동 소수점 숫자로 지정합니다.

- 7-27페이지의 *DCI*

  메모리의 워드를 할당하고 초기 내용을 지정합니다. 위치를 데이터가 아닌 코드로 표시합니다.

- 7-29페이지의 *DCQ 및 DCQU*

  메모리의 더블워드를 할당하고 초기 내용을 배정밀도 64비트 정수로 지정합니다.

- 7-30페이지의 *DCW 및 DCWU*

  메모리의 하프워드를 할당하고 초기 내용을 지정합니다.

- 7-31페이지의 *COMMON*

  기호에 메모리 블록을 할당하고 정렬을 지정합니다.

- 7-31페이지의 *DATA*

  코드 섹션 내의 데이터를 표시합니다. 역방향 호환성을 위해서만 사용됩니다.

7.3.1 LTORG

LTORG 지시어는 현재 리터럴 풀을 즉시 어셈블하도록 어셈블러에 지시합니다.

구문

LTORG

사용법

어셈블러는 코드 섹션이 끝날 때마다 현재 리터럴 풀을 어셈블합니다. 코드 섹션의 끝은 그 다음 섹션의 시작 부분이나 어셈블리 끝에 있는 AREA 지시어로 확인됩니다.

이러한 기본 리터럴 풀은 경우에 따라 일부 LDR, VLDR 및 WLDR 의사 명령어의 범위를 벗어날 수 있습니다. LTORG를 사용하면 리터럴 풀이 범위 내에서 어셈블되었는지 확인할 수 있습니다. 의사 명령어에 대한 자세한 내용은 다음을 참조하십시오.

- 4-159페이지의 *LDR 의사 명령어*
- 5-87페이지의 *VLDR 의사 명령어*
- 6-6페이지의 *Wireless MMX 로드 및 저장 명령어*

큰 프로그램에는 여러 개의 리터럴 풀이 필요할 수 있습니다. 프로세서에서 상수를 명령어로 실행하려고 하지 않도록 하려면 무조건 분기 또는 서브루틴 반환 명령어 다음에 LTORG 지시어를 배치합니다.

어셈블러는 리터럴 풀의 데이터를 워드로 정렬합니다.

예제

```
        AREA    Example, CODE, READONLY
start   BL      func1
func1                           ; function body
        ; code
        LDR     r1,=0x55555555  ; => LDR R1, [pc, #offset to Literal Pool 1]
        ; code
        MOV     pc,lr           ; end function
        LTORG                   ; Literal Pool 1 contains literal &55555555.
data    SPACE   4200            ; Clears 4200 bytes of memory,
                                ; starting at current location.
        END                     ; Default literal pool is empty.
```

7.3.2 MAP

MAP 지시어는 저장소 맵의 원점을 지정된 주소로 설정합니다. 저장소 맵 위치 카운터인 {VAR}도 이 주소로 설정됩니다. ^ 기호는 MAP의 동의어입니다.

구문

MAP *expr*{,*base-register*}

인수 설명:

expr 다음과 같은 숫자 또는 프로그램 상대 식입니다.

- *base-register*를 지정하지 않으면 *expr*은 저장 맵이 시작하는 주소로 평가됩니다. 저장 맵 위치 카운터도 이 주소로 설정됩니다.
- *expr*이 프로그램 기준 식이면 맵에서 이 식을 사용하기 전에 레이블을 정의해야 합니다. 어셈블러의 첫 번째 패스 중에 맵에는 레이블의 정의가 필요합니다.

base-register

레지스터를 지정합니다. *base-register*를 지정하지 않으면 저장 맵을 시작하는 주소는 *expr*과 *base-register* 런타임 값의 합입니다.

사용법

FIELD 지시어와 함께 MAP 지시어를 사용하여 저장 맵을 설명합니다.

*base-register*를 지정하여 레지스터 기준 레이블을 정의합니다. 기준 레지스터는 다음 MAP 지시어가 나올 때까지 이후 FIELD 지시어에 의해 정의된 모든 레이블에서 암시적 레지스터가 됩니다. 레지스터 기준 레이블은 로드 및 저장 명령어에서 사용할 수 있습니다. 예제를 보려면 7-20페이지의 *FIELD*를 참조하십시오.

MAP 지시어는 다중 저장소 맵을 정의하기 위해 횟수에 제한 없이 원하는 대로 사용할 수 있습니다.

{VAR} 카운터는 첫 번째 MAP 지시어가 사용되기 전에 0으로 설정됩니다.

예제

```
MAP     0,r9
MAP     0xff,r9
```

7.3.3 FIELD

FIELD 지시어는 MAP 지시어를 사용하여 정의한 저장소 맵 내의 공간을 설명합니다. # 기호는 FIELD의 동의어입니다.

구문

{*label*} FIELD *expr*

인수 설명:

label 선택적 레이블입니다. 지정된 경우 저장 위치 카운터인 {VAR} 값이 *label*에 할당되고 저장 위치 카운터는 다시 *expr* 값만큼 증가합니다.

expr 저장 카운터를 증가하는 데 사용할 바이트 수로 평가되는 식입니다.

사용법

저장소 맵이 *base-register*를 지정하는 MAP 지시어에 의해 설정될 경우 기준 레지스터는 다음 MAP 지시어가 나올 때까지 이후 FIELD 지시어에 의해 정의된 모든 레이블에서 암시적 레지스터가 됩니다. 이러한 레지스터 상대 레이블은 로드 및 저장 명령어에 인용할 수 있습니다 (7-19페이지의 *MAP* 참조).

예제

다음 예제에서는 MAP 및 FIELD 지시어를 사용하여 레지스터 상대 레이블을 정의하는 방법을 보여 줍니다.

```
    MAP     0,r9        ; set {VAR} to the address stored in r9
    FIELD   4           ; increment {VAR} by 4 bytes
Lab FIELD   4           ; set Lab to the address [r9 + 4]
                        ; and then increment {VAR} by 4 bytes
    LDR     r0,Lab      ; equivalent to LDR r0,[r9,#4]
```

7.3.4 SPACE 또는 FILL

SPACE 지시어는 0으로 채워진 메모리 블록을 예약합니다. % 기호는 SPACE의 동의어입니다.

FILL 지시어는 지정된 값으로 채워진 메모리 블록을 예약합니다.

구문

{*label*} SPACE *expr*

{*label*} FILL *expr*{,*value*{,*valuesize*}}

인수 설명:

label 선택적 레이블입니다.

expr 채울 바이트 수 또는 0으로 평가됩니다 (3-35페이지의 *숫자 식* 참조).

value 예약된 바이트를 채울 값으로 평가됩니다. *value*는 옵션이며 생략하는 경우 값은 0입니다. MOINIT 영역에서는 *value*가 0이어야 합니다.

valuesize *value*의 크기 (바이트) 입니다. 1, 2 또는 4일 수 있습니다. *valuesize*는 옵션이며 생략하는 경우 값은 1입니다.

사용법

ALIGN 지시어를 사용하여 SPACE 또는 FILL 지시어 뒤에 오는 모든 코드를 정렬합니다. 자세한 내용은 7-67페이지의 *ALIGN*을 참조하십시오.

추가 참고:

- 7-22페이지의 *DCB*
- 7-23페이지의 *DCD 및 DCDU*
- 7-24페이지의 *DCDO*
- 7-30페이지의 *DCW 및 DCWU*

예제

```
        AREA    MyData, DATA, READWRITE
data1   SPACE   255       ; defines 255 bytes of zeroed store
data2   FILL    50,0xAB,1 ; defines 50 bytes containing 0xAB
```

7.3.5 DCB

DCB 지시어는 메모리의 바이트를 하나 이상 할당하고 메모리의 초기 런타임 내용을 정의합니다. = 기호는 DCB의 동의어입니다.

구문

{*label*} DCB *expr*{,*expr*}...

인수 설명:

expr 다음 중 하나입니다.

- –128에서 255 사이의 정수로 평가되는 숫자 식 (3-35페이지의 *숫자 식* 참조) 입니다.
- 인용된 문자열입니다. 문자열의 문자는 저장소의 연속된 바이트로 로드됩니다.

사용법

DCB 다음에 명령어가 나오면 ALIGN 지시어를 사용하여 이 명령어가 정렬되도록 합니다. 자세한 내용은 7-67페이지의 *ALIGN*을 참조하십시오.

추가 참고:

- 7-23페이지의 *DCD 및 DCDU*
- 7-29페이지의 *DCQ 및 DCQU*
- 7-30페이지의 *DCW 및 DCWU*
- 7-21페이지의 *SPACE 또는 FILL*

예제

C 문자열과 달리 ARM 어셈블러 문자열은 Null로 끝나지 않습니다. 다음과 같이 DCB를 사용하여 Null로 끝나는 C 문자열을 만들 수 있습니다.

```
C_string   DCB  "C_string",0
```

7.3.6 DCD 및 DCDU

DCD 지시어는 4바이트 단위로 정렬된 메모리의 워드를 하나 이상 할당하고 메모리의 초기 런타임 내용을 정의합니다.

& 기호는 DCD의 동의어입니다.

메모리를 임의로 정렬한다는 점을 제외하면 DCDU는 이 지시어와 같습니다.

구문

{*label*} DCD{U} *expr*{,*expr*}

인수 설명:

expr 다음 중 하나입니다.
- 숫자 식 (3-35페이지의 *숫자 식* 참조)
- 프로그램 기준 식

사용법

DCD는 필요한 경우 4바이트 단위로 정렬하기 위해 첫 번째로 정의된 워드 앞에 최대 3바이트의 패딩을 삽입합니다.

정렬이 필요하지 않으면 DCDU를 사용합니다.

추가 참고:
- 7-22페이지의 *DCB*
- 7-27페이지의 *DCI*
- 7-30페이지의 *DCW 및 DCWU*
- 7-29페이지의 *DCQ 및 DCQU*
- 7-21페이지의 *SPACE 또는 FILL*

예제

```
data1   DCD     1,5,20      ; Defines 3 words containing
                            ; decimal values 1, 5, and 20
data2   DCD     mem06 + 4   ; Defines 1 word containing 4 +
                            ; the address of the label mem06
        AREA    MyData, DATA, READWRITE
        DCB     255         ; Now misaligned ...
data3   DCDU    1,5,20      ; Defines 3 words containing
                            ; 1, 5 and 20, not word aligned
```

7.3.7 DCDO

DCDO 지시어는 4바이트 단위로 정렬된 메모리의 워드를 하나 이상 할당하고 메모리의 초기 런타임 내용을 *정적 기준 레지스터*인 sb (r9) 의 오프셋으로 정의합니다.

구문

{*label*} DCDO *expr*{,*expr*}...

인수 설명:

expr 레지스터 상대 식이거나 레이블입니다. 기준 레지스터는 sb여야 합니다.

사용법

DCDO를 사용하여 재배치 가능한 정적 기준 레지스터 기준 주소를 위한 메모리에 공간을 할당합니다.

예제

```
        IMPORT  externsym
        DCDO    externsym   ; 32-bit word relocated by offset of
                            ; externsym from base of SB section.
```

7.3.8 DCFD 및 DCFDU

DCFD 지시어는 워드로 정렬된 배정밀도 부동 소수점 숫자를 위한 메모리를 할당하고 메모리의 초기 런타임 내용을 정의합니다. 배정밀도 숫자는 2워드를 차지하며, 산술 연산에 사용할 수 있도록 워드로 정렬되어야 합니다.

메모리를 임의로 정렬한다는 점을 제외하면 DCFDU는 이 지시어와 같습니다.

구문

{*label*} DCFD{U} *fpliteral*{,*fpliteral*}...

인수 설명:

fpliteral 배정밀도 부동 소수점 리터럴 (3-37페이지의 *부동 소수점 리터럴* 참조) 입니다.

사용법

어셈블러는 필요한 경우 4바이트 단위로 정렬하기 위해 첫 번째로 정의된 숫자 앞에 최대 3바이트의 패딩을 삽입합니다.

정렬이 필요하지 않으면 DCFDU를 사용합니다.

*fpliteral*을 내부 형식으로 변환할 때 사용되는 워드 순서는 선택한 부동 소수점 아키텍처에 의해 제어됩니다. --fpu none 옵션을 선택하면 DCFD 또는 DCFDU를 사용할 수 없습니다.

배정밀도 숫자의 범위는 다음과 같습니다.

- 최대 1.79769313486231571e+308
- 최소 2.22507385850720138e–308

7-26페이지의 *DCFS 및 DCFSU*도 참조하십시오.

예제

```
DCFD    1E308,-4E-100
DCFDU   10000,-.1,3.1E26
```

7.3.9 DCFS 및 DCFSU

DCFS 지시어는 워드로 정렬된 단정밀도 부동 소수점 숫자를 위한 메모리를 할당하고 메모리의 초기 런타임 내용을 정의합니다. 단정밀도 숫자는 1워드를 차지하며, 산술 연산에 사용할 수 있도록 워드로 정렬되어야 합니다.

메모리를 임의로 정렬한다는 점을 제외하면 DCFSU는 이 지시어와 같습니다.

구문

{*label*} DCFS{U} *fpliteral*{,*fpliteral*}...

인수 설명:

fpliteral 단정밀도 부동 소수점 리터럴 (3-37페이지의 *부동 소수점 리터럴* 참조) 입니다.

사용법

DCFS는 필요한 경우 4바이트 단위로 정렬하기 위해 첫 번째로 정의된 숫자 앞에 최대 3바이트의 패딩을 삽입합니다.

정렬이 필요하지 않으면 DCFSU를 사용합니다.

단정밀도 값의 범위는 다음과 같습니다.

- 최대 3.40282347e+38
- 최소 1.17549435e–38

7-25페이지의 *DCFD 및 DCFDU*도 참조하십시오.

예제

```
DCFS    1E3,-4E-9
DCFSU   1.0,-.1,3.1E6
```

7.3.10 DCI

ARM 코드에서 DCI 지시어는 4바이트 단위로 정렬된 메모리의 워드를 하나 이상 할당하고 메모리의 초기 런타임 내용을 정의합니다.

Thumb 코드에서 DCI 지시어는 2바이트 단위로 정렬된 메모리의 하프워드를 하나 이상 할당하고 메모리의 초기 런타임 내용을 정의합니다.

구문

{*label*} DCI{.W} *expr*{,*expr*}

인수 설명:

expr 숫자 식 (3-35페이지의 *숫자 식* 참조) 입니다.

.W 있을 경우 Thumb 코드에 4바이트를 삽입해야 함을 나타냅니다.

사용법

DCI 지시어는 위치가 데이터 대신 코드로 표시된다는 점을 제외하고 DCD 또는 DCW 지시어와 매우 유사합니다. DCI는 사용 중인 어셈블러 버전에서 지원하지 않는 새 명령어의 매크로를 작성할 때 사용합니다.

ARM 코드에서 DCI는 필요한 경우 4바이트 단위로 정렬하기 위해 첫 번째로 정의된 워드 앞에 최대 3바이트의 패딩을 삽입합니다. Thumb 코드에서 DCI는 필요한 경우 2바이트 단위로 정렬하기 위해 처음 1바이트의 패딩을 삽입합니다.

DCI를 사용하여 명령어 스트림에 비트 패턴을 삽입할 수 있습니다. 예를 들어 다음과 같이 사용합니다.

```
DCI 0x46c0
```

Thumb 연산 MOV r8,r8을 삽입합니다.

7-23페이지의 *DCD 및 DCDU* 및 7-30페이지의 *DCW 및 DCWU*도 참조하십시오.

예제 매크로

```
MACRO           ; this macro translates newinstr Rd,Rm
                ; to the appropriate machine code
newinst     $Rd,$Rm
DCI         0xe16f0f10 :OR: ($Rd:SHL:12) :OR: $Rm
MEND
```

Thumb-2 예제

```
DCI.W   0xf3af8000   ; inserts 32-bit NOP, 2-byte aligned.
```

7.3.11 DCQ 및 DCQU

DCQ 지시어는 4바이트 단위로 정렬된 메모리의 8바이트 블록을 하나 이상 할당하고 메모리의 초기 런타임 내용을 정의합니다.

메모리를 임의로 정렬한다는 점을 제외하면 DCQU는 이 지시어와 같습니다.

구문

{*label*} DCQ{U} {-}*literal*{,{-}*literal*}...

인수 설명:

literal 64비트 숫자 리터럴 (3-36페이지의 *숫자 리터럴* 참조) 입니다.

허용되는 숫자 범위는 0에서 2^{64}–1 사이입니다.

숫자 리터럴에 일반적으로 허용되는 문자 외에도 *literal*의 접두사로 빼기 기호를 사용할 수도 있습니다. 이 경우 허용되는 숫자 범위는 -2^{63}에서 –1 사이입니다.

-*n*을 지정한 결과와 2^{64}–*n*을 지정한 결과는 같습니다.

사용법

DCQ는 필요한 경우 4바이트 단위로 정렬하기 위해 첫 번째로 정의된 8바이트 블록 앞에 최대 3바이트의 패딩을 삽입합니다.

정렬이 필요하지 않으면 DCQU를 사용합니다.

추가 참고:

- 7-22페이지의 *DCB*
- 7-23페이지의 *DCD 및 DCDU*
- 7-30페이지의 *DCW 및 DCWU*
- 7-21페이지의 *SPACE 또는 FILL*

예제

```
        AREA    MiscData, DATA, READWRITE
data    DCQ     -225,2_101      ; 2_101 means binary 101.
        DCQU    number+4        ; number must already be defined.
```

7.3.12 DCW 및 DCWU

DCW 지시어는 2바이트 단위로 정렬된 메모리의 하프워드를 하나 이상 할당하고 메모리의 초기 런타임 내용을 정의합니다.

메모리를 임의로 정렬한다는 점을 제외하면 DCWU는 이 지시어와 같습니다.

구문

{*label*} DCW{U} *expr*{,*expr*}...

인수 설명:

expr –32768에서 6553 사이의 정수로 평가되는 숫자 식 (3-35페이지의 *숫자 식* 참조) 입니다.

사용법

DCW는 필요한 경우 2바이트 단위로 정렬하기 위해 첫 번째로 정의된 하프워드 앞에 1바이트의 패딩을 삽입합니다.

정렬이 필요하지 않으면 DCWU를 사용합니다.

추가 참고:

- 7-22페이지의 *DCB*
- 7-23페이지의 *DCD 및 DCDU*
- 7-29페이지의 *DCQ 및 DCQU*
- 7-21페이지의 *SPACE 또는 FILL*

예제

```
data    DCW     -225,2*number   ; number must already be defined
        DCWU    number+4
```

7.3.13 COMMON

COMMON 지시어는 지정된 기호에 정의된 크기의 메모리 블록을 할당합니다. 이때 메모리가 정렬되는 방식을 지정합니다. 정렬을 생략할 경우 기본 정렬은 4이고, 크기를 생략할 경우 기본 크기는 0입니다.

다른 메모리에 액세스할 때와 마찬가지로 이 메모리에 액세스할 수 있지만 객체 파일에 공간이 할당되지는 않습니다.

구문

COMMON *symbol*{,*size*{,*alignment*}}

인수 설명:

symbol 기호 이름입니다. 기호 이름은 대소문자를 구분합니다.

size 예약할 바이트 수입니다.

alignment 정렬입니다.

사용법

링커에서는 링크 단계 중에 필요한 공간을 0으로 초기화된 메모리로 할당합니다.

예제

```
COMMON    xyz,255,4   ; defines 255 bytes of ZI store, word-aligned
```

7.3.14 DATA

DATA 지시어는 어셈블러에서 무시되므로 더 이상 필요하지 않습니다.

7.4 어셈블리 제어 지시어

이 단원에서는 조건부 어셈블리, 루핑, 포함 및 매크로를 제어하는 다음 지시어에 대해 설명합니다.

- 7-33페이지의 *MACRO 및 MEND*
- 7-36페이지의 *MEXIT*
- 7-37페이지의 *IF, ELSE, ENDIF 및 ELIF*
- 7-40페이지의 *WHILE 및 WEND*

7.4.1 중첩 지시어

다음 구조는 총 깊이 256까지 중첩될 수 있습니다.

- `MACRO` 정의
- `WHILE...WEND` 루프
- `IF...ELSE...ENDIF` 조건부 구조
- `INCLUDE` 파일 포함

제한은 중첩 방식에 관계없이 함께 있는 모든 구조에 적용됩니다. 각 유형의 구조는 256개로 제한되지 *않*습니다.

7.4.2 MACRO 및 MEND

MACRO 지시어는 매크로 정의의 시작을 표시합니다. 매크로 확장은 MEND 지시어에서 끝납니다. 자세한 내용은 2-49페이지의 *매크로 사용*을 참조하십시오.

구문

두 개의 지시어가 매크로를 정의하는 데 사용됩니다. 구문은 다음과 같습니다.

```
          MACRO
{$label}  macroname{$cond} {$parameter{,$parameter}...}
          ; code
          MEND
```

인수 설명:

$label 매크로가 호출될 때 지정한 기호로 대체되는 매개변수입니다. 기호는 대개 레이블입니다.

macroname 매크로 이름으로, 명령어나 지시어로 시작하면 안 됩니다.

$cond 조건 코드를 포함하도록 설계된 특수 매개변수입니다. 유효한 조건 코드가 아닌 값이 허용됩니다.

$parameter 매크로가 호출될 때 대체되는 매개변수입니다. 매개변수의 기본값은 다음 형식을 사용하여 설정할 수 있습니다.

$parameter="default value"

기본값의 내부나 한 쪽 끝에 공백이 있을 경우 큰따옴표를 사용해야 합니다.

사용법

매크로 내에서 WHILE...WEND 루프나 IF...ENDIF 조건을 시작한 경우 MEND 지시어에 도달하기 전에 이들을 닫아야 합니다. 루프 내에서 매크로를 종료하는 등의 매크로 조기 종료를 활성화하려면 7-36페이지의 *MEXIT*를 참조하십시오.

매크로 본문 내에서 *$label*, *$parameter* 또는 *$cond*와 같은 매개변수는 다른 변수와 같은 방식으로 사용됩니다 (3-30페이지의 *변수의 어셈블리 타임 대체* 참조). 이러한 매개변수에는 매크로가 호출될 때마다 새 값이 지정됩니다. 매개변수는 다른 기호와 구분할 수 있도록 $ 기호로 시작해야 하며 개수에 관계없이 원하는 대로 사용할 수 있습니다.

*$label*은 선택적 매개변수로, 매크로가 내부 레이블을 정의하는 경우에 유용합니다. 이 매개변수는 매크로의 매개변수로 처리되며 매크로 확장 내의 첫 번째 명령어를 나타내지 않을 수도 있습니다. 매크로는 모든 레이블의 위치를 정의합니다.

|를 인수로 사용하여 매개변수의 기본값을 사용합니다. 인수를 생략할 경우 빈 문자열이 사용됩니다.

여러 개의 내부 레이블을 사용하는 매크로에서는 서로 다른 접미사를 사용하여 각 내부 레이블을 기본 레이블로 정의하는 것이 좋습니다.

확장에 공백이 필요하지 않을 경우 매개변수와 후행 텍스트 사이나 두 매개변수 사이에 마침표를 사용합니다. 선행 텍스트와 매개변수 사이에는 마침표를 사용하면 안 됩니다.

$*cond* 매개변수를 조건 코드에 사용할 수 있습니다. 반대 조건 코드를 찾으려면 단항 연산자 :REVERSE_CC:를 사용하고 조건 코드의 4비트 인코딩을 찾으려면 :CC_ENCODING:을 사용합니다.

매크로는 지역 변수 (7-7페이지의 *LCLA, LCLL 및 LCLS* 참조) 의 범위를 정의합니다.

매크로는 중첩될 수 있습니다 (7-32페이지의 *중첩 지시어* 참조).

예제

```
 ; macro definition
                MACRO                   ; start macro definition
$label          xmac    $p1,$p2
                ; code
$label.loop1    ; code
                ; code
                BGE     $label.loop1
$label.loop2    ; code
                BL      $p1
                BGT     $label.loop2
                ; code
                ADR     $p2
                ; code
                MEND                    ; end macro definition
 ; macro invocation
abc             xmac    subr1,de        ; invoke macro
                ; code                  ; this is what is
abcloop1        ; code                  ; is produced when
                ; code                  ; the xmac macro is
                BGE     abcloop1        ; expanded
abcloop2        ; code
```

```
        BL      subr1
        BGT     abcloop2
        ; code
        ADR     de
        ; code
```

매크로를 사용하여 어셈블리 타임 진단 만들기

```
        MACRO                       ; Macro definition
        diagnose  $param1="default"  ; This macro produces
        INFO      0,"$param1"       ; assembly-time diagnostics
        MEND                        ; (on second assembly pass)
 ; macro expansion
        diagnose            ; Prints blank line at assembly-time
        diagnose "hello"    ; Prints "hello" at assembly-time
        diagnose |          ; Prints "default" at assembly-time
```

조건부 매크로 예제

```
        AREA    codx, CODE, READONLY

; macro definition

        MACRO
        Return$cond
        [ {ARCHITECTURE} <> "4"
          BX$cond lr
          |
          MOV$cond pc,lr
        ]
        MEND

; macro invocation

fun     PROC
        CMP      r0,#0
        MOVEQ    r0,#1
        ReturnEQ
        MOV      r0,#0
        Return
        ENDP

        END
```

7.4.3 MEXIT

MEXIT 지시어는 매크로가 끝나기 전에 매크로 정의를 종료하는 데 사용됩니다.

사용법

MEXIT는 매크로를 중간에 중단해야 할 때 사용합니다. 매크로 본문 내의 닫히지 않은 WHILE...WEND 루프나 IF...ENDIF 조건은 매크로가 종료되기 전에 어셈블러에 의해 닫힙니다.

7-33페이지의 *MACRO 및 MEND*도 참조하십시오.

예제

```
        MACRO
$abc    example abc     $param1,$param2
        ; code
        WHILE condition1
            ; code
            IF condition2
                ; code
                MEXIT
            ELSE
                ; code
            ENDIF
        WEND
        ; code
        MEND
```

7.4.4 IF, ELSE, ENDIF 및 ELIF

IF 지시어는 명령어 및 지시어 시퀀스를 어셈블할지 여부를 결정하는 데 사용되는 조건을 추가합니다. [기호는 IF의 동의어입니다.

ELSE 지시어는 선행 조건이 실패할 경우 어셈블할 명령어 또는 지시어 시퀀스의 시작을 표시합니다. | 기호는 ELSE의 동의어입니다.

ENDIF 지시어는 조건부로 어셈블할 명령어 또는 지시어 시퀀스의 끝을 표시합니다.] 기호는 ENDIF의 동의어입니다.

ELIF 지시어는 ELSE IF와 같지만 조건을 중첩시키거나 반복하지 않아도 되는 구조체를 만듭니다. 자세한 내용은 7-38페이지의 *ELIF 사용*을 참조하십시오.

구문

```
IF logical-expression
  …;code
{ELSE
  …;code}
ENDIF
```

인수 설명:

logical-expression

{TRUE} 또는 {FALSE}로 평가되는 식입니다.

자세한 내용은 3-45페이지의 *관계 연산자*를 참조하십시오.

사용법

지정한 조건에서만 어셈블하거나 실행할 명령어 또는 지시어 시퀀스에 IF를 ENDIF와 함께 사용하거나, 경우에 따라 ELSE와 함께 사용합니다.

IF...ENDIF 조건은 중첩될 수 있습니다 (7-32페이지의 *중첩 지시어* 참조).

ELIF 사용

다음과 같이 ELIF를 사용하지 않고도 중첩된 조건부 명령어 세트를 만들 수 있습니다.

```
IF logical-expression
    instructions
ELSE
    IF logical-expression2
        instructions
    ELSE
        IF logical-expression3
            instructions
        ENDIF
    ENDIF
ENDIF
```

이러한 중첩된 구조는 최대 256수준 깊이까지 중첩될 수 있습니다.

ELIF를 사용하면 다음과 같이 위와 동일한 구조를 더 간단하게 작성할 수 있습니다.

```
IF logical-expression
    instructions
ELIF logical-expression2
    instructions
ELIF logical-expression3
    instructions
ENDIF
```

이 구조는 현재 중첩 깊이에 IF...ENDIF 쌍을 하나만 추가합니다.

예제

예제 7-3에서는 NEWVERSION이 정의된 경우 첫 번째 명령어 세트를 어셈블하고, 그렇지 않으면 대체 세트를 어셈블합니다.

예제 7-3 정의 중인 변수에 따른 조건부 어셈블리

```
        IF :DEF:NEWVERSION
            ; first set of instructions or directives
        ELSE
            ; alternative set of instructions or directives
        ENDIF
```

armasm을 다음과 같이 호출하면 NEWVERSION이 정의되어 명령어 및 지시어의 첫 번째 세트가 어셈블됩니다.

```
armasm --predefine "NEWVERSION SETL {TRUE}" test.s
```

armasm을 다음과 같이 호출하면 NEWVERSION이 정의되지 않은 채로 있어 명령어 및 지시어의 두 번째 세트가 어셈블됩니다.

```
armasm test.s
```

예제 7-4에서는 NEWVERSION의 값이 {TRUE}이면 첫 번째 명령어 세트를 어셈블하고, 그렇지 않으면 대체 세트를 어셈블합니다.

예제 7-4 변수 값에 따른 조건부 어셈블리

```
        IF NEWVERSION = {TRUE}
            ; first set of instructions or directives
        ELSE
            ; alternative set of instructions or directives
        ENDIF
```

armasm을 다음과 같이 호출하면 명령어 및 지시어의 첫 번째 세트가 어셈블됩니다.

```
armasm --predefine "NEWVERSION SETL {TRUE}" test.s
```

armasm을 다음과 같이 호출하면 명령어 및 지시어의 두 번째 세트가 어셈블됩니다.

```
armasm --predefine "NEWVERSION SETL {FALSE}" test.s
```

7.4.5 WHILE 및 WEND

WHILE 지시어는 반복적으로 어셈블할 명령어 또는 지시어 시퀀스를 시작합니다. 이러한 시퀀스는 WEND 지시어로 끝납니다.

구문

WHILE *logical-expression*

code

WEND

인수 설명:

logical-expression

{TRUE} 또는 {FALSE}로 평가될 수 있는 식입니다 (3-38페이지의 *논리식* 참조).

사용법

WEND 지시어와 함께 WHILE 지시어를 사용하여 명령어 시퀀스를 여러 번 어셈블합니다. 반복 횟수는 0일 수 있습니다.

WHILE...WEND 루프 내에서 IF...ENDIF 조건을 사용할 수 있습니다.

WHILE...WEND 루프는 중첩될 수 있습니다 (7-32페이지의 *중첩 지시어* 참조).

예제

```
count   SETA    1                   ; you are not restricted to
        WHILE   count <= 4          ; such simple conditions
count   SETA    count+1             ; In this case,
            ; code                  ; this code will be
            ; code                  ; repeated four times
        WEND
```

7.5 프레임 지시어

이 단원에서는 다음 지시어에 대해 설명합니다.

- 7-43페이지의 *FRAME ADDRESS*
- 7-45페이지의 *FRAME POP*
- 7-46페이지의 *FRAME PUSH*
- 7-48페이지의 *FRAME REGISTER*
- 7-49페이지의 *FRAME RESTORE*
- 7-50페이지의 *FRAME RETURN ADDRESS*
- 7-51페이지의 *FRAME SAVE*
- 7-52페이지의 *FRAME STATE REMEMBER*
- 7-53페이지의 *FRAME STATE RESTORE*
- 7-54페이지의 *FRAME UNWIND ON*
- 7-54페이지의 *FRAME UNWIND OFF*
- 7-55페이지의 *FUNCTION 또는 PROC*
- 7-56페이지의 *ENDFUNC 또는 ENDP*

다음은 이러한 지시어의 올바른 사용 방법입니다.

- armlink --callgraph 옵션을 설정하여 어셈블러 함수의 스택 사용량을 계산합니다.

  다음 규칙이 스택 사용량을 확인하는 데 사용됩니다.

  — 함수가 PROC 또는 ENDP로 표시되지 않을 경우 스택 사용량을 알 수 없습니다.

  — 함수가 FRAME PUSH 또는 FRAME POP 없이 PROC 또는 ENDP로만 표시될 경우 스택 사용량은 0으로 간주됩니다. 즉, FRAME PUSH 0 또는 FRAME POP 0을 수동으로 추가하지 않아도 됩니다.

  — 함수가 PROC 또는 ENDP 및 FRAME PUSH n 또는 FRAME POP n으로 표시될 경우 스택 사용량은 n바이트로 간주됩니다.

- 함수 생성 시 특히, 기존 코드를 수정할 때 오류가 발생하지 않도록 도와 줍니다.
- 어셈블러에서 함수를 생성할 때 오류가 발생하면 경고 메시지를 표시하도록 합니다.
- 디버그하는 동안 함수 호출을 백트레이싱할 수 있도록 합니다.

- 디버거에서 어셈블러 함수를 프로파일링할 수 있도록 합니다.

어셈블러 함수를 프로파일링해야 하지만 프레임 설명 지시어를 다른 목적으로 사용하지 않으려는 경우

- FUNCTION 및 ENDFUNC 또는 PROC 및 ENDP 지시어를 사용해야 합니다.
- 다른 FRAME 지시어를 생략할 수 있습니다.
- 프로파일링할 함수에 대해서만 FUNCTION 및 ENDFUNC 지시어를 사용해야 합니다.

DWARF에서 표준 프레임 주소는 중단된 함수의 호출 프레임이 있는 위치를 지정하는 스택상의 주소를 나타냅니다.

7.5.1 FRAME ADDRESS

FRAME ADDRESS 지시어는 이후 명령어의 표준 프레임 주소를 계산하는 방법을 설명합니다. 이 지시어는 FUNCTION 및 ENDFUNC 또는 PROC 및 ENDP 지시어가 있는 함수 내에서만 사용할 수 있습니다.

구문

FRAME ADDRESS *reg*[,*offset*]

인수 설명:

reg 표준 프레임 주소의 기반이 될 레지스터로, 함수에서 별도의 프레임 포인터를 사용하지 않을 경우 sp입니다.

offset *reg*를 기준으로 한 표준 프레임 주소의 오프셋으로, *offset*이 0일 경우 생략할 수 있습니다.

사용법

코드에서 표준 프레임 주소의 기반이 되는 레지스터를 변경하거나 레지스터를 기준으로 한 표준 프레임 주소의 오프셋을 변경할 경우 FRAME ADDRESS를 사용합니다. FRAME ADDRESS는 표준 프레임 주소의 계산을 변경하는 명령어 바로 다음에 사용해야 합니다.

참고

코드에서 단일 명령어를 사용하여 레지스터를 저장하고 스택 포인터를 변경하는 경우 FRAME ADDRESS와 FRAME SAVE를 둘 다 사용하는 대신 FRAME PUSH를 사용할 수 있습니다 (7-46페이지의 *FRAME PUSH* 참조).

코드에서 단일 명령어를 사용하여 레지스터를 로드하고 스택 포인터를 변경하는 경우 FRAME ADDRESS와 FRAME RESTORE를 둘 다 사용하는 대신 FRAME POP을 사용할 수 있습니다 (7-45페이지의 *FRAME POP* 참조).

예제

```
_fn     FUNCTION        ; CFA (Canonical Frame Address) is value
                        ; of sp on entry to function
        PUSH    {r4,fp,ip,lr,pc}
        FRAME PUSH {r4,fp,ip,lr,pc}
        SUB     sp,sp,#4            ; CFA offset now changed
        FRAME ADDRESS sp,24         ; - so we correct it
        ADD     fp,sp,#20
        FRAME ADDRESS fp,4          ; New base register
        ; code using fp to base call-frame on, instead of sp
```

7.5.2 FRAME POP

FRAME POP 지시어를 사용하여 호출 수신자가 레지스터를 다시 로드할 때 어셈블러에 이를 알립니다. 이 지시어는 FUNCTION 및 ENDFUNC 또는 PROC 및 ENDP 지시어가 있는 함수 내에서만 사용할 수 있습니다.

함수의 마지막 명령어 뒤에서는 이렇게 하지 않아도 됩니다.

구문

FRAME POP의 대체 구문에는 다음과 같은 세 가지가 있습니다.

```
FRAME POP {reglist}

FRAME POP {reglist},n

FRAME POP n
```

인수 설명:

reglist 함수 진입점에 있는 값으로 복원할 레지스터 목록입니다. 이 목록에는 최소한 하나 이상의 레지스터가 있어야 합니다.

n 스택 포인터가 이동하는 바이트 수입니다.

사용법

FRAME POP은 FRAME ADDRESS 지시어와 FRAME RESTORE 지시어를 모두 사용하는 것과 같습니다. 이 지시어는 단일 명령어가 레지스터를 로드하고 스택 포인터를 변경할 때 사용할 수 있습니다.

FRAME POP은 해당 지시어가 참조하는 명령어 바로 다음에 사용해야 합니다.

*n*이 지정되지 않거나 0일 경우 어셈블러는 {*reglist*}를 기준으로 표준 프레임 주소의 새 오프셋을 계산합니다. 어셈블러는 다음을 가정합니다.

- 팝된 각 ARM 레지스터가 스택에서 4바이트를 차지합니다.
- 팝된 각 VFP 단정밀도 레지스터가 스택에서 4바이트 외에도 각 목록을 위한 4바이트 워드를 추가로 차지합니다.
- 팝된 각 VFP 배정밀도 레지스터가 스택에서 8바이트 외에도 각 목록을 위한 4바이트 워드를 추가로 차지합니다.

7-43페이지의 *FRAME ADDRESS* 및 7-49페이지의 *FRAME RESTORE*를 참조하십시오.

7.5.3 FRAME PUSH

FRAME PUSH 지시어를 사용하여 주로 함수 진입점에서 호출 수신자가 레지스터를 저장할 때 어셈블러에 이를 알립니다. 이 지시어는 FUNCTION 및 ENDFUNC 또는 PROC 및 ENDP 지시어가 있는 함수 내에서만 사용할 수 있습니다.

구문

FRAME PUSH의 대체 구문에는 다음과 같은 두 가지가 있습니다.

```
FRAME PUSH {reglist}
FRAME PUSH {reglist},n
FRAME PUSH n
```

인수 설명:

reglist 표준 프레임 주소 아래에 연속적으로 저장된 레지스터 목록입니다. 이 목록에는 최소한 하나 이상의 레지스터가 있어야 합니다.

n 스택 포인터가 이동하는 바이트 수입니다.

사용법

FRAME PUSH는 FRAME ADDRESS 지시어와 FRAME SAVE 지시어를 모두 사용하는 것과 같습니다. 이 지시어는 단일 명령어가 레지스터를 저장하고 스택 포인터를 변경할 때 사용할 수 있습니다.

FRAME PUSH는 해당 지시어가 참조하는 명령어 바로 다음에 사용해야 합니다.

*n*이 지정되지 않거나 0일 경우 어셈블러는 {*reglist*}를 기준으로 표준 프레임 주소의 새 오프셋을 계산합니다. 어셈블러는 다음을 가정합니다.

- 푸시된 각 ARM 레지스터가 스택의 4바이트를 차지합니다.
- 푸시된 각 VFP 단정밀도 레지스터가 스택에서 4바이트 외에도 각 목록을 위한 4바이트 워드를 추가로 차지합니다.
- 팝된 각 VFP 배정밀도 레지스터가 스택에서 8바이트 외에도 각 목록을 위한 4바이트 워드를 추가로 차지합니다.

7-43페이지의 *FRAME ADDRESS* 및 7-51페이지의 *FRAME SAVE*를 참조하십시오.

예제

```
p   PROC ; Canonical frame address is sp + 0
    EXPORT  p
    PUSH    {r4-r6,lr}
         ; sp has moved relative to the canonical frame address,
         ; and registers r4, r5, r6 and lr are now on the stack
    FRAME PUSH {r4-r6,lr}
         ; Equivalent to:
         ; FRAME ADDRESS    sp,16       ; 16 bytes in {r4-r6,lr}
         ; FRAME SAVE    {r4-r6,lr},-16
```

7.5.4 FRAME REGISTER

FRAME REGISTER 지시어를 사용하여 레지스터에 들어 있는 함수 인수의 위치에 대한 레코드를 유지 관리합니다. 이 지시어는 FUNCTION 및 ENDFUNC 또는 PROC 및 ENDP 지시어가 있는 함수 내에서만 사용할 수 있습니다.

구문

FRAME REGISTER *reg1*,*reg2*

인수 설명:

reg1 함수 진입점에 있는 인수가 포함된 레지스터입니다.

reg2 값이 저장되는 레지스터입니다.

사용법

레지스터를 사용하여 함수 진입점에 있는 다른 레지스터에 포함된 인수를 저장할 때 FRAME REGISTER 지시어를 사용합니다.

7.5.5 FRAME RESTORE

FRAME RESTORE 지시어를 사용하여 지정한 레지스터의 내용이 함수 진입점에 있는 값으로 복원되었음을 어셈블러에 알립니다. 이 지시어는 FUNCTION 및 ENDFUNC 또는 PROC 및 ENDP 지시어가 있는 함수 내에서만 사용할 수 있습니다.

구문

FRAME RESTORE {*reglist*}

인수 설명:

reglist 해당 내용이 복원된 레지스터 목록입니다. 이 목록에는 최소한 하나 이상의 레지스터가 있어야 합니다.

사용법

호출 수신자가 스택에서 레지스터를 다시 로드한 후 바로 FRAME RESTORE를 사용합니다. 함수의 마지막 명령어 뒤에서는 이렇게 하지 않아도 됩니다.

*reglist*에는 정수 레지스터나 부동 소수점 레지스터 중 하나만 포함될 수 있으며 둘 다 포함될 수는 없습니다.

참고

코드에서 단일 명령어를 사용하여 레지스터를 로드하고 스택 포인터를 변경하는 경우 FRAME RESTORE와 FRAME ADDRESS를 둘 다 사용하는 대신 FRAME POP을 사용할 수 있습니다 (7-45페이지의 *FRAME POP* 참조).

7.5.6 FRAME RETURN ADDRESS

FRAME RETURN ADDRESS 지시어는 복귀 주소로 r14 이외의 레지스터를 사용하는 함수를 제공합니다. 이 지시어는 FUNCTION 및 ENDFUNC 또는 PROC 및 ENDP 지시어가 있는 함수 내에서만 사용할 수 있습니다.

참고

복귀 주소로 r14 이외의 레지스터를 사용하는 함수는 AAPCS와 호환되지 않습니다. 이러한 함수는 내보내면 안 됩니다.

구문

FRAME RETURN ADDRESS *reg*

인수 설명:

reg 복귀 주소에 사용되는 레지스터입니다.

사용법

FRAME RETURN ADDRESS 지시어를 복귀 주소로 r14를 사용하지 않는 모든 함수에 사용합니다. 이렇게 하지 않으면 디버거가 함수를 백트레이싱할 수 없습니다.

함수를 시작하는 FUNCTION 또는 PROC 지시어 바로 다음에 FRAME RETURN ADDRESS를 추가합니다.

7.5.7 FRAME SAVE

FRAME SAVE 지시어는 표준 프레임 주소를 기준으로 하는 저장된 레지스터 내용의 위치를 설명합니다. 이 지시어는 FUNCTION 및 ENDFUNC 또는 PROC 및 ENDP 지시어가 있는 함수 내에서만 사용할 수 있습니다.

구문

FRAME SAVE {*reglist*}, *offset*

인수 설명:

reglist 표준 프레임 주소의 *offset*에서 시작하여 연속적으로 저장된 레지스터 목록입니다. 이 목록에는 최소한 하나 이상의 레지스터가 있어야 합니다.

사용법

호출 수신자가 스택에서 레지스터를 다시 로드한 후 바로 FRAME RESTORE를 사용합니다.

reglist 에는 백트레이싱할 필요가 없는 레지스터가 포함될 수 있습니다. 어셈블러는 DWARF 호출 프레임 정보에 기록해야 하는 레지스터를 확인합니다.

참고

코드에서 단일 명령어를 사용하여 레지스터를 저장하고 스택 포인터를 변경하는 경우 FRAME SAVE 및 FRAME ADDRESS를 둘 다 사용하는 대신 FRAME PUSH를 사용할 수 있습니다 (7-46페이지의 *FRAME PUSH* 참조).

7.5.8 FRAME STATE REMEMBER

FRAME STATE REMEMBER 지시어는 표준 프레임 주소와 저장된 레지스터 값의 위치를 계산하는 방법에 대한 현재 정보를 저장합니다. 이 지시어는 FUNCTION 및 ENDFUNC 또는 PROC 및 ENDP 지시어가 있는 함수 내에서만 사용할 수 있습니다.

구문

FRAME STATE REMEMBER

사용법

인라인 종료 시퀀스 중에 표준 프레임 주소와 저장된 레지스터 값의 위치를 계산하는 방법에 대한 정보가 변경될 수 있습니다. 종료 시퀀스 후에는 이전과 동일한 정보를 사용하여 다른 분기가 계속될 수 있습니다. FRAME STATE REMEMBER를 사용하면 이 정보를 저장할 수 있고 FRAME STATE RESTORE를 사용하면 이 정보를 복원할 수 있습니다.

이러한 지시어는 중첩될 수 있습니다. 각 FRAME STATE RESTORE 지시어에는 대응하는 FRAME STATE REMEMBER 지시어가 있어야 합니다. 다음을 참조하십시오.

- 7-53페이지의 *FRAME STATE RESTORE*
- 7-55페이지의 *FUNCTION 또는 PROC*

예제

```
        ; function code
        FRAME STATE REMEMBER
            ; save frame state before in-line exit sequence
        POP     {r4-r6,pc}
            ; do not have to FRAME POP here, as control has
            ; transferred out of the function
        FRAME STATE RESTORE
            ; end of exit sequence, so restore state
exitB   ; code for exitB
        POP     {r4-r6,pc}
        ENDP
```

7.5.9 FRAME STATE RESTORE

FRAME STATE RESTORE 지시어는 표준 프레임 주소와 저장된 레지스터 값의 위치를 계산하는 방법에 대한 정보를 복원합니다. 이 지시어는 FUNCTION 및 ENDFUNC 또는 PROC 및 ENDP 지시어가 있는 함수 내에서만 사용할 수 있습니다.

구문

```
FRAME STATE RESTORE
```

사용법

참조:

- 7-52페이지의 *FRAME STATE REMEMBER*
- 7-55페이지의 *FUNCTION 또는 PROC*

7.5.10 FRAME UNWIND ON

FRAME UNWIND ON 지시어는 현재 함수를 포함한 이후의 모든 함수에 대한 *해제* 테이블을 생성하도록 어셈블러에 지시합니다.

구문

```
FRAME UNWIND ON
```

사용법

이 지시어는 함수 외부에서 사용할 수 있습니다. 이 경우 어셈블러는 FRAME UNWIND OFF 지시어에 도달할 때까지 이후의 모든 함수에 대한 *해제* 테이블을 생성합니다.

참고

FRAME UNWIND 지시어로는 예외 테이블 생성을 설정할 수 없습니다. 또한 다른 FRAME 지시어가 없는 FRAME UNWIND 지시어에는 어셈블러가 *해제* 정보를 생성하는 데 충분한 정보가 없습니다.

3-20페이지의 *예외 테이블 생성 제어*도 참조하십시오.

7.5.11 FRAME UNWIND OFF

FRAME UNWIND OFF 지시어는 현재 함수와 이후의 모든 함수에 대한 *해제 없음* 테이블을 생성하도록 어셈블러에 지시합니다.

구문

```
FRAME UNWIND OFF
```

사용법

이 지시어는 함수 외부에서 사용할 수 있습니다. 이 경우 어셈블러는 FRAME UNWIND ON 지시어에 도달할 때까지 이후의 모든 함수에 대한 *해제 없음* 테이블을 생성합니다.

3-20페이지의 *예외 테이블 생성 제어*도 참조하십시오.

7.5.12 FUNCTION 또는 PROC

FUNCTION 지시어는 함수의 시작을 표시합니다. PROC는 FUNCTION의 동의어입니다.

구문

label FUNCTION [{*reglist1*} [, {*reglist2*}]]

인수 설명:

reglist1 호출 수신자가 저장하는 ARM 레지스터의 선택적 목록입니다. *reglist1*이 없을 경우 디버거는 레지스터 사용을 검사할 때 AAPCS가 사용 중이라고 간주합니다.

reglist2 호출 수신자가 저장하는 VFP 레지스터의 선택적 목록입니다.

사용법

FUNCTION을 사용하여 함수의 시작을 표시합니다. 어셈블러는 FUNCTION을 사용하여 ELF에 대한 DWARF 호출 프레임 정보를 생성할 때 함수의 시작을 확인합니다.

FUNCTION은 표준 프레임 주소를 r13 (sp) 으로 설정하고 프레임 상태 스택을 빈 상태로 설정합니다.

각 FUNCTION 지시어에는 대응하는 ENDFUNC 지시어가 있어야 합니다. FUNCTION 및 ENDFUNC 쌍은 중첩될 수 없으며, PROC 또는 ENDP 지시어를 포함할 수 없습니다.

사용자 고유의 디버거를 사용하는 경우 선택적 *reglist* 매개변수를 사용하여 대체 프로시저 호출 표준에 대해 디버거에 알릴 수 있습니다. 일부 디버거에서는 이 기능이 지원하지 않습니다. 자세한 내용은 디버거 설명서를 참조하십시오.

7-43페이지의 *FRAME ADDRESS* ~ 7-53페이지의 *FRAME STATE RESTORE*도 참조하십시오.

참고

FUNCTION은 자동으로 워드 단위 (Thumb의 경우 하프워드 단위) 로 정렬되도록 하지 않습니다. 정렬이 필요하면 ALIGN을 사용합니다. 그렇지 않으면 호출 프레임이 함수의 시작을 가리키지 않을 수 있습니다. 자세한 내용은 7-67페이지의 *ALIGN*을 참조하십시오.

예제

```
        ALIGN      ; ensures alignment
dadd    FUNCTION   ; without the ALIGN directive, this might not be word-aligned
        EXPORT  dadd
        PUSH       {r4-r6,lr}    ; this line automatically word-aligned
        FRAME PUSH {r4-r6,lr}
        ; subroutine body
        POP        {r4-r6,pc}
        ENDFUNC
func6   PROC {r4-r8,r12},{D1-D3} ; non-AAPCS-conforming function
        ...
        ENDP
```

7.5.13 ENDFUNC 또는 ENDP

ENDFUNC 지시어는 AAPCS 준수 함수의 끝을 표시합니다 (7-55페이지의 *FUNCTION 또는 PROC* 참조). ENDP는 ENDFUNC의 동의어입니다.

7.6 보고 지시어

이 단원에서는 다음 지시어에 대해 설명합니다.

- *ASSERT*
  어셈블리 중에 어설션이 실패할 경우 오류 메시지를 생성합니다.
- 7-59페이지의 *INFO*
  어셈블리 중에 진단 정보를 생성합니다.
- 7-60페이지의 *OPT*
  목록 옵션을 설정합니다.
- 7-62페이지의 *TTL 및 SUBT*
  목록에 제목과 부제목을 삽입합니다.

7.6.1 ASSERT

ASSERT 지시어는 지정한 어설션이 실패할 경우 어셈블리의 두 번째 패스 중에 오류 메시지를 생성합니다.

구문

ASSERT *logical-expression*

인수 설명:

logical-expression

{TRUE} 또는 {FALSE}로 평가될 수 있는 어설션입니다.

사용법

ASSERT를 사용하여 어셈블리 중에 필요한 모든 조건이 충족되도록 합니다.

어설션이 실패할 경우 오류 메시지가 생성되고 어셈블리가 실패합니다.

7-59페이지의 *INFO*도 참조하십시오.

예제

```
ASSERT  label1 <= label2    ; Tests if the address
                            ; represented by label1
                            ; is <= the address
                            ; represented by label2.
```

7.6.2 INFO

INFO 지시어는 어셈블리 패스에서 진단 정보를 생성할 수 있도록 합니다.

! 기호는 간단한 진단 정보를 생성한다는 점을 제외하고 INFO와 매우 유사합니다.

구문

INFO *numeric-expression, string-expression*

인수 설명:

numeric-expression

어셈블리 중에 평가되는 숫자 식입니다. 이 식이 0으로 평가되면 다음과 같이 됩니다.

- 첫 번째 패스 중에 아무런 작업도 수행되지 않습니다.
- 두 번째 패스 중에 *string-expression*이 인쇄됩니다.

이 식이 0으로 평가되지 않으면 *string-expression*이 오류 메시지로 인쇄되고 어셈블리가 실패합니다.

string-expression

문자열로 평가되는 식입니다.

사용법

INFO는 사용자 지정 오류 메시지를 생성하는 유연한 방법을 제공합니다. 숫자 및 문자열 식에 대한 자세한 내용은 3-35페이지의 *숫자 식* 및 3-34페이지의 *문자열 식*을 참조하십시오.

7-57페이지의 *ASSERT*도 참조하십시오.

예제

```
INFO    0, "Version 1.0"
IF endofdata <= label1
    INFO    4, "Data overrun at label1"
ENDIF
```

7.6.3 OPT

OPT 지시어는 소스 코드 내에서 목록 옵션을 설정합니다.

구문

OPT *n*

인수 설명:

n OPT 지시어 설정값입니다. 표 7-2에서는 유효한 설정값을 보여 줍니다.

표 7-2 OPT 지시어 설정값

OPT n	효과
1	일반 목록을 설정합니다.
2	일반 목록을 해제합니다.
4	페이지 넘김. 폼 피드를 실행하고 새 페이지를 시작합니다.
8	행 번호 카운터를 0으로 리셋합니다.
16	SET, GBL 및 LCL 지시어의 목록을 설정합니다.
32	SET, GBL 및 LCL 지시어의 목록을 해제합니다.
64	매크로 확장 목록을 설정합니다.
128	매크로 확장 목록을 해제합니다.
256	매크로 호출 목록을 설정합니다.
512	매크로 호출 목록을 해제합니다.
1024	첫 번째 패스 목록을 설정합니다.
2048	첫 번째 패스 목록을 해제합니다.
4096	조건부 지시어의 목록을 설정합니다.
8192	조건부 지시어의 목록을 해제합니다.
16384	MEND 지시어의 목록을 설정합니다.
32768	MEND 지시어의 목록을 해제합니다.

사용법

--list= 어셈블러 옵션을 지정하여 목록을 설정합니다.

기본적으로 --list= 옵션은 변수 선언, 매크로 확장, 호출 조건부 지시어 및 MEND 지시어가 포함된 일반 목록을 생성합니다. 이 목록은 두 번째 패스에서만 생성됩니다. OPT 지시어를 사용하면 소스 코드 내에서 기본 목록 옵션을 수정할 수 있습니다. --list= 옵션에 대한 자세한 내용은 3-17페이지의 *파일에 출력 나열*을 참조하십시오.

OPT를 사용하여 코드 목록의 형식을 지정할 수 있습니다. 예를 들어 함수와 섹션 앞에 새 페이지를 지정할 수 있습니다.

예제

```
        AREA    Example, CODE, READONLY
start   ; code
        ; code
        BL      func1
        ; code
        OPT 4               ; places a page break before func1
func1   ; code
```

7.6.4 TTL 및 SUBT

TTL 지시어는 목록 파일에 있는 각 페이지의 시작 부분에 제목을 삽입합니다. 제목은 TTL 지시어가 나타날 때까지 각 페이지에 인쇄됩니다.

SUBT 지시어는 목록 파일의 각 페이지에 부제목을 삽입합니다. 부제목은 SUBT 지시어가 나타날 때까지 각 페이지에 인쇄됩니다.

구문

TTL *title*

SUBT *subtitle*

인수 설명:

title 제목입니다.

subtitle 부제목입니다.

사용법

TTL 지시어를 사용하여 목록 파일에 있는 각 페이지의 맨 위에 제목을 삽입합니다. 첫 페이지에 제목을 표시하려면 TTL 지시어가 소스 파일의 첫 행에 있어야 합니다.

추가 TTL 지시어를 사용하여 제목을 변경합니다. 각 새 TTL 지시어는 다음 페이지의 맨 위에서부터 적용됩니다.

SUBT를 사용하여 목록 파일에 있는 각 페이지의 맨 위에 부제목을 삽입합니다. 부제목은 제목 행 아래에 표시됩니다. 첫 페이지에 부제목을 표시하려면 SUBT 지시어가 소스 파일의 첫 행에 있어야 합니다.

추가 SUBT 지시어를 사용하여 부제목을 변경합니다. 각 새 SUBT 지시어는 다음 페이지의 맨 위에서부터 적용됩니다.

예제

```
TTL     First Title     ; places a title on the first
                        ; and subsequent pages of a
                        ; listing file.
SUBT    First Subtitle  ; places a subtitle on the
                        ; second and subsequent pages
                        ; of a listing file.
```

7.7 명령어 세트 및 구문 선택 지시어

이 단원에서는 다음 지시어에 대해 설명합니다.

- 7-64페이지의 *ARM, THUMB, THUMBX, CODE16 및 CODE32*

7.7.1 ARM, THUMB, THUMBX, CODE16 및 CODE32

ARM 지시어와 CODE32 지시어는 같습니다. 이 두 지시어는 UAL 또는 Thumb-2 이전 ARM 어셈블러 언어 구문을 사용하여 후속 명령어를 ARM 명령어로 해석하도록 어셈블러에 지시합니다.

THUMB 지시어는 UAL 구문을 사용하여 후속 명령어를 Thumb 명령어로 해석하도록 어셈블러에 지시합니다.

THUMBX 지시어는 UAL 구문을 사용하여 후속 명령어를 Thumb-2EE 명령어로 해석하도록 어셈블러에 지시합니다.

CODE16 지시어는 UAL 이전 어셈블리 언어 구문을 사용하여 후속 명령어를 Thumb 명령어로 해석하도록 어셈블러에 지시합니다.

또한 필요한 경우 이들 지시어는 최대 3바이트의 패딩을 삽입하여 다음 워드 경계에 맞게 정렬 (ARM) 하거나, 최대 1바이트의 패딩을 삽입하여 다음 하프워드 경계에 맞게 정렬 (Thumb 또는 Thumb-2EE) 합니다.

구문

```
ARM
THUMB
THUMBX
CODE16
CODE32
```

사용법

다양한 명령어 세트를 사용하는 코드가 포함된 파일에서 다음과 같이 합니다.

- ARM 지시어가 ARM 코드 앞에 와야 합니다. CODE32는 ARM의 동의어입니다.
- THUMB 지시어가 UAL 구문으로 작성된 Thumb 코드 앞에 와야 합니다.
- THUMBX 지시어가 UAL 구문으로 작성된 Thumb-2EE 코드 앞에 와야 합니다.
- CODE16 지시어가 UAL 이전 구문으로 작성된 Thumb 코드 앞에 와야 합니다.

이러한 지시어는 명령어로 어셈블되지 않으며 상태를 변경하지도 않습니다. 다만 ARM, Thumb 또는 Thumb-2EE 또는 Thumb 명령어를 적절하게 어셈블하도록 어셈블러에 지시하고, 필요한 경우 패딩을 삽입합니다.

예제

이 예제에서는 ARM 및 THUMB을 사용하여 ARM 명령어에서 Thumb 명령어로 분기하는 방법을 보여 줍니다.

```
        AREA ToThumb, CODE, READONLY    ; Name this block of code
        ENTRY                           ; Mark first instruction to execute
        ARM                             ; Subsequent instructions are ARM
start
        ADR     r0, into_thumb + 1      ; Processor starts in ARM state
        BX      r0                      ; Inline switch to Thumb state
        THUMB                           ; Subsequent instructions are Thumb
into_thumb
        MOVS    r0, #10                 ; New-style Thumb instructions
```

7.8 기타 지시어

이 단원에서는 다음 지시어에 대해 설명합니다.

- 7-67페이지의 *ALIGN*
- 7-70페이지의 *AREA*
- 7-74페이지의 *ATTR*
- 7-76페이지의 *END*
- 7-76페이지의 *ENTRY*
- 7-77페이지의 *EQU*
- 7-78페이지의 *EXPORT 또는 GLOBAL*
- 7-80페이지의 *EXPORTAS*
- 7-81페이지의 *GET 또는 INCLUDE*
- 7-82페이지의 *IMPORT 및 EXTERN*
- 7-84페이지의 *INCBIN*
- 7-85페이지의 *KEEP*
- 7-86페이지의 *NOFP*
- 7-86페이지의 *REQUIRE*
- 7-87페이지의 *REQUIRE8 및 PRESERVE8*
- 7-88페이지의 *ROUT*

7.8.1 ALIGN

ALIGN 지시어는 NOP 명령어나 0으로 패딩하여 지정한 경계에 맞춰 현재 위치를 정렬합니다.

구문

ALIGN {*expr*{,*offset*{,*pad*{,*padsize*}}}}

인수 설명:

expr	2^0에서 2^{31} 사이의 2의 제곱으로 평가되는 숫자 식입니다.
offset	임의의 숫자 식일 수 있습니다.
pad	임의의 숫자 식일 수 있습니다.
padsize	1, 2 또는 4일 수 있습니다.

연산

현재 위치는 아래 형식의 다음 주소에 맞춰 정렬됩니다.

offset + *n* * *expr*

*expr*을 지정하지 않으면 ALIGN이 현재 위치를 다음 워드 (4바이트) 경계에 설정합니다. 이전 위치와 새 현재 위치 사이의 사용하지 않는 바이트는 다음으로 채워집니다.

- *pad*를 지정한 경우, *pad*의 복사본다음 조건을 모두 만족하는 경우,
- NOP 명령어
  - *pad*를 지정하지 않은 경우
  - ALIGN 지시어가 ARM 또는 Thumb 명령어 뒤에 나오는 경우
  - 현재 섹션에서 CODEALIGN 특성이 AREA 지시어에 설정되어 있는 경우
- 그 외의 경우는 0

*pad*는 *padsize*의 값에 따라 바이트, 하프워드 또는 워드로 처리됩니다. *padsize*를 지정하지 않으면 *pad*가 데이터 섹션에서는 바이트로, Thumb 코드에서는 하프워드로, ARM 코드에서는 워드로 기본적으로 처리됩니다.

사용법

ALIGN을 사용하여 데이터와 코드가 적절한 경계에 맞춰 정렬되도록 합니다. 일반적으로 이 지시어는 다음 경우에 필요합니다.

- ADR Thumb 의사 명령어가 워드로 정렬된 주소만 로드할 수 있지만 Thumb 코드 내의 레이블이 워드로 정렬되지 않을 수 있습니다. ALIGN 4를 사용하면 Thumb 코드 내의 주소를 4바이트로 정렬할 수 있습니다.
- ALIGN을 통해 일부 ARM 프로세서에서 캐시를 이용합니다. 예를 들어 ARM940T에서 16바이트 행이 있는 캐시를 이용합니다. ALIGN 16을 사용하면 16바이트 단위로 함수 항목을 정렬하고 캐시 효율성을 최대화할 수 있습니다.
- LDRD 및 STRD 더블워드 데이터 전송이 8바이트로 정렬되어야 합니다. LDRD 또는 STRD를 사용하여 데이터에 액세스해야 할 경우 DCQ (7-16페이지의 *데이터 정의 지시어* 참조) 같은 메모리 할당 지시어 앞에 ALIGN 8을 사용합니다.
- 행의 레이블은 자동으로 임의로 정렬될 수 있습니다. 다음 ARM 코드는 워드로 정렬됩니다 (Thumb 코드의 경우 하프워드로 정렬됨). 따라서 레이블이 코드 주소를 올바로 지정하지 않습니다. 레이블 앞에 ALIGN 4 (Thumb의 경우, ALIGN 2) 를 사용합니다.

정렬은 루틴이 있는 ELF 섹션의 시작 부분을 기준으로 지정됩니다. 이 섹션은 동일하거나 덜 정교한 경계에 맞춰 정렬되어야 합니다. AREA 지시어의 ALIGN 특성은 이와 다르게 지정됩니다 (7-70페이지의 *AREA* 및 7-69페이지의 *예제* 참조).

예제

```
        AREA    cacheable, CODE, ALIGN=3
rout1   ; code              ; aligned on 8-byte boundary
        ; code
        MOV     pc,lr       ; aligned only on 4-byte boundary
        ALIGN   8           ; now aligned on 8-byte boundary
rout2   ; code


        AREA    OffsetExample, CODE
        DCB     1           ; This example places the two
        ALIGN   4,3         ; bytes in the first and fourth
        DCB     1           ; bytes of the same word.


        AREA    Example, CODE, READONLY
start   LDR     r6,=label1
        ; code
        MOV     pc,lr
label1  DCB     1           ; pc now misaligned
        ALIGN               ; ensures that subroutine1 addresses
subroutine1                 ; the following instruction.
        MOV r5,#0x5
```

7.8.2 AREA

AREA 지시어는 새 코드 또는 데이터 섹션을 어셈블하도록 어셈블러에 지시합니다. 섹션은 링커에서 조작하는 코드 또는 데이터의 더 이상 나눌 수 없는 독립적인 명명된 청크입니다. 자세한 내용은 2-16페이지의 *ELF 섹션 및 AREA 지시어*를 참조하십시오.

구문

AREA *sectionname*{,*attr*}{,*attr*}...

인수 설명:

sectionname
섹션에 지정할 이름입니다.

아무 이름이나 선택할 수 있지만 영문자 이외의 문자로 시작하는 이름은 막대로 묶어야 합니다. 그렇지 않으면 섹션 이름이 없다는 오류 메시지가 생성됩니다. 예를 들면 |1_DataArea|와 같습니다.

일부 이름은 기존 방식을 사용하여 지정됩니다. 예를 들어 |.text|는 C 컴파일러가 생성한 코드 섹션이나 C 라이브러리와 다른 방식으로 연결된 코드 섹션에 사용됩니다.

attr
하나 이상의 콤마로 구분된 섹션 특성입니다. 유효한 특성은 다음과 같습니다.

ALIGN=*expression*
기본적으로 ELF 섹션은 4바이트 단위로 정렬됩니다. *expression*은 0에서 31 사이의 모든 정수 값을 가질 수 있습니다. 섹션은 2*expression*-바이트 경계를 기준으로 정렬됩니다. 예를 들어 *expression*이 10이면 섹션은 1KB 경계를 기준으로 정렬됩니다.
이 지시어는 ALIGN 지시어와 다른 방법으로 지정됩니다. 자세한 내용은 7-67페이지의 *ALIGN*을 참조하십시오.

참고

ARM 코드 섹션에 ALIGN=0 또는 ALIGN=1을 사용하면 안 됩니다.

Thumb 코드 섹션에 ALIGN=0을 사용하면 안 됩니다.

ASSOC=*section*

*section*은 연결된 ELF 섹션을 지정합니다.

*sectionname*은 *section*이 들어 있는 모든 링크에 포함되어야 합니다.

CODE 시스템 명령어를 포함합니다. READONLY가 기본값입니다.

CODEALIGN

ALIGN 지시어가 다른 패딩을 지정하지 않는 경우, 섹션 내의 ARM 또는 Thumb 명령어 뒤에 ALIGN 지시어가 사용되면 NOP 명령어를 삽입하도록 어셈블러에 지시합니다.

COMDEF 공통 섹션 정의입니다. 이 ELF 섹션은 코드나 데이터를 포함할 수 있으며 다른 소스 파일에서 같은 이름을 가진 다른 모든 섹션과 동일해야 합니다.

같은 이름을 가진 동일한 ELF 섹션은 링커에서 같은 메모리 섹션에 겹쳐서 배치합니다. 다른 ELF 섹션이 있을 경우 링커에서 경고를 생성하고 섹션을 겹쳐서 배치하지 않습니다. *링커 사용 설명서*에서 3장 *기본 링커 기능 사용*을 참조하십시오.

COMGROUP=*symbol_name*

공통 그룹 섹션입니다. 공통 그룹 안의 모든 섹션은 공통입니다. 객체가 링크될 때 다른 객체 파일에 *symbol_name* 서명이 포함된 GROUP이 있을 수 있습니다. 한 그룹만 최종 이미지에 유지됩니다.

COMMON 공통 데이터 정의입니다. 이 안에 들어 있는 코드나 데이터는 사용자가 정의할 수 없으며 링커에서 0으로 초기화합니다. 같은 이름을 가진 모든 공통 섹션은 링커에서 같은 메모리 섹션에 겹쳐서 배치합니다. 이러한 공통 섹션은 모두 크기가 달라야 합니다. 링커에서는 각 이름의 가장 큰 공통 섹션에 필요한 공간을 할당합니다.

DATA 명령어가 아니라 데이터를 포함합니다. READWRITE가 기본값입니다.

FINI_ARRAY

현재 영역의 ELF 형식을 SHT_FINI_ARRAY로 설정합니다.

GROUP=*symbol_name*

그룹에 대한 서명이며 소스 파일이나 소스 파일에 포함된 파일로 정의되어야 합니다. 동일한 *symbol_name* 서명이 포함된 모든 AREAS가 동일한 그룹에 배치됩니다. 그룹 내의 섹션은 유지되거나 함께 검색됩니다.

INIT_ARRAY

현재 영역의 ELF 형식을 SHT_INIT_ARRAY로 설정합니다.

LINKORDER=*section*

이미지에서 현재 섹션의 상대 위치를 지정합니다. 그러면 LINKORDER 속성을 포함하는 모든 섹션의 순서가 서로에 대해 이미지의 명명된 해당 *sections* 순서와 같아집니다.

MERGE=*n* 링커가 MERGE=*n* 속성을 사용하여 현재 섹션을 다른 섹션과 병합할 수 있음을 나타냅니다. *n*은 섹션의 요소 크기 (예: 문자의 경우 *n*은 1) 입니다. 그러나 이 속성이 링커가 섹션을 병합하도록 지시하는 것은 아니므로, 섹션이 병합될 것이라고 간주하면 안 됩니다.

NOALLOC 타겟 시스템의 메모리가 현재 영역에 할당되지 않았음을 나타냅니다.

NOINIT 데이터 섹션이 초기화되지 않았거나 0으로 초기화되었음을 나타냅니다. 여기에는 공간 예약 지시어인 SPACE나 0으로 초기화된 값이 있는 DCB, DCD, DCDU, DCQ, DCQU, DCW 또는 DCWU만 포함될 수 있습니다. 영역이 초기화되지 않았는지 아니면 0으로 초기화되었는지 여부는 링크 타임에 확인할 수 있습니다. *링커 사용 설명서*에서 3장 *기본 링커 기능 사용*을 참조하십시오.

PREINIT_ARRAY

현재 영역의 ELF 형식을 SHT_PREINIT_ARRAY로 설정합니다.

READONLY 현재 섹션에서 쓰기 작업을 수행할 수 없음을 나타냅니다. 이 값이 코드 영역의 기본값입니다.

READWRITE 현재 섹션에서 읽기 및 쓰기 작업을 수행할 수 있음을 나타냅니다. 이 값이 데이터 영역의 기본값입니다.

SECFLAGS=*n*

*n*으로 표시되는 하나 이상의 ELF 플래그를 현재 섹션에 추가합니다.

SECTYPE=*n*

현재 섹션의 ELF 형식을 *n*으로 설정합니다.

STRINGS SHF_STRINGS 플래그를 현재 섹션에 추가합니다. STRINGS 속성을 사용하려면 MERGE=1 속성도 사용해야 합니다. 섹션의 내용은 DCB 지시어를 사용하여 Null로 끝나는 문자열이어야 합니다.

사용법

AREA 지시어를 사용하여 소스 파일을 여러 ELF 섹션으로 나눕니다. 둘 이상의 AREA 지시어에 같은 이름을 사용할 수 있습니다. 같은 이름을 가진 모든 영역은 동일한 ELF 섹션에 배치됩니다. 이 경우 첫 번째 AREA 지시어의 특성만 적용됩니다.

일반적으로 코드와 데이터에 대해 각기 다른 ELF 섹션을 사용해야 합니다. 큰 프로그램은 보통 여러 코드 섹션으로 쉽게 나눌 수 있습니다. 대개 큰 독립적 데이터 세트는 여러 섹션에 배치하는 것이 좋습니다.

지역 레이블의 범위는 AREA 지시어에 의해 정의되며, 경우에 따라 ROUT 지시어에 의해 나뉩니다 (3-32페이지의 *지역 레이블* 및 7-88페이지의 *ROUT* 참조).

어셈블리에 대한 AREA 지시어는 최소한 하나 이상 있어야 합니다.

예제

다음 예제에서는 Example이라는 읽기 전용 코드 섹션을 정의합니다.

```
AREA    Example,CODE,READONLY   ; An example code section.
        ; code
```

7.8.3 ATTR

ATTR set 지시어는 ABI 빌드 속성 값을 설정합니다.

ATTR 범위 지시어는 설정한 값을 적용할 범위를 지정합니다.

구문

```
ATTR FILESCOPE
ATTR SCOPE name
ATTR settype, tagid, value
```

인수 설명:

name 섹션 이름 또는 기호 이름입니다.

settype 다음 중 하나일 수 있습니다.

- SETVALUE
- SETSTRING
- SETCOMPATIBLEWITHVALUE
- SETCOMPATIBLEWITHSTRING

tagid ARM 아키텍처용 ABI에 정의된 속성 태그 이름 또는 해당 숫자 값입니다.

value *settype*에 따라 달라집니다.

- *settype*이 SETVALUE 또는 SETCOMPATIBLEWITHVALUE이면 32비트 정수 값입니다.
- *settype*이 SETSTRING 또는 SETCOMPATIBLEWITHSTRING이면 Null로 끝나는 문자열입니다.

사용법

ATTR FILESCOPE 지시어 뒤에 오는 ATTR set 지시어는 전체 객체 파일에 적용됩니다. ATTR SCOPE *name* 지시어 뒤에 오는 ATTR set 지시어는 명명된 섹션 또는 기호에만 적용됩니다.

정수가 필요한 태그에는 SETVALUE 또는 SETCOMPATIBLEWITHVALUE를 사용해야 합니다. 문자열이 필요한 태그에는 SETSTRING 또는 SETCOMPATIBLEWITHSTRING을 사용해야 합니다. 태그 이름 목록을 보려면 *ARM 아키텍처용 ABI의 추가 목록 및 정오표*를 참조하십시오.

객체 파일도 호환되는 태그 값을 설정하려면 SETCOMPATIBLEWITHVALUE 및 SETCOMPATIBLEWITHSTRING을 사용합니다.

예제

```
ATTR  SETSTRING Tag_CPU_raw_name, "Cortex-A8"
ATTR  SETVALUE  Tag_VFP_arch, 3   ; VFPv3 instructions were permitted.
ATTR  SETVALUE  10, 3             ; 10 is the numerical value of
                                  ; Tag_VFP_arch.
```

7.8.4 END

END 지시어는 소스 파일의 끝에 도달하면 이를 어셈블러에 알립니다.

구문

END

사용법

모든 어셈블리 언어 소스 파일은 별도의 행에서 END로 끝나야 합니다.

GET 지시어가 상위 파일에 소스 파일을 포함시킨 경우 어셈블러는 상위 파일로 돌아가서 GET 지시어 다음에 나오는 첫 번째 행에서 어셈블리를 계속합니다. 자세한 내용은 7-81페이지의 *GET 또는 INCLUDE*를 참조하십시오.

END가 첫 번째 패스 중에 오류 없이 최상위 소스 파일에 도달하면 두 번째 패스가 시작됩니다.

END가 두 번째 패스 중에 최상위 소스 파일에 도달하면 어셈블리가 완료되고 적절한 출력이 기록됩니다.

7.8.5 ENTRY

ENTRY 지시어는 프로그램에 대한 진입점을 선언합니다.

구문

ENTRY

사용법

프로그램에 대한 ENTRY 포인트를 최소한 하나 이상 지정해야 합니다. ENTRY가 없으면 링크 타임에 경고가 생성됩니다.

단일 소스 파일에서 둘 이상의 ENTRY 지시어를 사용하면 안 됩니다. 모든 소스 파일에 ENTRY 지시어가 있을 필요는 없습니다. 단일 소스 파일에 둘 이상의 ENTRY가 있으면 어셈블리 타임에 오류 메시지가 생성됩니다.

예제

```
AREA    ARMex, CODE, READONLY
ENTRY                   ; Entry point for the application
```

7.8.6 EQU

EQU 지시어는 숫자 상수, 레지스터 상대 값 또는 프로그램 기준 값에 기호 이름을 지정합니다. * 기호는 EQU의 동의어입니다.

구문

name EQU *expr*{, *type*}

인수 설명:

name 값에 지정할 기호 이름입니다.

expr 레지스터 기준 주소, 프로그램 기준 주소, 절대 주소 또는 32비트 정수 상수입니다.

type 선택 사항입니다. *type*은 다음 중 하나일 수 있습니다.

- ARM
- THUMB
- CODE32
- CODE16
- DATA

*type*은 *expr*이 절대 주소일 경우에만 사용할 수 있습니다. *name*을 내보낸 경우 객체 파일에 있는 기호 테이블의 *name* 항목은 *type*에 따라 ARM, THUMB, CODE32, CODE16 또는 DATA로 표시됩니다. 이 항목은 링커에서 사용할 수 있습니다.

사용법

EQU를 사용하여 상수를 정의합니다. 이 지시어는 C에서 **#define**을 사용하여 상수를 정의하는 것과 같습니다.

기호 내보내기에 대한 자세한 내용은 7-85페이지의 *KEEP* 및 7-78페이지의 *EXPORT 또는 GLOBAL*을 참조하십시오.

예제

```
abc EQU 2               ; assigns the value 2 to the symbol abc.
xyz EQU label+8         ; assigns the address (label+8) to the
                        ; symbol xyz.
fiq EQU 0x1C, CODE32    ; assigns the absolute address 0x1C to
                        ; the symbol fiq, and marks it as code
```

7.8.7 EXPORT 또는 GLOBAL

EXPORT 지시어는 별도의 객체 및 라이브러리 파일에 있는 기호 참조를 확인하기 위해 링커에서 사용할 수 있는 기호를 선언합니다. GLOBAL은 EXPORT의 동의어입니다.

구문

```
EXPORT {[WEAK]}
EXPORT symbol {[type]}
EXPORT symbol [attr{,type}]
EXPORT symbol [WEAK{,attr}{,type}]
```

인수 설명:

symbol 내보낼 기호 이름입니다. 기호 이름은 대소문자를 구분합니다. *symbol*을 생략하면 모든 기호가 내보내집니다.

WEAK 다른 소스에서 대체 *symbol*을 내보내지 않는 경우에만 *symbol*을 다른 소스로 가져오게 됩니다. [WEAK]를 *symbol* 없이 사용하면 내보낸 기호는 모두 weak 기호가 됩니다.

attr 다음 중 하나일 수 있습니다.

- DYNAMIC ELF 기호 표시를 STV_DEFAULT로 설정합니다.
- PROTECTED ELF 기호 표시를 STV_PROTECTED로 설정합니다.
- HIDDEN ELF 기호 표시를 STV_HIDDEN으로 설정합니다.
- INTERNAL ELF 기호 표시를 STV_INTERNAL로 설정합니다.

type 기호 형식을 지정합니다.

- DATA *symbol*은 소스를 어셈블 및 링크할 때 데이터로 간주됩니다.
- CODE *symbol*은 소스를 어셈블 및 링크할 때 코드로 간주됩니다.
- ELFTYPE=*n* *symbol*은 *n* 값으로 지정되는 특정 ELF 기호로 간주됩니다. 여기서 *n*은 0에서 15 사이의 숫자일 수 있습니다.

지정하지 않는 경우 어셈블러가 가장 적합한 형식을 결정합니다.

사용법

EXPORT를 사용하여 다른 파일의 코드에서 현재 파일의 기호에 액세스할 수 있도록 합니다.

다른 소스에 있는 *symbol*의 다른 인스턴스를 사용할 수 있는 경우 [WEAK] 특성을 사용하여 이 다른 인스턴스가 현재 인스턴스보다 우선함을 링커에 알립니다. [WEAK] 특성은 모든 기호 표시 속성과 함께 사용할 수 있습니다.

7-82페이지의 *IMPORT 및 EXTERN*도 참조하십시오.

기호 표시에 대한 자세한 내용은 www.infocenter.arm.com의 *ARM 아키텍처용 ELF* ABI 설명서를 참조하십시오.

예제

```
        AREA    Example,CODE,READONLY
        EXPORT  DoAdd           ; Export the function name
                                ; to be used by external
                                ; modules.
DoAdd   ADD     r0,r0,r1
```

내보내기가 중복될 경우 기호 표시가 오버라이드될 수 있습니다. 다음 예제에서는 마지막 EXPORT가 바인딩 및 표시 유형에 대한 우선권을 갖습니다.

```
        EXPORT  SymA[WEAK]      ; Export as weak-hidden
        EXPORT  SymA[DYNAMIC]   ; SymA becomes non-weak dynamic.
```

7.8.8 EXPORTAS

EXPORTAS 지시어는 소스 파일에 있는 다른 기호에 해당하는 기호를 객체 파일에 내보낼 수 있도록 합니다.

구문

EXPORTAS *symbol1*, *symbol2*

인수 설명:

symbol1 소스 파일에 있는 기호 이름입니다. *symbol1*은 이미 정의되어 있어야 합니다. 영역 이름, 레이블 또는 상수가 포함된 임의의 기호일 수 있습니다.

symbol2 객체 파일에 표시할 기호 이름입니다.

기호 이름은 대소문자를 구분합니다.

사용법

EXPORTAS를 사용하면 소스 파일에 있는 모든 인스턴스를 변경하지 않고도 객체 파일에 있는 기호를 변경할 수 있습니다.

7-78페이지의 *EXPORT 또는 GLOBAL*도 참조하십시오.

예제

```
    AREA data1, DATA       ; starts a new area data1
    AREA data2, DATA       ; starts a new area data2
    EXPORTAS data2, data1  ; the section symbol referred to as data2 will
                           ; appear in the object file string table as data1.
one EQU  2
    EXPORTAS one, two
    EXPORT one             ; the symbol 'two' will appear in the object
                           ; file's symbol table with the value 2.
```

7.8.9 GET 또는 INCLUDE

GET 지시어는 어셈블 중인 파일에 다른 파일을 포함시킵니다. 포함된 파일은 GET 지시어의 위치에서 어셈블됩니다. INCLUDE는 GET의 동의어입니다.

구문

GET *filename*

인수 설명:

filename 어셈블리에 포함할 파일의 이름입니다. 어셈블러에서는 UNIX 또는 MS-DOS 형식의 파일 이름을 사용할 수 있습니다.

사용법

GET은 어셈블리에 매크로 정의, EQU 및 저장 맵을 포함하는 데 유용합니다. 포함된 파일의 어셈블리가 완료되면 GET 지시어 다음에 나오는 행에서 어셈블리가 계속됩니다.

기본적으로 어셈블러는 포함된 파일의 현재 위치를 검색합니다. 현재 위치는 호출하는 파일이 있는 디렉토리입니다. -i 어셈블러 명령 행 옵션을 사용하면 검색 경로에 디렉토리를 추가할 수 있습니다. 공백이 포함된 파일 이름과 디렉토리 이름은 큰따옴표로 묶으면 안 됩니다.

포함된 파일에는 다른 파일을 포함하기 위한 추가 GET 지시어가 포함될 수 있습니다 (7-32페이지의 *중첩 지시어* 참조).

포함된 파일이 현재 위치와 다른 디렉토리에 있으면 포함된 파일이 끝날 때까지 이 디렉토리가 현재 위치가 됩니다. 포함된 파일이 끝나면 이전 현재 위치가 복원됩니다.

GET은 객체 파일을 포함하는 데 사용할 수 없습니다 (7-84페이지의 *INCBIN* 참조).

예제

```
AREA    Example, CODE, READONLY
GET     file1.s              ; includes file1 if it exists
                             ; in the current place.
GET     c:\project\file2.s   ; includes file2
GET     c:\Program files\file3.s  ; space is permitted
```

7.8.10 IMPORT 및 EXTERN

이러한 지시어는 현재 어셈블리에 정의되어 있지 않은 이름을 어셈블러에 제공합니다.

구문

directive symbol {[*type*]}

directive symbol [*attr*{,*type*}]

directive symbol [WEAK{,*attr*}{,*type*}]

인수 설명:

지시어 다음 중 하나일 수 있습니다.

- IMPORT 기호를 무조건 가져옵니다.
- EXTERN 현재 어셈블리에서 기호를 참조하는 경우에만 기호를 가져옵니다.

symbol 별도로 어셈블된 소스 파일, 객체 파일 또는 라이브러리에 정의된 기호 이름입니다. 기호 이름은 대소문자를 구분합니다.

WEAK 기호가 다른 위치에 정의되어 있지 않은 경우 링커에서 오류 메시지를 생성하지 않도록 합니다. 또한 링커에서 아직 포함되어 있지 않은 라이브러리를 검색하지 않도록 합니다.

attr 다음 중 하나일 수 있습니다.

- DYNAMIC ELF 기호 표시를 STV_DEFAULT로 설정합니다.
- PROTECTED ELF 기호 표시를 STV_PROTECTED로 설정합니다.
- HIDDEN ELF 기호 표시를 STV_HIDDEN으로 설정합니다.
- INTERNAL ELF 기호 표시를 STV_INTERNAL로 설정합니다.

type 기호 형식을 지정합니다.

- DATA *symbol*은 소스를 어셈블 및 링크할 때 데이터로 간주됩니다.
- CODE *symbol*은 소스를 어셈블 및 링크할 때 코드로 간주됩니다.
- ELFTYPE=*n* *symbol*은 *n* 값으로 지정되는 특정 ELF 기호로 간주됩니다. 여기서 *n*은 0에서 15 사이의 숫자일 수 있습니다.

지정하지 않는 경우 링커가 가장 적합한 형식을 결정합니다.

사용법

이름은 링크 타임에 별도의 객체 파일에 정의되어 있는 기호로 확인됩니다. 기호는 프로그램 주소로 처리됩니다. [WEAK]를 지정하지 않으면 링크 타임에 해당 기호를 찾을 수 없는 경우 링커에서 오류 메시지를 생성합니다.

[WEAK]를 지정했지만 링크 타임에 해당 기호를 찾을 수 없는 경우 다음과 같이 됩니다.

- 참조가 B 또는 BL 명령어의 대상일 경우 기호 값이 그 다음 명령어의 주소로 지정됩니다. 이 경우 B 또는 BL 명령어를 효과적으로 NOP로 만들 수 있습니다.
- 그렇지 않으면 기호 값이 0으로 지정됩니다.

기호 표시에 대한 자세한 내용은 www.infocenter.arm.com의 *ARM 아키텍처용 ELF* ABI 설명서를 참조하십시오.

예제

이 예제에서는 C++ 라이브러리가 링크되었고 결과에 따라 조건부로 분기되는지 여부를 테스트합니다.

```
AREA    Example, CODE, READONLY
EXTERN  __CPP_INITIALIZE[WEAK]  ; If C++ library linked, gets the address of
                                ; __CPP_INITIALIZE function.
LDR     r0,=__CPP_INITIALIZE    ; If not linked, address is zeroed.
CMP     r0,#0                   ; Test if zero.
BEQ     nocplusplus             ; Branch on the result.
```

7.8.11 INCBIN

INCBIN 지시어는 어셈블 중인 파일에 다른 파일을 포함시킵니다. 파일은 어셈블되지 않고 있는 그대로 포함됩니다.

구문

INCBIN *filename*

인수 설명:

filename 어셈블리에 포함할 파일의 이름입니다. 어셈블러에서는 UNIX 또는 MS-DOS 형식의 파일 이름을 사용할 수 있습니다.

사용법

INCBIN을 사용하여 실행 가능 파일, 리터럴 또는 임의의 데이터를 포함할 수 있습니다. 파일의 내용은 어떤 방식으로든 해석되지 않고 현재 ELF 섹션에 바이트 단위로 추가됩니다. 어셈블리는 INCBIN 지시어 다음에 나오는 행에서 계속됩니다.

기본적으로 어셈블러는 포함된 파일의 현재 위치를 검색합니다. 현재 위치는 호출하는 파일이 있는 디렉토리입니다. -i 어셈블러 명령 행 옵션을 사용하면 검색 경로에 디렉토리를 추가할 수 있습니다. 공백이 포함된 파일 이름과 디렉토리 이름은 큰따옴표로 묶으면 안 됩니다.

예제

```
AREA    Example, CODE, READONLY
INCBIN  file1.dat               ; includes file1 if it
                                ; exists in the
                                ; current place.
INCBIN  c:\project\file2.txt    ; includes file2
```

7.8.12 KEEP

KEEP 지시어는 객체 파일의 기호 테이블에 지역 기호를 유지하도록 어셈블러에 지시합니다.

구문

KEEP {*symbol*}

인수 설명:

symbol 유지할 지역 기호의 이름입니다. *symbol*을 지정하지 않으면 레지스터 기준 기호를 제외한 모든 지역 기호가 유지됩니다.

사용법

기본적으로 어셈블러가 해당 출력 객체 파일을 통해 설명하는 유일한 기호는 다음과 같습니다.

- 내보낸 기호
- 다시 재배치된 기호

KEEP을 사용하여 디버깅에 유용하게 사용할 수 있는 지역 기호를 유지합니다. 유지된 기호는 ARM 디버거와 링커 맵 파일에 표시됩니다.

KEEP은 레지스터 상대 기호를 유지할 수 없습니다 (7-19페이지의 *MAP* 참조).

예제

```
label   ADC     r2,r3,r4
        KEEP    label       ; makes label available to debuggers
        ADD     r2,r2,r5
```

7.8.13 NOFP

NOFP 지시어는 어셈블리 언어 소스 파일에 부동 소수점 명령어가 포함되지 않도록 합니다.

구문

NOFP

사용법

NOFP를 사용하여 소프트웨어나 타겟 하드웨어에서 부동 소수점 명령어를 지원하지 않을 경우 부동 소수점 명령어가 사용되지 않도록 합니다.

NOFP 지시어 다음에 부동 소수점 명령어가 실행되면 알 수 없는 opcode 오류가 생성되고 어셈블리가 실패합니다.

NOFP 지시어가 부동 소수점 명령어 다음에 실행되면 어셈블러에서 다음 오류를 생성합니다.

부동 소수점 명령어를 금지하기에는 늦었습니다.

그런 다음 어셈블리가 실패합니다.

7.8.14 REQUIRE

REQUIRE 지시어는 섹션 간의 종속 관계를 지정합니다.

구문

REQUIRE *label*

인수 설명:

label 필요한 레이블의 이름입니다.

사용법

REQUIRE를 사용하여 관련 섹션이 직접 호출되지 않은 경우에도 포함되도록 합니다. REQUIRE 지시어에 들어 있는 섹션이 링크에 포함되어 있으면 링커에서 지정된 레이블의 정의가 들어 있는 섹션도 포함합니다.

7.8.15 REQUIRE8 및 PRESERVE8

REQUIRE8 지시어는 현재 파일에 스택의 8바이트 정렬이 필요함을 지정하고, 링커에 알림을 제공하기 위해 REQ8 빌드 특성을 설정합니다.

PRESERVE8 지시어는 현재 파일에서 스택의 8바이트 정렬을 유지함을 지정하고, 링커에 알림을 제공하기 위해 PRES8 빌드 특성을 설정합니다.

링커에서는 스택의 8바이트 정렬을 필요로 하는 코드가 직접적으로든 간접적으로든 스택의 8바이트 정렬을 유지하는 코드에 의해서만 호출되는지 검사합니다.

구문

REQUIRE8 {*bool*}

PRESERVE8 {*bool*}

인수 설명:

bool 선택적 부울 상수인 {TRUE} 또는 {FALSE}입니다.

사용법

필요한 경우 코드에서 스택의 8바이트 정렬을 유지하면 PRESERVE8을 사용하여 PRES8 빌드 특성을 파일에 설정합니다. 코드에서 스택의 8바이트 정렬을 유지하지 않으면 PRESERVE8 {FALSE}를 사용하여 PRES8 빌드 특성이 설정되지 않도록 합니다.

참고

PRESERVE8과 PRESERVE8 {FALSE}를 둘 다 생략하면 어셈블러에서 sp를 수정하는 명령어를 검사하여 PRES8 빌드 특성을 설정할지 여부를 결정합니다. PRESERVE8을 명시적으로 지정하는 것이 좋습니다.

다음을 사용하여 경고를 활성화할 수 있습니다.

```
armasm --diag_warning 1546
```

자세한 내용은 3-2페이지의 *명령 구문*을 참조하십시오.

이렇게 하면 다음과 같은 경고가 표시됩니다.

```
"test.s", line 37: Warning: A1546W: Stack pointer update potentially
                                   breaks 8 byte stack alignment
       37 00000044         STMFD    sp!,{r2,r3,lr}
```

예제

```
REQUIRE8
REQUIRE8     {TRUE}      ; equivalent to REQUIRE8
REQUIRE8     {FALSE}     ; equivalent to absence of REQUIRE8
PRESERVE8    {TRUE}      ; equivalent to PRESERVE8
PRESERVE8    {FALSE}     ; NOT exactly equivalent to absence of PRESERVE8
```

7.8.16 ROUT

ROUT 지시어는 지역 레이블 범위의 경계를 표시합니다 (3-32페이지의 *지역 레이블* 참조).

구문

{*name*} ROUT

인수 설명:

name 범위에 지정할 이름입니다.

사용법

ROUT 지시어를 사용하여 지역 레이블 범위를 제한합니다. 이렇게 하면 실수로 잘못된 레이블을 참조하는 문제를 보다 쉽게 방지할 수 있습니다. ROUT 지시어가 없으면 지역 레이블의 범위는 전체 영역이 됩니다 (7-70페이지의 *AREA* 참조).

name 옵션을 사용하여 각 참조가 올바른 지역 레이블에 대한 것인지 확인합니다. 레이블 이름이나 레이블에 대한 참조가 선행 ROUT 지시어와 일치하지 않으면 오류 메시지가 생성되고 어셈블리가 실패합니다.

예제

```
            ; code
routineA    ROUT            ; ROUT is not necessarily a routine
            ; code
3routineA   ; code          ; this label is checked
            ; code
            BEQ     %4routineA   ; this reference is checked
            ; code
            BGE     %3      ; refers to 3 above, but not checked
            ; code
4routineA   ; code          ; this label is checked
            ; code
otherstuff  ROUT            ; start of next scope
```