제3장 ARM 명령어

명령어에 대한 요약

3장은 ARM920T 코어의 ARM 명령어에 대해서 설명한다.

포맷에 대한 요약

ARM 명령어에 대한 포맷은 아래와 같다.

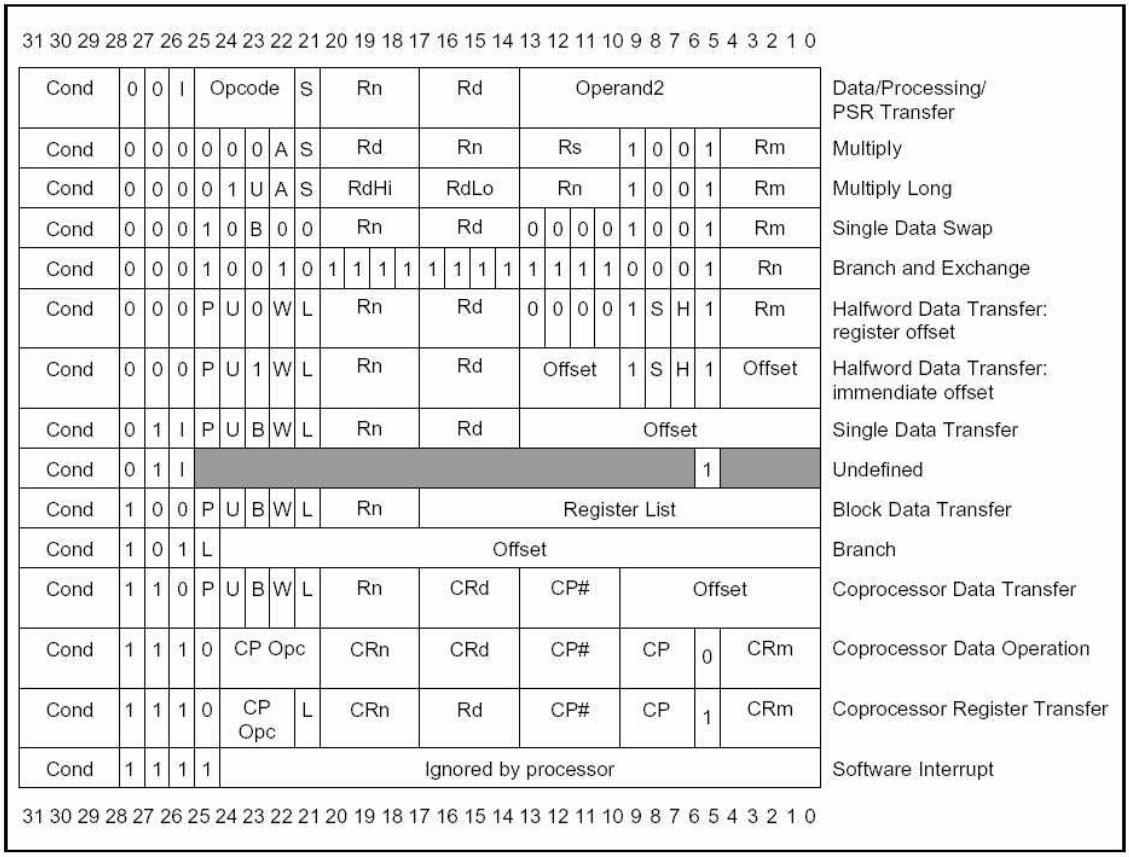

그림 3-1. ARM 명령어 포맷

NOTES

어떤 명령어 코드들은 정의되지 않았지만, 비트 6을 갖는 인스턴스 멀티 명령어가 비트 1로 바뀌는 데 대해서 Undefined 명령어 트랩의 발생 원인은 아니다. 이러한 명령어들은 자신들의 활동이 나중에 ARM 적용 시에 변경될 때에도 사용되지 않는다.

명령어에 대한 요약

표 3-1. ARM 명령어

Mnemonic	Instruction	Action
ADC	Add with carry	Rd: = Rn + Op2 + Carry
ADD	Add	Rd: = Rn + Op2
AND	AND	Rd: = Rn AND Op2
B	Branch	R15: = address
BIC	Bit Clear	Rd: = Rn AND NOT Op2
BL	Branch with Link	R14: = R15, R15: = address
BX	Branch and Exchange	R15: = Rn, T bit: = Rn[0]
CDP	Coprocessor Data Processing	(Coprocessor-specific)
CMN	Compare Negative	CPSR flags: = Rn + Op2
CMP	Compare	CPSR flags: = Rn - Op2
EOR	Exclusive OR	Rd: = (Rn AND NOT Op2) OR (Op2 AND NOT Rn)
LDC	Load coprocessor from memory	Coprocessor load
LDM	Load multiple registers	Stack manipulation (Pop)
LDR	Load register from memory	Rd: = (address)
MCR	Move CPU register to coprocessor register	cRn: = rRn {<op>cRm}
MLA	Multiply Accumulate	Rd: = (Rm × Rs) + Rn
MOV	Move register or constant	Rd: = Op2

Mnemonic	Instruction	Action
MRC	Move from coprocessor register to CPU register	Rn: = cRn {<op>cRm}
MRS	Move PSR status/flags to register	Rn: = PSR
MSR	Move register to PSR status/flags	PSR: = Rm
MUL	Multiply	Rd: = Rm × Rs
MVN	Move negative register	Rd: = 0 × FFFFFFFF EOR Op2
ORR	OR	Rd: = Rn OR Op2
RSB	Reverse Subtract	Rd: = Op2 - Rn
RSC	Reverse Subtract with Carry	Rd: = Op2 - Rn - 1 + Carry
SBC	Subtract with Carry	Rd: = Rn - Op2 - 1 + Carry
STC	Store coprocessor register to memory	address: = CRn
STM	Store Multiple	Stack manipulation (Push)
STR	Store register to memory	<address>: = Rd
SUB	Subtract	Rd: = Rn - Op2
SWI	Software Interrupt	OS call
SWP	Swap register with memory	Rd: = [Rn], [Rn] := Rm
TEQ	Test bitwise equality	CPSR flags: = Rn EOR Op2
TST	Test bits	CPSR flags: = Rn AND Op2

condition field

ARM 상태에서, 모든 명령어는 CPSR condition code와 명령어의 condition field의 상태에 따라서 수행된다. 이 field(bit 31:28)는 명령어가 실행될 환경을 결정한다. C, N, Z과 V 플래그의 상태가 field의 엔코딩된 codition을 fulfil 하며 명령어가 실행되며, 그렇지 않으면 무시

된다.

명령어 mnemonic에 추가될 수 있는 2문자 suffix로 나타나는 16개의 가능한 condition이 있다. 예로, 브랜치(어셈블리 언어의 B)는 Z 플래그가 설정될 때만 발생하는 Branch를 의미하는 “Branch if Equal"이 BEQ가 된다.

실제로, 15개의 다른 condition이 사용 된다: 표 3-2에 목록이 나와 있다. 16번째(1111)은 예약되 있고 사용되지 않는다.

suffix가 없으면, 명령어의 condition field는 “Always"로 설정된다.(suffix AL) 이것은 명령어가 CPSR condition code에 상관없이 항상 실행된다는 것을 의미한다.

표 3-2. Condition Code에 대한 요약

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

BRANCH AND EXCHANGE(BX)

이 명령어는 condition이 true인 경우에만 실행된다. 여러 가지의 condition이 표 2-3에 정의되어 있다.

이 명령어는 일반 레지스터 Rn의 내용을 프로그램 카운터인 PC에 복사해서 브랜치를 실행한다. 이 브랜치는 파이프라인 flush를 발생시키며, Rn에 의한 어드레스에서 refill을 한다. 또한 실행할 명령어를 허가한다. 명령어가 실행되면, Rn[0]의 값은 명령어 stream이 ARM 명령어로 디코딩 될지 혹은 THUMB 명령어로 디코딩 될지를 결정한다.

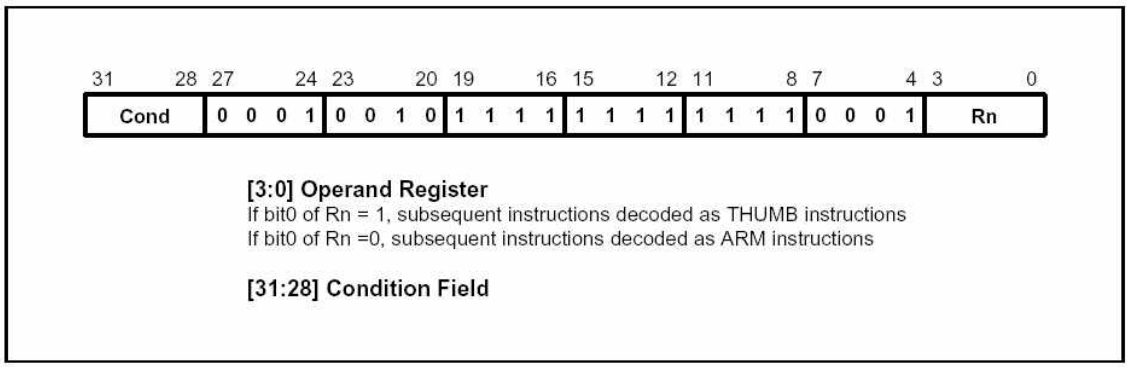

그림 3-2. Branch and Exchange 명령어

명령어의 사이클 타임

BX 명령어가 실행되려면, 2S+1N의 사이클이 필요하며 S와 N은 sequential(S-사이클)과 non-sequential(N-사이클)로 정의된다.

어셈블러 문법

BX - branch and exchange

BX{cond}Rn

{cond} 2개의 문자 condition mnemonic. 표 3-2를 참조

Rn 유효한 레지스터 number를 평가하는 표현

R15를 오퍼랜드로 사용하기

R15를 오퍼랜드로 사용하면, 동작이 정의되지 않는다.

예제

```
        ADR     R0, Into_THUMB + 1      ; 브랜치 타겟 어드레스를 발생
                                        ; 비트0을 high로 설정
                                        ; THUMB 상태에 도달
        BX      R0                      ; 브랜치 하고 THUMB 상태로 변경
        CODE16                          ; 다음에 오는 코드를
        Into_THUMB                      ; THUMB 명령어로 어셈블 함
        •
        •
        •
        ADR     R5, Back_to_ARM         ; 워드 정렬 어드레스에 브랜치 타겟 발생
                                        ; 여기서 비트 0은 low이며, ARM 상태로 복귀함
        BX      R5                      ; 브랜치 하고 ARM 상태로 복귀
        •
        •
```

•

```
ALIGN                          ; 워드 정렬
CODE32                         ; 다음의 코드를 ARM 명령어로 어셈블 함
Back_to_ARM
```

Branch and Branch with Link(B, BL)

이 명령어는 condition이 true인 경우에만 실행된다. 여러 가지의 condition이 표 3-2에 정의되어 있다. 명령어 엔코딩은 그림 3-3에 나타나 있다.

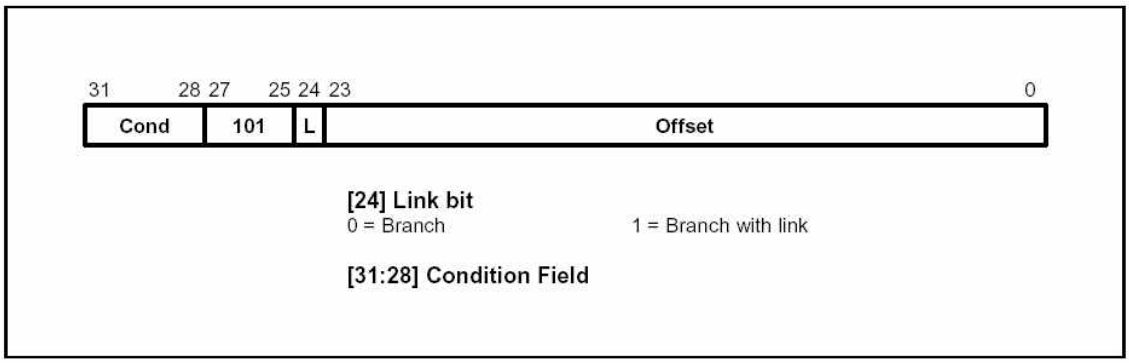

그림 3-3. Branch 명령어

브랜치 명령어는 부호가 있는 2의 보수로 이루어진 24비트 옵셋을 가진다. 왼쪽으로 2비트를 쉬프트하면 부호가 32비트로 확장되며 PC에 더한다. 즉, 명령어는 +/- 32MByte의 브랜치를 정의한다. 브랜치 옵셋은 현재의 명령어 전에 PC가 2워드(8byte)로 되는 prefetch 동작을 취해야 한다.

+/- 32Mbyte 안팎의 브랜치는 이전에 레지스터에 로딩되었던 절대 목적지나 옵셋을 사용해야 한다. 이러한 경우에 Branch with Link 형태의 동작이 필요하면 PC는 수동으로 R14에 저장되어야 한다.

Link 비트

BL은 현재 뱅크의 링크 레지스터(R14)에 이전의 PC 값을 기록한다. R14에 기록되는 PC의 값은 prefetch를 위해서 조정되며, 다음의 branch and link 명령어의 어드레스를 포함한다.

링크 레지스터가 Rn에 의해서 스택 포인터에 저장되고 링크 레지스터가 아직도 유효하거나 LDM Rn!, {..PC}이면 Branch with Link에 의해서 호출되는 루틴에서 복귀하기 위해서 MOV PC, R14를 이용한다.

명령어 사이클 시간

Branch and Branch with Link 명령어는 2S+1N 증가 사이클이 걸리며, 여기서 S와 N은 S-사이클과 I-사이클로 정의된다.

어셈블러 문법

{}의 아이템은 옵션이다. <>의 아이템이 나타나야 한다.

B{L}{cond} <expression>

{L}　　명령어의 Branch with Link를 요청하는데 사용된다. 없으면, R14는 이 명령어에 의해서 영향을 받지 않는다.

{cond}　2개의 문자 mnemonic이 표 3-2에 나타나 있다. 없으면, AL(ALways)가 사용된다.

<expression>　　목적지. 어셈블러는 옵셋을 계산한다.

예제

```
here    BAL     here    ; 0xEAFFFFFE로 어셈블 한다.
        B       there   ; 디폴트로 Always condition이 사용된다.
        CMP     R1, #0  ; R1을 0과 비교하고 fred를 위해서 브랜치 함
                        ; R1이 0이면, 다른 경우를 계속함
        BEQ     fred    ; 다음의 명령어를 계속 실행
        BL      sub+ROM         ; 계산된 어드레스에서 서브루틴을 호출
        ADDS    R1, #1  ; 레지스터1에 1을 더하고, CPSR 플래그를 셋팅
                        ; C 플래그가 클리어되면 서브루틴을 호출
        BLCC    sub     ; R1이 0xFFFFFFFF가 아니면 이 경우가 발생
```

데이터 처리

데이터 처리 명령어는 condition이 true인 경우에만 실행된다. condition은 표 3-2에 정의되어 있다. 명령어 엔코딩은 그림 3-4에 나타나 있다.

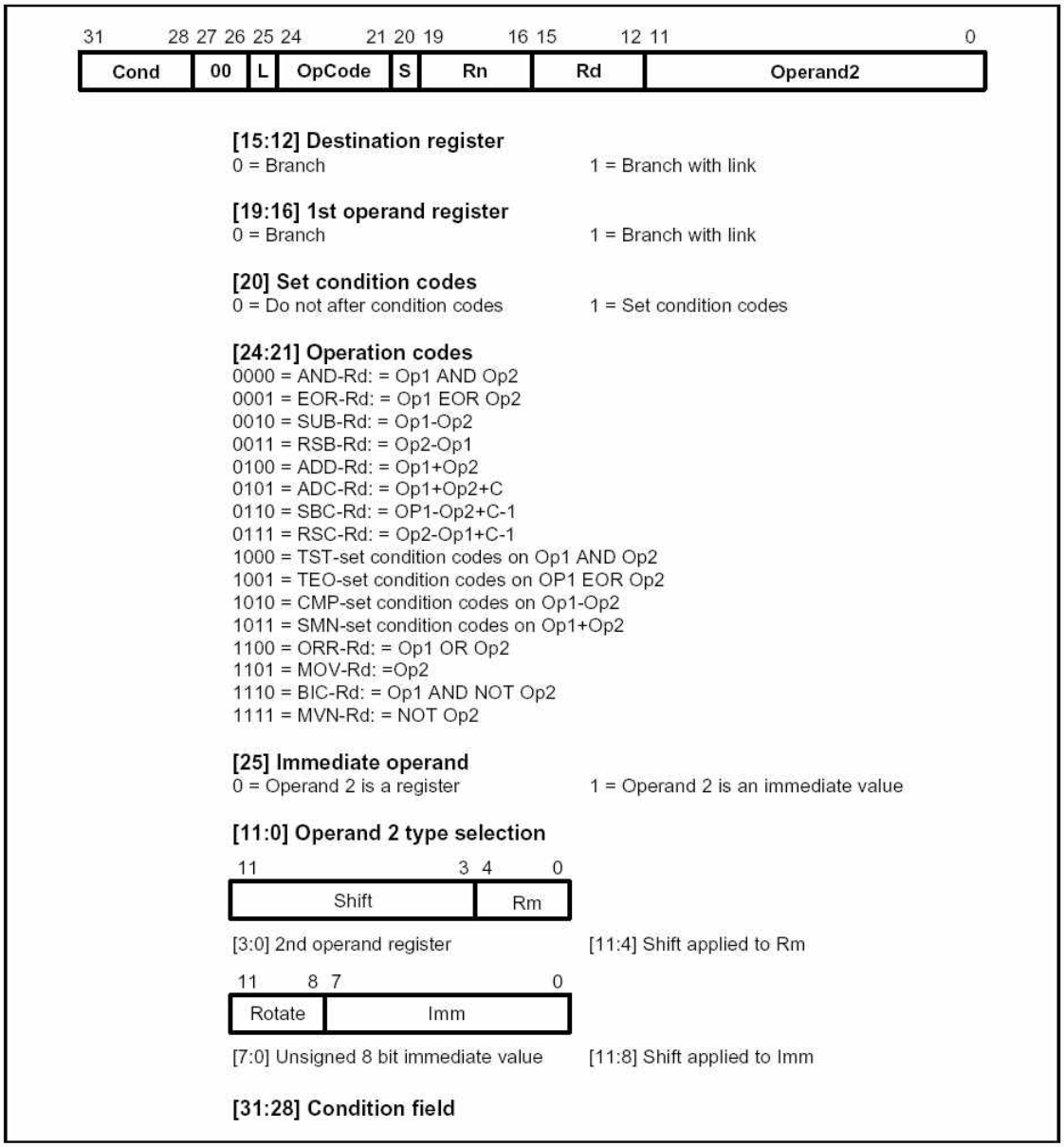

그림 3-4. 데이터 처리 명령어

이 명령어는 1개 혹은 2개의 오퍼랜드를 이용해서 특수한 연산이나 논리 동작을 통해서 수행되는 결과를 산출한다. 첫 번째 오퍼랜드는 항상 레지스터(Rn)이다.

2번째 오퍼랜드는 명령어의 I비트의 값에 따라서 쉬프트 레지스터(Rm)나 회전의 8비트 immediate 값(Imm)이다. CPSR의 condition 코드는 명령어의 S 비트 값에 따라서 명령어의 결과로 업데이트 되거나 보존될 수 있다.

어떤 동작(TST, TEQ, CMP, CMN)은 Rd에 결과를 기록하지 않는다. 테스트를 수행하거나 결과 상의 condition code를 설정하는데 사용되며 항상 S비트를 가진다. 명령어와 효과들은 표 3-3에 나와 있다.

CPSR 플래그

데이터 처리 동작은 논리 혹은 연산으로 분류된다. 논리 동작(ADN, EOR, TST, TEQ, ORR, MOV, BIC, MVN)은 오퍼랜드의 모든 대응되는 비트 상에서 논리 동작을 수행한다. S비트가 CPSR의 V 플래그를 설정하면, C 플래그는 배럴 쉬프터에서 캐리 아웃으로 설정되며, Z 플래그는 결과가 모드 0일 경우에만 설정되며, N 플래그는 비트 31의 논리 값으로 설정된다.

표 3-3. ARM 데이터 처리 명령어

Assembler Mnemonic	OP Code	Action
AND	0000	Operand1 AND operand2
EOR	0001	Operand1 EOR operand2
WUB	0010	Operand1 - operand2
RSB	0011	Operand2 operand1
ADD	0100	Operand1 + operand2
ADC	0101	Operand1 + operand2 + carry
SBC	0110	Operand1 - operand2 + carry - 1
RSC	0111	Operand2 - operand1 + carry - 1
TST	1000	As AND, but result is not written
TEQ	1001	As EOR, but result is not written
CMP	1010	As SUB, but result is not written
CMN	1011	As ADD, but result is not written
ORR	1100	Operand1 OR operand2
MOV	1101	Operand2 (operand1 is ignored)
BIC	1110	Operand1 AND NOT operand2 (Bit clear)
MVN	1111	NOT operand2 (operand1 is ignored)

연산동작(SUB, RSB, ADD, ADC, SBC, RSC, CMP, CMN)은 각 오퍼랜드를 32비트의 integer로 다룬다. S비트가 설정되면 CPSR의 V 플래그도 설정되며 비트 31에 오버플로어가 발생한다; 오퍼랜드가 부호가 없다고 판단되면 무시되지만, 오퍼랜드가 2의 보수로 부호가 있으면 가능한 에러에 대한 경고가 뜬다. C 플래그는 ALU의 비트 31을 캐리 아웃하기 위해서 설정되며, Z 플래그는 결과가 0인 경우에만 설정되며, N 플래그는 sqlxm 31의 값으로 설정된다.

쉬프트

2번째 오퍼랜드가 쉬프트 레지스터에 기록되면, 배럴 쉬프터의 동작이 명령어의 쉬프트 field에 의해서 컨트롤된다. 이 field는 수행될 쉬프트의 형태를 나타낸다. 쉬프트 되어야 하는 레지스터의 수는 명령어의 immediate field에 포함되거나 다른 레지스터의 bottom byte에 포함된다. 여러 쉬프트 형태에 대한 엔코딩은 그림 3-5에 나타나 있다.

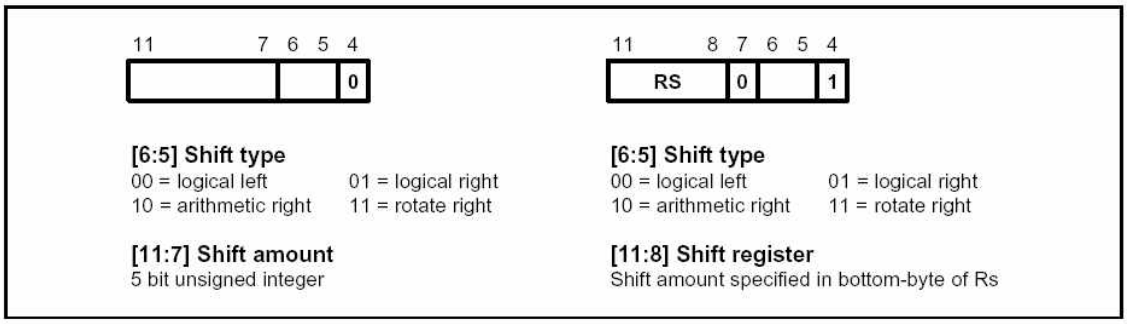

그림 3-5. ARM 쉬프트 동작

명령어 specified 쉬프트 amount

쉬프트 amount가 명령어에 기록되면, 0에서 31까지의 값을 가질 수 있는 5비트의 field를 포함한다. LSL(logical shift left)는 Rmdml 내용을 취하며 좀 더 중요한 위치에 specified amount에 의해서 각 비트를 이동한다. LSB 결과는 0으로 채워지며, ALU 동작이 logical class에서 이루어질 때 CPSR의 C 비트에 래치되는 stnlvmxm 캐리 아웃이 되는 버려지는 LSB를 제외하고, 결과에 맵핑되지 않는 Rm의 high 비트는 버려진다. 예로 LSL #5의 효과는 그림 3-6에 나타나 있다.

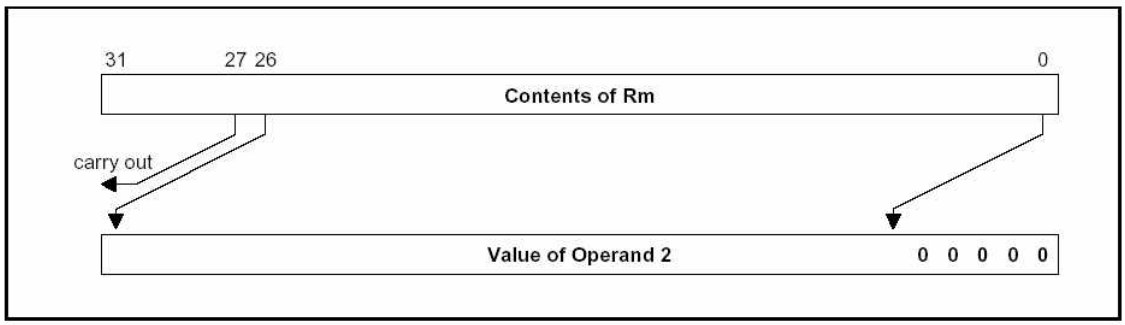

그림 3-6. Logical shift left

NOTES

LSL #0는 쉬프터 캐리 아웃이 CPSR C 플래그의 이전 값인 특별한 경우이다. Rm의 내용은 직접 두 번째 오퍼랜드로 사용된다. LSR은 유사하지만, Rm의 내용은 결과의 LSB 위치로 이동한다. LSR #5는 그림 3-7에 나타나 있다.

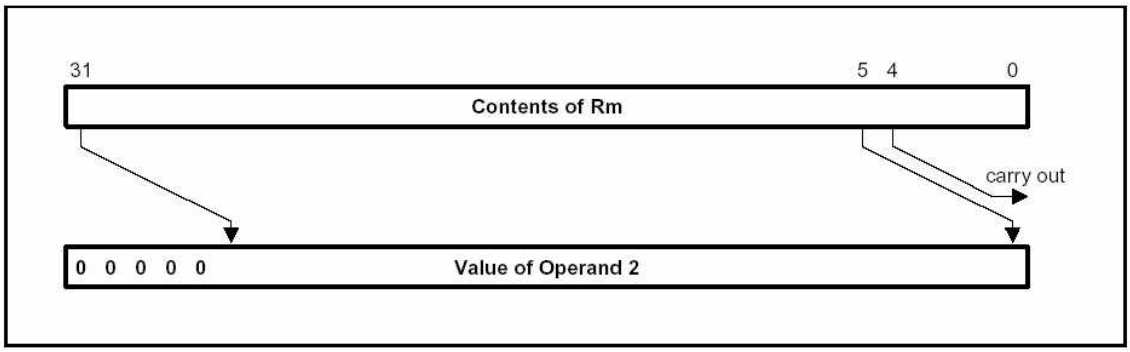

그림 3-7. Logical Shift Right

LSR #0에 대응되는 shift field의 형식은 캐리 아웃되는 Rm의 비트 31이 0인 LSR #32를 엔코딩하는데 사용된다. Logical shift right 0은 logical shift left 0과 같이 redundant하며 어셈블러는 LSR #0을 LSL #0으로 변환하며 LSR #32가 기록되도록 한다.

ASR(Arithmetic Shift Right)은 high 비트가 0 대신에 Rm의 비트 31로 채워지는 logical shift right와 유사하다. 부호가 있는 2의 보수를 보존한다. 예로, ASR #5는 그림 3-8에 나타나 있다.

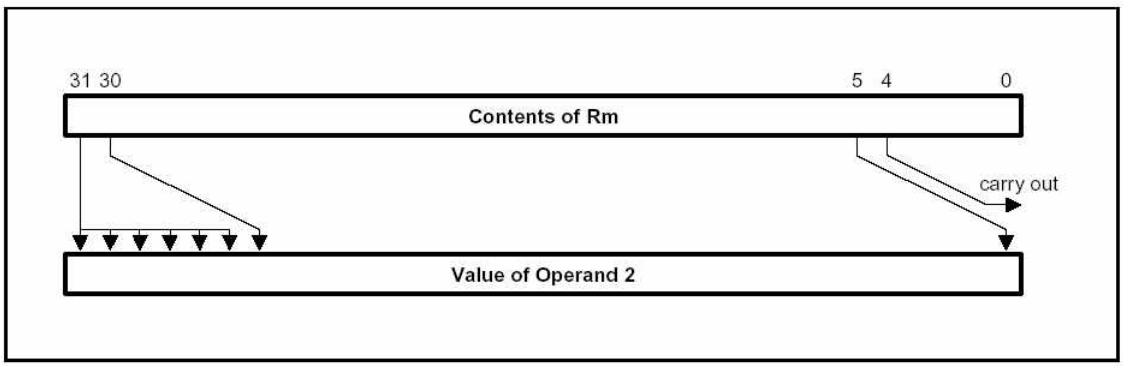

그림 3-8. Arithmetic Shift Right

ASR #0에 제공하는 shift field의 형식은 ASR #32를 엔코딩 하는데 사용된다. Rm의 비트 31은 캐리 아웃하는데 사용되며 오퍼랜드 2의 각 비트는 Rm의 비트 31과 같다. 결과는 Rm의 비트 31의 값에 따라서 모두 1 혹은 모두 0이 된다.

오른쪽 회전(ROR) 동작은 result의 high end에서 logical right 동작의 high end를 채우는데 사용되는 0 대신에 이들을 다시 소개해서 logical shift right 동작에서 오버슈트하는 비트를 재사용한다. 예로 그림 3-9의 ROR #5를 참조하시오.

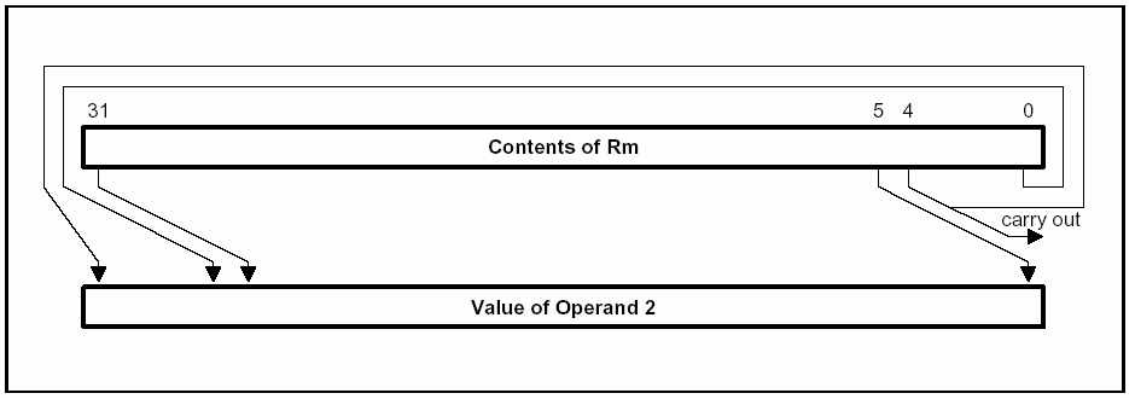

그림 3-9. Rotate Right

ROR #0에 제공되는 shift field의 형식은 배럴 쉬프터, RRX(rotate right extended)의 특별 기능을 엔코딩 하는데 사용된다. 그림 3-10에 Rm의 내용 MSB에 대한 CPSR C 플래그를 append 하는 33비트 중 1비트를 오른쪽으로 회전한다.

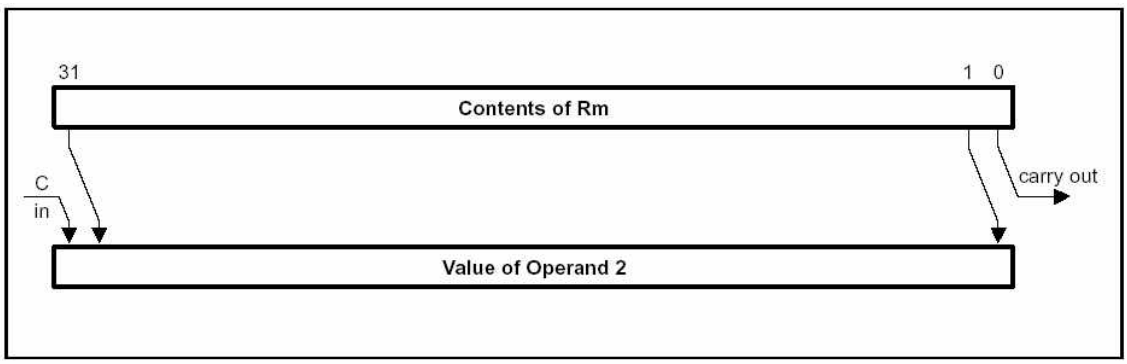

그림 3-10. Rotate Right Extended

Shift amount에 기록되는 레지스터

shift amount를 결정하는데 Rs 내용의 least significant byte가 사용된다. Rs는 R16 이상의 일반 레지스터이다.

이 바이트가 0이면, Rm의 변경되지 않은 값은 2번째 오퍼랜드로 사용되며, CPSR C 플래그의 이전 값이 쉬프트 캐리 아웃으로 통과된다.

바이트의 값이 1에서 31 사이에 있으면, 쉬프트 된 결과는 명령어에 기록된 값과 일치한다.

바이트의 값이 32 이상이면, 결과는 쉬프트의 logical extension 이다:

1. 32의 LSL은 0을 가지며, 캐리 아웃은 Rm의 비트 0과 같다.
2. 32 이상의 LSL은 0을 가지며, 캐리아웃은 0이다.
3. 32의 LSL은 0을 가지며, 캐리 아웃은 Rm의 비트 31과 같다.
4. 32 이상의 LSL은 0을 가지며, 캐리 아웃은 0이다.
5. 32 이상의 ASR은 채워지며, 캐리 아웃은 Rm의 비트 31과 같다.
6. 32의 ROR은 Rm과 같으며, 캐리 아웃은 Rm의 비트 31과 같다.
7. 32보다 훨씬 큰 n의 ROR은 같은 결과를 가져오며 캐리아웃은 n-32의 ROR이다; amount

가 1에서 32 사이일 때까지 n에서 32를 반복해서 뺀다.

NOTES

명령어 비트 7이 반드시 0이어야 한다; 이 비트가 1이면 여러개의 명령어가 실행되거나 정의되지 않은 명령어가 실행된다.

Immediate operand rotates

immediate operand rotate filed는 8비트 immediate 값을 쉬프트 하는 4비트의 부호없는 integer이다. 값은 32비트로 확장된 0이며, rotate field의 값을 2배로 해서 오른쪽으로 회전한다. 예로 모든 파워를 2로 하면서 많은 common 상수들을 인에이블 한다.

R15에 기록하기

Rd가 R15 이상의 레지스터 일 때, CPSR의 condition code 플래그는 위에서 설명한 바와 같이 ALU 플래그에서 업데이트 된다.
Rd가 R15이고 이 명령어의 S 플래그가 R15에 놓인 동작 결과를 설정하지 않을 때 CPSR은 영향을 받지 않는다.

Rd가 R15이고 S 플래그가 R15에 놓인 동작 결과를 설정할 때 현재의 모드에 대응되는 SPSR은 CPSR로 이동한다. PC와 CPSR에 자동적으로 저장된 상태가 변경된다. 이러한 명령어 형식은 사용자 모드에서 사용되어서는 안된다.

R15를 오퍼랜드로 사용하기

R15(PC)가 데이터 처리 명령어의 오퍼랜드로 사용되면 레지스터가 직접 사용된다.
PC 값은 명령어의 어드레스가 되며, 명령어 prefetch를 위해서 8이나 12바이트를 더한다. 쉬프트 amount가 명령어에 기록되면, PC는 8바이트 앞서게 된다. 레지스터가 쉬프트 amount를 기록하는데 사용되면, PC는 12바이트 앞으로 나가게 된다.

TEQ, TST, CMP and CMN OPCODE

NOTES

TEQ, TST, CMP와 CMN은 동작 결과를 기록하지는 않지만, CPSR에 플래그를 설정한다. mnemonic에 기록되지 않더라도 이러한 명령어에 대해서 어셈블러는 항상 S 플래그를 설정해야 한다.

ARM920T의 TEQP 활동은 프로세서가 특권 모드에 있고 사용자 모드가 아니면 CPSR에 SPSR_<mode>를 이동한다.

명령어 사이클 시간

데이터 처리 명령어는 아래와 같이 여러 가지의 사이클 증가를 한다:

표 3-4. 증가되는 사이클 타임

Processing Type	Cycles
Normal data processing	1S
Data processing with register specified shift	1S + 1I
Data processing with PC written	2S + 1N
Data processing with register specified shift and PC written	2S + 1N +1I

NOTE : S, N, I는 S-사이클, N-사이클, I-사이클로 정의된다.

어셈블러 문법

- MOV, MVN(single 오퍼랜드 명령어)
  <opcode> {cond} {S} Rd, <Op2>
- CMP, CMN, TEQ, TST(결과를 도출하지 않는 명령어)
  <opcode> {cond} Rn, <Op2>
- AND, EOR, SUB, RSB, ADD, ADC, SBC, RSC, ORR, BIC
  <opcode> {cond} {S} Rd, Rn, <Op2>

여기서:

<Op2> Rm{,<shift>} 혹은 <#expression>

{cond} 2문자 condition mnemonic. 표 3-2를 참조

{S} S가 있으면(CMP, CMN, TEQ, TST에 적용), condition code를 설정

레지스터 number를 평가하는 Rd, Rn 과 Rm 표현

<#expression> 이 표현이 사용되면, 어셈블러는 expression을 매칭하는데 shift immediate 8비트 field를 생성하려고 할 것이다. 불가능하면, 에러를 표시한다.

<shift> <Shiftname> <register> <shiftname> #expression, RRX(extend를 갖는 1비트 오른쪽 회전)

<shiftname>s ASL, LSL, LSR, ASR, ROR. (ASL은 LSL의 동의어이며, 같은 코드를 어셈블한다.)

예제

```
ADDEQ       R2, R4, R5          ; Z 플래그가 R2:R4+R5로 설정되면
TEQS        R4, #3              ; R4가 3과 같은지를 테스트
                                ; (S는 어셈블러가 자동적으로 삽입될 때
                                ; redundant가 된다.)
SUB         R4, R5, R7, LSR R2  ; R2의 bottom byte에 의한 logical right
                                ; shift R7
                                ; R5에서 결과를 빼고, R4에 답변을 함
MOV         PC, R14             ; 서브루틴에서 복귀
MOVS        PC, R14             ; exception에서 복귀하며, SPSR_mode에서
                                ; CPSR을 회복함
```

PSR 전송(MRS, MSR)

이 명령어는 condition이 true일 때 실행된다. 여러 가지의 conditoin이 표 3-2에 정의되어 있다.

MRS와 MSR 명령어는 데이터 처리 동작의 subset에서 형성되며 S 플래그 설정 없이도 TEQ, TST, CMN과 CMP를 사용한다. 엔코딩이 그림 3-11에 나타나 있다.

이러한 명령어는 CPSR과 SPSR 레지스터에 액세스 된다. MRS 명령어는 CPSR이나 SPSR_<mode>의 내용을 일반 레지스터로 이동한다. MSR 명령어는 일반 레지스터의 내용을 CPSR이나 SPSR_<mode>레지스터로 이동한다.

MSR 명령어는 컨트롤 비트를 조작하지 않고도 CPSR이나 SPSR_<mode>의 confition code 비트에 immediate 값이나 레지스터의 내용을 전달한다. 이러한 경우에, 레지스터 내용의 상위 4비트나 32비트의 immediate 값이 관련된 PSR의 상위 4비트에 기록된다.

오퍼랜드 제한

☞ 사용자 모드에서, CPSR의 컨트롤 비트는 변경이 금지되며, CPSR의 오직 condition code 플래그만이 변경될 수 있다. 다른 모드(특권)에서 전체 CPSR은 변경될 수 있다.

☞ CPSR의 T비트의 상태를 소프트웨어를 이용해서는 결코 변경할 수 없다. 이렇게 되면, 프로세서는 예측할 수 없는 상태로 진입한다.

☞ 액세스 되는 SPSR 레지스터는 실행 시간에 모드에 의존한다. 예로, 프로세서가 FIQ 모드에 있을 경우에 SPSR_fiq가 액세스 가능하다.

☞ 소스나 목적지 레지스터로써 R15를 기록해서는 안 된다.

☞ 또한, 이러한 레지스터가 존재하지 않기 때문에 사용자 모드에서 SPSR에 액세스 할 시도를 하지 않는다.

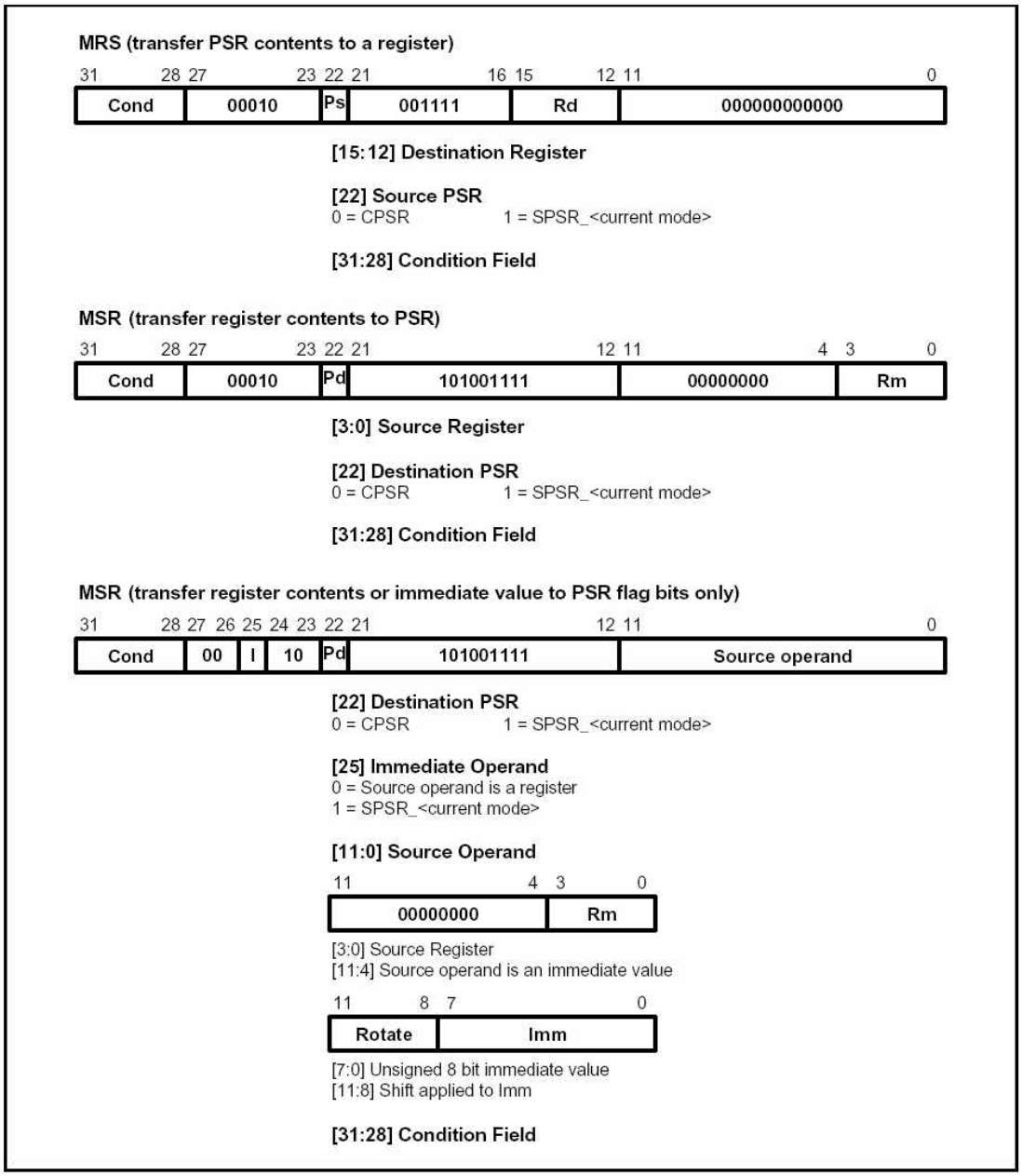

그림 3-11. PSR 전송

예약 비트

PSR의 오직 2비트만이 ARM920T(N, Z, C, V, I, F, T & M[4:0])에 정의된다; 남은 비트는 프로세서의 나중 버전에서 사용하도록 예약되어 있다. PSR 비트의 전체 설명은 그림 2-6을 참조하시오.

ARM920T 프로그램과 나중 버전의 프로세서 사이의 최대 호환성을 유지하기 위해서, 아래와 같은 규칙이 있다:

☞ 예약 비트는 PSR의 값이 변경될 때 보존되어야 한다.

☞ 프로그램은 PSR 상태를 체크할 때 예약 비트로부터의 특수한 값에 의존하지 않는데, 이 유는 나중의 프로세서에서 1이나 0으로 읽힐 수 있기 때문이다.

읽기-변경-쓰기 전략은 PSR 레지스터의 컨트롤 비트를 변경할 때 사용되어야 한다; MRS 명령어를 이용해서 일반 레지스터에 적정한 PSR 레지스터의 전송을 포함하며, 관련된 비트만 변경하고 MSR 명령어를 사용해서 PSR 레지스터의 변경된 값을 전송한다.

예제

아래의 시퀀스는 모드 변경을 수행한다:

```
MRS     R0, CPSR                ; CPSR의 복사본을 취한다.
BIC     R0, R0, #0x1F           ; 모드 비트를 클리어 한다.
ORR     R0, R0, #new_mode       ; 새로운 모드를 선택한다.
MSR     CPSR, R0                ; 변경된 CPSR을 기록한다.
```

PSR의 condition code 플래그를 변경할 때, 값이 컨트롤 비트를 조작하지 않고도 플래그 비트에 직접 기록된다. 아래의 명령어는 N, Z, C와 V 플래그를 설정한다:

```
MSR     CPSR_flg, 0xF0000000    ; 이전의 상태에 상관없이 모든 플래그를 선택
                                ; (다른 컨트롤 비트에 영향을 미치지 않는다.)
```

예약 비트를 보존할 수 없기 때문에 whole PSR에 8비트 immediate 값을 기록해서는 안된다.

명령어 사이클 시간

PSR은 1S의 사이클을 증가시키며, 여기서 S는 S-사이클로 정의된다.

어셈블리 문법

☞ MRS - PSR의 내용을 레지스터에 전송
MRS{cond} Rd, <psr>

☞ MSR - 레지스터의 내용을 PSR에 전송
MSR{cond} <psr>, Rm

☞ MSR - 레지스터의 내용을 PSR 플래그 비트에만 전송
MSR{cond} <psrf>, Rm

레지스터 내용중 4개의 MSB는 각각 N, Z, C & V 플래그에 기록된다.

☞ MSR - immediate 값을 PSR 플래그 비트에만 전송
MSR {cond} <psrf>, <#expression>

expression은 4개의 MSB가 N, Z, C와 V 플래그에 기록되는 32비트 값이어야 한다.

Key :

{cond}	2문자의 condition mnemonic. 표 3-2를 참조
Rd 와 Rm	R15 이상에 레지스터의 number를 평가하는 expression
<psr>	CPSR, CPSR_all, SPSR 혹은 SPSR_all (CPSR과 CPSR_all은 SPSR과 SPSR_all과 같다.)

<psrf> CPSR_flg나 SPSR_flg

<#expression> 어셈블러가 expression과 매칭을 위해서 쉬프트 immedate 8비트 field를 발생시킨다. 사용이 불가능하면, 에러를 발생한다.

예제

사용자 모드에서 명령어는 아래와 같은 실행을 한다:

```
MSR   CPSR_all, Rm            ; CPSR[31:28] <- Rm[31:28]
MSR   CPSR_flg, Rm            ; CPSR[31:28] <- Rm[31:28]
MSR   CPSR_flg, #0xA0000000   ; CPSR[31:28] <- 0xA(N, C 설정;Z, V 클리어)
MRS   Rd, CPSR                ; Rd[31:0] <- CPSR[31:0]
```

특권 모드에서, 명령어는 아래와 같은 실행을 한다:

```
MSR   CPSR_all, Rm            ; CPSR[31:0] <- Rm[31:0]
MSR   CPSR_flg, Rm            ; CPSR[31:28] <- Rm[31:28]
MSR   CPSR_flg, #0x50000000   ; CPSR[31:28] <- 0x5(Z, V 설정;N, C 클리어)
MSR   SPSR_all, Rm            ; SPSR_<mode>[31:0] <- Rm[31:0]
MSR   SPSR_flg, Rm            ; SPSR_<mode>[31:28] <- Rm[31:28]
MSR   SPSR_flg, #0xC0000000   ; SPSR_<mode>[31:28] <- 0xC(N,Z 설정;C,V 클리
      어)
MRS   Rd, SPSR                ; Rd[31:0] <- SPSR_<mode>[31:0]
```

Multiply and Multiply-Accumulate(MUL, MLA)

condition이 true인 경우에만 실행된다. 다양한 condition이 표 3-2에 정의되어 있다. 명령어 엔코딩은 그림 3-12에 나타나 있다.

multiply and multiply-accumulate 명령어는 integer multiplication 수행을 위해서 8비트의 Booth 알고리즘을 사용한다.

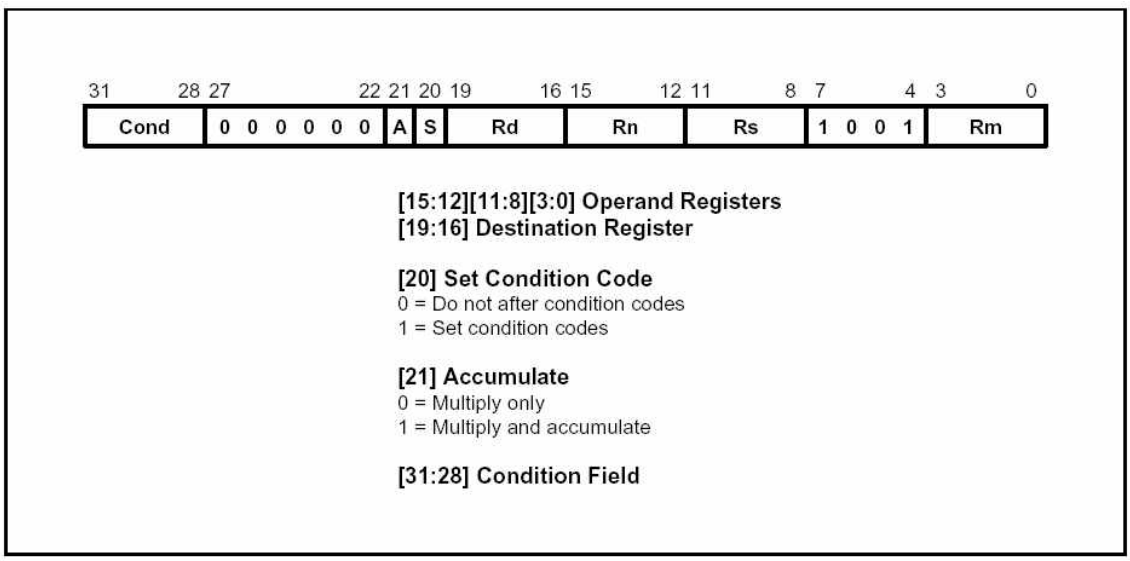

그림 3-12. Multiply 명령어

명령어의 곱셈 형식은 Rd:=Rm*Rs이다. Rn이 무시되면, 명령어를 나중에 업그레이드 할 시에 호환성을 위해서 0으로 설정해야 한다. 곱셈의 덧셈은 Rd:=Rm*Rs+Rn이며, 어떠한 환경에서도 ADD 명령어를 저장 할 수 있다. 오퍼랜드 상에서 동작하는 명령어는 부호 있는 정수나 부호없는 정수형이 될 수 있다.

32비트의 오퍼랜드를 부호가 있을 경우와 없을 경우에 곱셈을 한 결과는 상위 32비트에서만 차이가 있다. - 하위 32비트의 결과는 동일하다. 이러한 명령어는 하위 32비트의 곱셈을 하며, 부호있는 경우와 부호 없는 경우의 곱셈에 이용된다.

아래의 예제는 오퍼랜드의 곱셈을 나타낸다:

오퍼랜드 A	오퍼랜드 B	결과
0xFFFFFFF6	0x00000001	0xFFFFFF38

오퍼랜드가 부호가 있을 경우

오퍼랜드 A는 -10의 값을 갖고, 오퍼랜드 B는 20을 가지면, 결과는 0xFFFFFF38을 나타내는 -200의 값이다.

오퍼랜드가 부호가 없을 경우

오퍼랜드 A가 4294967286이고, 오퍼랜드 B가 20이면, 결과는 0x13FFFFFF38을 나타내는 85899345720이며, 32비트로 하면 0xFFFFFF38이다.

오퍼랜드에 대한 제한

목적지 레지스터 Rd는 오퍼랜드 레지스터 Rm과 같을 필요는 없다. R15는 오퍼랜드나 목적지 레지스터로 사용되어서는 안된다.

모든 다른 레지스터들의 결합은 정확한 값이 되며, Rd, Rn, Rs는 필요 시에 같은 레지스터를 이용한다.

CPSR 플래그

CPSR 플래그의 셋팅은 옵션이며, 명령어의 S비트에 의해서 컨트롤 된다. N과 Z 플래그는 정확한 결과로 셋팅된다. C 플래그는 의미 없는 값이며 V 플래그는 영향을 미치지 않는다.

명령어 사이클 시간

MUL은 1S + ml을 취하며 MLA 1S+(m+1)I 사이클이 수행되며 여기서 S와 I는 S-사이클과 I-사이클로 정의된다.

m	곱셈에 필요한 8비트 곱셈 배열 사이클의 수 Rs에 의해서 정의되는 곱셈 오퍼랜드의 값으로 컨트롤 됨. 가능한 값은 아래와 같다.
1	곱셈 오퍼랜드의 비트[32:8]가 모두 0이거나 모두 1일 때
2	곱셈 오퍼랜드의 비트[32:16]가 모두 0이거나 모두 1일 때
3	곱셈 오퍼랜드의 비트[32:24]가 모두 0이거나 모두 1일 때
4	모든 다른 경우

어셈블러 문법

MUL{cond}{S} Rd, Rm, Rs
MLA{cond}{S} Rd, Rm, Rs, Rn

{cond} 2문자 condition mnemonic. 표 3-2를 참조
{s} S가 있으면 condition 코드 설정
Rd, Rm, Rs와 Rn expression은 R15 이외의 레지스터 number를 평가

예제

```
MUL     R1, R2, R3          ; R1:R2*R3
MLAEQS  R1, R2, R3, R4      ; R1:R2*R3+R4, condition code 셋팅
```

Multiply Long and Multiply-Accumulate Long(MULL, MLAL)

condition이 true인 경우에만 실행된다. 여러 가지의 condition이 표 3-2에 정의되어 있다. 명령어의 엔코딩은 그림 3-13에 나타나 있다.

multiply long 명령어는 2개의 32비트 오퍼랜드의 integer 곱셈을 수행하며 64비트의 결과를 가져온다. 덧셈은 옵션으로 갖고 부호 있는 곱셈과 부호 없는 곱셈은 4가지의 변경을 가져 온다.

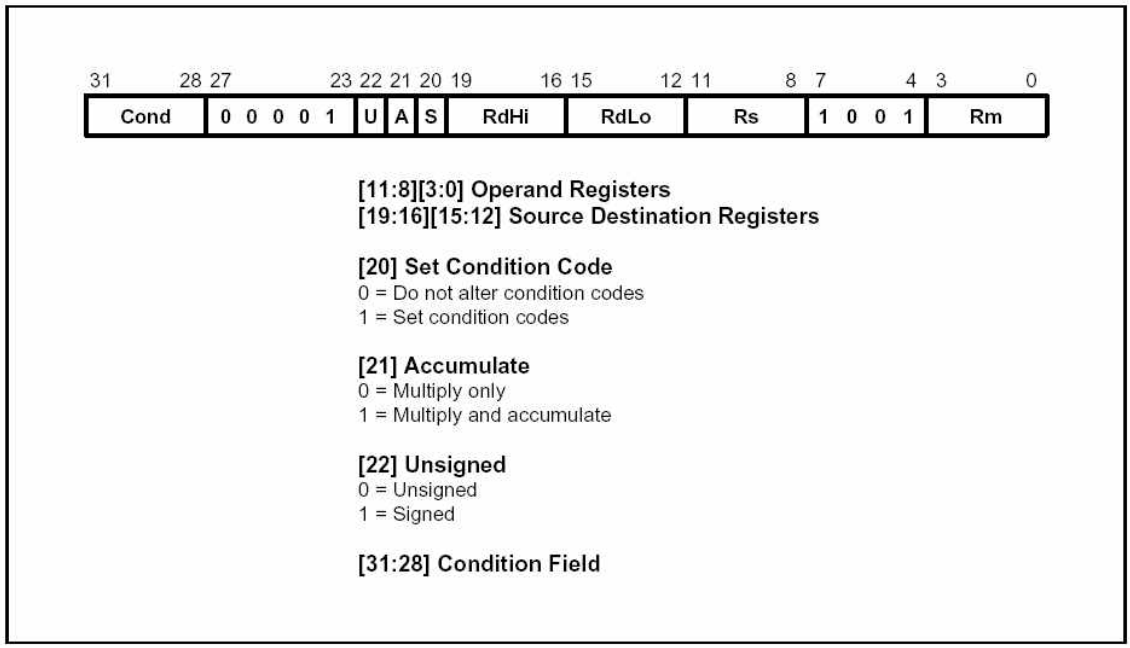

그림 3-13. Multiply Long 명령어

곱셈 형식(UMULL과 SMULL)은 2개의 32비트 number를 가지며, RdHi, RdLo:=Rm*Rs라는 64비트의 형식을 가져온다. 64비트 중 하위 32비트는 RdLo에 기록되며, 상위 32비트는 RdHi에 기록된다.

곱셈-덧셈 형식(UMLAL과 SMLAL)은 2개의 32비트 number를 취하며, 이들을 곱하고 나서 RdHi, RdLo:=Rm*Rs+RdHi, RdLo의 형식에 64비트의 결과를 가져오도록 64비트를 더한다. 더하기 위해서 64비트 중 하위 32비트는 RdLo에서 읽는다. 64비트의 결과 중 하위 32비트는

RdLo에 기록된다. 64비트 중 상위 32비트는 RdHi에 기록된다.

UMULL과 UMLAL 명령어는 자신의 모든 오퍼랜드를 부호없는 바이러니 number로 다루며 부호없는 64비트의 결과에 기록한다. SMULL과 SMLAL 명령어는 자신의 오퍼랜드를 부호 있는 2의 보수로 다루며 부호 있는 64비트의 2의 보수를 기록한다.

오퍼랜드 제한

☞ R15는 오퍼랜드나 목적지 레지스터로 사용되어서는 안된다.
☞ RdHi, RdLo, Rm은 모든 다른 레지스터를 기록해야 한다.

CPSR 플래그

CPSR 셋팅은 옵션이며, 명령어의 S비트에 의해서 결정된다. N과 Z 플래그는 결과를 정확하게 설정한다. C와 V 플래그의 설정은 의미가 없다.

명령어 사이클 시간

MULL은 1S+(m+1)I를 취하며 MLAL 1S+(m+2)I 사이클이 수행되는데, 여기서 m은 Rs로 표시되는 곱셈기 오퍼랜드의 값에 의해서 결정되는 곱셈에 필요한 8비트의 곱셈기 배열 사이클의 수이다.
아래와 같은 값들이 가능하다:

부호 있는 명령어 SMULL, SMLAL에 대해서:

☞ 곱셈기 오퍼랜드의 [31:8] 비트가 모두 0이거나 모두 1일 경우
☞ 곱셈기 오퍼랜드의 [31:16] 비트가 모두 0이거나 모두 1일 경우
☞ 곱셈기 오퍼랜드의 [31:24] 비트가 모두 0이거나 모두 1일 경우
☞ 모든 다른 경우

부호 없는 명령어 UMULL, UMLAL에 대해서:

☞ 곱셈기 오퍼랜드의 [31:8] 비트가 모두 0이거나 모두 1일 경우
☞ 곱셈기 오퍼랜드의 [31:16] 비트가 모두 0이거나 모두 1일 경우
☞ 곱셈기 오퍼랜드의 [31:24] 비트가 모두 0일 경우
☞ 모든 다른 경우

S와 I는 각각 S-사이클과 I-사이클을 정의한다.

어셈블러 문법

표 3-5 어셈블러 Syntax 설명

Mnemonic	Description	Purpose
UMULL{cond}{S} RdLo,RdHi,Rm,Rs	Unsigned Multiply Long	32 x 32 = 64
UMLAL{cond}{S} RdLo,RdHi,Rm,Rs	Unsigned Multiply & Accumulate Long	32 x 32 + 64 = 64
SMULL{cond}{S} RdLo,RdHi,Rm,Rs	Signed Multiply Long	32 x 32 = 64
SMLAL{cond}{S} RdLo,RdHi,Rm,Rs	Signed Multiply & Accumulate Long	32 x 32 + 64 = 64

여기서:

{cond}	2문자 condition mnenomic. 표 3-2를 참조
{S}	S가 있으면 condition code를 설정
RdLo, RdHi, Rm, Rs	R15 이외의 레지스터를 평가하는 expression

예제

```
        UMULL       R1, R4, R2, R3   ; R4, R1:=R2*R3
        UMLALS      R1, R5, R2, R3   ; R5, R1:=R2*R3+R5,
                                     ; R1은 condition code를 셋팅
```

single 데이터 전송(LDR, STR)

condition이 true이면 실행된다. 여러 가지의 condition이 표 3-2에 정의되어 있다. 명령어 엔코딩은 그림 3-14에 나타나 있다.

single 데이터 전송 명령어는 single 바이트아 워드의 데이터를 로딩하거나 저장할 때 사용한다. 전송에 사용되는 메모리 어드레스는 base 레지스터에 옵셋을 더하거나 빼서 계산한다.

계산 결과는 auto-indexing이 필요할 경우에 base로 재 기록 된다.

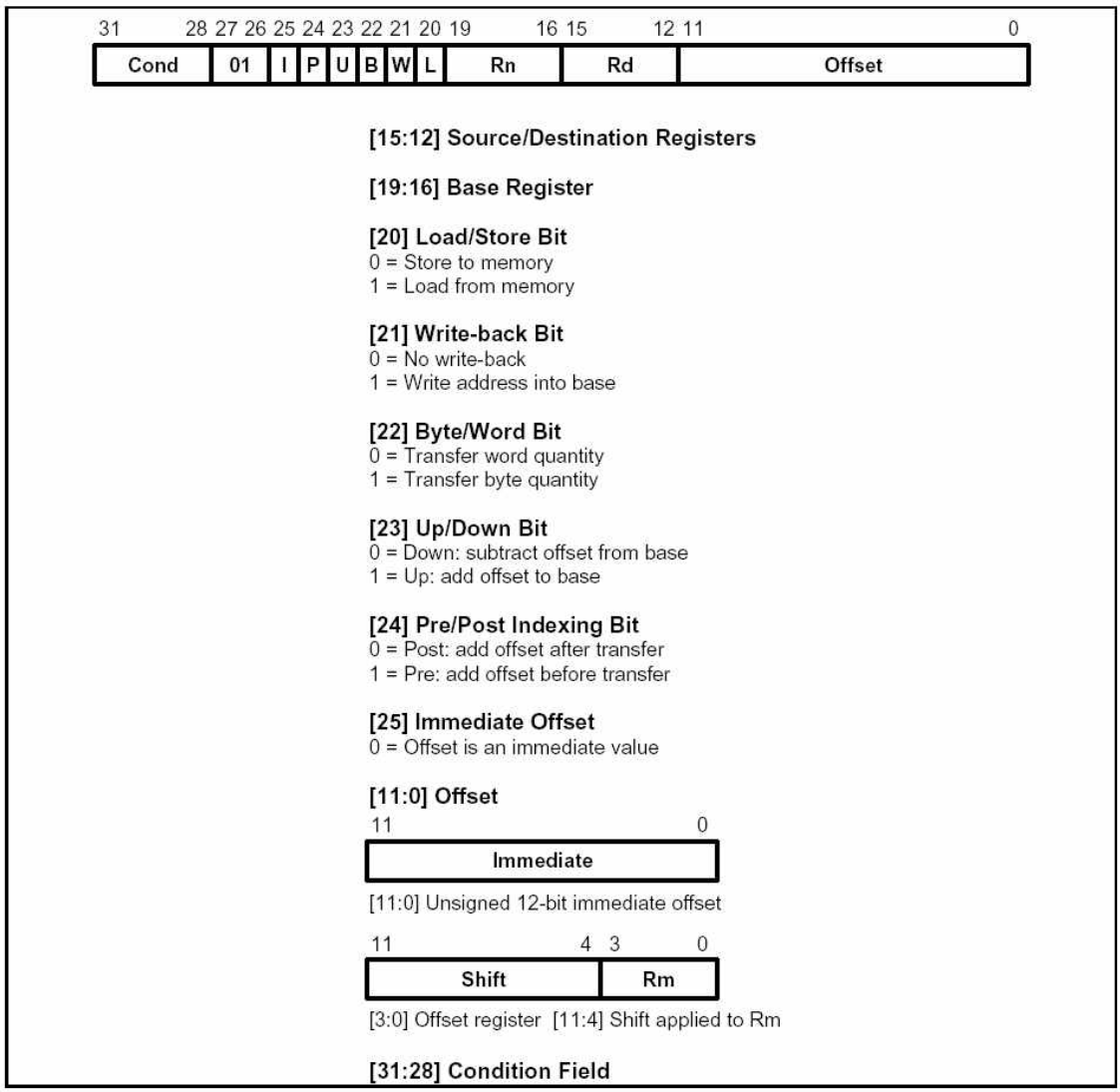

그림 3-14. single 데이터 전송 명령어

옵셋과 auto-indexing

base의 옵셋은 명령어 혹은 2번째 레지스터의 12비트 부호없는 바이너리 immediate 값이 될 수 있다. 옵셋은 base 레지스터 Rn(u=0)에서 빼거나 (U=1)에 더해질 수 있다. 옵셋 변경은 base가 전송 어드레스로 사용되기 전이나 후에 수행될 수 있다.

W비트는 옵션이며 어드레스 모드를 증가시키거나 감소시킬 수 있다. 변경된 base 값은 base(W=1)에 기록되거나 이전의 base 값을 유지(W=0)할 수 도 있다. post-indexed 어드레싱의 경우에, 쓰기 비트는 redundant이며 항상 0으로 설정되는데 이유는 이전의 base 값이 옵셋을 0으로 셋팅해서 얻기 때문이다. 그러므로, post-indexed 데이터 전송은 항상 변경된 base를 기록한다. post-indexed 데이터 전송에서 W비트는 특권 모드에서 사용되며, W비트의 셋팅은 전송시에는 비특권 모드에서 이루어진다. 여기서 OS는 메모리 관리 하드웨어가 유용한 시스템의 사용자 어드레스를 만든다.

쉬프트 레지스터 옵셋

8비트의 컨트롤 비트는 데이터 처리 명령어 부분에 설명된다. 어쨌든, shift amount에 기록된 레지스터는 이러한 명령어 부류에서 유용하지 않다. 그림 3-5를 참조하시오.

바이트와 워드

이 명령어 부류는 ARM920T 레지스터와 메모리 사이에 바이트나(B=1) 워드(B=0)를 전송하는데 사용된다. LDR(B)와 STR(B) 명령어는 ARM920T 코어의 BIGEND 컨트롤 신호의 영향을 받는다. 2개의 가능한 설정이 아래에서 설명되고 있다.

리틀-엔디안 설정

어드레스가 워드 경계상에 있으면 바이트 호출(LDRB)은 데이터 버스 입력 7에서 0까지이며, 워드 어드레스에 1바이트를 더하면 데이터 버스 입력 15에서 8까지 이다. 선택된 바이트는 목적지 레지스터의 하위 8비트에 놓이게 되며, 남은 레지스터의 비트는 0으로 채워진다. 그림 2-2를 참조하시오.

바이트 저장(STRB)은 데이터 버스 출력 31에서 0까지 4 타임 동안에 소스 레지스터의 하위 8비트를 반복한다. 외부의 메모리 시스템은 데이터를 저장하기 위해서 적정한 바이트 서브시스템으로 동작해야 한다.

워드 호출(LDR)은 워드 정렬 어드레스를 사용한다. 어쨌든, 워드 경계로부터의 어드레스 옵셋은 어드레스 바이트가 비트0에서 7까지 복사하기 위해서 데이터가 레지스터에 회전되도록 한다. 즉, 워드 경계에서 옵셋0과 2에 접근되는 half-word가 레지스터의 비트0에서 15까지 정확하게 호출됨을 의미한다. 2개의 쉬프트 동작은 상위 16비트를 클리어하거나 부호를 확장하는데 필요하다.

워드 저장(STR)은 워드 정렬 어드레스를 발생한다. 어드레스가 워드 정렬이 아니면 데이터 버스에 나타나는 워드는 영향을 받지 않는다. 즉, 저장되는 레지스터의 비트 31은 데이터 버스 출력 31에 나타난다.

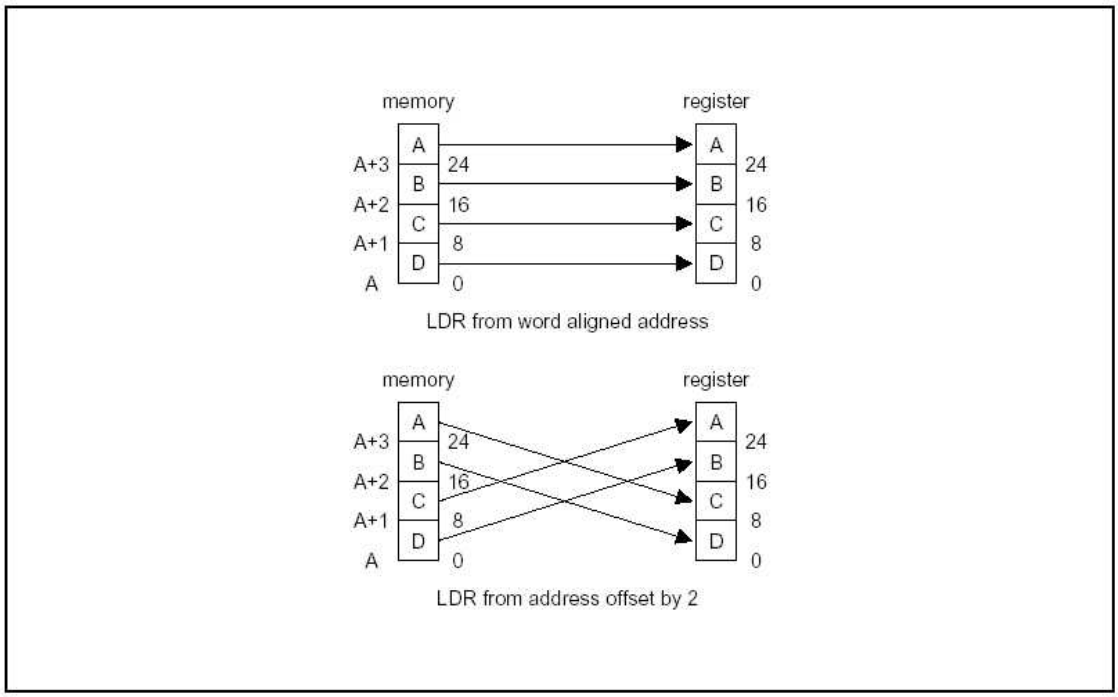

그림 3-15. 리틀-엔디안 옵셋 어드레싱

빅-엔디안 설정

바이트 호출(LDRB)는 어드레스가 워드 경계에 있으면 데이터 버스 입력 31에서 24까지 데이터를 기대할 수 있으며, 워드 어드레스에 1바이트를 더하면 데이터 버스 입력은 23에서 16이 된다. 선택된 바이트는 목적지 레지스터의 하위 8비트에 놓이며, 레지스터의 남은 비트는 0으로 채워진다. 그림 2-1을 참조하시오.

바이트 저장(STRB)은 데이터 버스 출력 31에서 0까지 4타임 동안 소스 레지스터의 하위 8비트를 반복한다. 외부의 메모리 시스템은 데이터를 저장하기 위해서 적정한 바이트 서브시스템에서 동작해야 한다.

워드 호출(LDR)은 워드 정렬 어드레스를 발생해야 한다. 워드 경계로부터의 어드레스 옵셋0이나 2는 어드레스 바이트를 31비트에서 24비트까지 복사하기 위해서 데이터를 레지스터에 회전하도록 한다. 이러한 옵셋에 액세스 되는 half-word가 레지스터의 비트16에서 31까지 정확하게 호출된다는 것을 의미한다. 쉬프트 동작은 하위 16비트에 데이터를 옮기는데 필요하다. 경계로부터의 어드레스 옵셋1이나 3은 어드레스 바이트가 비트15에서 8까지 복사하기 위해서 레지스터에 데이터를 회전하도록 한다.

워드 저장(STR)은 워드 정렬 어드레스를 발생해야 한다. 데이터 버스에 나타나는 워드는 어드레스가 워드 정렬이 아니면 영향을 받지 않는다. 즉, 저장되는 레지스터의 비트 31은 데이터 버스 출력 31에 항상 나타난다.

R15의 사용

R15가 base 레지스터(Rn)으로 정의되면, Write-back은 기록되지 않아야 한다. R15를 base 레지스터로 사용하면, 현재의 명령어 어드레스에서 어드레스 8바이트를 포함해야 함을 기억해야 한다.

R15는 레지스터 옵셋(Rm)으로 정의되지 않아야 한다.

R15가 레지스터 저장(STR) 명령어의 소스 레지스터(Rd)이면, 저장되는 값은 명령어에 2를 더한 어드레스가 된다.

base 레지스터의 사용에 대한 제한

late abort로 설정될 때, 아래의 예제 코드는 abort 핸들러가 시작하기 전에 base 레지스터 Rn으로 풀기가 어렵다. 때때로, 초기 값을 계산하는 것이 불가능하다.

예제:

```
LDR     R0, [R1], R1
```

그러므로 Rm이 Rn과 같은 레지스터인 post-indexed LDR이나 STR이 사용되어서는 안된다.

데이터 abort

legal 어드레스의 전송은 메모리 관리 시스템에 문제를 일으킬 수도 있다. 예로, 가상 메모리를 사용하는 시스템에서, 필요한 데이터가 메인 메모리에 없을 수도 있다. 메모리 관리자는 프로세서 ABORT에서 생기는 문제를 위해서 데이터 Abort 트랩이 발생하도록 신호를 High로 만든다. 이러한 시스템 소프트웨어를 이용하면, 문제의 원인 해결이 가능하며, 명령어가 재시작되고 원래의 프로그램이 계속된다.

명령어 사이클 시간

일반의 LDR 명령어는 1S+1N+1I가 걸리며, LDR PC는 2S+2N+1I의 사이클이 증가하는데, 여기서 S, N, I는 각각 S-사이클, N-사이클, I-사이클을 나타낸다. STR 명령어는 2N의 증가 사이클을 갖는다.

어셈블러 문법

<LDR|STR>{cond}{B}{T} Rd, <Address>

여기서:

LDR	메모리에서 레지스터로 호출
STR	레지스터에서 메모리에 저장
{cond}	2문자 condition mnemonic. 표 3-2를 참조하시오.
{B}	B가 바이트 전송을 나타내면, 그 외에는 워드 전송을 나타냄
{T}	T가 post-indexed 명령어에 W비트를 설정하면, 전송 사이클에 대해서 non-priviledged 모드로 전환함. TSms pre-indexed 어드레싱

모드가 기록되거나 사용되었으면 허가를 하지 않음.

Rd 유효한 레지스터 개수를 평가하는 expression.

Rn과 Rm 레지스터의 갯수를 평가하는 expression. Rn이 R15이면, 어셈블러는 옵셋 값에서 ARM920T 파이프라인을 위해서 8을 뺀다. 이러한 경우에, base write-back는 적용되지 않는다.

<address>:

1 어드레스를 발생하는 expression:
어셈블러는 base로 PC를 사용하는 명령어를 발생하려고 하며, 어드레스 위치에 대한 교정된 immediate 옵셋이 expression을 평가해서 주어진다.
PC와 관련이 있으며, pre-indexed 어드레스이다. 어드레스의 범위가 벗어나면, 에러가 발생된다.

2 pre-indexed 어드레싱 표현:

[Rn] 옵셋 0

[Rn, <#expression>]{!} <expression> 바이트 옵셋

[Rn, {+/-} Rm{,<shift>}]{!} <shift>에 의해서 쉬프트 된 인덱스 레지스터의 +/- 옵셋

3. post-indexed 어드레싱 표현:

[Rn], <#expression> <expression> 바이트 옵셋

[Rn], {+/-} Rm{,<shift>} <shift>에 의해서 쉬프트 된 인덱스 레지스터의 +/- 옵셋

<shift> 쉬프트 동작을 발생하지만 레지스터에 의한 shift amount를 기록할 수 없음.

{!} !이 존재하면 base 레지스터를 write back 한다.

예제

```
STR     R1, [R2, R4]!           ; R1을 R2+R4에 저장하고
                                ; R2에 back 어드레스를 기록한다.
STR     R1, [R2], R4            ; R1을 R2에 저장하고 R2+R4를 R2로 back 기록
                                ; 한다.
LDR     R1, [R2, #16]           ; R2+16에서 R1을 호출, write back 하지 않음
LDR     R1, [R2, R3, LSL#2]     ; R2+R3*4에서 R1을 호출
LDREQB R1, [R6, #5]             ; R6+5에서 바이트를 호출해서
                                ; R1 비트0에서 7까지 넣고, 비트8에서 비트31
                                ; 까지를 0으로 채운다.
STR     R1, PLACE               ; 어드레스 PLACE에 PC 관련 옵셋을 발생시킴.
PLACE
```

Halfword와 부호있는 데이터 전송(LDRH/STRH/LDRSB/LDRSH)

condition이 true인 경우에만 실행되는 명령어 이다. 다양한 condition이 표 3-2에 정의되어 있다. 명령어 엔코딩은 그림 3-16에 나타나 있다.

이 명령어는 half-word의 데이터를 호출하거나 저장하는데 사용되며 sign-extended 바이트아 half-word의 데이터를 호출한다. 전송시에 사용되는 메모리 어드레스는 base 레지스터에 옵셋을 더하거나 빼서 계산한다. 이러한 계산 결과는 auto-indexing이 필요할 경우에 base 레지스터에 write back 될 수 있다.

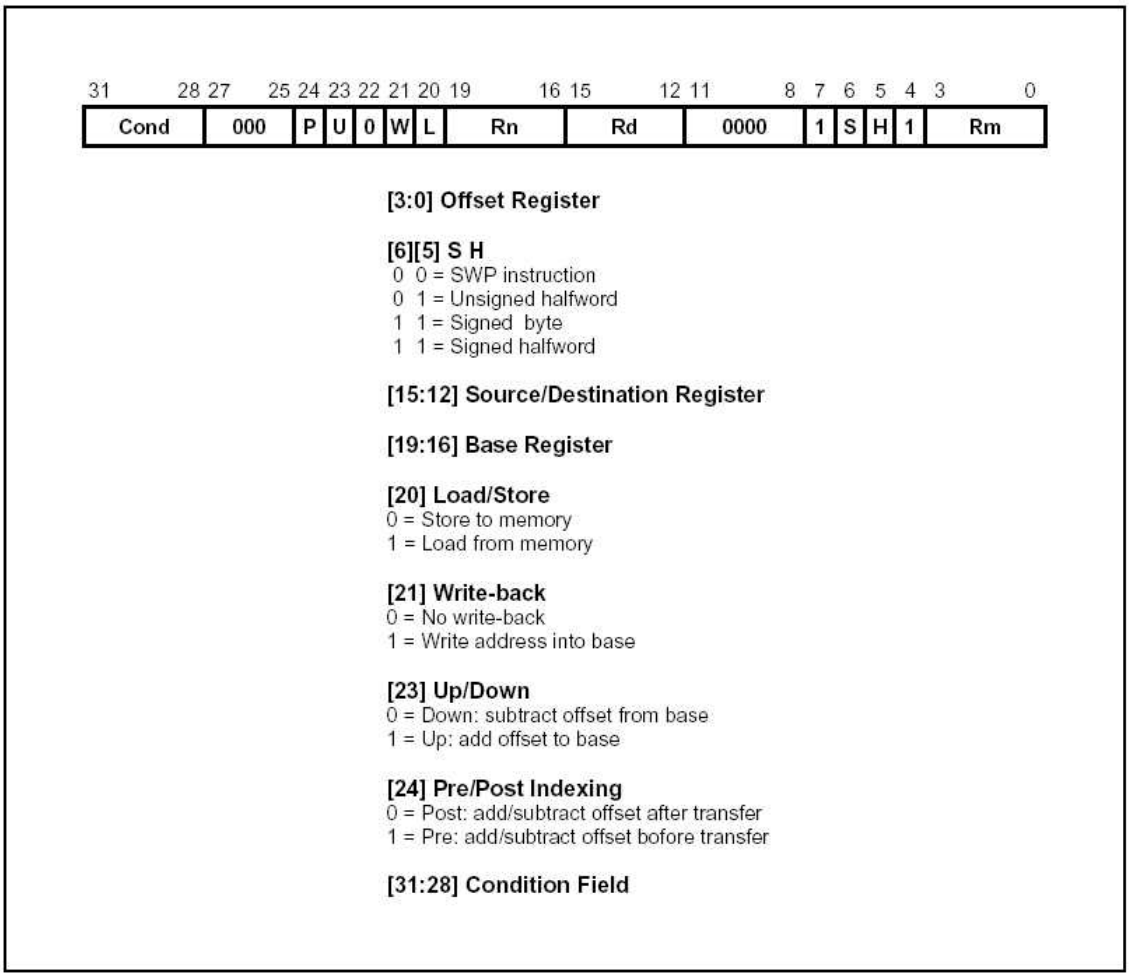

그림 3-16. 레지스터 옵셋을 갖는 halfword와 부호있는 데이터 전송

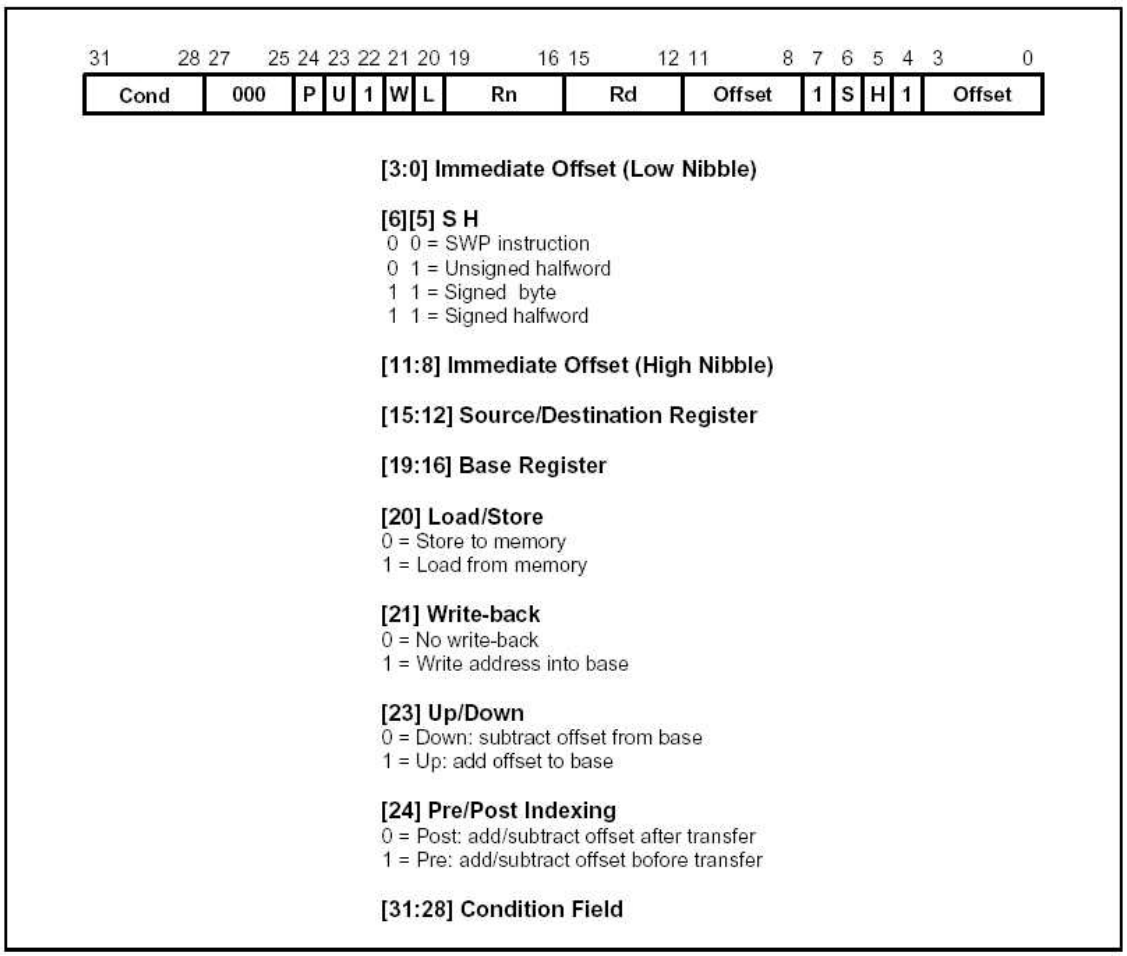

그림 3-17. immediate 옵셋과 auto-indexing을 갖는 halfword와 부호있느 sep이터 전송

옵셋과 auto-indexing

base로 부터의 옵셋은 명령어나 2번째 레지스터에서 8비트의 부호없는 바이너리 immediate 값이 될 수도 있다. 8비트 옵셋은 비트 11이 MSB가 되고 비트 0이 LSB가 되는 워드 명령어의 비트 3에서 0까지와 비트 11에서 8까지를 연결시킨다. 옵셋은 base 레지스터 Rn을 더하거나 (U=1) 뺄 수(U=0) 있다. 옵셋의 변경은 base 레지스터가 전송 어드레스로 사용되기 전(P=1)이나 후(P=0)에 수행될 수 있다.

W비트는 어드레싱 모드를 오토로 증가하거나 감소하는 옵션을 제공한다. 변경된 base 값은 base에 back 기록되거나(W=1), 이전의 base 값을 유지(W=0)할 수도 있다. post-indexed 어드레싱의 경우에, write back 비트는 감소하며 항상 0으로 설정되는데, 이유는 옵셋을 0으로 설정하는 일이 필요하면 이전의 base 값이 얻어지기 때문이다. 그러므로 post-indexed 데이터 전송은 항상 변경된 base를 back write 한다.

write-back 비트는 post-indexed 어드레싱이 선택될 때 high(W=1)로 되지 않는다.

Halfword 호출과 저장

S=0과 H=1로 셋팅하면 ARM920T 레지스터와 메모리 사이에 부호없는 half-word를 전송하는데 사용된다.

LDRH와 STRH 명령어의 사용은 BIGEND 컨트를 신호에 의해서 영향을 받는다. 2개의 가능한 설정이 아래에 설명되어 있다.

부호있는 바이트와 halfword 호출

S비트는 sign-extended 데이터의 호출을 컨트롤 한다. S=1이면 H비트는 바이트(H=0)와 Halfword(H=1) 사이를 선택한다. L비트는 부호있는(S=1) 동작이 선택될 때 low(저장)로 되어서는 안된다.

LDRSB 명령어는 목적지 레지스터의 비트 7에서 0까지 선택된 바이트롤 호출하고 목적지 레지스터의 비트 31에서 8까지는 비트 7의 값으로 설정한다.

LDRSH 명령어는 목적지 레지스터에 선택된 Half-word를 비트15에서 0에 호출하고 목적지 레지스터의 비트 31에서 16의 값은 비트 15의 값으로 설정한다.

LDRSB와 LDRSH 명령어의 활동은 BIGEND 컨트롤 신호에 의해서 영향을 받는다. 2개의 가능한 설정 상태가 아래에 설명되어 있다.

엔디안 설정과 바이트/halfword 선택

리틀-엔디안 설정

부호있는 바이트 호출(LDRSB)은 어드레스가 워드 경계에 있으면 데이터 버스 입력 7에서 0까지 기대하며, 워드 어드레스에 1바이트를 더하면 데이터 버스 입력 15에서 8까지이다. 선택된 바이트는 목적지 레지스터의 하위 8비트에 놓이며 레지스터의 남은 비트는 바이트의 부호있는 비트 7로 채워진다. 그림 2-2를 참조하시오.

Halfword 호출(LDRSH 혹은 LDRH)은 어드레스가 워드 경계에 있으면 데이터 버스 입력 15에서 0까지가 되며 halfword 경계이면 데이터 버스 입력 31에서 16까지 이다. 어드레스는 항상 halfword 경계 이어야 한다. 어드레스의 비트0이 High이면 ARM920T는 예측할 수 없는 값을 호출한다. 선택된 halfword는 목적지 레지스터의 하위 16비트에 놓인다. 부호없느 노밀-word(LDRH)에 대해서, 레지스터의 상위 16비트는 0으로 채워지며, 부호 있는 half-워드(LDRSH)에 대해서 상위 16비트는 부호 있는 halfword의 비트15로 채워진다.

halfword 저장(STRH)은 데이터 버스 출력 31에서 0까지의 2배인 소스 레지스터의 하위 16비트를 반복한다. 외부의 메모리 시스템은 데이터를 저장하기 위해서 적정한 halfword 서브시스템으로 동작해야 한다. 어드레스는 halfword 할당이어야 하며, 어드레스의 비트0이 high이면, 예측할 수 없는 동작이 생긴다.

빅-엔디안 설정

부호있는 바이트 호출(LDRSB)은 어드레스가 워드 경계에 있으면, 데이터 버스 입력31에서 24까지가 되며, 워드 어드레스에 1바이트를 더하면 데이터 버스 입력 23에서 16까지 된다. 선택된 바이트는 목적지 레지스터의 하위 8비트에 놓이며, 레지스터의 남은 비트는 바이트의 부호 있는 비트7로 채워진다. 그림 2-1을 참조하시오.

halfword 호출(LDRSH 혹은 LDRH)은 어드레스가 워드 경계에 있으면 데이터 버스 입력 31에서 16까지이며 halfword 경계이면 데이터 버스 입력 15에서 0까지 이다. 어드레스는 항상 halfword 경계에 있다. 어드레스의 비트 0이 high이면 ARM920T는 예측할 수 없는 값을 호출

한다. 선택된 halfword는 목적지 레지스터의 하위 16비트에 위치한다. 부호없는 half-word(LDRH)에 대해서, 레지스터의 상위 16비트는 0으로 채워지며, 부호있는 half-word(LDRSH)에 대해서 상위 16비트는 halfword의 부호있는 비트15로 채워진다.

halfword 저장(STRH)은 데이터 버스 출력 31에서 0까지의 2배인 소스 레지스터의 하위 16비트를 반복한다. 외부의 메모리 시스템은 데이터를 저장하기 위해서 적정한 halfword 서브시스템으로 동작해야 한다. 어드레스는 halfword 정렬이어야 하며, 어드레스의 비트0이 High이면 예측할 수 없는 동작을 하게 된다.

R15의 사용

R15가 base 레지스터(Rn)으로 정의되지 않으면 write-back는 정의되지 않아야 한다. R15를 base 레지스터로 사용할 때 현재의 명령어 어드레스에서 어드레스 8바이트를 포함한다.
R15는 레지스터 옵셋(Rm)으로 정의되지 않는다.

R15가 half-word 저장(STRH) 명령어의 소스 레지스터(Rd)일 때, 저장된 어드레스는 명령어에 12를 더한 어드레스가 된다.

데이터 abort

legal 어드레스의 전송은 메모리 관리 시스템에 문제를 발생할 수도 있다. 예로, 가사으이 메모리를 사용하는 시스템에서, 필요한 데이터가 메임 메모리에 없을 수도 있다. 메모리 관리자는 프로세서 ABORT에서 생기는 문제를 위해서 데이터 abort 트랩이 발생하느 srht에 입력 high를 준다. 시스템 소프트웨어를 이용해서 문제의 원인을 해결하면, 명령어가 다시 시작되고 원래에 실행된 프로그램이 계속된다.

명령어 사이클 시간

Normal LDR(H, SH, SB) 명령어는 1S+1N+1I를 필요로 한다. LDR(H, SH, SB) PC는 2S+2N+1I의 증가 사이클을 취한다. S, N, I는 S-사이클, N-사이클, I-사이클이다. STRH 명령어는 2N의 증가 사이클을 필요로 한다.

어셈블러 문법

<LDR|STR>{cond}<H|SH|SB> Rd, <address>

LDR	메모리에서 레지스터로 호출
STR	레지스터에서 메모리에 저장
{cond}	2문자 condition mnemonic. 표 3-2를 참조하시오.
H	halfword quantity 전송
SB	부호있는 extended 바이트를 호출(오직 LDR에서만 유효함)
SH	부호있는 extended halfword를 호출(오직 LDR에서만 유효함)
Rd	유효한 레지스터 개수를 평가하는 expression

<address>

1	어드레스를 발생시키는 expression: 어셈블러는 PC를 base로 이용하는 명령러를 발생시키며 어드레스 위치에

교정된 immediate 옵셋이 expression을 평가하는데 주어진다. PC와 관련이 있으며, pre-indexed 어드레스 이다. 어드레스의 범위가 벗어나면, 에러가 발생한다.

2 pre-indexed 어드레싱 표현:

[Rn] 옵셋 0

[Rn, <#expression>]{!} <expression> 바이트의 옵셋

[Rn, {+/-}Rm]{!} 인덱스 레지스터의 +/- 옵셋

3 post-indexed 어드레싱 표현:

[Rn], <#expression> <expression> 바이트의 옵셋

[Rn], {+/-}Rm 인덱스 레지스터의 +/- 옵셋

4 Rn과 Rm은 레지스터의 개수를 평가하는 expression이다. Rn이 R15이면 어셈블러는 ARM920T 파이프 라인에 대한 옵셋 값에서 8을 뺀다. 이러한 경우에 base write-back는 적용되지 않는다.

{!} !가 존재하면 base 레지스터를 write back 함(W비트를 설정)

예제

```
LDRH        R1, [R2-R3]!          ; R2-R3에 포함된 halfword 어드레스의
                                  ; 내용에서 R1을 호출하고
                                  ; R2에 back 어드레스를 기록.
STRH        R3, [R4, #14]         ; R14+14에 R3의 halfword를 저장
                                  ; back write 하지 않음
LDRSB       R8, [R2], #-223       ; R2에 포함된 바이트 어드레스의 부호
                                  ; 있는 확장 내용으로 R8을 호출
LDRNESH     R11, [R0]             ; R0에 포함된 halfword 어드레스의
                                  ; 부호있는 확장 내용으로 R11을 호출
HERE                              ; 어드레스 FRED에 대한 PC 관련 옵셋
                                  ; 을 발생
STRH        R5, [PC, #(FRED-HERE-8)]   ; 어드레스 FRED에 R5 halfword
                                       ; 를 저장
FRED
```

블록 데이터 전송(LDM, STM)

conditon이 true일 경우에 명령어가 실행된다. 다양한 condition이 표3-2에 정의되어 있다. 명령어 엔코딩은 그림 3-18에 나타나 있다.

블록 데이터 전송 명령어는 현재 나타나 레지스터의 임의의 서브셋을 호출(LDM)하거나 저장(STM)하는데 사용된다. 모든 사용기 가능한 스택 모드를 지원하며 메모리를 업하거나 다운하는 full 혹은 empty 스택을 관리하며 내용을 저장하거나 재호출하거나 메인 메모리에 대용량의 데이터 블록을 이용하는데 매우 효율적인 명령어이다.

레지스터 목록

명령어는 현재의 뱅크 안의 임의의 레지스터를 전송하는데 사용된다. 레지스터 목록은 16 비트 field로 이루어진 명령어이며, 각 비트는 레지스터에 대응된다. 레지스터 field의 비트0 의 1은 R0를 전송하며 0은 전송이 안되도록 한다; 비슷하게 비트1은 R1의 전송을 컨트롤한다.

레지스터의 서브셋이나 모든 레지스터가 기록될 수도 있다. 1가지 제한이 있다면 레지스터 목록은 비어있지 않아야 한다.

R15가 메모리에 저장될 때마다 저장된 값은 STM 명령어에 12를 더한 어드레스가 된다.

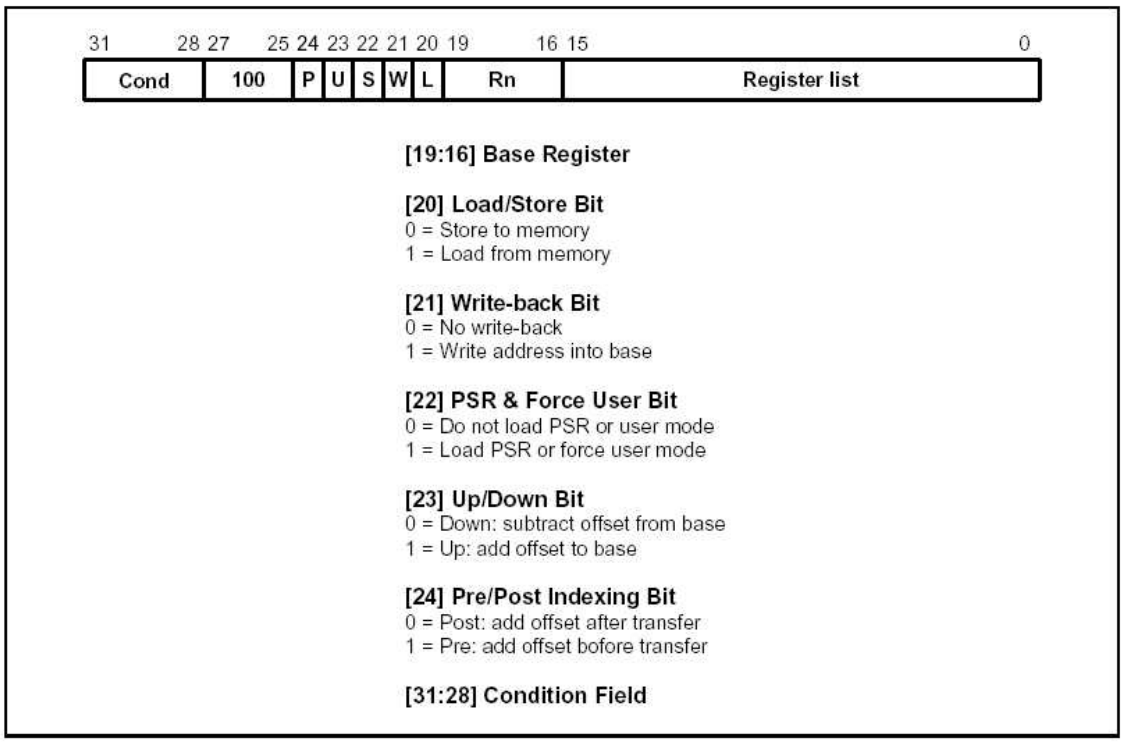

그림 3-18. 블록 데이터 전송 명령어

어드레싱 모드

어드레스 전송은 base 레지스터(Rn), pre/post 비트(P)와 up/다운 비트(U)의 내용으로 결정된다. 레지스터는 하위 차순에서 상위 차순으로 전송되며 R15는 항상 마지막에 전송된다. 가장 낮은 차순의 레지스터는 가장 낮은 차순의 메모리 어드레스에 전송된다. 이러한 방식으로, Rn=0x1000이고 변경된 base의 write back가 필요한(W=1) 경우에 R1, R5, R7의 전송을 고려한다. 그림 3.19-22는 레지스터 전송, 사용되는 어드레스의 순서와 명령어가 완료된 후의 Rn의 값을 나타낸다.

모든 경우에, 변경된 base의 write back은 필요하지 않으며(W=0), 호출된 값에 덮어 쓰기를 할 때 멀티 레지스터 명령어의 전송 목록에 있지 않으면 Rn은 0x1000의 초기 값을 얻는다.

어드레스 정렬

어드레스는 워드 정렬이며 non-워드 정렬 어드레스는 명령어에 영향을 받지 않는다. 어쨌든, 어드레스의 하위 2비트가 A[1:0]에 나타나며, 메모리 시스템에 의해서 해석되어진다.

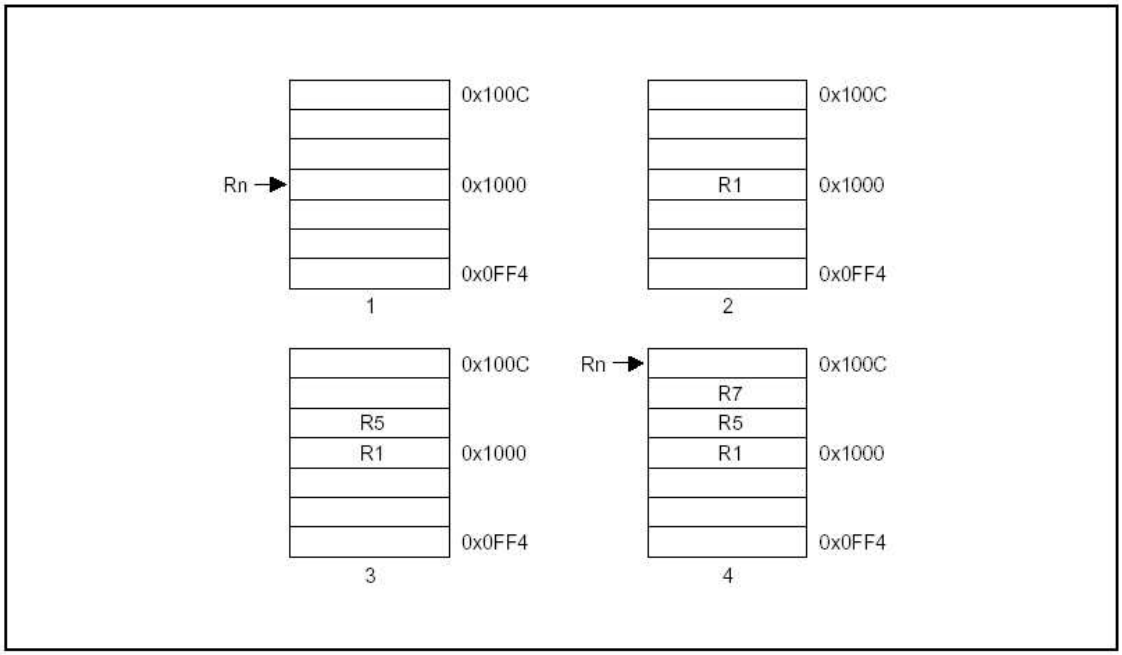

그림 3-19. post-증가 어드레싱

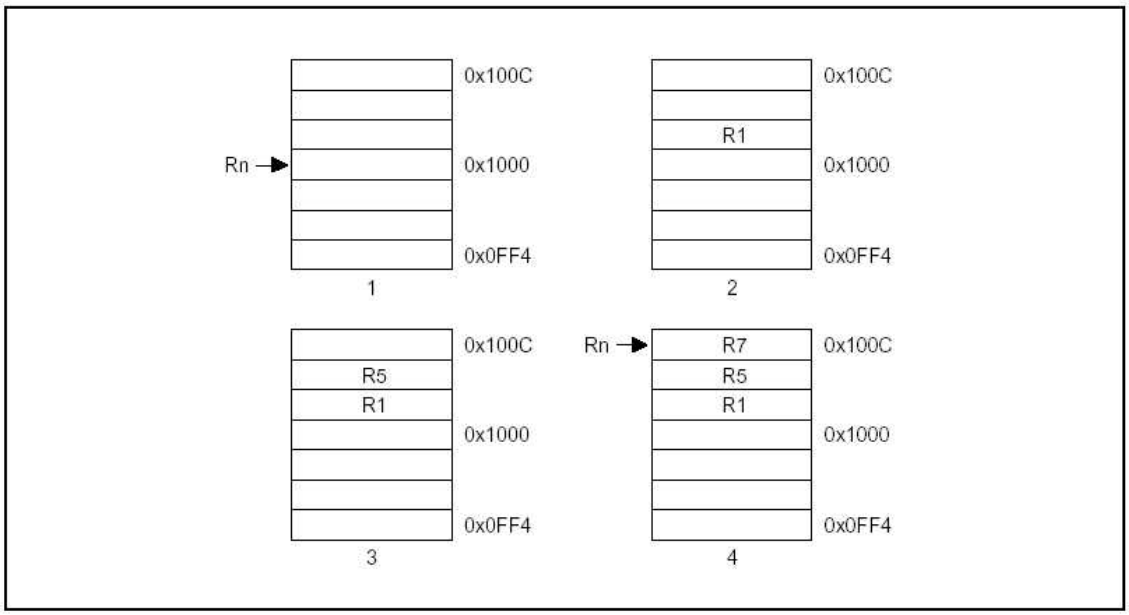

그림 3-20. pre-증가 어드레싱

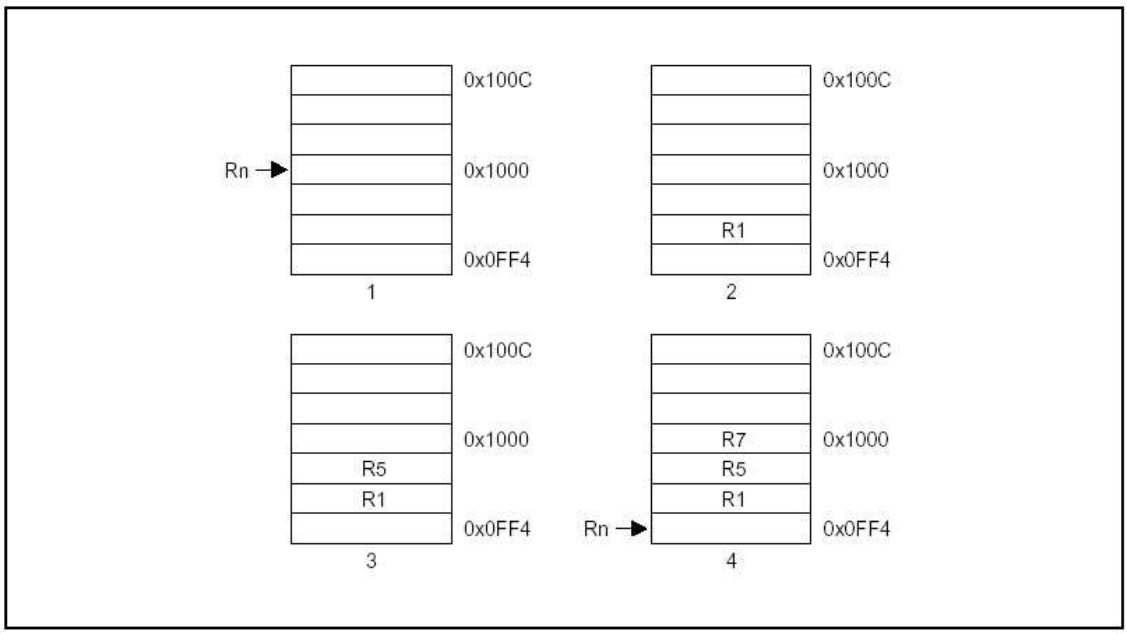

그림 3-21. post-감소 어드레싱

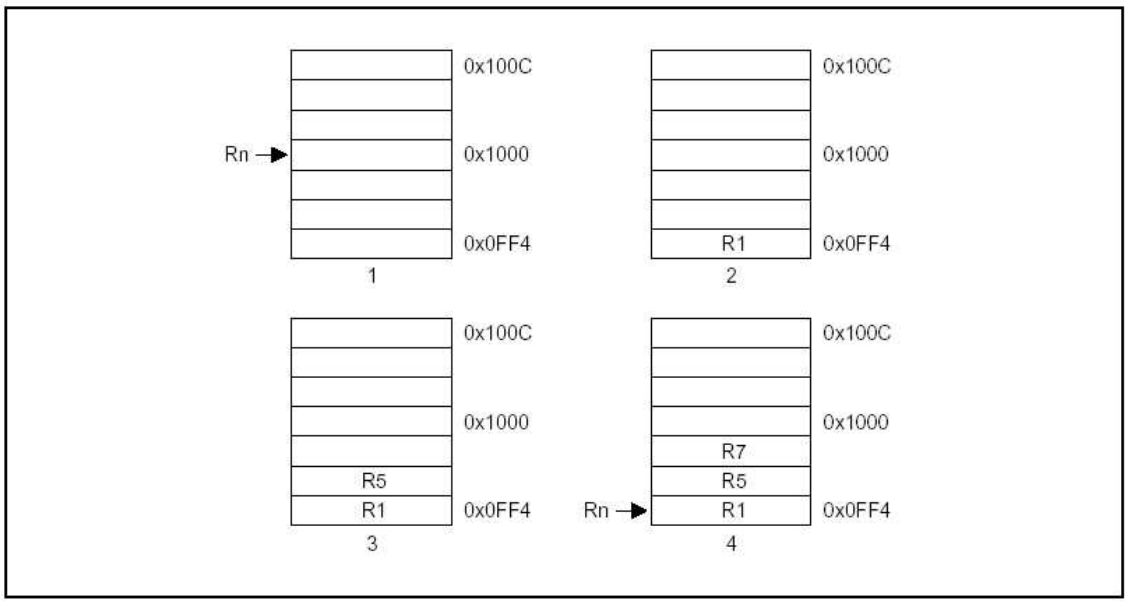

그림 3-22. pre-감소 어드레싱

S비트 사용

S비트가 LDM/STM 명령어에 설정될 때 R15가 전송 목록에 있는지와 명령어의 형태를 나타낸다. S비트는 명령어가 특권 모드에서 실행될 경우에만 설정되어야 한다.

전송목록과 S비트 설정(모드 변경)안의 R15를 갖는 LDM

명령어가 LDM이면 SPSR_<mode>는 R15가 호출됨과 동시에 CPSR에 전송된다.

전송목록에 R15를 갖는 STM과 S비트 설정(사용자 뱅크 전송)

전송되는 레지스터는 현재의 모드에 대응되는 뱅크 보다는 사용자 뱅크로부터 온다. 프로세스 스위칭에서 사용자의 상태를 저장하는데 유용하다. 이 메카니즘이 적용되면 base write-back가 사용되어서는 안된다.

R15가 목록에 없으며 S비트 설정(사용자 뱅크 전송)

LDM과 STM 명령어에 대해서, 사용자 뱅크 레지스터는 현재의 모드에 대응되는 레지스터 뱅크보다 전송된다. 프로세스 스위칭 상태에서 사용자의 상태를 저장하는데 유용하다. 이러한 메카니즘이 적용될 때 base write-back는 사용되지 않는다.

명령어가 LDM이면, 다음의 사이클 동안에 뱅크 레지스터로부터 읽을 수 없다.

R15를 base로 사용하기

R15는 LDM이나 STM 명령어에서 base 레지스터로 사용될 수 없다.

레지스터 목록에 base 포함하기

write-back가 정의되면, base는 명령어의 2번째 끝에서 write back 된다. STM 동안에, 처음 레지스터는 두 번째 사이클의 시작에서 기록된다. base를 저장될 처음 레지스터로 가지면서 base를 저장하는 STM은 변경되지 않은 값을 저장하지만, 2번째 base 이상의 전송 차순은 변경된 값을 저장한다. base가 목록에 있으면 LDM은 항상 업데이트 된 base를 덮어쓰기를 한다.

데이터 abort

몇 가지의 legal 어드레스는 메모리 관리 시스템에서 받아들여지지 않으며, 메모리 관리자는 abort 신호를 high로 만들어서 어드레스를 갖는 문제를 가질 수 있다. 이렇게 되면, 멀티 레지스터의 호출이나 저장 동안에 전송이 이루어 질 수 있으며, ARM920T가 가상의 메모리 시스템에 사용되면 회복이 가능하다.

STM 명령어 동안의 Abort

여러 개의 명령어를 저장하는 동안에 abort가 발생하면, ARM920T는 명령어가 완료될 때까지 약간의 액션을 취하며 데이터 abort 트랩으로 진입한다. 메모리 관리자는 메모리에 에러가 있는 쓰기를 방지할 책임이 있다. 프로세서의 내부 상태에 대한 변경은 write-back가 규정되면 base 레지스터의 변경이 이루어지며, 명령어가 재실행 되기 전에 소프트웨어에 의해서 전환된다.

LDM 명령어 동안의 abort

ARM920T가 여러 가지의 명령어를 호출 하는 동안에 데이터를 검출할 때, 회복이 가능한 명령어의 동작을 변경한다.

☞ abort가 발생할 때 레지스터를 덮어쓰기를 해서 정지한다. aborting 호출은 발생하지 않지만, 초기에 레지스터를 덮어쓰기를 할 수도 있다. PC는 항상 기록될 마지막 레지스터이며 보존된다.

☞ base 레지스터는 write-back가 요청되면 변경된 값에 복구된다. base 레지스터가 전송 목록에 있는 경우에는 abort가 발생되기 전에 덮어쓰기를 한다.
데이터 abort 트랩이 호출된 여러 가지의 명령어가 완료되면 발생하며, 시스템 소프트웨어는 명령어를 재시작하기 전에 base 변경을 하지 않앙 i한다.

명령어 사이클 시간

Normal LDM 명령어는 nS+1N+1I의 시가닝 걸리며 LDM PC는 (n+1)S+2N+1I의 사이클이 증가하며, 여기서, S와 N과 I는 S-사이클, N-사이클, I-사이클로 정의된다. STM 명령어는 (n-1)S+2N의 증가 사이클이 걸리며, 여기서 n은 전송될 워드의 개수이다.

어셈블러 문법

<LDM|STM>{cond}<FD|ED|FA|EA|IA|IB|DA|DB> Rn{!}, <Rlist>{^}

여기서:

{cond}	2문자 condition mnemonic. 표 3-2를 참조
Rn	유효한 레지스터 개수를 평가하는 expression
<Right>	{} 안의 레지스터의 목록과 레지스터의 범위(즉 {R0, R2-R7, R10})
{!}	write-back 요청이 있으면(W=1), 그렇지 않으면 W=0.
{^}	PC를 갖는 CPSR을 호출하기 위해서 S비트를 설정하거나, 특권 모드에 있을 경우네 사용자 뱅크를 강제로 전송함

어드레싱 모드의 이름

명령어가 다른 목적으로 스택을 지원하는데 사용되는 명령어에 의존하는 각 어드레싱 모드에 대해서 여러 가지의 어셈블러 mnemonic이 존재한다. 명령어 안의 이릅과 비트 값 사이의 등가관계가 표3-6에 나타나 있다.

표 3-6. 어드레싱 모드의 이름

Name	Stack	Other	L bit	P bit	U bit
Pre-Increment Load	LDMED	LDMIB	1	1	1
Post-Increment Load	LDMFD	LDMIA	1	0	1
Pre-Decrement Load	LDMEA	LDMDB	1	1	0
Post-Decrement Load	LDMFA	LDMDA	1	0	0
Pre-Increment Store	STMFA	STMIB	0	1	1
Post-Increment Store	STMEA	STMIA	0	0	1
Pre-Decrement Store	STMFD	STMDB	0	1	0
Post-Decrement Store	STMED	STMDA	0	0	0

FD, ED, FA, EA는 요구되는 스택 형태에 대한 레퍼런스의 pre/post 인덱싱과 업/다운 비트를 정의한다. F와 E는 "full"이나 “empty" 스택을 참조하는데, 여기서 pre-index는 스택을 저장하기 전에 이루어진다. A와 D는 스택이 상승할지 하강할지를 나타낸다. 상승하면 STM은 올라가고 LDM은 내려오는데, 하강하면 반대로 된다.

IA, IB, DA, DB는 LDM/STM이 스택용으로 사용되지 않고 증가 후, 증가 전, 감소 후, 감소

전을 의미하며 컨트롤 된다.

예제

```
LDMFD  SP!,{R0, R1, R2}        ; 3 레지스터의 unstack
STMIA  R0,{R0-R15}             ; 모든 레지스터 저장하기
LDMFD  SP!,{R15}               ; R15 <- (SP), CPSR이 바뀌지 않음
LDMFD  SP!,{R15}^              ; R15 <- (SP), CPSR <- SPSR_mode
STMFD  R13,{R0-R14}^           ; 스택에 사용자 모드 레지스터를 저장
```

이러한 명령어들은 서브루틴 엔트리의 상태를 저장하는데 사용되며, 콜 루틴에 대한 리턴을 저장한다:

```
STMED  SP!,{R0-R3,R14}         ; workspace로 사용하기 위해서 R0를 R3에 저장
                               ; 하고 R4를 반환한다.
BL     somewhere               ; nested 콜이 R14를 덮어쓰기를 한다.
LDMED  SP!,{R0-R3,R15}         ; workspace를 복귀하고 반환한다.
```

단일 데이터 스왑(SWP)

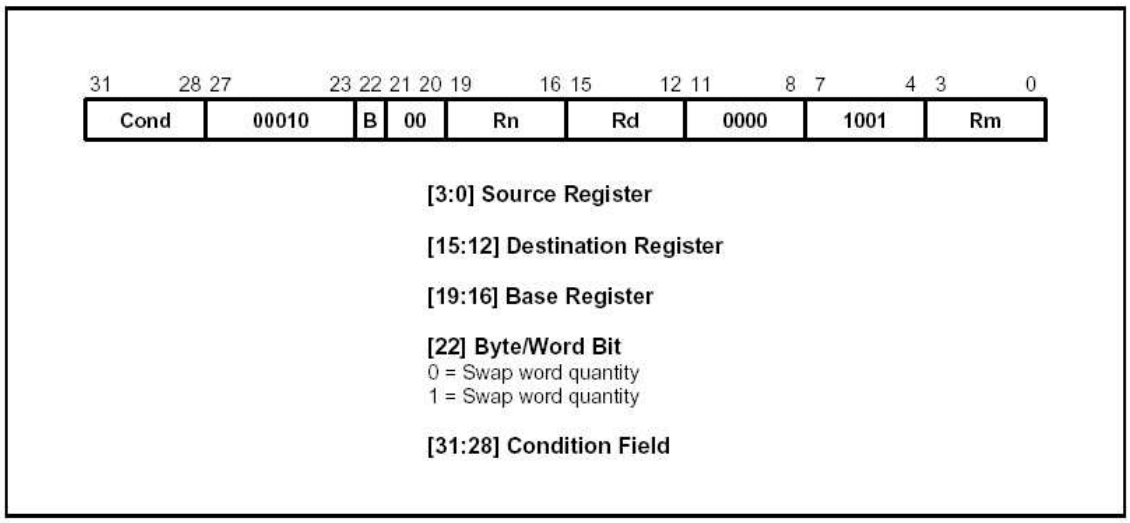

그림 3-23. 스왑 명령어

이 명령어는 condition이 true 인 경우에만 실행된다. 다양한 상태가 표 3-2에 정의되어 있다. 명령어 엔코딩은 그림 3-23에 나타나 있다.

데이터 스왑 명령어는 레지스터와 외부의 메모리 사이의 바이트나 워드를 스왑하는데 사용된다. 이 명령어는 함께 잠긴 메모리 쓰기와 같이 메모리 읽기에 적용된다. 이러한 부류의 명령어는 특히 소프트웨어 세마포어를 적용하는데 유용하다.

스왑 어드레스는 base 레지스터(Rn)의 내용에 의해서 결정된다. 먼저 프로세서는 스왑 어드레스의 내용을 읽는다. 그러고 나서 소스 레지스터의 내용(Rm)을 스왑 어드레스에 쓰기를 하며 이전의 메모리 내용을 목적지 레지스터(Rd)에 저장한다. 같은 레지스터가 소스와 목적지를 규정한다.

LOCK 출력은 읽기와 쓰기 동작 동안에 High로 되며 함께 잠기며 인터럽트의 개입없이도 완료되는 외부의 메모리 관리자에 신호를 보낸다. 스왑 명령어가 세마포어를 적용하는데 사용되는 보이지 않는 명령어를 갖는 멀티-프로세서 시스템에서 중요하다; lock 동작을 수행하는 동안에 메모리 컨트롤은 프로세서로부터 제거되어서는 안된다.

바이트와 워드

이 명령어 부류는 ARM920T와 메모리 사이에 바이트(B=1)나 워드(B=0)를 스왑하는데 사용된다. SWP 명령어는 STR에 의한 LDR로 적용되며 이러한 명령어의 설명은 단일 데이터 전송 부분을 참조하시오. 특히, 빅 엔디안과 리틀 엔디안 설정이 SWP 명령어에 적용된다.

R15를 사용하기

R15를 SWP 명령어의 오프랜드(Rd, Rn 혹은 Rs)로 사용하지 마시오.

데이터 Abort

스왑 용으로 사용되는 어드레스가 메모리 관리 시스템에서 예측이 불가능하면, 메모리 관리자는 abort를 high로 만들어서 문제를 해결한다. 이렇게 하면 읽기 사이클이나 쓰기 사이클이 발생하며 이러한 경우에 데이터 abort 트랩이 발생한다. 시스템 소프트웨어가 이러한 문제를 해결하는 것이 필요하며 명령어가 재시작 되고 원래의 프로그램이 계속된다.

명령어 사이클 시간

스왑 명령어는 1S+2N+1I의 증가 사이클이 필요하며, 여기서 S, N, I는 S-사이클, N-사이클, I-사이클이다.

어셈블러 문법

<SWP>{cond}{B}Rd, Rm, [Rn]

{cond} 2문자 condition mnemonic. 표 3-2를 참조하시오.

{B} B가 바이트 전송을 나타내며, 그렇지 않으면 워드 전송을 나타냄

Rd, Rm, Rn 유효한 레지스터 개수를 평가하는 expression

예제

```
        SWP     R0, R1, [R2]    ; R2에 의한 워드 어드레스를 갖는 R0를 호출
                                ; R1을 R2에 저장
        SWPB    R2, R3, [R4]    ; R4에 의한 바이트 어드레스를 갖는 R2를 호출
                                ; R3의 비트0에서 7까지를 R4에 저장
        SWPEQ   R0, R0, [R1]    ; R0를 갖는 R1에 의한 워드 어드레스의 내용을
                                ; 스왑
```

소프트웨어 인터럽트(SWI)

이 명령어는 condition이 true인 경우에만 실행된다. 다양한 상태가 표 3-2에 나타나 있다. 명령어 엔코딩은 그림 3-24에 나타나 있다.

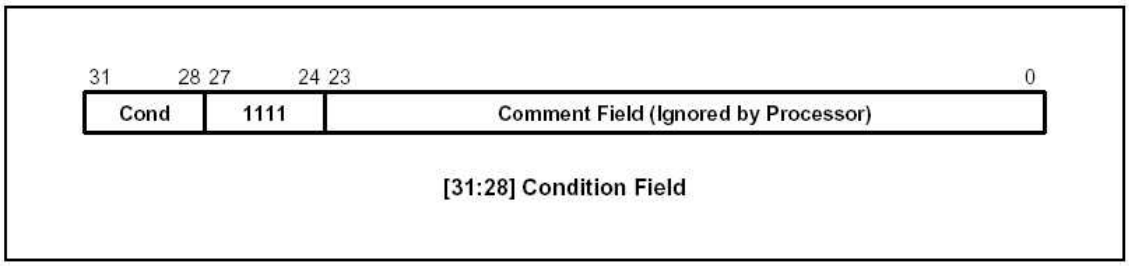

그림 3-24. 소프트웨어 인터럽트 명령어

소프트웨어 인터럽트 명령어는 컨트롤러 방식에서 슈퍼바이저 모드로 진입하는데 사용된다. 이 명령어는 모드를 변경할 수 있는 소프트웨어 인터럽트 트랩을 발생한다. PC는 고정값(0x08)으로 고정되며, CPSR은 SPSR_svc에 저장된다. SWI 벡터 어드레스가 사용자에 의해서 변경이 불가능하면, 보호 OS가 만들어진다.

슈퍼바이저에서 복귀

PC는 소프트웨어 인터럽트 트랩으로 진입하면서 R14_svc에 저장되며, PC는 SWI 명령어 후에 워드에 대한 포인트로 조정된다. MOVS PC, R14_svc는 프로그램을 호출하면서 복귀되고 CPSR을 회복한다.

링크 메카니즘은 re-entrant가 아니며, 즉 슈퍼바이저 코드가 자신의 소프트웨어 인터럽트를 사용하고자 하면 먼저 복귀 어드레스와 SPSR의 복사본을 저장해야 한다.

comment field

명령어의 하위 24비트는 프로세서에 의해서 무시되며, 슈퍼바이저 코드에 정보를 제공하는데 사용된다. 예로, 슈퍼바이저는 이 field를 관찰하고 다양한 슈퍼바이저 기능을 수행하는 루틴에 대한 엔트리 포인트의 배열에 인덱싱하는데 사용한다.

명령어 사이클 시간

소프트웨어 인터럽트 명령어는 2S+1N의 증가 사이클을 소비하는데, 여기서 S와 N은 각각 S-사이클과 N-사이클을 나타낸다.

어셈블러 문법

SWI{cond}<expression>

{cond}　　2문자 conditoin mnemonic, 표 3-2를 참조하시오.

<expression>　　평가하고 comment field에 놓는다.

예제

```
        SWI     ReadC           ; 읽기 stream에서 다음의 문자를 얻는다.
        SWI     Writel+"k"      ; 쓰기 stream에 "k"를 출력
        SWINE   0               ; comment field에 0을 기록해서 슈퍼바이저를 호출
```

슈퍼바이저 코드

이전의 예제는 예로, 적정한 슈퍼바이저 코드가 있다고 가정한다:

```
        0x08 B Supervisor          ; SWI 엔트리 포인트
        EntryTable                 ; 슈퍼바이저 루틴의 어드레스
        DCD ZeroRtn
        DCD ReadCRtn
        DCD WritelRtn
        ...
        Zero    EQU    0
ReadC   EQU    256
Writel  EQU    512

        Supervisor                 ; SWI는 비트 8-23에 필요한 루틴과 비트0-7의 데이터를
                                   ; 가진다. R13_svc는 적정한 스택에 대한 포인트
        STMFD  R13,{R0-R2,R14}  ; 작업한 레지스터를 저장하고 어드레스를 반환한다.
        LDR    R0,[R14,#-4]     ; SWI 명령어를 얻는다.
        BIC    R0,R0,#0xFF000000        ; 상위 8비트를 클리어 한다.
        MOV    R1,R0,LSR#8      ; 루틴 옵셋을 얻는다.
        ADR    R2,EntryTable    ; 엔트리 테이블의 시작 어드레스를 얻는다.
        LDR    R15,[R2,R1,LSL#2]        ; 적정한 루틴으로 브랜치한다.
        WritelRtn                       ; R0의 비트0-7 문자로 진입한다.
        ...
        LDMFD  R13,{R0-R2,R15}^          ; workspace를 회복하고 복귀한다.
                                   ; 프로세서 모드와 플래그를 회복한다.
```

코프로세서 데이터 동작(CDP)

이 명령어는 condition이 true인 경우에만 실행된다. 다양한 condition이 표 3-2에 정의되어 있다. 명령어 엔코딩은 그림 3-25에 나타나 있다.

이러한 부류의 명령어는 코프로세서가 몇가지의 내부 동작을 수행하는데 사용된다. ARM920T와 back 통신할 수 없으며, 동작을 완료할 때가지 기다리지 않는다. 코프로세서는 실행을 지연하는 명령어와 같은 큐를 포함하며, 자신의 실행은 다른 동작에 대해서 오버랩되며 코프로세서와 ARM920T가 독립적인 임무를 병렬로 수행하도록 한다.

코프로세서 명령어

다른 ARM-기반의 프로세서와는 달리 S3C2410X는 외부의 코프로세서 인터페이스를 갖지 않는다. 또한 온-칩의 코프로세서를 갖지 않는다.

즉, 모든 코프로세서 명령어는 S3C2410X에서 취해지는 정의되지 않는 명령어 트랩의 원인이 된다. 이러한 코프로세서 명령어는 정의되지 않은 트랩 핸들러에 의해서 에뮬레이팅 될 수 있다. 외부의 코프로세서가 S3C2410X에 연결되지 않더라도, 코프로세서 명령어는 여기서 설명된다.

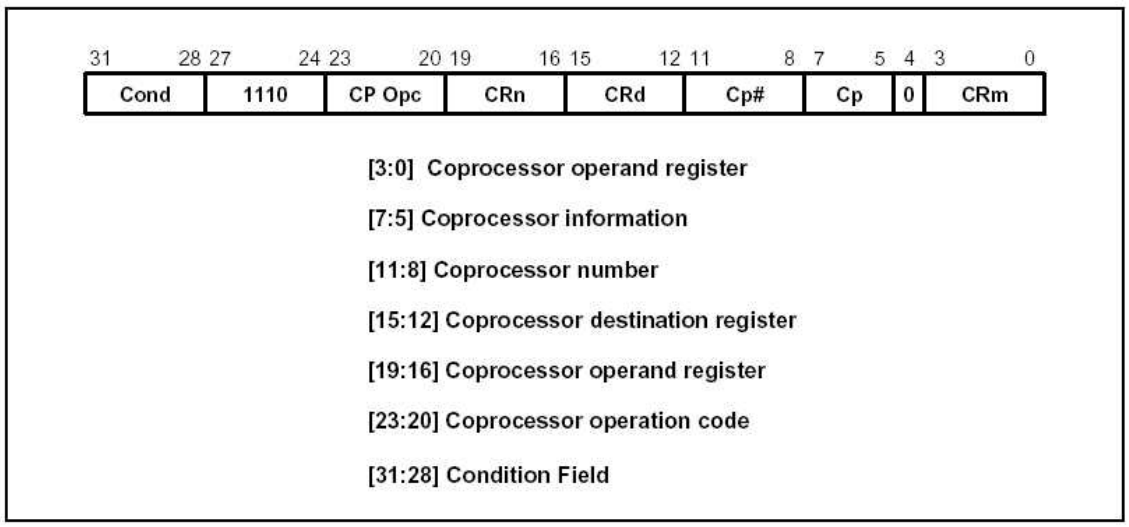

그림 3-25. 코프로세서 데이터 동작 명령어

비트4와 비트24에서 31까지의 코프로세서 field가 ARM920T에는 중요하다. 나머지 비트는 코프로세서에 의해서 사용된다. 위의 field 이름은 원래부터 있는 것이며 특히 코프로세서는 CP#을 제외한 모든 field를 다시 정의할 수 있다. CP#은 각 코프로세서에 대한 개수를 증명하는데 사용되며, 코프로세서는 CP# field 안의 number을 포함하지 않는 명령어를 무시한다. 명령어에 대한 전통적인 해석은 코프로세서가 CRn과 CRm의 내용에서 CP Opc filed에 정의된 동작을 수행하며 결과는 CRd에 놓이게 된다.

명령어 사이클 시간

코프로세서 데이터 동작은 1S+bL 증가 사이클을 필요로 하며, 여기서 b는 코프로세서 busy-wait 루프에서 소비되는 사이클의 개수이다.
S와 I는 S-사이클과 I-사이클로 정의된다.

어셈블러 문법

CDP{cond} p#, <expression1>, cd, cn, cm{,<expression2>}
{cond} 2문자 condition mnemonic. 표 3-2를 참조하시오.
p# 코프로세서에 필요한 유일한 숫자
<expression1> CP Opc filed에 놓이는 상수에 대한 평가
cd, cn과 cm 각각 유효한 코프로세서 레지스터 개수 CRd, CRn, CRm에 대한 평가
<expression2> CP field에 놓이는 상수에 대한 평가

예제

```
CDP    p1, 10, c1, c2, c3       ; CR2와 CR3의 동작 10에 대한 coproc 1을
                                ; 요청하고 결과를 CR1에 놓음
CDPEQ  p2, 5, c1, c2, c3, 2     ; Z 플래그가 CR2와 CR3의 동작 5에 대한
                                ; coproc 2를 요청하도록 설정되고,
```

; 결과는 CR1에 놓임

코프로세서 데이터 전송(LDC, STC)

condition이 true인 경우에만 실행된다. 다양한 상태가 표 3-2에 정의된다. 명령어 엔코딩은 그림 3-26에 나타나 있다.

이러한 부류의 명령어는 코프로세서 레지스터의 서브셋을 직접 메모리에 호출(LDC)하거나 저장(STC)하는데 사용된다. ARM920T는 메모리 어드레스를 공급하는 책임이 있으며 코프로세서는 전송되는 워드의 개수를 컨트롤 하고 데이터를 받아들이는데 사용된다.

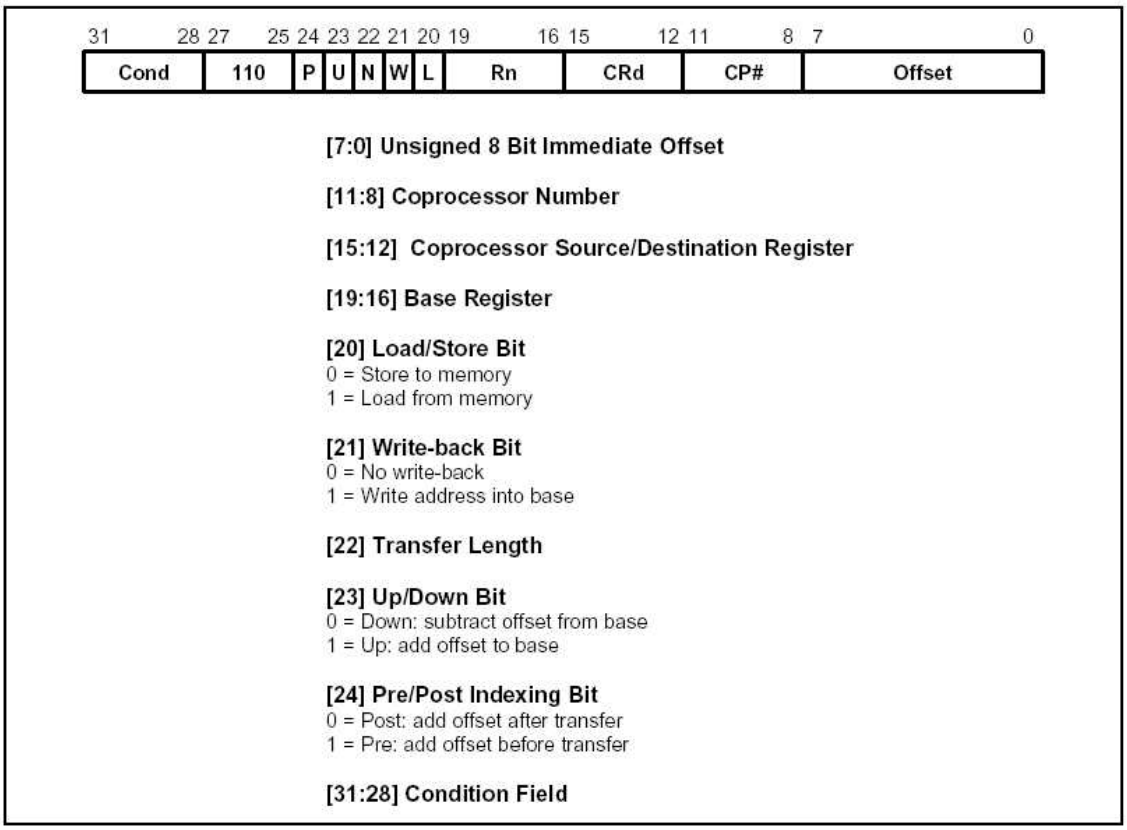

그림 3-26. 코프로세서 데이터 전송 명령어

코프로세서 field

CP#은 데이터를 공급하거나 받아들이는데 필요한 코프로세서를 증명하는데 사용되며, 자신의 숫자가 field의 내용과 매칭이 되면 코프로세서는 응답하게 된다.

CRd filed와 N비트는 여러 가지의 코프로세서에 의한 다른 방식으로 해석되는 코프로세서에 대한 정보를 포함하지만, 기존의 CRd는 전송되는 레지스터이며, N비트는 2가지의 전송 길이 옵션 중 1개를 선택하는데 사용된다. 예로, N=0이면, 단일 레지스터의 전송을 선택하고 N=1이면 내용 스위칭에 대한 모든 레지스터를 전송하는데 선택된다.

어드레싱 모드

ARM920T는 전송 용 메모리 시스템에 의해서 사용되는 어드레스를 제공하며, 어드레싱 모

드는 단일 데이터 전송 명령어에서 사용되는 서브셋에 유용하다. 어쨌든, immediate 옵셋은 8비트의 너비를 가지며 전송될 코프로세서 데이터의 워드 옵셋을 규정하며, 여기서 이들은 12비트의 너비를 가지며 단일 데이터 전송에 대한 옵셋을 규정한다.

8비트의 부호없는 immediate 옵셋은 왼쪽으로 2비트 쉬프트되며, base 레지스터(Rn)에 더하거나(U=1) 뺀다(U=0); 이러한 계산은 baserk 전송 어드레스로 사용되기 전(P=1)이나 후(P=1)에 수행된다. 변경된 base 값은 base 레지스터에 back 덮어쓰기를 하거나(W=1이면), base의 이전 값이 보존된다(W=0). post-indexed 어드레싱 모드는 post-indexed할 때 항상 write-back 되는 LDR이나 STR과는 달리 W의 셋팅을 필요로 한다.

pre-indexed 명령어의 옵셋에 의해서 변경되는 base 레지스터의 값은 첫 번째 워드의 전송 용 어드레스에 사용된다. 두번째 워드는 처음 전송보다 1워드가 높은 어드레스로부터 오며, 어드레스는 각 subsequent 가 전송될 때마다 1워드씩 증가된다.

어드레스 정렬

base 어드레스는 일반적으로 워드 정렬이어야 한다. 어드레스의 하위 2비트는 A[1:0]에서 발생되며 메모리 시스템에 의해서 해석이 가능하다.

R15의 사용

Rn이 R15이면, 사용되는 값은 명령어에 8바이트를 더한 어드레스가 된다. R15에 base write-back는 규정되지 않는다.

데이터 abort

어드레스가 legal이지만 메모리 관리자는 abort를 발생시키면, 데이터 트랩이 발생한다. 변경된 base에 write-back이 발생되면, 모든 다른 상태가 보존된다. 코프로세서는 abort의 원인이 해결된 후에 데이터 전송이 재시작 될 수 있으며, 명령어가 재시도 될 때 반복되는 동작이 있어야 한다.

명령어 사이클 시간

코프로세서 데이터 전송 명령어는 (n-1)S+2N+bI 증가 사이클이 필요하다. 여기서:

n	전송될 워드의 개수
b	코프로세서가 busy-wait 루프에서 소비하는 사이클의 개수

S, N, I는 S-사이클, N-사이클, I-사이클로 정의된다.

어셈블러 문법

<LDC|STC>{cond}{L} p#, cd, <Address>

LDC	메모리에서 코프로세서로 호출
STC	코프로세서에서 메모리에 저장
{L}	긴 전송을 할 때(N=1), 그렇지 않으면 짧은 전송을 할 때(N=0)
{cond}	2문자 condition mnemonic. 표 3-2를 참조
p#	코프로세서에 필요한 유일한 숫자
cd	CRd field에 놓이는 유효한 코프로세서 레지스터의 숫자를 평가하는

expression

<Address>	수행:
1	어드레스를 발생시키는 expression: 어셈블러는 base로 PC를 사용하는 명령어를 발생시키며 어드레스 위치에 교정된 immediate 옵셋이 expression을 평가하는데 주어진다. 이것은 PC와 관련이 있으며, pre-indexed 어드레스이다. 어드레스의 범위가 벗어나면, 에러가 발생된다.
2	pre-indexed 어드레싱 표현: [Rn] 옵셋 0 [Rn,<#expression>]{!} <expression> 바이트의 옵셋
3	post-indexed 어드레싱 표현: [Rn], <#expression> <expression> 바이트의 옵셋 {!} !이 있으면 base 레지스터에 back 기록하기 Rn 유효한 ARM920T 레지스터 숫자를 평가하는 expression

NOTES

Rn,이 R15이면, 어셈블러는 ARM920T 파이프라인에 대한 옵셋 값에서 8을 뺀다.

예제

```
LDC     p1, c2, table          ; 어드레스 테이블에서 coproc 1의 c2를 호출
                               ; PC 관련 어드레스를 이용
STCEQL p2, c3, [R5,#24]!       ; R5의 어드레스 24바이트에 coproc 2의
                               ; c3를 저장하고, R5에 이 어드레스를
                               ; back 기록하고, 긴 전송 옵션을 사용한다.
```

NOTES

어드레스 옵셋이 바이트로 표현되지만, 명령어 옵셋 field는 워드이다. 어셈블러는 옵셋을 적정하게 조정한다.

코프로세서 레지스터 전송(MRC, MCR)

이 명령어는 condition이 true 인 경우에 실행된다. 다양한 값들이 표 3-2에 정의되어 있다. 명령어 엔코딩은 그림 3-27에 나타나 있다.

이러한 부류의 명령어는 ARM920T와 코프로세서 사이의 정보를 직접 전달하는데 사용된다. ARM920T 레지스터 전송(MRC) 명령어에 대한 코프로세서의 예제는 코프로세서 안의 플로팅 포인트 값의 FIX가 될 것이며, 여기서 플로팅 포인트 넘버는 코프로세서 안의 32비트 정수로 변환되고 결과는 ARM920T 레지스터로 전송된다. 코프로세서 안의 플로팅 포인트 값으로 ARM920T 레지스터의 32비트 값의 FLOAT은 코프로세서 전송(MCR)에 대한 ARM920T 레지스터의 사용 예를 나타낸다.

이러한 명령어의 중요한 사용은 코프로세서에서 ARM920T CPSR 플래그에 직접 정보를

전달한다. 예로, 코프로세서 안의 2개의 플로팅 포인트 값의 비교 결과는 실행될 subsequent 흐름을 컨트롤하도록 CPSR로 이동할 수 있다.

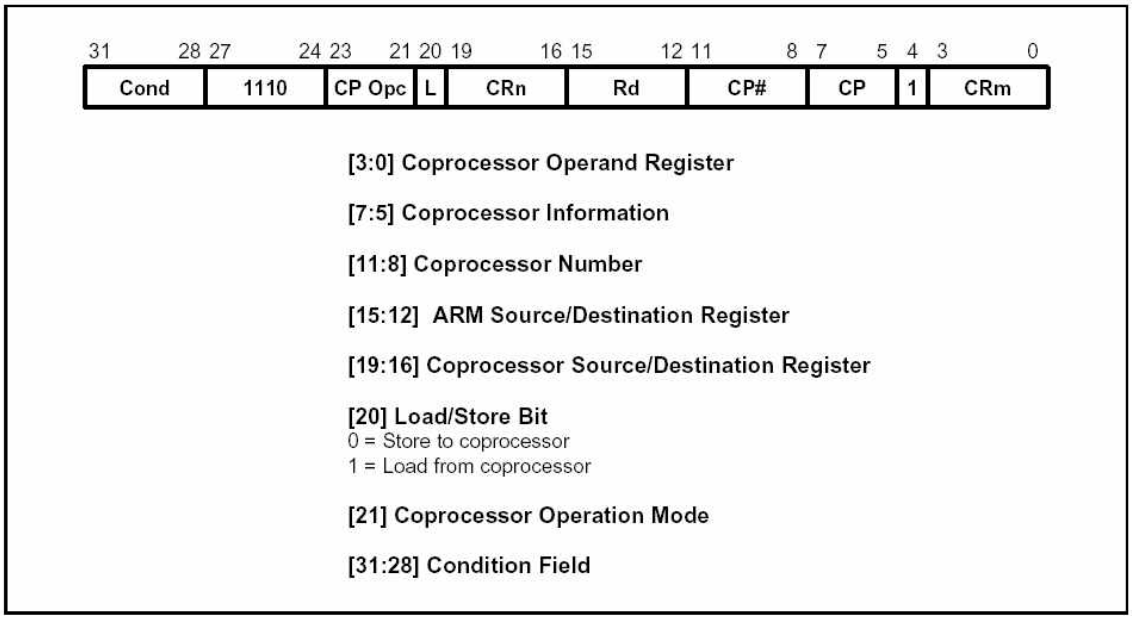

그림 3-27. 코프로세서 레지스터 전송 명령어

코프로세서 field

CP# field는 모든 코프로세서 명령어에 대해서 사용되며, 코프로세서가 호출된다.
CP Opc, CRn, CP와 CRm field는 코프로세서에 의해서만 사용되며 여기에 나타난 설명은 이전부터 있던 것이다. 코프로세서의 기능이 이것과 호환되지 않는 해석도 있다. 이전의 해석은 CP Opc와 CP field가 코프로세서가 수행하는데 필요한 동작을 정의하며, CRn은 전송될 정보의 소스나 목적을 나타내는 코프로세서 레지스터이며, CRm은 특별한 동작에 의존하는 방식으로 포함되는 2번째 코프로세서의 레지스터이다.

R15에 전송

ARM920T에 전송될 코프로세서 레지스터는 목적지로 R15를 가지며, 전송될 워드의 비트 31, 30, 29, 28은 N, Z, C, V 플래그에 복사된다. 전송될 워드의 다른 비트는 무시되며, CP와 다른 CPSR 비트는 전송에 영향을 받지 않는다.

R15로부터의 전송

MRC 명령어는 1S+(b+1)I+1C만큼 증가하는 사이클이며, S, I, C는 S-사이클, I-사이클, 코프로세서 레지스터 전송(C-사이클)을 나타낸다. MCR 명령어는 1S+bI+1C의 사이클이 증가되며, 여기서 b는 코프로세서 busy-wait 루프에서 소비되는 사이클의 개수이다.

어셈블러 문법

<MCR|MRC>{cond} p#,<expression1>, Rd, cn, cm{,<expression2>}
MRC 코프로세서에서 ARM920T 레지스터로 이동(L=1)

MCR	ARM920T 레지스터에서 코프로세서로 이동(L=0)
{cond}	2문자 condition mnemonic. 표 3-2를 참조
p#	필요한 코프로세서의 유일한 숫자
<expression1>	CP Opc field에 놓이는 상수에 대한 평가
Rd	유효한 ARM920T 레지스터 숫자를 평가하는 expression
cn과 cm	유효한 코프로레서 레지스터 숫자 CRn과 CRm을 평가하는 expression
<expression2>	CP field 안의 놓이느 s상수를 평가

예제

```
        MRC    p2, 5, R3, c5, c6        ; c5와 c6의 동작 5를 수행하기 위해서
                                        ; coproc 2를 요청하고 결과를 R3에 back 전송
        MCR    p6, 0, R4, c5, c6        ; R4의 동작 0을 수행하는데 coproc6을
                                        ; 요청하고 결괄르 c6에 넣음
        MRCEQ  p3, 9, R3, c5, c6, 2     ; c5와 c6에서 동작 9를 수행하기 위해서
                                        ; coproc 3을 요청하고
                                        ; 결과를 R3에 back 전송함
```

정의되지 않은 명령어

condition이 true 인 경우에만 명령어가 실행된다. 다양한 상태가 표 3-2에 규정되어 있다. 명령어 형식이 그림 3-28에 나타나 있다.

그림 3-28. 정의되지 않은 명령어

condition이 true이면, 정의되지 않은 명령어 트랩이 발생한다.
정의되지 않은 명령어 메카니즘이 나타나는 코프로세서에 이 명령어를 제공하며, 모든 코프로세서는 CPA와 CPB를 high 구동해서 받으들이거나 거절해야 한다.

명령어 사이클 시간

이 명령어는 2S+1I+1N의 사이클이 필요하며, 여기서 S, N, I는 S-사이클, N-사이클, I-사이클로 정의된다.

어셈블러 문법

어셈블러는 이러한 명령어를 발생하는 mnemonic을 가지고 있지 않다. 몇몇의 사용을 위해서 나중에 채택되면, 적정한 mnemonic이 어셈블러에 추가된다. 이 시간 까지는 이 명령어는 사용되지 않는다.

명령어 설정 예제

아래의 예제는 기본적인 ARM920T 명령어가 코드를 효율적으로 사용하는 방법에 대해서 나타낸다. 이러한 방법이 실행시간을 크게 단축시키는 것은 아니지만, 코드를 줄일 수 있다.

confitional 명령어를 사용하기

logical OR에 대한 conditional 사용

```
        CMP     Rn, #p          ; Rn=p OR Rm=q이면 GOTO 레벨로
        BEQ     Label
        CMP     Rm, #q
        BEQ     Label
```

다음과 같이 대체된다.

```
        CMP     Rn, #p
        CMPNE   Rm, #q          ; condition이 다른 테스트 시도를 만족하지 않을 경우
        BEQ     Label
```

절대 값

```
        TEQ     Rn, #0          ; 테스트 sign
        RSBMI   Rn, Rn, #0      ; 필요하면 2의 보수
```

4, 5, 6의 곱셈

```
        MOV     Rc, Ra, LSL#2   ; 4를 곱하기
        CMP     Rb,#5           ; 테스크 값
        ADCS    Rc, Rc, Ra      ; 5를 곱하기 완료
        ADDHI   Rc, Rc, Ra      ; 6을 곱하기 완료
```

discrete 결합과 범위 테스트

```
        TEQ     Rc, #127        ; discrete 테스트
        CMPNE   Rc, #""-1       ; 범위 테스트
        MOVLS   Rc, #""         ; Rc<="" OR Rc=ASCII(127)이면
                                ; Rc:=""임
```

나눗셈과 나머지

특별한 어플리케이션 용의 나누기 루틴은 ARM크로스 개발 툴킷으로 제공되는 ANSIㅊ 라이브러리의 파트로써 소스 형식으로 제공된다. 짧은 범용 나눗셈 루틴은 아래와 같다.

```
                                ; Ra와 Rb에 숫자 입력
        MOV     Rcnt,#1         ; 나누기 컨트롤 비트
Div1    CMP     Rb,#0x80000000  ; Ra보다 클 때까지 Rb로 이동
        CMPCC   Rb, Ra
        MOVCC   Rb,Rb,ASL#1
```

```
        MOVCC  Rcnt,Rcnt,ASL#1
        BCC    Div1
        MOV    Rc,#0
Div2    CMP    Ra,Rb            ; 가능한 뺄셈 테스트
        SUBCS  Ra,Ra,Rb         ; ok이면 빼기
        ADDCS  Rc,Rc,Rcnt       ; 관련 비트를 결과에 넣음
        MOVS   Rcnt,Rcnt,LSR#1  ; 컨트롤 비트를 쉬프트
        MOVNE  Rb,Rb,LSR#1      ; 끝나지 않으면 2등분
        BNE    Div2             ; Rc에 결과를 나누고, Ra에 나머지
```

ARM920T의 오버플로우 검출

1. 32비트 결과를 갖는 부호없는 곱셈의 오버플로우

```
        UMULL  Rd,Rt,Rm,Rn      ; 6사이클에 대한 3
        TEQ    Rt,#0            ; +1 사이클과 레지스터
        BNE    overflow
```

2. 32비트 결과를 갖는 부호있는 곱셈의 오버플로우

```
        SMULL  Rd,Rt,Rm,Rn      ; 6사이클에 대한 3
        TEQ    Rt,Rd ASR#31     ; +1 사이클과 레지스터
        BNE    overflow
```

3. 32비트 결과를 갖는 부호없는 곱셈 축적의 오버플로우

```
        UMULL  Rd,Rt,Rm,Rn      ; 7사이클에 대한 4
        TEQ    Rt,#0            ; +1 사이클과 레지스터
        BNE    overflow
```

4. 32비트 결과를 갖는 부호있는 곱셈 축적의 오버플로우

```
        SMLAL  Rd,Rt,Rm,Rn      ; 7사이클에 대한 4
        TEQ    Rt,Rd ASR#31     ; +1 사이클과 레지스터
        BNE    overflow
```

5. 64비트 결과를 갖는 부호없는 곱셈 축적의 오버플로우

```
        UMULL  RI,Rh,Rm,Rn      ; 6사이클에 대한 3
        ADDS   RI,RI,Ra1        ; 하위 축적
        ADC    Rh,Rh,Ra2        ; 상위 축적
        BCS    overflow         ; 1사이클과 2레지스터
```

6. 64비트 결과를 갖는 부호있는 곱셈 축적의 오버플로우

```
        SMULL  RI,Rh,Rm,Rn      ; 6사이클에 대한 3
        ADDS   RI,RI,Ra1        ; 하위 축적
        ADC    Rh,Rh,Ra2        ; 상위 축적
        BVS    overflow         ; 1사이클과 2레지스터
```

NOTES

오버플로우가 위와 같은 계산으로 발생되기 않기 때문에 64비트를 갖는 부호없는/부호있는 멀티에 오버플로우 체킹이 적용되지 않는다.

의사-랜덤 바이너리 시퀀스 발생기

의사-랜덤 숫자를 발생하는 것이 필요하며 대부분의 효율적인 알고리즘은 주기적인 redundancy 체크 발생기 보다는 exclusitive-OR 피드백을 갖는 쉬프트 발생기에 기반한다. 불행하게도 32비트 발생기의 시퀀스는 최대 길이에 비해서 피드백 탭 이상이 필요하며, 이 예제는 비트33과 20에서 탭을 갖는 32비트 레지스터를 이용한다. 기본적인 알고리즘은 newbit:=bit 33 eor 비트 20이며, 왼쪽으로 33비트 쉬프트 하며 하위에 newbit를 놓는다; 이러한 동작은 필요한 모든 newbits에서 수행된다. 전체 동작은 5S 사이클 동안에 실행될 수 있다:

```
                                    ; Ra, Rb에 seed를 입력
                                    ; Rc를 사용
        TST     Rb,Rb,LSR#1          ; 캐리 할 상위 비트
        MOVS    Rc,Ra,RRX           ; 33비트 오른쪽으로 회전
        ADC     Rb,Rb,Rb            ; Rb의 Isb에 캐리
        EOR     Rc,Rc,Ra,LSL#12      ;
        EOR     Ra,Rc,Rc,LSR#20      ; 전과 같이 Ra,Rb의 새로운 seed
```

배럴 쉬프터를 이용한 상수 곱셈

2^n(1,2,4,8,16,32...)에 의한 곱셈

```
        MOV     Ra,Rb,LSL#n
```

2^n+1(3,5,7,9,17...)에 의한 곱셈

```
        ADD     Ra,Ra,Ra,LSL#n
```

2^n-1(3,7,15...)에 의한 곱셈

```
        RSB     Ra,Ra,Ra,LSL#n
```

6에 의한 곱셈

```
        ADD     Ra,Ra,Ra,LSL#1       ; 3으로 곱함
        MOV     Ra,Ra,LSL#1          ; 2로 곱함
```

10으로 곱하고 외부의 수를 더함

```
        ADD     Ra,Ra,Ra,LSL#2       ; 5로 곱함
        ADD     Ra,Rc,Ra,LSL#1       ; 2로 곱하고 다음의 디지트를 더함
```

Rb:=Ra*C에 대한 일반적인 방법, C는 상수:

1. C가 짝수이면, C=2^n*D이며, D는 홀수:

```
        D=1:    MOV Rb,Ra,LSL #n
        D<>1:   {Rb:=Ra*D}
        MOV     Rb,Rb,LSL #n
```

2. C MOD 4=1이면, C=2^n*D+1이며, D는 홀수이고, n>1임:

```
        D=1:    ADD Rb,Ra,Ra,LSL #n
        D<>1:   {Rb:=Ra*D}
        MOV     Rb,Ra,Rb,LSL #n
```

3. C MOD 4=3이면, C=2^n*D-1이며, D는 홀수이고, n>1임:

```
        D=1:    RSB Rb,Ra,Ra,LSL #n
        D<>1:   {Rb:=Ra*D}
        RSB     Rb,Ra,Rb,LSL #n
```

최적은 아니지만, 최적 상태에 가깝다. 비-최적의 예는 45로 곱해서 이루어진다:

```
        RSB     Rb,Ra,Ra,LSL #2         ; 3을 곱함
        RSB     Rb,Ra,Rb,LSL #2         ; 4*3-1=11을 곱함
        ADD     Rb,Ra,Rb,LSL #2         ; 4*11+1=45를 곱함
```

다음 상황도 고려:

```
        ADD     Rb,Ra,Ra,LSL #3         ; 9를 곱함
        ADD     Rb,Rb,Rb,LSL #2         ; 5*9=45를 곱함
```

모르는 정렬에서 워드를 호출

```
                                ; Rd의 Rb,Rc를 사용하는 Ra의 어드레스를 입력
                                ; d는 0, 1보다 작아야 한다.
        BIC     Rb,Ra,#3         ; 워드 정렬 어드레스 얻기
        LDMIA   Rb,{Rd,Rc}        ; 64비트 포함 답변 얻기
        AND     Rb,Ra,#3         ; 바이트의 교정된 요소
        MOVS    Rb,Rb,LSL#3      ; 정렬되면 비트를 테스트
        MOVNE   Rd,Rd,LSR Rb     ; 결과 워드의 하위 도출
        RSBNE   Rb,Rb,#32        ; 다른 쉬프트 amount 얻기
        ORRNE   Rd,Rd,Rc,LSL Rb  ; 결과를 얻기 위해서 2개의 halve 결합하기
```