

ARM 프로세서의 명령어

- ❖ ARM 프로세서 명령어의 특징에 대하여 이해한다.
- ❖ 32비트 ARM 명령어의 구조 및 사용법에 대하여 이해한다.
- ❖ Architecture v5TE에서 새로 추가된 명령어를 안다.

1. ARM 프로세서의 명령어
2. 32 비트 ARM 명령어
3. 16 비트 Thumb 명령어
4. v5TE에 추가된 명령어

❖ ARM 프로세서의 명령어

- 32 비트 ARM 명령어
- 16 비트 Thumb 명령어
 - 최근의 프로세서의 경우 Thumb-2 명령어 지원
- v5TE에 추가된 명령어

❖ Jazelle core를 확장한 프로세서

- 8 비트 Java 바이트 명령어

- ❖ 모든 ARM 명령은 32 비트로 구성되어 있다.
 - Load/Store와 같은 메모리 참조 명령이나 Branch 명령에서는 모두 상대주소 (Indirect Address)방식을 사용한다,
 - Immediate 상수는 32 비트 명령어 내에 표시된다,
 - Immediate 상수는 32비트 이하의 상수이다,
- ❖ 모든 ARM 명령어는 조건부 실행이 가능하다.
- ❖ Load/Store Architecture를 사용한다.

❖ Thumb 명령어

- 32비트의 ARM 명령을 16비트로 재구성한 명령

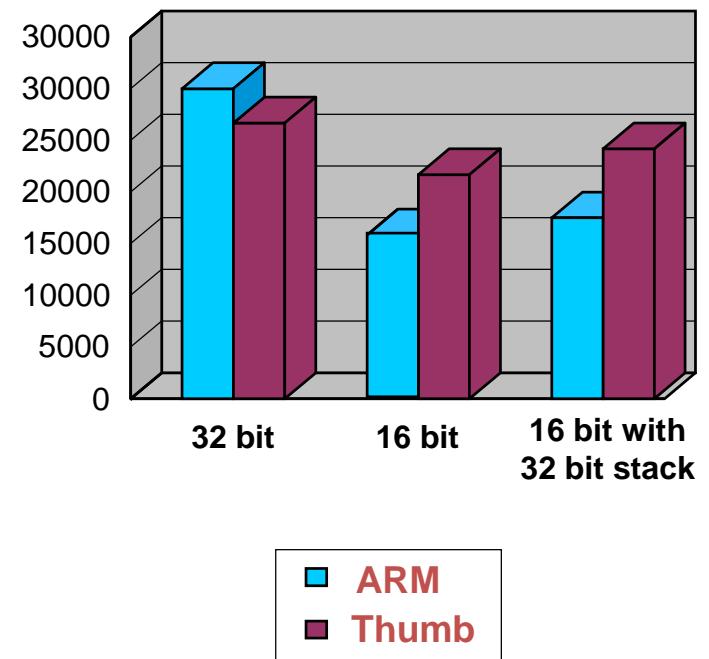
Dhrystone 2.1 / sec
@20MHz

❖ Thumb 명령어의 장점

- 코드의 크기를 줄일 수 있다.
 - ARM 명령의 65%
- 8비트 나 16비트와 같은 좁은 메모리 인터페이스에서 ARM 명령을 수행할 때 보다 성능이 우수하다.

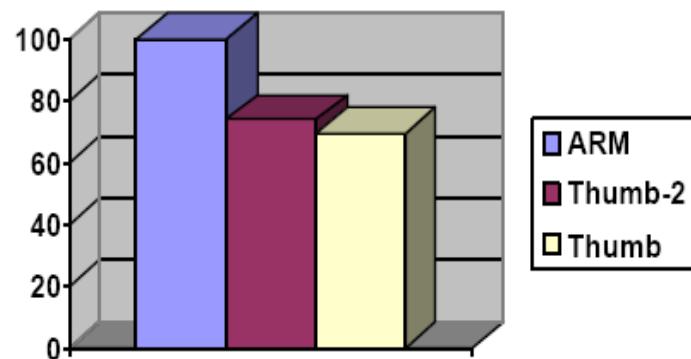
❖ Thumb 명령어의 단점

- 조건부 실행이 안 된다.
- Immediate 상수 값이 표현 범위가 적다.

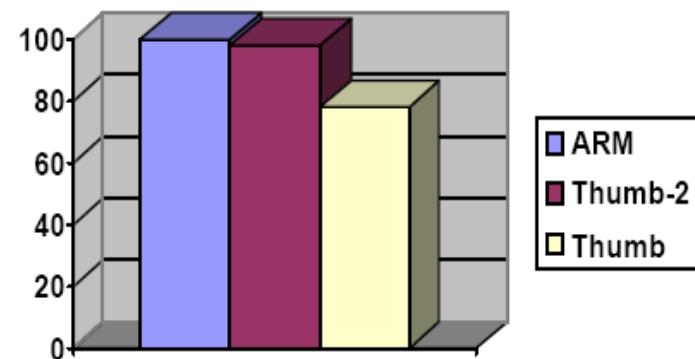


- ❖ ARM Architecture v6 이상에서 지원되는 16비트 명령어
 - 새롭게 추가된 ARM 명령과 equivalent 한 명령어 추가
- ❖ Code density와 performance를 개선
 - Memory footprint는 ARM 명령의 74 %
 - 기존의 ARM 명령에 비해 74%
 - 기존의 Thumb 명령에 비해 25% 빠르다.

Code Size



Performance



- ❖ Jazelle core가 확장되면 8 비트 Java 명령어 수행 가능
- ❖ 일반적으로 사용되는 Java 바이트 코드의 95 %를 하드웨어로 구현
 - 소프트웨어 JVM : 1.0 Caffeinemarks/MHz
 - Jazelle 내장된 ARM core : 5.5 Caffeinemarks/MHz
- ❖ Jazelle core는 12K 보다 작은 gates 수로 구현이 가능하다.

1. ARM 프로세서의 명령어
2. 32 비트 ARM 명령어
3. 16 비트 Thumb 명령어
4. v5TE에 추가된 명령어

	31	28 27		16 15	8 7	0	<u>Instruction type</u>	
Cond	0 0	I	Opcode	S	Rn	Rd	Operand 2	
Cond	0 0 0 0 0 0	A S		Rd	Rn	Rs	1 0 0 1	
Cond	0 0 0 0 1	U A S		RdHi	RdLo	Rs	1 0 0 1	
Cond	0 0 0 1 0	B 0 0		Rn	Rd	0 0 0 0	1 0 0 1	
Cond	0 1	I P U B W L		Rn	Rd	Offest		
Cond	1 0 0	P U S W L		Rn	Register List			
Cond	0 0 0	P U 1 W L		Rn	Rd	Offset1	1 S H 1	
Cond	0 0 0	P U 0 W L		Rn	Rd	0 0 0 0	1 S H 1	
Cond	1 0 1	L	Offset					
Cond	0 0 0 1	0 0 1 0	1 1 1 1	1 1 1 1	1 1 1 1	0 0 0 1	Rn	
Cond	1 1 0	P U N W L		Rn	CRd	CPNum	Offset	
Cond	1 1 1 0	Op1		CRn	CRd	CPNum	Op2	0
Cond	1 1 1 0	Op1	L	CRn	Rd	CPNum	Op2	1
Cond	1 1 1 1	SWI Number						CRm

- ❖ ARM에서 모든 명령을 조건에 따라 실행 여부 결정 가능
- ❖ Branch 명령의 사용을 줄인다.
 - Branch 명령이 사용되면 pipeline이 stall 되고 새로운 명령을 읽어오는 사이클의 낭비가 따른다.
- ❖ Condition field
 - 모든 명령어는 condition field를 가지고 있으며 CPU가 명령의 실행 여부를 결정하는데 사용된다.
 - Current Program Status Register(CPSR)의 Condition flag의 값을 사용하여 조건을 검사

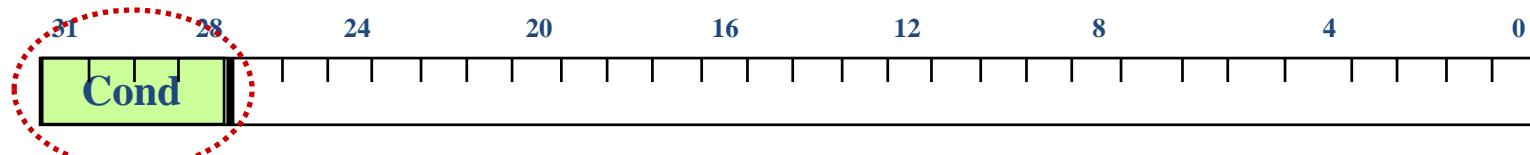
- ❖ 조건에 따라 명령어를 실행하도록 하기위해서는 적절한 조건을 접미사로 붙여주면 됨 :
 - 조건 없이 실행

ADD r0,r1,r2 ; $r0 = r1 + r2$ (ADDAL)

- Zero flag가 세트 되어 있을 때만 실행하고자 하는 경우

ADDEQ r0,r1,r2 ; If zero flag set then...
; ... $r0 = r1 + r2$

조건 필드(condition field)



Opcode [31:28]	Mnemonic extension	Interpretation	Status flag state for execution
0000	EQ	Equal / equals zero	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set / unsigned higher or same	C set
0011	CC/LO	Carry clear / unsigned lower	C clear
0100	MI	Minus / negative	N set
0101	PL	Plus / positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N equals V
1011	LT	Signed less than	N is not equal to V
1100	GT	Signed greater than	Z clear and N equals V
1101	LE	Signed less than or equal	Z set or N is not equal to V
1110	AL	Always	any
1111	NV	Never (do not use!)	none

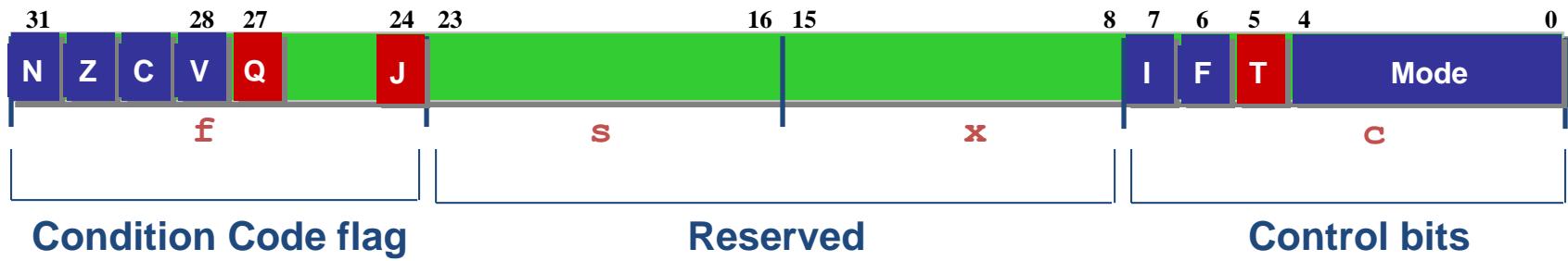
❖ Condition flag의 변경은 명령어에 “S” 접미사를 삽입

- Data processing 명령의 경우 “S” 접미사가 없으면 condition flag에 영향을 미치지 않는다.
- ADD 연산 실행 후 결과를 가지고 Condition Flag를 세트

```
ADD$ r0,r1,r2           ; r0 = r1 + r2  
                         ; ... and set flags
```

❖ Data Processing 명령 중 비교를 위한 명령은 별도로 “S” 접미사를 삽입하지 않아도 Condition Flag가 변경된다.

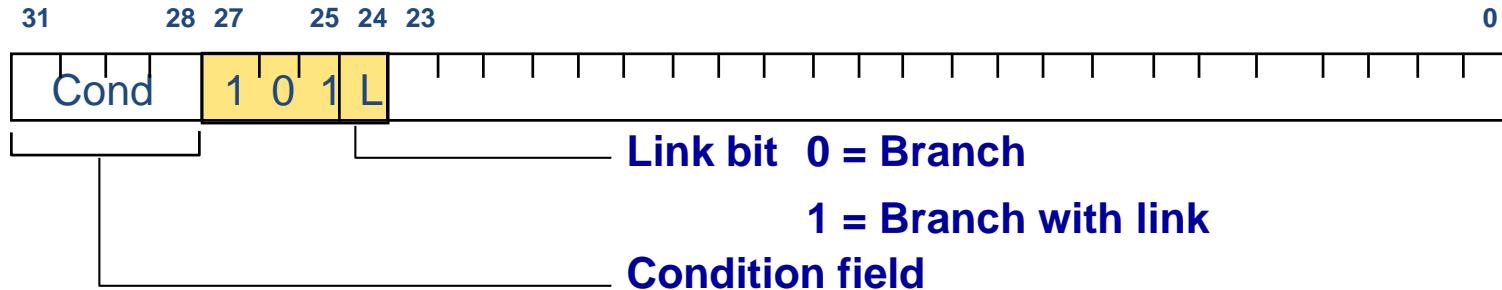
조건 플래그(Condition Flags)



Flag	논리 연산	산술 연산
Negative (N=1)	사용되지 않는다	Signed 연산에서 비트 31이 세트되어 Negative 결과 발생
Zero (Z=1)	연산 결과가 모두 0	연산 결과가 0
Carry (C=1)	Shift 동작 결과 carry 발생	연산 결과가 32 비트를 넘으면 세트
oVerflow (O=1)	사용되지 않는다	연산 결과가 31 비트를 넘어 sign bit 상실

	Instruction Type	Instruction (명령)
1	Branch, Branch with Link	B, BL
2	Data Processing 명령	ADD, ADC, SUB, SBC, RSB, RSC, AND, ORR, BIC, MOV, MVN, CMP, CMN, TST, TEQ
3	Multiply 명령	MUL, MLA, SMULL, SMLAL, SMULL, UMLAL
4	Load/Store 명령	LDR, LDRB, LDRBT, LDRH, LDRSB, LDRSH, LDRT, STR, STRB, STRBT, STRH, STRT
5	Load/Store Multiple 명령	LDM, STM
6	Swap 명령	SWP, SWPB
7	Software Interrupt(SWI) 명령	SWI
8	PSR Transfer 명령	MRS, MSR
9	Coprocessor 명령	MRC, MCR, LDC, STC
10	Branch Exchange 명령	BX
11	Undefined 명령	

명령어	B, BL	Branch, Branch Link
형식	<p>B{L}{cond} <expression> {L}은 branch with link로 R14(LR)에 PC 값을 저장하도록 한다. {cond}는 조건부 실행을 위한 condition 조건을 나타낸다. <expression>은 destination을 나타내며 offset으로 계산된다.</p>	
동작	<p>{cond} 조건에 맞으면, <expression>이 가리키는 위치로 분기하고, {L} option이 사용되면 LR에 PC 값을 저장한다.</p>	
Flag	영향 없음	
사용 예	<p>B there ; 무조건 there로 분기하라 CMP R1,#0 ; R1과 0을 비교하고, BEQ fred ; R1이 0이면 fred로 분기 BL sub+ROM ; 어드레스를 계산한 후 서브루틴 콜</p>	



❖ 분기할 주소 계산

- PC 값을 기준으로 비트 [23:0]에 해당하는 Offset이 사용된다.
- 24 비트의 Offset
 - 맨 상위 비트는 +/- sign 비트
 - 나머지 23 비트를 이용하여 2비트 왼쪽으로 shift 한 값을 사용
 - 분기 가능한 어드레스 영역 : PC +/- 32MB

❖ Branch with Link (BL) 명령

- 다음에 수행할 명령의 위치를 LR에 저장한다.
- $LR = PC - 4$

❖ Subroutine에서 Return

- LR에 저장된 주소를 PC에 옮긴다.

```
MOV pc, lr ; pc = lr
```

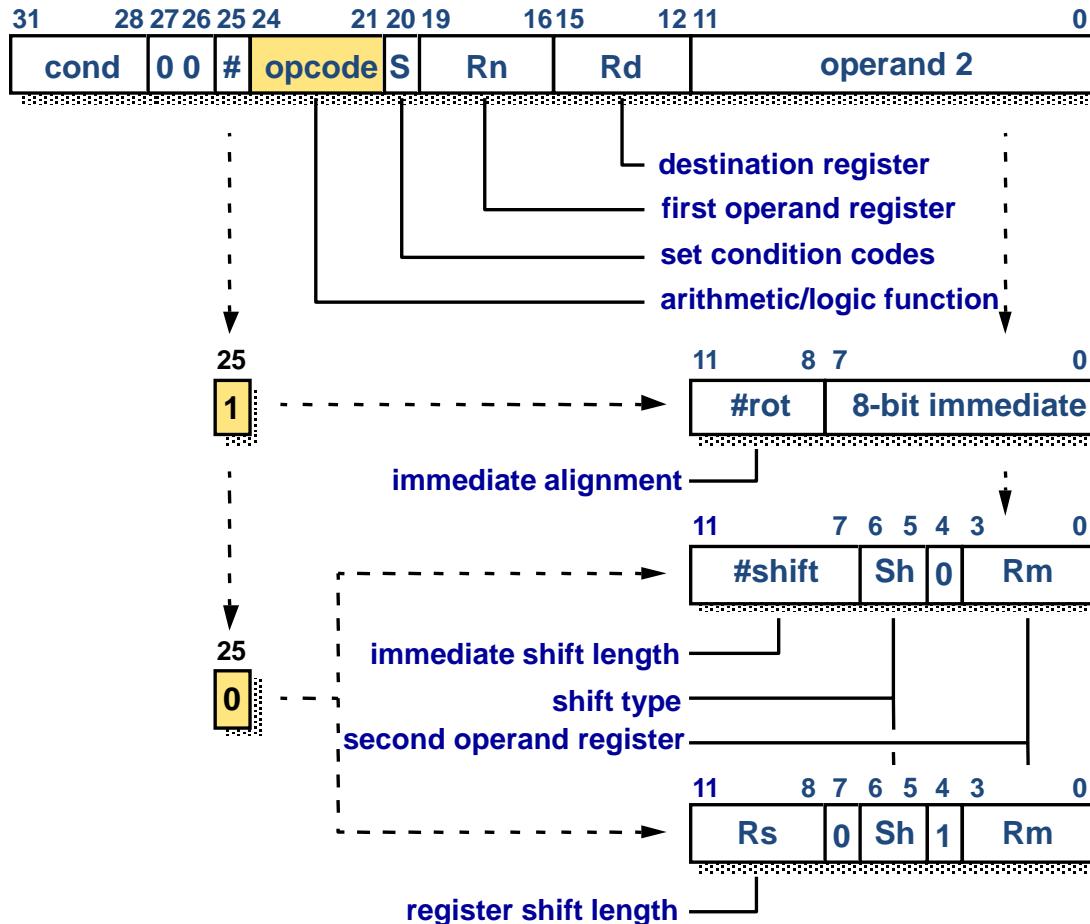
❖ 관련 명령어 종류

- Arithmetic(산술) 연산
- Logical(논리) 연산
- 레지스터간의 데이터 Move(이동) 명령
- Comparison(비교) 명령

❖ Load /Store architecture

- Data Processing 명령은 Register 내에서만 이루어지고 메모리에서 데이터를 읽거나 쓸 때는 Load 또는 Store 명령을 사용한다.

Data Processing 명령어의 구조



❖ 산술 연산 명령어

opcode	명령	동 작	설 명
0100	ADD	$Rd := Rn + Op2$	Add
0101	ADC	$Rd := Rn + Op2 + Carry$	Add with Carry
0010	SUB	$Rd := Rn - Op2$	Subtract
0110	SBC	$Rd := Rn - Op2 - 1 + Carry$	Subtract with Carry
0011	RSB	$Rd := Op2 - Rn$	Reverse subtract
0111	RSC	$Rd := Op2 - Rn - 1 + Carry$	Reverse Subtract with Carry

❖ 문법:

- <Operation> {<cond>} {S} Rd, Rn, Operand2

❖ 예제

- ADD r0, r1, r2
- SUBGT r3, r3, #1
- RSBLES r4, r5, #5

❖ 논리 연산 명령어

opcode	명령	동 작	설 명
0000	AND	Rd := Rn AND Op2	AND
1111	ORR	Rd := Rn OR Op2	OR
0001	EOR	Rd := Rn XOR Op2	Exclusive OR
1110	BIC	Rd := Rn AND (NOT Op2)	Bit Clear

❖ 문법:

- <Operation> {<cond>} {S} Rd, Rn, Operand2

❖ 예제:

- AND r0, r1, r2
- BICEQ r2, r3, #7
- EORS r1,r3,r0

- ❖ 비교 연산의 결과는 조건 플래그를 변경하는 것
 - S bit를 별도로 set 할 필요가 없다.
- ❖ 비교 연산 명령어

opcode	명령	동 작	설 명
1010	CMN	CPSR flags := Rn + Op2	Compare Negative
1011	CMP	CPSR flags := Rn-Op2	Compare
1000	TEQ	CPSR flags := Rn EOR Op2	Test bitwise equality
1001	TST	CPSR flags := Rn AND Op2	Test bits

- ❖ 문법:
 - <Operation> {<cond>} Rn, Operand2
 - Destination 레지스터가 없다.
- ❖ 예제:
 - CMP r0, r1
 - TSTEQ r2, #5

❖ 데이터 Move 명령어

opcode	명령	동 작	설 명
1101	MOV	Rd := Op2	Move register or constant
1111	MVN	Rd := 0xFFFFFFFF EOR Op2	Move Negative register

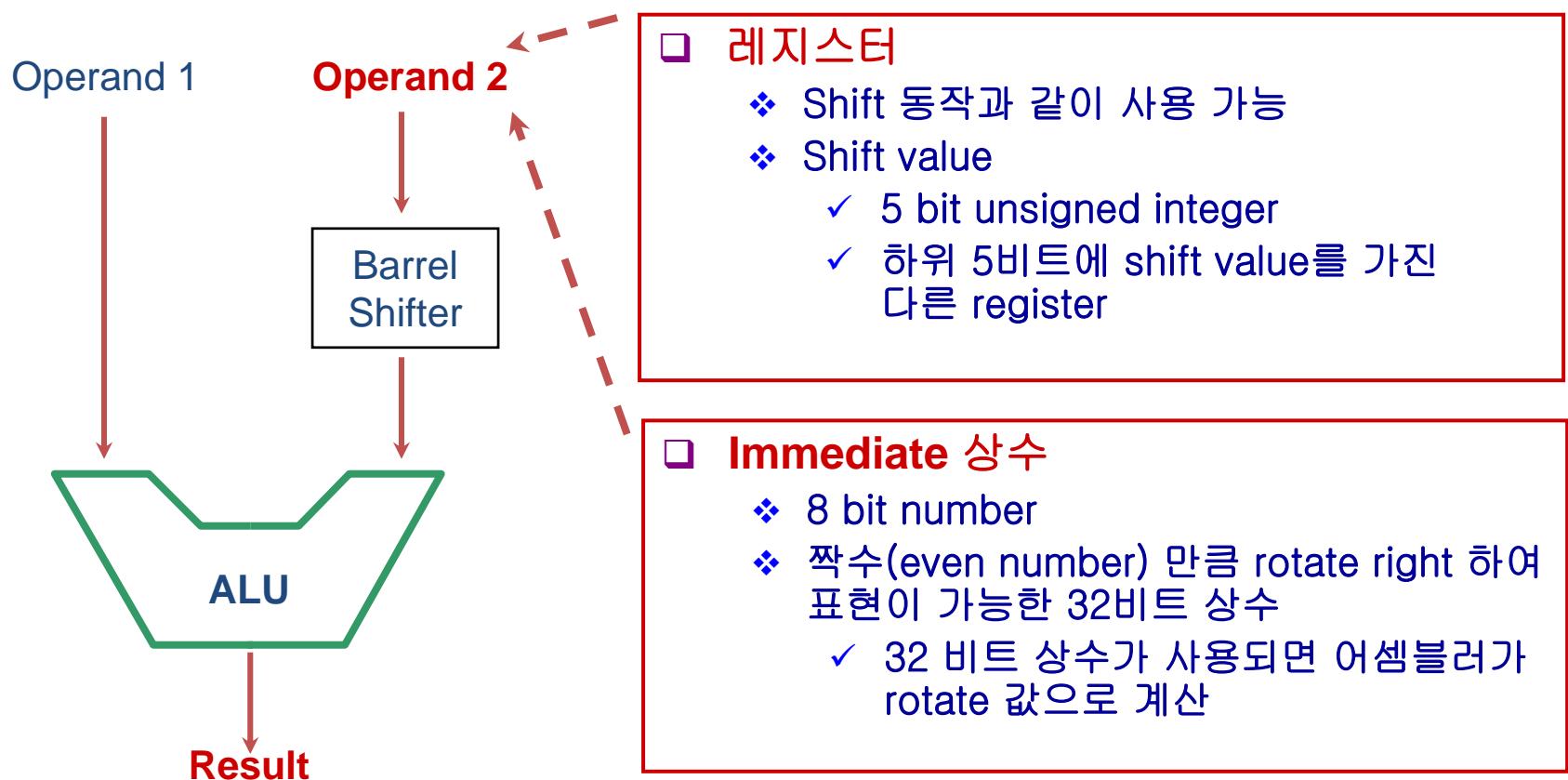
❖ 문법:

- <Operation> {<cond>} {S} Rd, Operand2
- Operand 10이 없다.

❖ 예제:

- MOV r0, r1 ; r1 -> r0
- MOVS r2, #10 ; 10 -> r2
- MVNEQ r1,#0 ; if zero flag set then 0 -> r1

Operand2와 Barrel Shifter



❖ Shifts Left

- Multiplies by powers of 2
- e.g.
 - LSL #5 = multiply by 32
 - LSL = ASL

❖ Logical Shift Right

- Divide by powers of 2 (unsigned)
- e.g.
 - LSR #5 = divide by 32

❖ Arithmetic Shift Right

- Divide by powers of 2 (signed)
 - Sign bit을 유지
 - 2's complement operations.
- e.g.
 - ASR #5 = divide by 32 (signed)



❖ Rotate Right (ROR)

- LSB가 MSB로 돌아(wrap-around)온다



❖ Rotate Right Extended (RRX)

- CPSR C flag를 33번째 비트로 사용하면서
1비트가 rotate right



❖ 레지스터가 SHIFT되는 횟수는 2 가지 방법으로 지정 가능

- 명령어에 5비트의 immediate 상수를 지정
 - Shift 동작이 1 사이클 내에 같이 처리된다.

ADD r5, r5, r3, LSL #3

- 레지스터를 사용하고, 레지스터에는 5비트의 값을 지정
 - 추가적으로 레지스터 값을 읽어 오는 사이클이 필요하다.
 - ARM에는 2개의 레지스터 read 포트만 존재한다.

ADD r5, r5, r3, LSL r2

❖ Shift 가 명시되지 않으면 default shift 적용

- LSL #0
- barrel shifter 동작이 없다.

❖ ARM의 Multiply Operation

- Multiplier를 사용하는 경우 수개의 내부 사이클이 별도로 소요 된다.
- Data Processing 명령과 Shift Operation을 사용하여 1사이클 내에 Multiply 연산 가능

ADD r0, r1, r1, LSL #2 ; $r0 = r1 * 5 \rightarrow r0 = r1 + (r1 * 4)$

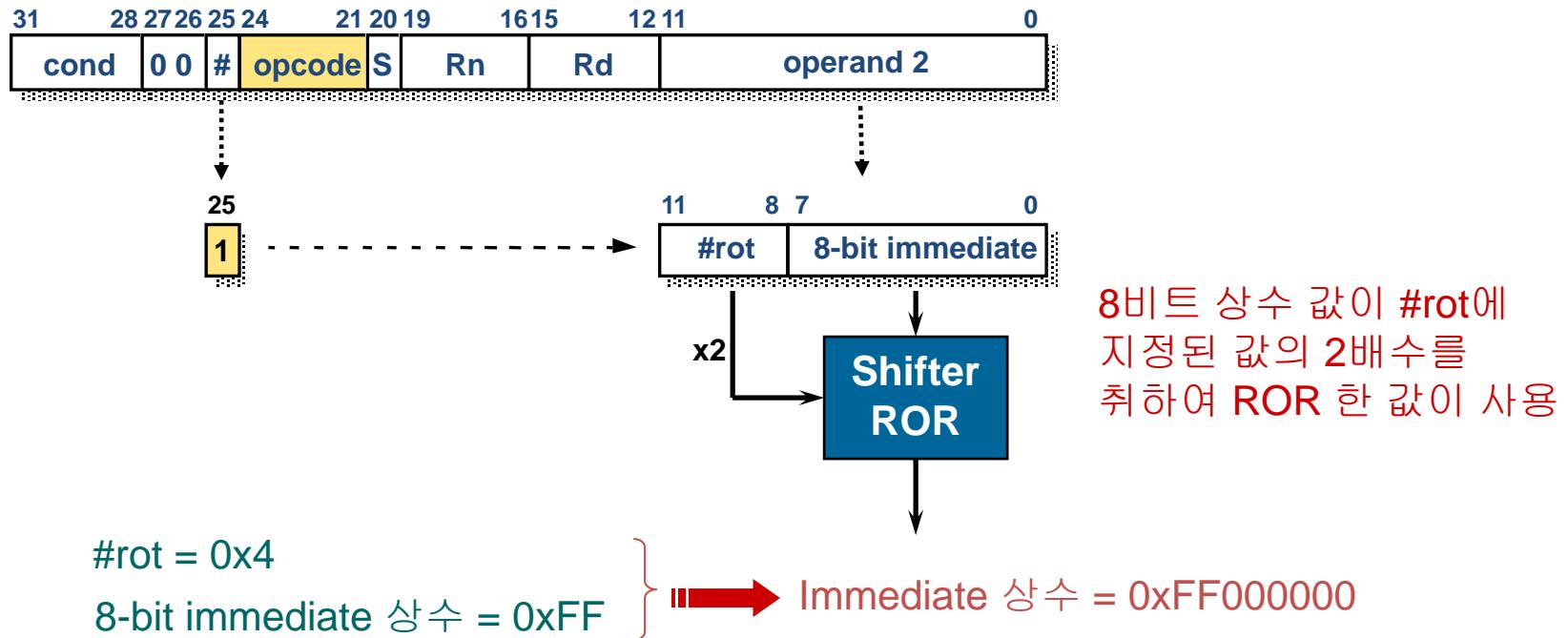
❖ ARM Divide Operation

- ARM에는 Divider가 없다.
 - 소프트웨어 라이브러리로 처리, 속도가 늦다.
- Data Processing 명령과 Shift Operation을 사용하면 Divide 연산 가능

MOV r0, r1, ASR #4 ; $r0 = r1 / 16$

❖ Operand 2로 Immediate 상수로 사용 가능

- 32비트 명령어 내에 그 값이 지정된다.
 - Immediate 상수는 32비트 상수를 가질 수 없다.



- ❖ 32비트 상수를 레지스터로 옮길 수 있는 방법 제공
 - MOV 또는 MVN 명령으로는 32비트 상수를 레지스터로 move 불가
 - Assembler에서 32비트 상수를 읽을 수 있는 방법 제공
- ❖ Data Transfer 명령을 이용한 32비트 상수 loading
 - **LDR rd, =constant**
- ❖ 어셈블러에서는 사용된 값(constant)에 따라 유효한 명령어 사용
 - 8 비트와 ROR*2로 표현 가능하면 MOV 또는 MVN으로 변환

LDR r0, =0xFF

MOV r0, #0xFF

- 32비트로만 표현이 가능하면 코드 영역 내에 상수 값(Literal pool)을 저장하고 PC-relative 한 LDR 명령으로 변환

LDR r0, =0x5555AAAA

LDR r0, [PC, #offset]

DCD 0x5555AAAA

- ❖ 실습을 통하여 Branch 와 Data Processing 명령 사용법을 익힌다.

❖ PSR(Program Status Register) Transfer 명령

- 1개의 CPSR, 5개의 SPSR의 PSR 레지스터와 ARM의 내부 레지스터 R0~R15 사이의 데이터 전송 명령
- MRS : Move PSR to Register
- MSR : Move Register to PSR

❖ CPSR 레지스터 액세스

- 모든 Privilege 모드에서 액세스 가능
 - ‘T’ 비트는 강제로 어떤 값을 write 할 수 없다.
- User 모드에서는 모든 비트를 읽을 수는 있으나 Flag field 만 write 가능

❖ SPSR 레지스터 액세스

- 각각의 동작 모드에서 제정된 SPSR 만 액세스 가능
- User 모드의 경우에는 액세스 가능한 SPSR이 없다

명령어	MRS	Move PSR to Registers
형식	<p>MRS{cond} Rd, <PSR></p> <p>{cond}는 조건부 실행을 위한 condition 조건을 나타낸다.</p> <p><PSR>은 CPSR 또는 SPSR을 말한다.</p>	
동작	<p>PSR 레지스터의 내용을 Rd에 옮긴다.</p>	
사용 예	<p>MRS r0, CPSR ; r0 := CPSR</p>	

명령어	MSR	Move Register to PSR
형식		<p>MSR{cond} <PSR[_fields]>, Rm</p> <p>{cond}는 조건부 실행을 위한 condition 조건을 나타낸다.</p> <p><PSR>은 CPSR 또는 SPSR을 말한다.</p> <p>[_fields]는 f,x,s,c의 4 가지를 말하며 같이 사용도 가능하다.</p>
동작		Rm 레지스터의 내용을 <PSR[_fields]>에 옮긴다.
사용 예		<p>MSR CPSR, r0 ; CPSR := r0</p> <p>MSR CPSR_f, r0 ; CPSR의 condition flags 비트 := r0</p> <p>MSR CPSR_fc, r1 ; CPSR의 flags, control 비트 := r0</p> <p>MSR CPSR_c, #0x80 ; CPSR의 control 비트 := 0x80</p>

명령어	MUL	32 비트 Multiply
형식	MUL{cond}{S} Rd, Rm, Rs {cond}는 조건부 실행을 위한 condition 조건을 나타낸다. {s}는 CPSR의 flags 비트의 세트 여부를 결정한다. Rd 는 Destination 레지스터 Rm 과 Rs 는 곱셈이 되는 레지스터	
동작	$Rd := Rm * Rs$ 레지스터 동작만 사용 가능하고 Immediate 상수는 사용 할 수 없다. 32비트 이상의 결과는 버린다.	
사용 예	MUL R1, R2, R3 ; R1 := R2 * R3	

32 비트 Multiply-Accumulate

명령어	MLA	32 비트 Multiply-Accumulate
형식	<p>MLA{cond}{S} Rd, Rm, Rs, Rn</p> <p>{cond}는 조건부 실행을 위한 condition 조건을 나타낸다.</p> <p>{s}는 CPSR의 flags 비트의 세트 여부를 결정한다.</p> <p>Rd는 Destination 레지스터</p> <p>Rm 과 Rs는 곱셈이 되는 레지스터, Rn은 더해지는 레지스터</p>	
동작	$Rd := Rm * Rs + Rn$	
사용 예	MLA R1, R2, R3, R4 ; R1 := R2 * R3 + R4	

64 비트 Multiply

명령어	MULL	64 비트 Multiply
형식	<p>[U S]MULL{cond}{S} RdLo, RdHi, Rm, Rs</p> <p>U: unsigned, S:signed</p> <p>{cond}는 조건부 실행을 위한 condition 조건을 나타낸다.</p> <p>{s}는 CPSR의 flags 비트의 세트 여부를 결정한다.</p> <p>RdLo, RdHi 는 64비트 결과가 저장 되는 레지스터</p> <p>Rm 과 Rs는 곱셈이 되는 레지스터</p>	
동작	<p>RdHi,RdLo := Rm * Rs</p> <p>64비트의 결과를 얻는다.</p>	
사용 예	<p>MULL R1, R2, R3, R4 ; R2,R1 := R3 * R4</p>	

64 비트 Multiply-Accumulate

명령어	MLAL	64 비트 Multiply-Accumulate
형식	<p>[U S]MLAL{cond}{S} RdLo, RdHi, Rm, Rs</p> <p>U: unsigned, S:signed</p> <p>{cond}는 조건부 실행을 위한 condition 조건을 나타낸다.</p> <p>{S}는 CPSR의 flags 비트의 세트 여부를 결정한다.</p> <p>RdLo, RdHi 는 64비트 결과가 저장 되는 레지스터</p> <p>Rm 과 Rs는 곱셈이 되는 레지스터</p>	
동작	$RdHi, RdLo := Rm * Rs + (RdHi, RdLo)$	
사용 예	MLAL R1, R2, R3, R4 ; R2,R1 := R3 * R4 + (R2,R1)	

❖ ARM의 Load / Store Architecture

- memory to memory 데이터 처리 명령을 지원하지 않음
- 처리하고자 하는 데이터는 무조건 레지스터로 이동해야 함
- 처리 절차
 - ①데이터를 메모리에서 레지스터로 이동
 - ②레지스터에 읽혀진 값을 가지고 처리
 - ③연산 결과를 메모리에 저장

❖ ARM의 메모리 액세스 명령

- Single register data transfer (LDR / STR).
- Block data transfer (LDM/STM).
- Single Data Swap (SWP).

❖ Single Register Data Transfer 명령

액세스 단위	Load 명령	Store 명령
Word	LDR	STR
Byte	LDRB	STRB
Halfword	LDRH	STRH
Signed byte	LDRSB	
Signed halfword	LDRSH	

- ❖ 모든 명령어는 STR / LDR 다음에 적절한 Condition 코드를 삽입하여 조건부 실행 가능
- e.g. LDREQB

명령어	LDR	Load
형식	<p>LDR{cond}{size}{T} Rd, <Address></p> <p>{cond}는 조건부 실행을 위한 condition 조건을 나타낸다.</p> <p>{size}는 전송되는 데이터 크기를 나타내며, B(byte), H(halfword), SB(signed byte) 와 SH(signed halfword)가 있다.</p> <p>{T}는 post-indexed 모드에서 non-privilege 모드로 데이터가 전송 된다.</p> <p>Rd는 읽어온 메모리 값이 write 되는 레지스터</p> <p><Address> 부분은 베이스 레지스터와 Offset으로 구성된다.</p>	
동작	<p><Address>가 나타내는 위치의 데이터를 {size}만큼 읽어서 Rd에 저장한다.</p>	
사용 예	<p>LDR R1, [R2,R4] ; R2+R4 위치의 데이터를 R1에 저장</p> <p>LDREQB R1, [R6,#5] ; 조건이 EQ이면 R6+5 위치의 데이터를 바이트 만큼 읽어 R1에 전달</p>	

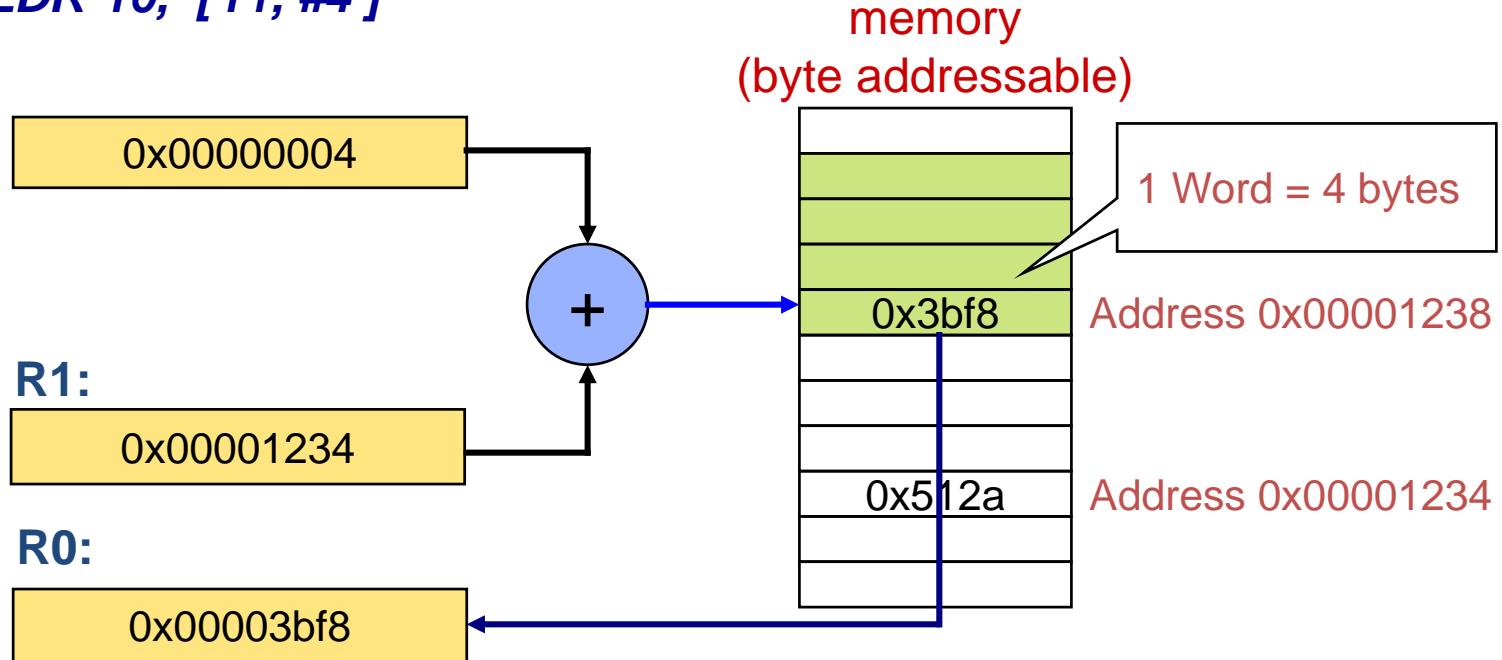
명령어	STR	Store
형식	<p>STR{cond}{size}{T} Rd, <Address></p> <p>{cond}는 조건부 실행을 위한 condition 조건을 나타낸다.</p> <p>{size}는 B(byte), H(halfword) 가 있다.</p> <p>{T}는 post-indexed 모드에서 non-privilege 모드로 데이터가 전송 된다.</p> <p>Rd는 write 할 데이터가 들어있는 레지스터</p> <p><Address> 부분은 베이스 레지스터와 Offset으로 구성된다.</p>	
동작	<Address> 위치에 Rd를 저장한다.	
사용 예	<p>STR R1, [R2,R4] ; R2+R4 위치에 R1의 내용을 저장</p> <p>STREQB R1, [R6,#5] ; 조건이 EQ이면 R6+5 위치에 R1의 내용을 저장.</p>	

명령어	<Address> fields	Effective 어드레스
Pre-indexed 방법 [Rn, <offset>]	[Rn, +/- expression]{!}	Rn +/- expression
	[Rn, +/- Rm]{!}	Rn +/- Rm
	[Rn, +/- Rm, shift cnt]{!}	Rn +/- (Rm shift cnt)
Post-indexed 방법 [Rn], <offset>	[Rn], +/- expression	Rn
	[Rn], +/- Rm	Rn
	[Rn], +/- Rm, shift cnt	Rn
<p>Rn: base 레지스터 expression: -4095 ~ +4095 범위의 12비트 Immediate 상수 shift: LSL, LSR, ASR, ROR 와 RRX의 shift 동작 cnt: 1~31 범위의 4비트 값 !: auto-update 또는 write-back, pre-indexed 방식의 경우 base 레지스터를 update * Halfword 또는 signed halfword인 경우에는 8비트 Immediate 상수 또는 shift 동작 없는 레지스터가 사용이 가능하다.</p>		

- ❖ 레지스터 뱅크에서 B 버스를 통해서 Barrel shifter를 경유하여 ALU에 전달
- ❖ Offset으로 사용될 수 있는 표현
 - Unsigned 12비트 immediate 상수 (0 ~ 4095 바이트)
 - 레지스터
 - Immediate 값으로 shift operation 지정 가능
- ❖ ‘+’ 또는 ‘-’ 연산을 함께 사용 가능

- ❖ Base 레지스터(Rn)과 <offset>으로 주소 계산 후 데이터 전송
 - 데이터 전송 이후에도 Rn의 값은 별도 지정이 없으면 변하지 않는다

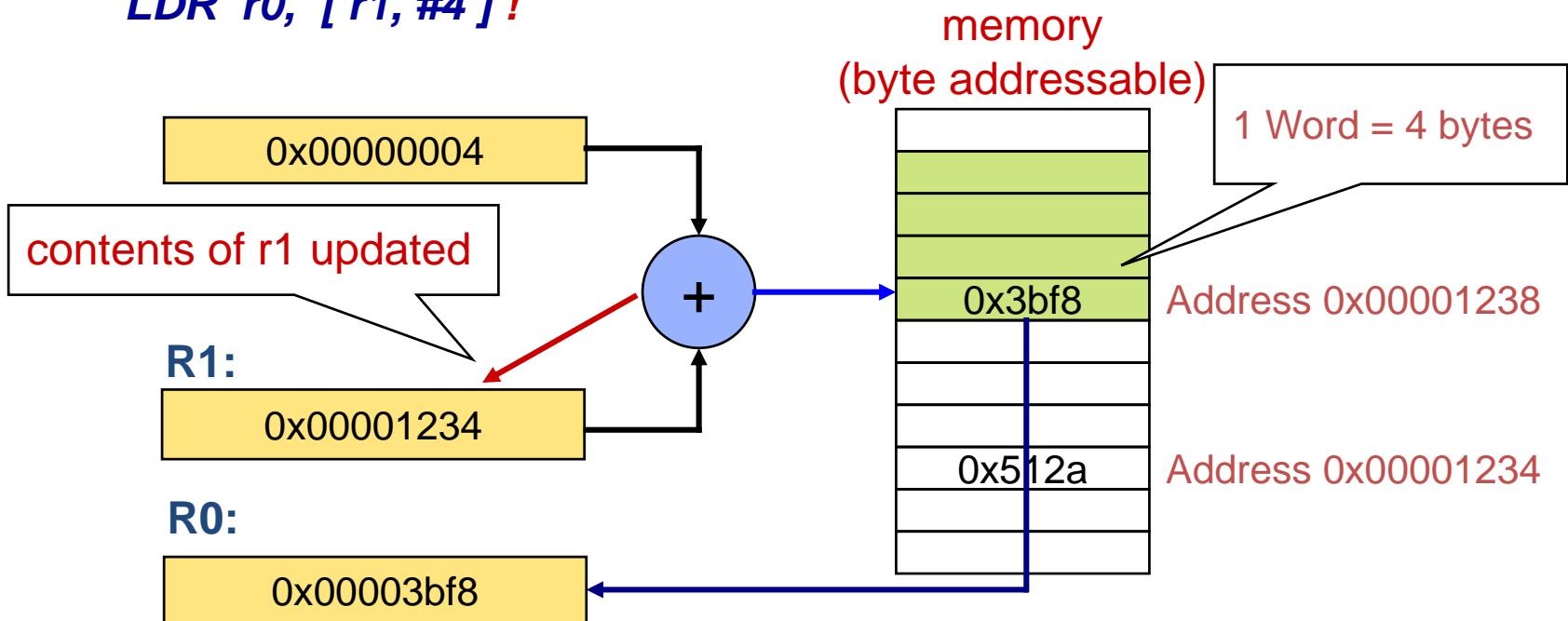
LDR r0, [r1, #4]



Pre-indexed 방식 과 Auto update

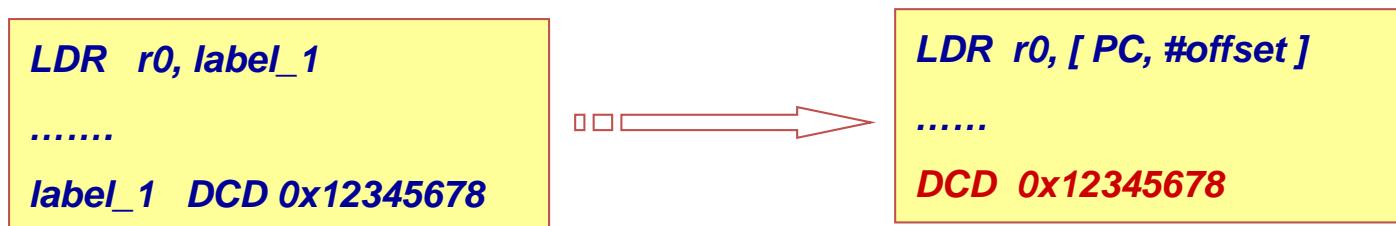
- ❖ Pre-indexed 방식을 사용하여 참조 후 Base 레지스터 Rn 갱신

LDR r0, [r1, #4] !



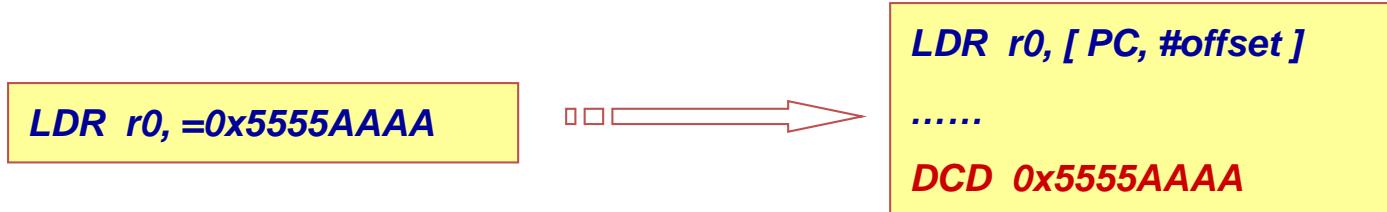
❖ LABEL 지정에 의한 어드레스 지정

- 어셈블리어에서 LABEL을 지정하면 어셈블러가 [PC+LABEL]형태의 주소로 변환하여 참조



❖ literal pool을 사용한 32 비트 데이터의 Load

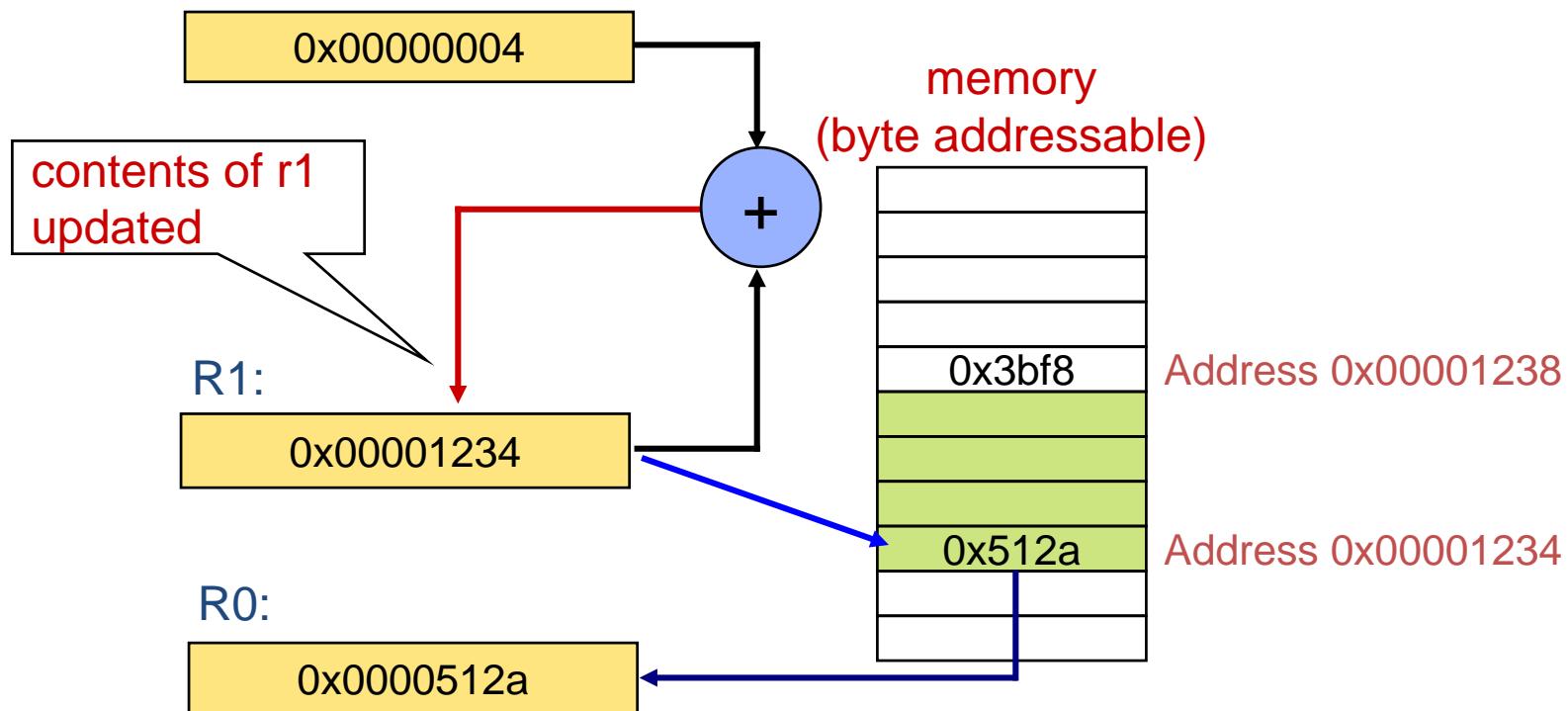
- 어셈블러가 코드 영역 내에 데이터 저장 후 [PC+LABEL]형태의 주소로 변환하여 참조



Post-indexed 어드레스 지정 방식

- Base 레지스터(Rn)가 지정하는 주소에 데이터의 전송 후 Rn 값과 <Offset>의 연산 결과로 Rn 갱신

LDR r0, [r1], #4



❖ Block Data Transfer 명령

- 하나의 명령으로 메모리와 프로세서 레지스터 사이에 여러 개의 데이터를 옮기는 명령
- Load 명령인 LDM 과, Store 명령인 STM 명령이 있다.

❖ Block Data Transfer 명령의 응용

- Memory copy, memory move 등
- Stack operation
 - ARM에는 별도의 stack 관련 명령이 없다
 - LDM/STM 이용하여 pop 또는 push 동작 구현

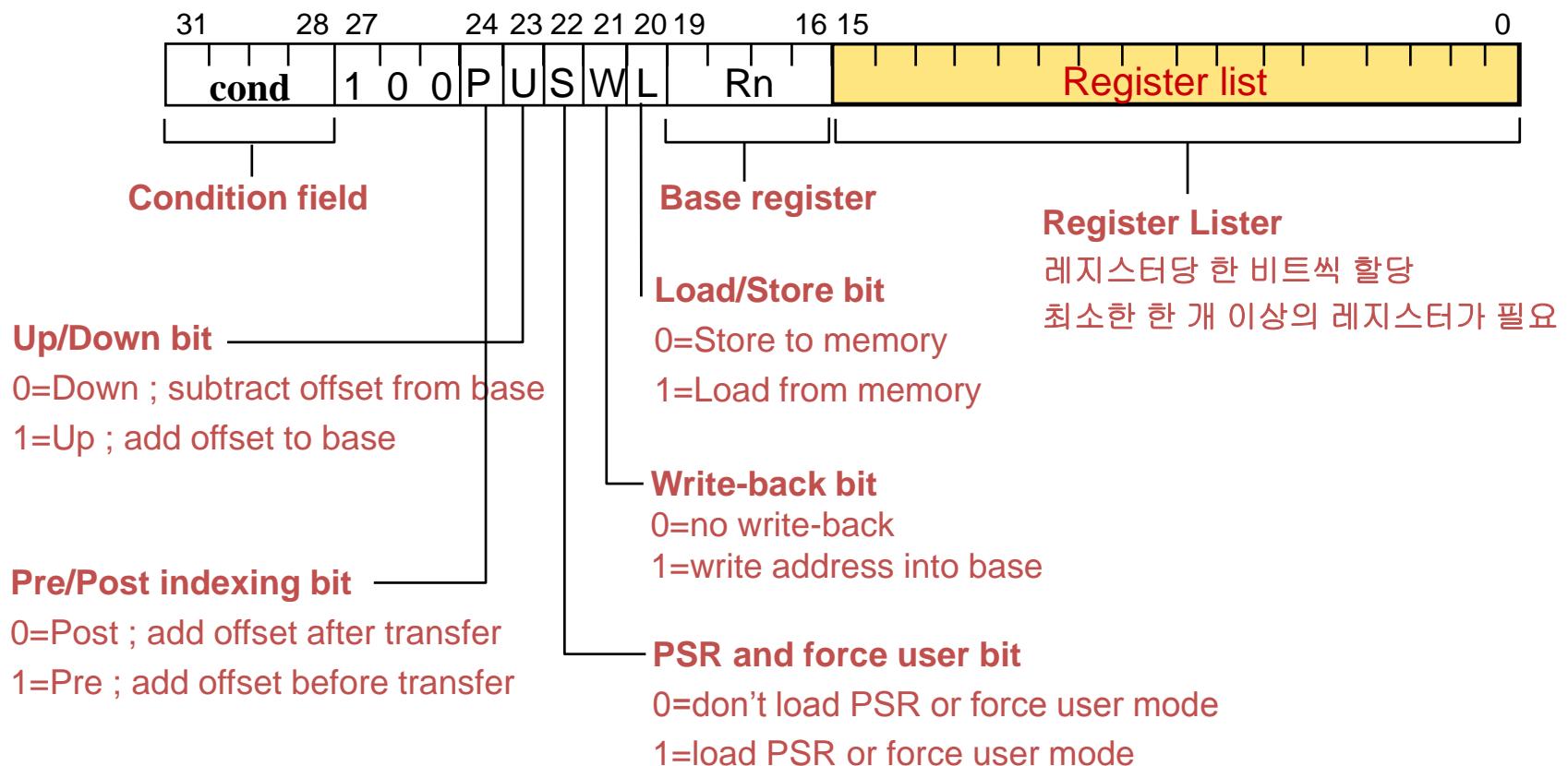
Load Multiple(LDM) 명령

명령어	LDM	Load Multiple
형식	<p>LDM{cond}<addressing mode> Rn{!}, <register_list>{^}</p> <p>{cond}는 조건부 실행을 위한 condition 조건을 나타낸다.</p> <p>{address mode}는 어드레스의 생성 방법을 정의 한다.</p> <p>Rn은 base 레지스터를 나타낸다.</p> <p>{!}는 데이터 전송 후 base 레지스터를 auto-update</p> <p><register_list> 읽어온 데이터를 저장할 레지스터 지정</p> <p>{^}는 privilege 모드에서 stack 동작 할 때 PC와 CPSR 복원</p>	
동작	<p>Base 레지스터 Rn이 지정한 번지에서 여러 개의 데이터를 읽어</p> <p><register_list>로 읽어 들이는 명령</p>	
사용 예	<p>LDMIA R0, {R1,R2,R3} ; R0의 위치에서 데이터를 읽어 R1,R2,R3에 저장.</p> <p>LDMFD sp, {pc}^ ; sp(stack point)에서 return될 위치(PC)를 읽어</p> <p>PC를 update하고 CPSR을 SPSR로부터 복사</p>	

Store Multiple(STM) 명령

명령어	STM	Store Multiple
형식	<p>STM{cond}<addressing mode> Rn{!}, <register_list></p> <p>{cond}는 조건부 실행을 위한 condition 조건을 나타낸다.</p> <p>{address mode}는 어드레스의 생성 방법을 정의 한다.</p> <p>Rn은 base 레지스터를 나타낸다.</p> <p>{!}는 데이터 전송 후 base 레지스터를 auto-update</p> <p><register_list> 저장할 데이터를 가지고 있는 레지스터 지정</p>	
동작	<p>Base 레지스터 Rn이 지정한 번지에 <register_list>에 있는 여러 개의 데이터를 저장하는 명령</p>	
사용 예	<p>STMIA R0, {R1,R2,R3} ; R1,R2,R3의 데이터를 R0의 저장</p>	

Block Data Transfer 명령의 구조



- ❖ <register_list>에서 사용 가능한 레지스터
 - R0에서 R15(PC)까지 최대 16개
- ❖ 연속된 레지스터 표현
 - {r0-r5}와 같이 “-”로 표현 가능
- ❖ <register_list>의 순서 지정
 - 항상 low order의 register에서 high order 순으로 지정

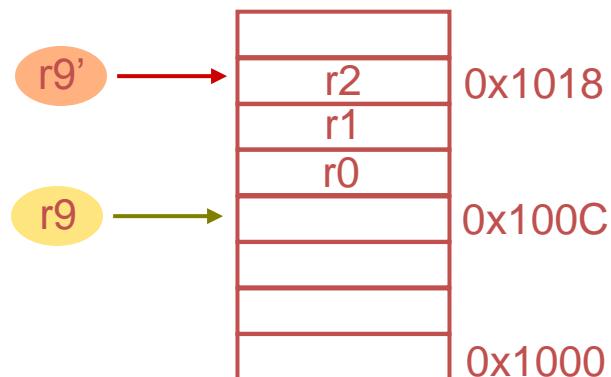


Addressing Mode	키워드(표현방식)		유효 어드레스 계산
	데이터	스택	
Pre-increment Load	LDMIB	LDMED	Increment before load
Post-increment Load	LDMIA	LDMFD	Increment after load
Pre-decrement Load	LDMDB	LDMEA	Decrement before load
Post-decrement Load	LDMDA	LDMFA	Decrement after load
Pre-increment Store	STMIB	STMFA	Increment before store
Post-increment Store	STMIA	STMEA	Increment after store
Pre-decrement Store	STMDB	STMFD	Decrement before store
Post-decrement Store	STMDA	STMED	Decrement after store

STMIB r9!, {r0,r1,r2}

Base 레지스터(r9) = 0x100C

❖ 어드레스 증가 후 데이터 저장
(Increment Before)

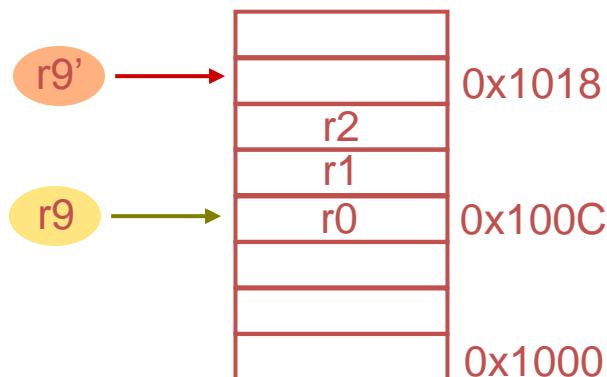


- ① r9 -> 0x1010 증가 후 r0 저장
- ② r9 -> 0x1014 증가 후 r1 저장
- ③ r9 -> 0x1018 증가 후 r2 저장
- ④ {}, auto-update 옵션이 있으면
r9 값을 0x1018로 변경

STMIA r9!, {r0,r1,r2}

Base 레지스터(r9) = 0x100C

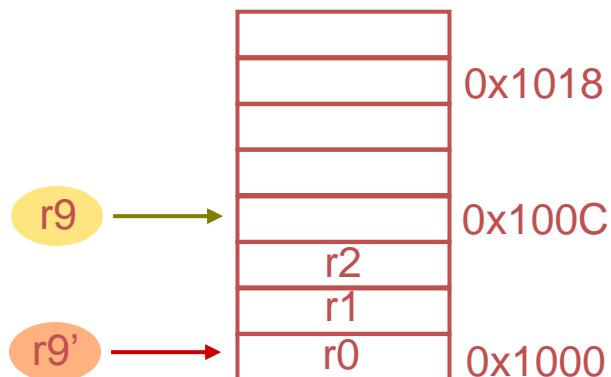
- ❖ 데이터 저장 후 어드레스 증가
(Increment After)



- ① 0x100c에 r0 저장 후 r9-> 0x1010 증가
- ② 0x1010에 r1 저장 후 r9-> 0x1014 증가
- ③ 0x1014에 r2 저장 후 r9-> 0x1018 증가
- ④ {}, auto-update 옵션이 있으면
r9 값을 0x1018로 변경

STMDB r9!, {r0,r1,r2}

Base 레지스터(r9) = 0x100C



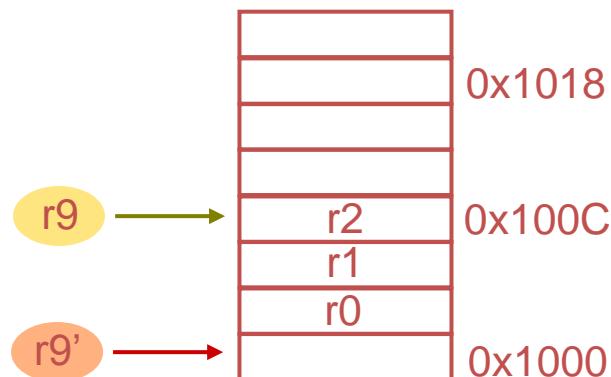
❖ 어드레스를 <register_list> 개수 만큼
감소해 놓고, 어드레스를 증가하면서
데이터 저장

(Decrement Before)

- ① 어드레스를 0x1000로 감소
- ② 0x1000에 r0 저장 후 어드레스 증가
- ③ 0x1004에 r1 저장 후 어드레스 증가
- ④ 0x1008에 r2 저장 후 어드레스 증가
- ⑤ {}, auto-update 옵션이 있으면
r9 값을 0x1000로 변경

STMIA r9!, {r0,r1,r2}

Base 레지스터(r9) = 0x100C



❖ 어드레스를 <register_list> 개수 만큼
감소해 놓고, 어드레스를 증가하면서
데이터 저장

(Decrement After)

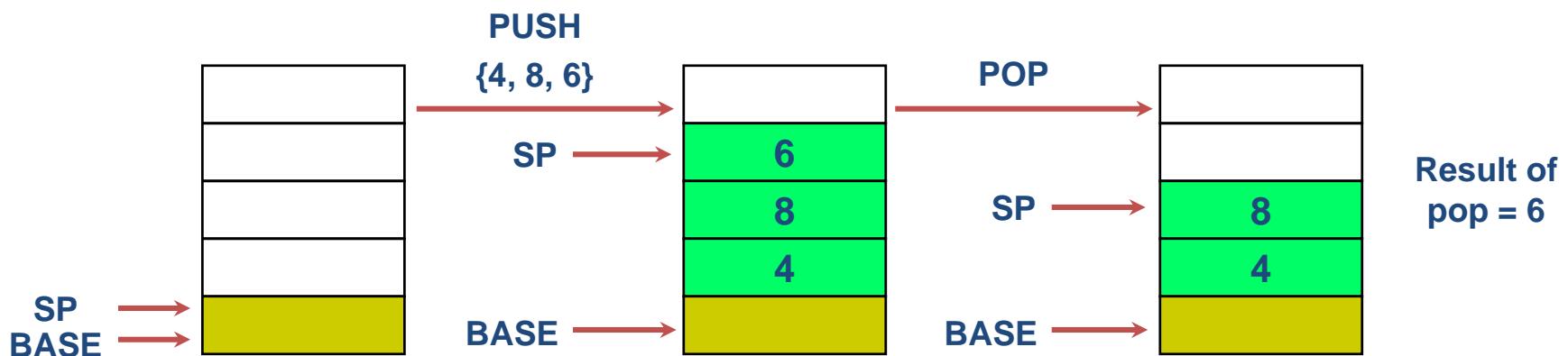
- ① 어드레스를 0x1000로 감소
- ② 어드레스 증가 후 0x1004에 r0 저장
- ③ 어드레스 증가 후 0x1008에 r1 저장
- ④ 어드레스 증가 후 0x100C에 r2 저장
- ⑤ {}, auto-update 옵션이 있으면
r9 값을 0x1000로 변경

❖ 스택 동작

- 새로운 데이터를 “PUSH”를 통해 “top”위치에 삽입하고, “POP”을 통해 가장 최근에 삽입된 데이터를 꺼내는 자료구조 형태.

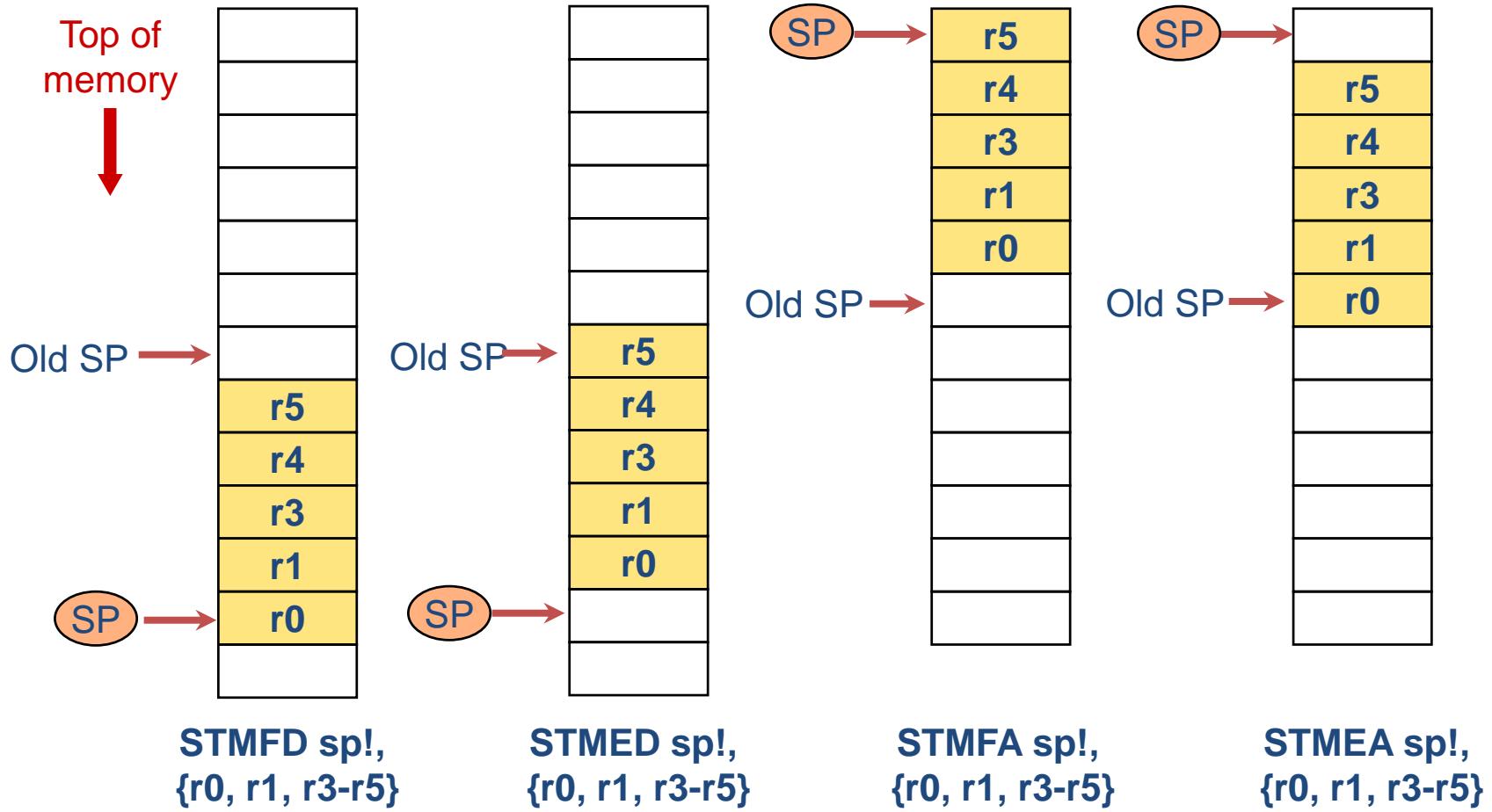
❖ 스택의 위치 지정

- Base pointer : Stack의 bottom 위치를 지정
- Stack pointer : Stack의 top 위치를 지정



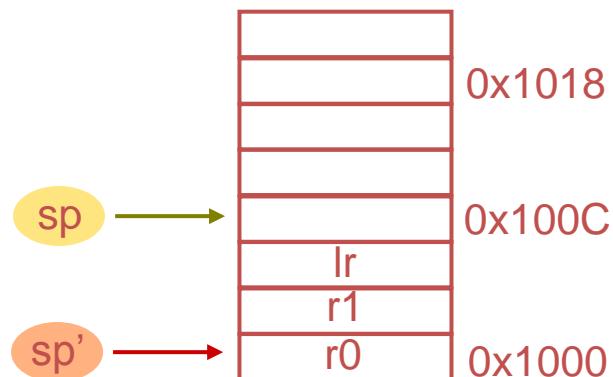
- ❖ Stack pointer가 나타내는 위치에 따라 Stack의 형태 지정
 - Full stack : Stack pointer가 Stack의 마지막 주소를 지정
 - Empty stack : Stack pointer가 Stack의 다음 주소를 지정
- ❖ Stack의 Type
 - Full Descending stack
 - Stack pointer가 stack의 top을 지정하고 push가 필요하면 어드레스 감소하면서 사용
 - ARM compiler의 경우는 항상 이 방법을 사용한다.
 - STMFD / LDMFD
 - Full Ascending stack
 - STMFA / LDMFA
 - Empty Descending stack
 - STMED / LD MED
 - Empty Ascending stack
 - STMEA / LDMEA

스택 type에 따른 포인트 변화



STMFD sp!, {r0-r1,lr}

Base 레지스터(sp) = 0x100C



❖ Stack에 context 정보 저장

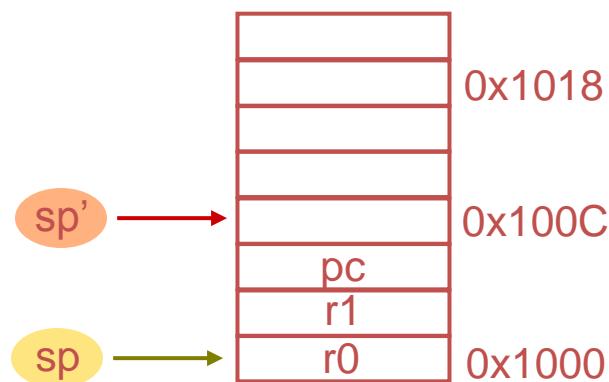
- ① 어드레스를 0x1000로 감소
- ② 레지스터 값 저장 후 어드레스 증가
- ③ 링크 레지스터 저장 후 어드레스 증가
- ④ Stack의 위치를 0x1000로 변경

LDMFD sp!, {r0-r1,pc}

Base 레지스터(sp) = 0x1000

❖ Stack에서 context 정보를 읽는다.

- ① 레지스터 값을 읽은 후 어드레스 증가
- ② 링크 레지스터(lr) 값을 읽어 프로그램 카운터(pc)에 저장
- ③ Stack의 위치를 0x100C로 변경



- ❖ 스택의 용도 중 하나는 서브루틴을 위한 일시적인 레지스터 저장소를 제공하는 것.
- ❖ 서브루틴에서 사용되는 데이터를 스택에 push하고, caller 함수로 return 하기 전에 pop 을 통해 원래의 정보로 환원시키는 데 사용:

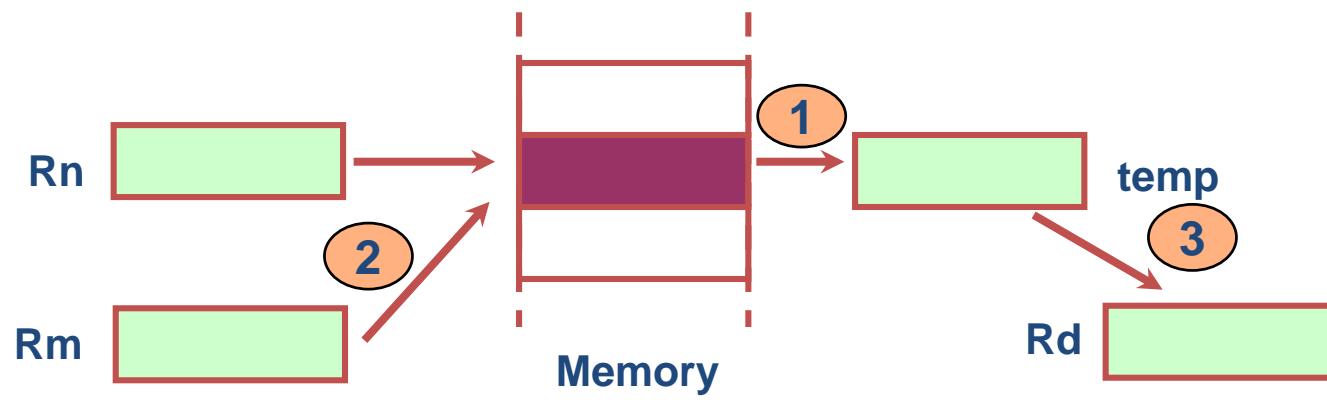
STMFD sp!,{r4-r12, lr}	; stack all registers
.....	; and the return address
.....	
LDMFD sp!,{r4-r12, pc}	; load all the registers
	; and return automatically

- ❖ privilege 모드에서 LDM을 사용하여 Pop을 할 때 ‘S’ bit set 옵션인 ‘^’ 가 레지스터 리스트에 있으면 SPSR이 CPSR로 복사 된다.

명령어	SWP	Swapping Data between register and memory
형식		<p>SWP{cond}{B} Rd,Rm,[Rn]</p> <p>{cond}는 조건부 실행을 위한 condition 조건을 나타낸다.</p> <p>{B}는 byte 단위의 데이터 교환을 나타낸다.</p> <p>Rd는 Destination 레지스터로 메모리의 내용이 기록된다.</p> <p>Rn은 base 레지스터를 나타낸다.</p> <p>Rm은 메모리에 기록될 데이터가 저장된 레지스터이다.</p>
동작		Rn이 지정하는 위치의 데이터를 읽어 Rd에 저장하고, Rm의 데이터를 메모리에 기록 한다.
사용 예		SWP r0,r1,[r2] ; r2가 지정하는 word 어드레스에서 데이터를 읽어 r0에 기록하고, r1을 r2의 어드레스에 저장한다.

Swap 명령의 동작

1. Rn이 지정하는 메모리의 내용을 임시 버퍼에 저장
2. Rm에 있는 값을 메모리에 write
3. 임시 버퍼의 내용을 Rd로 load



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

실습 2 : Memory Copy 실습

mds ACADEMY

- ❖ 메모리 액세스 명령을 이용한 Memory Copy 구현 실습

Software Interrupt(SWI) 명령

명령어	SWI	Software Interrupt
형식	<p>SWI{cond} <expression> {cond}는 조건부 실행을 위한 condition 조건을 나타낸다. <expression>은 SWI number를 기입한다.</p>	
동작	<p><expression>에 지시한 번호로 소프트웨어 인터럽트를 호출한다.</p>	
사용 예	<p>SWI 0x23 ; 0x23번 소프트웨어 인터럽트 호출</p>	



❖ SWI 명령의 응용

- 프로그램 제어 목적으로 사용
- OS의 시스템 콜 구현
 - User 모드에서 SWI의 호출에 의해 Supervisor 모드로 전환 가능
- 디버깅 등

❖ SWI가 호출 되면

- SWI Exception 발생
- SPSR_svc에 CPSR 저장
- CPSR 변경
- LR_svc에 PC 값 저장
- PC는 0x08 번지(SWI Vector)로 분기
- SWI 핸들러에서 호출된 SWI 번호에 맞는 동작 실행

- ❖ ARM은 16개의 coprocessor 지원
 - 기능과 명령의 확장을 위해
- ❖ Coprocessor 관련 명령
 - Coprocessor 데이터 처리 명령
 - CDP 명령
 - Coprocessor의 데이터 처리 초기화
 - Coprocessor와 ARM 사이 레지스터 내용 전송 명령
 - MRC 명령
 - Coprocessor 레지스터 내용을 ARM 레지스터로 전송
 - MCR 명령
 - ARM 레지스터의 내용을 Coprocessor 레지스터로 전송
 - Coprocessor와 외부 메모리 사이 데이터 전송 명령
 - LDC 명령
 - 메모리에서 coprocessor로 데이터 load
 - STC 명령
 - Coprocessor에서 메모리에 데이터 store

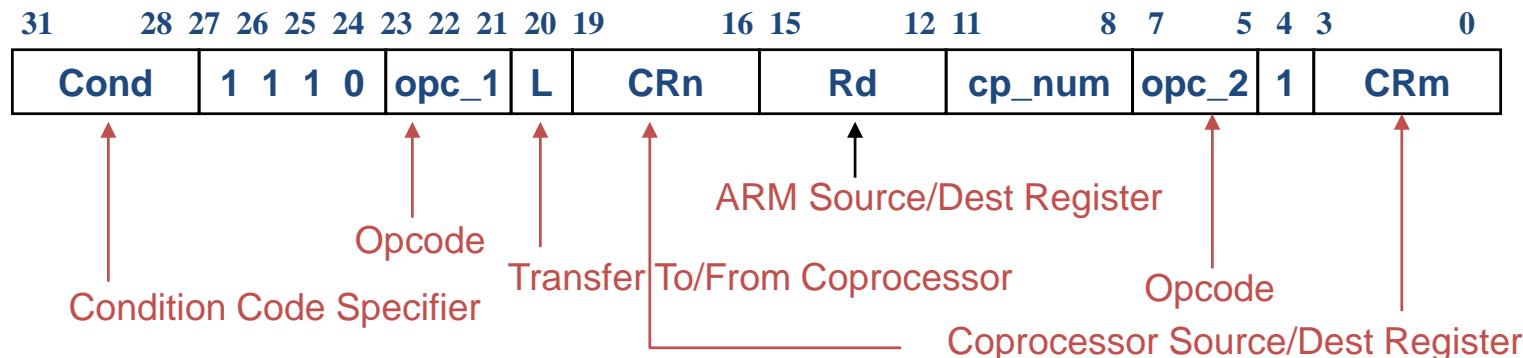
Coprocessor Register Transfer 명령

❖ Coprocessor와 ARM 사이 레지스터 내용 전송 명령

- MRC : Move to Register from Coprocessor
 - Coprocessor 레지스터 내용을 ARM 레지스터로 전송
 - MCR : Move to Coprocessor from Register
 - ARM 레지스터의 내용을 Coprocessor 레지스터로 전송

무법

<MRC | MCR>{<cond>}<cp_num>, <opc_1>, Rd, CRn, CRm, <opc_2>



❖ 프로세서의 state

- ARM state : 프로세서가 32비트 ARM 명령을 처리 상태
- Thumb state : 프로세서가 16비트 Thumb 명령을 처리 상태

명령어	BX	Branch and Exchange
형식	BX{cond} Rn {cond}는 조건부 실행을 위한 condition 조건을 나타낸다. Rn은 R0~R15까지의 레지스터가 사용될 수 있다. 비트 0에는 ARM state 로 변환 되는지 아니면 Thumb state 로 변환 되는지를 나타낸다.	
동작	Rn이 지정하는 어드레스로 분기하고 state 를 변환한다.	
사용 예	BX r0+1 ; r0 가 지정하는 위치로 분기하고, Thumb state 로 변환 BX r0 ; r0 가 지정하는 위치로 분기하고, ARM state 로 변환, 이때 비트 0은 0이다.	

- ❖ Branch 명령에서 PC 값을 기준으로 분기 가능한 어드레스 범위는 ?

- ❖ Data processing 명령에서 Operand 2가 레지스터로 사용되는 경우 ALU로 전달되는 경로는 어디인가 ?

- ❖ 다음 중 Data Processing 명령에서 Operand 2가 Immediate 상수로 사용이 가능한 상수는 ?
1) 0x12345678 2) 0x0000ffff 3) 0x0001f800 4) 0xffffffffffff

- ❖ Multiply 명령을 사용하지 말고 다음의 각 동작을 어셈블리어로 작성하라.
1) $r0 = r1 * 3$
2) $r0 = r1 / 16$

- ❖ Barrel shifter에서 지원 가능한 동작은 ?

- ❖ 다음 명령은 무엇을 의미 하는가 ?
 - 1) LDR r0, [r1, #8] !
 - 2) LDR r0, [r1], #8
 - 3) LDR r0, =0x12345678
- ❖ LDR/STR 명령의 어드레스 지정 방법에서 pre-indexed 와 post-indexed 방법의 차이는 무엇인가 ?
- ❖ LDM/STM의 4가지 주소 지정 방식에 대하여 설명하고 4가지의 경우에 대해서 데이터를 전송할 때 사용되는 어셈블러 키워드는 무엇인가 ?

Mnemonic	Instruction	Action
ADC	Add with carry	$Rd = Rn + Op2 + \text{Carry}$
ADD	Add	$Rd = Rn + Op2$
AND	AND	$Rd = Rn \text{ AND } Op2$
B	Branch	$R15 = \text{address}$
BIC	Bit Clear	$Rd = Rn \text{ AND NOT } Op2$
BL	Branch with Link	$R14 = R15, R15 = \text{address}$
CDP	Coprocessor Data Processing	(Coprocessor-specific)
CMN	Compare Negative	CPSR flags: $= Rn + Op2$
CMP	Compare	CPSR flags: $= Rn - Op2$
EOR	Exclusive OR	$Rd = (Rn \text{ AND NOT } Op2) \text{ OR } (Op2 \text{ AND NOT } Rn)$

Mnemonic	Instruction	Action
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
MRC	Move from coprocessor register to CPU register	Rn: = cRn {<op>cRm}
MRS	Move PSR status/flags to register	Rn: = PSR

Mnemonic	Instruction	Action
MSR	Move register to PSR status/flags	PSR: = Rm
MUL	Multiply	Rd: = Rm * Rs
MVN	Move negative register	Rd: = 0xFFFFFFFF 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)

Mnemonic	Instruction	Action
STR	Store register to memory	$\langle\text{address}\rangle := \text{Rd}$
SUB	Subtract	$\text{Rd} := \text{Rn} - \text{Op2}$
SWI	Software Interrupt	OS call
SWP	Swap register with Memory	$\text{Rd} := [\text{Rn}], [\text{Rn}] := \text{Rm}$
TEQ	Test bitwise equality	CPSR flags: = Rn EOR Op2
TST	Test bits	CPSR flags: = Rn AND Op2

1. ARM 프로세서의 명령어
2. 32 비트 ARM 명령어
3. 16 비트 Thumb 명령어
4. v5TE에 추가된 명령어

❖ 코드 사이즈 감소

- 컴파일 완료 후 바이너리 이미지의 크기가 ARM보다 65% 감소
- Flash 메모리와 같은 저장 장치의 크기를 줄여 제품 단가 감소

❖ 16비트 메모리 인터페이스에서 향상된 성능 지원

- 명령어가 16비트로 구성되면 16비트 메모리 인터페이스에서 한번만 fetch를 하므로 성능 향상
 - 32비트 ARM 명령은 2번에 걸쳐 fetch 동작을 수행한다.
- 16비트 메모리 인터페이스로 구성하면 칩의 핀 수를 줄여 단가 및 전력 소모 감소
- 시스템의 안정성 향상과 전력 소모 감소

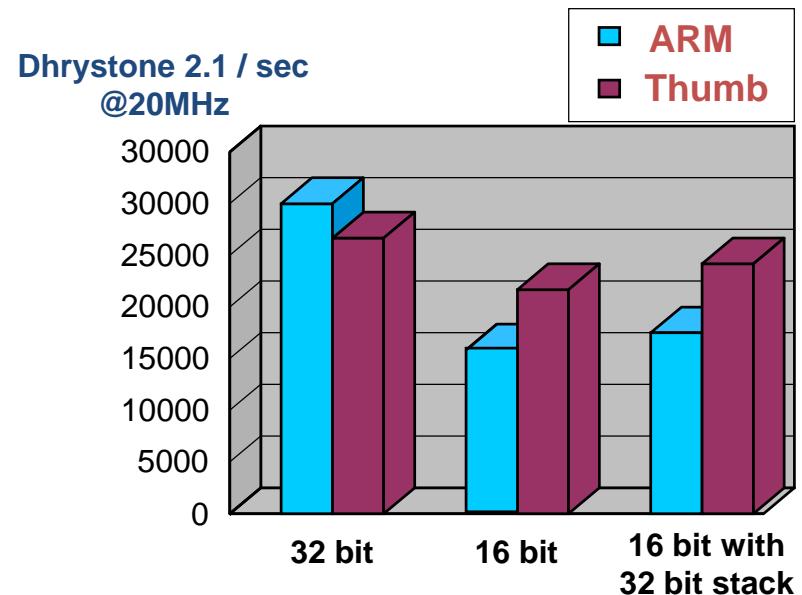
- ❖ ARM 명령과 Thumb 명령의 메모리 인터페이스에 따른 성능 비교

- ❖ Dhystone 2.1

- 소프트웨어 성능 측정 도구
- 높을 수록 좋은 성능을 나타낸다

- ❖ 성능 비교

- 32비트 메모리 인터페이스
 - ARM/Thumb 명령 처리 성능이 가장 우수
 - ARM 명령을 사용하는 경우 더욱더 우수
- 16비트 메모리 인터페이스
 - Thumb 명령이 ARM 명령보다 처리 성능 우수
 - Stack은 32비트로 동작되므로 32비트 메모리 인터페이스 보다는 성능 감소
- 16비트 메모리 인터페이스와 32비트 stack 메모리
 - Thumb 명령의 성능이 32비트 메모리 인터페이스의 ARM 명령 성능과 유사



- ❖ 조건부 실행이 안 된다. (Branch 제외)
- ❖ 사용되는 레지스터가 R0~R7로 제한된다.
- ❖ Immediate 상수 값의 사용 범위가 제한적이다.
- ❖ Inline barrel shifter의 사용이 제한적이다.
- ❖ Thumb 명령으로 처리할 수 없는 동작
 - 모든 exception handling
 - Exception이 발생하면 ARM은 무조건 ARM state가 된다.
 - PSR(Program Status Register) Transfer 명령
 - 인터럽트 제어, 프로세서 모드 변환
 - Coprocessor access
 - Cache, MMU 제어 등 포함

- ❖ Thumb 명령어는 이미지의 크기가 작고, 좁은 메모리 인터페이스에서 높은 성능 보유하지만 Thumb 명령 만으로는 사용 불가능하다. 따라서 아무리 16비트 Thumb 명령을 사용하고자 하여도 32비트 ARM 명령은 반드시 필요하다.
- ❖ ARM state와 Thumb state의 변경
 - Interworking 이라 하며 Assembler 작성시 ATPCS를 준수한다.
 - Assemble option
 - “ -apcs /interwork ”
 - C/C++ Compiler option
 - “ -apcs /interwork ”

❖ ARM Architecture v4T 이하

- Core에 'T' 가 있는 경우만 Thumb 명령 사용 가능
 - 예를 들어 Architecture v4는 Interworking이 지원 안됨
- BX 명령
 - Branch with Exchange 명령

사용법 : BX{cond} Rn

❖ ARM Architecture v5T 이상

- BX명령과 함께 BLX 명령 추가
- BLX 명령
 - Branch with Link and Exchange

사용법 : BLX <offset>



❖ ARM state서 Thumb state로

- Branch 할 주소가 있는 레지스터의 비트 0를 1로 한다.

```
adr r0, thumb_func + 1 ; load Thumb_func address and +1  
                      ; +1 to exchange Thumb state  
bx  r0               ; goto Thumb_func and exchange to Thumb state
```

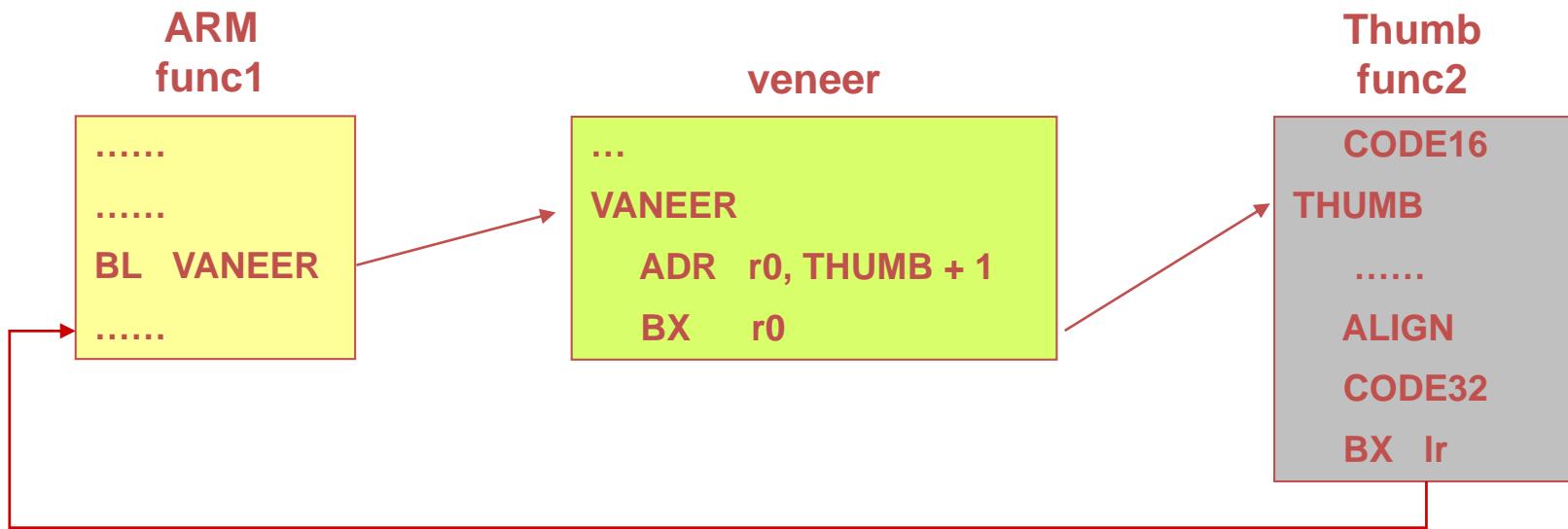
- ADR은 의사(Pseudo)명령으로 라벨형태의 어드레스위치를 읽어온다.

❖ Thumb state서 ARM state로

```
adr r0, arm_func      ; load arm_func address  
bx  r0               ; goto arm_func and exchange to ARM state
```

❖ BX 명령과 Subroutine call

- BX 명령은 되돌아갈 어드레스를 Link register에 저장하는 기능이 없다.
- 별도의 코드가 필요
 - Interworking veneer가 추가 된다.



❖ Subroutine Call 과 Veneer

- ARM 명령으로 작성된 `func1`에서 Thumb 명령으로 작성된 `func2`을 subroutine call 하면 linker가 veneer를 추가
- Veneer
 - BL 명령을 BX 명령으로 바꾼다.
 - Link register에 되돌아갈 주소 저장 등의 동작도 같이 수행
 - 별도의 branch 명령이 사용되어 속도의 저하 및 코드 사이즈 증가

❖ Architecture v5T 이상에서 추가된 명령

- Architecture v4T의 BX 명령과 subroutine call 할 때 발생되는 단점 보완
- Link register에 되돌아갈 어드레스를 저장하면서 state를 변환한다.
- 링커가 불리는 함수가 Thumb 명령이면 BL 명령을 BLX 명령으로 변환 한다.
- 사용법
 - BLX <offset>

ARM state : PC 값을 기준으로 +/- 32MB 이내
Thumb state : PC 값을 기준으로 +/-4MB 이내

Thumb 명령어 형식

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0												
0	0	0	Op		Offeset5				Rs		Rd																
0	0	0	1	1	I	Op	Rn/offset3			Rs		Rd															
0	0	1	Op		Rd			Offeset8																			
0	1	0	0	0	0	0	Op				Rs		Rd														
0	1	0	0	0	1	Op	H1	H2	Rs/Hs		Rd/Hd																
0	1	0	0	1	Rd			Word8																			
0	1	0	1	L	B	0	Ro			Rb		Rd															
0	1	0	1	H	S	1	Ro			Rb		Rd															
0	1	1	B	L	Offset5				Rb		Rd																
1	0	0	0	L	Offset5				Rb		Rd																
1	0	0	1	L	Rd			Word8																			
1	0	1	0	SP	Rd			Word8																			
1	0	1	1	0	0	0	0	S	SWord7																		
1	0	1	1	L	1	0	R	Rlist																			
1	1	0	0	L	Rb			Rlist																			
1	1	0	1	Cond				Softset8																			
1	1	0	1	1	1	1	1	Value8																			
1	1	1	0	0	Offset11																						
1	1	1	1	H	Offset																						

Instruction type

Move Shifted register

Add/subtract

Move/compare/add/subtract immediate

ALU operations

Hi register operations/branch exchange

PC-relative load

Load/store with register offset

Load/store sign-extended byte/halfword

Load/store with immediate offset

Load/store halfword

SP-relative load/store

Load address

Add offset to stack pointer

Push/pop register

Multiple load/store

Conditional branch

Software interrupt

Unconditional branch

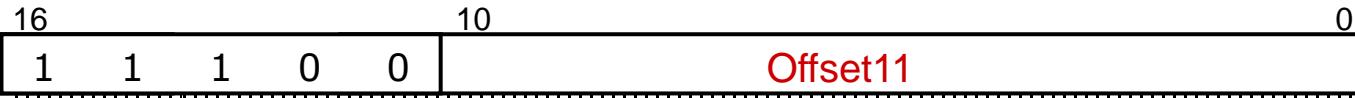
Long branch with link

❖ 무조건 분기

명령	Thumb 명령어	ARM 명령어 동작
B	B <expression>	BAL <expression>

<expression>은 destination을 나타내며 offset으로 계산된다.

11 비트의 Offset을 가지며 맨 상위 비트는 +/- sign 비트이고 나머지 10비트를 이용하여 1비트 왼쪽으로 shift 한 값을 사용. 분기 가능한 어드레스 영역 : PC +/- 2KB



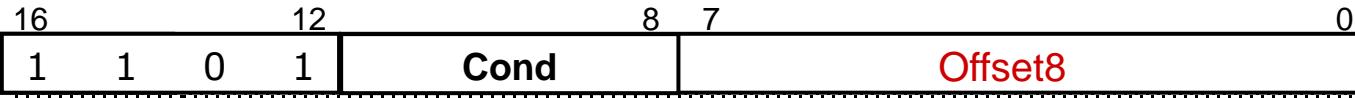
❖ 조건부 분기실행

명령	Thumb 명령어	ARM 명령어 동작
B	B{cond} <expression>	B{cond} <expression>

{cond}는 조건부 실행을 위한 condition 조건을 나타낸다.

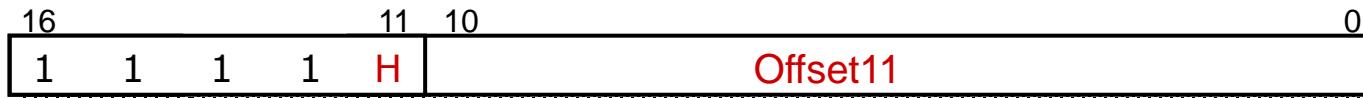
<expression>은 destination을 나타내며 offset으로 계산된다.

8 비트의 Offset을 가지며 맨 상위 비트는 +/- sign 비트이고 나머지 7비트를 이용하여 1비트 왼쪽으로 shift 한 값을 사용. 분기 가능한 어드레스 영역 : PC +/- 2^8B



❖ Long Branch with Link

명령	Thumb 명령어
BL	BL <expression> R14(LR)에 다음 명령어의 주소 값을 저장한다. 이 명령어는 2개의 16비트 Thumb 명령어로 변환되며 각각 High Offset과 Low Offset 비트를 가진다. <expression>은 destination을 나타내며 offset으로 계산된다. 11 비트씩의 Low/High Offset을 가지며 High Offset의 맨 상위 비트는 +/- sign 비트이고 Low Offset의 11비트를 이용하여 1비트 왼쪽으로 shift 한 값을 사용. 분기 가능한 어드레스 영역 : PC +/- 4MB



Low/High Offset Bit

0 = Offset high

1 = Offset low

❖ 산술 연산

opcode	명령	Thumb 명령어	ARM 명령어 동작
0	ADD	ADD Rd, Rs, Rn/#Offset3	ADDS Rd, Rs, Rn/#Offset3
10	ADD	ADD Rd, #Offset8	ADDS Rd, Rd, #Offset8
1	SUB	SUB Rd, Rs, Rn/#Offset3	SUBS Rd, Rs, Rn/#Offset3
11	SUB	SUB Rd, #Offset8	SUBS Rd, Rd, #Offset8
0101	ADC	ADC Rd, Rs	ADCS Rd, Rd, Rs
0110	SBC	SBC Rd, Rs	SBCS Rd, Rd, Rs
1001	NEG	NEG Rd, Rs	RSBS Rd, Rs, #0

Rd: Destination 레지스터

Rs: Source 레지스터

Rn: 레지스터 Operand

#Offset3: 3비트 Immediate 상수, #Offset8: 8비트 Immediate 상수

❖ 논리 연산

opcode	명령	Thumb 명령어	ARM 명령어 동작
0000	AND	AND Rd, Rs	ANDS Rd, Rd, Rs
0001	EOR	EOR Rd, Rs	EORS Rd, Rd, Rs
1100	ORR	ORR Rd, Rs	ORRS Rd, Rd, Rs
1110	BIC	BIC Rd, Rs	BICS Rd, Rd, Rs

Rd: Destination 레지스터, Rs: Source 레지스터

❖ 비교 연산

opcode	명령	Thumb 명령어	ARM 명령어 동작
01	CMP	CMP Rd, #Offset8	CMP Rd, #Offset8
1010	CMP	CMP Rd, Rs	CMP Rd, Rs
1000	TST	TST Rd, Rs	TST Rd, Rs
1011	CMN	CMN Rd, Rs	CMN Rd, Rs

Rd: Destination 레지스터, Rs: Source 레지스터
#Offset8: 8비트 Immediate 상수

❖ Shift 동작

opcode	명령	Thumb 명령어	ARM 명령어 동작
00	LSL	LSL Rd, Rs, #Offset5	MOVS Rd, Rs, LSL #Offset5
0010	LSL	LSL Rd, Rs	MOVS Rd, Rd, LSL Rs
01	LSR	LSR Rd, Rs, #Offset5	MOVS Rd, Rs, LSR #Offset5
0011	LSR	LSR Rd, Rs	MOVS Rd, Rd, LSR Rs
10	ASR	ASR Rd, Rs, #Offset5	MOVS Rd, Rs, ASR #Offset5
0100	ASR	ASR Rd, Rs	MOVS Rd, Rd, ASR Rs
0111	ROR	ROR Rd, Rs	MOVS Rd, Rd, ROR Rs
Rd: Destination 레지스터, Rs: Source 레지스터, #Offset5: 5비트 shift 값			

❖ 데이터 Move 명령

opcode	명령	Thumb 명령어	ARM 명령어 동작
00	MOV	MOV Rd, #Offset8	MOVS Rd, #Offset8
1111	MVN	MVN Rd, Rs	MVNS Rd, Rs
Rd: Destination 레지스터, Rs: Source 레지스터, #Offset8: 8비트 Immediate 상수			

- ❖ Thumb 상태에서는 모든 명령어에서 하위 레지스터(R0 ~ R7) 액세스가 가능하고 상위 레지스터(R8 ~ R12)는 MOV, ADD, CMP, BX 명령어일 경우에만 가능하다.
- ❖ 상위 레지스터 액세스 명령

명령	Thumb 명령어	ARM 명령어 동작
ADD	ADD Rd, Rs	ADD Rd, Rd, Rs
CMP	CMP Rd, Rs	CMP Rd, Rs
MOV	MOV Rd, Rs	MOV Rd, Rs
BX	BX Rs	BX Rs

Rd: Destination 레지스터, Rs: Source 레지스터

Rd/Rs는 하위 레지스터(R0 ~ R7), 상위 레지스터(R8 ~ R12), r13(SP), r14(LR), r15(PC) 모두 액세스 가능.

ADD의 경우 Rd/Rs 모두 하위 레지스터(R0 ~ R7)가 사용될 경우 형식은 ADD Rd, Rd, Rs를 사용.

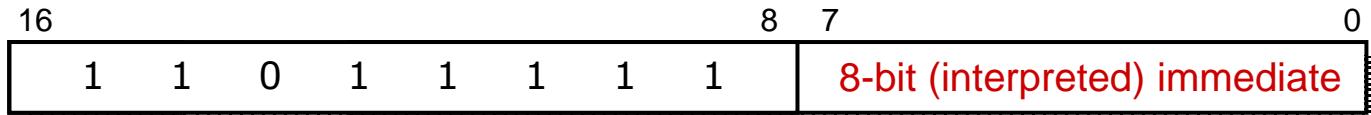
- ❖ Thumb에서는 CPSR, SPSR을 직접 액세스하는 명령어가 없다.

Software Interrupt(SWI), Multiply 명령어

mds ACADEMY

❖ Software Interrupt(SWI)

명령	Thumb 명령어	ARM 명령어 동작
SWI	SWI Immediate8	SWI Immediate8
#Immediate8: SWI number를 기입한다.(0 ~ 255)		



❖ 32 비트 Multiply

명령	Thumb 명령어	ARM 명령어 동작
MUL	MUL Rd, Rs	MULS Rd, Rs, Rd
Rd: Destination 레지스터 Rs: Source 레지스터 $Rd := Rs * Rd[31:0]$		

❖ LDR/STR 명령

명령	Thumb 명령어	ARM 명령어 동작
LDR/STR	LDR/STR{size} Rd, [Rb, Ro]	LDR/STR{size} Rd, [Rb, Ro]
	LDS{size} Rd, [Rb, Ro]	LDRS{size} Rd, [Rb, Ro]
	LDR/STR{size} Rd, [Rb, #Immediat5]	LDR/STR{size} Rd, [Rb, #Immediat5]
상대주소 지정방식	LDR Rd, [PC, #Immediat8]	LDR Rd, [R15, #Immediat8]
	LDR/STR Rd, [SP, #Immediat8]	LDR/STR Rd, [R13 #Immediat8]
주소계산	ADD Rd, PC, #Immediat8	ADD Rd, R15, #Immediat8
	ADD SP, #Immediat7	ADD R13, R13, #Immediat7

Rd: Destination 레지스터, Rb: Base 레지스터, Ro: Offset 레지스터

#Immediat5: 0 ~ 31 범위의 Immediate 상수

#Immediat7: 0 ~ 127 범위의 Immediate 상수

#Immediat8: 0 ~ 255 범위의 Immediate 상수

각 Immediate 상수의 범위는 halfword 액세스 경우에는 1비트, word 액세스 일 때는 2비트 늘어난다. (반드시 word-aligned 또는 halfword-aligned여야 한다.)

Size: 전송되는 데이터 크기를 나타내며 B(byte), H(Halfword)이고 LDS는 Signed load를 의미

Load / Store Multiple 명령어

❖ LDM/STM 명령

명령	Thumb 명령어	ARM 명령어 동작
LDM	LDMIA Rb!, { Register_list }	LDMIA Rb!, { Register_list }
STM	STMIA Rb!, { Register_list }	STMIA Rb!, { Register_list }

Rb: Base 레지스터
{ Register_list }는 데이터를 저장 혹은 저장할 레지스터를 지정. Low order의 레지스터에서 high order 순으로 지정하며 범위는 R0 ~ R7이다.
IA(Increment After) 어드레스 지정 방식만을 지원한다.

❖ PUSH/POP 명령

명령	Thumb 명령어	ARM 명령어 동작
PUSH	PUSH {Register_list, (LR) }	STMFD R13!, {Register_list, (R14) }
POP	POP {Register_list, (PC) }	LDMFD R13!, {Register_list, (R15) }

{ Register_list }는 데이터를 저장 혹은 저장할 레지스터를 지정. Low order의 레지스터에서 high order 순으로 지정하며 범위는 R0 ~ R7이다.
FD(Full Descending) 어드레스 지정 방식만을 지원한다.

1. ARM 프로세서의 명령어
2. 32 비트 ARM 명령어
3. 16 비트 Thumb 명령어
4. v5TE에 추가된 명령어

Architecture	특 징	프 로 세 서
v1, v2, v3	이전 버전, 현재는 거의 사용 안됨	ARM610, ...
v4	System 모드 지원	StrongARM(SA-1110, ...)
v4T	v4의 기능과 Thumb 명령 지원	ARM7TDMI, ARM720T ARM9TDMI, ARM940T, ARM920T
v5TE	v4T의 기능 ARM/Thumb Interwork 개선 CLZ 명령, saturation 명령, DSP Multiply 명령 추가	AEM9E-S, ARM966E-S, ARM946E-S ARM1020E XScale
v5TEJ	v5TE의 기능 Java 바이트 코드 실행	ARM7EJ-S, ARM9EJ-S, ARM1020EJ-S
v6	SIMD 명령(MPEG4 같은 미디어 처리용) Unaligned access 지원	ARM1136EJ-S

T : Thumb 명령 지원, E : DSP 기능 확장, J : Java 명령 지원

ARMxxx-S : Synthesizable core

- ❖ Architecture v4T의 ARM / Thumb 명령 지원
- ❖ ARM / Thumb interworking 개선을 위한 BLX 명령 추가
- ❖ Breakpoint 명령 추가
- ❖ Count Leading Zero(CLZ) 명령 추가
- ❖ Coprocessor 관련 명령 확장
- ❖ Saturate 연산 명령 추가
- ❖ Cache Preload 명령 추가
- ❖ Double word 단위의 Load / Store 명령 추가
- ❖ Double word 단위의 coprocessor 전송 명령 추가

❖ Saturate 연산

- 연산 결과가 ‘+’의 최대값과 ‘-’의 최소값이 넘지 않도록 한다.
- 만약 값이 넘으면 CPSR의 Q-flag를 세트 한다.
 - 세트 된 Q-flag는 사용자가 클리어 한다.

❖ 연산 결과 비교

- 일반적인 산술 연산의 경우
 - $0x7FFFFFFF + 1 \Rightarrow 0x80000000$, 즉 -1이 된다.
 - $0x80000000 - 1 \Rightarrow 0x7FFFFFFF$, 즉 +값이 된다.
- Saturate 연산의 경우
 - $0x7FFFFFFF + 1 \Rightarrow 0x7FFFFFFF$ 유지, Q-flag 세트
 - $0x80000000 - 1 \Rightarrow 0x80000000$ 유지, Q-flag 세트

❖ 분수 연산 지원

- QDADD 등을 사용화면 Multiply와 함께 +1 ~ -1 까지 분수 연산 가능

Saturate 연산 명령

명령어	QADD	Add and Saturating
형식	QADD{cond} Rd, Rm, Rn	
동작	Rd := SAT(Rm + Rn)	

명령어	QSUB	Subtract and Saturating
형식	QSUB{cond} Rd, Rm, Rn	
동작	Rd := SAT(Rm - Rn)	

명령어	QDADD	Add and Double Saturating
형식	QDADD{cond} Rd, Rm, Rn	
동작	Rd := SAT(Rm + SAT(Rn*2))	

명령어	QDSUB	Subtract and Double Saturating
형식	QDSUB{cond} Rd, Rm, Rn	
동작	Rd := SAT(Rm - SAT(Rn*2))	

- ❖ Signed 16 x 16과 signed 32 x 16 곱셈 명령 추가

명령어	SMULxy	Signed 16 * 16 bit Multiply
형식	SMULxy{cond}	Rd, Rm, Rs
동작	Rd := Rm[x] * Rs[y]	

명령어	SMULwy	Signed 32 * 16 bit Multiply
형식	SMULwy{cond}	Rd, Rm, Rs
동작	Rd := (Rm * Rs[y])[47:16]	

Signed Multiply-Accumulate 명령

명령어	SMLAx_y	Signed 16 * 16 bit Multiply and Accumulate
형식	SMLAx_y{cond} Rd, Rm, Rs, Rn	
동작		$Rd := Rn + Rm[x] * Rs[y]$

명령어	SMLAW_y	Signed 32 * 16 bit Multiply and Accumulate
형식	SMLAW_y{cond} Rd, Rm, Rs, Rn	
동작		$Rd := Rn + (Rm * Rs[y])[47:16]$

명령어	SMLALx_y	Signed long 16 * 16 bit Multiply and Accumulate
형식	SMLALx_y{cond} RdLo, RdHi, Rm, Rs	
동작		$RdHi, RdLo := RdHi, RdLo + Rm[x] * Rs[y]$

❖ Doubleword (64비트) 단위의 LDR / STR 지원

- Destination 레지스터는 2개씩 사용되는데, 짹수 번호로 시작하는 레지스터로 지정하면 다음 레지스터와 한 쌍을 이룬다.
 - R0, R2, R4, R6, R8, R10, R12로 지정

명령어	LDRD	Load Double Word
형식	LDR{cond}D Rd, <address_mode>	
동작	Rd := <address>, R(d+1) := <address + 4 byte>	

- ❖ Cache의 사용에 앞서 Cache를 update하는 명령

명령어	PLD	Cache Preload
형식	PLD <address_mode>	
동작		Load instruction to cache in <address_mode>

- ❖ MRRC와 MCRR 명령 추가

- 2개의 ARM 레지스터의 내용과 Coprocessor 사이의 데이터 Transfer

- ❖ 다양한 기능을 제공하기 위한 명령 추가

- CDP2, LDC2, STC2, MCR2 그리고 MRC2

❖ Architecture v4T의 state 변경 명령

- BX 명령 제공
- 서브루틴콜 하는 경우 링크 레지스터에 되돌아갈 주소를 저장하는 기능이 없다.
- 링커가 ARM에서 Thumb으로 서브루틴이 콜되는 경우 Veneer라 불리는 루틴을 추가하여 링크 레지스터에 되돌아갈 주소 저장
 - 성능 감소 및 코드 사이즈 증가

❖ Architecture v5TE의 state 변경 명령

- BLX 명령
- State의 변경과 링크 레지스터에 되돌아갈 주소 저장

명령어	BLX	Branch with link and exchange
형식	<p>BLX label</p> <p>Label must be within +/- 32MB</p> <p>BLX{cond} Rm</p> <p>Rm은 branch 할 주소를 가지고 있고, Thumb으로 branch 하는 경우 bit [0]은 1로 되어 있어야 한다.</p>	
동작	Branch to label or address in Rm and change the state	

명령어	BKPT	Breakpoint
형식	BKPT <immediate_16> <immediate_16> 은 ARM에서는 무시되나 디버거에 부가적으로 정보를 전달하기 위해서 사용이 가능하다.	
동작	프로세서를 멈추고 Debug state 로 가도록 한다.	

❖ CLZ 명령

- MSB로부터 최초 1이 나타내는 위치 검색

❖ CLZ 명령 응용

- 인터럽트 Pending 비트 검사
- Normalize

명령어	CLZ	Count Leading Zero
형식	<p>CLZ{cond} Rd, Rm</p> <p>{cond}는 조건부 실행을 위한 condition 조건을 나타낸다.</p> <p>Rd는 결과를 저장한다.</p> <p>Rm은 검사할 레지스터를 나타낸다.</p>	
동작	<p>Rm 레지스터의 내용을 검사하여 MSB로부터 맨 먼저 1이 나타나는 위치를 Rd에 저장한다.</p>	

질의 응답