1.1. ARM7의 특징

아래의 Table 1-1은 ARM7의 일반적인 특징들을 보여 주고 있다.

특 징	내 용
32 Bit RISC Processor	ARM7은 내부적으로 32Bit의 데이터 버스와 32bit의 어드레스 버스를 제공한다. - load/store Architecture - Uniform & fixed length instruction fields
Big/Little Endian Mode	메모리에 저장되는 순서인 Big/Little Endian Mode를 모두 제공한다.
High Performance RISC	낮은 전원에서도 높은 성능을 발휘한다.
Fast Interrupt Response	인터럽트 처리를 빠르게 해줄 수 있는 Fast Interrupt 기능을 제공한다.
Excellent high level language support	C나 다른 언어에서 일반적으로 사용하는 기능들을 바로 구현할 수 있는 명령어들이 제공된다. 예) ++, -- 연산자와 같은 기능인 Auto increment/decrement addressing mode 제공한다.
Simple & Powerful Instruction Set	명령어의 종류가 적으나, 옵션들을 사용하여 다양하게 적용시킬 수 있으며, 강력한 기능을 제공한다. - load/store multiple instructions - Conditional execution of all instructions

Table 1-1. ARM7의 특징들

1.2. ARM7의 Register

ARM7에는 32bit의 General Purpose Register 31개와 6개의 Status Register를 가지고 있다. 아래의 Table 1-2는 General Purpose Register중 특별하게 사용되고 있는 Register와 Status Register에 대한 설명이다.

Register		Description
Special General Purpose Register	Program Counter(R15)	r15는 다른 CPU처럼 PC와 같은 역할을 합니다. 다만 차이가 있다면 r15를 일반 다른 레지스터처럼 오퍼랜드로 사용할 수 있다는 점하고, Instruction Set에서 pc라는 키워드와 r15를 동일하게 취급한다.
	Link Register(R14)	r14는 링크 레지스터라고 부릅니다. ARM7에서는 CALL, RET 같은 명령이 없어 바로 서브루틴을 호출할 수 없습니다. 대신 Branch with Link라는 명령(BL)이 있는데, 해당 명령을 수행하면, CALL과 비슷하게 다음에 수행될 PC값을 스택이 아니라 r14에 넣고 분기 번지를 PC에 넣어 분기 합니다. 즉, 스택을 하지 않는 것입니다. 복귀할 때는 RET대신 mov pc, lr이라는 데이터 전송명령으로 복귀 합니다. 만약에 서브루틴 안에 서브루틴을 호출할 경우에는 r14를 스택에 저장하여야 합니다.
	Stack Pointer(R13)	ARM7에는 스택을 위한 명령어가 따로 없습니다. 즉, Push나 Pop등의 명령어가 제공되지 않습니다. 그러나 sp라는 키워드를 사용하여 r13을 쓸 수 있습니다. Push, Pop명령이 없으므로, ARM7에서는 같은 기능을 일반 데이터 전송 명령을 통해 해결 합니다. ARM7의 데이터 전송 명령은 Auto Increment 기능이 있어서 하나의 명령으로 Push나 Pop과 동일한 기능을 수행할 수 있습니다.
Status Register	CPSR (Current Processor Status Register)	ARM7에는 32bit의 Status Register가 6개 있습니다. 이 중에서 현재의 프로세서의 상태를 저장하기 위해서 사용된다.
	SPSR	Saved Processor Status Register는 CPSR값을 저장해 두는 역할을 한다.

Table 1-2. Special General Register & Status Register

1.2. ARM7의 Register(Continued)

아래의 Figure 1-1은 Status Register의 Bit Configuration을 나타내며, Table 1-3은 각 각의 Bit Configuration에 대한 설명이다.

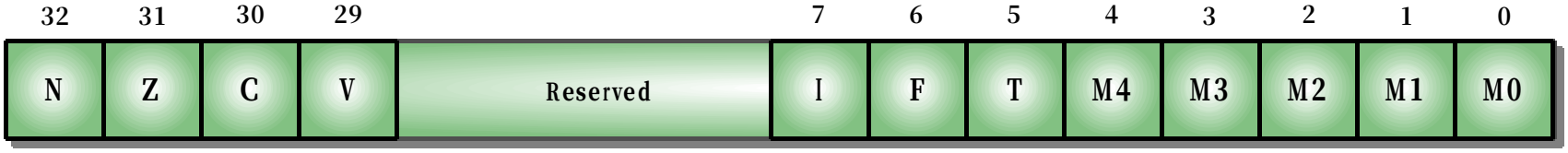

Figure 1-1. Status Register

Bits	Description
N(Negative/less than) Flag	연산의 결과가 마이너스인 경우에 세트되는 플래그이다.
Z(Zero) Flag	연산의 결과가 0인 경우에 세트되는 플래그이다.
C(Carry/Brrow/Extend) Flag	자리올림이나 내림이 발생할 경우, 그리고 shift 연산 등에서 사용되는 플래그이다.
V(Overflow) Flag	연산의 결과가 Overflow 되었을 경우에 사용되는 플래그이다.
I(Interrupt) Control	인터럽트 중에서 IRQ를 금지 시킬 수 있는 플래그이다. 이 값이 세트되면 IRQ를 Disable된다.
F(Fast Interrupt) Control	인터럽트 중에서 FIQ를 금지 시킬 수 있는 플래그이다. 이 값이 세트되면 FIQ를 Disable된다.
T(Thumb) Control	Thumb를 Enable/Disable하는 플래그이다. 이 값이 세트되면 Thumb State가 된다.
M4 - M0(Mode) Control	이 비트의 조작에 따라 프로세서의 동작 모드가 변하게 된다.

M[4:0]	Mode	M[4:0]	Mode
0b10000	User	0b10111	Abort
0b10001	FIQ	0b11011	Undefined
0b10010	IRQ	0b11111	System
0b10011	Supervisor	-	-

Table 1-3. Bits Configuration of Status Register

1.2. ARM7의 Register(Continued)

아래의 Table 1-4는 프로세서의 동작모드에 대해 설명하고 있다.

Mode	Description	Mode	Description
User(usr)	Normal program execute mode	Abort(abt)	Implements virtual memory and/or memory protection
FIQ(fiq)	Supports a high-speed data transfer or channel process	Undefined(und)	Supports software emulation of hardware coprocessors
IRQ(irq)	Used for general purpose interrupt handling	System(sys)	Runs privileged operating system tasks
Supervisor(svc)	A protected mode for the operating system		

Table 1-4. 프로세서의 동작모드

동작모드는 왜 있는 걸까요?

ARM7은 각 동작모드에 따라서 몇 가지 기능을 제공합니다. 가장 큰 것으로 레지스터를 리맵핑시키는 기능이 있습니다. 프로세서의 동작모드는 보통 때는 User 모드 입니다. 이 때는 기본적으로 R0에서 R15까지를 사용하고 있습니다. 이 때 FIQ가 발생했다면 동작모드는 FIQ 모드로 바뀌게 됩니다. 이와 동시에 R8과 R14까지 의 7개의 레지스터는 FIQ 전용 레지스터로 리 맵핑됩니다. 즉, User 모드의 레지스터와는 별개였던 레지스터 7개가 R8부터 R14까지 의 위치에 배치되는 것입니다. 물론 기존의 User 모드의 사용하던 R8-R14까지 의 레지스터와는 별개의 레지스터입니다. 이처럼 사용되었을 때의 장점을 보면, 일반적인 프로세서의 인터럽트 처리루틴을 보면 인터럽트 발생시 가장 먼저 하는 일이 사용 중이던 레지스터를 스택에 저장하고, 또 처리 루틴이 종료될 때 다시 복구시키는 일을 합니다. 이 부분에서 로드가 많이 발생하게 됩니다. 그러나 ARM7에서는 레지스터를 따로 관리를 하기 때문에 스택에 저장하거나 복구과정을 생략해도 좋습니다. 그만큼 실행 속도가 향상시킬 수 있는 것이죠.

1.2. ARM7의 Register(Continued)

아래의 Figure 1-2는 프로세서의 동작모드에 따라 사용하고 있는 Register들을 보여 주고 있다.

System/User	FIQ	Supervisor	Abort	IRQ	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12
R13(SP)	R13_fiq	R13_svc	R13_abt	R13_irq	R13_und
R14(LR)	R14_fiq	R14_svc	R14_abt	R14_irq	R14_und
R15(PC)	R15	R15	R15	R15	R15
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

Figure 1-2. General Purpose Register and Status Register

1.3. Exception

아래의 Table 1-5는 Exception에 대하여 설명한 것이다.

Address	Exception	Mode on Entry	Priorities	Description
0x0000_0000	Reset	supervisor	1	Reset 핀이 low에서 high가 될 때 발생된다.
0x0000_0004	Undefined instruction	Undefined	6	ARM7에 정의 되어 있지 않은 명령어를 만났을 경우 발생된다. 이 기능은 코프로세서를 사용하는 경우와 관련되어 사용됩니다.
0x0000_0008	Software Interrupt	Supervisor	6	프로그램에서 임의로 인터럽트를 호출하는 경우입니다. ARM7의 명령중 SWI가 이에 해당합니다. 이 Exception이 발생하면 프로세서 동작모드가 Supervisor 모드로 바뀌도록 되어 있고, 보통 OS의 시스템의 Call를 구현하기 위해 사용됩니다.
0x0000_000C	Abort(prefetch)	Abort	5	프로세서가 메모리로부터 instruction을 가져 오거나, 혹은 instruction을 동작시키면서 데이터를 가져오려고 할 경우, 해당 메모리를 Access할 수 없다면 Abort Exception이 발생합니다.
0x0000_0010	Abort(data)	Abort	2	
0x0000_001C	Reserved	Reserved	Reserved	-
0x0000_0018	IRQ	IRQ	4	일반적으로 I/O 장치로부터의 입력이 들어오면 반응을 하는 흔히 말하는 인터럽트 Exception입니다. IRQ의 종류로는 내부 타이머나 시리얼, 혹은 외부 IRQ입력 등이 될 수 있다.
0x0000_001C	FIQ	FIQ	3	개념적으로는 IRQ와 거의 유사한데, 다만 보다 빠른 처리를 할 수 있도록 제공되는 Exception입니다.

Table 1-5. Exception

1.3. Exception(Continued)

아래의 list 1-1에서 list 1-2는 Exception이 일어 났을 때의 처리 과정을 보여 주고 있다.

Reset Exception 절차

1. Reset 핀의 입력으로 Exception이 발생

2. PC의 값을 R14-svc에 넣고, SPSR_svc에 CPSR의 값을 넣는다. PC과 CPSR은 의미 없는 값이다.

3. CPSR의 bit중에 I와 F를 세트하여 인터럽트를 Disable하고 ,T는 Clear하며, M[4:0]에 b10011을 넣어 Supervisor Mode 로 변경한다.

4. PC에 0x00을 넣어 Next Instruction을 fetch한다.

5. 필요하다면 ARM state를 변경하고 Execution을 시작한다.

list 1-1. Reset Exception 절차

IRQ Exception 절차

1. User 모드에서 R0 - R15를 사용하고 있는 도중에 IRQ 발생

2. 이 시점의 PC값은 IRQ 처리 이루 복귀할 번지이다. 그 값을 R14(이미 IRQ 모드가 되었으므로 R14_irq)에 넣는다.

3. SPSR_irq에 CPSR를 넣어 준다.

4. ARM 프로세서는 동작모드를 IRQ 모드로 바꾼다.
(M[4:0]은 0b10010을 넣어 모드를 변경하고, I 비트를 세트하여 IRQ를 Disable한다. 이때 R13과 R14는 IRQ 전용 레지스터로 대치된다.)

5. PC에 0x18을 넣어 분기한다.

6. 해당 IRQ의 내용을 수행한다.

7. CPSR에 SPSR_irq값을 넣는다.

8. IRQ에서 Return하기 위해서는 subs pc, r14_irq, #4를 수행하여야 한다.

list 1-2. IRQ Exception 절차

2.1. ARM7 Instruction의 특징

아래의 Table 2-1은 ARM7 Instruction의 특징을 보여 주고 있다.

특 징	내 용
모든 명령어가 32bit 하나의 Word로 구성된다.	CISC Core인 경우 명령어에 따라 1바이트 명령부터 여러 바이트로 구성이 된다. 그런데 ARM7은 모든 명령어를 한 워드로 처리 합니다. 주소는 상대주소 방식을 사용하며, Immediate 상수값도 32비트 값은 그대로 넣을 수 없습니다. 예로 R0에 32비트 상수를 넣고 싶다면 해당 메모리를 상대주소로 참조해서 얻어 와야 합니다. 이 특징으로 인하여 모든 명령어를 같은 사이즈로 처리함에 따라 파이프라인 구현이 용이하고, 명령어 해석기를 설계할 경우 예외 처리부분이 없으므로, 쉽고 고속으로 처리할수 있게 된다. 하지만 이렇게 설계되다 보니 코딩시에 몇몇 제한이 따르게 됩니다. 특히 Immediate 상수를 오퍼랜드에 지정할 경우에 문제가 됩니다.
대부분의 명령어에 S라는 접미사를 사용하여 플래그 레지스터에 영향을 줄지 여부를 결정할 수 있다.	연산명령을 수행할 때 'S'를 붙이면 해당 결과에 따라서 플래그 값들이 변하게 되고, 붙이지 않으면 영향을 미치지 않도록 할 수 있습니다.

Table 2-1. ARM7 Instruction의 특징

2.1. ARM7 Instruction의 특징(Continued)

특 징	내 용

모든 명령어가 조건적으로 실행시킬 수 있다.	8x86에서는 조건적으로 실행하기 위해 jc, jz와 같은 명령을 사용합니다. 이 의미는 제로 플래그가 설정되어 있으면 점프를 해라, 혹은 캐리 플래그가 설정되어 있으면 점프를 해라.. 라는 의미임을 입니다. ARM의 경우엔 그런 플래그의 사용이 점프명령에 국한되지 않고, 예외 없이 모든 명령어에 사용할 수 있습니다.
	예) BEQ jmp_1 ; 만약 Z flag가 세트 되어 있으면 jmp_1로 점프한다.
	MOVEQ r0, r1 ; 만약 Z flag가 세트 되어 있으면 r0에 r1을 저장한다.
	
	위의 경우처럼 B와 같은 Branch 명령뿐만아니라 데이터 전송명령에도 플래그 옵션을 사용했음을 볼수 있습니다. 이런 ARM의 특징은 명령을 단축시킬수 있다는 것입니다.
	예) C언어에서 a = (b==c)?d:e; a,b,c,d,e가 차례대로 r1, r2, r3, r4, r5에 할당되다고 하면 이것을 Assemble로 표현을 하면 아래와 같다. 8x86에서는 JMP명령이 하나는 적어도 사용되었을 것이다.
	CMPS r2, r3
	MOVEQ r1, r4
	MOVNE r1, r5
	
	Instruction의 조건 옵션 아래의 Table 2-2을 참조 하세요.

Table 2-1. ARM7 Instruction의 특징(Continued)

2.1. ARM7 Instruction의 특징(Continued)

아래의 Table 2-2는 CPSR에 있는 Condition Fields에 대하여 요약한 것이며, 이 것들은 Instruction Set의 조건 옵션으로 사용됩니다.

Code	Suffix	Flags	Meaning
0b0000	EQ	Z set	equal
0b0001	NE	Z clear	not equal
0b0010	CS	C set	unsinged higher or same
0b0011	CC	C clear	unsigned lower
0b0100	MI	N set	negative
0b0101	PL	N clear	positive or zero
0b0110	VS	V set	overflow
0b0111	VC	V clear	no overflow
0b1000	HI	C set and Z clear	unsigned higher
0b1001	LS	C clear and Z set	unsigned lower or same
0b1010	GE	N equals V	greater or equal
0b1011	LT	N not equals to V	less than
0b1100	GT	Z clear AND (N equals V)	greater than
0b1101	LE	Z set OR (N not equals to V)	less than or equal
0b1110	AL	(ignored)	always

Table 2-2. Condition Fields

2.2. Instruction Format

아래의 Table 2-3은 ARM에서 사용하는 Instruction의 Format을 보여 주고있다.

Instruciton	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Data Processing/ PSR Transfer	Cond				0	0	I	Opcode				S	Rn				Rd				Operand 2											
Multiply	Cond				0	0	0	0	0	0	A	S	Rd				Rn				Rs				1	0	0	1	Rm			
Multiply Long	Cond				0	0	0	0	1	U	A	S	RdHi				RdLo				Rn				1	0	0	1	Rm			
Sigle Data Swap	Cond				0	0	0	1	0	B	0	0	Rn				Rd				0	0	0	0	1	0	0	1	Rm			
Branch and Exchange	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			
Halfword Data Transfer: register offset	Cond				0	0	0	P	U	0	W	L	Rn				Rd				0	0	0	0	1	S	H	1	Rm			
Halfword Data Transfer: Immediate offset	Cond				0	0	0	P	U	1	W	L	Rn				Rd								1	S	H	1	Offset			
Single Data Transfer	Cond				0	1	I	P	U	B	W	L	Rn				Rd				Offset											
Undefined	Cond				0	1	1																					1				
Block Data Transfer	Cond				0	0	0	P	U	S	W	L	Rn				Register List															
Branch	Cond				1	0	1	L	Offset																							
Coprocessor Data Transfer	Cond				1	1	0	P	U	N	W	L	Rn				CRd				CP#				Offset							
Coprocessor Data Operation	Cond				1	1	1	0	CP Opc				CRn				CRd				CP#				CP			0	CRm			
Coprocessor Register Transfer	Cond				1	1	1	0	CP Opc			L	CRn				Rd				CP#				CP			1	CRm			
Software Interrupt	Cond				1	1	1	1	Ignored by processor																							

Table 2-3. Instruction Format

2.3. Instruction Operation

아래의 Figure 2-1은 오퍼랜드의 data flow format을 보여 주고있다.

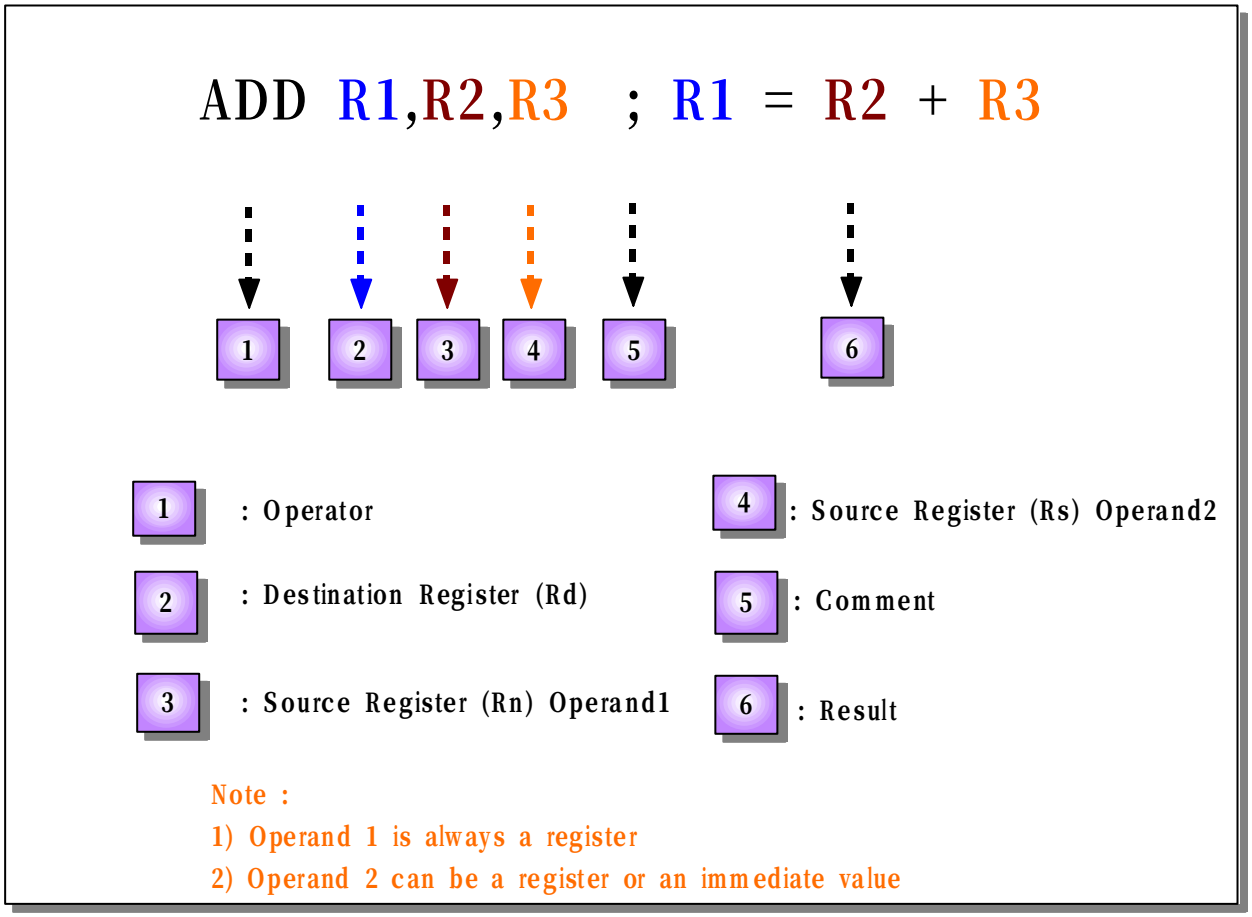

Figure 2-1. 오퍼랜드의 Data Flow Format

2.4. Branch Instruction

아래의 Table 2-4는 Branch Instruction Format이다.

Cond	1	0	1	L	Offset

Table 2-4. Branch Instruction Format

Note

Cond 부분은 조건 옵션 부분이고, 0b101은 B 명령의 코드입니다. L 부분은 '1'인 경우 BL이 되는 것이고, '0'이면 B 명령이 된다. B는 JMP라고 생각하면 되고, BL은 CALL로 생각하면 됩니다. 다만 BL의 경우엔 PC값을 스택에 넣는 것이 아니라 r14(LR)에 넣는 다는 것이 차이가 있습니다.나머지 하위 24비트가 Offset으로 사용되는데, ARM7은 모든 명령어들이 Word단위이므로 총 +/- 32MByte 영역을 커버합니다.

```
예1) .text
     .global _start
     _start:b reset       ; Noncondtion jump

      reset:
            mov r1, #0x90000000

예2) bl print_str         ; Function call

     print_str:
            mov pc, r14 ; Return function
```

2.4. Data Processing and PSR Instruction

아래의 Table 2-5 Data Processing 와 PSR의 Instruction Format이다.

Cond	0	0	I	Opcode	S	Rn	Rd	Operand 2

Table 2-5. Data Processing and PSR Instruction Format

Note

- Data Processing and PSR Instruction Format의 구성하고 있는 비트의 의미

1) Cond ([31:28],4bit) : 해당 명령의 조건 실행 플래그입니다. 해당 플래그를 통해 명령을 현재 플래그 레지스터(CPSR)의 상태에 따라 실행 여부를 결정하는데 사용되는 플래그입니다. 모든 명령어에 포함되므로 데이터 프로세싱 명령에도 포함됩니다.
2) I ([25], 1bit) : Operation 2 로 지정되어 있는 부분이 Immediate Operand 인지 아닌지 여부를 나타내는 비트입니다. Immediate Operand라 함은, 상수값을 얘기합니다. 8x86의 MOV AX, 1234이라고 할 때 1234을 의미합니다.
3) Opcode ([24:21],4bit) : 데이터 프로세싱 명령중 어떤 명령인지를 나타내는 필드입니다.
4) S ([20], 1bit) : S 비트가 1인 경우 데이터 프로세싱 명령의 결과가 CPSR에 영향을 미칩니다. 즉, 0인 경우에는 CPSR의 값은 변하지 않게 됩니다.
5) Rn ([19:16],4bit) : ARM 데이터 프로세싱 명령은 그 결과와 첫번째 오퍼랜드는 항상 레지스터로 지정해야 합니다. Rn은 첫번째 오퍼랜드를 가리키는 것입니다. ARM에서 한번에 볼 수 있는 범용 레지스터는 sp, lr, pc등을 포함해서 r0에서 r15까지이므로 4bit를 통하여 레지스터를 나타나게 됩니다.
6) Rd ([15:12],4bit) : 오퍼레이션의 결과가 저장될 레지스터를 의미합니다. 역시 레지스터를 가리키므로 4비트를 사용하고 모든 명령에서 디폴트로 사용되는 필드입니다. ARM 명령의 결과 값은 항상 레지스터로 저장되어야 합니다.
7) Operand 2([11:0], 12bit) : ARM의 ALU에 연산명령을 내릴 경우 한쪽 입력은 오퍼랜드 1이 들어오고, 다른 쪽에서 레지스터나 메모리와 값이 barrel shifter를 거쳐 들어오는데 이 것이 Operand 2이다. ARM7에서는 Operand 2를 지정할 때, 해당 값을 쉬프트 시켜서 사용할 수 있습니다.

2.4.1. 미리 알아야 할 사항

아래 List 1-1은 Operand 2에 대한 설명을 하고 있다.

ARM7의 데이터 프로세싱 명령의 Operand 2의 타입은 Immediate Operand 와 레지스터 Operand가 있습니다.

1) Register Operand

I 필드가 0일 경우엔 Operand 2가 레지스터임을 의미합니다. 레지스터를 나타내시 위해서는 아시다시피 4비트만 있으면 됩니다. 하지만 Operand 2 위한 비트 수는 총 12 비트입니다. 구체적인 레지스터 Operand 2의 형식은 Figure 2-2와 같습니다.

Figure 2-2. Operand 2의 Format중 Register Operand

위의 그림처럼 Rm이라고 레지스터를 지정하는 4비트가 포함되어 있습니다. 그리고 레지스터는 ALU에 집어 넣을 때 Shifter를 통해서 넣어 주도록 되어 있습니다. Shift라는 필드(8bit)는 레지스터의 값을 얼만큼 쉬프트해주는 역할을 해주는 것입니다. Shift필드에는 Shift회수를 지정하는 방법이 두 가지가 있습니다. 첫째는 Shift 필드안에 그 회수를 포함시키는 방법이고, 두번째는 Shift 회수가 들어 있는 레지스터를 Shift 필드 안에서 지정하는 방법입니다. 아래의 Figure 2-3은 Shift필드에 대하여 Shift 회수를 넣어 주는 방법을 보여 주고 있습니다.

Figure 2-3. Shift 회수 넣어 주는 방법

4번 비트인 최 하위 비트의 값이 0이냐 1이냐에 따라 두가지로 구분이 됩니다. 이 값이 0이면 직접 Shift 회수를 지정하는 방법으로, Count부분이 5 비트로서 그 값을 의미합니다. ARM7의 레지스터는 32비트이므로 5비트는 있어야 Shift회수를 표현할 수 있습니다. 그러면 Type이라는 필드는 어떤 것을 의미하는 것일 까요? 아래의 Table 2-6은 Type 필드의 내용을 보여 주고 있습니다.

List 2-1. Operand 2에 대한 설명

Value	Instrucion	Mnemonic
0b00	logical left	LSL
0b01	logical right	LSR
0b10	arithmetic right	ASR
0b11	rotate right	ROR

Table 2-6. Operand 2의 Type 필드

2) Immediate Operand

레지스터 오퍼랜드의 경우 총 12 비트 중에서 4비트 레지스터를 지정하고, 나머지 8 비트는 해당 레지스터 값을 쉬프트하는 방법을 지정하도록 되어 있습니다. Immediate 오퍼랜드는 몇가지 문제가 있습니다. 32 비트 레지스터에 값을 써 넣고 싶은데 공간은 12비트 밖에는 없다는 것이지요. 80x86등에서야 그냥 여러 바이트로 명령을 처리 했었으니 그런 문제가 없으나, ARM에서는 이것 때문에 모든 명려어 32 비트라는 규칙을 깰 수는 없었습니다. 그래서 Immediate Operand는 Figure 2-4 처럼 구성이 됩니다.

Figure 2-4. Operand 2의 Format중 Immdiate Operand

위의 그림을 보면, 이번엔 Rotate 필드(4비트)가 있고, 또 8비트의 Imm 필드가 있습니다. Imm은 말 그대로 Immediate Value를 넣는 공간입니다. 이 공간이 8비트 까지는 아무런 문제가 없이 넣을 수 있습니다. 그러면 Rotate 필드의 하는 역학은 무었일까요? Immediate Operand rotate 필드는 부호없는 4비트 정수로서 8비트 Immediate 갑에 Shift 오퍼레이션을 할 수 있도록 사용되는 값이다. rotate 필드 * 2 값 만큼 8비트 Immediate 값을 Right Rotate 시킵니다. 결국 8비트 값을 32비트로 확장시키는 역할을 하며, 이를 통하여 자주 사용되는 많은 상수들을 사용가능하도록 합니다.

8비트의 값이 ALU로 들어가기 전에 Shifter로 들어가면, 이미 32 비트로 확장이 됩니다. 이후 이 값을 그냥 쓰면 8비트 영역만 표현할 수 있지만 Rotate 필드값을 * 2 한 만큼 Rotate Right 시킨다.

예) SUBS r4, r5, r7 LSR r2 ; r4:= r5 - (r7 >> r2)

List 2-1. Operand 2에 대한 설명(Continued)

2.4.2. Data Processing 의 명령어

아래의 Table 1-5는 Data Processing 의 명령어를 보여 주고 있다.

Opcode	Mnemonic	Instruction	Action
0b0000	AND	AND	Rd:= Op1 AND Op2
0b0001	EOR	Exclusive OR	Rd:= Op1 XOR Op2
0b0010	SUB	Subtract	Rd:= Op1 - Op2
0b0011	RSB	Reverse Subtract	Rd:= Op2 - Op1
0b0100	ADD	Add	Rd:= Op1 + Op2
0b0101	ADC	Add with carry	Rd:= Op1 + Op2 + C
0b0110	SBC	Subtract with carry	Rd:= Op1 - Op2 + C - 1
0b0111	RSC	Reverse Subtract with carry	Rd:= Op2 - Op1 + C - 1
0b1000	TST	Test bits	Op1 AND Op2 -> CPSR
0b1001	TEQ	Test bitwist equality	Op1 XOR Op2 -> CPSR
0b1010	CMP	Compare	Op1 - Op2 -> CPSR
0b1011	CMN	Compare Negative	Op1 + Op2 -> CPSR
0b1100	ORR	OR	Rd:= Op1 OR Op2
0b1101	MOV	Mov register or constant	Rd:= Op2
0b1110	BIC	Bit Clear	Rd:= Op1 AND (NOT Op2)
0b1111	MVN	Move negatice register	Rd:= NOT Op2

Table 1-5. Data Processing 명령어

2.4.2. Data Processing 의 명령어 (Continued)

아래의 list 2-2는 Data Processing의 명령들에 대해 설명하고 있다.

1. MOV 명령

1) MOV{cond}{S} Rd,Op2

어셈블리어에서 가장 기본적으로 사용되는 데이터 전송명령입니다. 해당 명령은 8086에서의 MOV, Z80이나 196에서의 에서의 LD 명령과 같은 역할을 합니다. 다만 차이라면 ARM7에서의 MOV는 그 타겟이 되는 부분이 항상 레지스터 라는 점입니다. 예를 들어 8x86에서는 MOV AX,[BX] 의 형태와 MOV AX,BX 와 같은 명령의 형태를 같은 MOV명령을 통해 해결합니다만, ARM에서의 MOV는 후자의 것만을 의미합니다. 즉, 레지스터나 Immediate 값 같은 것들을 레제스터에 넣는 명령입니다. 전자의 경우와 같이 외부 메모리를 억세스하는 명령은 따로 있고, 분류 자체를 다르게 합니다.

MOV명령의 형식중에서 {cond}부분은 실행 조건을 제한하기 위한 조건 옵션입니다. ARM7에서는 모든 명령어에 해당 옵션을 적용시켜 해당 조건에 맞오록 플래그 레지스터가 설정이 되어 있어야 MOV 멸령이 수행됩니다.

{S}옵션은 해당 명령의 결과가 플래그 레지스터에 영향을 미칠지 여부를 결정하는 옵션입니다. ARM에서의 MOV는 Data Processing 명령으로 분류되어 Rd에 들어가는 값에 따라 플래그 레지스터가 영향을 받을 수 있습니다. 하지만, 특별히 사용되는 경우가 있습니다. 타겟 레지스터가 PC 인 경우 S 옵션을 사용하면, Exception모드에서 보통상태로 빠져 나오는 역할을 하게 됩니다.

Rd는 타겟이 되는 레지스터 입니다. MOV명령의 타겟은 항상 레지스터이어야 합니다. 물론 MOV 뿐만이 아니라 Data Processing 명령으로 분류된 모든 명령들이 다 그렇습니다.

Op2는 MOV명령에 의해 Rd로 들어갈 내용을 가리키는 오퍼랜드입니다. Op2 부분에는 단 순히 레지스터가 들어갈 수도 있고, 어떤 immediate 값이 들어갈 수도 있고... 또 Shift 오퍼레이션을 가한 레지스터 내용이 들어갈 수도 있습니다.

list 2-2. Data Processing의 명령들

2.4.2. Data Processing 의 명령어 (Continued)

예1) MOV r0,r1
: 단순히 r0:=r1 을 하는 명령입니다.

예)) MOV r0,#0
: r0에 상수 0을 넣는 명령입니다. ARM에서는 196에서와 같이 상수에는 #을 붙여서 사용합니다. 또 진법 표현은 C 에서의 방법 을 사용합니다. 즉 MOV r0,#0x30 이런식으로 사용할 수도 있습니다. 혹은 &를 붙여서 16진수를 나타낼 수도 있습니다.

예3) MOV r0,#0xfc000003
: r0에 상수값 0xfc000003 을 넣는 명령입니다. 해당 값은 8비트 값 0xFF를 32비트로 확장하고 오른 쪽으로 두번 Rotate 시킨 값입니다. 그래서 에러가 나지 않는 것이지요.

예4) MOV r0,r1,LSL #1
: 이 명령은 r0:= r1 <<1 을 수행하는 명령입니다. LSL은 logical Shift Left 를 의미하는 키워드이고, Op2 부분에 위와 같은 형식으로 올 수있습니다.

예5) MOV r0,r1,LSR r2
: 이 명령은 r0:=r1 >> r2 에 해당하는 명령이고, LSR은 Logical Shift Right를 의미합니다.

예6) MOV r0,r0,ASR #24
:이 명령은 r0:=r0 >> 24 를 의미합니다. ASR은 Arithmetic Shift Right 를 의미하며 LSR과의 차이는 최 상위 비트가 1 인경우 새로 계속 해당 비트값을 유지시킨다는 것입니다.

```
예7) MOVS   r0,r1,LSR #1    ; C(flag) := r1[0]
     MOVCC  r0, #10         ; if C=0 then r0:=10
     MOVCS  r0, #11         ; if C=1 then r0:=11
예8) MOVS   r0,r4           ; if r4=0 then r0=0
     MOVNE  r0,#1           ; else r0=1
```

:첫번째 명령에서 r4값을 r0로 넣을 때 {S}옵션에 따라서 플래그 레지스터 값이 변화합니다. 다음 명령에서 NE 조건을 사용했기 때문에, 참고로 NE는 Not Equal, Z플래그가 0일 경우입니다., r4값이 0이었다면 NE는 False가 되어 두번째 명령을 수행하지 않습니다. 결과적으로는 r0에 0이 들어간 셈이지요. 만약 r4가 0이 아니었다면 두번째 명령이 수행되어서 r0에는 1이 들어갑니다.

list 2-2. Data Processing의 명령들(Continued)

2.4.2. Data Processing 의 명령어 (Continued)

2. MVN 명령

1) MVN{cond}{S} Rd,Op2

해당 명령은 Rd:=NOT Operand2 의 의미를 가진 명령입니다. 기존 어셈블러에서는 볼 수 없었던 명령입니다. MOV처럼 값을 넣는데 NOT를 해서 넣는 명령입니다. 이런 명령이 ARM7에서 8비트Resolution Immediate 오퍼랜드을 위해 사용되고 있습니다.

예1) MVN r0,#0 ; r0 := -1
: MOV r0,#0xFFFFFFFF 이렇게 하면 에러가 발생합니다. ARM7의 모든 명령이 32비트 한 워드이고 Immediate 오퍼랜드처리를 8비트 값을 Rotate시키는 방식으로 사용하기 때문인데, MVN을 사용함으로서 해당 값을 넣을 수 있습니다.

3. ADD/ADC 명령

1) ADD{cond}{S} Rd, Rn, Op2
ADC{cond}{S} Rd, Rn, Op2
더하기 명령입니다. 8086과 큰 차이는 오퍼랜드를 3개 지정한다는 것입니다. Rd는 결과가 저장될 레지스터로, Data Processing 명령 모두는 그 결과가 레지스터(Rd)로 들어가야만 합니다. 첫번째 Rd가 결과가 저장 될 레지스터이고, 나머지 두개가 서로 더해질 오퍼랜드입니다.
두개의 오퍼랜드 중 하나는 또 항상 레지스터여야만 합니다. Op2는 Shifted Register 혹은 Rotated Immediate Value 중 하나입니다.

ADD와 ADC는 널리 사용되는 대로 더하기, 더하는데 캐리플래그랑 같이 더하기... 정도입니다.

예1) ADD r0,r0,#1 ; r0:=r0+1

list 2-2. Data Processing의 명령들(Continued)

2.4.2. Data Processing 의 명령어 (Continued)

예2) ADD r0,r1,r2 ; r0:=r1+r2

예3) ADDS r0,r1,r1, LSL #2 ; r0:=r1*5
: 위에서 Rd는 r0, Rn은 r1, Op2 부분이 r1,LSL #2 입니다. r1값 더하기 r1을 두번 쉬프트 한 값, 즉 r1*4값이므로 결과적으로는 *5가 되 겠지요... S 옵션이 사용되었으므로 플래그 레지스터에 영향을 미칩니다. 위의 1번 2번 예제는 수행하더라도 플래그 레지스터는 영향을 주지 않습니다.

4. SUB/SBC/RSB/RSC 명령

1) SUB{cond}{S} Rd, Rn, Op2 / SBC{cond}{S} Rd, Rn, Op2
RSB{cond}{S} Rd, Rn, Op2 / RSC{cond}{S} Rd, Rn, Op2

위의 네 명령은 빼기 명령입니다. SUB와 SBC는 Operand 1(Rn)에서 Op2를 빼는데, R로 시작하는 두개가 Op2에서 Operand 1(Rn)을 뺍니다.즉, SUB 명령이 Rd=Rn-Op2 라면 RSC는 Rd=Op2-Rn 입니다. SBC와 RSC는 캐리플래그가 관련되어 있는 명령입니다. 그런데 명령 해석을 보면 다음과 같이 나와 있습니다. SBC Rd, Rn, Op2 = Rd=Rn-Op2+Carry-1이므로 2의 보수로 뺄셈연산을 하므로 자리 내림이 발생을 하면 캐리가 클리어 되는 구조입니다. 즉 1을 빼고 싶은 경우 1의 2의 보수를 취해 더해주죠. 해당 값은 FFFFFFFF 이고, 대상이 0이 아니라면 더하기를 통해 캐리가 발생하겠군요. 근데 원래 하고자 했던것이 뺄셈이었으니까...피감수가 0인 경우에 캐리가 발생하지 않았고, 나머지의 경우에는 모두 캐리가 발생합니다. 어떻든 중요한건 뺄셈 연산을 통해 borrow가 발생한다면 Carry는 0이고 발생하지 않는다면 Carry는 1이 되는 것이죠. 따라서 SBC와 같은 명령에서는 Carry를 오히려 더해주고 다시 1을 빼 줌으로써 8x86 등에서 사용되는것과 동일한 효과를 얻게 됩니다.

예) sub r0, r4, r6 ; r0 = r4 - r6

list 2-2. Data Processing의 명령들(Continued)

2.4.2. Data Processing 의 명령어 (Continued)

5 AND/EOR/ORR 명령

1) AND{cond}{S} Rd, Rn, Op2
   EOR{cond}{S} Rd, Rn, Op2
   ORR{cond}{S} Rd, Rn, Op2

: 이번에는 Bit연산에 관련된 명령어들입니다. AND는 그대로 AND,EOR은 XOR쯤으로, ORR은 OR로 생각하시면 됩니다. B 명령을 제외하고는 웬만한 명령어는 모두 3글자인데, 그래서 ORR로 표기하는 듯합니다. 역시 오퍼랜드는 같은 의미입니다.

예1) AND r0,r0,#0xFF : r0의 8비트만 남기는 명령입니다.

예2) ANDCSS r0,r1,r2,ASR r3
: 만약 캐리 플래그가 설정되어 있다면...(CS) r1 AND (r2>>r3) 정도를 하는데, 해당 명령의 결과가 플래그 레지스터에 영향을 미칩니다. ASR과 LSR의 차이는 ASR의 경우 최 상위 비트가 1이라면 해당 비트는 계속 1로 유지를 하는 것이죠.LSR이라면 새로 들어 오는 비트는 항상 0입니다. 이것은 음수를 2의 보수로 사용하는 시스템에서 필요한 방법입니다. 참 결과는 r0에 넣습니다.

예3) EORS r0,r0,r0 : 해당 명령을 수행하면 r0는 0이 될 것이고, N 플래그도 0이 되고 Z플래그는 1이 됩니다.

예4) MOV r0,#0xFF
     ORR r0,r0,#0xFF00
   : r0에 0xFFFF를 넣는 방법입니다.

list 2-2. Data Processing의 명령들(Continued)

2.4.2. Data Processing 의 명령어 (Continued)

6. BIC 명령

1) BIC{cond}{S} Rd, Rn, Op2

: Rd := Rn AND (NOT Op2) 로 설명이 되어 있는 명령입니다. 별별 명령이 다 있다는 생각이 드는군요. AND하기전에 NOT을 한번 해서 처리하는 명령입니다. 의미는 비트를 클리어하게 됩니다. Op2에 지정한 비트가 0으로 지워지는 결과가 나타납니다.

예) BIC r0,r1,#3
: r0 := r1 and 0xFFFFFFFC 의 의미입니다.

7. TEQ/TST 명령

1) TEQ{cond} Rn, Op2/TST{cond} Rn, Op2

: CMP와 거의 비슷한 구조를 갖는 명령입니다. 기능도 거의 유사합니다. CMP가 -, CMN이 + 라면, TEQ는 XOR, TST는 AND 연산을 통해 같은 일을 합니다. 하지만 특징이라면 Logical 연산이 일어날 경우 플래그중에서 V 플래그(오버플로우 플래그)는 영향을 받지 않습니다. 이걸 잘 응용하면 다음과 같은 처리가 가능합니다.

```
예1)  CMP    r0,#31          ; test r0<=31 ?
      TEQ    r0,#127         ; test r0==127 ?
      MOVLS  r0,#'.'         ; if either then r0='.'

예2)  TST    r1,#3           ; is r1 word aligned?
      MOVEQ  r0,#1           ; r0:=1 if so
      MOVNE  r0,#0           ; else r0:=0
```

list 2-2. Data Processing의 명령들(Continued)

2.4.2. Data Processing 의 명령어 (Continued)

8. CMP/CMN 명령

1) CMP{cond} Rn, Op2
   CMN{cond} Rn, Op2
: 비교 명령입니다. MOV, MVN처럼 인수가 2개인데요, 이번에는 Rd가 있지 않습니다. CMP는 흔히 보던 그 CMP입니다. Rn에서 Op2를 빼 보는 명령이죠. 아시다시피 레지스터값에는 영향을 미치지 않고 플래그에만 영향을 미칩니다. 그러고보니 {S}옵션도 없네요... 이 명령들은 디폴트로 {S}를 준 효과가 나타납니다.

CMN은 좀 새롭군요. 이 명령은 CMP가 빼기를 하는데 반해, 두 오퍼랜드를 더해 보는 명령입니다.

```
예1) CMP     r2,#23
     MOVEQ   r2,#45    : 만약 r2가 23이라면 45를 넣어라...

예2) CMP     r0,#0
     CMPEQ   r1,#0
     CMPEQ   r2,#0
     CMPEQ   r3,#0
     MOVEQ   r4,#12   : r0부터 r3까지 모두 0이라면 r4에 12를 넣는 명령입니다.

예3) CMN     r1,r2
     MOVEQ   r0,#0
     MVNNE   r0,#1    : r0 = ((r1+r2)==0) ? 0:-1;
```

list 2-2. Data Processing의 명령들(Continued)

2.4.3. Single Data Transfer 명령어

아래의 list 2-3은 Single Data Transfer 명령에 대한 설명이다.

1. LDR/STR 명령

해당 명령은 레지스터와 외부 메모리와의 데이터 전송을 담당하는 명령입니다. 무척 사용빈도가 높은 명령입니다. 8x86에서는 MOV하나로 레지스터간의 데이터 전송과, 외부메모리와의 데이터 전송의 두가지 목적으로 사용하지만, ARM7 에서는 구분이 되어있습니다. 즉, MOV 명령은 레지스터간의 전송명령으로 분류도 되고, 데이터 처리명령으로도 분류됩니다.하지만 ARM에서는 데이터 전송명령으로는 LDR과 STR 이 있습니다. 여기에 사용되는 옵션이 많은데요, 우선, LDR은 로드라는 의미이므로 외부 메모리에서 레지스터로 데이터를 읽어오는 명령이고, STR은 반대의 기능을 하는 명령입니다. 아래의 Table 2-6은 LDR과 STR에서 사용하는 전송모드를 나타냅니다.

1) LDR{cond}{B} Rd, address{!} ; Rd:= contents of address
   LDR{cond}{B} Rd, =expression ; Rd:= expression
   STR{cond}{B} Rd, address{!} ; contents of address := Rd

Mode	Effective address	Indexing
[Rn]	Rn	none
[Rn, ±expression]	Rn, ± expression	Pre-indexed
[Rn, ± Rm]	Rn, ± Rm	Pre-indexed
[Rn, ± Rm, shift cnt]	Rn, ± (Rm shift by cnt)	Pre-indexed
[Rn], ± expression	Rn	Post-indexed
[Rn], ±Rm	Rn	Post-indexed
[Rn], ±Rm, shift cnt	Rn	Post-indexed

Table 2-6. LDR과 STR에서 사용하는 전송모드

list 2-3. Single Data Transfer 명령들

2.4.3. Single Data Transfer 명령어(Continued)

LDR과 STR의 경우 전송 단위를 바이트, 혹은 워드(32bit)단위로 수행할 수 있습니다. 바이트 단위 전송의 경우 해당 레지스터의 어떤 부분이 사용될지의 여부는 해당 프로세서의 Endian에 달려 있습니다. 또, 워드 전송 명령을 사용할 경우, 메모리 어드레스는 word align이 되어야 합니다.

이 명령들은 list2-2처럼 Index 모드에 따라 명령형식이 차이가 납니다. 각 각의 모드에 대해 예를 들어 설명하겠습니다.

1) Pre-Indexed Addressing Mode

- 어드레스를 지정하기 위해서 Rn을 베이스 번지로 사용합니다.
- 베이스 번지에 옵셋을 더하거나 뺄 수 있습니다.
- 옵셋으로 부호가 있는 12Bit Immediate값을 사용하거나 레지스터를 지정해서 해당 값을 사용할 수 있습니다.
- 레지스터를 옵셋으로 사용하는 경우는 Shift을 이용한 형태을 사용할 수 있습니다.

예1) LDR r0, [r1] : r1을 베이스로 사용하고 옵셋은 지정하지 않은 형태입니다. r0에 r1을 번지로하는 워드(4바이트)를 읽어 드립니다.

예2) LDR r0, [r1, #132]: 옵셋으로 132를 지정한 형태입니다.

예3) STR r0, [r1, r2]: 베이스 번지는 r1이고, 옵셋은 r2을 사용합니다. r0의 값을 r1+r2의 어드레스에 넣게 됩니다.

예4) LDR r0, [r1, r2, LSL #2]:베이스 번지는 r1이고, 옵셋은 r2<<2입니다. r0에 r1+(r2<<2)의 어드레스에 있는 값을 넣습니다.

list 2-3. Single Data Transfer 명령들(Continued)

2.4.3. Single Data Transfer 명령어(Continued)

Write-Back이란?
베이스 번지를 담고 있는 레지스터에 옵셋 값 더한 후에 다시 이 레지스터로 값을 넣는 기능입니다.
LDR명령의 경우 옵션으로 Write-Back을 지정할 수 있습니다. '!'기호를 사용하여 나타냅니다.

```
예)             LDR   r0, =table_end
                LDR   r1, =table
                MOV   r2, #0
        loop    STR   r2,[r0, #-4]!
                ADD   r2, r2, #1
                CMP   r0, r1
                BNE   loop
                .. 중간 생략 ..
                ALIGN
        table   %     table_length*4
        table_end
```

예를들어 1000번지부터 2개의 entry를 가정한다면, table은 1000이고 table_end은 1008이 될 것이다. 이때 STR명령에서 1008을 베이스 번지로 하고, -4를 옵셋으로 설정했으므로 실제로 r2값은 1004번지부터 4바이트를 값을 읽어 기록하게 되고, 이 후에 Write-Back 옵션이 적용되어 해당 명령이 끝나면 r0에 1004로 값이 변경됩니다.

2) Post-Indexed Addressing Mode

- Rn은 항상 베이스 어드레스를 나타내는 레지스터 값이다.
- 기본적으로 Write-Back 기능을 사용합니다.

list 2-3. Single Data Transfer 명령들(Continued)

2.4.3. Single Data Transfer 명령어(Continued)

예1) LDR r0, [r1], r2: 대괄호의 사용이 Pre-Indexed의 경우와 다릅니다. 실제 수행되는 것은 r0에다 r1을 베이스 번지로 하는 값을 읽어와서 저장하고, r1에는 r2를 더해준다.
1. r0 ← [r1]
2. r1 ← r1 + r2

예2) STR r0, [r1], #20 : [r1] ← r0, r1 ← r1 + 20

예3)
```
        LDR     r0, table
        LDR     r1, =table_end
        MOV     r2, #0
loop    STR     r2, [r0], #4
        ADD     r2, r2, #1
        CMP     r0, r1
        BNE     loop

        .. 중간 생략 ..

        ALIGN
table   %       table_length*4
table_end
```

예를들어 1000번지부터 2개의 entry를 가정한다면, table은 1000이고 table_end은 1008이 될 것이다. 이때 STR명령에서 1000을 베이스 번지이므로 r2값은 1000번지부터 읽은 4바이트를 기록하게 되고, r0 레지스터에 있는 값에 4를 더해 1004번지가 된다.

list 2-3. Single Data Transfer 명령들(Continued)

2.4.3. Single Data Transfer 명령어(Continued)

3) Relative Addressing Mode

- 이 모드는 어셈블러가 적절히 지원하여 변환해 주는 모드이다.

```
예1)            LDR     r5, ThreeCubed

                .. 중간 생략 ..

    ThreeCubed  DCD     27
```

: LDR r5, [PC, #constant]형태의 코드로 변역한다. PC를 베이스 번지로 하여 코드를 생성한다. 만약에 해당 심볼이 지정하는 범위가 너무 커서 상수도 지정할 수 없다면 어셈블을 할 때 에러를 내게 된다.

```
예2) LDR  r0, =12345678
```

: 어셈블러가 해당 상수값을 Literal Pool이라는 특정공간에 넣어 명령 코드를 생성하게 된다. 이때로 PC를 베이스 번지로 사용하게 됩니다.

list 2-3. Single Data Transfer 명령들(Continued)

2.4.4. Block Data Transfer 명령어

아래의 list 2-4는 Block Data Transfer 명령에 대한 설명이다.

1. LDM/STM 명령

LDR명령은 메모리 번지의 내용을 지정된 레지스터로 가져 오는 명령이라면 LDM은 여러개의 레지스터의 내용을 한 번에 가져 오는 명령입니다. 베이스 레지스터(Rn)로 지정된 번지에서 레지스터 목록으로 지정된 각 레지스터의 내용을 읽어들이는 명령이고 STM은 LDM과 반대의 명령이다. 일반적으로 스택 연산에 많이 사용되고 있으며, 엄밀히 말하면 블럭 전송이 명령이 아니라 Multiple Register 전송명령 입니다.

1) LDM{cond}mode Rn{!}, {reg_list}{^}
   STM{cond}mode Rn{!}, {reg_list}{^}

위 명령에서 {cond}는 늘 보아오던 명령어 실행 조건이고, mode라고 되어 있는 부분은 여러개의 레지스터를 메모리에 넣거나 가져올 때 어떤 방식으로 동작할지를 지정합니다. mode의 종류는 4가지 입니다. Rn은 전송에 사용될 베이스 번지를 지정하는 레지스터이며, '!'를 붙이면 Write-Back 기능이됩니다. {reg_list} 부분은 전송하거나 전송 받을 레지스터의 목록을 나타내는 부분입니다. 예를들면 STMEA sp!, {r0,r1,r2,r3}이다. 이처럼 중괄호 사이에 전송대상이 되는 레지스터를 넣어주면 됩니다. ARM7에서는 한시점에 넣을 수 있는 레지스터의 개수는 r0에서 r15까지 16개이므로 LDM이나 STM명령에서 지정할 수 있는 레지스터의 개수는 최대 16개 입니다. 즉, 한 명령으로 모든 레지스터를 저장하거나, 가져올 수 있다는 의미입니다. 이 것이 가능한 것은 이들 명령의 instruction format에 16비트의 reg_list 필드가 존재하여 각 비트가 레지스터 r0에서 r15까지 일대일로 대응이 됩니다. 따라서 {reg_list}에는 어떠한 레지스터의 조합도 올 수 있습니다. {r0, r2}의 형식이나 {r0-r5}와 같은 형식, {r0-r3, r6-r7} 이런 형식도 사용할 수 있습니다. reg_list 필드에 각 각에 해당되는 레지스터 비트가 있으므로 레지스터의 순서는 상관없이 나열해됩니다. {r0,r1,r2}나 {r2,r1,r0}는 어셈블러가 같은 것을 취급합니다. {^}은 만약 S bit가 세트되면 privileged mode일때만 PC의 값과 CPSR을 load하게 됩니다.

list 2-4. Block Data Transfer 명령들

2.4.4. Block Data Transfer 명령어(Continued)

LDM이나 STM 명령 모두 Addressing Mode를 4가지씩 가지고 있다. 여기서 Addressing Mod는 여러개의 레지스터값을 메모리로(혹은 메모리로부터) 전송할 경우 해당 메모리 번지를 증가시키면서 저장할지, 혹은 감소시키면서 저장할지를 지정하는 것과, 증/감을 하는데, 저장하기 전에 증/감을 할지, 아니면 저장하고 나서 증/감을 할지를 지정하는 것을 의미합니다. 아래의 Table 2-7은 Adressing Mode Names에 대하여 보여 주고 있습니다.

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 stroe	STMED	STMDA	0	0	0

Table 2-7 LDM/STM Addressing Mode Names

Note

LDM/STM의 뒤에 붙는 접미사의 의미
1) Stack에서 사용할 경우 : 'E'는 Empty, 'F'는 Full, 'D'는 Descending, 'A'는 Ascending이다.
2) Stack에서 사용하지 않을 경우: 'I'는 Increment, 'D'는 decrement, 'B'는 Befroe, 'A'는 After이다.

list 2-4. Block Data Transfer 명령들(Continued)

2.4.4. Block Data Transfer 명령어(Continued)

1> Post-Increment Addressing

이 방식은 값을 저장한 후에 어드레스가 증가합니다. 아래의 Figure 2-5는 Post-Increment Store에 대한 저장 과정을 보여 주고 있다.

예) STMIA Rn, {r1, r5, r7}
Rn이 가리키고 있는 베이스 번지가 0x1000일때

1. 0x1000 번지에 r1이 저장된다.
2. 0x1004 번지에 r5가 저장된다.
3. 0x1008 번지에 r7을 저장된다.
4. 베이스 번지값은 0x100C로 증가한다.

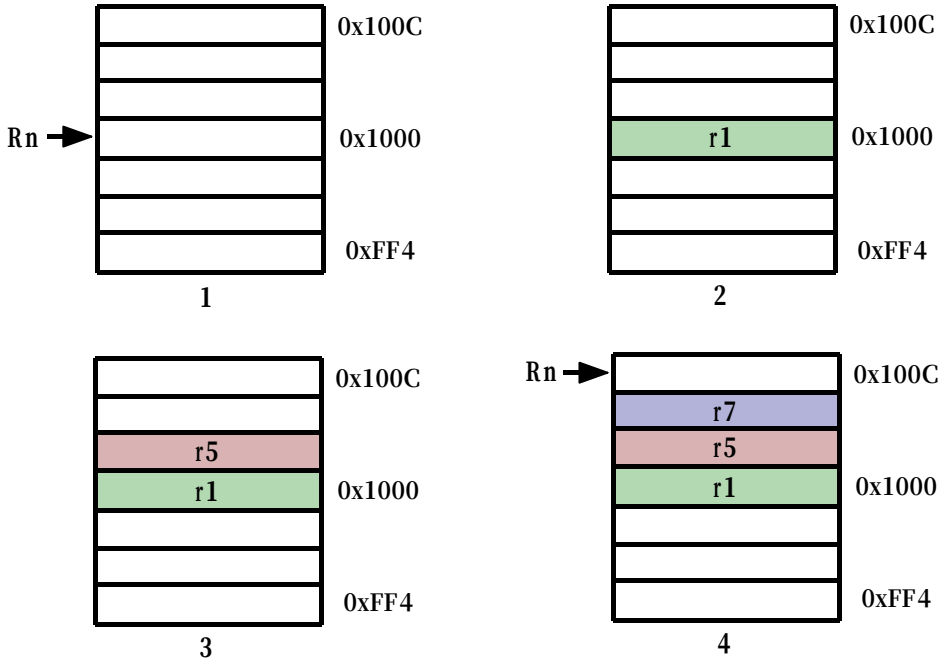

Figure 2-5. Post-Increment Store의 저장 과정

list 2-4. Block Data Transfer 명령들(Continued)

2.4.4. Block Data Transfer 명령어(Continued)

2> Pre-Increment Addressing

이 방식은 먼저 어드레스를 증가하고 다음에 저장합니다. 아래의 Figure 2-6은 Pre-Increment Store에 대한 저장 과정을 보여 주고 있다.

예) STMIB Rn, {r1, r5, r7}
Rn이 가리키고 있는 베이스 번지가 0x1000일때

1. 베이스 번지값이 0x1004로 증가한다
2. .0x1004 번지에 r1이 저장된다.
3. 0x1008 번지에 r5가 저장된다.
4. 0x100C 번지에 r7을 저장된다.

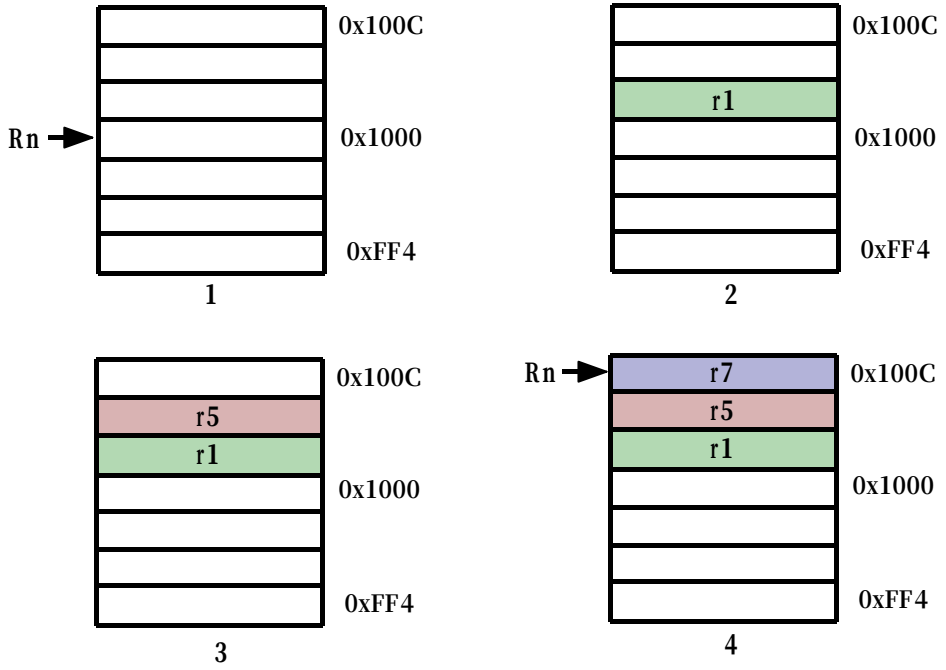

Figure 2-6. Pre-Increment Store의 저장 과정

list 2-4. Block Data Transfer 명령들(Continued)

2.4.4. Block Data Transfer 명령어(Continued)

3> Post-Decrement Addressing

이 방식은 값을 저장한 후에 어드레스가 감소합니다. 증가 모드에서는 레지스터 번호가 작은 것 부터 넣었는데, 감소 모드로 값을 저장할 때는 레지스터 번호가 큰 것 부터 넣게 됩니다. 결과적으로 메모리에 저장되는 레지스터의 순서는 항상 동일하게 됩니다. 아래의 Figure 2-7은 Post-Decrement Store에 대한 저장 과정을 보여 주고 있다.

예) STMDA Rn, {r1, r5, r7}
Rn이 가리키고 있는 베이스 번지가 0x1000일때

1. 0xFF8 번지에 r1이 저장된다.
2. .0xFFC 번지에 r5이 저장된다.
3. 0x1000 번지에 r7가 저장된다.
4. 베이스 번지값은 0xFF4로 감소한다.

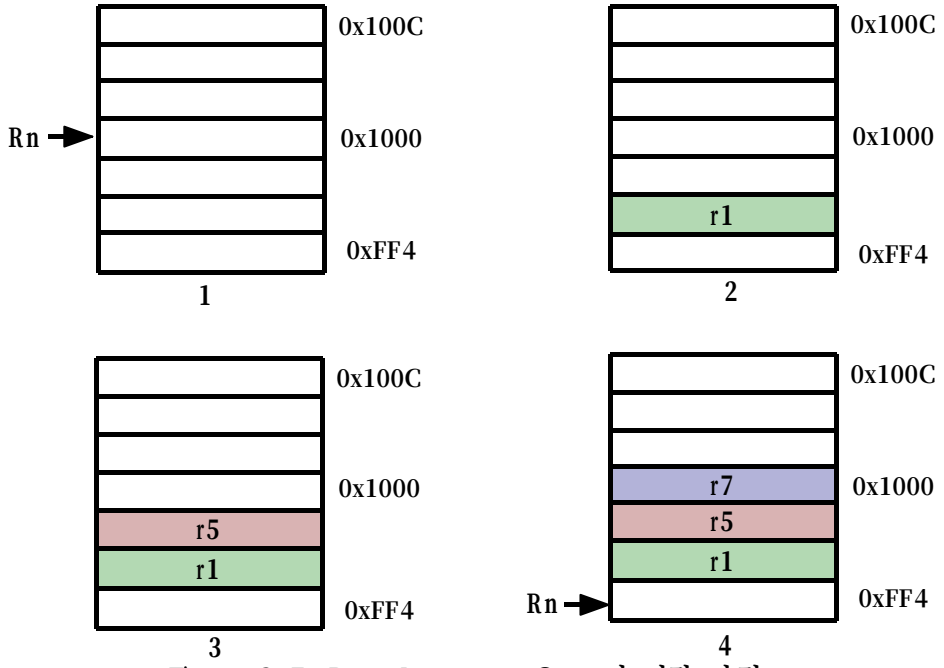

Figure 2-7. Post-Increment Store의 저장 과정

list 2-4. Block Data Transfer 명령들(Continued)

2.4.4. Block Data Transfer 명령어(Continued)

4> Pre-Decrement Addressing

이 방식은 어드레스가 먼저 감소되고 값을 저장합니다. 아래의 Figure 2-8은 Pre-Decrement Store에 대한 저장 과정을 보여 주고 있다.

예) STMDB Rn, {r1, r5, r7}

Rn이 가리키고 있는 베이스 번지가 0x1000일때

1. 0xFF4 번지에 r1이 저장된다.
2. .0xFF8 번지에 r5이 저장된다.
3. 0xFFC 번지에 r7가 저장된다.
4. 베이스 번지값은 0xFFC된다..

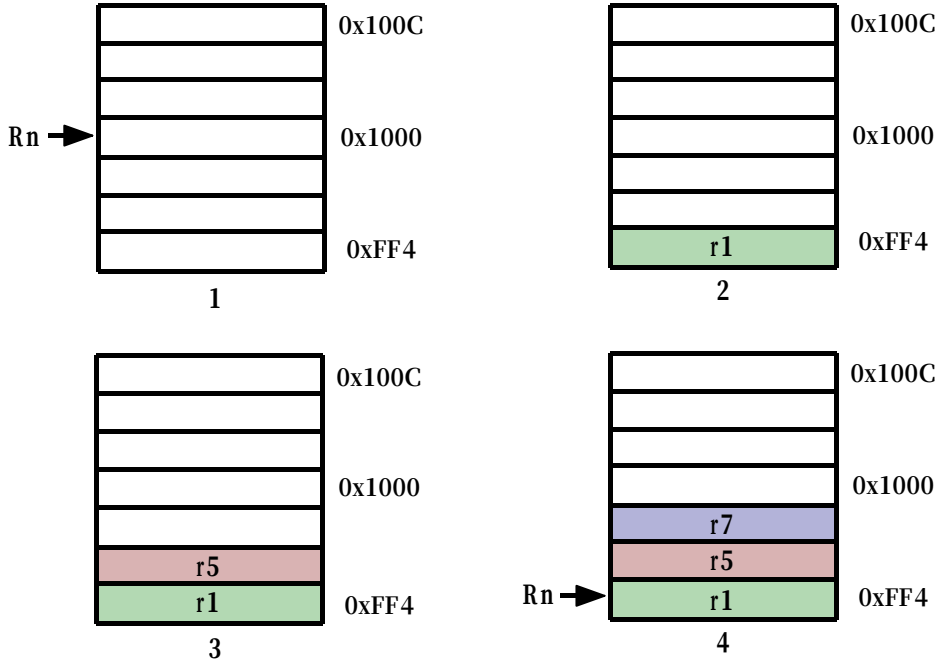

Figure 2-8. Pre-Increment Store의 저장 과정

list 2-4. Block Data Transfer 명령들(Continued)

3.1. ARM7의 Example

1) Data Processing Instruction

```
        mov     r0,   r1
        mov     r0,   #fc000003
        mov     r0,   r1,   lsl #1
        mov     r0,   r1,   lsl  r2

        movs    r0,   r1,   lsr  #1
        movcc   r0,   #10
        movcs   r0,   #11
byteswap :
        mov     r2,   #0xff
        orr     r2,   r2,   #0xff0000
        mov     r3,   r2,   lsl #8
        and     r1,   r2,   r0,   ror  #24
        and     r0,   r3,   r0,   ror  #8
        orr     r0,   r0,   r1
```

2) Procedure call and return

```
        bl      function
        ... ...
function :
        ... ...
        mov     pc, lr
```

3) Conditional compare instructions

```
<in C code>             <in asm code>
if ( a==0 || b==1 )     cmp   r0, r1
c = d + e;              cmpne r1, #1
                        addeqr2,r3,r4
```

4) Loop variables

```
        mov     r0,   #loopcount

loop :
        ... ...
        subs    r0,   r0,   #1
        bne     loop
```