
ARM 아키텍처

- **ARM** 아키텍처
- **Programming** 모델
- **ARM** 프로세서 명령어
- 예외처리와 시스템 리셋
- **ARM9TDMI** 프로세서

ARM 아키텍처: ARM 소개

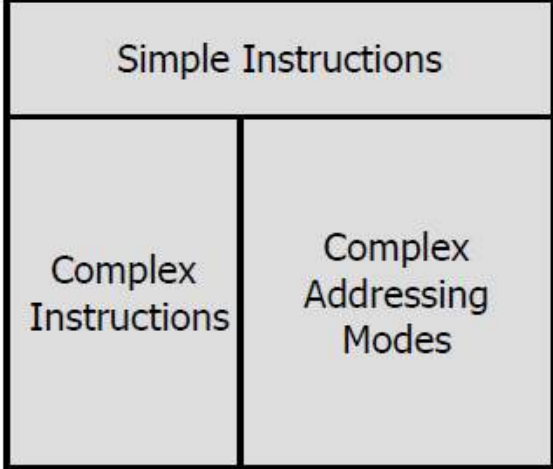
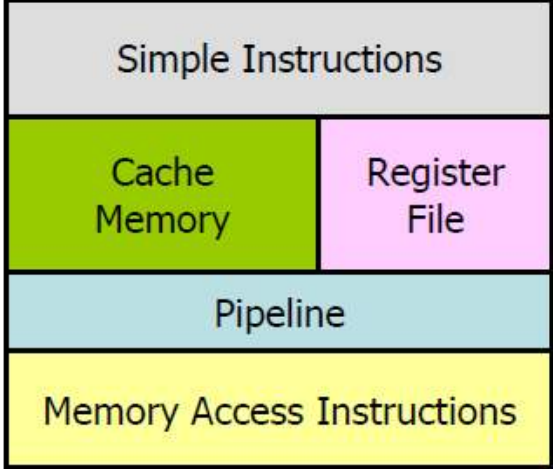
- ARM (Advanced RISC Machines)
 - ❖ 1990년 설립
 - ❖ UK-based joint venture
 - Apple Computer, Acorn Computer Group, and VLSI Technology
- **32-bit RISC Intellectual Property** 제공
 - ❖ ARM은 silicon을 제조 판매하지는 않는다.
 - ❖ Hardware IPs
 - ARM core designs
 - Peripheral IP (Intellectual Property)
 - ✓ *Design Libraries*
 - ❖ Software IPs
 - Power Management / Security Monitor
 - ❖ Development tools : ADS, RVDS
 - Modeling / Design / Debugging
- 현재 70개 이상의 semiconductor 파트너 보유

ARM의 32비트 RISC 프로세서 IP

- **ARM**사는 직접 반도체를 제조하여 판매하는 것이 아니라 설계한 프로세서를 반도체 회사에 **Hard Macrocell** 또는 **Synthesizable core**로 제공
- 반도체 제조회사 또는 SoC 제조사에서는 **ARM**사로부터 제공받은 **ARM core**와 주변장치를 추가하여 **SoC(System on a Chip)**를 만들어 사용자에게 판매 하거나 자체 제품에서 사용

CISC와 RISC 비교

□ ARM cores are basically a RISC...

	CISC	RISC
Components in Cores		
Properties	(+) Short Program Size <ul style="list-style-type: none"> • Less memory requirement • Smaller chip size (+) Less Semantic Gap <ul style="list-style-type: none"> • High level language vs. assembly implementation (-) High Hardware Complexity	(+) Fixed Length Instructions <ul style="list-style-type: none"> • Uniform memory access (+) Large Cache (+) Pipeline (+) Simple Addressing Modes (+) Large Register File (-) Higher Clock Frequency (-) Need for Compiler Support

RISC 개념

❑ Reduced number of instructions

- ❖ Simple instructions
 - Can be executed in a single cycle
 - Complicated operations by combining several simple instructions
- ❖ Fixed-length instructions for making pipeline simple

❑ Pipelines

- ❖ Ideally, one instruction per clock cycle

❑ Registers

- ❖ Large general-purpose register set
- ❖ Act as a fast local memory store for all data processing operations

❑ Load-store architecture

- ❖ Separate load and store instructions
- ❖ Enable memory access optimization (memory access scheduling, out-of-order memory access, etc.)

ARM의 추가적인 개념

□ Ultimate Goal

- ❖ Low power consumption for portable embedded applications
- ❖ Reduction of development costs

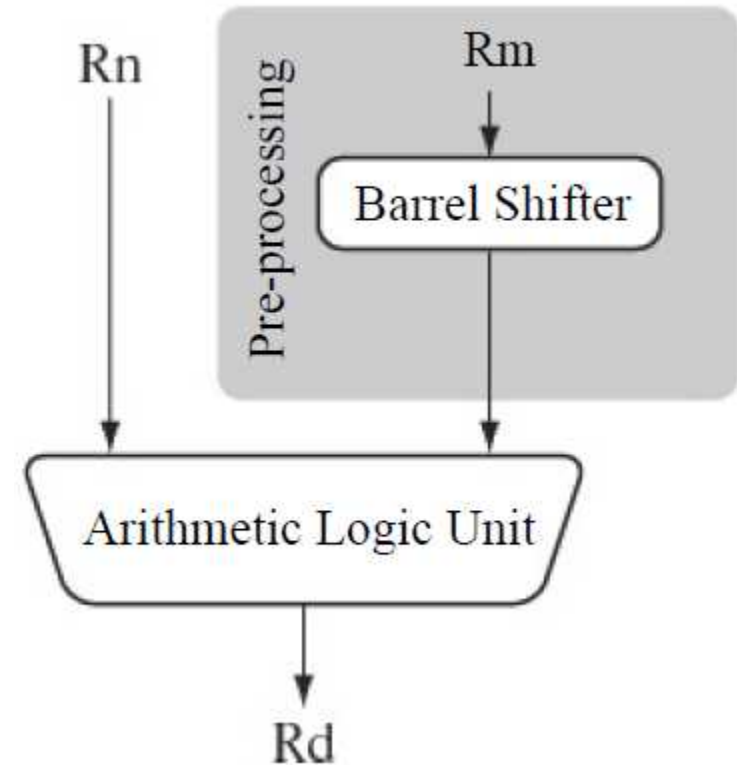
□ Requirements

- ❖ High code density
- ❖ Support for slow and low-cost memory devices
- ❖ Reduction of die area
- ❖ Hardware debug technology

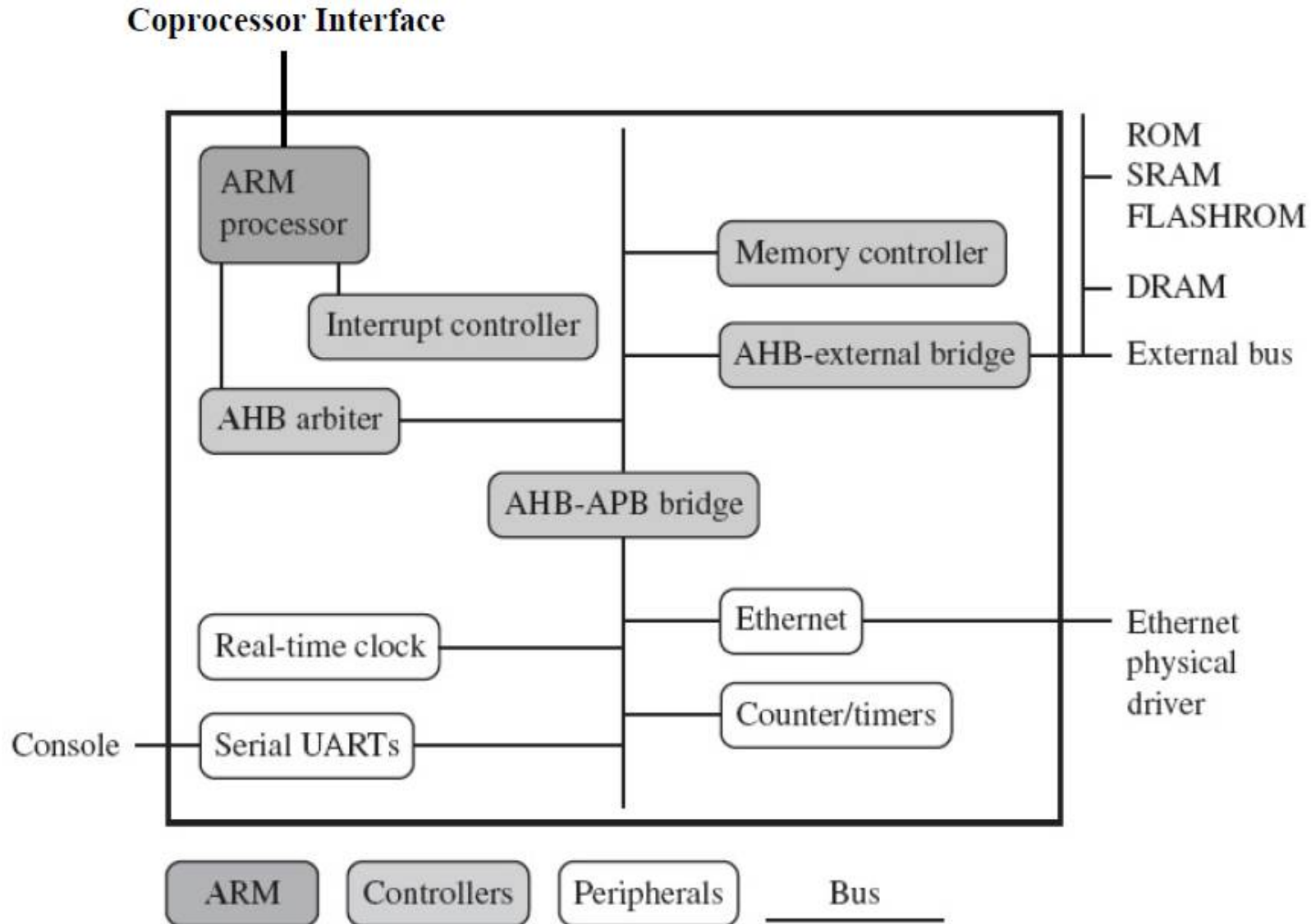
➔ The ARM core is not a pure RISC architecture

Unique Features of ARM Instructions Sets

- ❑ **Variable cycle execution for certain instructions**
 - ❖ Code density is more important
 - ❖ E.g. load-store-multiple instructions
- ❑ **Inline barrel shifter leading to more complex instructions**
 - ❖ Both shift and arithmetic in an instruction
- ❑ **Thumb 16-bit instruction set**
 - ❖ To improve code density
- ❑ **Conditional execution**
 - ❖ To improve pipeline performance
 - ❖ To improve code density
- ❑ **Enhanced DSP instructions**
 - ❖ No need for extra DSP core



ARM-based 임베디드 소자의 예



ARM Architecture

□ Programming Model은 ARM Architecture의 분류 기준

❖ Architecture가 동일하면 Programming model이 동일하다

➤ 프로그램의 호환이 가능하다

❖ Programming Model

➤ 명령어(Instruction Set)

➤ 데이터 구조

➤ 동작 모드

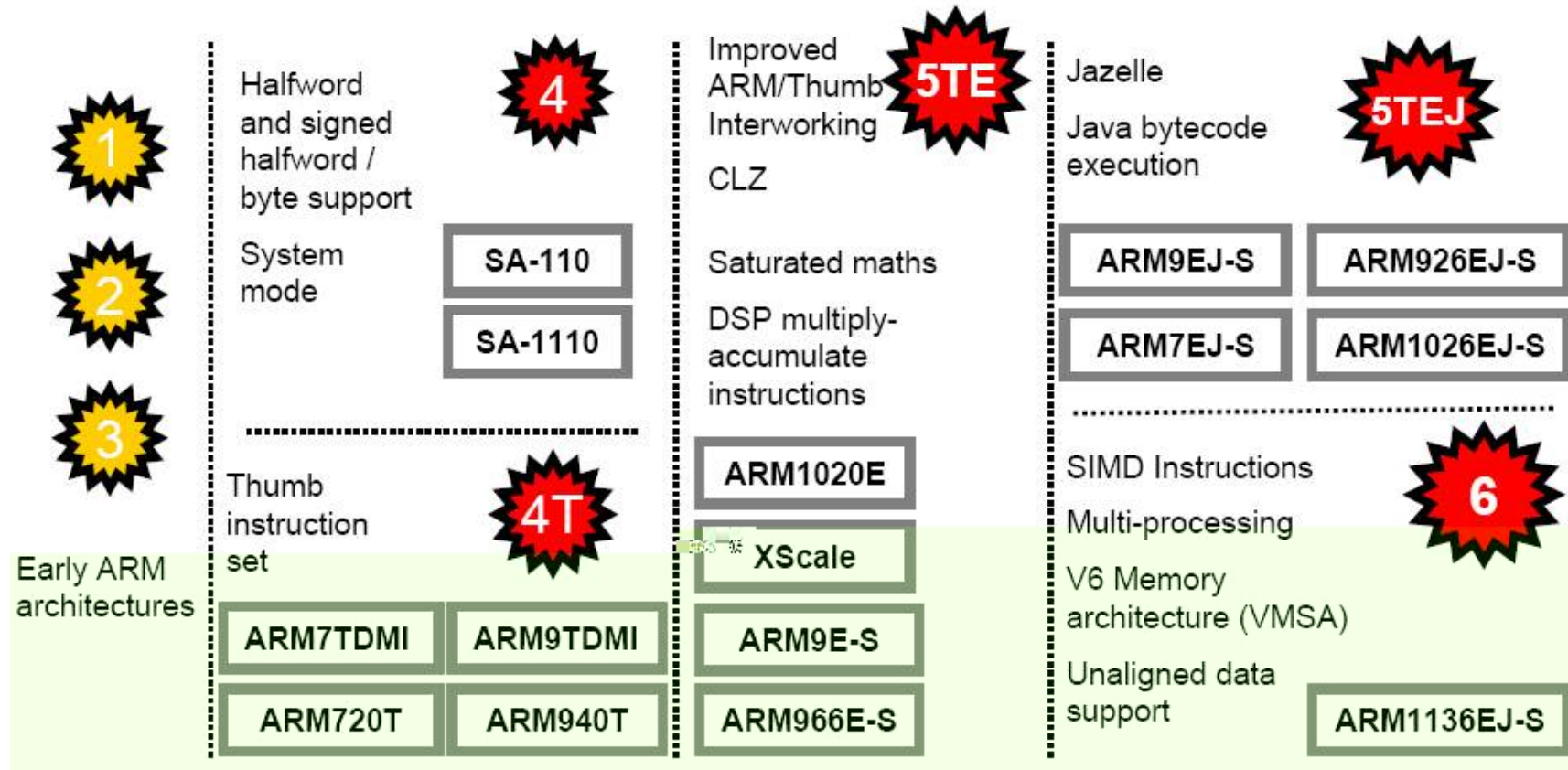
➤ 레지스터의 구조

➤ Exception 처리 방식 등

□ 동일한 **Architecture**에도 여러 개의 프로세서가 있을 수 있다

□ 동일한 프로세서 **Family** 라도 **Architecture**가 서로 다를 수 있다.

ARM 구조의 변천



ARM Architecture 와 프로세서

Architecture	특 징	프 로 세 서
v1, v2, v3	이전 버전, 현재는 거의 사용 않 됨	ARM610, ...
v4	System 모드 지원	StrongARM(SA-1110, ...)
v4T	v4 의 기능과 Thumb 명령 지원	ARM7TDMI, ARM720T ARM9TDMI, ARM940T, ARM920T
v5TE	v4T 의 기능 ARM/Thumb Interwork 개선 CLZ 명령, saturation 명령, DSP Multiply 명령 추가	ARM9E-S, ARM966E-S, ARM946E-S ARM1020E XScale
v5TEJ	v5TE 의 기능 Java 바이트 코드 실행	ARM7EJ-S, ARM9EJ-S, ARM1020EJ-S
v6	SIMD 명령(MPEG4 같은 미디어 처리용) Unaligned access 지원	ARM1136EJ-S, ARM1176JZ
v7	NEON SIMD Dynamic Compiler Support	Cortex-A, Cortex-R, Cortex-M

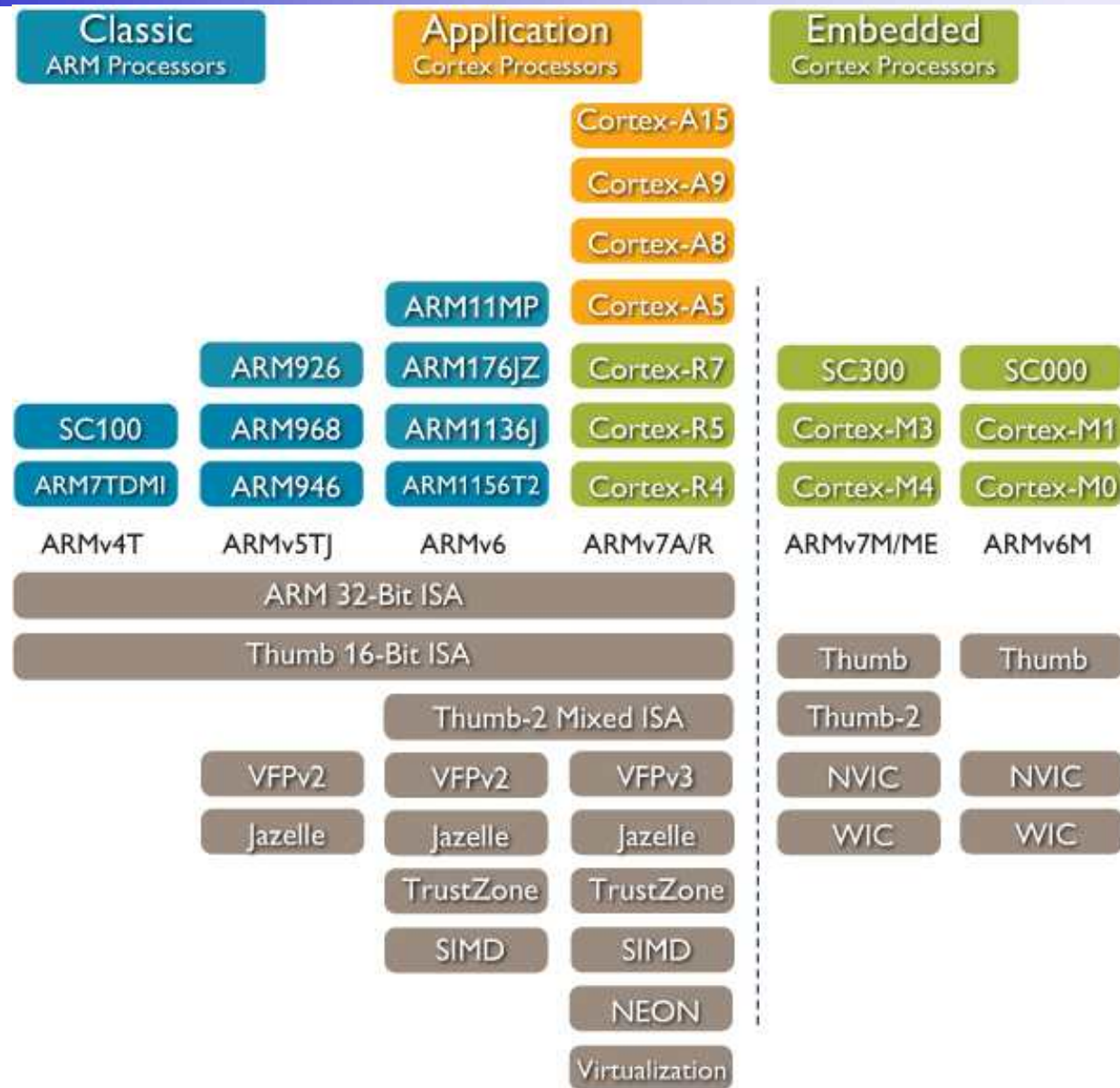
T : Thumb 명령 지원, D : JTAG Debug, M: Fast Multiplier,

I: Embedded ICE macrocell, E : DSP 기능 확장,

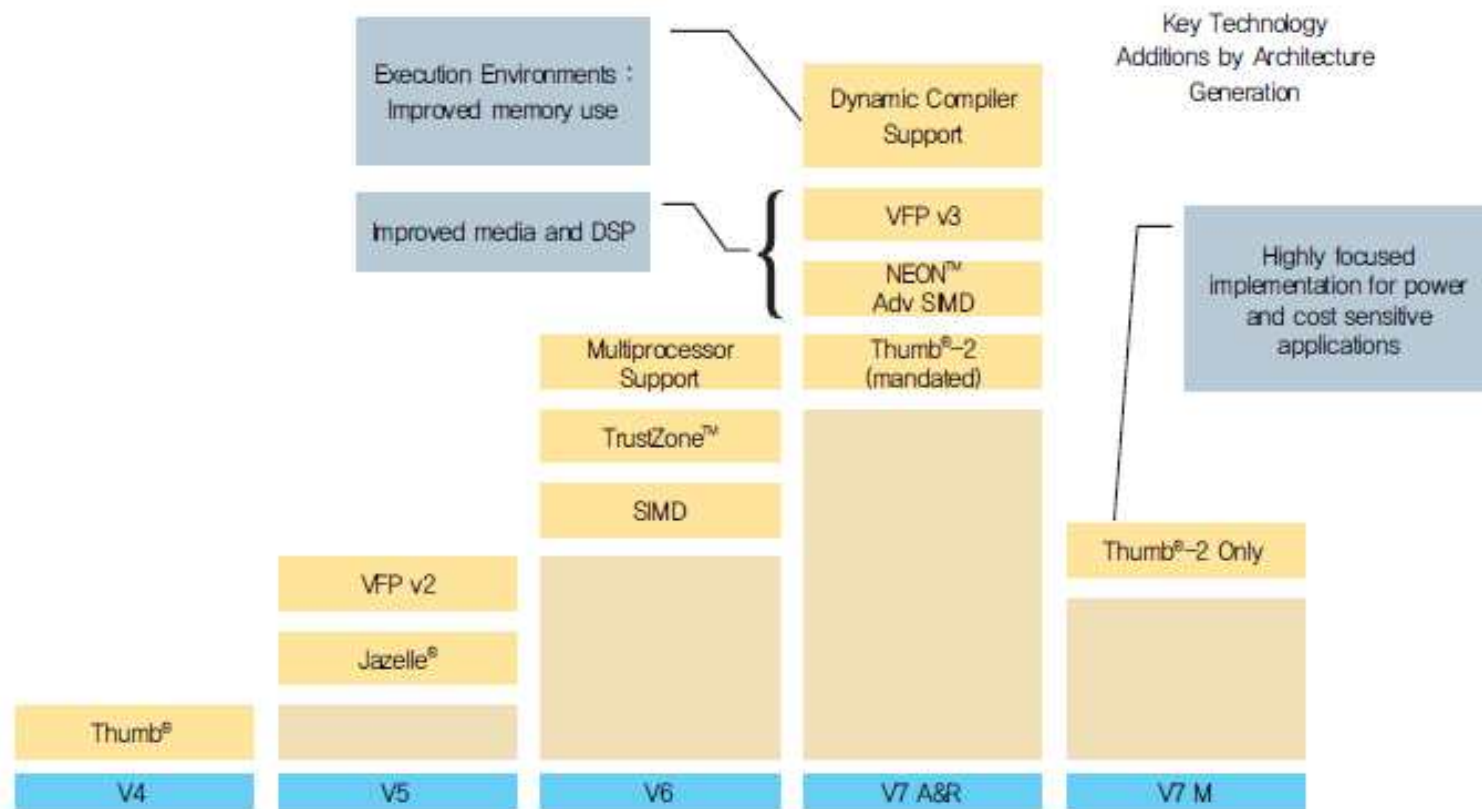
J : Jazelle Java 명령 지원, F: Vector Floating-point unit,

ARMxxx-S : Synthesizable version

ARM architecture overview



ARM Architecture 비교



Cortex-A : Cortex-A5, Cortex-A8, Cortex-A9 MPCore, Cortex-A9 Single Core

복잡한 OS 및 사용자 응용프로그램에 사용

Cortex-R : Cortex-R4 및 Cortex-R4F,

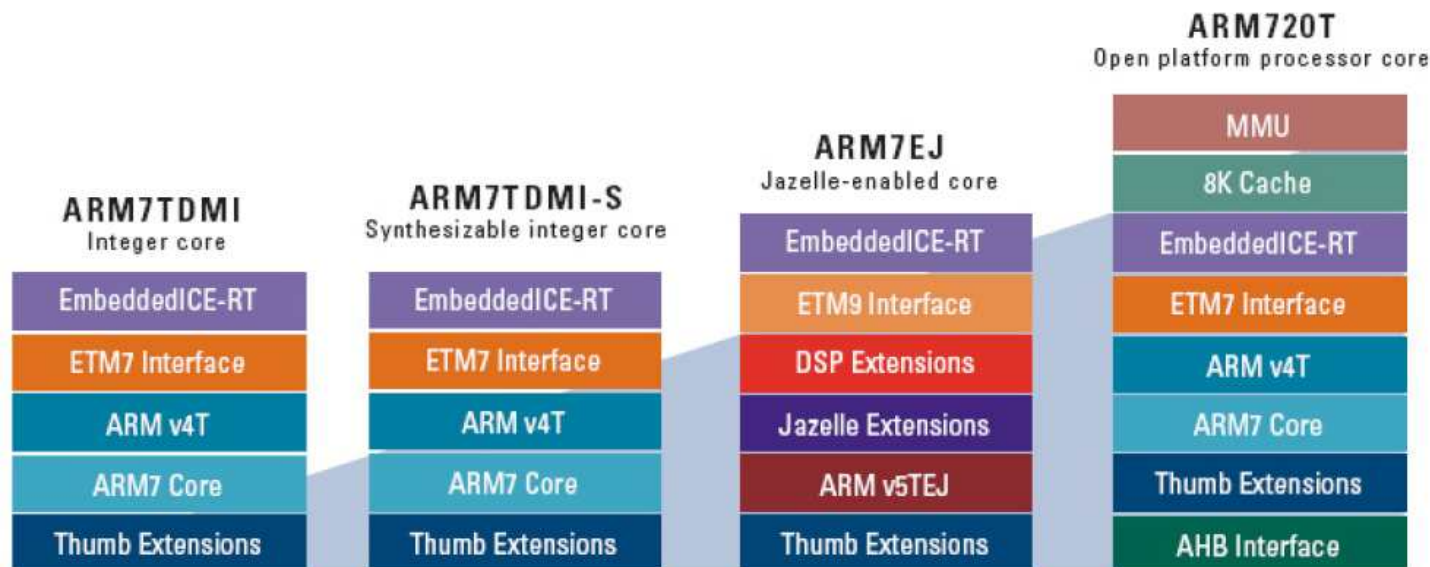
실시간 시스템용 임베디드용

Cortex-M : Cortex-M0, Cortex-M1FPGA, Cortex-M3, Cortex-M4(F)

비용이 중요한 응용프로그램용으로 최적화된 임베디드 수준이 높은 용도

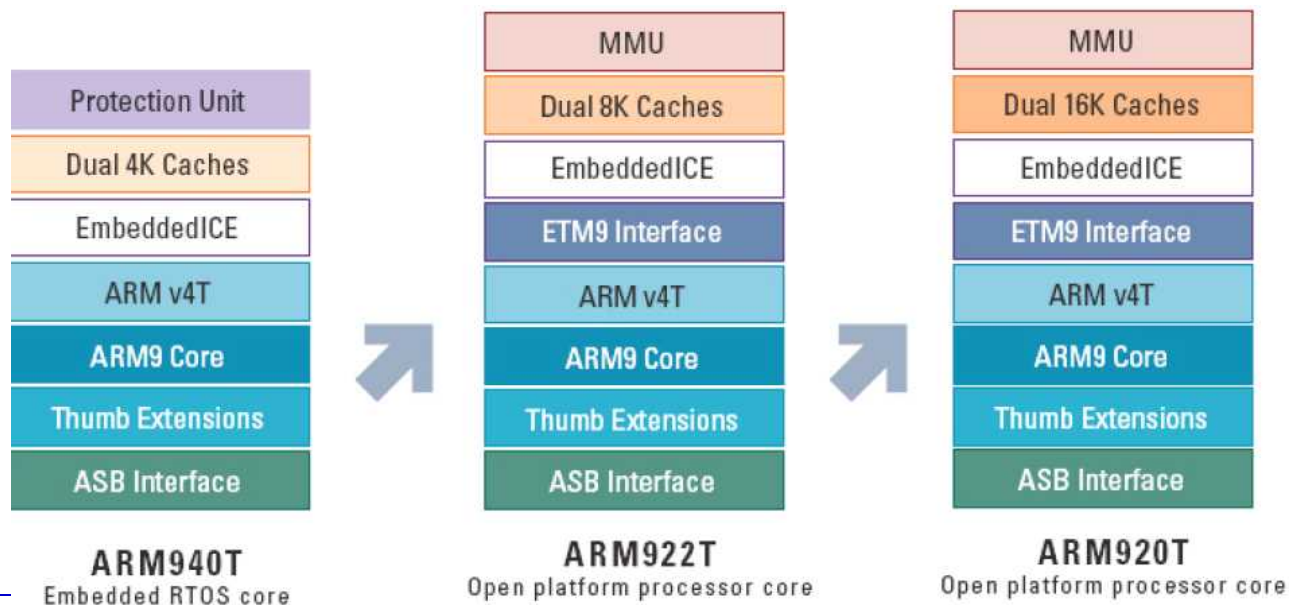
ARM7 Family

- ❑ 32/16-bit RISC Architecture
- ❑ Unified bus architecture
 - ❖ Both instructions and data use the same bus
- ❑ 3 stage pipelining
 - ❖ Fetch / Decode / Execution
- ❑ Coprocessor interface
- ❑ EmbeddedICE-RT support, JTAG interface unit
- ❑ Optional support for MMU (Memory Management Unit)
 - ❖ Easy to port operating systems with virtual addressing capability



ARM9 Family

- ❑ 32/16-bit RISC Architecture
- ❑ Harvard Architecture - Separate memory bus architecture
- ❑ 5 stage pipelining
- ❑ Coprocessor interface
- ❑ EmbeddedICE-RT support, JTAG interface unit
- ❑ Embedded Trace Macro Cell
 - ❖ Trace instruction and data execution in real time on the processor
 - ❖ Useful when debugging applications with time-critical segments



ARM10 Family

- ❑ 6 stage pipeline
- ❑ Parallel load/store unit
 - ❖ Allow computation to continue while data transfers complete
- ❑ 64-bit data bus
- ❑ Optional support for Vector Floating-Point
 - ❖ 7th stage in pipeline
 - ❖ Increase FP performance
- ❑ Out-of-order completion

ARM10EJ-S integer core in ARMv5TEJ implementation

- 32-bit ARM
- 16-bit Thumb
- 8-bit Jazelle instruction set

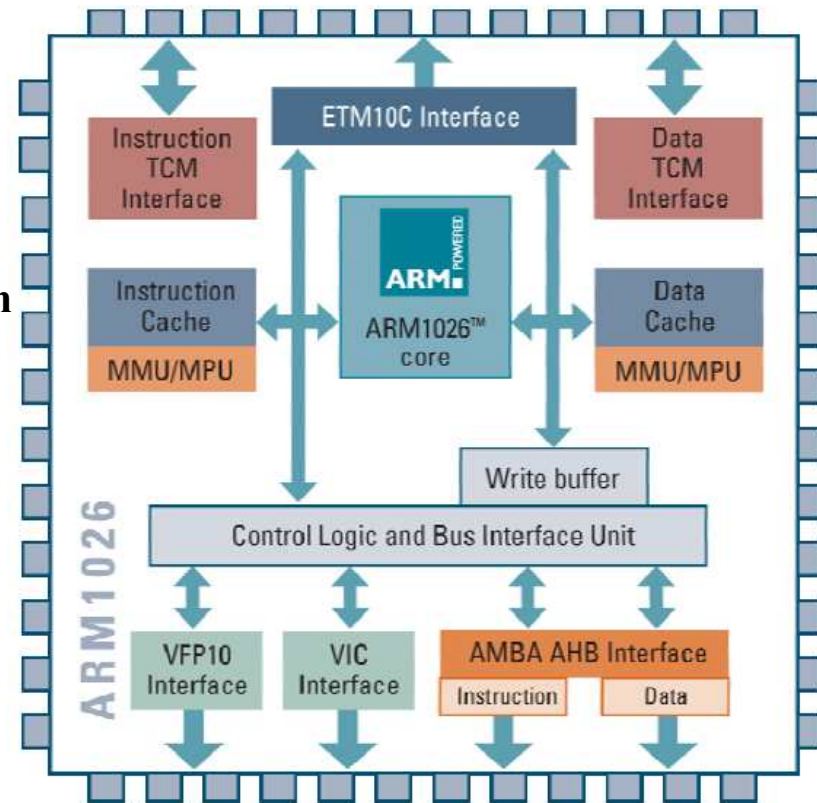
MMU - Single TLB for both instruction and data

Memory Protection Unit (MPU) - Partition external memory into 8- protection regions

Cache (I/D) - Configurable to 32 B or 128 B

Tightly Coupled Memory (TCM)

- Configurable to 32 B or 128 B-1MB

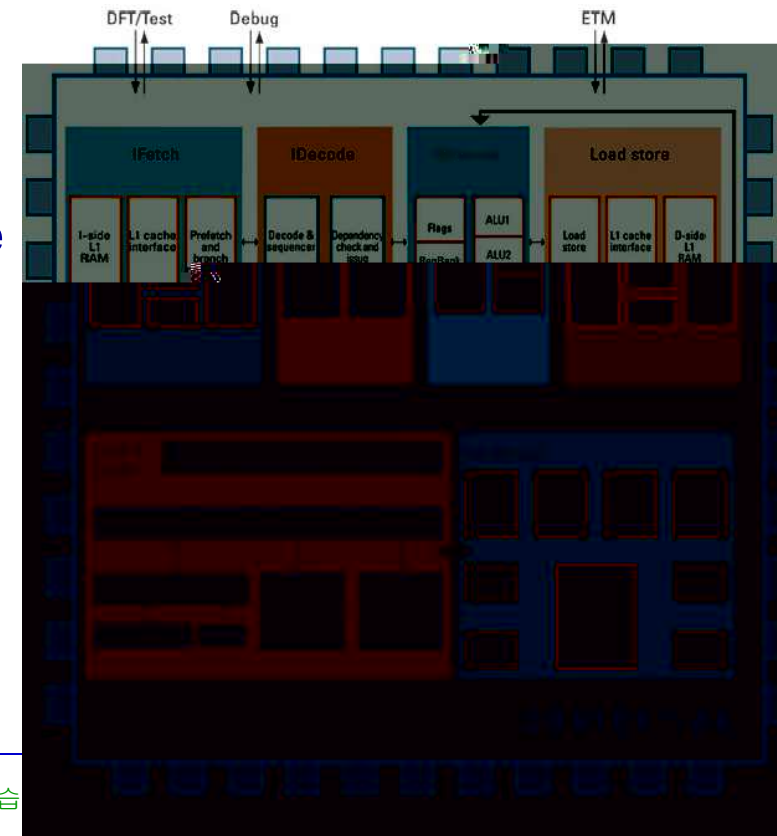


ARM 11 Family

- ❑ 8-stage pipeline
 - ❖ Load-store pipeline
 - ❖ Arithmetic pipeline
- ❑ Parallel load/store unit
 - ❖ Allow computation to continue while data transfers complete
 - ❖ Non-blocking cache
- ❑ 64-bit data bus
- ❑ Out-of-order completion
- ❑ The first core to implement ARMv6
 - ❖ SIMD (Single Instruction Multiple Data) extensions for media processing, specifically designed to increase video processing performance
 - ❖ DSP support (dual 16-bit MAC instructions)
- ❑ Example
 - ❖ ARM1136J-S
 - ❖ ARM1136JF-S

Cortex-A8

- ❑ Successor to the ARM11
- ❑ Dual-issue in-order execution superscalar pipeline
 - ❖ 13 stages
- ❑ Support for NEON signal processing extensions
 - ❖ 64/128-bit hybrid SIMD architecture
 - ❖ Multimedia and signal processing applications
 - 3D graphics, speech processing, telephony, image processing, sound synthesis, compressed audio decoding, etc.
 - ❖ own pipeline (10 stages) and register file
 - ❖ Need for vectorizing compiler
 - ❖ Target for OpenMAX API
- ❑ Thumb-2 instruction set



ARM Programming 모델

□ Programming Model

- ❖ 프로그래머가 프로그램을 작성하는데 필요한 각종 정보
 - 여기서 프로그램은 C나 C++이 아닌 어셈블리어를 의미
- ❖ Programming Model은 ARM의 Architecture를 구분 하는 기준

□ 프로그래머가 알아야 할 정보 들

- ❖ 명령어
- ❖ 프로세서의 동작 모드
- ❖ 프로세서 내부 레지스터의 구성 및 사용법
- ❖ 메모리 구조
- ❖ 예외 처리

ARM 명령어와 파이프 라인 : 데이터 처리 명령

R0 := R1 - R2



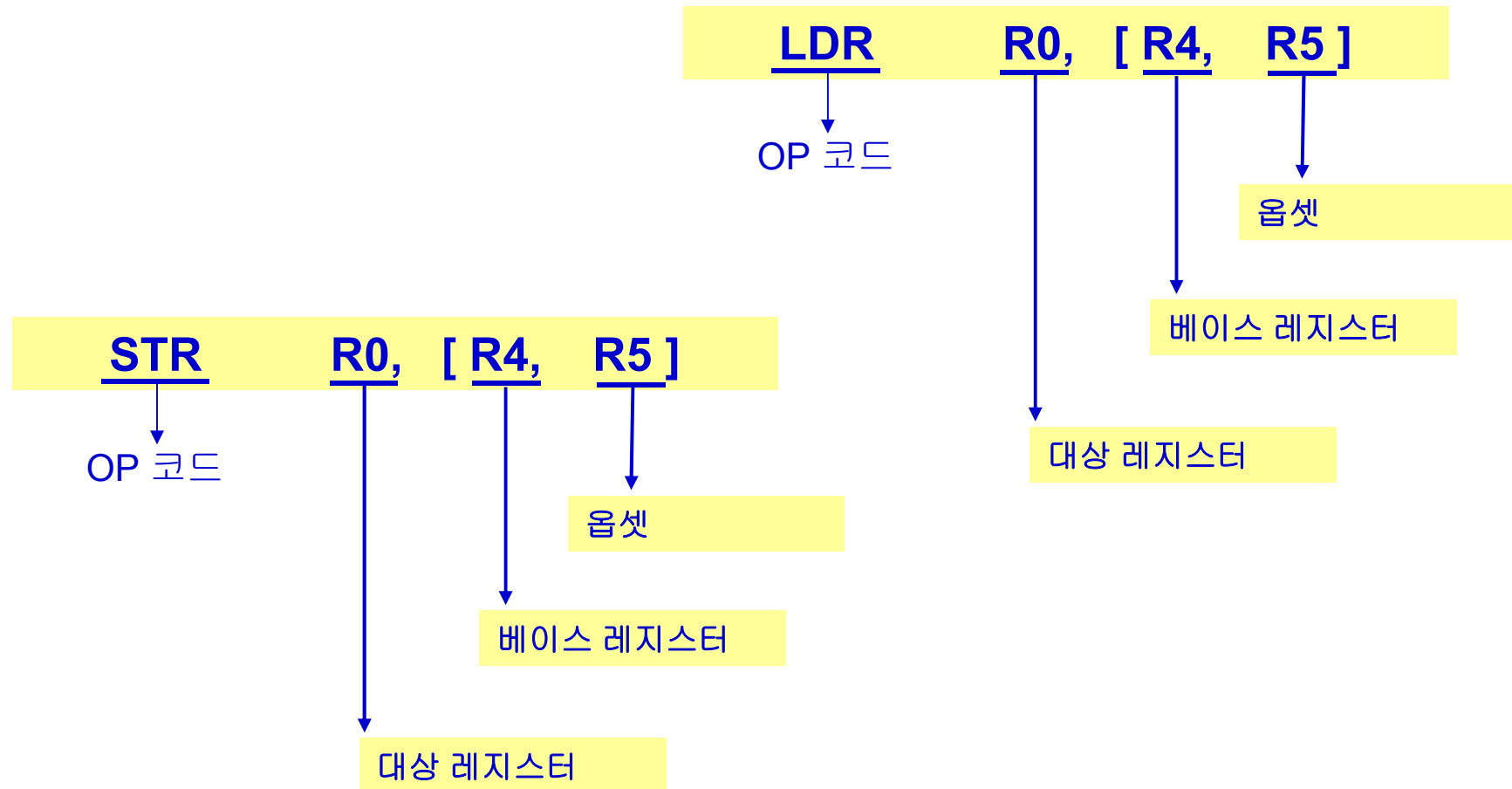
SUB
↓
OP 코드

R0,
↓
대상 레지스터

R1,
↓
오퍼랜드 1 (소스 레지스터)

OP2
↓
오퍼랜드 2

LDR/STR 명령



ARM 어셈블리어 프로그래밍

어드레스	명령	설명
0x1000	LDR R0, [R4, R5]	; R4 = 0x2000, R5 = 0xC ; R0에 0x200C 번지 메모리의 데이터 LOAD
0x1004	LDR R1, [R4, #8]	; R1에 0x2008 번지 메모리 데이터 LOAD
0x1008	ADD R2, R0, #5	; R2 := R0 + 5
0x100C	SUB R3, R1, R2	; R3 := R1 - R2
0x1010	STR R3, [R4, #4]	; R3의 데이터를 0x2004 번지 메모리에 저장

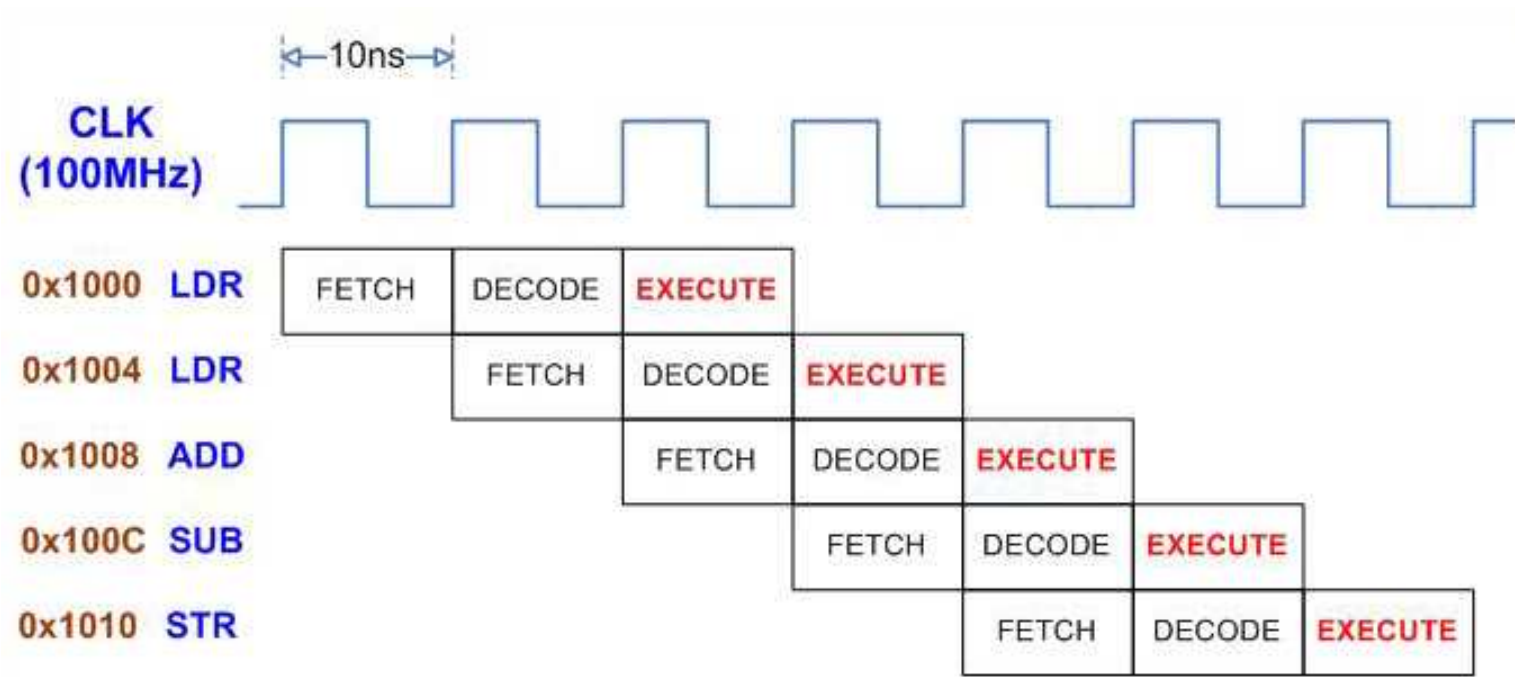
어드레스	메모리
0x200C	0x1234
0x2008	0x6789
0x2004	0x5550
0x2000	0xAAAA

최종 결과값

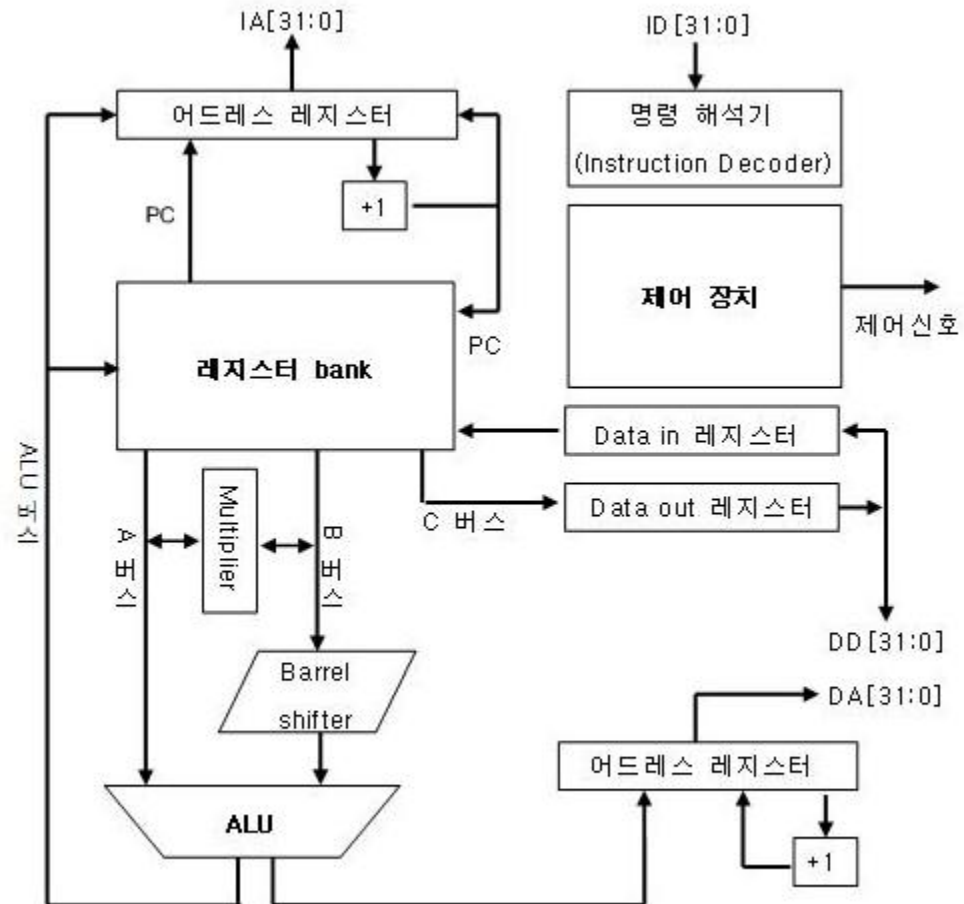
ARM 명령어 처리 과정



ARM 명령어 파이프 라인



ARM 구조와 명령어 실행 : ARM 코어의 구조



ARM 명령어(Instruction Set)

□ ARM 프로세서의 명령어

- ❖ 32 비트 ARM instruction set
- ❖ 16 비트 Thumb instruction set

□ Jazelle core를 확장한 프로세서

- ❖ 8 비트 Java 바이트 명령어

32 비트 ARM 명령어

□ 32 비트 ARM 명령어의 특징

- ❖ 모든 ARM 명령어는 조건부 실행이 가능하다.
- ❖ 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 표시 된다
 - ✓ Immediate 상수는 32비트 이하의 상수이다
- ❖ ARM 명령은 크게 11개의 기본적인 Type으로 구분된다.

□ 32 비트의 고정된 명령 길이를 사용하는 이유

- ❖ Pipeline 구성이 용이
- ❖ 명령 디코더의 구현이 쉽다
- ❖ 고속으로 처리 가능

ARM 명령어 요약

	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	

16 비트 Thumb 명령어

□ Thumb 명령어

- ❖ 32비트의 ARM 명령을 16비트로 재구성한 명령

□ 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 안된다.
- ❖ Immediate 상수 값의 표현 범위가 적다

ARM / Thumb Interwork

□ ARM state와 Thumb state

❖ ARM state

- 32 비트 ARM 명령을 수행하는 상태
- Reset 및 SWI, IRQ, FIQ, UNDEF, ABORT와 같은 Exception 발생시 프로세서는 무조건 ARM state가 된다.

❖ Thumb state

- 16 비트 Thumb 명령을 수행하는 상태

□ ARM / Thumb Interwork

- ❖ ARM state에서 Thumb state로 Thumb state에서 ARM state로 상태가 전환 되는 작업
- ❖ BX 명령에 의해서 이루어 진다.

Java 명령

- **Jazelle core**가 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 core**는 **12K** 보다 작은 **gates** 수로 구현이 가능하다.

파이프라인

□ ARM 프로세서 코어는 파이프라인을 지원

- ❖ 프로세서의 모든 부분들과 메모리 장치들이 계속적으로 쉬지 않고 동작한다.
- ❖ 하나의 명령이 수행되고 있는 동안에 다음에 수행할 명령을 **decode** 하고, 그 다음에 수행할 명령이 메모리로부터 읽어 온다.

동작모드 : Operating Mode (1)

□ **Operating** 모드는 현재 프로세서가 어떤 권한을 가지고 어떤 종류의 작업을 하고 있는지를 나타낸다.

❖ **User Mode**

- User task 또는 어플리케이션을 수행할 때의 프로세서의 동작 모드
- ARM 프로세서의 동작 모드 중 유일하게 **Un-privileged** 모드로 메모리나 I/O 장치와 같은 시스템 자원을 사용하는데 제한이 있다

❖ **FIQ(Fast Interrupt Request) Mode**

- 빠른 인터럽트의 처리를 위한 프로세서의 동작 모드

❖ **IRQ(Interrupt Request) Mode**

- 일반적으로 사용되는 인터럽트를 처리하기 위한 프로세서의 동작 모드

❖ **SVC(Supervisor) Mode**

- 시스템 자원을 관리할 수 있는 프로세서의 동작 모드
- **Operating** 시스템의 커널이나 드라이버를 처리하는 모드
- **Reset**이나 소프트웨어 인터럽트(**SWI**)가 발생하면 ARM은 Supervisor 모드가 된다.

Operating Mode (2)

❖ Abort Mode

- 명령이나 데이터를 메모리로부터 읽거나 쓸 때 오류가 발생하면 ARM은 Abort 모드가 된다.
- Abort는 외부의 메모리 제어기에서 발생된다.

❖ Undefined Mode

- ARM이 정의되지 않은 명령을 수행하려고 하면 수행되는 프로세서의 동작 모드

❖ System Mode

- User 모드와 동일한 용도로 사용되지만 **privilege** 모드이다.
- ARM Architecture v4이후부터 지원된다.

Operating 모드의 변경

□ 예외처리(Exception)

- ❖ IRQ, FIQ를 비롯한 대부분의 Operating 모드는 외부에서 발생하는 조건에 의해서 ARM 프로세서가 하드웨어적으로 변경한다.
- ❖ 외부에서 발생하는 물리적인 조건에 의해서 정상적인 프로그램의 실행을 미루고 예외적인 현상을 처리하는 것을 Exception이라 한다.
- ❖ Operating 모드의 변경은 소프트웨어에 의하여 제어 할 수도 있다.

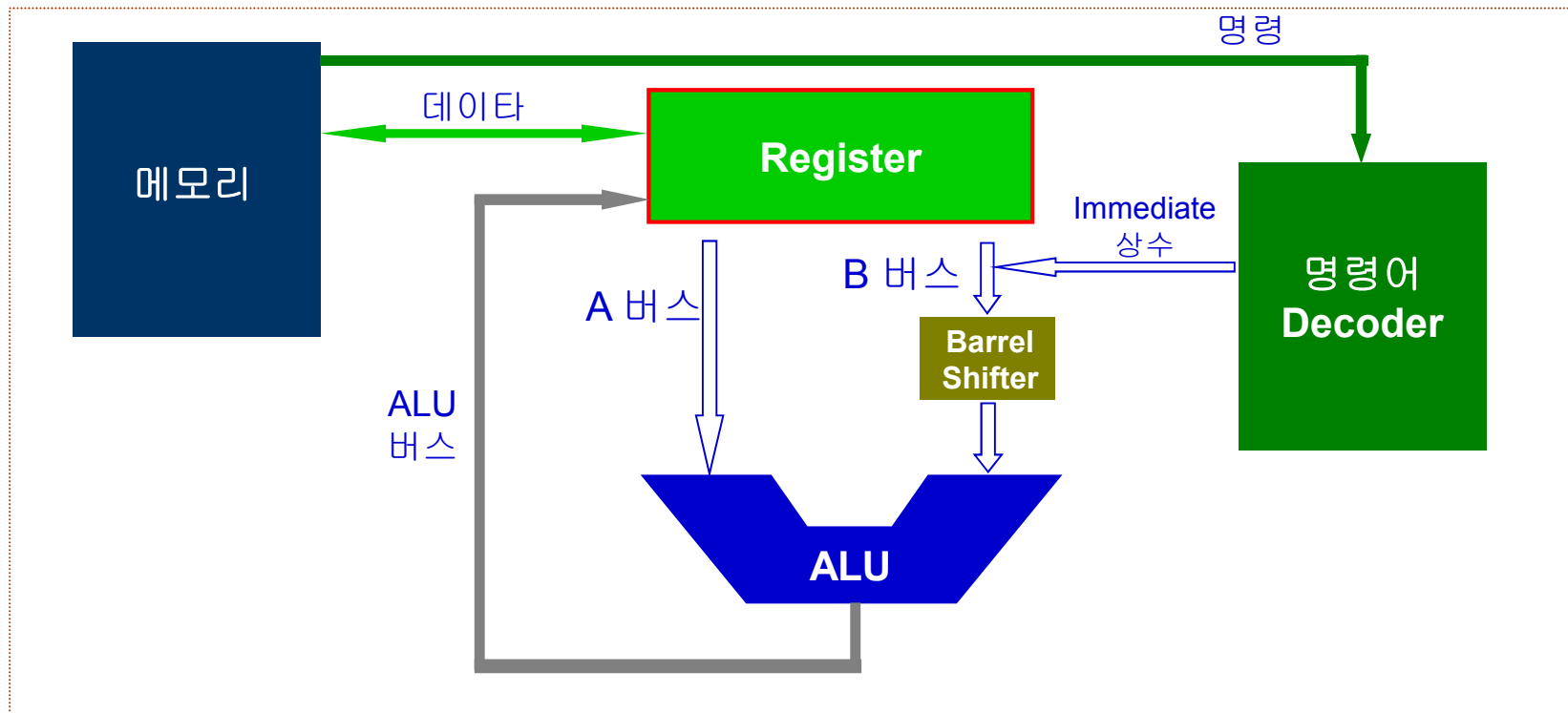
□ 시스템 콜

- ❖ User 모드에서 수행되는 프로그램에서 Supervisor 모드로 전환하여 시스템 자원을 사용할 수 있게 해주는 인터페이스
- ❖ OS에서는 소프트웨어 인터럽트(SWI)를 사용하여 시스템 콜을 구현

레지스터 : ARM 프로세서의 레지스터

□ 레지스터

- ❖ 프로세서가 작업을 하는데 사용되는 값을 저장하는 공간
- ❖ ARM에는 32비트 길이의 37개의 레지스터가 있다.



Operating 모드별 사용 가능한 레지스터

□ ARM state의 경우

- ❖ 16개의 범용 레지스터
 - R0에서 R15의 키워드를 사용하여 관리
 - R0에서 R12는 연산 명령과 같은 범용으로 사용
 - R13(SP), R14(LR), R15(PC)는 특수한 목적으로 사용
 - ✓ 연산 명령에서 사용 할 수도 있다.
- ❖ 1개의 CPSR
- ❖ Privilege 모드의 경우 각각 1개의 SPSR

□ Thumb state의 경우

- ❖ 8개의 범용 레지스터
 - R0에서 R7
- ❖ R13(SP), R14(LR), R15(PC) 레지스터
 - 연산 명령에서 사용 불가
- ❖ 1개의 CPSR
- ❖ Privilege 모드의 경우 각각 1개의 SPSR

ARM 상태의 레지스터

User/System	SVC	Abort	Undef	IRQ	FIQ
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	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12_fiq
R13_usr	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14_usr	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)

ARM state에서의 상태레지스터

CPSR	CPSR SPSR_svc	CPSR SPSR_abt	CPSR SPSR_un	CPSR SPSR_irq	CPSR SPSR_fiq
------	------------------	------------------	-----------------	------------------	------------------

특정모드에서의 레지스터 예

Current Visible Registers

Abort Mode

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
cpsr
spsr

Banked out Registers

User	FIQ	IRQ	SVC	Undef
	r8			
	r9			
	r10			
	r11			
	r12			
r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)
r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)
	spsr	spsr	spsr	spsr

Thumb 상태의 레지스터

User/System	SVC	Abort	Undef	IRQ	FIQ
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
SP_usr	SP_svc	SP_abt	SP_und	SP_irq	SP_fiq
LR_usr	LR_svc	LR_abt	LR_und	LR_irq	LR_fiq
PC	PC	PC	PC	PC	PC

Thumb state에서의 상태레지스터

CPSR	CPSR SPSR_svc	CPSR SPSR_abt	CPSR SPSR_un	CPSR SPSR_irq	CPSR SPSR_fiq
------	------------------	------------------	-----------------	------------------	------------------

Stack Pointer (SP 또는 R13)

- 프로그램에서 사용하는 스택의 위치를 저장하는 레지스터
- 프로세서의 동작 모드마다 별도로 할당된 **SP** 레지스터를 가지고 있다
- **ARM**은 별도의 스택 명령이 없다
 - ❖ Push나 Pop 과 같은 별도의 스택 명령을 제공하지 않는다
 - ❖ LDR/STR, LDM/STM 같은 데이터 전송 명령을 사용하여 스택을 처리

Link Register (LR 또는 R14)

□ 서브루틴에서 되돌아 갈 위치 정보를 저장하고 있는 레지스터

- ❖ 서브루틴을 호출하는 BL 명령을 사용하면 PC 값을 자동으로 LR에 저장
- ❖ 되돌아 갈 때는 MOV 명령을 이용하여 LR 값을 PC에 저장

```
MOV PC, LR
```

- ❖ ADD나 SUB와 같은 연산 명령을 이용하여 되돌아갈 위치를 조정 할 수도 있다.

□ 프로세서의 동작 모드마다 별도로 할당된 **LR** 레지스터를 가지고 있다

Program Counter (PC 또는 R15)

- 프로그램을 수행하는 위치를 저장하고 있는 레지스터
 - ❖ ARM state의 경우 위치 정보 비트 [31:2] 저장
 - ❖ Thumb state의 경우 위치 정보 비트 [31:1] 저장
- 다른 범용 레지스터와 마찬가지로 **ADD, SUB**와 같은 연산 명령을 사용하여 프로그램이 분기할 위치를 조정 가능
- 프로세서의 모든 동작 모드에 대하여 하나만 존재한다.

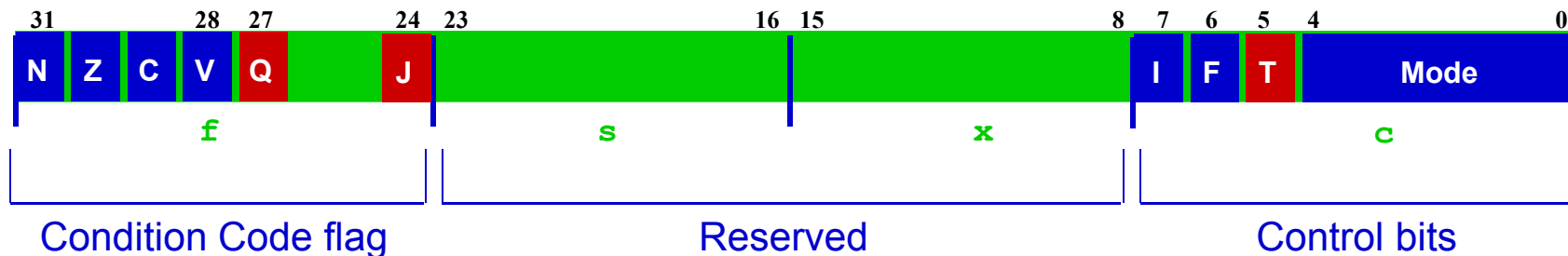
Program Status Register (PSR)

□ ARM의 Program Status Register

- ❖ 1개의 CPSR(Current Program Status Register)
- ❖ 5개의 SPSR(Saved Program Status Register)

□ PSR 레지스터의 정보

- ❖ Condition code flag
 - ALU의 연산 결과 정보를 가지는 flag 정보를 가지고 있다
- ❖ Control bits
 - 프로세서를 제어하기 위한 비트로 구성되어 있다
- ❖ Reserved



PSR 레지스터 – flag bits

□ **Flag bits**는 **ALU**를 통한 명령의 실행 결과를 나타내는 부분이다.

❖ **Negative flag ('N' 비트)**

➤ ALU 연산 결과 마이너스가 발생한 경우 세트

❖ **Zero flag ('Z' 비트)**

➤ ALU 연산 결과 0가 발생한 경우 세트

❖ **Carry flag ('C' 비트)**

➤ ALU 연산 결과 자리올림이나 내림이 발생한 경우 세트

➤ shift 연산에서 carry가 발생한 경우 세트

❖ **oVerflow flag ('V' 비트)**

➤ ALU 연산 결과 overflow가 발생하면 세트

❖ **Q flag ('Q' 비트)**

➤ ARM Architecture v5TE/J에서 새롭게 추가된 **saturation** 연산 명령의 수행 결과 **saturation**이 발생하면 세트된다.

➤ 이 비트는 사용자에게 의해서만 클리어 된다.

❖ **'J' 비트**

➤ ARM Architecture v5TEJ에서 새롭게 추가된 **Java** 바이트 코드를 수행하는 **Jazelle state**임을 나타낸다.

ARM 메모리 구조

- 메모리는 프로그램과 데이터를 저장하는 공간이다.
- **Big/Little Endian** 지원
 - ❖ ARM core는 Big-Endian 과 Little-Endian을 모두 지원
 - ❖ 외부의 핀에 의해서 하드웨어적으로 결정된다.
- **ARM** 에서 사용 가능한 데이터 **Type**
 - ❖ byte : 8비트
 - ❖ Halfword : 16 비트, 2 바이트
 - ❖ Word : 32 비트, 4 바이트
- **Un-aligned access**
 - ❖ 기존의 ARM Architecture v4T, v5T, v5TE 등은 지원되지 않는다.
 - Data Abort 발생
 - ❖ ARM Architecture v6에서는 지원한다.

Aligned 과 Un-aligned 액세스

□ 프로세서는 메모리 액세스

❖ Byte, halfword(2바이트) 또는 word(4바이트) 단위로만 가능

Aligned Access


31	24	23	16	15	8	7	0	
11	22	33	44					0x0C
11	22	33	44					0x08
11	22	33	44					0x04
11	22	33	44					0x00


Word 단위 Access :
0x00, 0x04, 0x08, ...

Half-Word Access :
0x00, 0x02, 0x04, 0x06, 0x08, ...

Un-aligned Access

31	24	23	16	15	8	7	0	
11	22	33	44					0x0C
11	22	33	44					0x08
11	22	33	44					0x04
11	22	33	44					0x00

Word 단위 Access :  Abort
0x01, 0x06, 0x07, ...

Half-Word Access :  Abort
0x01, 0x03, 0x05 ...

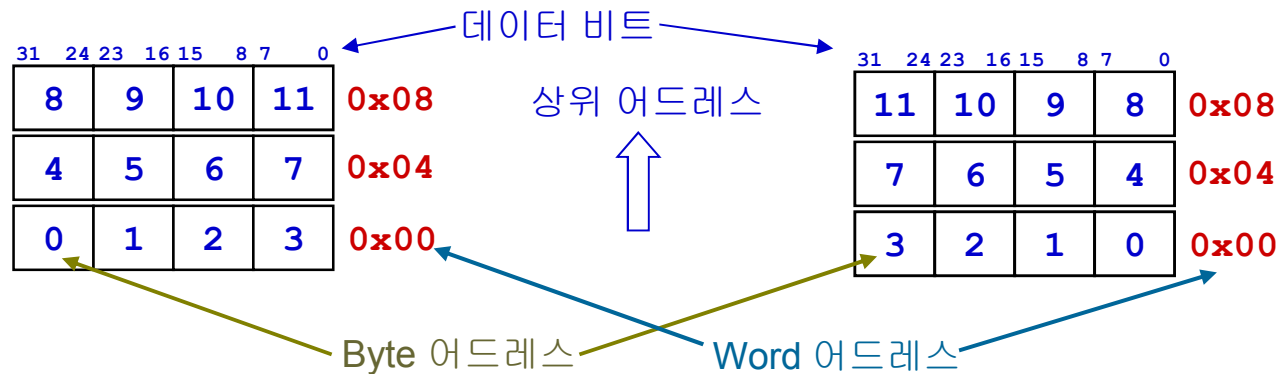
Little-Endian과 Big-Endian

□ Big-Endian

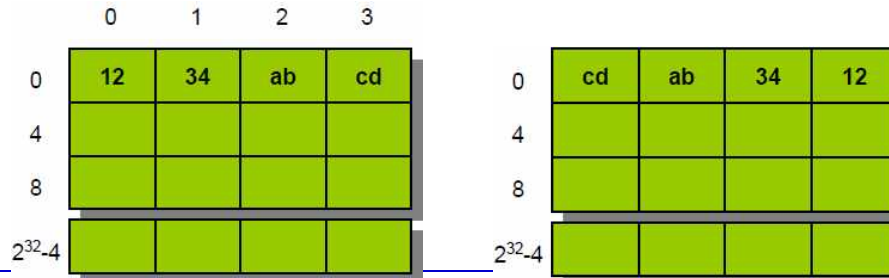
- ❖ 메모리의 하위 어드레스(Byte 어드레스 “0”)에 MSB(Most Significant Byte)가 위치하고 있는 메모리 구조
- ❖ MSB는 메모리 데이터의 가장 상위 비트 24 에서 31 까지

□ Little-Endian

- ❖ 메모리의 하위 어드레스(Byte 어드레스 “0”)에 LSB(Least Significant Byte)가 위치하고 있는 메모리 구조
- ❖ LSB는 메모리 데이터의 가장 하위 비트 0 에서 7 까지



- 0,4,8은 word 어드레스, 0,2,4,6,8,10은 halfword 어드레스, 0,1,2,3,4,...은 byte 어드레스를 나타낸다.
- word로 메모리를 액세스 하는데 1,2 또는 3 번지를 액세스 하면 un-aligned 액세스가 된다.
- 32bit data 저장 예- 0x1234abcd



예외처리 (Exception Handler)

□ 예외처리(Exception)

- ❖ 외부의 요청이나 오류에 의해서 정상적으로 진행되는 프로그램의 동작을 잠시 멈추고 프로세서의 동작 모드를 변환하고 미리 정해진 프로그램을 이용하여 외부의 요청이나 오류에 대한 처리를 하도록 하는 것
- ❖ Exception의 예
 - I/O 장치에서 인터럽트를 발생시키면 IRQ Exception이 발생하고, 프로세서는 발생한 IRQ Exception을 처리하기 위해 IRQ 모드로 전환되어 요청된 인터럽트에 맞는 처리 동작 수행

□ ARM의 Exception

- ❖ Reset
- ❖ Undefined Instruction
- ❖ Software Interrupt
- ❖ Prefetch Abort
- ❖ Data Abort
- ❖ IRQ(Interrupt Request)
- ❖ FIQ(Fast Interrupt Request)

예외처리 **Vector** 와 우선순위

□ 예외처리 벡터

- ❖ Exception이 발생하면 미리 정해진 어드레스의 프로그램을 수행
- ❖ 미리 정해진 프로그램의 위치를 **Exception Vector**라 한다.

□ 예외처리 벡터 테이블

- ❖ 발생 가능한 각각의 **Exception**에 대하여 **Vector**를 정의해 놓은 테이블
 - 각 **Exception** 별로 1 word 크기의 명령어 저장 공간을 가진다.
- ❖ **Vector Table**에는 **Branch** 또는 이와 유사한 명령어로 실제 **Exception**을 처리하기 위한 루틴으로 분기 할 수 있는 명령어로 구성되어 있다.
 - FIQ의 경우는 **Vector Table**의 맨 상위에 위치하여 분기명령 없이 처리루틴을 프로그램 할 수 있다.
- ❖ ARM은 기본적으로 0x00000000에 **Vector Table**을 둔다.
(MMU 제어 프로그램에 의해 위치 변경 가능)

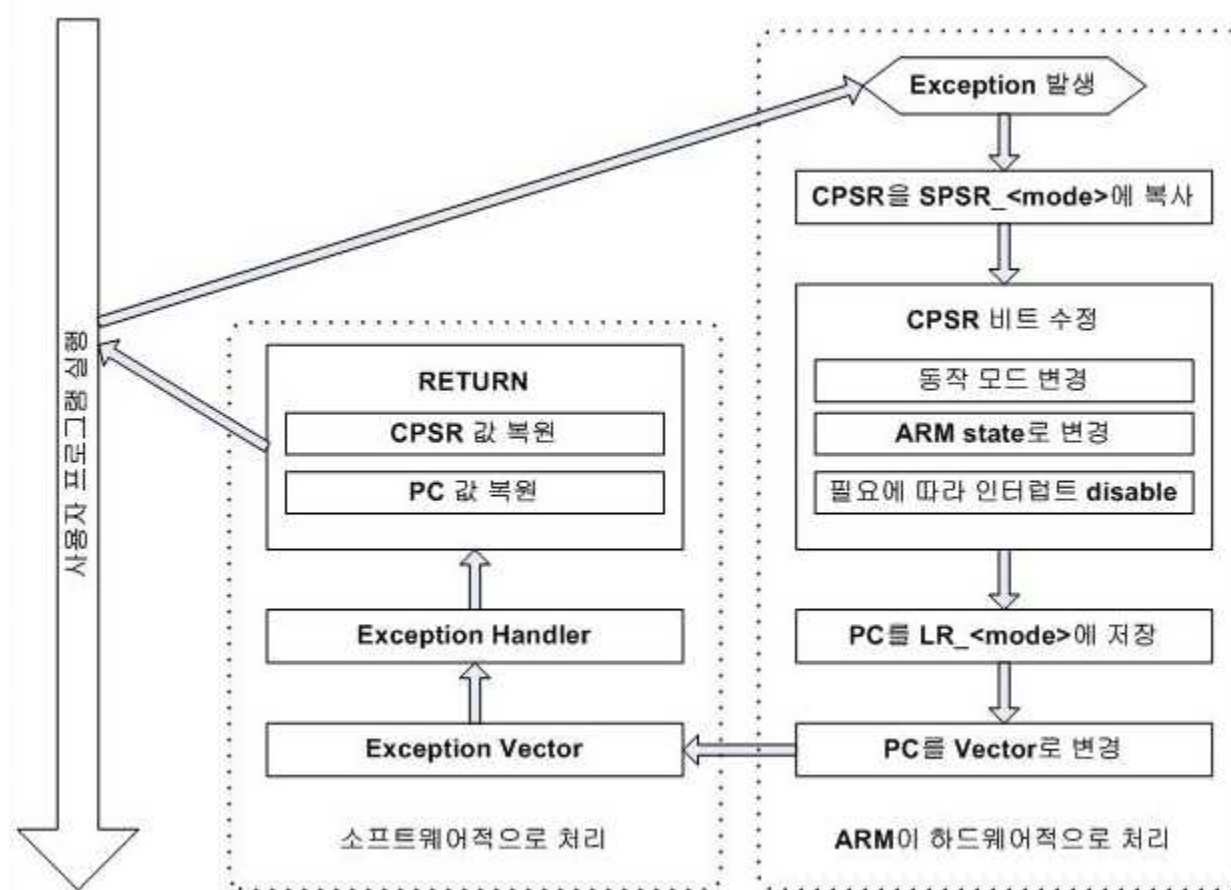
□ **Exception** 우선 순위

- ❖ 동시에 **Exception**이 발생하는 경우 처리를 위해 우선 순위 지정

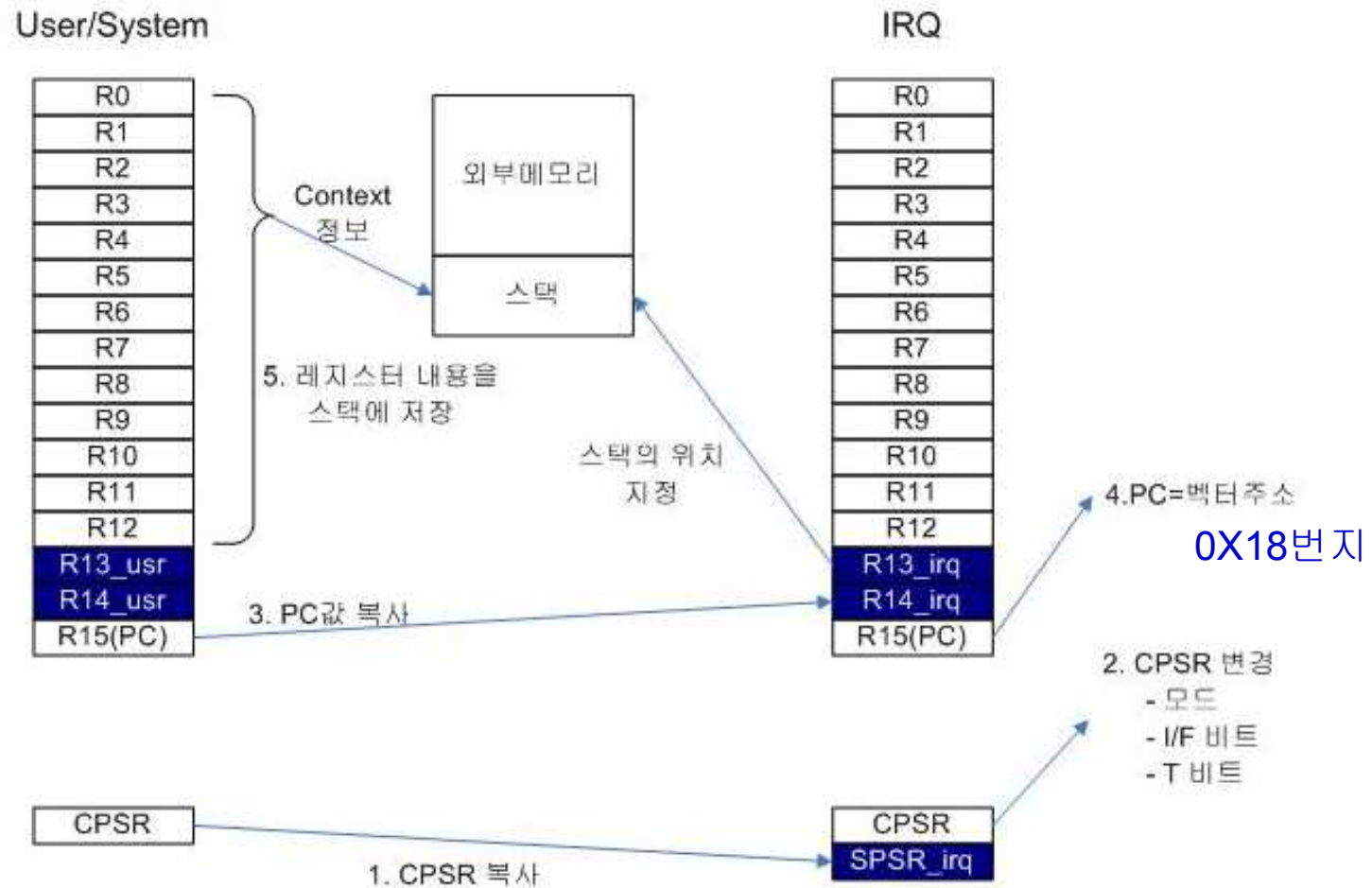
Exception Vector Table

Exception	Vector Address	우선순위	전환되는 동작모드
Reset	0x0000 0000	1 (High)	Supervisor(SVC)
Undefined Instruction	0x0000 0004	6 (Low)	Undefined
Software Interrupt(SWI)	0x0000 0008	6	Supervisor(SVC)
Prefetch Abort	0x0000 000C	5	Abort
Data Abort	0x0000 0010	2	Abort
Reserved	0x0000 0014		
IRQ	0x0000 0018	4	IRQ
FIQ	0x0000 001C	3	FIQ

예외처리 과정



IRQ 예외 처리와 레지스터 사용 예



□ ARM 프로세서 명령어

ARM 프로세서 명령어의 특징

32비트 ARM 명령어

아키텍처 v5TE의 명령어

16비트 Thumb 명령어

ARM 프로세서 명령어의 특징 : 명령어(Instruction Set)

□ 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

32 비트 ARM 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 표시 된다
 - Immediate 상수는 32비트 이하의 상수이다
- 모든 **ARM** 명령어는 조건부 실행이 가능하다.
- **Load/Store Architecture**를 사용한다.

16 비트 Thumb 명령어

□ Thumb 명령어

- ❖ 32비트의 ARM 명령을 16비트로 재구성한 명령

□ Thumb 명령어의 장점

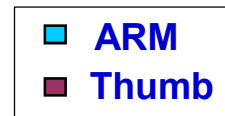
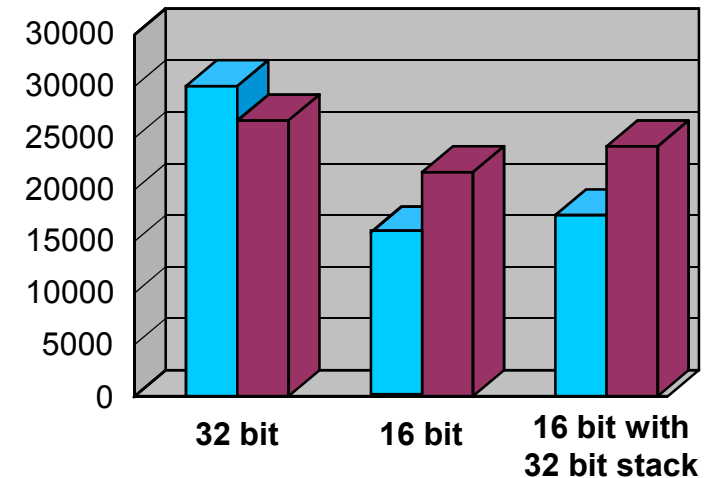
- ❖ 코드의 크기를 줄일 수 있다
 - ARM 명령의 65%
- ❖ 8비트 나 16비트와 같은 좁은 메모리 인터페이스에서 ARM 명령을 수행할 때 보다 성능이 우수하다

□ Thumb 명령어의 단점

- ❖ 조건부 실행이 안된다.
- ❖ Immediate 상수 값의 표현 범위가 적다

CPU 성능 측정

Dhrystone 2.1 / sec
@20MHz



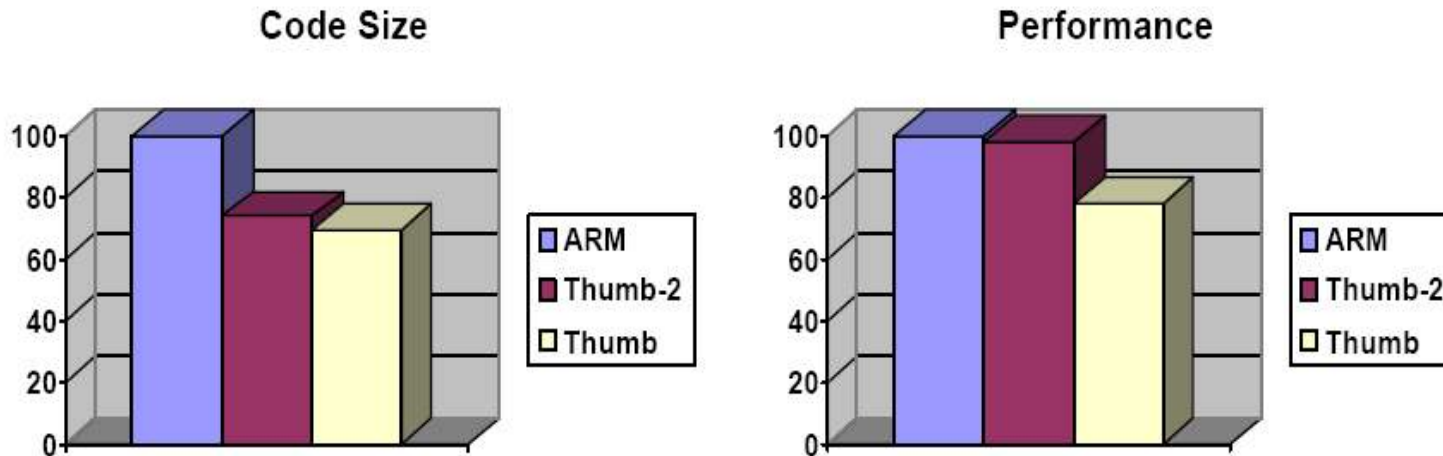
Thumb-2 명령어

□ 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



Java 명령

- **Jazelle core**가 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 core**는 **12K** 보다 작은 **gates** 수로 구현이 가능하다.

32비트 ARM 명령어

	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	

32비트 ARM 명령어의 구조

31			28 2			20			16 1			11			8			<u>Instruction type</u>					
Cond		0 0		I	Opcode			S	Rn		Rd		Operand 2						Data processing / PSR Transfer				
Cond		0 0 0 0 0 0						A	S	Rd		Rn		Rs		1 0 0 1		Rm		Multiply			
Cond		0 0 0 0 1				U		A	S	RdHi		RdLo		Rs		1 0 0 1		Rm		Long Multiply			
Cond		0 0 0 1 0				B		0 0		Rn		Rd		0 0 0 0		1 0 0 1		Rm		Swap			
Cond		0 1		I	P	U	B	W	L	Rn		Rd		Offest						Load / Store Byte /Word			
Cond		1 0 0			P	U	S	W	L	Rn		Register List								Load / Store Multiple			
Cond		0 0 0			P	U	1	W	L	Rn		Rd		Offset1		1	S	H	1	Offset2		Halfword transf. Imm. Offset(v4)	
Cond		0 0 0			P	U	0	W	L	Rn		Rd		0 0 0 0		1	S	H	1	Rm		Halfword transf. Reg. Offset(v4)	
Cond		1 0 1		L		Offset														Branch			
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		Branch Exchange (v4T)
Cond		1 1 0			P	U	N	W	L	Rn		CRd		CPNum		Offset						Coprocessor data transfer	
Cond		1 1 1 0				Op1				CRn		CRd		CPNum		Op2		0	CRm		Coprocessor data operation		
Cond		1 1 1 0				Op1				L	CRn		Rd		CPNum		Op2		1	CRm		Coprocessor register transfer	
Cond		1 1 1 1				SWI Number														Software Interrupt			

조건부 실행

□ **ARM**에서 모든 명령은 조건에 따라 실행 여부 결정 가능

□ 분기 명령의 사용을 줄인다.

- ❖ 분기 명령이 사용되면 파이프라인이 스톱(stall) 되고 새로운 명령을 읽어오는 사이클의 낭비가 따른다.

□ 조건 필드(**Condition field**)

- ❖ 모든 명령어는 조건 필드를 가지고 있으며 **CPU**가 명령의 실행 여부를 결정하는데 사용된다.
- ❖ **Current Program Status Register(CPSR)**의 조건 플래그의 값을 사용하여 조건을 검사

조건부 실행 방법

- 조건에 따라 명령어를 실행하도록 하기 위해서는 적절한 조건을 접미사로 붙여주면 됨 :

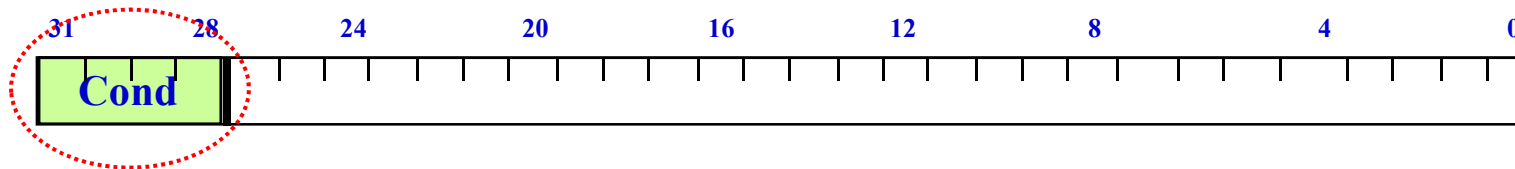
- ❖ 조건 없이 실행

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

조건 플래그 변경

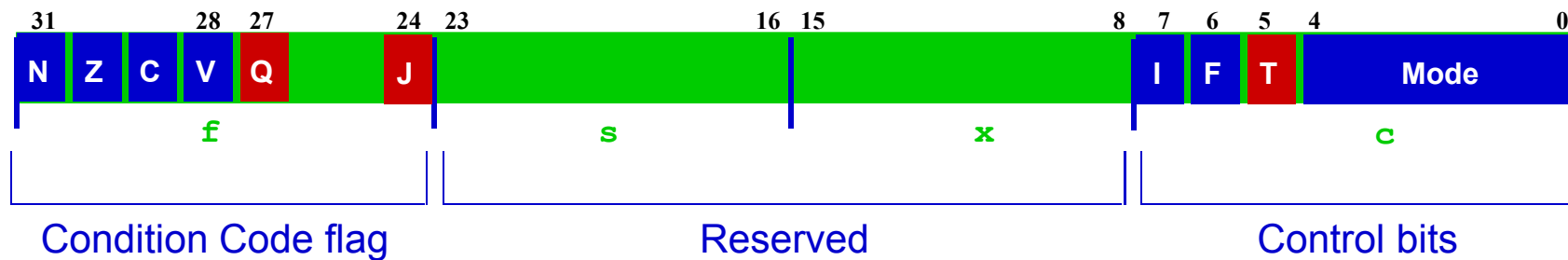
□ 조건 플래그의 변경은 명령어에 “S” 접미사를 삽입

- ❖ Data processing 명령의 경우 “S” 접미사가 없으면 condition flag에 영향을 미치지 않는다.
- ❖ SUB 연산 실행 후 결과를 가지고 조건 플래그를 세트

```
SUBS r0,r1,r2           ; r0 = r1 - r2  
                        ; ... and set flags
```

□ 데이터 처리 명령 중 비교를 위한 명령은 별도로 “S” 접미사를 삽입하지 않아도 조건 플래그가 변경된다.

조건 플래그(Condition Flags)

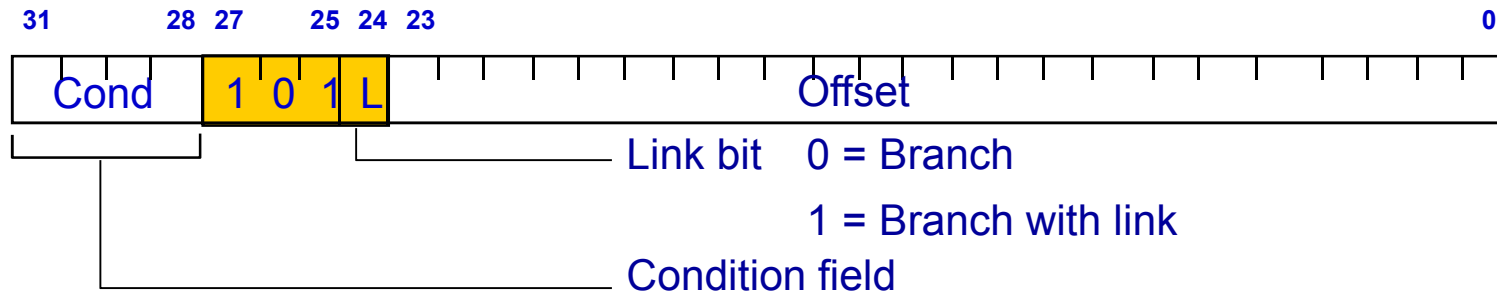


Flag	논리 연산	산술 연산
Negative (N=1)	사용되지 않는다	Signed 연산에서 비트 31이 세트 되어 Negative 결과 발생
Zero (Z=1)	연산 결과가 모두 0	연산 결과가 0
Carry (C=1)	Shift 동작 결과 carry 발생	연산 결과가 32 비트를 넘으면 세트
oVerflow (V=1)	사용되지 않는다	연산 결과가 31 비트를 넘어 sign bit 상실

분기 명령어

명령어	B, BL	Branch, Branch Link
형식	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.	
동작	{cond} 조건에 맞으면, <expression>이 가리키는 위치로 분기하고, {L} option이 사용되면 LR에 PC 값을 저장한다.	
Flag	영향 없음	
사용 예	B there ; 무조건 there로 분기하라 BL sub+label ; 어드레스를 계산한 후 서브루틴 콜 CMP R1,#0 ; R1과 0을 비교하고, BEQ fred ; R1 0 fred	

분기 명령어의 구성



□ 분기할 주소 계산

- ❖ PC 값을 기준으로 비트 [23:0]에 해당하는 **Offset**이 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

Subroutine 구현

□ Branch with Link (BL) 명령

- ❖ 다음에 수행할 명령의 위치를 LR에 저장한다.
- ❖ $LR = PC - 4$ ← 파이프 라인 단계에서 F- D - Ex - ... 즉 BL이 실행중일때는 다음 PC는 BL로부터 2번째 명령어를 인출중이다. 따라서 돌아갈 번지는 BL로 부터 첫번째 명령어를 저장

□ Subroutine에서 Return

- ❖ LR에 저장된 주소를 PC에 옮긴다.

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

데이터 처리 명령어

□ 관련 명령어 종류

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

□ Load /Store 구조

- ❖ 데이터 처리 명령은 레지스터 내의 값과 상수로만 연산 이루어지고
메모리에서 데이터를 읽거나 쓸 때는 **Load** 또는 **Store** 명령을 사용한다.

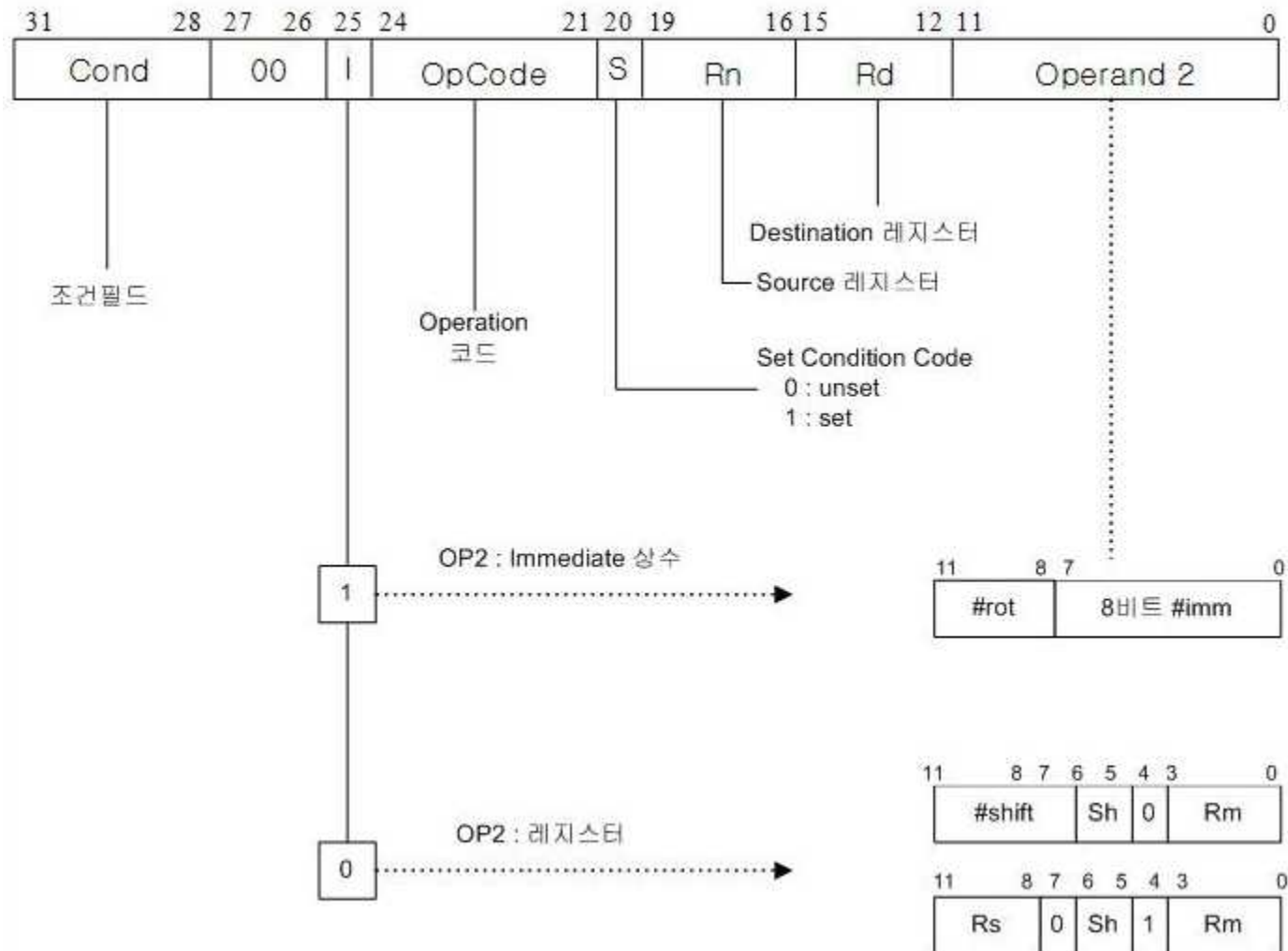
데이터 처리 명령어

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

cond	0	0	I	Opcode	S	Rn	Rd	Shifter_operand
------	---	---	---	--------	---	----	----	-----------------

Opcode	Mnemonic	Meaning	Action
0000	AND	Logical AND	$Rd = Rn \text{ AND shifter_operand}$
0001	EOR	Logical Exclusive OR	$Rd = Rn \text{ EOR shifter_operand}$
0010	SUB	Subtract	$Rd = Rn - \text{shifter_operand}$
0011	RSB	Reverse subtract	$Rd = \text{shifter_operand} - Rn$
0100	ADD	Add	$Rd = Rn + \text{shifter_operand}$
0101	ADC	Add with carry	$Rd = Rn + \text{shifter_operand} + \text{Carry}$
0110	SBC	Subtract with carry	$Rd = Rn - \text{shifter_operand} - \text{NOT}(\text{Carry})$
0111	RSC	Reverse Subtract with carry	$Rd = \text{shifter_operand} - Rn - \text{NOT}(\text{Carry})$
1000	TST	Test	Update flags after $Rn \text{ AND shifter_operand}$
1001	TEQ	Test Equivalence	Update flags after $Rn \text{ EOR shifter_operand}$
1010	CMP	Compare	Update flags after $Rn - \text{shifter_operand}$
1011	CMN	Commom	Update flags after $Rn + \text{shifter_operand}$
1100	ORR	Logical OR	$Rd = Rn \text{ OR shifter_operand}$
1101	MOV	Move	$Rd = \text{shifter_operand}$
1110	BIC	Bit clear	$Rd = Rn \text{ AND NOT}(\text{shifter_operand})$
1111	MVN	Move Not	$Rd = \text{NOT}(\text{shifter_operand})$

데이터 처리 명령어의 구조



산술 연산

□ 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 ; $r0 := r1 + r2$, ADDC r0, r1, r2 ; $r0 := r1 + r2 + C$
- ❖ SBC r0, r1, r2 ; $r0 := r1 - r2 + C - 1$
- ❖ SUBEQ r3, r3, #1 ; if Z=1, $r3 := r3 - \#1$
- ❖ RSBLES r4, r5, r7 ; if Z=1 or N=1, $N!=V$, $r4 := r7 - r5$, and set flags

논리 연산

□ 논리 연산 명령어

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

□ 문법:

❖ $\langle \text{Operation} \rangle \{ \langle \text{cond} \rangle \} \{ S \} Rd, Rn, \text{Operand2}$

□ 예제:

❖ AND r0, r1, r2

❖ BICEQ r2, r3, #7 ; $r2 := r3 \text{ and } (\text{not } \#7)$

❖ EORS r1,r3,r0 ; $r1 := r3 \text{ xor } r0$, and set flags

비교 연산

- 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

조건플래그에 의한 실행 비교

```
while (a != b) {  
    if (a > b) a -= b;  
    else      b -= a;  
}
```

```
gcd      CMP      r1, r2  
         BEQ      complete  
         BLT      lessthan  
         SUB      r1, r1, r2  
         B        gcd  
lessthan SUB      r2, r2, r1  
         B        gcd  
complete . . .
```

```
gcd      CMP      r1, r2  
         SUBGT     r1, r1, r2  
         SUBLT     r2, r2, r1  
         BNE      gcd  
         . . .
```

데이터 Move 명령

□ 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 1이 없다

□ 예제:

❖ MOV r0, r1 ; r0 <- r1

❖ MOVS r2, #10 ; r2 <- 10, set flag

❖ MVNEQ r1, r2 ; if zero flag set then r1 <- ~r2

❖ MVN r0, #0xff000000 ; r0 <- 0x00ffffff

오퍼랜드2와 배럴시프터 (Shift)

❑ 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)



오퍼랜드2와 배럴시프터 (Rotate)

□ Rotate Right (ROR)

- ❖ LSB가 MSB로 돌아(wrap-around)온다



□ Rotate Right Extended (RRX)

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



오퍼랜드2에서 배럴시프터 사용법

□ 레지스터가 **SHIFT**되는 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.

✓ *ARM*에는 2개의 레지스터 *read* 포트만 존재한다.

```
ADD r5, r5, r3 LSL r2
```

□ **Shift** 가 명시되지 않으면 **default shift** 적용

❖ LSL #0

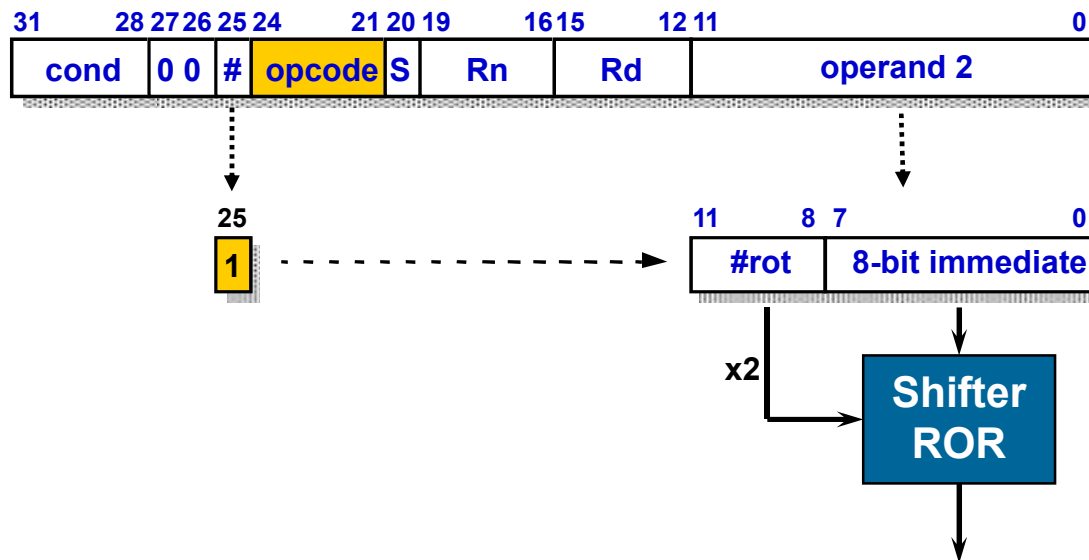
❖ barrel shifter 동작이 없다.

오퍼랜드2와 Immediate 상수

□ Operand 2로 Immediate 상수로 사용 가능

❖ 32비트 명령어 내에 그 값이 지정된다.

➤ Immediate 상수는 32비트 상수를 가질 수 없다



8비트 상수 값이 #rot에
지정된 값의 2배수를
취하여 ROR 한 값이 사용

#rot = 0x4
8-bit immediate 상수 = 0xFF } **→ Immediate 상수 = 0xFF000000**

PSR 전송 명령

□ 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 지정된 SPSR 만 액세스 가능
- ❖ User 모드의 경우에는 액세스 가능한 SPSR이 없다

PSR 전송 명령 (MSR)

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> <p>f,x,s,c는 PSR의 각 8비트 단위 필드</p>	
동작	Rm 레지스터의 내용을 <PSR[_fields]>에 옮긴다.	
사용 예	<pre>MSR CPSR, r0 ; CPSR := r0 MSR CPSR_f, r0 ; CPSR의 condition flags 비트 := r0 MSR CPSR_fc, r1 ; CPSR의 flags, control 비트 := r1 MSR CPSR_c, #0x80 ; CPSR의 control 비트 := 0x80 , IRQ disable</pre>	

32 비트 Multiply

명령어	MUL	32 비트 Multiply
형식	<p>MUL{cond}{S} Rd, Rm, Rs</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>Rm과 Rs는 곱셈이 되는 레지스터</p>	
동작	<p>$Rd := Rm * Rs$</p> <p>레지스터 동작만 사용 가능하고 Immediate 상수는 사용 할 수 없다.</p> <p>32비트 이상의 결과는 버린다.</p>	
사용 예	<p>MUL R1, R2, R3 ; $R1 := R2 * R3$</p>	

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>Rm과 Rs는 곱셈이 되는 레지스터, Rn은 더해지는 레지스터</p>	
동작	$Rd := Rm * Rs + Rn$	
사용 예	<p>MLA R1, R2, R3, R4 ; R1 := R2 * R3 + R4</p>	

64 비트 Multiply

Long Multiply : Multiply signed/unsigned 64-bit value

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>SMULL R0, R1, R2, R3 ; [R1,R0] := R2 * R3</p> <p>UMULL R0, R1, R2, R3 ; [R1,R0] := R2 * R3</p>	

64 비트 Multiply-Accumulate

Long Multiply and Add : Multiply signed/unsigned 64-bit value

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	<p>$RdHi, RdLo := Rm * Rs + (RdHi, RdLo)$</p>	
사용 예	<p>SMLAL R0, R1, R2, R3 ; $[R1, R0] := [R1, R0] + R2 * R3$</p> <p>UMLAL R0, R1, R2, R3 ; $[R1, R0] := [R1, R0] + R2 * R3$</p>	

ARM의 데이터 전송 명령

□ ARM의 Load / Store 구조

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

□ ARM의 메모리 액세스 명령

- ❖ 단일 레지스터 데이터 전송 명령 (LDR / STR)
- ❖ 블록 단위 데이터 전송 명령 (LDM/STM)
- ❖ 단일 레지스터를 사용한 메모리와 레지스터 내용을 스왑 (SWP)

단일 레지스터를 사용한 데이터 전송 명령

□ 단일 레지스터 데이터 전송 명령 명령

엑세스 단위	Load 명령	Store 명령
워드(Word)	LDR	STR
바이트(Byte)	LDRB	STRB
하프워드(Halfword)	LDRH	STRH
Signed 바이트	LDRSB	
Signed 하프워드	LDRSH	

□ 모든 명령어는 **STR / LDR** 다음에 적절한 **Condition** 코드를 삽입하여 조건부 실행 가능

❖ e.g. LDREQB

LDR(Load) 명령

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 주소의 메모리 데이터를 R1에 저장</p> <p>LDREQB R1, [R6,#5] ; 조건이 EQ이면 R6+5 주소의 데이터를 바이트만큼 읽어 R1에 저장</p>	

STR(Store) 명령

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 주소에 R1의 내용을 저장</p> <p>STREQB R1, [R6,#5] ; 조건이 EQ이면 R6+5 주소에 R1의 내용을 byte 단위 저장</p>	

LDR/STR의 어드레스 지정 방법

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

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.

LDR/STR의 <offset>

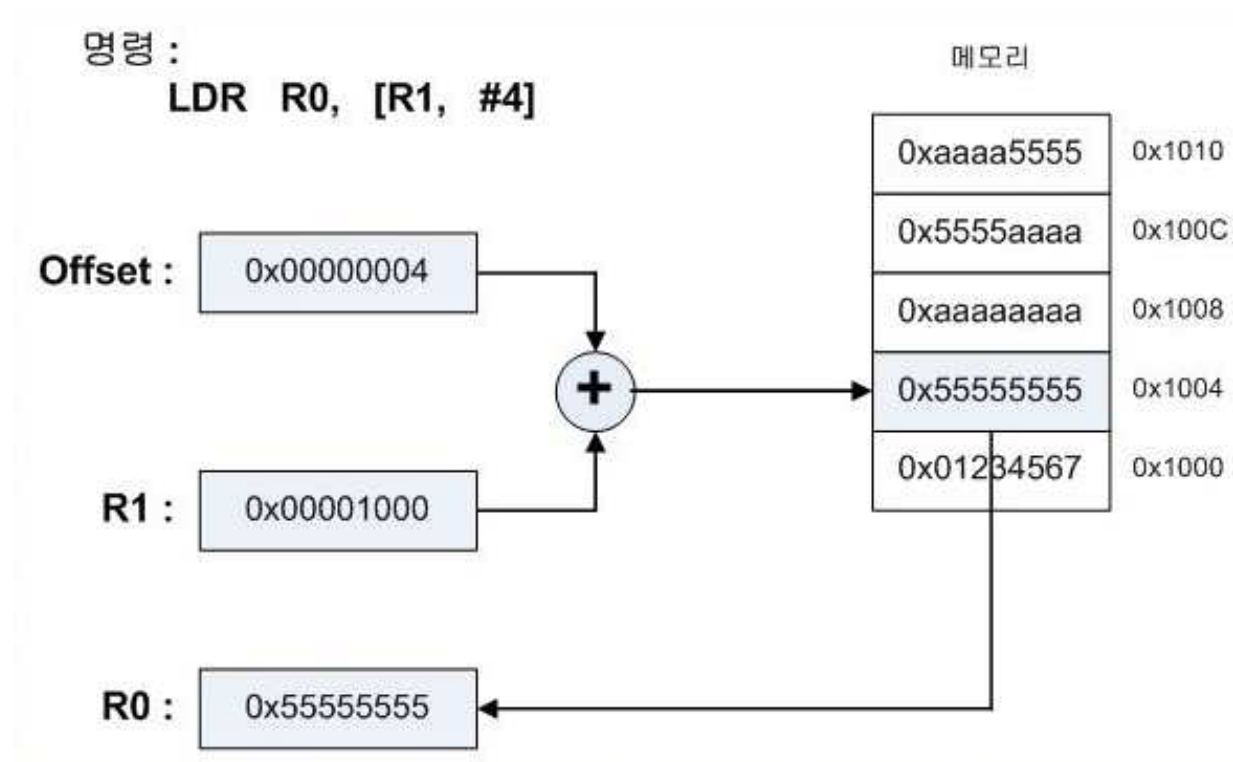
- 레지스터 बैं크에서 **B** 버스를 통해서 **Barrel shifter**를 경유하여 **ALU**에 전달
- **Offset**으로 사용될 수 있는 표현
 - ❖ Unsigned 12비트 immediate 상수 (0 ~ 4095 바이트)
 - ❖ 레지스터
 - Immediate 값으로 shift operation 지정 가능
- '+' 또는 '-' 연산을 함께 사용 가능

LDR/STR 사용예

LDR R1, [R0]	; Load R1 from the address in R0
LDR R8, [R3, #4]	; Load R8 from the address in R3 + 4
LDR R12, [R13, #-4]	; Load R12 from R13 - 4
STR R2, [R1, #0x100]	; Store R2 to the address in R1 + 0x100
LDRB R5, [R9]	; Load byte into R5 from R9
	; (zero top 3 bytes)
LDRB R3, [R8, #3]	; Load byte to R3 from R8 + 3
	; (zero top 3 bytes)
STRB R4, [R10, #0x200]	; Store byte from R4 to R10 + 0x200
LDR R11, [R1, R2]	; Load R11 from the address in R1 + R2
STRB R10, [R7, -R4]	; Store byte from R10 to addr in R7 - R4
LDR R11, [R3, R5, LSL #2]	; Load R11 from R3 + (R5 x 4)
LDR R1, [R0, #4]!	; Load R1 from R0 + 4, then R0 = R0 + 4
STRB R7, [R6, #-1]!	; Store byte from R7 to R6 - 1, then R6 = R6 - 1
LDR R3, [R9], #4	; Load R3 from R9, then R9 = R9 + 4
STR R2, [R5], #8	; Store R2 to R5, then R5 = R5 + 8
LDR R0, [PC, #40]	; Load R0 from PC + 0x40 (= address of LDR + 8 + 0x40)
LDR R0, [R1], R2	; Load R0 from R1, then R1 = R1 + R2
LDRSH R5, [R9]	; Load signed halfword to R5 from R9
LDRSB R3, [R8, #3]	; Load signed byte to R3 from R8 + 3
LDRSB R4, [R10, #0xC1]	; Load signed byte to R4 from R10 + 0xC1
STRH R10, [R7, -R4]	; Store halfword from R10 to R7 - R4

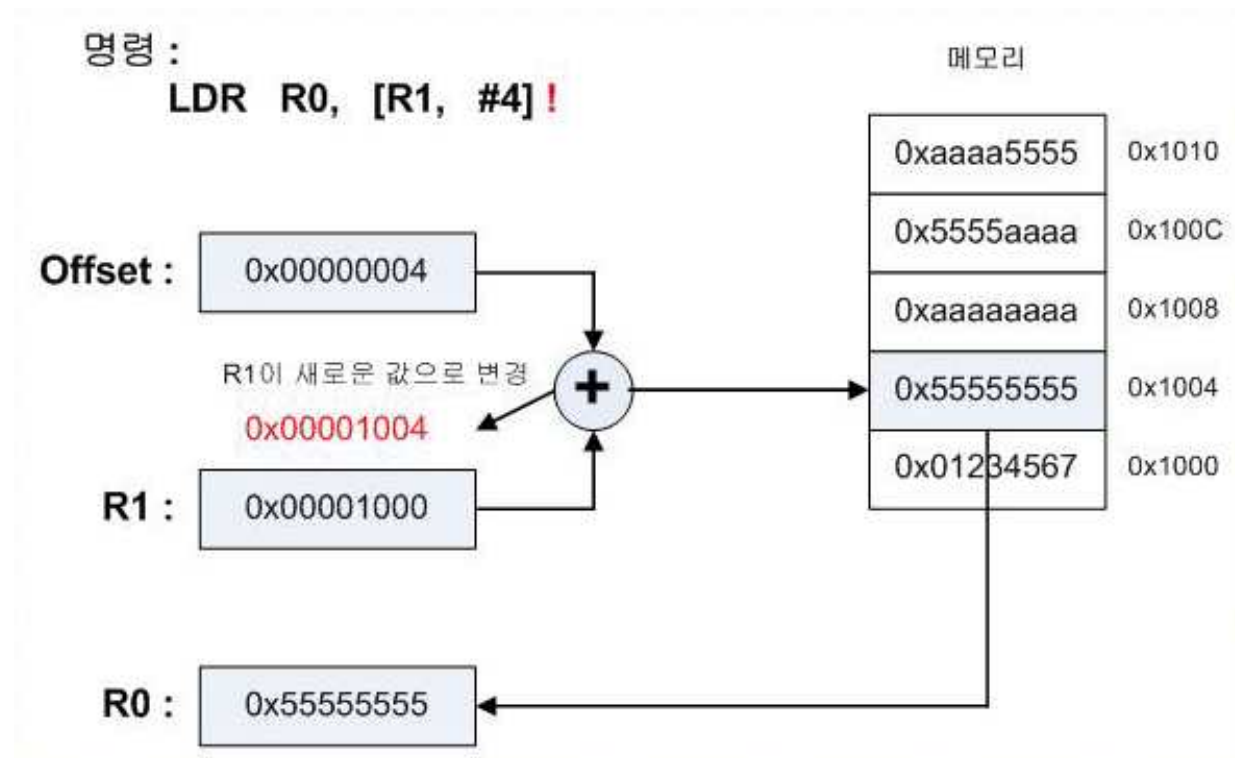
Pre-indexed 어드레스 지정 방식

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



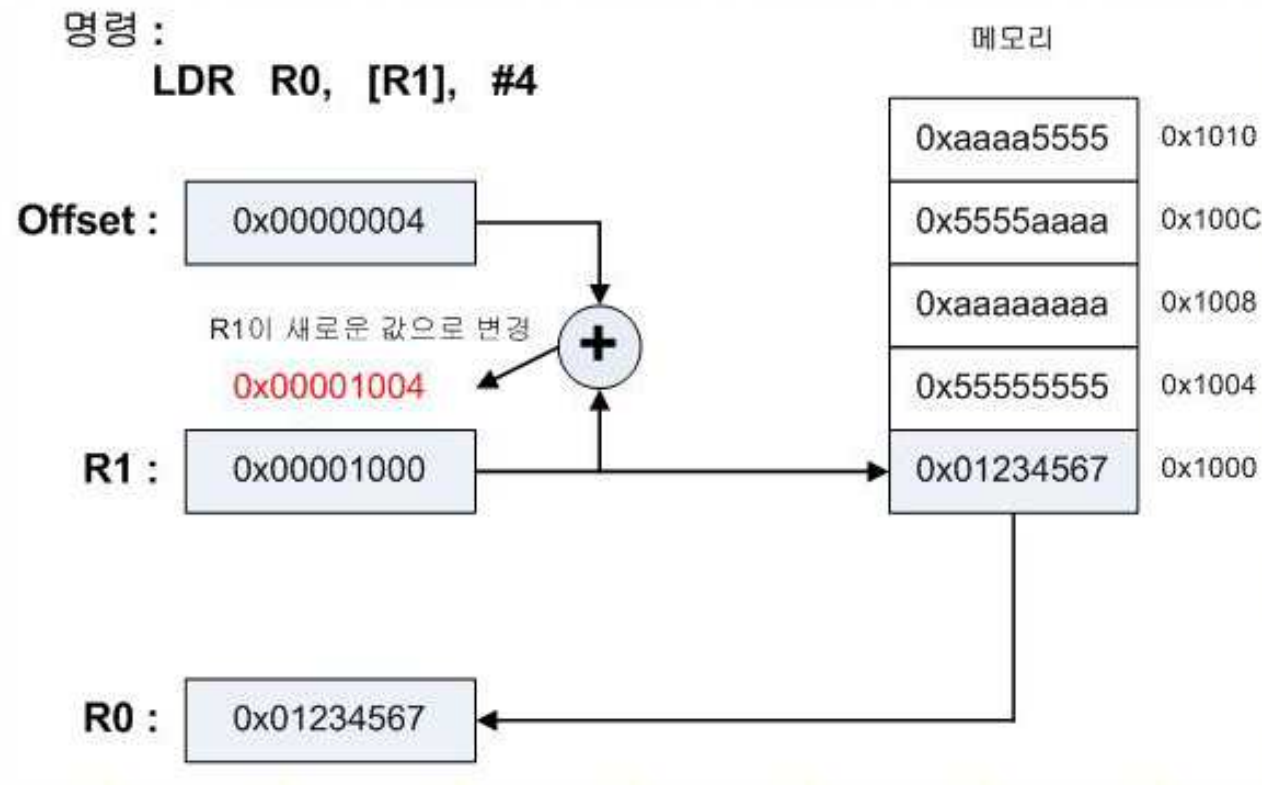
Pre-indexed 방식 과 Auto update

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



Post-indexed 어드레스 지정 방식

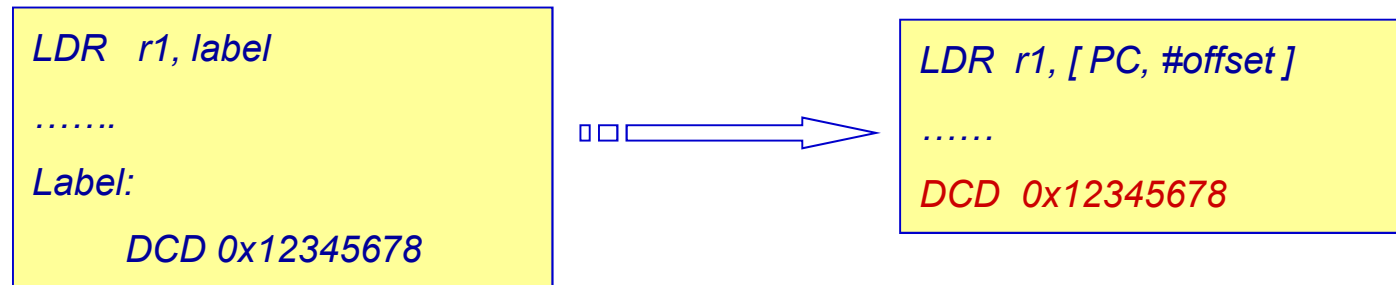
- Base 레지스터(Rn)가 지정하는 주소에 데이터의 전송 후 Rn 값과 <Offset>의 연산 결과로 Rn 갱신



PC Relative 어드레스 지정 방식

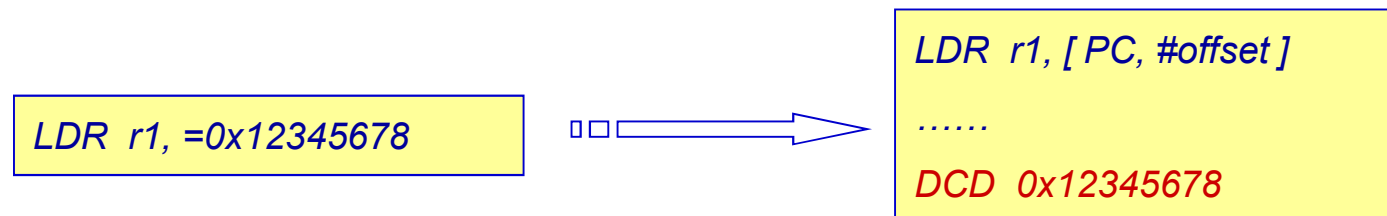
□ LABEL 지정에 의한 어드레스 지정

- ❖ 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



블록 단위 데이터 전송 명령

□ 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

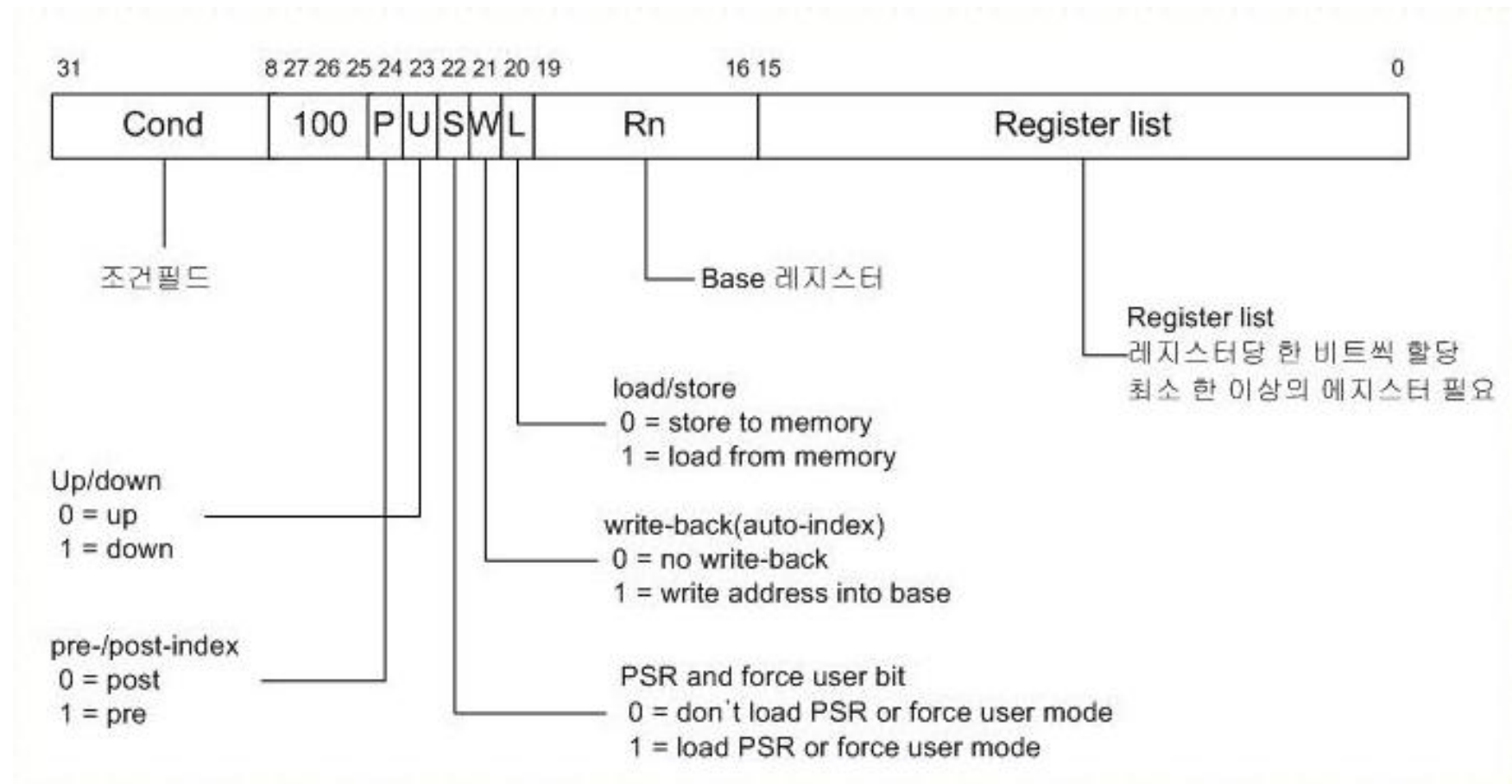
Store Multiple(STM) 명령

명령어	STM	Store Multiple
형식	<p>STM{cond}<addressing mode> Rn{!}, <register_list> {cond}는 조건부 실행을 위한 condition 조건을 나타낸다. {address mode}는 어드레스의 생성 방법을 정의 한다. Rn은 base 레지스터를 나타낸다. {!}는 데이터 전송 후 base 레지스터를 auto-update <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 위치주소부터 차례로 저장 , 동작 완료후 R0는 변화없다. STMIA R0!, {R1,R2,R3} ; R1,R2,R3의 데이터를 R0의 위치주소부터 차례로 저장 , 동작 완료후 R0:= R0+12 (3워드 주소값)로 변화.</p>	

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>	

Block Data Transfer 명령의 구조



LDM/STM의 레지스터 리스트

□ <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

`LDM r10, {r2,r3,r1}`

→
실제 동작

`LDM r10, {r1,r2,r3}`

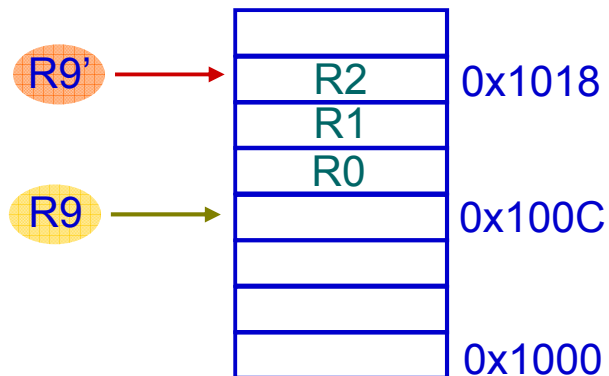
LDM/STM의 어드레스 지정 방식

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

Pre-Increment 어드레스 지정

STMIB R9!, {R0,R1,R2}

Base 레지스터(R9) = 0x100C



□ 어드레스 증가 후 데이터 저장

(Increment Before)

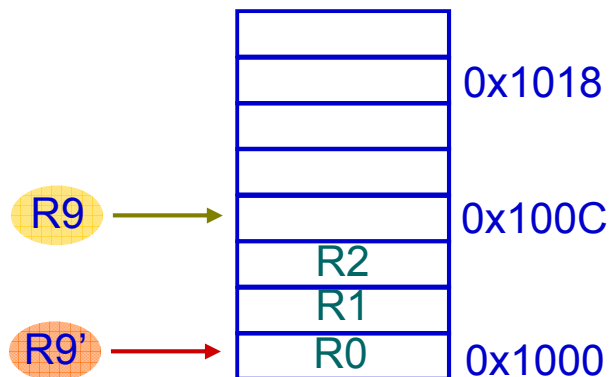
- ① R9 -> 0x1010 증가 후 R0 저장
- ② R9 -> 0x1014 증가 후 R1 저장
- ③ R9 -> 0x1018 증가 후 R2 저장
- ④ {!}, auto-update 옵션이 있으면
R9 값을 0x1018로 변경

Pre-Decrement 어드레스 지정

STMDB R9!, {R0,R1,R2}

Base 레지스터(R9) = 0x100C

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



(**D**ecrement **B**efore)

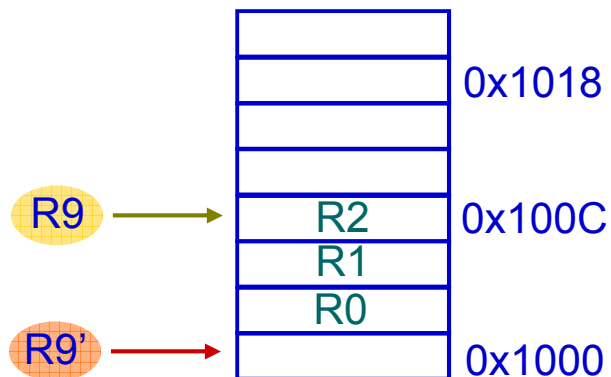
- ① 어드레스를 0x1000로 감소
- ② 0x1000에 R0 저장 후 어드레스 증가
- ③ 0x1004에 R1 저장 후 어드레스 증가
- ④ 0x1008에 R2 저장 후 어드레스 증가
- ⑤ {!}, auto-update 옵션이 있으면
R9 값을 0x1000로 변경

Post-Decrement 어드레스 지정

STMDA R9!, {R0,R1,R2}

Base 레지스터(R9) = 0x100C

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

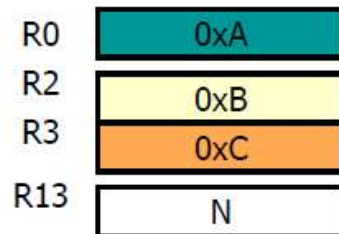


(Decrement After)

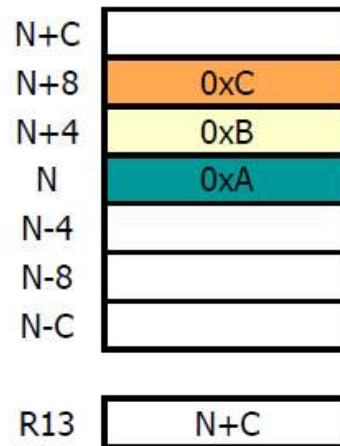
- ① 어드레스를 0x1000로 감소
- ② 어드레스 증가 후 0x1004에 R0 저장
- ③ 어드레스 증가 후 0x1008에 R1 저장
- ④ 어드레스 증가 후 0x100C에 R2 저장
- ⑤ {!}, auto-update 옵션이 있으면
R9 값을 0x1000로 변경

STM/LDM

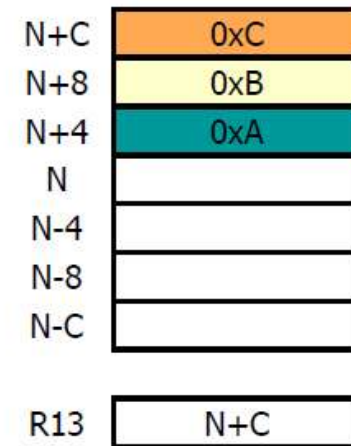
STMIA{IA|IB|DA|DB} R13!, {R0, R2-R3}



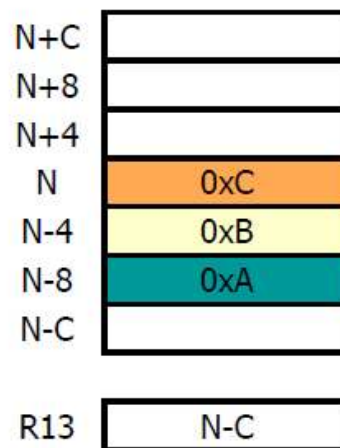
STMIA R13!, {R0,R2-R3}



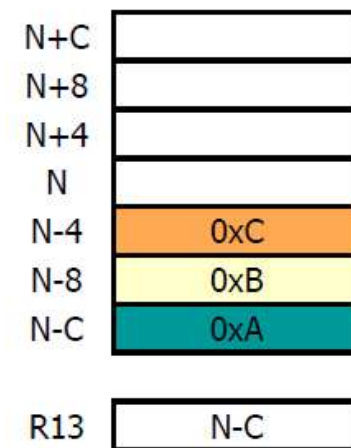
STMIB R13!, {R0,R2-R3}



STMDA R13!, {R0,R2-R3}



STMDB R13!, {R0,R2-R3}



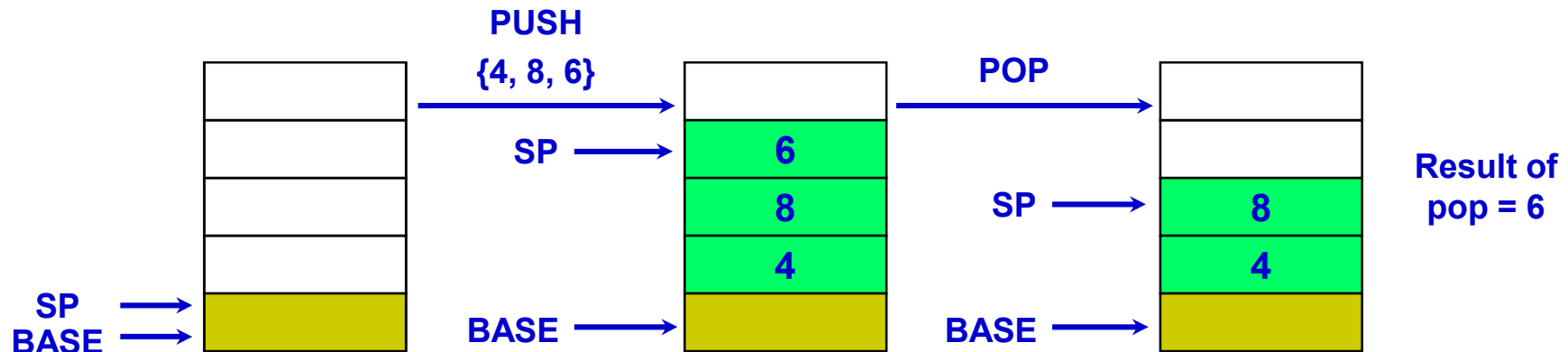
LDM/STM의 스택(Stacks) 동작

□ 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



Stack의 Type

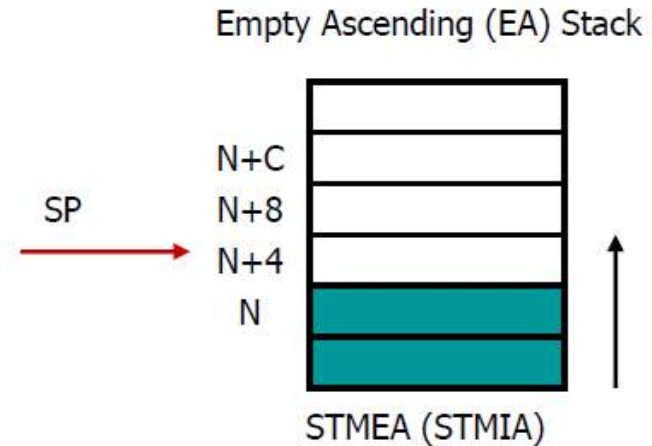
□ Stack pointer가 나타내는 위치에 따라 Stack의 형태 지정

- ❖ Full stack : Stack pointer가 Stack의 마지막 주소를 지정
- ❖ Empty stack : Stack pointer가 Stack의 다음 주소를 지정

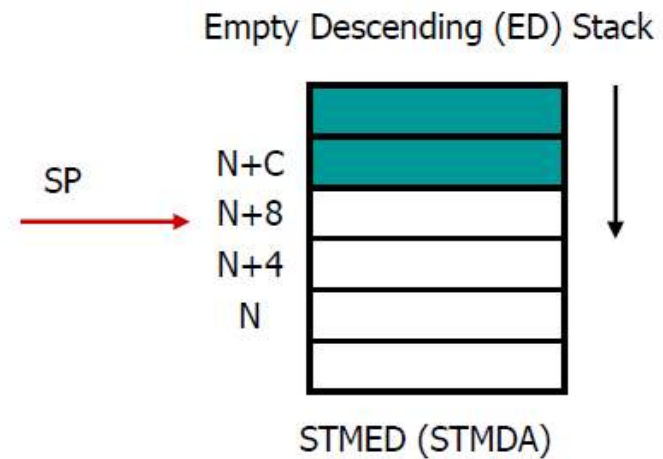
□ Stack의 Type

- ❖ Full Descending stack
 - Stack pointer가 stack의 top을 지정하고 push가 필요하면 어드레스 감소 하면서 사용
 - ARM compiler의 경우는 항상 이 방법을 사용한다.
 - STMFD / LDMFD
- ❖ Full Ascending stack
 - STMFA / LDMFA
- ❖ Empty Descending stack
 - STMED / LDMED
- ❖ Empty Ascending stack
 - STMEA / LDMEA

Ascending Stack



Descending Stack

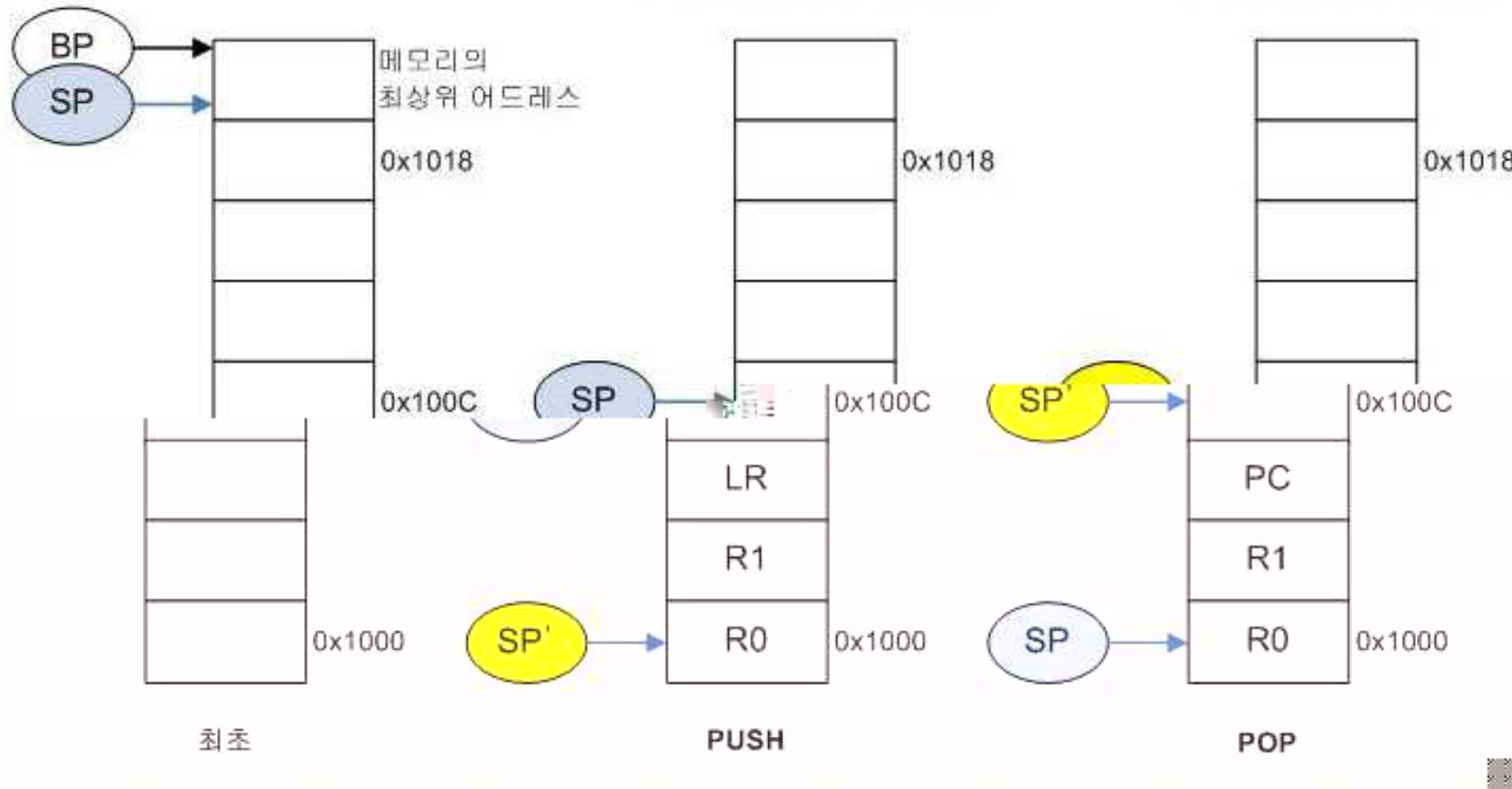


ARM의 스택 동작

BP : Base Pointer
SP : Stack Pointer

STMFD SP!, {R0, R1, LR}
스택 포인터(SP) = 0x100C

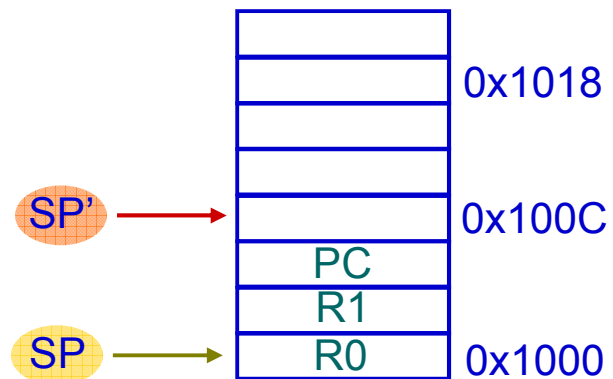
LDMFD SP!, {R0, R1, PC}
스택 포인터(SP) = 0x1000



Stack의 Pop 동작

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

**참고 : 스택과 서브루틴

- 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 **LDM**을 사용하여 **Pop**을 할 때 ‘S’ bit set 옵션인 ‘^’가 레지스터 리스트에 있으면 **SPSR**이 **CPSR**로 복사 된다.

메모리 데이터와 레지스터의 Swap 명령

명령어	SWP	Swapping Data between register and memory
형식	SWP{cond}{B} Rd,Rm,[Rn] {cond}는 조건부 실행을 위한 condition 조건을 나타낸다. {B}는 byte 단위의 데이터 교환을 나타낸다. Rd는 Destination 레지스터로 메모리의 내용이 기록된다. Rn은 base 레지스터를 나타낸다. Rm은 메모리에 기록될 데이터가 저장된 레지스터 이다.	
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, r1 을 r2의 어드레스에 저장한다.	

소프트웨어 인터럽트(SWI) 명령

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



****참고 : SWI 명령의 호출**

□ 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

Coprocessor 관련 명령

□ ARM은 16개의 Coprocessor 지원

- ❖ 기능과 명령의 확장을 위해

□ Coprocessor 관련 명령

❖ Coprocessor 와 ARM 사이 레지스터 내용 전송 명령

➤ MRC 명령

- ✓ 코프로세서 레지스터 내용을 **ARM** 레지스터로 전송

➤ MCR 명령

- ✓ **ARM** 레지스터의 내용을 코프로세서 레지스터로 전송

❖ Coprocessor 와 외부 메모리 사이 데이터 전송 명령

➤ LDC 명령

- ✓ 메모리에서 코프로세서로 데이터 **load**

➤ STC 명령

- ✓ 코프로세서에서 메모리에 데이터 **store**

❖ Coprocessor 데이터 처리 명령

➤ CDP 명령

- ✓ 코프로세서의 데이터 처리 초기화

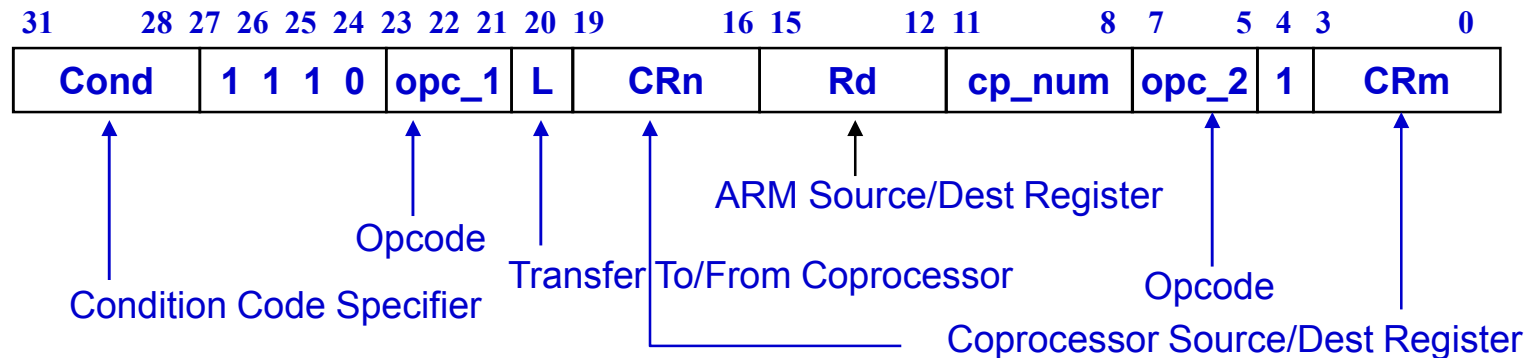
Coprocessor 레지스터 전송 명령

□ Coprocessor 와 ARM 사이 레지스터 내용 전송 명령

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

□ 문법

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



프로세서의 상태 변경 명령

□ 프로세서의 상태(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 0 에는 ARM state로 변환 되는지 아니면 Thumb state로 변환 되는지를 나타낸다.	
동작	Rn이 지정하는 어드레스로 분기하고 state를 변환한다.	
사용 예	BX r0+1 ; r0가 지정하는 위치로 분기하고, Thumb state로 변환 BX r0 ; r0가 지정하는 위치로 분기하고, ARM state로 변환, 이때 비트 0는 0이다.	

아키텍처 v5TE의 명령어

Architecture	특징	프로세서
v1, v2, v3	이전 버전, 현재는 거의 사용 않됨	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	ARM9E-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

Count Leading Zero 명령

□ CLZ 명령

- ❖ MSB로부터 최초 1이 나타내는 위치 검색

□ CLZ 명령 응용

- ❖ 인터럽트 Pending 비트 검사
- ❖ Normalize

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

Saturate 연산

□ 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 사용하면 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 Multiply 명령

□ 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]	

❖ x,y : B 또는 T
 ➤ B는 bottom, T는 top

SMULTBEQ R8, R7, R9 ; EQ 조건이면 R7[31:16]과 R9[15:0]을 곱해서
 ; R8에 저장한다.

SMULWB R2, R4, R7 ; R4와 R7[15:0]를 곱해서 R2에 저장한다.

Signed Multiply-Accumulate 명령

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

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

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

SMLALTB R1, R2, R7, R8 ; R7[31:16]과 R8[15:0]를 곱하고
; R1, R2 값을 더해서 R1, R2에 저장한다.

Doubleword 단위 LDR / STR

□ Doubleword (64비트) 단위의 LDR / STR 지원

❖ Destination 레지스터는 2개씩 사용되는데, 짝수 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>	

LDRD R6, [R10, #0x10] ; R10+0x10 위치의 한 워드를 R6에 읽어오고
; R10+0x14의 한 워드는 R7에 읽어온다.
STRD R6, [R10, #0x10] ; R6의 내용을 R10+0x10의 위치에 저장하고
; R7의 내용을 R10+0x14에 저장한다.

프로세서의 **state** 변경 명령

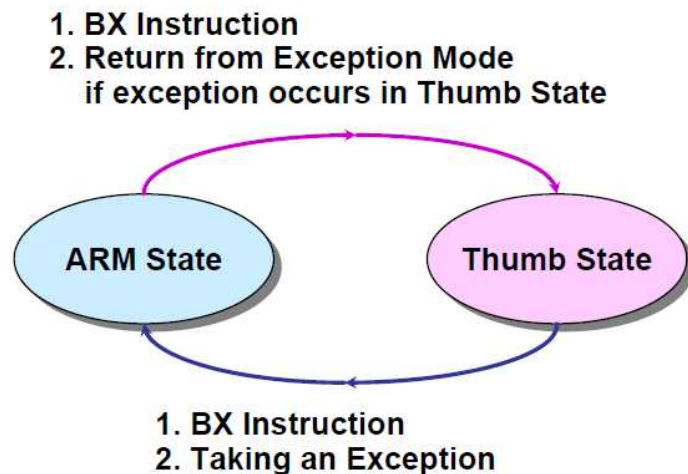
□ Architecture v4T의 **state** 변경 명령

❖ BX 명령 제공

❖ 서브루틴 콜 하는 경우

링크 레지스터에 되돌아갈

주소를 저장하는 기능이 없다.



```
ADR r0, into_T+1 ; 분기주소 지정  
BX r0             ; into_T 상태로 분기  
CODE16           ; thumb 명령어
```

into_T

```
.....  
ADR r5, back_to_ARM ; ARM 주소 지정  
BX r5
```

```
ALIGN  
CODE32 : ARM 명령어  
back_to_ARM  
.....
```

□ Architecture v5TE의 **state** 변경 명령

❖ BLX 명령

❖ State의 변경과 링크 레지스터에 되돌아갈 주소 저장

BLX 명령

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

BLXNE R2 ; NE 조건이면 R2가 지정하는 위치로 분기하면서
 ; 비트 0의 정보에 따라 상태를 전환하고
 ; 되돌아올 주소를 LR(R14)에 저장한다.

BLX thumfunc ; thumfunc로 분기하면서 Thumb 상태로 전환하고
 ; LR에 되돌아올 주소를 저장한다.

16비트 Thumb 명령어

□ 장점

- 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

□ 단점

- 조건부 실행이 안된다.
- 사용되는 레지스터가 R0~R7로 제한된다.
- Immediate 상수 값의 사용 범위가 제한적이다.
- Inline barrel shifter를 사용할 수 없다.

Thumb 명령의 성능 비교

□ ARM 명령과 Thumb 명령의 메모리 인터페이스에 따른 성능 비교

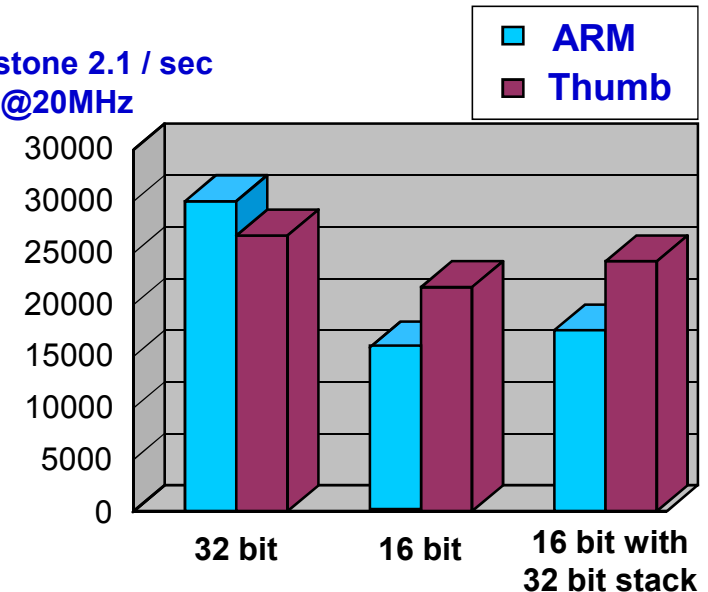
□ Dhrystone 2.1

- ❖ 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

Dhrystone 2.1 / sec
@20MHz



ARM or Thumb ?

□ Thumb 명령 사용의 필요성

- ❖ 이미지의 크기가 작고, 좁은 메모리 인터페이스에서 높은 성능 보유

□ Thumb 명령 만으로는 사용 불가

- ❖ 일부 명령, PSR 전송명령, Coprocessor 명령 등을 위한 Thumb 명령이 없다
- ❖ Exception 발생하면 ARM 프로세서는 ARM state로 무조건 전환된다.
- ❖ 따라서 **Thumb** 명령만 사용하고자 하여도 항상 **ARM** 명령과 같이 사용된다.

□ ARM state와 Thumb state의 변경

- ❖ Interworking 이라 한다.
- ❖ BX 명령 사용한다. → ARM 명령어의 BX 명령어 설명 참조

□ Thumb 명령어 → 어셈블리어로 직접 작성하지 않는다

- ❖ C로 프로그램하고 컴파일 과정에서 Thumb 명령어로 컴파일

ARM Pseudo Instruction

□ ADR/ADRL

- ❖ Load a program-relative or register-relative address into a register
- ❖ ADR: generates one instruction (ADD or SUB)
 - +/-255 bytes for non-aligned address
 - +/-1020 bytes for word-aligned address
- ❖ ADRL: generates two data processing instructions
 - +/-64KB for non-aligned address
 - +/-256KB for word-aligned address

□ Syntax

- ❖ ADR{<cond>}{L} <Rd>, <expr>
<expr> := a program-relative or register-relative expression

□ Example

```
start      MOV    r0, #10
           ADR     r4, start          ; => SUB r4, pc, #0xC
           ADRL    r5, start + 60000 ; => ADD r5, pc, #0xe800
           ADD     r5, r5, #0x250
```

ARM Pseudo Instruction

❑ LDR

- ❖ Load a register with either
 - a 32-bit constant value or an address

❑ Syntax

- ❖ LDR{<cond>} <Rd>, =[<expr>|<label_expr>]

❑ Example

```
LDR r1,=0xfff           ; LDR r1, [pc, offset_to_litpool]
                        ; ...
                        ; litpool DCD 0xfff
LDR r2,=place           ; LDR r2, [pc, offset_to_litpool]
                        ; ...
                        ; litpool DCD place
```

Directives

□ ARM Assembly Language Basic

❖ Simple Example

```
AREA    ARMex, CODE, READONLY
; Name is ARMex
ENTRY   ; Mark first instruction to execute
start
MOV     r0, #10        ; Set up parameters
MOV     r1, #1         ; Set up parameters
ADD     r0, r0, r1      ; r0 = r0+r1
; ...
MOV     r0, #0x18
LDR     r1, =0x20026     ; Application Exit
SWI     0x123456         ; ARM semihosting SWI
; ...
END      ; Mark end of file
; ...
```

ARM Assembly Language Basic

- ❑ Line

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

- ❑ Case rules

- ❖ can be upper or lower cases, but not mixed

- ❑ Line length

- ❖ long line can be split onto several lines by backslash (~~W~~)

- but limited by 128 ~ 256

- ❑ Labels

- ❖ symbols that represent address

- calculated during assembly

- ❑ Comments

- ❖ ;

ARM Assembly Language Basic

❑ Constants

❖ Numbers

- Decimal: 123
- Hexadecimal: 0x78
- n_xxx
 - ✓ *n: base between 2 and 9*
 - ✓ *xxx: number in that base*

❑ Boolean

❖ {TRUE}, {FALSE}

❑ Characters

❖ 'a', 'b', ...

❑ Strings

❖ "abcde123 \$\$ xyz"

Directives

□ AREA

- ❖ instructs assembler to assemble a new code or data section
la mnrv a new code new code new code or CODE, DAT 3
- ❖ sections are independent indivisible chunks of code or data
manipulated by the linker
- ❖ CODE, DAT 3 3

Directives

❑ DCB (=), DCW, DCD (&), DCQ, DCI

- ❖ Allocates 1, 2, 4, 8 and integer of memory, respectively
- ❖ Integer defining an ARM or Thumb instruction

{label} DCx expr {, expr}

➤ expr : numeric expression, program-relative expression

❑ Example

Data1	DCD	1, 5, 20	; defines 3 word of 1, 5, 20
Hello	DCB	"hello", 0	
Data2	DCD	mem06+4	; defines 1 word of label mem06 + 4
MyVal	DCI	0xEA000000	

Directives

❑ MACRO and MEND

- ❖ Start of macro definition and End

```
MACRO
{$label} macroname {$parameter {,$parameter}...}
; code
MEND
```

- ❖ \$label: parameter that is substituted with a symbol given when the macro is invoked → usually a label
- ❖ macroname: macro name
- ❖ \$parameter: substituted when the macro is invoked

❑ Example

```
$label    MACRO                ; Definition
          test $p1, $p2
          mov $p1, $p2
          MEND
abc       test r1, r2           ; Usage
                                   ; = abc   movr1, r2
```

Program Example

❑ Calling subroutine

```
AREA    subrout, CODE, READONLY

        ENTRY
start    MOV     r0, #10
        MOV     r1, #3
        BL      doadd
stop     MOV     r0, #0x18
        LDR     r1, =0x20026
        SWI     0x123456

doadd    ADD     r0, r0, r1
        MOV     pc, lr
        END
```

Program Example

❑ Thumb assembly language

```
AREA    ThumbSub, CODE, READONLY

        ENTRY
        CODE32

header  ADR    r0, start+1
        BX     r0

        CODE16

start   MOV     r0, #10
        MOV     r1, #3
        BL      doadd

stop    MOV     r0, #0x18
        LDR     r1, =0x20026
        SWI     0xAB                ;Thumb semihosting SWI

doadd   ADD     r0, r0, r1
        MOV     pc, lr
        END
```

□ 예외처리와 시스템 리셋

ARM 프로세서의 예외처리

예외처리

시스템 리셋

ARM 프로세서의 Exception

□ 예외처리(Exception)

- ❖ 외부의 요청이나 오류에 의해서 정상적으로 진행되는 프로그램의 동작을 잠시 멈추고 프로세서의 동작 모드를 변환하고 미리 정해진 프로그램을 이용하여 외부의 요청이나 오류에 대한 처리를 하도록 하는 것
- ❖ Exception의 예
 - I/O 장치에서 인터럽트를 발생시키면 IRQ Exception이 발생하고, 프로세서는 발생한 IRQ Exception을 처리하기 위해 IRQ 모드로 전환되어 요청된 인터럽트에 맞는 처리 동작 수행

□ ARM의 Exception

- ❖ Reset
- ❖ Undefined Instruction
- ❖ Software Interrupt
- ❖ Prefetch Abort
- ❖ Data Abort
- ❖ IRQ(Interrupt Request)
- ❖ FIQ(Fast Interrupt Request)

예외처리 Vector 와 우선순위

□ 예외처리 벡터

- ❖ Exception이 발생하면 미리 정해진 어드레스의 프로그램을 수행
- ❖ 미리 정해진 프로그램의 위치를 Exception Vector라 한다.

□ 예외처리 벡터 테이블

- ❖ 발생 가능한 각각의 Exception에 대하여 Vector를 정의해 놓은 테이블
 - 각 Exception 별로 1 word 크기의 명령어 저장 공간을 가진다.
- ❖ Vector Table에는 Branch 또는 이와 유사한 명령어로 실제 Exception을 처리하기 위한 루틴으로 분기 할 수 있는 명령어로 구성되어 있다.
 - FIQ의 경우는 Vector Table의 맨 상위에 위치하여 분기명령 없이 처리루틴을 프로그램 할 수 있다.
- ❖ ARM은 기본적으로 0x00000000에 Vector Table을 둔다.
(MMU 제어 프로그램에 의해 위치 변경 가능)

□ Exception 우선 순위

- ❖ 동시에 Exception이 발생하는 경우 처리를 위해 우선 순위 지정

Exception Vector Table

Exception	Vector Address	우선순위	전환되는 동작모드
Reset	0x0000 0000	1 (High)	Supervisor(SVC)
Undefined Instruction	0x0000 0004	6 (Low)	Undefined
Software Interrupt(SWI)	0x0000 0008	6	Supervisor(SVC)
Prefetch Abort	0x0000 000C	5	Abort
Data Abort	0x0000 0010	2	Abort
Reserved	0x0000 0014		
IRQ	0x0000 0018	4	IRQ
FIQ	0x0000 001C	3	FIQ

예외처리 벡터에 사용되는 명령어

□ Exception 마다 1 word 크기의 명령어 저장 공간 할당

- ❖ Exception vector 테이블에서는 1 개의 ARM 명령만을 사용할 수 있다.
 - 실제 Exception Handler가 있는 분기 명령으로 만들어 진다.
- ❖ FIQ의 Vector Table은 맨 상위에 있으므로 핸들러를 직접 작성할 수 있다.

□ Exception Vector Table에서 사용할 수 있는 명령어

- ❖ Branch 명령 (B)
 - 가장 일반적으로 사용된다.
 - Branch 명령은 PC 값을 기준으로 +/- 32MB 내에 있어야 한다.
 - ✓ *Handler가 32MB 영역을 벗어나면 다른 명령을 사용하여야 한다.*
- ❖ Move 명령 (MOV)
 - Destination 레지스터를 PC로 하면 Branch 명령과 같이 사용 가능
 - 한 사이클 내에 처리된다.
 - Handler 어드레스가 8비트 상수와 ROR로 표시 가능해야 사용 가능하다.
- ❖ Load 명령(LDR)과 Literal Pool
 - 메모리 영역 내의 어떤 위치라도 이동이 가능하다.
 - 메모리에서 주소를 읽기 위한 1 사이클이 더 필요하다.

예외처리 벡터와 명령어 파이프라인

□ 예외처리와 명령어 파이프라인

- ❖ Exception 종류별로 CPU에서 발생한 Exception을 인식하는 시점의 pipeline stage가 모두 다르다.
- ❖ 따라서 복원 되는 PC 값의 조정이 필요하다.

□ 예외처리 복귀 명령

Exception	Pipeline Stage	Return 명령
Undefined Instruction	Decode stage	MOV S PC, LR
Software Interrupt(SWI)	Decode stage	MOV S PC, LR
Prefetch Abort	Execute stage	SUB S PC, LR, #4
Data Abort	Memory stage	SUB S PC, LR, #8
IRQ	Execute stage	SUB S PC, LR, #4
FIQ	Execute stage	SUB S PC, LR, #4

예외처리 복귀

□ Exception 처리가 완료되면 다음의 절차를 수행하여 복귀

1. LR_<mode> 값을 PC에 복사한다.
 2. SPSR_<mode>를 CPSR에 복사한다.
- ❖ 주의사항 : 2가지 동작이 하나의 명령으로 처리되어야 한다.

□ Exception 복귀 명령

- ❖ Data processing 명령에 S 접미사를 사용하고 PC를 destination 레지스터로 사용하는 방법

- Privilege 모드에서 S 접미사를 사용하면 CPSR 복원

SUBS PC, LR, #4

- ❖ LDM 명령을 사용하고 register list 뒤에 ^ 옵션 사용

- Stack에 되돌아갈 주소 값이 계산 되어 들어가 있어야 한다.

LDM SP!, {PC}^

예외처리 : 리셋 예외처리

□ 발생 조건

❖ ARM로 Reset 신호가 입력되면 Reset Exception 발생

□ 예외처리 발생에 따른 **ARM**의 동작

순서	동 작		동 작 설 명
1	SPSR_svc = CPSR		
2	CPSR 값 변경	CPSR[5] = 0	ARM state로 전환
		CPSR[6] = 1	FIQ disable
		CPSR[7] = 1	IRQ disable
		CPSR[4:0] = 10011'b	SVC 모드로 전환
3	LR_svc = unpredictable value		
4	PC = 0x00		

Undefined Instruction 예외처리

- 발생 조건
 - ❖ ARM에 정의 되지 않은 명령을 실행 하고자 하면 발생
 - ❖ Coprocessor에서 응답이 없으면 발생
- 예외처리 발생에 따른 **ARM**의 동작

순서	동 작		동 작 설 명
1	SPSR_undef = CPSR		
2	CPSR 값 변경	CPSR[5] = 0	ARM state로 전환
		CPSR[6] = no change	
		CPSR[7] = 1	IRQ disable
		CPSR[4:0] = 11011'b	Undefined 모드로 전환
3	LR_undef = address of undefined inst + 4		
4	PC = 0x04		

- Return 명령
 - ❖ MOV_S PC, LR

소프트웨어 인터럽트 예외처리(SWI)

- 발생 조건
 - ❖ SWI 명령이 실행되면 Exception 발생
- 예외처리 발생에 따른 **ARM**의 동작

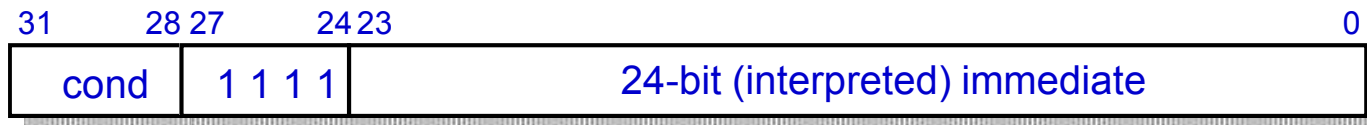
순서	동 작		동 작 설 명
1	SPSR_svc = CPSR		
2	CPSR 값 변경	CPSR[5] = 0	ARM state로 전환
		CPSR[6] = no change	
		CPSR[7] = 1	IRQ disable
		CPSR[4:0] = 10011'b	SVC 모드로 전환
3	LR_svc = address of SWI + 4		
4	PC = 0x08		

- Return 명령
 - ❖ MOV_S PC, LR

SWI Handler

□ SWI 처리 순서

- ❖ SWI 명령이 있는 위치를 찾는다.
- ❖ SWI 명령에 있는 SWI 번호를 알아낸다.



- ❖ SWI 번호에 해당하는 동작을 수행한다.
- ❖ SWI의 Argument는 R0, R1, R2, R3을 통해 전달된다.

Prefetch Abort 예외처리

□ 발생 조건

- ❖ 잘못된 어드레스 공간에서 명령을 읽으려고 하면 Exception 발생
- ❖ MMU나 메모리 controller서 발생한 Abort 신호 입력에 의해서 발생

□ 예외처리 발생에 따른 ARM의 동작

순서	동 작		동 작 설 명
1	SPSR_abt = CPSR		
2	CPSR 값 변경	CPSR[5] = 0	ARM state로 전환
		CPSR[6] = no change	
		CPSR[7] = 1	IRQ disable
		CPSR[4:0] = 10111'b	Abort 모드로 전환
3	LR_abt = address of abort +4		
4	PC = 0x0C		

□ Return 명령

- ❖ SUBS PC, LR, #4

Data Abort 예외처리

□ 발생 조건

- ❖ 잘못된 어드레스 공간에서 데이터를 읽거나 쓰려고 하면 Exception 발생
- ❖ MMU나 메모리 controller서 발생한 Abort 신호 입력에 의해서 발생

□ 예외처리 발생에 따른 ARM의 동작

순서	동 작		동 작 설 명
1	SPSR_abt = CPSR		
2	CPSR 값 변경	CPSR[5] = 0	ARM state로 전환
		CPSR[6] = no change	
		CPSR[7] = 1	IRQ disable
		CPSR[4:0] = 10111'b	Abort 모드로 전환
3	LR_abt = address of abort + 8		
4	PC = 0x10		

□ Return 명령

- ❖ SUBS PC, LR, #8

IRQ 예외처리

□ 발생 조건

❖ 외부 장치에서 발생한 인터럽트(IRQ) 신호가 ARM에 입력되면 Exception 발생

□ 예외처리 발생에 따른 ARM의 동작

순서	동 작		동 작 설 명
1	SPSR_irq = CPSR		
2	CPSR 값 변경	CPSR[5] = 0	ARM state로 전환
		CPSR[6] = no change	
		CPSR[7] = 1	IRQ disable
		CPSR[4:0] = 10010'b	IRQ 모드로 전환
3	LR_irq = address of next inst + 4		
4	PC = 0x18		

□ Return 명령

❖ SUBS PC, LR, #4

FIQ 예외처리

□ 발생 조건

❖ 외부 장치에서 발생한 인터럽트(FIQ) 신호가 ARM에 입력되면 Exception 발생

□ 예외처리 발생에 따른 ARM의 동작

순서	동 작		동 작 설 명
1	SPSR_fiq = CPSR		
2	CPSR 값 변경	CPSR[5] = 0	ARM state로 전환
		CPSR[6] = 1	FIQ disable
		CPSR[7] = 1	IRQ disable
		CPSR[4:0] = 10001'b	FIQ 모드로 전환
3	LR_fiq = address of next inst + 4		
4	PC = 0x1C		

□ Return 명령

❖ SUBS PC, LR, #4

시스템 **reset** : 프로세서의 **reset** 과 부트 코드

□ 프로세서에 리셋 신호가 입력되면 실행 중이던 명령을 멈추고

- ① SPSR_svc에 CPSR 값을 복사
- ② CPSR의 값을 변경
 - Mode bit M[4:0]를 Supervisor 모드인 10011'b로 변경
 - I 비트와 F 비트를 1로 세트하여 인터럽트를 disable
 - T 비트를 0으로 클리어하여 ARM state로 변경
- ③ PC 값을 LR_svc 레지스터에 복사
- ④ PC 값을 Reset Vector 어드레스인 0x00000000으로 변경
- ⑤ **Reset** 핸들러로 분기하여 시스템의 초기화 수행

리셋 handler

- ❑ **Reset Vector**에서 분기된 어셈블리어로 작성된 처리 루틴
- ❑ 주요 동작
 - ❖ 하드웨어 설정
 - 불필요한 하드웨어 동작 중지
 - 시스템의 클럭 설정
 - 메모리 제어기 설정
 - 필요에 따라 MMU나 MPU 설정
 - ❖ 소프트웨어 동작 환경 설정
 - 예외처리
 - 스택(stack) 영역 설정
 - 힙(heap) 영역 설정
 - 변수영역 초기화
- ❑ 리셋 핸들러의 마지막에서는 **main()** 함수와 같은 **C**로 구성된 함수를 호출
- ❑ 이 부분을 **startup 코드** 또는 **부트코드**라고 부른다.

부트 코드

□ 부트 코드를 작성하기 위해서는

- ❖ Programmer's model, 특히 명령어

- ❖ 시스템 하드웨어 구조 및 기능에 대한 전반적인 사항

등을 모두 알아야 한다.

ARM9TDMI 프로세서

- **ARM Architecture v4T** 사용
- **5** 단의 **pipeline**을 사용
- **Harvard Architecture** 사용
 - ❖ 메모리 인터페이스 분리
 - 명령어를 읽는 메모리 인터페이스
 - 데이터를 읽고 쓰기 위한 메모리 인터페이스
 - ❖ 동시에 명령어의 **fetch**와 데이터의 액세스가 가능
- **Clock speed** 향상
 - ❖ 반도체 기술의 발달
 - ❖ 늘어난 pipeline stage
- **1.5 CPI(Clock Cycle Per Instruction)**

ARM9TDMI 프로세서

□ ARM9TDMI는 ARM9 core에 다음과 같은 기능이 확장된 프로세서이다

❖ T : Thumb 명령 지원

➤ 32 비트 ARM 명령과 16 비트 Thumb 명령 모두 지원

❖ D : Debug 지원

➤ Core 내부에 디버그를 지원하기 위한 기능과 디버그를 위한 입출력 신호가 추가

➤ 디버거를 사용하기 위해서는 반드시 필요한 부분

❖ M : Enhanced Multiplier 지원

➤ 64 비트 결과를 낼 수 있는 32x8의 Enhanced Multiplier 지원

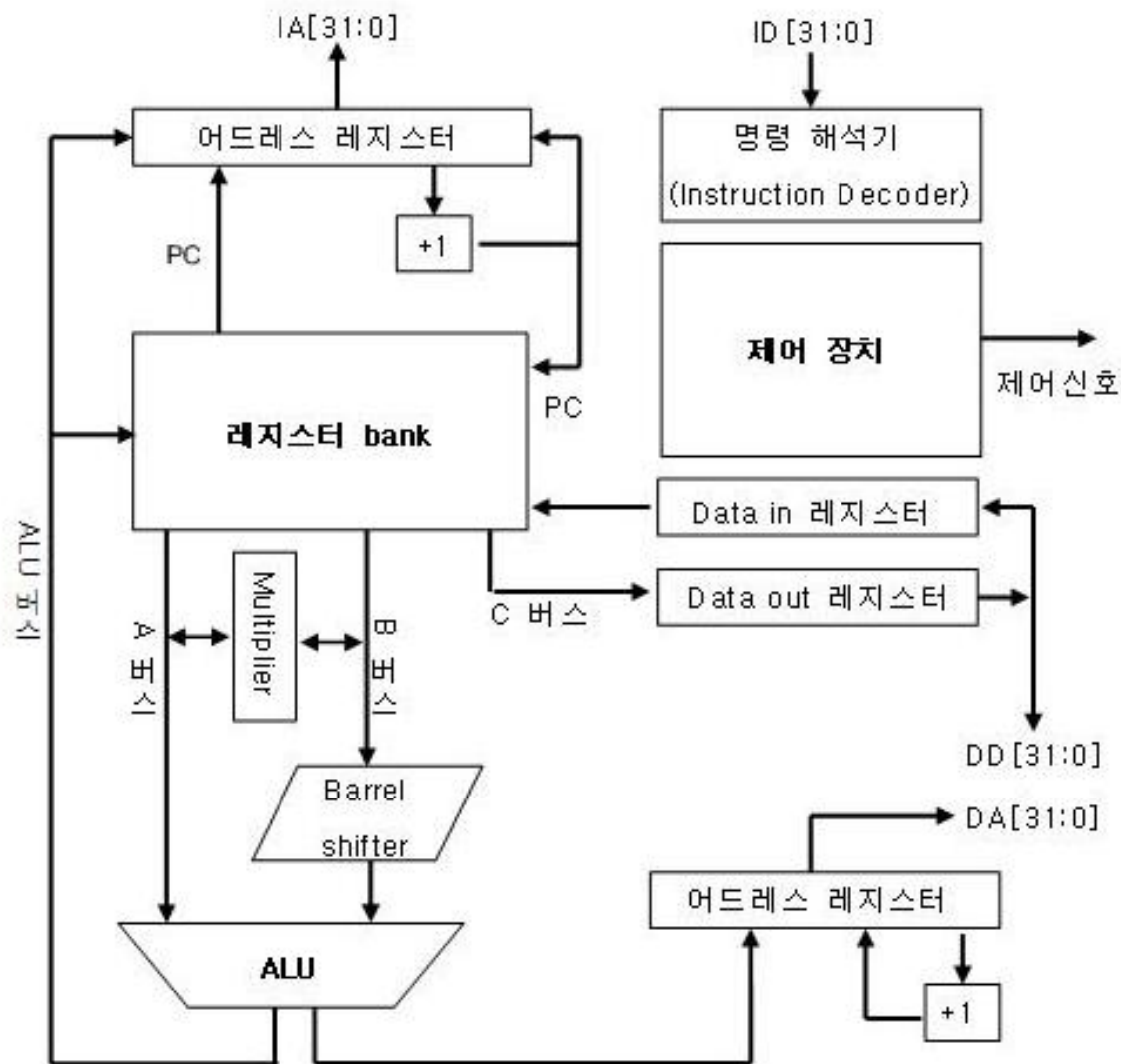
➤ 기본은 2비트, M 확장된 8비트로 계산 시간 단축

❖ I : Embedded ICE Logic

➤ 디버거를 이용하여 명령어 breakpoint나 데이터 watchpoint를 설정 가능하도록 한다.

➤ 디버거를 사용하기 위해서는 Debug 지원(D 확장)과 함께 반드시 필요

ARM9TDMI 내부 구조



ARM9TDM core 내부 구조

□ 레지스터 Bank

- ❖ 37개의 32비트 레지스터
- ❖ 데이터 처리를 위한 저장 공간으로 사용

□ 명령어 해석기

- ❖ 입력되는 명령을 해석하는 장치

□ 제어 장치

- ❖ 명령에 따라 필요한 제어 동작 수행하고 필요에 따라 제어 신호를 외부로 구동

□ ALU

- ❖ 32비트 산술 및 논리 연산 수행

□ Multiplier

- ❖ 32비트의 Booth's 곱셈기가 연결되어 32비트 곱셈 연산 수행

ARM9TDM 코어 내부 구조

□ 어드레스 Incrementer

- ❖ 순차적인 명령어를 fetch 할 때 사용 (Instruction 용)
- ❖ LDM 이나 STM 같은 block 단위의 데이터 전송명령이 실행될 때 사용 (Data 용)

□ 메모리 인터페이스 분리

❖ Harvard Architecture

- 명령어를 읽는 메모리 인터페이스
 - ✓ 어드레스 $IA[31:0]$, 데이터 $ID[31:0]$
- 데이터를 읽고 쓰기 위한 메모리 인터페이스
 - ✓ 어드레스 $DA[31:0]$, 데이터 $DD[31:0]$

ARM9TDMI 코어의 CPU 내부 버스

□ A 버스

- ❖ 레지스터 뱅크에서 바로 ALU에 연결 된다.
- ❖ Data Processing 명령의 경우 Operand 1(Source 레지스터)가 전달되는 경로
- ❖ Data Transfer 명령의 경우 Base 레지스터 값이 전달 되는 경로

□ B 버스

- ❖ 레지스터 뱅크에서 Barrel Shifter를 통하여 ALU에 연결 된다
- ❖ Data Processing 명령의 경우 Operand 2가 전달되는 경로
- ❖ Data Transfer 명령의 경우 Offset 값이 전달 되는 경로
- ❖ Barrel shifter :
 - Operand 2와 Offset 을 지정할 때 shift operation을 같이 사용할 수 있도록 한다.

ARM9TDM 코어의 CPU 내부 버스

□ C 버스

- ❖ 데이터를 위한 전용 버스

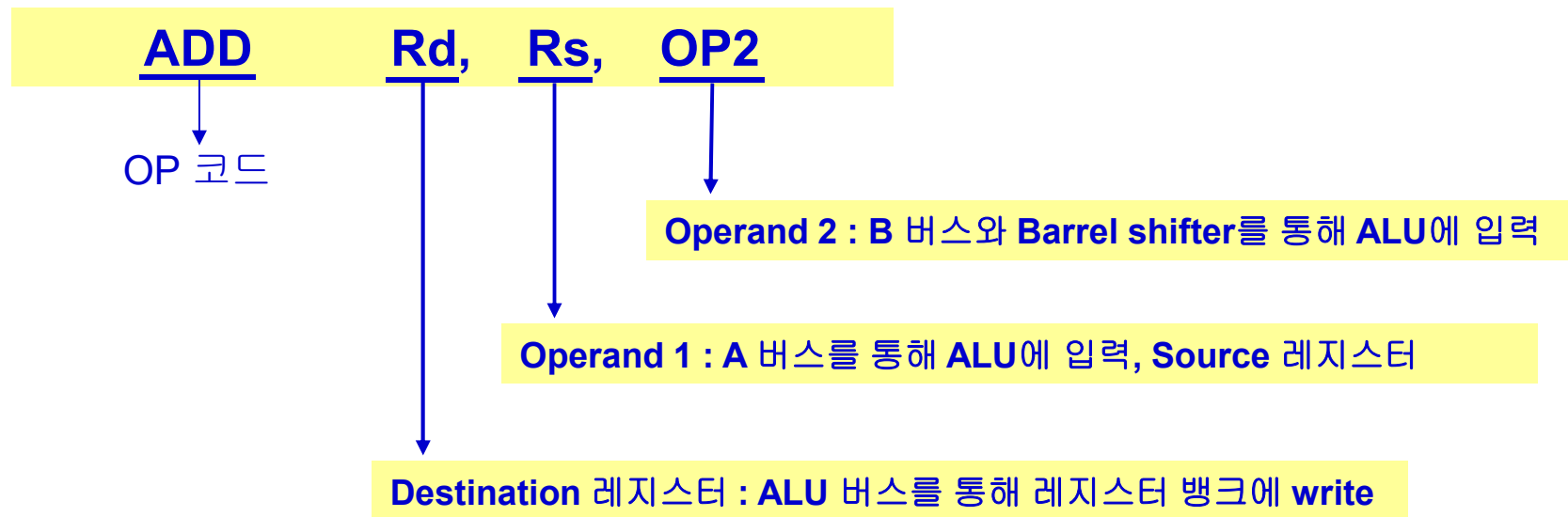
□ 데이터 입력 버스

- ❖ 외부 데이터 버스에서 데이터를 읽어 내부 레지스터로 전달하는 경로

□ ALU 버스

- ❖ ALU의 연산 결과가 전달되는 경로
- ❖ Data Processing 명령의 경우 레지스터 뱅크의 **Destination** 레지스터에 결과 값 저장
- ❖ Data Transfer 명령의 경우 어드레스 레지스터에 저장되어 어드레스를 생성

데이터 처리 명령과 내부 버스



데이터 전송 명령과 내부 버스



Multiplier 와 Shifter

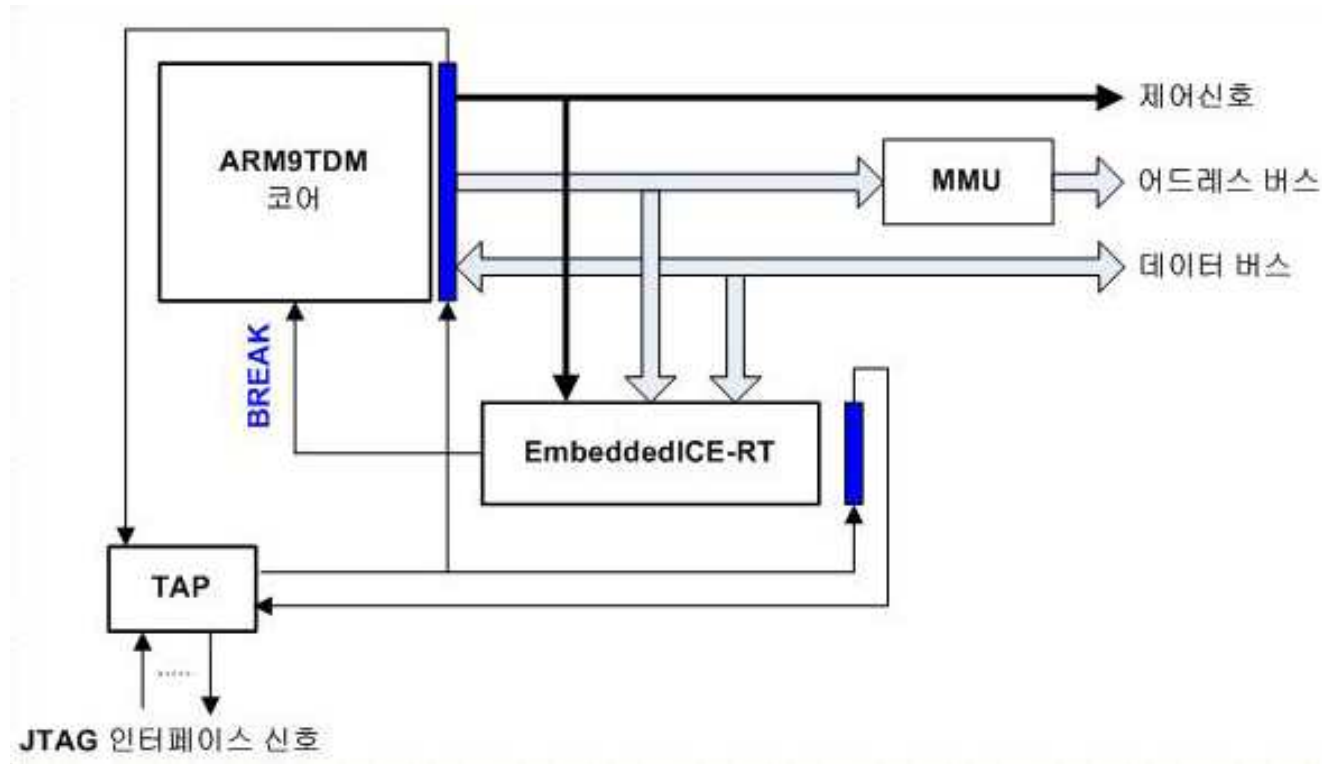
□ Booth's 곱셈기

- ❖ 32비트의 두 입력을 받아서 곱하여, 결과가 32비트를 넘더라도 넘는 부분은 버리고 32비트만 남긴다.

□ Barrel Shifter

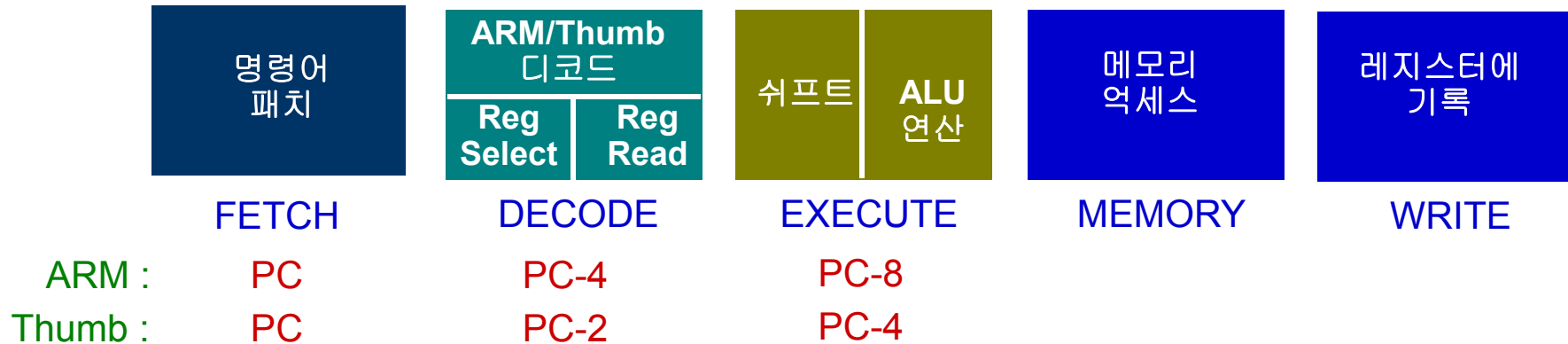
- ❖ 한번의 연산으로 데이터 워드 내에 있는 다수의 비트를 shift하거나 rotate 시킬 수 있다.
- ❖ 1 clock 사이클 동안에 최대 32비트 shift 가능

ARM9TDMI 코어와 EmbeddedICE 로직



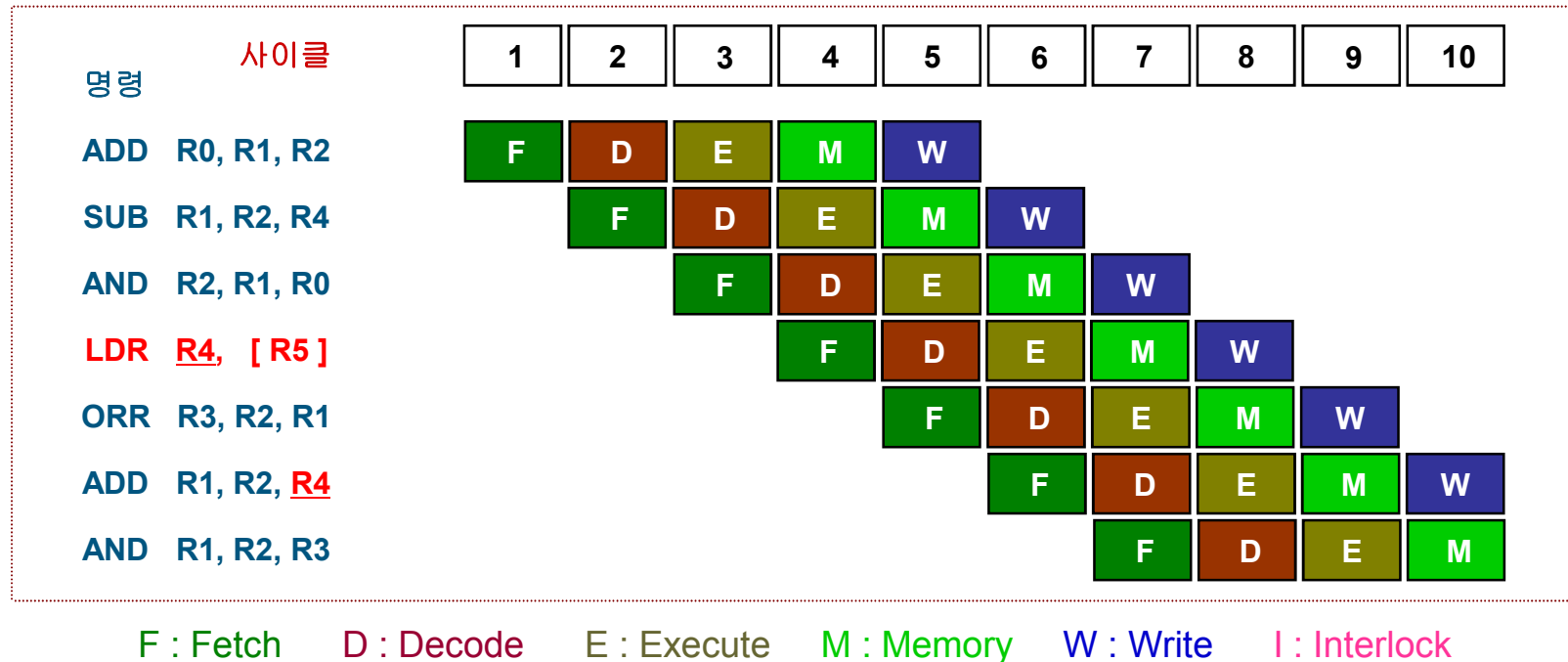
ARM9TDMI는 ARM9TDM 코어에 EmbeddedICE 로직이 추가된 프로세서이다.

ARM9TDMI 파이프라인



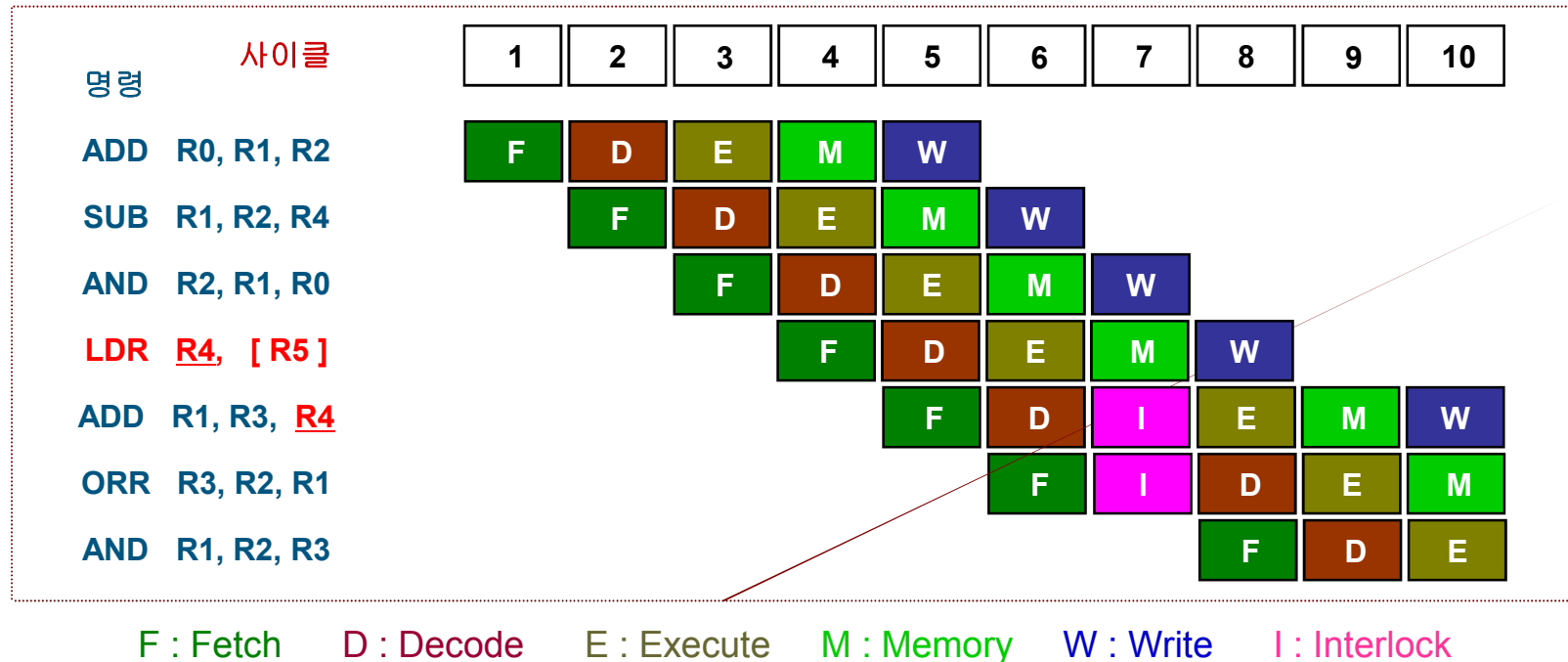
- ARM9의 경우 ARM7과 달리 32 비트 ARM 명령어와 16 비트 Thumb 명령어 **decoder**를 별도로 가지고 있다.
- **Memory stage**가 추가되어 **LDR** 명령이 사용되더라도 매 사이클마다 하나의 명령이 실행된다.

ARM9TDMI의 기본 파이프라인 동작



- LDR 명령이 사용되더라도 매 사이클마다 하나의 명령어 실행
 - ❖ CPI = 1
 - ❖ 단 LDR 명령의 Destination 레지스터가 다음 명령에서 사용되면 안된다.

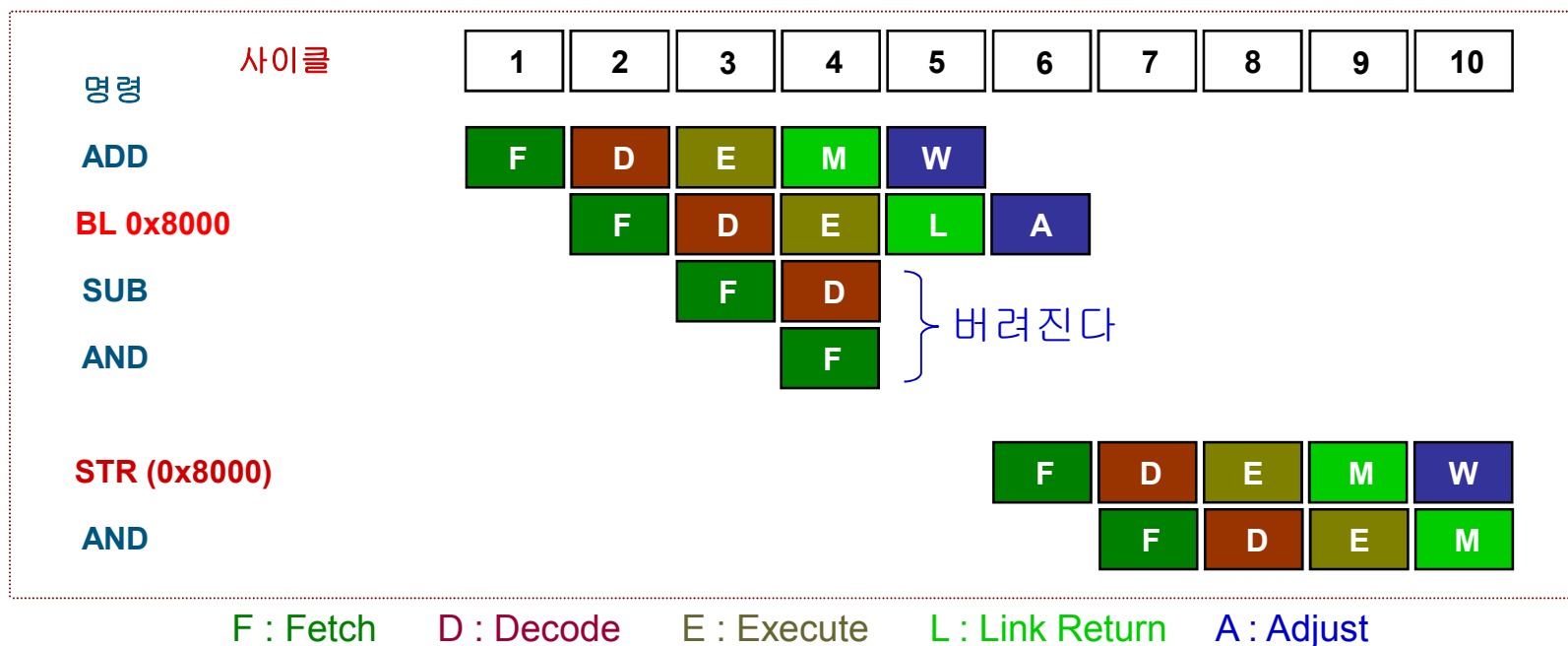
LDR 명령과 interlock 사이클



□ LDR 명령과 Interlock 사이클

- ❖ LDR 명령의 Destination 레지스터가 다음 명령에서 사용되는 경우 발생
- ❖ CPI = 1.2

분기 명령과 파이프라인 동작



□ Branch 명령이 사용되면

- ❖ Fetch한 명령을 모두 버리고 지정된 분기하여 새로운 명령을 읽는다
 - Pipeline이 깨진다(break)

ARM9TDMI 프로세서 Family

□ ARM9TDMI 프로세서 Family

- ❖ ARM9TDMI core를 사용하여 만들어진 프로세서

□ Cached ARM9TDMI Processor

- ❖ ARM9TDMI core와 Cache를 가지고 있는 프로세서

- ❖ ARM920T

- 16K Instruction/Data Cache
- Memory Management Unit (MMU)
- Write Buffer

- ❖ ARM922T ← Excalibur's core

- 8K Instruction/Data Cache
- Memory Management Unit (MMU)
- Write Buffer

- ❖ ARM940T

- 4K Instruction/Data Cache
- Memory Protection Unit (MPU)
- Write Buffer

ARM9 프로세서의 구조

