

# 3 ARM CPU Architecture

# 차례

1. Introduction to the ARM Architecture
2. Programmer's Model
3. The ARM Instruction Set
4. ARM Addressing Modes
5. ARM Instruction Set Examples

# 1. Introduction to the ARM Architecture

- 1) About the ARM Architecture
- 2) ARM Instruction Set
- 3) Thumb Instruction Set
- 4) ARM Architecture Versions

# About the ARM Architecture

- ARM은 아래와 같은 RISC(Reduced Instruction Set Computer)의 특징을 가진다.
  - A large uniform register file: 31개의 32-bit registers.
  - A load/store architecture: data processing은 register에서만 가능.
  - Simple addressing modes: 모든 load/store 주소는 instruction field 및 register 내용에 따라 정해짐.
  - Uniform fixed-length instruction fields: instruction decoding이 간단함.
- ARM은 또한 아래와 같은 특징을 추가적으로 가진다.
  - 대부분의 data processing 명령이 ALU와 shifter를 동시에 사용.
  - Auto-increment/auto-decrement addressing mode: program의 loop을 최적화함.
  - Load and store multiple instructions: data throughput을 maximize함.
  - 대부분의 instruction에서 conditional execution이 가능: execution throughput을 maximize함.

# General-Purpose Registers

- ARM은 총 31개의 32-bit general-purpose registers가 있음.
  - Processor mode에 따라 이 중 16개(R0 - R15)만 접근 가능.
  - R13, R14, R15는 각각 SP(Stack Pointer), LR(Link Register), PC(Program Counter)로 사용됨.

# Status Registers

- ARM은 총 6개의 32-bit status register가 있음.
  - 1개의 CPSR(Current Program Status Register).
  - 5개의 SPSR(Saved Program Status Register): 각 exception mode 마다 하나씩 있음.
- Status register가 표현하는 내용 (총 20 bits 사용)
  - 4 bits condition flags: N(negative), Z(zero), C(carry), V(overflow)
  - 1 bit DSP saturation flag: Q
  - 4 bits GE(greater than or equal) flags (주: ARMv6 SIMD)
  - 2 bits interrupt disable flags: I, F
  - 1 bits imprecise data abort mask: A (주: ARMv6)
  - 5 bits processor mode flags
  - 2 bits ARM/Thumb 및 Jazelle 상태 flags: T, J
  - 1 bits endian flag: E (주: ARMv6)

# Exceptions

- ARM은 7가지 종류의 exception이 있고 이들 exception에 대응되는 processor mode가 5개가 있다.

Exceptions	Processor Modes
Reset	Supervisor
Undefined instruction	Undefined
Software interrupt (SWI)	Supervisor
Prefetch abort	Abort
Data abort	Abort
IRQ: normal interrupt	IRQ
FIQ: fast interrupt	FIQ

- ARM은 위 5개 processor mode외에 user mode 및 system mode가 있다. (주: ARM은 총 7개의 processor mode가 있음.)

# ARM Instruction Set

- Branch instructions
- Data-processing instructions
- Status register transfer instructions
- Load and store instructions
- Exception-generating instructions
- Coprocessor instructions

참고: 대부분의 instruction은 4-bit로 표현되는 condition이 있어서 이 condition이 true인 경우에 instruction이 수행된다.



# Branch Instructions

- Branch instruction은 현재 위치에서  $\pm 2^{25}-1$  ( $=\pm 32\text{M}$ ) 까지 branch할 수 있다. (주: 상대적)
- Branch and link instruction는 다음 instruction의 주소(return address)를 R14(=LR)에 저장한 후 branch한다. 이 instruction은 subroutine call 시 사용된다.

참고: PC(Program Counter)가 general-purpose register R15이므로 R15에 어떤 값을 저장하면 어디든지 branch가 가능함. (주: 절대적)

# Data Processing Instructions

- ARM은 4가지 종류의 data processing instruction이 있다.
  - Arithmetic/logic instructions
  - Comparison instructions
  - SIMD(Single Instruction Multiple Data) instructions
  - Multiply instructions
  - Miscellaneous data processing instructions

# Arithmetic/Logic Instructions

- Arithmetic/Logic instruction은 2개까지의 source operand와 1개의 destination operand를 가진다.
  - Source operand 중 하나는 register이고 나머지 하나는 immediate value이든지 register이다. (주: register인 경우 shift하여 사용될 수 있음.)
  - Destination operand는 항상 register이다.

참고: ARM은 shift 전용 instruction은 없음. PC에 계산 결과 값을 저장하면 jump가 일어남.

# Comparison Instructions

- Comparison instruction은 2개의 source operand를 가진다. (주: Destination operand는 없음.)
  - Source operand 중 하나는 register이고 나머지 하나는 immediate value이든지 register이다. (register인 경우 shift하여 사용될 수 있음.)
- Comparison instruction의 수행 결과는 CPSR(Current Program Status Register)의 condition flag들의 변화로 나타난다.

# SIMD Instructions

- SIMD instruction은 multiple data에 대한 add/subtract 동작을 수행한다.
- SIMD instruction은 ARMv6에 있음.

# Multiply Instructions

- Multiply instruction은 2개의 32-bit register를 곱하여 그 결과를 1개의 32-bit register에 저장하거나 2개의 32-bit register에 저장한다.
- Multiply instruction은 동시에 accumulation을 수행할 수 있다. (예:  $R_d := (R_m * R_s) + R_n$ )

# Miscellaneous Data Processing Instructions

- Counting Leading Zeros: 이 instruction은 operand로 주어지는 register의 MSB 부분에 있는 0의 숫자를 센다.

# Status Register Transfer Instructions

- CPSR 및 SPSR은 다른 register로 복사할 수 있다.
- CPSR 내에 있는 특별한 bit들을 변경할 수 있다.
  - Condition flags 변경
  - Interrupt enable bit 변경
  - Processor mode 및 state 변경
  - Endianness 변경



# Load and Store Instructions

- Load instruction은 memory에 저장되어 있는 8-bit byte, 16-bit halfword 혹은 32-bit word를 32-bit register로 가져온다. 이 때 byte나 halfword를 사용하면 register의 내용이 자동으로 zero-extended 혹은 sign-extended가 된다.
- Store instruction은 register에 저장된 8-bit byte, 16-bit halfword 혹은 32-bit word를 memory에 저장한다.
- Load/store instruction은 memory 주소로 base register와 offset을 사용한다. (참고: 아래 3가지 선택 가능.)
  - offset: (base register 값 + offset)이 주소로 사용됨.
  - pre-indexed: (base register 값 + offset)이 주소로 사용된 후 이 주소 값이 base register에 저장됨.
  - post-indexed: base register 값이 주소로 사용된 후 offset이 base register 값에 더해짐.
- Load multiple/store multiple instruction은 block transfer를 수행한다.
- Register 값과 memory 값을 swap하는 instruction이 있다. (참고: OS에서 semaphore 구현을 위하여 atomic 수행 instruction 필요.)

# Coprocessor Instructions

- 3가지 종류의 coprocessor instruction이 있다.
  - Register transfer: coprocessor register와 ARM register 사이의 data transfer를 한다.
  - Data transfer: coprocessor register와 memory 사이의 data transfer를 한다.
  - Data processing: coprocessor 동작을 시작시킨다.

# Exception-Generating Instructions

- 2가지 종류의 exception generating instruction 이 있다.
  - Software interrupt instruction: OS service routine을 부를 때 사용하는데 ARM processor의 mode가 supervisor mode로 변경된다.
  - Software breakpoint instruction: debugger 구현에 필요한 instruction으로 ARM processor의 mode가 abort mode로 변경된다.

# Thumb Instruction Set

- Thumb instruction set은 ARM instruction set의 부분집합으로 구성되며 1개 instruction이 16-bit로 encode되어 있다.

참고: 본 slide에서는 다루지 않음.

# ARM Architecture Versions

- Version 1: Basic data processing, branch/branch with link, SWI, 26-bit address space, obsolete.
- Version 2: Multiple/multiple-accumulate instruction, coprocessor, FIQ에 2개 banked register 추가, SWP/SWPB, obsolete.
- Version 3: 32-bit address space, R15가 CPSW로 변경, SPSR 들 추가, abort/undefined mode 추가, obsolete.
- Version 4: halfword load/save 추가, ARM/Thumb state, system mode 추가, T variants 등장.
- Version 5: Count leading zeros, efficient integer division, software breakpoint, E/J variant 등장.
- Version 6: SIMD(audio/video codec), TrustZone, multi-processing support, efficient exception handling.
- Version 7: NEON advanced SIMD, vector floating point.

# Variants

- T variants: 16bits Thumbs instruction set.
- D variants: Debugging features.
- M variants: Long multiply instructions.
- I variants: In-Circuit emulator(ICE) features.
- E variants: Enhanced DSP instructions.
- J variants: Jazelle(Java bytecode execution).
- F variants: Vector floating-point(VFP) features.
- xP variants: E variants에서 5개의 instruction이 빠진 것.

# ARM Architecture Version Name

- ARM architecture version name은 아래 string를 연결하여 이름을 만든다.
  1. ARMv
  2. ARM version 숫자
  3. Variant 문자 (주: M은 version 4 혹은 그 이후 version에서 표준이므로 넣지 않음.)
  4. 각 version에서 표준인 variant가 빠진 경우 x를 먼저 쓰고 variant 문자 (예: xP)

예: ARMv5T, ARMv5xM, ARMv5TExp

# ARM Product Family

Architecture Versions	Product Family
ARMv4	ARM7TDMI, StrongARM, ARM8, ARM9TDMI
ARMv5	ARM9E, ARM10E, XScale,
ARMv6	ARM11
ARMv7	Cortex-A, Cortex-R, Cortex-M



## 2. Programmer's Model

- 1) Data Types
- 2) Processor Modes
- 3) Registers
- 4) General-Purpose Registers
- 5) Program Status Registers
- 6) Exceptions
- 7) Endian Support
- 8) Unaligned Access Support
- 9) Synchronization Primitives
- 10) The Jazelle Extensions
- 11) Saturated Integer Arithmetic

# Data Types

- ARM processor가 제공하는 3개의 data type.
  - Byte: 8 bits
  - Halfword: 16 bits
  - Word: 32 bits

## 참고:

- 위 3개의 data type은 version 4 및 그 뒤 version에서 제공됨. Version 4 이전에는 byte와 word만 제공되었음.
- ARMv6에서는 halfword 및 word에 대하여 unaligned access 제공.
- Unsigned의 경우 n-bit data는 0 to  $2^n-1$  범위를 표현.
- Signed의 경우 n-bit data는 2's complement로  $-2^{n-1}$  to  $+2^{n-1}-1$  범위를 표현.
- 대부분의 data operation은 word 단위로 수행.
- Byte나 halfword를 load하는 경우 자동으로 zero-extending 혹은 sign-extending이 일어남.
- ARM instruction의 크기는 word이며 4-byte align되고, Thumb instruction의 크기는 halfword이고 2-byte align된다.

# Processor Modes - 1/2

- ARM processor는 7개의 수행 mode가 있다.
  - ① User(usr): 일반 프로그램 수행 모드.
  - ② FIQ(fiq): 고속 data transfer 혹은 channel process 용 모드.
  - ③ IRQ(irq): 일반 목적의 interrupt 처리용 모드.
  - ④ Supervisor(svc): OS를 위한 보호 모드.
  - ⑤ Abort(abt): virtual memory 혹은 memory protection을 구현하기 위한 모드.
  - ⑥ Undefined(und): hardware coprocessor를 software로 emulation하는 경우 사용되는 모드.
  - ⑦ system(sys): OS task 수행 모드. (ARMv4 및 이후에서 제공)

# Processor Modes - 2/2

- Mode의 변화: Mode는 내부에서 exception 발생 instruction을 수행하거나 외부에서 exception이 발생하면 변화된다.
- User mode: 응용 프로그램이 수행되는 mode로 protected system resource를 access할 수 없다.
- Privileged mode: User mode가 아닌 나머지 6개 mode를 privileged mode라고 하는데 이들 mode에서는 자유롭게 system resource를 access할 수 있다.
- Exception mode: Privileged mode 중 5개인 FIQ, IRQ, Supervisor, Abort, Undefined는 특정 exception 발생 시 들어 가게 되는 exception mode라고 한다. (주: 각 exception mode는 몇개의 추가적인 register를 가지고 있다.)
- System mode: privilege를 가지는 특별한 user mode로서 user mode에서 사용되는 같은 register를 사용한다.

# Registers

- ARM은 37개의 32-bit register를 가지고 있다.
  - General-purpose register: 31개
  - Program status register: 6개

참고: ARM은 processor의 수행 mode에 따라 사용되는 register가 결정된다.


# Register Organization

General-purpose registers

Modes						
		Privileged modes				
		Exception modes				
User	System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	PC	PC	PC	PC	PC	PC

Program status registers

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

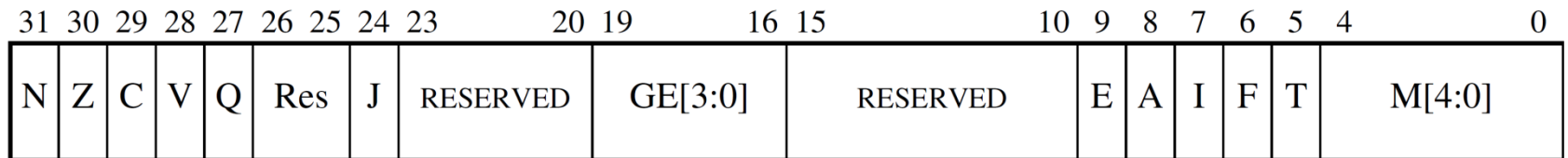
 indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode

# General-Purpose Registers

- General purpose register: R0 - R15.
  - Unbanked registers: R0 - R7 (모든 processor mode가 같은 register 사용.)
  - Banked registers: R8 - R12 (FIQ mode의 경우 다른 register 사용.)
  - Banked registers: R13 - R14 (모든 exception mode가 다른 register 사용.)
  - Unbanked register R15 (모든 processor mode가 같은 register 사용.)
- General purpose register 중 특별한 의미를 가지는 register.
  - R13: SP(Stack Pointer)
  - R14: LR(Link Register)
  - R15: PC(Program Counter)
- Program counter도 general-purpose register이기 때문에 read하거나 write할 수 있다.
  - PC를 read하면 ARM state 경우 (현재 수행 중인 instruction의 address + 8) 이 되고 Thumb state 경우 (현재 instruction의 address + 4)가 된다.
  - 어떤 값을 PC에 write하면 write한 값을 다음 address로 생각하고 jump가 일어난다.

# Program Status Registers - 1/2

- CPSR(Current Program Status Register)는 모든 processor mode가 같은 register를 사용한다. (1개)
- 각 exception mode마다 exception 발생 시 CPSR을 저장하는 SPSR(Saved Program Status Register)가 하나씩 있다. (5개)



Program status register 모습



# Program Status Registers - 2/2

- Condition code flags: N, Z, C, V
- DSP overflow/saturation flag: Q
- SIMD GE(greater than or equal) bits
- Endian bit for load/save: E
- Disable imprecise data abort: A
- Interrupt disable flags: I, F
- ARM/Thumb/Jazelle state: T, J
- Processor mode bits: M[4:0]

# Exceptions

- Exception이 발생하면 processor가 주어진 exception mode로 바뀌고 exception vector address로 프로그램의 수행을 옮긴다.

Exception type	Mode	Normal address	High vector address
Reset	Supervisor	0x00000000	0xFFFF0000
Undefined instructions	Undefined	0x00000004	0xFFFF0004
Software interrupt (SWI)	Supervisor	0x00000008	0xFFFF0008
Prefetch Abort (instruction fetch memory abort)	Abort	0x0000000C	0xFFFF000C
Data Abort (data access memory abort)	Abort	0x00000010	0xFFFF0010
IRQ (interrupt)	IRQ	0x00000018	0xFFFF0018
FIQ (fast interrupt)	FIQ	0x0000001C	0xFFFF001C

Exception types, modes 및 vector addresses

# Exception Processing의 일반 절차

- Exception 발생 시 (hardware가 수행)

```
R14 <exception_mode> = return link
SPSR <exception_mode> = CPSR
CPSR[4:0] = exception mode number
CPSR[5] = 0 /* Execute in ARM state */
if <exception_mode> == Reset or FIQ then
    CPSR[6] = 1 /* Disable fast interrupts */
    CPSR[7] = 1 /* Disable normal interrupts */
    if <exception_mode> != UNDEF or SWI then
        CPSR[8] = 1 /* Disable imprecise aborts (v6 only) */
    CPSR[9] = CP15_reg1_EEbit /* Endianness on exception entry */
PC = exception vector address
```

- Exception 처리 후 (software가 atomic하게 수행)

```
CPSR = SPSR <exception_mode>
PC = R14
```

## Reset Exception - 1/2

- Reset input이 processor에게 assert되면 ARM processor는 현재 instruction의 수행을 중단한다.
- Reset input이 de-assert되면 reset exception이 발생한다.
- Reset exception의 return에 대한 정의는 없다.

# Reset Exception - 2/2

- Exception 발생 시

```
R14_svc = UNPREDICTABLE value
SPSR_svc = UNPREDICTABLE value
CPSR[4:0] = 0b10011          /* Enter Supervisor mode */
CPSR[5] = 0                  /* Execute in ARM state */
CPSR[6] = 1                  /* Disable fast interrupts */
CPSR[7] = 1                  /* Disable normal interrupts */
CPSR[8] = 1                  /* Disable Imprecise Aborts (v6 only) */
CPSR[9] = CP15_reg1_EEbit    /* Endianness on exception entry */
if high vectors configured then
    PC = 0xFFFFF0000
else
    PC = 0x000000000
```

# Undefined Instruction Exception - 1/2

- 만약 현재 instruction이 coprocessor instruction이라면 ARM은 외부 coprocessor가 instruction을 수행할 수 있다는 acknowledge가 올 때까지 기다린다. 만약 coprocessor의 acknowledge 응답이 없으면 undefined instruction exception이 발생한다.
- Undefined instruction exception은 coprocessor instruction을 software적으로 emulation할 때 사용할 수 있으며, general instruction set의 확장에도 사용될 수 있다.
- Undefined instruction을 emulation한 후 return하기 위해서는 "MOVS PC, R14"를 수행하는데 이는 R14\_und를 PC에 복사하고 SPSR\_und를 CPSR에 복사한다.
- 만약 coprocessor가 2개 이상 동작하고 coprocessor 하나가 현재 instruction에 대하여 exception을 발생시키면서 다른 coprocessor의 acknowledge를 막는 경우에는 exception 처리 프로그램은 exception의 발생을 clear 시킨 후 "SUBS PC, R14, #4"를 수행하여 return함으로써 둘째 coprocessor의 응답을 유도할 수 있다. 이 instruction은 (R14\_und - 4)를 PC에 복사하고 SPSR\_und를 CPSR에 복사한다

# Undefined Instruction Exception - 2/2

- Exception 발생 시

```
R14_und = address of next instruction after the Undefined instruction
SPSR_und = CPSR
CPSR[4:0] = 0b11011          /* Enter Undefined Instruction mode */
CPSR[5] = 0                  /* Execute in ARM state */
/* CPSR[6] is unchanged */
CPSR[7] = 1                  /* Disable normal interrupts */
/* CPSR[8] is unchanged */
CPSR[9] = CP15_reg1_EEbit    /* Endianness on exception entry */
if high vectors configured then
    PC = 0xFFFF0004
else
    PC = 0x00000004
```

- Exception 처리 후

```
MOVS PC, R14
```

# Software Interrupt Exception - 1/2

- Software interrupt exception은 ARM instruction SWI 수행 시 발생한다.
- 이 exception이 발생하면 processor는 supervisor mode에 들어가게 된다.
- 이 exception은 응용 프로그램이 OS가 제공하는 서비스 함수를 호출(=system function call)하여 supervisor mode에 들어가게 하기 위하여 사용된다.



# Software Interrupt Exception - 2/2

- Exception 발생 시

```
R14_svc = address of next instruction after the SWI instruction
SPSR_svc = CPSR
CPSR[4:0] = 0b10011          /* Enter Supervisor mode */
CPSR[5] = 0                  /* Execute in ARM state */
/* CPSR[6] is unchanged */
CPSR[7] = 1                  /* Disable normal interrupts */
/* CPSR[8] is unchanged */
CPSR[9] = CP15_reg1_EEbit    /* Endianness on exception entry */
if high vectors configured then
    PC = 0xFFFF0008
else
    PC = 0x00000008
```

- Exception 처리 후

```
MOVS PC, R14
```

## Prefetch Abort (Instruction Fetch Memory Abort) - 1/2

- Instruction fetch 시 address에 문제가 있으면 memory system(예: Memory Protection Unit)은 processor에게 abort signal을 보내고 의미 없는 invalid instruction을 넘겨준다.
- Processor가 이 invalid instruction을 수행하려고 하면 prefetch abort exception이 발생한다. (참고: pipeline에서 branch instruction이 먼저 수행되면 뒤에 있는 invalid instruction이 수행되지 않을 수도 있으며 이 경우 prefetch abort exception은 발생하지 않는다.)
- ARMv5 혹은 그 이후 version의 경우 instruction BKPT를 수행하면 이 exception이 발생한다.
- Abort된 instruction을 수정한 후 return하기 위해서는 "SUBS PC, R14, #4"를 수행하는데 이는 (R14\_abt - 4)를 PC에 복사하고 SPSR\_abt를 CPSR에 복사한다.

## Prefetch Abort (Instruction Fetch Memory Abort) - 2/2

- Exception 발생 시

```
R14_abt = address of the aborted instruction + 4
SPSR_abt = CPSR
CPSR[4:0] = 0b10111          /* Enter Abort mode */
CPSR[5] = 0                  /* Execute in ARM state */
/* CPSR[6] is unchanged */
CPSR[7] = 1                  /* Disable normal interrupts */
CPSR[8] = 1                  /* Disable Imprecise Data Aborts (v6 only) */
CPSR[9] = CP15_reg1_EEbit    /* Endianness on exception entry */
if high vectors configured then
    PC = 0xFFFF000C
else
    PC = 0x0000000C
```

- Exception 처리 후 (abort된 이유를 수정한 경우)

```
SUBS PC, R14, #4
```

# Data Abort (Data Access Memory Abort) - 1/2

- Load/store instruction 수행 시 address에 문제가 있으면 memory system(예: Memory Protection Unit)이 data abort exception을 발생시킨다.
- 만약 store 시 data abort가 발생하면, memory system이 write를 허용하지 않을 경우, 그 주소의 값은 변경되지 않는다.
- 만약 load 시 data abort가 발생하면 address 계산에 사용된 여러 register의 값은 정해진 규칙에 따라 결정된다.

참고: Data abort는 synchronous한 precise data abort와 asynchronous한 imprecise data abort(예: Write Buffer의 내용이 memory에 write될 때 발생하는 abort.)로 나뉘어진다.  
(주: Imprecise data abort는 ARMv6부터 도입된 개념임.)

# Data Abort (Data Access Memory Abort) - 2/2

- Exception 발생 시

```
R14_abt = address of the aborted instruction + 8
SPSR_abt = CPSR
CPSR[4:0] = 0b10111          /* Enter Abort mode */
CPSR[5] = 0                  /* Execute in ARM state */
/* CPSR[6] is unchanged */
CPSR[7] = 1                  /* Disable normal interrupts */
CPSR[8] = 1                  /* Disable Imprecise Data Aborts (v6 only) */
CPSR[9] = CP15_reg1_EEbit    /* Endianness on exception entry */
if high vectors configured then
    PC = 0xFFFF0010
else
    PC = 0x00000010
```

- Exception 처리 후 (abort된 이유를 수정한 경우)

```
SUBS PC, R14, #8
```

- Exception 처리 후 (abort된 instruction을 다시 수행하지 않을 경우)

```
SUBS PC, R14, #4
```

# Interrupt Request (IRQ) Exception - 1/2

- IRQ exception은 processor의 IRQ input을 assert하면 발생한다.
- 이 exception은 FIQ보다 우선 순위가 낮다.
- FIQ exception 처리 시 IRQ exception은 disable 된다.
- CPSR의 I bit이 1이면 interrupt는 disable된다.
- CPSR의 I bit이 0이면 processor는 각 instruction 수행 시 마다 IRQ 발생을 점검한다.
- CPSR의 I bit은 processor의 privileged mode에서 수정 가능하다.

# Interrupt Request (IRQ) Exception - 2/2

- Exception 발생 시

```
R14_irq = address of next instruction to be executed + 4
SPSR_irq = CPSR
CPSR[4:0] = 0b10010          /* Enter IRQ mode */
CPSR[5] = 0                   /* Execute in ARM state */
/* CPSR[6] is unchanged */
CPSR[7] = 1                   /* Disable normal interrupts */
CPSR[8] = 1                   /* Disable Imprecise Data Aborts (v6 only) */
if high vectors configured then
    PC = 0xFFFF0018
else
    PC = 0x00000018
```

- Exception 처리 후

```
SUBS PC, R14, #4
```

# Fast Interrupt Request (FIQ) Exception - 1/2

- FIQ exception은 processor의 FIQ input을 assert하면 발생한다.
- FIQ는 고속 data transfer 혹은 channel process를 위하여 설계되었다.
- FIQ mode에서는 private register가 제공되기 때문에 context-switching overhead를 최소화할 수 있다.
- CPSR의 F bit이 1이면 FIQ는 disable된다.
- CPSR의 F bit이 0이면 processor는 각 instruction 수행 시마다 FIQ 발생을 점검한다.
- CPSR의 F bit은 processor의 privileged mode에서 수정 가능하다.



# Fast Interrupt Request (FIQ) Exception - 2/2

- Exception 발생 시

```
R14_fiq = address of next instruction to be executed + 4
SPSR_fiq = CPSR
CPSR[4:0] = 0b10001          /* Enter FIQ mode */
CPSR[5] = 0                  /* Execute in ARM state */
CPSR[6] = 1                  /* Disable fast interrupts */
CPSR[7] = 1                  /* Disable normal interrupts */
CPSR[8] = 1                  /* Disable Imprecise Data Aborts (v6 only) */
CPSR[9] = CP15_reg1_EEbit /* Endianness on exception entry */
if high vectors configured then
    PC = 0xFFFFF001C
else
    PC = 0x00000001C
```

- Exception 처리 후

```
SUBS PC, R14, #4
```

# Exception Priorities

- 동시에 여러 exception이 발생할 경우 priority가 높은 exception이 먼저 처리된다. (Undefined instruction exception과 SWI exception은 같은 priority를 가지나 동시에 발생하지 않게 구현되어 있음.)

Priority		Exception
Highest	1	Reset
	2	Data Abort
	3	FIQ
	4	IRQ
	5	Prefetch Abort
Lowest	6	Undefined instruction SWI

# High Vectors

- ARM processor가 normal vector address인 0x00000000 - 0x0000001C를 사용하지 않고 high vector address인 0xFFFF0000 - 0xFFFF001C를 사용하게 구현될 수 있다.
- 이 경우 high vector를 사용할 것인지 normal vector를 사용할 것인지를 설정하는 hardware configuration이 있어야 한다. (참고: 이를 설정하기 위한 ARM instruction은 없음.)

# New Instructions for Exception Handling

- ARMv6부터 제공되는 instruction들.
  - SRS(Store Return State): R14\_<current\_mode>와 SPSR\_<current\_mode>를 주어진 memory address에 저장한다. (주: Exception handler 시작 부분에서 사용.)
  - RFE(Return From Exception): 주어진 memory address에서 R14 및 CPSR를 load한다. (주: Exception handler 끝 부분에서 사용.)
  - CPS(Change Processor State): CPSR의 interrupt masks 및 mode bits를 변경한다.

# Endian Support

- Little endian: data의 LSB 부분을 lower address에 저장하는 방식
- Big endian: data의 MSB 부분을 lower address에 저장하는 방식.
- ARM의 endianness는 implementation defined이다.
- ARMv6는 mixed endian이 가능하다.

**Table 2-5 Big-endian memory system**

31	24	23	16	15	8	7	0
Word at address A							
Halfword at address A				Halfword at address A+2			
Byte at address A		Byte at address A+1		Byte at address A+2		Byte at address A+3	

**Table 2-6 Little-endian memory system**

31	20	19	12	11	10	9	8	5	4	3	2	1	0
Word at address A													
Halfword at address A+2							Halfword at address A						
Byte at address A+3			Byte at address A+2			Byte at address A+1			Byte at address A				

# Address Space

- ARM은 address 0 to  $(2^{32}-1)$ 를 가지며 한 address에는 1 byte를 저장한다.
- 1개의 word를 address A에 저장하는 경우 A는 4의 배수이고 address A, A+1, A+2, 및 A+4를 사용한다.
- 1개의 halfword를 address A에 저장하는 경우 A는 2의 배수이고 address A 및 A+1를 사용한다.

# Memory and Memory-Mapped I/O

- ARM에서 I/O 수행의 표준 방법은 memory-mapped I/O 방식이다.
- Memory-mapped I/O는 I/O 기능을 수행하기 위하여 미리 지정된 memory location을 사용한다.
- Memory-mapped I/O를 위하여 지정된 location은 일반 memory와 다른 behavior를 가진다.
  - 같은 주소에 대한 연속적인 load가 다른 값을 load할 수 있다.
- 위와 같은 특성 때문에 이들 location에 대하여는 cache 및 write buffer가 사용되지 않는다.
- LDM(Load multiple) 및 STM(Store multiple)과 같은 instruction은 memory-mapped I/O location에는 적합하지 않을 수도 있다.

# Unaligned Accesses Support

- Unaligned instruction accesses
  - ARM 상태 수행 시 word align되지 않은 값이 PC에 저장되면 결과는 unpredictable이거나 address의 bits[1:0]이 무시된다.
  - Thumb 상태 수행 시 halfword align되지 않은 값이 PC에 저장되면 address의 bit[0]이 무시된다.
- Unaligned data accesses
  - Load/store 수행 시 unaligned address가 사용될 경우 아래 중 하나가 선택된다.
    - unpredictable
    - lower order bit들을 무시한다.
    - lower order bit들을 무시하고 이들 bit들을 data의 rotation에 사용한다. (LDR 혹은 SWP의 경우)



# Synchronization Primitives

- Shared memory synchronization 구현을 위하여 swap instruction SWP 및 SWPB가 제공된다.
  - SWP(Swap): Register 값과 memory 값을 swap.
  - SWPB:(Swap Byte)
- ARMv6에서는 LDREX 및 STREX가 사용된다.
  - LDREX(Load Exclusive): 주어진 address에 대하여 exclusive load를 제공한다.
  - STREX(Store Exclusive): 주어진 address에 대하여 exclusive store를 제공한다.

참고: STREX에 사용된 address는 가장 최근에 실행된 LDREX에 사용된 address와 같아야 함.

# The Jazelle Extensions

- ARMv5부터 Jazelle extension이 제공됨.
- ARM의 Jazelle state에서는 효율적인 Java bytecode 수행이 이루어진다.
- Jazelle state로 들어가는 instruction BXJ가 제공됨.

참고: 본 slide에서는 다루지 않음.

# Saturated Integer Arithmetic

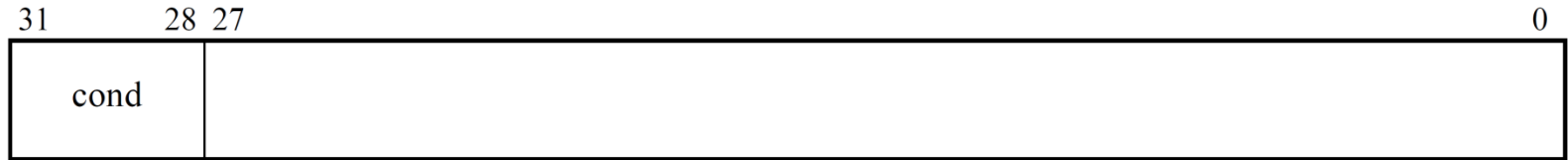
- Signed 32-bit인 경우 표현 가능한 정수는  $-2^{31} \sim +2^{31}-1$ 이다.
- DSP 응용 프로그램에서 ADD/SUB instruction의 계산 결과가 이 범위를 벗어나는 경우 문제가 발생하기 때문에 이의 해결을 위하여 아래와 같은 saturated signed arithmetic인 QADD/QSUB instruction이 제공된다.
  - 계산 결과가  $-2^{31}$ 보다 작을 경우  $-2^{31}$ 로 수정함.
  - 계산 결과가  $+2^{31}-1$ 보다 클 경우  $+2^{31}-1$ 로 수정함.

# 3. The ARM Instruction Set

- 1) The Condition Field
- 2) Branch Instructions
- 3) Data-Processing Instructions
- 4) Multiply Instructions
- 5) Parallel Addition and Subtraction Instructions
- 6) Extend Instructions
- 7) Miscellaneous Arithmetic Instructions
- 8) Status Register Access Instructions
- 9) Load and Store Instructions
- 10) Miscellaneous Load and Store Instructions
- 11) Load and Store Multiple Instructions
- 12) Semaphore Instructions
- 13) Exception-Generating Instructions
- 14) Coprocessor Instructions
- 15) Extending the Instruction set

# The Condition Field

- 대부분의 ARM instruction은 조건부 수행을 의미하는 mnemonic extension을 가질 수 있다.
- 예를 들어 "B"는 branch를 의미하는데 "BCC"는 CC(=carry clear) 조건 시 branch를 의미한다.
- 이를 구현하기 위하여 대부분의 ARM instruction은 4-bit의 condition field를 가지고 있다.



- Instruction 수행 시 이 condition field가 의미하는 조건과 CPSR의 condition flag이 의미하는 조건이 일치하지 않을 경우 instruction은 수행되지 않는다.

# Condition Fields의 의미

Opcode [31:28]	Mnemonic extension	Meaning	Condition flag state
0000	EQ	Equal	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 set and V set, or N clear and V clear (N == V)
1011	LT	Signed less than	N set and V clear, or N clear and V set (N != V)
1100	GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0, N == V)
1101	LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
1110	AL	Always (unconditional)	-
1111	-	See Condition code 0b1111	-

# Branch Instructions: B {L}

- Syntax

$B\{L\} \{ \langle \text{cond} \rangle \} \langle \text{target address} \rangle$

- Instruction B(Branch)는  $(PC + (\langle \text{target\_address} \rangle \ll 2))$ 로 branch한다. 여기서  $\langle \text{target\_address} \rangle$ 가 24 bit이기 때문에  $\pm 32\text{MB}$ 까지 branch할 수 있다.
- Instruction BL(Branch and Link)는 branch 다음 instruction의 address를 LR에 저장한 후 branch한다.

참고: BL은 subroutine call 시 사용한다. Subroutine return 시에는 LR을 PC에 복사한다.

# Branch Instructions: B{L}X

- Syntax

BX{<cond>} <Rm>

BLX <target address>

BLX{<cond>} <Rm>

- Instruction BX(Branch and Exchange)는 Rm의 bit[0]가 1이면 CPSR의 T bit을 set한 후 Rm 값을 target address로 branch한다.
- Instruction BLX(Branch with Link and Exchange)는 Rm의 bit[0]가 1이면 CPSR의 T bit을 set하고, branch 다음 instruction의 address를 LR에 저장한 후 (PC+(<target\_address><<2)) 혹은 Rm 값을 target address로 branch한다.

참고: 만약 Rm의 bit[0]가 1이면 위 instruction 수행에 의하여 processor는 ARM 상태에서 Thumb 상태로 변화됨.



# Branch Instructions: BXJ

- Syntax

BXJ {<cond>} <Rm>

- Instruction BXJ(Branch and change to Jazelle state)는 만약 수행 중인 processor가 Jazelle 기능이 있을 경우 Jazelle 상태로 들어간다. 그렇지 않을 경우 instruction BX와 같은 기능을 수행한다.

# Data-Processing Instructions - 1/2

- ARM은 16가지 종류의 data processing instruction을 가지고 있다.

Opcode	Mnemonic	Operation	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	$Rd := Rn + \text{shifter\_operand} + \text{Carry Flag}$
0110	SBC	Subtract with Carry	$Rd := Rn - \text{shifter\_operand} - \text{NOT(Carry Flag)}$
0111	RSC	Reverse Subtract with Carry	$Rd := \text{shifter\_operand} - Rn - \text{NOT(Carry Flag)}$
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	Compare Negated	Update flags after $Rn + \text{shifter\_operand}$
1100	ORR	Logical (inclusive) OR	$Rd := Rn \text{ OR shifter\_operand}$
1101	MOV	Move	$Rd := \text{shifter\_operand}$ (no first operand)
1110	BIC	Bit Clear	$Rd := Rn \text{ AND NOT}(\text{shifter\_operand})$
1111	MVN	Move Not	$Rd := \text{NOT shifter\_operand}$ (no first operand)

# Data-Processing Instructions - 2/2

- Syntax

ADD | SUB | RSB | ADC | SBC | RSC | AND | BIC | EOR | ORR {<cond>} {S}  
    <Rd>, <Rn>, <shifter\_operand>

CMP | CMN | TST | TEQ {<cond>} <Rd>, <shifter\_operand>

MOV | MVN {<cond>} {S} <Rd>, <shifter\_operand>

- Instruction mnemonic에 s가 추가되어 있으면 수행 결과에 따라 CPSR의 condition code flag를 변경한다.
- 사용되는 <shifter\_operand>는 상수 값인 #<immediate> 값이거나 register <Rm> 혹은 register <Rm>을 shift한 값을 의미한다. (주: 뒤 [4장](#)에서 설명.)

# Multiply Instructions - 1/2

- Syntax

MUL {<cond>} {S} <Rd>, <Rm>, <Rs>

MLA {<cond>} {S} <Rd>, <Rm>, <Rs>, <Rn>

- Multiply instruction은 source register <Rm>, <Rs>와 destination register <Rd>를 가진다.
- Multiply and accumulate instruction은 source register <Rm>, <Rs>, <Rn>와 destination register <Rd>를 가진다.

참고: ARM은 constant와 multiply하는 instruction은 없음.

# Multiply Instructions - 2/2

- ARM은 MUL, MLA 뿐만 아니라 아래와 같이 다양한 종류의 multiply가 있다.

종류	Mnemonics
Normal (32-bit x 32-bit, bottom 32-bit result)	MUL, MLA
Long (32-bit x 32-bit, 64-bit result)	SMULL, UMULL
Halfword (16-bit x 16-bit, 32-bit result)	SMULxy, SMLAxy, SMLALxy
Word and halfword (16-bit x 32-bit, top 32-bit result)	SMULWy, SMLAWy
Most significant word (32-bit x 32-bit, top 32-bit result)	SMMUL, SMMLA, SMMLS
Dual halfword (dual 16-bit x 16-bit, 32-bit result)	SMUAD, SMUSD, SMLAD, SMLSD, SMLALD, SMLS LD

# Parallel Addition and Subtraction Instructions

- ARMv6 부터는 SIMD parallel add/subtract instruction이 있다. (주: 32-bit을 2개의 16-bit 데이터 혹은 4개의 8-bit 데이터로 계산함.)
  - ADD8: 8-bit 씩 더하기.
  - SUB8: 8-bit 씩 빼기.
  - ADD16: 16-bit 씩 더하기.
  - SUB16: 16-bit 씩 빼기.
  - ADDSUBX: top 16-bit는 더하고, bottom 16-bit는 빼기.
  - SUBADDX: top 16-bit는 빼고, bottom 16-bit는 더하기.

참고: Prefix로 S(signed), Q(saturation), SH(signed/avoid overflow), U(unsigned) 추가 가능.

# Extend Instructions

- ARMv6 부터는 byte를 halfword나 word로, halfword를 word로 확장하는 명령이 있다.
  - XTB: bits[7:0]을 32-bit로 확장.
  - XTH: bits[15:0]을 32-bit로 확장.
  - XTB16: bits[23:16]과 bits[7:0]을 각각 16-bit로 확장.
  - XTAB: bits[7:0]을 32bit로 확장하고, register 값과 더함.
  - XTAH: bits[15:0]을 32-bit로 확장하고, register 값과 더함.
  - XTAB16: bits[23:16]과 bits[7:0]을 각각 16-bit로 확장하고, register의 값들과 더함.

참고: Prefix로 S(signed) 및 U(unsigned) 추가 가능.

# Miscellaneous Arithmetic Instructions

- CLZ(Count Leading Zeros): ARMv5 부터 있음.
  - Operand는 2개의 register.
  - Operand 2에 주어지는 register의 MSB에서 부터 연속된 0의 개수를 센다. 결과 값은 operand 1에 주어지는 register에 저장한다.
- USAD8(Unsigned Sum of Absolute Differences): ARMv6 부터 있음.
  - Operand는 3개의 register.
  - Operand 2 및 3에 주어지는 두 register를 4개의 byte로 구분하여 뺀 다음 그 결과를 더한다. 결과 값은 operand 1에 주어지는 register에 저장한다.
- USADA8(Unsigned Sum of Absolute Differences and Accumulate): ARMv6 부터 있음.
  - Operand는 4개의 register.
  - Operand 3 및 4에 주어지는 두 register를 각각 4개의 byte로 구분하여 뺀 다음 그 결과를 더한 후 다른 operand 2에 주어지는 register의 값과 더한다. 결과 값은 operand 1에 주어지는 register에 저장한다.



# Status Register Access Instructions - 1/2

- Syntax

MRS {<cond>} <Rd>, <cpsr | spsr>

MSR {<cond>} <cpsr | spsr>\_<fields>, #immediate

MSR {<cond>} <cpsr | spsr>\_<fields>, <Rm>

- MRS은 <cpsr | spsr>을 register <Rd>으로 복사.
- MSR은 #immediate 값 혹은 register <Rm>를 <cpsr | spsr>\_<fields>에 복사.

참고: MSR에서 <fields>에 가능한 문자로 c는 control field mask (0x000000ff), x는 extension field mask (0x0000ff00), s는 status field mask (0x00ff0000), f는 flags field mask (0xff000000)를 의미한다.

# Status Register Access Instructions - 2/2

- CPS(Change Processor State): ARMv6 부터.
  - Operand에 주어지는 값으로 CPSR의 processor mode 혹은 A, I, F flag을 변경.
  - 현재 processor의 mode가 privileged mode가 아닌 경우 NOP임.
- SETEND(Modifies the CPSR endianness)
  - Operand에 주어지는 값(예: BE=1 혹은 LE=0)으로 CPSR의 E bit(Endian)을 변경.

# Load and Store Instructions

- Syntax

LDR|STR {<cond>} {B} {T} <Rd>, <addressing\_mode\_2>

- Instruction LDR(Load Register)은 <addressing\_mode\_2>에 주어지는 메모리 주소에 저장된 값을 register <Rd>에 복사한다.
- Instruction STR(Store Register)은 register <Rd>의 값을 <addressing\_mode\_2>에 주어지는 메모리 주소에 복사한다.
- 사용되는 <addressing\_mode\_2>는 메모리 주소를 의미한다.  
(주: 뒤 4에서 설명.)

참고: B는 byte를 의미하고, T(=translation)는 privileged mode에서 수행할 때 user mode로 access한다는 의미이다.

# Miscellaneous Load and Store Instructions

- Syntax

LDR|STR {<cond>} D|H|SH|SB <Rd>, <addressing\_mode\_3>

- Instruction LDR(Load Register)은 <addressing\_mode\_3>에 주어지는 메모리 주소에 저장된 값을 register <Rd>에 복사한다.
- Instruction STR(Store Register)은 register <Rd>의 값을 <addressing\_mode\_3>에 주어지는 메모리 주소에 복사한다.
- 사용되는 <addressing\_mode\_3>는 메모리 주소를 의미한다.  
(주: 뒤 4에서 설명.)

참고: D는 double word, H는 halfword, SH는 signed halfword, SB는 signed byte를 의미한다.

# Load and Store Multiple Instructions

- Syntax

LDM{<cond>}<addressing\_mode\_4> <Rn>{!}, <registers>{^}

STM{<cond>}<addressing\_mode\_4> <Rn>{!}, <registers>{^}

where: <addressing\_mode\_4> = IA | IB | DA | DB | FD | FA | ED | EA

- Instruction LDM(Load Multiple)은 base address <Rn>에서 여러 word를 <registers>에 복사한다.
- Instruction STM(Store Multiple)은 <registers>에 지정된 여러 word들을 base address <Rn>에 복사한다.

참고: !는 Rn 값의 수정을 의미하고, ^는 <registers>에 PC가 포함되어 있는 LDM의 경우 SPSR을 CPSR에 복사한다는 의미이고, <registers>에 PC가 포함되어 있지 않는 LDM 및 모든 STM일 경우 user mode register를 사용한다는 의미이다.

참고: <addressing\_mode\_4>에서 IA는 increment after, IB는 increment before, DA는 decrement after, DB는 decrement before, FD는 full descending, FA는 full ascending, ED는 empty descending, EA는 empty ascending을 의미한다. (주: 뒤 4에서 설명.)

# Semaphore Instructions

- Syntax

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

- Instruction SWP(Swap)은 register <Rn>이 지정하는 메모리에 저장된 값을 tmp에 복사하고, register <Rm>의 값을 메모리로 복사한 후 tmp를 <Rd>에 복사한다.
- Instruction SWPB(Swap Byte)는 1 byte swap임.
- Register <Rd>와 <Rm>이 같은 register이면 단순 swap이 된다.

참고: ARMv6에서 deprecated됨.

# Exception-Generating Instructions

- Syntax

SWI {<cond>} <immed\_24>

BKPT <immediate>

- Instruction SWI(Software Interrupt)는 SWI exception을 발생시켜 processor를 supervisor mode로 바꾼다. <immed\_24>는 ARM hardware에 의하여 무시되지만 software에서 system call 정보를 표시하기 위하여 사용된다.
- Instruction BKPT(Breakpoint)는 prefetch abort exception을 발생시켜 processor를 abort mode로 바꾼다. Operand <immediate>는 ARM hardware에 의하여 무시되지만 software에서 breakpoint 정보를 표시하기 위해 사용된다.

참고: Instruction SWI는 OS system function call 구현에 사용되고, BKPT는 debugger 구현에 사용된다.

# Coprocessor Instructions - 1/2

- Syntax

MCR{<cond>} <coproc>, <opcode\_1>, <Rd>, <CRn>, <CRm>{, <opcode\_2>}

MCRR{<cond>} <coproc>, <opcode\_1>, <Rd>, <Rn>, <CRm>

MRC{<cond>} <coproc>, <opcode\_1>, <Rd>, <CRn>, <CRm>{, <opcode\_2>}

MRRC{<cond>} <coproc>, <opcode\_1>, <Rd>, <Rn>, <CRm>

- Register transfer

- MCR: Move to Coprocessor from ARM Register.
- MCRR: Move to Coprocessor from two ARM Registers.
- MRC: Move to ARM Register from Coprocessor.
- MRRC: Move to two ARM Registers from Coprocessor.

참고: <coproc>은 coprocessor 번호(p0-p15), <opcode\_1> 및 <opcode\_2>는 coprocessor-specific op-code, <Rd> 및 <Rn>은 ARM register, <CRn> 및 <CRm>은 coprocessor register임.



# Coprocessor Instructions - 2/2

- Data transfer(=Load and Store Coprocessor)
  - LDC: Load Coprocessor Register.
  - STC: Store Coprocessor Register.

참고: Load and store coprocessor instruction은 하나의 coprocessor register와 하나의 <addressing\_mode\_5>를 가진다. (주: 뒤 4에서 설명.)
- Data processing
  - CDP: Coprocessor Data Operations.

# Extending the Instruction Set

- ARM은 version의 증가에 따라 instruction이 확장되어 왔다.
- 아직 여러 분야의 instruction이 확장될 수 있게 space가 남아 있다.
  - Media instruction space
  - Multiply instruction extension space
  - Control and DSP instruction extension space
  - Load/store instruction extension space
  - Architecturally Undefined Instruction space
  - Coprocessor instruction extension space
  - Unconditional instruction extension space

## 4. ARM Addressing Modes

- 1) Addressing Mode 1 - Data-Processing Operands
- 2) Addressing Mode 2 - Load and Store
- 3) Addressing Mode 3 - Miscellaneous Loads and Stores
- 4) Addressing Mode 4 - Load and Store Multiple
- 5) Addressing Mode 5 - Load and Store Coprocessor

# Addressing Mode 1 - Data-processing Operands

- Data processing instruction은 아래와 같은 형식을 가진다.

`<opcode>{<cond>} {S} <Rd>, <Rn>, <shifter_operand>`

- Addressing mode 1은 data processing instruction에 사용되는 `<shifter_operand>`를 의미하며 아래와 같이 11가지 형식이 있다.

- |                                                    |                                                     |
|----------------------------------------------------|-----------------------------------------------------|
| 1) <code>#&lt;immediate&gt;</code>                 | <code>// immediate</code>                           |
| 2) <code>&lt;Rm&gt;</code>                         | <code>// register</code>                            |
| 3) <code>&lt;Rm&gt;, LSL #&lt;shift_imm&gt;</code> | <code>// logical shift left by immediate</code>     |
| 4) <code>&lt;Rm&gt;, LSL &lt;Rs&gt;</code>         | <code>// logical shift left by register</code>      |
| 5) <code>&lt;Rm&gt;, LSR #&lt;shift_imm&gt;</code> | <code>// logical shift right by immediate</code>    |
| 6) <code>&lt;Rm&gt;, LSR &lt;Rs&gt;</code>         | <code>// logical shift right by register</code>     |
| 7) <code>&lt;Rm&gt;, ASR #&lt;shift_imm&gt;</code> | <code>// arithmetic shift right by immediate</code> |
| 8) <code>&lt;Rm&gt;, ASR &lt;Rs&gt;</code>         | <code>// arithmetic shift right by register</code>  |
| 9) <code>&lt;Rm&gt;, ROR #&lt;shift_imm&gt;</code> | <code>// rotate right by immediate</code>           |
| 10) <code>&lt;Rm&gt;, ROR &lt;Rs&gt;</code>        | <code>// rotate right by register</code>            |
| 11) <code>&lt;Rm&gt;, RRX</code>                   | <code>// rotate right with extend</code>            |

# Addressing Mode 2 - Load and Store

- Load and store instruction은 아래와 같은 형식을 가진다.  
LDR|STR{<cond>} {B} {T} <Rd>, <addressing\_mode\_2>
- Addressing mode 2은 load and store instruction에 사용되는 <addressing\_mode\_2>를 의미하며 아래와 같이 9가지 형식이 있다. (주: {T}를 사용할 경우 7) - 9)만 가능.)
  - 1) [<Rn>, #+/-<offset\_12>] // immediate offset
  - 2) [<Rn>, +/-<Rm>] // register offset
  - 3) [<Rn>, +/-<Rm>, <shift> #<shift\_imm>] // scaled register offset
  - 4) [<Rn>, #+/-<offset\_12>]! // immediate pre-indexed
  - 5) [<Rn>, +/-<Rm>]! // register pre-indexed
  - 6) [<Rn>, +/-<Rm>, <shift> #<shift\_imm>]! // scaled register pre-indexed
  - 7) [<Rn>], #+/-<offset\_12> // immediate post-indexed
  - 8) [<Rn>], +/-<Rm> // register post-indexed
  - 9) [<Rn>], +/-<Rm>, <shift> #<shift\_imm> // scaled register post-indexed

참고: Pre-indexed는 <Rn>을 기반으로 계산된 address를 사용한 후 <Rn>을 그 address로 수정한다. Post-indexed는 <Rn>을 address로 사용한 후 <Rn>을 새로 계산된 값으로 수정한다. <shift>에는 LSL, LSR, ASR, ROR, RRX가 가능함.

## Addressing Mode 3 - Miscellaneous Loads and Stores

- Miscellaneous loads and stores instruction은 아래와 같은 형식을 가진다.

`LDR|STR{<cond>}D|H|SH|SB <Rd>, <addressing_mode_3>`

- Addressing mode 3은 miscellaneous load and store instruction에 사용되는 <addressing\_mode\_3>를 의미하며 아래와 같이 6가지 형식이 있다.

- |                                                                     |                           |
|---------------------------------------------------------------------|---------------------------|
| 1) [ <code>&lt;Rn&gt;</code> , <code>#+/-&lt;offset_8&gt;</code> ]  | // immediate offset       |
| 2) [ <code>&lt;Rn&gt;</code> , <code>+/-&lt;Rm&gt;</code> ]         | // register offset        |
| 3) [ <code>&lt;Rn&gt;</code> , <code>#+/-&lt;offset_8&gt;</code> ]! | // immediate pre-indexed  |
| 4) [ <code>&lt;Rn&gt;</code> , <code>+/-&lt;Rm&gt;</code> ]!        | // register pre-indexed   |
| 5) [ <code>&lt;Rn&gt;</code> ], <code>#+/-&lt;offset_8&gt;</code>   | // immediate post-indexed |
| 6) [ <code>&lt;Rn&gt;</code> ], <code>+/-&lt;Rm&gt;</code>          | // register post-indexed  |

# Addressing Mode 4 - Load and Store Multiple

- Load and store multiple instruction은 아래와 같은 형식을 가진다.  
`LDM|STM{<cond>}<addressing_mode_4> <Rn>{!}, <registers>{^}`
- Addressing mode 4는 load and store multiple instruction에 사용되는 `<addressing_mode_4>`를 의미하며 아래와 같이 4가지 형식이 있다.

- 1) IA: Increment After
- 2) IB: Increment Before
- 3) DA: Decrement After
- 4) DB: Decrement Before

start	end	Rn!
Rn	$Rn+4N-4$	$Rn+4N$
$Rn+4$	$Rn+4N$	$Rn+4N$
$Rn-4N+4$	Rn	$Rn-4N$
$Rn-4N$	$Rn-4$	$Rn-4N$

참고: LDM/STM이 stack의 pop/push으로 사용될 때 `<addressing_mode_4>`의 형식.

- |                               |                          |
|-------------------------------|--------------------------|
| 1) FA: Full Ascending Stack   | LDMFA=LDMDA, STMFA=STMIB |
| 2) FD: Full Descending Stack  | LDMFD=LDMIA, STMFD=STMDB |
| 3) EA: Empty Ascending Stack  | LDMEA=LDMDB, STMEA=STMIA |
| 4) ED: Empty Descending Stack | LDMED=LDMIB, STMED=STMDA |

참고: !는 Rn 값의 수정을 의미하고, ^는 `<registers>`에 PC가 포함되어 있는 LDM의 경우 SPSR을 CPSR에 복사한다는 의미이고, `<registers>`에 PC가 포함되어 있지 않는 LDM 및 모든 STM일 경우 user mode register를 사용한다는 의미이다.

# Addressing Mode 5 - Load and Store Coprocessor

- Load and store coprocessor instruction은 아래와 같은 형식을 가진다.

`LDC|STC {<cond>} {L} <coproc>, <CRd>, <addressing_mode_5>`

- Addressing mode 5는 load and store coprocessor instruction에 사용되는 <addressing\_mode\_5>를 의미하며 아래와 같이 4가지 형식이 있다.

- 1) `[<Rn>, #+/-<offset_8>*4]` // immediate offset
- 2) `[<Rn>, #+/-<offset_8>*4]!` // immediate pre-indexed
- 3) `[<Rn>], #+/-<offset_8>*4` // immediate post-indexed
- 4) `[<Rn>], <option>` // unindexed

참고: L은 long을 의미하고, <coproc>는 coprocessor 이름(예: p0, p1, ..., p15)을 의미하고, <CRd>는 coprocessor register 이름을 의미한다.



# 5. ARM Instruction Set Examples

- 1) Branch Instructions
- 2) Data-Processing Instructions
- 3) Multiply Instructions
- 4) Status Register Access Instructions
- 5) Load and Store Instructions
- 6) Load and Store Multiple Instructions
- 7) Semaphore Instructions
- 8) Exception-Generating Instructions
- 9) Coprocessor Instructions

# Branch Instructions

<u>B forward</u> ADD r1, r2, #4 ADD r0, r6, #2 ADD r3, r7, #4 forward SUB r1, r2, #4	ADD r4, r6, r7 <u>B backward</u>
backward ADD r1, r2, #4 SUB r1, r2, #4	<u>BL subroutine</u> CMP r1, #5 MOVEQ r1, #0 ... subroutine <subroutine code> MOV pc, lr

# Data-Processing Instructions - 1/2

<pre>PRE  r0 = 0x00000000       r1 = 0x00000002       r2 = 0x00000001       <u>SUB r0, r1, r2</u> POST r0 = 0x00000001</pre>	<pre>PRE  r0 = 0x00000000       r1 = 0x00000005       <u>ADD r0, r1, r1, LSL #1</u> POST r0 = 0x0000000f       r1 = 0x00000005</pre>
<pre>PRE  r0 = 0x00000000       r1 = 0x00000077       <u>RSB r0, r1, #0</u> POST r0 = -r1 = 0xffffffff89</pre>	<pre>PRE  r0 = 0x00000000       r1 = 0x02040608       r2 = 0x10305070       <u>ORR r0, r1, r2</u> ; r0 = r1   r2 POST r0 = 0x12345678</pre>
<pre>PRE  cpsr = nzcvtqiFt_USER       r1 = 0x00000001       <u>SUBS r1, r1, #1</u> POST cpsr = nZCvtqiFt_USER       r1 = 0x00000000</pre>	<pre>PRE  r1 = 0b1111       r2 = 0b0101       <u>BIC r0, r1, r2</u> ; r0 = r1 &amp; ~r2 POST r0 = 0b1010</pre>

# Data-Processing Instructions - 2/2

<pre> PRE  cpsr = nzcvtqiFt_USER       r0 = 4       r9 = 4       <u>CMP r0, r9</u> POST cpsr = nZcvtqiFt_USER         </pre>	<pre>       r7 = 8       <u>MOV r7, r5, LSL #2</u> POST  r5 = 5       r7 = 20         </pre>
<pre> PRE  r5 = 5       r7 = 8       <u>MOV r7, r5</u> POST  r5 = 5       r7 = 5         </pre>	<pre> PRE  cpsr = nzcvtqiFt_USER       r0 = 0x00000000       r1 = 0x80000004       <u>MOVS r0, r1, LSL #1</u> POST  cpsr = nZCvtqiFt_USER       r0 = 0x00000008       r1 = 0x80000004         </pre>
<pre> PRE  r5 = 5         </pre>	

# Multiply Instructions

PRE    r0 = 0x00000000

      r1 = 0x00000002

      r2 = 0x00000002

MUL r0, r1, r2

POST  r0 = 0x00000004

      r1 = 0x00000002

      r2 = 0x00000002

PRE    r0 = 0x00000000

      r1 = 0x00000000

      r2 = 0xf0000002

      r3 = 0x00000002

      ; [r1, r0] = r2\*r3

UMULL r0, r1, r2, r3

POST  ; = RdLo

      r0 = 0xe0000004

      ; = RdHi

      r1 = 0x00000001

# Status Register Access Instructions

PRE    `cpsr = nzcvqIFt_SVC`

`MRS r1, cpsr`

`BIC r1, r1, #0x80`

`MSR cpsr_c, r1`

POST `cpsr = nzcvqiFt_SVC`

참고: 0x80=b10000000이며  
CPSR의 bit 7은 I bit임. 즉 I  
bit을 0으로 clear한다.

참고: MSR에서 <fields>에 가

능한 문자로 c는 control  
field mask (0x000000ff), x  
는 extension field mask  
(0x0000ff00), s는 status  
field mask (0x00ff0000), f  
는 flags field mask  
(0xff000000)를 의미한다.

# Load and Store Instructions

```
PRE  r0 = 0x00000000
      r1 = 0x00009000
      mem32[0x00009000]
          = 0x01010101
      mem32[0x00009004]
          = 0x02020202
```

```
; Immediate offset
      LDR r0, [r1, #4]
POST r0 = 0x02020202
      r1 = 0x00009000
```

```
; immediate pre-indexed
      LDR r0, [r1, #4]!
POST r0 = 0x02020202
      r1 = 0x00009004
```

```
; immediate post-indexed
      LDR r0, [r1], #4
POST r0 = 0x01010101
      r1 = 0x00009004
```

# Load and Store Multiple Instructions - 1/2

```

PRE  mem32[0x80018] = 0x03
      mem32[0x80014] = 0x02
      mem32[0x80010] = 0x01
      r0 = 0x00080010
      r1 = 0x00000000
      r2 = 0x00000000
      r3 = 0x00000000
      LDMIA r0!, {r1-r3}

```

```

POST r0 = 0x0008001c
      r1 = 0x00000001
      r2 = 0x00000002
      r3 = 0x00000003

```

PRE	Address	Data	POST	Address	Data	
	0x80020	0x00000005		0x80020	0x00000005	r0 →
	0x8001c	0x00000004		0x8001c	0x00000004	
	0x80018	0x00000003		0x80018	0x00000003	r3
	0x80014	0x00000002		0x80014	0x00000002	r2
r0 →	0x80010	0x00000001		0x80010	0x00000001	r1
	0x8000c	0x00000000		0x8000c	0x00000000	

```

PRE  r0 = 0x00009000
      r1 = 0x00000009
      r2 = 0x00000008
      r3 = 0x00000007
      STMIB r0!, {r1-r3}
      MOV r1, #1
      MOV r2, #2
      MOV r3, #3

```

```

PRE  r0 = 0x0000900c
      r1 = 0x00000001
      r2 = 0x00000002
      r3 = 0x00000003
      LDMDA r0!, {r1-r3}

```

```

POST r0 = 0x00009000
      r1 = 0x00000009
      r2 = 0x00000008
      r3 = 0x00000007

```

PRE	Address	Data
	0x9014	?
	0x9010	?
	0x900c	?
	0x9008	?
	0x9004	?
r0 →	0x9000	?

POST	Address	Data	
	0x9014	?	
	0x9010	?	
r0 →	0x900c	0x00000007	r3
	0x9008	0x00000008	r2
	0x9004	0x00000009	r1
	0x9000	?	

PRE	Address	Data
	0x9014	?
	0x9010	?
	0x900c	0x00000007
	0x9008	0x00000008
	0x9004	0x00000009
r0 →	0x9000	?



# Load and Store Multiple Instructions - 2/2

PRE     $r1 = 0x00000002$   
         $r4 = 0x00000003$   
         $sp = 0x00080014$   
        STMFD  $sp!$ , { $r1, r4$ }

POST  $r1 = 0x00000002$   
        $r4 = 0x00000003$   
        $sp = 0x0008000c$

PRE	Address	Data	POST	Address	Data
$sp \rightarrow$	0x80018	0x00000001	$sp \rightarrow$	0x80018	0x00000001
	0x80014	0x00000002		0x80014	0x00000002
	0x80010	Empty		0x80010	0x00000003
	0x8000c	Empty		0x8000c	0x00000002

PRE     $r1 = 0x00000002$   
         $r4 = 0x00000003$   
         $sp = 0x00080010$   
        STMED  $sp!$ , { $r1, r4$ }

POST  $r1 = 0x00000002$   
        $r4 = 0x00000003$   
        $sp = 0x00080008$

PRE	Address	Data	POST	Address	Data
$sp \rightarrow$	0x80018	0x00000001	$sp \rightarrow$	0x80018	0x00000001
	0x80014	0x00000002		0x80014	0x00000002
	0x80010	Empty		0x80010	0x00000003
	0x8000c	Empty		0x8000c	0x00000002
	0x80008	Empty		0x80008	Empty

# Semaphore Instructions

```
PRE  mem32[0x9000]
      = 0x12345678
      r0 = 0x00000000
      r1 = 0x11112222
      r2 = 0x00009000
      SWP r0, r1, [r2]
POST mem32[0x9000]
      = 0x11112222
      r0 = 0x12345678
      r1 = 0x11112222
      r2 = 0x00009000
```

참고: SWP r0, r1, [r2]는  
tmp = mem32[r2]  
mem32[r2] = r1  
r0 = tmp.

# Exception-Generating Instructions

PRE    cpsr = nzcVqift\_USER

      pc = 0x00008000

      lr = 0x003ffffff

      r0 = 0x12

0x00008000

SWI 0x123456

POST   cpsr = nzcVqIf\_t\_SVC

      spsr\_svc

      = nzcVqift\_USER

      pc = 0x00000008

      lr\_svc = 0x00008004

r0 = 0x12

# Coprocessor Instructions

- 아래 instruction은 CP15의 register c0를 ARM register r10으로 복사한다.

MRC p15, 0, r10, c0, c0, 0

참고:

- ① Coprocessor 15는 system control coprocessor라고 하는데 system identification, cache control, write buffer control, memory management 등의 기능을 수행한다.
- ② Coprocessor 15의 primary

register X 및 secondary register Y를 간단하게 "CP15:cX:cY"로 표시한다. 만약 opcode1 혹은 opcode2가 0이 아닌 o1 및 o2로 주어진다면 "CP15:o1:cX:cY:o2"로 표시한다.

- ③ 앞의 보기에서 CP15:c0:c0는 processor identification 정보를 가지고 있다.

## 참고 문헌

1. ARM Limited, ARM<sup>®</sup> Architecture Reference Manual, ARM DDI 0100I, July 2005.
2. ARM Limited, ARM<sup>®</sup> Architecture Reference Manual ARM<sup>®</sup>v7-A and ARM<sup>®</sup>v7-R Edition, ARM DDI 0406B, July 2009.