4. ARM 명령어 소개

ARM의 특징

- 32 bit RISC 프로세서
  - 32 bit 의 Data bus/Address bus
  - Big/Little Endian 지원
- High Performance RISC
  - A load/store architecture
  - Data processing instructions act only on registers
  - Most instructions execute in a single cycle.
- Excellent high level language support
- Simple & Powerful Instruction set
  - Every instruction can be conditionally executed.
  - Shifted Register Operand
- Instruction set extension via coprocessors

프로세서 동작 모드(1)

Processor mode			Description
1	User	USR	normal program execution mode
2	FIQ	FIQ	support a high-speed data transfer or channel process
3	IRQ	IRQ	general-purpose interrupt handling
4	Supervisor	SVC	a protected mode for the operating system
5	Abort	ABT	implement virtual memory and/ or memory protection
6	Undefined	UND	support software emulation of hardware coprocessors
7	System	SYS	turn *privileged* operating system tasks (Version 4 이상 only)

- 모드 변경은 소프트웨어 제어 혹은 외부 인터럽트나 예외 처리에 의해 발생
- 대부분의 응용 프로그램은 User mode에서 실행
- 다른 특권 모드는 인터럽트 혹은 예외상황 처리 때와 제한된 자원에 접근하는 경우에 진입

프로세서 동작 모드(2)

특권 모드에서는 상호 자유로운 이동이 가능하나 사용자 모드에서는 이동이 안됨

레지스터 구조

- 프로세서 모드에 따라 몇 개의 레지스터 bank로 나누어짐
- 31개의 32비트 범용 레지스터
  - 16개는 모든 모드에서 사용가능
  - 15개는 모드에 따라서 레지스터 맵핑되어 사용
- 상태 레지스터
  - CPSR (current program status register) – 모든모드에서 접근
  - SPSR (saved program status register) – 특권모드에서 접근
- 범용 레지스터의 용도
  - r0-r12: 범용 레지스터
  - r13: stack pointer(SP)
  - r14: link register(LR)
  - r15: program counter(PC)

레지스터 맵핑

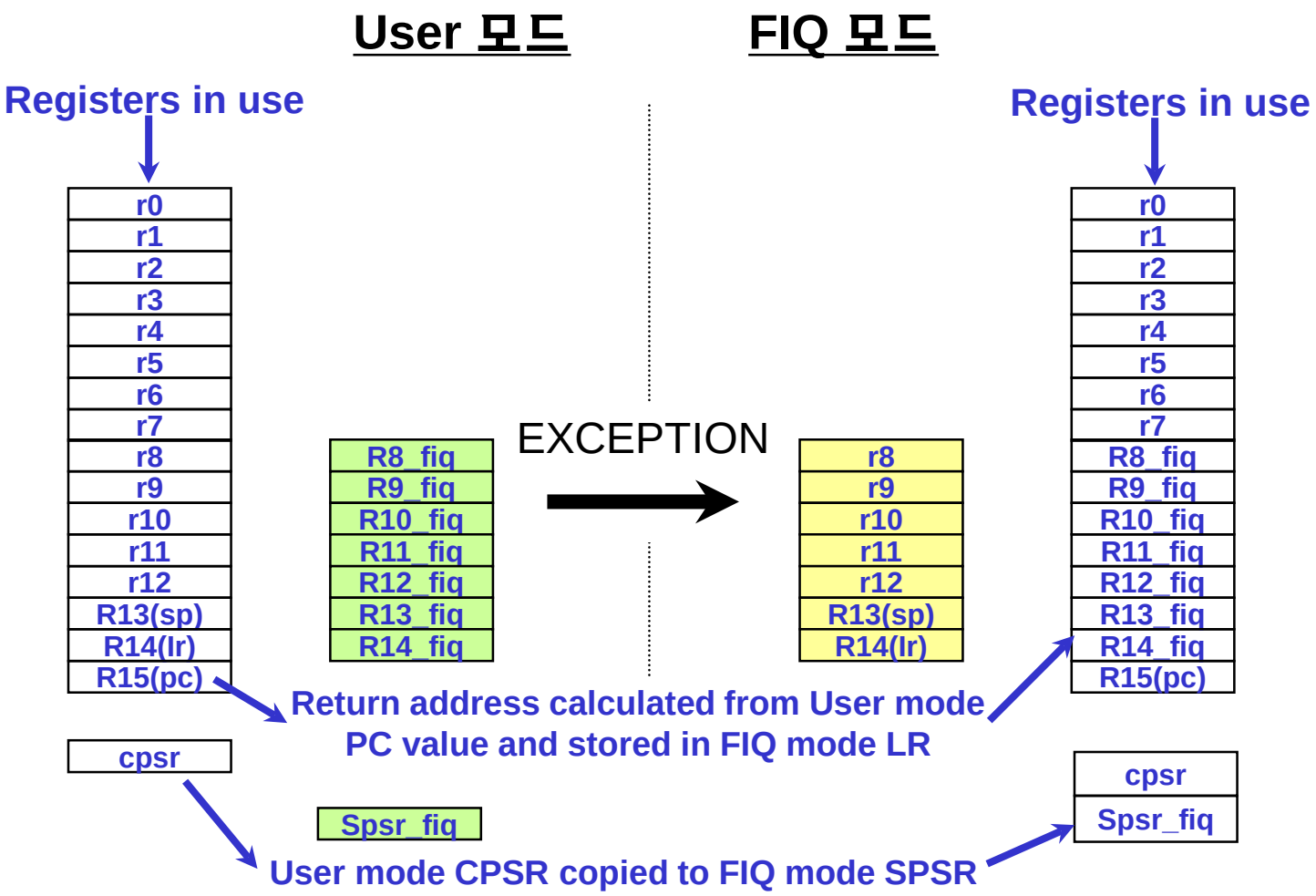
USR/SYS	SVC	ABT	UND	IRQ	FIQ
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8_FIQ
R9	R9	R9	R9	R9	R9 _FIQ
R10	R10	R10	R10	R10	R10 _FIQ
R11	R11	R11	R11	R11	R11 _FIQ
R12	R12	R12	R12	R12	R12 _FIQ
R13	R13_SVC	R13_ABORT	R13_UNDEF	R13_IRQ	R13 _FIQ
R14	R14_SVC	R14_ABORT	R14_UNDEF	R14_IRQ	R14 _FIQ
PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_SVC	SPSR_ABORT	SPSR_UNDEF	SPSR_IRQ	SPSR_FIQ

APCS 레지스터 사용 규칙

- APCS (ARM Procedure Call Standard)
  - restrictions on the use of registers
  - conventions for using the stack
  - passing/returning arguments between function calls
- 레지스터 사용 규칙

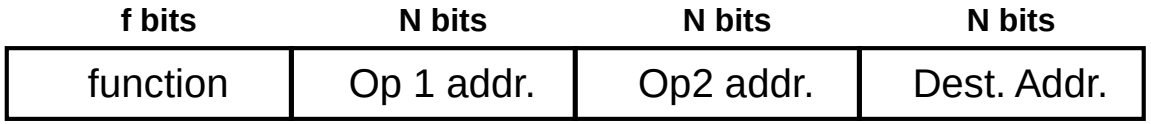
레지스터	APCS표기	용도
R0-R3	a1 - a4	argument / result / scratch register
R4-R9	v1 - v5	variable register
R9	sb / v6	stack base / variable register
R10	sl / v7	stack limit / variable register
R11	fp / v8	frame pointer / variable register
R12	ip	intra-procedure-call scratch register
R13	sp	stack pointer
R14	lr	link register (return address)
R15	pc	program counter

Register 사용 예: User to FIQ Mode

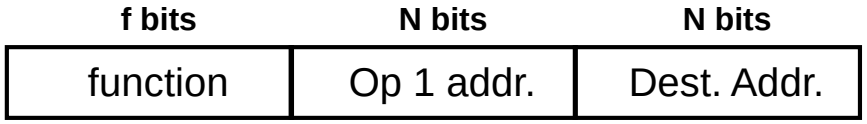


Instruction Format

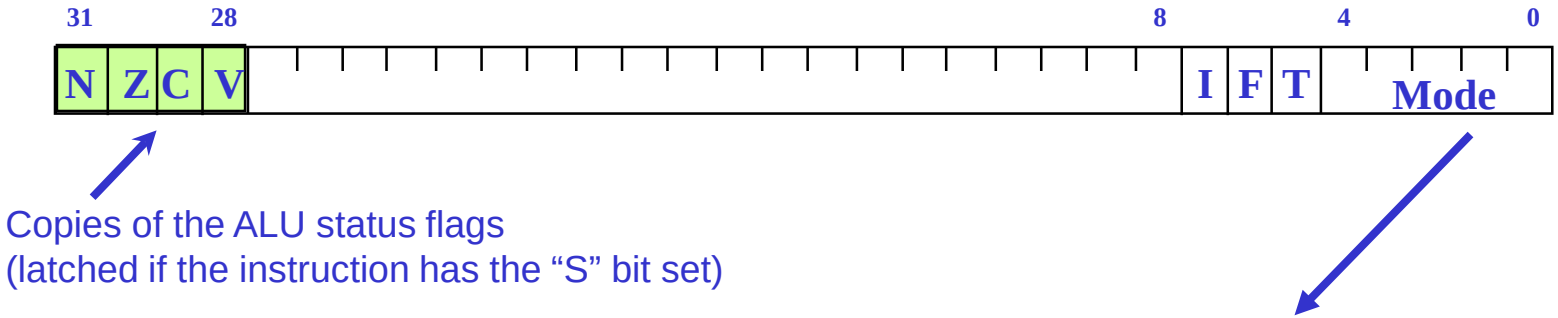
- Instruction Format
  - 3 address instruction format
    - used in ARM state



  - 2 address instruction format
    - used in ARM and THUMB state



Program Status Registers (CPSR & SPSRs)



- Condition Code Flags
  - N = Negative result from ALU flag.
  - Z = Zero result from ALU flag.
  - C = ALU operation Carried out
  - V = ALU operation oVerflowed
- Interrupt Disable bits.
  - I = 1, disables the IRQ.
  - F = 1, disables the FIQ.
- T Bit: Processor in ARM (0) or Thumb (1)
- Mode Bits: processor mode

M[4:0]	Mode
10000	User
10001	FIQ
10010	IRQ
10011	SVC
10111	Abort
11011	Undef
11111	System

조건 플래그(Condition Flags)

Flag	Logical Instruction	Arithmetic instruction
N	No meaning	Bit 31 of the result has been set. Indicates a negative number in signed operations
Z	Result is all zeroes	Result of operation was Zero
C	After shift operation 1 was left in carry flag	Result was greater than 32 bits
V	No meaning	Result was greater than 31 bits. Indicates a possible corruption of the sign bit in signed numbers

프로그램 카운터(Program Counter : R15)

- 프로세서가 ARM state에서 수행될 때..
  - All instructions are 32 bits in length
  - All instructions must be word aligned
  - Therefore the PC value is stored in bits [31:2] with bits [1:0] equal to zero (as instruction cannot be halfword or byte aligned).
- R14
  - is used as the subroutine link register (LR) and
  - stores the return address when **BL** (Branch with Link) operations are performed, calculated from the PC.
- Return from a linked branch

```
MOV r15, r14  또는
MOV pc, lr
```

별도의 return명령 없음

예외 상황 처리

- 예외 상황 발생 시
  - Copies CPSR into SPSR_<mode>
  - Sets appropriate CPSR bits
    - enter ARM state if necessary
    - Mode field bits
    - Interrupt disable flags if appropriate.
  - Maps in appropriate banked registers
  - Stores the “return address” in LR_<mode>
  - Sets PC to vector address

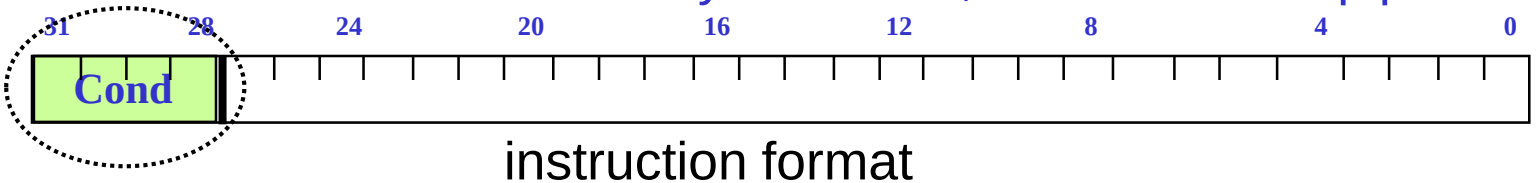
- 예외 상황으로 부터 복귀시:
  - Restore CPSR from SPSR_<mode>
  - Restore PC from LR_<mode>

Exceptions

Vector Address	Exception	Mode on Entry	Priorities
0x00000000	Reset	Supervisor	1
0x00000004	Undefined instruction	Undefined	6
0x00000008	Software interrupt	Supervisor	6
0x0000000C	Abort (prefetch)	Abort	5
0x00000010	Abort (data)	Abort	2
0x00000014	Reserved	Reserved	Reserved
0x00000018	IRQ	IRQ	4
0x0000001C	FIQ	FIQ	3

조건부 실행

- ARM에서 모든 명령을 조건에 따라 실행 여부 결정 가능
- Removes the need for many branches, which stall the pipeline



instruction format

Mnemonic Extension	Interpretation	Mnemonic Extension	Interpretation
EQ	Equal / equals zero	HI	Unsigned higher
NE	Not equal	LS	Unsigned lower or same
CS/HS	carry set/unsigned higher or same	GE	Signed greater than or equal
CC/LO	carry clear/unsigned lowerMinus/ negative	LT	Signed less than
MI		GT	Signed greater than
PL	Plus / positive or zero	LE	Signed less than or equal
VS	Overflow	AL	Always
VC	No overflow	NV	Never (do not use!)

조건 필드의 사용과 변경

- 조건에 따라 명령어를 실행하도록 하기위해서는 조건 접미사 사용

```
ADD r0,r1,r2          ; r0 = r1 + r2  (항상수행=ADDAL)
ADDEQ r0,r1,r2        ; If Z=1, then r0 = r1 + r2
```

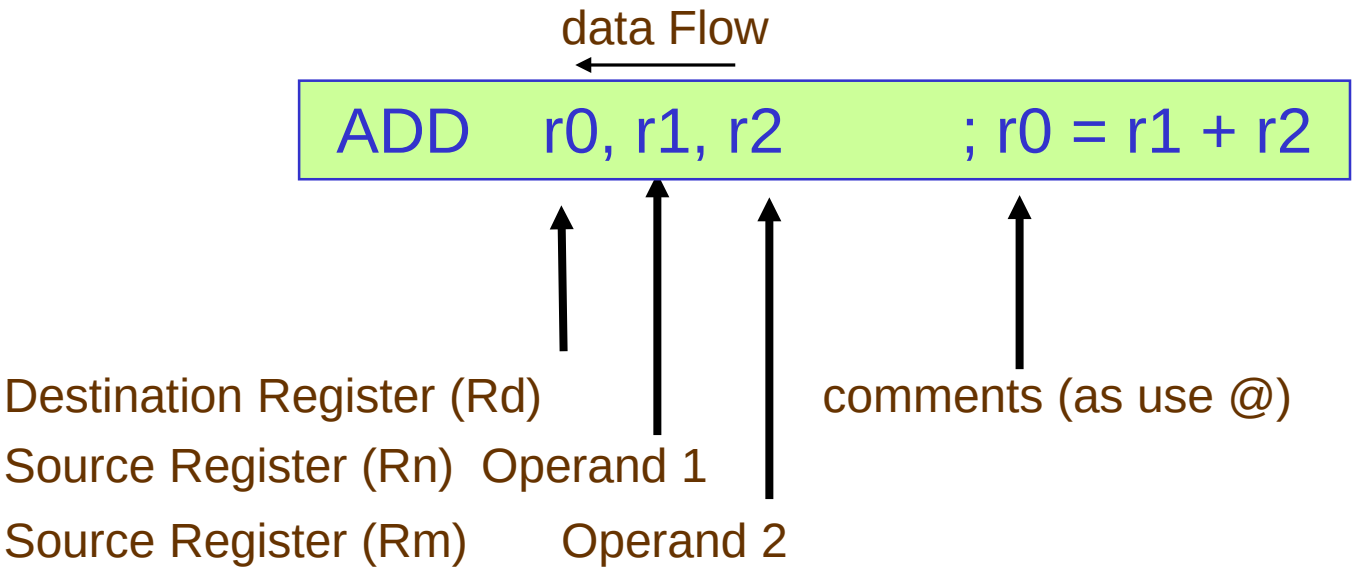
- 데이터 처리 명령어는 기본적으로 조건 플래그에 영향을 미치지 못함

- 조건 플래그가 변경이 되도록 하기위해서는, 명령에 “S”를 접미사로 명시하여, S bit가 설정되도록 해야 함.

```
ADDS r0,r1,r2         ; r0 = r1 + r2  , flag에 영향
ADD r0, r1, r2        ; r0 = r1 + r2 ,  flag에 영향 없음
```

ARM 명령어 기본 형식

- Standard operand and data flow format for ARM(Not Thumb)



- Operand 1 is always a register
- Operand 2 can be a register or an immediate vale

명령어 집합 요약(1)

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

명령어 집합 요약(2)

Mnemonic	Instruction	Action
LDC	**Load coprocessor from memory**	**Coprocessor load**
LDM	**Load multiple registers**	**Stack manipulation (Pop)**
LDR	**Load register from memory**	**Rd: = (address)**
MCR	**Move CPU register to coprocessor register**	**cRn: = rRn {<op>cRm}**
MLA	**Multiply Accumulate**	**Rd: = (Rm * Rs) + Rn**
MOV	**Move register or constant**	**Rd: = Op2**
MRC	**Move from coprocessor register to CPU register**	**Rn: = cRn {<op>cRm}**
MRS	**Move PSR status/flags to register**	**Rn: = PSR**

명령어 집합 요약(3)

Mnemonic	Instruction	Action
MSR	**Move register to PSR status/flags**	**PSR: = Rm**
MUL	**Multiply**	**Rd: = Rm * Rs**
MVN	**Move negative register**	**Rd: = 0xFFFFFFFF XOR Op2**
ORR	**OR**	**Rd: = Rn OR Op2**
RSB	**Reverse Subtract**	**Rd: = Op2 – Rn**
RSC	**Reverse Subtract with Carry**	**Rd: = Op2 – Rn – 1 + Carry**
SBC	**Subtract with Carry**	**Rd: = Rn – Op2 - 1 + Carry**
STC	**Store coprocessor register to memory**	**address: = CRn**
STM	**Store Multiple**	**Stack manipulation (Push)**

명령어 집합 요약(4)

Mnemonic	Instruction	Action
STR	**Store register to memory**	**<address>: = Rd**
SUB	**Subtract**	**Rd: = Rn – Op2**
SWI	**Software Interrupt**	**OS call**
SWP	**Swap register with Memory**	**Rd: = [Rn], [Rn] := Rm**
TEQ	**Test bitwise equality**	**CPSR flags: = Rn XOR Op2**
TST	**Test bits**	**CPSR flags: = Rn AND Op2**

데이터 처리 명령어

- 관련 명령어 종류
  - 산술 연산
  - 비교 연산 (no results - just set condition codes)
  - 논리 연산
  - 레지스터간의 데이터 이동 연산
- ARM is a **load / store architecture**
  - Instructions only work on registers and NOT on memory
- Perform a specific operation on one or two operands.
  - First operand always a register - Rn
  - Second operand sent to the ALU via **barrel shifter**.

산술연산

- 산술 연산 명령어 및 동작:
  - ADD operand1 + operand2
  - ADC operand1 + operand2 + carry
  - SUB operand1 - operand2
  - SBC operand1 - operand2 + carry -1 (carry=0이면 borrow=1)
  - RSB operand2 - operand1
  - RSC operand2 - operand1 + carry - 1
- 문법:
  - <Operation>**{<cond>}{S}** Rd, Rn, Operand2
- 예제
  - ADD r0, r1, r2
  - SUB**GT** r3, r3, #1
  - RSB**LES** r4, r5, #5

비교 연산

- 비교 연산의 결과는 조건 플래그를 변경하는 것
  - S bit를 별도로 set 할 필요가 없다.
- 명령어 및 동작:
  - CMP operand1 - operand2, but result not written
  - CMN operand1 + operand2, but result not written
  - TST operand1 AND operand2, but result not written
  - TEQ operand1 XOR operand2, but result not written
- 문법:
  - <Operation>{<cond>} Rn, Operand2
- 예제:
  - CMP r0, r1
  - TST**EQ** r2, #5

논리 연산

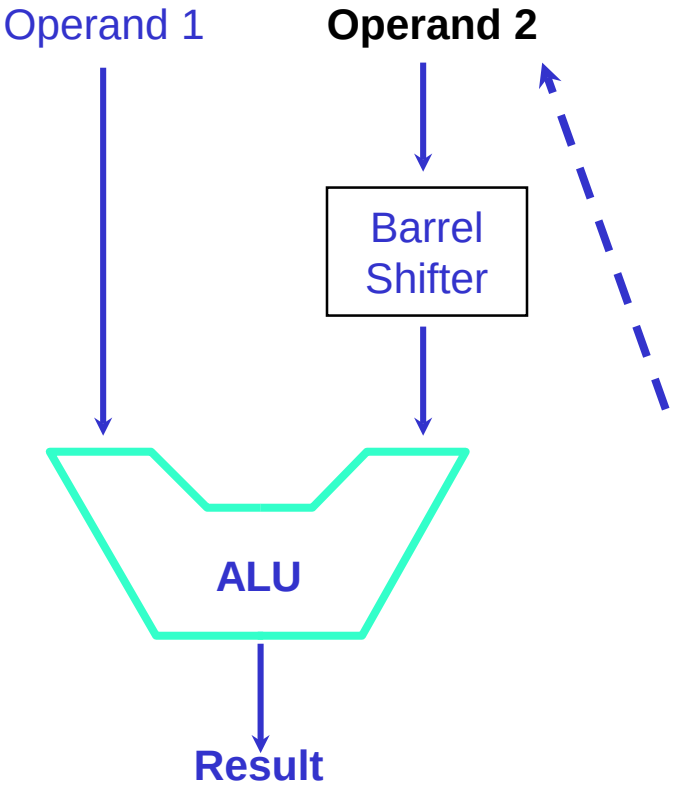
- 논리 연산 명령어:
  - AND operand1 AND operand2
  - EOR operand1 XOR operand2
  - ORR operand1 OR operand2
  - BIC operand1 AND NOT operand2 [i.e. bit clear]
- 문법:
  - <Operation>{<cond>}{S} Rd, Rn, Operand2
- 예제:
  - AND r0, r1, r2
  - BIC**EQ** r2, r3, #7
  - EOR**S** r1,r3,r0

데이터 이동 연산

- 관련 명령어:
  - MOV operand2
  - MVN NOT operand2
    - Note that these make no use of operand1.
- 문법:
  - <Operation> {<cond>} {S} Rd, Operand2
- 예제:
  - MOV r0, r1 ; r1 -> r0
  - MOVS r2, #10 ; 10 -> r2
  - MVNEQ r1,#0 ; if zero flag set then ~0 -> r1

배럴 쉬프터

- ARM은 다른 명령어의 일부로서 쉬프트 연산을 제공하는 배럴 쉬프터(barrel shifter)를 제공

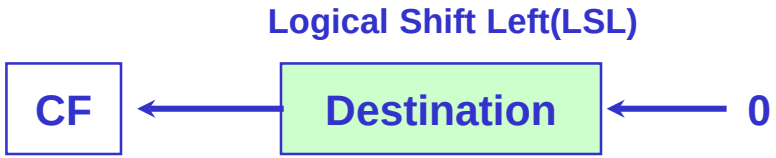


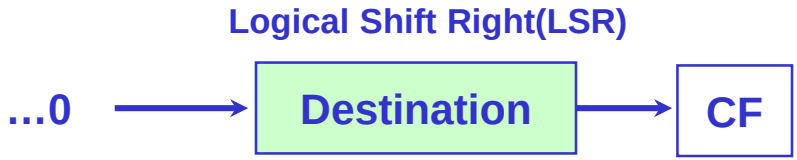
- Register, optionally with shift operation applied.
- Shift value can be either be:
  - 5 bit unsigned integer
  - specified in bottom byte of another register.

- Immediate value
  - 8 bit number
  - Can be rotated right through an even number of positions.
  - Assembler will calculate rotate for you from constant

논리/산술 SHIFT

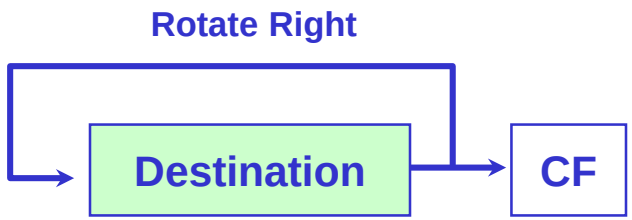
- Logical Shift Left
  - LSL #5 = multiply by 32
  - LSL = ASL
- Logical Shift Right
  - LSR #5 = divide by 32 (unsigned)
- Arithmetic Shift Right:
  - preserves the sign bit
  - ASR #5 = divide by 32 (signed)

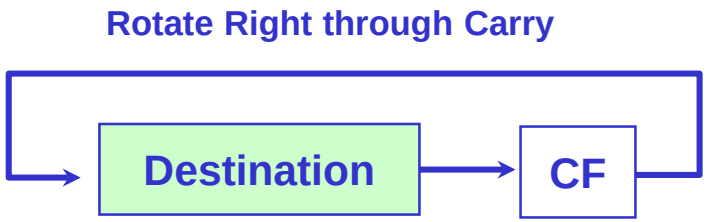




Rotations

- Rotate Right (ROR)
  - ROR #5
  - Note the last bit rotated is also used as the Carry Out.
- Rotate Right Extended (RRX)
  - uses C flag as a 33rd bit.
  - Rotates right by 1 bit (only)
  - Encoded as ROR #0.





Shifted Register Operand

- 레지스터가 SHIFT되는 양은 2 가지 방법으로 지정.
  - the immediate 5-bit field in the instruction
    - Shift is done for free - executes in single cycle.
    - ADD r5, r5, **r3 LSL #3**;
  - the bottom byte of a register (not PC)
    - This then takes extra cycle to execute
    - ARM doesn’t have enough read ports to read 3 registers at once.
    - ADD r5, r5, **r3 LSL r2**;
- Shift 가 명시되지 않으면 default shift 적용: LSL #0
  - i.e. barrel shifter has no effect on value in register.

Shifted Register Operand

- Using a multiplication instruction to multiply by a constant
  - load the constant into a register
  - Wait for a number of internal cycles for the multiplication to complete.
- A more optimum solution can be often found by using some combination of MOVs, ADDs, SUBs and RSBs with shifts.
  - Multiplications by a constant equal to a ((power of 2) ± 1) can be done in one cycle.
- (예제) r0 = r1 * 5 → r0 = r1 + (r1 * 4)
  - ADD r0, r1, r1, LSL #2
- (예제) r2 = r3 * 105 → r2 = r3 * 15 * 7 → r2 = r3 * (16 - 1) * (8 - 1)
  - RSB r2, r3, r3, LSL #4 ; r2 = r3 * 15
  - RSB r2, r2, r2, LSL #3 ; r2 = r2 * 7

곱셈 명령어

- ARM은 기본적으로 2 개의 곱셈 명령어 제공
  - Multiply
    - **MUL** {<cond>} {S} Rd, Rm, Rs ; Rd = Rm * Rs
  - Multiply Accumulate - does addition for free
    - **MLA** {<cond>} {S} Rd, Rm, Rs,Rn ; Rd = (Rm * Rs) + Rn
- 사용 상의 제한:
  - Rd 과 Rm는 동일한 레지스터이어서는 안됨
    - Can be avoid by swapping Rm and Rs around.
    - Cannot use PC. Will be picked up by the assembler if overlooked.
- Operands can be considered signed or unsigned
  - Up to user to interpret correctly.

Multiply-Long and Multiply-Accumulate Long

- 명령어
  - MULL RdHi,RdLo := Rm*Rs
  - MLAL RdHi,RdLo := (Rm*Rs) + RdHi,RdLo
- However the full 64 bit of the result now matter (lower precision multiply instructions simply throws top 32bits away)
  - Need to specify whether operands are signed or unsigned
- 관련 명령어 문법:
  - UMULL {<cond>} {S} RdLo,RdHi,Rm,Rs
  - UMLAL {<cond>} {S} RdLo,RdHi,Rm,Rs
  - SMULL {<cond>} {S} RdLo, RdHi, Rm, Rs
  - SMLAL {<cond>} {S} RdLo, RdHi, Rm, Rs
- Not generated by the compiler.
  - Warning : Unpredictable on non-M ARMs.
    (cf) M-variant ARM: long multiply지원

Load / Store 명령어

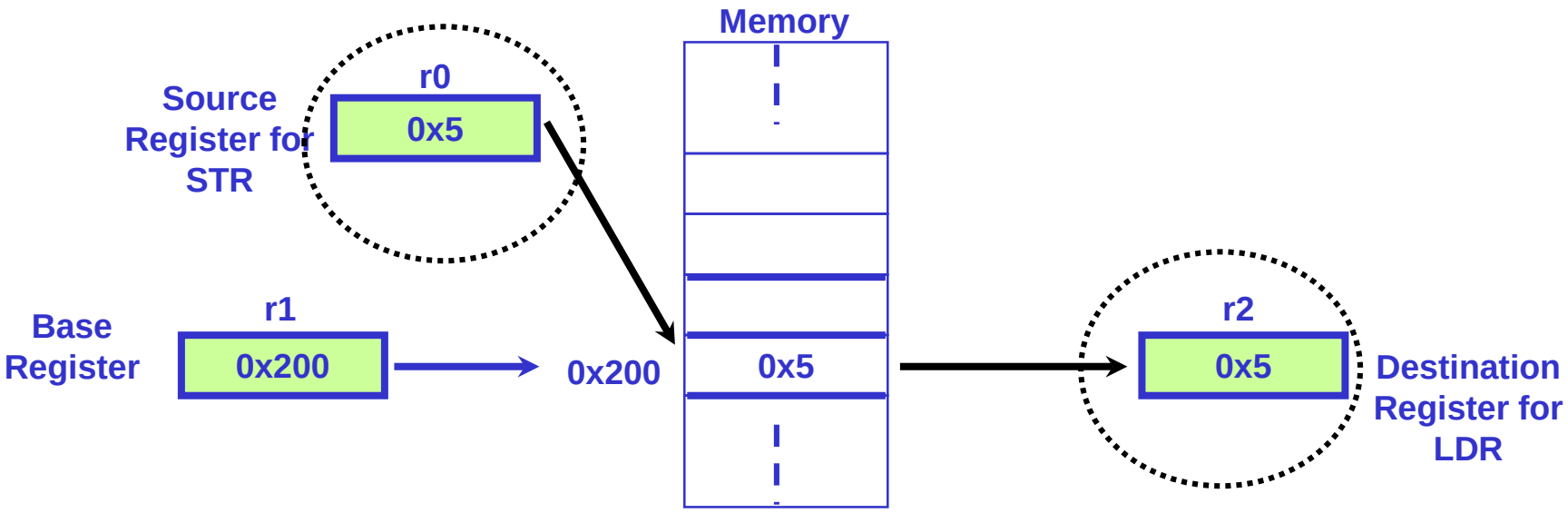
- **The ARM is a Load / Store Architecture:**
  - memory to memory 데이터 처리 명령을 지원하지 않음
  - 일단 사용하려는 데이터를 레지스터로 이동해야 함
- 비효율적으로 보이나 실제는 그렇지 않음:
  - Load data values from memory into registers.
  - Process data in registers using a number of data processing instructions which are not slowed down by memory access.
  - Store results from registers out to memory.
- ARM은 메모리와 load/store하는 세종류의 명령어 집합이 있음.
  - Single register data transfer (LDR / STR).
  - Block data transfer (LDM / STM).
  - Single Data Swap (SWP).

Single Register Data Transfer

- 기본적인 load/store 명령:
  - Load and Store Word or Byte
    - LDR / STR / LDRB / STRB
- ARM Architecture v4는 halfwords와 signed data 추가 지원
  - Load and Store Halfword
    - LDRH / STRH
  - Load Signed Byte or Halfword - load value and sign extend it to 32 bits.
    - LDRSB / LDRSH
- 모든 명령어는 STR / LDR 조건 접미사를 삽입하므로써 조건에 따라 실행하도록 할 수 있음.
  - e.g. LDR**EQ**B
- 문법:
  - <LDR|STR> {<cond>} {<size>} Rd, <address>

Load and Store Word or Byte: Base Register

- 접근될 메모리 위치는 base register에 유지함
  - STR r0, [r1] ; [r1] ← r0
  - LDR r2, [r1] ; r2 ← [r1]

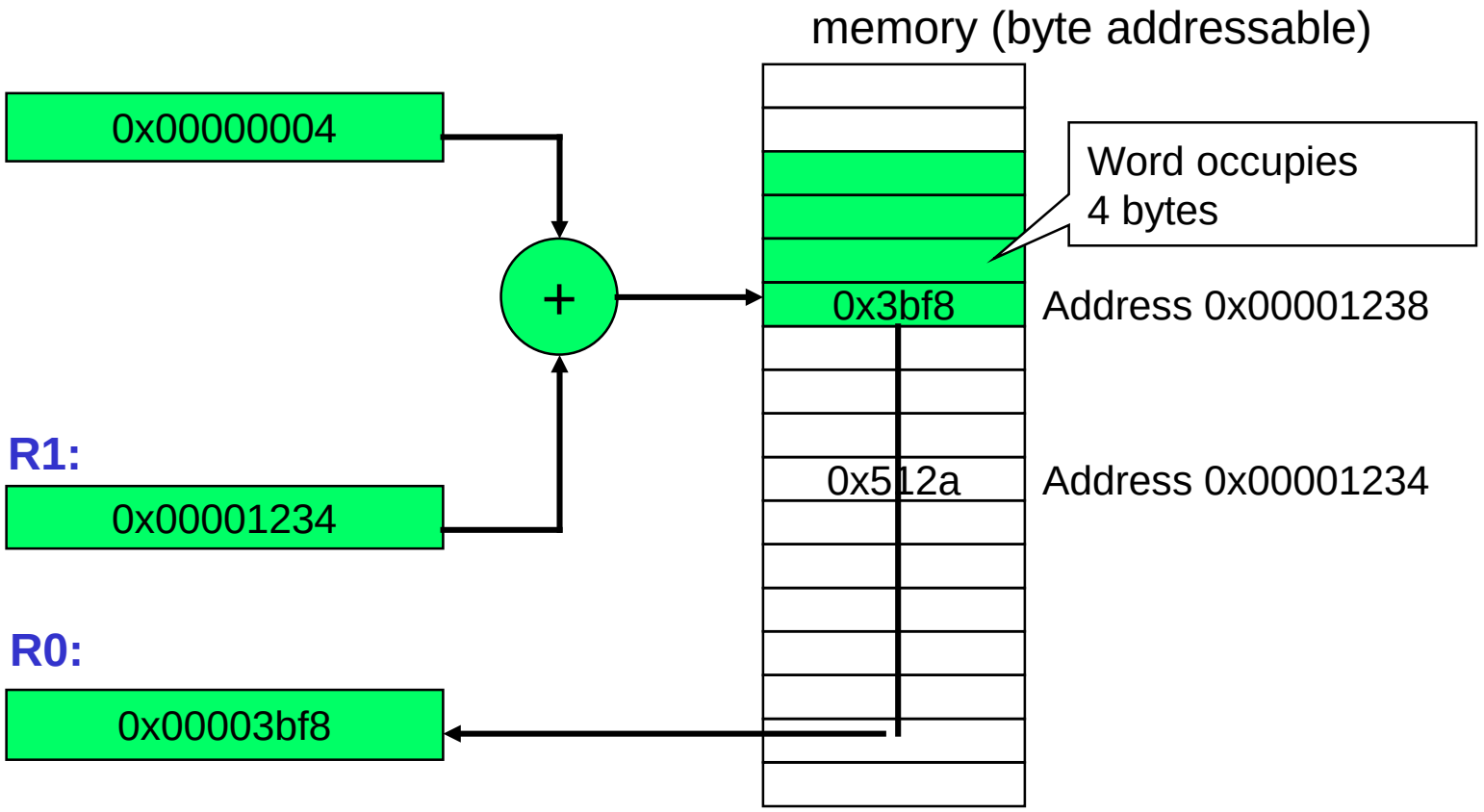


Load and Store Word or Byte: with Offsets

- base register pointer로부터 offset 만큼 떨어진 위치의 정보를 access하는 명령 구조
- Offset 정보는 :
  - An unsigned 12 bit **immediate value** (i.e. 0 - 4095 bytes).
  - **A register**, optionally shifted by an immediate value
- This can be either **added** or **subtracted** from the base register:
  - Prefix the offset value or register with '+' (default) or '-'.
- This offset can be applied:
  - **before** the transfer is made: Pre-indexed addressing
    - **optionally auto-incrementing** the base register, by postfixing the instruction **with an '!'.**
  - **after** the transfer is made: Post-indexed addressing
    - causing the base register to be **auto-incremented**.

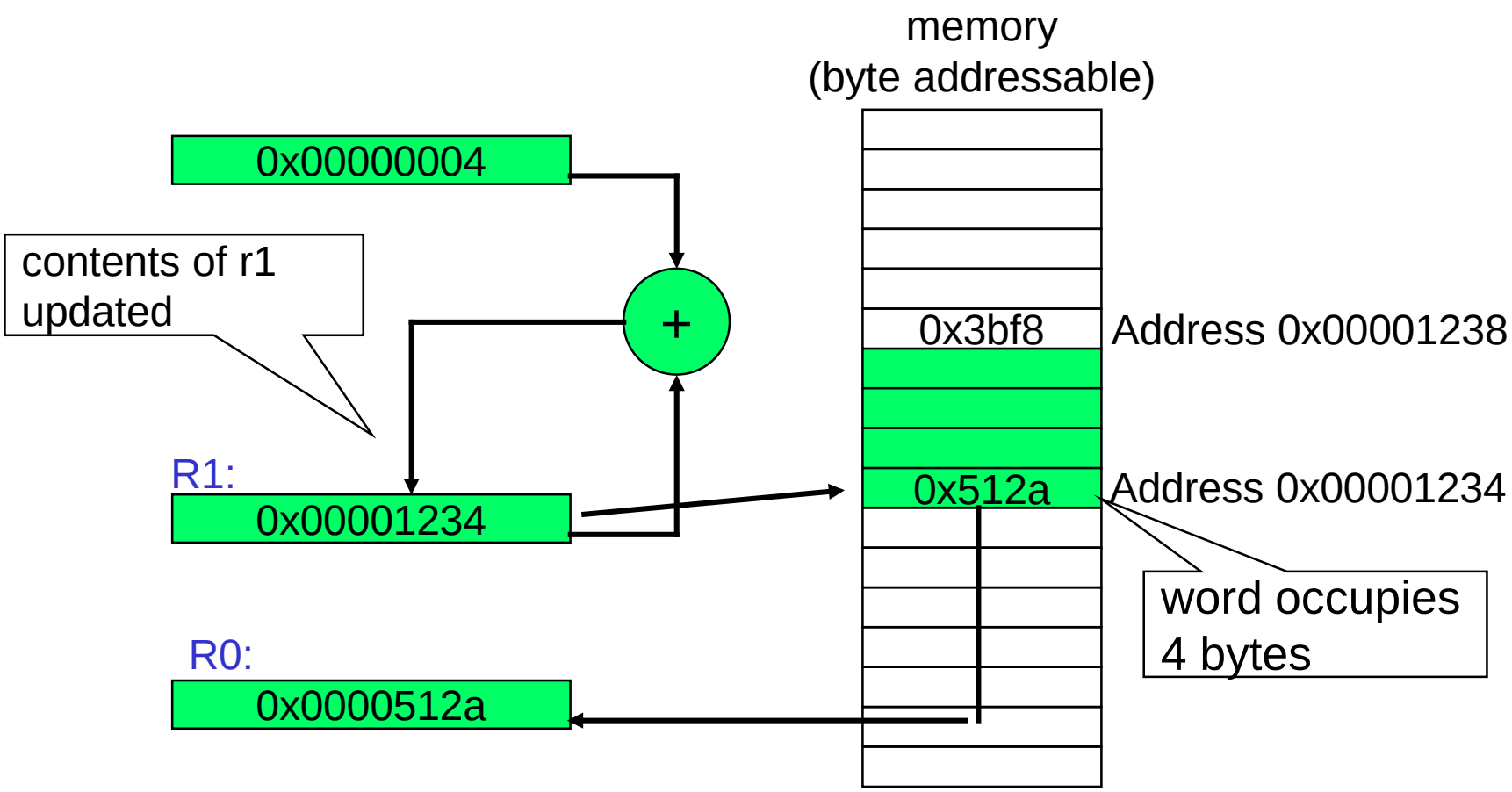
Pre-indexed Mode

LDR r0, **[r1, #4]** ; r0 ← [r1+4]



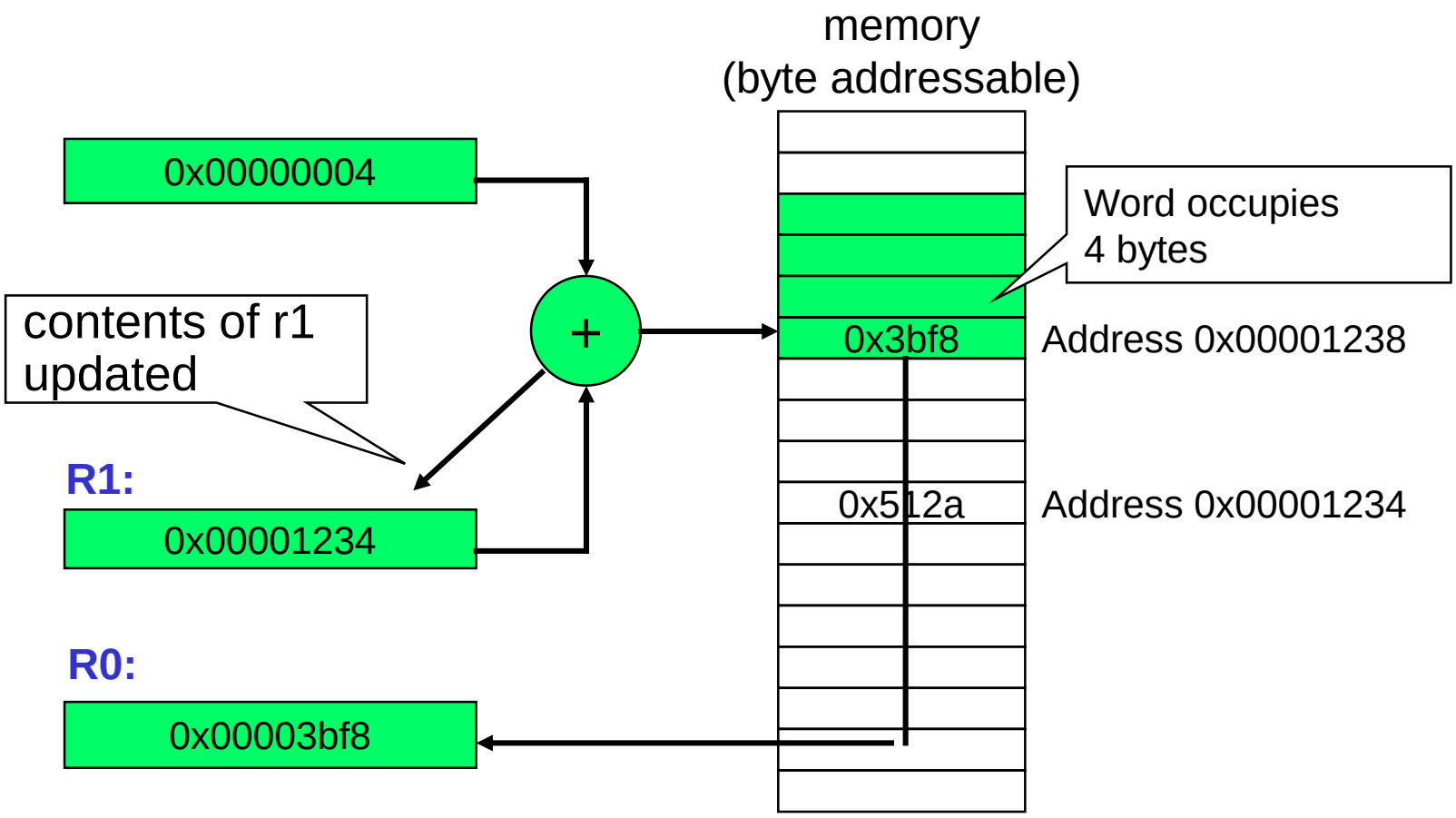
Post-indexed Mode

LDR r0, **[r1], #4** ; r0 ← [r1], r1 ← r1+4



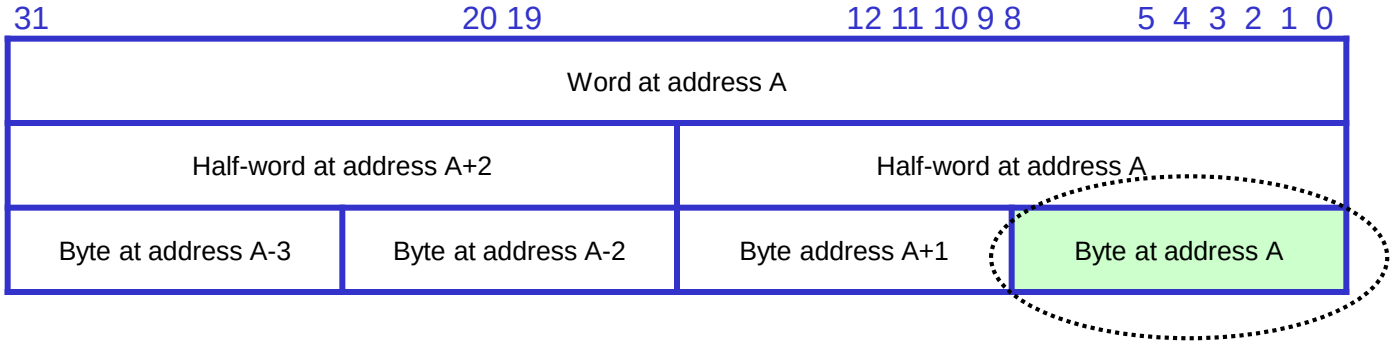
Auto Indexing Mode (Pre-indexed Mode w/autoinc)

LDR r0, **[r1, #4]!** ; r1← r1+4, r0 ← [r1]

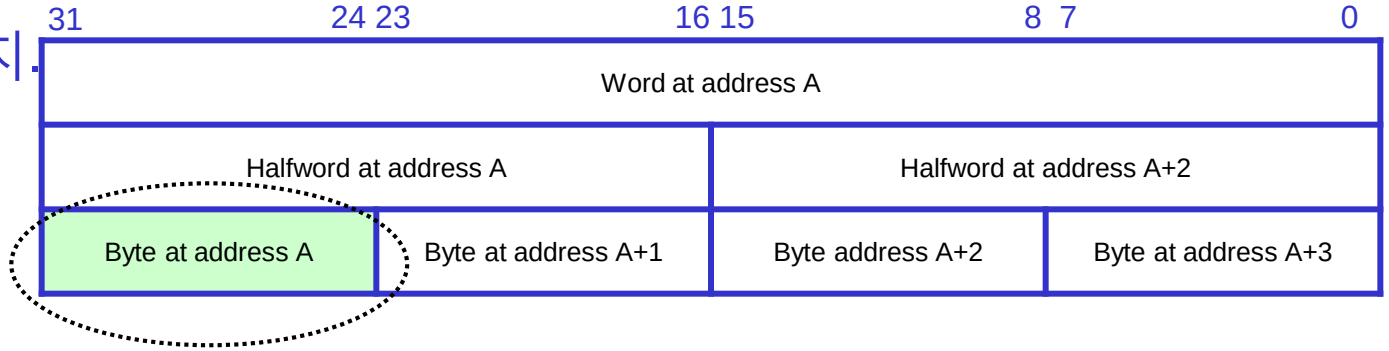


Endianess

- ARM은 little 혹은 big endian format 으로 데이터 접근 가능
- **Little endian**: word 의 LSB(Least significant byte)가 bits 0-7에 위치.



- **Big endian**: word 의 LSB(Least significant byte)가 bits 24-31에 위치.



Block Data Transfer (1)

- The Load and Store Multiple 명령어: LDM / STM
  - (1 ~16 registers) - memory 간의 데이터 전달
- 전달 가능한 레지스터:
  - Any subset of the current bank of registers (default).
  - Any subset of the user mode bank of registers when in a privileged mode (**postfix instruction with '^'**).
- **Base register는 메모리 접근이 일어날 위치를 결정**
  - 4 different addressing modes
    - increment and decrement
    - inclusive or exclusive of the base register location.
  - Base register can be optionally updated following the transfer (by appending it **with an '!'**).

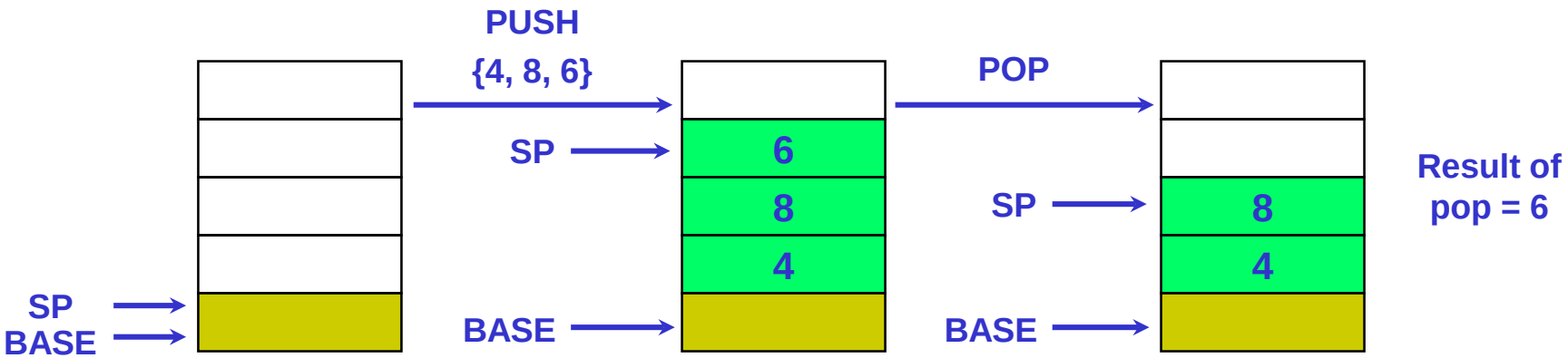
Block Data Transfer (2)

- These instructions are very efficient for
  - Saving and restoring context
    - Useful to view memory as a stack.
  - Moving large blocks of data around memory
    - Useful to directly represent functionality of the instructions.

- 예제
  - STMFD r13!, {r0-r2, r14}
  - LDMFD r13!, {r8-r12,pc}^

스택(Stacks)

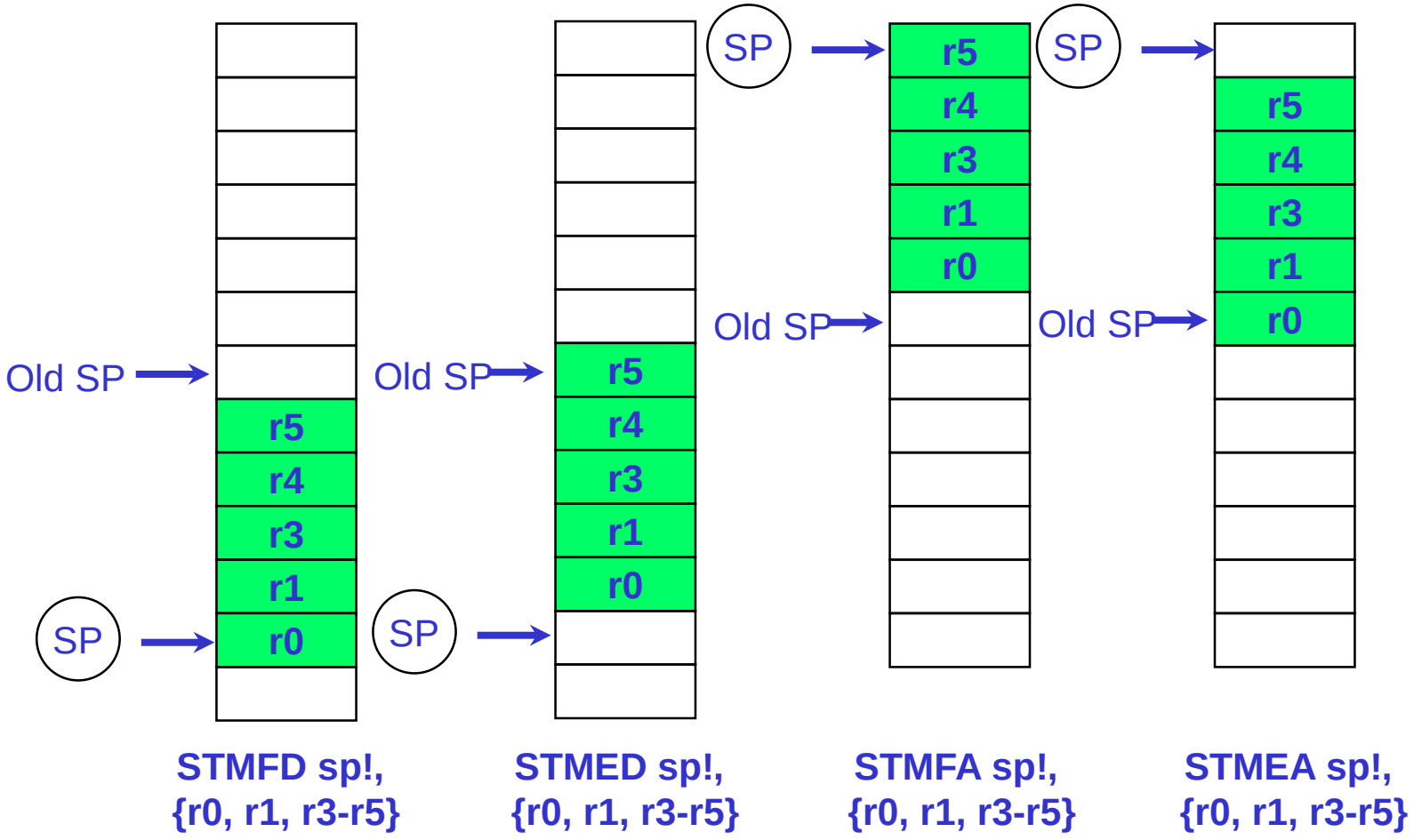
- 스택 연산을 제공하기 위해 2 개의 포인터 정의.
  - A base pointer (frame pointer):
    - used to point to the “bottom” of the stack (the first location).
  - A stack pointer:
    - used to point the current “top” of the stack.



스택 type

- The value of the stack pointer can either:
  - Point to the **last occupied address** (Full stack) and so needs **pre-decrementing** (i.e. before the push)
  - Point to the **next occupied address** (Empty stack) and so needs **postdecrementing** (i.e. after the push)
- The stack type to be used is given by the postfix to the instruction:
  - STM**FD** / LDMFD : Full Descending stack
    - Note: ARM Compiler will always use a Full descending stack.
  - STM**FA** / LDMFA : Full Ascending stack.
  - STM**ED** / LDMED : Empty Descending stack
  - STM**EA** / LDMEA : Empty Ascending stack

스택 type에 따른 포인트 변화



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!,{r0-r12, lr}      ; stack all registers
                            ; and the return address
        .....
        .....
LDMFD sp!,{r0-r12, pc}      ; load all the registers
                            ; and return automatically
```

- If the pop instruction also had the ‘S’ bit set (using ‘^’) then the transfer of the PC when in a privileged mode would also cause the SPSR to be copied into the CPSR.

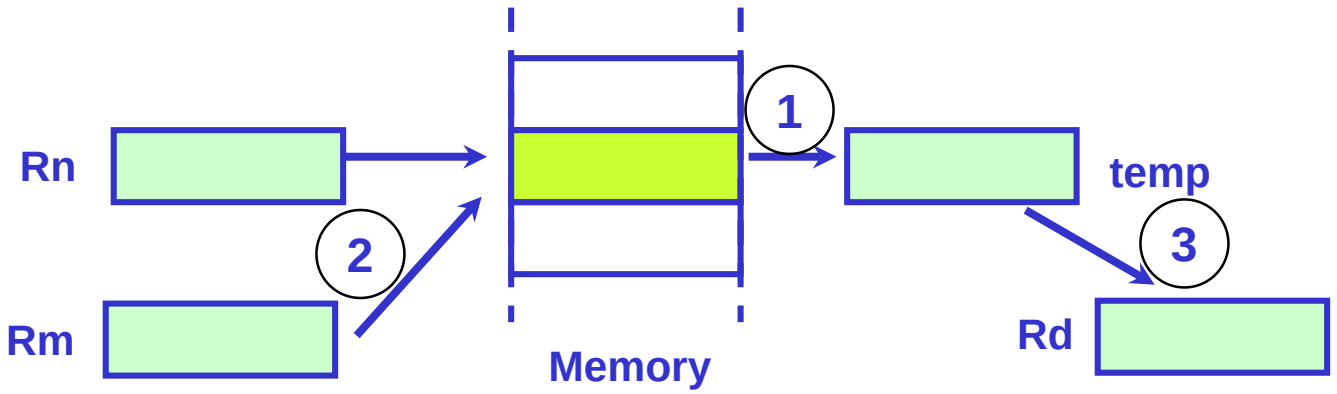
Block Data Transfer의 직접기능

- LDM / STM support a further syntax in addition to the stack one:
  - STMIA / LDMIA : Increment After
  - STMIB / LDMIB : Increment Before
  - STMDA / LDMDA : Decrement After
  - STMDB / LDMDB : Decrement Before

		Ascending		Descending	
		Full	Empty	Full	Empty
Increment	Before	STMIB STMFA			LDMIB LDMED
	After		STMIA STMEA	LDMIA LDMFD	
Decrement	Before		LDMDB LDMEA	STMDB STMFD	
	After	LDMDA LDMFA			STMDA STMED

Swap 명령어

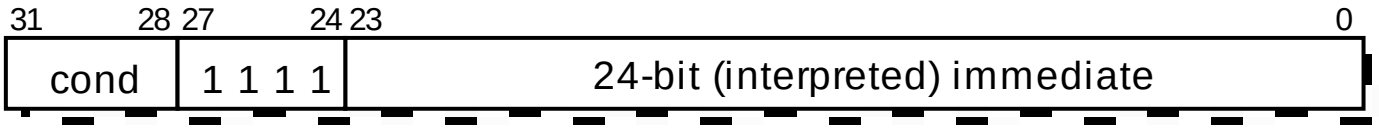
- Atomic operation of a memory read followed by a memory write which moves byte or word quantities between registers and memory.
- Syntax:
  - **SWP** {<cond>} {B} Rd, Rm, [Rn]



- Thus to implement an actual swap of contents make Rd = Rm.
- The compiler cannot produce this instruction.

Software Interrupt (SWI)

- Binary Encoding



- Description
  - Save the address of the next instruction in r14_svc
  - Save the CPSR in SPSR_svc
  - Enter supervisor mode and disable IRQ (but not FIQ) by setting CPSR[4:0] to 10011_2 and CPSR[7] to 1
  - Set the PC to 08_{16} and begin executing the instruction here
- By making use of the SWI mechanism, an operating system can implement a set of privileged operations, which, applications running in user mode can request.
- 예제:

```
MOV     r0, #'A'            ; get 'A' into r0
SWI     SWI_WriteC
```

Branch 명령어 (1)

- Branch : **B**{<cond>} label
- Branch with Link : **BL**{<cond>} sub_routine_label

- The offset for branch instructions is calculated by the assembler:
  - target address – branch instruction address – 8
    (to allow for the pipeline).
  - This gives a 26 bit offset which is right shifted 2 bits (as the bottom two bits are always zero as instructions are word – aligned) and stored into the instruction encoding.
  - This gives a range of ± 32 Mbytes.

Branch 명령어 (2)

- The "Branch with link" instruction implements a subroutine call by writing PC – 4 into the LR of the current bank.
  - i.e. the address of the next instruction following the branch with link (allowing for the pipeline).
- To return from subroutine, simply restore the PC from the LR:

  MOV pc, lr
  - Again, pipeline has to refill before execution continues.
- The "Branch" instruction does not affect LR.

MRS/MSR 명령어

- CPSR/SPSR을 읽거나 쓰기 위해 사용하는 명령어
- CPSR/SPSR 와 범용 레지스터 간의 데이터 이동 가능
  - **MRS** : Move to Register from Status Register
  - **MSR** : Move to Status Register from Register
  - All of status register, or just the flags, can be transferred.
- 문법:
  - MRS {<cond>} Rd,<psr> ; Rd = <psr>
  - MSR {<cond>} <psr>,Rm ; <psr> = Rm
  - MSR {<cond>} <psrf>,Rm ; <psrf> = Rm
    - where

      <psr> = CPSR, CPSR_all, SPSR or SPSR_all

      <psrf> = CPSR_flg or SPSR_flg
- 즉치 형태(immediate form)
  - MSR {<cond>} <psrf>,#Immediate
  - This immediate must be a 32-bit immediate, of which the 4 most significant bits are written to the flag bits.

General register → Status register

- Assembly 형식
  - MSR{<cond>} CPSR_f|SPSR_f, #<32-bit immediate>
  - MSR{<cond>} CPSR_<field>|SPSR_<field>, Rm
    - Field
      - c – control – PSR[7:0], x –extension – PSR[15:8]
      - s – status – PSR[23:16], f – flags – PSR[31:24]
- 예제
  - To set the N, Z, C, and V flags
    - MSR CPSR_f, #&f0000000 ; set all the flags
  - To set just the C flag, preserving N, Z, and V
    - MRS r0, CPSR ; move the CPSR to r0
    - ORR r0, r0, #&20000000 ; set bit 29 of r0
    - MSR CPSR_f, r0 ; move back to CPSR
  - To switch from supervisor mode into IRQ mode
    - MRS r0, CPSR ; move the CPSR to r0
    - BIC r0, r0, #&1f ; clear the bottom 5 bits
    - ORR r0, r0, #&12 ; set the bits to IRQ mode
    - MSR CPSR_c, r0 ; move back to CPSR

Coprocessor(1)

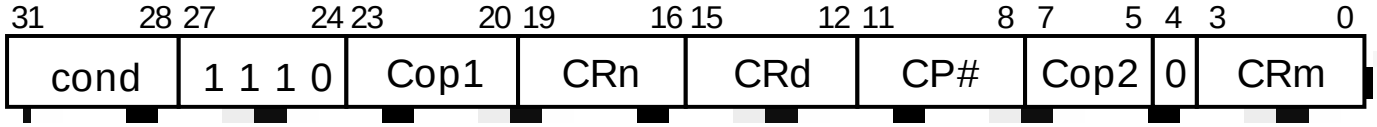
- Coprocessor
  - General mechanism to extend the instruction set through the addition to the core
  - Example : system controller such as MMU & cache. FPU
  - Registers
    - private to coprocessor
    - ARM controls the data flow
      - Coprocessor concerns only the data processing and memory transfer operations

Coprocessor(2)

- The ARM architecture supports 16 coprocessors
- Each coprocessor instruction set occupies part of the ARM instruction set.
- There are three types of coprocessor instruction
  - Coprocessor data processing
  - Coprocessor (to/from ARM) register transfers
  - Coprocessor memory transfers (load and store to/from memory)
- Assembler macros can be used to transform custom coprocessor mnemonics into the generic mnemonics understood by the processor.
- A coprocessor may be implemented
  - in hardware
  - in software (via the undefined instruction exception)
  - in both (common cases in hardware, the rest in software)

코프로세서 데이터 연산 명령어

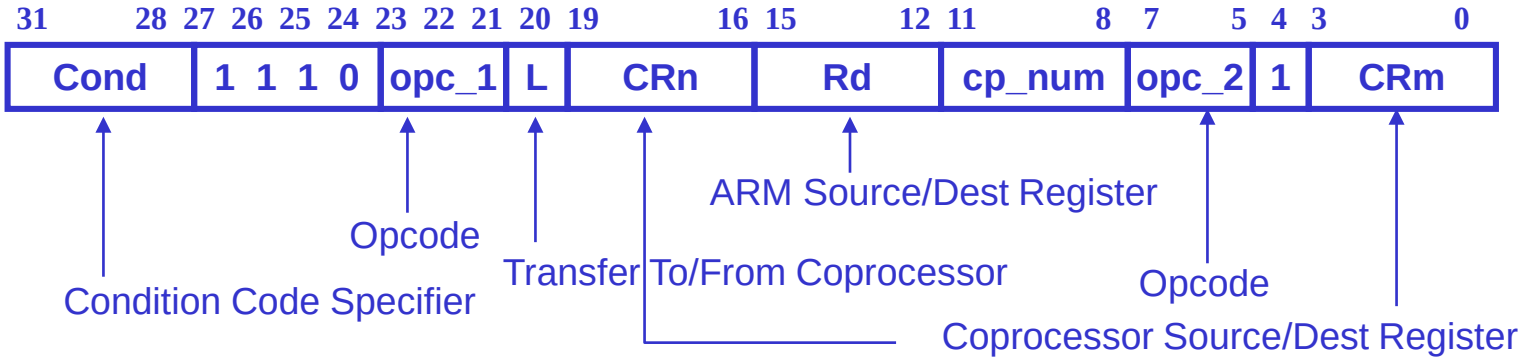
- Coprocessor data operation (CDP)
  - This class of instruction is used to tell a coprocessor to perform some internal operation
  - No result is communicated back to ARM, and it will not wait for the operation to complete
- Binary encoding



- 문법:
  - **CDP**{<cond>} <CP#>, <Cop1>, CRd, CRn, CRm{, <Cop2>}
- 예제:
  - CDP p2, 3, C0, C1, C2
  - CDPEQ p3, 6, C1, C5, C7, 4

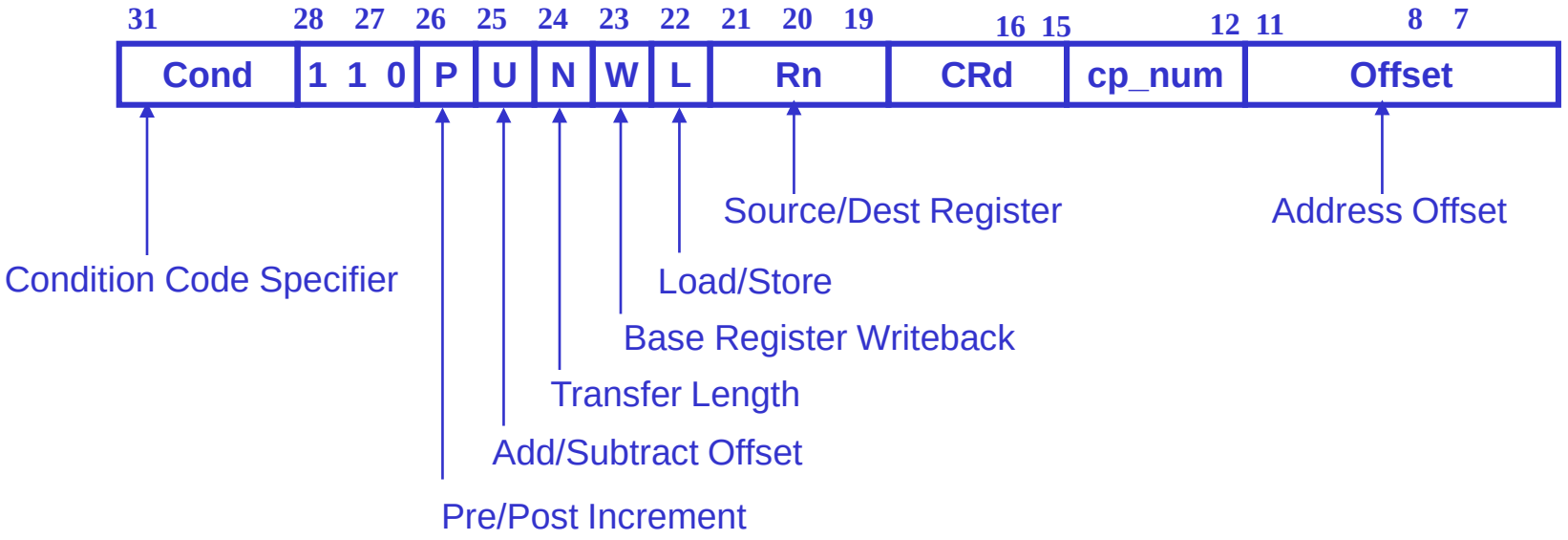
Coprocessor Register Transfers

- ARM registers 와 coprocessor registers 간의 데이터 이동 수단
  - MRC : Move to Register from Coprocessor
  - MCR : Move to Coprocessor from Register
- An operation may also be performed on the data as it is transferred
  - For example a Floating Point Convert to Integer instruction can be implemented as a register transfer to ARM that also converts the data
- Syntax
  - <**MRC**|**MCR**>{<cond>} <cp_num>,<opc_1>,Rd,CRn,CRm,<opc_2>



Coprocessor Memory Transfers (1)

- Load from memory to coprocessor registers
- Store to memory from coprocessor registers



Coprocessor Memory Transfers (2)

- Syntax of these is similar to word transfers between ARM and memory:
  - <**LDC**|**STC**> {<cond>} {<L>} <cp_num>,CRd,<address>
    - PC relative offset generated if possible, else causes an error.
  - <LDC|STC> {<cond>} {<L>} <cp_num>,CRd,<[Rn,offset]{!}>
    - Pre-indexed form, with optional writeback of the base register
  - <LDC|STC> {<cond>} {<L>} <cp_num>,CRd,<[Rn],offset>
    - Post-indexed form

  where
  - <L> when present causes a “long” transfer to be performed (N=1) else causes a “short” transfer to be performed (N=0).
    - Effect of this is coprocessor dependant.