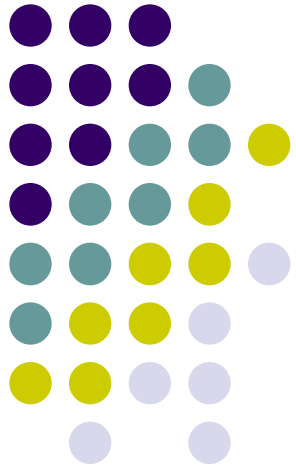
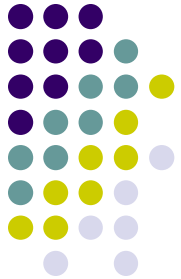
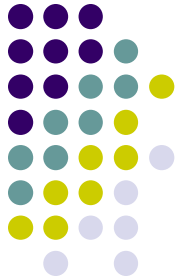
ARM 명령어 집합





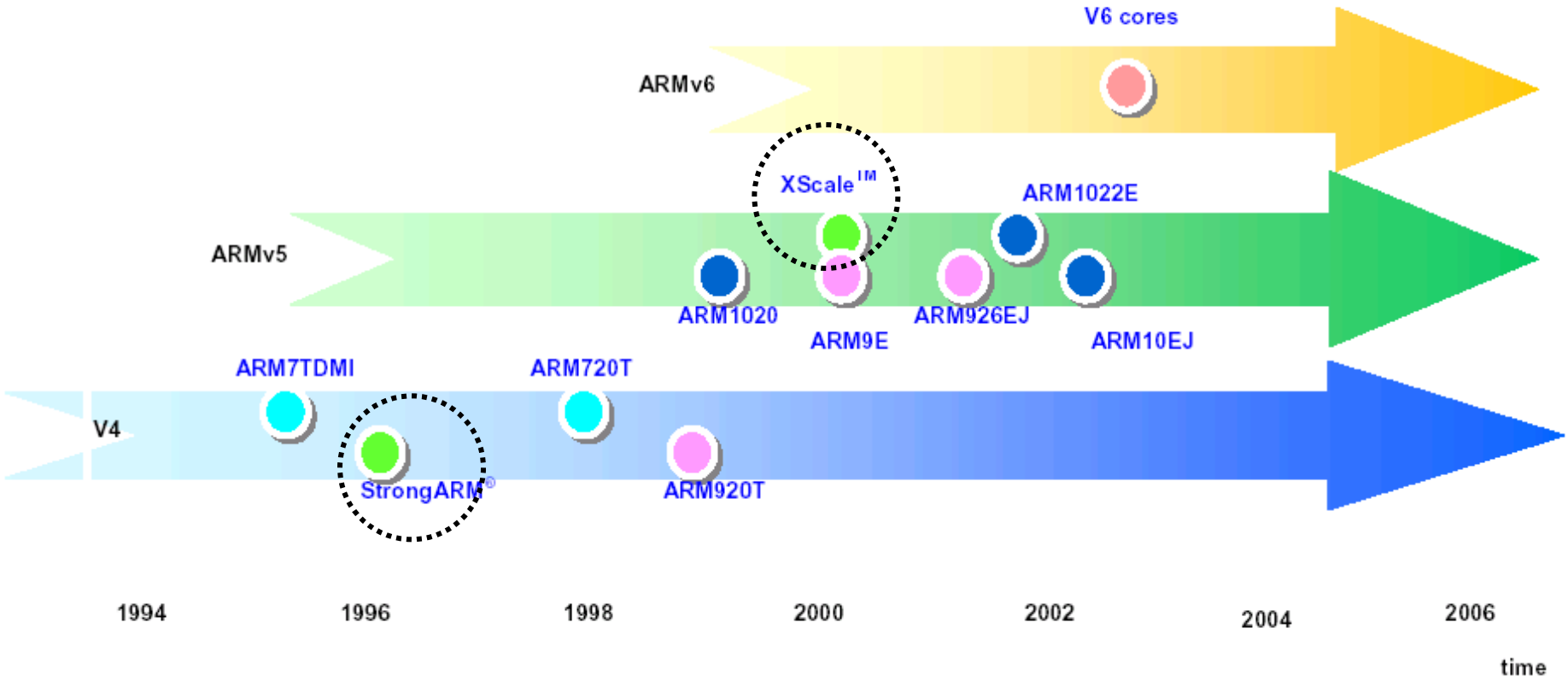
목 차

ARM 구조 특징

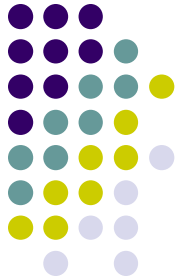


- 모든 명령어가 32bit 크기를 가짐
- 대부분의 명령어가 1cycle에 수행됨
- 32bit 크기의 많은 레지스터
- load/store architecture
- 모든 명령어에서 조건 수행 기능 있음
- 간단한 어드레싱 모드
- 같은 규격의 간단한 명령어
- 한 명령어에서 ALU와 shifter 동시 사용 가능
- loop을 위한 자동 증가/감소 어드레싱 모드
- 많은 데이터 처리를 위한 multiple load/store 명령어

ARM 구조 버전


V6 cores
ARMv6
XScale™
ARM1022E
ARMv5
ARM1020
ARM9E
ARM926EJ
ARM10EJ
ARM7TDMI
ARM720T
V4
StrongARM®
ARM920T
1994
1996
1998
2000
2002
2004
2006
time

프로세서 수행 모드



사용자 모드	usr	사용자 프로그램 수행할 때 **(User)**
특권 모드	sys	**Kernel** 프로그램 수행 할 때 **(System)**
특권 모드, 예외 모드	svc	전원 **reset** 발생 시 혹은 소프트웨어 인터럽트 발생 시 **(Supervisor)**
특권 모드, 예외 모드	abt	메모리 보호 공간 혹은 없는 메모리 공간 액세스 시 **(Abort)**
특권 모드, 예외 모드	und	잘못된 명령어를 사용할 때 **(Undefined)**
특권 모드, 예외 모드	irq	일반 인터럽트 발생 시 **(Interrupt)**
특권 모드, 예외 모드	fiq	우선 순위 인터럽트 발생 시 **(Fast Interrupt)**

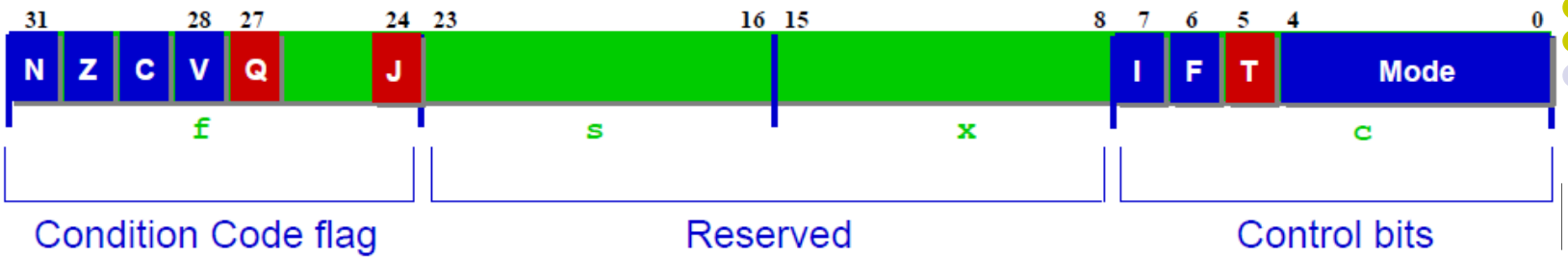
레지스터 구조

- 31개의 범용 레지스터(R0 ~ R15)가 프로세서 모드에 따라 선택되어 사용됨
- 특별한 의미를 가지는 범용 레지스터
  - R13 = SP(stack pointer)
  - R14 = LR(link register)
  - R15 = PC(program counter)
- 예외가 발생되면 R13 및 R14가 예외용 R13 및 R14로 대치됨
- fiq 예외는 R8 ~ R14가 대치됨
- 6개의 프로그램 상태 레지스터(CPSR 1개 및 SPSR 5개) – 특별한 명령어로만 접근 가능

User/ System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
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 (SP)	R13_SVC	R13_ABORT	R13_UNDEF	R13_IRQ	R13 _FIQ
R14 (LR)	R14_SVC	R14_ABORT	R14_UNDEF	R14_IRQ	R14 _FIQ
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_SVC	SPSR_ABORT	SPSR_UNDEF	SPSR_IRQ	SPSR_FIQ

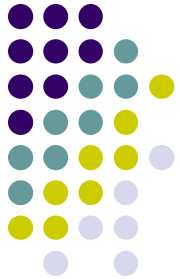
현재 프로그램 상태 레지스터 (CPSR)



- 조건 코드 비트
  - N = 계산 결과가 음수
  - Z = 계산 결과가 '0'
  - C = 계산 후 carry 발생
  - V = 계산 후 overflow 발생
- 인터럽트 disable 비트
  - I = 1, IRQ를 disable함
  - F = 1, FIQ를 disable함
- T 비트: '0'이면 명령어가 ARM 명령어이고 '1'이면 Thumb 명령어를 표시
- mode 비트: 프로세서의 7개 모드 중 하나를 표시

mode	M[4:0]
usr	10000
fiq	10001
irq	10010
svc	10011
abt	10111
und	11011
sys	11111

프로그램 카운터(PC)

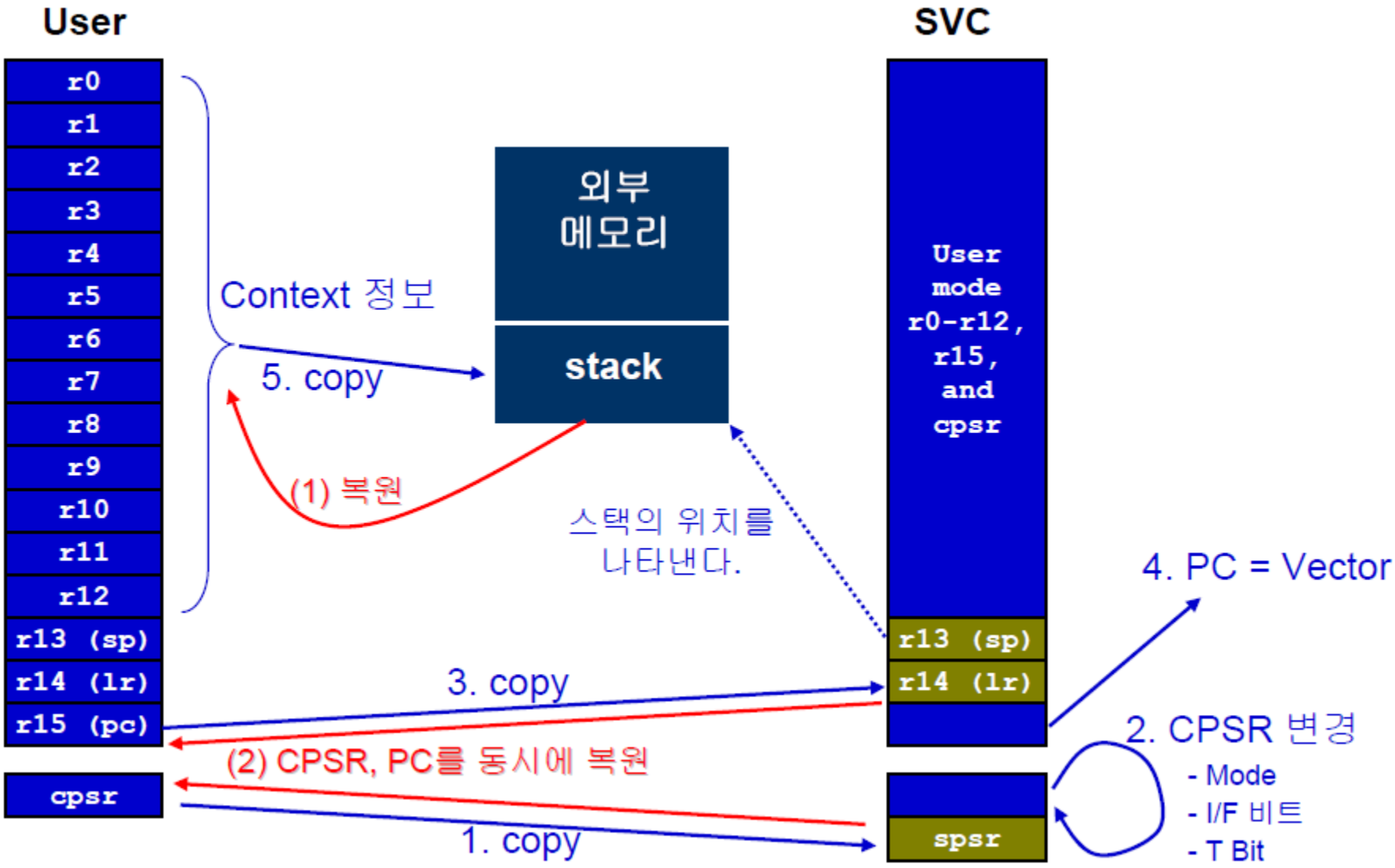


- 모든 명령어는 길이가 32 bit이고 메모리에 저장될 때 word(=4 bytes) align되어야 함
- R15(=PC)의 값은 bit 2~31에 저장되고 bit 0~1은 항상 '0'임
- R14(=LR)는 명령어 BL(Branch with Link)를 사용하여 subroutine 호출 시 return 주소가 자동 저장됨
  - subroutine call할 때
    - BL SUB1
  - subroutine return할 때
    - MOV R15, R14 (혹은 MOV PC, LR)

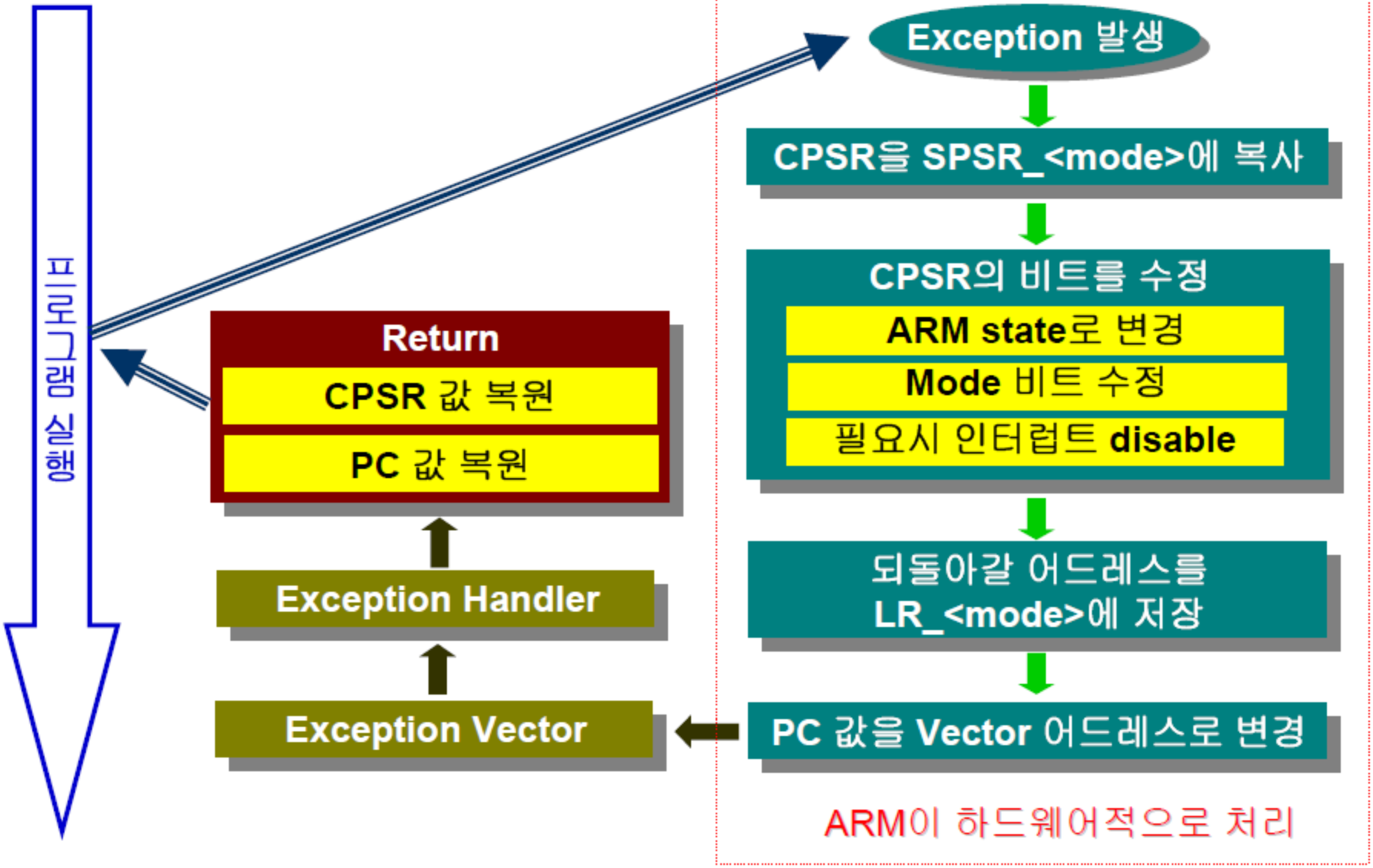
예외(exception) 처리 과정

- 예외 발생 시 (프로세서가 실행)
  - SPSR_<mode>에 CPSR을 복사
  - CPSR의 모드 bit 변경
  - CPSR의 I bit를 세트하고 mode가 fiq이면 F bit도 세트
  - 해당 모드의 레지스터 사용으로 변경
  - LR_<mode>에 return 주소 저장
  - PC에 해당 예외 vector 주소를 저장
- 예외 복귀 시 (프로그램에서 수행)
  - CPSR에 SPSR_<mode>를 복사
  - PC에 LR_<mode>를 복사

예외 종류	예외 모드	예외 vector 주소
Reset	svc	0x00000000
Undefined instructions	und	0x00000004
Software Interrupt(SWI)	svc	0x00000008
Instruction fetch memory abort	abt	0x0000000c
Data access memory abort	abt	0x00000010
IRQ (Interrupt)	irq	0x00000018
FIQ (Fast Interrupt)	fiq	0x0000001c

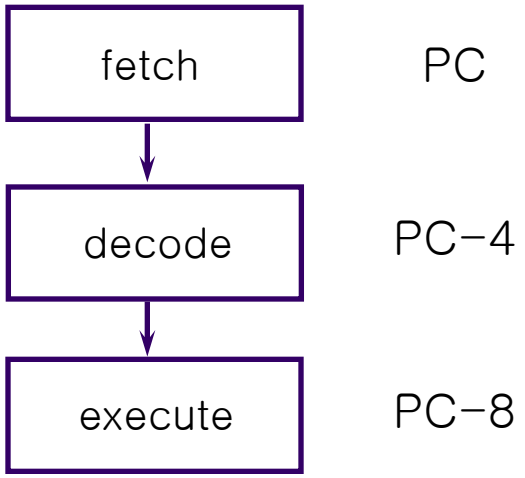

User
r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
cpsr
외부
메모리
stack
Context 정보
5. copy
(1) 복원
스택의 위치를
나타낸다.
SVC
User
mode
r0-r12,
r15,
and
cpsr
r13 (sp)
r14 (lr)
spsr
4. PC = Vector
3. copy
(2) CPSR, PC를 동시에 복원
1. copy
2. CPSR 변경
- Mode
- I/F 비트
- T Bit

Exception Handling

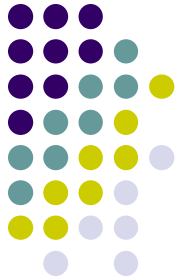


ARM 파이프라인

- ARM은 빠른 명령어 수행을 위하여 파이프라인을 사용
- 3 stage 파이프라인 경우 (1990년~1995년)
  - fetch – 명령어를 메모리에서 레지스터로 복사
  - decode – 명령어를 디코드
  - execute – 레지스터의 내용을 사용하여 계산한 후 레지스터에 저장



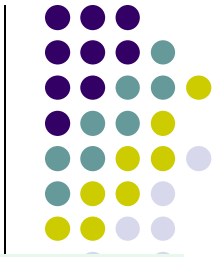
ARM 명령어 형식



- 기본 형식

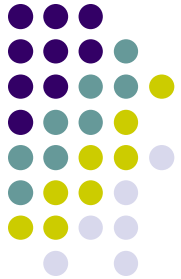
```
ADD Rd, Rn, Op2          ; Rd = Rn + Op2
```

Rd = destination register
Rn = operand 1 (항상 register임)
Op2 = operand 2 (register 혹은 immediate 값)
; 다음은 comment



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						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	Rn	1	0	0	1	Rm	Multiply Long
cond	0	0	0	1	0	B	0	0	Rn	Rd	0 0 0 0	1	0	0	1	Rm	Data Swap
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	Rm	Branch and Exchange
cond	0	0	0	P	U	I	W	L	Rn	Rd	Ad_mode	1	S	H	1	Ad_mode	HW Data Transfer
cond	0	1	1	P	U	B	W	L	Rn	Rd	Offset						Single Data Transfer
cond	1	0	1												1		Undefined
cond	1	0	0	P	U	S	W	L	Rn	Register List							Block Data Transfer
cond	1	0	1	L	Signed_immed_24												Branch
cond	1	1	0	P	U	N	W	L	Rn	CRd	CP#	Offset					Co-processor Data Transfer
cond	1	1	1	0	Op_1				CRn	CRd	CP#	Op_2			0	CRm	Co-processor Data Operation
cond	1	1	1	0	Op_1			L	CRn	Rd	CP#	Op_2			1	CRm	Co-processor Register Transfer
cond	1	1	1	1	SWI number												Software Interrupt

조건 수행



- CPSR의 조건 코드 비트의 값에 따라 명령어를 실행하도록 하기 위해서는 적절한 조건을 접미사로 붙여주면 됨 :

```
ADD r0, r1, r2                  ; r0 = r1 + r2
ADDEQ r0, r1, r2                ; If Z is 1 then r0 = r1 + r2
```

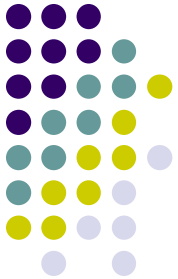
- 데이터 처리 명령어들의 결과는 CPSR의 조건 코드 비트에 영향을 미치지 못하도록 설계되어 있음
- 데이터 처리 명령어의 결과가 CPSR의 조건 코드 비트를 변경하기 위해서는 명령어에 접미사로 'S'를 명시하여야 함

```
ADDS r0, r1, r2                 ; r0 = r1 + r2 and set condition bits
```

조건 수행 접미사

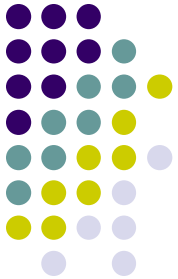
Cond	Mnemonic	Meaning	Condition flag state
0000	EQ	Equal	Z = 1
0001	NE	Not Equal	Z = 0
0010	CS/HS	Carry set / unsigned >=	C = 1
0011	CC/LO	Carry clear / unsigned <	C = 0
0100	MI	Minus/Negative	N = 1
0101	PL	Plus/Positive or Zero	N = 0
0110	VS	Overflow	O = 1
0111	VC	No overflow	O = 0
1000	HI	Unsigned higher	C=1 & Z=0
1001	LS	Unsigned lower or same	C=0 | Z=1
1010	GE	Signed >=	N==V
1011	LT	Signed <	N!=V
1100	GT	Signed >	Z==0,N==V
1101	LE	Signed <=	Z==1 or N!=V
1110	AL	Always	
1111	(NV)	Unpredictable	

데이터 처리 명령어



- 관련 명령어 종류
  - 산술 연산
  - 비교 연산
  - 논리 연산
  - 데이터 이동 연산
- ARM은 load/store architecture
  - 데이터 처리 명령어는 메모리에 직접 적용이 불가능하고 레지스터에만 적용됨.
- 하나 혹은 두개의 operand에 대하여 계산 처리
  - 첫째 operand는 항상 레지스터 (Rn)
  - 둘째 operand는 레지스터 혹은 immediate 값으로 barrel shifter를 통하여 ALU로 보내짐

산술 연산

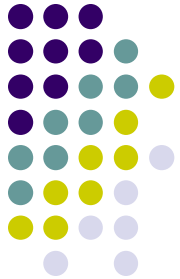


- 산술 연산 명령어 및 동작:
  - ADD: operand1 + operand2
  - ADC: operand1 + operand2 + carry
  - SUB: operand1 – operand2
  - SBC: operand1 – operand2 + carry –1
  - RSB: operand2 – operand1
  - RSC: operand2 – operand1 + carry – 1
- 문법:
  - <Operation>{<cond>}{S} Rd, Rn, Operand2
- 예제
  - ADD r0, r1, r2
  - SUBGT r3, r3, #1
  - RSBLES r4, r5, #5

비교 연산

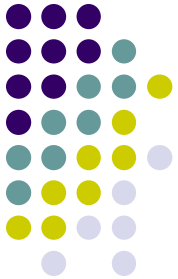
- 비교 연산의 결과는 CPSR의 조건 코드 비트를 변경 ('S' 비트를 set 할 필요가 없음)
- 명령어 및 동작:
  - CMP: operand1 – operand2, but result not written
  - CMN: operand1 + operand2, but result not written
  - TST: operand1 AND operand2, but result not written
  - TEQ: operand1 EOR operand2, but result not written
- 문법:
  - <Operation>{<cond>} Rn, Operand2
- 예제:
  - CMP r0, r1
  - TSTEQ r2, #5

논리 연산



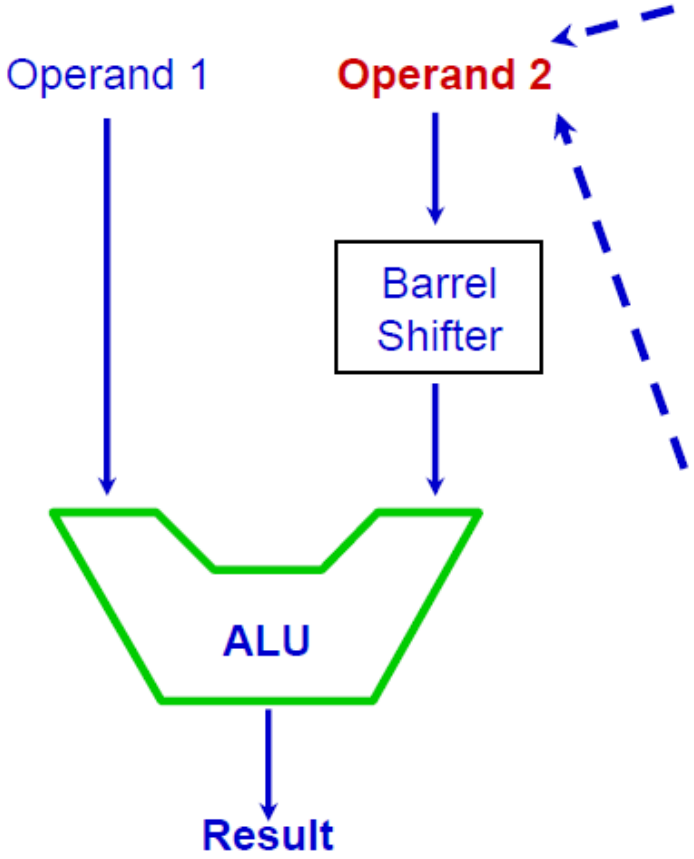
- 관련 명령어:
  - AND: operand1 AND operand2
  - EOR: operand1 EOR operand2
  - ORR: operand1 OR operand2
  - BIC: operand1 AND NOT operand2 (bit clear)
- 문법:
  - <Operation> {<cond>} {S} Rd, Rn, Operand2
- 예제:
  - AND r0, r1, r2
  - BICEQ r2, r3, #7
  - EORS r1, r3, r0

데이터 이동 연산



- 관련 명령어:
  - MOV: operand1 <- operand2
  - MVN: operand1<- not operand2
- 문법:
  - <Operation> {<cond>} {S} Rd, Operand2
- 예제:
  - MOV r0, r1 ; r0 <- r1
  - MOVS r2, #10 ; r2 <- 10 and set 'S' bit
  - MVNEQ r1, #0 ; if zero flag set then r1 <- not 0

Shift



- 레지스터
  - Shift 동작과 같이 사용 가능
  - Shift value
    - ✓ 5 bit unsigned integer
    - ✓ 하위 5비트에 shift value를 가진 다른 register

- **Immediate** 상수
  - 8 bit number
  - 짝수(even number) 만큼 rotate right 하여 표현이 가능한 32비트 상수
    - ✓ 32 비트 상수가 사용되면 어셈블러가 rotate 값으로 계산

논리/산술 Shift

- Logical Shifts Left

  LSL #5 ; multiply by 32

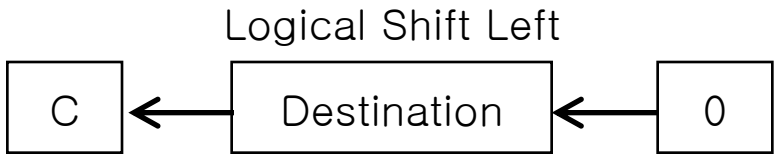
  (LSL = ASL)

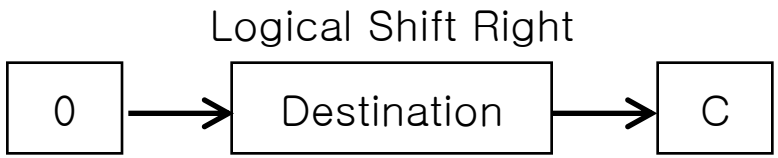
- Logical Shift Right

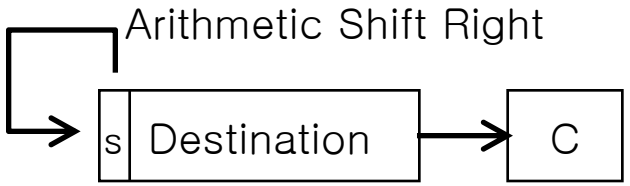
  LSR #5 ; divide by 32

- Arithmetic Shift Right

  ASR #5 ; divide by 32 (signed)





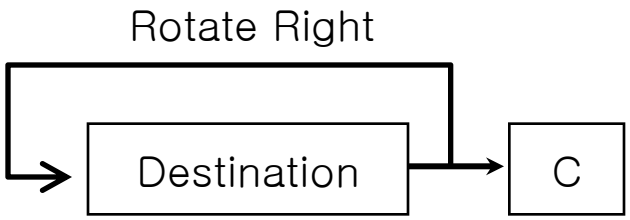


sign 비트 유지

회전 Shift

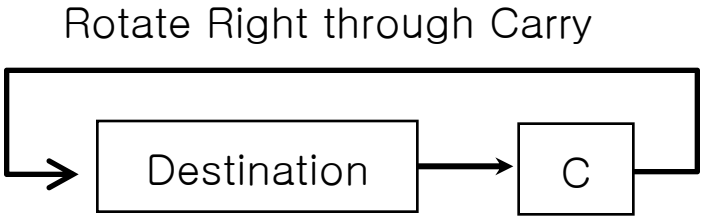
- Rotate Right (ROR)

  ROR #5

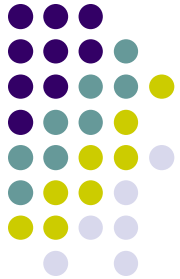


- Rotate Right Extended (RRX)

  RRX #5

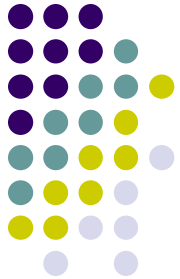


Shift Operand



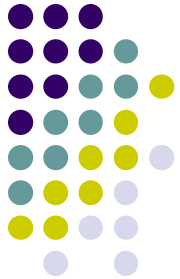
- 레지스터가 shift되는 양은 2가지 방법으로 지정.
  - immediate 5-bit
    - ADD r5, r5, r3 LSL #3
    - 추가 cycle 필요 없음
  - 레지스터의 bottom byte 사용 (PC는 안됨)
    - ADD r5, r5, r3 LSL r2
    - 추가 cycle 필요
- Shift가 명시되지 않으면 default shift 적용: LSL #0

곱셈 명령어



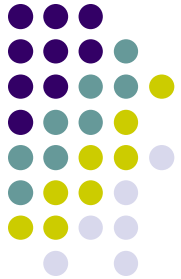
- ARM은 기본적으로 2 개의 곱셈 명령어 제공
  - Multiply
    - MUL {<cond>} {S} Rd, Rm, Rs ; Rd = Rm * Rs
  - Multiply Accumulate – does addition for free
    - MLA {<cond>} {S} Rd, Rm, Rs, Rn ; Rd = (Rm * Rs) + Rn
- 사용 상의 제한:
  - Rd 과 Rm 는 동일한 레지스터이어서는 안됨

Load/Store 명령어



- ARM은 load/store architecture:
  - memory to memory 데이터 처리 명령을 지원하지 않음
  - 따라서 일단 사용하려는 데이터를 레지스터로 이동해야 함
- ARM은 메인 메모리와 상호 작용하는 세 종류의 명령어 집합이 있음
  - 1개 레지스터(single register) 데이터 이동 (LDR/STR).
  - 다수 레지스터(multiple register) 데이터 이동 (LDM/STM).
  - 데이터 스웝 (SWP).

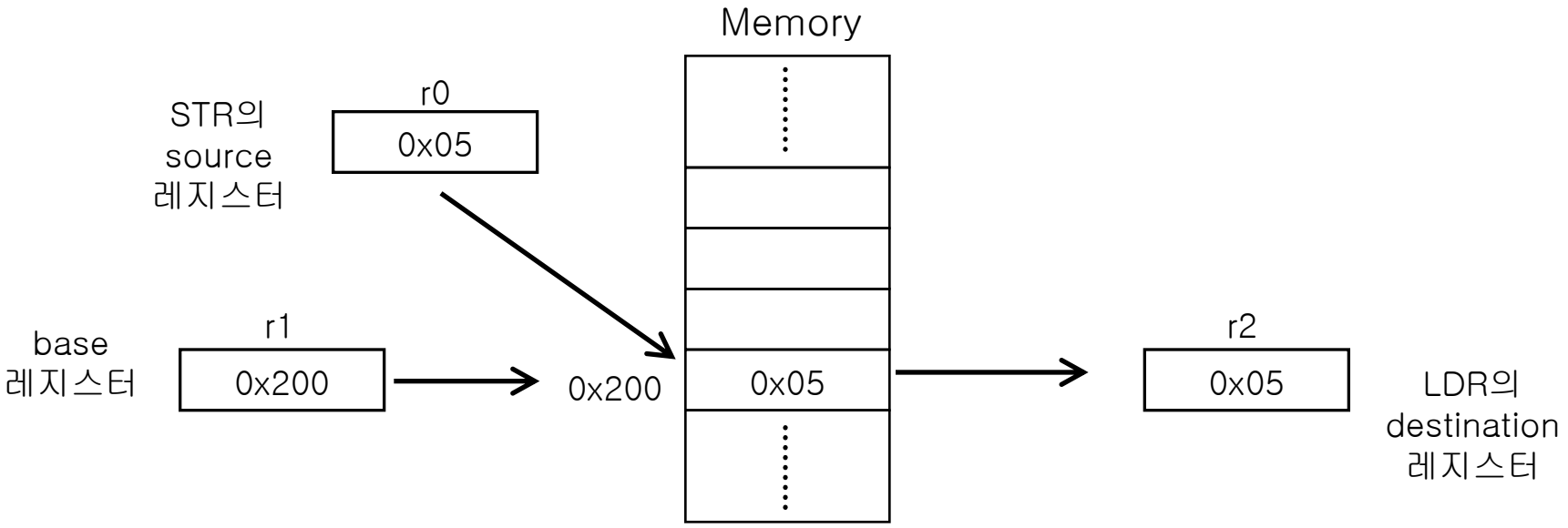
Load/Store – Single Register



- 기본적인 load/store 명령어
  - Word(4 bytes): LDR, STR
  - Halfword(2 bytes): LDRH, STRH
  - Byte(1 byte): LDRB, STRB

Load/Store – Base Register

- 접근될 메모리 위치는 base 레지스터에 유지함
  - STR r0, [r1] ; r0의 내용을 r1이 가리키는 메모리 위치에 저장
  - LDR r2, [r1] ; r1이 포인팅하는 메모리 위치의 정보를 r2에 저장



Load/Store – Offset

- base 레지스터 값으로부터 offset 만큼 떨어진 위치의 정보를 접근하는 명령어
  - pre-indexed

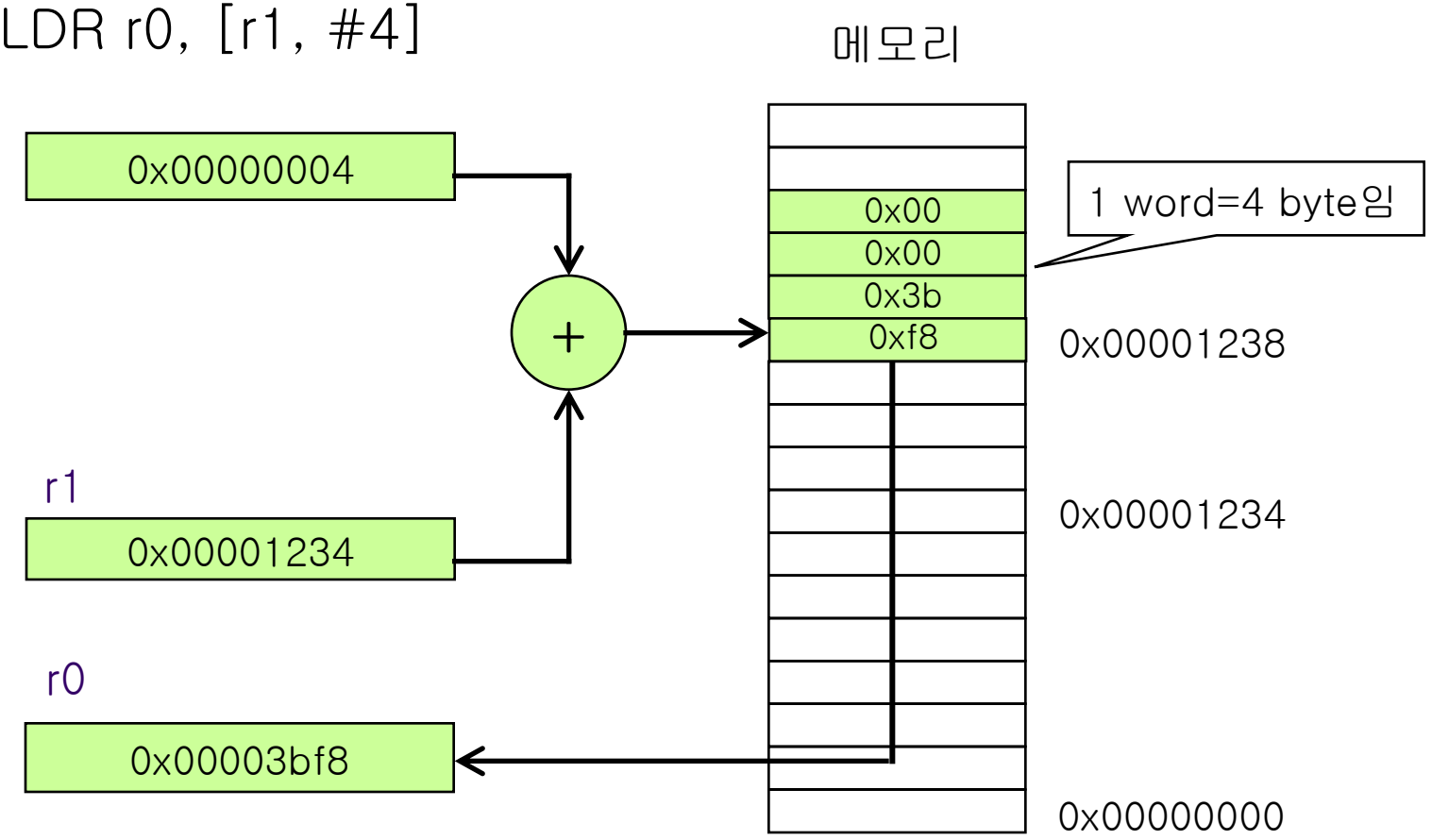
    LDR r0, [r1, #4] ; r0 := mem_{32}[r1 + 4]
  - post-indexed

    LDR r0, [r1], #4 ; r0 := mem_{32}[r1], r1 := r1 + 4
  - auto-indexing

    LDR r0, [r1, #4]! ; r0 := mem_{32}[r1 + 4], r1 := r1 + 4

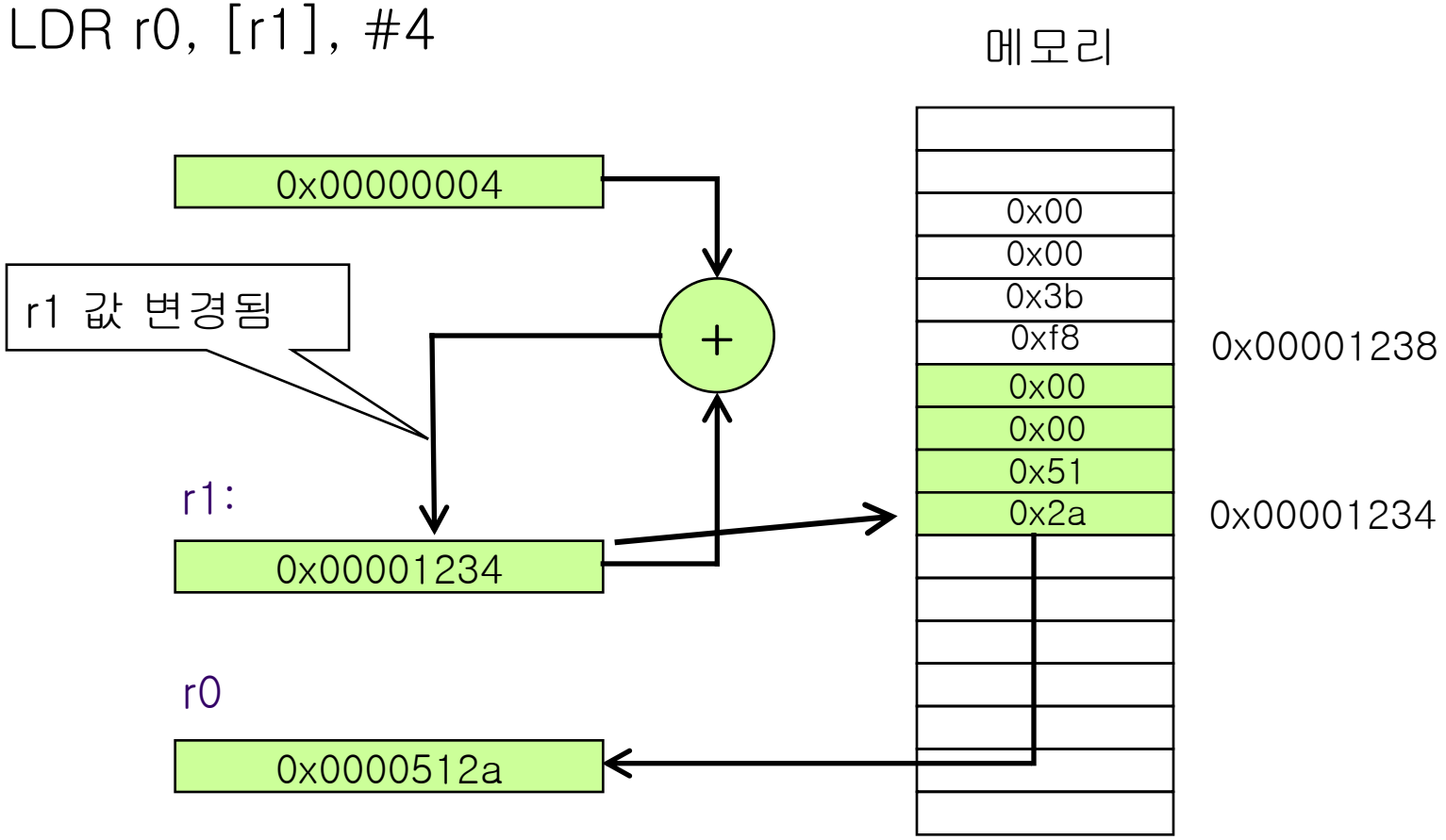
Load/Store – Pre-indexed

LDR r0, [r1, #4]



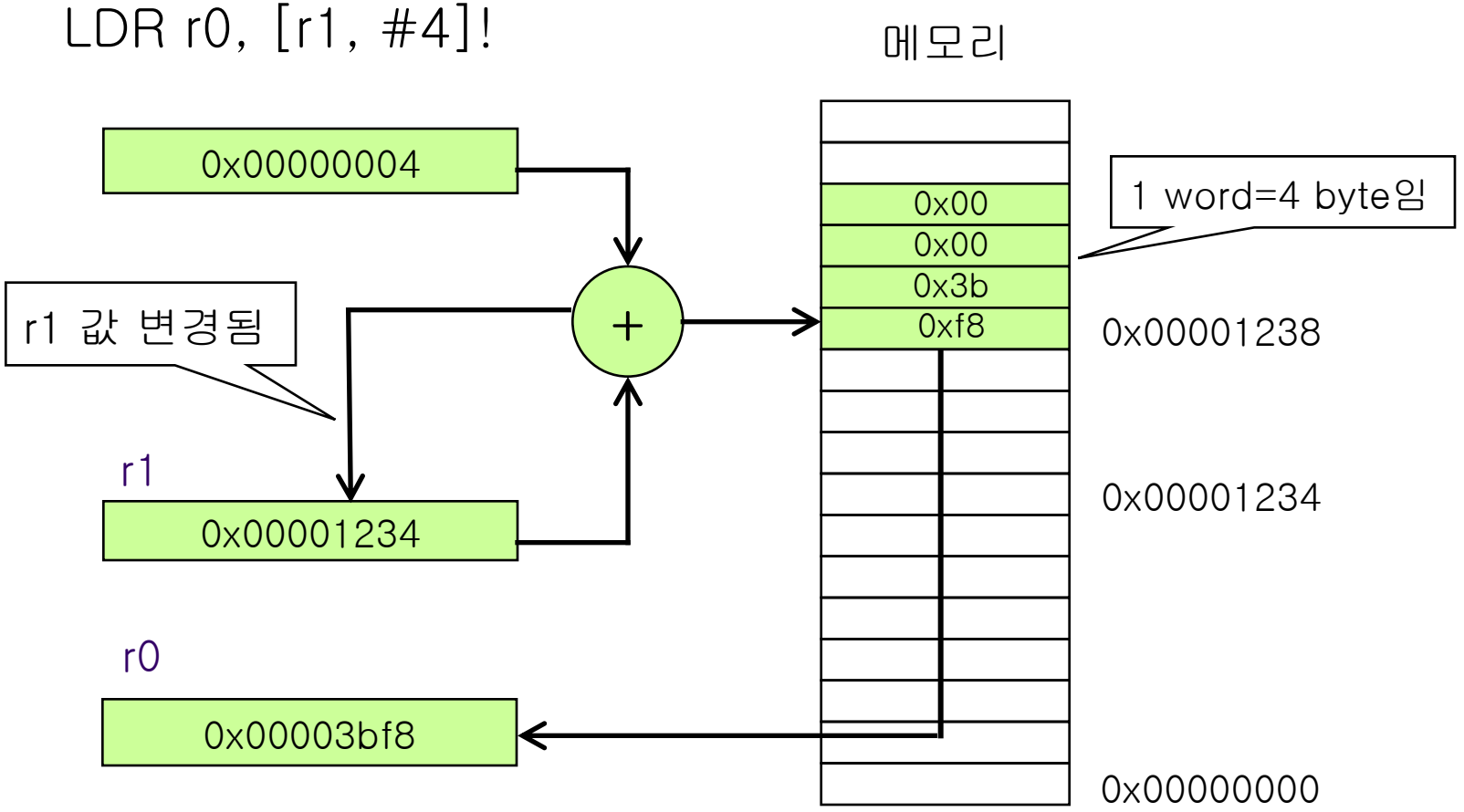
Load/Store – Post-indexed

LDR r0, [r1], #4

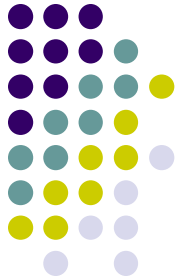


Load/Store – Auto Indexing

LDR r0, [r1, #4]!



다중 레지스터 Load/Store



- 기본 명령어
  - STM/LDM
- 확장 명령어
  - STMIA/LDMIA: Increment After
  - STMIB/LDMIB: Increment Before
  - STMDA/LDMDA: Decrement After
  - STMDB/LDMDB: Decrement Before

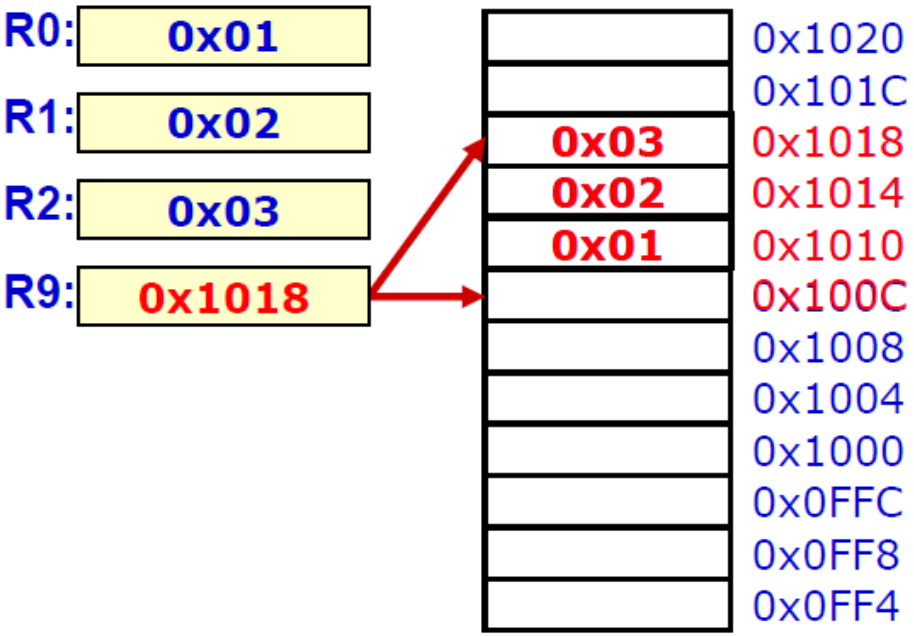
LDM/STM의 어드레스 지정 방식

Addressing Mode	키워드(표현방식)		유효 어드레스 계산
	데이타	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



- 데이터 저장 전 어드레스 증가

  (Increment Before)

  ① r9 -> 0x1010 증가 후 r0 저장

  ② r9 -> 0x1014 증가 후 r1 저장

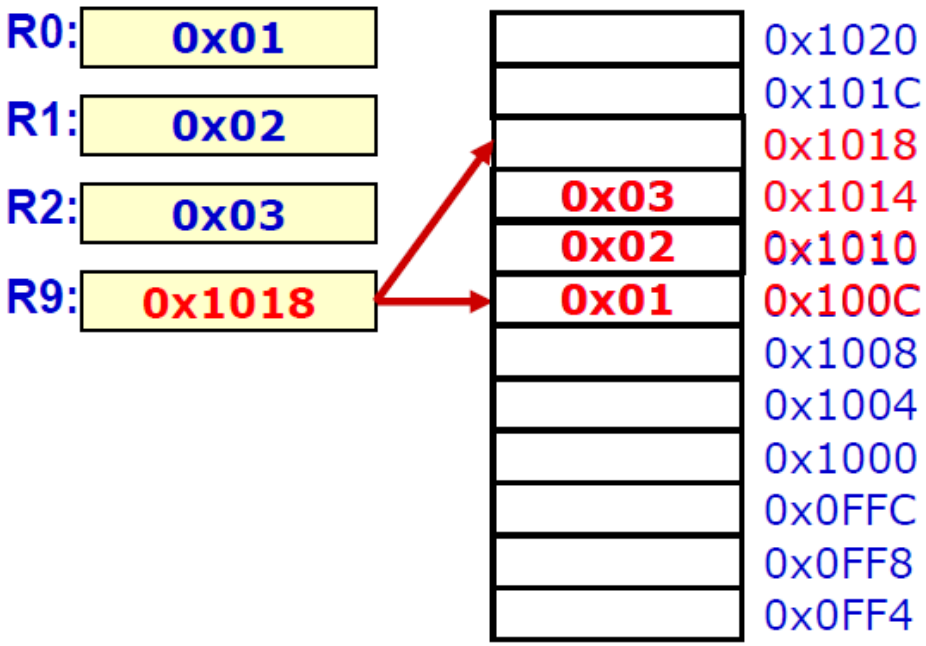
  ③ r9 -> 0x1018 증가 후 r2 저장

  ④ {!}, auto-update 옵션이 있으면 r9 값을 0x1018로 변경

Post-Increment 어드레스 지정

STMIA r9!, {r0,r1,r2}

Base 레지스터(r9) = 0x100C



- 데이터 저장 후 어드레스 증가

  (Increment After)

  ① 0x100c에 r0 저장 후 r9 -> 0x1010 증가

  ② 0x1010에 r1 저장 후 r9 -> 0x1014 증가

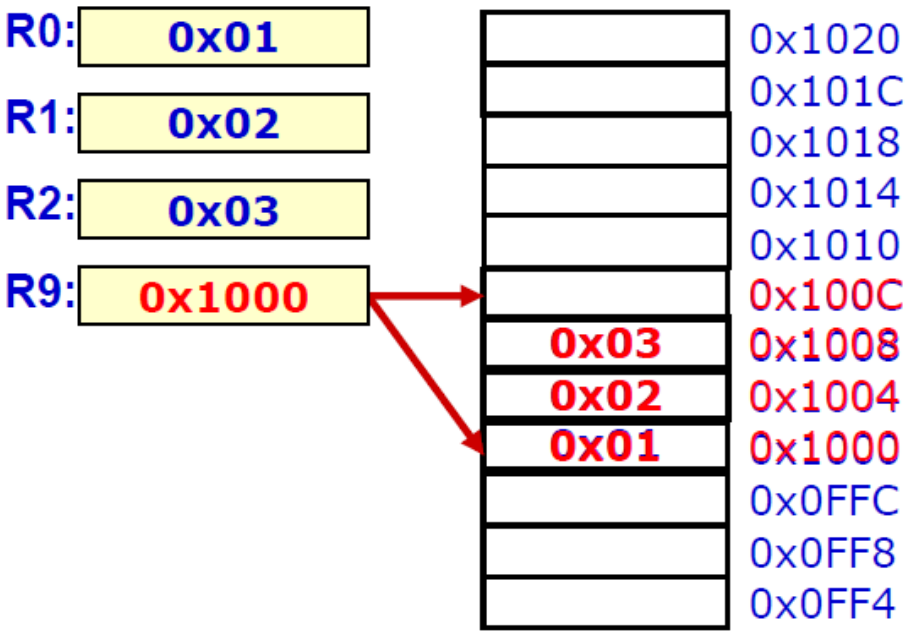
  ③ 0x1014에 r2 저장 후 r9 -> 0x1018 증가

  ④ {!}, auto-update 옵션이 있으면 r9 값을 0x1018로 변경

Pre-Decrement 어드레스 지정

STMDB r9!, {r0,r1,r2}

Base 레지스터(r9) = 0x100C



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

(Decrement Before)

① 어드레스를 0x1000로 감소

② 0x1000에 r0 저장 후 어드레스 증가

③ 0x1004에 r1 저장 후 어드레스 증가

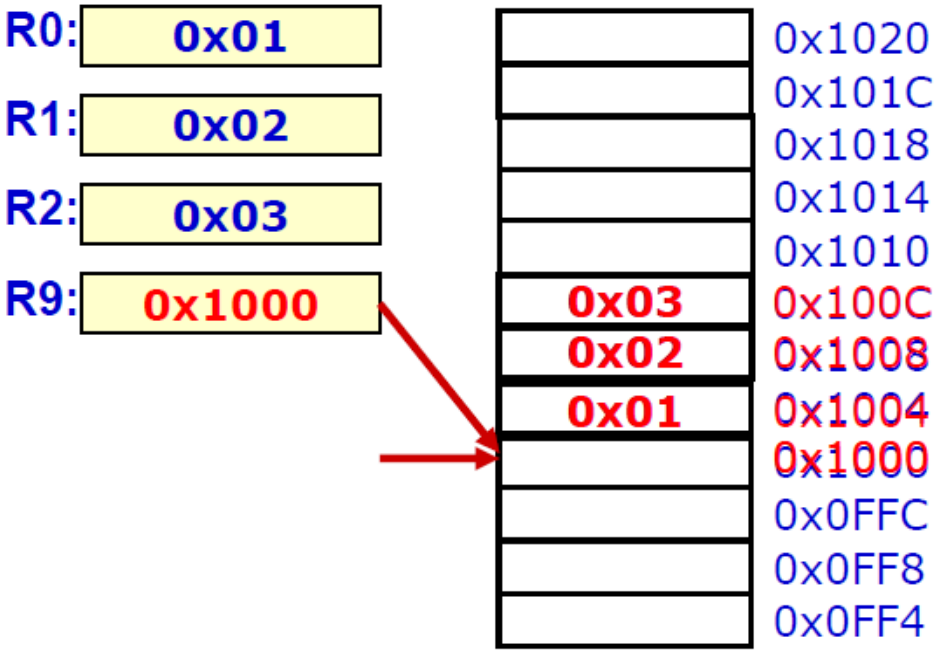
④ 0x1008에 r2 저장 후 어드레스 증가

⑤ {!}, auto-update 옵션이 있으면 r9 값을 0x1000로 변경

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

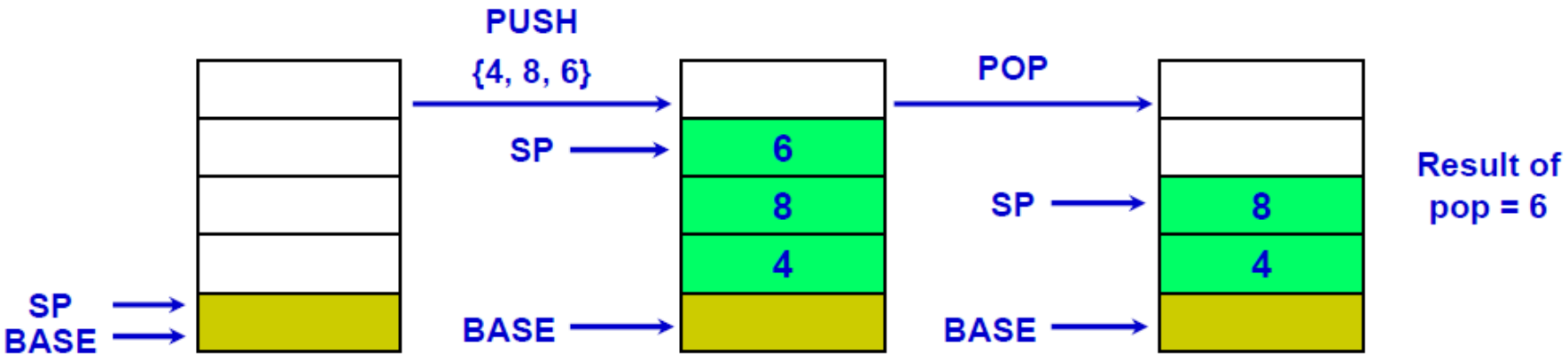
③ 어드레스 증가 후 0x1008에 r1 저장

④ 어드레스 증가 후 0x100C에 r2 저장

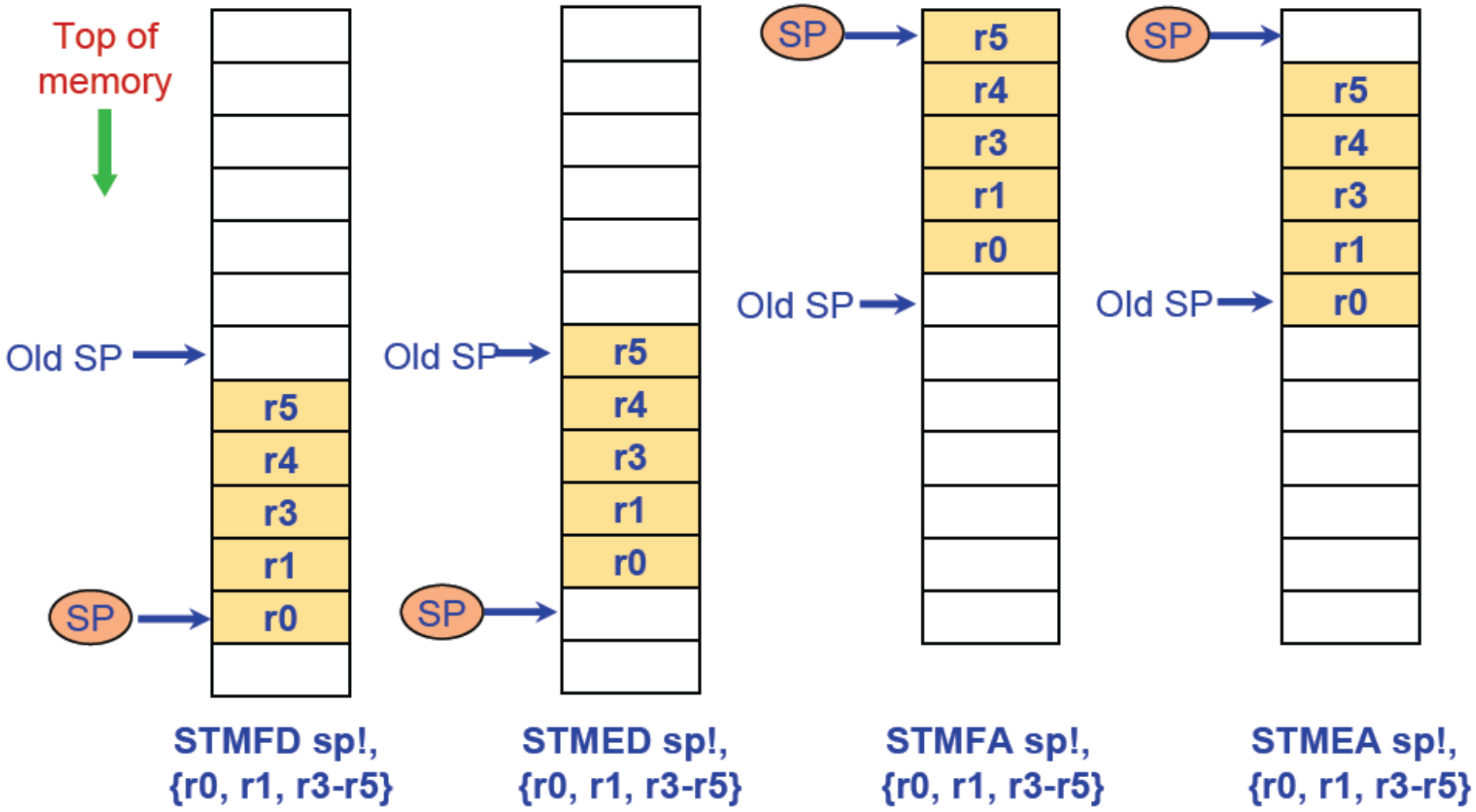
⑤ {!}, auto-update 옵션이 있으면 r9 값을 0x1000로 변경

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



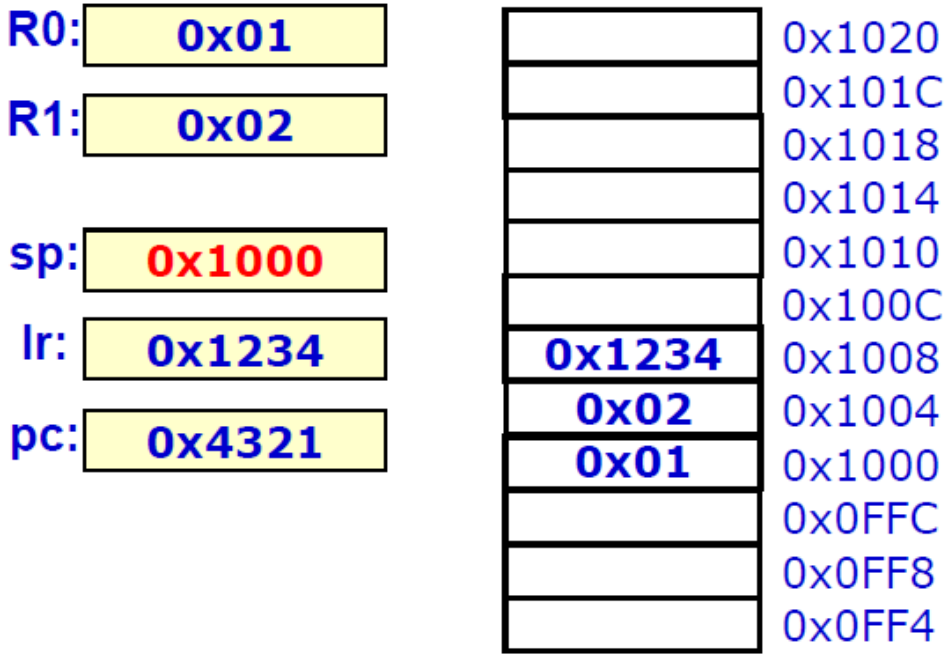
스택 type에 따른 포인트 변화



Stack의 Push 동작

STMFD sp!, {r0-r1,lr}

Base 레지스터(sp) = 0x100C

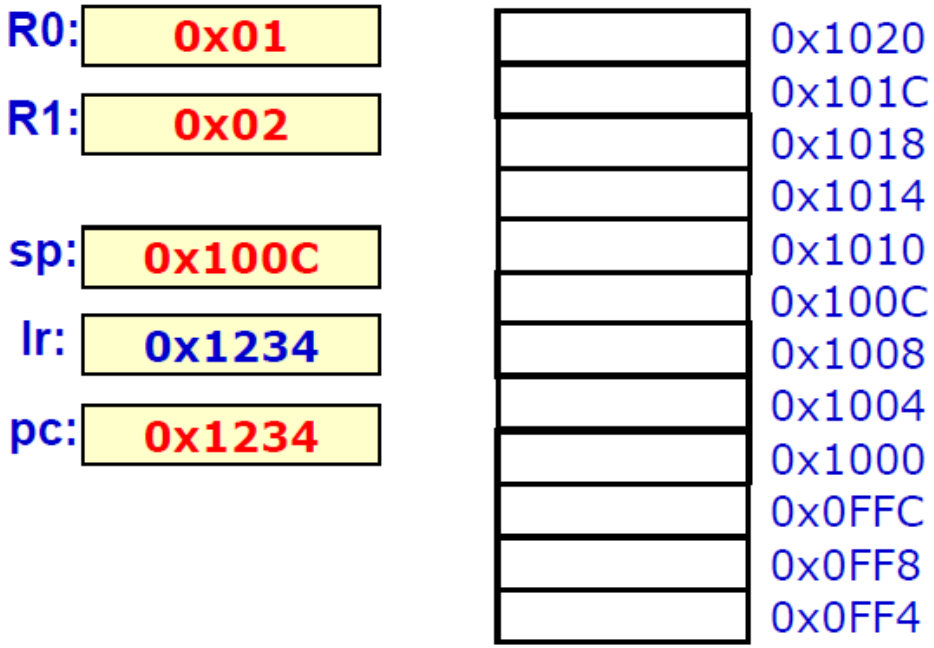


- **Stack에 context 정보 저장**
  1. 어드레스를 0x1000로 감소
  2. 레지스터 값 저장 후 어드레스 증가
  3. 링크 레지스터 저장 후 어드레스 증가
  4. Stack의 위치를 0x1000로 변경

Stack의 Pop 동작

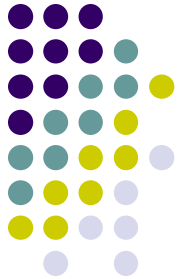
LDMFD sp!, {r0-r1,pc}

Base 레지스터(sp) = 0x1000



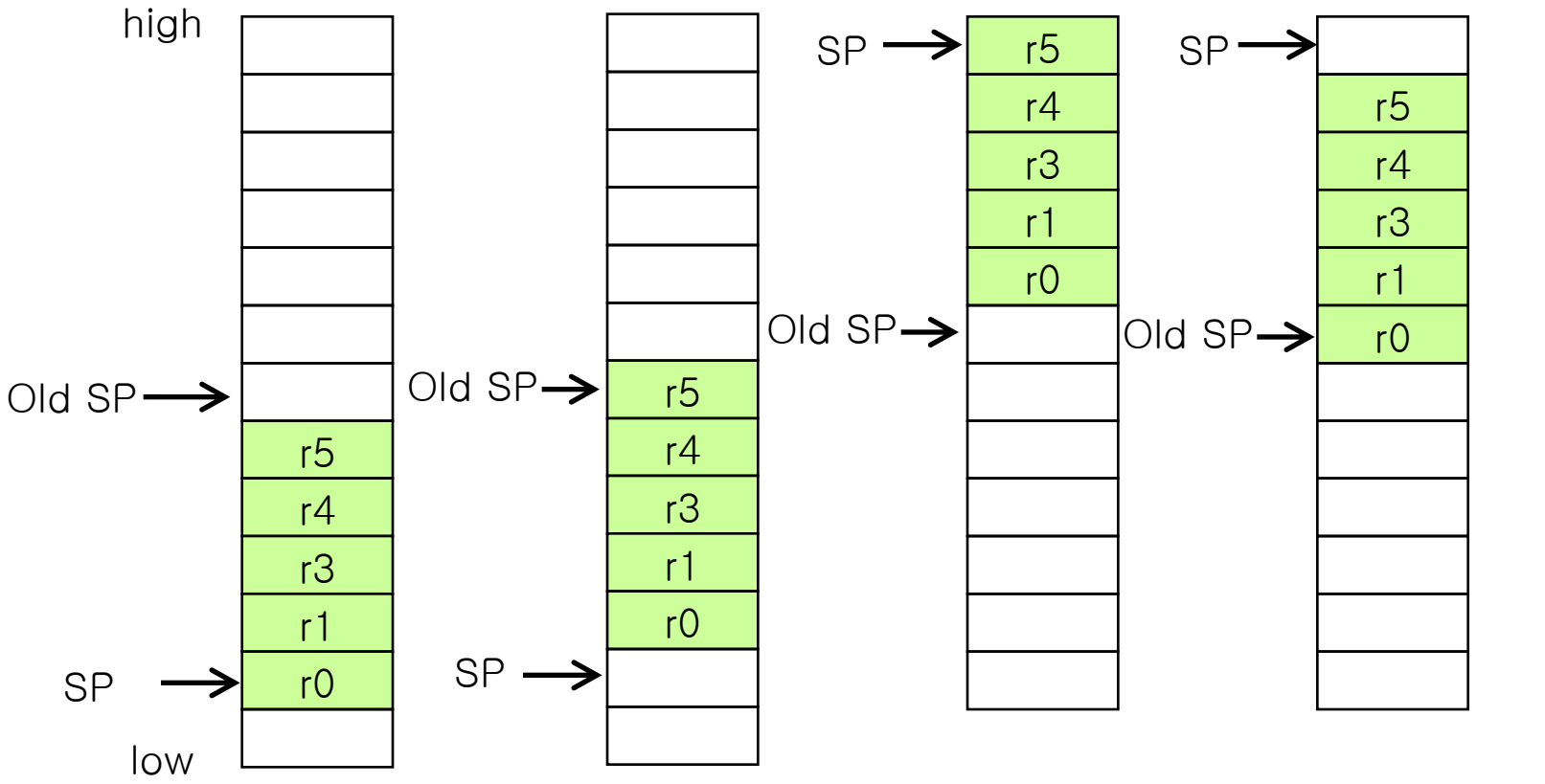
- **Stack에서 context 정보를 읽는다**
  1. 레지스터 값을 읽은 후 어드레스 증가
  2. 링크 레지스터(lr) 값을 읽어 프로그램 카운터(pc)에 저장
  3. Stack의 위치를 0x100C로 변경

스택

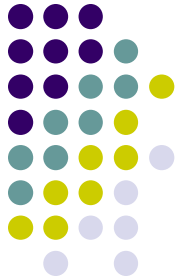


- 어떤 종류의 스택을 사용하느냐에 따라 스택 명령어의 접미사가 달라짐
  - STMFD/LDMFD: Full Descending stack
    - 참고: ARM 컴파일러가 사용하는 스택
  - STMFA/LDMFA: Full Ascending stack.
  - STMED/LDMED: Empty Descending stack
  - STMEA/LDMEA: Empty Ascending stack

스택 사용 보기


high
Old SP
r5
r4
r3
r1
SP
r0
low
STMFD sp!,
{r0, r1, r3-r5}
Old SP
r5
r4
r3
r1
SP
r0
STMED sp!,
{r0, r1, r3-r5}
SP
r5
r4
r3
r1
r0
Old SP
STMFA sp!,
{r0, r1, r3-r5}
SP
r5
r4
r3
r1
r0
Old SP
STMEA sp!,
{r0, r1, r3-r5}

스택과 서브루틴



- 스택의 용도 중 하나는 서브루틴 내에 선언된 지역 변수에 대한 일시적인 저장소를 제공하는 것
- 서브루틴 call 시 현재 사용하는 레지스터를 스택에 push하고, 서브루틴 return 시 스택의 pop을 통해 원래의 레지스터 값으로 환원시킴

```
STMFD sp!, {r0-r12, lr}      ; push
                .........
                .........
LDMFD sp!, {r0-r12, pc}      ; pop
```

스택과 서브루틴

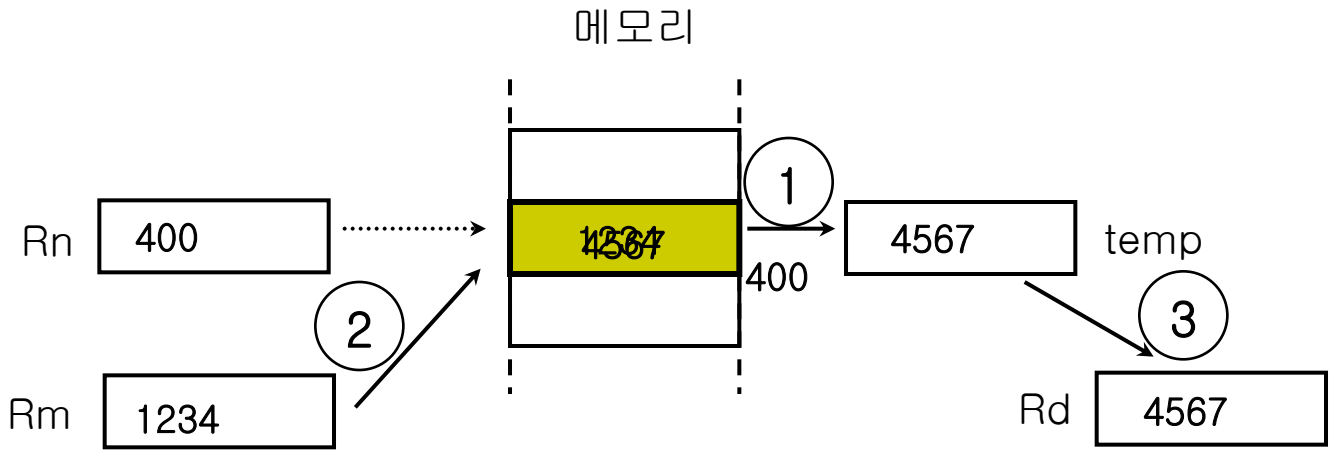
- 스택의 용도 중 하나는 서브루틴을 위한 일시적인 레지스터 저장소를 제공하는 것.
- 서브루틴에서 사용되는 데이터를 스택에 **push**하고, **caller** 함수로 **return** 하기 전에 **pop** 을 통해 원래의 정보로 환원시키는 데 사용:

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

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

Swap 명령어 1

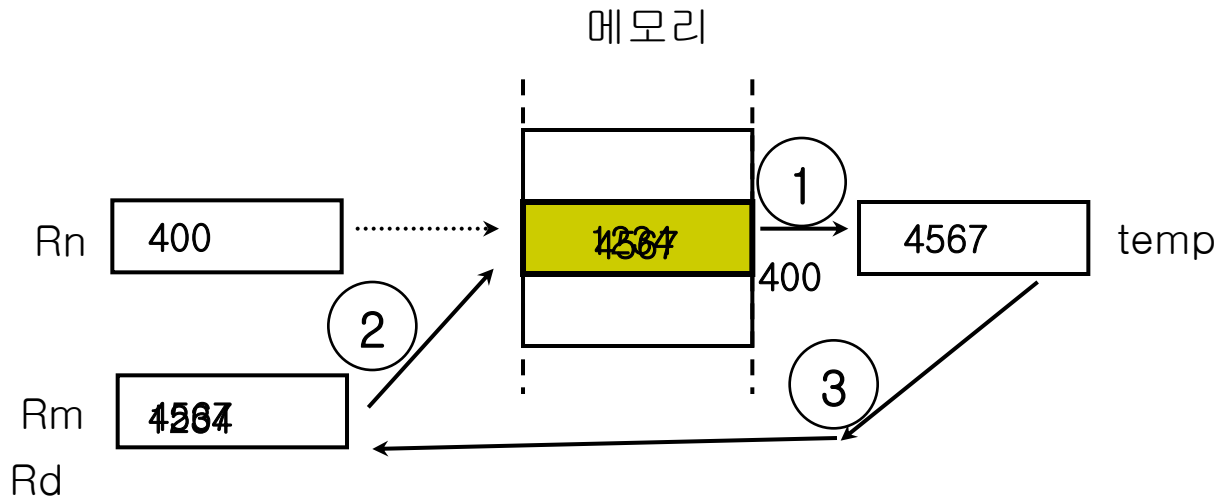
- 메모리 읽기와 쓰기를 atomic하게 수행하는 명령어
  - SWP {<cond>} {B} Rd, Rm, [Rn]



  - 보통 Rm과 Rd을 같게 하여 Rm과 mem[Rn]을 swap함

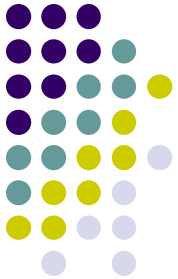
Swap 명령어 2

- 메모리 읽기와 쓰기를 atomic하게 수행하는 명령어
  - SWP {<cond>} {B} Rm, Rm, [Rn]



  - 보통 Rm과 Rd을 같게 하여 Rm과 mem[Rn]을 swap함

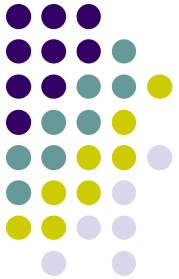
Software Interrupt (SWI)



- svc 예외를 발생시킴
  - 예외 발생시 처리하는 과정이 이루어짐
  - 0x00000008 번지로 jump하게 됨
- System call을 할 때 사용

```
MOV     r0, #'A'          ; r0= 'A'
SWI     SWI_WriteC
```

Branch 명령어



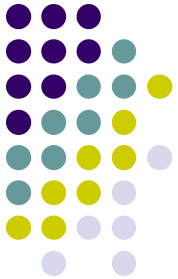
- Branch

  B{<cond>} label

- Branch with Link

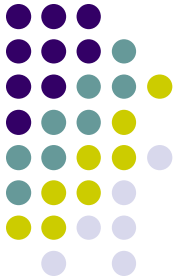
  BL{<cond>} sub_routine_label

상태 레지스터 접근 명령어



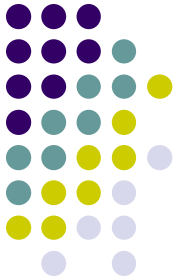
- CPSR/SPSR 와 범용 레지스터 간의 데이터 이동 가능
  - MRS : 범용 레지스터 <- Status 레지스터
  - MSR : Status 레지스터 <- 범용 레지스터 혹은 Immediate 값
- 명령어 종류
  - MRS {<cond>} Rd,<psr> ; Rd = <psr>
  - MSR {<cond>} <psr>,Rm ; <psr> = Rm
  - MSR {<cond>} <psr>, #Immediate ; <psr> = #Immediate

명령어 요약 – 1/4



Mnemonic	Instruction	Action
ADC	Add with carry	Rd: = Rn + Op2 + C
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 AND NOT Op2) OR (op2 AND NOT Rn)

명령어 요약 – 2/4

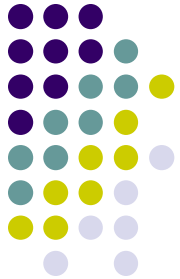


Mnemonic	Instruction	Action
LDC	Load coprocessor from memory	Coprocessor load
LDM	Load multiple registers	Stack manipulation (Pop)
LDR	Load register from memory	Rd: = (address)
MCR	Move CPU register to coprocessor register	cRn: = rRn {<op>cRm}
MLA	Multiply Accumulate	Rd: = (Rm * Rs) + Rn
MOV	Move register 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 요약 – 3/4

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

명령어 요약 – 4/4



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 EOR Op2
TST	Test bits	CPSR flags: = Rn AND Op2