Computer Architecture

Instruction: Language of the Computer
Part 2. ARM assembly programming

Kyusik Chung
kchung@ssu.ac.kr

2장 Contents

Part 1: ARM 명령어 기초

- **ARM** 명령어 종류
- **ARM** 내부구조
- 숫자표현
- **CPU** 연산후 상태 비트 계산

Part 2:ARM assembly programming

- **ARM** 명령어 사용법 소개

Part3: 코드 최적화 및 **ARM assembly program** 예제

- 간단한 코드 최적화
- 함수호출
- **Sorting** 예제

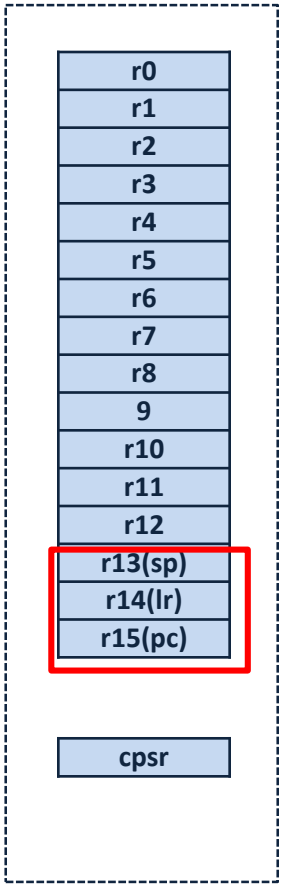
2장 Part 2 목표 - ARM 명령어 사용법 이해

ARM 어셈블리 언어

종류	명령어	예	의미	비고
산술	add	ADD r1,r2,r3	r1 = r2 + r3	레지스터 피연산자 3개
	subtract	SUB r1,r2,r3	r1 = r2 - r3	레지스터 피연산자 3개
데이터 전송	load register	LDR r1, [r2,#20]	r1 = Memory[r2 + 20]	워드를 메모리에서 레지스터로
	store register	STR r1, [r2,#20]	Memory[r2 + 20] = r1	워드를 레지스터에서 메모리로
	load register halfword	LDRH r1, [r2,#20]	r1 = Memory[r2 + 20]	하프워드를 메모리에서 레지스터로
	load register halfword signed	LDRHS r1, [r2,#20]	r1 = Memory[r2 + 20]	하프워드를 메모리에서 레지스터로
	store register halfword	STRH r1, [r2,#20]	Memory[r2 + 20] = r1	하프워드를 레지스터에서 메모리로
	load register byte	LDRB r1, [r2,#20]	r1 = Memory[r2 + 20]	바이트를 메모리에서 레지스터로
	load register byte signed	LDRBS r1, [r2,#20]	r1 = Memory[r2 + 20]	바이트를 메모리에서 레지스터로
	store register byte	STRB r1, [r2,#20]	Memory[r2 + 20] = r1	바이트를 레지스터에서 메모리로
	swap	SWP r1, [r2,#20]	r1 = Memory[r2 + 20], Memory[r2 + 20] = r1	레지스터와 메모리 간의 원자적 교환
	mov	MOV r1, r2	r1 = r2	값을 레지스터로 복사
논리	and	AND r1, r2, r3	r1 = r2 & r3	레지스터 피연산자 3개; 비트 대 비트 AND
	or	ORR r1, r2, r3	r1 = r2 | r3	레지스터 피연산자 3개; 비트 대 비트 OR
	not	MVN r1, r2	r1 = ~ r2	레지스터 피연산자 3개; 비트 대 비트 NOT
	logical logical shift left (optional operation)	LSL r1, r2, #10	r1 = r2 << 10	상수만큼 좌측 자리이동
	logical shift right (optional operation)	LSR r1, r2, #10	r1 = r2 >> 10	상수만큼 우측 자리이동
조건부 분기	compare	CMP r1, r2	cond. flag = r1 - r2	조건부 분기를 위한 비교
	branch on EQ, NE, LT, LE, GT, GE, LO, LS, HI, HS, VS, VC, MI, PL	BEQ 25	if (r1 == r2) go to PC + 8 + 100	조건 테스트; PC-상대 주소
무조건 분기	branch (always)	B 2500	go to PC + 8 + 10000	분기
	branch and link	BL 2500	r14 = PC + 4; go to PC + 8 + 10000	프로시저 호출용

P10, 11에서 이 명령어들 사용법 설명함

[보충설명] 특수목적 레지스터



- ARM 레지스터 R0-R15 중 R13-R15는 특수목적 레지스터, R0-R12는 연산시 사용하는 범용레지스터(예, ADD R3, R1, R2 에서 R1, R2, R3자리에 올 수 있슴)
  - R15(일명 PC라 부름, program counter): CPU가 현재 수행중인 명령어를 완료한 뒤 다음 수행할 명령어가 있는 메모리 주소를 저장
  - R14(일명 LR 라 부름, Link Register): subroutine(함수)를 수행한 뒤 return 할 메모리 주소를 저장
  - R13(일명 SP라 부름, Stack Pointer): 메모리 일부를 stack 영역으로 사용한다. Stack의 Top에 해당하는 메모리 주소를 저장

[보충설명] 메모리에서의 명령어 및 데이터 저장

- 32비트 ARM CPU에서 하나의 명령어를 32비트로 표현, 4 byte 차지. 메모리주소로는 4개를 차지한다. (예) 명령어가 1000번지에 있다는 얘기는 메모리 1000 번지부터 1001, 1002, 1003 번지까지 저장되어 있슴을 의미 (메모리 1번지에는 8 비트가 저장됨)
- 32비트 ARM CPU에서 하나의 word data는 32비트로 표현, 4 byte 차지. 메모리주소로는 4개를 차지한다. (예) word data가 2000번지에 있다는 얘기는 메모리 2000 번지부터 2001, 2002, 2003 번지까지 저장되어 있슴을 의미 (메모리 1번지에는 8 비트가 저장됨)
- 32비트 ARM CPU에서 하나의 half word data는 16비트로 표현, 2 byte 차지. 메모리주소로는 2개를 차지한다. (예) half data가 2000번지에 있다는 얘기는 메모리 2000 번지부터 2001 번지까지 저장되어 있슴을 의미
- 32비트 ARM CPU에서 하나의 byte data는 8비트로 표현, 1 byte 차지. 메모리주소로는 1개를 차지한다. (예) byte data가 2000번지에 있다는 얘기는 메모리 2000 번지에만 저장되어 있슴을 의미

메모리에 저장된 명령어 예제

메모리 1000번지에 ADD r1, r2, r3
메모리 1004번지에 LDR r1, [r2]가 저장된 경우

메모리 번지수	메모리 값	
1000	03	ADD r1, r2, r3 명령어임 - 32비트로 되어 있는데 16 진수로 이 명령어를 표현하면 0x E0421003 - Little endian으로 표현한 방식임
1001	10	
1002	42	
1003	E0	
1004	00	LDR r1, [r2] 명령어임 - 32비트로 되어 있는데 16 진수로 이 명령어를 표현하면 0x E5821000 - Little endian으로 표현한 방식임
1005	10	
1006	82	
1007	E5	

메모리에 저장된 data 예제

메모리 2000번지에 data 0x04030201
메모리 2004번지에 data 0x08070605가 저장된 경우

메모리 번지수	메모리 값	
2000	01	- 32비트 데이터인데 16 진수로 표현하면 0x 04030201 - Little endian으로 표현한 방식임 (데이터중 최하위 바이트를 가장 낮은 메모리 주소에 저장)
2001	02	
2002	03	
2003	04	
2004	05 (08)	- 32비트 데이터인데 16 진수로 표현하면 0x 08070605 - Little endian으로 표현한 방식임 - Big endian으로 표현하면 괄호안 파란색
2005	06 (07)	
2006	07 (06)	
2007	08 (05)	

[보충설명] 명령어 수행 4단계

- 4장 진입부에서 자세한 설명이 나옴
- 하나의 명령어 수행 4 단계. 각 단계는CPU 1 clock 동안 수행. 1&2 단계는 모든 명령어에 대해 공통. 3&4 단계는 명령어에 따라 다르다
  1. 명령어 갖고 오기 (fetch): (예) 메모리 1000 번지에 저장된 명령어를 CPU로 읽어온다.
  2. 명령어 해독하기(decode): CPU 내 control unit(제어유닛)이 읽어온 명령어를 해독한다. (예) 만일 “ADD R3, R1, R2” 이라면 R1값과 R2값을 더하여 결과를 R3 저장하는 덧셈 명령어임을 알게 된다.
  3. 명령어 수행하기(execute): ALU를 이용하여 연산 또는 논리동작을 수행한다. (예) R1 값과 R2 값을 꺼내어 ALU 입력단으로 보내어 덧셈을 수행한다.
  4. 수행결과 저장하기(store): 수행결과를 레지스터 또는 메모리에 저장한다. (예) 앞에서 수행한 덧셈 결과를 R3에 저장한다.

명령어 set 1

종류	명령어	예	의미	비고
산술	add	ADD r1,r2,r3	r1 = r2 + r3	레지스터 피연산자 3개
	subtract	SUB r1,r2,r3	r1 = r2 - r3	레지스터 피연산자 3개
데이터 전송	load register	LDR r1, [r2,#20]	r1 = Memory[r2 + 20]	워드를 메모리에서 레지스터로
	store register	STR r1, [r2,#20]	Memory[r2 + 20] = r1	워드를 레지스터에서 메모리로
	load register halfword	LDRH r1, [r2,#20]	r1 = Memory[r2 + 20]	하프워드를 메모리에서 레지스터로
	load register halfword signed	LDRHS r1, [r2,#20]	r1 = Memory[r2 + 20]	하프워드를 메모리에서 레지스터로
	store register halfword	STRH r1, [r2,#20]	Memory[r2 + 20] = r1	하프워드를 레지스터에서 메모리로
	load register byte	LDRB r1, [r2,#20]	r1 = Memory[r2 + 20]	바이트를 메모리에서 레지스터로
	load register byte signed	LDRBS r1, [r2,#20]	r1 = Memory[r2 + 20]	바이트를 메모리에서 레지스터로
	store register byte	STRB r1, [r2,#20]	Memory[r2 + 20] = r1	바이트를 레지스터에서 메모리로
	swap	SWP r1, [r2,#20]	r1 = Memory[r2 + 20], Memory[r2 + 20] = r1	레지스터와 메모리 간의 원자적 교환
	mov	MOV r1, r2	r1 = r2	값을 레지스터로 복사

명령어 set 2

논리	and	AND r1, r2, r3	r1 = r2 & r3	레지스터 피연산자 3개; 비트 대 비트 AND
	or	ORR r1, r2, r3	r1 = r2 | r3	레지스터 피연산자 3개; 비트 대 비트 OR
	not	MVN r1, r2	r1 = ~ r2	레지스터 피연산자 3개; 비트 대 비트 NOT
	logical logical shift left (optional operation)	LSL r1, r2, #10	r1 = r2 << 10	상수만큼 좌측 자리이동
	logical shift right (optional operation)	LSR r1, r2, #10	r1 = r2 >> 10	상수만큼 우측 자리이동
조건부 분기	compare	CMP r1, r2	cond. flag = r1 - r2	조건부 분기를 위한 비교
	branch on EQ, NE, LT, LE, GT, GE, LO, LS, HI, HS, VS, VC, MI, PL	BEQ 25	if (r1 == r2) go to PC + 8 + 100	조건 테스트; PC-상대 주소
무조건 분기	branch (always)	B 2500	go to PC + 8 + 10000	분기
	branch and link	BL 2500	r14 = PC + 4; go to PC + 8 + 10000	프로시저 호출용

= 25*4

= 2500*4

조건부 분기에서 조건 종류

Code	*Meaning*	*Requires*
EQ	Equal	Z = 1
NE	Not equal	Z = 0
GE	Signed ≥ ("Greater than or Equal")	N = V
LT	Signed < ("Less Than")	N ≠ V
GT	Signed > ("Greater Than")	Z = 0 && N = V
LE	Signed ≤ ("Less than or Equal")	Z = 1 || N ≠ V
HS (CS)	Unsigned ≥ ("Higher or Same") or Carry Set	C = 1
LO (CC)	Unsigned < ("Lower") or Carry Clear	C = 0
HI	Unsigned > ("Higher")	C = 1 && Z = 0
LS	Unsigned ≤ ("Lower or Same")	C = 0 || Z = 1
MI	Minus/negative	N = 1
PL	Plus - positive or zero (non-negative)	N = 0
VS	Overflow	V = 1
VC	No overflow	V = 0
AL	Always (unconditional)	(Rarely used)

(예) ch2-part1 노트 P49에서
BHI 2000, BGT 2000

조건부 분기에서 조건 종류

Code	*Meaning*	*Requires*
EQ	Equal	EQ and NE are same for signed and unsigned.
NE	Not equal	
GE	Signed ≥ ("Greater than or Equal")	N = V
LT	Signed < ("Less Than")	These are used for signed comparisons
GT	Signed > ("Greater Than")	
LE	Signed ≤ ("Less than or Equal")	Z = 1 || N ≠ V
HS (CS)	Unsigned ≥ ("Higher or Same") or Carry Set	C = 1
LO (CC)	Unsigned < ("Lower") or Carry Clear	These are used for unsigned comparisons
HI	Unsigned > ("Higher")	
LS	Unsigned ≤ ("Lower or Same")	C = 0 || Z = 1
MI	Minus/negative	N = 1
PL	Plus - positive or zero (non-negative)	N = 0
VS	Overflow	V = 1
VC	No overflow	V = 0
AL	Always (unconditional)	Never used. Included for completeness.

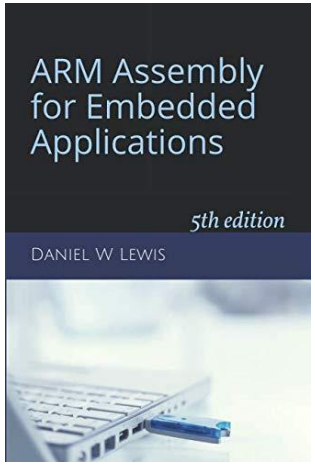
[보충설명] 명령어들 수행순서

- C 코드들 수행순서 분석할 경우 코드 line #를 갖고 설명한다. 어셈블리 코드들에서도 마찬가지로 코드 line #를 갖고 설명할 수 있다. 기계어코드로 수행순서 분석할 경우 명령어들이 들어간 메모리 주소를 갖고 설명한다. (예) 1000번지, 1004번지,...
- 수행과정 유형 분류
  1. 순차적인 수행 (P9, 10에서 산술, 데이터 전송, 논리 명령어 유형)
     - (예) 1000번지 명령어 (ADD R3, R1, R2)수행
     - 그 다음에는 바로 다음 1004번지 명령어(SUB R5, R3, R4) 수행
  2. Jump (branch) 발생하는 경우 (P10에서 조건부 분기, 무조건 분기에서 branch 유형)
     - (예) 1000 번지 명령어 (B 2500) 수행
     - 그 다음에는 11008번지 (PC+8+2500 *4) 명령어 (ORR R3, R1, R2)수행
  3. Call 함수를 수행하는 경우 (P10에서 무조건 분기에서 branch and link 유형)
     - (예) 1000번지 명령어 (BL 2500) 수행
     - 그 다음에는 11008번지 (PC+8+2500 *4) 명령어 (ORR R3, R1, R2)수행
     - 11008 번지 명령어부터 11040 번지 명령어까지 순차적으로 수행한 후 (함수종료) 그 다음엔 되돌아가서 (return) 1004번지 명령어(SUB R5, R3, R4) 수행한다.

2장-Part2 Contents

Part 2: ARM assembly programming

- **Adapted from the lecture notes of Prof. D. W. Lewis (www.gngr.scu.edu/~dlewis/book3/)**
- 이 내용은 **ARM Cortex-M4F (32 bit CPU)**를 위한 **assembly programming**을 다룬다.



Chapter 1

Introduction

WHY YOU SHOULD LEARN ASSEMBLY

- **To create system software for new processors:**
  - Compilers, Optimizers, Assemblers, Linkers, Device Drivers.
- **To better understand the limitations of fundamental data types:**
  - Precision, Range, Overflow, Representation Error
- **To better understand high level languages:**
  - Visualizing pass-by-value, pass-by-reference, recursion, pointers, and writing code to take advantage of memory hierarchies based on locality of reference

WHEN ASSEMBLY IS NEEDED

- **To optimize performance:**
  - Implementing code fragments that account for most of the execution time.
  - Using fixed-point arithmetic when the processor has no floating-point instructions
- **For code not easily implemented in a high-level language:**
  - Reversing the order of bits and bytes
- **Low-level access to hardware resources:**
  - Device drivers, interrupt routines.
- **Reverse engineering to understand and eradicate malware (악성 SW 를 근절하다) – SW 보안**

WHAT ASSEMBLY PROGRAM LOOKS LIKE

```
.syntax     unified
.cpu        cortex-m4
.text
.thumb_func
.align      2

// int64_t Mult32x32(int32_t mpier, int32_t mcand)

// Input Parameters:    R0 = mpier, R1 = mcand (32-bits each)
// Return Value:        R1.R0 = 64-bit product

.global     Mult32x32

Mult32x32:  SMULL       R0,R1,R0,R1 // R1.R0 <-- R0*R1
            BX          LR          // Return

.end
```

Assembler directives

Comment lines

Executable Instructions

Label

Instruction Mnemonic ("OpCode")

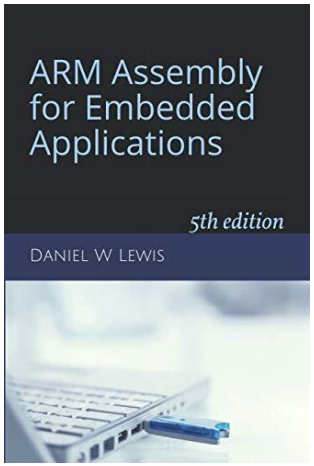
Instruction Operands

Comments

IDENTIFIER CONVENTIONS

- **Variable Names**
  - All lowercase
  - Append digits to indicate size in bits
  - Prefix with ‘s’ for signed, ‘u’ for unsigned

  (ex. int32_t, uint16_t)
- **Function Names**
  - Capitalize 1st letter of each word
- **Macros and Symbolic Constants**
  - All Caps



Chapter 3

Writing Functions in Assembly

Approach

- **Most of a program is written in a high-level language (HLL) such as C**
  - easier to implement
  - easier to understand
  - easier to maintain

- **Only a few functions are written in assembly**
  - to improve performance, or
  - because it's difficult or impossible to do in a HLL.

Sample Program

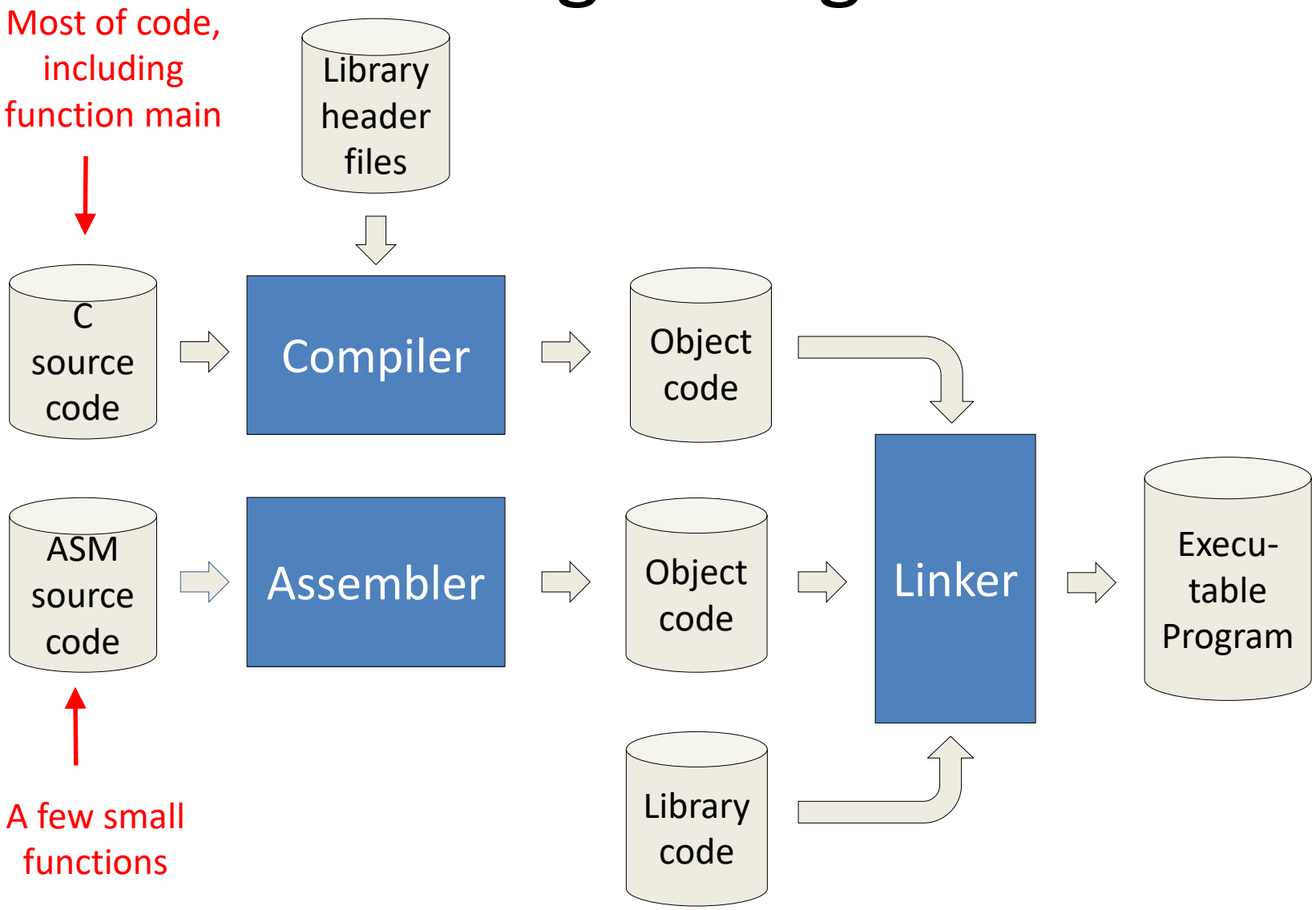
Main Program written in C

```c
#include <stdio.h>
#include <stdint.h>
#include "library.h"

extern uint32_t  Add1(uint32_t x) ;

int main(void)
   {
   uint32_t strt, stop, before, after ;

   InitializeHardware(HEADER, PROJECT_NAME) ;

   before = 0 ;
   while (1)  // Never exit
      {
      strt  = GetClockCycleCount() ;
      after = Add1(before) ;
      stop  = GetClockCycleCount() ;

      printf("Before = %u\n", (unsigned) before) ;
      printf(" After = %u\n", (unsigned) after) ;
      printf("CPU Clock Cycles: %u\n", (unsigned) (stop - strt)) ;

      WaitForPushButton() ;
      ClearDisplay() ;
      before = after ;
      }
   }
```

Function written in Assembly

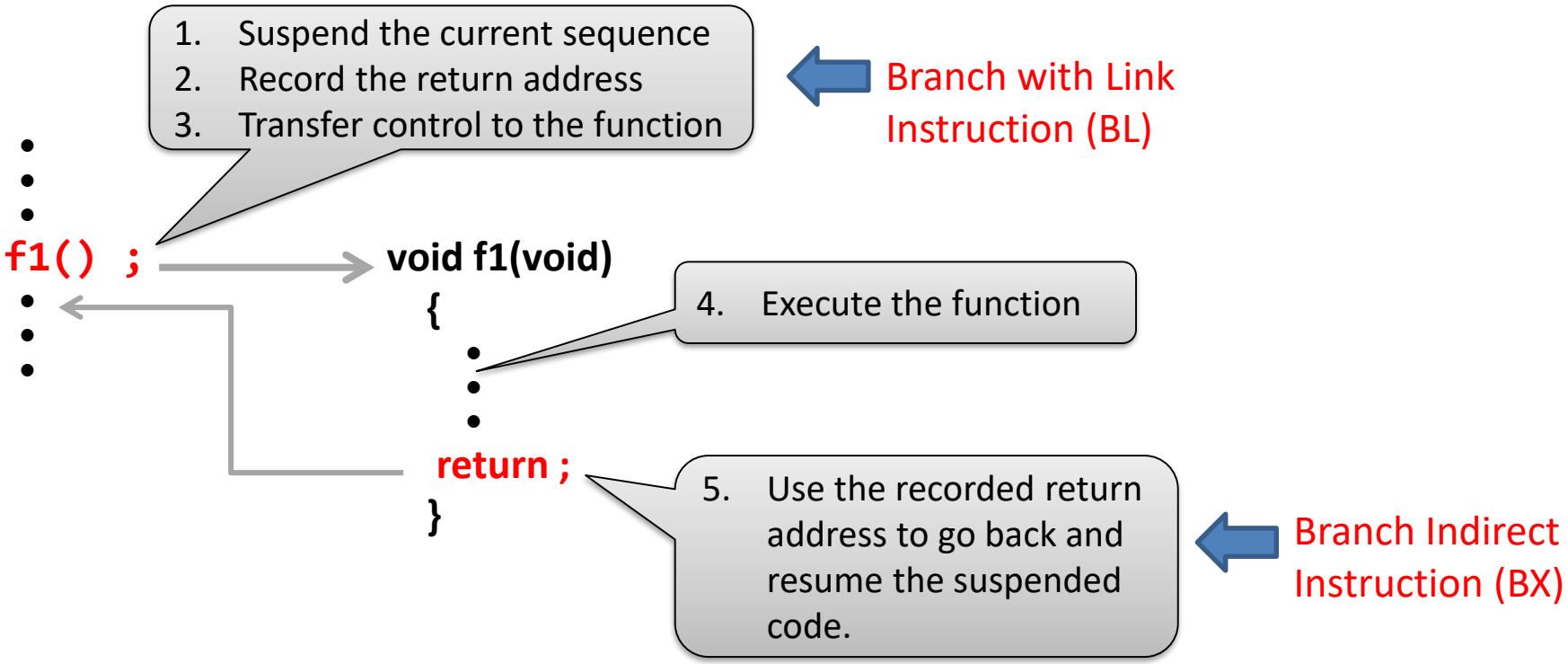
```asm
        .syntax   unified
        .cpu      cortex-m4
        .text
        .thumb_func
        .align    2

// uint32_t  Add1(uint32_t  x) ;

        .global   Add1
Add1:   ADD       R0,R0,1              // Return x + 1
        BX        LR

        .end
```

Creating a Program



FUNCTION CALL AND RETURN

Simple Call-Return in C

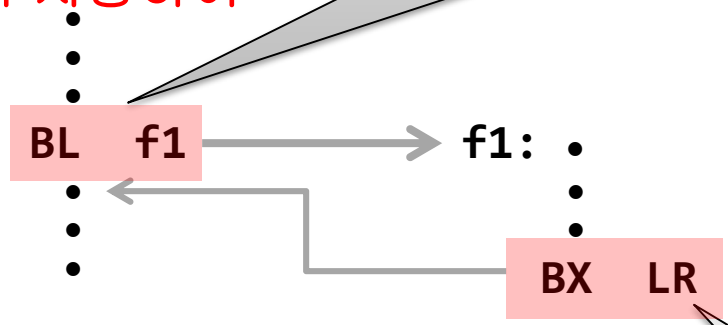


FUNCTION CALL AND RETURN

Simple Call-Return in Assembly

BL f1 명령어 바로 앞 명령어들에서, f1에 전달할 parameter를 R0, R1, R2, R3에 저장하여 전달

The "Branch with Link" instruction (BL) saves the address of the instruction immediately following it (the return address) in the Link Register (LR).



Consider: There is only one Link Register. What if inside function f1 there is a call to another function?

BX LR 명령어 바로 앞 명령어들에서, return 할 값들을 R0, R1에 저장하여 return

The "Branch Indirect" instruction (BX) copies the return address from LR into PC, thus transferring control back to where the function had been called.

FUNCTION CALL AND RETURN

ARM instructions used to call and return from functions

Instruction	***Format***	***Operation***
Branch with Link	BL *label*	***Function Call:*** LR ← return address, PC ← address of label
Branch Indirect	BX LR	***Function Return:*** PC ← LR

LR (aka R14) and PC (aka R15) are two of the 16 registers inside the CPU.
LR is the "Link Register" – used w/function calls to hold the return address
PC is the "Program Counter" – holds the address of the next instruction.

ARM PROCEDURE CALL STANDARD

Registers used as input parameters

Register	*AAPCS Name*	*Usage*
R0	A1	1st parameter
R1	A2	2nd parameter
R2	A3	3rd parameter
R3	A4	4th parameter

One register is used for each 8, 16 or 32-bit parameter.
A sequential register pair is used for each 64-bit parameter.

PARAMETER PASSING

void foo(int8_t, int32_t, int64_t) ;

C Function Call	*Compiler Output*

`int8_t x8 ;`
`int32_t y32 ;`
`int64_t z64 ;`
⋮
`foo(x8, y32, z64) ;`
⋮
`foo(5, -10, 20) ;`
⋮
	`LDR R0,=x8 // R0 <-- &x8`
	`LDRSB R0,[R0] // R0 <-- x8`
	`LDR R1,=y32 // R1 <-- &y32`
	`LDR R1,[R1] // R1 <-- y32`
	`LDR R2,=z64 // R2 <-- &z64`
	`LDRD R2,R3,[R2] // R3.R2 <-- z64`
	`BL foo`
	⋮
	`LDR`
	`LDR R1,=-10 // R1 <-- -10`
	`LDR R2,=20 // R3.R2 <-- 20`
	`LDR R3,=0`
	`BL foo`

Inside foo, you must use the register copies of the actual arguments.

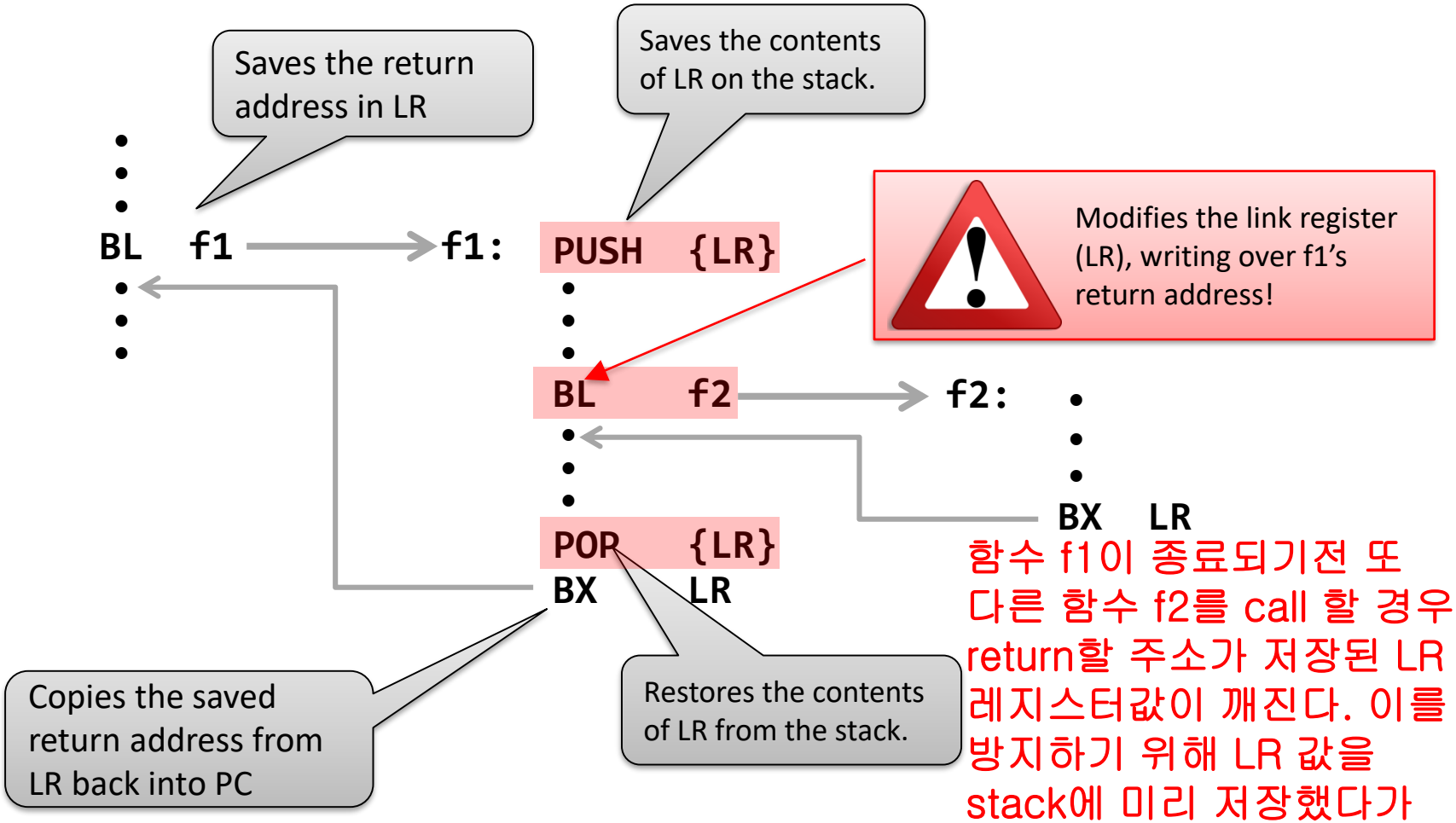
ARM PROCEDURE CALL STANDARD

Registers used to return function result

Register	*AAPCS Name*	*Usage*
R0	A1	8, 16 or 32-bit result, or the least-significant half of a 64-bit result
R1	A2	The most-significant half of a 64-bit result

FUNCTION CALL AND RETURN

Nested Call-Return in Assembly

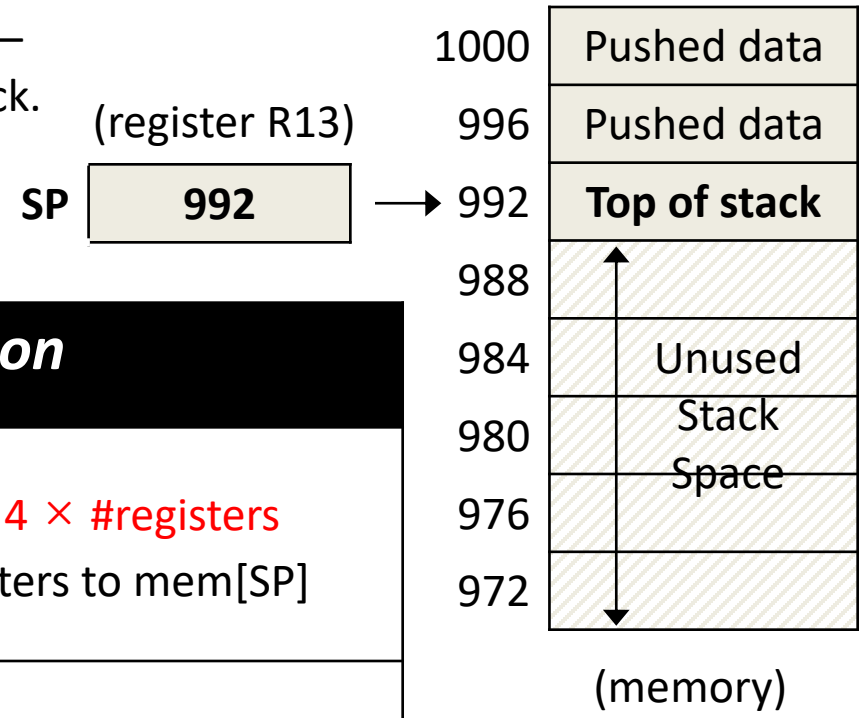


함수 f1이 종료되기전 또 다른 함수 f2를 call 할 경우 return할 주소가 저장된 LR 레지스터값이 깨진다. 이를 방지하기 위해 LR 값을 stack에 미리 저장했다가 필요할 때복구하여 사용한다.

FUNCTION CALL AND RETURN

ARM instructions used in functions that ***call other functions***

SP is one of the 16 CPU registers (R13) – holds the address of the top of the stack.



Instruction	*Format*	*Operation*
Push registers onto stack	PUSH *register list*	SP ← SP – 4 × #registers Copy registers to mem[SP]
Pop registers from stack	POP *register list*	Copy mem[SP] to registers, SP ← SP + 4 × #registers

"register list" format: { reg, reg, reg-reg, …}

Review Summary

- Call functions using Branch with Link (BL)
  - Saves the return address in Link Register (LR)
  - Copies the target address into Program Counter (PC)
- Return from a function using BX LR
- Writing functions that call other functions:
  - Using BL to call another function changes LR
  - Use PUSH {LR} / POP {LR} to preserve LR
- Pass parameters using registers R0-R3
  - 64-bit parameters in consecutive register pair
- Return 8, 16 and 32 bit results using R0
  - 64-bit result returned in R1.R0 register pair

REGISTER USAGE CONVENTIONS

ARM PROCEDURE CALL STANDARD

Register	*AAPCS Name*	*Usage*	*Notes*
R0	A1	Argument / result /scratch register 1	**Do not have to preserve original contents**
R1	A2	Argument / result /scratch register 2	
R2	A3	Argument / scratch register 3	
R3	A4	Argument / scratch register 4	
R4	V1	Variable register 1	**Must preserve original contents (call된 함수안에서 이 레지스터를 사용하기 원하면 함수시작할 때 해당 레지스터값을 stack에 저장했다가 함수return 직전에 레지스터 값 복구해야함)**
R5	V2	Variable register 2	
R6	V3	Variable register 3	
R7	V4	Variable register 4	
R8	V5	Variable register 5	
R9	V6	Variable register 6	
R10	V7	Variable register 7	
R11	V8	Variable register 8	
R12	IP	Intra-Procedure-call scratch register	
R13	SP	Stack Pointer	**Reserved, Do not use**
R14	LR	Link Register	
R15	PC	Program Counter	

FUNCTION CODING CONVENTIONS

Functions that modify only registers R0 - R3, R12

CASE 1

Using only R0-R3, R12
and no function call

```
f1:
        OK to
modify
        R0-R3 and
R12

        BX      LR
```

CASE 2

Using only R0-R3, R12
and calling another function

```
f2: PUSH{LR}
        OK to modify R0-R3, R12
        BL      f3
        OK to modify R0-R3, R12
    POP {PC}
```

Function f3 might modify R0 – R3, R12

FUNCTION CODING CONVENTIONS

Functions that modify for example registers R4 and R5

CASE 3

Using R4 and R5
and no function call

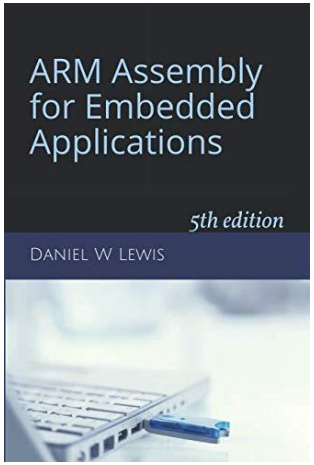
```
f1:     PUSH    {R4,R5}
        ●
        ●       OK to
modify
        ●       R0-R5
and R12
        ●
        POP     {R4,R5}
        BX      LR
```

CASE 4

Using R4 and R5
and calling another function

```
f2:     PUSH    {R4,R5,LR}
        ●
        ●       OK to modify
        ●       R0-R5 and R12
        ●
        BL
        ●
        ●       OK to modify
        ●       R0-R5 and R12
        ●
        POP     {R4,R5,PC}
```

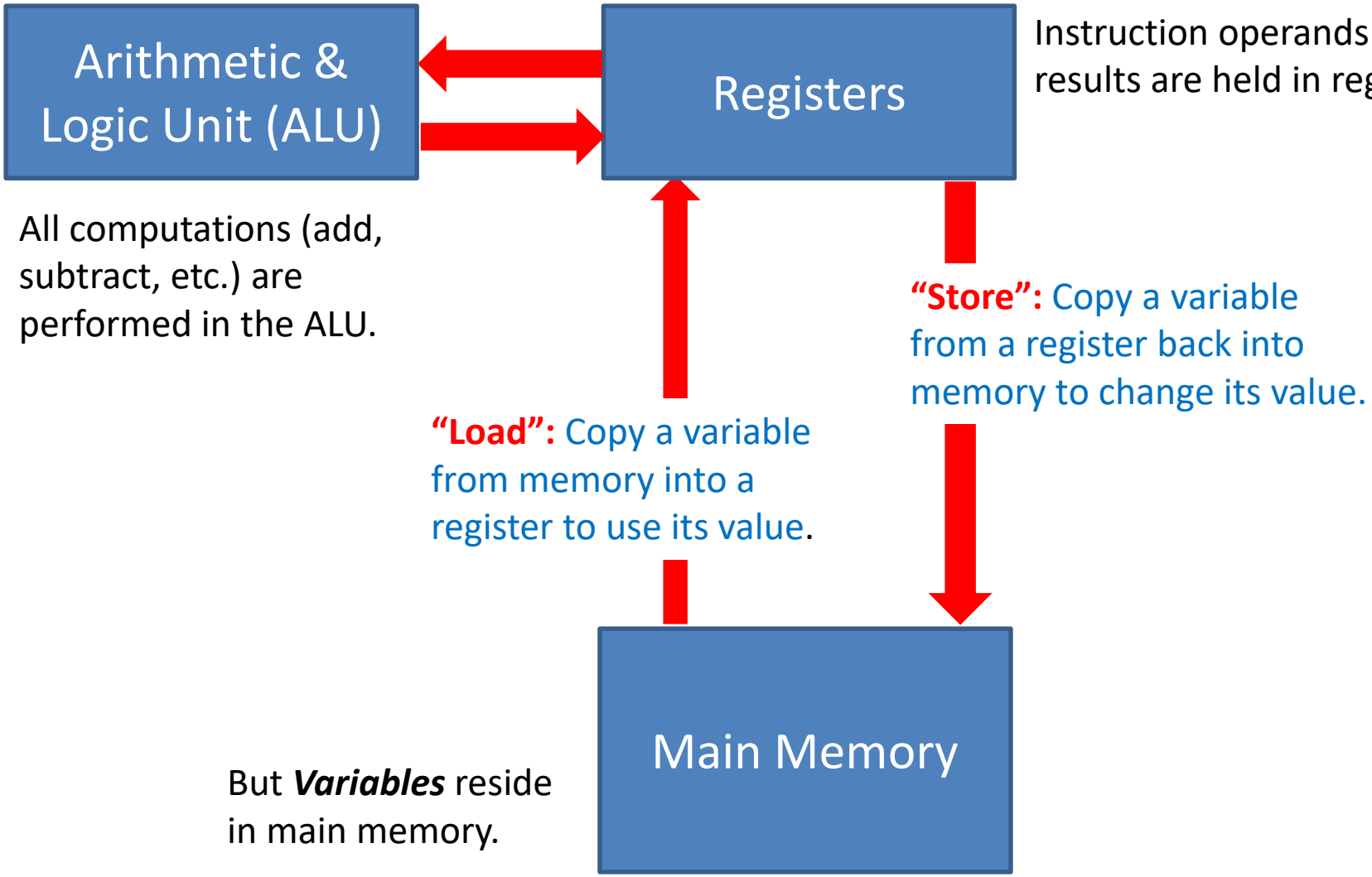
Function f3 might modify R0-R3 and R12, but preserves R4-R11



Chapter 4

Copying Data

LOAD/STORE ARCHITECTURE

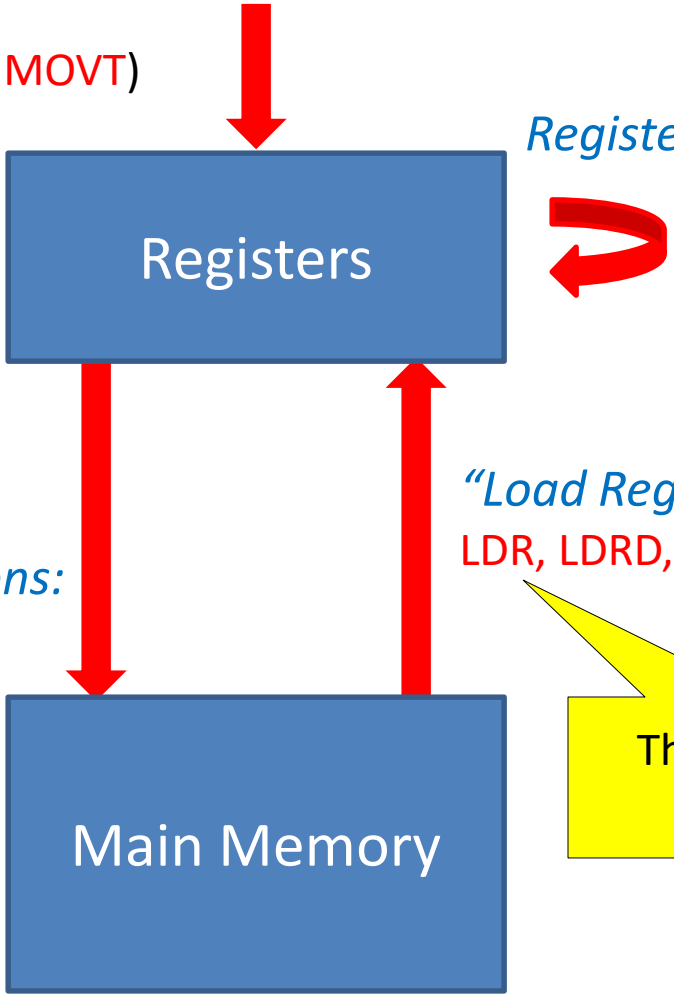


INSTRUCTIONS FOR COPYING DATA

constant

Constant to Register:
LDR* (MOV, MVN, MOVW, MOVT)

This LDR is a "pseudo" instruction



Register to Register: MOV

"Load Register" Instructions:
LDR, LDRD, LDRB, LDRH, LDRSB, LDRSH

This LDR is a real instruction

"Store Register" Instructions:
STR, STRD, STRB, STRH

REGISTER ← CONSTANT

```
MOV     R0,15        // R0 ← 15
MVN     R0,15        // R0 ← ~15 (-16)

MOV     R0,-15       // R0 ← -15
        // Assembler replaces this by MVN R0,14

MOVW    R0,1000      // R0 ← 1000
MOVT    R0,1000      // R0 ← 1000 << 16

MOVW    R0,100000 & 0xFFFF // LS 16 bits
MOVT    R0,100000 >> 16    // MS 16 bits
```

하위 8-bit data: 0-255

하위 8-bit data: 0-255

하위 16-bit data: 0-65535

상위 16-bit data: 0-65535

두개를 조합하면 임의의 32-bit 상수 생성 가능

The LDR Pseudo-Instruction

A "*pseudo-instruction*" is not a real ARM instruction. When used, the assembler replaces it with an equivalent operation using a real instruction.

Format: **LDR *Rd,=constant***

The equals sign distinguishes this pseudo-instruction from a real LDR instruction.

The pseudo-instruction is replaced by one of the following if possible, else it is replaced by a real LDR that loads the constant from memory:

Instruction Format and Width			Range
MOV	Rd,imm8	16	0-255
MOVW	Rd,imm16	32	0-65535
MVN	Rd,imm8	16	0-255

상수 크기는 8 bit, 16 bit에 국한

WRITING INTEGER CONSTANTS

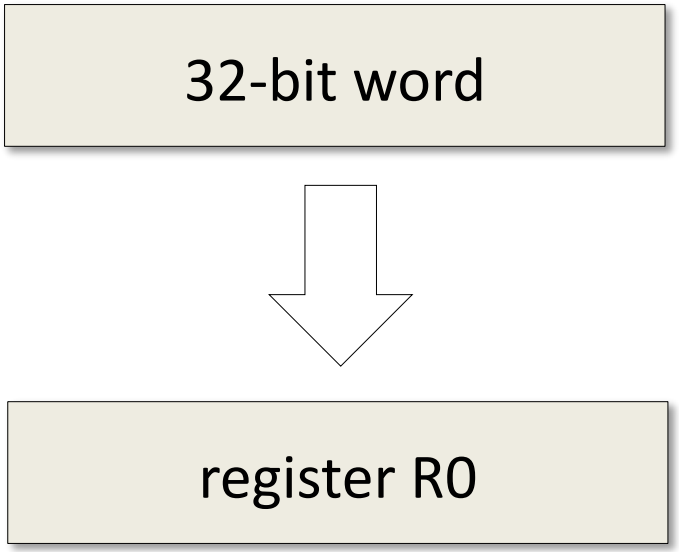
- Decimal: 123
- Binary: **0b**10110111
- Octal: **0**123
- Hexadecimal: **0x**FACE
- ASCII Character (8 bits) **'**a

C-style character constants also work ('a'), but not all escape sequences ('\0 or '\0') do. It's safer to use hex (0x00) instead.

REGISTER ← MEMORY (32-BITS)
Load Register from Word

```
LDR R0,[R1]
```

```
// Copies a 32-bit word
// from the memory location
// whose address is in R1
// into register R0.
```



Used with data of type **int32_t, uint32_t,** and **all pointers**

Copying variables < 32 bits wide

(32-bit register copy must have same value)

Unsigned	Signed (2's complement)
Zero-Extend: Add leading 0's	Sign-Extend: Replicate sign bit

4-bit example:

$1101_2 = 13_{10}$

$00\cdots001101_2 = 13_{10}$

(예) UXTB R0, R1
– R1의 하위 8bit를 복사, 나머지 bit들은 all zero
(예) UXTH R0, R1
– R1의 하위 16 bit를 복사, 나머지 bit들은 all zero

4-bit example:

$1101_2 = -3_{10}$

$11\cdots 111101_2 = -3_{10}$

Instructions that zero-extend:
UXTB, UXTH, LDRB, LDRH

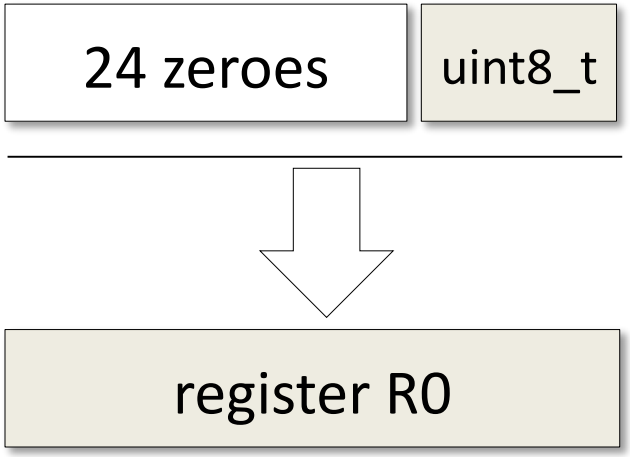
Instructions that sign-extend:
SXTB, SXTH, LDRSB, LDRSH

REGISTER ← MEMORY (8-BITS UNSIGNED)
Load Register from (Unsigned) Byte

```
LDRB  R0,[R1]
```

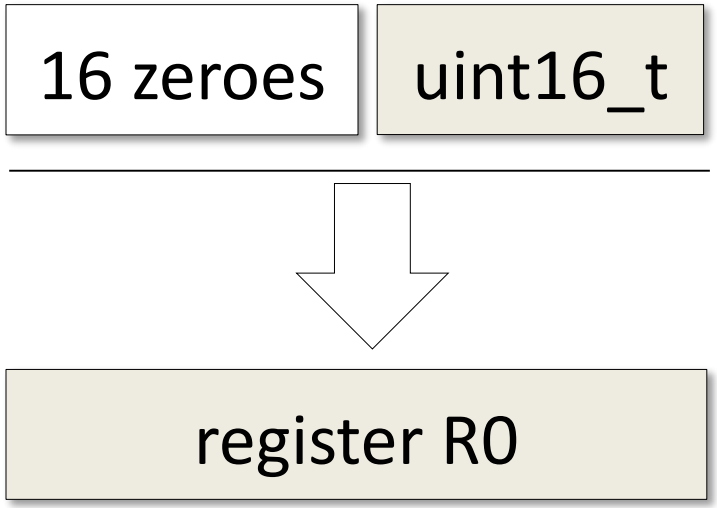
```
// Copies the unsigned
// value held in the
// 8-bit memory location
// whose address is in R1
// into bits 0-7 of register
// R0 and 0's into bits 8-31.
```

Used with data of type **uint8_t**



REGISTER ← MEMORY (16-BITS UNSIGNED)
Load Register from (Unsigned) HalfWord

```
LDRH  R0,[R1]

// Copies the unsigned
// value held in the
// 16-bit memory location
// whose address is in R1
// into bits 0-15 of register
// R0 and 0's into bits 16-31.
```



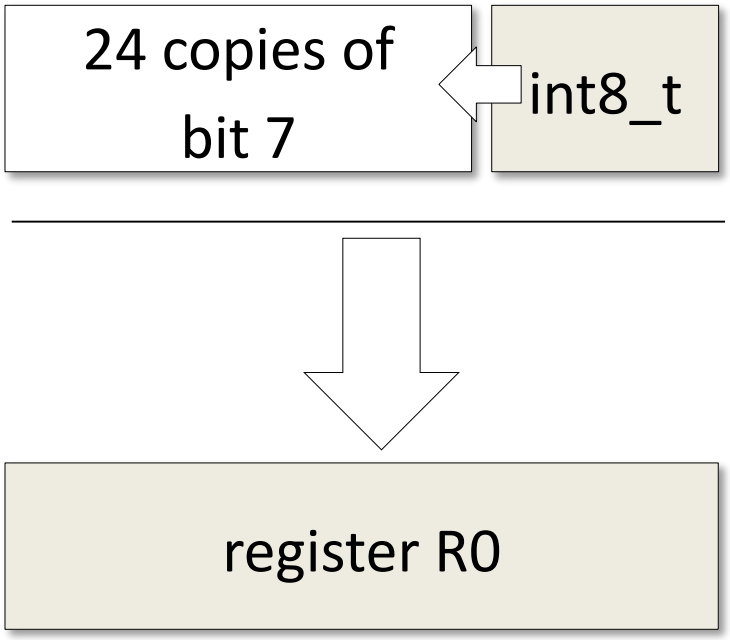
Used with data of type **uint16_t**

REGISTER ← MEMORY (8-BITS SIGNED)
Load Register from Signed Byte

LDRSB R0,[R1]

```
// Copies the signed
// value held in the
// 8-bit memory location
// whose address is in R1
// into bits 0-7 of
// register R0 and 24
// copies of bit 7 of
// sbyte8 into bits 8-31.
```

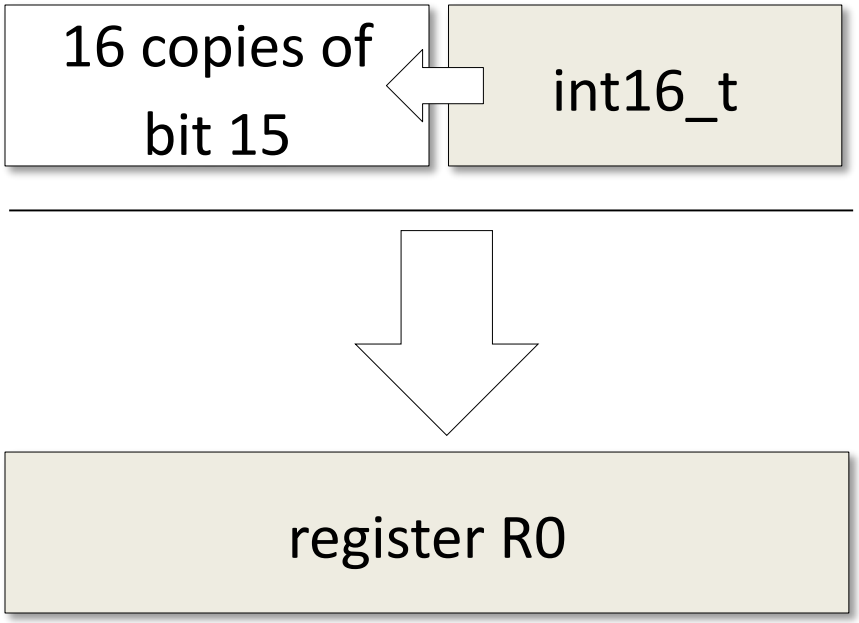
Used with data of type **int8_t**



REGISTER ← MEMORY (16-BITS SIGNED)
Load Register from Signed HalfWord

LDRSH R0,[R1]

```
// Copies the signed
// value held in the
// 16-bit memory location
// whose address is in R1
// into bits 0-15 of R0 and
// 16 copies of bit 15 of
// shalf16 into bits 16-31.
```

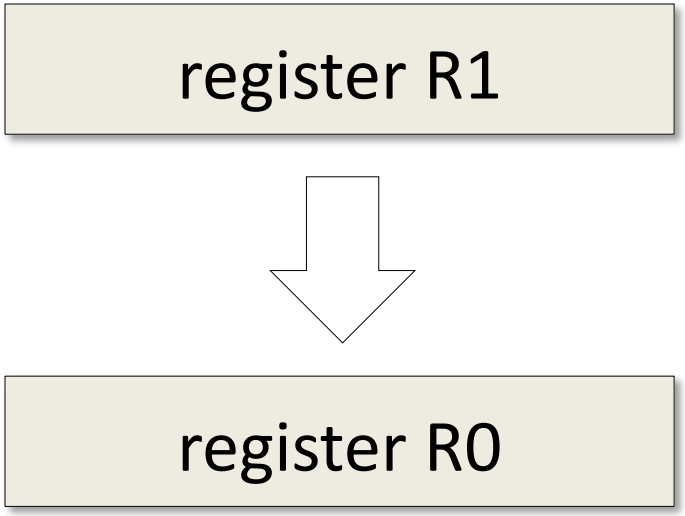


Used with data of type **int16_t**

REGISTER ← REGISTER Move Instruction

```
MOV   R0,R1
```

```
// Copies all 32 bits
// of the value held
// in register R1 into
// the register R0
```

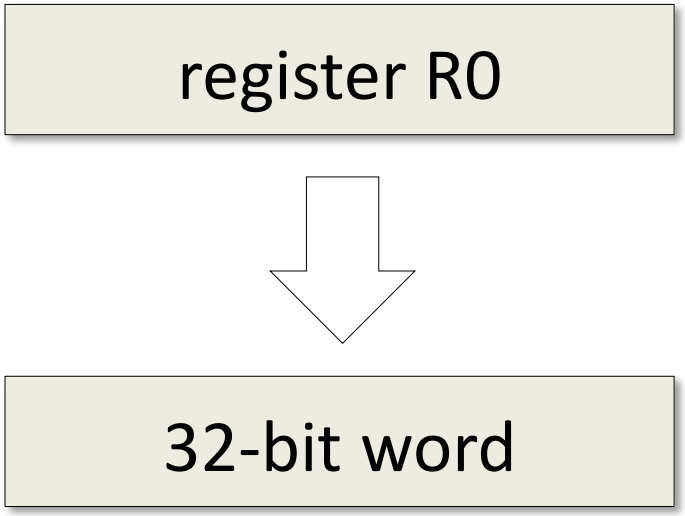


REGISTER → MEMORY (32-BITS)
Store Register to Word

```
STR   R0,[R1]
```

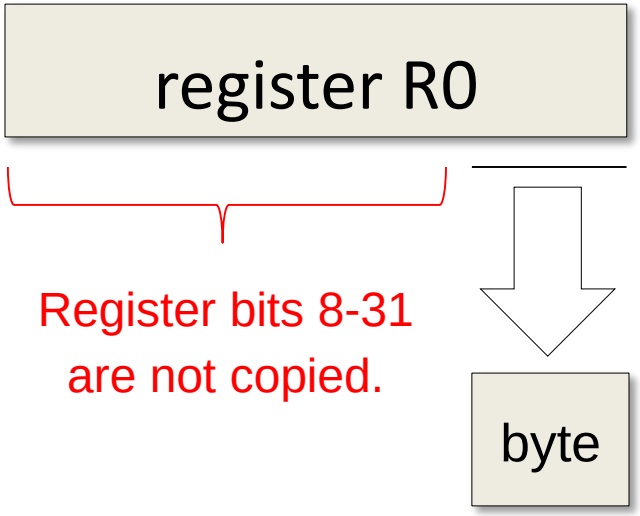
```
// Copies all 32 bits
// of the value held
// in register R0 into
// the 32-bit memory
// location whose address
// is in register R1
```

Used with data of type **int32_t, uint32_t,** and **all pointers**



REGISTER → MEMORY (8-BITS)
Store Register to Byte

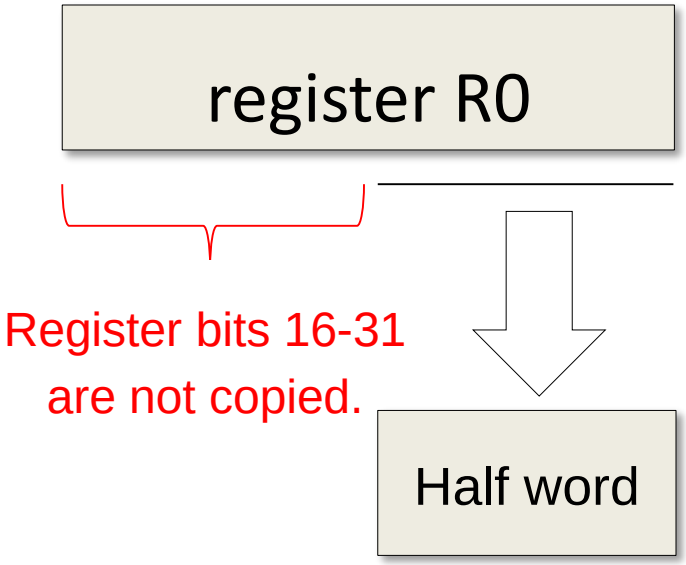
```
STRB  R0,[R1]

// Copies bits 0-7 of
// the value held in
// register R0 into
// the 8-bit memory
// location whose address
// is in register R1
```



Used with data of type **int8_t** and **uint8_t**

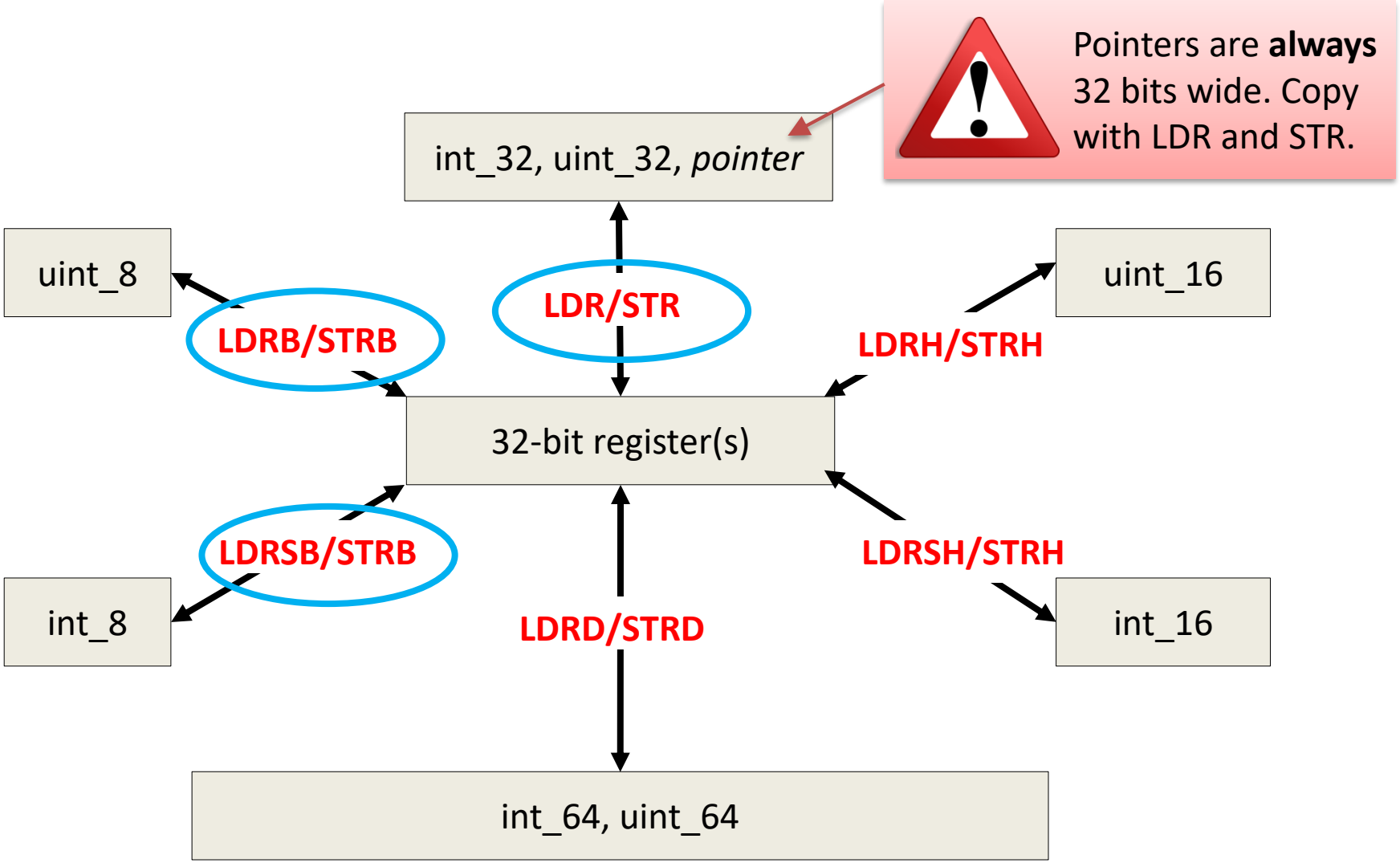
REGISTER → MEMORY (16-BITS)
Store Register to HalfWord

```
STRH  R0,[R1]

// Copies bits 0-15
// of the value held
// in register R0
// into the 16-bit
// memory location
// whose address is
// in register R1.
```



Used with data of type **int16_t** and **uint16_t**

DATA COPYING INSTRUCTIONS



ADR versus LDR

LDR R0,operand	; LDR copies the contents of a memory ; operand (i.e., a variable) into a register.

ADR R0,operand	; ADR copies the address of a memory ; operand (i.e., a constant) into a register.

Function call in C	***Code produced by the compiler***

`void f1(int32_t *) ;`
 	
`int32_t s32 ;`
•
•
•
`f1(&s32) ;`
•
•
•
	•
	•
	•
	`LDR R0,=s32 // load R0 with &s32`
	`BL f1 // call function f1`
	•
	•
	•

ADDRESSING MODES
(Calculating a Memory Address)

Immediate Offset Mode:

[R0]

[R0,4]

R0 is the 'base'; the optional constant 4 is the 'offset' from the base.

Register Offset Mode:

[R0,R1]

[R0,R1,LSL 2]

R0 is the 'base'; R1 or R1,LSL2 is the 'offset' from the base.

Pre-Indexed Mode:

[R0,4]!

1. R0 ← R0 + 4
2. R0 provides address

R0=1000 이라고 가정
LDA R1, [R0, 4] ! 수행하면
R0는 1004, R1에는 1004번지 데이터 저장

Post-Indexed Mode:

[R0],4

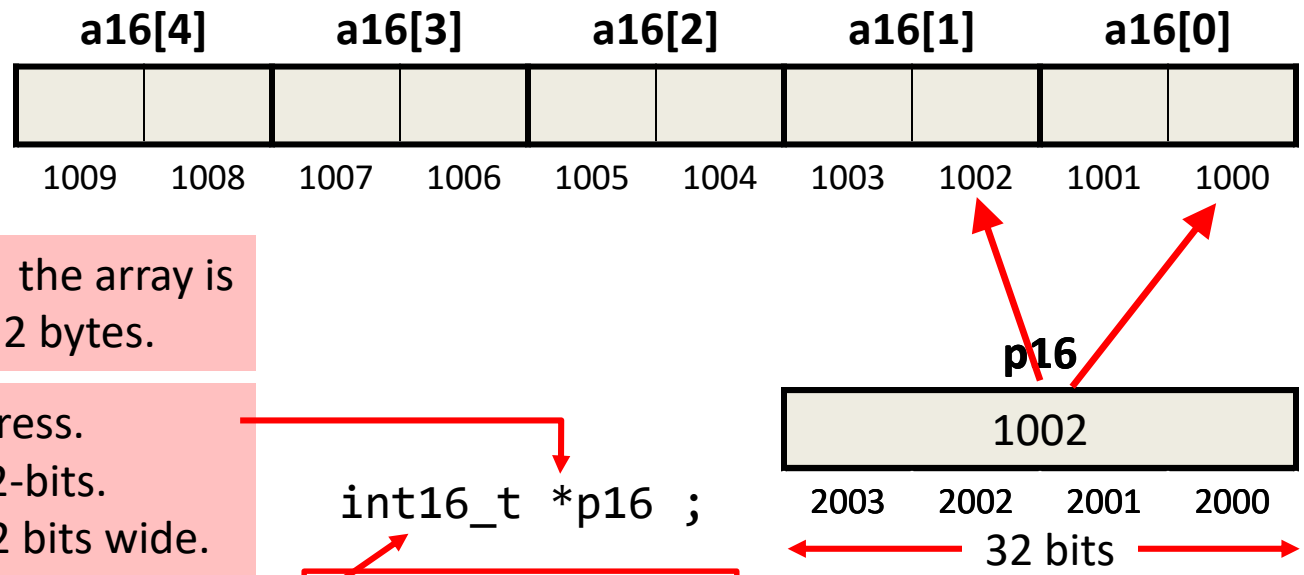
1. R0 provides address
2. R0 ← R0 + 4

R0=1000 이라고 가정
LDA R1, [R0], 4 수행하면
R1에는 1000번지 데이터 저장, R0는 1004

Use these in loops to reduce the number of instructions.

Review: Pointer Arithmetic

```
int16_t a16[5] ;
```



Note: Each member of the array is an object consisting of 2 bytes.

A pointer holds an address.
An address is always 32-bits.
Thus ***all*** pointers are 32 bits wide.

```
int16_t *p16 ;
```

The data type (int16_t) indicates the size and signedness of the ***objects*** that the pointer points to.

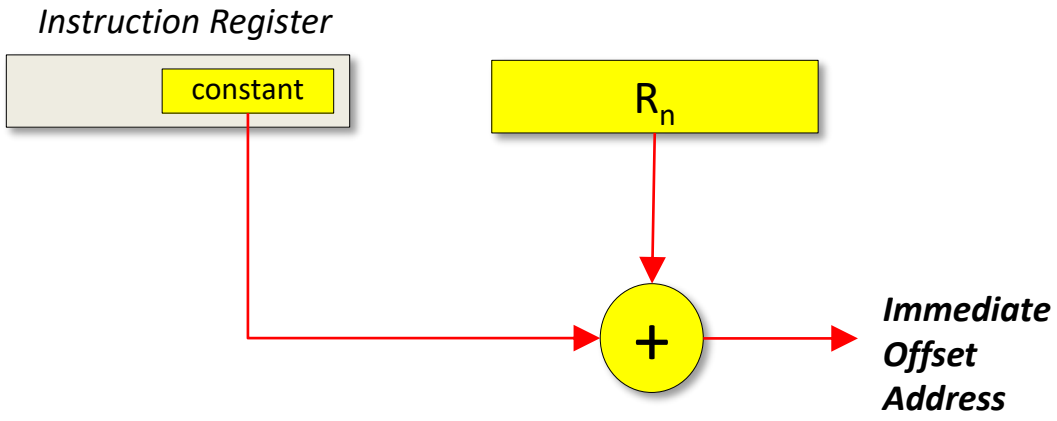
```
p16 = &a16[0] ;
```

```
p16 = p16 + 1 ;
```

Adding 1 to a pointer causes it to point to the next ***object***. Since each object is 2 bytes, the address must increase by 2.

IMMEDIATE OFFSET MODE

Syntax	*Address*	*Examples*
[R_n{,constant}]	R_n + constant	1.[R5,100] 2.[R5]



IMMEDIATE OFFSET: POINTERS & ARRAYS

Function in C	***Function in assembly***

`void f1(int32_t *p32)`
`{`
`*p32 = 0 ;`
`*(p32 + 1) = 0 ;`
`}`
	`f1: LDR R1,=0 // R1 <-- 0`
	`STR R1,[R0] // R1 --> memory[R0]`
	`STR R1,[R0,4] // R1 --> memory[R0+4]`
	`BX LR // return`

Pointer arithmetic! Adding 1 to p32 adds 4 to address.

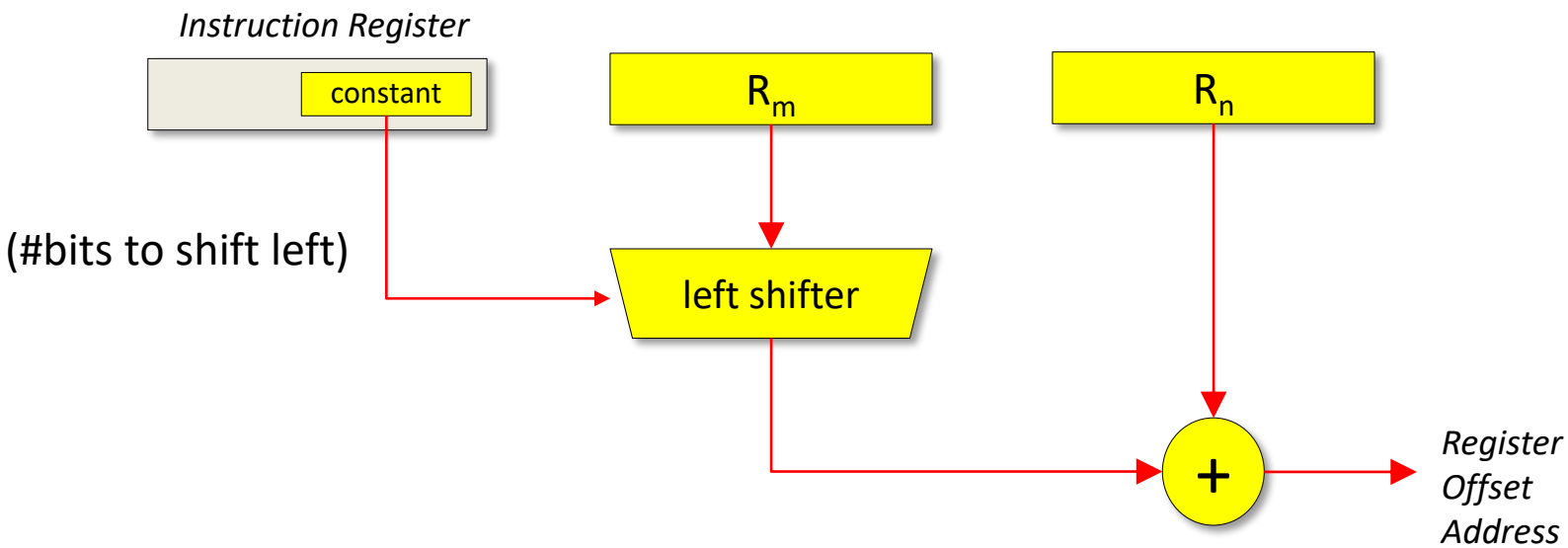
Function in C	***Function in assembly***

`void f2(int32_t a32[])`
`{`
`a32[0] = 0 ;`
`a32[1] = 0 ;`
`}`
	`f2: LDR R1,=0`
	`STR R1,[R0]`
	`STR R1,[R0,4]`
	`BX LR`

Array and pointer parameters are treated the same

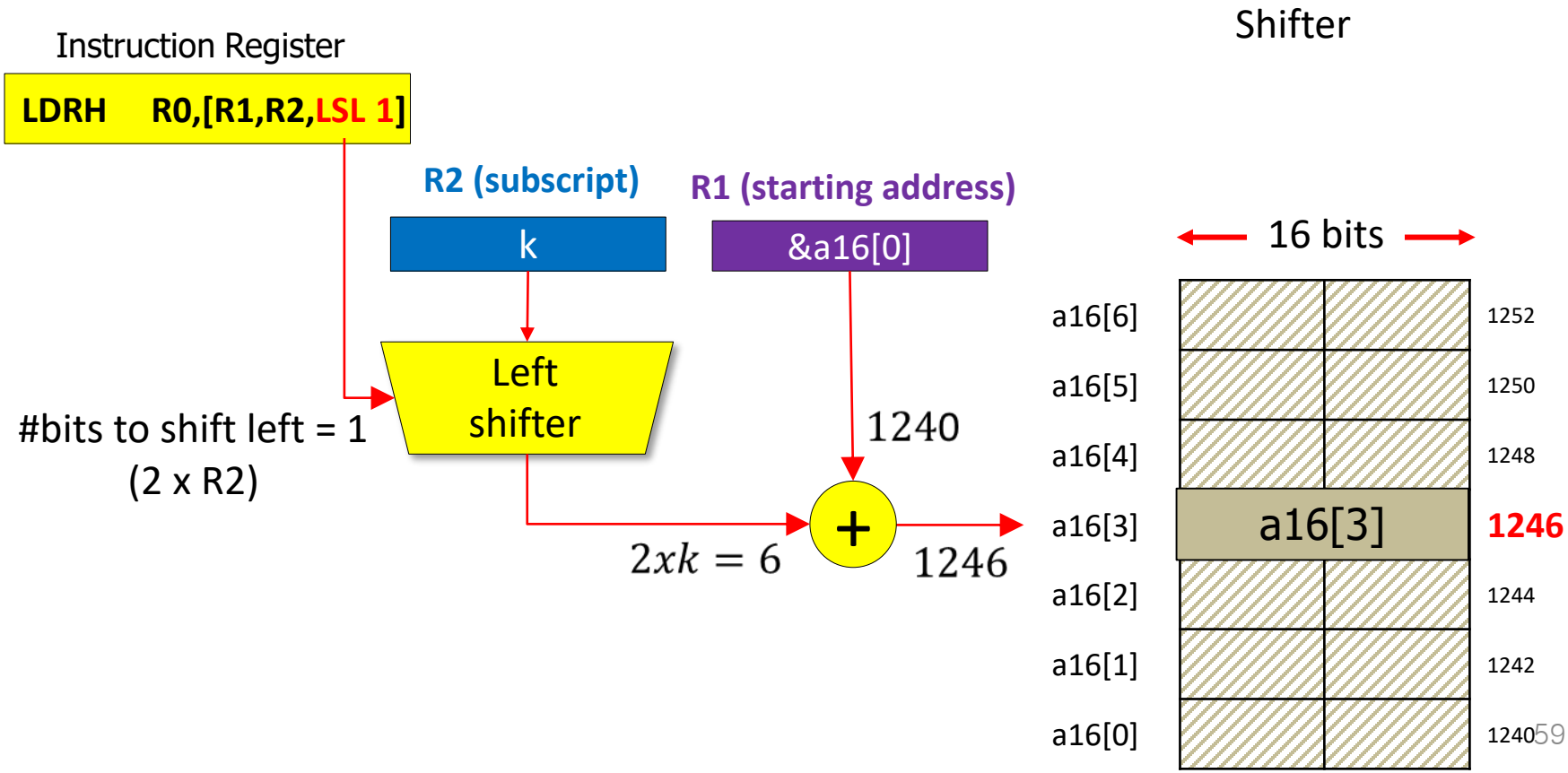
REGISTER OFFSET MODE

Syntax	*Address*	*Example*
$[R_n,R_m]$	$R_n + R_m$	[R4,R5]
$[R_n,R_m,\text{LSL constant}]$	$R_n + (R_m << \text{constant})$	[R4,R5,LSL 2]



Subscripting(index): a16[k] = 0

```
LDR     R0,=0          // R0 ← 0 (data)
LDR     R1,=a16        // R1 ← address of array (&a16[0] = 1240)
LDR     R2,k           // R2 ← subscript (k=3)
STRH    R0,[R1,R2,LSL 1]   // R0 → a16[k]
```



REGISTER OFFSET: POINTERS & ARRAYS

Function in C	***Function in assembly***

`void f1(int8_t *p8, int16_t *p16, int32_t k32)`
`{`
`*(p8  + k32) = 0 ;`
`*(p16 + k32) = 0 ;`
`}`
	`f1: LDR R3,=0`
	`STRBR3,[R0,R2]`
	`STRHR3,[R1,R2,LSL 1]`
	`BX  LR`

Pointer arithmetic!

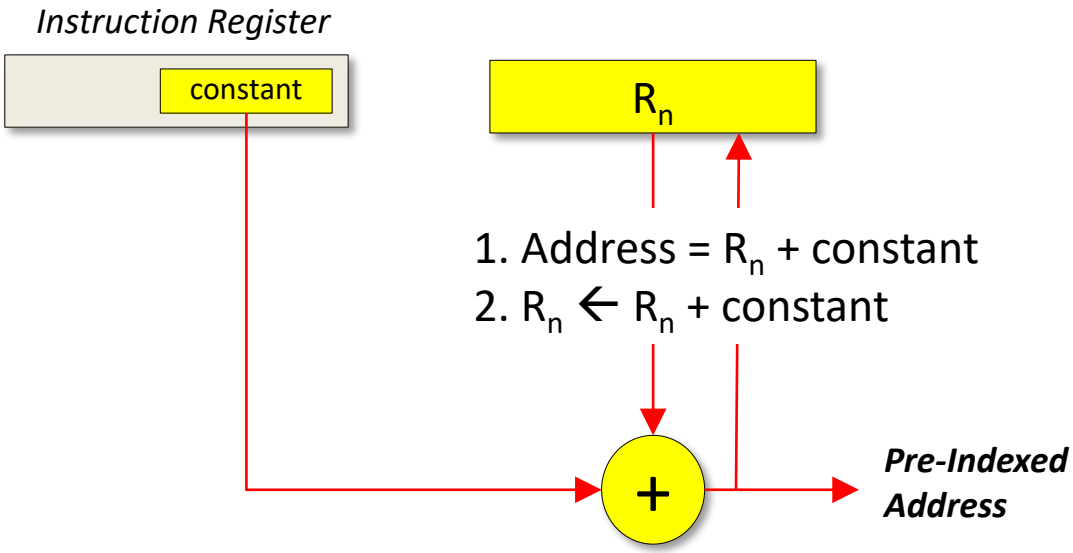
R2,LSL 1 = 2*k32.

Function in C	***Function in assembly***

`void f2(int8_t a8[], int16_t a16[], int32_t k32)`
`{`
`a8[k32]  = 0 ;`
`a16[k32] = 0 ;`
`}`
	`f2: LDR R3,=0`
	`STRBR3,[R0,R2]`
	`STRHR3,[R1,R2,LSL 1]`
	`BX  LR`

PRE-INDEXED MODE

Syntax	*Address*	*Example*	*Side Effect*
$[R_n,\text{constant}]!$	$R_n + \text{constant}$	[R5,4]!	R5 ← R5 + 4



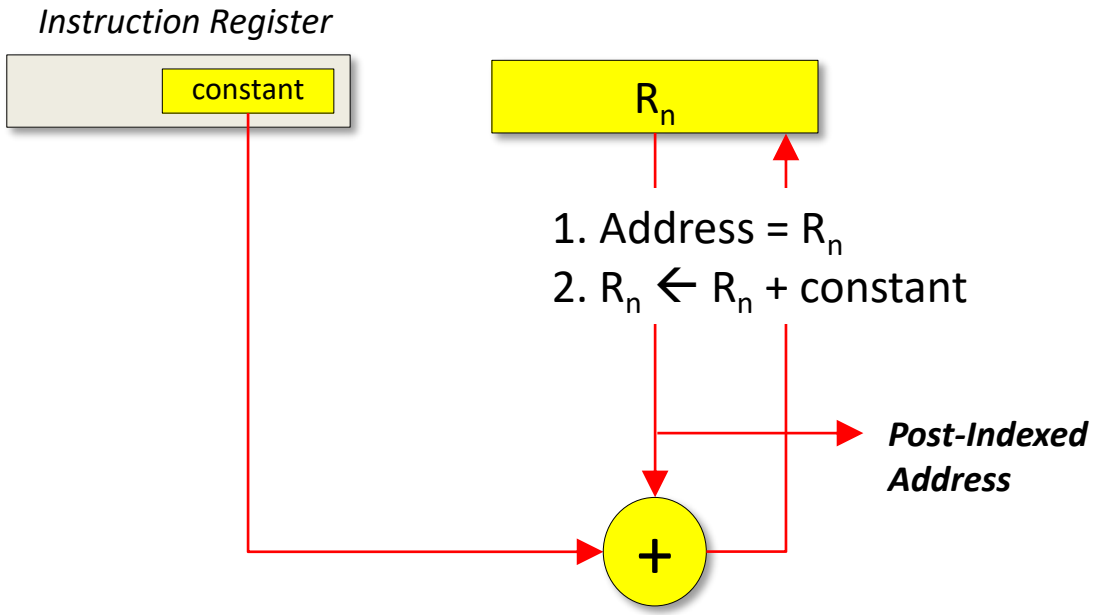

```
ADD     R1,R1,4
LDR     R0,[R1]
```

```
LDR     R0,[R1,4]!
```

Eliminates 1 instruction

POST-INDEXED MODE

Syntax	*Address*	*Example*	*Side Effect*
$[R_n]$,constant	R_n	[R5],4	R5 ← R5 + 4

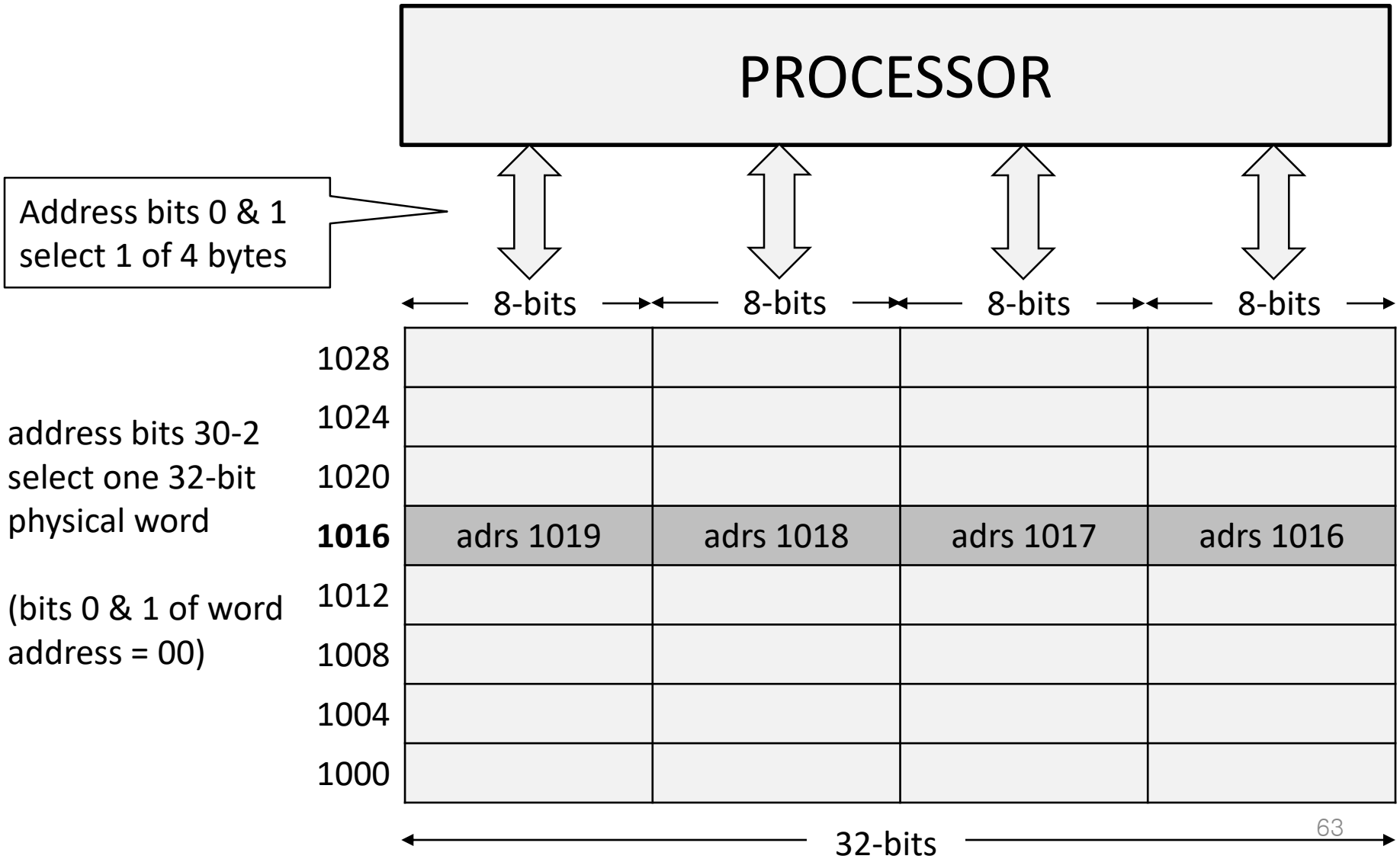



```
LDR     R0,[R1]
ADD     R1,R1,4
```

```
LDR     R0,[R1],4
```

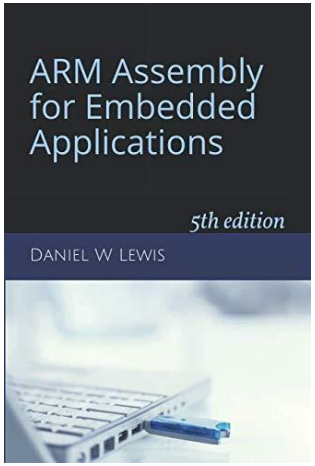
Eliminates 1 instruction

PHYSICAL MEMORY DESIGN



Memory Addressing

- Logically, memory is organized into bytes
- Every byte has its own 32-bit address.
- Every memory read retrieves a 32-bit *physical* word.
- Accessing a byte:
  - Most-significant 30 bits of address select the physical word
  - Least-significant 2 bits of address select one of 4 bytes within the word



Chapter 5

Integer Arithmetic

CONDITION FLAGS

Processor Status Register (PSR)

This subset is the Application Processor Status Register (APSR)

31	30	29	28	27	26 … 0
N	Z	C	V	Q	*reserved*

DSP overflow and saturation flag: *A value of 1 indicates that a saturated arithmetic instruction limited its result.*

Overflow flag: A value of 1 indicates a 2's complement overflow during an addition, subtraction or compare.

Carry or Borrow flag: A value of 1 indicates a carry out from addition or *NO* borrow out during subtraction.

Zero flag: A value of 1 indicates a result (or difference) of zero

Negative flag: A value of 1 indicates a negative result

명령어 끝에 S를 붙여야만 수행후 상태비트들이 update 됨 !

ADDITION AND SUBTRACTION

Instruction	*Format*	*Operation*	*Flags*
Add	ADD{S} Rd,Rn,Op2	Rd ← Rn + Op2	N,Z,C,V
Add with Carry	ADC{S} Rd,Rn,Op2	Rd ← Rn + Op2 + Carry	N,Z,C,V
Subtract	SUB{S} Rd,Rn,Op2	Rd ← Rn – Op2	N,Z,C,V
Subtract with Carry	SBC{S} Rd,Rn,Op2	Rd ← Rn – Op2 – ~Carry	N,Z,C,V
Reverse Subtract	RSB{S} Rd,Rn,Op2	Rd ← Op2 – Rn	N,Z,C,V

"Op2" may only be a constant, a register, or a shifted register.

"S" must be appended to affect the flags!

MULTIPLICATION
For Single-Length Products

Instruction	*Format*	*Operation*
32-bit Multiply	MUL{S} R_d,R_n,R_m	$R_d \leftarrow$ (int32_t) $R_n \times R_m$
32-bit Multiply with Accumulate	MLA R_d,R_n,R_m,R_a	$R_d \leftarrow R_a$ + (int32_t) $R_n \times R_m$
32-bit Multiply & Subtract	MLS R_d,R_n,R_m,R_a	$R_d \leftarrow R_a$ – (int32_t) $R_n \times R_m$

MULS affects flags N and Z. No other multiply instruction affects the flags.

All multiply instructions require their operands to be in registers. No constants or memory operands.

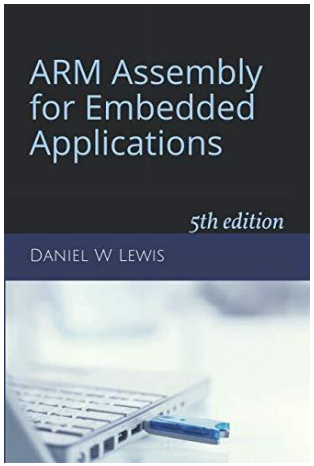
Note: MLA and MLS use the product of the *middle* two registers.

MULTIPLICATION
For Double-Length Products

Instruction	*Format*	*Operation*
64-bit Unsigned Multiply	**U**MULL $R_{dlo}, R_{dhi}, R_n, R_m$	$R_{dhi}R_{dlo} \leftarrow$ (uint64_t) $R_n \times R_m$
64-bit Unsigned Multiply with Accumulate	**U**MLAL $R_{dlo}, R_{dhi}, R_n, R_m$	$R_{dhi}R_{dlo} \leftarrow R_{dhi}R_{dlo}$ + (uint64_t) $R_n \times R_m$
64-bit Signed Multiply	**S**MULL $R_{dlo}, R_{dhi}, R_n, R_m$	$R_{dhi}R_{dlo} \leftarrow$ (int64_t) $R_n \times R_m$
64-bit Signed Multiply with Accumulate	**S**MLAL $R_{dlo}, R_{dhi}, R_n, R_m$	$R_{dhi}R_{dlo} \leftarrow R_{dhi}R_{dlo}$ + (int64_t) $R_n \times R_m$

Multiplication Summary

<table>
<tr><th>32 ← 32 x 32
(single length)</th><th>64 ← 32 x 32
(double length)</th><th>64 ← 64 x 64
(single length)</th></tr>
<tr><td rowspan="2">MUL R3,R1,R2
$\begin{pmatrix} signed\ or \\ unsigned \end{pmatrix}$</td><td>UMULL R2,R3,R0,R1
$(unsigned)$</td><td rowspan="2">UMULL R4,R5,R0,R2
MLA R5,R1,R2,R5
MLA R5,R0,R3,R5
$(signed\ or\ unsigned)$</td></tr>
<tr><td>SMULL R2,R3,R0,R1
$(signed)$</td></tr>
<tr><td>R3←R1 × R2</td><td>R3.R2←R0 × R1</td><td>R5.R4←R1.R0 × R3.R2</td></tr>
</table>

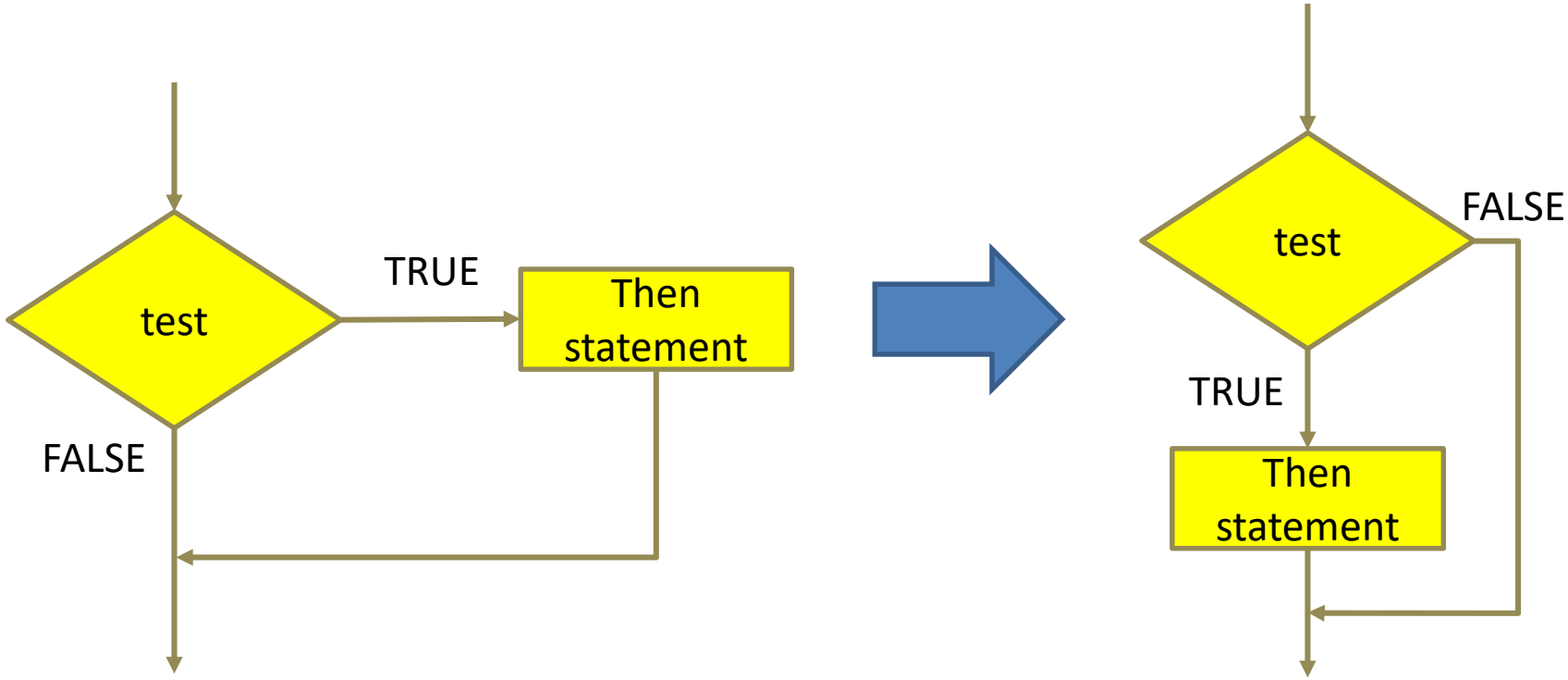


Chapter 6

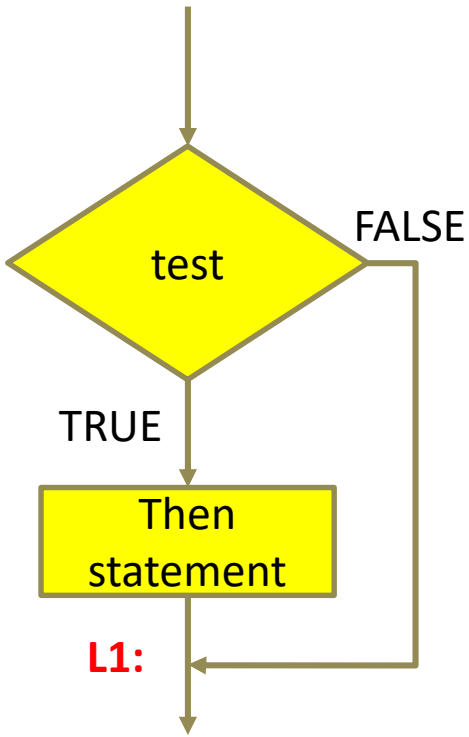
Making Decisions and Writing Loops

HLL(High Level Lang.) versus Assembly

- In assembly, all statements arranged vertically, so we rearrange the flowchart:



This arrangement requires a “goto”




```
    if (test == FALSE) goto L1

    Then-statement

L1:     ….
```

This is similar to assembly, where decisions are implemented as a test followed by an instruction that branches (or not) depending on the result of the test.

COMPARE INSTRUCTIONS

Decisions in assembly require a two instruction sequence. A compare instruction compares two operands, followed by a conditional branch instruction that makes the actual decision to branch or not.

Instruction	***Syntax***	***Operation***	***Notes***
Compare	CMP R_n,*Op2*	R_n – *Op2*	Updates flags N,Z,C and V
Compare Negative	CMN R_n,*Op2*	R_n + *Op2*	

Op2 may be a small constant, a register, or a shifted register.

CMP is identical to the SUBS instruction, but discards the actual difference.

The assembler automatically replaces CMP R_n,–*constant* by CMN R_n,*constant*

TEST INSTRUCTIONS

Instruction	*Syntax*	*Operation*	*Notes*
Test	TST R_n,*Op2*	R_n & *Op2*	Updates flags N and Z; if shifted, *Op2* may affect C flag
Test Equivalence	TEQ R_n,*Op2*	R_n ^ *Op2*	

^는 EX-OR를 의미

TST is used to test whether any of the bits in R_n specified by Op2 are 1's.

TEQ is used to test whether R_n = Op2 without affecting the C or V flags.

CONDITIONAL BRANCH INSTRUCTIONS

If-then statement in C

```
int32_t s32 ;

if (s32 > 10)
•••
```

then statement

Equivalent C code using goto and label

```
int32_t s32 ;

if (s32 <= 10) goto L1 ;
```

then statement

```
L1:    •••
```

Equivalent if-then statement in assembly

```
LDR   R0,=s32// R0 <-- &s32
LDR   R0,[R0]// CMP requires operand to be in a register
CMP   R0,10   // Compare s32 to 10 and ...
BLE   L1      // if (s32 <= 10) goto L1
```

then statement

```
L1: •••
```

Decisions in assembly are like an if statement that controls a goto statement.

A conditional branch instruction is written as “B” followed a condition code.

CONDITIONAL BRANCH INSTRUCTIONS

unsigned ← $0111_2 > 1100_2$? → *signed*

7 > 12 ? **(FALSE)**

+7 > -4 ? **(TRUE)**

Signed and unsigned require different condition codes for magnitude comparisons!

Condition	*Signed*	*Unsigned*
>	**GT** *(Greater Than)*	**HI** *(Higher Than)*
>=	**GE** *(Greater Than or Equal)*	**HS** *(Higher Than or Same)*
<	**LT** *(Less Than)*	**LO** *(Lower Than)*
<=	**LE** *(Less Than or Equal)*	**LS** *(Lower Than or Same)*
==	**EQ** *(Equal)*	**EQ** *(Equal)*
!=	**NE** *(Not Equal)*	**NE** *(Not Equal)*

CHOOSING THE CONDITION

C Source Code	*Incorrect Assembly*	*Correct Assembly*

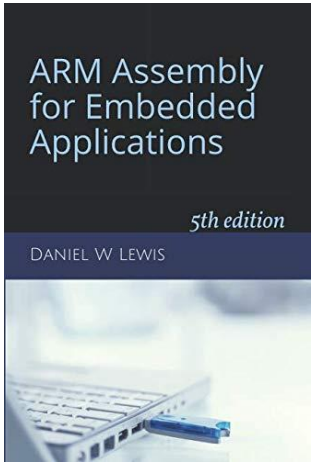
`int32_t s32 ;`
 	
`if (s32 > 10)`
 	
`then statement`
	`LDR R0,=s32`
	`LDR R0,[R0]`
	`CMP R0,10`
	`BGT L1 // NO!`
	
	`then statement`
	
	`L1: ...`
		`LDR R0,=s32`
		`LDR R0,[R0]`
		`CMP R0,10`
		`BLE L1 // YES!`
		
		`then statement`
		
		`L1: ...`

You can't use the same condition as used in C to control an if-statement.

If a condition doesn't include the equality case, the opposite must. Vice-Versa!

COMPLETE CONDITION LIST

Code	*Meaning*	*Requires*
EQ	Equal	EQ and NE are same for signed and unsigned.
NE	Not equal	
GE	Signed ≥ ("Greater than or Equal")	N = V
LT	Signed < ("Less Than")	These are used for signed comparisons
GT	Signed > ("Greater Than")	
LE	Signed ≤ ("Less than or Equal")	Z = 1 || N ≠ V
HS (CS)	Unsigned ≥ ("Higher or Same") or Carry Set	C = 1
LO (CC)	Unsigned < ("Lower") or Carry Clear	These are used for unsigned comparisons
HI	Unsigned > ("Higher")	
LS	Unsigned ≤ ("Lower or Same")	C = 0 || Z = 1
MI	Minus/negative	N = 1
PL	Plus - positive or zero (non-negative)	N = 0
VS	Overflow	V = 1
VC	No overflow	V = 0
AL	Always (unconditional)	Never used. Included for completeness.



Chapter 7

Manipulating Bits

SHIFTING

APPLICATIONS OF SHIFTING

Multiplication by 2^k: $2^k \times A = A << k$

Division by 2^k: $A \div 2^k = A >> k$ (*sort of*)

Subscript Scaling: &a32[k] = &a32[0] + $4 \times k$

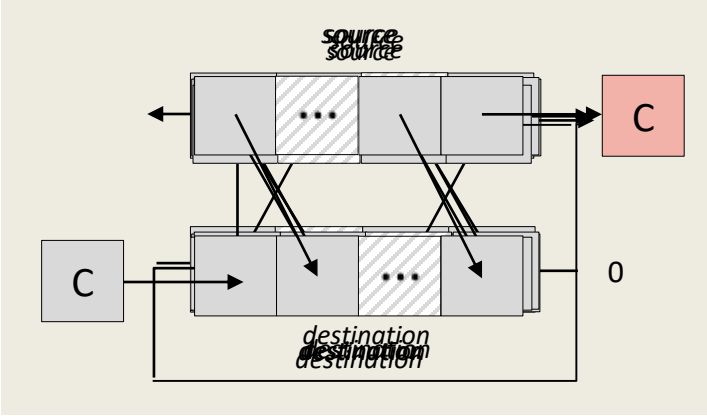
Creating Multiples: $7 \times A = 8 \times A - 1 \times A$

SHIFT INSTRUCTIONS

Instruction	*Syntax*		*Operation*	*Flags*	*Notes*
Logical Shift Left	LSL{S}	R_d,R_n,*bits*	$R_d \leftarrow R_n << bits$	N,Z,C	Zero fills
Logical Shift Right	LSR{S}	R_d,R_n,*bits*	$R_d \leftarrow R_n >> bits$	N,Z,C	Zero fills
Arithmetic Shift Right	ASR{S}	R_d,R_n,*bits*	$R_d \leftarrow R_n >> bits$	N,Z,C	Sign extends
Rotate Right	ROR{S}	R_d,R_n,*bits*	$R_d \leftarrow R_n >> bits$	N,Z,C	right rotate
Rotate Right w/Extend	RRX{S}	R_d,R_n	$R_d \leftarrow R_n >> 1$	N,Z,C	right shift, fill w/C

The only difference is what is used to fill the left-most bit position.

Append "S" to capture the last bit shifted out in the carry (C) flag.



3 PLACES WHERE SHIFT CAN BE USED

ROR R0,R1,R2 **// R0 ← R1 rotated right R2 times**

1. As a regular shift ***instruction***.

The only context in which the shift count may be a variable (e.g., R2 here).

ADD R0,R1,R2,LSR 3 **// R0 ← R1 + (R2 >> 3)**

2. To provide a pre-shifted copy of the value in a register as the last operand of an instruction.

(예) ADD r3, r2, r1, LSL #3
-r1, LSR r2
-r1, ASR #3
-r1, ASR r3
-r1, ROR #2
-r1, ROR r2
-r1, RRX

STR R0,[R1,R2,LSL 2] **// R0 → mem_{32}[R1+4×R2]**

3. To multiply a subscript (R2) by the # of bytes per element in a subscripted array reference.

Only a Logical Shift Left (LSL) by 1, 2 or 3 bits may be used in this context.

BITWISE OPERATIONS

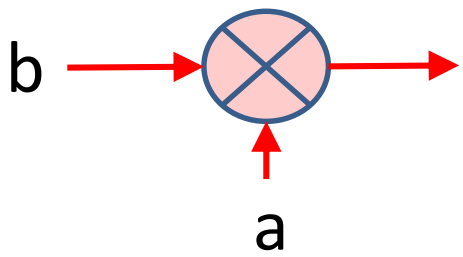
Review: Bitwise Operators

a =	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
b =	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
result =	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0

Each bit position of the result depends <u>only</u> on bits in the same position within the operands.

Bitwise AND

	a	b	a AND b	
a = 0	0	0	0	0
	0	1	0	
a = 1	1	0	0	b
	1	1	1	



If a=0, output = 0

a=1, output = b

Use bitwise AND to **isolate/test** a single bit

	b_3	$\mathbf{b_2}$	b_1	b_0
&	0	**1**	0	0
	0	$\mathbf{b_2}$	0	0

bits & (1 << 2)

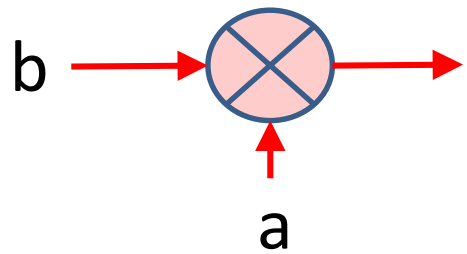
Use bitwise AND to **clear** a single bit to 0

	b_3	$\mathbf{b_2}$	b_1	b_0
&	1	**0**	1	1
	b_3	**0**	b_1	b_0

bits & ~(1 << 2)

Bitwise OR

	a	b	a OR b	
a = 0	0	0	0	b
	0	1	1	
a = 1	1	0	1	1
	1	1	1	



If a=0, output = b

a=1, output = 1

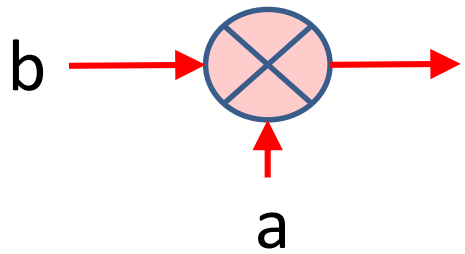
Use bitwise OR to **set** a single bit to 1

	b_3	$\mathbf{b_2}$	b_1	b_0
|	0	**1**	0	0
	b_3	**1**	b_1	b_0

bits | (1 << 2)

Bitwise Exclusive-OR

	a	b	a XOR b	
a = 0	0	0	0	b
	0	1	1	
a = 1	1	0	1	$\overline{b}$
	1	1	0	



If a=0, output = b

a=1, output = $\overline{b}$

Use bitwise XOR to **change** a single bit

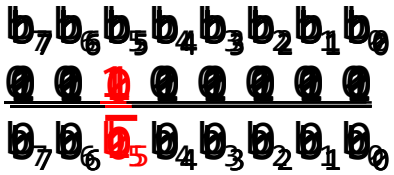
	b_3	$\mathbf{b_2}$	b_1	b_0
^	0	**1**	0	0
	b_3	$\overline{\mathbf{b_2}}$	b_1	b_0

bits ^ (1 << 2)

BITWISE INSTRUCTIONS

Instruction	*Syntax*	*Operation*	*Flags*
Bitwise AND	AND{S} R_d,R_n,*Op2*	$R_d \leftarrow R_n$ & *Op2*	N,Z,C
Bit Clear	BIC{S} R_d,R_n,*Op2*	$R_d \leftarrow R_n$ & ~*Op2*	N,Z,C
Bitwise OR	ORR{S} R_d,R_n,*Op2*	$R_d \leftarrow R_n$ | *Op2*	N,Z,C
Exclusive OR	EOR{S} R_d,R_n,*Op2*	$R_d \leftarrow R_n$ ^ *Op2*	N,Z,C
Bitwise OR NOT	ORN{S} R_d,R_n,*Op2*	$R_d \leftarrow R_n$ | ~*Op2*	N,Z,C
Move NOT	MVN{S} R_d,R_n	$R_d \leftarrow$ ~R_n	N,Z,C

```
AND     R0,R0,1 << 5        // Isolate bit #5
AND     R0,R0,~(1 << 5)     // Clear bit #5 to 0
BIC     R0,R0,1 << 5        // Clear bit #5 to 0
ORR     R0,R0,1 << 5        // Set bit #5 to a 1
EOR     R0,R0,1 << 5        // Change the value of bit #5
```



USING BITWISE INSTRUCTIONS

- Set a bit to 1 without affecting other bits:

  ```
  X |= (1 << 5)
  ```

  ```
  LDR    R1,=x
  LDR    R0,[R1]
  ORR    R0,R0,1<<5
  STR    R0,[R1]
  ```

- Clear a bit to 0 without affecting other bits:

  ```
  X &= ~(1 << 5) ;
  ```

  ```
  LDR    R1,=x
  LDR    R0,[R1]
  BIC    R0,R0,1<<5
  STR    R0,[R1]
  ```

- Change a bit without affecting other bits.

  ```
  X ^= (1 << 5) ;
  ```

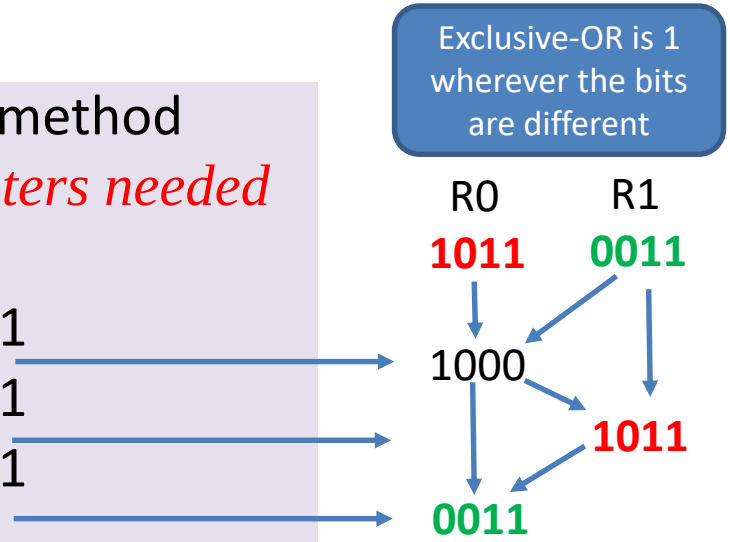
  ```
  LDR    R1,=x
  LDR    R0,[R1]
  EOR    R0,R0,1<<5
  STR    R0,[R1]
  ```

Tricks

Register Swap Using Exclusive-OR

```
// Standard method
// (requires 3rd register)

MOVS    R2,R0
MOVS    R0,R1
MOVS    R1,R2
```

```
// Exclusive-OR method
// No extra registers needed

EOR     R0,R0,R1
EOR     R1,R0,R1
EOR     R0,R0,R1
```



Selection Without a Branch

Consider Bitwise OR when at least one input is 0:

When a=0,
output = b

a	b	a OR b
0	0	0
0	1	1
1	0	1
1	1	1

When b=0,
output = a

// Selection: R5 = (R0 < 0) ? R1 : R2

```
AND   R3,R1,R0,ASR 31  // R3 = (R0 <  0) ? R1 : 0
BIC   R4,R2,R0,ASR 31  // R4 = (R0 >= 0) ? R2 : 0
ORR   R5,R3,R4         // One of R3 or R4 will be all 0's
```

A Fast Absolute Value Function

	a	b	a XOR b	
a = 0	0	0	0	b
	0	1	1	
a = 1	1	0	1	$\overline{b}$
	1	1	0	

Recall that exclusive OR can be used to pass an operand through unmodified or inverted

All 1's if s32 < 0
All 0's if s32 ≥ 0

```
// uint32_t AbsValue(int32_t s32)

AbsValue:                          // s32 < 0:        else:
    EOR    R1,R0,R0,ASR 31         //   R1 = ~s32        R1 = s32
    ADD    R0,R1,R0,LSR 31         //   R0 = ~s32 + 1    R0 = s32 + 0
    BX     LR                      // Return
    .end
```

1 if s32 < 0
0 if s32 ≥ 0

Fast! Avoids using making a compare and branch so that the instruction pipeline never stalls.

2장 Part 2 목표 - ARM 명령어 사용법 이해

ARM 어셈블리 언어

종류	명령어	예	의미	비고
산술	add	ADD r1,r2,r3	r1 = r2 + r3	레지스터 피연산자 3개
	subtract	SUB r1,r2,r3	r1 = r2 - r3	레지스터 피연산자 3개
데이터 전송	load register	LDR r1, [r2,#20]	r1 = Memory[r2 + 20]	워드를 메모리에서 레지스터로
	store register	STR r1, [r2,#20]	Memory[r2 + 20] = r1	워드를 레지스터에서 메모리로
	load register halfword	LDRH r1, [r2,#20]	r1 = Memory[r2 + 20]	하프워드를 메모리에서 레지스터로
	load register halfword signed	LDRHS r1, [r2,#20]	r1 = Memory[r2 + 20]	하프워드를 메모리에서 레지스터로
	store register halfword	STRH r1, [r2,#20]	Memory[r2 + 20] = r1	하프워드를 레지스터에서 메모리로
	load register byte	LDRB r1, [r2,#20]	r1 = Memory[r2 + 20]	바이트를 메모리에서 레지스터로
	load register byte signed	LDRBS r1, [r2,#20]	r1 = Memory[r2 + 20]	바이트를 메모리에서 레지스터로
	store register byte	STRB r1, [r2,#20]	Memory[r2 + 20] = r1	바이트를 레지스터에서 메모리로
	swap	SWP r1, [r2,#20]	r1 = Memory[r2 + 20], Memory[r2 + 20] = r1	레지스터와 메모리 간의 원자적 교환
	mov	MOV r1, r2	r1 = r2	값을 레지스터로 복사
논리	and	AND r1, r2, r3	r1 = r2 & r3	레지스터 피연산자 3개; 비트 대 비트 AND
	or	ORR r1, r2, r3	r1 = r2 | r3	레지스터 피연산자 3개; 비트 대 비트 OR
	not	MVN r1, r2	r1 = ~ r2	레지스터 피연산자 3개; 비트 대 비트 NOT
	logical logical shift left (optional operation)	LSL r1, r2, #10	r1 = r2 << 10	상수만큼 좌측 자리이동
	logical shift right (optional operation)	LSR r1, r2, #10	r1 = r2 >> 10	상수만큼 우측 자리이동
조건부 분기	compare	CMP r1, r2	cond. flag = r1 - r2	조건부 분기를 위한 비교
	branch on EQ, NE, LT, LE, GT, GE, LO, LS, HI, HS, VS, VC, MI, PL	BEQ 25	if (r1 == r2) go to PC + 8 + 100	조건 테스트; PC-상대 주소
무조건 분기	branch (always)	B 2500	go to PC + 8 + 10000	분기
	branch and link	BL 2500	r14 = PC + 4; go to PC + 8 + 10000	프로시저 호출용

부록

LDM/STM의 레지스터 리스트

- **<register_list>**에서 사용 가능한 레지스터
  - R0에서 R15(PC)까지 최대 16개
- 연속된 레지스터 표현
  - {r0-r5}와 같이 "-"로 표현 가능
- **<register_list>**의 순서 지정
  - 항상 low order의 register에서 high order 순으로 지정

LDM r10, {r2,r3,r1} → *LDM r10, {r1,r2,r3}*

실제 동작

Subroutine return Instructions

P34에서

```
f2: PUSH{LR}
      •
      •
      BL        f3
      •
      •
    POP {PC}
```

- **Return from a leaf subroutine call**
  - MOV pc, r14 ; r14=LR

- **Return from nested subroutine call**
  - **SUB1 STMFD r13!, {r4-r6, r14} ; save work regs and link**
    BL SUB2
    .....
    LDMFD r13!, {r4-r6, pc } ; restore work regs and return

M[r13−4]=r14
M[r13−8]=r6
M[r13−12]=r5
M[r13−16]=r4
r13'=r13 −16

STMFD = STMDB = PUSH
LDMFD = LDMIA = POP

r4=M[r13']
r5=M[r13'+4]
r6=M[r13'+8]
pc=M[r13'+12]
r13''=r13'+16 =r13

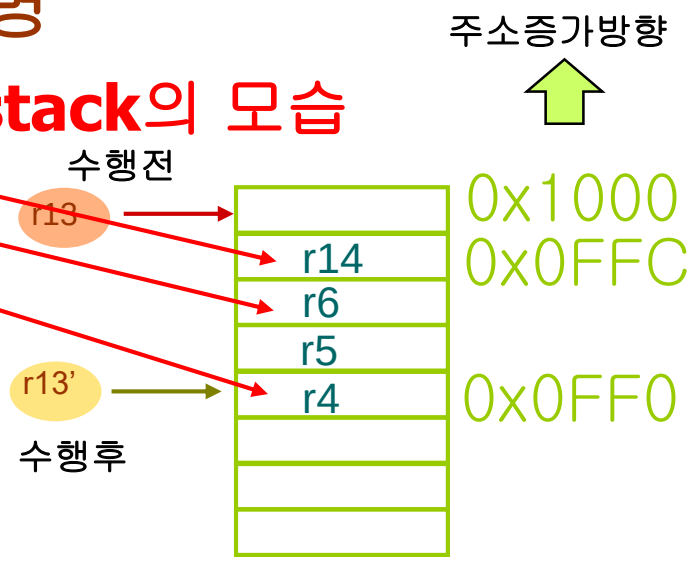
SUB1 안에서 r4−r6 레지스터를 사용할 경우임. SUB1 종료후 return 시 r4−r6 값은 원상복구되어야 함

PUSH (=STMFD=STMDB) 및 POP(=LTMFD=LDMIA) 동작설명

- **PUSH r13!, {r4-r6, r14} 수행후 stack의 모습**

M[r13-4]=r14
M[r13-8]=r6
M[r13-12]=r5
M[r13-16]=r4
r13'=r13 -16

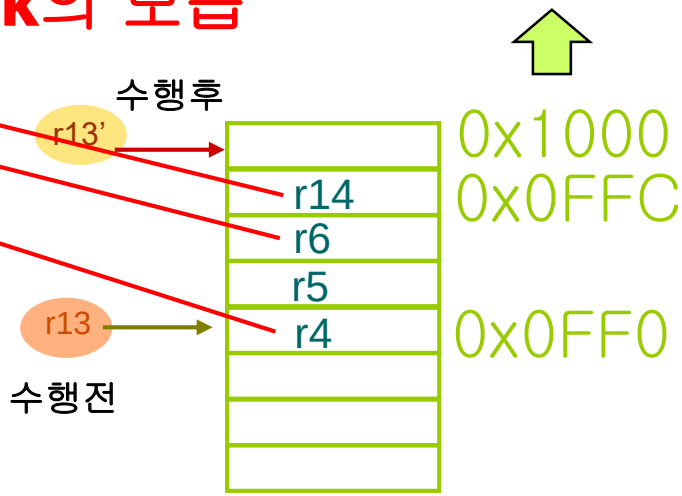
(예) r13=1000
M[0FFC]=r14
M[0FF8]=r6
M[0FF4]=r5
M[0FF0]=r4
r13'=0FF0



- **POP r13!, {r4-r6, pc } 수행후 stack의 모습**

r4=M[r13']
r5=M[r13'+4]
r6=M[r13'+8]
pc=M[r13'+12]
r13''=r13'+16 =r13

(예) r13=0FF0
r4=M[0FF0]
r5=M[0FF4]
r6=M[0FF8]
pc=M[0FFC]
r13''=1000



LDM/STM의 어드레스 지정 방식

ARM compiler에서 사용

Addressing Mode	키워드(표현방식)		유효 어드레스 계산
	데이터	스택	
Pre-increment Load	LDMIB	LDMED	Increment before load
Post-increment Load	LDMIA	LDMFD	Increment after load
Pre-decrement Load	LDMDB	LDMEA	Decrement before load
Post-decrement Load	LDMDA	LDMFA	Decrement after load
Pre-increment Store	STMIB	STMFA	Increment before store
Post-increment Store	STMIA	STMEA	Increment after store
Pre-decrement Store	STMDB	STMFD	Decrement before store
Post-decrement Store	STMDA	STMED	Decrement after store

데이터 필드에 있는 명령어들은 메모리에서 블럭 데이터 저장함을 나타내는 표현이고 스택 필드에 있는 명령어들은 해당 메모리가 stack으로 사용될 경우 stack에 블럭 데이터를 저장함을 나타내는 표현임. 바로 옆에 있는 명령어들은 이름만 다를 뿐 실제로는 동일한 동작. (예)LDMIA=LDMFD

Pre-Increment 어드레스 지정

R0를 메모리에 저장하기 전에 R9를 증가시킨다.

STMIB R9!, {R0,R1,R2}

Base 레지스터(R9) = 0x100C

- **R9(어드레스) 증가 후 데이터 저장**

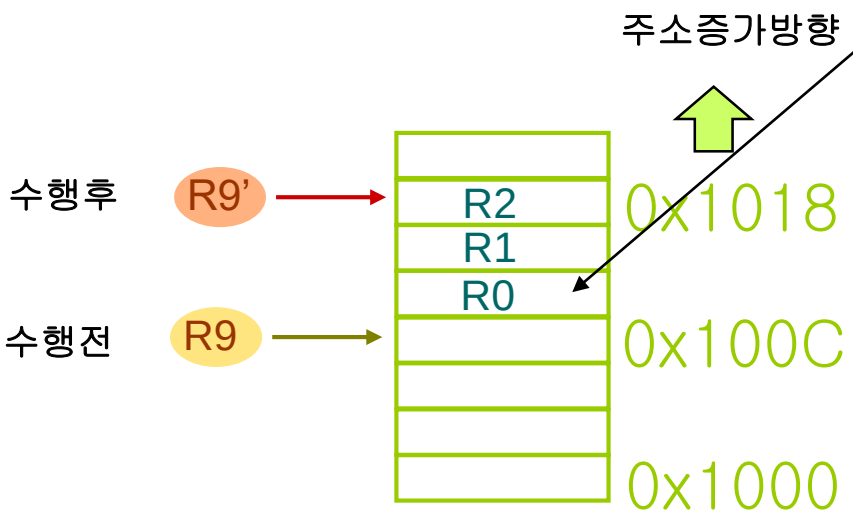
(Increment Before)

① **R9 -> 0x1010 증가 후 R0 저장**

② R9 -> 0x1014 증가 후 R1 저장

③ R9 -> 0x1018 증가 후 R2 저장

④ {!}, auto-update 옵션이 있으면 R9 값을 0x1018로 변경



P100에서 STMIB = STMFA.
FA는 Full Ascending 의 약자임. Stack 동작으로 보면 위 그림 R9은 stack pointer와 같음. Full은 R9이 가리키는 곳에 data가 들어 있는 경우를 의미. Ascending은 stack 에 데이터가 저장될 수록 주소가 증가함을 의미함.

Post-Increment 어드레스 지정

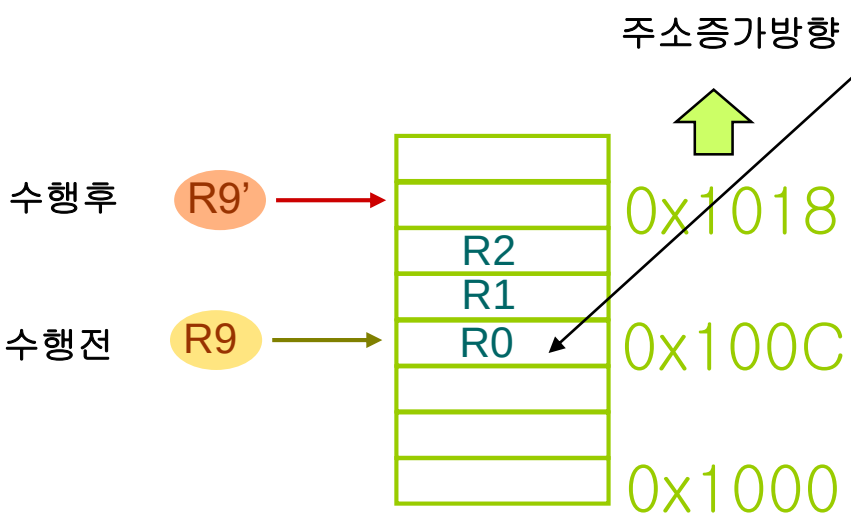
R0를 메모리에 저장한후 R9를 증가시킨다.

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

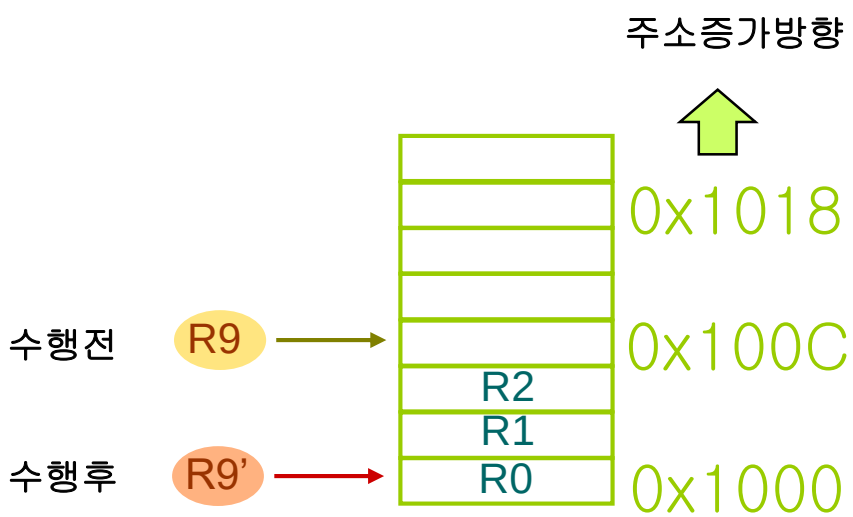
P100에서 STMIA = STMEA.
EA는 Empty Ascending 의 약자임. Stack 동작으로 보면 위 그림 R9은 stack pointer와 같슴. Empty는 R9이 가리키는 곳에 data가 비어있슴을 의미. Ascending은 stack 에 데이터가 저장될 수록 주소가 증가함을 의미함.

Pre-Decrement 어드레스 지정

P100의 PUSH에 해당하는 동작

STMDB R9!, {R0,R1,R2}

Base 레지스터(R9) = 0x100C



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

(Decrement Before)-> (Increment After)

① 어드레스를 **0x1000**로 감소

② 0x1000에 R0 저장 후 어드레스 증가

③ 0x1004에 R1 저장 후 어드레스 증가

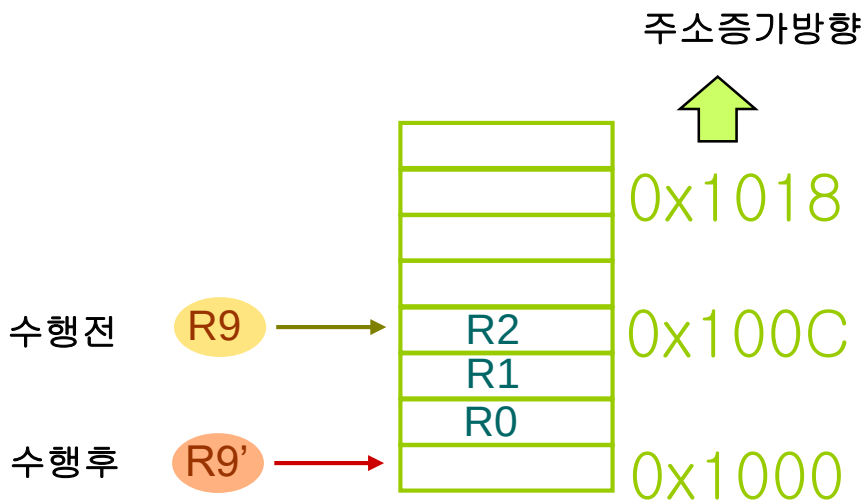
④ 0x1008에 R2 저장 후 어드레스 증가

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) -> (Increment Before)

① 어드레스를 **0x1000**로 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