

Computer Architecture

Instruction: Language of the Computer
Part 3. 코드 최적화 및 ARM 프로그램 예제

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 3. Contents

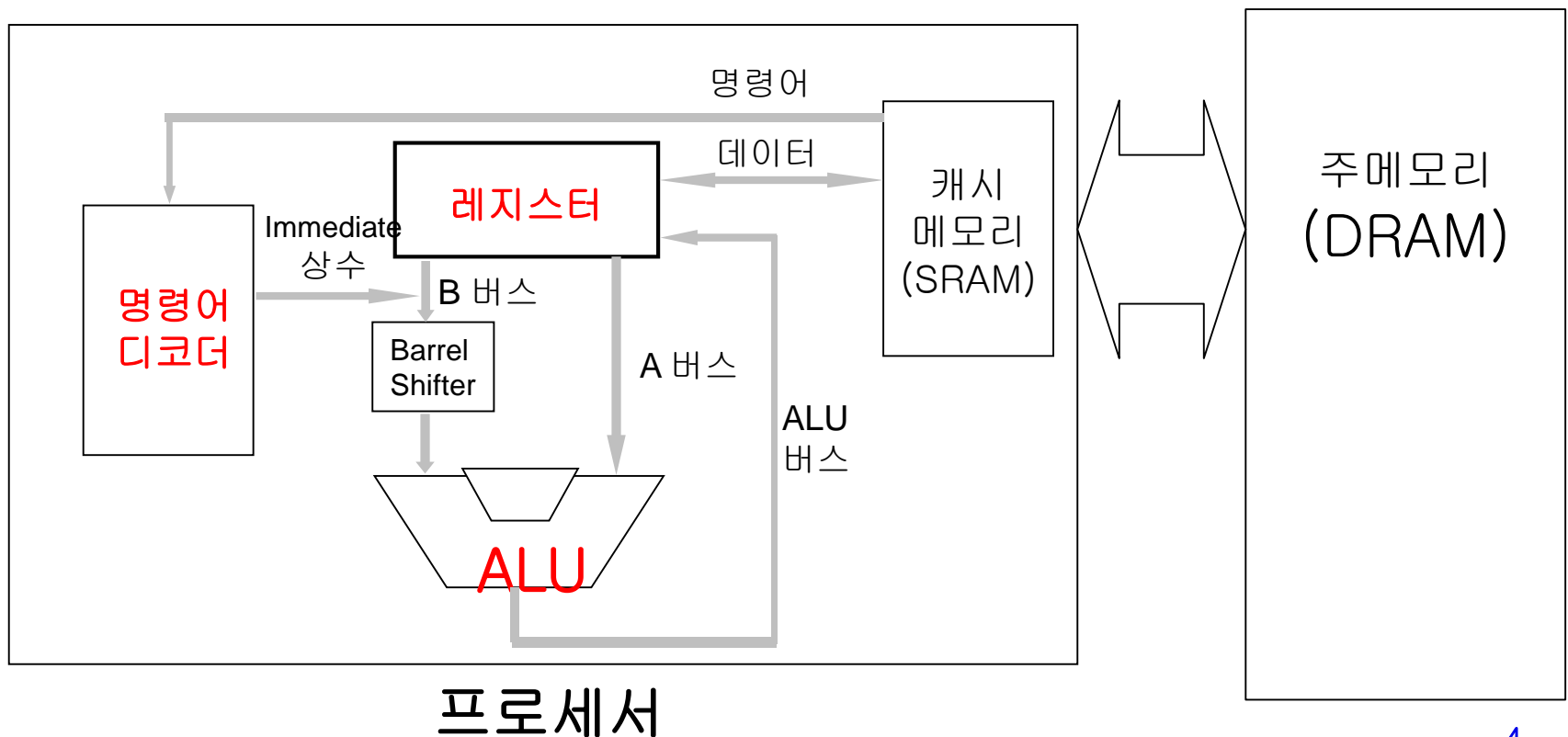
Part3: 코드 최적화 및 **ARM assembly program** 예제

- 간단한 코드 최적화
- 함수호출
- **Sorting** 예제

[복습]ARM 프로세서의 레지스터

■ 레지스터

- ❖ 프로세서가 연산 작업을 하는데 사용되는 값을 임시로 저장하는 공간
- ❖ **ARM**에는 32비트 길이의 **37개**의 레지스터가 있다.



[복습] ARM 레지스터 집합

System mode

User mode

IRQ

FIQ

Undef

Abort

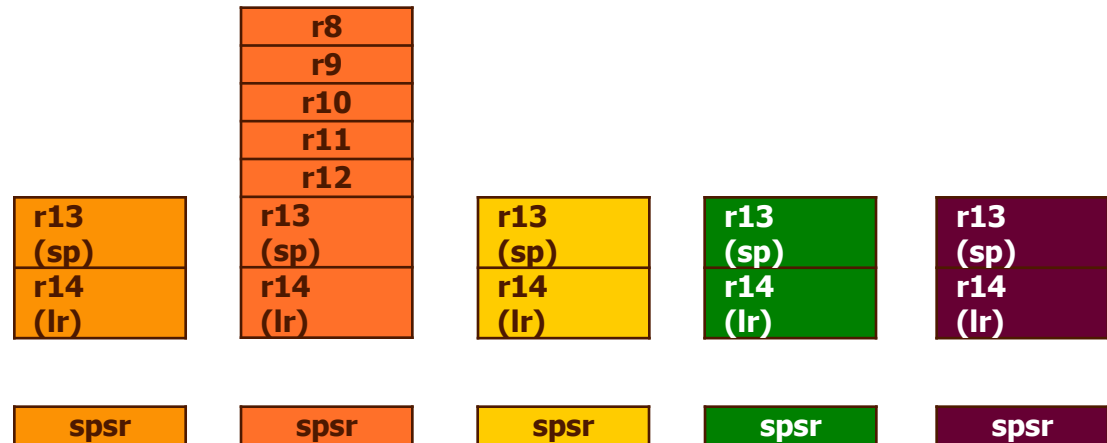
SVC

r0
r1
r2
r3
r4
r5
r6
r7
r8
9
r10
r11
r12
r13(sp)
r14(lr)
r15(pc)

cpsr

current mode

-그림상에 있는 레지스터 총 갯수는 37개
-System mode 또는 User mode 인 경우는
맨 왼쪽 column r0~r15, cpsr 총 17개 사용



Banked out registers

[복습] 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]	$r1 = \text{Memory}[r2 + 20]$	워드를 메모리에서 레지스터로
	store register	STR r1, [r2, #20]	$\text{Memory}[r2 + 20] = r1$	워드를 레지스터에서 메모리로
	load register halfword	LDRH r1, [r2, #20]	$r1 = \text{Memory}[r2 + 20]$	하프워드를 메모리에서 레지스터로
	load register halfword signed	LDRSH r1, [r2, #20]	$r1 = \text{Memory}[r2 + 20]$	하프워드를 메모리에서 레지스터로
	store register halfword	STRH r1, [r2, #20]	$\text{Memory}[r2 + 20] = r1$	하프워드를 레지스터에서 메모리로
	load register byte	LDRB r1, [r2, #20]	$r1 = \text{Memory}[r2 + 20]$	바이트를 메모리에서 레지스터로
	load register byte signed	LDRSB r1, [r2, #20]	$r1 = \text{Memory}[r2 + 20]$	바이트를 메모리에서 레지스터로
	store register byte	STRB r1, [r2, #20]	$\text{Memory}[r2 + 20] = r1$	바이트를 레지스터에서 메모리로
	swap	SWP r1, [r2, #20]	$r1 = \text{Memory}[r2 + 20], \text{Memory}[r2 + 20] = r1$	레지스터와 메모리 간의 원자적 교환
	mov	MOV r1, r2	$r1 = r2$	값을 레지스터로 복사
	and	AND r1, r2, r3	$r1 = r2 \& r3$	레지스터 피연산자 3개; 비트 대 비트 AND
논리	or	ORR r1, r2, r3	$r1 = r2 r3$	레지스터 피연산자 3개; 비트 대 비트 OR
	not	MVN r1, r2	$r1 = \sim r2$	레지스터 피연산자 3개; 비트 대 비트 NOT
	logical shift left (optional operation)	LSL r1, r2, #10	$r1 = r2 \ll 10$	상수만큼 좌측 자리이동
	logical shift right (optional operation)	LSR r1, r2, #10	$r1 = r2 \gg 10$	상수만큼 우측 자리이동
	compare	CMP r1, r2	$\text{cond. flag} = r1 - r2$	조건부 분기를 위한 비교
조건부 분기	branch on EQ, NE, LT, LE, GT, GE, LO, LS, HI, HS, VS, VC, MI, PL	BBQ 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	$r14 = \text{PC} + 4$; go to PC + 8 + 10000	프로시저 호출용

[복습]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
	load register	LDR r1, [r2,#20]	$r1 = \text{Memory}[r2 + 20]$	워드를 메모리에서 레지스터로
	store register	STR r1, [r2,#20]	$\text{Memory}[r2 + 20] = r1$	워드를 레지스터에서 메모리로
	load register halfword	LDRH r1, [r2,#20]	$r1 = \text{Memory}[r2 + 20]$	하프워드를 메모리에서 레지스터로
데이터 전송	load register halfword signed	LDRHS r1, [r2,#20]	$r1 = \text{Memory}[r2 + 20]$	하프워드를 메모리에서 레지스터로
	store register halfword	STRH r1, [r2,#20]	$\text{Memory}[r2 + 20] = r1$	하프워드를 레지스터에서 메모리로
	load register byte	LDRB r1, [r2,#20]	$r1 = \text{Memory}[r2 + 20]$	바이트를 메모리에서 레지스터로
	load register byte signed	LDRBS r1, [r2,#20]	$r1 = \text{Memory}[r2 + 20]$	바이트를 메모리에서 레지스터로
	store register byte	STRB r1, [r2,#20]	$\text{Memory}[r2 + 20] = r1$	바이트를 레지스터에서 메모리로
	swap	SWP r1, [r2,#20]	$r1 = \text{Memory}[r2 + 20], \text{Memory}[r2 + 20] = r1$	레지스터와 메모리 간의 원자적 교환
	mov	MOV r1, r2	$r1 = r2$	값을 레지스터로 복사

[복습]명령어 set 2

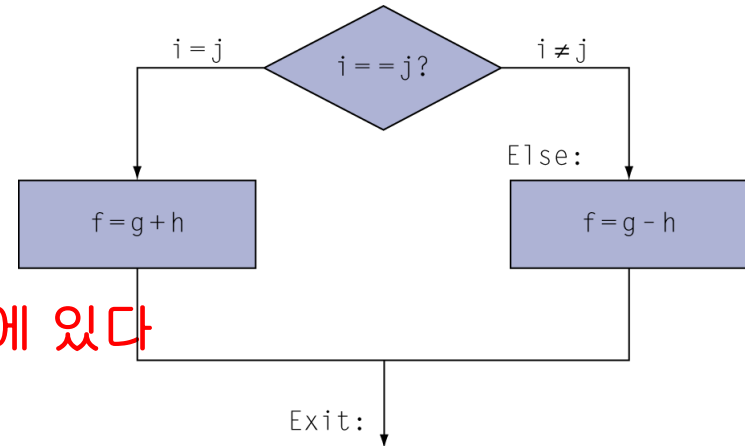
논리	and	AND r1, r2, r3	r1 = r2 & r3	레지스터 피연산자 3개; 비트 대 비트 AND
	or	ORR r1, r2, r3	r1 = r2 r3	레지스터 피연산자 3개; 비트 대 비트 OR
	not	MVN r1, r2	r1 = ~ r2	레지스터 피연산자 3개; 비트 대 비트 NOT
	logical logical shift left (optional operation)	LSL r1, r2, #10	r1 = r2 << 10	상수만큼 좌측 자리이동
	logical shift right (optional operation)	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 = 25*4
무조건 분기	branch (always)	B 2500	go to PC + 8 + 10000	분기 = 2500*4
	branch and link	BL 2500	r14 = PC + 4; go to PC + 8 + 10000	프로시저 호출용

If문 컴파일하기

- **C code:**

```
if (i==j) f = g+h;  
else f = g-h;
```

❖ 변수 **f, g, h, i, j** 가 각기 **r0, ..., r4**에 있다

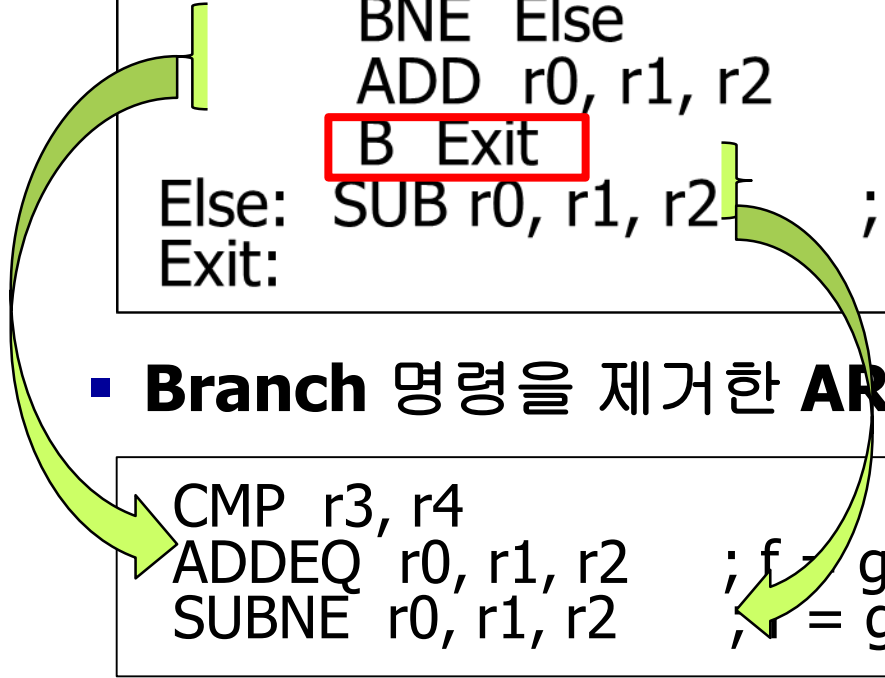


- **Compiled ARM code:**

	CMP	r3, r4	
	BNE	Else	; go to Else if i ≠ j
	ADD	r0, r1, r2	; f = g + h (skipped if i ≠ j)
	B	Exit	; go to Exit
Else	SUB	r0, r1, r2	; f = g - h (skipped if i = j)
Exit			

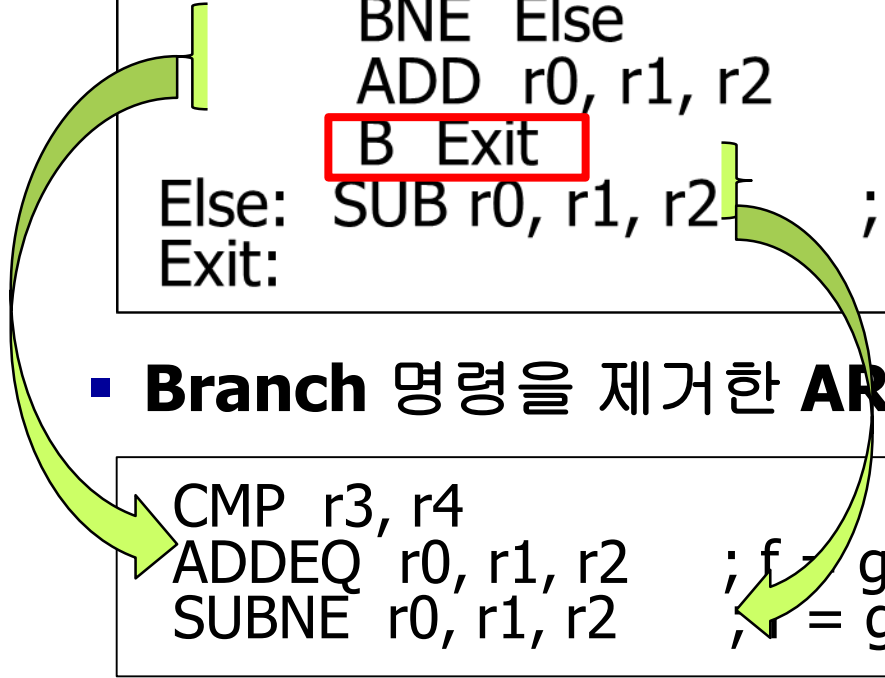
If문 컴파일하기 [Branch 명령 제거]

- if 문장을 **branch** 명령으로 실행한 **ARM Code**



```
CMP r3, r4
BNE Else      ; go to Else if i ≠ j
ADD r0, r1, r2 ; f = g + h (skipped if i ≠ j)
B Exit        ; go to Exit
Else: SUB r0, r1, r2 ; f = g - h (skipped if i = j)
Exit:
```

- **Branch** 명령을 제거한 **ARM Code**




```
CMP r3, r4
ADDEQ r0, r1, r2 ; f = g + h (skipped if i ≠ j)
SUBNE r0, r1, r2 ; f = g - h (skipped if i = j)
```

- **Branch** 명령이 파이프라인 프로세서의 성능을 떨어뜨린다.
→ **Branch** 명령을 제거하는 것이 좋다.

Loop문 컴파일하기

■ **C code:**

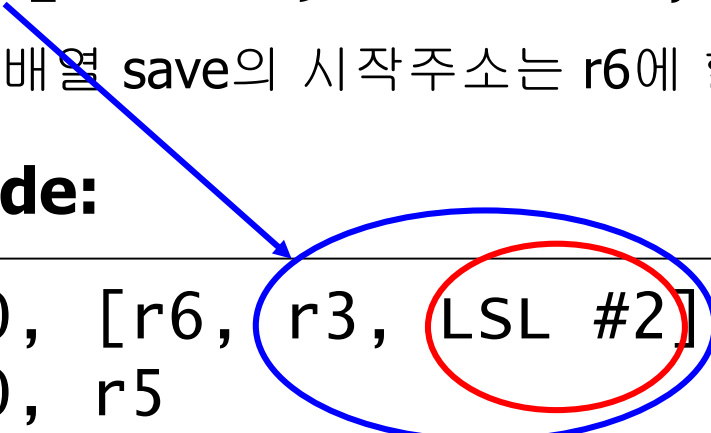
```
while (save[i] == k) i += 1;
```



❖ 변수 i는 r3, k는 r5, 배열 save의 시작주소는 r6에 할당되었다고 가정

■ **Compiled ARM code:**

```
Loop    LDR    r0, [r6, r3, LSL #2]
        CMP    r0, r5
        BNE    Exit
        ADD    r3, r3, #1
        B      Loop
Exit
```



Loop문 컴파일하기 [Branch 명령 제거]

■ C code:

```
while (save[i] == k) i += 1;
```

❖ 변수 i는 r3, k는 r5, 배열 save의 시작주소는 r6에 할당되었다고 가정

■ Compiled ARM code:

```
Loop    LDR     r0, [r6, r3, LSL #2]
        CMP     r0, r5
        ADDEQ   r3, r3, #1
        BEQ     Loop
Exit
```

2장-Part 3. Contents

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

- 간단한 코드 최적화
- 함수호출
- **Sorting** 예제

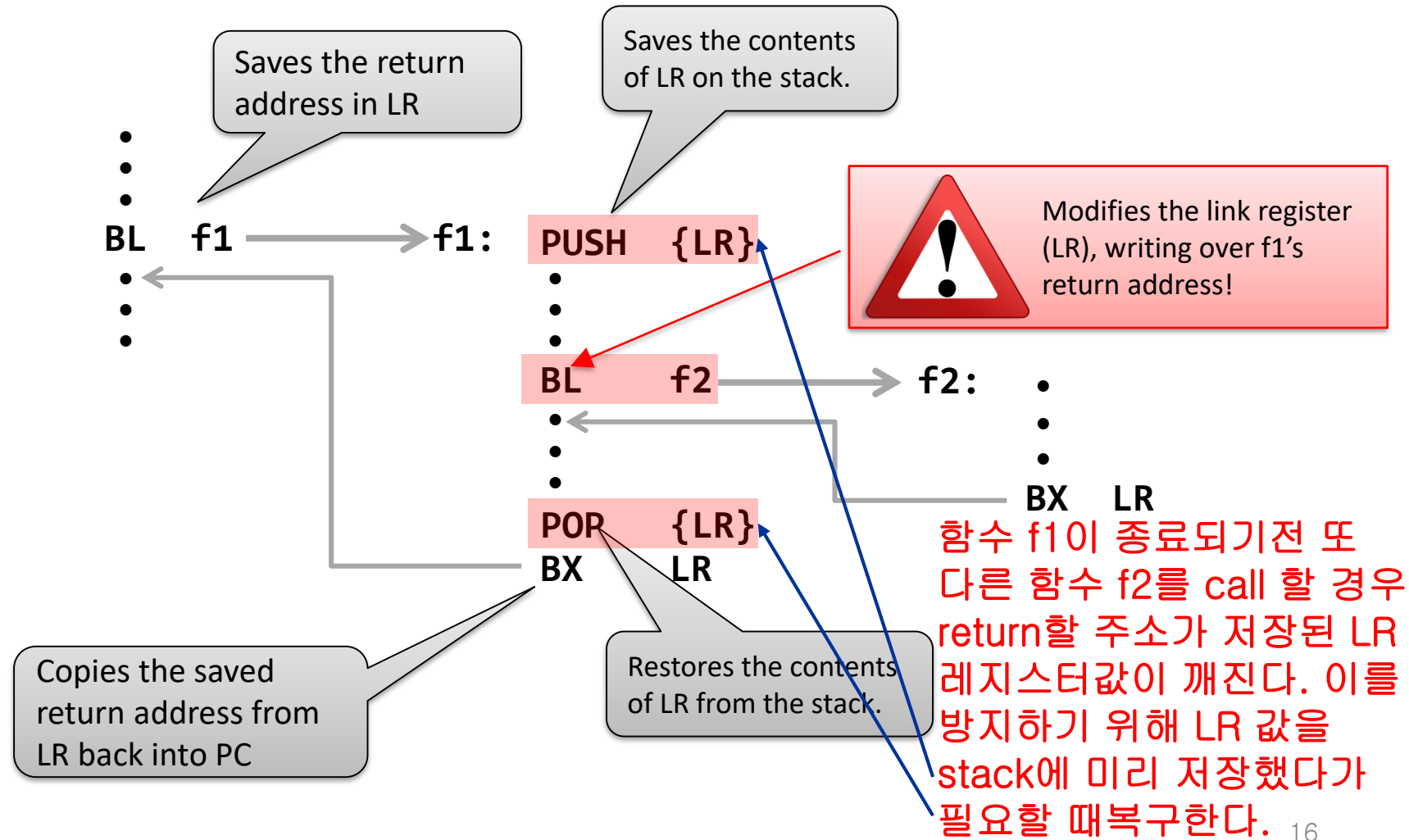
함수 호출

- 스택을 이용한 **Procedure call** (함수호출)
- **Recursive call** (중첩 호출)

함수 호출 과정

- 호출자 함수(**caller**)가 인수 값들을 **r0-r3**에 넣는다.
- **BL X**를 사용하여 함수 **X (callee)**로 점프한다.
- 피호출 함수(**callee**)는 계산을 수행한다.
- 수행 결과를 **r0** 와 **r1**에 넣는다.
- **MOV pc, lr**을 이용하여 호출자 함수에게 제어를 넘긴다.
- 여기 **lr**에는 호출자 함수가 **BL X** 수행시 **BL X** 다음 명령어 번지가 들어가 있음. 예를 들어 **BL X**가 **1000**번지에 있었다면 그 다음 명령어는 **1004**번지에 들어 있으므로 **lr**에는 **1004**가 들어가 있음.

[복습] PUSH(STMFD)/POP(LDMFD) 사용



PUSH/POP 사용

- Return from a leaf subroutine call

- ❖ **MOV pc, r14 ; r14 = LR** 인데, **BX LR** 명령어와 동일한 결과

- Return from nested subroutine call

- ❖ **f1** **PUSH {r4-r6, r14}** ; save work regs and link

- # **PUSH** 명령어 수행하면 **f2**를 **call** 하기전 **r14** 뿐만 아니라 **r4-r6**를

- # 스택에 저장. **r14**를 저장한 이유는 **f1 call**한 다음 명령어 번지로

- # 정확하게 **return**하기 위함. **r4-r6** 저장한 이유는 **f1** 수행시

- # 그 레지스터들을 사용하는데 **f1** 수행 끝나고 **return**시 **f1** 들어올 때

- # **r4-r6** 값을 원래 값대로 복원하기 위함

BL f2

.....

- POP {r4-r6, pc}** ; restore work regs and return

- # **POP** 명령어 수행하면 **PC**에는 스택에 저장되었던 **r14** 값이 들어감.

- # 이는 **MOV PC, LR** 또는 **BX LR** 명령어 사용한 것과 동일한 효과

스택의 필요성

- 레지스터 스�필링 (**Register Spilling**) – 레지스터 내용을 메모리에 저장하기
 - ❖ 한 함수가 사용한 모든 레지스터를 해당 함수가 호출되기 전 상태로 깨끗이 복구해 놓은 방법
 - ❖ 레지스터 스�필링을 위한 이상적인 자료 구조는 스택이다
- **ARM의 레지스터 스�필링**
 - ❖ ARM에서 구동되는 소프트웨어는 레지스터를 두 종류로 분리
 - ◆ r0-r3, r12 : 함수 호출 시, 피호출 함수(callee)가 값을 보존해 주지 않는 인수 또는 스크래치 레지스터
 - ◆ **r4-r11 : 함수 호출 전과 후에 레지스터 값이 동일하게 유지되어야 하는 변수 레지스터 8개(피호출 함수가 이 레지스터를 사용하면 원래 값을 저장했다가 원상 복구해야 한다.)**

[복습] APCS

■ ARM Procedure Call Standard

레지스터	APCS	역 할	비 고
r0	a1	argument 1 / integer result / scratch register	Caller save
r1	a2	argument 2 / scratch register	Caller save
r2	a3	argument 3 / scratch register	Caller save
r3	a4	argument 4 / scratch register	Caller save
r4	v1	register variable 1	Callee save
r5	v2	register variable 2	Callee save
r6	v3	register variable 3	Callee save
r7	v4	register variable 4	Callee save
r8	v5	register variable 5	Callee save
r9	sb/v6	static base / register variable 6	Callee save
r10	sl/v7	stack limit / register variable 7	Callee save
r11	fp	frame pointer	Callee save
r12	ip	scratch reg. / new sb in inter-link-unit calls	Caller save
r13	sp	Lower end of current stack frame	-
r14	lr	link address / scratch register	Caller save
r15	pc	program counter	-

APCS 이해

The diagram illustrates the APCS (Application Binary Interface) conventions for variable storage. It shows a global variable declaration and a function definition with various annotations explaining where data is stored based on the number of arguments and local variables.

```
Int gv1 = 100 ;  
  
int function ( int a1, int a2, int a3, int a4, int a5, int a6, int a7 )  
{  
    int v1, int v2, int v3, int v4, int v5, int v6, int v7 ;  
    int v8, int v9, int v10 ;  
    int ret;  
    .  
    .  
    .  
    v10 = gv1 + v4 ;  
    .  
    .  
    .  
    ret = v7 - v1 ;  
    return ret ;  
}
```

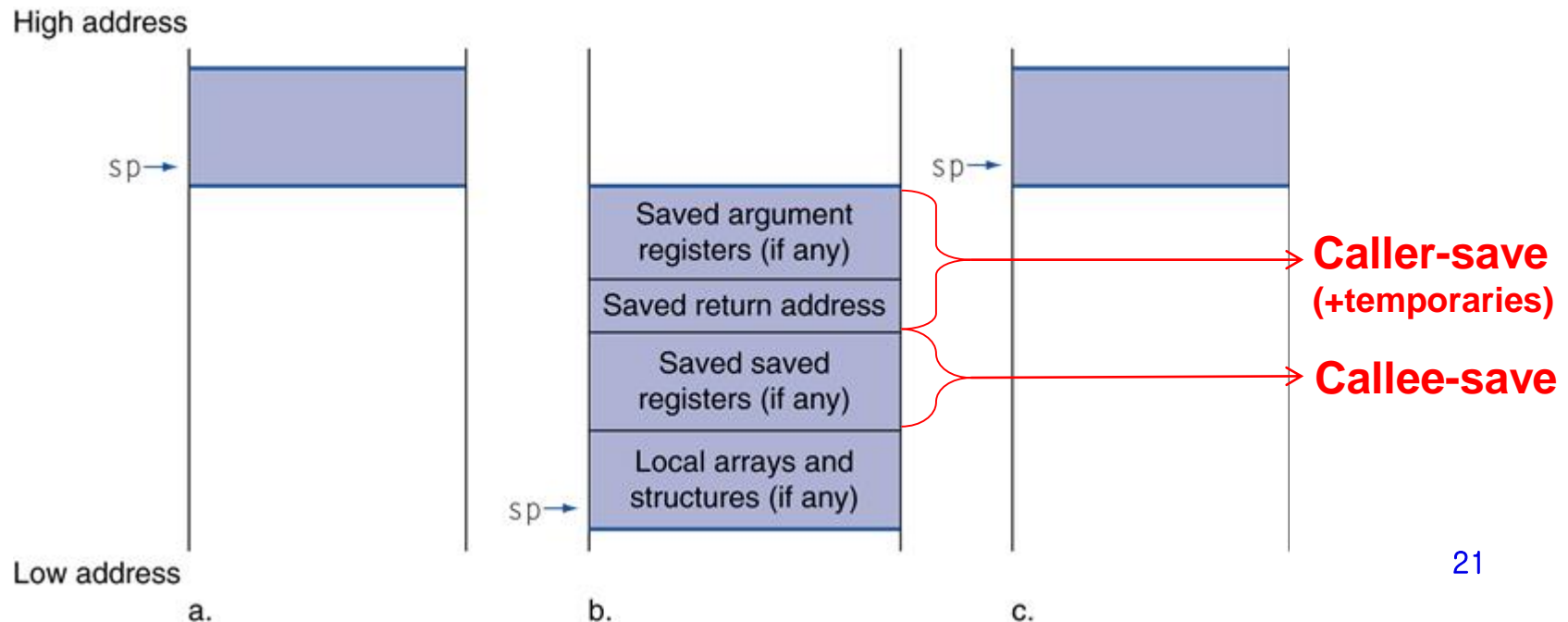
Annotations and their corresponding code elements:

- 글로벌 변수는 RW 데이터 영역에 할당** (Global variables are assigned to the RW data area) points to `Int gv1 = 100 ;`.
- r0 ~ r3의 함수 인자 용 레지스터에 할당** (Registers r0 ~ r3 are used for function arguments) points to the first four arguments: `int a1, int a2, int a3, int a4`.
- 4개 이상의 인자는 스택에 할당** (Arguments with 4 or more are assigned to the stack) points to the last three arguments: `int a5, int a6, int a7`.
- 로컬 변수의 사용량이 늘어나면 스택에 할당** (Local variables are assigned to the stack as usage increases) points to the first three local variables: `int v1, int v2, int v3`.
- r4 ~ r10의 변수용 레지스터에 할당** (Registers r4 ~ r10 are used for variables) points to the next three local variables: `int v4, int v5, int v6`.
- r0를 통해서 결과 값 전달** (Result value is passed through r0) points to the `return ret ;` statement.

스택의 구조

■ 스택 포인터(Stack Pointer)

- ❖ 스택 포인터는 'sp'로 표시하며, reg13에 할당
- ❖ 스택에 데이터를 넣은 작업을 'push'
- ❖ 스택에서 데이터를 꺼내는 작업을 'pop'
- ❖ 스택은 높은 주소에서 낮은 주소 쪽으로 성장
 - ◆ 스택에 데이터를 푸시할 때는 스택 포인터 값을 감소,
 - ◆ 스택에서 데이터를 팝할 때는 스택 포인터 값을 증가



최하단(Leaf) 프로시저 예

■ C code:

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

- ❖ 함수 인자 g, \dots, j 는 레지스터 $r0, \dots, r3$ 에 할당된다
- ❖ 자동 변수 f 는 레지스터 $r4$ 에 할당된다
- ❖ 함수 리턴값은 $r0$ 에 저장한다

최하단(Leaf) 프로시저 예

leaf_example:		
SUB sp, sp, #12		프로시저 레이블
STR r6, [sp, #8]		스택에 3개 워드(12bytes) 저장할 자리
STR r5, [sp, #4]		나중에 사용할 수 있도록 r4, r5, r6 값을
STR r4, [sp, #0]		스택에 임시 저장
ADD r5, r0, r1		
ADD r6, r2, r3		
SUB r4, r5, r6		프로시저 연산 과정
MOV r0, r4		return f; 과정, 최종 결과값을 r0에 복사
LDR r4, [sp, #0]		
LDR r5, [sp, #4]		임시로 저장해 두었던 값을 스택에서
LDR r6, [sp, #8]		꺼내어 레지스터로 원상 복귀
ADD sp, sp, #12		스택 포인터 3개 워드를 delete
MOV pc, 1r		Jump back to calling routine

PUSH {r4, r5, r6}와 동일

POP {r4, r5, r6}와 동일

최하단(Leaf) 프로시저 예 [STM/LDM 버전]

leaf_example:
SUB sp, sp, #12
STR r6, [sp, #8]
STR r5, [sp, #4]
STR r4, [sp, #0]
ADD r5, r0, r1
ADD r6, r2, r3
SUB r4, r5, r6
MOV r0, r4
LDR r4, [sp, #0]
LDR r5, [sp, #4]
LDR r6, [sp, #8]
ADD sp, sp, #12
MOV pc, lr

leaf_example:
STMFD sp!, {r4-r6}
ADD r5, r0, r1
ADD r6, r2, r3
SUB r4, r5, r6
MOV r0, r4
LDMFD sp!, {r4-r6}
MOV pc, lr

STMFD sp! 대신 **PUSH**, LDMFD
sp! 대신 **POP**을 사용해도 됩니다.

비최하단(Non-Leaf) 프로시저

- 다른 프로시저들을 호출하는 프로시저들
- 중첩된 호출인 경우, 호출자는 스택에 다음 항목들을 저장해야 한다:
 - ❖ 호출자의 복귀 주소
 - ❖ 프로시저 호출 이후에 필요한 파라미터들과 임시 계산값들
- 프로시저 호출 후에 스택으로부터 상기 항목들을 복구해야 한다
- **recursive factorial** 함수 코드를 이용하여 스택에 저장, 복구하는 사례를 학습한다.

비최하단(Non-Leaf) 프로시저 예

■ C code:

```
int fact (int n)
{
    if (n < 1) return 1;
    else return n * fact(n - 1);
}
```

- ❖ 함수 인자 n은 r0에 할당된다
- ❖ 함수 리턴값은 r0에 할당된다

비최하단(Non-Leaf) 프로시저 예

ARM code:

비최하단 프로시저 코드		fact	
비최하단 프로시저 코드	{	SUB	sp, sp, #8 ; adjust sp to push 2 items
		STR	lr, [sp, #4] ; save the return address
		STR	r0, [sp, #0] ; save the argument n
		CMP	r0, #1 ; compare n to 1
최하단 프 로시저 코 드	{	BGE	L1 ; if n >= 1, go to L1
		MOV	r0, #1 ; return 1
		ADD	sp, sp, #8 ; pop 2 items off stack
		MOV	pc, lr ; return to the caller
비최하단 프로시저 코드	L1	SUB	r0, r0, #1 ; n >= 1: argument gets (n-1)
		BL	fact ; call fact with (n-1)
	{	MOV	r12, r0 ; save the return value
		LDR	r0, [sp, #0] ; restore argument n
		LDR	lr, [sp, #4] ; restore the return address
		ADD	sp, sp, #8 ; adjust sp to pop 2 items
	{	MUL	r0, r0, r12 ; return n * fact(n-1)
		MOV	pc, lr ; return to the caller

return n * fact(n - 1);

비최하단(Non-Leaf) 프로시저 예 [STM/LDM 버전]

fact		
	SUB	sp, sp, #8
	STR	lr, [sp, #4]
	STR	r0, [sp, #0]
	CMP	r0, #1
	BGE	L1
	MOV	r0, #1
	ADD	sp, sp, #8
	MOV	pc, lr
L1	SUB	r0, r0, #1
	BL	fact
	MOV	r12, r0
	LDR	r0, [sp, #0]
	LDR	lr, [sp, #4]
	ADD	sp, sp, #8
	MUL	r0, r0, r12
	MOV	pc, lr

fact		
	STMFD	sp!, {r0, lr}
	CMP	r0, #1
	BGE	L1
	MOV	r0, #1
	ADD	sp, sp, #8
	MOV	pc, lr
L1	SUB	r0, r0, #1
	BL	fact
	MOV	r12, r0
	LDMFD	sp!, {r0, lr}
	MUL	r0, r0, r12
	MOV	pc, lr

STMFD sp! 대신 PUSH, LDMFD
sp! 대신 POP을 사용해도 됩니다.

비최하단(Non-Leaf) 프로시저 예

```
int main() {
    fact(2);
    printf();
}
```

fact(2)

fact		
2	SUB	sp, sp, #8
	STR	lr, [sp, #4]
	STR	r0, [sp, #0]
	CMP	r0, #1
	BGE	L1
	MOV	r0, #1
	ADD	sp, sp, #8
	MOV	pc, lr
	L1	SUB r0, r0, #1
		BL fact
	MOV	r12, r0
11	LDR	r0, [sp, #0]
	LDR	lr, [sp, #4]
	ADD	sp, sp, #8
	MUL	r0, r0, r12
	MOV	pc, lr

2 x fact(1)

fact(1)

fact		
4	SUB	sp, sp, #8
	STR	lr, [sp, #4]
	STR	r0, [sp, #0]
	CMP	r0, #1
	BGE	L1
	MOV	r0, #1
	ADD	sp, sp, #8
	MOV	pc, lr
	L1	SUB r0, r0, #1
		BL fact
	MOV	r12, r0
9	LDR	r0, [sp, #0]
	LDR	lr, [sp, #4]
	ADD	sp, sp, #8
	MUL	r0, r0, r12
	MOV	pc, lr

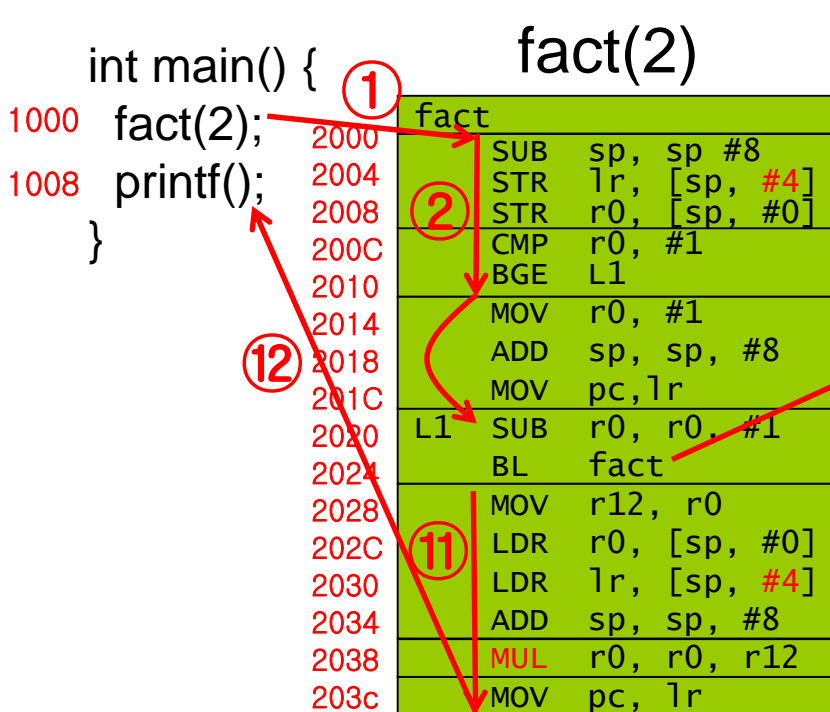
1 x fact(0)

fact(0)

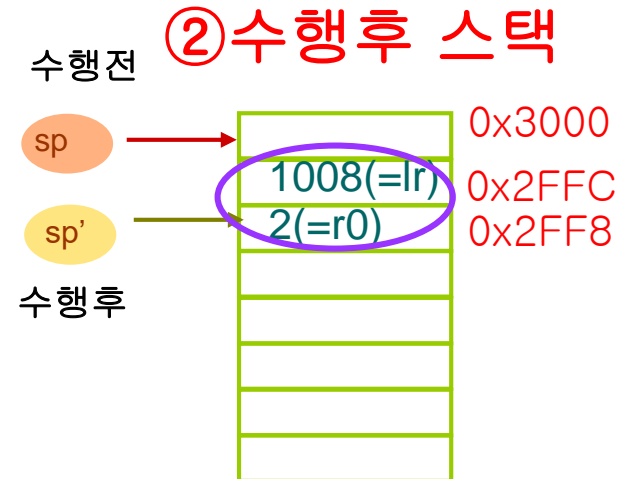
fact		
6	SUB	sp, sp, #8
	STR	lr, [sp, #4]
	STR	r0, [sp, #0]
	CMP	r0, #1
	BGE	L1
7	MOV	r0, #1
	ADD	sp, sp, #8
	MOV	pc, lr
	L1	SUB r0, r0, #1
		BL fact
	MOV	r12, r0
	LDR	r0, [sp, #0]
	LDR	lr, [sp, #4]
	ADD	sp, sp, #8
	MUL	r0, r0, r12
	MOV	pc, lr

return 1

비최하단(Non-Leaf) 프로시저 예



2 x fact(1)



- main C 코드, fact 어셈블리 코드, 스택의 각각 왼쪽 또는 오른쪽 빨간 숫자는 메모리 주소를 나타냄
- 가정: main C 코드는 메모리 1000번지부터, fact() 어셈블리 코드는 2000번지부터, 스택포인터는 3000번지에 위치
- ① main C 코드에서 fact(2) call
- ② fact(2) 어셈블리 코드에서 2000~2008 번지 명령어 수행후 스택모습이 상단 그림

비최하단(Non-Leaf) 프로시저 예

fact(1)

2000
2004
2008
200C
2010
2014
2018
201C
2020
2024
2028
202C
2030
2034
2038
203C

fact		
③	SUB	sp, sp, #8
④	STR	lr, [sp, #4]
	STR	r0, [sp, #0]
	CMP	r0, #1
	BGE	L1
	MOV	r0, #1
	ADD	sp, sp, #8
	MOV	pc, lr
	L1	SUB r0, r0, #1
		BL fact
⑧	MOV	r12, r0
⑨	LDR	r0, [sp, #0]
	LDR	lr, [sp, #4]
	ADD	sp, sp, #8
	MUL	r0, r0, r12
	MOV	pc, lr

1 x fact(0)

fact(0)

fact		
⑤	SUB	sp, sp, #8
⑥	STR	lr, [sp, #4]
	STR	r0, [sp, #0]
	CMP	r0, #1
	BGE	L1
⑦	MOV	r0, #1
	ADD	sp, sp, #8
	MOV	pc, lr
	L1	SUB r0, r0, #1
		BL fact
	MOV	r12, r0
	LDR	r0, [sp, #0]
	LDR	lr, [sp, #4]
	ADD	sp, sp, #8
	MUL	r0, r0, r12
	MOV	pc, lr

return 1

②수행후 스택

1008(=lr)	0x3000
2(=r0)	0x2FFC
2028(=lr)	0x2FF8
1(=r0)	0x2FF4
2028(=lr)	0x2FF0
0(=r0)	0x2FEC
	0x2FE8

④수행후 스택

⑥수행후 스택

SP

⑦수행후 스택

SP

⑨수행후 스택

r0=1, lr=2028

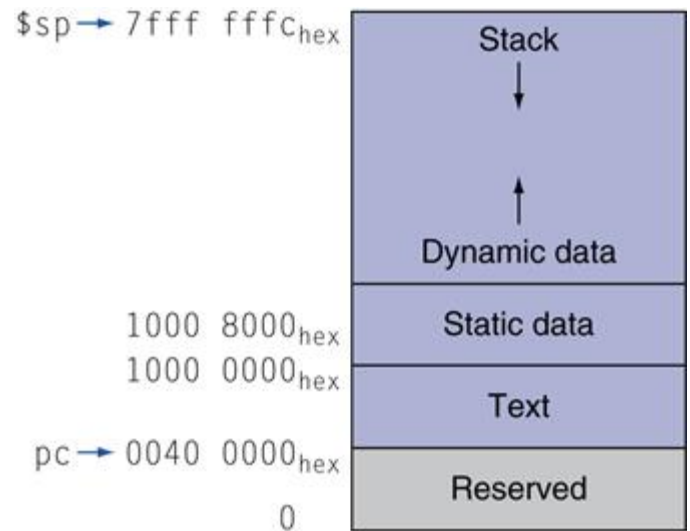
SP

1008(=lr)	0x3000
2(=r0)	0x2FFC
2028(=lr)	0x2FF8
1(=r0)	0x2FF4
2028(=lr)	0x2FF0
0(=r0)	0x2FEC
	0x2FE8

1008(=lr)	0x3000
2(=r0)	0x2FFC
2028(=lr)	0x2FF8
1(=r0)	0x2FF4
2028(=lr)	0x2FF0
0(=r0)	0x2FEC
	0x2FE8

메모리 배치

- 텍스트: 프로그램 기계어 코드
- 정적 데이터: 전역 변수들
 - ❖ 예: C 정적 변수들, 상수 배열들, 문자열들
- 동적 데이터: 힙(heap)
 - ❖ 예: C의 malloc 함수, Java의 new 연산자
 - ❖ Memory leak: free 함수
- 스택: 자동 저장소 (automatic storage)

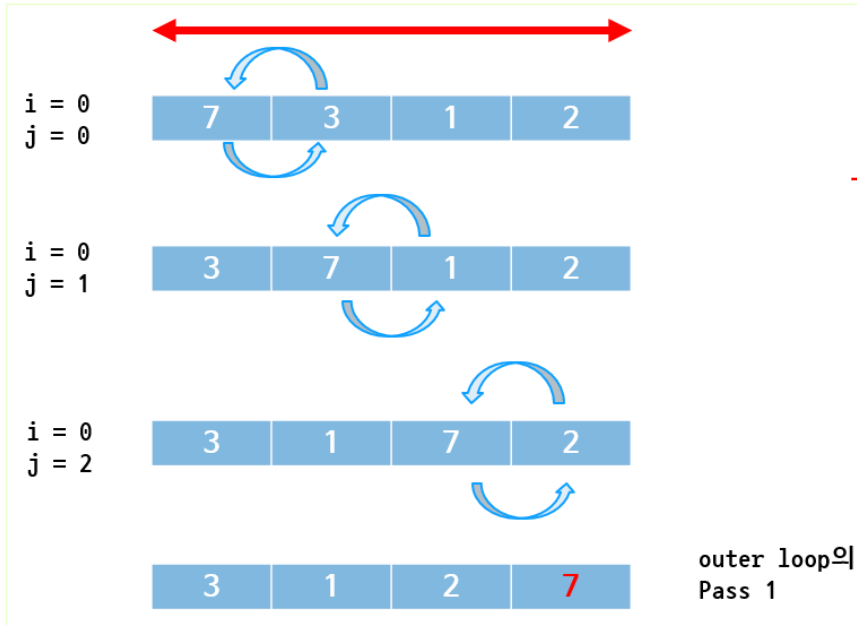


2장-Part 3. Contents

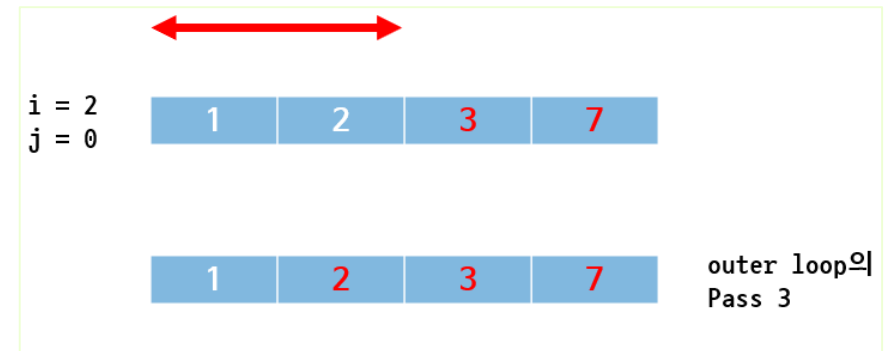
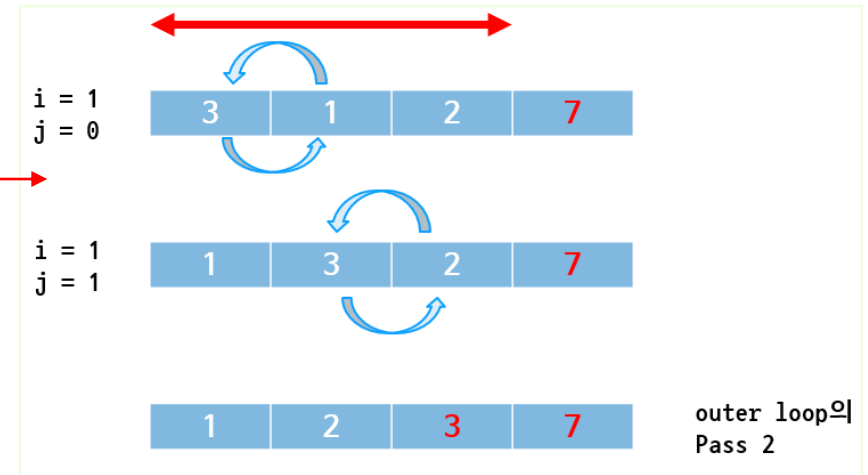
Part3: 코드 최적화 및 **ARM assembly program** 예제

- 간단한 코드 최적화
- 함수호출
- **Sorting** 예제

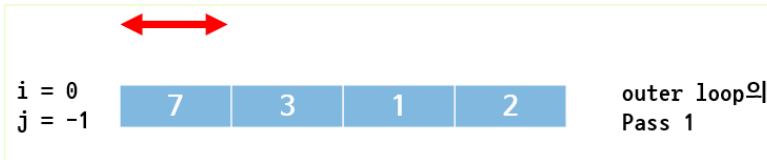
bubble sort(방법1)



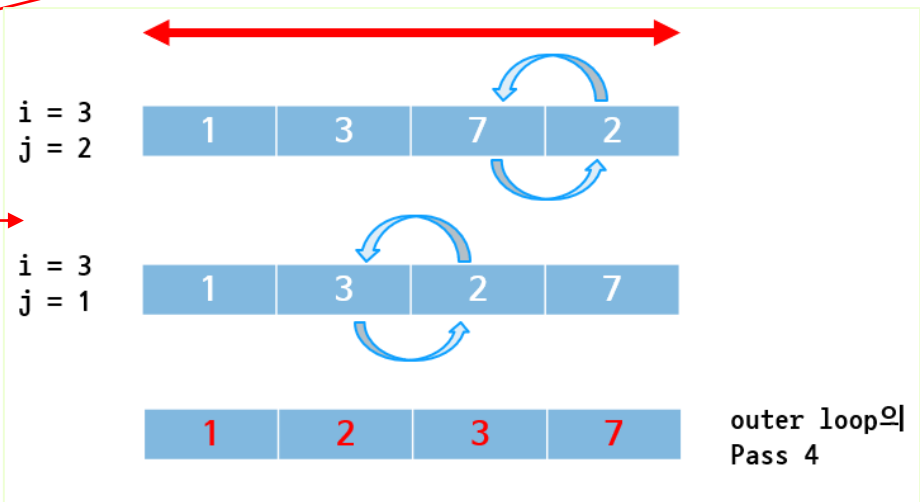
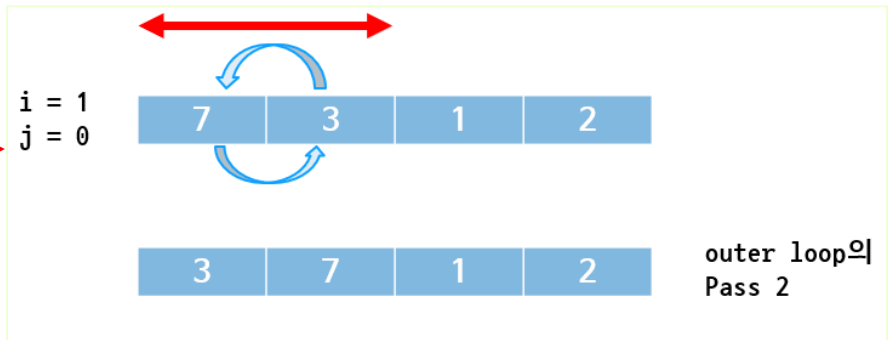
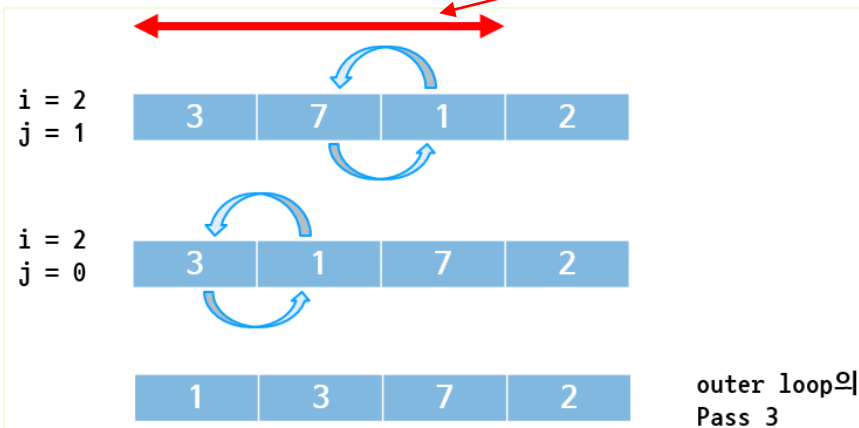
```
void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n-1; i ++){
        for (j = 0; j < n-1-i; j ++){
            if (v[j] > v[j + 1]) swap(v,j);
        }
    }
}
```



bubble sort (방법2-교재내용)



```
void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i++)
        for (j = i - 1; j >= 0 ; j --)
            if (v[j] > v[j + 1]) swap(v,j);
}
```



C Sort Example to Put It All Together

- **C Sort Example**
- **Illustrates use of assembly instructions for a C bubble sort function**
- **Swap procedure (leaf)**

```
void swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

The Procedure Swap

Assembler directive

v	RN 0	; 1st argument address of v
k	RN 1	; 2nd argument index k
temp	RN 2	; local variable
temp2	RN 3	; temporary variable for v[k+1]
vkAddr	RN 12	; to hold address of v[k]

Procedure body

```
swap: ADD    vkAddr, v, k, LSL #2    ; reg vkAddr = v + (k * 4)
                                   ; reg vkAddr has the address of v[k]
      LDR     temp, [vkAddr, #0]      ; temp (temp) = v[k]
      LDR     temp2, [vkAddr, #4]     ; temp2 = v[k + 1]
                                   ; refers to next element of v
      STR     temp2, [vkAddr, #0]     ; v[k] = temp2
      STR     temp, [vkAddr, #4]     ; v[k+1] = temp
```

Procedure return

```
MOV     pc, lr                    ; return to calling routine
```

The Sort Procedure in C

- **Non-leaf (calls swap)**

```
void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i ++ ) {
        for (j = i - 1; j >= 0; j --) {
            if (v[j] > v[j + 1]) swap(v, j);
        }
    }
}
```

이 bubble sorting에 대한 어셈블리코드(P29-P31)는 Lab3에서 실습하게 됨

Register allocation and saving registers for *sort*

Register allocation

v	RN 0	; 1st argument address of v
n	RN 1	; 2nd argument index n
i	RN 2	; local variable i
j	RN 3	; local variable j
vjAddr	RN 12	; to hold address of v[j]
vj	RN 4	; to hold a copy of v[j]
vj1	RN 5	; to hold a copy of v[j+1]
vcopy	RN 6	; to hold a copy of v
ncopy	RN 7	; to hold a copy of n

Saving registers

```
sort:    SUB    sp,sp,#20                ; make room on stack for 5 registers
         STR    lr,[sp,#16]              ; save lr on stack
         STR    ncopy,[sp,#12]           ; save ncopy on stack
         STR    vcopy,[sp,#8]            ; save vcopy on stack
         STR    j,[sp,#4]                ; save j on stack
         STR    i,[sp,#0]                ; save i on stack
```

Procedure body – *sort*

```
void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i++)
        for (j = i - 1; j >= 0; j--)
            if (v[j] > v[j + 1]) swap(v, j);
}
```

Move parameters	<pre>MOV vcopy, v ; copy parameter v into vcopy (save r0) MOV ncopy, n ; copy parameter n into ncopy (save r1)</pre>
Outer loop	<pre>for1tst: MOV i, #0 ; i = 0 CMP i, n ; if i ≥ n BGE exit1 ; go to exit1 if i ≥ n</pre>
Inner loop	<pre>for2tst: SUB j, i, #1 ; j = i - 1 CMP j, #0 ; if j < 0 BLT exit2 ; go to exit2 if j < 0 ADD vjAddr, v, j, LSL #2 ; reg vjAddr = v + (j * 4) LDR vj, [vjAddr, #0] ; reg vj = v[j] LDR vj1, [vjAddr, #4] ; reg vj1 = v[j + 1] CMP vj, vj1 ; if vj ≤ vj1 BLE exit2 ; go to exit2 if vj ≤ vj1</pre>
Pass parameters and call	<pre>MOV r0, vcopy ; first swap parameter is v MOV r1, j ; second swap parameter is j BL swap ; swap code shown in Figure 2.23</pre>
Inner loop	<pre>SUB j, j, #1 ; j -= 1 B for2tst ; branch to test of inner loop</pre>
Outer loop	<pre>exit2: ADD i, i, #1 ; i += 1 B for1tst ; branch to test of outer loop</pre>

Restoring registers and return - *sort*

```
exit1:  LDR    i,  [sp, #0]      ; restore i from stack
        LDR    j,  [sp, #4]      ; restore j from stack
        LDR    vcopy, [sp, #8]   ; restore vcopy from stack
        LDR    ncopy, [sp, #12]  ; restore ncopy from stack
        LDR    lr, [sp, #16]     ; restore lr from stack
        ADD    sp, sp, #20       ; restore stack pointer
```

Procedure return

```
MOV     pc, lr                  ; return to calling routine
```

부록 1. - MIPS 명령어/ARM v8 명령어 소개

■ 배경설명

- ❖ 본 강의 4장부터는 **MIPS** 명령어로 설명 (교재 원본)
- ❖ ARM 버전이 추가로 나와서 처음에 32 비트인 ARM v7 로 설명한 교재가 나왔다가 현재는 64 비트인 ARM v8 버전의 부분집합인 LEGv8 명령어 사용
- ❖ 본 강의 2장은 **ARM v7** 어셈블리 언어 및 프로그래밍을 다룸
- ❖ 버전에 상관없이 교재에서 사용하는 **CPU hardware**는 동일

■ MIPS 명령어 소개

- ❖ ARM 명령어와의 비교

■ ARM v8 & LEG v8 명령어 소개

ARM Instructions and MIPS Instructions

- **ARM: the most popular embedded core**
- **Similar basic set of instructions to MIPS**

	ARM	MIPS
Date announced	1985	1985
Instruction size	32 bits	32 bits
Address space	32-bit flat	32-bit flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Registers	15 × 32-bit	31 × 32-bit
Input/output	Memory mapped	Memory mapped

ARM vs MIPS

교재 4장에서 사용되는 MIPS 명령어

BEQ, B

beq, j

	명령어 이름	ARM	MIPS
레지스터-레지스터	Add	ADD	addu, addiu
	Add (trap if overflow)	ADDS, SWIV	add
	Subtract	SUB	subu
	Subtract (trap if overflow)	SUBS, SWIVS	sub
	Multiply	MUL	mult, multu
	Divide	—	div, divu
	And	AND	and
	Or	ORR	or
	Xor	EOR	xor
	Load high part register	MOVT	lui
	Shift left logical	LSL ¹	sllv, sll
	Shift right logical	LSR ¹	srlv, srl
	Shift right arithmetic	ASR ¹	srav, sra
	Compare	CMP, CMN, TST, TEQ	slt/i, slt/iu
데이터 전송	Load byte signed	LDRSB	lb
	Load byte unsigned	LDRB	lbu
	Load halfword signed	LDRSH	lh
	Load halfword unsigned	LDRH	lhu
	Load word	LDR	lw
	Store byte	STRB	sb
	Store halfword	STRH	sh
	Store word	STR	sw
	Read, write special registers	MRS, MSR	move
	Atomic Exchange	SWP, SWPB	ll;sc

Set on
less than

MIPS 명령어

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Three register operands
	subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Three register operands
	add immediate	addi \$s1,\$s2,100	$\$s1 = \$s2 + 100$	Used to add constants
Data transfer	load word	lw \$s1,100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$	Word from memory to register
	store word	sw \$s1,100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$	Word from register to memory
	load half	lh \$s1,100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$	Halfword memory to register
	store half	sh \$s1,100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$	Halfword register to memory
	load byte	lb \$s1,100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$	Byte from memory to register
	store byte	sb \$s1,100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$	Byte from register to memory
	load upper immed.	lui \$s1,100	$\$s1 = 100 * 2^{16}$	Loads constant in upper 16 bits
Logical	and	and \$s1,\$s2,\$s3	$\$s1 = \$s2 \& \$s3$	Three reg. operands; bit-by-bit AND
	or	or \$s1,\$s2,\$s3	$\$s1 = \$s2 \mid \$s3$	Three reg. operands; bit-by-bit OR
	nor	nor \$s1,\$s2,\$s3	$\$s1 = \sim (\$s2 \mid \$s3)$	Three reg. operands; bit-by-bit NOR
	and immediate	andi \$s1,\$s2,100	$\$s1 = \$s2 \& 100$	Bit-by-bit AND reg with constant
	or immediate	ori \$s1,\$s2,100	$\$s1 = \$s2 \mid 100$	Bit-by-bit OR reg with constant
	shift left logical	sll \$s1,\$s2,10	$\$s1 = \$s2 \ll 10$	Shift left by constant
	shift right logical	srl \$s1,\$s2,10	$\$s1 = \$s2 \gg 10$	Shift right by constant
Conditional branch	branch on equal	beq \$s1,\$s2,25	if ($\$s1 == \$s2$) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne \$s1,\$s2,25	if ($\$s1 \neq \$s2$) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt \$s1,\$s2,\$s3	if ($\$s2 < \$s3$) $\$s1 = 1$; else $\$s1 = 0$	Compare less than; for beq, bne
	set less than immediate	slti \$s1,\$s2,100	if ($\$s2 < 100$) $\$s1 = 1$; else $\$s1 = 0$	Compare less than constant
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr \$ra	go to \$ra	For switch, procedure return
	jump and link	jal 2500	$\$ra = \text{PC} + 4$; go to 10000	For procedure call

MIPS 명령어

MIPS Reference Data

CORE INSTRUCTION SET

NAME	MNE- MON- IC	FOR- MAT	OPERATION (in Verilog)	OPCODE/ PUNCT (Hex)
Add	add	R	$R[rd] = R[rs] + R[rt]$	(1) 0 / 20 _{hex}
Add Immediate	addi	I	$R[rt] = R[rs] + \text{SignExtImm}$	(1)(2) 8 _{hex}
Add Imm. Unsigned	addiu	I	$R[rt] = R[rs] + \text{SignExtImm}$	(2) 9 _{hex}
Add Unsigned	addu	R	$R[rd] = R[rs] + R[rt]$	0 / 21 _{hex}
And	and	R	$R[rd] = R[rs] \& R[rt]$	0 / 24 _{hex}
And Immediate	andi	I	$R[rt] = R[rs] \& \text{ZeroExtImm}$	(3) c _{hex}
Branch On Equal	beq	I	if($R[rs] == R[rt]$) $PC = PC + 4 + \text{BranchAddr}$	(4) 4 _{hex}
Branch On Not Equal	bne	I	if($R[rs] != R[rt]$) $PC = PC + 4 + \text{BranchAddr}$	(4) 5 _{hex}
Jump	j	J	$PC = \text{JumpAddr}$	(5) 2 _{hex}
Jump And Link	jal	J	$R[31] = PC + 8; PC = \text{JumpAddr}$	(5) 3 _{hex}
Jump Register	jr	R	$PC = R[rs]$	0 / 08 _{hex}
Load Byte Unsigned	lbu	I	$R[rt] = \{24'b0, M[R[rs]](7:0)\} + \text{SignExtImm}$	(2) 24 _{hex}
Load Halfword Unsigned	lhu	I	$R[rt] = \{16'b0, M[R[rs]](15:0)\} + \text{SignExtImm}$	(2) 25 _{hex}
Load Upper Imm.	lui	I	$R[rt] = \{imm, 16'b0\}$	5 _{hex}
Load Word	lw	I	$R[rt] = M[R[rs]] + \text{SignExtImm}$	(2) 23 _{hex}
Not	nor	R	$R[rd] = \sim (R[rs] R[rt])$	0 / 27 _{hex}
Or	or	R	$R[rd] = R[rs] R[rt]$	0 / 25 _{hex}
Or Immediate	ori	I	$R[rt] = R[rs] \text{ZeroExtImm}$	(3) d _{hex}
Set Less Than	slt	R	$R[rd] = (R[rs] < R[rt]) ? 1 : 0$	0 / 2a _{hex}
Set Less Than Imm.	slti	I	$R[rt] = (R[rs] < \text{SignExtImm}) ? 1 : 0$	(2) a _{hex}
Set Less Than Imm. Unsigned	sltiu	I	$R[rt] = (R[rs] < \text{SignExtImm}) ? 1 : 0$	(2)(6) b _{hex}
Set Less Than Unsigned	sltu	R	$R[rd] = (R[rs] < R[rt]) ? 1 : 0$	(6) 0 / 2b _{hex}
Shift Left Logical	sll	R	$R[rd] = R[rt] << \text{shamt}$	0 / 00 _{hex}
Shift Right Logical	srl	R	$R[rd] = R[rt] >> \text{shamt}$	0 / 02 _{hex}
Store Byte	sb	I	$M[R[rs] + \text{SignExtImm}](7:0) = R[rt](7:0)$	(2) 28 _{hex}
Store Halfword	sh	I	$M[R[rs] + \text{SignExtImm}](15:0) = R[rt](15:0)$	(2) 29 _{hex}
Store Word	sw	I	$M[R[rs] + \text{SignExtImm}] = R[rt]$	(2) 2b _{hex}
Subtract	sub	R	$R[rd] = R[rs] - R[rt]$	(1) 0 / 22 _{hex}
Subtract Unsigned	subu	R	$R[rd] = R[rs] - R[rt]$	0 / 23 _{hex}

- (1) May cause overflow exception
 (2) SignExtImm = { 16{immediate[15]}, immediate }
 (3) ZeroExtImm = { 16{1b'0}, immediate }
 (4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
 (5) JumpAddr = { PC+4[31:28], address, 2'b0 }
 (6) Operands considered unsigned numbers (vs. 2's comp.)

BASIC INSTRUCTION FORMATS

R	opcode	rs	rt	rd	shamt	funct
	31	26-25	21-20	16-15	11-10	6-5
I	opcode	rs	rt	immediate		
	31	26-25	21-20	16-15		
J	opcode	address				
	31	26-25				

Branch On Equal	beq	I	if($R[rs] == R[rt]$) $PC = PC + 4 + \text{BranchAddr}$	(4)	4 _{hex}
Branch On Not Equal	bne	I	if($R[rs] != R[rt]$) $PC = PC + 4 + \text{BranchAddr}$	(4)	5 _{hex}
Jump	j	J	$PC = \text{JumpAddr}$	(5)	2 _{hex}
Jump And Link	jal	J	$R[31] = PC + 8; PC = \text{JumpAddr}$	(5)	3 _{hex}
Jump Register	jr	R	$PC = R[rs]$		0 / 08 _{hex}

- (1) May cause overflow exception
 (2) SignExtImm = { 16{immediate[15]}, immediate }
 (3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
 (6) Operands considered unsigned numbers (vs. 2's comp.)

MIPS Register Convention

Name	Register Number	Usage
\$zero	0	the constant value 0
\$at	1	reserved for assembler
\$v0-\$v1	2-3	values for results and expression evaluation
\$a0-\$a3	4-7	arguments
\$t0-\$t9	8-15,24-25	temporaries
\$s0-\$s7	16-23	saved
\$k0-\$k1	26-27	reserved for operating system
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address

Example: $A[300] = h + A[300];$

- lw **\$t0**, 1200(\$t1) # Temporary reg. \$t0 gets A[300]
- add **\$t0**, \$s2, \$t0 # Temporary reg. \$t0 gets $h + A[300]$
- sw **\$t0**, 1200(\$t1) # Stores $h + A[300]$ back into A[300]

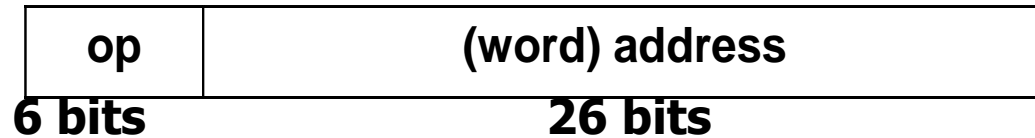
t1 레지스터는 배열 A 시작주소를 가짐

1200은 300 (index 값) x 4 (word 당 차지하는 주소 개수)에서 나온 것임

\$t1 대신 \$s2가 올 수 있음. 이는 레지스터 2번을 의미

Addressing in Branches and Jumps

■ J-type instruction format



■ Pseudo-direct addressing

- ❖ Upper 4 bits of PC are unchanged.
- ❖ Address boundary of 256 MB
- ❖ Jump address



PC의 상위 4 비트는 바꾸지 않고 하위 28비트만 바꾸어서 32 비트 주소 생성

Branch Address

■ PC-relative addressing mode

- ❖ Branch target address = $(PC+4) + \text{Branch offset}$
- ❖ $\text{New PC} = (PC+4) \pm 2^{15} \text{ word}$

op	rs	rt	(word) address
6 bits	5 bits	5 bits	16 bits

부록 2. - ARM v8(64비트 CPU) 명령어 소개

■ ARM v8(64비트 CPU) 명령어 set

- ARM v8(64비트 CPU) 에서 사용하는 명령어
 - 32비트로 표현

ARM v8 Instruction Format

Field	opcode	Rm	shamt	Rn	Rd
Bit positions	31:21	20:16	15:10	9:5	4:0

a. R-type instruction

Field	1986 or 1984	address	0	Rn	Rt
Bit positions	31:21	20:12	11:10	9:5	4:0

b. Load or store instruction

Field	180	address	Rt
Bit positions	31:26	23:5	4:0

c. Conditional branch instruction

FIGURE 4.14 The three instruction classes (R-type, load and store, and conditional branch) use three different instruction formats. The unconditional branch instruction uses another format, which we will discuss shortly. (a) Instruction format for R-format instructions, have three register operands: Rn, Rm, and Rd. Fields Rn and Rm are sources, and Rd is the destination. The ALU function is in the opcode field and is decoded by the ALU control design in the previous section. The R-type instructions that we implement are ADD, SUB, AND, and ORR. The shamt field is used only for shifts; we will ignore it in this chapter. (b) Instruction format for load (opcode = 1986_{ten}) and store (opcode = 1984_{ten}) instructions. The register Rn is the base register that is added to the 9-bit address field to form the memory address. For loads, Rt is the destination register for the loaded value. For stores, Rt is the source register whose value should be stored into memory. (c) Instruction format for compare and branch on zero (opcode = 180). The register Rt is the source register that is tested for zero. The 19-bit address field is sign-extended, shifted, and added to the PC to compute the branch target address.

Full ARM v8 instruction set(1/4)

There is also a version of the MOV wide instructions (MOVN) that complements all 64 bits that are created from the 16-bit constant; that is, the other 48 bits are ones instead of zeros and the 16-bit immediate field is complemented too, which bumps the count to 53.

165

Type	Mnemonic	Instruction	Type	Mnemonic	Instruction
Arithmetic Register	ADD	Add	Logical Immediate	ANDI	Bitwise AND Immediate
	ADDS	Add and set flags		ANDIS	Bitwise AND and set flags Immediate
	SUB	Subtract		ORRI	Bitwise inclusive OR Immediate
	SUBS	Subtract and set flags		EORI	Bitwise exclusive OR Immediate
	CMP	Compare		TSTI	Test bits Immediate
	CMN	Compare negative	Shift Register Shift Immed	LSL	Logical shift left Immediate
	NEG	Negate		LSR	Logical shift right Immediate
	NEGS	Negate and set flags		ASR	Arithmetic shift right Immediate
Arithmetic Immediate	ADDI	Add Immediate		ROR	Rotate right Immediate
	ADDIS	Add and set flags Immediate		LSRV	Logical shift right register
	SUBI	Subtract Immediate		LSLV	Logical shift left register
	SUBIS	Subtract and set flags Immediate		ASRV	Arithmetic shift right register
	CMPI	Compare Immediate		RORV	Rotate right register
	CMNI	Compare negative Immediate	Move Wide Immed iate	MOVZ	Move wide with zero
Arithmetic Extended	ADD	Add Extended Register		MOVK	Move wide with keep
	ADDS	Add and set flags Extended		MOVN	Move wide with NOT
	SUB	Subtract Extended Register		MOV	Move register
	SUBS	Subtract and set flags Extended	tract	BFM	Bitfield move
	CMP	Compare Extended Register		SBFM	Signed bitfield move
	CMN	Compare negative Extended		UBFM	Unsigned bitfield move (32-bit)

Full ARM v8 instruction set(2/4)

Arithmetic with Carry	ADC	Add with carry	Bit Field Insert & Extract	BFI	Bitfield insert
	ADCS	Add with carry and set flags		BFXIL	Bitfield extract and insert low
	SBC	Subtract with carry		SBFIZ	Signed bitfield insert in zero
	SBCS	Subtract with carry and set flags		SBFX	Signed bitfield extract
	NGC	Negate with carry		UBFIZ	Unsigned bitfield insert in zero
	NGCS	Negate with carry and set flags		UBFX	Unsigned bitfield extract
Logical Register	AND	Bitwise AND	Sign Extend	EXTR	Extract register from pair
	ANDS	Bitwise AND and set flags		SXTB	Sign-extend byte
	ORR	Bitwise inclusive OR		SXTH	Sign-extend halfword
	EOR	Bitwise exclusive OR		SXTW	Sign-extend word
	BIC	Bitwise bit clear		UXTB	Unsigned extend byte
	BICS	Bitwise bit clear and set flags		UXTH	Unsigned extend halfword
	ORN	Bitwise inclusive OR NOT	Bit Operation	CLS	Count leading sign bits
	EON	Bitwise exclusive OR NOT		CLZ	Count leading zero bits
	MVN	Bitwise NOT		RBIT	Reverse bit order
	TST	Test bits		REV	Reverse bytes in register
				REV16	Reverse bytes in halfwords
				REV32	Reverses bytes in words

FIGURE 2.41 The list of assembly language instructions for the integer operations in the full ARMv8 instruction set.

To manipulate fields of bits, the full ARMv8 instruction set includes 10 instructions that can extract a bit field from a register and insert it into another.

Full ARM v8 instruction set(3/4)

The final set of data transfer instructions for integers perform exclusive access to memory in multiprocessor environments. We saw two examples earlier in `LDXR` and `STXR`. In addition to providing exclusive access to doublewords, there are

Type	Mnemonic	Description
Unscaled	<code>LDUR</code>	Load register (unscaled offset)
	<code>LDURB</code>	Load byte (unscaled offset)
	<code>LDURSB</code>	Load signed byte (unscaled offset)
	<code>LDURH</code>	Load halfword (unscaled offset)
	<code>LDURSH</code>	Load signed halfword (unscaled offset)
	<code>LDURSW</code>	Load signed word (unscaled offset)
	<code>STUR</code>	Store register (unscaled offset)
	<code>STURB</code>	Store byte (unscaled offset)
	<code>STURH</code>	Store halfword (unscaled offset)
	<code>STURW</code>	Store word (unscaled offset)
Scaled, Extended, Pre- & Post-Indexed	<code>LDA</code>	Load address
	<code>LDR</code>	Load register
	<code>LDRB</code>	Load byte
	<code>LDRSB</code>	Load signed byte
	<code>LDRH</code>	Load halfword
	<code>LDRSH</code>	Load signed halfword
	<code>LDRSW</code>	Load signed word
	<code>STR</code>	Store register
	<code>STRB</code>	Store byte
	<code>STRH</code>	Store halfword
Exclusive	<code>LDXR</code>	Load Exclusive register
	<code>LDXRB</code>	Load Exclusive byte
	<code>LDXRH</code>	Load Exclusive halfword
	<code>LDXP</code>	Load Exclusive Pair
	<code>STXR</code>	Store Exclusive register
	<code>STXRB</code>	Store Exclusive byte
	<code>STXRH</code>	Store Exclusive halfword
	<code>STXP</code>	Store Exclusive Pair
	<code>LDAXR</code>	Load-acquire Exclusive register
	<code>LDAXRB</code>	Load-acquire Exclusive byte
Exclusive Acquire/Release	<code>LDAXRH</code>	Load-acquire Exclusive halfword
	<code>LDAXP</code>	Load-acquire Exclusive Pair
	<code>STLXR</code>	Store-release Exclusive register
	<code>STLXRB</code>	Store-release Exclusive byte
	<code>STLXRH</code>	Store-release Exclusive halfword
	<code>STLXP</code>	Store-release Exclusive Pair
Pair	<code>LDP</code>	Load Pair
	<code>LDPSW</code>	Load Pair signed words
	<code>STP</code>	Store Pair
	<code>ADRP</code>	Compute address of 4KB page at a PC-relative offset
PC	<code>ADR</code>	Compute address of label at a PC-relative offset

X11 contains 100,000_{ten}

```
LDR X10, [X11, #16] // scaled addressing mode
```

will load the double word (8 bytes) at address 100,128_{ten} (100,000 + 8*16) into register X10.

er data transfer operations in the full
a pseudoinstruction, and bold italic means it is a

Full ARM v8 instruction set(4/4)

The next nine instructions store a value into a register based on the condition codes. If the condition is true, the destination register gets the first register. If not, it gets the second register. The idea behind condition select instructions is to replace conditional branches, which can cause problems in pipelined execution if they can't be predicted (see Chapter 4). After adding the nine condition select instructions, we're up to 19.

The final four instructions are similar to the conditional select instructions, except the destination for these instructions is the condition codes. That is, these

Type	Mnemonic	
Conditional Branch	B . cond	Branch conditionally
	CBNZ	Compare and branch if nonzero
	CBZ	Compare and branch if zero
	TBNZ	Test bit and branch if nonzero
	TBZ	Test bit and branch if zero
Unconditional Branch	B	Branch unconditionally
	BL	Branch with link
	BLR	Branch with link to register
	BR	Branch to register
	RET	Return from subroutine
Conditional Select	<i>CSEL</i>	Conditional select
	<i>CSINC</i>	Conditional select increment
	<i>CSINV</i>	Conditional select inversion
	<i>CSNEG</i>	Conditional select negation
	<i>CSET</i>	Conditional set
	<i>CSETM</i>	Conditional set mask
	<i>CINC</i>	Conditional increment
	<i>CINV</i>	Conditional invert
	<i>CNEG</i>	Conditional negate
Conditional Compare	CCMP	Conditional compare register
	CCMPI	Conditional compare immediate
	CCMN	Conditional compare negative register
	CCMNI	Conditional compare negative immediate

FIGURE 2.43 The list of assembly language instructions for the branches of the ARMv8 instruction set. Bold means the instruction is also in LEGv8, italic means it is a pseudoinstruction, and bold italic means it is a pseudoinstruction that is also in LEGv8.

LEG v8 instruction set(1/2)

LEGv8 instructions	Name	Format	Pseudo LEGv8	Name	Format
add	ADD	R	move	MOV	R
subtract	SUB	R	compare	CMP	R
add immediate	ADDI	I	compare immediate	CMPI	I
subtract immediate	SUBI	I	load address	LDA	M
add and set flags	ADDIS	R			
subtract and set flags	SUBS	R			
add immediate and set flags	ADDIS	I			
subtract immediate and set flags	SUBIS	I			
load register	LDUR	D			
store register	STUR	D			
load signed word	LDURSW	D			
store word	STURW	D			
load half	LDURH	D			
store half	STURH	D			
load byte	LDURB	D			
store byte	STURB	D			
load exclusive register	LDXR	D			
store exclusive register	STXR	D			

LEG v8 instruction set(2/2)

move wide with zero	MOVZ	IM
move wide with keep	MOVK	IM
and	AND	R
inclusive or	ORR	R
exclusive or	EOR	R
and immediate	ANDI	I
inclusive or immediate	ORRI	I
exclusive or immediate	EORI	I
logical shift left	LSL	R
logical shift right	LSR	R
compare and branch on equal 0	CBZ	CB
compare and branch on not equal 0	CBNZ	CB
branch conditionally	B.cond	CB
branch	B	B
branch to register	BR	R
branch with link	BL	B

FIGURE 2.45 The LEGv8 instruction set covered so far, with the real LEGv8 instructions on the left and the pseudoinstructions on the right. Section 2.19 describes the full ARMv8 architecture. Figure 2.1 shows more details of the LEGv8 architecture revealed in this chapter. The information given here is also found in Columns 1 and 2 of the LEGv8 Reference Data Card at the front of the book.

Register Operands

- **Arithmetic instructions use register operands**
- **LEGv8 has a 32×64 -bit register file**
 - ❖ Use for frequently accessed data
 - ❖ 64-bit data is called a “doubleword”
 - ◆ 31 x 64-bit general purpose registers X0 to X30
 - ❖ 32-bit data called a “word”
 - ◆ 31 x 32-bit general purpose sub-registers W0 to W30

LEGv8 Registers

- **X0 – X7: procedure arguments/results**
- **X8: indirect result location register**
- **X9 – X15: temporaries**
- **X16 – X17 (IP0 – IP1): may be used by linker as a scratch register, other times as temporary register**
- **X18: platform register for platform independent code; otherwise a temporary register**
- **X19 – X27: saved**
- **X28 (SP): stack pointer**
- **X29 (FP): frame pointer**
- **X30 (LR): link register (return address)**
- **XZR (register 31): the constant value 0**

Register Operand Example

- **C code:**

f = (g + h) - (i + j);

❖ f, ..., j in X19, X20, ..., X23

- **Compiled LEGv8 code:**

ADD X9, X20, X21

ADD X10, X22, X23

SUB X19, X9, X10

ARM v8(64비트 CPU) 명령어 갯수

within the same architecture. In assembly language, programmers use registers named W0, W1, ... instead of the X0, X1, ... to specify 32-bit operations. Thus, this 64-bit operation

```
ADD    X9,X21,X9
```

can be turned into a 32-bit operation by writing instead

```
ADD    W9,W21,W9
```

Figure 2.40 counts this as one assembly language instruction but two machine language instructions since they have different opcodes.

Class	Loads/Stores		Operations		Branches		Total	
	AL	ML	AL	ML	AL	ML	AL	ML
Integer	49	145	74	105	--	--	123	250
Floating Point & Int Mul/Div	0	18	63	156	--	--	63	174
SIMD/Vector	16	166	229	371	--	--	245	537
System/Special	11	55	52	40	--	--	63	95
—	—	—	--	--	23	14	23	14
Total	76	384	418	672	23	14	517	1070

- ❖ AL: counts for Assembly Language instructions
- ❖ ML: counts for Machine Language instructions

ARM vs MIPS

ARM v7

MIPS (교재
4장에서 사용)

ARM v8

BEQ, B

beq, j

❖ ADD

❖ SUB

❖ AND

❖ ORR 레지스터-레지스터

❖ CBZ
(compare
& branch
on zero)

❖ LDUR 데이터 전송

❖ STUR

명령어 이름	ARM	MIPS
Add	ADD	addu, addiu
Add (trap if overflow)	ADDS, SWIV	add
Subtract	SUB	subu
Subtract (trap if overflow)	SUBS, SWIVS	sub
Multiply	MUL	mult, multu
Divide	—	div, divu
And	AND	and
Or	ORR	or
Xor	EOR	xor
Load high part register	MOVT	lui
Shift left logical	LSL ¹	sllv, sll
Shift right logical	LSR ¹	srlv, srl
Shift right arithmetic	ASR ¹	srav, sra
Compare	CMP, CMN, TST, TEQ	slt/i, slt/iu
Load byte signed	LDRSB	lb
Load byte unsigned	LDRB	lbu
Load halfword signed	LDRSH	lh
Load halfword unsigned	LDRH	lhu
Load word	LDR	lw
Store byte	STRB	sb
Store halfword	STRH	sh
Store word	STR	sw
Read, write special registers	MRS, MSR	move
Atomic Exchange	SWP, SWPB	ll;sc

Set on less than
{immediate, unsigned}

Slt \$s1, \$s2, \$s3

If (\$s2 < \$s3) \$s1=1;
else \$s1=0

부록 3 - 2장 내용 복습

어셈블리 프로그래밍 ?

- **C 프로그래밍 vs ARM 어셈블리 프로그래밍**
 - ❖ (예) 자동차 운전, AUTO (자동) 운전 vs STICK(수동) 운전
- **C 프로그래밍에서 추상화 vs 어셈블리 프로그래밍에서 구체화**
 - ❖ C 프로그램에서 사용하는 변수, 배열, **pointer** 등은 추상화의 예제. 이 추상화된 용어에 대한 어셈블리 프로그램에서 구체화된 내용은 메모리에 저장된 값임. 이 값에 접근하려면 메모리 주소가 필요함.
 - ❖ 변수값을 갖고 오는 것은 해당 변수값이 저장된 메모리에서 가서(메모리 주소) 그 값을 갖고 오는 것임. 변수 별로 각기 다른 메모리 번지가 할당됨
 - ❖ 배열, **pointer**도 각각 메모리 번지를 할당함

어셈블리 프로그래밍 – 레지스터 활용

■ 어셈블리 프로그래밍에서 레지스터 활용

- ❖ 성능이 높은 어셈블리 프로그램을 위해서는 매우 중요
- ❖ 명령어에서 **operand**는 메모리 또는 레지스터에 있는 데이터를 사용
- ❖ 레지스터에 있는 값들을 **ALU** 이용하여 연산하는 동작은 **CPU 1** 클럭 소모.
- ❖ 메모리에 있는 값들 연산의 경우는 1) 일단 레지스터로 **copy** 해오는데 **CPU 1** 클럭 소모, 2) 레지스터 값들을 **ALU** 이용하여 연산하는 데 **CPU 1** 클럭 소모. 총 **CPU 2**개 클럭소모
- ❖ 자주 사용하는 변수는 메모리 대신에 레지스터에 위치하게 함. 소모하는 **CPU** 클럭 개수를 줄이는 효과
- ❖ 똑 같은 일을 하는 프로그램 **A**와 **B** 의 성능 비교
 - ◆ 총 기계어 코드 갯수 – 갯수가 작을 수록 적은 숫자의 **CPU** 클럭 소모
 - ◆ 수행도중 메모리 참조(**memory read** 또는 **memory write**) 총 횟수 – 횟수가 작을 수록 적은 숫자의 **CPU** 클럭 소모
 - ◆ **compiler** 에서 코드 최적화한다는 의미는 생성된 코드 총 갯수를 줄이면서 메모리참조 총 횟수를 줄이는 것을 의미함
- ❖ 어셈블리 프로그래밍한다는 것은 **complier**가 하는 것과 동일한 작업임

C 코드의 수행과정 ?

- C 프로그램 **$z = x + y$** 코드의 하드웨어 위에서 수행과정
 - ❖ 기계어 코드로 변환 (by compiler)
 - ❖ 기계어 코드를 수행 (by CPU)
 - ❖ (예) C 코드에서 `int x=1, y=2, z;` 선언후 `z=x + y;` 수행
 - ❖ (예) 변수 **x**가 저장된 주소를 **r3**가 갖고 있다고 가정, 변수 **y**는 변수 **x**가 저장된 주소+4에 있다고 가정, 변수 **z**는 변수 **x**가 저장된 주소+8에 있다고 가정
 - ❖ (예) ARM 어셈블리(명령어) 코드 예제
 - ◆ `LDR r0, [r3]` ; x 갖고 오기 (r3가 x 저장된 메모리 번지 갖는다고 가정)
 - ◆ `LDR r1, [r3, #4]` ; y 갖고 오기
 - ◆ `ADD r2, r0, r1` ; r0와 r1를 더한 결과를 r2에 저장하기
 - ◆ `STR r2, [r3, #8]` ; z에 저장하기
 - ❖ ARM 어셈블리 코드에서 각 명령어 수행과정은 ?

ARM 명령어 수행과정 ?

■ ARM 에서 명령어 수행과정은 ?

1. 명령어 갖고 오기 (**fetch**): (예) 메모리 1000 번지에 저장된 명령어를 CPU 로 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.

각 단계는 CPU 클럭 1개씩을 소모

ARM 구조의 특이점

■ 조건부 명령어 지원

- ❖ (예) ADDEQ r0, r1, r2 ; f = g + h
(skipped if i ≠ j)
- ❖ (예) SUBNE r0, r1, r2 ; f = g - h
(skipped if i = j)
- ❖ Branch 명령어가 나타나면 **Pipelining** 성능이 저하될 수 있다 (4장 내용). 이를 방지하기 위한 방법임

■ Shift 횟수 무관하게 Shift 동작을 하드웨어로 구현

- ❖ (예) ANDR3,R1,R0,ASR 1에서 shift 횟수 1 대신 31이 오더라도 똑 같은 시간내에 수행된다. Shift 횟수에 상관없이 동일한 시간에 수행됨. 그 이유는 shift 동작을 하드웨어로 구현함. 오른쪽 그림에서 Barrel Shifter가 그 역할 담당
- ❖ 레지스터 R0가 ALU 입력단으로 전달되는 중간단계에서 Barrel Shifter가 ASR 1을 수행. Shift 횟수를 31으로 변경해도 Barrel Shifter가 조합회로로 구현되어 있어서 동일한 시간에 처리함

