
임베디드시스템설계 실습 (2)

Embedded System Design

**Real-Time Computing and Communications Lab.
Hanyang University**

목차

1. **APCS**
2. **Structure of an ARM Assembly Module**
3. **General Guide to ARM Assembly**
4. **ARM Instruction Set**
5. **Quiz**

APCS

Banked Registers(ARM)

<i>User and system</i>					
<i>r0</i>					
<i>r1</i>					
<i>r2</i>					
<i>r3</i>					
<i>r4</i>					
<i>r5</i>					
<i>r6</i>					
<i>r7</i>					
<i>r8</i>	<i>r8_fiq</i>				
<i>r9</i>	<i>r9_fiq</i>				
<i>r10</i>	<i>r10_fiq</i>				
<i>r11</i>	<i>r11_fiq</i>				
<i>r12</i>	<i>r12_fiq</i>				
<i>r13 sp</i>	<i>r13_fiq</i>	<i>r13_irq</i>	<i>r13_svc</i>	<i>r13_undef</i>	<i>r13_abt</i>
<i>r14 lr</i>	<i>r14_fiq</i>	<i>r14_irq</i>	<i>r14_svc</i>	<i>r14_undef</i>	<i>r14_abt</i>
<i>r15 pc</i>					
<i>cpsr</i>					
-	<i>spsr_fiq</i>	<i>spsr_irq</i>	<i>spsr_svc</i>	<i>spsr_undef</i>	<i>spsr_abt</i>

APCS

❑ ARM Procedure Call Standard

- Provides a mechanism for writing tightly defined routines which may be interwoven with other routines

❑ APCS defines

- Restrictions on the use of registers
- Conventions for using the stack
- Passing/returning arguments between function calls

Meaning of Registers

Register	APCS	Meaning
r0	a1	argument 1 / integer result / scratch register
r1	a2	argument 2 / scratch register
r2	a3	argument 3 / scratch register
r3	a4	argument 4 / scratch register
r4	v1	register variable
r5	v2	register variable
r6	v3	register variable
r7	v4	register variable
r8	v5	register variable
r9	v6/sb	static base / register variable
r10	v7/sl	stack limit / stack chunk handle / reg. variable
r11	fp	frame pointer
r12	ip	scratch register
r13	sp	lower end of current stack frame
r14	lr	link address / scratch register
r15	pc	program counter

Example

r0 – r3 and Stack

```
int add(int a1, int a2, int a3, int a5, int a6, int a7)
{
    int v1, v2, v3, v4, v5, v6;
    int sum;

    v1 = a1;
    v2 = a2;
    ...

    sum = v1 + v2 + ... + v6;
    return sum;
}
```

r4 – r10 and Stack

r0

Summary

❑ Passing/returning arguments between function calls

- Argument : r0 ~ r3
- Variable : r4 ~ r10
- Return : r0

❑ Other Registers

- Stack pointer : r13
- Return address : r14
- PC(Program Counter) : r15

❑ Stack

- Full Descending Stack

STRUCTURE OF AN ARM ASSEMBLY MODULE

Structure of an ARM Assembly Module

□ A simple example

```
        AREA Example, CODE, READONLY        ; name this block of code
        ENTRY                                ; mark first instruction
                                              ; to execute

start
        MOV     r0, #15                      ; Set up parameters
        MOV     r1, #20
        BL      firstfunc                    ; Call subroutine
        SWI     0x11                         ; terminate
firstfunc                                ; Subroutine firstfunc
        ADD     r0, r0, r1                   ; r0 = r0 + r1
        MOV     pc, lr                       ; Return from subroutine
                                              ; with result in r0
        END                                  ; mark end of file
```

Example : C Programming

```
int add()
{
    int a, b;
    int sum;

    a = 15;
    b = 20;
    sum = a+b;

    return sum;
}
```

Structure of an ARM Assembly Module

□ The AREA directive

- Areas are chunks of data or code that are manipulated by the linker
- A complete application will consist of one or more areas
- The example above consists of a single area which contains code and is marked as being read-only
- **A single CODE area is the minimum required to produce an application**

```
        AREA Example, CODE, READONLY      ; name this block of code
        ENTRY                             ; mark first instruction
                                           ; to execute

start
        MOV     r0, #15                    ; Set up parameters
        MOV     r1, #20
        BL      firstfunc                  ; Call subroutine
        SWI     0x11                       ; terminate
firstfunc                                ; Subroutine firstfunc
        ADD     r0, r0, r1                 ; r0 = r0 + r1
        MOV     pc, lr                     ; Return from subroutine
                                           ; with result in r0
        END                                ; mark end of file
```

Structure of an ARM Assembly Module

□ The ENTRY directive

- **The first instruction to be executed** within an application is marked by the ENTRY directive
- An application can contain only a single entry point and so in a multi-source-module application, only a single module will contain an ENTRY directive
- Note that when an application contains C code, the entry point will usually be contained within the C library

```
AREA Example, CODE, READONLY      ; name this block of code
ENTRY                             ; mark first instruction
                                   ; to execute

start
    MOV    r0, #15                  ; Set up parameters
    MOV    r1, #20
    BL     firstfunc                ; Call subroutine
    SWI     0x11                    ; terminate
firstfunc                          ; Subroutine firstfunc
    ADD    r0, r0, r1                ; r0 = r0 + r1
    MOV    pc, lr                   ; Return from subroutine
                                   ; with result in r0
    END                             ; mark end of file
```

Structure of an ARM Assembly Module

□ General layout

- The general form of lines in an assembler module is:
 - label <whitespace> instruction <whitespace> ;comment
- The important thing to note is that the three sections are **separated by at least one whitespace character (such as a space or a tab)**
- All three sections are optional and the assembler will also accept blank lines to improve the clarity of the code

Structure of an ARM Assembly Module

□ A simple example

```
        AREA Example, CODE, READONLY        ; name this block of code
        ENTRY                               ; mark first instruction
                                           ; to execute

start
        MOV     r0, #15                     ; Set up parameters
        MOV     r1, #20
        BL      firstfunc                   ; Call subroutine
        SWI     0x11                        ; terminate
firstfunc                               ; Subroutine firstfunc
        ADD     r0, r0, r1                  ; r0 = r0 + r1
        MOV     pc, lr                      ; Return from subroutine
                                           ; with result in r0
        END                                 ; mark end of file
```

Structure of an ARM Assembly Module

□ Description of the module

- The main routine of the program (labelled start) loads the values 15 and 20 into registers 0 and 1
- The program then calls the subroutine firstfunc by using a branch with link instruction (BL)
- The subroutine adds together the two parameters it has received and places the result back into r0, as required by the APCS
- It then stores in the link register, lr (r14), the address of the next instruction to be executed in the main routine
- Upon return from the subroutine, the main program simply terminates using software interrupt 11
 - This instructs the program to exit cleanly and return control to the debugger

Structure of an ARM Assembly Module

□ The END directive

- This directive **instructs the assembler to stop processing this source file**
- Every assembly language source module must finish with an END directive on a line by itself

```
        AREA Example, CODE, READONLY      ; name this block of code
        ENTRY                             ; mark first instruction
                                           ; to execute

start
        MOV     r0, #15                    ; Set up parameters
        MOV     r1, #20
        BL      firstfunc                  ; Call subroutine
        SWI     0x11                       ; terminate
firstfunc                                ; Subroutine firstfunc
        ADD     r0, r0, r1                  ; r0 = r0 + r1
        MOV     pc, lr                     ; Return from subroutine
                                           ; with result in r0
        END                                ; mark end of file
```

GENERAL GUIDE TO ARM ASSEMBLY

Formats

- ❑ The general form of source lines in an ARM assembly language module is:

```
{symbol} {instruction|directive|pseudo-instruction} {;comment}
```

- ❑ Case rules

- Instruction mnemonics, directives, and symbolic register names can be written in uppercase or lowercase, but not mixed

- ❑ Line length

- To make source files easier to read, a long line of source can be split onto several lines by placing a backslash character '\ ' at the end of the line
- The backslash must not be followed by any other characters (including spaces and tabs)
- The backslash/end-of-line sequence is treated by the assembler as white space

Symbols: Variables, Constants, Labels, and Local Labels

- ❑ You can use symbols to represent variables, addresses, and numeric constants
 - Symbols representing addresses are also called labels
 - You can use uppercase letters, lowercase letters, numeric characters, or the underscore character in symbol names.
 - **Do not use numeric characters for the first character of symbol names, except in local labels**
 - **Symbol names are case-sensitive**
 - If you need to use a wider range of characters in symbols, for example, when working with compilers, use single bars to delimit the symbol name
 - For example: `|.text|`

Symbols: Variables, Constants, Labels, and Local Labels

□ Variables are of three types

- Numeric, logical, and string
 - The possible values of a logical variable are {TRUE} or {FALSE}

Symbols: Variables, Constants, Labels, and Local Labels

- ❑ Use the **GBLA**, **GBLL**, **GBLS**, **LCLA**, **LCLL**, and **LCLS** Directives to declare symbols representing variables, and assign values to them using the **SETA**, **SETL**, and **SETS** directives

❑ Example

	GBLA	VersionNumber
VersionNumber	SETA	21
	GBLL	Debug
Debug	SETL	{TRUE}
	GBLS	VersionString
VersionString	SETS	"Version 1.0"

Symbols: Variables, Constants, Labels, and Local Labels

□ Constants

- Constants can be numeric, boolean, character or string:
- **Numbers: Numeric constants are accepted in three forms:**
 - Decimal, for example, 123
 - Hexadecimal, for example, 0x7B
 - n_xxx where:
 - ✓ n is a base between 2 and 9
 - ✓ xxx is a number in that base
- **Boolean: The Boolean constants TRUE and FALSE must be written as {TRUE} and {FALSE}**

Symbols: Variables, Constants, Labels, and Local Labels

□ Constants

▪ Characters

- consist of opening and closing single quotes, enclosing either a single character or an escaped character, using the standard C escape characters

▪ Strings

- consist of opening and closing double quotes, enclosing characters and spaces
 - ✓ If double quotes or dollar signs are used within a string as literal text characters, they must be represented by a pair of the appropriate character
 - ✓ For example, you must use \$\$ if you require a single \$ in the string

Symbols: Variables, Constants, Labels, and Local Labels

□ Labels

- **Labels are symbols that represent addresses**
- The address given by a label is calculated during assembly
- **Program-relative labels**
 - Represent the program counter, plus or minus a numeric constant
- **Register-relative labels**
 - Represent a named register plus a numeric constant

Symbols: Variables, Constants, Labels, and Local Labels

□ Local labels

- **Local labels are a subclass of label**
- A local label begins with a number in the range 0-99
- **Syntax**
 - n{routename}
 - ex) 3routineA, 4routineA
- Unlike other labels, a local label can be defined many times
- The scope of local labels is limited by the AREA directive
- You can use the ROUT directive to limit the scope more tightly

Predefined Register and Processor Names

r0-r15 and R0-R15

a1-a4 (argument, result, or scratch registers, synonyms for r0 to r3)

v1-v8 (variable registers, r4 to r11)

sb and SB (static base, r9)

sl and SL (stack limit, r10)

fp and FP (frame pointer, r11)

ip and IP (intra-procedure-call scratch register, r12)

sp and SP (stack pointer, r13)

lr and LR (link register, r14)

pc and PC (program counter, r15).

cpsr and CPSR (current program status register)

spsr and SPSR (saved program status register).

f0-f7 and F0-F7 (FPA registers)

s0-s31 and S0-S31 (VFP single-precision registers)

d0-d15 and D0-D15 (VFP double-precision registers).

p0-p15 (coprocessors 0-15)

c0-c15 (coprocessor registers 0-15).

Built-In Variables

{PC} or .	Address of current instruction.
{VAR} or @	Current value of the storage area location counter.
{TRUE}	Logical constant true.
{FALSE}	Logical constant false.
{OPT}	Value of the currently-set listing option. The OPT directive can be used to save the current listing option, force a change in it, or restore its original value.
{CONFIG}	Has the value 32 if the assembler is assembling ARM code, or 16 if it is assembling Thumb code.
{ENDIAN}	Has the value big if the assembler is in big-endian mode, or little if it is in little-endian mode.
{CODESIZE}	Is a synonym for {CONFIG}.
{CPU}	Holds the name of the selected cpu. The default is ARM7TDMI. If an architecture was specified in the command line -cpu option, {CPU} holds the value "Generic ARM".

Built-In Variables

{FPU}	Holds the name of the selected fpu. The default is SoftVFP.
{ARCHITECTURE}	Holds the name of the selected ARM architecture.
{PCSTOREOFFSET}	Is the offset between the address of the STR pc,[...] or STM Rb,{..., pc} instruction and the value of pc stored out. This varies depending on the CPU or architecture specified.
{ARMASM_VERSION}	Holds an integer that increases with each version. See also <i>Determining the armasm version at assembly time</i> on page 3-11
ads\$version	Has the same value as {ARMASM_VERSION}.
{INTER}	Has the value True if /inter is set. The default is False.
{ROPI}	Has the value True if /ropi is set. The default is False.
{RWPI}	Has the value True if /rwpi is set. The default is False.
{SWST}	Has the value True if /swst is set. The default is False.
{NOSWST}	Has the value True if /noswst is set. The default is False.

Built-in variables cannot be set using the SETA, SETL, or SETS directives. They can be used in expressions or conditions, for example:

```
IF {ARCHITECTURE} = "4T"
```

Expressions and Literals

❑ String expressions

- consist of combinations of string literals, string variables, string manipulation operators, and parentheses

❑ String literals consist of a series of characters contained between double quote characters

- The length of a string literal is restricted by the length of the input line
- ex)

```
SETS    "this string contains only one "" double quote"
```

```
SETS    "this string contains only one $$ dollar symbol"
```

Expressions and Literals

❑ Numeric expressions

- consist of combinations of numeric constants, numeric variables, ordinary numeric literals, binary operators, and parentheses

❑ Numeric literals can take any of the following forms

- decimal-digits, 0xhexadecimal-digits, &hexadecimal-digits, base-n-digits, and character

❑ Floating-point literals can take any of the following forms

- {-}digits E{-}digits, {-}{digits}.digits{E{-}digits}, 0xhexdigits, and &hexdigits

Unary Operators

Operator	Usage	Description
?	?A	Number of bytes of executable code generated by line defining symbol A.
BASE	:BASE:A	If A is a pc-relative or register-relative expression, BASE returns the number of its register component BASE is most useful in macros.
INDEX	:INDEX:A	If A is a register-relative expression, INDEX returns the offset from that base register. INDEX is most useful in macros.
+ and -	+A -A	Unary plus. Unary minus. + and - can act on numeric and program-relative expressions.
LEN	:LEN:A	Length of string A.
CHR	:CHR:A	One-character string, ASCII code A.
STR	:STR:A	Hexadecimal string of A. STR returns an eight-digit hexadecimal string corresponding to a numeric expression, or the string "T" or "F" if used on a logical expression.
NOT	:NOT:A	Bitwise complement of A.
LNOT	:LNOT:A	Logical complement of A.
DEF	:DEF:A	{ TRUE } if A is defined, otherwise { FALSE }.
SB_OFFSET_19_12	:SB_OFFSET_19_12: label	Bits[19:12] of (label - sb). See <i>Example of use of :SB_OFFSET_19_12: and :SB_OFFSET_11_0</i> on page 3-27
SB_OFFSET_11_0	:SB_OFFSET_11_0: label	Least-significant 12 bytes of (label - sb).

Binary Operators

Table 3-5 Multiplicative operators

Operator	Usage	Explanation
*	A*B	Multiply
/	A/B	Divide
MOD	A:MOD:B	A modulo B

Table 3-6 String manipulation operators

Operator	Usage	Explanation
LEFT	A:LEFT:B	The left-most B characters of A
RIGHT	A:RIGHT:B	The right-most B characters of A
CC	A:CC:B	B concatenated onto the end of A

Binary Operators

Table 3-7 Shift operators

Operator	Usage	Explanation
ROL	A:ROL:B	Rotate A left by B bits
ROR	A:ROR:B	Rotate A right by B bits
SHL	A:SHL:B	Shift A left by B bits
SHR	A:SHR:B	Shift A right by B bits

Table 3-8 Addition, subtraction, and logical operators

Operator	Usage	Explanation
+	A+B	Add A to B
-	A-B	Subtract B from A
AND	A:AND:B	Bitwise AND of A and B
OR	A:OR:B	Bitwise OR of A and B
EOR	A:EOR:B	Bitwise Exclusive OR of A and B

Binary Operators

Table 3-9 Relational operators

Operator	Usage	Explanation
=	A=B	A equal to B
>	A>B	A greater than B
>=	A>=B	A greater than or equal to B
<	A<B	A less than B
<=	A<=B	A less than or equal to B
/=	A/=B	A not equal to B
◇	A◇B	A not equal to B

Table 3-10 Boolean operators

Operator	Usage	Explanation
LAND	A:LAND:B	Logical AND of A and B
LOR	A:LOR:B	Logical OR of A and B
LEOR	A:LEOR:B	Logical Exclusive OR of A and B

Operator Precedence

- ❑ The precedence is not exactly the same as in C
 - recommended to use brackets to make the precedence explicit

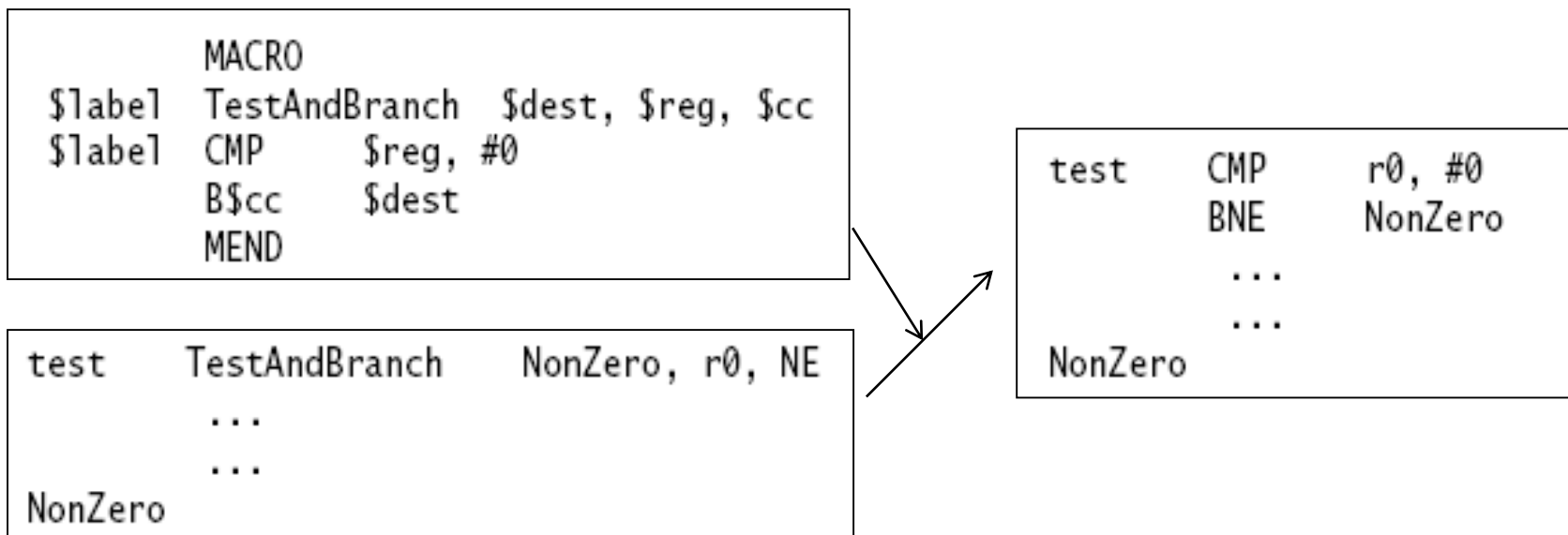
armasm precedence	equivalent C operators
unary operators	unary operators
* / :MOD:	* / %
string manipulation	n/a
:SHL: :SHR: :ROR: :ROL:	<< >>
+ - :AND: :OR: :EOR:	+ - &
= > >= < <= /= <>	== > >= < <= !=
:LAND: :LOR: :LEOR:	&&

Macros

- ❑ A macro definition is a block of code enclosed between **MACRO** and **MEND** directives
 - It defines a name that can be used instead of repeating the whole block of code

- ❑ This has two main uses
 - to make it easier to follow the logic of the source code, by replacing a block of code with a single, meaningful name
 - to avoid repeating a block of code several times.

Macros



Using the C Preprocessor

□ Using the C preprocessor

- You can include the C preprocessor command `#include` in your assembly language source file
- If you do this, you must preprocess the file using the C preprocessor, before using `armasm` to assemble it
- Example shows the commands you write to preprocess and assemble a file, `sourcefile.s`
- In this example, the preprocessor outputs a file called `preprocessedfile.s`, and `armasm` assembles `preprocessedfile.s`

```
armcpp -E < sourcefile.s > preprocessedfile.s  
armasm preprocessedfile.s
```

ARM INSTRUCTION SET

데이터 이동 명령어

□ MOV

- Syntax : `mov{<cond>}{S} Rd, N`
- `Rd = N`
- `N` : 레지스터 값이나 상수 값
- `N` 값을 레지스터로 복사
- 초기 값을 설정하거나 레지스터 간에 데이터를 이동하는데 사용
- 예제 : `mov r0, r1`

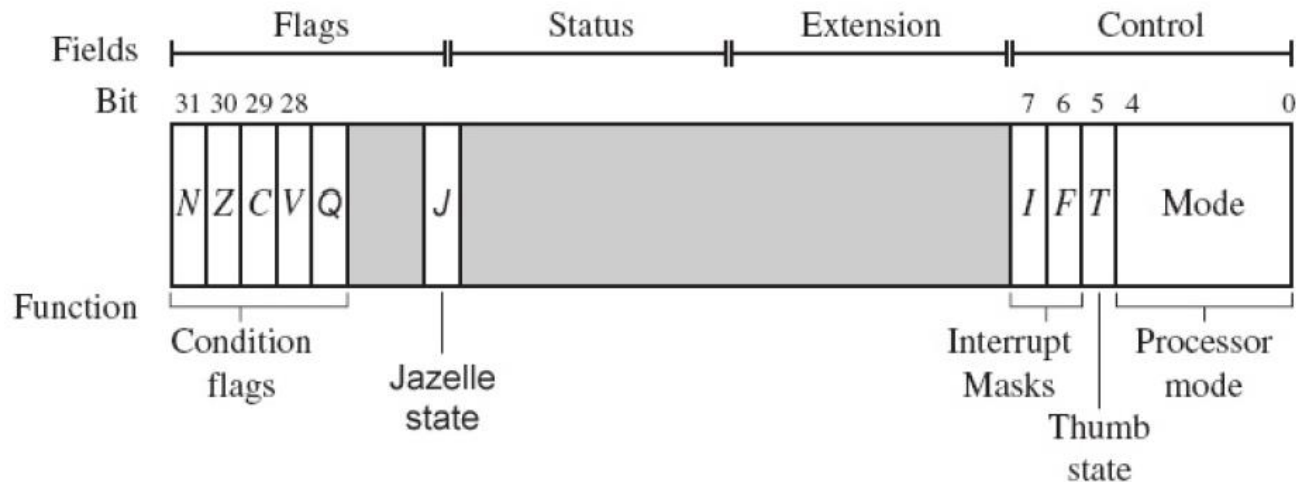
□ MOV의 응용

- 서브 루틴으로 분기 후, 원래 루틴으로 복귀할 때 `mov` 사용
→ `mov pc, lr`

<instruction>{S}

□ 접미사 S

- 데이터 처리 명령어일 경우 명령어 뒤에 추가
 - 데이터 이동, 배럴 시프터, 산술, 논리, 곱셈
 - MOVS, SUBS, ...
- 데이터를 처리한 후, 그 결과로 **CPSR**의 플래그들을 업데이트
 - C(캐리 플래그), N(음수 플래그), Z(제로 플래그) 업데이트



산술 명령어

□ ADD

- Syntax : `add{<cond>}{S} Rd, Rn, N`
- $Rd = Rn + N$
- 32비트 값의 덧셈
- 예제 : `add r0, r1, r2`

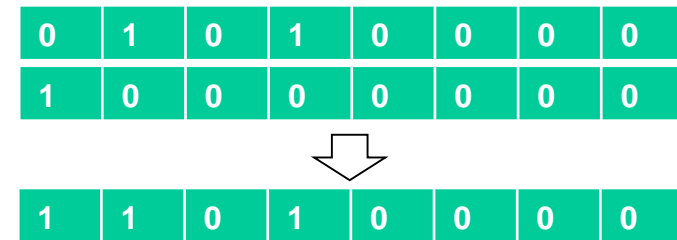
□ SUB

- Syntax : `sub{<cond>}{S} Rd, Rn, N`
- $Rd = Rn - N$
- 32비트 값의 뺄셈
- 예제 : `sub r0, r1, #4`

논리 명령어

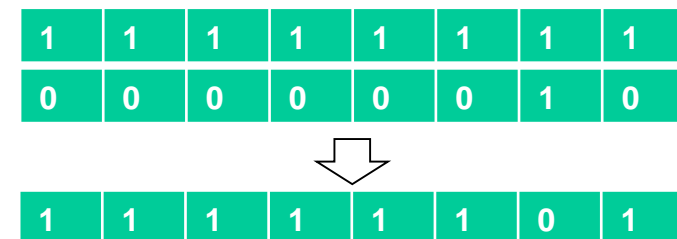
□ ORR

- Syntax : `orr{<cond>}{S} Rd, Rn, N`
- $Rd = Rn \mid N$
- OR 논리 연산
- 특정 비트를 1로 **set**할 때 사용
- 예제 : `orr r0, r0, #0x80`



□ BIC

- Syntax : `bic{<cond>}{S} Rd, Rn, N`
- $Rd = Rn \& \sim N$
- 비트 클리어(AND NOT) 논리 연산
- 특정 비트를 0으로 **clear**할 때 사용
- 예제 : `bic r0, r0, #0x02`



비교 명령어

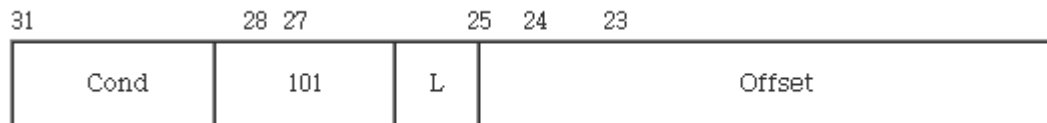
□ CMP

- **Syntax : `cmp{<cond>} Rn, N`**
- **Rn - N의 결과에 따라 `cpsr` 플래그 비트를 업데이트**
 - 접미사 S가 필요 없음
- **조건부 실행을 사용하여 프로그램의 흐름을 변경할 수 있음**
- **예제 : `cmp r0, r9`**
 - r0와 r9의 값이 같을 경우 zero flag가 1로 set
 - 조건이 EQ일 경우 해당 명령어 실행
 - ✓ BEQ, MOVEQ

조건부 실행

□ 조건부 실행?

- 특정 조건을 만족해야만 **ARM** 명령어가 실행되도록 할 수 있음
- 명령어의 최상위 비트 **[31:28]**는 조건 필드
 - Negative, Zero, Carry, oVerflow
- 조건부 실행 명령어를 사용하면, 명령이 실행될 때 조건 필드와 **CPSR**의 조건 플래그 값을 검사하여 명령의 실행 여부 결정



Branch 명령어 구조

조건부 실행

명령어의 접미사	CPSR의 플래그	의미
EQ	Z set	같다
NE	Z clear	같지 않다
CS	C set	크거나 같다 (unsigned)
CC	C clear	작다 (unsigned)
MI	N set	음수 (negative)
PL	N clear	양수 (positive) 또는 0
VS	V set	오버플로우
VC	V clear	오버플로우가 발생하지 않음
HI	C set && Z clear	크다 (unsigned)
LS	C clear	작다 (unsigned)
GE	N == V	크거나 같다
LT	N != V	작다
GT	Z clear & (N == V)	크다
LE	Z set (N != V)	작거나 같다
AL	Always	무조건 실행

분기 명령어

□ label

- 명령어의 주소를 표현하기 위해 사용
- 코딩할 때는 메모리에서 명령어의 위치를 알 수 없으므로 **label**을 사용해 해당 명령어의 주소를 표현
- 예제

start

```
mov    r0, #15
mov    r1, #20
bl     firstfunc
swi    0x11
```


분기 명령어

□ B

- Branch
- Syntax : b{<cond>} label
- pc = label
- pc에 label이 가리키는 주소 값을 저장하여 해당 명령어를 실행


□ BL

- Branch with link
- Syntax : bl{<cond>} label
- pc = label, lr = BL 다음 명령어의 주소
- Link register에 복귀할 주소를 저장하기 때문에 서브루틴 호출을 수행하는 데 사용됨


분기 명령어

□ 예제

```
b    forward
add  r1, r2, #4
add  r0, r6, #2
add  r3, r7, #4
forward
sub  r1, r2, #4
```



```
backward
add  r1, r2, #4
sub  r1, r2, #4
add  r4, r6, r7
b    backward
```



분기 명령어

□ BEQ, BNE

- 분기 명령어의 조건부 실행
- EQ는 Zero flag가 1로 set되었을 때 분기
- NE는 Zero flag가 0으로 clear되었을 때 분기
- 예제

```
mov    r0, #0  
cmp    r0, #0  
beq    firstfunc
```

로드-스토어 명령어

□ LDR

- Syntax : `ldr{<cond>} {B|SB|H|SH} Rd, addressing`
- `Rd ← mem32[address]`
- 메모리에서 레지스터로 한 워드를 읽어들이м
- 예제 : `ldr r0, [r1]`

□ STR

- Syntax : `str{<cond>} {B|H} Rd, addressing`
- `Rd → mem32[address]`
- 레지스터에서 메모리로 한 워드를 저장
- 예제 : `str r0, [r1]`
`str sp, stack_ptr`

ldr Rd, =constant

□ 상수값은 어떻게 저장할 것인가?

- ARM 명령어는 32비트이기 때문에 명령어에 32비트의 상수 값을 포함할 수 없음
- 상수 값 저장을 위해 **Pseudo instruction**을 제공
 - ldr Rd, =constant
 - ✓ 8bit(0-255)의 수는 'mov Rd, #constant'로 변환하여 저장
 - ✓ 256이상의 수는 'ldr Rd, [pc, #offset]'로 변환하여 저장

□ 예제

ldr r0, =0x55555555

스택 명령어

❑ Push & Pop

- ARM은 **push, pop** 명령어가 없음
 - 다중-레지스터 전송 명령 사용
- **Pop** → **LDM** 명령어 사용
- **Push** → **STM** 명령어 사용

❑ Ascending & Descending

- 스택을 메모리 위쪽으로 증가시킬 지 아래쪽으로 증가시킬 지를 결정해야 함
- **Ascending stack(A)** → 메모리의 상위 주소 방향으로 스택이 자라는 것
- **Descending stack(D)** → 메모리의 하위 주소 방향으로 스택이 자라는 것

❑ Full & Empty

- 스택 포인터 **sp**가 가리키는 위치를 설정
- **Full stack(F)** → 스택 포인터 **sp**는 마지막으로 사용된 위치의 주소를 가리킴
- **Empty stack(E)** → **sp**가 스택의 마지막 아이템 다음의 빈 공간을 가리킴

❑ 스택 명령어는 위 3가지 설정을 하나씩 선택하여 결정

- **STMFD, LDMEA, ...**

스택 명령어

□ ARM은 APCS에 의해 full descending stack을 사용

- STMFD와 LDMFD로 push와 pop 기능을 제공

□ 예제

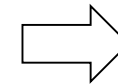
r1 = 0x02

r4 = 0x03

sp = 0x80014

>>STMFD sp!, {r1, r4}

	Address	Data
	0x80018	0x01
sp →	0x80014	0x02
	0x80010	Empty
	0x8000c	Empty

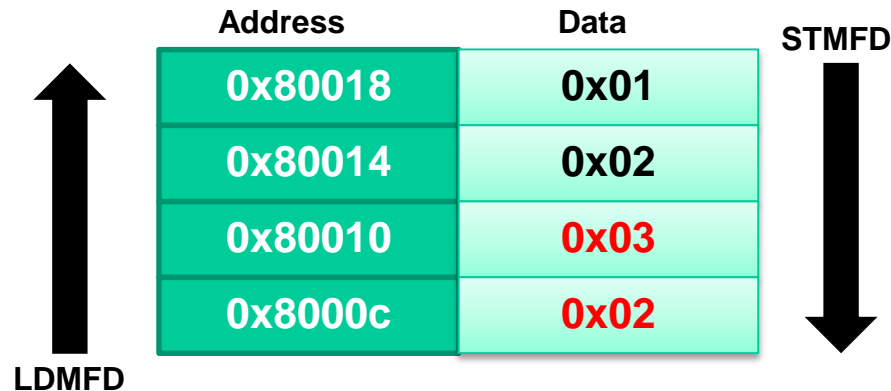


	Address	Data
	0x80018	0x01
	0x80014	0x02
	0x80010	0x03
sp →	0x8000c	0x02

스택 명령어

□ STMFD & LDMFD

- STMFD sp!, {r1, r2, r3, r4}
 - r4 → r1 순으로 스택에 push
- LDMFD sp!, {r1, r2, r3, r4}
 - r1 → r4 순으로 스택에서 pop



PSR 명령어

□ PSR 명령어

- CPSR과 SPSR을 읽고 쓰기 위해 사용
- MRS와 MSR 명령어가 존재

□ MRS

- Syntax : `mrs{<cond>} Rd, <cpsr | spsr>`
- `Rd = psr`
- PSR 레지스터를 범용 레지스터에 복사
- 예제 : `mrs r0, cpsr`

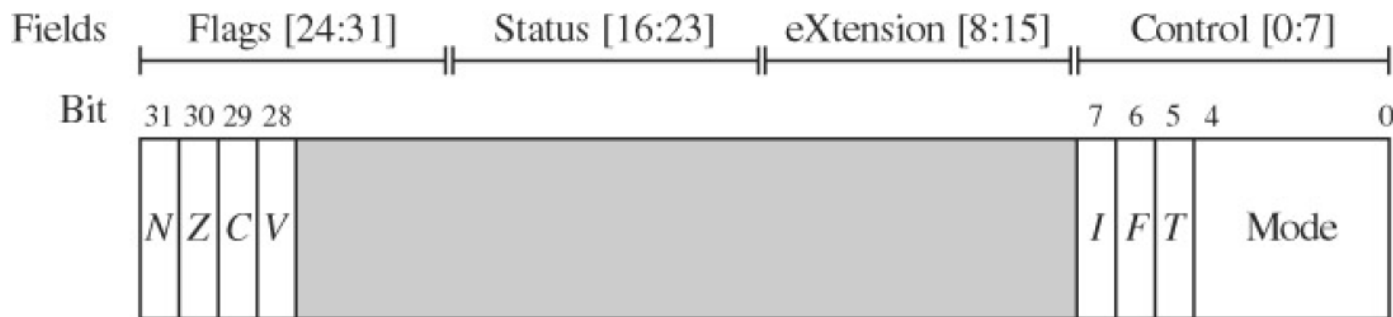
□ MSR

- Syntax : `msr{<cond>} <cpsr | spsr>_<fields>, Rm`
- `psr[field] = Rm`
- 범용 레지스터를 PSR 레지스터로 이동
- 예제 : `msr cpsr_c, r1`

PSR 명령어

□ MSR명령어의 fields

- PSR레지스터의 특정 바이트 영역의 값만 바꿀 때 사용
- **cpsr_c**
 - 프로세서 mode와 Thumb 상태, 인터럽트 마스크 값만 수정하고자 할 때 사용
- **cpsr_cxsf**
 - 'cpsr'과 동일. CPSR의 모든 바이트 영역을 수정



코프로세서 명령어

□ 코프로세서 명령어

- 명령어 세트를 확장하기 위해 사용
 - 계산 능력 확장
 - 캐시와 메모리 관리 장치를 포함한 메모리 서브 시스템 제어

□ MCR

- **Syntax : MCR{<cond>}, cp, opcode1, Rd, CRn, CRm {, opcode2}**
- 범용 레지스터의 데이터를 코프로세서로 이동
- **cp** : p0 ~ p15에 해당하는 코프로세서 번호
- **opcode** : 코프로세서 상에서 발생하게 될 동작
- **CRn, CRm** : 코프로세서의 레지스터

□ MRC

- **Syntax : MRC{<cond>}, cp, opcode1, Rd, CRn, CRm {, opcode2}**
- 코프로세서 레지스터의 데이터를 범용 레지스터로 이동

코프로세서 명령어

□ MCR 예제

▪ mcr p15, 0, r0, c7, c5, 0

- p15 : 메모리 관리 장치, 쓰기 버퍼 제어, 캐시 제어 등
- 모든 instruction cache를 무효화

CRn	Op1	CRm	Op2					
<u>c7</u>	<u>0</u>	c0	0-3	Undefined	-	-	-	-
			4	NOP (WFI)	WO	WO	-	page 3-2
			5-7	Undefined	-	-	-	-
		c1-c3	0-7	Undefined	-	-	-	-
		c4	0	Physical Address	R/W	R/W, B	0x00000000	page 3-71
			1-7	Undefined	-	-	-	-
<u>c5</u>	<u>0</u>			Invalidate all instruction caches to point of unification	WO	WO	-	page 3-68
			1	Invalidate instruction cache line to point of unification	WO	WO	-	page 3-68
			2-3	Undefined	-	-	-	-

코프로세서 명령어

□ MRC 예제

- **mrc p15, 0, r0, c1, c0, 0**

- System control register의 데이터를 범용 레지스터 r0로 이동

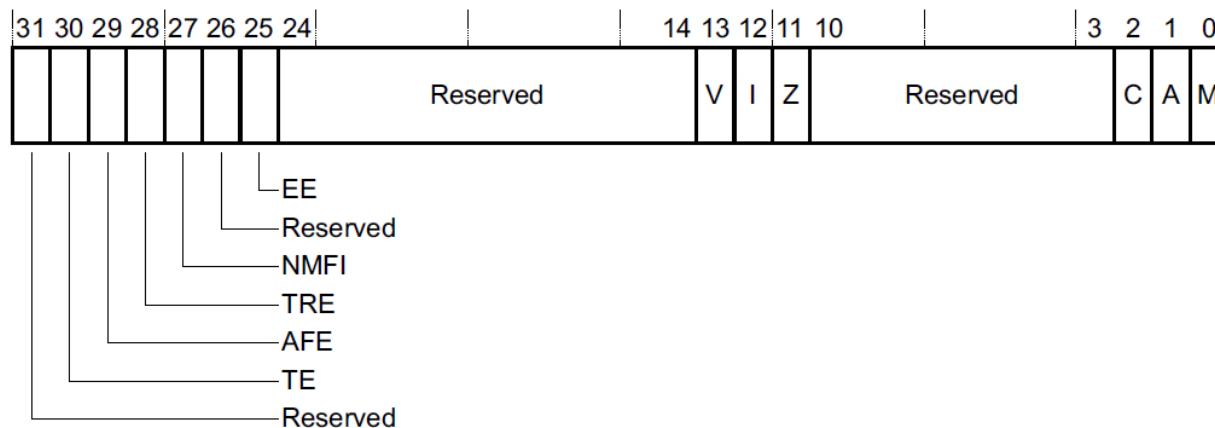


Figure 3-20 Control Register bit assignments

Quiz

1. 'beq firstfunc'은 실행되는가? 그 이유도 함께 설명하시오 (4점)

```
mov    r0, #5
cmp    r0, #5
addne  r0, r0, #1
cmp    r0, #6
beq    firstfunc
```

2. 다음 그림의 빈칸의 값을 순서대로 쓰시오 (3점)

r2 = 0x02

r3 = 0x03

r4 = 0x04

sp = 0x80018

>> stmfd sp!, {r2, r3, r4}

	Address	Data
sp →	0x80018	0x01
	0x80014	(1)
	0x80010	(2)
	0x8000c	(3)

3. 2번 문제의 **STMFD** 명령을 실행한 뒤, “ldmfd sp!, {r3}”를 실행했을 때, r3의 값은 얼마인가? 그리고 sp가 가리키는 위치는 어디인가? (3점)

제출 방법

□ 제출 방법

- 각 문제의 정답을 워드나 한글에서 작성하여 메일에 첨부
- 해당 문서에도 학번과 이름을 적을 것

□ E-mail (반드시 아래 2개의 메일 계정으로 모두 전송)

- mkchoi@rtcc.hanyang.ac.kr
- ksjun@rtcc.hanyang.ac.kr

□ 메일 제목

- [임베디드 시스템 실습 과제1]학번_이름

□ 마감일

- 다음 실습 수업시간 전까지

수고하셨습니다.