3. ARM 어셈블리 언어 프로그래밍

– 전체 요약

The ARM processor is very easy to program at the assembly level, though for most applications it is more appropriate to program in a high-level language such as C or C++.

Assembly language programming requires the programmer to think at the level of the individual machine instruction. An ARM instruction is 32 bits long, so there are around 4 billion different binary machine instructions. Fortunately there is considerable structure within the instruction space, so the programmer does not have to be familiar with each of the 4 billion binary encodings on an individual basis. Even so, there is a considerable amount of detail to be got right in each instruction. The assembler is a computer program which handles most of this detail for the programmer.

In this chapter we will look at ARM assembly language programming at the user level and see how to write simple programs which will run on an ARM development board or an ARM emulator (for example, the ARMulator which comes as part of the ARM development toolkit). Once the basic instruction set is familiar we will move on, in Chapter 5, to look at system-level programming and at some of the finer details of the ARM instruction set, including the binary-level instruction encoding.

Some ARM processors support a form of the instruction set that has been compressed into 16-bit 'Thumb' instructions. These are discussed in Chapter 7.

3.1 데이터 처리 명령어

- 데이터 처리 명령어는 데이터 값을 수정할 수 있는 유일한 명령어이고 두 개의 소스 오퍼랜드와 결과를 저장하는 목적지 오퍼랜드가 필요

- ARM 데이터 처리 명령어 규칙

  ▷ 모든 오퍼랜드의 크기는 32 비트이고 오퍼랜드는 레지스터나 상수를 사용

  ▷ 오퍼랜드는 독립적으로 설정 가능 (3 주소 명령어)

  ▷ 명령어 형식 : ADD r0, r1, r2 ; r0 = r1 + r2

- 데이터 처리 명령어 종류

  ▷ 산술 연산 명령어 : 덧셈, 뺄셈, 역뺄셈 연산을 실행

```
ADD     r0, r1, r2          ; r0 := r1 + r2
ADC     r0, r1, r2          ; r0 := r1 + r2 + C
SUB     r0, r1, r2          ; r0 := r1 - r2
SBC     r0, r1, r2          ; r0 := r1 - r2 + C - 1
RSB     r0, r1, r2          ; r0 := r2 - r1
RSC     r0, r1, r2          ; r0 := r2 - r1 + C - 1
```

▷ 논리 연산 명령어 : AND, OR, XOR 등의 부울 논리 연산 실행

```
AND     r0, r1, r2          ; r0 := r1 and r2
ORR     r0, r1, r2          ; r0 := r1 or r2
EOR     r0, r1, r2          ; r0 := r1 xor r2
BIC     r0, r1, r2          ; r0 := r1 and not r2
```

▷ 레지스터 이동 명령어 : 두 번째 오퍼랜드를 목적지 오퍼랜드로 이동

```
MOV     r0, r2              ; r0 := r2
MVN     r0, r2              ; r0 := not r2
```

▷ 비교 명령어 : 두 레지스터 값을 비교하는 명령어로 비교 결과에 따라 CPSR 레지스터의 해당 비트 설정

```
CMP     r1, r2              ; set cc on r1 - r2
CMN     r1, r2              ; set cc on r1 + r2
TST     r1, r2              ; set cc on r1 and r2
TEQ     r1, r2              ; set cc on r1 xor r2
```

- 상수 오퍼랜드

▷ 레지스터에 상수를 더하거나 뺄 때 편리하고 두 번째 오퍼랜드를 상수 오퍼랜드로 사용

▷ 상수를 표시하기 위해 #을 사용하고 16진수 표기를 위해 &를 추가

```
ADD     r3, r3, #1          ; r3 := r3 + 1
AND     r8, r7, #&ff        ; r8 := r7[7:0]
```

▷ 상수를 포함한 명령어 길이가 32 비트이므로 32 비트 상수를 저장 불가능

유효한 상수 범위 : $(0 \rightarrow 255) \times 2^{2n}$, $(0 \le n \le 12)$

- 자리이동 레지스터 오퍼랜드

▷ 두 번째 소스 오퍼랜드를 자리이동 한 후에 데이터 처리 명령 실행

ADD r3, r2, r1, LSL #3 ; r3 = r2 + r1<<3 = r2 + 8 × r1

▷ 하나의 명령어 사이클에서 실행, 대부분의 프로세서가 자리이동 명령어를 별도로 제공하지만 ARM은 한 명령어에 ALU와 자리이동 동작을 동시 수행

▷ 제공하는 자리이동 동작

- LSL: logical shift left by 0 to 31 places; fill the vacated bits at the least significant end of the word with zeros.
- LSR: logical shift right by 0 to 32 places; fill the vacated bits at the most significant end of the word with zeros.
- ASL: arithmetic shift left; this is a synonym for LSL.
- ASR: arithmetic shift right by 0 to 32 places; fill the vacated bits at the most significant end of the word with zeros if the source operand was positivc, or with ones if the source operand was negative.
- ROR: rotate right by 0 to 32 places; the bits which fall off the least significant end of the word are used, in order, to fill the vacated bits at the most significant end of the word.
- RRX: rotate right extended by 1 place; the vacated bit (bit 31) is filled with the old value of the C flag and the operand is shifted one place to the right. With appropriate use of the condition codes (see below) a 33-bit rotate of the operand and the C flag is performed.

▷ 자리 이동량을 레지스터에 저장 가능

ADD r5, r5, r3, LSL r2 ; r5 = r5 + r3 × 2^{r2} (4주소 명령어)

- 조건 코드 설정

▷ 데이터 처리 명령어는 프로그래머가 원하면 조건 코드 (N, Z, C, V) 설정

▷ 비교 명령어를 제외한 데이터 처리 명령어는 명령어에 S를 첨가하여 연산 결과에 따라 조건 코드를 설정

▷ 64 비트 덧셈 (r1-r0, r3-r2) 등에서 C 비트는 계산 중간의 캐리를 저장

```
ADDS    r2, r2, r0 ; 32-bit carry out -> C..
ADC     r3, r3, r1 ; .. and added into high word
```

▷ 사용자에 의해 조건 코드 반영 유무를 제어 가능

▷ 비교 명령어 (CMP, CMN)는 결과에 따라 모든 조건 코드가 영향을 받음

▷ 논리와 이동 명령어는 C, V에는 영향을 주지 않고 N, Z에 영향을 줌

▷ 조건 코드에 따라 조건 분기 명령어에 의한 프로그램 제어 흐름을 결정

- 곱셈 명령어

▷ 곱셈 명령어의 형식 : MUL r4, r3, r2 ; $r4 = r3 \times r2_{[31:0]}$

▷ 상수 오퍼랜드는 허용되지 않고 64 비트 결과의 하위 32 비트만 레지스터에 저장, 64 비트 전체를 저장하는 long 곱셈 명령어도 제공

▷ 곱셈-덧셈 명령어

MLA r4, r3, r2, r1 ; $r4 = (r3 \times r2 + r1)_{[31:0]}$

▷ 상수 곱셈은 상수를 레지스터에 저장한 후 곱셈 명령어를 사용하여 구현할 수 있지만 자리 이동, 덧셈, 뺄셈 명령어를 사용하여 효율적으로 구현 가능

3.2 데이터 이동 명령어

- 데이터 이동 명령어의 기본 형태

▷ 싱글 레지스터 적재와 저장 명령어 : 하나의 데이터를 이동할 때 사용

▷ 다중 레지스터 적재와 저장 명령어 : 많은 양의 데이터를 효율적으로 이동하기 위해 사용, 데이터 블록을 복사, 활용 공간 레지스터를 저장하고 복원

▷ 싱글 레지스터 교환 명령어 : 레지스터와 메모리의 데이터를 상호 교환, 사용자 모드에서는 거의 사용되지 않음

- 레지스터 간접 어드레싱

▷ ARM 데이터 이동 명령어는 레지스터 간접 어드레싱, Base-plus-offset, Base-plus-index 어드레싱 등을 토대로 실행

▷ 레지스터 간접 어드레싱을 토대로 다른 어드레싱 방법이 만들어짐

▷ 레지스터 간접 어드레싱에 의한 명령어 형태

```
LDR     r0, [r1]          ; r0 := mem32[r1]
STR     r0, [r1]          ; mem32[r1] := r0
```

- 어드레스 포인터의 초기화

▷ 특정 메모리 위치에 load, store하기 위해 ARM 레지스터가 특정 메모리 위치의 주소를 가지도록 초기화되어야 함

▷ 현재 실행하고 있는 명령어 부근의 위치를 가리키기 위해 r15에 더할 작은 offset을 알아야 하는데 이를 위해 ADR이라는 pseudo 명령어를 제공

▷ ADR 대신 메모리 위치를 계산하면 프로그램이 길어지고 실행시간이 길어짐

▷ ADR 명령어의 사용 예

```
COPY    ADR     r1, TABLE1          ; r1 points to TABLE1
        ADR     r2, TABLE2          ; r2 points to TABLE2
        ..
TABLE1                              ; < source of data >
        ..
TABLE2                              ; < destination >
        ..
```

- 싱글 레지스터 적재, 저장 명령어

▷ 레지스터 간접 어드레싱에 의한 적재, 저장 명령어 사용 예

```
COPY    ADR     r1, TABLE1          ; r1 points to TABLE1
        ADR     r2, TABLE2          ; r2 points to TABLE2
LOOP    LDR     r0, [r1]            ; get TABLE1 1st word
        STR     r0, [r2]            ; copy into TABLE2
        ADD     r1, r1, #4          ; step r1 on 1 word
        ADD     r2, r2, #4          ; step r2 on 1 word
        ???                         ; if more go back to LOOP
```

- Base plus offset 어드레싱

▷ 베이스 레지스터에 4K 바이트 범위내의 offset을 더하여 메모리 주소 계산

▷ pre-indexed 어드레싱 모드

```
LDR     r0, [r1, #4]        ; r0 = mem32[r1 + 4]
```

▷ auto-indexing pre-indexed 어드레싱 모드

```
LDR     r0, [r1, #4]!       ; r0 = mem32[r1 + 4], r1 = r1 + 4
```

▷ 데이터가 메모리에서 fetch되는 동안 데이터패스에서 auto-indexing이 실행되기 때문에 auto-indexing을 위한 추가적인 코드와 시간으로 인한 비용이 들지 않음

▷ post-indexed 어드레싱 모드

```
LDR     r0, [r1], #4        ; r0 = mem32[r1], r1 = r1 + 4
```

```
COPY    ADR     r1, TABLE1          ; r1 points to TABLE1
        ADR     r2, TABLE2          ; r2 points to TABLE2
LOOP    LDR     r0, [r1], #4        ; get TABLE1 1st word
        STR     r0, [r2], #4        ; copy into TABLE2
        ???                         ; if more go back to LOOP
```

▷ unsigned 8 비트 데이터 이동 명령어

```
LDRB    r0, [r1]            ; r0 = mem8[r1]
```

- 다중 레지스터 데이터 이동

▷ 하나의 명령어로 여러 레지스터 데이터를 동시에 저장, 적재

▷ 싱글 레지스터 데이터 이동보다는 사용할 수 있는 어드레싱 모드가 제한

```
LDMIA   r1, {r0,r2,r5}      ; r0 := mem32[r1]
                            ; r2 := mem32[r1 + 4]
                            ; r5 := mem32[r1 + 8]
```

▷ r0에서 r15까지 모든 레지스터를 사용 가능

▷ 베이스 레지스터에 !를 첨가함으로써 auto-indexing을 동시에 실행 가능

- 스택 어드레싱

▷ 스택은 스택에 데이터를 저장할 때 주소가 증가하느냐 감소하느냐에 따라 ascending, descending 스택으로 구분되고 스택 포인터가 가리키고 있는 데이터에 따라 full, empty 스택으로 구분 (4가지 형태의 스택 존재 가능)

▷ 4가지 형태의 스택

- Full ascending: the stack grows up through increasing memory addresses and the base register points to the highest address containing a valid item.
- Empty ascending: the stack grows up through increasing memory addresses and the base register points to the first empty location above the stack.
- Full descending: the stack grows down through decreasing memory addresses and the base register points to the lowest address containing a valid item.
- Empty descending: the stack grows down through decreasing memory addresses and the base register points to the first empty location below the stack.

- 블록 복사 어드레싱

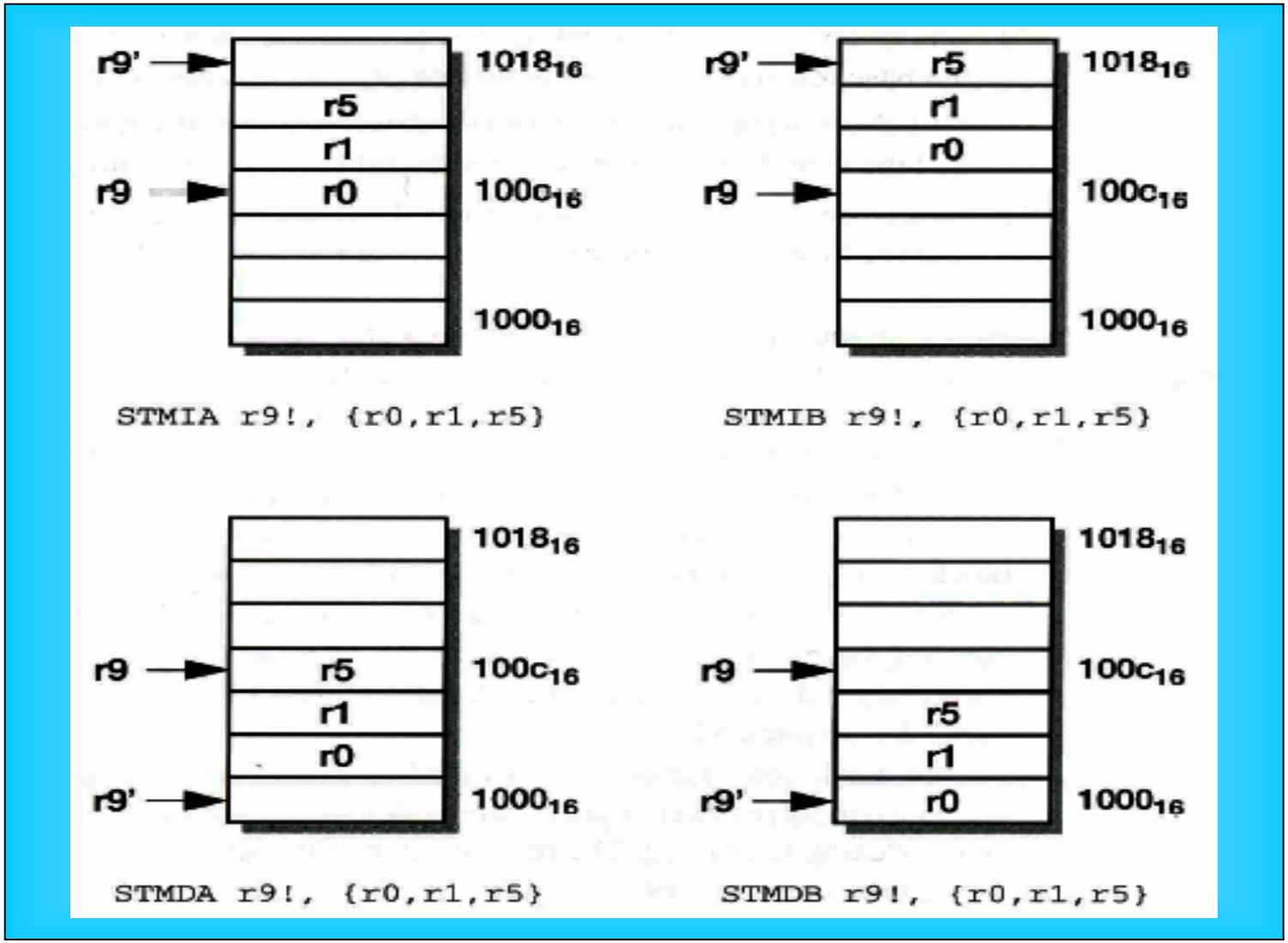
▷ 메모리의 특정 데이터 블록을 다른 위치로 복사할 때 편리한 어드레싱 방법

▷ 블록 복사 방법에 따라 베이스 주소부터 주소가 증가하는 방향이나 감소하는 방향으로 데이터를 저장할 수 있고 베이스 주소도 데이터 저장 전이나 후에 주소를 증가하거나 감소할 수 있음

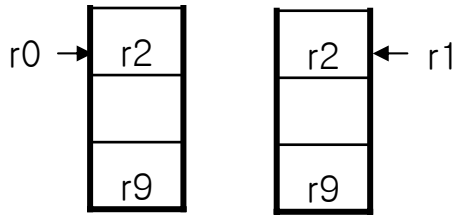
▷ 다중 데이터 저장 적재를 위한 스택과 블록복사 관점 사이의 관계

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

▷ 베이스 주소가 r9이고 r0, r1, r5데이터를 저장하는 4가지 방법

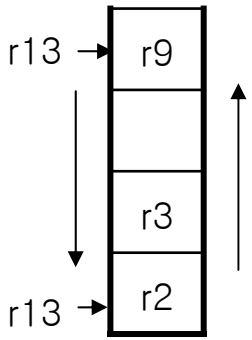


▷ r0 메모리 위치에서 r1 메모리 위치로 8워드를 복사하는 코드




```
LDMIA   r0!, {r2-r9}
STMIA   r1,  {r2-r9}
```

▷ 블록 복사를 위해 r2-r9 데이터를 스택에 저장하고 복원하는 코드




```
STMDB
STMFD   r13!, {r2-r9}      ; save regs onto stack
LDMIA   r0!,  {r2-r9}
STMIA   r1,   {r2-r9}
LDMFD   r13!, {r2-r9}      ; restore from stack
LDMIA
```

▷ 다중 레지스터 적재 저장 명령어는 싱글 레지스터 적재 저장 명령어 보다 코드 길이를 줄이고 최대 4배까지 속도 증가시킴

▷ 다중 레지스터 데이터 이동 명령어는 독립적인 명령어와 데이터 캐쉬를 가지고 있더라도 싱글 사이클에 실행할 수 없기 때문에 순수한 RISC idea는 아님, 현재 대부분의 RISC 프로세서는 다중 데이터 이동 명령어를 제공

3.5 제어 흐름 명령어

– 분기 명령어

▷ 프로그램이 실행되는 흐름을 바꾸는 명령어

```
        B       LABEL
        ..
LABEL   ..
```

▷ 현재의 명령어 전과 후로 이동 가능, 이동할 실제 거리는 컴파일러가 계산

– 조건 분기

▷ 반복 루프를 탈출하기 위해 조건 분기 구현 필요

▷ 조건에 해당하는 조건 코드가 맞을 때에 분기가 일어남

```
        MOV     r0, #0              ; initialize counter
LOOP    ..
        ADD     r0, r0, #1          ; increment loop counter
        CMP     r0, #10             ; compare with limit
        BNE     LOOP                ; repeat if not equal
        ..                          ; else fall through
```

▷ 분기 조건

Branch	Interpretation	Normal uses
B	Unconditional	Always take this branch
BAL	Always	Always take this branch
BEQ	Equal	Comparison equal or zero result
BNE	Not equal	Comparison not equal or non-zero result
BPL	Plus	Result positive or zero
BMI	Minus	Result minus or negative
BCC	Carry clear	Arithmetic operation did not give carry-out
BLO	Lower	Unsigned comparison gave lower
BCS	Carry set	Arithmetic operation gave carry-out
BHS	Higher or same	Unsigned comparison gave higher or same
BVC	Overflow clear	Signed integer operation; no overflow occurred
BVS	Overflow set	Signed integer operation; overflow occurred
BGT	Greater than	Signed integer comparison gave greater than
BGE	Greater or equal	Signed integer comparison gave greater or equal
BLT	Less than	Signed integer comparison gave less than
BLE	Less or equal	Signed integer comparison gave less than or equal
BHI	Higher	Unsigned comparison gave higher
BLS	Lower or same	Unsigned comparison gave lower or same

– 조건 실행

▷ 모든 명령어의 조건 실행 가능, 조건 분기 명령어 사용을 줄일 수 있음

```
        CMP     r0, #5
        BEQ     BYPASS              ; if (r0 != 5) {
        ADD     r1, r1, r0          ;   r1 := r1 + r0 - r2
        SUB     r1, r1, r2          ; }
BYPASS  ..
```

This may be replaced by:

```
        CMP     r0, #5              ; if (r0 != 5) {
        ADDNE   r1, r1, r0          ;   r1 := r1 + r0 - r2
        SUBNE   r1, r1, r2          ; }
        ..
```

▷ 조건 분기 명령을 사용하는 것보다 코드가 짧고 실행 시간이 작게 걸림

▷ 모든 조건 실행 ARM 명령어는 조건이 만족되지 않으면 실행되지 않음

▷ 조건 실행 명령어를 효율적으로 사용하면 코드를 최소화 가능

▷ 조건 명령을 사용한 프로그램 예

```
; if ((a==b) && (c==d)) e++;

CMP     r0, r1
CMPEQ   r2, r3
ADDEQ   r4, r4, #1
```

- 분기 링크 (branch and link) 명령어

▷ 서브 루틴을 실행하기 위해 구현되는 명령어로 서브 루틴을 실행하기 전에 복귀 주소를 저장하여야 함

▷ ARM은 링크 레지스터 (r14)에 분기 종료 후 실행될 명령어의 주소를 저장

```
        BL      SUBR            ; branch to SUBR
        ..                      ; return to here
SUBR    ..                      ; subroutine entry point
        MOV     pc, r14         ; return
```

▷ 계속해서 서부 루틴을 호출하는 경우 복귀 주소들은 차례로 스택에 저장

▷ 서브 루틴 실행 전에 저장할 필요가 있는 이전 레지스터 값들도 다중 데이터 저장 명령어를 사용하여 스택에 데이터를 저장

```
        BL      SUB1
        ..
SUB1    STMFD   r13!, {r0-r2,r14}  ; save work & link regs
        BL      SUB2
        ..
SUB2    ..
```

- 서브루틴 복귀 명령어

▷ 복귀 주소를 r15로 다시 복사하여 서브 루틴에서 복귀

▷ 복귀하기 전에 스택에 저장된 레지스터 데이터를 복원하여야 함

```
SUB1    STMFD   r13!, {r0-r2,r14}; save work regs & link
        BL      SUB2
        ..
        LDMFD   r13!, {r0-r2,pc} ; restore work regs & return
```

- Supervisor Calls

  ▷ 시스템의 차이는 있지만 대부분의 경우 사용자는 하드웨어 장치를 직접적으로 접근할 수 없으므로 supervisor 루틴을 요청

  ▷ Supervisor call은 시스템 소프트웨어로 구현되고 Supervisor 루틴을 요청할 때 SWI (software interrupt) 명령어 사용

  ▷ 디스플레이 장치에 한 문자를 출력 (r0의 하위 바이트에 대한 문자 출력)

```
SWI        SWI_WriteC          ; output r0[7:0]
```

- 점프 테이블

  ▷ 프로그램에 의해 계산된 값에 따라 서브 루틴 중 하나의 루틴으로 이동

```
            BL      JUMPTAB
            ..
JUMPTAB     CMP     r0, #0
            BEQ     SUB0
            CMP     r0, #1
            BEQ     SUB1
            CMP     r0, #2
            BEQ     SUB2
```

3.4 어셈블리 언어 프로그램 작성 방법

- 큰 프로그램은 C, C++로 작성하는 것이 편리하고 시간 제약이 큰 프로그램은 어셈블리 프로그램이 효율적임
- 어셈블러를 통해 ARM 실행 코드를 생성하고 ARM 시스템이나 Emulator에서 실행

```
        AREA    BlkCpy,CODE,READONLY
SWI_WriteC      EQU     &0              ; output character in r0
SWI_Exit        EQU     &11             ; finish program
        ENTRY                           ; code entry point
        ADR     r1, TABLE1              ; r1 -> TABLE1
        ADR     r2, TABLE2              ; r2 -> TABLE2
        ADR     r3, T1END               ; r3 -> T1END
LOOP1   LDR     r0, [r1], #4            ; get TABLE1 1st word
        STR     r0, [r2], #4            ; copy into TABLE2
        CMP     r1, r3                  ; finished?
        BLT     LOOP1                   ; if not, do more
        ADR     r1, TABLE2              ; r1 -> TABLE2
LOOP2   LDRB    r0, [r1], #1            ; get next byte
        CMP     r0, #0                  ; check for text end
        SWINE   SWI_WriteC              ; if not end, print ..
        BNE     LOOP2                   ;  .. and loop back
        SWI     SWI_Exit                ; finish
TABLE1  =       "This is the right string!", &0a, &0d, 0
T1END
        ALIGN                           ; ensure word alignment
TABLE2  =       "This is the wrong string!", 0
        END
```

- 프로그램 설계 단계

  ▷ 요구 사항을 정확하게 이해해야 함, 그렇지 않으면 프로그램 실행 시 원하는 결과를 얻을 수 없음

  ▷ 요구 사항은 명확한 형태로 정의되어야 하고 이를 토대로 프로그램 구조와 데이터 구조를 정의하고 필요한 최적인 알고리즘 개발

  ▷ 프로그램 설계가 종료되면 어셈블리 코드 작성 시작

  ▷ 작성이 완료된 코드가 요구 사항을 만족할 때까지 위의 과정을 반복

- 복잡하고 큰 프로그램은 고급 언어를 사용하여 프로그램하지만 중요한 응용에 대한 최적인 성능을 얻기 위해 일부분은 어셈블리 언어로 작성 필요