

자주 사용되는 ARM 어셈블리 명령어 요약

1. MOV

ARM 어셈블리 명령어는 MOV 명령어와 논리 및 사칙연산 명령어에 모두 쉬프트 연산이 가능한데, 이것을 나타내는 표지가 끝에 붙을 수 있다는 것에 유의한다.

쉬프트 연산에는 ASR(오른쪽 쉬프트, 빈자리는 부호가 따라옴),

LSR(오른쪽으로 쉬프트, 빈자리는 0으로 채워짐),

LSL(왼쪽으로 쉬프트, 빈자리는 0으로 채워짐),

ROR(오른쪽으로 rotation)

정도를 알아두면 유용하다.

예)

MOV r0, [r2,r4] ; r2+r4 의 주소에 있는 값을 읽어서 r0에 저장한다.

MOV r1, r2, ROR #1 ; r2를 오른쪽으로 한 비트만큼 rotation 해서 r1에 저장

2. ADD, SUB, AND, ORR

예)

ADD r1, r2, #4 ; r2에 4를 더해서 r1에 저장

SUB r1, r1, #1 ; r1의 값을 하나 감소

ORR r4, r5, r7, LSR r2 ; r7을 오른쪽으로 논리 쉬프트를 r2만큼 한 다음 그 결과를 ; r5와 or 연산하여 r4에 저장한다.

3. UMULL, SMULL

곱하기 연산이다. 32비트짜리 두 개를 곱하면 64비트짜리가 나오므로 결과값을 저장하는데 두 개의 레지스터가 필요하다. 결과 레지스터의 위치에 주의한다. UMULL은 부호가 없는 곱하기이고, SMULL은 부호가 있는 곱하기 이다.

예)

UMULL r4, r5, r1, r2 ; r1과 r2를 곱해서 상위 32비트는 r5에 저장하고 하위 32비트는 ; r4에 저장한다.

5. B, BL, BNE, BEQ, CMP

BL은 분기 명령어이다.

예)

B there ; 라벨이 there인 곳으로 무조건 분기한다.

BL sub+ROM ; 계산된 위치의 서브루틴을 호출한다.

BNE(0이 아닌 경우 분기)와 BEQ(0이면 분기) 는 branch 명령어이고 CMP는 비교 명령

어이지만 둘이 같이 쓰이는 경우가 많으므로 한꺼번에 설명한다.

예)

`CMP r1, #4` ; r1이 4이면 플래그가 0으로 셋팅된다.

`BEQ there` ; 플래그가 0이면 라벨이 `there`인 곳으로 분기하고, 그렇지 않으면
; 다음 명령어가 수행된다.

6. LDR, STR

LDR은 `load` 명령이다. LDR에는 불러오는 변수의 크기에 따라 `LDRB`, `LDRH`, `LDR`의 세 가지 종류가 있다. `LDRB`는 `byte` 변수를 불러올 때, `LDRH`는 `short` 변수를 불러올 때, `LDR`은 `int` 변수를 불러올 때 쓴다. STR는 `store` 명령으로 마찬가지로 `STRB`, `STRH`, `STR`이 있다.

첫 번째 인자는 레지스터가 두 번째 인자는 주소가 된다. 세 번째 인자는 `load/store` 연산을 한 다음 주소값을 증가시키고자 할 때, 얼마만큼 증가시킬 지를 지정한다.

예)

`LDR r1, [r2, #16]` ; r2에 16 byte만큼 더한 주소에서 정수형 값을 읽어와 r1에 저장한다.

`STR r1, [r2], #4` ; r2의 주소에 r1을 저장하고 난 후, r2를 4만큼 증가시킨다.

7. LDMFD, STMFD

LDM/STM은 LDR/STR의 변종으로 블록 단위로 `load/store` 할 때 사용한다. 중요한 용도는 스택에 레지스터 값을 저장하거나 복원하는 것이다. 왜냐하면 스택에 저장/복원할 때는 여러 개의 레지스터를 저장/복원해야 하기 때문이다.

스택과 관련해서는 `LDMFD/STMFD`, `LDMED/STMED`, `LDMFA/STMFA`, `LDMEA/STMEA` 등이 사용되고, 스택과 관련없이 사용할 때는 `LDMIA`, `LDMIB`, `LDMDA`, `LDMDB`, `STMIA`, `STMIB`, `STMDA`, `STMDB` 가 사용된다.

중요한 것은 스택과 관련해서 실제 사용할 때, 쌍으로 사용한다는 것이다. `LDMFD/STMFD` 정도만 잘 사용하면 된다. 자세한 사항은 `ADS` 문서를 참고하기 바란다.

예)

`STMFD sp!, {r4-r6, lr}` ; 스택에 r4-r6와 lr 레지스터를 저장하고 sp를 그만큼 감소시킨다.

`LDMFD sp!, {r4-r6, pc}` ; 스택에서 r4-r6와 pc를 복원하고 sp를 그만큼 증가시킨다.

ARM Developer Suite(ADS) 1.2 에서 C 코드와 ASM 코드 섞어 쓰기

1. C 코드 내에 어셈블리 코드를 inline으로 사용하기

(1) 사용방식

```
asm("instruction[;instruction]");
```

또는 C 컴파일러의 구문을 사용하면 다음과 같다.

```
__asm  
{  
instruction [; instruction]  
...  
[instruction]  
}
```

(2) 분기문에 사용되는 라벨은 끝에 ':'를 찍는다.

```
__asm  
{  
loop :  
...  
[instruction]  
}
```

(3) C 코드에서 사용하는 char, short, int 타입의 변수를 그대로 가져다 쓸 수 있다.

```
void my_strcpy(const char *src, char *dst)  
{  
    int ch;  
    __asm  
    {  
        loop:  
        // ARM version  
        LDRB ch, [src], #1  
        STRB ch, [dst], #1  
    }  
}
```

(4) 일반적인 어셈블리 코드와 다른 점

- ▶ PC(Program Counter)값을 읽어오거나 쓸 수 없다.
- ▶ LDR Rn, =variable_name 과 같은 구문을 쓸 수 없다.

- ▶ C 변수명으로 **r0, v1** 과 같은 레지스터명을 쓸 수 없다.
- ▶ **stack** 과 관련된 명령어를 쓸 수 없으며, 쓸 필요도 없다
 - ☞ 컴파일러가 자동으로 해준다
- ▶ 일반적인 함수호출과 관련된 레지스터는 원래의 의미가 사라진다. 예를 들어 **r0-r3**는 더 이상 함수호출에서 입력으로 들어오는 인자를 나타내지 않는다.
- ▶ 레지스터를 쓸 때는 컴파일러가 그 값을 다른 용도로 쓰지 않는지 주의한다.
 - ☞ C 에서 사용하던 변수를 그대로 쓰는 것이 안전하다.

(5) 예 : long 타입의 변수 곱하기

```
long long smull(int x, int y)
{
    long long res;
    __asm { SMULL ((int*)&res)[0], ((int*)&res)[1], x, y }
    return res;
}
```

2. C 코드와 어셈블리 코드를 함께 링크해서 사용하기

(1) C 전역 변수를 어셈블리 코드 안에서 사용하기

load/store 류의 명령어와 '='를 사용한다.

☞ inline asm 에서는 반대로 '='를 사용할 수 없다.

참조하고자 하는 변수가 char 타입이면 LDRB/STRB를 사용한다.

Short 타입이면 LDRH/STRH, 또는 아키텍처에 따라 LDRB/STRB를 두 번 사용한다.

int 타입이면 LDR/STR를 사용한다.

signed 변수이면 LDRSB LDRSH 와 같은 부호 명령어를 사용한다.

예)

```
AREA globals, CODE, READONLY
EXPORT asmsubroutine
IMPORT globvar
asmsubroutine
LDR r1, =globvar ; read address of globvar into
; r1 from literal pool
LDR r0, [r1]
ADD r0, r0, #2
STR r0, [r1]
MOV pc, lr
```

END

(2) C에서 어셈블리 코드를 불러다 쓰기

C 코드 쪽에서는 **extern**을 선언해주면 된다.

예)

```
#include <stdio.h>

extern void strcpy(char *d, const char *s);

int main()
{ const char *srcstr = "First string - source ";
  char dststr[] = "Second string - destination ";

  /* dststr is an array since we' re going to change it */
  printf("Before copying:\n");
  printf(" %s\n %s\n",srcstr,dststr);
  strcpy(dststr,srcstr);
  printf("After copying:\n");
  printf(" %s\n %s\n",srcstr,dststr);
  return (0);
}
```

어셈블리 코드 쪽에서는 **EXPORT** 해주면 된다.

☞ 어셈블리 코드에서는 **inline asm** 과 달리 라벨을 붙일 때, ':'가 없다.

예)

```
AREA SCopy, CODE, READONLY

EXPORT strcpy

strcpy          ; r0 points to destination string.
                ; r1 points to source string.
LDRB r2, [r1],#1 ; Load byte and update address.
STRB r2, [r0],#1 ; Store byte and update address.
CMP r2, #0 ; Check for zero terminator.
BNE strcpy ; Keep going if not.
MOV pc,lr ; Return.

END
```

3. ADS에서 함수 호출과 관련된 중요한 규약 (ATPCS : ARM-Thumb Procedure Call Standard)

(1) 레지스터 규칙

▶ r0-r3는 a1-a4라고도 하는데, 함수 호출에서 입력과 출력 인자가 된다. 인자가 5개 이

상인 것은 스택을 이용하는데, 속도가 현저히 떨어지므로 가급적 4개 이내로 인자를 쓴다.

- ▶ r4-r11은 v1-v8 이라고도 하는데, 함수 안에서 내부적인 변수로 사용한다.
- ▶ r12는 ip라고도 부르는데, 프로시저 간에 호출할 때, 다용도로 사용한다.
- ▶ r13은 sp라는 별칭의 스택 포인터이다. 다른 용도로는 사용할 수 없다.
- ▶ r14는 lr이라고도 하는데, 링크 레지스터로 return address를 저장한다. 다른 곳에 별도로 이 값이 저장된 경우 다른 용도로 사용할 수 있다.
- ▶ r15는 pc로 program counter 이다. 다른 용도로 절대 사용 못한다.

(2) 함수를 호출할 때는 내부적인 용도로 변수를 사용한 경우, 스택에 저장했다가 return 할 때, 원상 복구해야 한다. 이것은 LDM/STM 의 스택 관련 명령어를 사용하면 된다.

예)

utoa

```
STMFD    sp!, {v1-v3, lr}      ; save 2 variable registers and
                                   ; the return address
MOV      v1, a1                 ; keep char *buffer for later
MOV      v2, a2                 ; and keep the number for later
MOV      a1, a2
...
ADD      v2, v2, #'0'          ; final digit
STRB     v2, [a1], #1          ; store digit at end of buffer
LDMFD    sp!, {v1-v3, pc}      ; restore and return

END
```

함수 호출 시에 스택을 저장하고 복원하는 것은 매우 정형화된 루틴이므로 그대로 사용하면 된다. sp 레지스터 끝에 '!'는 스택에 저장/복원한 다음 그만큼 sp를 이동하라는 뜻이므로 꼭 붙여줘야 한다.

참고문헌

ARM Developer Suite 1.2 Developer Guide ch. 2, ch. 4
ARM Developer Suite 1.2 Assembler Guide ch. 4