

TI DSP, MCU 및 Xilinx Zynq FPGA 프로그래밍 전문가 과정

강사 : Innova Lee(이상훈)

gcccompil3r@gmail.com

학생 : 황수정

sue100012@naver.com

45 일차 (2018. 05. 01)

목차

- ARM assembly 명령어 형식

- 학습예제

mov 로테이션(1)

mov 로테이션(2)/add

lsl(1)

lsl(2)

lsl(3) / [주의!!]

asr

mrs / 중요!

mul

mla

umull

umlal

ldr(1)

ldr(2)

ldreqb

strb

ldr !옵션

ldr(3) / [주의!]

stmia

stmia !옵션

asm 명령어 간편하게 작성하는 법

ldmia

➤ 함수 ARM 환경에서 디버깅하기

gdb(1)

gdb(2)

ARM assembly 명령어 형식

대부분의 명령어들은 아래와 같은 형식을 취한다.

명령어<조건> Rd, Rn {, N} {, 배럴시프트연산}

Rd 는 최종 결과가 저장될 레지스터이며, Rn 과 Rm 또는 N 은 처리될 소스 데이터이다. 즉, Rd 에 옆의 연산한 값을 저장하는 방식이라고 볼 수 있다. 소스 데이터는 1 개나 2 개의 인자가 쓰일 수 있으며, 명령에 따라 Rn 이나 N 값은 상수데이터가 될 수 있다. 그리고, 배럴 시프트 연산이 마지막 인자로 덧붙을 수 있다. 명령어 뒤에 조건문이 붙으면 이전 비교문의 결과에 따라 해당 조건이 충족되는 경우에만 명령을 수행한다. 데이터 처리나 연산 명령의 경우 마지막에 S 를 붙이면 명령어 처리 후 CPSR 레지스터의 Flag 를 업데이트하도록 할 수 있다.

즉, **ARM assembly 는 기본 뼈대 명령어 뒤에 조건이나 덧붙임 명령어를 더 붙여서 활용**하는 것이다.

➤ 배럴시프트

ARM 에서는 데이터나 연산 명령 처리 시 데이터가 사용되기 전에 배럴 시프트를 이용하여 시프트 처리를 하여 명령을 수행할 수 있도록 도와주는 기능이 있다. 연산 전 Rm 의 값을 배럴 시프트를 통해 시프트 처리를 수행한 결과를 데이터로 사용할 수 있다. 배럴 시프트 처리는 아래와 같이 다섯 가지 동작 방식을 지원한다.

LSL: 왼쪽으로 논리 시프트. $x \ll y$

LSR: 오른쪽으로 논리 시프트. $(\text{unsigned})x \gg y$

ASR: 오른쪽으로 산술 시프트. $(\text{signed})x \gg y$

ROR: 오른쪽으로 로테이트. $((\text{unsigned})x \gg y) \mid (x \ll (32-y))$

RRX: 오른쪽으로 확장 로테이트. $(c \text{ 플래그 } \ll 31) \mid (\text{unsigned})x \gg 1$

주로 데이터 처리 명령어 뒤에 붙어 연산을 도와주는 역할을 한다.

`mov r0, r1, lsl #1`

➤ 배럴 시프트에 의해 r1 의 내용이 왼쪽으로 1 만큼 시프트 된 후 r0 로 복사된다.

- 데이터 처리 명령어

➤ 데이터 이동

명령어 형식: **명령어<조건>{S} Rd, N**

MOV: 레지스터 값이나 상수 값을 레지스터에 가져오는 명령어이다.

MVN: 값을 가져올 때 NOT 연산을 수행하여 가져온다.

➤ 산술 명령

명령어 형식: **명령어<조건>{S} Rd, Rn, N**

ADD: 덧셈 수행. $Rd = Rn + N$

ADC: Carry 비트도 같이 더함. $Rd = Rn + N + C$

SUB: 뺄셈 수행. $Rd = Rn - N$

SBC: Carry 비트 고려하여 뺄. $Rd = Rn - N - !C$

RSB: 뺄셈을 거꾸로 수행. $Rd = N - Rn$

RSC: Carry 비트 고려하여 RSB 수행. $Rd = N - Rn - !C$

➤ 논리 명령

명령어 형식: **명령어<조건>{S} Rd, Rn, N**

AND: $Rd = Rn \& N$

ORR: $Rd = Rn \mid N$

EOR: $Rd = Rn \wedge N$

BIC: Bit Clear(AND NOT), $Rd = Rn \& \sim N$

➤ 비교 명령

32 비트 값을 가진 레지스터를 비교하고 테스트하기 위해 사용된다. 명령의 결과에 따라 CPSR 플래그 비트를 업데이트하며, 명령에 사용된 레지스터 값에는 영향을 미치지 않는다.

명령어 형식: 명령어 <조건> Rn, N

CMP: 양수 비교, $Rn - N$ 의 결과에 따라 상태 플래그를 업데이트

CMN: 음수 비교, $Rn + N$ 의 결과에 따라 상태 플래그를 업데이트

TEQ: 두 값이 같은지 비교, $Rn \wedge N$ 의 결과에 따라 상태 플래그 업데이트

TST: 테스트 비트, $Rn \& N$ 의 결과에 따라 상태 플래그를 업데이트

➤ 곱셈 명령

MLA: 32 비트 곱셈후 덧셈을 수행하는 명령

형식: MLA<조건>{S} Rd, Rm, Rs, Rn

$Rd = (Rm * Rs) + Rn$

MUL: 32 비트 곱셈 명령

형식: MUL<조건>{S} Rd, Rm, Rs

$Rd = Rm * Rs$

학습예제

mov 로테이션(1)

```
#include <stdio.h>
```

```
int main(void)
```

```
{
    register unsigned int r0 asm("r0")=0;
    register unsigned int r1 asm("r1")=0;
    register unsigned int r2 asm("r2")=0;
    register unsigned int r3 asm("r3")=0;
    register unsigned int r4 asm("r4")=0;
    register unsigned int r5 asm("r5")=0;
```

```
    asm volatile("mov r0, #0xff, 8"); //mov 3 번째 인자를 통해 몇 비트를 로테이션 시킬 것인지 설정.
```

```
    printf("r0 = 0x%x\n", r0);
```

```
    return 0;
```

```
}
```

```
r0 = 0xff000000
```

```
QEMU: Terminated via GDBstub
```

- arm 은 32bit 단위이다. 따라서, $0xff \rightarrow 0x000000ff$ 를 뜻한다. mov 의 3 번째가 8 이므로 8bit 로케이션 한다는 뜻이다. 즉, $0x00\ 00\ 00\ ff \rightarrow 0xff\ 00\ 00\ 00$ 가 되는 것이다.

mov 로테이션(2)/add

```
#include <stdio.h>
```

```
int main(void)
```

```
{  
    register unsigned int r0 asm("r0")=0;  
    register unsigned int r1 asm("r1")=0;  
    register unsigned int r2 asm("r2")=0;  
    register unsigned int r3 asm("r3")=0;  
    register unsigned int r4 asm("r4")=0;  
    register unsigned int r5 asm("r5")=0;
```

```
    asm volatile("mov r0, #0xff, 8"); // mov 로테이션(1)과 같은 결과
```

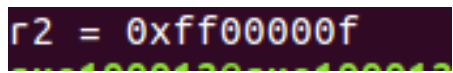
```
    asm volatile("mov r1, #0xf");    // r1 에 #0xf
```

```
    asm volatile("add r2, r1, r0"); // r2 = r1+r0
```

```
    printf("r2 = 0x%x\n", r2);
```

```
    return 0;
```

```
}
```



r2 = 0xff00000f

lsl(1)

```
#include <stdio.h>
```

```
int main(void)
```

```
{  
    register unsigned int r0 asm("r0")=0;  
    register unsigned int r1 asm("r1")=0;  
    register unsigned int r2 asm("r2")=0;  
    register unsigned int r3 asm("r3")=0;  
    register unsigned int r4 asm("r4")=0;  
    register unsigned int r5 asm("r5")=0;
```

```
    asm volatile("mov r1, #7");
```

```
    asm volatile("mov r2, #3");
```

```
    asm volatile("add r0, r1, r2, lsl #7"); //r2 를 왼쪽으로 7 만큼 쉬프트하고 r1 과 더한 뒤 r0 에 저장
```

```
    printf("r0 = 0x%x\n", r0);
```

```
    return 0;
```

```
}
```



r0 = 0x187

== 10 진법으로 397

lsl(2)

```
#include <stdio.h>
```

```
int main(void)
```

```
{
```

```
    register unsigned int r0 asm("r0")=0;
```

```
    register unsigned int r1 asm("r1")=0;
```

```
    register unsigned int r2 asm("r2")=0;
```

```
    register unsigned int r3 asm("r3")=0;
```

```
    register unsigned int r4 asm("r4")=0;
```

```
    register unsigned int r5 asm("r5")=0;
```

```
    asm volatile("mov r1, #7");
```

```
    asm volatile("mov r2, #3");
```

```
    asm volatile("mov r3, #2");
```

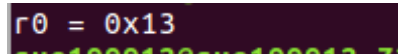
```
    asm volatile("add r0, r1, r2, lsl r3"); //r3 값 만큼 r2 를 왼쪽으로 쉬프트 시킨 뒤 r1 과 더하고 r0 에 저장
```

```
    printf("r0 = 0x%x\n", r0);
```

```
    return 0;
```

```
}
```

- 숫자 대신 레지스터를 lsl 인자로 주면 그 레지스터 안의 값으로 실행한다는 뜻이다.



== 10 진법으로 19

lsl(3) / [주의!!]

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    register unsigned int r0 asm("r0")=0;
```

```
    register unsigned int r1 asm("r1")=0;
```

```
    register unsigned int r2 asm("r2")=0;
```

```
    register unsigned int r3 asm("r3")=0;
```

```
    register unsigned int r4 asm("r4")=0;
```

```
    register unsigned int r5 asm("r5")=0;
```

```
    asm volatile("mov r1, #2");
```

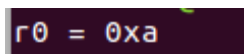
```
    asm volatile("add r0, r1, r1, lsl #2"); // r1 을 2^2 만큼 왼쪽으로 쉬프트 시키고 기존 r1 이랑 더한 뒤 r0  
    에 저장
```

```
    printf("r0 = 0x%x\n", r0);
```

```
    return 0;
```

```
}
```

- r1 값을 갱신시켜서 더하는 것이 아니다.



== 10 진법으로 10

$2^{2 \times 2} == 8$

asr

```
#include <stdio.h>
```

```
int main(void)
```

```
{  
    register unsigned int r0 asm("r0")=0;  
    register unsigned int r1 asm("r1")=0;  
    register unsigned int r2 asm("r2")=0;  
    register unsigned int r3 asm("r3")=0;  
    register unsigned int r4 asm("r4")=0;  
    register unsigned int r5 asm("r5")=0;
```

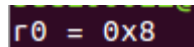
```
    asm volatile("mov r1, #32");
```

```
    asm volatile("add r0, r1, asr #2"); //r1 을 2^2 만큼 오른쪽으로 쉬프트하고 r0 과 더하여 r0 에 저장
```

```
    printf("r0 = 0x%x\n", r0);
```

```
    return 0;
```

```
}
```



== 10 진법으로 8

$32 \gg 4 == 32/4 == 8$

mrs / 중요!

```
#include <stdio.h>
```

```
void show_reg(unsigned int reg)
```

```
{  
    int i;  
    for(i = 31; i >= 0;)  
        printf("%d", (reg >> i--) & 1);  
    printf("\n");  
}
```

```
int main(void)
```

```
{  
    register unsigned int r0 asm("r0")=0;  
    register unsigned int r1 asm("r1")=0;  
    register unsigned int r2 asm("r2")=0;  
    register unsigned int r3 asm("r3")=0;  
    register unsigned int r4 asm("r4")=0;  
    register unsigned int r5 asm("r5")=0;
```

```
    asm volatile("mov r1, #32");
```

```
    asm volatile("add r0, r1, asr #2");
```

```
asm volatile("mrs r0, cpsr"); //cpsr 값을 r0 에 저장
```

```
show_reg(r0);  
return 0;  
}
```

- mrs 는 cpsr 이나 spsr 의 내용을 레지스터로 읽어오는 명령어이다.
pushf 가 생각나야 한다. 주로 인터럽트를 끄고 켤 때, 많이 사용한다.

mul

```
#include <stdio.h>
```

```
int main (void)  
{  
    register unsigned int r0 asm("r0")=0;  
    register unsigned int r1 asm("r1")=0;  
    register unsigned int r2 asm("r2")=0;  
    register unsigned int r3 asm("r3")=0;  
    register unsigned int r4 asm("r4")=0;  
    register unsigned int r5 asm("r5")=0;
```

```
    asm volatile("mov r2, #3");
```

```
    asm volatile("mov r3, #7");
```

```
    asm volatile("mul r1, r2, r3"); //r2 와 r3 를 곱한 결과 값을 r1 에 저장
```

```
    printf("r1 = %d\n", r1);
```

```
    return 0;
```

```
}
```

mal

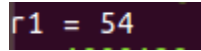
```
#include <stdio.h>
```

```
int main (void)  
{  
    register unsigned int r0 asm("r0")=0;  
    register unsigned int r1 asm("r1")=0;  
    register unsigned int r2 asm("r2")=0;  
    register unsigned int r3 asm("r3")=0;  
    register unsigned int r4 asm("r4")=0;  
    register unsigned int r5 asm("r5")=0;
```



```
asm volatile("mov r2, #3");
asm volatile("mov r3, #7");
asm volatile("mov r4, #33");
asm volatile("mla r1, r2, r3, r4"); //r2 와 r3 을 곱하고 r4 를 더함. mla 명령어를 1 클럭만에 처리
```

```
printf("r1 = %d\n", r1);
return 0;
}
```



- mla r1, r2, r3, r4 == r1= r2 *r3 +r4 원래는 20~30 클럭이 걸리는데 dsp 로는 1 클럭(mac 이 있으니까!)으로 끝난다.

umull

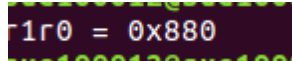
```
#include <stdio.h>
```

```
int main (void)
```

```
{
register unsigned int r0 asm("r0")=0;
register unsigned int r1 asm("r1")=0;
register unsigned int r2 asm("r2")=0;
register unsigned int r3 asm("r3")=0;
register unsigned int r4 asm("r4")=0;
register unsigned int r5 asm("r5")=0;
```

```
asm volatile("mov r2, #0x44, 8");
asm volatile("mov r3, #0x200");
asm volatile("umull r0, r1, r2, r3"); // r2 와 r3 곱한 뒤 상위 32 비트는 r1 에, 하위 32 비트는 r0 에 넣음
```

```
printf("r1r0 = 0x%x%x\n", r1, r0);
return 0;
```



```
}
```

- 참일 수도 거짓일 수도 있다. 상황에 따라 다르다. 그런데 왜 umull 을 쓰는 것일까? 이 예제를 예시로 보면 16 진수 곱셈도 10 진수 곱셈과 마찬가지로 이루어진다.

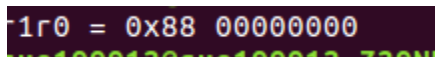
r2 = 0x44 00 00 00

r3 = 0x200

r2*r3 = 0x88 00 00 00 00 →32 비트로 하면 값이 날라가거나 오버플로우 발생한다.

32 비트 넘으면 오버 플로우가 난다. 그래서 umull 을 쓰는 것이다. 상위는 r1 에 하위는 r0 에 들어가는 것이다. 여기서 문제가 발생한다. 하위로 들어갈 때, 00 00 00 00 이 들어가는데 이것 그냥 0 으로 인식해서 결과가 값이 0x880 이 나오는 것이다. 이 문제를 해결하는 방법은 아래와 같다.

printf("r1r0 = 0x%x %08x\n", r1, r0); 로 변경 해주면 된다.



umal

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    register unsigned int r0 asm("r0")=0;
```

```
    register unsigned int r1 asm("r1")=0;
```

```
    register unsigned int r2 asm("r2")=0;
```

```
    register unsigned int r3 asm("r3")=0;
```

```
    register unsigned int r4 asm("r4")=0;
```

```
    register unsigned int r5 asm("r5")=0;
```

```
    asm volatile("mov r0, #0xf");
```

```
    asm volatile("mov r1, #0x1");
```

```
    asm volatile("mov r2, #0x44, 8");
```

```
    asm volatile("mov r3, #0x200");
```

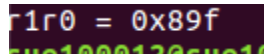
```
    asm volatile("umlal r0, r1, r2, r3"); //r2 와 r3 를 곱한 뒤 상위비트는 r1 과 합한 뒤 r1 에 저장하고  
                                           하위비트는 r0 과 합한 뒤 r0 에 저장
```

```
    printf("r1r0 = 0x%x%x\n", r1, r0);
```

```
    return 0;
```

```
}
```

➤ umlal r0, r1, r2, r3 == r0 = r2*r3 하위비트 + r0, r1= r2*r3 상위비트 + r1



ldr(1)

```
#include <stdio.h>
```

```
unsigned int arr[5] = {1, 2, 3, 4, 5};
```

```
int main(void)
```

```
{
```

```
    register unsigned int r0 asm("r0")=0;
```

```
    register unsigned int *r1 asm("r1")=NULL;
```

```
    register unsigned int *r2 asm("r2")=NULL;
```

```
    register unsigned int r3 asm("r3")=0;
```

```
    register unsigned int r4 asm("r4")=0;
```

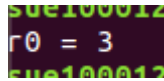
```
    register unsigned int r5 asm("r5")=0;
```

```
    r1 = arr;
```

```
    asm volatile("mov r2, #0x8");
```

asm volatile("ldr r0, [r1, r2]"); // r1 에 r2 바이트만큼 이동한 위치에 정보를 r0 에 저장

```
printf("r0 = %u\n", r0);
return 0;
}
```



- ldr 은 메모리의 정보 값을 레지스터에 넣어준다는 뜻이다. mov 로는 메모리의 값을 읽지 못하기 때문이다. r1 에 배열의 시작주소를 저장한다. r2 가 0x8 을 저장하고 r1 에서 8byte 를 이동하면 int 형 배열이므로 2 칸이동해서 3 이 출력된 것이다.
- ldr 명령어는 메모리에서 워드를 레지스터로 읽어 들이는 명령어이다. 메모리의 주소를 레지스터로 읽어온다. **메모리에서 레지스터**
그 반대는 str 이다.
str 명령어는 레지스터에서 워드를 메모리에 저장하는 명령어이다. **레지스터에서 메모리**

ldr(2)

```
#include <stdio.h>
```

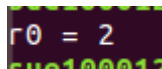
```
unsigned int arr[5] = {1, 2, 3, 4, 5};
```

```
int main(void)
{
    register unsigned int r0 asm("r0")=0;
    register unsigned int *r1 asm("r1")=NULL;
    register unsigned int *r2 asm("r2")=NULL;
    register unsigned int r3 asm("r3")=0;
    register unsigned int r4 asm("r4")=0;
    register unsigned int r5 asm("r5")=0;
```

```
r1 = arr;
```

```
asm volatile("ldr r0, [r1, #0x4]"); //r1 에서 4 바이트 이동해서 그 값을 r0 저장
```

```
printf("r0 = %u\n", r0);
return 0;
}
```



ldreqb

```
#include <stdio.h>
```

```
char *test = "HelloARM";
```

```
int main(void)
```

```

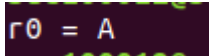
{
    register unsigned int r0 asm("r0")=0;
    register char *r1 asm("r1")=NULL;
    register unsigned int *r2 asm("r2")=NULL;
    register unsigned int r3 asm("r3")=0;
    register unsigned int r4 asm("r4")=0;
    register unsigned int r5 asm("r5")=0;

    r1 = test;

    asm volatile("ldreqb r0, [r1, #0x5]"); //r1 에서 5 바이트 왼쪽으로 이동한 뒤 r0 에 저장

    printf("r0 = %c\n", r0);
    return 0;
}

```



- ldreqb 는 cpsr 의 **z 플래그**를 읽고 셋팅이 되어 있으면 실행한다.

strb

```
#include <stdio.h>
```

```
char test[] = "HelloARM";
```

```
int main(void)
```

```

{
    register unsigned int r0 asm("r0")=0;
    register char *r1 asm("r1")=NULL;           // char 형 포인터로 한 글자를 받아올 수 있다.
    register unsigned int *r2 asm("r2")=NULL;
    register unsigned int r3 asm("r3")=0;
    register unsigned int r4 asm("r4")=0;
    register unsigned int r5 asm("r5")=0;

```

```
    r1 = &test[5];           // r1 = test; r1 에 test 배열에서 5 번째 주소값을 받아오는 것이다.
```

```

    asm volatile("mov r0, #61");
    asm volatile("strb r0, [r1]"); //strb r0, [r1, #5]

```

```

    printf("test = %s\n", test);
    return 0;
}

```

결과

test = Hello=RM

- char *test = "HelloARM"; 보단 char test[] = "HelloARM";을 사용하는 것이 더 좋다. 이는 *test 는 문자열 상수가 되고 test[] 문자열 변수가 되기 때문이다. 이는 포인터 test 를 통해 문자열 상수 "HelloARM"를 가르키겠다는 의미로 문자열의 첫 문자인 'H'를 가르키게 된다. 즉, test 는 메모리에 저장되어 있는 문자열 상수 "HelloARM"을 단순히 가르키고만 있는 것이기 때문이다.
- b 는 byte 를 뜻한다. 아스키코드의 61 번은 =이다.
strb 는 ldr 의 반대로 레지스터에서 메모리로 간다. r0 를 r1 으로 넣는데 r1 이 =으로 바뀌면서 test 배열에 영향을 주어 A 이 =으로 바뀌어서 출력 된 것이다.

ldr !옵션

```
#include <stdio.h>
```

```
char *test = "HelloARM";
```

```
int main(void)
```

```
{  
    register unsigned int r0 asm("r0")=0;  
    register char *r1 asm("r1")=NULL;  
    register unsigned int *r2 asm("r2")=NULL;  
    register unsigned int r3 asm("r3")=0;  
    register unsigned int r4 asm("r4")=0;  
    register unsigned int r5 asm("r5")=0;
```

```
    r1 = test;          // r1 = test;
```

```
    asm volatile("mov r2, #0x5");
```

```
    asm volatile("ldr r0, [r1, r2]!"); //r1 을 r2 의 값인 5 바이트만큼 이동하고 그 이 후 값으로 셋팅 r0 에 저장
```

```
    printf("test = %s, r1 = %s\n", test, r1);
```

```
    return 0;
```

```
}
```

결과

```
test = HelloARM, r1 = ARM
```

- ARM 에서 !는 이동한 곳까지 값을 갱신한다. lseek 함수 처럼 r1 이 변경 되었다는 것을 알기 위해 r1 값을 출력한 것이다.
- 함수의 리턴 값은 항상 r0 에 저장된다. printf()를 하면 r0 의 값이 바뀐다.

ldr(3) / [주의!]

```
#include <stdio.h>
```

```
unsigned int arr[5] = {1, 2, 3, 4, 5};
```

```
int main(void)
```

```
{  
    register unsigned int r0 asm("r0")=0;  
    register unsigned int *r1 asm("r1")=NULL;  
    register unsigned int *r2 asm("r2")=NULL;  
    register unsigned int r3 asm("r3")=0;  
    register unsigned int r4 asm("r4")=0;  
    register unsigned int r5 asm("r5")=0;
```

```
    r1 = arr;
```

```
    asm volatile("mov r2, #0x4");
```

asm volatile("ldr r0, [r1], r2"); //r1 이 시작주소 , (주의!) 많이 쓰는 문법이다. r0 에 r1 주소 첫 값이 들어가고 r1 에 r2 가 들어간다. 따로 들어가는 것이다!

```
    printf("r0 = %u, r1 = %u\n", r0, *r1);
```

```
    return 0;
```

```
}
```

결과

r0 = 1, r1 = 2

- r1 은 배열의 시작주소이다. 그래서 r0 가 1 이 출력 된 것이다. r2 의 값은 4 로, r1 의 위치에서 4 바이트만큼 이동한 값이 갱신되어 2 가 출력된 것이다.

stmia

```
#include <stdio.h>
```

```
int main (void)
```

```
{  
    int i;  
    unsigned int test_arr[5] = {0};  
  
    register unsigned int *r0 asm("r0")=0;  
    register unsigned int r1 asm("r1")=0;  
    register unsigned int r2 asm("r2")=0;  
    register unsigned int r3 asm("r3")=0;  
    register unsigned int r4 asm("r4")=0;  
    register unsigned int r5 asm("r5")=0;
```

```

r0 = test_arr;

asm volatile("mov r1, #0x3");
asm volatile("mov r2, r1, lsl#2");
asm volatile("mov r4, #0x2");
asm volatile("add r3, r1, r2, lsl r4");
asm volatile("stmia r0, {r1, r2, r3}"); // r0 가 가리키는거에 순서대로 r1,r2,r3 를 집어넣음

for(i=0; i<5; i++)
    printf("test_arr[%d] = %d\n", i, test_arr[i]);

return 0;
}

```

결과

```

test_arr[0] = 3
test_arr[1] = 12
test_arr[2] = 51
test_arr[3] = 0
test_arr[4] = 0

```

- stmia 연산으로 r1, r2, r3 값들을 배열에 순서대로 넣어서 값이 출력된 것이다. 4 번째와 5 번째 배열은 집어 넣어준 값이 없으므로 초기값 0 을 그대로 가지고 있다.
- stmia 는 store multiple increment after 란 뜻으로, 위 예제 stmia r0, {r1, r2,r3}는 r0 주소에 인덱스 별 저장공간에 레지스터를 순서대로 저장한다는 것이다. 즉, **레지스터 값을 메모리에 넣는 연산**이다. increment after 이므로 레지스터의 값을 메모리(스택)에 넣을 때 주소를 증가시키고 값을 집어 넣는다. 이는 **레지스터에서 메모리로 저장(스택) 하고 index 증가**시킨다는 뜻이다.

stmia !옵션

```
#include <stdio.h>
```

```

int main (void)
{
    int i;
    unsigned int test_arr[5] = {0};

    register unsigned int *r0 asm("r0")=0;
    register unsigned int r1 asm("r1")=0;
    register unsigned int r2 asm("r2")=0;
    register unsigned int r3 asm("r3")=0;
    register unsigned int r4 asm("r4")=0;
    register unsigned int r5 asm("r5")=0;

```

```

r0 = test_arr;

asm volatile("mov r1, #0x3");
asm volatile("mov r2, r1, lsl#2");
asm volatile("mov r4, #0x2");
asm volatile("add r3, r1, r2, lsl r4");
asm volatile("stmia r0!, {r1, r2, r3}"); // r0 에 차례대로 r1,r2,r3 을 넣고 이동한 위치로 갱신
asm volatile("str r4, [r0]");          //!의 역할은 마지막 값으로 갱신한다.

for(i=0; i<5; i++)
    printf("test_arr[%d] = %d\n", i, test_arr[i]);

return 0;
}

```

결과

```

test_arr[0] = 3
test_arr[1] = 12
test_arr[2] = 51
test_arr[3] = 2
test_arr[4] = 0

```

➤ ! 이동한 위치로 갱신시킨다. 그래서 2 가 배열 4 번째에 들어간 것이다.

asm 명령어 간편하게 작성하는 법

```
#include <stdio.h>
```

```

int main (void)
{
    int i;
    unsigned int test_arr[5] = {0};

    register unsigned int *r0 asm("r0")=0;
    register unsigned int r1 asm("r1")=0;
    register unsigned int r2 asm("r2")=0;
    register unsigned int r3 asm("r3")=0;
    register unsigned int r4 asm("r4")=0;
    register unsigned int r5 asm("r5")=0;

    r0 = test_arr;

    asm volatile("mov r1, #0x3\n"

```



```

"mov r2, r1, lsl#2\n"
"mov r4, #0x2\n"
"add r3, r1, r2, lsl r4\n"
"stmia r0!, {r1, r2, r3}\n"
"str r4, [r0]");

```

```

for(i=0; i<5; i++)
    printf("test_arr[%d] = %d\n", i, test_arr[i]);

return 0;
}

```

결과

```

test_arr[0] = 3
test_arr[1] = 12
test_arr[2] = 51
test_arr[3] = 2
test_arr[4] = 0

```

- 결과는 위의 예제와 같다. 코드가 뜻하는 것이 같기 때문이다. 다만 이 예제는 앞으로 asm volatile 를 일일이 다 작성하지 않아도 된다는 뜻으로 쓰인 것이다. 다만 개행은 해주어야 한다.

ldmia

```
#include <stdio.h>
```

```

int main (void)
{
    int i;
    unsigned int test_arr[7] = {0};

    register unsigned int *r0 asm("r0")=0;
    register unsigned int r1 asm("r1")=0;
    register unsigned int r2 asm("r2")=0;
    register unsigned int r3 asm("r3")=0;
    register unsigned int r4 asm("r4")=0;
    register unsigned int r5 asm("r5")=0;
    register unsigned int r6 asm("r6")=0;

    r0 = test_arr;

    asm volatile("mov r1, #0x3\n"
                 "mov r2, r1, lsl#2\n"

```

```

"mov r4, #0x2Wn"
"add r3, r1, r2, lsl r4Wn"
"stmia r0!, {r1, r2, r3}Wn" // r0 에 r1, r2, r3 레지스터 값을 저장하고 위치 값을 갱신
"str r4, [r0]Wn"           //r0 에 r4 레지스터의 값을 저장
"mov r5, #128Wn"
"mov r6, r5, lsr #3Wn"
"stmia r0, {r4, r5, r6}Wn" //r0 에 r4, r5, r6 을 저장
"sub r0, r0, #12Wn"        //r0-12 즉, r0 의 위치 값에서 4 바이트*3 인 주소 값을 빼서
                           r0 에 넣는다. r0 위치를 원상복구 시킨다.
"ldmia r0, {r4, r5, r6}"); //r0 가 가리키는 메모리 값을 r4, r5, r6 에 차례대로 저장

```

```

for(i=0; i<7; i++)
    printf("test_arr[%d] = %dWn", i, test_arr[i]);

printf("r4 = %u, r5 = %u, r6 = %uWn", r4, r5, r6);
return 0;
}

```

결과

```

test_arr[0] = 3
test_arr[1] = 12
test_arr[2] = 51
test_arr[3] = 2
test_arr[4] = 128
test_arr[5] = 16
test_arr[6] = 0
r4 = 3, r5 = 12, r6 = 51

```

[중요] 함수 ARM 환경에서 디버깅하기

```

#include <stdio.h>

int my_func(int num)
{
    return num * 2;
}

int main (void)
{
    int res, num = 2;
    res = my_func(num);
}

```

```

    printf("res = %d\n", res);
    return 0;
}

#include <stdio.h>

int my_func(int n1, int n2, int n3, int n4, int n5)
{
    return n1 + n2 + n3 + n4 + n5;
}

int main (void)
{
    int res, n1 = 2, n2 = 3, n3 = 4, n4 = 5, n5 = 6;
    res = my_func(n1, n2, n3, n4, n5);
    printf("res = %d\n", res);
    return 0;
}

```

[중요] 두 코드의 차이점 아는 것이 중요하다!

- bl(branch link) 은 함수로 들어가는 명령어이다. 함수로 들어갈 때 lr 레지스터에 복귀주소를 남긴다. jmp+ call 과 같다.
- Intel 은 함수의 인자를 스택으로 전달 하고, arm 에서 함수의 인자는 레지스터로 전달한다. 하지만, 인자를 4 개 이상 쓰면 스택을 사용하게 되어서 속도가 떨어진다.