TI DSP, MCU 및 Xilinx Zynq FPGA 프로그래밍 전문가 과정

2018-05-01 (45 회차)

강사: Innova Lee(이상훈)
gcccompil3r@gmail.com
학생: 정유경
ucong@naver.com

1. rotation.c

```
#include <stdio.h>

int main(void)
{
        register unsigned int r0 asm("r0")=0;

        asm volatile("mov r0, #0xff, 8");
        printf("r0=0x%x\n",r0);

        return 0;
}
```

결과: r0=0xff000000

ARM 은 32bit(4byte)
버림없이 쉬프트 하는 것을 ‘rotation’이라고 한다.

// 0xff 를 오른쪽으로 8 비트만큼 rotation 해서 r0 에 저장한다.

2. rot_add.c

```
#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;

        asm volatile("mov r0,#0xff,8");
        asm volatile("mov r1, #0xf");
        asm volatile("add r2,r1,r0");

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

        return 0;
}
```

결과: r2=0xff00000f

// 0xff 를 8 비트 로테이션 해서 r0 에 넣는다
(0xff 00 00 00)
// r1= 0xf
// r2 = r1+r0

3. lsl.c

```
#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;

        asm volatile("mov r1, #7");
        asm volatile("mov r2, #3");
        asm volatile("add r0,r1,r2,lsl #7");
        printf("r0=0x%x\n",r0);
        return 0;
}
```

결과: r0=0x187

// r1 = 7
// r2 = 3
// logical shift(논리쉬프트)
r0 = r1 + (r2 << 7)
391 = 7+(3*128)

4. lsl_reg.c

```
#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;

        asm volatile("mov r1, #7");
        asm volatile("mov r2, #3");
        asm volatile("mov r3, #2");
        asm volatile("add r0,r1,r2,lsl r3");

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

        return 0;
}
```

결과: r0=0x13

```
// r1 =7
// r2 =3
// r3 =2
// r0 = r1 + (r2 를 왼쪽으로 r3 비트 쉬프트)
19 = 7 + (3 << 2)
```

5. lsl_2.c

```
#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;

        asm volatile("mov r1, #2");
        asm volatile("add r0,r1, r1,lsl #2");

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

결과 : r0=0xa

```
// r1 = 2
// r0 = r1 + (r1 << 2)
// 10 = 2 + (2  * 4)
<주의>
r1 은 fix 된 값이다!
뒷부분 연산되었다고 2 가 8 이 되지 않는다
```

6. asr.c

```
#include <stdio.h>

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

결과: r0=0x8

```
// arithmatic shift right
```

```
	asm volatile("mov r1, #32");
	asm volatile("add r0,r1,asr #2");
	printf("r0=0x%x\n",r0);

	return 0;
}
```

// r1 =32
// r0 = (r1 >> 2)

7. cpsr.c

```
#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;

	asm volatile("mov r1, #32");
	asm volatile("add r0,r1,asr #2");
	asm volatile("mrs r0,cpsr");

	show_reg(r0);

	return 0;
}
```

결과: 01100000000000000000000000010000

mrs : cpsr 레지스터의 상태값을 특정 레지스터로 복사해 온다
(x86: eflars regiser - pushf, ARM: cpsr - mrs)

인텔 함수 리턴값 저장하는 레지스터는 ax
ARM 은 대표적으로 r0

printf 도 함수
함수 호출이후 r0 에는 리턴값이 저장

ARM 의 곱셈명령
/*앞에서 했던 기본적인 명령어들외에도 ARM 에는 MAC 이 있었다
MAC 이 사용하는게 곱셈 명령어 */

8. multiply.c

```
#include <stdio.h>

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

결과: r1=21

// ARM 은 운영체제가 관리하는 별도 명령어 mrs 없이 접근 못하게 막아놓음
asm 으로 접근불가

```
	asm volatile("mov r2, #3");
	asm volatile("mov r3, #7");
	asm volatile("mul r1,r2,r3");

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

	return 0;
}
```

9. unsign_mul_long.c

```
#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;

	asm volatile("mov r2, #0x44,8");
	asm volatile("mov r3, #0x200");
	asm volatile("umull r0,r1,r2,r3");

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

// 올바른 표기:  printf("r1 r0 = 0x %x %08x",r1, r0);
	return 0;
}
```

결과: r1 r0 = 0x 88 0
올바른 표기: r1 r0 = 0x 88 00000000

unsigned multiply long : umull

%d 가 00000 이런식으로 표시해 주기 때문에 결과가 맞을수도 틀릴수도 있다

r2 : 0x 44 00 00 00
r3 = 0x200
곱하면 0x 88 00 00 00 00 (40bit)
(그냥 mul 이었다면 오버플로우)
하위 상위 나누어 상위가 r1:88 하위가 r0:0

올바른 표기가 아니다 0 을 8 개 출력해줘야 한다
올바른표기: printf("r1 r0 = 0x %x %08x",r1, r0);

10. umlal.c

```
#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;

	asm volatile("mov r0, #0xf");
	asm volatile("mov r1, #0x1");
	asm volatile("mov r2, #0x44,8");
```

결과: r1 r0 = 0x 89 f
(r1 r0 = 0x89 0000000f)

곱한다음 상위끼리 더하고 하위끼리 더한다

```
        asm volatile("mov r3, #0x200");
        asm volatile("umlal r0,r1,r2,r3");

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

        return 0;
}
```

11. ldr.c

```
#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")=0;

        r1 = arr;
        asm volatile("mov r2, #0x8");
//1.    asm volatile("ldr r0,[r1,r2]");
//2.    asm volatile("ldr r0,[r1,#0x4]");

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

        return 0;
}
```

결과: r0 = 3 / r0 = 2

// r1 에 배열의 주소를 저장함, 포인터
// r2 = 8
// ldr : 메모리정보를 레지스터로 가져온다
// r1 을 8 바이트 이동(int 형이므로 4 바이트당 1 칸)

// r2 에 * 를 빼도 동작된다.

12. ldr2.c

```
#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;

        r1=test;

        asm volatile("mov r2,#0x5");
        asm volatile("ldr r0,[r1,r2]!");

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

결과: test = HElloARM, r1 = ARM

// 이번엔 ldr 에 !붙어있음

LDR : 메모리에서 레지스터로 로드
STR : 레지스터 메모리에 저장

// r2 =5
// r1 을 5 바이트 이동한 위치에 r1 을 고정한다.
이때 r1 이 가리키는 건 'A'

return 0; }	

13. ldreqb.c

#include <stdio.h>
 	
char test[]= "HElloARM";
 	
int main(void)
{
register unsigned int r0 asm("r0")=0;
register char *r1 asm("r1")=NULL;
 	
r1 = test;
asm volatile("ldreqb r0,[r1,#0x5]");
printf("r0 = %c\n",r0);
 	
return 0;
}
	결과: r0 = A
	
	// 위에 사실 뭔가 조건이 있었어야 겠지만
	어차피 처음에는 제로 플래그가 설정되어 있으니까
	eq 붙여준것, b 는 바이트
	//1 바이트씩 5 개 이동

14. strb.c

#include <stdio.h>
 	
char test[]= "HElloARM";
 	
int main(void)
{
register unsigned int r0 asm("r0")=0;
register char *r1 asm("r1")=NULL;
 	
r1 = &test[5]; // r1=test;
 	
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
	
	STRB : 레지스터에서 메모리로 한바이트를 저장
	
	61 이 아스키로 '='
	
	r0 를 r1 위치에 대입
	넣는데 r1 은 test[5] 의 A → =

15. ldr3.c

#include <stdio.h>	결과: r0 = 1, r1 = 2

```
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;

        r1=arr;
        asm volatile("mov r2,#0x4");
        asm volatile("ldr r0,[r1],r2");

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

        return 0;
}
```

(중요하다)
알 2 에 4
알 0 에 알 1 배열의 시작주소 들어가니까 알 0 에 들어가고
알 2 가 4 니까 시작주소에서 4 바이트 갱신되면 2

// r1 에 배열의 시작주소
// r2 = 4
// r0 에는 r1(배열의 시작주소)들어가고, 배열 시작주소에서 4 바이트 갱신된 값은 2

17. stmia.c

```
#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;

        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}");
        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

store multiple : 레지스터 값 여러개를 한꺼번에 메모리(즉 보편적으로 스택)에 넣는다

먼저 증가시키고 값을 넣는다
ia: incremenr after

r1 =3
r2 = 3 >> 2 =12
r4 =2
r3 = 3 +(12 >>4) =51

r0 에 저장된 test_arr 의 시작주소에
r1, r2, r3 순서대로 넣는다

18. stmia2.c

```
#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}");
	asm volatile("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


LDM : 다중 레지스터 로드
STM : 다중 레지스터 저장



r1 = 3
r2 =12
r4 =2
r3 =51
r0 에 값을 넣고 위치 갱신!
(! 기호가 있을 경우 최종 주소가 R0 에 다시 기록)
//2

asm volatile 선언은 다음과 같은 방법도 가능하다
	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]");
```

21. ldmia.c

```
#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");
	asm volatile("mov r2, r1,lsl #2"); //12
	asm volatile("mov r4, #0x2"); //2
	asm volatile("add r3,r1, r2,lsl r4"); // 51=3+48
	asm volatile("stmia r0!,{r1,r2,r3}");
//  STMIA r1!, {r4-r11}	; r4~r11 레지스터에 적재된 값들을 목적지에 저장.
	asm volatile("str r4, [r0]");
	asm volatile("mov r5, #128");
	asm volatile("mov r6, r5,lsr #3"); // 16 =128 /8
	asm volatile("stmia r0, {r4,r5,r6}"); // r0 는 4 번째에 위치
	asm volatile("sub r0, r0,#12"); // 배열의 시작주소로
	asm volatile("ldmia r0,{r4,r5,r6}"); // ldm : multi load (m->reg)

	for(i=0;i<7;i++)
		printf("test_arr[%d]= %d\n", i,test_arr[i]);
	printf("r4=%u, r5=%u, r6=%u\n", r4,r5,r6); // ! = 연산한 위치까지 옮겨라

	return 0;
}
/*
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

*/
```

22. func.c

```
#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;
}

arm-linux-gnueabi-gcc -g func.c
 qemu-arm-static -g 1234 -L /usr/arm-linux-gnueabi ./a.out
 b *0x00010460
 disas
 info registers
 si (si: 는 인스트럭션을 수행하고, 함수 내부로 들어감 ↔ ni )


Dump of assembler code for function main:
sp          0xf6ffeeb8
=> 0x00010460 <+0>:    push    {r11, lr}
sp          0xf6ffeeb0
  0x00010464 <+4>:     add     r11, sp, #4
sp          0xf6ffeeb0
r11         0xf6ffeeb4
  0x00010468 <+8>:     sub     sp, sp, #8
sp          0xf6ffeea8
  0x0001046c <+12>:    mov     r3, #2
r3          0x2     2
  0x00010470 <+16>:    str     r3, [r11, #-12]
sp          0xf6ffeea8
r11         0xf6ffeeb4
[r11, #-12] = sp
  0x00010474 <+20>:    ldr     r0, [r11, #-12]
r0          0x2   2
  0x00010478 <+24>:    bl      0x10438 <my_func>
r0          0x4     4
  0x0001047c <+28>:    str     r0, [r11, #-8]
(gdb) p/x *0xf6ffeeac
$12 = 0x4
  0x00010480 <+32>:    ldr     r1, [r11, #-8]
r1          0x4     4
pc          0x10484
  0x00010484 <+36>:    ldr     r0, [pc, #16]      ; 0x1049c <main+60>
r0          0x10510 ..?
  0x00010488 <+40>:    bl      0x102e0 <printf@plt>
  0x0001048c <+44>:    mov     r3, #0
r3          0x0     0
  0x00010490 <+48>:    mov     r0, r3
r0          0x0     0
  0x00010494 <+52>:    sub     sp, r11, #4
r11         0xf6ffeeb4     -150999372
```

```
sp          0xf6ffeeb0      0xf6ffeeb0
  0x00010498 <+56>:    pop     {r11, pc}
프롤로그
  0x0001049c <+60>:    andeq   r0, r1, r0, lsl r5
End of assembler dump.




Dump of assembler code for function my_func:
pc          0x10438
lr          0x1047c (복귀주소 저장)
sp          0xf6ffeea8
에필로그
=> 0x00010438 <+0>:    push    {r11}               ; (str r11, [sp, #-4]!)
r11         0xf6ffeeb4
sp          0xf6ffeea4(4 감소)
  0x0001043c <+4>:     add     r11, sp, #0
r11         0xf6ffeea4
  0x00010440 <+8>:     sub     sp, sp, #12
sp          0xf6ffee98
  0x00010444 <+12>:    str     r0, [r11, #-8]
r0          0x2     2
p/x *0xf6ffee90 이상한 값 출력되네..?
r11 은 현재 sp
  0x00010448 <+16>:    ldr     r3, [r11, #-8]
r3          0x2     2
  0x0001044c <+20>:    lsl     r3, r3, #1
r3 = r3 >>1
r3          0x4     4
  0x00010450 <+24>:    mov     r0, r3
r0          0x4     4 (r0: 함수의 반환값을 저장)
  0x00010454 <+28>:    sub     sp, r11, #0
sp          0xf6ffeea4
  0x00010458 <+32>:    pop     {r11}               ; (ldr r11, [sp], #4)
sp          0xf6ffeea8
  0x0001045c <+36>:    bx      lr
pc          0x1047c
프롤로그
End of assembler dump.
```

23. func2.c

```
#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;
}
```

```
Dump of assembler code for function main:
=>  0x00010488 <+0>:   push    {r11, lr}
  0x0001048c <+4>:     add     r11, sp, #4
  0x00010490 <+8>:     sub     sp, sp, #32
  0x00010494 <+12>:    mov     r3, #2
  0x00010498 <+16>:    str     r3, [r11, #-28]    ; 0xffffffe4
  0x0001049c <+20>:    mov     r3, #3
  0x000104a0 <+24>:    str     r3, [r11, #-24]    ; 0xffffffe8
  0x000104a4 <+28>:    mov     r3, #4
  0x000104a8 <+32>:    str     r3, [r11, #-20]    ; 0xffffffec
  0x000104ac <+36>:    mov     r3, #5
  0x000104b0 <+40>:    str     r3, [r11, #-16]
  0x000104b4 <+44>:    mov     r3, #6
  0x000104b8 <+48>:    str     r3, [r11, #-12]
  0x000104bc <+52>:    ldr     r3, [r11, #-12]
  0x000104c0 <+56>:    str     r3, [sp]
  0x000104c4 <+60>:    ldr     r3, [r11, #-16]
  0x000104c8 <+64>:    ldr     r2, [r11, #-20]    ; 0xffffffec
  0x000104cc <+68>:    ldr     r1, [r11, #-24]    ; 0xffffffe8
  0x000104d0 <+72>:    ldr     r0, [r11, #-28]    ; 0xffffffe4
  0x000104d4 <+76>:    bl      0x10438 <my_func>
  0x000104d8 <+80>:    str     r0, [r11, #-8]
  0x000104dc <+84>:    ldr     r1, [r11, #-8]
  0x000104e0 <+88>:    ldr     r0, [pc, #16]      ; 0x104f8 <main+112>
  0x000104e4 <+92>:    bl      0x102e0 <printf@plt>
  0x000104e8 <+96>:    mov     r3, #0
  0x000104ec <+100>:   mov     r0, r3
  0x000104f0 <+104>:   sub     sp, r11, #4
  0x000104f4 <+108>:   pop     {r11, pc}
  0x000104f8 <+112>:   andeq   r0, r1, r12, ror #10
```

```
End of assembler dump.


Dump of assembler code for function my_func:
=> 0x00010438 <+0>:    push    {r11}           ; (str r11, [sp, #-4]!)
   0x0001043c <+4>:    add     r11, sp, #0
   0x00010440 <+8>:    sub     sp, sp, #20
   0x00010444 <+12>:   str     r0, [r11, #-8]
   0x00010448 <+16>:   str     r1, [r11, #-12]
   0x0001044c <+20>:   str     r2, [r11, #-16]
   0x00010450 <+24>:   str     r3, [r11, #-20]   ; 0xffffffec
   0x00010454 <+28>:   ldr     r2, [r11, #-8]
   0x00010458 <+32>:   ldr     r3, [r11, #-12]
   0x0001045c <+36>:   add     r2, r2, r3
   0x00010460 <+40>:   ldr     r3, [r11, #-16]
   0x00010464 <+44>:   add     r2, r2, r3
   0x00010468 <+48>:   ldr     r3, [r11, #-20]   ; 0xffffffec
   0x0001046c <+52>:   add     r2, r2, r3
   0x00010470 <+56>:   ldr     r3, [r11, #4]
   0x00010474 <+60>:   add     r3, r2, r3
   0x00010478 <+64>:   mov     r0, r3
   0x0001047c <+68>:   sub     sp, r11, #0
   0x00010480 <+72>:   pop     {r11}           ; (ldr r11, [sp], #4)
   0x00010484 <+76>:   bx      lr
End of assembler dump.
```

ARM 은 32bit RISC 프로세서를 사용하며, Load Store Architecture 이다.

Func2.c 의 예제를 보면 ARM 은 함수 호출시 x86 과는 다른 독특한 방법을 사용한다는 것을 알 수 있다.
ARM 에서 함수를 호출할 때 인자는 4 개까지는 레지스터를 이용하여 전달되고, 4 개 이상이면 스택(메모리)을 사용한다.
메모리 아키텍쳐에서 레지스터에 비해 메모리는 속도가 느리다.
따라서 ARM 에서 성능을 고려하여 프로그래밍 하려면 인자를 4 개 이하로 사용하는 것이 좋다.
또는 구조체를 사용하여 인자를 4 개이상 전달할 수 있다.

ARM Calling Convention

함수를 호출하는 방식에 대한 일종의 규약. 인수 전달 방법, 반환 값, 인수 전달을 위해 사용한 메모리 정리 등을 규정한 것으로, 호출하는 쪽과 호출되는 쪽의 어느 한쪽이 약속을 어길 경우 함수도 제대로 동작하지 않을 뿐더러 메모리가 엉망이 되기 때문에 프로그램은 정상적인 실행을 계속할 수 없다. 호출 규약은 컴파일러 내부에서 일어나는 일로 컴파일러의 내부 동작과 함수의 호출 과정을 이해함으로써 재귀 호출이나 가변 인수 등의 고급 기법과 저수준 디버깅에도 활용할 수 있으며, C 나 C++가 아닌 다른 언어로 만든 함수를 호출하는 방법도 알게 된다.

ARM 에서 사용하는 레지스터
함수의 인자로 사용하는 레지스터는 r0~r3

이 중에서 r0 는 함수의 반환되는 인자값둘이 저장됨
따라서 일반적인 경우 r4~r7 을 사용하는게 좋다
r7 은 시스템콜번호를 저장할때 에 사용한다.
r13, r14, r15 는 각각 sp(스택포인터), lr(링크레지스터), pc(프로그램카운터) 로 사용한다
r11 은 인텔의 bp 와 동일한 각용도로 사용할 수도 있다.

- ARM 에서 함수 호출 과정 -

1) 프롤로그 (서브루틴을 호출하기 직전)에 r4 부터 r11 까지 스택에 저장(push)하고 r14(리턴어드레스)를 스택에 저장(push)한다.
2) r0 - r3 중에 함수에 전달할 인자값이 있으면 이것을 r4 - r11 (임의)로 복사한다.
3) 나머지 지역변수들은 r4 - r11 중 남아있는 곳에 할당한다.
4) 연산을 수행한 후 다른 서브루틴이 있다면 호출한다.
5) r0 에 리턴값(결과)를 저장한다.
6) 에필로그(원래있던 곳으로 복귀)에 스택에서 r4 - r11 을 꺼내고 r15(프로그램 카운터)에서 리턴어드레스(복귀주소)를 꺼낸다.