EDDI
Electronic Design
Development Institute

에디로봇아카데미
임베디드 마스터 Lv1 과정

제 5기

2023. 06. 09

어광선

CONTENTS

EDDI
Electronic Design
Development Institute

1. 이슈 정리
2. ARM32 Stack Frame
3. ARM32 vs. x86 Assembly 비교 분석
4. 제어문 (cpsr, eflags)

이슈 정리

```
root@gseo:/home/eddi/EmbeddedMasterLv1/5기 /GwangseonEo/lecture/4_week# qemu-arm-static -g 1234 -L /
usr/arm-linux-gnueabi ./a.out
bind: Address already in use
qemu: could not open gdbserver on 1234
```

이슈 1. qemu를 통해 remote port를 지정하고, gdb remote을 돌릴 시, address가 이미 사용되고 있다는 이슈가 발생한다. netstat –lntp 명령어를 통해 port 정보를 검색한다.

```
root@gseo:/home/eddi/EmbeddedMasterLv1/5기 /GwangseonEo/lecture/4_week# netstat -lntp
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name

tcp        0      0 127.0.0.53:53           0.0.0.0:*               LISTEN      104/systemd-resolve

tcp        0      0 127.0.0.1:43577         0.0.0.0:*               LISTEN      20279/node

tcp        0      0 127.0.0.1:39455         0.0.0.0:*               LISTEN      16011/node

tcp        0      0 0.0.0.0:1234            0.0.0.0:*               LISTEN      19224/qemu-arm-stat
```

Local Address Port 1234가 PID가 19224 확인되었고, kill -9 19224 명령어를 통해 19224를 종료시킨다. 다시 퀘뮤를 실행하면 remot가 정상적으로 동작한다.

ARM32 Stack Frame

EDDI
Electronic Design Development Institute

<ARM32 Calling Convention>
R0~R12 : 범용 레지스터, R11(Stack Frame Pointer)
R0 : 함수 리턴 값 저장 (EAX 같은 느낌)
R0 ~ R3 : 함수 호출 인자 전달
R13 ~ R15 : 특수 레지스터
R13(SP) 스택 포인터 : 스택 최상위를 가리킴
R14(LR) 링크 레지스터 : 서브루틴 후에 돌아갈 리턴 주소 저장
R15(PC) 프로그램 카운터 : 현재 fetch되고 있는 명령어의 주소 – 따라서 현재 실행되는 명령어의 다음다음 주소
CPSR(Computer Program Status Register) : 현재 프로그램 상태 레지스터

ARM32 Stack Frame

```
(gdb) disas
Dump of assembler code for function main:
=> 0x00010448 <+0>:     push    {r11, lr}
   0x0001044c <+4>:     add     r11, sp, #4
   0x00010450 <+8>:     sub     sp, sp, #8
   0x00010454 <+12>:    mov     r3, #3
```

```
r11            0x20f14             134932
r12            0x3ffa7000          1073377280
sp             0x408002d0          0x408002d0
lr             0x3fe3d958          1071896920
pc             0x10448             0x10448 <main>
cpsr           0x60000010          1610612752
```

명령어 수행 전

```
r11            0x20f14             134932
r12            0x3ffa7000          1073377280
sp             0x408002c8          0x408002c8
lr             0x3fe3d958          1071896920
pc             0x1044c             0x1044c <main+4>
cpsr           0x60000010          1610612752
```

명령어 수행 후

	02d0
lr	02cc
r11	02c8

sp

arm32 시스템에서는 sp가 이동하는 단위가 4byte인데 lr과 r11을 push, 즉 2번 push해서 8byte 이동하였다.

```
(gdb) x/gx 0x408002c8
0x408002c8:     0x3fe3d95800020f14
```

ARM32 Stack Frame

```
   0x00010448 <+0>:     push    {r11, lr}
=> 0x0001044c <+4>:     add     r11, sp, #4
   0x00010450 <+8>:     sub     sp, sp, #8
   0x00010454 <+12>:    mov     r3, #3
```

r11 = sp + 0x4 = 0x40802c8 + 0x4 = 0x408002cc

r11 : 0x20f14 → 0x408002cc
pc : 0x1044c → 0x10450

add 명령어에서 sp 값은 변하지 않는다.

	02d0
lr	02cc
r11	02c8
	02c4

sp

ARM32 Stack Frame

```
   0x00010448 <+0>:     push    {r11, lr}
   0x0001044c <+4>:     add     r11, sp, #4
=> 0x00010450 <+8>:     sub     sp, sp, #8
   0x00010454 <+12>:    mov     r3, #3
```

sp = sp – 0x8 = 0x408002c8 – 0x8 = 0x408002c0

sp : 0x408002c8 → 0x408002c0
pc : 0x10450 → 0x10454

들어갈 인자 공간을 만들어주기 위해 8byte 뺀다.

	02d0	
lr	02cc	
r11	02c8	sp
	02c4	
	02c0	sp
	02bc	

```
   0x00010448 <+0>:     push    {r11, lr}
   0x0001044c <+4>:     add     r11, sp, #4
   0x00010450 <+8>:     sub     sp, sp, #8
=> 0x00010454 <+12>:    mov     r3, #3
```

r3 = 0x10448 → 0x3

ARM32 Stack Frame

```
   0x00010454 <+12>:    mov     r3, #3
=> 0x00010458 <+16>:    str     r3, [r11, #-12]
   0x0001045c <+20>:    ldr     r0, [r11, #-12]
   0x00010460 <+24>:    bl      0x10420 <multiply_two>
```

*(r11-12)에 r3을 store한다.

	02d0
lr	02cc
r11	02c8
408002e8	02c4
00000003	02c0
	02bc

sp

```
   0x00010454 <+12>:    mov     r3, #3
   0x00010458 <+16>:    str     r3, [r11, #-12]
=> 0x0001045c <+20>:    ldr     r0, [r11, #-12]
   0x00010460 <+24>:    bl      0x10420 <multiply_two>
```

r0에 *(r11-12) 값을 가져와 넣는다.

r0 : 0x1 -> 0x3

```
(gdb) p/x $r0
$15 = 0x3
```

ARM32 Stack Frame

```
   0x0001045c <+20>:    ldr     r0, [r11, #-12]
=> 0x00010460 <+24>:    bl      0x10420 <multiply_two>
   0x00010464 <+28>:    str     r0, [r11, #-8]
   0x00010468 <+32>:    ldr     r1, [r11, #-8]
   0x0001046c <+36>:    ldr     r0, [pc, #16]   ; 0x10484 <main+60>
```

0x10420으로 분기 한다. multiply_two 함수 시작

	02d0
lr	02cc
r11	02c8
408002e8	02c4
00000003	02c0
	02bc

sp

```
=> 0x00010420 <+0>:     push    {r11}           ; (str r11, [sp, #-4]!)
   0x00010424 <+4>:     add     r11, sp, #0
   0x00010428 <+8>:     sub     sp, sp, #12
   0x0001042c <+12>:    str     r0, [r11, #-8]
```

r11을 push 한다.

r11	02c8
408002e8	02c4
00000003	02c0
408002cc	02bc

sp

ARM32 Stack Frame

```
   0x00010420 <+0>:     push    {r11}
=> 0x00010424 <+4>:     add     r11, sp, #0
   0x00010428 <+8>:     sub     sp, sp, #12
   0x0001042c <+12>:    str     r0, [r11, #-8]
```

r11 = sp + 0x0 = 0x408002bc + 0x0 = 0x408002bc

r11	02c8
408002e8	02c4
00000003	02c0
408002cc	02bc

sp

```
   0x00010420 <+0>:     push    {r11}
   0x00010424 <+4>:     add     r11, sp, #0
=> 0x00010428 <+8>:     sub     sp, sp, #12
   0x0001042c <+12>:    str     r0, [r11, #-8]
```

sp = sp – 12 = 0x408002bc – 0xc = 0x408002b0

408002cc	02bc
	02b8
	02b4
	02b0

sp

sp

ARM32 Stack Frame

```
=> 0x0001042c <+12>:    str     r0, [r11, #-8]
   0x00010430 <+16>:    ldr     r3, [r11, #-8]
   0x00010434 <+20>:    lsl     r3, r3, #1
   0x00010438 <+24>:    mov     r0, r3
```

*(r11 – 8) 에 r0을 store한다.
0x408002bc – 0x8 = 0x408002b4

```
   0x0001042c <+12>:    str     r0, [r11, #-8]
=> 0x00010430 <+16>:    ldr     r3, [r11, #-8]
   0x00010434 <+20>:    lsl     r3, r3, #1
   0x00010438 <+24>:    mov     r0, r3
```

r3 레지스터에 *(r11 – 8) 값을 Load 한다.
r3 = *(0x408002bc – 8)
r3 = 0x3

408002cc	02bc
	02b8
3	02b4
	02b0

sp

ARM32 Stack Frame

```
=> 0x00010434 <+20>:    lsl     r3, r3, #1
   0x00010438 <+24>:    mov     r0, r3
   0x0001043c <+28>:    add     sp, r11, #0
   0x00010440 <+32>:    pop     {r11}           ; (ldr r11, [sp], #4)
```

r3을 left로 한번 shift한 값을 r3에 load 한다.
r3 = 6

```
=> 0x00010438 <+24>:    mov     r0, r3
   0x0001043c <+28>:    add     sp, r11, #0
   0x00010440 <+32>:    pop     {r11}           ; (ldr r11, [sp], #4)
   0x00010444 <+36>:    bx      lr
```

r3을 r0에 넣는다.
r0 = 6

```
=> 0x00010438 <+24>:    mov     r0, r3
   0x0001043c <+28>:    add     sp, r11, #0
   0x00010440 <+32>:    pop     {r11}           ; (ldr r11, [sp], #4)
   0x00010444 <+36>:    bx      lr
```

add sp, r11, #0을 실행하면
sp = r11 + 0 = 0x408002bc + 0 = 0x408002bc

408002cc	02bc
	02b8
3	02b4
	02b0

sp

ARM32 Stack Frame

```
   0x00010438 <+24>:    mov     r0, r3
   0x0001043c <+28>:    add     sp, r11, #0
=> 0x00010440 <+32>:    pop     {r11}           ;
   0x00010444 <+36>:    bx      lr
```

pop r11이 실행되면 r11 값을 스택 메모리에서 사라진다. sp값은 상위주소로 4byte만큼 올라간다.

~~r11~~	02c8
408002e8	02c4
00000003	02c0
408002cc	02bc

sp

sp

```
   0x00010438 <+24>:    mov     r0, r3
   0x0001043c <+28>:    add     sp, r11, #0
   0x00010440 <+32>:    pop     {r11}
=> 0x00010444 <+36>:    bx      lr
```

lr 레지스터에는 0x10464가 저장되어 있으면 해당 주소로 jump 한다.

20f14	02c8
408002e8	02c4
00000003	02c0
408002cc	02bc

sp

ARM32 Stack Frame

```
=> 0x00010464 <+28>:    str     r0, [r11, #-8]
   0x00010468 <+32>:    ldr     r1, [r11, #-8]
   0x0001046c <+36>:    ldr     r0, [pc, #16]   ; 0x10484 <main+60>
   0x00010470 <+40>:    bl      0x10304 <printf@plt>
```

*(r11- 8) = r0 = 0x6

20f14	02c8
6	02c4
00000003	02c0
408002cc	02bc

sp

```
   0x00010464 <+28>:    str     r0, [r11, #-8]
=> 0x00010468 <+32>:    ldr     r1, [r11, #-8]
   0x0001046c <+36>:    ldr     r0, [pc, #16]   ; 0x10484 <main+60>
   0x00010470 <+40>:    bl      0x10304 <printf@plt>
```

r1 = *(r11 – 8) = 6

20f14	02c8
6	02c4
00000003	02c0
408002cc	02bc

sp

ARM32 Stack Frame

```
   0x00010464 <+28>:    str     r0, [r11, #-8]
   0x00010468 <+32>:    ldr     r1, [r11, #-8]
=> 0x0001046c <+36>:    ldr     r0, [pc, #16]   ; 0x10484 <main+60>
   0x00010470 <+40>:    bl      0x10304 <printf@plt>
```

*(pc + 16) 값을 r0에 저장
r0 = *(0x1046c + 16) = 0x10524

20f14	02c8
6	02c4
00000003	02c0
408002cc	02bc

sp

```
=> 0x00010474 <+44>:    mov     r3, #0
   0x00010478 <+48>:    mov     r0, r3
   0x0001047c <+52>:    sub     sp, r11, #4
   0x00010480 <+56>:    pop     {r11, pc}
   0x00010484 <+60>:    andeq   r0, r1, r4, lsr #10
```

r3에 0을 대입
그 밑에 r0에 0을 대입

ARM32 Stack Frame

```
   0x00010478 <+48>:    mov     r0, r3
=> 0x0001047c <+52>:    sub     sp, r11, #4
   0x00010480 <+56>:    pop     {r11, pc}
   0x00010484 <+60>:    andeq   r0, r1, r4, lsr #10
```

sp = r11 – 4 = 0x408002c8

20f14	02c8	sp
6	02c4	
00000003	02c0	sp
408002cc	02bc	

```
   0x0001047c <+52>:    sub     sp, r11, #4
=> 0x00010480 <+56>:    pop     {r11, pc}
   0x00010484 <+60>:    andeq   r0, r1, r4, lsr #10
```

push 명령어로 stack에 박아놓았던 return address를 다시 pop하여 r11과 pc에 넣어주는 역할을 한다.

마지막 andeq r0, r1, r4, lsr #10은
r8을 10만큼 오른쪽으로 shift한 값과 r1과의 & 연산이 r0와 같으면 분기이다.

ARM32 vs x86

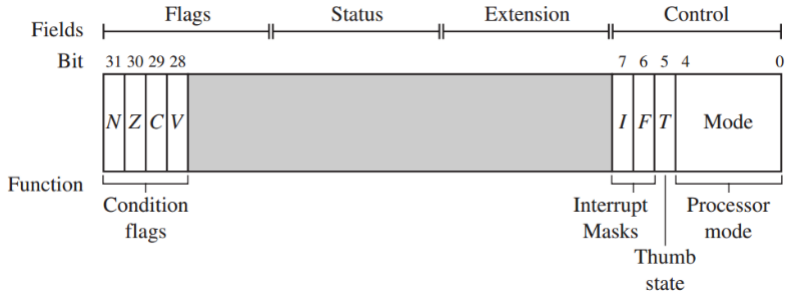
EDDI
Electronic Design Development Institute

x86은 return address를 stack memory에 저장하지만, ARM32는 Linker Register에 저장한다.

ARM은 RISC style 아키텍쳐이며 명령어들은 Regular size(32-bit for standard ARM)를 가진다.

제어문(if, for 사용)

CPSR(Current Program Status Register) : 현재 status를 저장하는 register



31 bit : N(Negative), Negative result from ALU
30 bit : Z(Zero), Zero result from ALU
29 bit : C(Carry), ALU operation caused Carry
28 bit : V(over flow), ALU operation overflowed
7 bit : I(interrupt), 1 : disable irq, 0 : enable irq
6 bit : F(fast interrtupt), 1 : disable fiq, 0 : enable fiq
5 bit : T(Thumb), 1 : thumb mode, 0 : arm mode
[4:0] bits : Mode

10000 : User	10011 : SVC(supervisor)
11111 : System	10111 : Abort
10001 : FIQ,	11011 : Undefined
10010 : IRQ,	

제어문(if, for 사용)

```
(gdb) l
9                       arr[i] = i;
10                      printf("arr[%d] = %d\r\n", i, arr[i]);
11              }
12      }
13
14      int main(void){
15              int a = 1;
16              int b = 2;
17
18              if (b>a) test();
```

초기 register 값
r11 = 0x20f04
sp = 0x408002c0
lr = 0x3fe3d958
cpsr = 0x60000010

제어문(if, for 사용)

```
(gdb) disas
Dump of assembler code for function main:
=> 0x0001057c <+0>:     push    {r11, lr}
   0x00010580 <+4>:     add     r11, sp, #4
   0x00010584 <+8>:     sub     sp, sp, #8
   0x00010588 <+12>:    mov     r3, #1
```

lr과 r11값을 push한다.

lr	02bc
r11	02b8

sp

r11	0x20f04	0x20f04
sp	0x408002c0	0x408002b8
lr	0x3fe3d958	0x3fe3d958
cpsr	0x60000010	0x60000010

제어문(if, for 사용)

```
(gdb) disas
Dump of assembler code for function main:
=> 0x0001057c <+0>:     push    {r11, lr}
   0x00010580 <+4>:     add     r11, sp, #4
   0x00010584 <+8>:     sub     sp, sp, #8
   0x00010588 <+12>:    mov     r3, #1
```

<+0> lr과 r11값을 push한다.

lr	02bc
r11	02b8

sp

r11	0x20f04	0x20f04
sp	0x408002c0	0x408002b8
lr	0x3fe3d958	0x3fe3d958
cpsr	0x60000010	0x60000010

제어문(if, for 사용)

```
   0x0001057c <+0>:     push    {r11, lr}
=> 0x00010580 <+4>:     add     r11, sp, #4
   0x00010584 <+8>:     sub     sp, sp, #8
   0x00010588 <+12>:    mov     r3, #1
```

<+4> r11 = sp + 4
= 0x408002b80 + 4 = 0x408002bc

lr	02bc
r11	02b8

sp

r11	0x20f04	0x408002bc
sp	0x408002b8	
lr	0x3fe3d958	
cpsr	0x60000010	

제어문(if, for 사용)

```
   0x0001057c <+0>:     push    {r11, lr}
   0x00010580 <+4>:     add     r11, sp, #4
=> 0x00010584 <+8>:     sub     sp, sp, #8
   0x00010588 <+12>:    mov     r3, #1
```

sp = sp- 8
= 0x408002b8 – 8 = 0x408002b0

r11	0x408002bc	
sp	0x408002b8	0x408002b0
lr	0x3fe3d958	
cpsr	0x60000010	

lr	02bc	
r11	02b8	sp
	02b4	
	02b0	sp

제어문(if, for 사용)

```
=> 0x00010588 <+12>:    mov     r3, #1
   0x0001058c <+16>:    str     r3, [r11, #-12]
   0x00010590 <+20>:    mov     r3, #2
   0x00010594 <+24>:    str     r3, [r11, #-8]
```

r3 = 0x1

lr	02bc
r11	02b8
	02b4
	02b0

sp

r11	0x408002bc	
sp	0x408002b0	
lr	0x3fe3d958	
cpsr	0x60000010	
r3		0x1

제어문(if, for 사용)

```
=> 0x0001058c <+16>:    str     r3, [r11, #-12]
   0x00010590 <+20>:    mov     r3, #2
   0x00010594 <+24>:    str     r3, [r11, #-8]
   0x00010598 <+28>:    ldr     r2, [r11, #-8]
```

r3값을 r11-12 위치에 워드만큼 저장
0x408002bc – 12 = 0x408002b0에 r3값 저장

lr	02bc
r11	02b8
	02b4
0x1	02b0

sp

r11	0x408002bc	
sp	0x408002b0	
lr	0x3fe3d958	
cpsr	0x60000010	
r3	0x1	

제어문(if, for 사용)

```
=> 0x00010590 <+20>:    mov     r3, #2
   0x00010594 <+24>:    str     r3, [r11, #-8]
   0x00010598 <+28>:    ldr     r2, [r11, #-8]
   0x0001059c <+32>:    ldr     r3, [r11, #-12]
   0x000105a0 <+36>:    cmp     r2, r3
```

r3에 0x2 저장

lr	02bc
r11	02b8
	02b4
0x1	02b0

sp

r11	0x408002bc	
sp	0x408002b0	
lr	0x3fe3d958	
cpsr	0x60000010	
r3	0x1	0x2

제어문(if, for 사용)

```
   0x00010590 <+20>:    mov     r3, #2
=> 0x00010594 <+24>:    str     r3, [r11, #-8]
   0x00010598 <+28>:    ldr     r2, [r11, #-8]
   0x0001059c <+32>:    ldr     r3, [r11, #-12]
   0x000105a0 <+36>:    cmp     r2, r3
```

r3을 r11 – 8 한 위치에 저장
0x408002b4에 0x2 저장

lr	02bc	
r11	02b8	
0x2	02b4	
0x1	02b0	sp

r11	0x408002bc	
sp	0x408002b0	
lr	0x3fe3d958	
cpsr	0x60000010	
r3	0x2	

제어문(if, for 사용)

```
   0x00010594 <+24>:    str     r3, [r11, #-8]
=> 0x00010598 <+28>:    ldr     r2, [r11, #-8]
   0x0001059c <+32>:    ldr     r3, [r11, #-12]
   0x000105a0 <+36>:    cmp     r2, r3
```

r2 = *(r11-8) = 0x1

lr	02bc
r11	02b8
0x2	02b4
0x1	02b0

sp

r11	0x408002bc	
sp	0x408002b0	
lr	0x3fe3d958	
cpsr	0x60000010	
r3	0x2	
r2		0x2

제어문(if, for 사용)

```
   0x00010594 <+24>:    str     r3, [r11, #-8]
   0x00010598 <+28>:    ldr     r2, [r11, #-8]
=> 0x0001059c <+32>:    ldr     r3, [r11, #-12]
   0x000105a0 <+36>:    cmp     r2, r3
```

r3에 *(r11 – 12) 값을 저장
r3 = *(0x408002b0) = 0x1

lr	02bc
r11	02b8
0x2	02b4
0x1	02b0

sp

r11	0x408002bc	
sp	0x408002b0	
lr	0x3fe3d958	
cpsr	0x60000010	
r3	0x2	0x1
r2	0x2	

제어문(if, for 사용)

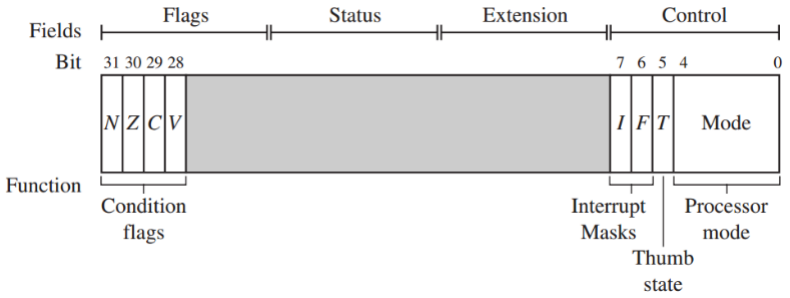
```
=> 0x000105a0 <+36>:    cmp     r2, r3
   0x000105a4 <+40>:    ble     0x105ac <main+48>
   0x000105a8 <+44>:    bl      0x104bc <test>
   0x000105ac <+48>:    mov     r3, #0
   0x000105b0 <+52>:    mov     r0, r3
```

r2와 r3 레지스터를 비교. cpsr에 flag에 저장

lr	02bc
r11	02b8
0x2	02b4
0x1	02b0

sp

r11	0x408002bc	
sp	0x408002b0	
lr	0x3fe3d958	
cpsr	0x60000010	0x20000010
r3	0x1	
r2	0x2	



제어문(if, for 사용)

```
   0x000105a0 <+36>:    cmp     r2, r3
=> 0x000105a4 <+40>:    ble     0x105ac <main+48>
   0x000105a8 <+44>:    bl      0x104bc <test>
   0x000105ac <+48>:    mov     r3, #0
```

다음 명령어로 넘어간다.

lr	02bc
r11	02b8
0x2	02b4
0x1	02b0

sp

r11	0x408002bc	
sp	0x408002b0	
lr	0x3fe3d958	
cpsr	0x20000010	0x20000010
r3	0x1	
r2	0x2	

제어문(if, for 사용)

```
   0x000105a0 <+36>:    cmp     r2, r3
=> 0x000105a4 <+40>:    ble     0x105ac <main+48>
   0x000105a8 <+44>:    bl      0x104bc <test>
   0x000105ac <+48>:    mov     r3, #0
```

r2가 r3보다 크므로 다음 명령어로 넘어간다.

lr	02bc
r11	02b8
0x2	02b4
0x1	02b0

sp

r11	0x408002bc	
sp	0x408002b0	
lr	0x3fe3d958	
cpsr	0x20000010	0x20000010
r3	0x1	
r2	0x2	

제어문(if, for 사용)

```
   0x000105a4 <+40>:    ble     0x105ac <main+48>
=> 0x000105a8 <+44>:    bl      0x104bc <test>
   0x000105ac <+48>:    mov     r3, #0
   0x000105b0 <+52>:    mov     r0, r3
   0x000105b4 <+56>:    sub     sp, r11, #4
   0x000105b8 <+60>:    pop     {r11, pc}
```

test함수를 실행한다. 새로운 스택 프레임을 생성한다.

lr	02bc
r11	02b8
0x2	02b4
0x1	02b0

sp

r11	0x408002bc	
sp	0x408002b0	
lr	0x10530	
cpsr	0x60000010	
r3	0x0	
r2	0x0	

제어문(if, for 사용)

```
=> 0x000105ac <+48>:    mov     r3, #0
   0x000105b0 <+52>:    mov     r0, r3
   0x000105b4 <+56>:    sub     sp, r11, #4
   0x000105b8 <+60>:    pop     {r11, pc}
```

r3에 0x0을 저장한다.
그 다음 명령어에서 r0에 0x0을 저장한다.

lr	02bc
r11	02b8
0x2	02b4
0x1	02b0

sp

r11	0x408002bc	
sp	0x408002b0	
lr	0x10530	
cpsr	0x60000010	
r3	0x0	
r2	0x0	

제어문(if, for 사용)

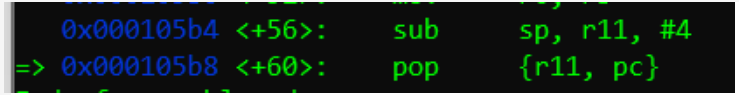
```
   0x000105b0 <+52>:    mov     r0, r3
=> 0x000105b4 <+56>:    sub     sp, r11, #4
   0x000105b8 <+60>:    pop     {r11, pc}
```

sp에 r11 – 4 값을 넣는다.
0x408002b8

lr	02bc	
r11	02b8	sp
0x2	02b4	
0x1	02b0	sp

r11	0x408002bc	
sp	0x408002b0	0x408002b8
lr	0x10530	
cpsr	0x60000010	
r3	0x0	
r2	0x0	

제어문(if, for 사용)



r11, pc값을 pop한다.
stack에 박아놓았던 return address를 다시 pop하여 r11과 pc에 넣어주는 역할.

lr	02bc
r11	02b8
	02b4
	02b0

sp

r11	0x408002bc	
sp	0x408002b8	
lr	0x10530	
cpsr	0x60000010	
r3	0x0	
r2	0x0	