
# C Programming – 4

임베디드스쿨2기

Lv2과정

2021. 04. 14

박태인

# 1. 모든 것은 포인터다

## 1) 포인터 통일

해당 내용을 통해서 결국 C언어의 모든 것이
포인터라는 것을 확인 할 수 있을 것이다.
이중 포인터, 삼중 포인터, 배열, 다중 배열,
포인터 배열, 함수 포인터를 별개로 볼 필요가 없다.
다만 이것을 진행하기 위해서는 몇개지 개념이 필요 하다.

**1 . 메모리 계층 구조**
**2. 스택(Stack)은 아래로 자란다.**
**3. GP Register에 대한 명확한 개념과 각각의 용도**

## 2) 디버깅 명령어

Info registers : 실제 HW 레지스터 정보를 확인 할 수 있고, **여기서는 펌웨어 제어와 관련된 레지스터 정보는 보여 주지 않는다.**
우리가 이 내용을 진행하면서 주의를 둘 부분은 아래와 같다.

rsp, rbp, rip, rax, rcx 정도에 해당한다.

- rsp : 현재 스택의 최상위
- rbp : 현재 스택의 기준점
- rip : 다음에 실행 할 instruction의 주소 값을 가르킴
- rax : 무조건적으로 함수의 리턴값이 저장되며 연산용으로도 활용 가능
- rcx : 보편적으로 for 루프의 카운트에 활용이 되며 연산용으로도 활용 가능

- si : 어셈블리 명령어 기준으로 한 줄씩 실행한다.
- p/x : 16진수로 특정 결과를 출력한다.
- x : 메모리의 내용을 살펴본다.

# 1. 디버깅 과정(test_func.c) - (1)

```c
#include <stdio.h>

int my_func(int num)
{
        return num >> 1;
}

int main(void)
{
        int num = 3, res;

        res = my_func(num);

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

        return 0;
}
```

```
res = 1
```

```
04$ gcc -g -o test_func test_func.c
04$ gdb test_func
```

```
[Inferior 1 (proces
(gdb) b main
Breakpoint 1 at 0x5
(gdb) r
Starting program: /

Breakpoint 1, main
9       {
(gdb) disas
```

0000 0010 = 3
>> 0000 0001 = 1

오른쪽 쉬프트 연산하여
위와
같이 **num 3**의 값이
**Res = 1 의 값으로**
출력 된다.
지금부터 이 소스 코드를
이용해 gdb 를 분석한다!

```
Dump of assembler code for function main:
=> 0x000055555555515b <+0>:     endbr64
   0x000055555555515f <+4>:     push   %rbp
   0x0000555555555160 <+5>:     mov    %rsp,%rbp
   0x0000555555555163 <+8>:     sub    $0x10,%rsp
   0x0000555555555167 <+12>:    movl   $0x3,-0x8(%rbp)
   0x000055555555516e <+19>:    mov    -0x8(%rbp),%eax
   0x0000555555555171 <+22>:    mov    %eax,%edi
   0x0000555555555173 <+24>:    callq  0x555555555149 <my_func>
   0x0000555555555178 <+29>:    mov    %eax,-0x4(%rbp)
   0x000055555555517b <+32>:    mov    -0x4(%rbp),%eax
   0x000055555555517e <+35>:    mov    %eax,%esi
   0x0000555555555180 <+37>:    lea    0xe7d(%rip),%rdi        # 0x555555556004
   0x0000555555555187 <+44>:    mov    $0x0,%eax
   0x000055555555518c <+49>:    callq  0x555555555050 <printf@plt>
   0x0000555555555191 <+54>:    mov    $0x0,%eax
   0x0000555555555196 <+59>:    leaveq
   0x0000555555555197 <+60>:    retq
```

# 1. 디버깅 과정(test_func.c) - (2)

```
Dump of assembler code for function main:
=> 0x000055555555515b <+0>:     endbr64
   0x000055555555515f <+4>:     push   %rbp
   0x0000555555555160 <+5>:     mov    %rsp,%rbp
   0x0000555555555163 <+8>:     sub    $0x10,%rsp
   0x0000555555555167 <+12>:    movl   $0x3,-0x8(%rbp)
   0x000055555555516e <+19>:    mov    -0x8(%rbp),%eax
   0x0000555555555171 <+22>:    mov    %eax,%edi
   0x0000555555555173 <+24>:    callq  0x555555555149 <my_func>
   0x0000555555555178 <+29>:    mov    %eax,-0x4(%rbp)
   0x000055555555517b <+32>:    mov    -0x4(%rbp),%eax
   0x000055555555517e <+35>:    mov    %eax,%esi
   0x0000555555555180 <+37>:    lea    0xe7d(%rip),%rdi        # 0x555555556004
   0x0000555555555187 <+44>:    mov    $0x0,%eax
   0x000055555555518c <+49>:    callq  0x555555555050 <printf@plt>
   0x0000555555555191 <+54>:    mov    $0x0,%eax
   0x0000555555555196 <+59>:    leaveq
   0x0000555555555197 <+60>:    retq
```

=> 표시는 아직 실행되지 않았고
다음에 실행 하게 될 것이라는 표시!

자... 이제 진짜.. 디버깅을 시작해 봅시다!

먼저 push rbp로 이동을 해본다. (si 실행)

```
Dump of assembler code for function main:
   0x000055555555515b <+0>:     endbr64
=> 0x000055555555515f <+4>:     push   %rbp
```

이후 rsp 값 기록한다(si 실행) : 0x7fffffffdf98 (다음에 새로 시작하면 값이 바뀔수도 있으니 주의한다.)

```
End of assembler dump.
(gdb) x $rsp
0x7fffffffdf98: 0xf7deb0b3
(gdb) x $rbp
0x0:    Cannot access memory at address 0x0
(gdb) p/x $rsp
$1 = 0x7fffffffdf98
```

마다 8

실제로 처음에 이 값은 시작할때
바이트씩 바뀌더라..

부가 설명)
현재 최상위 스택인 rsp가
0x7fffffffdf98 이라는 가상의 주소라는 것이고,
현재 스택의 기준점 값인 rbp는 아직
주소 값이 정해지지 않았고,
값은 '0x0' 이라는 것이다!

# 1. 디버깅 과정(test_func.c) - (3)

```
Dump of assembler code for function main:
   0x000055555555515b <+0>:     endbr64
   0x000055555555515f <+4>:     push   %rbp
=> 0x0000555555555160 <+5>:     mov    %rsp,%rbp
   0x0000555555555163 <+8>:     sub    $0x10,%rsp
```

자.. 이제 si를 한번 더 실행해서 push %rbp명령어를 실행 했습니다.
Push 명령어는 **현재 스택의 최상위 메모리(rsp)**에 값을 저장하는 명령어 입니다.
즉, **현재 스택의 최상위 메모리(rsp)**에 **rbp값을 저장하라**는 의미 겠죠.
그런데, 앞 ppt에서 보듯이 rbp의 값은 0x0 이었으므로 아래와 같이 구성 되겠습니다.

| 0x0 (rbp) | 0x0x7fffffffdf90 (rsp) |
|---|---|

```
   0x000055555555515b <+0>:     endbr64
   0x000055555555515f <+4>:     push   %rbp
   0x0000555555555160 <+5>:     mov    %rsp,%rbp
=> 0x0000555555555163 <+8>:     sub    $0x10,%rsp
```

```
(gdb) p/x $rsp
$2 = 0x7fffffffdf90
(gdb) x $rsp
0x7fffffffdf90: 0x00000000
(gdb) x $rbp
0x7fffffffdf90: 0x00000000
(gdb) p/x $rbp
$3 = 0x7fffffffdf90
```

ㄴ 사라진 경계선

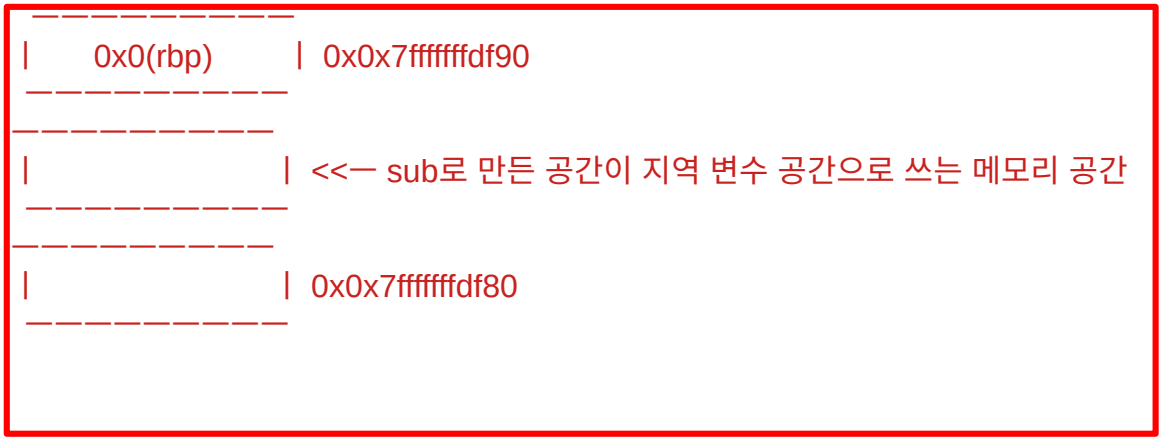
자 이제 한번 더 si를 실행해서 mov %rsp, %rbp 를 시행 해봅시다.
Mov 명령어는 내용을 복사하는 것 입니다.
즉, mov rsp rbp는 rbp에 rsp 값을 복사합니다. (rsp 값 → rbp 값)
ㄴ일반적인 A = B 꼴에서 B 값이 A로 들어가는 것이 아닌 반대 방향으로 생각해야 하는 것에 주의 하자!

그러면 어떻게 되겠나요?
ㄴ 결국, rbp에 rsp 값을 넣어 버림 으로써 **rsp, rbp 주소 값이 서로 같아지면서 스택의 경계선이 사라집니다!**
→ 이것은 **새로운 스택을 생성 할 준비를 하는 과정** 입니다.

# 1. 디버깅 과정(test_func.c) - (4)

```
   0x000055555555515b <+0>:     endbr64
   0x000055555555515f <+4>:     push   %rbp
   0x0000555555555160 <+5>:     mov    %rsp,%rbp
   0x0000555555555163 <+8>:     sub    $0x10,%rsp
=> 0x0000555555555167 <+12>:    movl   $0x3,-0x8(%rbp)
   0x000055555555516e <+19>:    mov    -0x8(%rbp),%eax
```

자 이번에는 sub 명령어를 실행 했습니다.
Sub 명령어는 뺄셈 명령 입니다.
Sub 0x10, rsp는 현재 rsp 에서 **16바이트를 빼겠다는 의미** 입니다. (0x10 → 0001 0000 : 2^4 = 16 바이트,
**가상 주소 공간에서는 바이트 단위로 움직입니다**!)
그림으로 나타내어 보면 아래와 같습니다.




ㄴ 이런 구조 때문에 스택은 아래로 자란다고 한 것이다!!

```
(gdb) x $rbp
0x7fffffffdf90: 0x00000000
(gdb) x $rsp
0x7fffffffdf80: 0xffffe080
```

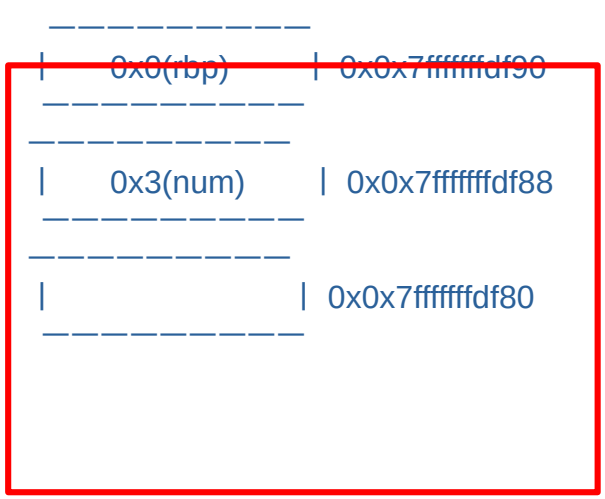
: 현재 스택의 최상위 rsp 80
현재 스택의 기준점 rbp 90

# 1. 디버깅 과정(test_func.c) - (5)

```
   0x0000555555555160 <+5>:     mov    %rsp,%rbp
   0x0000555555555163 <+8>:     sub    $0x10,%rsp
   0x0000555555555167 <+12>:    movl   $0x3,-0x8(%rbp)
=> 0x000055555555516e <+19>:    mov    -0x8(%rbp),%eax
   0x0000555555555171 <+22>:    mov    %eax,%edi
```

자, 이번에는 movl $0x3, -0x8(%rbp)이 보인다.
L 이 들어가면 4바이트 처리를 하겠다는 의미 이며
Q 가 들어가면 8바이트 처리를 하겠다는 뜻이 된다.
여튼, 위 명령어의 의미는 **rbp를 기준으로 8바이트 뺀 자리에 0x3을 복사한다는 의미** 이다.

그림으로 나타내 보면 아래와 같다.



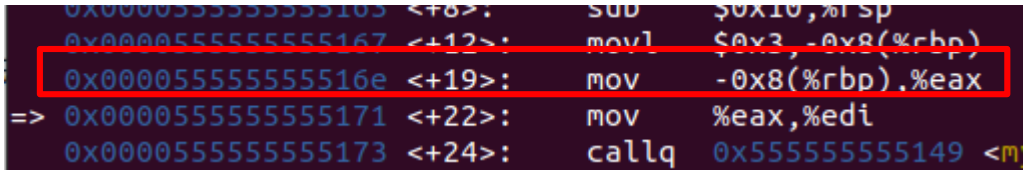

```
(gdb) x $rsp
0x7fffffffdf80: 0xffffe080
(gdb) x $rbp
0x7fffffffdf90: 0x00000000
(gdb) x $rbp-8
0x7fffffffdf88: 0x00000003
(gdb) x $rsp+8
0x7fffffffdf88: 0x00000003
```

ㄴ 그림 처럼 rsp+8 위치 혹은 rbp-8 위치에 3의 값이 들어 간 것 을 확인 할 수 있다!

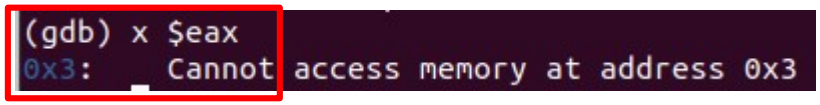
```c
#include <stdio.h>

int my_func(int num)
{
        return num >> 1;
}

int main(void)
{
        int num = 3, res;

        res = my_func(num);

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

        return 0;
}
```

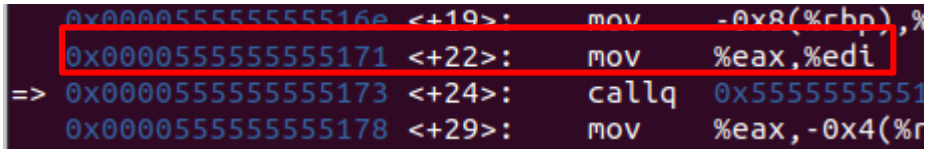
ㄴ 원소스의 num에 3을 넣는 과정

# 1. 디버깅 과정(test_func.c) - (6)

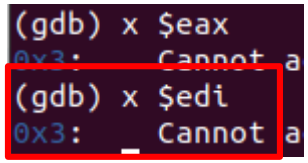



자, 이번에도 mov 명령어이다. 앞서 설명한 바와 같이 mov명령어는 내용을 복사하는 것이다.
그렇다면 mov -0x8(%rbp), %eax 는
**eax레지스터(4바이트 레지스터)에 rbp기준 -8 바이트 값, 즉 0x3(num)을 eax에 넣겠다**는 것이다.
이 것은 연산에 사용되며, 연산 이후 ax 레지스터를 확인하면 변경된 값을 볼 수 있다.




그리고 다음의 mov eax, edi는 그냥 복사이다.
따라서, 아래와 같이 edi에 3이 들어 가는 것이다.



# 1. 디버깅 과정(test_func.c) - (7)

```
   0x0000555555555163 <+8>:     sub    $0x10,%rsp
   0x0000555555555167 <+12>:    movl   $0x3,-0x8(%rbp)
   0x000055555555516e <+19>:    mov    -0x8(%rbp),%eax
=> 0x0000555555555171 <+22>:    mov    %eax,%edi
   0x0000555555555173 <+24>:    callq  0x555555555149 <my_func>
   0x0000555555555178 <+29>:    mov    %eax,-0x4(%rbp)
   0x000055555555517b <+32>:    mov    -0x4(%rbp),%eax
```

Callq 0x555555555149 은 매우 중요한 연산이다.
**Call 은 기본적으로 push + jmp로 구성**되어 있다.
**함수 호출이 끝난 이후에 실행해야 할 어셈블리 명령어의 주소값을 psuh로 저장**한다.
이후 함수 호출을 수행하기 위해 jmp를 수행한다.

결국 아래와 같은 메모리를 가지게 된다.

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdf90 |
| 0x3(num) | 0x0x7fffffffdf88 |
| | 0x0x7fffffffdf80 |
| 0x0000555555555178(복귀주소) | 0x0x7fffffffdf78 |

```
(gdb) x $rsp
0x7fffffffdf78: 0x55555178
```

```
(gdb) x $rsp+4
0x7fffffffdf7c: 0x00005555
```

ㄴ 스택의 최상위 값인 rsp가 78이 되고 안에
복귀 주소인 0x0000555555555178 이 저장된 것을 볼 수 있다.
(복귀 주소가 너무 길어서 rsp+4 명령어 사용해서 뷰!)

# 1. 디버깅 과정(test_func.c) - (8)

```
Dump of assembler code for function my_func:
   0x0000555555555149 <+0>:     endbr64
=> 0x000055555555514d <+4>:     push   %rbp
   0x000055555555514e <+5>:     mov    %rsp,%rbp
   0x0000555555555151 <+8>:     mov    %edi,-0x4(%rbp)
   0x0000555555555154 <+11>:    mov    -0x4(%rbp),%eax
   0x0000555555555157 <+14>:    sar    %eax
   0x0000555555555159 <+16>:    pop    %rbp
   0x000055555555515a <+17>:    retq
```

위는 함수로 이동하게 된 모습이다.
여기서 **push rbp를 진행하게 되면 rsp 값이 추가로 8바이트 빠지므로** 메모리는 아래와 같은 구성을 하게 될 것이다.
또한, **기존의 스택의 기준점인 rbp를 저장**하게 될 것이다.

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdf90 |
| 0x3(num) | 0x0x7fffffffdf88 |
| | 0x0x7fffffffdf80 |
| 0x0000555555555178(복귀주소) | 0x0x7fffffffdf78 |
| 0x0x7fffffffdf90(이전 함수의 rbp) | 0x0x7fffffffdf70 |

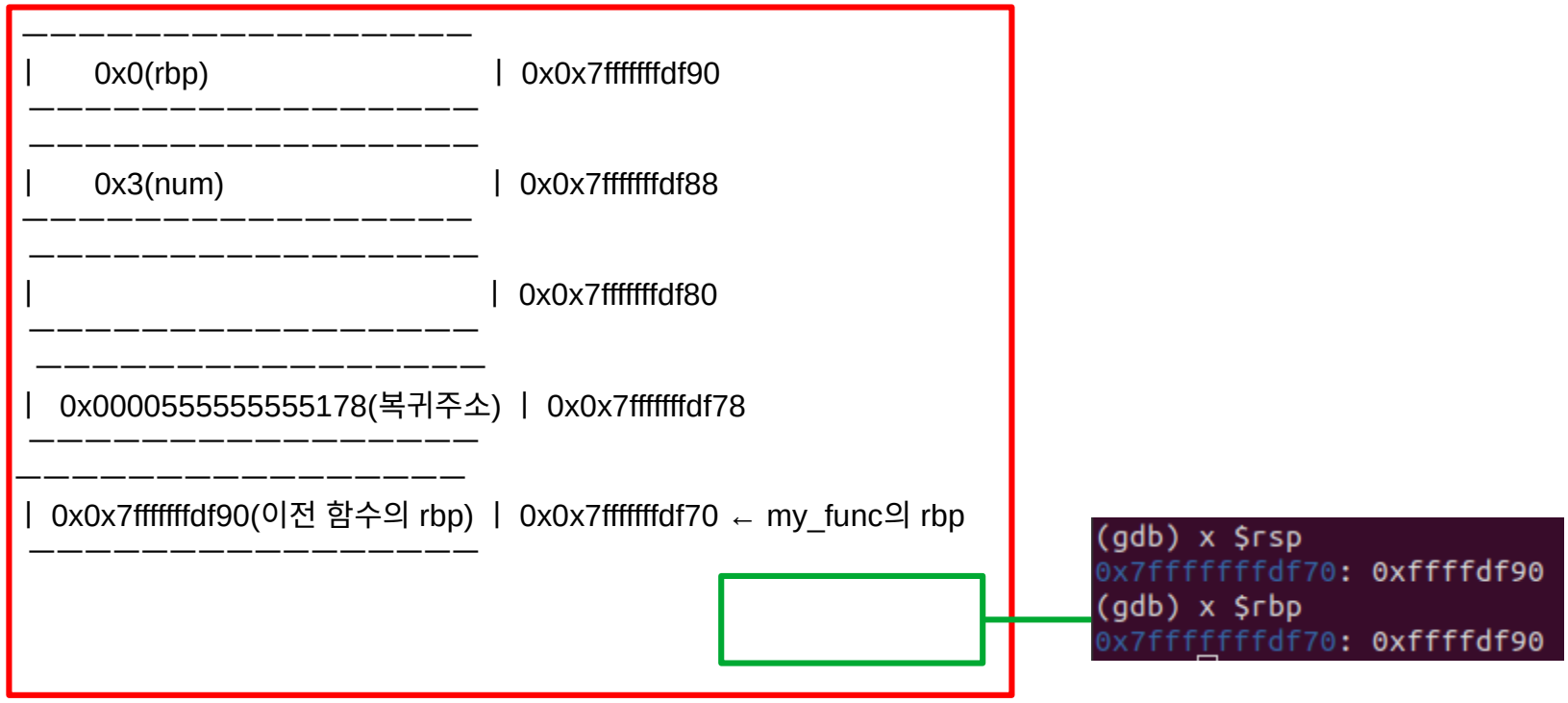
```
(gdb) x $rsp
0x7fffffffdf70: 0xffffdf90
```

ㄴ 스택의 최상위 값인 rsp가 70이 되고 안에
**이전 함수의 rbp가 저장**된다(아직 rbp는 바뀐적이 없으므로)

# 1. 디버깅 과정(test_func.c) - (9)

```
   0x000055555555514d <+4>:     push   %rbp
   0x000055555555514e <+5>:     mov    %rsp,%rbp
=> 0x0000555555555151 <+8>:     mov    %edi,-0x4(%rbp)
   0x0000555555555154 <+11>:    mov    -0x4(%rbp),%eax
   0x0000555555555157 <+14>:    sar    %eax
```

이후 다시 **mov rsp, rbp를 하면 새로운 rbp가 생성**된다. ( rsp → rbp)
**이것이 my_func에 해당하는 새로운 스택의 기준점**이 된다.
**결국 함수를 호출 할 때 마다 스택을 새롭게 생성한다는 것**이 되며,
우리가 **포인터를 사용하는 이유**는
**바로 이 경계선을 넘어 자유롭게 왔다 갔다 할 수 있기 때문**이다.




```
(gdb) x $rsp
0x7fffffffdf70: 0xffffdf90
(gdb) x $rbp
0x7fffffffdf70: 0xffffdf90
```

# 1. 디버깅 과정(test_func.c) - (10)

```
Dump of assembler code for function my_func:
   0x0000555555555149 <+0>:     endbr64
   0x000055555555514d <+4>:     push   %rbp
   0x000055555555514e <+5>:     mov    %rsp,%rbp
   0x0000555555555151 <+8>:     mov    %edi,-0x4(%rbp)
=> 0x0000555555555154 <+11>:    mov    -0x4(%rbp),%eax
   0x0000555555555157 <+14>:    sar    %eax
   0x0000555555555159 <+16>:    pop    %rbp
   0x000055555555515a <+17>:    retq
```

이후 **edi 값을 rbp-4 위치에 배치** 한다.
**edi에는 eax에서 옮겨온 인자값 3이 배치**가 된다.

```
#include <stdio.h>

int my_func(int num)
{
        return num >> 1;
}

int main(void)
{
        int num = 3, res;

        res = my_func(num);

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

        return 0;
}
```

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdf90 |
| 0x3(num) | 0x0x7fffffffdf88 |
| | 0x0x7fffffffdf80 |
| 0x0000555555555178(복귀주소) | 0x0x7fffffffdf78 |
| 0x0x7fffffffdf90(이전 함수의 rbp) | 0x0x7fffffffdf70 ← my_func의 rbp |
| 0x3 (num아니고 my_func의 num) | 0x0x7fffffffdf6c |

```
(gdb) p/x $edi
$2 = 0x3
(gdb) x $rbp-4
0x7fffffffdf6c: 0x00000003
```

ㄴ rbp 기준으로 -4 위치에 0x3 의 값 적용

# 1. 디버깅 과정(test_func.c) - (11)

```
Dump of assembler code for function my_func:
   0x0000555555555149 <+0>:     endbr64
   0x000055555555514d <+4>:     push   %rbp
   0x000055555555514e <+5>:     mov    %rsp,%rbp
   0x0000555555555151 <+8>:     mov    %edi,-0x4(%rbp)
=> 0x0000555555555154 <+11>:    mov    -0x4(%rbp),%eax
   0x0000555555555157 <+14>:    sar    %eax
   0x0000555555555159 <+16>:    pop    %rbp
   0x000055555555515a <+17>:    retq
```

**Rbp-4 위치에 있는 값을 eax에 배치** 한다.
(여기에 rbp-4에는 **my_func의 num** 값이 들어간다.)

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdf90 |
| 0x3(num) | 0x0x7fffffffdf88 |
| | 0x0x7fffffffdf80 |
| 0x0000555555555178(복귀주소) | 0x0x7fffffffdf78 |
| 0x0x7fffffffdf90(이전 함수의 rbp) | 0x0x7fffffffdf70 ← my_func의 rbp |
| 0x3 (num아니고 my_func의 num) | 0x0x7fffffffdf6c |

```
(gdb) x $eax
0x3:    Cannot
```

# 1. 디버깅 과정(test_func.c) - (12)

```
   0x0000555555555149 <+0>:     endbr64
   0x000055555555514d <+4>:     push   %rbp
   0x000055555555514e <+5>:     mov    %rsp,%rbp
   0x0000555555555151 <+8>:     mov    %edi,-0x4(%rbp)
   0x0000555555555154 <+11>:    mov    -0x4(%rbp),%eax
   0x0000555555555157 <+14>:    sar    %eax
=> 0x0000555555555159 <+16>:    pop    %rbp
   0x000055555555515a <+17>:    retq
```

자, 이번에는 sar 명령어 이다. **Sar 은 Shift Arithmetic Right의 약자**로 오른쪽 쉬프트이다.
Sar %eax의 경우 **뒤쪽에 비트가 표현이 안되어 있으므로 기본적으로 1이다 ( >> 1을 의미)**

eax는 3이므로 결과는 1이 될 것이다.
ax는 함수의 리턴값을 가진다고 했다.
실제로 이 값은 my_func이 리턴하는 값에 해당한다.

```
(gdb) x $eax
0x1:    Canno
```

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdf90 |
| 0x3(num) | 0x0x7fffffffdf88 |
| | 0x0x7fffffffdf80 |
| 0x0000555555555178(복귀주소) | 0x0x7fffffffdf78 |
| 0x0x7fffffffdf90(이전 함수의 rbp) | 0x0x7fffffffdf70 ← my_func의 rbp |
| 0x3 (num아니고 my_func의 num) | 0x0x7fffffffdf6c |

# 1. 디버깅 과정(test_func.c) - (13)

```
   0x000055555555514e <+5>:     mov    %rsp,%rbp
   0x0000555555555151 <+8>:     mov    %edi,-0x4(%rbp)
   0x0000555555555154 <+11>:    mov    -0x4(%rbp),%eax
   0x0000555555555157 <+14>:    sar    %eax
   0x0000555555555159 <+16>:    pop    %rbp
=> 0x000055555555515a <+17>:    retq
```

다음으로 pop %rbp를 수행하는데 **pop은 현재 rsp 에서 값을 빼서**
**뒤쪽에 배치된 메모리나 레지스터에 값을 넘겨 준다.(여기서는 현재 rsp 값을 rbp에 주는 것 이겠죠)**

결국 값을 빼내기 때문에 rsp값은 70에서 78로 증가하게 되고
내부에 있는 값은 rbp로 들어가므로 이전의 rbp가 복원된다.
(**결국 스택을 복원하는 작업의 일부에 해당함**)

이 시점의 메모리는 다음과 같다.

```
(gdb) x $rsp
0x7fffffffdf78:
(gdb) x $rbp
0x7fffffffdf90:
```

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdf90 ← rbp (rsp안의 값이 이전 함수의 rbp) |
| 0x3(num) | 0x0x7fffffffdf88 |
| | 0x0x7fffffffdf80 |
| 0x0000555555555178(복귀주소) | 0x0x7fffffffdf78 ← rsp (밑에 있던 rsp가 올라옴) |
| 0x0x7fffffffdf90(이전 함수의 rbp) | 0x0x7fffffffdf70 |
| 0x3 (num아니고 my_func의 num) | 0x0x7fffffffdf6c |

# 1. 디버깅 과정(test_func.c) - (14)

```
   0x000055555555514e <+5>:     mov    %rsp,%rbp
   0x0000555555555151 <+8>:     mov    %edi,-0x4(%rbp)
   0x0000555555555154 <+11>:    mov    -0x4(%rbp),%eax
   0x0000555555555157 <+14>:    sar    %eax
   0x0000555555555159 <+16>:    pop    %rbp
=> 0x000055555555515a <+17>:    retq
```

다음으로 retq를 진행하는데 retq는 아래와 같은 의미를 가진다.
Pop rip 에 해당하는 연산이다.
즉, rsp에서 값을 빼서 rip에 배치하는 것이다.

결국 값을 빼내기 때문에 rsp값은 78에서 80로 증가하게 되고 내부에 있는 값은 rip으로 들어간다..

이 시점의 메모리는 다음과 같다.

```
(gdb) x $rsp
0x7fffffffdf80:
(gdb) x $rip
0x555555555178
```

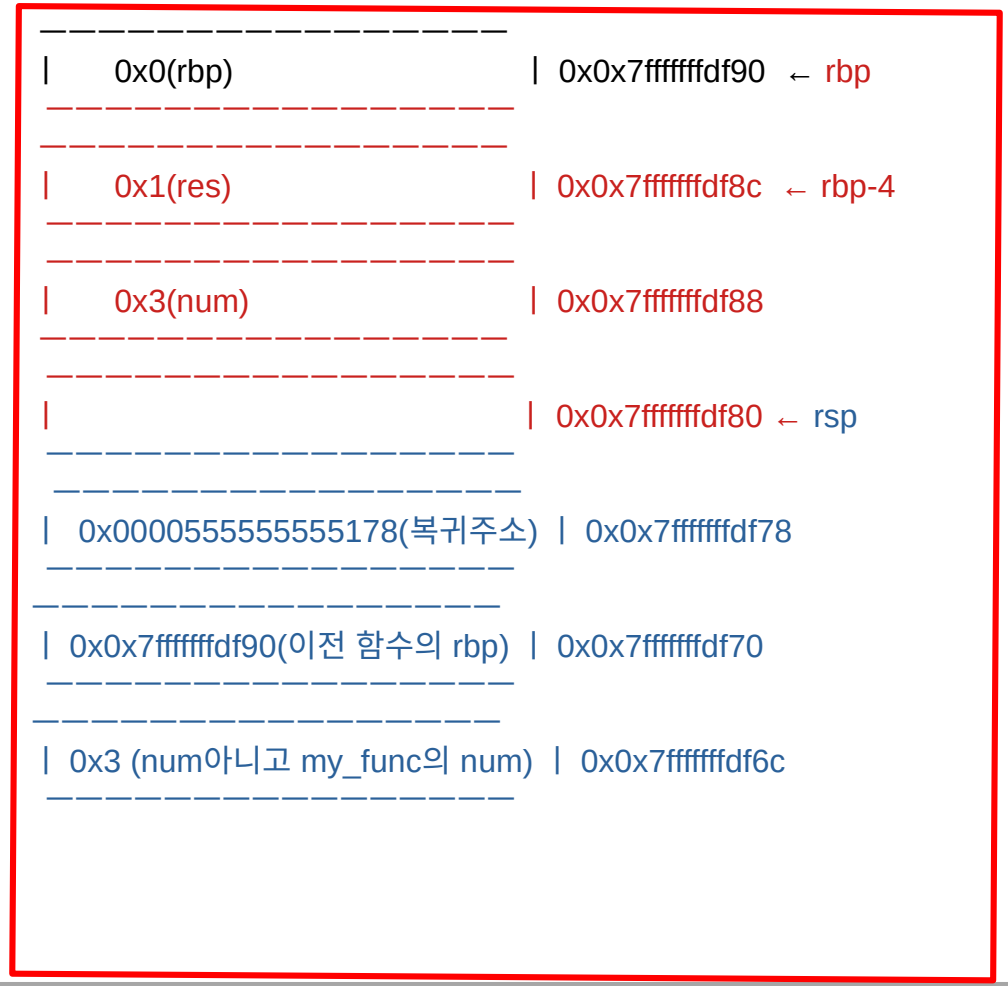
| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdf90 ← rbp |
| 0x3(num) | 0x0x7fffffffdf88 |
| | 0x0x7fffffffdf80 ← rsp (밑에 있던 rsp가 올라옴) |
| 0x0000555555555178(복귀주소) | 0x0x7fffffffdf78 |
| 0x0x7fffffffdf90(이전 함수의 rbp) | 0x0x7fffffffdf70 |
| 0x3 (num아니고 my_func의 num) | 0x0x7fffffffdf6c |

# 1. 디버깅 과정(test_func.c) - (15)

```
   0x0000555555555167 <+12>:    movl   $0x3,-0x8(%rbp)
   0x000055555555516e <+19>:    mov    -0x8(%rbp),%eax
   0x0000555555555171 <+22>:    mov    %eax,%edi
   0x0000555555555173 <+24>:    callq  0x555555555149 <my_func>
=> 0x0000555555555178 <+29>:    mov    %eax,-0x4(%rbp)
   0x000055555555517b <+32>:    mov    -0x4(%rbp),%eax
   0x000055555555517e <+35>:    mov    %eax,%esi
   0x0000555555555180 <+37>:    lea    0xe7d(%rip),%rdi        #
```

복귀 후에 rbp-4 자리에 리턴값인 eax 레지스터를 배치한다.
메모리 구조는 오른쪽과 같다.

```
(gdb) x $rbp-4
0x7fffffffdf8c: 0x00000001
```



# 1. 디버깅 과정(test_func.c) - (16)

```
   0x000055555555517b <+32>:    mov    -0x4(%rbp),%eax
=> 0x000055555555517e <+35>:    mov    %eax,%esi
   0x0000555555555180 <+37>:    lea    0xe7d(%rip),%rdi
   0x0000555555555187 <+44>:    mov    $0x0,%eax
   0x000055555555518c <+49>:    callq  0x555555555050 <pri
   0x0000555555555191 <+54>:    mov    $0x0,%eax
```

다음으로 rbp-4에 값을 eax에 배치 한다.
즉, 0x1(res)를 eax에 넣는거죠.

```
(gdb) x $rbp-4
0x7fffffffdf8c: 0x00000001
(gdb) x $eax
0x1:    Cannot access memory
```

# 1. 디버깅 과정(test_func.c) - (17)

```
   0x0000555555555178 <+29>:    mov    %eax,-0x4(%rbp)
   0x000055555555517b <+32>:    mov    -0x4(%rbp),%eax
=> 0x000055555555517e <+35>:    mov    %eax,%esi
   0x0000555555555180 <+37>:    lea    0xe7d(%rip),%rdi
   0x0000555555555187 <+44>:    mov    $0x0,%eax
```

다음으로 eax 값은 esi로 복사.
즉, 0x1값이 eax, esi에 각각 배치 된다.

```
(gdb) x $eax
0x1:    Cannot
(gdb) x $esi
0x1:    Cannot
```

```
   0x000055555555517e <+35>:    mov    %eax,%esi
   0x0000555555555180 <+37>:    lea    0xe7d(%rip),%rdi        # 0x555555556004
=> 0x0000555555555187 <+44>:    mov    $0x0,%eax
```

Lea 명령어의 경우 **배열**인데, 0xe7d(%rip)의 값이 %rdi에 배치 된다고 라고만 이해하자.

```
(gdb) x $rdi
0x555555556004: 0x20736572
```

```
   0x0000555555555180 <+37>:    lea    0xe7d(%rip),%r
=> 0x0000555555555187 <+44>:    mov    $0x0,%eax
   0x000055555555518c <+49>:    callq  0x555555555050
   0x0000555555555191 <+54>:    mov    $0x0,%eax
```

0x0 값을 eax에 복사 배치 한다.

```
(gdb) x $eax
0x0:    Cannot
```

# 1. 디버깅 과정(test_func.c) - (18)

```
   0x0000555555555171 <+22>:    mov    %eax,%edi
   0x0000555555555173 <+24>:    callq  0x555555555149 <my_func>
   0x0000555555555178 <+29>:    mov    %eax,-0x4(%rbp)
   0x000055555555517b <+32>:    mov    -0x4(%rbp),%eax
   0x000055555555517e <+35>:    mov    %eax,%esi
   0x0000555555555180 <+37>:    lea    0xe7d(%rip),%rdi        # 0x
=> 0x0000555555555187 <+44>:    mov    $0x0,%eax
   0x000055555555518c <+49>:    callq  0x555555555050 <printf@plt>
   0x0000555555555191 <+54>:    mov    $0x0,%eax
   0x0000555555555196 <+59>:    leaveq
```

Callq 0x555555555050 의 연산이다.(앞서 만들었던 함수 호출처럼)
**Call 은 기본적으로 push + jmp로 구성**되어 있다.
**함수 호출이 끝난 이후에 실행해야 할 어셈블리 명령어의 주소값을 psuh로 저장**한다.
이후 함수 호출을 수행하기 위해 jmp를 수행한다.

결국 아래와 같은 메모리를 가지게 된다.

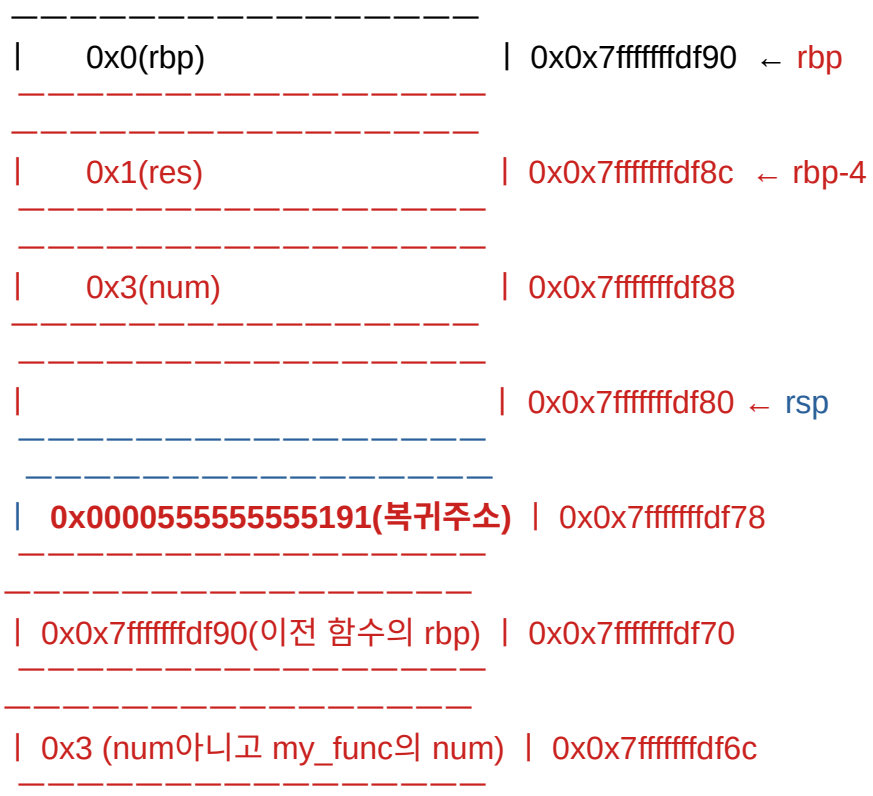
| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdf90 ← rbp |
| 0x1(res) | 0x0x7fffffffdf8c ← rbp-4 |
| 0x3(num) | 0x0x7fffffffdf88 |
| | 0x0x7fffffffdf80 ← rsp |
| **0x0000555555555191(복귀주소)** | 0x0x7fffffffdf78 |
| 0x0x7fffffffdf90(이전 함수의 rbp) | 0x0x7fffffffdf70 |
| 0x3 (num아니고 my_func의 num) | 0x0x7fffffffdf6c |

부가 설명 : push로 주소값 저장하게 되므로 현재 rsp에서 8바이트 더해진 주소에 복귀주소가 저장된다. (앞 있던 복귀 주소가 엎어쓰기 된다!!)

# 1. 디버깅 과정(test_func.c) - (19)

```
=> 0x0000555555555050 <+0>:     endbr64
   0x0000555555555054 <+4>:     bnd jmpq *0x2f75(%rip)        # 0x555555557fd0 <printf@got.plt>
   0x000055555555505b <+11>:    nopl   0x0(%rax,%rax,1)
```

Pritnf 함수의 어셈블리 내용이다.
0x2f75의 포인터 배열에 값을 rip에 배치를 하고
널 값을 리턴 후 나오는 구조 인듯하다.
(si를 하면 미궁에 빠지므로 ni로 넘길 것)



# 1. 디버깅 과정(test_func.c) - (20)

```
   0x000055555555518c <+49>:    callq  0x555555555050 <printf@plt>
   0x0000555555555191 <+54>:    mov    $0x0,%eax
=> 0x0000555555555196 <+59>:    leaveq
   0x0000555555555197 <+60>:    retq
```

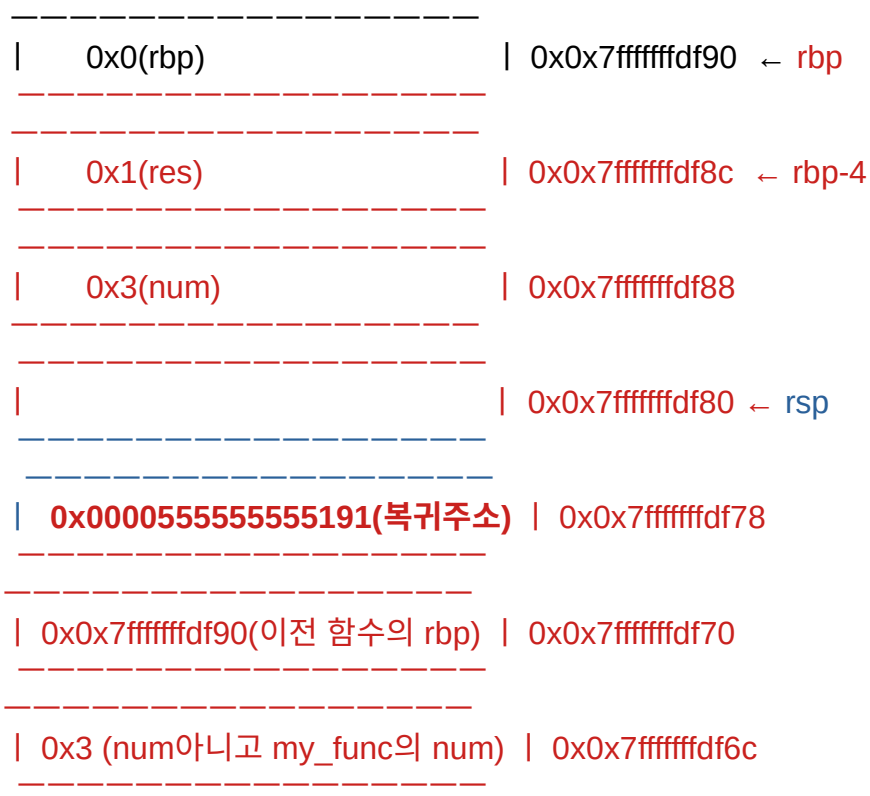
**mov로 0x0값을 eax에 배치 하고,**

```
(gdb) x $eax
0x0:    Cannot
```

**Leaveq 는 스택해제 명령어** 이다.

```
(gdb) x $rsp
0x7fffffffdf98:
```

ㄴrsp가 안드로메다로 갔쥬



# 2. if문에 대한 기계어 분석 (test_if.c) - (1)

```c
#include <stdio.h>

int main(void)
{
        int num1 = 1, num2 = 2;


        if(num1 > num2)
        {
                printf("num1(%d)가 num2(%d)보다 큽니다.\n", num1, num2);
        }
        else
        {
                printf("num2(%d)가 num1(%d)보다 큽니다.\n", num2, num1);
        }

        return 0;
}
```

If 문 예제 작성

If 문의 어셈블리 동작을 분석하기 위한 코드.

Num1,2의 변수를 비교해서 값을 출력한다.

# 2. if문에 대한 기계어 분석 (test_if.c) - (2)

```
   0x0000555555555149 <+0>:     endbr64
   0x000055555555514d <+4>:     push   %rbp
=> 0x000055555555514e <+5>:     mov    %rsp,%rbp
   0x0000555555555151 <+8>:     sub    $0x10,%rsp
```

자.. 이제 si를 실행해서 push %rbp명령어를 실행 했습니다.
Push 명령어는 **현재 스택의 최상위 메모리(rsp)**에 값을 저장하는 명령어 입니다.
즉, **현재 스택의 최상위 메모리(rsp)**에 **rbp값을 저장하라**는 의미 겠죠.
그런데, 앞 ppt에서 보듯이 rbp의 값은 0x0 이었으므로 아래와 같이 구성 되겠습니다.

| 0x0 (rbp) | 0x0x7fffffffdfa0 (rsp) |
|---|---|

```
(gdb) x $rsp
0x7fffffffdfa0:
(gdb) x $rbp
```

```
   0x000055555555514e <+5>:     mov    %rsp,%rbp
=> 0x0000555555555151 <+8>:     sub    $0x10,%rsp
```

```
(gdb) x $rsp
0x7fffffffdfa0:
(gdb) x $rbp
0x7fffffffdfa0:
```

자 이제 한번 더 si를 실행해서 mov %rsp, %rbp 를 시행 해봅시다.
Mov 명령어는 내용을 복사하는 것 입니다.
즉, mov rsp rbp는 rbp에 rsp 값을 복사합니다. (rsp 값 → rbp 값)
ㄴ일반적인 A = B 꼴에서 B 값이 A로 들어가는 것이 아닌 반대 방향으로 생각해야 하는 것에 주의 하자! ㄴ 사라진 경계선

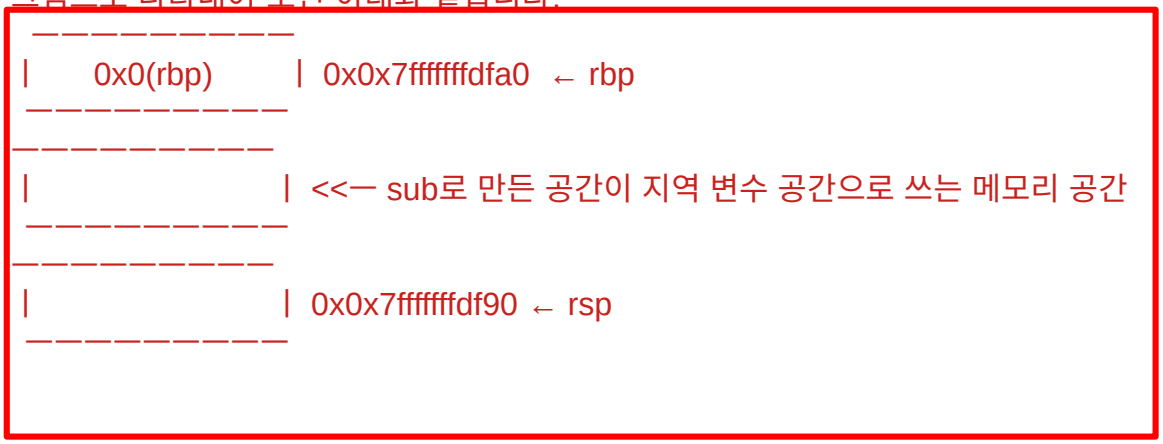
그러면 어떻게 되겠나요?
ㄴ 결국, rbp에 rsp 값을 넣어 버림 으로써 **rsp, rbp 주소 값이 서로 같아지면서 스택의 경계선이 사라집니다!**
→ 이것은 **새로운 스택을 생성 할 준비를 하는 과정** 입니다.

# 2. if문에 대한 기계어 분석 (test_if.c) - (3)

```
   0x000055555555514d <+4>:     push   %rbp
   0x000055555555514e <+5>:     mov    %rsp,%rbp
   0x0000555555555151 <+8>:     sub    $0x10,%rsp
=> 0x0000555555555155 <+12>:    movl   $0x1,-0x8(%rbp)
   0x000055555555515c <+19>:    movl   $0x2,-0x4(%rbp)
```

자 이번에는 sub 명령어를 실행 했습니다.
Sub 명령어는 뺄셈 명령 입니다.
Sub 0x10, rsp는 현재 rsp 에서 **16바이트를 빼겠다는 의미** 입니다. (0x10 → 0001 0000 : 2^4 = **16 바이트,**
**가상 주소 공간에서는 바이트 단위로**

**움직입니다**!)
그림으로 나타내어 보면 아래와 같습니다.




ㄴ 이런 구조 때문에 스택은 아래로 자란다고 한 것이다!!

```
(gdb) x $rbp
0x7fffffffdfa0: 0x00000000
(gdb) x $rsp
0x7fffffffdf90: 0xffffe090
```

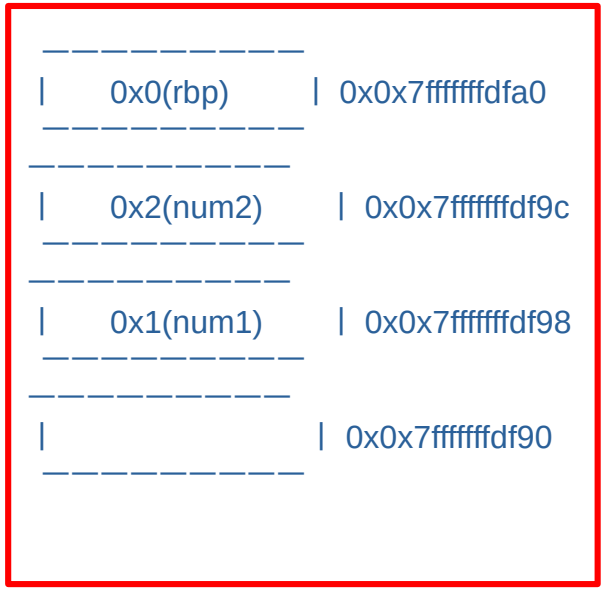
: 현재 스택의 최상위 rsp 90
현재 스택의 기준점 rbp a0

# 2. if문에 대한 기계어 분석 (test_if.c) - (4)

```
   0x0000555555555155 <+12>:    movl   $0x1,-0x8(%rbp)
   0x000055555555515c <+19>:    movl   $0x2,-0x4(%rbp)
=> 0x0000555555555163 <+26>:    mov    -0x8(%rbp),%eax
   0x0000555555555166 <+29>:    cmp    -0x4(%rbp),%eax
```

자, 이번에는 movl $0x1, -0x8(%rbp)이 보인다.
L 이 들어가면 4바이트 처리를 하겠다는 의미 이며
Q 가 들어가면 8바이트 처리를 하겠다는 뜻이 된다.
여튼, 위 명령어의 의미는 **rbp를 기준으로 8바이트 뺀 자리에 0x2을 복사한다는 의미** 이다.
그리고 movl $0x2, -0x4(%rbp), **rbp를 기준으로 4바이트 뺀 자리에 0x1을 복사한다는 의미**.

그림으로 나타내 보면 아래와 같다.




```
(gdb) x $rbp
0x7fffffffdfa0: 0x00000000
(gdb) x $rsp
0x7fffffffdf90: 0xffffe090
(gdb) x $rbp-8
0x7fffffffdf98: 0x00000001
(gdb) x $rbp-4
0x7fffffffdf9c: 0x00000002
```

ㄴ 그림 처럼 rbp-8에 0x1 배치, rbp-4에 0x2 배치

```c
#include <stdio.h>

int main(void)
{
        int num1 = 1, num2 = 2;


        if(num1 > num2)
        {
                printf("num1(%d)가 num
        }
        else
        {
                printf("num2(%d)가 num
        }

        return 0;
}
```

ㄴ 원소스의 num1에 1을
num2에 2를 넣는 과정.

# 2. if문에 대한 기계어 분석 (test_if.c) - (5)

```
   0x000055555555515c <+19>:    movl   $0x2,-0x4(%rbp)
   0x0000555555555163 <+26>:    mov    -0x8(%rbp),%eax
=> 0x0000555555555166 <+29>:    cmp    -0x4(%rbp),%eax
   0x0000555555555169 <+32>:    jle    0x555555555186 <main+61>
   0x000055555555516b <+34>:    mov    -0x4(%rbp),%edx
   0x000055555555516e <+37>:    mov    -0x8(%rbp),%eax
```

자, 이번에는 cmp -0x4(%rbp) **[source]** , %eax **[dest]**
는 rbp 기준 -4바이트 위치(9c)의 값(num2)과 eax값(이전에 num1의 1을 넣어둠)을 비교 합니다.

그림으로 나타내 보면 아래와 같다.

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdfa0 |
| 0x2(num2) | 0x0x7fffffffdf9c |
| 0x1(num1) | 0x0x7fffffffdf98 |
| | 0x0x7fffffffdf90 |

```c
#include <stdio.h>

int main(void)
{
        int num1 = 1, num2 = 2;


        if(num1 > num2)
        {
                printf("num1(%d)가 num
        }
        else
        {
                printf("num2(%d)가 num
        }

        return 0;
}
```

ㄴ 원소스의 num1 > num2 비교 후
else로 점프

cmp의 경우 보통 jmp 명령어와 같이 사용하게 되는데,
**Jle = jump less or equal** 로 작거나 같을 때 점프한다는 의미
이다.**(dest가 기준이다)**

num2(source) , num1(dest)

dest인 num1(1)은 souce인 num2(2)보다 작으므로 jle 명령어가
발동하여 **0x55555555186** 주소로 점프 합니다.

# 2. if문에 대한 기계어 분석 (test_if.c) - (6)

```
   0x0000555555555186 <+61>:    mov    -0x8(%rbp),%edx
   0x0000555555555189 <+64>:    mov    -0x4(%rbp),%eax
   0x000055555555518c <+67>:    mov    %eax,%esi
=> 0x000055555555518e <+69>:    lea    0xe9b(%rip),%rdi
```

0x000055555555186 으로 점프 되었고, mov 를 통해
-0x8(%rbp)의 값인 0x1을 edx에
-0x4(%rbp)의 값인 0x2를 eax에
Eax 값을 esi에 배치 합니다.

```
(gdb) p/x $edx
$4 = 0x1
(gdb) p/x $eax
$5 = 0x2
(gdb) p/x $esi
$6 = 0x2
```

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdfa0 |
| 0x2(num2) | 0x0x7fffffffdf9c |
| 0x1(num1) | 0x0x7fffffffdf98 |
| | 0x0x7fffffffdf90 |

```
   0x000055555555518c <+67>:    mov    %eax,%esi
=> 0x000055555555518e <+69>:    lea    0xe9b(%rip),%rdi        # 0x555555556030
   0x0000555555555195 <+76>:    mov    $0x0,%eax
```

ㄴ 다음으로 lea는 배열 명령어 이며 0xe9b(%rip)값이 %rdi에 배치 되었다고 보면 된다.

```
(gdb) x $rdi
0x555555556030: 0x326d756e
```

```
   0x000055555555518c <+67>:    mov    %eax,%esi
=> 0x000055555555518e <+69>:    lea    0xe9b(%rip),%rdi        # 0x555555556030
   0x0000555555555195 <+76>:    mov    $0x0,%eax
```

ㄴ 다음으로 0의 값을 eax에 배치 한다.

```
(gdb) x $eax
0x0:    Cannot
```

# 2. if문에 대한 기계어 분석 (test_if.c) - (7)

```
   0x0000555555555195 <+76>:    mov    $0x0,%eax
=> 0x000055555555519a <+81>:    callq  0x555555555050 <printf@plt>
   0x000055555555519f <+86>:    mov    $0x0,%eax
   0x00005555555551a4 <+91>:    leaveq
   0x00005555555551a5 <+92>:    retq
```

Printf 함수는 ni로 넘긴다.

Callq 0x555555555050 은 매우 중요한 연산이다.
**Call 은 기본적으로 push + jmp로 구성**되어 있다.
**함수 호출이 끝난 이후에 실행해야 할 어셈블리 명령어의 주소값을 psuh로 저장**한다.
이후 함수 호출을 수행하기 위해 jmp를 수행한다.

결국 아래와 같은 메모리를 가지게 된다.

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdfa0 |
| 0x2(num2) | 0x0x7fffffffdf9c |
| 0x1(num1) | 0x0x7fffffffdf98 |
| | 0x0x7fffffffdf90 |
| 0x000055555555519f(복귀주소) | 0x0x7fffffffdf88 |

```
(gdb) x $rsp
0x7fffffffdf88: 0x5555519f
(gdb) x $rsp+4
0x7fffffffdf8c: 0x00005555
```

ㄴ 스택의 최상위 값인 rsp가 88이 되고 안에
복귀 주소인 0x000055555555519f 이 저장된 것을 볼 수 있다.
(복귀 주소가 너무 길어서 rsp+4 명령어 사용해서 뷰!)

# 2. if문에 대한 기계어 분석 (test_if.c) - (8)

```
   0x0000555555555195 <+76>:    mov    $0x0,%eax
=> 0x000055555555519a <+81>:    callq  0x555555555050 <printf@plt>
   0x000055555555519f <+86>:    mov    $0x0,%eax
   0x00005555555551a4 <+91>:    leaveq
   0x00005555555551a5 <+92>:    retq
```

Mov 로 0x0을 eax 에 배치 시키고

```
(gdb) x $eax
0x0:    Canno
```

**Leaveq 는 스택해제 명령어** 이다.

```
(gdb) x $rsp
0x7fffffffdfa8: 0xf7deb0b3
```

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdfa0 |
| 0x2(num2) | 0x0x7fffffffdf9c |
| 0x1(num1) | 0x0x7fffffffdf98 |
| | 0x0x7fffffffdf90 |
| 0x000055555555519f(복귀주소) | 0x0x7fffffffdf88 |

# 3. for문에 대한 기계어 분석 (test_for.c) - (1)

```c
#include <stdio.h>

int main(void)
{
        int i;

        for(i=0; i<10; i++)
        {
                printf("%d ",i);
        }
        printf("\n");

        return 0;
}
```

for 문 예제 작성

for 문의 어셈블리 동작을 분석하기 위한 코드.

for문을 통해 I 값을 출력 한다. ( 0 ~ 9 )

```
0 1 2 3 4 5 6 7 8 9
```

# 3. if문에 대한 기계어 분석 (test_for.c) - (2)

```
   0x0000555555555169 <+0>:     endbr64
=> 0x000055555555516d <+4>:     push   %rbp
   0x000055555555516e <+5>:     mov    %rsp,%rbp
   0x0000555555555171 <+8>:     sub    $0x10,%rsp
   0x0000555555555175 <+12>:    movl   $0x0,-0x4(%rbp)
```

자.. 이제 si를 실행해서 push %rbp명령어를 실행 했습니다.
Push 명령어는 **현재 스택의 최상위 메모리(rsp)**에 값을 저장하는 명령어 입니다.
즉, **현재 스택의 최상위 메모리(rsp)**에 **rbp값을 저장하라**는 의미 겠죠.
그런데, 앞 ppt에서 보듯이 rbp의 값은 0x0 이었으므로 아래와 같이 구성 되겠습니다.

| 0x0 (rbp) | 0x0x7fffffffdf**90** (rsp) |
|---|---|

```
(gdb) x $rsp
0x7fffffffdf90:
(gdb) x $rbp
0x0:    Cannot
```

```
=> 0x000055555555516e <+5>:     mov    %rsp,%rbp
   0x0000555555555171 <+8>:     sub    $0x10,%rsp
   0x0000555555555175 <+12>:    movl   $0x0,-0x4(%rbp)
```

```
(gdb) x $rsp
0x7fffffffdf90:
(gdb) x $rbp
0x7fffffffdf90:
```

자 이제 한번 더 si를 실행해서 mov %rsp, %rbp 를 시행 해봅시다.
Mov 명령어는 내용을 복사하는 것 입니다.
즉, mov rsp rbp는 rbp에 rsp 값을 복사합니다. (rsp 값 → rbp 값)
ㄴ일반적인 A = B 꼴에서 B 값이 A로 들어가는 것이 아닌 반대 방향으로 생각해야 하는 것에 주의 하자! ㄴ 사라진 경계선

그러면 어떻게 되겠나요?
ㄴ 결국, rbp에 rsp 값을 넣어 버림 으로써 **rsp, rbp 주소 값이 서로 같아지면서 스택의 경계선이 사라집니다!**
→ 이것은 **새로운 스택을 생성 할 준비를 하는 과정** 입니다.

# 3. for문에 대한 기계어 분석 (test_for.c) - (3)

```
   0x000055555555514d <+4>:     push   %rbp
   0x000055555555514e <+5>:     mov    %rsp,%rbp
   0x0000555555555151 <+8>:     sub    $0x10,%rsp
=> 0x0000555555555155 <+12>:    movl   $0x1,-0x8(%rbp)
   0x000055555555515c <+19>:    movl   $0x2,-0x4(%rbp)
```

자 이번에는 sub 명령어를 실행 했습니다.
Sub 명령어는 뺄셈 명령 입니다.
Sub 0x10, rsp는 현재 rsp 에서 **16바이트를 빼겠다는 의미** 입니다. (0x10 → 0001 0000 : 2^4 = **16 바이트,**
**가상 주소 공간에서는 바이트 단위로 움직입니다**!)

그림으로 나타내어 보면 아래와 같습니다.

```
 ──────────
|    0x0(rbp)    | 0x0x7fffffffdf90  ← rbp
 ──────────
──────────
|                | <<─ sub로 만든 공간이 지역 변수 공간으로 쓰는 메모리 공간
 ──────────
──────────
|                | 0x0x7fffffffdf80 ← rsp
 ──────────
```

ㄴ 이런 구조 때문에 스택은 아래로 자란다고 한 것이다!!

```
(gdb) x $rbp
0x7fffffffdf90:
(gdb) x $rsp
0x7fffffffdf80:
```

: 현재 스택의 최상위 rsp 80
현재 스택의 기준점 rbp 90

# 3. for문에 대한 기계어 분석 (test_for.c) - (4)

```
=> 0x0000555555555171 <+8>:     sub    $0x10,%rsp
   0x0000555555555175 <+12>:    movl   $0x0,-0x4(%rbp)
   0x000055555555517c <+19>:    jmp    0x555555555198 <main+47>
```

자, 이번에는 movl $0x1, -0x4(%rbp)이 보인다.
L 이 들어가면 4바이트 처리를 하겠다는 의미 이며
Q 가 들어가면 8바이트 처리를 하겠다는 뜻이 된다.
여튼, 위 명령어의 의미는 **rbp를 기준으로 4바이트 뺀 자리에 0을 복사한다는 의미** 이다

그림으로 나타내 보면 아래와 같다.

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdf90 |
| 0x0(i) | 0x0x7fffffffdf8c |
| | 0x0x7fffffffdf80 |

```
#include <stdio.h>

int main(void)
{
        int i;

        for(i=0; i<10; i++)
        {
                printf("%d ",i);
        }
        printf("\n");

        return 0;
}
```

ㄴ 원소스의 i를 초기화 하고 배치

```
(gdb) x $rbp
0x7fffffffdf90: 0x00000000
(gdb) x $rsp
0x7fffffffdf80: 0xffffe080
(gdb) x $rbp-4
0x7fffffffdf8c: 0x00000000
```

ㄴ 그림 처럼 rbp-4에 0x0 배치

# 3. for문에 대한 기계어 분석 (test_for.c) - (5)

```
0x000055555555517c <+19>:    jmp    0x555555555198 <main+47>
0x000055555555517e <+21>:    mov    -0x4(%rbp),%eax
0x0000555555555181 <+24>:    mov    %eax,%esi
0x0000555555555183 <+26>:    lea    0xe7a(%rip),%rdi        # 0x
0x000055555555518a <+33>:    mov    $0x0,%eax
0x000055555555518f <+38>:    callq  0x555555555070 <printf@plt>
0x0000555555555194 <+43>:    addl   $0x1,-0x4(%rbp)
=> 0x0000555555555198 <+47>:    cmpl   $0x9,-0x4(%rbp)
0x000055555555519c <+51>:    jle    0x55555555517e <main+21>
0x000055555555519e <+53>:    mov    $0xa,%edi
0x00005555555551a3 <+58>:    callq  0x555555555060 <putchar@plt>
```

ㄴ 0x0x7fffffffdf8c(rbp-4) 의 값을 9 와 비교.

그림으로 나타내 보면 아래와 같다.

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdf90 |
| 0x0(i) | 0x0x7fffffffdf8c |
| | 0x0x7fffffffdf80 |

```
0x000055555555517e <+21>:    mov    -0x4(%rbp),%eax
0x0000555555555181 <+24>:    mov    %eax,%esi
0x0000555555555183 <+26>:    lea    0xe7a(%rip),%rdi        # 0x
0x000055555555518a <+33>:    mov    $0x0,%eax
0x000055555555518f <+38>:    callq  0x555555555070 <printf@plt>
0x0000555555555194 <+43>:    addl   $0x1,-0x4(%rbp)
0x0000555555555198 <+47>:    cmpl   $0x9,-0x4(%rbp)
=> 0x000055555555519c <+51>:    jle    0x55555555517e <main+21>
```

ㄴ jle , less & equal 의 의미로 작거나 같으면 jump
0은 비교대상인 9보다 작으므로 jump

ㄴ 그림 처럼 rbp-4에 0x0 배치

# 3. for문에 대한 기계어 분석 (test_for.c) - (6)

```
=> 0x000055555555517e <+21>:    mov    -0x4(%rbp),%eax
   0x0000555555555181 <+24>:    mov    %eax,%esi
   0x0000555555555183 <+26>:    lea    0xe7a(%rip),%rdi        # 0x555555556004
   0x000055555555518a <+33>:    mov    $0x0,%eax
   0x000055555555518f <+38>:    callq  0x555555555070 <printf@plt>
   0x0000555555555194 <+43>:    addl   $0x1,-0x4(%rbp)
   0x0000555555555198 <+47>:    cmpl   $0x9,-0x4(%rbp)
   0x000055555555519c <+51>:    jle    0x55555555517e <main+21>
```

- mov -0x4(%rbp), %eax → 0x0(i)를 eax에 배치.
- mov %eax, %esi → eax 값을 esi에 배치
- lea 0xe7a(%rip), %rdi 배열의 값이 rdi에 배치
- mov 0x0, %eax → eax 값 0으로 초기화.
- callq → push와 jump의 합성 동작으로, 함수 호출이 완료 된 후 복귀 해야 할 주소를 rsp 다음 8바이트에 저장

```
(gdb) x $eax
0x0:    Cannot
(gdb) x $esi
0x0:    Cannot
```

그림으로 나타내 보면 아래와 같다.

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdf90 |
| 0x0(i) | 0x0x7fffffffdf8c |
| | 0x0x7fffffffdf80 |
| 0x00005555555194(복귀주소) | 0x0x7fffffffdf88 |

```c
#include <stdio.h>

int main(void)
{
        int i;

        for(i=0; i<10; i++)
        {
                printf("%d ",i);
        }
        printf("\n");

        return 0;
}
```

ㄴ pritnf 동작 전 값 배치 및 동작

# 3. for문에 대한 기계어 분석 (test_for.c) - (7)

```
=> 0x000055555555517e <+21>:    mov    -0x4(%rbp),%eax
   0x0000555555555181 <+24>:    mov    %eax,%esi
   0x0000555555555183 <+26>:    lea    0xe7a(%rip),%rdi        # 0x555555556004
   0x000055555555518a <+33>:    mov    $0x0,%eax
   0x000055555555518f <+38>:    callq  0x555555555070 <printf@plt>
   0x0000555555555194 <+43>:    addl   $0x1,-0x4(%rbp)
   0x0000555555555198 <+47>:    cmpl   $0x9,-0x4(%rbp)
   0x000055555555519c <+51>:    jle    0x55555555517e <main+21>
```

- addl $0x1, -0x4(%rbp)는 C언어로 하면 I = I(rbp-4) + 1 의 이미이다.
  ㄴ 1의 값을 rbp-4 의 값에 더한다라는 의미.
- cmpl 앞 서 1 더한 값을 9와 비교
- jle , 작거나 같으면 점프 이므로 1은 9보다 작으므로, 0x...17e주소로 점프하게 된다.

그림으로 나타내 보면 아래와 같다.

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdf90 |
| 0x0(i) | 0x0x7fffffffdf8c |
| | 0x0x7fffffffdf80 |
| 0x00005555555194(복귀주소) | 0x0x7fffffffdf88 |

```
   0x000055555555517c <+19>:    jmp    0x555555555198 <main+47>
   0x000055555555517e <+21>:    mov    -0x4(%rbp),%eax
   0x0000555555555181 <+24>:    mov    %eax,%esi
   0x0000555555555183 <+26>:    lea    0xe7a(%rip),%rdi        # 0x
   0x000055555555518a <+33>:    mov    $0x0,%eax
   0x000055555555518f <+38>:    callq  0x555555555070 <printf@plt>
=> 0x0000555555555194 <+43>:    addl   $0x1,-0x4(%rbp)
   0x0000555555555198 <+47>:    cmpl   $0x9,-0x4(%rbp)
   0x000055555555519c <+51>:    jle    0x55555555517e <main+21>
   0x000055555555519e <+53>:    mov    $0xa,%edi
```

ㄴ 이처럼 I 값이 9보다 작으면 숫자를 프린트 하고 I 값을 +1 하며,
이 값이 9보다 작다면 계속 반복하게 된다.

# 3. for문에 대한 기계어 분석 (test_for.c) - (8)

```
0x0000555555555198 <+47>:    cmpl   $0x9,-0x4(%rbp)
0x000055555555519c <+51>:    jle    0x55555555517e <main+21>
0x000055555555519e <+53>:    mov    $0xa,%edi
0x00005555555551a3 <+58>:    callq  0x555555555060 <putchar@plt>
0x00005555555551a8 <+63>:    mov    $0x0,%eax
0x00005555555551ad <+68>:    leaveq
0x00005555555551ae <+69>:    retq
```

- mov $0xa, 값을 %edi에 복사
- callq rsp+8에 복귀 주소를 복사하고 동작을 한 뒤 점프
- mov $0x0 값을 eax에 복사
- leaveq : 스택 해제
- retq : 복귀 주소가 rip(다음으로 가게 될 주소) 으로 들어간다.

그림으로 나타내 보면 아래와 같다.

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdf90 |
| 0x9(i) | 0x0x7fffffffdf8c |
| | 0x0x7fffffffdf80 |
| 0x000055555551a8(복귀주소) | 0x0x7fffffffdf88 |

```
0x000055555555517c <+19>:    jmp    0x555555555198 <main+47>
0x000055555555517e <+21>:    mov    -0x4(%rbp),%eax
0x0000555555555181 <+24>:    mov    %eax,%esi
0x0000555555555183 <+26>:    lea    0xe7a(%rip),%rdi        # 0x
0x000055555555518a <+33>:    mov    $0x0,%eax
0x000055555555518f <+38>:    callq  0x555555555070 <printf@plt>
=> 0x0000555555555194 <+43>:    addl   $0x1,-0x4(%rbp)
0x0000555555555198 <+47>:    cmpl   $0x9,-0x4(%rbp)
0x000055555555519c <+51>:    jle    0x55555555517e <main+21>
0x000055555555519e <+53>:    mov    $0xa,%edi
```

ㄴ 이처럼 i 값이 9보다 작으면 숫자를 프린트 하고 i 값을 +1 하며,
이 값이 9보다 작다면 계속 반복하게 된다.

# 4. while문에 대한 기계어 분석 (test_while.c) - (1)

```c
#include <stdio.h>

int main(void)
{
        int n=4;

        while(n--)
        {
                printf("%d ",n);
        }
        printf("\n");

        return 0;
}
```

while 문 예제 작성

while 문의 어셈블리 동작을 분석하기 위한 코드.

while문을 통해 n 값을 출력 한다. ( 3 ~ 0 )

```
3 2 1 0
```

# 4. while문에 대한 기계어 분석 (test_while.c) - (2)

```
   0x0000555555555169 <+0>:     endbr64
=> 0x000055555555516d <+4>:     push   %rbp
   0x000055555555516e <+5>:     mov    %rsp,%rbp
   0x0000555555555171 <+8>:     sub    $0x10,%rsp
   0x0000555555555175 <+12>:    movl   $0x0,-0x4(%rbp)
```

자.. 이제 si를 실행해서 push %rbp명령어를 실행 했습니다.
Push 명령어는 **현재 스택의 최상위 메모리(rsp)**에 값을 저장하는 명령어 입니다.
즉, **현재 스택의 최상위 메모리(rsp)**에 **rbp값을 저장하라**는 의미 겠죠.
그런데, 앞 ppt에서 보듯이 rbp의 값은 0x0 이었으므로 아래와 같이 구성 되겠습니다.

| 0x0 (rbp) | 0x0x7fffffffdf**90** (rsp) |
|---|---|

```
(gdb) x $rsp
0x7fffffffdf90:
(gdb) x $rbp
0x0:    Cannot
```

```
   0x000055555555516d <+4>:     push   %rbp
=> 0x000055555555516e <+5>:     mov    %rsp,%rbp
   0x0000555555555171 <+8>:     sub    $0x10,%rsp
   0x0000555555555175 <+12>:    movl   $0x0,-0x4(%rbp)
```

```
(gdb) x $rsp
0x7fffffffdf90:
(gdb) x $rbp
0x7fffffffdf90:
```

자 이제 한번 더 si를 실행해서 mov %rsp, %rbp 를 시행 해봅시다.
Mov 명령어는 내용을 복사하는 것 입니다.
즉, mov rsp rbp는 rbp에 rsp 값을 복사합니다. (rsp 값 → rbp 값)
ㄴ일반적인 A = B 꼴에서 B 값이 A로 들어가는 것이 아닌 반대 방향으로 생각해야 하는 것에 주의 하자! ㄴ 사라진 경계선

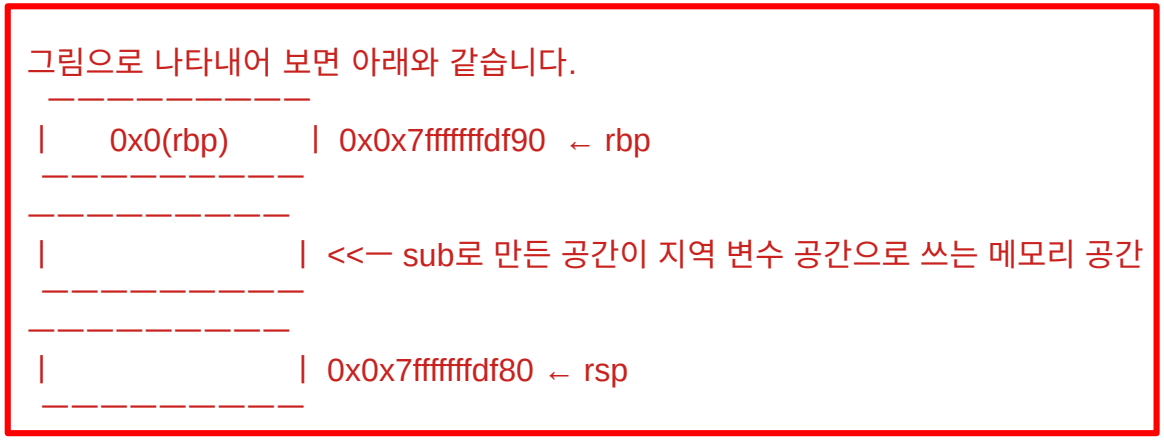
그러면 어떻게 되겠나요?
ㄴ 결국, rbp에 rsp 값을 넣어 버림 으로써 **rsp, rbp 주소 값이 서로 같아지면서 스택의 경계선이 사라집니다!**
→ 이것은 **새로운 스택을 생성 할 준비를 하는 과정** 입니다.

# 4. while문에 대한 기계어 분석 (test_while.c) - (3)

```
   0x000055555555514d <+4>:     push   %rbp
   0x000055555555514e <+5>:     mov    %rsp,%rbp
   0x0000555555555151 <+8>:     sub    $0x10,%rsp
=> 0x0000555555555155 <+12>:    movl   $0x1,-0x8(%rbp)
   0x000055555555515c <+19>:    movl   $0x2,-0x4(%rbp)
```

자 이번에는 sub 명령어를 실행 했습니다.
Sub 명령어는 뺄셈 명령 입니다.
Sub 0x10, rsp는 현재 rsp 에서 **16바이트를 빼겠다는 의미** 입니다. (0x10 → 0001 0000 : 2^4 = **16 바이트,**
**가상 주소 공간에서는 바이트 단위로 움직입니다**!)

그림으로 나타내어 보면 아래와 같습니다.




ㄴ 이런 구조 때문에 스택은 아래로 자란다고 한 것이다!!

```
(gdb) x $rbp
0x7fffffffdf90:
(gdb) x $rsp
0x7fffffffdf80:
```

: 현재 스택의 최상위 rsp 80
현재 스택의 기준점 rbp 90

# 4. while문에 대한 기계어 분석 (test_while.c) - (4)

```
   0x0000555555555171 <+8>:     sub    $0x10,%rsp
=> 0x0000555555555175 <+12>:    movl   $0x4,-0x4(%rbp)
   0x000055555555517c <+19>:    jmp    0x555555555194 <main+43>
```

자, 이번에는 movl $0x4, -0x4(%rbp)이 보인다.
L 이 들어가면 4바이트 처리를 하겠다는 의미 이며
Q 가 들어가면 8바이트 처리를 하겠다는 뜻이 된다.
여튼, 위 명령어의 의미는 **rbp를 기준으로 4바이트 뺀 자리에 4을 복사한다는 의미** 이다

그림으로 나타내 보면 아래와 같다.

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdf90 |
| 0x4(n) | 0x0x7fffffffdf8c |
| | 0x0x7fffffffdf80 |

```
(gdb) x $rbp
0x7fffffffdf90: 0x00000000
(gdb) x $rsp
0x7fffffffdf80: 0xffffe080
(gdb) x $rbp-4
0x7fffffffdf8c: 0x00000004
```

ㄴ 그림 처럼 rbp-4에 0x4 배치

```c
#include <stdio.h>

int main(void)
{
        int n=4;

        while(n--)
        {
                printf("%d ",n);
        }
        printf("\n");

        return 0;
}
```

ㄴ 원소스의 n를 초기화 하고 배치

# 4. while문에 대한 기계어 분석 (test_while.c) - (5)

```
   0x000055555555517c <+19>:    jmp    0x555555555194 <main+43>
   0x000055555555517e <+21>:    mov    -0x4(%rbp),%eax
   0x0000555555555181 <+24>:    mov    %eax,%esi
   0x0000555555555183 <+26>:    lea    0xe7a(%rip),%rdi        # 0x555555556004
   0x000055555555518a <+33>:    mov    $0x0,%eax
   0x000055555555518f <+38>:    callq  0x555555555070 <printf@plt>
=> 0x0000555555555194 <+43>:    mov    -0x4(%rbp),%eax
   0x0000555555555197 <+46>:    lea    -0x1(%rax),%edx
   0x000055555555519a <+49>:    mov    %edx,-0x4(%rbp)
   0x000055555555519d <+52>:    test   %eax,%eax
   0x000055555555519f <+54>:    jne    0x55555555517e <main+21>
```

- 0x...194 로 점프,
- mov 명령어로 rbp-4 의 값 (0x4)를 eax에 배치
- **rax의 -1뺀 값**을 의 값을 **edx**에 배치
- **edx의 값(3)**을 rbp-4에 배치
- test 연산의 설명은 아래를 참고 한다.
  (**간단히 얘기 하면 AND연산을 해봄으로써 eax의 값이 0인지 확인해 보는 것이다.)**

그림으로 나타내 보면 아래와 같다.

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdf90 |
| 0x3(n) | 0x0x7fffffffdf8c |
| | 0x0x7fffffffdf80 |

TEST[Operand 1] [Operand 2]

위의 명령어 의미는, [Operand 1]과 [Operand 2]를 AND 연산 하라는 것이다. 이 연산의 결과는 ZF에만 영향을 미치고 Operand 자체에는 영향을 미치지 않는다. 보통 TEST EAX, EAX의 식으로 많이 사용하는데, EAX의 값이 0인지 확인할 때 사용된다. (0일 때만 결과값이 0이 나올테니깐 말이다.) 만약 TEST의 연산결과가 0이라면 ZF는 1로, 연산결과가 1이라면 ZF는 0으로 세트된다.

# 4. while문에 대한 기계어 분석 (test_while.c) - (6)

```
   0x000055555555517c <+19>:    jmp    0x555555555194 <main+43>
   0x000055555555517e <+21>:    mov    -0x4(%rbp),%eax
   0x0000555555555181 <+24>:    mov    %eax,%esi
   0x0000555555555183 <+26>:    lea    0xe7a(%rip),%rdi        # 0x555555556004
   0x000055555555518a <+33>:    mov    $0x0,%eax
   0x000055555555518f <+38>:    callq  0x555555555070 <printf@plt>
   0x0000555555555194 <+43>:    mov    -0x4(%rbp),%eax
   0x0000555555555197 <+46>:    lea    -0x1(%rax),%edx
   0x000055555555519a <+49>:    mov    %edx,-0x4(%rbp)
   0x000055555555519d <+52>:    test   %eax,%eax
=> 0x000055555555519f <+54>:    jne    0x55555555517e <main+21>
```

- **Jne 값이 0 과 not equal 같지 않으면 jump!**
- rbp-4에 있던 값(3)을 eax에 배치하고
- eax 값을 esi 에 복사하고
- 0xe7a(%rip)배열 값을 rdi에 배치하고
- 0 값을 eax에 배치 한다.

그림으로 나타내 보면 아래와 같다.

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdf90 |
| 0x3(n) | 0x0x7fffffffdf8c |
| | 0x0x7fffffffdf80 |

```
   0x000055555555517c <+19>:    jmp    0x555555555194 <main+43>
   0x000055555555517e <+21>:    mov    -0x4(%rbp),%eax
   0x0000555555555181 <+24>:    mov    %eax,%esi
   0x0000555555555183 <+26>:    lea    0xe7a(%rip),%rdi        # 0x555555556004
   0x000055555555518a <+33>:    mov    $0x0,%eax
   0x000055555555518f <+38>:    callq  0x555555555070 <printf@plt>
   0x0000555555555194 <+43>:    mov    -0x4(%rbp),%eax
   0x0000555555555197 <+46>:    lea    -0x1(%rax),%edx
   0x000055555555519a <+49>:    mov    %edx,-0x4(%rbp)
   0x000055555555519d <+52>:    test   %eax,%eax
=> 0x000055555555519f <+54>:    jne    0x55555555517e <main+21>
```

- callq : 복귀주소를 push 한 뒤 pritnf 함수로 점프
- pritnf 함수에서 n 값을 출력하고
- 아래 mov 절차를 재차 반복 후..
- **test 에서 eax가 0임을 알게 되는 경우가 생김 (whilte 종료 조건)**

그림으로 나타내 보면 아래와 같다.

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdf90 |
| 0x3(n) | 0x0x7fffffffdf8c |
| | 0x0x7fffffffdf80 |
| 0x0..194(복귀주소) | 0x0x7fffffffdf88 |

# 4. while문에 대한 기계어 분석 (test_while.c) - (7)

```
   0x000055555555519d <+52>:    test   %eax,%eax
=> 0x000055555555519f <+54>:    jne    0x55555555517e <main+21>
   0x00005555555551a1 <+56>:    mov    $0xa,%edi
   0x00005555555551a6 <+61>:    callq  0x555555555060 <putchar@plt>
   0x00005555555551ab <+66>:    mov    $0x0,%eax
   0x00005555555551b0 <+71>:    leaveq
   0x00005555555551b1 <+72>:    retq
```

- mov $0xa, 값을 %edi에 복사
- callq rsp+8에 복귀 주소를 복사하고 동작을 한 뒤 점프
- mov $0x0 값을 eax에 복사
- leaveq : 스택 해제
- retq : 복귀 주소가 rip(다음으로 가게 될 주소) 으로 들어간다.

그림으로 나타내 보면 아래와 같다.

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdf90 |
| 0x3(n) | 0x0x7fffffffdf8c |
| | 0x0x7fffffffdf80 |
| 0x000055555551ab(복귀주소) | 0x0x7fffffffdf88 |

# 5. 피보나치 수열(재귀함수), 분석 - (1)

```c
int recursive_fib(int num)
{
        if(num <= 0)
        {
                printf("올바른 값을 입력 하세요!\n");
                return -1;
        }
        else if(num < 3)
        {
                return 1;
        }
        else
        {
                return recursive_fib(num-1) + recursive_fib(num-2);
        }
}

int main(void)
{
        int num, res;

        printf("몇 번째 피보나치 항을 구할까요 ? ");
        scanf("%d", &num);

        res = recursive_fib(num);
        printf("res = %d\n", res);

        return 0;
}
```

## 코드 해석

- Main 문에서 값을 알기 위한 피보나치 값을 입력 받는다.
- recursive_fib 함수가 호출된다.
- 입력받은 값이 음수이면 올바른 값을 입력 받으라는 메세지와 함께 '-1'의 값을 리턴 한다.
- 입력 받은 값이 3 미만 일 경우 '1'의 값을 리턴한다.
- 다른 경우

  **점화식 f(n) = f(n-1) + f(n-2)의 규칙**을 따른다.
  점화식 f(n)의 값을 계산 할 때 실행 중인 함수와 같은 형식의 함수 형태를 가지므로 **자기자신에서 다시 또 같은 함수를 호출**하게 된다.
  (더이상 **재귀 호출**이 필요 없을 때 까지)
  ㄴ다음 페이지에 그림으로 나타내 보자.

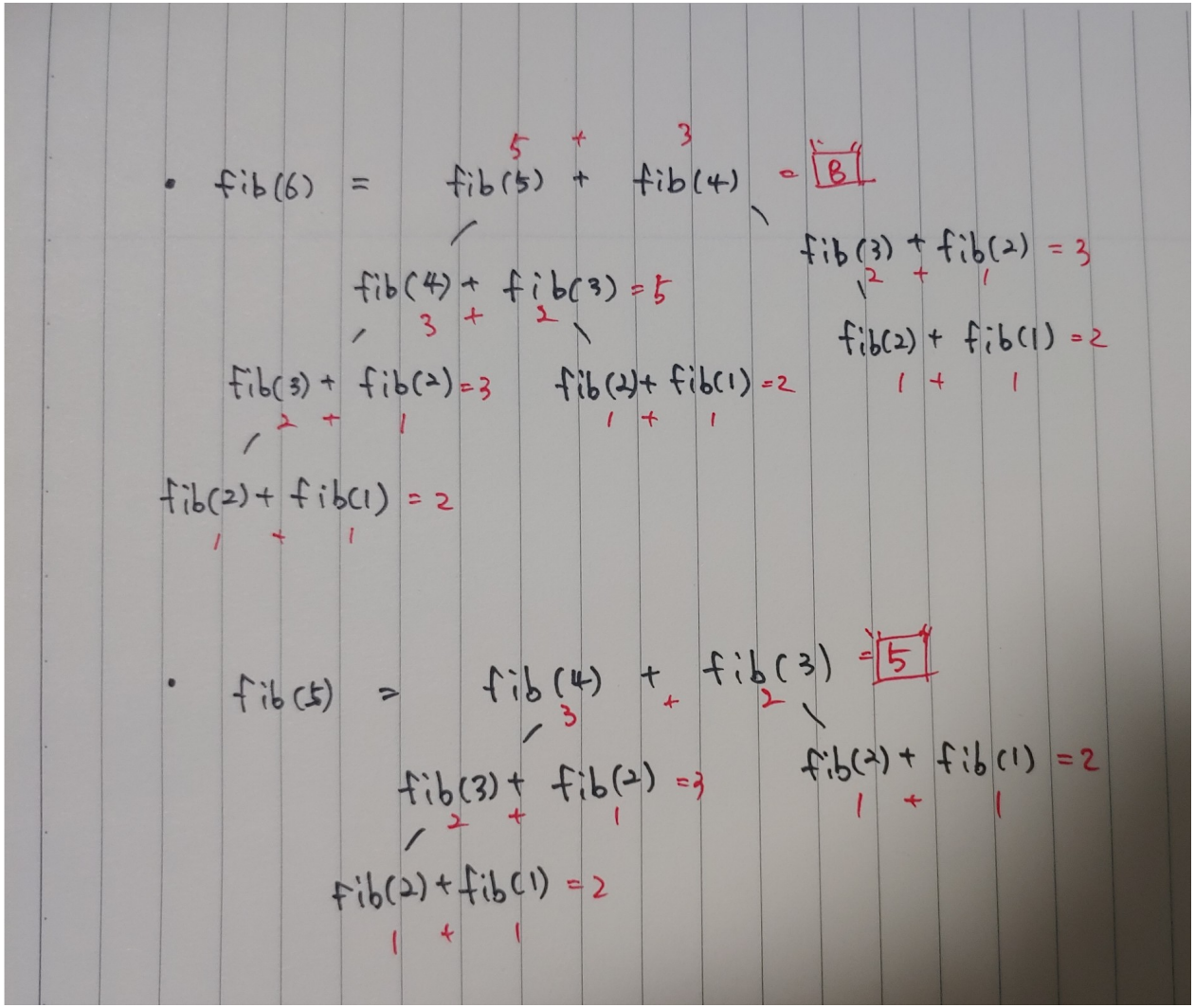
- 함수에서 리턴된 결과 값은 res 변수에 저장되고
- printf 를 통해 출력 하게 된다.

```
몇 번째 피보나치 항을 구할까요 ? 6
res = 8
```

# 5. 피보나치 수열(재귀함수), 분석 - (2)

피보나치 수열 : 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...

점화식 f(n) = f(n-1) + f(n-2)



# 6. 피보나치 재귀함 대한 기계어 분석 - (1)

```
   0x0000555555555169 <+0>:     endbr64
=> 0x000055555555516d <+4>:     push   %rbp
   0x000055555555516e <+5>:     mov    %rsp,%rbp
   0x0000555555555171 <+8>:     sub    $0x10,%rsp
   0x0000555555555175 <+12>:    movl   $0x0,-0x4(%rbp)
```

자.. 이제 si를 실행해서 push %rbp명령어를 실행 했습니다.
Push 명령어는 **현재 스택의 최상위 메모리(rsp)**에 값을 저장하는 명령어 입니다.
즉, **현재 스택의 최상위 메모리(rsp)**에 **rbp값을 저장하라**는 의미 겠죠.
그런데, 앞 ppt에서 보듯이 rbp의 값은 0x0 이었으므로 아래와 같이 구성 되겠습니다.

| 0x0 (rbp) | 0x0x7fffffffdf**90** (rsp) |
|---|---|

```
(gdb) x $rsp
0x7fffffffdf90:
(gdb) x $rbp
0x0:    Cannot
```

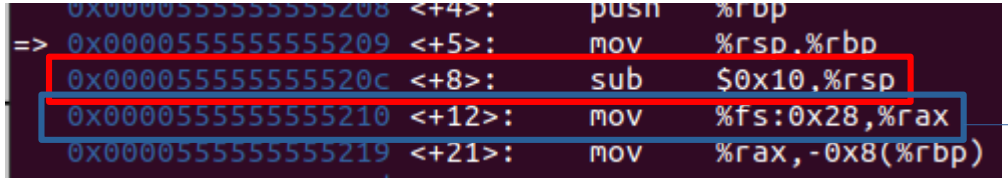
```
=> 0x000055555555516e <+5>:     mov    %rsp,%rbp
   0x0000555555555171 <+8>:     sub    $0x10,%rsp
   0x0000555555555175 <+12>:    movl   $0x0,-0x4(%rbp)
```

```
(gdb) x $rsp
0x7fffffffdf90:
(gdb) x $rbp
0x7fffffffdf90:
```

자 이제 한번 더 si를 실행해서 mov %rsp, %rbp 를 시행 해봅시다.
Mov 명령어는 내용을 복사하는 것 입니다.
즉, mov rsp rbp는 rbp에 rsp 값을 복사합니다. (rsp 값 → rbp 값)
ㄴ일반적인 A = B 꼴에서 B 값이 A로 들어가는 것이 아닌 반대 방향으로 생각해야 하는 것에 주의 하자! ㄴ 사라진 경계선

그러면 어떻게 되겠나요?
ㄴ 결국, rbp에 rsp 값을 넣어 버림 으로써 **rsp, rbp 주소 값이 서로 같아지면서 스택의 경계선이 사라집니다!**
→ 이것은 **새로운 스택을 생성 할 준비를 하는 과정** 입니다.

# 6. 피보나치 재귀함 대한 기계어 분석 - (2)




스택 공격하는 방어 코드 입니다.
해킹 방지라고 보면 됩니다.
(참고 : 스택오버플로우 공격 방지용이라 합니다.)

자 이번에는 sub 명령어를 실행 했습니다.
Sub 명령어는 뺄셈 명령 입니다.
Sub 0x10, rsp는 현재 rsp 에서 **16바이트를 빼겠다는 의미** 입니다. (0x10 → 0001 0000 : 2^4 = **16 바이트,**
**가상 주소 공간에서는 바이트 단위로 움직입니다**!)

그림으로 나타내어 보면 아래와 같습니다.

| 0x0(rbp) | 0x0x7fffffffdf90 ← rbp |
|---|---|
| | <<— sub로 만든 공간이 지역 변수 공간으로 쓰는 메모리 공간 |
| | 0x0x7fffffffdf80 ← rsp |

ㄴ 이런 구조 때문에 스택은 아래로 자란다고 한 것이다!!

```
(gdb) x $rbp
0x7fffffffdf90:
(gdb) x $rsp
0x7fffffffdf80:
```

: 현재 스택의 최상위 rsp 80
현재 스택의 기준점 rbp 90

# 6. 피보나치 재귀함 대한 기계어 분석 - (2)

```
=> 0x0000555555555209 <+5>:     mov    %rsp,%rbp
   0x000055555555520c <+8>:     sub    $0x10,%rsp
   0x0000555555555219 <+21>:    mov    %rax,-0x8(%rbp)
```

Rax 값을 rbp-8에 배치 한다.


```
(gdb) x $rbp
0x7fffffffdf90:
(gdb) x $rsp
0x7fffffffdf80:
```

: 현재 스택의 최상위 rsp 80
현재 스택의 기준점 rbp 90

```
0x0000555555555219 <+21>:    mov    %rax,-0x8(%rbp)
0x000055555555521d <+25>:    xor    %eax,%eax
```

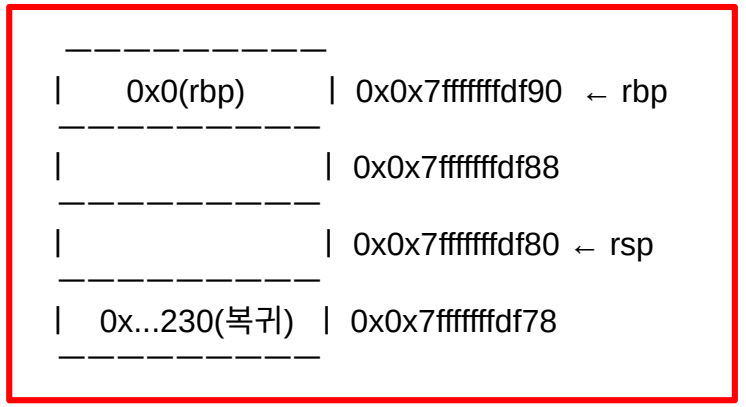
XOR 연산이며, XOR 는 비교 값이 다를 경우에만 '1' 이다.
따라서 XOR는 **서로 같은 값을 가질 경우 '0'** 이니
**초기화 방법 중의 하나** 이다.

# 6. 피보나치 재귀함 대한 기계어 분석 - (3)

```
=> 0x000055555555521d <+25>:    xor    %eax,%eax
   0x000055555555521f <+27>:    lea    0xe0a(%rip),%rdi        # 0x555555556030
   0x0000555555555226 <+34>:    mov    $0x0,%eax
   0x000055555555522b <+39>:    callq  0x5555555550a0 <printf@plt>
   0x0000555555555230 <+44>:    lea    -0x10(%rbp),%rax
   0x0000555555555234 <+48>:    mov    %rax,%rsi
   0x0000555555555237 <+51>:    lea    0xe21(%rip),%rdi        # 0x55555555605f
   0x000055555555523e <+58>:    mov    $0x0,%eax
   0x0000555555555243 <+63>:    callq  0x5555555550b0 <__isoc99_scanf@plt>
```

- lea 0xe0a(%rip), $rdi
  ㄴ 0xe0a(%rip) 배열 값 rdi 에 배치 한다.
- mov $0x0, %eax
  ㄴ eax 값을 0 초기화
- callq 복귀주소를 다음 8 바이트 스택에 push 하고 printf 함수로 점프
- rbp-16 주소를 rax에 배치
- rax를 rsi에 배치
- lea %rip 배열 값을 rdi에 배치
- eax를 0으로 초기화

```
(gdb) x $rsp-8
0x7fffffffdf78: 0x55555230
(gdb) x $rsp
0x7fffffffdf80: 0xffffe080
(gdb) x $rbp
0x7fffffffdf90: 0x00000000
```



# 6. 피보나치 재귀함 대한 기계어 분석 - (4)

```
=> 0x0000555555555243 <+63>:    callq  0x5555555550b0 <__isoc99_scanf@plt>
   0x0000555555555248 <+68>:    mov    -0x10(%rbp),%eax
   0x000055555555524b <+71>:    mov    %eax,%edi
   0x000055555555524d <+73>:    callq  0x55555555551a9 <recursive_fib>
   0x0000555555555252 <+78>:    mov    %eax,-0xc(%rbp)
```

ㄴ 복귀 주소인 248을 push 하고 scanf 로 점프.

```
(gdb) ni
몇 번째 피보나치 항을 구할까요 ? 5
35              res = recursive_fib(num);
```

ㄴ 5 의 항 값을 입력.(scanf)

```
(gdb) x $rsp
0x7fffffffdf80: 0x00000005
(gdb) x $rsp-8
0x7fffffffdf78: 0x55555248
```

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdf90 ← rbp |
| | 0x0x7fffffffdf88 |
| 5 | 0x0x7fffffffdf80 ← rsp |
| 0x...248(복귀) | 0x0x7fffffffdf78 |

```
   0x0000555555555243 <+63>:    callq  0x5555555550b0 <__isoc99_scanf@plt>
=> 0x0000555555555248 <+68>:    mov    -0x10(%rbp),%eax
   0x000055555555524b <+71>:    mov    %eax,%edi
   0x000055555555524d <+73>:    callq  0x55555555551a9 <recursive_fib>
   0x0000555555555252 <+78>:    mov    %eax,-0xc(%rbp)
```

ㄴ scanf 값을 받은 뒤 rbp-16 값을 eax에 배치 한다.
ㄴ eax 값을 edi 에 복사한다
ㄴ 드디어.. 재귀함수다.. 재귀함수 호출을 위해
복귀 주소를 복사한뒤 함수로 점프 한다.

```
(gdb) x $eax
0x5:    Cannot access memory
(gdb) x $rbp-16
0x7fffffffdf80: 0x00000005
```

```
(gdb) x $rsp
0x7fffffffdf78: 0x55555252
```

: 복귀주소인 252가 들어가고 rsp가 이것이 됨.

# 6. 피보나치 재귀함 대한 기계어 분석 - (5)

```
Dump of assembler code for function recursive_fib:
=> 0x00005555555551a9 <+0>:     endbr64
   0x00005555555551ad <+4>:     push   %rbp
   0x00005555555551ae <+5>:     mov    %rsp,%rbp
   0x00005555555551b1 <+8>:     push   %rbx
   0x00005555555551b2 <+9>:     sub    $0x18,%rsp
   0x00005555555551b6 <+13>:    mov    %edi,-0x14(%rbp)
   0x00005555555551b9 <+16>:    cmpl   $0x0,-0x14(%rbp)
   0x00005555555551bd <+20>:    jg     0x5555555551d2 <recursive_fib+41>
   0x00005555555551bf <+22>:    lea    0xe42(%rip),%rdi        # 0x555555556008
   0x00005555555551c6 <+29>:    callq  0x555555555080 <puts@plt>
   0x00005555555551cb <+34>:    mov    $0xffffffff,%eax
   0x00005555555551d0 <+39>:    jmp    0x5555555551fd <recursive_fib+84>
   0x00005555555551d2 <+41>:    cmpl   $0x2,-0x14(%rbp)
   0x00005555555551d6 <+45>:    jg     0x5555555551df <recursive_fib+54>
```

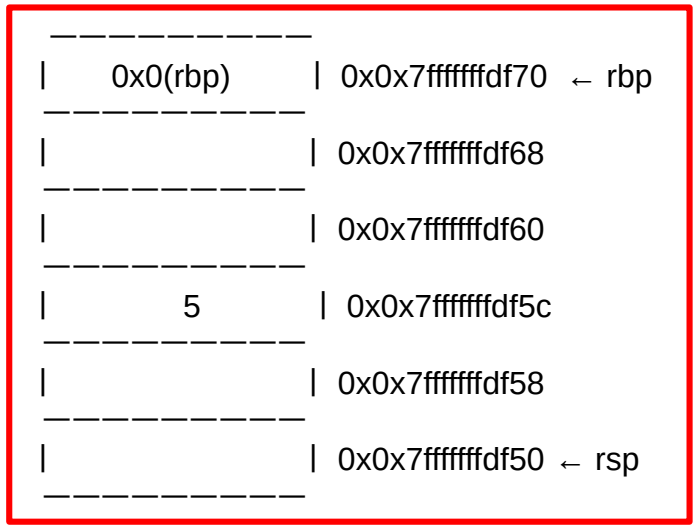
재귀 함수 호출 된 모습이다.

- rbp push
- rsp , rbp 경계선 없어짐
- rbx push
- 32 바이트 아래까지 rsp 생성
- edi(5) 값을 rpb-20에 배치
- **cmpl 비교, 0값과 rpb-20 (5) 값과 비교**
- **jg (결과가 크면 점프) 1d2로 점프**
  **→ 앞의 비교 값이 0보다 크므로 다음으로 점프**

```
(gdb) x $rsp
0x7fffffffdf50:
(gdb) x $rbp
0x7fffffffdf70:
```

```c
int recursive_fib(int num)
{
        if(num <= 0)
        {
                printf("올바른 값을 입력 하세요!\n");
                return -1;
        }
        else if(num < 3)
        {
                return 1;
        }
```

원문의 0 과의 비교에서 0 보다 큰 값이 들어 왔으니 아래 조건문으로 점프 한 것.



# 6. 피보나치 재귀함 대한 기계어 분석 - (6)

```
=> 0x00005555555551d2 <+41>:    cmpl   $0x2,-0x14(%rbp)
   0x00005555555551d6 <+45>:    jg     0x5555555551df <recursive_fib+54>
   0x00005555555551d8 <+47>:    mov    $0x1,%eax
   0x00005555555551dd <+52>:    jmp    0x5555555551fd <recursive_fib+84>
   0x00005555555551df <+54>:    mov    -0x14(%rbp),%eax
   0x00005555555551e2 <+57>:    sub    $0x1,%eax
   0x00005555555551e5 <+60>:    mov    %eax,%edi
   0x00005555555551e7 <+62>:    callq  0x5555555551a9 <recursive_fib>
   0x00005555555551ec <+67>:    mov    %eax,%ebx
```

이번에는 2의 값과 rpb-20 (5)의 값을 비교해
Jb 비교하는 값이 더 크므로
1df 주소로 다시 한번 점프하게 된다.

```
int recursive_fib(int num)
{
        if(num <= 0)
        {
                printf("올바른 값을 입력 하세요!\n");
                return -1;
        }
        else if(num < 3)
        {
                return 1;
        }
```

원문의 3 과의 비교에서 3 보다 큰 값이 들어 왔으니 아래 조건문으로 점프 한 것.

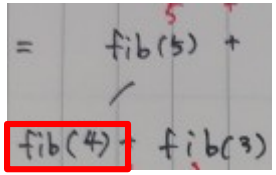
| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdf70 ← rbp |
| | 0x0x7fffffffdf68 |
| | 0x0x7fffffffdf60 |
| 5 | 0x0x7fffffffdf5c |
| | 0x0x7fffffffdf58 |
| | 0x0x7fffffffdf50 ← rsp |

# 6. 피보나치 재귀함 대한 기계어 분석 - (7)

```
=> 0x00005555555551d2 <+41>:    cmpl   $0x2,-0x14(%rbp)
   0x00005555555551d6 <+45>:    jg     0x5555555551df <recursive_fib+54>
   0x00005555555551d8 <+47>:    mov    $0x1,%eax
   0x00005555555551dd <+52>:    jmp    0x5555555551fd <recursive_fib+84>
   0x00005555555551df <+54>:    mov    -0x14(%rbp),%eax
   0x00005555555551e2 <+57>:    sub    $0x1,%eax
   0x00005555555551e5 <+60>:    mov    %eax,%edi
   0x00005555555551e7 <+62>:    callq  0x5555555551a9 <recursive_fib>
   0x00005555555551ec <+67>:    mov    %eax,%ebx
```

1df로 넘어와서는 rbp-20(5)의 값을 eax에 배치한다.
1의 값을 eax에서 뺀다
Eax 값을 edi(4)에 복사한다.

그리고 다시 재귀함수를 호출한다.
복귀 주소 1ec를 rsp-8에 push하고
재귀함수 안에서 다시 또 재호출 한다.




```
        }
        else if(num < 3)
        {
                return 1;
        }
        else
        {
                return recursive_fib(num-1) + recursive_fib(num-2);
        }
}
```

원문의 3 과의 비교에서 3 보다 큰 값이 들어 왔으니 아래 조건문으로 점프 한 것.

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdf70 ← rbp |
| | 0x0x7fffffffdf68 |
| | 0x0x7fffffffdf60 |
| 5 | 0x0x7fffffffdf5c |
| | 0x0x7fffffffdf58 |
| | 0x0x7fffffffdf50 |
| 1ec | 0x0x7fffffffdf48 ← rsp |

```
(gdb) x $rsp
0x7fffffffdf48: 0x555551ec
```

포기하면 얻는 건 아무것도 없다.

# 6. 피보나치 재귀함 대한 기계어 분석 - (8)

```
   0x00005555555551a9 <+0>:     endbr64
   0x00005555555551ad <+4>:     push   %rbp
   0x00005555555551ae <+5>:     mov    %rsp,%rbp
   0x00005555555551b1 <+8>:     push   %rbx
=> 0x00005555555551b2 <+9>:     sub    $0x18,%rsp
   0x00005555555551b6 <+13>:    mov    %edi,-0x14(%rbp)
   0x00005555555551b9 <+16>:    cmpl   $0x0,-0x14(%rbp)
   0x00005555555551bd <+20>:    jg     0x5555555551d2 <recursive_fib+41>
   0x00005555555551bf <+22>:    lea    0xe42(%rip),%rdi        # 0x555555556008
   0x00005555555551c6 <+29>:    callq  0x555555555080 <puts@plt>
   0x00005555555551cb <+34>:    mov    $0xffffffff,%eax
   0x00005555555551d0 <+39>:    jmp    0x5555555551fd <recursive_fib+84>
   0x00005555555551d2 <+41>:    cmpl   $0x2,-0x14(%rbp)
   0x00005555555551d6 <+45>:    jg     0x5555555551df <recursive_fib+54>
```

이런 젠장..
다시 한번 재귀 함수 호출 된 모습이다.
다시 시작된 rbp..

- rbp push
- rsp , rbp 경계선 없어짐
- rbx push
- 32 바이트 아래까지 rsp 생성
- edi(4) 값을 rpb-20에 배치
- **cmpl 비교, 0값과 rpb-20 (4) 값과 비교**
- **jg (결과가 크면 점프) 1d2로 점프**
  **→ 앞의 비교 값이 0보다 크므로 다음으로 점프**

```
int recursive_fib(int num)
{
        if(num <= 0)
        {
                printf("올바른 값을 입력 하세요!\n");
                return -1;
        }
        else if(num < 3)
        {
                return 1;
        }
```

원문의 0 과의 비교에서 0 보다 큰 값이 들어 왔으니 아래 조건문으로 점프 한 것.

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdf40 ← rbp |
| | 0x0x7fffffffdf38 |
| | 0x0x7fffffffdf30 |
| 4 | 0x0x7fffffffdf2c |
| | 0x0x7fffffff28 |
| | 0x0x7fffffffdf20 ← rsp |

# 6. 피보나치 재귀함 대한 기계어 분석 - (9)

```
   0x00005555555551d0 <+39>:    jmp    0x5555555551fd <recursive_fib+84>
=> 0x00005555555551d2 <+41>:    cmpl   $0x2,-0x14(%rbp)
   0x00005555555551d6 <+45>:    jg     0x5555555551df <recursive_fib+54>
   0x00005555555551d8 <+47>:    mov    $0x1,%eax
   0x00005555555551dd <+52>:    jmp    0x5555555551fd <recursive_fib+84>
   0x00005555555551df <+54>:    mov    -0x14(%rbp),%eax
```

1d2로 넘어와서는 rbp-20(4)의 값을
2의 값과 비교하고

Jb 4의 값은 2보다 크므로
1df 주소로 jump 한다.

```
(gdb) x $rbp-20
0x7fffffffdf2c: 0x00000004
```

```
int recursive_fib(int num)
{
        if(num <= 0)
        {
                printf("올바른 값을 입력 하세요!\n");
                return -1;
        }
        else if(num < 3)
        {
                return 1;
        }
```

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdf40 ← rbp |
| | 0x0x7fffffffdf38 |
| | 0x0x7fffffffdf30 |
| 4 | 0x0x7fffffffdf2c |
| | 0x0x7fffffff28 |
| | 0x0x7fffffffdf20 ← rsp |

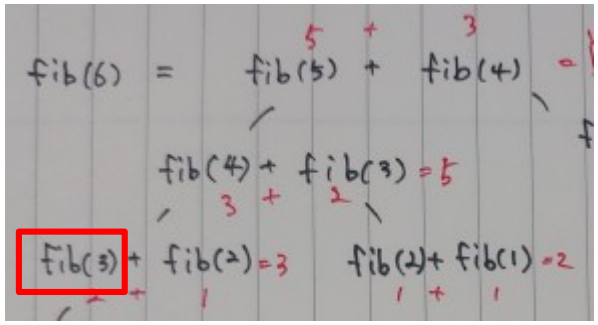
원문의 3 과의 비교에서 2 보다 큰 값이 들어 왔으니 아래 조건문으로 점프 한 것.

# 6. 피보나치 재귀함 대한 기계어 분석 - (10)

```
=> 0x00005555555551d2 <+41>:    cmpl   $0x2,-0x14(%rbp)
   0x00005555555551d6 <+45>:    jg     0x5555555551df <recursive_fib+54>
   0x00005555555551d8 <+47>:    mov    $0x1,%eax
   0x00005555555551dd <+52>:    jmp    0x5555555551fd <recursive_fib+84>
   0x00005555555551df <+54>:    mov    -0x14(%rbp),%eax
   0x00005555555551e2 <+57>:    sub    $0x1,%eax
   0x00005555555551e5 <+60>:    mov    %eax,%edi
   0x00005555555551e7 <+62>:    callq  0x5555555551a9 <recursive_fib>
   0x00005555555551ec <+67>:    mov    %eax,%ebx
```

1df로 넘어와서는 rbp-20(4)의 값을 eax에 배치한다.
1의 값을 eax에서 뺀다
Eax 값을 edi(3)에 복사한다.

그리고 다시 재귀함수를 호출한다.
복귀 주소 1ec를 rsp-8에 push하고
재귀함수 안에서 다시 또 재호출 한다. (하이고마..)




```
        }
        else if(num < 3)
        {
                return 1;
        }
        else
        {
                return recursive_fib(num-1) + recursive_fib(num-2);
        }
}
```

원문의 3 과의 비교에서 3 보다 큰 값이 들어 왔으니 아래 조건문으로 점프 한 것.

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdf40 ← rbp |
| | 0x0x7fffffffdf38 |
| | 0x0x7fffffffdf30 |
| 4 | 0x0x7fffffffdf2c |
| | 0x0x7fffffff28 |
| | 0x0x7fffffffdf20 |
| 1ec | 0x0x7fffffffdf18 ← rsp |

```
(gdb) x $rsp
0x7fffffffdf18: 0x555551ec
```

포기하면 얻는 건 아무것도 없다.

# 6. 피보나치 재귀함 대한 기계어 분석 - (11)

```
   0x00005555555551a9 <+0>:     endbr64
   0x00005555555551ad <+4>:     push   %rbp
   0x00005555555551ae <+5>:     mov    %rsp,%rbp
   0x00005555555551b1 <+8>:     push   %rbx
=> 0x00005555555551b2 <+9>:     sub    $0x18,%rsp
   0x00005555555551b6 <+13>:    mov    %edi,-0x14(%rbp)
   0x00005555555551b9 <+16>:    cmpl   $0x0,-0x14(%rbp)
   0x00005555555551bd <+20>:    jg     0x5555555551d2 <recursive_fib+41>
   0x00005555555551bf <+22>:    lea    0xe42(%rip),%rdi        # 0x555555556008
   0x00005555555551c6 <+29>:    callq  0x555555555080 <puts@plt>
   0x00005555555551cb <+34>:    mov    $0xffffffff,%eax
   0x00005555555551d0 <+39>:    jmp    0x5555555551fd <recursive_fib+84>
   0x00005555555551d2 <+41>:    cmpl   $0x2,-0x14(%rbp)
   0x00005555555551d6 <+45>:    jg     0x5555555551df <recursive_fib+54>
```

이런 젠장..
다시 한번 재귀 함수 호출 된 모습이다.
다시 시작된 rbp..

- rbp push
- rsp , rbp 경계선 없어짐
- rbx push
- 32 바이트 아래까지 rsp 생성
- edi(3) 값을 rpb-20에 배치
- **cmpl 비교, 0값과 rpb-20 (3) 값과 비교**
- **jg (결과가 크면 점프) 1d2로 점프**
  **→ 앞의 비교 값이 0보다 크므로 다음으로 점프**

```
int recursive_fib(int num)
{
        if(num <= 0)
        {
                printf("올바른 값을 입력 하세요!\n");
                return -1;
        }
        else if(num < 3)
        {
                return 1;
        }
```

원문의 0 과의 비교에서 0 보다 큰 값이 들어 왔으니 아래 조건문으로 점프 한 것.

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdf10 ← rbp |
| | 0x0x7fffffffdf08 |
| | 0x0x7fffffffdf00 |
| 3 | 0x0x7fffffffdffc |
| | 0x0x7fffffff8 |
| | 0x0x7fffffffdff0 ← rsp |

```
(gdb) x $rsp
0x7fffffffdef0: 0x00000000
(gdb) x $rbp
0x7fffffffdf10: 0xffffdf40
(gdb) x $rbp-20
0x7fffffffdefc: 0x00000003
```

포기하면 얻는 건 아무것도 없다.

# 6. 피보나치 재귀함 대한 기계어 분석 - (12)

```
   0x00005555555551d0 <+39>:    jmp    0x5555555551fd <recursive_fib+84>
=> 0x00005555555551d2 <+41>:    cmpl   $0x2,-0x14(%rbp)
   0x00005555555551d6 <+45>:    jg     0x5555555551df <recursive_fib+54>
   0x00005555555551d8 <+47>:    mov    $0x1,%eax
   0x00005555555551dd <+52>:    jmp    0x5555555551fd <recursive_fib+84>
   0x00005555555551df <+54>:    mov    -0x14(%rbp),%eax
```

1d2로 넘어와서는 rbp-20(3)의 값을
2의 값과 비교하고

Jb 4의 값은 2보다 크므로
1df 주소로 jump 한다.

```
(gdb) x $rbp-20
0x7fffffffdefc: 0x00000003
```

```
int recursive_fib(int num)
{
        if(num <= 0)
        {
                printf("올바른 값을 입력 하세요!\n");
                return -1;
        }
        else if(num < 3)
        {
                return 1;
        }
```

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdf10 ← rbp |
| | 0x0x7fffffffdf08 |
| | 0x0x7fffffffdf00 |
| 3 | 0x0x7fffffffdffc |
| | 0x0x7ffffffff8 |
| | 0x0x7fffffffdff0 ← rsp |

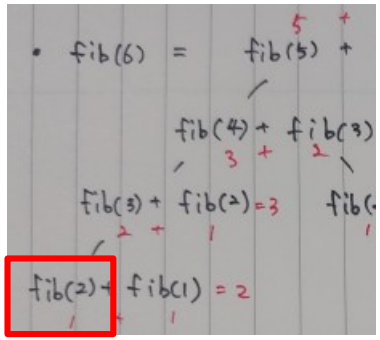
원문의 3 과의 비교에서 2 보다 큰 값이 들어 왔으니 아래 조건문으로 점프 한 것.

# 6. 피보나치 재귀함 대한 기계어 분석 - (13)

```
=> 0x00005555555551d2 <+41>:    cmpl   $0x2,-0x14(%rbp)
   0x00005555555551d6 <+45>:    jg     0x5555555551df <recursive_fib+54>
   0x00005555555551d8 <+47>:    mov    $0x1,%eax
   0x00005555555551dd <+52>:    jmp    0x5555555551fd <recursive_fib+84>
   0x00005555555551df <+54>:    mov    -0x14(%rbp),%eax
   0x00005555555551e2 <+57>:    sub    $0x1,%eax
   0x00005555555551e5 <+60>:    mov    %eax,%edi
   0x00005555555551e7 <+62>:    callq  0x5555555551a9 <recursive_fib>
   0x00005555555551ec <+67>:    mov    %eax,%ebx
```

1df로 넘어와서는 rbp-20(3)의 값을 eax에 배치한다.
1의 값을 eax에서 뺀다
Eax 값을 edi(2)에 복사한다.

그리고 다시 재귀함수를 호출한다.
복귀 주소 1ec를 rsp-8에 push하고
재귀함수 안에서 다시 또 재호출 한다. (하이고마..)




```
        }
        else if(num < 3)
        {
                return 1;
        }
        else
        {
                return recursive_fib(num-1) + recursive_fib(num-2);
        }
}
```

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdf10 ← rbp |
| | 0x0x7fffffffdf08 |
| | 0x0x7fffffffdf00 |
| 3 | 0x0x7fffffffdffc |
| | 0x0x7fffffff8 |
| | 0x0x7fffffffdff0 |
| 1ec | 0x0x7fffffffdfe8 ← rsp |

```
(gdb) x $rsp
0x7fffffffdee8: 0x555551ec
```

# 6. 피보나치 재귀함 대한 기계어 분석 - (14)

```
   0x00005555555551a9 <+0>:     endbr64
   0x00005555555551ad <+4>:     push   %rbp
   0x00005555555551ae <+5>:     mov    %rsp,%rbp
   0x00005555555551b1 <+8>:     push   %rbx
=> 0x00005555555551b2 <+9>:     sub    $0x18,%rsp
   0x00005555555551b6 <+13>:    mov    %edi,-0x14(%rbp)
   0x00005555555551b9 <+16>:    cmpl   $0x0,-0x14(%rbp)
   0x00005555555551bd <+20>:    jg     0x5555555551d2 <recursive_fib+41>
   0x00005555555551bf <+22>:    lea    0xe42(%rip),%rdi        # 0x555555556008
   0x00005555555551c6 <+29>:    callq  0x555555555080 <puts@plt>
   0x00005555555551cb <+34>:    mov    $0xffffffff,%eax
   0x00005555555551d0 <+39>:    jmp    0x5555555551fd <recursive_fib+84>
   0x00005555555551d2 <+41>:    cmpl   $0x2,-0x14(%rbp)
   0x00005555555551d6 <+45>:    jg     0x5555555551df <recursive_fib+54>
```

이런 젠장..
다시 한번 재귀 함수 호출 된 모습이다.
다시 시작된 rbp..

- rbp push
- rsp , rbp 경계선 없어짐
- rbx push
- 32 바이트 아래까지 rsp 생성
- edi(2) 값을 rpb-20에 배치
- **cmpl 비교, 0값과 rpb-20 (2) 값과 비교**
- **jg (결과가 크면 점프) 1d2로 점프**
  **→ 앞의 비교 값이 0보다 크므로 다음으로 점프**

```
int recursive_fib(int num)
{
        if(num <= 0)
        {
                printf("올바른 값을 입력 하세요!\n");
                return -1;
        }
        else if(num < 3)
        {
                return 1;
        }
```

원문의 0 과의 비교에서 0 보다 큰 값이 들어 왔으니 아래 조건문으로 점프 한 것.

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdfe0 ← rbp |
| | 0x0x7fffffffdfd8 |
| | 0x0x7fffffffdfd0 |
| 2 | 0x0x7fffffffdfcc |
| | 0x0x7fffffffc8 |
| | 0x0x7fffffffdfc0 ← rsp |

```
(gdb) x $rsp
0x7fffffffdec0: 0x00000380
(gdb) x $rbp
0x7fffffffdee0: 0xffffdf10
(gdb) x $rbp-20
0x7fffffffdecc: 0x00000002
```

# 6. 피보나치 재귀함 대한 기계어 분석 - (15)

```
   0x00005555555551d0 <+39>:    jmp    0x5555555551fd <recursive_fib+84>
=> 0x00005555555551d2 <+41>:    cmpl   $0x2,-0x14(%rbp)
   0x00005555555551d6 <+45>:    jg     0x5555555551df <recursive_fib+54>
   0x00005555555551d8 <+47>:    mov    $0x1,%eax
   0x00005555555551dd <+52>:    jmp    0x5555555551fd <recursive_fib+84>
   0x00005555555551df <+54>:    mov    -0x14(%rbp),%eax
```

1d2로 넘어와서는 rbp-20(2)의 값을
2의 값과 비교하고

Jb 2의 값은 2보다 크지 않으므로
Mov 1의 값을 eax에 복사 한다.

그리고 1fd의 주소로 점프.

```
(gdb) p/x $eax
$5 = 0x1
```

```
(gdb) x $rbp-20
0x7fffffffdecc: 0x00000002
```

```
int recursive_fib(int num)
{
        if(num <= 0)
        {
                printf("올바른 값을 입력 하세요!\n");
                return -1;
        }
        else if(num < 3)
        {
                return 1;
        }
```

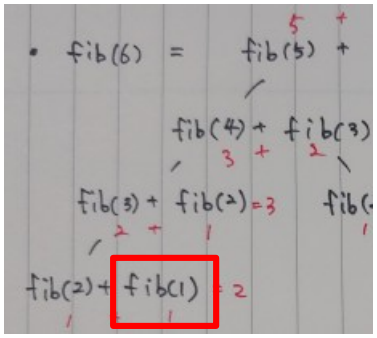
원문의 2 과의 비교에서 2의 값보다 크지 않으므로 return 1 을 하게 된다.

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdfe0 ← rbp |
| | 0x0x7fffffffdfd8 |
| | 0x0x7fffffffdfd0 |
| 2 | 0x0x7fffffffdfcc |
| | 0x0x7fffffffc8 |
| | 0x0x7fffffffdfc0 ← rsp |

# 6. 피보나치 재귀함 대한 기계어 분석 - (16)

```
   0x00005555555551f6 <+77>:    callq  0x5555555551a9 <recursive_fib>
   0x00005555555551fb <+82>:    add    %ebx,%eax
=> 0x00005555555551fd <+84>:    add    $0x18,%rsp
   0x0000555555555201 <+88>:    pop    %rbx
   0x0000555555555202 <+89>:    pop    %rbp
   0x0000555555555203 <+90>:    retq
```

- 1fd로 점프되었으며,
- Rsp에 +28 바이트를 합니다. (d8)
- Pop : 스택으로 부터 값을 뽑아 냅니다.
  - ㄴ rbx값
  - ㄴ rbp값
- retq , 본 함수로 복귀.(rsp에 있던 복귀주소로 복귀)
  - ㄴ n-2 함수 실행으로 감.




```
        }
        else if(num < 3)
        {
                return 1;
        }
        else
        {
                return recursive_fib(num-1) + recursive_fib(num-2);
        }
}
```

```
(gdb) x $rbp
0x7fffffffdf10: 0xffffdf40
(gdb) x $rsp
0x7fffffffdef0: 0x00000000
(gdb) x $rbp-20
0x7fffffffdefc: 0x00000003
```

ㄴ 이것이 바로 왼쪽의 저 곳으로 돌아 왔다는 증거다!!

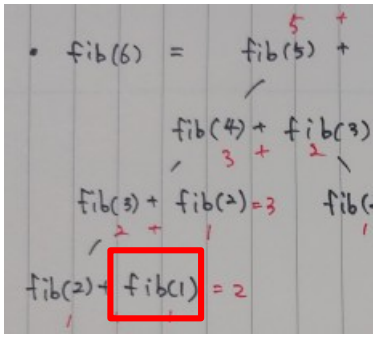
| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdf10 ← rbp |
| | 0x0x7fffffffdf08 |
| | 0x0x7fffffffdf00 |
| 3 | 0x0x7fffffffdffc |
| | 0x0x7ffffffff8 |
| | 0x0x7fffffffdff0 |

# 6. 피보나치 재귀함 대한 기계어 분석 - (17)

```
   0x000055555555551e7 <+62>:    callq  0x555555551a9 <recursive_fib>
=> 0x000055555555551ec <+67>:    mov    %eax,%ebx
   0x000055555555551ee <+69>:    mov    -0x14(%rbp),%eax
   0x000055555555551f1 <+72>:    sub    $0x2,%eax
   0x000055555555551f4 <+75>:    mov    %eax,%edi
```

- eax 값을 ebx로 복사
- rbp-20(3)을 eax에 배치
- eax – 2
- eax 값을 edi 에 복사

```
(gdb) p/x $eax
$9 = 0x1
```




```
(gdb) x $rbp
0x7fffffffdf10: 0xffffdf40
(gdb) x $rsp
0x7fffffffdef0: 0x00000000
(gdb) x $rbp-20
0x7fffffffdefc: 0x00000003
```

ㄴ 이것이 바로 왼쪽의 저 곳으로 돌아 왔다는 증거다!!
f(3)을 넣은거에 f(2)가 return 되고 fib(1)차례가 된 것이다.

```
    }
    else if(num < 3)
    {
            return 1;
    }
    else
    {
            return recursive_fib(num-1) + recursive_fib(num-2);
    }
}
```

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdf10 ← rbp |
| | 0x0x7fffffffdf08 |
| | 0x0x7fffffffdf00 |
| 3 - 2 | 0x0x7fffffffdffc |
| | 0x0x7ffffffff8 |
| | 0x0x7fffffffdff0 |

# 6. 피보나치 재귀함 대한 기계어 분석 - (18)

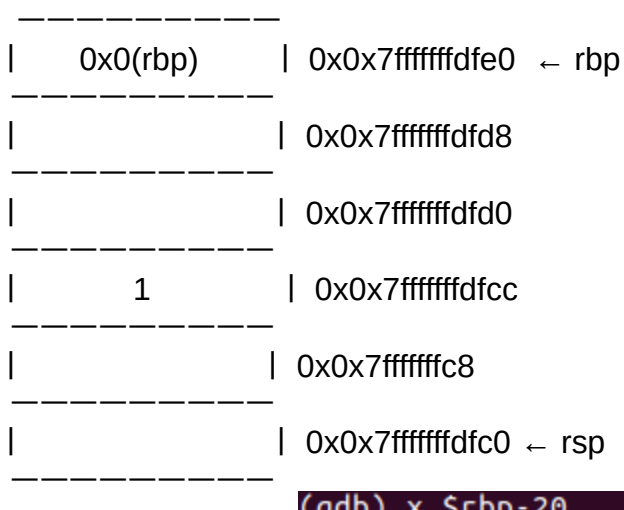
```
   0x00005555555551a9 <+0>:     endbr64
   0x00005555555551ad <+4>:     push   %rbp
   0x00005555555551ae <+5>:     mov    %rsp,%rbp
   0x00005555555551b1 <+8>:     push   %rbx
=> 0x00005555555551b2 <+9>:     sub    $0x18,%rsp
   0x00005555555551b6 <+13>:    mov    %edi,-0x14(%rbp)
   0x00005555555551b9 <+16>:    cmpl   $0x0,-0x14(%rbp)
   0x00005555555551bd <+20>:    jg     0x5555555551d2 <recursive_fib+41>
   0x00005555555551bf <+22>:    lea    0xe42(%rip),%rdi        # 0x555555556008
   0x00005555555551c6 <+29>:    callq  0x555555555080 <puts@plt>
   0x00005555555551cb <+34>:    mov    $0xffffffff,%eax
   0x00005555555551d0 <+39>:    jmp    0x5555555551fd <recursive_fib+84>
   0x00005555555551d2 <+41>:    cmpl   $0x2,-0x14(%rbp)
   0x00005555555551d6 <+45>:    jg     0x5555555551df <recursive_fib+54>
```

다시 시작된 rbp..

- rbp push
- rsp , rbp 경계선 없어짐
- rbx push
- 32 바이트 아래까지 rsp 생성
- edi(2) 값을 rpb-20에 배치
- **cmpl 비교, 0값과 rpb-20 (1) 값과 비교**
- **jg (결과가 크면 점프) 1d2로 점프**
  **→ 앞의 비교 값이 0보다 크므로 다음으로 점프**

```
int recursive_fib(int num)
{
        if(num <= 0)
        {
                printf("올바른 값을 입력 하세요!\n");
                return -1;
        }
        else if(num < 3)
        {
                return 1;
        }
```

원문의 0 과의 비교에서 0 보다 큰 값이 들어 왔으니 아래 조건문으로 점프 한 것.




```
(gdb) x $rbp-20
0x7fffffffdecc: 0x00000001
(gdb) x $rbp
0x7fffffffdee0: 0xffffdf10
(gdb) x $rsp
0x7fffffffdec0: 0x00000380
```



포기하면 얻는 건 아무것도 없다.

# 6. 피보나치 재귀함 대한 기계어 분석 - (19)

```
   0x00005555555551d0 <+39>:    jmp    0x5555555551fd <recursive_fib+84>
=> 0x00005555555551d2 <+41>:    cmpl   $0x2,-0x14(%rbp)
   0x00005555555551d6 <+45>:    jg     0x5555555551df <recursive_fib+54>
   0x00005555555551d8 <+47>:    mov    $0x1,%eax
   0x00005555555551dd <+52>:    jmp    0x5555555551fd <recursive_fib+84>
   0x00005555555551df <+54>:    mov    -0x14(%rbp),%eax
```

1d2로 넘어와서는 rbp-20(1)의 값을
2의 값과 비교하고

Jb 2의 값은 2보다 크지 않으므로
Mov 1의 값을 eax에 복사 한다.

그리고 1fd의 주소로 점프.

```
(gdb) p/x $eax
$5 = 0x1
```

```
(gdb) x $rbp-20
0x7fffffffdecc: 0x00000002
```

```
int recursive_fib(int num)
{
        if(num <= 0)
        {
                printf("올바른 값을 입력 하세요!\n");
                return -1;
        }
        else if(num < 3)
        {
                return 1;
        }
```

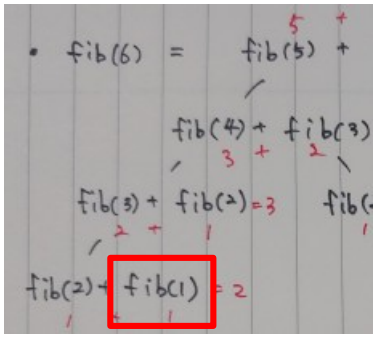
원문의 2 과의 비교에서 2의 값보다 크지 않으므로 return 1 을 하게 된다.

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdfe0 ← rbp |
| | 0x0x7fffffffdfd8 |
| | 0x0x7fffffffdfd0 |
| 1 | 0x0x7fffffffdfcc |
| | 0x0x7fffffffc8 |
| | 0x0x7fffffffdfc0 ← rsp |

# 6. 피보나치 재귀함 대한 기계어 분석 - (20)

```
   0x00005555555551f6 <+77>:    callq  0x5555555551a9 <recursive_fib>
   0x00005555555551fb <+82>:    add    %ebx,%eax
=> 0x00005555555551fd <+84>:    add    $0x18,%rsp
   0x0000555555555201 <+88>:    pop    %rbx
   0x0000555555555202 <+89>:    pop    %rbp
   0x0000555555555203 <+90>:    retq
```

- 1fd로 점프되었으며,
- Rsp에 +28 바이트를 합니다. (d8)
- Pop : 스택으로 부터 값을 뽑아 냅니다.
  - ㄴ rbx값
  - ㄴ rbp값
- retq로 함수가 종료 된다.




```
(gdb) x $rbp
0x7fffffffdf10: 0xffffdf40
(gdb) x $rsp
0x7fffffffdef0: 0x00000000
(gdb) x $rbp-20
0x7fffffffdefc: 0x00000003
```

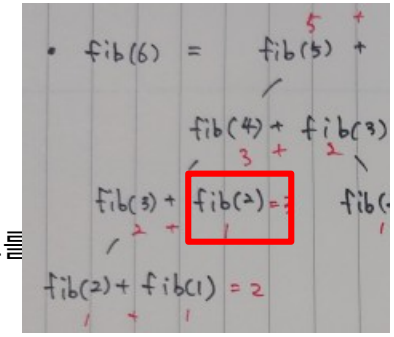
ㄴ 다시 돌아 왔다.

| | |
|---|---|
| 0x0(rbp) | 0x0x7fffffffdf10 ← rbp |
| | 0x0x7fffffffdf08 |
| | 0x0x7fffffffdf00 |
| 3 | 0x0x7fffffffdffc |
| | 0x0x7fffffff8 |
| | 0x0x7fffffffdff0 |

# 6. 피보나치 재귀함 대한 기계어 분석 - (21)

```
=> 0x00005555555551ec <+67>:    mov    %eax,%ebx
   0x00005555555551ee <+69>:    mov    -0x14(%rbp),%eax
   0x00005555555551f1 <+72>:    sub    $0x2,%eax
   0x00005555555551f4 <+75>:    mov    %eax,%edi
   0x00005555555551f6 <+77>:    callq  0x555555551a9 <recursive_fib>
   0x00005555555551fb <+82>:    add    %ebx,%eax
   0x00005555555551fd <+84>:    add    $0x18,%rsp
   0x0000555555555201 <+88>:    pop    %rbx
   0x0000555555555202 <+89>:    pop    %rbp
   0x0000555555555203 <+90>:    retq
```

=> 이제 오른쪽 사진의 함수쪽으로 올라가서 실행할거다.
Eax 값을 ebx로 복사
rbp-20의 값은 eax에 복사후
4-2를 한다.
그것을 이제 다시 재귀 함수 호출 하게 될 것이다.
아래 복귀 주소를 기억한 뒤! (Callq 은 rsp-8에 복귀 주소를
넣어두기 때문)




```
(gdb) x $rbp-20
0x7fffffffdf2c: 0x00000004
(gdb) x $rbp
0x7fffffffdf40: 0xffffdf70
(gdb) x $rsp
0x7fffffffdf20: 0x00000000
```

ㄴ fib(4)쪽으로 돌아왔다.
n-2항을 실행하기 위해서다!

```
(gdb) x $rsp-8
0x7fffffffdf10: 0xffffdf40
```

```
        }
        else if(num < 3)
        {
                return 1;
        }
        else
        {
                return recursive_fib(num-1) + recursive_fib(num-2);
        }
}
```

| | |
|---|---|
| df70 | 0x0x7fffffffdf40 ← rbp |
| | 0x0x7fffffffdf38 |
| | 0x0x7fffffffdf30 |
| 4 -2 | 0x0x7fffffffdf2c |
| | 0x0x7fffffff28 |
| | 0x0x7fffffffdf20 ← rsp |
| df40(복귀) | 0x0x7fffffffdf10 |

# 6. 피보나치 재귀함 대한 기계어 분석 - (22)

```
   0x00005555555551a9 <+0>:     endbr64
   0x00005555555551ad <+4>:     push   %rbp
   0x00005555555551ae <+5>:     mov    %rsp,%rbp
   0x00005555555551b1 <+8>:     push   %rbx
=> 0x00005555555551b2 <+9>:     sub    $0x18,%rsp
   0x00005555555551b6 <+13>:    mov    %edi,-0x14(%rbp)
   0x00005555555551b9 <+16>:    cmpl   $0x0,-0x14(%rbp)
   0x00005555555551bd <+20>:    jg     0x5555555551d2 <recursive_fib+41>
   0x00005555555551bf <+22>:    lea    0xe42(%rip),%rdi        # 0x555555556008
   0x00005555555551c6 <+29>:    callq  0x555555555080 <puts@plt>
   0x00005555555551cb <+34>:    mov    $0xffffffff,%eax
   0x00005555555551d0 <+39>:    jmp    0x5555555551fd <recursive_fib+84>
   0x00005555555551d2 <+41>:    cmpl   $0x2,-0x14(%rbp)
   0x00005555555551d6 <+45>:    jg     0x5555555551df <recursive_fib+54>
```

다시 시작된 rbp..

- rbp push
- rsp , rbp 경계선 없어짐
- rbx push
- 32 바이트 아래까지 rsp 생성
- edi(2) 값을 rpb-20에 배치
- **cmpl 비교, 0값과 rpb-20 (1) 값과 비교**
- **jg (결과가 크면 점프) 1d2로 점프**
  **→ 앞의 비교 값이 0보다 크므로 다음으로 점프**

```
int recursive_fib(int num)
{
        if(num <= 0)
        {
                printf("올바른 값을 입력 하세요!\n");
                return -1;
        }
        else if(num < 3)
        {
                return 1;
        }
```

| | |
|---|---|
| df40 | 0x0x7fffffffdf10 ← rbp |
| | 0x0x7fffffffdf08 |
| | 0x0x7fffffffdf00 |
| 4 -2 | 0x0x7fffffffdffc |
| | 0x0x7fffffffdff8 |
| | 0x0x7fffffffdff0 ← rsp |

원문의 0 과의 비교에서 0 보다 큰 값이 들어 왔으니 아래 조건문으로 점프 한 것.

```
(gdb) x $rbp-20
0x7fffffffdecc: 0x00000001
(gdb) x $rbp
0x7fffffffdee0: 0xffffdf10
(gdb) x $rsp
0x7fffffffdec0: 0x00000380
```

# 6. 피보나치 재귀함 대한 기계어 분석 - (23)

```
   0x00005555555551d0 <+39>:    jmp    0x5555555551fd <recursive_fib+84>
=> 0x00005555555551d2 <+41>:    cmpl   $0x2,-0x14(%rbp)
   0x00005555555551d6 <+45>:    jg     0x5555555551df <recursive_fib+54>
   0x00005555555551d8 <+47>:    mov    $0x1,%eax
   0x00005555555551dd <+52>:    jmp    0x5555555551fd <recursive_fib+84>
   0x00005555555551df <+54>:    mov    -0x14(%rbp),%eax
```

1d2로 넘어와서는 rbp-20(2)의 값을
2의 값과 비교하고

Jb 2의 값은 2보다 크지 않으므로
Mov 1의 값을 eax에 복사 한다.

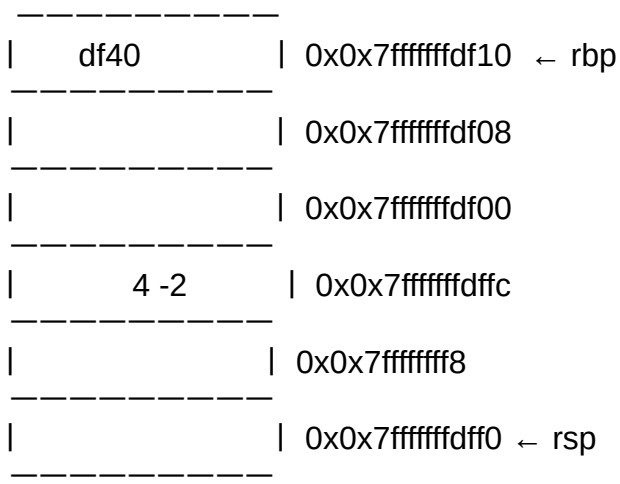
그리고 1fd의 주소로 점프.

```
(gdb) p/x $eax
$5 = 0x1
```

```
(gdb) x $rbp-20
0x7fffffffdecc: 0x00000002
```

```
int recursive_fib(int num)
{
        if(num <= 0)
        {
                printf("올바른 값을 입력 하세요!\n");
                return -1;
        }
        else if(num < 3)
        {
                return 1;
        }
```

원문의 2 과의 비교에서 2의 값보다 크지 않으므로 return 1 을 하게 된다.

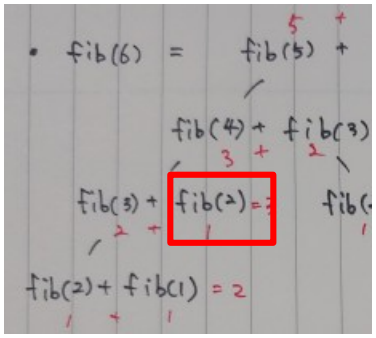


# 6. 피보나치 재귀함 대한 기계어 분석 - (24)

```
   0x00005555555551fb <+82>:    add    %ebx,%eax
=> 0x00005555555551fd <+84>:    add    $0x18,%rsp
   0x0000555555555201 <+88>:    pop    %rbx
   0x0000555555555202 <+89>:    pop    %rbp
   0x0000555555555203 <+90>:    retq
```

```
(gdb) x $ebx
0x2:    Cannot
(gdb) x $eax
0x1:    Cannot
```

Ebx(2) + eax(1)을 더한다.
Rsp+24 바이트르 한다. (F0 → 08)
Rbx 값 추출
rbp값 추출(복귀주소)
Retq 복귀!




```
(gdb) x $rbp
0x7fffffffdf40: 0xffffdf70
(gdb) x $rsp
0x7fffffffdf20: 0x00000000
(gdb) x $rbp-20
0x7fffffffdf2c: 0x00000004
```

ㄴ 이렇게 rbp 주소가 40인 상위단의 스택으로 돌아옴. (여기서 rbp가 70주소를 가지므로 거슬러 올라 가게 되면 70이 되겠지)

```
=> 0x00005555555551fb <+82>:    add    %ebx,%eax
   0x00005555555551fd <+84>:    add    $0x18,%rsp
   0x0000555555555201 <+88>:    pop    %rbx
   0x0000555555555202 <+89>:    pop    %rbp
   0x0000555555555203 <+90>:    retq
```

: ebx(2) + eax(1)
Rsp+24 바이트
Rbx 값 추출
Rbp 추출
복귀!

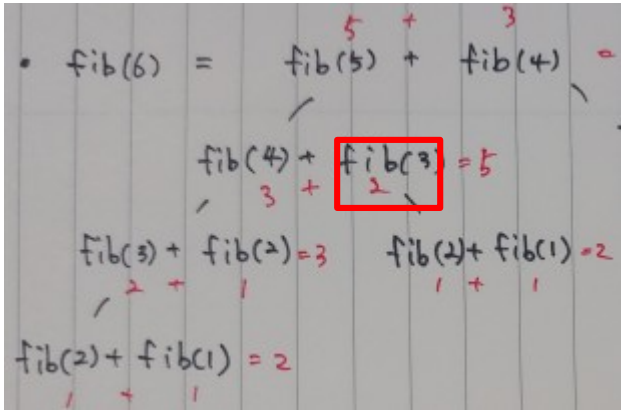
```
(gdb) x $ebx
0x2:    Canno
(gdb) x $eax
0x1:    Canno
```

| | |
|---|---|
| df40 | 0x0x7fffffffdf10 ← rbp |
| | 0x0x7fffffffdf08 ← rsp |
| | 0x0x7fffffffdf00 |
| 4 -2 | 0x0x7fffffffdffc |
| | 0x0x7ffffffff8 |
| | 0x0x7fffffffdff0 |

# 6. 피보나치 재귀함 대한 기계어 분석 - (25)

```
(gdb) x $rbp
0x7fffffffdf70: 0xffffdf90
(gdb) x $rsp
0x7fffffffdf50: 0xffffdf76
(gdb) x $rbp-20
0x7fffffffdf5c: 0x00000005
```

좋다. 이번에 역시 그 다음 위의 스택인 70으로 돌아 왔다..
슬슬 감이 잡힌다..




```
=> 0x000055555555551ec <+67>:    mov    %eax,%ebx
   0x000055555555551ee <+69>:    mov    -0x14(%rbp),%eax
   0x000055555555551f1 <+72>:    sub    $0x2,%eax
   0x000055555555551f4 <+75>:    mov    %eax,%edi
   0x000055555555551f6 <+77>:    callq  0x5555555551a9 <recursive_fib>
   0x000055555555551fb <+82>:    add    %ebx,%eax
   0x000055555555551fd <+84>:    add    $0x18,%rsp
   0x0000555555555201 <+88>:    pop    %rbx
   0x0000555555555202 <+89>:    pop    %rbp
   0x0000555555555203 <+90>:    retq
```

Eax(3) 값을 ebx에 배치
Rbp-20 값(5)를 eax에 배치
5에서 2를 뺀다 . (n-2)항을 계산할 차례이기 때문이지...
그리고 fib(3)이 되고 이 것을 또
Callq 이므로 복귀주소를 rsp-8에 넣고 jump 한다.

```
(gdb) x $rbp
0x7fffffffdf40: 0xffffdf70
(gdb) x $rsp
0x7fffffffdf20: 0x00000000
```

ㄴfib(3)들어가서 다시 생긴 스택인데
여기 보면 또 rbp에 복귀 주소가 70인걸 볼수 있죠!

```
(gdb) x $eax
0x3:    Cannc
```

```
=> 0x000055555555551f6 <+77>:
   0x000055555555551fb <+82>:
   0x000055555555551fd <+84>:
   0x0000555555555201 <+88>:
   0x0000555555555202 <+89>:
   0x0000555555555203 <+90>:
End of assembler dump.
(gdb) x $rsp
0x7fffffffdf50: 0xffffdf76
(gdb) si
recursive_fib (num=4) at fib_
62      {
(gdb) x $rsp
0x7fffffffdf48: 0x555551fb
```

| | |
|---|---|
| df90 | 0x0x7fffffffdf70 ← rbp |
| | 0x0x7fffffffdf68 |
| | 0x0x7fffffffdf60 |
| 5 -2 | 0x0x7fffffffdf5c |
| | 0x0x7fffffff58 |
| | 0x0x7fffffffdf50 ← rsp |

**결론...**

**fib(5)에서 n-1항을 차근차근 실행하고 마지막 return 까지 실행 후 다시 스택을 올라 오면서 n-2항을 실행하게 되는데, 그 와중에 또 새로운 fib가 생기면 n-1부터 한다!!**