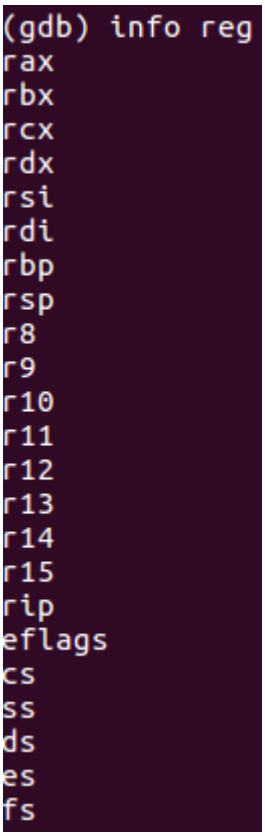
<x86 어셈블리어 분석>

[목적]

Intel x86에서 사용하는 레지스터와 어셈블리어 연산을 분석하고, 함수 호출 시 Stack 변화를 그림을 그려 살펴본다.

[x86 범용 레지스터]



rax : 리턴 값 저장

rcx : 특정 작업 반복 횟수 저장

rbp : 스택의 기준점
rsp : 현재 스택의 최상위점

rip : 다음에 실행할 명령어의 주소를 가리킴
eflags : 상태 플래그, 제어 플래그. 시스템 플래그

##
rax : 64bit
eax : 32bit
ax : 16bit
ah, al : 8bit

[x86 어셈블리어]

- push A : 현재 sp를 기준으로 A를 스택에 저장
- mov A B : B를 A에 복사

##

- movl : 4byte를 복사
- movq : 8byte를 복사

- sub : 뺄셈
- call A : push + jmp, 복귀 주소를 stack에 저장, A로 jmp
- jmp A : A로 이동
- add : 덧셈
- pop A : 현재 sp의 값을 A에 저장
- retq : pop rip, 현재 스택의 최상위에 저장되어 있는 값을 rip(다음 실행할 명령어 주소를 담는 레지스터)에 저장.

[gdb 로 디버깅 순서]

- gdb를 이용하여 디버깅을 하는 순서

(1) .c 파일을 -g 옵션을 주어서 gcc로 컴파일한다.
(.c 파일의 이름을 adder.c 로 설정)

```
roro@roro-Lenovo-Y520-15IKBN:~/git_repo/workspace/x86_asm_note$ gcc -g adder.c
```

(2) gdb로 실행파일을 연다.

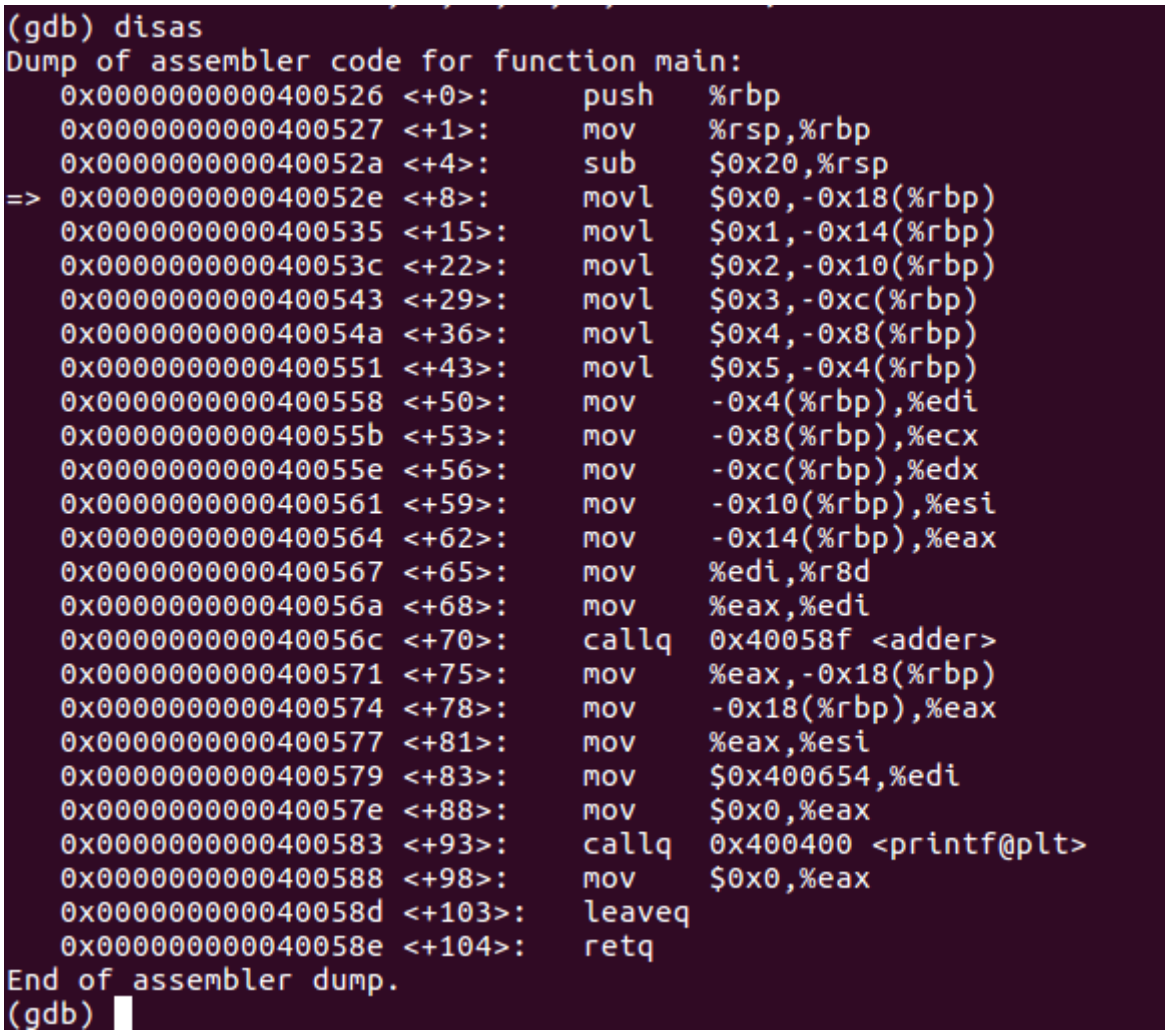
```
roro@roro-Lenovo-Y520-15IKBN:~/git_repo/workspace/x86_asm_note$ gdb a.out
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from a.out...done.
(gdb)
```

(3) main에 break point를 걸고(b main), r 로 프로그램을 실행시킨다.

```
(gdb) b main
Breakpoint 1 at 0x40052e: file adder.c, line 8.
(gdb) r
Starting program: /home/roro/git_repo/workspace/x86_asm_note/a.out

Breakpoint 1, main () at adder.c:8
8               int a, b, c, d, e, res = 0;
(gdb)
```

(4) disas 로 어셈블리 명령어를 확인한다. (=> 표시가 되어 있는 곳이 break point가 걸린 곳이다)

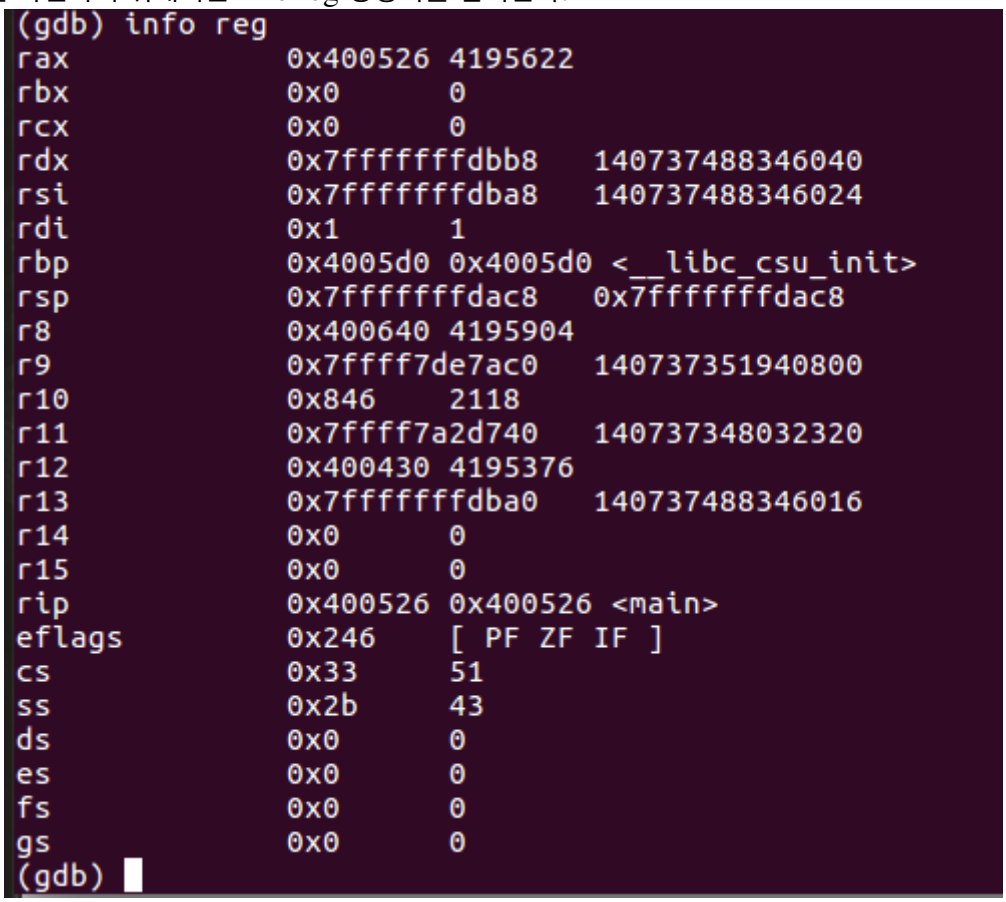


(5) break point를 변경하고 싶을 경우에는 명령어 주소를 복사하여 b *<복사한 명령어> 를 입력한 후 r 명령어로 다시 실행한다.
- 예를 들어 push %rbp로 break point를 변경하고 할 경우에는 0x400526을 복사하여 b *0x400526을 입력한다.

```
(gdb) b *0x0000000000400526
Note: breakpoint 2 also set at pc 0x400526.
Breakpoint 3 at 0x400526: file adder.c, line 7.
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/roro/git_repo/workspace/x86_asm_note/a.out

Breakpoint 2, main () at adder.c:7
7       {
(gdb) disas
Dump of assembler code for function main:
=> 0x0000000000400526 <+0>:     push   %rbp
   0x0000000000400527 <+1>:     mov    %rsp,%rbp
   0x000000000040052a <+4>:     sub    $0x20,%rsp
   0x000000000040052e <+8>:     movl   $0x0,-0x18(%rbp)
   0x0000000000400535 <+15>:    movl   $0x1,-0x14(%rbp)
   0x000000000040053c <+22>:    movl   $0x2,-0x10(%rbp)
   0x0000000000400543 <+29>:    movl   $0x3,-0xc(%rbp)
   0x000000000040054a <+36>:    movl   $0x4,-0x8(%rbp)
   0x0000000000400551 <+43>:    movl   $0x5,-0x4(%rbp)
   0x0000000000400558 <+50>:    mov    -0x4(%rbp),%edi
   0x000000000040055b <+53>:    mov    -0x8(%rbp),%ecx
   0x000000000040055e <+56>:    mov    -0xc(%rbp),%edx
   0x0000000000400561 <+59>:    mov    -0x10(%rbp),%esi
   0x0000000000400564 <+62>:    mov    -0x14(%rbp),%eax
   0x0000000000400567 <+65>:    mov    %edi,%r8d
   0x000000000040056a <+68>:    mov    %eax,%edi
   0x000000000040056c <+70>:    callq  0x40058f <adder>
   0x0000000000400571 <+75>:    mov    %eax,-0x18(%rbp)
   0x0000000000400574 <+78>:    mov    -0x18(%rbp),%eax
   0x0000000000400577 <+81>:    mov    %eax,%esi
   0x0000000000400579 <+83>:    mov    $0x400654,%edi
   0x000000000040057e <+88>:    mov    $0x0,%eax
   0x0000000000400583 <+93>:    callq  0x400400 <printf@plt>
   0x0000000000400588 <+98>:    mov    $0x0,%eax
   0x000000000040058d <+103>:   leaveq
   0x000000000040058e <+104>:   retq
End of assembler dump.
(gdb)
```

(5) si 명령어로 어셈블리 명령어를 하나씩 실행 시킬 수 있다.
- 레지스터를 확인하기 위해서는 info reg 명령어를 입력한다.



[덧셈 프로그램 디버깅]

- main에서 선언한 5개의 변수를 인자로 받는 adder 함수를 구현한다.
- adder 함수에서 인자로 받은 5개의 함수의 합을 return한다.

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

int adder(int a, int b, int c, int d, int e);

int main(void)
{
    int a, b, c, d, e, res = 0;
    a = 1, b = 2, c = 3, d = 4, e = 5;

    res = adder(a, b, c, d, e);
    printf("res = %d\n", res);

    return 0;
}

int adder(int a, int b, int c, int d, int e)
{
    int sum = 0;

    sum = a + b + c + d + e;

    return sum;
}
```

- 실행 결과

```
roro@roro-Lenovo-Y520-15IKBN:~/git_repo/workspace/x86_asm_note$ ./a.out
res = 15
```

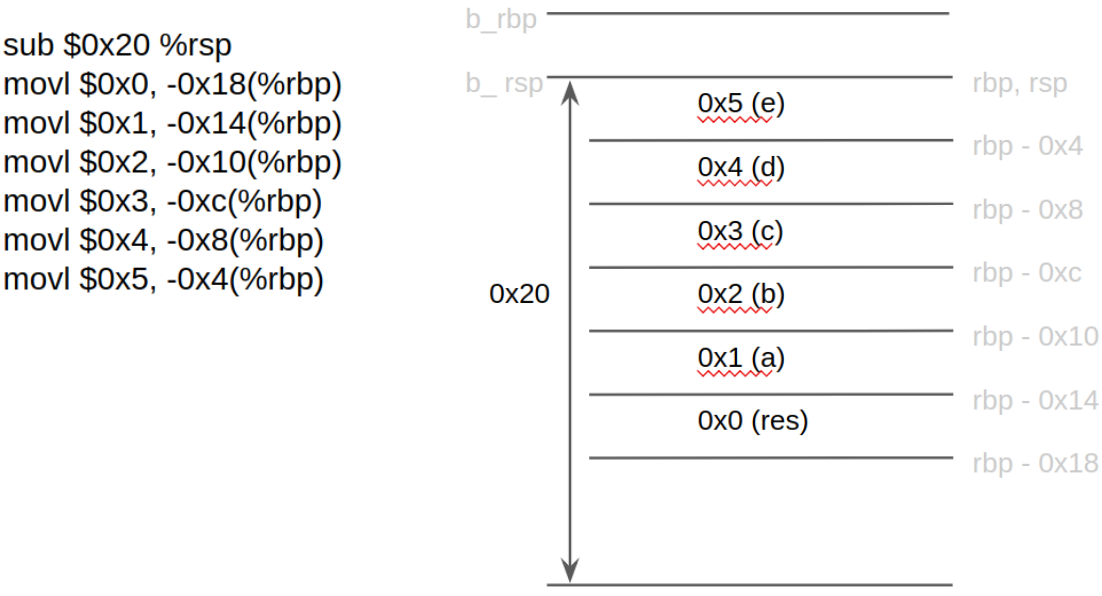
[덧셈 프로그램 어셈블리어]

```
Dump of assembler code for function main:
=> 0x0000000000400526 <+0>:     push   %rbp
   0x0000000000400527 <+1>:     mov    %rsp,%rbp
   0x000000000040052a <+4>:     sub    $0x20,%rsp
   0x000000000040052e <+8>:     movl   $0x0,-0x18(%rbp)
   0x0000000000400535 <+15>:    movl   $0x1,-0x14(%rbp)
   0x000000000040053c <+22>:    movl   $0x2,-0x10(%rbp)
   0x0000000000400543 <+29>:    movl   $0x3,-0xc(%rbp)
   0x000000000040054a <+36>:    movl   $0x4,-0x8(%rbp)
   0x0000000000400551 <+43>:    movl   $0x5,-0x4(%rbp)
   0x0000000000400558 <+50>:    mov    -0x4(%rbp),%edi
   0x000000000040055b <+53>:    mov    -0x8(%rbp),%ecx
   0x000000000040055e <+56>:    mov    -0xc(%rbp),%edx
   0x0000000000400561 <+59>:    mov    -0x10(%rbp),%esi
   0x0000000000400564 <+62>:    mov    -0x14(%rbp),%eax
   0x0000000000400567 <+65>:    mov    %edi,%r8d
   0x000000000040056a <+68>:    mov    %eax,%edi
   0x000000000040056c <+70>:    callq  0x40058f <adder>
   0x0000000000400571 <+75>:    mov    %eax,-0x18(%rbp)
   0x0000000000400574 <+78>:    mov    -0x18(%rbp),%eax
   0x0000000000400577 <+81>:    mov    %eax,%esi
   0x0000000000400579 <+83>:    mov    $0x400654,%edi
   0x000000000040057e <+88>:    mov    $0x0,%eax
   0x0000000000400583 <+93>:    callq  0x400400 <printf@plt>
   0x0000000000400588 <+98>:    mov    $0x0,%eax
   0x000000000040058d <+103>:   leaveq
   0x000000000040058e <+104>:   retq
End of assembler dump.
```

(1) 스택 프레임 생성
- push, mov를 통해 main 함수의 스택을 생성한다.

(2) 변수 저장공간 생성, 변수를 스택에 저장
- sub $0x20 %rsp : rsp에 저장되어 있는 값에서 20을 빼서 rsp에 저장
→ 변수를 저장할 공간을 stack에 생성한다.

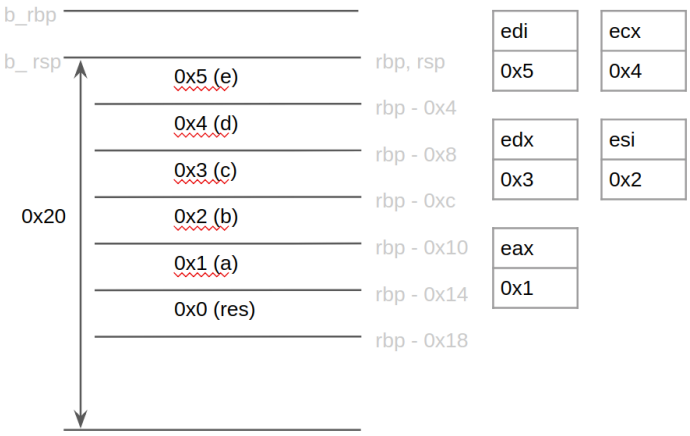
- movl 명령어 : adder.c 프로그램에서 초기화 된 변수들(res, a, b, c, d, e)를 rbp를 기준으로 스택에 저장한다.



함수 호출 규약(Calling Convention)
- Intel x86 : 함수 호출 시 인자를 stack(memory)에 저장
- ARM : 함수 호출 시 인자 4개까지는 register 0~3에 저장, 5개부터 stack(memory)에 저장

(3) 레지스터 연산을 위해 stack에서 register로 값 복사

```
mov -0x4(%rbp), %edi
mov -0x8(%rbp), %ecx
mov -0xc(%rbp), %edx
mov -0x10(%rbp), %esi
mov -0x14(%rbp), %eax
```



- 어셈블리 명령어에 있는 edi, ecx 등의 e-인 레지스터는 32bit 레지스터를 의미한다.

Register	Accumulator		
64-bit	RAX		
32-bit		EAX	
16-bit		AX	
8-bit		AH	AL

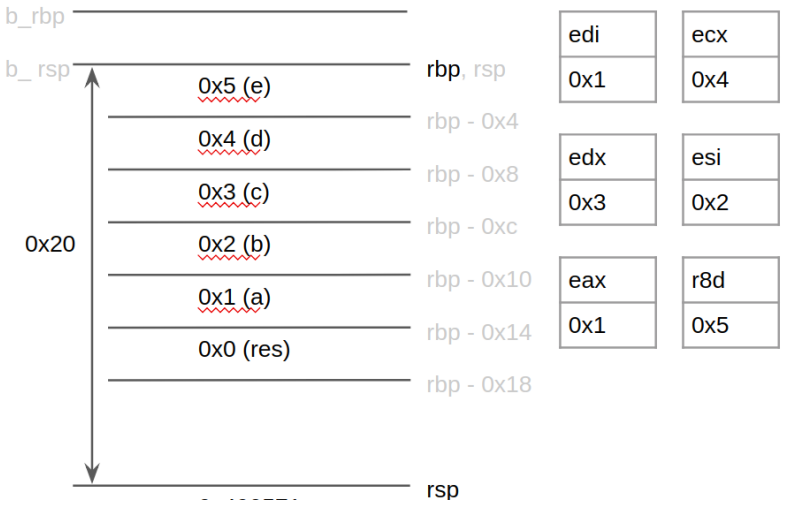
- mov 전/후 의 레지스터 값

```
(gdb) info reg
rax            0x400526 4195622
rbx            0x0      0
rcx            0x0      0
rdx            0x7fffffffdbb8
rsi            0x7fffffffdba8
rdi            0x1      1
rbp            0x7fffffffdac0
rsp            0x7fffffffdaa0
r8             0x400640 4195904
r9             0x7ffff7de7ac0
r10            0x846    2118
r11            0x7ffff7a2d740
r12            0x400430 4195376
r13            0x7fffffffdba0
r14            0x0      0
r15            0x0      0
rip            0x400558 0x400558
eflags         0x206    [ PF IF
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
```

```
(gdb) info reg
rax            0x1      1
rbx            0x0      0
rcx            0x4      4
rdx            0x3      3
rsi            0x2      2
rdi            0x5      5
rbp            0x7fffffffdac0
rsp            0x7fffffffdaa0
r8             0x400640 4195904
r9             0x7ffff7de7ac0
r10            0x846    2118
r11            0x7ffff7a2d740
r12            0x400430 4195376
r13            0x7fffffffdba0
r14            0x0      0
r15            0x0      0
rip            0x400567 0x400567
eflags         0x206    [ PF IF
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
```

(4) 레지스터 값 이동

mov %edi, %r8d
mov %eax, %edi

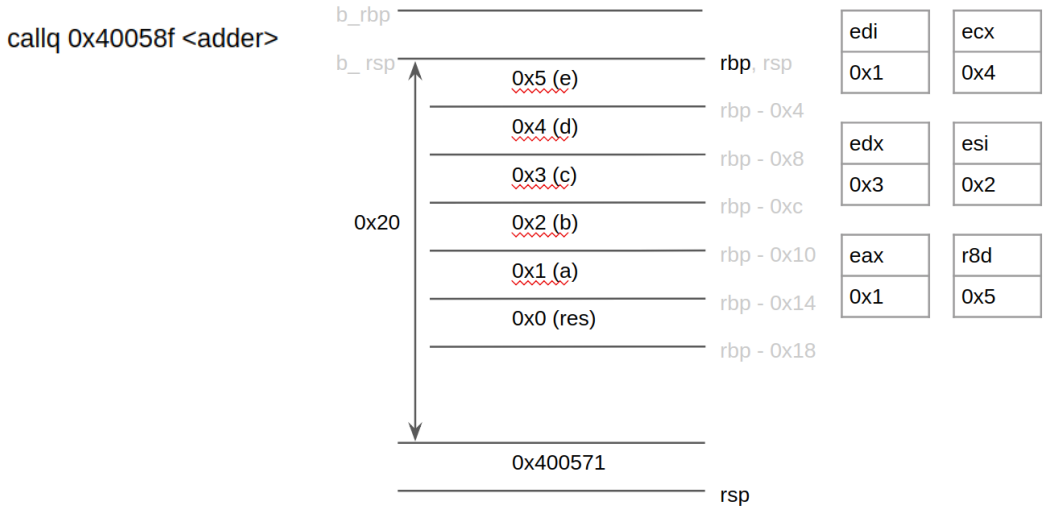


- mov 명령어 전/후

```
(gdb) info reg
rax            0x1      1
rbx            0x0      0
rcx            0x4      4
rdx            0x3      3
rsi            0x2      2
rdi            0x5      5
rbp            0x7fffffffdac0   0
rsp            0x7fffffffdaa0   0
r8             0x400640 4195904
r9             0x7ffff7de7ac0   1
r10            0x846    2118
r11            0x7ffff7a2d740   1
r12            0x400430 4195376
r13            0x7fffffffdba0   1
r14            0x0      0
r15            0x0      0
rip            0x400567 0x400567
eflags         0x206    [ PF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
```

```
(gdb) info reg
rax            0x1      1
rbx            0x0      0
rcx            0x4      4
rdx            0x3      3
rsi            0x2      2
rdi            0x1      1
rbp            0x7fffffffdac0   0
rsp            0x7fffffffdaa0   0
r8             0x5      5
r9             0x7ffff7de7ac0   1
r10            0x846    2118
r11            0x7ffff7a2d740   1
r12            0x400430 4195376
r13            0x7fffffffdba0   1
r14            0x0      0
r15            0x0      0
rip            0x40056c 0x40056c
eflags         0x206    [ PF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
```

(5) 다음에 실행될 명령어를 stack에 저장하고, 0x40058f에 저장된 <adder> 함수를 호출한다.
- <adder> 함수가 끝나고 return 될 0x40056c callq 다음 명령어인 0x400571 mov를 stack에 저장한다.




```
0x000000000040056c <+70>:    callq  0x40058f <adder>
0x0000000000400571 <+75>:    mov    %eax,-0x18(%rbp)
```

- rsp가 가리키고 있는 주소인 0x7fffffffda98에 저장된 값이 0x400571임을 확인.

```
$4 = 0x7fffffffda98
(gdb) p/x *0x7fffffffda98
$5 = 0x400571
```

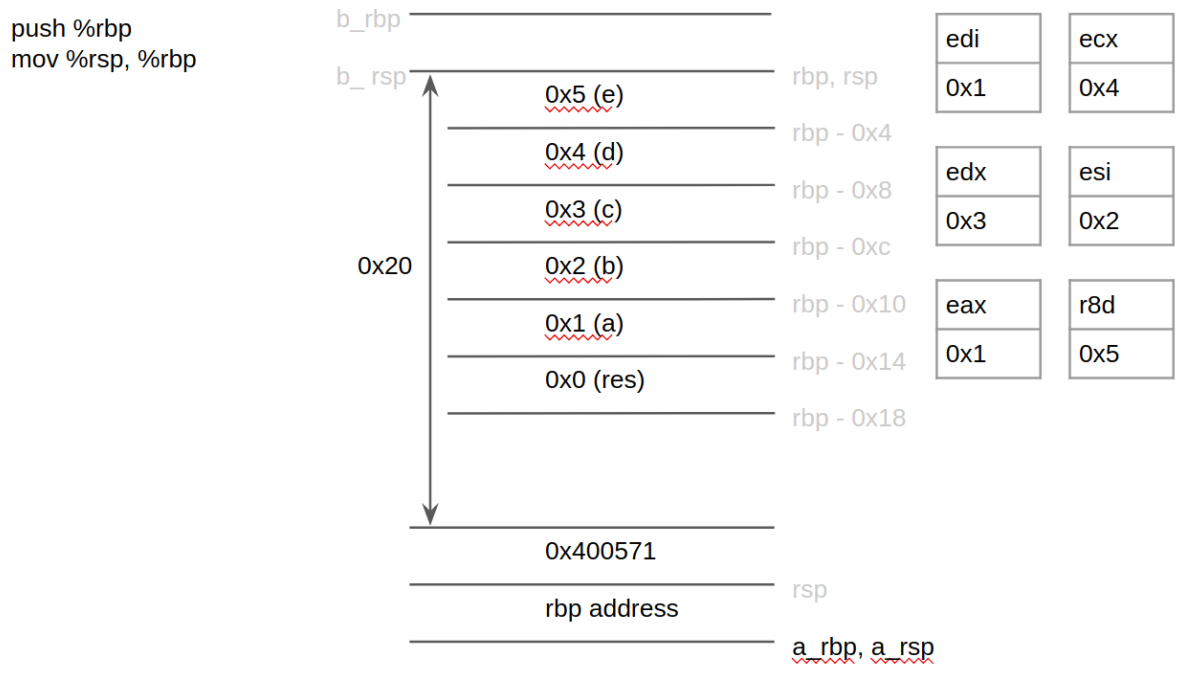
- 다음에 실행될 명령어를 가리키는 rip 레지스터는 <adder> 함수가 저장된 0x40058f를 가리킨다.

```
(gdb) info reg
rax            0x1      1
rbx            0x0      0
rcx            0x4      4
rdx            0x3      3
rsi            0x2      2
rdi            0x1      1
rbp            0x7fffffffdac0
rsp            0x7fffffffda98
r8             0x5      5
r9             0x7ffff7de7ac0
r10            0x846    2118
r11            0x7ffff7a2d740
r12            0x400430 4195376
r13            0x7fffffffdba0
r14            0x0      0
r15            0x0      0
rip            0x40058f 0x40058f
eflags         0x206    [ PF IF
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
```

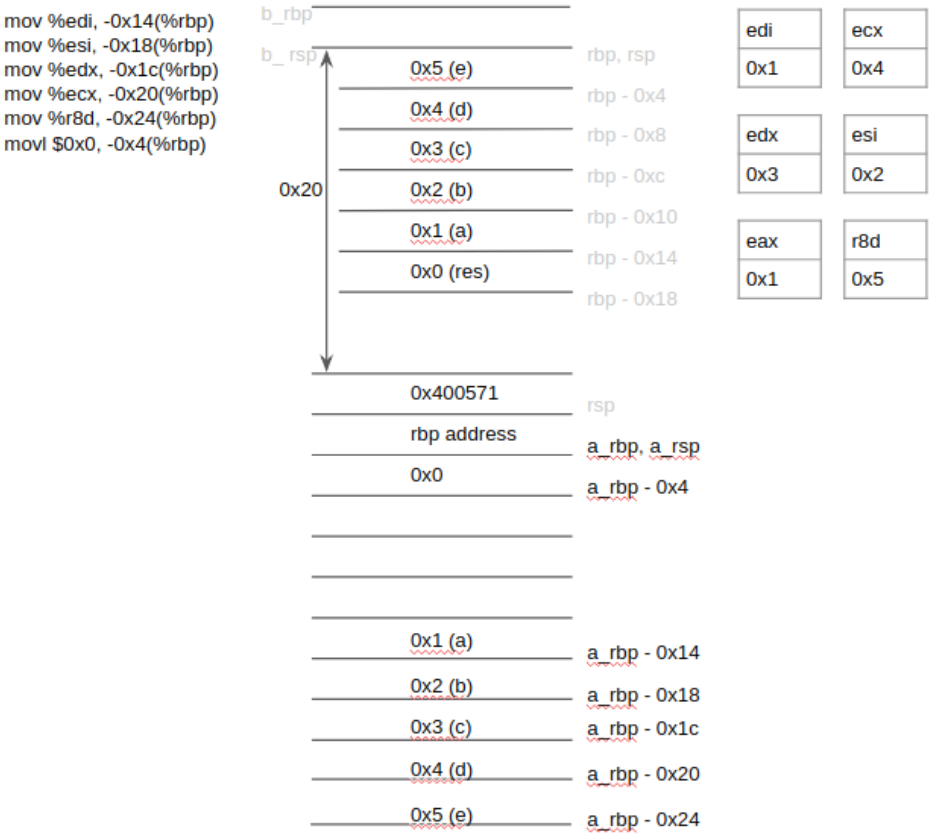
(6) adder 함수를 실행한다.

```
(gdb) disas
Dump of assembler code for function adder:
=> 0x000000000040058f <+0>:     push   %rbp
   0x0000000000400590 <+1>:     mov    %rsp,%rbp
   0x0000000000400593 <+4>:     mov    %edi,-0x14(%rbp)
   0x0000000000400596 <+7>:     mov    %esi,-0x18(%rbp)
   0x0000000000400599 <+10>:    mov    %edx,-0x1c(%rbp)
   0x000000000040059c <+13>:    mov    %ecx,-0x20(%rbp)
   0x000000000040059f <+16>:    mov    %r8d,-0x24(%rbp)
   0x00000000004005a3 <+20>:    movl   $0x0,-0x4(%rbp)
   0x00000000004005aa <+27>:    mov    -0x14(%rbp),%edx
   0x00000000004005ad <+30>:    mov    -0x18(%rbp),%eax
   0x00000000004005b0 <+33>:    add    %eax,%edx
   0x00000000004005b2 <+35>:    mov    -0x1c(%rbp),%eax
   0x00000000004005b5 <+38>:    add    %eax,%edx
   0x00000000004005b7 <+40>:    mov    -0x20(%rbp),%eax
   0x00000000004005ba <+43>:    add    %eax,%edx
   0x00000000004005bc <+45>:    mov    -0x24(%rbp),%eax
   0x00000000004005bf <+48>:    add    %edx,%eax
   0x00000000004005c1 <+50>:    mov    %eax,-0x4(%rbp)
   0x00000000004005c4 <+53>:    mov    -0x4(%rbp),%eax
   0x00000000004005c7 <+56>:    pop    %rbp
   0x00000000004005c8 <+57>:    retq
```

- adder 함수 스택을 형성(push, mov)

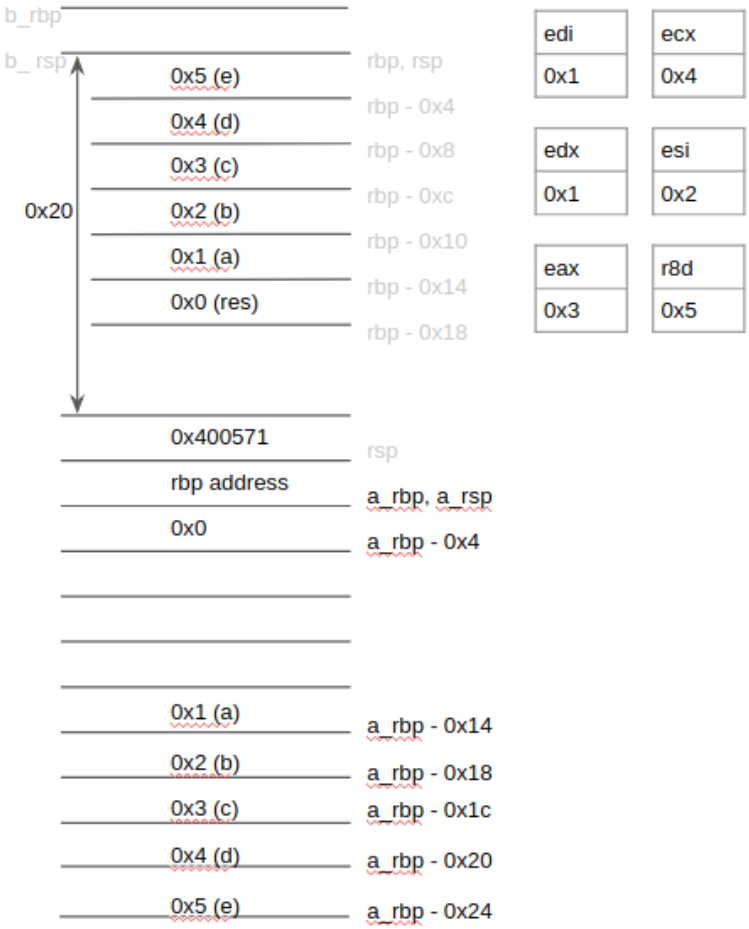


- 덧셈 연산 (sum = a + b + c + d + e)



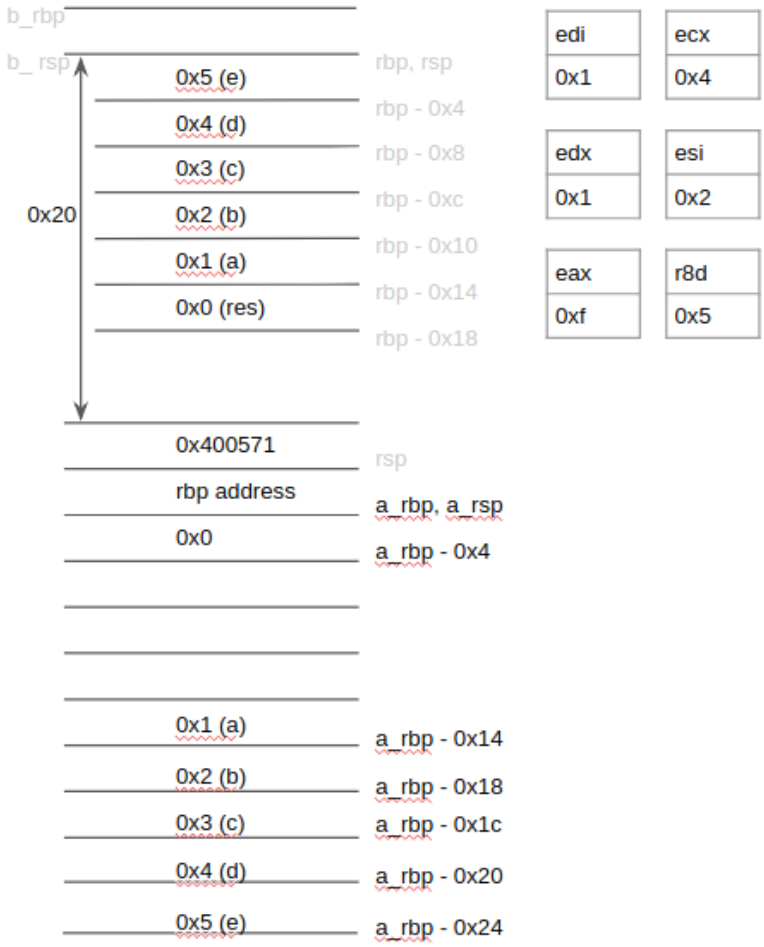
mov -0x14(%rbp), %edx
mov -0x18(%rbp), %eax
add %eax, %edx

=> a + b를 eax에 저장



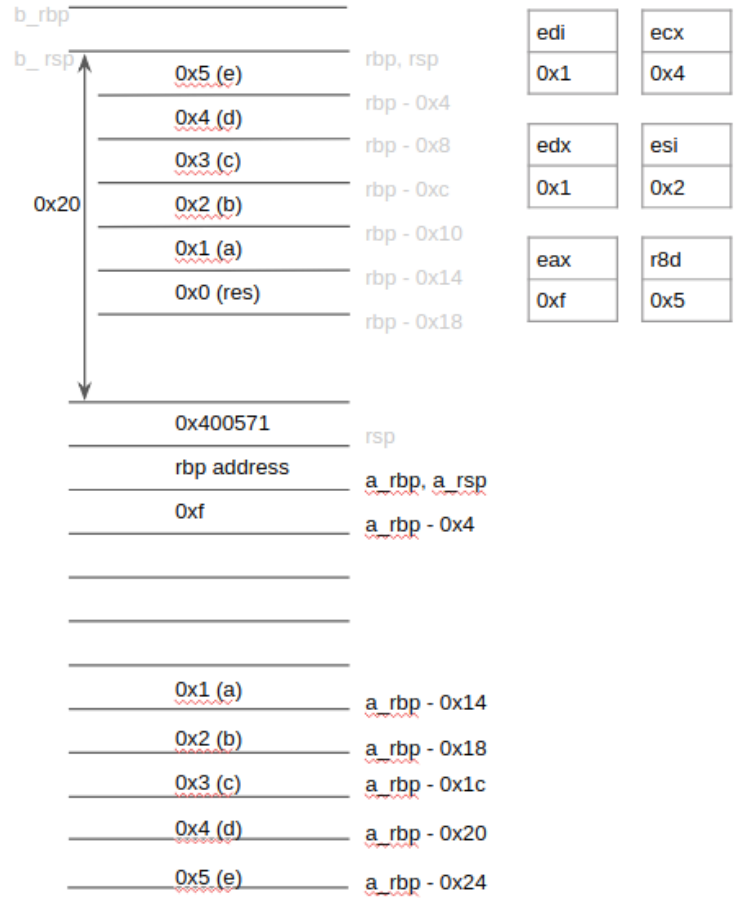
```
mov -0x1c(%rbp), %eax
add %eax, %edx
mov -0x20(%rbp), %eax
add %eax, %edx
mov -0x24(%rbp), %eax)
add %eax, %edx
```

=> a + b + c + d + e를 eax에 저장



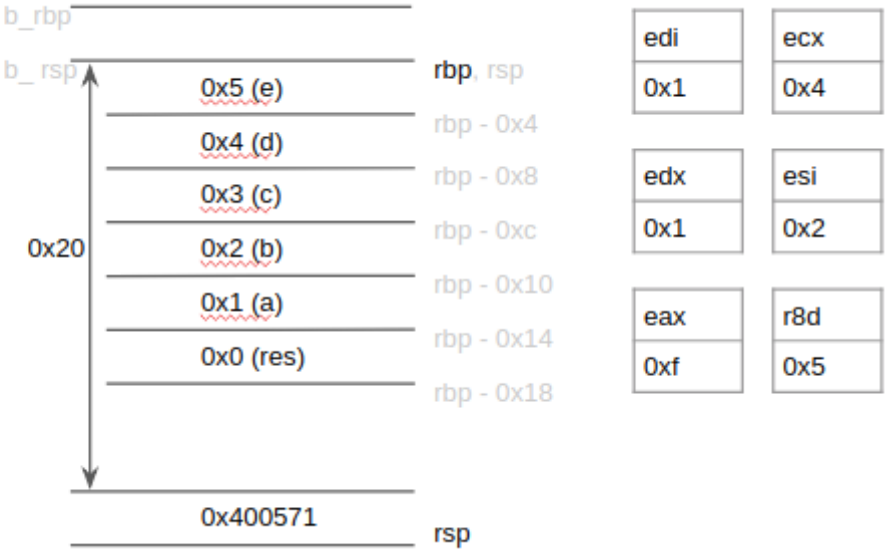

```
mov %eax, -0x4(%rbp)
mov -0x4(%rbp), %eax
```

=> a + b + c + d + e를 스택에 저장

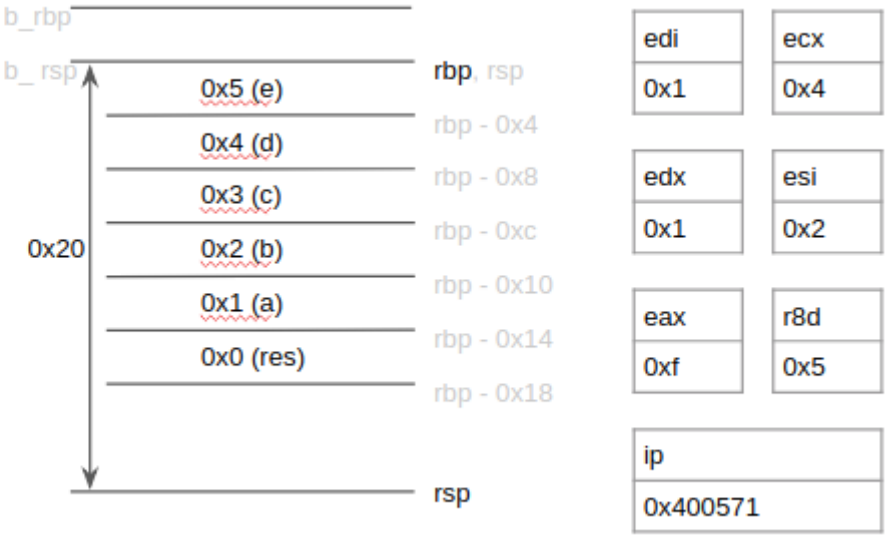


- adder 함수 스택 해제하고 main 함수로 복귀(pop, retq)
: retq 명령을 통해 0x400571을 ip에 pop한다.

pop %rbp

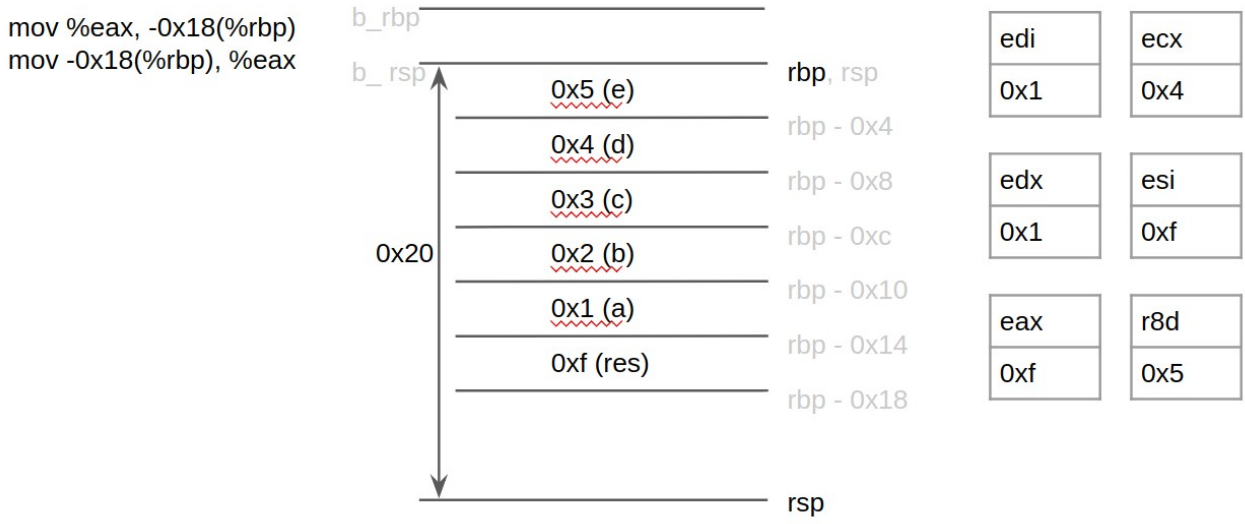


retq



(7) adder 함수에서 리턴 한 후 0x400571부터 실행된다.

```
Dump of assembler code for function main:
=> 0x0000000000400526 <+0>:     push   %rbp
   0x0000000000400527 <+1>:     mov    %rsp,%rbp
   0x000000000040052a <+4>:     sub    $0x20,%rsp
   0x000000000040052e <+8>:     movl   $0x0,-0x18(%rbp)
   0x0000000000400535 <+15>:    movl   $0x1,-0x14(%rbp)
   0x000000000040053c <+22>:    movl   $0x2,-0x10(%rbp)
   0x0000000000400543 <+29>:    movl   $0x3,-0xc(%rbp)
   0x000000000040054a <+36>:    movl   $0x4,-0x8(%rbp)
   0x0000000000400551 <+43>:    movl   $0x5,-0x4(%rbp)
   0x0000000000400558 <+50>:    mov    -0x4(%rbp),%edi
   0x000000000040055b <+53>:    mov    -0x8(%rbp),%ecx
   0x000000000040055e <+56>:    mov    -0xc(%rbp),%edx
   0x0000000000400561 <+59>:    mov    -0x10(%rbp),%esi
   0x0000000000400564 <+62>:    mov    -0x14(%rbp),%eax
   0x0000000000400567 <+65>:    mov    %edi,%r8d
   0x000000000040056a <+68>:    mov    %eax,%edi
   0x000000000040056c <+70>:    callq  0x40058f <adder>
   0x0000000000400571 <+75>:    mov    %eax,-0x18(%rbp)
   0x0000000000400574 <+78>:    mov    -0x18(%rbp),%eax
   0x0000000000400577 <+81>:    mov    %eax,%esi
   0x0000000000400579 <+83>:    mov    $0x400654,%edi
   0x000000000040057e <+88>:    mov    $0x0,%eax
   0x0000000000400583 <+93>:    callq  0x400400 <printf@plt>
   0x0000000000400588 <+98>:    mov    $0x0,%eax
   0x000000000040058d <+103>:   leaveq
   0x000000000040058e <+104>:   retq
End of assembler dump.
```



(8) printf 함수 실행해서 res 출력 (생략)