EDDI
Electronic Design
Development Institute

에디로봇아카데미

임베디드 마스터 Lv1 과정

제 4기

2022. 09. 16

진동민

학습목표 & 2회차 날짜

EDDI
Electronic Design Development Institute

학습목표

- gdb 내에서의 명령어 배우기
- 레지스터 용도 배우기
- 어셈블리어 명령어 배우기
- 어셈블리어를 그림으로 그리며 실제 메모리 분석하는 법 배우기

수업 날짜

2022-08-27 (토) 오후 6시~9시

목차

gcc 소개 및 사용법

EDDI
Electronic Design Development Institute

1) gcc 란

C언어로 작성된 프로그램을 컴파일하는 컴파일러이다. 컴파일하면 기계어로 변환하여 실행파일을 생성한다.

2) gcc 옵션

- -o 실행파일명(*.*): 실행파일의 이름을 직접 지정
  - 예) gcc -o result main.c ← result라는 실행파일을 생성
  - o 옵션 뒤에는 실행파일이름이 붙어야함 (한 세트라고 생각하기)

- -g: 디버깅 활성화
  - 예) gcc -g main.c
  - 만약 o 옵션을 같이 사용하고 싶다면 gcc -g -o main main.c를 입력

(참고)
- 컴파일할 때 소스파일의 위치는 상관없다. (gcc -o main **main.c** -g, gcc **main.c** -o main -g)
- o 옵션을 생략하고 컴파일하면 기본적으로 a.out 이라는 이름을 가진 실행파일이 생성되는데, 실행 방법은 터미널에서 ./a.out을 입력한다.

gdb 소개 및 사용법

EDDI
Electronic Design Development Institute

1) gdb 란

디버깅 소프트웨어

2) 사용법

(터미널에서) gdb 실행파일명 ← 실행파일을 디버깅 시작

2-1) gdb 내에서 명령어

- b 함수이름: 해당 함수에 중단점을 삽입
- r: 중단점 전까지 프로그램을 실행
- disas: 현재 실행 중인 함수를 디스어셈블하여 표시
- info registers: 실제 CPU 레지스터의 정보를 표시
- si: 실행 (단위: 어셈블리어)
- s: 실행 (단위: C언어)
- p 이름: 이름에 해당하는 레지스터나 변수의 값을 출력
- shell clear: 화면을 깨끗이하기

(참고) 용어의 의미

- b(breakpoint): 중단점
  - 사용 목적: 어떤 위치에서 실행을 일시 중지하려는 경우
- r(run): 실행
- disas(disassemble): 기계어에서 어셈블리어로 변환
- info registers
  - 사용 목적: 디버깅 과정에서 사용하고 있는 레지스터의 종류와 값을 보려는 경우
- si(step instruction)
- s(step)

레지스터

1) 레지스터는 무엇인가

- 레지스터는 컴퓨터의 프로세서(CPU) 내에서 자료를 보관하는 아주 빠른 기억 장소이다. 일반적으로 현재 계산을 수행중인 값을 저장하는 데 사용된다.
- 레지스터는 메모리 계층의 최상위에 위치하며, 가장 빠른 속도로 접근 가능한 메모리이다.

문장과 그림 출처: 위키피디아 (ko.wikipedia.org/wiki/메모리_계층_구조)

2) 반드시 알아야하는 레지스터들

- **ax**: 리턴값
- **sp**: 현재 스택의 최상위
- **bp**: 스택의 base (기준점)
  - 함수 스택 시작 위치를 저장하는 레지스터
- ip: PC(program counter) 다음에 실행할 명령어의 주소

메모리 계층 구조

(참고)
뒤에 나올 실습 과정에서 레지스터 이름을 보면 위의 '반드시 알아야하는 레지스터들'과 이름이 조금 다르다.

다른 이유 : 이름 앞에 아무 것도 붙지 않은 경우 16비트, e가 붙은 경우 32비트(eax), r이 붙은 경우 64비트(rsp)이다.

가장 중요한 개념

EDDI
Electronic Design
Development Institute

- 스택은 거꾸로 자란다. (이건 배열의 인덱스가 0부터 시작하는 것처럼 이해의 요소가 아닌 암기에 가깝다)
- 모든 것은 메모리다. (이제 증명해보자)

어셈블리어 명령어

EDDI
Electronic Design Development Institute

- **push**: 메모리에 정보를 배치 (기준: sp)
  - 예) push %rbp ← sp 값을 8바이트 감소 후, sp 위치에 rbp 값을 저장(백업)
  - 위의 코드를 두 줄로 나타내면 다음과 같음
  - sub $0x8, %rsp
  - mov %rbp, %rsp

- **mov**: 데이터 복사
  - 예) mov %rsp, %rbp ← rsp 값을 rbp에 저장
  - movl: 4바이트 처리
  - movq: 8바이트 처리

- sub: 뺄셈 연산
  - sub source, destination ← destination 값에서 source 값을 빼고 그 값을 destination에 저장

- add: 덧셈 연산
  - add source, destination ← destination 값에서 source 값을 더하고 그 값을 destination에 저장

어셈블리어 명령어

EDDI
Electronic Design Development Institute

- **pop**: sp 값을 destination에 배치 후, sp 값이 8 증가
  - 예) pop %rbp ← sp에 위치한 값을 rbp에 배치 후, sp 값을 8 증가
  - 위의 코드를 두 줄로 나타내면 다음과 같음
  - mov %rsp, %rbp
  - add $0x8, %rsp

- ret: pop ip와 동의어
  - 설명은 생략^^

- **call**: 함수 호출
  - push + jmp
  - 복귀주소 저장 (함수 호출 후 동작할 주소)
  - 이 명령어를 이렇게 이해하는 것이 좋을 듯 하다 → push $복귀주소(함수가 끝나고 다음에 실행할 주소)

func.c의 어셈블리어 분석

EDDI
Electronic Design Development Institute

1) func.c

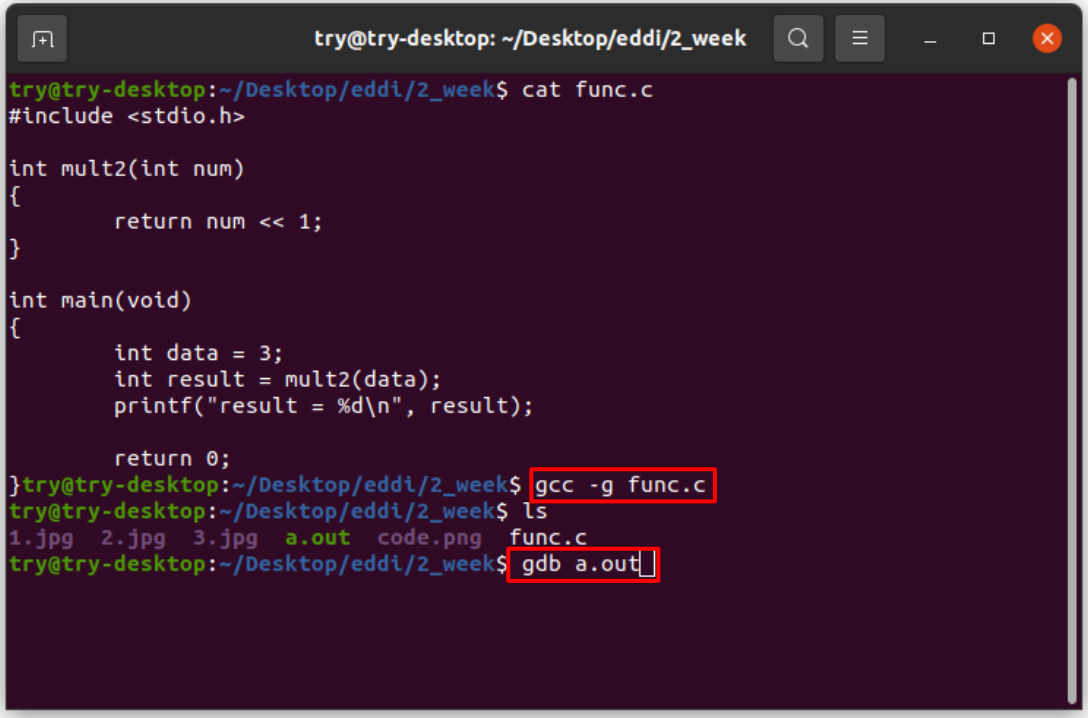
```c
#include <stdio.h>

int mult2(int num)
{
    return num << 1;
}

int main(void)
{
    int data = 3;
    int result = mult2(data);
    printf("result = %d\n", result);

    return 0;
}
```

https://carbon.now.sh/ 사용

func.c의 어셈블리어 분석

EDDI
Electronic Design Development Institute

2) g 옵션으로 컴파일 후, gdb 실행

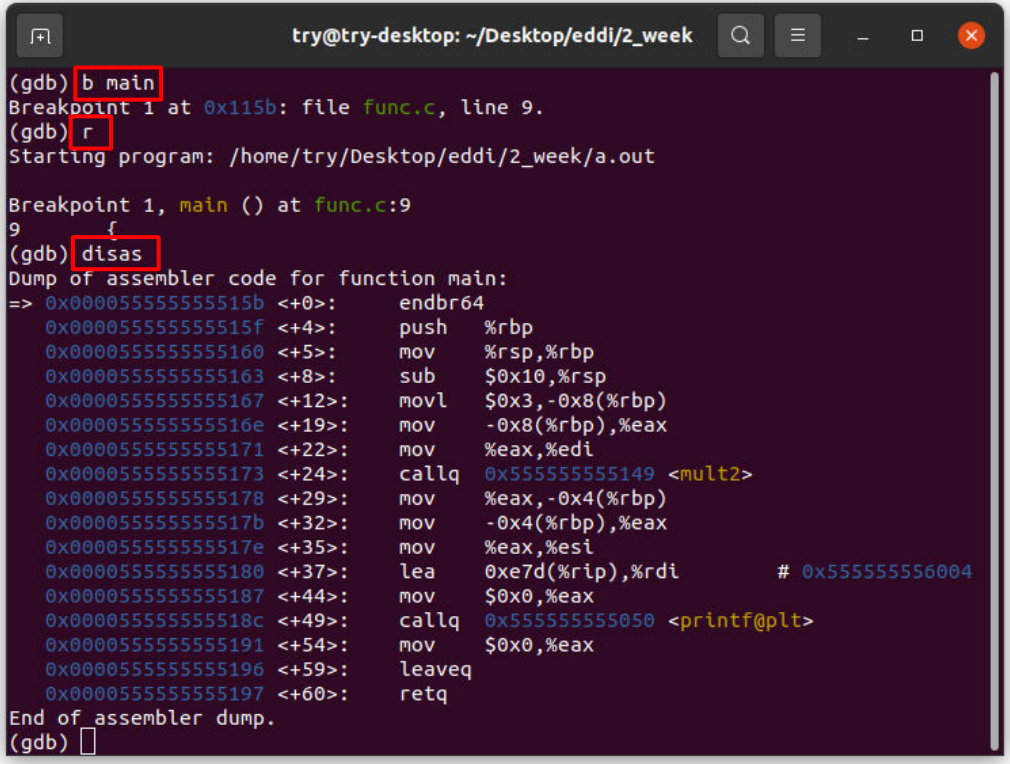


(참고) 어셈블리어를 분석하는 환경

- CPU: **AMD** 라이젠5 PRO 4650G
- 메인보드: ASUS PRIME B550M-A
- 메모리: 삼성전자 DDR4-3200 (8GB) * 2개

func.c의 어셈블리어 분석

3) 분석 준비 단계



설명

1. b main
   - main 함수에 중단점 삽입
1. r
   - main 함수 호출 직전까지 실행
1. disas
   - 현재 위치해 있는 함수를 어셈블리어로 출력
   - 이 명령어에 의해 출력되는 내용 중에서 화살표의 의미는 다음에 시작할 어셈블리어 명령어를 의미함

func.c의 어셈블리어 분석

4) 레지스터 값 확인하기

```
(gdb) info registers
rax            0x55555555515b      93824992235867
rbx            0x5555555551a0      93824992235936
rcx            0x5555555551a0      93824992235936
rdx            0x7fffffffdf18      140737488346904
rsi            0x7fffffffdf08      140737488346888
rdi            0x1                 1
rbp            0x0                 0x0
rsp            0x7fffffffde18      0x7fffffffde18
r8             0x0                 0
r9             0x7ffff7fe0d60      140737354009952
r10            0x7ffff7ffcf68      140737354125160
r11            0x202               514
r12            0x555555555060      93824992235616
r13            0x7fffffffdf00      140737488346880
r14            0x0                 0
r15            0x0                 0
rip            0x55555555515b      0x55555555515b <main>
eflags         0x246               [ PF ZF IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
(gdb)
```

레지스터 이름 16진수 값 10진수

설명

명령어를 사용하면 왼쪽처럼 출력이 이루어지는데, 그 정보는 다음과 같다

- 왼쪽: 레지스터 이름
- 중간: 실제 레지스터 내에 저장된 값을 16진수 출력
- 오른쪽: 중간에 있는 값을 사용자가 어떤 값인지 알기 쉽게 주소 값이면 16진수, 그냥 값이면 10진수로 출력

(주소는 간략히 작성)

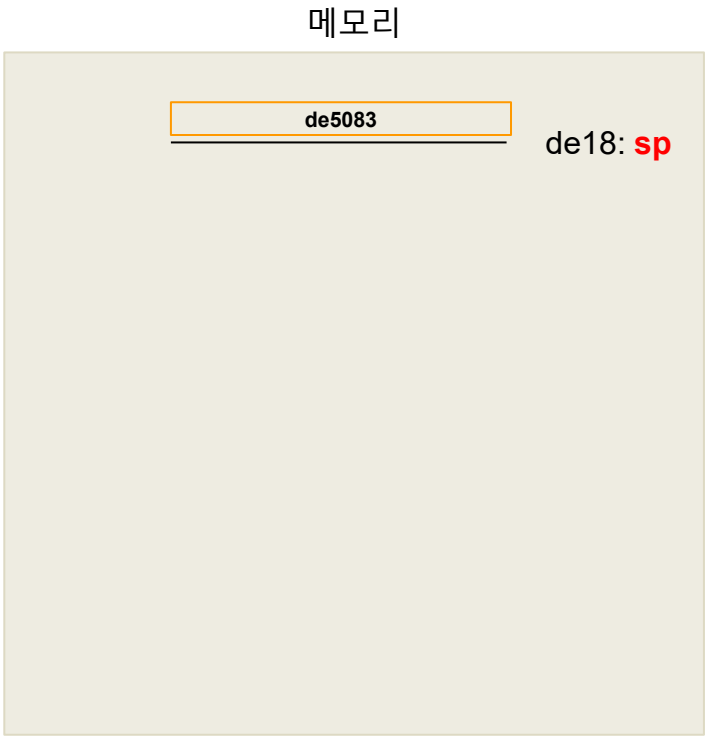
- rbp 값은 0
- rsp 값은 de18
- rip 값은 515b 인데, main 함수의 시작 주소를 가리킨다.

이제 si 명령어를 입력하여 메모리에서 레지스터의 값과 스택이 어떻게 구성이 되는지 살펴보자.

참고로 ***endbr64 명령어는 생략하겠다***

func.c의 어셈블리어 분석

앞의 슬라이드를 그림으로 보여줌



설명

main 함수에 진입한 상태이다.

rsp 값은 de18
rbp 값은 0

```
(gdb) x/g $rsp
0x7fffffffde18: 0x00007ffff7de5083
```

rsp에 저장된 주소값의 위치(이하 rsp 위치)에는 de5083이라는 값이 저장되어 있다.

func.c의 어셈블리어 분석

5) push 명령어 분석

```
(gdb) p $rsp
$6 = (void *) 0x7fffffffde18
(gdb) x $rsp-8
0x7fffffffde10: 0x00005555555551a0
```

rsp 값

push 명령어 실행하기 전

```
(gdb) si
0x0000555555555160      9       {
(gdb) disas
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 <mult2>
```

실행

push 명령어 실행

```
(gdb) p $rsp
$7 = (void *) 0x7fffffffde10
(gdb) x/g $rsp
0x7fffffffde10: 0x0000000000000000
```

rsp 값

rsp 위치에 저장된 값

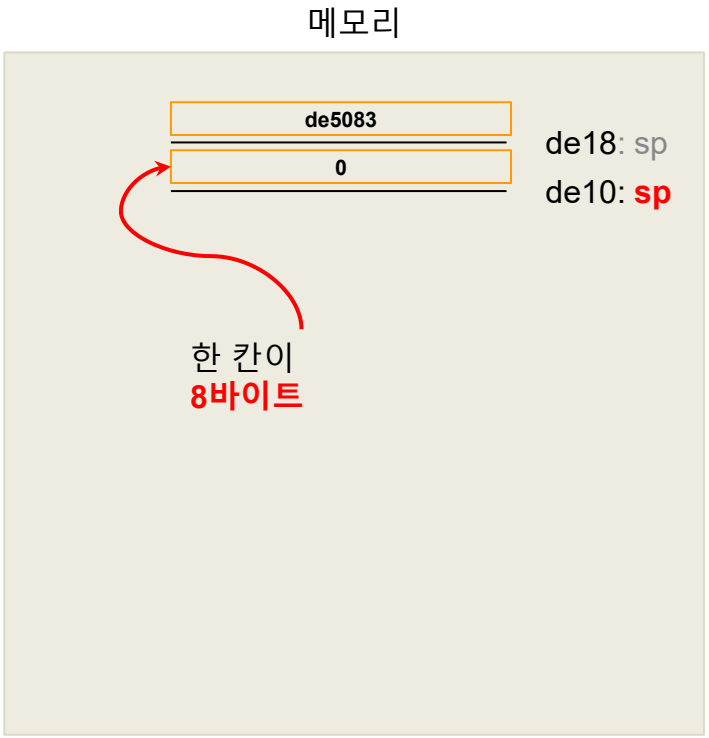
push 명령어 실행한 후

설명

- disas 명령어에 의해 출력된 내용 중 push 명령어 아래에 있는 줄을 가리키는 화살표의 의미는 바로 위에 있는 push 명령어를 실행했다는 것임
- 레지스터 값을 push 명령어를 실행하기 전과 실행한 후를 비교해보면 **rsp의 레지스터 값이 8(바이트)이 감소**했다는 것을 알 수 있음
- 또한, push 명령어를 실행하기 전에 de10 위치에는 51a0 이라는 값이 저장되어 있었는데, 명령어를 실행한 후에는 **값이 감소한 rsp 위치에 rbp의 값이 저장**

func.c의 어셈블리어 분석

앞의 슬라이드를 그림으로 보여줌



설명

rsp 값은 de10

메모리에서 sp 위치에 저장된 값은 rbp 값이다.

func.c의 어셈블리어 분석

6) mov 명령어 분석

```
(gdb) p $rsp
$10 = (void *) 0x7fffffffde10
(gdb) p $rbp
$11 = (void *) 0x0
```

rsp 값

rbp 값

mov 명령어 실행하기 전

```
(gdb) disas
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 <mult2>
```

실행

mov 명령어 실행

```
(gdb) p $rsp
$13 = (void *) 0x7fffffffde10
(gdb) p $rbp
$14 = (void *) 0x7fffffffde10
```

rsp 값

rbp 값

mov 명령어 실행한 후

설명

- mov 명령어 실행 후, **rsp 값이 rbp에 복사되어 값이 동일해짐.**

그 다음 sub 명령어를 실행하자

func.c의 어셈블리어 분석

앞의 슬라이드를 그림으로 보여줌

설명

rsp 값과 rbp 값이 de10으로 동일

func.c의 어셈블리어 분석

7) sub 명령어 분석

```
(gdb) p $rsp
$16 = (void *) 0x7fffffffde10
```

rsp 값

sub 명령어 실행하기 전

```
(gdb) disas
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 <mult2>
```

실행

sub 명령어 실행

```
(gdb) p $rsp
$17 = (void *) 0x7fffffffde00
```

rsp 값

sub 명령어 실행한 후

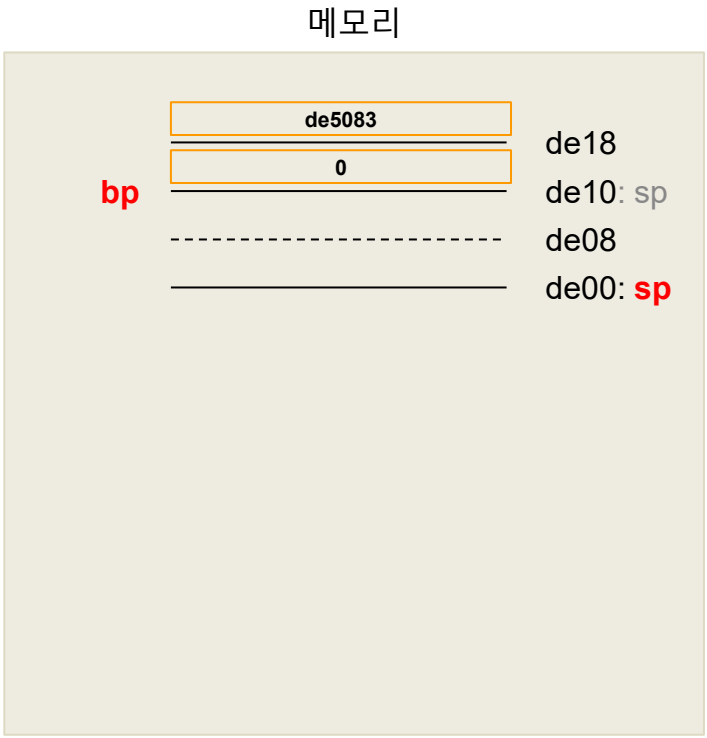
설명

- sub 명령어 실행 후, **rsp의 레지스터 값이 16(바이트)이 감소**함.

그 다음 movl 명령어를 실행하자

func.c의 어셈블리어 분석

앞의 슬라이드를 그림으로 보여줌



설명

rsp 값이 16 감소하여 de00이 됨

func.c의 어셈블리어 분석

8) mov 명령어 분석

```
(gdb) x/w $rbp-8
0x7fffffffde08: 0x00000000
```

mov 명령어 실행하기 전

```
(gdb) disas
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 <mult2>
```

실행

mov 명령어 실행

```
(gdb) x/w $rbp-8
0x7fffffffde08: 0x00000003
```

mov 명령어 실행한 후

설명

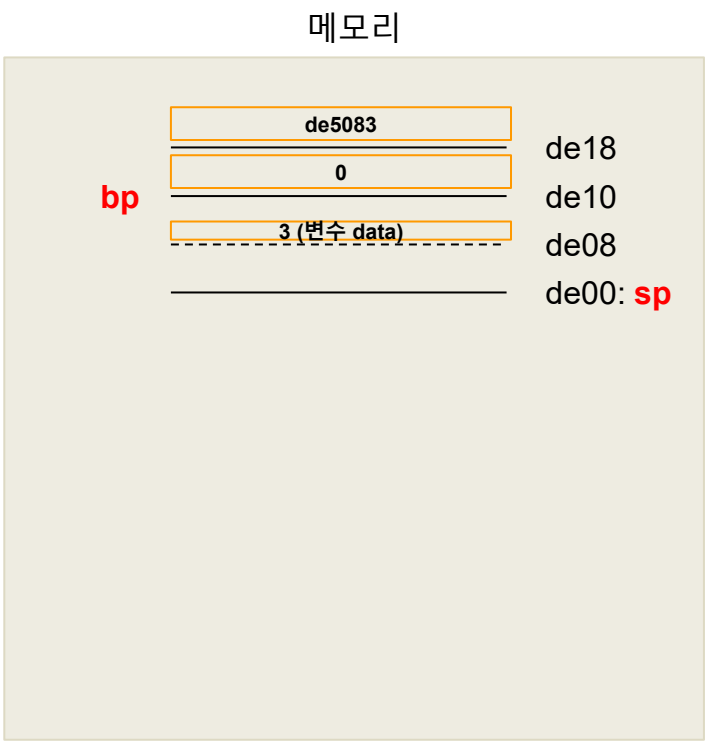
- mov 뒤에 'l'이 붙었으므로, 4바이트로 처리
- mov 명령어 실행 후, **rbp 값에서 8을 뺀 주소값에 3을 4바이트로 저장**
- 이 부분이 C 언어 코드에서 int data = 3; 에 해당함

그 다음 mov 명령어를 실행하자

x 명령어의 출력결과로 왼쪽은 주소값(또는 계산된 주소값), 오른쪽은 왼쪽주소에 저장되어 있는 값

func.c의 어셈블리어 분석

앞의 슬라이드를 그림으로 보여줌



설명

rbp-8에 해당하는 주소에 정수 3을 4바이트로 저장

func.c의 어셈블리어 분석

9) mov 명령어 두 줄 분석

```
(gdb) p/x $rax
$31 = 0x555555555515b
(gdb) p $rdi
$32 = 1
```

rax 값

rdi 값

두 mov 명령어 실행하기 전

```
(gdb) disas
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 <mult2>
```

실행

두 mov 명령어 실행

```
(gdb) p $rax
$40 = 3
(gdb) p $rdi
$41 = 3
```

rax 값

rdi 값

두 mov 명령어 실행한 후

설명

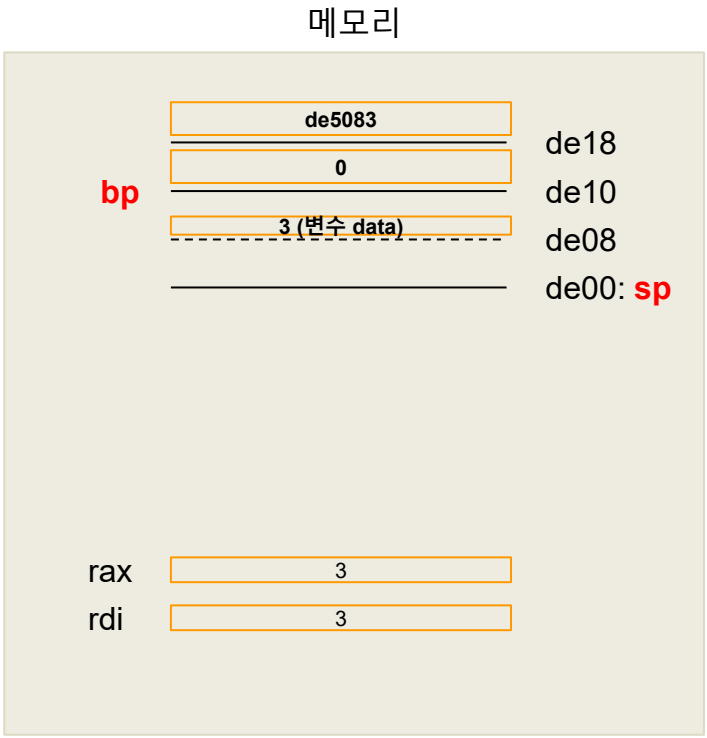
1. 첫 번째 mov 명령어에 의해 3을 저장했던 공간의 값을 eax로 복사

1. 두 번째 mov 명령어에 의해 eax 값을 edi로 복사

그 다음 callq 명령어를 실행하자

func.c의 어셈블리어 분석

앞의 슬라이드를 그림으로 보여줌



설명

레지스터 rax와 rdi에 정수 3이 저장되어있음

func.c의 어셈블리어 분석

10) mult2 함수 호출하는 call 명령어 분석

```
(gdb) p $rsp
$4 = (void *) 0x7fffffffde00
(gdb) x/g $rsp-8
0x7fffffffddf8: 0x0000555555555060
```

rsp 값

call 명령어 실행하기 전

```
(gdb) disas
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 <mult2>
   0x0000555555555178 <+29>:    mov    %eax,-0x4(%rbp)
```

실행

mult2 함수 끝나고 실행할 주소

call 명령어 실행

```
(gdb) p $rsp
$5 = (void *) 0x7fffffffddf8
(gdb) x/g $rsp
0x7fffffffddf8: 0x0000555555555178
```

rsp 값

rsp 위치에 저장된 값 = 복귀주소

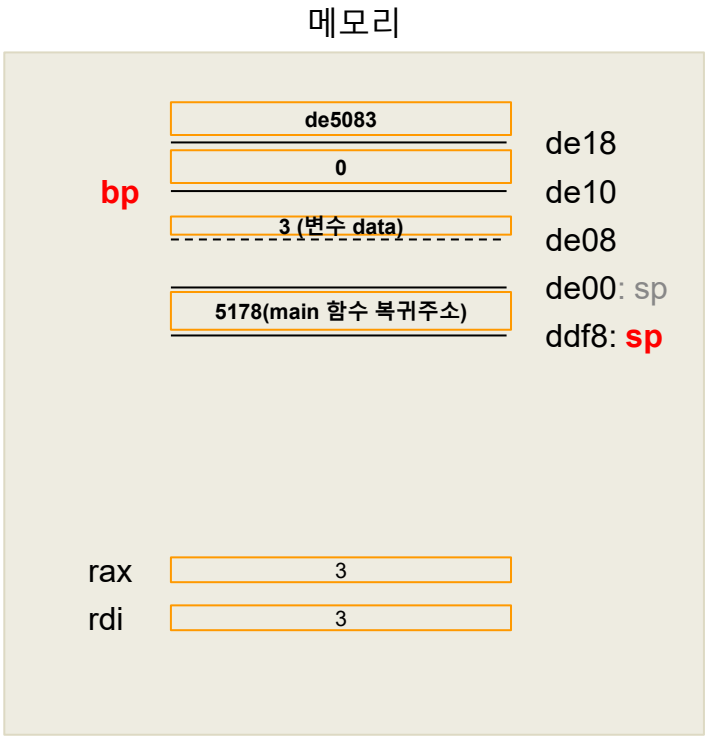
call 명령어 실행한 후

설명

- call 명령어에 의해 **rsp의 레지스터 값이 8(바이트)이 감소**하고 **rsp 위치에 함수 호출이 끝나면 돌아갈 복귀주소를 저장**함.
- 그리고, **mult2 함수의 시작위치로 점프**

func.c의 어셈블리어 분석

앞의 슬라이드를 그림으로 보여줌



설명

rsp 값은 ddf8

rsp 위치에 main 함수 복귀주소가 저장되어있음

func.c의 어셈블리어 분석

11) mult2 함수 시작 부분에서 add 명령어 직전까지 실행

```
(gdb) p $rsp
$6 = (void *) 0x7fffffffddf8
(gdb) p $rbp
$7 = (void *) 0x7fffffffde10
(gdb) p $rax
$8 = 3
```

rsp 값

rbp 값

명령어 실행하기 전

```
(gdb) disas
Dump of assembler code for function mult2:
=> 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>:    add    %eax,%eax
   0x0000555555555159 <+16>:    pop    %rbp
   0x000055555555515a <+17>:    retq
End of assembler dump.
```

mult2 함수

실행

명령어 실행

```
(gdb) p $rsp
$19 = (void *) 0x7fffffffddf0
(gdb) p $rbp
$20 = (void *) 0x7fffffffddf0
(gdb) x/gx $rsp
0x7fffffffddf0: 0x00007fffffffde10
(gdb) x/wd $rsp-4
0x7fffffffddec: 3
(gdb) p $rax
$21 = 3
```

rsp 값

rbp 값

rsp 위치 값 = main 함수 bp 주소

매개변수 num 값

rax 값

명령어 실행한 후

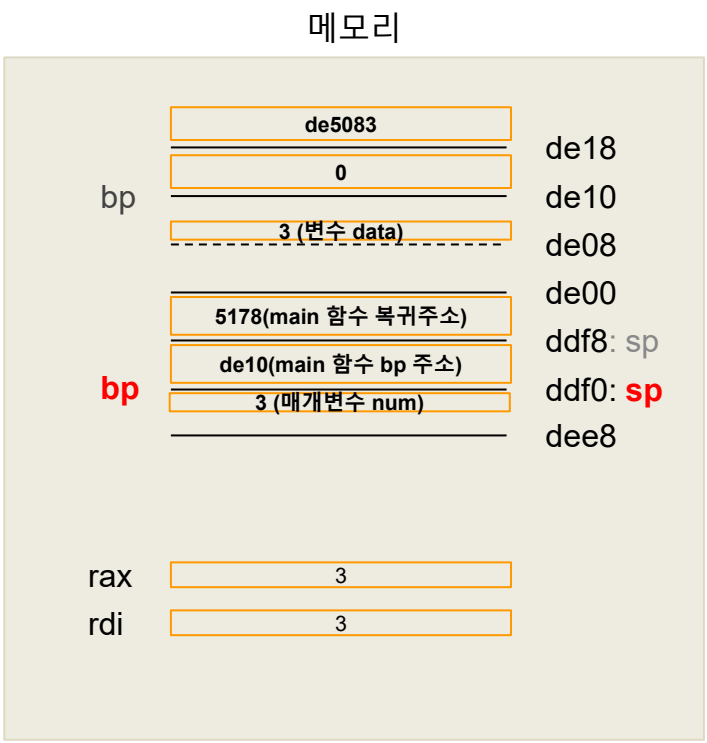
설명

1. push 명령어에 의해 rsp 값이 8 감소, 해당 위치에 rbp 값을 복사
1. 첫 번째 mov 명령어에 의해 rsp 값이 rbp에 복사
1. 두 번째 mov 명령어에 의해 edi 값을 스택(함수 매개변수 num)에 복사
1. 세 번째 mov 명령어에 의해 매개변수를 eax로 복사

그 다음 add 명령어를 실행하자

func.c의 어셈블리어 분석

앞의 슬라이드를 그림으로 보여줌



설명

rsp 값은 ddf0
rsp 위치에 main 함수의 bp 주소가 저장(백업)

rbp 값은 ddf0
rbp 바로 밑에 4바이트로 매개변수 num인 3이 저장

func.c의 어셈블리어 분석

12) add 명령어 분석

```
(gdb) p $rax
$25 = 3
```
rax 값

add 명령어 실행하기 전

```
(gdb) disas
Dump of assembler code for function mult2:
   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>:    add    %eax,%eax
   0x0000555555555159 <+16>:    pop    %rbp
   0x000055555555515a <+17>:    retq
End of assembler dump.
```
실행

add 명령어 실행

```
(gdb) p $rax
$26 = 6
```
rax 값

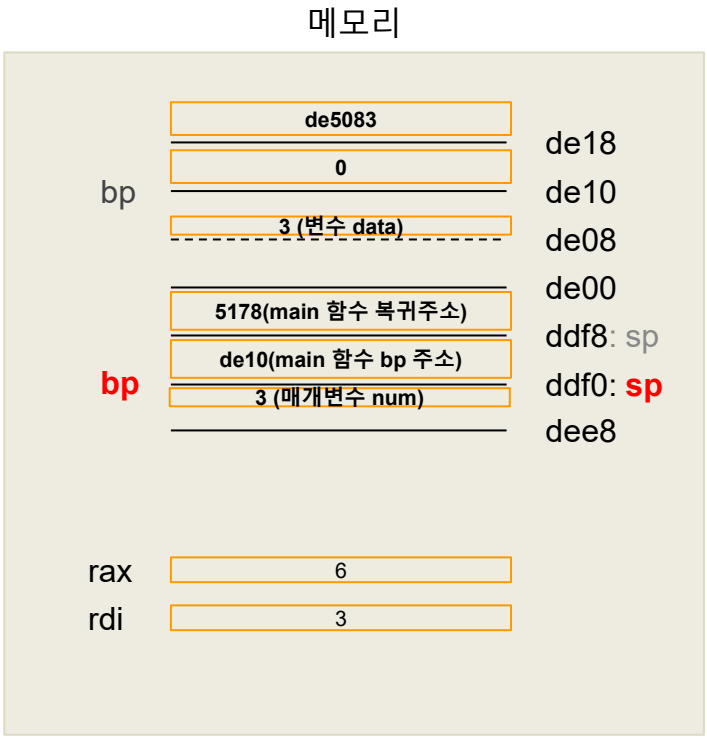
add 명령어 실행한 후

설명

- add 명령어에 의해 rax 값에 rax 값을 더하여, 6을 저장

func.c의 어셈블리어 분석

앞의 슬라이드를 그림으로 보여줌



설명

rax 값은 3에서 6으로 바뀜

func.c의 어셈블리어 분석

13) pop 명령어 분석

```
(gdb) p $rsp
$27 = (void *) 0x7fffffffddf0
(gdb) p $rbp
$28 = (void *) 0x7fffffffddf0
(gdb) x/gx $rsp
0x7fffffffddf0: 0x00007fffffffde10
```

rsp 값

rbp 값

rsp 위치 값
=
main 함수 bp 값

pop 명령어 실행하기 전

```
(gdb) disas
Dump of assembler code for function mult2:
   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>:    add    %eax,%eax
=> 0x0000555555555159 <+16>:    pop    %rbp
   0x000055555555515a <+17>:    retq
End of assembler dump.
```

실행

pop 명령어 실행

```
(gdb) p $rsp
$29 = (void *) 0x7fffffffddf8
(gdb) p $rbp
$30 = (void *) 0x7fffffffde10
```

rsp 값

rbp 값

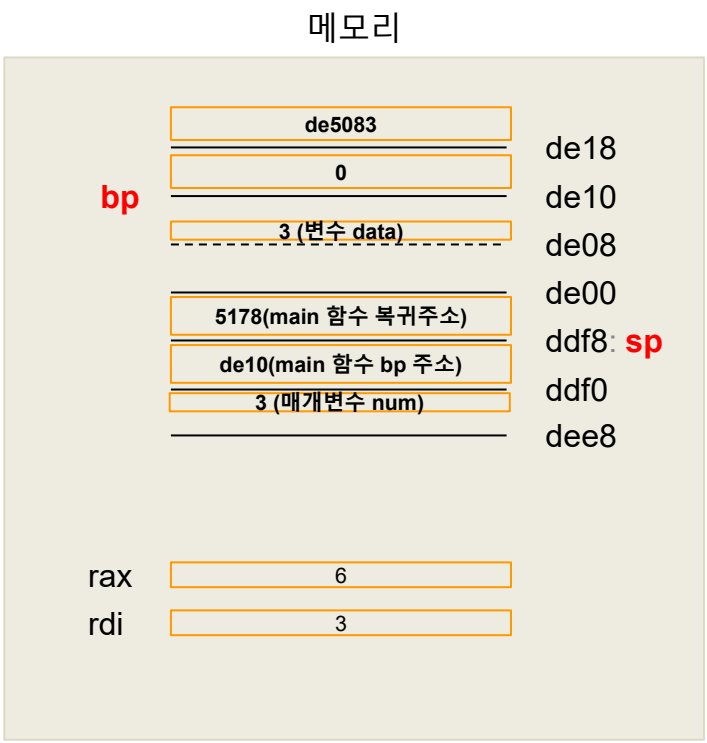
pop 명령어 실행한 후

설명

- pop 명령어에 의해 **rsp 위치에 있는 값이 rbp에 복사** 후, **rsp 값이 8(바이트) 증가**

func.c의 어셈블리어 분석

앞의 슬라이드를 그림으로 보여줌



설명

rsp(ddf0)에 있는 값인 main 함수 (스택의) bp 주소를 rbp에 복사하고, rsp 값을 8증가 시킴

rsp 값은 ddf8
rbp 값은 de10

func.c의 어셈블리어 분석

14) ret 명령어 분석

```
(gdb) p $rsp
$31 = (void *) 0x7fffffffddf8
(gdb) p $rip
$32 = (void (*)()) 0x55555555515a <mult2+17>
(gdb) x/g $rsp
0x7fffffffddf8: 0x0000555555555178
```

rsp 값

rip 값

rsp 위치 값

=

main 함수 복귀주소

ret 명령어 실행하기 전

```
(gdb) disas
Dump of assembler code for function mult2:
   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>:    add    %eax,%eax
   0x0000555555555159 <+16>:    pop    %rbp
=> 0x000055555555515a <+17>:    retq
End of assembler dump.
```

실행

ret 명령어 실행

```
(gdb) p $rsp
$33 = (void *) 0x7fffffffde00
(gdb) p $rip
$34 = (void (*)()) 0x555555555178 <main+29>
```

rsp 값

rip 위치 값

=

main 함수 복귀주소

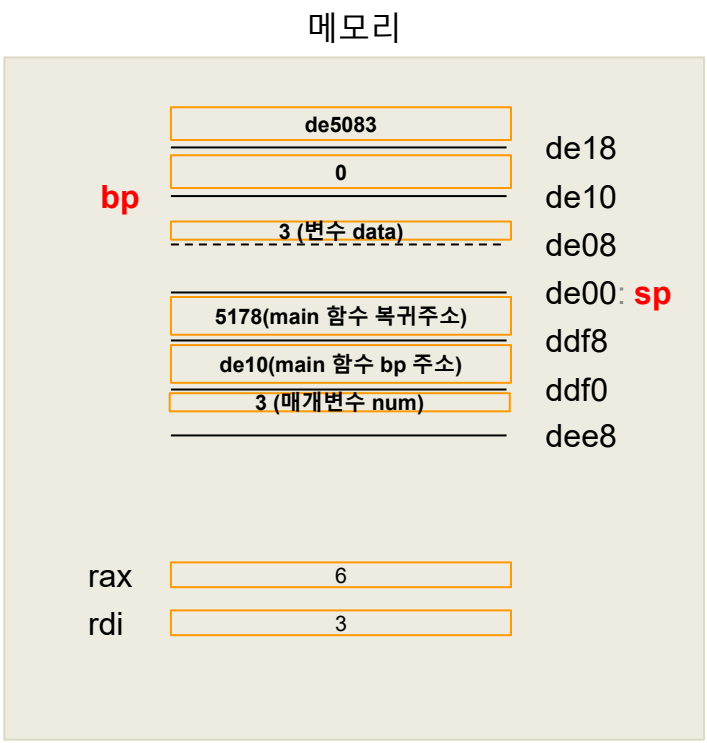
ret 명령어 실행한 후

설명

- ret 명령어에 의해 **rsp 위치에 있는 값이 rip에 복사** 후, **rsp 값이 8(바이트) 증가**

func.c의 어셈블리어 분석

앞의 슬라이드를 그림으로 보여줌



설명

rsp(ddf8)에 있는 값인 main 함수 복귀 주소를 rip에 복사하고, rsp 값을 8증가 시킴

rsp 값은 de00

func.c의 어셈블리어 분석

15) mult2 함수 반환 후, lea 명령어 직전까지 실행

```
(gdb) x/wd $rbp-4
0x7fffffffde0c: 0
(gdb) p $rax
$43 = 6
(gdb) p $rsi
$44 = 140737488346888
(gdb)
```

rax 값

rsi 값

명령어 실행하기 전

```
   0x0000555555555173 <+24>:    callq  0x555555555149 <mult2>
=> 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
End of assembler dump.
```

실행

명령어 실행

```
(gdb) x/wd $rbp-4
0x7fffffffde0c: 6
(gdb) p $rax
$47 = 6
(gdb) p $rsi
$48 = 6
```

지역변수 result 값

rax 값

rsi 값

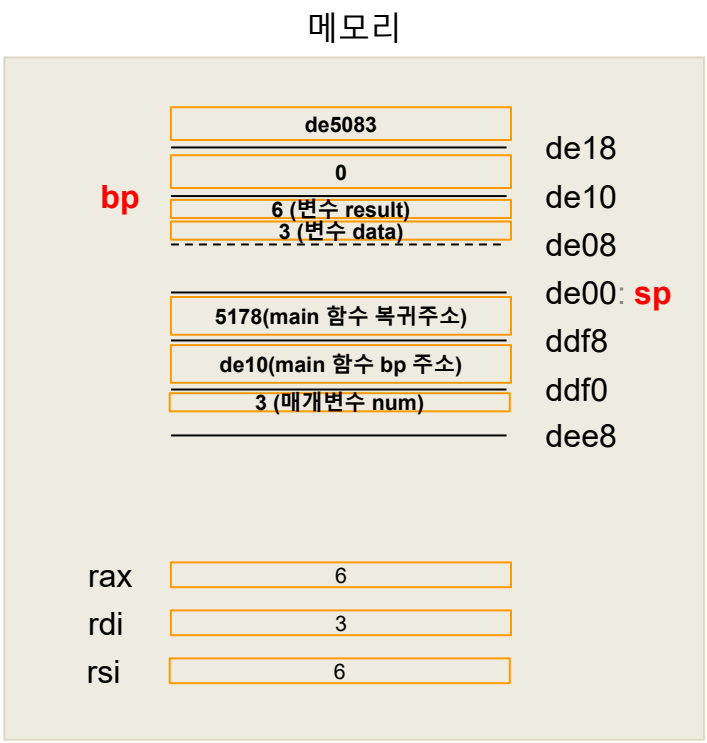
명령어 실행한 후

설명

1. 첫 번째 mov 명령어에 의해 eax 값을 스택(함수 지역변수 result)에 복사
1. 두 번째 mov 명령어에 의해 매개변수를 eax에 복사
1. 세 번째 mov 명령어에 의해 eax 값을 esi에 복사

func.c의 어셈블리어 분석

앞의 슬라이드를 그림으로 보여줌



설명

mult2 함수에서 계산한 6을 eax에 저장했는데, 이 값을 main 함수 스택(지역변수 result)에 배치

저장된 지역변수 result 값을 eax에 복사

eax 값을 esi에 복사

func.c의 어셈블리어 분석

15) lea 명령어 맛보기

```
   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
End of assembler dump.
```

```
(gdb) x/bc 0x555555556004
0x555555556004: 114 'r'
(gdb)
0x555555556005: 101 'e'
(gdb)
0x555555556006: 115 's'
(gdb)
0x555555556007: 117 'u'
(gdb)
0x555555556008: 108 'l'
(gdb)
0x555555556009: 116 't'
(gdb)
```

설명

1. lea 명령어는 왼쪽의 주소값을 오른쪽에 저장한다
1. gdb 디버거는 사용자가 직접 계산할 필요 없이 오른쪽에 계산된 값을 제공해주고 있다
1. 이 주소값에 접근해 보자
1. printf 함수의 인자로 전달한 문자열의 문자가 저장된 것을 보아, 문자열의 시작주소값을 rdi에 저장하고 있는 명령어임을 알 수 있음

결론

gdb를 이용하여 어셈블리어를 분석한 결과는 모두 최종적으로 레지스터를 이용한 데이터의 이동과 연산 뿐으로,
모든 것은 메모리다를 증명하였다.

(참고)
어셈블리어를 분석한 결과로 스택 관리를 하는 명령어가 함수의 시작 부분과 끝 부분에 있었고, 간단한
프로그램인데도 불구하고 gcc 컴파일러가 멍청한 짓을 한다는 것을 깨달았다.

결론 = 모든 것은 메모리