C언어 - HW4

임베디드스쿨2기
Lv1과정
2021. 04. 07
차현호

어셈블리언어 분석

아래 c 코드를 컴파일하고 gdb를 이용하여 어셈블리어를 분석해보자.

```c
#include <stdio.h>

int my_func(int num)
{
    // 쉬프트 연산자로 비트를 전
    // num << 1 은 1칸, num << 2
    // 비트의 2^n을 의미하므로 곱

    // num >> 1은 1칸, num >> 2는
    // 그러나 이 방식은 우측 이동
    // 그러므로 나누기와 같은데
    // 소수점을 취급하지 않아 0.
    return num >> 1;
}

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

    res = my_func(num);

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

    return 0;
}
```

아래 c 코드를 컴파일하고 gdb를 이용하여 어셈블리어를 분석해보자.

어셈블리언어 분석

movl $0x3, -0x8(%rbp)는 main함수의 최상단 스택 주소(rbp)로 부터 8바이트 만큼 뺀 주소에 3을 대입하는 구문이다.

%rbp주소에서 8바이트만큼 뺀곳은 변수 num의 주소이므로 c코드에서 보는것처럼 num에다가 3을 대입하는 연산임을 알 수 있다.

Mov –0x8(%rbp), %eax 구문은 rbp에서 8바이트만큼 뺀주소(num)값을 eax 레지스터에 대입하는 구문이다.

마지막으로 mov %eax,%edi는 edi레지스터에 eax의값(3)을 넣는 연산이다.

```
0x0000555555554658 <+0>:     push   %rbp
0x0000555555554659 <+1>:     mov    %rsp,%rbp
0x000055555555465c <+4>:     sub    $0x10,%rsp
0x0000555555554660 <+8>:     movl   $0x3,-0x8(%rbp)
0x0000555555554667 <+15>:    mov    -0x8(%rbp),%eax
0x000055555555466a <+18>:    mov    %eax,%edi
```

어셈블리언어 분석

그다음은 my_func함수를 호출하는 부분인데 callq 명령어에 대해 짚고 넘어가자

callq는 push + jmp로 되어있는데 함수호출이 끝난후에 다음에 실행할 주소를 스택에 저장한다.

그러면 실제로 callq가 실행 된 후에 스택에 0x0000555555554671이 저장되었는지 보자

```
0x000055555555466c <+20>:    callq  0x55555555464a <my_func>
```

```
=> 0x000055555555464a <+0>:     push   %rbp
   0x000055555555464b <+1>:     mov    %rsp,%rbp
   0x000055555555464e <+4>:     mov    %edi,-0x4(%rbp)
   0x0000555555554651 <+7>:     mov    -0x4(%rbp),%eax
   0x0000555555554654 <+10>:    sar    %eax
   0x0000555555554656 <+12>:    pop    %rbp
   0x0000555555554657 <+13>:    retq
End of assembler dump.
(gdb) x 0x7fffffffdf08
0x7fffffffdf08: 0x55554671
(gdb) x 0x7fffffffdf08+4
0x7fffffffdf0c: 0x00005555
```

실제 0x0000555555554671이 저장된것을 확인 할 수 있다.

어셈블리언어 분석

이제는 my_func 함수를 분석해보자 먼저 main함수처럼 push %rbp와
mov %rsp,%rbp를 수행하고 분기하기전에 변수 num값인 eax레지스터를 복제한
edi레지스터의 값을 새로운 스택프레임의 rbp주소보다 4바이트 뺀 주소에 대입한다.

sar은 Shift Arithmetic Right의 양자로 오른쪽 쉬프트하는 명령어이다.

그다음 pop명령은 스택에 저장되어있던 이전함수의 rbp값을 복구해준다.

```
0x000055555555464a <+0>:     push   %rbp
0x000055555555464b <+1>:     mov    %rsp,%rbp
0x000055555555464e <+4>:     mov    %edi,-0x4(%rbp)
0x0000555555554651 <+7>:     mov    -0x4(%rbp),%eax
0x0000555555554654 <+10>:    sar    %eax
0x0000555555554656 <+12>:    pop    %rbp
0x0000555555554657 <+13>:    retq
```

스택 프레임

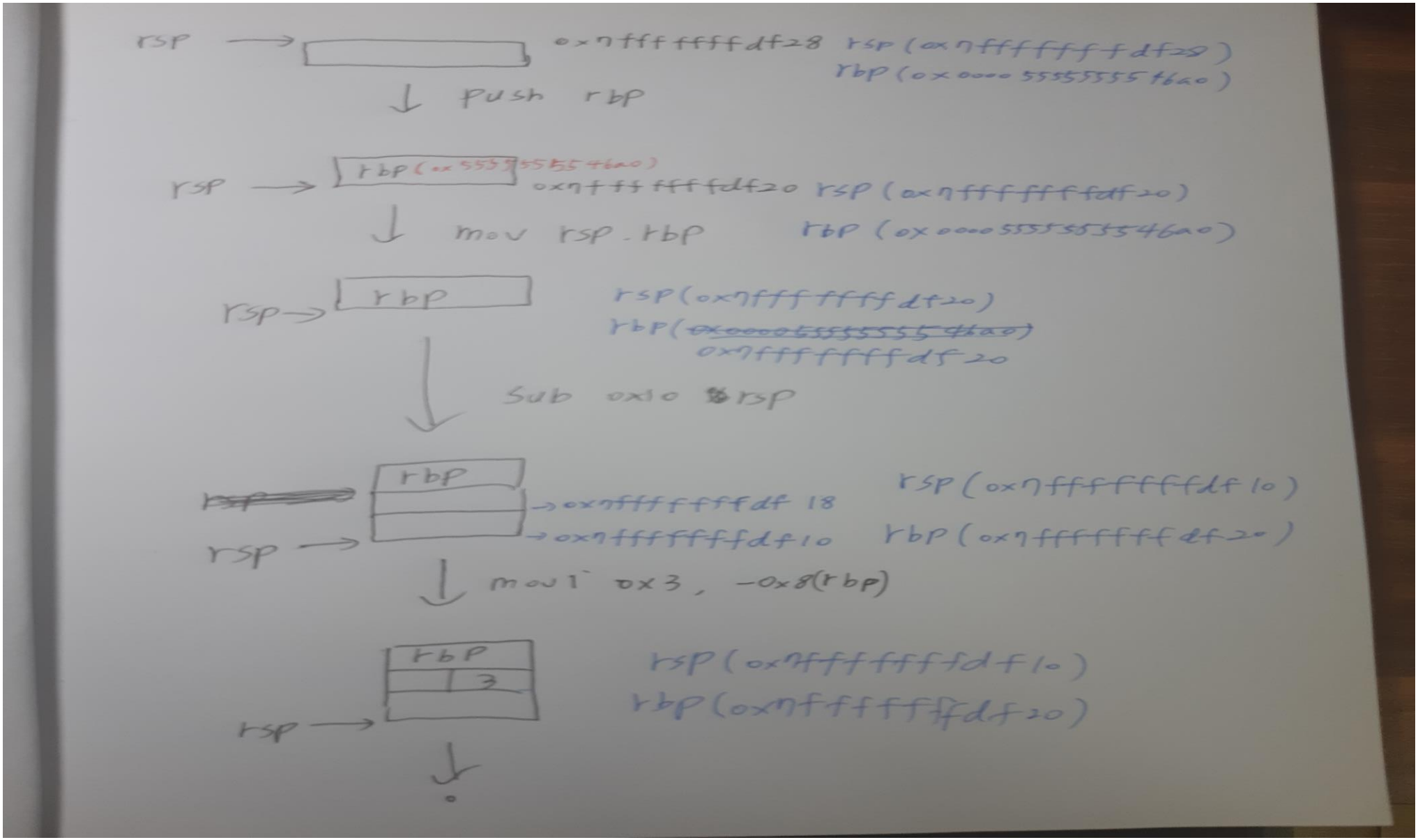
앞에서 우리는 함수가 호출될 때마다 push %rbp, mov %rsp, %rbp가 항상 실행되는 것을 확인할 수 있었다. 이부분에 대해 알아보려면 스택 프레임이라는 것을 알고 가면 도움이 된다.

스택 프레임이란 함수가 호출되었을 때 그 함수가 가지는 공간 구조이다.

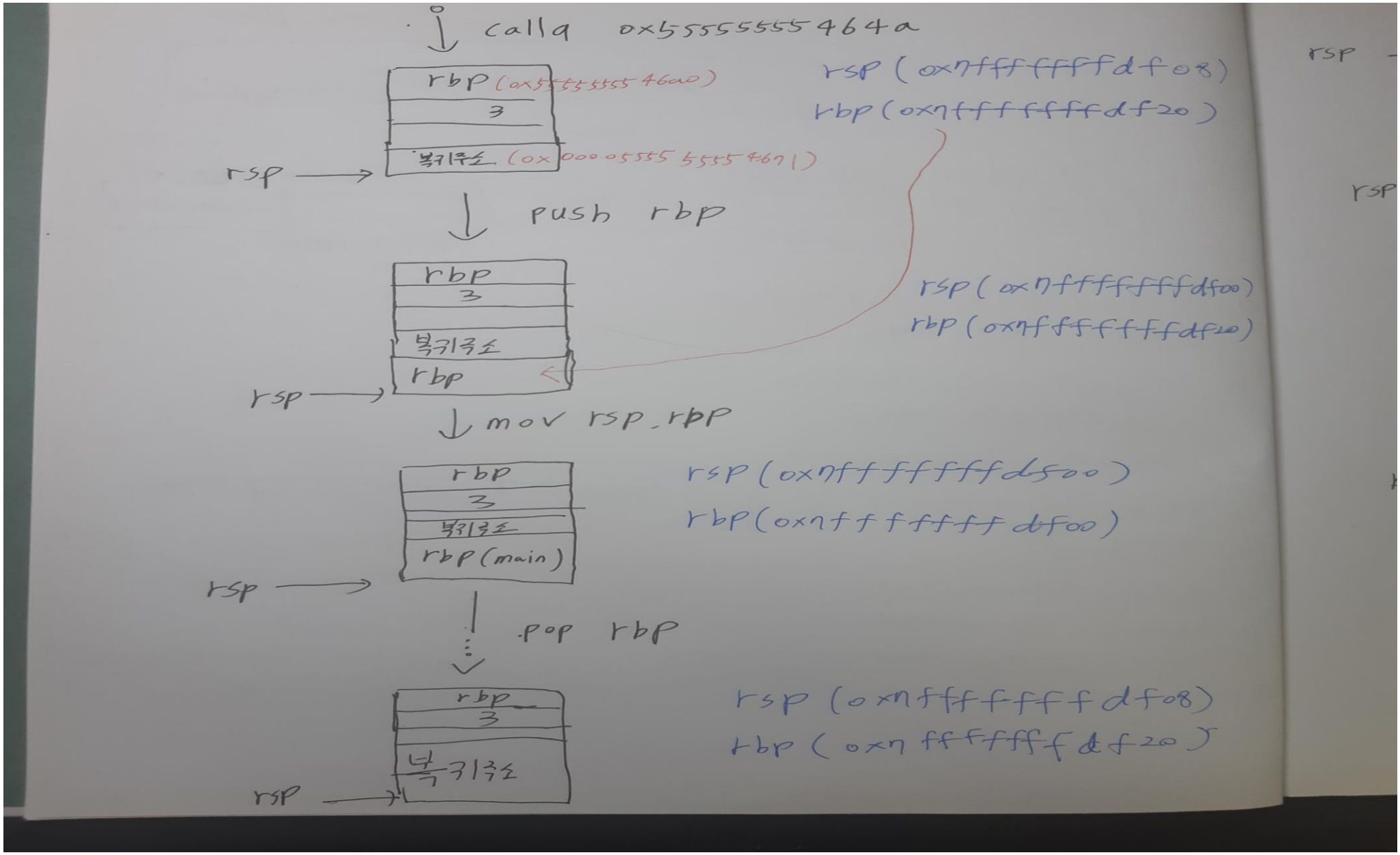
우리가 분석한 my_func의 호출을 보면 push %rbp를 하여 my_func가 호출되기전의 함수의 rbp레지스터값을 스택에 넣어두고 함수가 끝날때 pop을 통해 다시 복귀 시키는것을 볼 수 있다.

```
0x000055555555464a <+0>:     push   %rbp
0x000055555555464b <+1>:     mov    %rsp,%rbp
0x000055555555464e <+4>:     mov    %edi,-0x4(%rbp)
0x0000555555554651 <+7>:     mov    -0x4(%rbp),%eax
0x0000555555554654 <+10>:    sar    %eax
0x0000555555554656 <+12>:    pop    %rbp
0x0000555555554657 <+13>:    retq
```

그려보기



그려보기



포기하면 얻는 건 아무것도 없다.

if문 어셈블리

if문이 실행될때 어셈블리어를 분석해보자 c 코드는 아래와 같다.

```c
int main(void)
{
    int num1 = 1;
    int num2 = 1;

    if (num1 == num2)
    {
        num1++;
    }

    if (num1 > num2)
    {
        num2 += 2;
    }

    if (num1 < num2)
    {
        num1++;
    }

    return 0;
}
```

변수 num1과 num2를 선언하고 이 두 변수를 계속해서 비교해가는 코드이다.

if문 어셈블리

if문 실행부분의 어셈블리를 살펴보면 cmp와 jne 명령어가 있는것을 확인할수 있다. 먼저 cmp 전 부분을 보면 rbp 주소에서 8바이트 만큼 뺀 주소의 값(num1)을 eax 레지스터에 저장한다. 그다음 cmp 구문에서 rbp 주소에서 4바이트 만큼 뺀주소의 값(num2)를 eax에 넣어서 비교한다.

```
0x000055555555460c <+18>:    mov    -0x8(%rbp),%eax
0x000055555555460f <+21>:    cmp    -0x4(%rbp),%eax
0x0000555555554612 <+24>:    jne    0x555555554618 <main+30>
0x0000555555554614 <+26>:    addl   $0x1,-0x8(%rbp)
```

```
int num1 = 1;
int num2 = 1;

if (num1 == num2)
{
    num1++;
}
```

if문 어셈블리

그렇다면 cmp는 어떻게 동작하는지 알아보자.

0	0x0001	CF	Carry flag	Status
1	0x0002		Reserved, always 1 in **EFLAGS** [2][3]	
2	0x0004	PF	Parity flag	Status
3	0x0008		Reserved[3]	
4	0x0010	AF	Adjust flag	Status
5	0x0020		Reserved[3]	
6	0x0040	ZF	Zero flag	Status
7	0x0080	SF	Sign flag	Status
8	0x0100	TF	Trap flag (single step)	Control
9	0x0200	IF	Interrupt enable flag	Control
10	0x0400	DF	Direction flag	Control
11	0x0800	OF	Overflow flag	Status
12-13	0x3000	IOPL	I/O privilege level (286+ only), always 1 [*clarification needed*] on 8086 and 186	System
14	0x4000	NT	Nested task flag (286+ only), always 1 on 8086 and 186	System
15	0x8000		Reserved, always 1 on 8086 and 186, always 0 on later models	

cmp는 먼저들어온 값과 다음에 들어온 값을 빼서 상태레지스터 값들을 업데이트한다.

이 중에서 ZF 비트와 SF 비트를 업데이트한다. 만약 서로 뺀 값이 같으면 ZF비트가 1로되고 뺀값이 음수면 SF비트가 1로 세팅된다.

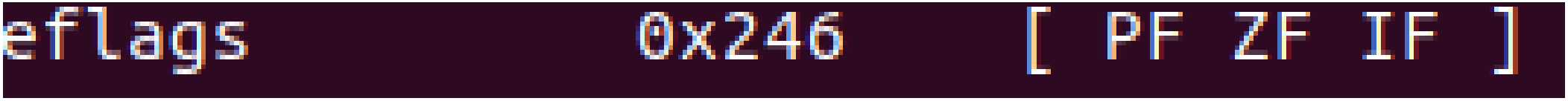
그리고 뺀값이 양수인경우 ZF, SF 둘다 0으로 유지된다.

출처 : 위키

if문 어셈블리

이전 슬라이드의 cmp구문 실행시 우리의 코드는 num1과 num2가 같기에 ZF가 1로 세팅된다.

아래 그림을 보면 eflags가 상태레지스터인데 ZF 비트가 1로되어 대괄호안에 ZF가 생긴것을 볼수 있다.



if문 어셈블리

다음으로 jne명령어를 보자 jne명령어는 구문을 점프하는 명령어인데 점프하는 명령어는 여러가지가 있다. 그중에서 jne는 ZF가 0이 아닐때 그러니까 비교하는 값이 서로 다를때 점프하는 구문이다.

우리코드에서는 같지 않을시 0x555555554618 로 점프하는데 이는 addl 다음구문이므로 같지않을시 if문 안의 코드를 실행하지 않는것을 확인할 수 있다.

```
0x000055555555460c <+18>:    mov    -0x8(%rbp),%eax
0x000055555555460f <+21>:    cmp    -0x4(%rbp),%eax
0x0000555555554612 <+24>:    jne    0x555555554618 <main+30>
0x0000555555554614 <+26>:    addl   $0x1,-0x8(%rbp)
```

if문 어셈블리

마찬가지로 num1과 num2 의 크기를 비교하는 구문이며 jle는 비교하는게 크지 않으면 실행되는 구문이다.

```
0x000055555554618 <+30>:    mov    -0x8(%rbp),%eax
0x000055555554618 <+33>:    cmp    -0x4(%rbp),%eax
0x000055555554618 <+36>:    jle    0x555555554624 <main+42>
0x000055555554620 <+38>:    addl   $0x2,-0x4(%rbp)
```

```
if (num1 > num2)
{
    num2 += 2;
}
```

if문 어셈블리

조건부 점프문은 다음과 같은 종류가 있다.

Instruction	Description	Instruction	Description
JO	Jump if overflow	JA JNBE	Jump if above Jump if not below or equal
JNO	Jump if not overflow	JL JNGE	Jump if less Jump if not greater or equal
JS	Jump if sign	JGE JNL	Jump if greater or equal Jump if not less
JNS	Jump if not sign	JLE JNG	Jump if less or equal Jump if not greater
JE JZ	Jump if equal	JG JNLE	Jump if greater Jump if not less or equal
JNE JNZ	Jump if not equl	JP JPE	Jump if parity Jump if parity even
JB JNAE	Jump if below Jump if not above or equal	JNP JPO	Jump if not parity Jump if parity odd

for문 어셈블리

for문이 실행될때 어셈블리어를 분석해보자 c 코드는 아래와 같다.

```c
int main(void)
{
    int i;
    int result = 0;

    for (i = 0; i < 10; i++)
    {
        result += 1;
    }

    return 1;

}
```

10번동안 변수 result에다가 1을 더하는 코드이다.

for문 어셈블리

어셈블리 코드를 살펴보자 jmp 부분부터 보면 0x555555554616로 분기하는것을 알수있다. 위의 주소로 가면 cmpl $0x09, -0x08(%rbp) 가 있는것을 볼수있는데 앞에서 if 문 공부한것과 비슷하다. 기존의 i 값은 0 이였는데 여기서 9를 뺀값은 -9이다. 그다음 jle가 실행되는데 jle는 eflag 레지스터의 ZF 또는 SF가 1일때실행된다.

현재 SF 비트가 1이므로 0x55555555460e로 분기하여 result와 i에 1을 더하는 명령을 수행하는것을 볼 수있다.

```
0x00005555555545fe <+4>:     movl   $0x0,-0x4(%rbp)
0x0000555555554605 <+11>:    movl   $0x0,-0x8(%rbp)
0x000055555555460c <+18>:    jmp    0x555555554616 <main+28>
0x000055555555460e <+20>:    addl   $0x1,-0x4(%rbp)
0x0000555555554612 <+24>:    addl   $0x1,-0x8(%rbp)
0x0000555555554616 <+28>:    cmpl   $0x9,-0x8(%rbp)
0x000055555555461a <+32>:    jle    0x55555555460e <main+20>
0x000055555555461c <+34>:    mov    $0x1,%eax
0x0000555555554621 <+39>:    pop    %rbp
0x0000555555554622 <+40>:    retq
```

while문 어셈블리

while문은 앞에서 구현한 for문과 같은 로직으로 구현하였으며 c코드는 아래와 같다.

```c
int main(void)
{
    int i = 0;
    int result = 0;

    while(i < 10)
    {
        result += 1;
        i += 1;
    }


    return 0;
}
```

10번동안 변수 result에다가 1을 더하는 코드이다.

while문 어셈블리

어셈블리 코드를 살펴보니 놀랍게도 앞에서 for문을 봤을때랑 같은것을 확인 할 수 있었다.

```
0x00005555555545fe <+4>:     movl   $0x0,-0x8(%rbp)
0x0000555555554605 <+11>:    movl   $0x0,-0x4(%rbp)
0x000055555555460c <+18>:    jmp    0x555555554616 <main+28>
0x000055555555460e <+20>:    addl   $0x1,-0x4(%rbp)
0x0000555555554612 <+24>:    addl   $0x1,-0x8(%rbp)
0x0000555555554616 <+28>:    cmpl   $0x9,-0x8(%rbp)
0x000055555555461a <+32>:    jle    0x55555555460e <main+20>
0x000055555555461c <+34>:    mov    $0x0,%eax
0x0000555555554621 <+39>:    pop    %rbp
0x0000555555554622 <+40>:    retq
```

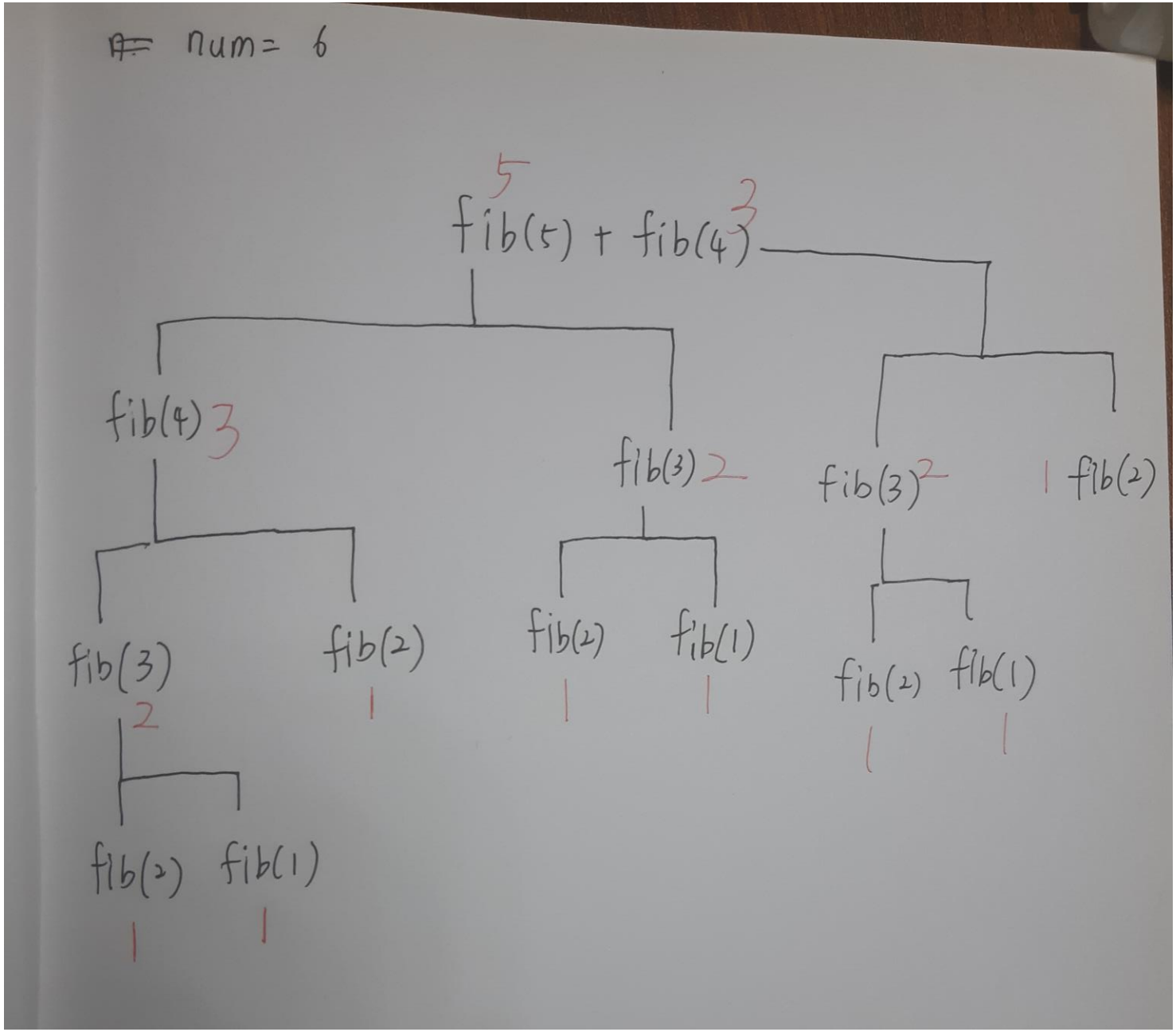
피보나치 수열

지난시간에 for문으로 풀었던 피보나치 수열문제를 이번에는 재귀함수를 이용하여 풀어보자
코드는 아래와 같다.

```c
int recursive_fib(int num)
{
    if(num <= 0)
    {
        printf("올바른 값을 입력하세요!\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;
}
```

Recursive_fib 함수를 보면 num이 3이상일때 재귀적으로 호출하는것을 확인 할 수 있다.

피보나치 수열


num = 6
fib(5) + fib(4)
fib(4)
fib(3)
fib(3)
fib(2)
fib(2)
fib(1)
fib(3)
fib(2)
fib(2)
fib(1)
fib(2)
fib(1)

피보나치 수열

피보나치수열의 재귀호출 관계를 트리 형태로 그려서 확인하였다. 이처럼 for문 대신 재귀호출을 이용하여 문제를 해결할수있다.

그러면 재귀호출의 단점이 없을까?

재귀 호출의 단점

우리가 사용하는 임베디드 시스템의 메모리사이즈는 일반 컴퓨터의 사이즈보다 훨씬작고 제한된 용량을 사용한다.

따라서 함수호출을 많이하는 재귀호출의 경우 아래의 스택프레임부분이 계속 호출된다.

이렇게되면 스택에 rbp레지스터 값이 계속 쌓이므로 스택낭비가 될수있고 재귀호출의 탈출구문을 잘못짤 경우 스택오버플로우가 되어버리는 경우가 발생할수있다.

```
0x000055555555476a <+0>:     push   %rbp
0x000055555555476b <+1>:     mov    %rsp,%rbp
0x000055555555476e <+4>:     push   %rbx
0x000055555555476f <+5>:     sub    $0x18,%rsp
```

질문

Q1. 재귀 호출시에 for문보다 성능(처리속도)면에서 뒤쳐지나요?