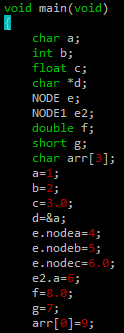
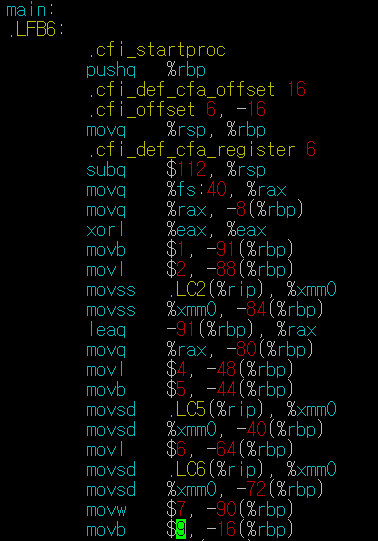
Proj\_20161595 배성현

**1. activation record 구조**

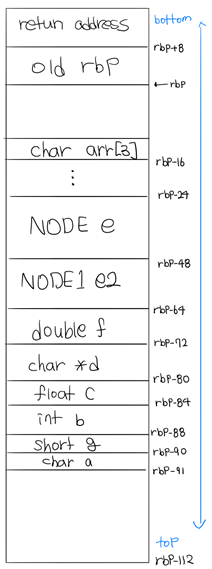


(main함수의 activation record, assembly code 일부, c code 일부)

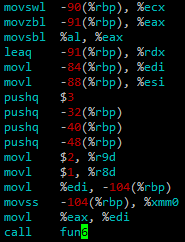
main함수의 local variable이 activation record에 어떻게 할당되는지 알아보기 위하여, 각 local variable에 각기 다른 수들을 초기화 하여, 어떤 변수가 activation record의 어느 위치에 할당되는지 알아보았다. Assembly를 기준으로 보게 되면 먼저, pushq %rbp를 통하여 왼쪽의 activation record에 old rbp를 넣어주게 된다. 그리고 나서 movq %rsp, %rbp를 해주어 %rbp를 스택포인터와 같게 갱신시켜준다. 그리고 나서 subq $112, %esp를 해주어 112byte만큼의 공간을 할당하여 준 이후에 아래와 같이 각 선언문에 해당하는 문장을 수행하게 된다.

|  |  |
| --- | --- |
| movb $1, -91(%rbp) | a=1; |
| movl $2, -88(%rbp) | b=2; |
| movss .LC2(%rip), %xmm0  movss %xmm0, -84(%rbp) | c=3.0; |
| leaq -91(%rbp), %rax  movq %rax, -80(%rbp) | d=&a; |
| movl $4, -48(%rbp) | e.nodea=4; |
| movb $5, -44(%rbp) | e.nodeb=5; |
| movsd .LC5(%rip), %xmm0  movsd %xmm0, -40(%rbp) | e.nodec=6.0; |
| movl $6, -64(%rbp) | e2.a=6; |
| movsd .LC6(%rip), %xmm0  movsd %xmm0, -72(%rbp) | f=8.0; |
| movw $7, -90(%rbp) | g=7; |
| movb $9, -16(%rbp) | arr[0]=9; |

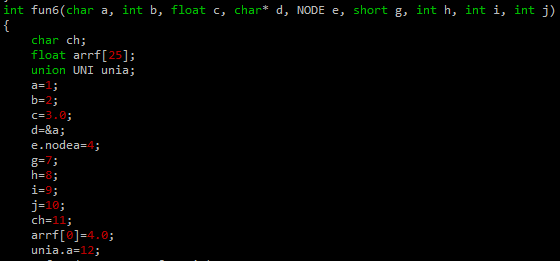
따라서 위의 표에 대응되게 activation record를 그리면 아래와 같다.



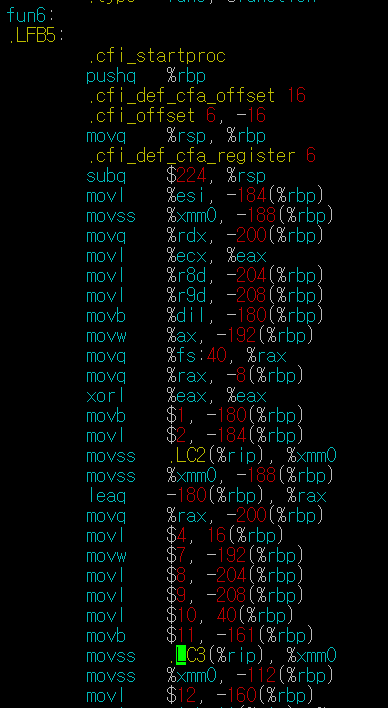
위의 activation record를 보게 되면 main함수의 local variable이 배열을 제외하고는 선언 순서와 관계 없이 자료형의 크기 순서대로 activation record에 할당이 된다는 것을 알 수 있다. 가장 자료형의 크기가 작은 char이 위의 그림에서 밑부분(activation record의 top과 가까운 부분)에 할당이 되어있고, 구조체로 만들어 가장 자료형의 크기가 큰 NODE가 activation record의 bottom과 가까운 부분에 할당이 되어있는 모습을 볼 수 있다. 이 때 배열은 local variable중에 가장 윗부분(activation record의 bottom과 가까운 부분)에 할당이 되어있는 모습 역시 볼 수 있다. 즉 **배열을 제외하고는 local variable의 자료 형의 크기가 작을수록 rbp에서 먼 곳(activation record의 top과 가까운 부분)에 할당되고, 자료 형의 크기가 클수록 rbp에서 가까운 곳(activation record의 bottom과 가까운 부분)에 할당이 되게 된다. 또 배열의 경우 local variable들중에 rbp에서 가장 가까운 곳에 할당되게 된다.**



위의 코드는 main함수에서 fun6을 호출하는 어셈블리 코드의 일부이다. 코드를 보게 되면 parameter로 넘겨주게 되는 어떠한 변수의 값들은 register에 저장되어, 또 다른 어떠한 변수의 값들은 stack에 push되어 passing되게 된다.(자세한 설명은 3번 문항). 그리고 나서 가장 밑에 있는 call fun6을 수행하게 되는데 이 때 fun6함수를 종료했을 때 돌아와야 하는 main함수상의 부분 즉 fun6 함수 activation record의 return address를 stack에 push하게 된다.



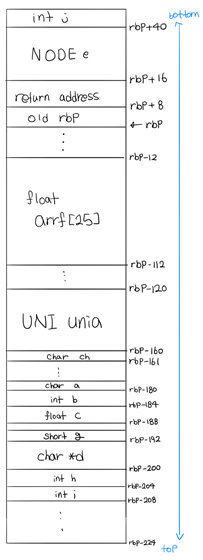
(fun6함수 c 코드의 일부)

 (위에 대응하는 fun6함수 어셈블리코드의 일부)

위에서 main함수에서 fun6을 호출하기 전에 parameter를 레지스터와 스택에 넣어주는 것과 call fun6을 통하여 return address를 스택에 넣어주는 것을 통해, fun6의 **activation record의 bottom에 가장 가까운 부분에 stack을 이용하여 passing된 parameter들이 있고, 그 이후에 return address가 저장되어 있다는 것을 알 수 있다.** 그리고 나서 fun6이 계속 진행되게 되는데, return address의 밑에 rbp를 push해준다. **이 때 rbp를 push해서 stack에 저장하여 주는 과정이 caller(main함수)의 stack에서 위치를 알 수 있도록 하여주는 dynamic link에 해당하게 된다**. rbp에 rsp의 값을 저장해주고, rsp를 224만큼 빼서 fun6함수 activation record를 위한 공간을 할당하는 것을 확인할 수 있다. 그리고 나서 이제 register에 저장해놨던 나머지 parameter의 값들을 stack의 특정한 공간으로 옮기는 것을 확인할 수 있다. 이를 또 직관적으로 알아보기 위하여, 각 parameter와 local variable에 각기 다른 수들을 초기화 하여, 나머지 parameter와 변수들이 activation record의 어느 위치에 할당되는지 알아보았다.

|  |  |
| --- | --- |
| movb $1, -180(%rbp) | a=1; |
| movl $2, -184(%rbp) | b=2; |
| movss .LC2(%rip), %xmm0  movss %xmm0, -188(%rbp) | c=3.0; |
| leaq -180(%rbp), %rax  movq %rax, -200(%rbp) | d=&a; |
| movl $4, 16(%rbp) | e.nodea=4; |
| movw $7, -192(%rbp) | g=7; |
| movl $8, -204(%rbp) | h=8; |
| movl $9, -208(%rbp) | i=9; |
| movl $10, 40(%rbp) | j=10; |
| movb $11, -161(%rbp) | ch=11; |
| movss .LC3(%rip), %xmm0  movss %xmm0, -112(%rbp) | arrf[0]=4.0; |
| movl $12, -160(%rbp) | unia.a=12; |

따라서 위의 표에 대응되게 activation record를 그리면 아래와 같다.



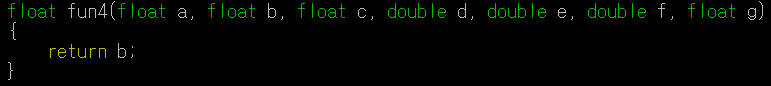
위의 activation record의 local variable만 보았을 때 char ch, union UNI unia, float arrf[25]가 위의 main함수에서와 마찬가지로 local variable의 자료형의 크기가 작을수록 현재 rbp에서 먼 곳(activation record의 top과 가까운 부분)에 할당되고, 자료형의 크기가 클수록 rbp에서 가까운 곳(activation record의 top과 먼 부분)에 할당이 되며 또 배열의 경우 local variable중 rbp에서 가장 가까운 곳에 할당된 것을 볼 수 있다. 또 **parameter의 경우에 caller에서 callee(fun6)을 call할 때에, stack에 push되어 전달된 parameter들은 activation record의 bottom부분에 존재하는 것을 확인 할 수 있고, register를 통하여 전달된 parameter들은 activation record의 top과 가까운 부분에 존재하는 것을 확인 할 수 있다. 이 때 register를 통하여 전달된 parameter들은 위의 short g와 같이 word alignment에 의하여 생긴 남는 공간에 넣어준 것과 같은 케이스를 제외하고는 인자의 순서대로 저장되게 된다. 따라서 이를 통하여 bottom->top순으로 정리하여 보자면 activation record의 가장 bottom에 stack에 push되어 전달된 parameter가 할당되고, 그 다음으로 return address, 그 다음으로 dynamic link(rbp), local variable(지역 변수), register를 통하여 전달된 parameter가 할당되게 된다.**



 (fun6 함수의 마지막 부분)

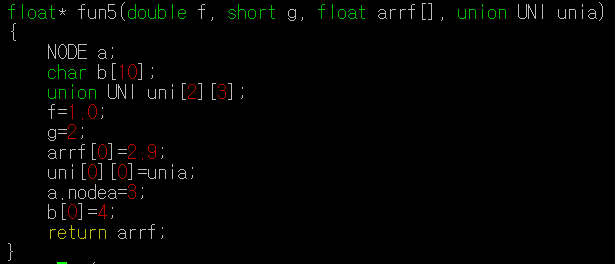
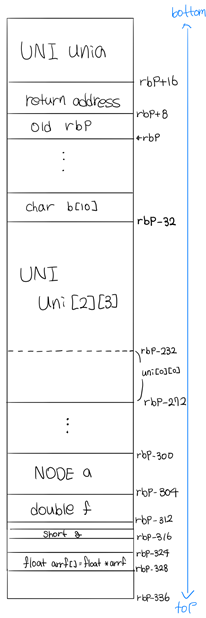
 (이에 대응하는 assembly code)

정수형 return value는 어느 공간에 저장되어 return 되는지에 관하여 확인하기 위하여 위의 fun6의 return부분을 보게 되면 fun6은 위와 같이 b+e.nodea+5를 return하게 된다. 다음으로 어셈블리 코드를 보게 되면 위의 activation record에서 보았듯이 16(%rbp)에는 e.nodea가 담겨있는데 이를 edx레지스터로 옮겨준 것을 확인할 수 있다. 그리고 나서 -184(%rbp)를 eax에 옮겨주게 되는데 위의 activation record에 따르면 -184(%rbp)는 b에 해당하게 된다. 따라서 현재 eax레지스터에 b의 값이 담겨있고, edx레지스터에 e.nodea의 값이 담겨있는 것을 알 수 있는데 그 다음 문장인 addl %edx, %eax를 통하여 eax레지스터에 b+e.nodea가 담기게 되고 addl $5, %eax를 해줌으로써 최종적으로 eax레지스터에 return에 해당하는 b+e.nodea+5의 값이 저장되게 된다. **따라서 이를 통하여 정수형의 return value는 최종적으로 eax레지스터에 저장되어 있게 된다는 것을 확인할 수 있다.**

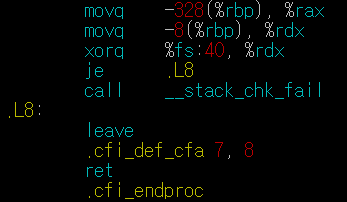




실수형 return value는 어느 공간에 저장되어 return 되는지에 관하여 확인하기 위하여 위의 fun4의 return부분을 보게 되면 fun4는 위와 같이 b를 return하게 된다. 그리고 이러한 b는 fun4에서 rbp를 기준으로 -8(%rbp)에 저장되어 있다. 따라서 movss -8(%rbp), %xmm0문장을 통하여 return 값인 b를 최종적으로 xmm0레지스터에 저장되게 된다. **따라서 이를 통하여 실수형의 return value는 최종적으로 xmm0레지스터에 저장되어 있게 된다는 것을 확인할 수 있다.**

(fun5의 c code와 activation record)



포인터 형 return value는 어느 공간에 저장되어 return 되는지에 관하여 확인하기 위하여 위의 fun5의 return부분을 보게 되면 fun5는 위와 같이 arrf의 주소를 return하게 된다. 그리고 위의 activation record에서 볼 수 있듯이 arrf의 주소는 fun5에서 rbp를 기준으로 -328(%rbp)에 저장되어 있다. 따라서 movq -8(%rbp), %rax문장을 통하여 return 값인 arrf의 주소를 최종적으로 rax레지스터에 저장되게 된다. **따라서 이를 통하여 포인터형의 return value는 최종적으로 rax레지스터에 저장되어 있게 된다는 것을 확인할 수 있다.**

이제 전체를 종합하여 보면, **bottom->top순으로 activation record의 가장 bottom부분으로 stack에 push되어 전달된 parameter가 할당되고, 그 다음으로 return address, 그 다음으로 dynamic link(rbp), local variable(지역 변수), register를 통하여 전달된 parameter가 할당되게 된다. 이 때 register를 통하여 전달된 parameter들은 word alignment에 의하여 생긴 남는 공간에 들어갈 수 있는 인자를 할당해준 것과 같은 케이스를 제외하고는 인자의 순서대로 저장된다. 또 local variable은 자료형의 크기가 작을수록 activation record의 top과 가까운 부분에 할당되고, 자료형의 크기가 클수록 bottom과 가까운 부분에 할당이 되며 배열의 경우 local variable중 bottom과 가장 가까운 부분에 할당이 되게 된다. 마지막으로 return value의 경우에는 정수형의 return value는 최종적으로 eax레지스터에 저장되어 있게 되며 실수형은 xmm0레지스터에 저장되어 있게 되며, 포인터형의 return value는 rax(eax계열)에 저장되게 된다.**

**2. Subprogram상에서 parameter 및 local 변수, global 변수 참조방법**

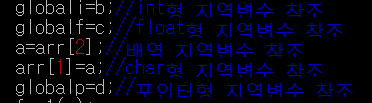
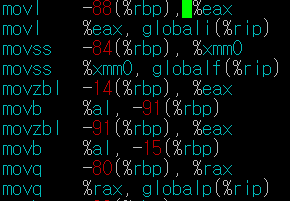
먼저 parameter의 참조를 알아보기 위하여 fun6함수 안에 아래와 같은 statement를 집어 넣었다.



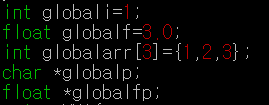
(fun6함수에서 parameter를 참조하는 c코드 및 assembly코드)

위의 assembly코드를 보게 되면 **parameter 참조의 경우에는 rbp를 기준으로 activation record상에서 해당 parameter의 위치에 있는 값을 해당 parameter의 타입에 해당하는 register(정수형의 경우 eax, 실수형의 경우 xmm0, 포인터형의 경우 rax)로 옮겨준 뒤, register에서 저장하고자 하는 변수의 위치로 옮겨주는 방식으로 참조가 일어난다는 것을 알 수 있다.**

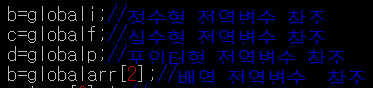
다음으로 local variable의 참조를 알아보기 위하여 main함수 안에 아래와 같은 statement를 집어 넣었다.

위의 assembly코드를 보게 되면 **local variable의 참조 역시 parameter참조와 마찬가지로 rbp를 기준으로 activation record상에서 해당 local variable의 위치에 있는 값을 해당 local variable의 타입에 해당하는 register(정수형의 경우 eax, 실수형의 경우 xmm0, 포인터형의 경우 rax)로 옮겨준 뒤, register에서 저장하고자 하는 변수의 위치로 옮겨주는 방식으로 참조가 일어난다는 것을 알 수 있다.**

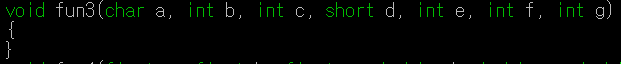
 (global 변수 선언부)

다음으로 global 변수의 참조를 알아보기 위하여 fun6함수 안에 아래와 같은 statement를 집어 넣었다.



위의 assembly코드를 보게 되면 전역 변수는 stack영역에 있는 것이 아닌 데이터영역에 있기 때문에 rip를 이용하여 참조하게 된다. 따라서 **global 변수의 참조는 rip레지스터를 기준으로 해당 global변수의 위치에 있는 값을 해당 해당 global변수의 타입(정수형의 경우 eax, 실수형의 경우 xmm0, 포인터형의 경우 rax)에 해당하는 register로 옮겨준 뒤, register에서 저장하고자 하는 변수로 옮겨준다는 것을 알 수 있으며, 배열 전역변수의 경우에는 index와 자료형을 이용하여 참조에 필요한 값의 위치를 계산해 rip레지스터를 기준으로 참조하여 배열의 타입에 해당하는 register(정수형의 경우 eax, 실수형의 경우 xmm0, 포인터형의 경우 rax)로 옮겨준 뒤, register에서 저장하고자 하는 변수로 옮겨주는 것을 알 수 있다.**

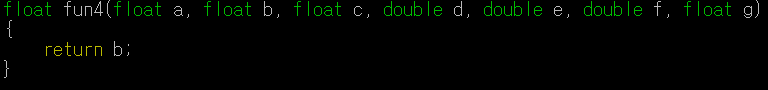
**3. 다양한 데이터 타입에 대한 Parameter Passing 방식:**

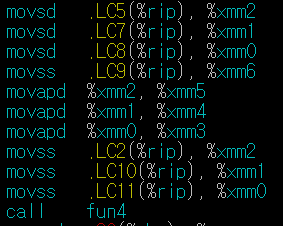




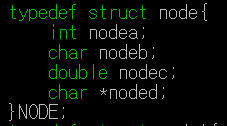
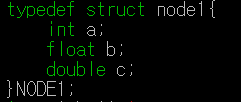
 (main함수 어셈블리 코드 안의 fun3 호출부분)

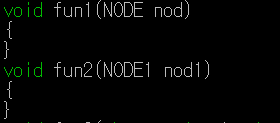
정수형 자료형의 parameter passing을 테스트하기 위하여 위와 같은 함수를 작성하여 main에서 호출하였다. 함수를 보았을 때 정수형 자료형인 char, short, int의 경우 앞의 6개의 인자에 있어서는 레지스터에 값을 복사하여 parameter passing을 하게 되고, 6번째 이후의 인자부터는 stack에 값을 push해서 parameter로 넘기게 되는 것을 확인 할 수 있다. **정수형의 경우에는, 정수형과 포인터 형을 포함하여 인자 1~6번째 parameter는 register에 저장하여 parameter를 passing하고 7번째부터는스택에 push하여 parameter를 passing하게 된다.**

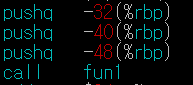
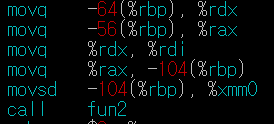


(main함수 어셈블리 코드 안의 fun4 호출부분)

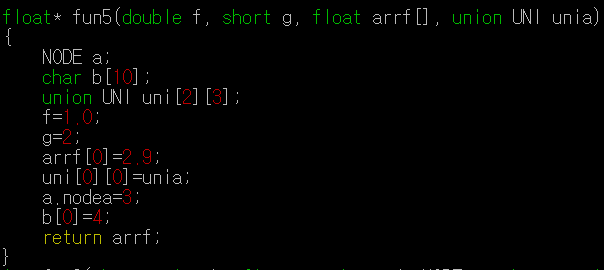
실수형 자료형의 parameter passing을 테스트하기 위하여 위와 같은 함수를 작성하여 main에서 호출하였다. 함수를 보았을 때 **실수형 자료형인 float, double의 경우 실수형 레지스터에 값을 복사하여 parameter passing을 하는 것을 확인 할 수 있다.** 정수형 자료형의 경우 7번째 인자부터는 stack에 push되는 것을 볼 수 있었지만 실수형 자료형인 float, double의 경우 7번째 인자 역시도 register에 들어간 것을 확인 할 수 있다.

 (구조체 선언)

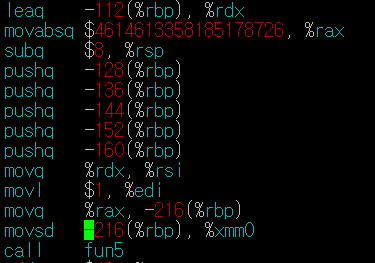
 (구조체 호출)

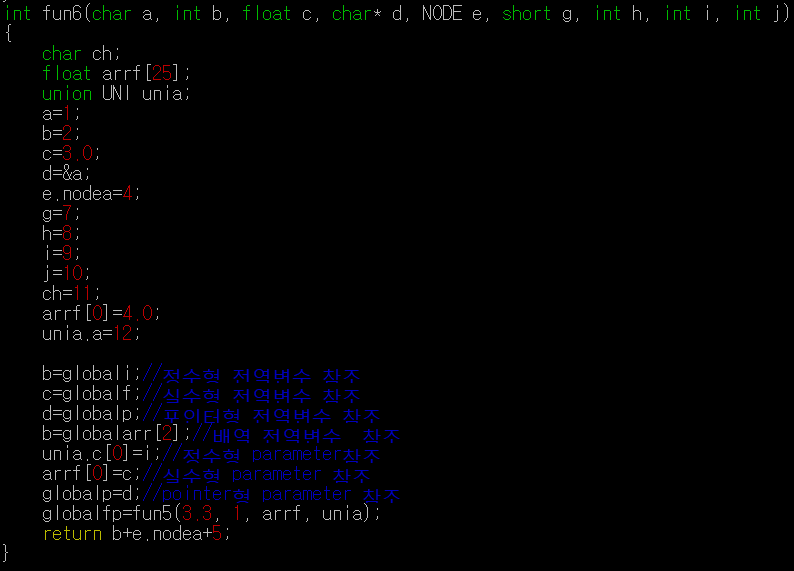
구조체의 parameter passing을 테스트하기 위하여 위와 같은 함수 두 개를 작성하여 main에서 호출하였다. 이를 보았을 때에, fun1과 fun2를 비교하여 보면 fun1에서는 24byte의 NODE구조체를 인자로 넘겨주게 되고, fun2에서는 16byte의 NODE1 구조체를 인자로 넘겨주게 된다. 이를 어셈블리 코드로 보면 fun1에서는 NODE를 8바이트씩 쪼개 stack에 push하여 parameter를 passing한 반면에, fun2에서는 NODE를 8바이트씩 쪼개 register로 값을 옮겨서 parameter를 passing하는 것을 확인 할 수 있다. 따라서 **NODE와 같은 structure에서 24byte 이상인 경우에는 stack에 push하여 parameter를 passing하게 되고, 그 미만인 경우에는 register로 값을 복사하여 parameter를 passing하게 된다.**

 (fun5 함수)

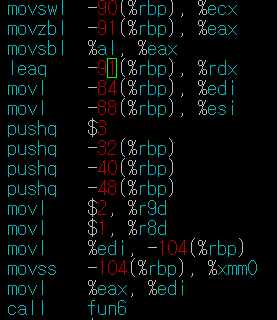
 (fun6함수에서 fun5 함수 호출)

 (fun6에서 fun5를 호출하는 assembly코드)

Array와 union의 parameter passing을 테스트하기 위하여 fun5함수를 만들어 fun6에서 fun5함수를 호출하였다. 위의 fun6의 activation record에서 확인하였듯이 로컬 배열 arrf는 -112(%rbp)부터 값을 저장하게 된다. 그리고 위의 fun5를 호출하는 assembly 코드를 보면 이러한 arrf의 첫 주소인 -112(%rbp)를 복사하여 레지스터에 넣는 것을 볼 수 있다. 따라서 **Array는 array의 시작주소를 레지스터에 저장하여 array의 시작주소를 passing 하게 되고, 배열을 넘기는 것은 float \*arrf의 포인터로 시작주소를 passing하는 것과 같다는 것을 알 수 있다.** Union unia 역시 위의 activation record에서 -160(%rbp)부터 40byte를 차지하는 것을 볼 수 있었다. 따라서 여기서는 8바이트씩 5번 stack에 넣어주어 stack을 통하여 parameter를 passing함을 확인 할 수 있다. **UNION 역시 Structure와 마찬가지로 크기가 24byte가 넘어가게 되면 stack에 넣어서 passing 하게 되고, 24byte보다 작으면 register에 값을 복사하여 passing하게 된다.**

 (fun6 함수)

 (main에서 fun6함수 호출)

 (main에서 fun6을 호출하는 어셈블리 코드)

위의 main의 activation record에서 확인하였듯이 로컬 변수 char a는 -91(%rbp)에 저장되어 있다. 위에서 포인터 형인 char \*형으로 char a의 주소 &a를 인자로 넘겨주는 것을 보게 되면 leaq -91(%rbp), %rdx이라고 되어 레지스터에 주소 값을 복사하는 것을 볼 수 있다. 따라서 **pointer형은 변수에 해당하는 주소 값을 레지스터에 복사하여(레지스터를 이용하여) parameter passing을 하게 되는데 정수형을 포함하여 인자 1~6번째 parameter는 register에 저장하여 parameter를 passing하고 7번째부터는 스택에 push하여 parameter를 passing하게 된다.** 위의 fun6함수의 인자를 보게 되면 정수형과 포인터형의 개수의 합이 총 7개인 것을 확인할 수 있다. 따라서 이 중의 마지막 인자인 int j로 passing되는 3이 stack에 push되어 전달되는 것을 확인 할 수 있다.

**4. Return value 전달 방식 분석**

위의 1번 문항에서 정수형을 return하는 callee함수의 return value는 최종적으로 eax 레지스터에 저장되어 있게 된다고 하였다. 따라서 이러한 eax레지스터에 저장되어 있는 return value가 어떻게 caller함수로 전달되기 위해 알아보기 위해 fun6(callee)의 caller인 main함수에서 함수의 return값을 넘겨받는 과정을 보았다.

(main에서의 fun6의 return 값은 b에 저장된다.)



위의 어셈블리코드에서, addq $32, %rsp를 하여 stack으로 넘겨주었던 parameter에 관한 부분을 회수하고, eax에 있는 값을 -88(%rbp)[main함수의 local variable b]로 옮겨주는 것을 확인 할 수 있다. **따라서 정수형의 경우 callee함수가 return 되기 직전에 return value를 eax레지스터에 넣어주고, 다시 caller함수로 흐름이 넘어왔을 때 바로, 이 eax레지스터의 값(return value)을 변수로 옮겨주는 방식으로 return value가 전달되게 된다.**

또 실수형을 return하는 callee함수의 return value는 최종적으로 xmm0레지스터에 저장되어 있게 된다고 하였다. 따라서 이러한 xmm0레지스터에 저장되어 있는 return value가 어떻게 caller함수로 전달되기 위해 알아보기 위해 fun4(callee)의 caller인 main함수에서 함수의 return값을 넘겨받는 과정을 보았다.





위의 어셈블리코드에서, xmm0레지스터에 있는 값을 eax레지스터로 옮겨주고 또 eax를 포인터형 변수 c에 해당하는 -84(%rbp)에 넣어주는 것을 확인 할 수 있다. **따라서 실수형의 경우 callee함수가 return 되기 직전에 return value를 xmm0레지스터에 넣어주고, 다시 caller함수로 흐름이 넘어왔을 때 바로, 이 xmm0레지스터의 값을 eax레지스터로 옮기고 난 후 eax레지스터를 실수형의 변수로 옮겨주는 xmm0레지스터에서 eax레지스터를 경유하여 변수에 저장되는 방식으로 return value가 전달되게 된다.**

또 포인터형을 return하는 callee함수의 return value는 최종적으로 rax레지스터에 저장되어 있게 된다고 하였다. 따라서 이러한 rax레지스터에 저장되어 있는 return value가 어떻게 caller함수로 전달되기 위해 알아보기 위해 fun5(callee)의 caller인 fun6함수에서 함수의 return값을 넘겨받는 과정을 보았다.





위의 어셈블리코드에서, addq $48, %rsp를 하여 stack으로 넘겨주었던 parameter에 관한 부분을 회수하고, rax에 있는 값을 globalfp(%rip)[global variable]로 옮겨주는 것을 확인 할 수 있다. **따라서 포인터형의 경우 callee함수가 return 되기 직전에 return value(주소 값)를 rax레지스터에 넣어주고, 다시 caller함수로 흐름이 넘어왔을 때 바로, 이 rax레지스터의 값(주소값)을 포인터형의 변수로 옮겨주는 방식으로 return value가 전달되게 된다.**