

Assembly Code in C

尹嘉权 3120000419 计科1205

Problem Description

Many language rules are checked by the compiler, and it is possible to bypass the rules using assembly language after compilation. Consider the following C program:

```
#include <stdio.h>
int x=3;
int main(void) {
    int x=5;
    printf("x = %d", x);
    return 0;
}
```

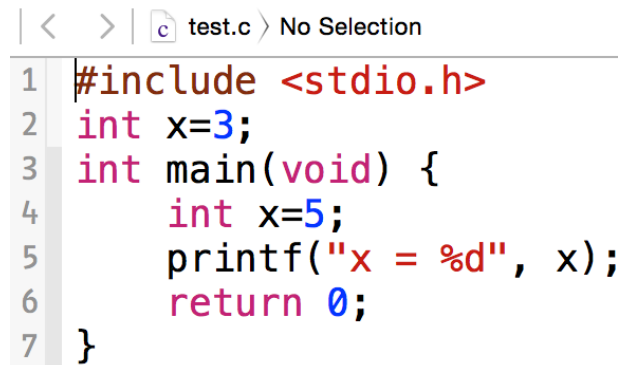
Compile the program and generate its assembly code.

Understand the assembly code and modify it to let the program print the global variable x instead of the local variable x.

Submit a report discussing the generated assembly code and your modification. Explain your way in detail.

Step 0. Compile the program, and generate assembly code

Input the code and save as test.c

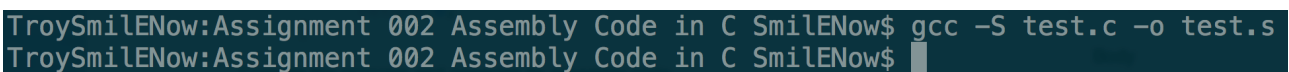
A screenshot of a code editor window. The title bar shows a file icon, 'test.c', and '> No Selection'. The editor contains the following C code:

```
1 #include <stdio.h>
2 int x=3;
3 int main(void) {
4     int x=5;
5     printf("x = %d", x);
6     return 0;
7 }
```

Then open the terminal, cd to the current directory, and input

```
gcc -S test.c -o test.s
```

to generate the assembly code

A screenshot of a terminal window. The prompt is 'TroySmileNow:Assignment 002 Assembly Code in C SmileNow\$'. The command 'gcc -S test.c -o test.s' has been entered and executed. The prompt is now 'TroySmileNow:Assignment 002 Assembly Code in C SmileNow\$' followed by a cursor.

```
TroySmileNow:Assignment 002 Assembly Code in C SmileNow$ gcc -S test.c -o test.s
TroySmileNow:Assignment 002 Assembly Code in C SmileNow$
```

Open test.s, to check it.

```

1      .section      __TEXT,__text,regular,pure_instructions
2      .globl _main
3      .align 4, 0x90
4      _main:                                     ## @main
5          .cfi_startproc
6      ## BB#0:
7          pushq %rbp
8      Ltmp2:
9          .cfi_def_cfa_offset 16
10     Ltmp3:
11         .cfi_offset %rbp, -16
12         movq %rsp, %rbp
13     Ltmp4:
14         .cfi_def_cfa_register %rbp
15         subq $16, %rsp
16         leaq L_.str(%rip), %rdi
17         movl $0, -4(%rbp)
18         movl $5, -8(%rbp)
19         movl -8(%rbp), %esi
20         movb $0, %al
21         callq _printf
22         movl $0, %esi
23         movl %eax, -12(%rbp)      ## 4-byte Spill
24         movl %esi, %eax
25         addq $16, %rsp
26         popq %rbp
27         retq
28     .cfi_endproc
29
30     .section      __DATA,__data
31     .globl _x                                     ## @x
32     .align 2
33     _x:
34         .long 3                                     ## 0x3
35
36     .section      __TEXT,__cstring,cstring_literals
37     L_.str:                                         ## @.str
38         .asciz "x = %d"
39
40
41     .subsections_via_symbols
42

```

Step 1. Analyse the assembly code

Because I haven't studied X86 assembly, especially AT&T assembly, so it's hard for me to understand the assembly code that I generated before.

So I need to learn some knowledge that relates AT&T and do more experiments.

Here is the website link I have studied:

<http://cs.nyu.edu/courses/fall11/CSCI-GA.2130-001/x64-intro.pdf>

In the test.s, %rsp and %rbp represents the stack pointer and base pointer, respectively. When accessing main, saving rbp to rsp, and subtract rbp to generate the space for main function.

And we can refer lead L_.str(%rip), %rdi in Line 16:

16

```
leaq L_.str(%rip), %rdi
```

In my understanding, %rip is a pointer, or a register which saves the data items of global data area. Because L_.str has been defined in the last of the assembly code:

```
37 L_.str:                                     ## @.str
38     .asciz "x = %d"
```

And it shows as the string in the printf (referencing to the test.c Line5) :

```
printf("x = %d", x);
```

Then, let's break the assembly code by Line 21:

```
callq _printf
```

Before Line21,

```
subq $16, %rsp // subtract rsp to generate the space for main
```

leaq L_.str(%rip), %rdi // load the string of printf, from the data area defined at the last of assembly code

```
movl $0, -4(%rbp) // make -4(%rbp) to immediate 0
movl $5, -8(%rbp) // make -8(%rbp) to immediate 5
// It shows the local variable x stored in
-8(%rbp), and x = 5, with this assembly code.
```

```
movl -8(%rbp), %esi // Here, %esi is the input parameter of
printf, and we put the value of local variable x to the parameter
```

```
movb $0, %al // make register %al to immediate 0
```

In order to print the global variable x, instead of the local variable x, we must change the input parameter of printf, which means we need to change the left-head of movl:

```
19     movl    -8(%rbp), %esi
```

After callq _printf(Line 21):

```
movl $0, %esi // make %esi to immediate 0
```

```
movl %eax, -12(%rbp) ## 4-byte Spill
// %eax represents the return value of
printf, and save to -12(%rbp)
```

```
movl %esi, %eax // make %eax to immediate 0
```

```
// restore rsp and return
addq $16, %rsp
popq %rbp
retq
```

After calling printf, it seems that we don't need to change anything. Because I am not sure for this, I write another code to prove my idea, especially the line after printf, and justify whether %eax is the return value of printf.

```
1  #include <stdio.h>
2  int x=3;
3  int main(void) {
4      int x=5;
5      int ret = printf("x = %d", x);
6      printf("ret = %d",ret);
7      return 0;
8  }
```

In the terminal we print
gcc -S test_ret.c -o test_ret.s

And check the assembly code of test_ret.c

```
13  Lttmp4:
14      .cfi_def_cfa_register %rbp
15      subq    $16, %rsp
16      leaq    L_.str(%rip), %rdi
17      movl    $0, -4(%rbp)
18      movl    $5, -8(%rbp)
19      movl    -8(%rbp), %esi
20      movb    $0, %al
21      callq   _printf
22      leaq    L_.str1(%rip), %rdi
23      movl    %eax, -12(%rbp)
24      movl    -12(%rbp), %esi
25      movb    $0, %al
26      callq   _printf
27      movl    $0, %esi
28      movl    %eax, -16(%rbp)      ## 4-byte Spill
29      movl    %esi, %eax
30      addq    $16, %rsp
31      popq    %rbp
32      retq
33      .cfi_endproc
34
35      .section    __DATA,__data
36      .globl    _x      ## @x
37      .align    2
38  _x:
39      .long    3      ## 0x3
40
41      .section    __TEXT,__cstring,cstring_literals
42  L_.str:      ## @.str
43      .asciz    "x = %d"
44
45  L_.str1:      ## @.str1
46      .asciz    "ret = %d"
```

In this code, we prove that %eax is exactly the return value of printf. Because we use ret, local variable int, to represents the return value of printf, with

```
int ret = printf("x = %d", x);
```

And in test_ret.s, Line23 - 24,

```
movl %eax, -12(%rbp) // save the return value of printf to int
ret, and -12(%rbp) is local variable int, ret
```

```
movl -12(%rbp), %esi // put the value of ret to the parameter of
next printf, then call the second printf
```

After calling the second printf, we also get its return value of a new %eax, and save it to -16(%rbp):

Line 27 - 29:

```
movl $0, %esi
movl %eax, -16(%rbp)      ## 4-byte Spill
movl %esi, %eax
```

And another piece of good news is that, we find out the way of using, or representing the data area. And in this way we can represent the global variable x, then by changing the input parameter of printf, we can print the global variable x instead of the local variable x.

```
13  Ltmp4:
14      .cfi_def_cfa_register %rbp
15      subq $16, %rsp
16      leaq L_str(%rip), %rdi
17      movl $0, -4(%rbp)
18      movl $5, -8(%rbp)
19      movl -8(%rbp), %esi
20      movb $0, %al
21      callq _printf
22      leaq L_str1(%rip), %rdi
23      movl %eax, -12(%rbp)
24      movl -12(%rbp), %esi
25      movb $0, %al
26      callq _printf
27      movl $0, %esi
28      movl %eax, -16(%rbp)      ## 4-byte Spill
29      movl %esi, %eax
30      addq $16, %rsp
31      popq %rbp
```

It seems that we can represent the global data with _Label(%rip) !

So we return the first program and its assembly code, and change assembly code from

Line19: movl -8(%rbp), %esi

to

Line19: movl _x(%rip), %esi

```

13 Ltmp4:
14     .cfi_def_cfa_register %rbp
15     subq    $16, %rsp
16     leaq    L_.str(%rip), %rdi
17     movl    $0, -4(%rbp)
18     movl    $5, -8(%rbp)
19     movl    _x(%rip), %esi
20     movb    $0, %eax
21     callq   _printf
22     movl    $0, %esi
23     movl    %eax, -12(%rbp)    ## 4-byte Spill
24     movl    %esi, %eax
25     addq    $16, %rsp
26     popq    %rbp
27     retq

```

By changing it, we compile the assembly code to execute file, and execute it.

```

TroySmileNow:Assignment 002 Assembly Code in C SmileNow$ gcc test.s -o test
TroySmileNow:Assignment 002 Assembly Code in C SmileNow$ ./test
x = 3TroySmileNow:Assignment 002 Assembly Code in C SmileNow$ █

```

Happily we print the global variable x (x = 3) , instead of the local variable x (x=5).