Tango with code

A blog about frustration and anger

Mixing Assembly and C

Feb 27th, 2016 10:24 am | Comments

In many applications, mixing Assembly and C is routine (pun intended). There are many reasons for it, but, in general, you want to use Assembly when you want to deal with the hardware directly or perform a task with maximum speed and minimum use of resources, while you use C to perform some high level stuffs that don’t attend the former requirements. In either case, you’ll need one integrated system.

There are three ways to mix Assembly and C:

- Using Assembly-defined functions into C
- Using C-defined functions into Assembly
- Using Assembly code in C

We’ll explore them all in this tutorial.

Using Assembly-defined functions in C

Let’s first take the example of a function that takes no parameters and doesn’t return anything, like one that just prints something on screen.

hello_world.s

```
.globl hello_world
.type hello_world, @function
.section .data
message: .ascii "Hello, World!\n"
length: .quad . - message
.section .text
hello_world:
  mov $1, %rax
  mov $1, %rdi
  mov $message, %rsi
  mov length, %rdx
  syscall
  ret
```

(If you don't quite understand the above syntax, read my previous tutorial)

Now let's create a C program to call this function:

hello_world.c

```
extern void hello_world();

int main()
{
  hello_world();
  return 0;
}
```

Notice the use of `extern` keyword. It tells the compiler that the definition of a given function or variable is defined in somewhere else other than the current file. It's the **linker** job to connect this declaration with the actual definition.

Now let's compile and link our both programs at the same time in order to obtain an executable file:

```
gcc hello_world.c hello_world.s -o hello_world
```

That's all! Pretty easy, right? Now let's advance to a more challenging scenario: A function that returns a value. As I said on previous tutorial, by convention, Assembly functions return values on `AX` register. This is also true for C programs. Check out this example:

return_10.s

```
.globl return_10
.type return_10, @function
return_10:
  movl $10, %eax
  ret
```

This function only puts the value ‘10’ into the EAX register. Now on C side:

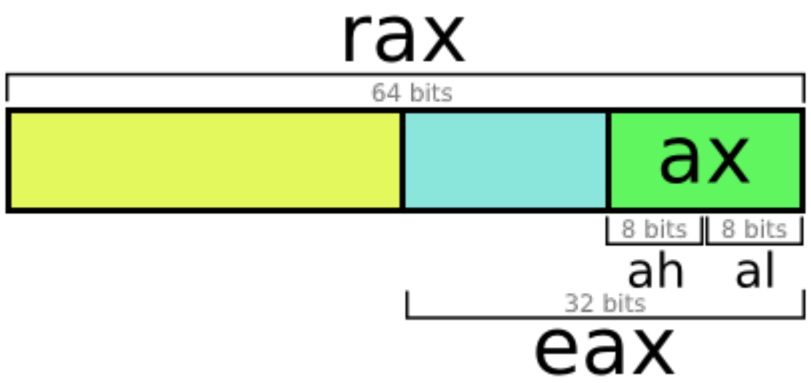
return_10.c

```
#include <stdio.h>

extern int return_10();

int main()
{
  printf("%d\n", return_10());
}
```

It’s worth noting that, on Assembly side, I’m moving a two words value into the EAX register. I could move a four words value to the RAX register instead, but it would print 0. Why? Here’s the reason:



As you may know, RAX is the 64 bits version of the AX register, hence it can store 64 bits simultaneously. Those bits are stored from left to right, i.e., let’s suppose we move the decimal value ‘10’ into the RAX register. It would appear that way:

01010000000000…0 (0101 + 60 zeroes).

The EAX holds the 32 most significant bits (the lower half), therefore, if I access this sequence through EAX, I would only see zero values! And this is what the `int` datatype is implicitly converted to, since it’s a datatype with size equals to 32 bits. In order to avoid this problem, I should either stick with EAX, EBX… registers or use `long int` on C side.

> Lesson learnt: One must check if the size of registers match the size of types in C.

Now the last scenario: A function that takes parameters and returns a value, like that one that returns the sum of two values:

sum.c

```c
#include <stdio.h>

extern int sum(int, int);

int main()
{
  printf("%d\n", sum(2, 3));
  return 0;
}
```

Now the Assembly definition:

sum.as

```
.globl sum
.type sum, @function
sum:
  addl %edi, %esi
  movl %esi, %eax
  ret
```

You may be asking: Hey, what's wrong? Why am I using the `edi` and `esi` registers?

Here's the trick: In GCC compiler, instead of the parameters being pushed into the stack by the callee to be read from the calling function, they are stored in registers. It's the calling function job to push them into the stack if they need to. Those registers are used in the following order:

- _di: Holds the first argument
- _si: Holds the second argument
- _dx: Holds the third argument
- _cx: Holds the fourth argument
- r8d: Holds the fifth argument
- r9d: Holds the sixth argument

And so on... In the above example, the value `2` is stored in the `edi` register and the value `3` is stored in the `esi` register. Therefore, we simply sum them (through the `addl` instruction) and move the result to `eax` register.

Using C-defined functions into Assembly

Here's the first example: Using the `printf` C function into Assembly:

hello_world.s

```
.extern printf
.globl main
.section .data
message: .ascii "Hello, World!\n"
```

```
format: .ascii "%s"
.section .text
main:
  mov $format, %rdi
  mov $message, %rsi
  mov $0, %rax
  call printf
  ret
```

Now compile the Assembly program with GCC:

```
gcc hello_world.s -o hello_world
```

The GCC will automatically link with the function definition. In the same way we used the `extern` keyword in C, we use the `.extern` directive to tell the Assembler that `printf` is defined externally.

That is equivalent to the following C program:

hello_world.c

```
#include <stdio.h>

int main()
{
  return printf("%s", "Hello, World!\n");
}
```

When compiling Assembly programs with GCC, the starting symbol is no longer `_start` but `main`. `main` is a function, therefore it must have the `ret` instruction in the end of it.

The `printf` in C takes two or more parameters: The format and the value(s). As said previously, the first parameters goes to `rdi` register while the second parameter goes to `rsi` register. Note: Before calling the function, the value of `rax` must be zero!

Our second example is using the `scanf` function. Like `printf`, it takes two more parameters: The format and the destinating addresses where the standard input will be stored. Note: The second and so on parameters are no longer values, but memory addresses (pointers).

example_scanf.s

```
.extern scanf
.globl main
.section .data
a: .double 0
b: .double 0
```

```
format: .ascii "%d %d"
.section .text
main:
  mov $format, %rdi
  mov $a, %rsi
  mov $b, %rdx
  mov $0, %rax
  call scanf
  movl a, %eax
  movl b, %ebx
  addl %ebx, %eax
  ret
```

First, we declare three "variables" in data section:

- a: A two words (32 bits) region of memory that initially stores the value zero;
- b: A two words (32 bits) region of memory that initially stores the value zero;
- format: A region of memory that stores the ASCII string "%d %d".

We then pass the address of `format` as first parameter, the address of `a` as second parameter and the address of `b` as third parameter. Before calling `scanf`, we set `RAX` to 0 (just like in the printf example). After it, we move the value stored in `a` address to `eax` register and the value stored in `b` address to `ebx` register. We then sum them both and store the result in `eax`.

After executing the program, if we echo the program execution status:

```
echo $?
```

We'll able to see the sum of both typed numbers.

The above example is equivalent to the following C program:

example_scanf.c

```
#include <stdio.h>

int main()
{
  int a = 0;
  int b = 0;
  char* format = "%d %d";
  scanf(format, &a, &b);
  return a + b;
}
```

Using Assembly code in C

Our third category is pretty straight-forward. See the example:

sum.c

```
#include <stdio.h>

int sum(int a, int b)
{
  asm("addl %edi, %esi");
  asm("movl %esi, %eax");
}

int main()
{
  printf("%d\n", sum(2, 3));
  return 0;
}
```

Now you can compile it normally:

```
gcc sum.c -o sum
```

The compiler will simply insert the assembly code in the appropriated place in the compiled code.

Conclusion

We’ve just learnt very very powerful tools! Learning how to mix Assembly and C give us a deep insight of how the C compiler actually works. I strongly recommend this website for further learning. Play with it around, try some snippets, and see how it’s translated into Assembly.

Posted by Abner Matheus Feb 27th, 2016 10:24 am assembly, c

« Factorial function in x86_64 Assembly Multithreaded K-Means in Java »

Comments

0 Comments Tango with code 1 Login

Recommend Tweet Share Sort by Best

Start the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Be the first to comment.

ALSO ON TANGO WITH CODE

How to Install Sahara and Sahara Dashboard on OpenStack Newton

3 comments • 2 years ago

tosky — Sahara developer (QE) here, we would for sure appreciated some bugs about the missing/incomplete notes in the documentation

Eyeball Tracking for Mouse Control in OpenCV

16 comments • 2 years ago

Jihen Hammedi — it works now thank u

Is It a Cat or Dog? A Neural Network Application in OpenCV

14 comments • 3 years ago

Waheed — Great work sir. can u help me? I'm building a system to recognize human activities

Creating a Telegram Bot in NodeJS

4 comments • 3 years ago

Sion — HI , very helpful article !can you please help me out with ((((insults[Math.floor(Math.random() *

Recent Posts

- Intepreting a Hand-drawn Hash Game
- Eye Tracking for Mouse Control in OpenCV
- How to Install Sahara and Sahara Dashboard on OpenStack Newton (Ubuntu)
- Multithreaded K-Means in Java
- Mixing Assembly and C

GitHub Repos

- Status updating…

@PicoleDeLimao on GitHub