

# Disassembling hello world

written by Stefan Seufert & Sebastian Wolfgarten  
([stefan@seuf.de](mailto:stefan@seuf.de) & [sebastian@wolfgarten.com](mailto:sebastian@wolfgarten.com))

4th October 2005

## 0.1 Preamble

All the examples below were created on a Gentoo linux system with kernel 2.6.12 and gcc 3.3.6. If you have any questions, feel free to ask us (we will at least try to answer them if we can) and feel free to correct us (we have surely done mistakes every here and then).

## 0.2 Getting started

In order to get started you must create a text file on Linux which will later on contain all the source code of our little hello world program. So execute the following command on Linux:

```
$ vi hello.c
```

This command will fire up an editor called vi which you can use to type in the following program:

---

**Algorithm 1** hello world in C

---

```
main() {  
    printf("Hello world!\n");  
}
```

---

If you are not familiar with vi, please refer to [http://math.la.asu.edu/vi\\_tutorial/vicontents.html](http://math.la.asu.edu/vi_tutorial/vicontents.html) or search for any other online vi tutorial. Save the file and start to compile it using the following command:

```
$ gcc -O1 -g hello.c -o hello
```

This command will use gcc to statically (-g) compile the file hello.c and write the binary output to a file called "hello" in the current directory. Now you can test the file by simply executing it:

```
$ ./hello  
Hello world!
```

As expected you can see "Hello world" is printed to the screen.

## 0.3 Using gdb

It's time to start the debugger gdb:

```
$ gdb hello
```

After loading the file it can now be disassembled:

Figure 1: Disassembly of the "hello world" program

```
(gdb) disass main
Dump of assembler code for function main:
0x08048384 <main+0>: push %ebp
0x08048385 <main+1>: mov %esp,%ebp
0x08048387 <main+3>: sub $0x8,%esp
0x0804838a <main+6>: and $0xffffffff,%esp
0x0804838d <main+9>: mov $0x0,%eax
0x08048392 <main+14>: sub %eax,%esp
0x08048394 <main+16>: movl $0x80484a4, (%esp)
0x0804839b <main+23>: call 0x80482a8 <_init+56>
0x080483a0 <main+28>: leave
0x080483a1 <main+29>: ret
End of assembler dump.
```

There are two different types of assembly syntax: the Intel and the AT&T one. Gdb's disassembly above is in AT&T syntax as all the registers are prefixed with a '%'. Hence we read the instructions like "instruction source,dest" and not the other way around ("instruction dest,source") as we would do if this would be Intel-style assembly (which it isn't!). Now let's analyze the code.

## 0.4 Understanding the assembly

The code starts with what is called the prologue which looks like this (everything but the assembly instructions was removed):

```
push %ebp
```

This firstly copies (=pushes) the current value of the basepointer (which is stored in the ebp register) to the stack in order to restore it's original value upon exit of the current function. This is important as the function/program which called hello world relies on it's own value of the base pointer just as hello world does on its values for locating parameters, variables and finally restoring the stack pointer.

```
mov %esp,%ebp
```

This line overwrites the ebp register with the current value of the esp register which points to the top of the stack. This is done to access the variables the program/function uses.

```
sub $0x8,%esp
```

This step decreases the current value of the esp register by 8 byte. The reason being the stack is growing down so variables used by the program will always be stored "below" the esp register. Now the prologue is over and the actual program code starts with the following line:

```
and $0xfffff0,%esp
```

We skip this line, hell knows what it does. The next line stores the value "0" in the EAX register:

```
mov $0x0,%eax
```

This defines the exit code of my program which is "0" ("everything ok"). The next line is quite funny as the compiler probably forgot to optimize it:

```
sub %eax,%esp
```

As we know from above EAX is "0" so subtracting 0 from the value of ESP is useless and doesn't make any sense, does it? This line will be removed if you add "-O1" when compiling the C code ("gcc -O1 -g hello.c -o hello").

```
movl $0x80484a4,(%esp)
```

As we move along the instruction above simply pushes the address 0x80484a4 onto the stack that's literally the same as using "push \$0x80484a4" but without ever touching the stack pointer. Next is the call of printf, 0x80482a8 is the address of the function printf in the memory:

```
call 0x80482a8 <_init+56>
```

Printf then outputs the data that is currently stored on the top of the stack (which is "0x80484a4", the address of our "hello world" string). Finally the program resets the stack pointer and returns to the calling function:

```
leave  
ret
```

That's it.