"Hello World" in Assembly

Posted by **Jacob Bramley**,
Leave Comment
11 February 2010

Assembly language can be fairly daunting, even for experienced software engineers. The lists of strange instructions and squiggles can be hard to read at the best of times; indeed, that is why we use languages such as C, where the compiler worries about such things so you don't have to. However, understanding the instruction set of your processor can make C-level optimizations easier to spot and implement, and will help you to gain an understanding of what your program is really doing. In addition, it can enable you to create some finely-tuned code for specific tasks that are hard to implement in C. If nothing else, it's fun!

This post aims to provide a simple introduction to ARM assembly language. The code will be presented in such a way that you can understand what's going on without having to understand the nuances and specifics of each instruction. Future posts will explain the mechanisms in more detail.

Tools

In order to actually do anything interesting, you'll need an ARM device and a suitable tool-chain. If you have a reasonably powerful device with a desktop-like operating system (such as Ubuntu), you can work directly on the board; this is native development. On Ubuntu, you can use the built-in `apt-get` utility to get a tool-chain; just enter `apt-get install build-essential` (as root) and you'll get a moderately recent version of GCC. In this case, development becomes pretty much identical to development on your PC, except that you'll be writing ARM assembly code rather than x86 assembly code.

If you don't have a particularly powerful ARM device or you don't have a platform that allows you to easily build natively, you'll want to use a cross-compiler. You can get Code Sourcery's free ARM cross-compiler from their website; this is essentially a pre-built ARM cross-compiler (and assembler), so you don't have to worry about building one yourself. In this case, you would have to compile your code on a PC, then move the binary to your platform before executing it there.

Assembly Files

Assembly is essentially a human-readable form of machine code. Each assembly instruction maps more-or-less onto one machine instruction so you can very finely control what the processor is doing. The syntax is much simpler than C; you can't form complex compound statements without explicitly listing the instructions required to calculate the statement. For example, the C expression `a=(b+c)*d` might look like this in ARM assembly:

CODE
```
    add     r0, r1, r2
    mul     r0, r3, r0
```

The expression must be split into `a=(b+c)` and `a=a*d`.

It's important to note at this stage that most assemblers use a different syntax, even though they essentially do the same job. ARM's RVCT includes an assembler that uses a different syntax to the GNU assembler in GCC, for example. Here, we use GCC syntax by default because the GCC tool-chain is readily available for free and for multiple platforms. In addition, the GNU assembler uses a different line-comment delimiter for each platform. On ARM, it is `@`. The GNU assembler also allows the use of C-style multi-line comments (such as "`/* ... */`").

Hello World

Standard C Implementation

A traditional introduction to many languages is the "Hello World" program. In C, this looks something like this:

CODE
```
#include <stdio.h>

int main(void)
{
    printf("Hello, world.\n");
    return 0;
}
```

That's all very well and good, but what does it actually mean to the processor? How does it execute that? The assembly version of the same program is remarkably similar. I won't explain the details of each instruction here, but I'll present the code and we'll discuss the various mechanisms in future posts.

Assembly Implementation

For convenience, the full example program can be downloaded from here (1.38KB .tar.gz), but it is also listed below:

CODE
```
    .syntax unified

    @ --------------------------------
    .global main
main:
    @ Stack the return address (lr) in addition to a dummy register (ip) to
    @ keep the stack 8-byte aligned.
    push    {ip, lr}

    @ Load the argument and perform the call. This is like 'printf("...")' in C.
    ldr     r0, =message
    bl      printf

    @ Exit from 'main'. This is like 'return 0' in C.
    mov     r0, #0      @ Return 0.
    @ Pop the dummy ip to reverse our alignment fix, and pop the original lr
    @ value directly into pc — the Program Counter — to return.
    pop     {ip, pc}

    @ --------------------------------
    @ Data for the printf calls. The GNU assembler's ".asciz" directive
    @ automatically adds a NULL character termination.
message:
    .asciz  "Hello, world.\n"
```

We can assemble and run this program using the following (on an ARM Linux-like platform):

CODE
```
$ gcc -o hello_world hello_world.s
$ ./hello_world
```

You should then see the text "Hello, world." on the console.

If you're using a cross-compiler (such as RVCT or the Code Sourcery edition of GCC) you'll need to run the first step on your PC — probably substituting `gcc` with something like `arm-none-linux-gnueabi-gcc` — and then copy the output binary to an ARM target before running the program itself.

If you're curious about how this relates to what the C compiler would do, try compiling the C version using `gcc -S hello_world.c -O2` instead of your usual compile command. You can also examine existing objects or binaries using `objdump` to disassemble the output. There will be a few differences, and you'll see different results if you provide different `-O` flags to the compiler. The compiler output will also vary between compiler versions.

The details of each mechanism and instruction will be discussed in other posts, but the approximate mapping between the C and assembly implementation should be evident from the example.

Shortlink to this post: http://bit.ly/bOOxa3
Share This Entry:

Back to Software Enablement →

0 Comments On This Entry

Please log in above to add a comment or register for an account