1 Lab 10

Date: Oct 24, 2019

This document first describes the aims of this lab. It then provides necessary background. It then describes the exercises which need to be performed.

1.1 Aims

The aim of this lab is to familiarize you with mixing assembly language code with C. After completing this lab, you should be familiar with the following topics:

- Compiling and linking assembly language files with C files to build a program.
- A glimpse at the syntax used for writing in-line assembly in `gcc`.

1.2 Background

C is a very flexible language and allows access to many of the low-level details of a machine. However, because of the limited semantics of C, some aspects of the machine simply cannot be accessed in C. In those cases, it may be necessary to mix assembly language with C.

Mixing assembly language with C can be done in two ways:

1. Having separate assembly language `.s` files containing external definitions (functions and variables) implemented entirely in assembly language. Those files can be assembled using an assembler to create `.o` object files. The object files can be linked together with other object files (possibly compiled from `.c` files) and libraries to build an executable program.

2. Most C compilers provide facilities to include *inline assembly* when it is necessary to use assembly language for a unit smaller than a function.

A main advantage of writing code in C is portability. If the code uses only standard C libraries, moving the code to a different architecture is a simple matter of recompiling. Adding assembly language to the mix makes the code non-portable. Hence assembly language should only be used in limited situations, like the following:

- Accessing hardware facilities which are impossible to access using pure C. Examples could include kernel-level instructions used for I/O, locking, memory management, CPU table management.
- When programming embedded systems, it is often necessary to revert to assembly (even though the C programming environments for embedded

systems usually provide many extensions, they may still be insufficient to access all the hardware facilities).

- It is possible to write some code much more efficiently in assembly language rather than in C and that code is used in performance-critical portions of the program. With modern optimizing compilers, this situation is extremely unlikely to arise.

1.3 Exercises

1.3.1 Starting up

Use the startup directions from the earlier labs to create a `work/lab10` directory and fire up a terminal whose output you are logging using the `script` command. Make sure that your `lab10` directory contains a copy of the files directory.

1.3.2 Exercise 1: Rotate Instructions

Most modern architectures provide some kind of rotate instruction which rotate a register circularly. C does not provide any rotate semantics and it seems necessary to use assembly language if rotations are needed in performance-critical code (like encryption applications).

However, existing C compilers are extremely capable and are setup to recognize certain idioms for rotation. Change over to the rotate directory and look at the rotate.c file. It provides implementation of rotates in terms of shifts: a rotate is computed as a shift in the rotate direction or'd with bits rotated around; the bits rotated around are computed by shifting bit-length - shift-length bits in the opposite direction.

Build an assembly language file by typing `make` and look at the generated `rotate.s`. You will see that the generated code does make use of rotate instructions. Since the `rotl()` and `rotr()` functions have been declared inline, the compiler has replaced the calls with single `rotl` and `rotr` instructions respectively.

The takeaway from this exercise is that when a situation seems to require the use of assembly language, it is worth checking whether the situation can be handled satisfactorily while staying entirely within C.

1.3.3 Exercise 2: cpuid for Vendor Information

Since around the time Pentium processors were introducted, the `x86` architecture has provided a `cpuid` instruction which provides information about the CPU. The operation of the `cpuid` instruction is controlled by the incoming value in

the `eax` register. Information is returned in the `eax`, `ebx`, `ecx` and `edx` registers with the details of the returned information depending on what was requested by the incoming `eax` register.

When the incoming `eax` register is specified as 0, the returned `eax` value is the largest value which can be used for an incoming `eax` value. `ebx`, `edx` and `ecx` (in that order) spell out a *vendor string* corresponding to the CPU manufacturer. Non-zero incoming values for `eax`, returns many low-level details of the CPU.

Change over to the cpuid1 directory and look at the files contained there. The `cpuid.s` file provides a `get_cpuid()` function which executes the `cpuid` instruction with the incoming `eax` set to 0 and returns the returned information via 4-incoming pointer arguments in `rdi`, `rsi`, `rdx` and `rcx`. This function is exercised by a program in `main.c` which decodes and prints out the vendor string.

Build the program by typing `make` and run it by typing `./cpuid`.

1.3.4 Exercise 3: Arbitrary cpuid Information

The previous exercise only used the `cpuid` instruction with the incoming `eax` register set to 0. Change over to the cpuid2 directory: it contains a main.c which is setup to treat its first command-line argument as the value of `eax` used to select the `cpuid` operation. However, the provided `cpuid.s` file does not provide this functionality and has the same code as that from the previous exercise. Modify this file to take an additional initial parameter which is the desired `cpuid` op.

After making your changes, build and test. Check out its operation for values of `eax` other than 0, verifying your result with the information given in wikipedia.

1.3.5 Exercise 4: Parity

A bit-word is defined to have *even parity* if the number of bits in the word is even; otherwise it has *odd parity*. Parity is one of the most simple error-detecting mechanisms; for example, memory words often have an extra parity bit to allow detecting single-bit errors (for example, a 32-bit memory word will have 33-bits with the extra parity bit used to detect errors in the 32 data bits).

Computing the parity of a word in C would require a sequence of bit operations. However, in x86 assembly language, the parity of a word is available by testing the `parity` flag `P` after simply `test`'ing the word.

Change over to the parity directory and look at the provided files. Complete the code in the parity.s file to return `eax` as 1 iff the incoming argument has even parity. You should use the `testl` instruction to set the parity flag. You can

then test the flag using the `jpe` ("jump if even parity") instruction to ensure that the correct return value is set up in register `eax`.

Build using `make` and test by checking the parity of the command-line arguments to the program.

```
$ ./parity
usage: ./parity INT1...
$ ./parity 1 2 3 4 5 6 7 15 21
parity(1) = 0
parity(2) = 0
parity(3) = 1
parity(4) = 0
parity(5) = 1
parity(6) = 1
parity(7) = 0
parity(15) = 1
parity(21) = 0

$ ./parity -1 0 -2
parity(-1) = 1
parity(0) = 1
parity(-2) = 0
$
```

1.3.6 Exercise 5: Accessing the Instruction Pointer

`gcc` allows inline assembly language as a compiler extension. The inline assembly must be written within a string which is the first argument of the `__asm__()`. This argument can contain assembly instructions which must be separated by a `;` or `\n\t` sequence.

This string can also be used as a template. In that case, the remaining arguments to `__asm__()` (separated by `:`) provide constraints on the template arguments. The details of the format are given in the references.

Change over to the rip directory and look at the rip.c file. This file provides a function `get_rip()` which returns the instruction pointer for the the `return` in the function. It uses inline assembly to access the current instruction pointer. The exact syntax:

__asm__("leaq (%%rip), %0": =r(rip));

specifies the template string as `"leaq (%%rip), %0"` (the holes in the template are indicated using `%`d for small integer d; note the clumsiness of having to quote the `%` used in the register specifier by doubling as in `%%rip`). The second argument (after the `:`) specifies the output values affected by the template:

in this case we want it to correspond to the template variable `%0` and make it correspond to the register allocated to the `rip` C variable.

Note that it would have been more direct to have had a template which resulted in something like `movq %rip, %rax`. However, that is not a legal instruction as none of the x86 instructions allow reading `rip` directly. Instead, the `leaq (%rip), %rax` gets the value of `rip` indirectly by using it as a base register in an address computation.

Build the program by typing `make` and run it using `./rip`. Observe that the printed `rip` pointer is consistent with the address printed for the `get_rip()` function (look at the `rip.dump` file produced by the `make` and verify the values).

1.3.7 Exercise 6: Accessing the Stack Pointer

Change over to the rsp directory and look at the files contained there. Then go into the rsp.c file and add inline assembly to the `get_rsp()` function to return the value of the `rsp` register just before the `return`. Note that since x86 allows direct reading of the `rsp` register, getting hold of its value is much easier than reading the value of the `rip` register in the previous exercise.

1.4 References

- Wikipedia article on the cpuid instruction.
- Wikipedia article on rotates
- Clark Coleman *Using Inline Assembly With gcc*. Three relatively old documents.
- *Inline Assembly in gcc*. The official gcc inline assembly docs.
- *GCC-Inline-Assemble-HOWTO*.
- *GCC Inline ASM*.
- 0xAX, *Inline assembly*. Overall, looks like an interesting book on the Linux kernel.