2.
Optimizing subroutines in assembly language
An optimization guide for x86 platforms

By Agner Fog. Technical University of Denmark.

Contents

1 Introduction

This is the second in a series of five manuals:

1. Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms.

2. Optimizing subroutines in assembly language: An optimization guide for x86 platforms.

3. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers.

4. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs.

5. Calling conventions for different C++ compilers and operating systems.

The latest versions of these manuals are always available from www.agner.org/optimize. Copyright conditions are listed on page 166 below.

The present manual explains how to combine assembly code with a high level programming language and how to optimize CPU-intensive code for speed by using assembly code.

This manual is intended for advanced assembly programmers and compiler makers. It is assumed that the reader has a good understanding of assembly language and some experience with assembly coding. Beginners are advised to seek information elsewhere and get some programming experience before trying the optimization techniques described here. I can recommend the various introductions, tutorials, discussion forums and newsgroups on the Internet (see links from www.agner.org/optimize) and the book "Introduction to 80x86 Assembly Language and Computer Architecture" by R. C. Detmer, 2. ed. 2006.

The present manual covers all platforms that use the x86 and x86-64 instruction set. This instruction set is used by most microprocessors from Intel, AMD and VIA. Operating systems that can use this instruction set include DOS, Windows, Linux, FreeBSD/Open BSD, and Intel-based Mac OS. The manual covers the newest microprocessors and the newest instruction sets. See manual 3 and 4 for details about individual microprocessor models.

Optimization techniques that are not specific to assembly language are discussed in manual 1: "Optimizing software in C++". Details that are specific to a particular microprocessor are covered by manual 3: "The microarchitecture of Intel, AMD and VIA CPUs". Tables of instruction timings etc. are provided in manual 4: "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs". Details about calling conventions for different operating systems and compilers are covered in manual 5: "Calling conventions for different C++ compilers and operating systems".

Programming in assembly language is much more difficult than high-level language. Making bugs is very easy, and finding them is very difficult. Now you have been warned! Please don't send your programming questions to me. Such mails will not be answered. There are various discussion forums on the Internet where you can get answers to your programming questions if you cannot find the answers in the relevant books and manuals.

Good luck with your hunt for nanoseconds!

1.1 Reasons for using assembly code

Assembly coding is not used as much today as previously. However, there are still reasons for learning and using assembly code. The main reasons are:

1. Educational reasons. It is important to know how microprocessors and compilers work at the instruction level in order to be able to predict which coding techniques are most efficient, to understand how various constructs in high level languages work, and to track hard-to-find errors.

2. Debugging and verifying. Looking at compiler-generated assembly code or the disassembly window in a debugger is useful for finding errors and for checking how well a compiler optimizes a particular piece of code.

3. Making compilers. Understanding assembly coding techniques is necessary for making compilers, debuggers and other development tools.

4. Embedded systems. Small embedded systems have fewer resources than PC's and mainframes. Assembly programming can be necessary for optimizing code for speed or size in small embedded systems.

5. Hardware drivers and system code. Accessing hardware, system control registers etc. may sometimes be difficult or impossible with high level code.

6. Accessing instructions that are not accessible from high level language. Certain assembly instructions have no high-level language equivalent.

7. Self-modifying code. Self-modifying code is generally not profitable because it interferes with efficient code caching. It may, however, be advantageous for example to include a small compiler in math programs where a user-defined function has to be calculated many times.

8. Optimizing code for size. Storage space and memory is so cheap nowadays that it is not worth the effort to use assembly language for reducing code size. However, cache size is still such a critical resource that it may be useful in some cases to optimize a critical piece of code for size in order to make it fit into the code cache.

9. Optimizing code for speed. Modern C++ compilers generally optimize code quite well in most cases. But there are still cases where compilers perform poorly and where dramatic increases in speed can be achieved by careful assembly programming.

10. Function libraries. The total benefit of optimizing code is higher in function libraries that are used by many programmers.

11. Making function libraries compatible with multiple compilers and operating systems. It is possible to make library functions with multiple entries that are compatible with different compilers and different operating systems. This requires assembly programming.

The main focus in this manual is on optimizing code for speed, though some of the other topics are also discussed.

1.2 Reasons for not using assembly code

There are so many disadvantages and problems involved in assembly programming that it is advisable to consider the alternatives before deciding to use assembly code for a particular task. The most important reasons for *not* using assembly programming are: