

2.

Optimizing subroutines in assembly language

An optimization guide for x86 platforms

By Agner Fog. Technical University of Denmark.
Copyright © 1996 - 2015. Last updated 2015-12-23.

Contents

1	Introduction	4
1.1	Reasons for using assembly code	5
1.2	Reasons for not using assembly code	5
1.3	Operating systems covered by this manual.....	6
2	Before you start.....	7
2.1	Things to decide before you start programming	7
2.2	Make a test strategy.....	8
2.3	Common coding pitfalls.....	9
3	The basics of assembly coding.....	11
3.1	Assemblers available	11
3.2	Register set and basic instructions.....	14
3.3	Addressing modes	18
3.4	Instruction code format	25
3.5	Instruction prefixes.....	26
4	ABI standards.....	27
4.1	Register usage.....	28
4.2	Data storage	28
4.3	Function calling conventions	29
4.4	Name mangling and name decoration	30
4.5	Function examples.....	31
5	Using intrinsic functions in C++	34
5.1	Using intrinsic functions for system code	35
5.2	Using intrinsic functions for instructions not available in standard C++	36
5.3	Using intrinsic functions for vector operations	36
5.4	Availability of intrinsic functions.....	36
6	Using inline assembly.....	36
6.1	MASM style inline assembly	37
6.2	Gnu style inline assembly	42
6.3	Inline assembly in Delphi Pascal.....	45
7	Using an assembler.....	45
7.1	Static link libraries	46
7.2	Dynamic link libraries	47
7.3	Libraries in source code form.....	48
7.4	Making classes in assembly.....	49
7.5	Thread-safe functions	50
7.6	Makefiles	51
8	Making function libraries compatible with multiple compilers and platforms	52
8.1	Supporting multiple name mangling schemes	52
8.2	Supporting multiple calling conventions in 32 bit mode	53
8.3	Supporting multiple calling conventions in 64 bit mode	56
8.4	Supporting different object file formats	58
8.5	Supporting other high level languages	59
9	Optimizing for speed	60
9.1	Identify the most critical parts of your code	60
9.2	Out of order execution	60

9.3	Instruction fetch, decoding and retirement	63
9.4	Instruction latency and throughput	64
9.5	Break dependency chains.....	65
9.6	Jumps and calls	66
10	Optimizing for size.....	73
10.1	Choosing shorter instructions.....	73
10.2	Using shorter constants and addresses	75
10.3	Reusing constants	76
10.4	Constants in 64-bit mode	76
10.5	Addresses and pointers in 64-bit mode	77
10.6	Making instructions longer for the sake of alignment.....	79
10.7	Using multi-byte NOPs for alignment	81
11	Optimizing memory access.....	82
11.1	How caching works	82
11.2	Trace cache	83
11.3	μop cache	83
11.4	Alignment of data	84
11.5	Alignment of code	86
11.6	Organizing data for improved caching	88
11.7	Organizing code for improved caching	88
11.8	Cache control instructions.....	89
12	Loops	89
12.1	Minimize loop overhead	89
12.2	Induction variables	92
12.3	Move loop-invariant code	93
12.4	Find the bottlenecks	93
12.5	Instruction fetch, decoding and retirement in a loop	94
12.6	Distribute μops evenly between execution units.....	94
12.7	An example of analysis for bottlenecks in vector loops	95
12.8	Same example on Core2	98
12.9	Same example on Sandy Bridge.....	100
12.10	Same example with FMA4	101
12.11	Same example with FMA3	101
12.12	Loop unrolling	102
12.13	Vector loops using mask registers (AVX512)	104
12.14	Optimize caching	105
12.15	Parallelization	106
12.16	Analyzing dependences	107
12.17	Loops on processors without out-of-order execution	111
12.18	Macro loops	113
13	Vector programming.....	115
13.1	Conditional moves in SIMD registers	116
13.2	Using vector instructions with other types of data than they are intended for	119
13.3	Shuffling data.....	121
13.4	Generating constants.....	125
13.5	Accessing unaligned data and partial vectors	127
13.6	Using AVX instruction set and YMM registers	131
13.7	Vector operations in general purpose registers	136
14	Multithreading.....	138
14.1	Hyperthreading	138
15	CPU dispatching.....	139
15.1	Checking for operating system support for XMM and YMM registers	140
16	Problematic Instructions	141
16.1	LEA instruction (all processors).....	141
16.2	INC and DEC	142
16.3	XCHG (all processors)	143
16.4	Shifts and rotates (P4)	143
16.5	Rotates through carry (all processors)	143

16.6 Bit test (all processors)	143
16.7 LAHF and SAHF (all processors)	143
16.8 Integer multiplication (all processors)	143
16.9 Division (all processors)	143
16.10 String instructions (all processors)	147
16.11 Vectorized string instructions (processors with SSE4.2).....	147
16.12 WAIT instruction (all processors)	148
16.13 FCOM + FSTSW AX (all processors)	149
16.14 FPREM (all processors)	150
16.15 FRNDINT (all processors).....	150
16.16 FSCALE and exponential function (all processors)	150
16.17 FPTAN (all processors).....	152
16.18 FSQRT (SSE processors).....	152
16.19 FLDCW (Most Intel processors)	152
16.20 MASKMOV instructions.....	153
17 Special topics	153
17.1 XMM versus floating point registers	153
17.2 MMX versus XMM registers	154
17.3 XMM versus YMM registers	154
17.4 Freeing floating point registers (all processors).....	155
17.5 Transitions between floating point and MMX instructions	155
17.6 Converting from floating point to integer (All processors)	155
17.7 Using integer instructions for floating point operations	157
17.8 Using floating point instructions for integer operations	160
17.9 Moving blocks of data (All processors).....	160
17.10 Self-modifying code (All processors).....	163
18 Measuring performance.....	163
18.1 Testing speed	163
18.2 The pitfalls of unit-testing	165
19 Literature	165
20 Copyright notice	166

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:

[Click here to download full PDF material](#)