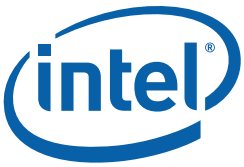


White Paper
David Kreitzer
Compiler Engineer
Max Domeika
Technical Consultant
Intel Corporation

Ensuring Development Success by Understanding and Analyzing Assembly Language

For IA-32 and Intel® 64 Instruction Set Architecture

January 2009

321059

Executive Summary

The ability to understand assembly language is a difficult but essential skill for developers of embedded applications. Even developers who write their applications in a high level language like C will sometimes need to examine assembly code for debugging or performance tuning. Assembly language for the IA-32 and Intel® 64 architectures is especially challenging to read and understand due to the size and variety of the instruction sets. This paper will impart a basic understanding of assembly language and the IA-32 and Intel® 64 instruction sets so that the reader will feel comfortable diving down to the assembly level when necessary.

Assembly language for the IA-32 and Intel® 64 architectures is especially challenging to read and understand due to the size and variety of the instruction sets.

Specifically, this paper provides:

- A basic overview of the architecture.
- Details on Intel® Streaming SIMD Extensions and Intel® Advanced Vector Extensions.
- Practical guidelines on employing assembly language listings and how to map instructions back to source code.
- Examples which show the effects of compiler optimizations on generated assembly language.
- An example showing how analysis of the assembly language leads to tuning improvements to the original source code.

Contents

Executive Summary 2

Introduction 4

Basic Architecture 4

- Register Sets and Data Types 4
- Instruction Formats and Memory Addressing Modes 7

Instruction Set Extensions 9

- Intel® Streaming SIMD Extensions 9
- Intel® Advanced Vector Extensions (AVX) 12

Effectively using Assembly Language Listings 14

- Generating Assembly Language Listings 14
- Effects of compiler optimization 17

Code Sample Assembly Analysis 22

Summary 26

Introduction

Assembly Language is a representation of the machine language that executes on a given architecture. Reading and understanding IA-32 and Intel® 64 architecture assembly languages are difficult tasks. Embedded software engineers are daunted by the sheer longevity of the instruction set and breadth that includes:

- 8-bit, 16-bit, 32-bit, and 64-bit general purpose register sets
- Variable length instructions
- Multiple instructions that perform the same action
- Aggressive compiler optimization
- New instructions and register sets
- x87 floating point

Regardless, there are times when an embedded software engineer need to look at the disassembly of an application binary or the source code compiled into assembly to perform debugging or performance tuning. This paper equips the reader with the competence necessary to perform these actions. First, architecture basics are detailed with information on the register sets, data types, and memory and instruction formats. Next, instruction set extensions are detailed which include Intel® Streaming SIMD Extensions (Intel® SSE) and Intel® Advanced Vector Extensions (Intel® AVX). The third section shows how to effectively generate and read assembly language. The final section provides an analysis of a small C program compiled to assembly language.

Basic Architecture

The IA-32 and Intel® 64 architectures are described in terms of their register sets, data types, memory addressing modes and instruction formats. The next subsections summarize these aspects of the architectures.

Register Sets and Data Types

The original Intel® 8086 processor implemented a 16-bit architecture that defined eight 16-bit general purpose registers named AX, BX, CX, DX, SP, BP, SI, and DI. The upper and lower 8 bits of AX-DX could be accessed independently as AL-DL and AH-DH respectively. The Intel386™ processor extended the architecture to 32 bits, creating the general purpose register file found in the modern IA-32 architecture. The registers grew to 32 bits but

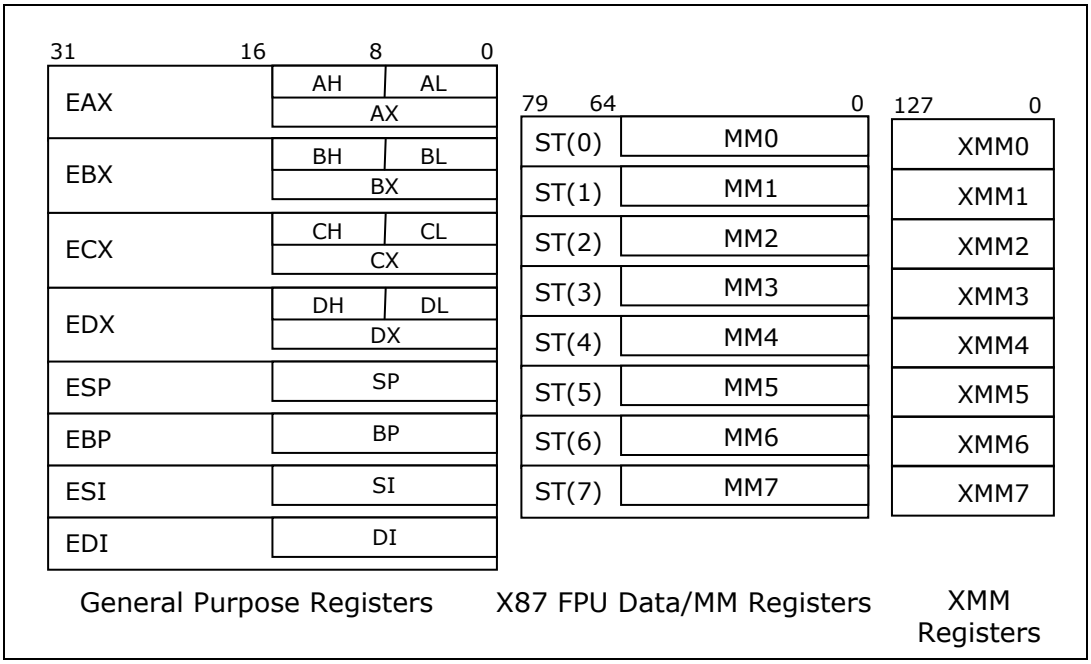
retained their 8- and 16-bit aliases. The general purpose registers typically hold integer data, and the architecture provides instructions to operate on 8-, 16-, 32-, and in some cases 64-bit signed and unsigned integers. Figure 1 illustrates the general purpose registers in the IA-32 architecture.

The x87 floating-point unit (FPU) was first implemented as an optional coprocessor, but modern implementations of the IA-32 architecture integrate the FPU into the processor. The x87 register set consists of eight 80-bit values accessed as a stack as Figure 1 illustrates. A floating point value in an x87 register is represented internally in a double extended precision format, but the architecture provides instructions that can load and store single precision data and double precision data as well. When loading single or double precision data, there is an implicit conversion up to the internal 80-bit format. Similarly, when storing data in single or double precision, there is an implicit conversion from the internal 80-bit format to the smaller format. All three data types conform to the IEEE standard for binary floating-point arithmetic.

The MMX™ technology instruction set was introduced in some of the later Intel® Pentium® processor models. It was the first of many single-instruction multiple-data (SIMD) extensions. The MMX™ instructions operate on eight 64-bit MM registers, and these registers alias the lower 64 bits of the x87 registers as Figure 1 illustrates. Unlike the x87 registers, the MM registers are accessed directly and not as a stack. (Note that Figure 1 shows MM0 aliasing ST(0), MM1 aliasing ST(1), etc. That is an approximation of reality, because the numbering of the x87 registers changes as values are pushed to the stack and popped from the stack.) The MM registers can hold packed integer data of various sizes, and there are different instructions for operating on data in different underlying types. For example, the PADDB, PADDW, PADDD, and PADDQ instructions perform packed addition of 8-, 16-, 32-, and 64-bit integers, respectively.

The Intel® Pentium® III processor introduced the XMM register file as part of the Intel® Streaming SIMD Extensions (Intel® SSE). As Figure 1 shows, there are eight 128-bit direct access XMM registers, and these registers can hold scalar or packed single precision floating-point data, scalar or packed double precision floating-point data, or packed integer data of various sizes. As with the MMX™ instruction set, the instruction determines the data type.

Figure 1. IA-32 Architecture Data Registers



The Intel® 64 architecture was introduced primarily to expand the addressable memory space beyond the 32-bit limit, but it also significantly expands the data registers of the IA-32 architecture. First, it extends the size of the general purpose registers to 64 bits. It preserves all the existing register names for accessing the lower 32 bits of the registers and adds the names RAX-RDX, RSP, RBP, RSI, and RDI for accessing the full 64-bit registers. Second, it adds the ability to independently reference the lower 8 bits of RSP, RBP, RSI, and RDI using the names SPL, BPL, SIL, and DIL. Third, it adds eight new general purpose registers named R8-R15. The lower 32, 16, or 8 bits of these registers can be accessed using the names R8D-R15D, R8W-R15W, and R8B-R15B, respectively. Finally, it adds eight new XMM registers named XMM8-XMM15. Figure 2 illustrates the Intel® 64 architecture data registers. Note that the x87 and MM registers are identical between the IA-32 and Intel® 64 architectures.

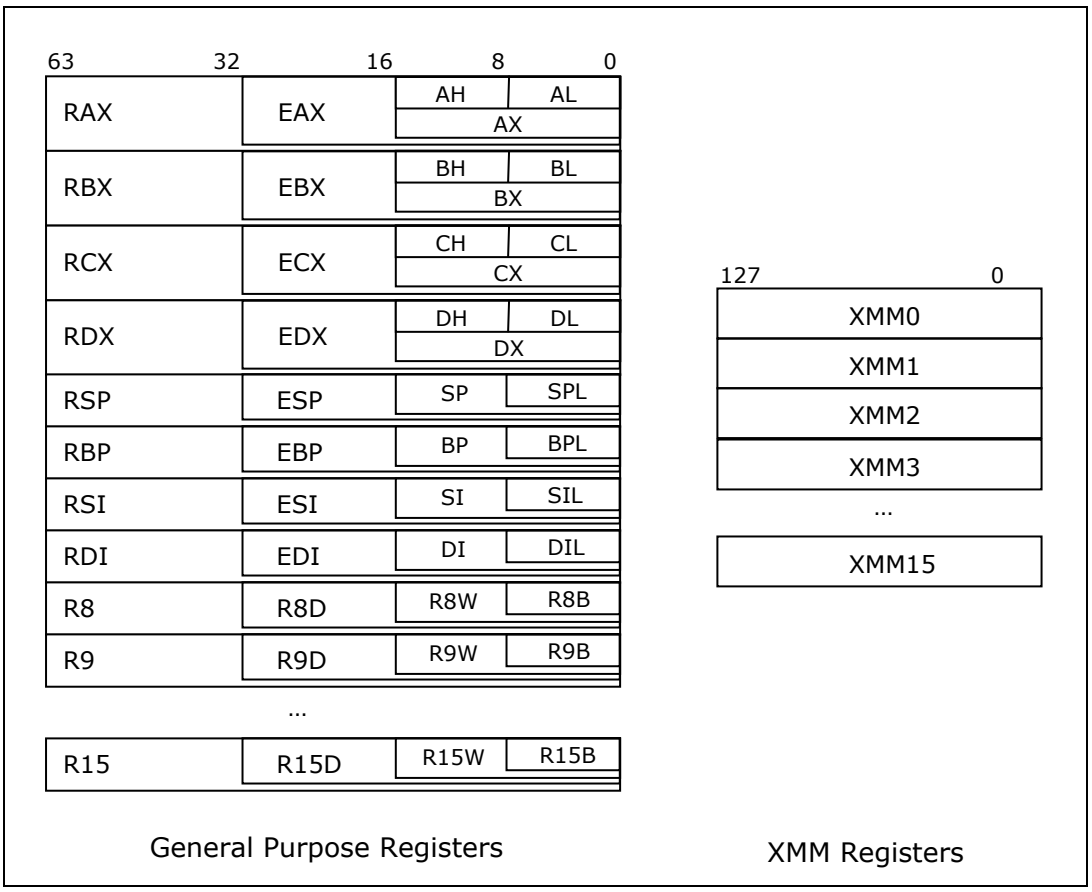


Figure 2. Intel® 64 Architecture Data Registers

Instruction Formats and Memory Addressing Modes

Assembly instructions begin with the mnemonic. A mnemonic is a short string that is usually an abbreviation of the instruction's function. For example, MULPS is the mnemonic used to represent a **MUL**tiply of **P**acked **S**ingle precision floating-point values.

Following the mnemonic are zero or more operands. The legal number and types of operands depend on the instruction, but by convention, in AT&T syntax assembly the output operand is always the rightmost operand.

An operand may be a register, a memory reference, or a literal value, usually called an immediate. Register operands can be used as input, output, or both and are indicated in assembly by prefixing the register name with the % character. Immediate operands are always used as input and are indicated in assembly by prefixing the literal value with the $ character. Sometimes, the literal value is something that is not known until link time such as the address of a global variable. In assembly, a link-time constant is represented symbolically as shown in the following instruction, which moves the address of x into register EAX.

```
movl     $x, %eax
```

Operands not prefixed by either % or $ are memory operands. Like register operands, memory operands can be used as input, output, or both. In the IA-32 and Intel® 64 architectures, instructions can use fairly complex expressions to specify the address for a memory reference. The general form of a memory expression appears as follows in assembly.

```
offset(base register, index register, scale)
```

The effective address for this memory expression is computed by

```
offset + base register + (index register * scale)
```

All components of the address expression are optional except that a scale must accompany an index register. The offset can be either an 8-bit or 32-bit immediate value and is often a link time constant. It is sign extended to the address size. Scale is an immediate value and must be one of 1, 2, 4, or 8. In the IA-32 ISA, the base and index registers must be 32-bit registers. Values stored in 8- or 16-bit registers must first be extended to 32 bits before use in an address expression. Similarly, in the Intel® 64 ISA, the base and index registers must be 64-bit registers. Smaller values must be extended to 64 bits before use in address expressions.

In the Intel® 64 architecture, there is also a special RIP-relative addressing mode. This mode always uses register RIP as the base register with a 32-bit offset and no index register. Register RIP is the instruction pointer register. It contains the address of the next instruction to be executed. This addressing mode significantly improves the efficiency of position independent code.

In the current IA-32 and Intel® 64 architectures, binary operations such as MULPS use a 2-operand format. Conceptually, MULPS has two input operands, the multiplicands, and one output operand, the result. But the architecture requires that the output overwrites one of the inputs. So, for example, the semantics of the instruction

```
mulps %xmm1, %xmm2
```

are to multiply the contents of register XMM1 by the contents of register XMM2 and store the result in register XMM2. If a program needs to preserve the values of both multiplicands, it must use an extra copy instruction.

```
movaps   %xmm2, %xmm3
mulps %xmm1, %xmm3
```

For many instructions, the types of operands can vary. Continuing with the MULPS example, the first operand can either be a register or a memory location. Suppose you wanted to multiply the contents of register XMM2 by 4 elements in an array of floats. One method is to use separate load and multiply instructions.

```
movaps    x(,%eax,4), %xmm1
mulps %xmm1, %xmm2
```

An alternative is to use a single multiply instruction with a memory operand.

```
mulps     x(,%eax,4), %xmm2
```

The latter is more efficient in the number of instructions, code size, XMM registers, and usually execution time.

Many integer instructions show even greater variety in the types of operands they can handle. For example, the following are all valid forms of a 32-bit integer addition.

```
addl      %ebx, %edx
addl      (%eax), %edx
addl      $42, %edx
addl      $42, (%eax)
addl      %ebx, (%eax)
```

Instruction Set Extensions

Intel routinely enhances the instruction sets of the IA-32 and Intel® 64 architectures. MMXTM technology introduced a set of instructions that process data in a single instruction, multiple data fashion. The instructions were specifically designed to improve the performance of integer programs with a high degree of data parallelism such as signal processing. Successive generations of Intel architecture processors have added further instruction set extensions including Intel® SSE, and Intel® AVX. Understanding these instruction set extensions is crucial to taking full advantage of the processor's capabilities. In some cases, they can enable you to efficiently execute applications on the Intel® architecture processor that have previously been executed on special purpose hardware such as a graphics processing unit (GPU).

Intel® Streaming SIMD Extensions

Intel® SSE provides SIMD processing of single precision floating-point values. SIMD processing is also known as vector processing and allows the same computation to be performed on multiple pieces of data (Multiple Data) using only one instruction (Single Instruction). The Intel® SSE instruction set features 70 instructions that perform vector and scalar floating-point arithmetic, comparisons, type conversions between floating-point and integer data, boolean logic, and data rearrangement. Figure 3 illustrates the kind of computation that can benefit from Intel® SSE. The loop multiplies two 4-element vectors together, and stores the result in a third vector. With Intel®

SSE, the entire computation can be performed using only three instructions. The assembly code required to perform the same function using scalar x87 instructions is much longer and slower than what Intel® SSE provides.

Figure 3. Intel® SSE Vector Multiply Example

Source Code

```
float x[4], y[4], z[4];
int i;
...
for (i = 0; i < 4; i++) {
    z[i] = x[i] * y[i];
}
```

Intel SSE Assembly Implementation

```
movaps    x, %xmm0
mulps     y, %xmm0
movaps    %xmm0, z
```

x87 Assembly Implementation

```
flds      x
fmuls     y
flds      4+x
fmuls     4+y
flds      8+x
fmuls     8+y
flds      12+x
fmuls     12+y
fxch      %st(3)
fstps     z
fxch      %st(1)
fstps     4+z
fstps     8+z
fstps     12+z
```

Intel® SSE also introduced a software prefetch instruction, a non-temporal store instruction, and several MMX™ technology enhancements specifically targeted at accelerating media encoding and decoding. Effective use of Intel® SSE can increase performance tremendously for applications such as 3D graphics that use single-precision floating point and tend to employ common operations on vast amounts of data.

Intel® SSE first introduced aligned loads and stores to the architecture. Most 128-bit memory references require the address to be 16-byte aligned. The idea is that memory references that cross a cache line boundary are costly on most processors, and enforcing 16-byte alignment avoids that potential penalty. There are dedicated 128-bit unaligned load and store instructions for when the data is known to be misaligned. For example, the following two instructions both load 128 bits of data from the address in register EAX. The only functional difference in behavior is that the MOVAPS instruction faults if the address in EAX is not an even multiple of 16.

```
movaps    (%eax), %xmm0
movups    (%eax), %xmm0
```

Intel® SSE2 adds SIMD processing for double precision floating-point and also 128-bit SIMD processing for integer data. The capabilities Intel® SSE2 provides for double precision floating-point are similar to what Intel® SSE provides for single precision. The vector width is smaller since a 128-bit vector register can hold 4 single precision values but only two double precision values. For integer data, Intel® SSE2 effectively doubles the vector width over MMX™ technology. But otherwise the capabilities are similar. Figure 4 illustrates a vector average of unsigned bytes that can be performed efficiently using Intel® SSE2.

Figure 4. Intel® SSE2 Vector Average Example

Source Code

```
unsigned char a[16], b[16], c[16];
int i;
...
for (i = 0; i < 16; i++) {
    c[i] = (a[i] + b[i] + 1) / 2;
}
```

Assembly Implementation

```
movdqa    a, %xmm0
pavgb     b, %xmm0
movdqa    %xmm0, c
```

Intel® Supplemental Streaming SIMD Extensions 3 (SSE3) provides a small number of instructions primarily designed to improve the efficiency of single-precision and double-precision complex multiplication and division.

Intel® SSSE3 provides additional instructions for operating on integer data. For example, it provides horizontal addition and subtraction instructions for operating on data elements within a single vector. It also provides packed absolute value and conditional negation instructions.

Intel® Supplemental Streaming SIMD Extensions 4.1 (SSE4.1) provides a variety of additional instructions for operating on vector data in the XMM registers. Some of these instructions make it easier for a compiler to automatically vectorize a scalar loop. For example, there are instructions for vector sign and zero extension, vector and scalar floating-point rounding, and vector integer max/min. The vector blend instructions expose new opportunities to write vector code for loops containing conditionals. For example, Figure 5 illustrates a simple 4-iteration loop with a conditional select construct. With Intel® SSE4.1, the entire loop can be implemented with just a few instructions. The XORPS and CMPLTPS instructions compare the elements of the x vector to zero and produce a mask of the comparison results in register XMM0. The BLENDVPS instruction uses the mask in

register XMM0 to select the corresponding element from either the y vector (in memory) or the z vector (in register XMM1).

Figure 5. Intel® SSE4.1 Vector Blend Example

Source Code

```
float x[4], y[4], z[4];
int i;
...
for (i = 0; i < 4; i++) {
    x[i] = (x[i] > 0) ? y[i] : z[i];
}
```

Assembly Implementation

```
movaps    z, %xmm1
xorps     %xmm0, %xmm0
cmpltps   x, %xmm0
blendvps  %xmm0, y, %xmm1
movaps    %xmm1, x
```

Intel® SSE4.2 adds four new string processing instructions that perform advanced string comparison and search operations in a single instruction. It also provides a 64-bit packed integer comparison instruction.

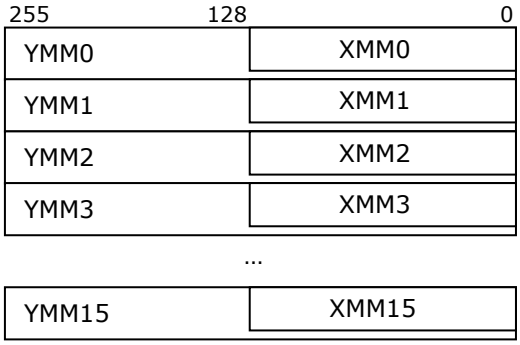
The Intel® Core™ i7 processor added two application targeted accelerators, POPCNT and CRC32. Strictly speaking, these are not part of Intel® SSE but are instruction set extensions designed to boost the performance of specific performance critical algorithms.

Intel® Advanced Vector Extensions (AVX)

The Intel® Advanced Vector Extensions (AVX) are instruction set enhancements that further extend the processor's SIMD capabilities. Intel plans to ship processors that implement Intel® AVX starting in 2010.

Intel® AVX widens the XMM register file from 128 to 256 bits, doubling the maximum SIMD vector length. The full 256-bit registers are referenced in assembly using the names YMM0-YMM15, as illustrated in Figure 6. In 32-bit mode, only registers YMM0-YMM7 are available.

Figure 6. Intel® AVX Register Extensions



YMM Registers

The Intel® AVX instructions logically extend the 128-bit vector Intel® SSE instructions to 256 bits. Simple operations like vector multiplies operate on twice as many data elements. Data rearrangement operations like shuffles usually operate "in-lane", which means that the same data rearrangement operation is applied to both 128-bit halves or "lanes" of a YMM register. New 128-bit permute, insertion, and extraction instructions enable a program to move data across lanes. The Intel® AVX architecture also provides new primitives like broadcasts and masked loads and stores. These primitives make it easier to generate efficient vector code, and they make it easier for a compiler to vectorize code automatically.

In addition to wider vectors, Intel® AVX provides 3-operand instruction forms so that the destination operand is independent of the input operands. Intel® AVX also provides alternate 3-operand forms for all existing 128-bit vector and scalar operations. Using the MULPS example, even if the program needs to preserve both multiplicands, it can implement the multiply using one instruction.

```
vmulps    %xmm1, %xmm2, %xmm3
```

AVX preserves the convention that the destination operand is the rightmost one, so the operation being performed by this instruction is shown by the following expression.

```
XMM3 ← XMM1 * XMM2
```

Finally, for most instructions Intel® AVX lifts the restriction that vector loads and stores be aligned. Explicit "move aligned" instructions such as `VMOVDQA` still require addresses to be aligned on vector size boundaries. But other vector loads and stores can be unaligned. This is most useful in creating opportunities to merge load and store instructions with other instructions. Previously, loads could not be combined with subsequent instructions unless

the load addresses were provably aligned, leading to multi-instruction sequences like the following.

```
movups   x(,%eax,4), %xmm1
mulps %xmm1, %xmm2
```

Intel® AVX enables these two instructions to be combined.

```
vmulps   x(,%eax,4), %xmm2, %xmm2
```

Effectively using Assembly Language Listings

This section provides useful tips for correlating assembly language to the original source code. First, we explain the basics of assembly language listings including how to generate and read them. Afterwards, several examples are discussed which show how compiler optimizations can increase the difficulty of correlating assembly code to source code and how you can effectively understand the transformations made by the compiler to decipher your assembly language listings.

Generating Assembly Language Listings

The first task in effective use of assembly language is to know how to easily generate it. Two techniques for generating assembly for your application code are:

- Disassemble the application binary
- Compile the source code to assembly language.

Disassembly of an application binary requires a tool called a disassembler which maps the machine language contained in the binary to the equivalent assembly language mnemonics. A disassembler is a common tool on modern operating systems. On Linux* systems, the `objdump -d` command produces a disassembly listing. For example, to disassemble the `cat` command, type:

```
objdump -d `which cat`
```

The disassemble command is not the same on all systems. On Windows* and Mac* OS, the disassemble commands are `dumpbin /disasm` and `otool -t -V` respectively. Many debuggers feature the option to view the code under debug as assembly language and effectively call a disassembler to display the instructions.

To produce an assembly file with a compiler, consult your compiler's manual to find the correct option. Using gcc and other compatible compilers on a Linux* system, the option is `-S`. Using the Intel® C++ Compiler to compile the code in Figure 7 produces an assembly listing, a portion of which is in Figure 8. This resulting listing is an example using Intel® 64 ISA assembly language.

Figure 7. Source Code for Function Simple_loop

```
long simple_loop(long x)
{
    long i, ret_val = 0;
    #pragma novector
    for (i=0; i< x; i++) {
        ret_val += i;
    }
    return ret_val;
}
```

Figure 8. Assembly Language for Function Simple_loop

```
simple_loop:
# parameter 1: %rdi
..B1.1:                         # Preds ..B1.0
..___tag_value_simple_loop.1:   #2.1
        xorl      %eax, %eax    #3.19
        xorl      %edx, %edx    #5.8
        testq     %rdi, %rdi    #5.16
        jle       ..B1.5        # Prob 10% #5.16
                   #LOE rax rdx rbx rbp rdi r12 r13 r14 r15
..B1.3:                         # Preds ..B1.1 ..B1.3
        addq      %rdx, %rax    #6.5
        addq      $1, %rdx      #5.19
        cmpq      %rdi, %rdx    #5.16
        jl        ..B1.3        # Prob 82%   #5.16
..B1.5:                         # Preds ..B1.3 ..B1.1
        ret                     #8.10
        .align    2,0x90
```

The four components of an assembly listing are:

- Instructions
- Directives
- Comments
- Labels

Instructions map 1-to-1 to the low level machine language instructions that the processor executes. An example instruction is:

```
addq %rdx, %rax
```

This represents a 64-bit addition of register RDX and register RAX. The result of the operation is stored in register RAX. Please refer to the section, Instruction Formats and Memory Addressing Modes for a description of the instruction format for the IA-32 and Intel® 64 architectures.

Directives are commands to the assembler that control its generation of machine language code. The directive used in Figure 8, denoted ".align" instructs the assembler to align the next instruction on a particular address boundary. The assembler has the ability to know how many bytes previous instructions have taken and can pad the address range with nops (no operations) so a particular instruction is aligned. Code alignment is a performance optimization for the instruction fetch unit and instruction cache. It is often good to align the target of a branch or call to guarantee it is the start of an instruction fetch line or cache line. Other directives include type, size, data, and section.

Labels are generic identifiers for locations in the program that can either be data or instructions. In Figure 8, the label denoted "`..B1.3`" is an identifier for the location of the instruction that it immediately precedes. Labels identify addresses that are not explicitly known until the assembly code is assembled or until the machine language code is loaded for execution. For example, a jump to a label requires either the absolute address of the target location or a relative offset from the address of the instruction immediately following the branch to the address of the target location. In some cases, the absolute address is not known until the operating system assigns addresses for the program. It is possible to compute the relative offset of a branch during assembly. For example, the branch instruction JL in Figure 8 must use a relative offset. The number of bytes between it and the target address is something that can be computed during assembly.

Comments are denoted by the # symbol and indicate extra information that is ignored by the assembler. In compiler generated assembly listings, the comments often provide useful information about the program. In the case of the assembly listing in Figure 8, the comments were generated by the Intel® C++ Compiler Professional Edition for Linux* OS version 11.0. There are many comments of the form "5.19" that follow instructions in the assembly listing. These comments are particularly valuable, because they tell you the line number of the source program that resulted in the assembly language instruction. They use the form "line number.column number". The first comment, "`# parameter 1: %rdi`", tells you the location used to pass argument x to the function. There are several comments of the form "`Prob 82%`" that follow branch instructions and give the compiler's estimate of how likely the branch will be taken. The "`Preds`" comments provide information about the control flow within the function. For example, the "`# Preds`

`..B1.3 ..B1.1`" comment indicates that basic block B1.5 can either be reached by block B1.3 or B1.1. Finally, the LOE comment that appears at the end of a basic block indicates the set of registers that hold meaningful values on exit from the block.

Effects of compiler optimization

Aggressive compiler optimization can transform C or C++ code into very difficult to read assembly language. The following are examples of common optimizations that make the assembly code more difficult to correlate to the C source code:

- Basic block placement
- Inlining
- Strength reduction
- Alternate entries

The examples in this section use Intel® 64 ISA assembly language unless otherwise noted.

Basic block placement is an optimization for branch prediction and for the instruction cache that attempts to place basic blocks that execute close together in time as close together address-wise as possible. A basic block is a set of instructions where if one instruction executes, then program flow dictates all of the other instructions will execute. Typically a basic block begins as the result of being a branch target and ends as a result of a branch instruction. If basic block placement is performed over a large function, basic blocks that make up a tiny portion of the function can be scattered throughout the assembly language listing for the function. The key for determining the logical flow of your program to identify such things as loops is to create a graph of basic blocks. Figure 9 is C source code for a switch statement where the comments indicate the frequency of execution for each case. Figure 10 shows how a compiler may rearrange the order of the assembly language for each of the case statements based upon measured execution frequency. As you can see, the compiler placed the assembly language code associated with the "`case 4`" code that occurs 25% of the time to be the first case tested and if equal control branches to label "`..B1.8`". The code associated with the "`case 8`" is therefore closer address-wise and can take advantage of the spatial locality of the cache. The compiler can determine execution frequency by employing static heuristics or profile-guided optimization.

Figure 9. Source Code for Basic Block Placement Example

```
for (i=0; i< NUM_BLOCKS; i++) {
    switch (check(i)) {
        case 4:     /* 25% */
            ret = 5 ; break;
        case 8:     /* 75% */
            ret = 2 ; break;
        default:    /* 0% */
            ret = 0 ; break;
    }
    printf("%d\n", ret);
}
```

Figure 10. Assembly Language for Basic Block Placement Example

```
..B1.3:           # 100         # Preds ..B1.2
        cmpl      $4, %eax                                  #11.19
        je        ..B1.8        # Prob 25%                  #11.19
                                # LOE rbp r12 r13 r14 r15 eax ebx
..B1.4:           # 75          # Preds ..B1.3
        movl      $2, %edx                                  #17.9
        xorl      %esi, %esi                                #17.9
        cmpl      $8, %eax                                  #17.9
        cmove     %edx, %esi                                #17.9
                                # LOE rbp r12 r13 r14 r15 ebx esi
..B1.5:           # 100         # Preds ..B1.4 ..B1.8
        movl      $_2__STRING.0.0, %edi                     #19.5
        xorl      %eax, %eax                                #19.5
        call      printf
..B1.8:           # 25          # Preds ..B1.3   # Infreq
        movl      $5, %esi                                  #13.9
        jmp       ..B1.5        # Prob 100%                 #13.9
```

Inlining is an optimization that reduces function call and return overhead by replacing a call to a function with the instructions that make up the called function. Once a function is inlined, other optimizations can modify the assembly language to a larger degree resulting in difficult to understand assembly code.

Figure 11 provides a sample implementation of the function check called in the example from Figure 9. Line numbers are added to the listing for convenience. Figure 12 is the assembly listing of function check compiled separately and without being inlined into the code from Figure 9. The line numbers in the assembly listing enable easy correlation to the source code:

- The increment of zee is represented by the first three assembly instructions.
- The modulus operation is represented by the instruction "`andl $3, %eax`".
- The code at labels "`..B1.2`" and "`..B1.3`" represent the if and else clause respectively.

Figure 11. Source Code for Function Check

```
Line
  3    int check(int i)
  4    {
  5        static int zee = 0;
  6        zee++;
  7        if (zee % 4 == 0) return 4;
  8        else return 8;
  9    }
```

Figure 12. Assembly Language for Function Check

```
        movl      zee.2.0.0(%rip), %eax                 #6.5
        addl      $1, %eax                              #6.5
        movl      %eax, zee.2.0.0(%rip)                 #6.5
        andl      $3, %eax                              #7.15
        jne       ..B1.3        # Prob 50%              #7.20
..B1.2:                         # Preds ..B1.1
        movl      $4, %eax                              #7.30
        ret                                             #7.30
                                # LOE
..B1.3:                         # Preds ..B1.1
        movl      $8, %eax                              #8.17
        ret                                             #8.17
```

Figure 13 is an assembly listing of a portion of the code example from Figure 9 after check has been inlined. It is more difficult to correlate the assembly code with the source code. One technique is to use the source code and separate assembly listing of check as a guide. It is easy to observe the increment to zee in Figure 13 (first two instructions in the listing). The return value of the function check which is either 4 or 8 has been optimized away by the compiler since the value is used to control setting of another value, ret. The compiler sets ret directly. This insight into compiler optimization and elimination of code can only be gained from analyzing both the non-inlined and inlined version of the assembly code.

Figure 13. Assembly Language for Inlined Function Check

```
..B1.2:                         # Preds ..B1.7 ..B1.3
        addl      $1, %r12d                                     #19.13
        movl      %r12d, zee.119.0.0(%rip)                      #19.13
        movl      %r12d, %ecx                                   #19.13
        andl      $3, %ecx                                      #19.13
        movl      $5, %r8d                                      #23.9
        movl      $2, %esi                                      #23.9
        cmpl      $0, %ecx                                      #23.9
        cmove     %r8d, %esi                                    #23.9
        movl      $_2__STRING.0.0, %edi                         #27.5
        xorl      %eax, %eax                                    #27.5
        call      printf                                        #27.5
```

Strength reduction is the replacement of one type of operation with a different set of operations that is faster to execute. Some examples of strength reduction are:

- Replacing multiplication with a series of shifts and additions,
- Transforming exponentiation into multiplication,
- Replacing division with reciprocal multiplication, and
- Initialization using less expensive operations.

Figure 14 is a code example and the resulting assembly language that shows several strength reduction optimizations. The compiler optimized the integer multiplication by 4 into a shift left by two bits. It also optimized the floating point division by computing (1.0 / d) outside the loop, storing the result in register XMM1, and multiplying by this value inside the loop. This example shows that you cannot always find the section of assembly that corresponds to a piece of source code by searching for specific distinguishing operations.

Figure 14. Code for Strength Reduction Example

Source Code

```
int i, *f;
double d, *g;

for (i = 0; i < N; i++) {
    f[i] *= 4;
    g[i] = g[i] / d;
}
```

Assembly Implementation

```
..B1.2:                         # Preds ..B1.2 ..B1.1
        shll      $2, (%rdi)                            #8.9
        movsd     (%rsi), %xmm0                         #9.16
        mulsd     %xmm1, %xmm0                          #9.23
        movsd     %xmm0, (%rsi)                         #9.9
        addq      $4, %rdi                              #7.24
        addq      $8, %rsi                              #7.24
        incq      %rax                                  #7.24
        cmpq      $1024, %rax                           #7.21
        jl        ..B1.2        # Prob 99%              #7.21
```

Alternate entries occur as a result of an optimization that reduces the call & return overhead by allowing the calling routine to keep values in registers and jumping to a point of a called function where the registers have the expected values. The IA-32 ISA application binary interface specifies that function call arguments are to be placed on the stack. The called function would then move these items off of the stack into registers which essentially costs multiple moves to and from memory. Figure 15 shows the assembly language for a function that contains the normal function entry point and an alternate entry. This resulting listing is an example using IA-32 ISA assembly language. Notice at the normal function entry point (label "`multiply_d`") the first instructions move the values from the stack into registers whereas the alternate entry (label "`multiply_d`.") assumes the values are already there.

Figure 15. Assembly Code for Alternate Entry Code Example

```
        multiply_d:
..B4.1:           # 1           # Preds ..B4.0
        movl      4(%esp), %eax                         #11.1
        movl      8(%esp), %edx                         #11.1
        movl      12(%esp), %ecx                        #11.1

multiply_d.:                                            #
        pushl     %edi                                  #11.1
```

There are many other compiler optimizations that affect source code position and what is detailed here is just a small sample. Being aware of the issue

and continued study on compiler optimization will improve your skills at correlating assembly code with source code.

Code Sample Assembly Analysis

This section details an analysis of a code sample and its assembly language listings and shows how insight gleamed from the assembly language aids subsequent optimization. Figure 16 is a listing of the code sample whose function is to compute a vector maximum. The listings in this section all use Intel® 64 ISA assembly language. Due to space issues, it is impractical to include the entire assembly list described at each step. Instead, we provide the key portions of the assembly listing relevant to the discussion.

Figure 16. Source Code for Analysis Example (vector_max.c)

```
void vector_max(__int64 *a, __int64 *b, int size)
{
    int i;
    #pragma vector always
    for (i=0;i<size;i++) {
        a[i] = (a[i] > b[i]) ? a[i] : b[i];
    }
}
```

The first step in the analysis is to compile the code to assembly by issuing the following compile command:

```
icc -xSSE4.2 -S vector_max.c
```

The –xSSE4.2 option tells the compiler to take full advantage of all ISA extensions up to and including SSE4.2, and the `#pragma vector always` directive instructs the compiler to always vectorize the loop.

Analyzing the generated assembly listing reveals a great amount about how the compiler optimized the source code. The portion of the assembly listing relevant to our analysis is listed in Figure 17. Analysis reveals these observations:

- The parameters, *a, *b, and size are stored in registers RDI, RSI, and EDX respectively.
- The ‘`testq r8, r8`’ instruction checks to see if size is equal to 0. The function immediately returns if true.
- The code in blocks “`..B1.3`” and “`..B1.4`” is checking the memory region spanned by *a and *b to determine if there is overlap.

The third observation above is critical. The compiler added this overlap checking to ensure safe automatic vectorization. Unfortunately, performing this runtime overlap test increases both runtime and code size overhead. The code size overhead claim is substantiated later in this section.

Figure 17. Assembly Listing 1 for Analysis Example

```
vector_max:
# parameter 1: %rdi
# parameter 2: %rsi
# parameter 3: %edx
..B1.1:                         # Preds ..B1.0
..___tag_value_vector.1:                                        #2.1
        movslq    %edx, %r8                                     #2.1
        movl      %r8d, %eax                                    #
        testq     %r8, %r8                                      #5.14
        jle       ..B1.38       # Prob 50%                      #5.14
                                # LOE rbx rbp rsi rdi r8 r12 r13 r14 r15 eax
..B1.2:                         # Preds ..B1.1
        cmpl      $6, %eax                                      #5.3
        jle       ..B1.32       # Prob 50%                      #5.3
                                # LOE rbx rbp rsi rdi r8 r12 r13 r14 r15 eax
..B1.3:                         # Preds ..B1.2
        cmpq      %rsi, %rdi                                    #6.22
        jbe       ..B1.5        # Prob 50%                      #6.22
                                # LOE rbx rbp rsi rdi r8 r12 r13 r14 r15 eax
..B1.4:                         # Preds ..B1.3
        lea       (,%r8,8), %rdx                                #6.22
        movq      %rdi, %rcx                                    #6.22
        subq      %rsi, %rcx                                    #6.22
        cmpq      %rcx, %rdx                                    #6.22
        jb        ..B1.7        # Prob 50%                      #6.22
```

Assume the programmer knows that the regions pointed to by *a and *b do not overlap. It is possible to eliminate the overhead by using the restrict type qualifier which instructs the compiler to assume that the specified pointer does not point to the same memory as another pointer. Add the restrict type qualifier by changing the parameter definition of function `vector` to the following.

```
void vector_max(__int64 *restrict a, __int64 *b, int size)
```

Using the restrict type qualifier requires an additional option to turn on the feature. Recompile the source file by issuing

```
icc -restrict -xSSE4.2 -S vector_max.c
```

A portion of the resulting assembly file is shown in Figure 18. We made the following observations about the new assembly code:

- The assembly code to check for overlap between *a and *b is not in the listing.
- The code in blocks "B1.2" and "B1.13"checks the alignment of *a and *b respectively and ultimately determines which of two vectorized loops are used to perform the max operation: one that uses unaligned loads (MOVDQU) or one that uses aligned loads (MOVDQA).

This runtime alignment check adds execution time and code size overhead.

Figure 18. Assembly Listing 2 for Analysis Example

```
..B1.2:                         # Preds ..B1.1
        movslq    %edx, %r8                                 #5.3
        movq      %rdi, %rdx                                #5.3
        andq      $15, %rdx                                 #5.3
        testl     %edx, %edx                                #5.3
        je        ..B1.5        # Prob 50%                  #5.3


..B1.13:                        # Preds ..B1.11 ..B1.6
        movl      %edx, %edx                                #6.22
        lea       (%rsi,%rdx,8), %r9                        #5.3
        testq     $15, %r9                                  #5.3
        je        ..B1.18       # Prob 60%                  #5.3
```

If the programmer can guarantee that *a and *b point to 16-byte aligned memory, providing this information to the compiler enables better optimization.

Add the following `#pragma` to the source code before the for loop

```
#pragma vector aligned
```

Recompile the source file by issuing

```
icc -restrict -xSSE4.2 -S vector_max.c
```

Analysis of the resulting assembly file reveals the following.

- The assembly code to check the alignment of *a and *b is not in the listing.
- Only one loop with vector instructions appears in the assembly listing, and this loop only uses aligned forms of the vector instructions.

Figure 19 shows the aligned loop. Another interesting tidbit is that the loop has been unrolled four times. Also note that the compiler makes effective use of the SSE4.2 instruction PCMPGTQ. This instruction does a vector compare of 64-bit integers for greater-than, and is followed by several logical operations that blend the *a and *b vectors based on the result of the compare to compute the vector max.

Figure 19. Assembly Listing 3 for Analysis Example

```
..B1.4:                         # Preds ..B1.4 ..B1.3
        movdqa    (%rdi,%rax,8), %xmm0                    #7.7
        movdqa    16(%rdi,%rax,8), %xmm3                  #7.7
        movdqa    32(%rdi,%rax,8), %xmm6                  #7.7
        movdqa    48(%rdi,%rax,8), %xmm9                  #7.7
        movdqa    %xmm0, %xmm2                            #7.7
        movdqa    %xmm3, %xmm5                            #7.7
        movdqa    %xmm6, %xmm8                            #7.7
        movdqa    (%rsi,%rax,8), %xmm1                    #7.7
        movdqa    16(%rsi,%rax,8), %xmm4                  #7.7
        movdqa    32(%rsi,%rax,8), %xmm7                  #7.7
        movdqa    48(%rsi,%rax,8), %xmm10                 #7.7
        pcmpgtq   %xmm1, %xmm2                            #7.7
        pxor      %xmm1, %xmm0                            #7.7
        pand      %xmm0, %xmm2                            #7.7
        pxor      %xmm1, %xmm2                            #7.7
        movdqa    %xmm2, (%rdi,%rax,8)                    #7.7
        pcmpgtq   %xmm4, %xmm5                            #7.7
        pcmpgtq   %xmm7, %xmm8                            #7.7
        pxor      %xmm4, %xmm3                            #7.7
        pand      %xmm3, %xmm5                            #7.7
        pxor      %xmm4, %xmm5                            #7.7
        movdqa    %xmm5, 16(%rdi,%rax,8)                  #7.7
        pxor      %xmm7, %xmm6                            #7.7
        pand      %xmm6, %xmm8                            #7.7
        pxor      %xmm7, %xmm8                            #7.7
        movdqa    %xmm8, 32(%rdi,%rax,8)                  #7.7
        movdqa    %xmm9, %xmm11                           #7.7
        pcmpgtq   %xmm10, %xmm11                          #7.7
        pxor      %xmm10, %xmm9                           #7.7
        pand      %xmm9, %xmm11                           #7.7
        pxor      %xmm10, %xmm11                          #7.7
        movdqa    %xmm11, 48(%rdi,%rax,8)                 #7.7
        addq      $8, %rax                                #6.3
        cmpq      %rdx, %rax                              #6.3
        jl        ..B1.4        # Prob 82%                #6.3
```

The last portion of this analysis summarizes the code size impact of these optimizations. Table 1 shows the text section size of the object files for the original code, the code with the restrict type qualifier, and the code with both the restrict type qualifier and `#pragma vector aligned`. As you can see, the text size shrinks from 688 bytes to 576 bytes with the restrict type qualifier. With both the restrict type qualifier and the `#pragma vector aligned`, the code size drops to 288 bytes, a significant savings.

Table 1. Code Size Comparison of Analysis Example

Version	Text Section Size (in bytes)
Original	688
Original + restrict	576
Original + restrict + vector aligned	288

Summary

By investing the time to learn the basics of assembly language, you add a valuable tool to your software development toolbox. Analyzing code at the assembly level is often the best way to track down a tricky bug or to tune application performance. Understanding IA-32 and Intel® 64 architecture assembly language is challenging due to the sheer number of instructions, instruction forms, and register sets. But by understanding the capabilities of the architecture, learning how to read an assembly listing, and becoming familiar with common compiler transformations, you will be less daunted by the idea of diving down to the assembly level in your debug and performance work.

Authors

David Kreitzer is a compiler engineer with Software & Services Group at Intel Corporation.

Max Domeika is a technical consultant with Software & Services Group at Intel Corporation.

Acronyms

FPU	Floating-point Unit
Intel® AVX	Intel Advanced Vector Extensions
Intel® SSE	Intel® Streaming SIMD Extensions
ISA	Instruction Set Architecture
SIMD	Single Instruction, Multiple Data
SSSE3	Supplementation Streaming SIMD Extensions 3

References

Domeika, M., Software Development for Embedded Multi-core Systems: A Practical Guide Using Embedded Intel® Architecture, ed. Newnes. 2008, Boston, MA.

§