X86 Disassembly/Assemblers and Compilers

Assemblers

Assemblers are significantly simpler than compilers, and are often implemented to simply translate the assembly code to binary machine code via one-to-one correspondence. Assemblers rarely optimize beyond choosing the shortest form of an instruction or filling delay slots.

Because assembly is such a simple process, disassembly can often be just as simple. Assembly instructions and machine code words have a one-to-one correspondence, so each machine code word will exactly map to one assembly instruction. However, disassembly has some other difficulties which cannot be accounted for using simple code-word lookups. We will introduce assemblers here, and talk about disassembly later.

Assembler Concepts

Assemblers, on a most basic level, translate assembly instructions into machine code bytes with a 1 to 1 correspondence. Assemblers also allow for named variables that get translated into hard-coded memory addresses. Assemblers also translate labels into their relative code addresses.

Assemblers, in general, do not perform code optimization. The machine code that comes out of an assembler is equivalent to the assembly instructions that go into the assembler. Some assemblers have high-level capabilities in the form of *Macros.*

Some information about the program is lost during the assembly process. First and foremost, program data is stored in the same raw binary format as the machine code instructions. This means that it can be difficult to determine which parts of the program are actually instructions. Notice that you can disassemble raw data, but the resultant assembly code will be nonsensical. Second, textual information from the assembly source code file, such as variable names, label names, and code comments are all destroyed during assembly. When you disassemble the code, the instructions will be the same, but all the other helpful information will be lost. The code will be accurate, but more difficult to read.

Compilers, as we will see later, cause even more information to be lost, and decompiling is often so difficult and convoluted as to become nearly impossible to do accurately.

Intel Syntax Assemblers

Because of the pervasiveness of Intel-based IA-32 microprocessors in the home PC market, the majority of assembly work done (and the majority of assembly work considered in this wikibook) will be x86 based. Many of these assemblers (or new versions of them) can handle amd64/x86_64/EMT64 code as well, although this wikibook will focus primarily on 32 bit (x86/IA-32) code examples.

MASM

MASM is Microsoft's assembler, an abbreviation for "Macro Assembler." However, many people use it as an acronym for "Microsoft Assembler," and the difference isn't a problem at all. MASM has a powerful macro feature, and is capable of writing very low-level syntax, and pseudo-high-level code with its macro feature. MASM 6.15 is currently available as a free-download from Microsoft, and MASM 7.xx is currently available as part of the Microsoft platform DDK.

- MASM writes in Intel Syntax.
- MASM is used by Microsoft to implement some low-level portions of its Windows Operating systems.

- MASM, contrary to popular belief, has been in constant development since 1980, and is upgraded on a needs-basis.
- MASM has always been made compatible by Microsoft to the current platform, and executable file types.
- MASM currently supports all Intel instruction sets, including SSE2.

Many users love MASM, but many more still dislike the fact that it isn't portable to other systems.

TASM

TASM, Borland's "Turbo Assembler," is a functional assembler from Borland that integrates seamlessly with Borland's other software development tools. Current release version is version 5.0. TASM syntax is very similar to MASM, although it has an "IDEAL" mode that many users prefer. TASM is not free.

NASM

NASM, the "Netwide Assembler," is a portable, retargetable assembler that works on both Windows and Linux. It supports a variety of Windows and Linux executable file formats, and even outputs pure binary. NASM is not as "mature" as either MASM or TASM, but is a) more portable than MASM, b) cheaper than TASM, and c) strives to be very user-friendly.

NASM comes with its own disassembler, and supports 64-bit (x86-64/x64/AMD64/Intel 64) CPUs.

NASM is released under the LGPL.

FASM

FASM, the "Flat Assembler" is an open source assembler that supports x86, and IA-64 Intel architectures.

(x86) AT&T Syntax Assemblers

AT&T syntax for x86 microprocessor assembly code is not as common as Intel-syntax, but the GNU Assembler (GAS) uses it, and it is the *de facto* assembly standard on Unix and Unix-like operating systems.

GAS

The GNU Assembler (GAS) is the default back-end to the GNU Compiler Collection (GCC) suite. As such, GAS is as portable and retargetable as GCC is. However, GAS uses the AT&T syntax for its instructions as default, which some users find to be less readable than Intel syntax. Newer versions of gas can be switched to Intel syntax with the directive ".intel_syntax noprefix".

GAS is developed specifically to be used as the GCC backend. GCC always feeds GAS syntactically-correct code, so GAS often has minimal error checking.

GAS is available as a part of either the GCC package or the GNU binutils package. [1]

Other Assemblers

HLA

HLA, short for "High Level Assembler" is a project spearheaded by Randall Hyde to create an assembler with high-level syntax. HLA works as a front-end to other compilers such as FASM (the default), MASM, NASM, and GAS. HLA supports "common" assembly language instructions, but also implements a series of higher-level constructs such as loops, if-then-else branching, and functions. HLA comes complete with a comprehensive standard library.

Since HLA works as a front-end to another assembler, the programmer must have another assembler installed to assemble programs with HLA. HLA code output therefore, is as good as the underlying assembler, but the code is much easier to write for the developer. The high-level components of HLA may make programs less efficient, but that cost is often far outweighed by the ease of writing the code. HLA high-level syntax is very similar in many respects to Pascal, (which in turn is itself similar in many respects to C), so many high-level programmers will immediately pick up many of the aspects of HLA.

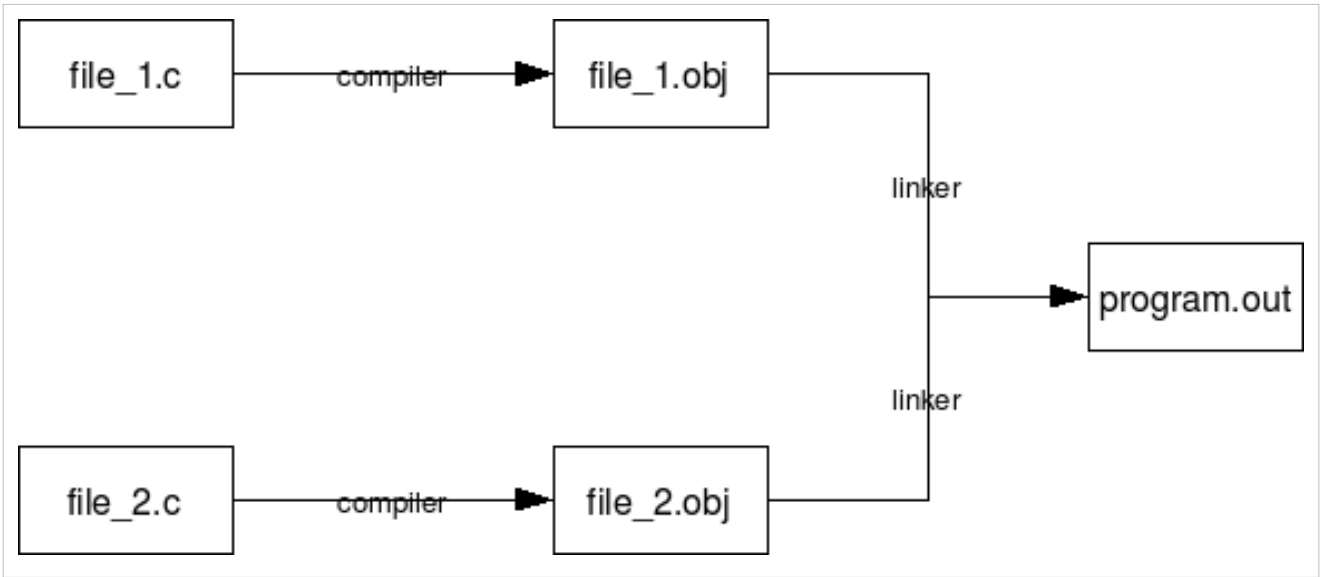
Here is an example of some HLA code:

```
mov(src, dest);  //C++  style comments
pop(eax);
push(ebp);
for(mov(0, ecx); ecx < 10; inc(ecx)) do
   mul(ecx);
endfor;
```

Some disassemblers and debuggers can disassemble binary code into HLA-format, although none can faithfully recreate the HLA macros.

Compilers

A compiler is a program that converts instructions from one language into equivalent instructions in another language. There is a common misconception that a compiler always directly converts a high level language into machine language, but this isn't always the case. Many compilers convert code into assembly language, and a few even convert code from one high level language into another. Common examples of compiled languages are: C/C++, Fortran, Ada, and Visual Basic. The figure below shows the common compile-time steps to building a program using the C programming language. The compiler produces object files which are linked to form the final executable:



For the purposes of this book, we will only be considering the case of a compiler that converts C or C++ into assembly code or machine language. Some compilers such as the Microsoft C compiler will compile C and C++ source code directly into machine code. GCC on the other hand will compile C and C++ into assembly language, and an assembler is used to convert that into the appropriate machine code. From the standpoint of a disassembler, it does not matter exactly how the original program was created. Notice also that it is not possible to exactly reproduce the C or C++ code used originally to create an executable. It is, however, possible to create code that compiles identically, or code that performs the same task.

C language statements do not share a one to one relationship with assembly language. Consider that the following C statements will typically all compile into the same assembly language code:

```
*arrayA = arrayB[x++];

*arrayA = arrayB[x]; x++;

arrayA[0] = arrayB[x++];

arrayA[0] = arrayB[x]; x++;
```

Also, consider how the following loop constructs perform identical tasks, and are likely to produce similar or even identical assembly language code:

```
for(;;) { ... }

while(1) { ... }

do { ... } while(1)
```

Common C/C++ Compilers

The purpose of this chapter is to list some of the most common C and C++ Compilers in use for developing *production-level* software. There are many many C compilers in the world, but the reverser doesn't need to consider all cases, especially when looking at professional software. This page will discuss each compiler's strengths and weaknesses, its availability (download sites or cost information), and it will also discuss how to generate an assembly listing file from each compiler.

Microsoft C Compiler

The Microsoft C compiler is available from Microsoft for free as part of the Windows Server 2003 SDK. It is the same compiler and library as is used in MS Visual Studio, but doesn't come with the fancy IDE. The MS C Compiler has a very good optimizing engine. It compiles C and C++, and has the option to compile C++ code into MSIL (the .NET bytecode).

Microsoft's compiler only supports Windows systems, and Intel-compatible 16/32/64 bit architectures.

The Microsoft C compiler is **cl.exe** and the linker is **link.exe**

Listing Files

In this wikibook, cl.exe is frequently used to produce assembly listing files of C source code. To produce an assembly listing file yourself, use the syntax:

```
cl.exe /Fa<assembly file name> <C source file>
```

The "/Fa" switch is the command-line option that tells the compiler to produce an assembly listing file.

For example, the following command line:

```
cl.exe /FaTest.asm Test.c
```

would produce an assembly listing file named "Test.asm" from the C source file "Test.c". Notice that there is no space between the /Fa switch and the name of the output file.

GNU C Compiler

The GNU C compiler is part of the GNU Compiler Collection (GCC) suite. This compiler is available for most systems and it is free software. Many people use it exclusively so that they can support many platforms with just one compiler to deal with. The GNU GCC Compiler is the *de facto* standard compiler for Linux and Unix systems. It is retargetable, allowing for many input languages (C, C++, Obj-C, Ada, Fortran, etc...), and supporting multiple target OSes and architectures. It optimizes well, but has a non-aggressive IA-32 code generation engine.

The GCC frontend program is "gcc" ("gcc.exe" on Windows) and the associated linker is "ld" ("ld.exe" on Windows).

Listing Files

To produce an assembly listing file in GCC, use the following command line syntax:

```
gcc.exe -S <C sourcefile>.c
```

For example, the following commandline:

```
gcc.exe -S test.c
```

will produce an assembly listing file named "test.s". Assembly listing files generated by GCC will be in GAS format. On x86 you can select the syntax with -masm=intel or -masm=att. GCC listing files are frequently not as well commented and laid-out as are the listing files for cl.exe.

Intel C Compiler

This compiler is used only for x86, x86-64, and IA-64 code. It is available for both Windows and Linux. The Intel C compiler was written by the people who invented the original x86 architecture: Intel. Intel's development tools generate code that is tuned to run on Intel microprocessors, and is intended to squeeze every last ounce of speed from an application. AMD IA-32 compatible processors are not guaranteed to get the same speed boosts because they have different internal architectures.

Metrowerks CodeWarrior

This compiler is commonly used for classic MacOS and for embedded systems. If you try to reverse-engineer a piece of consumer electronics, you may encounter code generated by Metrowerks CodeWarrior.

Green Hills Software Compiler

This compiler is commonly used for embedded systems. If you try to reverse-engineer a piece of consumer electronics, you may encounter code generated by Green Hills C/C++.

References

[1] http://www.gnu.org/software/binutils/

Article Sources and Contributors

X86 Disassembly/Assemblers and Compilers *Source*: http://en.wikibooks.org/w/index.php?oldid=2173592 *Contributors*: Adrignola, AlbertCahalan, Az1568, DavidCary, EleoTager, Gcaprino, Panic2k4, Scientes, Sigma 7, Whiteknight, 35 anonymous edits

Image Sources, Licenses and Contributors

Image:C language building steps.png *Source*: http://en.wikibooks.org/w/index.php?title=File:C_language_building_steps.png *License*: Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors*: Albedo-ukr, Joey-das-WBF, Thedsadude, WikipediaMaster

License