the Assembly Bible (64-bit Linux Edition)

Table of Contents

Introduction or "In the Beginning ... "

Why ?

The eternal answer is, "Why Not ?". In more specific terms, a background in electronics engineering and amateur radio, I work with the components and equipment that make up a larger system.

When studying any form of programming, I have the same mindset; it is all part of the machine. The problem is that High Level Languages (HLL's) make too much abstraction. I have no understanding what is happening or how things work. Too often I see just a strange collection of words, euphemisms and historical acronyms that people once thought were funny.

Assembly has been the only language I got some understanding out of in relation to what the computer was doing for the instructions written in the source code.

You could argue that any HLL can be used in the same way once you set up the function correctly and use object-oriented code. However OO-based philosophies don't quite line up with a view based on circuits and components. I tried OO-code and it just doesn't work. OO-based code is built around hiding what is happening and treats objects like a black box, unseen to the programmer and more so to the PC user.

The functional paradigm is a bit closer to what I am talking about but just replaces objects with functions and does similar black box thinking. It just is not as bad as OO-based stealth.

First learn 32-bit Assembly

Oi, learn the old muck and then learn current assembly code ? Well, yes; in truth not much has changed and the old code can still be used, within the new structure and practice. In reality even old 32-bit code will still work on modern machines if you tell the computer to specifically create a 32-bit program. All new 64-bit processors still use the 32-bit operating modes.

Now the second answer as to why change, is that old 32-bit programs cannot access the larger or more efficient 64-bit capabilities; including new registers, instructions and features like encryption or virtualization. So there are reasons to study or use 64-bit assembly code.

And remember even if you don't write code in assembly, it can help to learn it in order to write better programs in your chosen HLL-based code.

Using Gas ATT Syntax

HLA is not available for 64-bit Assembly; if using 32-bit Assembly, HLA is preferred. I enjoyed studying HLA but caught the tail end of 32-bit CPUs and the author of HLA does not support 64-bit code. Oh, well; remember that 32-bit code still works. See http://webster.cs.ucr.edu for HLA-related information. (Please note that he has retired and the university no longer hosts the website, your browser should be redirected to the more recent site.)

The Gas assembler, as, will be used instead with the ATT syntax; which has operands in the 'source, destination' order, which is easier to read for most people. There are those that think ATT is ugly or confusing. Often this relates to address modes like the Scale Index Address (you'll see this later). While the code for this feature is wierd, each syntax has its own peculiar wierdness that makes others prefer their own chosen syntax.

For me I need the source, destination order of parameters. It just makes more sense to "move this to there" (move $3, %eax) than to say "put in there, this" (mov %eax, $3). This has nothing to do with english or left to right precedence in languages. It has to do with matching cpu code to your language thought process. It would be better if there was a common syntax to match actual cpu operation but nevermind (that's why I kinda liked HLA, the code matched how you think).

Compilers used for 64-bit Assembly

hla – not available for 64-bit code but can output assembly code for review.

as – standard gnu assembler also called Gas (command is "as"). If not installed (should be default in most Linux distros), try installing the **bin-utils** package. Don't forget **binutils-doc** for documentation files.
fasm – a self-written assembler that can produce 32 or 64 bit code. You will have to find and download this yourself as most Linux repos do not keep a copy available. At last check, Debian does have this in their repos.
nasm – the Netwide ASseMbler, the most common alternative to gas.
yasm – the Yet Another ASseMbler, not as common as nasm but comes in close third. Also capable (like HLA) of output in various formats.
ld – linker program to compile object code into an executable. Like as this is also found in the binutils package.

gcc – common Clanguage compiler found in most Linux distros.
clang – alternate C lang compiler found in BSD and other non-GPL distros.
llvm – backend for clang, capable of outputing code in multiple formats including assembly.
lld – linker for clang.

Other useful packages

Editors :
gedit, scite / kate – text editor for either Gnome or KDE (respectively) should already be installed in distro desktop of choice. If not using KDE or Gnome, either should be in your repos. Scite also has a project management package available.
bless, jeex, wxhexeditor / okteta / dhex, hexcurse, hexedit, le, lfhex, tweak – hex editor, displays files in hexadecimal. Used to view sturcture of object files or executables. Bless and okteta are tied into desktops and have a lot of dependencies, hexcurse and hexedit are lighter but not as pretty.
bsdmainutils – has the hexdump utility to display files in hexadecimal. Not an editor like bless or okteta.
xxd – same as hexdump but can be reversed.

Utilities :
make – an utility for managing projects and compiling source code.
pmake / bmake – NetBSD variant of make.
ninja-build – another project/buld management program, aims for speed.
cmake – graphical frontend to make or a standalone make utility.
remake – advanced version of make, includes a debugger.
intel2gas – converts nasm or intel syntax into ATT syntax code.
diffutils / patchutils – Programs that will show differences between files and can patches to update files to match.

Debuggers :
ddd – graphical frontend to gdb, the GNU Debugger.
lldb – debugger for clang / llvm.
kdbg – another frontend to gdb.
gdb – and of course gdb itself.

Other Utilities :
abs-guide – great document on learning to script in the bash shell terminal, common terminal shell used in most Linux distros. Good for automating project management.
build-essential – Not really required but some other packages depend on it. Also used to build packages for Debian-based distros. If building other third-party development software from source, this package may be required. On Debian it is standard to install this when developing software.
freebsdutils – package of BSD variants of common Linux build tools like make. Used with other BSD-based programs on Linux distros.
util-linux – package of useful utilities including linux32 and linux64 which let you execute programs built for another cpu architechture.

Getting Assembly code from gcc or clang

Two most common programs for C language (an HLL close to machine level code like assembly), gcc and clang, can produce assembly code as part of their normal operation. The gcc, Gnu Compiler Collection, is a series of programs for programming in a number of languages. Clang is a library that will call the correct compiler for the selected language. Both produce assembly in the ATT syntax for compiling into a program.

```
gcc -O2 -S file.c
clang -O2 -S file.c
```

Where file is the original c source code filename. These are handy commands to get assembly output from C source code or to see what Assembly code looks like that these programs produce. Most applications are not developed in Assembly anymore but it is still used as an intermediate format on Linux systems by gcc, cc or clang. Note that clang can also produce it own format of Assembly code. We will not be using that format / syntax.

The option "-O2" (the capital letter O) specifies optimization level, it may be omitted for the command. The second level of optimization is not complicated and will display common adjustments that you make by hand.

The option "-S" (the capital letter S) tells the compiler to stop processing after writing the Assembly file. The resulting file should be named file.s; where file is the filename of the input C source code file.

You can include a "-v" option to produce verbose output for most Linux programs. Others will print a version number to the screen instead. The "-v" option for gcc or clang should be verbose, it will print every action it takes. The "-###" option is similar but will not actually execute the commands.

Gas is installed with the binutils package and is common to Linux and BSD, it should already be installed or easily installed. NASM and YASM have packages in most repos for download.

HLA is available for Windows, Mac, Linux and BSD. It can output assembly code for HLA, Gas, FASM, MASM (microsoft's macro assembler) and maybe others.

For this text I'll be using **as / ld** with **make** and **ddd** or **kdbg**. My Linux system uses KDE for the desktop and kde-based software and utilities like **Kate** will be used.

What does Assembly look like ?

Here is the ever common (popular ?) "Hello, World" in Assembly Code :

HLA (32-bit) :

```
program helloWorld;

#include( "stdlib.hhf" );

begin helloWorld;
stdout.put( "Hello, World of Assembly Language", nl );
end helloWorld;
```

Gas (32-bit) :

```
.global _start
.data
msg:
    .ascii "Hello, World!\n"
    len = . - msg
.text
_start:
    movl     $4, %eax
    movl     $1, %ebx
    movl     $msg, %ecx
    movl     $len, %edx
    int      $0x80
    movl     $1, %eax
    movl     $0, %ebx
    int      $0x80
```

Gas (64-bit) :

```
.global _start
.data
msg:
    .ascii "Hello, World!\n"
    len = . - msg
.text
_start:
    movq     $1, %rax
    movq     $1, %rdi
    movq     $msg, %rsi
    movq     $len, %rdx
    syscall
    movq     $60, %rax
    movq     $0, %rdi
    syscall
```

What does it all mean ?

At the start of either source code you find :

```
program helloWorld;
```

or

```
.global _start
```

both tell the compiler where the program starts, the function that is the first instruction that is executed. In the case of HLA, it is the name that follows the "program" keyword (or directive). These are instructions executed by the compiler and not the computer cpu itself. For Gas, it is the keyword ".global" which in Gas all keywords start with a period (' . ') .

It is a convention for assembly in Gas and most other assemblers to use an underscore (' _ ') and the name "start" or "main" for the beginning of a program. This is also called the "entry point". Collectively all names used in this fashion are called " Labels ". Labels are unique in both case and spelling. Treat Label, label and LABEL as different names unless documentation for your assembler says otherwise.

In Linux and using C library functions, use the label "_main" for your entry point, it is expected by the library functions.

This code :

```
#include( "stdlib.hhf" );
```

is an example of a directive in HLA; it is a common include statement found in most HLL's, like C or C++. In most HLL's (high-level languages), the statements are terminated by a semi-colon, in assembly this is optional unless the documentation says otherwise. In general you should get used to using them.

```
.data
msg:
    .ascii "Hello, World!\n"
    len = . - msg
```

The above code is declaring a variable in Gas assembly. The HLA example does not have a variable as it is declared in the function itself. The ".data" directive indicates the section of code is used for storing information. Data sections must have storage of memory reserved but you do not have to give the variable a value, it may be empty. An empty value is assumed to be 0 (not null or actually empty).

The "msg:" is a label and indicates the name of a variable. It does not need to be on a separate line as written here.

The ".ascii" directive indicates value type or size. In this case it indicates that the data is an ascii string. Strings are quoted and may have what is called an escape sequence (the ' \ ' character). It is used to indicate a control sequence or other character not normally printed. Here it marks the newline character, telling the assembler to print any information that follows on a new line in the terminal (like your command prompt).

Strings should be zero-terminated unless the .asciz directive is used. Here the terminator is neglected but the program compiles anyways, oops. The length of the string is specified and can be used anyways. Do not rely on that, always zero terminate strings unless declaring them with .asciz . Strings are quoted with double-quotes. If a double quote is needed in the string, use an escape character. If the escape character is needed, use it twice (' \\ ').

The line "len = . - msg" is an expression to find the length without counting the number of characters in each string declared. In short it means to find the length by starting here (the ' . ' character) and counting back to the beginning (' - msg '). What you are doing is taking the memory address of the end of the string and subtracting the memory address of the beginning, thus getting the total length of the string.

```
begin helloWorld;
```

This is the _start label of an HLA program. For gas it is indicated by :

```
.text
_start:
```

The ".text" is the directive declaring the section that holds the instructions or functions to be executed. In some other assemblers the directive ".code" is used instead. The reason for the names are historical and are not needed to be known. The label "_start:" was declared by the .global directive (or ' .globl ') and must match here or the code won't compile.

This is the function or instructions to be executed by the program :

```
stdout.put( "Hello, World of Assembly Language", nl );
end helloWorld;
```

As explained earlier, HLA can have the string declared in the function instead of creating a variable. Either method could have been used here. Like Gas the string is quoted but the newline sequence follows the string as ", nl);" . This is not required, HLA could use escape sequences as well. Either method is acceptable in HLA.

The line "end helloWorld;" indicates the terminus of the program. Here it ends and the program returns control to the Operating System. For Gas it is not required as we will use the exit system call which does the same thing. If not, you can use the ".end" directive which is the same thing as what HLA uses.

```
    movq     $1, %rax
    movq     $1, %rdi
    movq     $msg, %rsi
    movq     $len, %rdx
    syscall
```

In Gas for 64-bit assembly, these five lines are used in place of the single "stdout.put" function from HLA. If you ask HLA to output assembly source file instead of compiling to a program you should see a similar output of five lines.

The "movq" instruction tells Gas to move a 64-bit sized value from the number one (decimal) to the register rax (registers are prefixed by ' % ' in ATT syntax). You can omit the % prefix if you set the assembler to the ".att_syntax noprefix" directive before the .data or .text sections (or use -mnakedreg at the command line). There is no reason to do this for HLA.

The "syscall" instruction tells Gas to call the Linux system command to write a value. In 32-bit code this is $4 instead (syscalls are remapped between 32- and 64-bit values).

An instruction or opcode (operating code) are followed by the values to be operated on. Depending on the instruction these can be between zero or four values (not usually higher than four, and rarely at that).

Gas uses a comma to separate values in source then destination order. Intel syntax and other assemblers may place the values in destination then source order. Be careful when reading code from other assemblers. HLA also uses source then destination order unless told to produce code for other assemblers.

In Gas an immediate value (one typed directly in the instruction as shown) is always prefixed with a dollar sign character (' $ '). And as stated the registers can be either ' % ' or not depending on configuration, by default they are prefixed.

A register is a storage value found in the cpu. It is an actual piece of hardware that is physical memory. The size of registers vary according to CPU. There will be a list of registers a little bit later.

For 32-bit programs these values change a bit. You will notice the registers start with an ' e ' instead of an ' r '. This is a particular difference between 32- and 64-bit codes; like register sizes. Another difference is the replacement of "syscall" with "int $0x80". This is because a direct call to the interrupts of the computer (' int ' means interrupt in this context), is not available in 64-bit programming. You must call syscall (which uses an OS-based interface) instead.

Otherwise you will see a one to one match between 32- and 64-bit programming in assembly.

The last bit of code is :

```
    movq     $60, %rax
    movq     $0, %rdi
    syscall
```

These three lines are equal to the "end helloWorld;" from the HLA code. It terminates the program by calling the exit system call of the Unix operating system (be it Linux or BSD, maybe MacOS too). Each OS will use a different numeric value for the syscall. In 32-bit code it is $1 while in 64-bit code it is $60, there is a C language header file used in the source code that will indicate the values for different system calls. In Linux you should find this C header in the location "/usr/include/x86_64-linux-gnu/asm/" as either "unistd_32.h" or "unistd_64.h" depending on wether you are writing 32- or 64-bit code.

The exact location may vary depending on your Linux distro, the above example was for Debian Linux (version 9, to be exact).

The line "movq $0, %rdi" gives an exit status to the Linux OS. The exit syscall does not do this itself, you have to specify it in the program.

How do I make this program ?

Once you have written the source code into a plain text file (do not use a word processor unless you can save the file as plain text), you create the program by running two programs; the assembler and the linker.

An Assembler is like a compiler in a high-level language, it takes the code you have written and translate it into the instructions a computer understands. This file is then called an object file. Technically you are finished as there is little difference between an object file and an executable. On some embedded platforms, like Arduinos and ESP32 chips, this is in fact the last step and you upload the binary into the chip as firmware. However, for more complex computers there is one more step.

The second program is called the Linker, this program converts the object code into an executable file for your PC. This conversion takes the information used by the object code and puts it into a format for your PC and operating system. Linux and other OS's are more complex and require additional information to process an executable and understand the format. A Linker handles this responsibility. If you compare an object file to an executable, you will find the same instructions with extra fields for the format and processing of the program.

Randall Hyde's book series "Write Great Code" has excellent examples of this comparision.

To assemble code in Linux, which is used in this text, you need the 'as' program from the binutils package. This package might already be installed in your distro or it is easily found in the repos for your distro. On BSD this should already be installed, or a similar assembler is available.

The command to build 64-bit code is :

```
    $ as -v -o hello.o hello.s
```

To build 32-bit code on a 64-bit processor is :

```
    $ as -v --32 -o hello.o hello.s
```

Be sure two hyphens are used in front of 32 in that command (in case the word processor spellchecks them to something else). The "-v" parameter sets the program to report the version of the assembler, it can be omitted, Linux has a convention of "-v" to produce verbose output of what is happening for troubleshooting purposes. Both assembler and linker are based of old standards and may or may not honor that convention; do not worry about it. The "--32" parameter tells the OS to build code for 32-bit systems, it can be omitted to build 64-bit code. This works if as is being run on a 64-bit machine, otherwise use "--64" to build 64-bit code specifically. To produce object code using a specific name, use the "-o" parameter followed by the name to be used.

By convention all object code uses the ".o" file extension. The last parameter is the assembly source code to be built. These files use the ".s" file extension by convention.

The dollar sign used in commands is a Linux convention to indicate a user command prompt. It is not required to type that when giving a command, it should already be displayed at the end of your terminal prompt.

A root terminal (or the administrator in Windows-speak) is indicated by using a "#" character instead.

To link object code use a program called "ld", like as this is installed with binutils. Same install procedures apply, you should find it in the repos if not already installed on your Linux distro.

To build object code into 64-bit executable :

```
$ ld -v -m elf_x86_64 -o hello hello.o
```

To build obejct code into 32-bit executable on a 64-bit cpu :

```
$ ld -v -m elf_i386 -o hello32 hello32.o
```

Both as and ld share similar output parameters. Specific use is given below. For simple compiling of code they mean the same thing.

If not specified, the output file will be called "a.out"; for use in a binary only capacity (like the firmware example used earlier). These files can still be executed as programs. But it is preferred to specify an output name.

In order to avoid confusion, I named the 32-bit version with a 32 suffix to the file names. Note the ELF format indicated by the "-m parameter". Windows prefers to use the COFF format for executables and Linux/BSD/Unix uses ELF formats. Gas/as/ld can produce either and also the Windows PE format for executables. This ELF or COFF is the format information mentioned earlier that is the difference between an object file and an executable. Again, the books by Randall Hyde for "Write Great Code" does explain this difference. For the most part you will build using the format for your OS and there is no need to be concerned with this information.

The "-m" parameter to specify object format is not required for 64-bit code if built on 64-bit cpu's, you may omit it.

When you need to build a specifically efficient and fast program is when the object/executable format will become important. Keep the info handy in case but do not be worried about it.

You can easily search for these commands using "$ which as" or "$ whereis as", if the system returns a file path to your screen; then the program is installed. Same applies for the ld program. Note that how to use these programs are found in the manual pages either "$ man as" or "$ man ld"

'as' lives at :

```
$ which as
/usr/bin/as
```

'ld' lives at :

```
$ which ld
/usr/bin/ld
```

for 'gcc' and 'clang' from earlier :

```
$ which gcc
/usr/bin/gcc
$ which clang
/usr/bin/clang
```

While HLL programmers can use gcc to build their program in one command, using assembly directly requires the manual use of both as and ld to compile code. GCC and Clang build code automatically but they also build the assembly source file instead of skipping that step and building straight to the object code. You can make use of this to read the assembly code they build if you wanted to examine it. These commands were given near the beginning of the text.

```
gcc -O2 -S hello.c
clang -O2 -S hello.c
```

Will assume that hello.c is the HLL equivalant of the aseembly code we have just been working with.

Here is ths C code :

```
#include <stdio.h>
int main(void)
{
    printf("Hello World!\n");
    return 0;
}
```

Now the directive should look familiar, it is how all HLL's write their directives. The line "int main(void)" is the function declaration of the entry point as C specifies it. There are other parameters and function information but this is the simple use of it for our purpose. The { or } on thier own lines are for pretty-ness. These mark the instructions carried out by the function (including other functions). The "printf("Hello World\n");" line is what is called a C Format string, again you will find most HLL's write their strings in this way. And finally a "return 0;" which is the exit code for the return status to the Linux OS.

Now here is the assembly code produce by the -S parameter in gcc :

```
     .file     "hello.c"
     .section  .rodata
.LC0:
     .string   "Hello World!"
     .text
     .globl    main
     .type     main, @function
main:
.LFB0:
     .cfi_startproc
     pushq     %rbp
     .cfi_def_cfa_offset 16
     .cfi_offset 6, -16
     movq %rsp, %rbp
     .cfi_def_cfa_register 6
     leaq .LC0(%rip), %rdi
     call puts@PLT
     movl $0, %eax
     popq %rbp
     .cfi_def_cfa 7, 8
     ret
     .cfi_endproc
.LFE0:
     .size     main, .-main
     .ident    "GCC: (Debian 6.3.0-18+deb9u1) 6.3.0 20170516"
     .section  .note.GNU-stack,"",@progbits
```

Okay, break this down to something simple. First I'll remove all the debug symbols and sections. We want something simple to show like our hand-coded example.

```
     .file     "hello.c"
     .section  .rodata
.LC0:
     .string   "Hello World!"
     .text
     .globl    main
     .type     main, @function
main:
.LFB0:
     pushq     %rbp
     movq      %rsp, %rbp
     leaq      .LC0(%rip), %rdi
     call      puts@PLT
     movl      $0, %eax
     popq      %rbp
     ret
```

Alright we can work with this. First the lines ' .file "hello.c" ' and ".section .rodata" are optional. The first indicates the source file and is not needed, while the second is a named section for our string data. We can use the .data section for this and making a specific read-only section is not required. It would have been easier to just use .data instead.

The label .LC0 just indcates our variable, we named it msg specifically earlier; here they just used a label for the whole thing instead. This causes no real difference but our way may be more readable.

GCC also put the .global directive after the .text section; not a problem just a readablility issue. The "@function" value just indicates our entry is a function. This is not really neccesary and can be omitted. Also the .type section is not needed either.

The "main:" label is then followed by another label ".LFB0:", this is not needed but they created a label for the function printing the string like they did for the string itself. Not required, but not an issue either.

The "pushq `%rbp`" and "popq `%rbp`" instructions from the function are new; they preserve what is called the stack. First you push the stack to preserve what was there and then you pop it to remove your data and restore the stacks last value. Our program was simpler and didn't affect the stack. But here they manipulated it a bit.

The stack is a temporary place to hold data while the program is running. It will be brought up again later.

The "leaq `.LC0(%rip), %rdi`" instruction is also new and so is the value it uses. This called 'load effective address', lea is used to find memory address of a value rather than specify it yourself. Sometimes a value might not be where you expect it based on how the program runs. Using lea is a safer alternative. The value ".LC0(%rip)" means find the address starting at rip and move a distance equal to .LC0 size. This gives the address of the string you want to print (remember that .LC0 is the label for the string).

The "call" instruction here is making a function call and not a syscall. It is getting the C function puts (put string) from the C std (standard) library. This is known as an External Function, since it is not in the assembly source itself. This is unneccessary but is an example of using an existing function from another file. It would be simpler to call the write function yourself.

The last new instruction is "ret", which means return. It is used after the call function is done (the callee) and gives control back to the original program (the caller). You will notice that an exit status is given, "movl $0,%eax", but no exit syscall is made. This example shows GCC letting the program run its course instead of specifically making a graceful exit. This is bad form and you should always make an exit syscall (actually GCC is making the exit call from the puts function but that isn't visible from the assembly code and looks like it isn't made,hence why this code is a little more unreadable.).

Final Review

The hand-coded program was 15 lines and before removing the debug information, the GCC compiler was 26 lines (16 after). For a simple hello world example, 10 lines (almost double) is pretty bad for an HLL compiler. And Clang's code is worse. It's 40 lines and I just won't bother with it. Now it uses alignment directives (.align) which is a good thing to keep your code on specific memory boundaries, but it is full of the same debug mess and uses some wierder function calls. It is a fine example of a compiler not producing human-readable code for the sake of better efficiency.

Running away with it

Finally you can execute your program from the command line of Linux with either the 32-bit or 64-bit version, depending on which you compiled.

```
$ ./hello
Hello, World!
$
```

```
$ ./hello32
Hello, World!
$
```

So now what ?

Now go over the structure of an assembly program and the syntax of the language. Lets start making some headway into 'understanding the machine' as a previously specified author might put it.

First, remember this is using ATT syntax because it uses the source, destination order in opcodes and makes it easier to read instructions. This is the default in Linux and as but can be specified by using ".att_syntax" in the directives. You can specify ".att_syntax noprefix" if you don't want to use ' % ' before an register name.

Second, this will be Gas on a 64-bit machine. There may be an occasional mention of HLA or 32-bit code but this is not the preferred code we will be using. On a 64-bit cpu, no parameters need to be given. You can specify bitness of the code by using ".code32" or ".code64" directives or using "--32" or "--64" parameters for the as program. If you need to, there is also a ".code16" directive for the source code file.

When running your own programs, be sure to use "./" in front of your program's name. This is because your program won't be in the normal places Linux looks for binary executables unless you tell it where to look. The "." tells Linux to start here or "this" and the "/" character is used to define a directory or a sub-directory (a folder in Windows-speak). Even though your program may be sitting in the top folder of a directory, you always use "/" before the name because the contents of a folder is always a sub-directory of the "this" character.

```
$ ls -a1
.
..
hello
hello32
hello32.o
hello32.s
hello.o
hello.s
makefile
```

The "ls -a1" is a command to list (ls) all programs (-a) in the current directory one per line (-1). As you can see there is an entry for "." and "..", the first is the name for "this" and the second indicates the parent directory. All contents of a folder are children of this so to run either executable above in linux be sure to use "./hello" or "./hello32", the alternative is to either copy these programs into a location Linux will check or add the current directory to what is called the $PATH variable. Both can be a security risk so run the programs yourself instead.

Genesis

(Source Code Grammar and CPU design)

Source Code

Registers

Memory

Stack

Directives

Literals and Datatypes

Op Code Instructions

Macros

Conditional Source

Exodus

(External Programs and Source Code)

Assembler

Linker

External Includes

External Libraries

Shared Binaries

Debugger

Leviticus
(Standards, Conventions and Practices)

Numbers

(Integers, Numbers and Math)

Datatypes

Arithmetic Instructions

Floating–Point Math

Optimizing Math

Deuteronomy

(Characters and Strings)

Apocrypha
(Assembly–written Software)

References

(What is Expected to be Remembered)

Linux bash Shell

Assembler Options

Linker Options

Make Options

Directives

Instructions

Most x86 (32-bit) instructions are still usable in 64–bit code except where otherwise depreciated or invalid. Instructions that assume genreal–purpose register use, use the 64–bit registers instead of the 32–bit registers.

Format = Op Code (Source), Op (Src, Dest) or Op (Src1, Src2, Dest)
IRM or I/R/M = Immediate (or Literal), Register or Memory
Italic *sz* indicates a size code (data type)

Code	Integer Types	Bit Size	Code	Float Types	Bit Size
b	byte	8	ss	scalar, single	64 / 128
w	word	16	sd	scalar, double	128
l	dword	32	ps	packed, single	128 / 256
q	qword	64	pd	packed, double	128 / 256

Italic *cc* indicates a required condition code.

Code	Condition	FLAGs
a / nbe	above or not below and equal	CF = 0 ZF = 0
ae /nb	above and equal or not below	CF = 0
b / nae	below or not above and equal	CF = 1
be / na	below and equal or not above	CF =1 or ZF = 1
e / z	equal or zero	ZF = 1
ne / nz	not equal or not zero	ZF = 0
g / nle	greater or not less than and equal	ZF = 0 SF = OF
ge / nl	greater and equal or not less than	SF = OF
l / nge	less than or not greater and equal	SF != OF
le / ng	less than and equal or not greater	ZF = 1 or SF !=OF
s	signed	SF = 1
ns	not signed	SF = 0
c	carry	CF = 1
nc	not carry	CF = 0
o	overflow	OF = 1
no	not overflow	OF = 0
p / pe	parity or parity even	PF = 1
np po	no parity or parity odd	PF = 0

Invalid / Depreciated Instructions

aaa, aad, aam, aas, bound, daa, das, into, popa, popad, pusha, pushad

SIMD Instructions

[Single Instruction Multiple Data (SIMD)]
[Streaming SIMD Extensions (SSE)]
[Advanced Vector eXtensions (AVX)]

Instructions flagged with SSE or AVX in their Use column are floating point or packed types used in SSE, SSE4/5, AVX or AVX256 instructions. A compatible CPU is required (most modern CPUs circa 2000-ish should have up to AVX instructions). Note that if the opcode is defined in more than one version (maybe additional formats were added), the higher version is used for the label.

Some instructions require specific feature sets. In addition to an AVX compatible cpu, the cpu must also support the indicated feature. Most features are supported by a cpu that also supports AVX2 instructions. Some common sets are FMA or BMI.

Transfer Instructions

Moving or exchanging data.

Op Code	Format	Use
mov*sz*	Op (I/R/M, R/M) Op (R, R/M)	Move value from source to destination SSE2, Move float from XMM to XMM register or memory
movsx	Op (R/M, R)	Sign–extend source into destination.
movzx	Op (R/M, R)	Zero–extend source into destination.
cmov*cc*	Op (R/M, R)	Move if *cc* condition evaluates.
push	Op (I/R/M)	Push value from source onto stack.
pop	Op (R/M)	Pop value from stack into destination.
xadd	Op (R, R/M)	Exchange data and place sum in destination
xchg	Op (R, R/M)	Exchange data between source and destination

Transfer Instructions (SSE opcodes)

Moving or exchanging data. These require an SSE compatible cpu, most modern PCs should have these opcodes available. There are several versions of these extensions, your cpu needs to support the specified version. Most modern PCs should support SSE/SSE2 and maybe SSE3. More recent PCs can support SSE 4, 4.1, 4.2 or SSE5 opcodes.

Op Code	Format	Use
mova*sz*	Op (R/M, R)	SSE2, Move pack type from XMM to XMM.
movu*sz*	Op (R/M, R)	SSE2, Move (unaligned) pack type from XMM to XMM.
movl*sz*	Op (R/M, R/M)	SSE2, Move low–order qword into destination.
movh*sz*	Op (R/M, R/M)	SSE2, Move high–order qword into destination.
movlh*sz*	Op (R, R)	SSE, Move low–order qword into high–order destination.
movhl*sz*	Op (R, R)	SSE, Move high–order qword into low–order destination.
movmsk*sz*	Op (R/M, R)	SSE2, Store sign bits into low bits of dest. High bits zero
movdqa		SSE2,
movdqu		SSE2,
movq2dq		SSE2,
movdq2q		SSE2,
movsldup	Op (R/M, R/M)	SSE3, Copies low qword into dest then copies into high.
movshdup	Op (R/M, R/M)	SSE3, Copies high qword into dest then copies into low.
movddup	Op (R/M, R/M)	SSE3, Copies low qword into low and high of dest.
pmovsx*szsz*		SSE4.1, sign extend low-order into destination. (bwdq)
pmovzx*szsz*		SSE4.1, zero extend low-order into destination. (bwdq)
pmovmskb		Move byte mask.

Transfer Instructions (AVX opcodes)

Moving or exchanging data. These require an AVX compatible cpu, most recent modern PCs should have these opcodes available.

Op Code	Format	Use
vbroadcast*sz*		AVX2, Copy value (i/f) to all positions in dest.
vbroadcastf128		AVX, Copy 128-bit float to all positions.
vbroadcasti128		AVX2, Copy 128-bit integer to all positions.
vmaskmov*sz*		AVX, Condition copy float src2 to dest by src1 mask.
vpmaskmov*sz*		AVX2, Condition copy int src2 to dest by src1 mask.
vgather*szsz*		AVX2 Condition copy 4 or 8 floats or integers from memory by VSIB address.

Arithmetic Instructions

Common arithmetic and mathematical instructions.

Op Code	Format	Use
add	Op (I/R/M, R/M) Op (R, R/M)	Sign/Unsign addition of source to destination. SSE2, Add float from XMM to destination.
adc	Op (I/R/M, R/M)	Sign/Unsign carry addition of source to destination.
sub	Op (I/R/M, R/M) Op (R, R/M)	Sign/Unsign subtraction of source from destination. SSE2, Subtract float XMM from destination.
sbb	Op (I/R/M, R/M)	Sign/Unsign carry subtraction of src from dest.
imul	Op (R/M) Op (I/R/M, R/M) Op (I, R/M, R)	Signed Multiplication, assumes RAX is src. Signed Multiplication, stores result in dest. Signed Multiply of two sources, stores result in dest.
mul	Op (R/M) Op (R, R/M)	Unsigned Multiplication, assumes RAX is src. SSE2, Multiply destination by float XMM.
idiv	Op (R/M)	Signed division, assumes RAX is src. Stores result in RAX:RDX (quotient:remainder)
div	Op (R/M) Op (R, R/M)	Unsigned division, assumes RAX is src. Stores result in RAX:RDX (quotient:remainder) SSE2, Divide destination by float XMM.
inc	Op (R/M)	Increase source by one. Does not affect FLAGS.
dec	Op (R/M)	Decrease source by one. Does not affect FLAGS.
neg	Op (R/M)	Two's complement negation

Arithmetic Instructions (SSE)

Common arithmetic and mathematical instructions.

Op Code	Format	Use
addsub*sz*	Op (R/M, R/M)	SSE3, Add odd packed and subtract even packed data.
hadd*sz*	Op (R/M, R/M)	SSE3, Add adjacent values between two packed types.
hsub*sz*	Op (R/M, R/M)	SSE3, Subtract adjacent values between two packed types.
sqrt*sz*	Op (R/M)	SSE2, Calculate square root of source.
max*sz*	Op (R, R/M)	SSE2, Compare source to dest store highest in dest.
min*sz*	Op (R, R/M)	SSE2, Compare source to dest store lowest in dest.
round*sz*	Op (R, R/M)	SSE4.1, Round value as indicated by second operand.
rcpss	Op (R/M)	SSE, Compute reciprical of value.
rsqrtss	Op (R/M)	SSE, Compute reciprical square root of value.
dp*sz*	Op (R/M)	SSE4.1, dot product of packed type
pmulld		SSE4.1,
pmuldq		SSE4.1,
pminu*sz*		SSE4.1, b,w or d, Compare and store lowest unsign value.
pmins*sz*		SSE4.1, b,w or d, Compare and store lowest sign value.
pmaxu*sz*		SSE4.1, b,w or d, Compare and store highest unsign value.
pmaxs*sz*		SSE4.1, b,w or d, Compare and store highest sign value.

Arithmetic Instructions (AVX)

Common arithmetic and mathematical instructions.

Op Code	Format	Use
vfmadd132*sz* vfmadd213*sz* vfmadd231*sz*		FMA4, fused mathematics according to indicated algorithm. Same algorithm used for all fuse instructions. dest = src * src + src
vfmsub*sz*		FMA4, fused mathematics according to indicated algorithm. dest = src * src - src
vfmaddsub*sz*		FMA4, fused mathematics according to indicated algorithm. odd: dest = src * src + src even: dest = src * src - src
vfmsubadd*sz*		FMA4, fused mathematics according to indicated algorithm. odd: dest = src * src - src even: dest = src * src + src
vfnmadd*sz*		FMA4, fused mathematics according to indicated algorithm. dest = -(src * src) + src
vfnmsub*sz*		FMA4, fused mathematics according to indicated algorithm. dest = -(src * src) - src
mulx		BMI2, unsigned mult EDX and src into dest1 and dest2, no FLAGs.
rdrand		RDRAND, load hardware random number into dest register.

Comparison Instructions

Test instructions comparing two values and setting FLAG register based on results.

Op Code	Format	Use
cmp	Op (I/R/M, R/M) Op ()	Compares two values and sets FLAGs accordingly. SSE2, Compares two floats and stores result in dest.
cmps*sz*		Compares memory pointed by ESI and EDI, sets FLAGs.
cmpxchg	Op (R/M)	Compares values with RAX and exchanges based on results.
cmpxchg8b	Op (R/M)	Compares RAX to 8-bit values and exchanges based on results.
cmpxchg16b		Compares RDX:RAX to 16-bit values and exchanges based on results.

Comparison Instructions (SSE)

Test instructions comparing two values and setting FLAG register based on results.

Op Code	Format	Use
comi*sz*	Op (I/R/M, R/M)	SSE2, Ordered compare of two values, sets FLAGs
ucom*sz*	Op (I/R/M, R/M)	SSE2, Unordered compare of two values, sets FLAGs
pcmpeq*sz*		SSE4.1, bwdq,
pcmpgt*sz*		SSE4.2, bwdq,

Comparison Instructions (AVX)

Test instructions comparing two values and setting FLAG register based on results.

Op Code	Format	Use

Conversion Instructions

Converts data from one type to another. Will typically use RAX or RDX:RAX for conversion.

Op Code	Format	Use
cbw	Op	Sign–extend byte and store into word value. AL -> AX
cwde	Op	Sign–extend word and store into dword value. AX -> EAX
cwd	Op	Sign–extend word and store into dword value. AX -> DX:AX
cdq	Op	Sign–extend dword and store into qword value. EAX -> EDX:EAX
cdqe		Sign–extend dword and store into qword value. EAX -> RAX
cqo		Sign–extend dword and store into qword value. RAX -> RDX:RAX
bswap	Op (R)	Reverses bytes from Little–Endian ot Big–Endian.
movbe	Op (R, M) Op (M, R)	Reverses source and stores in destination.
xlatb	Op	Convert AL according to lookup table in EBX

Conversion Instructions (SSE)

Converts data from one type to another. Will typically use RAX or RDX:RAX for conversion.

Op Code	Format	Use
cvtsi2*sz*	Op (R/M, R)	SSE2, Convert sign-dword into float.
cvt*sz*2si	Op (R/M, R)	SSE2, Convert float into sign-dword.
cvtt*sz*2si	Op (R/M, R)	SSE2, Convert float into sign-dword. Truncates values.
cvt*sz*2*sz*	Op (R/M, R)	SSE2, Convert float to float (change precision).
cvtpi2*sz*	Op (R/M, R)	SSE2, Convert packed integers into pack floats.
cvt*sz*2pi	Op (R/M, R)	SSE2, Convert pack float into pack integers.
cvtt*sz*2pi	Op (R/M, R)	SSE2, Convert pack float into pack integers, Truncates.
cvtdq2*sz*	Op (R/M, R/M)	SSE2, Convert pack qword into pack floats.
cvt*sz*2dq	Op (R/M, R/M)	SSE2, Convert pack float into pack qwords.
cvtt*sz*2dq	Op (R/M, R/M)	SSE2, Convert pack float into pack qwords. Truncates.
cvt*sz*2*sz*	Op (R/M, R/M)	SSE2, Convert pack float into pack float. Change prec.
packuswb		SSE4.1, Convert pack to 2x pack integers, saturation.
packusdw		SSE4.1, Convert pack to 2x pack integers, saturation.

Conversion Instructions (AVX)

Converts data from one type to another. Will typically use RAX or RDX:RAX for conversion.

Op Code	Format	Use
vcvtph2ps		AVX, Convert half-precision float to single precision.
vcvtph2ps		AVX, Convert single precision float to half-precision.

Logical Instructions

Boolean tests of comparison, unless otherwise stated these are bit–wise operations (except for not).

Op Code	Format	Use
and	Op (I/R/M, R/M)	And evaluation SSE2, And evaluation of each pack type.
or	Op (I/R/M, R/M)	Or evaluation. SSE2, Or evaluation of each pack type.
xor	Op (I/R/M, R/M)	Exclusive OR evaluation. SSE2, Exclusive OR evaluation of each pack type.
not	Op (R/M)	One's compliment negation.
test	Op (I/R/M, R/M)	Bitwise test to affect FLAGs, no results stored.

Logical Instructions (SSE)

Boolean tests of comparison, unless otherwise stated these are bit–wise operations (except for not).

Op Code	Format	Use
andn*sz*	Op (I/R/M, R/M)	SSE2, dest Bitwise NOT followed by Bitwise AND into dest.

Logical Instructions (AVX)

Boolean tests of comparison, unless otherwise stated these are bit–wise operations (except for not).

Op Code	Format	Use
andn		BMI1, Bitwise AND of inverted src1 and src2 into dest.

Rotate/Shift Instructions

Moves data within location, often used as a short cut for arithmetic. Use RCX to count number of rotated or shifted places.

Op Code	Format	Use
rcl	Op (R/M)	Rotate with carry to left.
rcr	Op (R/M)	Rotate with carry to right.
rol	Op (R/M)	Rotate to left.
ror	Op (R/M)	Rotate to right.
sal/shl	Op (R/M)	Shift to left, can perform math.
sar	Op (R/M)	Shift to right, can perform math.
shr	Op (R/M)	Shift right.
shld	Op (R/M)	Shift left, double precision math.
shrd	Op (R/M)	Shift right, double precision math.

Rotate/Shift Instructions (SSE)

Moves data within location, often used as a short cut for arithmetic. Use RCX to count number of rotated or shifted places.

Op Code	Format	Use
pslldq		SSE2, Shift pack left.
psrldq		SSE2, Shift pack right.

Rotate/Shift Instructions (AVX)

Moves data within location, often used as a short cut for arithmetic. Use RCX to count number of rotated or shifted places.

Op Code	Format	Use
vpsllv		AVX2, Shift integer left src1 by src2 count into dest.
vpsrav		AVX2, Sign shift integer right src1 by src2 count into dest.
vpsrlv		AVX2, Shift integer right src1 by src2 count into dest.
roxr		BMI2, Rotate src by value into dest, no FLAGs.
sarx		BMI2, Math shift right src1 by src2 count into dest, no FLAGs.
shlx		BMI2, Shift src1 left by src2 into dest.
shrx		BMI2, Shift src1 right by src2 into dest.

Bit Instructions

Moves data within location, often used as a short cut for arithmetic. Use RCX to count number of rotated or shifted places.

Op Code	Format	Use
set*cc*	Op*cc* (R/M)	Set byte to 1 if conditions are met.
bt	Op (R/M)	Copy bit into FLAGS
bts	Op (R/M)	Copy bit to FLAGs and set to '1'.
btr	Op (R/M)	Copy bit to FLAGs and set to '0'.
btc	Op (R/M)	Copy bit to FLAGs and set to '0'.
bsf	Op (R/M, R/M)	Scan source and store in dest the lsb. Affects Zero FLAG.
bsr	Op (R/M, R/M)	Scan source and store in dest the msb. Affects Zero FLAG.

Bit Instructions (SSE)

Moves data within location, often used as a short cut for arithmetic. Use RCX to count number of rotated or shifted places.

Op Code	Format	Use
shuf*sz*	Op (b, R/M, R/M)	SSE2, Moves pack from src to dest according to 8-bit mask
unpckl*sz*	Op (R/M, R/M)	SSE2, Unpack and interweave src low into dest.
unpckh*sz*	Op (R/M, R/M)	SSE2, Unpack and interweave src high into dest.
insertps	Op (I, R/M, R)	SSE4, Insert single into pack according to immediate.
extractps	Op (I, R/M, R)	SSE4, Extraact single from pack according to immediate.
blend*sz*	Op (I, R/M, R/M)	SSE4, Copy float pack from src to dest by immediate.
blendv*sz*	Op (R, R/M, R/M)	SSE4, Copy float pack from src to dest by register.
pshufd		SSE2, Copy src to dest by immediate value.
pshuflw		SSE2, Copy low-src to low-dest by immediate value.
pshufhw		SSE2, Copy high-src to high-dest by immediate value.
punpcklqdq		SSE2, Copy low-src to high-dest.
punpckhqdq		SSE2, Copy high-src to high-dest, high-dest to low dest.
pinsr*sz*		SSE4.1, Copy integer to XMM register by immediate value.
pextr*sz*		SSE4.1, Copy integer from XMM by immediate value.
pblendw		SSE4.1, Copy words from src to dest by immediate.
pblendvb		SSE4.1, Copy bytes from src to dest by XMM0.

Bit Instructions (AVX)

Moves data within location, often used as a short cut for arithmetic. Use RCX to count number of rotated or shifted places.

Op Code	Format	Use
vpblend	Op (src, src, dest)	AVX2, Condition copy value from src to dest by mask.
vperm*sz*		AVX2, Reorders dword by 2nd src mask. d, q, ps, pd sz
vpermil*sz*		AVX, Reorders float by 2nd src, independent lanes.
vperm2f128		AVX2, Reorders 128-bit float of 2 src by mask.
vperm2i128		AVX2, Reorders 128-bit integer of 2 src by mask.
vextracti128		AVX2, Extract int from src to dest by mask.
vinserti128		AVX2, Insert int from src to dest by mask.
bextr		BMI1, Extract bit field from src1 by src2 index+length into dest.
blsi		BMI1, Extract lowest 1 bit into dest, zero other bits.
blsmask		BMI1, Extract bit by mask into dest, set lower bit to 1, clear other bits.
bzhi		BMI2, Copy src1 into dest, clear high-order dest by src2 mask.
lzcnt		LZCNT, Count leading zero in src, result in dest.
pdep		BMI2, Scatters low bit from src1 to dest by src2 mask, other bits are clear.
pext		BMI2, Copy low bit from src1 to dest by src2 mask.
tzcnt		BMI1, Count trailing zero of src, result in dest.
vzeroupper		AVX-64, Zeros upper bits in YMM register.
vzerall		AVX-64, Zeroes all bits in YMM register.

String Instructions

Manipulates text strings or blocks of data. Rep accepts e, ne, z, nz condition codes (or no code at all).

Op Code	Format	Use
cmps*sz*	Op	Compares values pointed by RSI and RDI, sets FLAGs accordingly.
lods*sz*	Op	Load string pointed by RSI into RAX.
movs*sz*	Op	Move string from RSI to RDI.
scas*sz*	Op	Compares string in pointer of RDI to RAX, sets FLAGs.
stos*sz*	Op	Store string pointed by RDI into RAX.
rep*cc*	Op	No *cc*, Repeat String while RCX is not 0. If e or z, repeat while RCX != 0 AND Zero FLAG = 1 If ne or nz, repeat while RCX != 0 AND Zero FLAG = 0

String Instructions (SSE)

Manipulates text strings or blocks of data. Known strings are of known length, unknown strings require EOS (terminator, end-of-string), character.

Op Code	Format	Use
pcmpestri		SSE4.2, Compare two known strings, results in ECX.
pcmpestrm		SSE4.2, Compare two known strings, results in XMM0.
pcmpistri		SSE4.2, Compare two unknown strings, results in ECX.
pcmpistrm		SSE4.2, Compare two unknown strings, results in XMM0.

String Instructions (AVX)

Manipulates text strings or blocks of data.

Flag Instructions

Manipulate the flag register. MSB > SF, ZF, 0, AF, 0, PF, 1, CF > LSB

Op Code	Format	Use
clc	Op	CLEAR carry FLAG
stc	Op	SET carry FLAG
cmc	Op	TOGGLE carry FLAG
std	Op	SET direction FLAG
cld	Op	CLEAR direction FLAG
lahf	Op	Load AH with FLAG register.
sahf	Op	Store AH into FLAG register.
pushfd	Op	Push EFLAG register onto stack.
pushfq		Push RFLAG register onto stack.
popfd	Op	Pop stack into EFLAG register.
popfq		Pop stack into RFLAG register.

Flag Instructions (SSE)

Manipulate the MXCSR flag register.

Op Code	Format	Use
ldmxscr		SSE, Load MXSCR from memory.
stmxcsr		SSE, Store MXCSR into memory.
fxsave		SSE, Store FPU, MMX, XMM and MXSCR to memory.
fxrstor		SSE, Load FPU, MMX, XMM and MXSCR from memory.

Flag Instructions (AVX)

Manipulate the MXCSR flag register.

Op Code	Format	Use

Control Instructions

Program flow control; Low-level, does not include conditional statements found in HLL programming languages.

Op Code	Format	Use
jmp	Op (I/R/M)	Jump to location specified by value.
j*cc*	Op (I/R/M)	Jump if condition is true.
call	Op (R/M) Op func	Push RIP onto stack and jump to address. Push RIP onto stack and jump to function (address label).
ret	Op	Pops call off stack and jump to previous address.
enter	Op (R/M)	Creates stack frame and allocates memory for parameters.
leave	Op	Destroys frame and deallocates memory.
jecxz	Op (R/M)	Jump to location if ECX = 0
jrcxz	Op (R/M)	Jump to location if RCX = 0
loop*cc*	Op (I/R/M)	No *cc*, Decrease ECX and jump to location if ECX = 0 If e or z, also test ZF = 1 If ne or nz, also test if ZF = 0

Control Instructions (SSE)

Program flow control; Low-level, does not include conditional statements found in HLL programming languages.

Op Code	Format	Use
movnti		SSE2, Copy register to memory by non-temporal hint.
movntdq		SSE2, Copy XMM register to memory by non-temporal hint.
maskmovdqu		SSE2, Condition Copy bytes from XMM to memory by hint.
movntdqa		SSE4.1, Load memory dword into XMM by NT hint.
sfence		SSE, Serialize memory store operations.
lfence		SSE2, Serialize memory load operations.
mfence		SSE2, Serialize memory store and load operations.
prefetchH		SSE, Provide hint to load memory into cache, H is hint.
clflush		SSE2, Flush cache line of memory, src is memory address.

Control Instructions (AVX)

Program flow control; Low-level, does not include conditional statements found in HLL programming languages.

Other Instructions

Instructions with no specific use or category, general actions for programs.

Op Code	Format	Use
lea	Op (R/M, R)	Calculate effective address and store in register.
nop	Op	Do nothing, increments RIP to next instruction.
cpuid	Op	Obtain CPU informatino and supported features.

Other Instructions (SSE)

Instructions with no specific use or category, general actions for programs.

Op Code	Format	Use
crc32		SSE4.2, Accerlate 32-bit CRC algorithms.
popcnt	Op (R/M, R)	SSE4.2, Counts bits that are 1 in src, result in dest.

Other Instructions (AVX)

Instructions with no specific use or category, general actions for programs.

Op Code	Format	Use

Linux Syscalls