A fundamental introduction to x86 assembly programming

0. Introduction

The x86 instruction set architecture is at the heart of CPUs that power our home computers and remote servers for over two decades. Being able to read and write code in low-level assembly language is a powerful skill to have. It enables you to write faster code, use machine features unavailable in C, and reverse-engineer compiled code.

But starting out can be a daunting task. The official documentation manuals from Intel are well over a thousand pages long. Twenty years of continual evolution with backward compatibility have produced a landscape with clashing design principles from different eras, deprecated features occupying space, layers upon layers of mode switches, and an exception to every pattern.

In this tutorial, I will help you gain a solid understanding of the x86 ISA from basic principles. I will focus more on *building a clear mental model* of what's happening, rather than giving every detail precisely (which would be long and boring to read). If you want to make use of this knowledge, you should simultaneously refer to another tutorial that shows you how to write and compile a simple function, and also have a list of CPU instructions open for referencing. My tutorial will start out in familiar territory and slowly add complexity in manageable steps – unlike other documentation that tend to lay out the information all at once.

The prerequisites to reading this tutorial are working with binary numbers, moderate experience programming in an imperative language (C/C++/Java/Python/etc.), and the concept of memory pointers (C/C++). You do not need to know how CPUs work internally or have prior exposure to assembly language.

Contents

1. Tools and testing

When reading this tutorial, it's helpful to write and test your own assembly language programs. This is most easily done on Linux (harder but possible on Windows). Here is a sample function in assembly language:

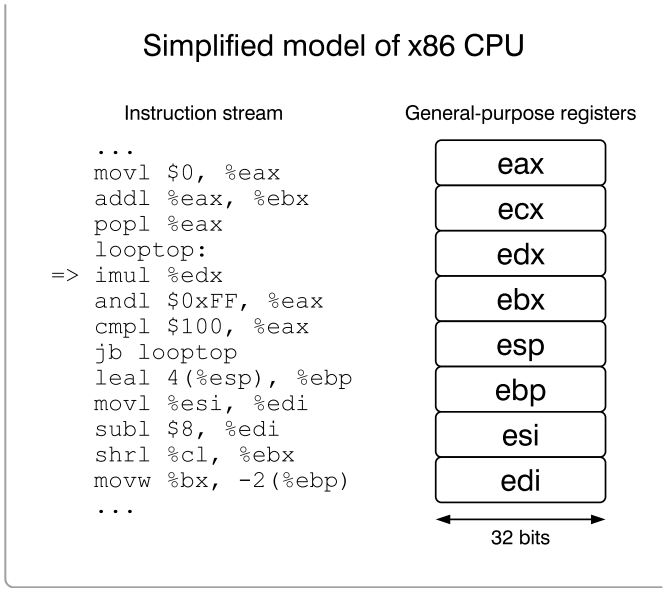
```
.globl myfunc
myfunc:
    retl
```

Save it in a file called my-asm.s, and compile it with the command: `gcc -m32 -c -o my-asm.o my-asm.s`. There is no way to run the code at the moment, because it would require either interfacing with a C program or writing boilerplate code for interacting with the OS to handle the starting/printing/stopping/etc. At the very least, being able to compile code gives you a way to verify that your assembly programs are syntactically correct.

Note that my tutorial uses the AT&T assembly language syntax instead of Intel syntax. The underlying concepts are still the same in both cases, but the notation is a bit different. It's possible to mechanically translate from one syntax to the other, so there is no need for much concern.

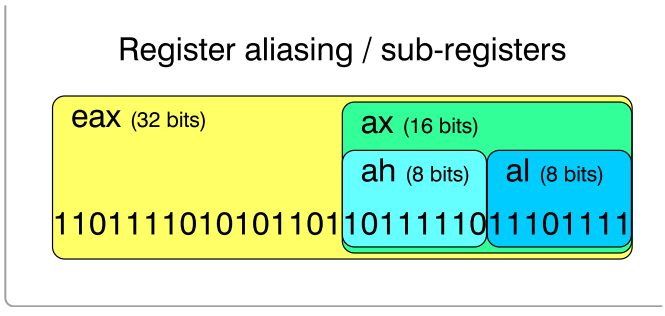
2. Basic execution environment

An x86 CPU has eight 32-bit general-purpose registers. For historical reasons, the registers are named {`eax`, `ecx`, `edx`, `ebx`, `esp`, `ebp`, `esi`, `edi`}. (Other CPU architectures would simply name them `r0`, `r1`, ..., `r7`.) Each register can hold any 32-bit integer value. The x86 architecture actually has over a hundred registers, but we will only cover specific ones when needed.



As a first approximation, a CPU executes a list of instructions sequentially, one by one, in the order listed in the source code. Later on, we will see how the code path can go non-linearly, covering concepts like if-then, loops, and function calls.

There are actually eight 16-bit and eight 8-bit registers that are subparts of the eight 32-bit general-purpose registers. These features come from the 16-bit era of x86 CPUs, but still have some occasional use in 32-bit mode. The 16-bit registers are named {`ax`, `cx`, `dx`, `bx`, `sp`, `bp`, `si`, `di`} and represent the bottom 16 bits of the corresponding 32-bit registers {`eax`, `ecx`, ..., `edi`} (the prefix "e" stands for "extended"). The 8-bit registers are named {`al`, `cl`, `dl`, `bl`, `ah`, `ch`, `dh`, `bh`} and represent the low and high 8 bits of the registers {`ax`, `cx`, `dx`, `bx`}. Whenever the value of a 16-bit or 8-bit register is modified, the upper bits belonging to the full 32-bit register will remain unchanged.



3. Basic arithmetic instructions

The most basic x86 arithmetic instructions operate on two 32-bit registers. The first operand acts as a source, and the second operand acts as both a source and destination. For example: `addl %ecx, %eax` – in C notation, this means: `eax = eax + ecx;`, where `eax` and `ecx` have the type `uint32_t`. Many instructions fit this important schema – for example:

- `xorl %esi, %ebp` means `ebp = ebp ^ esi;`.
- `subl %edx, %ebx` means `ebx = ebx - edx;`.
- `andl %esp, %eax` means `eax = eax & esp;`.

A few arithmetic instructions take only one register as an argument, for example:

- `notl %eax` means `eax = ~eax;`.
- `incl %ecx` means `ecx = ecx + 1;`.

The bit shifting and rotation instructions take a 32-bit register for the value to be shifted, and the fixed 8-bit register `cl` for the shift count. For example: `shll %cl, %ebx` means `ebx = ebx << cl;`.

Many arithmetic instructions can take an immediate value as the first operand. The immediate value is fixed (not variable), and is coded into the instruction itself. Immediate values are prefixed with `$`. For example:

- `movl $0xFF, %esi` means `esi = 0xFF;`.
- `addl $-2, %edi` means `edi = edi + (-2);`.
- `shrl $3, %edx` means `edx = edx >> 3;`.

Note that the `movl` instruction copies the value from the first argument to the second argument (it does not strictly “move”, but this is the customary name). In the case of registers, like `movl %eax, %ebx`, this means to copy the value of the `eax` register into `ebx` (which overwrites `ebx`’s previous value).

An aside

Now is a good time to talk about one principle in assembly programming: Not every desirable operation is directly expressible in one instruction. In typical programming languages that most people use, many constructs are composable and adaptable to different situations, and arithmetic can be nested. In assembly language however, you can only write what the instruction set allows. To illustrate with examples:

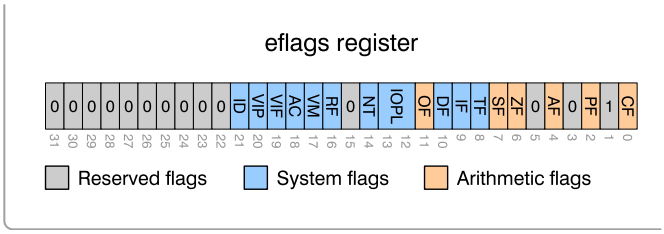
- You can’t add two immediate constants together, even though you can in C. In assembly you’d either compute the value at compile time, or express it as a sequence of instructions.
- You can add two 32-bit registers in one instruction, but you can’t add three 32-bit registers – you’d need to break it up into two instructions.
- You can’t add a 16-bit register to a 32-bit register. You’d need to write one instruction to perform a 16-to-32-bit widening conversation, and another instruction to perform

the addition.

- When performing bit shifting, the shift count must be either a hard-coded immediate value or the register `cl`. It cannot be any other register. If the shift count was in another register, then the value needs to be copied to `cl` first.

The takeaway messages are that you shouldn't try to guess or invent syntaxes that don't exist (such as `addl %eax, %ebx, %ecx`); also that if you can't find a desired instruction in the long list of supported instructions then you need to manually implement it as a sequence of instructions (and possibly allocate some temporary registers to store intermediate values).

4. Flags register and comparisons



There is a 32-bit register named `eflags` which is *implicitly* read or written in many instructions. In other words, its value plays a role in the instruction execution, but the register is not mentioned in the assembly code.

Arithmetic instructions such as `addl` usually update `eflags` based on the computed result. The instruction would set or clear flags like carry (CF), overflow (OF), sign (SF), parity (PF), zero (ZF), etc. Some instructions read the flags – for example `adcl` adds two numbers and uses the carry flag as a third operand: `adcl %ebx, %eax` means `eax = eax + ebx + cf;`. Some instructions set a register based on a flag – for example `setz %al` sets the 8-bit register `al` to 0 if ZF is clear or 1 if ZF is set. Some instructions directly affect a single flag bit, such as `cld` clearing the direction flag (DF).

Comparison instructions affect `eflags` without changing any general-purpose registers. For example, `cmpl %eax, %ebx` will compare the two registers' value by subtracting them in an unnamed temporary place and set the flags according to the result, so that you can tell whether `eax < ebx` or `eax == ebx` or `eax > ebx` in either unsigned mode or signed mode. Similarly, `testl %eax, %ebx` computes `eax & ebx` in a temporary place and sets the flags accordingly. Most of the time, the instruction after a comparison is a conditional jump (covered later).

So far, we know that some flag bits are related to arithmetic operations. Other flag bits are concerned with how the CPU behaves – such as whether to accept hardware interrupts, virtual 8086 mode, and other system management stuff that is mostly of concern to OS developers, not to application developers. For the most part, the `eflags` register is largely ignorable. The system flags are definitely ignorable, and the arithmetic flags can be forgotten except for comparisons and bigint arithmetic operations.

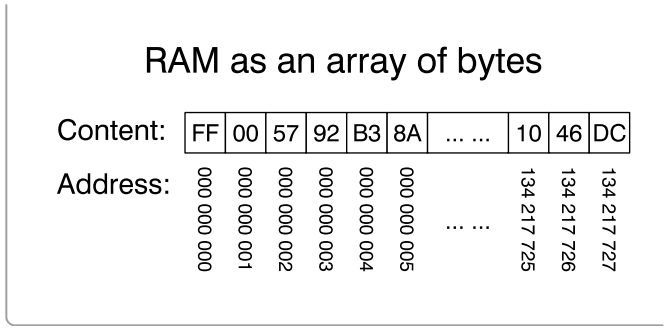
5. Memory addressing, reading, writing

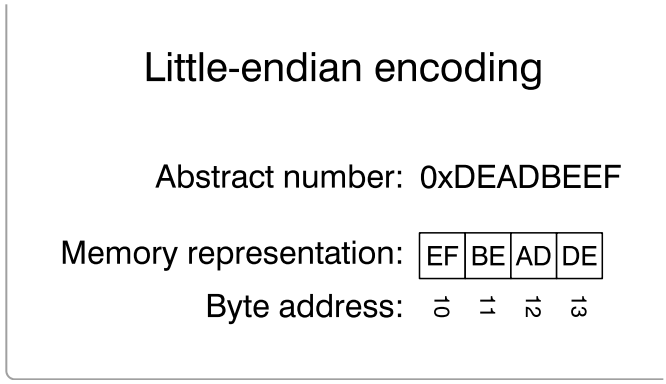
The CPU by itself does not make a very useful computer. Having only 8 data registers severely limits what computations you can do because you can't store much information. To augment the CPU, we have RAM as a large system memory. Basically,

RAM is an enormous array of bytes – for example, 128 MiB of RAM is 134 217 728 bytes that you can store any values to.

When storing a value longer than a byte, the value is encoded in little endian. For example if a 32-bit register contained the value 0xDEADBEEF and this register needs to be stored in memory starting at address 10, then the byte value 0xEF goes into RAM address 10, 0xBE into address 11, 0xAD into address 12, and finally 0xDE into address 13. When reading values from memory, the same rule applies – the bytes at lower memory addresses get loaded into the lower parts of a register.

It should go without saying that the CPU has instructions to read and write memory. Specifically, you can load or store one or more bytes at any memory address you choose. The simplest thing you can do with memory is to read or write a single byte:





- `movb (%ecx), %al` means `al = *ecx;`. (this reads the byte at memory address `ecx` into the 8-bit `al` register)
- `movb %bl, (%edx)` means `*edx = bl;`. (this writes the byte in `bl` to the byte at memory address `edx`)
- (In the illustrative C code, `al` and `bl` have the type `uint8_t`, and `ecx` and `edx` are being casted from `uint32_t` to `uint8_t*`.)

Next, many arithmetic instructions can take one memory operand (never two). For example:

- `addl (%ecx), %eax` means `eax = eax + (*ecx);`. (this reads 32 bits from memory)
- `addl %ebx, (%edx)` means `*edx = (*edx) + ebx;`. (this reads and writes 32 bits in memory)

Addressing modes

When we write code that has loops, often one register holds the base address of an array and another register holds the current index being processed. Although it's possible to manually compute the address of the element being processed, the x86 ISA provides a more elegant solution – there are memory addressing modes that let you add and multiply certain registers together. This is probably easier to illustrate than describe:

- `movb (%eax,%ecx), %bh` means `bh = *(eax + ecx);`.
- `movb -10(%eax,%ecx,4), %bh` means `bh = *(eax + (ecx * 4) - 10);`.

The address format is *offset*(*base*, *index*, *scale*), where *offset* is an integer constant (can be positive, negative, or zero), *base* and *index* are 32-bit registers (but a few combinations are disallowed), and *scale* is either {1,2,4,8}. For example if an array holds a series of 64-bit integers, we would use *scale* = 8 because each element is 8 bytes long.

The memory addressing modes are valid wherever a memory operand is permitted. Thus if you can write `sbbl %eax, (%eax)`, then you can certainly write `sbbl %eax, (%eax,%eax,2)` if you need the indexing capability. Also note that the address being computed is a temporary value that is not saved in any register. This is good because if you wanted to compute the address explicitly, you would need to allocate a register for it, and having only 8 GPRs is rather tight when you want to store other variables.

There is one special instruction that uses memory addressing but does not actually access memory. The `leal` (load effective address) instruction computes the final memory address according to the addressing mode, and stores the result in a register. For example, `leal 5(%eax,%ebx,8), %ecx` means `ecx = eax + ebx*8 + 5;`. Note that this is entirely an arithmetic operation and does not involve dereferencing a memory address.

6. Jumps, labels, and machine code

Each assembly language instruction can be prefixed by zero or more labels. These labels will be useful when we need to jump to a certain instruction. Examples:

```
foo:  /* A label */
negl %eax  /* Has one label */

addl %eax, %eax  /* Zero labels */

bar: qux: sbbl %eax, %eax  /* Two labels */
```

The `jmp` instruction tells the CPU to go to a labelled instruction as the next instruction to execute, instead going to the next instruction below by default. Here is a simple infinite loop:

```
top: incl %ecx
jmp top
```

Although `jmp` is unconditional, it has sibling instructions that look at the state of `eflags`, and either jumps to the label if the condition is met or otherwise advances to the next instruction below. Conditional jump instructions include: `ja` (jump if above), `jle` (jump if less than or equal), `jo` (jump if overflow), `jnz` (jump if non-zero), et cetera. There are 16 of them in all, and some have synonyms – e.g. `jz` (jump if zero) is the same as `je` (jump if equal), `ja` (jump if above) is the same as `jnbe` (jump if not below or equal). An example of using conditional jump:

```
jc skip  /* If carry flag is on, then jump away */
/* Otherwise CF is off, then execute this stuff */
notl %eax
/* Implicitly fall into the next instruction */
```

```
skip:
adcl %eax, %eax
```

Label addresses are fixed in the code when it is compiled, but it is also possible to jump to an arbitrary memory address computed at run time. In particular, it is possible to jump to the value of a register: `jmp *%ecx` essentially means to copy `ecx`'s value into `eip`, the instruction pointer register.

Now is a perfect time to discuss a concept that was glossed over in section 1 about instructions and execution. Each instruction in assembly language is ultimately translated into 1 to 15 bytes of machine code, and these machine instructions are strung together to create an executable file. The CPU has a 32-bit register named `eip` (extended instruction pointer) which, during program execution, holds the memory address of the current instruction being executed. Note that there are very few ways to read or write the `eip` register, hence why it behaves very differently from the 8 main general-purpose registers. Whenever an instruction is executed, the CPU knows how many bytes long it was, and advances `eip` by that amount so that it points to the next instruction.

Assembly vs. machine code

Machine code bytes | Assembly language statements

```
                   foo:
B8 22 11 00 FF     movl $0xFF001122, %eax
01 CA              addl %ecx, %edx
31 F6              xorl %esi, %esi
53                 pushl %ebx
8B 5C 24 04        movl 4(%esp), %ebx
8D 34 48           leal (%eax,%ecx,2), %esi
39 C3              cmpl %eax, %ebx
72 EB              jnae foo
C3                 retl
```

Instruction stream

```
B8 22 11 00 FF 01 CA 31 F6 53 8B 5C 24
04 8D 34 48 39 C3 72 EB C3
```

While we're on this note about machine code, assembly language is not actually the lowest level that a programmer can go; raw binary machine code is the lowest level. (Intel insiders have access to even lower levels, such as pipeline debugging and microcode – but ordinary programmers can't go there.) Writing machine code by hand is very unpleasant (I mean, assembly language is unpleasant enough already), but there are a couple of minor capabilities gained. By writing machine code, you can encode some instructions in alternate ways (e.g. a longer byte sequence that has the same effect when executed), and you can deliberately generate invalid instructions to test the CPU's behavior (not every CPU handles errors the same way).

7. The stack

The stack is conceptually a region of memory addressed by the `esp` register. The x86 ISA has a number of instructions for manipulating the stack. Although all of this functionality can be achieved with `movl`, `addl`, etc. and with registers other than `esp`, using the stack instructions is more idiomatic and concise.

In x86, the stack grows downward, from larger memory addresses toward smaller ones. For example, "pushing" a 32-bit value onto the stack means to first decrement `esp` by 4, then take the 4-byte value and store it starting at address `esp`. "Popping" a value performs the reverse operations – load 4 bytes starting at address `esp` (either into a given register or discarded), then increment `esp` by 4.

The stack is important for function calls. The `call` instruction is like `jmp`, except that before jumping it first pushes the next instruction address onto the stack. This way, it's possible to go back by executing the `retl` instruction, which pops the address into `eip`. Also, the standard C calling convention puts some or all the function arguments on the stack.

Note that stack memory can be used to read/write the `eflags` register, and to read the `eip` register. Accessing these two registers is awkward because they cannot be used in typical `movl` or arithmetic instructions.

8. Calling convention

When we compile C code, it is translated into assembly code and ultimately machine code. A calling convention defines how C functions receive arguments and return a value, by putting values on the stack and/or in registers. The calling convention applies to a C function calling another C function, a piece of assembly code calling a C function, or a C function calling an assembly function. (It does not apply to a piece of assembly code calling an arbitrary piece of assembly code; there are no restrictions in this case.)

On 32-bit x86 on Linux, the calling convention is named cdecl. The function caller (parent) pushes the arguments from right to left onto the stack, calls the target function (callee/child), receives the return value in `eax`, and pops the arguments. For example:

```
int main(int argc, char **argv) {
  print("Hello", argc);
  /*
  The above call to print() would translate
  into assembly code like this:

  pushl %registerContainingArgc
  pushl $ADDRESS_OF_HELLO_STRING_CONSTANT
  call print
  // Receive result in %eax
  popl %ecx  // Discard argument str
  popl %ecx  // Discard argument foo
  */
}

int print(const char *str, int foo) {
  ....
  /*
  In assembly language, these 32-bit values exist on the stack:
    0(%esp) has the address of the caller's next instruction.
    4(%esp) has the value of the argument str (char pointer).
    8(%esp) has the value of the argument foo (signed integer).
  Before the function executes retl, it needs to
  put some number into %eax as the return value.
  */
}
```

9. Repeatable string instructions

The class of "string" instructions use the `esi` and `edi` registers as memory addresses, and automatically increment/decrement them after the instruction. For example, `movsb %esi, %edi` means `*edi = *esi; esi++; edi++;` (copies one byte). (Actually, `esi` and `edi` increment if the direction flag is 0; otherwise they

decrement if DF is 1.) Examples of other string instructions include `cmpsb`, `scasb`, `stosb`.

A string instruction can be modified with the `rep` prefix (see also `repe` and `repne`) so that it gets executed `ecx` times (with `ecx` decrementing automatically). For example, `rep movsb %esi, %edi` means:

```
while (ecx > 0) {
  *edi = *esi;
  esi++;
  edi++;
  ecx--;
}
```

These string instructions and the `rep` prefixes bring some iterated compound operations into assembly language. They represent some of the mindset of the CISC design, where it is normal for programmers to code directly in assembly, so it provides higher level features to make the work easier. (But the modern solution is to write in C or an even higher level language, and rely on a compiler to generate the tedious assembly code.)

10. Floating-point and SIMD

The x87 math coprocessor has eight 80-bit floating-point registers (but all x87 functionality has been incorporated into the main x86 CPU now), and the x86 CPU also has eight 128-bit `xmm` registers for SSE instructions. I do not have much experience with FP/x87, and you should refer to other guides available on the web. The way the x87 FP stack works is a bit weird, and these days it's better to do floating-point arithmetic using `xmm` registers and SSE/SSE2 scalar instructions instead.

As for SSE, a 128-bit `xmm` register can be interpreted in many ways depending on the instruction being executed: as sixteen byte values, as eight 16-bit words, as four 32-bit doublewords or single-precision floating-point numbers, or as two 64-bit quadwords or double-precision floating-point numbers. For example, one SSE instruction would copy 16 bytes (128 bits) from memory into an `xmm` register, and one SSE instruction would add two `xmm` registers together treating each one as eight 16-bit words in parallel. The idea behind SIMD is to execute one instruction to operate on many data values at once, which is faster than operating on each value individually because fetching and executing every instruction incurs a certain amount of overhead.

It should go without saying that all SSE/SIMD operations can be emulated more slowly using basic scalar operations (e.g. the 32-bit arithmetic covered in section 3). A cautious programmer might choose to prototype a program using scalar operations, verify its correctness, and gradually convert it to use the faster SSE instructions while ensuring it still computes the same results.

11. Virtual memory

Up to now, we assumed that when an instruction requests to read from or write to a memory address, it will be exactly the address handled by the RAM. But if we

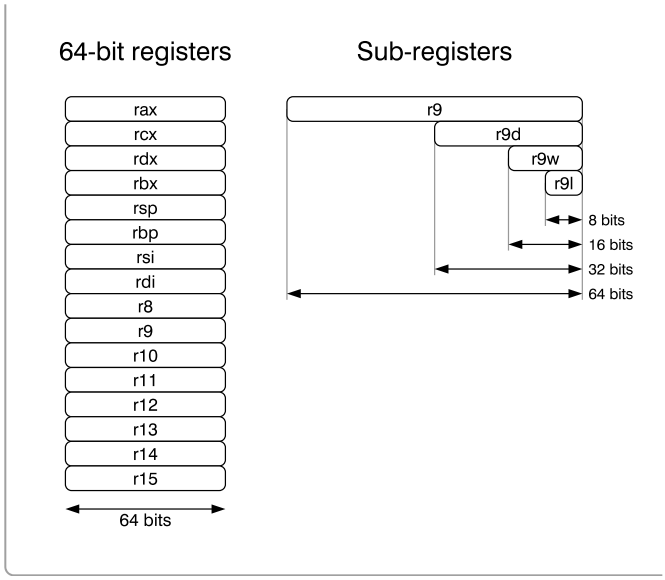
introduce a translation layer in between, we can do some interesting things. This concept is known as virtual memory, paging, and other names.

The basic idea is that there is a page table, which describes what each page (block) of 4096 bytes of the 32-bit virtual address space is mapped to. For example, if a page is mapped to nothing then trying to read/write a memory address in that page will trigger a trap/interrupt/exception. Or for example, the same virtual address 0x08000000 could be mapped to a different page of physical RAM in each application process that is running. Also, each process could have its own unique set of pages, and never see the contents of other processes or the operating system kernel. The concept of paging is mostly of concern to OS writers, but its behavior sometimes affects the application programmer so they should be aware of its existence.

Note that the address mapping need not be 32 bits to 32 bits. For example, 32 bits of virtual address space can be mapped onto 36 bits of physical memory space (PAE). Or a 64 bit virtual address space can be mapped onto 32 bits of physical memory space on a computer with only 1 GiB of RAM.

12. 64-bit mode

Here I will only talk a little about the x86-64 mode and give a sketch of what has changed. Elsewhere on the web there are plenty of articles and reference materials to explain all the differences in detail.



Obviously, the 8 general-purpose registers have been extended to 64 bits long. The new registers are named {`rax`, `rcx`, `rdx`, `rbx`, `rsp`, `rbp`, `rsi`, `rdi`}, and the old 32-bit registers {`eax`, ..., `edi`} occupy the low 32 bits of the aforementioned 64-bit registers. There are also 8 new 64-bit registers {`r8`, `r9`, `r10`, `r11`, `r12`, `r13`, `r14`, `r15`}, bringing the total to 16 GPRs now. This largely alleviates the register pressure when working with many variables. The new registers have subregisters too – for example the 64-bit register `r9` contains the 32-bit `r9d`, the 16-bit `r9w`, and the 8-bit `r9l`. Also, the low byte of {`rsp`, `rbp`, `rsi`, `rdi`} are addressable now as {`spl`, `bpl`, `sil`, `dil`}.

Arithmetic instructions can operate on 8-, 16-, 32-, or 64-bit registers. When operating on 32-bit registers, the upper 32 bits are cleared to zero – but narrower operand widths will leave all the high bits unchanged. Many niche instructions are removed from the 64-bit instruction set, such as BCD-related ones, most instructions involving 16-bit segment registers, and pushing/poping 32-bit values on the stack.

For the application programmer, there isn't much to say about x86-64 programming versus the old x86-32. Generally speaking, the experience is better because there are more registers to work with, and a few minor unnecessary features have been removed. All memory pointers must be 64 bits (this takes some time to get accustomed to), whereas data values can be 32 bits, 64 bits, 8 bits, etc. depending on the situation (you are not forced to use 64 bits for data). The revised calling convention makes it much easier to retrieve function arguments in assembly code, because the first 6 or so

arguments are placed in registers instead of on the stack. Other than these points, the experience is quite similar. (Though for systems programmers, x86-64 introduces new modes, new features, new problems to worry about, and new cases to handle.)

13. Compared to other architectures

RISC CPU architectures do a couple of things differently from x86. Only explicit load/store instructions touch memory; ordinary arithmetic ones do not. Instructions have a fixed length such as 2 or 4 bytes each. Memory operations usually need to be aligned, e.g. loading a 4-byte word must have a memory address that is a multiple of 4. In comparison, the x86 ISA has memory operations embedded in arithmetic instructions, encodes instructions as variable-length byte sequences, and almost always allows unaligned memory accesses. Additionally, while x86 features a full suite of 8-, 16-, and 32-bit arithmetic operations due to its backward-compatible design, RISC architectures are usually purely 32-bit. For them to operate on narrower values, they load a byte or word from memory and extend the value into a full 32-bit register, do the arithmetic operations in 32 bits, and finally store the low 8 or 16 bits to memory. Popular RISC ISAs include ARM, MIPS, and RISC-V.

VLIW architectures allow you to explicitly execute multiple sub-instructions in parallel; for example you might write `add a, b; sub c, d` on one line because the CPU has two independent arithmetic units that work at the same time. x86 CPUs can execute multiple instructions at once too (known as superscalar processing), but instructions are not explicitly coded this way – the CPU internally analyzes the parallelism in the instruction stream and dispatches acceptable instructions to multiple execution units.

14. Summary

We began the discussion of the x86 CPU by treating it as a simple machine that has a couple of registers and follows a list of instructions sequentially. We covered the basic arithmetic operations that can be performed on these registers. Then we learned about jumping to different places in the code, comparisons, and conditional jumps. Next we visited the concept of RAM as a huge addressable data storage, and how the x86 addressing modes can be used to compute addresses concisely. Finally we looked briefly at the stack, calling convention, advanced instructions, virtual memory address translation, and differences in the x86-64 mode.

I hope this tutorial was sufficient to get you oriented with how the x86 instruction set architecture generally works. There are countless details I could not cover in this introductory-level article – such as a full walkthrough on writing a basic function, debugging common mistakes, using SSE/AVX effectively, working with segmentation, covering system data structures like page tables and interrupt descriptors, discussing privileges and security, and so much more. But with a solid mental model of how an x86 CPU operates, you are now in a good position to seek more advanced tutorial texts, try writing some code with an awareness of what happens and why it works that way, and maybe even try scanning through Intel's extremely detailed thousand-page CPU manuals.

15. More info

- University of Virginia CS216: x86 Assembly Guide
- Wikipedia: x86 instruction listings
- Intel® 64 and IA-32 Architectures Software Developer Manuals

Categories: Programming, Writing, x86 assembly Last updated: 2016-01-07

Feedback: Question/comment? Contact me  ProjectNayuki: Like, comment, follow updates on Facebook