7.3. Arithmetic Instructions

The x86 ISA implements several instructions that correspond to arithmetic operations performed by the ALU. Table 1 lists several arithmetic instructions that one may encounter when reading assembly.

Table 1. Common Arithmetic Instructions.

Instruction	Translation
`add S, D`	S + D → D
`sub S, D`	D - S → D
`inc D`	D + 1 → D
`dec D`	D - 1 → D
`neg D`	-D → D
`imul S, D`	S × D → D
`idiv S`	`%rax` / S: quotient → `%rax`, remainder → `%rdx`

The `add` and `sub` instructions correspond to addition and subtraction and take two operands each. The next three entries show the single-register instructions for the increment (`x++`), decrement (`x--`), and negation (`-x`) operations in C. The multiplication instruction operates on two operands and places the product in the destination. If the product requires more than 64 bits to represent, the value is truncated to 64 bits.

The division instruction works a little differently. Prior to the execution of the `idiv` instruction, it is assumed that register `%rax` contains the dividend. Calling `idiv` on operand S divides the contents of `%rax` by S and places the quotient in register `%rax` and the remainder in register `%rdx` .

7.3.1. Bit Shifting Instructions

Bit shifting instructions enable the compiler to perform bit shifting operations. Multiplication and division instructions typically take a long time to execute. Bit shifting offers the compiler a shortcut for multiplicands and divisors that are powers of 2. For example, to compute `77 * 4` , most compilers will translate this operation to `77 << 2` to avoid the use of an `imul` instruction. Likewise, to compute `77 / 4` , a compiler typically translates this operation to `77 >> 2` to avoid using the `idiv` instruction.

Keep in mind that left and right bit shift translate to different instructions based on whether the goal is an arithmetic (signed) or logical (unsigned) shift.

Table 2. Bit Shift Instructions

Instruction	Translation	Arithmetic or Logical?
`sal v, D`	`D << v` → D	arithmetic
`shl v, D`	`D << v` → D	logical
`sar v, D`	`D >> v` → D	arithmetic
`shr v, D`	`D >> v` → D	logical

Each shift instruction takes two operands, one which is usually a register (denoted by D) and the other which is a shift value (*v*). On 64-bit systems, the shift value is encoded as a single byte (since it doesn't make sense to shift past 63). The shift value *v* must either be a constant or stored in register `%cl` .

NOTE

Different Versions of Instructions Help Distinguish Types at an Assembly Level

At the assembly level, there is no notion of types. However, recall that the compiler will use component registers based on types. Similarly, recall that shift right works differently depending on whether or not the value is signed or unsigned. At the assembly level, the compiler uses separate instructions to distinguish between logical and arithmetic shifts!

7.3.2. Bitwise Instructions

Bitwise instructions enable the compiler to perform bitwise operations on data. One way the compiler uses bitwise operations is for certain optimizations. For example, a compiler may choose to implement 77 mod 4 with the operation `77 & 3` in lieu of the more expensive `idiv` instruction.

Table 3 lists common bitwise instructions.

Table 3. Bitwise Operations

Instruction	Translation
`and S, D`	`S & D` → D
`or S, D`	`S | D` → D
`xor S, D`	`S ^ D` → D

Instruction	Translation
`not D`	~D → D

Remember that bitwise `not` is distinct from negation (`neg`). The `not` instruction flips the bits but does not add 1. Be careful not to confuse these two instructions.

WARNING

Use bitwise operations only when needed in your C code!

After reading this section, it may be tempting to replace common arithmetic operations in your C code with bitwise shifts and other operations. This is *not* recommended. Most modern compilers are smart enough to replace simple arithmetic operations with bitwise operations when it makes sense, making it unnecessary for the programmer to do so. As a general rule, programmers should prioritize code readability whenever possible and avoid premature optimization.

7.3.3. The Load Effective Address Instruction

What's lea got to do (got to do) with it?

What's lea, but an effective address loading?

~With apologies to Tina Turner

We finally come to the **load effective address** or `lea` instruction, which is probably the arithmetic instruction that causes students the most consternation. It is traditionally used as a fast way to compute the address of a location in memory. The `lea` instruction operates on the same operand structure that we've seen thus far but does *not* include a memory lookup. Regardless of the type of data contained in the operand (whether it be a constant value or an address), `lea` simply performs arithmetic.

For example, suppose register `%rax` contains the constant value 0x5, register `%rdx` contains the constant value 0x4, and register `%rcx` contains the value 0x808 (which happens to be an address). Table 4 depicts some example `lea` operations, their translations, and corresponding values.

Table 4. Example lea Operations

Instruction	Translation	Value
`lea 8(%rax), %rax`	8 + `%rax` → `%rax`	13 → `%rax`
`lea (%rax, %rdx), %rax`	`%rax` + `%rdx` → `%rax`	9 → `%rax`
`lea (,%rax,4), %rax`	`%rax` × 4 → `%rax`	20 → `%rax`

Instruction	Translation	Value
`lea -0x8(%rcx), %rax`	`%rcx - 8 → %rax`	0x800 → `%rax`
`lea -0x4(%rcx, %rdx, 2), %rax`	`%rcx + %rdx × 2 - 4 → %rax`	0x80c → `%rax`

In all cases, the `lea` instruction performs arithmetic on the operand specified by the source S and places the result in the destination operand D. The `mov` instruction is identical to the `lea` instruction *except* that the `mov` instruction is *required* to treat the contents in the source operand as a memory location if it is in a memory form. In contrast, `lea` performs the same (sometimes complicated) operand arithmetic *without* the memory lookup, enabling the compiler to cleverly use `lea` as a substitution for some types of arithmetic.

Contents

7.3. Arithmetic Instructions

7.3.1. Bit Shifting Instructions

7.3.2. Bitwise Instructions

7.3.3. The Load Effective Address Instruction