# Module III
# Machine Language

**Dr. Arijit Roy**
**Computer Science and Engineering Group**
**Indian Institute of Information Technology Sri City**

The concepts are based on Carnegie Mellon University course and the text-book, authored by Patterson and Hennesy

## Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations

# Address Computation Instruction

- leaq Src, Dst
  - Src is address mode expression
  - Set Dst (must be a register) to address (**Effective address**) denoted by expression

Only leaq has no other variant, rather we can consider it as a variant of movq instruction

```
leaq S,D  //D← &S; Load effective address
```

- Uses
  - Computing addresses without a memory reference
    - E.g., translation of p = &x[i];
  - Computing arithmetic expressions of the form x + k*y
    - k = 1, 2, 4, or 8

Operations are divided in to four groups – load effective address, unary, binary, and shift

Compilers often find clever uses of leaq that have nothing to do with effective address computations. The destination operand must be a register.

- Example

```
long m12(long x)
{
  return x*12;
}
```

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax # t <- x+x*2
salq $2, %rax            # return t<<2
```

# Some Arithmetic Operations

- One Operand Instructions

| | | |
|---|---|---|
| incq | Dest | Dest = Dest + 1 |
| decq | Dest | Dest = Dest – 1 |
| negq | Dest | Dest = – Dest |
| notq | Dest | Dest = ~Dest |

Unary

- See book for more instructions

# Some Arithmetic Operations

- Two Operand Instructions:

| Format | | Computation | |
|---|---|---|---|
| addq | Src,Dest | Dest = Dest + Src | Add |
| subq | Src,Dest | **Dest = Dest – Src** | Subtract |
| imulq | Src,Dest | Dest = Dest * Src | Multiply |
| salq | Src,Dest | Dest = Dest << Src | Left shift |
| sarq | Src,Dest | Dest = Dest >> Src | Arithmetic right shift |
| shrq | Src,Dest | Dest = Dest >> Src | Logical right shift |
| xorq | Src,Dest | Dest = Dest ^ Src | Exclusive-or |
| andq | Src,Dest | Dest = Dest & Src | And |
| orq | Src,Dest | Dest = Dest \| Src | OR |

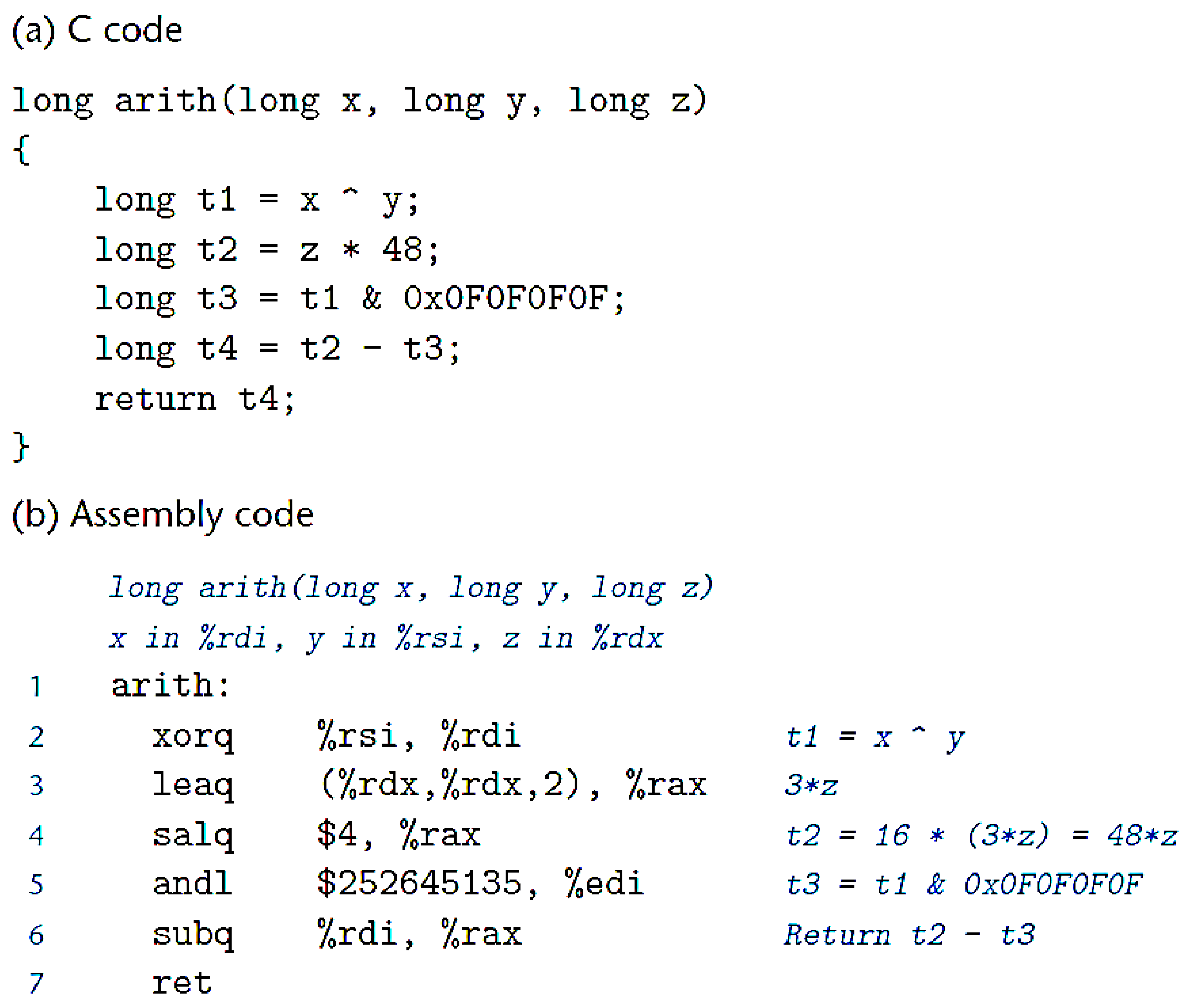
Binary

- Watch out for argument order!
- No distinction between signed and unsigned int (why?)

# Shift

- Both arithmetic and logical right shifts are possible
- The different shift instructions can specify the shift amount either as an immediate value or with the single-byte register %cl. (These instructions are unusual in only allowing this specific register as the operand.)
- With x86-64, a shift instruction operating on data values that are *w* bits long determines the shift amount from the low-order *m* bits of register %cl, where $2^m = w$. The higher-order bits are ignored.
- So, for example, when register %cl has hexadecimal value 0xFF, then instruction salb would shift by 7, while salw would shift by 15, sall would shift by 31, and salq would shift by 63.

# Arithmetic Expression Example



(a) C code

```
long arith(long x, long y, long z)
{
    long t1 = x ^ y;
    long t2 = z * 48;
    long t3 = t1 & 0x0F0F0F0F;
    long t4 = t2 - t3;
    return t4;
}
```

(b) Assembly code

```
long arith(long x, long y, long z)
x in %rdi, y in %rsi, z in %rdx
arith:
  xorq    %rsi, %rdi            t1 = x ^ y
  leaq    (%rdx,%rdx,2), %rax   3*z
  salq    $4, %rax              t2 = 16 * (3*z) = 48*z
  andl    $252645135, %edi      t3 = t1 & 0x0F0F0F0F
  subq    %rdi, %rax            Return t2 - t3
  ret
```

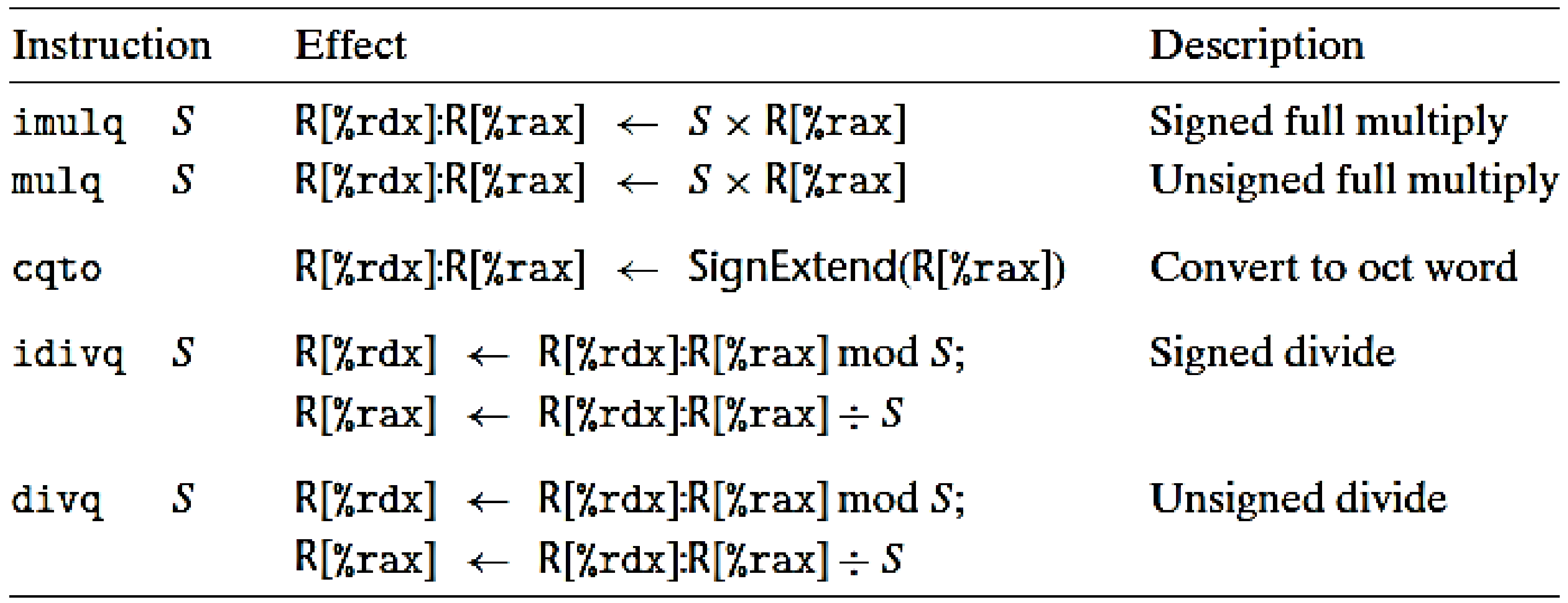
**Figure 3.11** **C and assembly code for arithmetic function.**

# Special Arithmetic Operations

Multiplying two 64-bit signed or unsigned integers can yield a product that requires 128 bits to represent. The x86-64 instruction set provides limited support for operations involving 128-bit (16-byte) numbers.

The imulq instruction has two different forms One form, serves as a “two operand”

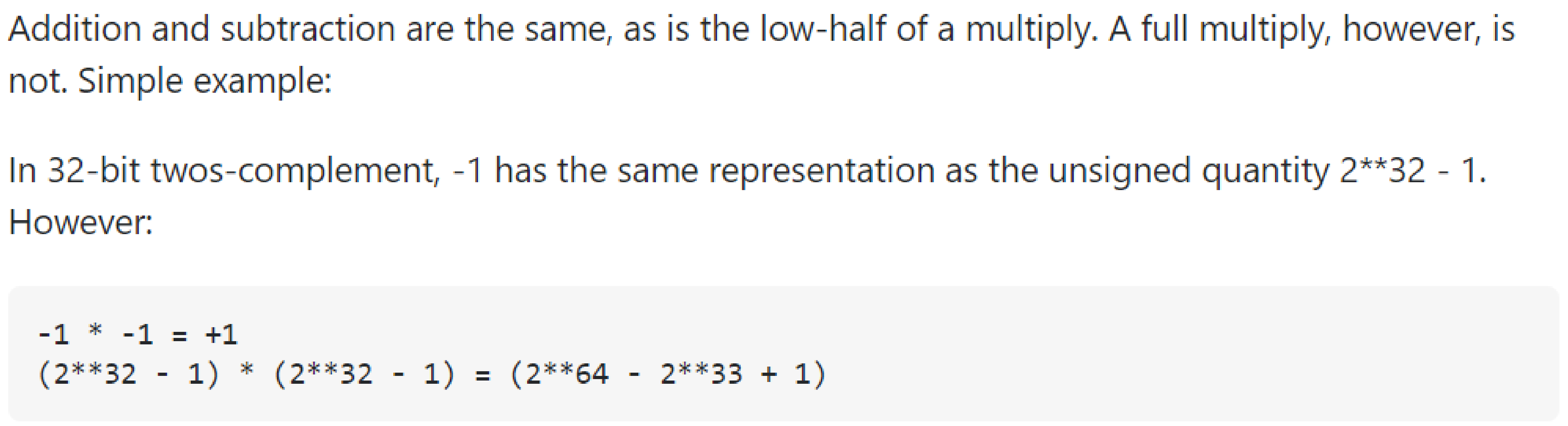
multiply instruction, generating a 64-bit product from two 64-bit operands. The other version is given below:



| Instruction | | Effect | Description |
|---|---|---|---|
| `imulq` | $S$ | R[`%rdx`]:R[`%rax`] $\leftarrow$ $S \times$ R[`%rax`] | Signed full multiply |
| `mulq` | $S$ | R[`%rdx`]:R[`%rax`] $\leftarrow$ $S \times$ R[`%rax`] | Unsigned full multiply |
| `cqto` | | R[`%rdx`]:R[`%rax`] $\leftarrow$ SignExtend(R[`%rax`]) | Convert to oct word |
| `idivq` | $S$ | R[`%rdx`] $\leftarrow$ R[`%rdx`]:R[`%rax`] mod $S$;<br>R[`%rax`] $\leftarrow$ R[`%rdx`]:R[`%rax`] $\div S$ | Signed divide |
| `divq` | $S$ | R[`%rdx`] $\leftarrow$ R[`%rdx`]:R[`%rax`] mod $S$;<br>R[`%rax`] $\leftarrow$ R[`%rdx`]:R[`%rax`] $\div S$ | Unsigned divide |

Figure 3.12 **Special arithmetic operations.** These operations provide full 128-bit multiplication and division, for both signed and unsigned numbers. The pair of registers `%rdx` and `%rax` are viewed as forming a single 128-bit oct word.

# Why separate instructions for signed multiplication and division?



Addition and subtraction are the same, as is the low-half of a multiply. A full multiply, however, is not. Simple example:

In 32-bit twos-complement, -1 has the same representation as the unsigned quantity 2**32 - 1. However:

```
-1 * -1 = +1
(2**32 - 1) * (2**32 - 1) = (2**64 - 2**33 + 1)
```

## Machine Programming I: Summary

- History of Intel processors and architectures
  - Evolutionary design leads to many quirks and artifacts
- C, assembly, machine code
  - New forms of visible state: program counter, registers, ...
  - Compiler must transform statements, expressions, procedures into low-level instruction sequences
- Assembly Basics: Registers, operands, move
  - The x86-64 move instructions cover wide range of data movement forms
- Arithmetic
  - C compiler will figure out different instruction combinations to carry out computation