Assembly

תכנות באסמבלי

Motivation

- Low level programming
- Better understanding of program structure
- Disassembly (Debugging or Reverse Engineering)

Assembly vs. Higher level languages

- There are NO variables' type definitions.
  - All kinds of data are stored in the same registers.
  - We need to know what we are working on in order to use the right instructions.
  - Memory = a large, byte-addressable array.
- Only a limited set of registers is used to store data while running the program.
  - If we need more room we must save the data into memory and later reread it.
- No special structures (instructions) for "if" / "switch" / "loops" (for, while, do-while), or even functions!

How to - Disassembly of code

- Compilation of code:
  - gcc -c code.c
  - We get the file: code.o
- Disassembly:
  - objdump -d code.o
  - We get an assembly-like code that represents the c code appeared in file code.c
- Or:
  - gcc -s code.c
  - We get a code.s file that contains an assembly code created by the compiler.

Standard data types

Assembly

C declaration	Intel data type	GAS suffix	x86-64 Size (Bytes)
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double word	`l`	4
`unsigned`	Double word	`l`	4
`long int`	Quad word	`q`	8
`unsigned long`	Quad word	`q`	8
`char *`	Quad word	`q`	8
`float`	Single precision	`s`	4
`double`	Double precision	`d`	8
`long double`	Extended precision	`t`	16

In Assembly: size = type of variable.

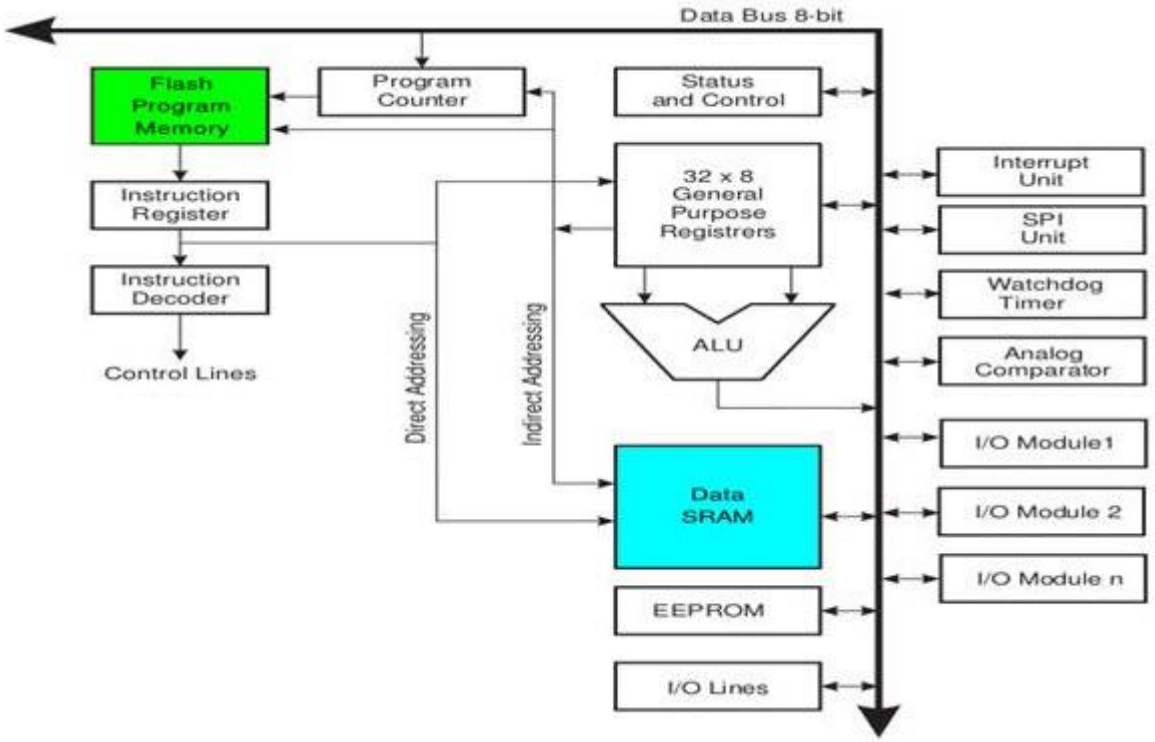
Words, double words ….

- Due to its origins as a 16-bit architecture that expanded into a 32-bit one, Intel uses the term “word” to refer to a 16-bit data type.
- 32-bit quantities as “double words”.
- 64-bit quantities as “quad words”.
- Most instructions we will encounter operate on bytes, double words or quad words.
- Each instruction has 4 variants, depending on its suffix (‘b’ – byte / ‘w’ – word / ‘l’ – double word / ‘q’ –quad word).

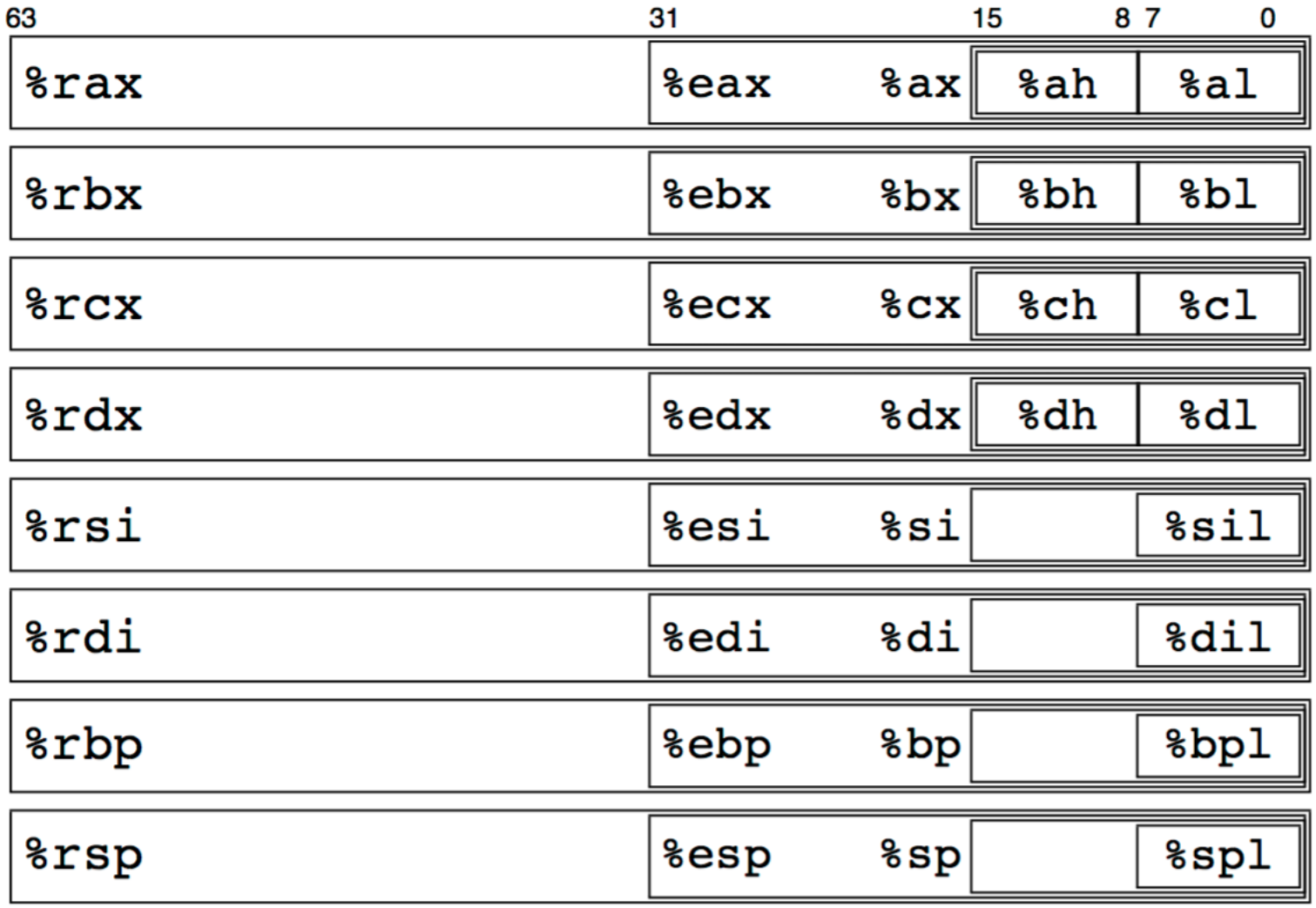
The Registers

- An x86-64 CPU contains a set of 16 *registers* storing 64-bit values. These registers are used to store integer data as well as pointers.
- The registers names all begin with %r, but otherwise they have peculiar names.
- In the original 8086 CPU each register (%ax to %bp) had a specific target (and hence it got its name). Today **most** of these targets are less significant.
  - Some instructions use fixed registers as sources and/or destinations.
  - A set of programming conventions governs how the registers are to be used for passing arguments, returning values from functions and storing local data.
  - %rsp contains a pointer to important places in the program stack (%rbp sometimes, as well).

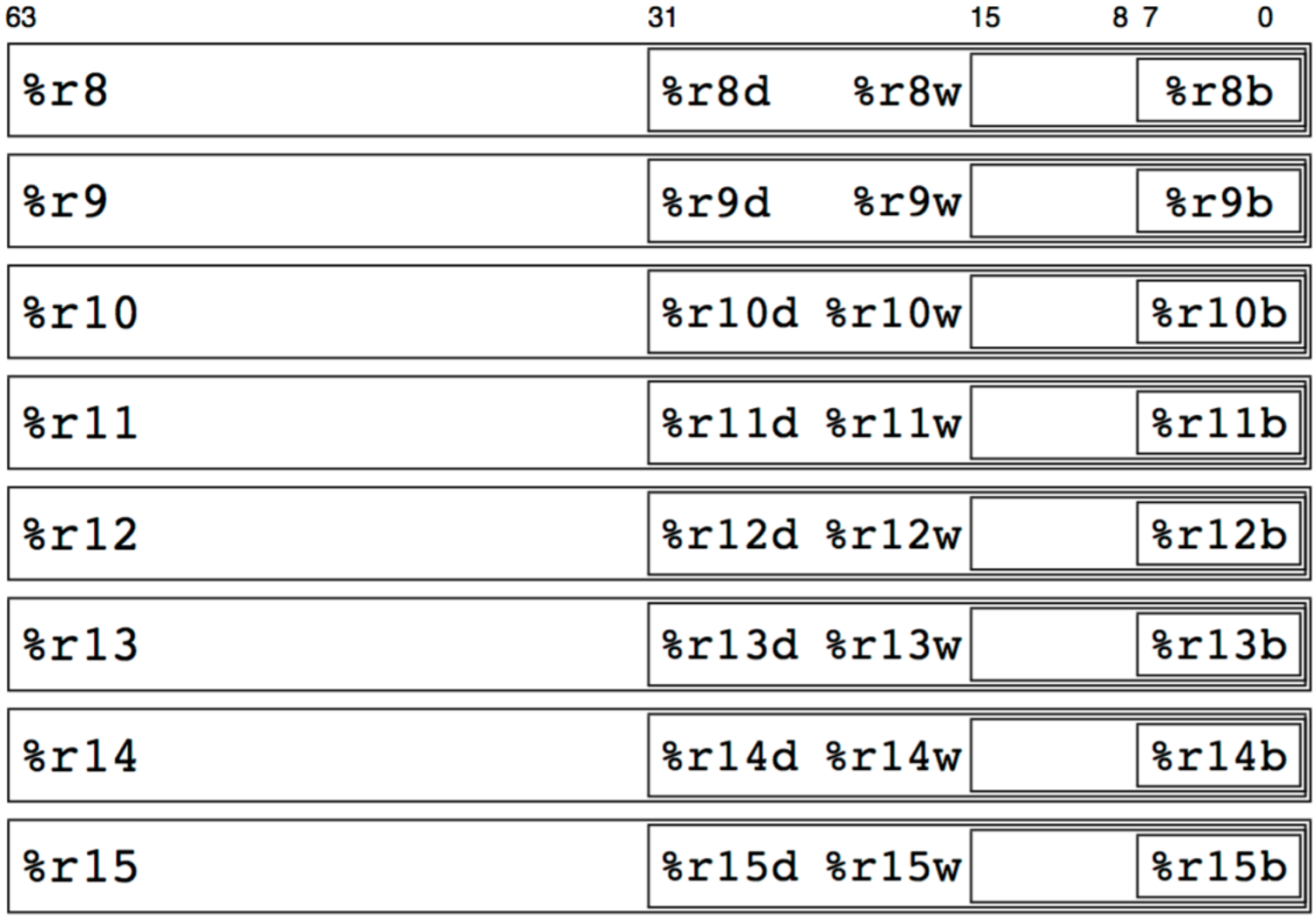
תזכורת


Data Bus 8-bit
Flash Program Memory
Program Counter
Status and Control
Instruction Register
Instruction Decoder
Control Lines
32 x 8 General Purpose Registrers
ALU
Direct Addressing
Indirect Addressing
Data SRAM
EEPROM
I/O Lines
Interrupt Unit
SPI Unit
Watchdog Timer
Analog Comparator
I/O Module1
I/O Module 2
I/O Module n

The Register File



The Register File – Cont.



Partial access to a register

- The **first** low-order byte of each register can be accessed directly.
- The **second** low-order byte of %rax, %rbx, %rcx, %rdx can also be accessed directly (backward compatibility).
- When a byte instruction updates one of these single-byte “register elements,” the remaining bytes of the register do not change.
- Same goes for the low-order 16 bits of each register, using word operation instructions.
- What about writing to the low order 4 bytes? Later.

Operand Forms

Type	Form	Operand Value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	E_a	$Reg[\mathrm{E}_a]$	Register
Memory	Imm	$Mem[Imm]$	Absolute
Memory	(E_a)	$Mem[Reg[\mathrm{E}_a]]$	Indirect
Memory	$Imm\,(\mathrm{E}_b)$	$Mem[Imm + Reg[\mathrm{E}_b]]$	Base + Displacement
Memory	$(\mathrm{E}_b, \mathrm{E}_i)$	$Mem[Reg[\mathrm{E}_b] + Reg[\mathrm{E}_i]]$	Indexed
Memory	$Imm\,(\mathrm{E}_b, \mathrm{E}_i)$	$Mem[Imm + Reg[\mathrm{E}_b] + Reg[\mathrm{E}_i]]$	Indexed
Memory	$(, \mathrm{E}_i, s)$	$Mem[Reg[\mathrm{E}_i] \cdot s]$	Scaled Indexed
Memory	$Imm\,(, \mathrm{E}_i, s)$	$Mem[Imm + Reg[\mathrm{E}_i] \cdot s]$	Scaled Indexed
Memory	$(\mathrm{E}_b, \mathrm{E}_i, s)$	$Mem[Reg[\mathrm{E}_b] + Reg[\mathrm{E}_i] \cdot s]$	Scaled Indexed
Memory	$Imm\,(\mathrm{E}_b, \mathrm{E}_i, s)$	$Mem[Imm + Reg[\mathrm{E}_b] + Reg[\mathrm{E}_i] \cdot s]$	Scaled Indexed

Figure 3.3: **Operand Forms.** Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor s must be either 1, 2, 4, or 8.

Important Suffixes

- ‘q’ - quad word
- ‘l’ - double word
- ‘w’ - word
- ‘b’ - byte
- ‘s’ - single (for floating point)
- ‘t’ - special extension (– we won’t get into that!)

Move to / from memory Instructions

Instruction		Effect	Description
`movq`	S, D	$D \leftarrow S$	Move quad word
`movabsq`	I, R	$R \leftarrow I$	Move quad word
`movslq`	S, R	$R \leftarrow \text{SignExtend}(S)$	Move sign-extended double word
`movsbq`	S, R	$R \leftarrow \text{SignExtend}(S)$	Move sign-extended byte
`movzbq`	S, R	$R \leftarrow \text{ZeroExtend}(S)$	Move zero-extended byte
`pushq`	S	R[%rsp] $\leftarrow$ R[%rsp] $- 8$; M[R[%rsp]] $\leftarrow S$	Push
`popq`	D	$D \leftarrow$ M[R[%rsp]]; R[%rsp] $\leftarrow$ R[%rsp] $+ 8$	Pop

- movq I,D: D← 32-bit immediate
- movabsq I,D: D← 64-bit immeidate

`movq` Operand Combinations

	Source	Destination		C Analog
`movq`	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	*Reg*	`movq (%rax),%rdx`	`temp = *p;`

Cannot do memory-memory transfers with a single instruction

movb & movw & movl

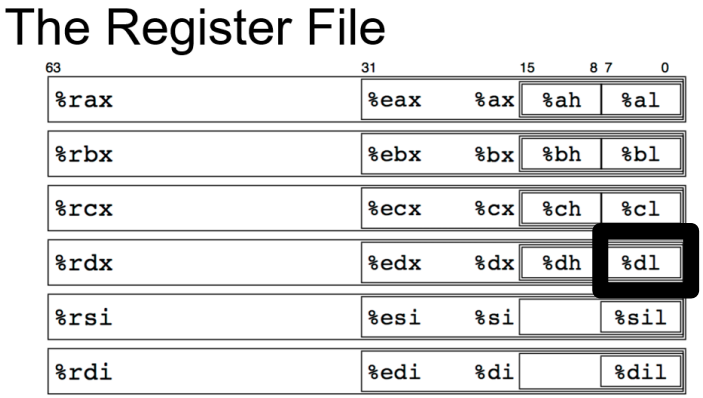
- The movb instruction is similar, but it moves just a single byte. When one of the operands is a register, it must be one of the single-byte register elements.
- Similarly, the movw instruction moves two bytes. When one of its operands is a register, it must be one of the two-byte register elements.
- The movl instruction moves four bytes. When one of its operands is a register, the high-order four bytes are set to zero!

movsbq & movzbq

- Both the movsbq and the movzbq instructions serve to copy a byte and to set the remaining bits in the destination:
  - movsbq - signed extension.
  - movzbq - zero extension.
- Other variations are acceptable
  - movsbw, movsbl, movswl, movswq, movslq
  - movzbw, movzbl, movzwl, movzwq
- Why is there no “mov**zlq**”?
  - Because same result with movl

Another example

(Assume initially that %dl = 0x8D, %rax = 0x0000000098765432)



- movb %dl,%al →%rax = 0…09876548D
- movsbq %dl,%rax →%rax = F…FFFFFFF8D
- movzbl %dl,%eax →%rax = 0…00000008D
- movsbl %dl,%eax →%rax = 0…0FFFFFF8D

C vs. Assembly example

```c
long exchange(long *xp, long y)
{
    long x = *xp;

    *xp = y;
    return x;
}
```

```
# xp in %rdi, y in %rsi
movq    (%rdi), %rax    # Get x at *xp
movq    %rsi, (%rdi)    # Store y at xp
# %rax holds the return value
```

Arithmetic & Logical Operations

Instruction		Effect	Description
`leaq`	S, D	$D \leftarrow$ `&`S	Load effective address
`incq`	D	$D \leftarrow D$ `+ 1`	Increment
`decq`	D	$D \leftarrow D$ `- 1`	Decrement
`negq`	D	$D \leftarrow$ `-`D	Negate
`notq`	D	$D \leftarrow$ `~`D	Complement
`addq`	S, D	$D \leftarrow D$ `+` S	Add
`subq`	S, D	$D \leftarrow D$ `-` S	Subtract
`imulq`	S, D	$D \leftarrow D$ `*` S	Multiply
`xorq`	S, D	$D \leftarrow D$ `^` S	Exclusive-or
`orq`	S, D	$D \leftarrow D$ `|` S	Or
`andq`	S, D	$D \leftarrow D$ `&` S	And
`salq`	k, D	$D \leftarrow D$ `<<` k	Left shift
`shlq`	k, D	$D \leftarrow D$ `<<` k	Left shift (same as `salq`)
`sarq`	k, D	$D \leftarrow D$ `>>` k	Arithmetic right shift
`shrq`	k, D	$D \leftarrow D$ `>>` k	Logical right shift

Arithmetic & Logical Operations (2)

- With the exception of leaq, each of these instructions has a counterpart that operates on double words (32 bits), words (16 bits) and on bytes (by replacing the suffix).
- Again, cannot do memory-memory transfers with single instruction

“Load Effective Address” (leaq)

- The “Load Effective Address” (leaq) instruction is actually a variant of the movq instruction.
- Its first operand appears to be a memory reference, but instead of reading from the designated location, the instruction copies the effective address to the destination.
- This instruction can be used to generate pointers for later memory references.

leaq (2)

- The leaq Instruction can be used to compactly describe common arithmetic operations.
- If register %rdx contains value x, then the instruction:

  leaq 7(%rdx,%rdx,4), %rax

  will set register %rax to 5x + 7.
- It is commonly used to perform simple arithmetic:
  - (%rax = x; %rcx = y)
  - leaq 6(%rax), %rdx = x+6
  - leaq (%rax,%rcx), %rdx = x+y
  - leaq (%rax,%rcx,4), %rdx = x+4y
  - leaq 7(%rax,%rax,8), %rdx = 9x+7
  - leaq 0xA(,%rcx,4), %rdx = 4y+10
  - leaq 9(%rax,%rcx,2), %rdx =x+2y+9

Shift

- Either logical or arithmetic
- k is a number between 0 and 63, or the single-byte register %cl
- Suppose that x and n are stored at memory locations with offsets 16 and 24, respectively, relative to the address in register %rbp

  - get n → movq 24(%rbp), %rcx
  - get x → movq 16(%rbp), %rax
  - x <<= 2 → salq $2,%rax
  - x >>= n → sarq %cl,%rax

C vs. Assembly example

```c
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;
}
```

```
# x in %rdi, y in %rsi, z in %rdx
addq    %rsi, %rdi              # Compute t1 = x + y
leaq    (%rdx,%rdx,2), %rax     # Compute 3*z
salq    $4, %rax                # Compute t2 = 48*z
andq    $252645135, %rdi        # Compute t3 = t1 & 0x0F0F0F0F
imulq   %rdi, %rax              # Compute t4 = t2 * t3
```

mul & div Instructions

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
`cltq`	R[%rax] $\leftarrow$ SignExtend(R[%eax])	Convert %eax to quad word
`cqto`	R[%rdx]:R[%rax] $\leftarrow$ SignExtend(R[%rax])	Convert to oct word
`idivq` S	R[%rdx] $\leftarrow$ R[%rdx]:R[%rax] mod S; R[%rax] $\leftarrow$ R[%rdx]:R[%rax] $\div S$	Signed divide
`divq` S	R[%rdx] $\leftarrow$ R[%rdx]:R[%rax] mod S; R[%rax] $\leftarrow$ R[%rdx]:R[%rax] $\div S$	Unsigned divide

- 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

Code example

(x at %rbp+16, y at %rbp+24)

- movq 16(%rbp),%rax →Put x in %rax
- imulq 24(%rbp) →Multiply by y
- pushq %rdx →Push high-order 64 bits
- pushl %rax →Push low-order 64 bits

Yet, another example

(x at %rbp+16, y at %rbp+24)

- movq 16(%rbp),%rax →Put x in %rax
- cqto →Sign extend into %rdx
- idivq 24(%rbp) →Divide by y
- pushq %rax →Push x / y
- pushq %rdx →Push x % y

Division with 32-bit operands

```
1 movl    $7, %ecx     # put the divisor into %ecx
2 movl    $23, %eax    # put the dividend into %eax
3 cltd                 # sign-extend %eax to %edx:%eax
4 idivl   %ecx         # divide the quad-word %edx:%eax by %ecx
```

- Or:

```
1 movl    $7, %ecx     # put the divisor into %ecx
2 movl    $23, %eax    # put the dividend into %eax
3 cltq                 # sign-extend %eax to entire %rax
4 cqto                 # sign-extend %rax to %rdx:%rax
5 idivl   %ecx         # divide the oct-word %rdx:%rax by %ecx
```