x86 processors totally dominate the laptop/desktop/server market.

Evolutionary Design
- Starting in 1978 with 8086
- Added more features over time.

Complex Instruction Set Computer (CISC)
- Still support many old, now obsolete, features.
- There are many different instructions with many different formats, but only a small subset are encountered with Linux programs.
- Hard to match performance of Reduced Instruction Set Computers (RISC), though Intel has done just that!

BTW: We’re through with Y86 for a while, and starting the x86. We’ll come back to the Y86 later for pipelining.
Historically
- AMD has followed behind Intel
- A little bit slower, a lot cheaper

Then
- Recruited top circuit designers from Digital Equipment Corp. (DEC) and other downward trending companies
- Built Opteron: tough competitor to Pentium 4
- Developed x86-64, their own extension to 64 bits

Recent Years
- Intel got its act together; leads the world in semiconductor technology
- AMD has fallen behind; relies on external semiconductor manufacturers

Transmeta
Radically different approach to implementation.
- Translate x86 code into “very long instruction word” (VLIW) code.
- Very high degree of parallelism.

Centaur / Via
- Continued evolution from Cyrix, the 3rd x86 vendor. Low power, design team in Austin.
- 32-bit processor family.
  - At 2 GHz, around 2 watts; at 600 MHz around 0.5 watt.
- 64-bit processor family, used by HP, Lenovo, OLPC, IBM.
  - Very low power, only a few watts at 1.2 GHz.
  - Full virtualization and SSE support.

Definitions:

Architecture: (also ISA or instruction set architecture). The parts of a processor design one needs in order to understand or write assembly/machine code.
- Examples: instruction set specification, registers

Microarchitecture: implementation of the architecture.
- Examples: cache sizes and core frequency

Code Forms:
- Machine code: the byte-level programs that a processor executes
- Assembly code: a human-readable textual representation of machine code

Example ISAs:
- Intel: x86, IA32, Itanium, x86-64
- ARM: used in almost all mobile phones
Assembly Programmer’s View

Programmer Visible State
- **PC (Program Counter):** address of next instruction. Called `%rip` in x86-64.
- **Condition codes:**
  - Store status info about most recent arithmetic operation.
  - Used for conditional branching.
- **Register file:** heavily used program data.
- **Memory**
  - Byte addressable array.
  - Code, user data, (some) OS data.
  - Includes stack.

ISA Principles
- **Contract between programmer and the hardware.**
  - Defines visible state of the system.
  - Defines how state changes in response to instructions.
- **For Programmer:** ISA is model of how a program will execute.
- **For Hardware Designer:** ISA is formal definition of the correct way to execute a program.
  - With a stable ISA, SW doesn’t care what the HW looks like under the hood.
  - Hardware implementations can change drastically.
  - As long as the HW implements the same ISA, all prior SW should still run.
  - Example: x86 ISA has spanned many chips; instructions have been added but the SW for prior chips still runs.
- **ISA specification:** the binary encoding of the instruction set.

ISA Basics

Architecture vs. Implementation
- **Architecture:** defines what a computer system does in response to a program and set of data.
  - *Programmer visible* elements of computer system.
- **Implementation (microarchitecture):** defines how a computer does it.
  - Sequence of steps to complete operations.
  - Time to execute each operation.
  - Hidden “bookkeeping” function.

*If the architecture changes, some programs may no longer run or return the same answer. If the implementation changes, some programs may run faster/slower/better, but the answers won’t change.*
Examples

Which of the following are part of the architecture and which are part of the implementation? Hint: if the programmer can see/use it (directly) in a program, it’s part of the architecture.

- Number/names of general purpose registers
- Width of memory bus
- Binary representation of each instruction
- Number of cycles to execute a FP instruction
- Condition code bits set by a move instruction
- Size of the instruction cache
- Type of FP format

Turning C into Object Code

- Code in files: p1.c, p2.c
- For minimal optimization, compile with command: `gcc -Og p1.c p2.c -o p`
- Use optimization (-Og); new to recent versions of gcc
- Put resulting binary in file p

Compiling into Assembly

C Code (sum.c):

```c
long plus(long x, long y);

void sumstore(long x, long y, long *dest) {
    long t = plus(x, y);
    *dest = t;
}
```

Run command: `gcc -Og -S sum.c` produces file `sum.s`.

Assembly Characteristics

Minimal Data Types

- “Integer” data of 1, 2, 4 or 8 bytes
- Addresses (untyped pointers)
- Floating point data of 4, 8 or 10 bytes
- No aggregate types such as arrays or structures
- Just contiguously allocated bytes in memory

Primitive Operations

- Perform arithmetic functions on register or memory data
- Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory
- Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches
**Object Code**

- Translates .s into .o
- Binary encoding of each inst.
- Nearly complete image of executable code
- Missing linkages between code in different files

**Linker**

- Resolves references between files
- Combines with static run-time libraries; e.g., code for malloc, printf
- Some libraries are dynamically linked (just before execution)

### Machine Instruction Example

**C Code**

- Store value \( t \) where designated by dest

**Assembly**

- Move 8-byte value to memory (quad word in x86 parlance).
- Operands:
  - \( t \): Register \%rax
  - dest: Register \%rbx
  - \(*\text{dest} \): Memory \( M[\%rbx] \)

**Object Code**

- 3-byte instruction
- Stored at address 0x40059e

---

**Disassembling Object Code**

*This is disassembly of the .o file (no main routine). Offsets are relative.*

```
> objdump -d sumstore.o

sumstore.o: file format elf64-x86-64
Disassembly of section .text:
0000000000000000 <sumstore>:
  0: 53            push %rbx
  1: 48 89 d3      mov %rdx,%rbx
  4: e8 00 00 00 00 callq 9 <sumstore+0x9>
  9: 48 89 03      mov %rax,(%rbx)
 c: 5b            pop %rbx
 d: c3            retq
```

- Useful tool for examining object code
- Analyzes bit pattern of series of instructions
- Produces approximate rendition of assembly code
- Can be run on either a.out (complete executable) or .o file

---

**Alternate Disassembly**

*This is disassembly of the .o file (no main routine). Offsets are relative.*

```
Dump of assembler code for function sumstore:
  0x0000000000000000 <+0>:  push %rbx
  0x0000000000000001 <+1>:  mov %rdx,%rbx
  0x0000000000000004 <+4>:  callq 0x9 <sumstore+9>
  0x0000000000000009 <+9>:  mov %r9,<sumstore+9>
  0x000000000000000c <+12>: pop %rbx
  0x000000000000000d <+13>: retq
```

- Within gdb debugger:
  
  ```
gdb sum
  disassemble sumstore
  x/14xb sumstore
  Examine the 14 bytes starting at sumstore.
  ```
What Can be Disassembled?

- Anything that can be interpreted as executable code.
- Disassembler examines bytes and reconstructs assembly source.

```
% objdump -d WINWORD.EXE

WINWORD.EXE:       file format pei--i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text >:
push %ebp
30001000:  55
30001001:  8b ec
30001003:  6a ff
30001005:  68 90 10 00 30
3000100a:  68 91 dc 4c 30
```

Which Assembler?

**Intel/Microsoft Format**
```
lea   rax, [rcx+rcx*4]
sub   rsp, 8
cmp   quad ptr [ebp-8], 0
mov   rax, quad ptr [rax+4+10h]
```

**GAS/Gnu Format**
```
leaq (%rcx,%rcx,4), %rax
subq $8,%rsp
cmpq $0,-8(%rbp)
movq $0x10(,%rax,4),%rax
```

Intel/Microsoft Differs from GAS

- Operands are listed in opposite order:
  - `mov Dest, Src` vs `movq Src, Dest`
- Constants not preceded by '$'; denote hex with 'h' at end.
  - $0x10
- Operand size indicated by operands rather than operator suffix.
  - `sub` vs `subq`
- Addressing format shows effective address computation.
  - `[rax*4+10h]` vs `$0x10(,%rax,4)`

*From now on we’ll always use GAS assembler format.*

x86-64 Integer Registers

For each of the 64-bit registers, the LS 4 bytes are named 32-bit registers.

<table>
<thead>
<tr>
<th>Reg.</th>
<th>LS 4 bytes</th>
<th>LS 4 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%eax</td>
<td>%r8</td>
</tr>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%r9</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%r10</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%r11</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%r12</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%r13</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
<td>%r14</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
<td>%r15</td>
</tr>
</tbody>
</table>

You can also reference the LS 16-bits (2 bytes) and LS 8-bits (1 byte). For the numbered registers (%r8-%r15) the components are named e.g., %r8d (32-bits), %r8w (16-bits), %r8b (8-bits).

Decomposing the %rax Register

All of the x86’s 64-bit registers have 32-bit, 16-bit and 8-bit accessible internal structure. It varies slightly among the different registers. Example, only %rax, %rbx, %rcx, %rdx allow direct access to byte 1 (%ah).
### Some History: IA32 Registers

<table>
<thead>
<tr>
<th>32-bit reg</th>
<th>16-bit reg</th>
<th>8-bit reg</th>
<th>8-bit Reg</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>%ax</td>
<td>%ah</td>
<td>%al</td>
<td>accumulator</td>
</tr>
<tr>
<td>%ecx</td>
<td>%cx</td>
<td>%ch</td>
<td>%cl</td>
<td>counter</td>
</tr>
<tr>
<td>%edx</td>
<td>%dx</td>
<td>%dh</td>
<td>%dl</td>
<td>data</td>
</tr>
<tr>
<td>%ebx</td>
<td>%bx</td>
<td>%bh</td>
<td>%bl</td>
<td>base</td>
</tr>
<tr>
<td>%esi</td>
<td>%si</td>
<td>%sil*</td>
<td>%bl</td>
<td>base</td>
</tr>
<tr>
<td>%edi</td>
<td>%di</td>
<td>%dil*</td>
<td>%dil*</td>
<td>dest. index</td>
</tr>
<tr>
<td>%esp</td>
<td>%sp</td>
<td>%spl*</td>
<td>%spl*</td>
<td>stack pointer</td>
</tr>
<tr>
<td>%ebp</td>
<td>%bp</td>
<td>%bpl*</td>
<td>%bpl*</td>
<td>base pointer</td>
</tr>
</tbody>
</table>

*These are only available in 64-bit mode.

### Simple Addressing Modes (Same as Y86)

- **Immediate**: value
  - `movq $0xab, %rbx`

- **Register**: Reg[R]
  - `movq %rcx, %rbx`

- **Normal (R)**: Mem[Reg[R]]
  - Register R specifies memory address.
  - This is often called *indirect* addressing.
  - Aha! Pointer dereferencing in C
  - `movq (%rcx), %rax`

- **Displacement D(R)**: Mem[Reg[R] + D]
  - Register R specifies start of memory region.
  - Constant displacement D specifies offset
  - `movq 8(%rcb),%rdx`

### Moving Data

- **Form**: `movq Source, Dest`
  - Move 8-byte “long” word
  - Lots of these in typical code

#### Operand Types

- **Immediate**: Constant integer data
  - Like C constant, but prefixed with ‘$’
  - E.g., $0x400, $-533
  - Encoded with 1, 2, or 4 bytes

- **Register**: One of 16 integer registers
  - Example: %rax, %r13
  - But %rsp is reserved for special use
  - Others have special uses for particular instructions

- **Memory**: source/dest is first address of block
  - Example: (%rax), 0x20(%rbx)
  - Various “addressing modes”

### movq Operand Combinations

Unlike the Y86, we don’t distinguish the operator depending on the operand addressing modes.

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest.</th>
<th>Assembler</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>Register</td>
<td>movq $0x4,%rax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td>Immediate</td>
<td>Memory</td>
<td>movq $-147,(%rax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td>Register</td>
<td>Register</td>
<td>movq %rax,%rdx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td>Register</td>
<td>Memory</td>
<td>movq %rax,(%rdx)</td>
<td>*p = temp;</td>
</tr>
</tbody>
</table>

Direct memory-memory transfers are not supported.
C programming model is close to machine language.
- Machine language manipulates memory addresses.
  - For address computation;
  - To store addresses in registers or memory.
- C employs pointers, which are just addresses of primitive data elements or data structures.

Examples of operators * and &:
- int a, b; /* declare integers a and b */
- int *a_ptr; /* a is a pointer to an integer */
- a_ptr = a; /* illegal, types don’t match*/
- a_ptr = &a; /* a_ptr holds address of a */
- b = *a_ptr; /* dereference a_ptr and assign value to b */

```c
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```machine
swap:
    movq (%rdi), %rax  # t0 = *xp
    movq (%rsi), %rdx  # t1 = *yp
    movq %rdx, (%rdi)  # *xp = t1
    movq %rax, (%rsi)  # *yp = t0
    ret
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>xp</td>
<td>points into memory</td>
</tr>
<tr>
<td>%rsi</td>
<td>yp</td>
<td>points into memory</td>
</tr>
<tr>
<td>%rax</td>
<td>t0</td>
<td>temporary storage</td>
</tr>
<tr>
<td>%rdx</td>
<td>t1</td>
<td>temporary storage</td>
</tr>
</tbody>
</table>

Initial State:

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi 0x120</td>
<td>123 0x120</td>
</tr>
<tr>
<td>%rsi 0x100</td>
<td>0x118 0x110</td>
</tr>
<tr>
<td>%rax</td>
<td>0x108 0x100</td>
</tr>
<tr>
<td>%rdx</td>
<td>456 0x120</td>
</tr>
</tbody>
</table>
Understanding Swap (3)

swap:

\[
\begin{align*}
\text{movq} & \ (% \ rdi) , \ %rax \quad \# \ t0 = *xp, \leftarrow \text{PC here} \\
\text{movq} & \ (% \ rsi) , \ %rdx \quad \# \ t1 = *yp \\
\text{movq} & \ %rdx , \ (% \ rdi) \quad \# \ *xp = t1, \leftarrow \text{PC here} \\
\text{movq} & \ %rax , \ (% \ rsi) \quad \# \ *yp = t0 \\
\text{ret} &
\end{align*}
\]

Registers

<table>
<thead>
<tr>
<th>%rdi</th>
<th>0x120</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td>123</td>
</tr>
<tr>
<td>%rdx</td>
<td>456</td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th></th>
<th>0x120</th>
<th>0x118</th>
<th>0x110</th>
<th>0x108</th>
<th>0x100</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>123</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rsi</td>
<td></td>
<td>123</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rax</td>
<td></td>
<td></td>
<td>123</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td></td>
<td></td>
<td></td>
<td>123</td>
<td></td>
</tr>
</tbody>
</table>

Understanding Swap (4)

\[
\begin{align*}
\text{movq} & \ (% \ rdi) , \ %rax \quad \# \ t0 = *xp \\
\text{movq} & \ (% \ rsi) , \ %rdx \quad \# \ t1 = *yp, \leftarrow \text{PC here} \\
\text{movq} & \ %rdx , \ (% \ rdi) \quad \# \ *xp = t1 \\
\text{movq} & \ %rax , \ (% \ rsi) \quad \# \ *yp = t0 \\
\text{ret} &
\end{align*}
\]

Registers

<table>
<thead>
<tr>
<th>%rdi</th>
<th>0x120</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td>123</td>
</tr>
<tr>
<td>%rdx</td>
<td>456</td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th></th>
<th>0x120</th>
<th>0x118</th>
<th>0x110</th>
<th>0x108</th>
<th>0x100</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>123</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rsi</td>
<td></td>
<td>123</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rax</td>
<td></td>
<td></td>
<td>123</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td></td>
<td></td>
<td></td>
<td>123</td>
<td></td>
</tr>
</tbody>
</table>

Understanding Swap (5)

swap:

\[
\begin{align*}
\text{movq} & \ (% \ rdi) , \ %rax \quad \# \ t0 = *xp \\
\text{movq} & \ (% \ rsi) , \ %rdx \quad \# \ t1 = *yp \\
\text{movq} & \ %rdx , \ (% \ rdi) \quad \# \ *xp = t1 \\
\text{movq} & \ %rax , \ (% \ rsi) \quad \# \ *yp = t0, \leftarrow \text{PC here} \\
\text{ret} &
\end{align*}
\]

Registers

<table>
<thead>
<tr>
<th>%rdi</th>
<th>0x120</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td>123</td>
</tr>
<tr>
<td>%rdx</td>
<td>456</td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th></th>
<th>0x120</th>
<th>0x118</th>
<th>0x110</th>
<th>0x108</th>
<th>0x100</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>123</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rsi</td>
<td></td>
<td>123</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rax</td>
<td></td>
<td></td>
<td>123</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td></td>
<td></td>
<td></td>
<td>123</td>
<td></td>
</tr>
</tbody>
</table>

Understanding Swap (6)

\[
\begin{align*}
\text{movq} & \ (% \ rdi) , \ %rax \quad \# \ t0 = *xp \\
\text{movq} & \ (% \ rsi) , \ %rdx \quad \# \ t1 = *yp \\
\text{movq} & \ %rdx , \ (% \ rdi) \quad \# \ *xp = t1 \\
\text{movq} & \ %rax , \ (% \ rsi) \quad \# \ *yp = t0, \leftarrow \text{PC here} \\
\text{ret} &
\end{align*}
\]

Registers

<table>
<thead>
<tr>
<th>%rdi</th>
<th>0x120</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td>123</td>
</tr>
<tr>
<td>%rdx</td>
<td>456</td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th></th>
<th>0x120</th>
<th>0x118</th>
<th>0x110</th>
<th>0x108</th>
<th>0x100</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>123</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rsi</td>
<td></td>
<td>123</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rax</td>
<td></td>
<td></td>
<td>123</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td></td>
<td></td>
<td></td>
<td>123</td>
<td></td>
</tr>
</tbody>
</table>

CS429 Slideset 7: 33
Instruction Set Architecture II
Simple Addressing Modes

- **Immediate**: value
  
  ```assembly```
  movq $0xab, %rbx
  ```

- **Register**: Reg[R]
  
  ```assembly```
  movq %rcx, %rbx
  ```

- **Normal (R)**: Mem[Reg[R]]
  
  - Register R specifies memory address.
  - This is often called *indirect* addressing.
  - Aha! Pointer dereferencing in C
  
  ```assembly```
  movq (%rcx), %rax
  ```

- **Displacement D(R)**: Mem[Reg[R] + D]
  
  - Register R specifies start of memory region.
  - Constant displacement D specifies offset
  
  ```assembly```
  movq 8(%rcx),%rdx
  ```

Indexed Addressing Modes

- **Most General Form**:
  
  ```assembly```
  D(Rb, Ri, S) [Mem[Reg[Rb] + S*Reg[Ri] + D]
  ```

  - D: Constant “displacement” of 1, 2 or 4 bytes
  - Rb: Base register, any of the 16 integer registers
  - Ri: Index register, any except %rsp (and probably not %rbp)
  - S: Scale, must be 1, 2, 4 or 8.

- **Special Cases**:
  
  ```assembly```
  (Rb, Ri) [Mem[Reg[Rb] + Reg[Ri]]
  ```

  ```assembly```
  D(Rb, Ri) [Mem[Reg[Rb] + Reg[Ri] + D]
  ```

  ```assembly```
  (Rb, Ri, S) [Mem[Reg[Rb] + S * Reg[Ri]]
  ```

The scaling factor s can only be 1, 2, 4, or 8.
Indexed addressing modes are extremely useful when iterating over an array.

```c
long sumArray (long A[], int len) {
    long i;
    long sum = 0;

    for (i = 0; i < len; i++)
        sum += A[i];
    return sum;
}
```

> gcc -S -Og test.c
causes `sumArray` on the previous slide to compile to:

```assembly
sumArray:
    movl  $0, %eax
    movl  $0, %edx
    jmp   .L2
.L3:
    addq  (%rdi,%rdx,8), %rax
    addq  $1, %rdx
.L2:
    movslq %esi, %rcx
    cmpq  %rcx, %rdx
    jl    .L3
    rep ret
```

### Some Arithmetic Operations

#### Two operand instructions:

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq Src, Dest</td>
<td>Dest = Dest + Src</td>
</tr>
<tr>
<td>subq Src, Dest</td>
<td>Dest = Dest - Src</td>
</tr>
<tr>
<td>imulq Src, Dest</td>
<td>Dest = Dest * Src</td>
</tr>
<tr>
<td>salq Src, Dest</td>
<td>Dest = Dest &lt;&lt; Src</td>
</tr>
<tr>
<td>sarq Src, Dest</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>shrq Src, Dest</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>xorq Src, Dest</td>
<td>Dest = Dest ^ Src</td>
</tr>
<tr>
<td>andq Src, Dest</td>
<td>Dest = Dest &amp; Src</td>
</tr>
<tr>
<td>orq Src, Dest</td>
<td>Dest = Dest</td>
</tr>
</tbody>
</table>

Watch out for argument order!

There’s no distinction between signed and unsigned. Why?
Address Computation Instruction

**Form:** leaq Src, Dest
- Src is address mode expression.
- Sets Dest to *address* denoted by the expression

LEA stands for “load effective address.”

After the effective address computation, place the *address*, not the contents of the address, into the destination.

### Consider the following computation:

<table>
<thead>
<tr>
<th>Reg.</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x100</td>
</tr>
<tr>
<td>%rbx</td>
<td>0x200</td>
</tr>
</tbody>
</table>

- movq 0x10(%rbx, %rax, 4), %rcx
- leaq 0x10(%rbx, %rax, 4), %rdx

After this sequence,
- %rcx will contain the **contents** of location 0x610;
- %rdx will contain the number (address) 0x610.

What should the following do?

```
leaq %rbx, %rdx
```

It really shouldn’t be legal since %rbx doesn’t have an address. However, the semantics makes it equal to movq %rbx, %rdx.

### The leaq instruction is widely used for address computations and for some general arithmetic computations.

**Uses:**
- Computing address without doing a memory reference:
  - E.g., translation of \( p = &x[i]; \)
- Computing arithmetic expressions of the form \( x + k \times y \), where \( k \in \{1,2,4,8\} \)

**Example:**

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

**Converted to ASM by compiler:**

```
leaq (%rdi,%rdi,2),%rax  # t <- x+x+2
salq $2,%rax  # ret. t<<2
```
long arith
(long x, long y, long z)
{
    long t1 = x+y;
    long t2 = z+t1;
    long t3 = x+4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}

interesting instructions:

- leaq: address computation
- salq: shift
- imulq: multiplication, but only used once

isa ii: 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, etc.
- Compiler must transform statements, expressions, procedures into low-level instruction sequences

assembly basics: registers, operands, move
- The x86-64 move instructions cover a wide range of data movement forms

arithmetic
- C compiler will figure out different instruction combinations to carry out computation