CS429: Computer Organization and Architecture
Instruction Set Architecture II

Dr. Bill Young
Department of Computer Sciences
University of Texas at Austin

Last updated: June 26, 2018 at 09:35
Topics of this Slideset

- Assembly Programmer’s Execution Model
- Accessing Information
- Registers
- Memory
- Arithmetic operations

BTW: We’re through with Y86 for a while, and starting the x86. We’ll come back to the Y86 later for pipelining.
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!
Machine Evolution

<table>
<thead>
<tr>
<th>Model</th>
<th>Date</th>
<th>Trans.</th>
</tr>
</thead>
<tbody>
<tr>
<td>386</td>
<td>1985</td>
<td>0.3M</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>3.1M</td>
</tr>
<tr>
<td>Pentium/MMX</td>
<td>1997</td>
<td>4.5M</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1995</td>
<td>6.5M</td>
</tr>
<tr>
<td>Pentium III</td>
<td>1999</td>
<td>8.2M</td>
</tr>
<tr>
<td>Pentium 4</td>
<td>2001</td>
<td>42M</td>
</tr>
<tr>
<td>Core 2 Duo</td>
<td>2006</td>
<td>291M</td>
</tr>
<tr>
<td>Core i7</td>
<td>2008</td>
<td>731M</td>
</tr>
</tbody>
</table>

Added Features

- Instructions to support multimedia operations
- Instructions to enable more efficient conditional operations
- Transition from 32 to 64 bits
- More cores
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
x86 Evolution: Clones

**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
### Abstract vs. Concrete Machine Models

#### Machine Models

- **C**
  - Memory
  - Processor

#### Data
1. char
2. int, float
3. double
4. struct, array
5. pointer

#### Control
1. loops
2. conditionals
3. switch
4. proc. call
5. proc. return

#### Assembly

- Memory
- Stack
- Registers (Regs)
- Processor
- ALU

#### Data
1. byte
2. 2-byte word
3. 4-byte long word
4. 8-byte quad word
5. contiguous byte allocation
6. address of initial byte

1. branch/jump
2. call
3. ret
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.
Instruction formats
  Instruction types
  Addressing modes

Instruction
  Op | Mode | Ra | Rb

Before State
  Memory
  Regs

Data type
  Operations
  Interrupts / Events

After State
  Memory
  Regs

Machine State
Memory organization
Register organization
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.*
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
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`.

```
sumstore:
    pushq %rbx # save %rbx
    movq %rdx, %rbx # temp ← dest
    call plus
    movq %rax, (%rbx) # *dest ← t
    popq %rbx # restore %rbx
    ret
```

Warning: you may get different results due to variations in gcc and compiler settings.
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

0x0400595:
  0x53
  0x48  # total of
  0x89  # 14 bytes
  0xd3
  0xe8  # each inst
  0xf2  # 1, 3, or
  0xff  # 5 bytes
  0xff
  0xff  # starts at
  0x48  # addr
  0x89  # 0x00595
  0x03
  0x5b
  0xc3

Assembler

- 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
  - *dest: Memory M[%rbx]

Object Code
- 3-byte instruction
- Stored at address 0x40059e

\*dest = t;

movq %rax, (%rbx)
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

- objdump -d sum
- 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.*

<table>
<thead>
<tr>
<th>Offset</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000000000000000</td>
<td>push %rbx</td>
<td>0x0000000000000001</td>
</tr>
<tr>
<td>0x0000000000000009</td>
<td>mov %rax,(%rbx)</td>
<td>0x000000000000000c</td>
</tr>
<tr>
<td>0x000000000000000d</td>
<td>retq</td>
<td></td>
</tr>
</tbody>
</table>

End of assembler dump.

Within gdb debugger:

```
gdb sum
disassemble sumstore
x/14xb sumstore
```

Examine the 14 bytes starting at sumstore.
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 >:
30001000: 55 push %ebp
30001001: 8b ec mov %esp, %ebp
30001003: 6a ff push $0xffffffff
30001005: 68 90 10 00 30 push $0x30001090
3000100a: 68 91 dc 4c 30 push $0x304cdc91
Which Assembler?

Intel/Microsoft Format

\[
\begin{align*}
\text{le a } & \text{ rax, [ rcx+rcx*4] } \\
\text{sub} & \text{ rsp, 8} \\
\text{cmp} & \text{ quad ptr [ebp−8], 0} \\
\text{mov} & \text{ rax, quad ptr [rax*4+10h]} \\
\end{align*}
\]

GAS/Gnu Format

\[
\begin{align*}
\text{le a q } & \text{ (%rcx,%rcx ,4), %rax} \\
\text{sub q} & \text{ $8,%rsp} \\
\text{cmp q} & \text{ $0,—8(%rbp)} \\
\text{mov q} & \text{ $0x10(,%rax,4),%rax} \\
\end{align*}
\]

Intel/Microsoft Differs from GAS

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

\text{From now on we’ll always use GAS assembler format.}
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>Reg.</th>
<th>LS 4 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%eax</td>
<td>%r8</td>
<td>%r8d</td>
</tr>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
<td>%r15</td>
<td>%r15d</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).

```
%rax (64)  %eax (32)  %ah  %al  %ax (16)
```
## 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></td>
<td>%sil*</td>
<td>source index</td>
</tr>
<tr>
<td>%edi</td>
<td>%di</td>
<td></td>
<td>%dil*</td>
<td>dest. index</td>
</tr>
<tr>
<td>%esp</td>
<td>%sp</td>
<td></td>
<td>%spl*</td>
<td>stack pointer</td>
</tr>
<tr>
<td>%ebp</td>
<td>%bp</td>
<td></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

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”
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>
<tr>
<td>Memory</td>
<td>Register</td>
<td>movq (%rax),%rdx</td>
<td>temp = *p</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;
}
```

Swap:
```
movq (%rdi), %rax
movq (%rsi), %rdx
movq %rdx, (%rdi)
movq %rax, (%rsi)
ret
```
```c
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

### Registers

<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>
swap:

\[
\begin{align*}
\text{movq} & \ (%r_d\ i\ ) , \ %r_{ax} & \# & t_0 = *x_p \\
\text{movq} & \ (%r_s\ i\ ) , \ %r_{dx} & \# & t_1 = *y_p \\
\text{movq} & \ %r_{dx} , \ (%r_d\ i\ ) & \# & *x_p = t_1 \\
\text{movq} & \ %r_{ax} , \ (%r_s\ i\ ) & \# & *y_p = t_0 \\
\text{ret} &
\end{align*}
\]

Initial State:

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td>0x0</td>
</tr>
<tr>
<td>%rdx</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>123</th>
<th>0x120</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td></td>
<td>456</td>
<td>0x100</td>
</tr>
</tbody>
</table>
Understanding Swap (3)

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

<table>
<thead>
<tr>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
</tr>
<tr>
<td>%rsi</td>
</tr>
<tr>
<td>%rax</td>
</tr>
<tr>
<td>%rdx</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
</tr>
<tr>
<td>0x118</td>
</tr>
<tr>
<td>0x110</td>
</tr>
<tr>
<td>0x108</td>
</tr>
<tr>
<td>456</td>
</tr>
</tbody>
</table>
swap:

\[
\begin{align*}
\text{movq} & \ (\% rdi), \ %rax \ \# \ t0 = *x_p \\
\text{movq} & \ (%rsi), \ %r dx \ \# \ t1 = *y_p, \leftarrow \text{PC here} \\
\text{movq} & \ %r dx, \ (%rdi) \ \# \ *x_p = t1 \\
\text{movq} & \ %rax, \ (%rsi) \ \# \ *y_p = t0 \\
\text{ret} & \ 
\end{align*}
\]

Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
</tr>
<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>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>123</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td>456</td>
</tr>
</tbody>
</table>
swap:
    movq (%rdi), %rax  # t0 = *xp
    movq (%rsi), %rdx  # t1 = *yp
    movq %rdx, (%rdi)  # *xp = t1, <-- PC here
    movq %rax, (%rsi)  # *yp = t0
    ret

Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
</tr>
<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>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>456</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td>456</td>
</tr>
</tbody>
</table>
swap:

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

**Registers**

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
</tr>
<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**

\[
\begin{array}{cc}
456 & 0x120 \\
0x118 & 0x110 \\
0x108 & 0x100 \\
123 & \\
\end{array}
\]
Simple Addressing Modes

- **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(%rcx), %rdx
  ```
Indexed Addressing Modes

Most General Form:

\[ D(Rb, Ri, S) \rightarrow \text{Mem}[\text{Reg}[Rb] + S*\text{Reg}[Ri] + D] \]

- **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:

- \((Rb, Ri)\) \rightarrow \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri]]
- \(D(Rb, Ri)\) \rightarrow \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] + D]
- \((Rb, Ri, S)\) \rightarrow \text{Mem}[\text{Reg}[Rb] + S*\text{Reg}[Ri]]
## Addressing Modes

<table>
<thead>
<tr>
<th>Type</th>
<th>Form</th>
<th>Operand value</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>$D$</td>
<td>D</td>
<td>Immediate</td>
</tr>
<tr>
<td>Register</td>
<td>$E_a$</td>
<td>$R[E_a]$</td>
<td>Register</td>
</tr>
<tr>
<td>Memory</td>
<td>D</td>
<td>$M[D]$</td>
<td>Absolute</td>
</tr>
<tr>
<td>Memory</td>
<td>$(E_a)$</td>
<td>$M[R[E_a]]$</td>
<td>Indirect</td>
</tr>
<tr>
<td>Memory</td>
<td>$D(E_b)$</td>
<td>$M[D + R[E_b]]$</td>
<td>Base + displacement</td>
</tr>
<tr>
<td>Memory</td>
<td>$(E_b, E_i)$</td>
<td>$M[R[E_b] + R[E_i]]$</td>
<td>Indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$D(E_b, E_i)$,</td>
<td>$M[D + R[E_b] + R[E_i]]$</td>
<td>Indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$(E_i, s)$</td>
<td>$M[R[E_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$D(E_i, s)$</td>
<td>$M[D + R[E_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$(E_b, E_i, s)$</td>
<td>$M[R[E_b] + R[E_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$D(E_b, E_i, s)$</td>
<td>$M[D + R[E_b] + R[E_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
</tbody>
</table>

The scaling factor $s$ can only be 1, 2, 4, or 8.
### Address Computation Example

<table>
<thead>
<tr>
<th>Expression</th>
<th>Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%rdx)</td>
<td>0xf000 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>(%rdx, %rcx)</td>
<td>0f000 + 0x100</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%rdx, %rcx, 4)</td>
<td>0xf000 + 4*0x100</td>
<td>0xf400</td>
</tr>
<tr>
<td>0x80(,%rdx, 2)</td>
<td>2*0xf000 + 0x80</td>
<td>0x1e080</td>
</tr>
<tr>
<td>0x80(%rdx, 2)</td>
<td>Illegal. <strong>Why?</strong></td>
<td></td>
</tr>
<tr>
<td>0x80(,%rdx, 3)</td>
<td>Illegal. <strong>Why?</strong></td>
<td></td>
</tr>
</tbody>
</table>

### Variables

<table>
<thead>
<tr>
<th>%rdx</th>
<th>0xf000</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rcx</td>
<td>0x100</td>
</tr>
</tbody>
</table>
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>
<th>Order</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addq Src, Dest</code></td>
<td><code>Dest = Dest + Src</code></td>
<td></td>
</tr>
<tr>
<td><code>subq Src, Dest</code></td>
<td><code>Dest = Dest - Src</code></td>
<td></td>
</tr>
<tr>
<td><code>imulq Src, Dest</code></td>
<td><code>Dest = Dest * Src</code></td>
<td></td>
</tr>
<tr>
<td><code>salq Src, Dest</code></td>
<td><code>Dest = Dest &lt;&lt; Src</code></td>
<td>same as <code>shlq</code></td>
</tr>
<tr>
<td><code>sarq Src, Dest</code></td>
<td><code>Dest = Dest &gt;&gt; Src</code></td>
<td>arithmetic</td>
</tr>
<tr>
<td><code>shrq Src, Dest</code></td>
<td><code>Dest = Dest &gt;&gt; Src</code></td>
<td>logical</td>
</tr>
<tr>
<td><code>xorq Src, Dest</code></td>
<td><code>Dest = Dest ^ Src</code></td>
<td></td>
</tr>
<tr>
<td><code>andq Src, Dest</code></td>
<td><code>Dest = Dest &amp; Src</code></td>
<td></td>
</tr>
<tr>
<td><code>orq Src, Dest</code></td>
<td>`Dest = Dest</td>
<td>Src`</td>
</tr>
</tbody>
</table>

- **Watch out for argument order!**
- **There’s no distinction between signed and unsigned. Why?**
### One operand instructions:

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>incq Dest</td>
<td>Dest = Dest + 1</td>
</tr>
<tr>
<td>decq Dest</td>
<td>Dest = Dest - 1</td>
</tr>
<tr>
<td>negq Dest</td>
<td>Dest = -Dest</td>
</tr>
<tr>
<td>notq Dest</td>
<td>Dest = ~Dest</td>
</tr>
</tbody>
</table>

More instructions in the book.
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>

\[
\text{movq} \ 0\times10(\%rbx, \%rax, 4), \%rcx \\
\text{leaq} \ 0\times10(\%rbx, \%rax, 4), \%rdx
\]

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

What should the following do?

\[
\text{leaq} \ %rbx, \%rdx
\]
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>

\[
\text{movq } 0\times10(\%rbx, \%rax, 4), \%rcx \\
\text{leaq } 0\times10(\%rbx, \%rax, 4), \%rdx
\]

After this sequence,

- \%rcx will contain the \textit{contents} of location 0x610;
- \%rdx will contain the number (address) 0x610.

**What should the following do?**

\[
\text{leaq } \%rbx, \%rdx
\]

It really shouldn’t be legal since \%rbx doesn’t have an address. However, the semantics makes it equal to \textit{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
```
Arithmetic Expression Example

\begin{verbatim}
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;
}
\end{verbatim}

Interesting instructions:

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