Machine-Level Representation

CSCI 2021: Machine Architecture and Organization

Antonia Zhai
Department Computer Science and Engineering
University of Minnesota
http://www.cs.umn.edu/~zhai

With Slides from Bryant and O'Hallaron

Detour: A short history of PC
Architecture design is driven by Moore's law

Evolutionary Design
- Starting in 1978 with 8086
- Added more features as time goes on
- Still support old features, although obsolete

Complex Instruction Set Computer (CISC)
- Many different instructions with many different formats
  - But, only small subset encountered with Linux programs
- Hard to match performance of Reduced Instruction Set Computers (RISC)
- But, Intel has done just that!
X86 Evolution: Programmer's View

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>1978</td>
<td>29K</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 16-bit processor. Basis for IBM PC &amp; DOS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Limited to 1MB address space. DOS only gives you 640K</td>
</tr>
<tr>
<td>80286</td>
<td>1982</td>
<td>134K</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added elaborate, but not very useful, addressing scheme</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Basis for IBM PC-AT and Windows</td>
</tr>
<tr>
<td>386</td>
<td>1985</td>
<td>275K</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Extended to 32 bits. Added “flat addressing”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Capable of running Unix</td>
</tr>
<tr>
<td>486</td>
<td>1989</td>
<td>1.9M</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>3.1M</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1995</td>
<td>6.5M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Big change in underlying microarchitecture</td>
</tr>
<tr>
<td>10-core Xeon</td>
<td>2011</td>
<td>2.6B 3800</td>
</tr>
</tbody>
</table>

Pentium Pro

• History
  • Announced in Feb. '95
  • Basis for Pentium II, Pentium III, and Celeron processors
  • Pentium 4 similar idea, but different details

• Features
  • Dynamically translates instructions to more regular format
    • Very wide, but simple instructions
  • Executes operations in parallel
    • Up to 5 at once
  • Very deep pipeline
    • 12-18 cycle latency
X86 Evolution: Clones

• Advanced Micro Devices (AMD)
  • Historically
    • AMD has followed just behind Intel
    • A little bit slower, a lot cheaper
  • Recently
    • Recruited top circuit designers from Digital Equipment Corp.
    • Exploited the fact that Intel distracted by IA64
    • Now are close competitors to Intel
    • Developing own extension to 64 bits

X86 Evolution: Clones

• Transmeta
  • Recent start-up
  • Radically different approach to implementation
    • Translates x86 code into "Very Long Instruction Word" (VLIW) code
    • High degree of parallelism
  • Shooting for low-power market
Road to IA64

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Itanium</td>
<td>2001</td>
<td>10M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• A 64-bit architecture</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Radically new instruction set designed for high performance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Will be able to run existing IA32 programs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• On-board &quot;x86 engine&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Joint project with Hewlett-Packard</td>
</tr>
<tr>
<td>Itanium 2</td>
<td>2002</td>
<td>221M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Big performance boost</td>
</tr>
</tbody>
</table>

x86 Clones: Advanced Micro Devices (AMD)

• Historically
  • AMD has followed just behind Intel
  • A little bit slower, a lot cheaper

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

- Intel Attempted Radical Shift from IA32 to IA64
  - Totally different architecture (Itanium)
  - Executes IA32 code only as legacy
  - Performance disappointing
- AMD Stepped in with Evolutionary Solution
  - x86-64 (now called "AMD64")
- Intel Felt Obligated to Focus on IA64
  - Hard to admit mistake or that AMD is better
- 2004: Intel Announces EM64T extension to IA32
  - Extended Memory 64-bit Technology
  - Almost identical to x86-64!
- All but low-end x86 processors support x86-64
  - But, lots of code still runs in 32-bit mode

Definitions

- **Architecture**: (also instruction set architecture: ISA) The parts of a processor design that one needs to understand to write assembly code.
  - Examples: instruction set specification, registers.
- **Microarchitecture**: Implementation of the architecture.
  - Examples: cache sizes and core frequency.
- **Example ISAs (Intel)**: x86, IA, IPF

With Slides from Bryant and O’Hallaron
What is computer architecture?

ISA is a contract between the software and the hardware

Programmers

Software

Instruction Set Architecture

Hardware

Silicon

What do we study in computer Architecture?

Computer architecture: Cover many levels of abstractions including ISA, organization and hardware

With Slides from Bryant and O’Hallaron
This Course

• IA32
  • The traditional x86

• x86-64
  • The new standard

• Book
  • Sections 3.1—3.12: IA32
  • Section 3.13: x86-64

• This class
  • Mostly IA32

Assembly Language Details
Assembly Programmer's View

Programmer-Visible State
• EIP: Program Counter
  • Address of next instruction

• Register File
  • Heavily used program data

• Condition Codes
  • Store status information about most recent arithmetic operation

Memory: Byte addressable array
• Code, user data, (some) OS data
• Includes stack

Turning C into Object Code

• Code in files \( p1.c \ p2.c \)
• Compile with command: \( \text{gcc} -O \ p1.c \ p2.c \ -o \ p \)
  • Use optimizations (-O), Put resulting binary in file (-o) p

C program (p1.c p2.c)  Compiler (gcc -S)

Asm program (p1.s p2.s)  Assembler (gcc -c or as)

Object program (p1.o p2.o)  Static libraries (.a)

Linker (gcc or ld)

Executable program (p)

With Slides from Bryant and O'Hallaron
C to Assembly

**C Code**

```c
int sum(int x, int y)
{
    int t = x+y;
    return t;
}
```

**Generated IA32 Assembly**

```
sum:
pushl  %ebp
movl  %esp, %ebp
movl  12(%ebp), %eax
addl  8(%ebp), %eax
addl  8(%ebp), %eax
popl  %ebp
ret
```

Obtain with command

```
gcc -S -O code.c
```

Produces file `code.s`

Some compilers use instruction “leave”

---

Assembly

**Minimal Data Types**

- "Integer" data of 1, 2, or 4 bytes
  - Data values
  - 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 function 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 for sum

0x000000 <sum>: Assembler
0x55
0x89
0xe5
0x8b
0x45
0x0c
0x03
0x04
0x08
0x5d
0xc3

```c
int t = x+y;
```

```assembly
addl 8(%ebp),%eax
```

Similar to expression
x += y

Machine Instruction Example

- **C Code**: Add two signed integers
- **Assembly**: Add 2 4-byte integers
  - Signed/unsigned? Same instruction
  - Operands:
    - x: Register %eax
    - y: Memory M[ebp+8]
    - t: Register %eax
- **Return function value in %eax**
- **Object Code**
  - 3-byte instruction
  - Stored at address 0x6

With Slides from Bryant and O'Hallaron
Disassemble

```
% objdump -d /soft/gcc-4.0.0/debian/bin/gcc
/software/gcc-4.0.0/debian/bin/gcc:     file format elf32-i386
Disassembly of section .init:
  08048e44 <_init>:
  8048e57:       e8 88 04 00 00          call   80492e4 <call_gmon_start>
  8048e5c:       e8 df 04 00 00          call   8049340 <frame_dummy>
  8048e61:       e8 5a e5 00 00          call   80573c0 <_do_global_ctors_aux>
  8048e66:       5b                      pop    %ebx
  8048e67:       c9                      leave
  8048e68:       c3                      ret
Disassembly of section .plt:
  08048e6c <nl_langinfo@plt-0x10>:
  8048e6c:       ff 35 e0 d5 05 08       pushl  0x805d5e0
```

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

Machine Model

```
CPU

<table>
<thead>
<tr>
<th>Register: Reg[R_name]</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
</tr>
<tr>
<td>%edx</td>
</tr>
<tr>
<td>%ecx</td>
</tr>
<tr>
<td>%ebx</td>
</tr>
<tr>
<td>%esi</td>
</tr>
<tr>
<td>%edi</td>
</tr>
<tr>
<td>%esp</td>
</tr>
<tr>
<td>%ebp</td>
</tr>
</tbody>
</table>

Mem[addr]

0x00000000
0x00000001
0x00000010
0x00000011
0x00000100
0x00000101
0x00000110
0xFFF7FFFFE
0xFFF7FFFFF
```
Data Access

Register:
- Just name the register: Ex. %eax
- For the rest of this class, Reg[R] ➔ the value in the register R

Memory:
- Normal (R) Mem[Reg[R]]
  - Register R specifies memory address
    movl (%ecx), %eax
- Displacement D(R) Mem[Reg[R]+D]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset
    movl 8(%ebp), %edx

Moving Data: IA32

Moving Data
movl Source, Dest:

Operand Types
- Immediate: Constant integer data
  - Example: $0x400, $-533
  - Like C constant, but prefixed with ‘$’
  - Encoded with 1, 2, or 4 bytes
- Register: One of 8 integer registers
  - Example: %eax, %edx
  - But %esp and %ebp reserved for special use
  - Others have special uses for particular instructions
- Memory: 4 consecutive bytes of memory at address given by register
  - Simplest example: (%eax)
  - Various other "address modes"
**movl Operand Combinations**

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Imm</strong></td>
<td><strong>Reg</strong></td>
<td>movl $0x4,%eax  temp = 0x4;</td>
</tr>
<tr>
<td><strong>Mem</strong></td>
<td><strong>Reg</strong></td>
<td>movl $-147,(%eax)  *p = -147;</td>
</tr>
<tr>
<td><strong>Reg</strong></td>
<td><strong>Reg</strong></td>
<td>movl %eax,%edx  temp2 = temp1;</td>
</tr>
<tr>
<td><strong>Mem</strong></td>
<td><strong>Reg</strong></td>
<td>movl %eax,(%edx)  *p = temp;</td>
</tr>
<tr>
<td><strong>Mem</strong></td>
<td><strong>Reg</strong></td>
<td>movl (%eax),%edx  temp = *p;</td>
</tr>
</tbody>
</table>

Cannot do memory-memory transfers with single instruction

---

**An Example**

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

```assembly
swap:
pushl %ebp
movl %esp,%ebp
pushl %ebx
movl 12(%ebp),%ecx
movl 8(%ebp),%edx
movl (%ecx),%eax
movl (%edx),%ebx
movl %eax,(%edx)
movl %ebx,(%ecx)
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Body
Understanding Swap

void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}

Stack

Register    Variable
%ecx    yp
%edx    xp
%eax    t1
%ebx    t0

Movl 12(%ebp),%ecx  # ecx = yp
Movl 8(%ebp),%edx  # edx = xp
Movl (%ecx),%eax  # eax = *yp (t1)
Movl (%edx),%ebx  # ebx = *xp (t0)
Movl %eax,(%edx)  # *xp = eax
Movl %ebx,(%ecx)  # *yp = ebx

Address

123  0x124
456  0x120

%eax
%edx
%ecx
%ebx
%esi
%edi
%esp
%ebp  0x104

Movl 12(%ebp),%ecx  # ecx = yp
Movl 8(%ebp),%edx  # edx = xp
Movl (%ecx),%eax  # eax = *yp (t1)
Movl (%edx),%ebx  # ebx = *xp (t0)
Movl %eax,(%edx)  # *xp = eax
Movl %ebx,(%ecx)  # *yp = ebx
Understanding Swap

With Slides from Bryant and O'Hallaron

movl 12(%ebp),%ecx  # ecx = yp
movl 8(%ebp),%edx  # edx = xp
movl (%ecx),%eax  # eax = *yp (t1)
movl (%edx),%ebx  # ebx = *xp (t0)
movl %eax,(%edx)  # *xp = eax
movl %ebx,(%ecx)  # *yp = ebx

Address

Offset

YP  12  0x120  0x110
xp   8  0x124  0x10c
Rtn adr

%ebp  0  0x104  0x100

With Slides from Bryant and O'Hallaron
### Understanding Swap

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>456</td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td>123</td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

The code snippet is as follows:

```assembly
movl 12(%ebp),%ecx  # ecx = yp
movl 8(%ebp),%edx  # edx = xp
movl (%ecx),%eax  # eax = *yp (t1)
movl (%edx),%ebx  # ebx = *xp (t0)
movl %eax,(%edx)  # *xp = eax
movl %ebx,(%ecx)  # *yp = ebx
```

### Address Table

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp</td>
<td>12</td>
</tr>
<tr>
<td>xp</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>Rtn adr</td>
</tr>
<tr>
<td>%ebp</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>-4</td>
</tr>
</tbody>
</table>

With Slides from Bryant and O'Hallaron
### Understanding Swap

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>456</td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td>123</td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

#### Movl Instructions

- `movl 12(%ebp),%ecx` # ecx = yp
- `movl 8(%ebp),%edx` # edx = xp
- `movl (%ecx),%eax` # eax = *yp (t1)
- `movl (%edx),%ebx` # ebx = *xp (t0)
- `movl %eax,(%edx)` # *xp = eax
- `movl %ebx,(%ecx)` # *yp = ebx

![Address Table](image.png)

With Slides from Bryant and O'Hallaron
More on Addressing Modes

Most General Form

\[ D(R_b, R_i, S) \rightarrow Mem[Reg[R_b]+S*Reg[R_i]+D] \]

- **D**: Constant "displacement" 1, 2, or 4 bytes
- **Rb**: Base register: Any of 8 integer registers
- **Ri**: Index register: Any, except for %esp
  - Unlikely you'd use %ebp, either
- **S**: Scale: 1, 2, 4, or 8

Special Cases

- \((R_b, R_i)\)
  - \(Mem[Reg[R_b]+Reg[R_i]]\)
- \(D(R_b, R_i)\)
  - \(Mem[Reg[R_b]+Reg[R_i]+D]\)
- \((R_b, R_i, S)\)
  - \(Mem[Reg[R_b]+S*Reg[R_i]]\)

Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0x8(%edx))</td>
<td>0xf000 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>((%edx, %ecx))</td>
<td>0xf000 + 0x100</td>
<td>0xf100</td>
</tr>
<tr>
<td>((%edx, %ecx, 4))</td>
<td>0xf000 + 4*0x100</td>
<td>0xf400</td>
</tr>
<tr>
<td>(0x80(%edx, 2))</td>
<td>2*0xf000 + 0x80</td>
<td>0x1e080</td>
</tr>
</tbody>
</table>
Address Computation Instruction

- `leal Src, Dest`
  - `Src` is address mode expression
  - Set `Dest` to address denoted by expression
- Uses
  - Computing address without doing memory reference
    - E.g., translation of `p = 4x[i];`
  - Computing arithmetic expressions of the form `x + k*y`
    - `k` = 1, 2, 4, or 8.

Some Arithmetic Operations

- **Two Operand Instructions:**

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addl</code> Src, Dest</td>
<td>Dest = Dest + Src</td>
</tr>
<tr>
<td><code>subl</code> Src, Dest</td>
<td>Dest = Dest - Src</td>
</tr>
<tr>
<td><code>imull</code> Src, Dest</td>
<td>Dest = Dest * Src</td>
</tr>
<tr>
<td><code>sall</code> Src, Dest</td>
<td>Dest = Dest &lt;&lt; Src</td>
</tr>
<tr>
<td><code>sarh</code> Src, Dest</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td><code>xorl</code> Src, Dest</td>
<td>Dest = Dest ^ Src</td>
</tr>
<tr>
<td><code>andl</code> Src, Dest</td>
<td>Dest = Dest &amp; Src</td>
</tr>
<tr>
<td><code>orl</code> Src, Dest</td>
<td>Dest = Dest</td>
</tr>
</tbody>
</table>

- **Watch out for argument order!**
- **No distinction for argument order**!

With Slides from Bryant and O’Hallaron
Some Arithmetic Operations

- **One Operand Instructions**
  
  incl  Dest  Dest = Dest + 1  
  decl Dest  Dest = Dest - 1  
  negl Dest  Dest = - Dest  
  notl Dest  Dest = ~Dest  

- See book for more instructions

---

Understanding arith

int arith
(int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}

With Slides from Bryant and O'Hallaron

movl 8(%ebp),%eax  # eax = x
movl 12(%ebp),%edx  # edx = y
leal (%edx,%eax),%ecx  # ecx = x+y  (t1)
leal (%edx,%edx,2),%edx  # edx = 3*y
sall $4,%edx  # edx = 48*y  (t4)
addl 16(%ebp),%ecx  # ecx = z+t1  (t2)
leal 4(%edx,%eax),%eax  # eax = 4+t4+x  (t5)
imull %ecx,%eax  # eax = t5*t2  (rval)

With Slides from Bryant and O'Hallaron
Understanding arith

```c
int arith (int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

- Instructions in different order from C code
- Some expressions require multiple instructions
- Some instructions cover multiple expressions
- Get exact same code when compile:
  - $(x+y+z) \times (x+4+48y)$

```asm
# eax = x
movl 8(%ebp),%eax
# edx = y
movl 12(%ebp),%edx
# ecx = x+y (t1)
leal (%edx,%eax),%ecx
# edx = 3y
leal (%edx,%edx,2),%edx
# edx = 48y (t4)
sall $4,%edx
# ecx = z+t1 (t2)
addl 16(%ebp),%ecx
# eax = 4+t4+x (t5)
```

Another Example

```c
int logical(int x, int y)
{
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

```asm
pushl %ebp
movl %esp,%ebp
movl 12(%ebp),%eax
movl 8(%ebp),%eax
xorl 8(%ebp),%eax
sarl $17,%eax
andl $8185,%eax
popl %ebp
ret
```

With Slides from Bryant and O’Hallaron
Another Example

```c
int logical(int x, int y)
{
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

With Slides from Bryant and O'Hallaron

Another Example

```c
int logical(int x, int y)
{
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

With Slides from Bryant and O'Hallaron
Another Example

```c
int logical(int x, int y)
{
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

```
X86-64 Assembly

movl 12(%ebp),%eax  # eax = y
xorl 8(%ebp),%eax  # eax = x^y (t1)
sarl $17,%eax      # eax = t1>>17  (t2)
andl $8185,%eax    # eax = t2 & mask (rval)
```

With Slides from Bryant and O'Hallaron
Data Representations: IA32 + x86-64

- Sizes of C Objects (in Bytes)

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Generic 32-bit</th>
<th>Intel IA32</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>8</td>
<td>10/12</td>
<td>16</td>
</tr>
<tr>
<td>char *</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

*Or any other pointer

With Slides from Bryant and O’Hallaron

x86-64 Integer Registers

%rax %eax %r8   %r8d
%rbx %ebx %r9   %r9d
%rcx %ecx %r10  %r10d
%rdx %edx %r11  %r11d
%rsi %esi %r12  %r12d
%rdi %edi %r13  %r13d
%rsp %esp %r14  %r14d
%rbp %ebp %r15  %r15d

- Extend existing registers. Add 8 new ones.
- Make %ebp/%rbp general purpose

With Slides from Bryant and O’Hallaron
Instructions

- Long word l (4 Bytes) ↔ Quad word q (8 Bytes)

- New instructions:
  - movl ↔ movq
  - addl ↔ addq
  - sall ↔ salq
  - etc.

- 32-bit instructions that generate 32-bit results
  - Set higher order bits of destination register to 0
  - Example: addl

32-bit code for swap

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

```assembly
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx

    movl 8(%ebp), %edx
    movl 12(%ebp), %ecx
    movl (%edx), %ebx
    movl (%ecx), %eax
    movl %eax, (%edx)
    movl %ebx, (%ecx)

    popl %ebx
    popl %ebp
    ret
```

With Slides from Bryant and O'Hallaron
### 64-bit code for swap

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

- Operands passed in registers (why useful?)
- First (`xp`) in `%rdi`, second (`yp`) in `%rsi`
- 64-bit pointers
- No stack operations required
- 32-bit data
  - Data held in registers `%eax` and `%edx`
  - `movl` operation

---

### Summary

**Machine Models**

- **C**
  - `mem` — `proc`

**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**

- `mem` — `Stack`
  - `regs` — `alu`
  - Cond. Codes — `processor`

1) `byte`
2) 2-byte word
3) 4-byte long word
4) contiguous byte allocation
5) `address of initial byte`

---

With Slides from Bryant and O’Hallaron