Machine-Level Programming I: Basics

15-213/18-213: Introduction to Computer Systems
5th Lecture, Sep. 9, 2014

Instructors:
Greg Ganger, Greg Kesden, and Dave O’Hallaron
Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Intro to x86-64
Intel x86 Processors

- Dominate laptop/desktop/server market

- Evolutionary design
  - Backwards compatible up until 8086, introduced in 1978
  - Added more features as time goes on

- 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!
    - In terms of speed. Less so for low power.
## Intel x86 Evolution: Milestones

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
<th>MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>1978</td>
<td>29K</td>
<td>5-10</td>
</tr>
<tr>
<td>386</td>
<td>1985</td>
<td>275K</td>
<td>16-33</td>
</tr>
<tr>
<td>Pentium 4F</td>
<td>2004</td>
<td>125M</td>
<td>2800-3800</td>
</tr>
<tr>
<td>Core 2</td>
<td>2006</td>
<td>291M</td>
<td>1060-3500</td>
</tr>
<tr>
<td>Core i7</td>
<td>2008</td>
<td>731M</td>
<td>1700-3900</td>
</tr>
</tbody>
</table>

- First 16-bit Intel processor. Basis for IBM PC & DOS
- 1MB address space
- First 32 bit Intel processor, referred to as IA32
- Added “flat addressing”, capable of running Unix
- First 64-bit Intel processor, referred to as x86-64
- First multi-core Intel processor
- Four cores (our shark machines)
Intel x86 Processors, cont.

Machine Evolution

- 386 1985 0.3M
- Pentium 1993 3.1M
- Pentium/MMX 1997 4.5M
- PentiumPro 1995 6.5M
- Pentium III 1999 8.2M
- Pentium 4 2001 42M
- Core 2 Duo 2006 291M
- Core i7 2008 731M

Added Features

- Instructions to support multimedia operations
- Instructions to enable more efficient conditional operations
- Transition from 32 bits to 64 bits
- More cores
2013 State of the Art

- Core i7 Haswell 2013 1.4B

Features:
- 4 cores
- Max 4.0 GHz Clock
- 84 Watts
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 History

- 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
Our Coverage

- **IA32**
  - The traditional x86
  - `shark> gcc -m32 hello.c`

- **x86-64**
  - The emerging standard
  - `shark> gcc hello.c`
  - `shark> gcc -m64 hello.c`

- **Presentation**
  - Book presents IA32 in Sections 3.1—3.12
  - Covers x86-64 in 3.13
  - We will cover both simultaneously
  - Some labs will be based on x86-64, others on IA32
Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Intro to x86-64
Definitions

- **Architecture:** (also ISA: instruction set architecture) 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): IA32, x86-64
Assembly Programmer’s View

**CPU**
- **PC**: Program counter
  - Address of next instruction
  - Called “EIP” (IA32) or “RIP” (x86-64)

**Registers**

**Condition Codes**
- Store status information about most recent arithmetic or logical operation
- Used for conditional branching

**Memory**
- Byte addressable array
- Code and user data
- Stack to support procedures

**Programmer-Visible State**

- **PC**: Program counter
  - Address of next instruction
  - Called “EIP” (IA32) or “RIP” (x86-64)

- **Register file**
  - Heavily used program data

- **Condition codes**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

**Overview Diagram**
- CPU
- Registers
- Condition Codes
- PC
- Memory
  - Code
  - Data
  - Stack
- Addresses
  - Data
  - Instructions
Turning C into Object Code

- Code in files `p1.c` `p2.c`
- Compile with command: `gcc -O1 p1.c p2.c -o p`
  - Use basic optimizations (`-O1`)
  - Put resulting binary in file `p`

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

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

  Object program (p1.o p2.o)
  Linker (gcc or ld)

  Executable program (p)
```

Static libraries (.a)
Compiling Into Assembly

C Code (p1.c)

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

Generated IA32 Assembly

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

Obtain (on shark machine) with command

```
gcc -O1 -m32 -S p1.c
```

Produces file p1.s

**Warning**: Will get very different results on non-Shark machines (Andrew Linux, Mac OS-X, ...) due to different versions of gcc and different compiler settings.
Assembly Characteristics: 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
Assembly Characteristics: 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

Code for `sum`

0x08483f4 <sum>:
- 0x55
- 0x89
- 0xe5
- 0x8b
- 0x45
- 0x0c
- 0x03
- 0x45
- 0x08
- 0x5d
- 0xc3

- Total of 11 bytes
- Each instruction 1, 2, or 3 bytes
- Starts at address 0x08483f4

- Assembler
  - Translates .s into .o
  - Binary encoding of each instruction
  - 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
    - Linking occurs when program begins execution
Machine Instruction Example

**C Code**
- Add two signed integers

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

**Assembly**
- Add 2 4-byte integers
  - “Long” words in GCC parlance
  - Same instruction whether signed or unsigned
- Operands:
  - `x`: Register `%eax`
  - `y`: Memory `M[%ebp+8]`
  - `t`: Register `%eax`
  - Return function value in `%eax`

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

Similar to expression:
- `x += y`

More precisely:
- `int eax;`
- `int *ebp;`
- `eax += ebp[2]`

**Object Code**
- 3-byte instruction
- Stored at address `0x80483fa`

```assembly
0x80483fa: 03 45 08
```
Disassembling Object Code

Disassembled

080483f4 <sum>:
  80483f4:  55  push  %ebp
  80483f5:  89 e5  mov  %esp,%ebp
  80483f7:  8b 45 0c  mov  0xc(%ebp),%eax
  80483fa:  03 45 08  add  0x8(%ebp),%eax
  80483fd:  5d  pop  %ebp
  80483fe:  c3  ret

- Disassembler

  objdump -d p

  - 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

Object

Disassembled

Dump of assembler code for function sum:
0x080483f4 <sum+0>:     push   %ebp
0x080483f5 <sum+1>:     mov    %esp,%ebp
0x080483f7 <sum+3>:     mov    0xc(%ebp),%eax
0x080483fa <sum+6>:     add    0x8(%ebp),%eax
0x080483fd <sum+9>:     pop     %ebp
0x080483fe <sum+10>:    ret

■ Within gdb Debugger

    gdb p
disassemble sum
    ▪ Disassemble procedure
    x/11xb sum
    ▪ Examine the 11 bytes starting at sum
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>:
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
Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Intro to x86-64
## Integer Registers (IA32)

<table>
<thead>
<tr>
<th>Register</th>
<th>General Purpose</th>
<th>16-bit Virtual Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>%ax</td>
<td>%ah, %al</td>
</tr>
<tr>
<td>%ecx</td>
<td>%cx</td>
<td>%ch, %cl</td>
</tr>
<tr>
<td>%edx</td>
<td>%dx</td>
<td>%dh, %dl</td>
</tr>
<tr>
<td>%ebx</td>
<td>%bx</td>
<td>%bh, %bl</td>
</tr>
<tr>
<td>%esi</td>
<td>%si</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td>%di</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td>%sp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>%bp</td>
<td></td>
</tr>
</tbody>
</table>

### Origin (mostly obsolete)
- accumulate
- counter
- data
- base
- source
- index
- destination
- index
- stack
- pointer
- base
- pointer

16-bit virtual registers (backwards compatibility)
## Moving Data: IA32

### Moving Data

\texttt{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”

<table>
<thead>
<tr>
<th>%eax</th>
<th>%ecx</th>
<th>%edx</th>
<th>%ebx</th>
<th>%esi</th>
<th>%edi</th>
<th>%esp</th>
<th>%ebp</th>
</tr>
</thead>
</table>
### movl Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src,Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>movl</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl $-147, (%eax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl (%eax), (%edx)</td>
<td>*p = temp;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl (%eax), (%edx)</td>
<td>*p = temp;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl (%eax), (%edx)</td>
<td>*p = temp;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl (%eax), (%edx)</td>
<td>*p = temp;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl (%eax), (%edx)</td>
<td>*p = temp;</td>
</tr>
</tbody>
</table>

Cannot do memory-memory transfer with a single instruction
Simple Memory Addressing Modes

- **Normal** (R) \text{Mem[Reg[R]]}
  - Register R specifies memory address
  - Aha! Pointer dereferencing in C

  \text{movl (\%ecx),\%eax}

- **Displacement** \text{D(R)} \text{Mem[Reg[R]+D]}
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

  \text{movl 8(\%ebp),\%edx}
Example of Simple Addressing Modes

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

**swap:**

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

**Set Up**

**Body**

**Finish**
Using Simple Addressing Modes

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

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

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

popl %ebx
popl %ebp
ret
```

- **Set Up**
- **Body**
- **Finish**
# Understanding Swap()

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>xp</td>
</tr>
<tr>
<td>%eax</td>
<td>yp</td>
</tr>
<tr>
<td>%ecx</td>
<td>t0</td>
</tr>
<tr>
<td>%ebx</td>
<td>t1</td>
</tr>
</tbody>
</table>

```

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

<table>
<thead>
<tr>
<th>%eax</th>
<th>%edx</th>
<th>%ecx</th>
<th>%ebx</th>
<th>%esi</th>
<th>%edi</th>
<th>%esp</th>
<th>%ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x104</td>
<td></td>
</tr>
</tbody>
</table>


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

- **%eax**
- **%edx** 0x124
- **%ecx**
- **%ebx**
- **%esi**
- **%edi**
- **%esp**
- **%ebp** 0x104

### Registers

- edx: 0x124
- %eax
- %ecx
- %ebx
- %esi
- %edi
- %esp
- %ebp

### Addresses

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0x120</td>
</tr>
<tr>
<td>8</td>
<td>0x124</td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>0</td>
<td>0x104</td>
</tr>
<tr>
<td>-4</td>
<td>0x100</td>
</tr>
</tbody>
</table>

### MOV instructions

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td>0x120</td>
<td>0x110</td>
</tr>
<tr>
<td>0x120</td>
<td>0x124</td>
<td>0x10c</td>
</tr>
<tr>
<td>0x118</td>
<td>0x114</td>
<td>0x108</td>
</tr>
<tr>
<td>0x114</td>
<td>Rtn adr</td>
<td>0x104</td>
</tr>
<tr>
<td>0x100</td>
<td></td>
<td>0x100</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>%eax</th>
<th>0x120</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>123</td>
</tr>
<tr>
<td>%ebx</td>
<td></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 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %eax  # eax = yp
movl (%edx), %ecx  # ecx = *xp (t0)
movl (%eax), %ebx  # ebx = *yp (t1)
movl %ebx, (%edx)  # *xp = t1
movl %ecx, (%eax)  # *yp = t0
Understanding Swap()

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0x120</td>
</tr>
<tr>
<td>8</td>
<td>0x124</td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>0</td>
<td>0x108</td>
</tr>
<tr>
<td>-4</td>
<td>0x100</td>
</tr>
</tbody>
</table>

%eax | 0x120
%edx | 0x124
%ecx | 123
%ebx | 456
%esi |
%edi |
%esp |
%ebp | 0x104

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
<th>yp</th>
<th>xp</th>
<th>Rtn adr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td>12</td>
<td>0x120</td>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x120</td>
<td>8</td>
<td>0x124</td>
<td>0x10c</td>
<td></td>
</tr>
<tr>
<td>0x118</td>
<td>4</td>
<td>0x114</td>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x114</td>
<td>0</td>
<td>0x110</td>
<td>0x104</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td>-4</td>
<td>0x104</td>
<td>0x100</td>
<td></td>
</tr>
</tbody>
</table>

| %eax    | 0x120  |
| %edx    | 0x124  |
| %ecx    | 123    |
| %ebx    | 456    |
| %esi    |        |
| %edi    |        |
| %esp    |        |
| %ebp    | 0x104  |

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

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td>4</td>
<td>456</td>
</tr>
<tr>
<td>0x120</td>
<td>12</td>
<td>123</td>
</tr>
<tr>
<td>0x118</td>
<td>8</td>
<td>0x124</td>
</tr>
<tr>
<td>0x114</td>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>0x108</td>
<td>0</td>
<td>%ebp</td>
</tr>
<tr>
<td>0x104</td>
<td>-4</td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %eax  # eax = yp
movl (%edx), %ecx    # ecx = *xp (t0)
movl (%eax), %ebx    # ebx = *yp (t1)
movl %ebx, (%edx)    # *xp = t1
movl %ecx, (%eax)    # *yp = t0
```
Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Intro to x86-64
# 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>10/16</td>
</tr>
<tr>
<td>char *</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

- Or any other pointer
### x86-64 Integer Registers

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
<th>%r8</th>
<th>%r8d</th>
</tr>
</thead>
<tbody>
<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>

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

- Long word 1 (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;
}
```

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

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

    popl %ebx
    popl %ebp
    ret
```
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 %edx and %eax
  - movl operation
64-bit code for long int swap

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

**64-bit data**
- Data held in registers `%rdx` and `%rax`
- `movq` operation
  - “q” stands for quad-word

```assembly
swap_l:
    movq (%rdi), %rax
    movq (%rsi), %rdx
    movq %rdx, (%rdi)
    movq %rax, (%rsi)
    ret
```
Machine Programming I: Summary

- History of Intel processors and architectures
  - Evolutionary design leads to many quirks and artifacts

- C, assembly, machine code
  - Compiler must transform statements, expressions, procedures into low-level instruction sequences

- Assembly Basics: Registers, operands, move
  - The x86 move instructions cover wide range of data movement forms

- Intro to x86-64
  - A major departure from the style of code seen in IA32
Complete Memory Addressing Modes

**Most General Form**

$$D(R_b, R_i, S) \quad \text{Mem}[\text{Reg}[R_b]+S*\text{Reg}[R_i]+D]$$

- **D:** Constant “displacement” 1, 2, or 4 bytes
- **R_b:** Base register: Any of 8 integer registers
- **R_i:** Index register: Any, except for %esp
  - Unlikely you’d use %ebp, either
- **S:** Scale: 1, 2, 4, or 8 (*why these numbers?*)

**Special Cases**

- $$(R_b, R_i) \quad \text{Mem}[\text{Reg}[R_b]+\text{Reg}[R_i]]$$
- $$D(R_b, R_i) \quad \text{Mem}[\text{Reg}[R_b]+\text{Reg}[R_i]+D]$$
- $$(R_b, R_i, S) \quad \text{Mem}[\text{Reg}[R_b]+S*\text{Reg}[R_i]]$$