Machine-Level Programming I: Basics

15-213/18-213: Introduction to Computer Systems
5th Lecture, Sep. 13, 2011

Instructors:
Dave O’Hallaron, Greg Ganger, and Greg Kesden
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

- Totally 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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>386</td>
<td>1985</td>
<td>275K</td>
<td>16-33</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 i7</td>
<td>2008</td>
<td>731M</td>
<td>2667-3333</td>
</tr>
</tbody>
</table>

- **8086** (1978):
  - First 16-bit processor. Basis for IBM PC & DOS
  - 1MB address space

- **386** (1985):
  - First 32 bit processor, referred to as IA32
  - Added “flat addressing”
  - Capable of running Unix
  - 32-bit Linux/gcc uses no instructions introduced in later models

- **Pentium 4F** (2004):
  - First 64-bit processor, referred to as x86-64

- **Core i7** (2008):
  - Our shark machines
# Intel x86 Processors: Overview

<table>
<thead>
<tr>
<th>Architectures</th>
<th>Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>X86-16</td>
<td>8086</td>
</tr>
<tr>
<td></td>
<td>286</td>
</tr>
<tr>
<td>X86-32/IA32</td>
<td>386</td>
</tr>
<tr>
<td></td>
<td>486</td>
</tr>
<tr>
<td>MMX</td>
<td>Pentium</td>
</tr>
<tr>
<td>SSE</td>
<td>Pentium MMX</td>
</tr>
<tr>
<td>SSE2</td>
<td>Pentium III</td>
</tr>
<tr>
<td>SSE3</td>
<td>Pentium 4</td>
</tr>
<tr>
<td>X86-64 / EM64t</td>
<td>Pentium 4E</td>
</tr>
<tr>
<td>SSE4</td>
<td>Pentium 4F</td>
</tr>
<tr>
<td></td>
<td>Core 2 Duo</td>
</tr>
<tr>
<td></td>
<td>Core i7</td>
</tr>
</tbody>
</table>

IA: often redefined as latest Intel architecture
Intel x86 Processors, contd.

**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
  - Parallel operations on 1, 2, and 4-byte data, both integer & FP
  - Instructions to enable more efficient conditional operations

**Linux/GCC Evolution**
- Two major steps: 1) support 32-bit 386. 2) support 64-bit x86-64
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
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)**: x86, IA
Assembly Programmer’s View

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 operation
  - Used for conditional branching

- **Memory**
  - Byte addressable array
  - Code and user data
  - Stack to support procedures
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`
Compiling Into 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
popl %ebp
ret
```

Some compilers use instruction “leave”

Obtain with command

```
/usr/local/bin/gcc -O1 -S code.c
```

Produces file `code.s`
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`

0x401040  <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 0x401040

**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

- **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

- **Object Code**
  - 3-byte instruction
  - Stored at address \texttt{0x80483ca}
Disassembling Object Code

Disassembled

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>080483c4</td>
<td>push %ebp</td>
<td>55</td>
</tr>
<tr>
<td>080483c5</td>
<td>mov %esp,%ebp</td>
<td>89 e5</td>
</tr>
<tr>
<td>080483c7</td>
<td>mov 0xc(%ebp),%eax</td>
<td>8b 45 0c</td>
</tr>
<tr>
<td>080483ca</td>
<td>add 0x8(%ebp),%eax</td>
<td>03 45 08</td>
</tr>
<tr>
<td>080483cd</td>
<td>pop %ebp</td>
<td>5d</td>
</tr>
<tr>
<td>080483ce</td>
<td>ret</td>
<td>c3</td>
</tr>
</tbody>
</table>

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

| 0x401040: | 0x55  |
|          | 0x89  |
|          | 0xe5  |
|          | 0x8b  |
|          | 0x45  |
|          | 0x0c  |
|          | 0x03  |
|          | 0x45  |
|          | 0x08  |
|          | 0x5d  |
|          | 0xc3  |

Disassembled

Dump of assembler code for function sum:

```
0x080483c4 <sum+0>: push %ebp
0x080483c5 <sum+1>: mov %esp,%ebp
0x080483c7 <sum+3>: mov 0xc(%ebp),%eax
0x080483ca <sum+6>: add 0x8(%ebp),%eax
0x080483cd <sum+9>: pop %ebp
0x080483ce <sum+10>: ret
```

■ Within gdb Debugger

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

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

**General Purpose**
- Origin: mostly obsolete
- 16-bit virtual registers (backwards compatibility)

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

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

- **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
  ```
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), %ecx
    movl (%edx), %ebx
    movl (%ecx), %eax
    movl %eax, (%edx)
    movl %ebx, (%ecx)

    popl %ebx
    popl %ebp
    ret
```
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), %ecx
    movl (%edx), %ebx
    movl (%ecx), %eax
    movl %eax, (%edx)
    movl %ebx, (%ecx)
    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>%ecx</td>
<td>yp</td>
</tr>
<tr>
<td>%ebx</td>
<td>t0</td>
</tr>
<tr>
<td>%eax</td>
<td>t1</td>
</tr>
</tbody>
</table>

**Stack (in memory)**

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

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

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

<table>
<thead>
<tr>
<th>%ebp</th>
<th>0x104</th>
</tr>
</thead>
<tbody>
<tr>
<td>%esp</td>
<td>0x100</td>
</tr>
<tr>
<td>Rtn adr</td>
<td>0x108</td>
</tr>
<tr>
<td>xp</td>
<td>0x124</td>
</tr>
<tr>
<td>yp</td>
<td>0x120</td>
</tr>
</tbody>
</table>

Offset

| yp | 12 |
| xp | 8 |
| Rtn adr | 0 |

Address

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
</tr>
<tr>
<td>0x124</td>
</tr>
<tr>
<td>0x118</td>
</tr>
<tr>
<td>0x11c</td>
</tr>
<tr>
<td>0x110</td>
</tr>
<tr>
<td>0x10c</td>
</tr>
</tbody>
</table>

movl 8(%ebp), %edx # edx = xp
movl 12(%ebp), %ecx # ecx = yp
movl (%edx), %ebx # ebx = *xp (t0)
movl (%ecx), %eax # eax = *yp (t1)
movl %eax, (%edx) # *xp = t1
movl %ebx, (%ecx) # *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>0x124</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x104</td>
<td>0x100</td>
</tr>
</tbody>
</table>

```
  movl 8(%ebp), %edx  # edx = xp
  movl 12(%ebp), %ecx # ecx = yp
  movl (%edx), %ebx   # ebx = *xp (t0)
  movl (%ecx), %eax   # eax = *yp (t1)
  movl %eax, (%edx)   # *xp = t1
  movl %ebx, (%ecx)   # *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>0x120</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>0x104</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td></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>

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

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

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

<table>
<thead>
<tr>
<th>Register</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>

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

<table>
<thead>
<tr>
<th>Register</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>

## Variables

<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>

## Instructions

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

**Most General Form**

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

- **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 (*why these numbers?*)

**Special Cases**

- \((Rb,Ri)\) \quad \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]]
- \(D(Rb,Ri)\) \quad \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]+D]
- \((Rb,Ri,S)\) \quad \text{Mem}[\text{Reg}[Rb]+S^*\text{Reg}[Ri]]
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>16</td>
</tr>
<tr>
<td>char *</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>
## 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
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), %ecx
    movl (%edx), %ebx
    movl (%ecx), %eax
    movl %eax, (%edx)
    movl %ebx, (%ecx)
    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 `%eax` and `%edx`
- `movl` operation
64-bit code for long int swap

swap_l:

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

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

Set Up

Body

Finish
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