Some slides adapted (and slightly modified) from:
• Clark Barrett
• Jinyang Li
• Randy Bryant
• Dave O’Hallaron
Intel x86 Processors

• Evolutionary design
  – Backwards compatible up until 8086, introduced in 1978

• Complex instruction set computer (CISC)
  – Many instructions, many formats
  – By contrast, ARM architecture (in most cell phones) is RISC
# Intel x86 Evolution: Milestones

<table>
<thead>
<tr>
<th>Name</th>
<th>Transistors</th>
<th>MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086 (1978)</td>
<td>29K</td>
<td>5-10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>First 16-bit processor. Basis for IBM PC &amp; DOS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1MB address space</td>
<td></td>
</tr>
<tr>
<td>386 (1985)</td>
<td>275K</td>
<td>16-33</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>First 32-bit processor, referred to as <strong>IA32</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Capable of running Unix</td>
<td></td>
</tr>
<tr>
<td>Pentium 4F (2004)</td>
<td>125M</td>
<td>2800-3800</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>First 64-bit processor, referred to as <strong>x86-64</strong></td>
<td></td>
</tr>
<tr>
<td>Core i7 (2008)</td>
<td>731M</td>
<td>2667-3333</td>
</tr>
<tr>
<td>Xeon E7 (2011)</td>
<td>2.2B</td>
<td>~2400</td>
</tr>
</tbody>
</table>

We cover both IA32 and x86-64. Labs are done in IA32.
Assembly Programmer’s View

- **Execution context**
  - **PC**: Program counter
    - Address of next instruction
    - Called “EIP” (IA32) or “RIP” (x86-64)
  - **Registers**
    - Heavily used program data
  - **Condition codes**
    - Info of recent arithmetic operation
    - Used for conditional branching
Assembly Data Types

• “Integer” data of 1, 2, or 4 bytes
  – Represent either data value
  – or address (untyped pointer)

• Floating point data of 4, 8, or 10 bytes

• No arrays or structures
3 Kind of Assembly Operations

• Perform arithmetic on register or memory data
  – Add, subtract, multiplication...

• 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
Turning C into Object Code

- Code in files p1.c p2.c
- Compile with command: gcc -O1 p1.c p2.c -o p

Text

C program (p1.c p2.c)

Compiler (gcc -S)

Asm program (p1.s p2.s)

Assembler (gcc -c)

Object program (p1.o p2.o)

Linker

Executable program (p)

Static libraries (.a)
Compiling Into Assembly

**sum.c**

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

**sum.s**

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

```assembly
80483c4:  55 89 e5 8b 45 0c 03 45 08 5d c3
```

Note: If your platform is 64-bit, you may want to force it to generate 32-bit assembly by `gcc -m32 -S sum.c` to get the above output.
Compiling Into Assembly

sum.c

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

sum.s

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

Refer to register %eax
Refer to memory at address %ebp+8

sum.o

```
80483c4:  55 89 e5 8b 45 0c 03 45 08 5d c3
```
## Integer Registers (IA32)

### Origin
- (mostly obsolete)

### General Purpose
- %eax
- %ecx
- %edx
- %ebx
- %esi
- %edi

### 16-bit Virtual Registers
- %esp
- %ebp

### Virtual Registers
- %ax
- %cx
- %dx
- %bx
- %si
- %di

### Accumulate Counter
- %ah
- %ch
- %dh
- %bh

### Data Base Source Index Destination Stack Pointer Base Pointer
- %al
- %cl
- %dl
- %bl
Moving Data: IA32

• `movl Source, Dest`

• Operand Types
  – **Immediate**: Integer constant
    • e.g. $0x400
  – **Register**: One of 8 integer registers
    • e.g. `%eax`
  – **Memory**: 4 consecutive bytes of memory at address given by register
    • Simplest example (%eax)
### 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>Imm</strong></td>
<td><strong>Reg</strong></td>
<td>movl $0x4, %eax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>movl $-147, (%eax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td><strong>Reg</strong></td>
<td><strong>Reg</strong></td>
<td>movl %eax, %edx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td></td>
<td>Mem</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>

**No memory-to-memory instruction**
Memory Addressing Modes

• Normal (R) \( \text{Mem}[\text{Reg}[R]] \)
  - Register R specifies memory address

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

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

  \text{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
```
Understanding Swap

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 0x124</th>
</tr>
</thead>
<tbody>
<tr>
<td>ecx</td>
<td>ebx</td>
</tr>
<tr>
<td>esi</td>
<td>edi</td>
</tr>
<tr>
<td>esp</td>
<td>ebp 0x104</td>
</tr>
</tbody>
</table>

Offset

- yp 12 0x120 0x110
- xp  8 0x124 0x10c
- ebp  4
- %ebp 0
- -4

Address

- 123 0x124
- 456 0x120
- 114 0x11c
- 118 0x118
- 108 0x10c
- 104 0x108
- 100 0x100

```
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

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>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x124</td>
</tr>
<tr>
<td>%edx</td>
<td>0x120</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>Address</th>
<th>Offset</th>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td>12</td>
<td>yp</td>
<td>0x120</td>
</tr>
<tr>
<td>0x120</td>
<td>8</td>
<td>xp</td>
<td>0x124</td>
</tr>
<tr>
<td>0x114</td>
<td>4</td>
<td>Rtn adr</td>
<td>0x10c</td>
</tr>
<tr>
<td>0x104</td>
<td>0</td>
<td>%ebp</td>
<td>0x100</td>
</tr>
<tr>
<td>0x100</td>
<td>-4</td>
<td></td>
<td></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>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
</tr>
<tr>
<td>0x120</td>
</tr>
<tr>
<td>0x11c</td>
</tr>
<tr>
<td>0x118</td>
</tr>
<tr>
<td>0x114</td>
</tr>
<tr>
<td>0x108</td>
</tr>
<tr>
<td>0x104</td>
</tr>
<tr>
<td>0x100</td>
</tr>
</tbody>
</table>

### Offset

| yp    | 12 | 0x120 | 0x110 |
| xp    | 8  | 0x124 | 0x10c |
|       | 4  |       |       |
| %ebp  | 0  |       |       |
|       | -4 |       |       |

### Assembly Code

```
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>Address</th>
<th>Offset</th>
<th>Rtn adr</th>
<th>yp</th>
<th>xp</th>
<th>%ebp</th>
<th>456</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td>12</td>
<td>0x120</td>
<td></td>
<td></td>
<td>0x104</td>
<td>456</td>
</tr>
<tr>
<td>0x120</td>
<td>8</td>
<td>0x124</td>
<td></td>
<td></td>
<td>0x108</td>
<td>123</td>
</tr>
<tr>
<td>0x11c</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>0x104</td>
<td>0x124</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x108</td>
<td>0x11c</td>
</tr>
<tr>
<td>0x114</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x108</td>
<td>0x118</td>
</tr>
<tr>
<td>0x10c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x108</td>
<td>0x114</td>
</tr>
<tr>
<td>0x108</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x108</td>
<td>0x10c</td>
</tr>
<tr>
<td>0x104</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x108</td>
<td>0x108</td>
</tr>
<tr>
<td>0x100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x108</td>
<td>0x104</td>
</tr>
</tbody>
</table>

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

```c
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
```
General Memory Addressing Modes

• Most General Form

\[ D( Rb, Ri, S ) \]

Base register \hspace{1cm} Index register \hspace{1cm} Scale \hspace{1cm} Constant displacement

(no \%esp) \hspace{1cm} \hspace{1cm} (1,2,4,8) \hspace{1cm} \text{displacement}

\[ \text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]+D] \]

• Special Cases

\( (Rb,Ri) \) \hspace{1cm} \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]]

\( D(Rb,Ri) \) \hspace{1cm} \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]+D]

\( (Rb,Ri,S) \) \hspace{1cm} \text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]]

Constant displacement: (1,2,4,8)
Size of C objects on IA32 and x86-64

<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>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>Extent existing registers</th>
<th>Add 8 new ones</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rax</code></td>
<td><code>%eax</code></td>
</tr>
<tr>
<td><code>%rbx</code></td>
<td><code>%ebx</code></td>
</tr>
<tr>
<td><code>%rcx</code></td>
<td><code>%ecx</code></td>
</tr>
<tr>
<td><code>%rdx</code></td>
<td><code>%edx</code></td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td><code>%esi</code></td>
</tr>
<tr>
<td><code>%rdi</code></td>
<td><code>%edi</code></td>
</tr>
<tr>
<td><code>%rsp</code></td>
<td><code>%esp</code></td>
</tr>
<tr>
<td><code>%rbp</code></td>
<td><code>%ebp</code></td>
</tr>
</tbody>
</table>
Instructions

• New instructions for 8-byte types:
  – movl → movq
  – addl → addq
  – sall → salq
  – etc.
64-bit code for swap

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

swap:
...
movl (%rdi), %edx
movl (%rsi), %eax
movl %eax, (%rdi)
movl %edx, (%rsi)
...
ret

• Arguments passed in registers
  – First (xp) in %rdi, second (yp) in %rsi
  – Why hold data in %eax and %edx instead of %rax %rdx?
  – Why movl operation instead of movq?
64-bit code for long int swap

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

```
swap_l:
    movq (%rdi), %rdx
    movq (%rsi), %rax
    movq %rax, (%rdi)
    movq %rdx, (%rsi)
    ret
```
Conclusions

- 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