Many of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook “Computer Systems: A Programmer’s Perspective,” 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O’Hallaron in Fall 2010. These slides are indicated “Supplied by CMU” in the notes section of the slides.
Data Types on Intel x86

• “Integer” data of 1, 2, or 4 bytes (plus 8 bytes on x86-64)
  – data values
    » whether signed or unsigned depends on interpretation
  – 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
Most instructions come in three (on IA32) or four (on x86-64) forms, one for each possible operand size.
General-Purpose Registers (IA32)

<table>
<thead>
<tr>
<th>%eax</th>
<th>%ax</th>
<th>%ah</th>
<th>%al</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>%cx</td>
<td>%ch</td>
<td>%cl</td>
</tr>
<tr>
<td>%edx</td>
<td>%dx</td>
<td>%dh</td>
<td>%dl</td>
</tr>
<tr>
<td>%ebx</td>
<td>%bx</td>
<td>%bh</td>
<td>%bl</td>
</tr>
<tr>
<td>%esi</td>
<td>%si</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td>%di</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td>%sp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>%bp</td>
<td></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)

Supplied by CMU.
Supplied by CMU.

Note that though esp and ebp have special uses, they may also be used in both source and destination operands.

Note that some assemblers (in particular, those of Intel and Microsoft) place the operands in the opposite order. Thus the example of the slide would be “addl %eax, 8(%ebp)”. The order we use is that used by gcc, known as the “AT&T syntax” because it was used in the original Unix assemblers, written at Bell Labs, then part of AT&T.
## 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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Imm</td>
<td>Reg</td>
<td>movl $0x4,%eax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl $-147,(%eax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td>Mem</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 (normally) do memory-memory transfer with a single instruction*
Supplied by CMU.

If one thinks of there being an array of registers, then “Reg[R]” selects register “R” from this array.
Using Simple Addressing Modes

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

```
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
```

Supplied by CMU.
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>

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

Supplied by CMU.
Supplied by CMU.
Supplied by CMU.
Supplied by CMU.
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
```

Supplied by CMU.
## Understanding Swap

<table>
<thead>
<tr>
<th>%eax</th>
<th>456</th>
</tr>
</thead>
<tbody>
<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>

### Address
- 0x124
- 0x120
- 0x11c
- 0x118
- 0x114
- 0x110
- 0x10c
- 0x108
- 0x104
- 0x100

### Offset
- YP
  - 12
  - 0x120
  - 0x110
- xp
  - 8
  - 0x124
  - 0x10c
- %ebp
  - 0
  - 0x108
  - 0x104
  - 0x100

---

```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
```

---

Supplied by CMU.
Supplied by CMU.
Understanding Swap

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1124</td>
<td>12</td>
</tr>
<tr>
<td>0x120</td>
<td>8</td>
</tr>
<tr>
<td>0x11c</td>
<td>4</td>
</tr>
<tr>
<td>0x118</td>
<td>0</td>
</tr>
<tr>
<td>0x114</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x10c</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x104</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{movl } 8(\%ebp), \%edx \quad \# \text{ edx } = \text{ xp} \\
\text{movl } 12(\%ebp), \%ecx \quad \# \text{ ecx } = \text{ yp} \\
\text{movl } (\%edx), \%ebx \quad \# \text{ ebx } = *\text{ xp} \ (t0) \\
\text{movl } (\%ecx), \%eax \quad \# \text{ eax } = *\text{ yp} \ (t1) \\
\text{movl } %eax, (\%edx) \quad \# *\text{ xp } = t1 \\
\text{movl } %ebx, (\%ecx) \quad \# *\text{ yp } = t0
\]

Supplied by CMU.
Quiz 1

```
movl -4(%ebp), %eax
movl (%eax), %eax
movl (%eax), %eax
movl %eax, -8(%ebp)
```

Which C statements best describe the assembler code?

```
// a
int x;
int y;
y = x;

// b
int *x;
int y;
y = *x;

// c
int **x;
int y;
y = **x;

// d
int ***x;
int y;
y = ***x;
```
Complete Memory-Addressing Modes

- Most general form
  \[ D(Rb, Ri, S) \quad \text{Mem}[\text{Reg}[Rb]+S\times\text{Reg}[Ri]+D] \]
  - D: constant “displacement”
  - 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
  \begin{align*}
  & (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\times\text{Reg}[Ri]] \\
  & D \quad \text{Mem}[D]
  \end{align*}
### Address-Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address 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>$0xf000 + 0x0100$</td>
<td>$0xf100$</td>
</tr>
<tr>
<td>$(%edx, %ecx, 4)$</td>
<td>$0xf000 + 4*0x0100$</td>
<td>$0xf400$</td>
</tr>
<tr>
<td>$0x80(, %edx, 2)$</td>
<td>$2*0xf000 + 0x80$</td>
<td>$0x1e080$</td>
</tr>
</tbody>
</table>
Supplied by CMU.

Note that a function returns a value by putting it in %eax.
Quiz 2

What value ends up in %ecx?

movl $1000, %eax
movl $1, %ebx
movl 2(%eax, %ebx, 4), %ecx

a) 0x02030405
b) 0x05040302
c) 0x06070809
d) 0x09080706

Hint: %eax → 0x00

1009: 0x09
1008: 0x08
1007: 0x07
1006: 0x06
1005: 0x05
1004: 0x04
1003: 0x03
1002: 0x02
1001: 0x01
1000: 0x00
Note that `%ebp/%rbp` may be used as a base register as on IA32, but they don't have to be used that way. This will become clearer when we explore how the runtime stack is accessed. The convention on Linux is for the first 6 arguments of a function to be in registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9`. The return value of a function is put in `%rax`.
On x86-64, for instructions with 32-bit (long) operands that produce 32-bit results going into a register, the register must be a 32-bit register; the higher-order 32 bits are filled with zeroes.
Supplied by CMU.

Note that for the IA32 architecture, arguments are passed on the stack.
64-bit code for swap

```c
void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
    movl (%rdi), %edx
    movl (%rsi), %eax
    movl %eax, (%rdi)
    movl %edx, (%rsi)
    ret
}
```

- Arguments 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

Supplied by CMU.

Note that no more than six arguments can be passed in registers. If there are more than six arguments (which is unusual), then remaining arguments are passed on the stack, and referenced via %rsp.
64-bit code for long int swap

```
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

Supplied by CMU.
Note that normally one does not ask gcc to produce assembler code, but instead it compiles C code directly into machine code (producing an object file). Note also that the gcc command actually invokes a script; the compiler (also known as gcc) compiles code into either assembler code or machine code; if necessary, the assembler (as) assembles assembler code into object code. The linker (ld) links together multiple object files (containing object code) into an executable program.
### Object Code

#### Code for sum

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040</td>
<td><code>&lt;sum&gt;</code>:</td>
</tr>
<tr>
<td>0x55</td>
<td></td>
</tr>
<tr>
<td>0x89</td>
<td></td>
</tr>
<tr>
<td>0xe5</td>
<td></td>
</tr>
<tr>
<td>0x8b</td>
<td></td>
</tr>
<tr>
<td>0x45</td>
<td></td>
</tr>
<tr>
<td>0xc0</td>
<td></td>
</tr>
<tr>
<td>0x03</td>
<td></td>
</tr>
<tr>
<td>0x45</td>
<td></td>
</tr>
<tr>
<td>0x08</td>
<td></td>
</tr>
<tr>
<td>0x5d</td>
<td></td>
</tr>
<tr>
<td>0xc3</td>
<td></td>
</tr>
</tbody>
</table>

- **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 printf`
  - some libraries are *dynamically linked*
    - linking occurs when program begins execution

---

Supplied by CMU.
Instruction Format

This is taken from Intel Architecture Software Developer’s Manual, Volume 2: Instruction Set Reference; Order Number 243191, Intel Corporation, 1999.
Disassembling Object Code

Disassembled

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>080483c4</td>
<td>&lt;sum&gt;</td>
<td></td>
</tr>
<tr>
<td>80483c4:</td>
<td>55</td>
<td>push</td>
</tr>
<tr>
<td>80483c5:</td>
<td>89 e5</td>
<td>mov</td>
</tr>
<tr>
<td>80483c7:</td>
<td>8b 45 0c</td>
<td>mov 0xc(%ebp),%eax</td>
</tr>
<tr>
<td>80483ca:</td>
<td>03 45 08</td>
<td>add 0x8(%ebp),%eax</td>
</tr>
<tr>
<td>80483cd:</td>
<td>5d</td>
<td>pop</td>
</tr>
<tr>
<td>80483ce:</td>
<td>c3</td>
<td>ret</td>
</tr>
</tbody>
</table>

- Disassembler
  - `objdump -d <file>`
  - useful tool for examining object code
  - analyzes bit pattern of series of instructions
  - produces approximate rendition of assembly code
  - can be run on either executable or object (.o) file

Supplied by CMU.
**Alternate Disassembly**

### 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 <file>`
  
  `disassemble sum`
  
  `- disassemble procedure`
  
  `x/11xb sum`
  
  `- examine the 11 bytes starting at sum`

---

Supplied by CMU.
How Many Instructions are There?

- We cover ~29
- Implemented by Intel:
  - 80 in original 8086 architecture
  - 7 added with 80186
  - 17 added with 80286
  - 33 added with 386
  - 6 added with 486
  - 6 added with Pentium
  - 1 added with Pentium MMX
  - 4 added with Pentium Pro
  - 8 added with SSE
  - 8 added with SSE2
  - 2 added with SSE3
  - 14 added with x86-64
  - 10 added with VT-x
  - 2 added with SSE4a
- Total: 198
- Doesn’t count:
  - floating-point instructions
  - SIMD instructions
  - AMD-added instructions
  - undocumented instructions

The source for this is http://en.wikipedia.org/wiki/X86_instruction_listings, viewed on 9/18/2012, and also depends upon my ability to count.
**Some Arithmetic Operations**

- **Two-operand instructions:**

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>add1</td>
<td>$Dest = Dest + Src$</td>
<td></td>
</tr>
<tr>
<td>sub1</td>
<td>$Dest = Dest - Src$</td>
<td></td>
</tr>
<tr>
<td>imull</td>
<td>$Dest = Dest \times Src$</td>
<td></td>
</tr>
<tr>
<td>sall</td>
<td>$Dest = Dest \ll Src$</td>
<td>Also called shll</td>
</tr>
<tr>
<td>sarl</td>
<td>$Dest = Dest \gg Src$</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>shrl</td>
<td>$Dest = Dest \gg Src$</td>
<td>Logical</td>
</tr>
<tr>
<td>xorl</td>
<td>$Dest = Dest \oplus Src$</td>
<td></td>
</tr>
<tr>
<td>andl</td>
<td>$Dest = Dest &amp; Src$</td>
<td></td>
</tr>
<tr>
<td>orl</td>
<td>$Dest = Dest \lor Src$</td>
<td></td>
</tr>
</tbody>
</table>

- watch out for argument order!
- no distinction between signed and unsigned int (why?)
Some Arithmetic Operations

- **One-operand Instructions**
  
  - incl Dest = Dest + 1
  - dec1 Dest = Dest - 1
  - neg1 Dest = - Dest
  - not1 Dest = ~Dest

- **See book for more instructions**

Supplied by CMU.
Arithmetic Expression Example

```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;
}
```

```
arith:
leal (%rdi, %rsi), %eax
addl %edx, %eax
leal (%rsi, %rsi, 2), %edx
 sal $4, %edx
 leal 4(%rdi, %rdx), %ecx
imull %ecx, %eax
ret
```
Supplied by CMU, but converted to x86-64.
Supplied by CMU, but converted to x86-64.

By convention, the first three arguments to a procedure are placed in registers rdi, rsi, and rdx, respectively. Note that, also by convention, procedures put their return values in register eax/rax.
Observations about 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 might require multiple instructions
- Some instructions might cover multiple expressions

```
leal (%rdi,%rsi), %eax  # eax = x+y   (t1)
addl %edx, %eax         # eax = t1+z   (t2)
leal (%rsi,%rsi,2), %edx # edx = 3*y   (t4)
sall $4, %edx            # edx = t4*16 (t4)
leal 4(%rdi,%rdx), %ecx # ecx = x+4+t4 (t5)
imull %ecx, %eax        # eax *= t5   (rval)
ret
```

Supplied by CMU, but converted to x86-64.
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;
}
```

$2^{13} = 8192, 2^{13} - 7 = 8185$

- `xorl %esi, %edi` # edi = $x^y$ (t1)
- `sarl $17, %edi` # edi = t1 $\gg$ 17 (t2)
- `movl %edi, %eax` # eax = edi
- `andl $8185, %eax` # eax = t2 & mask (rval)

Supplied by CMU, but converted to x86-64.
Quiz 3

- What is the final value in %esi?

```
xorl %esi, %esi
incl %esi
sall %esi, %esi
addl %esi, %esi
```

a) 2  
b) 4  
c) 8  
d) indeterminate