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.
The early computers of the x86 family had 16-bit words, starting with the 386, they supported 32-bit words.
$2^{32} = 4$ gigabytes.

$2^{64} = 16$ exbibytes

All SunLab computers are x86-64.
Data Types on IA32 and x86-64

• “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

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

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><strong>Reg</strong></td>
<td>movl $0x4,%eax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td></td>
<td><strong>Mem</strong></td>
<td>movl $-147,(%eax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td><strong>movl</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Reg</strong></td>
<td>movl %eax,%edx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td></td>
<td><strong>Mem</strong></td>
<td>movl %eax,(%edx)</td>
<td>*p = temp;</td>
</tr>
<tr>
<td></td>
<td><strong>Mem</strong></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.
Supplied by CMU.

If one thinks of there being an array of registers, then “Reg[R]” selects register “R” from this array.
We discuss the “set up” and “finish” in a subsequent lecture. They have to do with facilitating the calling of functions.
Understanding Swap

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

```
Register       Value
%edx           xp
%ecx           yp
%ebx           t0
%eax           t1

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>Reg</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>0x11c</td>
</tr>
<tr>
<td>%ebx</td>
<td>0x118</td>
</tr>
<tr>
<td>%esi</td>
<td>0x114</td>
</tr>
<tr>
<td>%edi</td>
<td>0x110</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x10c</td>
</tr>
<tr>
<td>%esp</td>
<td>0x108</td>
</tr>
</tbody>
</table>

#### Code Snippet

```assembly
tmp = %ebp + 0x100
movl 0(%ebp), %edx  # edx = xp
dwld %ebp + 0x104
```

---

Supplied by CMU.
## Understanding Swap

<table>
<thead>
<tr>
<th>Location</th>
<th>0x104</th>
<th>0x108</th>
<th>0x110</th>
<th>0x114</th>
<th>0x120</th>
<th>0x124</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ecx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ebx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%esi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*ebp</td>
<td>0x104</td>
<td></td>
<td></td>
<td></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

%eax
%edx 0x124
%ecx 0x120
%ebx
%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

Offset

Address

123
456
0x120
0x11c
0x118
0x114
0x110
0x10c
0x108
0x104
0x100

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
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>Rtn adr</td>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x114</td>
<td>0</td>
<td>0x104</td>
<td>0x104</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td>-4</td>
<td>0x100</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>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>0x124</td>
</tr>
<tr>
<td>0x110</td>
<td>0x114</td>
</tr>
<tr>
<td>0x10c</td>
<td>0x108</td>
</tr>
<tr>
<td>0x104</td>
<td>0x108</td>
</tr>
<tr>
<td>0x100</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>

```
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
```
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  // b  // c  // d
int x;  int *x;
int y;  int y;
y = x;  y = *x;

CS33 Intro to Computer Systems  X-19  Copyright © 2018 Thomas W. Doepner. All rights reserved.
Complete Memory-Addressing Modes

- **Most general form**
  
  \[ D(Rb,Ri,S) \rightarrow Mem[Reg[Rb]+S*Reg[Ri]+D] \]
  
  - **D**: constant "displacement"
  - **Rb**: base register: any of 8 integer registers
  - **Ri**: index register: any, except for %esp
    
    » unlikely you’d use %esp either
  - **S**: scale: 1, 2, 4, or 8

- **Special cases**

<table>
<thead>
<tr>
<th>(Rb,Ri)</th>
<th>Mem[Reg[Rb]+Reg[Ri]]</th>
</tr>
</thead>
<tbody>
<tr>
<td>D(Rb,Ri)</td>
<td>Mem[Reg[Rb]+Reg[Ri]+D]</td>
</tr>
<tr>
<td>(Rb,Ri,S)</td>
<td>Mem[Reg[Rb]+S*Reg[Ri]]</td>
</tr>
<tr>
<td>D</td>
<td>Mem[D]</td>
</tr>
</tbody>
</table>

Supplied by CMU.
## Address-Computation Examples

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>∘edx</td>
<td>0xf000</td>
<td></td>
</tr>
<tr>
<td>∘ecx</td>
<td>0x0100</td>
<td></td>
</tr>
</tbody>
</table>

<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.
Address-Computation Instruction

- `leal src, dest`
  - `src` is address mode expression
  - set `dest` to address denoted by expression

- Uses
  - computing addresses without a memory reference
    » e.g., translation of \( p = s x[i] \)
  - computing arithmetic expressions of the form \( x + k \cdot y \)
    » \( k = 1, 2, 4, \) or 8

- Example

```c
int mul12(int x)
{
    return x*12;
}
```

Converted to ASM by compiler:

```asm
movl 8(%ebp), %eax      # get arg
leal (%eax,%eax,2), %eax # t <- x+x*2
sall $2, %eax           # return t<<2
```

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:
Supplied by CMU.

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.

Note also that each register, in addition to having a 32-bit version, also has an 8-bit (one-byte) version. For the numbered registers, it’s, for example, %r10b. For the other registers it’s the same as for IA32.
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.

32-bit Instructions on x86-64

- `addl 4(%rdx), %eax`
  - memory address must be 64 bits
  - operands (in this case) are 32-bit
    - result goes into %eax
    - lower half of %rax
    - upper half is filled with zeroes
Bytes

- Each register has a byte version
  - e.g., %r10: %r10b
- Needed for byte instructions
  - movb (%rax, %rsi), %r10b
  - sets only the low byte in %r10
    » other seven bytes are unchanged
- Alternatives
  - movzbq (%rax, %rsi), %r10
    » copies byte to low byte of %r10
    » zeroes go to higher bytes
  - movsbq (%rax, %rsi), %r10
    » copies byte to low byte of %r10
    » sign is extended to all higher bits

Note that using single-byte versions of registers has a different behavior from using 4-byte versions of registers. Putting data into the latter using mov causes the upper bytes to be zeroed. But with the byte versions, putting data into them does not affect the upper bytes.
Supplied by CMU.

Note that for the IA32 architecture, arguments are passed on the stack.
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

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

- **Set Up**
  - `movq (%rdi), %rdx`
  - `movq (%rsi), %rax`
  - `movq %rax, (%rdi)`
  - `movq %rdx, (%rsi)`
- **Body**
  - `ret`
- **Finish**

**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.
Example

```c
int sum(int a, int b) {
    return (a+b);
}
```
Object Code

Code for `sum`

0x401040 `<sum>`:
- 0x55
- 0x89
- 0xe5
- 0x8b
- 0x45
- 0xc0
- 0x03
- 0x5d
- 0xc3

- **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.
This is taken from Intel 64 and IA-32 Architecture Software Developer’s Manual, Volume 2: Instruction Set Reference; Order Number 325462-043US, Intel Corporation, May 2012.
Disassembling Object Code

Disassembled

```
080483c4 <sum>:
080483c4: 55  push  ebp
080483c5: 89 e5  mov  esp,ebp
080483c7: 8b 45 0c  mov  0xc(%ebp),%eax
080483ca: 03 45 08  add  0x8(%ebp),%eax
080483cd: 5d  pop  ebp
080483ce: c3  ret
```

- **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

Object

<table>
<thead>
<tr>
<th>0x401040:</th>
<th>Disassembled</th>
</tr>
</thead>
<tbody>
<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>0x0c</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>

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 ~30
- 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
    - ~90
  - SIMD instructions
    - lots
  - AMD-added instructions
  - undocumented instructions

The source for this is http://en.wikipedia.org/wiki/X86_instruction_listings, viewed on 6/20/2017, which comes with the caveat that it may be out of date.
Some Arithmetic Operations

- **Two-operand instructions:**

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>addl</td>
<td>Source, Destination</td>
<td>Dest = Dest + Source</td>
</tr>
<tr>
<td>subl</td>
<td>Source, Destination</td>
<td>Dest = Dest - Source</td>
</tr>
<tr>
<td>imull</td>
<td>Source, Destination</td>
<td>Dest = Dest * Source</td>
</tr>
<tr>
<td>sall</td>
<td>Source, Destination</td>
<td>Dest = Dest &lt;&lt;&lt; Source</td>
</tr>
<tr>
<td>sarl</td>
<td>Source, Destination</td>
<td>Dest = Dest &gt;&gt;&gt; Source</td>
</tr>
<tr>
<td>shr1</td>
<td>Source, Destination</td>
<td>Dest = Dest &gt;&gt;&gt; Source</td>
</tr>
<tr>
<td>xorl</td>
<td>Source, Destination</td>
<td>Dest = Dest ^ Source</td>
</tr>
<tr>
<td>andl</td>
<td>Source, Destination</td>
<td>Dest = Dest &amp; Source</td>
</tr>
<tr>
<td>orl</td>
<td>Source, Destination</td>
<td>Dest = Dest</td>
</tr>
</tbody>
</table>

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

Supplied by CMU.

Note that for shift instructions, the Source operand (which is the size of the shift) must either be a immediate operand or be a designator for a one-byte register (e.g., %cl – see the slide on general-purpose registers for IA32).
Some Arithmetic Operations

- **One-operand Instructions**
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>incl</td>
<td>Dest + 1</td>
</tr>
<tr>
<td>decl</td>
<td>Dest − 1</td>
</tr>
<tr>
<td>negl</td>
<td>− Dest</td>
</tr>
<tr>
<td>notl</td>
<td>~Dest</td>
</tr>
</tbody>
</table>

- **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;
}
```

```assembly
arith:
    leal (%rdi,%rsi), %eax
    addl %edx, %eax
    leal (%rsi,%rsi,2), %edx
    sall $4, %edx
    leal 4(%rdi,%rdx), %ecx
    imull %ecx, %eax
    ret
```

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

```
leal (%rdi,%rsi), %eax
addl %edx, %eax
leal (%rsi,%rsi,2), %edx
sall $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
```
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>>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 %ecx?

```assembly
xorl %ecx, %ecx
incl %ecx
sall %cl, %ecx  # %cl is the low byte of %ecx
addl %ecx, %ecx
```

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