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.
Jump Instructions

- Unconditional jump
  - just do it
- Conditional jump
  - to jump or not to jump determined by condition-code flags
  - field in the op code indicates how this is computed
  - in assembler language, simply say
    » je
      - jump on equal
    » jne
      - jump on not equal
    » jgt
      - jump on greater than
    » etc.

Jump instructions cause the processor to start executing instructions at some specified address. For conditional jump instructions, whether to jump or not is determined by the values of the condition codes. Fortunately, rather than having to specify explicitly those values, one may use mnemonics as shown in the slide.

We'll see examples of their use in the next lecture, when we start looking at x86 assembler instructions.
In the C code above, the assignment to \( a \) might be coded in assembler as shown in the box in the lower left. But this brings up the question, where are the values represented by \( a, b, c, \) and \( d? \) Variable names are part of the C language, not assembler. Let’s assume that these global variables are located at addresses 1000, 1004, 1008, and 1012, as shown on the right. Thus correct assembler language would be as in the middle box, which deals with addresses, not variable names. Note that “mov 1004,%acc” means to copy the contents of location 1004 to the accumulator register; it does not mean to copy the integer 1004 into the register!

Beginning with this slide, whenever we draw pictures of memory, lower memory addresses are at the bottom, higher addresses are at the top. This is the opposite of how we’ve been drawing pictures of memory in previous slides.
Here we rearrange things a bit. \( b \) is a global variable, but \( a \) is a local variable within \( \text{func} \), and \( c \) and \( d \) are arguments. The issue here is that the locations associated with \( a \), \( c \), and \( d \) will, in general, be different for each call to \( \text{func} \). Thus we somehow must modify the assembler code to take this into account.
Note that both positive and negative offsets might be used.
Here we load the value 10,000 into the base register (recall that the “$” means what follows is a literal value; a “%” sign means that what follows is the name of a register), then store the value 10 into the memory location 10100 (the contents of the base register plus 100): the notation \( n(%\text{base}) \) means the address obtained by adding \( n \) to the contents of the base register.
Here we return to our earlier example. We assume that, as part of the call to `func`, the base register is loaded with the address of the beginning of `func`’s current stack frame, and that the local variable `a` and the parameters `c` and `d` are located within the frame. Thus we refer to them by their offset from the beginning of the stack frame, which are assumed to be -16, -8, and -12. Since the stack grows from higher addresses to lower addresses, these offsets are negative. Note that the first assembler instruction copies the contents of location 1000 into `%acc.
Quiz 1

Suppose the value in base is 10,000. What is the address of c?

a) 9992  
b) 9996  
c) 10,004  
d) 10,008

mov 1000, %acc  
add -8(%base), %acc  
mul -12(%base), %acc  
mov %acc, -16(%base)
We've now seen four registers: the instruction pointer, the accumulator, the base register, and the condition codes. The accumulator is used to hold intermediate results for arithmetic; the base register is used to hold addresses for relative addressing. There's no particular reason why the accumulator can't be used as the base register and vice versa: thus they may be used interchangeably. Furthermore, it is useful to have more than two such dual-purpose registers. As we will see, the x86 architecture has eight such registers; the x86-64 architecture has 16.
Why do we make the distinction between registers and memory? Registers are in the processor itself and can be read from and written to very quickly. Memory is on separate hardware and takes much more time to access than registers do. Thus operations involving only registers can be executed very quickly, while significantly more time is required to access memory. Processors typically have relatively few registers (the IA-32 architecture has eight, the x86-64 architecture has 32; some other architectures have many more, perhaps as many as 256); memory is measured in gigabytes.

Note that memory access-time is mitigated by the use of on-processor caches, something that we will discuss in a few weeks.
The early computers of the x86 family had 16-bit words, starting with the 386, they supported 32-bit words.
$2^{64}$

- $2^{32}$ used to be considered a large number
  - one couldn’t afford $2^{32}$ bytes of memory, so no problem with that as an upper bound
- Intel (and others) saw need for machines with 64-bit addresses
  - devised IA64 architecture with HP
    » became known as Itanium
    » very different from x86
- AMD also saw such a need
  - developed 64-bit extension to x86, called x86-64
- Itanium flopped
- x86-64 dominated
- Intel, reluctantly, adopted x86-64

$2^{32} = 4$ 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.
Supplied by CMU.
Moving Data: IA32

- **Moving data**
  \[
  \text{movl source, dest}
  \]

- **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(s)
    - simplest example: `%eax`
    - various other “address modes”

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>movl</td>
<td><strong>Reg</strong></td>
<td>movl %eax,%edx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td>Mem</td>
<td><strong>Reg</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.
Supplied by CMU.

We discuss the “set up” and “finish” in a subsequent lecture. They have to do with facilitating the calling of functions.
Supplied by CMU.
Understanding Swap

Supplied by CMU.
# Understanding Swap

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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x100</td>
</tr>
<tr>
<td>4</td>
<td>0x108</td>
</tr>
<tr>
<td>8</td>
<td>0x124</td>
</tr>
<tr>
<td>12</td>
<td>0x120</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 | 0x124 |
| %edx | 0x120 |
| %ecx | 0x11c |
| %ebx | 0x118 |
| %esi | 0x114 |
| %edi | 0x110 |
| %ebp | 0x10c |
| %esp | 0x108 |
| %ebp | 0x104 |
| %ebp | 0x100 |

<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>

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

%eax
%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

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

Offset
YP 12
xp 8
4 Rtn adr
0
-4

Supplied by CMU.
## 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>

### 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

### Register Table

<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>0x100</td>
</tr>
<tr>
<td>-4</td>
<td>0x104</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x114</td>
</tr>
<tr>
<td></td>
<td>0x118</td>
</tr>
<tr>
<td></td>
<td>0x11c</td>
</tr>
<tr>
<td></td>
<td>0x120</td>
</tr>
<tr>
<td></td>
<td>0x124</td>
</tr>
<tr>
<td></td>
<td>0x10c</td>
</tr>
<tr>
<td></td>
<td>0x108</td>
</tr>
<tr>
<td></td>
<td>0x104</td>
</tr>
<tr>
<td></td>
<td>0x100</td>
</tr>
</tbody>
</table>

Supplied by CMU.
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

Supplied by CMU.
## 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>0x100</td>
</tr>
</tbody>
</table>

**Address**: 0x124  
0x120  
0x11c  
0x118  
0x114  
0x110  
0x10c  
0x108  
0x104  
0x100  

<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>

```plaintext
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 2

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 **x;      int ***x;
int y;       int y;       int y;        int y;
y = x;       y = *x;      y = **x;      y = ***x;
Complete Memory-Addressing Modes

• Most general form

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

- \( D \): constant “displacement”
- \( R_b \): base register: any of 8 integer registers
- \( R_i \): index register: any, except for \( %esp \)
  - unlikely you’d use \( %esp \) either
- \( S \): scale: 1, 2, 4, or 8

• Special cases

\begin{align*}
(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\times\text{Reg}[R_i]] \\
D & \quad \text{Mem}[D]
\end{align*}

Supplied by CMU.
### Address-Computation Examples

<table>
<thead>
<tr>
<th>Command</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\times0x0100)</td>
<td>(0xf400)</td>
</tr>
<tr>
<td>(0x80(,%edx,2))</td>
<td>(2\times0xf000 + 0x80)</td>
<td>(0x1e080)</td>
</tr>
</tbody>
</table>

---

Supplied by CMU.
Note that a function returns a value by putting it in %eax.
Quiz 3

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:
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.
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;
    ret
}
```

- **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
  0xb
  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 `printf`
  - some libraries are *dynamically linked*
    - linking occurs when program begins execution

Supplied by CMU.
Instruction Format

Disassembling Object Code

Disassembled

```
080483c4 <sum>:
  0x80483c4:  55    push   %ebp
  0x80483c5:  e5 89    mov    %esp,%ebp
  0x80483c7:  8b e5 45 0c    mov    0xc(%ebp),%eax
  0x80483ca:  03 45 08 45    add    0x0(%ebp),%eax
  0x80483cd:  c3 5d    pop    %ebp
  0x80483ce:  c3 0f    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 | Disassembled
---|---
0x401040: 0x55 0x89 0xe5 0x8b 0x45 0xc 0x03 0x45 0x08 0x5d 0xc3

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`
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
  - 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>add1</td>
<td>Src,Dest</td>
<td>Dest = Dest + Src</td>
</tr>
<tr>
<td>sub1</td>
<td>Src,Dest</td>
<td>Dest = Dest – Src</td>
</tr>
<tr>
<td>imul1</td>
<td>Src,Dest</td>
<td>Dest = Dest * Src</td>
</tr>
<tr>
<td>sall1</td>
<td>Src,Dest</td>
<td>Dest = Dest &lt;&lt; Src</td>
</tr>
<tr>
<td>sar1</td>
<td>Src,Dest</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>shr1</td>
<td>Src,Dest</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>xor1</td>
<td>Src,Dest</td>
<td>Dest = Dest ^ Src</td>
</tr>
<tr>
<td>and1</td>
<td>Src,Dest</td>
<td>Dest = Dest &amp; Src</td>
</tr>
<tr>
<td>or1</td>
<td>Src,Dest</td>
<td>Dest = Dest</td>
</tr>
</tbody>
</table>

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

Note that for shift instructions, the Src 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

incl Dest = Dest + 1
decl Dest = Dest − 1
negl Dest = −Dest
notl Dest = “Dest

• See book for more instructions
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
    sal $4, %edx
    leal 4(%rdi,%rdx), %ecx
    imull %ecx, %eax
    ret
```
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.
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

```assembly
lea (%rdi,%rsi), %eax        # eax = x+y   (t1)
addl %edx, %eax              # eax = t1+z   (t2)
lea (%rsi,%rsi,2), %edx      # edx = 3*y    (t4)
sll $4, %edx                 # edx = t4*16   (t4)
lea 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>>17 (t2)
- movl %edi, %eax  # eax = edi
- andl $8185, %eax  # eax = t2 & mask (rval)

Supplied by CMU, but converted to x86-64.
Quiz 4

* What is the final value in %ecx?

```
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