

CS 33

Machine Programming (1)

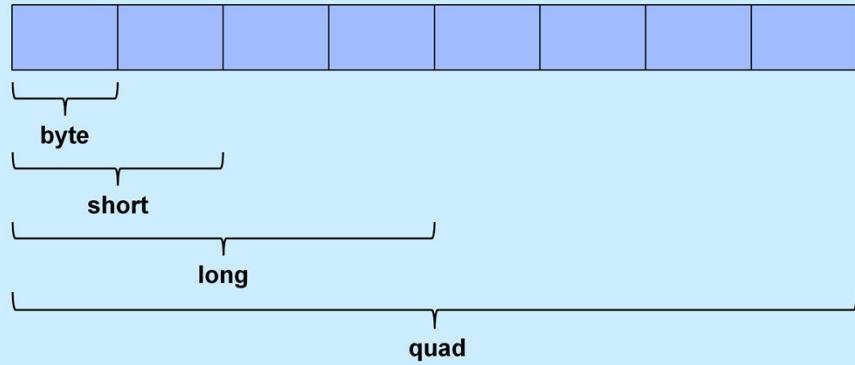
Many of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook “Computer Systems: A Programmer’s Perspective,” 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O’Hallaron in 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

Supplied by CMU.

Operand Size



- Rather than `mov` ...

- `movb`
- `movs`
- `movl`
- `movq (x86-64 only)`

Most instructions come in three (on IA32) or four (on x86-64) forms, one for each possible operand size.

Machine-Instruction Example

- **C code**
 - add two signed integers
- **Assembly**
 - add two 4-byte integers
 - » ints in C parlance
 - » same instruction whether signed or unsigned
 - **Operands:**
 - x: register %eax
 - y: memory M[%ebp+8]
- **Object code**
 - 3-byte instruction
 - stored at address 0x80483ca

```
addl 8(%ebp),%eax
```

Similar to expression:

x += y

Sort of like:

```
int eax;  
int *ebp;  
eax += ebp[2];
```

```
0x80483ca: 03 45 08
```

Supplied by CMU.

Note that rather than “add”, the instruction used is “addl”, which indicates that it is operating on 32-bit values.

The “M” in “M[%ebp+8]” is the conceptual memory array. In the case of this instruction, we are adding 8 to the contents of register ebp to produce an address, which is treated as an index into the “memory array.”

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.

General-Purpose Registers (IA32)

Origin
(mostly obsolete)

%eax	%ax	%ah	%al	accumulate
%ecx	%cx	%ch	%cl	counter
%edx	%dx	%dh	%dl	data
%ebx	%bx	%bh	%bl	base
%esi	%si			source index
%edi	%di			destination index
%esp	%sp			stack pointer
%ebp	%bp			base pointer

16-bit virtual registers
(backwards compatibility)

Supplied by CMU.

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(s)

- » simplest example: (%eax)

- » various other “address modes”

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

Supplied by CMU.

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

movl Operand Combinations

	Source	Dest	Src, Dest	C Analog
movl	Imm	<i>Reg</i>	movl \$0x4,%eax	temp = 0x4;
		<i>Mem</i>	movl \$-147,(%eax)	*p = -147;
	Reg	<i>Reg</i>	movl %eax,%edx	temp2 = temp1;
	<i>Mem</i>	<i>Mem</i>	movl %eax,(%edx)	*p = temp;
		<i>Reg</i>	movl (%eax),%edx	temp = *p;

Cannot (normally) do memory-memory transfer with a single instruction

Supplied by CMU.

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
```

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

```
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)
```

Set
Up

Body

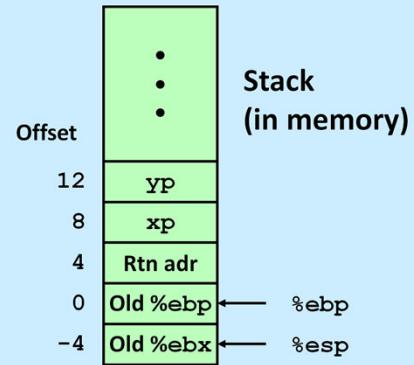
Finish

Supplied by CMU.

Understanding Swap

```
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
```

Supplied by CMU.

Understanding Swap



	Offset		Address
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
YP	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp	0		0x104
	-4		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
```

Supplied by CMU.

Understanding Swap

%eax	
%edx	0x124
%ecx	
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

	Offset	Address	Value
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
YP	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp	0		0x104
	-4		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
```

Supplied by CMU.

Understanding Swap

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

	Offset	Address
123	12	0x124
456	8	0x120
	4	0x11c
	0	0x118
	-4	0x114
YP	12	0x120
xp	8	0x124
	4	0x110
Rtn adr	0	0x10c
	-4	0x108
	-8	0x104
	-12	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
```

Supplied by CMU.

Understanding Swap

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

Offset	Address
12	123
8	456
4	
0	Rtn adr
-4	

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

Offset	Address
12	0x124
8	0x120
4	0x10c
0	0x108
-4	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

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

Offset	Address
12	0x124
8	0x120
4	0x11c
0	0x118
-4	0x114
	0x110
	0x10c
	0x108
	0x104
	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
```

Supplied by CMU.

Understanding Swap

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

Offset	Address
456	0x124
123	0x120
	0x11c
	0x118
	0x114
YP	0x120
xp	0x124
4	0x10c
Rtn adr	0x108
%ebp → 0	0x104
-4	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
```

Supplied by CMU.

Complete Memory-Addressing Modes

- Most general form

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

- D: constant “displacement”
- Rb: base register: any of 8 integer registers
- Ri: index register: any, except for %esp
 - » unlikely you'd use %ebp either
- S: scale: 1, 2, 4, or 8

- Special cases

$$(Rb, Ri) \quad \text{Mem}[Reg[Rb]+Reg[Ri]]$$

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

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

$$D \quad \text{Mem}[D]$$

Supplied by CMU.

Address-Computation Examples

%edx	0xf000
%ecx	0x0100

Expression	Address Computation	Address
0x8(%edx)	0xf000 + 0x8	0xf008
(%edx,%ecx)	0xf000 + 0x0100	0xf100
(%edx,%ecx,4)	0xf000 + 4*0x0100	0xf400
0x80(,%edx,2)	2*0xf000 + 0x80	0x1e080

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 $p = \&x[i];$
 - computing arithmetic expressions of the form $x + k^*y$
 - » $k = 1, 2, 4,$ or 8
- **Example**

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

Converted to ASM by compiler:

```
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.

x86-64 General-Purpose Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- Extend existing registers to 64 bits. Add 8 new ones.
- No special purpose for %ebp/%rbp

Supplied by CMU.

Note that %ebp/%rbp may be used 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.

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

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 code for swap

```
swap:  
void swap(int *xp, int *yp)  
{  
    int t0 = *xp;  
    int t1 = *yp;  
    *xp = t1;  
    *yp = t0;  
}  
  
pushl %ebp  
movl %esp,%ebp  
pushl %ebx } Set Up  
  
movl 8(%ebp), %edx  
movl 12(%ebp), %ecx  
movl (%edx), %ebx  
movl (%ecx), %eax  
movl %eax, (%edx)  
movl %ebx, (%ecx) } Body  
  
popl %ebx  
popl %ebp  
ret } Finish
```

Supplied by CMU.

64-bit code for swap

```
swap:  
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
```

} Set Up
} Body
} Finish

- 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

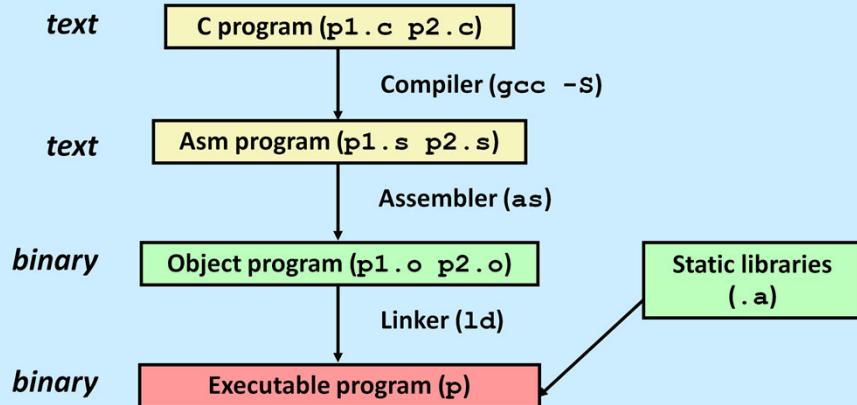
```
swap_1:  
void swap(long *xp, long *yp)  
{  
    long t0 = *xp;  
    long t1 = *yp;  
    *xp = t1;  
    *yp = t0;  
}  
  
        movq (%rdi), %rdx  
        movq (%rsi), %rax  
        movq %rax, (%rdi)  
        movq %rdx, (%rsi)  
ret
```

} Set Up
} Body
} Finish

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

Turning C into Object Code

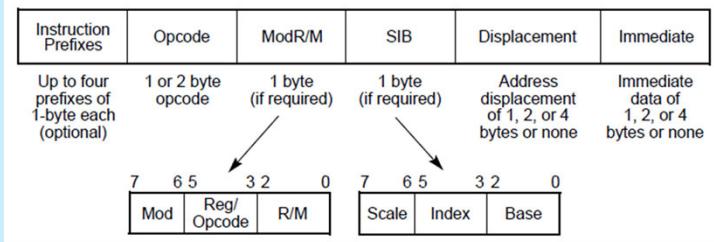
- Code in files `p1.c p2.c`
- Compile with command: `gcc -O1 p1.c p2.c -o p`
 - » use basic optimizations (`-O1`)
 - » put resulting binary in file `p`



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.

Instruction Format



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

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

Supplied by CMU.

Disassembling Object Code

Disassembled

```
080483c4 <sum>:  
80483c4: 55      push    %ebp  
80483c5: 89 e5    mov     %esp,%ebp  
80483c7: 8b 45 0c  mov     0xc(%ebp),%eax  
80483ca: 03 45 08  add    0x8(%ebp),%eax  
80483cd: 5d      pop    %ebp  
80483ce: 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

```
0x401040:  
0x55  
0x89  
0xe5  
0xb8  
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 <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:

<i>Format</i>	<i>Computation</i>	
addl <i>Src,Dest</i>	Dest = Dest + Src	
subl <i>Src,Dest</i>	Dest = Dest - Src	
imull <i>Src,Dest</i>	Dest = Dest * Src	
sall <i>Src,Dest</i>	Dest = Dest << Src	<i>Also called shll</i>
sar1 <i>Src,Dest</i>	Dest = Dest >> Src	<i>Arithmetic</i>
shrl <i>Src,Dest</i>	Dest = Dest >> Src	<i>Logical</i>
xorl <i>Src,Dest</i>	Dest = Dest ^ Src	
andl <i>Src,Dest</i>	Dest = Dest & Src	
orl <i>Src,Dest</i>	Dest = Dest Src	

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

Supplied by CMU.

Some Arithmetic Operations

- **One-operand Instructions**

incl	$Dest = Dest + 1$
decl	$Dest = Dest - 1$
negl	$Dest = -Dest$
notl	$Dest = \sim Dest$

- **See book for more instructions**

Supplied by CMU.

Arithmetic Expression Example

```
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
    sall $4, %edx
    leal 4(%rdi,%rdx), %ecx
    imull %ecx, %eax
    ret
```

Supplied by CMU, but converted to x86-64.

Understanding arith

```
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
```

%rdx		z
%rsi		y
%rdi		x

Supplied by CMU, but converted to x86-64.

Understanding arith

```
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;
}
```

%rdx		z
%rsi		y
%rdi		x

```
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.

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

```
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

```
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.