CS 33

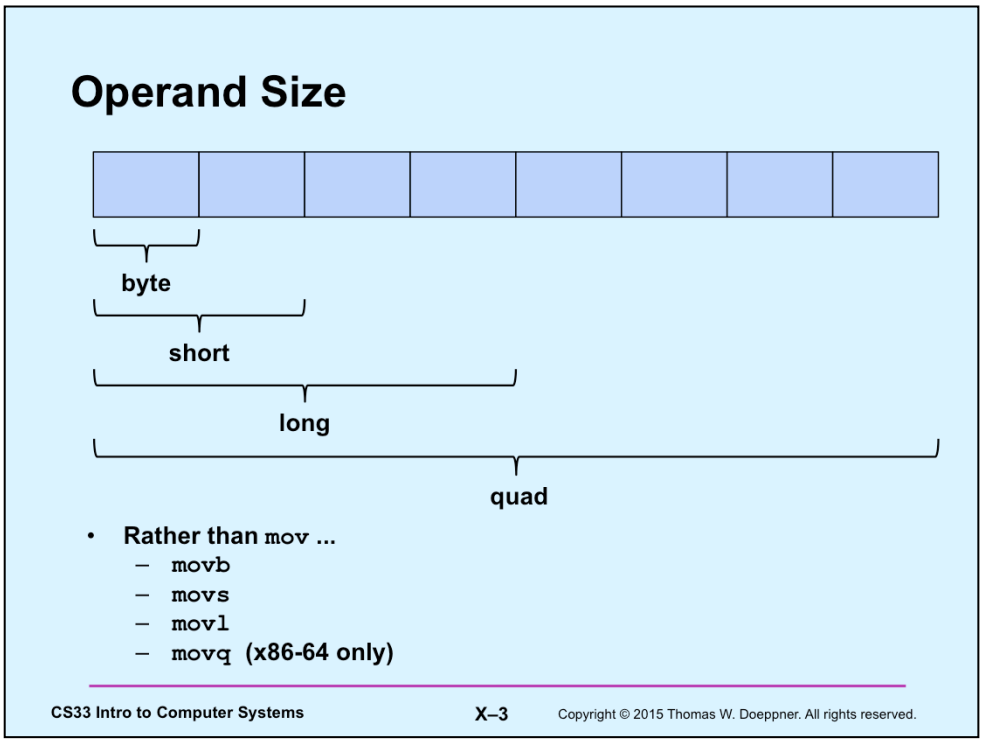
Machine Programming (1)

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.

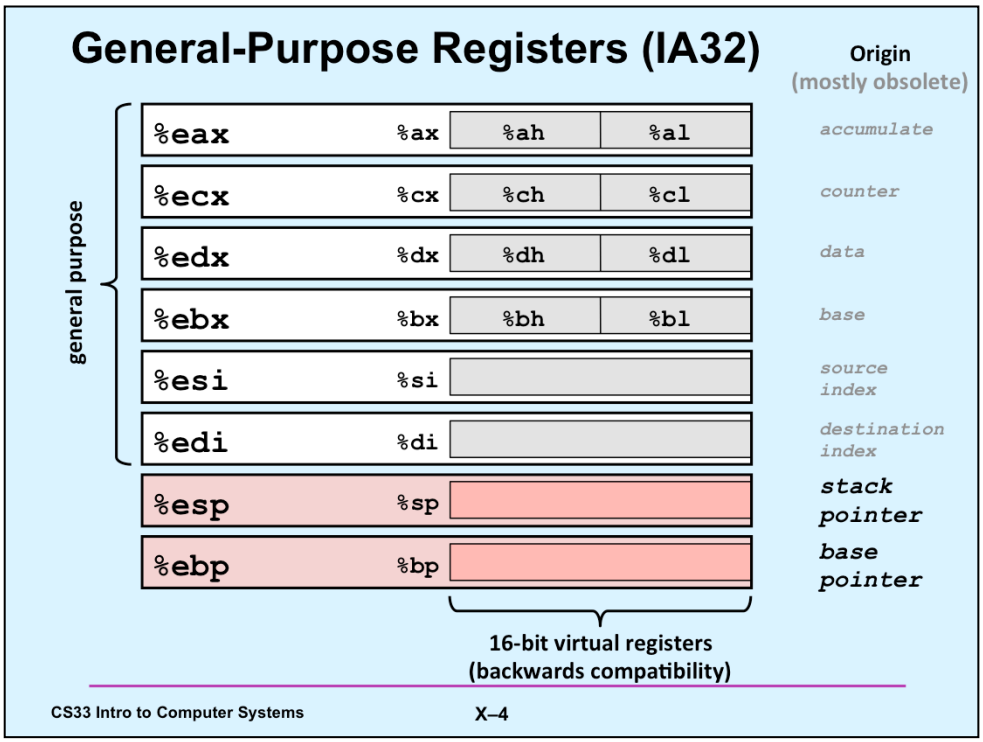
Data Types on Intel x86

- "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

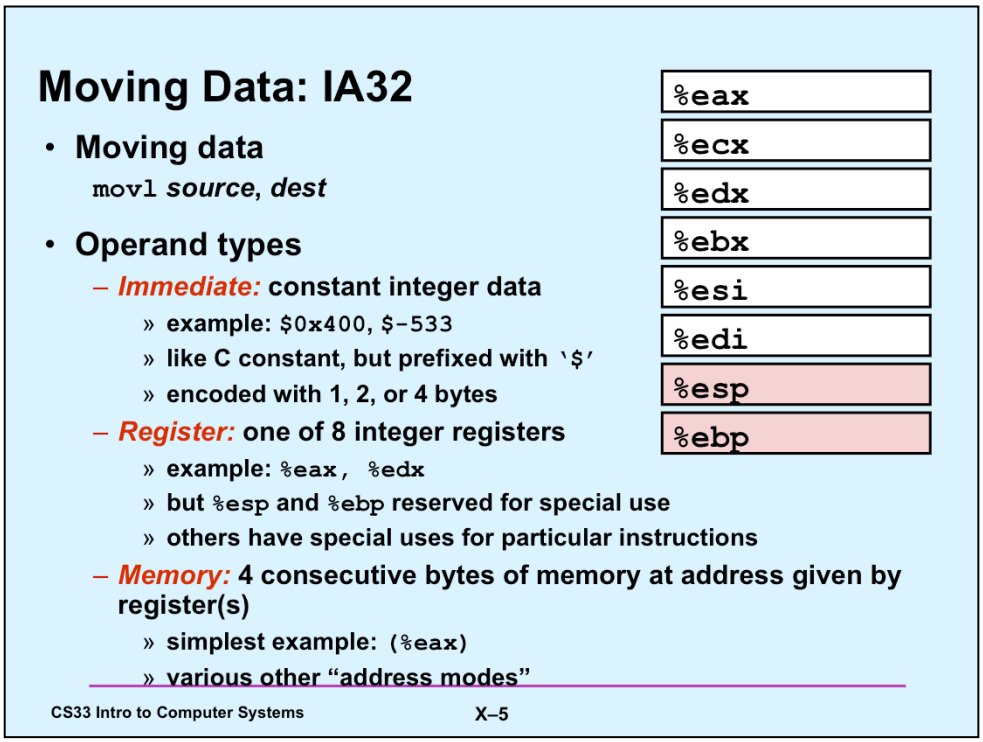
Supplied by CMU.



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



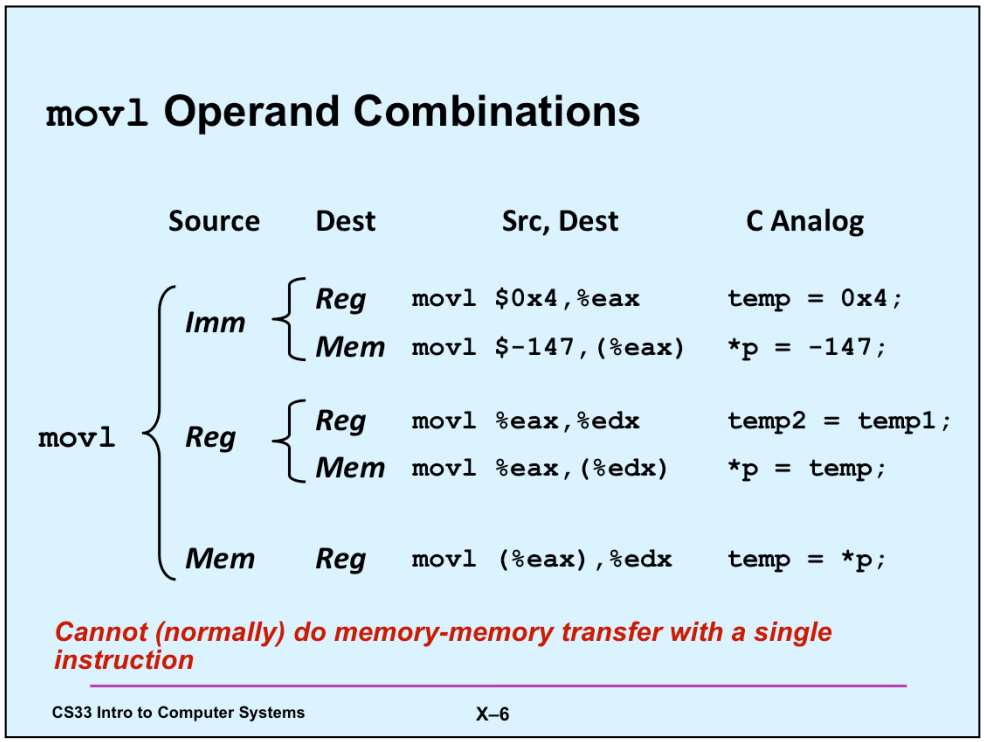
Supplied by CMU.



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.



Supplied by CMU.

Simple Memory Addressing Modes

- **Normal (R) Mem[Reg[R]]**
  - **register R specifies memory address**

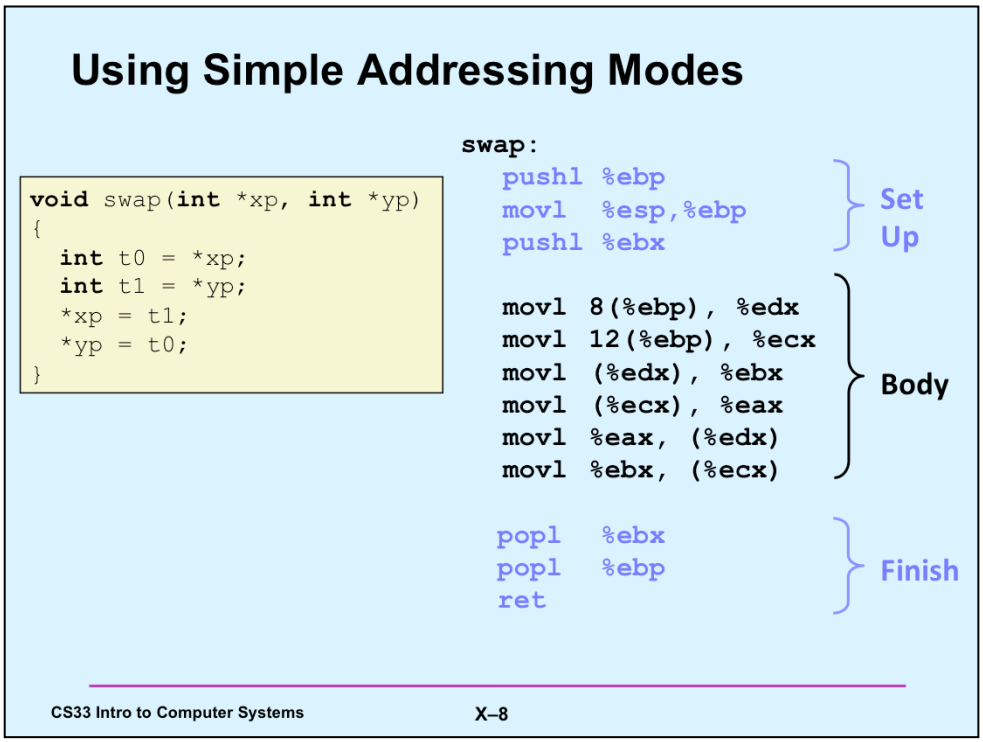
```
movl (%ecx),%eax
```

- **Displacement D(R) Mem[Reg[R]+D]**
  - **register R specifies start of memory region**
  - **constant displacement D specifies offset**

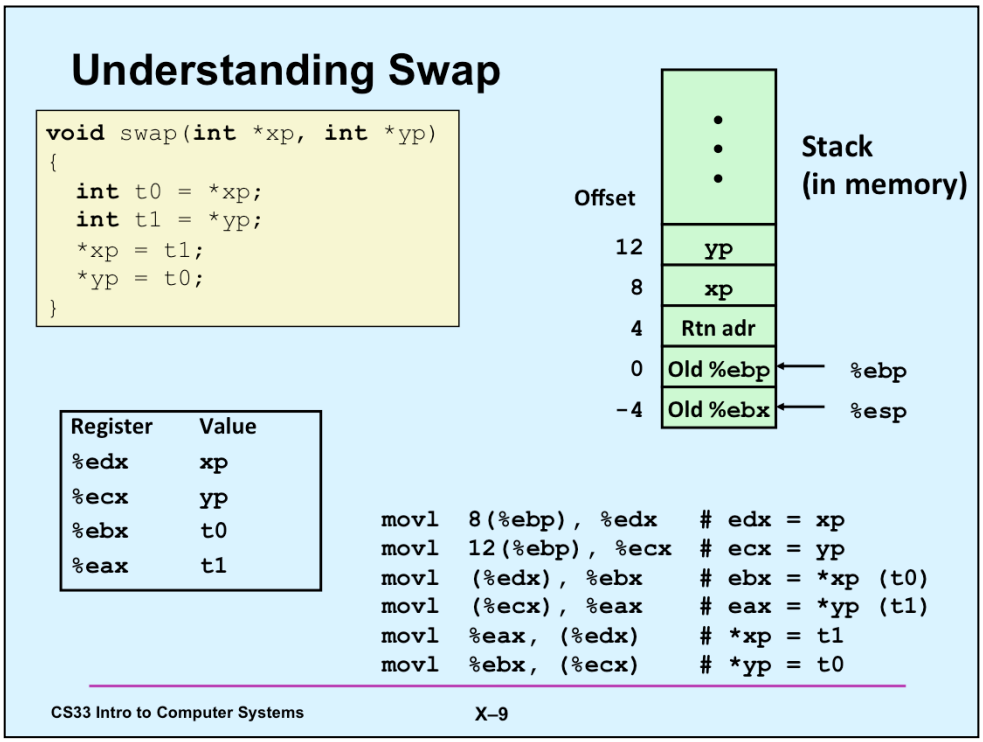
```
movl 8(%ebp),%edx
```


Supplied by CMU.

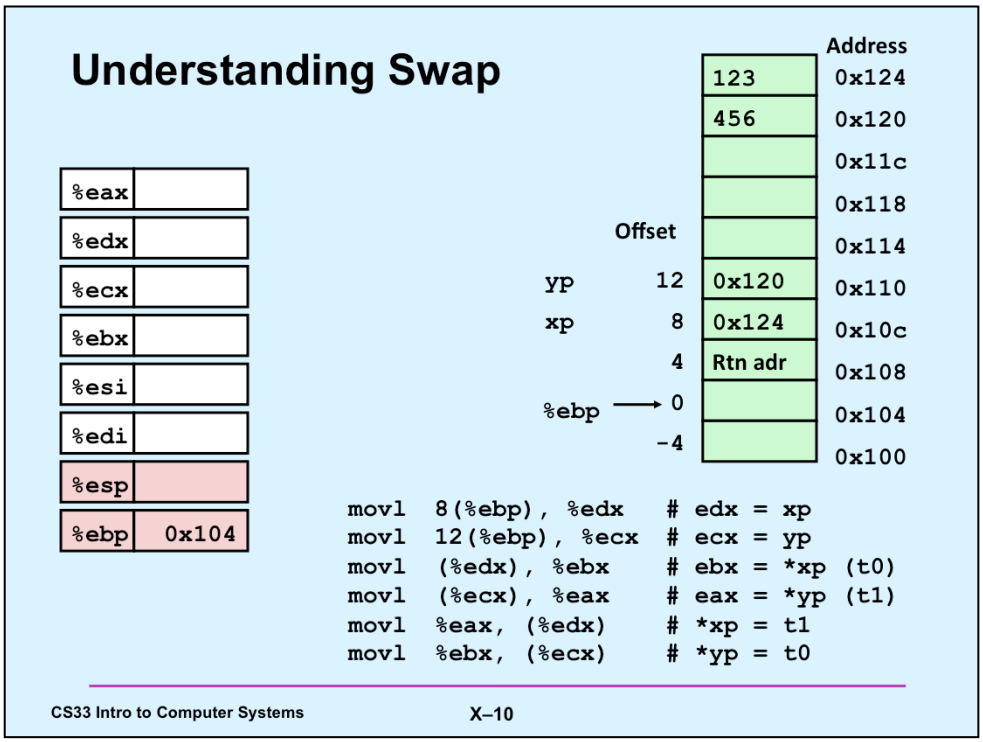
If one thinks of there being an array of registers, then “Reg[R]” selects register “R” from this array.



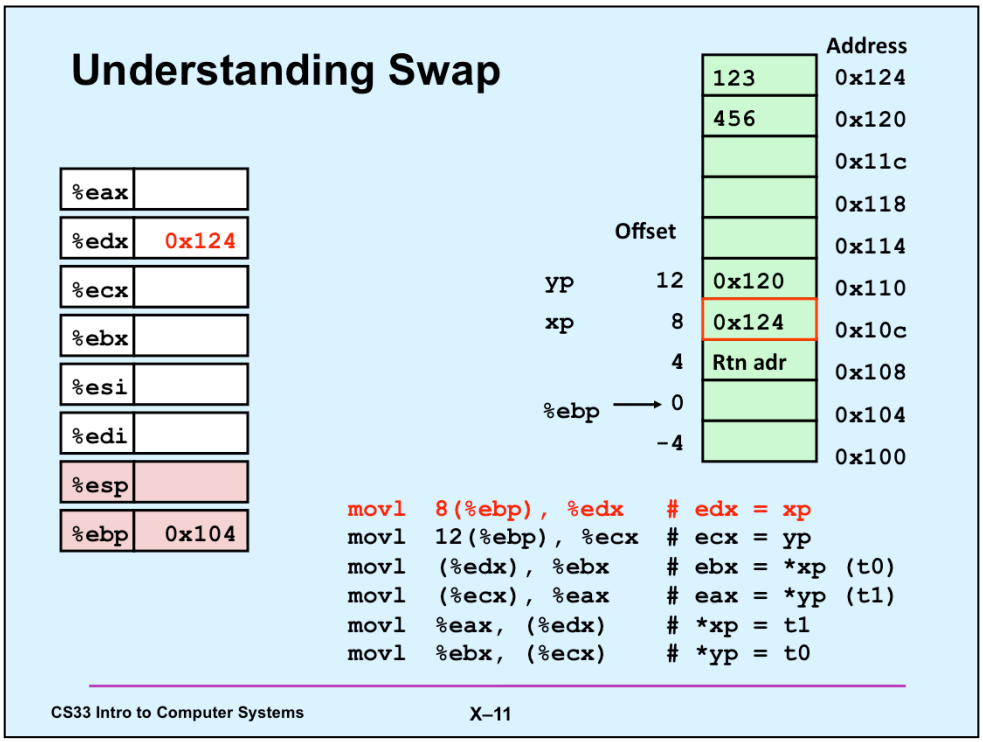
Supplied by CMU.



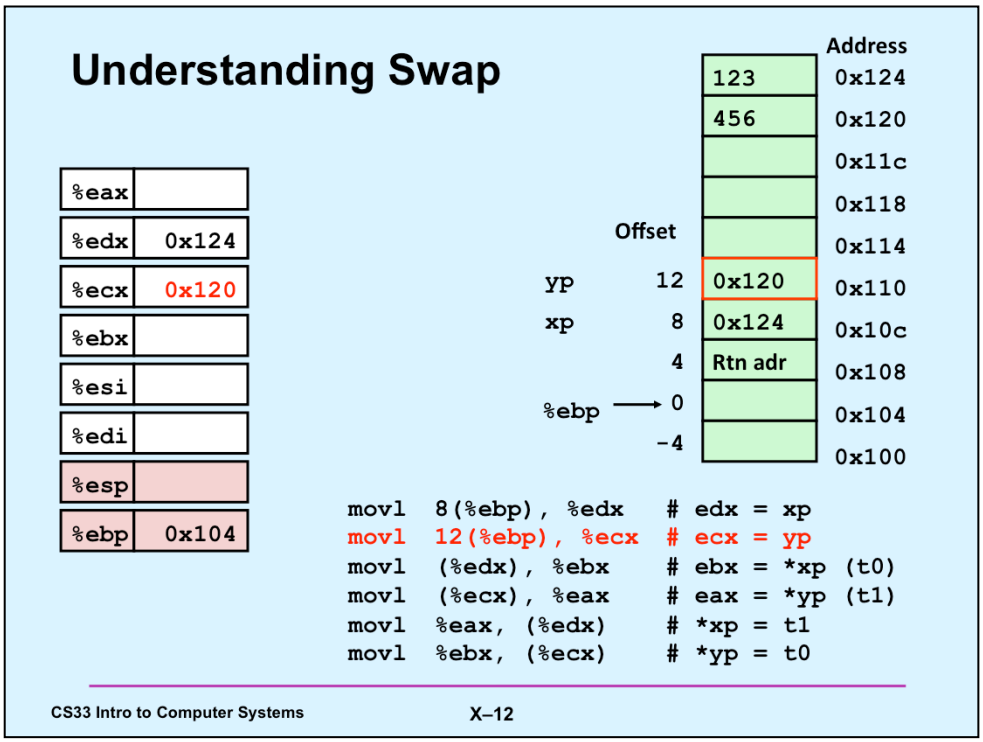
Supplied by CMU.



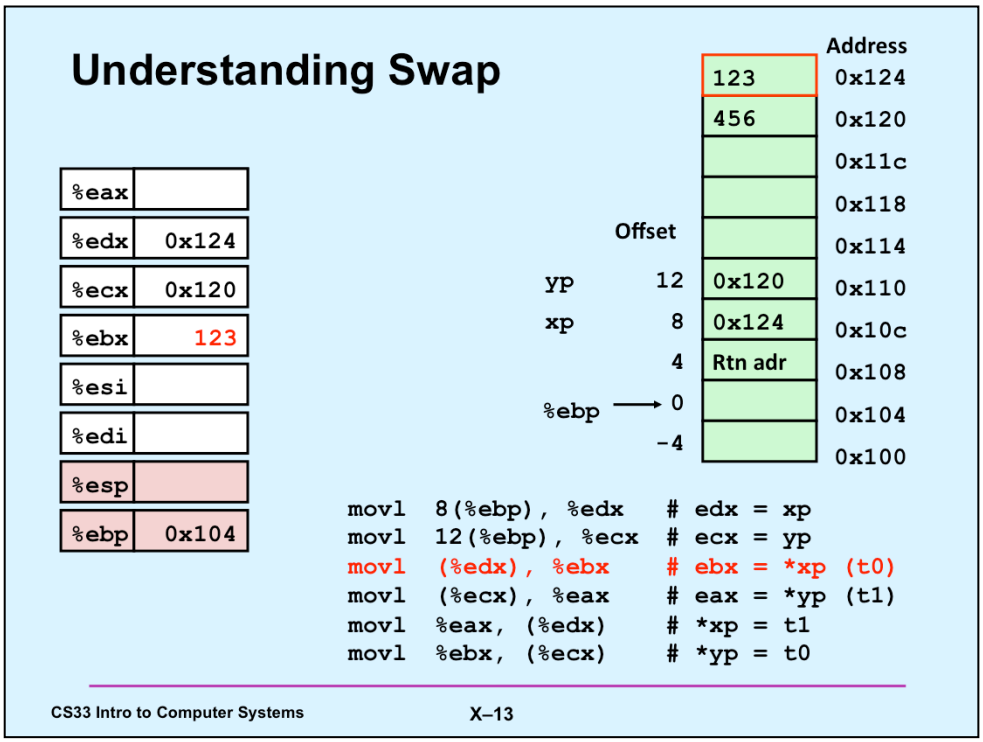
Supplied by CMU.



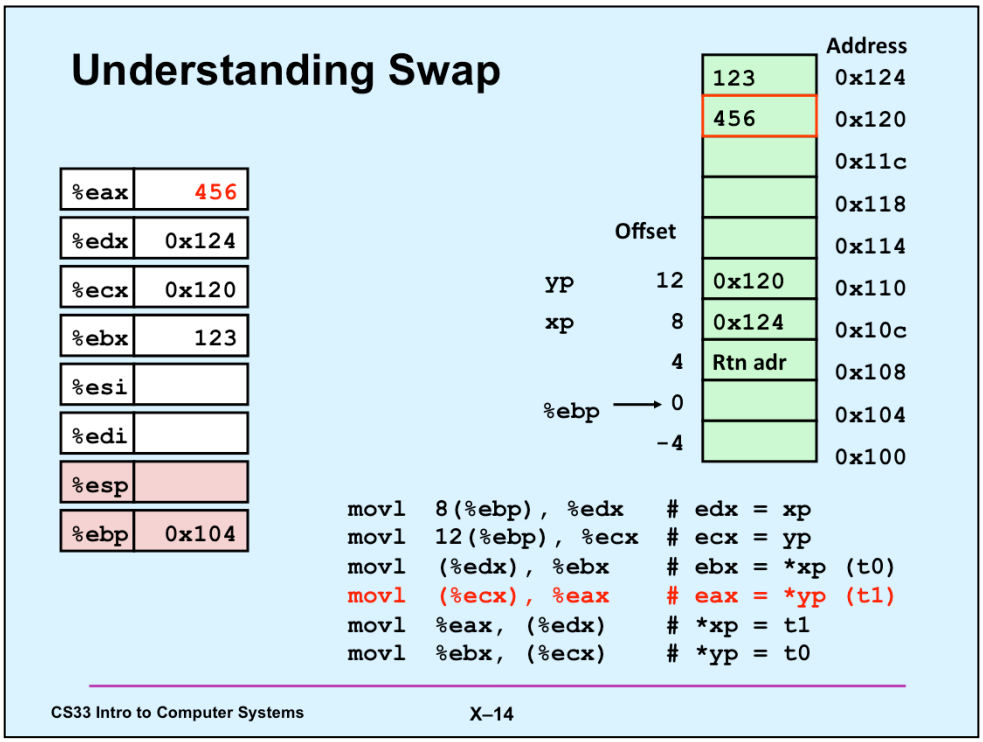
Supplied by CMU.



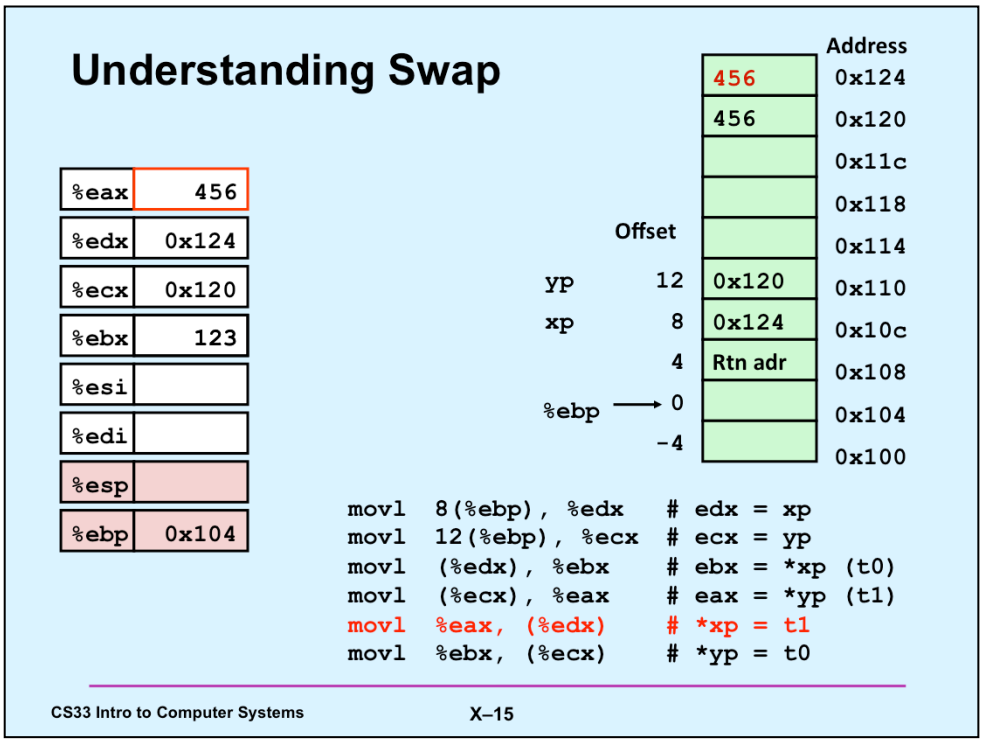
Supplied by CMU.



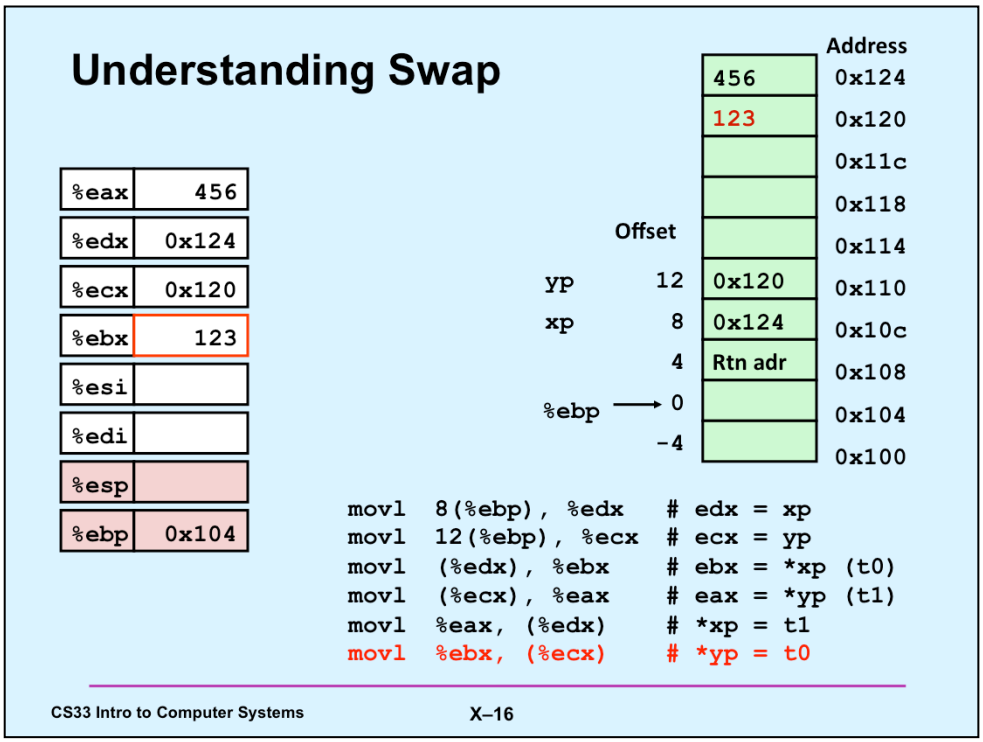
Supplied by CMU.



Supplied by CMU.



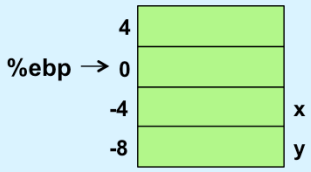
Supplied by CMU.



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) 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)	**Mem[Reg[Rb]+Reg[Ri]]**
D(Rb,Ri)	**Mem[Reg[Rb]+Reg[Ri]+D]**
(Rb,Ri,S)	**Mem[Reg[Rb]+S*Reg[Ri]]**
D	**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.

Quiz 2

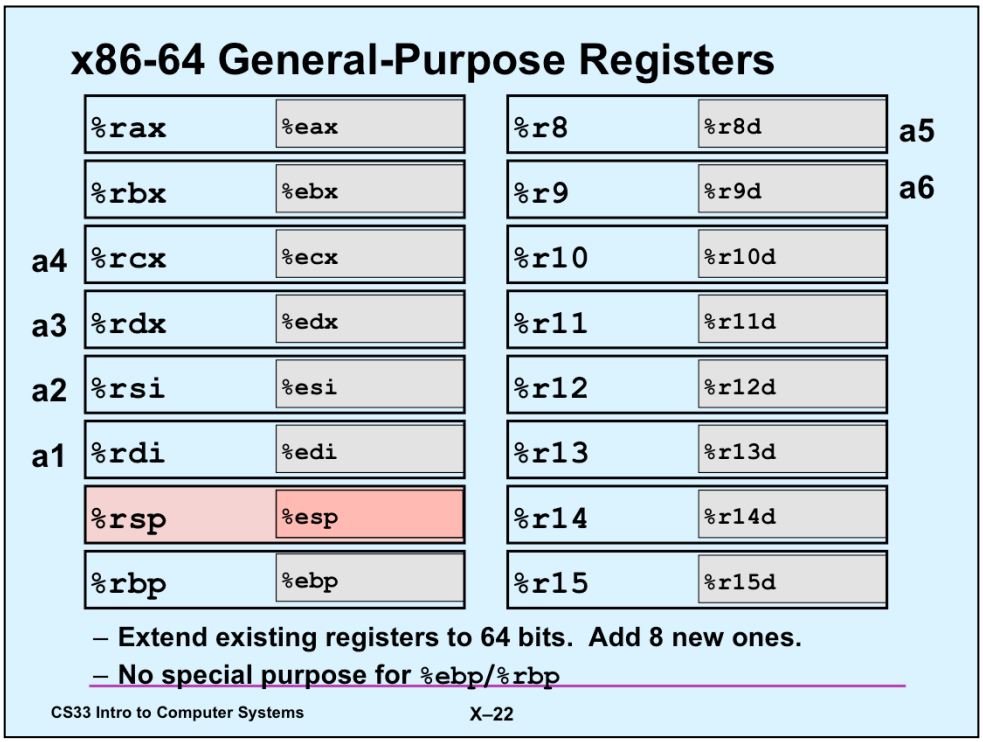
What value ends up in %ecx?

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

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

	1009:	0x09
	1008:	0x08
	1007:	0x07
	1006:	0x06
	1005:	0x05
	1004:	0x04
	1003:	0x03
	1002:	0x02
	1001:	0x01
%eax →	1000:	0x00

Hint:



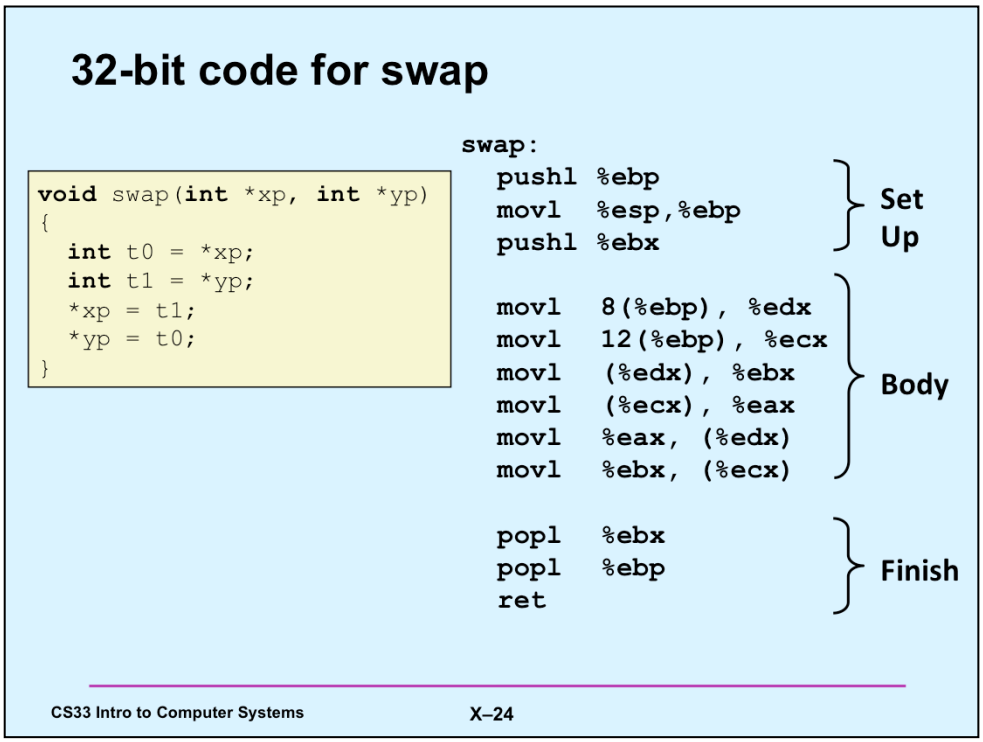
Supplied by CMU.

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.

32-bit Instructions on x86-64

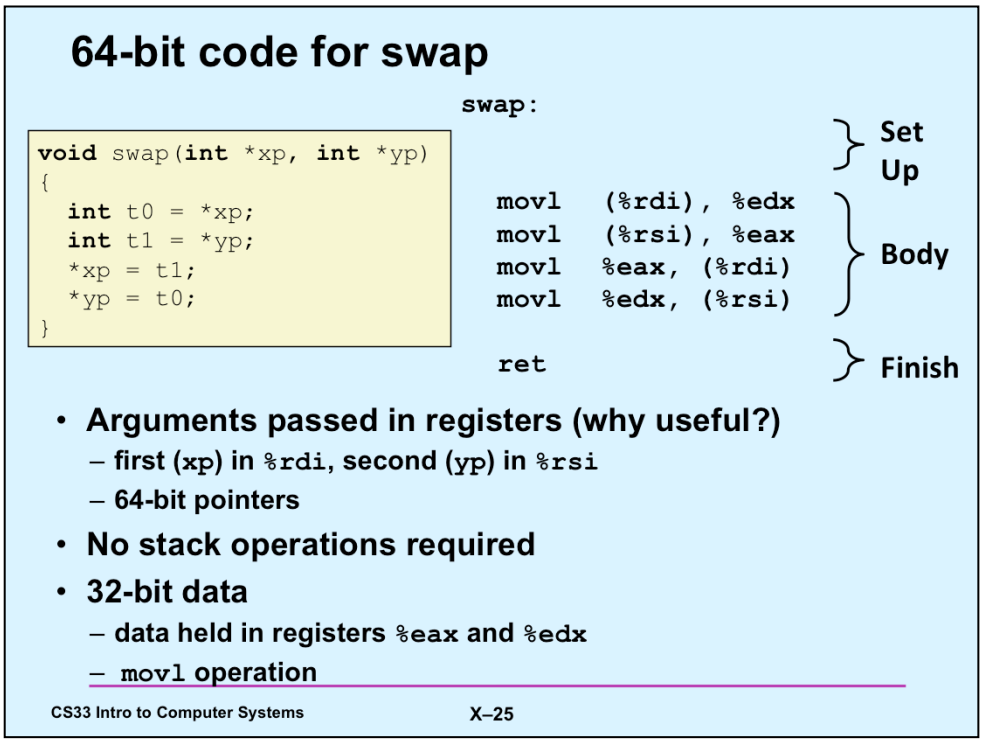
- addl 4(%rdx), %eax
  - memory address must be 64 bits
  - operands (in this case) are 32-bit
    - » result goes into %eax
      - lower half of %rax
      - upper half is filled with zeroes

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.



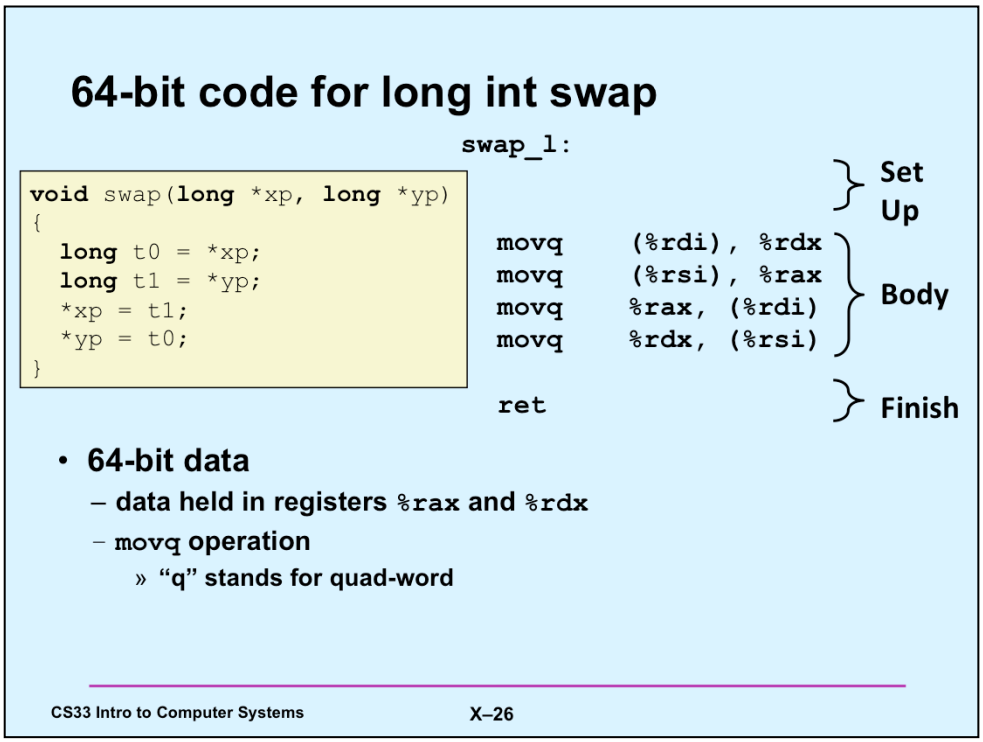
Supplied by CMU.

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

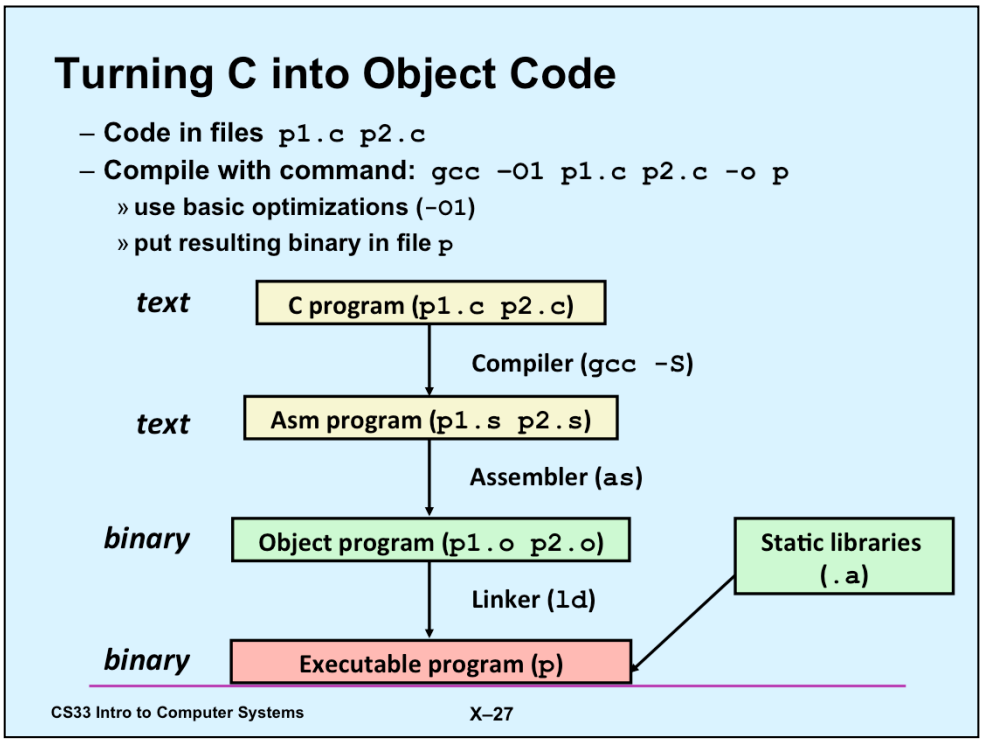


Supplied by CMU.

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.

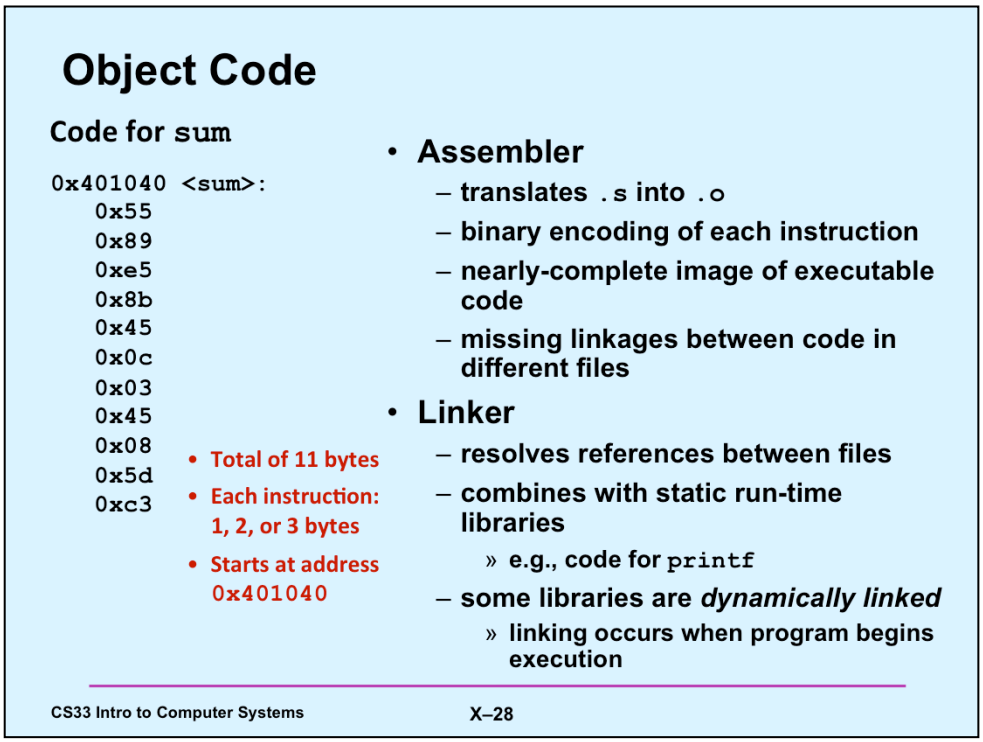


Supplied by CMU.



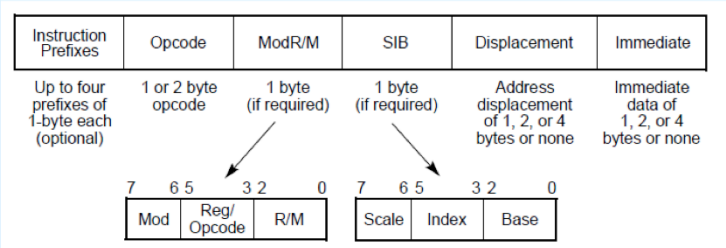
Supplied by CMU.

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.



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

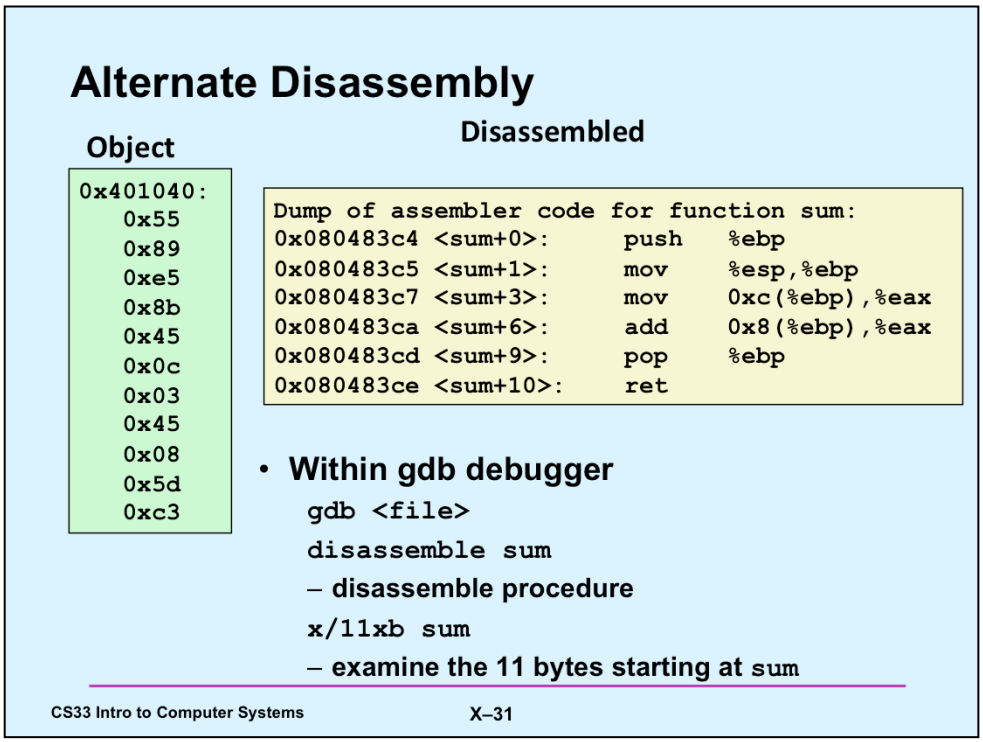
```
080483c4 <sum>:
 80483c4:   55          push   %ebp
 80483c5:   89 e5       mov    %esp,%ebp
 80483c7:   8b 45 0c    mov    0xc(%ebp),%eax
 80483ca:   03 45 08    add    0x8(%ebp),%eax
 80483cd:   5d          pop    %ebp
 80483ce:   c3          ret
```

- **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.



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:**

Format	Computation		
`addl`	Src,Dest	Dest = Dest + Src	
`subl`	Src,Dest	Dest = Dest - Src	
`imull`	Src,Dest	Dest = Dest * Src	
`sall`	Src,Dest	Dest = Dest << Src	Also called shll
`sarl`	Src,Dest	Dest = Dest >> Src	Arithmetic
`shrl`	Src,Dest	Dest = Dest >> Src	Logical
`xorl`	Src,Dest	Dest = Dest ^ Src	
`andl`	Src,Dest	Dest = Dest & Src	
`orl`	Src,Dest	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	= ~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.

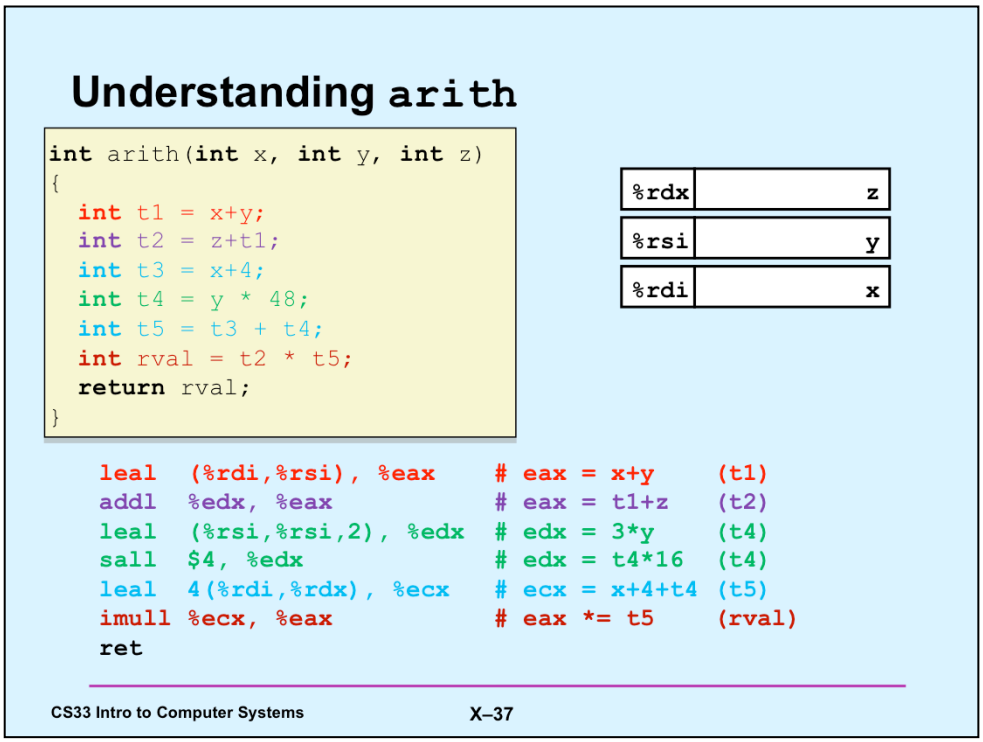
Understanding `arith`

```
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;
}
```

%rdx	z
%rsi	y
%rdi	x

```
    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`

```
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.

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