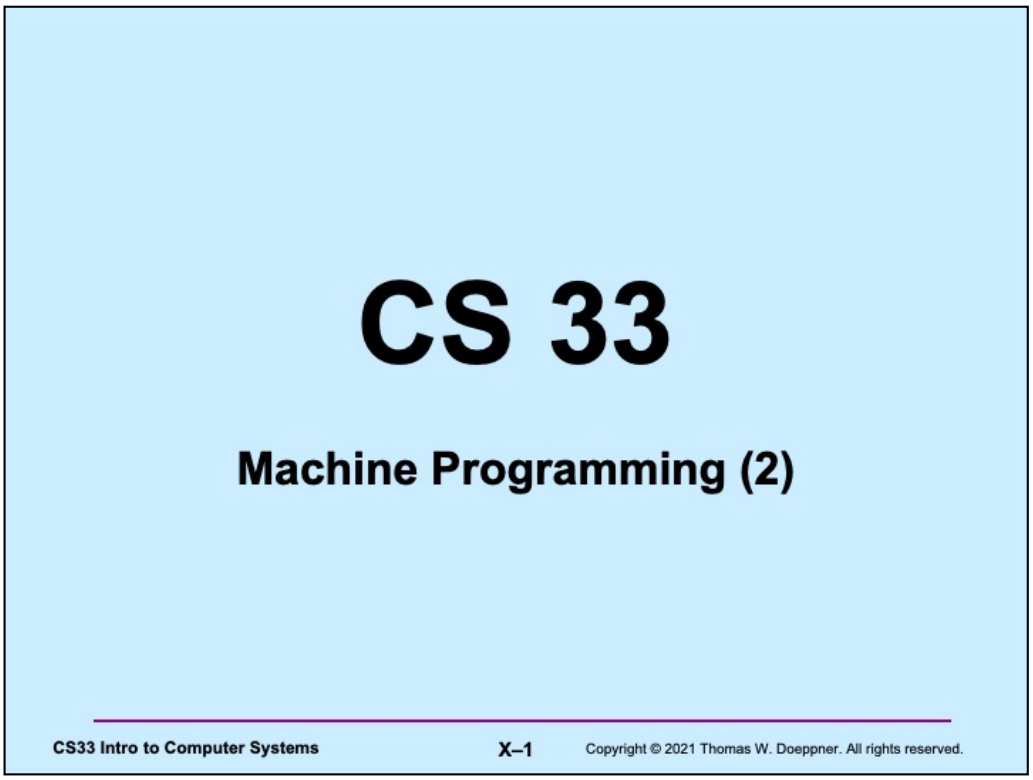
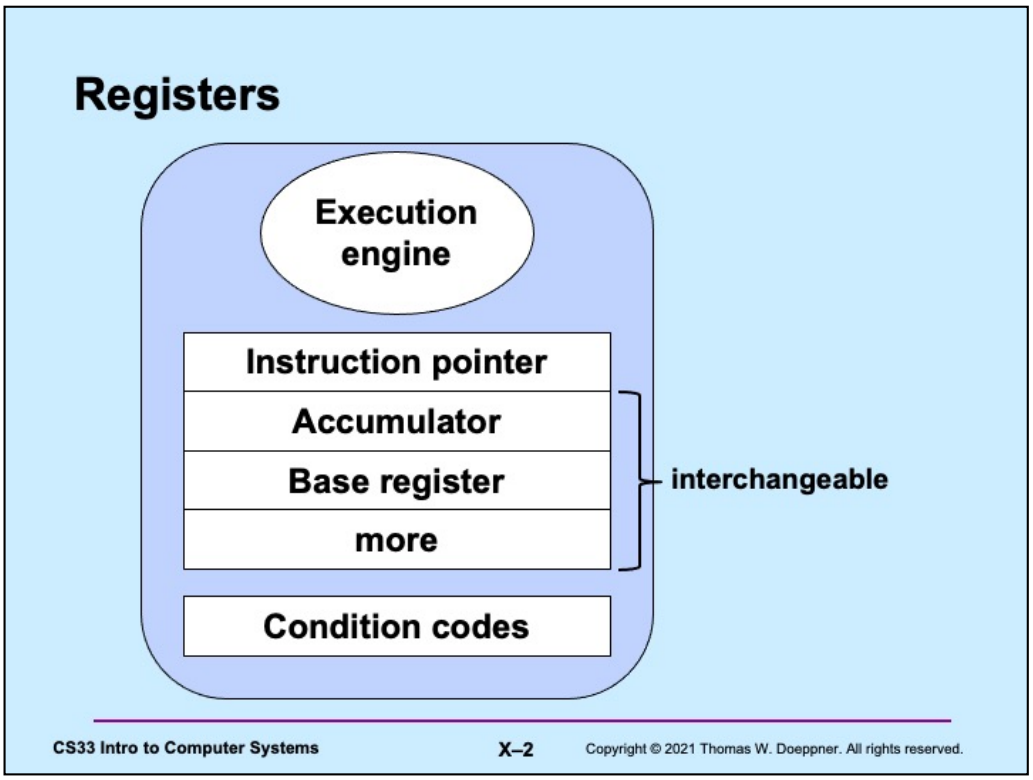
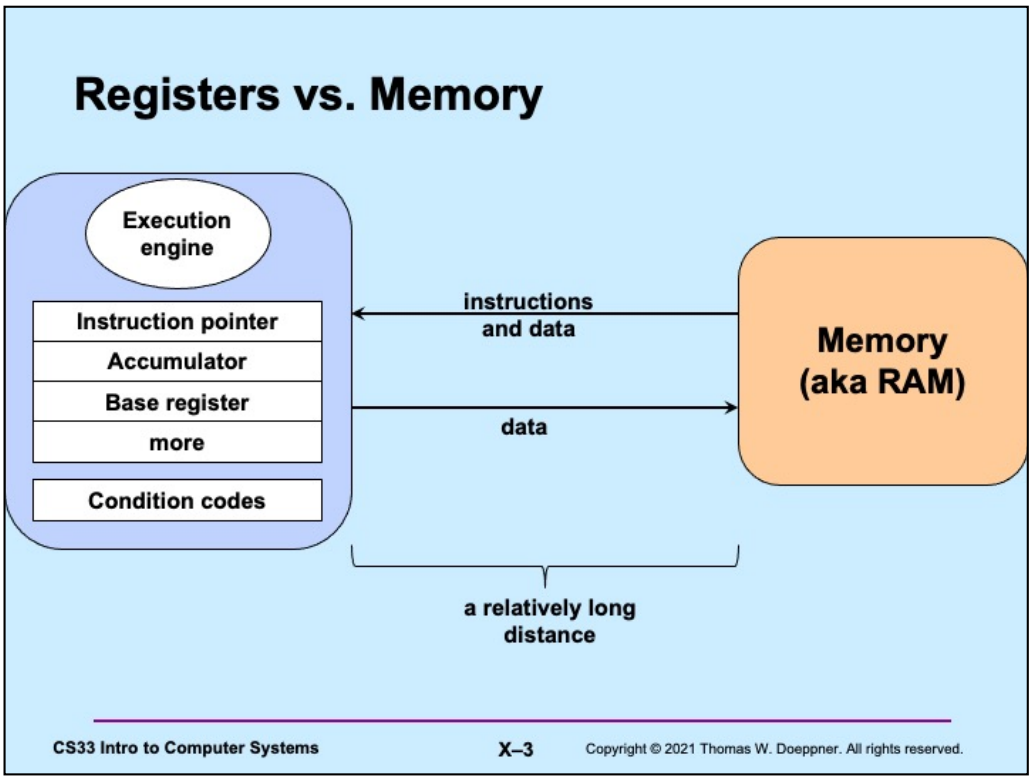


Many of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook "Computer Systems: A Programmer's Perspective," 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O'Hallaron in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

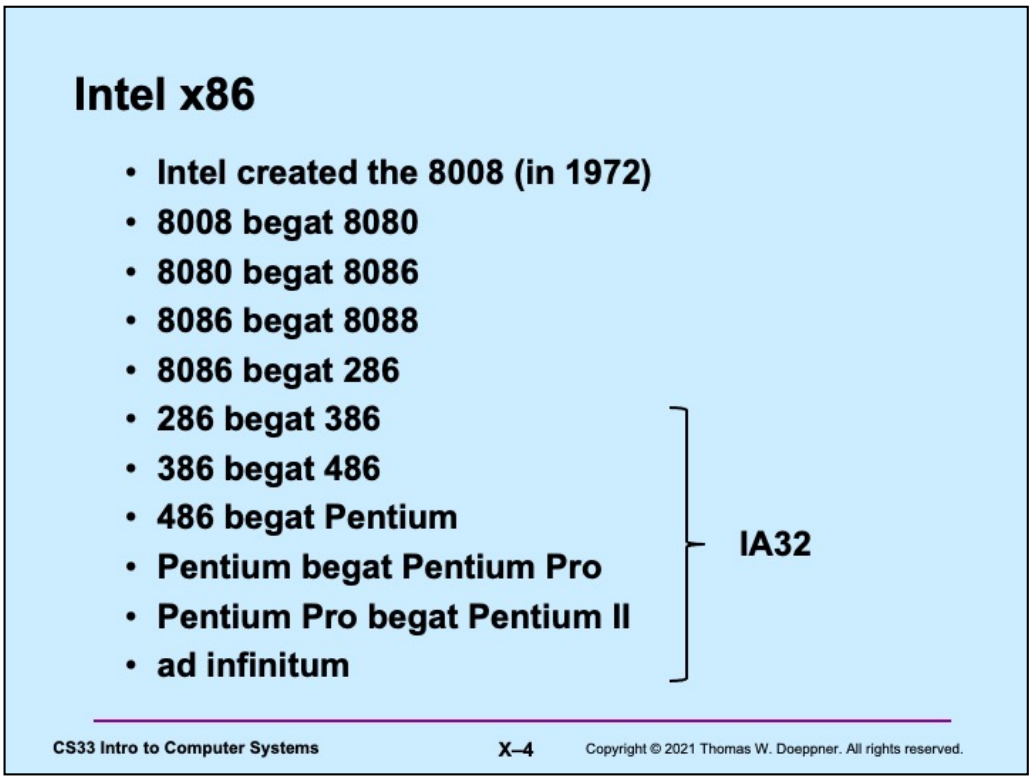


We've now seen four registers: the instruction pointer, the accumulator, the base register, and the condition codes. The accumulator is used to hold intermediate results for arithmetic; the base register is used to hold addresses for relative addressing. There's no particular reason why the accumulator 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 16; 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 in-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.

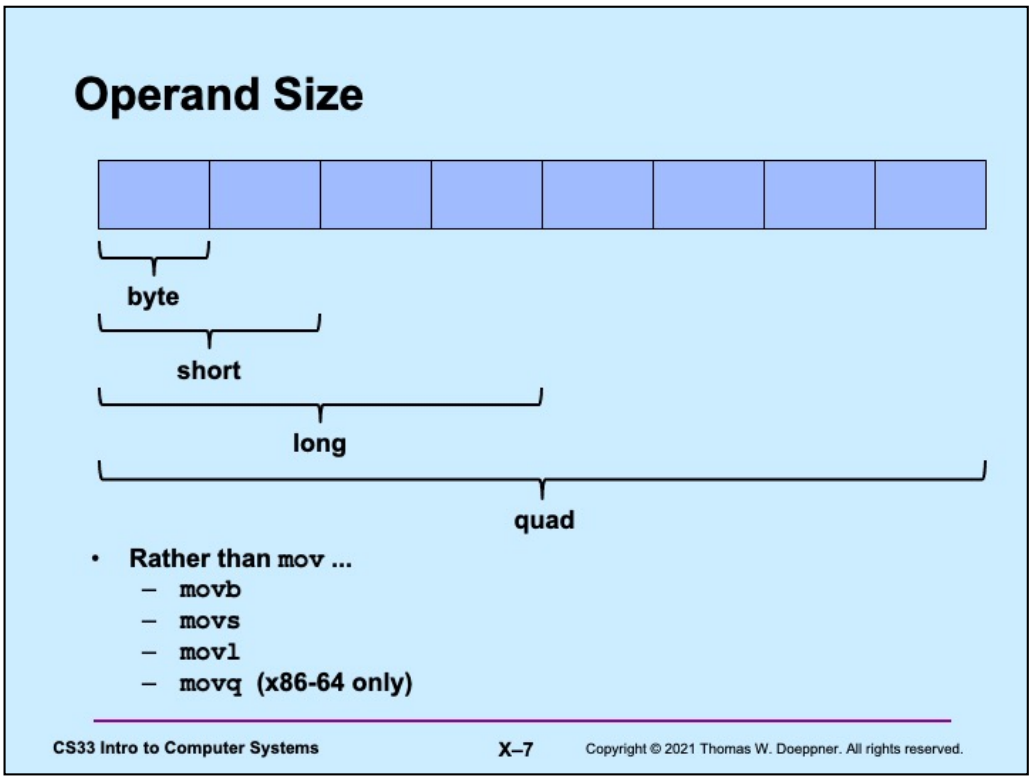
2^{64} = 16 exbibytes

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

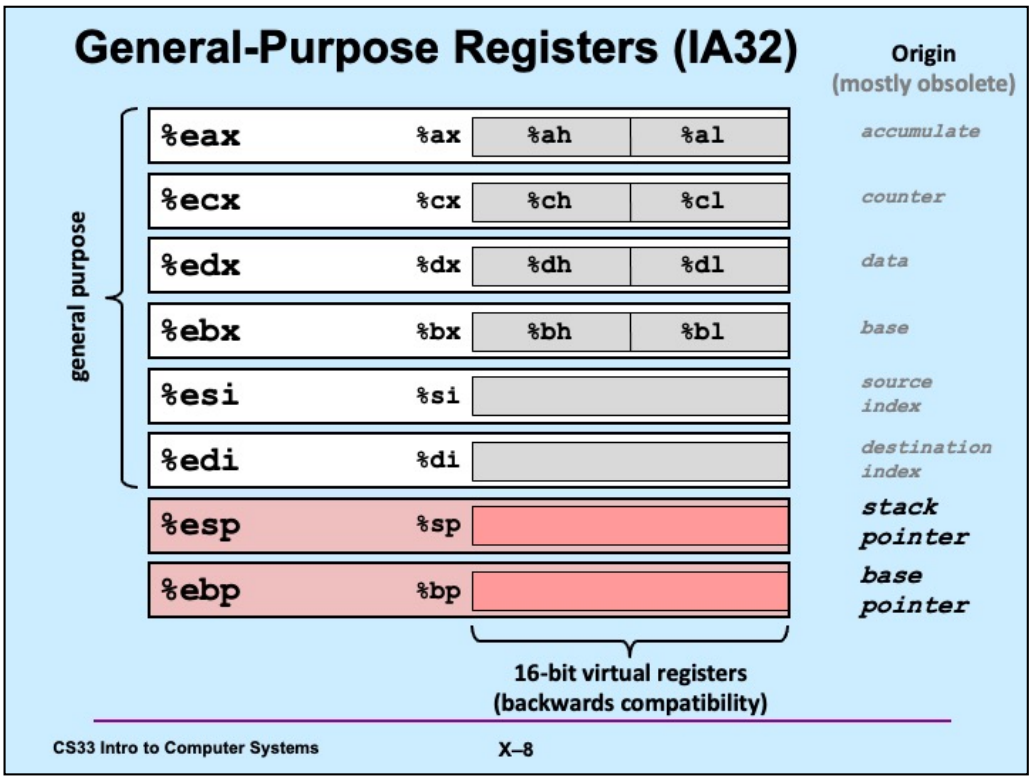
Supplied by CMU.



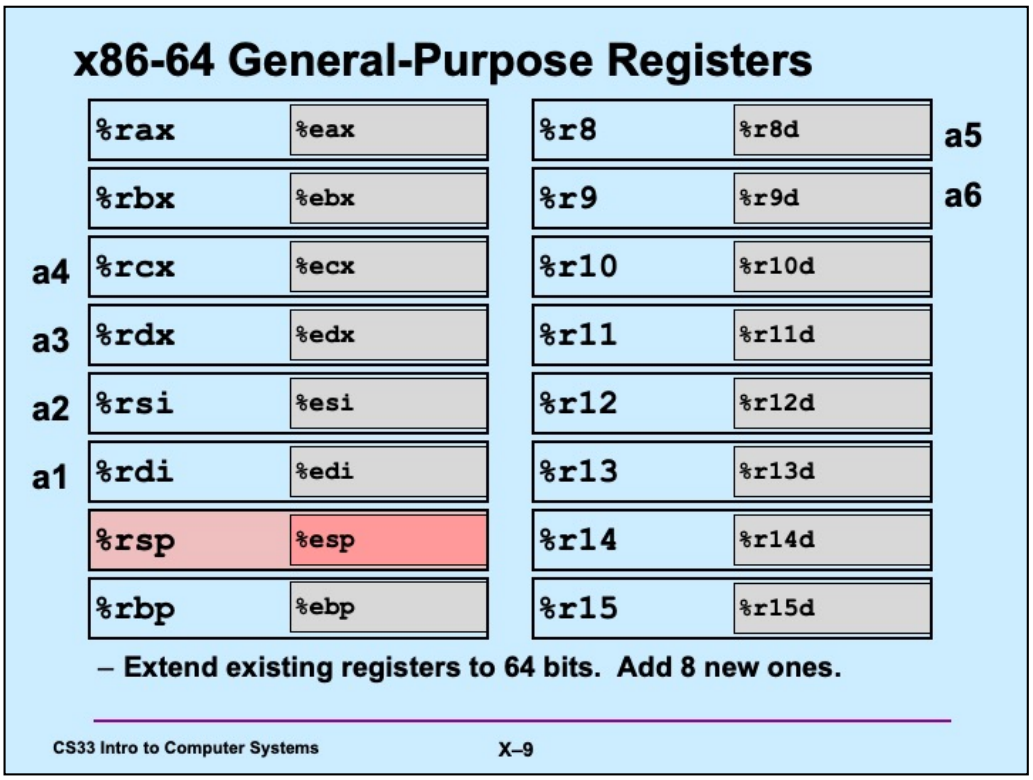
Most instructions come in three (on IA32) or four (on x86-64) forms, one for each possible operand size.

Note the confusion: long on x86 is 32 bits, but long in C is 64 bits.

Note that some assemblers (in particular, those of Microsoft and Intel) use a different syntax. Rather than tag the mnemonic for the instruction with the operand size, they tag the operands.



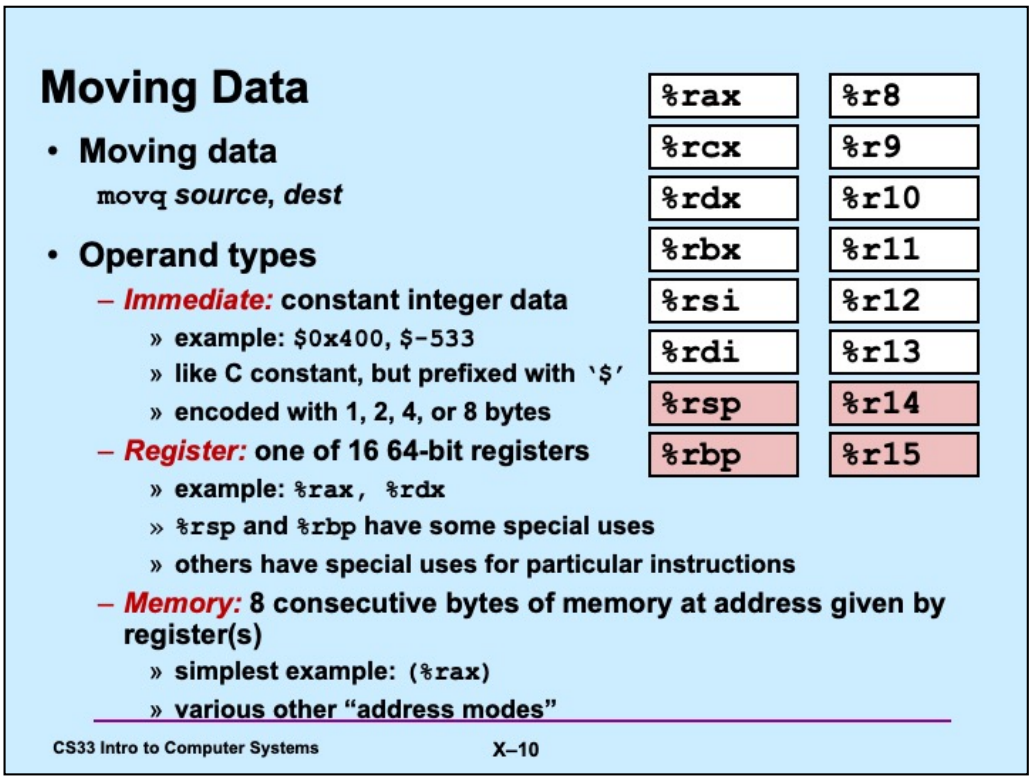
Supplied by CMU.



Supplied by CMU.

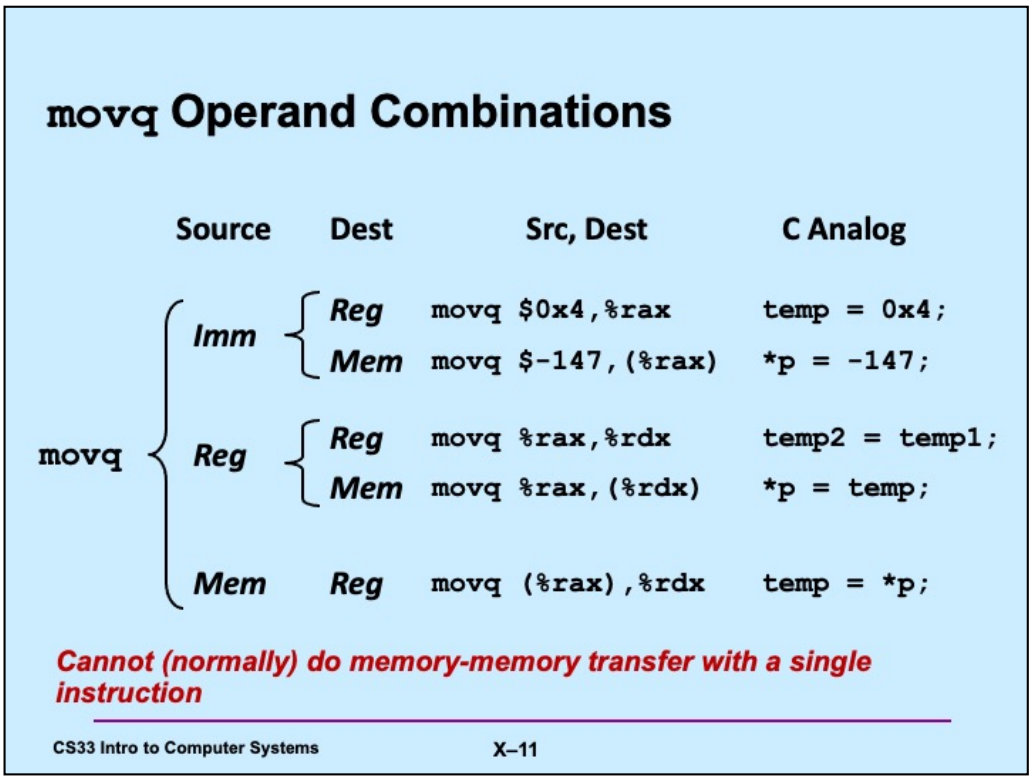
Note that %ebp/%rbp may be used as a base register as on IA32, but they don't have to be used that way. This will become clearer when we explore how the runtime stack is accessed. The convention on Linux is for the first 6 arguments of a function to be in registers %rdi, %rsi, %rdx, %rcx, %r8, and %r9. 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.

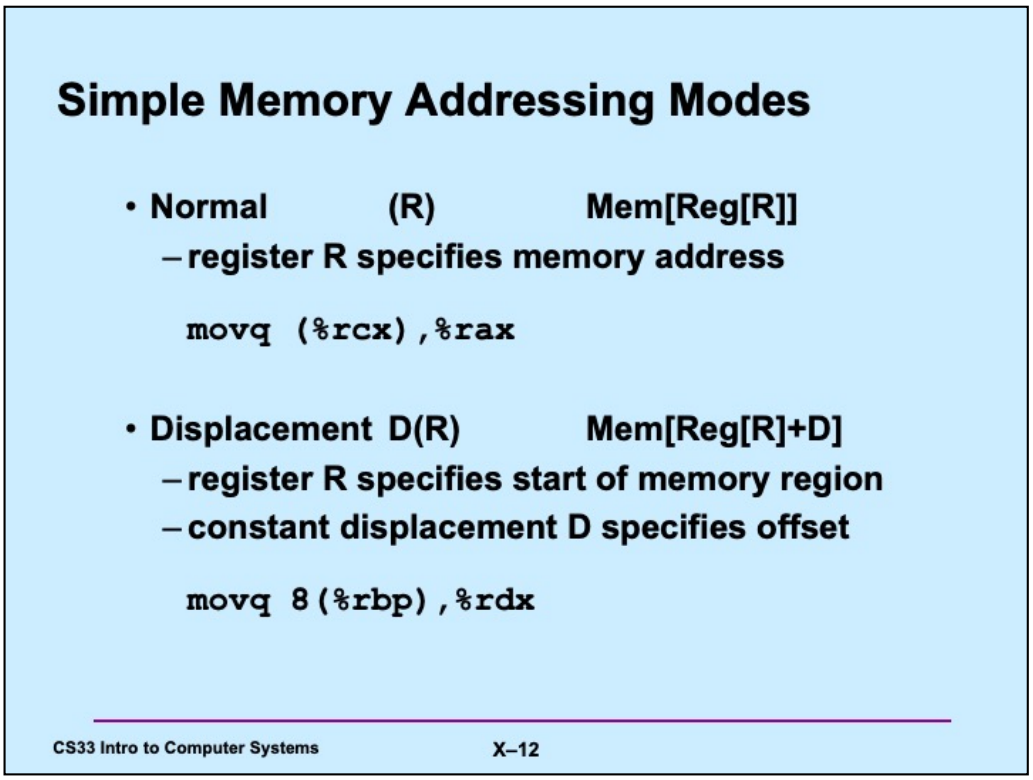


Based on a slide supplied by CMU.

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 %rax,8(%rbp)". 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.



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

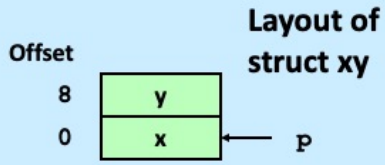
```
struct xy {
  long x;
  long y;
}
void swapxy(struct xy *p){
  long temp = p->x;
  p->x = p->y;
  p->y = temp;
}
```

```
swap:
  movq (%rdi), %rax
  movq 8(%rdi), %rdx
  movq %rdx, (%rdi)
  movq %rax, 8(%rdi)
  ret
```


Here we have a simple function that swaps the two components of a structure that's passed to it. (Assume that %rdi contains the argument.)

Understanding Swapxy

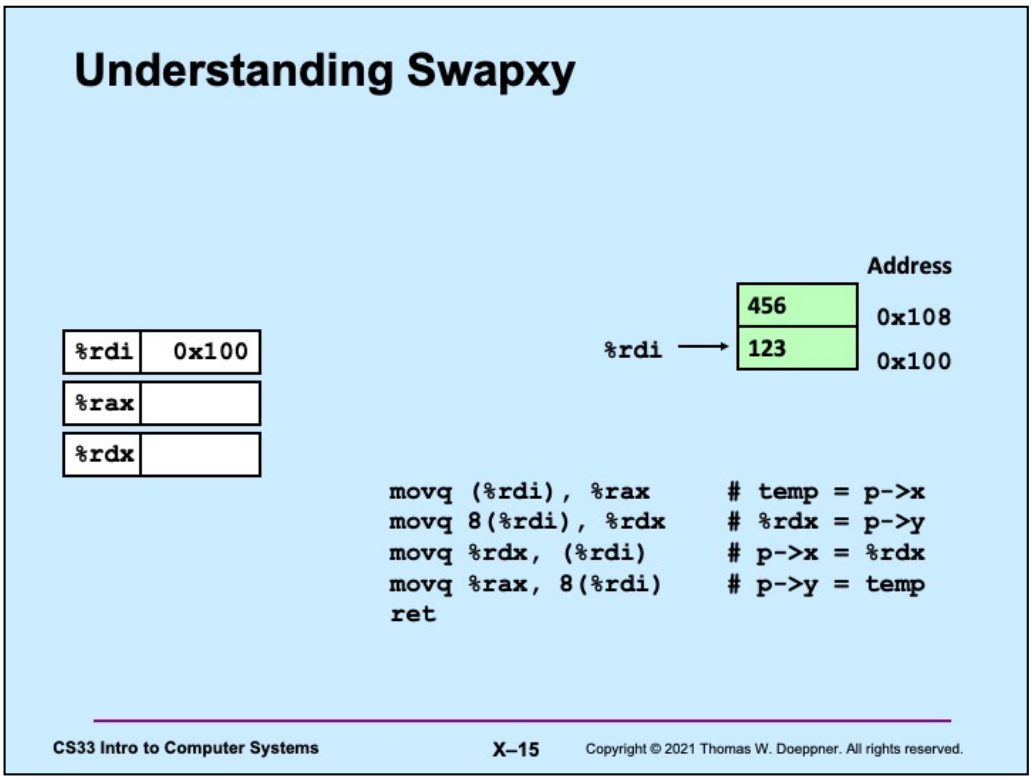
```
struct xy {
  long x;
  long y;
}
void swapxy(struct xy *p){
  long temp = p->x;
  p->x = p->y;
  p->y = temp;
}
```



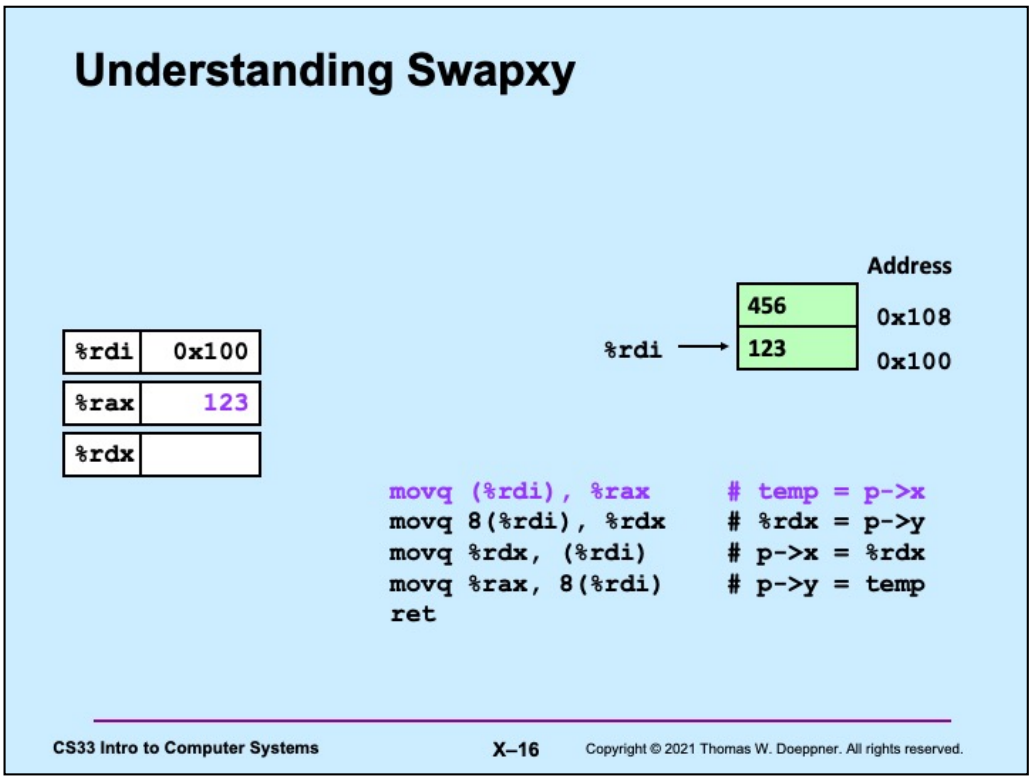
Register	Value
%rdi	p
%rax	temp
%rdx	p->y

```
movq (%rdi), %rax      # temp = p->x
movq 8(%rdi), %rdx     # %rdx = p->y
movq %rdx, (%rdi)      # p->x = %rdx
movq %rax, 8(%rdi)     # p->y = temp
ret
```

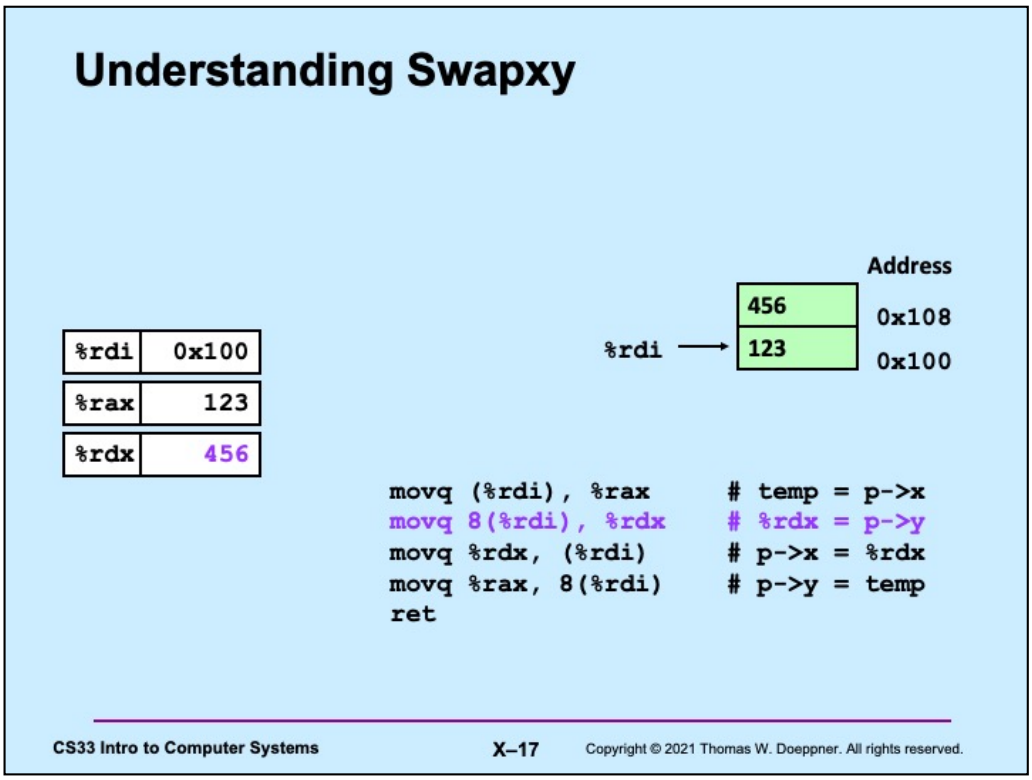

In addition to using %rdi to contain the argument (the address of the structure), we use %rax to contain the value of *temp* and %rdx to effectively be another temporary that holds the value of p->y.



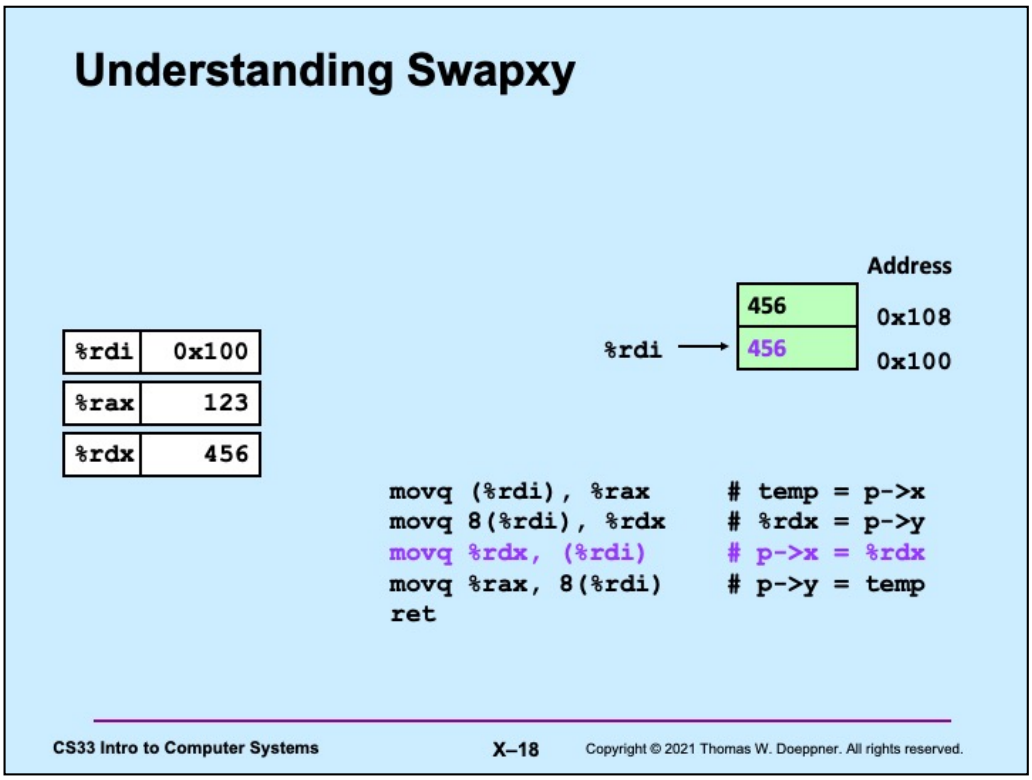
When we enter **swapxy**, %rdi contains the address of the structure.



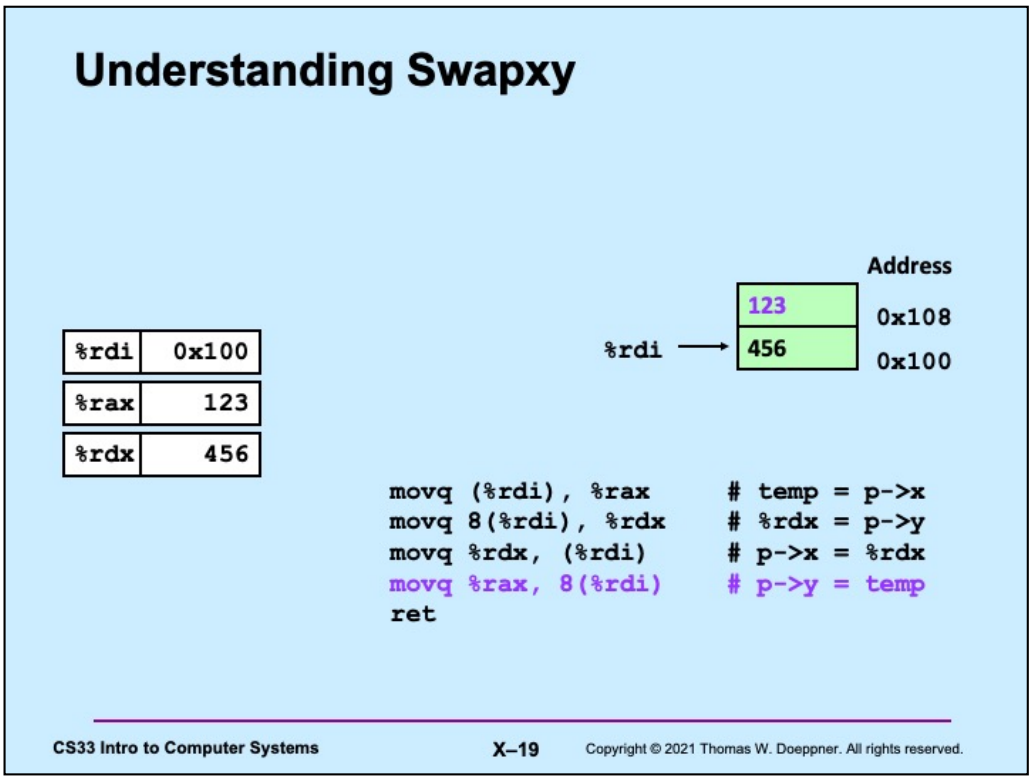
We copy the first component of p into **temp**, which is held in %rax.



We then copy the second component into %rdx.



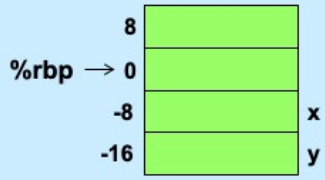
The second component, which we'd copied into %rdx, is now copied into the the first component of the structure itself.



Finally, we update the second component, copying into it what had been the first component.

Quiz 1

```
movq -8(%rbp), %rax
movq (%rax), %rax
movq (%rax), %rax
movq %rax, -16(%rbp)
```



Which C statements best describe the assembler code?

```
// a
long x;
long y;
y = x;
```

```
// b
long *x;
long y;
y = *x;
```

```
// c
long **x;
long y;
y = **x;
```

```
// d
long ***x;
long 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 16† registers
  - **Ri:** index register: any, except for `%rsp`
  - **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]**

†The instruction pointer may also be used (for a total of 17 registers)

Adapted from a slide supplied by CMU.

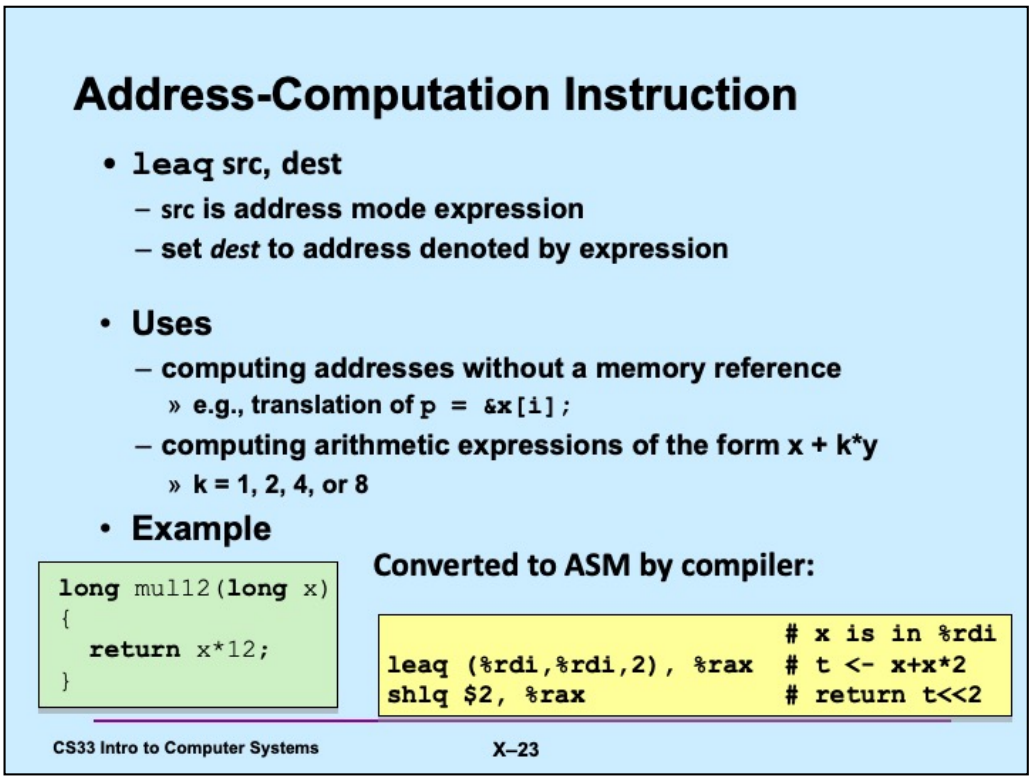
The instruction pointer is referred to as %rip. We'll see its use (in addressing) a bit later in the course.

Address-Computation Examples

%rdx	0xf000
%rcx	0x0100

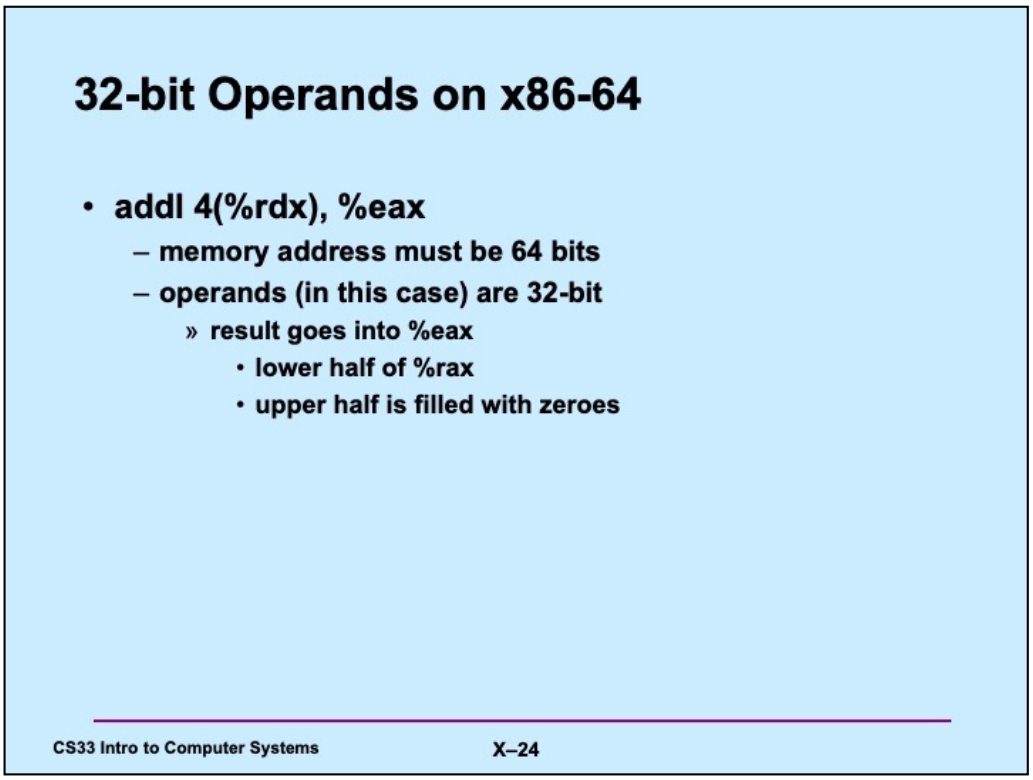
Expression	Address Computation	Address
0x8(%rdx)	0xf000 + 0x8	0xf008
(%rdx, %rcx)	0xf000 + 0x100	0xf100
(%rdx, %rcx, 4)	0xf000 + 4*0x0100	0xf400
0x80(,%rdx, 2)	2*0xf000 + 0x80	0x1e080

Adapted from a slide from CMU



Adapted from a slide supplied by CMU.

Note that a function returns a value by putting it 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.

Quiz 2

What value ends up in %ecx?

```
movq $1000,%rax
movq $1,%rbx
movl 2(%rax,%rbx,2),%ecx
```

a) 0x04050607
b) 0x07060504
c) 0x06070809
d) 0x09080706

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

Hint:

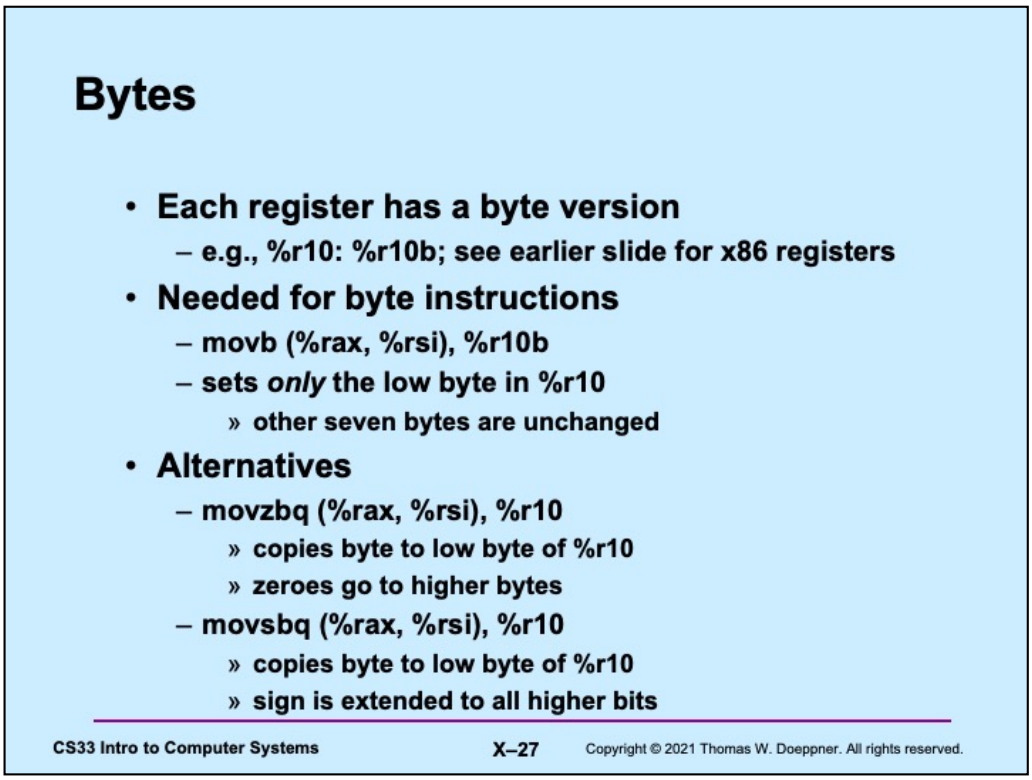
Swapxy for Ints

```
struct xy {
  int x;
  int y;
}
void swapxy(struct xy *p){
  int temp = p->x;
  p->x = p->y;
  p->y = temp;
}
```

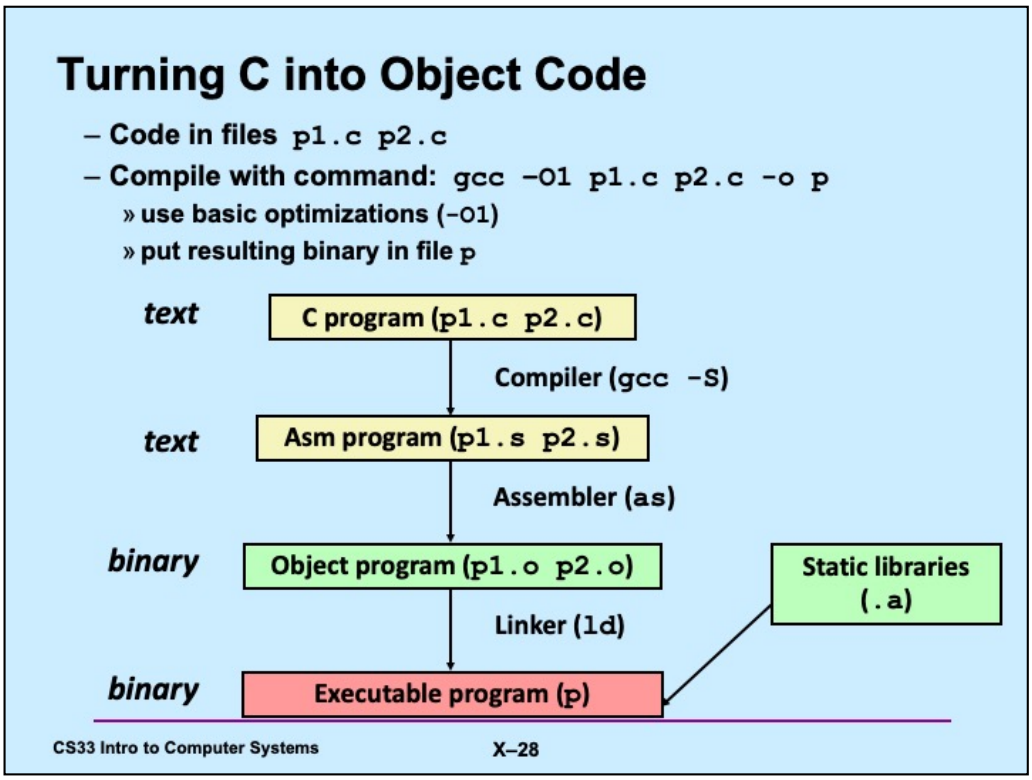
```
swap:
  movl (%rdi), %eax
  movl 4(%rdi), %edx
  movl %edx, (%rdi)
  movl %eax, 4(%rdi)
  ret
```

- Pointers are 64 bits
- What they point to are 32 bits

Here we have a simple function that swaps the two components of a structure that's passed to it. (Assume that %rdi contains the argument.) Note that even though we use the "e" form of the registers to hold the (32-bit) data, we need the "r" form to hold the 64-bit addresses.



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.

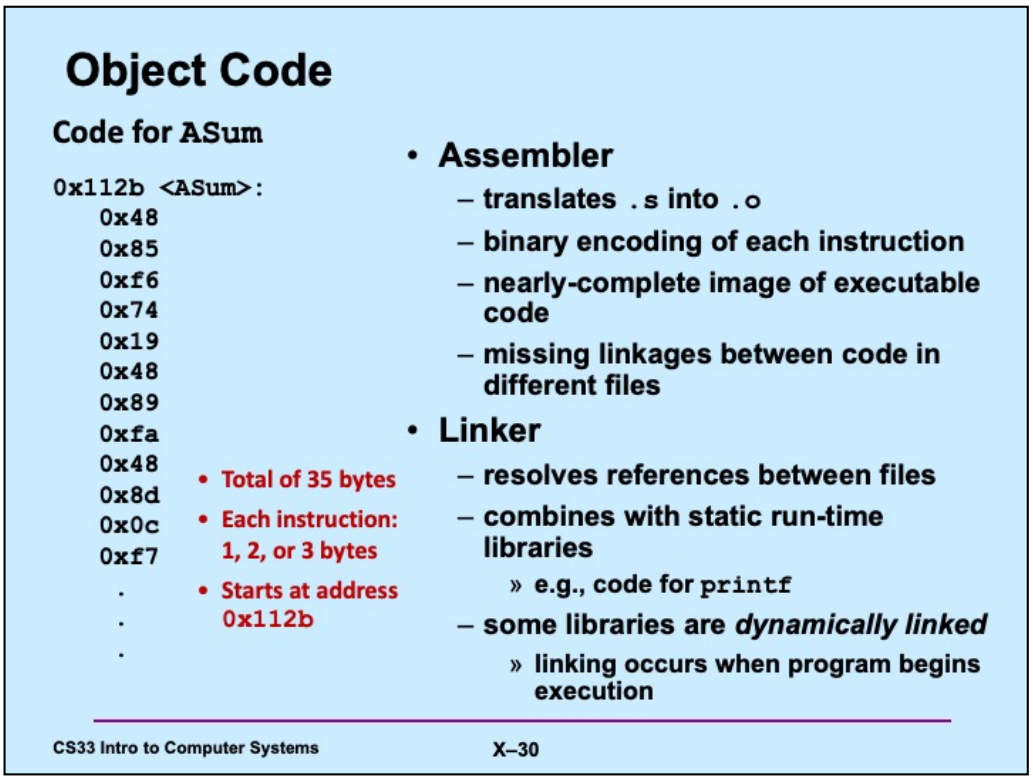


Supplied by CMU.

Note that 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.

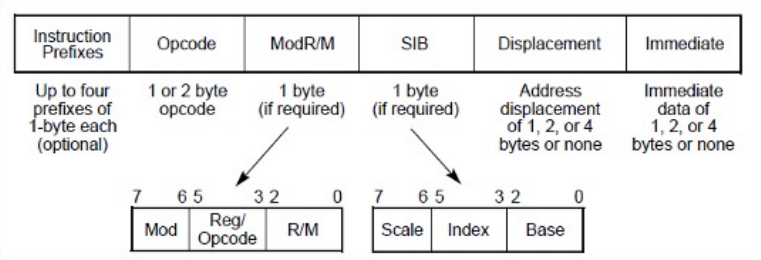
Example

```
long ASum(long *a, unsigned long size) {
   long i, sum = 0;
   for (i=0; i<size; i++)
      sum += a[i];
   return sum;
}
```

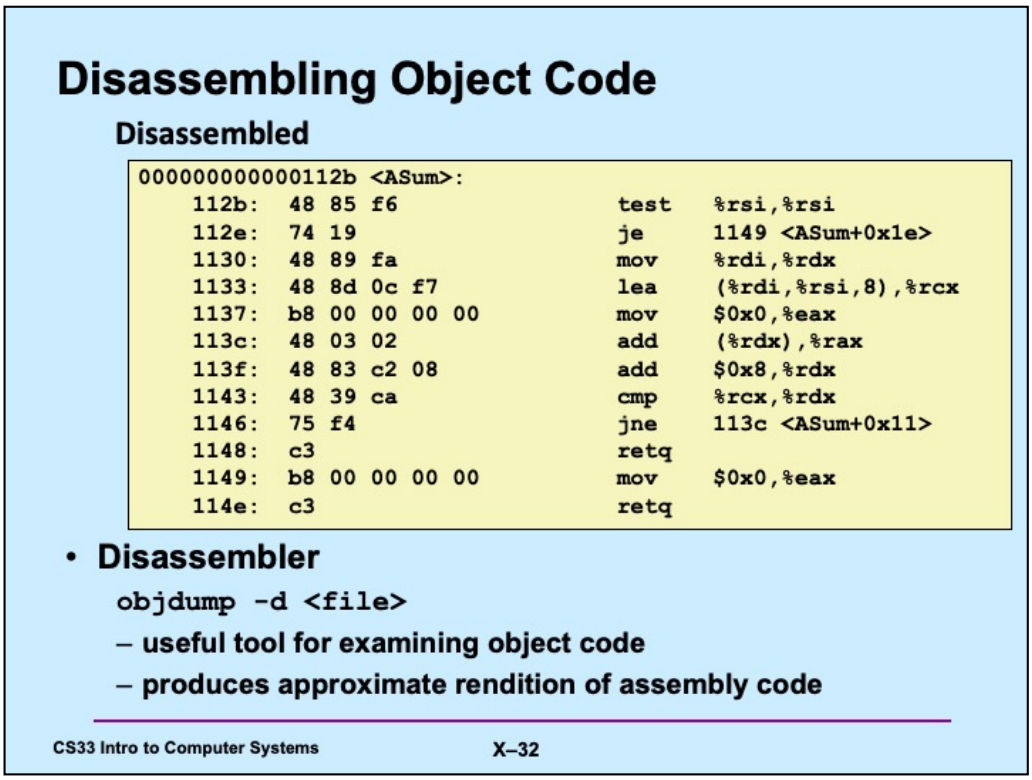



Adapted from a slide supplied by CMU.

Instruction Format

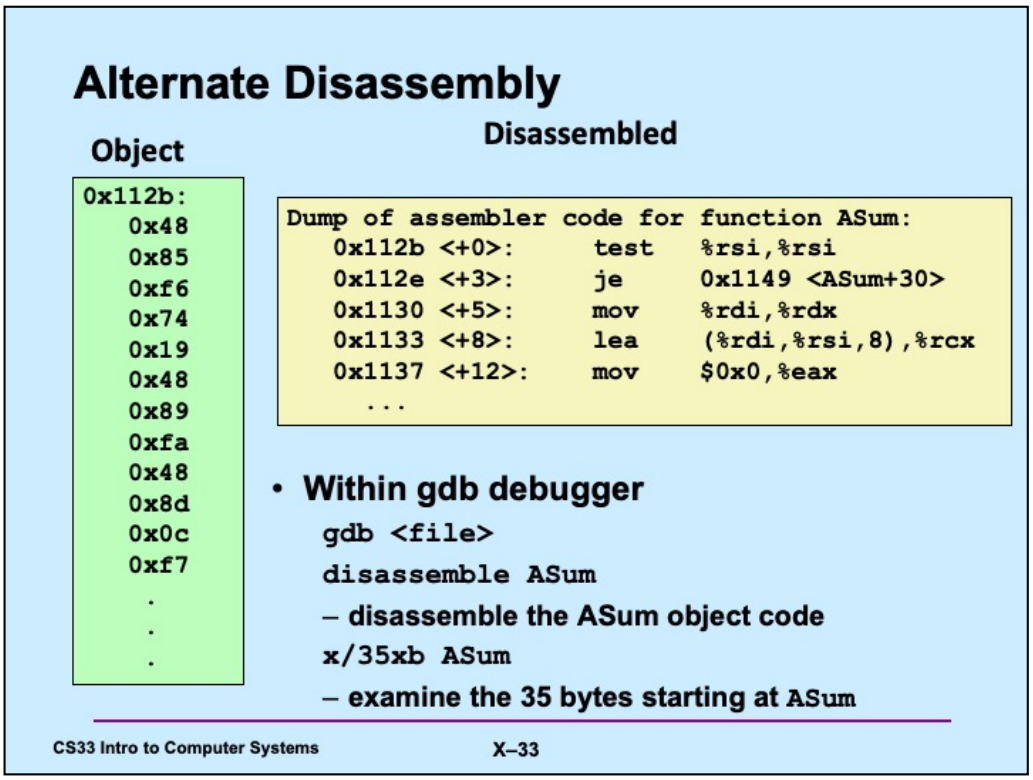


This is taken from Intel 64 and IA-32 Architecture Software Developer's Manual, Volume 2: Instruction Set Reference; Order Number 325462-043US, Intel Corporation, May 2012 (https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4)



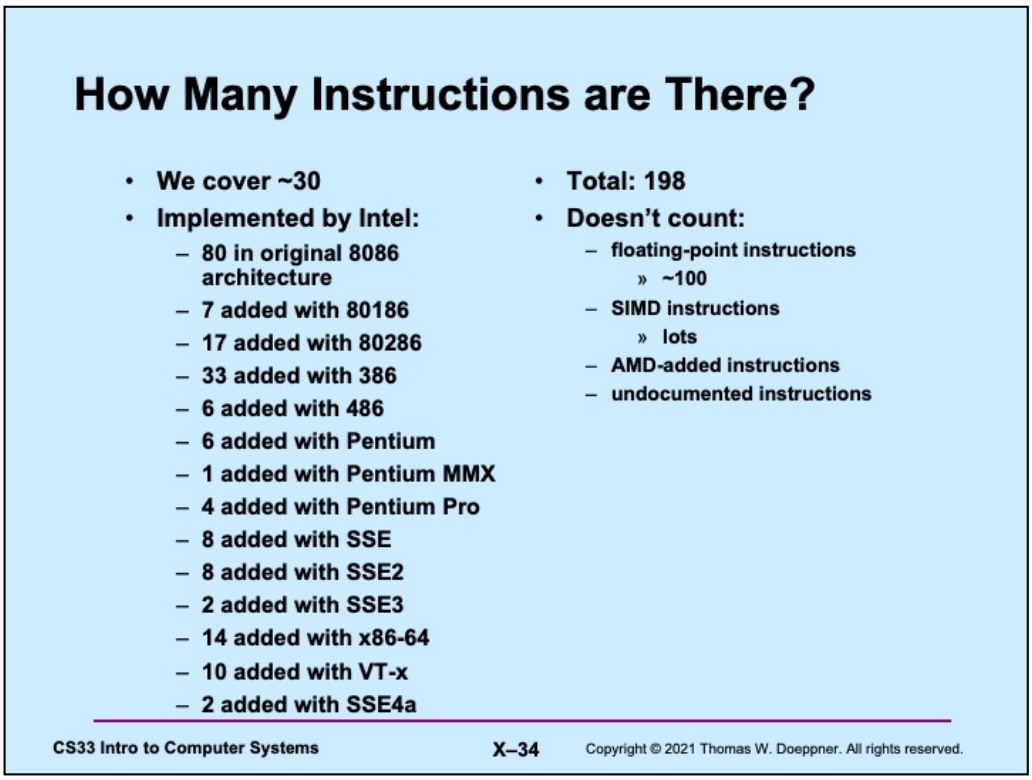
Adapted from a slide supplied by CMU.

objdump's rendition is approximate because it assumes everything in the file is assembly code, and thus translates data into (often really weird) assembly code.

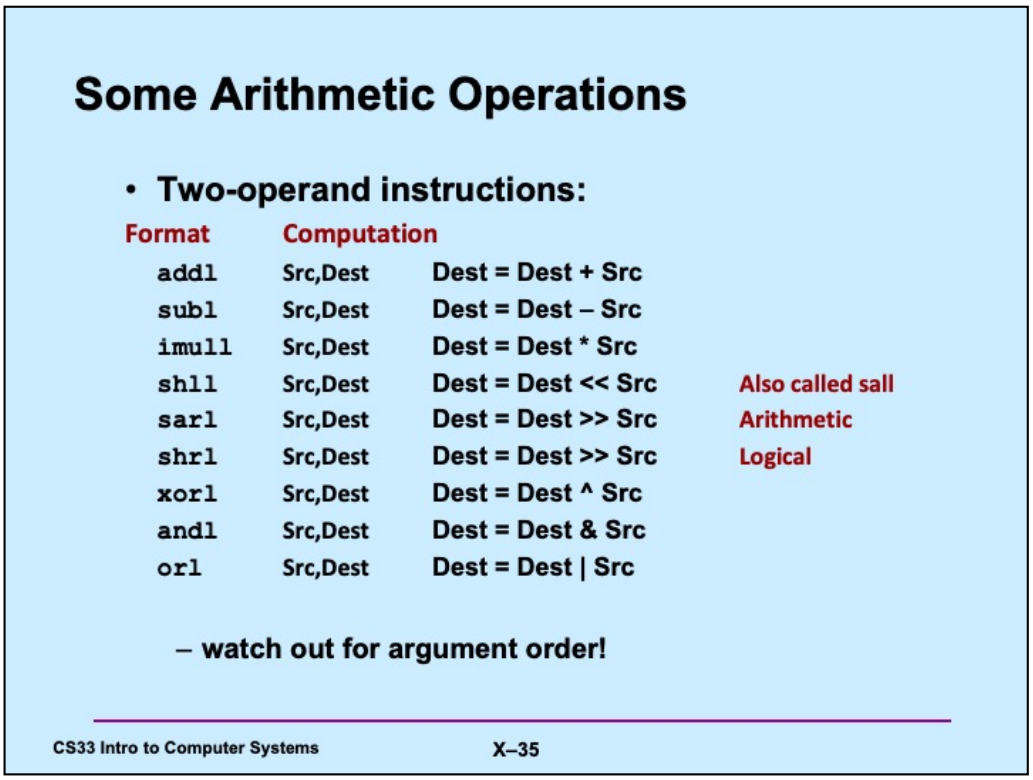


Adapted from a slide supplied by CMU.

The "x/35xb" directive to gdb says to examine (first x, meaning print) 35 bytes (b) viewed as hexadecimal (second x) starting at ASum.



The source for this is http://en.wikipedia.org/wiki/X86_instruction_listings, viewed on 6/20/2017, which came with the caveat that it may be out of date. While it's likely that more instructions have been added since then, we won't be covering them in 33!



Supplied by CMU.

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

Also note that what's given in the slide are the versions for 32-bit operands. There are also versions for 8-, 16-, and 64-bit operands, with the "l" replaced with the appropriate letter ("b", "s", or "q").

Some Arithmetic Operations

- **One-operand Instructions**

`incl`	Dest	= Dest + 1
`decl`	Dest	= Dest – 1
`negl`	Dest	= – Dest
`notl`	Dest	= ~Dest

- **See textbook for more instructions**
- **See Intel documentation for even more**

Adapted from a slide 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
    shll   $4, %edx
    leal   4(%rdi,%rdx), %ecx
    imull  %ecx, %eax
    ret
```


Supplied by CMU, but converted to x86-64.

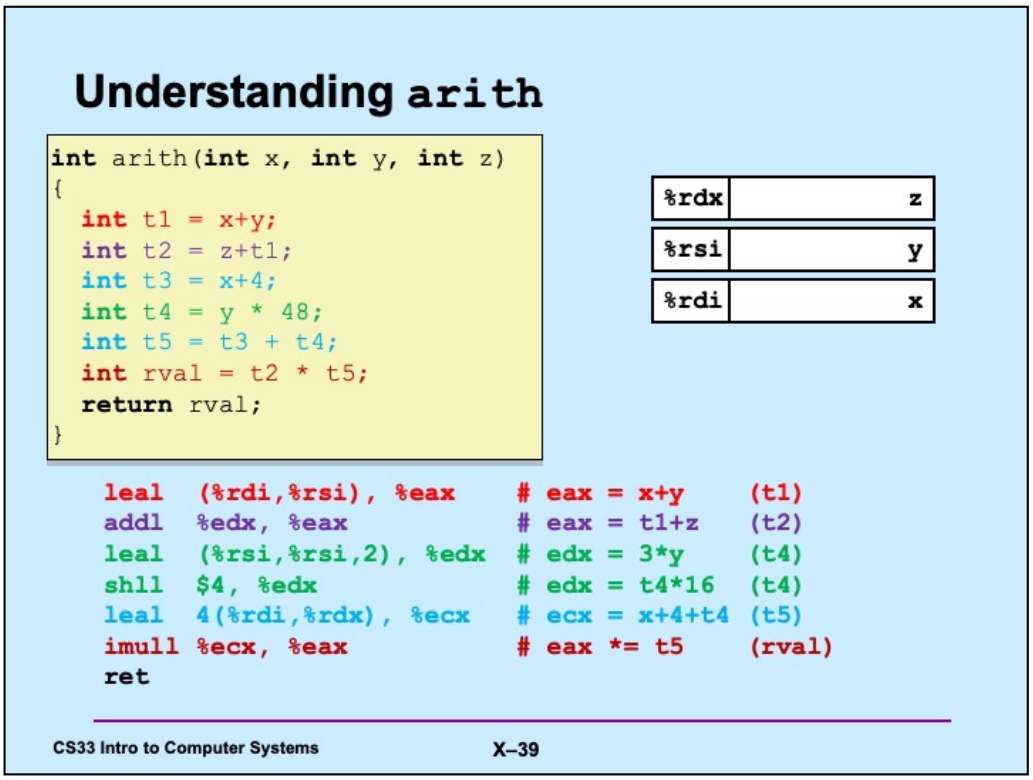
Understanding `arith`

```
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
  shll  $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 function are placed in registers **rdi**, **rsi**, and **rdx**, respectively. Note that, also by convention, functions put their return values in register **eax**/**rax**.

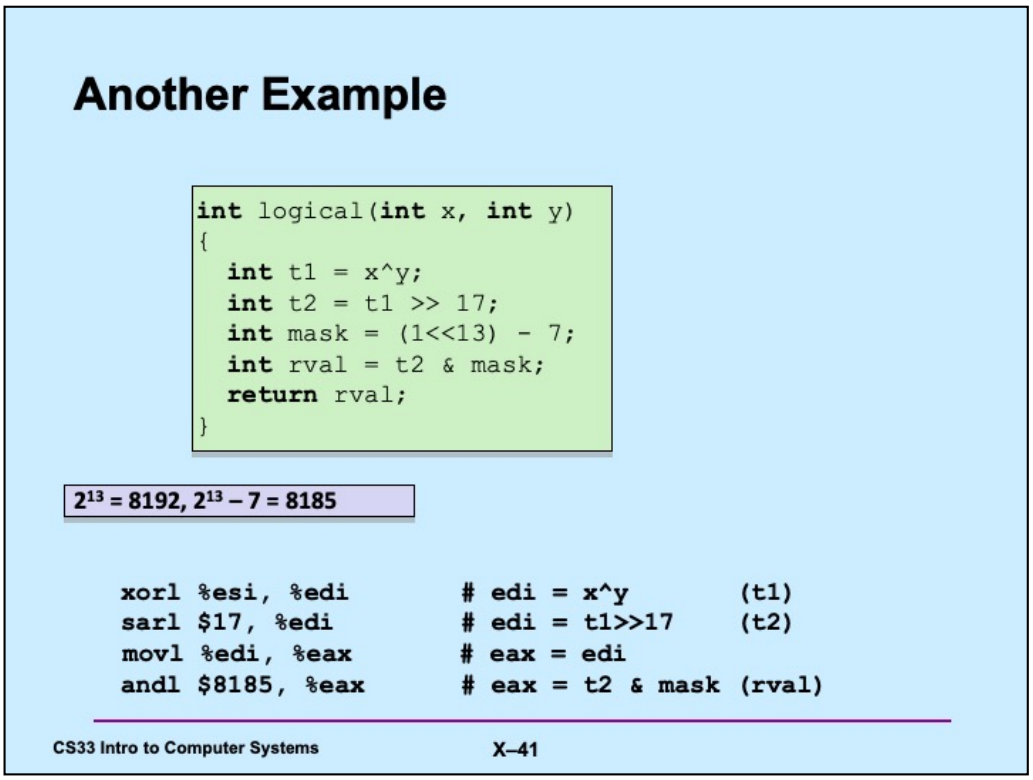
Observations about `arith`

```
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)
shll   $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.



Supplied by CMU, but converted to x86-64.

Quiz 3

- **What is the final value in %ecx?**

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

a) 0
b) 2
c) 4
d) 8

Note that xor'ing anything with itself results in 0.