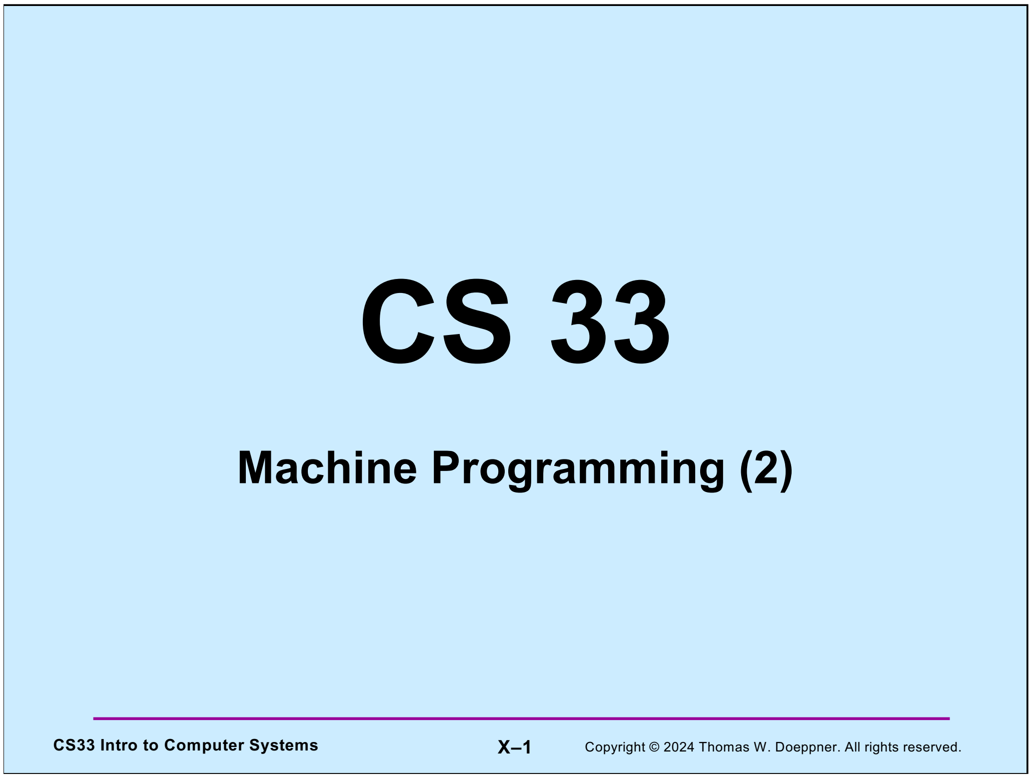
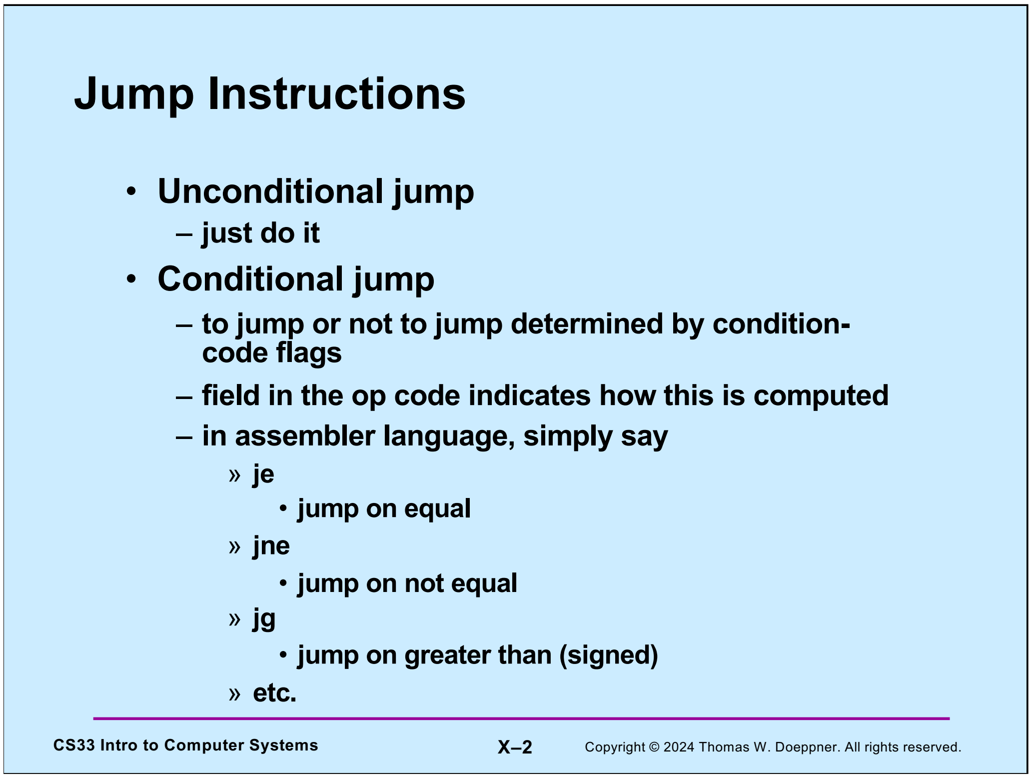


CS 33

Machine Programming (2)

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.

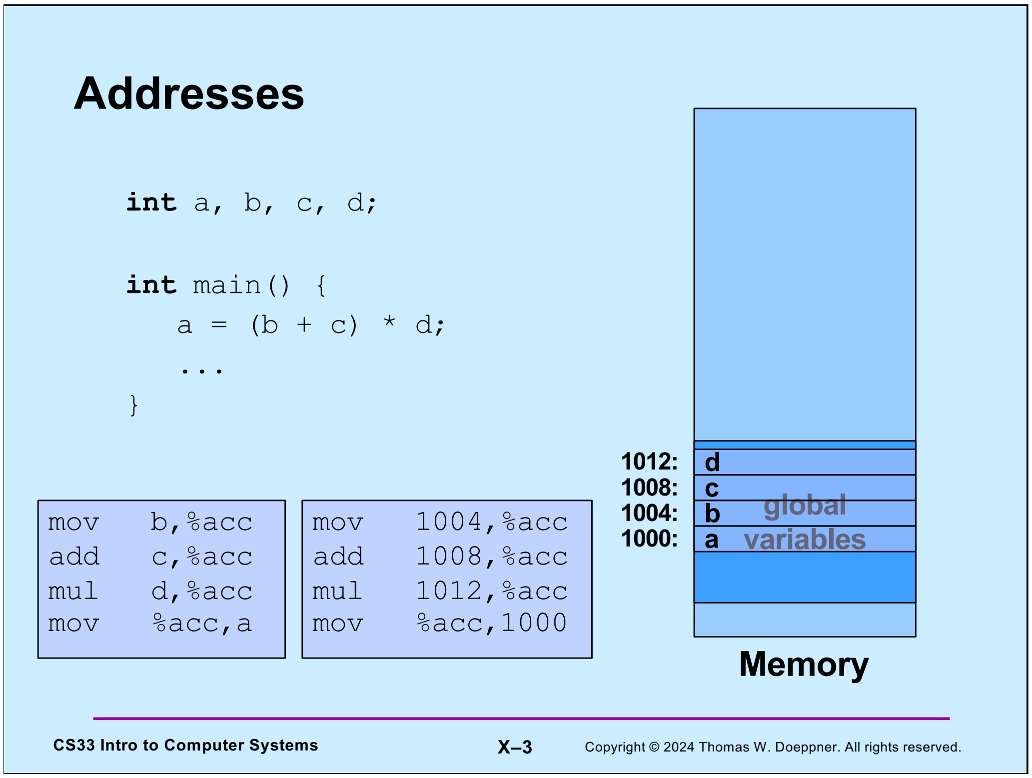


Jump Instructions

- 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
    - » jg
      - jump on greater than (signed)
    - » 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 an upcoming lecture, when we're looking at x86 assembler instructions.



Addresses

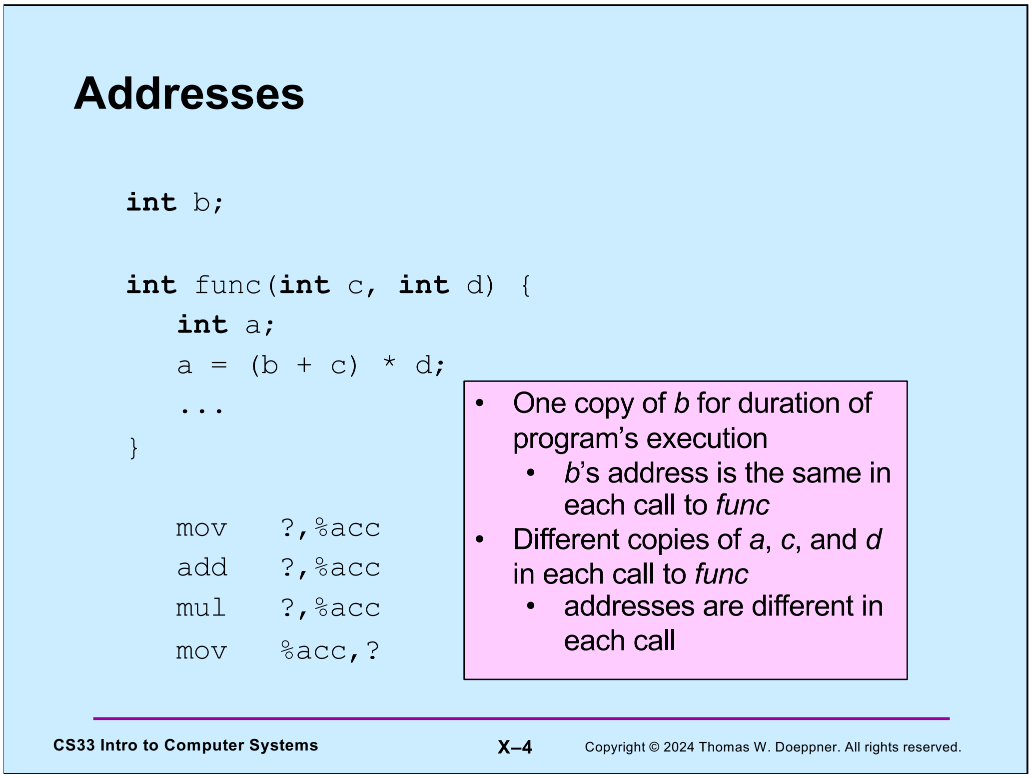
```
int a, b, c, d;

int main() {
   a = (b + c) * d;
   ...
}
```

```
mov    b,%acc
add    c,%acc
mul    d,%acc
mov    %acc,a
```

```
mov    1004,%acc
add    1008,%acc
mul    1012,%acc
mov    %acc,1000
```

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.

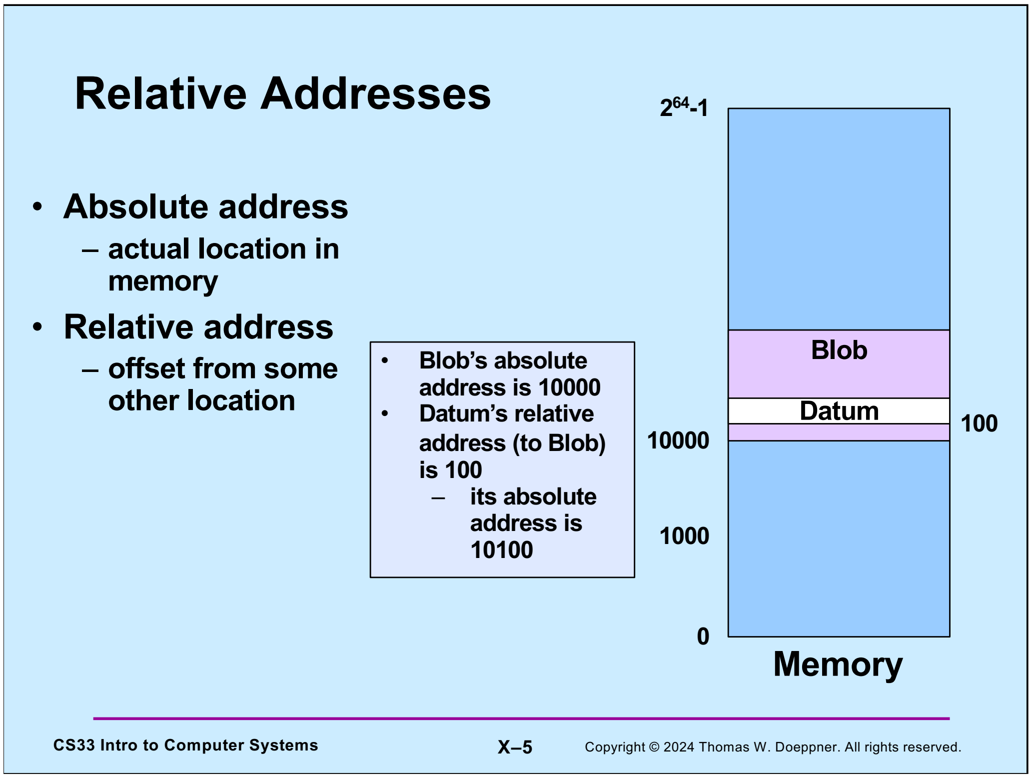


Addresses

```
int b;

int func(int c, int d) {
   int a;
   a = (b + c) * d;
   ...
}

   mov   ?,%acc
   add   ?,%acc
   mul   ?,%acc
   mov   %acc,?
```

- One copy of *b* for duration of program's execution
  - *b*'s address is the same in each call to *func*
- Different copies of *a*, *c*, and *d* in each call to *func*
  - addresses are different in each call

Here we rearrange things a bit. **b** is a global variable, but a is a local variable within **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 **func**. Thus, we somehow must modify the assembler code to take this into account.

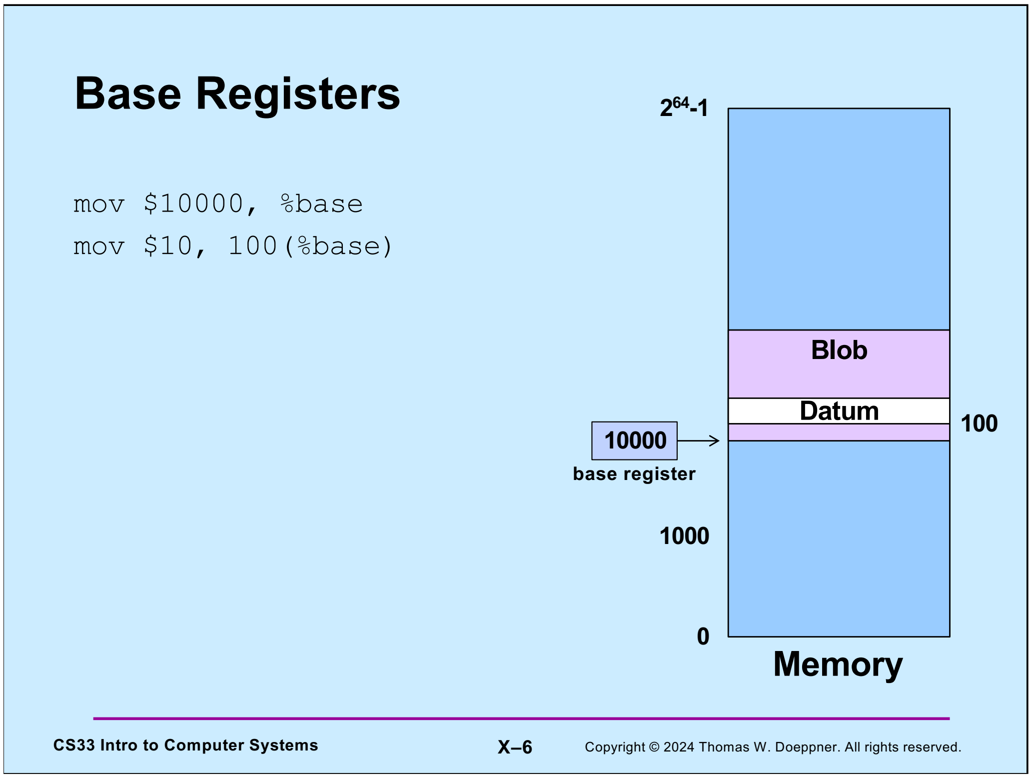


Relative Addresses

- **Absolute address**
  - actual location in memory
- **Relative address**
  - offset from some other location

- Blob's absolute address is 10000
- Datum's relative address (to Blob) is 100
  - its absolute address is 10100

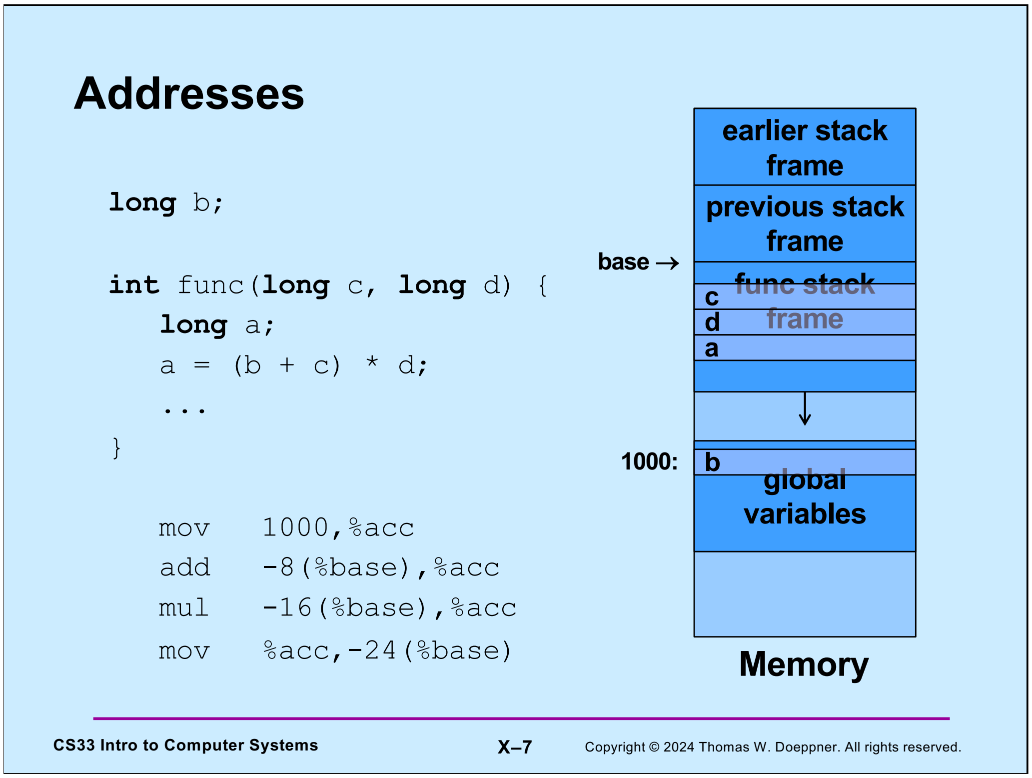
Note that both positive and negative offsets might be used.



Base Registers

```
mov $10000, %base
mov $10, 100(%base)
```

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(%base)** means the address obtained by adding **n** to the contents of the base register.



Addresses

```
long b;

int func(long c, long d) {
   long a;
   a = (b + c) * d;
   ...
}

   mov   1000,%acc
   add   -8(%base),%acc
   mul   -16(%base),%acc
   mov   %acc,-24(%base)
```

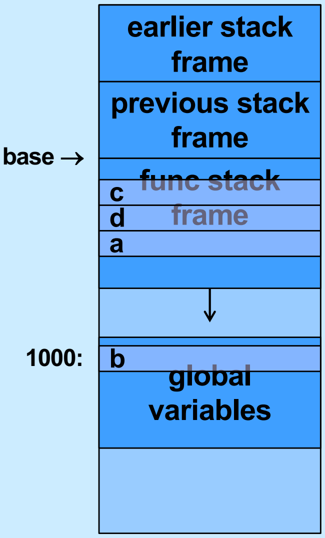
Here we return to our earlier example. We assume that, as part of the call to **func**, the base register is loaded with the address of the beginning of **func**'s current stack frame, and that the local variable **a** and the parameters **c** and **d** are located within the frame. Thus, we refer to them by their offset from the beginning of the stack frame, which are assumed to be **-24**, **-8**, and **-16**. 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*?

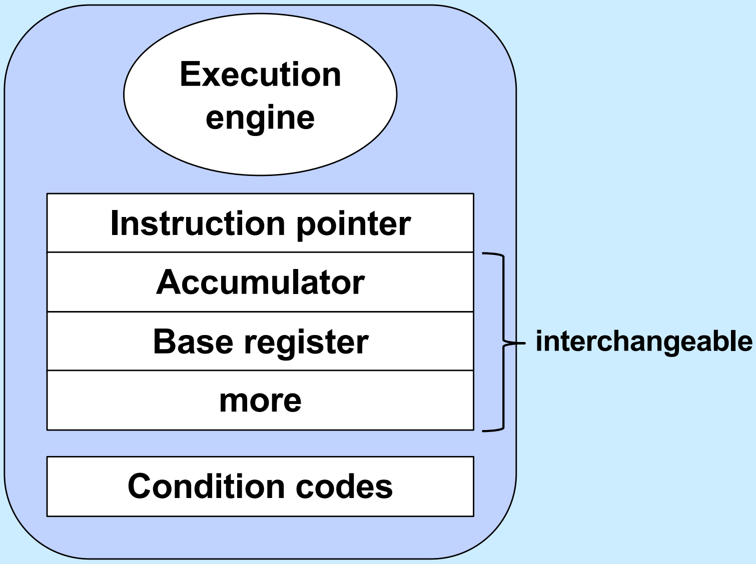
a) 9984
b) 9992
c) 10,008
d) 10,016

```
mov    1000,%acc
add    -8(%base),%acc
mul    -16(%base),%acc
mov    %acc,-24(%base)
```



Memory

Registers



Execution engine

Instruction pointer

Accumulator

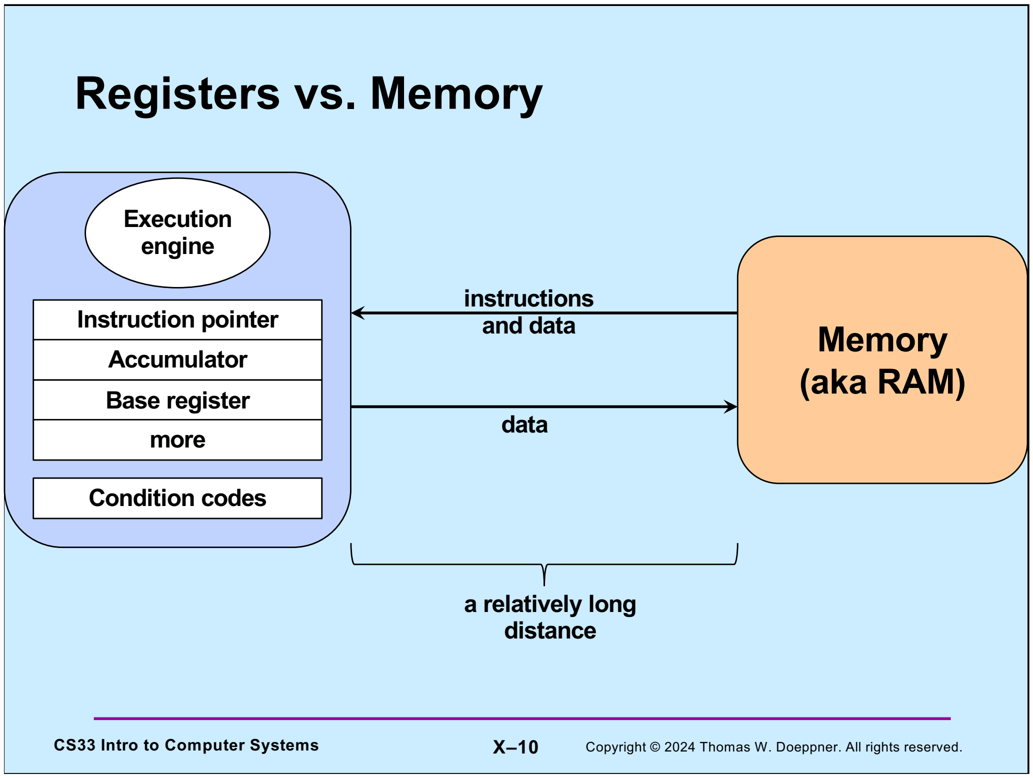
Base register

more

interchangeable

Condition codes

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.



Registers vs. Memory

Execution engine

Instruction pointer

Accumulator

Base register

more

Condition codes

instructions and data

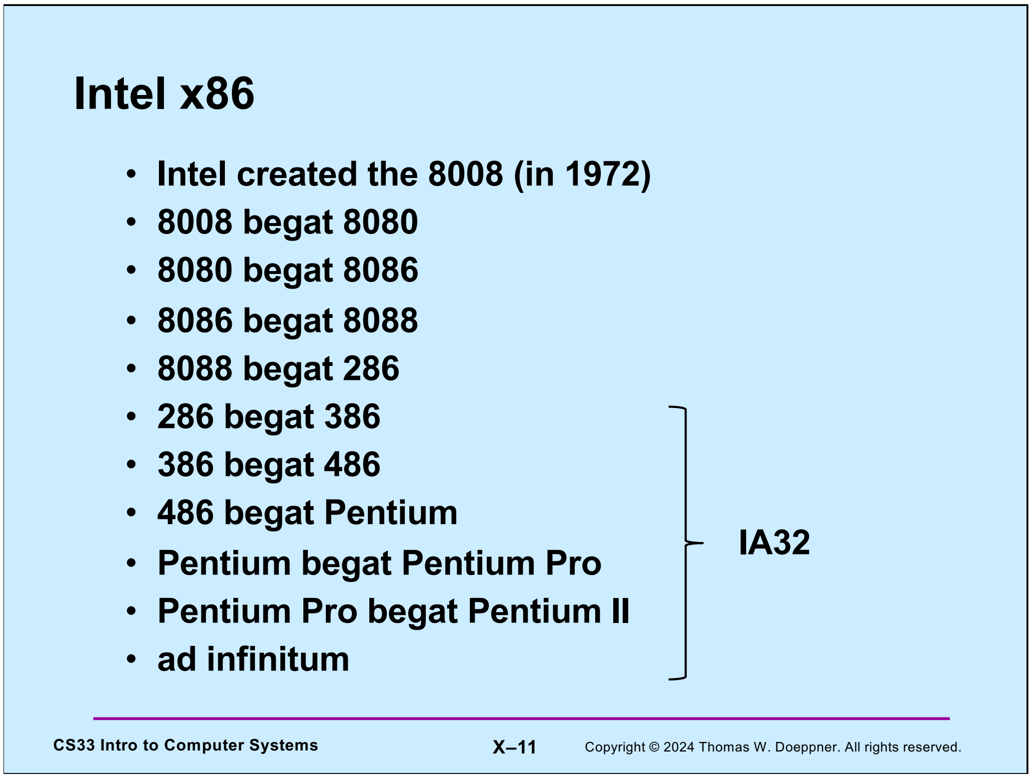
data

Memory (aka RAM)

a relatively long distance

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.

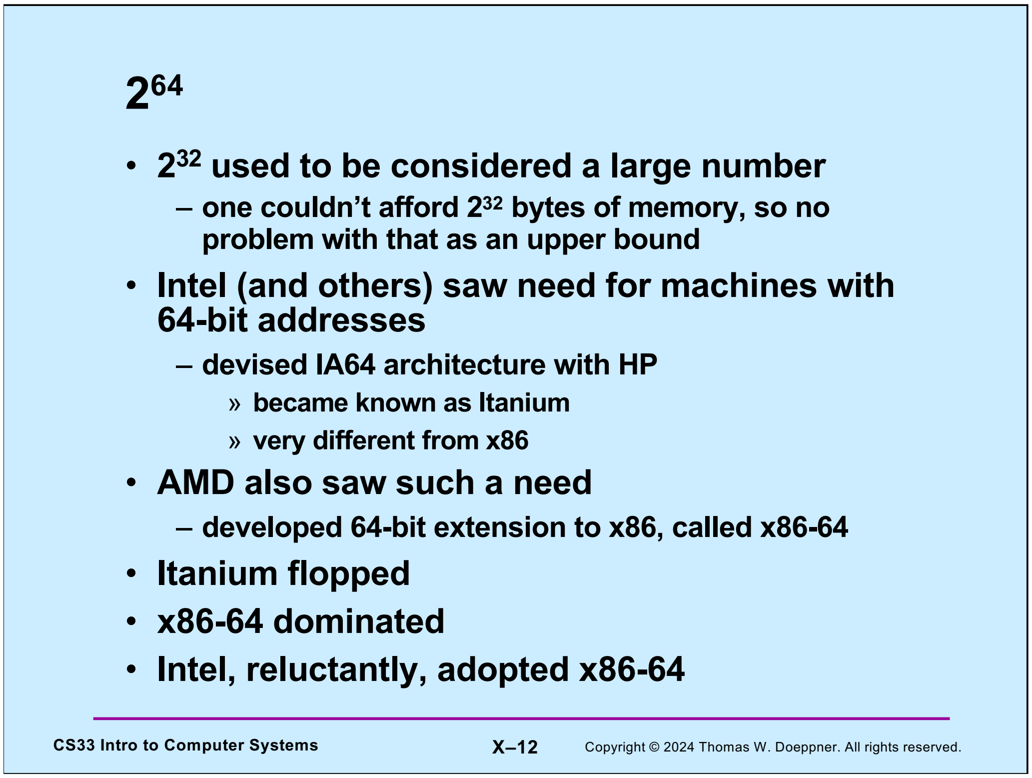


Intel x86

- Intel created the 8008 (in 1972)
- 8008 begat 8080
- 8080 begat 8086
- 8086 begat 8088
- 8088 begat 286
- 286 begat 386
- 386 begat 486
- 486 begat Pentium
- Pentium begat Pentium Pro
- Pentium Pro begat Pentium II
- ad infinitum

$\left.\begin{array}{l}\text{286 begat 386}\\ \vdots \\ \text{ad infinitum}\end{array}\right\}$ IA32

The early computers of the x86 family had 16-bit words; starting with the 386, they supported 32-bit words.



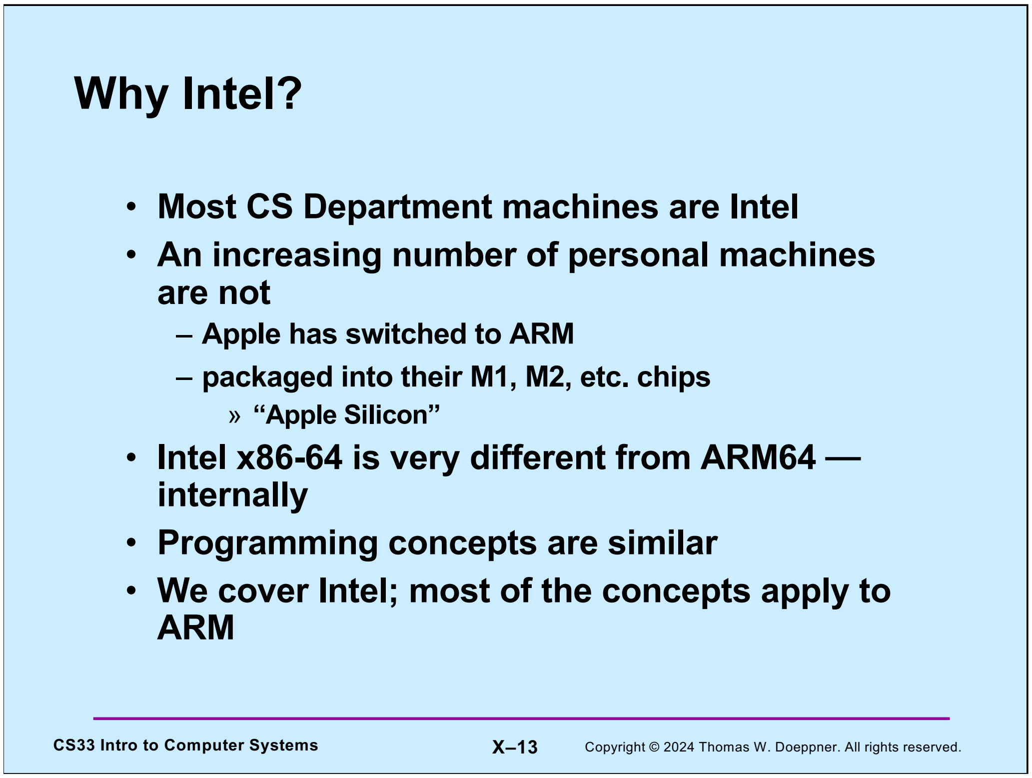
2^{64}

- 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.



Why Intel?

- Most CS Department machines are Intel
- An increasing number of personal machines are not
  - – Apple has switched to ARM
  - – packaged into their M1, M2, etc. chips
    - » “Apple Silicon”
- Intel x86-64 is very different from ARM64 — internally
- Programming concepts are similar
- We cover Intel; most of the concepts apply to ARM

ARM originally stood for Acorn RISC machine. Acorn was a British computer company that was established in 1978, but no longer exists. RISC stands for Reduced Instruction Set Computer. The RISC concept was devised in the 1980s and was very popular in the 80s and 90s. The idea is to design computers with relatively few instructions, but implement those instructions so they can execute very quickly. The fastest computers in the 80s and 90s were RISC computers. But Intel, who built computer chips with fairly complex instruction sets (CISC), learned how to make their computers run really fast as well. That, coupled with the fact that Windows ran exclusively on Intel, helped Intel stay in the lead.

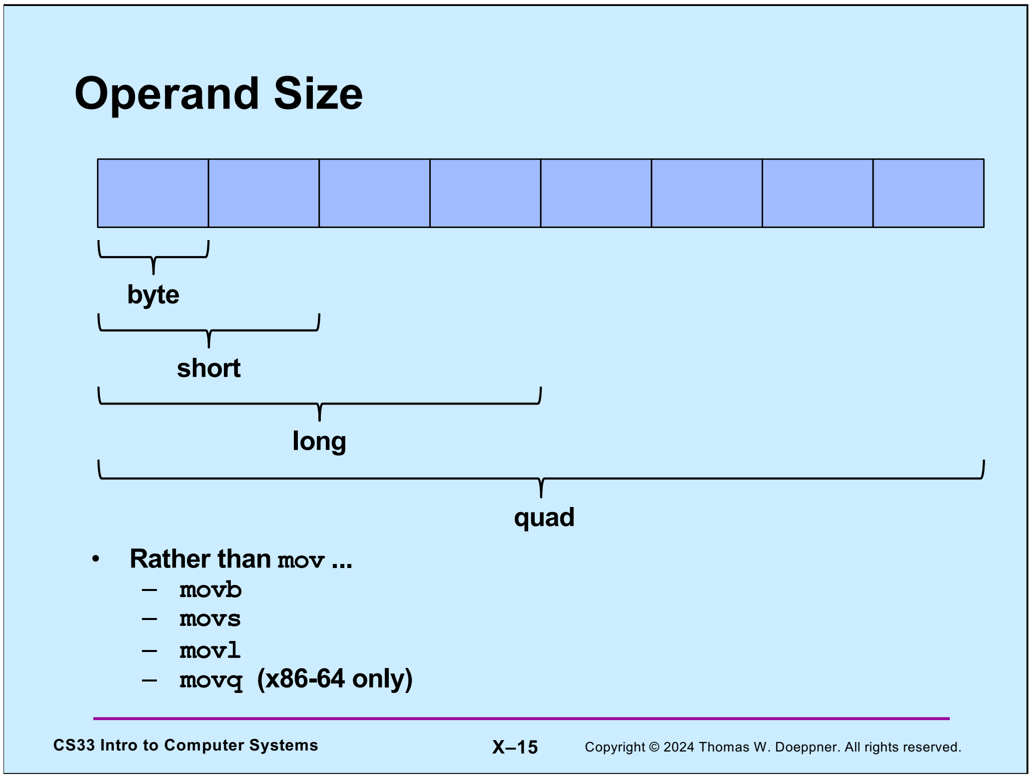
ARM later became Advanced RISC Machine. Now, it doesn’t stand for anything, It’s just ARM.

Apple (whose computers originally ran Motorola 68000 processors before they switched to Intel) decided that they could make more cost-effective and faster processors by adapting the ARM design and including GPUs (graphics processing units). GPUs are specialized processors that help with image processing, but also can be used with other computations that have a lot of inherent parallelism. Apple refers to their new chips as M1 and M2 (presumably an M3 is not far behind).

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.



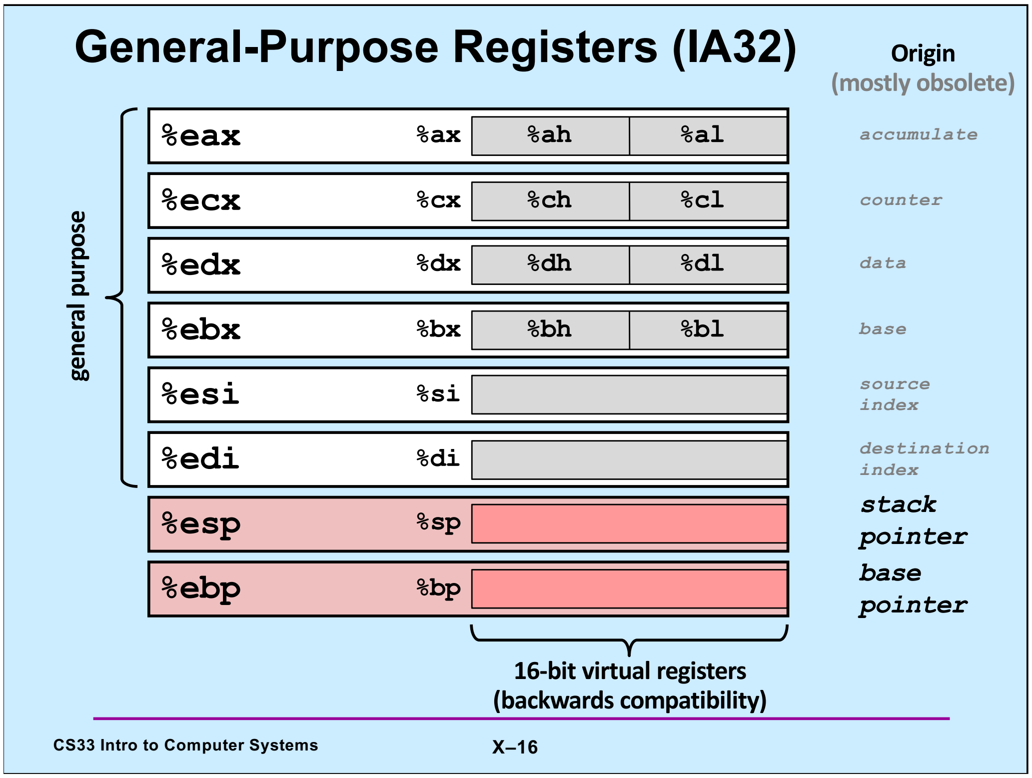
Operand Size

- **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.

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.



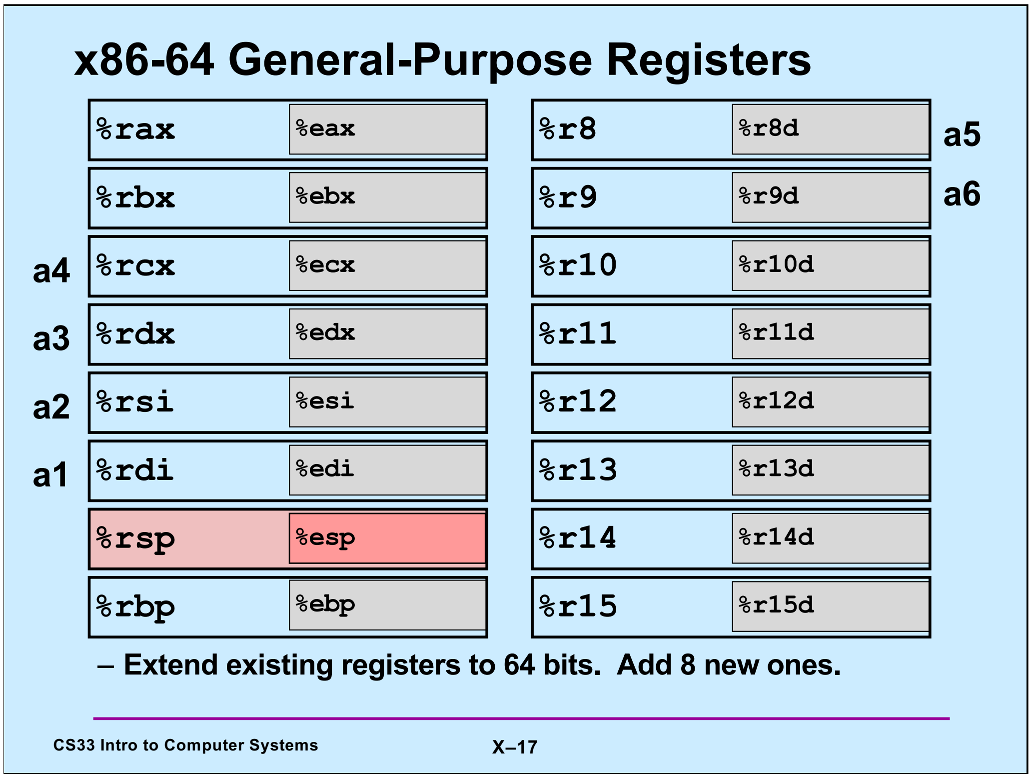
General-Purpose Registers (IA32)

general purpose

				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.



x86-64 General-Purpose Registers

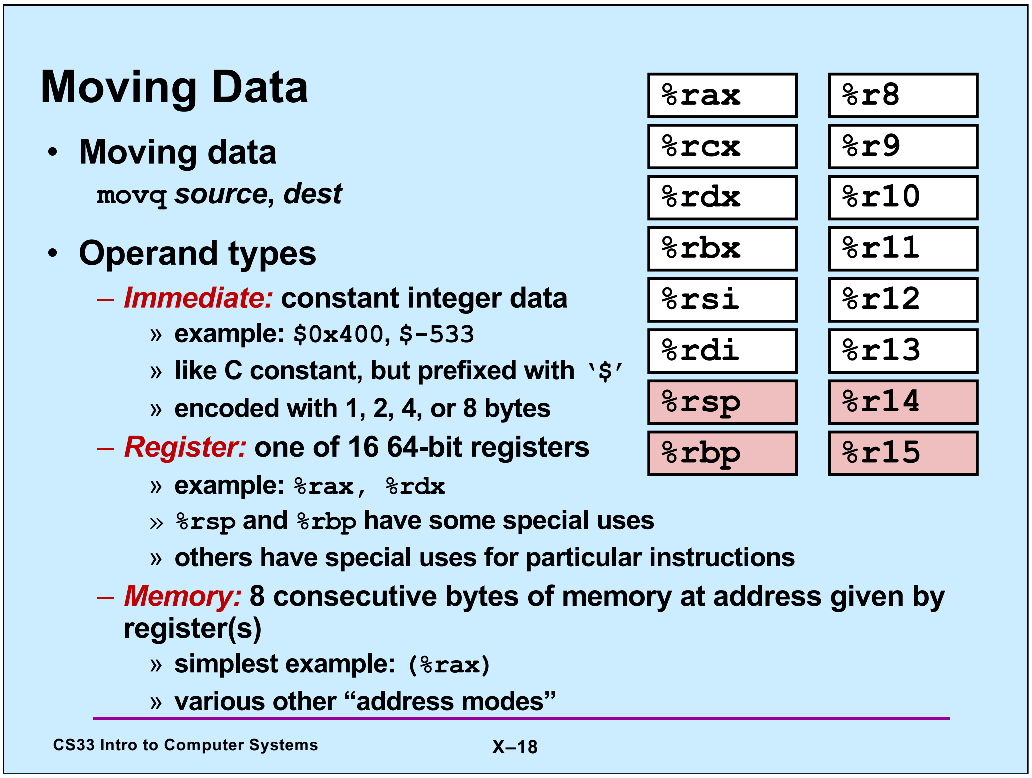
	`%rax`	`%eax`	`%r8`	`%r8d`	a5
	`%rbx`	`%ebx`	`%r9`	`%r9d`	a6
a4	`%rcx`	`%ecx`	`%r10`	`%r10d`	
a3	`%rdx`	`%edx`	`%r11`	`%r11d`	
a2	`%rsi`	`%esi`	`%r12`	`%r12d`	
a1	`%rdi`	`%edi`	`%r13`	`%r13d`	
	`%rsp`	`%esp`	`%r14`	`%r14d`	
	`%rbp`	`%ebp`	`%r15`	`%r15d`	

– **Extend existing registers to 64 bits. Add 8 new ones.**

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.



Moving Data

- **Moving data**

  `movq` ***source*, *dest***

- **Operand types**
  - ***Immediate:*** **constant integer data**
    - » **example:** `$0x400`, `$-533`
    - » **like C constant, but prefixed with** `'$'`
    - » **encoded with 1, 2, 4, or 8 bytes**
  - ***Register:*** **one of 16 64-bit registers**
    - » **example:** `%rax, %rdx`
    - » `%rsp` **and** `%rbp` **have some special uses**
    - » **others have special uses for particular instructions**
  - ***Memory:*** **8 consecutive bytes of memory at address given by register(s)**
    - » **simplest example:** `(%rax)`
    - » **various other "address modes"**

`%rax`	`%r8`
`%rcx`	`%r9`
`%rdx`	`%r10`
`%rbx`	`%r11`
`%rsi`	`%r12`
`%rdi`	`%r13`
`%rsp`	`%r14`
`%rbp`	`%r15`

Based on a slide supplied by CMU.

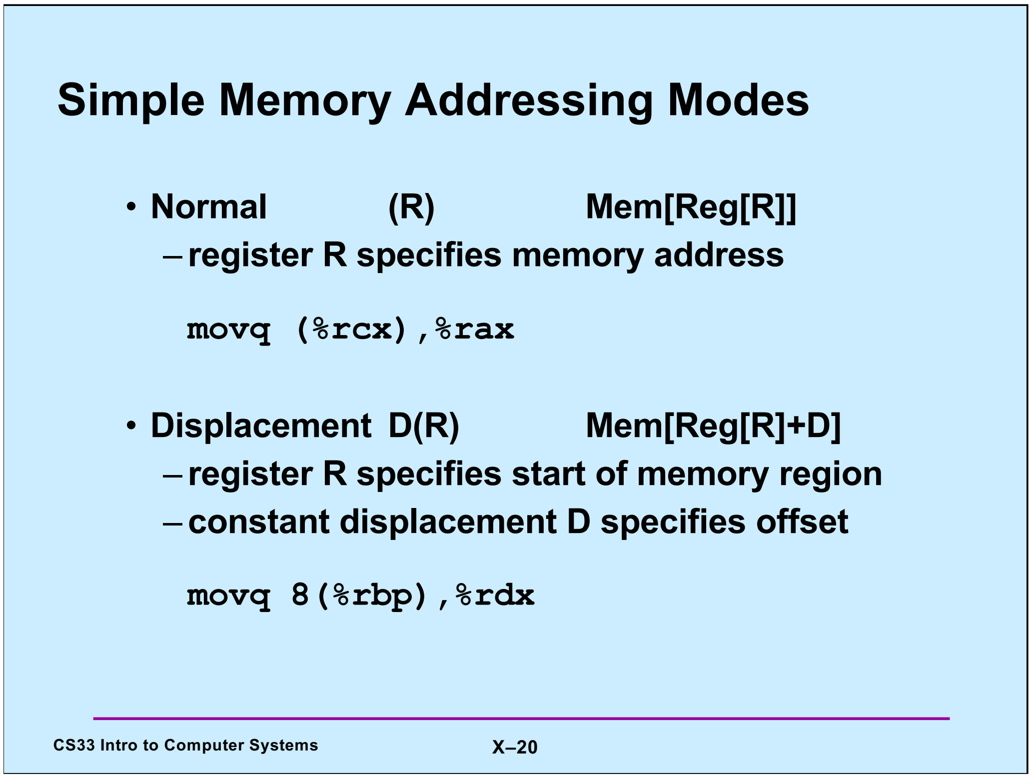
Some assemblers (in particular, those of Intel and Microsoft) place the operands in the opposite order. Thus, the example of the slide would be "addl %rax,8(%rbp)". The order we use is that used by gcc, known as the "AT&T syntax" because it was used in the original Unix assemblers, written at Bell Labs, then part of AT&T.

`movq` Operand Combinations

	Source	Dest	Src, Dest	C Analog
`movq`	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	*Reg*	`movq (%rax),%rdx`	`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**

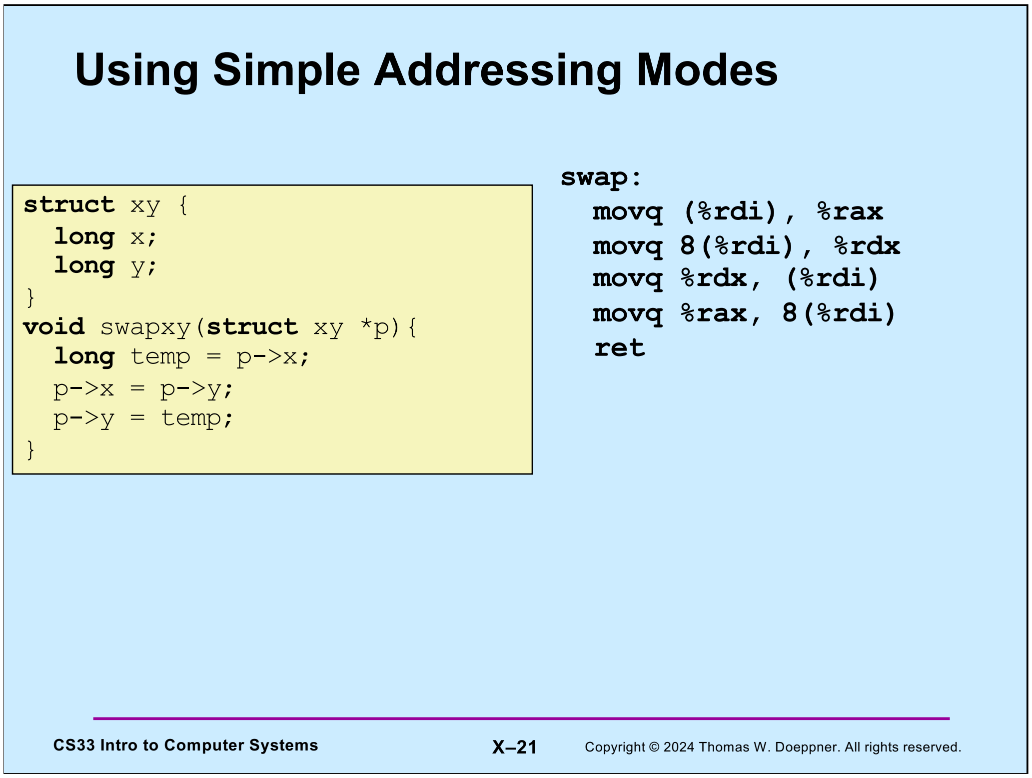
```
movq (%rcx),%rax
```

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

```
movq 8(%rbp),%rdx
```

Supplied by CMU.

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

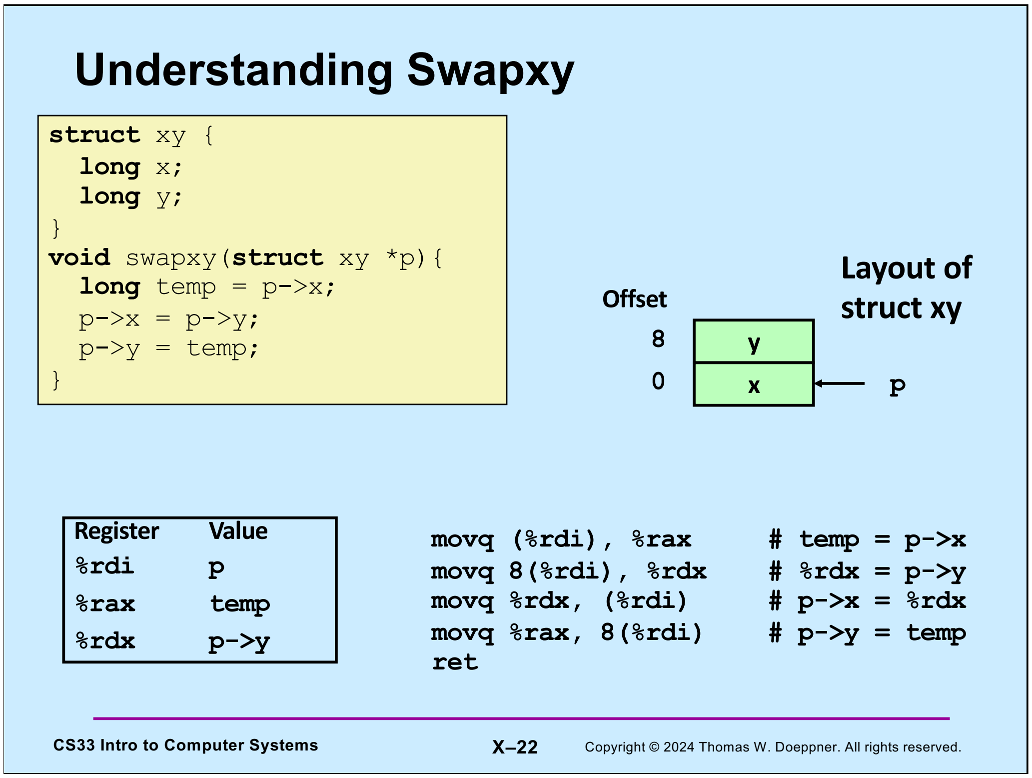


Using Simple Addressing Modes

```
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

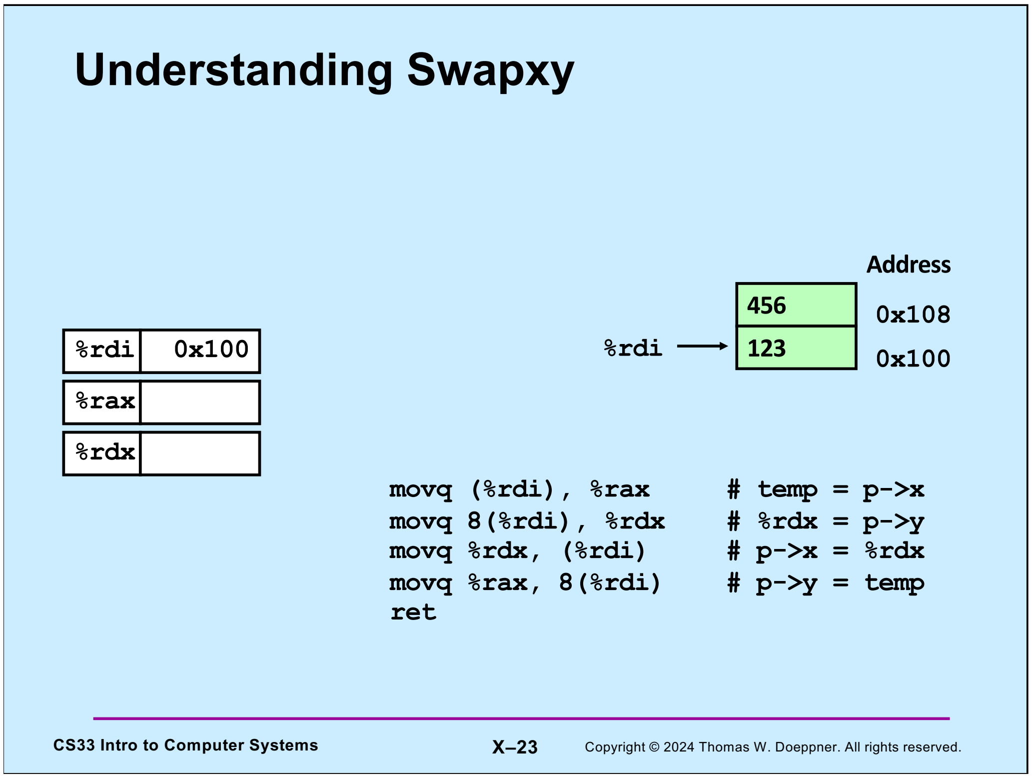
```
struct xy {
  long x;
  long y;
}
void swapxy(struct xy *p){
  long temp = p->x;
  p->x = p->y;
  p->y = temp;
}
```

Layout of struct xy

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.



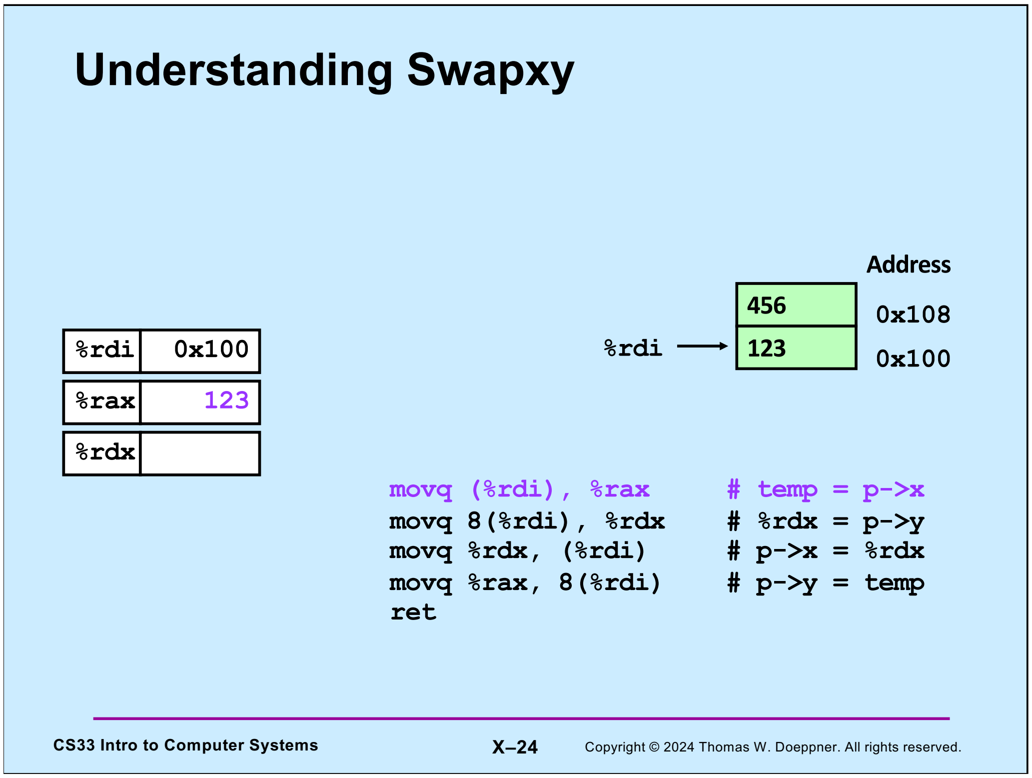
Understanding Swapxy

%rdi	0x100
%rax	
%rdx	

Value	Address
456	0x108
123	0x100

```
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
```

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



Understanding Swapxy

%rdi	0x100
%rax	123
%rdx	

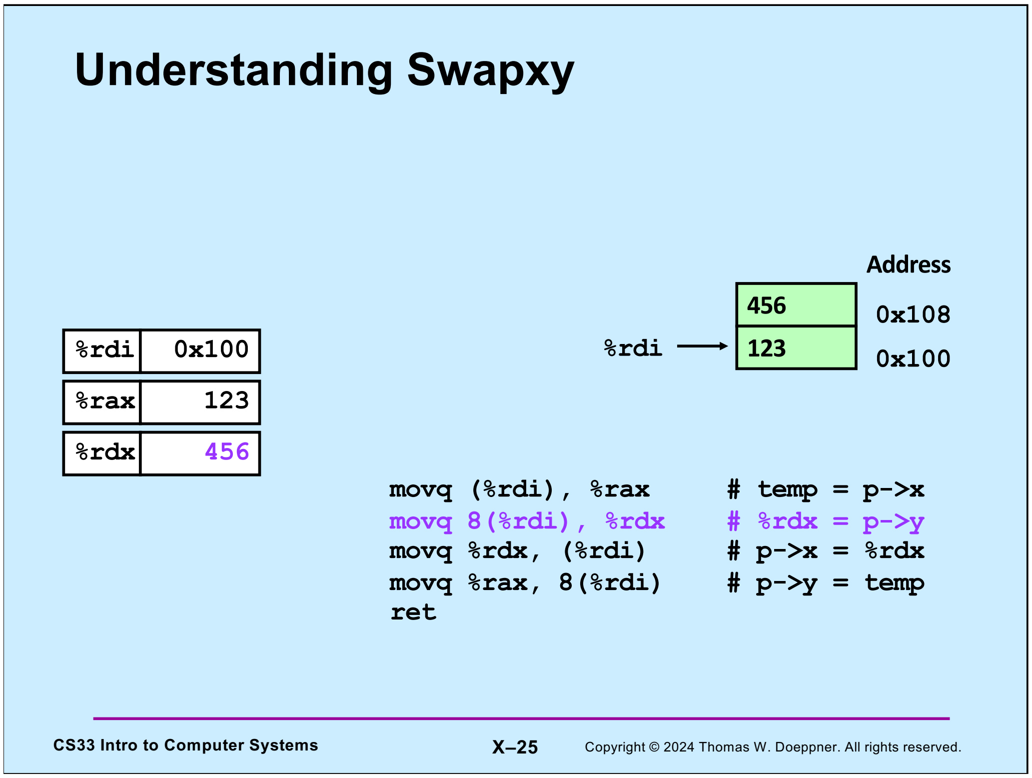
Address

%rdi → 123 0x100

456 0x108

```
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
```

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



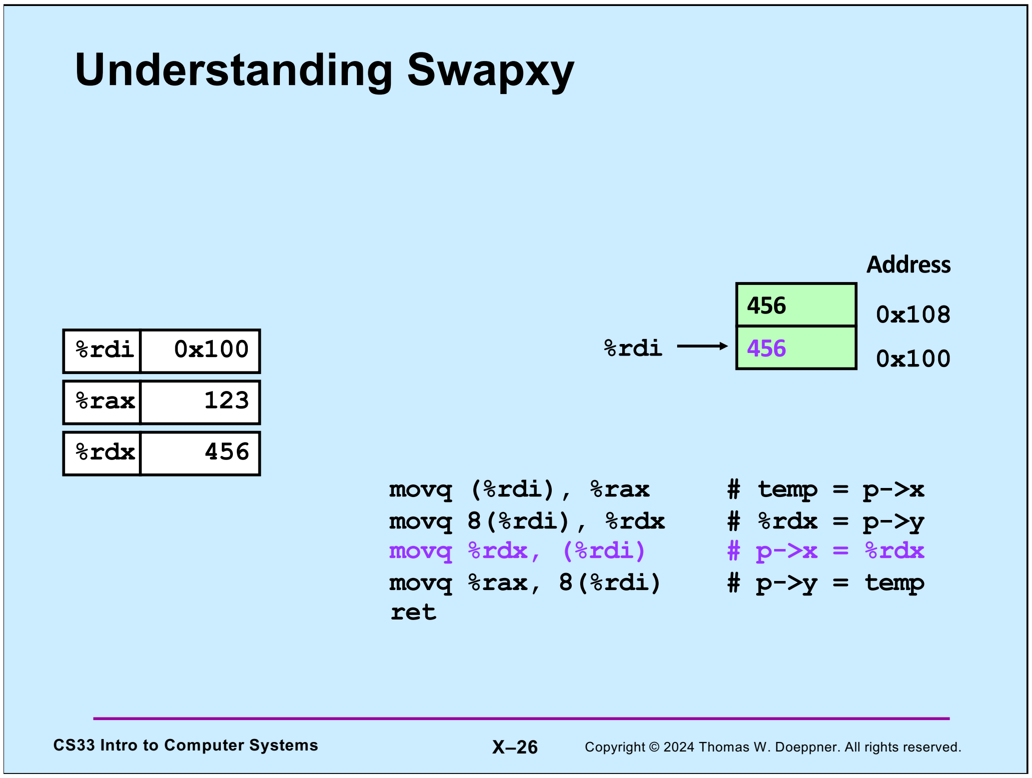
Understanding Swapxy

%rdi	0x100
%rax	123
%rdx	456

	Address
456	0x108
123	0x100

```
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
```

We then copy the second component into %rdx.



Understanding Swapxy

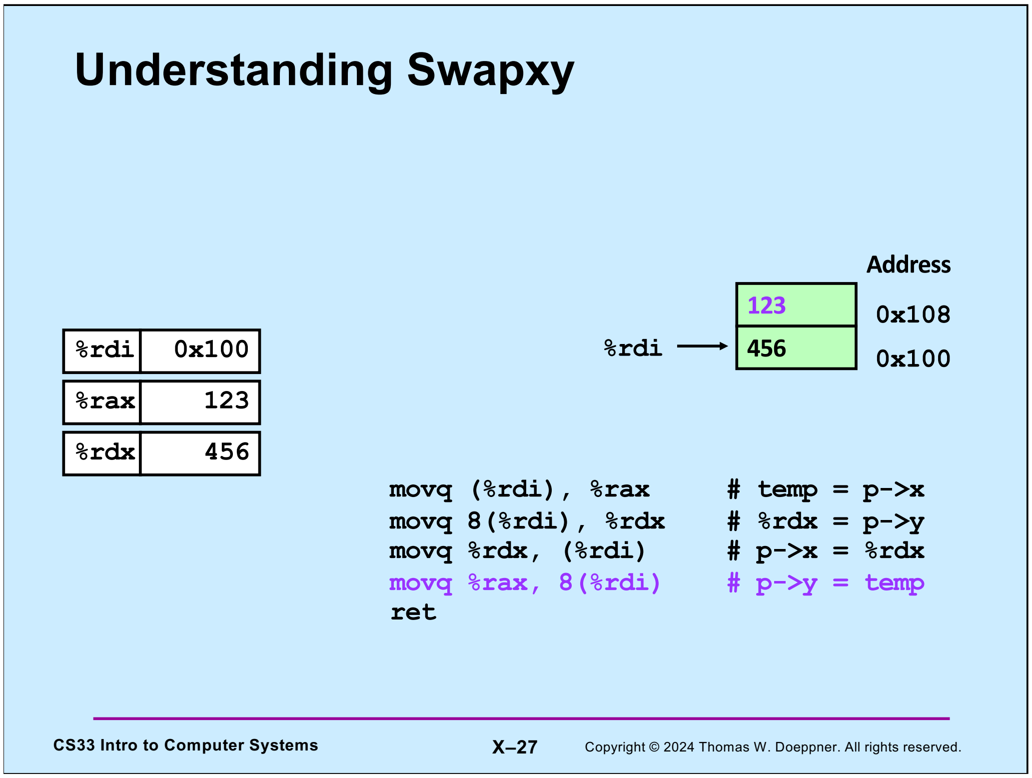
%rdi	0x100
%rax	123
%rdx	456

%rdi →

	Address
456	0x108
456	0x100

```
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
```

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



Understanding Swapxy

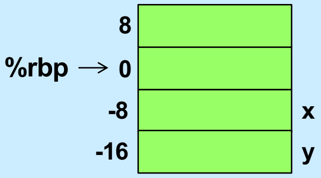
%rdi	0x100
%rax	123
%rdx	456

```
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
```

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

Quiz 2

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



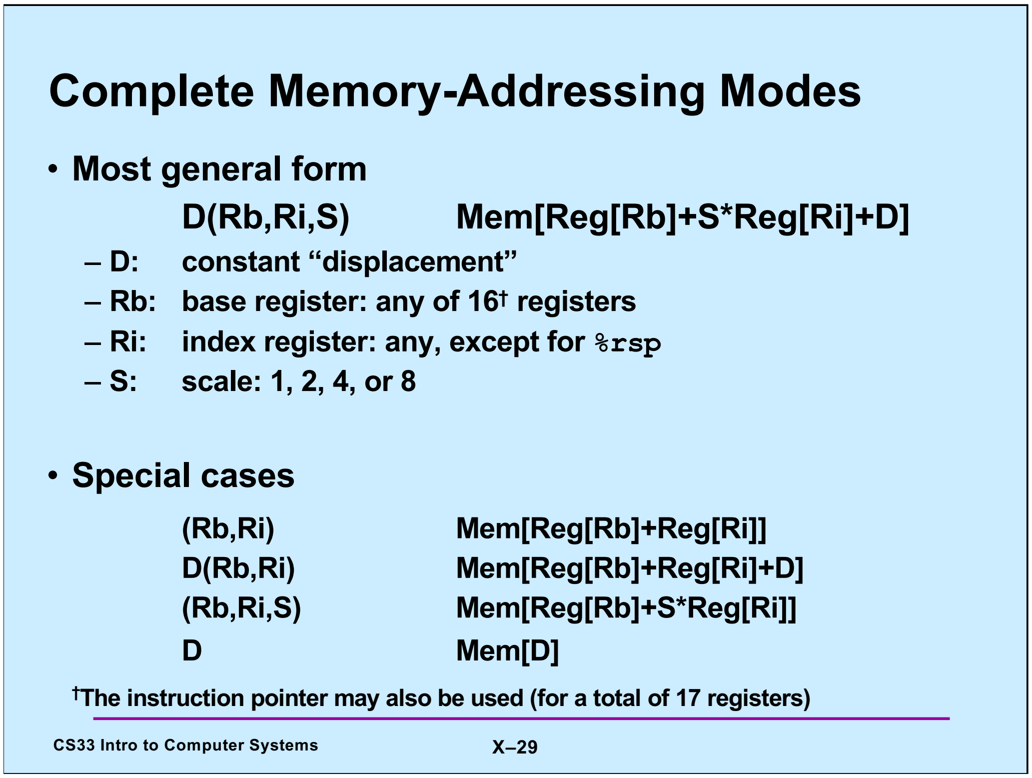
Which C statements best describe the assembler code?

```
// 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

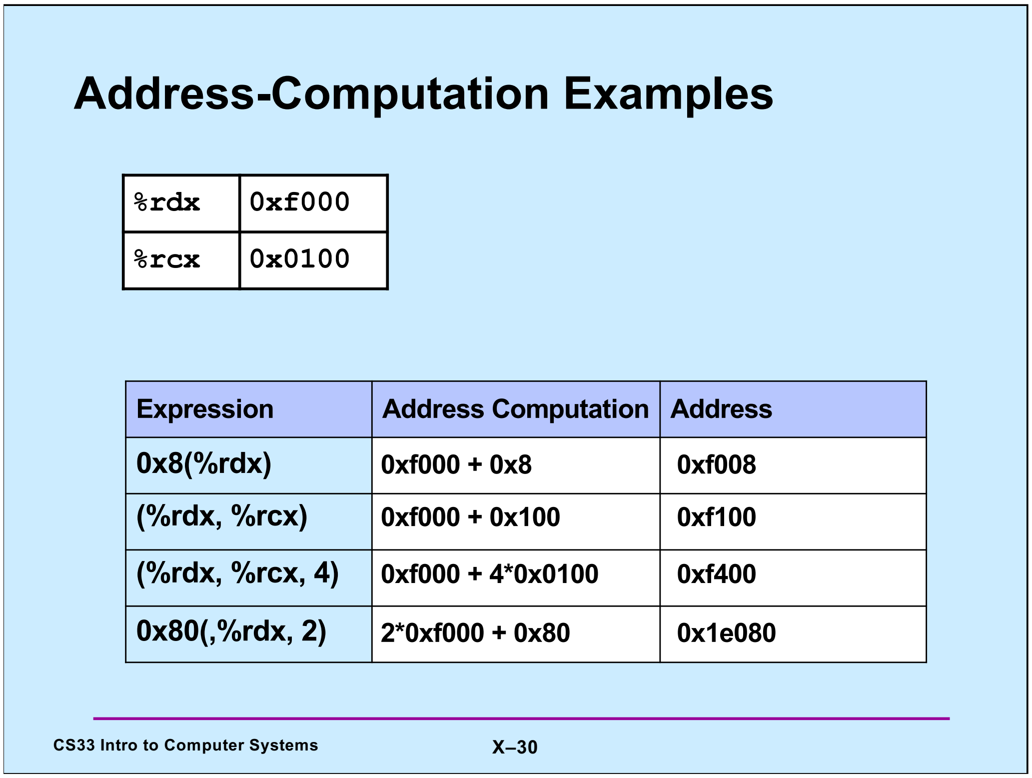
- Special cases

(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.

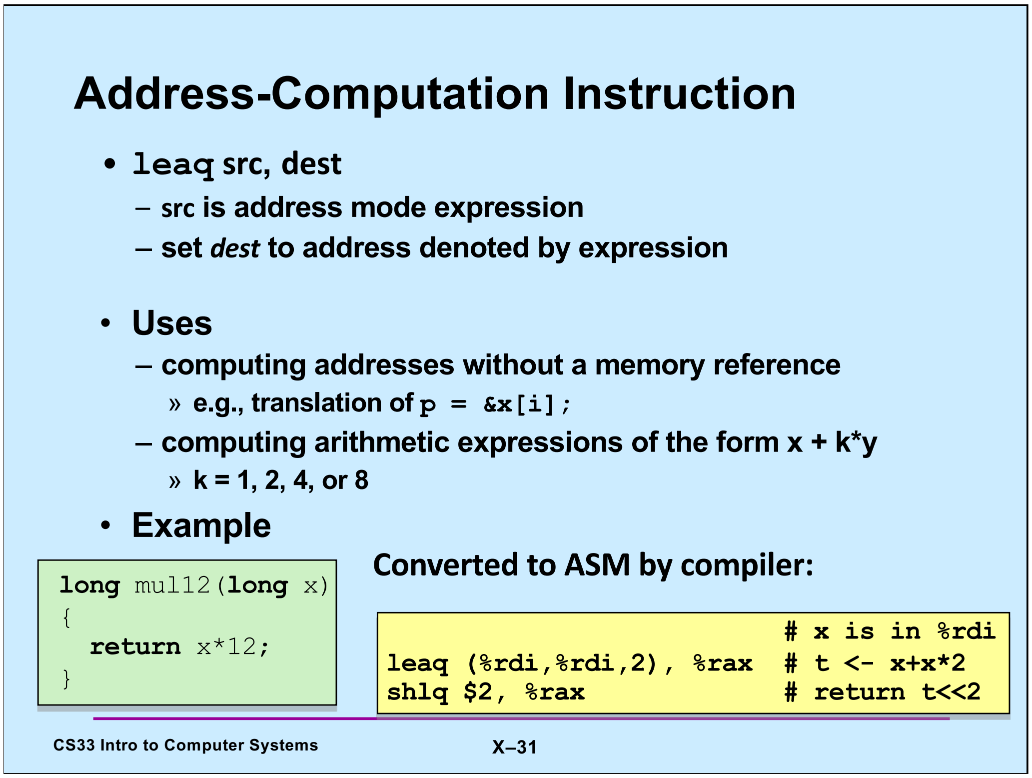


Address-Computation Examples

`%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



Address-Computation Instruction

- `leaq` 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

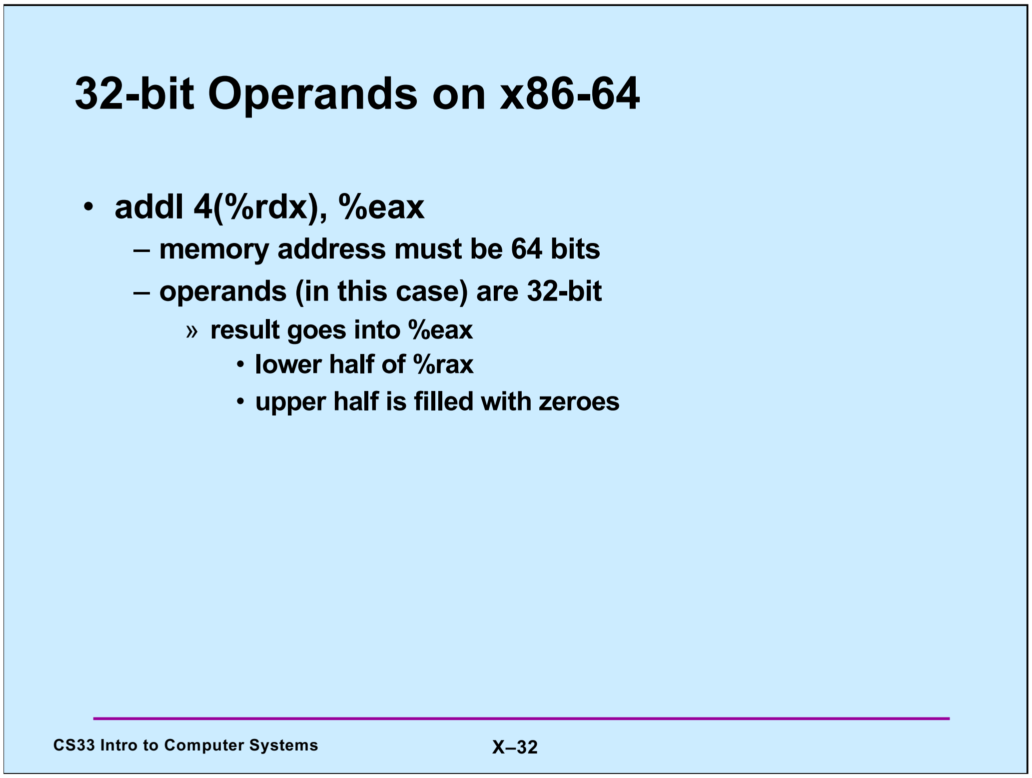
```
long mul12(long x)
{
  return x*12;
}
```

Converted to ASM by compiler:

```
                              # x is in %rdi
leaq (%rdi,%rdi,2), %rax      # t <- x+x*2
shlq $2, %rax                 # return t<<2
```

Adapted from a slide supplied by CMU.

Note that a function returns a value by putting it in %rax.



32-bit Operands 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.

Quiz 3

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: