CS 33

Machine Programming (2)

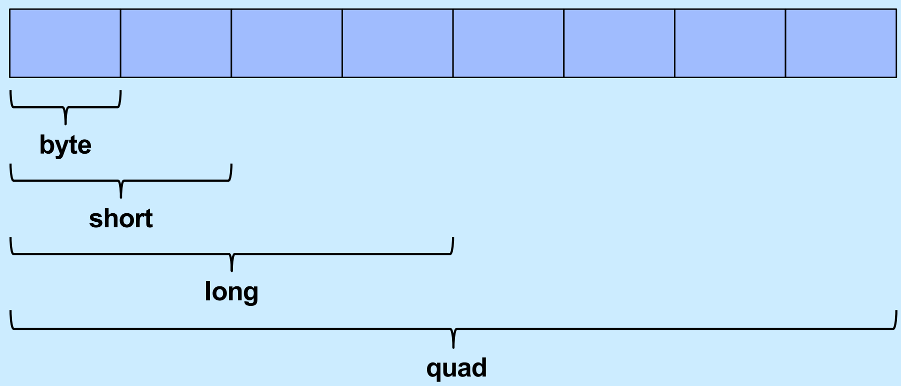
Many of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook "Computer Systems: A Programmer's Perspective," 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O'Hallaron in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

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

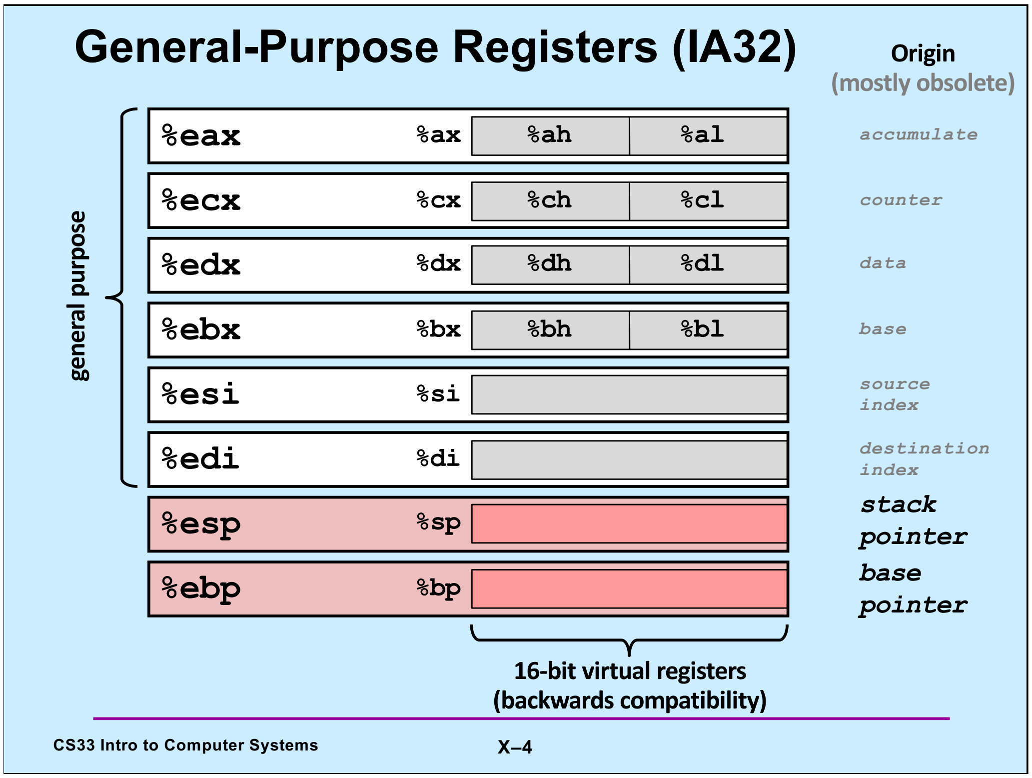


- Rather than `mov` ...
  - `movb`
  - `movs`
  - `movl`
  - `movq` (x86-64 only)

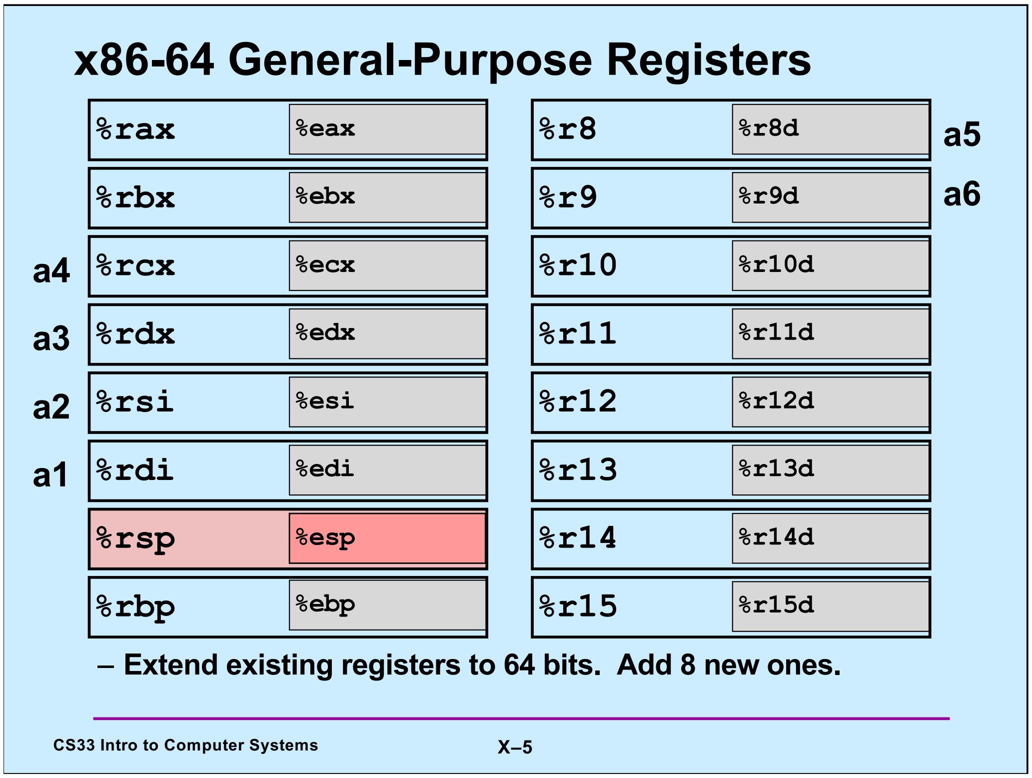
Most instructions come in three (on IA32) or four (on x86-64) forms, one for each possible operand size.

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.



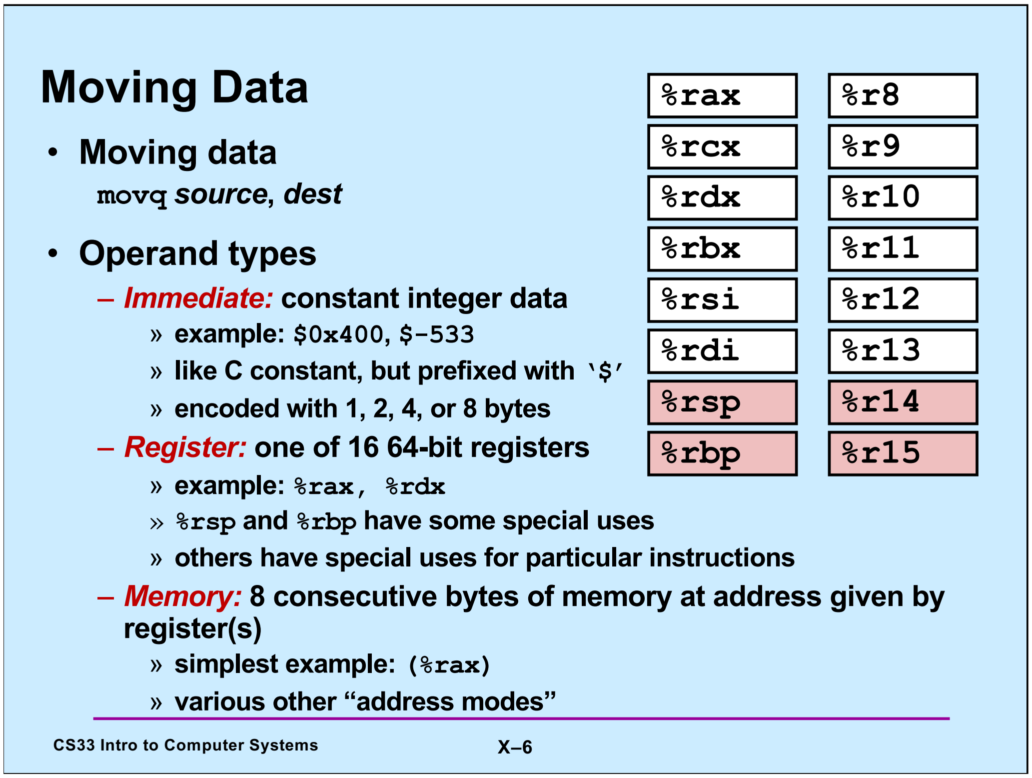
Supplied by CMU.



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.

`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.

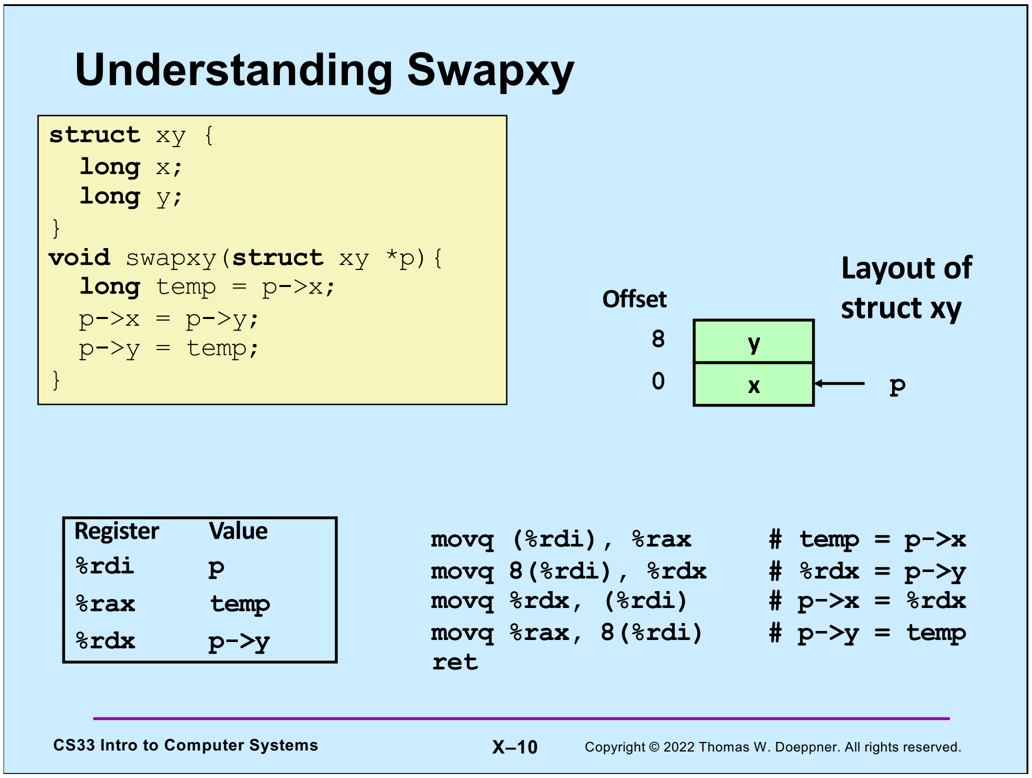
If one thinks of there being an array of registers, then “Reg[R]” selects register “R” from this array.

Using Simple Addressing Modes

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

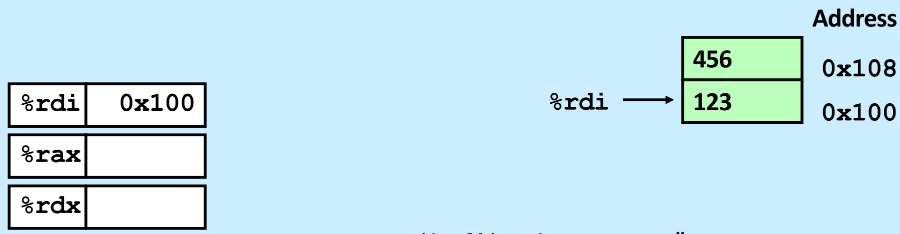
```
swap:
  movq (%rdi), %rax
  movq 8(%rdi), %rdx
  movq %rdx, (%rdi)
  movq %rax, 8(%rdi)
  ret
```


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.)



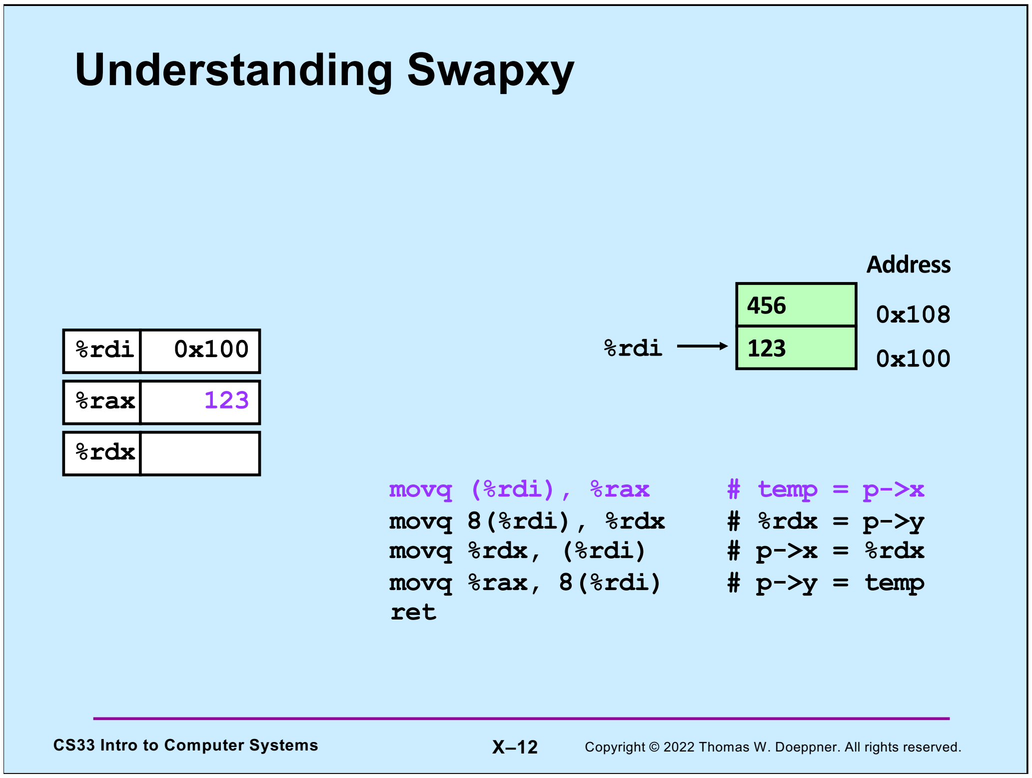
In addition to using %rdi to contain the argument (the address of the structure), we use %rax to contain the value of **temp** and %rdx to effectively be another temporary that holds the value of p->y.

Understanding Swapxy




```
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
```

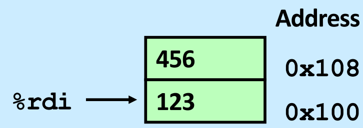

When we enter **swapxy**, %rdi contains the address of the structure.



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

Understanding Swapxy

%rdi	0x100
%rax	123
%rdx	456

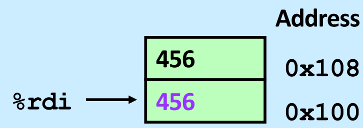



```
movq (%rdi), %rax      # temp = p->x
movq 8(%rdi), %rdx     # %rdx = p->y
movq %rdx, (%rdi)      # p->x = %rdx
movq %rax, 8(%rdi)     # p->y = temp
ret
```


We then copy the second component into %rdx.

Understanding Swapxy

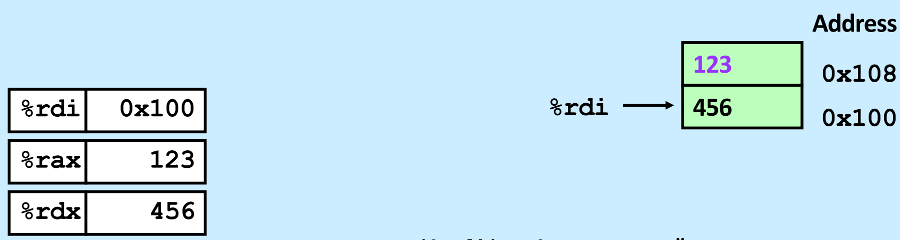
%rdi	0x100
%rax	123
%rdx	456




```
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
```


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

Understanding Swapxy

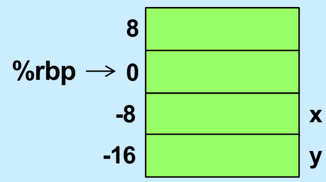



```
movq (%rdi), %rax      # temp = p->x
movq 8(%rdi), %rdx     # %rdx = p->y
movq %rdx, (%rdi)      # p->x = %rdx
movq %rax, 8(%rdi)     # p->y = temp
ret
```


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

Quiz 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

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.

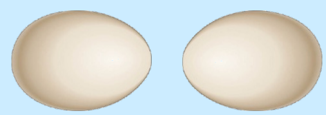
Quiz 2

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
```

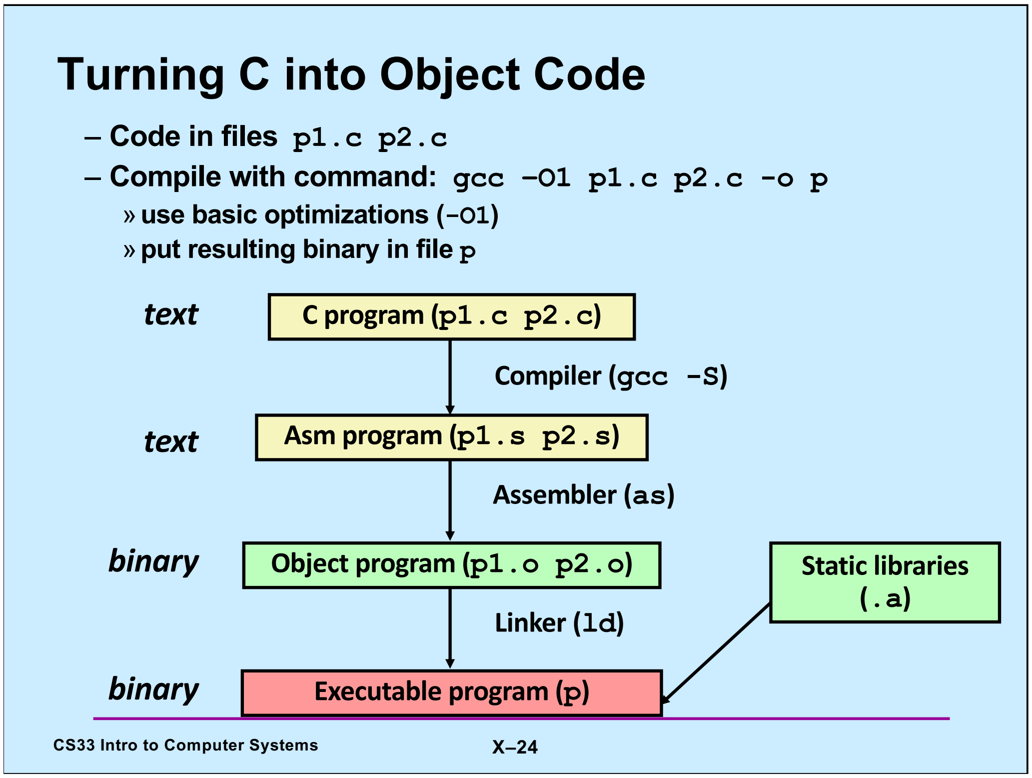
- **Pointers are 64 bits**
- **What they point to are 32 bits**

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.

Bytes

- Each register has a byte version
  - e.g., %r10: %r10b; see earlier slide for x86 registers
- Needed for byte instructions
  - movb (%rax, %rsi), %r10b
  - sets *only* the low byte in %r10
    - » other seven bytes are unchanged
- Alternatives
  - movzbq (%rax, %rsi), %r10
    - » copies byte to low byte of %r10
    - » zeroes go to higher bytes
  - movsbq (%rax, %rsi), %r10
    - » copies byte to low byte of %r10
    - » sign is extended to all higher bits

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;
}
```

Assembler Code

```
ASum:
    testq     %rsi, %rsi
    je        .L4
    movq      %rdi, %rdx
    leaq      (%rdi,%rsi,8), %rcx
    movl      $0, %edx
.L3:
    addq      (%rax), %rdx
    addq      $8, %rax
    cmpq      %rcx, %rdx
    jne       .L3
.L1:
    movq      %rdx, %rax
    ret
.L4:
    movl      $0, %eax
    jmp       .L1
```


Here is the assembler code produced by gcc from the C code of the previous slide.

Object Code

Code for `ASum`

```
0x112b <ASum>:
    0x48
    0x85
    0xf6
    0x74
    0x19
    0x48
    0x89
    0xfa
    0x48
    0x8d
    0x0c
    0xf7
     .
     .
     .
```

- Total of 35 bytes
- Each instruction: 1, 2, or 3 bytes
- Starts at address `0x112b`

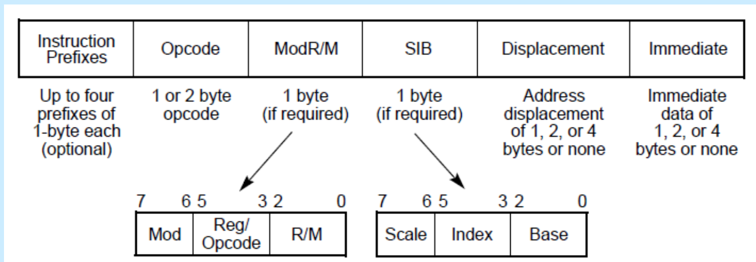
- **Assembler**
  - translates `.s` into `.o`
  - binary encoding of each instruction
  - nearly complete image of executable code
  - missing linkages between code in different files
- **Linker**
  - resolves references between files
  - combines with static run-time libraries
    - » e.g., code for `printf`
  - some libraries are *dynamically linked*
    - » linking occurs when program begins execution

Adapted from a slide supplied by CMU.

The lefthand column shows the object code produced by gcc. This was produced either by assembling the code of the previous slide, or by compiling the C code of the slide before that.

Suppose that all we have is the object code – we don’t have the assembler code and the C code. Can we translate for object code to assembler code? (This is known as disassembling.)

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)

The point of the slide is that the instruction format is complicated, too much so for a human to deal with. Which is why we talk about **disassemblers** in the next slides.

Disassembling Object Code

Disassembled

```
000000000000112b <ASum>:
    112b:  48 85 f6              test   %rsi,%rsi
    112e:  74 19                 je     1149 <ASum+0x1e>
    1130:  48 89 fa              mov    %rdi,%rdx
    1133:  48 8d 0c f7           lea    (%rdi,%rsi,8),%rcx
    1137:  b8 00 00 00 00        mov    $0x0,%eax
    113c:  48 03 02              add    (%rdx),%rax
    113f:  48 83 c2 08           add    $0x8,%rdx
    1143:  48 39 ca              cmp    %rcx,%rdx
    1146:  75 f4                 jne    113c <ASum+0x11>
    1148:  c3                    retq
    1149:  b8 00 00 00 00        mov    $0x0,%eax
    114e:  c3                    retq
```

- **Disassembler**

  ```
  objdump -d <file>
  ```

  – **useful tool for examining object code**

  – **produces approximate rendition of assembly code**

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. Also, it leaves off the suffix at the end of each instruction, assuming it can be determined from context.

Alternate Disassembly

Object

```
0x112b:
    0x48
    0x85
    0xf6
    0x74
    0x19
    0x48
    0x89
    0xfa
    0x48
    0x8d
    0x0c
    0xf7
     .
     .
     .
```

Disassembled

```
Dump of assembler code for function ASum:
   0x112b <+0>:     test   %rsi,%rsi
   0x112e <+3>:     je     0x1149 <ASum+30>
   0x1130 <+5>:     mov    %rdi,%rdx
   0x1133 <+8>:     lea    (%rdi,%rsi,8),%rcx
   0x1137 <+12>:    mov    $0x0,%eax
      . . .
```

- **Within gdb debugger**

  `gdb <file>`

  `disassemble ASum`

  – disassemble the ASum object code

  `x/35xb ASum`

  – examine the 35 bytes starting at `ASum`

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.

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
    - » ~100
  - 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 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!

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	
`shll`	Src,Dest	Dest = Dest << Src	Also called sall
`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!**

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.

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)
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.

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.

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.

Note, again, that the value that a function returns is put into %rax (or its 32-bit version, %eax).

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.