CS 33

Machine Programming (2)

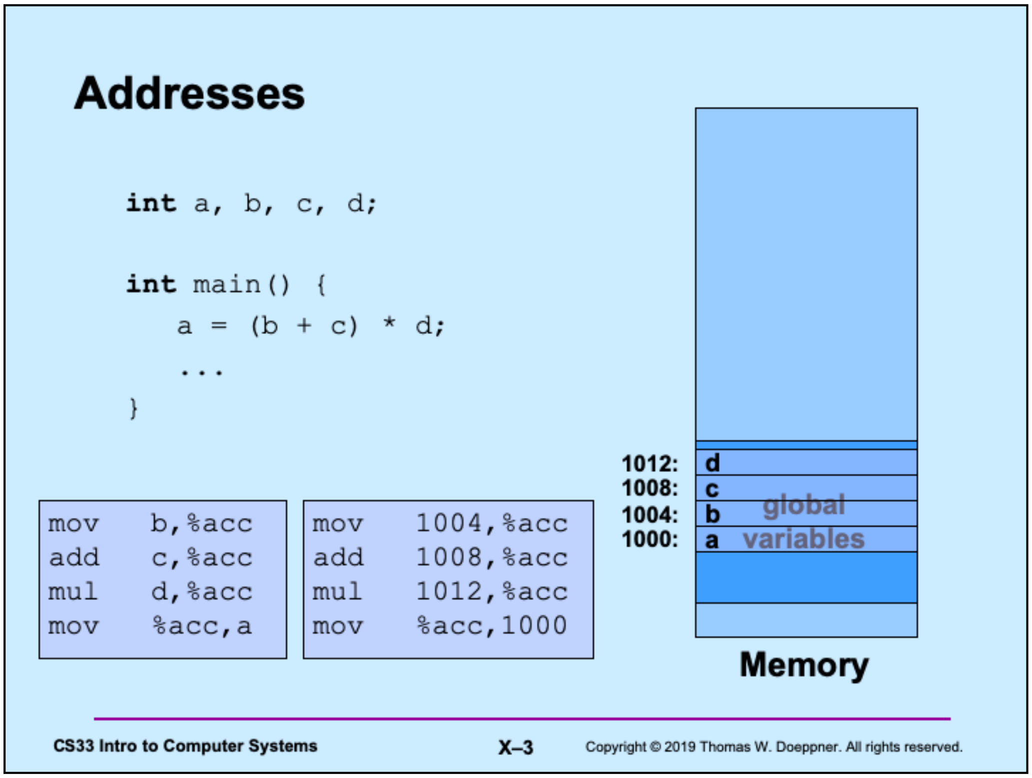
Many of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook "Computer Systems: A Programmer's Perspective," 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O'Hallaron in 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
    - » jgt
      - jump on greater than
    - » 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 the next lecture, when we start looking at x86 assembler instructions.



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!

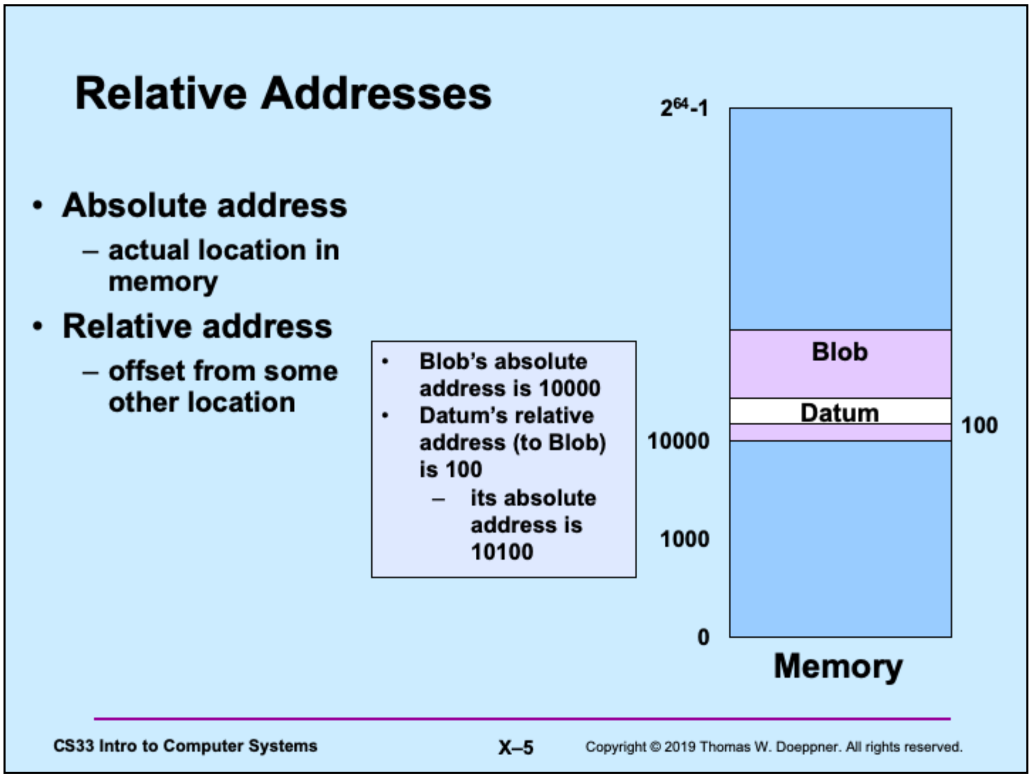
Beginning with this slide, whenever we draw pictures of memory, lower memory addresses are at the bottom, higher addresses are at the top. This is the opposite of how we've been drawing pictures of memory in previous slides.

Addresses

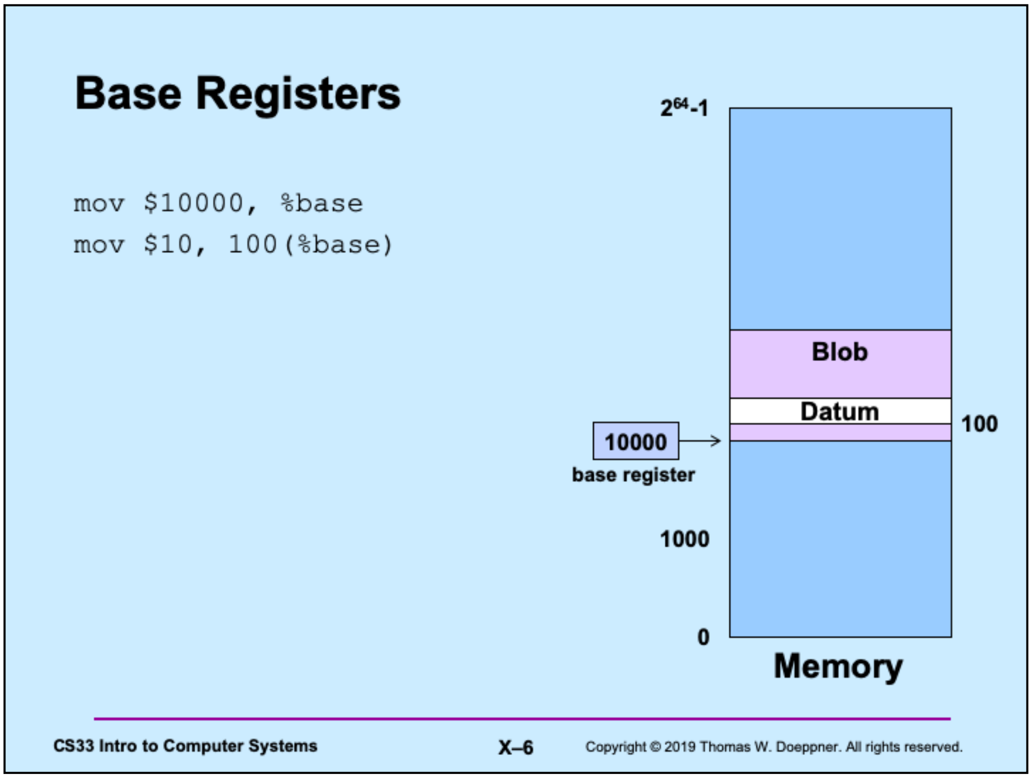
```
int b;

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 for each call to *func*
- Different copies of *a*, *c*, and *d* for 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.



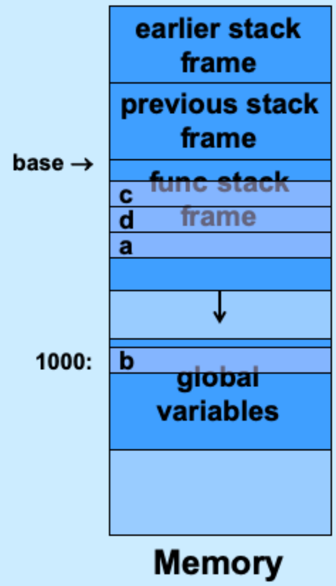
Note that both positive and negative offsets might be used.



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   -12(%base),%acc
   mov   %acc,-16(%base)
```



CS33 Intro to Computer Systems X–7

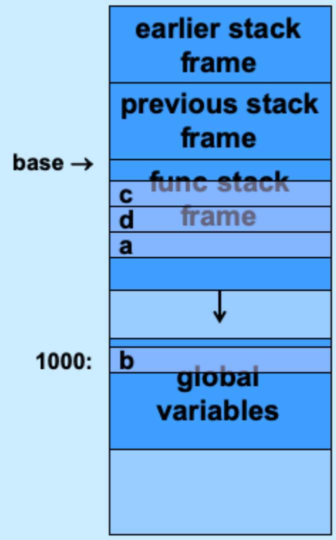
Here we return to our earlier example. We assume that, as part of the call to *func*, the base register is loaded with the address of the beginning of *func*'s current stack frame, and that the local variable *a* and the parameters *c* and *d* are located within the frame. Thus we refer to them by their offset from the beginning of the stack frame, which are assumed to be *-16*, *-8,* and *-12*. 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

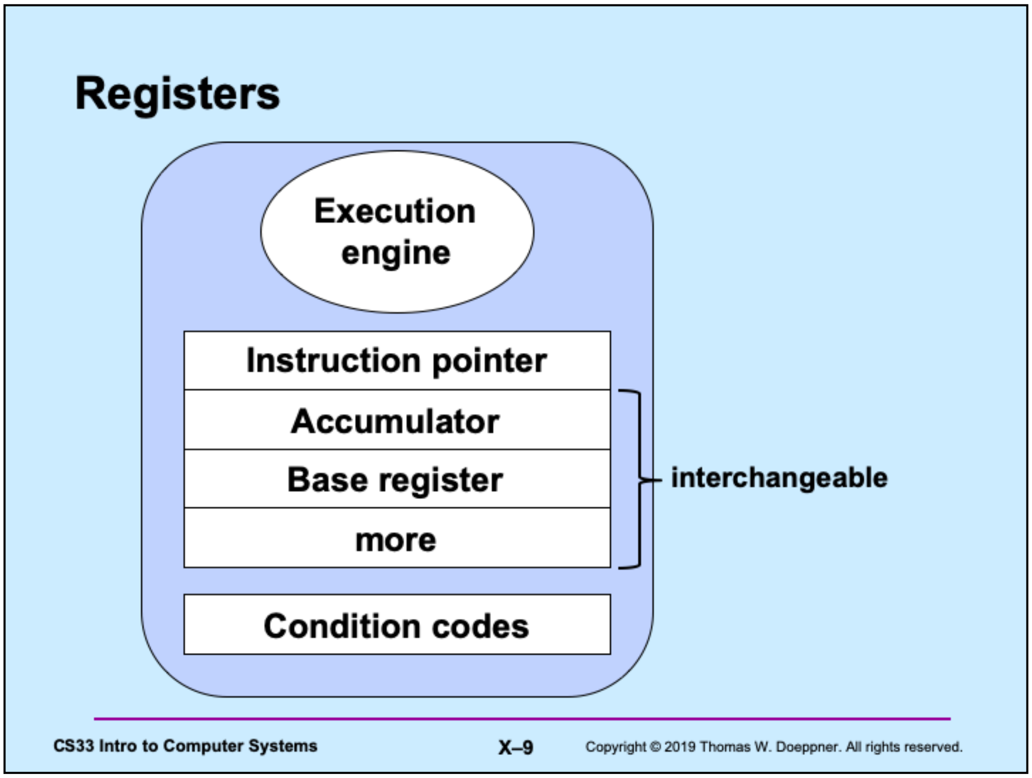
Suppose the value in *base* is 10,000. What is the address of *c*?

a) 9992
b) 9996
c) 10,004
d) 10,008

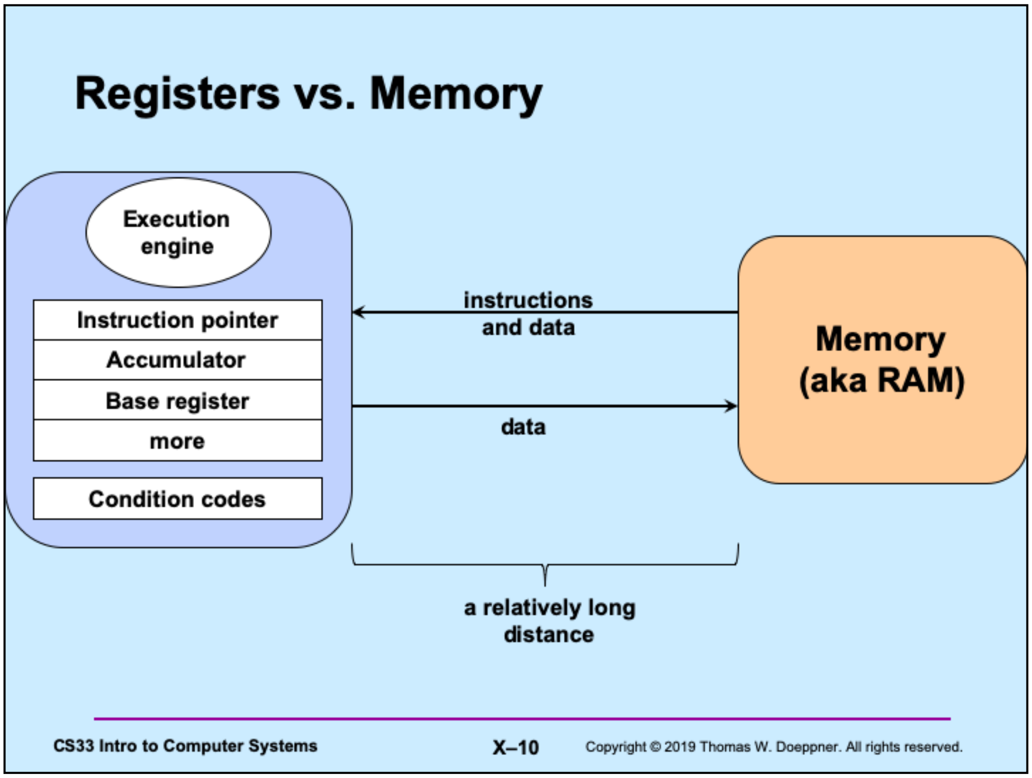
```
mov    1000,%acc
add    -8(%base),%acc
mul    -12(%base),%acc
mov    %acc,-16(%base)
```



Memory



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 32; some other architectures have many more, perhaps as many as 256); memory is measured in gigabytes.

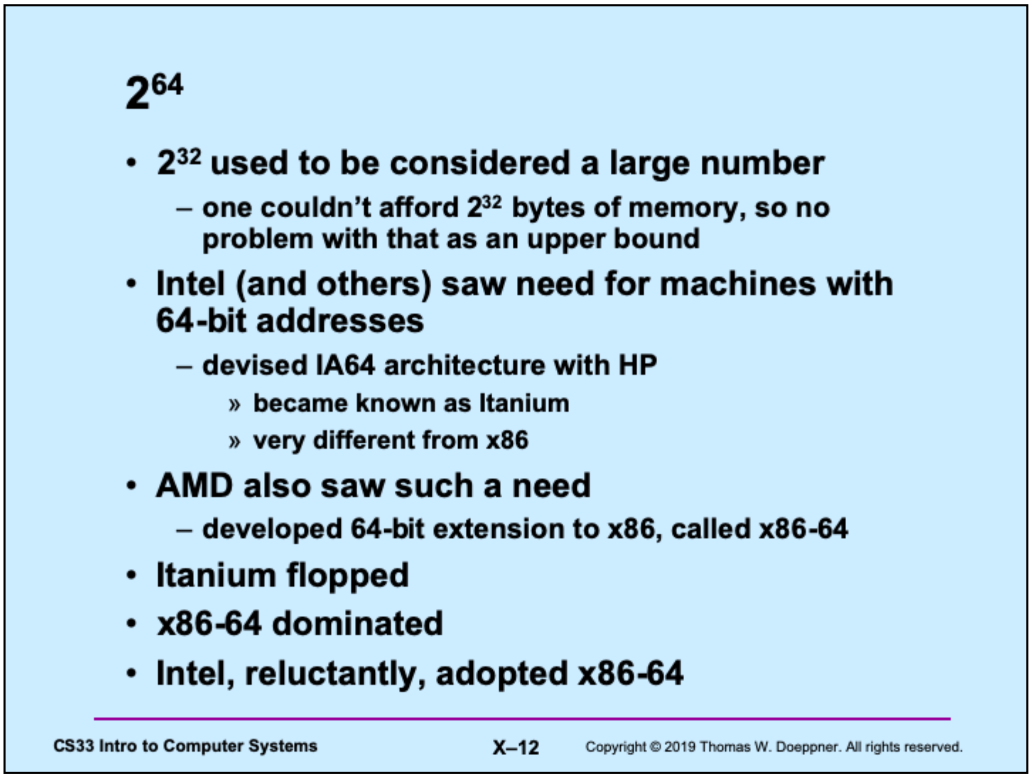
Note that memory access-time is mitigated by the use of on-processor caches, something that we will discuss in a few weeks.

Intel x86

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

IA32

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



2^{32} = 4 gigabytes.

2^{64} = 16 exbibytes

All SunLab computers are x86-64.

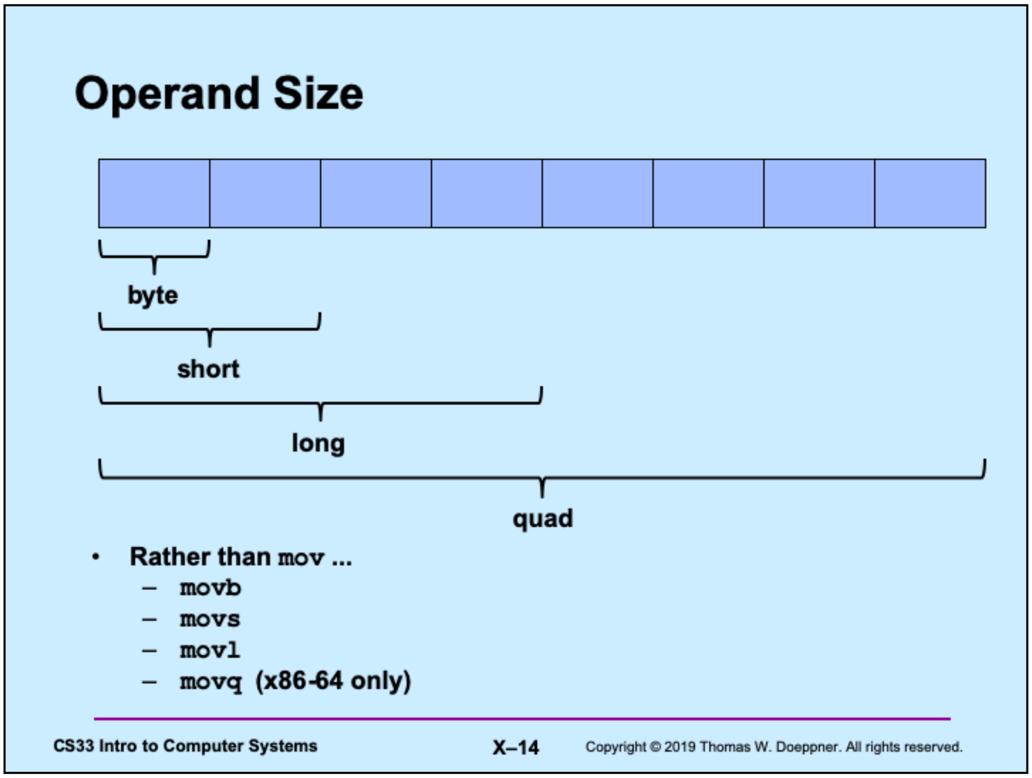
Data Types on IA32 and x86-64

- "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)

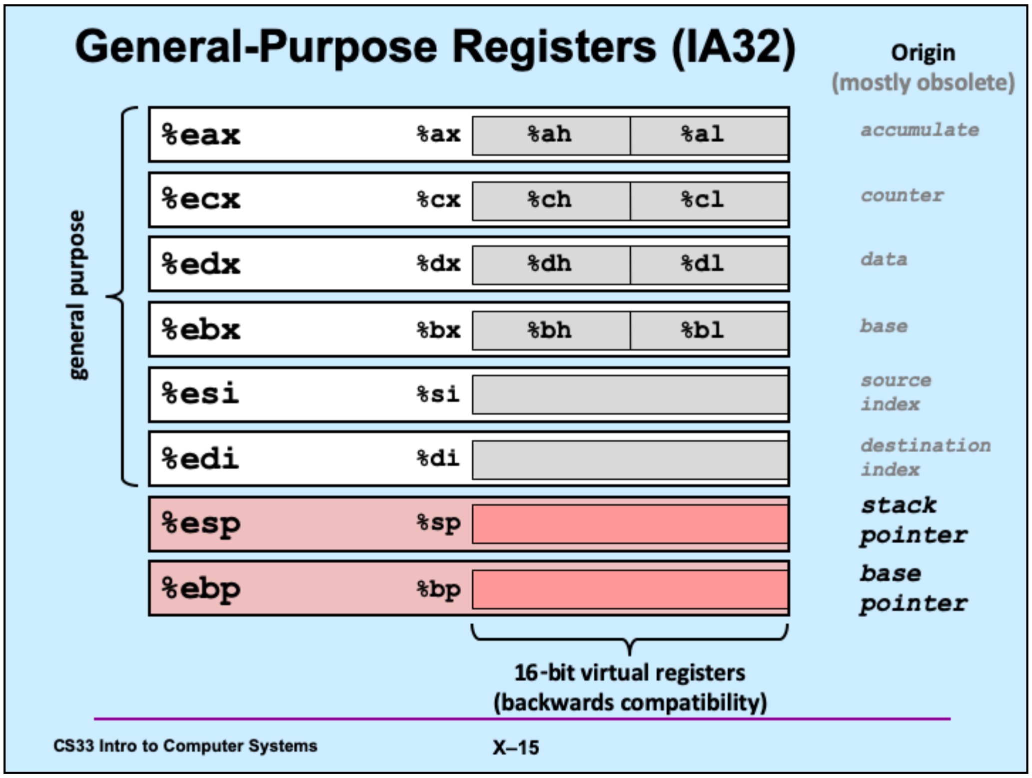
- Floating-point data of 4, 8, or 10 bytes

- No aggregate types such as arrays or structures
  - just contiguously allocated bytes in memory

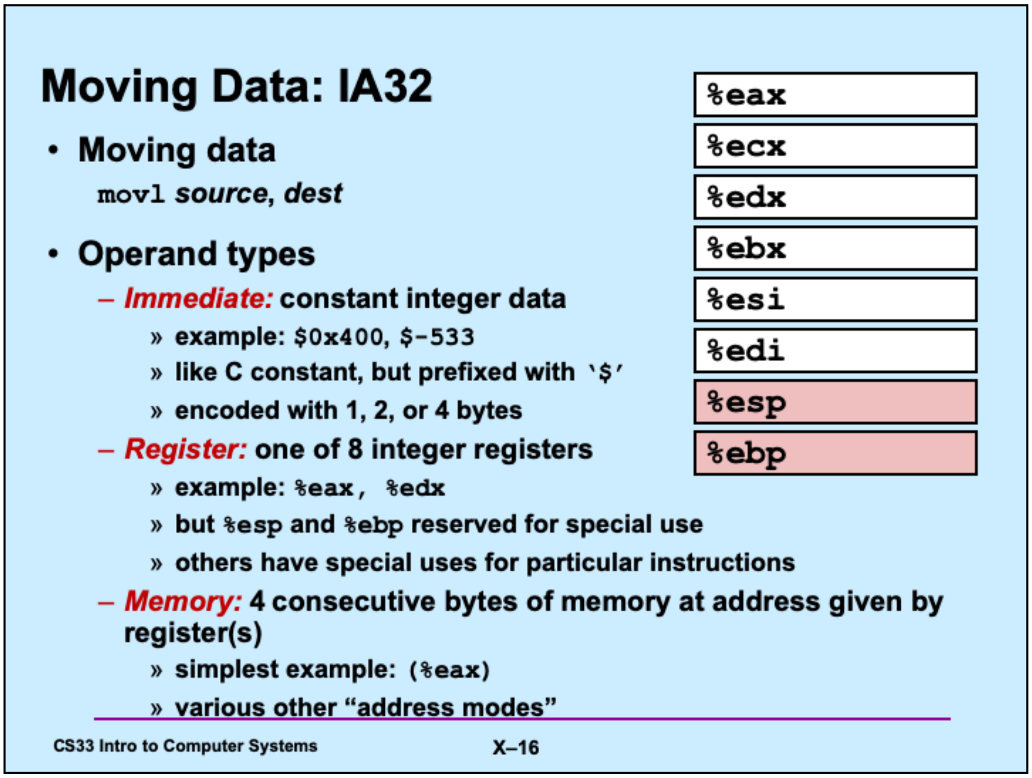
Supplied by CMU.



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



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.

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.

`movl` Operand Combinations

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

```c
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)

   popl  %ebx
   popl  %ebp
   ret
```

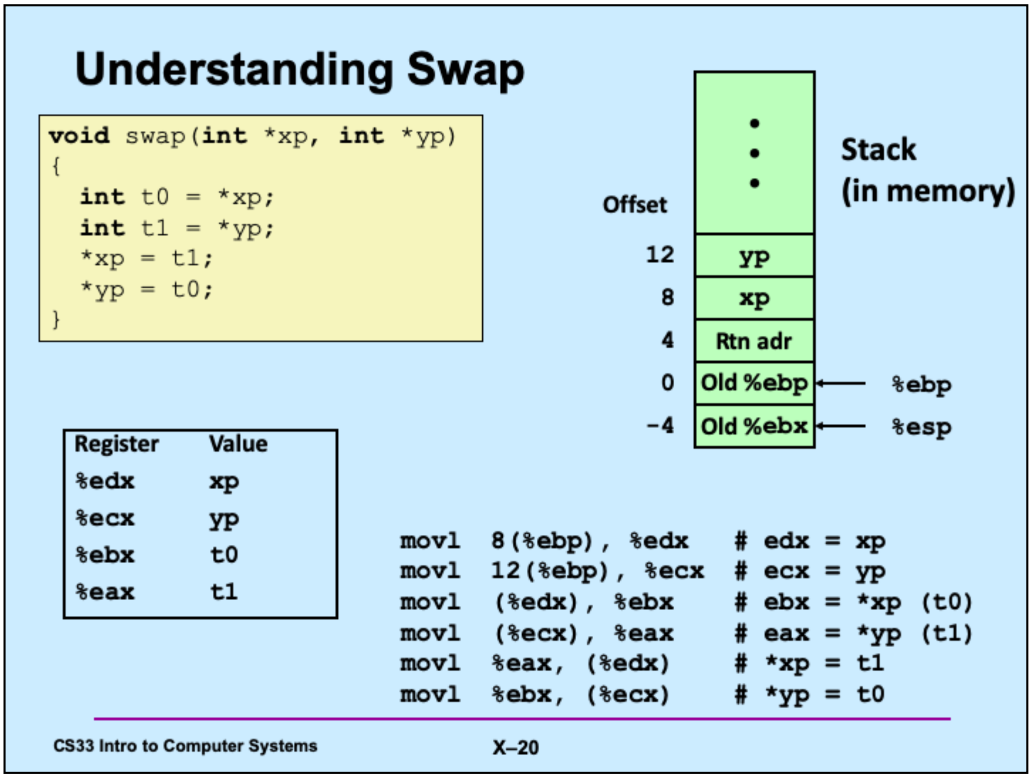
Set Up

Body

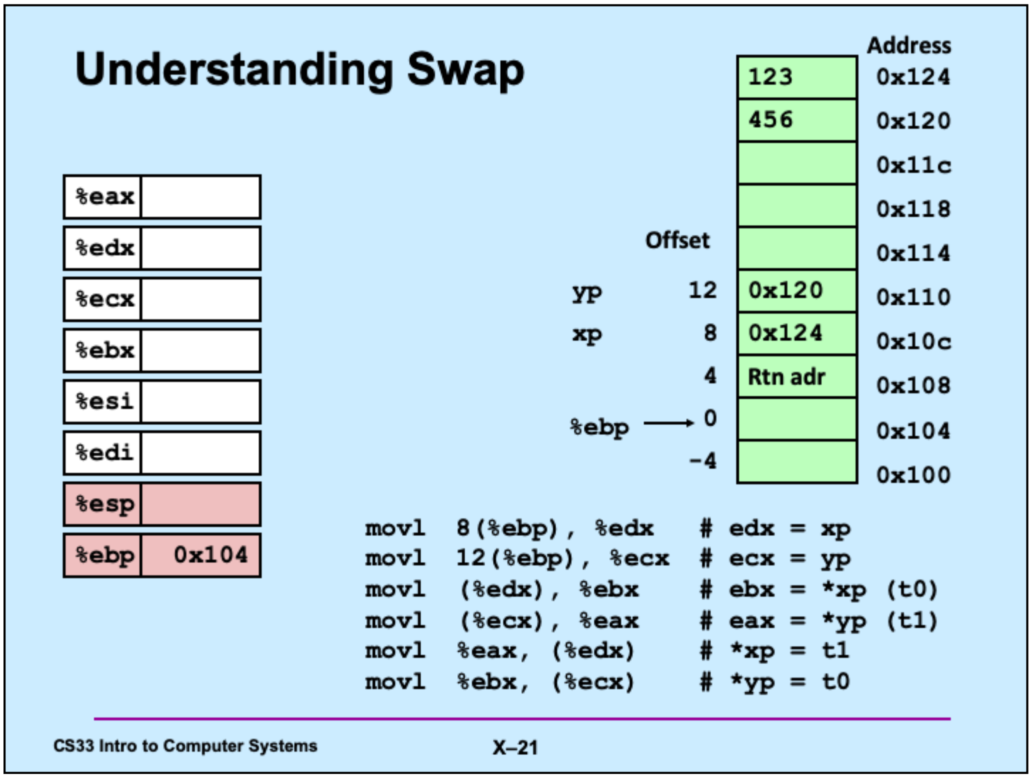
Finish

Supplied by CMU.

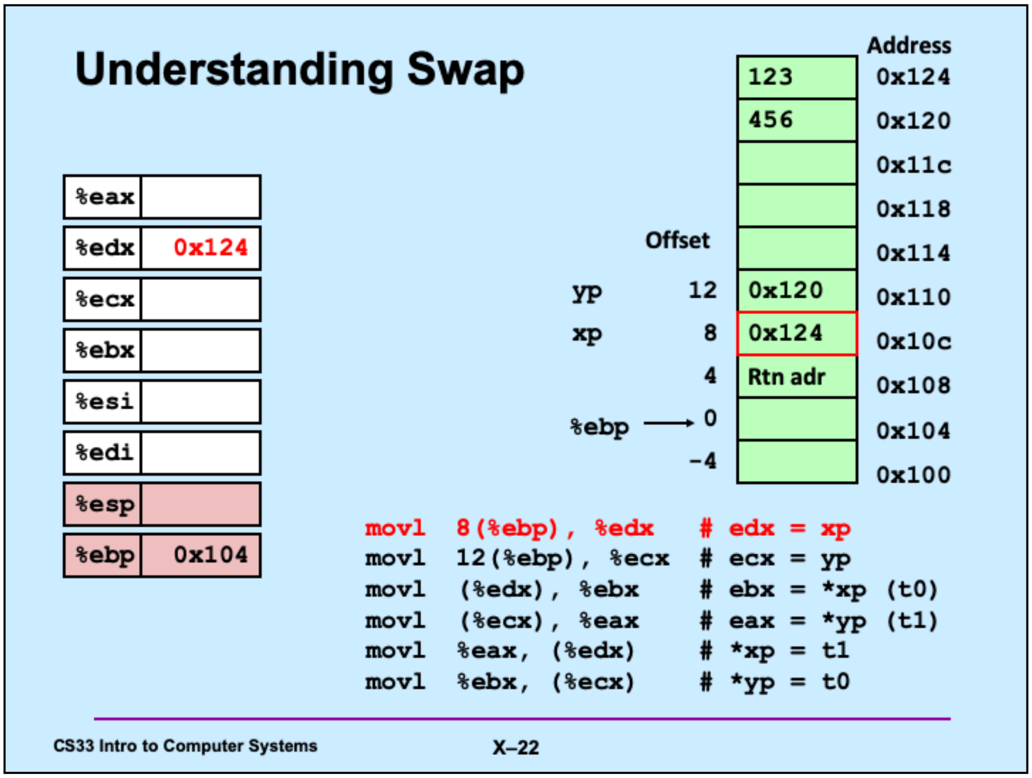
We discuss the “set up” and “finish” in a subsequent lecture. They have to do with facilitating the calling of functions.



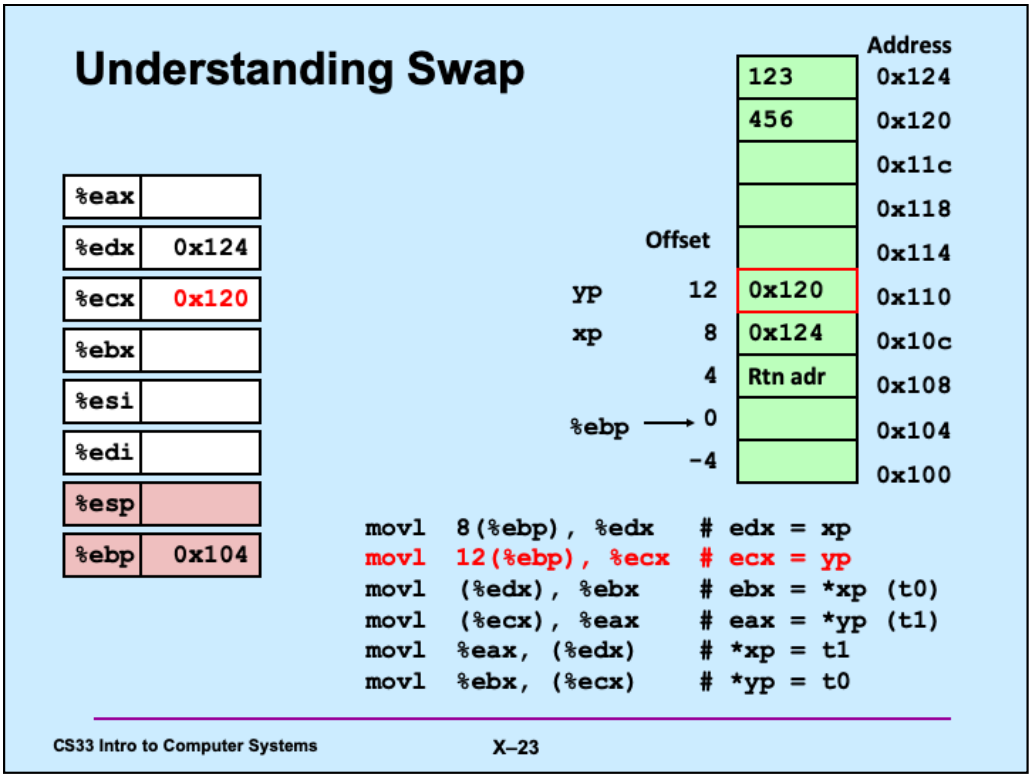
Supplied by CMU.



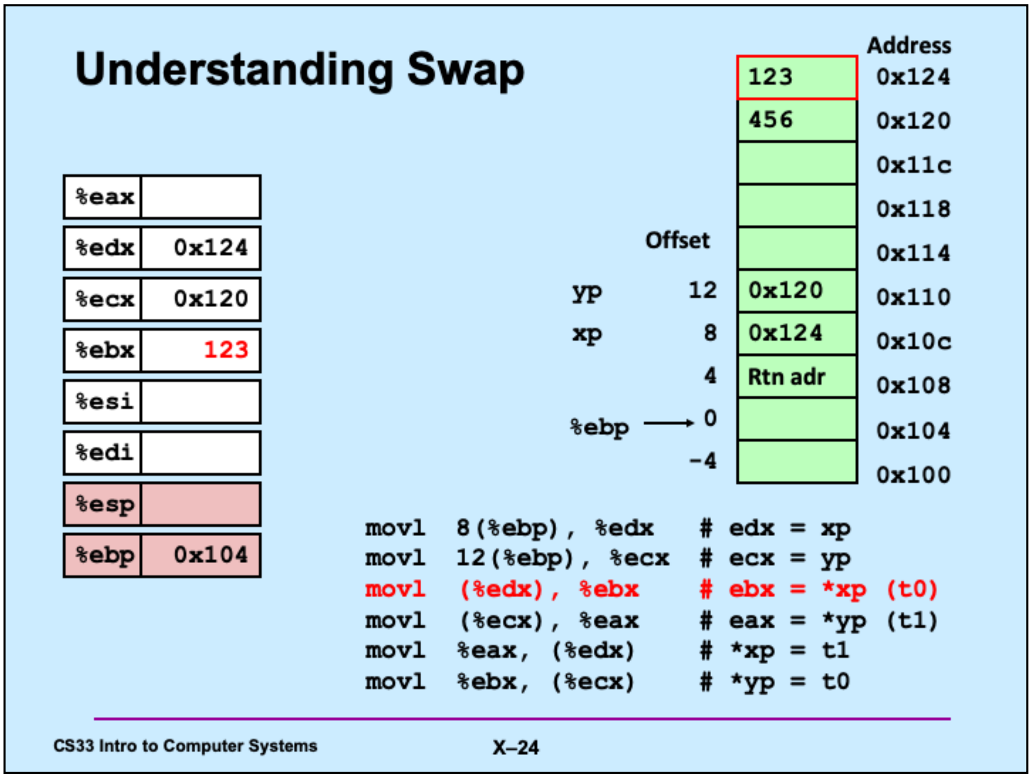
Supplied by CMU.



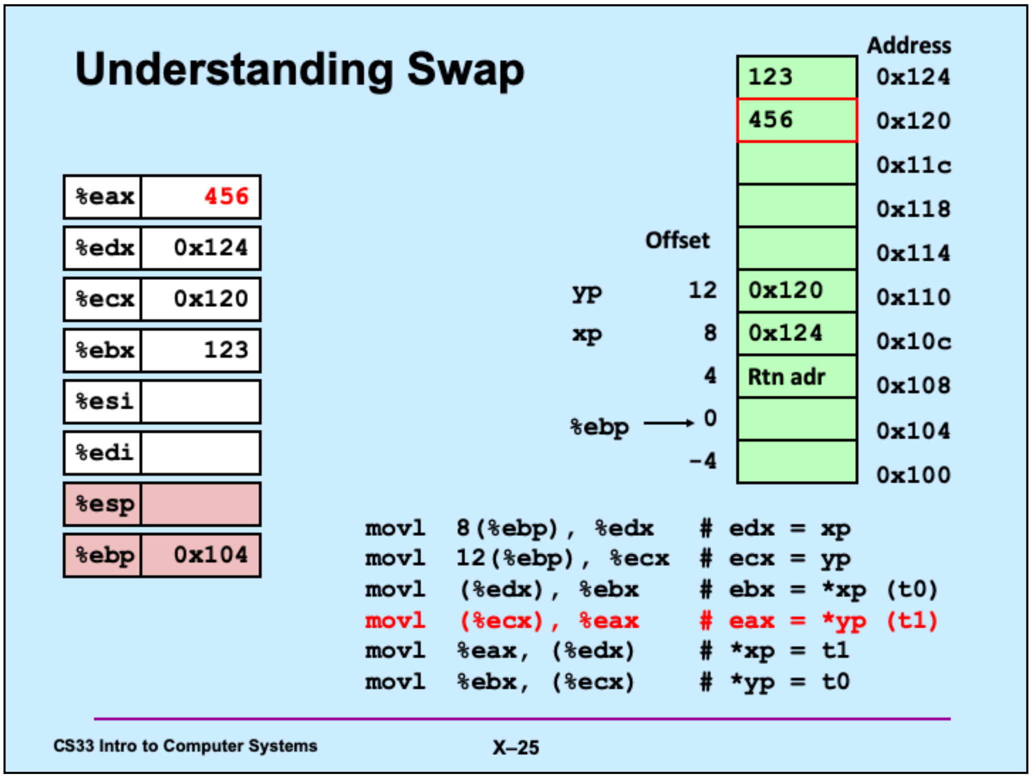
Supplied by CMU.



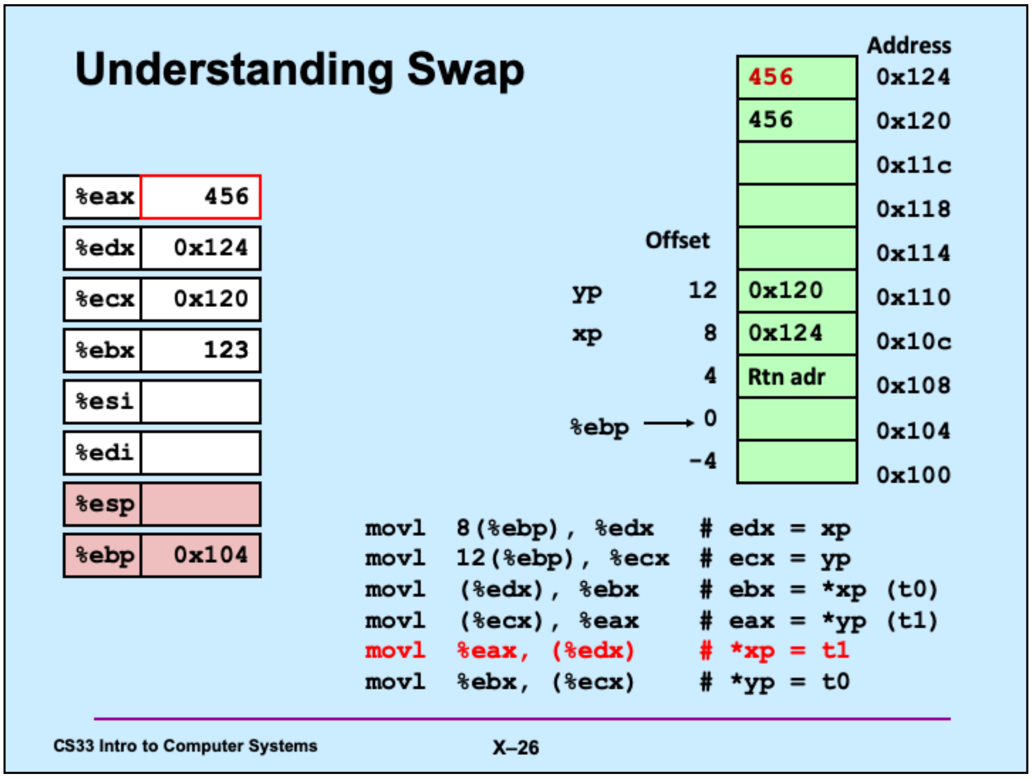
Supplied by CMU.



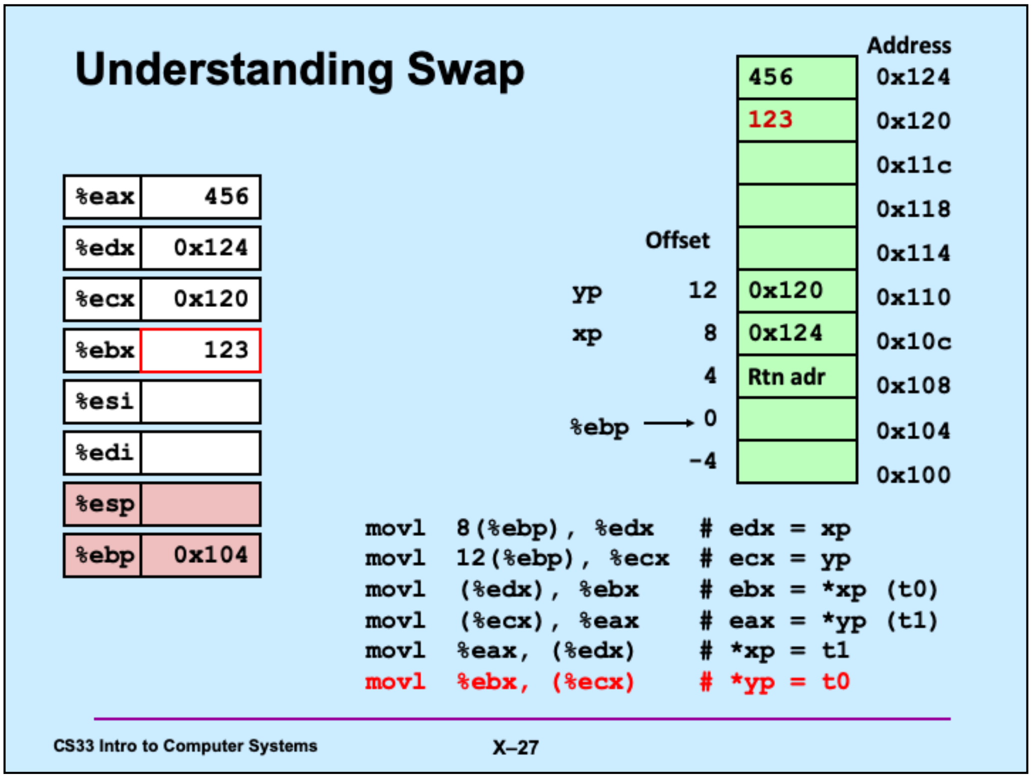
Supplied by CMU.



Supplied by CMU.



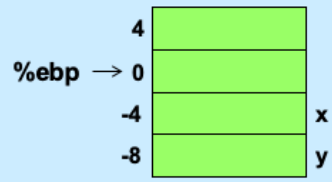
Supplied by CMU.



Supplied by CMU.

Quiz 2

```
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 3

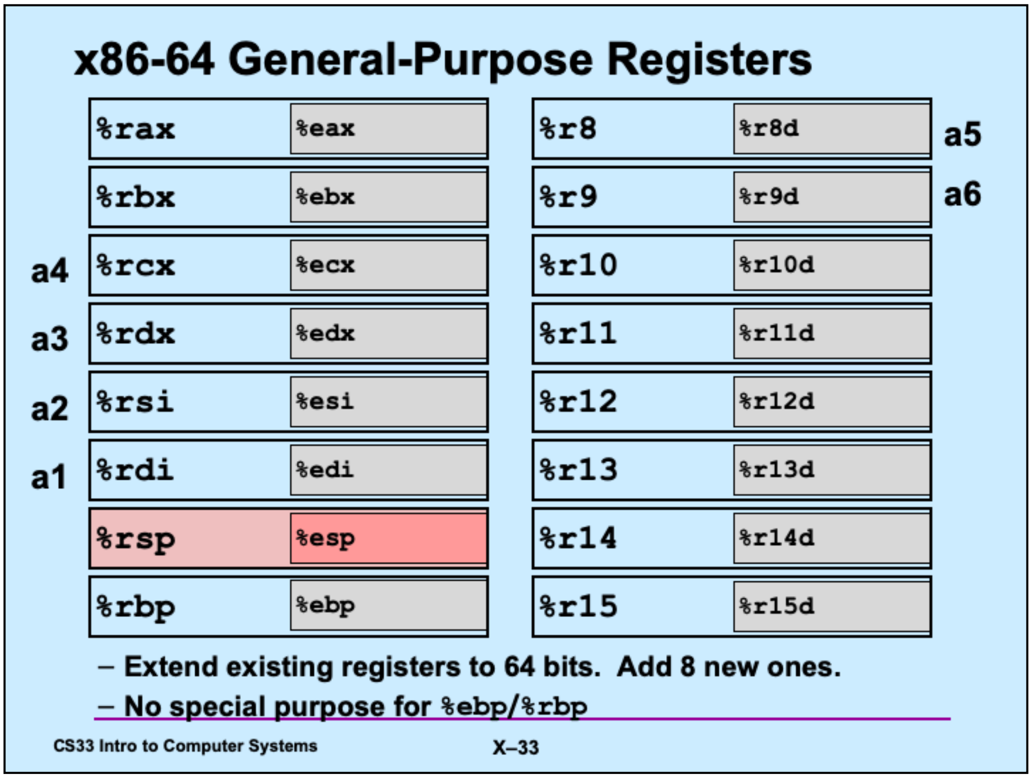
What value ends up in %ecx?

```
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.

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.

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.

Bytes

- **Each register has a byte version**
  - e.g., %r10: %r10b
- **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.

32-bit code for swap

```
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)

  popl  %ebx
  popl  %ebp
  ret
```

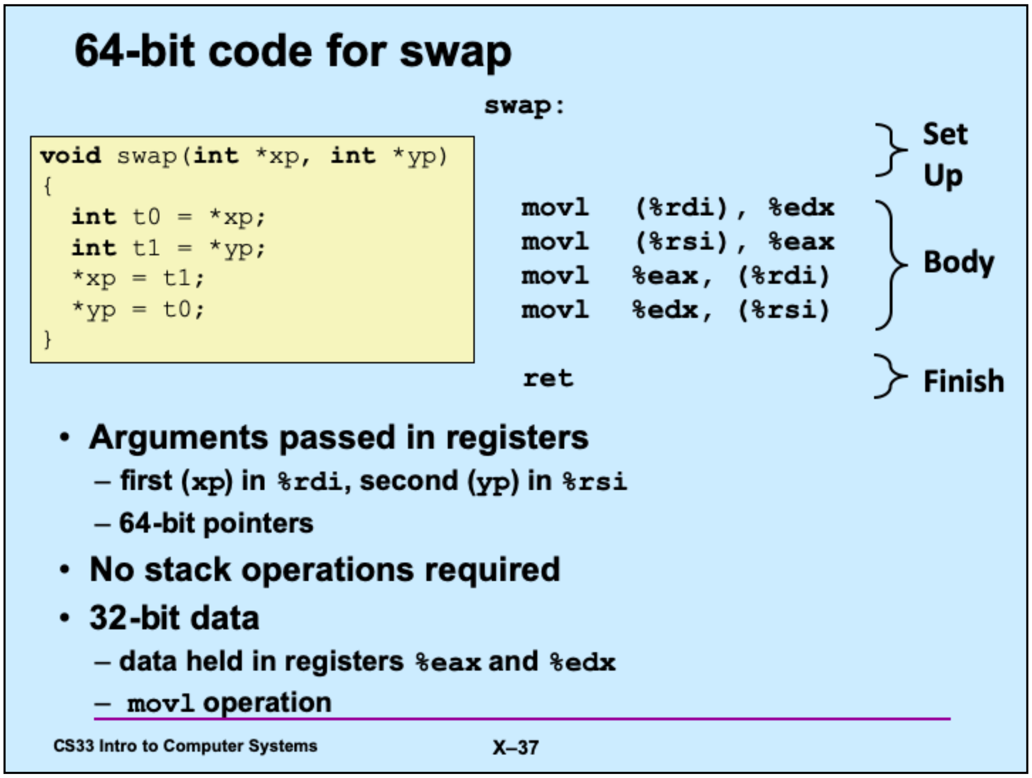
Set Up

Body

Finish

Supplied by CMU.

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



Supplied by CMU.

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.

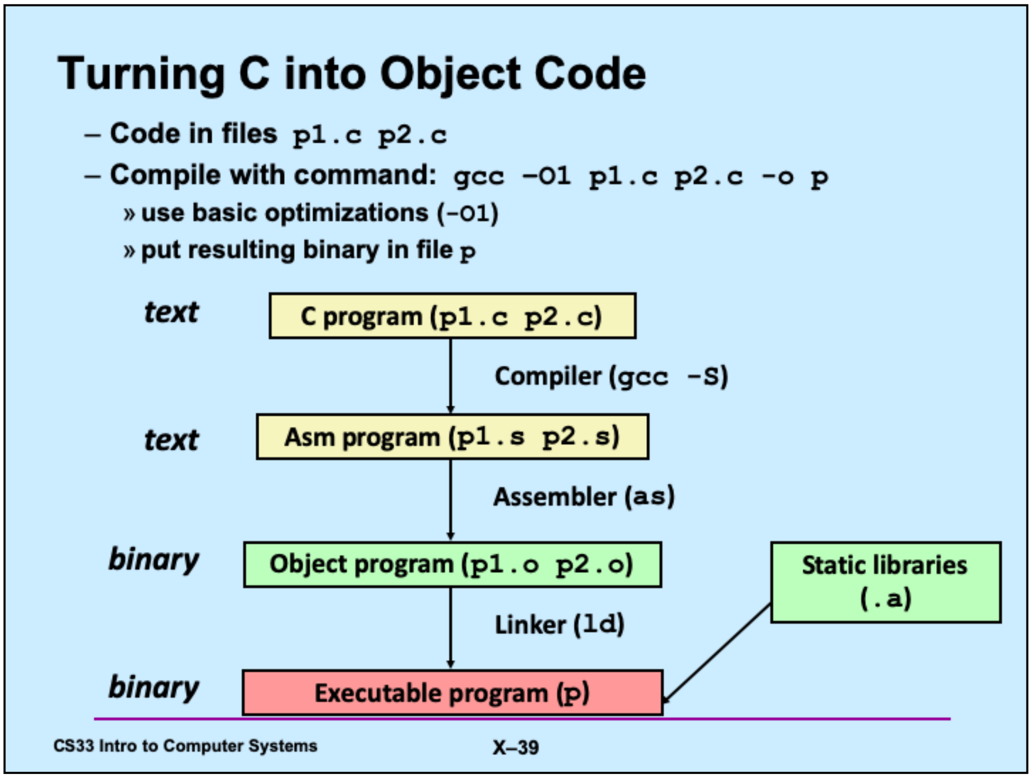
64-bit code for long int swap

```
void swap(long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap_l:
                                  } Set Up
    movq    (%rdi), %rdx
    movq    (%rsi), %rax          } Body
    movq    %rax, (%rdi)
    movq    %rdx, (%rsi)

    ret                           } Finish
```

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

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.

Example

```
int sum(int a, int b) {
    return(a+b);
}
```

Object Code

Code for `sum`

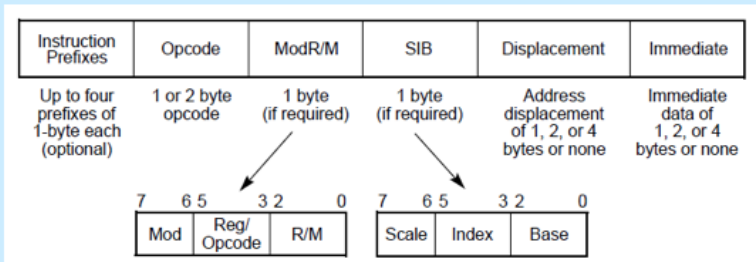
```
0x401040 <sum>:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x5d
    0xc3
```

- 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.

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)

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
    0x8b
    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
```

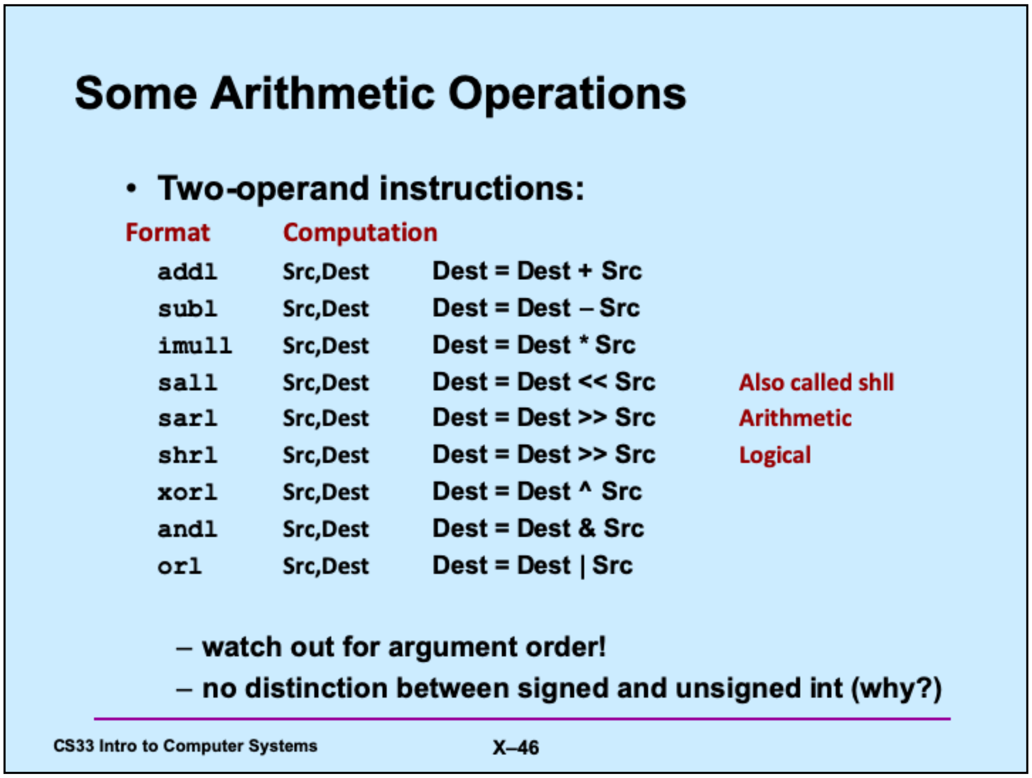
- **Within gdb debugger**
  - `gdb <file>`
  - `disassemble sum`
  - – disassemble procedure
  - `x/11xb sum`
  - – examine the 11 bytes starting at `sum`

Supplied by CMU.

How Many Instructions are There?

- We cover ~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 comes with the caveat that it may be out of date.



Supplied by CMU.

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

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.

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.

Quiz 4

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

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

a) **2**
b) **4**
c) **8**
d) **indeterminate**