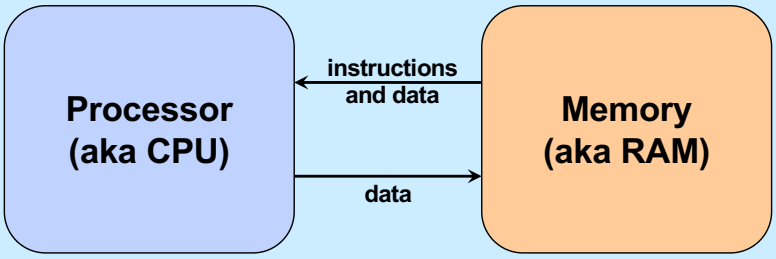
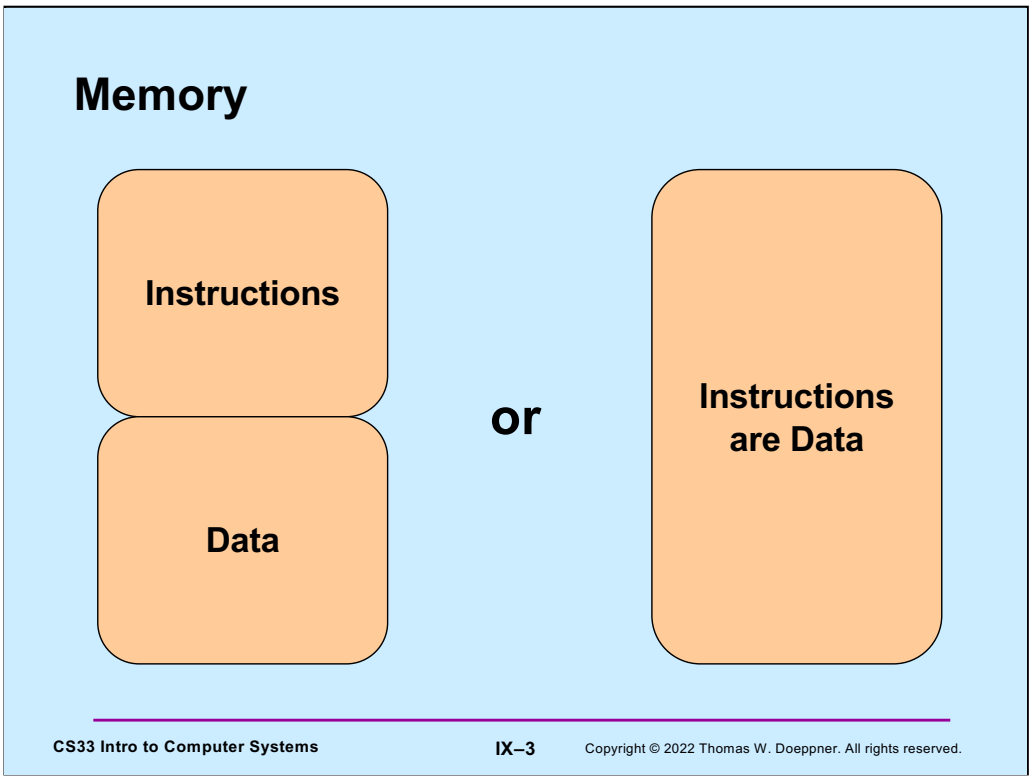
CS 33

Intro to Machine Programming

We begin our discussion of machine programming by covering some of the general principles involved. We look at a generic "machine language" that is similar, but not identical, to that used on Intel processors. After this brief introduction, we focus on the machine language used by Intel processors.

Machine Model

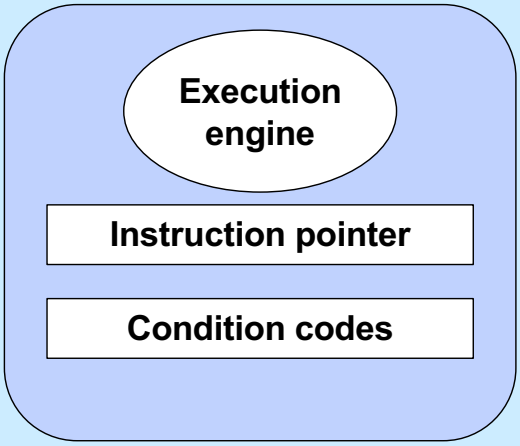




Generally, we think of their being two sorts of memory: that containing instructions and that containing data. Programs, in general, don't modify their own instructions on the fly. In reality, there's only one sort of memory, which holds everything. However, we arrange so that memory holding instructions cannot be modified and that, usually, memory holding data cannot be executed as instructions.

Of course, programs such as compilers and linkers produce executable code as data, but they don't directly execute it.

Processor: Some Details



Processor: Basic Operation

```
while (forever) {
  fetch instruction IP points at
  decode instruction
  fetch operands
  execute
  store results
  update IP and condition code
}
```


Instructions ...

Operands

- Form
  - immediate vs. reference
    - » value vs. address
- How many?
  - 3
    - » add a,b,c
      - c = a + b
  - 2
    - » add a,b
      - b += a

Operands (continued)

- Accumulator
  - special memory in the processor
    - » known as a *register*
    - » fast access
  - allows single-operand instructions
    - » add a
      - acc += a
    - » add b
      - acc += b

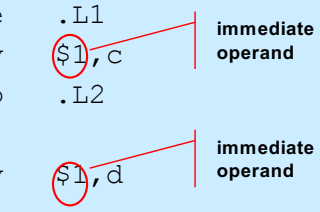
From C to Assembler ...

```
a = (b + c) * d;

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

```
if (a<b)
    c = 1;
else
    d = 1;

    cmp   a,b
    jge   .L1
    mov   $1,c
    jmp   .L2
.L1
    mov   $1,d
.L2
```



Note we're using the accumulator in two-operand instructions. The "%" makes it clear that "acc" is a register. The "$" indicates that what follows is an immediate operand; i.e., it's a value to be used as is, rather than as an address or a register.

Condition Codes

- Set of flags giving status of most recent operation:
  - zero flag
    - » result was zero
  - sign flag
    - » for signed arithmetic interpretation: sign bit is set
  - overflow flag
    - » for signed arithmetic interpretation
  - carry flag (generated by carry or borrow out of most-significant bit)
    - » for unsigned arithmetic interpretation
- Set implicitly by arithmetic instructions
- Set explicitly by compare instruction
  - cmp a,b
    - » sets flags based on result of b-a

We have one set of arithmetic instructions that work with both unsigned and signed (two's complement) interpretations of the bit values in a word.

The overflow flag is set when the result, interpreted as a two's-complement value should be positive, but won't fit in the word and thus becomes a negative number, or should be negative, but won't fit in the word and thus becomes a positive number.

The carry flag is set when computing the result, interpreted as an unsigned value, requires a borrow out of the most-significant bit (i.e., computing b-a when a is greater than b), or when it results in an overflow (e.g., for 32-bit unsigned integers, when the result should be greater than or equal to 2^{32} (but can't fit in a 32-bit word).

Examples (1)

- Assume 32-bit arithmetic
- x is `0x80000000`
  - TMIN if interpreted as two's-complement
  - 2^{31} if interpreted as unsigned
- x-1 (`0x7fffffff`)
  - TMAX if interpreted as two's-complement
  - $2^{31}-1$ if interpreted as unsigned
  - zero flag is not set
  - sign flag is not set
  - overflow flag is set
  - carry flag is not set

Examples (2)

- x is `0xffffffff`
  - -1 if interpreted as two's-complement
  - UMAX (2^{32}-1) if interpreted as unsigned
- x+1 (`0x00000000`)
  - zero under either interpretation
  - zero flag is set
  - sign flag is not set
  - overflow flag is not set
  - carry flag is set

Examples (3)

- x is `0xffffffff`
  - -1 if interpreted as two's-complement
  - UMAX (2^{32}-1) if interpreted as unsigned
- x+2 (`0x00000001`)
  - (+)1 under either interpretation
  - zero flag is not set
  - sign flag is not set
  - overflow flag is not set
  - carry flag is set

Quiz 1

- Set of flags giving status of most recent operation:
  - zero flag
    - » result was zero
  - sign flag
    - » for signed arithmetic interpretation: sign bit is set
  - overflow flag
    - » for signed arithmetic interpretation
  - carry flag (generated by carry or borrow out of most-significant bit)
    - » for unsigned arithmetic interpretation
- Set explicitly by compare instruction
  - cmp a,b
    - » sets flags based on result of b-a

Which flags are set to one by "cmp 2,1"?

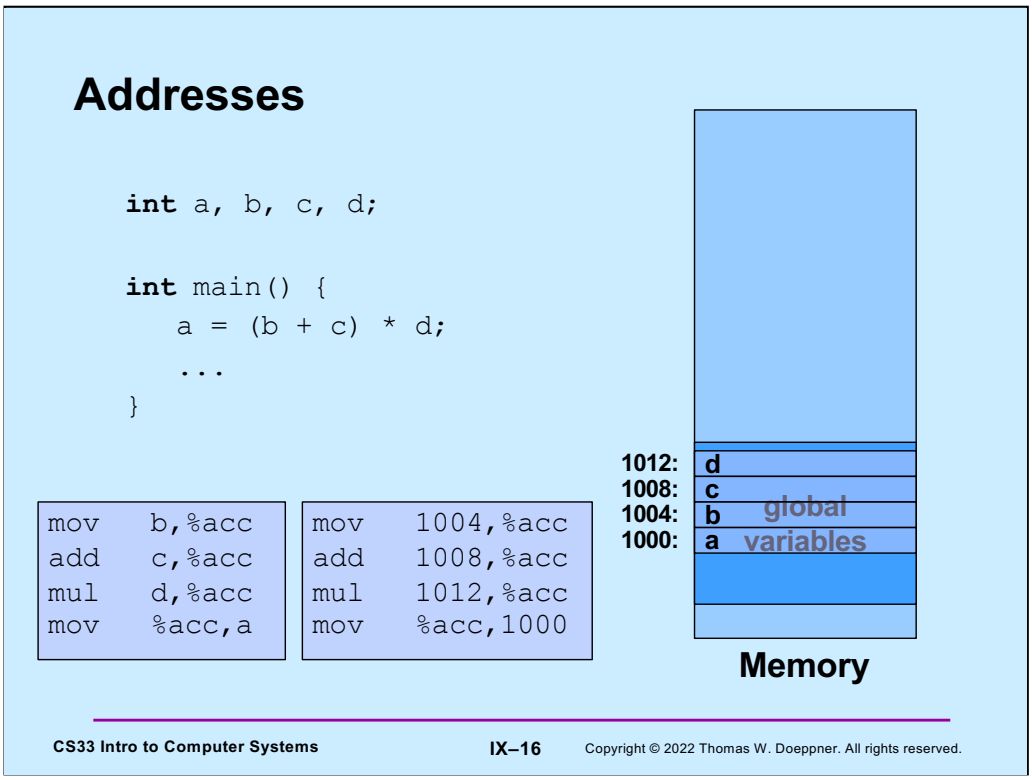
a) overflow flag only
b) carry flag only
c) sign and carry flags only
d) sign and overflow flags only
e) sign, overflow, and carry flags

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.



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

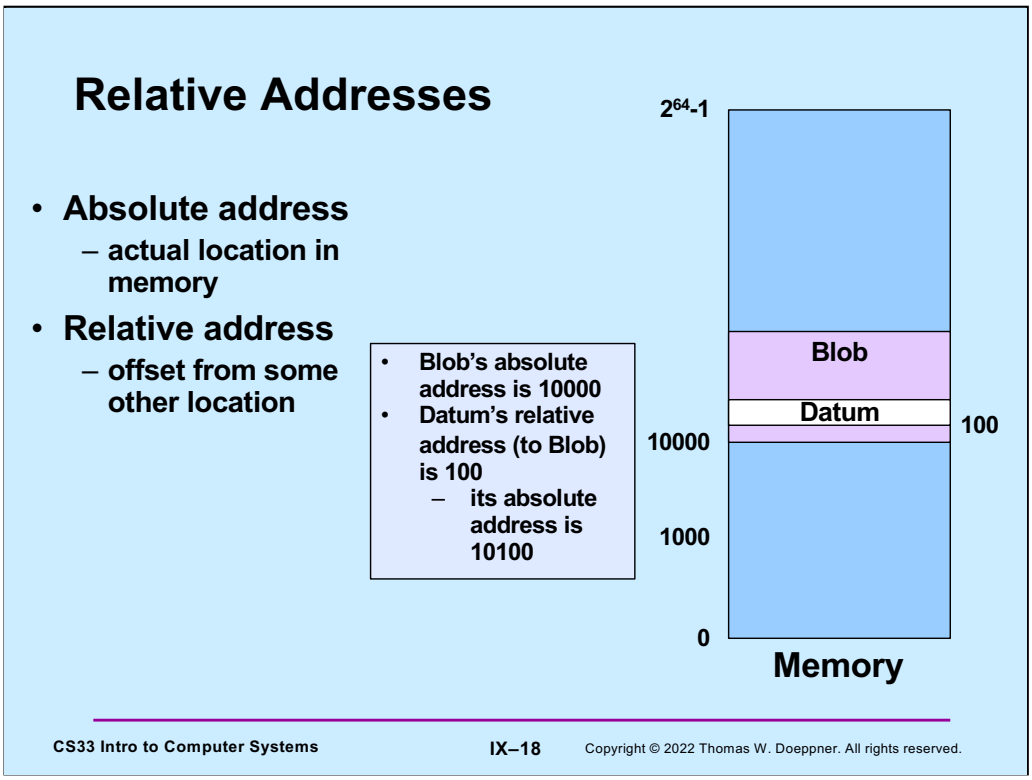
```
int b;

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

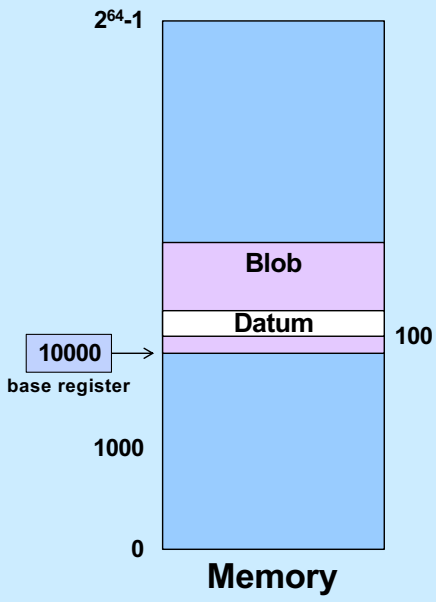
Here we rearrange things a bit. **b** is a global variable, but a is a local variable within **func**, and **c** and **d** are arguments. The issue here is that the locations associated with **a**, **c**, and **d** will, in general, be different for each call to **func**. Thus, we somehow must modify the assembler code to take this into account.



Note that both positive and negative offsets might be used.

Base Registers

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



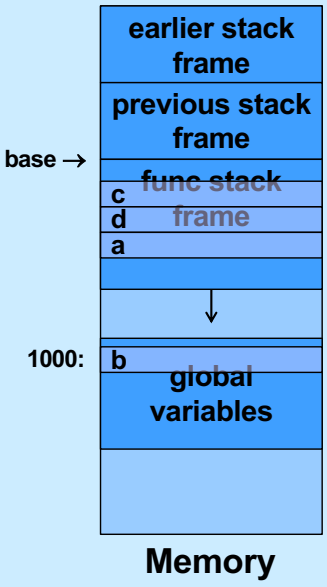
Here we load the value 10,000 into the base register (recall that the "$" means what follows is a literal value; a "%" sign means that what follows is the name of a register), then store the value 10 into the memory location 10100 (the contents of the base register plus 100): the notation **n(%base)** means the address obtained by adding **n** to the contents of the base register.

Addresses

```
int b;

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

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



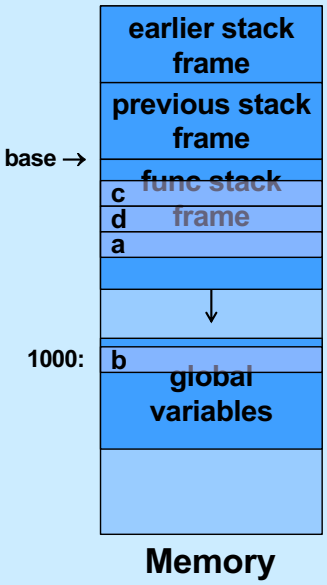
Here we return to our earlier example. We assume that, as part of the call to **func**, the base register is loaded with the address of the beginning of **func**'s current stack frame, and that the local variable **a** and the parameters **c** and **d** are located within the frame. Thus, we refer to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

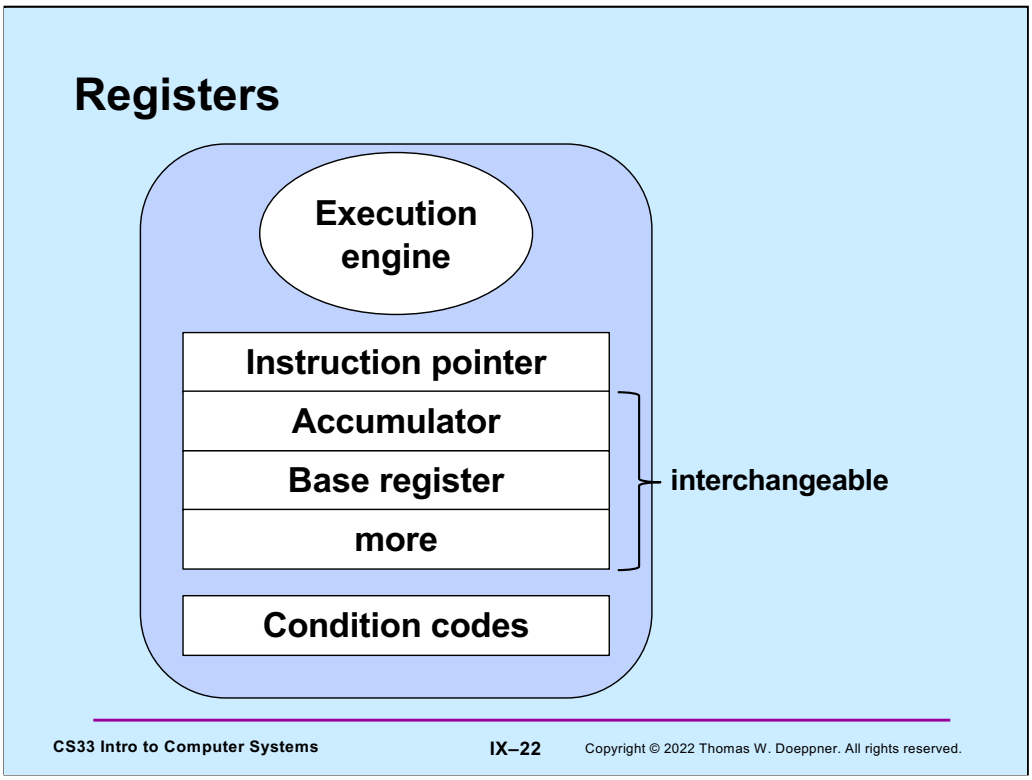
Quiz 2

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

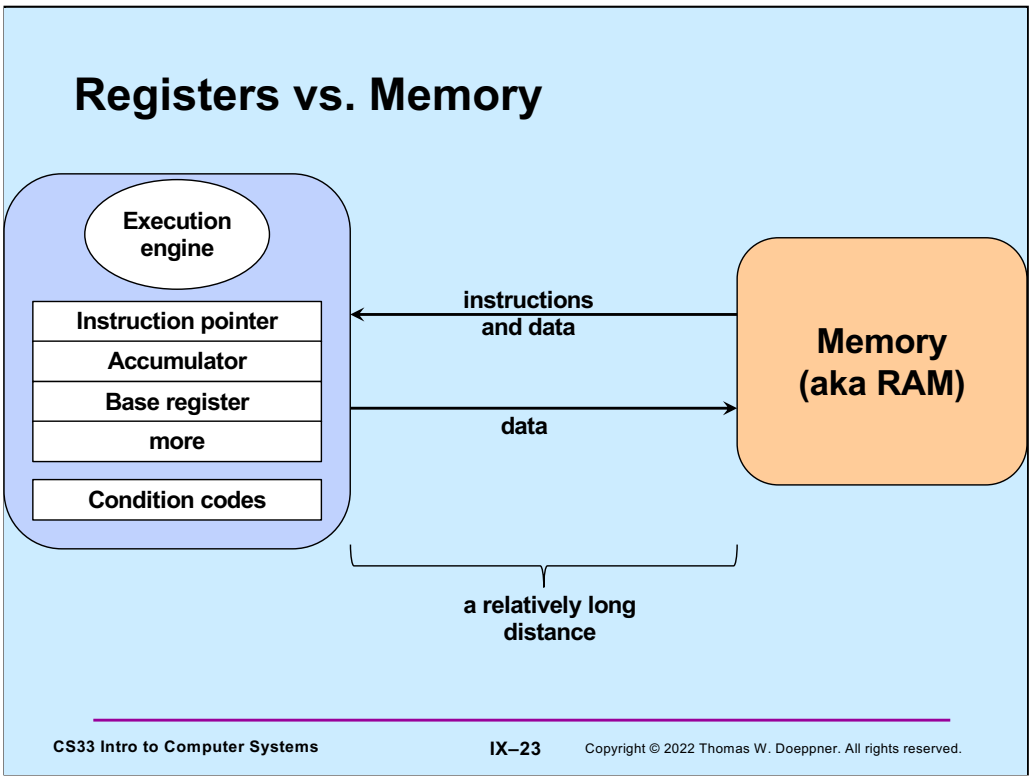
a) 10,008
b) 10,004
c) 9996
d) 9992

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





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.

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^{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.

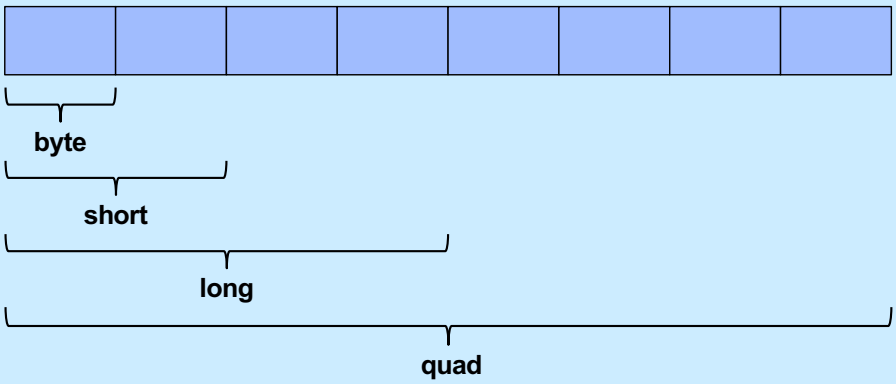
Apple (whose computers originally ran Motorola 68000 processors before they switched to Intel) decided that they could make more cost-effective and faster processors by adapting the ARM design and including GPUs (graphics processing units). GPUs are specialized processors that help with image processing, 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

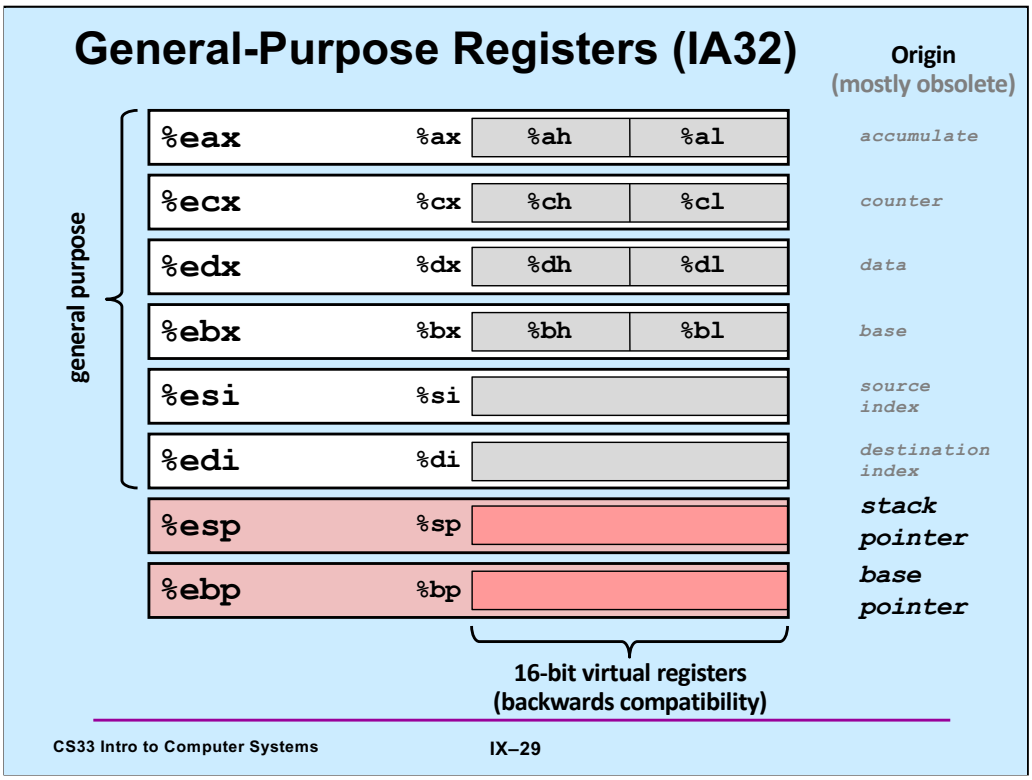


- Rather than `mov` ...
  - `movb`
  - `movs`
  - `movl`
  - `movq` (x86-64 only)

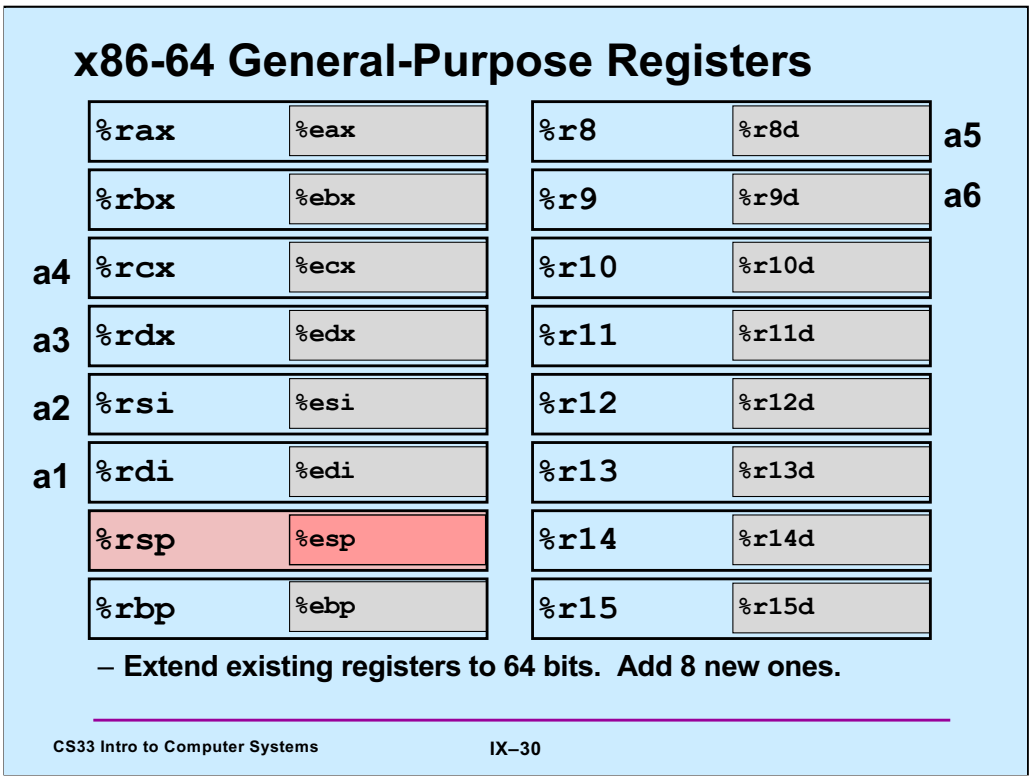
Most instructions come in three (on IA32) or four (on x86-64) forms, one for each possible operand size.

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.



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.

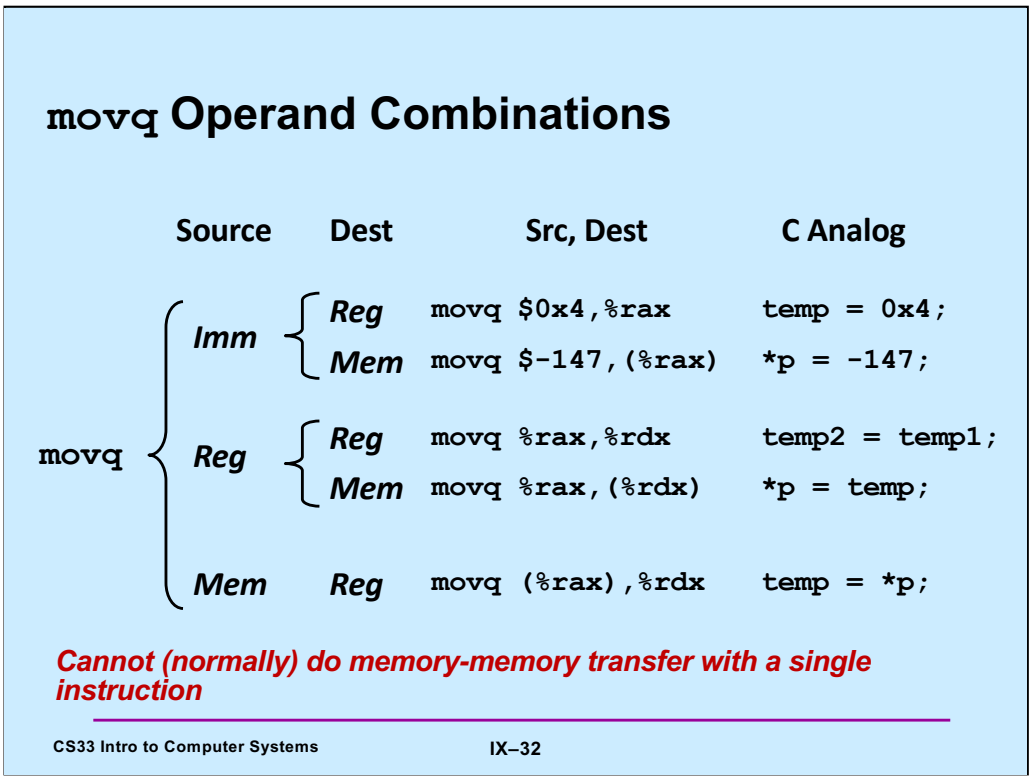
Moving Data

- **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.



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

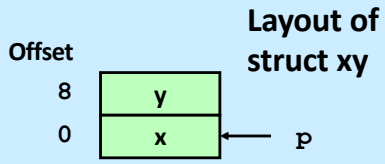
```
struct xy {
  long x;
  long y;
}
void swapxy(struct xy *p){
  long temp = p->x;
  p->x = p->y;
  p->y = temp;
}
```

```
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;
}
```

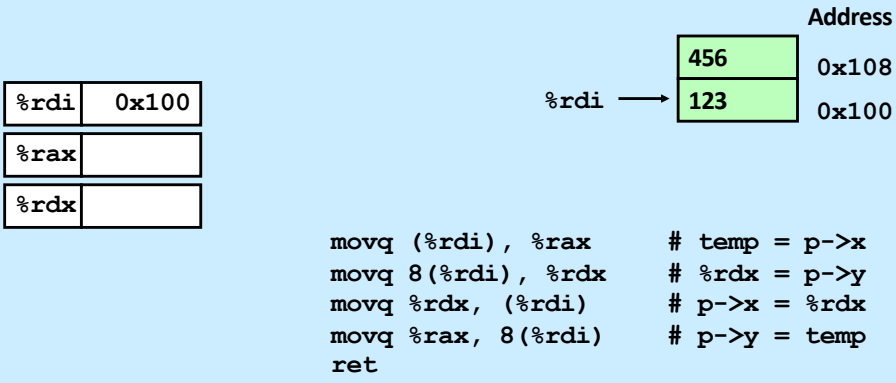


Register	Value
%rdi	p
%rax	temp
%rdx	p->y

```
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.

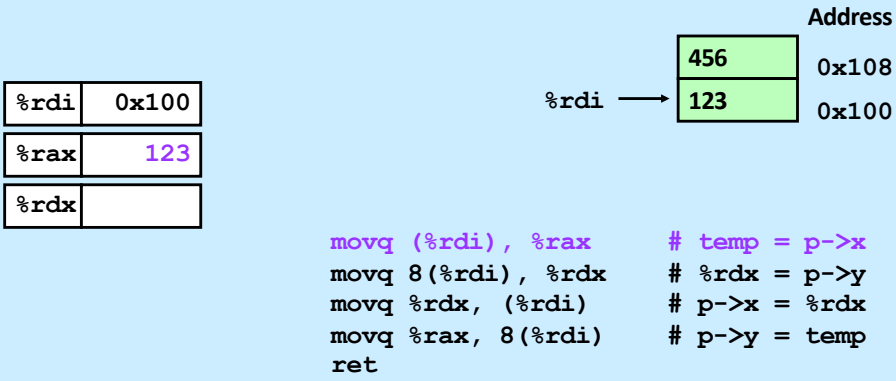
Understanding Swapxy




```
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.

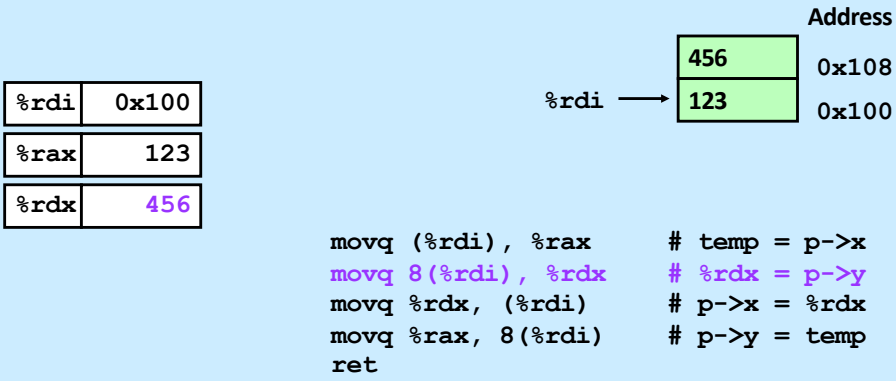
Understanding Swapxy




```
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

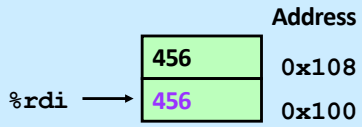



```
movq (%rdi), %rax       # temp = p->x
movq 8(%rdi), %rdx      # %rdx = p->y
movq %rdx, (%rdi)       # p->x = %rdx
movq %rax, 8(%rdi)      # p->y = temp
ret
```


We then copy the second component into %rdx.

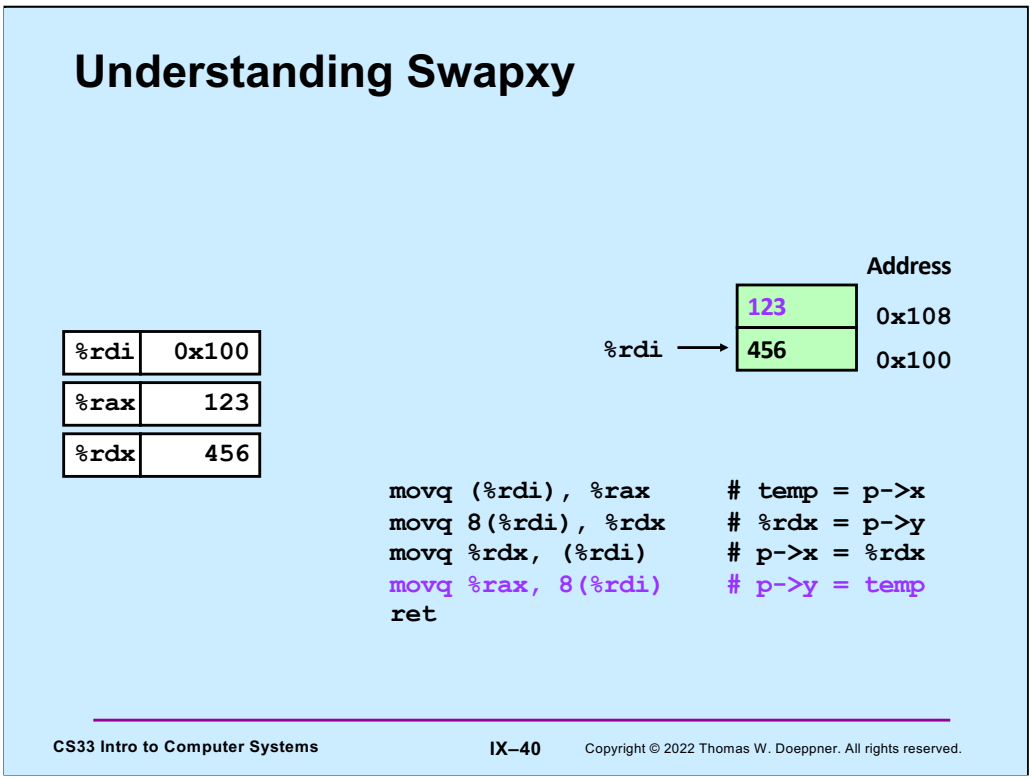
Understanding Swapxy

%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
```

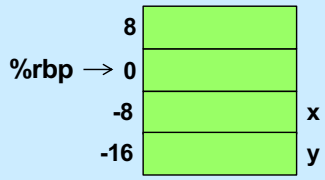

The second component, which we'd copied into %rdx, is now copied into the the first component of the structure itself.



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

Quiz 3

```
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

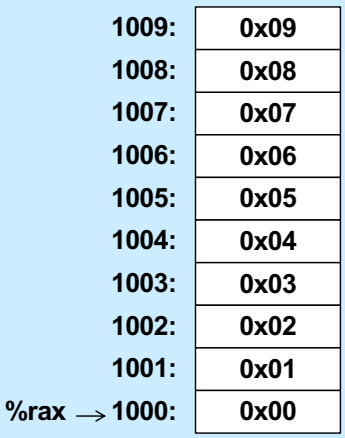
On x86-64, for instructions with 32-bit (long) operands that produce 32-bit results going into a register, the register must be a 32-bit register; the higher-order 32 bits are filled with zeroes.

Quiz 4

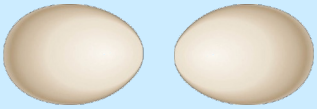
What value ends up in %ecx?

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

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



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
```

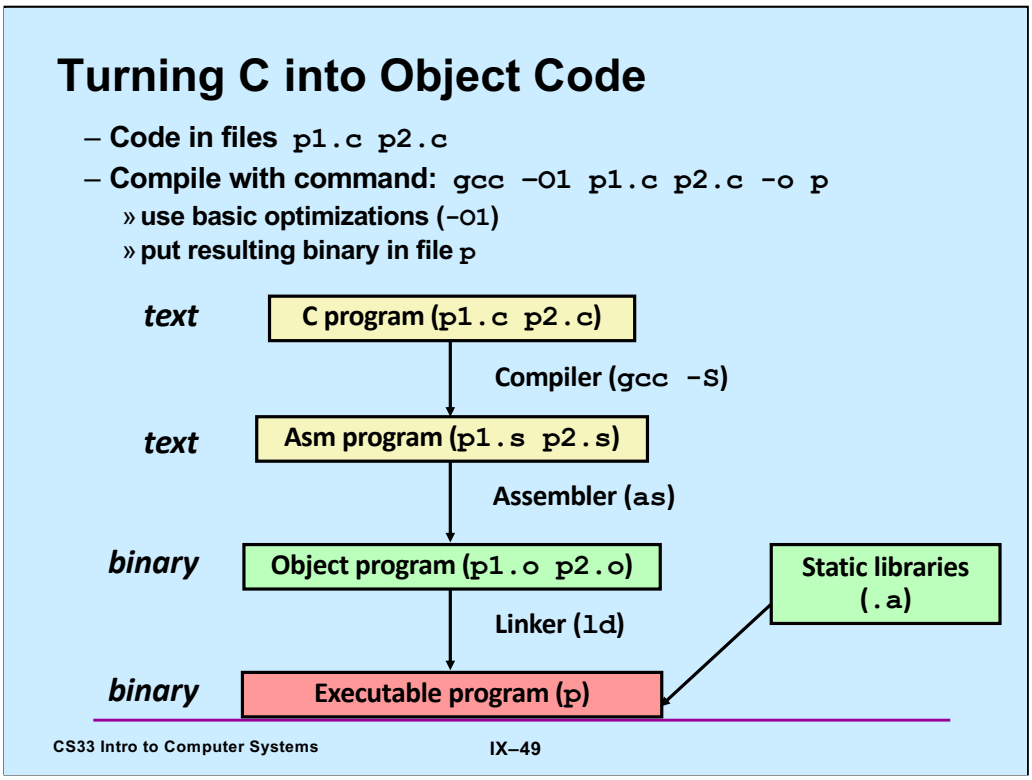
- **Pointers are 64 bits**
- **What they point to are 32 bits**

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;
}
```


Object Code

Code for `ASum`

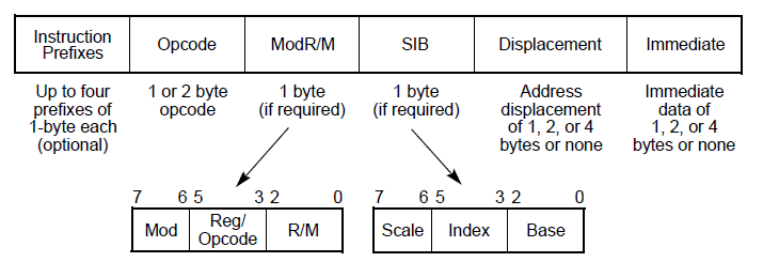
```
0x112b <ASum>:
    0x48
    0x85
    0xf6
    0x74
    0x19
    0x48
    0x89
    0xfa
    0x48
    0x8d
    0x0c
    0xf7
     .
     .
     .
```

- 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.

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

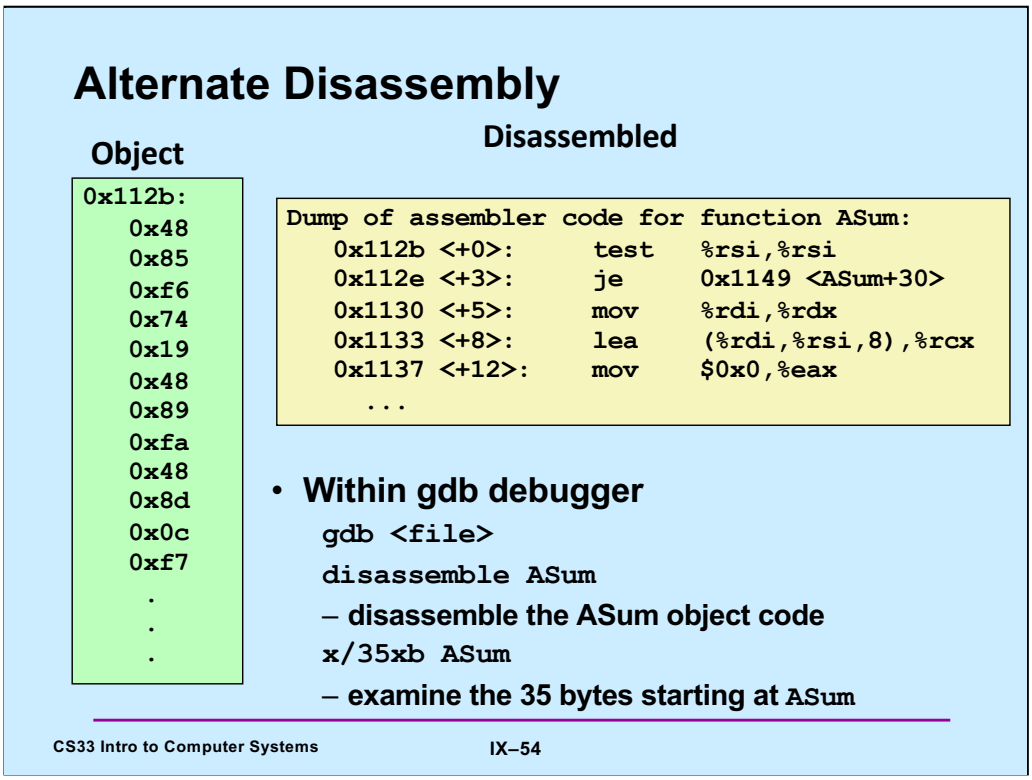
Disassembled

```
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.



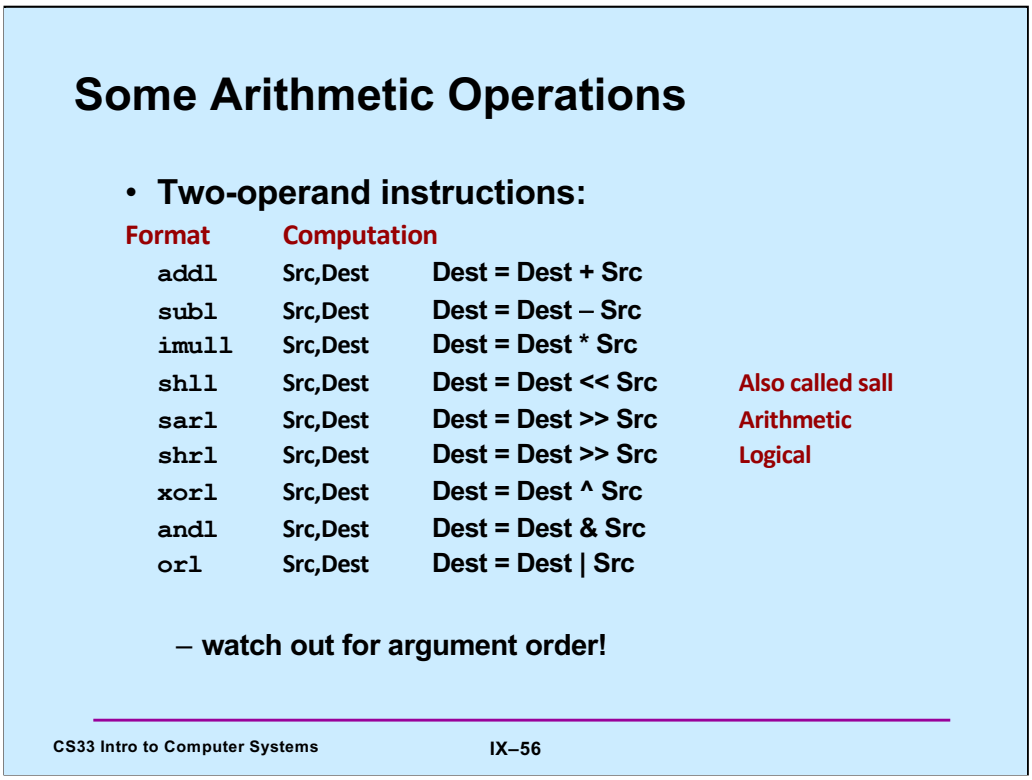
Adapted from a slide supplied by CMU.

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!



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.

Quiz 5

- **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.