CS 33

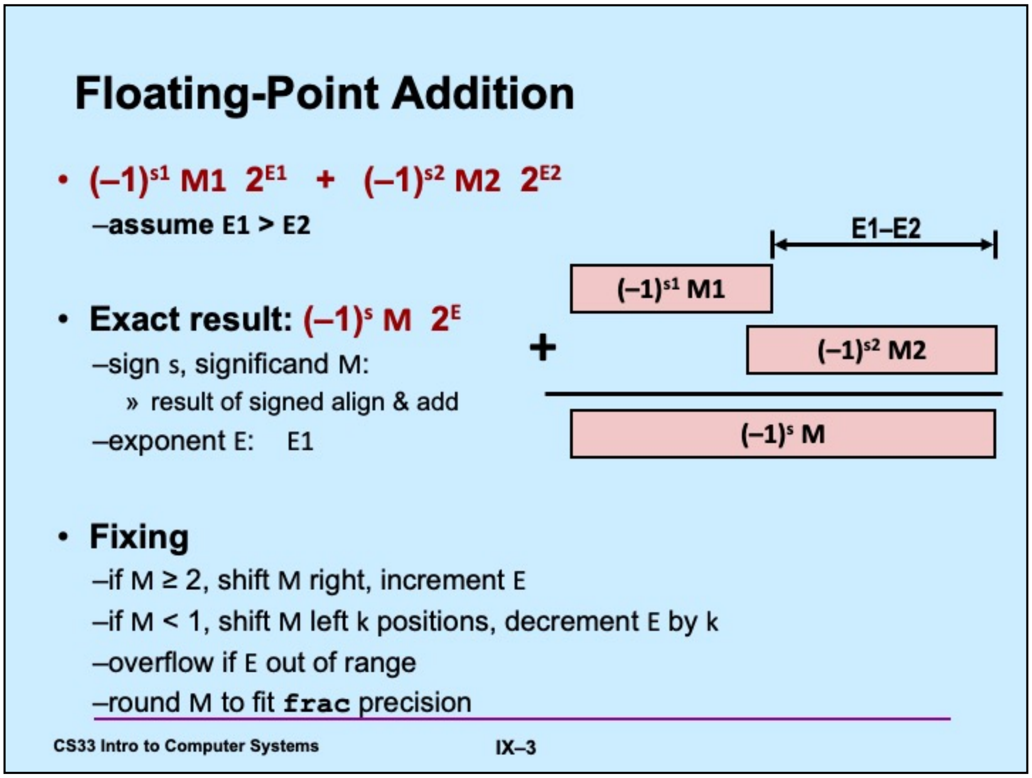
Data Representation (Part 3)

Floating-Point Multiplication

- $(-1)^{s1}$ M1 2^{E1} x $(-1)^{s2}$ M2 2^{E2}
- **Exact result:** $(-1)^{s}$ M 2^{E}
  - sign s: s1 ^ s2
  - significand M: M1 x M2
  - exponent E: E1 + E2
- **Fixing**
  - if M ≥ 2, shift M right, increment E
  - if E out of range, overflow (or underflow)
  - round M to fit `frac` precision
- **Implementation**
  - biggest chore is multiplying significands

Supplied by CMU.

Note that to compute E, one must first convert exp_1 and exp_2 to E_1 and E_2, then add them them together and check for underflow or overflow (corresponding to -∞ and +∞), and then convert to exp.



Supplied by CMU.

Note that, by default, overflow results in either +∞ or -∞.

Floating Point

- **Single precision (float)**

s	exp	frac
1	8-bits	23-bits

  - **range: $\pm 1.8\times10^{-38}$ – $\pm 3.4\times10^{38}$, ~7 decimal digits**

- **Double Precision (double)**

s	exp	frac
1	11-bits	52-bits

  - **range: $\pm 2.23\times10^{-308}$ – $\pm 1.8\times10^{308}$, ~16 decimal digits**

Floating Point in C

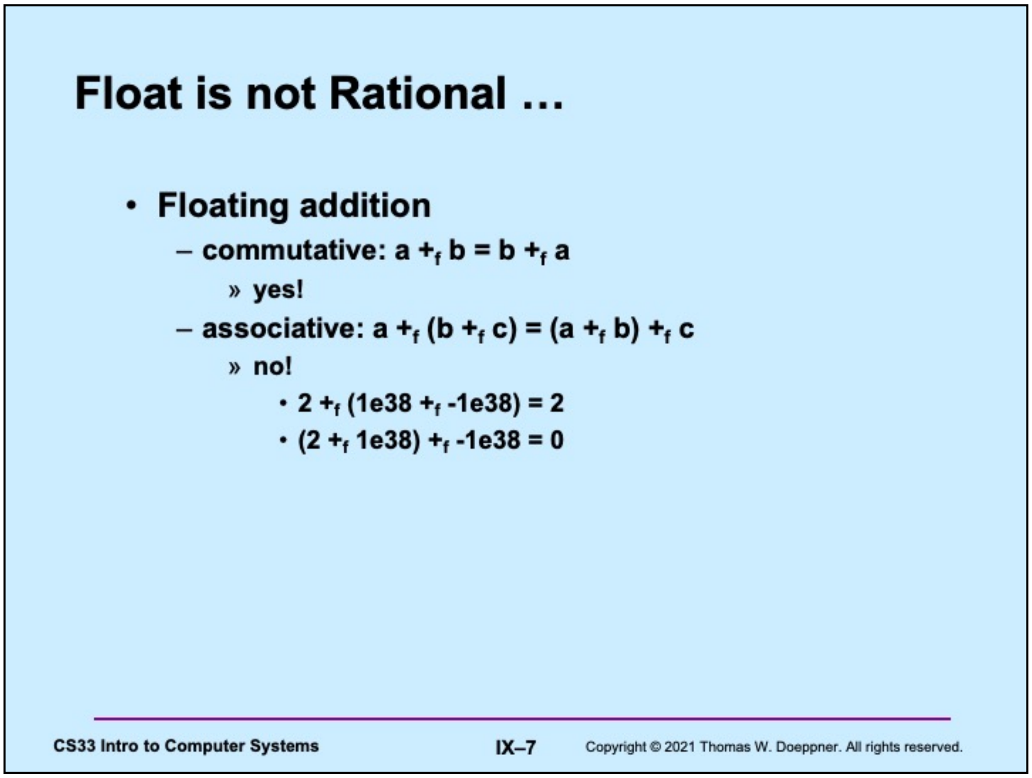
- **Conversions/casting**
  - casting between `int`, `float`, and `double` changes bit representation
  - `double`/`float` → `int`
    - » truncates fractional part
    - » like rounding toward zero
    - » not defined when out of range or NaN: generally sets to TMin
  - `int` → `double`
    - » exact conversion, as long as `int` has ≤ 53-bit word size
  - `int` → `float`
    - » will round according to rounding mode

Supplied by CMU.

Quiz 1

Suppose *f*, declared to be a `float`, is assigned the largest possible floating-point positive value (other than +∞). What is the value of *g = f+1.0*?

a) 0
b) f
c) +∞
d) NaN



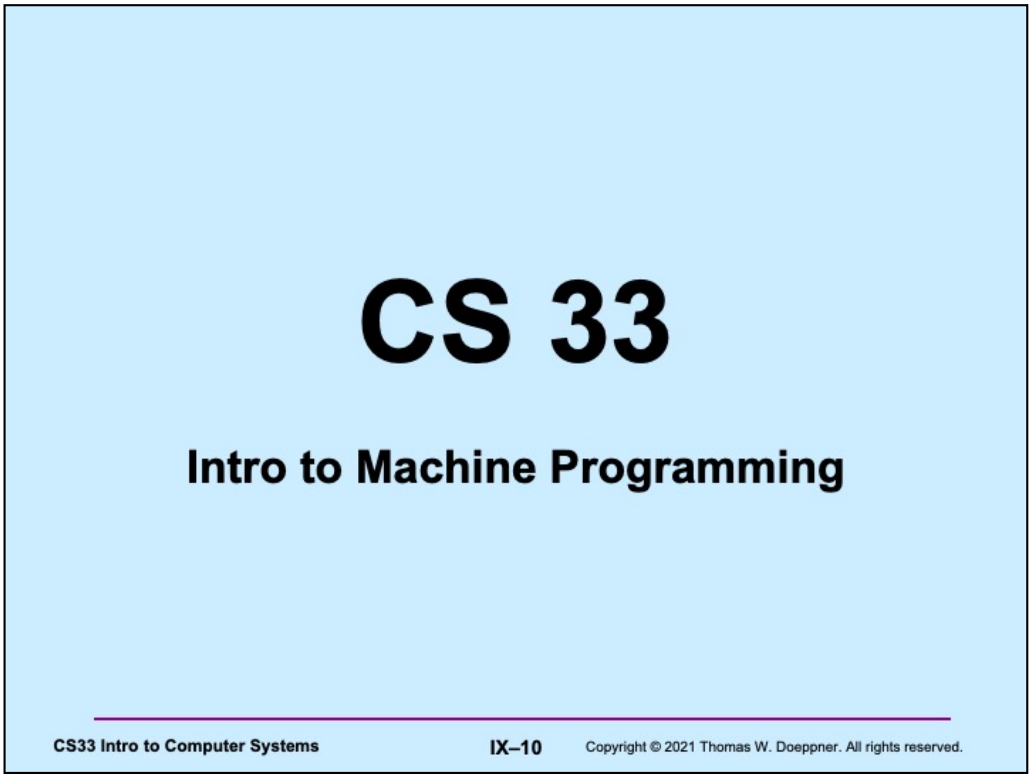
Note that the floating-point numbers in this and the next two slides are expressed in base 10, not base 2.

Float is not Rational …

- **Multiplication**
  - commutative: $a *_f b = b *_f a$
    - » yes!
  - associative: $a *_f (b *_f c) = (a *_f b) *_f c$
    - » no!
      - $1e37 *_f (1e37 *_f 1e-37) = 1e37$
      - $(1e37 *_f 1e37) *_f 1e-37 = +\infty$

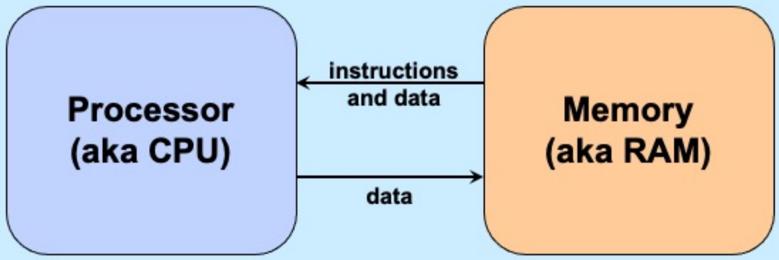
Float is not Rational …

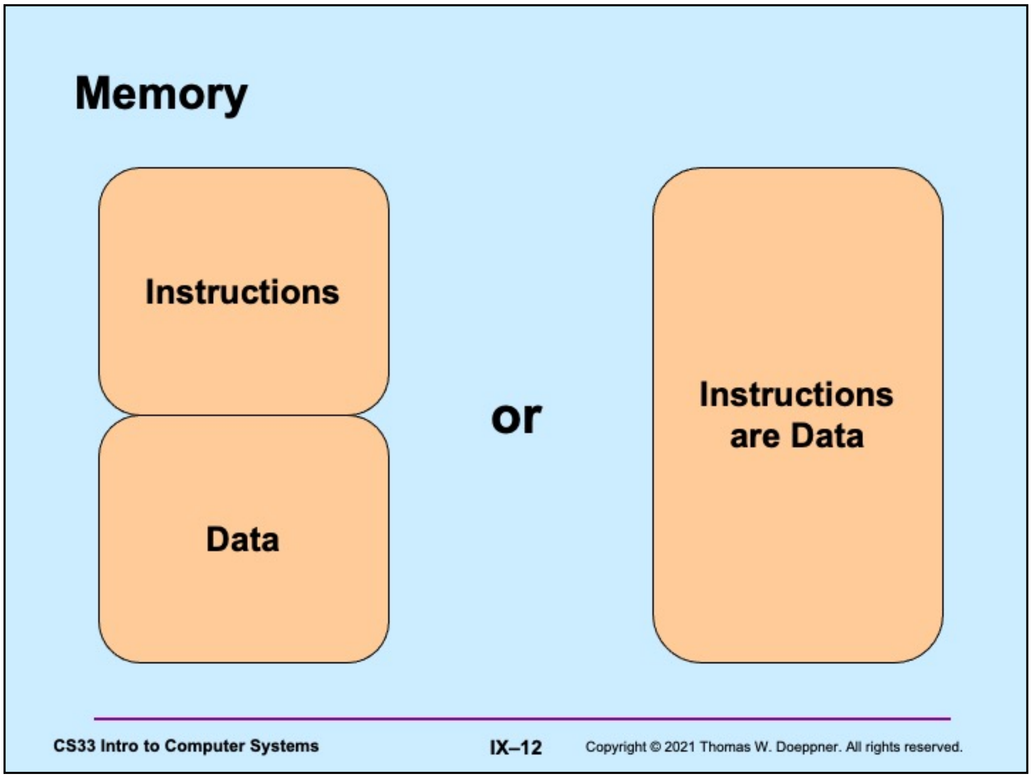
- More …
  - multiplication distributes over addition:
    $a *_f (b +_f c) = (a *_f b) +_f (a *_f c)$
    - » no!
    - » $1e38 *_f (1e38 +_f -1e38) = 0$
    - » $(1e38 *_f 1e38) +_f (1e38 *_f -1e38) = NaN$
  - insignificance:
    $x = y +_f 1$
    $z = 2 /_f (x -_f y)$
    $z == 2?$
    - » not necessarily!
      - consider y = 1e38



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

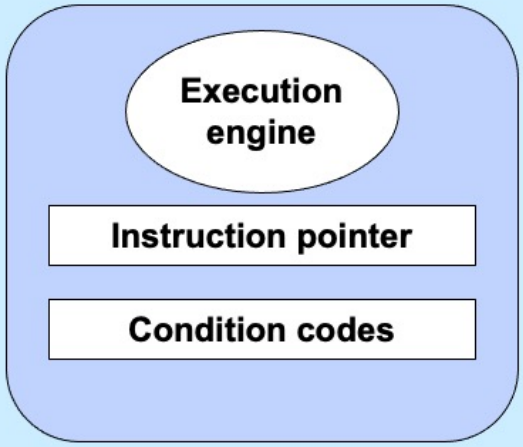




Generally, we think of their being two sorts of memory: that containing instructions and that containing data. Programs, in general, don't modify their own instructions on the fly. In reality, there's only one sort of memory, which holds everything. However, we arrange so that memory holding instructions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

Op code	Operand1	Operand2	...

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        immediate operand
    jmp   .L2
.L1
    mov   $1,d        immediate operand
.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)

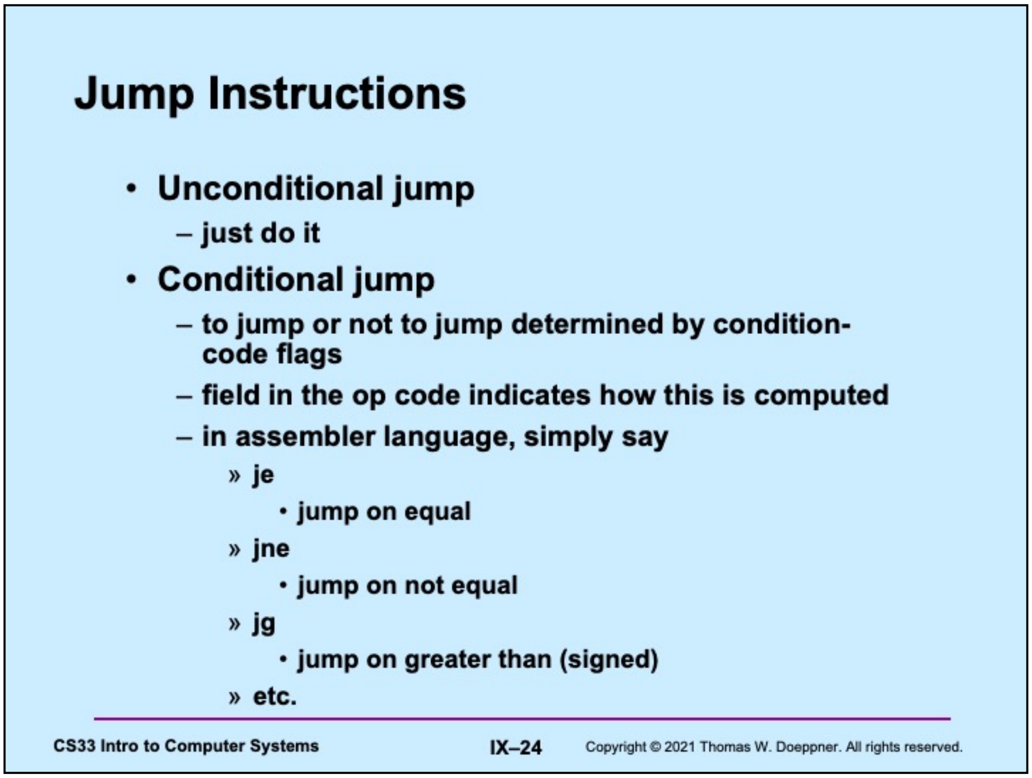
- x is 0xffffffff
  - -1 if interpreted as two's-complement
  - UMAX (2^{32}-1) if interpreted as unsigned
- x+2 (0x00000001)
  - (+)1 under either interpretation
  - zero flag is not set
  - sign flag is not set
  - overflow flag is not set
  - carry flag is set

Quiz 2

- **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

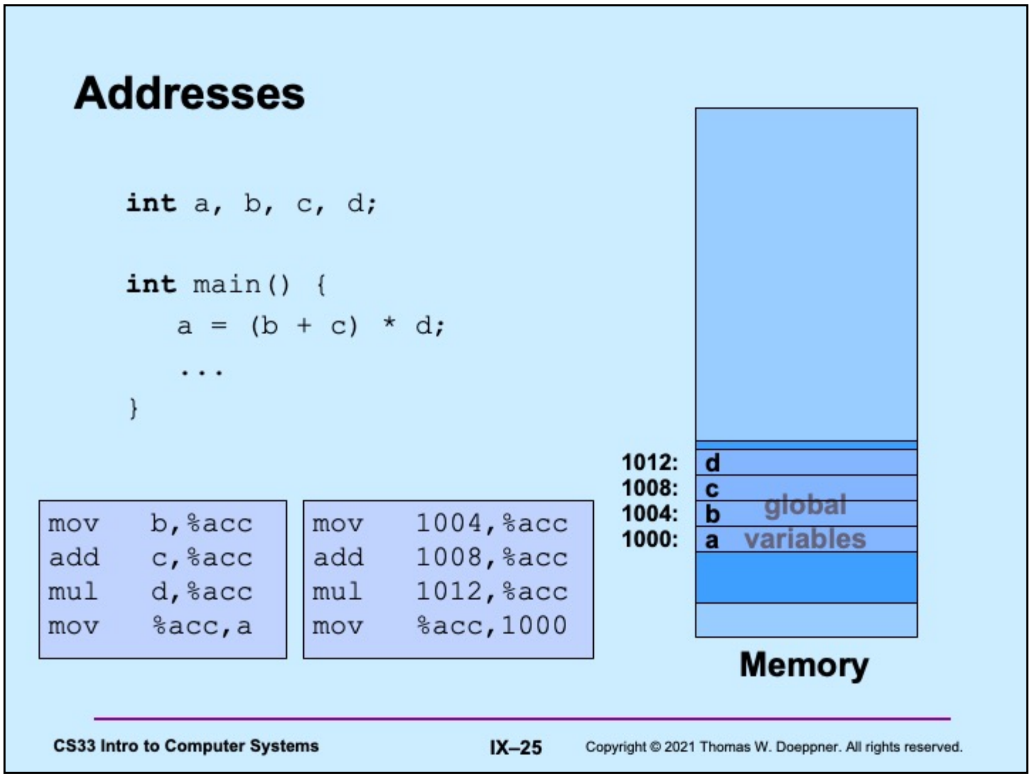
Which flags are set to one by "cmp 2,1"?

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 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!

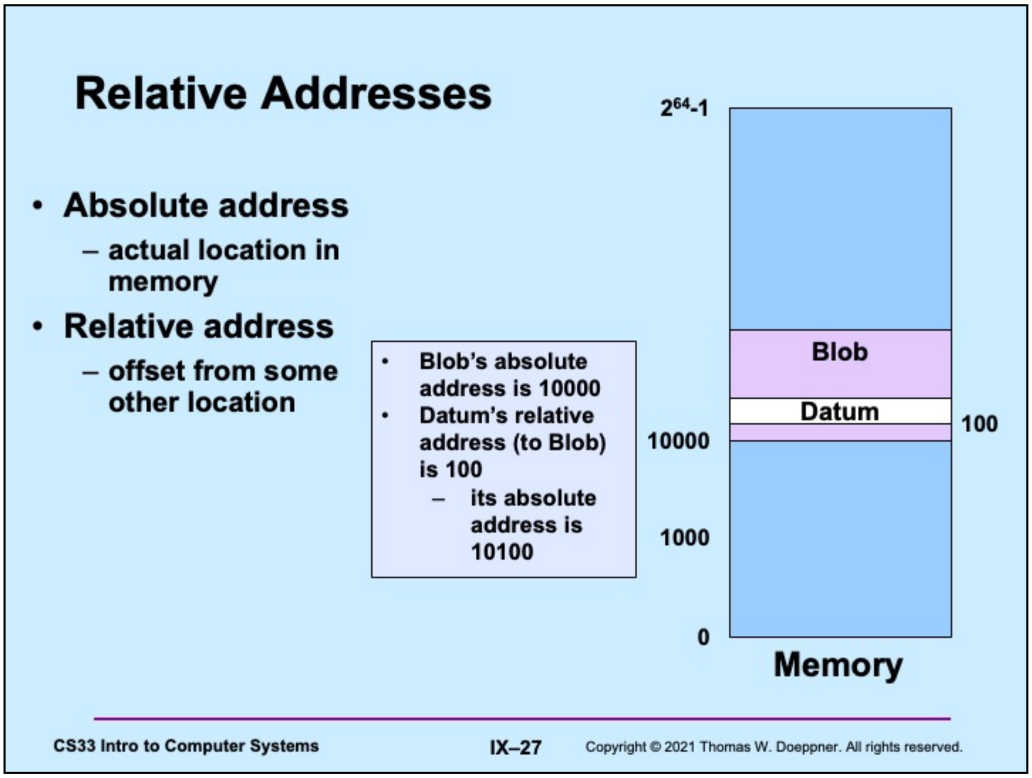
Beginning with this slide, whenever we draw pictures of memory, lower memory addresses are at the bottom, higher addresses are at the top. This is the opposite of how we've been drawing pictures of memory in previous slides.

Addresses

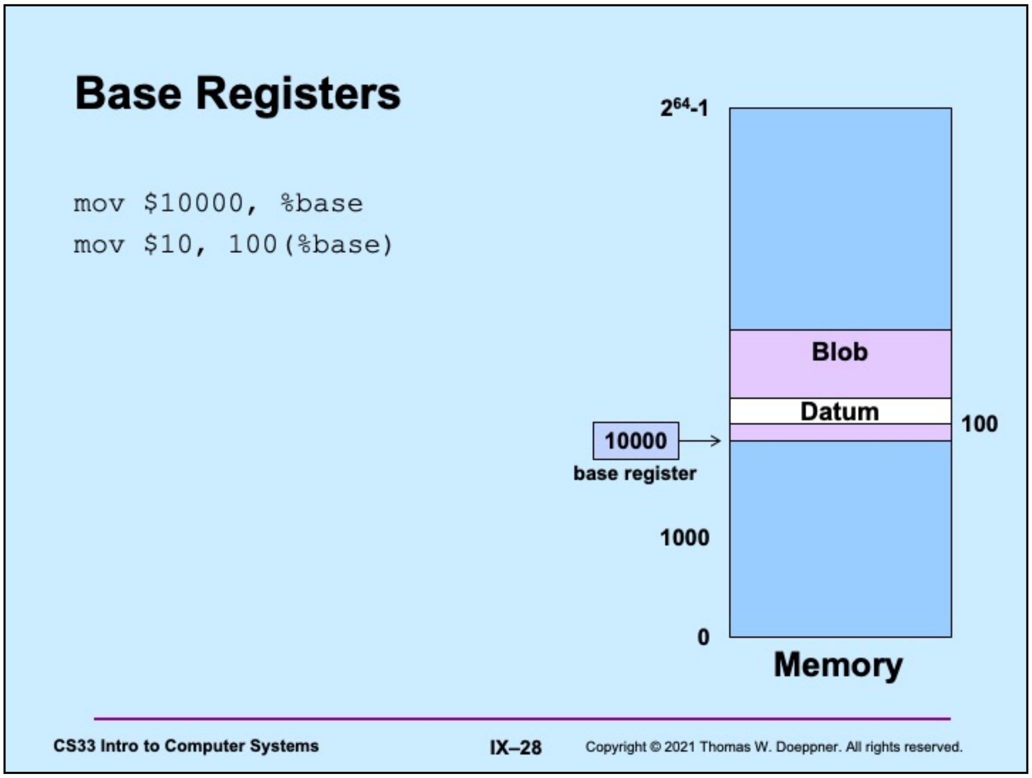
```
int b;

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

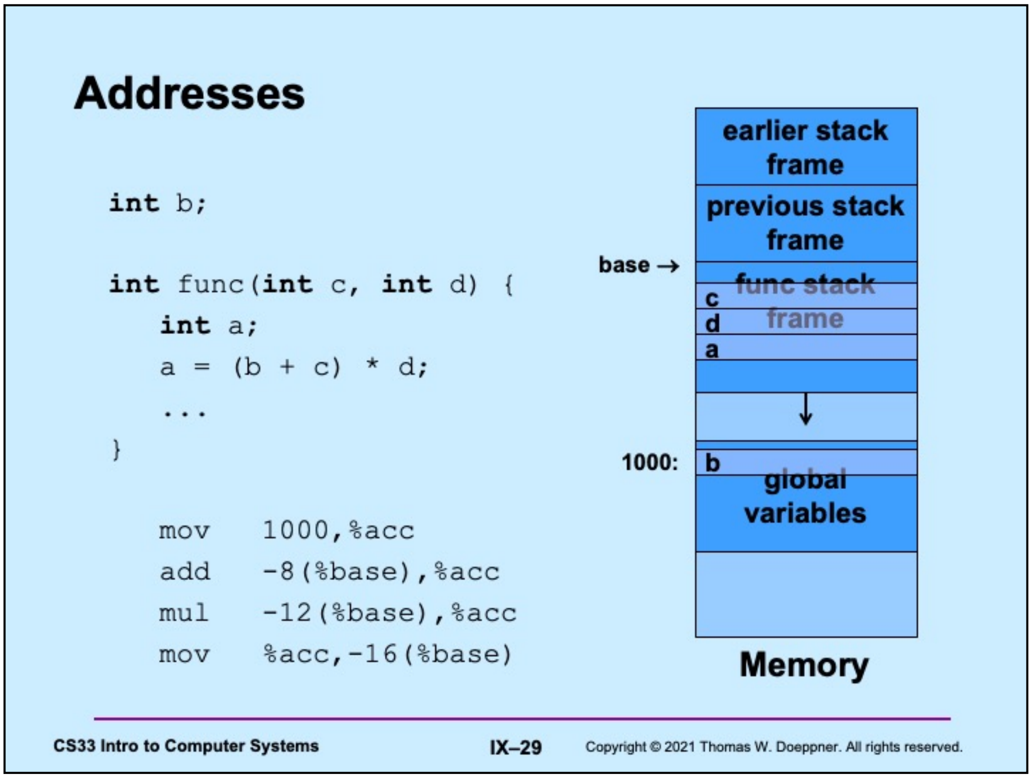
Here we rearrange things a bit. **b** is a global variable, but a is a local variable within **func**, and **c** and **d** are arguments. The issue here is that the locations associated with **a**, **c**, and **d** will, in general, be different for each call to **func**. Thus, we somehow must modify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



Note that both positive and negative offsets might be used.



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.



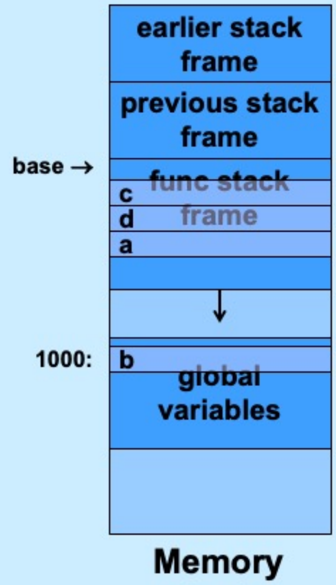
Here we return to our earlier example. We assume that, as part of the call to **func**, the base register is loaded with the address of the beginning of **func**'s current stack frame, and that the local variable **a** and the parameters **c** and **d** are located within the frame. Thus, we refer to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

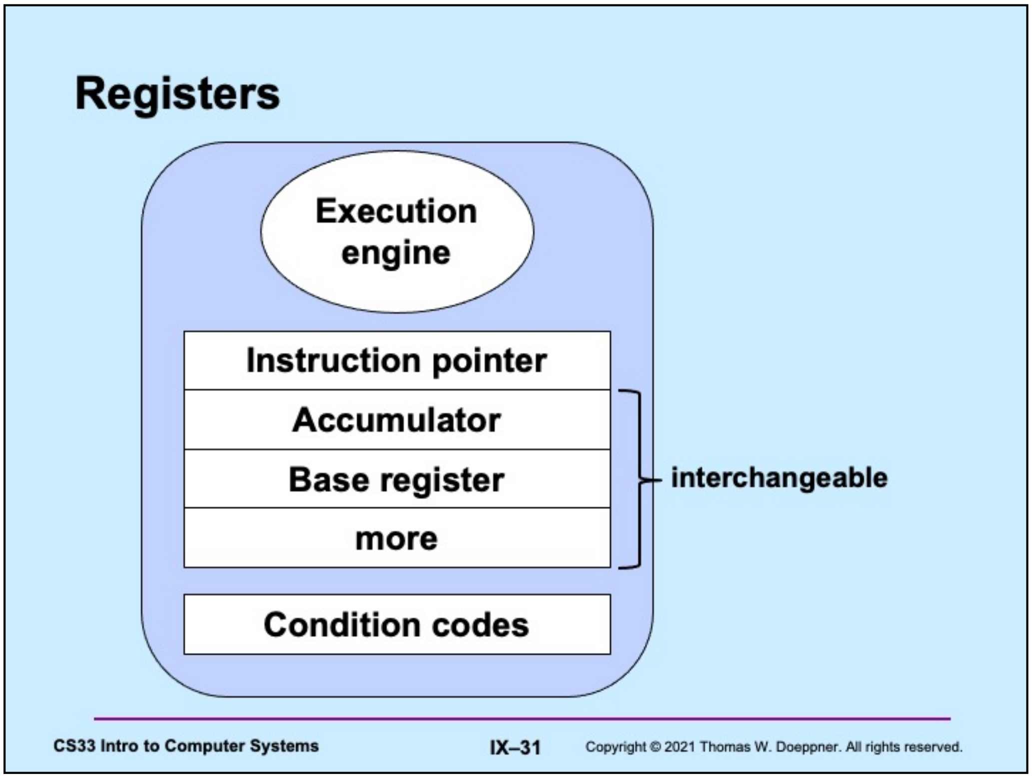
Quiz 3

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

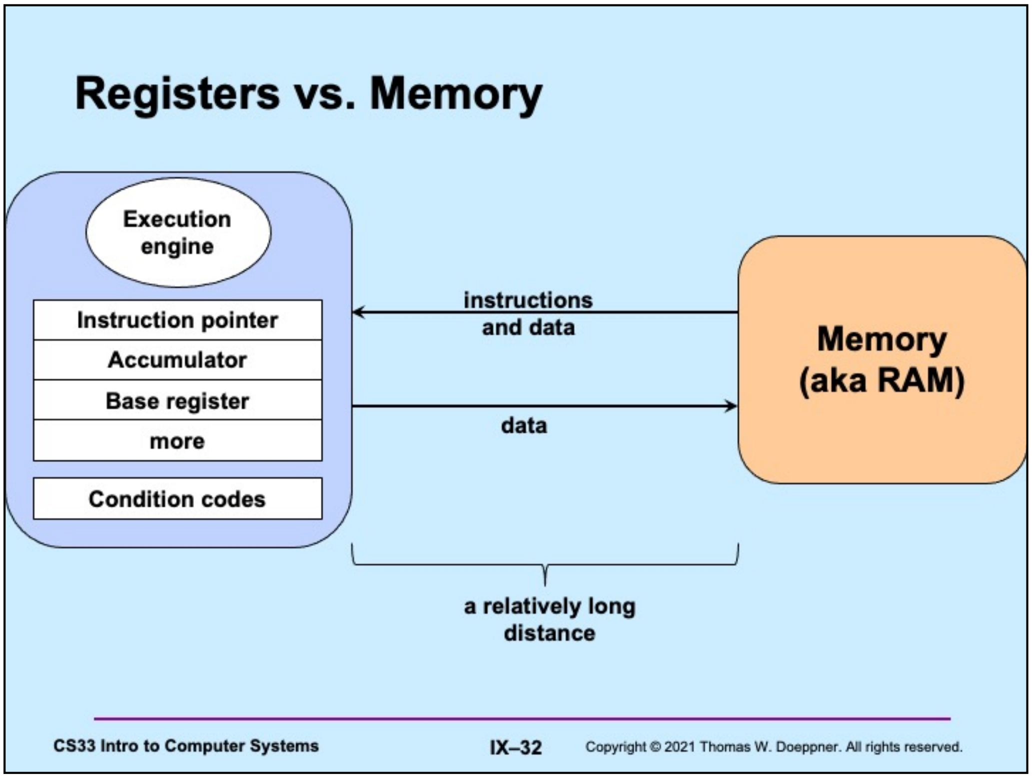
a) 10,004
b) 10,008
c) 9992
d) 9996

```
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.