

CS 33

Data Representation (Part 3)

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.

Byte Ordering

- **Four-byte integer**
 - 0x76543210
- **Stored at location 0x100**
 - which byte is at 0x100?
 - which byte is at 0x103?

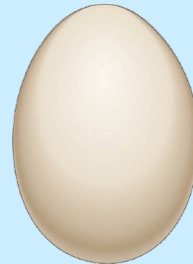


10	32	54	76
0x100	0x101	0x102	0x103

Little-endian

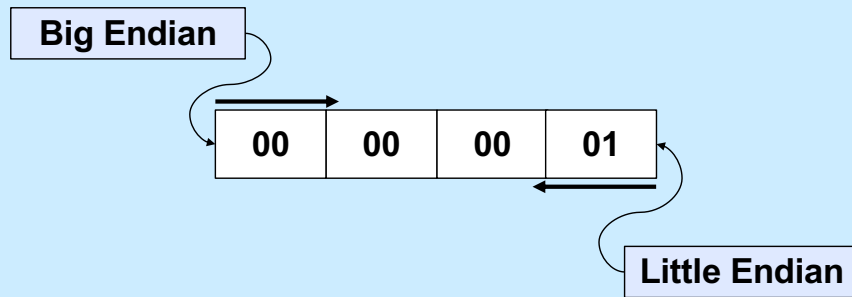
76	54	32	10
0x100	0x101	0x102	0x103

Big-endian



Read “Gulliver’s Travels” by Jonathan Swift for an explanation of the egg.

Byte Ordering (2)



Here we have a four-byte integer one. In the big-endian representation, the address of the integer is the address of the byte containing its most-significant bits (the big end), while in the little-endian representation, the address of the integer is the address of the byte containing its least-significant bits (the little end). Suppose we pass a pointer to this integer to some function. However, in a type-mismatch, the function assumes that what is passed it is a two-byte integer. On a big-endian system, it would think it was passed a zero, but on a little-endian system, it would think it was passed a one.

This is not an argument in favor of either approach, but simply an observation that behaviors could be different.

Quiz 1

```
int main() {  
    long x=1;  
    func((int *)&x);  
    return 0;  
}  
  
void func(int *arg) {  
    printf("%d\n", *arg);  
}
```

**What value is printed
on a big-endian 64-bit
computer?**

- a) 1
- b) 0
- c) 2^{32}
- d) $2^{32}-1$

Which Byte Ordering Do We Use?

```
int main() {
    unsigned int x = 0x03020100;
    unsigned char *xarray = (unsigned char *)&x;
    for (int i=0; i<4; i++) {
        printf("%02x", xarray[i]);
    }
    printf("\n");
    return 0;
}
```

Possible results:

```
00010203
03020100
```

This code prints out the value of `x`, one byte at a time, starting with the byte at the lowest address (little end). On x86-based and m1-based (and presumably m2-based) computers, it will print:

```
00010203
```

which means that the address of an `int` is the address of the byte containing its least significant digits (little endian).

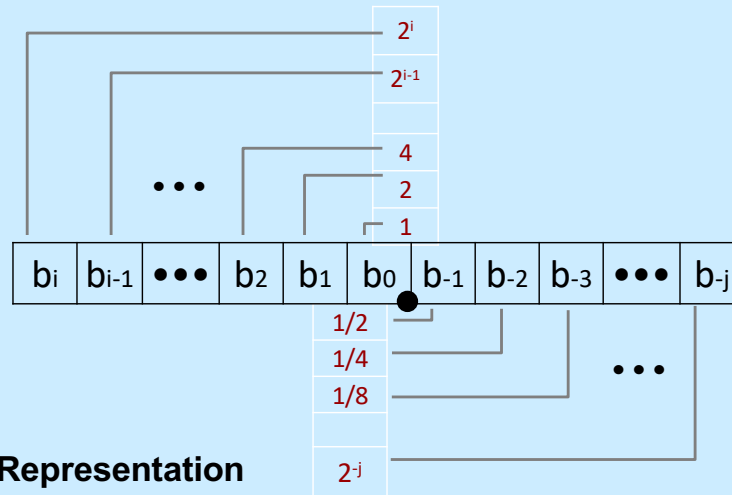
How does **printf** know that `xarray[i]` is an **unsigned char** (and thus one byte long) rather than an **int**? It turns out that **printf** is actually a macro (created using **#define**) that creates additional arguments that give the size (using **sizeof**) of its second and subsequent arguments. Thus, in this example, **printf** calls another function, passing it “%02x”, `xarray[i]`, and **sizeof(xarray[i])**. The “%02x” format code says to convert the argument to hexadecimal notation, print it in a field that’s two characters wide, and include leading 0s.

Fractional binary numbers

- What is 1011.101_2 ?

Supplied by CMU.

Fractional Binary Numbers



- **Representation**

- bits to right of “binary point” represent fractional powers of 2
- represents rational number:

$$\sum_{k=-j}^i b_k \times 2^k$$

Supplied by CMU.

Representable Numbers

- **Limitation #1**

- can exactly represent only numbers of the form $n/2^k$
 - » other rational numbers have repeating bit representations
- value representation
 - » 1/3 0.0101010101[01]₂
 - » 1/5 0.001100110011[0011]₂
 - » 1/10 0.0001100110011[0011]₂

- **Limitation #2**

- just one setting of decimal point within the w bits
 - » limited range of numbers (very small values? very large?)

Supplied by CMU.

IEEE Floating Point

- **IEEE Standard 754**
 - established in 1985 as uniform standard for floating point arithmetic
 - » before that, many idiosyncratic formats
 - supported on all major CPUs
- **Driven by numerical concerns**
 - nice standards for rounding, overflow, underflow
 - hard to make fast in hardware
 - » numerical analysts predominated over hardware designers in defining standard

Supplied by CMU.

IEEE is the Institute for Electrical and Electronics Engineers (pronounced "eye triple e").

Floating-Point Representation

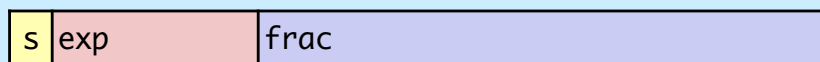
- Numerical Form:

$$(-1)^s M 2^E$$

- sign bit **s** determines whether number is negative or positive
- significand **M** normally a fractional value in range [1.0,2.0)
- exponent **E** weights value by power of two

- Encoding

- MSB **s** is sign bit **s**
- exp field encodes **E** (but is not equal to E)
- frac field encodes **M** (but is not equal to M)



Supplied by CMU.

Precision options

- **Single precision: 32 bits**



- **Double precision: 64 bits**



- **Extended precision: 80 bits (Intel only)**



Supplied by CMU.

On x86 hardware, all floating-point arithmetic is done with 80 bits, then reduced to either 32 or 64 as required.

“Normalized” Values

- When: $\text{exp} \neq 000\dots 0$ and $\text{exp} \neq 111\dots 1$
- Exponent coded as biased value: $E = \text{Exp} - \text{Bias}$
 - exp : unsigned value exp
 - $\text{bias} = 2^{k-1} - 1$, where k is number of exponent bits
 - » single precision: 127 (Exp: 1...254, E: -126...127)
 - » double precision: 1023 (Exp: 1...2046, E: -1022...1023)
- Significand coded with implied leading 1: $M = 1.\text{xxx}\dots\text{x}_2$
 - $\text{xxx}\dots\text{x}$: bits of frac
 - minimum when $\text{frac} = 000\dots 0$ ($M = 1.0$)
 - maximum when $\text{frac} = 111\dots 1$ ($M = 2.0 - \epsilon$)
 - get extra leading bit for “free”

Supplied by CMU.

Normalized Encoding Example

- **Value:** float $F = 15213.0$;

$$\begin{aligned} - 15213_{10} &= 11101101101101_2 \\ &= 1.1101101101101_2 \times 2^{13} \end{aligned}$$

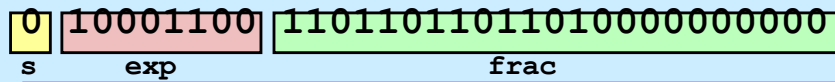
- **Significand**

$$\begin{aligned} M &= 1.\underline{1101101101101}_2 \\ \text{frac} &= \underline{1101101101101}0000000000_2 \end{aligned}$$

- **Exponent**

$$\begin{aligned} E &= 13 \\ \text{bias} &= 127 \\ \text{exp} &= 140 = 10001100_2 \end{aligned}$$

- **Result:**



Supplied by CMU.

Denormalized Values

- **Condition:** $\text{exp} = 000\dots 0$
- **Exponent value:** $E = -\text{Bias} + 1$ (instead of $E = 0 - \text{Bias}$)
- **Significand coded with implied leading 0:**
 $M = 0.\text{xxx}\dots\text{x}_2$
 - $\text{xxx}\dots\text{x}$: bits of frac
- **Cases**
 - $\text{exp} = 000\dots 0, \text{frac} = 000\dots 0$
 - » represents zero value
 - » note distinct values: $+0$ and -0 (why?)
 - $\text{exp} = 000\dots 0, \text{frac} \neq 000\dots 0$
 - » numbers closest to 0.0
 - » equispaced

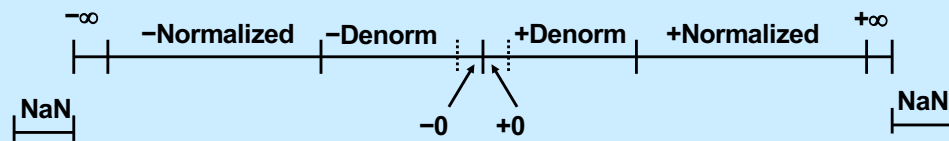
Supplied by CMU.

Special Values

- **Condition: $\text{exp} = 111\dots 1$**
- **Case: $\text{exp} = 111\dots 1, \text{frac} = 000\dots 0$**
 - represents value ∞ (infinity)
 - operation that overflows
 - both positive and negative
 - e.g., $1.0/0.0 = -1.0/-0.0 = +\infty$, $1.0/-0.0 = -\infty$
- **Case: $\text{exp} = 111\dots 1, \text{frac} \neq 000\dots 0$**
 - not-a-number (NaN)
 - represents case when no numeric value can be determined
 - e.g., $\text{sqrt}(-1)$, $\infty - \infty$, $\infty \times 0$

Supplied by CMU.

Visualization: Floating-Point Encodings



Supplied by CMU.

Tiny Floating-Point Example



- **8-bit Floating Point Representation**
 - the sign bit is in the most significant bit
 - the next four bits are the exponent, with a bias of 7
 - the last three bits are the *frac*
- **Same general form as IEEE Format**
 - normalized, denormalized
 - representation of 0, NaN, infinity

Supplied by CMU.

Dynamic Range (Positive Only)

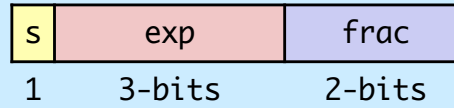
	s	exp	frac	E	Value	
Denormalized numbers	0	0000	000	-6	0	
	0	0000	001	-6	$1/8 * 1/64 = 1/512$	closest to zero
	0	0000	010	-6	$2/8 * 1/64 = 2/512$	
	...					
	0	0000	110	-6	$6/8 * 1/64 = 6/512$	
	0	0000	111	-6	$7/8 * 1/64 = 7/512$	largest denorm
Normalized numbers	0	0001	000	-6	$8/8 * 1/64 = 8/512$	smallest norm
	0	0001	001	-6	$9/8 * 1/64 = 9/512$	
	...					
	0	0110	110	-1	$14/8 * 1/2 = 14/16$	
	0	0110	111	-1	$15/8 * 1/2 = 15/16$	closest to 1 below
	0	0111	000	0	$8/8 * 1 = 1$	
	0	0111	001	0	$9/8 * 1 = 9/8$	closest to 1 above
	0	0111	010	0	$10/8 * 1 = 10/8$	
	...					
	0	1110	110	7	$14/8 * 128 = 224$	
	0	1110	111	7	$15/8 * 128 = 240$	largest norm
	0	1111	000	n/a	inf	

Supplied by CMU.

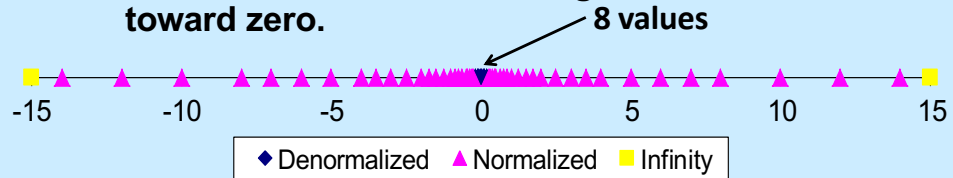
Distribution of Values

- 6-bit IEEE-like format

- e = 3 exponent bits
- f = 2 fraction bits
- bias is $2^{3-1}-1 = 3$



- Notice how the distribution gets denser toward zero.

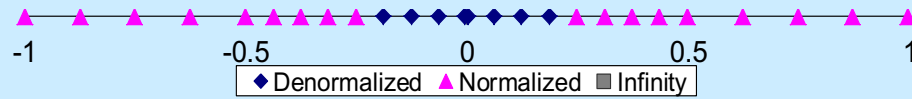


Supplied by CMU.

Distribution of Values (close-up view)

- **6-bit IEEE-like format**

- e = 3 exponent bits
- f = 2 fraction bits
- bias is 3

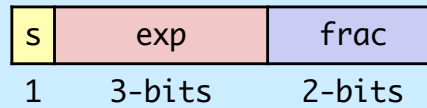


Supplied by CMU.

Quiz 2

- 6-bit IEEE-like format

- e = 3 exponent bits
- f = 2 fraction bits
- bias is 3

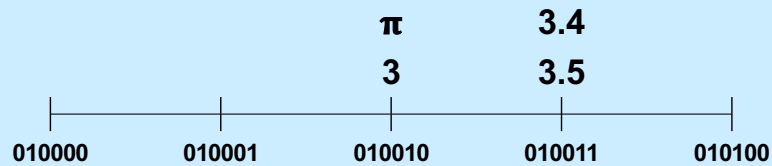


What number is represented by 0 010 10?

- a) 3
- b) 1.5
- c) .75
- d) none of the above

Mapping Real Numbers to Float

- The real number 3 is represented as
0 100 10
- The real number 3.5 is represented as
0 100 11
- How is the real number 3.4 represented?
0 100 11
- How is the real number π represented?
0 100 10



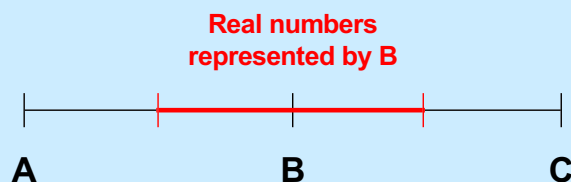
We're assuming here the six-bit floating-point format.

Mapping Real Numbers to Float

- If R is a real number, it's mapped to the floating-point number whose value is closest to R
- What if it's midway between two values?
 - rounding rules determine outcome

Floats are Sets of Values

- If A, B, and C are successive floating-point values
 - e.g., 010001, 010010, and 010011
- B represents all real numbers from midway between A and B through midway between B and C



A special case is 0. Positive 0 represents a range of values that are greater than or equal to 0. Negative 0 represents a range of values that are less than or equal to zero.

Significance

- **Normalized numbers**
 - for a particular exponent value E and an S -bit significand, the range from 2^E up to 2^{E+1} is divided into 2^S equi-spaced floating-point values
 - » thus each floating-point value represents $1/2^S$ of the range of values with that exponent
 - » all bits of the significand are important
 - » we say that there are S significant bits – for reasonably large S , each floating-point value covers a rather small part of the range
 - high accuracy
 - for $S=23$ (32-bit float), accurate to one in 2^{23} (.0000119% accuracy)

Significance

- **Unnormalized numbers**
 - high-order zero bits of the significand aren't important
 - in 8-bit floating point, 0 0000 001 represents 2^{-9}
 - » it is the only value with that exponent: 1 significant bit (either 2^{-9} or 0)
 - 0 0000 010 represents 2^{-8}
0 0000 011 represents $1.5 \cdot 2^{-8}$
 - » only two values with exponent -8: 2 significant bits (encoding those two values, as well as 2^{-9} and 0)
 - fewer significant bits mean less accuracy
 - 0 0000 001 represents a range of values from $.5 \cdot 2^{-9}$ to $1.5 \cdot 2^{-9}$
 - 50% accuracy

Recall that the bias for the exponent of 8-bit IEEE FP is 7, thus for unnormalized numbers the actual exponent is -6 (-bias+1). The significand has an implied leading 0, thus 0 0000 001 represents $2^{-6} \cdot 2^{-3}$.

With 8-bit IEEE FP, the value 0 0000 01 is interpreted as 2^{-9} , But the number represented could be 50% or 50% more.

+/- Zero

- **Only one zero for ints**
 - an int is a single number, not a range of numbers, thus there can be only zero
- **Floating-point zero**
 - a range of numbers around the real 0
 - it really matters which side of 0 we're on!
 - » a very large negative number divided by a very small negative number should be positive
 $-\infty / -0 = +\infty$
 - » a very large positive number divided by a very small negative number should be negative
 $+\infty / -0 = -\infty$

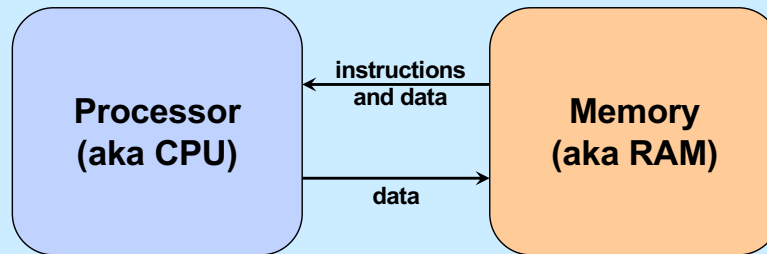
It's important to remember that a floating-point value is not a single number, but a range of numbers.

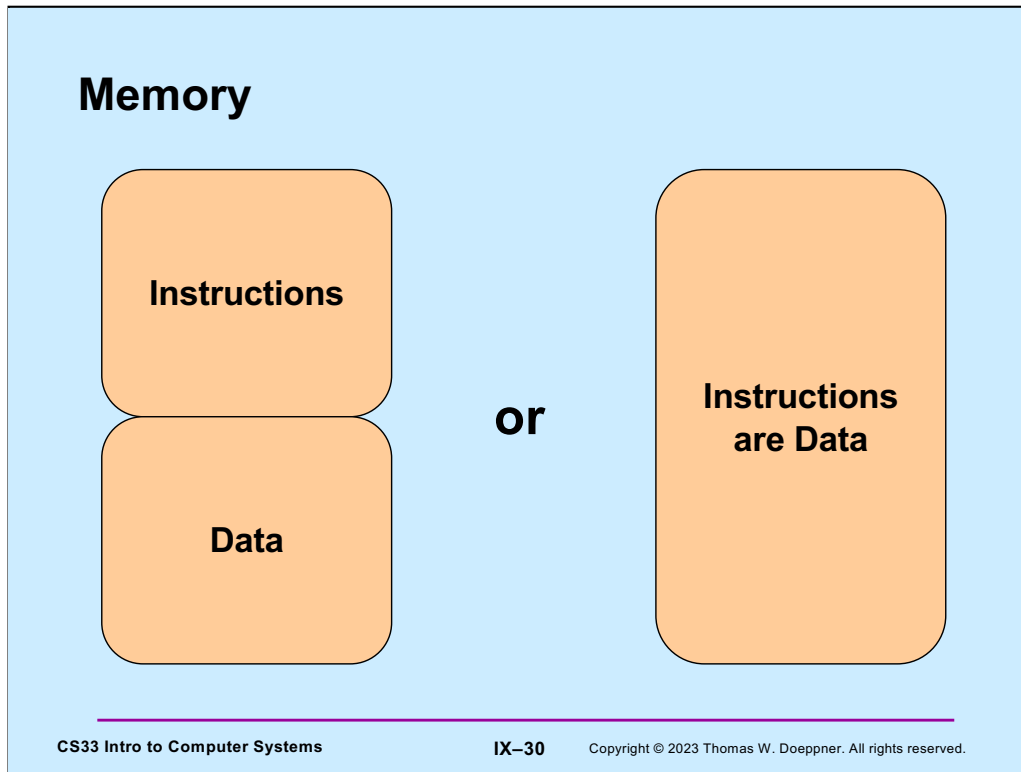
CS 33

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

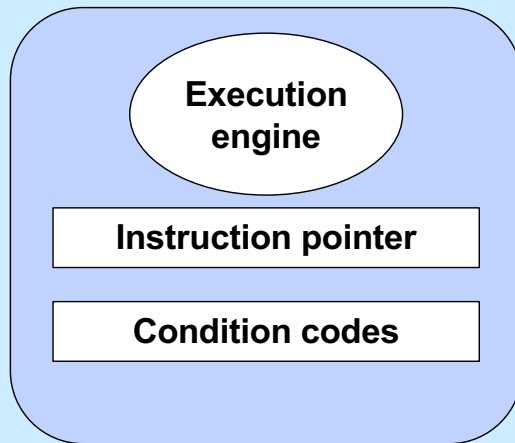




Generally, we think of there being two sorts of memory: that containing instructions and that containing data. Programs, in general, don't modify their own instructions on the fly. In reality, there's only one sort of memory, which holds everything. However, we arrange so that memory holding instructions 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  
jmp    .L2  
.L1  
mov    $1,d  
.L2
```

immediate operand

immediate operand

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 3

- **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.

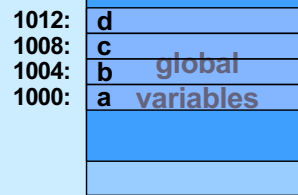
Addresses

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

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

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

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



Memory

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, ?
```

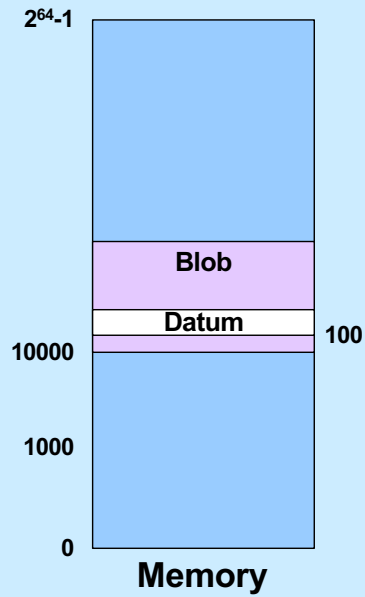
- One copy of *b* for duration of program's execution
 - *b*'s address is the same for each call to *func*
- Different copies of *a*, *c*, and *d* for each call to *func*
 - addresses are different in each call

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.

Relative Addresses

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

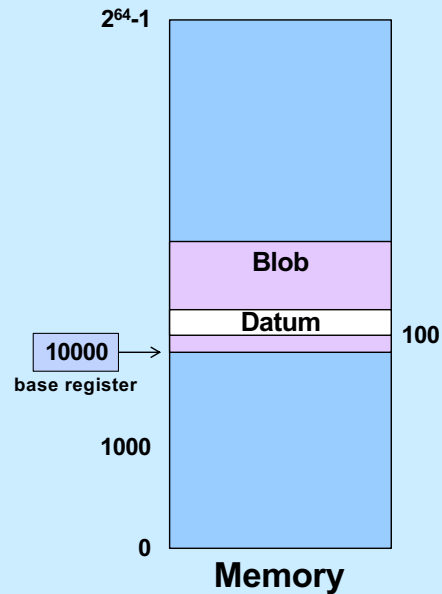
- Blob's absolute address is 10000
- Datum's relative address (to Blob) is 100
 - its absolute address is 10100



Note that both positive and negative offsets might be used.

Base Registers

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

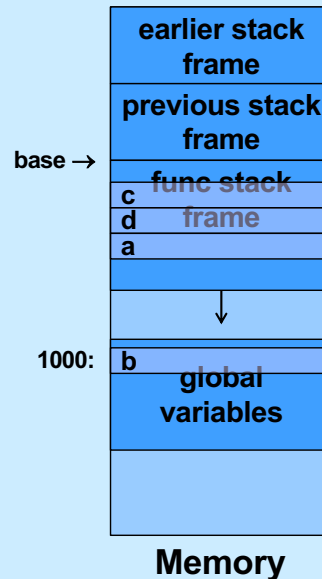


Here we load the value 10,000 into the base register (recall that the “\$” means what follows is a literal value; a “%” sign means that what follows is the name of a register), then store the value 10 into the memory location 10100 (the contents of the base register plus 100): the notation **$n(\%base)$**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

Addresses

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

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



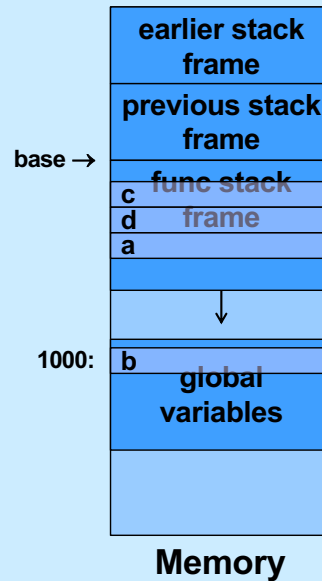
Here we return to our earlier example. We assume that, as part of the call to **func**, the base register is loaded with the address of the beginning of **func**'s current stack frame, and that the local variable **a** and the parameters **c** and **d** are located within the frame. Thus, we refer to them by their offset from the beginning of the stack frame, which are assumed to be **-24**, **-8**, and **-16**. Since the stack grows from higher addresses to lower addresses, these offsets are negative. Note that the first assembler instruction copies the contents of location 1000 into **%acc**.

Quiz 4

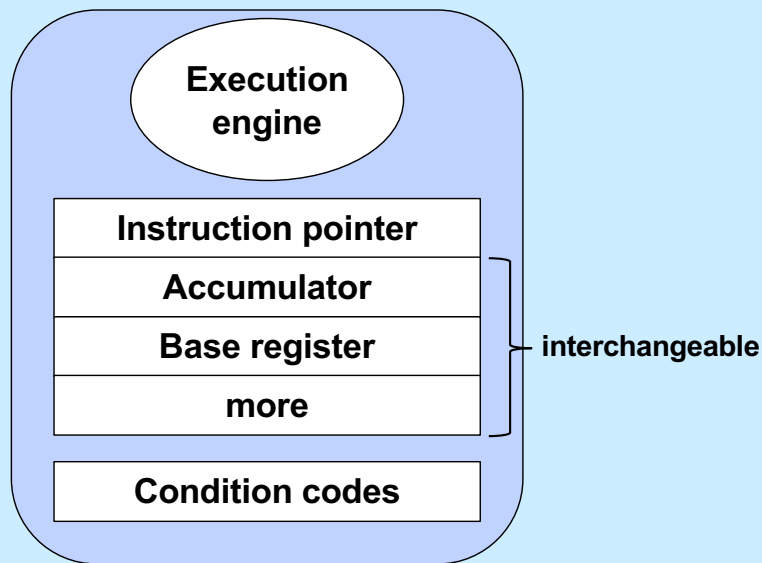
Suppose the value in *base* is 10,000. What is the address of *c*?

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

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

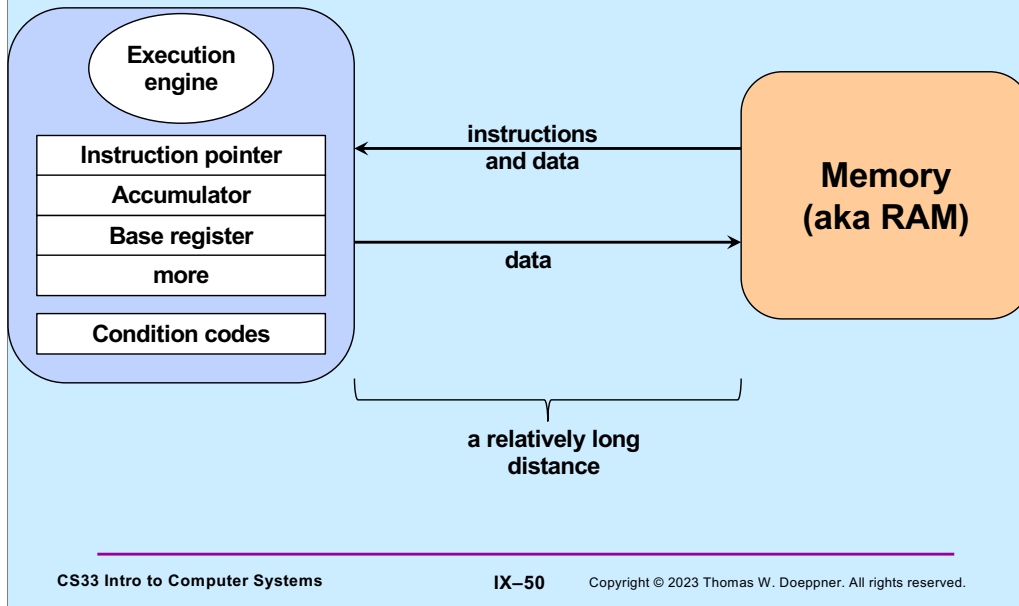


Registers



We've now seen four registers: the instruction pointer, the accumulator, the base register, and the condition codes. The accumulator is used to hold intermediate results for arithmetic; the base register is used to hold addresses for relative addressing. There's no particular reason why the accumulator can't be used as the base register and vice versa: thus, they may be used interchangeably. Furthermore, it is useful to have more than two such dual-purpose registers. As we will see, the x86 architecture has eight such registers; the x86-64 architecture has 16.

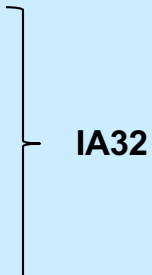
Registers vs. Memory



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).