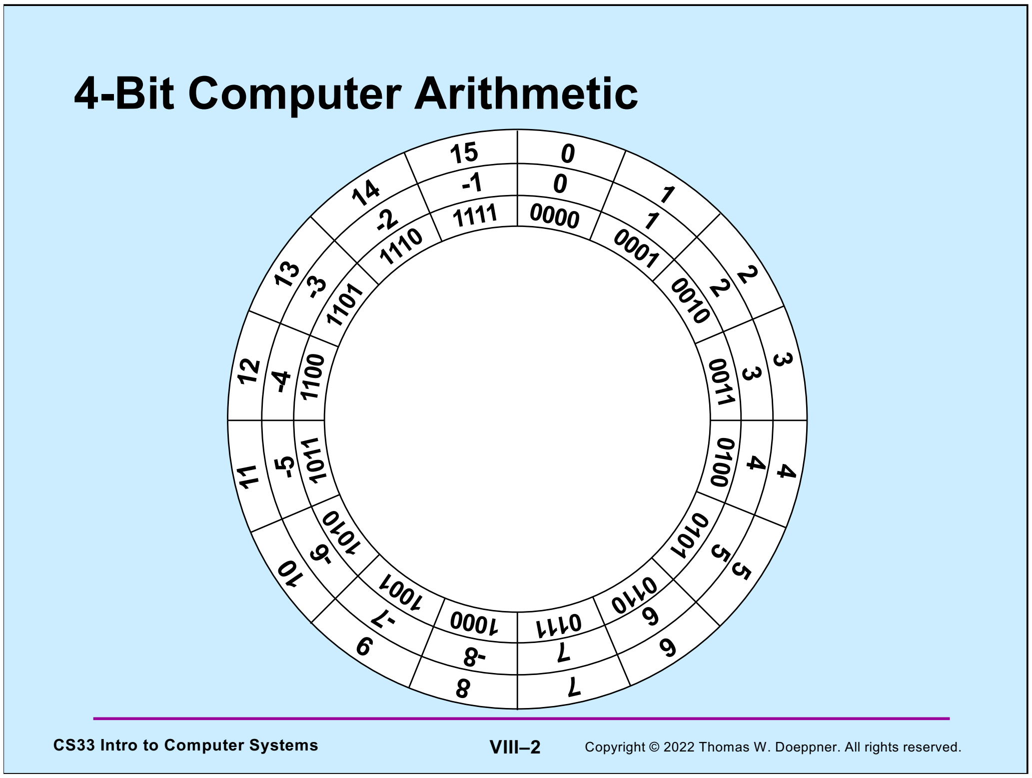
CS 33

Data Representation (Part 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.



Unsigned computer arithmetic is performed modulo 2 to the power of the computer's word size. The outer ring of the figure demonstrates arithmetic modulo 2^4. To see the result, for example, of adding 3 to 2, start at 2 and go around the ring three units in the clockwise direction. If we add 5 to 14, we start at 14 and move 5 units clockwise, to 3. Similarly, to subtract 3 from 1, we start at one and move three units counterclockwise to 14.

What about two's-complement computer arithmetic? We know that the values encoded in a 4-bit computer word range from -8 to 7. How do we arrange them in the ring? As shown in the second ring, it makes sense for the non-negative numbers to be in the same positions as the corresponding unsigned values. It clearly makes sense for the integer coming just before 0 to be -1, the integer just before -1 to be -2, etc. Thus, since we have a ring, the integer following 7 is -8. Now we can see how arithmetic works for two's-complement numbers. Adding 3 to 2 works just as it does for unsigned numbers. Subtracting 3 from 1 results in -2. But adding 3 to 6 results in -7; and adding 5 to -2 results in 3.

The innermost ring shows the bit encodings for the unsigned and two's-complement values. The point of all this is that, with only one implementation of arithmetic, we can handle both unsigned and two's-complement values. Thus, adding unsigned 5 and 9 is equivalent to adding two's-complement 5 and -7. The result will 1110, which, if interpreted as an unsigned value is 14, but if interpreted as a two's-complement value is -2.

Signed vs. Unsigned in C

- **Constants**
  - by default are considered to be signed integers
  - unsigned if have "U" as suffix

    `0U, 4294967259U`
- **Casting**
  - explicit casting between signed & unsigned

```
int tx, ty;
unsigned ux, uy; // "unsigned" means "unsigned int"
tx = (int) ux;
uy = (unsigned int) ty;
```

  - implicit casting also occurs via assignments and function calls

```
tx = ux;
uy = ty;
```


Supplied by CMU.

Note that the kind of casting done here is what we called "intimidation" in the previous lecture: no actual conversion takes place, but the value is reinterpreted according to the cast.

Casting Surprises

- Expression evaluation
  - if there is a mix of unsigned and signed in single expression, *signed values implicitly cast to unsigned*
  - including comparison operations <, >, ==, <=, >=
  - examples for $W = 32$: **TMIN = -2,147,483,648 , TMAX = 2,147,483,647**

$Constant_1$	$Constant_2$	Relation	Evaluation
0	0U	==	unsigned
-1	0	<	signed
-1	0U	>	unsigned
2147483647	-2147483647-1	>	signed
2147483647U	-2147483647-1	<	unsigned
-1	-2	>	signed
(unsigned)-1	-2	>	unsigned
2147483647	2147483648U	<	unsigned
2147483647	(int)2147483648U	>	signed

Supplied by CMU.

Quiz 1

What is the value of

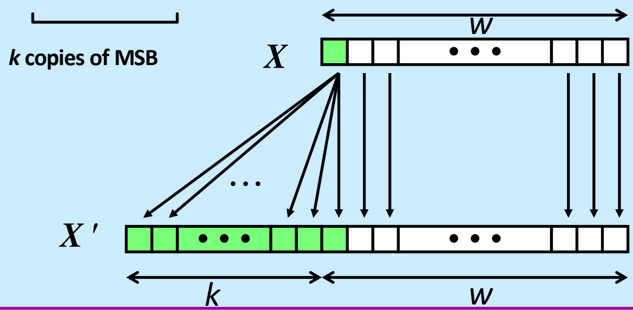
```
(unsigned long)-1 - (long)ULONG_MAX
```

???

a) 0
b) -1
c) 1
d) ULONG_MAX

Sign Extension

- **Task:**
  - **given *w*-bit signed integer *x***
  - **convert it to *w*+*k*-bit integer with same value**
- **Rule:**
  - **make *k* copies of sign bit:**
  - $X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0$



Supplied by CMU.

Sign Extension Example

```
short int x =  15213;
int      ix = (int) x;
short int y = -15213;
int      iy = (int) y;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

- Converting from smaller to larger integer data type
  - C automatically performs sign extension

Supplied by CMU.

Does it Work?

$$val_w = -2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i$$

$$val_{w+1} = -2^w + 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i$$

$$= -2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i$$

$$val_{w+2} = -2^{w+1} + 2^w + 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i$$

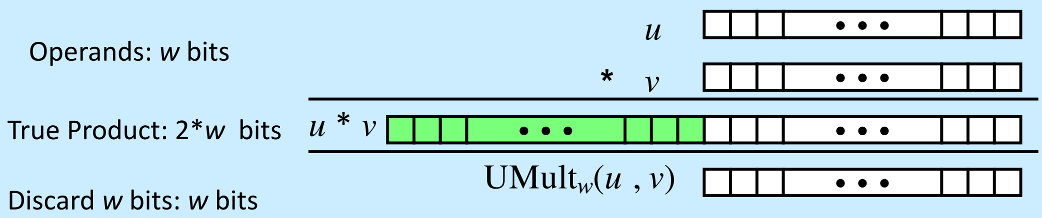
$$= -2^w + 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i$$

$$= -2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i$$

Sign extension clearly works for positive and zero values (where the sign bit is zero). But does it work for negative values? The first line of the slide shows the computation of the value of a w-bit item with a sign bit of one (i.e., it's negative). The next two lines show what happens if we extend this to a w+1-bit item, extending the sign bit. What had been the sign bit becomes one of the value bits, and its contribution to the value is now positive rather than negative. But this is compensated by the new sign bit, whose contribution is a negative value, twice as large as the original sign bit. Thus, the net effect is for there to be no change in the value.

We do this again, extending to a w+2-bit item, and again, the resulting value is the same as what we started with.

Unsigned Multiplication

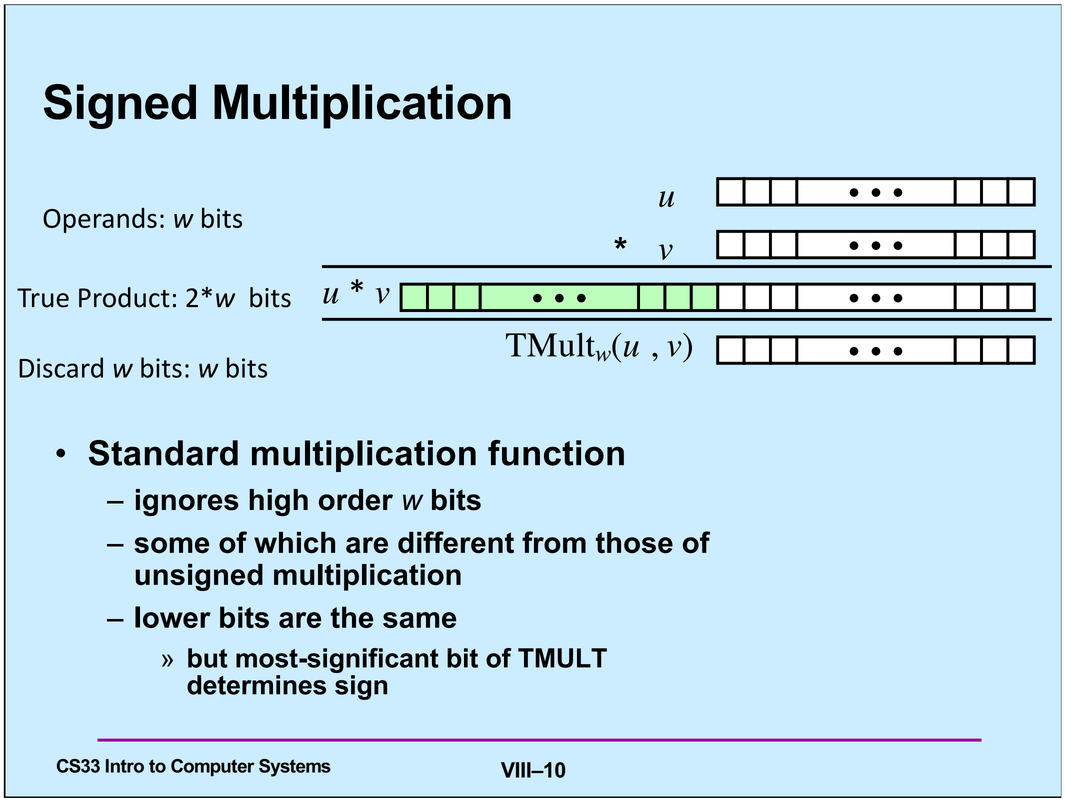


- **Standard multiplication function**
  - **ignores high order *w* bits**
- **Implements modular arithmetic**

  $\mathrm{UMult}_w(u\ ,\ v) \quad = \quad u \cdot v \bmod 2^w$

Supplied by CMU.

Note that to represent the true product of two arbitrary w-bit values, we need 2w bits.



Supplied by CMU.

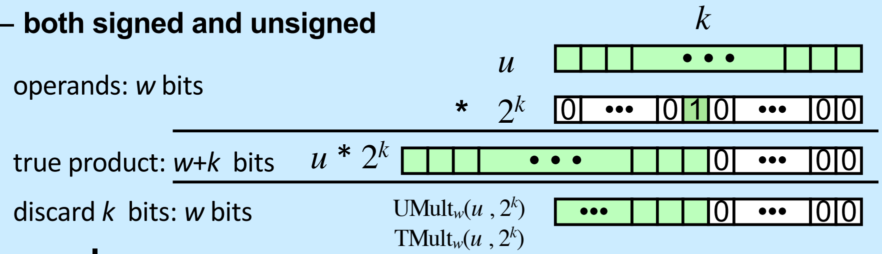
Why is it that the "true product" is different from that of unsigned multiplication? Consider what the true product should be if the multiplier is -1 and the multiplicand is 1. The multiplier is a w-bit word of all ones; the multiplicand is a w-bit word of all zeroes except for the least-significant bit, which is 1. The high-order w bits of the true product should be all ones (since it's negative), but with unsigned multiplication they'd be all zeroes. However, since we're ignoring the high-order w bits, this doesn't matter.

Note that the sign of the result depends on the most-significant bit of the w-bit result, which could have no relation to the signs of the multiplier or the multiplicand.

It may be particularly important to have 64-bit results when multiplying arbitrary 32-bit signed integers.

Power-of-2 Multiply with Shift

- **Operation**
  - `u << k` **gives** `u * ` ***2^k***
  - **both signed and unsigned**



- **Examples**

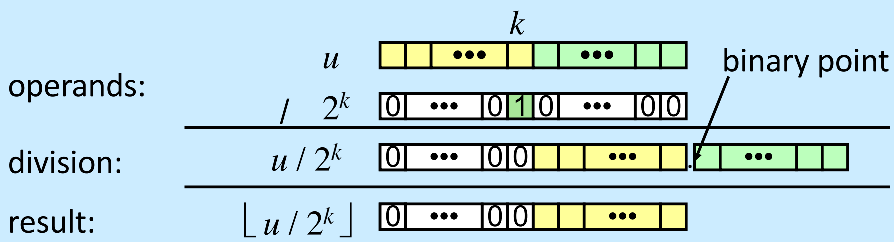
```
u << 3 ==     u * 8
u << 5 - u << 3 == u * 24
```

  - **most machines shift and add faster than multiply**
    - » **compiler generates this code automatically**

Supplied by CMU.

Unsigned Power-of-2 Divide with Shift

- **Quotient of unsigned and power of 2**
  - `u >> k` gives $\lfloor$ `u / ` $2^k \rfloor$
  - **uses logical shift**

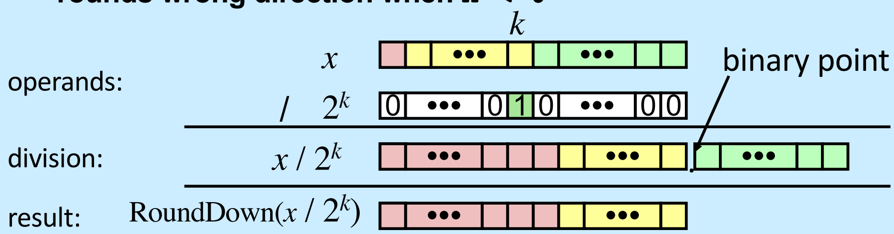


	Division	Computed	Hex	Binary
`x`	15213	15213	`3B 6D`	`00111011 01101101`
`x >> 1`	7606.5	7606	`1D B6`	`00011101 10110110`
`x >> 4`	950.8125	950	`03 B6`	`00000011 10110110`
`x >> 8`	59.4257813	59	`00 3B`	`00000000 00111011`

Supplied by CMU.

Signed Power-of-2 Divide with Shift

- **Quotient of signed and power of 2**
  - **`x >> k` gives $\lfloor$ `x / 2`^{k} $\rfloor$**
  - **uses arithmetic shift**
  - **rounds wrong direction when `x < 0`**



	Division	Computed	Hex	Binary
`y`	-15213	-15213	`C4 93`	`11000100 10010011`
`y >> 1`	-7606.5	-7607	`E2 49`	`11100010 01001001`
`y >> 4`	-950.8125	-951	`FC 49`	`11111100 01001001`
`y >> 8`	-59.4257813	-60	`FF C4`	`11111111 11000100`

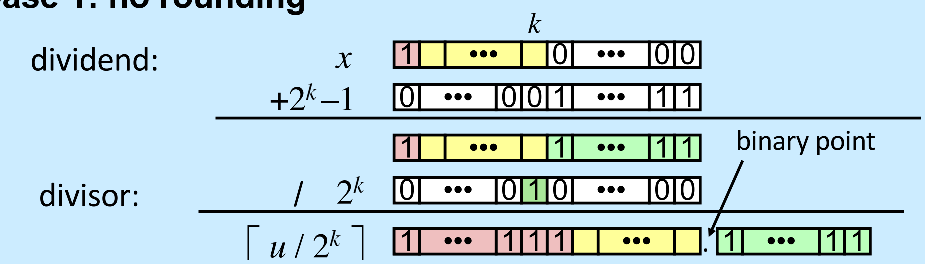
Supplied by CMU.

Recall that with two's-complement, all the bits other than the most-significant represent positive values. Thus, we are shifting off (to the right) bits that should be adding a positive value to the number, but now are lost. Thus, if any of these bits are one, after shifting the resulting value will be less than it should be (i.e., more negative).

Correct Power-of-2 Divide

- **Quotient of negative number by power of 2**
  - want $\lceil$ `x / ` $2^k \rceil$ (round toward 0)
  - compute as $\lfloor$ `(x+`2^k`-1) / ` $2^k \rfloor$
    - » in C: `(x + (1<<k)-1) >> k`
    - » biases dividend toward 0

Case 1: no rounding



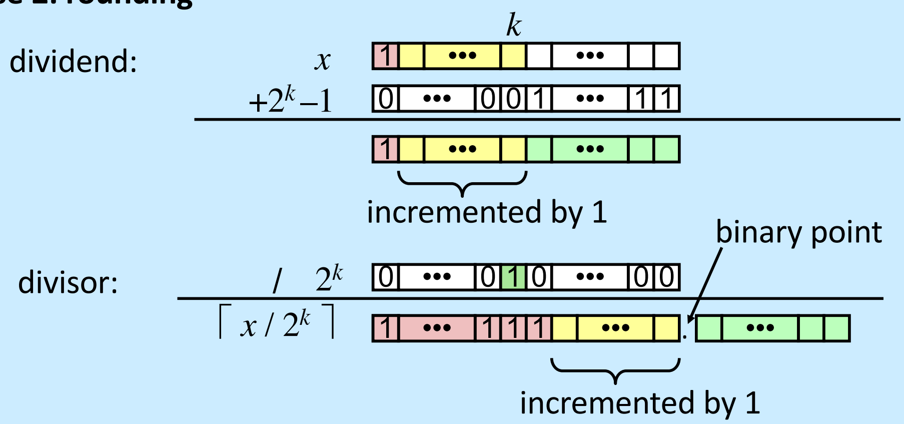
Biasing has no effect

Supplied by CMU.

If the least-significant k bits are all zeroes, then adding in the bias and shifting right by k bits eliminates any effect of adding the bias.

Correct Power-of-2 Divide (Cont.)

Case 2: rounding



Biasing adds 1 to final result

Supplied by CMU.

If any of the least-significant k bits are one, then adding the bias to them causes a carry of one to the bits to their left. Thus, after shifting, the number that's represented by the remaining bits is one greater (less negative) than it would have been if the bias had not been added.

Why Should I Use Unsigned?

- ***Don't*** **use just because number nonnegative**
  - **easy to make mistakes**

```
unsigned i;
for (i = cnt-2; i >= 0; i--)
  a[i] += a[i+1];
```

  - **can be very subtle**

```
#define DELTA sizeof(int)
int i;
for (i = CNT; i-DELTA >= 0; i-= DELTA)
  . . .
```

- ***Do*** **use when using bits to represent sets**
  - **logical right shift, no sign extension**

Supplied by CMU.

Note that “sizeof” returns an unsigned value. (Recall that, when mixing signed and unsigned items in an expression, the result will be unsigned.)

Word Size

- (Mostly) obsolete term
  - old computers had items of one size: the word size
- Now used to express the number of bits necessary to hold an address
  - 16 bits (really old computers)
  - 32 bits (old computers)
  - 64 bits (most current computers)

Byte Ordering

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

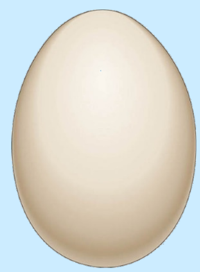
?

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

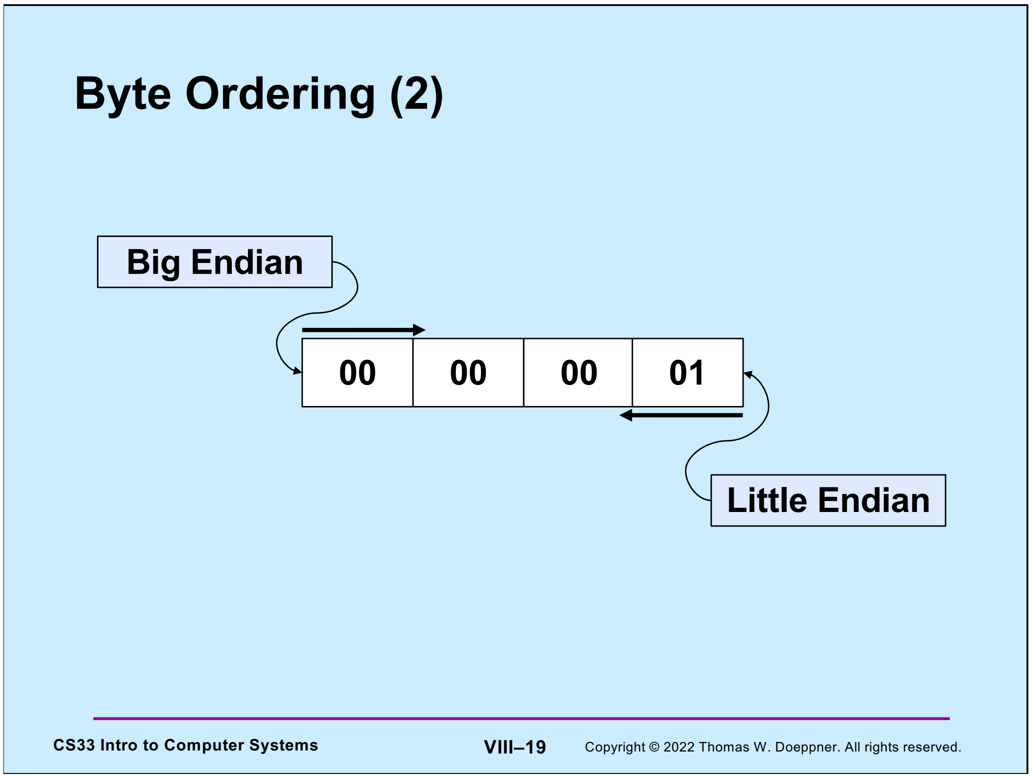
Little-endian

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

Big-endian



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



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 2

```
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).

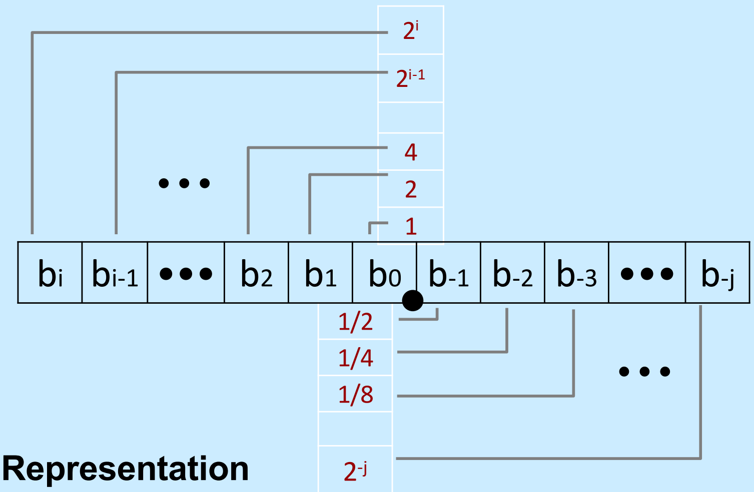
How does **printf** know that **xarray[i]** is an **unsigned char** (and thus one byte long) rather than an **int**? It turns out that **printf** is actually a macro (created using **#define**) that creates additional arguments that give the size (using **sizeof**) of its second and subsequent arguments. 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

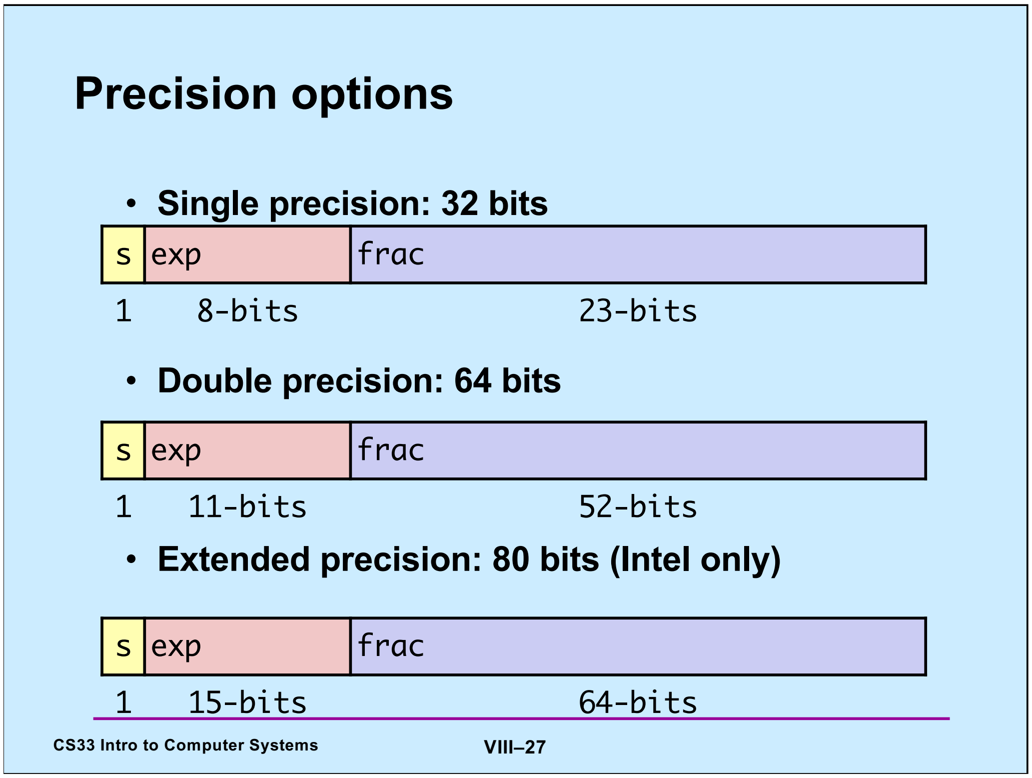
- **Numerical Form:**

$$(-1)^s \, M \;\, 2^E$$

  - 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)

s	exp	frac

Supplied by CMU.



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: exp ≠ 000…0 and exp ≠ 111…1

- Exponent coded as biased value: E = Exp – Bias
  - exp: unsigned value exp
  - 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: $M = 1.xxx…x_2$
  - xxx…x: bits of frac
  - minimum when frac=000…0 (M = 1.0)
  - maximum when frac=111…1 (M = 2.0 – ε)
  - get extra leading bit for “free”

Supplied by CMU.

Normalized Encoding Example

- **Value:** `float F = 15213.0;`
  - $15213_{10} = 11101101101101_2$
    $= 1.1101101101101_2 \times 2^{13}$

- **Significand**
  M = $1.\underline{1101101101101}_2$
  `frac` = $\underline{1101101101101}0000000000_2$

- **Exponent**
  E = **13**
  $bias$ = **127**
  exp = **140** = 10001100_2

- **Result:**

s	exp	frac
0	10001100	11011011011010000000000

Supplied by CMU.

Denormalized Values

- Condition: exp = `000…0`
- Exponent value: E = –Bias + 1 (instead of E = 0 – Bias)
- Significand coded with implied leading 0: M = $0.xxx\ldots x_2$
  - `xxx…x`: bits of `frac`
- Cases
  - `exp` = `000…0`, `frac` = `000…0`
    - » represents zero value
    - » note distinct values: +0 and –0 (why?)
  - `exp` = `000…0`, `frac` ≠ `000…0`
    - » numbers closest to 0.0
    - » equispaced

Supplied by CMU.

Special Values

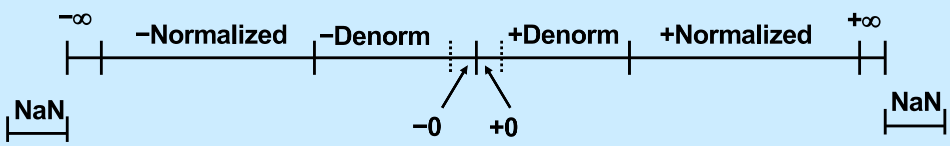
- **Condition: `exp` = 111…1**

- **Case: `exp` = 111…1, `frac` = 000…0**
  - represents value ∞ (infinity)
  - operation that overflows
  - both positive and negative
  - e.g., 1.0/0.0 = −1.0/−0.0 = +∞, 1.0/−0.0 = −∞

- **Case: `exp` = 111…1, `frac` ≠ 000…0**
  - not-a-number (NaN)
  - represents case when no numeric value can be determined
  - e.g., sqrt(–1), ∞ − ∞, ∞ × 0

Supplied by CMU.

Visualization: Floating-Point Encodings



Supplied by CMU.

Tiny Floating-Point Example

s	exp	frac
1	4-bits	3-bits

- **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	
	0 0000 000	-6	0	
	0 0000 001	-6	1/8*1/64 = 1/512	closest to zero
Denormalized numbers	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
	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
Normalized numbers	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.

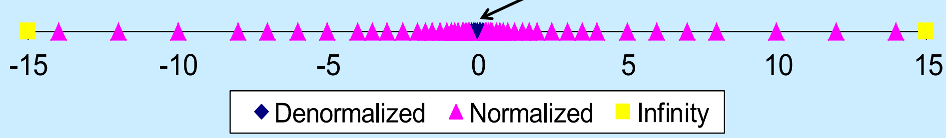
Distribution of Values

- **6-bit IEEE-like format**
  - **e = 3 exponent bits**
  - **f = 2 fraction bits**
  - **bias is $2^{3-1}-1 = 3$**

s	exp	frac
1	3-bits	2-bits

- **Notice how the distribution gets denser toward zero.**

8 values

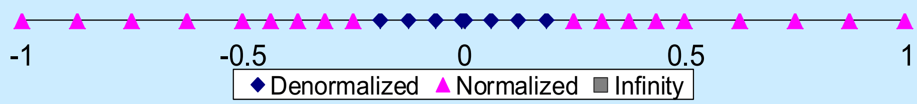


Supplied by CMU.

Distribution of Values (close-up view)

- **6-bit IEEE-like format**
  - **e = 3 exponent bits**
  - **f = 2 fraction bits**
  - **bias is 3**

s	exp	frac
1	3-bits	2-bits



Supplied by CMU.

Quiz 3

- **6-bit IEEE-like format**
  - **e = 3 exponent bits**
  - **f = 2 fraction bits**
  - **bias is 3**

s	exp	frac
1	3-bits	2-bits

What number is represented by 0 010 10?

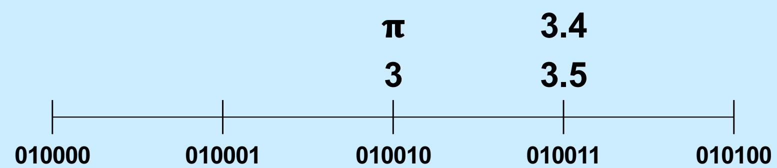
a) 3
b) 1.5
c) .75
d) none of the above

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



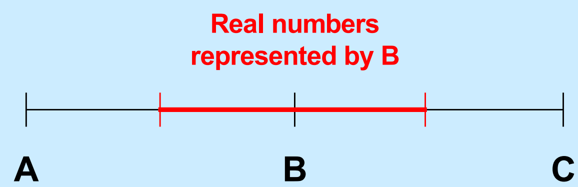
We're assuming here the six-bit floating-point format.

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 coming up soon!

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



CS33 Intro to Computer Systems VIII–40

Note that we still have to discuss rounding so as to accommodate values that are equidistant from A and B or from 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 signifcand 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 $1.5*2^{-8}$
    - » 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 $.5*2^{-9}$ to $1.5*2^{-9}$
  - 50% accuracy

CS33 Intro to Computer Systems VIII–42

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 $2^{-6} * 2^{-3}$.

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.