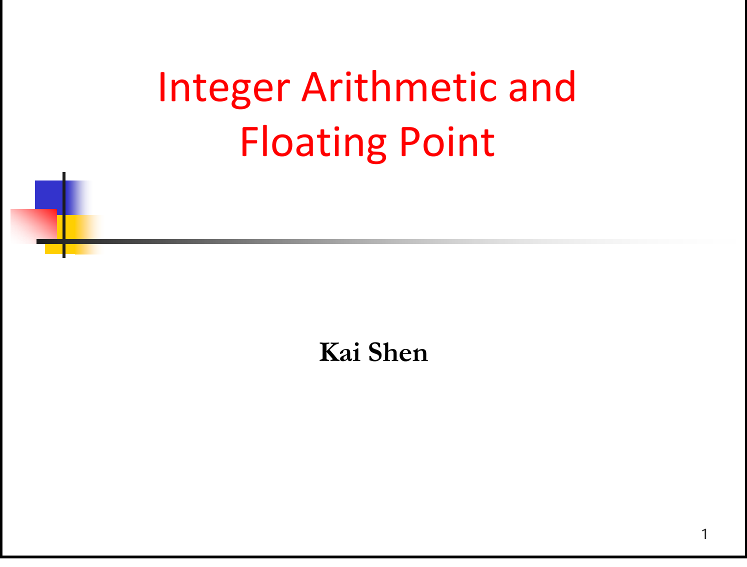
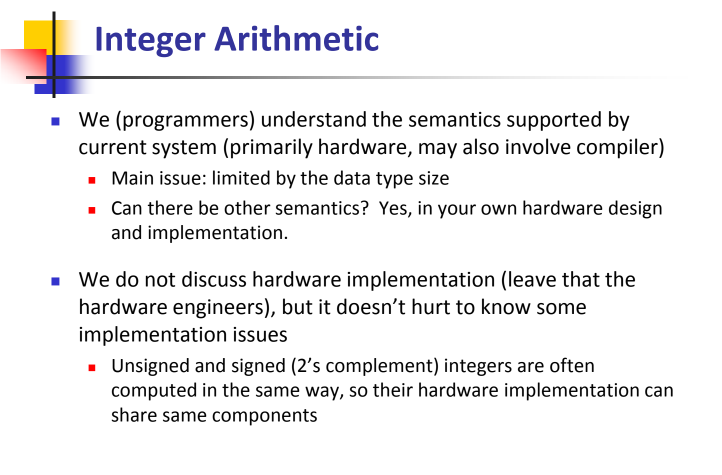




Unsigned Addition

Operands: w bits — u, $+\ v$

True sum: w+1 bits — $u + v$

Discard carry: w bits — $UAdd_w(u, v)$

- Semantics: standard addition, but ignore overflowed carry
  - Still commutative and associative
- Implements modular arithmetic

  $s \;=\; UAdd_w(u, v) \;=\; u + v \mod 2^w$

$$UAdd_w(u,v) = \begin{cases} u+v & u+v < 2^w \\ u+v-2^w & u+v \geq 2^w \end{cases}$$

3

Two's Complement Addition

Operands: w bits — u, $+\ v$

True sum: w+1 bits — $u + v$

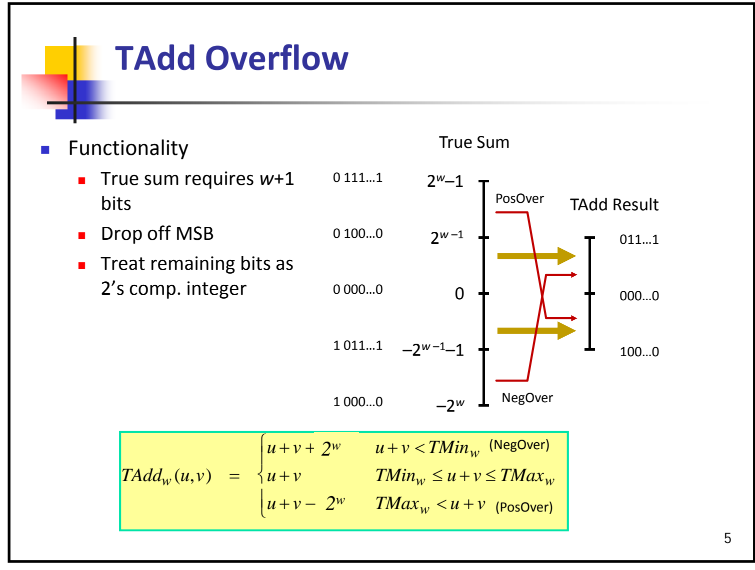
Discard carry: w bits — $TAdd_w(u, v)$

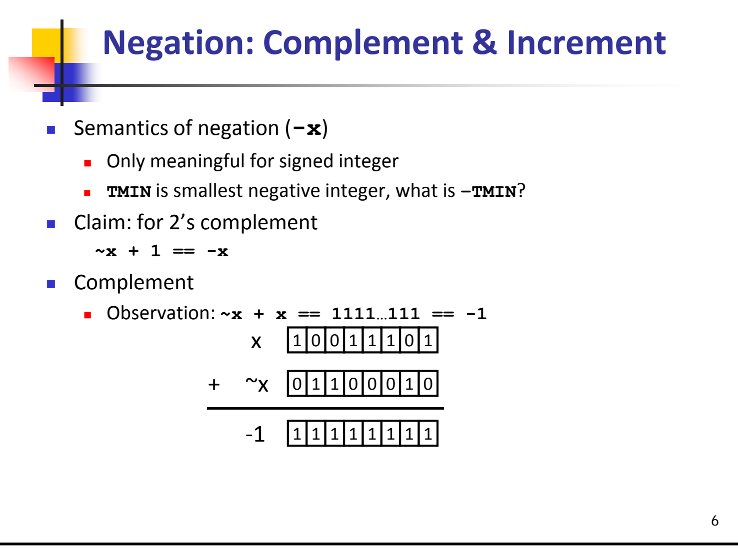
- TAdd and UAdd have identical bit-level computation
  - Signed vs. unsigned addition in C:

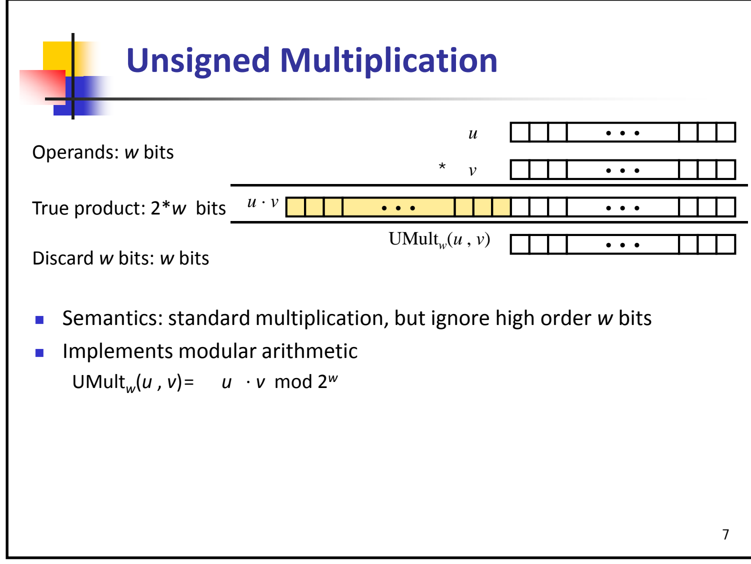
```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v
```

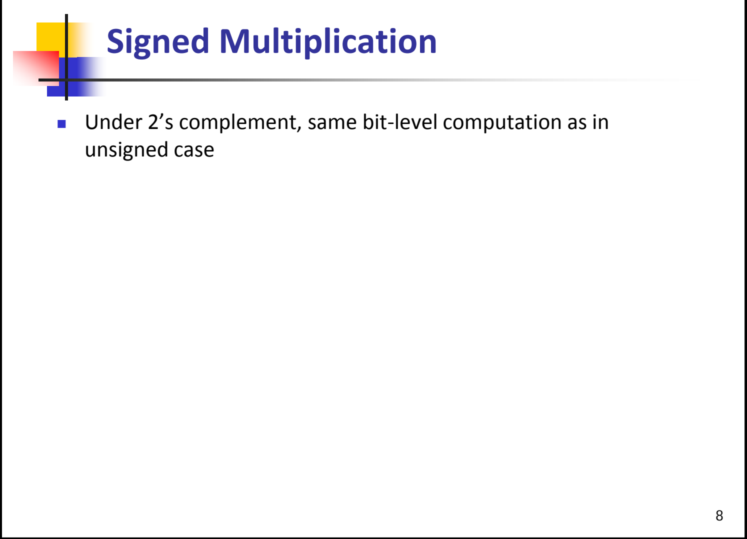
  - Will give `s == t`

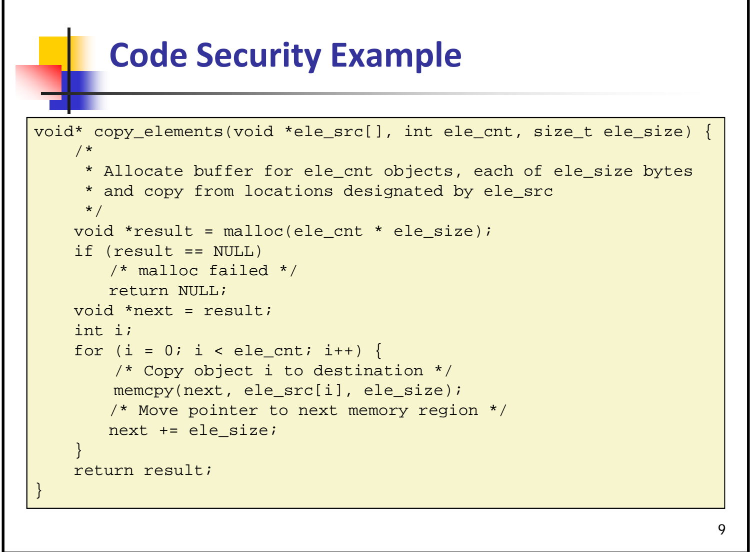
4

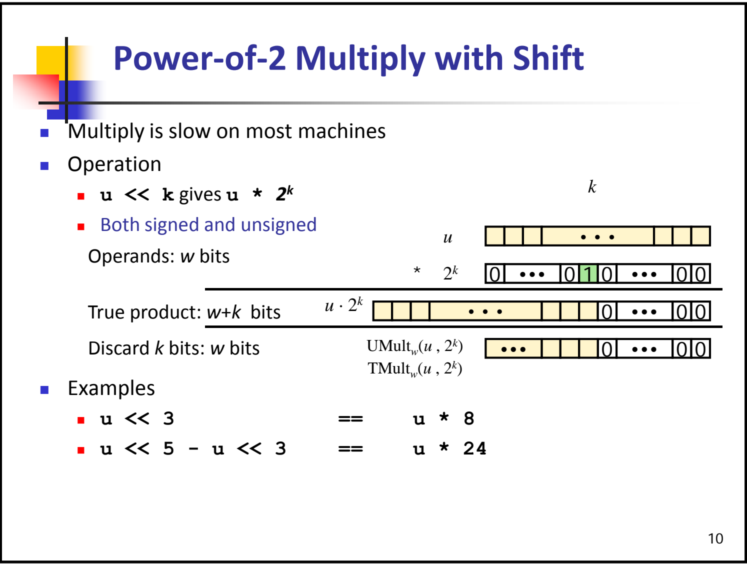

TAdd Overflow
Functionality
True sum requires w+1 bits
Drop off MSB
Treat remaining bits as 2's comp. integer
True Sum
0 111…1
0 100…0
0 000…0
1 011…1
1 000…0
PosOver
TAdd Result
011…1
000…0
100…0
NegOver
(NegOver)
(PosOver)
5

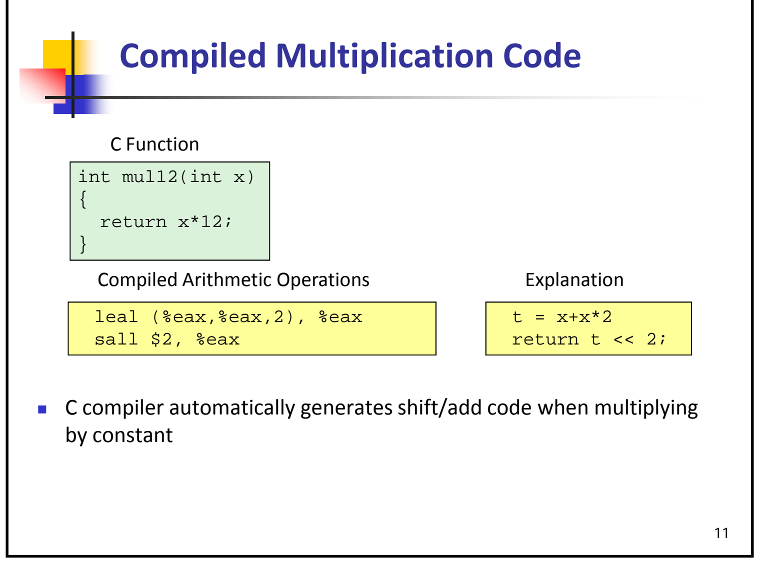

Negation: Complement & Increment
Semantics of negation (-x)
Only meaningful for signed integer
TMIN is smallest negative integer, what is -TMIN?
Claim: for 2's complement
~x + 1 == -x
Complement
Observation: ~x + x == 1111…111 == -1
x 1 0 0 1 1 1 0 1
+ ~x 0 1 1 0 0 0 1 0
-1 1 1 1 1 1 1 1 1
6

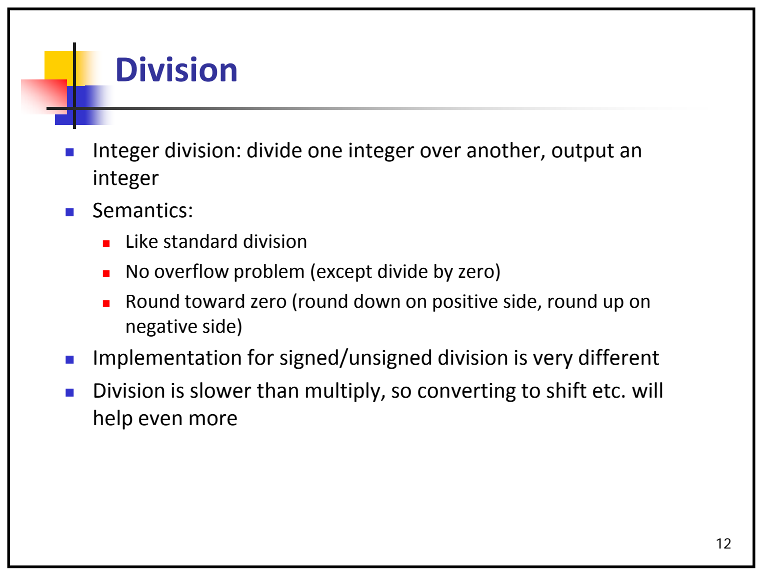

Unsigned Multiplication
Operands: w bits
True product: 2*w bits
Discard w bits: w bits
Semantics: standard multiplication, but ignore high order w bits
Implements modular arithmetic
7

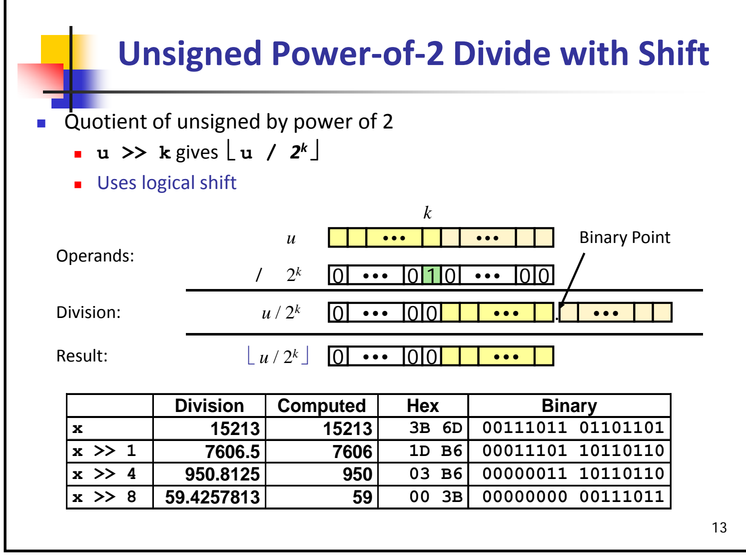

Signed Multiplication
Under 2's complement, same bit-level computation as in unsigned case
8


Code Security Example
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size) {
/*
* Allocate buffer for ele_cnt objects, each of ele_size bytes
* and copy from locations designated by ele_src
*/
void *result = malloc(ele_cnt * ele_size);
if (result == NULL)
/* malloc failed */
return NULL;
void *next = result;
int i;
for (i = 0; i < ele_cnt; i++) {
/* Copy object i to destination */
memcpy(next, ele_src[i], ele_size);
/* Move pointer to next memory region */
next += ele_size;
}
return result;
}
9

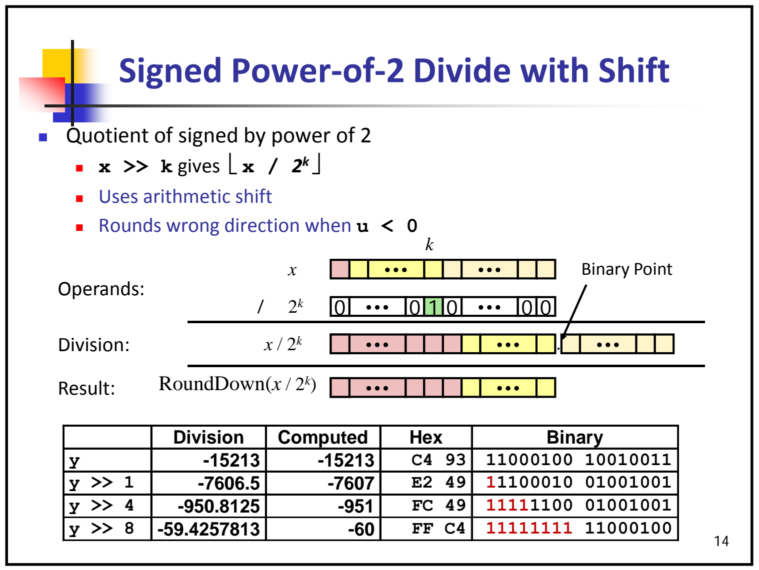

Power-of-2 Multiply with Shift
Multiply is slow on most machines
Operation
u << k gives u * 2^k
Both signed and unsigned
Operands: w bits
True product: w+k bits
Discard k bits: w bits
k
u
* 2^k
u · 2^k
UMult_w(u, 2^k)
TMult_w(u, 2^k)
Examples
u << 3 == u * 8
u << 5 - u << 3 == u * 24
10


Compiled Multiplication Code
C Function
int mul12(int x)
{
return x*12;
}
Compiled Arithmetic Operations
leal (%eax,%eax,2), %eax
sall $2, %eax
Explanation
t = x+x*2
return t << 2;
C compiler automatically generates shift/add code when multiplying by constant
11

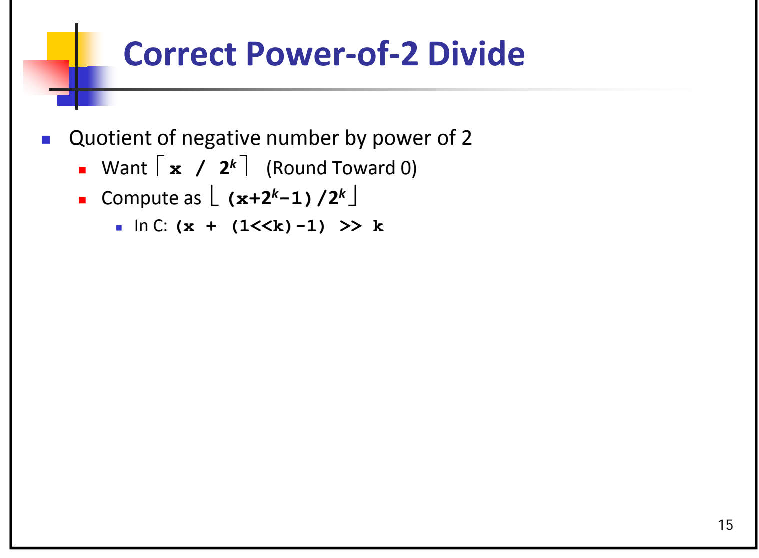

Division
Integer division: divide one integer over another, output an integer
Semantics:
Like standard division
No overflow problem (except divide by zero)
Round toward zero (round down on positive side, round up on negative side)
Implementation for signed/unsigned division is very different
Division is slower than multiply, so converting to shift etc. will help even more
12

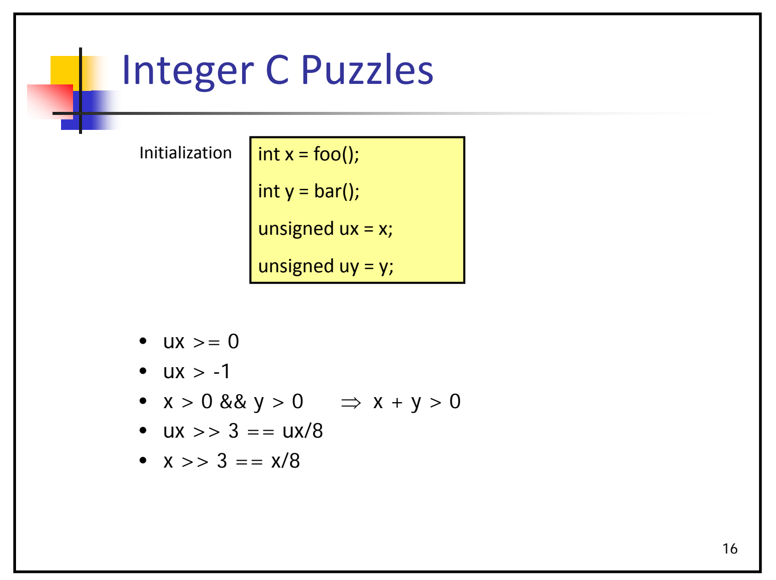


	Division	Computed	Hex	Binary
x	15213	15213	3B 6D	00111011 01101101
x >> 1	7606.5	7606	1D B6	00011101 10110110
x >> 4	950.8125	950	03 B6	00000011 10110110
x >> 8	59.4257813	59	00 3B	00000000 00111011



	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





Floating Point

- Background: fractional binary numbers
- Bit representation: IEEE floating point standard
- Rounding, addition, multiplication

17

Fractional Binary Numbers

What is 1011.101_2?

2^i, 2^{i-1}, …, 4, 2, 1

b_i	b_{i-1}	•••	b_2	b_1	b_0	b_{-1}	b_{-2}	b_{-3}	•••	b_{-j}

1/2, 1/4, 1/8, …, 2^{-j}

- Representation
  - Bits to right of "binary point" represent fractional powers of 2
  - Represents rational number: $\sum_{k=-j}^{i} b_k \times 2^k$

18

Fractional Binary Numbers: Examples

Value	Representation
5 & 3/4	101.11_2
2 & 7/8	10.111_2
1 & 7/16	1.0111_2

- **Observations**
- Divide by 2 by shifting right
- Multiply by 2 by shifting left
- Numbers of form $0.111111\ldots_2$ are just below 1.0
  - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
  - Use notation $1.0 - \varepsilon$

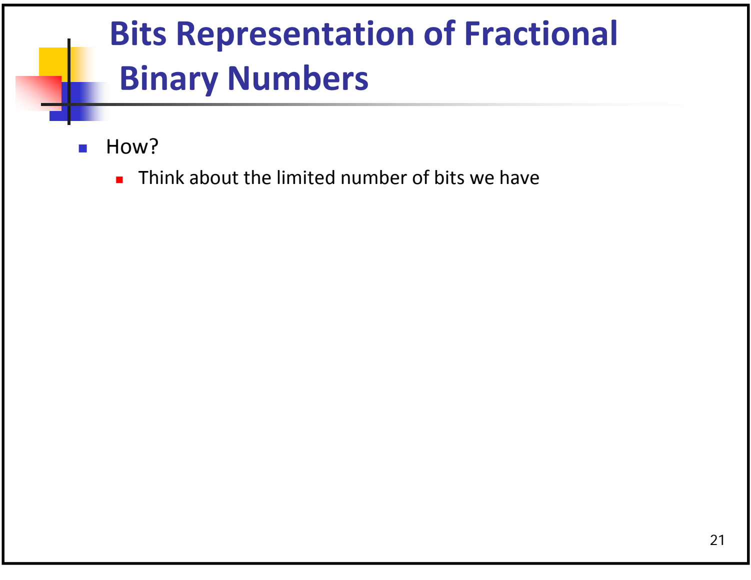
19

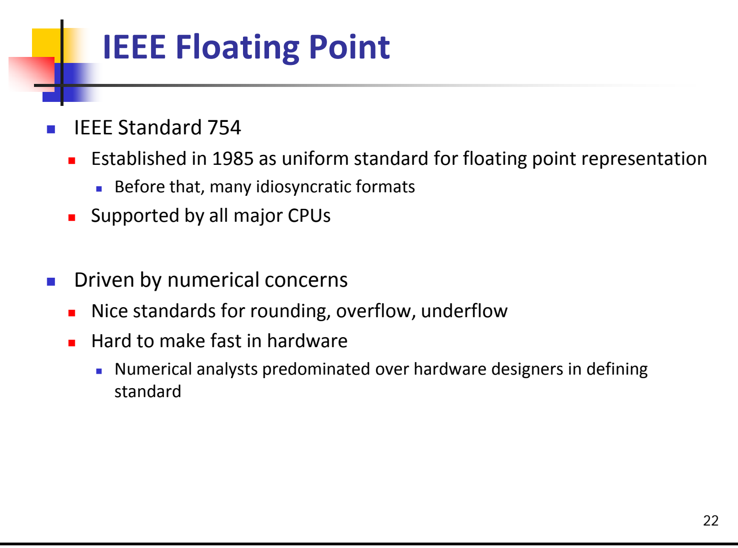
Representable Numbers

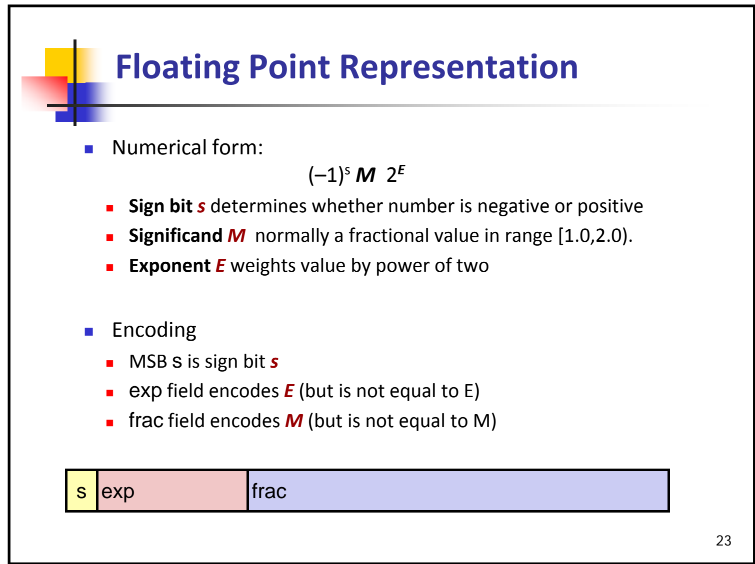
- Limitation
  - Can only exactly represent numbers of the form $x/2^k$
  - Other rational numbers have repeating bit representations

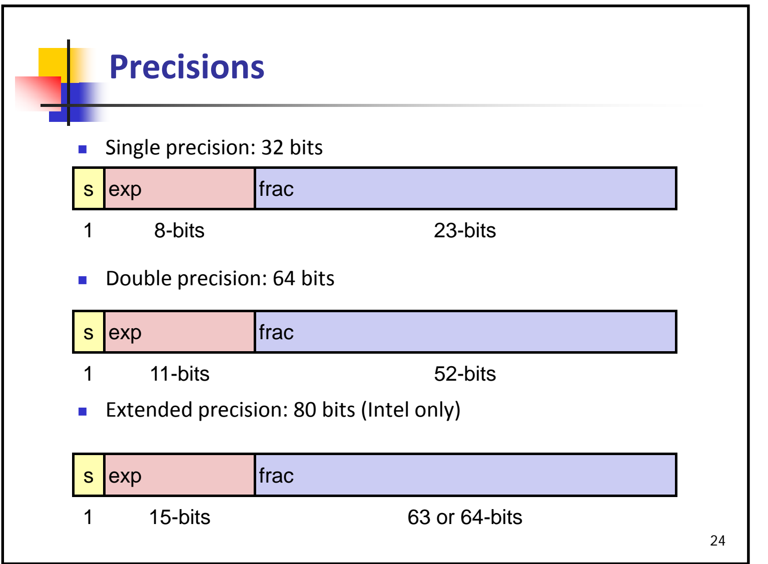
Value	Representation
1/3	$0.0101010101[01]\ldots_2$
1/5	$0.001100110011[0011]\ldots_2$
1/10	$0.0001100110011[0011]\ldots_2$

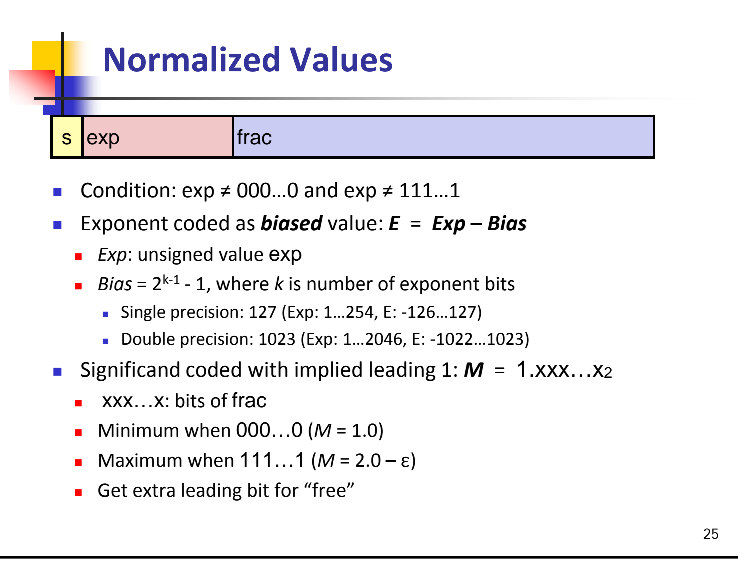
20

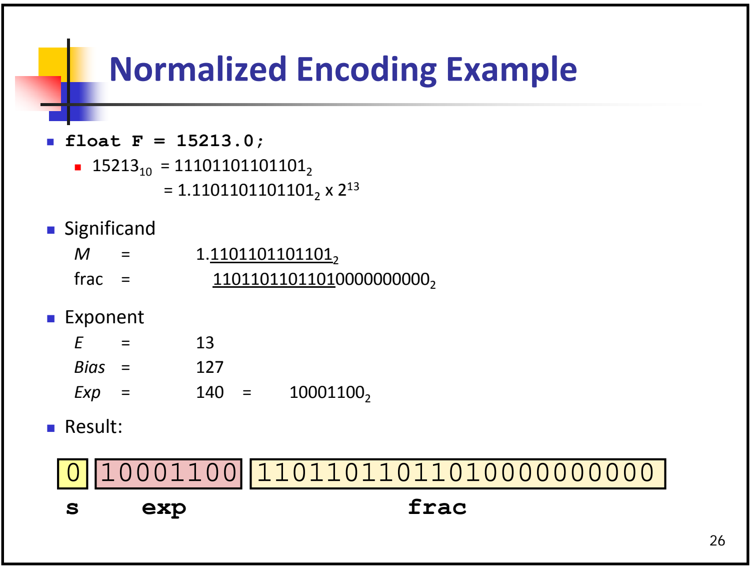

Bits Representation of Fractional Binary Numbers
How?
Think about the limited number of bits we have
21

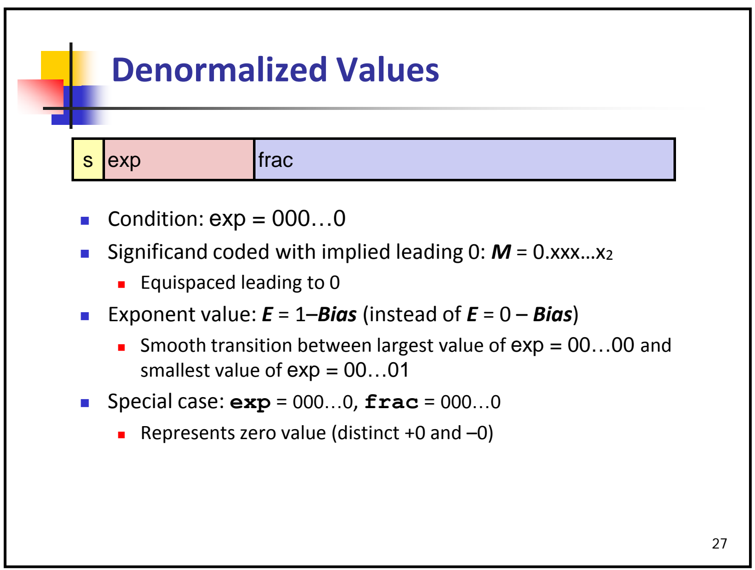

IEEE Floating Point
IEEE Standard 754
Established in 1985 as uniform standard for floating point representation
Before that, many idiosyncratic formats
Supported by all major CPUs
Driven by numerical concerns
Nice standards for rounding, overflow, underflow
Hard to make fast in hardware
Numerical analysts predominated over hardware designers in defining standard
22

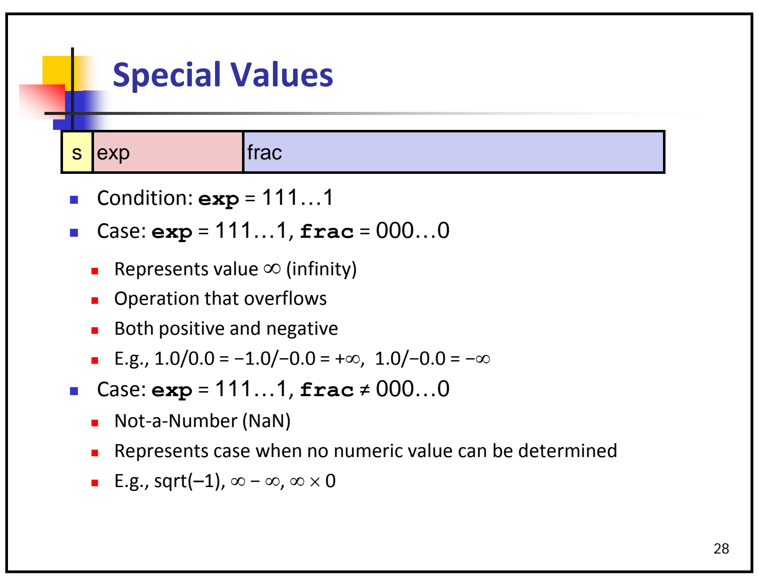

Floating Point Representation
Numerical form:
$(-1)^s M 2^E$
Sign bit s determines whether number is negative or positive
Significand M normally a fractional value in range [1.0,2.0).
Exponent E weights value by power of two
Encoding
MSB s is sign bit s
exp field encodes E (but is not equal to E)
frac field encodes M (but is not equal to M)
s
exp
frac
23

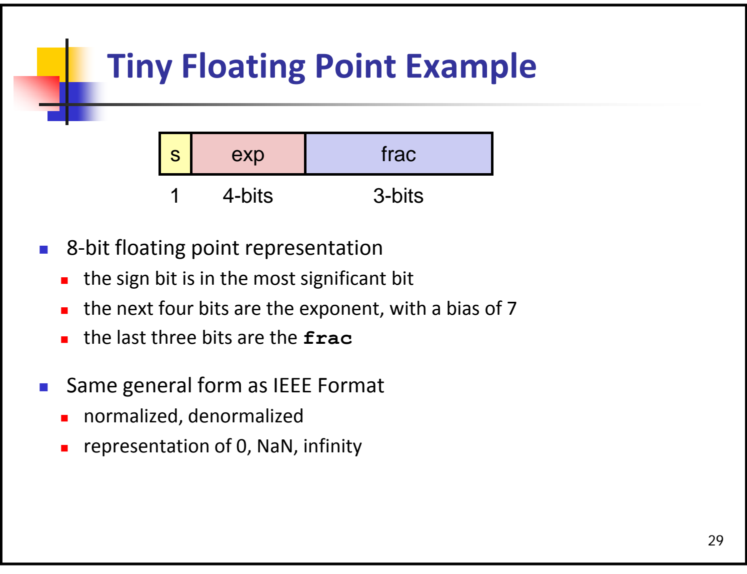

Precisions
Single precision: 32 bits
s
exp
frac
1
8-bits
23-bits
Double precision: 64 bits
s
exp
frac
1
11-bits
52-bits
Extended precision: 80 bits (Intel only)
s
exp
frac
1
15-bits
63 or 64-bits
24

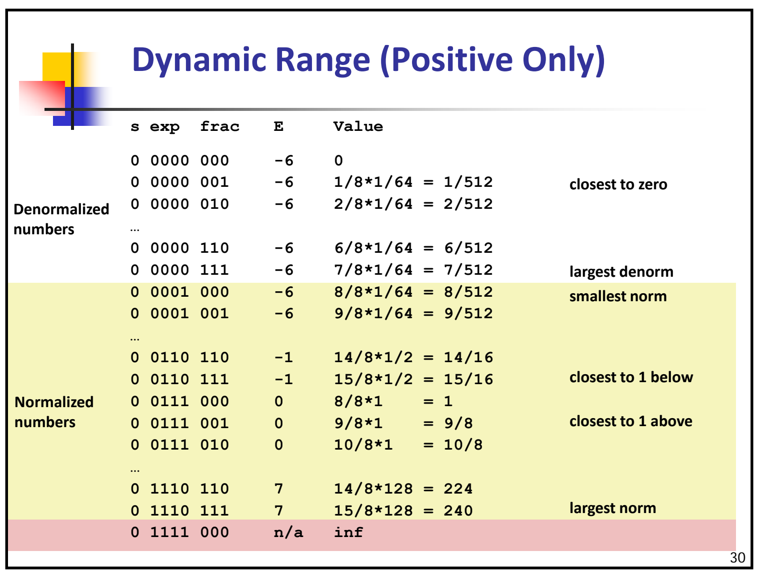

Normalized Values
s
exp
frac
Condition: exp ≠ 000…0 and exp ≠ 111…1
Exponent coded as biased value: E = Exp – Bias
Exp: unsigned value exp
Bias = 2^(k-1) - 1, where k is number of exponent bits
Single precision: 127 (Exp: 1…254, E: -126…127)
Double precision: 1023 (Exp: 1…2046, E: -1022…1023)
Significand coded with implied leading 1: M = 1.xxx…x2
xxx…x: bits of frac
Minimum when 000…0 (M = 1.0)
Maximum when 111…1 (M = 2.0 – ε)
Get extra leading bit for "free"
25

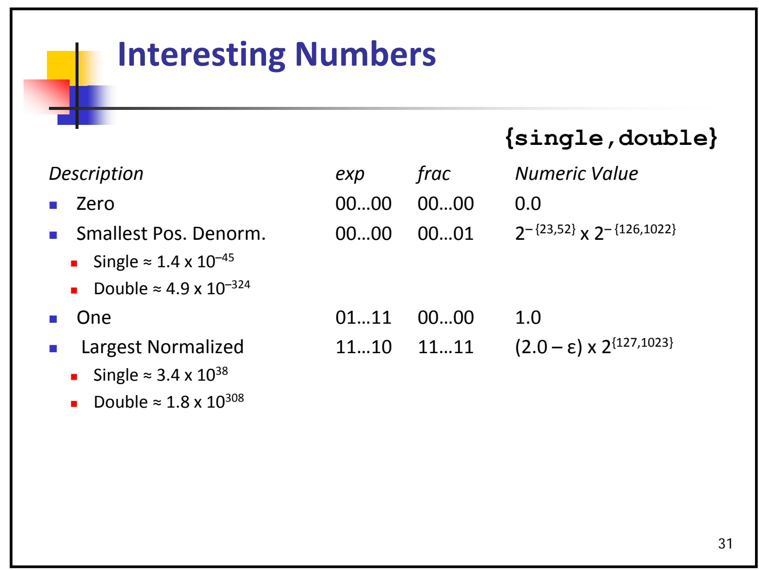

Normalized Encoding Example
float F = 15213.0;
15213_10 = 11101101101101_2
= 1.1101101101101_2 x 2^13
Significand
M = 1.1101101101101_2
frac = 11011011011010000000000_2
Exponent
E = 13
Bias = 127
Exp = 140 = 10001100_2
Result:
0 10001100 11011011011010000000000
s exp frac
26

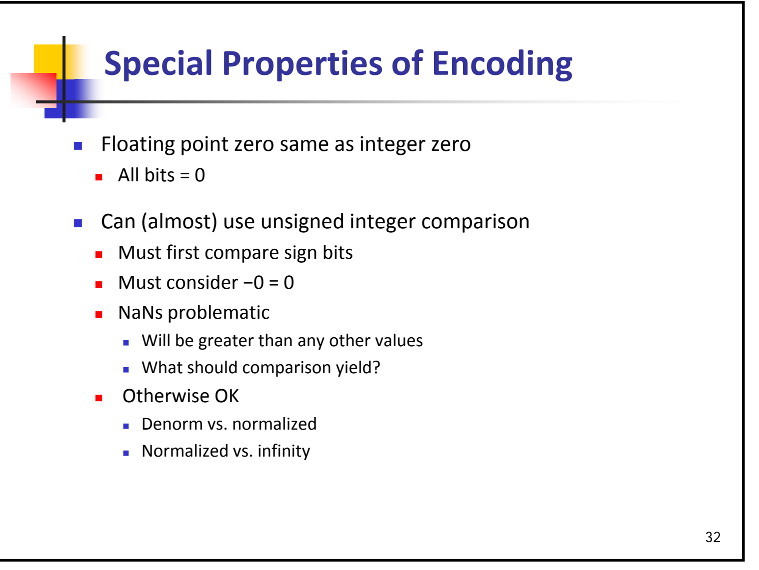

Denormalized Values
s
exp
frac
Condition: exp = 000…0
Significand coded with implied leading 0: M = 0.xxx…x2
Equispaced leading to 0
Exponent value: E = 1–Bias (instead of E = 0 – Bias)
Smooth transition between largest value of exp = 00…00 and smallest value of exp = 00…01
Special case: exp = 000…0, frac = 000…0
Represents zero value (distinct +0 and –0)
27

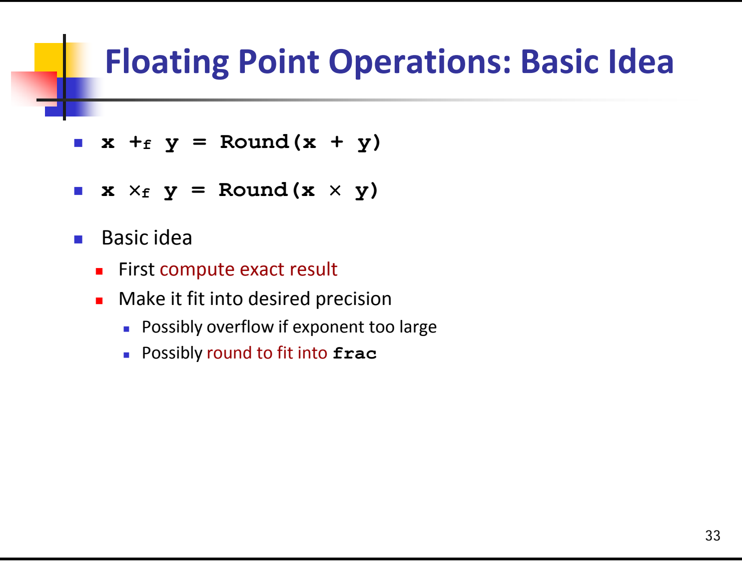

Special Values
s
exp
frac
Condition: exp = 111…1
Case: exp = 111…1, frac = 000…0
Represents value ∞ (infinity)
Operation that overflows
Both positive and negative
E.g., 1.0/0.0 = −1.0/−0.0 = +∞, 1.0/−0.0 = −∞
Case: exp = 111…1, frac ≠ 000…0
Not-a-Number (NaN)
Represents case when no numeric value can be determined
E.g., sqrt(–1), ∞ − ∞, ∞ × 0
28

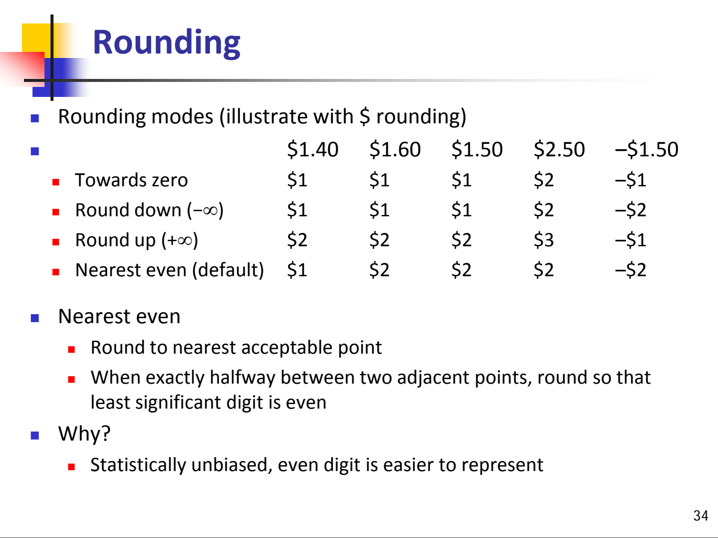

Tiny Floating Point Example
s
exp
frac
1
4-bits
3-bits
8-bit floating point representation
the sign bit is in the most significant bit
the next four bits are the exponent, with a bias of 7
the last three bits are the frac
Same general form as IEEE Format
normalized, denormalized
representation of 0, NaN, infinity
29

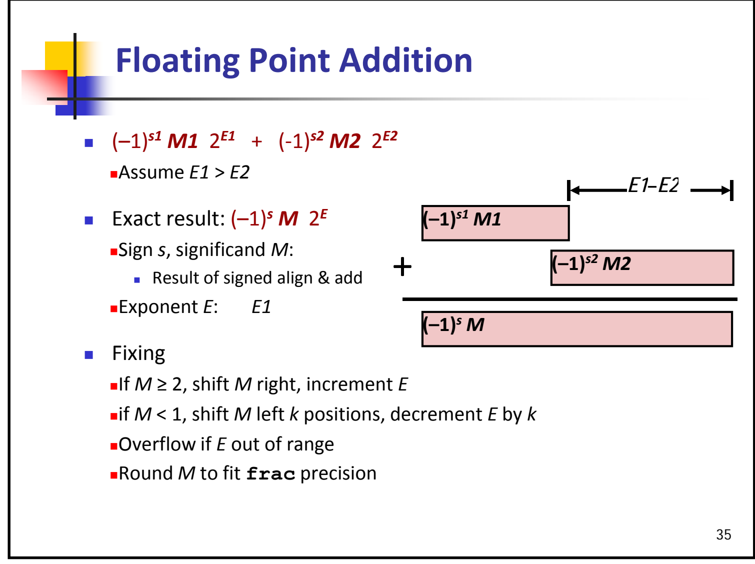

Dynamic Range (Positive Only)
s exp frac E Value
Denormalized numbers
0 0000 000 -6 0
0 0000 001 -6 1/8*1/64 = 1/512 closest to zero
0 0000 010 -6 2/8*1/64 = 2/512
…
0 0000 110 -6 6/8*1/64 = 6/512
0 0000 111 -6 7/8*1/64 = 7/512 largest denorm
Normalized numbers
0 0001 000 -6 8/8*1/64 = 8/512 smallest 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
30

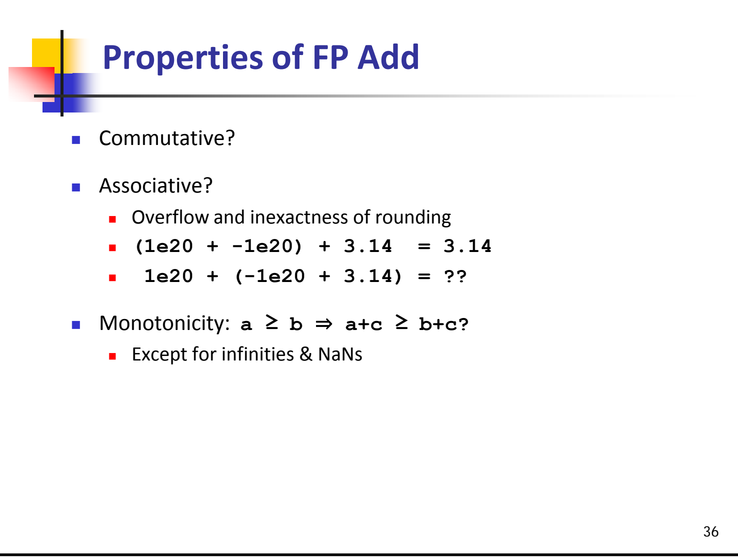

Interesting Numbers
{single,double}
Description exp frac Numeric Value
Zero 00…00 00…00 0.0
Smallest Pos. Denorm. 00…00 00…01 2– {23,52} x 2– {126,1022}
Single ≈ 1.4 x 10–45
Double ≈ 4.9 x 10–324
One 01…11 00…00 1.0
Largest Normalized 11…10 11…11 (2.0 – ε) x 2{127,1023}
Single ≈ 3.4 x 1038
Double ≈ 1.8 x 10308
31


Special Properties of Encoding
Floating point zero same as integer zero
All bits = 0
Can (almost) use unsigned integer comparison
Must first compare sign bits
Must consider –0 = 0
NaNs problematic
Will be greater than any other values
What should comparison yield?
Otherwise OK
Denorm vs. normalized
Normalized vs. infinity
32


Floating Point Operations: Basic Idea
x +f y = Round(x + y)
x ×f y = Round(x × y)
Basic idea
First compute exact result
Make it fit into desired precision
Possibly overflow if exponent too large
Possibly round to fit into frac
33


Rounding
Rounding modes (illustrate with $ rounding)
$1.40 $1.60 $1.50 $2.50 –$1.50
Towards zero $1 $1 $1 $2 –$1
Round down (-∞) $1 $1 $1 $2 –$2
Round up (+∞) $2 $2 $2 $3 –$1
Nearest even (default) $1 $2 $2 $2 –$2
Nearest even
Round to nearest acceptable point
When exactly halfway between two adjacent points, round so that least significant digit is even
Why?
Statistically unbiased, even digit is easier to represent
34


Floating Point Addition
(–1)^s1 M1 2^E1 + (-1)^s2 M2 2^E2
Assume E1 > E2
E1–E2
Exact result: (–1)^s M 2^E
(–1)^s1 M1
Sign s, significand M:
+
(–1)^s2 M2
Result of signed align & add
Exponent E: E1
(–1)^s M
Fixing
If M ≥ 2, shift M right, increment E
if M < 1, shift M left k positions, decrement E by k
Overflow if E out of range
Round M to fit frac precision
35


Properties of FP Add
Commutative?
Associative?
Overflow and inexactness of rounding
(1e20 + -1e20) + 3.14 = 3.14
1e20 + (-1e20 + 3.14) = ??
Monotonicity: a ≥ b ⇒ a+c ≥ b+c?
Except for infinities & NaNs
36

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
  - Round *M* to fit **`frac`** precision
- Implementation
  - Biggest chore is multiplying significands

37

Mathematical Properties of FP Mult

- Multiplication is commutative?
- Multiplication is associative?
  - Possibility of overflow, inexactness of rounding
- Monotonicity: **`a ≥ b & c ≥ 0 ⇒ a*c ≥ b*c?`**
  - Except for infinities & NaNs

38

Floating Point in C

- C guarantees two levels
  - **`float`** single precision
  - **`double`** double precision
- 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

39

Disclaimer

These slides were adapted from the CMU course slides provided along with the textbook of "Computer Systems: A programmer's Perspective" by Bryant and O'Hallaron. The slides are intended for the sole purpose of teaching the computer organization course at the University of Rochester.

40