Lecture 3
Integers and Floating Point Numbers

Integer Topics

- Numeric Encodings
  - Unsigned & Two's complement
- Programming Implications
  - C promotion rules
- Basic operations
  - Addition, negation, multiplication
- Programming Implications
  - Consequences of overflow
  - Using shifts to perform power-of-2 multiply/divide

C Puzzles

- Taken from old exams
- Assume machine with 32 bit word size, two's complement integers
- For each of the following C expressions, either:
  - Argue that is true for all argument values
  - Give example where not true

Initialization

```
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```

- `x < 0` ⇒ `((x*2) < 0)`
- `ux >= 0`
- `x & 7 == 7` ⇒ `(x<<30) < 0`
- `ux > -1`
- `x > y` ⇒ `-x < -y`
- `x * x >= 0`
- `x > 0 && y > 0` ⇒ `x + y > 0`
- `x >= 0` ⇒ `-x <= 0`
- `x <= 0` ⇒ `-x >= 0`

Encoding Integers

Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two's Complement

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

```
short int x =  15213;
short int y = -15213;
```

Sign Bit

- C `short` 2 bytes long

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
y	-15213	C4 93	11000100 10010011

Sign Bit

- For 2's complement, most significant bit indicates sign
  - 0 for nonnegative
  - 1 for negative

Encoding Example (Cont.)

```
x =     15213: 00111011 01101101
y =    -15213: 11000100 10010011
```

Weight	15213		-15213	
1	1	1	1	1
2	0	0	1	2
4	1	4	0	0
8	1	8	0	0
16	0	0	1	16
32	1	32	0	0
64	1	64	0	0
128	0	0	1	128
256	1	256	0	0
512	1	512	0	0
1024	0	0	1	1024
2048	1	2048	0	0
4096	1	4096	0	0
8192	1	8192	0	0
16384	0	0	1	16384
-32768	0	0	1	-32768
Sum		**15213**		**-15213**

Numeric Ranges

Unsigned Values

- $UMin = 0$
  000…0
- $UMax = 2^w - 1$
  111…1

Two's Complement Values

- $TMin = -2^{w-1}$
  100…0
- $TMax = 2^{w-1} - 1$
  011…1

Other Values

- Minus 1
  111…1

Values for $W = 16$

	Decimal	Hex	Binary
UMax	65535	FF FF	11111111 11111111
TMax	32767	7F FF	01111111 11111111
TMin	-32768	80 00	10000000 00000000
–1	-1	FF FF	11111111 11111111
0	0	00 00	00000000 00000000

Values for Different Word Sizes

	W			
	8	16	32	64
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

Observations

- $|TMin| = TMax + 1$
  - Asymmetric range
- $UMax = 2 * TMax + 1$

C Programming

- `#include <limits.h>`
  - **K&R App. B11**
- **Declares constants, e.g.,**
  - `ULONG_MAX`
  - `LONG_MAX`
  - `LONG_MIN`
- **Values platform-specific**

Unsigned & Signed Numeric Values

X	B2U(X)	B2T(X)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

Equivalence

- **Same encodings for nonnegative values**

Uniqueness

- **Every bit pattern represents unique integer value**
- **Each representable integer has unique bit encoding**

⇒ Can Invert Mappings

- **U2B(x) = B2U^{-1}(x)**
  - **Bit pattern for unsigned integer**
- **T2B(x) = B2T^{-1}(x)**
  - **Bit pattern for two's comp integer**

Casting Signed to Unsigned

C Allows Conversions from Signed to Unsigned

```
short int          x =  15213;
unsigned short int ux = (unsigned short) x;
short int          y  = -15213;
unsigned short int uy = (unsigned short) y;
```

Resulting Value

- **No change in bit representation**
- **Nonnegative values unchanged**
  - ***ux* = 15213**
- **Negative values change into (large) positive values**
  - ***uy* = 50323**

Relation Between Signed & Unsigned

Weight	-15213		50323	
1	1	1	1	1
2	1	2	1	2
4	0	0	0	0
8	0	0	0	0
16	1	16	1	16
32	0	0	0	0
64	0	0	0	0
128	1	128	1	128
256	0	0	0	0
512	0	0	0	0
1024	1	1024	1	1024
2048	0	0	0	0
4096	0	0	0	0
8192	0	0	0	0
16384	1	16384	1	16384
32768	1	*-32768*	1	*32768*
Sum		**-15213**		**50323**

$$ux = \begin{cases} x & x \geq 0 \\ x + 2^w & x < 0 \end{cases}$$

- $uy = y + 2 * 32768 = y + 65536 = x + 2^{16}$

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 same as U2T and T2U

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

- Implicit casting also occurs via assignments and procedure calls

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


Casting Surprises

Expression Evaluation

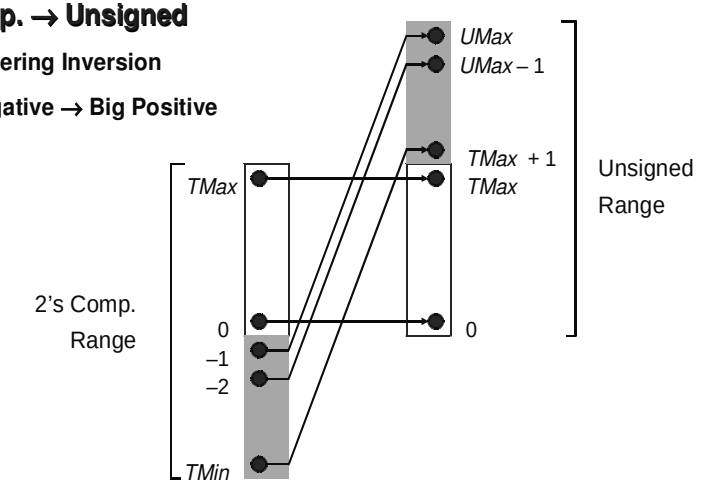
- If mix unsigned and signed in single expression, signed values implicitly cast to unsigned
- Including comparison operations <, >, ==, <=, >=
- Examples for $W = 32$

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

Explanation of Casting Surprises

2's Comp. → Unsigned

- Ordering Inversion
- Negative → Big Positive



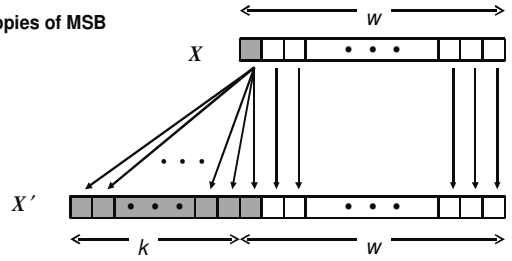
Sign Extension

Task:

- Given *w*-bit signed integer *x*
- Convert it to *w*+*k*-bit integer with same value

Rule:

- Make *k* copies of sign bit:
- $X' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2}, \ldots, x_0$



Sign Extension Example

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

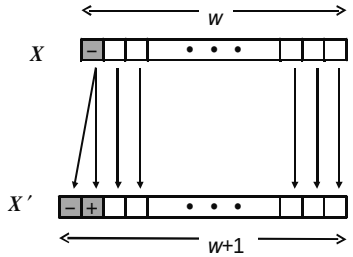
	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

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

Justification For Sign Extension

Prove Correctness by Induction on *k*

- Induction Step: extending by single bit maintains value



- Key observation: $-2^{w-1} = -2^{w} + 2^{w-1}$
- Look at weight of upper bits:

x $\quad -2^{w-1} x_{w-1}$

x' $\quad -2^{w} x_{w-1} + 2^{w-1} x_{w-1} \quad = \quad -2^{w-1} x_{w-1}$

Why Should I Use Unsigned?

***Don't* Use Just Because Number Nonzero**

- C compilers on some machines generate less efficient code

```
unsigned i;
for (i = 1; i < cnt; i++)
  a[i] += a[i-1];
```

- Easy to make mistakes

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

***Do* Use When Performing Modular Arithmetic**

- Multiprecision arithmetic
- Other esoteric stuff

***Do* Use When Need Extra Bit's Worth of Range**

- Working right up to limit of word size

Negating with Complement & Increment

Claim: Following Holds for 2's Complement

`~x + 1 == -x`

Complement

- **Observation:** `~x + x == 1111…11`$_2$ `== -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
```

Increment

- `~x + x + (-x + 1)  == -1 + (-x + 1)`
- `~x + 1  ==  -x`

Warning: Be cautious treating `int`'s as integers

- **OK here**

Comp. & Incr. Examples

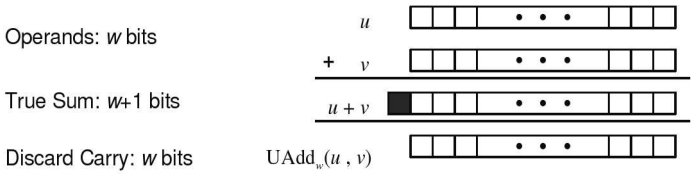
x = 15212

	Decimal	Hex	Binary
x	15212	3B 6D	00111011 01101100
~x	-15213	C4 92	11000100 10010011
~x+1	-15212	C4 93	11000100 10010**100**
y	-15212	C4 93	11000100 10010100

0

	Decimal	Hex	Binary
0	0	00 00	00000000 00000000
~0	-1	FF FF	11111111 11111111
~0+1	0	00 00	00000000 00000000

Unsigned Addition



Standard Addition Function

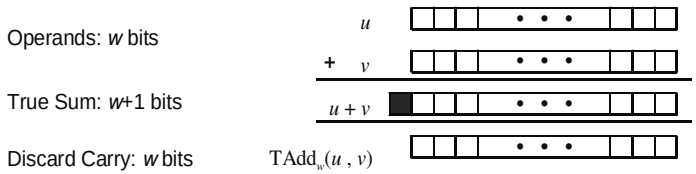
- **Ignores carry output**

Implements Modular Arithmetic

$s = \mathrm{UAdd}_w(u, v) = u + v \bmod 2^w$

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

Two's Complement Addition



TAdd and UAdd have Identical Bit-Level Behavior

- **Signed vs. unsigned addition in C:**

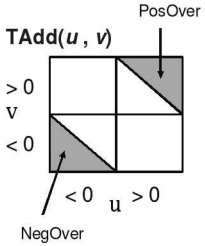
```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v
```

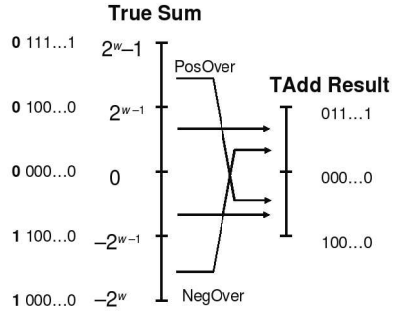
- **Will give `s == t`**

Characterizing TAdd

Functionality

- True sum requires w+1 bits
- Drop off MSB
- Treat remaining bits as 2's comp. integer





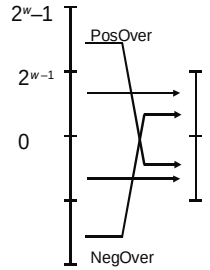
$$TAdd_w(u,v) = \begin{cases} u+v+2^{w-1} & u+v < TMin_w \text{ (NegOver)} \\ u+v & TMin_w \le u+v \le TMax_w \\ u+v-2^{w-1} & TMax_w < u+v \text{ (PosOver)} \end{cases}$$

Detecting 2's Comp. Overflow

Task

- Given s = $TAdd_w(u , v)$
- Determine if s = $Add_w(u , v)$
- Example

```
int s, u, v;
s = u + v;
```



Claim

- Overflow iff either:
  - $u, v < 0, s \ge 0$ (NegOver)
  - $u, v \ge 0, s < 0$ (PosOver)

```
ovf = (u<0 == v<0) && (u<0 != s<0);
```


Multiplication

Computing Exact Product of w-bit numbers x, y

- Either signed or unsigned

Ranges

- Unsigned: $0 \le x * y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Up to $2w$ bits
- Two's complement min: $x * y \ge (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
  - Up to $2w-1$ bits
- Two's complement max: $x * y \le (-2^{w-1})^2 = 2^{2w-2}$
  - Up to $2w$ bits, but only for $(TMin_w)^2$

Maintaining Exact Results

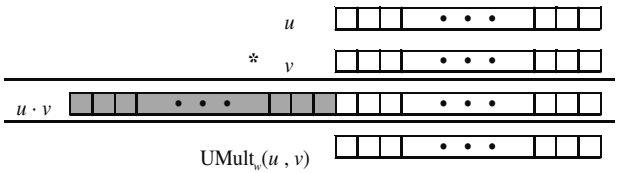
- Would need to keep expanding word size with each product computed
- Done in software by "arbitrary precision" arithmetic packages
- E.g. Integers in Python and BigInteger in Java

Unsigned Multiplication in C

Operands: w bits

True Product: $2*w$ bits

Discard w bits: w bits



Standard Multiplication Function

- Ignores high order w bits

Implements Modular Arithmetic

$UMult_w(u , v) = u \cdot v \mod 2^w$

Unsigned vs. Signed Multiplication

Unsigned Multiplication

```
unsigned ux = (unsigned) x;
unsigned uy = (unsigned) y;
unsigned up = ux * uy
```

- Truncates product to *w*-bit number $up = \text{UMult}_w(ux, uy)$
- Modular arithmetic: $up = ux \cdot uy \mod 2^w$

Two's Complement Multiplication

```
int x, y;
int p = x * y;
```

- Compute exact product of two *w*-bit numbers *x*, *y*
- Truncate result to *w*-bit number $p = \text{TMult}_w(x, y)$

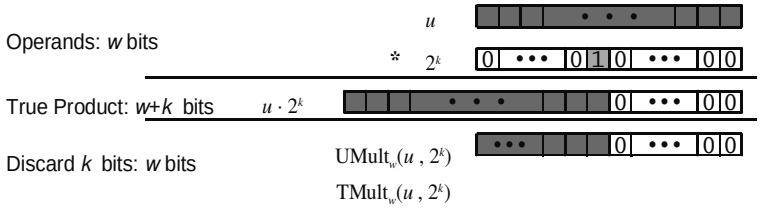
Relation

- Signed multiplication gives same bit-level result as unsigned
- `up == (unsigned)p`

Power-of-2 Multiply with Shift

Operation

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



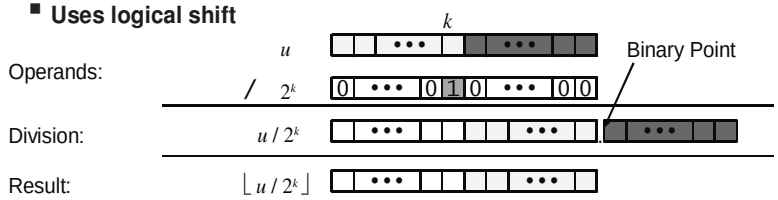
Examples

- `u << 3 == u * 8`
- `u << 5 - u << 3 == u * 24`
- Most machines shift and add much faster than multiply
  - Compiler generates this code automatically

Unsigned Power-of-2 Divide with Shift

Quotient of Unsigned by Power of 2

- `u >> k` gives $\lfloor u / 2^k \rfloor$
- Uses logical shift

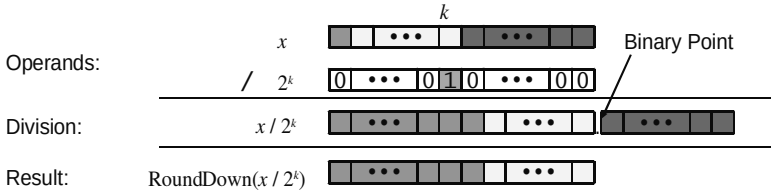


	Division	Computed	Hex	Binary
x	15213	15213	3B 6D	00111011 01101101
x >> 1	7606.5	7606	1D B6	**0**0011101 10110110
x >> 4	950.8125	950	03 B6	**0000** 0011 10110110
x >> 8	59.4257813	59	00 3B	**00000000** 00111011

Signed Power-of-2 Divide with Shift

Quotient of Signed by Power of 2

- `x >> k` gives $\lfloor x / 2^k \rfloor$
- Uses arithmetic shift
- Rounds wrong direction when `u < 0`



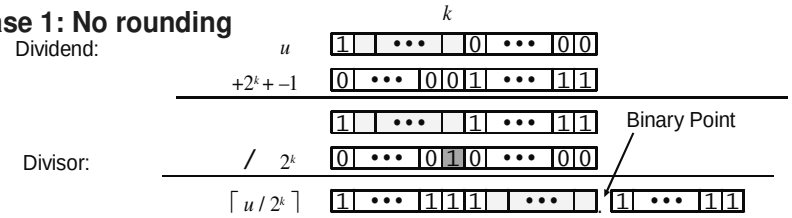
	Division	Computed	Hex	Binary
y	-15213	-15213	C4 93	11000100 10010011
y >> 1	-7606.5	-7607	E2 49	**1**1100010 01001001
y >> 4	-950.8125	-951	FC 49	**1111** 1100 01001001
y >> 8	-59.4257813	-60	FF C4	**11111111** 11000100

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

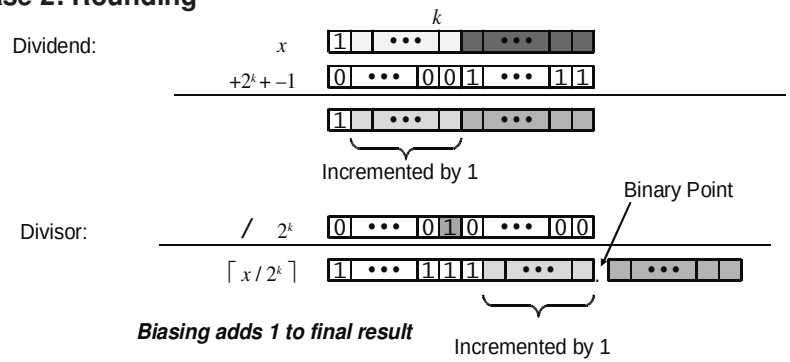
Case 1: No rounding



Biasing has no effect

Correct Power-of-2 Divide (Cont.)

Case 2: Rounding



Biasing adds 1 to final result

C Integer Puzzle Answers

- Assume machine with 32 bit word size, two's comp. integers
- *TMin* makes a good counterexample in many cases

`x < 0`	⇒	`((x*2) < 0)`	**False:**	*TMin*
`ux >= 0`			**True:**	0 = *UMin*
`x & 7 == 7`	⇒	`(x<<30) < 0`	**True:**	$x_1 = 1$
`ux > -1`			**False:**	0
`x > y`	⇒	`-x < -y`	**False:**	−1, *TMin*
`x * x >= 0`			**False:**	30426
`x > 0 && y > 0`	⇒	`x + y > 0`	**False:**	*TMax*, *TMax*
`x >= 0`	⇒	`-x <= 0`	**True:**	−*TMax* < 0
`x <= 0`	⇒	`-x >= 0`	**False:**	*TMin*

Floating Point Topics

- IEEE Floating Point Standard
- Rounding
- Floating Point Operations
- Mathematical properties

Floating Point Puzzles

- For each of the following C expressions, either:
  - Argue that it is true for all argument values
  - Explain why not true

```
int x = …;
float f = …;
double d = …;
```

Assume neither d nor f is NaN

- `x == (int)(float) x`
- `x == (int)(double) x`
- `f == (float)(double) f`
- `d == (float) d`
- `f == -(-f);`
- `2/3 == 2/3.0`
- `d < 0.0 ⇒ ((d*2) < 0.0)`
- `d > f ⇒ -f > -d`
- `d * d >= 0.0`
- `(d+f)-d == f`

IEEE Floating Point

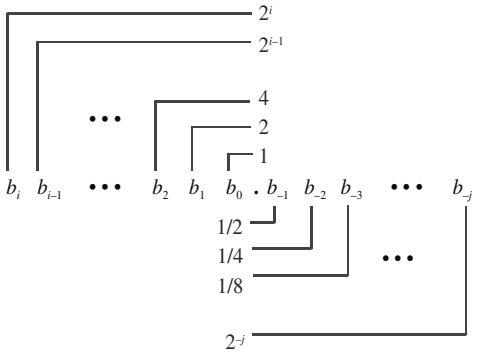
IEEE Standard 754

- Established in 1985 as uniform standard for floating point arithmetic
  - Before that, many idiosyncratic formats
- Supported by all major CPUs

Driven by Numerical Concerns

- Nice standards for rounding, overflow, underflow
- Hard to make go fast
  - Numerical analysts predominated over hardware types in defining standard

Fractional Binary Numbers



Representation

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

Frac. Binary Number Examples

Value	Representation
5 3/4	101.11_2
2 7/8	10.111_2
63/64	0.111111_2

Observations

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

Representable Numbers

Limitation

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

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

Floating Point Representation

Numerical Form

- $-1^s M \times 2^E$
  - 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
- exp field encodes *E*
- frac field encodes *M*

Floating Point Precisions

Encoding

- MSB s is sign bit
- exp field encodes *E*
- frac field encodes *M*

Sizes

- Single precision: 8 exp bits, 23 frac bits
  - 32 bits total
- Double precision: 11 exp bits, 52 frac bits
  - 64 bits total
- Extended precision: 15 exp bits, 63 frac bits
  - Only found in Intel-compatible machines
  - Stored in 80 bits
    - » 1 bit wasted
- Quad precision: 15 exp bits, 112 frac bits
  - Stored in 128 bits

"Normalized" Numeric Values

Condition

- exp ≠ 000…0 and exp ≠ 111…1

Exponent coded as *biased* value

E = Exp – Bias

- *Exp* : unsigned value denoted by exp
- *Bias* : Bias value
  - » Single precision: 127 (*Exp*: 1…254, *E*: -126…127)
  - » Double precision: 1023 (*Exp*: 1…2046, *E*: -1022…1023)
  - » Quad (Extended) precision: 16383 (*Exp*: 32766, *E*: -16382...16383)
  - » in general: *Bias* = $2^{e-1} - 1$, where e is number of exponent bits

Significand coded with implied leading 1

$M = 1.xxx\ldots x_2$

- 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"

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

Floating Point Representation:								
Hex:	4	6	6	D	B	4	0	0
Binary:	0100	0110	0110	1101	1011	0100	0000	0000
140:	100	0110	0					
15213:			*1*110	1101	1011	01		

Denormalized Values

Condition

- exp = 000…0

Value

- Exponent value *E* = –*Bias* + 1
- Significand value *M* = $0.xxx\ldots x_2$
  - xxx…x: bits of frac

Cases

- exp = 000…0, frac = 000…0
  - Represents value 0
  - Note that have distinct values +0 and –0
- exp = 000…0, frac ≠ 000…0
  - Numbers very close to 0.0
  - Lose precision as get smaller
  - "Gradual underflow"

Special Values

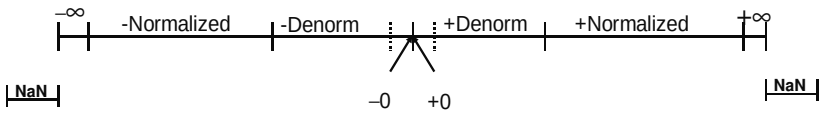
Condition

- exp = 111…1

Cases

- 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 = –∞
- exp = 111…1, frac ≠ 000…0
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., sqrt(–1), ∞ – ∞, 0/0

Summary of Floating Point Real Number Encodings

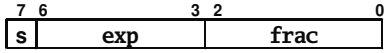


Tiny Floating Point Example

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



Values Related to the Exponent

Exp	exp	E	2^E	
0	0000	-6	1/64	(denorms)
1	0001	-6	1/64	
2	0010	-5	1/32	
3	0011	-4	1/16	
4	0100	-3	1/8	
5	0101	-2	1/4	
6	0110	-1	1/2	
7	0111	0	1	
8	1000	+1	2	
9	1001	+2	4	
10	1010	+3	8	
11	1011	+4	16	
12	1100	+5	32	
13	1101	+6	64	
14	1110	+7	128	
15	1111	n/a		(inf, NaN)

Dynamic Range

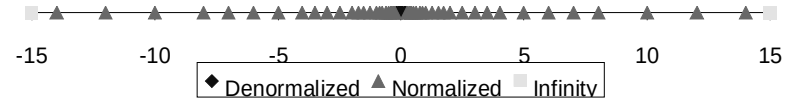
	s	exp	frac	*E*	Value	
Denormalized numbers	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	←closest to zero
	0	0000	010	-6	2/8*1/64 = 2/512	
	…					
	0	0000	110	-6	6/8*1/64 = 6/512	
	0	0000	111	-6	7/8*1/64 = 7/512	← largest denorm
Normalized numbers	0	0001	000	-6	8/8*1/64 = 8/512	← smallest norm
	0	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	

Distribution of Values

6-bit IEEE-like format

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

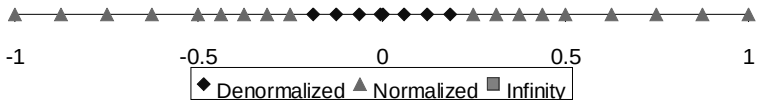
Notice how the distribution gets denser toward zero.



Distribution of Values (close-up view)

6-bit IEEE-like format

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



Interesting Numbers

Description	exp	frac	Numeric Value
Zero	00…00	00…00	0.0
Smallest Pos. Denorm.	00…00	00…01	$2^{-\{23,52\}} \times 2^{-\{126,1022\}}$
▪ Single ≈ 1.4×10^{-45}			
▪ Double ≈ 4.9×10^{-324}			
Largest Denormalized	00…00	11…11	$(1.0 - \varepsilon) \times 2^{-\{126,1022\}}$
▪ Single ≈ 1.18×10^{-38}			
▪ Double ≈ 2.2×10^{-308}			
Smallest Pos. Normalized	00…01	00…00	$1.0 \times 2^{-\{126,1022\}}$
▪ Just larger than largest denormalized			
One	01…11	00…00	1.0
Largest Normalized	11…10	11…11	$(2.0 - \varepsilon) \times 2^{\{127,1023\}}$
▪ Single ≈ 3.4×10^{38}			
▪ Double ≈ 1.8×10^{308}			

Special Properties of Encoding

FP Zero Same as Integer Zero

- All bits = 0
- But -0.0 = TMIN

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

Floating Point Operations

Conceptual View

- First compute exact result
- Make it fit into desired precision
  - Possibly overflow if exponent too large
  - Possibly round to fit into frac

Rounding Modes

	1.4	1.6	1.5	2.5	–1.5
▪ Round towards Zero	1	1	1	2	–1
▪ Round down (-∞)	1	1	1	2	–2
▪ Round up (+∞)	2	2	2	3	–1
▪ Round towards Nearest Even (default)	1	2	2	2	–2

Note:

1. Round down: rounded result is close to but no greater than true result.
2. Round up: rounded result is close to but no less than true result.

Closer Look at Round-To-Even

Default Rounding Mode

- Hard to get any other kind without dropping into assembly
- All others are statistically biased
  - Sum of set of positive numbers will consistently be over- or under-estimated

Applying to Other Decimal Places / Bit Positions

- When exactly halfway between two possible values
  - Round so that least significant digit is even
- E.g., round to nearest hundredth

1.2349999	1.23	(Less than half way)
1.2350001	1.24	(Greater than half way)
1.2350000	1.24	(Half way—round up)
1.2450000	1.24	(Half way—round down)

Rounding Binary Numbers

Binary Fractional Numbers

- "Even" when least significant bit is 0
- Half way when bits to right of rounding position = $100\ldots_2$

Examples

- Round to nearest 1/4 (2 bits right of binary point)

Value	Binary	Rounded	Action	Rounded Value
2 3/32	10.00011_2	10.00_2	(<1/2—down)	2
2 3/16	10.00110_2	10.01_2	(>1/2—up)	2 1/4
2 7/8	10.11100_2	11.00_2	(1/2—up)	3
2 5/8	10.10100_2	10.10_2	(1/2—down)	2 1/2

FP Multiplication

Operands

$(-1)^{s1}\ M1 \times 2^{E1}$ * $(-1)^{s2}\ M2 \times 2^{E2}$

Exact Result

$(-1)^{s}\ M\ 2^{E}$

- Sign s: $s1 \wedge s2$
- Significand M: $M1 * M2$
- Exponent E: $E1 + E2$

Fixing

- If $M \geq 2$, shift M right, increment E
- If E out of range, overflow
- Round M to fit `frac` precision

Implementation

- Biggest chore is multiplying significands

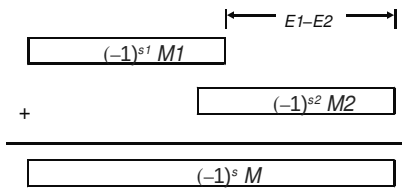
FP Addition

Operands

$(-1)^{s1}\ M1\ 2^{E1}$

$(-1)^{s2}\ M2\ 2^{E2}$

- Assume $E1 > E2$



Exact Result

$(-1)^{s}\ M\ 2^{E}$

- Sign s, significand M:
  - Result of signed align & add
- Exponent E: $E1$

Fixing

- If $M \geq 2$, 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

Floating Point in C

C Guarantees Two Levels

`float` single precision

`double` double precision

Conversions

- Casting between `int`, `float`, and `double` changes numeric values
- Note that casting between `int` and `float` changes bit-representation
- `Double` or `float` to `int`
  - Truncates fractional part
  - Like rounding toward zero
  - Not defined when out of range
    - » Generally saturates to TMin or TMax
- `int` to `double`
  - Exact conversion, as long as int has ≤ 53 bit word size
- `int` to `float`
  - Will round according to rounding mode

Answers to Floating Point Puzzles

```
int x = …;
float f = …;
double d = …;
```

Assume neither `d` nor `f` is NAN

- `x == (int)(float) x` — No: 24 bit significand
- `x == (int)(double) x` — Yes: 53 bit significand
- `f == (float)(double) f` — Yes: increases precision
- `d == (float) d` — No: loses precision
- `f == -(-f);` — Yes: Just change sign bit
- `2/3 == 2/3.0` — No: 2/3 == 0
- `d < 0.0 ⇒ ((d*2) < 0.0)` — Yes!
- `d > f ⇒ -f > -d` — Yes!
- `d * d >= 0.0` — Yes!
- `(d+f)-d == f` — No: Not associative

Summary

IEEE Floating Point Has Clear Mathematical Properties

- Represents numbers of form $M \times 2^E$
- Can reason about operations independent of implementation
  - As if computed with perfect precision and then rounded
- Not the same as real arithmetic
  - Violates associativity/distributivity
  - Makes life difficult for compilers & serious numerical applications programmers