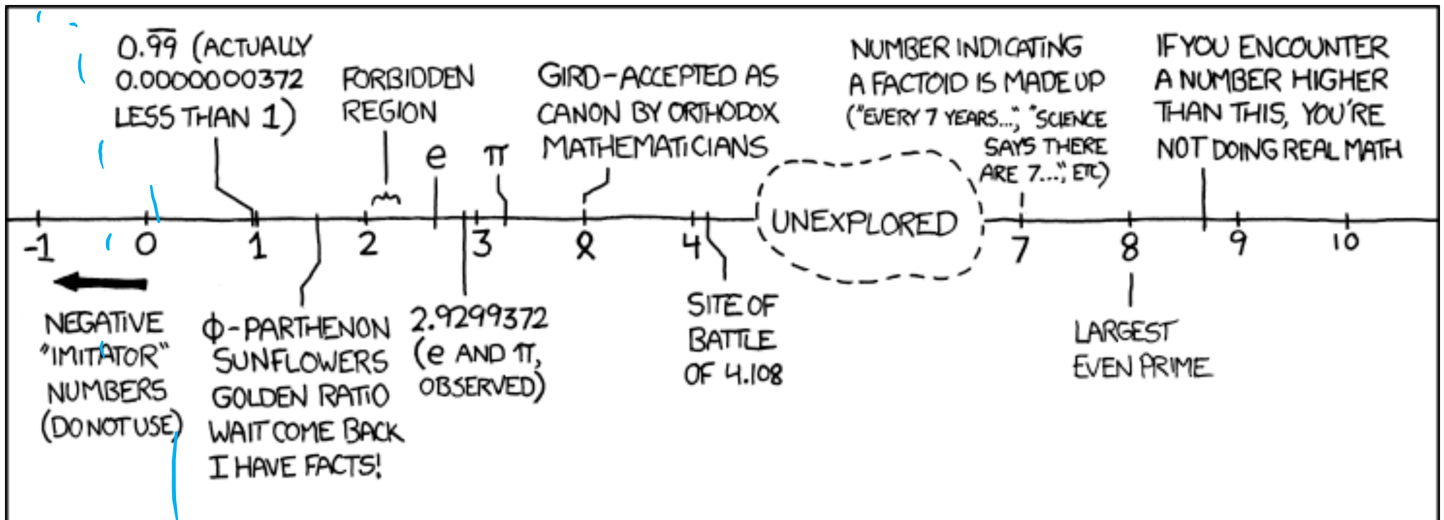
Ints and Floating Point

CSE 351 Winter 2017



http://xkcd.com/899/

Administrivia

- Lab 1 due Friday
  - How is it going?
- HW 1 out today
  - Numerical representation and executable inspection

Using Shifts and Masks

- Extract the 2nd most significant *byte* of an `int`:
  - First shift, then mask: `(x>>16) & 0xFF`

MSB LSB

x	00000001 00000010 ~~00000011 00000100~~
x>>16	00000000 00000000 00000001 00000010
0xFF	00000000 00000000 00000000 11111111
(x>>16) & 0xFF	00000000 00000000 00000000 00000010

Using Shifts and Masks

- Extract the *sign bit* of a signed `int`:
  - First shift, then mask: `(x>>31) & 0x1`
    - Assuming arithmetic shift here, but works in either case
    - Need mask to clear `1`s possibly shifted in

x	00000001 00000010 00000011 00000100
x>>31	00000000 00000000 00000000 00000000
0x1	00000000 00000000 00000000 00000001
(x>>31) & 0x1	00000000 00000000 00000000 00000000

x	10000001 00000010 00000011 00000100
x>>31	11111111 11111111 11111111 11111111
0x1	00000000 00000000 00000000 00000001
(x>>31) & 0x1	00000000 00000000 00000000 00000001

Using Shifts and Masks

- Conditionals as Boolean expressions
  - For `int x`, what does `(x<<31)>>31` do?

`x=!!123`	`00000000 00000000 00000000 00000001`
`x<<31`	`10000000 00000000 00000000 00000000`
`(x<<31)>>31`	`11111111 11111111 11111111 11111111`
`!x`	`00000000 00000000 00000000 00000000`
`!x<<31`	`00000000 00000000 00000000 00000000`
`(!x<<31)>>31`	`00000000 00000000 00000000 00000000`

!!x x!=0 x==0

  - Can use in place of conditional:
    - In C: `if(x) {a=y;} else {a=z;}` equivalent to `a=x?y:z;`
    - `a=(((x<<31)>>31)&y) | (((!x<<31)>>31)&z);`

Integers

- Binary representation of integers
  - Unsigned and signed
  - Casting in C
- Consequences of finite width representations
  - Overflow, sign extension
- Shifting and arithmetic operations
- Multiplication

Multiplication

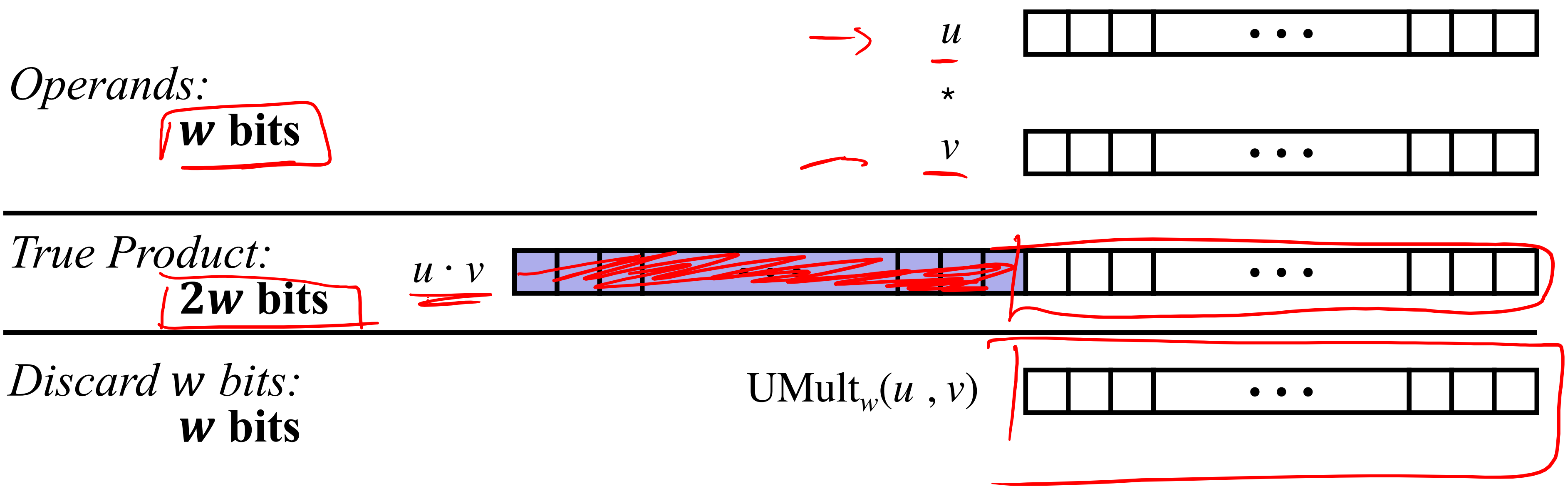
- What do you get when you multiply 9 x 9?

81

- What about 2^{10} x 2^{20} ?

$2^{10} + 2^{20} = 2^{30}$

Unsigned Multiplication in C



- Standard Multiplication Function
  - Ignores high order w bits
- Implements Modular Arithmetic
  - $\text{UMult}_w(u\ ,\ v) = u \cdot v \bmod 2^w$

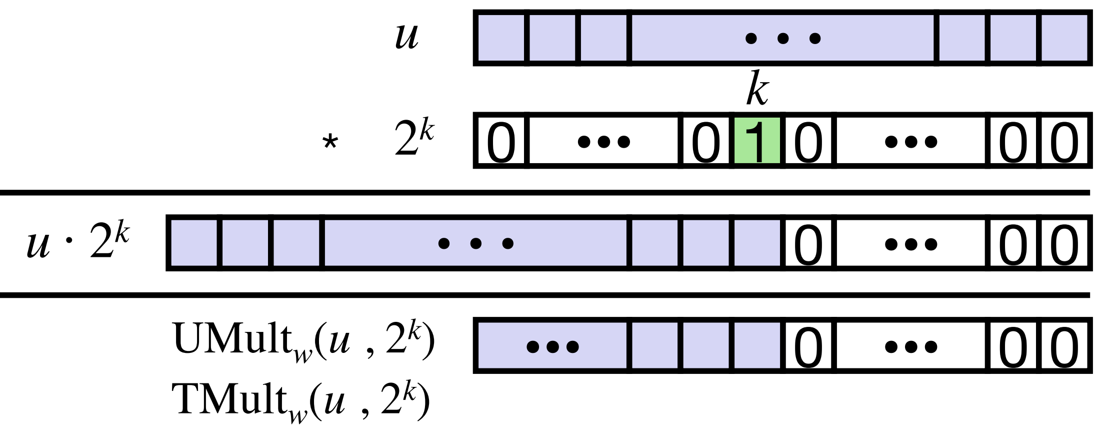
Multiplication with shift and add

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

Operands: ***w*** **bits**

True Product: ***w*** **+** ***k*** **bits**

Discard k bits: ***w*** **bits**



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

Number Representation Revisited

- What can we represent in one word?
  - Signed and Unsigned Integers
  - Characters (ASCII)
  - Addresses
- How do we encode the following:
  - Real numbers (e.g. 3.14159)
  - Very large numbers (e.g. 6.02×10^{23})
  - Very small numbers (e.g. 6.626×10^{-34})

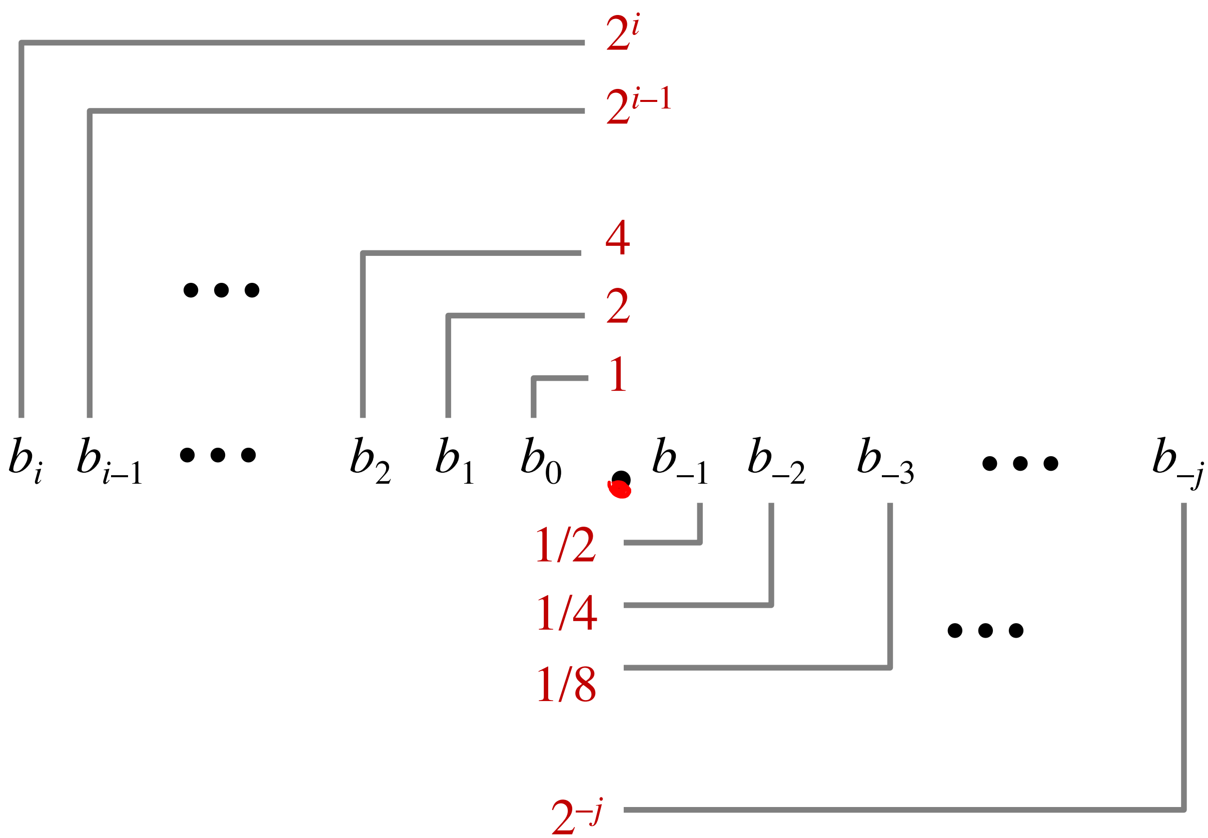
Fractional Binary Numbers

$$1\ 0\ 1\ 1\ .\ 1\ 0\ 1_{2}$$

8	4	2	1		½	¼	⅛
2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}

$$8+2+1+\tfrac{1}{2}+\tfrac{1}{8} = 11.625_{10}$$

Fractional Binary Numbers



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

Fractional Binary Numbers

- Value Binary:

5. 75

- 5.75 101.11 $4 + 1 + \frac{1}{2} + \frac{1}{4}$
- 2 and 7/8 10.111

$0.111111\ldots_2$ 1111111111111111

Fractional Binary Numbers

- Value Binary:
  - 5.75 101.11_2
  - 2 and 7/8 10.111_2
- Observations
  - Shift left = multiply by power of 2
  - Shift right = divide by power of 2
  - Numbers of the form $0.111111\ldots_2$ are just below 1.0
    - 1/2 + 1/4 + 1/8 + … + $1/2^i$ + … ➙ 1.0
    - Use notation 1.0 – ε

Limits of Representation

- Limitations:
  - Even given an arbitrary number of bits, can only **exactly** represent numbers of the form $x * 2^y$ (y can be negative)
  - Other rational numbers have repeating bit representations

Value:	Binary Representation:
• 1/3 = $0.333333\ldots_{10}$ =	$0.01010101[01]\ldots_2$
• 1/5 =	$0.001100110011[0011\ ]\ldots_2$
• 1/10 =	$0.0001100110011[0011\ ]\ldots_2$

Fixed Point Representation

- Binary point has a fixed position
  - Position = number of binary digits before and after
- Implied binary point. Two example schemes:

  #1: the binary point is between bits 2 and 3

  $b_7\ b_6\ b_5\ b_4\ b_3$ [.] $b_2\ b_1\ b_0$

  #2: the binary point is between bits 4 and 5

  $b_7\ b_6\ b_5$ [.] $b_4\ b_3\ b_2\ b_1\ b_0$
- Wherever we put the binary point, with fixed point representations there is a trade off between the amount of range and precision
- Fixed point = fixed *range* and fixed *precision*
  - range: difference between largest and smallest numbers possible
  - precision: smallest possible difference between any two numbers
- *Hard to pick how much you need of each*!
- *How do we fix this?*

“Rarely” used in practice. Not built-in.

Scientific Notation (Decimal)

60.2×10^{22}

mantissa

exponent

$6.02_{10} \times 10^{23}$

decimal point

radix (base)

- *Normalized form*: exactly one digit (non-zero) to left of decimal point
- Alternatives to representing 1/1,000,000,000
  - Normalized: 1.0×10^{-9}
  - Not normalized: $0.1\times10^{-8}, 10.0\times10^{-10}$

In binary?

Scientific Notation (Binary)

mantissa exponent

$$1.01_2 \times 2^{-1}$$

binary point radix (base)

- Computer arithmetic that supports this called floating point due to the "floating" of the binary point
  - Declare such variable in C as `float`

IEEE Floating Point

- IEEE 754 50-page
  - Established in 1985 as uniform standard for floating point arithmetic
  - Main idea: make numerically sensitive programs portable
  - Specifies two things: representation and result of floating operations
  - Now supported by all major CPUs

- Driven by numerical concerns
  - **Scientists**/numerical analysts want them to be as **real** as possible
  - **Engineers** want them to be **easy to implement** and **fast**
  - In the end:
    - Scientists mostly won out
    - Nice standards for rounding, overflow, underflow, but...
    - Hard to make fast in hardware
    - **Float operations can be an order of magnitude slower than integer ops**

Floating Point Representation

- Numerical form:

$$V_{10} = (-1)^{s} * M * 2^{E}$$

  - Sign bit ***s*** determines whether number is negative or positive
  - Significand (mantissa) ***M*** normally a fractional value in range **[1.0, 2.0)**
  - Exponent ***E*** weights value by a (possibly negative) power of two

Floating Point Representation

- Numerical form:

$$V_{10} = (-1)^{s} * M * 2^{E}$$

  - Sign bit ***s*** determines whether number is negative or positive
  - Significand (mantissa) ***M*** normally a fractional value in range **[1.0, 2.0)**
  - Exponent ***E*** weights value by a (possibly negative) power of two

- Representation in memory:
  - 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

Precisions

- Single precision: 32 bits

s	exp	frac
1 bit	8 bits	23 bits

- Double precision: 64 bits

s	exp	frac
1 bit	11 bits	52 bits

- Finite representation means not all values can be represented exactly. Some will be approximated.

Normalization and Special Values

$V = (-1)^S * M * 2^E$

s	exp	frac

0

1111

$1/2 + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} \ldots$

- "Normalized" = M has the form 1.xxxxx
  - As in scientific notation, but in binary
  - 0.011 x 2^5 and 1.1 x 2^3 represent the same number, but the latter makes better use of the available bits
  - Since we know the mantissa starts with a 1, we don't bother to store it

- How do we represent 0.0?
  Or special or undefined values like 1.0/0.0?

Normalization and Special Values

$V = (-1)^{S} * M * 2^{E}$

s	exp	frac

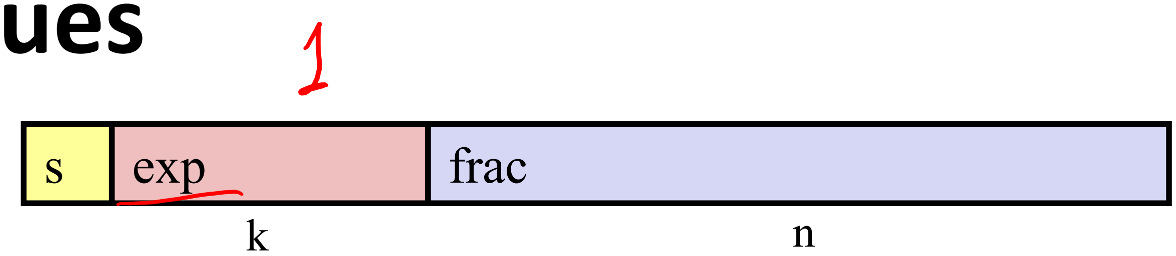
- "Normalized" = M has the form 1.xxxxx
  - As in scientific notation, but in binary
  - 0.011×2^5 and 1.1×2^3 represent the same number, but the latter makes better use of the available bits
  - Since we know the mantissa starts with a 1, we don't bother to store it.
- Special values ("denormalized"):
  - **Zero (0):** exp == 00...0, frac == 00...0
  - **+∞, -∞:** exp == 11...1, frac == 00...0

    1.0/0.0 = -1.0/-0.0 = +∞, 1.0/-0.0 = -1.0/0.0 = -∞
  - **NaN** ("Not a Number"): exp == 11...1 frac != 00...0

    Results from operations with undefined result: sqrt(-1), ∞–∞, ∞•0, ...
  - **Note**: **exp=11...1 and exp=00...0 are reserved**, limiting exp range...

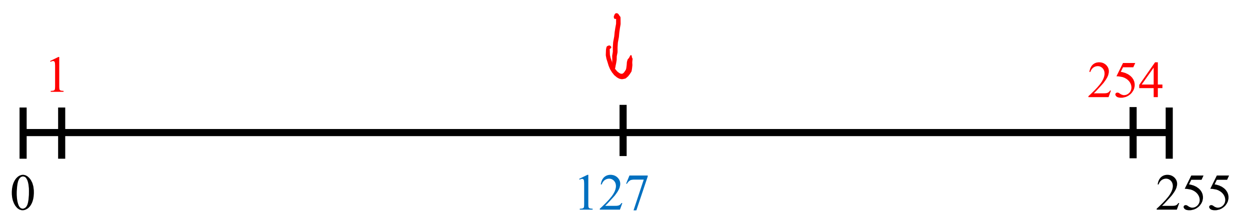
Normalized Values

$V = (-1)^{S} * M * 2^{E}$



- Condition: exp ≠ 000…0 and exp ≠ 111…1
- Exponent coded as *biased* value: *E* = *exp* - *Bias*
  - exp is an *unsigned* value ranging from 1 to 2^k-2 (k == # bits in exp)
  - *Bias* = 2^{k-1} - 1
    - Single precision: 127 (so *exp*: 1…254, *E*: -126…127)
    - Double precision: 1023 (so *exp*: 1…2046, *E*: -1022…1023)
  - These enable negative values for E, for representing very small values

2^{-126}



Normalized Values

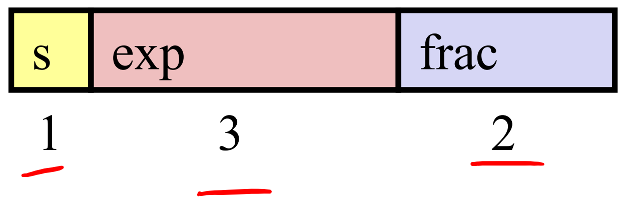
$V = (-1)^{s} * M * 2^{E}$

s	exp	frac
	k	n

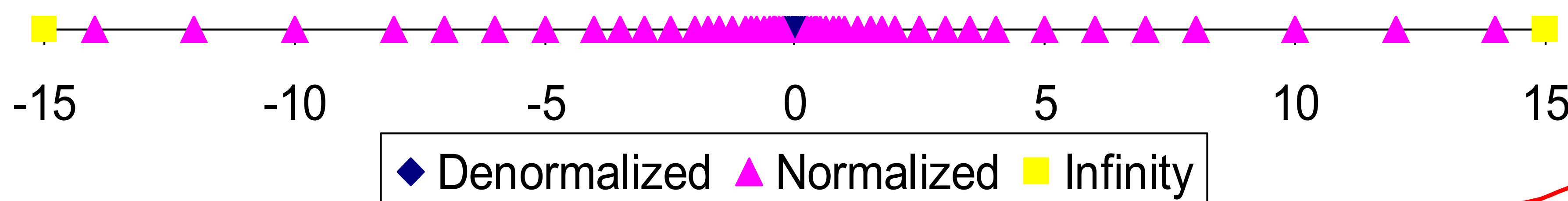
- Condition: exp ≠ 000…0 and exp ≠ 111…1
- Exponent coded as *biased* value: *E* = *exp - Bias*
  - exp is an *unsigned* value ranging from 1 to 2^k-2 (k == # bits in exp)
  - *Bias* = 2^{k-1} - 1
    - Single precision: 127 (so *exp*: 1…254, *E*: -126…127)
    - Double precision: 1023 (so *exp*: 1…2046, *E*: -1022…1023)
  - These enable negative values for E, for representing very small values
    - Could have encoded with 2's complement or sign-and-magnitude
    - This just made it easier for HW to do float-exponent operations
- Mantissa coded with implied leading 1: *M* = $1.xxx\ldots x_2$
  - xxx…x: the n bits of frac
  - Minimum when 000…0 (*M* = 1.0)
  - Maximum when 111…1 (*M* = 2.0 – ε)
  - Get extra leading bit for "free"

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.



Floating Point Operations

- Unlike the representation for integers, the representation for floating-point numbers is <u>not exact</u>
- We have to know how to round from the real value

Floating Point Operations: Basic Idea

$V = (-1)^{S} * M * 2^{E}$

s	exp	frac

- $x +_f y = \text{Round}(x + y)$

- $x *_f y = \text{Round}(x * y)$

- Basic idea for floating point operations:
  - First, compute the exact result
  - Then, *round* the result to make it fit into desired precision:
    - Possibly overflow if exponent too large
    - Possibly drop least-significant bits of mantissa to fit into frac

Floating Point Addition

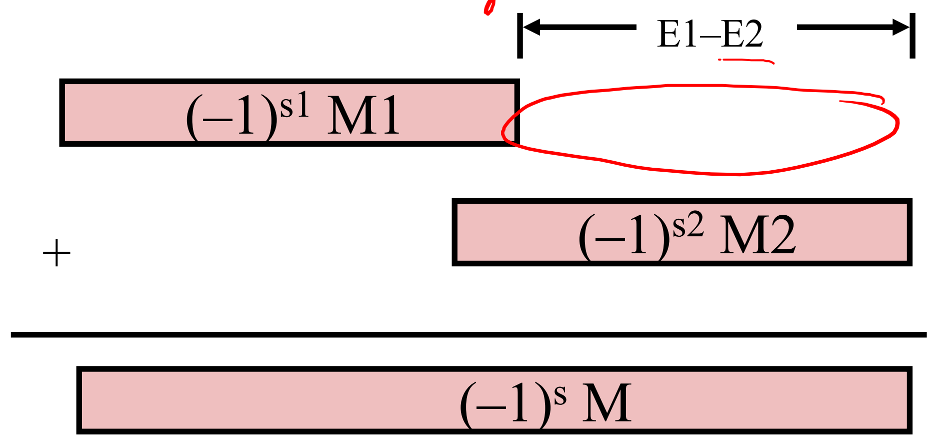
$(-1)^{s1} * M1 * 2^{E1} + (-1)^{s2} * M2 * 2^{E2}$

Assume *E1* > *E2*

$$\begin{array}{r} 1.010 * 2^{2} \\ +\ 1.000 * 2^{-1} \\ \hline ??? \end{array} \qquad \begin{array}{r} 1.0100 * 2^{2} \\ +\ \mathbf{0.0001} * \mathbf{2^{2}} \\ \hline 1.0101 * 2^{2} \end{array}$$

exp = exp + 3

M = M/8



- Exact Result: $(-1)^{s} * M * 2^{E}$
  - Sign s, mantissa 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 Multiplication

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

- Exact Result: $(-1)^{s} * M * 2^{E}$
  - Sign s: s1 ^ s2
  - Mantissa M: M1 * 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

Rounding modes

- Possible rounding modes (illustrated with dollar rounding):

	\$1.40	\$1.60	\$1.50	\$2.50	–\$1.50
Round-toward-zero	\$1	\$1	\$1	\$2	–\$1
Round-down (-∞)	\$1	\$1	\$1	\$2	–\$2
Round-up (+∞)	\$2	\$2	\$2	\$3	–\$1
Round-to-nearest	\$1	\$2	*??*	*??*	*??*
Round-to-even	\$1	\$2	\$2	\$2	–\$2

- Round-to-even avoids statistical bias in repeated rounding.
  - Rounds up about half the time, down about half the time.
  - Default rounding mode for IEEE floating-point

Mathematical Properties of FP Operations

- Exponent overflow yields +∞ or -∞
- Floats with value +∞, -∞, and NaN can be used in operations
  - Result usually still +∞, -∞, or NaN; sometimes intuitive, sometimes not
- Floating point ops do not work like real math, due to rounding!
  - Not associative: (3.14 + 1e100) – 1e100 != 3.14 + (1e100 – 1e100)
  - Not distributive: 100 * (0.1 + 0.2) != 100 * 0.1 + 100 * 0.2

    30.000000000000003553 30
  - Not cumulative
    - Repeatedly adding a very small number to a large one may do nothing

Floating Point in C

!!!

- C offers two (well, 3) levels of precision

  float 1.0f single precision (32-bit)

  double 1.0 double precision (64-bit)

  long double 1.0L *(double double, quadruple, or "extended")* precision (64-128 bits)

- #include <math.h> to get INFINITY and NAN constants
- Equality (==) comparisons between floating point numbers are tricky, and often return unexpected results
  - Just avoid them!

Floating Point in C

!!!

- Conversions between data types:
  - Casting between int, float, and double **changes** the bit representation.
  - int → float
    - May be rounded (not enough bits in mantissa: 23)
    - Overflow impossible
  - int → double or float → double
    - Exact conversion (32-bit ints; 52-bit frac + 1-bit sign)
  - long → double
    - Rounded or exact, depending on word size (64-bit → 52 bit mantissa ⇒ round)
  - double or float → int
    - Truncates fractional part (rounded toward zero)
      - E.g. 1.999 → 1, -1.99 → -1
    - "Not defined" when out of range or NaN: generally sets to Tmin (even if the value is a very big positive)

Number Representation Really Matters

- 1991: Patriot missile targeting error
  - clock skew due to conversion from integer to floating point
- 1996: Ariane 5 rocket exploded ($1 billion)
  - overflow converting 64-bit floating point to 16-bit integer
- 2000: Y2K problem
  - limited (decimal) representation: overflow, wrap-around
- 2038: Unix epoch rollover
  - Unix epoch = seconds since 12am, January 1, 1970
  - signed 32-bit integer representation rolls over to TMin in 2038
- other related bugs
  - 1982: Vancouver Stock Exchange 10% error in less than 2 years
  - 1994: Intel Pentium FDIV (floating point division) HW bug ($475 million)
  - 1997: USS Yorktown "smart" warship stranded: divide by zero
  - 1998: Mars Climate Orbiter crashed: unit mismatch ($193 million)

Floating Point and the Programmer

```
#include <stdio.h>

int main(int argc, char* argv[]) {

  float f1 = 1.0;
  float f2 = 0.0;
  int i;
  for (i = 0; i < 10; i++) {
    f2 += 1.0/10.0;
  }

  printf("0x%08x  0x%08x\n", *(int*)&f1, *(int*)&f2);
  printf("f1 = %10.8f\n", f1);
  printf("f2 = %10.8f\n\n", f2);

  f1 = 1E30;
  f2 = 1E-30;
  float f3 = f1 + f2;
  printf("f1 == f3? %s\n", f1 == f3 ? "yes" : "no" );

  return 0;
}
```

```
$ ./a.out
0x3f800000  0x3f800001
f1 = 1.000000000
f2 = 1.000000119

f1 == f3? yes
```

Q&A: THE PENTIUM FDIV BUG

(floating point division)

Q: What do you get when you cross a Pentium PC with a research grant?
A: A mad scientist.

Q: Complete the following word analogy:
Add is to Subtract as Multiply is to:

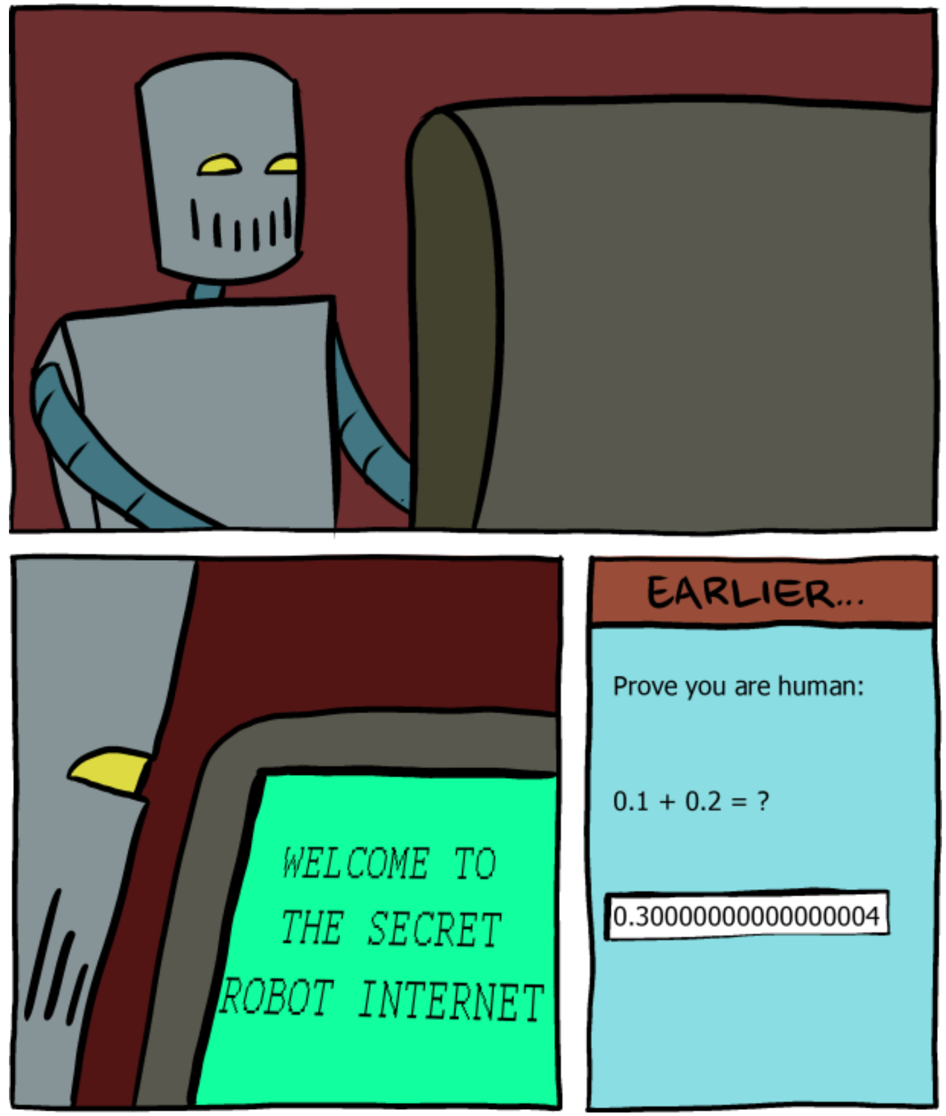
1) Divide
2) ROUND
3) RANDOM
4) On a Pentium, all of the above

A: Number 4.

Q: What algorithm did Intel use in the Pentium's floating point divider?
A: "Life is like a box of chocolates."
(Source: F. Gump of Intel)

Q: According to Intel, the Pentium conforms to the IEEE standards 754 and 854 for floating point arithmetic. If you fly in aircraft designed using a Pentium, what is the correct pronunciation of "IEEE"?
A: Aaaaaaaiiiiiiiiieeeeeeeeeeeee!

Source: http://www.columbia.edu/~sss31/rainbow/pentium.jokes.html



http://www.smbc-comics.com/?id=2999

Summary

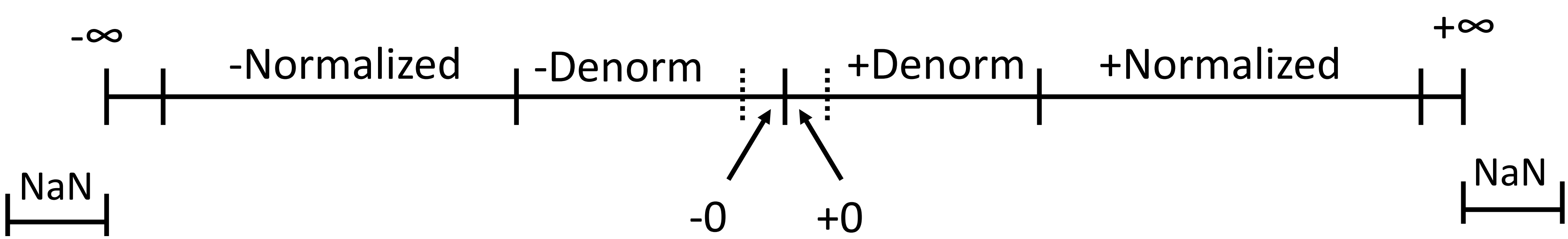
- As with integers, floats suffer from the fixed number of bits available to represent them
  - Can get overflow/underflow, just like ints
  - Some "simple fractions" have no exact representation (e.g., 0.2)
  - Can also lose precision, unlike ints
    - "Every operation gets a slightly wrong result"
- Mathematically equivalent ways of writing an expression may compute different results
  - Violates associativity/distributivity
- Never test floating point values for equality!
- Careful when converting between ints and floats!

BONUS SLIDES

More details for the curious. These slides expand on material covered today

- Tiny Floating Point Example
- Distribution of Values

Visualization: Floating Point Encodings



Tiny Floating Point Example

s	exp	frac
1	4	3

- 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

Dynamic Range (Positive Only)

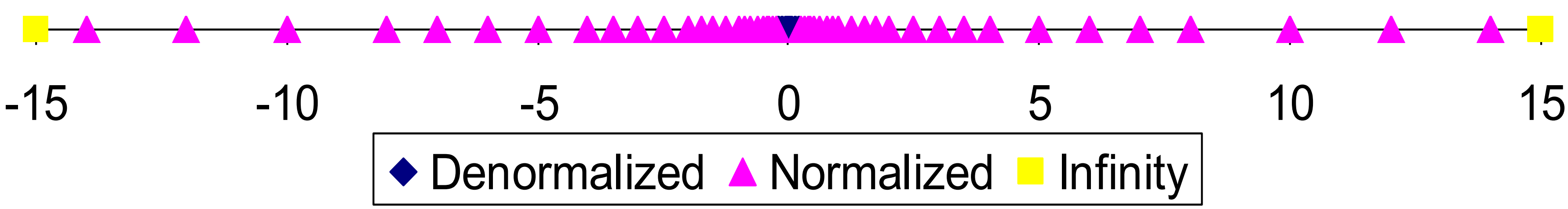
	s exp frac	E	Value	
Denormalized numbers	0 0000 000	-6	0	
	0 0000 001	-6	1/8*1/64 = 1/512	closest to zero
	0 0000 010	-6	2/8*1/64 = 2/512	
	…			
	0 0000 110	-6	6/8*1/64 = 6/512	
	0 0000 111	-6	7/8*1/64 = 7/512	largest denorm
Normalized numbers	0 0001 000	-6	8/8*1/64 = 8/512	smallest norm
	0 0001 001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

Distribution of Values

- 6-bit IEEE-like format
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is 23-1-1 = 3

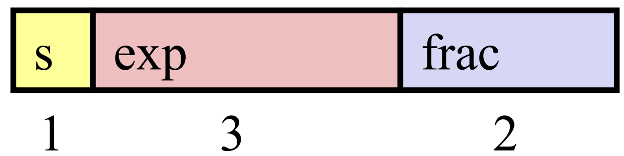
s	exp	frac
1	3	2

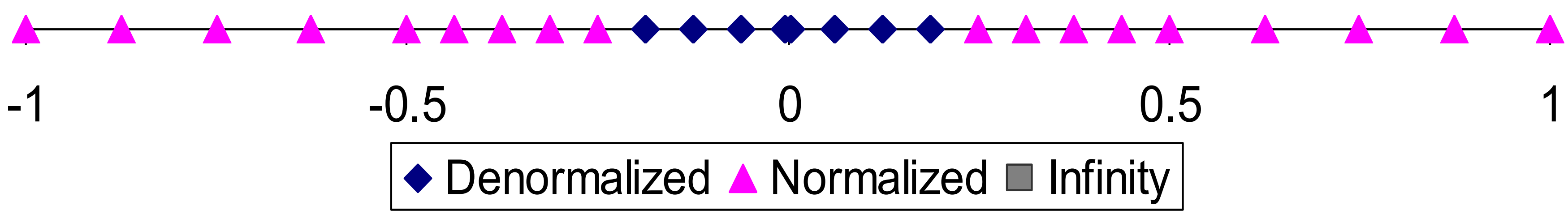
- Notice how the distribution gets denser toward zero.



Distribution of Values (close-up view)

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





Interesting Numbers

{single,double}

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

Special Properties of Encoding

- Floating point zero (0^+) exactly the same bits as integer zero
  - All bits = 0

- Can (Almost) Use Unsigned Integer Comparison
  - Must first compare sign bits
  - Must consider $0^- = 0^+ = 0$
  - NaNs problematic
    - Will be greater than any other values
    - What should comparison yield?
  - Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity