http://www.youtube.com/watch?v=-8gh6-WHGRs

Moore's "Law"

- # transistors per chip would double every year for next 10 years
  - Turns out, it's more like every 18 month
- Storage also follows an exponential curve
- What critical improvements are sub-exponential (linear)?

?’s about Lab 3?

1:

E = Exp – bias , bias = 2^(e-1) - 1

M = 1.xxx…x

2: exp == 000…0

E = 1-bias

M = 0.xxx…x

3: exp == 111…1

frac == 000…0

+/- inf

frac == 111…1

NaN

- Given a n-bit fraction, what's the smallest positive integer than cannot be represented exactly (because it would require n+1 bits in the fraction)? Assume exponent field size k is large enough that range of representable exponents does not provide a limitation for this problem
- What would this be when n=23?

Camera slowly pans left, taking in the cluttered office, stopping on Matt. He's sitting down, trying to work through a floating point word problem for class. With a concentrated look, he takes a large gulp from his double espresso. Camera does a tight zoom to his mouth, as he quickly keeps himself from spitting it back out onto the monitor and keyboard.

Nope, apparently he did not successfully remove all the dish soap from the glass before making coffee this morning.

Smart enough for a PhD, but outwitted by doing dishes.

Consider a 1-bit fraction

You can represent 1.1

You cannot represent 1.01

If you wanted to find the *smallest int* you couldn't represent:

1 = 1 * 2^0 (frac = 0)

2 = 1 * 2^1 (frac = 0)

3 = 1.1 * 2^1 (frac = 1)

4 = 1 * 2^2 (frac=0)

5 = 1.01 * 2^2 (frac = 01)

6 = 1.1 * 2^2 (frac = 10)

2^(n+1) + 1 = 5 for n=1

Consider an 2-bit fraction

You can represent 1.01

You cannot represent 1.001

If you wanted to turn these into the *smallest ints* possible:

1 = 1 * 2^0 (frac = 00)

2 = 1 * 2^1 (frac = 00)

3 = 1.1 * 2^1 (frac = 10)

4 = 1 * 2^2 (frac=00)

5 = 1.01 * 2^2 (frac = 01)

6 = 1.1 * 2^2 (frac = 10)

7 = 1.11 * 2^2 (frac = 11)

8 = 1 * 2^3 (frac = 00)

9 = 1.001 * 2^3 (frac = 001)

10 = 1.01 * 2^3 (frac = 01)

2^(n+1) + 1 = 9, for n=2

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

In C

- fenv.h defines constants which you can use to refer to the various rounding modes. Each one will be defined if and only if the FPU supports the corresponding rounding mode.

- FE_TONEAREST
  - Round to nearest.
- FE_UPWARD
  - Round toward +inf
- FE_DOWNWARD
  - Round toward -inf
- FE_TOWARDZERO
  - Round toward zero

```
int fesetrount (int round)
```

2.84

k-bit exponent, n-bit fraction

A) V=7.0

E = 2, M=1.11

frac = 1100…0

Exp = 1000…01

bias = 2^(e-1)-1 = 2^(k-1)-1 = 0111…1

To represent E=0, need to have Exp=bias: 0111…1

To represent E=2, need to have Exp=bias+10 = 1000…1

Floating Point Operations: Basic Idea

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

- $x \times_f y = \text{Round}(x \times y)$

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

Special Properties of Encoding

- **FP 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

FP Comparisons

- testIneq.c
- isnan()

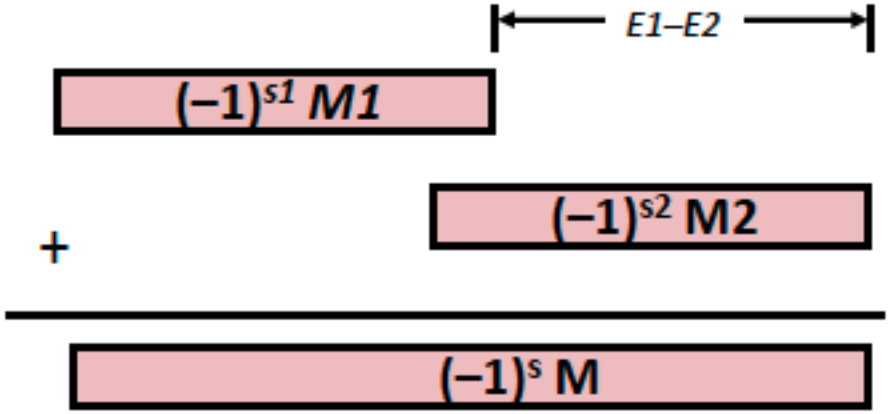
- FP Exceptions
  - http://www.gnu.org/software/libc/manual/html_node/FP-Exceptions.html#FP-Exceptions

- FP Classes
  - http://www.gnu.org/software/libc/manual/html_node/Floating-Point-Classes.html#Floating-Point-Classes

Floating Point Addition

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

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

Addition

- Commutative
- Not associative

  Single precision float:

  (3.14 + 1e10) - 1e10 = 0.0

  3.14 + (1e10 - 1e10) = 3.14

- Most inverses hold, except:

  $+\infty - \infty = \text{NaN}$

  NaN + x = NaN

Mathematical Properties of FP Add

- **Compare to those of Abelian Group**
  - Closed under addition? ***Yes***
    - But may generate infinity or NaN
  - Commutative? ***Yes***
  - Associative? ***No***
    - Overflow and inexactness of rounding
  - 0 is additive identity? ***Yes***
  - Every element has additive inverse ***Almost***
    - Except for infinities & NaNs
- **Monotonicity**
  - $a \geq b \Rightarrow a+c \geq b+c$? ***Almost***
    - Except for infinities & NaNs

FP Multiplication

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

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

Multiplication

- Not Associative

  1e20 * (1e20 * 1e-20) = 1e20

  (1e20 * 1e20) * 1e-20 = +∞

- Doesn't distribute over addition

  1e20 * (1e20 - 1e20) = 0.0

  (1e20 * 1e20) – (1e20 * 1e20) = NaN

Mathematical Properties of FP Mult

- **Compare to Commutative Ring**
  - Closed under multiplication? ***Yes***
    - But may generate infinity or NaN
  - Multiplication Commutative? ***Yes***
  - Multiplication is Associative? ***No***
    - Possibility of overflow, inexactness of rounding
  - 1 is multiplicative identity? ***Yes***
  - Multiplication distributes over addition? ***No***
    - Possibility of overflow, inexactness of rounding

- **Monotonicity**
  - $a \geq b \ \& \ c \geq 0 \Rightarrow a * c \geq b * c$? ***Almost***
    - Except for infinities & NaNs

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

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 no inf
Nor –Nan nor -inf

```
1.  x  ==  (int)(double) x
2.  x  ==  (int)(float) x
3.  f  ==  (float)(double) f
4.  d  ==  (double)(float) d
5.  f  ==  -(-f);
6.  1.0/2  ==  1/2.0
7.  d×d  >=  0.0
8.  (f+d)-f  ==  d
```

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

```
1.  x  ==  (int)(double) x      T
2.  x  ==  (int)(float) x       F
3.  f  ==  (float)(double) f    T
4.  d  ==  (double)(float) d    F
5.  f  ==  -(-f);               T
6.  1.0/2  ==  1/2.0            T
7.  d×d  >=  0.0                T
8.  (f+d)-f  ==  d              F
```

Before going on

- ASCII (1967) vs. Unicode (1991)?
  - 8 vs. 16 bits

- Error checking / correcting
  - CRC (cyclic redundancy check)
    - Send data and remainder of data divided by something
    - Does the calculation still check out by the receiver?
  - Parity Code: detect 1 bit error, correct 0
  - Hamming Codes: detect and correct
    - ECC memory
    - More bits used for overhead, better detection/correction

Compiling Into Assembly

C Code

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

Generated IA32 Assembly

```
sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

Obtain with command

```
gcc -O -S code.c
```

Produces file `code.s`

Some compilers use single instruction "`leave`"

Assembly Characteristics: Data Types

- "Integer" data of 1, 2, or 4 bytes
  - Data values
  - Addresses (untyped pointers)

- Floating point data of 4, 8, or 10 bytes

- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory

Assembly Characteristics: Operations

- Perform arithmetic function on register or memory data

- Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory

- Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches