Floating-point operations I

- The science of floating-point arithmetics
- IEEE standard
- Reference

  What every computer scientist should know about floating-point arithmetic, ACM computing survey, 1991

Why learn more about floating-point operations I

Example:

- A one-variable problem

$$\min_x f(x)$$
$$x \geq 0$$

- In your program, should you set an upper bound of x?
- x in your program may be wrongly increased to ∞

Why learn more about floating-point operations II

- What is the largest representable number in the computer ?
- Is there anything called infinity ?

Example:

- A ten-variable problem

$$\min f(x)$$
$$0 \leq x_i, i = 1, \ldots, 10$$

Why learn more about floating-point operations III

- After the problem is solved, want to know how many are zeros ?
- Should you use

```
for (i=0; i < 10; i++)
    if (x[i] == 0) count++ ;
```

- People said: don't do floating-point comparisons

```
epsilon = 1.0e-12 ;
for (i=0; i < 10; i++)
    if (x[i] <= epsilon) count++ ;
```

How do you choose ϵ ?

Why learn more about floating-point operations IV

- Is this true ?

Floating-point Formats I

- We know float (single): 4 bytes, double: 8 bytes
  Why ?
- A floating-point system
  base β, precision p, significand (mantissa) $d.d\ldots d$
- Example

$$\begin{aligned} 0.1 &= 1.00 \times 10^{-1} && (\beta = 10, p = 3) \\ &\approx 1.1001 \times 2^{-4} && (\beta = 2, p = 5) \end{aligned}$$

  exponent: -1 and -4
- Largest exponent $e_{\max}$, smallest $e_{\min}$

Floating-point Formats II

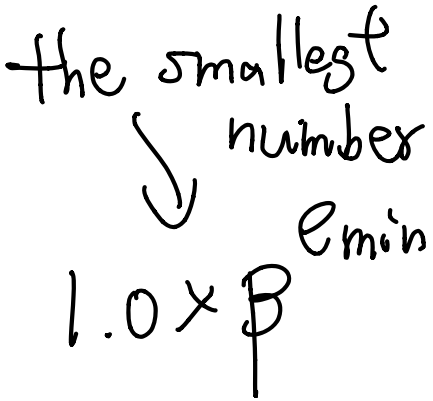
- β^p possible significands, $e_{\max} - e_{\min} + 1$ possible exponents

$$\lceil \log_2(e_{\max} - e_{\min} + 1) \rceil + \lceil \log_2(\beta^p) \rceil + 1$$

bits for storing a number
1 bit for $\pm$

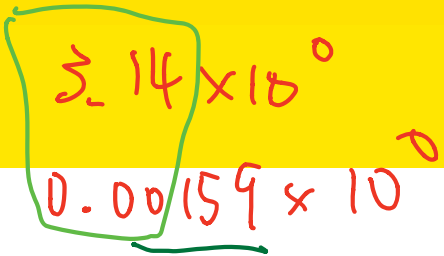
- But the practical setting is more complicated
  See the discussion of IEEE standard later
- Normalized: 1.00×10^{-1} (yes), 0.01×10^{1} (no)
- Now most used normalized representation
  $\Rightarrow$ cannot represent zero

Floating-point Formats III



- A natural way for 0: $1.0 \times \beta^{e_{\text{min}}-1}$ preserve the ordering
- Will use $p = 3, \beta = 10$ for most later explanation

Relative Errors and Ulps I

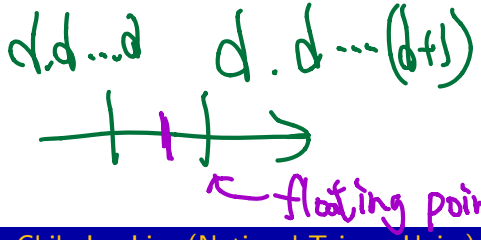


- When $\beta = 10, p = 3$, 3.14159 represented as 3.14×10^0

  $\Rightarrow$ error $= 0.00159 = 0.159 \times 10^{-2}$, i.e. 0.159 units in the last place

  10^{-2}: unit of the last place
- ulps: unit in the last place
- relative error $0.00159/3.14159 \approx 0.0005$
- For a number $d.d\ldots d \times \beta^e$, the largest error is

$$0.\underbrace{0\ldots0}_{p-1}\beta' \times \beta^e, \beta' = \beta/2$$



Relative Errors and Ulps II

- Error $= \frac{\beta}{2} \times \beta^{-p} \times \beta^{e}$ ex. $1\times10^5 \le\ ? < 9\times10^5$

$$1 \times \beta^e \le \text{ original value } < \beta \times \beta^e$$

relative error between

$$\frac{\frac{\beta}{2} \times \beta^{-p} \times \beta^e}{\beta^e} \quad \text{and} \quad \frac{\frac{\beta}{2} \times \beta^{-p} \times \beta^e}{\beta^{e+1}}$$

so

$$\text{relative error } \le \frac{\beta}{2}\beta^{-p} \tag{1}$$

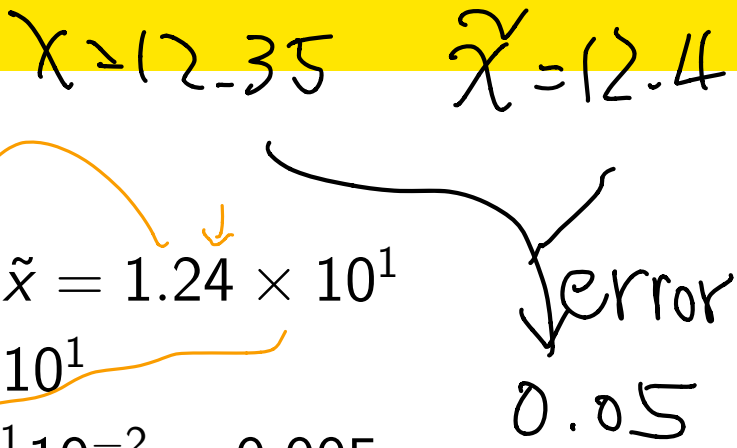
- $\frac{\beta}{2}\beta^{-p} = \beta^{-p+1}/2$: machine epsilon

Relative Errors and Ulps III

The bound in (1)

- When a number is rounded to the closest, relative error bounded by ϵ

ulps and ϵ I



- $p = 3,\ \beta = 10$
- Example: $x = 12.35 \Rightarrow \tilde{x} = 1.24 \times 10^1$
  error $= 0.05 = 0.005 \times 10^1$
- ulps $= 0.01 \times 10^1$, $\epsilon = \frac{1}{2}10^{-2} = 0.005$
- error 0.5 ulps
  relative error $0.05/12.35 \approx 0.004 = 0.8\epsilon$
- $8x = 98.8, 8\tilde{x} = 9.92 \times 10^1$
  error $=$ 4.0 ulps
  relative error $= 0.4/98.8 = 0.8\epsilon$.
- ulps and ϵ may be used interchangeably

Guard Digits I

- $p = 3, \beta = 10$
- Calculate $2.15 \times 10^{12} - 1.25 \times 10^{-5}$:
- Compute and then round

$$\begin{aligned} x &= 2.15 \times 10^{12} \\ y &= 0.00000000000000125 \times 10^{12} \\ x - y &= 2.14999999999999875 \times 10^{12} \end{aligned}$$

round to 2.15×10^{12}

Here we assume that computation is exactly done

Guard Digits II

- Round and then compute

$$\begin{aligned} x &= 2.15 \times 10^{12} \\ y &= 0.00 \times 10^{12} \\ x - y &= 2.15 \times 10^{12} \end{aligned}$$

Answer is the same

Reasonable as $x \approx x - y$

- Another example: $10.1 - 9.93 = 0.17$

ulps $= 10^{-2} \times 10^{-1}$

1.70×10^{-1}

2.00×10^{-1}

Guard Digits III

- Round and then compute

$$10.1 - 9.93 = 1.01 \times 10^1 - 0.99 \times 10^1 = 0.02 \times 10^1$$
$$= 2.00 \times 10^{-1}$$
$$\text{error} = 2.00 \times 10^{-1} - 0.17 = 0.03$$
$$\text{ulps} = 0.01 \times 10^{-1} = 10^{-3}$$
$$\text{error} = 0.03 = 30\text{ulps}$$

result

Correct

$\text{ulps} = \beta^{-(p-1)} \cdot \beta^{e}$

Relative error

$$= 0.03/0.17 = 3/17$$

.

Guard Digits IV

The error is quite large

- Compute and round

$$10.1 - 9.93 = 0.17 = 1.7 \times 10^{-1}$$

error $= 0$

The problem: cannot compute and then round

Guard Digits V

- How big can the error be? (if round and then compute)

Theorem

Using p digits with base β for $x - y$, the relative error can be as large as $\beta - 1$

Proof:

$x = 1.0\ldots0, y = 0.\underbrace{\eta\ldots\eta}_{p \text{ digits}}, \eta = \beta - 1$

Correct solution $x - y = \beta^{-p}$

Computed solution $= 1.0\ldots0 - 0.\underbrace{\eta\ldots\eta}_{p-1 \text{ digits}} = \beta^{-p+1}$

Guard Digits VI

Relative error

$$\frac{|\beta^{-p} - \beta^{-p+1}|}{\beta^{-p}} = \beta - 1$$

- Example: $p = 3$, $\beta = 10$

  $x = 1.00, y = 0.999, x - y = 0.001 = 10^{-3}$

  Computed solution $= 1.00 \times 10^0 - 0.99 \times 10^0$
  $= 0.01 \times 10^0 = 0.01$

  Relative error

$$\frac{|0.01 - 0.001|}{0.001} = 9$$

Super large

Guard Digits VII

Such large errors occur if x and y are close

- Single guard digit

p increased by 1 in the device for addition and subtraction

round and then compute

$$1.010 \times 10^1 - 0.993 \times 10^1 = 0.017 \times 10^1$$

Note $0.017 \times 10^1 = 1.70 \times 10^{-1}$ can be stored as $p = 3$

- That is, one additional digit in the process of subtraction. All values are still stored using $p = 3$

Guard Digits VIII

- So in the device for subtraction, we should put additional digits
- Another example:

$$
\begin{aligned}
& 110 - 8.59 \\
=& 1.100 \times 10^2 - 0.085 \times 10^2 \\
=& 1.015 \times 10^2 \approx 1.02 \times 10^2
\end{aligned}
$$

Correct answer 101.41

Relative error around 0.006

Guard Digits IX

$$\epsilon = \frac{1}{2}\beta^{-p+1} = \frac{1}{2}10^{-2} = 0.005$$

Theorem

Using $p+1$ digits for $x-y \Rightarrow$ relative rounding error $< 2\epsilon$ (ϵ: machine epsilon)

Proof:

- Assume $x > y$
- Assume $x = x_0.x_1 \cdots x_{p-1} \times \beta^0$

  The proof is similar if it's not β^0
- If $y = y_0.y_1 \cdots y_{p-1}$ no error

Guard Digits X

- If $y = 0.y_1 \cdots y_p \Rightarrow$ 1 guard digit, exact $x - y$ rounded to a closest number $\Rightarrow$ relative error $\leq \epsilon$
- In general $y = 0.0 \cdots 0 y_{k+1} \cdots y_{k+p}$

  $\bar{y}$: y truncated to $p+1$ digits

$$\begin{aligned} |y - \bar{y}| &< (\beta - 1)(\beta^{-p-1} + \beta^{-p-2} + \cdots + \beta^{-p-k}) \\ &\leq \beta^{-p} \end{aligned} \tag{2}$$

$-p-1$: we have $p+1$ digits now

(Think about $p = 3, \beta = 10$, first digit truncated $\leq 9 \times 0.0001 = 9 \times 10^{-4}$)

Guard Digits XI

After y is truncated, we need to calculate

$$x - \bar{y}$$

It's rounded to

$$x - \bar{y} + \delta$$

$$|\delta| \leq (\beta/2)\beta^{-p} = \epsilon$$

The inequality comes from rounding a number of $p + 1$ digits

$$0.\underbrace{0\ldots0}_{p \text{ digits}}(\beta/2)\ldots$$

Guard Digits XII

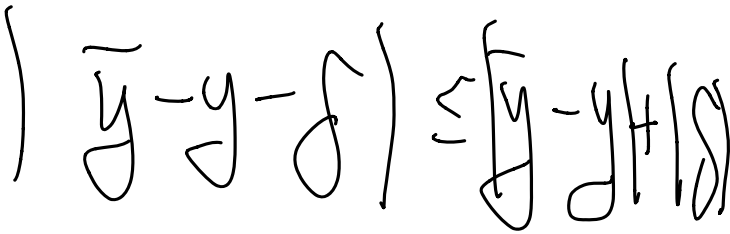
max error

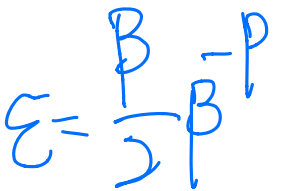
error: $(x - y) - (x - \bar{y} + \delta) = \bar{y} - y - \delta$

Guard Digits XIII

case 1: if $x - y \geq 1$,

$$\begin{aligned}
&\text{relative error} \\
&= \frac{|\bar{y} - y - \delta|}{x - y} \leq \frac{|\bar{y} - y - \delta|}{1} \\
&\leq \beta^{-p}[(\beta - 1)(\beta^{-1} + \cdots + \beta^{-k}) + \beta/2] \\
&= \beta^{-p}[(\beta - 1)\beta^{-k}(1 + \cdots + \beta^{k-1}) + \beta/2] \\
&= \beta^{-p}[(\beta - 1)\beta^{-k}\frac{1 - \beta^k}{1 - \beta} + \beta/2] \\
&= \beta^{-p}[(1 - \beta^{-k}) + \beta/2] \\
&< \beta^{-p}(1 + \beta/2) \leq 2\epsilon
\end{aligned}$$





Guard Digits XIV

$x - y < 1 \rightarrow$ or $x - \bar{y} \le 1$, $x - \bar{y} > 1$

case 2: $x - \bar{y} \le 1$: enough digits $\delta = 0$

the smallest $x - y$: (smallest x - largest y)

$$1.0 - 0.0 \ldots 0\rho \ldots \rho > (\beta - 1)(\beta^{-1} + \cdots + \beta^{-k})$$

k zeros, p ρ's, $\rho = \beta - 1$, from (2) the relative error

$$\le \frac{|\bar{y} - y - \delta|}{(\beta - 1)(\beta^{-1} + \cdots + \beta^{-k})}$$

$$< \frac{(\beta - 1)\beta^{-p}(\beta^{-1} + \cdots + \beta^{-k})}{(\beta - 1)(\beta^{-1} + \cdots + \beta^{-k})} = \beta^{-p} < 2\epsilon$$

$\sum_{i=0}^{2\beta} \beta^{-p} = 1$

case 3: $x - y < 1$ but $x - \bar{y} > 1$

we already prove: $|x-y| \geq 1$

We will then prove $|x-y| \leq 1$

$|x-y|<1$ → $x-\bar{y} \leq 1$ → Case 2

$x-\bar{y} > 1$ → Case 3

Case 2: $|x-y|<1$ and $x-\bar{y} \leq 1$

After $y \to \bar{y}$, $x-\bar{y} \Rightarrow \delta = 0$

$\bar{y}$: p+1 digits

p+1 digit

$x_0.x_1 x_2 \cdots x_{p-1} 0$

$1.0000\ldots$

$?.? \cdots ? \;?$

p+1 digits

if $x-\bar{y}=1 \Rightarrow f=0$

if $x-\bar{y}<1 \Rightarrow$ we can normalize this number so that $f=0$

ex. $0.9999 \Rightarrow 9.999$

Guard Digits XV

We show that this situation is impossible

If $x - \bar{y} = 1.\underbrace{0 \cdots 1}_{p} \Rightarrow x - y \geq 1$: a contradiction

Why $x - y$ must be ≥ 1: $|y - \bar{y}| < \beta^{-p} = 0.\underbrace{0 \cdots 1}_{p}$

- Conclusion: adding some guard digits can reduce the error

  Especially when subtracting two nearby numbers
- Cost: the adder is one bit wider (cheap)

  Most modern computers have guard digits

Cancellation I

- Catastrophic cancellation and benign cancellation
- Catastrophic cancellation :

  $b = 3.34, a = 1.22, c = 2.28$, $b^2 - 4ac = 0.0292$

  $b^2 \approx 11.2, 4ac \approx 11.1 \Rightarrow$ answer $= 0.1$

  error $= 0.1 - 0.0292 = 0.0708$

  answer $= 0.0292 = 2.92 \times 10^{-2}$

  ulps $= 0.01 \times 10^{-2} = 10^{-4}$

  $0.0708 \approx 708$ ulps → huge!
- Happens when subtracting two close numbers

Cancellation II

- Benign cancellation: subtracting exactly known numbers, by guard digits
- $\Rightarrow$ small relative error
- In the example, b^2 and $4ac$ already contain errors

Avoid Catastrophic Cancellation I

- By rearranging the formula
- Example

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a} \tag{3}$$

- If $b^2 \gg 4ac \Rightarrow$ no cancellation when calculating $b^2 - 4ac$ and $\sqrt{b^2 - 4ac} \approx |b|$

  Then $-b + \sqrt{b^2 - 4ac}$ has a catastrophic cancellation if $b > 0$

Avoid Catastrophic Cancellation II

- Multiplying $-b - \sqrt{b^2 - 4ac}$, if $b > 0$

$$\frac{2c}{-b - \sqrt{b^2 - 4ac}} \tag{4}$$

- Use (3) if $b < 0$, (4) if $b > 0$
- Difficult to remove all catastrophic cancellations, but possible to remove most by reformulations
- Another example: $x^2 - y^2$

  Assume $x \approx y$

  $(x - y)(x + y)$ is better than $x^2 - y^2$

Avoid Catastrophic Cancellation III

x^2, y^2 may be rounded $\Rightarrow$ $x^2 - y^2$ may be a catastrophic cancellation

$x - y$ by guard digit

$x-y$ must be better than x^2-y^2

- A catastrophic cancellation is replaced by a benign cancellation

Of course x, y may have been rounded and $x - y$ is still a catastrophic cancellation.

Again, difficult to remove all catastrophic cancellations, but possible to remove some

Avoid Catastrophic Cancellation IV

- Calculating area of a triangle

$$A = \sqrt{s(s-a)(s-b)(s-c)}, s = \frac{a+b+c}{2} \quad (5)$$

a, b, c: length of three edges

If $a \approx b + c, s = (a + b + c)/2 \approx a$, $s - a$ may have a catastrophic error

Example: $a = 9.00, b = c = 4.53$
$s = 9.03, A = 2.342$

Computed solution: $A = 3.04$, error ≈ 0.7

ulps $= 0.01$, error $=$ 70 ulps

Avoid Catastrophic Cancellation V

- A new formulation by Kahan [1986] , $a \geq b \geq c$

$$A = \frac{\sqrt{(a+(b+c))(c-(a-b))(c+(a-b))(a+(b-c))}}{4} \tag{6}$$

  $A \approx 2.35$, close to 2.342

- HW 1-1: Calculate $A = 3.04$ using (5) and $A = 2.35$ using (6)

(5): $a = 9.00 \quad b = c = 4.53$

$$s = \frac{9.00 + (4.53 + 4.53)}{2}$$

$4.53 + 4.53 = 9.06$

$9.06 + 9.00 = 18.06$

$= 1.81$

$1.81 / 2 = 0.905 = 9.05 = s$

$s - a = 9.05 - 9.00 = 0.05$

$s - b = 9.05 - 4.53 = 4.52$

$s - c = 9.05 - 4.53 = 4.52$

$s \cdot (s-a) \cdot (s-b) \cdot (s-c)$

$= 0.4525 \times 20.4304$

$= (4.53 \times 10^{-1}) \times (2.04 \times 10^{1})$

$s \cdot (s-a)$

9.05

$\times 0.05$

0.4525

$= 9.24$

$(s-b)\cdot(s-c)$

4.52^2

$\times 4.52$

9.04

2260

1808

20.4304

$\sqrt{9.24} = 3.039$

≈ 3.04

$s\cdot(s-a)$

$(s-b)(s-c)$

4.53

$\times 2.04$

1812

9060

9.2412

(6) $a + (b+c) = 9.00 + (4.53 + 4.53)$

$= 9.00 + (9.06)$

$= 18.06 \approx 1.81 \times 10^1$

$c - (a-b) = 4.53 - (9.00 - 4.53)$

$= 4.53 - (4.47)$

$= 0.06$

$$c+(a-b) = 4.53+(9.00-4.53)$$

$$= 4.53+4.47$$

$$= 9.00$$

$$a+(b-c) = 9.00+(4.53-4.53)$$

$$= 9.00$$

$$1.81 \times 0.06 \overset{\times 10^1}{=} 10.86 \approx 1.09$$

$$9.00 \times 9.00 = 81.00 = 8.10 \times 10^1$$

$$1.09 \times 8.1$$

$$= 8.83$$

$$\begin{array}{r} 1.09 \\ \times\ 8.1 \\ \hline 109 \\ 872 \\ \hline 8.829 \end{array}$$

$$\sqrt{8.83 \times 10^1} = 9.3968$$

$$\approx 9.40$$

$$9.40/4 = 2.35$$

#

$$\begin{array}{r} 2.35 \\ 4\overline{)\,9.40} \\ 8 \\ \hline 14 \\ 12 \\ \hline 20 \end{array}$$

Avoid Catastrophic Cancellation VI

Note: to get $A = 3.04$ you need to calculate s by

$$s = \frac{a + (b + c)}{2}$$

Note that for multiplication and square root we assume that exact calculation can be done and results are rounded.

- Conclusion: sometimes a formula can be rewritten to have higher accuracy using benign cancellation
- Only works if guard digit is used; most computers use guard digits now

Avoid Catastrophic Cancellation VII

- But reformulation is difficult!!

  You may think that you will never need to do this
- Two real cases:
- Line 213-216 of tron.cpp of LIBLINEAR version 2.11

  `http://www.csie.ntu.edu.tw/~cjlin/liblinear/oldfiles` → similar to b^2-4ac

  HW1-2: Check Eq. (13) of the paper

  `http://www.csie.ntu.edu.tw/~cjlin/papers/logistic.pdf`

  and explain how we avoid catastrophic cancellations

Avoid Catastrophic Cancellation VIII

We do not consider the latest version of LIBLINEAR because some more complicated settings have been used

- Probability outputs of LIBSVM

  HW1-3: Repeat the experiment on page 5, line 12 of the paper

  `http://www.csie.ntu.edu.tw/~cjlin/papers/plattprob.pdf`

  Discuss what you found

HW 1-2:

Let $\alpha = \tau$

$$\left| \vec{s} + \alpha \vec{d} \right| = \Delta k \quad (13)$$

$$(s + \alpha d)^T (s + \alpha d) = \Delta k^2$$

$$s^T s + \alpha s^T d + \alpha s^T d + \alpha^2 d^T d$$

$$= \Delta k^2$$

$$\alpha^2 \underbrace{d^T d}_{a} + 2\alpha \underbrace{s^T d}_{b} + \underbrace{s^T s - \Delta k^2}_{c} = 0$$

$\Rightarrow$ (I think $\alpha > 0$, so take +)

$$\alpha = \frac{-2b + \sqrt{4b^2 - 4ac}}{2a}$$

$$= \frac{-b + \sqrt{b^2 - ac}}{a} \quad \text{rad}$$

(reformulation)

$$\alpha = \frac{-b+\sqrt{b^2-ac}}{a} \cdot \frac{(-b-\sqrt{b^2-ac})}{(-b-\sqrt{b^2-ac})}$$

$$= \frac{b^2-(b^2-ac)}{-ab-a\sqrt{b^2-ac}}$$

$$= \frac{ac}{-ab-a\sqrt{b^2-ac}}$$

$$= -\frac{c}{b+\sqrt{b^2-ac}}$$

$$= \boxed{\frac{\Delta_k^2 - s^T s}{s^T d + \mathrm{rad}}} \quad //$$

Exactly Rounded Operations I

- Round then calculate $\Rightarrow$ may not be very accurate
- Exactly rounded: compute exactly then rounded to the nearest $\Rightarrow$ usually more accurate
- The definition of rounding
- 12.5 $\Rightarrow$ 12 or 13 ?
- Rounding up: 0, 1, 2, 3, 4 $\Rightarrow$ down, 5, 6, 7, 8, 9 $\Rightarrow$ up

  Why called "rounding up"? Always up for 5
- Rounding even:

  the closest value with even least significant digit

Exactly Rounded Operations II

50% probability up, 50% down

example: $12.5 \Rightarrow 12$; $11.5 \Rightarrow 12$

even

- Reiser and Knuth [1975] show rounding even may be better

Theorem

Let $x_0 = x, x_1 = (x_0 \ominus y) \oplus y, \ldots, x_n = (x_{n-1} \ominus y) \oplus y$, *if* $\oplus$ *and* $\ominus$ *are exactly rounded using rounding even, then* $x_n = x, \forall n$ *or* $x_n = x_1, \forall n \geq 1$.

$x \ominus y$: computed solution

- Consider rounding up,

Exactly Rounded Operations III

$\beta = 10, p = 3, x = 1.00, y = -0.555$

$x - y = 1.555, x \ominus y = 1.56, (x \ominus y) + y = 1.56 - 0.555 = 1.005, x_1 = (x \ominus y) \oplus y = 1.01$

$x_1 - y = 1.565, x_1 \ominus y = 1.57, (x_1 \ominus y) + y = 1.57 - 0.555 = 1.015, x_2 = (x_1 \ominus y) \oplus y = 1.02$

Increased by 0.01 until $x_n = 9.45$

- Rounding even:

even

$x - y = 1.555, x \ominus y = 1.56, (x \ominus y) + y = 1.56 - 0.555 = 1.005, x_1 = (x \ominus y) \oplus y = 1.00$

even

$x_1 - y = 1.555, x_1 \ominus y = 1.56, (x_1 \ominus y) + y = 1.56 - 0.555 = 1.005, x_2 = (x_1 \ominus y) \oplus y = 1.00$

Exactly Rounded Operations IV

- How to implement "exactly rounded operations"?

  We can use an array of words or floating-points

  But you don't have an infinite amount of spaces
- Goldberg [1990] showed that using 3 guard digits the result is the same as using exactly rounded operations

IEEE standard I

- IEEE 754 during 80s, now standard everywhere
- Two IEEE standards:
  1. 754: specify $\beta = 2, p = 24$ for single, $\beta = 2, p = 53$ for double
  2. 854 ($\beta = 2$ or 10, does not specify how floating-point numbers are encoded into bits)
- Why IEEE 854 allows $\beta = 2$ or 10 but not other numbers:

  10 is the base we use

  smaller β causes smaller relative error

IEEE standard II

$\varepsilon = \frac{\beta}{2}\beta^{-p}$ (relative error)

smaller β: more precision. For example,

$$\beta = 16, p = 1 \text{ versus } \beta = 2, p = 4$$

- 4 bits for significand

$$\epsilon = \frac{16}{2}16^{-1} = 1/2, \epsilon = \frac{2}{2}2^{-4} = 1/16$$

We can see that ϵ of $\beta = 2, p = 4$ is smaller

- However, IBM/370 uses $\beta = 16$. Why? Two possible reasons:

First,

IEEE standard III

a number: 4 bytes $= 32$ bits

$\beta = 16, p = 6$, significand: $4 \times 6 = 24$ bits,

exponents: $32 - 24 - 1 = 7$ bits (1 bit for sign),

16^{-2^6} to $16^{2^6} = 2^{2^8}$

for $\beta = 2 \Rightarrow 9$ bits (-2^8 to $2^8 = 2^9$) for exponents,

$\Rightarrow 32 - 9 - 1 = 22$ for significand

Same exponents, less significand for $\beta = 2$ (24 vs. 22)

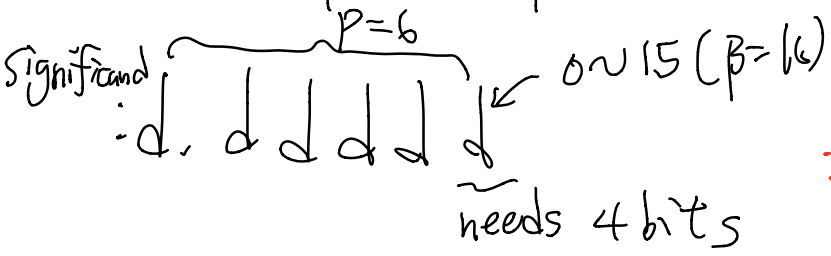
Second,

Shifting: $\beta = 16$, less frequently to adjust exponents when adding or subtracting two numbers

4 bytes = 32 bits

line up their exponents.

Setting 1: $\beta = 16, p = 6$



$\Rightarrow$ 6x (4 bits) = 24 bits

$\Rightarrow$ 32 bits - 24 - 1 = 7 bits (exponent

$\Rightarrow$ exp: $2^6 - 1 \sim -2^6$

$\Rightarrow$ represent: $16^{2^6-1} \sim 16^{-2^6}$

$\Rightarrow$ $2^{2^8-2^2} \sim 2^{-2^8}$

$\approx$ $2^8 \sim 2^{-2^8}$ $\approx$ 9 bits

if $\beta = 16$, $\beta = 2$ have same significands

$\beta = 16$'s exponent range is bigger

32 - 9 - 1 = 22 bits for significands

for $\beta = 2$, needs

IEEE standard IV

For modern computers, this saving is not important

- Single precision: $\beta = 2, p = 24$ (23 bits as normalized), exponent 8, 1 bit for sign ($32 = 23 + 8 + 1$)
- An example: $176.625 = 1.0101100101 \times 2^7$

sign 0 | 10000110 | 23 bits 01011001010000000000000

1 of $1.\cdots$ is not stored (normalized)

- Biased exponent (described later in detail)

$10000110 = 128 + 4 + 2 = 134, 134 - 127 = 7$

Note that we have negative exponent

IEEE standard V

(3 guard digits?)

- Use rounding even

Binary	rounded	reason
10.00011	10.00	($< 1/2$, down)
10.00110	10.01	($> 1/2$, up)
10.11100	11.00	($1/2$, up)
10.10100	10.10	($1/2$, down)

even

even

This example is from `http://www.cs.cmu.edu/afs/cs/academic/class/15213-s12/www/lectures/04-float-4up.pdf`

- A summary

IEEE standard VI

IEEE	Fortran	C	Bits	Exp.	Mantissa
Single	REAL*4	float	32	8	24
Single-extended			44	≤ 11	32
Double	REAL*8	double	64	11	53
Double-extended	REAL*10	long double	≥ 80	≥ 15	≥ 64

$32 = 8 + 24$ but $44 \neq 11 + 32$

- $44 \neq 11 + 32$:

  Hardware implementation of extended precision normal don't use a hidden bit

  (Remember we normalized each number so 1 is not stored)

IEEE standard VII

- It seems everyone is using double now

  But single is still needed sometime (if memory is not enough)
- Minimal normalized positive number

minimum value a single presion number can represent

$$1 \times 2^{-126} \approx 1.17 \times 10^{-38}$$

$e_{\min} = -126$ (Single precision)

- 8 bits for exponent: 0 to 255. IEEE uses biased approach exponent (i.e.

$$(0 \text{ to } 255) - 127 = -127 \text{ to } 128$$

IEEE standard VIII

- Why $e_{\min} = -126$ but $e_{\max} = 127$?

  reasons: $1/2^{e_{\min}}$ not overflow, $1/2^{e_{\max}}$ underflow, but less serious

- Thus, -127 for 0 and denormalized numbers (discussed later), -126 to 127 for exponents, 128 for special quantity
- Motivation for extended precision: from calculator, display 10 digits but 13 internally

  Some operations benefit from using more digits internally

IEEE standard IX

Example: binary-decimal conversion (Details not discussed here)

- Operations: IEEE standard requires results of addition, subtraction, multiplication and division exactly rounded.
- Exactly rounded: an array of words or floating-point numbers, expensive
- Goldberg [1990] showed using 3 guard digits the result is the same as using exactly rounded operations

  Only little more cost

IEEE standard X

- Reasons to specify operations

  run on different machines $\Rightarrow$ results the same
- HW 2-1: write the binary format of -250 as a double floating-point number
- IEEE: square root, remainder, conversion between integer and floating-point, internal formats and decimal are correctly rounded (i.e. exactly rounded operations)
- Binary to decimal conversion

  Think about reading numbers from files

HW 2-1: double = 64 bits = 52 + sign + 11 bits exp

$p = 52 + 1$ (hidden)

sign bit = 1

$250 = 1\times2^7 + 1\times2^6 + 1\times2^5 + 1\times2^4 + 1\times2^3 + 1\times2^1$

$= 11111010_2$

250 − 128 = 122

122 − 64 = 58

58 − 32 = 26

26 − 16 = 10

10 − 8 = 2

1.1111010×2^{7}

$7 + (2^{11} - 1) = 2^{11} + 6$

biased $= 10000000110$

⇒ In summary:

sign	exp	
1	10000000110 ($2^{11}+6$)	1111010---0

IEEE standard XI

When writing a binary number to a decimal number and read it back, can we get the same binary number?

- Writing 9 digits is enough for short

  Though $10^8 > 2^{24}$, 8 digits are not enough
- 17 for double precision (proof not provided). Example:

  numbers in a data set from Matrix market:

IEEE standard XII

```
> tail s1rmq4m1.dat
  8.25117360856184 38E+01   2.513452865992495
 -6.00429512550414 66E+00   8.659944220661552
  1.00261976195637 23E+01  -1.313683766184450
 -1.51083310403612 31E+01   5.142317399695508
 -1.16902863459613 63E+03   1.625072665580781
  8.25117360744732 20E+01   1.510833104036122
```

- Matrix market:

  `http://math.nist.gov/MatrixMarket/`

  A collection of matrix data

IEEE standard XIII

- Transcendental numbers:

  e.g., exp, log
- IEEE does not require transcendental functions to be exactly rounded

  Cannot specify the precision because they are arbitrarily long

Special quantities I

- On some computers (e.g., IBM 370) every bit pattern is a valid floating-point number
- For IBM 370, $\sqrt{-4} = 2$ and it prints an error message

  IEEE : `NaN`, not a number

  why $\sqrt{-4} = 2$ on IBM 370 $\Rightarrow$ every pattern is a number
- Special value of IEEE:

  $+0, -0$, denormalized numbers, $+\infty$, $-\infty$, NaNs (more than one NaN)

Special quantities II

- A summary

Exponent	significand	represents
$e = e_{\min} - 1$	$f = 0$	$+0, -0$
$e = e_{\min} - 1$	$f \neq 0$	$0.f \times 2^{e_{\min}}$
$e_{\min} \leq e \leq e_{\max}$		$1.f \times 2^{e}$
$e = e_{\max} + 1$	$f = 0$	$\pm\infty$
$e = e_{\max} + 1$	$f \neq 0$	NaN

- Why IEEE has NaN

  Sometimes even 0/0 occurs, the program can continue

Special quantities III

- Example: find $f(x) = 0$, try different x's, even $0/0$ happens, other values may be ok.
- If $b^2 - 4ac < 0$

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

returns NaN

$-b+$ NaN should be NaN

In general when a NaN is in an operation, result is NaN

- Examples producing NaN:

Special quantities IV

Operation	NaN by
$+$	$\infty + (-\infty)$
$\times$	$0 \times \infty$
$/$	$0/0, \infty/\infty$
REM	x REM 0, ∞ REM y
$\sqrt{}$	$\sqrt{x}$ when $x < 0$

Infinity I

- $\beta = 10, p = 3, e_{\max} = 98$, $x = 3 \times 10^{70}$,
  x^2 overflow and replaced by 9.99×10^{98}??
  In IEEE, the result is ∞
- Note $0/0 =$ NaN, $1/0 = \infty, -1/0 = -\infty$
  $\Rightarrow$ nonzero divided by 0 is ∞ or $-\infty$
  Similarly, $-10/0 = -\infty$, and $-10/-0 = +\infty$
  (± 0 will be explained later)
- $3/\infty = 0, 4 - \infty = -\infty, \sqrt{\infty} = \infty$
- How to know the result?
  replace ∞ with x, let $x \to \infty$

Infinity II

Example:

$$3/\infty : \quad \lim_{x \to \infty} 3/x = 0$$

If limit does not exist $\Rightarrow$ NaN

- $x/(x^2+1)$ vs $1/(x+x^{-1})$

  $x/(x^2+1)$: if x is large, x^2 overflow, $x/\infty = 0$ but not $1/x$.

  $1/(x+x^{-1})$: x large, $1/x$ ok

  $1/(x+x^{-1})$ looks better but what about $x = 0$?

  $x = 0$, $1/(0+0^{-1}) = 1/(0+\infty) = 1/\infty = 0$
- If no infinity arithmetic, an extra instruction needed to test if $x = 0$, may interrupt the pipeline

Signed zero I

- Why do we have $+0$ and -0 ?

  First, it is available (1 bit for sign)

  if no sign, $1/(1/x)) = x$ fails when $x = \pm\infty$

  $x = \infty, 1/x = 0, 1/0 = +\infty$

  $x = -\infty, 1/x = 0, 1/0 = +\infty$
- Compare $+0$ and -0: `if (x == 0)`

  IEEE defines $+0 = -0$
- IEEE: $3 \times (+0) = +0$, $+0/(-3) = -0$

Signed zero II

- ± 0 useful in the following situations:

$$\log x \equiv \begin{cases} -\infty & x = 0 \\ \text{NaN} & x < 0 \end{cases}$$

A small underflow negative number $\Rightarrow \log x$ should be NaN

x underflow $\Rightarrow$ round to 0, if no sign, $\log x$ is $-\infty$ but not NaN

Signed zero III

- With ± 0, we have

$$\log x = \begin{cases} -\infty & x = +0 \\ \text{NaN} & x = -0 \\ \text{NaN} & x < 0 \end{cases}$$

  Positive underflow $\Rightarrow$ round to $+0$

- Very useful in complex arithmetic

  $\sqrt{1/z}$ and $1/\sqrt{z}$

  $z = -1, \sqrt{1/-1} = \sqrt{-1} = i, 1/\sqrt{-1} = 1/i = -i$

  $\Rightarrow \sqrt{1/z} \neq 1/\sqrt{z}$

Signed zero IV

- This happens because square root is multi-valued.
  $i^2 = (-i)^2 = -1$
- However, by some restrictions (or ways of calculation), they can be equal
- $z = -1 = -1 + 0i$,
  $1/z = 1/(-1 + 0i) = -1 + (-0)i$
  so $\sqrt{1/z} = \sqrt{-1 + (-0)i} = -i$
  $\Rightarrow -0$ is useful
- Disadvantage of $+0$ and -0:
  $x = y \Leftrightarrow 1/x = 1/y$ is destroyed
  $x = 0, y = -0 \Rightarrow x = y$ under IEEE

Signed zero V

$$1/x = +\infty, 1/y = -\infty, +\infty \neq -\infty$$

- There are always pros and cons for floating-point design

HW 2-2 I

- If

  ```
  if (a < 0)
  ```

  always holds and b is neither too large nor too small, how do we guarantee

  ```
  if a/max(b, 0.0) < 0
  ```

  always holds
- If `max(b,0.0)` returns -0.0, then it may not hold
- For the max function, should we use

  ```
  (x>y)?  x:y
  ```

  or

HW 2-2 II

```
(x<y)?  y:x
```

- Your max need to return $+0.0$ but not -0.0
- How to specifically assign $+0.0$ and -0.0?
- How to use subroutines to get the sign of a number?
- In a regular program, if you write 0.0, is it $+0.0$ or -0.0?

  Find the statement in the manual saying that 0.0 means $+0.0$
- Do some experiments to check your arguments
- Use Java but not other systems

Denormalized number I

- $\beta = 10, p = 3, e_{\min} = -98$, $x = 6.87 \times 10^{-97}$, $y = 6.81 \times 10^{-97}$
- x, y are ok but $x - y = 0.6 \times 10^{-98}$ rounded to 0, even though $x \neq y$
- How important to preserve

$$x = y \Leftrightarrow x - y = 0$$

- `if (x ≠ y) {z = 1/(x-y);}`
  The statement is true, but z becomes ∞
  Tracking such bugs is frustrating

Denormalized number II

- IEEE uses denormalized numbers

  Guarantee $x = y \Leftrightarrow x - y = 0$

  Details of how this is done are not discussed here
- Most controversial part in IEEE standard

  It caused long delay of the standard
- If denormalized number is used, 0.6×10^{-98} is also a floating-point number
- Remember we do not store 1 of $1.d \cdots d$
- How to represent denormalized numbers ?

  Recall for valid value, $e \geq e_{\min}$ and we have $1.d \cdots d \times 2^e$

Denormalized number III

- For denormalized numbers, we let $e = e_{\min} - 1$ and the corresponding value be

$$0.d \cdots d \times 2^{e+1} = 0.d \cdots d \times 2^{e_{\min}}$$

- Why not

$$1.d \cdots d \times 2^{e_{\min}-1}$$

  can't represent

$$0.0x \ldots x \times 2^{e_{\min}}$$

- $6.87 \times 10^{-97} - 6.81 \times 10^{-97} \Rightarrow$ underflow due to cancellation

Denormalized number IV

Underflow: smaller than the smallest floating-point number

- An example of using denormalized numbers

$$\begin{aligned}\frac{a+bi}{c+di} &= \frac{(a+bi)(c-di)}{(c+di)(c-di)} \\ &= \frac{ac+bd}{c^2+d^2} + \frac{bc-ad}{c^2+d^2}i\end{aligned}$$

If c or $d > \sqrt{\beta}\beta^{e_{max}/2} \Rightarrow$ overflow

Denormalized number V

overflow: larger than the maximal floating-point number

- Smith's formula

$$\frac{a+bi}{c+di} = \begin{cases} \frac{a+b(d/c)}{c+d(d/c)} + \frac{b-a(d/c)}{c+d(d/c)}i & \text{if } (|d| < |c|) \\ \frac{b+a(c/d)}{d+c(c/d)} + \frac{-a+b(c/d)}{d+c(c/d)}i & \text{if } (|d| \geq |c|) \end{cases}$$

avoid overflow

- However, using Smith's formula, without denormalized numbers

Denormalized number VI

If

$$a = 2 \times 10^{-98}, b = 1 \times 10^{-98}, c = 4 \times 10^{-98},$$
$$d = 2 \times 10^{-98}$$

then

$$d/c = 0.5, c + d(d/c) = 5 \times 10^{-98},$$
$$b(d/c) = 1 \times 10^{-98} \times 0.5 = 0$$
$$a + b(d/c) = 2 \times 10^{-98}$$

Solution $= 0.4$, wrong

Denormalized number VII

If denormalized numbers are used, 0.5×10^{-98} can be stored,

$$a + b(d/c) = 2.5 \times 10^{-98} \Rightarrow 0.5$$

the correct answer

- Usually hardware does not support denormalized numbers directly

  Using software to simulate
- Programs may be slow if a lot of underflow

Exception, Flags, Trap handlers I

- We have mentioned things like overflow, underflow

  What are other exceptional situations ?
- Motivation: usually when exceptional condition like $1/0$ happens, you may want to know
- IEEE requires vendors to provide a way to get status flags
- IEEE defines five exceptions: overflow, underflow, division by zero, invalid operation, inexact
- overflow: larger than the maximal floating-point number

Exception, Flags, Trap handlers II

Underflow: smaller than the smallest floating-point number

- Invalid:

  $\infty + (-\infty), 0 \times \infty, 0/0, \infty/\infty$,
  x REM $0, \infty$ REM y, $\sqrt{x}, x < 0$, any comparison involves a NaN

- Invalid returns NaN; NaN may not be from invalid operations

- Inexact: the result is not exact

  $\beta = 10, p = 3, 3.5 \times 4.2 = 14.7$ exact,
  $3.5 \times 4.3 = 15.05 \Rightarrow 15.0$ not exact

Exception, Flags, Trap handlers III

inexact exception is raised so often, usually we ignore it

Exception	when trap disabled	argument to handler
overflow	$\pm\infty$ or $\pm 1.1\cdots 1 \times 2^{e_{max}}$	round$(x2^{-\alpha})$
underflow	$0, \pm 2^{e_{min}}$, or denormal	round$(x2^{\alpha})$
division by zero	∞	operands
invalid	NaN	operands
inexact	round(x)	round(x)

- Trap handler: special subroutines to handle exceptions

Exception, Flags, Trap handlers IV

You can design your own trap handlers

- In the above table, "when trap disabled" means results of operations if trap handlers not used
- $\alpha = 192$ for single, $\alpha = 1536$ for double
  reason: you cannot really store x
- Examples of using trap handlers described later

Compiler Options I

- Compiler may provide a way so the program stops if an exception occurs
- Easy for debugging
- Example: SUN's C compiler (I learned this on an old machine)
- Reason: gcc doesn't have this to explicitly detect exceptions
- **-ftrap=t**

Compiler Options II

- t: %all, %none, common, [no%]invalid, [no%]overflow, [no%]underflow, [no%]division, [no%]inexact.
- common: invalid, division by zero, and overflow.
- The default is -ftrap=%none.
- Example: -ftrap=%all,no%inexact means set all traps, except inexact.
- If you compile one routine with -ftrap=t, compile all routines of the program with the same -ftrap=t option

  otherwise, you can get unexpected results.

Compiler Options III

- Example: on the screen you will see

```
Note: IEEE floating-point exception flags rais
   Inexact;  Underflow;
See the Numerical Computation Guide, ieee_flag
```

- gcc:
- -fno-trapping-math: default -ftrapping-math

  Setting this option may allow faster code if one relies on "non-stop" IEEE arithmetic
- -ftrapv

Compiler Options IV

Generates traps for signed overflow on addition, subtraction, multiplication

Trap Handler I

- Example:

```
do  {
    ....
} while {not x >= 100;}
```

  If $x = \text{NaN}$, an infinite loop
  Any comparison involving NaN is wrong
- A trap handler can be installed to abort it
- Example:

Trap Handler II

Calculate $x_1 \times \cdots \times x_n$ may overflow in the middle (the total may be ok!):

```
for (i = 1; i <= n; i++)
    p = p * x[i] ;
```

- $x_1 \times \cdots \times x_r, r \leq n$ overflow but $x_1 \times \cdots \times x_n$ may be in the range
- $e^{\sum \log(x_i)} \Rightarrow$ a solution but less accurate and costs more
- A possible solution

Trap Handler III

```
for (i = 1; i <= n; i++) {
    if (p * x[i] overflow) {
        p = p * pow(10,-a);
        count = count + 1 ;
    }
    p = p * x[i] ;
}
p = p * pow(10, a*count) ;
```

An Example of Handlers I

- Example using SUN's numerical computation guide

  Again, old. Reason of not using existing systems such a glibc: so you can have HW
- standard math library libm.a
  exp, pow, log, ...
- On SUN machines, there are additional math library: libsunmath.a
  exp2, exp10, ..., ieee_flags, ieee_handler, ieee_retrospective
- A program:

An Example of Handlers II

```
#include <stdio.h>
#include <sys/ieeefp.h>
#include <sunmath.h>
#include <siginfo.h>
#include <ucontext.h>

void handler(int sig, siginfo_t *sip,
             ucontext_t *uap)
{
    unsigned    code, addr;

    code = sip->si_code;
```

An Example of Handlers III

```
    addr = (unsigned) sip->si_addr;
    fprintf(stderr, "fp exception %x at
        address %x \n", code, addr);
}
int main()
{
    double  x;

    /* trap on common floating point
       exceptions */
    if (ieee_handler("set", "common", handler)
        != 0)
```

An Example of Handlers IV

```
        printf("Did not set exception
                handler \n");

/* cause an underflow exception (not
   reported) */
x = min_normal();
printf("min_normal = %g \n", x);
x = x / 13.0;
printf("min_normal / 13.0 = %g \n", x);

/* cause an overflow exception
   (reported) */
```

An Example of Handlers V

```
    x = max_normal();
    printf("max_normal = %g \n", x);
    x = x * x;
    printf("max_normal * max_normal = %g \n",
           x);

    ieee_retrospective(stderr);
    return 0;
}
```

- Result:

An Example of Handlers VI

```
min_normal = 2.22507e-308
min_normal / 13.0 = 1.7116e-309
max_normal = 1.79769e+308
fp exception 4 at address 10d0c
max_normal * max_normal = 1.79769e+308
 Note: IEEE floating-point exception flags rais
    Inexact;  Underflow;
 IEEE floating-point exception traps enabled:
    overflow; division by zero; invalid operati
 See the Numerical Computation Guide, ieee_flag
ieee_handler(3M)
```

An Example of Handlers VII

- invalid, division, and overflow sometimes called common exceptions here

  ieee_handler("set", "common", handler) means handlers used for common exceptions
- min_normal / 13.0: using denormalized numbers

  handler: subroutines to handle exceptions
- HW 3-1: regenerate this example using GNU C library

An Example of Handlers VIII

- How to find GNU C library information: on linux, type
  % info libc
  check the category of "Arithmetics" and "Signal Handling"

The Use of Flags: An Example I

- Calculate x^n, n : integer

```
double pow(double x, int n)
{
        double tmp = x, ret = 1.0;

        for(int t=n; t>0; t/=2)
        {
                if(t%2==1) ret*=tmp;
                tmp = tmp * tmp;
        }
        return ret;
```

The Use of Flags: An Example II

```
}
```

$x^{16} = (x^2)^8 = \cdots$, $x^{15} = x(x^2)^7$, treat x^2 as the new x

$x^{15} = x(x^2)^7 = x(x^2)(x^4)^3 = x(x^2)(x^4)(x^8)^1$

- If $n < 0$, we need to use

$$x^n = (1/x)^{-n} = 1/(x)^{-n}$$

pow$(1/x, -n)$ less accurate, $1/$pow$(x, -n)$ is better

There is already error on $1/x$

The Use of Flags: An Example III

Example: $2^{-5} = (1/2)^5$ and $1/(2^5)$

- A small problem on using $1/\text{pow}(x, -n)$:

  if $\text{pow}(x, -n)$ underflow (i.e. when $x < 1, n < 0$), either underflow trap handler or underflow status flag set $\Rightarrow$ incorrect

  x^{-n} underflow, x^n overflow or be in range $(e_{\text{min}} = -126, 2^{-e_{\text{min}}} = 2^{126} < 2^{127} = 2^{e_{\text{max}}})$

- Turn off overflow & underflow trap enable bits, save overflow & underflow status bits

  Compute $1/\text{pow}(x, -n)$

The Use of Flags: An Example IV

If neither overflow nor underflow status is set $\Rightarrow$ restore them

If one is set, restore & calculate pow$(1/x, -n)$, which causes correct exception to occur

- Practically the calculation of pow() is more complicated

  e.g. google e_pow.c and e_log.c

- In glibc-2.17/sysdeps/ieee754/dbl-64, e_pow.c has 420 lines

The Use of Flags: An Example V

- Another example: calculate arccos using arctan

$$\arccos x = 2 \arctan \sqrt{\frac{1-x}{1+x}}$$

$$\cos\theta = x = 2\cos^2\frac{\theta}{2} - 1 = 1 - 2\sin^2\frac{\theta}{2}$$

$$\cos\frac{\theta}{2} = \sqrt{\frac{x+1}{2}}, \sin\frac{\theta}{2} = \sqrt{\frac{1-x}{2}}, \tan\frac{\theta}{2} = \sqrt{\frac{1-x}{1+x}}$$

Hence

$$\arccos x = 2 \arctan \sqrt{\frac{1-x}{1+x}}$$

The Use of Flags: An Example VI

- Consider $x = -1$

  $\arctan(\infty) = \pi/2 \Rightarrow \arccos(-1) = \pi$
- A small problem:

  $\frac{1-x}{1+x}$ causes the divide-by-zero flag set though $\arccos(-1)$ not exceptional
- Solution: save divide-by-zero flag, restore after arccos computation

A Real Study I

- Let's start with a simple example

```
#include <stdio.h>

int main()
{
float a = 123.123;
printf("%.10f\n", a);
printf("%.10f\n", a*a);

a = 123.125;
printf("%.10f\n", a);
```

A Real Study II

```
printf("%.10f\n", a*a);

}
```

- Results are

```
$gcc test.c;./a.out
123.1230010986
15159.2734375000
123.1250000000
15159.7656250000
$gcc -m32 test.c;./a.out
123.1230010986
```

A Real Study III

```
15159.2733995339
123.1250000000
15159.7656250000
```

- -m 32 generates code for a 32-bit environment (because we don't have a 32-bit machine)
- That is, same code gives different results under 32 and 64-bit environments
- Why?

A Real Study IV

- On 32 bit, 387 floating-point coprocessor is used. From gcc manual, "The temporary results are computed in 80-bit precision instead of the precision specified by the type, resulting in slightly different results compared to most of other chips."
- In other words, they somehow violate IEEE standard
- But 123.123 has infinite digits after transformed to binary

A Real Study V

- Compiler options can help to make things more consistent. For example, -ffloat-store: "Do not store floating-point variables in registers, and inhibit other options that might change whether a floating-point value is taken from a register or memory."

```
$gcc -ffloat-store test.c;./a.out
123.1230010986
15159.2734375000
123.1250000000
15159.7656250000
$gcc -ffloat-store -m32 test.c;./a.out
123.1230010986
```

A Real Study VI

```
15159.2734375000
123.1250000000
15159.7656250000
```

- Note that other issues such as order of operations can also affect results.
- Consider running a real example using a machine learning software LIBSVM
- 64 bit:

A Real Study VII

```
$ ./svm-train -c 100 -e 0.00001 heart_scale
.........*..*
optimization finished, #iter = 2872
nu = 0.148045
obj = -2526.925470, rho = 1.145512
nSV = 107, nBSV = 9
Total nSV = 107
```

- 32bit:

A Real Study VIII

```
$  ./svm-train -c 100 -e 0.00001 heart_scale
.........*..*
optimization finished, #iter = 2819
nu = 0.148045
obj = -2526.925470, rho = 1.145515
nSV = 107, nBSV = 9
Total nSV = 107
```

- They are different
- Adding `-ffloat-store -mfpmath=387` is not enough
- 64 bit:

A Real Study IX

```
$ make clean; make
$ ./svm-train -c 100 -e 0.00001 heart_scale
rm -f *~ svm.o svm-train svm-predict svm-sca
g++ -Wall -Wconversion -O3 -fPIC -ffloat-sto
g++ -Wall -Wconversion -O3 -fPIC -ffloat-sto
g++ -Wall -Wconversion -O3 -fPIC -ffloat-sto
g++ -Wall -Wconversion -O3 -fPIC -ffloat-sto
.........*..*
optimization finished, #iter = 2863
nu = 0.148045
obj = -2526.925470, rho = 1.145512
nSV = 107, nBSV = 9
```

A Real Study X

```
Total nSV = 107
```

- We also need to disable all optimization
- 64bit:

```
$ make clean; make
$ ./svm-train -c 100 -e 0.00001 heart_scale
rm -f *~ svm.o svm-train svm-predict svm-sca
g++ -ffloat-store -mfpmath=387 -c svm.cpp
g++ -ffloat-store -mfpmath=387 svm-train.c s
g++ -ffloat-store -mfpmath=387 svm-predict.c
g++ -ffloat-store -mfpmath=387 svm-scale.c -
.........*...*
```

A Real Study XI

```
optimization finished, #iter = 3051
nu = 0.148045
obj = -2526.925470, rho = 1.145515
nSV = 107, nBSV = 9
Total nSV = 107
```

- 32 bit:

```
$ make clean; make
$ ./svm-train -c 100 -e 0.00001 heart_scale
rm -f *~ svm.o svm-train svm-predict svm-sca
g++ -m32 -ffloat-store -mfpmath=387 -c svm.c
g++ -m32 -ffloat-store -mfpmath=387 svm-trai
```

A Real Study XII

```
g++ -m32 -ffloat-store -mfpmath=387 svm-prec
g++ -m32 -ffloat-store -mfpmath=387 svm-scal
.........*...*
optimization finished, #iter = 3051
nu = 0.148045
obj = -2526.925470, rho = 1.145515
nSV = 107, nBSV = 9
Total nSV = 107
```