

## Operations on Floating Point Numbers

There are very few:

Standard **arithmetic** operations:  $+$ ,  $-$ ,  $*$ ,  $/$ , plus **comparison**, **square root**, etc.

The **operands** must be available in the processor memory, and so are **floating point numbers**. But combining two floating point numbers may **not** give a floating point number. e.g. **multiplication** of two **24-bit** significands generally gives a **48-bit** significand.

When the result is **not** a floating point number, the IEEE standard requires that the computed result be the **correctly** rounded value of the **exact** result

**Q:** Using **round to nearest**, which of  $1 + 10^{-5}$ ,  $1 + 10^{-10}$ ,  $1 + 10^{-15}$  round to 1 ??

## IEEE Rule for Rounding

Let  $x$  and  $y$  be floating point numbers, and let  $\oplus, \ominus, \otimes, \oslash$  denote the **implementations** of  $+, -, *, /$  on the computer. Thus  $x \oplus y$  is the computer's **approximation** to  $x + y$ .

The **IEEE rule** is then precisely:

$$x \oplus y = \text{round}(x + y),$$

$$x \ominus y = \text{round}(x - y),$$

$$x \otimes y = \text{round}(x * y),$$

$$x \oslash y = \text{round}(x / y).$$

From our discussion of relative rounding errors, when  $x + y$  is a **normalized** number,

$$\boxed{x \oplus y = (x + y)(1 + \delta), \quad |\delta| \leq \epsilon,}$$

for **all** rounding modes.

Similarly for  $\ominus, \otimes$  and  $\oslash$ .

(Note that  $|\delta| \leq \epsilon/2$  for **round to nearest**).

## Implementing Correctly Rounded FPA

Consider **adding** two IEEE single precision CFPNs:  $x = m \times 2^E$  and  $y = p \times 2^F$ .

First (if necessary) **shift one significand**, so both numbers have the same **exponent**

$G = \max\{E, F\}$ . The significands are then **added**, and if necessary, the result **normalized** and **rounded**. e.g. adding

$3 = (1.100)_2 \times 2^1$  to  $3/4 = (1.100)_2 \times 2^{-1}$  :

$$\begin{aligned} & ( 1.10000000000000000000000000000000 )_2 \times 2^1 \\ + & ( 0.01100000000000000000000000000000 )_2 \times 2^1 \\ = & ( 1.11100000000000000000000000000000 )_2 \times 2^1. \end{aligned}$$

Further normalizing & rounding is not needed.

Now add 3 to  $3 \times 2^{-23}$ . We get

$$\begin{aligned} & ( 1.10000000000000000000000000000000 )_2 \times 2^1 \\ + & ( 0.00000000000000000000000000000001|1 )_2 \times 2^1 \\ = & ( 1.10000000000000000000000000000001|1 )_2 \times 2^1 \end{aligned}$$

Result is **not** an IEEE single precision CFPN, and so must be **correctly rounded**.

## Difficulties of Implementation

Correctly rounded floating point addition and subtraction is not trivial. e.g.  $x - y$  with  $x = (1.0)_2 \times 2^0$  and  $y = (1.1111 \dots 1)_2 \times 2^{-1}$ , where  $y$  is the next floating point number smaller than  $x$ . **Aligning** the significands:

$$\begin{aligned} & (1.00000000000000000000000000000000| \quad )_2 \times 2^0 \\ - & (0.11111111111111111111111111111111|1 \quad )_2 \times 2^0 \\ = & (0.00000000000000000000000000000000|1 \quad )_2 \times 2^0 \end{aligned}$$

an example of **cancellation**, (most bits in the two numbers cancel each other). The result is  $(1.0)_2 \times 2^{-24}$ , a floating point number, but to obtain this we must **carry out the subtraction using an extra bit, a guard bit**.

**Cray** supercomputers do not have a guard bit.

Cray XMP equivalent operation gives:

$x \ominus y = 2(x - y)$ , **wrong** by a factor of two.

Cray YMP gives  $x \ominus y = 0$ . Cray supercomputers do **not** use correctly rounded arithmetic.

Difficulties of Implementation, ctd.

Machines supporting the IEEE standard **do** have correctly rounded arithmetic, so that e.g.  $x \ominus y = \text{round}(x - y)$  **always** holds. How this is implemented depends on the machine. Typically floating point operations are carried out using **extended precision registers**, e.g. 80-bit registers.

Surprisingly, even **24 guard bits don't guarantee** correctly rounded addition with 24-bit significands when the rounding mode is **round to nearest**. Machines that implement correctly rounded arithmetic take such possibilities into account, and it turns out that **correctly rounded results can be achieved in all cases using only two guard bits together with an extra bit**, called a **sticky bit**, which is used to **flag** a rounding problem of this kind.

## Floating point multiplication

If  $x = m \times 2^E$  and  $y = p \times 2^F$ , then

$$x \times y = (m \times p) \times 2^{E+F}$$

Three steps: **multiply** the significands, **add** the exponents, and **normalize** and correctly **round** the result. **Single precision** significands are easily multiplied in an **extended precision** register, (the product of two **24-bit** significand bitstrings is a **48-bit** bitstring which is then correctly rounded to 24 bits after normalization). Multiplication of **double precision** or **extended precision** significands is not so straightforward: dropping bits may lead to incorrectly rounded results.

**Multiplication** and **division** are more complicated than **addition** and **subtraction**, and may require **more execution time**; certainly on PC's, not always on supercomputers.

**Q:** Assume that

$$x = m \times 2^E, \quad y = p \times 2^F$$

are normalized numbers, i.e.

$$1 \leq m < 2; \quad 1 \leq p < 2.$$

How many bits may it be necessary to shift the **significand** product  $m \times p$  left or right to normalize the result ??

## Exceptional Situations

When a reasonable response to exceptional data is possible, it should be used.

The simplest example is **division by zero**. Two **earlier** standard responses:

(i) generate the **largest floating point number** as the result. Rationale: user would notice the large number in the output and conclude something had gone wrong. **Disaster**: e.g.  $2/0 - 1/0$  would then have a result of 0, which is **completely meaningless**.

In general the user might **not even notice** that any error had taken place).

(ii) generate a **program interrupt**, e.g. “**fatal error — division by zero**”.

The burden was on the programmer to make sure that division by zero would **never** occur.

**Example:** Consider computing the **total resistance** in an electrical circuit with two resistors ( $R_1$  and  $R_2$  ohms) connected in parallel:

figure=resis.ps,height=1.2in,width=3in

Figure 5: The Parallel Resistance Circuit

The formula for the **total resistance** is

$$T = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2}}.$$

What if  $R_1 = 0$ ? If one resistor offers no resistance, **all** the current will flow through that and avoid the other; therefore, the total resistance in the circuit is **zero**. The formula for  $T$  also makes perfect sense **mathematically**:

$$T = \frac{1}{\frac{1}{0} + \frac{1}{R_2}} = \frac{1}{\infty + \frac{1}{R_2}} = \frac{1}{\infty} = 0.$$

## The IEEE FPA Solution

Why should a **programmer** have to worry about treating division by zero as an exceptional situation here?

In **IEEE floating point arithmetic**, if the initial floating point environment is set properly: **division by zero** does not generate an interrupt but **gives an infinite result**, program execution continuing normally.

In the case of **the parallel resistance formula** this leads to a final **correct result** of  $1/\infty = 0$ , following the mathematical concepts **exactly**:

$$T = \frac{1}{\frac{1}{0} + \frac{1}{R_2}} = \frac{1}{\infty + \frac{1}{R_2}} = \frac{1}{\infty} = 0.$$

## Other uses of $\infty$

We used some of the following:

$$a > 0 : a/0 \rightarrow \infty$$

$$a * \infty \rightarrow \infty,$$

$$a \text{ finite} : a + \infty \rightarrow \infty$$

$$a - \infty \rightarrow -\infty$$

$$a/\infty \rightarrow 0$$

$$\infty + \infty \rightarrow \infty.$$

But  $\infty \text{ REM } a$ ,  $a \text{ REM } 0$ ,  $\infty * 0$ ,  $0/0$ ,  $\infty/\infty$  and  $\infty - \infty$  make no sense. Computing any of these is called an **invalid operation**, and the IEEE standard sets the result to NaN (**Not a Number**). **Any** arithmetic operation on a NaN **also** gives a NaN result.

Whenever a NaN is discovered in the output, the programmer **knows something has gone wrong**.

(An  $\infty$  in the output may or may not indicate an error, depending on the context).

## Logical Expressions

If  $x_k \searrow 0$  we say  $1/x_k \rightarrow \infty$ , but  $x_k \nearrow 0$  implies  $1/x_k \rightarrow -\infty$ . This suggests a need for  $-0$ , so that the similar conventions  $a/0 = \infty$  and  $a/(-0) = -\infty$  may be followed, where  $a > 0$ .

It is essential that the **logical** expression  $\langle 0 = -0 \rangle$  has the value **true** while  $\langle \infty = -\infty \rangle$  has the value **false**. Thus it is possible the **logical expressions**  $\langle a = b \rangle$  and  $\langle 1/a = 1/b \rangle$  have **different** values, namely when  $a = 0$  and  $b = -0$  (or  $a = \infty$ ,  $b = -\infty$ ).

**Q:** In our total resistance example, what is the result of  $T = (R_1^{-1} + R_2^{-1})^{-1}$ , if  $R_1$  is: negative,  $-0$ , or NaN ??

## Arithmetic Comparisons

When  $a$  and  $b$  are finite **real** numbers, one of three conditions holds:  $a = b$ ,  $a < b$ , or  $a > b$ . The same is true if  $a$  and  $b$  are finite **floating point** numbers. In both cases we also have for **finite**  $a$ :  $-\infty < a < \infty$ .

**Note:** `cc` on the Sun 4 with  $a = \infty$  and  $b = \infty$ , gives  $\langle a = b \rangle$  **true**. (**Suspect** mathematically, **Useful** computationally).

However, if either  $a$  or  $b$  has a NaN value, **none** of the three conditions  $=$ ,  $<$ ,  $>$  can be said to hold. Instead,  $a$  and  $b$  are said to be **unordered**.

Consequently, although the **logical expressions**  $\langle a \leq b \rangle$  and  $\langle \text{not}(a > b) \rangle$  usually have the same value, they are **defined** to have **different** values (the first **false**, the second **true**) if either  $a$  or  $b$  is a NaN.

## Overflow and Underflow

**Overflow** is said to occur when

$$N_{max} < | \text{true result} | < \infty,$$

where  $N_{max}$  is the **largest** normalized floating point number.

Two pre-IEEE standard treatments:

- (i) Set the result to  $(\pm) N_{max}$ , or
- (ii) Interrupt with an **error message**.

In IEEE arithmetic, the standard response depends on the **rounding mode**. Suppose that the overflowed value is **positive**. Then **round up** gives the result  $\infty$ , while **round down** and **round towards zero** set the result to  $N_{max}$ . For **round to nearest**, the result is  $\infty$ . From a practical point of view this is important, since **round to nearest** is the **default** rounding mode and any other choice may lead to very misleading final computational results.

## Underflow

**Underflow** is said to occur when

$$0 < | \text{true result} | < N_{min},$$

where  $N_{min}$  is the **smallest** normalized floating point number.

Historically the response was usually:  
**replace the result by zero.**

In **IEEE arithmetic**, the result may be a **sub-normal** number instead of zero. This allows results **much smaller** than  $N_{min}$ .

Table 4: IEEE Standard Response to Exceptions

Invalid Opn.	Set result to NaN
Division by 0	Set result to $\pm\infty$
Overflow	Set to $\pm\infty$ or $\pm N_{max}$
Underflow	Set to zero or to an adjacent subnormal number
Precision or <b>Inexact</b>	Set result to correctly rounded value

The IEEE standard: an **exception** must be **signaled** by setting an associated **status flag**, and the programmer should have the option of either **trapping the exception**, or **masking the exception**, in which case the program continues with the response of the table.

It is usually best to rely on these standard responses.