# Chapter 12
# Floating Point Arithmetic

There are many data processing applications (e.g. image and voice processing), which use a large range of values and that need a relatively high precision. In such cases, instead of encoding the information in the form of integers or fixed-point numbers, an alternative solution is a floating-point representation. In the first section of this chapter, the IEEE standard for floating point is described. The next section is devoted to the algorithms for executing the basic arithmetic operations. The two following sections define the main rounding methods and introduce the concept of guard digit. Finally, the last few sections propose basic implementations of the arithmetic operations, namely addition and subtraction, multiplication, division and square root.

## 12.1 IEEE 754-2008 Standard

The IEEE-754 Standard is a technical standard established by the Institute of Electrical and Electronics Engineers for floating-point operations. There are numerous CPU, FPU and software implementations of this standard. The current version is IEEE 754-2008 [1], which was published in August 2008. It includes nearly all the definitions of the original IEEE 754-1985 and IEEE 854-1987 standards. The main enhancement in the new standard is the definition of decimal floating point representations and operations. The standard defines the arithmetic and interchange formats, rounding algorithms, arithmetic operations and exception handling.

J.-P. Deschamps et al., *Guide to FPGA Implementation of Arithmetic Functions*, Lecture Notes in Electrical Engineering 149, DOI: 10.1007/978-94-007-2987-2_12,

© Springer Science+Business Media Dordrecht 2012

### 12.1.1 Formats

Formats in IEEE 754 describe sets of floating-point data and encodings for interchanging them. This format allows representing a finite subset of real numbers. The floating-point numbers are represented using a triplet of natural numbers (positive integers). The finite numbers may be expressed either in base 2 (binary) or in base 10 (decimal). Each finite number is described by three integers: the *sign* (zero or one), the significand $s$ (also known as coefficient or mantissa), and the exponent $e$. The numerical value of the represented number is $(-1)^{sign} \times s \times B^e$, where $B$ is the base (2 or 10).

For example, if $sign = 1$, $s = 123456$, $e = -3$ and $B = 10$, then the represented number is $-123.456$.

The format also allows the representation of infinite numbers ($+\infty$ and $-\infty$), and of special values, called *Not a Number* (NaN), to represent invalid values. In fact there are two kinds of NaN: qNaN (quiet) and sNaN (signaling). The latter, used for diagnostic purposes, indicates the source of the NaN.

The values that can be represented are determined by the base ($B$), the number of digits of the significand (precision $p$), and the maximum and minimum values $e_{min}$ and $e_{max}$ of $e$. Hence, $s$ is an integer belonging to the range 0 to $B^{p-1}$, and $e$ is an integer such that $e_{min} \le e \le e_{max}$.

For example if $B = 10$ and $p = 7$ then $s$ is included between 0 and 9999999. If $e_{min} = -96$ and $e_{max} = 96$, then the smallest non-zero positive number that can be represented is $1 \times 10^{-101}$, the largest is $9999999 \times 10^{90}$ ($9.999999 \times 10^{96}$), and the full range of numbers is from $-9.999999 \times 10^{96}$ to $9.999999 \times 10^{96}$. The numbers closest to the inverse of these bounds ($-1 \times 10^{-95}$ and $1 \times 10^{-95}$) are considered to be the smallest (in magnitude) normal numbers. Non-zero numbers between these smallest numbers are called subnormal (also denormalized) numbers.

Zero values are finite values whose significand is 0. The sign bit specifies if a zero is +0 (positive zero) or −0 (negative zero).

### 12.1.2 Arithmetic and Interchange Formats

The arithmetic format, based on the four parameters $B$, $p$, $e_{min}$ and $e_{max}$, defines the set of represented numbers, independently of the encoding that will be chosen for storing and interchanging them (Table 12.1). The interchange formats define fixed-length bit-strings intended for the exchange of floating-point data. There are some differences between binary and decimal interchange formats. Only the binary format will be considered in this chapter. A complete description of both the binary and the decimal format can be found in the document of the IEEE 754-2008 Standard [1].

For the interchange of binary floating-point numbers, formats of lengths equal to 16, 32, 64, 128, and any multiple of 32 bits for lengths bigger than 128, are

**Table 12.1** Binary and decimal floating point format in IEEE 754-2008

| Parameter | Binary formats ($B = 2$) | | | | Decimal formats ($B = 10$) | | |
|---|---|---|---|---|---|---|---|
| | Binary 16 | Binary 32 | Binary 64 | Binary 128 | Decimal 132 | Decimal l64 | Decimal 128 |
| $p$, digits | 10 + 1 | 23 + 1 | 52 + 1 | 112 + 1 | 7 | 16 | 34 |
| $e_{max}$ | +15 | +127 | +1023 | +16383 | +96 | +384 | +16,383 |
| $e_{min}$ | −14 | −126 | −1022 | −16382 | −95 | −383 | −16,382 |
| Common name | Half precision | Single precision | Double precision | Quadruple precision | | | |

**Table 12.2** Binary interchange format parameters

| Parameter | Binary16 | Binary32 | Binary64 | Binary128 | Binary{$k$} ($k \geq 128$) |
|---|---|---|---|---|---|
| $k$, storage width in bits | 16 | 32 | 64 | 128 | Multiple of 32 |
| $p$, precision in bits | 11 | 24 | 53 | 113 | $k - w$ |
| $e_{max}$ | 15 | 127 | 1,023 | 16,383 | $2^{(k-p-1)} - 1$ |
| $bias$, $E - e$ | 15 | 127 | 1,023 | 16,383 | $e_{max}$ |
| $w$, exponent field width | 5 | 8 | 11 | 15 | Round(4·$\log_2 k$) − 13 |
| $t$, trailing significand bits | 10 | 23 | 52 | 112 | $k - w - 1$ |

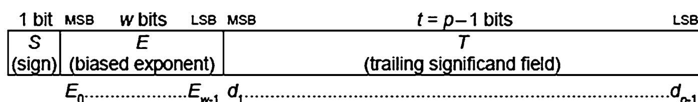


**Fig. 12.1** Binary interchange format

defined (Table 12.2). The 16-bit format is only for the exchange or storage of small numbers.

The binary interchange encoding scheme is the same as in the IEEE 754-1985 standard. The $k$-bit strings are made up of three fields (Fig. 12.1):

- a 1-bit sign S,
- a $w$-bit biased exponent $E = e + bias$,
- the $p - 1$ trailing bits of the significand; the missing bit is encoded in the exponent (hidden first bit).

Each binary floating-point number has just one encoding. In the following description, the significand $s$ is expressed in scientific notation, with the radix point immediately following the first digit. To make the encoding unique, the value of the significand $s$ is maximized by decreasing $e$ until either $e = e_{min}$ or $s \geq 1$ (normalization). After normalization, there are two possibilities regarding the significand:

- If $s \geq 1$ and $e \geq e_{min}$ then a normalized number of the form $1.d_1\ d_2 \ldots d_{p\ -1}$ is obtained. The first “1” is not stored (implicit leading 1).

- If $e = e_{min}$ and $0 < s < 1$, the floating-point number is called subnormal. Subnormal numbers (and zero) are encoded with a reserved biased exponent value. They have an implicit leading significand bit = 0 (Table 12.2).

The minimum exponent value is $e_{min} = 1 - e_{max}$. The range of the biased exponent $E$ is 1 to $2^w - 2$ to encode normal numbers. The reserved value 0 is used to encode $\pm$ 0 and subnormal numbers. The value $2^w - 1$ is reserved to encode $\pm\infty$ and NaNs.

The value of a binary floating-point data is inferred from the constituent fields as follows:

- If $E = 2^w - 1$ (all 1's in $E$), then the data is an NaN or infinity. If $T \neq 0$, then it is a qNaN or an sNaN. If the first bit of $T$ is 1, it is an sNaN. If $T = 0$, then the value is $(-1)^{sign} \times (+\infty)$.
- If $E = 0$ (all 0's in $E$), then the data is 0 or a subnormal number. If $T = 0$ it is a signed 0. Otherwise ($T \neq 0$), the value of the corresponding floating-point number is $(-1)^{Sign} \times 2^{emin} \times (0 + 2^{1-p} \times T)$.
- If $1 \leq E \leq 2^w - 2$, then the data is $(-1)^{sign} \times 2^{(E-bias)} \times (1 + 2^{1-p} \times T)$. Remember that the significand of a normal number has an implicit leading 1.

**Example 12.1**
Convert the decimal number −9.6875 to its binary32 representation.

- First convert the absolute value of the number to binary (Chap. 10): $9.6875_{10} = 1001.1011_2$.
- Normalize: $1001.1011 = 1.00110011 \times 2^3$. Hence $e = 3$, $s = 1.0011\,0011$.
- Hide the first bit and complete with 0's up to 23 bits: 00110011000000000000000.
- Add *bias* to the exponent. In this case, $w = 8$, $bias = 2^8 - 1 = 127$ and thus $E = e + bias = 3 + 127 = 130_{10} = 10000010_2$.
- Compose the final 32-bit representation:
- $1\,10000010\,00110011000000000000000_2 = \mathrm{C1198000}_{16}$.

**Example 12.2**
Convert the following binary32 numbers to their decimal representation.

- $\mathrm{7FC00000}_{16}$: $sig\ n = 0$, $E = \mathrm{FF}_{16}$, $T \neq 0$, hence it is an NaN. Since the first bit of $T$ is 1, it is a quiet NaN.
- $\mathrm{FF800000}_{16}$: $sig\ n = 0$, $E = \mathrm{FF}_{16}$, $T = 0$, hence it is $-\infty$.
- $\mathrm{6545AB78}_{16}$ : $sign = 0, E = \mathrm{CA}_{16}, = 202_{10}, e = E - bias = 202 - 127 = 75_{10}$,

  $T = 10001011010101101111000_2$,

  $s = 1.10001011010101101111000_2 = 1.5442953_{10}$.

  The number is $1.10001011010101101111_2 \times 2^{75} = 5.8341827 \times 10^{22}$.

- $12345678_{16}$ : $sign = 0,\ E = 24_{16} = 36_{10},\ e = E - bias = 36 - 127 = -91_{10}$,

  $T = 01101000101011001111000_2$,

  $s = 1.01101000101011001111000_2 = 1.408888_{10}$.

  The number is 1.01101000101011001111
  $\times 2^{-91} = 5.69045661 \times 10^{-28}$.8000000016: sign = 1, E = 0016, T = 0. The number is −0.0.
- 8000000016: sign = 1, E = 0016, T = 0. The number is −0.0.
- $00012345_{16}$:$sign = 0,\ E = 00_{16},\ T \neq 0$, hence it is a subnormal number, $e = E - bias = -127,\ T = 00000100100011010000101_2$,

  $s = 0.00000100100011010000101_2 = 0.0177777_{10}$.

  The number is $0.00000100100011010000101 \times 2^{-127} = 1.0448782 \times 10^{-40}$.

## 12.2 Arithmetic Operations

First analyze the main arithmetic operations and generate the corresponding computation algorithms. In what follows it will be assumed that the significand $s$ is represented in base $B$ (in binary if $B = 2$, in decimal if $B = 10$) and that it belongs to the interval $1 \leq s \leq B - ulp$, where $ulp$ stands for the unit in the last position or unit of least precision. Thus $s$ is expressed in the form $(s_0 \cdot s_{-1} \cdot s_{-2} \ldots \cdot s_{-p})$. $B^e$ where $e_{min} \leq e \leq e_{max}$ and $1 \leq s_0 \leq B - 1$.

**Comment 12.1**
The binary subnormals and the decimal floating point are not normalized numbers and are not included in the following analysis. This situation deserves some special treatment and is out of the scope of this section.

### *12.2.1 Addition of Positive Numbers*

Given two positive floating-point numbers $s_1 \cdot B^{e1}$ and $s_2 \cdot B^{e2}$ their sum $s \cdot B^e$ is computed as follows: assume that $e_1$ is greater than or equal to $e_2$; then (*alignment*) the sum of $s_1 \cdot B^{e1}$ and $s_2 \cdot B^{e2}$ can be expressed in the form $s \cdot B^e$ where

$$s = s_1 + s_2/\left(B^{e1-e2}\right) \text{ and } e = e_1. \tag{12.1}$$

The value of $s$ belongs to the interval

$$1 \leq s \leq 2 \cdot B - 2 \cdot ulp, \tag{12.2}$$

so that $s$ could be greater than or equal to $B$. If it is the case, that is if

$$B \leq s \leq 2 \cdot B - 2 \cdot ulp, \tag{12.3}$$

then (*normalization*) substitute $s$ by $s/B$, and $e$ by $e + 1$, so that the value of $s \cdot B^e$ is the same as before, and the new value of $s$ satisfies

$$1 \le s \le 2 - (2/B) \cdot ulp \le B - ulp. \tag{12.4}$$

The significands $s_1$ and $s_2$ of the operands are multiples of *ulp*. If $e_1$ is greater than $e_2$, the value of $s$ could no longer be a multiple of *ulp* and some rounding function should be applied to $s$. Assume that $s' < s < s'' = s' + ulp$, $s'$ and $s''$ being two successive multiples of *ulp*. Then the *rounding* function associates to $s$ either $s'$ or $s''$, according to some rounding strategy. According to (12.4) and to the fact that 1 and $B—ulp$ are multiples of *ulp*, it is obvious that $1 \le s' < s'' \le B—ulp$. Nevertheless, if the condition (12.3) does not hold, that is if $1 \le s < B$, $s$ could belong to the interval

$$B - ulp < s < B, \tag{12.5}$$

so that *rounding*($s$) could be equal to $B$, then a new *normalization step* would be necessary, i.e. substitution of $s = B$ by $s = 1$ and $e$ by $e + 1$.

**Algorithm 12.1: Sum of positive numbers**

```
if e1 ≥ e2 then e := e1; s := s1 + s2/B^(e1-e2);
else e := e2; s := s1/B^(e2-e1) + s2; end if;
if s ≥ B then e := e+1; s := s/B; end if; --1st normalization
s := round(s);
if s ≥ B then e := e+1; s := s/B; end if; --2nd normalization
```

**Examples 12.3**

Assume that $B = 2$ and $ulp = 2^{-5}$, so that the numbers are represented in the form $s \cdot 2^e$ where $1 \le s \le 1.11111_2$. For simplicity $e$ is written in decimal (base 10).

1. Compute $z = \left(1.10101 \times 2^3\right) + \left(1.00010 \times 2^{-1}\right)$.

   Alignment: $z = (1.10101 + 0.000100010) \times 2^3 = 1.101110010 \times 2^3$.

   Rounding: $s \cong 1.10111$.

   Final result: $z \cong 1.10111 \times 2^3$.

2. Compute $z = (1.11010 \times 2^3) + (1.00110 \times 2^2)$.

   Alignment: $z = (1.11010 + 0.100110) \times 2^3 = 10.011010 \times 2^3$.

   Normalization : $s = 1.0011010,\ e = 4$.

   Rounding: $s \cong 1.00110$.

   Final result: $z \cong 1.00110 \times 2^4$.

3. Compute $z = \left(1.10101 \times 2^3\right) + \left(1.10101 \times 2^1\right)$.

   Alignment: $z = (1.10010 + 0.0110101) \times 2^3 = 1.1111101 \times 2^3$.

   Rounding: $s \cong 10.00000$.

   Normalization: $s \cong 1.00000, e = 4$.

   Final result : $z \cong 1.00000 \times 2^4$.

**Comments 12.2**

1. The addition of two positive numbers could produce an *overflow* as the final value of $e$ could be greater than $e_{max}$.
2. Observe in the previous examples the lack of precision due to the small number of bits (6 bits) used in the significand $s$.

### *12.2.2 Difference of Positive Numbers*

Given two positive floating-point numbers $s_1 \cdot B^{e1}$ and $s_2 \cdot B^{e2}$ their difference $s \cdot B^e$ is computed as follows: assume that $e_1$ is greater than or equal to $e_2$; then (*for alignment*) the difference between $s_1 \cdot B^{e1}$ and $s_2 \cdot B^{e2}$ can be expressed in the form $s \cdot B^e$ where

$$s = s_1 - s_2/\left(B^{e1-e2}\right) \text{ and } e = e_1. \tag{12.6}$$

The value of $s$ belongs to the interval $-s \cdot -(B - ulp) \leq s \leq B - ulp$. If $s$ is negative, then it is substituted by and the sign of the final result will be modified accordingly. If $s$ is equal to 0, then an exception *equal_zero* could be raised. It remains to consider the case where $0 < s \leq B - ulp$. The value of $s$ could be smaller than 1. In order to normalize the significand, $s$ is substituted by $s \cdot B^k$ and $e$ by $e - k$, where $k$ is the minimum exponent $k$ such that $s \cdot B^k \geq 1$. Thus, the relation $1 \leq s \leq B$ holds. It remains to round (up or down) the significand and to normalize it if necessary.

In the following algorithm, the function *leading_zeroes(s)* computes the smallest $k$ such that $s \cdot B^k \geq 1$.

**Algorithm 12.2: Difference of positive numbers**

```
if e1 ≥ e2 then e := e1; s := s1 - s2/B^(e1-e2);
else e := e2; s := s1/B^(e2-e1) - s2; end if;
if s < 0 then s := -s; sign := 1; end if;
k := leading_zeroes(s);
s := s·B^k; e := e-k; --1st norm.
s := round(s);
if s ≥ B then e := e+1; s := s/B; end if; --2nd norm.
```

**Examples 12.4**
Assume again that $B = 2$ and $ulp = 2^{-5}$, so that the numbers are represented in the form $s \cdot 2^e$ where $1 \leq s \leq 1.11111_2$. For computing the difference, the 2's complement representation is used (one extra bit is used).

1. Compute $z = (1.10101 \times 2^{-2}) - \left(1.01010 \times 2^1\right)$.

   Alignment: $z = (0.00110101 - 1.01010) \times 2^1$.

   $2'$s complement addition: $(00.00110101 + 10.10101 + 00.00001) \times 2^1 =$

   $10.11100101 \times 2^1$.

   Change of sign: $-s = 01.00011010 + 00.00000001 = 01.00011011$.

   Rounding: $-s \cong 1.00011$.

   Final result: $z \cong -1.00011 \times 2^1$.

2. Compute $z = \left(1.00010 \times 2^3\right) - \left(1.10110 \times 2^2\right)$.

   Alignment: $z = (1.00010 - -0.110110) \times 2^3$.

   $2'$s complement addition: $(01.00010 + 11.001001 + 00.000001) \times 2^3 =$

   $00.001110 \times 2^3$.

   Leading zeroes: $k = 3$, $s = 1.11000$, $e = 0$.

   Final result: $z = 1.11000 \times 2^0$.

3. Compute $z = \left(1.01010 \times 2^3\right) - \left(1.01001 \times 2^1\right)$.

   Alignment: $z = (1.01010 - -0.0101001) \times 2^3 = 0.1111111 \times 2^3$.

   Leading zeroes: $k = 1$, $s = 1.111111$, $e = 2$.

   Rounding: $s \cong 10.00000$.

   Normalization: $s \cong 1.0000$, $e = 3$.

   Final result: $z \cong 1.00000 \times 2^3$.

**Comment 12.3**
The difference of two positive numbers could produce an *underflow* as the final value of $e$ could be smaller than $e_{min}$.

**Table 12.3** Effective operation in floating point adder–subtractor

| Operation | $Sign_1$ | $Sign_2$ | Actual operation |
|---|---|---|---|
| 0 | 0 | 0 | $s_1 + s_2$ |
| 0 | 0 | 1 | $s_1 - s_2$ |
| 0 | 1 | 0 | $-(s_1 - s_2)$ |
| 0 | 1 | 1 | $-(s_1 - s_2)$ |
| 1 | 0 | 0 | $s_1 - s_2$ |
| 1 | 0 | 1 | $s_1 + s_2$ |
| 1 | 1 | 0 | $-(s_1 + s_2)$ |
| 1 | 1 | 1 | $-(s_1 - s_2)$ |

### 12.2.3 Addition and Subtraction

Given two floating-point numbers $(-1)^{sign1} \cdot s_1 \cdot B^{e1}$ and $(-1)^{sign2} \cdot s_2 \cdot B^{e2}$, and a control variable *operation*, an algorithm is defined for computing

$$z = (-1)^{sign} \cdot s \cdot B^{e} = (-1)^{sign1} \cdot s_1 \cdot B^{e1} + (-1)^{sign2} \cdot s_2 \cdot B^{e2}, \quad \text{if } operation = 0,$$
$$z = (-1)^{sign} \cdot s \cdot B^{e} = (-1)^{sign1} \cdot s_1 \cdot B^{e1} - (-1)^{sign2} \cdot s_2 \cdot B^{e2}, \quad \text{if } operation = 1.$$

Once the significands have been aligned, the actual operation (addition or subtraction of the significands) depends on the values of *operation*, $sign_1$ and $sign_2$ (Table 12.3). The following algorithm computes $z$. The procedure *swap* (*a, b*) interchanges *a* and *b*.

**Algorithm 12.3: Addition and subtraction**

```
if operation = 1 then sign2 := 1 - sign2; end if;
if e1 < e2 then
  swap(sign1, sign2); swap(s1, s2); swap (e1, e2);
end if;
e := e1; s2 := s2/B^(e1-e2); sign := sign1;
if sign1 xor sign2 = 0 then --addition
  s := s1 + s2;
  if s ≥ B then e := e+1; s := s/B; end if;
else --subtraction
  if (e1 = e2) and (s1 < s2) then
    swap(s1, s2); sign := 1 - sign;
  end if;
  s := s1 - s2; k := leading_zeroes(s);
  s := s·B^k; e := e - k;
end if;
s := round(s);
if s ≥ B then e := e+1; s := s/B; end if;
```

### *12.2.4 Multiplication*

Given two floating-point numbers $(-1)^{sign1} \cdot s_1 \cdot B^{e1}$ and $(-1)^{sign2} \cdot s_2 \cdot B^{e2}$ their product $(-1)^{sign} \cdot s \cdot B^e$ is computed as follows:

$$sign = sign_1 \, xor \, sign_2, \; s = s_1 \cdot s_2, \; e = e_1 + e_2. \tag{12.7}$$

The value of $s$ belongs to the interval $1 \le s \le (B - ulp)^2$, and could be greater than or equal to $B$. If it is the case, that is if $B \le s \le (B - ulp)^2$, then (*normalization*) substitute $s$ by $s/B$, and $e$ by $e + 1$. The new value of $s$ satisfies

$$1 \le s \le (B - ulp)^2/B = B - 2.ulp + (ulp)^2/B < B - -ulp \tag{12.8}$$

($ulp < B$ so that $2 - ulp/B > 1$). It remains to round the significand and to normalize if necessary.

**Algorithm 12.4: Multiplication**

```
sign := sign1 xor sign2; s := s1·s2; e := e1 + e2;
if s ≥ B then e := e+1; s := s/B; end if; --1st normalization
s := round(s);
if s ≥ B then e := e+1; s := s/B; end if; --2nd normalization
```

**Examples 12.5**
Assume again that $B = 2$ and $ulp = 2^{-5}$, so that the numbers are represented in the form $s \cdot 2^e$ where $1 \le s \le 1.11111_2$. The exponent $e$ is represented in decimal.

1. Compute $z = \left(1.11101 \times 2^{-2}\right) \times \left(1.00010 \times 2^5\right)$.

   Multiplication : $z = 01.1100101100 \times 2^3$.

   Rounding : $s \cong 1.11001$.

   Final result : $z \cong 1.11001 \times 2^3$.

2. Compute $z = \left(1.11101 \times 2^3\right) \times (1.00011 \times 2^{-1})$.

   Multiplication : $z = 10.0001010111_2 \times 2^2$.

   Normalization : $s = 1.00001010111_2, \quad e = 3$.

   Rounding : $s \cong 1.00001$.

   Final result : $z \cong 1.00001 \times 2^3$.

3. Compute $z = \left(1.01000 \text{ x } 2^1\right) \times \left(1.10011 \times 2^2\right)$.

   Multiplication : $z = 01.111111000 \times 2^2$.

   Normalization : $s = 1.11111, \quad e = 3$.

   Rounding : $s \cong 10.00000$.

   Normalization : $s \cong 1, \quad e = 4$.

   Final result : $z \cong 1.00000_2 \times 2^4$.

**Comment 12.4**
The product of two real numbers could produce an *overflow* or an *underflow* as the final value of $e$ could be greater than $e_{max}$ or smaller than $e_{min}$ (addition of two negative exponents).

### *12.2.5 Division*

Given two floating-point numbers $(-1)^{sign1} \cdot s_1 \cdot B^{e1}$ and $(-1)^{sign2} \cdot s_2 \cdot B^{e2}$ their quotient
$(-1)^{sign} \cdot s \cdot B^e$ is computed as follows:

$$sign = sign_1 \text{ } xor \text{ } sign_2, \quad s = s_1/s_2, \quad e = e_1 - e_2. \tag{12.9}$$

The value of $s$ belongs to the interval $1/B < s \le B - ulp$, and could be smaller than 1. If that is the case, that is if $s = s_1/s_2 < 1$, then $s_1 < s_2, s_1 \le s_2 - ulp, \ s_1/s_2 \le 1 - ulp/s_2 < 1 - ulp/B$, and $1/B < s < 1 - ulp/B$.

Then (*normalization*) substitute $s$ by $s \cdot B$, and $e$ by $e - 1$. The new value of $s$ satisfies $1 < s < B - ulp$. It remains to round the significand.

**Algorithm 12.5: Division**

```
sign := sign1 xor sign2; s := s1/s2; e := e1 - e2;
if s < 1 then e := e-1; s := s·B; end if;
s := round(s);
```

**Examples 12.6**
Assume again that $B = 2$ and $ulp = 2^{-5}$, so that the numbers are represented in the form $s \bullet 2^e$ where $1 \le s \le 1.11111_2$. The exponent $e$ is represented in decimal.

1. Compute $z = \left(1.11101 \times 2^3\right) / \left(1.00011 \times 2^{-1}\right)$.

   Division: $z = 1.1011111000 \times 2^4$.

   Rounding: $s \cong 1.10111$.

   Final result: $z \cong 1.00001 \times 2^3$.

2. Compute $z = \left(1.01000 \times 2^1\right) / \left(1.10011 \times 2^2\right)$.

   Division: $z = 0.1100100011 \times 2^{-1}$.

   Normalization: $s \cong 1.100100011, \quad e = -2$.

   Rounding: $s \cong 1.10010$.

   Final result: $z \cong 1.10010 \times 2^{-2}$.

**Comment 12.5**
The quotient of two real numbers could produce an *underflow* or an *overflow* as the final value of $e$ could be smaller than $e_{min}$ or bigger than $e_{max}$. Observe that a second normalization is not necessary as in the case of addition, subtraction and multiplication.

## *12.2.6 Square Root*

Given a positive floating-point number $s_1 \cdot B^{e1}$, its square root $s \cdot B^e$ is computed as follows:

$$\text{if } e_1 \text{ is even}, \quad s = (s_1)^{1/2}, \quad e = e_1/2, \tag{12.10}$$

$$\text{if } e_1 \text{ is odd}, \quad s = (s_1/B)^{1/2}, \quad e = (e_1 + 1)/2. \tag{12.11}$$

In the first case (12.10), $1 \le s \le (B - ulp)^{1/2} < B - ulp$. In the second case $(1/B)^{1/2} \le s < 1$. Hence (*normalization*) $s$ must be substituted by $s \cdot B$ and $e$ by $e - 1$, so that $1 \le s < B$. It remains to round the significand and to normalize if necessary.

**Algorithm 12.6: Square root**

```
if (e1  mod 2) = 1 then s1 := s1/B; e1 := e1+1; end if;
s := square_root(s1); e := e1/2;
if s < 1 then e := e-1; s := s·B; end if; --1st norm.
s := round(s);
if s ≥ B then e := e+1; s := s/B; end if; --2nd norm.
```

An alternative is to replace (12.11) by:

$$\text{if } e_1 \text{ is odd}, \quad s = (s_1 \cdot B)^{1/2}, \quad e = (e_1 - 1)/2. \tag{12.12}$$

In this case $B^{1/2} \le s \le (B^2 - ulp \cdot B)^{1/2} < B$, then the first normalization is not necessary. Nevertheless, $s$ could be $B - ulp < s < B$, and then depending on the rounding strategy, normalization after rounding could be still necessary.

**Algorithm 12.7: Square root, second version**

```
if (e1 mod 2) = 1 then s1 := s1 * B; e1 := e1 - 1; end if;
s := square_root(s1); e := e1/2;
s := round(s);
if s ≥ B then e := e+1; s := s/B; end if; --2nd normalization
```

Note that the "round to nearest" (default rounding in IEEE 754-2008) and the "truncation" rounding schemes allow avoiding the second normalization.

**Examples 12.7**

Assume again that $B = 2$ and $ulp = 2^{-5}$, so that the numbers are represented in the form $s \cdot 2^e$ where $1 \le s \le 1.11111_2$. The exponent $e$ is represented in decimal form.

1. Compute $z = \left(1.11101 \times 2^4\right)^{1/2}$.

   Square rooting : $z = 1.01100001 \times 2^2$.

   Rounding : $s \cong 1.01100$.

   Final result : $z \cong 1.01100 \times 2^2$.

2. Compute $z = (1.00101 \times 2^{-1})^{1/2}$.

   Even exponent : $s = 10.0101, \quad e = -2$.

   Square rooting : $z = 1.10000101 \times 2^{-1}$.

   Rounding : $s \cong 1.10000$

   Final result : $z \cong 1.10000 \times 2^{-1}$.

3. Compute $z = \left(1.11111 \times 2^3\right)^{1/2}$.

   Even exponent; $s = 11.1111, \quad e = 2$.

   Square rooting : $z = 1.11111011 \times 2^1$.

   Rounding : $s \cong 1.11111$(round to nearest).

   Final result : $z \cong 1.11111_2 \times 2^1$.

However, some rounding schemes (e.g. toward infinite) generate $s \cong 10.00000$. Then, the result after normalization is $s \cong 1.00000$, $e = 2$, and the final result $z \cong 1.00000 \times 2^2$.

**Comment 12.6**
The square rooting of a real number could produce an *underflow* as the final value of $e$ could be smaller than $e_{min}$.

## 12.3 Rounding Schemes

Given a real number $x$ and a floating-point representation system, the following situations could happen:

$|x| < s_{min} \cdot B^{e_min}$, that is, an *underflow* situation,

$|x| > s_{max} \cdot B^{e_max}$, that is, an *overflow* situation,

$|x| = s \cdot B^e$, where $e_{min} \leq e \leq e_{max}$ and $s_{min} \leq s \leq s_{max}$.

In the third case, either $s$ is a multiple of *ulp*, in which case a rounding operation is not necessary, or it is included between two multiples $s'$ and $s''$ of *ulp*: $s' < s < s''$.

The *rounding* operation associates to $s$ either $s'$ or $s''$, according to some rounding strategy. The most common are the following ones:

- The *truncation* method (also called *round toward* 0 or *chopping*) is accomplished by dropping the extra digits, i.e. $round(s) = s'$ if $s$ is positive, $round(-s) = s''$ if $s$ is negative.
- The round toward plus infinity is defined by round(s) = $s''$.
- The round toward minus infinity is defined by round(s) = s$'$.
- The *round to nearest* method associates $s$ with the closest value, that is, if $s < s' + ulp/2$, $round(s) = s'$, and if $s > s' + ulp/2$, $round(s) = s''$.

If the distances to $s'$ and $s''$ are the same, that is, if $s = s' + ulp/2$, there are several options. For instance:

- $round(s) = s'$;
- $round(s) = s''$;
- $round(s) = s'$ if $s$ is positive, $round(s) = s''$ if $s$ is negative. It is the *round to nearest, ties to zero* scheme.
- $round(s) = s''$ if $s$ is positive, $round(s) = s'$if $s$ is negative. It is the *round to nearest, ties away from zero* scheme.
- $round(s) = s'$ if $s'$ is an even multiple of *ulp*, $round(s) = s''$ if $s''$ is an even multiple of *ulp*. It is the default scheme in the IEEE 754 standard.
- $round(s) = s'$ if $s'$ is an odd multiple of *ulp*, $round(s) = s''$ if $s''$ is an odd multiple of *ulp*.

The preceding schemes (*round to the nearest*) produce the smallest absolute error, and the two last (*tie to even, tie to odd*) also produce the smallest average absolute error (*unbiased* or 0-*bias* representation systems).

Assume now that the exact result of an operation, after normalization, is

$$s = 1.s_{-1}\, s_{-2}\, s_{-3} \ldots s_{-p} | s_{-(p+1)}\, s_{-(p+2)}\, s_{-(p+3)} \cdots$$

where *ulp* is equal to $B^{-p}$ (the | symbol indicates the separation between the digit which corresponds to the *ulp* and the following). Whatever the chosen rounding scheme, it is not necessary to have previously computed all the digits $s_{-(p+1)}\, s_{-(p+2)} \ldots$; it is sufficient to know whether all the digits $s_{-(p+1)}\, s_{-(p+2)} \ldots$ are equal to 0, or not. For example the following algorithm computes *round*($s$) if the *round to the nearest*, *tie to even* scheme is used.

**Algorithm 12.8**: **Round to the nearest, tie to even**

```
s1 := 1.s-1 s-2 ... s-p;
s2 := s - s1 - s-(p+1)·ulp/B;
--s2 = 0.00 ... 0|0 s-(p+2) s-(p+3) ...
if s-(p+1) < B/2 then round := s1;
elsif s-(p+1) > B/2 then round := s1 + ulp;
elsif s-(p+1) = B/2 and s2 > 0 then round := s1 + ulp;
elsif s-(p+1) = B/2 and s2 = 0 and (s-p mod 2) = 0 then
     round := s1;
elsif s-(p+1) = B/2 and s2 = 0 and (s-p mod 2) = 1 then
     round := s1 + ulp;
end if;
```

In order to execute the preceding algorithm it is sufficient to know the value of $s_1 = 1 \cdot s_{-1}\, s_{-2}\, s_{-3} \ldots s_{-p}$, the value of $s_{-(p+1)}$, and whether $s_2 = 0.00 \ldots 0|0\, s_{-(p+2)} s_{-(p+3)} \cdots$ is equal to 0, or not.

### 12.3.1 Rounding Schemes in IEEE 754

From the previous description, the IEEE 754-2008 standard defines five rounding algorithms. The two first round to a nearest value; the others are called *directed roundings*:

- *Round to nearest, ties to even*; this is the default rounding for binary floating-point and the recommended default for decimal.
- *Round to nearest, ties away from zero*.
- *Round toward* 0—directed rounding towards zero.
- *Round toward* $+\infty$—directed rounding towards positive infinity
- *Round toward* $-\infty$—directed rounding towards negative infinity.

## 12.4 Guard Digits

Consider the exact result $r$ of an operation, before normalization. According to the preceding paragraph:

$$r < B^2,\ \text{i.e.}\ r = r_1\, r_0\ \cdot\ r_{-1}\, r_{-2}\, r_{-3} \ldots r_{-p} | r_{-(p+1)}\, r_{-(p+2)}\, r_{-(p+3)} \ldots$$

The normalization operation (if necessary) is accomplished by

- dividing the result by $B$ (sum of positive numbers, multiplication),
- multiplying the result by $B$ (division),
- multiplying the result by $B^k$ (difference of positive numbers).

Furthermore, if the operation is a difference of positive numbers (Algorithm 12.2), consider two cases:

- if $e_1 - e_2 \geq 2$, then $r = s_1 - s_2/(B^{e1-e2}) > 1 - B/B^2 = 1 - 1/B \geq 1/B (\text{as}\, B \geq 2)$, so that the number $k$ of leading zeroes is equal to 0 or 1, and the normalization operation (if necessary i.e. $k = 1$) is accomplished by multiplying the result by $B$;
- if $e_1 - e_2 \leq 1$, then the result before normalization is either

$$r_0\ \cdot\ r_{-1}\, r_{-2}\, r_{-3} \ldots r_{-p} | r_{-(p+1)} 0\, 0 \ldots (e_1 - e_2 = 1),\ \text{or}$$
$$r_0\ \cdot\ r_{-1}\, r_{-2}\, r_{-3} \ldots r_{-p} | 0\, 0\, 0 \ldots (e_1 - e_2 = 0).$$

A consequence of the preceding analysis is that the result after normalization can be either

$$r_0\ \cdot\ r_{-1}\, r_{-2}\, r_{-3} \ldots r_{-p} | r_{-(p+1)}\, r_{-(p+2)}\, r_{-(p+3)} \ldots (\text{no normalization operation}), \tag{12.13}$$

or

$$r_1\ \cdot\ r_0\, r_{-1}\, r_{-2} \ldots r_{-p+1} | r_{-p}\, r_{-(p+1)}\, r_{-(p+2)} \ldots\ (\text{divide by}\, B), \tag{12.14}$$

or

$$r_{-1}\ \cdot\ r_{-2}\, r_{-3}\, r_{-4} \ldots r_{-(p+1)} | r_{-(p+2)}\, r_{-(p+3)}\, r_{-(p+4)} \ldots (\text{multiply by}\, B), \tag{12.15}$$

or

$$r_{-k}\ \cdot\ r_{-(k+1)}\, r_{-(k+2)} \ldots r_{-p}\, r_{-(p+1)} 0 \ldots 0 | 0\, 0 \ldots (\text{multiply by}\, B^k \text{where}\, k > 1). \tag{12.16}$$

For executing a rounding operation, the worst case is (12.15). In particular, for executing Algorithm 12.8, it is necessary to know

- the value of $s_1 = r_{-1} \cdot r_{-2}\, r_{-3}\, r_{-4} \ldots r_{-(p+1)}$,
- the value of $r_{-(p+2)}$,
- whether $s_2 = 0.00\ldots 0|0\, r_{-(p+3)}\, r_{-(p+4)} \ldots$ is equal to 0, or not.

The conclusion is that the result $r$ of an operation, before normalization, must be computed in the form

$$r \cong r_1\, r_0 \cdot r_{-1}\, r_{-2}\, r_{-3} \ldots r_{-p}|r_{-(p+1)}\, r_{-(p+2)}\, T,$$

that is, with two *guard digits* $r_{-(p+1)}$ and $r_{-(p+2)}$-, and an additional *sticky digit* $T$ equal to 0 if all the other digits $\left(r_{-(p+3)}, r_{-(p+4)}, \ldots\right)$ are equal to 0, and equal to any positive value otherwise.

After normalization, the significand will be obtained in the following general form:

$$s \cong 1.s_{-1}\, s_{-2}\, s_{-3} \ldots s_{-p}|s_{-(p+1)}\, s_{-(p+2)}\, s_{-(p+3)}.$$

The new version of Algorithm 12.8 is the following:

**Algorithm 12.9: Round to the nearest, tie to even, second version**

```
s1 := 1.s-1 s-2 ... s-p;
if s-(p+1) < B/2 then round := s1;
elsif s-(p+1) > B/2 then round := s1 + ulp;
elsif s-(p+2) > 0 or s-(p+3) >0 then round := s1 + ulp;
elsif (s-p mod 2) = 0 then round := s1;
else round := s1 + ulp;
end if;
```

Observe that in binary representation, the following algorithm is even simpler.

**Algorithm 12.10: Round to the nearest, tie to even, third version**

```
s1 := 1.s-1 s-2 ... s-p;
if s-(p+1) = '0' then round := s1;
elsif s-p-1 = '1' and (s-p-2 = '1' or s-p-3 = '1') then
  round := s1 + ulp;
elsif s-p='0' then round := s1; --s-p-1:-p-3 = "100"
else round := s1 + ulp;
end if;
```

## 12.5 Arithmetic Circuits

This section proposes basic implementations of the arithmetic operations, namely addition and subtraction, multiplication, division and square root. The implementation is based on the previous section devoted to the algorithms, rounding and guard digit.

The circuits support normalized binary IEEE 754-2008 operands. Regarding the binary subnormals the associated hardware to manage this situation is complex. Some floating point implementations solve operations with subnormals via software routines. In the FPGA arena, most cores do not support denormalized numbers. The dynamic range can be increased using fewer resources by increasing the size of the exponent (a 1-bit increase in exponent, roughly doubles the dynamic range) and is typically the solution adopted.

## 12.5.1 Adder–Subtractor

An adder-subtractor based on Algorithm 12.3 will now be synthesized. The operands are supposed to be in IEEE 754 binary encoding. It is made up of five parts, namely unpacking, alignment, addition, normalization and rounding, and packing. The following implementation does not support subnormal numbers; they are interpreted as zero.

### 12.5.1.1 Unpacking

The *unpacking* separates the constitutive parts of the Floating Points and additionally detects the special numbers (infinite, zeros and NaNs). The special number detection is implemented using simple comparators. The following VHDL process defines the unpacking of a floating point operand *FP*; *k* is the number of bits of *FP*, *w* is the number of bits of the exponent, and *p* is the significand precision.

```
sign := FP(k-1); e := FP(k-2..k-w-1); s := '1' & FP(p-2..0);
exp_FF := '1' when e = ALL_ONES else '0';
exp_Z := '1' when e = ALL_ZEROS else '0';
sig_Z := '1' when FP(p-2..0) = ALL_ZEROS;
isNaN := exp_FF and (not sig_Z);
isInf := exp_FF and sig_Z;
isZero := exp_Z and sig_Z;
isSubn := exp_Z and (not sig_Z);
```

The previous is implemented using two *w* bits comparators, one *p* bits comparator and some additional gates for the rest of the conditions.

### 12.5.1.2 Alignment

The alignment circuit implements the three first lines of the Algorithm 12.3, i.e.

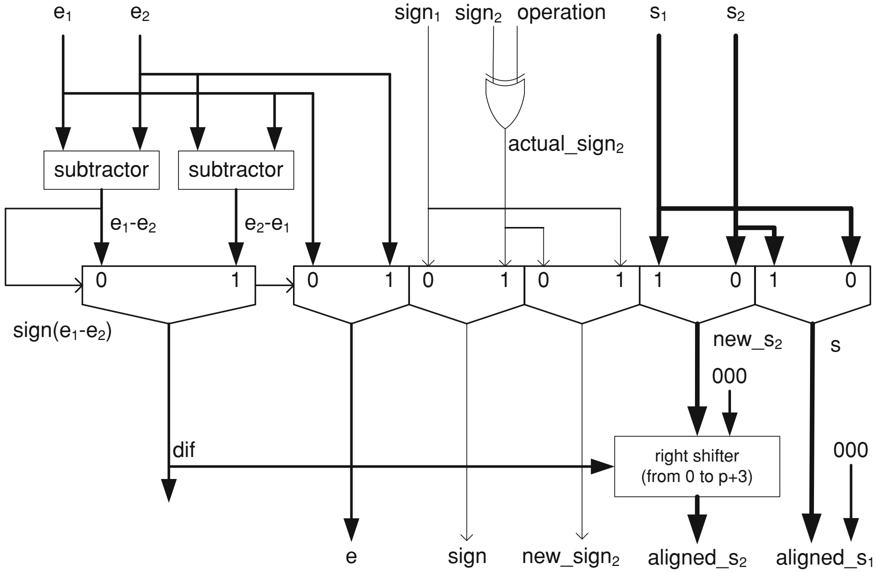


**Fig. 12.2** Alignment circuit in floating point addition/subtraction

```
if operation = 1 then sign2 := 1 - sign2; end if;
if e1 < e2 then
   swap(sign1, sign2); swap(s1, s2); swap (e1, e2);
end if;
e := e1; s2 := (s2 / B**(e1-e2); sign := sign1;
```

An example of implementation is shown in Fig. 12.2. The principal and more complex component is the right shifter.

Given a $(p + 3)$-bit input vector $[1\, a_{k-1}\, a_{k-2} \ldots a_1\, a_0\, 0\, 0\, 0]$, the shifter generates a $(p + 3)$-bit output vector. The *lsb* (least significand bit) of the output is the sticky bit indicating if the shifted bits are equal to zero or not. If $B = 2$, the sticky-digit circuit is an OR circuit.

Observe that if $e_1 - e_2 \geq p + 3$, then the shifter output is equal to [0 0… 0 1], since the last bit is the sticky bit and the input number is a non-zero. The operation with zero is treated differently.

### 12.5.1.3 Addition and Subtraction

Depending on the respective signs of the aligned operands, one of the following operations must be executed: if they have the same sign, the sum *aligned_*$s_1$ + *aligned_*$s_2$ must be computed; if they have different signs, the difference

*aligned_s₁—aligned_s₂* is computed. If the result of the significand difference is negative, then *aligned_s₂—aligned_s₁* must be computed. Moreover, the only situation in which the final result could be negative is when the $e_1 = e_2$.

In the circuit of Fig. 12.3 two additions are performed in parallel: *result = aligned_s₁ ± aligned_s₂*, where the actual operation is selected with the signs of the operands, and *alt_result* $= s_2 - s_1$. At the end of this stage a multiplexer selects the correct results. The operation selection is done as follows:

```
significand :=
  '0' & s1 & "000" when isZero2
  else '0' & s2 & "000" when isZero1
  else alt_result when alt_result > 0 and e1 = e2 and isSUB
  else result;
```

#### 12.5.1.4 Normalization and Rounding

The normalization circuit executes the following part of Algorithm 12.3:

```
if sign1 xor sign2 = 0 then --addition
   s := s1 + s2;
  if s ≥ B then e := e+1; s := s/B; end if;
else --subtraction
  if (e1 = e2) and (s1 < s2) then
    swap(s1, s2); sign := 1 - sign;
  end if;
  s := s1 - s2; k := leading_zeroes(s);
  s := s·B^k; e := e - k;
end if;
```

If the number of leading zeroes is greater than $p + 3$, i.e. $s_1 - s_2 > B^{-(p+2)}$, then $s_2 > s_1 - B^{-(p+2)}$. If $e_1$ were greater than $e_2$ then $s_2 \leq (B - ulp)/B = 1 - B^{-(p+1)}$ so that $1 - B^{-(p+1)} \geq s_2 < s_1 - B^{-(p+2)} \geq 1 - B^{-(p+2)}$, which is impossible. Thus, the only case where the number of leading zeroes can be greater than $p + 3$ is when $e_1 = e_2$ and $s_1 = s_2$. If more than $p + 3$ leading 0's are detected in the circuit of Fig. 12.4, a zero_flag is raised.

It remains to execute the following algorithm where *operation* is the internal operation computed in Fig. 12.3:

```
if operation = 0 then
  if s ≥ B then e := e+1; s := s/B; end if;
else
  k := leading_zeroes(s); s := s·B^k; e := e - k;
end if;
```

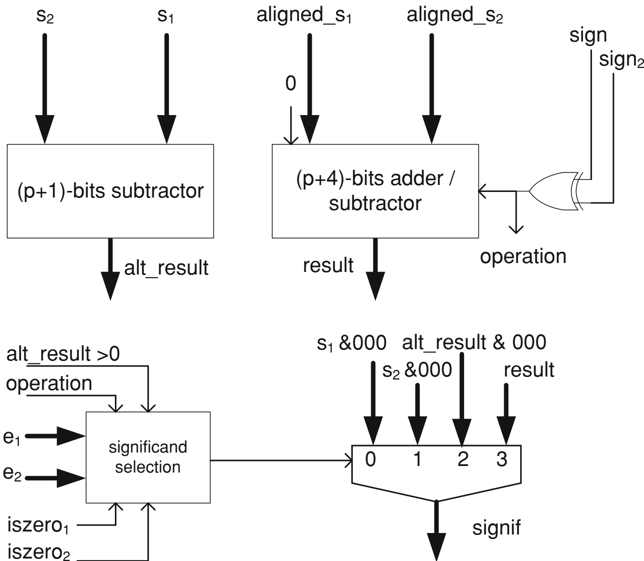


**Fig. 12.3** Effective addition and subtraction

The rounding depends on the selected rounding strategy. An example of *rounding* circuit implementation is shown in Fig. 12.4. If the *round to the nearest, tie to even* method is used (Algorithm 12.10), the block named *rounding decision* computes the following Boolean function *decision*:

```
isRoundUp := '1' when ((s2 = '1' and (s1 = '1' or s0 =
'1')) or s3..0 = "1100") else '0';
```

The second rounding is avoided in the following way. The only possibility to need a second rounding is when, as the result of an addition, the significand is 1.111…1111xx. This situation is detected in a combinational block that generates the signal "isTwo" and adds one to the exponent. After rounding, the resulting number is 10.000…000, but the two most significand bits are discarded and the hidden 1 is appended.

#### 12.5.1.5 Packing

The *packing* joins the constitutive parts of the floating point result. Additionally depending on special cases (infinite, zeros and NaNs), generates the corresponding codification.

**Example 12.8** (Complete VHDL code available)
Generate the VHDL model of an IEEE decimal floating-point adder-subtractor. It is made up of five previously described blocks. Fig. 12.5 summarizes the interconnections. For clearness and reusability the code is written using parameters,

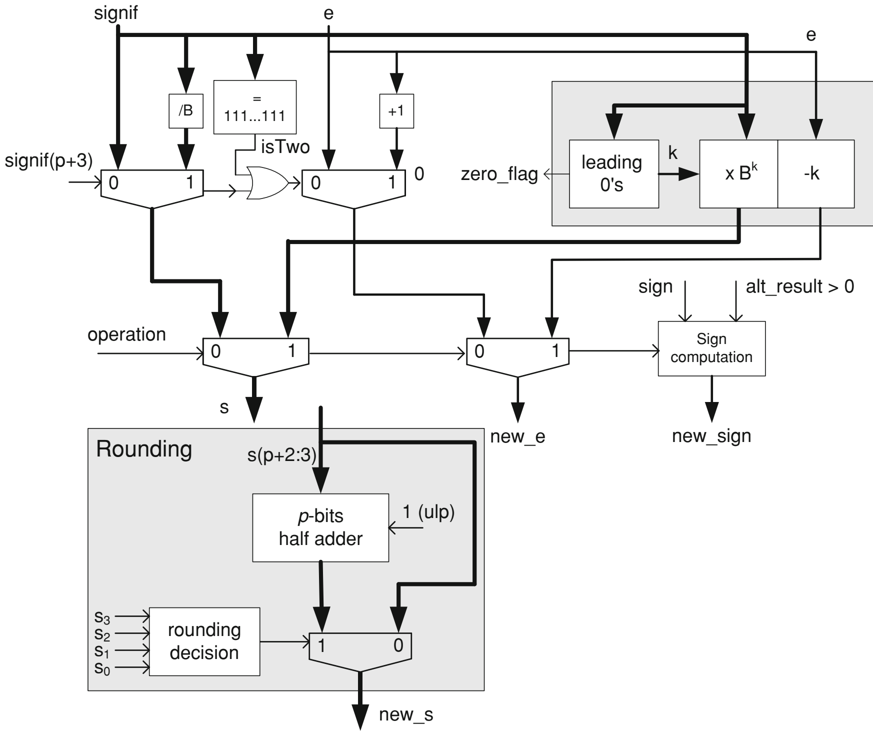


**Fig. 12.4** Normalization and rounding

where $K$ is the size of the floating point numbers (sign, exponent, significand), $E$ is the size of the exponent and $P$ is the size of the significand (including the hidden 1). The entity declaration of the circuit is:

```
entity FP_Add is
   generic(K:natural; P:natural; E:natural);
Port (
  FP_A : in  std_logic_vector (K-1 downto 0);
  FP_B : in  std_logic_vector (K-1 downto 0);
  add_sub: in std_logic;
  clk : in  std_logic;
  ce  : in  std_logic;
  FP_Z : out  std_logic_vector (K-1 downto 0));
end FP_Add;
```

For simplicity the code is written as a single VHDL code except additional files to describe the right shifter of Fig. 12.2 and the leading zero detection and shifting of Fig. 12.4. The code is available at the book home page.

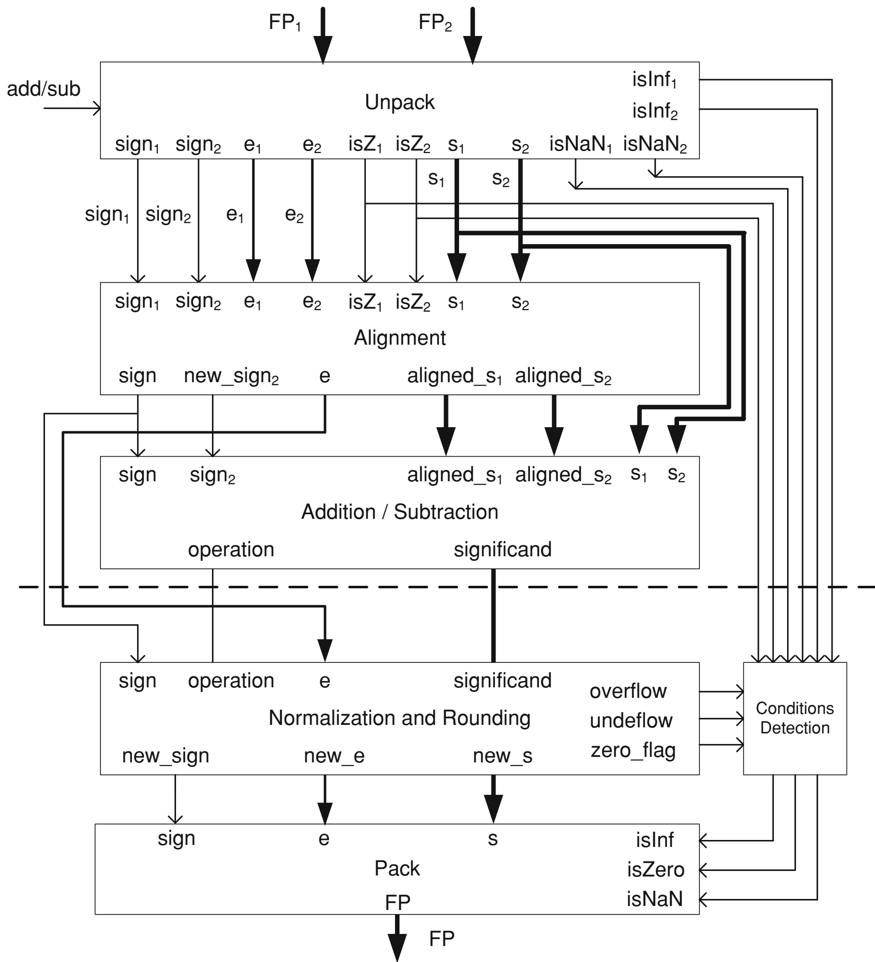


**Fig. 12.5** General structure of a floating point adder-subtractor

A two stage pipeline could be achieved dividing the data path between addtion/ subtraction and normalization and rounding stages (dotted line in Fig. 12.5).

### 12.5.2 Multiplier

A basic multiplier deduced from Algorithm 12.4 is shown in Fig. 12.6. The unpacking and packing circuits are the same as in the case of the adder-subtractor (Fig. 12.5, Sects. 12.5.1.1 and 12.5.1.5) and for simplicity, are not drawn. The "normalization and rounding" is a simplified version of Fig. 12.4, where the part related to the subtraction is not necessary.

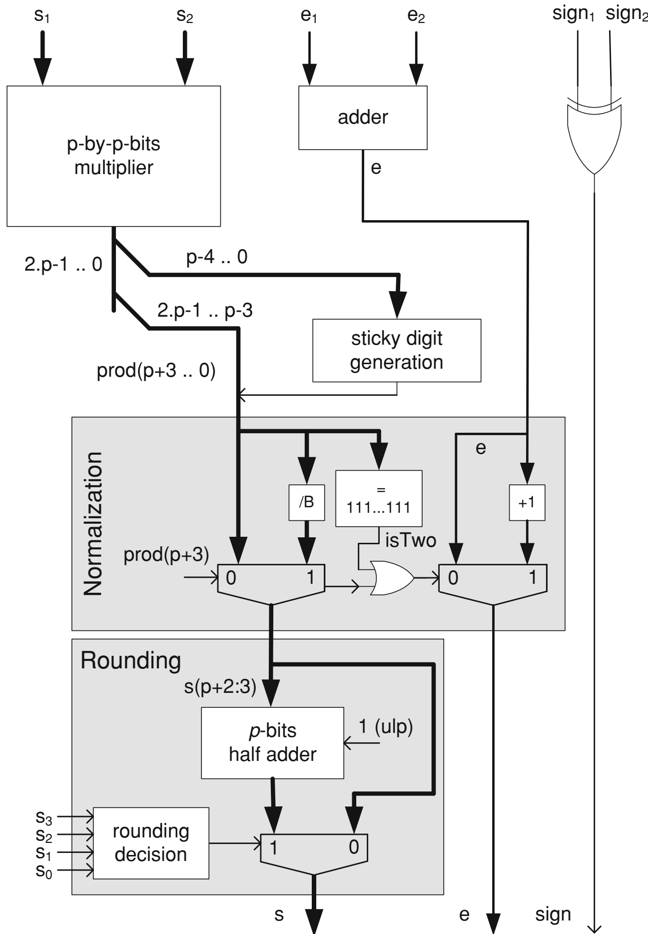


**Fig. 12.6** General structure of a floating point multiplier

The obvious method of computing the sticky bit is with a large fan-in OR gate on the low order bits of the product. Observe, in this case, that the critical path includes the $p$ by $p$ bits multiplication and the sticky digit generation.

An alternative method consists of determining the number of trailing zeros in the two inputs of the multiplier. It is easy to demonstrate that the number of trailing zeros in the product is equal to the sum of the number of trailing zeros in each input operand. Notice that this method does not require the actual low order product bits, just the input operands, so the computation can occur in parallel with the actual multiply operation, removing the sticky computation from the critical path.

The drawback of this method is that significant extra hardware is required. This hardware includes two long length priority encoders to count the number of

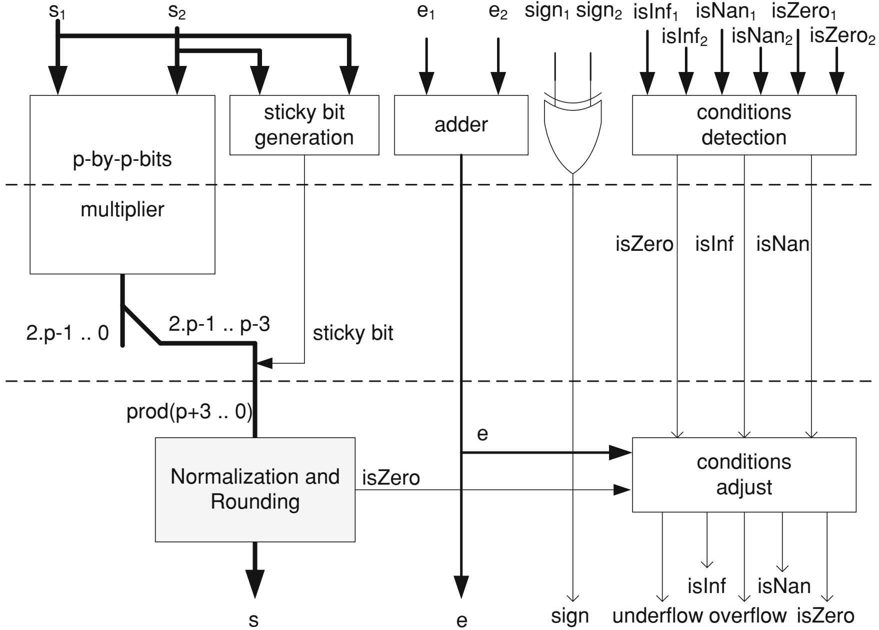


**Fig. 12.7** A better general structure of a floating point multiplier

trailing zeros in the input operands, a small length adder, and a small length comparator. On the other hand, some hardware is eliminated, since the actual low order bits of the product are no longer needed.

A faster floating point multiplier architecture that computes the $p$ by $p$ multiplication and the sticky bit in parallel is presented in Fig. 12.7. The dotted lines suggest a three stage pipeline implementation using a two stage $p$ by $p$ multiplication. The two extra blocks are shown to indicate the special conditions detections. In the second block, the range of the exponent is controlled to detect *overflow* and *underflow* conditions. In this figure the packing and unpacking process are omitted for simplicity.

**Example 12.9** (complete VHDL code available)
Generate the VHDL model of a generic floating-point multiplier. It is made up of the blocks depicted in Fig. 12.7 described in a single VHDL file. For clearness and reusability the code is written using parameters, where $K$ is the size of the floating point numbers (sign, exponent, significand), $E$ is the size of the exponent and $P$ is the size of the significand (including the hidden 1). The entity declaration of the circuit is:

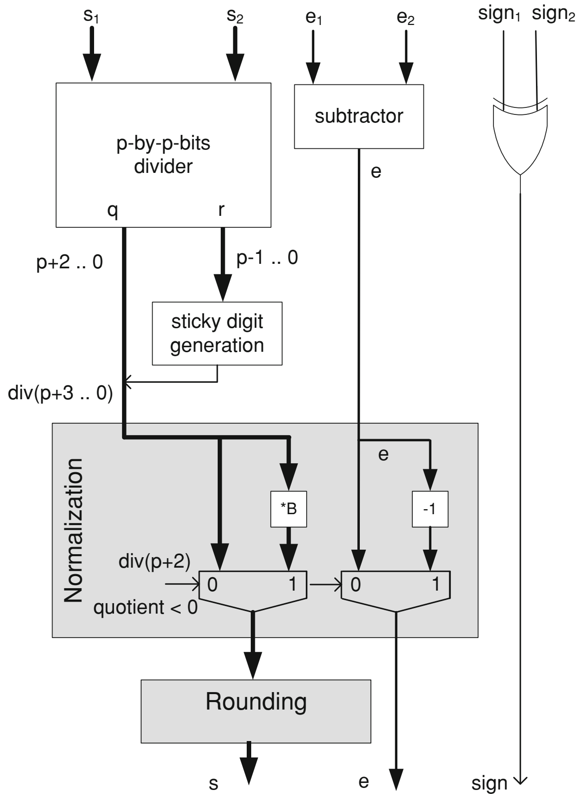


Fig. 12.8 A general structure of a floating point divider

```
entity FP_mul is
   generic(K: natural; P: natural; E: natural);
port (
   FP_A : in  std_logic_vector (K-1 downto 0);
   FP_B: in  std_logic_vector (K-1 downto 0);
   clk : in  std_logic;
   ce   : in  std_logic;
   FP_Z: out std_logic_vector (K-1 downto 0));
end FP_mul ;
```

The code is available at the home page of this book. The combinational circuit registers inputs and outputs to ease the synchronization. A two or three stage pipeline is easily achievable adding the intermediate registers as suggested in Fig. 12.7. In order to increase the clock frequency, more pipeline registers can be inserted into the integer multiplier.

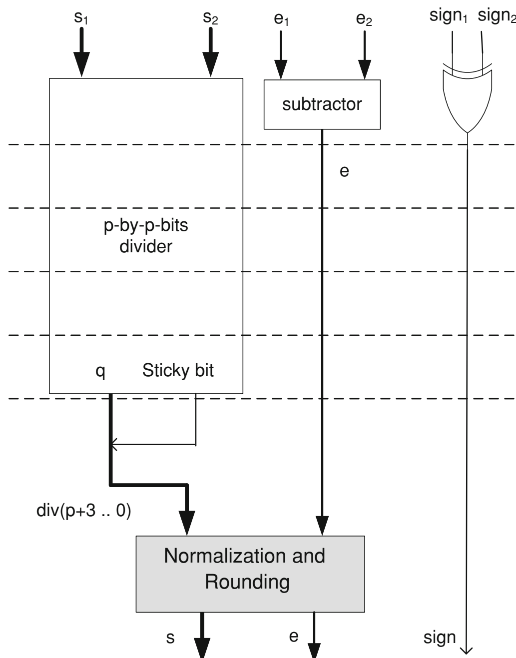


**Fig. 12.9** A pipelined structure of a floating point divider

## 12.5.3 Divider

A basic divider, deduced from Algorithm 12.6, is shown in Fig. 12.9. The unpacking and packing circuits are similar to those of the adder-subtractor or multiplier. The "normalize and rounding" is a simplified version of Fig. 12.4, where the part related to the subtraction is not necessary.

The inputs of the $p$-bit divider are $s_1$ and $s_2$. The first operator $s_1$ is internally divided by two ($s_1/B$, i.e. right shifted) so that the dividend is smaller than the divisor. The precision is chosen equal to $p + 3$ digits. Thus, the outputs quotient and remainder satisfy the relation $(s_1/B).B^{p+3} = s_2.q + r$ where $r < s_2$, that is,

$$s_1/s_2 = q.B^{-(p+2)} + (r/s_2) \cdot B^{-(p+2)} \text{ where } (r/s_2) \cdot B^{-(p+2)} < B^{-(p+2)}.$$

The sticky digit is equal to 1 if $r < 0$ and to 0 if $r = 0$. The final approximation of the exact result is

$$quotient = q.B^{-(p+2)} + sticky_digit.\, B^{-(p+3)}.$$

If a non-restoring divider is used, a further optimization could be done in the sticky bit computation. In the non-restoring divider the final remainder could be negative. In this case, a final correction should be done. This final operation can be avoided: the sticky bit is 0 if the remainder is equal to zero, otherwise it is 1.

A divider is a time consuming circuit. In order to obtain circuits, with frequencies similar to those of floating point multipliers or adders, more pipeline stages are necessary. Figure 12.9 shows a possible pipeline for a floating point divider where the integer divider is also pipelined.

**Example 12.10** (complete VHDL code available)
Generate the VHDL model of a generic floating-point divider. It is made up of the blocks depicted in Fig. 12.9. Most of the design is described in a single VHDL file, but for the integer divider. The integer divider is a non-restoring divider (*div_nr_wsticky.vhd*) that uses a basic non-restoring cell (*a_s.vhd*). For clearness and reusability the code is written using parameters, where $K$ is the size of the floating point numbers (sign, exponent, significand), $E$ is the size of the exponent and $P$ is the size of the significand (including the hidden 1). The entity declaration of the circuit is:

```
entity FP_div is
   generic(K: natural; P: natural; E: natural);
port ( FP_A : in  std_logic_vector (K-1 downto 0);
   FP_B: in  std_logic_vector (K-1 downto 0);
   clk : in  std_logic;
   ce  : in  std_logic;
   FP_Z: out std_logic_vector (K-1 downto 0));
end FP_div;
```

### 12.5.4 Square Root

A basic square rooter deduced from Algorithm 12.7 is shown in Fig. 12.10. The unpacking and packing circuits are the same as in previous operations and, for simplicity, are not drawn. Remember that the first normalization is not necessary, and for most rounding strategies the second normalization is not necessary either.

The exponent is calculated as follows:

$$E = e_1/2 + bias = (E_1 - bias)/2 + bias = E_1/2 + bias/2.$$

- If $e_1$ is even, then both $E_1$ and *bias* are odd (*bias* is always odd). Thus, $E = \lfloor E_1/2 \rfloor + \lfloor bias/2 \rfloor + 1$ where $\lfloor E_1/2 \rfloor$ and $\lfloor bias/2 \rfloor$ amount to right shifts.
- If $e_1$ is odd, then $E_1$ is even. The significand is multiplied by 2 and the exponent reduced by one unit. The biased exponent $E = (E_1 - 1)/2 + bias/2 = E_1/2 + \lfloor bias/2 \rfloor$.

To summarize, the biased exponent $E = \lfloor E_1/2 \rfloor + \lfloor bias/2 \rfloor + parity(E_1)$.

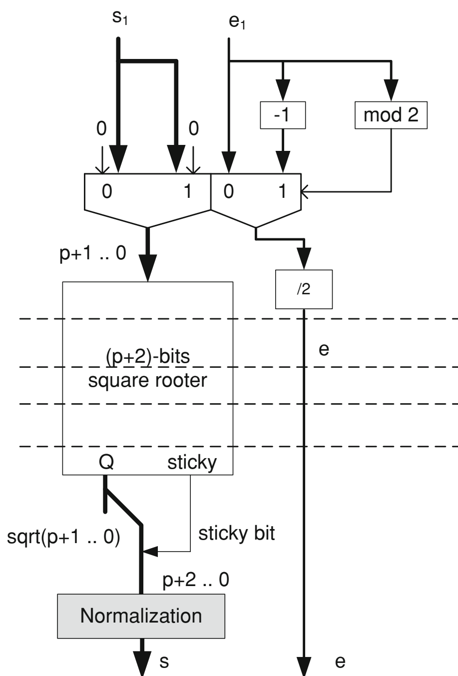


**Fig. 12.10** Structure of a floating point squarer

Since the integer square root is a complex operation, a pipelined floating-point square rooter should be based on a pipelined integer square root. The dotted line of Fig. 12.10 shows a possible five stage pipeline circuit.

**Example 12.11** (complete VHDL code available)
Generate the VHDL model of a generic floating-point square rooter. It is made up of the blocks depicted in Fig. 12.10. The design is described in a single VHDL file, but for the integer square root. The integer square root is based in a non-restoring algorithm (*sqrt_wsticky.vhd*) that uses two basic non-restoring cells (*sqrt_cell.vhd* and *sqrt_cell_00.vhd*). For clearness and reusability, the code is written using parameters, where *K* is the size of the floating point numbers (sign, exponent, significand), *E* is the size of the exponent and *P* is the size of the significand (including the hidden 1). The entity declaration of the circuit is:

```
entity FP_sqrt is
   generic(K: natural; P: natural; E: natural);
port ( FP_A : in  std_logic_vector (K-1 downto 0);
   clk : in  std_logic;
   ce  : in  std_logic;
   FP_Z: out std_logic_vector (K-1 downto 0));
end FP_sqrt;
```

**Table 12.4** Combinational floating point operators in binary32 format

| | FF | LUTs | DSP48 | Slices | Delay |
|---|---|---|---|---|---|
| FP_add | 96 | 699 | – | 275 | 11.7 |
| FP_mult | 96 | 189 | 2 | 105 | 8.4 |
| FP_mult_luts | 98 | 802 | – | 234 | 9.7 |
| FP_div | 119 | 789 | – | 262 | 46.6 |
| FP_sqrt | 64 | 409 | – | 123 | 38.0 |

**Table 12.5** Combinational floating point operators in binary64 format

| | FF | LUTs | DSP48 | Slices | Delay |
|---|---|---|---|---|---|
| FP_add | 192 | 1372 | – | 585 | 15.4 |
| FP_mult | 192 | 495 | 15 | 199 | 15.1 |
| FP_mult_luts | 192 | 3325 | – | 907 | 12.5 |
| FP_div | 244 | 3291 | – | 903 | 136.9 |
| FP_sqrt | 128 | 1651 | – | 447 | 97.6 |

**Table 12.6** Pipelined floating point operators in binary32 format

| | FF | LUTs | DSP48 | Slices | Period | Latency |
|---|---|---|---|---|---|---|
| FP_add | 137 | 637 | – | 247 | 6.4 | 2 |
| FP_mult | 138 | 145 | 2 | 76 | 5.6 | 2 |
| FP_mult_luts | 142 | 798 | – | 235 | 7.1 | 2 |
| FP_mult | 144 | 178 | 2 | 89 | 3.8 | 3 |
| FP_mult_luts | 252 | 831 | – | 272 | 5.0 | 3 |
| FP_sqrt | 384 | 815 | – | 266 | 9.1 | 6 |
| FP_div | 212 | 455 | –0 | 141 | 9.2 | 5 |

### *12.5.5 Implementation Results*

Several implementation results are now presented. The circuits were implemented in a Virtex 5, speed grade 2, device, using ISE 13.1 and XST for synthesis. Tables 12.4 and 12.5 show combinational circuit implementations for binary32 and binary64 floating point operators. The inputs and outputs are registered. When the number of registers (FF) is greater than the number of inputs and outputs, this is due to the register duplication made by synthesizer. The multiplier is implemented using the embedded multiplier (DSP48) and general purpose logic.

Table 12.6 shows the results for pipelined versions in the case of decimal32 data. The circuits include input and output registers. The adder is pipelined in two stages (Fig. 12.5). The multiplier is segmented using three pipeline stages (Fig. 12.7). The divider latency is equal to 6 cycles and the square root latency is equal to five cycles (Figs. 12.9 and 12.10).

## 12.6 Exercises

1. How many bits are there in the exponent and the significand of a 256-bit binary floating point number? What are the ranges of the exponent and the bias?
2. Convert the following decimal numbers to the binary32 and binary64 floating-point format. (a) 123.45; (b) $-1.0$; (c) 673.498e10; (d) qNAN; (e) $-1.345\text{e}{-129}$; (f) $\infty$; (g) 0.1; (h) 5.1e5
3. Convert the following binary32 number to the corresponding decimal number. (a) 08F05352; (b) 7FC00000; (c) AAD2CBC4; (d) FF800000; (e) 484B0173; (f) E9E55838; (g) E9E55838.
4. Add, subtract, multiply and divide the following binary floating point numbers with $B = 2$ and $ulp = 2^{-5}$, so that the numbers are represented in the form $s \cdot 2^e$ where $1 \leq s \leq 1.11111_2$. For simplicity $e$ is written in decimal (base 10).

   (a) $1.10101 \times 2^3$ op $1.10101 \times 2^1$
   (b) $1.00010 \times 2^{-1}$ op $1.00010 \times 2^{-1}$
   (c) $1.00010 \times 2^{-3}$ op $1.10110 \times 2^2$
   (d) $1.10101 \times 2^3$ op $1.00000 \times 2^4$

5. Add, subtract, multiply and divide the following decimal floating point numbers using $B = 10$ and $ulp = 10^{-4}$, so that the numbers are represented in the form $s \cdot 10^e$ where $1 \leq s \leq 9.9999$ (normalized decimal numbers).

   (a) $9.4375 \times 10^3$ op $8.6247 \times 10^2$
   (b) $1.0014 \times 10^3$ op $9.9491 \times 10^2$
   (c) $1.0714 \times 10^4$ op $7.1403 \times 10^2$
   (d) $3.4518 \times 10^{-1}$ op $7.2471 \times 10^3$

6. Analyze the consequences and implication to support denormalized (subnormal in IEEE 754-2008) numbers in the basic operations.
7. Analyze the hardware implication to deal with no-normalized significands ($s$) instead of normalized as in the binary standard.
8. Generate VHDL models adding a pipeline stage to the binary floating point adder of Sect. 12.5.1.
9. Add a pipeline stage to the binary floating point multiplier of Sect. 12.5.2.
10. Generate VHDL models adding two pipeline stages to the binary floating point multiplier of Sect. 12.5.2.
11. Generate VHDL models adding several pipeline stages to the binary floating point divider of Sect. 12.5.3.
12. Generate VHDL models adding several pipeline stages to the binary floating point square root of Sect. 12.5.4.

## Reference

1. IEEE (2008) IEEE standard for floating-point arithmetic, 29 Aug 2008