2. Computer Arithmetic

2.1 Introduction

In computer arithmetic two fundamental design principles are of great importance: number representation and the implementation of algebraic operations [29–33]. We will first discuss possible number representations, (e.g., fixed-point or floating-point), then basic operations like adder and multiplier, and finally efficient implementation of more difficult operations such as square roots, and the computation of trigonometric functions using the CORDIC algorithm or MAC calls.

FPGAs allow a wide variety of computer arithmetic implementations for the desired digital signal processing algorithms, because of the physical bit-level programming architecture. This contrasts with the programmable digital signal processors (PDSPs), with the fixed multiply accumulator core. Careful choice of the bit width in FPGA design can result in substantial savings.

---

**Fig. 2.1.** Survey of number representations
2.2 Number Representation

Deciding whether fixed- or floating-point is more appropriate for the problem must be done carefully, preferably at an early phase in the project. In general, it can be assumed that fixed-point implementations have higher speed and lower cost, while floating-point has higher dynamic range and no need for scaling, which may be attractive for more complicated algorithms. Figure 2.1 is a survey of conventional and less conventional fixed- and floating-point number representations. Both systems are covered by a number of standards but may, if desired, be implemented in a proprietary form.

2.2.1 Fixed-Point Numbers

We will first review the fixed-point number systems shown in Fig. 2.1. Table 2.1 shows the 3-bit coding for the 5 different integer representations.

**Unsigned Integer**

Let \( X \) be an \( N \)-bit unsigned binary number. Then the range is \([0, 2^N - 1]\) and the representation is given by:

\[
X = \sum_{n=0}^{N-1} x_n 2^n, \tag{2.1}
\]

where \( x_n \) is the \( n \)th binary digit of \( X \) (i.e., \( x_n \in [0, 1] \)). The digit \( x_0 \) is called the least significant bit (LSB) and has a relative weight of unity. The digit \( x_{N-1} \) is the most significant bit (MSB) and has a relative weight of \( 2^{N-1} \).

**Signed-Magnitude (SM)**

In signed-magnitude systems the magnitude and the sign are represented separately. The first bit \( x_{N-1} \) (i.e., the MSB) represents the sign and the remaining \( N - 1 \) bits the magnitude. The representation becomes:

\[
X = \begin{cases} 
\sum_{n=0}^{N-2} x_n 2^n & X \geq 0 \\
-\sum_{n=0}^{N-2} x_n 2^n & X < 0.
\end{cases} \tag{2.2}
\]

The range of this representation is \([-2^{N-1} - 1, 2^{N-1} - 1]\). The advantage of the signed-magnitude representation is simplified prevention of overflows, but the disadvantage is that addition must be split depending on which operand is larger.
Table 2.1. Conventional coding of signed binary numbers

<table>
<thead>
<tr>
<th>Binary</th>
<th>2C</th>
<th>1C</th>
<th>D1</th>
<th>SM</th>
<th>Bias</th>
</tr>
</thead>
<tbody>
<tr>
<td>011</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>010</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>-1</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>-2</td>
</tr>
<tr>
<td>000</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-3</td>
</tr>
<tr>
<td>111</td>
<td>-1</td>
<td>-0</td>
<td>-1</td>
<td>-3</td>
<td>4</td>
</tr>
<tr>
<td>110</td>
<td>-2</td>
<td>-1</td>
<td>-2</td>
<td>-2</td>
<td>3</td>
</tr>
<tr>
<td>101</td>
<td>-3</td>
<td>-2</td>
<td>-3</td>
<td>-1</td>
<td>2</td>
</tr>
<tr>
<td>100</td>
<td>-4</td>
<td>-3</td>
<td>-4</td>
<td>-0</td>
<td>1</td>
</tr>
<tr>
<td>1000</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Two’s Complement (2C)

An $N$-bit two’s complement representation of a signed integer, over the range $[-2^{N-1}, 2^{N-1} - 1]$, is given by:

$$X = \begin{cases} \sum_{n=0}^{N-2} x_n 2^n & X \geq 0 \\ -2^{N-1} + \sum_{n=0}^{N-2} x_n 2^n & X < 0. \end{cases}$$

The two’s complement (2C) system is by far the most popular signed numbering system in DSP use today. This is because it is possible to add several signed numbers, and as long as the final sum is in the $N$-bit range, we can ignore any overflow in the arithmetic. For instance, if we add two 3-bit numbers as follows:

$$3_{10} \leftrightarrow 011_{2C}$$
$$-2_{10} \leftrightarrow 110_{2C}$$
$$1_{10} \leftrightarrow 100_{2C}$$

the overflow can be ignored. All computations are modulo $2^N$. It follows that if it is possible to have intermediate values that cannot be correctly represented, but if the final value is valid then the result is correct. For instance, if we add the 3-bit numbers $2 + 2 - 3$, we would have an intermediate value of $010 + 010 = 100_{2C}$, i.e., $-4_{10}$, but the result $100 - 011 = 100 + 101 = 001_{2C}$ is correct.

Two’s complement numbers can also be used to implement modulo $2^N$ arithmetic without any change in the arithmetic. This is what we will use in Chap. 5 to design CIC filters.

One’s Complement (1C)

An $N$-bit one’s complement system (1C) can represent integers over the range $[-(2^{N-1} + 1), 2^{N-1} - 1]$. In a one’s complement code, positive and negative
numbers have bit-by-bit complement representations including for the sign bit. There is, in fact a redundant representation of zero (see Table 2.1). The representation of signed numbers in a 1C system is formally given by:

\[
X = \begin{cases} 
\sum_{n=0}^{N-2} x_n 2^n & X \geq 0 \\
-2^{N-1} + 1 + \sum_{n=0}^{N-2} x_n 2^n & X < 0.
\end{cases}
\] (2.4)

For example, the three-bit 1C representation of the numbers \(-3\) to \(3\) is shown in the third column of Table 2.1.

From the following simple example

\[
\begin{align*}
3_{10} & \longleftrightarrow 0 \ 1_{1C} \\
-2_{10} & \longleftrightarrow 1 \ 0_{1C} \\
1_{10} & \longleftrightarrow 1. \ 0 \ 0_{1C} \\
\text{Carry} & \leftrightarrow \rightarrow 1_{1C} \\
1_{10} & \longleftrightarrow 0 \ 0_{1C}
\end{align*}
\]

we remember that in one’s complement a “carry wrap-around” addition is needed. A carry occurring at the MSB must be added to the LSB to get the correct final result.

The system can, however, efficiently be used to implement modulo \(2^N - 1\) arithmetic without correction. As a result, one’s complement has specialized value in implementing selected DSP algorithms (e.g., Mersenne transforms over the integer ring \(2^N - 1\) [34]).

**Diminished One System (D1)**

A diminished one (D1) system is a biased system. The positive numbers are, compared with the 2C, diminished by 1. The range for \((N+1)\)-bit D1 numbers is \([-2^{N-1}, 2^{N-1}]\), excluding 0. The coding rule for a D1 system is defined as follows:

\[
X = \begin{cases} 
\sum_{n=0}^{N-2} x_n 2^n + 1 & X > 0 \\
-2^{N-1} + \sum_{n=0}^{N-2} x_n 2^n & X < 0 \\
2^N & X = 0.
\end{cases}
\] (2.5)

From adding two D1 numbers

\[
\begin{align*}
3_{10} & \longleftrightarrow 0 \ 1_{D1} \\
-2_{10} & \longleftrightarrow 1 \ 1_{D1} \\
1_{10} & \longleftrightarrow 1. \ 0 \ 0_{D1} \\
\text{Carry} & \leftrightarrow \rightarrow \not{\text{NOT}} \rightarrow 0_{D1} \\
1_{10} & \longleftrightarrow 0 \ 0_{D1}
\end{align*}
\]

we see that, for D1 a complement and add of the *inverted* carry must be computed.
D1 numbers can efficiently be used to implement modulo $2^N+1$ arithmetic without any change in the arithmetic. This fact can be used to implement Fermat NTTs in the ring $2^N + 1$ [34].

**Bias System**

The biased number system has a bias for all numbers. The bias value is usually in the middle of the binary range, i.e., $\text{bias} = 2^{N-1} - 1$. For a 3-bit system, for instance the bias would be $2^{3-1} - 1 = 3$. The range for $N$-bit biased numbers is $[-2^{N-1} - 1, 2^{N-1}]$. Zero is coded as the bias. The coding rule for a biased system is defined as follows:

$$X = \sum_{n=0}^{N-1} x_n 2^n - \text{bias}. \quad (2.6)$$

From adding two biased numbers

| $3_{10}$ | $110_{\text{bias}}$ |
| $+(-2_{10})$ | $001_{\text{bias}}$ |
| $4_{10}$ | $111_{\text{bias}}$ |
| $-\text{bias}$ | $011_{\text{bias}}$ |
| $1_{10}$ | $100_{\text{bias}}$ |

we see that, for each addition the bias needs to be subtracted, while for every subtraction the bias needs to be added.

Bias numbers can efficiently be used to simplify comparison of numbers. This fact will be used in Sect. 2.2.3 (p. 75) for coding the exponent of floating-point numbers.

### 2.2.2 Unconventional Fixed-Point Numbers

In the following we continue the review of number systems according to Fig. 2.1 (p. 57). The unconventional fixed-point number systems discussed in the following are not as often used as for instance the 2C system, but can yield significant improvements for particular applications or problems.

**Signed Digit Numbers (SD)**

The signed digit (SD) system differs from the traditional binary systems presented in the previous section in the fact that it is ternary valued (i.e., digits have the value $\{0, 1, -1\}$, where $-1$ is sometimes denoted as $\overline{1}$).

SD numbers have proven to be useful in carry-free adders or multipliers with less complexity, because the effort in multiplication can typically be estimated through the number of nonzero elements, which can be reduced by using SD numbers. Statistically, half the digits in the two’s complement
coding of a number are zero. For an SD code, the density of zeros increases to two thirds as the following example shows:

**Example 2.1: SD Coding**

Consider coding the decimal number 15 = 1111₂ using a 5-bit binary and an SD code. Their representations are as follows:

1) \[ 15_{10} = 16_{10} - 1_{10} = 1000\overline{1}_{SD} \]
2) \[ 15_{10} = 16_{10} - 2_{10} + 1_{10} = 100\overline{1}1_{SD} \]
3) \[ 15_{10} = 16_{10} - 4_{10} + 3_{10} = 10\overline{1}11_{SD} \]
4) etc.

The SD representation, unlike a 2C code, is nonunique. We call a *canonic signed digit* system (CSD) the system with the minimum number of non-zero elements. The following algorithm can be used to produce a classical CSD code.

**Algorithm 2.2: Classical CSD Coding**

Starting with the LSB substitute all 1 sequences equal or larger than two, with \(10\ldots\overline{0}1\).

This CSD coding is the basis for the C utility program `csd.exe` on the CD-ROM. This classical CSD code is also unique and an additional property is that the resulting representation has at least one zero between two digits, which may have values 1, \(\overline{1}\), or 0.

**Example 2.3: Classical CSD Code**

Consider again coding the decimal number 15 using a 5-bit binary and a CSD code. Their representations are: \(1111₂ = 1000\overline{1}_{CSD}\). We notice from a comparison with the SD coding from Example 2.1 that only the first representation is a CSD code.

As another example consider the coding of

\[ 27_{10} = 11011₂ = 1110\overline{0}₁₇_{CSD} = 100\overline{1}0₁₇_{CSD}. \]  

We note that, although the first substitution of \(011 \rightarrow 10\overline{1}\) does not reduce the complexity, it produces a length-three strike, and the complexity reduces from three additions to two subtractions.

On the other hand, the classical CSD coding does not always produce the optimal CSD coding in terms of hardware complexity, because in Algorithm 2.2 additions are also substituted by subtractions, when there should be no such substitution. For instance \(011₂\) is coded as \(10\overline{1}₇_{CSD}\), and if this coding is used to produce a constant multiplier the subtraction will need a full-adder instead of a half-adder for the LSB. The CSD coding given in the following

---

1 You need to copy the program to your harddrive first because the program writes out the results in a file `csd.dat`; you cannot start it from the CD directly.
will produce a CSD coding with the minimum number of nonzero terms, but also with the minimum number of subtractions.

<table>
<thead>
<tr>
<th>Algorithm 2.4: Optimal CSD Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Starting with the LSB substitute all 1 sequences larger than two with 10...01. Also substitute 1011 with 1101.</td>
</tr>
<tr>
<td>2) Starting with the MSB, substitute 101 with 011.</td>
</tr>
</tbody>
</table>

### Fractional (CSD) Coding

Many DSP algorithms require the implementation of fractional numbers. Think for instance of trigonometric coefficient like sine or cosine coefficients. Implementation via integer numbers only would result in a large quantization error. The question then is, can we also use the CSD coding to reduce the implementation effort of a fractional constant coefficient? The answer is yes, but we need to be a little careful with the ordering of the operands. In VHDL the analysis of an expression is usually done from left to right, which means an expression like \( y = 7 \times x / 8 \) is implemented as \( y = (7 \times x) / 8 \), and equivalently the expression \( y = x / 8 \times 7 \) is implemented as \( y = (x / 8) \times 7 \). The latter term unfortunately will produce a large quantization error, since the evaluation of \( x / 8 \) is in fact synthesized by the tool as a right shift by three bits, so we will lose the lower three bits of our input \( x \) in the computation that follows. Let us demonstrate this with a small HDL design example.

#### Example 2.5: Fractional CSD Coding

Consider coding the fractional decimal number 0.875 = 7/8 using a fractional 4-bit binary and CSD code. The 7 can be implemented more efficiently in CSD as \( 7 = 8 - 1 \) and we want to determine the quantization error of the following four mathematically equivalent representations, which give different synthesis results:

\[
\begin{align*}
y_0 &= 7 \times x / 8 = (7 \times x) / 8 \\
y_1 &= x / 8 \times 7 = (x / 8) \times 7 \\
y_2 &= x / 2 + x / 4 + x / 8 = ((x / 2) + (x / 4)) + (x / 8) \\
y_3 &= x - x / 8 = x - (x / 8)
\end{align*}
\]

Using parenthesis in the above equations it is shown how the HDL tool will group the expressions. Multiply and divide have a higher priority than add and subtract and the evaluation is from left to right. The VHDL code\(^3\) of the constant coefficient fractional multiplier is shown next.

```vhdl
-- --------------------------------------------------------
ENTITY cmul7p8 IS ------> Interface
PORT(x : IN INTEGER RANGE -16 TO 15; -- System input
y0, y1, y2, y3 : OUT INTEGER RANGE -16 TO 15);
```

\(^2\) Most HDL tools only support dividing by power-of-two values, which can be designed using a shifter, see Sect. 2.5, p. 93.

\(^3\) The equivalent Verilog code `cmul7p8.v` for this example can be found in Appendix A on page 797. Synthesis results are shown in Appendix B on page 881.
The design uses 48 LEs and no embedded multiplier. A registered performance cannot be measured since there is no register-to-register path. The simulated results of the fractional constant coefficient multiplier is shown in Fig. 2.2. Note the large quantization error for $y_1$. Looking at the results for the input value $x = 4$, we can also see that the CSD coding $y_3$ shows rounding to the next largest integer, while $y_0$ and $y_2$ show rounding to the next smallest integer. For negative value (e.g., $-4$) we see that the CSD coding $y_3$ shows rounding to the next smallest (i.e., $-4$) integer, while $y_0$ and $y_2$ show rounding to the next largest (i.e., $-3$) integer.

### Carry-Free Adder

The SD number representation can also be used to implement a carry-free adder. Tagaki et al. [35] introduced the scheme presented in Table 2.2. Here, $u_k$ is the interim sum and $c_k$ is the carry of the $k^{th}$ bit (i.e., to be added to $u_{k+1}$).

#### Example 2.6: Carry-Free Addition

The addition of 29 to $-9$ in the SD system is performed below.
Table 2.2. Adding carry-free binaries using the SD representation

<table>
<thead>
<tr>
<th>$x_k y_k$</th>
<th>00</th>
<th>01</th>
<th>01</th>
<th>01</th>
<th>01</th>
<th>11</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_{k-1} y_{k-1}$</td>
<td>–</td>
<td>neither</td>
<td>at least</td>
<td>neither</td>
<td>at least</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>$c_k$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>⊤</td>
<td>1</td>
<td>⊤</td>
</tr>
<tr>
<td>$u_k$</td>
<td>0</td>
<td>⊤</td>
<td>1</td>
<td>⊤</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
100\top01 x_k \\
+ \ 0\top1111 y_k \\
\hline
000\top11 c_k \\
1\top10\top0 u_k \\
\hline
1\top0100 s_k
\end{align*}
\]

However, due to the ternary logic burden, implementing Table 2.2 with FPGAs requires four-input operands for the $c_k$ and $u_k$. This translates into a $2^8 \times 4$-bit LUT when implementing Table 2.2.

Multiplier Adder Graph (MAG)

We have seen that the cost of multiplication is a direct function of the number of nonzero elements $a_k$ in $A$. The CSD system minimizes this cost. The CSD is also the basis for the Booth multiplier [29] discussed in Exercise 2.2 (p. 169).

It can, however, sometimes be more efficient first to factor the coefficient into several factors, and realize the individual factors in an optimal CSD sense [36–39]. Figure 2.3 illustrates this option for the coefficient 93. The direct binary and CSD codes are given by $93_{10} = 1011101_2 = 1100\top01_{\text{CSD}},$

![Fig. 2.3. Two realizations for the constant factor 93](image-url)
with the 2C requiring four adders, and the CSD requiring three adders. The coefficient 93 can also be represented as $93 = 3 \times 31$, which requires one adder for each factor (see Fig. 2.3). The complexity for the factor number is reduced to two. There are several ways to combine these different factors. The number of adders required is often referred to as the cost of the constant coefficient multiplier. Figure 2.4, suggested by Dempster et al. [38], shows all possible configurations for one to four adders. Using this graph, all coefficients with a cost ranging from one to four can be synthesized with $k_i \in \mathbb{N}_0$, according to:

Cost 1: 
1) $A = 2^{k_0}(2^{k_1} \pm 2^{k_2})$

Cost 2: 
1) $A = 2^{k_0}(2^{k_1} \pm 2^{k_2} \pm 2^{k_3})$
2) $A = 2^{k_0}(2^{k_1} \pm 2^{k_2})(2^{k_3} \pm 2^{k_4})$

Cost 3: 
1) $A = 2^{k_0}(2^{k_1} \pm 2^{k_2} \pm 2^{k_3} \pm 2^{k_4})$

Using this technique, Table 2.3 shows the optimal coding for all 8-bit integers having a cost between zero and three [5].

**Logarithmic Number System (LNS)**

The logarithmic number system (LNS) [40, 41] is analogous to the floating-point system with a fixed mantissa and a fractional exponent. In the LNS, a number $x$ is represented as:

$$X = \pm r^{\pm e_x},$$

where $r$ is the system’s radix, and $e_x$ is the LNS exponent. The LNS format consists of a sign-bit for the number and exponent, and an exponent assigned $I$ integer bits and $F$ fractional bits of precision. The format in graphical form is shown below:

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent sign</th>
<th>Exponent integer bits $I$</th>
<th>Exponent fractional bits $F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_X$</td>
<td>$S_e$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The LNS, like floating-point, carries a nonuniform precision. Small values of $x$ are highly resolved, while large values of $x$ are more coarsely resolved as the following example shows.

**Example 2.7: LNS Coding**

Consider a radix-2 9-bit LNS word with two sign-bits, three bits for integer precision and four-bit fractional precision. How can, for instance, the LNS coding 00011.0010 be translated into the real number system? The two sign bits indicate that the whole number and the exponent are positive. The integer part is 3 and the fractional part $2^{-3} = 1/8$. The real number representation is therefore $2^{3+1/8} = 2^{3.125} = 8.724$. We find also that $-2^{3.125} = 10011.0010$ and $2^{-3.125} = 01100.1110$. Note that the exponent is represented in fractional two’s complement format. The largest number that can be represented with this 9-bit LNS format is $2^{8-1/16} \approx 2^8 = 256$ and
the smallest is $2^{-8} = 0.0039$, as graphically interpreted in Fig. 2.5a. In contrast, an 8-bit plus sign fixed-point number has a maximal positive value of $2^8 - 1 = 255$, and the smallest nonzero positive value is one. A comparison of the two 9-bit systems is shown in Fig. 2.5b.

The historical attraction of the LNS lies in its ability to efficiently implement multiplication, division, square-rooting, or squaring. For example, the product $C = A \times B$, where $A$, $B$, and $C$ are LNS words, is given by:

$$C = r^{e_a} \times r^{e_b} = r^{e_a + e_b} = r^{e_c}. \quad (2.9)$$

That is, the exponent of the LNS product is simply the sum of the two exponents. Division and high-order operations immediately follow. Unfortunately,
Table 2.3. Cost $C$ (i.e., number of adders) for all 8-bit numbers using the multiplier adder graph (MAG) technique

<table>
<thead>
<tr>
<th>$C$</th>
<th>Coefficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1, 2, 4, 8, 16, 32, 64, 128, 256</td>
</tr>
<tr>
<td>1</td>
<td>3, 5, 6, 7, 9, 10, 12, 14, 15, 17, 18, 20, 24, 28, 30, 31, 33, 34, 36, 40, 48, 56, 60, 62, 63, 65, 66, 68, 72, 80, 96, 112, 120, 124, 126, 127, 129, 130, 132, 136, 144, 160, 192, 224, 240, 248, 252, 254, 255</td>
</tr>
<tr>
<td>4</td>
<td>171, 173, 179, 181, 203, 205, 211, 213</td>
</tr>
</tbody>
</table>

Minimum costs through factorization

<table>
<thead>
<tr>
<th>$C$</th>
<th>Coefficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>45 = 5 × 9, 51 = 3 × 17, 75 = 5 × 15, 85 = 5 × 17, 90 = 2 × 9 × 5, 93 = 3 × 31, 99 = 3 × 33, 102 = 2 × 3 × 17, 105 = 7 × 15, 150 = 2 × 5 × 15, 153 = 5 × 31, 165 = 5 × 33, 170 = 2 × 5 × 17, 180 = 4 × 5 × 9, 186 = 2 × 3 × 31, 189 = 7 × 9, 195 = 3 × 65, 198 = 2 × 3 × 33, 204 = 4 × 3 × 17, 210 = 2 × 7 × 15, 217 = 7 × 31, 231 = 7 × 33</td>
</tr>
<tr>
<td>3</td>
<td>171 = 3 × 57, 173 = 8 + 165, 179 = 51 + 128, 181 = 1 + 180, 211 = 1 + 210, 213 = 3 × 71, 205 = 5 × 41, 203 = 7 × 29</td>
</tr>
</tbody>
</table>

Addition or subtraction are by comparison far more complex. Addition and subtraction operations are based on the following procedure, where it is assumed that $A > B$.

\[
C = A + B = 2^{e_a} + 2^{e_b} = 2^{e_a} \left(1 + 2^{e_b - e_a}\right) = 2^{e_c}. \tag{2.10}
\]

Solving for the exponent $e_c$, one obtains $e_c = e_a + \phi^+(\Delta)$ where $\Delta = e_b - e_a$ and $\phi^+(u) = \log_2(\Phi^+(\Delta))$. For subtraction a similar table, $\phi^-(u) = \log_2(\Phi^-(\Delta))$, $\Phi^-(\Delta) = (1 - 2^{e_b - e_a})$, can be used. Such tables have been historically used for rational numbers as described in “Logarithmorum Completus,” Jurij Vega (1754–1802), containing tables computed by Zech. As a result, the term $\log_2(1 - 2^u)$ is usually referred to as a Zech logarithm.
2.2 Number Representation

LNS arithmetic is performed in the following manner [40]. Let \( A = 2^{e_a}, B = 2^{e_b}, C = 2^{e_c} \), with \( S_A, S_B, S_C \) denoting the sign-bit for each word:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply</td>
<td>( C = A \times B ) ( e_c = e_a + e_b; S_C = S_A \text{ XOR } S_B )</td>
</tr>
<tr>
<td>Divide</td>
<td>( C = A/B ) ( e_c = e_a - e_b; S_C = S_A \text{ XOR } S_B )</td>
</tr>
<tr>
<td>Add</td>
<td>( C = A + B ) ( e_c = \begin{cases} e_a + \phi^+(e_b - e_a) &amp; A \geq B \ e_b + \phi^+(e_a - e_b) &amp; B &gt; A \end{cases} )</td>
</tr>
<tr>
<td>Subtract</td>
<td>( C = A - B ) ( e_c = \begin{cases} e_a + \phi^-(e_b - e_a) &amp; A \geq B \ e_b + \phi^-(e_a - e_b) &amp; B &gt; A \end{cases} )</td>
</tr>
<tr>
<td>Square root</td>
<td>( C = \sqrt{A} ) ( e_c = e_a/2 )</td>
</tr>
<tr>
<td>Square</td>
<td>( C = A^2 ) ( e_c = 2e_a )</td>
</tr>
</tbody>
</table>

Methods have been developed to reduce the necessary table size for the Zech logarithm by using partial tables [40] or using linear interpolation techniques [42]. These techniques are beyond the scope of the discussion presented here.

**Residue Number System (RNS)**

The RNS is actually an ancient algebraic system whose history can be traced back 2000 years. The RNS is an integer arithmetic system in which the primitive operations of addition, subtraction, and multiplication are defined. The primitive operations are performed concurrently within noncommunicating small-wordlength channels [43,44]. An RNS system is defined with respect to a positive integer basis set \( \{m_1, m_2, \ldots, m_L\} \), where the \( m_l \) are all relatively (pairwise) prime. The dynamic range of the resulting system is \( M \), where \( M = \prod_{l=1}^L m_l \). For signed-number applications, the integer value of \( X \) is assumed to be constrained to \( X \in [-M/2, M/2] \). RNS arithmetic is defined within a ring isomorphism:
\[
\mathbb{Z}_M \cong \mathbb{Z}_{m_1} \times \mathbb{Z}_{m_2} \times \cdots \times \mathbb{Z}_{m_L},
\]
where \(\mathbb{Z}_M = \mathbb{Z}/(M)\) corresponds to the ring of integers modulo \(M\), called the residue class mod \(M\). The mapping of an integer \(X\) into an RNS \(L\)-tuple \(X \leftrightarrow (x_1, x_2, \ldots, x_L)\) is defined by \(x_l = X \mod m_l\), for \(l = 1, 2, \ldots, L\). Defining \(\Box\) to be the algebraic operations +, − or *, it follows that, if \(Z, X, Y \in \mathbb{Z}_M\), then:

\[
Z = X \Box Y \mod M
\]  
(2.12)
is isomorphic to \(Z \leftrightarrow (z_1, z_2, \ldots, z_L)\). Specifically:

\[
\begin{align*}
X &\xleftarrow{(m_1,m_2,\ldots,m_L)} \langle X \rangle_{m_1} & \langle X \rangle_{m_2} & \cdots & \langle X \rangle_{m_L} \\
Y &\xleftarrow{(m_1,m_2,\ldots,m_L)} \langle Y \rangle_{m_1} & \langle Y \rangle_{m_2} & \cdots & \langle Y \rangle_{m_L} \\
Z &\xleftarrow{(m_1,m_2,\ldots,m_L)} \langle X \Box Y \rangle_{m_1} & \langle X \Box Y \rangle_{m_2} & \cdots & \langle X \Box Y \rangle_{m_L}.
\end{align*}
\]

As a result, RNS arithmetic is pairwise defined. The \(L\) elements of \(Z = (X \Box Y) \mod M\) are computed concurrently within \(L\) small-wordlength mod \((m_l)\) channels whose width is bounded by \(w_l = \lceil \log_2(m_l) \rceil\) bits (typical 4 to 8 bits). In practice, most RNS arithmetic systems use small RAM or ROM tables to implement the modular mappings \(z_l = x_l \Box y_l \mod m_l\).

**Example 2.8: RNS Arithmetic**

Consider an RNS system based on the relatively prime moduli set \(\{2, 3, 5\}\) having a dynamic range of \(M = 2 \times 3 \times 5 = 30\). Two integers in \(\mathbb{Z}_{30}\), say 7\textsubscript{10} and 4\textsubscript{10}, have RNS representations 7 = (1, 1, 2)\textsubscript{RNS} and 4 = (0, 1, 4)\textsubscript{RNS}, respectively. Their sum, difference, and products are 11, 3, and 28, respectively, which are all within \(\mathbb{Z}_{30}\). Their computation is shown below.

\[
\begin{align*}
7 \xrightarrow{(2,3,5)} (1,1,2) & & 7 \xrightarrow{(2,3,5)} (1,1,2) & & 7 \xrightarrow{(2,3,5)} (1,1,2) \\
+4 \xrightarrow{(2,3,5)} (0,1,4) & & -4 \xrightarrow{(2,3,5)} (0,1,4) & & \times4 \xrightarrow{(2,3,5)} (0,1,4) \\
11 \xrightarrow{(2,3,5)} (1,2,1) & & 3 \xrightarrow{(2,3,5)} (1,0,3) & & 28 \xrightarrow{(2,3,5)} (0,1,3).
\end{align*}
\]

RNS systems have been built as custom VLSI devices [45], GaAs, and LSI [44]. It has been shown that, for small wordlengths, the RNS can provide a significant speed-up using the \(2^4 \times 2\)-bit tables found in Xilinx FPGAs [46]. For larger moduli, the M2K and M9K tables belonging to the Altera FPGAs are beneficial in designing RNS arithmetic and RNS-to-integer converters. With the ability to support larger moduli, the design of high-precision high-speed FPGA systems becomes a practical reality.

A historical barrier to implementing practical RNS systems, until recently, has been decoding [47]. Implementing RNS-to-integer decoder, division, or
magnitude scaling, requires that data first be converted from an RNS format to an integer. The commonly referenced RNS-to-integer conversion methods are called the Chinese remainder theorem (CRT) and the mixed-radix-conversion (MRC) algorithm [43]. The MRC actually produced the digits of a weighted number system representation of an integer while the CRT maps an RNS $L$-tuple directly to an integer. The CRT is defined below.

$$X \mod M \equiv \sum_{l=0}^{L-1} \hat{m}_l \langle \hat{m}_l^{-1} x_l \rangle_{m_l} \mod M, \quad (2.13)$$

where $\hat{m}_l = M/m_l$ is an integer, and $\hat{m}_l^{-1}$ is the multiplicative inverse of $\hat{m}_l \mod m_l$, i.e., $\hat{m}_l \hat{m}_l^{-1} \equiv 1 \mod m_l$. Typically, the desired output of an RNS computation is much less than the maximum dynamic range $M$. In such cases, a highly efficient algorithm, called the $\varepsilon$-CRT [48], can be used to implement a time- and area-efficient RNS to (scaled) integer conversion.

**Index Multiplier**

There are, in fact, several variations of the RNS. One in common use is based on the use of index arithmetic [43]. It is similar in some respects to logarithmic arithmetic. Computation in the index domain is based on the fact that, if all the moduli are primes, it is known from number theory that there exists a primitive element, a *generator* $g$, such that:

$$a \equiv g^\alpha \mod p \quad (2.14)$$

that generates all elements in the field $\mathbb{Z}_p$, excluding zero (denoted $\mathbb{Z}_p/\{0\}$). There is, in fact, a one-to-one correspondence between the integers $a$ in $\mathbb{Z}_p/\{0\}$ and the exponents $\alpha$ in $\mathbb{Z}_{p-1}$. As a point of terminology, the index $\alpha$, with respect to the generator $g$ and integer $a$, is denoted $\alpha = \text{ind}_g(a)$.

**Example 2.9: Index Coding**

Consider a prime moduli $p = 17$; a generator $g = 3$ will generate the elements of $\mathbb{Z}_p/\{0\}$. The encoding table is shown below. For notational purposes, the case $a = 0$ is denoted by $g^{-\infty} = 0$.

<table>
<thead>
<tr>
<th>$a$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>ind$_3(a)$</td>
<td>$-\infty$</td>
<td>0</td>
<td>14</td>
<td>1</td>
<td>12</td>
<td>5</td>
<td>15</td>
<td>11</td>
<td>10</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>13</td>
<td>4</td>
<td>9</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

Multiplication of RNS numbers can be performed as follows:

1) Map $a$ and $b$ into the index domain, i.e., $a = g^\alpha$ and $b = g^\beta$
2) Add the index values modulo \( p - 1 \), i.e., \( \nu = (\alpha + \beta) \mod (p - 1) \)
3) Map the sum back to the original domain, i.e., \( n = g^\nu \)

If the data being processed is in index form, then only exponent addition mod\((p - 1)\) is required. This is illustrated by the following example.

**Example 2.10: Index Multiplication**

Consider the prime moduli \( p = 17 \), generator \( g = 3 \), and the results shown in Example 2.9. The multiplication of \( a = 2 \) and \( b = 4 \) proceeds as follows:

\[
(\text{ind}_g(2) + \text{ind}_g(4)) \mod 16 = (14 + 12) \mod 16 = 10.
\]

From the table in Example 2.9 it is seen that ind\(_3\)(8) = 10, which corresponds to the integer 8, which is the expected result.

**Addition in the Index Domain**

Most often, DSP algorithms require both multiplication and addition. Index arithmetic is well suited to multiplication, but addition is no longer trivial. Technically, addition can be performed by converting index RNS data back into the RNS where addition is simple to implement. Once the sum is computed the result is mapped back into the index domain. Another approach is based on a Zech logarithm. The sum of index-coded numbers \( a \) and \( b \) is expressed as:

\[
d = a + b = g^\delta = g^\alpha + g^\beta = g^\alpha \left(1 + g^{\beta - \alpha}\right) = g^\beta \left(1 + g^{\alpha - \beta}\right).
\]

Adding numbers in the index domain, therefore, requires one addition, one subtraction, and a Zech LUT. The following small example illustrates the principle of adding 2 + 5 in the index domain.

**Example 2.12: Zech Logarithms**

A table of Zech logarithms, for a prime moduli 17 and \( g = 3 \), is shown below.

<table>
<thead>
<tr>
<th>( n )</th>
<th>(-\infty)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Z(n) )</td>
<td>0</td>
<td>14</td>
<td>12</td>
<td>3</td>
<td>7</td>
<td>9</td>
<td>15</td>
<td>8</td>
<td>13</td>
<td>(-\infty)</td>
<td>6</td>
<td>2</td>
<td>10</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>11</td>
</tr>
</tbody>
</table>
2.2 Number Representation

The index values for 2 and 5 are defined in the tables found in Example 2.9 (p. 71). It therefore follows that:

\[ 2 + 5 = 3^{14} + 3^5 = 3^5(1 + 3^9) = 3^{5+Z(9)} = 3^{11} \equiv 7 \mod 17. \]

The case where \( a + b \equiv 0 \) needs special attention, corresponding to the case where [49]:

\[ -X \equiv Y \mod p \iff g^{\alpha+(p-1)/2} \equiv g^\beta \mod p. \]

That is, the sum is zero if, in the index domain, \( \beta = \alpha+(p-1)/2 \mod (p-1). \)

An example follows.

**Example 2.13:** The addition of 5 and 12 in the original domain is given by

\[ 5 + 12 = 3^5 + 3^{13} = 3^5(1 + 3^8) = 3^{5+Z(8)} \equiv 3^{-\infty} \equiv 0 \mod 17. \]

**Complex Multiplication using QRNS**

Another interesting property of the RNS arises if we process complex data. This special representation, called QRNS, allows very efficient multiplication, which we wish to discuss next.

When the real and imaginary components are coded as RNS digits, the resulting system is called the complex RNS or CRNS. Complex addition in the CRNS requires that two real adds be performed. Complex RNS (CRNS) multiplication is defined in terms of four real products, an addition, and a subtraction. This condition is radically changed when using a variant of the RNS, called the quadratic RNS, or QRNS. The QRNS is based on known properties of Gaussian primes of the form \( p = 4k + 1 \), where \( k \) is a positive integer. The importance of this choice of moduli is found in the factorization of the polynomial \( x^2+1 \) in \( \mathbb{Z}_p \). The polynomial has two roots, \( j \) and \( -j \), where \( j \) and \( -j \) are real integers belonging to the residue class \( \mathbb{Z}_p \). This is in sharp contrast with the factoring of \( x^2+1 \) over the complex field. Here, the roots are complex and have the form \( x_{1,2} = \alpha \pm j\beta \) where \( j = \sqrt{-1} \) is the imaginary operator. Converting a CRNS number into the QRNS is accomplished by the transform \( f : \mathbb{Z}_p^2 \rightarrow \mathbb{Z}_p^2 \), defined as follows:

\[ f(a+jb) = ((a+jb) \mod p, (a-jb) \mod p) = (A, B). \]

(2.18)

In the QRNS, addition and multiplication is realized componentwise, and is defined as

\[ (a+ja) + (c+jd) \leftrightarrow (A + C, B + D) \mod p \]

(2.19)

\[ (a+jb)(c+jd) \leftrightarrow (AC, BD) \mod p \]

(2.20)
and the square of the absolute value can be computed with
\[ |a + jb|^2 \leftrightarrow (A \times B) \mod p. \]  
(2.21)

The inverse mapping from QRNS digits back to the CRNS is defined by:
\[ f^{-1}(A, B) = 2^{-1}(A + B) + j(2j)^{-1}(A - B) \mod p. \]  
(2.22)

Consider the Gaussian prime \( p = 13 \) and the complex product of \( (a + jb) = (2 + j1), (c + jd) = (3 + j2) \), is \((2 + j1) \times (3 + j2) = (4 + j7) \mod 13\). In this case four real multiplies, a real add, and real subtraction are required to complete the product.

**Example 2.14: QRNS Multiplication**

The quadratic equation \( x^2 \equiv (-1) \mod 13 \) has two roots: \( \hat{j} = 5 \) and \( -\hat{j} = -5 \equiv 8 \mod 13 \). The QRNS-coded data become:

\( (a + jb)= 2 + j \leftrightarrow (2 + 5 \times 1,2 + 8 \times 1) \equiv (A, B) = (7, 10) \mod 13 \)
\( (c + jd)= 3 + j2 \leftrightarrow (3 + 5 \times 2,3 + 8 \times 2) \equiv (C, D) = (0, 6) \mod 13 \).

Componentwise multiplication yields \((A, B)(C, D) = (7, 10)(0, 6) \equiv (0, 8) \mod 13\), requiring only two real multiplies. The inverse mapping to the CRNS is defined in terms of (2.22), where \( 2^{-1} \equiv 7 \) and \( (2j)^{-1} = 10^{-1} \equiv 4 \). Solving the equations for \( 2x \equiv 1 \mod 13 \) and \( 10x \equiv 1 \mod 13 \), produces 7 and 4, respectively. It then follows that
\[ f^{-1}(0, 8) = 7(0 + 8) + j \cdot 4(0 - 8) \mod 13 \equiv 4 + j7 \mod 13. \]  
(2.14)

Figure 2.6 shows a graphical interpretation of the mapping between CRNS and QRNS.
2.2 Number Representation

2.2.3 Floating-Point Numbers

Floating-point systems were developed to provide high resolution over a large dynamic range. Floating-point systems can often provide a solution when fixed-point systems, with their limited dynamic range, fail. Floating-point systems, however, bring a speed and complexity penalty. Most microprocessor floating-point systems comply with the published single- or double-precision IEEE floating-point standard [50], while FPGA-based systems often employ custom formats [51]. We will therefore discuss in the following the standard and custom floating-point formats, and in Sect. 2.7 (p. 109) the design of basic building blocks. Such arithmetic blocks are available from several “intellectual property” providers, and have recently been included in the VHDL-2008 standard.

A standard normalized floating-point word consists of a sign-bit \( s \), exponent \( e \), and an unsigned (fractional) normalized mantissa \( m \), arranged as follows:

| Sign \( s \) | Exponent \( e \) | Unsigned mantissa \( m \) |

Algebraically, a (normalized) floating-point word is represented by

\[
X = (-1)^s \times 1.m \times 2^{e-bias}.
\] (2.23)

Note that this is a signed magnitude format (see p. 59). The “hidden” one in the mantissa is not present in the binary coding of the normalized floating-point number. If the exponent is represented with \( E \) bits then the bias is selected to be

\[
bias = 2^{E-1} - 1.
\] (2.24)

To illustrate, let us determine the decimal value \( 9.25 \) in a 12-bit custom floating-point format.

**Example 2.15: A \((1,6,5)\) Floating-Point Format**

Consider a floating-point representation with a sign bit, \( E = 6 \)-bit exponent width, and \( M = 5 \)-bit for the mantissa (not counting the hidden one). Let us now determine the representation of \( 9.25_{10} \) in this \((1,6,5)\) floating-point format. Using (2.24) the bias is

\[
bias = 2^{E-1} - 1 = 31,
\]

and the mantissa needs to be normalized according to the \( 1.m \) format, i.e.,

\[
9.25_{10} = 1001.01_2 = 1.00101 \times 2^3.
\]

The biased exponent is therefore represented with

\[
e = 3 + bias = 34_{10} = 100010_2.
\]

Finally, we can represent \( 9.25_{10} \) in the \((1,6,5)\) floating-point format with
2. Computer Arithmetic

<table>
<thead>
<tr>
<th>Sign s</th>
<th>Exponent e</th>
<th>Unsigned mantissa m</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>100010</td>
<td>00101</td>
</tr>
</tbody>
</table>

Besides this fixed-point to floating-point conversion we also need the back conversion from floating-point to fixed-point or integer. So, let us assume the floating-point number

<table>
<thead>
<tr>
<th>s</th>
<th>Exponent e</th>
<th>Unsigned mantissa m</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>011111</td>
<td>00000</td>
</tr>
</tbody>
</table>

is given and we wish to find the fixed-point representation of this number. We first notice that the sign bit is one, i.e., it is a negative number. Adding the hidden one to the mantissa and subtracting the bias from the exponent, yields

\[-1.00000_2 \times 2^{31-\text{bias}} = -1.02^{0} = -1.0_{10}.\]

We note that in the floating-point to fixed-point conversion the bias is subtracted from the exponent, while in the fixed-point to floating-point conversion the bias is added to the exponent.

The IEEE standard 754-2008 for binary floating-point arithmetic [51] also defines some additional useful special numbers to handle, for instance, overflow and underflow. The exponent \( e = E_{\text{max}} = 1 \ldots 1_2 \) in combination with zero mantissa \( m = 0 \) is reserved for \( \infty \). Zeros are coded with zero exponent \( e = E_{\text{min}} = 0 \) and zero mantissa \( m = 0 \). Note that due to the signed magnitude representation, plus and minus zero are coded differently. There are two more special numbers defined in the 754 IEEE standard, but these additional representations are most often not supported in FPGA floating-point arithmetic. These additional number are denormals and NaNs (not a number). With denormalized numbers we can represent numbers smaller than \( 2^{E_{\text{min}}} \) by allowing the mantissa to represent numbers without the hidden one, i.e., the mantissa can represents numbers smaller than 1.0. The exponent in denormals is coded with \( e = E_{\text{min}} = 0 \), but the mantissa is allowed to be different from zero. NaNs have proven useful in software systems to reduce the number of “exceptions” that are called when an invalid operation is performed. Examples that produce such “quiet” NaNs include:

- Addition or subtraction of two infinities, such as \( \infty - \infty \)
- Multiplication of zero and infinite, e.g., \( 0 \times \infty \)
- Division of zeros or infinities, e.g., \( 0/0 \) or \( \infty/\infty \)
- Square root of negative operand

In the IEEE standard 754 for binary floating-point arithmetic NaNs are coded with exponent \( e = E_{\text{max}} = 1 \ldots 1_2 \) in combination with a nonzero mantissa \( m \neq 0 \). Table 2.4 shows the five major floating-point codings including the special numbers.

We wish now to compare the fixed-point and floating-point representation in terms of precision and dynamic range in the following example.
Table 2.4. The five major coding types in the 754-1985 and updated 754-2008 IEEE binary floating point standard

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent $e$</th>
<th>Mantissa $m$</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0/1</td>
<td>All-zeros</td>
<td>All-zeros</td>
<td>$\pm 0$</td>
</tr>
<tr>
<td>0/1</td>
<td>All-zeros</td>
<td>Nonzero</td>
<td>Denormalized: $(-1)^s 2^{-\text{Bias}} 0.m$</td>
</tr>
<tr>
<td>0/1</td>
<td>$1 &lt; e &lt; E_{\text{max}}$</td>
<td>$m$</td>
<td>Normalized: $(-1)^s 2^{e-\text{Bias}} 1.m$</td>
</tr>
<tr>
<td>0/1</td>
<td>All-ones</td>
<td>All-zeros</td>
<td>$\pm \infty$</td>
</tr>
<tr>
<td>-</td>
<td>All-ones</td>
<td>Nonzero</td>
<td>NaN</td>
</tr>
</tbody>
</table>

Example 2.16: 12-Bit Floating- and Fixed-Point Representations

Suppose we use again a (1,6,5) floating-point format as in the previous example. The (absolute) largest number we can represent is

$$\pm 1.11111_2 \times 2^{31} \approx \pm 4.23_{10} \times 10^9.$$  

The (absolutely measured) smallest number (not including denormals) that can be represented is

$$\pm 1.0_2 \times 2^{1-\text{bias}} = \pm 1.0_2 \times 2^{-30} \approx \pm 9.31_{10} \times 10^{-10}.$$  

If we also allow denormalized numbers then the smallest number that can be represented becomes

$$\pm 0.00001_2 \times 2^{-\text{bias}} = \pm 1.0_2 \times 2^{-31-5} \approx \pm 1.45_{10} \times 10^{-11},$$

i.e., a factor 64 smaller than the normalized smallest number. Note that $e = 0$ is reserved for the denormalized in the floating-point format; see Table 2.4.

For the 12-bit fixed-point format we use one sign bit, five integer bits, and six fractional bits. The maximum (absolute) value we can represent with this 12-bit fixed-point format is therefore

$$\pm 11111.11111_2 = \pm (16 + 8 + \cdots + \frac{1}{32} + \frac{1}{64})_{10} = \pm (32 - \frac{1}{64})_{10} \approx \pm 32.0_{10}.$$  

The (absolutely measured) smallest number that this 12-bit fixed-point format represents is

$$\pm 00000.000001_2 = \pm \frac{1}{64}_{10} = \pm 0.015625_{10}.$$  

From this example we notice the larger dynamic range of the floating-point representation ($4 \times 10^9$ compared with 32 for the fixed-point) but also a higher precision of the fixed-point representation. For instance, 1.0 and $1 + \frac{1}{64} = 1.015625$ are coded the same in (1,6,5) floating-point format, but can be distinguished in 12-bit fixed-point representation.

There are two rounding mode for fixed-point type and four supported rounding modes for floating-point type which are rounding-to-nearest-even
Table 2.5. Rounding examples for the four floating-point types

<table>
<thead>
<tr>
<th>Mode</th>
<th>32.5</th>
<th>33.25</th>
<th>33.5</th>
<th>33.75</th>
<th>−32.5</th>
<th>−32.25</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rounding-to-nearest-even</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>34</td>
<td>−32</td>
<td>−32</td>
</tr>
<tr>
<td>Rounding-to-zero</td>
<td>32</td>
<td>33</td>
<td>33</td>
<td>33</td>
<td>−32</td>
<td>−32</td>
</tr>
<tr>
<td>Rounding-to-∞</td>
<td>33</td>
<td>34</td>
<td>34</td>
<td>34</td>
<td>−32</td>
<td>−32</td>
</tr>
<tr>
<td>Round-to-negative-∞</td>
<td>32</td>
<td>33</td>
<td>33</td>
<td>33</td>
<td>−33</td>
<td>−33</td>
</tr>
</tbody>
</table>

(i.e., the default), rounding-to-zero (truncation), rounding-to-∞ (round up), and round-to-negative-∞ (round down). In MATLAB the equivalent rounding functions are round(), fix(), ceil(), and floor(), respectively. The only small difference between the MATLAB and IEEE 754 modes is the rounding-to-even-nearest for numbers with $0.5_{10} = 0.1_2$ fractional part. Only if the integer LSB is one do we round up – otherwise we round down; 32.5 is rounded down but 33.5 is rounded up in the rounding-to-even-nearest scheme. Table 2.5 shows an example rounding that may occur in the (1,6,5) floating-point format. It is interesting to observe that the default operation rounding-to-nearest-even is the most complicated scheme to implement and the rounding-to-zero is not only the cheapest but also may be used to reduce an undesired gain in the processing since we always round to zero, i.e., the amplitude does not grow due to rounding.

Although the IEEE standard 754-1985 for binary floating-point arithmetic [50] is not easy to implement with all its details, such as four different rounding modes, denormals, or NaNs, the early introduction in 1985 of the standard helped as it has become the most adopted implementation for microprocessors. The parameters of this IEEE single and double format can be seen from Table 2.6. Due to the fact that already single-precision 754 standard arithmetic designs will require

- a $24 \times 24$-bit multiplier, and
- FPGAs allow a more specific dynamic range design (i.e., exponent bit width) and precision (mantissa bit width) design

we find that sometimes FPGA designer do not adopt the IEEE 754 standard and define a special format. Shirazi et al. [52], for instance, have developed a modified format to implement various algorithms on their custom computing machine called SPLASH-2, a multiple-FPGA board based on Xilinx XC4010 devices. They used an 18-bit format so that they can transport two operands over the 36-bit wide system bus of the multiple-FPGA board. The 18-bit format has a 10-bit mantissa, 7-bit exponent and a sign bit, and can represent a range of $3.7 \times 10^{19}$. 
Table 2.6. IEEE floating-point 754-2008 standard interchange formats

<table>
<thead>
<tr>
<th></th>
<th>Short</th>
<th>Single</th>
<th>Double</th>
<th>Extended</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word length</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
</tr>
<tr>
<td>Mantissa</td>
<td>10</td>
<td>23</td>
<td>52</td>
<td>112</td>
</tr>
<tr>
<td>Exponent</td>
<td>5</td>
<td>8</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>Bias</td>
<td>15</td>
<td>127</td>
<td>1023</td>
<td>16383</td>
</tr>
<tr>
<td>Range</td>
<td>$2^{16} \approx 6.4 \times 10^4$</td>
<td>$2^{128} \approx 3.8 \times 10^{38}$</td>
<td>$2^{1024} \approx 1.8 \times 10^{308}$</td>
<td>$2^{16384} \approx 10^{4932}$</td>
</tr>
</tbody>
</table>

2.3 Binary Adders

A basic binary $N$-bit adder/subtractor consists of $N$ full-adders (FA). A full-adder implements the following Boolean equations:

$$s_k = x_k \text{ XOR } y_k \text{ XOR } c_k \quad (2.25)$$

$$= x_k \oplus y_k \oplus c_k \quad (2.26)$$

that define the sum-bit. The carry (out) bit is computed with

$$c_{k+1} = (x_k \text{ AND } y_k) \text{ OR } (x_k \text{ AND } c_k) \text{ OR } (y_k \text{ AND } c_k) \quad (2.27)$$

$$= (x_k \times y_k) + (x_k \times c_k) + (y_k \times c_k) \quad (2.28)$$

In the case of a 2C adder, the LSB can be reduced to a half-adder because the carry input is zero.

The simplest adder structure is called the “ripple carry adder” as shown in Fig. 2.7a in a bit-serial form. If larger tables are available in the FPGA, several bits can be grouped together into one LUT, as shown in Fig. 2.7b. For this “two bit at a time” adder the longest delay comes from the ripple of the carry through all stages. Attempts have been made to reduce the carry delays using techniques such as the carry-skip, carry lookahead, conditional sum, or carry-select adders. These techniques can speed up addition and can be used with older-generation FPGA families (e.g., XC 3000 from Xilinx) since these devices do not provide internal fast carry logic. Modern families, such as the Xilinx Spartan or Altera Cyclone, possess very fast “ripple carry logic” that is about a magnitude faster than the delay through a regular logic LUT [1]. Altera uses fast tables (see Fig. 1.12, p. 23), while the Xilinx uses hardwired decoders for implementing carry logic based on the multiplexer structure shown in Fig. 2.8; see also Fig. 1.11, p. 21. The presence of the fast-carry logic in modern FPGA families removes the need to develop hardware intensive carry look-ahead schemes.

Figure 2.9 summarizes the size and registered performance of $N$-bit binary adders, if implemented with the `lpm_add_sub` megafunction component. Beside the EP4CE115F29C7 from the Cyclone IV E family (that is built currently using 60-nm process technology), we have also included as a reference the data for mature families. The EP20K200EFC484-2X is from the
APEX20KE family and can be found on the Nios development boards, see Chap. 9. The APEX20KE family was introduced in 1999 and used a 0.18 μm process technology. The EPF10K70RC240-4 is from the FLEX10K family and can be found on the UP2 development boards. The FLEX10K family was introduced in 1995 and used a 0.42 μm process technology. Although the LE cell structure has not changed much over time we can see from the advance in process technology the improvement in speed. If the operands are placed in I/O register cells, the delays through the busses of a FPGA are
dominant and performance decreases. If the data are routed from local registers, performance improves. For this type of design additional LE register allocation will appear (in the project report file) as increased LE use by a factor of three or four. However, a synchronous registered larger system would not consume any additional resources since the data are registered at the previous processing stage. A typical design will achieve a speed between these two cases. For Flex10K the adder and register are not merged, and $4 \times N$ LEs are required. LE requirements for the Cyclone IV and APEX devices are $3 \times N$ for the speed data shown in Fig. 2.9.

### 2.3.1 Pipelined Adders

Pipelining is extensively used in DSP solutions due to the intrinsic dataflow regularity of DSP algorithms. Programmable digital signal processor MACs [6, 16, 17] typically carry at least four pipelined stages. The processor:

1) Decodes the command
2) Loads the operands in registers
3) Performs multiplication and stores the product, and
4) Accumulates the products, all concurrently.
The pipelining principle can be applied to FPGA designs as well, at little or no additional cost since each logic element contains a flip-flop, which is otherwise unused, to save routing resources. With pipelining it is possible to break an arithmetic operation into small primitive operations, save the carry and the intermediate values in registers, and continue the calculation in the next clock cycle. Such adders are sometimes called carry save adders (CSAs) in the literature. Then the question arises: In how many pieces should we divide the adder? Should we use bit level? For Altera’s Cyclone IV devices a reasonable choice will be always using an LAB with 16 LEs and 16 FFs for one pipeline element. The FLEX10K family has 8 LEs per LAB, while APEX20KE uses 10 LEs per LAB. So we need to consult the datasheet before we make a decision on the size of the pipelining group. In fact, it can be shown that if we try to pipeline (for instance) a 14-bit adder in our Cyclone IV devices, the performance does not improve, as reported in Table 2.7, because the pipelined 14-bit adder does not fit in one LAB.

Because the number of flip-flops in one LAB is 16 and we need an extra flip-flop for the carry-out, we should use a maximum block size of 15 bits for maximum registered performance. Only the blocks with the MSBs can be 16 bits wide, because we do not need the extra flip-flop for the carry. This observation leads to the following conclusions:

1) With one additional pipeline stage we can build adders up to a length $15 + 16 = 31$.

2) With two pipeline stages we can build adders with up to $15 + 15 + 16 = 46$-bit length.

3) With three pipeline stages we can build adders with up to $15 + 15 + 15 + 16 = 61$-bit length.

Table 2.8 shows the registered performance and LE utilization of this kind of pipelined adder. From Table 2.8 it can be concluded that although the

---

Table 2.7. Performance of a 14-bit pipelined adder for the EP2C35F672C6 using synthesis of predefined LPM modules with pipeline option

<table>
<thead>
<tr>
<th>Pipeline stages</th>
<th>MHz</th>
<th>LEs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>375.94</td>
<td>42</td>
</tr>
<tr>
<td>1</td>
<td>377.79</td>
<td>56</td>
</tr>
<tr>
<td>2</td>
<td>377.64</td>
<td>70</td>
</tr>
<tr>
<td>3</td>
<td>377.79</td>
<td>84</td>
</tr>
<tr>
<td>4</td>
<td>381.10</td>
<td>98</td>
</tr>
<tr>
<td>5</td>
<td>376.36</td>
<td>112</td>
</tr>
</tbody>
</table>

---

The name carry save adder is also used in the context of a Wallace multiplier, see Exercise 2.1, p. 168.
Table 2.8. Performance and resource requirements of adders with and without pipelining. Size and speed are for the maximum bit width, for 31-, 46-, and 61-bit adders.

<table>
<thead>
<tr>
<th>Bit width</th>
<th>No pipeline</th>
<th>With pipeline</th>
<th>Pipeline</th>
<th>Design file name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MHz LEs</td>
<td>MHz LEs</td>
<td>stages</td>
<td></td>
</tr>
<tr>
<td>17 – 31</td>
<td>263.50 93</td>
<td>350.63 125</td>
<td>1</td>
<td>add1p.vhd</td>
</tr>
<tr>
<td>32 – 46</td>
<td>215.66 138</td>
<td>243.43 233</td>
<td>2</td>
<td>add2p.vhd</td>
</tr>
<tr>
<td>47 – 61</td>
<td>173.13 183</td>
<td>231.43 372</td>
<td>3</td>
<td>add3p.vhd</td>
</tr>
</tbody>
</table>

Fig. 2.10. Pipelined adder. (a) Direct implementation. (b) FPGA optimized approach.

bit width increases the registered performance remains high if we add the appropriate number of pipeline stages.

The following example shows the code of a 31-bit pipelined adder. It turns out that the straightforward implementation of the pipelining would require two registers for the MSBs as shown in Fig. 2.10a. If we instead use adders for the MSBs, we can save a set of LEs, since each LE can implement a full adder, but only one flip-flop. This is graphically interpreted by Fig. 2.10b.

Example 2.17: VHDL Design of 31-bit Pipelined Adder
Consider the VHDL code of a 31-bit pipelined adder that is graphically interpreted in Fig. 2.10. The design runs at 350.63 MHz and uses 125 LEs.

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;
-- --------------------------------------------------------
ENTITY add1p IS
GENERIC (WIDTH : INTEGER := 31; -- Total bit width
          WIDTH1 : INTEGER := 15; -- Bit width of LSBs
          WIDTH2 : INTEGER := 16); -- Bit width of MSBs
PORT (x,y : IN STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
      -- Inputs
      sum : OUT STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
      -- Result
      LSBs_carry : OUT STD_LOGIC; -- Test port
      clk : IN STD_LOGIC); -- System clock
END add1p;
-- --------------------------------------------------------
ARCHITECTURE fpga OF add1p IS
SIGNAL l1, l2, s1 -- LSBs of inputs
: STD_LOGIC_VECTOR(WIDTH1-1 DOWNTO 0);
SIGNAL r1 -- LSBs of inputs
: STD_LOGIC_VECTOR(WIDTH1 DOWNTO 0);
SIGNAL l3, l4, r2, s2 -- MSBs of inputs
: STD_LOGIC_VECTOR(WIDTH2-1 DOWNTO 0);
BEGIN
PROCESS -- Split in MSBs and LSBs and store in registers
BEGIN
WAIT UNTIL clk = '1';
-- Split LSBs from input x,y
l1 <= x(WIDTH1-1 DOWNTO 0);
l2 <= y(WIDTH1-1 DOWNTO 0);
-- Split MSBs from input x,y
l3 <= x(WIDTH-1 DOWNTO WIDTH1);
l4 <= y(WIDTH-1 DOWNTO WIDTH1);
-------------- First stage of the adder ------------------
r1 <= ('0' & l1) + ('0' & l2);
r2 <= l3 + l4;
------------ Second stage of the adder --------------------
s1 <= r1(WIDTH1-1 DOWNTO 0);
-- Add result von MSBs (x+y) and carry from LSBs
s2 <= r1(WIDTH1) + r2;
END PROCESS;
LSBs_Carry <= r1(WIDTH1); -- Add a test signal
-- Build a single output word of WIDTH=WIDTH1+WIDTH2

The equivalent Verilog code add1p.v for this example can be found in Appendix A on page 797. Synthesis results are shown in Appendix B on page 881.
The simulated performance of the 15-bit pipelined adder shows Fig. 2.11. Note that the addition results for 32780 and 32770 produce a carry from the lower 15-bit adder, but there is no carry for $32\,760 + 5 = 32\,765 < 2^{15}$. 

2.3.2 Modulo Adders

Modulo adders are the most important building blocks in RNS-DSP designs. They are used for both additions and, via index arithmetic, for multiplications. We wish to describe some design options for FPGAs in the following discussion.

A wide variety of modular addition designs exists [54]. Using LEs only, the design of Fig. 2.12a is viable for FPGAs. The Altera FLEX devices contain a small number of M2K ROMs or RAMs (EABs) that can be configured as $2^8 \times 8, 2^9 \times 4, 2^{10} \times 2$ or $2^{11} \times 1$ tables and can be used for modulo $m_l$ correction. The next table shows size and registered performance 6, 7, and 8-bit modulo adder compile for Altera FLEX10K devices [55].

<table>
<thead>
<tr>
<th>Pipeline stages</th>
<th>Bits</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPX 0</td>
<td></td>
<td>41.3</td>
<td>46.5</td>
<td>33.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPX</td>
<td>MPX</td>
<td>MPX</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LEs</td>
<td>LEs</td>
<td>LEs</td>
</tr>
<tr>
<td>MPX 2</td>
<td></td>
<td>76.3</td>
<td>62.5</td>
<td>60.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPX</td>
<td>MPX</td>
<td>MPX</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LEs</td>
<td>LEs</td>
<td>LEs</td>
</tr>
<tr>
<td>MPX 3</td>
<td></td>
<td>151.5</td>
<td>138.9</td>
<td>123.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPX</td>
<td>MPX</td>
<td>MPX</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LEs</td>
<td>LEs</td>
<td>LEs</td>
</tr>
<tr>
<td>ROM 3</td>
<td></td>
<td>86.2</td>
<td>86.2</td>
<td>86.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ROM</td>
<td>ROM</td>
<td>ROM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EAB</td>
<td>EAB</td>
<td>EAB</td>
</tr>
</tbody>
</table>

Although the ROM shown in Fig. 2.12 provides high speed, the ROM itself produces a four-cycle pipeline delay and the number of ROMs is limited.
ROMs, however, are mandatory for the scaling schemes discussed before. The multiplexed-adder (MPX-Add) has a comparatively reduced speed even if a carry chain is added to each column. The pipelined version usually needs the same number of LEs as the unpipelined version but runs about three times as fast. Maximum throughput occurs when the adders are implemented with 3 pipeline stages and 6-bit width channels.

### 2.4 Binary Multipliers

The product of two $N$-bit binary numbers, say $X$ and $A = \sum_{k=0}^{N-1} a_k 2^k$, is given by the “pencil and paper” method as:

$$P = A \times X = \sum_{k=0}^{N-1} a_k 2^k X.$$  \hspace{1cm} (2.29)

It can be seen that the input $X$ is successively shifted by $k$ positions and whenever $a_k \neq 0$, then $X 2^k$ is accumulated. If $a_k = 0$, then the corresponding shift-add can be ignored (i.e., nop). With the introduction of embedded multipliers in recent FPGAs this FSM approach is not used often.

Because one operand is used in parallel (i.e., $X$) and the second operand $A$ is used bitwise, the multipliers we just described are called serial/parallel multipliers. If both operands are used serial, the scheme is called a serial/serial
multiplier [56], and such a multiplier only needs one full adder, but the latency of serial/serial multipliers is high $\mathcal{O}(N^2)$, because the state machine needs about $N^2$ cycles.

Another approach, which trades speed for increased complexity, is called an “array,” or parallel/parallel multiplier. A $4 \times 4$-bit array multiplier is shown in Fig. 2.13. Notice that both operands are presented in parallel to an adder array of $N^2$ adder cells.

This arrangement is viable if the times required to complete the carry and sum calculations are the same. For a modern FPGA, however, the carry computation is performed faster than the sum calculation and a different architecture is more efficient for FPGAs. The approach for this array multiplier is shown in Fig. 2.14, for an $8 \times 8$-bit multiplier. This scheme combines in the first stage two neighboring partial products $a_n X 2^n$ and $a_{n+1} X 2^{n+1}$ and the results are added to arrive at the final output product. This is a direct array form of the “pencil and paper” method and must therefore produce a valid product.

We recognize from Fig. 2.14 that this type of array multiplier gives the opportunity to realize a (parallel) binary tree of the multiplier with a total:

$$\text{number of stages in the binary tree multiplier} = \log_2(N).$$  (2.30)
This alternative architecture also makes it easier to introduce pipeline stages after each tree level. The necessary number of pipeline stages, according to (2.30), to achieve maximum throughput is:

<table>
<thead>
<tr>
<th>Bit width</th>
<th>2</th>
<th>3 – 4</th>
<th>5 – 8</th>
<th>9 – 16</th>
<th>17 – 32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal number of pipeline stages</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Since the data are registered at the input and output the number of delays in the simulation would be two larger than the pipeline stage we specified for the lpm_mul blocks.

Figure 2.15 reports the \( \text{Fmax} \) registered performance of pipelined \( N \times N \)-bit multipliers, using the Quartus II lpm_mult function, for \( 8 \times 8 \), to \( 24 \times 24 \) bits operands. Embedded multiplier are shown with dash lines and up to \( 16 \times 16 \)-bit the multiplier do not improve with pipelining since they fit in one embedded \( 18 \times 18 \)-bit array multiplier. The LE-based multiplier are shown with a solid line. Figure 2.16 shows the LEs effort for the multiplier. The pipelined \( 8 \times 8 \) bit multiplier outperforms the embedded multiplier if 2 or more pipeline stages are used. We can notice from Fig. 2.15 that, for pipeline delays longer than \( \log_2(N) \), there is no essential improvement for LE-based multi-
2.4 Binary Multipliers

Fig. 2.15. Performance of an array multiplier for FPGAs, LE-based multiplier (solid line) and embedded multiplier (dashed line)

pliers. The multiplier architecture (embedded or LEs) must be controlled via synthesis options in case we write behavioral code (e.g., `p <= a*b`). This can be done under Setting in the Assignments menu. There you find the DSP Block Balancing entry under the Analysis & Synthesis Settings → More Settings. Select DSP blocks if you like to use the embedded multiplier, Logic Elements to use the LEs only, or Auto, and the synthesis tool will first use the embedded multiplier; if there are not enough then use the LE-based multiplier. If we use the `lpm_mul` block we have direct control using the GENERIC MAP parameter DEDICATED_MULTIPLIER_CIRCUITRY => "YES" or "NO".

Other multiplier architectures typically used in the ASIC world include Wallace-tree multipliers and Booth multipliers. They are discussed in Exercises 2.1 (p. 168) and 2.2 (p. 169) but are rarely used in connection with FPGAs.

2.4.1 Multiplier Blocks

A $2N \times 2N$ multiplier can be defined in terms of an $N \times N$ multiplier block [33]. The resulting multiplication is defined as:
Fig. 2.16. Effort in LEs for array multipliers, LE-based multiplier (solid line) and embedded multiplier (dashed line)

\[
P = Y \times X = (Y_2 2^N + Y_1)(X_2 2^N + X_1)
= Y_2 X_2 2^{2N} + (Y_2 X_1 + Y_1 X_2) 2^N + Y_1 X_1,
\]

where the indices 2 and 1 indicate the most significant and least significant \(N\)-bit halves, respectively. This partitioning scheme can be used if the capacity of the FPGA is insufficient to implement a multiplier of desired size, or used to implement a multiplier using memory blocks. A \(36 \times 36\)-bit multiplier can be build with four \(18 \times 18\)-bit embedded multipliers and three adders. An \(8 \times 8\)-bit LUT-based multiplier in direct form would require an LUT size of \(2^{16} \times 16 = 1\) Mbit. The partitioning technique reduces the table size to four \(2^8 \times 8\) memory blocks and three adders. A \(16 \times 16\)-bit multiplier requires 16 memory blocks. The benefit of multiplier implementation via M9Ks versus LE-based is twofold. First, the number of LE is reduced. Secondly, the requirements on the routing resources of the devices are also reduced. Although some FPGAs families now have a limited number of embedded array multipliers, the number is usually small, and the LUT-based multiplier provides a way to enlarge the number of fast low-latency multipliers in these devices. In addition, some device families like Cyclone, Flex, or Excalibur do not have
embedded multipliers; therefore, the LUT or LE multipliers are the only option.

**Half-Square Multiplier**

Another way to reduce the memory requirement for LUT-based multipliers is to decrease the bits in the input domain. One bit decrease in the input domain decreases the number of LUT words by a factor of two. An LUT of a square operation of an $N$-bit word only requires an LUT size of $2^N \times 2^N$. The additive half-square (AHSM) multiplier

$$Y \times X = \frac{(X + Y)^2 - X^2 - Y^2}{2} = \left\lfloor \frac{(X + Y)^2}{2} \right\rfloor - \left\lfloor \frac{X^2}{2} \right\rfloor - \left\lfloor \frac{Y^2}{2} \right\rfloor - \left\lfloor \frac{(X - Y)^2}{2} \right\rfloor + \{ \begin{array}{ll} 1 & X, Y \text{ odd} \\ 0 & \text{others} \end{array} \right. \quad (2.33)$$

was introduced by Logan [57]. If the division by 2 is included in the LUT, this requires a correction of $-1$ in the event that $X$ and $Y$ are odd. A differential half-square multiplier (DHSM) can then be implemented as:

$$Y \times X = \frac{(X + Y)^2 - X^2 - Y^2}{2} = \left\lfloor \frac{X^2}{2} \right\rfloor + \left\lfloor \frac{Y^2}{2} \right\rfloor - \left\lfloor \frac{(X - Y)^2}{2} \right\rfloor + \{ \begin{array}{ll} 1 & X, Y \text{ odd} \\ 0 & \text{others} \end{array} \right. \quad (2.34)$$
A correction of 1 is required in the event that \( X \) and \( Y \) are odd. If the numbers are signed, an additional saving is possible by using the diminished-by-one (D1) encoding, see Sect. 2.2.1, p. 60. In D1 coding all numbers are diminished by 1, and the zero gets special encoding [58]. Figure 2.17 shows for 8-bit data the AHSM multiplier, the required LUTs, and the data range of 8-bit input operands. The absolute operation almost allows a reduction by a factor of 2 in LUT words, while the D1 encoding enables a reduction to the next power-of-two table size that is beneficial for the FPGA design. Since LUT inputs 0 and 1 both have the same square, LUT entry \([A^2/2]\), we share this value and do not need to use special encoding for zero. Without the division by 2, a 17-bit output word would be required. However, the division by two in the squaring table requires an increment (decrement) of the output result for the AHSM (DHSM) in case both input operands are odd values. Figure 2.18 shows a DHSM multiplier that only requires two D1 encoding compared with the AHSM design.

**Quarter-Square Multiplier**

A further reduction in arithmetic requirements and the number of LUTs can be achieved by using the quarter-square multiplication (QSM) principle that is also well studied in analog designs [59, 60]. The QSM is based on the following equation:

\[
Y \times X = \left\lfloor \frac{(X + Y)^2}{4} \right\rfloor - \left\lfloor \frac{(X - Y)^2}{4} \right\rfloor.
\]
It is interesting to note that the division by 4 in (2.35) does not require any correction for operation as in the HSM case. This can be checked as follows. If both operands are even (odd), then the sum and the difference are both even, and the squaring followed by a division of 4 produces no error (i.e., $4| (2u * 2v)$). If one operand is odd (even) and the other operand is even (odd), then the sum and the difference after squaring and a division by 4 produce a 0.25 error that is annihilated in both cases. No correction operation is necessary. The direct implementation of (2.35) would require LUTs of $(N + 1)$-bit inputs to represent the correct result of $X \pm Y$ as used in [61], which will require four $2^N \times 2^N$ LUTs. Signed arithmetic along with D1 coding will reduce the table to the next power-of-two value, allowing the design to use only two $2^N \times 2^N$ LUTs compared with the four in [61]. Figure 2.19 shows the D1 QSM circuit.

Synthesis results for AHSM, DHSM, and QSM can be found in [34].

2.5 Binary Dividers

Of all four basic arithmetic operations division is the most complex. Consequently, it is the most time-consuming operation and also the operation with the largest number of different algorithms to be implemented. For a given dividend (or numerator) $N$ and divisor (or denominator) $D$ the division produces (unlike the other basic arithmetic operations) two results: the quotient $Q$ and the remainder $R$, i.e.,

$$\frac{N}{D} = Q \text{ and } R \text{ with } |R| < D.$$  \hfill (2.35)

However, we may think of division as the inverse process of multiplication, as demonstrated through the following equation,
\[ N = D \times Q + R, \quad (2.36) \]

It differs from multiplication in many aspects. Most importantly, in multiplication all partial products can be produced parallel, while in division each quotient bit is determined in a sequential “trail-and-error” procedure.

Because most microprocessors handle division as the inverse process to multiplications, referring to (2.36), the numerator is assumed to be the result of a multiplication and has therefore twice the bit width of denominator and quotient. As a consequence, the quotient has to be checked in an awkward procedure to be in the valid range, i.e., that there is no overflow in the quotient. We wish to use a more general approach in which we assume that

\[ Q \leq N \quad \text{and} \quad |R| \leq D, \]

i.e., quotient and numerator as well as denominator and remainder are assumed to be of the same bit width. With this bit width assumptions no range check (except \( N = 0 \)) for a valid quotient is necessary.

Another consideration when implementing division comes when we deal with signed numbers. Obviously, the easiest way to handle signed numbers is first to convert both to unsigned numbers and compute the sign of the result as an XOR or modulo 2 add operation of the sign bits of the two operands. But some algorithms, (like the nonrestoring division discussed below), can directly process signed numbers. Then the question arises, how are the sign of quotient and remainder related. In most hardware or software systems (but not for all, such as in the PASCAL programming language), it is assumed that the remainder and the quotient have the same sign. That is, although \[ \frac{234}{50} = 5 \quad \text{and} \quad R = -16 \quad (2.37) \]

meets the requirements from (2.36), we, in general, would prefer the following results:

\[ \frac{234}{50} = 4 \quad \text{and} \quad R = 34. \quad (2.38) \]

Let us now start with a brief overview of the most commonly used division algorithms. Figure 2.20 shows the most popular linear and quadratic convergence schemes. A basic categorization of the linear division algorithms can be done according to the permissible values of each quotient digit generated. In the binary restoring, nonperforming or CORDIC algorithms the digits are selected from the set \( \{0, 1\} \).

In the binary nonrestoring algorithms a signed-digit set is used, i.e.,

\[ \{-1, 1\} = \{\text{T}, 1\}. \]

In the binary SRT algorithm, named after Sweeney, Robertson, and Tocher [33] who discovered the algorithms at about the same time, the digits from the ternary set
are used. All of the above algorithms can be extended to higher radix algorithms. The generalized SRT division algorithms of radix $r$, for instance, uses the digit set
\[ \{-2r^{-1}, \ldots, -1, 0, 1, \ldots, 2r^{-1}\} \].

We find two algorithms with quadratic convergence to be popular. The first algorithm is the division by reciprocation of the denominator, where we compute the reciprocal with the Newton algorithm for finding zeros. The second quadratic convergence algorithm was developed for the IBM 360/91 in the 1960s by Anderson et al. [62]. This algorithm multiplies numerator and denominator with the same factors and converges $N \to 1$, which results in $D \to Q$. Note, that the division algorithms with quadratic convergence produce no remainder.

Although the number of iterations in the quadratic convergence algorithms are in the order of $\log_2(b)$ for $b$ bit operands, we must take into account that each iteration step is more complicated (i.e., uses two multiplications) than the linear convergence algorithms, and speed and size performance comparisons have to be done carefully.

### 2.5.1 Linear Convergence Division Algorithms

The most obvious sequential algorithms is our “pencil-and-paper” method (which we have used many times before) translated into binary arithmetic. We align first the denominator and load the numerator in the remainder register. We then subtract the aligned denominator from the remainder and store the result in the remainder register. If the new remainder is positive
we set the quotient’s LSB to 1, otherwise the quotient’s LSB is set to zero and we need to restore the previous remainder value by adding the denominator. Finally, we have to realign the quotient and denominator for the next step. The recalculation of the previous remainder is why we call such an algorithm “restoring division.” The following example demonstrates a FSM implementation of the algorithm.

Example 2.18: 8-bit Restoring Divider

The VHDL description\(^6\) of an 8-bit divider is developed below. Division is performed in four stages. After reset, the 8-bit numerator is “loaded” in the remainder register, the 6-bit denominator is loaded and aligned (by \(2^{N-1}\) for a \(N\) bit numerator), and the quotient register is set to zero. In the second and third stages, sub and restore, the actual serial division takes place. In the fourth step, done, quotient and remainder are transferred to the output registers. Nominator and quotient are assumed to be 8 bits wide, while denominator and remainder are 6-bit values.

```vhdl
-- Restoring Division
LIBRARY ieee; USE ieee.std_logic_1164.ALL;
PACKAGE n_bits_int IS -- User defined types
  SUBTYPE SLVN IS STD_LOGIC_VECTOR(7 DOWNTO 0);
  SUBTYPE SLVD IS STD_LOGIC_VECTOR(5 DOWNTO 0);
END n_bits_int;
LIBRARY work; USE work.n_bits_int.ALL;
LIBRARY ieee; -- Using predefined packages
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;
-- --------------------------------------------------------
ENTITY div_res IS ------> Interface
  GENERIC(WN : INTEGER := 8;
    WD : INTEGER := 6;
    PO2WND : INTEGER := 8192; -- 2**(WN+WD)
    PO2WN1 : INTEGER := 128; -- 2**(WN-1)
    PO2WN : INTEGER := 255); -- 2**WN-1
  PORT(clk : IN STD_LOGIC; -- System clock
     reset : IN STD_LOGIC; -- Asynchronous reset
     n_in : IN SLVN; -- Nominator
     d_in : IN SLVD; -- Denumerator
     r_out : OUT SLVD; -- Remainder
     q_out : OUT SLVN); -- Quotient
END div_res;
-- --------------------------------------------------------
ARCHITECTURE fpga OF div_res IS

  SUBTYPE S14 IS INTEGER RANGE -PO2Wnd TO PO2Wnd-1;
  SUBTYPE U8 IS INTEGER RANGE 0 TO PO2WN;
  SUBTYPE U4 IS INTEGER RANGE 0 TO WN;

  TYPE STATE_TYPE IS (ini, sub, restore, done);

6 The equivalent Verilog code `div_res.v` for this example can be found in Appendix A on page 801. Synthesis results are shown in Appendix B on page 881.
2.5 Binary Dividers

SIGNAL state : STATE_TYPE;
BEGIN
-- Bit width: WN WD WN WD
-- Numerator / Denumerator = Quotient and Remainder
-- OR: Nominator = Quotient * Denumerator + Remainder
States: PROCESS(reset, clk) -- Divider in behavioral style
VARIABLE r, d : S14 := 0; -- N+D bit width
VARIABLE q : U8;
VARIABLE count : U4;
BEGIN
IF reset = '1' THEN -- asynchronous reset
state <= ini; q_out <= (OTHERS => '0');
r_out <= (OTHERS => '0');
ELSIF rising_edge(clk) THEN
CASE state IS
WHEN ini => -- Initialization step
state <= sub;
count := 0;
q := 0; -- Reset quotient register
d := PO2WN1 * CONV_INTEGER(d_in); -- Load denumerator.
r := CONV_INTEGER(n_in); -- Remainder = nominator
WHEN sub => -- Processing step
r := r - d; -- Subtract denumerator
state <= restore;
WHEN restore => -- Restoring step
IF r < 0 THEN
r := r + d; -- Restore previous remainder
q := q * 2; -- LSB = 0 and SLL
ELSE
q := 2 * q + 1; -- LSB = 1 and SLL
END IF;
count := count + 1;
d := d / 2;
IF count = WN THEN -- Division ready?
state <= done;
ELSE
state <= sub;
END IF;
WHEN done => -- Output of result
q_out <= CONV_STD_LOGIC_VECTOR(q, WN);
r_out <= CONV_STD_LOGIC_VECTOR(r, WD);
state <= ini; -- Start next division
END CASE;
END IF;
END PROCESS States;
END fpga;

Figure 2.21 shows the simulation result of a division of 234 by 50. The register d shows the aligned denominator values $50 \times 2^7 = 6400$, $50 \times 2^6 = 3200$, .... Every time the remainder r calculated in step sub is negative, the previous remainder is restored in step restore. In state done the quotient 4 and the
remainder 34 are transferred to the output registers of the divider. The design uses 106 LEs, no embedded multiplier, and runs with a registered performance of $F_{\text{max}} = 263.5$ MHz using the \textit{TimeQuest} slow 85C model.

The main disadvantage of the restoring division is that we need two steps to determine one quotient bit. We can combine the two steps using a \textit{non-performing} divider algorithm, i.e., each time the denominator is larger than the remainder, we do \textit{not} perform the subtraction. In VHDL we would write the new step as:

\begin{verbatim}
  t := r - d; -- temporary remainder value
  IF t >= 0 THEN -- Nonperforming test
    r := t; -- Use new denominator
    q := q \times 2 + 1; -- LSB = 1 and SLL
  ELSE
    q := q \times 2; -- LSB = 0 and SLL
  END IF;
\end{verbatim}

The number of steps is reduced by a factor of 2 (not counting initialization and transfers of results), as can be seen from the simulation in Fig. 2.22. Note also from the simulation shown in Fig. 2.22 that the remainder $r$ is never negative in the nonperforming division algorithms. On the downside the worst case delay path is increased when compared with the restoring division and the maximum registered performance is expected to be reduced; see Exercise 2.17 (p. 171). The nonperforming divider has two arithmetic operations and the if condition in the worst case path, while the restoring divider has (see step s2) only the if condition and one arithmetic operation in the worst case path.

A similar approach to the nonperforming algorithm, but that does \textit{not} increase the critical path, is the so-called \textit{nonrestoring} division. The idea behind the nonrestoring division is that if we have computed in the restoring division a negative remainder, i.e., $r_{k+1} = r_k - d_k$, then in the next step we will restore $r_k$ by adding $d_k$ and then perform a subtraction of the next aligned denominator $d_{k+1} \neq d_k/2$. So, instead of adding $d_k$ followed by subtracting $d_k/2$, we can just skip the restoring step and proceed with adding $d_k/2$, when
the remainder has (temporarily) a negative value. As a result, we have now quotient bits that can be positive or negative, i.e., \( q_k = \pm 1 \), but not zero. We can change this signed-digit representation later to a two’s complement representation. In conclusion, the nonrestoring algorithms works as follows: every time the remainder after the iteration is positive we store a 1 and subtract the aligned denominator, while for negative remainder, we store a \(-1 = \bar{1}\) in the quotient register and add the aligned denominator. To use only one bit in the quotient register we will use a zero in the quotient register to code the \(-1\). To convert this signed-digit quotient back to a two’s complement word, the straightforward way is to put all 1s in one word and the zeros, which are actually the coded \(-1 = \bar{1}\) in the second word as a one. Then we need just to subtract the two words to compute the two’s complement. On the other hand this subtraction of the \(-1s\) is nothing other than the complement of the quotient augmented by 1. In conclusion, if \( q \) holds the signed-digit representation, we can compute the two’s complement via

\[
q_{2C} = 2 \times q_{SD} + 1. \tag{2.39}
\]

Both quotient and remainder are now in the two’s complement representation and have a valid result according to (2.36). If we wish to constrain our results in a way that both have the same sign, we need to correct the negative remainder, i.e., for \( r < 0 \) we correct this via

\[
r := r + D \quad \text{and} \quad q := q - 1.
\]

Such a nonrestoring divider will now run faster than the nonperforming divider, with about the same registered performance as the restoring divider; see Exercise 2.18 (p. 171). Figure 2.23 shows a simulation of the nonrestoring divider. We notice from the simulation that register values of the remainder are allowed now again to be negative. Note also that the above-mentioned correction for negative remainder is necessary for this value. The not corrected result is \( q = 5 \) and \( r = -16 \) The equal sign correction results in \( q = 5 - 1 = 4 \) and \( r = -16 + 50 = 34 \), as shown in Fig. 2.23.
To shorten further the number of clock cycles needed for the division higher radix (array) divider can be built using, for instance, the SRT and radix 4 coding. This is popular in ASIC designs when combined with the carry-save-adder principle as used in the floating-point accelerators of the Pentium microprocessors. For FPGAs with a limited LUT size this higher-order schemes seem to be less attractive.

A totally different approach to improve the latency are the division algorithms with quadratic convergence, which use fast array multiplier. The two most popular versions of this quadratic convergence schemes are discussed in the next section.

2.5.2 Fast Divider Design

The first fast divider algorithm we wish to discuss is the division through multiplication with the reciprocal of the denominator \( D \). The reciprocal can, for instance, be computed via a look-up table for small bit width. The general technique for constructing iterative algorithms, however, makes use of the Newton method for finding a zero. According to this method, we define a function

\[
f(x) = \frac{1}{x} - D \rightarrow 0. \tag{2.40}
\]

If we define an algorithm such that \( f(x_\infty) = 0 \) then it follows that

\[
\frac{1}{x_\infty} - D = 0 \quad \text{or} \quad x_\infty = \frac{1}{D}. \tag{2.41}
\]

Using the tangent the estimation for the next \( x_{k+1} \) is calculated using

\[
x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}, \tag{2.42}
\]

with \( f(x) = 1/x - D \) we have \( f'(x) = 1/x^2 \) and the iteration equation becomes

\[
x_{k+1} = x_k - \frac{1}{x_k} \cdot \frac{1}{x_k} = x_k \left(2 - D \times x_k\right). \tag{2.43}
\]
Although the algorithm will converge for any initial \( D \), it converges much faster if we start with a normalized value close to 1.0, i.e., we normalized \( D \) in such a way that \( 0.5 \leq D < 1 \) or \( 1 \leq D < 2 \) as used for floating-point mantissa, see Sect. 2.7 (p. 109). We can then use an initial value \( x_0 = 1 \) to get fast convergence. Let us illustrate the Newton algorithm with a short example.

**Example 2.19: Newton Algorithm**

Let us try to compute the Newton algorithm for \( 1/D = 1/0.8 = 1.25 \). The following table shows in the first column the number of the iteration, in the second column the approximation to \( 1/D \), in the third column the error \( x_k - x_\infty \), and in the last column the equivalent bit precision of our approximation.

<table>
<thead>
<tr>
<th>( k )</th>
<th>( x_k )</th>
<th>( x_k - x_\infty )</th>
<th>Eff. bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.0</td>
<td>-0.25</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1.2</td>
<td>-0.05</td>
<td>4.3</td>
</tr>
<tr>
<td>2</td>
<td>1.248</td>
<td>-0.002</td>
<td>8.9</td>
</tr>
<tr>
<td>3</td>
<td>1.25</td>
<td>-3.2 \times 10^{-6}</td>
<td>18.2</td>
</tr>
<tr>
<td>4</td>
<td>1.25</td>
<td>-8.2 \times 10^{-12}</td>
<td>36.8</td>
</tr>
</tbody>
</table>

Figure 2.24 shows a graphical interpretation of the Newton zero-finding algorithm. The \( f(x_k) \) converges rapidly to zero.

Because the first iterations in the Newton algorithm only produce a few bits of precision, it may be useful to use a small look-up table to skip the first iterations. A table to skip the first two iterations can, for instance, be found in [33, p. 260].
We note also from the above example the overall rapid convergence of the algorithm. Only 5 steps are necessary to have over 32-bit precision. Many more steps would be required to reach the same precision with the linear convergence algorithms. This quadratic convergence applies for all values not only for our special example. This can be shown as follows:

\[ e_{k+1} = x_{k+1} - x_{\infty} = x_k(2 - D \times x_k) - \frac{1}{D} \]

\[ = -D \left( x_k - \frac{1}{D} \right)^2 = -De_k^2, \]

i.e., the error improves in a quadratic fashion from one iteration to the next. With each iteration we double the effective number of bit precision.

Although the Newton algorithm has been successfully used in microprocessor design (e.g., IBM RISC 6000), it has two main disadvantages: First, the two multiplications in each iteration are sequential, and second, the quantization error of the multiplication is accumulated due to the sequential nature of the multiplication. Additional guard bits are used in general to avoid this quantization error.

The following convergence algorithm, although similar to the Newton algorithm, has an improved quantization behavior and uses 2 multiplications in each iteration that can be computed parallel. In the convergence division scheme both numerator \( N \) and denominator \( D \) are multiplied by approximation factors \( f_k \), which, for a sufficient number of iterations \( k \), we find

\[ D \prod f_k \rightarrow 1 \quad \text{and} \quad N \prod f_k \rightarrow Q. \quad (2.44) \]

This algorithm, originally developed for the IBM 360/91, is credited to Anderson et al. [62], and the algorithm works as follows:

<table>
<thead>
<tr>
<th>Algorithm 2.20: Division by Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Normalize ( N ) and ( D ) such that ( D ) is close to 1. Use a normalization interval such as ( 0.5 \leq D &lt; 1 ) or ( 1 \leq D &lt; 2 ) as used for floating-point mantissa.</td>
</tr>
<tr>
<td>2) Initialize ( x_0 = N ) and ( t_0 = D ).</td>
</tr>
<tr>
<td>3) Repeat the following loop until ( x_k ) shows the desired precision.</td>
</tr>
<tr>
<td>( f_k = 2 - t_k )</td>
</tr>
<tr>
<td>( x_{k+1} = x_k \times f_k )</td>
</tr>
<tr>
<td>( t_{k+1} = t_k \times f_k )</td>
</tr>
</tbody>
</table>

It is important to note that the algorithm is self-correcting. Any quantization error in the factors does not really matter because numerator and denominator are multiplied with the same factor \( f_k \). This fact has been used in the IBM 360/91 design to reduce the required resources. The multiplier used for the first iteration has only a few significant bits, while in later iteration more multiplier bits are allocated as the factor \( f_k \) gets closer to 1.
Let us demonstrate the multiply by convergence algorithm with the following example.

**Example 2.21: Anderson–Earle–Goldschmidt–Powers Algorithm**

Let us try to compute the division-by-convergence algorithm for \( N = 1.5 \) and \( D = 1.2 \), i.e., \( Q = \frac{N}{D} = 1.25 \) The following table shows in the first column the number of the iteration, in the second column the scaling factor \( f_k \), in the third column the approximation to \( \frac{N}{D} \), in the fourth column the error \( x_k - x_\infty \), and in the last column the equivalent bit precision of our approximation.

<table>
<thead>
<tr>
<th>( k )</th>
<th>( f_k )</th>
<th>( x_k )</th>
<th>( x_k - x_\infty )</th>
<th>Eff. bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.8 ( \approx \frac{205}{256} )</td>
<td>1.5 ( \approx \frac{384}{256} )</td>
<td>0.25</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1.04 ( \approx \frac{267}{256} )</td>
<td>1.2 ( \approx \frac{307}{256} )</td>
<td>-0.05</td>
<td>4.3</td>
</tr>
<tr>
<td>2</td>
<td>1.0016 ( \approx \frac{256}{256} )</td>
<td>1.248 ( \approx \frac{320}{256} )</td>
<td>0.002</td>
<td>8.9</td>
</tr>
<tr>
<td>3</td>
<td>1.0 + 2.56 \times 10^{-6}</td>
<td>1.25</td>
<td>(-3.2 \times 10^{-6})</td>
<td>18.2</td>
</tr>
<tr>
<td>4</td>
<td>1.0 + 6.55 \times 10^{-12}</td>
<td>1.25</td>
<td>(-8.2 \times 10^{-12})</td>
<td>36.8</td>
</tr>
</tbody>
</table>

We notice the same quadratic convergence as in the Newton algorithm; see Example 2.19 (p. 101).

The VHDL description\(^7\) of an 8-bit fast divider is developed below. We assume that denominator and numerator are normalized as, for instance, typical for floating-point mantissa values, to the interval \( 1 \leq N, D < 2 \). This normalization step may require essential addition resources (leading-zero detection and two barrelshifters) when the denominator and numerator are not normalized. Nominator, denominator, and quotient are all assumed to be 9 bits wide. The decimal values 1.5, 1.2, and 1.25 are represented in a 1.8-bit format (1 integer and 8 fractional bits) as \( 1.5 \times 256 = 384 \), \( 1.2 \times 256 = 307 \), and \( 1.25 \times 256 = 320 \), respectively. Division is performed in three stages. First, the 1.8-formatted denominator and numerator are loaded into the registers. In the second state, run, the actual convergence division takes place. In the third step, done, the quotient is transferred to the output register.

```vhdl
-- Convergence division after Anderson, Earle, Goldschmidt,
LIBRARY ieee; USE ieee.std_logic_1164.ALL; -- and Powers

PACKAGE n_bits_int IS -- User defined types
  SUBTYPE U3 IS INTEGER RANGE 0 TO 7;
  SUBTYPE U10 IS INTEGER RANGE 0 TO 1023;
  SUBTYPE SLVN IS STD_LOGIC_VECTOR(8 DOWNTO 0);
  SUBTYPE SLVD IS STD_LOGIC_VECTOR(8 DOWNTO 0);
END n_bits_int;

LIBRARY work;
USE work.n_bits_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;
```  

\(^7\) The equivalent Verilog code `div_aegp.v` for this example can be found in Appendix A on page 803. Synthesis results are shown in Appendix B on page 881.
ENTITY div_aegp IS  
--- Interface
GENERIC(WN : INTEGER := 9; -- 8 bit plus one integer bit
WD : INTEGER := 9;
STEPS : INTEGER := 2;
TWO : INTEGER := 512; -- 2**(WN+1)
P02WN : INTEGER := 256; -- 2**(WN-1)
P02WN2 : INTEGER := 1023); -- 2**(WN+1)-1
PORT (clk : IN STD_LOGIC; -- System clock
reset : IN STD_LOGIC; -- Asynchronous reset
n_in : IN SLVN; -- Nominator
d_in : IN SLVD; -- Denumerator
q_out : OUT SLVD); -- Quotient
END div_aegp;
-- --------------------------------------------------------
ARCHITECTURE fpga OF div_aegp IS

TYPE STATE_TYPE IS (ini, run, done);
SIGNAL state : STATE_TYPE;
BEGIN
-- Bit width: WN WD WN WD
-- Nominator / Denumerator = Quotient and Remainder
-- OR: Nominator = Quotient * Denumerator + Remainder
States: PROCESS(reset, clk) -- Divider in behavioral style
VARIABLE x, t, f : U10 := 0; -- WN+1 bits
VARIABLE count : INTEGER RANGE 0 TO STEPS;
BEGIN
IF reset = '1' THEN -- Asynchronous reset
state <= ini;
q_out <= (OTHERS => '0');
ELSIF rising_edge(clk) THEN
CASE state IS
WHEN ini => -- Initialization step
state <= run;
count := 0;
t := CONV_INTEGER(d_in); -- Load denominator
x := CONV_INTEGER(n_in); -- Load nominator
WHEN run => -- Processing step
f := TWO - t;
x := x * f / P02WN;
t := t * f / P02WN;
count := count + 1;
IF count = STEPS THEN -- Division ready ?
state <= done;
ELSE
state <= run;
END IF;
ELSE
END CASE;
WHEN done => -- Output of results
q_out <= CONV_STD_LOGIC_VECTOR(x, WN);
state <= ini; -- start next division
END CASE;
END IF;
Fig. 2.25. Simulation results for a convergence divider

END PROCESS States;

END fpga;

Figure 2.25 shows the simulation result of the division 1.5/1.2. The variable $f$ (which becomes an internal net and is not shown in the simulation) holds the three scaling factors 205, 267, and 257, sufficient for 8-bit precision results. The $x$ and $t$ values are multiplied by the scaling factor $f$ and scaled down to the 1.8 format. $x$ converges to the quotient $1.25=320/256$, while $t$ converges to $1.0 = 255/256$, as expected. In state done the quotient $1.25 = 320/256$ is transferred to the output registers of the divider. Note that the divider produces no remainder. The design uses 45 LEs, four embedded multipliers and runs with a registered performance of $F_{\text{max}}=124.91 \, \text{MHz}$ using the TimeQuest slow 85C model.

Although the registered performance of the nonrestoring divider (see Fig. 2.23) is about twice as high, the total latency, however, in the convergence divider is reduced, because the number of processing steps are reduced from 8 to $\lceil \sqrt{8} \rceil = 3$ (not counting initialization in both algorithms). The convergence divider uses less LEs as the nonrestoring divider but also four embedded multipliers.

2.5.3 Array Divider

Obviously, as with multipliers, all division algorithms can be implemented in a sequential, FSM-like, way or in the array form. If the array form and pipelining is desired, a good option will then be to use the lpm_divide block, which implements an array divider with the option of pipelining. If no pipelining is required then Quartus II 12.1 is able to synthesis behavior code such as

```vhdl
SIGNAL n, d, q : INTEGER RANGE -128 TO 127;
```
Figure 2.26 shows the registered performance using the *TimeQuest* slow 85C model and Fig. 2.27 the LEs necessary for 8 × 8-, 16 × 16-, and 24 × 24-bit array dividers including the 4 I/O registers. Note the logarithmic like scaling for the number of pipeline stages on the horizontal axis. We conclude from the performance measurement that the optimal number of pipeline stages is the same as the number of bits in the denominator.

### 2.6 Fixed-Point Arithmetic Implementation

A substantial addition in VHDL-2008 relevant to DSP is the introduction of the unsigned and signed fixed-point data types `ufixed` and `sfixed` and the `float` data types discussed in the next section. Based on the Appendix G (pp. 522–537) of the VHDL-2008 LRM several textbooks [63–65] now cover this new data types and the operations too. Since in DSP we more often deal with signed than unsigned numbers, let us in the following focus on the `sfixed` data types. Although it is part of the VHDL-2008 standard, efforts
have been made for legacy support that the package is also available for VHDL-1993 through the call of an additional library. There are two files to be included. The first includes the type configuration definitions and the second library file includes the operation and conversion functions. In VHDL-1993 we would write

```vhdl
LIBRARY ieee_proposed;
USE ieee_proposed.fixed_float_types.ALL;
USE ieee_proposed.fixed_pkg.all;
```

These files are available free of charge from [www.eda.org/fphdl](http://www.eda.org/fphdl). The library is a complete package of operations and functions written by David Bishop and includes over 8K lines of VHDL code. At the time of writing, an equivalent Verilog code is, according to David Bishop, currently in progress and it is the hope that a Verilog version may become part of a future standard too. Since most vendors support only a subset of standard VHDL language, small modified versions are available that have been tested for Altera, Xilinx, Synopsys, Cadence, and MentorGraphics (ModelSim) tools. Here is a brief listing of supported operations and functions relevant to DSP for the `sfixed` data type:
Some of the arithmetic functions (i.e., divide, reciprocal, remainder, modulo) are available as function calls that allow a specification of additional parameters such as rounding modes, overflow style, and guard bits. As rounding style for the `sfixed` we have to choose between the style `fixed_round` and `fixed_truncate`. For the overflow the two styles are `fixed_saturate` and `fixed_wrap`. The default is `fixed_saturate` and `fixed_round`. Minimum HW resources are used if we use `fixed_wrap`, and `fixed_truncate`, or 0 guard bits. The `sfixed` library is designed to minimize the chance of overflow error. This is a departure from most other data types and as a consequence the result of an addition operation needs to have one addition guard bit to the left, i.e.,

```plaintext
SIGNAL a : SFIXED(3 DOWNTO -4) := TO_SFIXED(3.625,3,-4);
SIGNAL b : SFIXED(3 DOWNTO -4);
SIGNAL s : SFIXED(4 DOWNTO -4);
SIGNAL r : SFIXED(3 DOWNTO -4);
...

s <= a + b; -- Coding ok; needs 9 LEs
r <= a + b; -- not allowed
--Error:expression has 9 elements, but must have 8 elements
r <= RESIZE(a + b, r); -- Needs 16 LEs
r <= RESIZE(a + b, r, fixed_wrap,fixed_truncate);
-- Needs 8 LEs
```

We see that the operands `a` and `b` have 4-bit integer (index: 3...0) and 4 fractional bits (index: -1...-4). The value `a = 3.625` would be represented in the `sfixed` format as follows:

```
<table>
<thead>
<tr>
<th>Bit no.</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>-1</th>
<th>-2</th>
<th>-3</th>
<th>-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>0.5</td>
<td>0.25</td>
<td>0.125</td>
<td>0.0625</td>
</tr>
<tr>
<td>Coding</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
```

Note how convenient it is to convert the real numbers into `sfixed` format using `TO_SFIXED()`. This make a code with different coefficients much easier to read.

However, as mentioned, the `sfixed` format is strict in the requirement of additional guard bits such that the above statement of `s <= a + b;` will require nine LEs. The statement `r <= a + b` would not be valid, since `r` is of the same size as `a` and `b`. One additional integer bit is used in the result.
If you want the output in a 4.4 format like the inputs then you can use the resize method

\[ r \leq \text{RESIZE}(a+b, r); \]

where the second parameter specifies the left and right bounds of the result. The synthesis result now comes out even larger due to the fact that by default, the library uses `fixed_saturate` and `fixed_round` options. Three multiplexer are implemented and a total of 16 LEs are needed. To avoid the `saturate` and `rounding` we would code

\[ r \leq \text{RESIZE}(a + b, r, \text{fixed\_wrap}, \text{fixed\_truncate}); \]

and the synthesis result will be the desired eight LEs. Let us now assume the two numbers have range format \( A_L \ldots A_R \) and \( B_L \ldots B_R \) format; then Table 2.9 shows the bit width requirements for the popular `sfixed` operations.

### Table 2.9. Index range of typical `sfixed` operation

<table>
<thead>
<tr>
<th>Operations</th>
<th>Result range</th>
<th>Same range:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A+B, A-B )</td>
<td>( \max(A_L,B_L)+1 \ldots \min(A_R,B_R) )</td>
<td>( A_L=B_L=L; A_R=B_R=R )</td>
</tr>
<tr>
<td>( \text{abs}(A),-A )</td>
<td>( A_L+1 \ldots A_R )</td>
<td>( L+1 \ldots R )</td>
</tr>
<tr>
<td>( A*B )</td>
<td>( A_L+B_L+1 \ldots A_R+B_R )</td>
<td>( 2L+1 \ldots 2R )</td>
</tr>
<tr>
<td>( A/B )</td>
<td>( A_L-B_R+1 \ldots A_R-B_L )</td>
<td>( L-R+1 \ldots R-L )</td>
</tr>
<tr>
<td>reciprocal(A)</td>
<td>( -A_R \ldots -A_L-1 )</td>
<td>( -R \ldots -L-1 )</td>
</tr>
</tbody>
</table>

### 2.7 Floating-Point Arithmetic Implementation

Due to the large gate count capacity of current FPGAs the design of floating-point arithmetic has become a viable option. In addition, the introduction of the embedded 18 × 18-bit array multiplier in Altera Stratix or Cyclone and Xilinx Virtex or Spartan FPGA device families allows an efficient design of custom floating-point arithmetic. We will therefore discuss the design of basic building blocks such as a floating-point adder, subtractor, multiplier, reciprocals and divider, and the necessary conversion blocks to and from fixed-point data format. Such blocks are available from several IP providers, and are now part of the 1076-2008 IEEE VHDL standard. Details can be found in Appendix G of the LRM. Legacy support to 1076-1993 VHDL is possible through an additional fixed- and floating-point package that we will discuss later.

Most of the commercially available floating-point blocks use five or more pipeline stages to increase the throughput. To keep the presentation simple we
will not use pipelining and indeed the VHDL1076-2008 floating-point library does not use any pipelining. Later we will design a 32-bit floating-point unit, but as initial custom floating-point format we will use the (1,6,5) floating-point format introduced in Sect. 2.2.3 (p. 75). This format uses one sign bit, six bits for the exponent and five bits for the mantissa. We support special coding for zero and infinities; support for NaNs and denormals are also enabled as standard in the IEEE 1076-2008 package. Rounding is done via round-to-nearest. By default three guard bits are added to the bottom of every operation. The fixed-point format used in the examples has six integer bits (including a sign bit) and six fractional bits.

2.7.1 Fixed-Point to Floating-Point Format Conversion

As shown in Sect. 2.2.3 (p. 75), floating-point numbers use a signed-magnitude format and the first step is therefore to convert the two’s complement number to signed-magnitude form. If the sign of the fixed-point number is one, we need to compute the complement of the fixed-point number, which becomes the unnormalized mantissa. In the next step we normalize the mantissa and compute the exponent. For the normalization we first determine the number of leading zeros. This can be done with a LOOP statement within a sequential PROCESS in VHDL. Using this number of leading zeros, we shift the mantissa left, until the first 1 “leaves” the mantissa registers, i.e., the hidden one is also removed. This shift operation is actually the task of a barrelshifter, which can be inferred in VHDL via the SLL instruction. Unfortunately the SLL was only defined for BIT_VECTOR data type in the 1076-1993 standard, but not for the STD_LOGIC_VECTOR data type. The newer VHDL 1076-2008 compiler also supports the SLL for STD_LOGIC_VECTOR. However, we can design a barrelshifter in many different ways as Exercise 2.19 (p. 172) shows. Another alternative would be to design a function overloading for the STD_LOGIC_VECTOR that allows a shift operation; see Exercise 1.20, p. 53.

The exponent of our floating-point number is computed as the sum of the bias and the number of integer bits in our fixed-point format minus the leading zeros in the unnormalized mantissa.

Finally, we concatenate the sign, exponent, and the normalized mantissa to a single floating-point word if the fixed-point number is not zero, otherwise we also set the floating-point word to zero.

We have assumed that the range of the floating-point number is larger than the range of the fixed-point number, i.e., the special number $\infty$ will never be used in the conversion.

Figure 2.28 shows the conversion from 12-bit fixed-point data to the (1,6,5) floating-point data for five values $\pm 1$, absolute maximum, absolute minimum, and the smallest value. Rows one to three show the 12-bit fixed-point number and the integer and fractional parts. Rows four to seven show the complete floating-point number, followed by the three parts, sign, exponent, and mantissa.
2.7 Floating-Point Arithmetic Implementation

Fig. 2.28. Simulation results for a (1,5,6) fixed-point format to (1,6,5) floating-point conversion. The five represented values are: +1; −1; maximum $\approx 32$; minimum $= -32$; and smallest $= 1/64$

2.7.2 Floating-Point to Fixed-Point Format Conversion

The floating-point to fixed-point conversion is, in general, more complicated than the conversion in the other direction. Depending on whether the exponent is larger or smaller than the bias we need to implement a left or right shift of the mantissa. In addition, extra consideration is necessary for the special values $\pm \infty$ and $\pm 0$.

To keep the discussion as simple as possible, we assume in the following that the floating-point number has a larger dynamic range than the fixed-point number, but the fixed-point number has a higher precision, i.e., the number of fractional bits of the fixed-point number is larger than the bits used for the mantissa in the floating-point number.

The first step in the conversion is the correction of the bias in the exponent. We then place the hidden 1 to the left and the (fractional) mantissa to the right of the decimal point of the fixed-point word. We then check whether the exponent is too large to be represented with the fixed-point number and then set the fixed-point number to the maximum value. Also, if the exponent is too small, we set the output value to zero. If the exponent is in the valid range where the floating-point number can be represented with the fixed-point format, we shift left the $1.m$ mantissa value (format see (2.23), p. 75) for positive exponents, and shift right for negative exponent values. This, in general, can be coded with the `SLL` and `SRL` in VHDL, respectively, but these `BIT_VECTOR` operations are not supported in VHDL-1993 for `STD_LOGIC_VECTOR`; see Exercise 1.20, p. 53. In the final step we convert the signed magnitude representation to the two’s complement format by evaluating the sign bit of the floating-point number.

Figure 2.29 shows the conversion from (1,6,5) floating-point format to (1,5,6) fixed-point data for the five values $\pm 1$, absolute maximum, absolute minimum, and the smallest value. Rows one to four show the 12-bit floating-point number and the three parts, sign, exponent, and mantissa. Rows five to seven show the whole fixed-point number, followed by the integer and fractional parts. Note that the conversion is without any quantization error for $\pm 1$ and the smallest value. For the absolute maximum and minimum
values, however, the smaller precision in the floating-point numbers gives the imperfect conversion values compared with Fig. 2.28.

2.7.3 Floating-Point Multiplication

In contrast to fixed-point operations, multiplication in floating-point is the simplest of all arithmetic operations and we will discuss this first. In general, the multiplication of two numbers in scientific format is accomplished by multiplication of the mantissas and adding of the exponents, i.e.,

$$f_1 \times f_2 = (a_1 2^{e_1}) \times (a_2 2^{e_2}) = (a_1 \times a_2)2^{e_1+e_2}.$$ 

For our floating-point format with an implicit one and a biased exponent this becomes

$$f_1 \times f_2 = (-1)^{s_1} \left(1.m_1 2^{e_1-\text{bias}}\right) \times (-1)^{s_2} \left(1.m_2 2^{e_2-\text{bias}}\right)$$

$$= (-1)^{s_1+s_2 \mod 2} \left(1.m_1 \times 1.m_2\right) 2^{e_1+e_2-\text{bias}}$$

$$= (-1)^{s_3} 1.m_3 2^{e_3-\text{bias}}.$$ 

We note that the exponent sum needs to be adjusted by the bias, since the bias is included twice in both exponents. The sign of the product is the XOR or modulo-2 sum of the two sign bits of the two operands. We also need to take care of the special values. If one factor is $\infty$ the product should be $\infty$ too. Next, we check whether one factor is zero and set the product to zero if true. Because we do not support NaNs, this implies that $0 \times \infty$ is set to $\infty$. Special values may also be produced from original nonspecial operands. If we detect an overflow, i.e.,

$$e_1 + e_2 - \text{bias} \geq E_{\text{max}},$$

we set the product to $\infty$. Likewise, if we detect an underflow, i.e.,

$$e_1 + e_2 - \text{bias} \leq E_{\text{min}},$$
we set the product to zero. It can be seen that the internal representation of the exponent $e_3$ of the product must have two more bits than the two factors, because we need a sign and a guard bit. Fortunately, the normalization of the product $1.m_3$ is relatively simple; because both operands are in the range $1.0 \leq 1.m_1,2 < 2.0$, the mantissa product is therefore in the range $1.0 \leq 1.m_3 < 4.0$, i.e., a shift by one bit (and exponent adjustment by 1) is sufficient to normalize the product.

Finally, we build the new floating-point number by concatenation of the sign, exponent, and magnitude.

Figure 2.30 shows the multiplication in the (1,6,5) floating-point format of the following values:

1) $(-1) \times (-1) = 1.0_{10} = 1.0000_{2} \times 2^{31-\text{bias}}$

2) $1.75 \times 1.75 = 3.0625_{10} = 11.0001_{2} \times 2^{31-\text{bias}} = 1.10001_{2} \times 2^{32-\text{bias}}$

3) exponent: $7 + 7 - \text{bias} = -17 < E_{\text{min}} \rightarrow \text{underflow in multiplication}$

4) exponent : $62 + 62 - \text{bias} = 93 \geq E_{\text{max}} \rightarrow \text{overflow}$

5) $\infty \times \infty = \infty$

6) $0 \times \infty = \text{NaN}$

Rows one to four show the first floating-point number $a$ and the three parts: sign, exponent, and mantissa. Rows five to eight show the same for the second operand $b$, and rows nine to 12 the product $r$ and the decomposition of the three parts.

### 2.7.4 Floating-Point Addition

Floating-point addition is more complex than multiplication. Two numbers in scientific format

$$f_3 = f_1 + f_2 = (a_12^{e_1}) \pm (a_22^{e_2})$$
can only be added if the exponents are the same, i.e., $e_1 = e_2$. Without loss of generality we assume in the following that the second number has the (absolute) smaller value. If this is not true, we just exchange the first and second numbers. The next step is now to "denormalize" the smaller number by using the following identity:

$$a_22^e = a_2/2^d2^{e_2+d}.$$  

If we select the normalization factor such as $e_2 + d = e_1$, i.e., $d = e_1 - e_2$, we get

$$a_2/2^d2^{e_2+d} = a_2/2^{e_1-e_2}2^{e_1}.$$  

Now both numbers have the same exponent and we can, depending on the signs, add or subtract the first mantissa and the aligned second, according to

$$a_3 = a_1 \pm a_2/2^{e_1-e_2}.$$  

We also need to check whether the second operand is zero. This is the case if $e_2 = 0$ or $d > M$, i.e., the shift operation reduces the second mantissa to zero. If the second operand is zero the first (larger) operand is forwarded to the result $f_3$.

The two aligned mantissas are added if the two floating-point operands have the same sign, otherwise subtracted. The new mantissa needs to be normalized to have the $1.m_3$ format, and the exponent, initially set to $e_3 = e_1$, needs to be adjusted accordingly to the normalization of the mantissa. We need to determine the number of leading zeros including the first one and perform a shift logic left (SLL). We also need to take into account whether one of the operands is a special number, or whether over- or underflow occurs.

If the first operand is $\infty$ or the new computed exponent is larger than $E_{\text{max}}$ the output is set to $\infty$. A NaN results for $\infty - \infty$. If the new computed exponent is smaller than $E_{\text{min}}$, underflow has occurred and the output is set to zero. Finally, we concatenate the sign, exponent, and mantissa to the new floating-point number.

Figure 2.31 shows the addition in the (1,6,5) floating-point format of the following values:

1) $1.0 + (-1.0) = 0$
2) $9.25 + (-10.5) = -1.25_{10} = -1.01000_2 \times 2^{31-\text{bias}}$
3) $1.00111_2 \times 2^{2-\text{bias}} + (-1.00100_2 \times 2^{2-\text{bias}}) = 0.00011_2 \times 2^{2-\text{bias}} = 1.1_2 \times 2^{-2-\text{bias}} \rightarrow -2 < E_{\text{min}} \rightarrow \text{underflow} \rightarrow \text{denormalized number}$
4) $1.01111_2 \times 2^{62-\text{bias}} + 1.11110_2 \times 2^{62-\text{bias}} = 11.01110_2 \times 2^{62-\text{bias}} = 1.1263_{63-\text{bias}} \rightarrow 63 \geq E_{\text{max}} \rightarrow \text{overflow}$
5) $-\infty + 1 = -\infty$
6) $\infty - \infty = \text{NaN}$

Rows one to four show the first floating-point number $a$ and the three parts: sign, exponent, and mantissa. Rows five to eight show the same for the second operand $b$, and rows nine to 12 show the sum $r$ and the decomposition in the three parts, sign, exponent, and mantissa.
2.7 Floating-Point Arithmetic Implementation

2.7.5 Floating-Point Division

In general, the division of two numbers in scientific format is accomplished by division of the mantissas and subtraction of the exponents, i.e.,

$$f_1/f_2 = (a_12^{e_1})/(a_22^{e_2}) = (a_1/a_2)2^{e_1-e_2}.$$ 

For our floating-point format with an implicit one and a biased exponent this becomes

$$f_1/f_2 = (-1)^{s_1} (1.m_12^{e_1-bias}) / (-1)^{s_2} (1.m_22^{e_2-bias})$$

$$= (-1)^{s_1+s_2 \mod 2} \left(1.m_1/1.m_2\right) 2^{(e_1-e_2-bias)+bias}$$

$$= (-1)^{s_3} 1.m_32^{e_3+bias}.$$ 

We note that the exponent sum needs to be adjusted by the bias, since the bias is no longer present after the subtraction of the exponents. The sign of the division is the XOR or modulo-2 sum of the two sign bits of the two operands. The division of the mantissas can be implemented with any algorithm discussed in Sect. 2.5 (p. 93) or we can use the `lpm_divide` component. Because the denominator and quotient has to be at least M+1 bits wide, but numerator and quotient have the same bit width in the `lpm_divide` component, we need to use numerator and quotient with 2 × (M + 1) bits. Because the numerator and denominator are both in the range 1 ≤ 1.m_1,2 < 2, we conclude that the quotient will be in the range 0.5 ≤ 1.m_3 < 2. It follows that a normalization of only one bit (including the exponent adjustment by 1) is required.

We also need to take care of the special values. The result is ∞ if the numerator is ∞, the denominator is zero, or we detect an overflow, i.e.,
Fig. 2.32. Simulation results for division with floating-point numbers in the (1,6,5) format

\[ e_1 - e_2 + \text{bias} = e_3 \geq E_{\text{max}}. \]

Then we check for a zero quotient. The quotient is set to zero if the numerator is zero, denominator is \(\infty\), or we detect an underflow, i.e.,

\[ e_1 - e_2 + \text{bias} = e_3 \leq E_{\text{min}}. \]

In all other cases the result is in the valid range that produces no special result.

Finally, we build the new floating-point number by concatenation of the sign, exponent, and magnitude.

Figure 2.32 shows the division in the (1,6,5) floating-point format of the following values:

1) \((-1)/(−1) = 1.0_{10} = 1.000002 \times 2^{31-\text{bias}}\)
2) \(-10.5/9.25_{10} = -1.135_{10} \approx -1.001_{2} \times 2^{31-\text{bias}}\)
3) \(9.25/(-10.5)_{10} = -0.880952_{10} \approx -1.112_{2} \times 2^{30-\text{bias}}\)
4) exponent: \(60 - 3 + \text{bias} = 88 > E_{\text{max}} \rightarrow \text{overflow in division}\)
5) exponent: \(3 - 60 + \text{bias} = -26 < E_{\text{min}} \rightarrow \text{underflow in division}\)
6) \(1.0/0 = \infty\)
7) \(0/0 = \text{NaN}\)

Rows one to four show the first floating-point number and the three parts: sign, exponent, and mantissa. Rows five to eight show the same for the second operand, and rows nine to 12 show the quotient and the decomposition in the three parts.

### 2.7.6 Floating-Point Reciprocal

Although the reciprocal function of a floating-point number, i.e.,

\[
1.0/f = \frac{1.0}{(-1)^s1.m2^e} = (-1)^s2^{-e}/1.m
\]


seems to be less frequently used than the other arithmetic functions, it is nonetheless useful since it can also be used in combination with the multiplier to build a floating-point divider, because

\[ f_1/f_2 = \frac{1.0}{f_2} \times f_1, \]

i.e., the reciprocal of the denominator followed by multiplication is equivalent to the division.

If the bit width of the mantissa is not too large, we may implement the reciprocal of the mantissa via a look-up table implemented with a `case` statement or with a M9K memory block [34]. Because the mantissa is in the range \(1 \leq 1.m < 2\), the reciprocal must be in the range \(0.5 < \frac{1}{1.m} \leq 1\). The mantissa normalization is therefore a 1-bit shift for all values except \(f = 1.0\).

We also need to take care of the special values. The reciprocal of \(\infty\) is 0, and the reciprocal of 0 is \(\infty\). For all other values the new exponent \(e_2\) is computed with

\[ e_2 = -(e_1 - \text{bias}) + \text{bias} = 2 \times \text{bias} - e_1. \]

Finally, we build the reciprocal floating-point number by the concatenation of the sign, exponent, and magnitude.

Figure 2.33 shows the reciprocal in the (1,6,5) floating-point format of the following values:

1) \(-1/2 = -0.5_{10} = -1.0_{2} \times 2^{30-\text{bias}}\)
2) \(1/1.25_{10} = 0.8_{10} \approx (32 + 19)/64 = 1.10011_{2} \times 2^{30-\text{bias}}\)
3) \(1/1.031 = 0.9697_{10} \approx (32 + 30)/64 = 1.11110_{2} \times 2^{30-\text{bias}}\)
4) \(1.0/0 = \infty\)
5) \(1/\infty = 0.0\)

Rows one to four show the input floating-point number \(a\) and the three parts: sign, exponent, and mantissa. Rows five to eight show the reciprocal \(r\) and the decomposition in the three parts. Note that for the division by zero the transcript window reports: `RECIPROCAL: Floating Point divide by zero`
Implementing floating-point operations in HDL can become a labor intensive
task if we try to build our own complete library including all necessary op-
erations and conversion functions. Luckily, at least for VHDL we have seen
the introduction of a very sophisticated library for operations and functions
that can be used. This is part of the VHDL-2008 standard and a library with
over 7K line of code that is compatible with VHDL-1993 has been provided
by David Bishop and can be downloaded from www.eda.org/fphdl. Since
most vendors support only a subset of standard VHDL language, small mod-
ified versions are available on that webpage that have been tested for Altera,
Xilinx, Synopsys, Cadence, and MentorGraphics (ModelSim) tools. The new
floating-point standard is documented in Appendix G (pp. 537–549) of the
VHDL-2008 LRM and several textbook now cover this new floating-point
data types and the operations too [63–65]. To use the library at a minimum
in VHDL-1993 we would write

```vhdl
LIBRARY ieee_proposed;
USE ieee_proposed.fixed_float_types.ALL;
USE ieee_proposed.float_pkg.ALL;
```

The library allows us to use standard operators like we use for INTEGER
and STD_LOGIC_VECTOR data types:

- Arithmetic: `+`, `-`, `*`, `/`, `ABS`, `REM`, `MOD`
- Logical: `NOT`, `AND`, `NAND`, `OR`, `NOR`, `XOR`, `XNOR`
- Comparison: `=`, `/=`, `>`, `<`, `>=`, `<=`
- Conversion: `TO_SLV`, `TO_SFIXED`, `TO_FLOAT`
- Others: `RESIZE`, `SCALB`, `LOGB`, `MAXIMUM`, `MINIMUM`

There are also a few predefined constant values. The six values are: zero
= `zerofp`, NaN = `nanfp`, quite NaN = `qnanfp`, ∞ = `pos_inffp`, −∞ =
`neg_inffp`, −0 = `neg_zerofp`. Predefined types of length 32, 64, and 128
as in the IEEE standards 854 and 754 are called `FLOAT32`, `FLOAT64`, and
`FLOAT128`, respectively.

Assuming now we like to implement a floating-point number with one
sign, six exponent and five fractional bits as in Example 2.15 (p. 75) we
would define

```vhdl
SIGNAL a, b : FLOAT(6 DOWNTO -5);
SIGNAL s, p : FLOAT(6 DOWNTO -5);
```

and operations can be specified simply as

```vhdl
s <= a + b;
p <= a * b;
```
The code is short since left and right sides use the same data type. No scaling or resizing is required. However, the underlying arithmetic uses the default configuration setting that can be seen from the `fixed_float_types` library file. The rounding style is `round_nearest`, `denormalize` and `error_check` are set to true, and three guard bits are used. The minimum HW effort on the other end will happen if we set rounding to `round_zero` (i.e., truncation), `denormalize` and `error_check` false, and guard bits to 0, so basically the opposite to the default setting. Most operations in VHDL-2008 float type are also available in a function form, e.g., for arithmetic function we can use `ADD`, `SUBTRACT`, `MULTIPLY`, `DIVIDE`, `REMAINDER`, `MODULO`, `RECIPROCAL`, `MAC`, and `SQRT`. Then it is much easier to modify the rounding style and guard bits:

```vhdl
r <= ADD(l=>a, r=> b, -- Should be the "cheapest" design
    round_style => round_zero,
    guard => 0,
    check_error => false,
    denormalize => false);
```

The left and right operands are specified first, followed by the four synthesis parameters that should give the smallest number of LEs. Note that the IEEE VHDL-2008-1076 (p. 540) says “guard_bits” not “guard” as has been used in the library written by David Bishop.

Comparison can also be used as a function call (similar names as in FORTRAN) via `EQ`, `NE`, `GT`, `LT`, `GE`, and `LE`. For scaling the function `SCALB(y,n)` implements the operation $y \times 2^n$ with a reduced HW effort compared to normal multiplication or divide. `MAXIMUM`, `MINIMUM`, square root `SQRT`, and multiply-and-add `MAC` are additional functions that can be useful in DSP.

Now let us look at how these functions work in a small example. Since most simulators so far not fully support the new data types such as a negative index in arrays, it seems to be a good approach that we stay with the standard `STD_LOGIC_VECTOR` as I/O type. The library provides so-called bit-preserving conversion functions that just redefine the meaning of the bits in the `STD_LOGIC` vector to an `sfixed` or `float` type of the same length. Such a conversion is done by the VHDL preprocessor and should not consume any hardware resources. On the other hand we also need a `value preserving` operation if we do a conversion between the `sfixed` and `float` types. This conversion will preserve the value of our data, but that will indeed require substantiable hardware resources. Now let use have a look at the 32-bit floating-point unit (FPU) that performs some basic operations as well as data conversion.

**Example 2.22: A 32-bit Floating-Point Arithmetic Unit**
The VHDL design file `fpu.vhd` is shown below:

```vhdl
LIBRARY ieee; USE ieee.std_logic_1164.ALL;
PACKAGE n_bit_int IS -- User defined types
  SUBTYPE SLV4 IS STD_LOGIC_VECTOR(3 DOWNTO 0);
  SUBTYPE SLV32 IS STD_LOGIC_VECTOR(31 DOWNTO 0);
END n_bit_int;
LIBRARY work; USE work.n_bit_int.ALL;
LIBRARY ieee;
USE ieee.std_logic_1164.ALL; USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;
LIBRARY ieee_proposed;
USE ieee_proposed.fixed_float_types.ALL;
USE ieee_proposed.fixed_pkg.ALL;
USE ieee_proposed.float_pkg.ALL;
-- --------------------------------------------------------
ENTITY fpu IS
  PORT(sel : IN SLV4; -- FP operation number
dataa : IN SLV32; -- First input
datab : IN SLV32; -- Second input
n : IN INTEGER; -- Scale factor 2**n
result : OUT SLV32); -- System output
END;
-- --------------------------------------------------------
ARCHITECTURE fpga OF fpu IS
```

The equivalent Verilog code for this example cannot be composed since the Verilog language currently does not have floating-point library support.
-- OP Code of instructions:
-- CONSTANT fix2fp : STD_LOGIC_VECTOR(3 DOWNTO 0) := X"0":
-- CONSTANT fp2fix : STD_LOGIC_VECTOR(3 DOWNTO 0) := X"1":
-- CONSTANT add : STD_LOGIC_VECTOR(3 DOWNTO 0) := X"2":
-- CONSTANT sub : STD_LOGIC_VECTOR(3 DOWNTO 0) := X"3":
-- CONSTANT mul : STD_LOGIC_VECTOR(3 DOWNTO 0) := X"4":
-- CONSTANT div : STD_LOGIC_VECTOR(3 DOWNTO 0) := X"5":
-- CONSTANT rec : STD_LOGIC_VECTOR(3 DOWNTO 0) := X"6":
-- CONSTANT scale : STD_LOGIC_VECTOR(3 DOWNTO 0) := X"7":

TYPE OP_TYPE IS (fix2fp, fp2fix, add, sub, mul, div, rec, scale);

SIGNAL op : OP_TYPE;
SIGNAL a, b, r : FLOAT32;
SIGNAL sfixeda, sfixedr : SFIXED(15 DOWNTO -16);
BEGIN

-- Redefine SLV bit as FP number
a <= TO_FLOAT(dataa, a);
b <= TO_FLOAT(datab, b);
-- Redefine SLV bit as 16.16 sfixed number
sfixeda <= TO_SFIXED(dataa, sfixeda);

P1: PROCESS (a, b, sfixedr, sfixeda, sel, r, n, op)
BEGIN
    r <= (OTHERS => '0'); sfixedr <= (OTHERS => '0');
    CASE CONV_INTEGER(sel) IS
        WHEN 0 => r <= TO_FLOAT(sfixeda, r); op <= fix2fp;
        WHEN 1 => sfixedr <= TO_SFIXED(a, sfixedr);
                   op <= fp2fix;
        WHEN 2 => r <= a + b; op <= add;
        WHEN 3 => r <= a - b; op <= sub;
        WHEN 4 => r <= a * b; op <= mul;
        WHEN 5 => r <= a / b; op <= div;
        WHEN 6 => r <= reciprocal(arg=> a); op <= rec;
        WHEN 7 => r <= scalb(y=>a, n=>n); op <= scale;
        WHEN OTHERS => op <= scale;
    END CASE;
    -- Interpret FP or 16.16 sfixed bits as SLV bit vector
    IF op = fp2fix THEN
        result <= TO_SLV(sfixedr);
    ELSE
        result <= TO_SLV(r);
    END IF;
END PROCESS P1;

END fpga;

First the necessary libraries are called up. Note that the float and fixed packages need to be downloaded first from www.eda.org/fphdl if your tool does not support the VHDL-2008 types. The ENTITY then includes the selection of our operation, the two input vectors, the scale factor n, and the result. Data in 32-bit float and 16.16 sfixed are then defined. The signal
op is used to display the current operation in plain text in the simulation. The actual floating-point arithmetic unit is placed in the PROCESS environment P1. The first operation implemented is an sfixed to FLOAT32 conversion followed by FLOAT32 to sfixed. Then come the basic four arithmetic operations: +, -, *, / . The operation six is the reciprocal that has only one input. Number seven is the scale operation that implements power-of-two multiply and divide. The last IF statement is used to make a bit preserving conversion (a.k.a. redefinition of the bits) to the SLV type for I/O and display. Only for selection one do we have a sfixed type; for all the others the output is of type FLOAT32.

The design uses 8112 LEs and seven embedded multipliers. A registered performance cannot be measured since registers are not used. To simulate we first use a small test program in C or Matlab to compute some test bench data. In Matlab we can use the format \%tX and \%bX to display 32- and 64-bit floats as hex, respectively. For instance if we set \( x = \frac{1}{3} \) and then type

\[
\text{str} = \text{sprintf(}' \text{FLOAT32 := X"\%tX"; -- \%f',x,x);disp(str)
\]

in the Matlab prompt window the tool will produce

\[
\text{FLOAT32 := X"3EAAAAAB"; -- 0.333333}
\]

The book CD includes a small program called fp_ops.exe that computes test data for 32 and 64 float basic arithmetic operations. If we enter inputs 1/3 and 2/3 as a and b we will get the listing shown in Fig. 2.34 that provides test data for add, subtract, multiply, divide, and reciprocal. To test the input conversion we use the decimal value one that is 0001.0000 in sfixed and 3F800000 in hex code for FLOAT32. For the scale operation we use a scale by \( 2^1 \), i.e., \( \frac{1}{3} \times 2 = \frac{2}{3} \). The overall simulation is shown in Fig. 2.35.

\[ \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
\text{op} & \text{func} & \text{hitmsg} & \text{add} & \text{sub} & \text{mul} & \text{div} & \text{rec} & \text{scale} \\
\hline
\text{select} & 0 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\text{data} & 0.010000 & 0.010000 & 0.010000 & \text{AABBAA} \\
\text{data} & 0.010000 & 0.010000 & \text{AABBAA} \\
\text{output} & 0 & 0 & 0 & 0 \\
\text{result} & 3F800000 & 3F800000 & 3F800000 & \text{AABBAA} \\
\hline
\end{array} \]

\[ \begin{array}{|c|c|c|c|c|c|}
\hline
\text{instr} & \text{delay} \\
\hline
\text{hitmsg} & 100 ms \\
\text{add} & 200 ms \\
\text{sub} & 400 ms \\
\text{mul} & 600 ms \\
\text{div} & 800 ms \\
\hline
\end{array} \]

\[ \begin{array}{|c|c|c|c|c|c|}
\hline
\text{result} & \text{delay} \\
\hline
\text{hitmsg} & 100 ms \\
\text{add} & 200 ms \\
\text{sub} & 400 ms \\
\text{mul} & 600 ms \\
\text{div} & 800 ms \\
\hline
\end{array} \]

\[ \text{Cursor: 0.00 ns} \]

\[ \text{Fig. 2.35. Simulation of eight functions of the floating-point arithmetic unit fpu} \]

\[ \text{2.2.2} \]

2.7.8 Floating-Point Synthesis Results

The VHDL-2008 library allows us to write efficient compact code for any float specified. The only disadvantage is that the overall speed of such a design will not be very high since substantial arithmetic is used, but no pipelining is implemented. FPGA vendor usually provide predefined floating-point blocks in 32 and 64 bits that are highly pipelined. Xilinx offers the
Table 2.10. Pipelining in the Altera LPM block library for 32-bit floating-point data

<table>
<thead>
<tr>
<th>Block</th>
<th>Pipeline range</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>int/fix to fp</td>
<td>6, 6</td>
<td></td>
</tr>
<tr>
<td>fp to int/ fix</td>
<td>6, 6</td>
<td></td>
</tr>
<tr>
<td>add/sub</td>
<td>7...14, 11</td>
<td>11</td>
</tr>
<tr>
<td>multiply</td>
<td>5,6,10,11</td>
<td>5</td>
</tr>
<tr>
<td>divide</td>
<td>6,14,33</td>
<td>33</td>
</tr>
<tr>
<td>inv</td>
<td>20, 20</td>
<td></td>
</tr>
</tbody>
</table>

LOGiCORE floating-point IP and Altera has a full set of LPM functions. The LPM blocks can be used as graphical block or instantiated from the component library that can be found under `quartus → libraries → vhdl → altera-mf_components.vhd`.

Let us take a brief look at the pipeline requirements to achieve high throughput. Table 2.10 lists the available range and the default setting for Altera’s LPM [20]. Figure 2.36 shows a comparison of the throughput that can be achieved when using the VHDL-2008 library and the LPM blocks. The benefit of the pipelining in the LPM blocks is clearly visible. However, keep in mind that many systems we will discuss have feedback, and pipelining is then usually not possible. Then the LPM blocks cannot be used, unless we use an FSM and “wait” until the computation is completed. In order to measure the registered performance $F_{\text{max}}$ with the VHDL-2008 library, registers were added to the input and output ports, but no pipelining inside the block has been used.

Figure 2.37 shows the synthesis results for all eight basic building blocks in 32 and 64 bit width. The solid line shows the required LEs and multipliers with default VHDL-2008 settings, while the dashed lines show the results when synthesis options are set to minimum HW effort, i.e., rounding to `round_zero` (i.e., truncation), `denormalize` and `error_check` false, and guard bits to 0. Conversion between `fixed` and `FLOAT32` requires about 400 LEs. We see that basic math operations $+,-,\ast$ require about 1K LEs with standard setting and twice that for the divide. Note that division and reciprocal in the LPM functions use a substantial number of multipliers and one M9K memory block (not shown in the plot), while the equivalent VHDL-2008 operations do not use any memory or multiplier. The `SCALB` to multiply or divide by power-of-two is substantially cheaper than the equivalent multiply or divide.
2.8 Multiply-Accumulator (MAC) and Sum of Product (SOP)

DSP algorithms are known to be multiply-accumulate (MAC) intensive. To illustrate, consider the linear convolution sum given by

\[ y[n] = f[n] \times x[n] = \sum_{k=0}^{L-1} f[k] x[n - k] \]  

requiring \( L \) consecutive multiplications and \( L - 1 \) addition operations per sample \( y[n] \) to compute the sum of products (SOPs). This suggests that an \( N \times N \)-bit multiplier needs to be fused together with an accumulator; see Fig. 2.38a. A full-precision \( N \times N \)-bit product is \( 2N \) bits wide. If both operands are (symmetric) signed numbers, the product will only have \( 2N - 1 \) significant bits, i.e., two sign bits. The accumulator, in order to maintain sufficient dynamic range, is often designed to be an extra \( K \) bits in width, as demonstrated in the following example.

**Example 2.23:** The Analog Devices PDSP family ADSP21xx contains a \( 16 \times 16 \) array multiplier and an accumulator with an extra 8 bits (for a total accumulator width of \( 32 + 8 = 40 \) bits). With this eight extra bits, at least \( 2^8 \) accumulations are
possible without sacrificing the output. If both operands are symmetric signed, $2^9$ accumulation can be performed. In order to produce the desired output format, such modern PDSPs also include a barrelshifter, which allows the desired adjustment within one clock cycle.

This overflow consideration in fixed-point PDSP is important to mainstream digital signal processing, which requires that DSP objects be computed in real time without unexpected interruptions. Recall that checking and servicing accumulator overflow interrupts the data flow and carries a significant temporal liability. By choosing the number of guard bits correctly, the liability can be eliminated.

An alternative approach to the MAC of a conventional PDSP for computing an SOP will be discussed in the next section.

2.8.1 Distributed Arithmetic Fundamentals

_Distributed arithmetic_ (DA) is an important FPGA technology. It is extensively used in computing the sum of products:
\[ y = \langle c, x \rangle = \sum_{n=0}^{N-1} c[n] \times x[n]. \quad (2.46) \]

Besides convolution, correlation, DFT computation, and the RNS inverse mapping discussed earlier can also be formulated as such SOPs. Completing a filter cycle, when using a conventional arithmetic unit, would take approximately \( N \) MAC cycles. This can be shortened with pipelining but can, nevertheless, be prohibitively long. This is a fundamental problem when general-purpose multipliers are used.

In many DSP applications, a general-purpose multiplication is technically not required. If the filter coefficients \( c[n] \) are known a priori, then technically the partial product term \( c[n]x[n] \) becomes a multiplication with a constant (i.e., scaling). This is an important difference and is a prerequisite for a DA design.
The first discussion of DA can be traced to a 1973 paper by Croisier [66] and DA was popularized by Peled and Liu [67]. Yiu [68] extended DA to signed numbers, and Kammeyer [69] and Taylor [70] studied quantization effects in DA systems. DA tutorials are available from White [71] and Kammeyer [72]. DA also is addressed in textbooks [73,74]. To understand the DA design paradigm, consider the “sum of products” inner product shown below:

\[ y = \langle \mathbf{c}, \mathbf{x} \rangle = \sum_{n=0}^{N-1} c[n] \times x[n] \]

\[ = c[0]x[0] + c[1]x[1] + \ldots + c[N-1]x[N-1]. \]  

(2.47)

Assume further that the coefficients \( c[n] \) are known constants and \( x[n] \) is a variable. An unsigned DA system assumes that the variable \( x[n] \) is represented by:

\[ x[n] = \sum_{b=0}^{B-1} x_b[n] \times 2^b \quad \text{with} \quad x_b[n] \in [0, 1], \]  

(2.48)

where \( x_b[n] \) denotes the \( b \)th bit of \( x[n] \), i.e., the \( n \)th sample of \( \mathbf{x} \). The inner product \( y \) can, therefore, be represented as:

\[ y = \sum_{n=0}^{N-1} c[n] \times \sum_{b=0}^{B-1} x_b[n] \times 2^b. \]  

(2.49)

Redistributing the order of summation (thus the name “distributed arithmetic”) results in:

\[ y = c[0] \left( x_{B-1}[0]2^{B-1} + x_{B-2}[0]2^{B-2} + \ldots + x_0[0]2^0 \right) \]

\[ + c[1] \left( x_{B-1}[1]2^{B-1} + x_{B-2}[1]2^{B-2} + \ldots + x_0[1]2^0 \right) \]

\[ \vdots \]

\[ + c[N-1] \left( x_{B-1}[N-1]2^{B-1} + \ldots + x_0[N-1]2^0 \right) \]

\[ = (c[0]x_{B-1}[0] + c[1]x_{B-1}[1] + \ldots + c[N-1]x_{B-1}[N-1]) \times 2^{B-1} \]

\[ + (c[0]x_{B-2}[0] + c[1]x_{B-2}[1] + \ldots + c[N-1]x_{B-2}[N-1]) \times 2^{B-2} \]

\[ \vdots \]

\[ + (c[0]x_0[0] + c[1]x_0[1] + \ldots + c[N-1]x_0[N-1]) \times 2^0, \]

or in more compact form
\[
y = \sum_{b=0}^{B-1} 2^b \times \sum_{n=0}^{N-1} c[n] \times x_b[n] = \sum_{b=0}^{B-1} 2^b \times \sum_{n=0}^{N-1} f(c[n], x_b[n]). \tag{2.50}
\]

Implementation of the function \( f(c[n], x_b[n]) \) requires special attention. The preferred implementation method is to realize the mapping \( f(c[n], x_b[n]) \) using one LUT. That is, a \( 2^N \)-word LUT is preprogrammed to accept an \( N \)-bit input vector \( x_b = [x_b[0], x_b[1], \ldots, x_b[N-1]] \), and output \( f(c[n], x_b[n]) \). The individual mappings \( f(c[n], x_b[n]) \) are weighted by the appropriate power-of-two factor and accumulated. The accumulation can be efficiently implemented using a shift-adder as shown in Fig. 2.38b. After \( N \) look-up cycles, the inner product \( y \) is computed.

**Example 2.24: Unsigned DA Convolution**

A third order inner product is defined by the inner product equation

\[
\langle c, x \rangle = \sum_{n=0}^{2} c[n] x[n].
\]

Assume that the 3-bit coefficients have the values \( c[0] = 2, c[1] = 3 \), and \( c[2] = 1 \). The resulting LUT, which implements \( f(c[n], x_b[n]) \), is defined below:

<table>
<thead>
<tr>
<th>( x_b[2] )</th>
<th>( x_b[1] )</th>
<th>( x_b[0] )</th>
<th>( f(c[n], x_b[n]) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( 1 \times 0 + 3 \times 0 + 2 \times 0 = 0_{10} = 000_2 )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>( 1 \times 0 + 3 \times 0 + 2 \times 1 = 2_{10} = 010_2 )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>( 1 \times 0 + 3 \times 1 + 2 \times 0 = 3_{10} = 011_2 )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>( 1 \times 0 + 3 \times 1 + 2 \times 1 = 5_{10} = 101_2 )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>( 1 \times 1 + 3 \times 0 + 2 \times 0 = 1_{10} = 001_2 )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>( 1 \times 1 + 3 \times 0 + 2 \times 1 = 3_{10} = 011_2 )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>( 1 \times 1 + 3 \times 1 + 2 \times 0 = 4_{10} = 100_2 )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>( 1 \times 1 + 3 \times 1 + 2 \times 1 = 6_{10} = 110_2 )</td>
</tr>
</tbody>
</table>

The inner product, with respect to \( x[n] = \{x[0] = 1_{10} = 001_2, x[1] = 3_{10} = 011_2, x[2] = 7_{10} = 111_2 \} \), is obtained as follows:

<table>
<thead>
<tr>
<th>Step ( t )</th>
<th>( x_t[2] )</th>
<th>( x_t[1] )</th>
<th>( x_t[0] )</th>
<th>( f[t] )</th>
<th>( +ACC[t-1] )</th>
<th>( =ACC[t] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>( 6 \times 2^0 + )</td>
<td>0</td>
<td>= 6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>( 4 \times 2^1 + )</td>
<td>6</td>
<td>= 14</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>( 1 \times 2^2 + )</td>
<td>14</td>
<td>= 18</td>
</tr>
</tbody>
</table>

As a numerical check, note that

\[
= 2 \times 1 + 3 \times 3 + 1 \times 7 = 18. \checkmark
\]

For a hardware implementation, instead of shifting each intermediate value by \( b \) (which will demand an expensive barrelshifter) it is more appropriate to shift the accumulator content itself in each iteration one bit to the right. It is easy to verify that this will give the same results.
The bandwidth of an $N$th order $B$-bit linear convolution, using general-purpose MACs and DA hardware, can be compared. Figure 2.38 shows the architectures of a conventional PDSP and the same realization using distributed arithmetic.

Assume that a LUT and a general-purpose multiplier have the same delay $\tau = \tau(\text{LUT}) = \tau(\text{MUL})$. The computational latencies are then $B\tau(\text{LUT})$ for DA and $N\tau(\text{MUL})$ for the PDSP. In the case of small bit width $B$, the speed of the DA design can therefore be significantly faster than a MAC-based design. In Chap. 3, comparisons will be made for specific filter design examples.

### 2.8.2 Signed DA Systems

In the following, we wish to discuss how (2.47) should be modified, in order to process a signed two’s complement number. In two’s complement, the MSB is used to distinguish between positive and negative numbers. For instance, from Table 2.1 (p. 59) we see that decimal $-3$ is coded as $101_2 = -4 + 0 + 1 = -3_{10}$. We use, therefore, the following $(B + 1)$-bit representation

$$x[n] = -2^B \times x_B[n] + \sum_{b=0}^{B-1} x_b[n] \times 2^b.$$  \hspace{1cm} (2.51)

Combining this with (2.49), the outcome $y$ is defined by:

$$y = -2^B \times f(c[n], x_B[n]) + \sum_{b=0}^{B-1} 2^b \times \sum_{n=0}^{N-1} f(c[n], x_b[n]).$$  \hspace{1cm} (2.52)

To achieve the signed DA system we therefore have two choices to modify the unsigned DA system. They are

- An accumulator with add/subtract control
- Using a ROM with one additional input

Most often the switchable accumulator is preferred, because the additional input bit in the table requires a table with twice as many words. The following example demonstrates the processing steps for the add/sub switch design.

**Example 2.25: Signed DA Inner Product**

Consider again a third order inner product defined by the convolution sum

$$y = \langle c, x \rangle = \sum_{n=0}^{2} c[n]x[n].$$

Assume that the data $x[n]$ is given in 4-bit two’s complement encoding and that the coefficients are $c[0] = -2$, $c[1] = 3$, and $c[2] = 1$. The corresponding LUT table is given below:
The values of $x[k]$ are $x[0] = 1_{10} = 001_2C$, $x[1] = -3_{10} = 1101_2C$, and $x[2] = 7_{10} = 0111_2C$. The output at sample index $k$, namely $y$, is defined as follows:

<table>
<thead>
<tr>
<th>Step $t$</th>
<th>$x_t[2]$</th>
<th>$x_t[1]$</th>
<th>$x_t[0]$</th>
<th>$f[t] \times 2^t + Y[t - 1] = Y[t]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$2 \times 2^0 + 0 = 2$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$1 \times 2^1 + 2 = 4$</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$4 \times 2^2 + 4 = 20$</td>
</tr>
</tbody>
</table>

A numerical check results in $c[0]x[0] + c[1]x[1] + c[2]x[2] = -2 \times 1 + 3 \times (-3) + 1 \times 7 = -4 \sqrt{2.25}$

### 2.8.3 Modified DA Solutions

In the following we wish to discuss two interesting modifications to the basic DA concept, where the first variation reduces the size, and the second increases the speed.

If the number of coefficients $N$ is too large to implement the full word with a single LUT (recall that input LUT bit width = number of coefficients), then we can use partial tables and add the results. If we also add pipeline registers, this modification will not reduce the speed, but can dramatically reduce the size of the design, because the size of a LUT grows exponentially with the address space, i.e., the number of input coefficients $N$. Suppose the length $LN$ inner product

$$y = \langle c, x \rangle = \sum_{n=0}^{LN-1} c[n]x[n]$$

is to be implemented using a DA architecture. The sum can be partitioned into $L$ independent $N$th parallel DA LUTs resulting in

$$y = \langle c, x \rangle = \sum_{l=0}^{L-1} \sum_{n=0}^{N-1} c[0]n + c[1]n + c[2]n \cdots + c[L-1]n + n$$

$$= \sum_{l=0}^{L-1} \sum_{n=0}^{N-1} c[0]n + c[1]n + c[2]n \cdots + c[L-1]n + n.$$

$$= \sum_{l=0}^{L-1} \sum_{n=0}^{N-1} c[0]n + c[1]n + c[2]n \cdots + c[L-1]n + n.$$
If a digital signal processing algorithm is implemented with FPGAs and the algorithm uses a nontrivial (transcendental) algebraic function, like $\sqrt{x}$ or
arctan \( y/x \), we can always use the Taylor series to approximate this function, i.e.,

\[
f(x) = \sum_{k=0}^{K} \frac{f^k(x_0)}{k!} (x - x_0)^k,
\]

where \( f^k(x) \) is the \( k^{th} \) derivative of \( f(x) \) and \( k! = k \times (k - 1) \ldots \times 1 \). The problem is then reduced to a sequence of multiply and add operations. A more efficient, alternative approach, based on the Coordinate Rotation Digital Computer (CORDIC) algorithm can also be considered. The CORDIC algorithm is found in numerous applications, such as pocket calculators [75], and in mainstream DSP objects, such as adaptive filters, FFTs, DCTs [76], demodulators [77], and neural networks [45]. The basic CORDIC algorithm can be found in two classic papers by Volder [78] and Walther [79]. Some theoretical extensions have been made, such as the extension of range in the hyperbolic mode, or the quantization error analysis by Hu et al. [80], and Meyer-Bäse et al. [77]. VLSI implementations have been discussed in Ph.D. theses, such as those by Timmermann [81] and Hahn [82]. The first FPGA
implementations were investigated by Meyer-Bäse et al. [4, 77]. The realization of the CORDIC algorithm in distributed arithmetic was investigated by Ma [83]. A very detailed overview including details of several applications, was provided by Hu [76] in a 1992 IEEE Signal Processing Magazine review paper.

The original CORDIC algorithm by Volder [78] computes a multiplier-free coordinate conversion between rectangular \((x, y)\) and polar \((R, \theta)\) coordinates. Walther [79] generalized the CORDIC algorithm to include circular \((m = 1)\), linear \((m = 0)\), and hyperbolic \((m = -1)\) transforms. For each mode, two rotation directions are identified. For vectoring, a vector with starting coordinates \((X_0, Y_0)\) is rotated in such a way that the vector finally lies on the abscissa (i.e., \(x\) axis) by iteratively converging \(Y_K\) to zero. For rotation, a vector with a starting coordinate \((X_0, Y_0)\) is rotated by an angle \(\theta_0\) in such a way that the final value of the angle register, denoted \(Z\), converges to zero. The angle \(\theta_k\) is chosen so that each iteration can be performed with an addition and a binary shift. Table 2.11 shows, in the second column, the choice for the rotation angle for the three modes \(m = 1, 0,\) and \(-1\).

Now we can formally define the CORDIC algorithm as follows:

**Algorithm 2.26: CORDIC Algorithm**

At each iteration, the CORDIC algorithm implements the mapping:

\[
\begin{bmatrix}
X_{k+1} \\
Y_{k+1}
\end{bmatrix}
= 
\begin{bmatrix}
1 & m\delta_k 2^{-k} \\
\delta_k 2^{-k} & 1
\end{bmatrix}
\begin{bmatrix}
X_k \\
Y_k
\end{bmatrix}
\]

\(Z_{k+1} = Z_k + \delta_k \theta_k,\)

where the angle \(\theta_k\) is given in Table 2.11, \(\delta_k = \pm 1\), and the two rotation directions are \(Z_k \to 0\) and \(Y_K \to 0\).

This means that six operational modes exist, and they are summarized in Table 2.12. A consequence is that nearly all transcendental functions can be computed with the CORDIC algorithm. With a proper choice of the initial values, the function \(X \times Y, Y/X, \sin(Z), \cos(Z), \tan^{-1}(Y), \sinh(Z), \cosh(Z),\) and \(\tanh(Z)\) can directly be computed. Additional functions may be generated by choosing appropriate initialization, sometimes combined with multiple modes of operation, as shown in the following listing:

**Table 2.11. CORDIC algorithm modes**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Angle (\theta_k)</th>
<th>Shift sequence</th>
<th>Radius factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>circular (m = 1)</td>
<td>(\tan^{-1}(2^{-k}))</td>
<td>0, 1, 2, \ldots</td>
<td>(K_1 = 1.65)</td>
</tr>
<tr>
<td>linear (m = 0)</td>
<td>(2^{-k})</td>
<td>1, 2, \ldots</td>
<td>(K_0 = 1.0)</td>
</tr>
<tr>
<td>hyperbolic (m = -1)</td>
<td>(\tanh^{-1}(2^{-k}))</td>
<td>1, 2, 3, 4, \ldots</td>
<td>(K_{-1} = 0.80)</td>
</tr>
</tbody>
</table>

\[2.56\]
2. Computer Arithmetic

Fig. 2.41. CORDIC. (a) Modes. (b) Example of circular vectoring

\[
\tan(Z) = \frac{\sin(Z)}{\cos(Z)} \quad \text{Modes: } m=1, 0
\]
\[
\tanh(Z) = \frac{\sinh(Z)}{\cosh(Z)} \quad \text{Modes: } m=-1, 0
\]
\[
\exp(Z) = \sinh(Z) + \cosh(Z) \quad \text{Modes: } m=-1; \quad x = y = 1
\]
\[
\log_e(W) = 2 \tanh^{-1}(Y/X) \quad \text{Modes: } m=-1
\]
\[
\sqrt{W} = \sqrt{X^2 - Y^2} \quad \text{Modes: } m=-1
\]

A careful analysis of (2.56) reveals that the iteration vectors only approach the curves shown in Fig. 2.41a. The length of the vectors changes with each iteration, as shown in Fig. 2.41b. This change in length does not depend on the starting angle and after \( K \) iterations the same change (called radius factor) always occurs. In the last column of Table 2.11 these radius factors are shown. To ensure that the CORDIC algorithm converges, the sum of all remaining rotation angles must be larger than the actual rotation angle. This is the case for linear and circular transforms. For the hyperbolic mode, all iterations of the form \( n_{k+1} = 3n_k + 1 \) have to be repeated. These are the iterations 4, 13, 40, 121 . . .

Table 2.12. Modes \( m \) of operation for the CORDIC algorithm

<table>
<thead>
<tr>
<th>( m )</th>
<th>( Z_K \rightarrow 0 )</th>
<th>( Y_K \rightarrow 0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( X_K = K_1(X_0 \cos(Z_0) - Y_0 \sin(Z_0)) )</td>
<td>( X_K = K_1 \sqrt{X_0^2 + Y_0^2} )</td>
</tr>
<tr>
<td>( Y_K = K_1(Y_0 \cos(Z_0) + X_0 \sin(Z_0)) )</td>
<td>( Z_K = Z_0 + \arctan(Y_0/X_0) )</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>( X_K = X_0 )</td>
<td>( X_K = X_0 )</td>
</tr>
<tr>
<td>( Y_K = Y_0 + X_0 \times Z_0 )</td>
<td>( Z_K = Z_0 + Y_0/X_0 )</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>( X_K = K_{-1}(X_0 \cosh(Z_0) - Y_0 \sinh(Z_0)) )</td>
<td>( X_K = K_{-1} \sqrt{X_0^2 - Y_0^2} )</td>
</tr>
<tr>
<td>( Y_K = K_{-1}(Y_0 \cosh(Z_0) + X_0 \sinh(Z_0)) )</td>
<td>( Z_K = Z_0 + \tanh^{-1}(Y_0/X_0) )</td>
<td></td>
</tr>
</tbody>
</table>
Output precision can be estimated using a procedure developed by Hu [84] and illustrated in Fig. 2.42. The graph shows the effective bit precision for the circular mode, depending on the $X,Y$ path width, and the number of iterations. If $b$ bits is the desired output precision, the “rule of thumb” suggests that the $X,Y$ path should have $\log_2(b)$ additional guard bits. From Fig. 2.43, it can also be seen that the bit width of the $Z$ path should have the same precision as that for $X$ and $Y$.

In contrast to the circular CORDIC algorithm, the effective resolution of a hyperbolic CORDIC cannot be computed analytically because the precision depends on the angular values of $z(k)$ at iteration $k$. Hyperbolic precision can, however, be estimated using simulation. Figure 2.44 shows the minimum accuracy estimate computed over 1000 test values for each bit-width/number combination of the possible iterations. The 3D representation shows the number of iterations, the bit width of the $X/Y$ path, and the resulting minimum precision of the result in terms of effective bits. The contour lines allow an exchange between the number of iterations and the bit width. For example, to achieve 10-bit precision, one can use a 21-bit $X/Y$ path and 18 iterations, or 14 iterations at 24 bits.

2.9.1 CORDIC Architectures

Two basic structures are used to implement a CORDIC architecture: the more compact state machine or the high-speed, fully pipelined processor.

If computation time is not critical, then a state machine as shown in Fig. 2.45 is applicable. In each cycle, exactly one iteration of (2.56) will be computed. The most complex part of this design is the two barrelshifters. The

![Fig. 2.42. Effective bits in circular mode](image-url)
two barrelshifters can be replaced by a single barrelshifter, using a multiplexer as shown in Fig. 2.46, or a serial (right, or right/left) shifter. Table 2.13 compares different design options for a 13-bit implementation using Xilinx XC3K FPGAs.

If high speed is needed, a fully pipelined version of the design shown in Fig. 2.47 can be used. Figure 2.47 shows eight iterations of a circular
2.9 Computation of Special Functions Using CORDIC

CORDIC. After an initial delay of $K$ cycles, a new output value becomes available after each cycle. As with array multipliers, CORDIC implementations have a quadratic growth in LE complexity as the bit width increases (see Fig. 2.47).

The following example shows the first four steps of a circular-vectoring fully pipelined design.

**Example 2.27: Circular CORDIC in Vectoring Mode**

The first iteration rotates the vectors from the second or third quadrant to the first or fourth, respectively. The shift sequence is 0,0,1, and 2. The rotation angle of the first four steps becomes: \( \arctan(\infty) = 90^\circ \), \( \arctan(2^0) = \).
Fig. 2.46. CORDIC machine with reduced complexity

45°, \( \arctan(2^{-1}) = 26.5° \), and \( \arctan(2^{-2}) = 14° \). The VHDL code\(^9\) for 8-bit data can be implemented as follows:

```vhdl
PACKAGE n_bit_int IS -- User defined types
    SUBTYPE S8 IS INTEGER RANGE -128 TO 127;
    SUBTYPE S9 IS INTEGER RANGE -256 TO 256;
    TYPE A0_3S9 IS ARRAY (0 TO 3) OF S9;
END n_bit_int;
```

```vhdl
LIBRARY work;
USE work.n_bit_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
```

```vhdl
-- --------------------------------------------------------
ENTITY cordic IS ------> Interface
    PORT (clk : IN STD_LOGIC; -- System clock
           reset : IN STD_LOGIC; -- Asynchronous reset
           x_in : IN S8; -- System real or x input
           y_in : IN S8; -- System imaginary or y input
           r   : OUT S9; -- Radius result
           phi : OUT S9; -- Phase result
           eps : OUT S9); -- Error of results
END cordic;
```

```
ARCHITECTURE fpga OF cordic IS
```

\(^9\) The equivalent Verilog code `cordic.v` for this example can be found in Appendix A on page 804. Synthesis results are shown in Appendix B on page 881.
2.9 Computation of Special Functions Using CORDIC

\[
X \quad Y \\
\begin{align*}
X_1 & = Y_1 \\
X_2 & = \pm Y_1 \\
X_3 & = X_2 \pm 2^0 Y_2 \\
X_4 & = X_3 \pm 2^{-1} Y_3 \\
X_5 & = X_4 \pm 2^{-2} Y_4 \\
X_6 & = X_5 \pm 2^{-3} Y_5 \\
X_7 & = X_6 \pm 2^{-4} Y_6 \\
X_8 & = X_7 \pm 2^{-5} Y_7 \\
K_n X_n & = \theta_n
\end{align*}
\]

\[
\begin{align*}
Y_1 & = X_1 \\
Y_2 & = \mp X_1 \\
Y_3 & = Y_2 \mp 2^0 X_2 \\
Y_4 & = Y_3 \mp 2^{-1} X_3 \\
Y_5 & = Y_4 \mp 2^{-2} X_4 \\
Y_6 & = Y_5 \mp 2^{-3} X_5 \\
Y_7 & = Y_6 \mp 2^{-4} X_6 \\
Y_8 & = Y_7 \mp 2^{-5} X_7 \\
\theta_n & = \theta_2 \mp \alpha_2
\end{align*}
\]

Fig. 2.47. Fast CORDIC pipeline

```vhdl
--SIGNAL x, y, z : A0_3S9; -- Array of Bytes
BEGIN
P1: PROCESS(x_in, y_in, reset, clk) --> Behavioral Style
    VARIABLE x, y, z : A0_3S9; -- Array of Bytes
    BEGIN
    IF reset = '1' THEN -- Asynchronous clear
        FOR K IN 0 TO 3 LOOP
            x(k) := 0; y(k) := 0; z(k) := 0;
        END LOOP;
        r <= 0; eps <= 0; phi <= 0;
    ELSIF rising_edge(clk) THEN
        r <= x(3); -- Compute last value first in
        phi <= z(3); -- sequential VHDL statements !!
        eps <= y(3);
    IF y(2) >= 0 THEN -- Rotate 14 degrees

```
Figure 2.48 shows the simulation of the conversion of $X_0 = -41$, and $Y_0 = 55$. Note that the radius is enlarged to $R = X_K = 111 = 1.618 \sqrt{X_0^2 + Y_0^2}$ and the accumulated angle in degrees is $\arctan(Y_0/X_0) = 123^\circ$. The design requires 276 LEs and runs with a Speed synthesis optimization at 209.6 MHz using no embedded multiplier.
The actual LE count in the previous example is larger than that expected for a four-stage 8-bit pipeline design that is $5 \times 8 \times 3 = 120$ LEs. The increase by a factor of two comes from the fact that a FPGA uses an $N$-bit switchable adder/subtractor that needs $2N$ LEs. It needs $2N$ LEs because the LE has only three inputs in the fast arithmetic mode, and the switch mode needs four input LUTs. An ALM type LE, see Fig. 1.7a, p. 12, would be needed, with at least four inputs per LE, to reduce the count by a factor of two.

2.10 Computation of Special Functions using MAC Calls

The CORDIC algorithm introduced in the previous section allows one to implement a wide variety of functions at a moderate implementation cost. The only disadvantage is that some high-precision functions need a large number of iterations, because the number of bits is linearly proportional to the number of iterations. In a pipelined implementation this results in a large latency.

With the advent of fast embedded array multipliers in new FPGA families like Spartan or Cyclone, see Table 1.4 (p. 11), the implementation of special functions via a polynomial approximation has becomes a viable option. We have introduced the Taylor series approximation in (2.55), p. 132. The Taylor series approximation converges fast for some functions, e.g., $\exp(x)$, but
needs many product terms for some other special functions, e.g., \( \arctan(x) \), to approximate with sufficient precision. In these cases a Chebyshev approximation can be used to shorten the number of iterations or product terms required.

2.10.1 Chebyshev Approximations

The Chebyshev approximation is based on the Chebyshev polynomial

\[
T_k(x) = \cos (k \times \arccos(x)) \tag{2.57}
\]

defined for the range \(-1 \leq x \leq 1\). The \( T_k(x) \) may look like trigonometric functions, but using some algebraic identities and manipulations allow us to write (2.57) as a true polynomial. The first few polynomials look like

\[
\begin{align*}
T_0(x) &= 1 \\
T_1(x) &= x \\
T_2(x) &= 2x^2 - 1 \\
T_3(x) &= 4x^3 - 3x \\
T_4(x) &= 8x^4 - 8x^2 + 1 \\
T_5(x) &= 16x^5 - 20x^3 + 5x \\
T_6(x) &= 32x^6 - 48x^4 + 18x^2 - 1 \\
\end{align*}
\tag{2.58}
\]

In [85] we find a list of the first 12 polynomials. The first six polynomials are graphical interpreted in Fig. 2.49. In general, Chebyshev polynomials obey the following iterative rule

\[
T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x) \quad \forall \, k \geq 2. \tag{2.59}
\]

A function approximation can now be written as

\[
f(x) = \sum_{k=0}^{N-1} c(k)T_k(x). \tag{2.60}
\]

Because all discrete Chebyshev polynomials are orthogonal to each other it follows that forward and inverse transform are unique, i.e., bijective [86, p. 191]. The question now is why (2.60) is so much better than, for instance, a polynomial using the Taylor approximation (2.55)

\[
f(x) = \sum_{k=0}^{N-1} \frac{f^k(x_0)}{k!}(x - x_0)^k = \sum_{k=0}^{N-1} p(k)(x - x_0)^k, \tag{2.61}
\]

There are mainly three reasons. First (2.60) is a very close (but not exact) approximation to the very complicated problem of finding the function approximation with a minimum of the maximum error, i.e., an optimization
of the $l_{\infty}$ norm $\max(f(x) - \hat{f}(x)) \to \min$. The second reason we prefer (2.60) is the fact, that a pruned polynomial with $M << N$ still gives a minimum/maximum approximation, i.e., a shorter sum still gives a Chebyshev approximation as if we had started the computation with $M$ as the target from the very start. Last but not least we gain from the fact that (2.60) can be computed (for all functions of relevance) with much fewer coefficients than would be required for a Taylor approximation of the same precision. Let us study these special function approximation in the following for popular functions, like trigonometric, exponential, logarithmic, and the square root functions.

### 2.10.2 Trigonometric Function Approximation

As a first example we study the inverse tangent function

$$f(x) = \arctan(x), \quad (2.62)$$

where $x$ is specified for the range $-1 \leq x \leq 1$. If we need to evaluate function values outside this interval, we can take advantage of the relation

$$\arctan(x) = 0.5 - \arctan(1/x). \quad (2.63)$$
Embedded multipliers in Altera FPGAs have a basic size of $9 \times 9$ bits, i.e., 8 bits plus sign bit data format, or $18 \times 18$ bit, i.e., 17 bits plus sign data format. We will therefore in the following always discuss two solutions regarding these two different word sizes.

Fig. 2.50a shows the exact value and approximation for 8-bit quantization, and Fig. 2.50b displays the error, i.e., the difference between the exact function value and the approximation. The error has the typical alternating minimum/maximum behavior of all Chebyshev approximations. The approximation with $N = 6$ already gives an almost perfect approximation. If we use fewer coefficients, e.g., $N = 2$ or $N = 4$, we will have a more-substantial error; see Exercise 2.26 (p. 177).

For 8-bit precision we can see from Fig. 2.50d that $N = 6$ coefficients are sufficient. From Fig. 2.50c we conclude that all even coefficients are zero, because $\arctan(x)$ is an odd symmetric function with respect to $x = 0$. The function to be implemented now becomes
\[ f(x) = \sum_{k=0}^{N-1} c(k)T_k(x) \]

\[ f(x) = c(1)T_1(x) + c(3)T_3(x) + c(5)T_5(x) \]

\[ f(x) = 0.8284T_1(x) - 0.0475T_3(x) + 0.0055T_5(x). \]  
(2.64)

To determine the function values in (2.64) we can substitute the \( T_n(x) \) from (2.58) and solve (2.64). It is however more efficient to use the iterative rule (2.59) for the function evaluation. This is known as Clenshaw’s recurrence formula \([86, \text{p.} 193]\) and works as follows:

\[ d(N) = d(N + 1) = 0 \]

\[ d(k) = 2xd(k + 1) - d(k + 2) + c(k) \quad k = N - 1, N - 2, \ldots, 1 \]

\[ f(x) = d(0) = xd(1) - d(2) + c(0) \]  
(2.65)

For our \( N = 6 \) system with even coefficients equal to zero we can simplify (2.65) to

\[ d(5) = c(5) \]

\[ d(4) = 2xc(5) \]

\[ d(3) = 2xd(4) - d(5) + c(3) \]

\[ d(2) = 2xd(3) - d(4) \]

\[ d(1) = 2xd(2) - d(3) + c(1) \]

\[ f(x) = xd(1) - d(2). \]  
(2.66)

We can now start to implement the \( \arctan(x) \) function approximation in HDL.

**Example 2.28: \( \arctan \) Function Approximation**

If we implement the \( \arctan(x) \) using the embedded 9 × 9 bit multipliers we have to take into account that our values are in the range \(-1 \leq x < 1\). We therefore use a fractional integer representation in a 1.8 format. In our HDL simulation these fractional numbers are represented as integers and the values are mapped to the range \(-256 \leq x < 256\). We can use the same number format for our Chebyshev coefficients because they are all less than 1, i.e., we quantize

\[ c(1) = 0.8284 = 212/256, \]  
(2.67)

\[ c(3) = -0.0475 = -12/256, \]  
(2.68)

\[ c(5) = 0.0055 = 1/256. \]  
(2.69)

The following VHDL code\(^{10}\) shows the \( \arctan(x) \) approximation using polynomial terms up to \( N = 6 \):

```vhdl
PACKAGE n_bits_int IS -- User defined types
  SUBTYPE S9 IS INTEGER RANGE -2**8 TO 2**8-1;
  TYPE A1_5S9 IS ARRAY (1 TO 5) OF S9;
END n_bits_int;
```

\(^{10}\) The equivalent Verilog code `arctan.v` for this example can be found in Appendix A on page 806. Synthesis results are shown in Appendix B on page 881.
END n_bits_int;

LIBRARY work;
USE work.n_bits_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;

-- --------------------------------------------------------
ENTITY arctan IS ------> Interface
PORT (clk : IN STD_LOGIC; -- System clock
reset : IN STD_LOGIC; -- Asynchronous reset
x_in : IN S9; -- System input
d_o : OUT A1_5S9; -- Auxiliary recurrence
f_out : OUT S9); -- System output
END arctan;
-- --------------------------------------------------------
ARCHITECTURE fpga OF arctan IS

SIGNAL x, f : S9; -- Auxiliary signals
SIGNAL d : A1_5S9 := (0,0,0,0,0); -- Auxiliary array

-- Chebychev coefficients for 8-bit precision:
CONSTANT c1 : S9 := 212;
CONSTANT c3 : S9 := -12;
CONSTANT c5 : S9 := 1;

BEGIN

STORE: PROCESS(reset, clk) -----> I/O store in register
BEGIN
  IF reset = '1' THEN -- Asynchronous clear
    x <= 0; f_out <= 0;
  ELSIF rising_edge(clk) THEN
    x <= x_in;
    f_out <= f;
  END IF;
END PROCESS;

--> Compute sum-of-products:
SOP: PROCESS(x, d)
BEGIN
  -- Clenshaw’s recurrence formula
  d(5) <= c5;
  d(4) <= x * d(5) / 128;
  d(3) <= x * d(4) / 128 - d(5) + c3;
  d(2) <= x * d(3) / 128 - d(4);
  d(1) <= x * d(2) / 128 - d(3) + c1;
  f <= x * d(1) / 256 - d(2); -- last step is different
END PROCESS SOP;

d_o <= d; -- Provide some test signals as outputs
Fig. 2.51. VHDL simulation of the \( \arctan(x) \) function approximation for the values \( x = -1 = -256/256, x = -0.5 = -128/256, x = 0, x = 0.5 = 128/256, x = 1 \approx 255/256 \)

\[
\text{END fpga;}
\]

The first \texttt{PROCESS} is used to infer registers for the input and output data. The next \texttt{PROCESS} blocks \texttt{SOP} include the computation of the Chebyshev approximation using Clenshaw’s recurrence formula. The iteration variables \( d(k) \) are also connected to the output ports so we can monitor them. The design uses 106 LEs, three embedded multipliers, and has an \( \text{Fmax}=32.71 \text{ MHz} \) registered performance using the \texttt{TimeQuest slow 85C} model. Comparing FLEX and Cyclone synthesis data we can conclude that the use of embedded multipliers saves many LEs.

A simulation of the \( \arctan \) function approximation is shown in Fig. 2.51. The simulation shows the result for five different input values:

\[
\begin{array}{c|c|c|c|c}
 x & f(x) = \arctan(x) & \hat{f}(x) & |\text{error}| & \text{Eff. bits} \\
-1.0 & -0.7854 & -201/256 = -0.7852 & 0.0053 & 7.6 \\
-0.5 & -0.4636 & -118/256 = -0.4609 & 0.0027 & 7.4 \\
0 & 0.0 & 0 & 0 & - \\
0.5 & 0.4636 & 118/256 = 0.4609 & 0.0027 & 7.4 \\
1.0 & 0.7854 & 200/256 = 0.7812 & 0.0053 & 7.6 \\
\end{array}
\]

Note that, due to the I/O registers, the output values appear with a delay of one clock cycle.

If the precision in the previous example is not sufficient we can use more coefficients. The odd Chebyshev coefficients for 16-bit precision, for instance, would be

\[
c(2k + 1) = (0.82842712, -0.04737854, 0.00487733, -0.00059776, 0.00008001, -0.00001282).
\]
2.2. Computer Arithmetic

\begin{align*}
\arctan(x) &= x - \frac{x^3}{3} + \frac{x^5}{5} + \ldots (-1)^k \frac{x^{2k+1}}{2k+1} \\
p(2k+1) &= (1, -0.3, 0.2, -0.14285714, 0.1, -0.09)
\end{align*}

we see that the Taylor coefficients converge very slowly compared with the Chebyshev approximation.

There are two more common trigonometric functions. One is the \( \sin(x) \) and the other is the \( \cos(x) \) function. There is however a small problem with these functions. The argument is usually defined only for the first quadrant, i.e., \( 0 \leq x \leq \pi/2 \), and the other quadrants values are computed via

\[ \sin(x) = -\sin(-x) \quad \text{and} \quad \sin(x) = \sin(\pi/2 - x) \]  

\[ \cos(x) = \cos(-x) \quad \text{and} \quad \cos(x) = -\cos(\pi/2 - x). \]  

We may also find that sometimes the data are normalized \( f(x) = \sin(x\pi/2) \) or degree values are used, i.e., \( 0^\circ \leq x \leq 90^\circ \). Figure 2.52a shows the exact value

\[ \arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} + \ldots (-1)^k \frac{x^{2k+1}}{2k+1} \]  

\[ p(2k+1) = (1, -0.3, 0.2, -0.14285714, 0.1, -0.09) \]
and approximation for 16-bit quantization, and Fig. 2.52b displays the error, i.e., the difference between the exact function values and the approximation. In Fig. 2.53 the same data are plotted for the \( \cos(x \pi/2) \) function. The problem now is that our Chebyshev polynomials are only defined for the range \( x \in [-1, 1] \). Which brings up the question, how the Chebyshev approximation has to be modified to take care of different range values? Luckily this does not take too much effort, we just make a linear transformation of the input values. Suppose the function \( f(y) \) to be approximated has a range \( y \in [a, b] \) then we can develop our function approximation using a change of variable defined by

\[
y = \frac{2x - b - a}{b - a}.
\]  

(2.74)

Now if we have for instance in our \( \sin(x \pi/2) \) function \( x \) in the range \( x = [0, 1] \), i.e., \( a = 0 \) and \( b = 1 \), it follows that \( y \) has the range \( y = [(2 \times 0 - 1 - 0)/(1 - 0), (2 \times 1 - 1 - 0)/(1 - 0)] = [-1, 1] \), which we need for our Chebyshev approximation. If we prefer the degree representation then \( a = 0 \) and \( b = 90 \), and we will use the mapping \( y = (2x - 90)/90 \) and develop the Chebyshev approximation in \( y \).

The final question we discuss is regarding the polynomial computation. You may ask if we really need to compute the Chebyshev approximation via the Clenshaw’s recurrence formula (2.65) or if we can use instead the direct polynomial approximation, which requires one fewer add operation per iteration:

\[
f(x) = \sum_{k=0}^{N-1} p(k)x^k
\]

(2.75)

or even better use the Horner scheme

\[
s(N - 1) = p(N - 1)
\]

\[
s(k) = s(k + 1) \times x + p(k) \quad k = N - 2, \ldots, 0.
\]

(2.76)

We can of course substitute the Chebyshev functions (2.58) in the approximation formula (2.60), because the \( T_n(x) \) do not have terms of higher order than \( x^n \). However there is one important disadvantage to this approach. We will lose the pruning property of the Chebyshev approximation, i.e., if we use in the polynomial approximation (2.75) fewer than \( N \) terms, the pruned polynomial will no longer be an \( l_\infty \) optimized polynomial. Figure 2.50d (p. 144) shows this property. If we use all 6 terms the Chebyshev and the associated polynomial approximation will have the same precision. If we now prune the polynomial, the Chebyshev function approximation (2.60) using the \( T_n(x) \) has more precision than the pruned polynomial using (2.75). The resulting precision is much lower than the equivalent pruned Chebyshev function approximation of the same length. In fact it is not much better than the Taylor
So the solution to this problem is not complicated: if we want to shorten the length \( M < N \) of our polynomial approximation (2.75) we need to develop first a Chebyshev approximation for length \( M \) and then compute the polynomial coefficient \( g(k) \) from this pruned Chebyshev approximation. Let us demonstrate this with a comparison of 8- and 16-bit \( \arctan(x) \) coefficients. The substitution of the Chebyshev functions (2.58) into the coefficient (2.70) gives the following odd coefficients:

\[
g(2k + 1) = (0.99999483, -0.33295711, 0.19534659, -0.12044859, 0.05658999, -0.01313038).
\] (2.77)

If we now use the length \( N = 6 \) approximation from (2.67) the odd coefficient will be

\[
g(2k + 1) = (0.9982, -0.2993, 0.0876).
\] (2.78)

Although the pruned Chebyshev coefficients are the same, we see from a comparison of (2.77) and (2.78) that the polynomial coefficient differ essentially. The coefficient \( g(5) \) for instance has a factor of 2 difference.
We can summarize the Chebyshev approximation in the following procedure.

**Algorithm 2.29: Chebyshev Function Approximation**

1) Define the number of coefficients $N$.
2) Transform the variable from $x$ to $y$ using (2.74)
3) Determine the Chebyshev approximation in $y$.
4) Determine the direct polynomial coefficients $g(k)$ using Clenshaw’s recurrence formula.
5) Build the inverse of the mapping $y$.

If we apply these five steps to our $\sin(x \pi/2)$ function for $x \in [0,1]$ with four nonzero coefficients, we get the following polynomials sufficient for a 16-bit quantization

$$f(x) = \sin(x \pi/2)$$

$$= 1.57035062x + 0.00508719x^2 - 0.66666099x^3$$

$$+ 0.03610310x^4 + 0.05512166x^5$$

$$= (51457x + 167x^2 - 21845x^3 + 1183x^4 + 1806x^5)/32768.$$ 

Note that the first coefficient is larger than 1 and we need to scale appropriate. This is quite different from the Taylor approximation given by

$$\sin\left(\frac{x \pi}{2}\right) = \frac{x \pi}{2} - \frac{1}{3!} \left(\frac{x \pi}{2}\right)^3 + \frac{1}{5!} \left(\frac{x \pi}{2}\right)^5$$

$$+ \ldots + \frac{(-1)^k}{(2k+1)!} \left(\frac{x \pi}{2}\right)^{2k+1}.$$ 

Figure 2.52c shows a graphical illustration. For an 8-bit quantization we would use

$$f(x) = \sin(x \pi/2) = 1.5647x + 0.0493x^2 - 0.7890x^3 + 0.1748x^4$$

$$= (200x + 6x^2 - 101x^3 + 22x^4)/128. \quad (2.79)$$

Although we would expect that, for an odd symmetric function, all even coefficients are zero, this is not the case in this approximation, because we only used the interval $x \in [0,1]$ for the approximation. The $\cos(x)$ function can be derived via the relation

$$\cos\left(\frac{x \pi}{2}\right) = \sin\left(\left(x + 1\right) \frac{\pi}{2}\right) \quad (2.80)$$

or we may also develop a direct Chebyshev approximation. For $x \in [0,1]$ with four nonzero coefficients and get the following polynomial for a 16-bit quantization

$$f(x) = \cos\left(\frac{x \pi}{2}\right)$$

$$= 1.00000780 - 0.00056273x - 1.22706059x^2$$

$$- 0.02896799x^3 + 0.31171138x^4 - 0.05512166x^5$$

$$= (32768 - 18x - 40208x^2 - 949x^3 + 10214x^4 - 1806x^5)/32768.$$
For an 8-bit quantization we would use
\[ f(x) = \cos\left(\frac{x\pi}{2}\right) \]
\[ = (0.9999 + 0.0046x - 1.2690x^2 + 0.0898x^3 + 0.1748x^4) \]
\[ = (128 + x - 162x^2 + 11x^3 + 22x^4)/128. \]  
(2.81)
(2.82)

Again the Taylor approximation has quite different coefficients:
\[ \cos\left(\frac{x\pi}{2}\right) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \ldots + \frac{(-1)^k x^{2k}}{(2k)!}. \]

Figure 2.52c shows a graphical illustration of the coefficients. We notice from Fig. 2.52d that with the same number (i.e., six) of terms \(x^k\) the Taylor approximation only provides about 6 bit accuracy, while the Chebyshev approximation has 16-bit precision.

### 2.10.3 Exponential and Logarithmic Function Approximation

The exponential function is one of the few functions who’s Taylor approximation converges relatively fast. The Taylor approximation is given by
\[ f(x) = e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \ldots + \frac{x^k}{k!} \]
with \(0 \leq x \leq 1\). For 16-bit polynomial quantization computed using the Chebyshev coefficients we would use:
\[ f(x) = e^x \]
\[ = 1.00002494 + 0.99875705x + 0.50977984x^2 \]
\[ + 0.14027504x^3 + 0.06941551x^4 \]
\[ = (32769 + 32727x + 16704x^2 + 4597x^3 + 2275x^4)/32768. \]

Only terms up to order \(x^4\) are required to reach 16-bit precision. We notice also from Fig. 2.54c that the Taylor and polynomial coefficient computed from the Chebyshev approximation are quite similar. If 8 bits plus sign precision are sufficient, we use
\[ f(x) = e^x = 1.0077 + 0.8634x + 0.8373x^2 \]
\[ = (129 + 111x + 107x^2)/128. \]  
(2.83)
(2.84)

Based on the fact that one coefficient is larger than \(c(0) > 1.0\) we need to select a scaling factor of 128.

The input needs to be scaled in such a way that \(0 \leq x \leq 1\). If \(x\) is outside this range we can use the identity
\[ e^{sx} = (e^x)^s \]
2.10 Computation of Special Functions using MAC Calls 153

Because $s = 2^k$ is a power-of-two value this implies that a series of squaring operations need to follow the exponential computation. For a negative exponent we can use the relation

$$e^{-x} = \frac{1}{e^x},$$  \hspace{1cm} (2.86)

or develop a separate approximation. If we like to build a direct function approximation to $f(x) = e^{-x}$ we have to alternate the sign of each second term in (2.83). For a Chebyshev polynomial approximation we get additional minor changes in the coefficients. For a 16-bit Chebyshev polynomial approximation we use

$$f(x) = e^{-x} = 0.99998916 - 0.99945630x + 0.49556967x^2 - 0.15375046x^3 + 0.02553654x^4 = (65535 - 65500x + 32478x^2 - 10076x^3 + 1674x^4)/65536.$$
where \( x \) is defined for the range \( x \in [0, 1] \). Note that, based on the fact that all coefficients are less than 1, we can select a scaling by a factor of 2 larger than in (2.84). From Fig. 2.55d we conclude that three or five coefficients are required for 8- and 16-bit precision, respectively. For 8-bit quantization we would use the coefficients
\[
f(x) = e^{-x} = 0.9964 - 0.9337x + 0.3080x^2
= (255 - 239x + 79x^2)/256.
\] (2.87)

The inverse to the exponential function is the logarithm function, which is typically approximated for the argument in the range \([1, 2]\). As notation this is typically written as \( f(x) = \ln(1 + x) \) now with \( 0 \leq x \leq 1 \). Figure 2.56a shows the exact and 16-bit quantized approximation for this range. The approximation with \( N = 6 \) gives an almost perfect approximation. If we use fewer coefficients, e.g., \( N = 2 \) or \( N = 3 \), we will have a more substantial error; see Exercise 2.29 (p. 177).
The Taylor series approximation in no longer fast converging as for the exponential function

\[
f(x) = \ln(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} + \ldots + \frac{(-1)^{k+1}x^k}{k}
\]
as can be seen from the linear factor in the denominator. A 16-bit Chebyshev approximation converges much faster, as can be seen from Fig. 2.56d. Only six coefficients are required for 16-bit precision. With six Taylor coefficients we get less than 4-bit precision. For 16-bit polynomial quantization computed using the Chebyshev coefficients we would use

\[
f(x) = \ln(1 + x) = 0.00001145 + 0.99916640x - 0.48969909x^2
+0.28382318x^3 - 0.12995720x^4 + 0.02980877x^5
= (1 + 65481x - 32093x^2 + 18601x^3 - 8517x^4 + 1954x^5)/65536.
\]
Only terms up to order \( x^5 \) are required to get 16-bit precision. We also notice from Fig. 2.56c that the Taylor and polynomial coefficient computed from the Chebyshev approximation are similar only for the first three coefficients.

We can now start to implement the \( \ln(1 + x) \) function approximation in HDL.

**Example 2.30: \( \ln(1+x) \) Function Approximation**

If we implement the \( \ln(1 + x) \) using embedded \( 18 \times 18 \) bit multipliers we have to take into account that our values \( x \) are in the range \( 0 \leq x < 1 \). We therefore use a fractional integer representation with a 2.16 format. We use an additional guard bit that guarantees no problem with any overflow and that \( x = 1 \) can be exactly represented as \( 2^{16} \). We use the same number format for our Chebyshev coefficients because they are all less than 1.

The following VHDL code\(^{11}\) shows the \( \ln(1 + x) \) approximation using six coefficients:

```vhdl
PACKAGE n_bits_int IS -- User defined types
  SUBTYPE S9 IS INTEGER RANGE -2**8 TO 2**8-1;
  SUBTYPE S18 IS INTEGER RANGE -2**17 TO 2**17-1;
  TYPE A0_5S18 IS ARRAY (0 TO 5) of S18;
END n_bits_int;

LIBRARY work;
USE work.n_bits_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;
-- --------------------------------------------------------
ENTITY ln IS ------> Interface
  GENERIC (N : INTEGER := 5); -- Number of Coeffcients-1
  PORT (clk : IN STD_LOGIC; -- System clock
        reset : IN STD_LOGIC; -- Asynchron reset
        x_in : IN S18; -- System input
        f_out : OUT S18:=0); -- System output
END ln;
-- --------------------------------------------------------
ARCHITECTURE fpga OF ln IS

  -- Polynomial coefficients for 16 bit precision:
  -- \( f(x) = \frac{(1 + 65481 x -32093 x^2 + 18601 x^3
    -- -8517 x^4 + 1954 x^5)}{65536} \)
  CONSTANT p : A0_5S18 :=
    (1, 65481, -32093, 18601, -8517, 1954)/(65536);

BEGIN

  STORE: PROCESS(reset, clk) -----> I/O store in register
```

\(^{11}\) The equivalent Verilog code \texttt{ln.v} for this example can be found in Appendix A on page 807. Synthesis results are shown in Appendix B on page 881.
BEGIN
  IF reset = '1' THEN -- Asynchronous clear
    x <= 0; f_out <= 0;
  ELSIF rising_edge(clk) THEN
    x <= x_in;
    f_out <= f;
  END IF;
END PROCESS;

--> Compute sum-of-products:
SOP: PROCESS (x,s)
VARIABLE slv : STD_LOGIC_VECTOR(35 DOWNTO 0);
BEGIN
  -- Polynomial Approximation from Chebyshev coeffiecients
  s(N) <= p(N);
  FOR K IN N-1 DOWNTO 0 LOOP
    slv := CONV_STD_LOGIC_VECTOR(x,18)
       * CONV_STD_LOGIC_VECTOR(s(K+1),18);
    s(K) <= CONV_INTEGER(slv(33 downto 16)) + p(K);
  END LOOP; -- x*s/65536 problem 32 bits
  f <= s(0); -- make visible outside
END PROCESS SOP;

END fpga;

The first PROCESS is used to infer the registers for the input and output data. The next PROCESS blocks SOP includes the computation of the Chebyshev approximation using a sum of product computations. The multiply and scale arithmetic is implemented with standard logic vectors data types because the 36-bit products are larger than the valid 32-bit range allowed for integers. The design uses 88 LEs, ten embedded $9 \times 9$-bit multipliers (or half of that for $18 \times 18$-bit multipliers), and has an $F_{\text{max}}=29.2$ MHz registered performance using the TimeQuest slow 85C model.

A simulation of the function approximation is shown in Fig. 2.57. The simulation shows the result for five different input values:
If we compare the polynomial code of the \( \ln \) function with Clenshaw’s recurrence formula from Example 2.28 (p. 145), we notice the reduction by one adder in the design.

If 8 bit plus sign precision is sufficient, we use

\[
f(x) = \ln(1 + x) = 0.0006 + 0.9813x - 0.3942x^2 + 0.1058x^3
\]

\[
= (251x - 101x^2 + 27x^3)/256.
\]
Based on the fact that no coefficient is larger than 1.0 we can select a scaling factor of 256.

If the argument $x$ is not in the valid range $[0, 1]$, using the following algebraic manipulation with $y = sx = 2^k x$ we get

$$\ln(sx) = \ln(s) + \ln(x) = k \times \ln(2) + \ln(x), \quad (2.89)$$

i.e., we normalize by a power-of-two factor such that $x$ is again in the valid range. If we have determined $s$, the addition arithmetic effort is only one multiply and one add operation.

If we like to change to another base, e.g., base 10, we can use the following rule:

$$\log_a(x) = \ln(x)/\ln(a), \quad (2.90)$$

i.e., we only need to implement the logarithmic function for one base and can deduce it for any other base. On the other hand the divide operation may be expensive to implement too and we can alternatively develop a separate Chebyshev approximation. For base 10 we would use, in 16-bit precision, the following Chebyshev polynomial coefficients:

$$f(x) = \log_{10}(1 + x)$$

$$= 0.00000497 + 0.43393245x - 0.21267361x^2$$

$$+ 0.12326284x^3 - 0.05643969x^4 + 0.01294578x^5$$

$$= (28438x - 13938x^2 + 8078x^3 - 3699x^4 + 848x^5)/65536$$

for $x \in [0, 1]$. Figure 2.58a shows the exact and 8-bit quantized function of $\log_{10}(1 + x)$. For an 8-bit quantization we would use the following approximation:

$$f(x) = \log_{10}(1 + x)$$

$$= 0.0002 + 0.4262x - 0.1712x^2 + 0.0460x^3$$

$$= (109x - 44x^2 + 12x^3)/256, \quad (2.91)$$

which uses only three nonzero coefficients, as shown in Fig. 2.58d.

### 2.10.4 Square Root Function Approximation

The development of a Taylor function approximation for the square root cannot be computed around $x_0 = 0$ because then all derivatives $f^n(x_0)$ would be zero or even worse $1/0$. However, we can compute a Taylor series around $x_0 = 1$ for instance. The Taylor approximation would then be

$$f(x) = \sqrt{x}$$

$$= \frac{(x - 1)^0}{0!} + \frac{0.5(x - 1)^1}{1!} - \frac{0.5^2}{2!}(x - 1)^2 + \frac{0.5^2 1.5}{3!}(x - 1)^3 - \ldots$$

$$= 1 + \frac{x - 1}{2} - \frac{(x - 1)^2}{8} + \frac{(x - 1)^3}{16} - \frac{5}{128} (x - 1)^4 + \ldots$$
The coefficient and the equivalent Chebyshev coefficient are graphically interpreted in Fig. 2.59c. For 16-bit polynomial quantization computed using the Chebyshev coefficient we would use

\[ f(x) = \sqrt{x} = 0.23080201 + 1.29086721x - 0.88893983x^2 + 0.48257525x^3 - 0.11530993x^4 = (7563 + 42299x - 29129x^2 + 15813x^3 - 3778x^4)/32768. \]

The valid argument range is \( x \in [0.5, 1) \). Only terms up to order \( x^4 \) are required to get 16-bit precision. We also notice from Fig. 2.59c that the Taylor and polynomial coefficients computed from the Chebyshev approximation are not similar. The approximation with \( N = 5 \) shown in Fig. 2.59a is almost a perfect approximation. If we use fewer coefficients, e.g., \( N = 2 \) or \( N = 3 \), we will have a more-substantial error; see Exercise 2.30 (p. 178).

The only thing left to discuss is the question of how to handle argument values outside the range \( 0.5 \leq x < 1 \). For the square root operation this can
be done by splitting the argument $y = sx$ into a power-of-two scaling factor $s = 2^k$ and the remaining argument with a valid range of $0.5 \leq x < 1$. The square root for the scaling factor is accomplished by

$$\sqrt{s} = \sqrt{2^k} = \begin{cases} 2^{k/2} & k \text{ even} \\ \sqrt{2} \times 2^{(k-1)/2} & k \text{ odd} \end{cases}$$

We can now start to implement the $\sqrt{x}$ function approximation in HDL.

**Example 2.31: Square Root Function Approximation**

We can implement the function approximation in a parallel way using $N$ embedded $18 \times 18$ bit multiplier or we can build an FSM to solve this iteratively. Other FSM design examples can be found in Exercises 2.20, p. 172 and 2.21, p. 173. In a first design step we need to scale our data and coefficients in such a way that overflow-free processing is guaranteed. In addition we need a pre- and post-scaling such that $x$ is in the range $0.5 \leq x < 1$. We therefore use a fractional integer representation in 3.15 format. We use two additional guard bits that guarantee no problem with any overflow and that $x = 1$ can be exact represented as $2^{15}$. We use the same number format for our Chebyshev coefficients because they are all less than 2.

The following VHDL code\(^{12}\) shows the $\sqrt{x}$ approximation using $N = 5$ coefficients:

```vhdl
PACKAGE n_bits_int IS  -- User defined types
    SUBTYPE S9 IS INTEGER RANGE -2**8 TO 2**8-1;
    SUBTYPE S17 IS INTEGER RANGE -2**16 TO 2**16-1;
    TYPE A0_4S17 IS ARRAY (0 TO 4) of S17;
    TYPE STATE_TYPE IS
        (start, leftshift, sop, rightshift, done);
    TYPE OP_TYPE IS (load, mac, scale, denorm, nop);
END n_bits_int;

LIBRARY work;
USE work.n_bits_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;
-- --------------------------------------------------------
ENTITY sqrt IS ------> Interface
    PORT (clk : IN STD_LOGIC; -- System clock
          reset : IN STD_LOGIC; -- Asynchron reset
          count_o : OUT INTEGER RANGE 0 TO 3; -- Counter SLL
          x_in : IN S17; -- System input
          pre_o : OUT S17; -- Prescaler
          x_o : OUT S17; -- Normalized x_in
          post_o : OUT S17; -- Postscaler
          ind_o : OUT INTEGER RANGE -1 TO 4; -- Index to p
          imm_o : OUT S17; -- ALU preload value
          a_o : OUT S17; -- ALU factor;

END sqrt;
```

\(^{12}\) The equivalent Verilog code `sqrt.v` for this example can be found in Appendix A on page 809. Synthesis results are shown in Appendix B on page 881.
ARCHITECTURE fpga OF sqrt IS

SIGNAL s : STATE_TYPE;
SIGNAL op : OP_TYPE;

SIGNAL x : S17 := 0; -- Auxiliary
SIGNAL a, b, f, imm : S17 := 0; -- ALU data
-- Chebychev poly coefficients for 16 bit precision:
CONSTANT p : A0_4S17 :=
(7563,42299,-29129,15813,-3778);
SIGNAL pre, post : S17;

BEGIN

States: PROCESS(clk, reset) -----> SQRT in behavioral style
VARIABLE ind : INTEGER RANGE -1 TO 4 := 0;
VARIABLE count : INTEGER RANGE 0 TO 3;
BEGIN
IF reset = '1' THEN -- Asynchronous reset
s <= start;
f_out <= 0;
ELSIF rising_edge(clk) THEN
CASE s IS
WHEN start => -- Initialization step
s <= leftshift;
ind := 4;
imm <= x_in; -- Load argument in ALU
op <= load;
count := 0;
WHEN leftshift => -- Normalize to 0.5 .. 1.0
count := count + 1;
a <= pre;
op <= scale;
imm <= p(4);
IF count = 2 THEN
op <= NOP;
END IF;
IF count = 3 THEN -- Normalize ready ?
s <= sop;
op <= load;
x <= f;
END IF;
WHEN sop => -- Processing step
ind := ind - 1;
a <= x;
IF ind =-1 THEN -- SOP ready ?
s <= rightshift;
op <= denorm;
a <= post;
ELSE
    imm <= p(ind);
    op <= mac;
END IF;
WHEN rightshift => -- Denormalize to original range
    s <= done;
    op <= nop;
WHEN done => -- Output of results
    f_out <= f; ------> I/O store in register
    op <= nop;
    s <= start; -- start next cycle
END CASE;
END IF;
ind_o <= ind;
count_o <= count;
END PROCESS States;

ALU: PROCESS(clk, reset)
BEGIN
    IF reset = '1' THEN -- Asynchronous clear
        f <= 0;
    ELSIF rising_edge(clk) THEN
        CASE OP IS
            WHEN load => f <= imm;
            WHEN mac => f <= a * f /32768 + imm;
            WHEN scale => f <= a * f;
            WHEN denorm => f <= a * f /32768;
            WHEN nop => f <= f;
            WHEN others => f <= f;
        END CASE;
    END IF;
END PROCESS ALU;

EXP: PROCESS(x_in)
VARIABLE slv : STD_LOGIC_VECTOR(16 DOWNTO 0);
VARIABLE po, pr : S17;
BEGIN
    slv := CONV_STD_LOGIC_VECTOR(x_in, 17);
    pr := 2**14; -- Compute pre- and post scaling
    pre <= 0;
    FOR K IN 0 TO 15 LOOP
        IF slv(K) = '1' THEN
            pre <= pr;
        END IF;
        pr := pr / 2;
    END LOOP;
    po := 1; -- Compute pre- and post scaling
    FOR K IN 0 TO 7 LOOP
        IF slv(2*K) = '1' THEN -- even 2^k get 2^k/2
            po := 256 * 2**K;
        END IF;
    END LOOP;
    -- sqrt(2): CSD Error = 0.0000208 = 15.55 effective bits
    -- +1 +0. -1 +0 -1 +0 +1 +0 +1 +0 +0 +0 +0 +0 +1
Fig. 2.60. VHDL simulation of the $\sqrt{x}$ function approximation for the value $x = 0.75/8 = 3072/32768$

\[
\begin{align*}
\text{IF } \text{lsv}(2+K+1) = '1' \text{ THEN } & \text{ odd } k \text{ has } \sqrt{2} \text{ factor} \\
\text{po} := & \ 2**((K+9)-2**((K+7)-2**((K+5)+2**((K+3) \\
& +2**((K+1)+2**K/32)); \\
\text{END IF}; \\
\text{post} <= & \ \text{po}; \\
\text{END LOOP;}
\end{align*}
\]

END PROCESS EXP;

\[
\begin{align*}
a_o <= & \ a; \quad \text{-- Provide some test signals as outputs} \\
\text{imm}_o <= & \ \text{imm}; \\
f_o <= & \ f; \\
\text{pre}_o <= & \ \text{pre}; \\
\text{post}_o <= & \ \text{post}; \\
x_o <= & \ x;
\end{align*}
\]

END fpga;

The code consists of three major PROCESS blocks. The control part is placed in the FSM block, while the arithmetic parts can be found in the ALU and EXP blocks. The first FSM PROCESS is used to control the machine and place the data in the correct registers for the ALU and EXP blocks. In the start state the data are initialized and the input data are loaded into the ALU. In the leftshift state the input data are normalized such that the input $x$ is in the range $x \in [0.5, 1)$. The sop state is the major processing step where the polynomial evaluation takes place using multiply-accumulate operations performed by the ALU. At the end data are loaded for the denormalization step, i.e., rightshift state, that reverses the normalization done before. In the final step the result is transferred to the output register and the FSM is ready for the next square root computation. The ALU PROCESS block performs a $f = a \times f + \text{imm}$ operation as used in the Horner scheme (2.76), p. 149 to compute the polynomial function and will be synthesized to a single 18 × 18 embedded multiplier (or two 9 × 9-bit multiplier blocks as reported by Quartus) and some additional add and normalization logic. The block has the form of an ALU, i.e., the signal op is used to determine the current operation.
The accumulator register $f$ can be preloaded with an imm operand. The last PROCESS block EXP hosts the computation of the pre- and post-normalization factors according to (2.93). The $\sqrt{2}$ factor for the odd $k$ values of $2^k$ has been implemented with CSD code computed with the csd.exe program. The design uses 261 LEs, two embedded $9 \times 9$-bit multipliers (or half of that for $18 \times 18$-bit multipliers), and has an $F_{\text{max}} = 86.23$ MHz registered performance using the TimeQuest slow 85C model.

A simulation of the function approximation is shown in Fig. 2.60. The simulation shows the result for the input value $x = 0.75/8 = 0.0938 = 3072/2^{15}$. In the shift phase the input $x = 3072$ is normalized by a pre factor of 8. The normalized result $24576$ is in the range $x \in [0.5, 1) \approx [16384, 32768)$. Then several MAC operations are computed to arrive at $f = \sqrt{0.75 \times 2^{15}} = 28378$. Finally a denormalization with a post factor of $\sqrt{2} \times 2^{13} = 11585$ takes place and the final result $f = \sqrt{0.75/8 \times 2^{15}} = 10032$ is transferred to the output register.

If 8 bit plus sign precision is sufficient, we would build a square root via

$$f(x) = \sqrt{x} = 0.3171 + 0.8801x - 0.1977x^2 = (81 + 225x - 51x^2)/256.$$  \hspace{1cm} (2.94)

Based on the fact that no coefficient is larger than 1.0 we can select a scaling factor of 256.

### 2.10 Fast Magnitude Approximation

In some applications such as FFT, image processing, or automatic gain control (AGC) of an incoherent receiver that uses complex data of the type $x + jy$, a fast approximation of the magnitude $r = \sqrt{x^2 + y^2}$ is needed. In AGC application the magnitude estimation is used to adjust the gain of the incoming signals in such a way that they are neither so small that large quantization noise occurs nor large that the arithmetic will overflow.

Another example is the edge detection in image processing where we like to know, based on the gradient in $G_x$ and $G_y$ direction, whether the overall gradient $\sqrt{G_x^2 + G_y^2}$ has crossed the threshold and is considered to be an edge. Here also not much precision is needed. In image processing we find that a very coarse estimation such as

$$r \approx |x| + |y|$$  \hspace{1cm} (2.95)

is often used. Let us call this a zero $L_0$ approximation to the magnitude estimation. From the trigonometric relation $\sin(\phi) + \cos(\phi) = \sqrt{2}\sin(\phi + \pi/4)$ we see that the error of this estimation $L_0$ can be as large as 40%.

Much more precision is provided with the CORDIC algorithm or polynomial approximation but long delays and high resource penalties must be
expected. If the approximation should be fast, and needs somehow to be more precise than $L_0$, a max/min approximation of the type

$$ r \approx \alpha \max(|x|, |y|) + \beta \min(|x|, |y|) $$

(2.96)
can be used. The approximation will be of low latency and require only a few resources. Such an approximation will be symmetric to 45° and we may choose the factors $\alpha$ and $\beta$ to optimize the $L_1$ (minimum average error) or $L_{\infty}$ (minimum maximum error) norms. Table 2.14 shows the $L_1$ and $L_{\infty}$ optimal values and popular approximations. The $L_{\infty}$ approximation with $(1, 0.375)$, for instance, is used in the HSP50110 communication IC from Intersil [87]. From the effective number of bits shown in column five we see that even with full coefficient precision this is a coarse estimate and the magnitude does not have more than 5 bits precision. Figure 2.61 shows the computed magnitude approximation for these five $\alpha/\beta$ choices optimized by a linear search within a reasonable range. From a practical implementation approach the choice $\beta = 1/4$ seemed to be most interesting with low implementation effort and low average error norm $L_1$.

![Fig. 2.61. Magnitude approximation using max/min method](image_url)
Table 2.14. Coefficient characteristics of the max/min magnitude approximation. The effective bits are based on the $L_\infty$ norm

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$\beta$</th>
<th>$L_1$</th>
<th>$L_\infty$</th>
<th>eff. bits</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>27.1%</td>
<td>41.4%</td>
<td>1.2</td>
<td>$L_0$ opt.</td>
</tr>
<tr>
<td>0.943</td>
<td>0.386</td>
<td>1.97%</td>
<td>6.0%</td>
<td>4.1</td>
<td>$L_1$ approximation</td>
</tr>
<tr>
<td>1</td>
<td>1/4</td>
<td>3.17%</td>
<td>11.6%</td>
<td>3.1</td>
<td>$L_1$ optimum</td>
</tr>
<tr>
<td>0.962</td>
<td>0.396</td>
<td>2.45%</td>
<td>4.0%</td>
<td>4.6</td>
<td>$L_\infty$ optimum</td>
</tr>
<tr>
<td>1</td>
<td>3/8</td>
<td>4.22%</td>
<td>6.8%</td>
<td>3.9</td>
<td>$L_\infty$ approximation</td>
</tr>
</tbody>
</table>

Example 2.32: VHDL Design of Magnitude Circuit

Consider the VHDL code\textsuperscript{13} of a 16-bit magnitude approximation.

```vhdl
PACKAGE N_bit_int IS -- User define types
  SUBTYPE S16 IS INTEGER RANGE -2**15 TO 2**15-1;
END N_bit_int;

LIBRARY work; USE work.N_bit_int.ALL;
LIBRARY ieee; -- Using predefined Packages
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY magnitude IS ------> Interface
  PORT (clk : IN STD_LOGIC; -- System clock
        reset : IN STD_LOGIC; -- Asynchron reset
        x, y : IN S16; -- System inputs
        r : OUT S16 := 0); -- System output
END;

ARCHITECTURE fpga OF magnitude IS
  SIGNAL x_r, y_r : S16 := 0;
  BEGIN
    -- Approximate the magnitude via
    -- $r = \alpha \cdot \max(|x|,|y|) + \beta \cdot \min(|x|,|y|)$
    -- use $\alpha = 1$ and $\beta = 1/4$
    PROCESS(clk, reset, x, y, x_r, y_r)
      VARIABLE mi, ma, ax, ay : S16 := 0; -- temporals
      BEGIN
        IF reset = '1' THEN -- Asynchronous clear
          x_r <= 0; y_r <= 0;
        ELSIF rising_edge(clk) THEN
          x_r <= x; y_r <= y;
        END IF;
        ax := ABS(x_r); -- Take absolute values first
        ay := ABS(y_r);
        IF ax > ay THEN -- Determine max and min values
          mi := ay;
        END IF;
      END PROCESS;
  END;
```

\textsuperscript{13} The equivalent Verilog code \texttt{mag.v} for this example can be found in Appendix A on page 812. Synthesis results are shown in Appendix B on page 881.
First the input data are stored in registers. Then we take the absolute values of the input data. In the next IF condition the maximum and minimum values are determined and assigned to the variables $ma$ and $mi$. Finally the $\alpha = 1$ and $\beta = 0.25$ approximation is computed and the data are stored in registers. The design runs at $F_{\text{max}}=119.59$ MHz registered performance and uses 96 LEs and no embedded multiplier. Registers are added for the input and output to measure the pipelined performance of the circuit.

The simulation result of the 16-bit pipelined magnitude computation is shown in Fig. 2.62. Eight values with angle 0, 45°, 90°, ... are tested. Note that the computed $r$ values are delayed due to the pipeline register and that the magnitude $r = 1000$ is approximated, but at 45°, i.e., $x = y = 1000 \cos(\pi/2) = 707$ the error becomes substantial.

**Exercises**

**Note:** If you have no prior experience with the Quartus II software, refer to the case study found in Sect. 1.4.3, p. 32. If not otherwise noted use the EP4CE115F29C7 from the Cyclone IV E family for the Quartus II synthesis evaluations.

2.1: Wallace has introduced an alternative scheme for a fast multiplier. The basic building block of this type of multiplier is a carry-save adder (CSA). A CSA takes three $n$-bit operands and produces two $n$-bit outputs. Because there is no propagation of the carry, this type of adder is sometimes called a 3:2 compress or counter.
For an $n \times n$-bit multiplier we need a total of $n - 2$ CSAs to reduce the output to two operands. These operands then have to be added by a (fast) $2n$-bit ripple-carry adder to compute the final result of the multiplier.

(a) The CSA computation can be done in parallel. Determine the minimum number of levels for an $n \times n$-bit multiplier with $n \in [0, 16]$.
(b) Explain why, for FPGAs with fast two’s complement adders, these multipliers are not more attractive than the usual array multiplier.
(c) Explain how a pipelined adder in the final adder stage can be used to implement a faster multiplier. Use the data from Table 2.8 (p. 83) to estimate the necessary LE usage and possible speed for:

(c1) an $8 \times 8$-bit multiplier
(c2) a $12 \times 12$-bit multiplier

2.2: The Booth multiplier used the classical CSD code to reduce the number of necessary add/subtract operations. Starting with the LSB, typically two or three bits (called radix-4 and radix-8 algorithms) are processed in one step. The following table demonstrates possible radix-4 patterns and actions:

<table>
<thead>
<tr>
<th>$x_{k+1}$</th>
<th>$x_k$</th>
<th>$x_{k-1}$</th>
<th>Accumulator activity</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$\text{ACC} \rightarrow \text{ACC} + R^* (0)$</td>
<td>within a string of “0s”</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$\text{ACC} \rightarrow \text{ACC} + R^* (X)$</td>
<td>end of a string of “1s”</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$\text{ACC} \rightarrow \text{ACC} + R^* (X)$</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$\text{ACC} \rightarrow \text{ACC} + R^* (2X)$</td>
<td>end of a string of “1s”</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$\text{ACC} \rightarrow \text{ACC} + R^* (-2X)$</td>
<td>beginning of a string of “1s”</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$\text{ACC} \rightarrow \text{ACC} + R^* (-X)$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$\text{ACC} \rightarrow \text{ACC} + R^* (-X)$</td>
<td>beginning of a string of “1s”</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$\text{ACC} \rightarrow \text{ACC} + R^* (0)$</td>
<td>within a string of “1s”</td>
</tr>
</tbody>
</table>

The hardware requirements for a state machine implementation are an accumulator and a two’s complement shifter.
(a) Let $X$ be a signed 6-bit two’s complement representation of $-10 = 110110_{2C}$. Complete the following table for the Booth product $P = XY = -10Y$ and indicate the accumulator activity in each step.

<table>
<thead>
<tr>
<th>Step</th>
<th>$x_5$</th>
<th>$x_4$</th>
<th>$x_3$</th>
<th>$x_2$</th>
<th>$x_1$</th>
<th>$x_0$</th>
<th>$x_{-1}$</th>
<th>ACC</th>
<th>ACC + Booth rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) Compare the latency of the Booth multiplier, with the serial/parallel multiplier for the radix-4 and radix-8 algorithms.

2.3: (a) Compile the HDL file `add2p` with the Quartus II compiler with optimization for speed and area. How many LEs are needed? Explain the results.
(b) Conduct a simulation with $x = 32760$ and $y = 1,073,740,000,000, y = 5$, and $y = 10$.

2.4: Explain how to modify the HDL design `add1p` for subtraction.
(a) Modify the design and simulate as an example:
(b) $3 - 2$
(c) $2 - 3$
2.5: (a) Develop a 8 × 8 serial/parallel multiplier `mul_ser` in HDL file according to (2.29) (p. 86) with the Quartus II compiler. (b) Determine the registered performance using the TimeQuest slow 85C model and the used resources of the 8-bit design. What is the total multiplication latency?

2.6: Develop HDL design file `mul_ser` to multiply 12 × 12-bit numbers according to (2.29) (p. 86). (a) Simulate the new design with the values 1000 × 2000. (b) Determine the registered performance using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M9Ks). (c) What is the total multiplication latency of the 12 × 12-bit multiplier?

2.7: (a) Design a state machine in Quartus II to implement the Booth multiplier (see Exercise 2.2) for 6 × 6 bit signed inputs. (b) Determine the registered performance using the TimeQuest slow 85C model. (c) Design a state machine in Quartus II to implement the Booth multiplier (see Exercise 2.2) for 6 × 6 bit signed inputs. (d) Determine LE utilization for maximum speed.

2.8: (a) Design a generic CSA that is used to build a Wallace-tree multiplier for an 8 × 8-bit multiplier. (b) Implement the 8 × 8 Wallace tree using Quartus II. (c) Use a final adder to compute the product, and test your multiplier with a multiplication of 100 × 63. (d) Pipeline the Wallace tree. What is the maximum throughput of the pipelined design?

2.9: (a) Use the principle of component instantiation, using the predefined macros `LPM_ADD_SUB` and `LPM_MULT`, to write the VHDL code for a pipelined complex 8-bit multiplier, i.e., \((a + jb)(c + jd) = ac - bd + j(ad + bc)\), with all operands \(a, b, c,\) and \(d\) in 8-bit. (b) Determine the registered performance \(F_{\text{max}}\) using the TimeQuest slow 85C model. (c) Determine LE and embedded multipliers used for maximum speed synthesis. (d) How many pipeline stages does the optimal single `LPM_MULT` multiplier have? (e) How many pipeline stages does the optimal complex multiplier have in total if you use: (e1) LE-based multipliers? (e2) Embedded array multipliers?

2.10: An alternative algorithm for a complex multiplier is:

\[
\begin{align*}
    s[1] &= a - b \\
    s[2] &= c - d \\
    s[3] &= c + d \\
    m[1] &= s[1]d \\
\end{align*}
\]  

(2.98)

which, in general, needs five adders and three multipliers. Verify that if one coefficient, say \(c + jd\) is known, then \(s[2], s[3],\) and \(d\) can be prestored and the algorithm reduces to three adds and three multiplications. Also

(a) Design a pipelined 5/3 complex multiplier using the above algorithm for 8-bit signed inputs. Use the predefined macros `LPM_ADD_SUB` and `LPM_MULT`. (b) Measure the \(F_{\text{max}}\) registered performance using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M9Ks) for maximum speed synthesis. (c) How many pipeline stages does the single `LPM_MULT` multiplier have? (d) How many pipeline stages does the complex multiplier have in total if you use: (d1) LE-based multipliers? (d2) Embedded array multipliers?
2.11: Compile the HDL file \texttt{cordic} with the Quartus II compiler, and
(a) Conduct a simulation (using the ModelSim stimuli file \texttt{cordic.do}) with $x_{\text{in}} = \pm 30$ and $y_{\text{in}} = \pm 55$. Determine the radius factor for all four simulations.
(b) Determine the maximum errors for radius and phase, compared with an unquantized computation.

2.12: Modify the HDL design \texttt{cordic} to implement stages 4 and 5 of the CORDIC pipeline.
(a) Compute the rotation angle, and compile the VHDL code.
(b) Conduct a simulation with values $x_{\text{in}}=\pm 30$ and $y_{\text{in}}=\pm 55$.
(c) What are the maximum errors for radius and phase, compared with the unquantized computation?

2.13: Consider a floating-point representation with a sign bit, $E = 7$-bit exponent width, and $M = 10$ bits for the mantissa (not counting the hidden one).
(a) Compute the bias using (2.24) p. 75.
(b) Determine the (absolute) largest number that can be represented.
(c) Determine the (absolutely measured) smallest number (not including denormals) that can be represented.

2.14: Using the result from Exercise 2.13
(a) Determine the representation of $f_1 = 9.25_{10}$ in this $(1,7,10)$ floating-point format.
(b) Determine the representation of $f_2 = -10.5_{10}$ in this $(1,7,10)$ floating-point format.
(c) Compute $f_1 + f_2$ using floating-point arithmetic.
(d) Compute $f_1 * f_2$ using floating-point arithmetic.
(e) Compute $f_1 / f_2$ using floating-point arithmetic.

2.15: For the IEEE single-precision format (see Table 2.6, p. 79) determine the 32-bit representation of:
(a) $f_1 = -0$.
(b) $f_2 = \infty$.
(c) $f_3 = 9.25_{10}$.
(d) $f_4 = -10.5_{10}$.
(e) $f_5 = 0.1_{10}$.
(f) $f_6 = \pi = 3.14159310$.
(g) $f_7 = \sqrt{3}/2 = 0.866025410$.

2.16: Compile the HDL file \texttt{div_res} from Example 2.18 (p. 96) to divide two numbers.
(a) Simulate the design with the values 234/3.
(b) Simulate the design with the values 234/1.
(c) Simulate the design with the values 234/0. Explain the result.

2.17: Design a nonperforming divider based on the HDL file \texttt{div_res} from Example 2.18 (p. 96).
(a) Simulate the design with the values 234/50 as shown in Fig. 2.22, p. 99.
(b) Measure the registered performance $F_{\text{max}}$ using the TimeQuest slow 85C model, the used resources (LEs, multipliers, and M9Ks) and latency for maximum speed synthesis.

2.18: Design a nonrestoring divider based on the HDL file \texttt{div_res} from Example 2.18 (p. 96).
(a) Simulate the design with the values 234/50 as shown in Fig. 2.23, p. 100.
(b) Measure the registered performance $F_{\text{max}}$ using the TimeQuest slow 85C model, the used resources (LEs, multipliers, and M9Ks) and latency for maximum speed synthesis.

2.19: Shift operations are usually implemented with a barrelshifter, which can be inferred in VHDL via the SLL instruction. Unfortunately, the SLL is not supported for STD_LOGIC in VHDL-1993, but we can design a barrelshifter in many different ways to achieve the same function. We wish to design 12-bit barrelshifters, that have the following entity:

```vhdl
ENTITY lshift IS
    ------> Interface
    GENERIC (W1 : INTEGER := 12; -- data bit width
              W2 : integer := 4); -- ceil(log2(W1));
    PORT (clk : IN STD_LOGIC;
          distance : IN STD_LOGIC_VECTOR (W2-1 DOWNTO 0);
          data : IN STD_LOGIC_VECTOR (W1-1 DOWNTO 0);
          result : OUT STD_LOGIC_VECTOR (W1-1 DOWNTO 0));
END;
```

that should be verified via the simulation shown in Fig. 2.63. Use input and output registers for `data` and `result`, no register for the `distance`. Select one of the following devices:

(I) EP4CE115F29C7 from the Cyclone IV E family
(II) EP2C35F672C6 from the Cyclone II family
(III) EPM7128SLC84-7 from the MAX7000S family

(a1) Use a PROCESS and (sequentially) convert each bit of the distance vector in an equivalent power-of-two constant multiplication. Use `lshift` as the entity name.

(a2) Measure the registered performance $F_{\text{max}}$ using the TimeQuest slow 85C model and the resources (LEs, multipliers, and M4Ks/M9Ks).

(b1) Use a PROCESS and shift (in a loop) the input data always 1 bit only, until the loop counter and `distance` show the same value. Then transfer the shifted data to the output register. Use `lshiftloop` as the entity name.

(b2) Measure the registered performance $F_{\text{max}}$ using the TimeQuest slow 85C model and the resources (LEs, multipliers, and M4Ks/M9Ks).

(c1) Use a PROCESS environment and “demux” with a loop statement the distance vector in an equivalent multiplication factor. Then use a single (array) multiplier to perform the multiplication. Use `lshiftdemux` as the entity name.

(c2) Measure the registered performance $F_{\text{max}}$ using the TimeQuest slow 85C model and the resources (LEs, multipliers, and M4Ks/M9Ks).

(d1) Use a PROCESS environment and convert with a case statement the distance vector to an equivalent multiplication factor. Then use a single (array) multiplier to perform the multiplication. Use `lshiftmul` as the entity name.

(d2) Measure the registered performance $F_{\text{max}}$ using the TimeQuest slow 85C model and the resources (LEs, multipliers, and M4Ks/M9Ks).

(e1) Use the lpm_clshift megafunction to implement the 12-bit barrelshifter. Use `lshiftlpm` as the entity name.

(e2) Measure the registered performance $F_{\text{max}}$ using the TimeQuest slow 85C model and the resources (LEs, multipliers, and M4Ks/M9Ks).

(d) Compare all five barrelshifter designs in terms of registered performance, resources (LEs, multipliers, and M4Ks/M9Ks), and design reuse, i.e., effort to change data width and the use of software other than Quartus II.

2.20: (a) Design the PREP benchmark 3 shown in Fig. 2.64a with the Quartus II software. The design is a small FSM with eight states, eight data input bits $i$, $clk$, $o$. (b) Measure the registered performance $F_{\text{max}}$ using the TimeQuest slow 85C model, the used resources (LEs, multipliers, and M9Ks) and latency for maximum speed synthesis.
Fig. 2.63. Test bench for the barrel shifter from Exercise 2.19

\( \text{rst} \), and an 8-bit data-out signal \( \text{o} \). The next state and output logic is controlled by a positive-edge triggered \( \text{clk} \) and an asynchronous reset \( \text{rst} \), see the simulation in Fig. 2.64c for the function test. The following table shows next state and output assignments,

<table>
<thead>
<tr>
<th>Current state</th>
<th>Next state</th>
<th>( i ) (Hex)</th>
<th>( o ) (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>start</td>
<td>(3c)'</td>
<td>00</td>
</tr>
<tr>
<td>start</td>
<td>sa</td>
<td>3c</td>
<td>82</td>
</tr>
<tr>
<td>sa</td>
<td>sc</td>
<td>2a</td>
<td>40</td>
</tr>
<tr>
<td>sa</td>
<td>sb</td>
<td>1f</td>
<td>20</td>
</tr>
<tr>
<td>sa</td>
<td>(2a)'(1f)'</td>
<td>04</td>
<td></td>
</tr>
<tr>
<td>sb</td>
<td>se</td>
<td>aa</td>
<td>11</td>
</tr>
<tr>
<td>sb</td>
<td>sf</td>
<td>(aa)'</td>
<td>30</td>
</tr>
<tr>
<td>sc</td>
<td>sd</td>
<td>–</td>
<td>08</td>
</tr>
<tr>
<td>sd</td>
<td>sg</td>
<td>–</td>
<td>80</td>
</tr>
<tr>
<td>se</td>
<td>–</td>
<td>–</td>
<td>40</td>
</tr>
<tr>
<td>sf</td>
<td>sg</td>
<td>–</td>
<td>02</td>
</tr>
<tr>
<td>sg</td>
<td>start</td>
<td>–</td>
<td>01</td>
</tr>
</tbody>
</table>

where \( x' \) is the condition not \( x \).

(b) Determine the registered performance \( F_{\text{max}} \) using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M4Ks/M9Ks) for a single copy. Compile the HDL file with the synthesis Optimization Technique set to Speed, Balanced or Area; this can be found in the Analysis & Synthesis Settings section under Settings in the Assignments menu. Which synthesis options are optimal in terms of LE count and registered performance?

Select one of the following devices:

(b1) EP4CE115F29C7 from the Cyclone IV E family
(b2) EP2C35F672C6 from the Cyclone II family
(b3) EPM7128SLC84-7 from the MAX7000S family

(c) Design the multiple instantiation for benchmark 3 as shown in Fig. 2.64b.

(d) Determine the registered performance \( F_{\text{max}} \) using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M4Ks/M9Ks) for the design with the maximum number of instantiations of PREP benchmark 3. Use the optimal synthesis option you found in (b) for the following devices:

(d1) EP4CE115F29C7 from the Cyclone IV E family
(d2) EP2C35F672C6 from the Cyclone II family
(d3) EPM7128SLC84-7 from the MAX7000S family

2.21: (a) Design the PREP benchmark 4 shown in Fig. 2.65a with the Quartus II software. The design is a large FSM with sixteen states, 40 transitions, eight data input bits \( i[0..7] \), \( \text{clk} \), \( \text{rst} \) and 8-bit data-out signal \( o[0..7] \). The next state is controlled by a positive-edge-triggered \( \text{clk} \) and an asynchronous reset \( \text{rst} \), see the
Fig. 2.64. PREP benchmark 3. (a) Single design. (b) Multiple instantiation. (c) Test bench to check function simulation in Fig. 2.65c for a partial function test. The following shows the output decoder table:

<table>
<thead>
<tr>
<th>Current state</th>
<th>o[7..0]</th>
<th>Current state</th>
<th>o[7..0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>0 0 0 0 0 0 0</td>
<td>s1</td>
<td>0 0 0 0 1 1 0</td>
</tr>
<tr>
<td>s2</td>
<td>0 0 0 1 1 0 0 0</td>
<td>s3</td>
<td>0 1 1 0 0 0 0</td>
</tr>
<tr>
<td>s4</td>
<td>1 x x x x x 0</td>
<td>s5</td>
<td>x 1 x x x 0 x</td>
</tr>
<tr>
<td>s6</td>
<td>0 0 0 1 1 1 1 1</td>
<td>s7</td>
<td>0 1 1 1 1 1 1</td>
</tr>
<tr>
<td>s8</td>
<td>0 1 1 1 1 1 1 1</td>
<td>s9</td>
<td>1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>s10</td>
<td>x 1 x 1 x 1 x 1</td>
<td>s11</td>
<td>1 x 1 x 1 x 1</td>
</tr>
<tr>
<td>s12</td>
<td>1 1 1 1 1 0 1</td>
<td>s13</td>
<td>1 1 1 0 1 1 1</td>
</tr>
<tr>
<td>s14</td>
<td>1 1 0 1 1 1 1 1</td>
<td>s15</td>
<td>0 1 1 1 1 1 1</td>
</tr>
</tbody>
</table>

where X is the unknown value. Note that the output values does not have an additional output register as in the PREP 3 benchmark. The next state table is:
where \( ik \) is bit \( k \) of input \( i \), the symbol \(^\prime\) is the not operation, \( \times \) is the Boolean AND operation, \( + \) is the Boolean OR operation, \( \odot \) is the Boolean equivalence operation, and \( \oplus \) is the XOR operation.

(b) Determine the registered performance \( F_{\text{max}} \) using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M4Ks/M9Ks) for a single copy. Compile the HDL file with the synthesis Optimization Technique set to Speed, Balanced or Area; this can be found in the Analysis & Synthesis Settings section under Settings in the Assignments menu. Which synthesis options are optimal in terms of LE count and registered performance?

Select one of the following devices:

(b1) EP4CE115F29C7 from the Cyclone IV E family
(b2) EP2C35F672C6 from the Cyclone II family
(b3) EPM7128SLC84-7 from the MAX7000S family
(c) Design the multiple instantiation for benchmark 4 as shown in Fig. 2.65b.
(d) Determine the registered performance \( F_{\text{max}} \) using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M4Ks/M9Ks) for the design with the maximum number of instantiations of PREP benchmark 4. Use the optimal synthesis option you found in (b) for the following devices:

(d1) EP4CE115F29C7
(d2) EP2C35F672C6
(d3) EPM7128SLC84-7

### Exercises 175

2.22: (a) Design an \( 8 \times 8 \)-bit signed multiplier \( \text{smul8x8} \) using M9Ks memory blocks and the partitioning technique discussed in (2.32), p. 90.

(b) Use a short C or MatLab script to produce the three required MIF files. You need signed/signed, signed/unsigned, and unsigned/unsigned tables. The last entry in the table should be:

(b1) \( 11111111 : 11100001 \); \( 15 * 15 = 225 \) for unsigned/unsigned.
(b2) \( 11111111 : 11110001 \); \( -1 * 15 = -15 \) for signed/unsigned.
(b3) \( 11111111 : 00000001 \); \( -1 * (-1) = 1 \) for signed/signed.
(c) Verify the design with the three data pairs $-128 \times (-128) = 16384; -128 \times 127 = -16256; 127 \times 127 = 16129$.

(d) Measure the registered performance $\text{Fmax}$ using the TimeQuest slow 85C model and determine the resources used.

2.23: (a) Design an 8 $\times$ 8-bit additive half-square (AHSM) multiplier $\text{ahsm8x8}$ as shown in Fig. 2.17, p. 91.

(b) Use a short C or MATLAB script to produce the two required MIF files. You need a 7- and 8-bit D1 encoded square tables. The first entries in the 7-bit table should be:

```
depth= 128; width = 14;
address_radix = bin; data_radix = bin;
content begin
0000000 : 00000000000000; --> (1_d1 * 1_d1)/2 = 0
0000001 : 00000000000010; --> (2_d1 * 2_d1)/2 = 2
0000010 : 00000000000100; --> (3_d1 * 3_d1)/2 = 4
0000011 : 00000000001000; --> (4_d1 * 4_d1)/2 = 8
0000100 : 00000000001100; --> (5_d1 * 5_d1)/2 = 12
...```

(c) Verify the design with the three data pairs $-128 \times (-128) = 16384; -128 \times 127 = -16256; 127 \times 127 = 16129$.

(d) Measure the registered performance $\text{Fmax}$ using the TimeQuest slow 85C model and determine the resources used.

2.24: (a) Design an 8 $\times$ 8-bit differential half-square (DHSM) multiplier $\text{dhsm8x8}$ as shown in Fig. 2.18, p. 92.

(b) Use a short C or MATLAB script to produce the two required MIF files. You need an 8-bit standard square table and a 7-bit D1 encoded square tables. The last entries in the tables should be:

```
(b1) 1111111 : 10000000000000; --> (128_d1 * 128_d1)/2 = 8192 for the 7-bit
```
Exercises 177

D1 table.

**b2** 11111111 : 11111100000000; \( \rightarrow (255 \times 255) / 2 = 32512 \) for the 8-bit half-square table.

(c) Verify the design with the three data pairs \(-128 \times (-128) = 16384; -128 \times 127 = -16256; 127 \times 127 = 16129.\)

(d) Measure the registered performance \( F_{\text{max}} \) using the TimeQuest slow 85C model and determine the resources used.

2.25: (a) Design an 8 \( \times \) 8-bit quarter-square multiplication multiplier \( qsm8x8 \) as shown in Fig. 2.19, p. 93.

(b) Use a short C or MATLAB script to produce the two required MIF files. You need an 8-bit standard quarter square table and an 8-bit D1 encoded quarter square table. The last entries in the tables should be:

\[ (b2) \quad 11111111 : 11111110000000; \rightarrow (256 \times 256) / 4 = 16256 \] for the 8-bit quarter square table.

\[ (b2) \quad 11111111 : 100000000000000; \rightarrow (256 \times 256) / 4 = 16384 \] for the D1 8-bit quarter-square table.

(c) Verify the design with the three data pairs \(-128 \times (-128) = 16384; -128 \times 127 = -16256; 127 \times 127 = 16129.\)

(d) Measure the registered performance \( F_{\text{max}} \) using the TimeQuest slow 85C model and determine the resources used.

2.26: Plot the function approximation and the error function as shown in Fig. 2.50a and b (p. 144) for the arctan function for \( x \in [-1, 1] \) using the following coefficients:

(a) For \( N = 2 \) use \( f(x) = 0.0000 + 0.8704x = (0 + 223x) / 256.\)

(b) For \( N = 4 \) use \( f(x) = 0.0000 + 0.9857x + 0.0000x^2 - 0.2090x^3 = (0 + 252x + 10x^2 - 53x^3) / 256.\)

2.27: Plot the function approximation and the error function as shown, for instance, in Fig. 2.50a and b (p. 144) for the arctan function using the 8-bit precision coefficients, but with increased convergence range and determine the maximum error:

(a) For the arctan(\( x \)) approximation the using coefficients from (2.64), p. 145 with \( x \in [-2, 2] \)

(b) For the sin(\( x \)) approximation using the coefficients from (2.79), p. 151 with \( x \in [0, 2] \)

(c) For the cos(\( x \)) approximation using the coefficients from (2.82), p. 152 with \( x \in [0, 2] \)

(d) For the \( \sqrt{x} \) approximation using the coefficients from (2.94), p. 165 with \( x \in [0, 2] \)

2.28: Plot the function approximation and the error function as shown, for instance, in Fig. 2.54a and b (p. 153) for the \( e^x \) function using the 8-bit precision coefficients, but with increased convergence range and determine the maximum error:

(a) For the \( e^x \) approximation using the coefficients from (2.84), p. 152 with \( x \in [-1, 2] \)

(b) For the \( e^{-x} \) approximation using the coefficients from (2.87), p. 154 with \( x \in [-1, 2] \)

(c) For the \( \ln(1 + x) \) approximation using the coefficients from (2.88), p. 158 with \( x \in [0, 2] \)

(d) For the \( \log_{10}(1 + x) \) approximation using the coefficients from (2.92), p. 159 with \( x \in [0, 2] \)

2.29: Plot the function approximation and the error function as shown in Fig. 2.56a and b (p. 155) for the \( \ln(1 + x) \) function for \( x \in [0, 1] \) using the following coefficients:

(a) For \( N = 2 \) use \( f(x) = 0.0372 + 0.6794x = (10 + 174x) / 256.\)

(b) For \( N = 3 \) use \( f(x) = 0.0044 + 0.9182x - 0.2320x^2 = (1 + 235x - 59x^2) / 256.\)
2.30: Plot the function approximation and the error function as shown in Fig. 2.59a and b (p. 160) for the $\sqrt{x}$ function for $x \in [0.5, 1]$ using the following coefficients:

(a) For $N = 2$ use $f(x) = 0.4238 + 0.5815x = (108 + 149x)/256$.

(b) For $N = 3$ use $f(x) = 0.3171 + 0.8801x - 0.1977x^2 = (81 + 225x - 51x^2)/256$. 
Digital Signal Processing with Field Programmable Gate Arrays
Meyer-Baese, U.
2014, XXIII, 930 p. 459 illus., 11 illus. in color. With CD-ROM., Hardcover
ISBN: 978-3-642-45308-3