Computer Arithmetic

Mircea Vlăduţiu

Computer Arithmetic

Algorithms and Hardware Implementations

Springer

Mircea Vlăduţiu
Faculty of Automation and Computers
"Politehnica" University of Timişoara
Timişoara, Timiş, Romania

ISBN 978-3-642-18314-0 ISBN 978-3-642-18315-7 (eBook)
DOI 10.1007/978-3-642-18315-7
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2012948627

ACM Computing Classification (1998): B.2, B.6, B.3

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Contents

Chapter 1
The Representation of Numbers in Computing Systems

1.1 Information Classification

In computers, information is made up of binary digit sequences organized in words, which are the representation units in Computer Architecture and characterized by their length, let it be n, given in bits (short for "binary digit"). The value of n is established mainly by computer hardware considerations. Although during the evolution of computer systems there have been many attempts to establish a value of n, the values of this parameter have been stabilized to multiples of 8 bits (usually representing one byte). The information stored in words is, according to the tree-like diagram from Fig. 1.1 [Haye98], of two types, instructions and data.

To clearly define the main target of our concerns, we specify from the very beginning that we shall not refer to those instructions which represent the information analyzed and interpreted by that part of a control unit which is frequently referred to as the program control unit [Stal99, Haye98]. Instead, we shall approach the data analysis and processing through that part of a control unit which is referred to as the data processing unit. Mention should also be made that we shall not refer to non-numerical data except for logical operations (presented in Chap. 5), for whose representation there are conventionally accepted dedicated codes, of which we mention ASCII (American Standard Code for Information Interchange), EBCDIC (Extended Binary Coded Decimal Interchange Code) and Unicode Standard [Stal99]. We shall however approach, in more detail, the branch of numbers from Fig. 1.1, in order to refer to their representation, as well as to their processing.

When it comes to choosing a proper number representation for the purpose of using it in a computer, such factors as those given below need to be taken into account:

1. The specification of the type of number to be represented, because, for instance, the so-called formats or codes for integers differ from those for real numbers;
2. The range of values that has to be covered by the representation;

M. Vlăduţiu, *Computer Arithmetic*,
DOI 10.1007/978-3-642-18315-7_1, © Springer-Verlag Berlin Heidelberg 2012

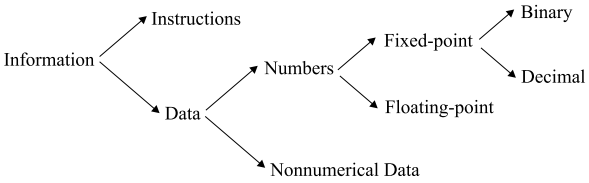


Fig. 1.1 Information taxonomy

3. The precision of the representation consisting of the maximum accuracy that has to be assured by the format or code;
4. The estimation of the hardware complexity required by the representation.

The perspective offered by these factors allows a clear distinction between the representation of fixed point numbers and floating point numbers. Roughly, the former allows for the representation of integers, but also of sub-unitary fractional numbers, covering a limited range of values, with the precision of their representation depending on the number of the word bits. Moreover, it entails a moderate hardware circuitry investment. On the other hand, the latter allows for the representation of real number ranges, and covers a larger range of values, as opposed to the former; this time the precision is given by the number of bits of one part of the representation (the so-called mantissa), with hardware requirements suitably increased if compared to the former [ErLa04].

When, regarding the fixed point format, we cover the problems connected with binary numbers, as well as those of decimal numbers—which are important for certain applications—the references to floating point will be restricted to binary numbers.

1.2 The Representation of Fixed Point Numbers

1.2.1 The Representation of Fixed Point Binary Numbers

It is known that a number can be expressed, in a number system with radix r, by means of a polynomial function of the following form:

$$X = \sum_{i=-m}^{n-1} x_i r^i \tag{1.1}$$

where the coefficients, i.e. the digits, are denoted by x_i, their values being restricted to the range defined by $(0 \leq x_i < r)$.

The number X from (1.1) has n digits in the integer part, and m digits in the fractional one, each of them associated with an (r^i) weight whose value strictly depends on the position of the digit within the number. The digits of the integer part are separated from the fractional part by a point, and those on the left of the point correspond to integer with weights increasing to the left part, starting with

the (r^0) weight for the least significant digit, and with the (r^{n-1}) weight for the most significant digit. Likewise, the digits on the right of the point correspond to the fractional part of the number and have decreasing weights starting with (r^{-1}), for the digit immediately on the right of the point, and (r^{-m}) for the "farthest" digit to the right relative to the point. Writing X in the form given in (1.1) justifies the use of the term positional notation for the number.

For instance, if $r = 10$, by applying (1.1), the decimal, ordinary representation of X will be obtained. Likewise, if $r = 2$, the binary representation of X will be obtained, using only the binary digits (0 and 1). It should be mentioned that the expression from Eq. (1.1), with $r = 2$, does not correspond, in fact, to an actual computer-represented binary number. This is because the position of the point, which is generally variable, requires a binary digit for its specification, which—if it could be omitted—would contribute to increasing the representation precision. This is the reason why, to avoid "sacrificing" accuracy, numbers which are represented in computers are either integers $X = \sum_{i=0}^{n-1} x_i 2^i$, or fractional $X = \sum_{i=-m}^{0} x_i 2^i$. The two categories do not require the specification of the point's position, which is implicitly placed either to the right of the least significant bit (lsb), in case of integers, or to the left of the most significant bit (msb), or between this and the following one, in case of fractional numbers. Regarding the latter, the former corresponds to unsigned fractional numbers, and the latter corresponds to signed fractional numbers. In other words, the representation of the two categories of numbers is accomplished, in positional notation, by admitting the point position as implicit, fixed.

When it comes to signed numbers, conventionally the leftmost bit, msb, corresponding to the sign, will have the value 0 for positive numbers and 1 for negative numbers. This rule holds for both integers and fractions. Thus, the first representation form of fixed point numbers, the so-called sign-magnitude format or code (in short, SM) is obtained. According to this representation, number X appears as a sequence of n bits of the form $X = x_{n-1}x_{n-2}\ldots x_i\ldots x_1x_0$, where the msb x_{n-1} is assigned to the sign, and the other digits have assigned weights whose values decrease by one unit for each position to the right, thus having the 2^{n-2} value for x_{n-2} and 2^0 for x_0 when referring to integers, as well as 2^{-1} for x_{n-2} and 2^{-n+1} for x_0 when referring to fractions. Using the well-known identity $\sum_{i=0}^{n-1} 2^i = 2^n - 1$, we see that the value range for integers in SM is $(0 \leq |X| \leq 2^{n-1} - 1)$, where $|X|$ is the absolute value of X. Likewise, the value range for fractional numbers in SM will be $(0 \leq |X| \leq 1 - 2^{-n+1})$ [ErLa04].

As far as precision is concerned, regarding words of n bits, in SM 2^{n-1} binary numbers can be represented, each of them having the magnitude part specified on $n - 1$ bits at the most. Their correspondents, in the more familiar number system with $r = 10$, and if the logarithmic radix transformation is used, will have at most $\lceil \frac{n-1}{\log_2 10} \rceil$ decimal digits, where the bars $\lceil\ \rceil$ signify the least integer whose value is greater or equal to that calculated for the expression between bars.

The advantages of SM representation of binary numbers consist of their conceptual simplicity, symmetrical value range, simple negation through sign bit inversion, as well as the low cost of the implementations which adhere to this format, as results, for instance, in the synthesis of multiplication devices. However, the SM code has

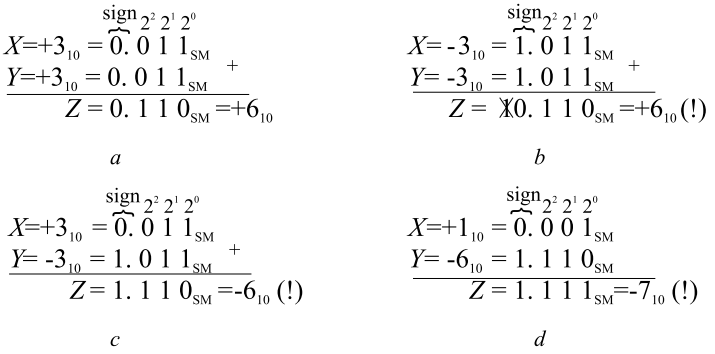


Fig. 1.2 Sign-magnitude addition examples

two important disadvantages, of which the first refers to the addition operation. It is known that this operation is very often resorted to, because most of the algorithms for complex operations are finally reduced to addition. On the other hand, one of the fundamental principles of computer design stipulates using technical solutions that favor those parts of the system which are most often requested ("Make the common case fast") [HePa03] to obtain the best performance possible. The quantitative estimation of the acceleration obtained by applying this principle is determined through the so-called Amdahl's law, according to which, the addition operation should be favored by finding solutions that make its implementation as easy as possible, and implicitly, as efficient as possible. But addition in SM may cause problems because its execution when the operands have the same sign differs from the situation when they have different signs. Thus, for $n = 4$ and integers, Fig. 1.2a presents the case when both operands are positive and there are enough bits for the representation of the result (the special situation of "overflow" has been avoided), when, as it can be observed, the sum is obtained correctly.

The first problem occurs when the operands have the same sign, both of them being negative (Fig. 1.2b). Although addition of the magnitude parts is correctly executed, the sign bit cannot be treated as an ordinary bit, instead requiring to be set up separately on the basis of the previous testing of the numbers' signs. This requires an additional operation, which has to be executed for the other two example additions from Fig. 1.2c and Fig. 1.2d, as well. In the last two cases, where the operands' signs differ, the addition must be substituted by subtraction, which has to be preceded by the comparison of the magnitude parts of the operands. To avoid the occurrence of erroneous results for SM addition, the structure of the addition device must be supplemented with a magnitude comparator, a separate subtraction scheme, and methods for testing the operands' signs. Such a solution will entail not only a supplementary hardware investment, but also a degradation of the operation's execution performance, the latter aspect being more important than the former, in the light of Amdahl's law. The solution based on selective precomplementation and postcomplementation may also be applied [Parh00], but this resorts to additional circuits and delays, as well.

Besides the disadvantage of the difficulties mentioned regarding the addition operation, there is one more disadvantage, i.e. in SM there are two representations for 0 ((+)0, i.e. 0.0 . . . 00 and (−)0, i.e. 1.0 . . . 00). Since testing whether result is equal to 0 is a frequently used operation that comparisons are usually based upon, its implementation should be made as simple as possible. In order to achieve this, there should be only one representation for 0, which is impossible in a binary system with a symmetrical range of values.

The above-mentioned drawbacks of SM representation can be avoided by first introducing the one's complement (C1) format or code. Used in some older computers of the mainframe age, this code maintains its importance through its contribution to forming the two's complement representation, this latter consisting the real competitor for the SM code [Parh00].

The C1 representation of a number X on n bits, is denoted by $\overline{X}$, with its value given by:

$$\overline{X} = \begin{cases} 0.x_{n-2} \dots x_i \dots x_1 x_0 & \text{for } X \geq 0 \\ 1.\overline{x_{n-2}} \dots \overline{x_i} \dots \overline{x_1}\,\overline{x_0} & \text{for } X \leq 0, \quad \text{where } \overline{x_i} = 1 - x_i \end{cases} \tag{1.2}$$

According to (1.2), in C1 the positive numbers have a representation which is identical to that in SM, and as far as the negative numbers are concerned, each binary digit from the SM format will be substituted with its one's complement. Mention should also be made that the arithmetic relation $\overline{x_i} = 1 - x_i$ is equivalent to the logical operation $\overline{x_i} = 1 \oplus x_i$ where $\oplus$ represents the EXCLUSIVE-OR operator or modulo 2 sum (the following pairs being excluded: $(x_i, \overline{x_i}) = (0, 0)$ and $(1, 1)$). Thus, C1 can be formed in a simple way by starting from the SM form, by processing this representation through a level of EXCLUSIVE-OR gates, one gate for each bit of X, all the gates having one common input, to which a 0 is applied (without effect in terms of EXCLUSIVE-OR) when $X > 0$ and a 1 (with complementing effect) when $X < 0$ [Omon94].

The representations in SM and C1 have similar characteristics concerning the range of values and the precision, but they are fundamentally different due to the fact that no positional notation specific to a weighted code corresponds to the negative numbers from C1. However, if the behavior of C1 format is analyzed in comparison with the shortcomings pointed out for the SM representation, regarding the addition of two positive numbers (refer also to Fig. 1.2), there are no differences. Supposing, for instance, that $n = 4$ and we are dealing with integer numbers, Fig. 1.3 presents significant situations which may appear on the addition of the numbers represented in C1. Thus, the case from Fig. 1.3a corresponds to the addition of two numbers of opposite signs, when the negative number is, in its absolute value, greater or equal to the positive one, i.e. without loss of generality, we have $X \leq |Y|$, where $X = X_{SM} = 0.X'_{SM} = X_{C1}$ (in Fig. 1.3a, $X = X_{SM} = 0.011 = X_{C1}$ with $X'_{SM} = 011$) and $Y = Y_{SM} = 1.Y'_{SM}$, and $Y_{C1} = 2^n - 1 - 0.Y'_{SM}$ (in Fig. 1.3a, $Y_{SM} = 1.100$ with $Y'_{SM} = 100$, and $Y_{C1} = 2^4 - 1 - 0.100 = 1.111 - 0.100 = 1.011$) where X_{SM} and Y_{SM} notations have been used for the representations of the two operands in SM, having the magnitude parts X'_{SM} and Y'_{SM} (concatenated at the

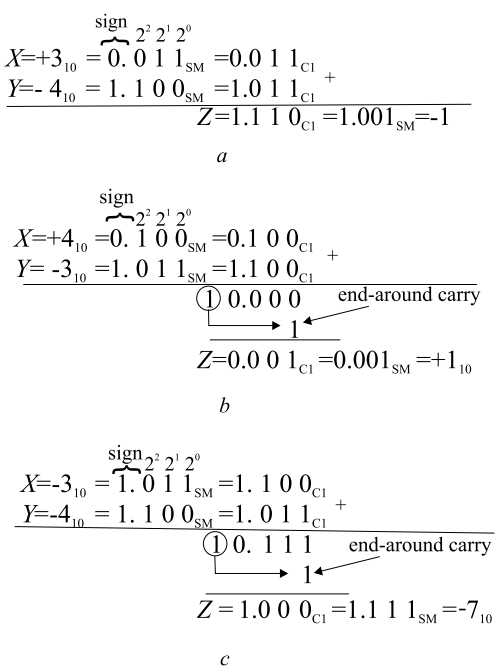


Fig. 1.3 One's complement addition examples

sign bits), and X_{C1} and Y_{C1} for the representations of the two operands in C1. Considering that $X'_{SM} \leq Y'_{SM}$ the sum Z_{C1} results: $Z_{C1} = X_{C1} + Y_{C1} = 0.X'_{SM} + 2^n - 1 - 0.Y'_{SM} = 2^n - 1 - (Y'_{SM} - X'_{SM})$ a value in C1 (no carry is generated from the sign bit. In Fig. 1.3a, $Z_{C1} = 1.111 - (100 - 011) = 1.110$, which is the one's complement of the decimal number (-1). But, if $X > |Y|$ i.e. $X'_{SM} > Y'_{SM}$ as in the example case from Fig. 1.3b ($X = X_{SM} = 0.100 = X'_{C1}$, $X'_{SM} = 100$, $Y = Y_{SM} = 1.011$, $Y'_{SM} = 011$, $Y_{C1} = 2^4 - 1 - 0.011 = 1.111 - 0.011 = 1.100$) the sum Z_{C1} results: $Z_{C1} = X_{C1} + Y_{C1} =$ a value greater than 2^n (since $X'_{SM} > Y'_{SM}$, then $X'_{SM} - Y^1_{SM} \geq 1$), which brings about a carry from the sign bit, forcing it to become 0, consequently the sum will be positive (with the same representation in SM and C1). The correct sum can be obtained by annihilating the effect of the (-1) subtraction by compensating the corrective addition of a binary unit. In Fig. 1.3b, $Z_{C1} = X_{C1} + Y_{C1} = 2^4 - 1 + (100 - 011) = 10.000 - 1 + 1(\text{correction}) + (100 - 011) = 0.000 + 001 = 0.001$, being the representation of the decimal number $(+1)$). The correction in this case consists of the addition of a 1, which is also called end-around carry addition [Haye98], as if the carry from the sign bit comes back being added to the lsb position of the sum. Finally, on the addition of two numbers of opposite signs in one's complement, mention should also be made that overflow never occurs because, supposing $X \geq 0$ and $Y \leq 0$, the whole range of values corresponding to the sum (ranging for integers and $n = 4$ between the minimum value $1.000_{C1} = -7_{10}$ and the maximum value $0.111 = +7_{10}$) can be entirely represented in C1.

Consider now, the case of two negative numbers corresponding to the example from Fig. 1.3c, i.e. when, by generalization, we have $X = X_{SM} = 1.X'_{SM}$ and $X_{C1} = 2^n - 1 - 0.X'_{SM}$ (in Fig. 1.3c, $X_{SM} = 1.011$, with $X'_{SM} = 011$, and $X_{C1} = 1.111 - 0.011 = 1.100$) and $Y = Y_{SM} = 1.Y'_{SM}$ and $Y_{C1} = 2^n - 1 - 0.Y'_{SM}$ (in Fig. 1.3c, $Y_{SM} = 1.100$ with $Y'_{SM} = 100$, $Y_{C1} = 1.111 - 0.100 = 1.011$). If the overflow is avoided ($X'_{SM} + Y'_{SM} \leq 2^{n-1} - 1$), the Z_{C1} sum results: $Z_{C1} = 2^n - 1 - 0.X'_{SM} + 2^n - 1 - 0.Y'_{SM} = 2^n + (2^n - 1 - (X'_{SM} + Y'_{SM}) - 1)$, where the first 2^n value is equivalent to a carry from the sign bit, and the $(X'_{SM} + Y'_{SM})$ sum requires at most $(n-1)$ bits for its representation. The correct result is the one's complemented value $(2^n - 1 - (X'_{SM} + Y'_{SM}))$, which requires the corrective addition of a compensatory binary unit for the subtraction of 1 from the Z_{C1} expression. Since the addition of 1 only occurs when we carry from the $(n-1)$ binary position, the above-mentioned carry can be interpreted as the same with the corrective addition of 1, and it is usually referred to as an end-around carry. Mention should also be made that, by adding the two bits of 1 corresponding to the signs, if carry had not been generated from the $(n-2)$ binary position, the sum would not be negative, which is correct. But this carry is always present, because $(2^n - 1 - (X'_{SM} + Y'_{SM})) \geq (2^n - 1 - (2^{n-1} - 1)) = 2^{n-1}$, being generated either during the preliminary addition, or during that of the end-around carry. If $(X'_{SM} + Y'_{SM}) > 2^{n-1} - 1$ then the conditions for the overflow occurrence are created, and it can be detected by observing the absence of the carry from the $(n-2)$ position before or after the addition of the end-around carry. Actually, at the limit, when $X'_{SM} + Y'_{SM} = 2^{n-1}$ we have $Z_{C1} = 2^n + (2^n - 1 - 2^{n-1} - 1) = 2^n + (2^{n-1} - 1 - 1)$, where, again, the first value 2^n corresponds to the carry from the sign bit, and the expression between parentheses can be represented on $(n-2)$ bits, without bringing about any carry from the $(n-2)$-th binary position, not even after the addition of the end-around carry.

If we refer to the second disadvantage pointed out for the SM representation, i.e. the disadvantage in "0 testing", it should be mentioned that the C1 format also has this drawback, the operation requiring the comparison with two values instead of one, namely with $0.0\ldots00$, for $(+0)$, and with $1.1\ldots11$, for (-0).

Both major deficiencies, pointed out for the SM format, and also present in the C1 representation, can be removed if the two's complement (C2) format or code is resorted to. Regarding its forming, for a number X on n bits, of integer type, if the representation in C2 is denoted by $(-X)$, we have:

$$-X = \begin{cases} 0.x_{n-2}\ldots x_i\ldots x_1x_0 & \text{for } X \geq 0 \\ (1.\overline{x_{n-2}}\ldots\overline{x_i}\ldots\overline{x_1}\,\overline{x_0} + 1) \bmod 2^n & \text{for } X < 0 \end{cases} \tag{1.3}$$

where mod 2^n signifies the sum is executed modulo 2^n.

Similarly, if number X on n bits is of fractional type, then its two's complement is given by:

$$-X = \begin{cases} 0.x_{n-2}\ldots x_i\ldots x_1x_0 & \text{for } X \geq 0 \\ (1.\overline{x_{n-2}}\ldots\overline{x_i}\ldots\overline{x_1}\,\overline{x_0} + 0.0\ldots0\ldots01) \bmod 2^n & \text{for } X < 0 \end{cases} \tag{1.4}$$

where, again, mod 2 signifies the sum is executed modulo 2.

Fig. 1.4 Two's complement addition examples

$$\overset{\text{sign}\ 2^2\ 2^1\ 2^0}{X=+3_{10}=\overbrace{0.}\ 0\ 1\ 1_{SM}=0.\ 0\ 1\ 1_{C1}=0.\ 0\ 1\ 1_{C2}}$$
$$Y=-4_{10}=1.\ 1\ 0\ 0_{SM}=1.\ 0\ 1\ 1_{C1}=1.\ 1\ 0\ 0_{C2}$$
$$Z=1.\ 1\ 1\ 1_{C2}=1.\ 0\ 0\ 1_{SM}=-1_{10}$$

a

$$\overset{\text{sign}\ 2^2\ 2^1\ 2^0}{X=+4_{10}=\overbrace{0.}\ 1\ 0\ 0_{SM}=0.\ 1\ 0\ 0_{C1}=0.\ 1\ 0\ 0_{C2}}$$
$$Y=-3_{10}=1.\ 0\ 1\ 1_{SM}=1.\ 1\ 0\ 0_{C1}=1.\ 1\ 0\ 1_{C2}$$
$$Z=\cancel{1}0.\ 0\ 0\ 1_{C2}=0.\ 0\ 0\ 1_{SM}=+1_{10}$$

b

$$\overset{\text{sign}\ 2^2\ 2^1\ 2^0}{X=-3_{10}=\overbrace{1.}\ 0\ 1\ 1_{SM}=1.\ 1\ 0\ 0_{C1}=1.\ 1\ 0\ 1_{C2}}$$
$$Y=-4_{10}=1.\ 1\ 0\ 0_{SM}=1.\ 0\ 1\ 1_{C1}=1.\ 1\ 0\ 0_{C2}$$
$$Z=\cancel{1}1.\ 0\ 0\ 1_{C2}=1.\ 1\ 1\ 1_{SM}=-7_{10}$$

c

As it can be seen from (1.3) and (1.4), the representation of positive numbers is identical to that from SM and C1. For negative numbers, the representation in C1 is formed first, and then a binary unit is added at its lsb position. During this last operation the incidental carry from the msb position corresponding to the sign bits is ignored, as specified in (1.3) and (1.4), by making the sums modulo 2^n, and modulo 2 respectively. It should be mentioned that there is also a practical rule for the forming of two's complement, according to which the starting point is the SM representation, which is run through from right to left by maintaining all the 0 bits, as well as the first 1 bit encountered, and then the other bits will be substituted by their binary complementary values, i.e. all the 0 bits become 1 bits, and all the 1 bits become 0.

The C2 format has the consequence that a summing device which enables the addition of a 1 to the lsb of C1 is added to the EXCLUSIVE-OR layer of circuits, which enable the generation of one's complement. The conclusion is that forming C2 is more complicated than that of the other two representations, i.e. SM and C1, which is a disadvantage. Another fact that can be mentioned here is that C2 does not represent a weighted code in accordance with the positional notation for the negative numbers.

These aspects which are not favorable for the two's complement are, however, counterbalanced by the easy solving of the problems of addition and testing, justifying the competition between C2 and SM for the implementation of the various procedures regarding numbers. Thus, as concerns addition, the effect of end-around carry from C1 is taken over in advance by the binary unit added to the forming of C2. Figure 1.4 includes the examples from Fig. 1.3, but this time executed in C2. First, considering the case when no end-around carry is generated in C1 and assuming, without loss of generality, that X is positive and Y is negative, with $X \leq |Y|$, and using the previously introduced notations, we have for the two operands X_{C2} and

Y_{C2}, expressed in C2, the following relations: $X = X_{SM} = 0.X'_{SM} = X_{C1} = X_{C2}$ (in Fig. 1.4a, $X = X_{SM} = 0.011 = X_{C1} = X_{C2}$ with $X'_{SM} = 011$) and $Y = Y_{SM} = 1.Y'_{SM}$, and $Y_{C2} = 2^n - 1 - 0.Y'_{SM} + 1 = 2^n - 0.Y'_{SM}$ (in Fig. 1.4a, $Y_{SM} = 1.100$ with $Y'_{SM} = 100$, and $Y_{C2} = 2^4 - 0.100 = 10.000 - 0.100 = 1.100$). Considering that $X'_{SM} \leq Y'_{SM}$, for the sum Z_{C2} there results $Z_{C2} = X_{C2} + Y_{C2} = 0.X'_{SM} + 2^n - 0.Y'_{SM} = 2^n - (Y'_{SM} - X'_{SM})$, i.e. a value expressed in C2 (no carry is generated from the sign bit) (in Fig. 1.4a, $Z_{C2} = 10.000 - (100 - 011) = 1.111$, this representing the two's complement of the decimal number (-1). But, if $X > |Y|$ i.e. $X'_{SM} > Y'_{SM}$ as shown in the example from Fig. 1.4b ($X = X_{SM} = 0.100 = X_{C1} = X_{C2}$, $X'_{SM} = 100$, $Y = Y_{SM} = 1.011$, $Y'_{SM} = 011$, $Y_{C2} = 2^4 - 0.011 = 10.000 - 0.011 = 1.101$), there results $Z_{C2} = X_{C2} + Y_{C2} = 0.X'_{SM} + 2^n - 0.Y'_{SM} = 2^n + (X'_{SM} - Y'_{SM})$, i.e. a value greater than 2^n, and thus 2^n represents a carry from the sign bit, forcing it to become 0, the obtained sum being positive (with the same representation in SM and in C2). Ignoring the carry from the sign bit, sum Z_{C2} is correctly obtained as $0.(X'_{SM} - Y'_{SM})$ (in Fig. 1.4b, $Z_{C2} = X_{C2} + Y_{C2} = 2^4 + (100 - 011) = 10.000 + 0.001 = 0.001$, being the representation of the decimal number $(+1)$). The addition of two numbers of opposite signs in C2 is similar to that in C1, i.e. overflow never occurs, and the entire range of values corresponding to the sum can be represented in C2.

If we now consider the case where both numbers are negative, as in the example from Fig. 1.4c, i.e. when, by generalization, we have $X = X_{SM} = 1.X'_{SM}$ and $X_{C2} = 2^n - 0.X'_{SM}$ (in Fig. 1.4c, $X_{SM} = 1.011$ with $X'_{SM} = 011$ and $X_{C2} = 10.000 - 0.011 = 1.101$) and $Y = Y_{SM} = 1.Y'_{SM}$ and $Y_{C2} = 2^n - 0.Y'_{SM}$ (in Fig. 1.4c, $Y_{SM} = 1.100$, with $Y'_{SM} = 100$ and $Y_{C2} = 10.000 - 0.100 = 1.100$), and if the condition for overflow avoidance ($X'_{SM} + Y'_{SM} \leq 2^{n-1} - 1$) is fulfilled, there results $Z_{C2} = 2^n - 0.X'_{SM} + 2^n - 0.Y'_{SM} = 2^n + (2^n - (X'_{SM} + Y'_{SM}))$ where the first value 2^n is equivalent to a carry from the sign bit, which, if ignored, leads to the correct result in the form of two's complemented value $2^n - (X'_{SM} + Y'_{SM})$. It should also be mentioned that by adding the two 1 bits corresponding to the signs, the sum would not be correctly negative, if there were no carry from the binary position $(n - 2)$. That carry, however, always exists because $(2^n - (X'_{SM} + Y'_{SM})) \geq (2^n - (2^{n-1} - 1)) = 2^{n-1} + 1 \geq 2^{n-1}$ and is absent only when the condition of overflow is fulfilled (namely when $X'_{SM} + Y'_{SM} > 2^{n-1} - 1$), but only if $X'_{SM} + Y'_{SM} > 2^{n-1}$, when $Z_{C2} = 2^n - (X'_{SM} + Y'_{SM}) < 2^{n-1}$. In this context, it has to be pointed out that when $X'_{SM} + Y'_{SM} = 2^{n-1}$, $Z_{C2} = 2^n - 2^{n-1} = 2^{n-1}$ will be obtained, all the bits of the two's complemented sum being 0, except the sign bit which is 1, an exceptional situation due to the asymmetry of two's complement, an issue which will be discussed later.

The above presentations, along with the examples included, show that in addition in C2, the sign bit can be treated like any other ordinary bit of the number, and that the devices for sign testing, magnitudes comparison, and subtraction, or selective pre- and post-complementing required for the correct execution of addition in SM, are not necessary. The consequence of the operation mode in C2 is that subtraction is reduced to addition, this consisting of the addition of the two's complement corresponding to the subtrahend to the minuend. It is therefore possible to implement the two fundamental operations through the same digital device which is adequately

Fig. 1.5 Fixed point encoding examples for decimal numbers

Decimal number	Fixed-point binary codes		
	SM	C1	C2
+7	0111	0111	0111
+6	0110	0110	0110
⋮	⋮	⋮	⋮
+2	0010	0010	0010
+1	0001	0001	0001
(+)0	0000	0000	0000
(-)0	1000	1111	
-1	1001	1110	1111
-2	1010	1101	1110
⋮	⋮	⋮	⋮
-6	1110	1001	1010
-7	1111	1000	1001
-8	-	-	1000

controlled. If these considerations are reviewed with regard to the perspective offered by Amdahl's law, the C2 fixed point representation is more suitable than the other ones, SM and C1.

As far as 0 testing is concerned, it can be observed that the two's complement format determines a unique representation for 0, thus requiring only one comparison. In Fig. 1.5 [Haye98], using $n = 4$ as previously, the fixed point binary codes for some of the decimal digits are given for comparison. The unique representation for 0 in C2 allows the combination of 1 followed to the right by 0s, a number whose code in C2 is the number itself. This asymmetry of the range of values, also known as the two's complement anomaly, presents a certain disadvantage as compared to the other two codes, both of them being symmetrical.

There are also other forms for the two's complement representation which differ from those given by (1.3) and (1.4), and shall be used in the Robertson multiplication procedure presented in Chap. 3. Thus, starting from (1.3), for the integer negative numbers we first have that $X_{C2} = (1.\overline{x_{n-2}} \ldots \overline{x_i} \ldots \overline{x_1}\ \overline{x_0} + 1) \bmod 2^n = 1.x'_{n-2} \ldots x'_i \ldots x'_1 x'_0$ where x'_i $(i = 0, 1, \ldots, n-1)$ is the notation for the values obtained through one's complement and the addition of the binary unit, which is specific for C2 forming. Separating the sign, we arrive at $X_{C2} = 1.0 \ldots 0 \ldots 00 + 0.x'_{n-2} \ldots x'_i \ldots x'_1 x'_0 = (10.0 \ldots 0 \ldots 00 - 1.0 \ldots 0 \ldots 00 + \sum_{i=0}^{n-2} x'_i 2^i) \bmod 2^n = -2^{n-1} + \sum_{i=0}^{n-2} x'_i 2^i$ or at the more general $X_{C2} = -x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x'_i 2^i$. If $x_{n-1} = 0$ the values x'_i are replaced by x_i and the form corresponding to a positive number is obtained, and if $x_{n-1} = 1$, the form of a negative number is obtained, which enables a convenient forming of the product for the Robertson multiplication process, as will be shown below. Let us consider, for instance, the decimal number $X = -89_{10}$ with the forms $X_{SM} = 1.1011001$ and $X_{C2} = 1.0100111$, the latter

value being obtained by using the above relation, through $X_{C2} = (-1)2^7 + (1 \cdot 2^5 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0) = -128 + 39 = -89$. Similarly, starting from (1.4), for fractional numbers, we have $X_{C2} = (1.\overline{x_{n-2}} \ldots \overline{x_i} \ldots \overline{x_1}\ \overline{x_0} + 0.0 \ldots 0 \ldots 01) \bmod 2^n = 1.x'_{n-2} \ldots x'_i \ldots x'_1 x'_0 = 1.0 \ldots 0 \ldots 00 + 0.x'_{n-2} \ldots x'_i \ldots x'_1 x'_0 = (10.0 \ldots 0 \ldots 00 - 1.0 \ldots 0 \ldots 00 + \sum_{i=1}^{n-1} x'_{n-i-1} 2^{-i}) \bmod 2^n = -2^0 + \sum_{i=1}^{n-1} x'_{n-i-1} 2^{-i}$ or the more general $X_{C2} = -x_{n-1}2^0 + \sum_{i=1}^{n-1} x'_{n-i-1} 2^{-i}$.

Below, we shall globally refer to the fixed point formats, highlighting the fact that as a limited number of bits (let it be n) are available for representation, the values can be affected by errors. Thus, an operation involving numbers on n bits frequently produces a result on more than n bits. If, for instance, we multiply two operands on n bits, up to $2n$ bits are necessary for product representation, but, since only n bits are available, the result of the operation will be affected by error. This may happen due to the total omission of some bits, such as the less significant n bits of a product of $2n$ bits. Consequently, a so-called truncation error will be generated [RaCa06]. The effect of this category of errors can be mitigated by resorting to a supplementary operation called rounding, which shall be described in detail in Chap. 5, that transforms this type of error into a roundoff error. Successive calculations affected by truncation or roundoff errors can result in an intolerable accumulation of errors. Therefore, the choice of fixed point formats has to be made so that there is sufficient precision, and, in some cases, when for certain results the representation on several words in the so-called multiple precision arithmetic is allowed, the calculations have to be executed with a higher degree of precision [Parh00].

1.2.2 The Representation of Fixed Point Decimal Numbers

As a rule, usually, the world outside the computer uses decimal arithmetic, as opposed to the computer which employs the binary number system. Consequently, data entry requires a conversion operation from the decimal system into the binary one, while extracting the results requires a reverse conversion, i.e. from the binary system into the decimal system. The conversion operations from the decimal system into the binary system are executed by means of the well-known remainder method for the integer part of the numbers, and through the multiplication method for the fractional part of the numbers, while the conversion from the binary system into the decimal system is made through polynomial evaluations [Parh00]. Although the application of these methods is essentially simple, they require a significant calculation time, and there are certain applications, e.g. from within the banking field, where conversion operations prevail, besides which they are only simple computations. These applications, based on the same Amdahl's law, require particularly simple conversion operations which enable a rapid execution, thus improving the computations as a whole [Omon94].

The above-mentioned requirements are satisfied by those numerical codes whose characteristic is the conversion of each decimal digit separately, in a binary combination, thus enabling the rapid execution of the operation. There exists a great family of such codes, also called decimal codes, but we shall refer only to the decimal

Fig. 1.6 Fixed point decimal codes for decimal digits

Decimal digit	Fixed-point decimal codes		
	BCD	E3	2-out-of-5
0	0000	0011	11000
1	0001	0100	00011
2	0010	0101	00101
3	0011	0110	00110
4	0100	0111	01001
5	0101	1000	01010
6	0110	1001	01100
7	0111	1010	10001
8	1000	1011	10010
9	1001	1100	10100

representations with binary codes (binary-coded decimal, BCD), excess-three (E3) and two-out-of-five, (2-out-of-5), whose binary equivalents of the decimal digits are given in Fig. 1.6 [Haye98]. The name BCD suits the three codes, and, generally, the decimal representations, because each decimal digit is coded in binary. However, this name is usually reserved for that code which assigns a binary tetrad (four binary digits) to each decimal digit, whose bits have various weights associated to them. BCD8421 code, BCD for short, is most often used, whose most significant bit has $2^3 = 8$ weight assigned to it, whose least significant bit has $2^0 = 1$ weight assigned to it. Thus, the conversion from decimal into BCD is immediate, instance, 9374_{10} is converted to $1001\ 0011\ 0111\ 0100_{BCD}$ without resorting to any arithmetic operation, just by consulting the table from Fig. 1.6. As far as BCD code is concerned, it should be noticed that it is a positional code, each bit having a weight of $10^i 2^j$, where the i exponent corresponds to a decimal digit, its numbering being increased towards the left, starting with $i = 0$ for the least significant decimal digit, and the j exponent corresponds to a bit of the tetrad, its numbering being increased towards the left, from 0 to 3. Thus, the weight of the marked bit from the BCD representation of the number 23, 00$\underline{1}$0 0011, is $10^1 2^1 = 20$. Generally, the decimal codes use more bits than the binary code for the conversion of a number. For instance, considering words made up of n bits which enable the binary coding of 2^n unsigned numbers, the above mentioned BCD code allows the coding of only $10^{\frac{n}{4}} \cong 2^{0.83n}$ numbers. Mention should also be made that the BCD numbers operation is usually made in SM format, and addition requires the selective adjustment of some of the tetrads of the result through the so-called "correction of 6", due to the fact that the weight variation between two bits that are positionally adjacent is not always 2, being sometimes (for adjacent bits situated at the interface between two decimal digits) $10^{i+1}2^0/10^i 2^3 = 10/8$, as will be shown in Chap. 2.

As far as E3 decimal code is concerned, as can be observed in Fig. 1.6, it is formed by adding the value of $3_{10} = 0011_2$ to the BCD code for each decimal digit, this representing the so-called excess or bias, which justifies the name of the representation. Thus, the number which has been used above as an example, i.e. 9374_{10},

has its E3 representation given by $1100\ 0110\ 1010\ 0111_{E3}$. If given n words, and using also four bits for the coding of a decimal digit, E3 code allows the representation of only approximately $2^{0.83n}$ unsigned numbers. Although it is not a weighted code, E3 representation has a certain importance because it somewhat simplifies the addition operation as compared to that executed in BCD, as will be seen in Chap. 2.

Finally, the decimal code 2-out-of-5 assigns five bits, of which two have the value 1, and the other three have the value 0, for the coding of each decimal digit. Thus, the same number, i.e 9374_{10}, has the two-out-of-five representation given by 10100 00110 10001 01001. For the same dimension (n) of the word, this code allows the representation of fewer numbers (approximately $2^{0.66}$) as compared to the other two, BCD and E3, and, moreover, it is not a positional, weighted code. These drawbacks are counterbalanced by its ability to allow the detection of a single bit error, also facilitating its location at the level of a binary quintet (five bits) which encodes a decimal digit. Any single bit error changes the sum of the binary units of a quintet, a fact which can be checked and detected, in a simple way, through a tree-like diagram of EXCLUSIVE-OR circuits, by executing the modulo 2 sum corresponding to the five bits. By providing such a tree for every decimal digit and executing an OR operation between the outputs of all trees, the error can be highlighted. In this way, we have a representation belonging to the family of error-detecting and correcting codes, which are of increasing importance within the ever acute requirements for information security.

1.3 The Representation of Floating Point Numbers

The range of values covered by the fixed point numbers is, generally, not sufficient for applications, mainly those which resort to scientific calculations, where very small and very large values are frequently encountered. The positional notation is no longer used for their description, a different notation being necessary, which is known as the scientific notation. The essential requirements to which it answers consist in covering an extended range of real number values, for which the point position is sliding, and in demanding a relatively restricted number of bits for representation. Consequently, for a floating point number X, the scientific notation provides the following form: $X = X_M B^{X_E}$, where X_M represents the so-called mantissa, B represents the so-called base, and X_E represents the so-called exponent.

Out of the three numbers which contribute to the description of X, base B is usually equal to 2 or, more rarely, to a power of 2. Since it is constant, it is not necessary to include it in the format of floating point numbers; instead it is implicitly considered to be built into the circuits. But there still remain the other two numbers, the exponent X_E and the mantissa X_M, both of them being encoded in fixed point and binary representation. Usually, X_E is an integer, and X_M is a fractional number, both of them represented in either SM or C2 formats.

As far as the range covered by X numbers is concerned, it is determined by the values of B and X_E. A floating point format of n bits allows, ignoring the sign, the

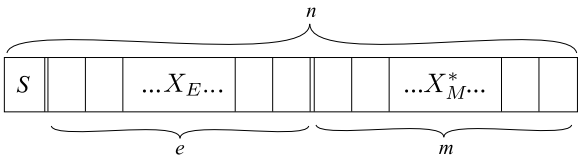


Fig. 1.7 Floating point general format

representation of 2^n numbers which have a certain distribution within the tolerated value field. If B or X_E are increased, the range can be significantly increased, but this implies larger intervals between numbers (a more rare distribution), the total of the potential representations remaining the same. On the other hand, the precision of floating point representation is determined by the number of bits assigned to the mantissa, the larger the number of bits representing X_M, the more accurate the precision, but this would require reducing the number of bits assigned to X_E.

On the basis of what has been presented so far, the word corresponding to a general floating point format can be configured. As shown in Fig. 1.7 [Haye98], it comprises the following fields: the S field, of one bit, which specifies (in the convention described for fixed point) the sign of the mantissa; the X_E field, of e bits, where the exponent number is represented in fixed point; and the X_M^* field, of m bits, where the number which, concatenated with S, forms the mantissa $X_M = S.X_M^*$, is represented in fixed point.

We shall further analyze the peculiarities of the values that can be represented in the fields of the format from Fig. 1.7. Referring first to the number from the X_E field, let us suppose that, during the calculations, we should have obtained an intermediate result with the mantissa X_M equal to 0, but, due to truncation, rounding, or other errors, that result shows a mantissa of some small value, however different from 0. Under these circumstances, if X_E is very large, an adequately amplified error will be obtained instead of the requested 0 value, which is quite different from 0. In order to obtain, within the above mentioned conditions, an ever smaller error, the smallest number which can be represented in the e bits, or, otherwise, the largest negative number will be stored in the exponent field. In SM, in the exponent field, numbers belonging to the closed interval $[(-2^{e-1}+1), (+2^{e-1}-1)]$ can be represented, with two representations for 0, and in C2 numbers belonging to the closed interval $[(-2^{e-1}), (+2^{e-1}-1)]$ can be represented, with a single representation for 0. Consequently, to the X_E field there is assigned the value $(-2^{e-1}+1)$ for the numbers representation in SM, and the value (-2^{e-1}) for the numbers representation in C2. Mention should also be made that when the mantissa becomes 0, the exponent field shall also be 0. Thus, zero will have the same representation in the fixed and floating point formats, the "0 testing" instructions being more easily implemented. This requirement suggests that the floating point exponents should be represented in an excess code of $(+2^{e-1}-1)$, and of $(+2^{e-1})$, an excess which is also called the bias. Thus, in order to favorably solve the problem of representation, the value from the X_E field will consist of the sum of the exponent real value and the bias value, and the number thus obtained is called the biased exponent, or, sometimes, the characteristic. Supposing that $e = 8$, the table from Fig. 1.8 [Haye98] presents several exponent bit patterns with the corresponding unsigned values and signed values, the

Fig. 1.8 Unsigned and signed values for 8 bit exponent patterns

Exponent bit pattern	Unsigned value	Signed value	
		Bias = 127	Bias = 128
11111111	255	+128	+127
11111110	254	+127	+126
⋮	⋮	⋮	⋮
10000001	129	+2	+1
10000000	128	+1	0
01111111	127	0	-1
01111110	126	-1	-2
⋮	⋮	⋮	⋮
00000001	1	-126	-127
00000000	0	-127	-128

latter being presented in two situations, i.e. with the bias equal to $2^{8-1} - 1 = 127$, and with the bias equal to $2^{8-1} = 128$.

Besides the exponent floating point characteristics, there are also certain characteristics of the mantissa. They are determined by the inherent redundancy of the representation, namely there are several forms for one and the same number (thus, $0.110 \cdot 2^3 = 1.100 \cdot 2^2 = 0.011 \cdot 2^4 = \cdots$). Even if the form of the floating point numbers is not restricted during the computer's internal processing, a unique, normal form of representation has been imposed which depends on the SM or C2 fixed point code corresponding to the mantissa, for the data entered into the computer, as well as for the data extraction from computer. Thus, in case of a fractional number in SM, the normalized form requires that the most significant bit of X_M^*, i.e. the bit positioned immediately on the right side of the point (the point is supposed to be between the sign bit and the msb of X_M^*), be 1. Consequently, leading 0s, situated between the 1 bit and the first bit of 1 of X_M^* at the passing through of X_M^* from left to right, are eliminated. Similarly, if the mantissa is a fractional number in C2, the normalized form requires the sign bit to have a different value from that of X_M^*'s msb, through which, for negative numbers, the leading 1s are eliminated. We also mention that the normalized forms are obtained through shift operations to the left and through the corresponding decrease of the exponent, or through shift operations to the right and the corresponding increase of the exponent. The shift of the mantissa to the left by one position is equivalent to the multiplication of its value by 2, thus a unit shall be subtracted from the value of the exponent so as not to modify the number's value. Similarly, the shift of the mantissa to the right by one position is equivalent to the division of its value by 2, thus a unit shall be added to the value of the exponent not so as to modify the number's value. The mantissa in its normalized form is also called "a packed mantissa", so that when entering and extracting data the normalized, packed forms are required, while, during computation, numbers can be processed in their unnormalized, unpacked form. Mention should

also be made that, through the normalization operation, the range of values for the mantissa representation is restricted to $((+1/2) \leq |X_M| < (+1))$.

To be more precise, we shall further refer to the standard format IEEE 754 [Kaha97] meant for the representation of floating point numbers, which is a set of conventions that appeared at a certain moment when the variety of the "rules" applied by various software producers for the floating point number representation rendered difficult, even impossible, software portability between computers. At that moment, the IEEE organization (Institute of Electrical and Electronics Engineers) financed the creation of a standard, conventions which are nowadays considered by almost all the computer manufacturers. The IEEE 754 standard refers to three floating point formats, on 32 bits, on 64 bits, and on 80 bits. We shall further refer to the first two formats [BrO'H03, Kuli02].

According to the IEEE 754 standard on 32 bits, the farthest bit to the left is allocated to the sign (S from Fig. 1.7), being followed, to the right, by the exponent field on 8 bits, where there are represented excess-127 binary integers (refer also to Fig. 1.8), and the remaining 23 bits are assigned to the mantissa. A specification is required as regards Fig. 1.7, namely, in this case, the mantissa represents a fractional part of the sign-magnitude binary significand with a hidden integer bit. First of all, let us explain that the hidden bit has been resorted to because the normalized numbers in SM have in the msb position of X_M^* (Fig. 1.7) a 1, and, consequently, this bit is not necessary to explicitly represent in its packed form, this bit being considered implicitly (wherefrom the name "hidden" given to it), like the value 2 for the base B. The significand field is extended with this bit, and, thus, the precision of the floating point numbers representation increases.

Having made these specifications, and according to the IEEE 754 standard on 32 bits, a number X is given by:

$$X = (-1)^S 2^{X_E - 127}(1.X_M) \tag{1.5}$$

where the restriction $0 < X_E < 255$ is required, the following notations having been used: S for the sign bit, X_E for the biased exponent with 127 bias, and X_M for the mantissa, and $(1.X_M)$ represents the significand number.

The limited range of values due to normalization $((+1/2) \leq |X_M| < (+1))$ refers now to the significand, covering, due to the hidden bit, the range $((+1) \leq |1.X_M| < (+2))$. Let us further evaluate the limits of the variation field within which values for floating point numbers in compliance with the IEEE 754 standard are tolerated. Thus, Fig. 1.9 presents the number line with four number values on it, denoted from X_1 to X_4, and the value 0. Numbers smaller than X_1 or greater than X_4 cannot be represented, since they exceed the available capacity in the negative direction (negative overflow), and in the positive direction (positive overflow) respectively. On the other hand, the numbers ranging between X_2 and 0, and those ranging between 0 and X_3 cannot be represented either, as per (1.5), because the available capacity is underexceeded in the negative direction (negative underflow), and in the positive direction (positive underflow) respectively. However, some numbers whose normalized representations lie in the underflow zones may have denormalized representations. In determining the values corresponding to the intervals' ends, one can

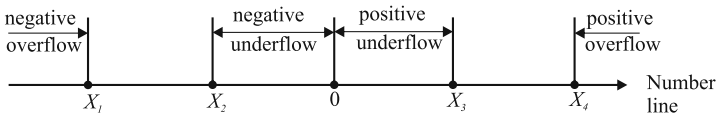


Fig. 1.9 Overflow and underflow regions

first observe the symmetry of the positioning with respect to 0, on the number line. We have $|-X_1| = X_4$ and $|-X_2| = X_3$, the evaluation of only two numbers being required. Regarding X_1, we have $S = 1$ (being in the negative part), $X_E = 254$ (the largest value tolerated for the exponent) and X_M represented by 23 bits of 1 (the largest representable value). At the unpacking of the above-described format, there also appears the hidden bit, so that, by taking it into account (1.5), the following can be written:

$$X_1 = (-1)^1 2^{254-127}\left(1 + 2^{-1} + 2^{-2} + \cdots + 2^{-23}\right) = -2^{127}\left(2 - 2^{-23}\right) \tag{1.6}$$

Converting the value obtained through (1.6) into the more familiar power system of 10, there results approximately $(-3.4 \cdot 10^{38})$, much larger than that corresponding to the end of the representation zone of the SM format on 32 bits, because we have $-2^{31} + 1 \cong 2.15 \cdot 10^9$ [Haye98] for it.

On the other hand, as regards X_2, we have $S = 1$, $X_E = 1$ (the smallest value tolerated for the exponent) and X_M represented by 23 bits of 0 (the smallest representable value). At the format unpacking, there also appears, as mentioned above, the hidden bit, so that, by taking into account (1.5), the following can be written:

$$X_2 = (-1)^1 2^{1-127}(1 + 0 + 0 + \cdots + 0) = -2^{-126} \tag{1.7}$$

Converting the value obtained through (1.7) into the more familiar power system of 10, there results approximately $(-1.18 \cdot 10^{-38})$.

Passing on to the format on 64 bits, its configuration provides the farthest bit to the left for the sign, followed by an exponent field on 11 bits, where excess-1023 binary integers are represented and the remaining 52 bits are allotted for the mantissa. A number X in the IEEE 754 standard format on 64 bits is given by the following:

$$X = (-1)^S 2^{X_E - 1023}(1.X_M) \tag{1.8}$$

where the restriction $0 < X_E < 2047$ is required, the same notations from (1.5) having been used, but for the X_E biased exponent, the excess value is now 1023.

Regarding the IEEE 754 standard, to which we return in more details in Chap. 5, mention should also be made that it enables signalling of four exception situations, for whose coding the exponent limit values are used, values which are not used for ordinary numbers, i.e. 0 and 255 for the format on 32 bits, and 0 and 2047 for the format on 64 bits. Thus, when the intermediate or final results are not valid floating point numbers—such as, for instance, the result of a division by 0—we have the exception situation called Not a Number (NaN). At the occurrence of such an anomaly,

the exponent field is set up at the value of 255 for the format on 32 bits, and at the value of 2047 for the format on 64 bits, and the mantissa field can have any value different from 0. However, if one of the results belongs to one of the overflow zones (Fig. 1.9), then the exponent field is set at the same values as above, but the field of the mantissa is brought to 0. The third exception corresponds to the case when a result belongs to one of the underflow zones (Fig. 1.9). In this case, the exponent field is set to 0 for both formats, and the mantissa field can take any value different from 0. The result is coded through the so-called denormalized form [ScST05], whose characteristic is the reduction of the underflow effect (gradual underflow) through a systematic loss of precision [HePa03]. A number X in denormalized form on 32 bits is given by:

$$X = (-1)^S 2^{-126}(0.X_M) \tag{1.9}$$

while the denormalized form, representation on 64 bits is given by:

$$X = (-1)^S 2^{-1022}(0.X_M) \tag{1.10}$$

In both relations, the bias, equal to 127 and to 1023 respectively, is subtracted from the minimum value (1) of the biased exponent. The loss of precision can be observed through the fact that the previous hidden bit 1 becomes 0. The last exception corresponds to the case when one of the results is 0, when both fields, that of the exponent and that of the mantissa, are set to 0, but the sign bit may also be 1.

To have a comparison reference for the IEEE 754 standard, we refer to the IBM floating point formats used in mainframe computers (S/360 and S/370). There are three formats—on 32, 64 and 128 bits—whose farthest bit to the left is allocated for the sign of the mantissa, this bit being succeeded by an exponent field on 7 bits which has the same dimension for all three formats, and where excess-64 integers can be represented. These formats differ in the number of bits left for the mantissa (24, 56 and 112, the last of them having a subfield of 8 bits which is not used), whose field is allotted (together with the sign bit) for representing fractional numbers in SM. However, there is a major difference from the IEEE 754 standard, namely that base B is equal to 16, and the mantissa number X_M is interpreted in the radix $r = 16$ number system, in which its normalization is made, as well.

Consequently, a number X represented in IBM formats, regardless of the number of bits, is given by:

$$X = (-1)^S 16^{X_E - 64}(0.X_M) \tag{1.11}$$

The formats differ through the number of hexadecimal digits (6, 14, and 28) of the X_M mantissa. The IBM representation rules do not cover the NaN, overflow, and denormalization exceptions, having only one correspondent for the representation of 0. Due to the large value of the base, the covered range is much larger than that of the IEEE 754 standard, having, for the format on 32 bits, the limiting values $X_3 \cong 5.4 \cdot 10^{-79}$ and $X_4 \cong 7.24 \cdot 10^{-75}$ for the positive variation interval (refer to Fig. 1.9) [Haye98].

Let us finally make, for instance, the hexadecimal signs sequences which correspond to the decimal number $X = -724.40625$, in compliance, on the one hand

with the IEEE 754 standard, and, on the other hand, with the IBM rules, both of them for formats on 32 bits. First of all, the number will be converted into binary by applying the residue method for the integer part of X, and the multiplicative method for its fractional part, with the result $X = -1011010100.01101$. In order to obtain the representation in IEEE 754 standard, X is brought into the form given by (1.5), with the result that $X = (-1)^1 \cdot 2^9 \cdot 1.01101010001101$. For packing, the hidden bit will be omitted, and the value $(9 + 127 = 136)$ results for the biased exponent. Consequently, under the IEEE 754 standard we have the following binary sequence: $1.10001000011010100011010 \ldots 0$, which, if converted into the hexadecimal system, leads to $X = \text{C4351A00}$. On the other hand, the representation in compliance with IBM rules can be obtained by bringing X into the form given by (1.11), through which the following will result: $X = (-1)^1 \cdot 16^3 \cdot 0.00101101010001101$. For packing, the value $(3 + 64 = 67)$ results for the biased exponent, so that the following binary sequence will be obtained: $1.10000110010110101000110 10 \ldots$, which, if converted into the hexadecimal system, leads to $X = \text{C32D4680}$.

Chapter 2
Functional Analysis and Synthesis of Binary and Decimal Adding and Subtracting Devices

2.1 Serial Adders

If we have two operands which consist of two binary vectors, they can be passed to an adding/subtracting device either bit by bit, in serial manner, or with all the bits at the same time, in parallel manner. Otherwise, the device inputs are supplied, under the control of a CLOCK pulse train, either with a pair of bits at every pulse, i.e. with one bit from each of the operand vectors, or with all the bits of both vectors at every pulse. First of all, we shall refer to the serial operating mode for binary addition. The device which executes this operation will be called a serial adder, to distinguish it from the concurrent technical solution based on the parallel operation which is called a parallel adder. At a rough analysis, the serial adder has a great disadvantage, as far as its performance is concerned, because it requires for addition, when the two operand vectors have n bits, a time interval consisting of n periods of the CLOCK train, while its parallel alternative requires the interval of only one CLOCK period. Even if this aspect can be attenuated to a certain extent, through the superposed execution of the operations, which are normally executed successively, the parallel variant is favored, but not decisively, because, besides the cost factor, which favors the serial solution, there have to be taken into account implementation aspects, such as reducing the number of interconnections for signals transmission and simplifying the interfaces between the devices, which results in saving integrated circuit area, and in reducing the dissipated energy [ErLa04]. Consequently, the serial arithmetic operation, in general, and addition, in particular, becomes an attractive solution for those applications which tolerate an increased latency. Within the same dispute, "serial versus parallel" we shall refer only to the serial version in this section, but hybrid solutions, in which some inputs and outputs are serial and others are parallel, are also of interest.

Typically, there are two serial operation modes, depending on the first pair of digits, namely [ErLa04]:

(a) The "least-significant digit first" mode (LSDF), characterized by the fact that addition begins with the least significant pair of bits, it being implied when "serial arithmetic" syntagma is used, because it was the first used.

M. Vlăduţiu, *Computer Arithmetic*,
DOI 10.1007/978-3-642-18315-7_2, © Springer-Verlag Berlin Heidelberg 2012

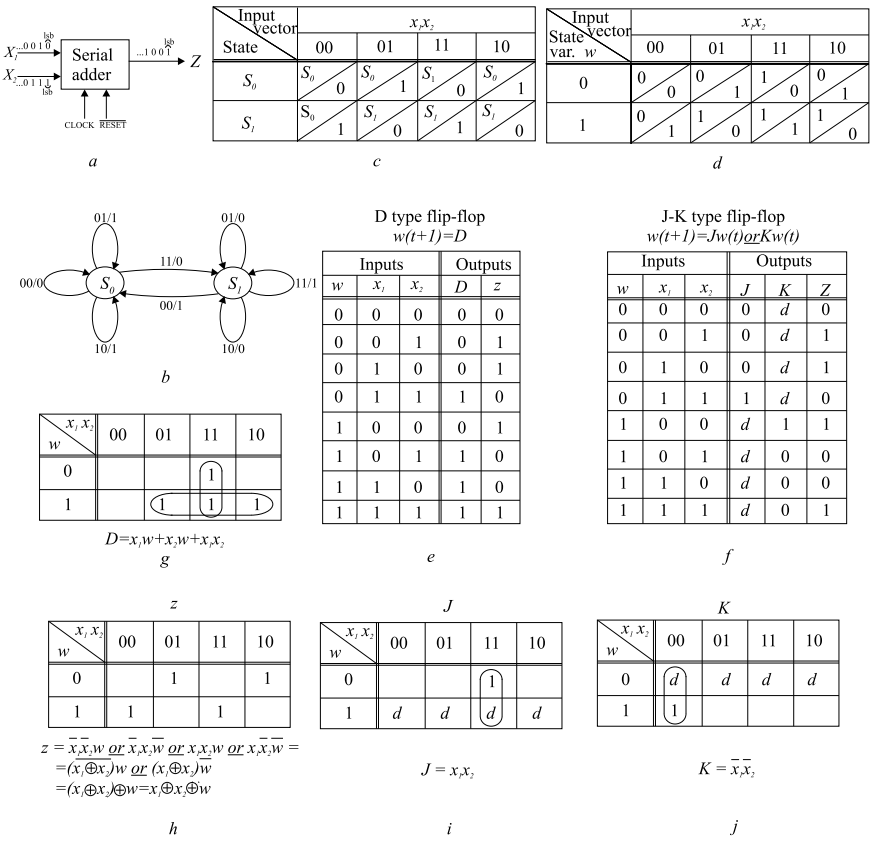


Fig. 2.1 Sequence of design steps for a serial adder

(b) The "most-significant digit first" mode (MSDF), characterized by the fact that, first, the most significant pair of bits is addressed, this arithmetic being known as "online arithmetic" [ErLa04]. The MSDF mode uses redundancy in the number representation system, based on flexibility in the output digit estimation, which requires only partial information about inputs. This enables several forms for a certain value, the best-known representation systems being the signed-digits system and the carry save system, which will be presented and used in the next chapter.

In order to get familiarized with the serial operation, we shall present the synthesis of a simple LSDF adder, which enables the addition of two binary unsigned numbers, represented on n bits, which have the format $X = (x_{n-1}, x_{n-2}, \ldots, x_1, x_0)$ and $Y = (y_{n-1}, y_{n-2}, \ldots, y_1, y_0)$. It is assumed that each of them is stored in a shift register, and the sum result $Z = (z_{n-1}, z_{n-2}, \ldots, z_1, z_0)$ is stored in a shift register, as well. Figure 2.1 presents such a serial adder. Starting with the least significant bit (lsb), a pair of bits will be supplied, one belonging to operand X and the other to

operand Y, at each CLOCK pulse, and the sum of the two bits is calculated taking into account the potential carry generated in the preceding time quantum at the application of the previous CLOCK pulse (Fig. 2.1a). In other words, the serial adder appears as a sequential circuit which has to memorize the carry generated at the addition of the previous pair of bits, and thus has to be able to enter into two distinct internal states, one which will be denoted S_0, where no carry is generated, and the other one which will be denoted S_1, where a carry is generated. The graph consisting of the state diagram associated with the sequential circuit represented by the serial adder is given in Fig. 2.1b, by using a Mealy states notation [Wake00, Yarb97], and Fig. 2.1c presents the corresponding state table. It can be observed that a node is associated with each internal state of the graph, and pairs of vectors of xy/z type are associated with the arcs which represent the transitions between the states, where x, y, z represent the Boolean variables assigned to the X and Y input operands, and to the output result Z respectively. Thus, for instance if the serial adder is supposed to be in the current internal state S_0, and $xy = 11$ vector is applied to its inputs, then the first CLOCK pulse determines the transition to the next internal state S_1, and the vector, of a single element, $z = 0$ will be generated at the output. Correspondingly, in the state table there is assigned a line for each current internal state of the serial adder, and a column for each input vector xy. At the intersection of a line with a column, there are two elements, the next internal state, into which the sequential circuit passes when a CLOCK pulse is applied to it, separated by "/" from the vector supplied at the circuit's observable output. In fact, the state table represents another form of the functional behavior description which has been represented in the state diagram. In a successive design stage, the state variables are assigned to the internal states, symbolized in an abstract way [Yarb97]. In the case of the serial adder, when there are only two internal states, i.e. S_0 and S_1, a single state variable, noted w, is sufficient. The coding of this variable associates the value 0 with S_0, and the value 1 with S_1; there results the so-called transition table [Yarb97] from Fig. 2.1d.

The synthesis goes on by choosing the storage element, and we have chosen, for comparison reasons, two distinct technical solutions, namely the D type flip-flop, and the J-K type flip-flop [Wake00, Yarb97]. With each of these flip-flop types is associated a characteristic equation, which expresses the state of the storage element after the active front of the CLOCK has been applied, denoted, in our case, by $w(t+1)$, as a function of the state before the active front of the CLOCK has been applied, denoted, in our case, by $w(t)$, and the logical values applied to the so-called synchronous inputs D, and J and K respectively. Thus, for the D type flip-flop we have the characteristic equation $w(t+1) = D$ and for the J-K flip-flop we have the characteristic equation $w(t+1) = J\overline{w(t)}$ or $\overline{K}w(t)$. Under these circumstances, starting from the transition table, and taking into account the characteristic equations that are specific to each flip-flop, the so-called excitation tables [Yarb97] for the two solutions result, i.e. with the D flip-flop (Fig. 2.1e), and with the J-K flip-flop (Fig. 2.1f), where d stands for the don't care logical value.

Following the serial adder design process, from the excitation table the excitation equations can be deduced for each flip-flop type, and also the output equations [Yarb97]. In the tables from Fig. 2.1e, and Fig. 2.1f, we can identify the minterms

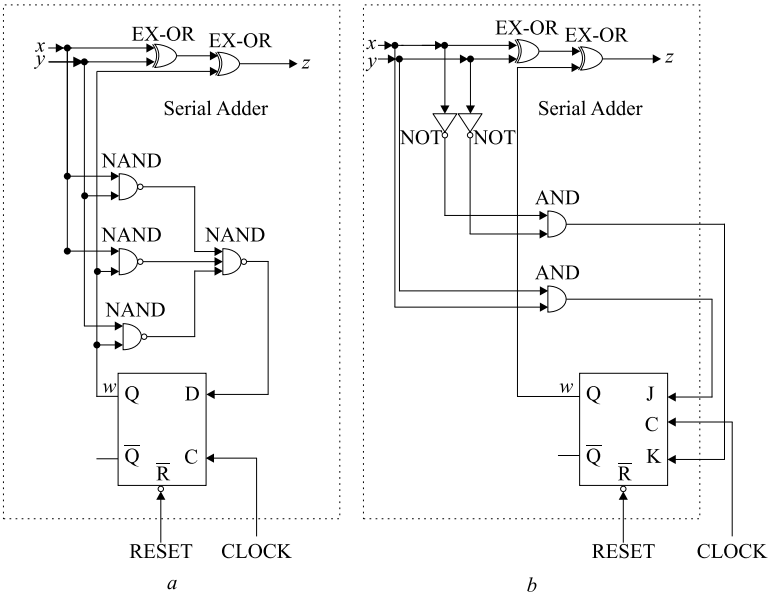


Fig. 2.2 Gate level implementation versions for a serial adder

(those product terms which contain each input variable only once) which determine the logical value 1 for a certain Boolean function in the canonical sum of products form [Wake00, Yarb97]. In order to obtain the excitation and output logical equations in minimized form, the number of inputs and state variables being reduced, we appeal to Karnaugh maps (where a square corresponds to a certain minterm). Thus, if we follow up the maximization of the logically adjacent minterm groups, the favorable covering of the binary units leads to the minimized Boolean expressions for the synchronous inputs D (Fig. 2.1g), under the form $D = wx \underline{\ or\ } xy \underline{\ or\ } yz$, and J (Fig. 2.1i), under the form $J = xy$, and K (Fig. 2.1j), under the form $K = \overline{x}\,\overline{y}$ (certain don't care minterms have also been used to form the prime implicants from the corresponding equations). Regarding the output function z, obviously identical for the two solutions of serial adder, it should be mentioned that all its minterms form essential prime implicants, so that the Boolean equation for z (Fig. 2.1h) is $z = \overline{x}w\overline{y} \underline{\ or\ } \overline{w}y\overline{x} \underline{\ or\ } \overline{w}x\overline{y} \underline{\ or\ } wxy$, which, by using properties of the EXCLUSIVE-OR operator, can be rewritten in the form $z = w \oplus x \oplus y$. The implementations of the combinational logic parts corresponding to the previously deduced Boolean excitation and output equations are presented, by using NAND and EXCLUSIVE-OR gates, and inverter (NOT), AND and EXCLUSIVE-OR gates, in Fig. 2.2a for the serial adder with D flip-flop, and in Fig. 2.2b for one with J-K flip-flop. Both versions, which do not differ essentially in terms of the elementary circuits used for synthesis, or the number of connections, form simple sequential diagrams which function in a synchronized way by means of a CLOCK train (they are initialized by activating the $\overline{RESET}$ line). Even if, due to the fact that the signals have to cross a reduced number

of logic levels, the CLOCK frequency can be high, the maximum adding time—the parameter used to judge the performance of an adder—that results through the cumulation of the n pulse periods, becomes, however, prohibitive for practical values of n. Consequently, those applications for which performance is important require a parallel adder solution.

2.2 Parallel Adders and Subtracters

2.2.1 Binary Adders Based on Serial Carry Propagation

Unlike serial adders, parallel adders generally require one CLOCK cycle (period) for addition. The simplest, but also the slowest adder of this type consists of the connection of n (number of operands bits) so-called full adder cells (FAC), through which the carry propagates from FAC to FAC, in a serial mode, wherefrom the name ripple carry adder (RCA). Thus, Fig. 2.3 presents the block diagram of such a device to which, as inputs, the operand vectors $X = (x_{n-1}, x_{n-2}, \ldots, x_i, \ldots, x_1, x_0)$ and $Y = (y_{n-1}, y_{n-2}, \ldots, y_i, \ldots, y_1, y_0)$ and, eventually, the input carry $c_{in} = c_0$, are supplied, and which supply, after a CLOCK cycle, the result vector $Z = (z_{n-1}, z_{n-2}, \ldots, z_i, \ldots, z_1, z_0)$ and, eventually, the output carry $c_{out} = c_n$. During the computations the carry vector $C = (c_n, c_{n-1}, \ldots, c_{i+1}, c_i, \ldots, c_2, c_1)$ is generated whose carry bits propagate from right to left, in a serial mode, and the CLOCK period has to cover, in time, the carry crossing the most unfavorable (the longest) chain of logic levels. Regarding FACs, their synthesis is based on the already known Boolean equations, which, according to the i rank of the RCA (Fig. 2.3), have the expressions $z_i = \overline{x}_i \overline{y_i} c_i \ \underline{or}\ \overline{x_i} y_i \overline{c_i} \ \underline{or}\ x_i \overline{y_i} \overline{c_i} \ \underline{or}\ x_i y_i c_i = x_i \oplus y_i \oplus c_i$ (the last form being also called the odd parity function) for the sum output, and $c_{i+1} = x_i y_i \ \underline{\text{or}}\ y_i c_i \ \underline{\text{or}}\ c_i x_i$ (also called the majority function [Parh00]) for the carry output to the next rank. Starting from these expressions, the implementation can be done in several ways, depending on the available elementary circuits [Wake00]. Thus, Fig. 2.4a presents the direct transposition version of the equations by using the smallest number of logic levels with inverter and NAND gates. Alternatively, starting from the carry equation as a function of the minterms, $c_{i+1} = \overline{x_i} y_i c_i \ \underline{\text{or}}\ x_i \overline{y_i} c_i \ \underline{\text{or}}\ x_i y_i \overline{c_i} \ \underline{\text{or}}\ x_i y_i c_i$, after some simple processing the $c_{i+1} = (\overline{x_i} y_i \ \underline{or}\ x_i \overline{y_i}) c_i \ \underline{\text{or}}\ x_i y_i (\overline{c_i} \ \underline{or}\ c_i) = (x_i \oplus y_i) c_i \ \underline{\text{or}}\ x_i y_i$ form will be obtained. On this basis, the implementations with EXCLUSIVE-OR and NAND gates (Fig. 2.4b), and with EXCLUSIVE-OR, AND and OR gates (Fig. 2.4c) respectively, are achieved, both of them being multilevel, and taking into account the EXCLUSIVE-OR gate involvement [ErLa04]. Figure 2.4d presents the implementation with seven inverters and two 4 to 1 multiplexers, which is suited for CMOS technology with transmission gates [Parh00]. The Boolean equations which stand at the basis of the synthesis are equivalent forms of the initial ones, namely, $\overline{z_i} = \overline{x_i}\ \overline{y_i}\ \overline{c_i} \ \underline{\text{or}}\ x_i y_i \overline{c_i} \ \underline{\text{or}}\ x_i \overline{y_i} c_i \ \underline{or}\ \overline{x_i} y_i c_i$, and $c_{i+1} = x_i \overline{y_i} c_i \ \underline{\text{or}}\ \overline{x_i} y_i c_i \ \underline{\text{or}}\ x_i y_i$. Obviously, to the versions from Fig. 2.4 [Parh00] the implementation under the form given by the combinational part of the serial adder from Fig. 2.2a will be added.

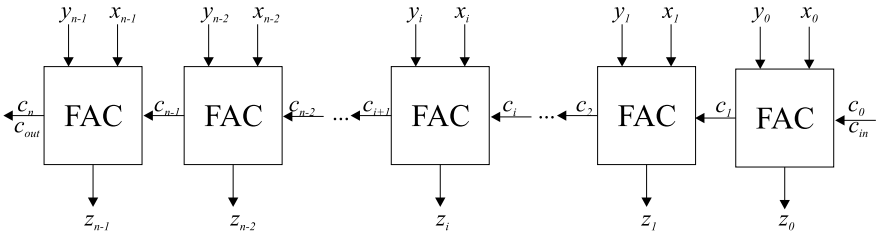


Fig. 2.3 Block diagram of a ripple carry adder

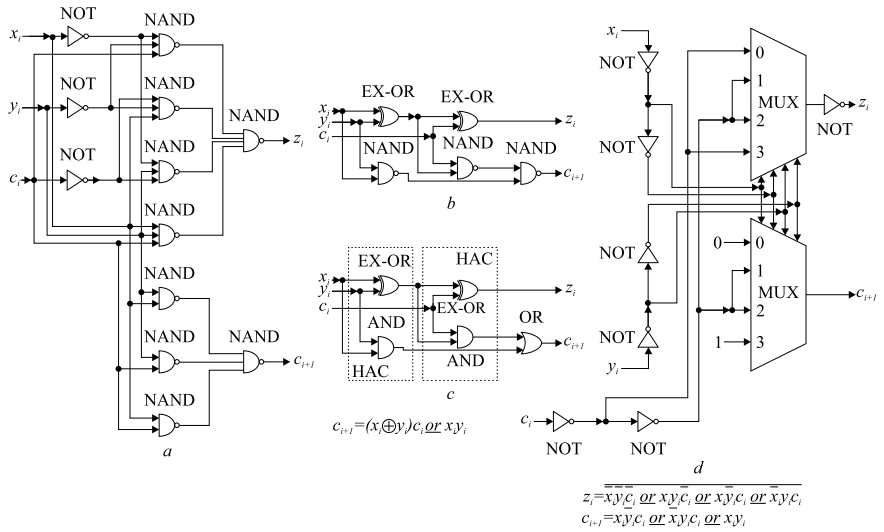


Fig. 2.4 Gate level implementation versions for a full adder cell

A certain reduction of the total delay on the critical path of the carry propagation, consisting of the chaining of all the ranks, can be obtained by substituting the FAC corresponding to the lsb (the farthest to the right, Fig. 2.3) with one so-called half-adder cell (HAC). This may happen only when the c_{in} input is not used (in most cases), namely when $c_{in} = c_0 = 0$, a situation in which the FAC-specific equations can be simplified, becoming $z_0 = \overline{x_0} y_0 \ \underline{\text{or}} \ x_0 \overline{y_0} = x_0 \oplus y_0$ for the sum output, and $c_1 = x_0 y_0$ for the carry output to the second rank. On the basis of these Boolean equations, Fig. 2.5 [Parh00] presents some possible implementations for a single HAC, using EXCLUSIVE-OR and AND gates (Fig. 2.5a), inverter and NOR gates (Fig. 2.5b), and inverter and NAND gates (Fig. 2.5c) [Parh00].

Let us mention that by means of FACs and HACs a variety of arithmetic functions can be achieved [Parh00]. Thus, a serial adder synthesized with D flip-flops represents an example of a FAC attachment to a storage element (Fig. 2.2a). FACs and HACs can be used in a multitude of chips, integrated on medium and large

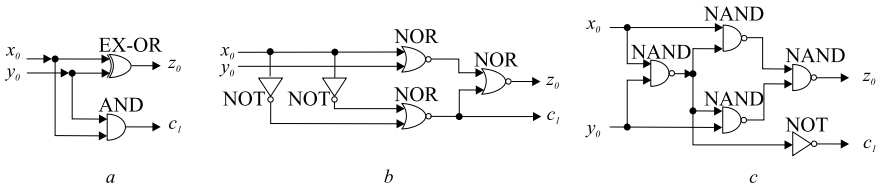


Fig. 2.5 Gate level implementation versions for a half adder cell

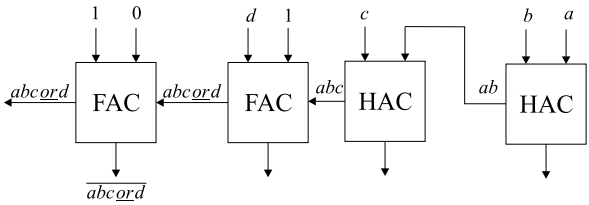


Fig. 2.6 Logical function implementation example using FACs and HACs

scale, designed to implement various arithmetic functions [Wake00, Yarb97]. But it is not this usage that we want to point out now, it is the fact that by means of FACs and HACs non-arithmetic functions can be computed, such as the logic function $f(a, b, c, d) = abc \underline{or}\ d$ and its complement (Fig. 2.6). It can be observed that, on the carry chain, there are generated, in turn, the logic subfunctions ab, abc, $abc \underline{or}\ d$ (the term $abcd$ is absorbed), $(abc \underline{or}\ d) \cdot 1$, and at the sum output of msb rank the logic function generated is $(1 \oplus 0 \oplus (abc \underline{or}\ d))$, i.e. the negation $\overline{abc \underline{or}\ d}$.

In reference to the carry propagation problem, it should be mentioned that, in fact, there are three critical paths, namely: the first begins from the inputs x_0 and y_0 and finishes at the msb of the output sum, z_{n-1} (Fig. 2.3), the second begins at the c_{in} input (when the lsb is a FAC, not a HAC, which correspond to some usages of the adder) and ends at the same z_{n-1}, and the third begins at c_{in} and ends at the c_{out} output of the adder's msb. Out of these three, whose values can differ depending on the technology of implementation, let us refer to the third. This delay between the moment the signal is applied to the c_{in} input of the lsb rank and the moment when the answer is received at the c_{out} output of the msb rank is directly proportional to the number n of ranks and it is the target to reduce through various improved solutions. Let us denote the unfavorable value of this parameter with D and let us suppose the implementation of the carry chain is with NAND gates (Fig. 2.2a or Fig. 2.4a), for each of which we accept the same delay d, no matter the number of inputs. Then for an RCA made up of FACs only, there results $D = 2nd$, and if the lsb is a HAC, then $D = (2n - 1)d$. In terms of complexity [ErLa04], one may affirm that RCA latency is $O(n)$, which has been found to be typical for a serial adder as well. However, the proportionality constant of a serial adder is larger due to the additional time intervals required by the state transition through multiple CLOCK pulses, as well as by the storage of values. The concerns related to the carry chain length are justified because the delay on the chain represents the essential objective to be investigated in case performance improvements are wanted regarding parallel adder solutions.

On the other hand, when included in data processing units, adders supply, besides the result of the operation, some additional information about it, which enable flags to be set. These storage elements are alien, being reunited together with flags for other purposes in what is known as the status register of a computer. The binary configuration from this register, or only a part of it, is used in the investigation of some exception status, as well as for the implementation of conditional jump instructions [HePa03], which brings out, dependent on the flag from the status register, the passing through of one or another branch of a program. Regarding the operation of an adder, such condition/exception flags are represented by "c_{out}", indicating that the result has a 1 at the c_{out} output of the msb, while "negative" and "zero" indicate that the result of the operation is negative and zero respectively, and "overflow", which indicates an exception, when the result does not represent the correct sum. Since the first three states do not require additional comments, we shall concentrate, to a certain extent, upon the state signaled by the "overflow" flag.

Thus, we shall mention, first of all that, if unsigned numbers are added, the occurrence of a $c_{out} = 1$ at the msb of the result means that it exceeds the value of the register capacity, which requires the setting of the "c_{out}" and "overflow" flags. But, if two numbers in two's complement are added, the exceptional state of overflow has to be highlighted when the operands' signs are the same, but differ from the sign of the result. If we denote by v the Boolean variable associated with the overflow, which determines the setting of the homonymous flag, then, for v, the logic equation $v = \overline{x_{n-1}}\ \overline{y_{n-1}} c_{n-1}\ \underline{or}\ x_{n-1} y_{n-1} \overline{c_{n-1}}$ results, where x_{n-1} and y_{n-1} represent the sign bits of the two numbers that are being added, and c_{n-1} represents the carry to the sign bit (refer also to Fig. 2.3). By using the Boolean identity $A\ \underline{\text{or}}\ B = A \oplus B \oplus AB$ [Wake00], we transform the v equation, so that, after the elimination of the term consisting of a logic product of complementary variables, it results that:

$$\begin{aligned} v &= \overline{x_{n-1}}\ \overline{y_{n-1}} c_{n-1} \oplus x_{n-1} y_{n-1} \overline{c_{n-1}} \\ &= (\overline{x_{n-1}}\ \overline{y_{n-1}} \oplus x_{n-1} y_{n-1}) c_{n-1} \oplus x_{n-1} y_{n-1} \end{aligned} \tag{2.1}$$

Since the operations OR and EXCLUSIVE-OR of the terms of the coincidence expression (EXCLUSIVE-NOR) are equivalent, we can substitute $(\overline{x_{n-1}}\ \overline{y_{n-1}} \oplus x_{n-1} y_{n-1})$ by $\overline{x_{n-1} \oplus y_{n-1}}$, and by $(x_{n-1} \oplus y_{n-1} \oplus 1)$ respectively, which allows us, after reordering the terms, to obtain the following:

$$v = x_{n-1}\, y_{n-1} \oplus x_{n-1}\, c_{n-1} \oplus y_{n-1}\, c_{n-1} \oplus c_{n-1} \tag{2.2}$$

On account of the fact that in two's complement the bit sign is not distinguished from an ordinary bit of the number, we have for the carry from the bit sign, c_n, the Boolean equation $c_n = x_{n-1}\, y_{n-1}\ \underline{or}\ x_{n-1}\, c_{n-1}\ \underline{or}\ y_{n-1}\, c_{n-1}$. If the previous Boolean identity is applied twice to this expression, it results that $c_n = x_{n-1}\, y_{n-1} \oplus x_{n-1}\, c_{n-1} \oplus y_{n-1}\, c_{n-1}$ which, substituted in (2.2), leads to the form we are interested in:

$$v = c_n \oplus c_{n-1} \tag{2.3}$$

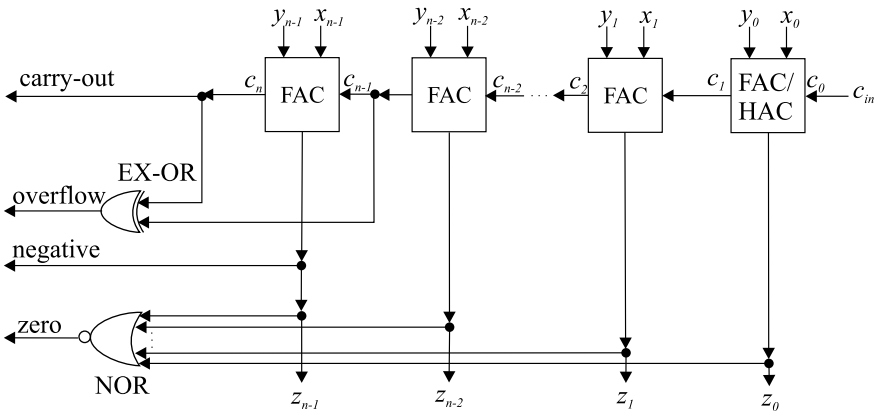


Fig. 2.7 Testing of exception conditions for an adder

Under these circumstances, the overflow for two's complement addition is detected by operating with EXCLUSIVE-OR on the carry variables of the most significant two ranks of the adder. It also should be mentioned that for the same addition in two's complement, c_{out} has no significance. Thus, Fig. 2.7 presents an RCA which allows the addition of unsigned numbers, as well as of those represented in two's complement, having attached additional logic for setting the condition and exception flags from the status register [Parh00]. There is an excessive number of inputs of NOR gates which test the zero result, whose implementation usually requires an OR gate tree succeeded by an inverter gate.

We already saw that the worst case delay varies linearly with respect to the operands' dimension n but the probability of this worst case scenario is reduced. Consequently one method to reduce the critical parameter represented by addition's latency consists of the use of fast components for the implementation of the carry propagation chain. Thus, the so-called Manchester chain is obtained, and the adder which includes it is called a Manchester (Kilburn) adder (MA) [Omon94]. More precisely, the Manchester chain consists, for each rank, of three switches controlled through variables deduced from the input bits. Regarding the binary position i with x_i and y_i inputs, the following variables are obtained: the generation variable $g_i = x_i y_i$, which signifies that, when the input vector is $(x_i, y_i) = (1, 1)$, on the carry propagation chain a 1 is generated; the propagation variable $p_i = x_i \overline{y_i}\ \underline{or}\ \overline{x_i} y_i = x_i \oplus y_i$, which signifies that, when the input vectors are $(x_i, y_i) = (1, 0)$ or $(x_i, y_i) = (0, 1)$, from the carry input of rank i a 1 may be propagated towards the carry output of this rank; finally, the annihilation or absorption variable $a_i = \overline{x_i}\ \overline{y_i}$, which signifies that, when the input vector is $(x_i, y_i) = (0, 0)$, even if at the carry input of rank i a 1 has been passed, at the carry output of this rank, a 0 will be transmitted, and therefore carry annihilation (absorption) takes place. At any moment, one and only one of the three variables takes the value 1, "closing the contact" of the switch, letting the current pass through. Figure 2.8a presents the generation of g_i, p_i and a_i variables in an implementation with logic gates, as well

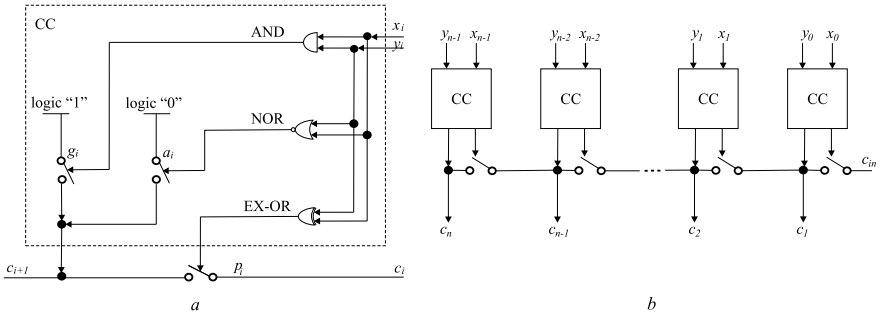


Fig. 2.8 Conceptual diagram of a Manchester adder cell and their interconnection

as the controlled switches representing the elements of the carry control chain (CC). Figure 2.8b presents the connections of the CCs to the carry path corresponding to a Manchester adder of n ranks.

Regarding a Manchester chain of n bits, the total delay consists of the following time components: t_1—required to obtain the control signals for switches (g_i, p_i, a_i), t_2—required for setting the switches (the switches of all bits shall be simultaneously set), and t_3—representing the delay concerning the signal propagation through the carry path. The first two components, t_1 and t_2, are small, and they are generally constant. The dominant component is t_3, which, in the best case, depends only linearly on n. In CMOS technology, the implementation of the switches is best achieved through so-called pass transistors [Parh00, Omon94], but their series connection may lead to delay proportional with n^2, making the Manchester principle applicable only to chains containing a small number of bits (up to 8 bits [Parh00]). For this reason, the MA solution is usually applied combined with other principles in adders forming hybrid configurations.

A last problem, regarding RCA adders, consists of the potential optimization of the structure when one of the operands to be added is a constant. We start from the observation that in addition of constant Y to operand X, Y may be odd, or else it consists of the concatenation of an odd Y', having at the right side a number δ of zeros, which makes Y an even number. In this last case, at addition, the δ least significant bits of X can be found in the sum, and the operation is executed between X and odd Y'. Let us consider, for instance, $\delta = 2$, and the constant $Y = (0, 1, 1, \ldots, 1, 0, 1, 0, 0)$ to be added to $X = (x_{n-1}, x_{n-2}, x_{n-3}, \ldots, x_4, x_3, x_2, x_1, x_0)$, where we obtain the sum Z in the form $Z = (z_{n-1}, z_{n-2}, z_{n-3}, \ldots, z_4, z_3, \overline{x_2}, x_1, x_0)$. The two least significant bits of X are found in the same positions in Z, and the bit from Z, corresponding to the first value of 1 when passing through Y from right to left, z_2, becomes equal to $\overline{x_2}$. Moreover, from this rank, carry $c_3 = x_2$ is generated to the left. The other bits of Z are obtained through an RCA of special construction, made up either from half-adder cells (HAC), when $y_i = 0$ (we have $z_i = x_i \oplus c_i$ and $c_{i+1} = x_i c_i$), or from modified half-adder cells (HAC*), when $y_i = 1$ (we have $z_i = x_i \oplus 1 \oplus c_i = \overline{x_i} \oplus c_i$ and $c_{i+1} = x_i \underline{\text{or}}\, c_i \underline{\text{or}}\, x_i c_i = x_i \underline{\text{or}}\, c_i$), as shown in Fig. 2.9. Thus, Fig. 2.9a and Fig. 2.9b

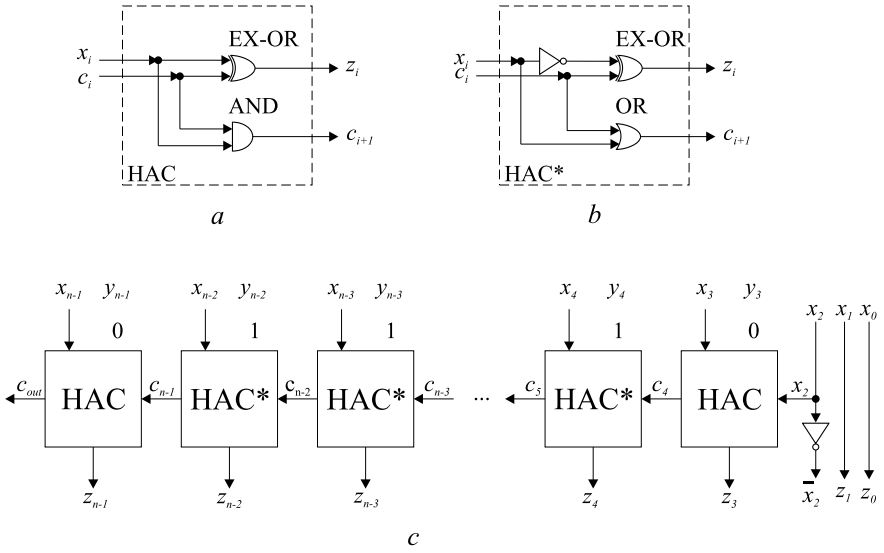


Fig. 2.9 Elements of addition of a constant to an operand

show potential implementations for HAC cells, and for HAC* cells respectively, and Fig. 2.9c shows the chaining of these cells in the particular case when to the operand X the constant Y given above is added.

2.2.2 *Decimal Adders Based on Serial Carry Propagation*

We specify, from the beginning, that we refer only to pure BCD (binary-code decimal), with the weights 8421, and excess-3 BCD code. Also the operation discussed is addition, and as operands, only unsigned numbers are admitted. Generally, addition in BCD can be done in two ways, namely, either by converting the operands into pure binary code, and executing the operation, and then by reconverting the result into BCD, or by executing the operation, directly, in BCD code. If this last method is employed, and if BCD8421 code is taken into account, we should mention that on the conversion of the numbers from binary into BCD through the well-known method of left shift (multiplication by two) [KeSc05, Omon94], for certain decimal digits the addition of the corrective value $0110_2 = 6_{10}$ shall be applied. Thus, multiplication by 2 of the decimal digits ranging between 0 and 4 gives the values from 0 to 8, having binary correspondents from 0000 to 1000, which in BCD8421 correspond to the required values from 0 to 8, without any correction. However, if the decimal digits from 5 to 9 are multiplied by 2, even numbers from 10 to 18 will be obtained which have the most significant digit 1 with 10^1 weight in the decimal number system, but with $2^4 = 16$ weight in the binary number system. This requires the addition of the correction $0110_2 = 6_{10}$ to the binary equivalents from 1010 to

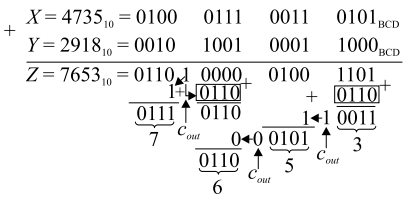


Fig. 2.10 BCD8421 code addition example

10010, obtaining the values from 10000 to 11000 to which the required numbers from 10 to 18, in BCD8421, correspond. As well as this, if the binary equivalents of two decimal digits are added, for instance, $0111_2 = 7_{10}$ with $1000_2 = 8_{10}$, the result will be $1111_2 = 15_{10}$, which, in the decimal system, has a carry (10^1 weight) towards the addition of the next pair of decimal digits, being equivalent to the subtraction from 1111_2 of $1010_2 = 10_{10}$. But this subtraction is executed by adding the two's complement, i.e. the addition of $0110_2 = 6_{10}$, there being obtained the binary equivalent $0101_2 = 5_{10}$ and the expected carry-out with the value 1. Generally, any time the addition of the binary equivalents corresponding to two decimal digits for the sum binary equivalent results in values ranging within 1010 and 1111, the correction 0110 has to be added and the resulting carry-out has to be transmitted to contribute to the addition of the binary equivalents corresponding to the next pair of decimal digits. On the other hand, by adding, this time, the binary equivalents of the decimal digits $8_{10} = 1000_2$ and $9_{10} = 1001_2$, a carry-out and the binary equivalent of the value 0001 will be obtained. In this situation, transformation between the number systems has caused 6 units to become lost. This requires the addition of 0110, so that the four bits (tetrad) of the sum are corrected to $0111_2 = 7_{10}$. Consequently, the addition in BCD8421 code is executed by adding, from right to left, the binary tetrads corresponding to the pairs of decimal digits, and the selective correction of the sum binary equivalents by adding 0110 in the above-mentioned cases, namely, when for the sum tetrad binary numbers from 1010 to 1111 are obtained, and when, evaluating the sum tetrad, carry-out results. Thus, Fig. 2.10 presents an example of addition between $X = 4735_{10}$ and $Y = 2918_{10}$, where all the tetrad additions are considered to take place simultaneously, having initially, all of them, the carry inputs c_{in} set on zero (a facility which can be assured in certain technologies, such as, for instance, CMOS, through precharging [HePa03]). It can be observed that on the addition of the tetrads corresponding to the two least significant decimal digits, the binary number 1101 is obtained, a value which has to be corrected with 0110, an operation which results in $c_{out} = 1$. This carry has to be added, secondly, to the sum tetrad corresponding to the second (from right to left) pair of decimal digits. The correction 0110 also has to be applied to the addition of the binary equivalents of decimal digits 7 and 9, but this time because of the generation of $c_{out} = 1$, a carry which will be afterwards added to the most significant sum tetrad. It also should be mentioned that the sum $Z = 0111\ 0110\ 0101\ 0011_{BCD} = 7653_{10}$ is obtained gradually, in time steps, following the operation of all the possible carries c_{out} between tetrads, a fact suggested through the presentation "in steps" of the operation execution in Fig. 2.10.

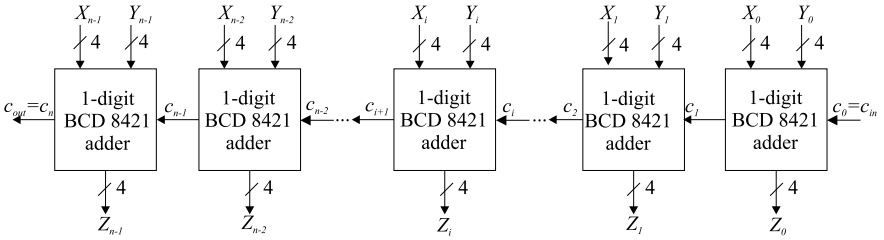


Fig. 2.11 RCA version of an BCD8421 decimal adder block diagram

A decimal adder which operates with unsigned numbers in BCD8421 coding can be synthesized according to the example addition from Fig. 2.10, in RCA mode, by serially chaining adders meant to operate on one tetrad from each operand, X and Y, obtaining a sum digit of the BCD8421 coding (1-digit BCD8421 adder), as shown in Fig. 2.11 [Haye98]. Each such adder, such as, for instance, the one which adds the tetrads X_i and Y_i, is composed of one level which achieves the conventional binary sum Z_i and a second one which allows the selective adding of the corrective value 0110. The activation of the second level takes place through the Boolean function Z'_i (Fig. 2.12a), obtained through minimization, using a Karnaugh map, of the expression consisting of the logic sum of the minterms corresponding to the sum tetrads from $z'_{i,3}z'_{i,2}z'_{i,1}z'_{i,0} = 1010$ to $z'_{i,3}z'_{i,2}z'_{i,1}z'_{i,0} = 1111$, or through c'_{i+1}, which represents the carry c_{out} of the conventional binary adder (Fig. 2.12b). Otherwise, for the carry c_{i+1}, to the tetrad which calculates the binary equivalent Z_{i+1} corresponding to the next decimal digit, we have the Boolean equation:

$$c_{i+1} = c'_{i+1} \underline{or}\ z'_{i+3}z'_{i+1} \underline{or}\ z'_{i+3}z'_{i+2} \tag{2.4}$$

When, based on relation (2.4), $c_{i+1} = 1$ results, this value is applied to the internal FACs of the second addition level, so that it may add to the tetrad Z'_i the value $2^2 + 2^1 = 6_{10}$, the corrected tetrad Z_i is obtained.

It should be mentioned that in the configuration of both the tetrad adders (Fig. 2.12), and the global adder for BCD8421 numbers (Fig. 2.11), for performance improvement methods of speeding up the addition process can be used of the type that will be presented in the following sections. Nevertheless, the penalties in the computation speed caused by the result correction determines a limited applicability, even for the adders of this most widely used decimal code represented by BCD8421, and much more for devices implementing far more complex operations, such as multiplication or division. However, there are several applications where the quantities to be processed are small and where conversions prevail at data introduction and extraction from/to human interfaces, and for which solutions based on decimal arithmetic, in general, and in particular on BCD8421 adders, are suited.

Within the same context of decimal adders, we shall briefly refer to the addition of decimal numbers represented in excess-3 BCD code, due to one of its interesting

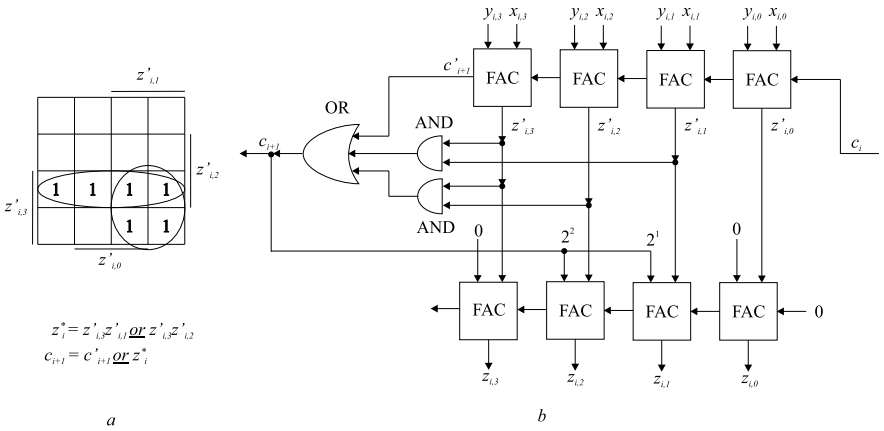


Fig. 2.12 Synthesis of a tetrad adder used in BCD8421 addition

Fig. 2.13 BCDE3 code addition example

$$
\begin{array}{lllll}
X = 4735_{10} = & 0111 & 1010 & 0110 & 1000_{\text{BCDE3}} \\
+\ Y = 2918_{10} = & 0101 & 1100 & 0100 & 1011_{\text{BCDE3}} \\
\hline
Z = 7653_{10} = & 1101 & 0110 & 1011 & 0011 \\
& 0011 & 0011 & 0011 & 0011 \\
\hline
& 1010 & 1001 & 1000 & 0110 \\
& 7 & 6 & 5 & 3
\end{array}
$$

property. We shall start from the fact that each decimal digit has a bias of 3, which leads, for the addition of such two digits, to the correct generation of carry-out, even in the critical cases when, in BCD8421 code, the sum tetrads result in values ranging between 1010 and 1111, because the correction $3 + 3 = 6$ is implicitly assured. Thus, carry-out is correctly produced on the addition of the excess-3 BCD representations of any two decimal digits, and, consequently, the addition of two numbers in excess-3 BCD can be achieved through a conventional binary adder. But, in order to obtain the sum representation in excess-3 BCD code, correction of all the tetrads is required after the execution of the addition operation. Thus, if on the addition of excess-3 BCD representations of two decimal digits $c_{out} = 1$ is obtained, then the value 0011_2 shall be added to the binary sum tetrad, for compensation, and, if on the given addition $c_{out} = 0$ is obtained then the value 0011_2 (corresponding to one of the two biases) shall be subtracted from the binary sum tetrad. Figure 2.13 shows the operation of the same example from Fig. 2.10, this time, in excess-3 BCD (BCDE3). The carry values between tetrads are pointed out, values which determine, as the case requires, the addition or the subtraction of the corrective value $0011_2 = 3_{10}$ of the sum tetrads. It should also be mentioned that this operation requires only two passes, one to obtain the conventional binary sum and one for correction, being better than that executed in BCD8421 code, as far as performance is concerned.

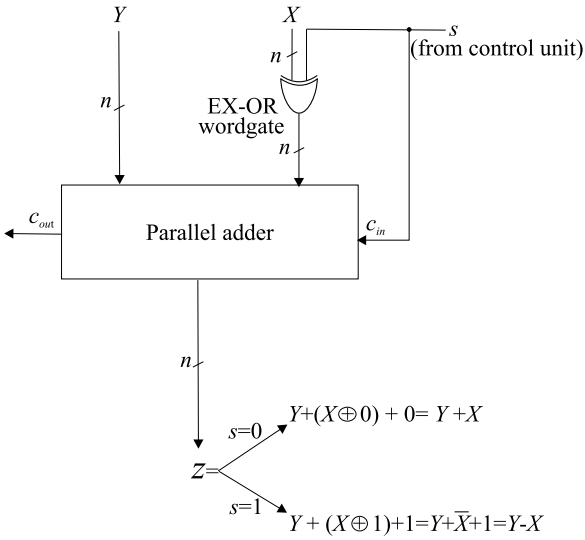


Fig. 2.14 Block diagram of a binary adder/subtracter

2.2.3 Subtracters Based on Serial Carry/Borrow Propagation

The binary subtraction operation is mainly executed by using an adder which allows the addition of the subtrahend, adequately negated, to the minuend. Thus, the most often used implementation of subtraction corresponds to the operands' representation in two's complement, when the subtrahend is one's complemented and then added to the minuend together with a binary unit which has also to be added to the least significant rank. The operation can be executed by using a parallel binary adder, regardless its type, to which the minuend Y is supplied, along with the subtrahend X, after it has passed through a layer of EXCLUSIVE-OR gates, on the whole word length (EX-OR wordgate), as shown in Fig. 2.14 [Haye98]. The binary unit is added to the least significant rank by means of the carry-in input (c_{in}) that remains available, by applying to it the variable s (from "subtract"), which also controls one of the inputs of all the EXCLUSIVE-OR gates of the wordgate. The control unit has the task to establish the logic value corresponding to s, configuring the parallel adder as a subtracter ($Y - X$) when $s = 1$, and leaving its function ($Y + X$) unchanged when $s = 0$.

But there are cases, such as the combinational array structure multiplication for multiplication based on Booth recoding from Sect. 3.9 (refer to Fig. 3.48), to which combinational array structures dedicated to binary division can also be added [Omon94], when it is useful to assure a conventional subtraction as a separate operation, executed by an independent device called a binary subtracter. The fundamental structural element of such a device is, by analogy with the full adder cell (FAC) of an RCA binary adder, a full subtracter cell (FSC). The logic design of an FSC is based on the behavioral description in the truth table from Fig. 2.15a, elaborated for some rank i of a structure which performs the subtraction between

Inputs			Outputs	
y_i	x_i	b_i	z_i	b_{i+1}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

a

z_i

y_i \ $x_i b_i$	00	01	11	10
0		1		1
1	1		1	

b

b_{i+1}

y_i \ $x_i b_i$	00	01	11	10
0		1	1	1
1			1	

c

Fig. 2.15 Tables used for logical equations' synthesis corresponding to a full subtracter cell's outputs

the subtrahend X and the minuend Y. As the inputs are the variables, y_i and x_i, as well as the borrow input b_i, requested by the previous rank $(i-1)$, and the outputs are represented by the difference functions z_i, as well as the borrow output b_{i+1}, requested to the next rank $(i+1)$. The completion of the logic values for the outputs z_i and b_{i+1} from the truth table (Fig. 2.15a) has been obtained by adding the values corresponding to the variables x_i and b_i, the resulting sum being subtracted from the value of y_i. Thus, for instance, for the triplet $(x_i, y_i, z_i) = 010$, we have $x_i + b_i = 1 + 0 = 1$, which, if subtracted from $y_i = 0$, leads to the difference bit $z_i = 1$ and to the borrow bit $b_{i+1} = 1$. The elaboration of the Boolean equations corresponding to the FSC output logic functions have been obtained by using Karnaugh maps represented in Fig. 2.15b for z_i, and in Fig. 2.15c for b_{i+1}. It should also should be mentioned that the Boolean expression for the difference output z_i is given by the same odd parity function $z_i = x_i \oplus y_i \oplus b_i$ corresponding to the sum output z_i of a FAC, if the carry variable c_i is substituted by the borrow variable b_i. This will help us to reconfigure a FAC into a FSC, when needed (refer to the combinational array structure from Fig. 3.48). On the other hand, the covering of the binary units from Fig. 2.15c leads, for b_{i+1}, to an expression similar to that for the carry to the next rank, c_{i+1}, namely $b_{i+1} = x_i\overline{y_i}$ *or* $x_i b_i$ *or* $\overline{y_i} b_i$. We also mention that for the synthesis of the borrow function b_{i+1}, we may also use the expression $b_{i+1} = \overline{x_i}\,\overline{y_i} b_i$ *or* $x_i\overline{y_i}\,\overline{b_i}$ *or* $x_i\overline{y_i} b_i$ *or* $x_i y_i b_i = \overline{y_i}(\overline{x_i} b_i$ *or* $x_i\overline{b_i})$ *or* $x_i b_i = (x_i \oplus b_i)\overline{y_i}$ *or* $x_i b_i$. Based on the Boolean equations for z_i and b_{i+1}, the FSC synthesis results, which can be used in the configuration of binary subtracters, one of the solutions being, for instance, the concatenation in cascade of the FSCs in an RCA manner.

Finally, we shall refer to the subtraction of operands represented in BCD8421 code. By analogy to the operation executed with binary numbers which consists of the addition to the minuend of the subtrahend's two's complement, in this case we will resort to an addition, as well, but the one's complement is substituted by the

Fig. 2.16 BCD8421 code subtraction example

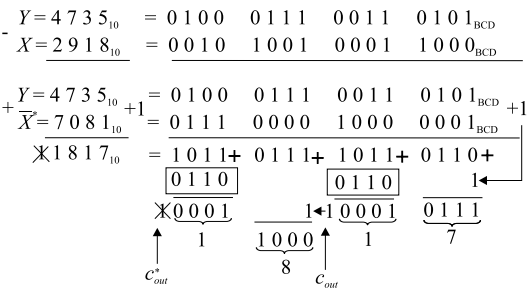


Fig. 2.17 Truth table for synthesis of a code translator for the nine's complement implementation

Inputs				Outputs			
$x_{i,3}$	$x_{i,2}$	$x_{i,1}$	$x_{i,0}$	$\overline{x}^*_{i,3}$	$\overline{x}^*_{i,2}$	$\overline{x}^*_{i,1}$	$\overline{x}^*_{i,0}$
0	0	0	0	1	0	0	1
0	0	0	1	1	0	0	0
0	0	1	0	0	1	1	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	0	0
0	1	1	0	0	0	1	1
0	1	1	1	0	0	1	0
1	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0

more expensive (as far as generation is concerned) nine's complement, to which a binary unit (similar to the two's complement case) is added. Thus, Fig. 2.16 presents the subtraction of the numbers added in Fig. 2.10 and Fig. 2.13, an operation in which from the minuend $Y = 4735_{10}$ the subtrahend $X = 2918_{10}$ is subtracted. In fact, this operation consists of the addition to Y of the nine's complement corresponding to X, namely $\overline{X^*} = 7081_{10}$, and one more unit. The addition operation is executed in the mode described in the example from Fig. 2.10, with the selective application of the correction 0110. The addition implementation can be done with the adder from Fig. 2.11, detailed in Fig. 2.12. The carry decimal digit c^*_{out}, in our case $c^*_{out} = 1$, shall be ignored, according to the binary operation model. As for the nine's complement implementation, it has to be executed separately for each decimal digit. In this case, an adder/subtracter on 4 bits can be used or a code translator having at the base of synthesis the truth table from Fig. 2.17 which is presented for a decimal digit X_i of the subtrahend. The translator synthesis can easily be done, resulting in the Boolean equations for the bits corresponding to the nine's complement of the decimal digit, by taking into account the fact that the unused

binary configurations in Fig. 2.17 were employed at minimization, for instance $\overline{X_i^*}$, $\overline{x_{i,3}^*} = \overline{x_{i,3}}\,\overline{x_{i,2}}\,\overline{x_{i,1}} = \overline{x_{i,3}}\ \underline{or}\ \overline{x_{i,2}}\ \underline{or}\ \overline{x_{i,1}}$, $\overline{x_{i,2}^*} = \overline{x_{i,3}}(x_{i,2} \oplus x_{i,1})$, $\overline{x_{i,1}^*} = \overline{x_{i,3}}x_{i,1}$ and $\overline{x_{i,0}^*} = \overline{x_{i,0}}$.

2.2.4 Carry-Lookahead Adders

An adder is a combinational circuit generally with n output functions $z_{n-1}, \ldots, z_i, \ldots, z_1, z_0$. Such a function can be expressed in the form of a logic sum of logic products (SOP). Thus, as already seen above, we have, for instance, $z_i = \overline{x_i}\,\overline{y_i}c_i\ \underline{or}\ \overline{x_i}y_i\overline{c_i}\ \underline{or}\ x_i\overline{y_i}\,\overline{c_i}\ \underline{or}\ x_iy_ic_i$ (refer also to the implementation from Fig. 2.4a). For all the logic products, the values of the input variables x_i and y_i are initially known, but the value of variable c_i is only known when it has propagated serially, from rank to rank, as has been seen in the adders based on the RCA principle. In order to accelerate the addition process, the sum bits will not be formed until the arrival of the carry, and the carry bits shall be anticipatorily generated, by directly using the values of the input variables of the previous ranks of the adder. Thus, an adder based on a principle which differs from that applied in the RCA will result, i.e. a carry-lookahead adder (CLA) [Stal99]. Let us start from the already known expression which corresponds to the carry generated in the rank i, namely $c_{i+1} = x_iy_i\ \underline{or}\ x_ic_i\ \underline{or}\ y_ic_i$, and let us rewrite it as a function of the already used generation variables, $g_i = x_iy_i$, and propagation variables, $p_i = x_i\ \underline{or}\ y_i$, so that the recurrent relation $c_{i+1} = g_i\ \underline{or}\ p_ic_i$ will be obtained. This expression shows that if $g_i = 1$ we have a carry-out at rank i whether or not the carry comes from the previous rank. Otherwise, if $p_i = 1$, then $c_{i+1} = c_i$, and the carry propagation is obtained. But an equation of similar form can also be written for c_i, i.e. $c_i = g_{i-1}\ \underline{or}\ p_{i-1}c_{i-1}$, and, recursively, the following can be obtained:

$$\begin{aligned} c_{i+1} &= g_i\ \underline{or}\ p_ic_i = g_i\ \underline{or}\ p_i(g_{i-1}\ \underline{or}\ p_{i-1}c_{i-1}) \\ &= g_i\ \underline{or}\ p_i\big(g_{i-1}\ \underline{or}\ p_{i-1}(g_{i-2}\ \underline{or}\ p_{i-2}c_{i-2})\big) = \cdots \\ &= g_i\ \underline{or}\ p_ig_{i-1}\ \underline{or}\ p_ip_{i-1}g_{i-2}\ \underline{or}\ \ldots\ \underline{or}\ p_ip_{i-1}\ldots p_1g_0\ \underline{or}\ p_ip_{i-1}\ldots p_1p_0c_0 \end{aligned} \tag{2.5}$$

where the values of all the variables g and p are obtained using the input variables x and y.

To be more exact, let us suppose that $i = 3$, and let us elaborate, by using (2.5), the Boolean equations corresponding to the carries from the first four ranks. Thus, we have:

$$\begin{aligned} c_1 &= g_0\ \underline{or}\ p_0c_0 \\ c_2 &= g_1\ \underline{or}\ p_1g_0\ \underline{or}\ p_1p_0c_0 \\ c_3 &= g_2\ \underline{or}\ p_2g_1\ \underline{or}\ p_2p_1g_0\ \underline{or}\ p_2p_1p_0c_0 \\ c_4 &= g_3\ \underline{or}\ p_3g_2\ \underline{or}\ p_3p_2g_1\ \underline{or}\ p_3p_2p_1g_0\ \underline{or}\ p_3p_2p_1p_0c_0 \end{aligned} \tag{2.6}$$

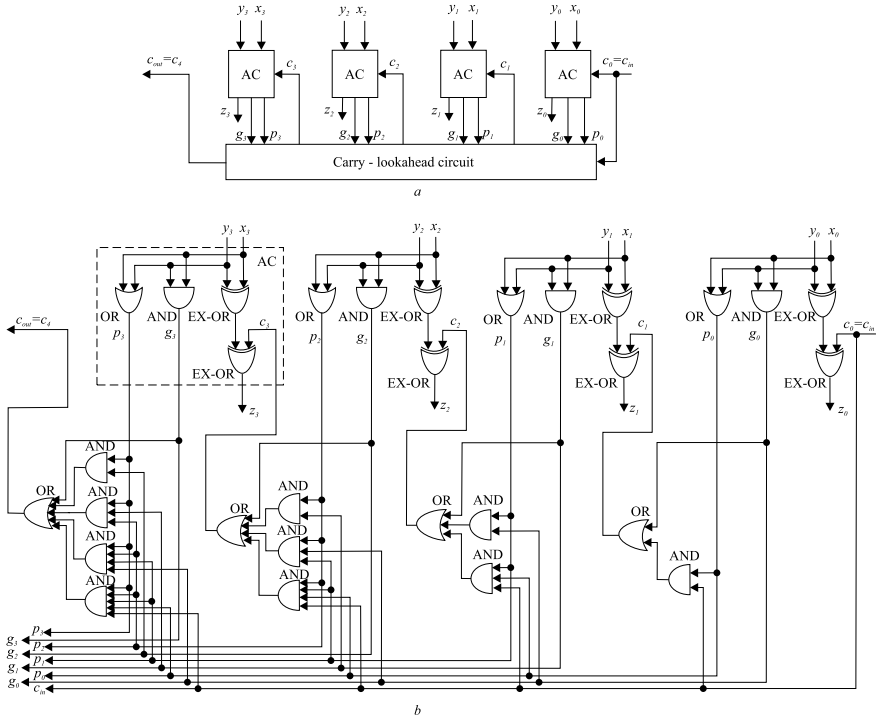


Fig. 2.18 Block diagram of a 4-bit conventional CLA and one of its gate level implementation versions

The implementation of these equations is assured by a carry-lookahead circuit, while the CLA adder also includes an adder cell (AC) for each rank. Unlike a FAC, the AC performs the sum function and generates the variables g and p. As far as the case taken into account is concerned, when $i = 3$, Fig. 2.18a presents the block diagram of the CLA adder, and Fig. 2.18b presents an implementation version using AND and OR gates for the carry-lookahead circuit, and EXCLUSIVE-OR gates for the sum function generation (according to the model from Fig. 2.4b, and Fig. 2.4c). A CLA adder, such as the adder presented in Fig. 2.18, which maintains the unitary carry generation mode, is also called a full carry-lookahead adder [Parh00].

If the CLA configuration from Fig. 2.18 is analyzed from the point of view of its performance, one can easily find out that on the path of the signals from the primary inputs, to which operands are applied, to the outputs where the sum bits are available, there are, in the worst case, four gate levels (the OR gate for the propagation variable forming, the AND and OR gates for the carry forming, and the EXCLUSIVE-OR final gate for the sum rank forming, the last one being considered to represent one logic level). Obviously, as compared to the delay on $2n$ logic levels characteristic for an RCA adder with n ranks, the reduction to only four logic levels,

whatever the number n of the ranks, which characterizes a CLA adder, makes this last solution a clearly superior one from the point of view of its performance.

However, the above-mentioned analysis has not taken into account the fact that, the more we advance towards the more significant ranks, the gates' fan-in increases, so that, according to the rank i, an AND gate and an OR gate used in the generation of a carry have a fan-in of $(i + 2)$. But it is known [Wake00] that the fan-in increase results in the degradation of certain dynamic parameters of the gate, our interest, in this context, being the propagation time t_{PD}. Besides this aspect, the libraries of integrated circuit manufacturers comprise gates with a limited number of inputs, so that for practical n values (e.g. $n = 32$), the functions achieved by the above mentioned AND and OR circuits require, for implementation, tree-like networks which lead to increased cost, and, more important, to increased latency. Moreover, signal p_i from relation (2.5) has an excessive fan-out, its application being necessary to $(i + 1)$ AND gates, and, consequently, requiring a power control solution with consequences, unfavorable as well, as regards performance. Finally, to the above mentioned aspects there is added another one which is essential when, for implementation purposes, the very large scale integration technology (VLSI) is used. A structure of the type from Fig. 2.18b is not regular, and there cannot be defined blocks with ordered interconnections because of the presence of alternating short and long connections. Consequently, the construction of full carry-lookahead adders is not practical, especially when n is large and justifies concern for the improved usage of the CLA principle, aimed at the avoidance, at least partly, of the above-mentioned disadvantages.

The method most frequently resorted to in practice [Kore02, Parh00] is based on the increase of the logic levels, wherefrom its name of multilevel lookahead, aiming to obtain an ordered structure characterized by regularity, that can be executed favorably in VLSI technology. This method is based on the fact that the carry generation and propagation can be done gradually, in steps, these functions being assured by means of some generation and propagation blocks. Thus, starting from the simplest equation of (2.6), i.e. $c_1 = g_0 \underline{\ or\ } p_0 c_0$, let us also keep this form of equation for c_2, substituting into the expression from (2.6) to get $c_2 = G_{0,1} \underline{\ or\ } P_{0,1} c_0$, where $G_{0,1} = g_1 \underline{\ or\ } p_1 g_0$ signifies the fact that the carry is generated in the block made up of the first two ranks, 0 and 1, and $P_{0,1} = p_1 p_0$ signifies the fact that the carry propagates through the given block.

Generalizing the above-mentioned aspects, let us take into account some indexes, i, j and k, with $i < j$ and $j + 1 < k$, and let us elaborate, according to the model of c_2, the Boolean equation for c_{k+1}. We shall obtain $c_{k+1} = G_{i,k} \underline{\ or\ } P_{i,k} c_i$, where $G_{i,k} = G_{j+1,k} \underline{\ or\ } P_{j+1,k} G_{i,j}$ signifies the fact that the carry is generated in the block consisting of the ranks from i to k, either through its generation in the subblock consisting of the more significant ranks, from $(j + 1)$ to k, or through its generation in the subblock consisting of the less significant ranks, from i to j, and then its propagation through the subblock consisting of the more significant ranks, from $(j + 1)$ to k, while $P_{i,k} = P_{i,j} P_{j+1,k}$, signifies the fact that the carry propagates through the given subblocks [HePa03]. In order to arrive from blocks consisting of several ranks at blocks corresponding to one rank, we shall set up $G_{i,i} = g_i$, and

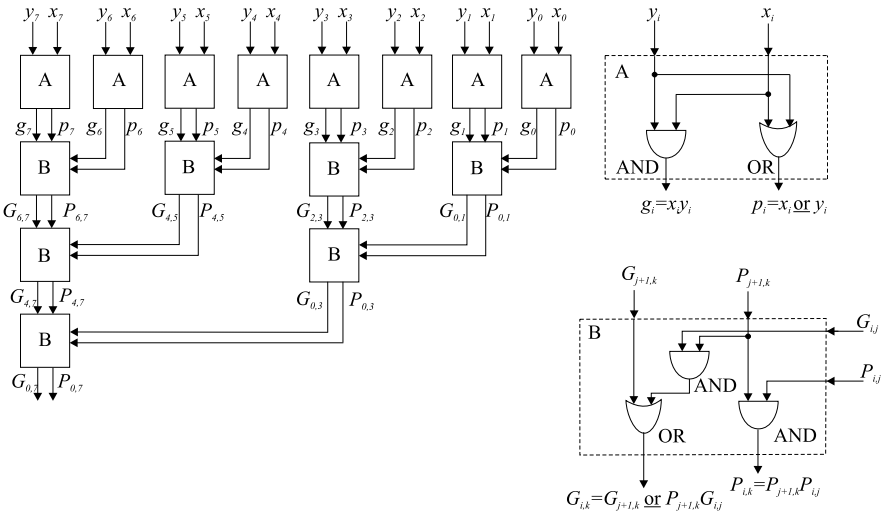


Fig. 2.19 Binary tree structure for building the generation and propagation functions of an 8-bit multilevel CLA

$P_{i,i} = p_i$, when the above-mentioned Boolean equations imply the adjustment of the relations between indexes at $i \leq j < k$.

For instance, let us take into account the block made up of the ranks from 0 to 3, and let us deduce the equation of c_4, given by (2.6), starting from the form $c_4 = G_{0,3}\ \underline{or}\ P_{0,3}c_0$. Thus, let us suppose, first of all, that the block is made up of the subblocks to which there correspond the ranks 0 and 1, and 2 and 3. By setting $i = 0$, $j = 1$ and $k = 3$, we can write $G_{0,3} = G_{2,3}\ \underline{or}\ P_{2,3}G_{0,1}$, and $P_{0,3} = P_{2,3}P_{0,1}$ respectively. Then, arriving at blocks to which there correspond individual ranks, and setting, for instance, $i = j = 2$ and $k = 3$, we have $G_{2,3} = G_{3,3}\ \underline{or}\ P_{3,3}G_{2,2}$. Since $G_{3,3} = g_3$, $G_{2,2} = g_2$ and $P_{3,3} = p_3$, we rewrite $G_{2,3} = g_3\ \underline{or}\ p_3g_2$. In case a similar procedure is applied to $P_{2,3}$, $G_{0,1}$ and $P_{0,1}$, the following will be obtained: $G_{0,3} = G_{3,3}\ \underline{or}\ P_{3,3}G_{2,2}\ \underline{or}\ P_{3,3}P_{2,2}(G_{1,1}\ \underline{or}\ P_{1,1}G_{0,0}) = g_3\ \underline{or}\ p_3g_2\ \underline{or}\ p_3p_2g_1\ \underline{or}\ p_3p_2p_1g_0$, and $P_{0,3} = P_{2,3}P_{0,1} = P_{3,3}P_{2,2}P_{1,1}P_{0,0} = p_3p_2p_1p_0$.

Following these specifications, let us present the construction of a multilevel CLA, considering, to be more concrete, that operands X and Y have eight ranks. Starting from the individual bits, we form, first of all, generation and propagation variables with which we shall first compose G and P functions corresponding to the subblocks, and, finally, to the entire block. Then, the given functions will be used for the evaluation of the carries. Thus, Fig. 2.19 presents the part of the generation of G and P functions in a binary tree type structure, in which the functions use two types of cells, A and B. Their internal configurations are detailed in Fig. 2.19, and they compute $g_i = x_i y_i$ and $p_i = x_i\ \underline{or}\ y_i$ functions in cells of type A, and $G_{i,k} = G_{j+1,k}\ \underline{or}\ P_{j+1,k}G_{i,j}$ and $P_{i,k} = P_{j+1,k}P_{i,j}$ functions in cells of type B.

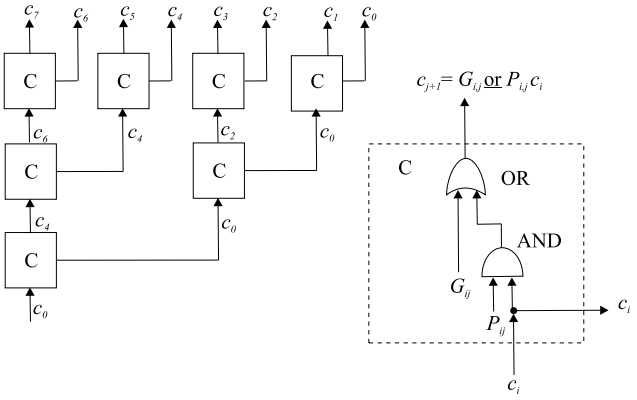


Fig. 2.20 Binary tree structure for building the carries of an 8-bit multilevel CLA

On the other hand, Fig. 2.20 presents a second binary tree structure meant to supply carries. It uses a third cell, of type *C*, implementing the Boolean equation $c_{j+1} = G_{i,j}$ *or* $P_{i,j}c_i$. In the diagram from Fig. 2.19, the signals "flow" downwards, but in the diagram from Fig. 2.20, also a tree diagram, the signals "flow" in the reverse direction, the c_0 signal being applied at its basis, all the carry bits being gradually generated. Each type *C* cell has to "know" the corresponding pair of values $(G_{i,j}, P_{i,j})$, but it can be seen that there is a one-to-one correspondence between the cells of type *B* from Fig. 2.19 and the cells of type *C* from Fig. 2.20, so that, through their combination, there are obtained the pairs of values we are interested in $(G_{i,j}, P_{i,j})$. Figure 2.21 presents the multilevel CLA structure, by overlapping the two tree diagrams from Fig. 2.19 and Fig. 2.20. There can be observed the combined cells $(B + C)$, which, as shown in the detailed diagram, combine the implementations of the separate diagrams, as well as the *AC* cells representing expansions of the cells of type *A* from Fig. 2.19 with the circuits that generate the sum bit (in our case, represented, for simplicity, by two EXCLUSIVE-OR gates). The operands to be added are applied in the upper part of the diagram, and the signals propagate downwards to combine with the carry c_0, and then they propagate upwards for the computation of the sum bits.

The analysis of the multilevel CLA structure from Fig. 2.21 as regards the performance/cost impact shows, first of all, that the signals have to cross, in the worst case, on the path from the input operands to the sum result, a total of 12 logic levels (1 for the generation of the pairs (g, p), $(2 \cdot 2)$ for the generation of the pairs (G, P), $(3 \cdot 2)$ for the generation of the carries, and 1 for the sum forming, if, again, the contribution of the EXCLUSIVE-OR final gate of one logic level is taken into account). If compared to the $2n = 2 \cdot 8 = 16$ logic levels corresponding to an RCA with operands of the same dimension, the improvement is small, but it may become considerable when *n* has greater values. This is because the number of the cell levels (B + C) equal to $3 = \log_2 8$ in Fig. 2.21, is, generally, $\log_2 n$, a situation in which the number of logic levels through which the signals propagate, in the worst case,

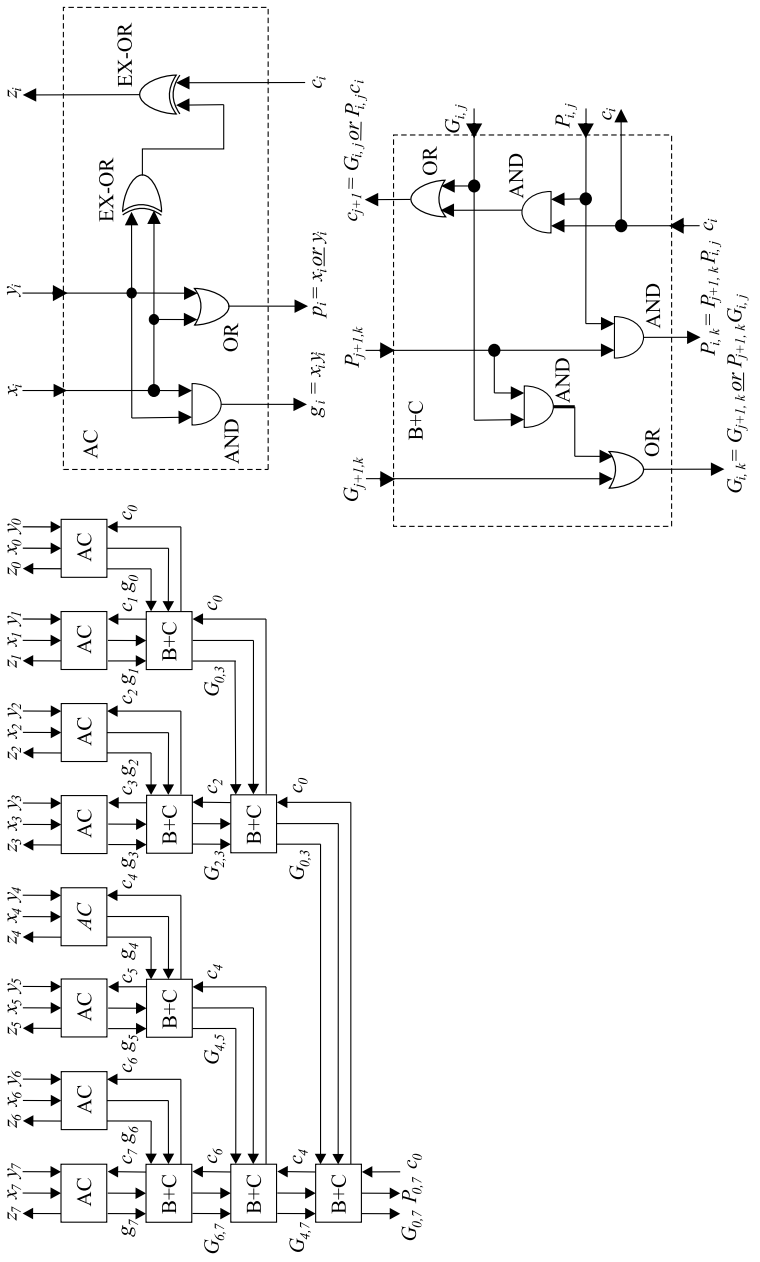


Fig. 2.21 Block diagram of an 8-bit multilevel CLA and gate level implementations of its cells

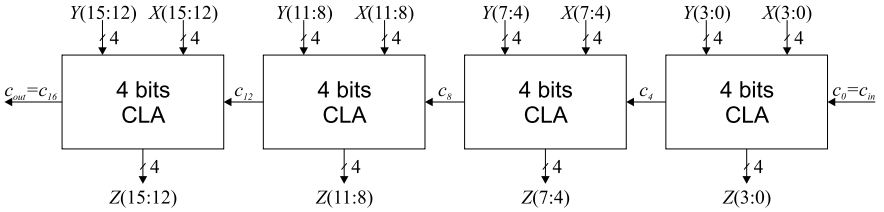


Fig. 2.22 Block diagram of a 16-bit hybrid adder consisting in connecting 4-bit CLA segments in a RCA manner

becomes equal to $(1 + (2\log_2 n - 2) + 2\log_2 n + 1) = 4\log_2 n$, under the same assumptions. When, for instance, $n = 64$, there result 24 logic levels, a much reduced number as compared to the 128 levels corresponding to an RCA of the same dimension. Regarding the aspects connected with the latency of the addition, let us increase the block dimension from two to four ranks. In this case, the height of the cell $(B + C)$ tree decreases from $\log_2 n$ to $\log_4 n$, and the number of logic levels, under the same assumptions as above, becomes $4\log_4 n$. This might lead, for the particular case when $n = 64$, to only 12 logic levels, but, in fact, the improvement is smaller due to the increased delay on the gates which have, in this case, an increased fan-in [Parh00].

If we refer to the cost of a multilevel CLA, and consider, roughly, the AC and $(B + C)$ cells of the same complexity, also equal to the complexity of a FAC, mention should be made that, as compared to an RCA, the number of cells is almost doubled. Thus, in case $n = 8$, we have 15 cells in the multilevel CLA from Fig. 2.21, as compared to only 8 cells in the RCA with the same dimension. This issue is not so dramatic since, in terms of VLSI technology, the investment consists of the silicon substrate area needed for structure integration. As concerns this aspect, the complexity of the area for a multilevel CLA layout of n ranks may be considered, to a good approximation, to be $(n \log_2 n)$ and not $2n$ [HePa03].

The deficiencies of both the RCA and CLA principles can be overcome, taking into account the technological factors introduced on account of the fact that certain technologies favor either one principle or the other, by appealing to hybrid solutions which combine the two methods. Thus, when there is available a certain technology where the CLA variant is easily implemented, it is suggested to serialize the segments built on the basis of this principle [Haye98]. For instance, let us consider $n = 16$, and CLA segments of four ranks, of the type presented in Fig. 2.18, in cascade connection, so that the structure from Fig. 2.22 is obtained. As shown in Fig. 2.18b, $4 \cdot 2 = 8$ logic levels are crossed on the channel between c_{in} and c_{out}, and if we refer to the latency, measured in terms of logic levels, between the input operands and the sum result, the value 10 will be obtained (1 for the forming of p_3, 2 in each of the 4 segments for forming the carries, and 1 represented by the EXCLUSIVE-OR gate for the formation of the sum bits in the last segment). The multilevel synthesis can also be applied to the configuration of CLA segments.

Alternatively, when technologies favorable to the RCA principle are available, a hybrid solution can be synthesized to generate the carries in multilevel CLA mode

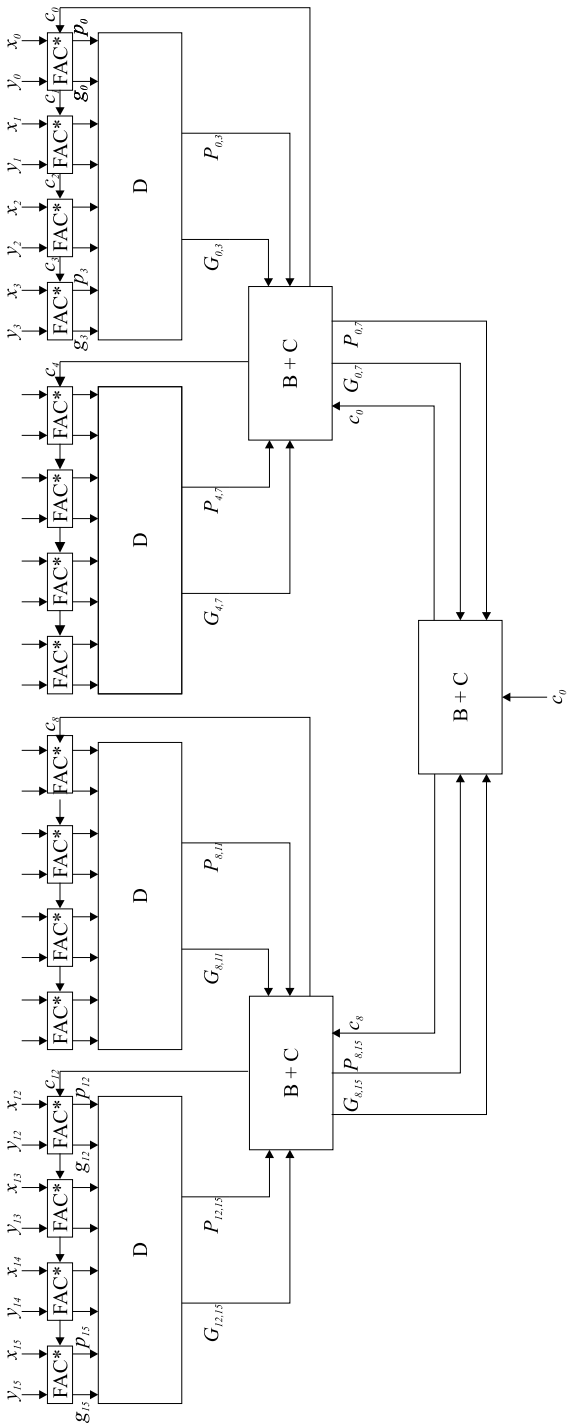


Fig. 2.23 Block diagram of a 16-bit hybrid adder consisting in connecting 4-bit RCA segments in a CLA manner

for FAC cells segments interconnected in RCA mode. Thus, Fig. 2.23, presents, for the previous particular case, i.e. $n = 16$, the structure of the second type of hybrid adder. In the superior stage, the carries are serially transmitted between the FAC* cells, representing FAC cells (Fig. 2.4) to which an AND gate and an OR gate for (g, p) pair forming are added. To each four ranks there is attached, this time, a D type cell meant for (G, P) pair generation for four ranks, in a similar mode to what has been presented above. The diagram also comprises three cells of type $(B + C)$, identical from the constructive point of view to those from Fig. 2.21. The adder outputs are omitted for clarity reasons. The latency of such a construction is near to that corresponding to a multilevel CLA adder only when the carry serial propagation has delay values comparable with those corresponding to a cell of type $(B + C)$.

2.2.5 Carry-Skip Adder

The design solutions of the adders based on RCA and CLA classical principles represent extreme ones, both in terms of performance, and cost. Between them, some hybrid solutions can be adopted which combine the two concepts, as can a solution which is based on the omission of carry serial propagation, known as the carry-skip adder (the short form CSA has been attributed to the adder based on the carry save principle, called the carry save adder, so we shall use for this adder the short form CSkA) [HePa03]. The construction of CSkA starts from the analysis of the Boolean equations for the functions G and P which, in case of the least significant segment of 4 bits, have the known expressions $G_{0,3} = g_3 \underline{or}\ p_3 g_2 \underline{or}\ p_3 p_2 g_1 \underline{or}\ p_3 p_2 p_1 g_0$, and $P_{0,3} = p_3 p_2 p_1 p_0$ respectively. Out of the two equations, the calculation of $P_{0,3}$ is much easier than that of $G_{0,3}$, it requires only an AND gate with four inputs. This feature can be exploited in the construction of a CSkA, namely when the propagation signal corresponding to one block becomes active, the carry does not propagate serially, from rank to rank, as happens in the RCA adder, instead a bypass of the block is created which skips the transmission of the carry between ranks. Thus, Fig. 2.24a schematically presents an RCA adder on 16 bits, and Fig. 2.24b shows it transformed into a CSkA adder. The omission of a block is achieved through AND gates which allow the passage of the carries c_4 and c_8 when the functions $P_{4,7}$ and $P_{8,11}$ respectively have the logic value 1, each of them being obtained through the logic product (AND gate) of four propagation variables p, which, in their turn, are obtained through an OR gate from the input variables x and y. The additional logic circuits mentioned are considered to be included in RCA* blocks, which are thus different from RCA ones (Fig. 2.24).

Mention should be made that the bypass principle becomes practical only when a technology is available which allows easy deletion of the carry signals for each adder block at the beginning of each operation. Precharging in CMOS technology enables the achievement of this requirement, allowing the avoidance of some non-authentic carry-out signal generation at each block. Starting from the initial state with all the block carries on 0, the carries will propagate serially, simultaneously and in parallel

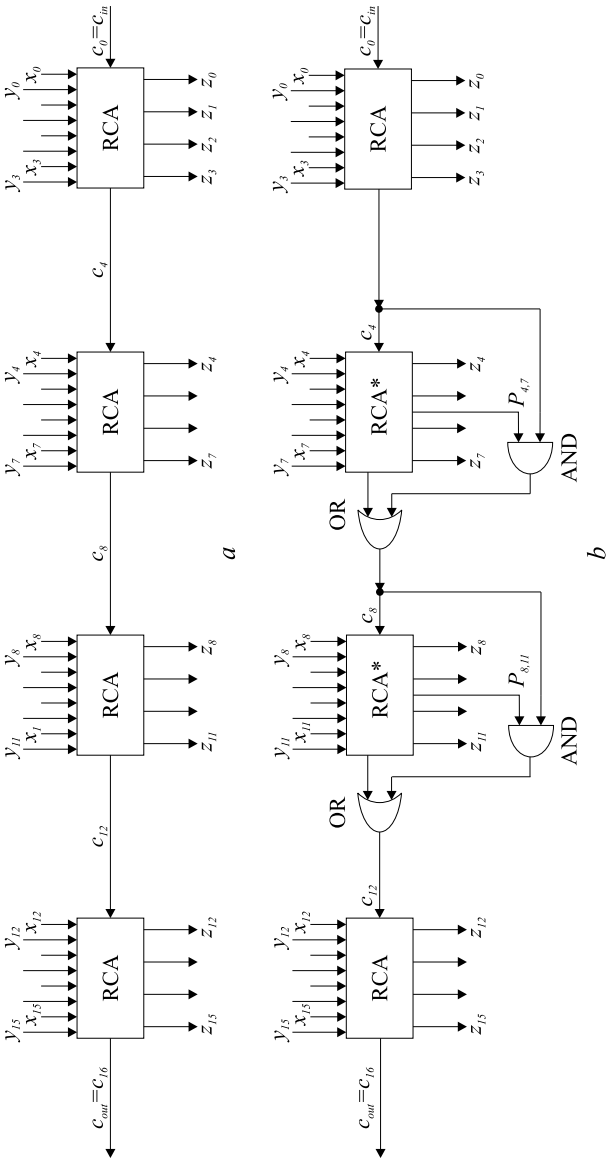


Fig. 2.24 Block diagram of a 16-bit CSkA

on each block. Thus, if a carry is generated in a certain block, then the carry-out of the given block is correctly obtained, even if the carry-in at that block has not yet reached the correct value. In a way, the block carry-outs are assimilated to the G functions which are specific to the multilevel CLA concept. Once the carry-out of a certain block is generated, as shown in Fig. 2.24b, it is applied not only to the next more significant block, but also to the AND gate meant to facilitate its bypass, when the P function associated with the given block allows it. Thus, in the CSkA from Fig. 2.24b, the worst case of the carry propagation delay is obtained when the carry is generated in the least significant 0 rank, passing serially through the ranks from 1 to 3, and then bypassing the following two blocks through AND-OR pairs of gates and also being serially propagated through the ranks from 12 to 15. This amounts to $(4 \cdot 2 + 2 \cdot 2 + 3 \cdot 2 + 1) = 19$ gate levels until the moment the sum bit z_{15} is correctly obtained. Consequently, there results an important performance improvement as compared to the $(2 \cdot 15 + 1) = 31$ gate levels of the RCA of the same dimension (Fig. 2.24a).

If the example from Fig. 2.24 is generalized, the worst propagation delay, corresponding to a CSkA of n ranks divided into blocks of b ranks, is obtained for the carry generation in the lsb rank and its propagation until the correct msb sum bit results. This also includes the addition of the $2b$ logic levels from the least significant block to which are added the $2((n/b) - 2)$ gate levels corresponding to the blocks situated between the extreme ones and bypassed through AND-OR gate pairs and the $(2(b-1)+1)$ gate levels for the serial propagation through the msb block up to the msb rank, including the EXCLUSIVE-OR final gate for the msb sum bit. Denoting by L the latency of a CSkA of n ranks with blocks of length b, in case the same delay d is assumed on the gates, no matter their type, the following will be obtained for this performance parameter:

$$L = \big(2b + 2\big((n/b) - 2\big) + 2(b-1) + 1\big)d = \big(2(n/b) + 4b - 5\big)d \tag{2.7}$$

Starting from (2.7), let us determine the optimum dimension, b_{opt}, for CSkA structure blocks, which occurs when the derivative of L w.r.t. b is 0, obtaining:

$$\frac{dL}{db} = -\frac{2nd}{b^2} + 4d = 0 \quad \Rightarrow \quad b_{opt} = \sqrt{\frac{n}{2}} \tag{2.8}$$

Substituting (2.8) in (2.7), there results the worst delay corresponding to CSkA segmentation in blocks of optimum length, namely:

$$L_{opt} = (4\sqrt{2n} - 5)d \tag{2.9}$$

For example, let us consider the construction of a CSkA with $n = 32$ ranks, for which there results, in accordance with (2.8), $b_{opt} = 4$ and, in accordance with (2.9), $L_{opt} = 27d$, which represents a performance of almost 2.3 times better than that corresponding to the RCA of the same dimension.

On the other hand, starting from the observation that, depending on the position of the block within the adder, a carry generated in one of the blocks has to cross

a different number of logic levels, this leads to the idea that a variable number of ranks with serial propagation can be assigned to the blocks. As regards the number of blocks with different length and their dimension, analysis is required which, however most of the time makes simplifying hypotheses which are not confirmed by engineering practice [Parh00]. One of the factors which decisively influences the optimization of a CSkA design is the technological factor, which, unfortunately, is usually proprietary information until it becomes obsolete [ErLa04, Parh00]. Without strict formal support, there are successful experiments for rank partitioning in blocks of variable dimension, such as a CSkA of 20 bits divided into 5 blocks of 2, 5, 6, 5 and 2 ranks [HePa03]. The worst latency, in terms of logic levels, for this CSkA implies signal propagation through $2 \cdot 6 = 12$ gates of the median block (the time interval required for the propagation of a possible carry generated in the lsb rank of this block is considered equal to that required for the propagation of a possible carry from the lsb rank of the previous block, of 5 ranks, which has to cross the OR-AND pair, as well), to which is added the OR gate from the median block output, the AND-OR pair of gates associated with the more significant block of 5 ranks, as well as the three gates (AND, OR and EXCLUSIVE-OR) from the last segment, the one with 2 ranks. There results a total of 18 logic levels, substantially fewer than the 39 required by an RCA of the same dimension, and also fewer than the solution of the CSkA partitioned into five blocks of the same dimension, of 4 ranks, for which, applying (2.7), $L = 21$ gate levels is obtained.

Finally, we also mention the possibility to configure multilevel CSkA where the carry bypassing of more blocks is allowed [Parh00]. Thus, the skip signals from the AND gate outputs are applied to an AND gate from a superior level, the number of inputs corresponding to it being equal to that of the bypassed blocks. Also, the OR gate connected to the output of the bypassed blocks group has an additional input. In this way, there results a layered structure with two levels, for which the problem of the optimum configuration remains open.

2.2.6 Carry-Select Adder

To achieve the target of logarithmic latency, typical for the CLA principle, for the CSkA adders, we appeal to parallel computations. The same fundamental idea, but applied in a different way, stands at the basis of the configuration of some adders, in which addition is performed by conditioning the sum through the carry values, the so-called conditional-sum addition algorithm [Parh00]. Essentially, this algorithm foresees that blocks of ranks compute the sum simultaneously, in parallel, in two variants, accepting a priori the values 0 and 1 respectively, for the carry-in in the given blocks, and selecting the correct sums from their outputs, subsequently, when the real, true value of the carry becomes known. Thus, the sum is computed on blocks in advance, in two variants, the correct sum being chosen by the value of the carry when it arrives, through serial propagation, at each pair of blocks. There are two categories of adders which function on the basis of this algorithm, namely those

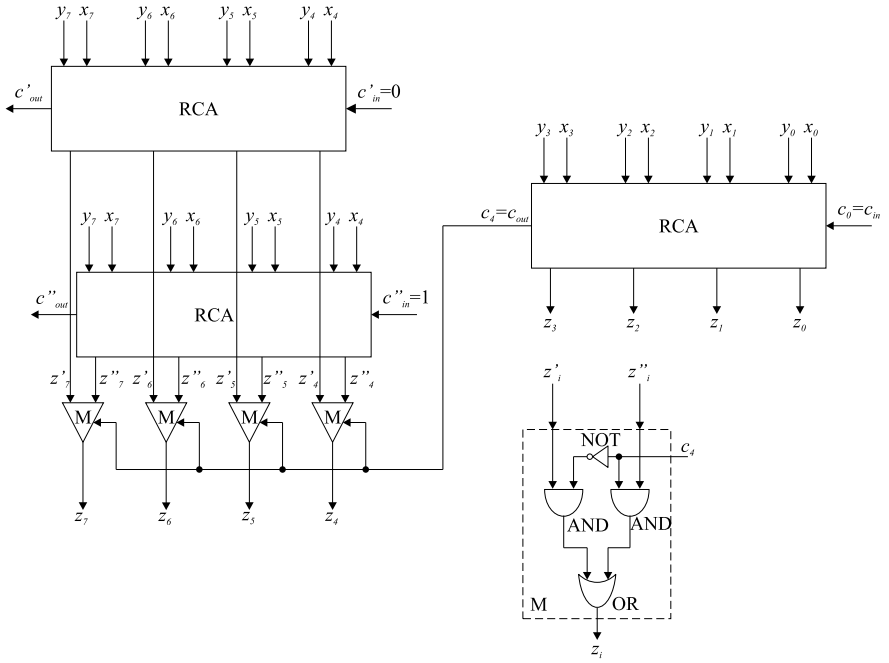


Fig. 2.25 Block diagram of an 8-bit CSeA

having the sum selected through the value of the carry, the so-called carry-select adders (for short CSeA, for the same reasons explained in the previous section on CSkA), and those having the sum conditioned by the value of the carry, the so-called conditional-sum adders (CSuA), which will be analyzed in the next section. The two types of adders have similar principles, but they differ in their implementation.

Analyzing the CSeA adders, we consider, without loss of generality, an RCA on n bits which we divide into two parts, and the part corresponding to the $n/2$ least significant ranks directly computes the $n/2$ bits of the sum. The other part of the $n/2$ more significant ranks is substituted by two RCA adders, each of which computes, at the same time as the RCA for the least significant part, the sum and the carry-out in two different situations, namely by applying value 0 to the carry-in input at one of the RCAs, and value 1 to the other RCA respectively. The three adders, functioning in parallel, finish the computations approximately at the same time, when the real carry-out generated by the RCA corresponding to the least significant bits of the sum becomes known, and it selects, out of the two previously computed values of the sum (and of the carry-out), only the correct one. Thus, for instance, Fig. 2.25 presents a CSeA on $n = 8$ ranks consisting of three adders on 4 ranks, assumed to be of RCA type. It can be observed that the more significant bits of the sum are computed in parallel for two cases, namely, for the case when $c'_{in} = 0$, resulting in the bits from z'_4 to z'_7, and for the case when $c''_{in} = 1$, resulting in the bits from z''_4 to z''_7. The selection between the two sum binary subvectors is made under the control of the carry-out

(c_4) generated by the RCA adder corresponding to the least significant part of the sum by means of a layer of multiplexers M, one of which is detailed at a gate level in Fig. 2.25. If this new adder configuration is compared to the RCA reference, we can observe that to the delay corresponding to the worst serial propagation on the least significant RCA (generally $2n/2 = n$ gate levels) there is added the delay on multiplexer M (which is considered to be of two logic levels). Otherwise, the latency expressed in logic levels is equal to $(n + 2)$ as compared to $2n$ which corresponds to the RCA of the same dimension. The performance difference between the two solutions is more obvious when n takes larger values, tending to reduce the latency to a half (of course, this estimation is too optimistic when the three adder segments are not of RCA type). Mention should be made that the above estimation shall be amended because the carry-out signal generated by the least significant RCA adder (in our case, c_4) generally controls a large number of multiplexers' select input, which, with certain technologies, may lead to serious performance degradation. On the other hand, if cost is estimated in terms of number of gates, the doubling of the RCA adder for the more significant sum bits and the multiplexers layer, roughly tends to double the cost for the CSeA as compared to that for the RCA. But these estimations are decisively influenced by the technological factor [HePa03].

The carry-select principle applied above in the adder partitioning into halves can be extended by dividing the adder into quarters, or by continuing the division, in which way further accelerations in the sum computation can be achieved. Moreover, the segments into which the adder is divided shall not contain the same number of ranks, it being of variable dimension. Such a technical solution starts from the observation that, on a doubled block of b ranks, we have a latency given by $2b$ gate levels to which are added the levels of the logic for obtaining the inter-block carries. For the synthesis of this latter logic, if we are to refer to the carry-outs, c'_{out} and c''_{out}, for the circuit in Fig. 2.26 [HePa03], we start from the observation that $c''_{out} = c'_{out}$ $\underline{or}$ $p_7p_6p_5p_4$, where we have $p_i = x_i$ $\underline{or}$ y_i, for $i = 4$ to 7. In these conditions, by considering exhaustively the configurations corresponding to the three variables (c''_{out}, c'_{out}, c_4), from the eight possible ones, we may exclude the triplets (0,1,0) and (0,1,1), whose appearance is impossible, and which can be used to obtain the minimum form of the Boolean equation for the inter-block carry-out function, denoted by c_{out}. If we take into consideration that, only for the triplets (1,0,1), (1,1,0) and (1,1,1), c_{out} becomes 1, then, the minimized Boolean equation for c_{out} is of the form: $c_{out} = c'_{out}$ $\underline{or}$ $c''_{out}c_4$. In consequence, in the synthesis of the logic circuit for the generation of the inter-blocks carry a pair of AND-OR logic circuits is involved, in other words, two levels, which are added to the $2b$ levels corresponding to a block. Since $(2b + 2 = 2(b + 1))$, it follows that the next block in the direction of the carry propagation can have $(b + 1)$ ranks, one rank more than the previous one. Thus, let us extend the adder of $n = 8$ ranks from Fig. 2.25, obtaining the configuration from Fig. 2.26, which presents, for the ranks from 4 to 18, the part of the diagram which is doubled and consists of three blocks of parallel adders (PA) that have, towards the more significant bits, one more rank each. Each block is doubled, having a superior stage where the carry-in is 0, and an inferior one (for the sake of clarity, at the latter stage the inputs have been omitted, they being the same

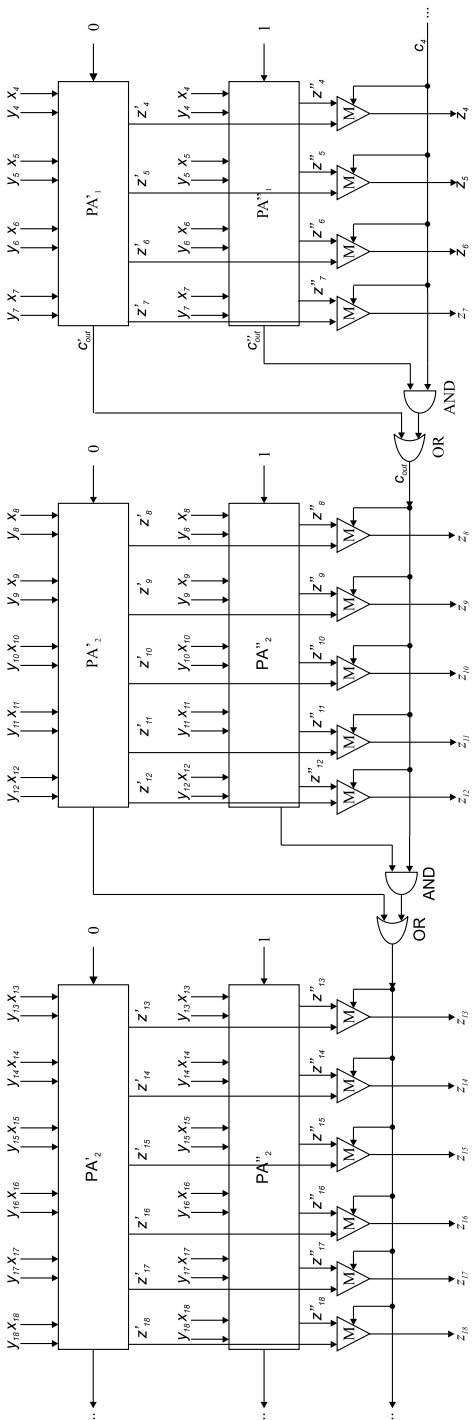


Fig. 2.26 Segments interconnection for a CSeA

as those from the superior stage) where the carry-in is 1. The choice of the sum subvectors is achieved through the M multiplexers, which select the result z_i' or z_i'' depending on the value of the carry-out which arrives from the previous block. The first block, the one corresponding to the least significant sum bits z_0 to z_3 (which is not doubled), has not been presented in the figure, for the sake of clarity, it being identical to the block from Fig. 2.25. Mention should be made that, this time, the constructive principle which has to be restricted in RCA, has not been restricted, it being considered a general PA, no matter the method which stands at the basis of its synthesis. The moment the carry-out (c_4) from the first block is known, new sum binary subvectors can be selected depending on the activation of the block carry-out signals, whose generation is assured through the AND-OR pairs of gates.

In a similar way, carry-select adders can be configured from blocks of variable dimension, in a similar way to carry-skip adders. In the same way also, both of them can have multilevel structures. In case of carry-select adders, a possible such construction divides the n ranks into blocks of $n/4$ ranks, of which that corresponding to the least significant ranks is not doubled, and the other three are doubled, functioning as already presented, with the carry-in inputs connected to 0, and logic 1 respectively [Parh00]. The implementation on two levels refers to the multiplexing part, more precisely, to the selection of the more significant half of the sum bits. Such a configuration may lead to a favorable implementation of the arithmetic pipeline but, in this case, as well, the option to appeal to the multilevel constructive principle is decisively influenced by the manufacturing technology [ErLa04, Parh00].

2.2.7 Conditional-Sum Adder

As presented in the previous paragraph, the CSuA adders are related to the CSeA ones, having at their basis the same algorithm, and they could be looked upon as a generalization of the latter, more exactly of the multilevel variant [YeJe03]. Thus, on the first level one bit from each operand is added, resulting in two pairs of carry-sum values, one corresponding to the case when the carry-in in the given rank is 0, the other to the case when the carry-in is 1, similar to the model of the two blocks of the CSeA, with the stipulation that now the blocks are reduced to only one rank. On the second level, blocks made up of two ranks are taken into account. Within them the corresponding carry-sum pair of values is chosen by means of the real value of the interface carry between the two ranks of the block. On the third level, the blocks now have four ranks with the same selection described above achieved through the real value of the interface carry between the blocks made up of two ranks. This procedure will continue in a binary tree type manner until the dimension of a block, doubled at each level, is equal to that of the operands. In this way, a hierarchical construction of $\log_2 n$ height (n being the operands' dimension) will be obtained to which one more logic level for the forming of the generation (g) and propagation (p) variables is added.

Sum bits	Carry bits
$z_0 = x_0 \oplus y_0 \quad (c_{in} = c_0 = 0)$	$c_1 = x_0 y_0 = g_0 \; (c_{in} = c_0 = 0)$
$z_1 = (x_1 \oplus y_1)\overline{c_1}$ or $(\overline{x_1 \oplus y_1})c_1$	$c_2 = x_1 y_1 \overline{c_1}$ or $(x_1$ or $y_1)c_1 = g_1 \overline{c_1}$ or $p_1 c_1$
$z_2 = (x_2 \oplus y_2)\overline{c_2}$ or $(\overline{x_2 \oplus y_2})c_2$	$c_3 = x_2 y_2 \overline{c_2}$ or $(x_2$ or $y_2)c_2 = g_2 \overline{c_2}$ or $p_2 c_2$ $\overline{c_3} = (\overline{x_2 y_2}$ or $c_2)(\overline{x_2$ or y_2} or $c_2) =$ $= \overline{x_2 y_2} c_2$ or $\overline{x_2 \text{ or } y_2} c_2 = \overline{g_2} c_2$ or $\overline{p_2} c_2$
$z_3 = (x_3 \oplus y_3)\overline{c_3}$ or $(\overline{x_3 \oplus y_3})c_3 =$ $= ((x_3 \oplus y_3)\overline{x_3 y_3}$ or $(\overline{x_3 \oplus y_3})x_3 y_3)\overline{c_2}$ or $((x_3 \oplus y_3)(\overline{x_3$ or $y_3})$ or $(\overline{x_3 \oplus y_3})(x_3$ or $y_3))c_2 =$ $= ((x_3 \oplus y_3)\overline{g_3}$ or $(\overline{x_3 \oplus y_3})g_3)\overline{c_2}$ or $((x_3 \oplus y_3)\overline{p_3}$ or $(\overline{x_3 \oplus y_3})p_3)c_2$	$c_4 = x_3 y_3 \overline{c_3}$ or $(x_3$ or $y_3)c_3 =$ $= ((x_3 y_3 \overline{x_3 y_3})$ or $(x_3$ or $y_3)x_3 y_3)\overline{c_2}$ or $(x_3 y_3 (\overline{x_3}$ or $\overline{y_3})$ or $(x_3$ or $y_3)(x_3$ or $y_3))c_2 =$ $= (g_3 \overline{g_3}$ or $p_3 g_3)\overline{c_2}$ or $(g_3 \overline{p_3}$ or $p_3 p_3)c_2$

Fig. 2.27 Logical equations for a 4-bit CSuA synthesis

For the synthesis of the hierarchical structure made up of a CsuA, it is necessary to rewrite the Boolean equations corresponding to the various ranks to allow the carry signals to act as they did as control signals in the multiplexers of the CSeA (refer to the signal carry c_4 from Fig. 2.25). Thus, the following equations will be applied: $z_i = x_i \oplus y_i \oplus c_i = (x_i \oplus y_i)\overline{c_i}$ or $(\overline{x_i \oplus y_i})c_i$, and $c_{i+1} = \overline{x_i} y_i c_i$ or $x_i \overline{y_i} c_i$ or $x_i y_i \overline{c_i}$ or $x_i y_i c_i = x_i y_i \overline{c_i}$ or $(x_i$ or $y_i)c_i = g_i \overline{c_i}$ or $p_i c_i$, where c_i acts in the same way as c_4 from the detail of multiplexer M (Fig. 2.25). These specifications being made, Fig. 2.27 presents the Boolean equations that are important for the sum bits, and for the carry bits corresponding to the first four ranks of a CSuA. The derivation of the expressions is done to obtain the forms of the control signals for a multiplexer M, first of all, as a function of c variables, and, subsequently, as a function of the variables p and g. Consequently, we mention the covering in the expression of $\overline{c_3}$ of the term $\overline{x_2}\,\overline{y_2}$ by $\overline{x_2 y_2}\,\overline{c_2}$ and $\overline{(x_2$ or $y_2)c_2}$. On the basis of the equations from Fig. 2.27, Fig. 2.28 presents the synthesis, at the level of logic gates, of a CSuA which generates the sum bits from z_0 to z_3. The synthesis levels of the constructive hierarchy of a CSuA are highlighted; these are then represented schematically, in Fig. 2.29, for the extension to an adder with $n = 8$ ranks. In this figure the blocks are identified by means of two indexes, B_{ij}, where i identifies the block level on the vertical, the numbering being made downwards starting with 0, and j allows the identifies of the blocks within the same level, the numbering being made from right to left starting with 0. Studying the details from Fig. 2.28 of some of the blocks of the schematic structure from Fig. 2.29, it can be observed that on block level 0 we have only a single level of gates, and, after this, the levels have similar constructions, of multiplexer type circuit, being implemented through a layer of AND gates (AND level) and one of OR gates (OR level). The connections between the blocks of the structure from Fig. 2.29 can be easily found in the details from Fig. 2.28; the figure does not contain other notations which might have made it to intricate. We also note that the number of block levels with multiplexer type structure, excepting the one with $i = 0$, is $\log_2 n$, which allows the simple estimation of a CSuA's performance, as well as the estimation of its cost.

As an example, Fig. 2.30 presents the addition of some operands, without loss of generality, on 16 ranks, the block levels i being pointed out, with c_{in} denoting the

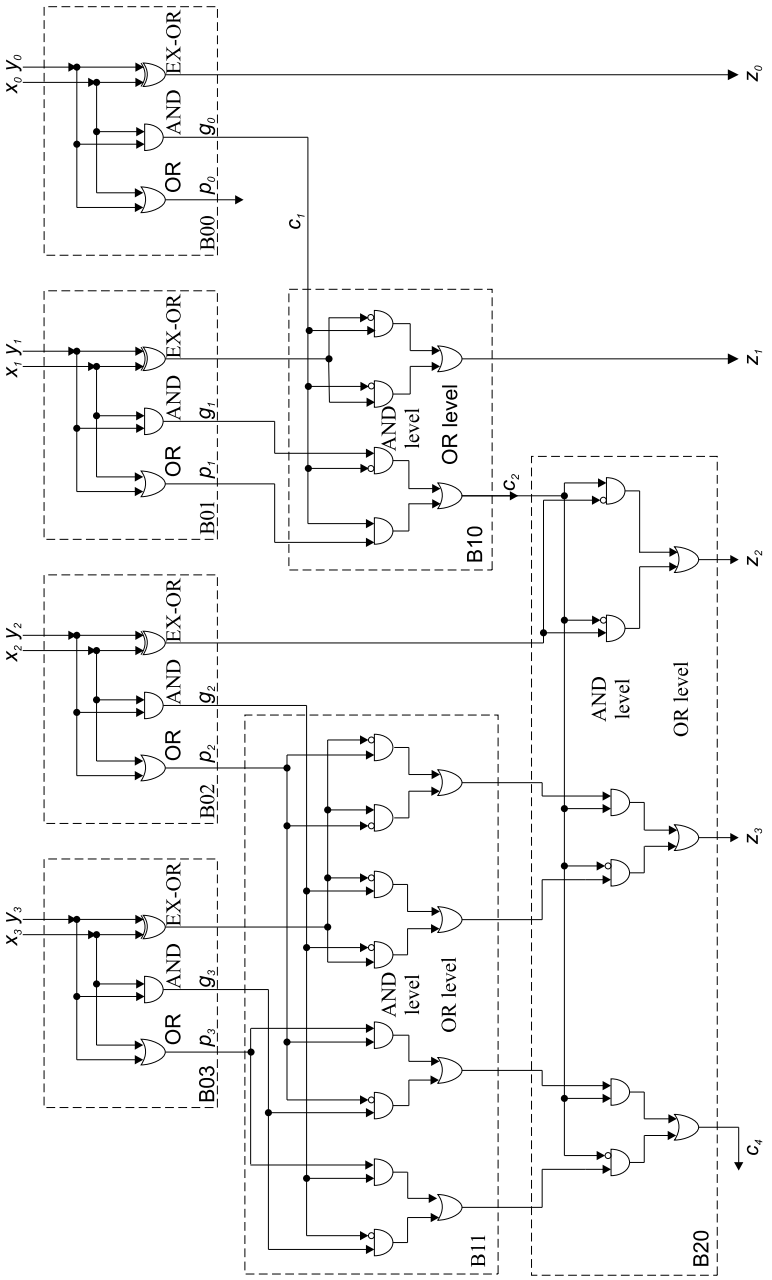


Fig. 2.28 Gate level synthesis for a 4-bit CSuA

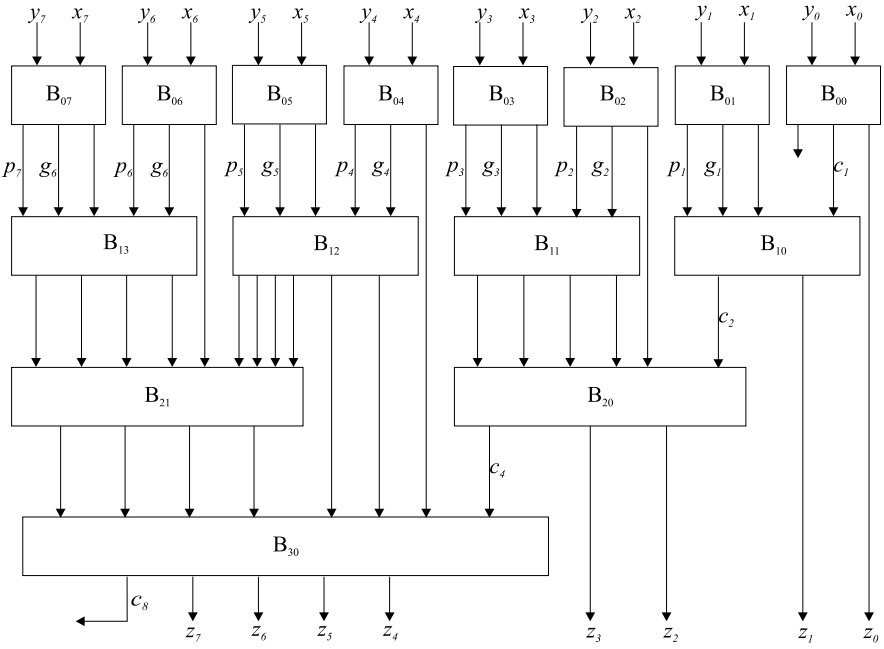


Fig. 2.29 Block diagram of an 8-bit CSuA

Range		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X		1	0	1	0	1	1	0	1	0	0	0	1	0	1	0	1
Y		0	0	1	1	0	1	1	0	1	0	1	0	1	1	0	1
Block level	Carry in	C S	C S	C S	C S	C S	C S	C S	C S	C S	C S	C S	C S	C S	C S	C S	C S
i=0	c_{in} =0	0 1	0 0	1 0	0 1	0 1	1 0	0 1	0 1	0 1	0 0	0 1	0 1	0 1	1 0	0 0	1 0
	c_{in} =1	1 0	0 1	1 1	1 0	1 0	1 1	1 0	1 0	1 0	0 1	1 0	1 0	1 0	1 1	0 1	
i=1	c_{in} =0	0 1	0	1 0	1	1 0	0	0 1	1	0 1	0	0 1	1	1 0	0	0 1	0
	c_{in} =1	0 1	1	1 1	0	1 0	1	1 0	0	0 1	1	1 0	0	1 0	1		
i=2	c_{in} =0	0 1	1	0	1	1 0	0	1	1	0 1	0	1	1	1 0	0	1	0
	c_{in} =1	0 1	1	1	0	1 0	1	0	0	0 1	1	0	0				
i=3	c_{in} =0	0 1	1	1	0	0	0	1	1	0 1	1	0	0	0	0	1	0
	c_{in} =1	0 1	1	1	0	0	1	0	0								
i=4	c_{in} =0	0 1	1	1	0	0	0	1	1	1	1	0	0	0	0	1	0

Fig. 2.30 CSuA manner addition example of some 16-bit operands

input carry in a block, with each block level required to have both values 0 and 1. Also, C and S denote the carry and sum bits, and, for each level, the blocks have been delimited through double partition lines. Starting with $c_{in} = 0$ in rank 0, it can be observed that at each block level crossing, the number of the correct sum bits is doubled.

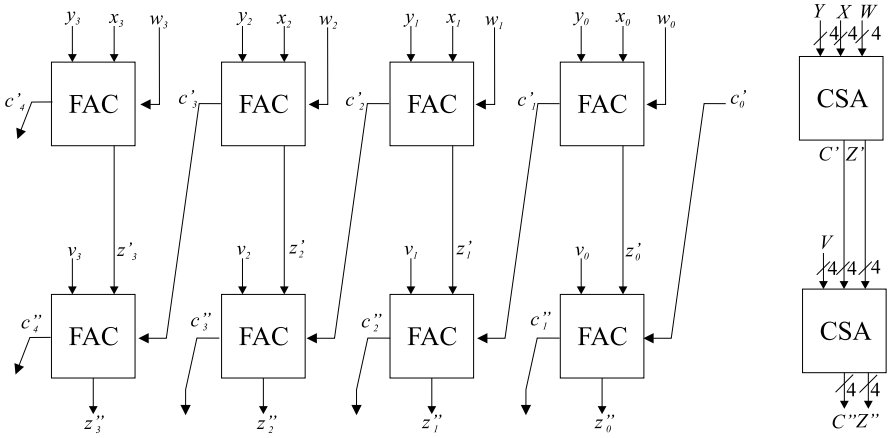


Fig. 2.31 FACs interconnections into a CSA structure

2.2.8 Carry-Save Adder

We have left the carry-save adder (CSA), which, as a matter of fact, is applied on a large scale, to be presented at the end of the chapter dedicated to the adders, because it does not realize a sum of two operands in the conventional meaning, but facilitates the addition of several operands (multioperand addition), as is required, for instance, in a multiplication operation [HuEr05]. This is the reason why the problems which are specific to this type of adder, are presented in extenso, in Sects. 3.6 to 3.10. Now, we introduce the CSA by mentioning only that, having operands of n bits, it is made up of n full adder cells (FAC) which are not connected to each other as an RCA, instead being disjoint. The carry-in inputs thus remain unconnected, and a third operand can be supplied to them, according to the model presented in Fig. 2.31. Thus, the first CSA level performs the addition of the operand vectors W, X and Y, generating two vectors, the sum Z', and carry C' vectors. After this CSA level, there can follow others, to which the carry vector has to be applied shifted by one binary rank to the left. Thus, the flow addition of several operand vectors [VeEN02] is possible, as shown schematically in Fig. 2.32 [Haye98], where, without loss of generality multioperand addition is used to perform binary multiplication. Without insisting upon the characteristics of this last operation, we shall consider the numbers represented by the multiplier X and the multiplicand Y to be unsigned integers. Following decoding of the multiplier X and one bit product forming, of the type $M_i = x_i Y 2^i$, through multiplicand gating, there follows the addition of these one bit products, for instance, from M_0 to M_5. In fact, this case has been presented to exemplify the implementation of multioperand addition by means of a CSA tree structure, each adder supplying a pair of carry and sum vectors.

Mention should be made that a certain bit of the sum vector is obtained without carry propagation, adding three bits no matter the result of this operation executed in the neighbor rank on the right. What is not sufficiently clear in Fig. 2.32 is how the

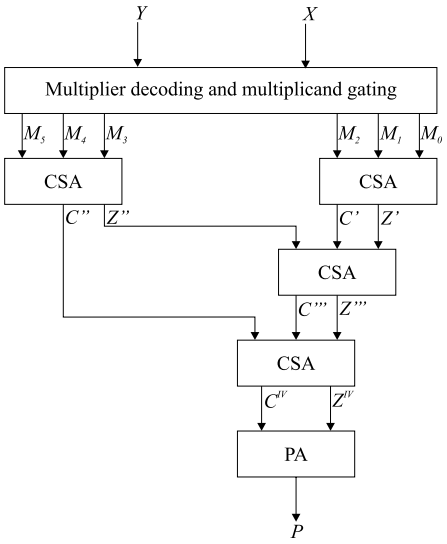


Fig. 2.32 Block diagram of a binary multiplier with CSA levels

connections between the CSAs on various levels for carry vector transmission are made. To allow the necessary carry propagation, this vector is applied shifted by one rank to the left, as suggested in Fig. 2.31. Another aspect that has to be pointed out in connection with multioperand addition through CSA levels is that the last carry-sum pair is added, conventionally, through one of the parallel adders (PA) presented in the previous sections, and in this way product $P = XY$ is obtained. To highlight the carry connections for the simpler case when the operands have the dimension $n = 4$, Fig. 2.33 presents the two CSAs detailed at FAC level. The functioning of the structure from Fig. 2.33 is exemplified in Fig. 2.34 for the operands $X = 13_{10}$ and $Y = 14_{10}$. The CSA adder from the first level adds the partial products M_0, M_1 and M_2, and the carry vector (C') is applied to the next level shifted by one rank to the left, which is equivalent to the doubling of its value ($2C'$). Then, the next CSA adds, in carry-save mode, the sum (Z') and carry ($2C'$) vectors, that have come from the first level, to the fourth product of one bit, M_3, and thus there is obtained the pair of sum (Z'')–carry (C'') vectors, which has to be added conventionally. In case of the structure from Fig. 2.33, this last operation is performed by an RCA having as inputs Z'' and shifted C'' ($2C''$) vectors.

The sections of Chap. 3 refer to performance and cost aspects that are specific to CSAs. They also contain various configurations that can be produced with this category of adders.

2.2.9 Binary Adders with Parity Control

Looking for solutions to improve performance and/or cost, adder designers have combined the above-mentioned constructive principles, arriving at various hybrid

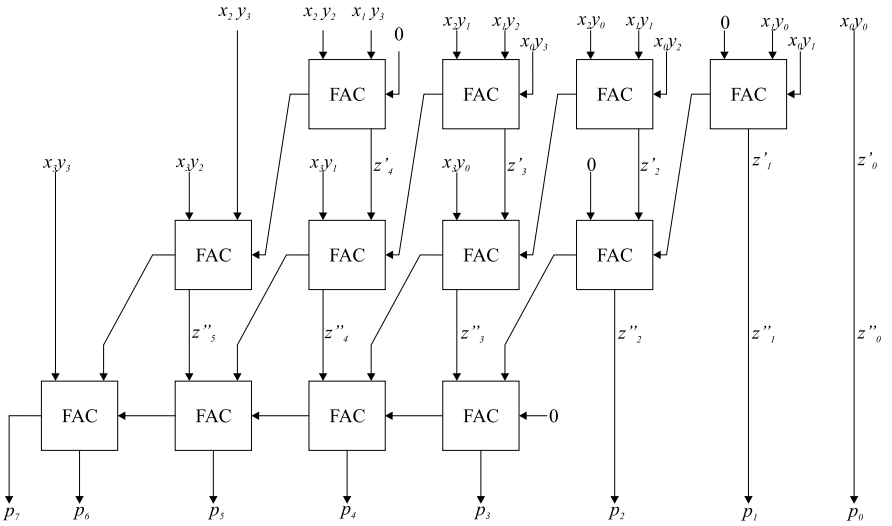


Fig. 2.33 Detailed FACs interconnections for a 4-bit operands binary multiplier having two CSA levels and an RCA level

Fig. 2.34 4-bit operands multiplication example using two CSA levels

$x_3x_2x_1x_0$
$X = 1\ 1\ 0\ 1$
$Y = 1\ 1\ 1\ 0$

$M_0 = x_0\ Y2^0 = 0\ 0\ 0\ 0\ 1\ 1\ 1\ 0$
$M_1 = x_1\ Y2^1 = 0\ 0\ 0\ 0\ 0\ 0\ 0$
$M_2 = x_2\ Y2^2 = 0\ 0\ 1\ 1\ 1\ 0$

$Z' = 0\ 0\ 1\ 1\ 0\ 1\ 1\ 0$
$C' = 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0$
$Z' = 0\ 0\ 1\ 1\ 0\ 1\ 1\ 0$
$2C' = 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0$
$M_3 = x_3\ Y2^3 = 0\ 1\ 1\ 1\ 0$

$Z'' = 0\ 1\ 0\ 1\ 0\ 1\ 1\ 0$
$C'' = 0\ 0\ 1\ 1\ 0\ 0\ 0\ 0$
$Z'' = 0\ 1\ 0\ 1\ 0\ 1\ 1\ 0$
$2C'' = 0\ 1\ 1\ 0\ 0\ 0\ 0\ 0$
$P = 1\ 0\ 1\ 1\ 0\ 1\ 1\ 0$

combinations, of RCA-CLA type, but also of CLA-CSeA, CLA-CSuA, CSeA-MA types, a.o. [Kore02, Kuli02, Parh00]. Often, in these structures, such attributes as reliability, maintainability, availability, and, generally, dependability are relegated to secondary status, or they are grafted on to solutions which are optimized for performance-cost [ALRL04]. We point out that in order to obtain "optimized" solutions with respect to these desiderata, which are of growing importance, it is necessary to address them as early as possible in the design process [COPR06]. On the other hand, since adders represent one of the most often used structural elements of a computer, extrapolating Amdahl's law [HePa03] to dependability aspects, we may say that design solutions favorable to the above mentioned attributes need to be

applied to adders. This is the reason why, at the end of this chapter, we refer to one of the multiple existing methods to facilitate checking of such devices.

To introduce the promised strategy, we mention that parity control and its generalizations are widely used in the checking of information transfer and storage operations [AbBF90]. But, different methods are used for arithmetic operations, such as, among others the use of residual codes [VeEN02, RaTy98]. Consequently, the tendency to apply checking with reference both to information transfer and storage operations, and to arithmetic operations, has been found to be of major interest. Thus, we present a possible approach, namely the extension of parity control code to addition through so-called parity-checked adders (PCA). For the construction of such a device, there will be attached to each of the two operands, X and Y, as well as to the sum result Z, a parity bit, estimated on the basis of the following relations:

$$\begin{aligned} x_p &= x_{n-1} \oplus x_{n-2} \oplus \cdots \oplus x_i \oplus \cdots \oplus x_1 \oplus x_0 \\ y_p &= y_{n-1} \oplus y_{n-2} \oplus \cdots \oplus y_i \oplus \cdots \oplus y_1 \oplus y_0 \\ z_p &= z_{n-1} \oplus z_{n-2} \oplus \cdots \oplus z_i \oplus \cdots \oplus z_1 \oplus z_0 \end{aligned} \tag{2.10}$$

where $\oplus$ represents the EXCLUSIVE-OR operator (equivalent to the modulo 2 sum).

Since we have from above $z_i = x_i \oplus y_i \oplus c_i$, by making the substitution in (2.10) for each rank, from 0 to $(n-1)$, the following will be obtained:

$$z_p = x_p \oplus y_p \oplus c_{n-1} \oplus c_{n-2} \oplus \cdots \oplus c_1 \oplus c_0 \tag{2.11}$$

Relation (2.11) stands, predictively, at the basis of the generation of the sum parity bit, which is compared to that actually formed of the sum bits and computed through the equation for z_p from (2.10). The synthesis of the checking thus described leads to the so-called parity checker diagram, essentially made up of an EXCLUSIVE-OR (EX-OR) tree. In fact, we have two subtrees, one of them implementing (2.11) (EX-OR tree 1), while the second implements the generation of the sum parity bit z_p based on the equation for z_p from (2.10).

Figure 2.35 presents an adder, which, for the sake of example, is of RCA type (but it may be, with the adequate amendments, of any type presented above) and the attached parity checker. The outputs of the two subtrees, EX-OR tree 1 and EX-OR tree 2, are passed to the EX-OR gate, which, in case of inequality, signals the occurrence of an error. The question is whether the parity-checker from Fig. 2.35 is sufficient to detect all the singular errors, which, it is well-known [RaFu89], represents the target of the parity control code. It is sure that any singular fault regarding the part of the circuits that are specific to the generation of a sum bit z_i (which excludes the circuits for the evaluation of the carry bit c_{i+1}—refer also to the implementation versions in Fig. 2.4) bring about an error which changes the parity given by z_p in (2.10) as compared to that which is predictively computed through (2.11), and, consequently, will be detected by determining ERROR = 1.

But the situation changes when the fault occurs in the chain of circuits which enables carry propagation, such as, for instance, the stuck-at-0 of NAND gate output [AbBF90] which generates carry c_{i+1} (Fig. 2.4a,b). In such a case, the faultmay

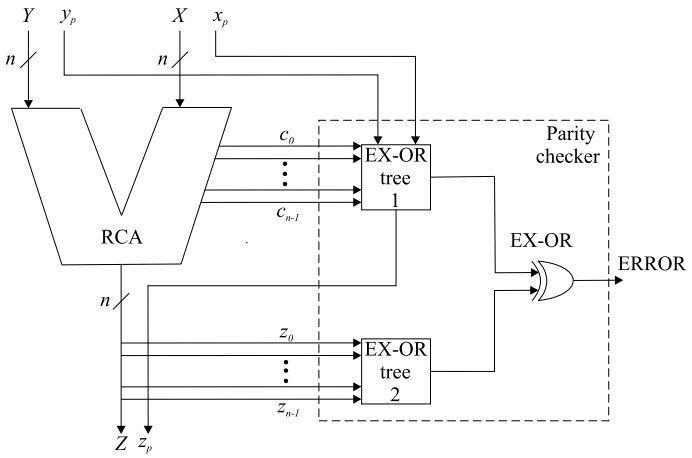


Fig. 2.35 Block diagram of a parity checked adder

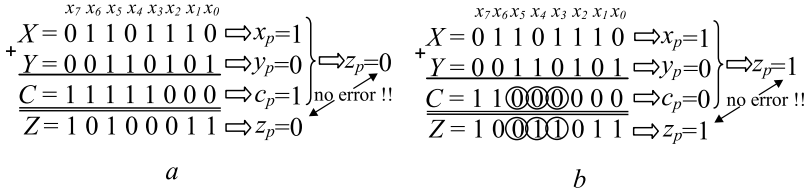


Fig. 2.36 RCA binary addition example with masked singular fault presence on the carry chain

bring about the error not only at the given carry bit, but also at the sum bit, successively, and, due to propagation, more sum and carry bits can be affected. But mention should be made that their total number is always even, because the number of the erroneous sum and carry bits is equal. Figure 2.36 presents an example of addition of two unsigned integer numbers, $X = 110_{10}$ and $Y = 53_{10}$, and, in the correct addition (Fig. 2.36a), the sum $Z = 163_{10}$ results. The sum parity bits, computed by means of relations (2.10), and (2.11) (in this case, we have used $z_p = x_p \oplus y_p \oplus c_p$, where we noted $c_p = c_{n-1} \oplus c_{n-2} \oplus \cdots \oplus c_i \oplus \cdots c_1 \oplus c_0$), result, in both cases, $z_p = 0$. On the other hand, accepting the stuck-at-0 fault of the final NAND gate which generates carry c_3, Fig. 2.36b presents, on the addition of the same operands, the number of erroneous bits brought about by the given fault. Thus, erroneous c_3 determines the introduction of errors into s_3 and c_4, and erroneous c_4 determines the introduction of errors into s_4 and c_5, the latter affecting only s_5, not c_6, because in rank 5, through $x_5 = y_5 = 1$, the conditions for the generation of a new carry are created, no matter the propagation of the carry, in our case, erroneous, which came from the previous ranks. The presence of the fault modifies the bits c_p, as well as z_p as compared to the situation from Fig. 2.36a. Consequently, the two values computed for z_p are again equal, no malfunction being signaled, and the fault is not detected. Thus, the parity checker diagram is not sufficient to assure correct-

ness of the adder in case of singular faults through the parity control code. Mention should be made that the analysis made for the stuck-at-0 fault can also be extended to stuck-at-1 faults (when a pair of bits (0,0) of the operands stops the propagation of the erroneous carry bit), as well as to other types of faults [RaFu89]. To highlight the possible singular faults on the chain of circuits which implement the carry propagation, it is necessary to appeal to supplementary circuits added to the parity checker. All these circuits, in fact redundant related to an economic design, are meant to assure the checking of the adder. There are two technical solutions for the additional circuits, namely, carry chain duplication, and changing the adder into a special one, i.e. the so-called carry-dependent sum adder (CDSA) [RaFu89]. Regarding the first solution, as its name shows, this provides, instead of one route for carry transmission, the provision of two such chains of circuits. The part of the circuits involved in the sum bits generation is not doubled. The use of this procedure is due to the fact that the fault being singular, it will affect the functioning of only one of the two routes, so that its effect may be detected by connecting to the parity checker the carries generated by only one propagation chain. Thus, Fig. 2.37a presents the duplication applied to an RCA and Fig. 2.37b, presents the same principle applied to a hybrid solution, CLA-RCA: more precisely, the carry chain, made in RCA mode, is attached, together with the parity checker, to a CLA full adder [RaFu89]. In Fig. 2.37a, the following notations have been used: SC (sum circuit) for that part of the circuits of a FAC which generates a sum bit, CC (carry circuit) for that part of the circuits of a FAC which generates a carry bit, and CC* for the duplication of a CC. Otherwise, for any combination SC + CC, and SC + CC* respectively, a FAC is obtained. As mentioned above, the singular fault on the carry chain can be detected by connecting only one of the carry vectors to the parity checker. On the other hand, in Fig. 2.37b, the following notations have been used: AC, for an adder cell which implements the sum function and generates g and p variables (refer to Fig. 2.18) which are specific to a CLA structure, and CC* with the same significance as above, representing the duplication of that part of a FAC which generates a carry bit.

Regarding the CDSA solution, it is necessary to redesign the fundamental structure element of the adder, represented by the adding cell. The approach to its synthesis is an additional reason for taking into account the design criteria favorable to dependability in as early a phase of design as possible. The basic idea consists of creating an "imbalance" between, on the one hand, the number of erroneous carry bits and, on the other hand, the number of erroneous sum bits. Thus, by adding the two numbers, there will result an odd value, which is detected by means of the parity control code. For clarity reasons, let us suppose that the singular fault determines, as the first erroneous bit, c_{i+1}, this bringing about the passing into error of the bits z_{i+1} and c_{i+2} and then, let us suppose that the error propagates as far as the bits c_{i+j} and z_{i+j}, the other carry and sum bits remaining unaffected. The total number of the erroneous bits, resulting from the example given in Fig. 2.36b, is even, which does not allow fault detection. Therefore, we shall induce the above mentioned "imbalance" by designing an adder cell (particularly, cell i) in such a manner that, when, because of the fault's presence, the carry-out bit (c_{i+1}) is erroneous, in

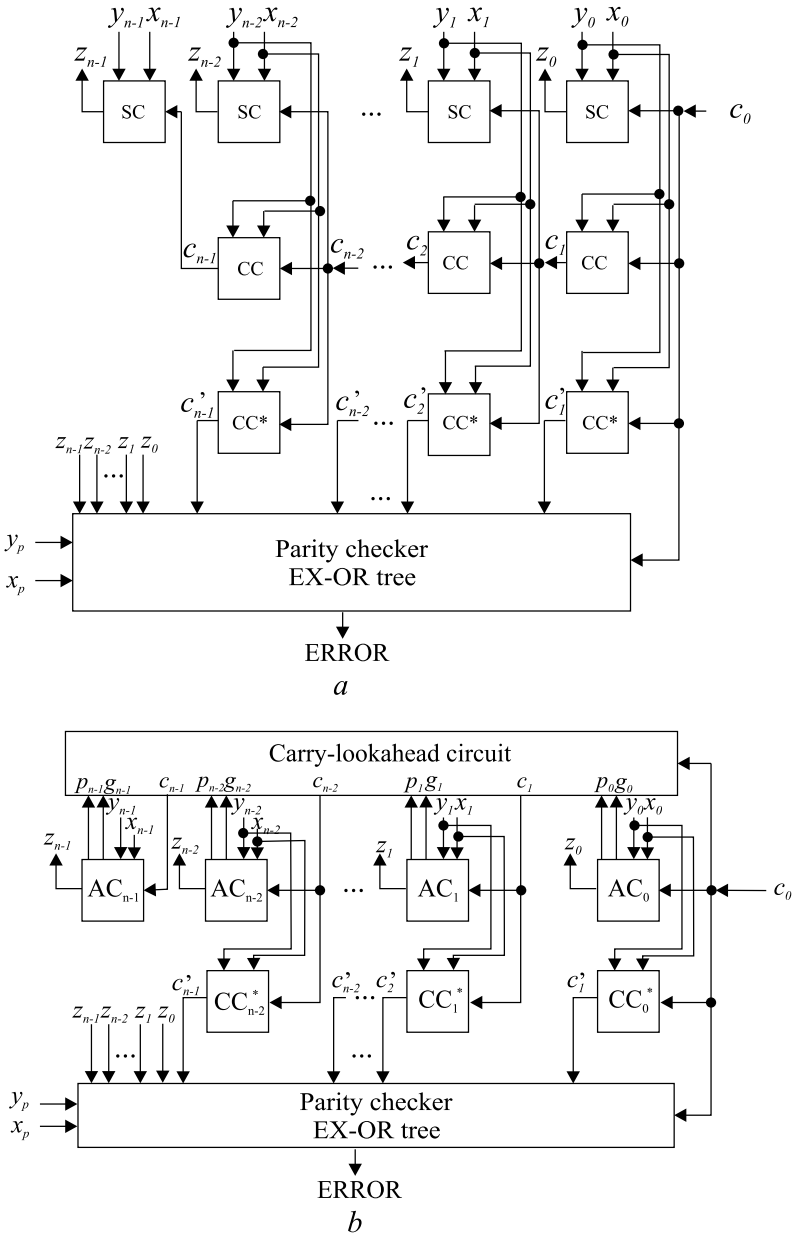


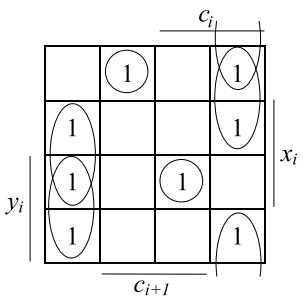
Fig. 2.37 Single fault detection solutions based on carry chain duplication

an artificial way, the sum bit (s_i) also becomes erroneous, a bit which otherwise would have been correct. Thus, the total number of erroneous bits becomes odd, ensuring the fault detection through the parity code. In this way, the design of an

Fig. 2.38 Truth table for the synthesis of a CDSA rank

y_i	x_i	c_i	c_{i+1}	z_i
0	0	0	0	0
0	0	0	(1)	(1)
0	0	1	0	1
0	0	1	(1)	(0)
0	1	0	0	1
0	1	0	(1)	(0)
0	1	1	(0)	(1)
0	1	1	1	0
1	0	0	0	1
1	0	0	(1)	(0)
1	0	1	(0)	(1)
1	0	1	1	0
1	1	0	(0)	(1)
1	1	0	1	0
1	1	1	(0)	(0)
1	1	1	1	1

Fig. 2.39 Minimization of the sum output's logical equation of a CDSA rank



adding cell for CDSA is done on the basis of the truth table from Fig. 2.38. The inputs are represented, first of all, by the variables x_i, y_i and c_i, there resulting two values for c_{i+1}, of which one is correct and the other is erroneous, the latter being marked by encircling. Then, the values for output z_i are deduced by assuming as inputs, besides the triplet (y_i, x_i, c_i), also c_{i+1}, in which way, when c_{i+1} is erroneous, z_i (marked by encircling) becomes incorrect, as well. For instance, let us consider the triplet $(y_i, x_i, c_i) = (0, 1, 1)$ which, in normal operation, determines the doublet $(c_{i+1}, z_i) = (1, 0)$, but when c_{i+1} becomes 0, in an erroneous way, $z_i = 1$ will be induced through the circuit, i.e. an incorrect value. Starting from the truth table from Fig. 2.38, and using the Karnaugh map from Fig. 2.39, following the favorable

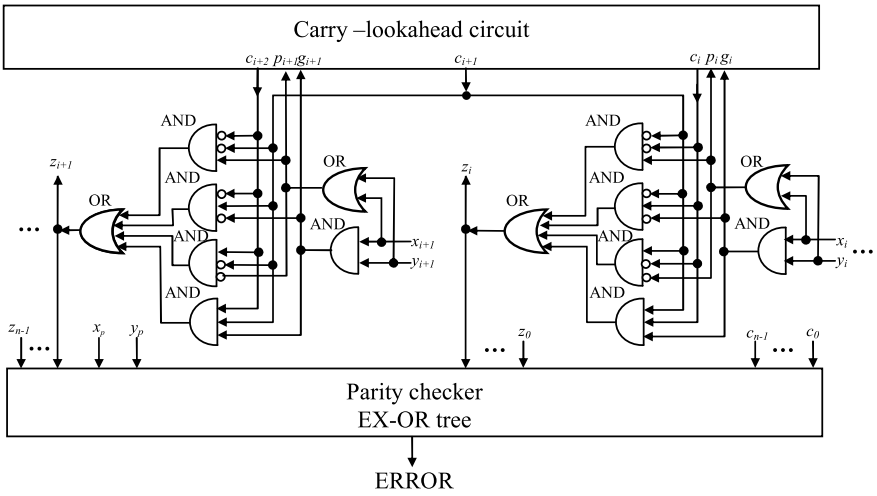


Fig. 2.40 Block diagram of a CDSA with gate level details for the ranks i and $i+1$

grouping of the binary units, the Boolean expression given below will be obtained for the sum output function z_i:

$$z_i = x_i \overline{c_i}\,\overline{c_{i+1}} \;\underline{or}\; y_i \overline{c_i}\,\overline{c_{i+1}} \;\underline{or}\; \overline{x_i} c_i \overline{c_{i+1}} \;\underline{or}\; \overline{y_i} c_i \overline{c_{i+1}} \;\underline{or}\; \overline{x_i}\,\overline{y_i}\,\overline{c_i} c_{i+1} \;\underline{or}\; x_i y_i c_i c_{i+1} \tag{2.12}$$

Taking into account the variables g_i and p_i which are specific to a synthesis of a CLA adder, relation (2.12) can be brought to the following form:

$$z_i = p_i \overline{c_i}\,\overline{c_{i+1}} \;\underline{or}\; \overline{g_i} c_i \overline{c_{i+1}} \;\underline{or}\; \overline{p_i}\,\overline{c_i} c_{i+1} \;\underline{or}\; g_i c_i c_{i+1} \tag{2.13}$$

Using (2.13) and appealing to an implementation with AND-OR gates, Fig. 2.40 presents, in one of the possible synthesis variants, the CDSA successive cells i and $(i+1)$ together with an acceleration circuit for the carry generation specific to a CLA, with the corresponding parity checker.

Chapter 3
Functional Analysis and Synthesis of Binary Multiplication Devices

3.1 Binary Multiplication Methods

The multiplication operation is generally performed over the operands made up of the multiplier and the multiplicand, denoted, for consistency, by X and Y. Undergoing computer processing the operands are represented by binary numbers which are considered, first of all for simplicity, integers without sign. The desired result consists of the product denoted by P, which, as is well known in conventional arithmetic, is obtained by repeatedly resorting to the fundamental operation of addition.

In the beginning we present the attempt to produce P by adding operand Y to itself X times. Translating this procedure in terms of hardware description language, which we shall use below, and whose characteristic elements are presented in Appendix A, we shall obtain the code sequence from Fig. 3.1 (adapted after [Haye98]). Generally, we consider the operands' dimension and implicitly that of the bus, assumed to be "split" into INBUS and OUTBUS, of 8 bits each. The multiplier device contains register CQ, for the initial storage of X, and register M, for the storage of Y during the entire length of the computation process. Product P will be stored in register CP, naturally provided to be of double dimension. At the presented registers' configuration is also added CM, a companion of M, whose initial content Y can be decremented. CM is periodically refreshed with the value Y, stored in M, at the beginning of each addition of Y. Following the operands' loading and the initialization of CP's content (which are elementary operations executed at two different CLOCK pulses labeled by BEGIN), it is tested whether one or both operands are zero (labeled by TEST1). This terminates the loop implied by the method. Each Y is added to CP's content unit by unit, under the control of CM's content (labeled by ADD and TEST2). Following the addition of a Y, the content of CQ is decremented and that of CM is restored (labeled SUB). Thus, X is gradually reduced, unit by unit, until it becomes 0 (determined through TEST3), and the operation ends with the returning, first of all, of the most significant part of the product and, then, at the following CLOCK pulse, of the less significant part of the product (labeled OUTPUT).

The analysis of the performance/cost impact reveals that this procedure does not suffer much, as far as the investment in circuitry is concerned, because the specific

M. Vlăduţiu, *Computer Arithmetic*,
DOI 10.1007/978-3-642-18315-7_3, © Springer-Verlag Berlin Heidelberg 2012

```
        multiplier 1
        declare register CQ[7:0], M[7:0], CM[7:0], CP[15:0];
        declare bus INBUS[7:0], OUTBUS[7:0];
BEGIN:  CP:=0, M:=INBUS;
        CQ:=INBUS, CM:=M;
TEST1:  if CQ=0 or M=0 then go to OUTPUT,
  ADD:  CP:=CP+1, CM:=CM-1;
TEST2:  if CM≠0 then go to ADD,
  SUB:  CQ:=CQ-1, CM:=M;
TEST3:  if CQ≠0 then go to ADD;
OUTPUT: OUTBUS[7:0]:=CP[15:8];
OUTPUT: OUTBUS[7:0]:=CP[7:0];
  END:
```

Fig. 3.1 Description of the binary multiplication as addition of the multiplicand to itself by a number of times equal to the value of the multiplier

$$
\begin{array}{r}
1\ 1\ 1\ 0 = y_3y_2y_1y_0 = Y \\
1\ 1\ 0\ 1 = x_3x_2x_1x_0 = X \\
\hline
1\ 1\ 1\ 0\ \ldots x_0Y2^0 \\
0\ 0\ 0\ 0\ \ldots\ldots x_1Y2^1 \\
1\ 1\ 1\ 0\ \ldots\ldots\ldots x_2Y2^2 \\
1\ 1\ 1\ 0\ \ldots\ldots\ldots\ldots x_3Y2^3 \\
\hline
1\ 0\ 1\ 1\ 0\ 1\ 1\ 0\ \ldots\ P=\sum_{i=0}^{3} x_iY2^i
\end{array}
$$

Fig. 3.2 "Paper and pencil" conventional binary multiplication example

part of this multiplication device (the companion register CM, the double dimension of CP, as well as the fact that all the registers that have the prefix letter C from "Count" are provided with the counting function) may be considered to be roughly in balance with the characteristics of other multipliers, as presented below. What restricts the area of application of this method, almost exclusively to the didactic field, is the prohibitive computation time which, leaving aside the input operands' loading and the returning of the result, requires, in terms of CLOCK pulses, an $O(n^2)$ complexity, n being the assumed dimension of the operands.

Unlike the above-mentioned method, we present below the conventional multiplication method, suggestively called "paper and pencil" [HePa03, Stal99, Haye98], which in its computer implementation version, does not appeal only to the iteration of the simple addition steps seen above, but also to some more complex addition-shift ones. But let us present, first of all, the classical multiplication operation, in binary version, this time appealing to an example which involves operands $X = x_3x_2x_1x_0 = 13_{10} = 1101_2$ and $Y = y_3y_2y_1y_0 = 14_{10} = 1110_2$, where x_i $(i = 0, 3)$ and y_j $(j = 0, 3)$ represent the binary digits of the two numbers (Fig. 3.2). The procedure is based on the previous forming of the one bit products of the operand $Y(x_iY)$, their progressive one bit left-shift (x_iY2^i) starting with the least significant bit of $X(x_0)$, and, finally, the addition of the one bit shifted products (x_iY2^i). This method is inadequate for computer implementation, because the intermediate storage of the one bit shifted products makes excessive use of the memory resource, requiring in practical cases large amounts of memory.

A first improvement regarding the case presented above consists of the sequential forming of a cumulative partial product, which is initially 0, and to which are successively added one bit products of Y adequately left-shifted (Fig. 3.3). Thus, it

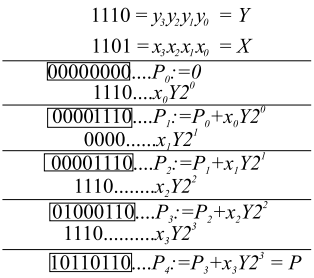


Fig. 3.3 Iterative binary multiplication example with unchanged position of partial and final products and left-shifted one bit products of the multiplicand

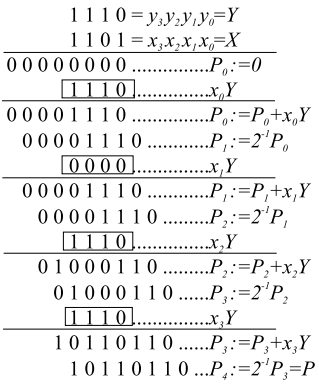


Fig. 3.4 Iterative binary multiplication example with unchanged position of one bit products of the multiplicand and right-shifted partial products

can be observed that, recurrently, the following iteration given for step $(i + 1)$ is used:

$$P_{i+1} := P_i + x_i Y 2^i \tag{3.1}$$

where P_{i+1} is the new partial product obtained by adding to the previous (P_i) the one bit product of Y, adequately left-shifted, this last operation being equivalent to multiplication by 2 ($x_i Y 2^i$). The result ($P = 10110110_2 = 182_{10}$) is obtained after the last iteration. During the operation, according to (3.1), each iteration requires the storage of only two numbers (P_i and $x_i Y 2^i$), avoiding the above-mentioned deficiency of the classical method. This procedure is characterized by the fact (seen in Fig. 3.3) that all partial products, including the final one, maintain their position unchanged, the one bit products of Y being progressively left-shifted.

A second improvement, equivalent to the above from the point of view of storage space, is also based on the same sequential forming of a cumulative partial product, which initially is 0 as well, but which has no fixed position. This time it is shifted to the right and to it are successively added one bit products of Y, with unchanged position throughout the entire operation (Fig. 3.4). Thus, it can be observed that, recurrently, the following compound iteration given for step $i + 1$ is used:

$$\begin{aligned} P_i &:= P_i + x_i Y \\ P_{i+1} &:= 2^{-1} P_i \end{aligned} \tag{3.2}$$

where P_{i+1} is the new partial product obtained by right-shifting (equivalent to division by 2, i.e $2^{-1}P_i$) the previous product P_i, to which has been added the one bit product of the non-shifted Y ($x_i Y$).

Although equivalent, as mentioned above, the two procedures differ regarding implementation, because the procedure based on iteration of (3.1) requires, at least at the first analysis, a $2n$ bits adder (n being, again, the dimension of the operands), while the procedure based on iteration of (3.2) requires a configuration artifice, presented in detail below, which will allow the conservation of the adder's rank number, maintaining it at the more reasonable dimension of n.

3.2 Sequential Sign-Magnitude Binary Multiplier

Supposing, without loss of generality, that the binary numbers are, this time, subunitary fractions represented in sign-magnitude, namely on 8 bits, let us present the procedure based on iteration (3.2) for this case. Thus, we have the following input operands:

$$\begin{aligned} X &= x_7 x_6 \dots x_i \dots x_0 = x_7 \sum_{i=0}^{6} x_i 2^{i-7} \\ Y &= y_7 y_6 \dots y_j \dots y_0 = y_7 \sum_{j=0}^{6} y_j 2^{j-7} \end{aligned} \tag{3.3}$$

where the most significant bits (x_7 and y_7) represent the numbers' signs and the other bits represent the magnitude part. The aim is to obtain the product result:

$$p = p_{15} p_{14} \dots p_k \dots p_1 p_0 = p_{15} \sum_{k=0}^{14} p_k 2^{k-15} \tag{3.4}$$

where, obviously, p_{15} is the sign given by the EXCLUSIVE-OR operation, $p_{15} = x_7 \oplus y_7$, the other binary digits being the magnitude part with the maximum dimension of 14 bits (ranging from p_{14} to p_1), to which is added the "harmless", regarding, the value of P, $p_0 = 0$, an artifice accepted to operate on 8 bit numbers or, more generally, a number of bits which are a multiple of 8 (consequently, in our case, P is on 16 bits). Except the operands' loading and the returning of the result, the essential part of the algorithm is represented by repeating the addition-shift (to the right) steps, given by (3.2), a number of times equal to the dimension in bits of the magnitude part, (in our case 7), and the evaluation, finally, of the product's sign.

Translated in terms of the same descriptive language (refer to Appendix A) the multiplication procedure can be associated the code sequence from Fig. 3.5 which in its turn corresponds to the hardware configuration of the multiplication device represented in Fig. 3.6 (adapted from [Haye98]). The essential feature of this device is the fact that the operation is executed by means of a series of CLOCK pulses, which

```
                multiplier 2
                declare register A[7:0], Q[7:0], M[7:0], COUNT[2:0];
                declare bus INBUS[7:0], OUTBUS[7:0];
BEGIN: A:=0, COUNT:=0,                                          {c0}
INPUT: M:=INBUS;
       Q:=INBUS;                                                {c1}
TEST1: if Q[0]=0 then go to RIGHTSHIFT,
  ADD: A[7:0]:=A[6:0]+M[6:0];                                   {c2}
RIGHTSHIFT: A[7]:=0, A[6:0].Q:=A.Q[7:1],                        {c3}
INCREMENT: COUNT:=COUNT+1;
TEST2: if COUNT7≠1 then go to TEST1,
 SIGN: A[7]=Q[0] ex-or M[7], Q[0]:=0;                           {c4}
OUTPUT: OUTBUS:=A;                                              {c5}
        OUTBUS:=Q;                                              {c6}
  END:                                                          {END}
```

Fig. 3.5 Description of sign-magnitude binary multiplication

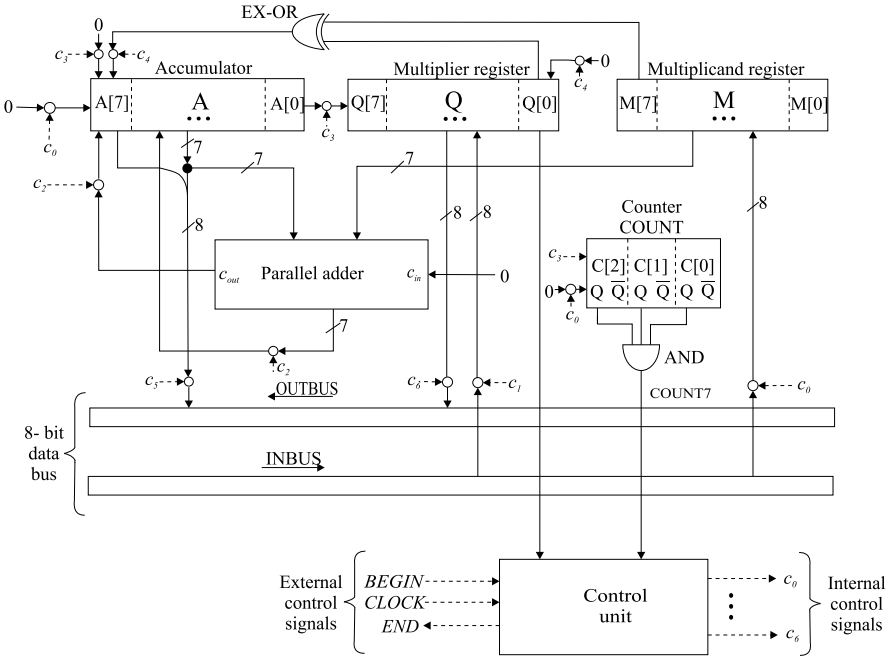


Fig. 3.6 Block diagram of a sequential sign-magnitude binary multiplier

we assume arrive from outside the device (they are included in the set of External control signals), from the Central Processing Unit (CPU). Regarding the structure registers we have, first of all, the two registers where are stored the initial operands, namely, the multiplier *X* in register Q and the multiplicand *Y* in register M. Although of secondary importance, the denotation by letter Q of this register is somehow standard [Haye98]. The name "multiplier" would have justified the denotation of the register by the letter M, but this letter has been preferred for the multiplicand regis-

ter which has the same initial letter M. Since the device from Fig. 3.6, with certain modifications which do not affect the register's configuration, can also be used for the implementation of binary division, a case when the multiplier register is used for the storage of the "quotient", the given register has been denoted by the initial letter from the English term, i.e. Q. We also have a third register denoted by A, from "accumulator". The A register makes up, together with Q, a double length register (in our case 16 bits) provided with the function of left-shifting (refer to the arrow which unites A with Q in Fig. 3.6) and where the partial products described by (3.2) are formed as well as the final one. Through a "flexible" association from the functional point of view, "the accumulator" syntagma has been taken from the adder of some obsolete computers [Haye88], having not only the function of executing the combinational operation of addition, but also that of the cumulative memorizing of the partial and final results.

The device also includes a COUNT register provided with a counting function meant to count the iterations, in order to ensure control of termination of the operation. Taking into account (3.3) and (3.4), the executed number of iterations (3.2) for a complete multiplication is 7 (a case when C[2]C[1]C[0] = 111, and signal COUNT7 = 1 is generated), which requires $\lceil \log_2 7 = 3 \rceil$ ranks for the iteration counter COUNT, where the bars $\lceil \rceil$ represent, again, the smallest integer with value greater or equal to the expression between the bars [Yarb97]. All the specified registers are adequately declared in the sequence from Fig. 3.5, where the bus declaration is also found. In fact the bus is bidirectional, but for didactical purposes, i.e. for highlighting the application of the control signals c_i $(i = 0, \ldots, 6)$, it has been "split" into the input bus INBUS and the output bus OUTBUS, each of them on 8 bits.

Before discussing the algorithm, we mention that the structure also contains, besides the elements specified above, a parallel adder which represents a combinational circuit of one of the types studied in the previous chapter, as well as a control unit. The adder enables the addition of two numbers with dimension corresponding to the magnitude parts of the two operands. Its carry input c_{in} (carry-in) is set to 0 logic (this connection corresponds to the case of the algorithm described above, but the given input may be connected, through reconfiguration, to a control signal c_i which will transform the adder into a subtracter, for another algorithm). The carry output c_{out} is connected to the most significant rank of A (A[7]), because on the addition of two numbers, one represented by the more significant magnitude of a partial product (refer to Fig. 3.4), and the other represented by the magnitude of Y from M, there may result a carry from the msb rank ($c_{out} = 1$).

Regarding the control unit, giving a more general character to this problem, we may associate to a local control unit, such as the control unit for our multiplication device (Fig. 3.6), the block diagram from Fig. 3.7 (adapted after [Haye98] and [PaHe96]). The control unit represents a sequential logic circuit to whose inputs are applied, besides the already mentioned CLOCK pulses, two categories of input signals. The first category comes from the CPU and consists of the signals from the outputs of the operation code decoder (or opcode decoder, for short) of the instructions from the dedicated instruction register. Our multiplier (Fig. 3.6) has only one signal belonging to this category, namely BEGIN (Fig. 3.7), which is activated

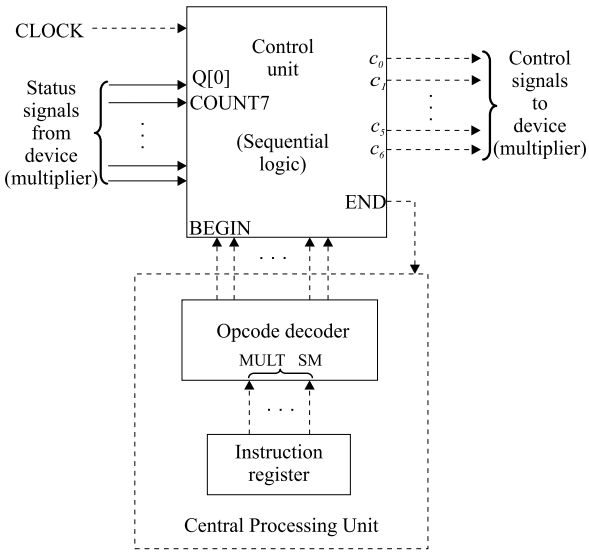


Fig. 3.7 Block diagram of a local control unit of a sequential binary multiplier

when the code of a multiplication instruction of MULT SM (Multiplication Sign-Magnitude) type is supplied to the opcode decoder. The second category of input signals corresponds to the states of the controlled device. The signals important for our device (Fig. 3.6) are, as results from the algorithm description (Fig. 3.5), two state signals, i.e. the signal corresponding to rank Q[0] of register Q, and the above mentioned COUNT7. Emphasizing the signal Q[0], depending on the state 0, respectively 1, of the Q[0] bit, the addition provided by the ADD label (Fig. 3.5) is omitted or executed. This implementation means a certain deviation from the "ad litteram" one of the algorithm based on (3.2) and exemplified in Fig. 3.4, because, as can be observed, when the current bit x_i from Q[0] is 0, the useless, but lengthy, additions to the partial product of 0 are avoided (i.e., $P_i = P_i + 0$) and it passes directly to the right-shift operation provided by the RIGHTSHIFT label (Fig. 3.5).

On the other hand, a control unit supplies as outputs the so-called control signals, which can also be divided into two categories. Some of them, usually denoted by c_i, are applied to the internal structure elements of the device (internal control signals), Fig. 3.6, but also in the more general Fig. 3.7, where there have been marked, using the same signal names, the particular inputs and outputs corresponding to the device from Fig. 3.7. The sequential generation, synchronized through CLOCK, of these signals allows the concatenation of the microoperations provided by the algorithm. The synthesis of the control unit, tending towards an as good as possible solution of the performance/cost impact, aims to attribute the microoperations to the control signals so that the number of the latter may be minimized, while ensuring the avoidance of any logic conflicts. Also to be taken into account are the loadings of the circuits' outputs (fan-out [Yarb97]) which generate these control signals. Regarding the algorithm from Fig. 3.5, mention should be made that to the signal c_0 has been attributed the non-conflicting microoperations for the initialization of the registers A

and COUNT (A := 0, COUNT := 0), and for the loading into register M of the multiplicand from the INBUS (M := INBUS). The next operand, the multiplier, being taken from the same INBUS, is loaded into register Q under the control of a distinct control signal (c_1), whose generation has to wait for the bus release from the multiplicand for the multiplier to be placed on the bus. Figure 3.6 presents the registers and the data lines controlled through the signals c_i. Otherwise, regarding the second category of control signals, which are externally sent from the device (the external control signals), these consist, in our case, of only the *END* signal (Fig. 3.6), which is meant to inform the CPU, in an asynchronous manner, that the operation has finished. However, if the waiting time interval covers the longest execution (involving the "worst case" operands), the signal *END* (and other signals "related" to it) no longer have to be generated.

The representation and functional conventions of Fig. 3.6, which we intend to rigorously apply to the other diagrams, are synthetically summarized below:

(a) The connections used for the data signals' transmission (including the state signals), which are marked with a solid line, are clearly distinguishable from the connections used for the transmission of the control signals, which are marked with a dotted line [Haye98].
(b) Generally, a number is associated with the data connections (refer to Fig. 3.6, value 8 corresponding to the connection between INBUS and register M), which is equal to that of the physical lines. The number may be missing, which means that there is only one physical line (refer to Fig. 3.6, the connection between the EX-OR gate and rank A[7]). Within the same context, certain data connections may branch off, possibly unequally, in which case the point for the specification of the branch is used, and on each branch the number of physical wires will be marked. To a bundle of connections other connections can also be added. In this case the connections' reunion will not be marked using the point, but a "milder" unification, such as the model of the eighth wire's attachment, corresponding to rank A[7], to the bundle of seven output connections of register A to OUTBUS (Fig. 3.6).
(c) On certain data connections circlets are provided and to them a control signal is applied, which signifies the validation of the information loading and/or downloading. For instance this may happen into and/or from a register, but it may take place in only one of its ranks. According to the technology of the available circuitry, there can be imagined various technical implementation solutions such as, for instance, the loading of register M with the multiplicand operand from INBUS by applying, to all the CLOCK inputs of M, the common control signal c_0. Another solution may be to appeal to an AND logic circuits "layer" which has the common signal c_0 and which, when activated, allows the information to penetrate from INBUS towards M. Finally, if we take into account a bidirectional bus of IOBUS type, the implementation can be done by means of three-state circuits, according to the model presented in Fig. 3.8 for bidirectional information traffic (from the lines of the IOBUS bus to the synchronous inputs of register Q ranks, and between the latched outputs of Q

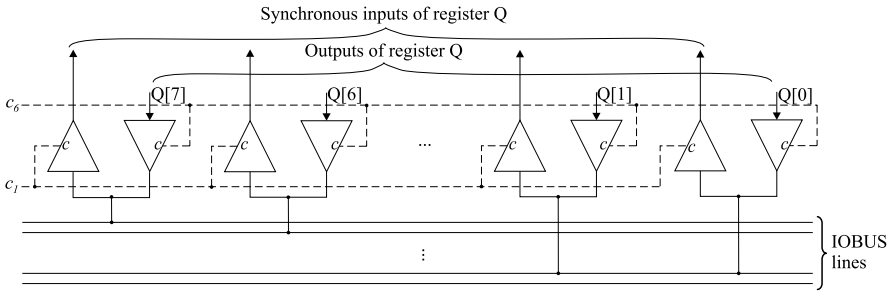


Fig. 3.8 Implementation of a bidirectional bus using tristate circuits

and the lines of the IOBUS bus) controlled through the same control signals c_1 and c_6 (Fig. 3.6).

(d) Regarding the parallel adder and, generally, any combinational circuit, only the circuit outputs are strobed through a common control signal (in our case c_2), the signals' modifications on inputs being of no importance.

(e) At the content's initialization of a register (such as those of registers A and COUNT through c_0, from Fig. 3.6), it is considered that the control signal is simultaneously applied to all the memorizing elements, for instance on the asynchronous reset input. A control signal is also applied to all ranks when this control signal implements a shift or count function such as c_3 in Fig. 3.6.

(f) Unless stated to the contrary, we shall consider that the implementation allows the "usage" of both edges of a control signal, for example, that on the rising edge the reading of the state of a memory element or of a register can be performed, and on the falling edge a new logic value can be written. It is assumed that the given signal has sufficient length so that these operations will not be mutually disturbed. This convention applies in case of the control signal c_4, which through the label SIGN (Fig. 3.5) allows both the reading of Q[0], for the establishing of the result sign in A[7], and the writing of the logic value 0 in Q[0].

These stipulations being made, let us discuss the multiplication algorithm described in Fig. 3.5. Except the declared statements of the structure of the registers and of the bus communication, three parts of the procedure can be distinguished. Thus, at the labels BEGIN and INPUT, under the control of the signals c_0 and c_1, the initialization of those registers (A and COUNT) which may remain with unwanted values from a previous traversal of the algorithm is achieved and the two input operands are loaded into the registers dedicated to them (M and Q). Besides this initial part, we also have a final one given by the labels SIGN and OUTPUT, which under the control of signals from c_4 to c_6, implements many elementary operations. First, the result sign is evaluated through the EXCLUSIVE-OR operation of the operands' signs, that of the multiplicand from M[7] and of the multiplier which has arrived, following the shifts, at the end in Q[0]. Second the result correction is

made through the annulling of Q[0] in case of a negative multiplier (because the product (3.4) on 16 bits is achieved with the harmless $p_0 = 0$). Finally, the result, in two portions of 8 bits each, is returned on the OUTBUS.

Between the two extremes there is the essential part of the algorithm, i.e. its body, which provides the repeating of the iteration (3.2) a number of times equal to that of the magnitude bits. Iteration (3.2) is applied in a form which avoids the addition to the current partial product of the binary equivalent corresponding to digit 0. Thus, as shown above, the value from Q[0] is tested at the line labelled TEST1, proceeding to the addition, provided by ADD, only if Q[0] = 1. This operation, which is executed when the control signal c_2 is activated, has 7 bit operands (A[6:0], M[6:0]), but the result may be on 8 bits (A[7:0]), because, depending on the value of the added numbers, there may result $c_{out} = 1$ (Fig. 3.5, Fig. 3.6). Then, whatever the value of the multiplier's current bit (from Q[0]) is, the right-shift operation (the RIGHTSHIFT label in Fig. 3.5) is executed, controlled by c_3. In bit A[7] the harmless 0 is introduced, which, anyway, does not take part in the subsequent addition, and the bit of the multiplier that has just been tested (at TEST1) will be lost. But the given value, which passes through Q[0] due to its right-shift capacity, is no longer necessary, because it has already been used. In fact this is the key to the implementation solution for the procedure exemplified in Fig. 3.4, because as the multiplier's length progressively decreases bit by bit, the cumulative partial product's length increases, in the same progressive way seen in the example. This artifice enables the use of a parallel adder whose number of ranks is equal to that of the magnitude in bits of the operands, and not equal to the magnitude in bits of the product. Being nonconflicting with the right-shift operation, the incrementing of the iteration counter COUNT (label INCREMENT) is controlled by the same signal c_3. Label TEST2 checks whether the required number of iterations (in our case 7) has been executed, which is true when the decoded contents of COUNT (C[2]C[1]C[0] = 111) generates the signal COUNT7 = 1. At this moment the multiplier's sign has arrived in Q[0], so that the result's sign can be evaluated (through SIGN). As long as COUNT7 = 0, the procedure is looped between TEST1 and TEST2, which delimits the central part (the body) of the algorithm.

Obviously, it might be interesting to study the synthesis of the control unit which allows the generation of the control signal sequence. In Appendix B several solutions are presented regarding this synthesis, starting from a different algorithm but similar to the above-mentioned one, namely Robertson's algorithm meant for the multiplication of binary numbers represented in two's complement. Robertson's algorithm has been chosen because it enables the highlighting of more characteristic elements of the various synthesis methods. Mention should be made that any of these methods can also be applied to the control unit from Fig. 3.6.

We shall now present an example (Fig. 3.9) that has been chosen in such a way that it permits the illustration, by comparison, of the specific aspects of the various algorithms. As regards the operands, we take the decimal values $X = -89 \cdot 2^{-7}$ and $Y = -105 \cdot 2^{-7}$, which, translated into binary using the sign-magnitude code, lead

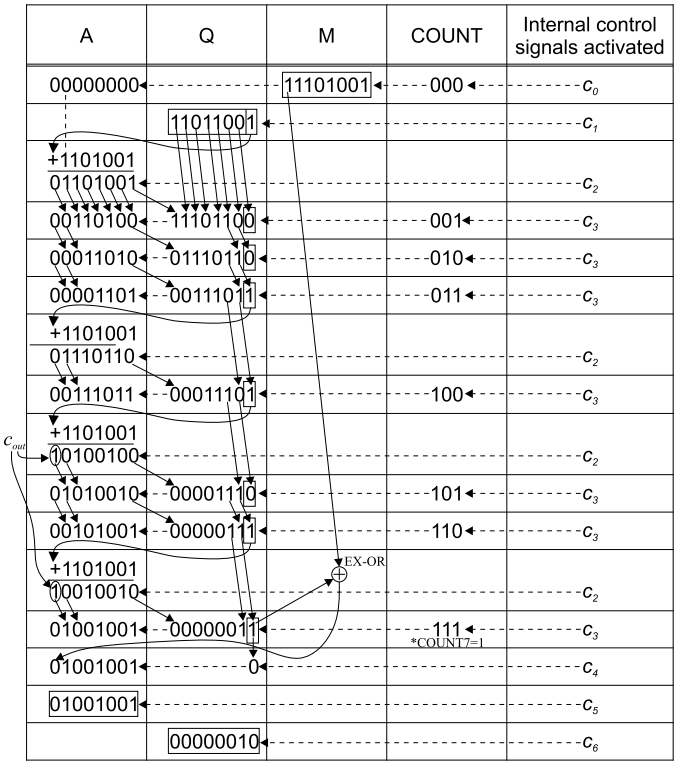


Fig. 3.9 Example of a binary multiplication in sign-magnitude code with microoperations' control signals activation

to the following representations $X_{SM} = 11011001 = -(2^{-1} + 2^{-3} + 2^{-4} + 2^{-7}) = -89 \cdot 2^{-7}$, and $Y_{SM} = 11101001$. Figure 3.9 presents in a table the contents of the structure registers, as well as the control signals which have to be activated to release a certain elementary operation. The values captured from the bus for the input operands, as well as the two halves of the product downloaded in the bus, are presented in frames. The values from rank Q[0], the generation moments of c_{out} and COUNT7, the shift operation are highlighted. To obtain the product $P_{SM} = 010010010000001 = +9345 \cdot 2^{-14}$ requires the activation of the parallel adder circuits four times, i.e. once for each binary unit from the magnitude part of X, there being necessary, as expected, seven right-shift operations.

Analyzing the procedure and if taking into account the worst case of a multiplier X with the magnitude containing only binary units and if the preliminary and final operations of operand loading and returning the result are ignored, the algorithm presented above may be characterized, in terms of CLOCK pulses, to have the complexity $O(n)$, this being a substantial improvement, as compared to the above-presented algorithm based on counters.

3.3 Sequential Two's Complement Binary Multiplier Based on Robertson's Procedure

Let us now use the structural elements of the multiplication device from Fig. 3.6 to compute, by an easily achieved reconfiguration, the product of the binary numbers given in both the fixed point representations of interest, the sign-magnitude (SM) and the two's complement (C2). Mention should be made that, unlike the case presented in the previous section, this time only numbers in C2 code are carried on the buses, both for the input operands and for the product result.

We shall analyse the entire use of the multiplier from Fig. 3.6 transforming the negative input operands from C2 into SM, and, if the final product is negative, it will be transformed in reverse order, from SM into C2. The worst case which has to be covered regarding this aspect involves the multiplication of contrary signs operands, when the resulting product is negative. This case requires three complementing operations (one for the negative operand and two for the product, the latter being returned in two separate parts). When both operands are negative, only two complementings are necessary, because the product is obviously positive and does not require complementing. But, as it is already known, a two's complementing requires at least the time for an addition, besides the operations for the signs' testing. Consequently, this solution leads to performance degradation caused by the additional complementing and recomplementing operations, which is why we shall use an approach which will not imply code transformations.

Thus, we shall start from the second form given in the previous chapter for the C2 representation, attributed to James Robertson [ErLa04, Stal99, Haye98], according to which, for a binary number $X = x_{n-1}x_{n-2}\ldots x_i\ldots x_1x_0$, code C2 is given by the following relations:

$$X_{C2} = -x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i' 2^i, \quad \text{for } X \text{ integer} \tag{3.5}$$

$$X_{C2} = -x_{n-1}2^{0} + \sum_{i=1}^{n-1} x_{n-i-1}' 2^{-i}, \quad \text{for } X \text{ fractional} \tag{3.6}$$

where the bits x_i' and x_{n-i-1}' coincide with x_i and x_{n-i-1}, in case of positive numbers ($x_{n-1} = 0$) and correspond to the bits of the C2 code of X in case of negative numbers ($x_{n-1} = 1$).

Taking into account relations (3.5) and (3.6), we shall adopt the binary configuration used in Fig. 3.9 for multiplier X, i.e. $X_{SM} = x_7x_6\ldots x_i\ldots x_1x_0 = 11011001$. If the well-known procedure for converting into C2 code is applied to this configuration, to it will correspond $X_{C2} = x_7x_6'\ldots x_i'\ldots x_1'x_0' = 10100111$. On the other hand, if X_{C2} is interpreted as an integer, on the basis of relation (3.5), we have

$$X_{C2} = -2^7 + \sum_{i=0}^{6} x_i' 2^i = -2^7 + \left(2^5 + 2^2 + 2^1 + 2^0\right) = -89.$$

However, if X_{C2} is interpreted as fractional, on the basis of relation (3.6), we have

$$X_{C2} = -2^0 + \sum_{i=1}^{7} x'_{7-i} 2^{-i} = -1 + \left(2^{-2} + 2^{-5} + 2^{-6} + 2^{-7}\right) = -89 \cdot 2^{-7}.$$

Regarding the use of the structure from Fig. 3.6 with as few modifications as possible for the multiplication of binary numbers in C2, we highlight the fact that on the multiplication of binary numbers in SM we have to parse multiplier X, bit by bit, from right to left, starting with the iterations (3.2) with the least significant bit, and advancing towards the sign bit. In fact, each bit of X is brought, by right-shift, into Q[0], where following the analysis of the bit value, it is eliminated. Depending on the bit value in Q[0], there is executed either a shift (if Q[0] = 0) or both an addition and a shift (if Q[0] = 1). It has to be pointed out that by representing X in one of the forms (3.5) or (3.6), the desideratum of enabling the implementation of the procedure based on steps is achieved. These steps are based either on shifting or on both addition and shifting, while X running through the multiplier, from right to left, when X is represented in C2. According to the sign of X two cases can be distinguished:

(α) When X is positive ($x_{n-1} = 0$ and $x'_i = x_i$, with $i = 0, \ldots, n-2$ for integers, and with $i = 1, \ldots, n-1$ for fractional numbers), the treatment of X is absolutely similar to that given by the procedure from Fig. 3.5.

(β) When X is negative ($x_{n-1} = 1$ and the bits x'_i correspond to the representation of X in C2, with $i = 0, \ldots, n-2$ for integers, and with $i = 1, \ldots, n-1$ for fractional numbers) it will run through the bits x'_i, the same way as in case α, by applying the iteration (3.2) and avoiding the addition of 0, until the sign bit is reached. Thus, there will be obtained the cumulative partial products P:

$$P' = \sum_{i=0}^{n-2} x'_i Y 2^i, \quad \text{for } X \text{ integer} \tag{3.7}$$

$$P' = \sum_{i=1}^{n-1} x'_{n-i-1} Y 2^{-i}, \quad \text{for } X \text{ fractional} \tag{3.8}$$

The values P' thus obtained shall undergo an additional subtraction operation (in case of negative numbers, when $x_{n-1} = 1$) corresponding to the sign bit, according to (3.5) and (3.6), which makes the correction for the fact that the bits x'_i do not belong to a direct, sign-magnitude, representation, but to the representation in C2. If (3.7) and (3.8) are taken into account, the final products P result through the final correction step which consists of the subtraction, properly aligned, of Y from the values of P':

$$P = P' - Y 2^{n-1} = \left(\left(\sum_{i=0}^{n-2} x'_i 2^i\right) - x_{n-1} 2^{n-1}\right) Y = XY, \quad \text{for } X \text{ integer} \tag{3.9}$$

$$P = P' - Y = \left(\left(\sum_{i=1}^{n-1} x'_{n-i-1} 2^{-i} \right) - x_{n-1} 2^0 \right) Y = XY, \quad \text{for } X \text{ fractional} \tag{3.10}$$

Through the correction step based on subtraction, an additional element has appeared in comparison with the algorithm from Fig. 3.5, which had only additions. But the subtraction can be performed by adding the two's complement and executing the artifice of transforming the parallel adder into a subtracter by adding, on inputs, an EX-OR circuits layer, and through the addition of a binary unit using the input c_{in}. Thus, the parallel adder from Fig. 3.6 can be used with the above-mentioned circuitry supplementations. Appealing to this technical solution, the sign bit does not require special treatment (as in Fig. 3.5), but it undergoes the same operations as any other, ordinary, bit of the representation, which is the reason for the extension by one rank of the parallel adder and of the connections corresponding to it (Fig. 3.6).

There is one more problem connected with the representation in C2, this time with the representation of the multiplicand $Y = y_{n-1} y_{n-2} \ldots y_j \ldots y_1 y_0$. When Y is positive ($y_{n-1} = 0$) no special aspects appear, because we are dealing with the multiplication of two positive numbers (recall that in case of a negative X, as well, the part of X corresponding to the bits x'_i is considered a positive number, as long as the sign bit is not included). Consequently, the partial products are positive numbers. This is important when the given products are right-shifted, because a harmless 0 as regards the shifted value is introduced in the most significant rank. But, if Y is negative ($y_{n-1} = 1$), the problem changes. Namely, when on running through X from right to left, the first 1 bit encountered, since X is "still" positive, results, because of (3.2), in a negative partial product. This determines, until the sign of X is interpreted, the perpetuation of negative values for the cumulative partial product (given by (3.7) and (3.8) respectively). However, if a negative number is right-shifted at the most significant bit, this time, a harmless 1 has to be introduced. It can be said that the partial products undergo an arithmetic shift [Haye98, Yarb97]. For the less significant bits of X, which have the value 0, whatever the value of Y is, null partial products result, which, through the right-shift, have to remain null. The consideration of the above mentioned cases in the synthesis of the multiplication device leads to the addition of a flag F, which will supply the binary value introduced in the most significant rank of the partial product. Consequently, depending on the operands' signs, the multiplication algorithm of the binary numbers represented in C2 has to activate, as applicable, the additional correction and the arithmetic shift functions, presented in Fig. 3.10. Following this description, we shall present in Fig. 3.11, using the same descriptive language as before, the code sequence corresponding to the procedure for the multiplication of binary numbers given in C2, according to the method elaborated by James Robertson [Haye98]. Then synthesis of the multiplication device named Robertson, whose characteristics are similar to that from Fig. 3.6, is presented in Fig. 3.12.

Signs		Correction step	Arithmetic shift
x_{n-1}	y_{n-1}		
0	0	no	no
0	1	no	yes
1	0	yes	no
1	1	yes	yes

Fig. 3.10 Correction and arithmetic shift steps depending on the operands' signs for the binary multiplication by the Robertson's procedure

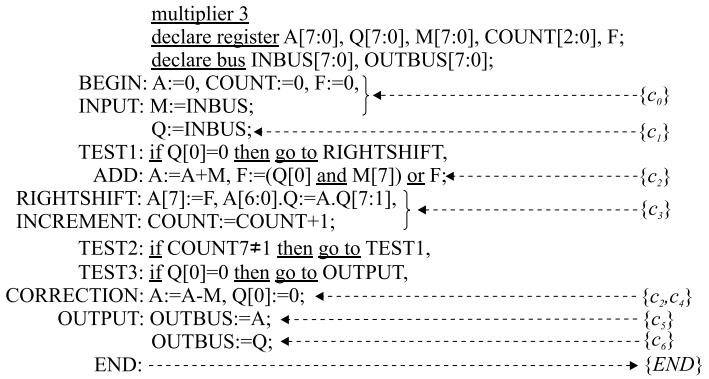


Fig. 3.11 Description of the two's complement binary multiplication based on Robertson's procedure

The new procedure (Fig. 3.11) is presented by comparison with that from Fig. 3.5, and, as regards the new device (Fig. 3.12), by taking into account the minute presentation already detailed, we will insist only on modifications that differ from the diagram given in Fig. 3.6. First of all, we shall mention the declaration of flag F, which is initialized by c_0 together with A and COUNT. F is set by c_2 at the same time as the addition (A+M), these two operations not being conflictive. In agreement with what has been presented above, when Y is negative (M[7] = 1) and the first binary unit occurs in Q[0], we have (Q[0] = 1 and M[7] = 1) and F is set to 1, being maintained in this state (with or F) until the sign bit is reached. Thus, the flag F ensures the implementation of the arithmetic shift (through A[7] := F). On the other hand, it can be observed that the parallel adder has eight ranks, the sign bit being treated in the same way as the other bits. Finally, as regards again the parallel adder, we mention the presence of EX-OR gates on the entire word length (wordgate), which allow the forming of the two's complement for Y when signal c_4 is activated; signal c_4 being also applied to the input c_{in} of the adder. Consequently, to the one's complemented value of Y a 1 is added and the two's complement for Y is formed. By its addition to the more significant bits (from A) of the cumulative partial product, when X is negative (Q[0] = 1), the correction step is performed (labeled CORRECTION, on which, nonconflictually, on the falling edge of c_4, the

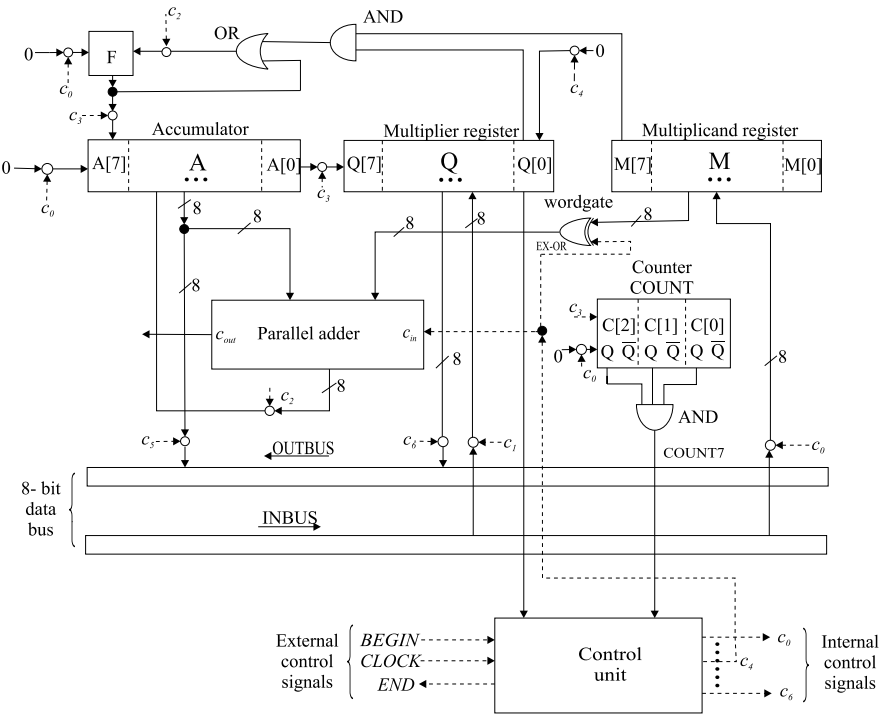


Fig. 3.12 Block diagram of a sequential two's complement binary multiplier based on Robertson's procedure

removal of the negative sign from Q[0] is also performed). As can easily be observed, for the correction subtraction, it is necessary to generate two control signals, c_2, ensuring the addition microoperation, and, c_4 ensuring the complementing one.

Figure 3.13 presents the actions of the circuit from Fig. 3.12 on the example from Fig. 3.9. The same fractional values for the two operands, $X = -89 \cdot 2^{-7}$ and $Y = -105 \cdot 2^{-7}$ are taken into account, but this time their representations are captured from the bus in C2, i.e. $X_{C2} = 10100111$, and $Y_{C2} = 10010111$. Product P is the same as that from Fig. 3.9, but this happens because, under these circumstances, P is positive. If P had been negative, because it had been obtained in C2, then its value would have been different from that obtained by the device from Fig. 3.6.

As mentioned above, Appendix B presents the synthesized versions of the control unit from a Robertson device, versions which generate the signals indicated in Fig. 3.13. Finally, mention should be made that the procedure presented in Fig. 3.11 does not differ, in terms of the above-indicated complexity, from the multiplication procedure for binary numbers in sign-magnitude, but it is more efficient than the procedure presented at the beginning of this section.

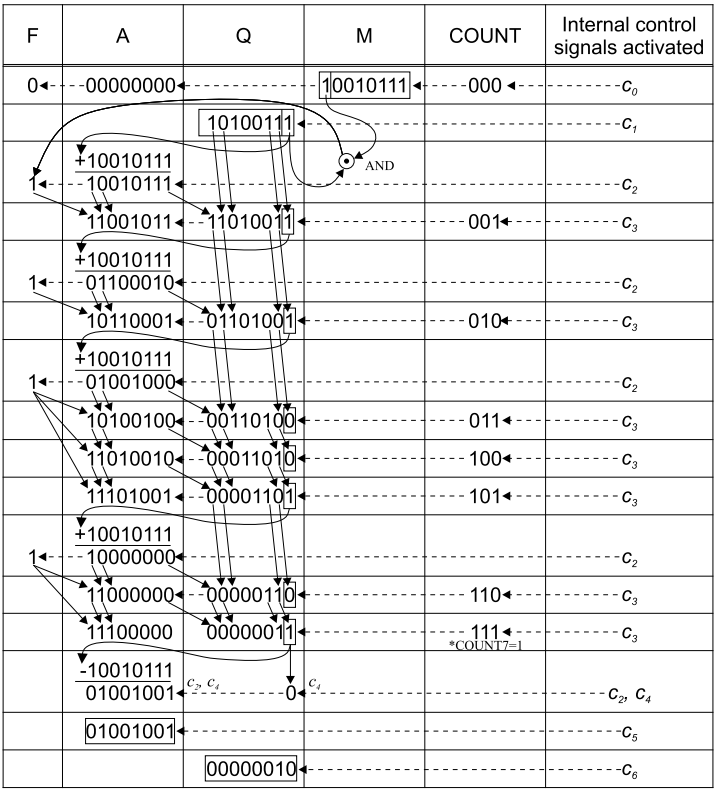


Fig. 3.13 Example of a binary multiplication in two's complement code by the Robertson's procedure with microoperations' control signals activation

3.4 Sequential Two's Complement Binary Multiplier Based on Booth's Procedures

It can easily be observed that the increase of the number of binary units in the structure of the multiplier X, whether it is represented in SM or in C2, results in the increase of the number of c_2 signals that have to be activated (Fig. 3.9 and Fig. 3.13), the time of the multiplication process being implicitly increased. Aiming to reduce the effect of the above-mentioned disadvantage, Andrew Booth has proposed, for binary numbers in C2, a method [ErLa04, BoTi05, Parh00, HePa03] which is not based on the inspection, at a certain time, of just one bit of the multiplier (that one in the Q[0] rank, Fig. 3.12), but on inspection of the values of two bits with adjacent positions. First of all, let us denote the two bits by $x_i x_{i-1}$ and let us point out that the simultaneous inspection is meant to detect a transition (0 to 1 and 1 to 0 respectively) at the interface of the two bits. The transition's presence or absence shall be corroborated with an elementary mathematical speculation whose effect will be followed up by supposing, without loss of generality, that multiplier X is a

positive integer and that X^* is a binary sequence, included in X and consisting of an uninterrupted sequence of binary units with a 0 at both ends, i.e.:

$$X^* = x_{i+k+1}x_{i+k}x_{i+k-1}\ldots x_{i+1}x_i x_{i-1} = 011\ldots110 \tag{3.11}$$

Based on Robertson's procedure the contribution of the X^* sequence to the product $P = XY$ is given, in the case specified by (3.11), by the part of the product P which we denote by P^*, and whose value is given below, if we take into account the weights associated with the bits in a similar way to that presented in Fig. 3.3:

$$\begin{aligned} P^* &= 0\cdot Y2^{i+k+1} + 1\cdot Y2^{i+k} + 1\cdot Y2^{i+k-1} + \cdots + 1\cdot Y2^{i+1} \\ &\quad + 1\cdot Y2^{i} + 0\cdot Y2^{i-1} \\ &= Y\left(2^{i+k} + 2^{i+k-1} + \cdots + 2^{i+1} + 2^i\right) \\ &= Y\left(2^k + 2^{k-1} + \cdots + 2^1 + 1\right)2^i \\ &= Y\left(2^{k+1} - 1\right)2^i = 1\cdot Y2^{i+k+1} - 1\cdot Y2^i \end{aligned} \tag{3.12}$$

where the well-known identity $2^a - 1 = (2-1)(2^{a-1} + \cdots + 1)$ has been used, with a a positive integer.

Interpreting (3.12), we have $(k+1)$ additions required by the computation of the contribution to the product P of part X^*, which can be substituted by just two operations, an addition and a subtraction. When the binary sequence X^* is run through from left to right and a transition from 0 to 1 is met, i.e. we have the binary pair $x_i x_{i-1} = 01$, an addition is executed. In a similar way, when a transition from 1 to 0 is encountered, i.e. we have the binary pair $x_i x_{i-1} = 10$, a subtraction is executed. When there are no transitions and the bits form sequences with identical binary values (through (3.12) the case of a binary unit sequence has been presented, but it can easily be shown that a binary zero sequence behaves in a similar way), i.e. when the binary pair is either $x_i x_{i-1} = 00$ or $x_i x_{i-1} = 11$, neither addition nor subtraction is executed.

With the aim of pointing out other characteristics, again without loss of generality, let us suppose that we have a fractional negative number of the following form:

$$X_{C2} = 1x'_{n-2}x'_{n-3}\ldots x'_i x'_{i-1}\ldots x'_1 x'_0 = 100\ldots00\ldots001 \tag{3.13}$$

In this case, if (3.6) is taken into account, for the final product P can be written:

$$P = X_{C2}Y = (-1)Y2^0 + 1\cdot Y2^{-n+1} \tag{3.14}$$

On one hand relation (3.14) confirms the previous suppositions, and, on the other hand, it shows that the evaluation of P can be correctly made, namely that:

$$\begin{aligned} P &= (-Y)2^{-n+1}\left(2^{n-1} - 1\right) \\ &= (-Y)2^{-n+1}(2-1)\left(2^{n-2} + 2^{n-3} + \cdots + 2 + 1\right) \\ &= (-Y)\left(2^{-1} + 2^{-2} + \cdots + 2^{-n+1}\right) = (-X_{SM})Y \end{aligned} \tag{3.15}$$

Fig. 3.14 Example of multiplier's Booth recoding

Range / Number	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{-1}
	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
$X_{SM}=$	1	1	0	1	1	0	0	1	
$X_{C2}=$	1	0	1	0	0	1	1	1	0
$X_B=$	$\overline{1}$	1	$\overline{1}$	0	1	0	0	$\overline{1}$	

As regards (3.13), one more stipulation concerning the lsb inspection has to be made. We began by exploring the values of pairs of bits from left to right, but when the lsb is reached, it remains alone. This situation implies the extension of multiplier X with one zero bit to the right of the lsb. Thus the value is not affected and, in fact, for X_{C2} from (3.13), we have:

$$X_{C2} = 1x'_{n-2}\ldots x'_i x'_{i-1}\ldots x'_1 x'_0 x_{-1} = 10\ldots00\ldots010 \tag{3.16}$$

where $x_{-1} = 0$ is the added bit.

Synthesizing the above presentation and taking into account the extension of X with $x_{-1} = 0$ to the right, we can introduce Booth's recoding, as the literature calls it [Parh00, Haye98, KaGa06, HePa03]. This consists of recoding the multiplier in the so-called signed digit form, characterized by the fact that the multiplier digits are provided with sign. Consequently, we appeal to the convention according to which the multiplier X is scanned from left to right (a fact also suggested by the notation of the pair of bits $x_i x_{i-1}$, starting therefore with $i = n-1$ and ending with $i = 0$, x_i being the current bit), so that, when the pair $x_i x_{i-1} = 01$ is encountered, for the current bit of the recoded form we use 1 (in fact $(+1)$), and when the pair $x_i x_{i-1} = 10$ is encountered, for the current bit of the recoded form we use $\overline{1}$ (in fact, (-1)). In the other two cases, when $x_i x_{i-1} = 00$ and $x_i x_{i-1} = 11$, for the current bit of the recoded form we use 0. Thus, if the multiplier of our example operation is considered (refer to Fig. 3.9 and Fig. 3.13), its Booth recoded form, denoted by X_B, is obtained as shown in Fig. 3.14. The following computation shows that the value of X_B is correct: $X_B = -2^0 + 2^{-1} - 2^2 + 2^{-4} - 2^{-7} = (-2^7 + 2^6 - 2^5 + 2^3 - 1)2^{-7} = -89 \cdot 2^{-7}$.

Recording of the multiplier is at the basis of the Booth multiplication procedure. Namely, it indicates the microoperations which the multiplication device has to execute at a certain moment (a 1 bit in X_B (Fig. 3.14) means that an addition followed by a shift has to be executed, $\overline{1}$ bit means that a subtraction followed by a shift has to be executed, and a 0 bit shows that only a shift has to be executed). Mention should be made that, unlike Robertson's procedure where a single subtraction was made, and this only for a negative multiplier X, Booth's procedure, as described, will appeal more frequently to subtraction. But this is not an inconvenience due to the simple implementation of this operation through the addition of numbers in two's complement (Fig. 3.12). However, one aspect regarding the subtractions must be mentioned namely that during the procedure there may occur, alternately, depending on the operands' values, both positive and negative partial products, a fact which is important in connection with the right-shift operation. This operation requires the introduction of a binary value in the msb without the modification of the

```
          multiplier 4
          declare register A[7:0], Q[7:-1], M[7:0], COUNT[2:0];
          declare bus INBUS[7:0], OUTBUS[7:0];
BEGIN: A:=0, COUNT:=0,                                          {c0}
INPUT: M:=INBUS;
          Q[7:0]:=INBUS[7:0], Q[-1]:=0;                         {c1}
TEST1: if Q[0]Q[-1]=01 then A:=A+M, go to TEST2;                {c2}
          else if Q[0]Q[-1]=10 then A:=A-M;                     {c2, c3}
TEST2: if COUNT7=1 then go to OUTPUT,
RIGHTSHIFT: A[7]:=A[7], A[6:0].Q:=A.Q[7:0],                     {c4}
INCREMENT: COUNT:=COUNT+1, go to TEST1;
OUTPUT: OUTBUS:=A, Q[0]:=0;                                     {c5}
          OUTBUS[7:0]:=Q[7:0];                                  {c6}
END:                                                            {END}
```

Fig. 3.15 Description of the two's complement binary multiplication based on Booth's procedure

number (0 for a partial positive product, and 1 for a partial negative one). This requirement can be fulfilled through a simple technical solution, i.e. the recirculation of the binary value of the msb.

Figure 3.15 presents the code sequence corresponding to the Booth procedure (adapted from [Haye98]) for comparison with the procedures mentioned above, in terms of the usual hardware description language. Connected with this presentation, Fig. 3.16 contains the hardware device in whose synthesis we have made as few modifications as possible as compared to the other diagrams (Fig. 3.6, Fig. 3.12), so that the reconfiguration may be executed as simply as possible. Regarding the algorithm, we mention the declaration of register Q, to which, as compared to the homologous register from the other versions, the rank Q[–1] is added, to make a pair with Q[0], thus assuring the simultaneous inspection of two bits. The right-shift function of register Q also extends over this 8th rank (refer to label RIGHTSHIFT). The state of Q[–1] has to be initialized, as established, by deleting its content, an action attributed to signal c_1, which controls the loading of the other ranks of register Q with multiplier X from INBUS. The ranks of Q have to be specified at the input, but they shall also be specified at the output, to show which of the Q ranks contain the less significant part of the product which has to be returned, under the control of c_6, to OUTBUS.

The essential and distinctive characteristic is the simultaneous testing of the values of a pair of bits starting with the two least significant bits and going on, bit by bit, towards the sign bit, as the multiplier is right-shifted. Mention should also be made of the recirculation of the msb rank contents of register A (A[7] := A[7]) and that, in the code writing (Fig. 3.15), the treatment of the sign bit has been attributed to the central part of the procedure (i.e. to the body of the procedure), not to its final part (through the labels SIGN, in Fig. 3.5, and CORRECTION, in Fig. 3.11). The right-shift with the conservation of the sign of the number ensures, in fact, the arithmetic shift.

Figure 3.17 presents, for the new procedure, the same multiplication example given in Fig. 3.9, and in Fig. 3.13 respectively. First, we mention that the sequence of operations, including that of control signals, corresponds exactly to the binary structure of the recoded multiplier (Fig. 3.14). Secondly, we remark that the complexity

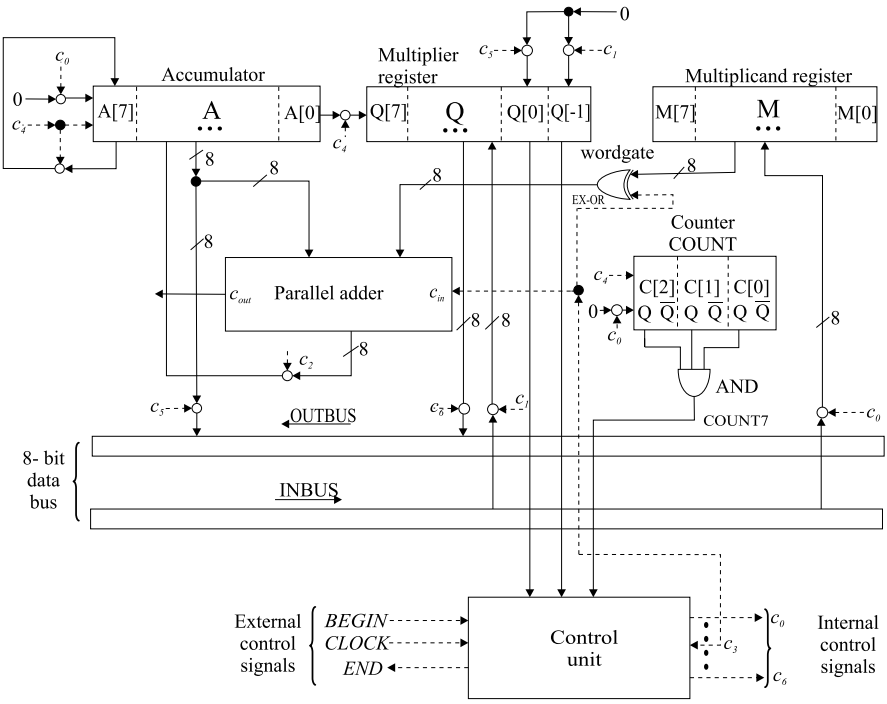


Fig. 3.16 Block diagram of a sequential two's complement binary multiplier based on Booth's procedure

of the method depends on the number of the generated control signals, which give a measure of the throughput, and implicitly the procedure performance. Regarding the particular example adopted, multiplier X contains, both in SM code and in C2 code, the same number of binary units (5) (and, implicitly, the same number of zeros (3)). Consequently, the procedures from Fig. 3.5 and Fig. 3.11 do not, normally, show differences, as far as performance is concerned. We should have expected an acceleration of the multiplication process by Booth's algorithm, but, as evaluated by the number of distinct control signals, this does not happen. The reason for this fact can be found if we analyse the particular form of X_B from Fig. 3.14, involving the same number of activations of the adder (five) that has been needed for the other methods. It is of no importance that some activations of the adder (three), of the total of five activations, use it as a subtracter. This is because the complementing signals (c_3) and the addition signals (c_2) can be derived from the same CLOCK pulse, even if they are slightly shifted—c_3 precedes c_2—so that the strobing (through c_2) of the result at the combinational diagram output, which is the adder, may be done correctly. Consequently, Booth's procedure does not bring a performance improvement for the example from Fig. 3.17 due to the particular binary configuration of multiplier X. But, when X_{C2} has significant sequences of binary units or of binary zeros,

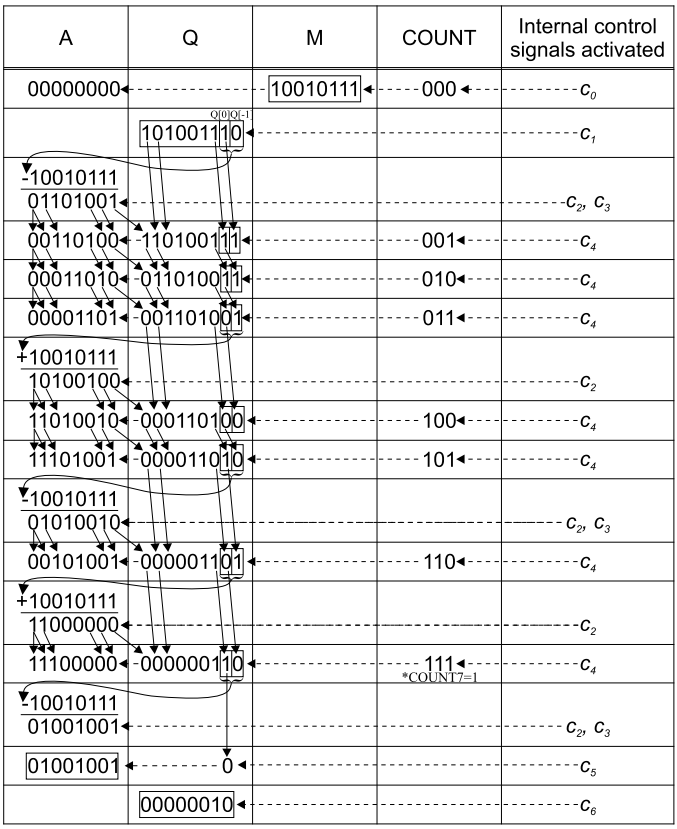


Fig. 3.17 Example of a binary multiplication in two's complement code by the Booth's procedure with microoperations' control signals activation

then, in the corresponding recoded form X_B, the number of zeros increases and this leads to the improvement of the throughput capacity of the procedure.

However, it is possible for a certain multiplier X_{C2} to have frequent alternations of the bits (of type ...0101...), when there may occur, not the expected improvement, but, on the contrary, a degradation of the performance, which may even be dramatic. Thus, if within X we have a 1 with a 0 on both sides (010), then in terms of Robertson's procedure only one addition operation (corresponding to the 1) is required, while, in terms of Booth's procedure, an addition (corresponding to the pair 01) is required, followed immediately by a subtraction (corresponding to the pair 10). However, if the triplet $X^*_{C2} = 010$ is more minutely analyzed, we have the situation from Fig. 3.18, where by X^*_B we have denoted Booth's recoding, according to Fig. 3.14. It can easily be observed that the two operations corresponding to X^*_B can be substituted by an addition. Namely, when at X_{C2} scanning the triplet X^*_{C2} is encountered, only one addition operation has to be made, as presented for X^*_{MB}. Similarly, when X_{C2} is scanned and the triplet $X^{**}_{C2} = 101$ is encountered, only one

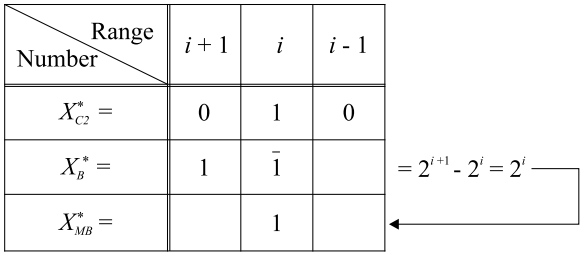


Fig. 3.18 Justifying the Booth's procedure modification for multiplier pattern consisting of a 1 flanked by 0s

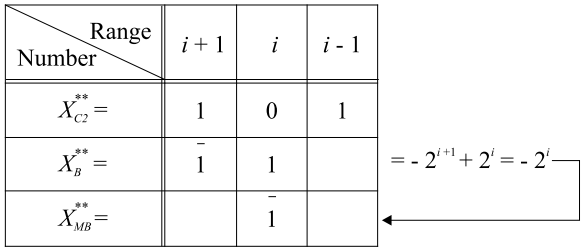


Fig. 3.19 Justifying the Booth's procedure modification for multiplier pattern consisting of a 0 flanked by 1s

subtraction has to be made, as presented for X^{**}_{MB} in Fig. 3.19. The two new forms X^*_{MB} and X^{**}_{MB} correspond to a new recoding of the multiplier, the so-called canonical multiplier recoding, to which corresponds the so-called canonical signed digit form [Haye88]. This new recoding stands at the basis of a new procedure attributed to Booth, which, to distinguish it from the previous one, has been called the modified Booth's algorithm [Parh00, Haye98, BoTi05] (wherefrom the indexes *MB* of the previous forms from Fig. 3.18 and Fig. 3.19). The canonical recoding maintains the rules used in the previous form X_B (the correspondences for the binary pairs from $X_{C2}: 01 \rightarrow 1$, $10 \rightarrow \bar{1}$ and $00 \rightarrow 0$, and $11 \rightarrow 0$ respectively are maintained), except for their application to the isolated values of 1 and 0, when the coding is performed according to Fig. 3.18 and Fig. 3.19. As mentioned above, two bits are simultaneously inspected, but the right-shift is made, each time, by only one binary position. Consequently, to detect whether a bit is isolated or not, the "history" of the scanned binary values has to be stored. Thus, a flag which distinguishes between the isolated bits and a run of 1s or of 0s made up of two or several identical bits is used. The flag is denoted by R (from "run", to distinguish it from F used in Robertson's procedure) specifying the following conventions for its setting up:

1. Initially, R is set to 0.
2. This time, multiplier X_{C2} is scanned from right to left, two bits being simultaneously inspected as before, but in the opposite direction as compared to the previous Booth procedure. This implies the extension of the multiplier by one bit to the left of its msb, doubling the sign bit, which does not affect the value of multiplier X_{C2}.
3. If in the previously mentioned scanning of multiplier X_{C2} the pair of bits $x_{i+1}x_i = 01$ is met, the state of R does not have to be changed. But, if the pair

Fig. 3.20 Synthesis of the rules for obtaining the multiplier's canonical recoding

Inputs			Outputs	
x_{i+1}	x_i	R	x_{iMB}	R*
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	0	0
1	0	1	$\overline{1}$	1
1	1	0	$\overline{1}$	1
1	1	1	0	1

$x_{i+1}x_i = 11$ is encountered, then the state of R has to be changed, the flag being set to 1.

4. If the multiplier X_{C2} scanning continues and the pair of bits $x_{i+1}x_i = 10$ is met, the state of R shall not be changed. But, if the pair $x_{i+1}x_i = 00$ is encountered, the state of R has to be changed, the flag being reset to 0.
5. Conventions 3 and 4 are applied alternately and repeatedly until the entire binary sequence corresponding to X_{C2} is scanned.

If we synthesize all the rules referring to the obtaining of the multiplier's canonical recoding, the table from Fig. 3.20 results, where the inputs are represented by the inspected pair of bits $x_{i+1}x_i$, made of the current bit x_i and the next bit to the left x_{i+1}, as well as the current value of the flag R. The outputs are represented by the current value of the bit x_{iMB} (with sign) of the recoded form and by the new state of the flag R, denoted by R*. In this table, the triplets (0, 1, 0) and (1, 0, 1) can be observed to correspond to binary values with some opposed values in their bits, and that only one operation is executed, of addition ($x_{iMB} = 1$) in case of an isolated 1, and of subtraction ($x_{iMB} = \overline{1}$) in case of an isolated 0. Then triplets (0, 0, 1) and (1, 1, 0) mark the beginning of sequences of 0s and 1s when toggling of flag R takes place. As regards triplets (0, 1, 1) and (1, 0, 0), there is uncertainty whether the transition, which exists in both cases, corresponds to an isolated bit or to a sequence of bits. Consequently, no operation will be executed ($x_{iMB} = 0$) and the flag's state will be maintained. This will also happen in case of the extreme triplets (0, 0, 0) and (1, 1, 1).

Let us apply the rules contained in Fig. 3.20 to the multiplier corresponding to the example from Fig. 3.9, Fig. 3.13 and Fig. 3.17 and thus the states of flag R and the canonical recoding X_{MB} from Fig. 3.21 will be obtained. The multiplier X_{C2} is scanned from right to left, the first triplet (1, 1, 0) being made up of $x_1 = 1, x_0 = 1$ and R = 0 (initial value), and it determines, according to Fig. 3.20, $x_{iMB} = \overline{1}$ and R = 1, and then the following triplet is $x_2 = 1, x_1 = 1$ and R = 1, etc. Interpreting

Range / Number	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
		2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}
$X_{SM}=$		1	1	0	1	1	0	0	1
$X_{C2}=$	1	1	0	1	0	0	1	1	1
R=		1	0	0	0	0	1	1	1	0
$X_{MB}=$		$\overline{1}$	0	1	0	1	0	0	$\overline{1}$

Fig. 3.21 Example of multiplier's canonical Booth recoding

the configuration obtained for X_{MB} we have: $X_{MB} = -2^0 + 2^{-2} + 2^{-4} - 2^{-7} = -(2^7 - 2^5 - 2^3 + 1)2^{-7} = -89 \cdot 2^{-7}$.

The table given in Fig. 3.20 can also be used for the easy deduction of the Boolean equation corresponding to the setting up of the flag R, which we denote by S_{R^*} and which, in terms of the input variables, is the following:

$$S_{R^*} = x_{i+1}x_i + x_{i+1}R + x_iR \tag{3.17}$$

Regarding the new recoding of X, mention should be made that isolated values may lead to the repetition of a certain binary unit with sign (1 and $\overline{1}$) until the binary unit of opposite sign is encountered (refer also to the example from Fig. 3.21, where between the two extreme $\overline{1}$ there appear two 1s). This differs radically from the previous Booth recoding which is characterized by the fact that binary units with sign alternate (they may or may not be separated by 0 values, refer also to the example from Fig. 3.14). As far as the operations are concerned, the above-mentioned alternation of additions and subtractions excludes the possibility of overflow, i.e. the register's capacity being exceeded. However, in the new Booth recoding, as well as in Robertson's method, overflow may occur, due to the feature mentioned above. In both procedures, the overflow phenomenon is important from the point of view of the value which is introduced, through right-shift, in the most significant rank of register A (in this case, A[7]). Regarding Robertson's algorithm, the occurrence of overflow (for instance, by multiplying $X = +127 \cdot 2^{-7}$ by $Y = +97 \cdot 2^{-7}$ there occurs, in cascade, six cases of overflow in all additions, except the first one) is "benign", i.e. the value introduced in A[7] is pre-established, as shown above, and, consequently, the phenomenon can be ignored. In case of the new Booth procedure, the overflow requires a special analysis, its occurrence being closely correlated with the signs of the values operated upon—in A[7] for the partial products, and in M[7] for the multiplicand—as well as with the "history" of the operations recorded through the state of the flag R. As far as flag R is concerned, we can observe in Fig. 3.20 that by scanning X in the convenient direction, when R becomes 0, an addition is executed followed by a shift and, as long as R remains 0, additions can be executed (but only additions, and no subtractions) followed by shifts, or only shifts. Similarly, if X is scanned in the same convenient direction, when R becomes 1, a subtraction is executed followed by a shift and, as long as R remains 1, substractions can be executed (but only subtractions, and no additions) followed by shifts, or only shifts.

Before analysing the cases that may occur, in the given context, we should like to add some notations to the above observations. Thus, by R we denote the status of the historic flag, by OVR the status of an imaginary flag indicating the status of

Fig. 3.22 Table synthesizing the specific cases of the modified Booth's procedure

R	OVR	A[7]	M[7]	NA[7]
0	0	0	0	0
0	0	0	1	0^*
0	0	1	0	1^*
0	0	1	1	1
0	1	0	0	-
0	1	0	1	1
0	1	1	0	0
0	1	1	1	-
1	0	0	0	0^*
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1^*
1	1	0	0	1
1	1	0	1	-
1	1	1	0	-
1	1	1	1	0

overflow, by A[7] the status of the most significant rank (sign) of a partial product, all these three statuses being considered at the final moment of an addition or subtraction step and, finally, by M[7] we denote the perpetual state, during multiplication, of the sign rank of the multiplicand. Through the exhaustive examination of the binary combinations of these variables, we obtain the table from Fig. 3.22. The last column of this table corresponds to the variable associated with the next value which is introduced in A[7] (next A[7], NA[7]) within the right-shift process. The values corresponding to NA[7] are obtained through investigations executed according to the model described below. Thus, let us consider, to begin with, the case of additions (R = 0) and associate 0 with the old value (previous to addition) from rank A[7], as well as with the (perpetual) value from rank M[7], combining these with all the possible pairs ((0, 0), (0, 1), (1, 0) and (1, 1)) for A[6] and M[6], under circumstances in which, on their addition, carry is captured or not (carry = 1, and 0 respectively). The result consists of the four situations from Fig. 3.23, the new value (following addition) of A[7], as well as the OVR variable (by operating EX-OR on the carry values that result from ranks 6 and 7) being evaluated for each of them. For the pair (OVR, A[7]) (0, 0) and (1, 1) will be obtained in both situations, because if two positive numbers are added, the result has to be positive, which is obvious for the pair (0, 0). In the other case, (1, 1), the result is negative (the new A[7] = 1), but OVR = 1 is also generated, which prevents NA[7] from becoming 1 and, consequently, for both cases, NA[7] = 0 (refer to the quartets (0, 0, 0, 0) and

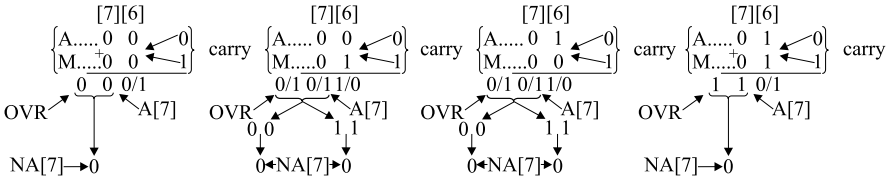


Fig. 3.23 Detailed analysis for two particular cases specific to the modified Booth's procedure

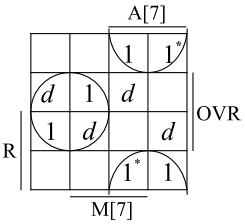


Fig. 3.24 Minimization of the logical equation of NA[7] as function of A[7] and OVR

(0, 1, 1, 0) from Fig. 3.22). The above-described investigation for the combination (0, 0) corresponding to the pair (old A[7], M[7]) shall be repeated, when R = 0, also for the other combinations ((0, 1), (1, 0) and (1, 1)), and then the entire operation has to be repeated for R = 1. Things can be simplified to a certain extent, because case R = 1 (subtraction) and M[7] = 1 may be conflated with R = 0 and M[7] = 0, but, generally, it is recommended that these investigations be made with computer aid.

Column NA[7] from Fig. 3.22 contains some defined logical values of 0 and 1, some of which have been marked with "*" to which we will return. On the other hand, for some of the input binary quartets (R, OVR, A[7], M[7]) the sign "–" highlights impossible situations, namely that in these cases overflow cannot be generated. If we insist, for instance, on the quartet (0, 1, 0, 0), it can be observed that for M[7] = 0, when A[7] = 0 OVR = 0 results, the same way as, when A[7] = 1, OVR = 1 results, so that the occurrence of the ((OVR, A[7]) = (10)) combination is excluded (a fact partly demonstrated in Fig. 3.23). If we adopt, corresponding to the given quartets, a don't care, *d*, logic value, we can obtain the minimized Boolean expression of NA[7] using the Karnaugh map from Fig. 3.24. The favorable covering of the binary units in combination with the *d* value leads to the following expression for NA[7]:

$$\mathrm{NA}[7] = \mathrm{A}[7] \cdot \overline{\mathrm{OVR}} + \overline{\mathrm{A}[7]} \cdot \mathrm{OVR} = \mathrm{A}[7] \oplus \mathrm{OVR} \tag{3.18}$$

The conclusion synthesized in relation (3.18) could be anticipated, namely that in A[7] the previous value from the respective rank is recirculated according to the model and with the motivation from the first Booth procedure, except in case of overflow. Corresponding to this last case, the obtained value, after the execution of the addition or subtraction, in A[7] is not correct (on the addition of two positive numbers A[7] = 1 results, and on the addition of two negative numbers A[7] = 0

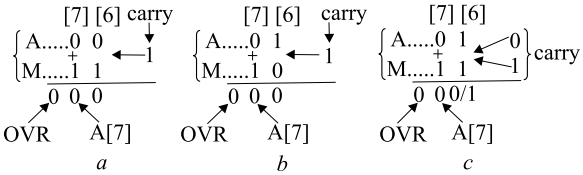


Fig. 3.25 Detailed analysis of impossible particular cases specific to the modified Booth's procedure

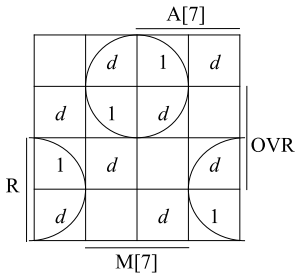


Fig. 3.26 Minimization of the logical equation of NA[7] as function of R and M[7]

results), but this anomaly is surmounted by application of the EX-OR operation to the respective value with OVR = 1.

Let us indicate for the mechanisms specific to the new Booth procedure, an insight by which we aim to obtain some synthesis variants, at least equivalent as far as performance is concerned. Thus, let us refer to the binary quartets whose values for NA[7] have been marked with "*" in Fig. 3.22 and, for these cases, let us perform analyses of the type accomplished in Fig. 3.23. Even if for NA[7] defined logic values result, the respective binary combinations cannot appear during the procedure's execution, due to the impossibility of overflow generation as in the case of the quartets to which corresponds "−". Let us consider, for instance, the combination (0, 0, 0, 1) from Fig. 3.22, which, if approached according to the model from Fig. 3.23, leads to the restriction of the cases of interest to those presented in Fig. 3.25, but none of them can appear in reality. Thus, if we concentrate on the situation *a* (Fig. 3.25) we find out that, through right-shift, the absolute value of the partial product will be equal to half of the respective value, at the most, for all the practical situations when R = 0 (in fact, the value of multiplicand *Y* is subtracted from a value which is, anyway, smaller than *Y*). Thus the occurrence of carry = 1 is excluded, which implies the elimination of the combination of R = 0 and M[7] = 1 with OVR = 0 and A[7] = 0. Through similar reasoning the impossibility of the appearance of the cases *b* and *c* can be stated, as well (Fig. 3.25), as they would correspond to some previous conditions of overflow (A[7]A[6] = 01 following the shift of A). Since, also, the other quartets with values marked with "*" for NA[7] (Fig. 3.22)—namely (0, 0, 1, 0) behaviorally equivalent to (1, 0, 1, 1), and (1, 0, 0, 0) behaviorally equivalent to the previously discussed (0, 0, 0, 1)—subject to similar analysis, lead to the same conclusion of the impossibility of their appearance during the procedure, the synthesis for the Boolean function NA[7] can be modified. Figure 3.26 presents the new Karnaugh map where, for the previous binary units corresponding to the quartets (0, 0, 1, 0) and (1, 0, 1, 1) appear, following

```
                multiplier 5
                declare register A[7:0], Q[8:0], M[7:0], COUNT[2:0], R;
                declare bus INBUS[7:0], OUTBUS[7:0];
      BEGIN: A:=0, COUNT:=0, R:=0,                                  {c0}
      INPUT: M:=INBUS;
                Q[7:0]:=INBUS[7:0], Q[8]:=INBUS[7];                 {c1}
  ZEROTEST: if ORQ=0 then go to OUTPUT,
                if ORM≠0 then go to TEST1, else Q:=0, go to OUTPUT;  {c2}
      TEST1: if ( M[7]=1 and Q[0]=0 ) then
                  A[7]:=0, A[6:0].Q:=A.Q[8:1], COUNT:=COUNT+1, go to TEST1;   {c3, c4}
      TEST2: if R=0 then begin
                  if Q[1]Q[0]=01 then A:=A+M;                       {c5}
                  if Q[1]Q[0]=11 then A:=A-M, R:=1, else go to TEST3;   {c5, c6, c7}
                        end
                if R=1 then begin
                  if Q[1]Q[0]=00 then A:=A+M, R:=0;                 {c5, c8}
                  else if Q[1]Q[0]=10 then A:=A-M;                  {c5, c6}
                        end
                if COUNT7=1 then go to OUTPUT,
      TEST3: A[7]:=R ex-or M[7], A[6:0].Q:=A.Q[8:1],
RIGHTSHIFT: COUNT:=COUNT+1, go to TEST2;                            {c3, c9}
INCREMENT: OUTBUS:=A, Q[1]:=0;                                      {c10}
     OUTPUT: OUTBUS[7:0]:=Q[8:1];                                   {c11}
        END:                                                        {END}
```

Fig. 3.27 Description of two's complement binary multiplication based on modified Booth's procedure

the above presentation, don't care logic values d, values which also appear for the pair of quartets $(0, 0, 0, 1)$ and $(1, 0, 0, 0)$. Using the d symbols in a different way as compared to that suggested for Fig. 3.24, the covering of the binary units leads to the following expression for NA[7]:

$$\mathrm{NA}[7] = \mathrm{R} \cdot \overline{\mathrm{M}[7]} + \overline{\mathrm{R}} \cdot \mathrm{M}[7] = \mathrm{R} \oplus \mathrm{M}[7] \tag{3.19}$$

In accordance with (3.19), the occurrence of overflow is completely masked, and can be totally ignored in the synthesis of the multiplication device. Obviously, in terms of circuitry, the gain is not spectacular (in the last analysis, an EX-OR gate), but, on the whole, if performance is taken into account (the slowness of EX-OR gates as compared to other gates being known, whatever the technology [Wake00]), and by using (3.19), an improvement may result. This may be partially or totally lost when we have a negative multiplicand Y (M[7] = 1) and multiplier X has in its least significant positions one or several 0 bits. Under these conditions, through successive shifts of X until its first 1 bit is reached, in A[7] 0 should be introduced (NA[7] = 0), but through (3.19) it results $\mathrm{NA}[7] = 0 \oplus 1 = 1$ and, an error is introduced, which must be avoided by special measures by the multiplier.

Having discussed the specific aspects regarding the modified Booth procedure, we present in Fig. 3.27, in terms of the already familiar language, the code sequence for the new method. The result of the synthesis of the associated hardware device is given in Fig. 3.28. In comparison with the other multiplication devices, it can be noticed that register Q is on 9 bits, the same as in Fig. 3.16. This time, the additional

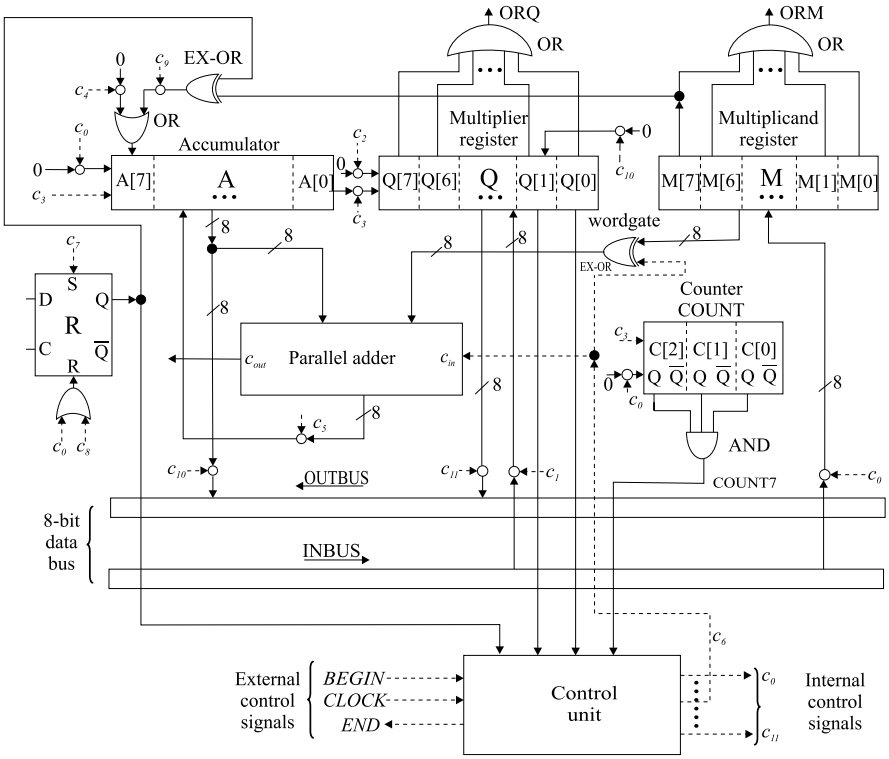


Fig. 3.28 Block diagram of a sequential two's complement binary multiplier based on modified Booth's procedure

rank is Q[8] and it is intercalated between registers A and Q, being used for the initial storage of the sign. In fact, the sign of multiplier X is doubled, appearing from the beginning in the ranks Q[8] and Q[7]. On the other hand, in this algorithm a test has been introduced to check whether one of the operands is zero, a situation in which the zero result is delivered directly to the OUTBUS bus. If $Y = 0$ and $X \neq 0$ there shall be executed the additional operation of initial clearing of the content of the register Q, through the signal c_2, on account of the fact that the result is in the double register A.Q[8:1]. Following these operations, provided at ZEROTEST, it will be checked whether we are in the previously noticed case of the multiplication of a negative multiplicand Y (M[7] = 1) by an X which has in its "tail" one or several zeros. Thus, at statement TEST1, rank Q[0] is tested and when it is 0 then in A[7] a 0 has to be introduced through rightshift. But the essential loop of the algorithm (the loop which begins at TEST2) implies, as required by both Booth procedures, the checking, at a certain moment, of the binary pair from the least significant ranks of Q and, consequently, the testing operation has not been modified. Finally, the last distinctive element of the synthesis from Fig. 3.28 is represented by the flag R. Based on Eq. (3.19) the state of flag R is operated on by EX-OR along

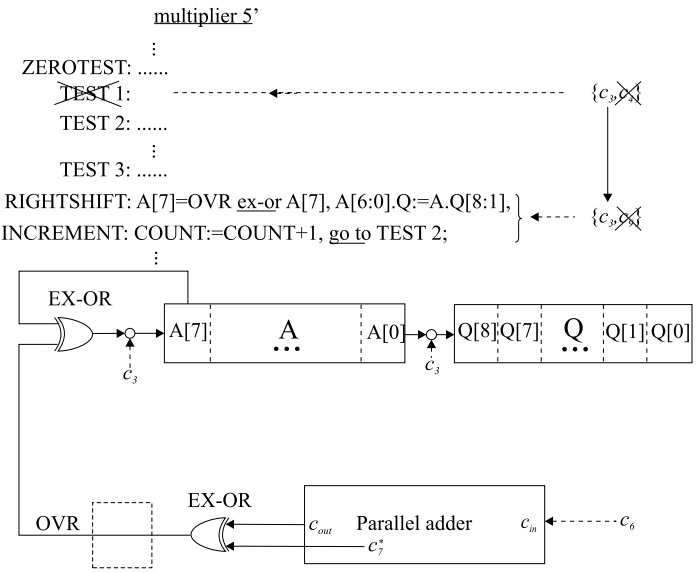


Fig. 3.29 Alternatives to the description and block diagram corresponding to the modified Booth's procedure

with the state of M[7], giving the particular value which is introduced to A[7]. More precisely, in the rank A[7] (whose state is initialized by c_0, together with the other ranks of A) either 0, through c_4, or the value produced by EX-OR, through c_9, is introduced. These two control signals are provided to occur, at timely moments, simultaneously with c_3. This last signal has the task to "push" to the right the binary chain starting with the bit stored in A[7] and ending with the one from Q[1]. Besides these operations which refer to register A (Fig. 3.28) the particular implementation of Boolean equation (3.17) should also be mentioned. Thus, without loss of generality, a flip-flop of type D is chosen using its asynchronous inputs of set (S) and reset (R) (Fig. 3.28), this representing, obviously, only one of the possible solutions. Having in view the investigation of the alternative synthesis solutions, let us turn to account the conclusions of the previous debate regarding the value to be introduced in A[7]. The modifications required are "grafted" both onto the code sequence from Fig. 3.27 and the synthesis from Fig. 3.28. These are summed up in Fig. 3.29. Thus, as far as the code is concerned, the statement provided by label TEST1, implicitly the signal c_4, is given up. This can be done because the problem of shifting over the 0s "from the tail" of X, when Y is negative is correctly solved by Eq. (3.18), OVR being 0. Certainly, in the statement RIGHTSHIFT, (3.19) is substituted by (3.18), with the elimination of c_9. The shift on the entire length A.Q, including the shift toward rank A[7], is assured through c_3 (refer to the circuit fragment from Fig. 3.29). This technical solution, at first sight simpler, can cause a global performance degradation, because the chain of two EX-OR gates (the first generating the variable OVR through EX-OR operation on c_{out} and the carry input in the sign bit, c_7^*, and the sec-

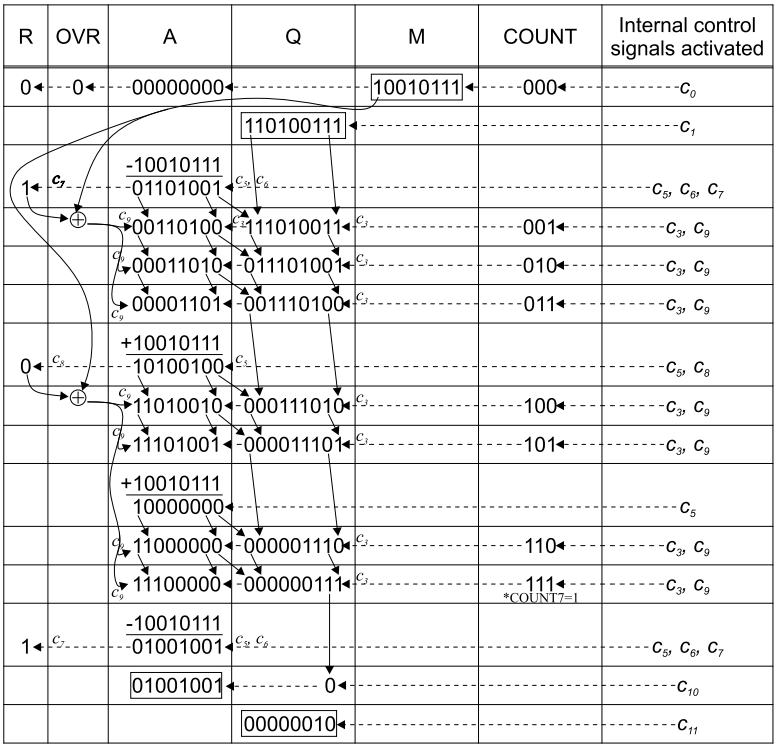


Fig. 3.30 Example of a binary multiplication in two's complement code by the modified Booth's procedure with microoperations' control signals activation

ond implementing (3.18)), might require an increase of the CLOCK pulse period. From the point of view of a reliable operation, mention should be made that an OVR flag in the dotted position in Fig. 3.29 is recommended. Its setting would be done, at each activation of the parallel adder, through a signal c_5 (properly shifted to ensure the covering of the signal propagation on the worst paths). Consequently, if the two versions are compared, mention should be made that they are in a quasiequilibrium, the decisive factor in the choice being the available circuitry technology.

Similar to the other multiplication methods, in Fig. 3.30 the modified Booth procedure is applied to the same example presented in Fig. 3.9, Fig. 3.13 and Fig. 3.17. The sequence of operations is exactly in accordance with the binary structure of the canonically recoded multiplier X_{MB} (Fig. 3.21), scanned from lsb to msb. We also mention that in Fig. 3.30 the OVR column has been provided, although this example is subject to the procedure from Fig. 3.27. The value of the OVR variable has no longer been marked, in the case of the example it maintains the initial value 0, which, operated on by EX-OR with the value of the bit A[7], is unchanged, this value being introduced (as NA[7]) in the most significant rank of A.

Finally, as far as the complexity and performance of the modified Booth procedure is concerned, mention should be made that it belongs to the same class as the

already presented algorithms, but it may lead to substantial increases of throughput capacity in case of favorable binary structures (as regards canonical recoding) of the multiplier (mainly, on a large number of bits, the probability to turn to account the acceleration characteristics of the method increases). Such an improvement, although reduced, i.e. consisting of the elimination of one activation of the parallel adder, can also be observed in the example from Fig. 3.30.

3.5 Binary Multiplication Process Speedup by Increasing Radix Value

Modern circuitry for arithmetic does not directly use Booth recodings, but they are the basis for the understanding of and approach to these problems in number systems with radix value larger than 2, the so-called higher radix Booth's recoding [Parh00, SeMM05, HePa03]. Obviously, the increase of radix r results in the reduction of the number of digits and an algorithm which achieve the multiplication in a digit-at-a-time manner will require a smaller number of cycles, as r increases. Thus, an operand on n bits can be interpreted as having $\lceil n/2 \rceil$ digits in $r = 4$, $\lceil n/3 \rceil$ digits in $r = 8$, etc., where the bars $\lceil\ \rceil$ signify the smallest integer larger or equal to the value between the bars. However, depending on the increased value of r, the algorithm has to ensure the simultaneous inspection of several digits, 2 at $r = 4$, 3 at $r = 8$, etc.

Supposing, to begin with, $r = 4$ and integers, let us adapt the iteration (3.2) to the new conjuncture:

$$\begin{cases} P_i := P_i + x_i Y, \\ P_{i+1} := 4^{-1} P_i \end{cases} \tag{3.20}$$

where, besides the already specified notations, mention should be made that this time x_i may take the values 0, 1, 2 and 3.

If the forming of the multiples by 0, 1 and 2 of Y is easy to achieve ($2Y$ represents the binary configuration of Y shifted by one bit to the left), as regards the value $3Y$, it requires at least a supplementary addition, because $3Y = Y + 2Y$. A first implementation option for the algorithm based on (3.20) consists of the preceding computation of $3Y$ and its storage in a register for subsequent use.

Before the detailed presentation of a procedure based on (3.20), it is time to mention that, as regards the fundamental algorithms, we have appealed to the employment of a detailed presentation both of the algorithms and of the hardware devices which achieve their implementation. This strategy has been used to enable understanding of the inner workings of the mechanisms, so that things may be as transparent as possible for their immediate physical-electronical implementation. Attempting to ensure as consistent a presentation as possible, we shall take over the notations that have become recognized in this work, but we shall adhere, from now on, to a more synthetical presentation, insisting, also to avoid monotony, only upon the elements that are specific to the various methods or diagrams. However, we shall endeavour to render the work a degree of clarity which, together with the inserted

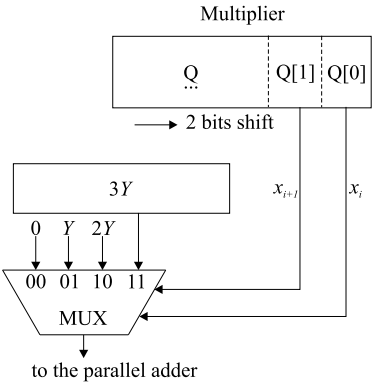


Fig. 3.31 Conceptual diagram for radix-4 binary multiplication

x_{i+1}	x_i	x_{i-1}	x_{Bi+1}	x_{Bi}	$x_{4Bi/2}$
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	1	$\overline{1}$	1
0	1	1	1	0	2
1	0	0	$\overline{1}$	0	$\overline{2}$
1	0	1	$\overline{1}$	1	$\overline{1}$
1	1	0	0	$\overline{1}$	$\overline{1}$
1	1	1	0	0	0

Fig. 3.32 Table with the characteristics of the radix-2 and radix-4 Booth's recoding

details, may allow, from case to case, a deeper understanding of the problems up to the level of the employed technology.

Thus, the multiplication procedure in radix 4, based on the precomputation of $3Y$ and on the iteration (3.20), can be associated with the diagram from Fig. 3.31 [Parh00]. In it, the control of the selection inputs of a multiplexer, MUX, can be observed through the pair of bits, i.e. the current one x_i, and its left neighbour, x_{i+1}, both stored in the least significant ranks of a register alias Q. Through MUX pass towards the parallel adder, depending on the combination from Q[1]Q[0], the multiples of Y (0, Y, $2Y$ or $3Y$) to be added, according to (3.20), to the partial cumulative product. Thus, the precomputed value $3Y$, stored in an additional register, is added to the content of a register alias A, when $x_{i+1}x_i = 11$. It is also important that the shift, when $r = 4$, is made by 2 bits (2-bits shift) to the right, which considerably accelerates the process.

The peculiarities of the $r = 4$ operation being pointed out, let us analyse, for this value of the number system radix, what is implied by Booth recoding of the multiplier X, namely the first such transformation, which is exemplified in Fig. 3.14. Thus, we consider a triplet of successive bits of X—which we denote by x_{i+1}, x_i, x_{i-1}—then a pair of successive bits of X_B—which we denote by x_{Bi+1}, x_{Bi}—associated with those from the triplet and, finally, we denote by $x_{4Bi/2}$ the bit assigned to the pair and which belongs to the Booth recoded form with radix 4, X_{4B}.

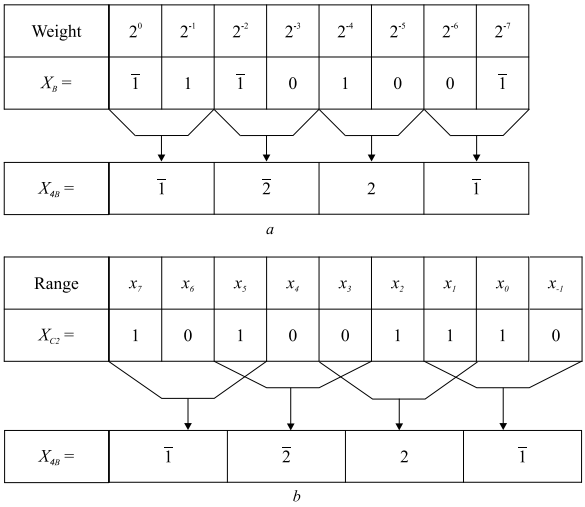


Fig. 3.33 Example for obtaining radix-4 Booth recoding

Exhaustively scanning the combinations of the triplet bits, results in the values from the table presented in Fig. 3.32. If we consider that each triplet is independent, the values from columns x_{Bi+1} and x_{Bi} can be immediately obtained according to the rules established for Booth recoding (refer also to Fig. 3.14). The values from the last column, $x_{4Bi/2}$, result, first, from the values x_{Bi+1} and x_{Bi} by taking into account the weights assigned to them, as well as their signs. Thus, for the pair $x_{Bi+1}x_{Bi} = \overline{1}1 = (-1)2^{i+1} + (+1)2^i = -2^i$, which leads, according to rank i (it has the weight 2^i) of recoding in radix 4, to the value $x_{4Bi} = \overline{1}$. Similarly, for the pair $x_{Bi+1}x_{Bi} = \overline{1}0 = (-1)2^{i+1} = (-2)2^i$, we obtain $x_{4Bi/2} = \overline{2}$. In order to perform Booth recoding radix 4, Booth recoding bits radix 2 are grouped in pairs and the correspondences that exist in the table from Fig. 3.32 are used. For instance, let us consider our multiplier $X = (-89)2^{-7}$ and its Booth recoding radix 2 (Fig. 3.14), a situation in which, in Fig. 3.33a, is obtained, according to the established rules, its recoded form radix 4, X_{4B}. But, passing through the recoded form radix 2 is not necessary, because the code conversion can be done directly by taking into account the triplets (x_{i+1}, x_i, x_{i-1}) and x_{4Bi} from the last column (Fig. 3.32). Here occurs an aspect connected to the particular value of the radix, namely the way in which the bits of the form X_{C2} are grouped in triplets. Since $r = 4$, the jump must be made over 2 bits and consequently the need appears to overlap the triplets at the level of one bit. We recall that, for this type of Booth recoding in radix 2 the extension of the form X_{C2} with a 0 bit to the right of the lsb ($x_{-1} = 0$) is typical. This bit is part of the rightmost triplet, being its lsb. The extension by one bit to the right, correlated with the use of the digits ± 2 in the recoding in $r = 4$ (which requires the extension by one bit to the left of the most significant part of the cumulative product and of the adder, as will be seen below), requires the correction of the Booth recoded form radix 4, for fractional numbers, through multiplication by 2. Thus, in Fig. 3.33b, are presented the overlapping triplets [HePa94] in the case of our example multiplier, whose weighted value X_{4B} is computed, under the

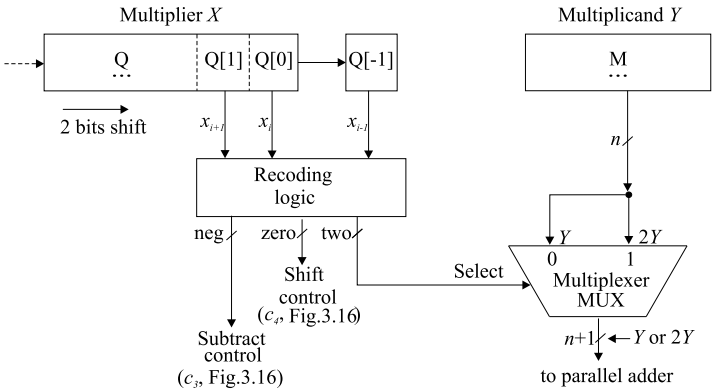


Fig. 3.34 Conceptual diagram for implementing the procedure based on radix-4 Booth's recoding

specified conditions, by: $X_{4B} = ((-1)4^{-1} + (-2)4^{-2} + (+2)4^{-3} + (-1)4^{-4})2 = = (-4^3 + (-2)4^2 + (+2)4 - 1)2^{-7} = (-89)2^{-7}$.

In Booth recoded form radix 4, there may be observed (e.g. in Fig. 3.33) the reduction by half of the number of digits (obviously, with favorable consequences as regards the performance of the procedure), but also the requirement of generating only the multiples $(0)Y$, $(\pm 1)Y$, $(\pm 2)Y$ (which are obtained through simple operations of complementing and/or shift). Consequently, we do not need multiple $3Y$ (it can be obtained only through a considerable time investment, implying the above mentioned addition) which is required by the previous multiplication procedure radix 4.

A possible implementation of the algorithm based on Booth recoding radix 4 is given in Fig. 3.34 (adapted from [Parh00]). In it registers Q and M can be recognized, at register Q being pointed out the 3 ranks we are interested in—Q[1], Q[0] and Q[–1]—where, at a certain moment, the triplet to be analysed (x_{i+1}, x_i, x_{i-1}) appears, as well as the fact that, at each shift, the contents of the double register A.Q is shifted by two bits to the right. The outputs of the three least significant ranks of Q are applied to the recoding logic. Its synthesis, executed according to the table from Fig. 3.32 (except columns x_{Bi+1}, x_{Bi}), can be achieved in a simple and efficient way, by generating three output functions, namely: "zero", obtained through the OR function between the decoded combinations corresponding to the first line ($\overline{Q[1]}\,\overline{Q[0]}\,\overline{Q[-1]}$) and to the last line ($Q[1]Q[0]Q[-1]$) of the table (Fig. 3.32), which activates the shift control (i.e. signal c_4—Fig. 3.16), "neg", obtained through the OR function (minimized) between the combinations corresponding to the triplets (1, 0, 0), (1, 0, 1) and (1, 1, 0)—having associated negative values for $x_{4Bi/2}$—which activate the subtract control (i.e. signal c_3—Fig. 3.16) and, finally, "two", obtained through the OR function between the combinations corresponding to the triplets (0, 1, 1) and (1, 0, 0) (Fig. 3.32)—having associated values ± 2 for $x_{4Bi/2}$—which activate the select signal of the multiplexer MUX. The multiplexer consists of $(n + 1)$ cells of the type from Fig. 3.35, allowing to pass towards the $(n + 1)$ inputs of the parallel adder, when "two" is active (1), the logical val-

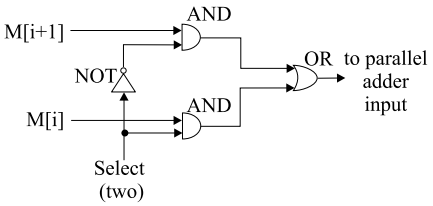


Fig. 3.35 Detailed diagram for the MUX multiplexer cell

A	Q				M	COUNT
	...	Q[1]	Q[0]	Q[-1]		
00000000					10010111	00
	101001	1	1	0		
-110010111						
001101001						
000011010	011010	0	1	1		01
+100101110						
101001000						
111010010	000110	1	0	0		10
-100101110						
010100100						
000101001	000001	1	0	1		11
-110010111						
010010010						
01001001						
0	000001	0	0	1		

Fig. 3.36 Example of a binary multiplication by the procedure based on radix-4 Booth's recoding

ues of the outputs of the ranks shifted by one bit to the left of the multiplicand Y from register M (i.e. those which correspond to $2Y$), and, when "two" is not active (0), the logical values from the outputs of the M ranks (i.e. those which correspond to Y). Index i varies between (-1) and (n), with $\mathrm{M}[-1] = \mathrm{M}[n] = 0$ and M[0] to $\mathrm{M}[n-1]$ representing the n bits of M, so that to the parallel adder are connected $(n+1)$ lines with the contents of M (i.e. Y, when two $= 0$) or, as applicable, with the contents from M shifted to the left (i.e. $2Y$, when two $= 1$). Besides the additional adding cell in the parallel adder, one more rank also has to be added to the register used for forming the cumulative product, alias register A (Fig. 3.16). The fact that Y penetrates through the MUX when we have multiple 0 ($Y = 0$), does not disturb us at all because, in this situation, the control unit is designed in such a way that it will not generate the homologous signal of c_2 from Fig. 3.16, but only that one which corresponds to the shift (c_4).

In Fig. 3.36, we have applied the procedure based on Booth recoding radix 4 to the example treated by the previous methods. There can be observed the extension of register A by one bit to the left (to enable the addition or subtraction of the values of $2Y$), which implies, for the operation with $(-Y)$, the extension of the sign bit by

one position to the left. Similarly to the algorithm from Fig. 3.15, and the device from Fig. 3.16 respectively, the right-shift is achieved through the recirculation, in the most significant rank of A, of its old contents. It can also be observed that when right-shift is performed, it is done, each time, on 2 bits, which implies the saving of one rank at the iterations counter COUNT. A last remark refers to the returning of the result obtained in the 16 ranks of register A.Q (9 from A, and 7, the most significant ones, from Q). This fact is connected with the above-mentioned correction that has been employed in the computation of value X_{4B} (Fig. 3.33).

Multiplication devices that use radices r larger than 4 (8, 16, or even larger) can be synthesized, according to the schematic model from Fig. 3.31, but the hardware required for the generation of the multiples ($3Y$, $5Y$ and $7Y$, in case $r = 8$) becomes complex, annulling through the delays on it, either mostly, or even totally, the gain in speed due to the smaller number of cycles. However, attractive hardware implementation solutions are also possible in situations when the radix of the number system is larger than 4 [ErLa04].

3.6 Binary Multiplication Speedup Using a Single Carry-Save Adder

As has already been seen, for a parallel adder, created even in the form of a binary tree based on the CLA (carry lookahead adder) approach, when the numbers that need to be added have a considerable number of bits, the time required for the execution of the operation, which has to be covered by the CLOCK period, becomes prohibitive. Thus, supposing that the numbers have 64 bits, when the multilevel carry lookahead approach implemented through a CLA tree adder is used, the time interval required, evaluated in 2.2.4 for specified hypothetical conditions, corresponds to the delay on $4 \log_2 64 = 24$ logical levels, a value which is significant even when rapid technologies are available. The studied sequential algorithms activate such an adder for a significant number of times, which, as has already been seen, depends on the particular binary structure of multiplier X. Moreover, these algorithms, which require variable times for the multiplication process, create problems in the synchronization with the CPU, in the efficient application of the pipelining design method, and in compiler optimizations [HePa03].

The above-signalled drawbacks can be greatly reduced by including a carry save adder (CSA) in the multiplication device structure, according to the model from Fig. 3.37. The CSA, as we already know and may be observed in the figure, consists of n independent full adder cells (FAC), n being the number of bits in the operands. Each FAC has three inputs (one bit for each number to be added, and one carry bit from the previous rank) and two outputs (one for the sum bit and another for the carry bit towards the next rank). The logical values which exist at the two outputs of the n FACs are stored in two registers of n bits, one dedicated to the sum bits, denoted in Fig. 3.37 by A (by analogy with the Accumulator register from the already studied devices), and the other dedicated to the carry bits, denoted in

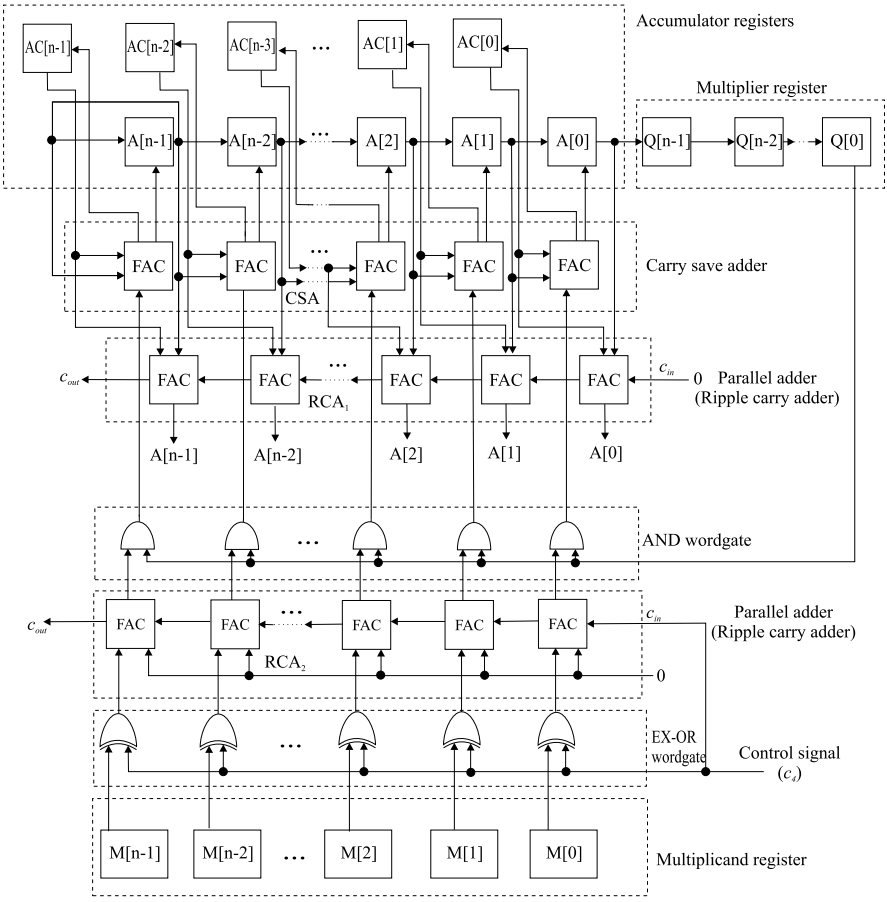


Fig. 3.37 Block diagram of a sequential binary multiplier with a single CSA for the Robertson's procedure implementation version

Fig. 3.37 by AC. The contents of the two registers are added to obtain the value of the more significant half of the partial cumulative product. The AC register appears as an additional hardware investment, redundant in comparison to the already presented solutions, a fact which justifies naming this multiplication procedure redundant [HePa03].

We shall discuss the implementation version corresponding to the Robertson method (Fig. 3.37), but the modifications to adapt it to the Booth procedures are minor, and they can easily be imagined. In a similar way to other devices, initially multiplier X is loaded in register Q, and multiplicand Y is loaded in register M, where it will remain throughout the entire development of the procedure. Also initially, the contents of accumulator registers (A and AC) will be cleared, as will that of the iterations counter COUNT, which can be observed in our example, presented in Fig. 3.38. In this case, as well, A and Q form a double length register, having the

right-shift capacity. We have a layer of AND logical circuits on the entire length of the word, whose role is to allow the application as input to the CSA, depending on the momentary value existing in Q[0], either of the contents of M, or of the value 00...00. The other two binary vectors, also representing inputs to the CSA, are the one contained in register A following the rightshift (refer to $A.Q2^{-1}$, Fig. 3.38) and that from register AC.

In this way, to the current value from A, the value from M or 0 is added, as well as the carry vector generated by the previous rank. At the CSA outputs, there are available the sum and carry words (to the next rank) after the signals cross only two logic levels, specific for a FAC—whatever the value of n—which is a substantial gain as compared to the 24 levels, for $n = 64$, specific for a CLA tree adder solution. Moreover, a fact which can also be observed from the example (Fig. 3.38), all the steps—except, possibly, the last two ones, which will be discussed later—have the same duration, enabling the surmounting of the previously mentioned disadvantage regarding the dependence on the binary configuration of the multiplier X operand.

The gain in performance and organization brought about by the use of CSA is partially balanced by the AC "redundant" register, as well as by the two supplementary conventional parallel adders, which, to maintain a minimum clarity of the figure, have been represented as the ripple carry adder (RCA) type in Fig. 3.37. The first of them, denoted by RCA_1, is meant to add, in the last step of the procedure, to the shifted sum binary vector from register A (refer to $A.Q2^{-1}$, Fig. 3.38), the carry vector (from register AC), to obtain, in this way, the more significant part of the product. More precisely, if we refer to the example from Fig. 3.38, it is the 7 bits from A that we are talking about (the msb is ignored because an additional rightshift is required—COUNT = 1000, as compared to the usual COUNT = 111—to enable the execution of the ordinary final addition, with carry propagation), they being concatenated to the 8 bits from Q to form the final product. Within this context, we underline the need to return, as such, the result of the multiplication operation to the output bus. Mention should also be made that, for the same reason of minimum clarity as regards the circuitry from Fig. 3.37, the outputs of RCA_1 ($A[n-1], \ldots, A[0]$) have not been "sent back" to register A. On the other hand, the second parallel adder, denoted by RCA_2, allows, together with the EX-OR wordgate layer, when the control signal c_4 is activated (denoted this way because it has the same functional role as its homonym from Fig. 3.12), the two's complementing of the contents of register M. This value is needed only in case of a negative multiplier X, corresponding to the correction step (in our case, the last but one step) from the Robertson procedure, when, in fact, the number stored in M has to be subtracted (actually, this number is added as M*, after the contents of an imaginary COUNT becomes 111—Fig. 3.38).

We also mention that the use of multiplexers and additional logic which enables reconfiguration, makes possible the saving of an RCA in the circuitry from Fig. 3.37, but we prefer to keep both of them for greater clarity of the informational flow. Naturally, Fig. 3.37 lacks some structural elements, as well as the logic part meant to facilitate the application of the signals generated by the control unit, which is also missing. Regarding the example from Fig. 3.38, besides the already made observations, note the marking with braces of the contents of the three registers (the shifted

Registers	Accumulator registers	Q	M	COUNT
AC	00000000	1010011(1)	10010111	0000
A	+ 00000000			
M	+ 10010111			
AC	00000000			
A	10010111			
$A.Q2^{-1}$	11001011	1101001(1)		0001
AC	+ 00000000			
M	+ 10010111			
AC	10000011			
A	01011100			
$A.Q2^{-1}$	00101110	01101001		0010
AC	+ 10000011			
M	+ 10010111			
AC	10000111			
A	00111010			
$A.Q2^{-1}$	00011101	00110100		0011
AC	+ 10000111			
0	+ 00000000			
AC	00000101			
A	10011010			
$A.Q2^{-1}$	11001101	00011010		0100
AC	+ 00000101			
0	+ 00000000			
AC	00000101			
A	11001000			
$A.Q2^{-1}$	11100100	00001101		0101
AC	+ 00000101			
M	+ 10010111			
AC	10000101			
A	01110110			
$A.Q2^{-1}$	00111011	00000110		0110
AC	+ 10000101			
0	+ 00000000			
AC	00000001			
A	10111110			
$A.Q2^{-1}$	11011111	0000001(1)		0111
AC	+ 00000001			
M^*	+ 01101001			
AC	01001001			
A	10110111			
$A.Q2^{-1}$	11011011	10000001		1000
AC	+ 01001001			
P	00100100	10000001		

Fig. 3.38 Binary multiplication example by a sequential multiplier with a single CSA for the Robertson's procedure implementation version

A, AC and M or 0) added in CSA mode, a fact which requires the re-writing of the AC contents.

Obviously, the technical solution from Fig. 3.37, with the corresponding example from Fig. 3.38, can have various implementation solutions (besides the above-mentioned ones, for instance, the A.Q rightshift can be replaced by an AC leftshift,

and, consequently, the precaution of returning the final product in the bus can be avoided), our choice taking into account, for the already mentioned reasons, the clarity of the informational flow. Even with this version, which is unfavorable regarding performance, the gain in speed, as compared to the previous solutions, is unquestionable, due to the consistent reduction of the time length of each step (cycle). Let us examine, below, the combination of this improvement with that specific to the previous method based on the increase of the value of the number system radix, consisting of the reduction of the number of steps (cycles).

3.7 Binary Multiplication Speedup Based on Radix 4 and a Carry-Save Adder

Let us combine the two speedup approaches presented above to obtain as substantial a performance gain as possible. Thus, adopting, this time, a synthetical presentation of the type given in Fig. 3.34, but referring to the structural elements from Fig. 3.37, in Fig. 3.39 a multiplication device radix 4 with CSA is outlined (after [Parh00]).

Regarding the new configuration, mention should first be made that the recoding logic is controlled by the same three bits from the ranks Q[1], Q[0] and Q[−1] of a corresponding multiplier Q register. It has the same three outputs as that from Fig. 3.34, with similar observations regarding the logic synthesis. But there are certain differences, namely "zero" is applied to the authorization input "Enable" of multiplexer MUX, which has to be provided with such an input (such as, for instance, the integrated circuit MUX 74LS153 [Yarb97]). Also, a "zero" output does not directly execute the shift control c_4, as in Fig. 3.34, but, when it is not activated, the MUX supplies, at its $(n + 1)$ outputs, the binary vector made up, exclusively, of 0s. This, together with vectors Y and $2Y$, generated in the manner suggested in Fig. 3.35, is supplied to a binary subtracter, synthesized through an EX-OR wordgate layer and a parallel adder, e.g. of RCA type, which, when its "neg" connection is active, forms, as presented in Fig. 3.37, the two's complement for $(-Y)$ and $(-2Y)$. Thus, at one of the three sets of CSA inputs the vectors corresponding to all the multiples provided in the table from Fig. 3.32 are available, i.e. 0, Y, $(-Y)$, $2Y$ and $(-2Y)$. To the other two sets of inputs are supplied the sum and carry vectors which are temporarily stored in registers A and AC of the Accumulator registers block with the general configuration given in Fig. 3.37.

However, the operation based on the Booth recoding in radix 4 requires certain distinctive elements as compared to the structure from Fig. 3.37. Thus, only the most significant $(n - 1)$ bits of the sum binary vector from the CSA output are loaded into the corresponding bits of register A, and in its two least significant ranks (A[1] and A[0]) the binary values from the outputs of an additional parallel adder (AA), represented in Fig. 3.39 of RCA type, are introduced. To the AA inputs, there are supplied, on the one hand, the two least significant bits of the sum vector, and, on the other hand, the least significant bit of the carry binary vector concatenated, to the right, with the bit stored in a carry flip-flop (CFF). This memorizes the binary value

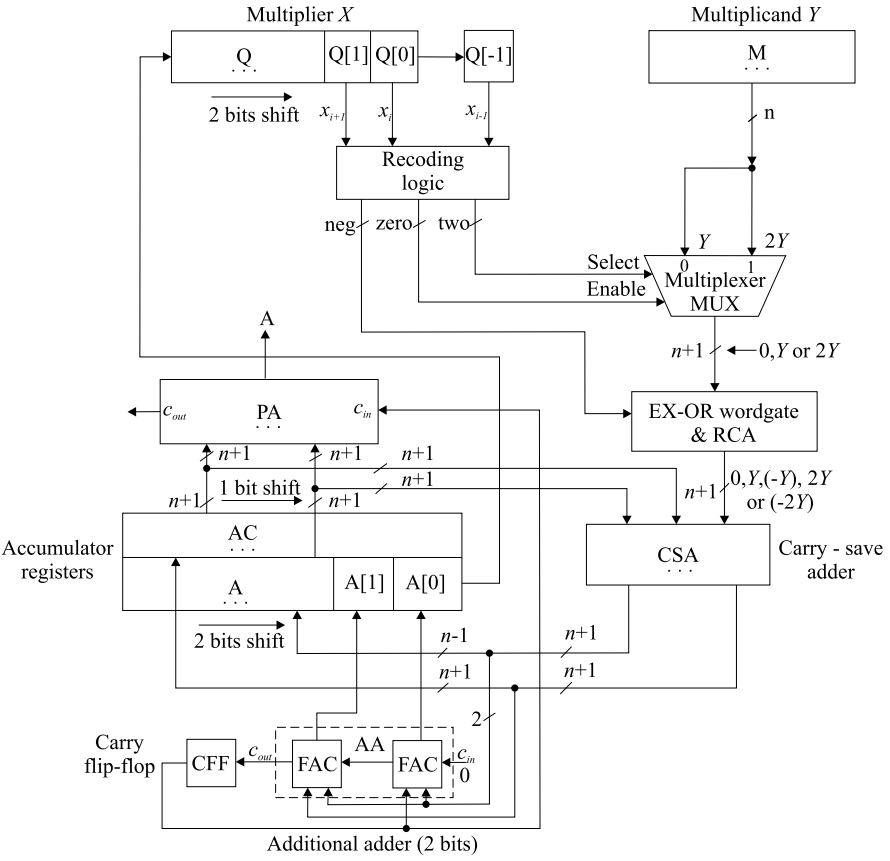


Fig. 3.39 Conceptual diagram of a sequential binary multiplier with a single CSA using radix-4

from the c_{out} output of the AA adder, a carry which may occur in the generation of those final bits of the product which are "pushed", during the right-shift process, from A into Q. These last bits are formed by adding to the two least significant bits of the CSA sum vector the least significant bits of the CSA carry vector, shifted by one position to the left and having attached, to the right, the memorized value (in the CFF) of the carry that has resulted from the previous activation of the CSA and AA adders. Consequently, the CFF output is applied to the lsb rank of AA, but also to the c_{in} input of the parallel adder (PA), which enables the obtaining—finally, through ordinary addition of the two last vectors, the sum and carry ones—of the part from A of the result. The loading of registers A and AC, as well as of the flip-flop CFF, has been arranged to be executed synchronously, which implies a latency covering the time interval required by the signal's propagation on the worst path (usually, the one that passes through the largest number of logic levels) from the combinational logic AA. Regarding the previous observation related to the generation of the real sum vector from the sum and carry CSA words, we also mention

that at the end of each step (cycle) of the procedure, register A.Q is arithmetically shifted by two positions to the right, while register AC has to be shifted by only one position, also arithmetically, in the same direction. Consequently, unlike register AC from Fig. 3.37, register AC has to be provided with shifting capacity. Of course, if the above-described procedure is precisely respected, other implementation versions can be imagined as well.

The application of the above-described procedure can be followed up on the same example operation used to illustrate the other procedures, in Fig. 3.40. Mention should be made that we have resorted to the same denotation and representation conventions used in Fig. 3.38 (M* for the two's complemented contents of M, $AC2^{-1}$ for the one position right-shift of AC's contents, and, by analogy, $A.Q4^{-1}$ for the two positions right-shift of the contents of the A.Q double register, as well as the brace that marks the three binary vectors added in a CSA manner), and in Fig. 3.36 (for the identification of the bits triad Q[1]Q[0]Q[–1] investigated at a certain moment). Moreover, there appear CFF and AA columns which illustrate the added pairs of bits, as well as the c_{out} generated by the addition and stored in CFF. The two least significant bits from A are presented in the same row of the given register, but in the column AA, and the sum bits from AA have no longer been "sent back" to the ranks A[1] and A[0] of register A, this time they have been "pushed" directly to register Q, this only to ensure a minimum clarity of the informational flow from Fig. 3.40.

3.8 About "Parallelizing" of the Sequential Devices for Binary Multiplication

One common feature that has occurred in the studied procedure implementations was the execution of the binary multiplication operation through a sequence of control signals, each of them requiring, in principle, a CLOCK pulse. Regarding the possible solutions, on one hand, there are the so-called "one-bit-at-a-time" procedures (which take the decision for addition-shift, or only for shift, depending on the value of a single bit at a certain moment) in radix 2, whose implementations are the most expensive in terms of number of CLOCK pulses. On the other hand, there are the implementations which require only one such pulse, the operation execution being so-called "completely parallelized". But these solutions require the time intervals needed for the signals' worst paths propagation to be covered by the CLOCK period, as well as the largest number of circuits, and, implicitly, the largest integration area. Between these two extreme possibilities there are procedures which, in comparison to the purely sequential ones, increase the degree of the parallelism, thus enabling the acceleration of the operation execution, and which, as compared to the purely parallel ones, reduce the circuits and integration area. "Parallelization" is obtained, starting with Booth recodings, by simultaneously investigating two bits and, then, as the number system radix increases, an ever larger number of bits. Thus, as has been seen at $r = 4$, three bits are inspected, with the use of multiples of Y in the integers value range $[-2, +2]$, similarly, for $r = 8$, four bits are simultaneously

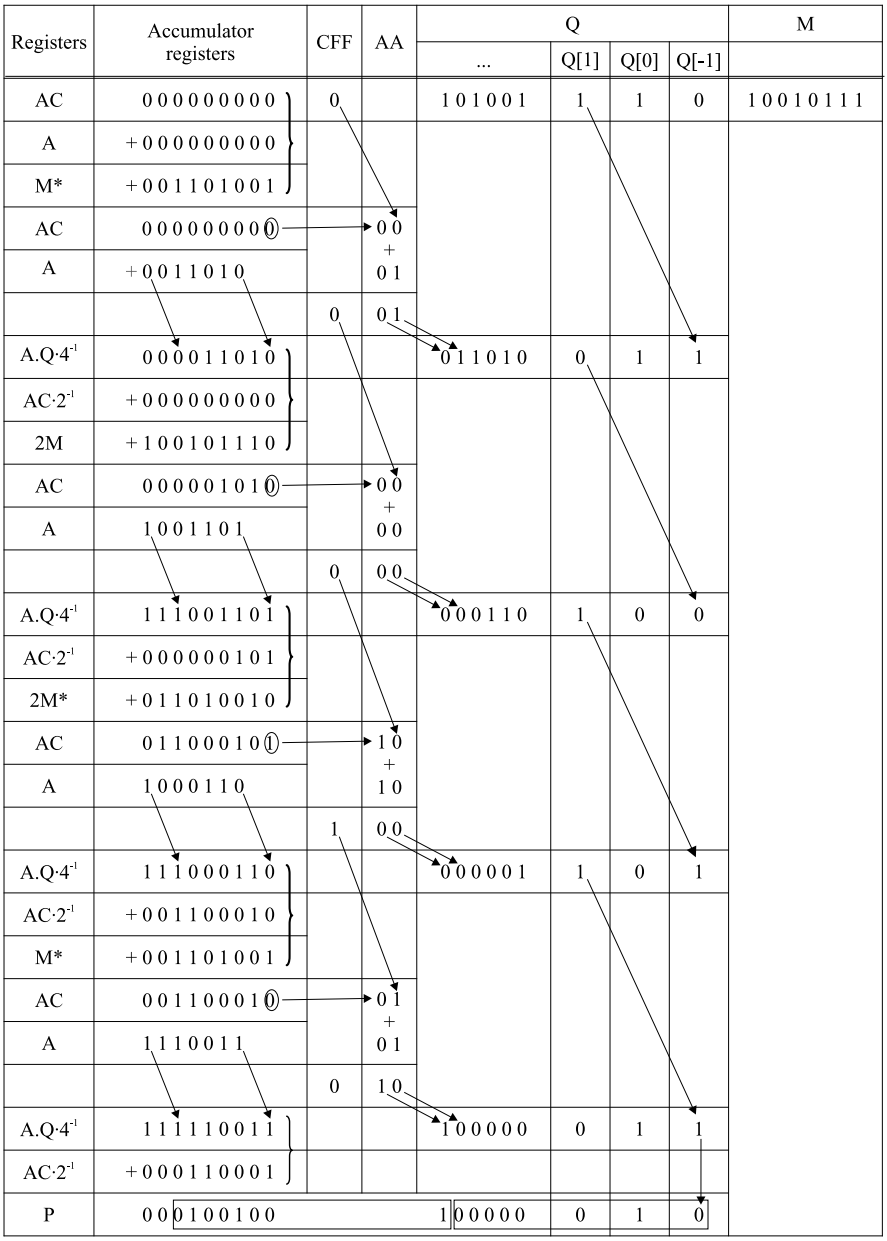


Fig. 3.40 Binary multiplication example by a sequential multiplier with a single CSA and using radix-4

analyzed, the integer interval of Y multiples becoming $[-4, +4]$, and for $r = 16$ the number of simultaneously consulted bits is increased to five, and the interval of Y multiples is increased to $[-8, +8]$. In principle, as the value of r increases, the essential structural elements remain the same as those from Fig. 3.39, with the caveat that the recoding logic becomes more intricate, its synthesis being based on the corresponding extension of a table of the type from Fig. 3.32. Certainly, the multiplexer MUX must also be correspondingly extended to be adapted to the number of positive multiples (plus 0). The blocks EX-OR wordgate & RCA, Accumulator registers (A and AC) and PA are maintained, with the observation that they are constructively dependent only on the value of n. The number of ranks in the AA adder must be modified, as well; for $r = 8$, this will be equal to 3, and for $r = 16$, it will be 4. Obviously, whatever the value of r, c_{out} will be kept in a CFF.

Special attention shall be given to the second fundamental speedup element which contributes, together with the increase in r, to the improvement of the parallelized solutions' performance, i.e. the CSA adder. The gain obtained through the use of CSA adders is undoubtable in comparison to other synthesis methods. However, mention should be made that, as the values of radix r and, implicitly, of the multiples of Y increase, CSA addition requires a larger number of levels. CSA tree arrangements [ErLa04, Oliv01] are often resorted to, which, as will be seen below, reduce the performance improvement. Thus, we have highlighted the tradeoffs which favors the choice of a certain solution. Practically, there is no reason to limit the radix, for instance, to $r = 16$, in [Parh00] there being discussed a multiplier with $r = 256$. The choice of an "optimum" (in reality "better") solution is not a simple task, it being influenced by a multitude of factors, of which the decisive role is played by the circuit technology available, as well as by the use of various artifices for exploiting the CLOCK parameters (three-phase clock [Parh00]).

We should like to refer to the integration complexity of the very large scale integration level (VLSI) of the implementations for the above presented multiplication devices [ITRS01]. Their structural components include, resulting from the various configurations, and mainly that from Fig. 3.39, registers, multiplexers, CSA adders, and a fast parallel adder (which, in Fig. 3.37, has been taken to be of RCA type only for the sake of a simple representation), as well as a generally reduced quantity of random logic for the control synthesis. Among these components, the CSA adder tree has an important influence over the VLSI complexity. Without loss of generality, referring to multipliers without Booth recoding of radix $r = 2^k$, the forming of the sum and carry vectors corresponding to the final product require k CSA adders. Thus, if $r = 4$, and since, as can also be seen in Fig. 3.31, it is necessary to generate multiple $3Y$, it results that we have four input vectors ($Y, 2Y$, and the sum and carry vectors from the previous partial cumulative product) whose CSA addition requires $k = 2$ adders. Similarly, when $r = 16$, it results that in order to form—by means of multiples $Y, 2Y, 4Y$ and $8Y$—all the multiples, it is necessary that we use $k = 4$ CSA adders, which, as shown below, can be interconnected in a tree arrangement, for instance, of Wallace or Dadda type [Parh00]. Generally, such a CSA tree has $(k + 2)$ inputs and the height by CSA levels can be approximated by $\lceil \log_2 k \rceil$, where the bars $\lceil\ \rceil$ have the known significance, indicating the smallest integer number, larger at least equal, to the value of the argument.

As regards the integration of such a CSA tree arrangement, one of the decisive factors is the complexity of the silicon wafer area, which will be denoted by A, this presenting the dependence $A = O(kn)$, where n is the number of bits in each operand to be multiplied. Since, related to A, the CSA tree prevails in comparison to the other structural elements, the final fast adder included, the silicon area requirement for the entire multiplication configuration can be estimated by:

$$A = O(kn) \tag{3.21}$$

On the other hand, investigating the performance, the time complexity corresponding to a CSA tree of Wallace type is given by $O(\lceil \log_2 k \rceil)$ and, since it is activated (n/k) times during a multiplication operation (refer also to the example from Fig. 3.40, even if, in this case, Booth recoding is used), we obtain, for the time component specific to this element of the multiplier device, the dependence $T_1 = O((n/k)\lceil \log_2 k \rceil)$. If to this we add the required component, in the fast mode (e.g. CLA), of final addition given by $T_2 = O(\lceil \log_2 n \rceil)$, then, for the time complexity of the entire multiplier, we have the following dependence:

$$T = O\big((n/k)\lceil \log_2 k \rceil + \lceil \log_2 n \rceil\big) \tag{3.22}$$

Starting from (3.21) and (3.22), and trying to evaluate the integration efficiency, we shall use the well-known metrics AT [Parh00, ErLa04] which, in the case of our multipliers, will have the following form:

$$AT = O\big(n^2\lceil \log_2 k \rceil + kn\lceil \log_2 n \rceil\big) \tag{3.23}$$

At the lower limit of the complexity spectrum, when $k = 1$, we have $AT = O(n\lceil \log_2 n \rceil)$, this being the case of the slower multipliers radix 2. But, if, we refer to the realm of the accelerated devices, for instance $k = 2$, there results $AT \cong O(n^2)$, and for $k = n$ (this being the completely parallel case, when all the bits of multiplier X are simultaneously inspected), there results $AT = O(n^2\lceil \log_2 n \rceil)$. But, it being known that, for an "optimum" design AT is, in the limit, proportional to $n\sqrt{n}$ [Parh00], and since none of the intermediate designs, between the extreme ones mentioned before, allows us to obtain better values for AT, the conclusion is that the multiplication devices remain asymptotically suboptimal for the entire value range of the parameter n.

3.9 Combinational Array Structures for Binary Multiplication

As mentioned in the previous section, opposite to "one-bit-at-a-time" procedures are those where, in the entirely parallel mode, all the bits of multiplier X are simultaneously investigated, the operation being executed in only one CLOCK pulse. Thus, in this case, one of the implementation solutions uses the so-called combinational array structures [Haye98]. To arrive at the synthesis of such multipliers, let us analyse once more the how the product is formed.

Without loss of generality, and for brevity of expression, let us consider the two operands X and Y representing two integers without sign, of 4 bits, i.e.:

$$X = x_3x_2x_1x_0 = \sum_{i=0}^{3} x_i 2^i \quad \text{and} \quad Y = y_3y_2y_1y_0 = \sum_{j=0}^{3} y_j 2^j \tag{3.24}$$

On the one hand, product $P = XY$ is formed, if the weight of each bit is taken into account, by means of the following expansion:

$$\begin{aligned} P = XY &= \left(\sum_{i=0}^{3} x_i 2^i\right)\left(\sum_{j=0}^{3} y_j 2^j\right) = \sum_{i=0}^{3} 2^i \left(\sum_{j=0}^{3} x_i y_j 2^j\right) \\ &= 2^0\left(x_0y_0 2^0 + x_0y_1 2^1 + x_0y_2 2^2 + x_0y_3 2^3\right) \\ &\quad + 2^1\left(x_1y_0 2^0 + x_1y_1 2^1 + x_1y_2 2^2 + x_1y_3 2^3\right) \\ &\quad + 2^2\left(x_2y_0 2^0 + x_2y_1 2^1 + x_2y_2 2^2 + x_2y_3 2^3\right) \\ &\quad + 2^3\left(x_3y_0 2^0 + x_3y_1 2^1 + x_3y_2 2^2 + x_3y_3 2^3\right) \end{aligned} \tag{3.25}$$

If in the form (3.25) the terms are rearranged by grouping the one bit products of the same weight, we will reach:

$$\begin{aligned} P &= x_3y_3 2^6 + (x_2y_3 + x_3y_2)2^5 + (x_1y_3 + x_2y_2 + x_3y_1)2^4 \\ &\quad + (x_0y_3 + x_1y_2 + x_2y_1 + x_3y_0)2^3 + (x_0y_2 + x_1y_1 + x_2y_0)2^2 \\ &\quad + (x_0y_1 + x_1y_0)2^1 + x_0y_0 2^0 \end{aligned} \tag{3.26}$$

Relation (3.26) stands at the basis of multiplier synthesis as a combinational array structure. This structure is made up of two circuit matrixes, of which the first is meant to form the terms of $x_i y_j$ type. Since, at the level of a single bit, the arithmetic product coincides with the logic product, one of the matrixes consists of $n \cdot n$ AND logic circuits, according to the model from Fig. 3.41. The outputs of these AND gates are applied to a second matrix formed, this time, of full adder cells (FAC), which, adequately interconnected, are meant to configure several RCA adders whose inputs are the single bit products $x_i y_j$. Their shifted addition, required by the multiplication process and highlighted in relations (3.25) and (3.26) by the various weights of 2, is achieved through the spatial arrangement of FAC cells, as shown in Fig. 3.42. Mention should be made that, in the additions of the sum terms given by the parentheses from (3.26), carries may be generated, so that p_k bits (where $k = 0,7$) of the product are obtained by taking into account the potential carry bits from the previous ranks, as well [Haye98].

Obviously, in a similar way to the other diagrams, the pulse period which strobes the supply of the two operands, X and Y, to the combinational array structure has to cover the time interval required by the signals to pass the worst path, involving the largest number of logic circuits. But, in the diagram from Fig. 3.42, it can be

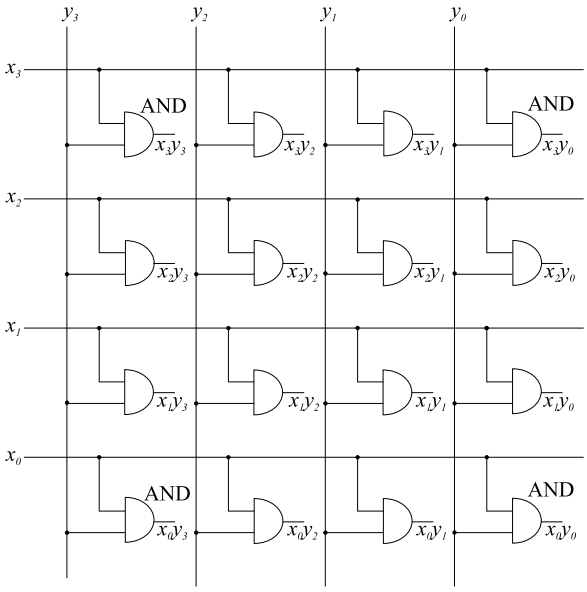


Fig. 3.41 AND logic gates matrix of a combinational array structure for 4-bit operands multiplication

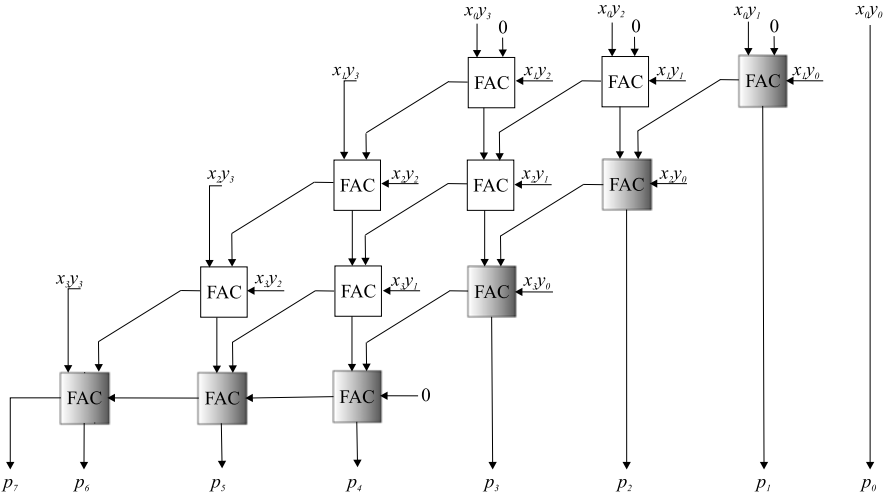


Fig. 3.42 FACs matrix of a combinational array structure for 4-bit operands multiplication

easily observed that the chaining (in an RCA) of the six shaded FACs is the longest path for the propagation of a possible carry, covering the worst case. If we denote by d the delay on a FAC and assume operands on n bits, for the matrix homologous to that from Fig. 3.42 the time T_{FAC}, corresponding to the above-mentioned circuits chain, results from the following relation:

$$T_{FAC} = (n-1)d + (n-1)d = 2(n-1)d \tag{3.27}$$

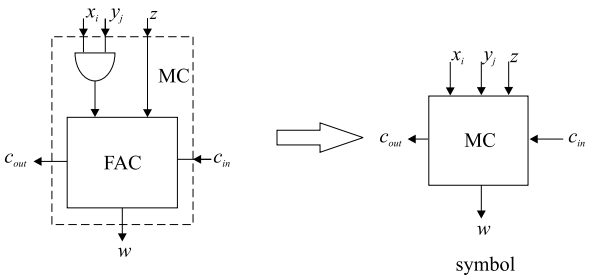


Fig. 3.43 Conceptual diagram and symbol for a multiplication cell

To this interval, there must also be added the delay, denoted by d', on an AND gate from the matrix represented in Fig. 3.41. Since all single bit products are formed in $T_{AND} = d'$, we have, for the entire structure, the time T_S given by the following:

$$T_S = T_{FAC} + T_{AND} = 2(n-1)d + d' \tag{3.28}$$

Since, if we suppose, for the sake of simplicity, that the delay d' is the same for all logic gates, whatever their type, then, if we assume $d \cong 2d'$, according to (3.28), we have $T_S = (4n-3)d'$ and, consequently, a time complexity T_C, as a function of the delay on an elementary logic circuit, of $T_C = 4n - 3$.

On the other hand, as regards the cost of the circuitry, and of the integration area, implicitly, the FACs matrix prevails, but its synthesis generally requires $n(n-1)$ FAC cells (Fig. 3.42). Consequently, the integration complexity, in terms of required circuitry or silicon area, can be estimated by $T = O(n^2)$.

Mention should also be made that the entire array structure is an ordered one, which favours the use of the VLSI integration technology. This aspect also results from the "flow" of the interconnections (refer to Fig. 3.41, but especially to Fig. 3.42) between the cells, but the integrated structure is not homogeneous, it being made up of two completely distinct arrays. This drawback of the VLSI integration can be surmounted by associating the AND circuitry from the first matrix (Fig. 3.41) to the FAC cells from the second matrix (Fig. 3.42). Thus there are formed so-called multiplier cells (MC), of which one is schematically presented in Fig. 3.43 together with the corresponding symbol. The inputs x_i and y_j are bit values of the two operands, X and Y, and z is an input usually connected to the output w of another MC [Haye98].

Using the new MC cells, let us configure the combinational array structure from Fig. 3.44 for the same multiplication operation whose product is given by the relation (3.26). It can be observed that the MCs from the first stage are used only for the AND gates which they include, forming the one bit products with x_0 as one of the factors. This "sacrifice" of circuitry is compensated through the more regular interconnections between the cells, which leads, globally, to a better management of the area available for the integration process. Then, the MCs contribute to the forming of the sums between the parentheses of the expression (3.26). It can be also observed that the MCs from the lower stage, which, essentially "collect" carries, have modified connections in comparison to those from other stages, namely because the

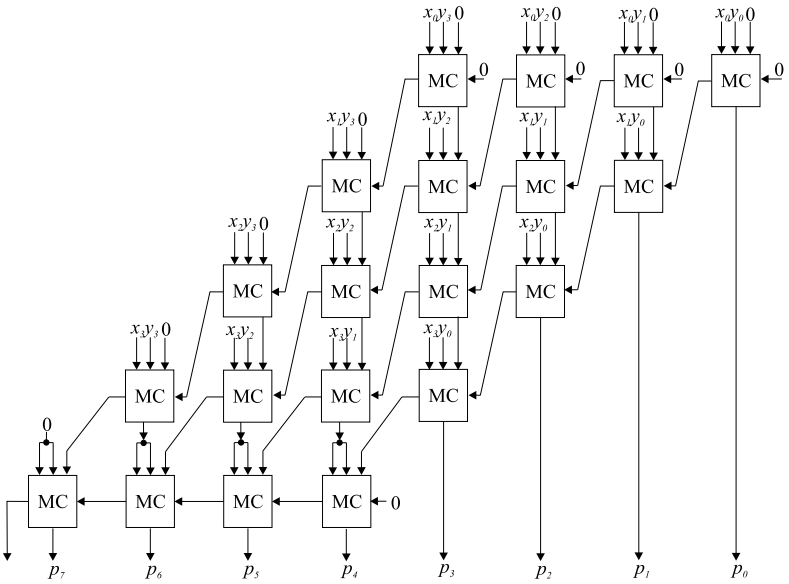


Fig. 3.44 Combinational array structure implemented with multiplication cells for 4-bit operands multiplication

operands bits are exhausted, the inputs x_i and y_j are both connected to the output w of a cell of the upper stage. Certainly, the interconnections from the physical layout of the structure might differ from those provided in Fig. 3.44, where the connections have been ordered to present a minimum number of crossings, which, as a matter of fact, is, generally, the primary requirement for the technological design of a VLSI layout. We can also observe the increase of the cells number to $n(n+1)$, but this "price rise" of the implementation is only an apparent one on account of the performance features of such a structure in VLSI technology [Vlăd82].

Following the presentation of the implementation of multiplication through combinational matricial structures, let us propose a more complex synthesis, namely, let us admit that the operands are integers with sign, and, moreover, let us suppose the procedures based on Booth recodings in radix 2 [Haye98]. Since it is known that both recodings apply both to addition and subtraction, as well as to no operation (NOP), the multiplier cell, as a fundamental element of the new structure, will differ from that used in Fig. 3.44. It shall be able to be configured, as applicable, so that, for the outputs w and c_{out} (we have adopted the notations used in Fig. 3.43), the Boolean equations given in the table from Fig. 3.45 will be achieved. At the output w of the new multiplier cell, which has been noted with MCB (the suffix letter B comes from Booth), the same logic function is executed for addition and subtraction, and in case of NOP it has to reproduce the input z representing the output of the MCB from the upper stage. Just as with the MC, y_j represents a bit of the multiplicand Y, but c_{in}, as well as c_{out}, represent the input carry variable in MCB, and the output carry variable from MCB, only for addition, because for subtraction

Operation	Output	
	w	c_{out}
Addition	$w=y_j$ *ex-or* z *ex-or* c_{in}	$c_{out}= y_jz$ *or* zc_{in} *or* $c_{in}y_j$
Subtraction	$w=y_j$ *ex-or* z *ex-or* c_{in}	$c_{out}= y_j\bar{z}$ *or* $\bar{z}c_{in}$ *or* $c_{in}y_j$
No operation	$w=z$	*irrelevant*

Fig. 3.45 Output functions of a multiplication cell of the combinational array structures implementing Booth's procedures

Operation	Control Variables	
	α	β
Addition	1	0
Subtraction	1	1
No operation	0	d

Fig. 3.46 Control variables encoding for a MCB

the carry variables have to be substituted by borrows, for which, for simplification reasons, we have taken over the same notations c_{in} and c_{out} from Fig. 3.43. Mention should be made that in case of NOP, c_{out} is irrelevant, because, anyway, $w = z$, c_{out} being obtained either by the relation corresponding to the addition or by the relation corresponding to the subtraction.

In order to distinguish between w corresponding to addition and substraction and w corresponding to NOP, we use a control variable that is denoted by α, and to distinguish between c_{out} corresponding to addition and c_{out} corresponding to substraction, we use a second control variable denoted by β. Thus, for α and β, we adopt the conventions from Fig. 3.46, where d comes from "don't care" and represents an indifferent logic value. With the variables already specified and aiming, through parallelization, to obtain the equations from Fig. 3.45, for w and c_{out}, the following Boolean expressions result without any difficulty:

$$\begin{aligned} w &= z \oplus \alpha(y_j \oplus c_{in}) \\ c_{out} &= (z \oplus \beta)(y_j \ \underline{or}\ c_{in}) \ \underline{or}\ y_j c_{in} \end{aligned} \tag{3.29}$$

Based on (3.29), for the MCB cell there results a possible implementation at the gate level given in Fig. 3.47, which contains the cell representation symbol, as well.

Let us consider, from now on, the same two operands X and Y, given by (3.24), by mentioning that, this time, the most significant bits x_3 and y_3 represent the signs of the two numbers. In this case we have a combinational array structure with the trapezoidal arrangement of MCBs from Fig. 3.48. At the upper stage there can be observed, first of all, the four MCBs to which the four bits of the multiplicand Y are passed. Up to the dimension of the final product, we have an extension of the sign, i.e. in the remaining three MCBs from the same upper stage y_3 is repeatedly supplied. This is because in the multiplication of two operands represented in C2, each operated partial product is a signed number, the extension of the last of which

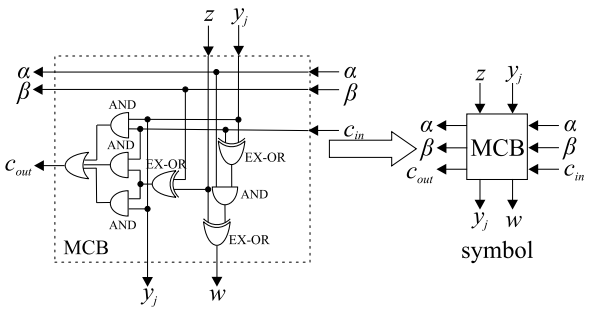


Fig. 3.47 Detailed diagram at the gate level and the symbol of a MCB

corresponds to the arithmetic shift. Consequently, the rhombus form structures from Fig. 3.42 and from Fig. 3.44 are reconfigured under trapezoid form for the analyzed case (Fig. 3.48). Mention should also be made that at each stage the rightmost MCBs from each stage have the input $c_{in} = 0$, and that all the MCBs belonging to a certain stage (line) i are crossed by control lines α_i and β_i, the value of index i increasing, for our example, from 0 to 3, downwards. The pair of Boolean functions (α_i, β_i) corresponding to stage i is generated by means of the combinational logic circuitry CL_i, to whose inputs the bits of multiplier X are supplied. The configuration of CL_i circuitry establishes through (α_i, β_i) together with the bits of the partial products (z), and of the multiplicand $Y(y_j)$, and the carries/borrows c_{in}, the logic values for the functions w and c_{out} supplied by each MCB cell. One special mention is made relative to the sign bit of the product (p_7 for the particular case of Fig. 3.48), which, according to the specifics of the Booth's procedure, is given by the extension of the most significant bit of the product (p_6 for the particular case of Fig. 3.48). In the same context, we will exclude from our considerations, based on a test realized, for example, through software routines, the extreme case corresponding to the multiplication of the smallest negative numbers, to which correspond codes associated with two's complement anomaly (for the particular case of Fig. 3.48, $X = Y = -8 = 1000_{C2}$, for which the product P results as $11000000_{C2} = -64$ instead of the correct value $01000000 = +64$).

Let us analyse the synthesis of CL_i circuitry for the particular case of our example. Thus, we elaborate, for each of the Booth recodings, a truth table, these tables being presented in Fig. 3.49, and in Fig. 3.50 respectively. The filling in of the tables begins by obtaining, from the set of input variables, the Booth recoded forms by applying the known rules (refer to the examples from Fig. 3.14 and Fig. 3.21 respectively). Then, for each variable x_{iB} and x_{iMB} the corresponding columns α_i and β_i, and α_i' and β_i' respectively will be filled in, where $i = 0,3$. All this is based on conventions that are in accordance with those specified in Fig. 3.46 with the mention that we considered for the variable β_i the don't care value d to be equal to 0.

Since they are the same for both tables, and if we refer only to Fig. 3.49, we have:

1. If $x_{iB} = 0$, then $a_i = b_i = 0$.
2. If $x_{iB} = 1$, then $a_i = 1$ and $b_i = 0$.
3. If $x_{iB} = \overline{1}$, then $a_i = 1$ and $b_i = 1$.

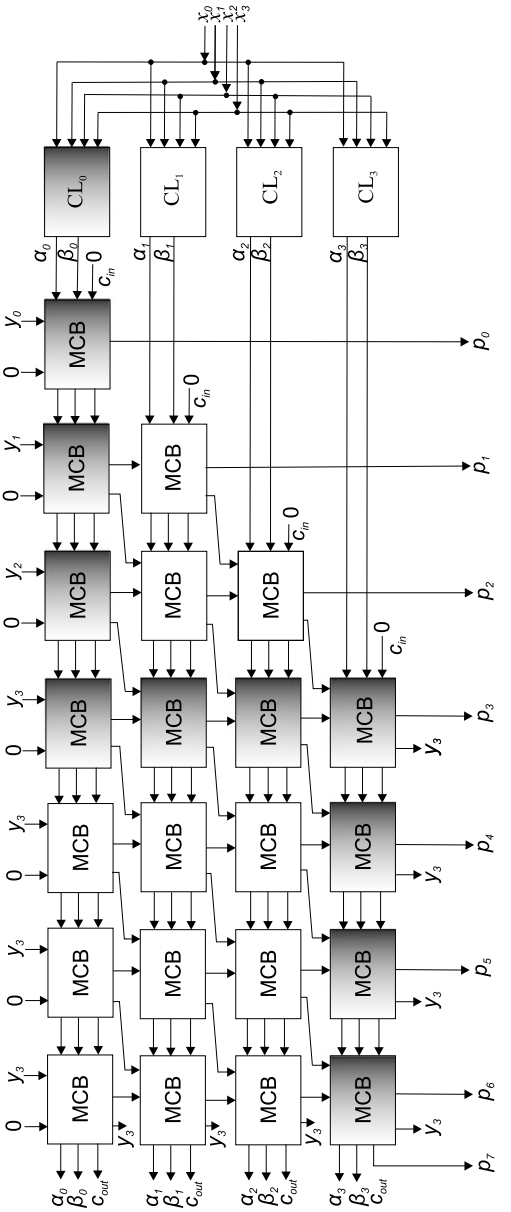


Fig. 3.48 Combinational array structure with MCBs for 4-bit operands multiplication

Input variables				Booth recoding				Control lines variables							
x_3	x_2	x_1	x_0	x_{3B}	x_{2B}	x_{1B}	x_{0B}	α_3	β_3	α_2	β_2	α_1	β_1	α_0	β_0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	1	$\overline{1}$	0	0	0	0	1	0	1	1
0	0	1	0	0	1	$\overline{1}$	0	0	0	1	0	1	1	0	0
0	0	1	1	0	1	0	$\overline{1}$	0	0	1	0	0	0	1	1
0	1	0	0	1	$\overline{1}$	0	0	1	0	1	1	0	0	0	0
0	1	0	1	1	$\overline{1}$	1	$\overline{1}$	1	0	1	1	1	0	1	1
0	1	1	0	1	0	$\overline{1}$	0	1	0	0	0	1	1	0	0
0	1	1	1	1	0	0	$\overline{1}$	1	0	0	0	0	0	1	1
1	0	0	0	$\overline{1}$	0	0	0	1	1	0	0	0	0	0	0
1	0	0	1	$\overline{1}$	0	1	$\overline{1}$	1	1	0	0	1	0	1	1
1	0	1	0	$\overline{1}$	1	$\overline{1}$	0	1	1	1	0	1	1	0	0
1	0	1	1	$\overline{1}$	1	0	$\overline{1}$	1	1	1	0	0	0	1	1
1	1	0	0	0	$\overline{1}$	0	0	0	0	1	1	0	0	0	0
1	1	0	1	0	$\overline{1}$	1	$\overline{1}$	0	0	1	1	1	0	1	1
1	1	1	0	0	0	$\overline{1}$	0	0	0	0	0	1	1	0	0
1	1	1	1	0	0	0	$\overline{1}$	0	0	0	0	0	0	1	1

Fig. 3.49 Truth table based on Booth's recoding for the synthesis of the logic circuits generating the control variables α_i and β_i

Following the elaboration of the control lines values columns, we can elaborate, for each function separately, the Boolean equation that will stand at the basis of the synthesis of the corresponding CL_i circuitry. Consequently, without loss of generality, we shall appeal to Karnaugh maps to obtain the minimized forms for each function. Thus, in Fig. 3.51 the Karnaugh maps for α_3 and the binary units covering mode are presented. From Fig. 3.51, the following Boolean expression results, by means of which CL_3 is partly synthesized:

$$\alpha_3 = x_3\overline{x_2} \ \underline{or} \ \overline{x_3}x_2 = x_3 \oplus x_2 \tag{3.30}$$

Input variables				Modified Booth recoding				Control lines variables							
x_3	x_2	x_1	x_0	x_{3MB}	x_{2MB}	x_{1MB}	x_{0MB}	α'_3	β'_3	α'_2	β'_2	α'_1	β'_1	α'_0	β'_0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1	0	0	0	0	0	0	1	0
0	0	1	0	0	0	1	0	0	0	0	0	1	0	0	0
0	0	1	1	0	1	0	$\overline{1}$	0	0	1	0	0	0	1	1
0	1	0	0	0	1	0	0	0	0	1	0	0	0	0	0
0	1	0	1	0	1	0	1	0	0	1	0	0	0	1	0
0	1	1	0	1	0	$\overline{1}$	0	1	0	0	0	1	1	0	0
0	1	1	1	1	0	0	$\overline{1}$	1	0	0	0	0	0	1	1
1	0	0	0	$\overline{1}$	0	0	0	1	1	0	0	0	0	0	0
1	0	0	1	$\overline{1}$	0	0	1	1	1	0	0	0	0	1	0
1	0	1	0	$\overline{1}$	0	1	0	1	1	0	0	1	0	0	0
1	0	1	1	0	$\overline{1}$	0	$\overline{1}$	0	0	1	1	0	0	1	1
1	1	0	0	0	$\overline{1}$	0	0	0	0	1	1	0	0	0	0
1	1	0	1	0	$\overline{1}$	0	1	0	0	1	1	0	0	1	0
1	1	1	0	0	0	$\overline{1}$	0	0	0	0	0	1	1	0	0
1	1	1	1	0	0	0	$\overline{1}$	0	0	0	0	0	0	1	1

Fig. 3.50 Truth table based on canonical Booth's recoding for the synthesis of the logic circuits generating the control variables α'_i and β'_i

In a similar way, for the other control functions, as well, we obtain the following:

$$\begin{cases} \beta_3 = x_3\overline{x_2} \\ \alpha_2 = x_2 \oplus x_1 \\ \beta_2 = x_2\overline{x_1} \\ \alpha_1 = x_1 \oplus x_0 \\ \beta_1 = x_1\overline{x_0} \\ \alpha_0 = x_0 = x_0 \oplus 0 \\ \beta_0 = x_0 = x_0 \cdot \overline{0} \end{cases} \tag{3.31}$$

We should like to make a secondary observation regarding the last two equations from (3.31), where neutral element 0 intervenes for modularity purposes (all

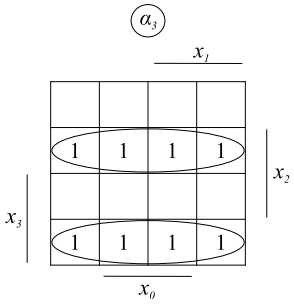


Fig. 3.51 Minimization of the logical equation corresponding to the control variable α_3

α_i variables are identically synthesized, but with different input variables, the same observation applying to β_i variables, as well), namely that this favors the implementation technology. If for the variable β_i we make use of the don't care value d in accordance to those presented in Fig. 3.46, then the logical equations for β_i, obtained based on a table from Fig. 3.49 correspondingly modified, can be further simplified, resulting $\beta_3 = x_3$, $\beta_2 = x_2(= \overline{x_1})$, $\beta_1 = x_1$ and $\beta_0 = 1$.

On the other hand, regarding the modified Booth recoding, if a similar operating procedure is adopted, the following control functions will be obtained:

$$\begin{cases} \alpha'_3 = x_3\overline{x_2}\,\overline{x_1} \ \underline{or}\ x_3\overline{x_2}\,\overline{x_0} \ \underline{or}\ \overline{x_3}x_2x_1 \\ \alpha'_2 = x_2\overline{x_1} \ \underline{or}\ \overline{x_2}x_1x_0 \\ \alpha'_1 = x_1\,\overline{x_0} \\ \alpha'_0 = x_0 \end{cases} \qquad \begin{cases} \beta'_3 = x_3\overline{x_2}\,\overline{x_1} \ \underline{or}\ x_3\overline{x_2}\,\overline{x_0} \\ \beta'_2 = x_3x_2\overline{x_1} \ \underline{or}\ x_3\overline{x_2}x_1x_0 \\ \beta'_1 = x_2x_1\overline{x_0} \\ \beta'_0 = x_1x_0 \end{cases} \tag{3.32}$$

Consequently, the synthesis given by (3.30) and (3.31) is more homogeneous than that given by (3.32), but we consider that the difference is not decisive. Moreover, for the relations given in (3.32), if for β'_i we make use of the don't care value d in accordance to those presented in Fig. 3.46, then the logical equations for β'_i, obtained based on a table from Fig. 3.50 correspondingly modified, can be further simplified, resulting $\beta'_3 = \beta'_2 = x_3$, $\beta'_1 = x_2$ and $\beta'_0 = x_1$.

Finally, as in the case of the other multiplication devices, we shall refer to the performance and cost of the structure of the type given in Fig. 3.48. First of all, we should like to point out that, similar to the multiplier from Fig. 3.44, the new combinational array structure is regular, the circuits' interconnection between the MCBs being ordered, which results in a layout without major problems, that favors the VLSI integration. On the other hand, as regards the time factor, the longest path that has to be run through by the signals is given by the shaded MCBs chaining with CL_0 (Fig. 3.48). If we suppose, again, the same delay d' on a logic gate, whatever its type (which represents a rather rough approximation, it being known, for instance, that EX-OR gates are slower as compared to others [Yarb97]), then on an MCB we have the delay $d = 3d'$ (the worst case, refer to Fig. 3.47). Thus, for the entire structure from Fig. 3.48, there results $T = 10d + d' = 31d'$, considering that CL_0 has only one logic level, according to (3.31) or (3.32). However, if we suppose, just on the line, that the delay on CL_0 is the same as the delay on an MCB, then, in terms

of d, for the general case, we have $T = (n + (n-2) + n + 1)d$, or a time complexity of $T_C = 3n - 1$.

We shall remain within the sphere of multipliers for signed integers, represented in C2, and which can be implemented with combinational array structures, but, this time, we shall present, at the level of principles the Baugh-Wooley method which is favorable for practical execution under certain circumstances [Parh00]. To investigate this method, let us consider again the operands given by (3.24), where x_3 and y_3 are sign bits, and let us use (3.5) to interpret the negative numbers. Under these circumstances, the product given by (3.26) becomes:

$$\begin{aligned} P = {} & x_3y_3 2^6 + (-x_2y_3 - x_3y_2)2^5 + (-x_1y_3 + x_2y_2 - x_3y_1)2^4 \\ & + (-x_0y_3 + x_1y_2 + x_2y_1 - x_3y_0)2^3 + (x_0y_2 + x_1y_1 + x_2y_0)2^2 \\ & + (x_0y_1 + x_1y_0)2^1 + x_0y_0 2^0 \end{aligned} \tag{3.33}$$

If (3.33) is used, certain weighted terms will become negative, and the Baugh-Wooley method aims to avoid them. Consequently, the one bit negative products from (3.33) are subject to some simple transformations, such as the ones made, for instance, for $(-x_0y_3)$:

$$-x_0y_3 = (1 - x_0)y_3 - y_3 = \overline{x_0}y_3 - y_3 \tag{3.34}$$

Applying transformations of the type from (3.34) to all the negative terms from (3.33), the following form will be obtained:

$$\begin{aligned} P = {} & x_3y_3 2^6 + (\overline{x_2}y_3 - y_3 + x_3\overline{y_2} - x_3)2^5 + (\overline{x_1}y_3 - y_3 + x_2y_2 \\ & + x_3\overline{y_1} - x_3)2^4 + (\overline{x_0}y_3 - y_3 + x_1y_2 + x_2y_1 + x_3\overline{y_0} - x_3)2^3 \\ & + (x_0y_2 + x_1y_1 + x_2y_0)2^2 + (x_0y_1 + x_1y_0)2^1 + x_0y_0 2^0 \end{aligned} \tag{3.35}$$

We modify the parenthesis with weight 2^3 by adding and subtracting the value $(x_3 + y_3)$, so that:

$$\begin{aligned} & \overline{x_0}y_3 - y_3 + x_1y_2 + x_2y_1 + x_3\overline{y_0} - x_3 \\ & = \overline{x_0}y_3 - y_3 + x_1y_2 + x_2y_1 + x_3\overline{y_0} - x_3 + (x_3 + y_3) - (x_3 + y_3) \\ & = \overline{x_0}y_3 + x_1y_2 + x_2y_1 + x_3\overline{y_0} + x_3 + y_3 - 2(x_3 + y_3) \end{aligned} \tag{3.36}$$

The addition of the first six terms from (3.36) gives a nonnegative value, and, due to the multiplication by 2, the value $(-(x_3 + y_3))$ is "pushed" into the parenthesis weighted with 2^4, which becomes:

$$\begin{aligned} & \overline{x_1}y_3 - y_3 + x_2y_2 + x_3\overline{y_1} - x_3 - (x_3 + y_3) \\ & = \overline{x_1}y_3 + x_2y_2 + x_3\overline{y_1} - 2(x_3 + y_3) \end{aligned} \tag{3.37}$$

Analyzing (3.37), it can be observed that, in a similar way to (3.36), the first three added terms give a nonnegative value, and the value $(-(x_3 + y_3))$ is "pushed" into the parenthesis weighted with 2^5, wherefrom it arrives at the last term, weighted with 2^6, which undergoes the following transformation:

$$\begin{aligned} x_3y_3 - (x_3 + y_3) &= x_3y_3 - x_3 - y_3 + 1 + 1 - 2 \\ &= x_3y_3 + (1 - x_3) + (1 - y_3) - 2 = x_3y_3 + \overline{x_3} + \overline{y_3} - 2 \end{aligned} \quad (3.38)$$

The value (-2) from (3.38) can be substituted by (-1) in the position with the weight 2^7, which can be substituted, in its turn, by 1 and a borrow from the next rank, it being non-existent. Under these circumstances, from (3.35) we have arrived at the form of the Baugh-Wooley product, given by the following expression:

$$\begin{aligned} P = {} & 2^7 + (x_3y_3 + \overline{x_3} + \overline{y_3})2^6 + (\overline{x_2}y_3 + x_3\overline{y_2})2^5 + (\overline{x_1}y_3 + x_2y_2 + x_3\overline{y_1})2^4 \\ & + (\overline{x_0}y_3 + x_1y_2 + x_2y_1 + x_3\overline{y_0} + x_3 + y_3)2^3 + (x_0y_2 + x_1y_1 + x_2y_0)2^2 \\ & + (x_0y_1 + x_1y_0)2^1 + x_0y_02^0 \end{aligned} \quad (3.39)$$

Similar to Booth procedures, besides the original Baugh-Wooley procedure, there is a modified Baugh-Wooley procedure. It is based on a new form of the product P, performing certain transformations over relation (3.39). Thus, referring again to the parenthesis with the weight 2^3 provided from (3.39), we execute the following easily visible modifications:

$$\begin{aligned} & \overline{x_0}y_3 + x_1y_2 + x_2y_1 + x_3\overline{y_0} + x_3 + y_3 \\ & \quad = (1 - x_0)y_3 + x_1y_2 + x_2y_1 + x_3(1 - y_0) + x_3 + y_3 \\ & \quad = -x_0y_3 + x_1y_2 + x_2y_1 - x_3y_0 + 2(x_3 + y_3) + 1 + 1 - 2 \\ & \quad = (1 - x_0y_3) + x_1y_2 + x_2y_1 + (1 - x_3y_0) + 2(x_3 + y_3 - 1) \\ & \quad = \overline{x_0y_3} + x_1y_2 + x_2y_1 + \overline{x_3y_0} + 2(x_3 + y_3 - 1) \end{aligned} \quad (3.40)$$

The first four added terms from (3.40) lead to a non-negative value, and the parenthesis $(x_3 + y_3 - 1)$ is "pushed" into that with the weight 2^4, which undergoes some similar processing, so that the following will be obtained:

$$\begin{aligned} & \overline{x_1}y_3 + x_2y_2 + x_3\overline{y_1} + x_3 + y_3 - 1 \\ & \quad = (1 - x_1)y_3 + x_2y_2 + x_3(1 - y_1) + x_3 + y_3 - 1 \\ & \quad = -x_1y_3 + x_2y_2 - x_3y_1 + 2(x_3 + y_3) - 1 + 1 + 1 - 2 \\ & \quad = (1 - x_1y_3) + x_2y_2 + (1 - x_3y_1) + 2(x_3 + y_3) + 1 - 2^2 \\ & \quad = \overline{x_1y_3} + x_2y_2 + \overline{x_3y_1} + 1 + 2(x_3 + y_3 - 2) \end{aligned} \quad (3.41)$$

Analysing (3.41), parenthesis $(x_3 + y_3 - 2)$ can be "pushed" into the parenthesis with the weight 2^5, which, following a similar treatment as above, becomes

$(\overline{x_2 y_3} + \overline{x_3 y_2})$ and the same value $(x_3 + y_3 - 2)$ is "pushed" into the parenthesis with the weight 2^6. Operating at the level of this parenthesis, there results:

$$x_3 y_3 + \overline{x_3} + \overline{y_3} + x_3 + y_3 - 2 = x_3 y_3 + 1 + 1 - 2 = x_3 y_3 \tag{3.42}$$

Synthesizing these transformations, from (3.39) we achieve the form of the Baugh-Wooley modified product, given by the following expression:

$$\begin{aligned} P = {} & 2^7 + x_3 y_3 2^6 + (\overline{x_2 y_3} + \overline{x_3 y_2}) 2^5 + (\overline{x_1 y_3} + x_2 y_2 \\ & + \overline{x_3 y_1} + 1) 2^4 + (\overline{x_0 y_3} + x_1 y_2 + x_2 y_1 + \overline{x_3 y_0}) 2^3 \\ & + (x_0 y_2 + x_1 y_1 + x_2 y_0) 2^2 + (x_0 y_1 + x_1 y_0) 2^1 + x_0 y_0 2^0 \end{aligned} \tag{3.43}$$

Both Baugh-Wooley forms, expressed through (3.39) and (3.43), allow the implementation of combinational array structures according to the model run through for the solutions from Fig. 3.41 and Fig. 3.42, and from Fig. 3.44 respectively, mentioning, obviously, that the numbers to be multiplied are now signed numbers. Comparing the two forms and analysing within them the parentheses with the largest number of added terms, it can be seen that we have, on the one hand, the value 6 in (3.39) (corresponding to the parenthesis with the weight 2^3) and, on the other hand, the value 4 in (3.43) (corresponding to the parentheses with the weights 2^3 and 2^4). The above-mentioned numbers give the measure of the "height" of the array structure, and, implicitly, of the number of levels that have to be run through, and, consequently, of the delay, and finally, of the performance. The modified Baugh-Wooley form appears more favorable under this aspect, saving two terms at the "critical" parenthesis, even if it increases from three to four the number of terms from the parenthesis with the weight 2^4, because the maximum value 4 is not exceeded.

Mention should also be made that, as far as implementation is concerned, the Baugh-Wooley procedures allow simpler structures than those based on negative weighted terms, such as the terms from relation (3.33). The Baugh-Wooley forms can also be used for the synthesis of parallel multiplication devices that are implemented not by means of FAC cells chained in RCA manner, but by means of CSA trees, as presented in the next section.

Finally, we shall once more refer to the solutions of multipliers implemented through combinational array structures, namely, we shall refer, without loss of generality, to a structure of the type from Fig. 3.42. More precisely, for the same example of numbers from (3.24), the design from Fig. 3.42 can be represented, at block diagram level, as a CSA adders chain, as can be observed in Fig. 3.52. Mentioning that the three pairs of carry-sum binary vectors $(C^{(i)}, S^{(i)}$, where $i = 1, 2, 3)$ are passed from stage to stage as presented in Fig. 3.42, let us specify that the last level, at whose output product P appears, is made up of a carry propagate adder (CPA). Even if, as regards the latency of the multiplication process, the solutions from Fig. 3.52 and Fig. 3.37 (that with only one CSA adder) do not differ essentially, there being executed, as a principle, the same number of additions, the version with more CSA adders is more favorable because the operation can be pipelined, which

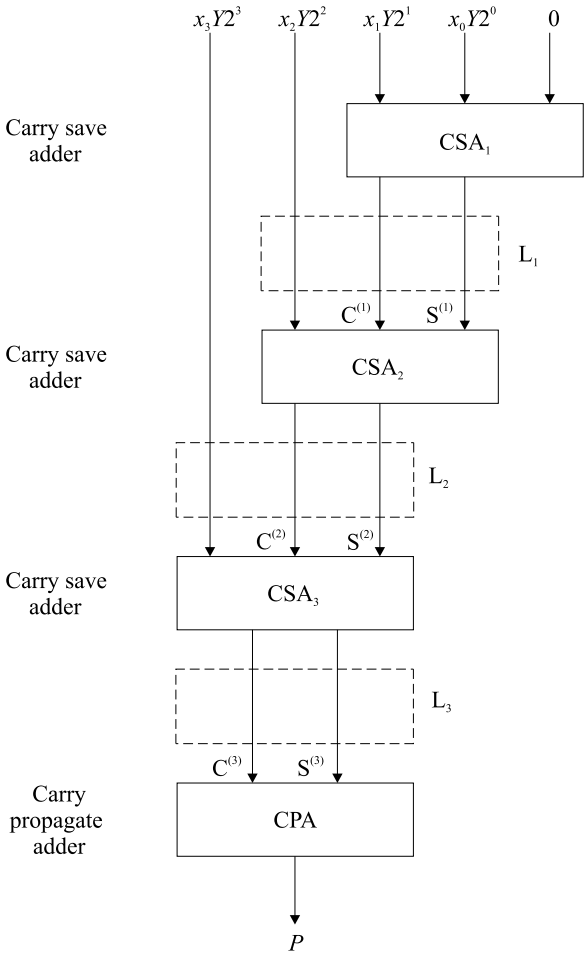


Fig. 3.52 Conceptual diagram of a combinational array multiplier implemented through CSAs chaining

increases of the throughput, and thus this version becomes attractive for application in array processors [HuEr05].

Thus, between the CSA levels there are interposed layers of latches L_i, where $i = 1, 2, 3$. Following the storage of the $C^{(i)}$ and $S^{(i)}$ vectors in the corresponding latches, L_i, the vectors from the CSA_i inputs can be modified. Thus, it is possible to execute, in overlapped manner, more operations that are in various phases of their execution. Similar to the overlapped execution of the instructions by a pipelined control [HePa03], the arithmetic operations, particularly multiplication, can be executed by a so-called arithmetic pipeline [Poll90, Kuli02]. Following its filling up, the number of operations that are simultaneously being executed at a certain moment represents the parameter called the pipeline depth, it being a measure of the "speedup" brought about by the approach. In fact, the operations are not executed more rapidly, but more can be executed, e.g. multiplications, within a preestablished time interval in the above-described overlapped manner.

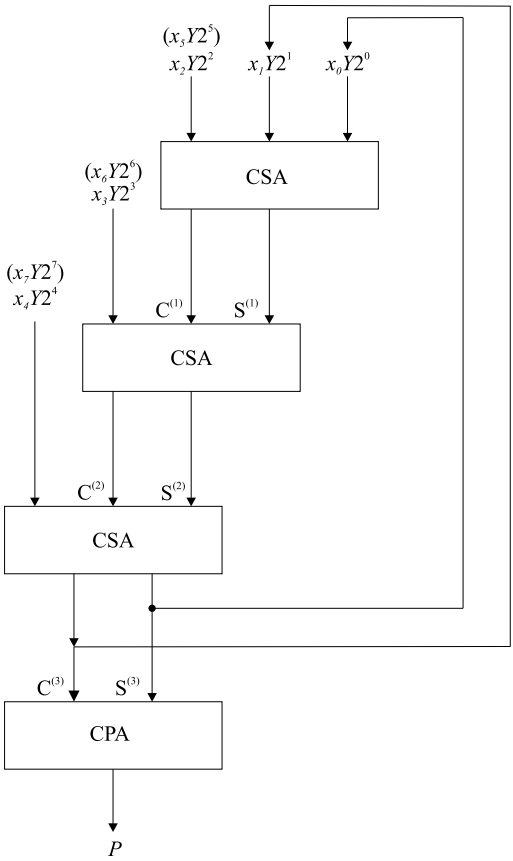


Fig. 3.53 Conceptual diagram of a combinational array multiplier implemented through CSAs chaining and two passes application of the operands

Indisputably, a design of the type given in Fig. 3.52 usually requires a large area of integration. When this last factor is represents a critical one, a solution may be the CSA adders chain which allows the execution of multiplication by appealing to two passes through the structure presented schematically in Fig. 3.53 [HePa03]. We suppose that, unlike the configuration from Fig. 3.52, which enables the multiplication of some numbers on four bits, this time the operands' dimension is 8 bits. At the first pass through the chain of the three CSAs there are added, in CSA manner, five one bit products, namely x_0Y2^0 to x_4Y2^4, obtaining the binary vectors $C^{(3)}$ and $S^{(3)}$ which replace x_1Y2^1 and x_0Y2^0, while simultaneously the other one bit products, $x_5Y2^5$to x_7Y2^7, substitute the previous, x_2Y2^2 to x_4Y2^4. Following the signal propagation at the second pass, new values result for the vectors $C^{(3)}$ and $S^{(3)}$, which are added in the conventional manner by the CPA, with carry propagation. This technical solution, which is very attractive as regards the saving of the silicon substrate area, comes up against certain difficulties. Thus, the time interval required for the signals passing the array structure, made up of CSAs, must be strictly monitored to allow the substitution of one bit products for the second pass at the proper moment. Then, mention should be made that the number of ranks is larger than in a structure

of the type presented in Fig. 3.52 (the reference is made at the multiplication of unsigned numbers). But the greatest disadvantage of the solution from Fig. 3.53 is that it does not allow the operation to be pipelined, the intermediate storage in of partial results in latches not being possible.

Consequently, the combinational array structures made up of CSA adder chains sometimes called "one-sided CSA trees" [Parh00], lead to solutions which are generally slower but more regular, and, implicitly, with a reduced chip area, as compared to the genuine tree structures which will be presented below. However, sometimes the choice may favour the above-presented configurations due to the operation's large throughput when they can be executed in arithmetic pipeline manner.

3.10 Combinational Tree Structures for Binary Multiplication

If the operation latency of the previous array structures is quasiproportional to the dimension n of the operands, a substantial reduction of the multiplication time, as well as bringing it to an approximate dependence on $\lceil \log_2 n \rceil$, is possible by appealing to tree structures of CSA adders. Obviously, the new configurations are much more expensive, but they are justified by those applications where the speed of the operation's execution is the critical factor.

A first solution which uses combinational tree structures is given in Fig. 3.54 [Parh00]. The idea which stands at the basis of this construction consists in the parallel execution of CSA additions by reducing, in comparison to the solution from Fig. 3.52, the paths to be run through by certain binary flows. Moreover, the synthesis may be done using modules of the type presented in Fig. 3.54 in dotted frames, which, as can be observed, have four input vectors and two output vectors, including two CSA levels. This configuration, as regards its implementation with the so-called 4-to-2 reduction modules, becomes a binary tree of the type presented in Fig. 3.55. This structure is characterized by the regularity of the interconnections, which leads to a more efficient layout [Parh00], compensating for the disadvantage that might result from the greater "height" of the diagram. Obviously, for a large n, a structure of the type presented in Fig. 3.54, and Fig. 3.55 respectively, may have a larger number of CSA levels, as compared to other tree structures, such as Wallace or Dadda trees (e.g., for $n = 128$, the binary trees solution of 4-to-2 reduction modules presents $6 \cdot 2 = 12$ CSA levels, while a Wallace tree, as will be seen below, allows the saving of one such level).

Aiming to achieve a structure that can be integrated in VLSI technology, as easily as possible, with as simple a layout as possible, as discussed above, we present below another multiplier version, which presents a binary tree configuration [TaYY85]. As regards this structure, the addition operation implies certain problems because it has to be executed by special adder cells of $(2, 1)$ type, these having at each input a digit and producing at output only one sum digit. They differ from the "classical" FACs, which are of $(3, 2)$ type, presenting, in addition, an input and an output that are both of them dedicated to the carry. In order to use the adder cells

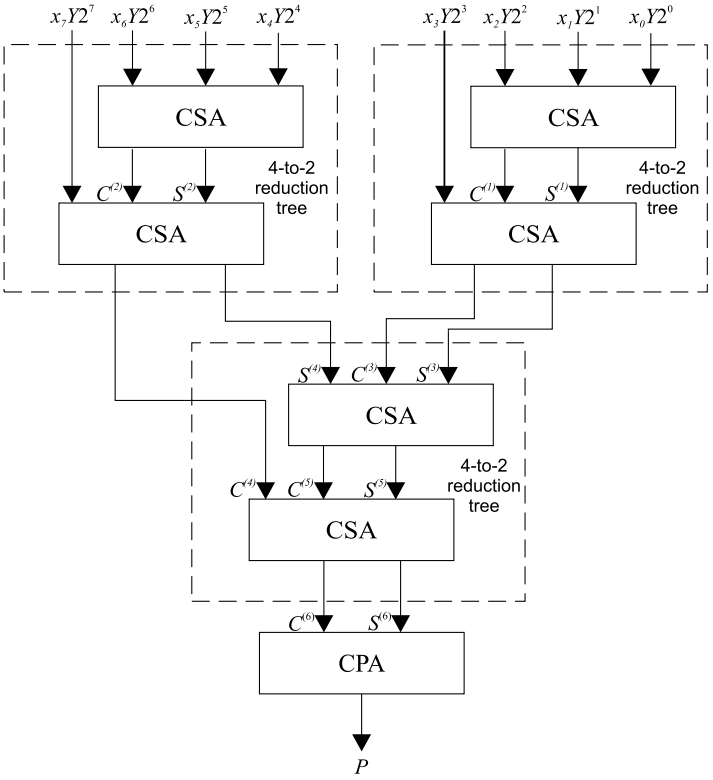


Fig. 3.54 Combinational tree structure for binary multiplication with CSAs grouped in 4-to-2 reduction modules

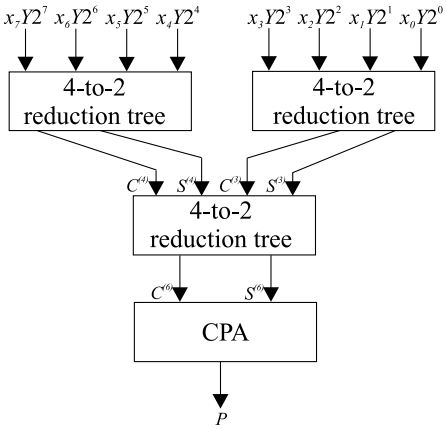


Fig. 3.55 Block diagram for a combinational tree structure implemented with 4-to-2 reduction modules

(2, 1), where the operands' binary coding is of no benefit, we shall resort to a different coding, i.e. to the signed-digit representation, which is used for Booth recodings, and which, as it is known, is characterized by tolerating the digits 0, 1 and $\overline{1}$.

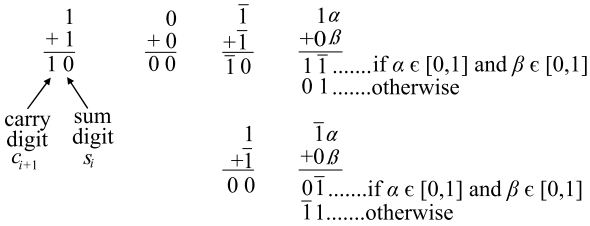


Fig. 3.56 Addition conventions upon which is based the synthesis of the binary-signed digit adders

As concerns the addition algorithm, its elaboration takes advantage of the signed-digit redundancy, there being generated several rules to execute the required carry propagation inhibition. The operation proper is executed in two stages, the first being dedicated to the addition of a pair of digits, each one belonging to one operand. Consequently, the conventions from Fig. 3.56 [HePa03] are employed, obtaining, for each pair, a carry digit c_{i+1} and a sum digit s_i. In the second stage of the addition operation the digits s_i and c_i are added, for each rank, when, due to the previously applied rules (Fig. 3.56) no carry is generated. Some of these rules are obvious (such as $1 + 1 = 10$ or $\overline{1} + \overline{1} = \overline{1}0$), and for the pairs $(1 + 0)$ and $(\overline{1} + 0)$ the values of the digits concatenated to the right and denoted generically with α and β have been taken into account. When α and β take the value 0 or 1, and neither of them is $\overline{1}$, $1 + 0 = 1\overline{1}$, while $\overline{1} + 0 = 0\overline{1}$, and for all the cases when one of the variables α or β, or both of them, take the value $\overline{1}$, we have $1 + 0 = 01$, and $\overline{1} + 0 = \overline{1}1$. These constraining conventions enable us to easily check that the carry inhibition requirement is fulfilled in the addition $(s_i + c_i)$ [HePa03]. Mention should also be made that in order to obtain a sum digit (2, 1), it is necessary to simultaneously investigate three bits of the operands, i.e. the current bit and two bits that precede it (running through the operands from right to left). After specifying this, it is possible to achieve the synthesis of some binary-signed digit (BSD) adders, which will substitute the above-mentioned CSAs and enable the configuration of the binary tree [Parh00].

The specific operating mode of such a structure is presented in Fig. 3.57, where, for simplicity reasons, two unsigned operands have been adopted. One can observe the parallel computation of the sums $(x_1Y2^1 + x_0Y2^0)$ to $(x_7Y2^7 + x_6Y2^6)$, by the BSD adders from the first level, with the corresponding shift of one bit products. Then, also by pairs and with the corresponding shift, there are added the results of the additions, denoted by Σ_i. The right side of Fig. 3.57 presents, schematically, the entire configuration of a binary tree. It can be seen that the last sum (in our case, Σ_7) is passed to a BSD-to-binary converter, which consists, essentially, of a fast conventional subtracter that subtracts the negative component (Σ_{7n}, Fig. 3.57) of the BSD form from the positive one (Σ_{7p}, Fig. 3.57) to obtain the binary form of product P.

Certainly, one of the decisive factors in choosing a certain multiplier structure is, and now we repeat what has already been mentioned above, the VLSI implementation technology. Consequently, the favoured ones are the iterative or recursive configurations of binary tree type, which enable the efficient achievement of computer-

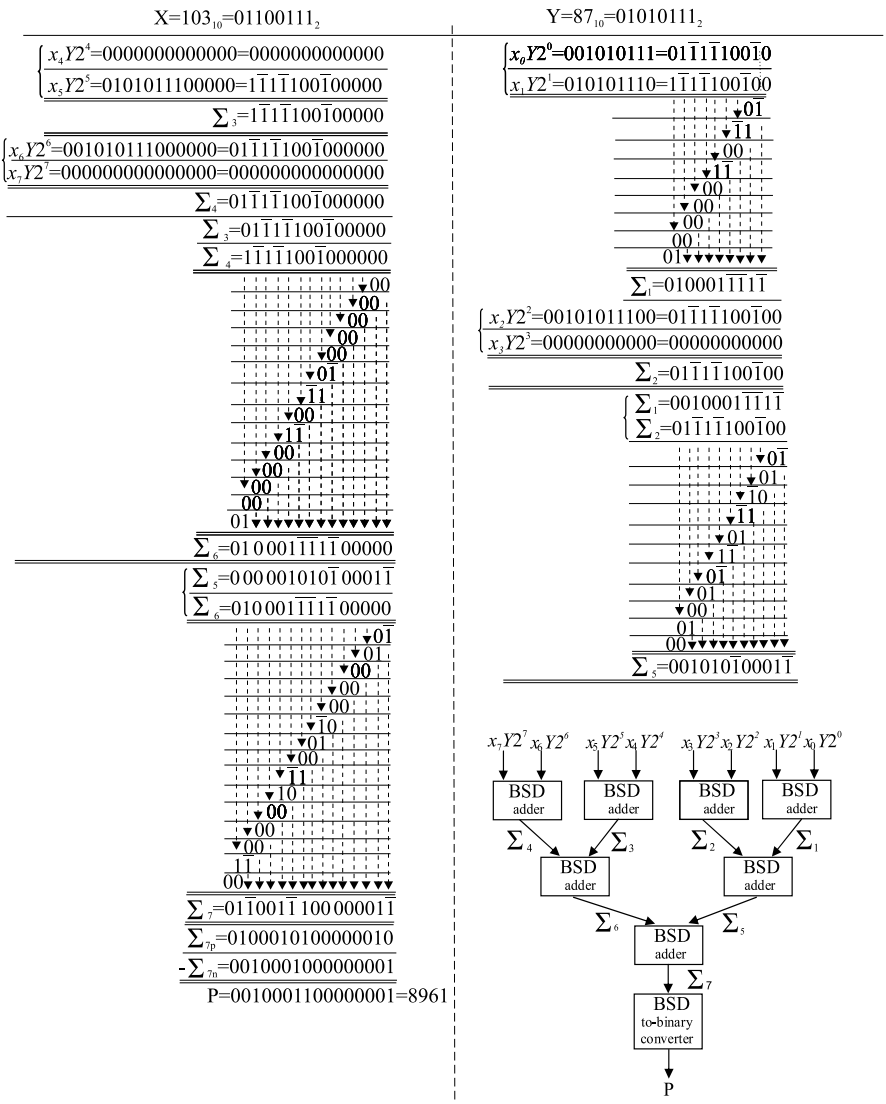


Fig. 3.57 Binary multiplication example using a binary tree of BSD adders and the block diagram corresponding to the multiplier

aided syntheses. Due to their regularity, the connections and the signal propagation paths do not significantly vary in length, which reduces the probability of the occurrence of logic hazards or shifted receipt of the same signal (the so-called signal skew), with favorable implications as regards both performance, and power consumption [RaPe96]. However, there are applications where the above-mentioned aspects are left in the background, and as great a reduction as possible has the pri-

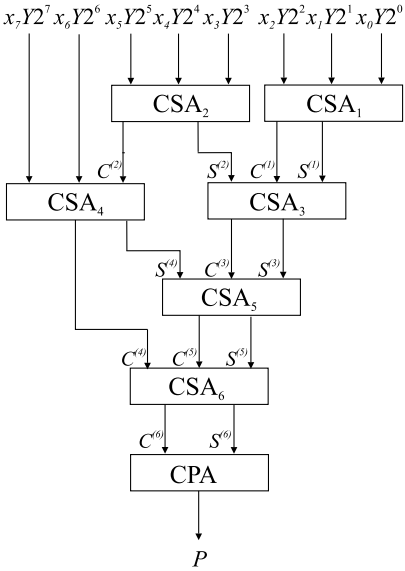


Fig. 3.58 Combinational tree structure with CSAs in a Wallace manner interconnection

ority, tending towards a time performance logarithmic in the "height" of the tree structure. It is another way of forcing a better performance, even sacrificing the regularity of the circuitry, with a unfavorable influence on obtaining a simpler and more efficient layout. Such a solution is represented by the configuration in the so-called Wallace tree of CSA adders [Kuli02, Parh00, PaHe96]. For our particular case of numbers on 8 bits, with the notations introduced by us and the above-mentioned observations, a possible structure of a Wallace tree is given in Fig. 3.58. The final sum ($S^{(6)}$) and carry ($C^{(6)}$) vectors are added by means of a fast conventional adder with carry propagation (CPA). What is of interest in the construction of such a tree structure, is the smallest number of CSA levels for a certain value corresponding to the dimension n of the operands, or, otherwise, the minimum height m of the tree for the number, also equal to n, of input vectors. This number is reduced at every CSA level by 3/2, so that the recurrent decrease from n, CSA level by CSA level of the input and output numbers can be followed by consulting the table from Fig. 3.59 [Parh00]. We have denoted by o_i the number of outputs in level i, which are inputs in the next level, and by r_i—where r_i belongs to the integer interval [0, 1, 2]—the residues of n and then o_i divisions by 3. As regards these divisions, we take into account only the integer value of the quotient, the rest being ignored, a fact marked by the bars $\lfloor\ \rfloor$. At the basis of such a tree structure (level m) there is always one CSA adder, obviously with 3 inputs and 2 outputs. If Fig. 3.59 is taken into account, it results without difficulties that, for $n = 64$, the number m of CSA levels is 10, and, for $n = 128$, m is 11. We can also determine the intervals of n to which there corresponds a certain value of m, to be found in the table from Fig. 3.60 [Parh00]. If these data are interpreted, it results, for instance, that for any n within the integer interval [43, 63], we have $m = 9$. Similarly, if the table is extended, for any $n \in [212, 316]$ it results that $m = 13$, this also covers the value particular of interest $n = 256$.

Number of inputs	Level i	Number of outputs o_i
n	1	$\left\lfloor \frac{n}{3} \right\rfloor 2 + r_1 = o_1$
o_1	2	$\left\lfloor \frac{o_1}{3} \right\rfloor 2 + r_2 = o_2$
o_2	3	$\left\lfloor \frac{o_2}{3} \right\rfloor 2 + r_3 = o_3$
...	...	...
$o_{m-1} = 3$	m	$o_m = 2$

Fig. 3.59 Evaluation of the number of CSA levels corresponding to a combinational Wallace tree structure

n	m	n	m	n	m
3	1	13	5	63	9
4	2	19	6	94	10
6	3	28	7	141	11
9	4	42	8	211	12

Fig. 3.60 Table with the correspondence between the number of inputs and the number of CSA levels for a Wallace tree structure

We have insisted upon the table from Fig. 3.60 because it is the starting point in the controversy between two implementation versions of combinational tree structures, namely that which recommends instead of the above-presented Wallace tree the so-called Dadda tree. Specific to the Dadda strategy [GoSA06], is that, in comparison with a Wallace solution, it maintains the same number of levels m in the structure, but reduces the number of inputs in the tree configuration to a value which corresponds to a combinational array diagram of tree type with a chain of CSAs (Fig. 3.42, Fig. 3.52). Both methods minimize the number of adder cells and appeals, any time the diagram allows it, to half adder cells (HAC), and using full adder cells (FAC) only when their employment cannot be avoided. It should be pointed out that the elements of the controversy between the two solutions, Wallace versus Dadda, are represented by the numbers of FACs and HACs, as well as by the number of bits in the final CPA adder (Fig. 3.58). Within this context, mention should be made that some of the FAC cells of the CSAs have an input connected to 0 (refer also to Fig. 3.52, and Fig. 3.42 respectively), so that they can be substituted by HACs, and we again point out that the number of bits in the CPA (it being the carry propagation adder) decisively influences the performance of the entire configuration [Parh00, Omon94].

These aspects being specified, we highlight the fact that a Wallace tree tends to obtain the sums of the one bit products ($x_i y_j$) corresponding to the final result (P) digits as early as possible, aiming at the fastest solution, while the Dadda

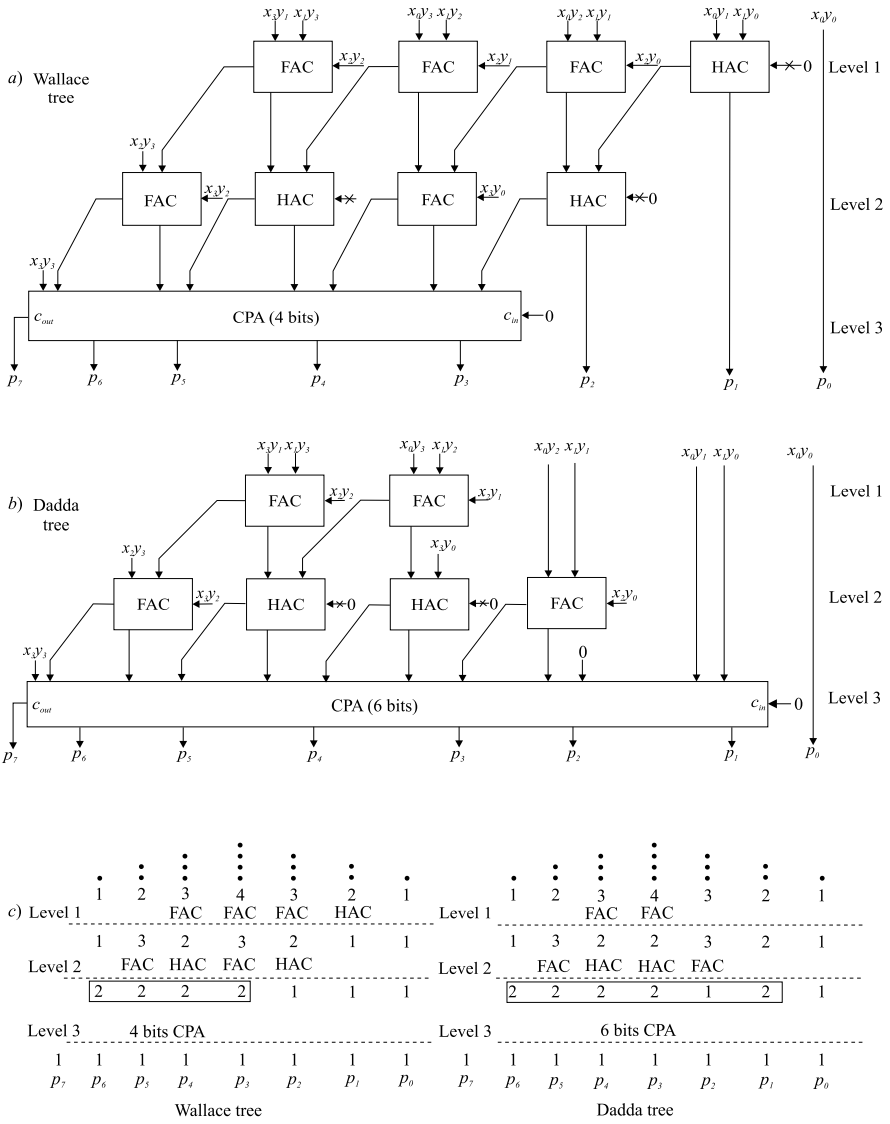


Fig. 3.61 Comparison in terms of number of FACs, HACs and CPA's ranks between the Wallace and Dadda tree structures

strategy, which keeps the length of the critical path in the CSA tree, postpones, as much as possible, the obtaining of these sums, usually leading to a simpler CSA structure, but possibly with a CPA which has an increased number of bits. For instance, let us consider the two options of tree devices for the case of 4·4 bits unsigned multiplication, presented in Fig. 3.61. Certainly, both of them start from the 16 one bit products $x_i y_j$ that have been obtained through an AND logic array, of

the type given in Fig. 3.41. In both designs, the HAC cells do not have the input connected to 0, marked as such in Fig. 3.61a and b. As regards the Wallace strategy (Fig. 3.61a), one can observe the exploitation of the early opportunity to combine the one bit products (refer to the FAC cell from the first level to which x_3y_1 is passed), so that, finally, for this solution five FAC cells, three HAC cells, and a CPA on 4 bits are necessary. For the same tree, Fig. 3.61c contains "a table" presentation where to each of the 16 one bit products (x_iy_j) corresponds a point in a special notation, the so-called "dot notation" [ErLa04]. These points are distributed to the final product digits, making up a "triangle" whose height is 4 (for x_0y_3, x_1y_2, x_2y_1 and x_3y_0) corresponding to p_3. The adding cells (FAC and HAC) from the first two levels (level 1 and level 2) with the corresponding input numbers, as well as the 4 bit CPA from level 3 are also presented. One can follow the connections reduction on levels, which results in the 4 ranks carry propagation adder. On the other hand, as regards the version based on the Dadda strategy (Fig. 3.61b), the addition of certain one bit products is delayed, being "pushed" to the structure depth (the case of the pairs (x_0y_2, x_1y_1) and (x_0y_1, x_1y_0) in level 2, and level 3 respectively), by which is obtained a saving of two cells in the first two levels (four FACs and two HACs in all, as compared to the five FACs and three HACs in the Wallace tree). This is counterbalanced by the increase by two ranks (from 4 to 6) of the carry propagation adder, making its construction attractive only when a very fast solution for the addition implementation is available. The way we arrive at the 6 bit CPA can be followed for the Dadda tree, as well, in Fig. 3.61c.

Mention should also be made that, by applying the product x_0y_1 to the free input (where there is 0, Fig. 3.61b) of the CPA rank with the output p_2, as well as the product x_1y_0 to the input c_{in} of the CPA, the total number of bits of the CPA can be reduced from 6 to 5. We also add that, as far as multiplier tree structures are concerned, hybrid solutions between the Wallace and Dadda trees may exist, which can lead to good performance-cost tradeoffs [Parh00].

At the end of this section, we shall synthetically refer to the combinational tree structures implementation of the Baugh-Wooley strategy. If we compare, for instance, expressions (3.26) and (3.39) concerning the parentheses with the largest number of terms (four at (3.26), as compared to six at (3.39)), it results that, in accordance with the values from Fig. 3.60, the Baugh-Wooley product form requires an additional CSA level (three as compared to two) with the consequential effects on performance. Within this context, trying to simplify the multiplier tree structure implementation, it is entirely justified to reduce the largest number of terms corresponding to the parentheses, as well as the height of the CSA construction implicitly, obtained by means of the Baugh-Wooley modified product form. Thus, investigating relation (3.43), we find that the parentheses weighted with 2^3 and 2^4 have only four terms, so that, according to Fig. 3.60, the reduction of the number of CSA levels (from three to two) can be achieved, which will result in a corresponding performance improvement.

3.11 Other Binary Multiplication Methods

In this section we shall briefly present some alternative trends in the construction of multiplication devices, compensating the section's concise form through the indication of an increased number of bibliographical works.

1. For many of the above-mentioned applications, but especially for array multiplications, computations of the type $w = xy + z$ frequently occur, implying a multiplication followed by an addition. Aiming to accelerate of such computations, many processors [HePa03, Haye98] are provided with special instructions regarding these multiply-add operations, as well as dedicated hardware units (combined multiply-add units) which enable the efficient implementation, as far as cost is concerned, of two operations. Similarly, many digital signal processors (DSPs) have built-in hardware facilities for multiply-add, as well as for multiply-accumulate operations [DaTa05, ErLa04], the latter representing a multiply-add option useful for the estimation of a sum of products.

   One efficient solution for the implementation of these units consists of appealing to so-called additive multiply modules [Parh00], whose feature is the inclusion of operand z, which is to be added after the xy multiplication process. Thus, there are constructions that comprise, for the multiplication operation, a CSA tree structure of the type seen above, which, before executing the final addition through a CPA, provides a supplementary CSA level for the addition of operand z. Alternatively, the multiply-add computation process is not executed in the CSA form in a successive way, but through a merged multiply-add operation. This last operation does not distinguish between the one bit products $x_i y_j$ and the terms z_k, of the same weight, of operand z that has to be added, but treats them in the same unitary way [ErLa04].
2. The construction of multiplication devices may involve the synthesis of such a device that realizes multiplication with operands of $2n \times 2n$ dimension when $n \times n$ multipliers are available. The solution consists of applying the "divide-and-conquer" strategy [Parh00]. Its nature is that it starts from the two operands on $2n$ bits, X and Y, which it considers to be halved, i.e., for multiplier X, we have the parts X_H and X_L (with indexes from "high" for the more significant part, and "low" for the less significant part), and for multiplicand Y, we have the parts Y_H and Y_L. Depending of how much the operation is parallelized, by means of one to four $n \times n$ multipliers the four partial products $X_L Y_L$, $X_L Y_H$, $X_H Y_L$ and $X_H Y_H$ are computed, as presented in Fig. 3.62 [Parh00, DaTa05]. These four values have to be added in order to obtain the final product. Rearranging the partial products as shown on the right side of Fig. 3.62, and considering that $X_H Y_H$ and $X_L Y_L$—since they do not overlap—form one number, we have to add, in fact, only three values. Consequently, the initial problem of the synthesis of a $2n \times 2n$ multiplier has been reduced to that of an $n \times n$ multiplier and an adder of three operands.

   Obviously, in a similar manner, one can configure multipliers for $3n \times 3n$ bits, $4n \times 4n$ bits, etc., on the basis of constructive blocks for $n \times n$ bit multiplication [ErLa04].

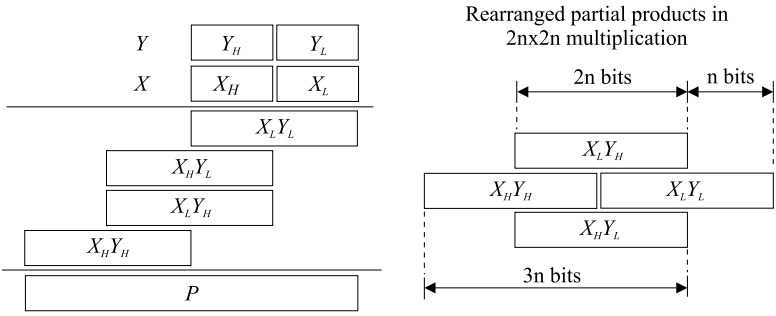


Fig. 3.62 Application of the "divide-and-conquer" strategy to the multiplication of $2n$-bit operands with an n-bit multiplier

3. A special case is squaring, and, by extension, exponentiation. Any of the presented multiplication devices can execute the operation $P = X^2$, where we have the multiplicand $Y = X$. However, in case squaring and, generally, exponentiations are more frequent, it is usually worth investing in a dedicated multiplier "built in" to the hardware, because its cost is reduced and its delay is lower than that of a universal multiplication device.

   The simplification brought about by a device dedicated to squaring can be ascertained if we revert to the example given by (3.24), where Y will be substituted by X, let us intervene in relation (3.26) by taking into account that $x_i x_i = x_i$ and $x_i x_j = x_j x_i$. Consequently, the following will be obtained:

$$P = x_3 2^6 + (x_2 x_3 + x_2 x_3)2^5 + (x_1 x_3 + x_2 + x_1 x_3)2^4 + (x_0 x_3 + x_1 x_2 + x_1 x_2 + x_0 x_3)2^3 + (x_0 x_2 + x_1 + x_0 x_2)2^2 + (x_0 x_1 + x_0 x_1)2^1 + x_0 2^0 \tag{3.44}$$

   If, in (3.44), we take into account that the multiplication of product $x_i x_j$ by 2 means, in fact, its moving into the parenthesis associated with the next larger weight, then we have:

$$P = (x_2 x_3 + x_3)2^6 + x_1 x_3 2^5 + (x_0 x_3 + x_1 x_2 + x_2)2^4 + x_0 x_2 2^3 + (x_0 x_1 + x_1)2^2 + x_0 2^0 \tag{3.45}$$

   Relation (3.45) can still be changed if the following obvious identities are taken into account:

$$x_0 x_1 + x_1 = 2x_0 x_1 + x_1 - x_0 x_1 = 2x_0 x_1 + (1 - x_0)x_1 = 2x_0 x_1 + \overline{x_0} x_1 \tag{3.46}$$

   If (3.46) is applied twice, (3.45) becomes:

$$P = (x_2 x_3 + x_3)2^6 + (x_1 x_2 + x_1 x_3)2^5 + (x_0 x_3 + \overline{x_1} x_2)2^4 + (x_0 x_1 + x_0 x_2)2^3 + \overline{x_0} x_1 2^2 + x_0 2^0 \tag{3.47}$$

The above-mentioned changes have a benefic effect because they reduce the number of terms from the parenthesis weighted with 2^4 (from three in relation (3.45) to two in (3.47)), thus enabling the saving of a CSA level in the synthesis realised, for instance, through a Wallace tree (Fig. 3.61a). Mention should also be made that (3.46) might have been applied to the parenthesis weighted with 2^6, as well, but the consequence would not have been a favorable one, but the opposite. Generally, speculations such as those seen above may result in improved solutions as far as performance and cost are concerned (reductions of the number of CSA levels, as well as of the number of CPA adder ranks may also result, when implementations with combinational tree structures are executed).

As regards squaring, we should like to point out that it may contribute to multiplication implementation by means of an artifice, namely use of the so-called "arithmetic lookup tables" [ErLa04, Omon94]. By means of this method, for numbers represented on n bits, the products are stored as elements of $2n$ bits in a table whose lines are the 2^n possible values of, for instance, multiplier X, and whose columns are the 2^n possible values of multiplicand Y. In these conditions, for such a lookup table meant for multiplication, there results a prohibitive dimension (roughly estimated at $2^n \times 2^n \times 2n$ bits), which, for practical values of n, makes the lookup process very difficult. This impasse is surmounted by squaring, whose lookup table, for the same number n of bits, has a much reduced dimension (estimated, under the same hypotheses, at $2^n \times 2n$ bits) with favorable consequences, both as regards the saving of memory area, and lookup simplification. Following this specification, let us appeal to an obvious identity, which presents the product of numbers X and Y as a difference of squares, namely: $XY = ((X+Y)^2 - (X-Y)^2)/4$. If an adder/subtracter is activated twice (for the computations $(X+Y)$, $(X-Y)$), and than if two lookups in the table with squared values are executed (to determine $(X+Y)^2$,$(X-Y)^2$), the difference $((X+Y)^2 - (X-Y)^2)$ is taken, and, finally, if a right-shift by two ranks of this difference is executed, the intended value of product XY is obtained.

Let us also refer to exponentiation of the form X^α, where α is an unsigned number. This can be executed through a sequence of squaring steps or through combinations of such steps with some multiplication ones. For instance, we have $X^{11} = X(X(X^2)^2)^2$.

4. A special case is represented by the residue numbers which have various applications in the field of computation, of which mention should be made of error detection and correction for arithmetic operations. The multiplication of these numbers is done by using special devices, the so-called modular multipliers [Parh00, KaTa05, RaFu89]. Such a multiplier allows the obtaining of the modulo product related to a fixed, constant value, called the module and denoted by μ. One of the implementation solutions is represented by attaching to the output of a binary multiplier, of the studied type, a diagram which will enable the residue to be obtained when the resulting product value is divided by μ, executing the so-called modular reduction operation. The disadvantage of this solution is the requirement of the intermediate storage of some values, generally implying a significant number of bits. This disadvantage can be mitigated by combining

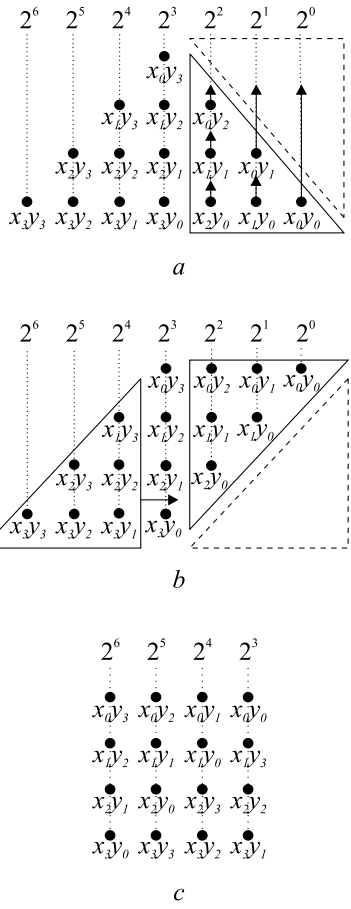


Fig. 3.63 Identification of one bit products in terms of dot notation and their regrouping in order to synthesize a modulo 15 4-bit operands multiplier

the modular reduction with the accumulation of the partial products, an aspect upon which we shall insist below.

The synthesis of modular multipliers depends, decisively, upon the value of the module μ. Without detailing the discussion connected with the choosing of μ [EfVN03, Vlăd86], we show that, especially the cases when $\mu = 2^n$ and $\mu = 2^n - 1$, where, again, n represents the dimension of the operands, are cases of interest on account of being simple. Without loss of generality, we shall refer to unsigned integer operands. Thus, accumulating partial products through CSA addition, for $\mu = 2^n$, the carry output from the most significant adder cell $(n - 1)$ shall be ignored, while for $\mu = 2^n - 1$, the same carry output shall be applied to the least significant adder cell 0 from the next stage. We mention, without further discussion [RaFu89], that if it is supposed that $n = 4$, then for $\mu = 2^4 = 16$, the carry from rank 3 of the CSA represents the value 16, but 16 mod 16 = 0, which justifies us to ignore it. Similarly, for $\mu = 2^4 - 1 = 15$, the same carry of value

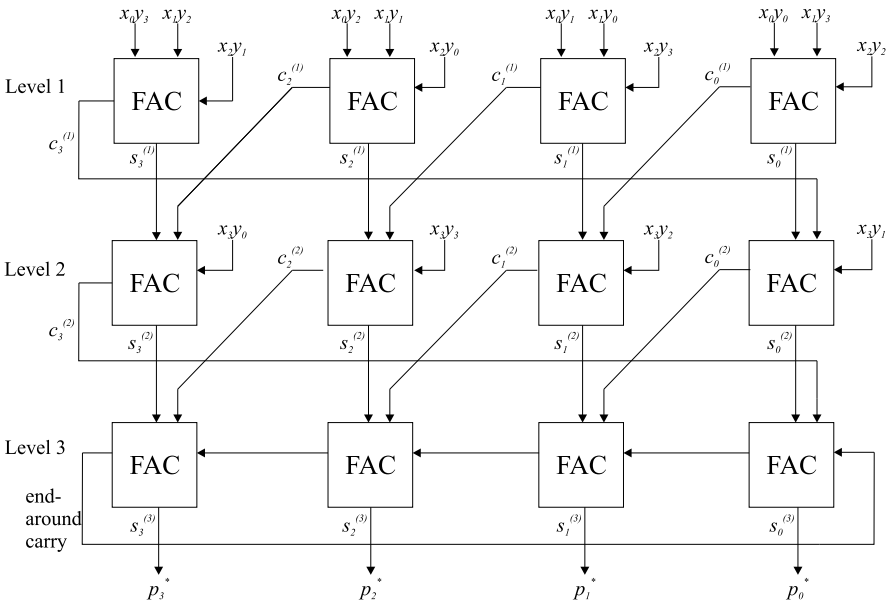


Fig. 3.64 FACs interconnections into a modulo 15 4-bit operands binary multiplier

16 leads, due to 16 mod 15 = 1, to one binary unit being added to rank 0 of the next CSA level.

Referring to the same example, and adopting $\mu = 2^4 - 1 = 15$, if we appeal to dot notation, according to relation (3.26), and also to Fig. 3.61c, we have the distribution of the dots that correspond to one bit products, as presented in Fig. 3.63a, marking, through the triangle plotted with a solid line, those dots corresponding to the products, which, if added, have the weights $2^0(x_0y_0)$, $2^1(x_0y_1 + x_1y_0)$ and $2^2(x_0y_2 + x_1y_1 + x_2y_0)$. Since the weights of the given sums do not change, they can be moved to the vacant places marked by the triangle with a dotted line, so that the dots distribution from Fig. 3.63b is obtained. Mention should be made that for the weights corresponding to the dots marked with the triangle plotted with a solid line in Fig. 3.63b, we have the following equalities: $2^4 \bmod 15 = 16 \bmod 15 = 1$ (weight 2^0), $2^5 \bmod 15 = 32 \bmod 15 = 2$ (weight 2^1), and $2^6 \bmod 15 = 64 \bmod 15 = 4$ (weight 2^2). This justifies the given movement of dots into the vacated positions marked by the triangle plotted with dotted line in Fig. 3.63b. Consequently, the dots have been redistributed, resulting in the matrix configuration from Fig. 3.63c, which shows the new positions of some of the one bit products, and the way they have to be added in order to obtain modulo 15 partial products.

If we foresee an implementation through CSA vector computations, then the FACs' structure corresponds to the distribution of dots from Fig. 3.63c, the rhomboid form of the structure from Fig. 3.42 being substituted by a rectangular one, as presented in Fig. 3.64. As mentioned above, one can observe the connection

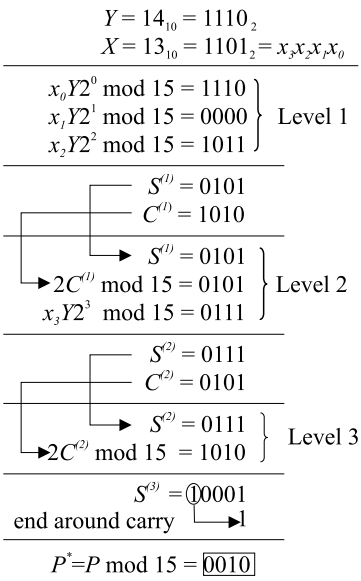


Fig. 3.65 Example of a modulo 15 4-bit binary multiplication

of the carry-out lines (c_{out}) from the msb rank of a CSA level to the lsb rank of the next CSA level, as well as the end-around carry connection to the CPA, which has been assumed to be of RCA type, for simplicity reasons.

For instance, in Fig. 3.65, we have taken into account the multiplication $XY = 13 \times 14$, whose product $P = 182$ leads, if modulo 15 is used, to the value $0010_2 = 2_{10}$. As can be seen, it has not been obtained by computing the product in extenso, i.e. the value $182_{10} = 10110110_2$, and, then by applying to it the modular reduction, but this last operation has been combined with the accumulation of the partial products.

The statements applied to the previous rudimentary case can be extended to various values of the module μ, granting, by means of the residual codes, an important alternative for the checking of arithmetic operations, in general, and for binary multiplication, in particular.

Chapter 4
Functional Analysis and Synthesis of Binary Division Devices

4.1 Binary Division Methods

It is specified from the beginning that this chapter refers to the execution problems of binary division especially in fixed point, some of the solutions being applicable also to floating point units. The operands are made up of the dividend Y and the divisor X, and the results of the operation are represented by the quotient Q and the remainder R. If these numbers are integers, then they are in the correlation known as the identity of division, according to which:

$$Y = XQ + R \tag{4.1}$$

where the sign of the remainder R is the same as that of the dividend Y and $|R| < |X|$, the bars $|\ |$ signifying absolute values, and R may also be 0 [ErLa04, HePa03, Parh00].

To get accustomed to the problems specific to this operation, we shall first refer to the simpler case of unsigned integers, whose division is, obviously, completely defined by relation (4.1). Thus, let us consider the case where the values X, Q and R are represented on $n = 4$ bits, while Y has double the number of bits, $2n = 8$. The operation development according to the well-known conventional method "paper and pencil" [Wake00, Haye98], which, in the sequential way, allows the quotient to be obtained one bit at a time, can be followed in Fig. 4.1. The dividend $Y = 153_{10} = 10011001_2$ is divided by divisor $X = 11_{10} = 1011_2$, the operation being executed through a sequence of steps sequence, in each of which the subtraction given by:

$$R_{i+1} := R_i - q_{n-1-i} 2^{n-1-i} X \tag{4.2}$$

is executed where R_i and R_{i+1} represent the current partial remainder and the next one, while q_{n-1-i} represents the current binary digit of the quotient Q.

If the difference between the remainder R_i and the divisor X, adequately shifted to the right, $R_i - 2^{n-1-i} X$, is negative (refer to the value 1 of the borrow from the position of the "sign", in Fig. 4.1), then to the current bit of the quotient, q_{n-1-i}, is allocated the value 0, and when it is zero or when it is positive (refer to the value

M. Vlăduţiu, *Computer Arithmetic*,
DOI 10.1007/978-3-642-18315-7_4, © Springer-Verlag Berlin Heidelberg 2012

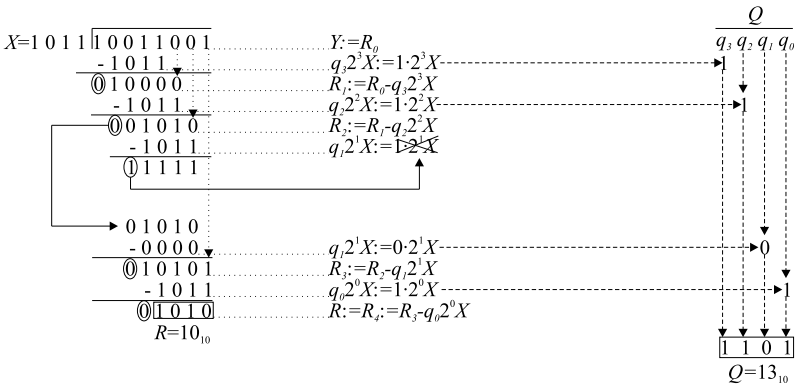


Fig. 4.1 "Paper and pencil" conventional binary division example

0 of the borrow from the position of the "sign", Fig. 4.1), then to the current bit of the quotient, q_{n-1-i}, is allotted the value 1. Characteristic of this procedure is the fact that the partial remainders (in Fig. 4.1, only the bits of interest have been specified at each moment) maintain their fixed position during the entire procedure, while divisor X is subject to a right-shift at every step by one binary position.

Undoubtedly, the central problem of division operation is to determine the current digit of the quotient. If we have a number system with radix r, then this problem requires the execution of r comparisons of the r multiples of divisor X with the current partial remainder R_i. In the binary case, when $r = 2$, the two comparisons can be substituted by the subtraction, given by (4.2), through which it is tested whether to the quotient current bit can be allocated the value 1, a fact due to which this operation is, sometimes [Omon94, Haye98], called trial subtraction.

Our aim being the implementation of the binary division operation through fast hardware mechanisms, in as economical a way as possible, let us try to make use of the same structural elements which have been used in the binary multiplication device. This being our priority, we shall make certain changes in the conventional division procedure, which uses iteration (4.2) and is exemplified in Fig. 4.1. Adopting a procedure similar to that of multiplication, we shall maintain the fixed position of divisor X and shall shift, this time to the left, the partial remainders. Anticipating the same presentation manner based mainly on examples, let us reconsider the case from Fig. 4.1. The operation development can be followed in Fig. 4.2. The commentaries will be made by comparison. Thus, there is a difference between the consideration of the initial partial remainder, namely $Y := R_0$ in the conventional method, and $Y := 2R_0$ in the new procedure. The iteration is also changed, as compared to that described in (4.2), namely:

$$R_{i+1} := 2R_i - q_{n-1-i}X \tag{4.3}$$

where, besides the notations with the same significance as in (4.2), $2R_i$ appears which is the current partial remainder shifted by one binary position to the left.

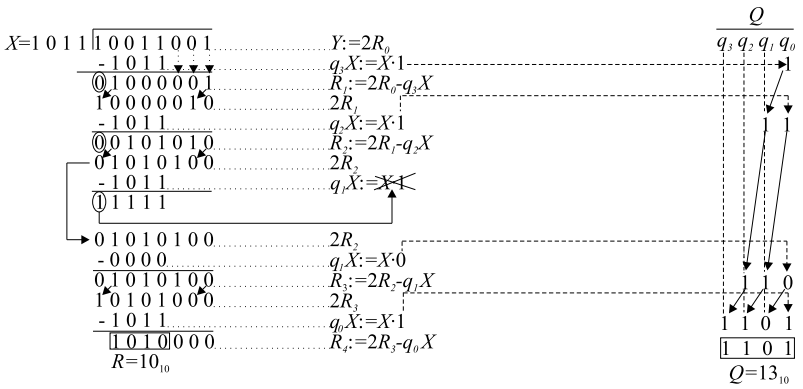


Fig. 4.2 Binary division example with unchanged position of the divisor and left-shifted partial remainders

Even if the quotient bits are generated in the same order, in Fig. 4.2 it can be observed that they are subject to a shift, at each iteration, by one position to the left. Correlated with this, the final remainder R does not coincide with that which has been obtained at the end of the last iteration (R_4), as in Fig. 4.1, but represents the remainder R shifted to the left, in our case (Fig. 4.2), by three binary positions, i.e. $R_4 := 2^3 R$.

In the two figures we have tried to show the information flow at the bit level, which is why we shall not give a commentary on the procedure. However, we should like to reveal the iteration that corresponds to the determining of the bit q_1, when the attempt to set its value to 1 fails, a situation which requires $2R_2$ to be used as the value of the partial remainder. Mention should also be made that the values of the quotient $Q = 1101_2 = 13_{10}$, and of the final remainder $R = 1010_2 = 10_{10}$ coincide in the two procedures, and these values together with those of the operands X and Y, precisely satisfy the identity (4.1).

It is just the failed attempt to set the current bit of the quotient to 1 which requires different solutions to the problem of restoring the deteriorated remainder through the corresponding trial subtraction. The specific ways this problem is solved is the difference between the fundamental procedures of binary division. Essentially, there are two such procedures, namely with and without restoring the remainder, whose characteristics will be analyzed below.

4.1.1 Restoring Division

Referring to the favorable computer implementation procedure, whose iterative steps are described in (4.3) and which is exemplified in Fig. 4.2, let us suppose that in step i it has resulted, following the trial subtraction, that $2R_i < X$. This implies the assigning of value 0 to the current bit of the quotient, q_{n-1-i}, but in the

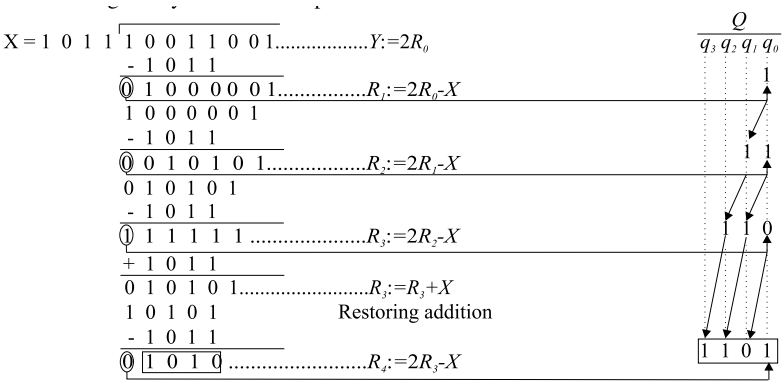


Fig. 4.3 Restoring binary division example

meantime, to the next partial remainder, R_{i+1}, there has been attributed the value of the difference $(2R_i - X)$. According to (4.3), and since $q_{n-1-i} = 0$, the remainder R_{i+1} has to return to the value $2R_i$, which can be done by the immediate addition of divisor X to the difference $(2R_i - X)$, when $2R_i < X$ i.e. $R_{i+1} := 2R_i - X + X$ or, otherwise, $R_{i+1} := R_{i+1} + X$. This is the case of remainder $2R_2$ from Fig. 4.2, which from 01010, through the subtraction of X, becomes 11111 and has to be restored to the old value, which requires a restoring addition, as shown in Fig. 4.3. As can be observed, the current value of a bit of the quotient is given by the complement of the borrow bit from the msb position of the partial remainder. When this bit obtains the value 1, restoring addition becomes necessary. Mention should also be made that, unlike the presentation from Fig. 4.2, the partial remainders have no longer been completed with 0s at the right side of the lsb of dividend Y. We have proceeded in this way, anticipating the procedure implementation, to "create a place" for the quotient bits which will be put into the released binary positions.

Nonetheless, the restoring additions increase the number of elementary operations corresponding to the procedure, increasing the latency required by division. Thus, if we suppose, generally, the quotient has n bits and that on average half of its binary digits are 0, this means that we require of $3n/2$ activations of an adder/subtracter by the division device configuration, because to the n trial subtractions there are added $n/2$ restoring additions.

The situation can be improved by appealing to the method known as the "nonperforming division", whose essential characteristic consists in the storage of the partial remainders until the result of the trial subtraction is known. Thus, the remainder $2R_i$ is kept in a register, and in case $2R_i < X$, instead of restoring the remainder by a restoring addition, the value is taken from the register where it has been stored. Obviously, this solution saves an addition for each 0 bit of the quotient, but requires a supplementary investment in the mechanism of remainder conservation. Some references in the literature [Omon94] consider nonperforming division as a distinct division method, but most of them [ErLa04, HePa03, Parh00] treat it as

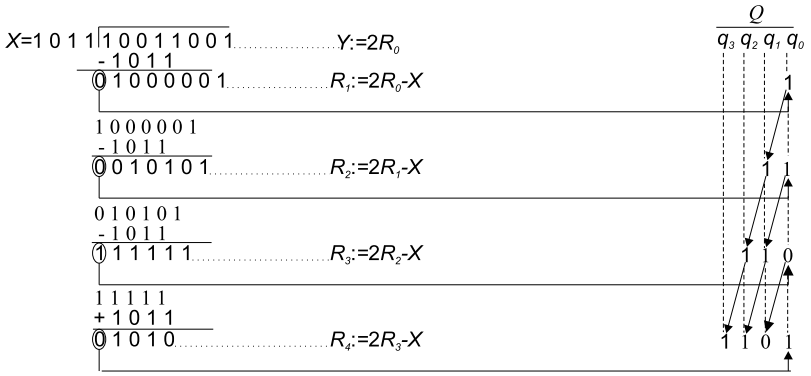


Fig. 4.4 Non-restoring binary division example

an alternative to restoring division, with which it has all the elements in common, except the way in which the remainder is restored.

4.1.2 Non-restoring Division

To introduce this procedure, let us suppose that in step $(i-1)$, which precedes the current one, the value 0 has resulted for the quotient bit, i.e. $q_{n-i}=0$, which, according to restoring division, requires the restoring of the remainder through $R_i := R_i + X$. Respecting the procedure, it follows the shift to the left by one position of the current remainder $(2R_i)$ and, then, immediately, the trial subtraction, i.e. $R_{i+1} := 2R_i - X$, where, through the merging of the two relations, the following will be obtained:

$$R_{i+1} := 2(R_i + X) - X := 2R_i + X \tag{4.4}$$

Otherwise, after assigning the value 1 to the quotient bit $(q_{n-i}=1)$, we have, as we have had so far, a subtraction $(R_{i+1} := 2R_i - X)$ followed by the left-shift, but, after attributing value 0 to the quotient bit $(q_{n-i}=0)$, we have, and here appears the difference from the restoring method, an addition $(R_{i+1} := 2R_i + X)$, followed, this time, as well, by a left-shift. Since it is no longer necessary to restore the remainder, the new procedure has been called non-restoring division. Figure 4.4 resumes the example used to illustrate the other methods according to the new procedure conditions, where the saving of one activation of the adder/subtracter can be observed for the quotient bit $q_1=0$. Generally, if we suppose the hypothetical case mentioned before, when the probabilities for the occurrence, in the quotient binary configuration, of 0 and 1 bits are equal, the non-restoring division method requires only n activations of the adder/subtracter, saving, on the one hand, the $n/2$ additions from the restoring division, and, on the other hand, the storage and shifting mechanisms (when we have quotient bits of 0) of the partial remainders from the non-performing division, but the non-restoring division requires a more complex control.

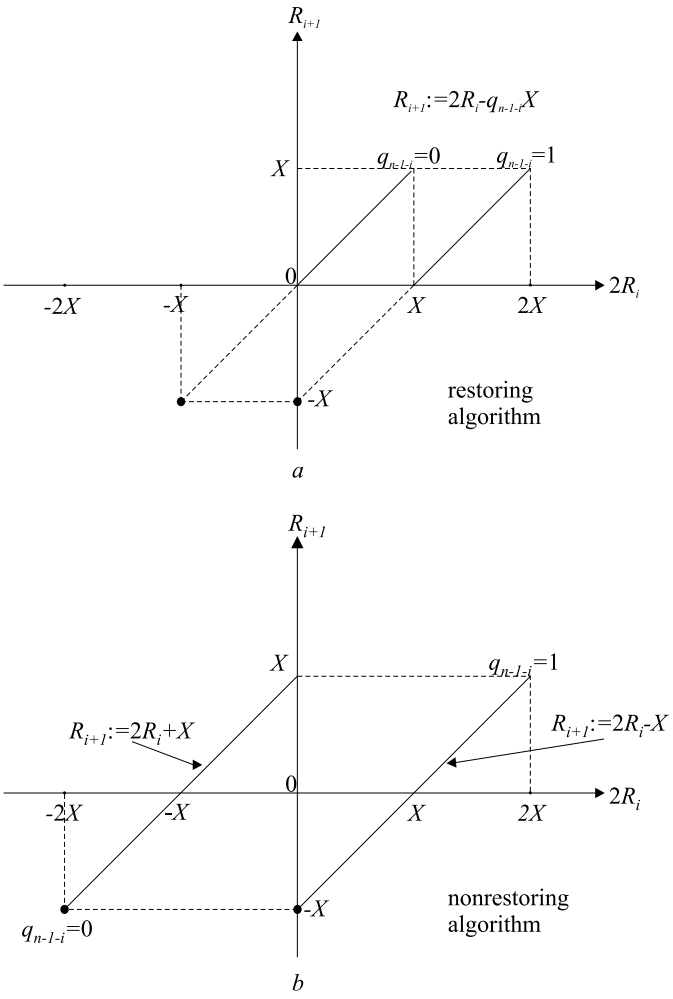


Fig. 4.5 Robertson diagrams corresponding to the restoring and non-restoring algorithms

One more minor complication appears in the non-restoring algorithm, namely in that particular case when the last but one partial shifted remainder is smaller than divisor X, i.e. when the final remainder is "negative" (if numbers without sign are operated on, this situation shall be avoided). Correctively, one has to return to the value of the last but one partial shifted remainder, by using a supplementary addition of X to the final "negative" remainder.

Let us try to highlight the differences between the two methods, restoring and non-restoring, appealing to the graphical form associated with each of them, also called the Robertson diagram [Omon94]. This consists of a rectangular system of axes, with the current shifted partial remainder represented in the abscissa ($2R_i$), and the next partial remainder represented in the ordinate (R_{i+1}). Within this system

are traced the lines corresponding to (4.3), the result being the dependences from Fig. 4.5a for the restoring algorithm, and the dependences (corresponding to the two different equations) from Fig. 4.5b for the non-restoring algorithm. In Fig. 4.5a, the two lines (for $q_{n-1-i} = 0$, and for $q_{n-1-i} = 1$ respectively) are dotted in the region of the negative partial remainders R_{i+1}, to take into account the effect of the previously specified restoring additions. By taking into account the variation intervals of the partial remainders, for both procedures, the lower ends of these lines, unlike their upper ends, were marked using emphasized dots.

Coming back to some aspects of general character that are specific to division, we should like to point out that, as regards quotient Q represented on n bits, its largest value may be $(2^n - 1)$, which, corroborated with condition $R < X$ and by applying (4.1), leads to the restriction $Y < (2^n - 1)X + X = 2^n X$. This means that the value corresponding to the most significant n bits of dividend Y shall be, strictly, smaller than that of divisor X. The above-mentioned restriction has to be tested before applying the division algorithm proper, representing the so-called overflow check, which may appear at the quotient level—the quotient overflow [ErLa04]. Mention should also be made that through this test is detected, in the case of unsigned division, the anomaly of the divide-by-0 (zero) condition.

Up to now, we have referred to the division of operands representing unsigned integers, but if the operands are signed integers then no essential differences appear. Regarding signed fractional numbers, things must be reworded to a certain extent. Thus, if both members of relation (4.1) are multiplied by 2^{-2n}, there results [Parh00]:

$$\begin{aligned} &Y2^{-2n} = \left(X2^{-n}Q2^{-n}\right) + R2^{-2n}, \quad \text{i.e.} \\ &Y_{frac} = (X_{frac}Q_{frac}) + R_{frac}2^{-n} \end{aligned} \tag{4.5}$$

where the index *frac* has been used to specify fractional values.

Otherwise, the same procedures can be applied both to integers and to fractions, with the specification that the final remainder has to be shifted by n positions to the right in the case of fractional numbers. Mention should also be made that one must avoid quotient overflow, which, in the case of unsigned integers, has the form $|Y| < 2^n|X|$, and in the case of signed fractions, has the form $|Y_{frac}| < X_{frac}$.

4.2 Sequential Binary Divider for Unsigned Integers

We shall start this section by mentioning that as regards multiplication and division there are, besides inherent differences, certain similarities. In the sequential version, there is generated one bit of the result (product and quotient) at each iteration of the procedure, in a similar manner, through steps that imply arithmetic operations succeeded by steps that provide shifts. If we refer to the basic procedures and not to their improvements, then we may say that the arithmetic operations differ, i.e. in the case of multiplication there have to be made repeated additions, while in the case of division there have to be made repeated subtractions. The direction of shifts

also differs. In multiplication, the partial products are shifted to the right, while in division, the partial remainders and the quotient are shifted to the left. There are also certain less obvious differences, such as the supplementary problem encountered by division and which consists of the selection or, as will be seen below, the estimation of the quotient digits. There is one more aspect in connection with the fact that the multiplication of two numbers of n bits will always give a product represented on $2n$ bits, while the quotient, on the division of a number of $2n$ bits by one of n bits, may result of a length longer than n bits (which requires the overflow check, including the divide-by-0 condition testing). But the essential aspect, as far as the objective of this section is concerned, is that the implementations corresponding to the two operations have many constructive elements in common, which, by reconfiguring the structures, enables the synthesis of some most favorable arithmetic units, as regards cost.

We also add that our references include the implementation of the better performing algorithm, which, as has already been seen, is, generally, the non-restoring one [ErLa04, Parh00]. We construct the presentation in the manner that has been used with the multiplication devices, supposing, the general character being still maintained, that the length of the registers is 8 bits. Thus, Fig. 4.6 presents, in terms of the adopted description language, the code sequence associated with the hardware configuration of the division device given in Fig. 4.7 (after [Haye98]). As regards the description and the diagram, mention should be made that, for simplicity reasons, there have been omitted the overflow check and the divide-by-0 condition testing, which are supposed to be solved through software routines. The registers' structure can be recognized from the multiplication devices, but the registers' functions change. Thus, dividend Y is loaded into the double register A.Q, with the most significant bits in A. We appeal to one more simplification, namely, in order to set the first bit of the quotient in Q[0], we consider that the dividend is restricted, as regards its dimension, to 15 bits, and, generally, to $(2n - 1)$ bits. Obviously, the problem of the storage of the first binary digit of the quotient can be solved in other ways, as well, for instance, that which starts with an iteration with one step shift to the left [Omon94]. Due to the bus width, the loading of Y into the double register A.Q is controlled through the successive signals c_0 and c_1. Subsequently, the partial remainders and the quotient are formed in register A.Q, so that, finally, in Q we shall obtain the quotient, while the remainder will be obtained in A. Register A.Q is provided with left-shift capacity, as required by the algorithm (Fig. 4.4), the partial remainders being diminished, at each iteration, by one bit, the quotient bits being "pushed" into the released positions. There can also be observed a certain resemblance with the multiplication operation, where, through right-shift, there have been "lost" multiplier X bits, there being "gained" bits of the partial products, and, in the division operation, through left-shift, there have been "lost" bits of the partial remainders, there being "gained" bits of quotient Q. Consequently, the double register A.Q must be provided with bidirectional shifting capacity in order to be used in both operations [Stal99].

As regards register A, mention should be made that it is extended to the left by one rank (S), which could have been denoted A[8], but to which has been assigned

```
          divider 1
          declare register A[7:0], Q[7:0], M[8:0], COUNT[2:0], S;
          declare bus INBUS[7:0], OUTBUS[7:0];
   BEGIN: COUNT:=0, S:=0,                                              {c0}
   INPUT: A:=INBUS ;
          Q[7:1]:=INBUS[7:1];                                          {c1}
          M[7:0]:=INBUS[7:0], M[8]:=0;                                 {c2}
SUBTRACT: S.A:=S.A-M;                                                  {c3,c4}
    TEST: if S=0 then
            begin
              Q[0]:=1,                                                 {c5}
              if COUNT7=1 then go to CORRECTION; else
                 begin COUNT:=COUNT+1,S.A.Q[7:1]:=A.Q; end             {c6}
              S.A:=S.A-M, go to TEST;                                  {c3,c4}
            end
          else
            begin
              Q[0]:=0,                                                 {c7}
              if COUNT7=1 then go to CORRECTION; else
                 begin COUNT:=COUNT+1,S.A.Q[7:1]:=A.Q; end             {c6}
              S.A:=S.A+M, go to TEST;                                  {c3}
            end
CORRECTION: if S=1 then S.A:=S.A+M;                                    {c3}
  OUTPUT: OUTBUS:=Q;                                                   {c8}
          OUTBUS:=A;                                                   {c9}
     END:                                                              {END}
```

Fig. 4.6 Description of the restoring binary division for unsigned integers

the name S, from "sign", to highlight its function. At each iteration, through left-shift, the msb of the current partial remainder is placed in S. Then, S contributes to the operation (addition or subtraction), and stores the sign of the result represented by the next partial remainder. The value thus obtained is tested and, depending on the result is set up in Q[0], the quotient current bit, a fact suggested in Fig. 4.7 by passing the control signals c_5 and c_7 to the asynchronous inputs of the flip-flop Q[0] (Set(S*) and Reset (R)). As has been established above, the left-shift is performed on the double register A.Q, with the extension S included, i.e. on S.A.Q, an elementary operation which is under the control of signal c_6, which also controls the incrementing of the iterations counter COUNT, this being a nonconflictual operation.

The other operand represented by divisor X, whose position remains fixed during the entire division, is loaded in register M. Like A, M has an extension to the left by one rank, (M[8]), this being set to 0, controlled through c_2, synchronously with the loading of X into the other ranks of M. Bit M[8] represents the correspondent of the previously specified bit S attached to A, so that, for the extended operand X will be obtained, by the EX-OR wordgate, and controlled by c_4, the two's complement. In the case of subtraction, it will be added, under the control of c_3, to the most significant part of the partial shifted remainder from S.A. On the other hand, in the case of addition, the value 0 (i.e. the positive "sign" of the number without sign X) from M[8] has no effect.

After trial subtraction (provided by SUBTRACT), and depending on the value of S, follows the setting of the quotient current bit (Q[0]), when there are two alternative subsequences (provided by TEST). Both of them also include operations

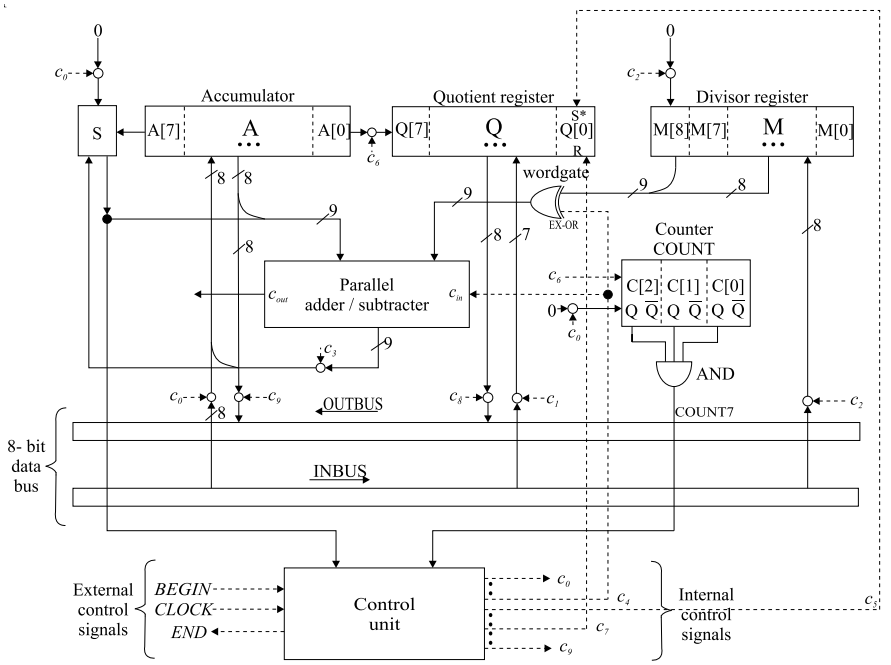


Fig. 4.7 Block diagram of a sequential binary divider for unsigned integers based on restoring procedure

that test the end of the division (when COUNT7 = 1), that increment the iterations counter COUNT, as well as left-shift operations by one binary position of the partial remainder and of the quotient (S.A.Q[7:1] := A.Q). The new partial remainder is also formed, which, according to Fig. 4.4, implies, when the current bit of the quotient is set to 1, a subtraction (S.A := S.A − M), and, when the current bit of the quotient is set to 0, an addition (S.A := S.A + M). If the final result is "negative" ($S = 1$), then it will be adjusted by adding to it the divisor, an operation provided by the label CORRECTION. The results, consisting of the quotient and the "shifted" remainder, are delivered to the bus under the control of signals c_8 and c_9.

As has happened in most of the multiplication methods, let us consider an example in this case as well. We shall adopt the example that has been used in the illustration of multiplications, but this time we shall process integers without sign. Thus, let us suppose dividend $Y = 9345_{10} = 10010010000001_2$ and divisor $X = 89_{10} = 01011001_2$. The division process can be followed in Fig. 4.8. Before the setting of the values of S, let us mention that the values corresponding to Q[0] are marked with an underline, signifying the fact that, anyway, they will be overwritten. After the computation process, which can be followed at the bit level, quotient $Q = 01101001_2 = 105_{10}$ and remainder $R = 0$ will be obtained.

Finally, mention should be made that as regards the implementation from Fig. 4.7, an alternative solution may consist of merging the control signals c_5 and c_6,

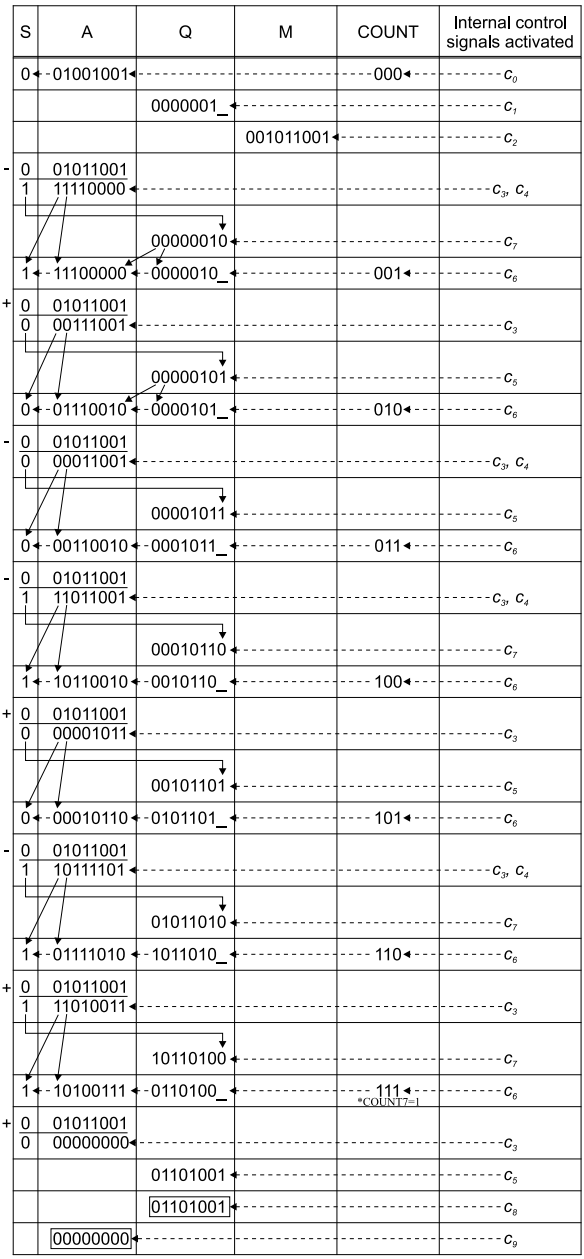


Fig. 4.8 Binary division example of unsigned integers by the non-restoring procedure with microoperations' control signals activation

on the one hand, and c_7 and c_6 on the other hand by making use, through an elaborate design, of both fronts of the CLOCK pulse. However, the elimination of a control signal might require the extension of the CLOCK period, and, consequently, the best solution depends, decisively, on the available technology.

We also add that the changes required by the restoring procedure and by its alternative, the nonperforming procedure, are obvious, and they can easily be grafted onto both the behavioural description from Fig. 4.6, and the diagram presented in Fig. 4.7, so we shall not discuss this aspect.

4.3 Combinational Array Structures for Binary Division

Before considering other binary division methods, let us present, for the fundamental procedures that have been introduced, synthesis versions alternative to the "one-bit-at-a-time" sequential ones, of the type presented in the previous section, aiming to obtain—in a totally parallel way—all the bits of the quotient and of the remainder through only one CLOCK pulse. As for binary multiplication, we appeal to combinational array structures for this purpose.

We shall present one solution for the remainder non-restoring procedure and one solution for the restoring one. In both solutions, the problems of overflow check and divide-by-zero will be considered to be solved through the corresponding software routines. In order to cover as many difficulties met with at implementation as possible, we shall suppose, for the non-restoring case, fractional operands provided with signs, and for the restoring case, operands which are integers without signs.

4.3.1 Combinational Array Structure Based on Non-restoring Division

The operands being provided with signs, as mentioned above, we shall first point out the specific aspects of this case. We shall gradually arrive at the synthesis of a combinational array structure, presenting, first of all, the description associated with a sequential variant of the non-restoring algorithm, according to the model from Fig. 4.6. The general character being still maintained, we shall consider the fractional numbers represented on 5 bits, of which the msb represents the sign. Aiming, in fact, at the synthesis of the parallel variant, we shall ignore the control signals which are specific to the sequential implementation. These specifications being made, the description is given by the code sequence from Fig. 4.9 [VlPe94].

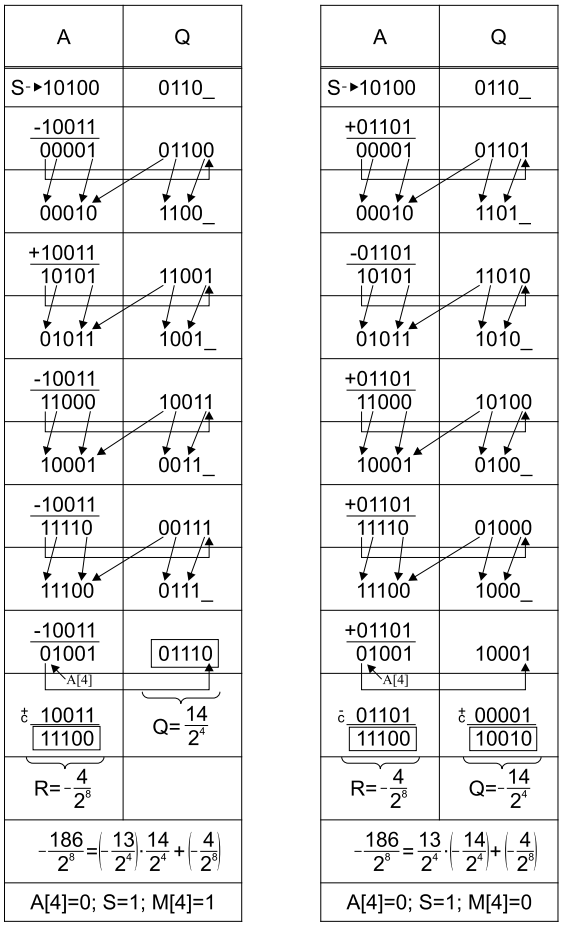
In a similar way to that which has been adopted in the sequential procedure described in the previous section, we maintain the restriction that the dividend Y have 9 bits at the most, of which, now, the msb represents the sign. As regards the sign, it is maintained during the entire procedure in the flip-flop denoted by S, it being finally necessary in establishing the correction. The signs of the partial remainders are stored in the rank A[4], whose initial value coincides with that of S, while that of the divisor is in M[4]. As a function of the signs of the two operands, an aspect checked by TEST1, the operation begins with a trial subtraction (indicated by SUBTRACT) when the signs are equal, and with a trial addition (indicated by ADD) when the signs differ.

```
                divider 2
                declare register A[4:0], Q[4:0], M[4:0], COUNT[2:0], S;
                declare bus INBUS[4:0], OUTBUS[4:0];
        BEGIN: COUNT:=0,
        INPUT: A:=INBUS, S=INBUS[4];
                Q[4:1]:=INBUS[4:1];
                M:=INBUS;
        TEST1: if (A[4]ex-norM[4])=1 then go to SUBSTRACT,
          ADD: A:=A+M, go to TEST2;
   SUBTRACT: A:=A-M;
       TEST 2: if (A[4]ex-orM[4])=1 then go to ALT,
                  begin
                    Q[0]:=1,
                    if COUNT4=1 then go to CORRECTION; else
                      begin COUNT:=COUNT+1,A.Q[4:1]:=A[3:0].Q; end
                    A:=A-M,go to TEST2;
                  end
           ALT:   begin
                    Q[0]:=0,
                    if COUNT4=1 then go to CORRECTION; else
                      begin COUNT:=COUNT+1,A.Q[4:1]:=A[3:0].Q; end
                    A:=A+M,go to TEST2;
                  end
CORRECTION: if (S ex-or M[4])=1 then Q:=Q+0.0001,
                  if ((A[4] and S and M[4]) or (A[4] and S and M[4]))=1
                    then A:=A+M,
                  if ((A[4] and S and M[4]) or (A[4] and S and M[4]))=1
                    then A:=A-M;
     OUTPUT: OUTBUS:=Q;
                  OUTBUS:=A;
        END:
```

Fig. 4.9 Description of the division procedure for signed fractional binary numbers corresponding to a sequential implementation

Subsequently, the decision to assign the value for the current bit of the quotient is taken as a function of the signs' coincidence, on the one hand of the current partial remainder, and, on the other hand of the divisor, a checking operation provided by the label TEST 2. The two subsequences corresponding to the assigning of a 1 to the bit Q[0], and to the assigning of the alternative value (indicated by the label ALT) together with the successive microoperations, are identical to those of the procedure described in Fig. 4.6. Obviously, the iterations counter COUNT is tested for its limit status COUNT4 and not COUNT7.

An essential difference of the sequential procedure in Fig. 4.9, as compared to that from Fig. 4.6, relates to the problem of correction, which, starting from the requirement that the sign of the final remainder R coincide with that of the dividend Y, is consequently more complicated in the new circumstances. But, to get used to this problem according to the same style of presentation followed throughout the book, let us appeal to an example, which, as a function of the operands' signs, will be run through in two edifying hypostases. Thus, let us divide $Y = -186 \cdot 2^{-8} = 101000110_2$, by $X = -13 \cdot 2^{-4} = 10011_2$ (Fig. 4.10a), and then let us divide the same Y by $X = +13 \cdot 2^{-4} = 01101_2$ (Fig. 4.10b). The division's execution has been performed in the manner illustrated in Fig. 4.8, but only the contents of registers A and Q are illustrated. Both the operands and the results are represented in two's complement. At the end of each operation corrections marked by "c" are made. Thus, when COUNT4 = 1 is reached, for the computations from Fig. 4.10a, the remainder is $R = 01001_2 = +9 \cdot 2^{-8}$, but the sign A[4] = 0 does not coincide with that of Y, i.e. $S = 1$. Consequently, to make the remainder have the



a *b*

Fig. 4.10 Significant division examples to mark out the corrections needed to assure the same sign for dividend and remainder

same sign as Y, X is added, the procedure, adapted to this situation, is similar to that described in Fig. 4.6. On the other hand, in the operation from Fig. 4.10b we obtain the same remainder ($R = +9 \cdot 2^{-8}$). Consequently, A[4] = 0 does not coincide with S in this case, as well, Y being negative. This time, the correction shall be applied not only to the remainder, but also to the quotient. The latter requires an adjustment through the addition of a binary unit to its lsb. Before the correction, the identity (4.5) has the following form: $-186 \cdot 2^{-8} = (+13 \cdot 2^{-4})(-15 \cdot 2^{-4}) + (+9 \cdot 2^{-8})$, being thus fulfilled, but the sign of the remainder differs from that of Y. In order to satisfy this last requirement, X will be subtracted from the value of the remainder, but this requires a compensation, namely to add 1 to the lsb of Q to fulfill the division identity in the form $Y = X(Q + 1) + R - X$. The new form of the identity (4.5) can be seen in Fig. 4.10b together with the values of the essential bits that are involved in the correction mechanism (A[4], S and M[4]). All these, together

Fig. 4.11 Truth table corresponding to the corrections needed to assure the same sign for dividend and remainder

Inputs			Outputs		
A[4]	S	M[4]	A:= A+M	A:= A-M	Q:=Q+0.0001
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	0
1	0	1	0	1	1
1	1	0	0	0	1
1	1	1	0	0	0

with those presented in Fig. 4.10a, can be found in the truth table from Fig. 4.11 [VlPe94]. The first two output columns ($A := A + M$, and $A := A - M$) represent immediate consequences of the transposition of the CORRECTION statement from the description, given in Fig. 4.6, in the case of signed operands. The third column ($Q := Q + 0.0001$) comprises the adjustment of the quotient value, when the signs of the operands differ, either in cases such as that presented in Fig. 4.10b, or in the easily checked cases when the signs of the dividend (S) and of the final remainder (A[4]) coincide.

Starting from the description given in Fig. 4.9, let us configure the combinational array structure by using the so-called division D cells as "building blocks", the diagram at the logic gate level and the suggested representation symbol for one of them being given in Fig. 4.12. The primary condition in the synthesis of D consists of the requirement that it can be used both in construction of an adder, and in that of a subtracter. Consequently, we shall start from a full adder cell (FAC) to which there will be added an EX-OR gate controlled through the input $Q^*[j]$, so that, when $Q^*[j] = 0$, the D cell is a genuine FAC, and, when $Q^*[j] = 1$, the D cell may be configured as a binary subtracter based on the addition of the two's complement.

The Boolean equations which stand at the basis of D synthesis are given by the following:

$$R_{n-j+1}[i] = \big(R_{n-j}[i-1] \oplus c[i] \oplus \big(M[i] \oplus Q^*[j]\big)\big) \tag{4.6}$$

$$c[i+1] = R_{n-j}[i-1]c[i] \ \underline{or}\ c[i]\big(M[i] \oplus Q^*[j]\big) \ \underline{or}\ R_{n-j}[i-1]\big(M[i] \oplus Q^*[j]\big) \tag{4.7}$$

where by $R_{n-j}[i-1]$ and $R_{n-j+1}[i]$ two of the current and next remainder bits have been denoted, and by $c[i]$ and $c[i+1]$ the current and next carries/borrows have been denoted, while by $M[i]$ the contents of a rank of the register where divisor X is

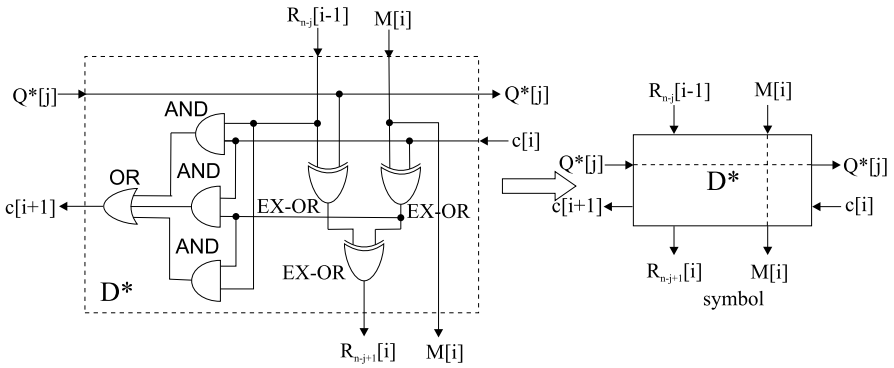


Fig. 4.12 Detailed diagram at the gate level and the symbol of a divider cell for a non-restoring combinational array division structure

loaded have been denoted, and by $Q^*[j]$ the signal which controls whether addition or subtraction operations are performed by a certain cell has been denoted [VlPe94].

As regards relations (4.6) and (4.7), certain specifications are required. The first is connected with the indexes notations that have been employed. In principle, index i is used to indicate a certain column of D cells, and index j is used to indicate a certain row of D cells. For the purpose of a unitary ordered notation of the partial remainders and of the quotient digits, by convention, index i takes values within the integer range [0, 4], increasing from right to left, and j takes values within the integer range [0, 5], decreasing from the top downward (Fig. 4.13). Moreover, we suppose that $n = 5$ and that for the inputs to the superior level we have $R_{n-j}[i-1] = A[i]$, which implies $R_0[-1] = A[0]$, as well as $R_0[3] = A[4] = S$, because, initially, as stated above, $A[4]$ and S coincide.

On the other hand, relations (4.6) and (4.7) allow the sum to be obtained when $Q^*[j] = 0$, and the difference to be obtained when $Q^*[j] = 1$. Mention should be made that the subtraction is executed by adding the two's complement, which implies the supplementary addition of 1 to the lsb cell of the subtracter. As a function of the value $Q^*[j]$, the Boolean equation (4.7) corresponds to the carry ($Q^*[j] = 0$) or to the borrow ($Q^*[j] = 1$).

These specifications being made, the combinational array structure for the non-restoring procedure is presented in Fig. 4.13, comprising, at its left side, the quotient corrector (through the addition, as applicable, of 1 to the lsb), and, in its lower part, the remainder corrector (through the addition or subtraction, as applicable, of X). The synthesis transposes exactly, by means of D cells, the description from Fig. 4.9. Thus, taking over the registers notations which are already familiar to us, we consider dividend Y to be stored in a register of 9 ranks (generally, $2n - 1$), the first 5 being denoted by A[4] to A[0], and the last 4 being denoted by Q[4] to Q[1]. Divisor X is considered to be stored in register M whose contents remain unchanged, while, as presented in Fig. 4.13, the contents of registers A and Q change: in A arrives the final remainder, and in Q arrives the quotient of the operation. This solution is

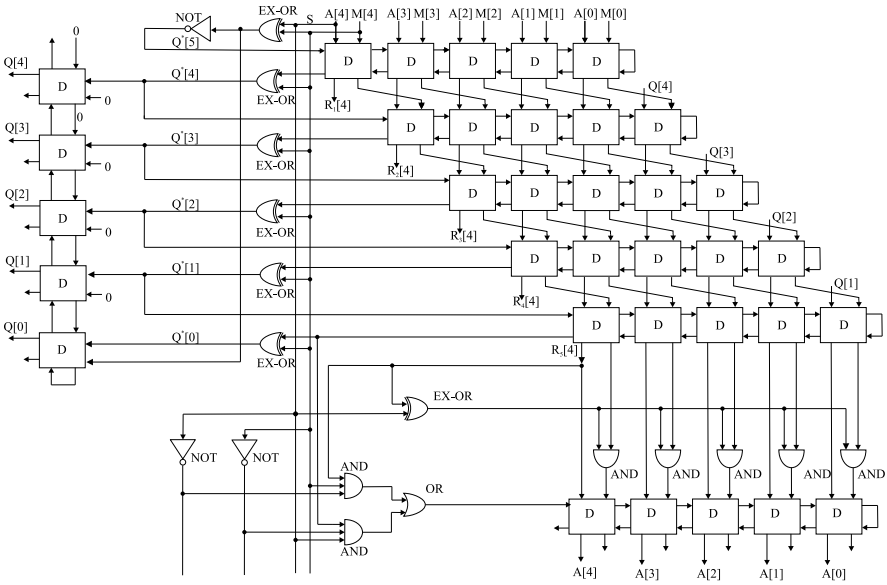


Fig. 4.13 Non-restoring combinational division structure for 5-bit signed fractions

intended to entirely use the notations from Fig. 4.9, but, obviously, A and Q may be distinct registers, which allows the conservation of the dividend [VlPe94].

Commenting upon the synthesis, mention should be made of the fact that signal $Q^*[5] = \overline{A[4] \oplus M[4]}$ corresponds to TEST1 (Fig. 4.9), namely, when it is 1, the first row of D cells is configured as a subtracter (it being also applied to the input $c[0]$ of the lsb cell), and when $Q^*[5] = 0$, the first row of D cells is configured as an adder. Subsequently, as regards the following D cells levels, we have for $Q^*[j]$ (we have denoted by "*" the corresponding uncorrected values of the binary digits $Q[j]$ of the quotient, except $Q^*[5]$, for which there is no corresponding bit $Q[5]$), according to the TEST1 statement (Fig. 4.9), the following dependence on the msb of the partial remainder, and on the sign of X:

$$Q^*[j] = \overline{R_{n-j+1}[4] \oplus M[4]} = \overline{R_{n-j+1}[4]} \oplus M[4] \tag{4.8}$$

On the other hand, as regards $R_{n-j+1}[4]$, we shall consider, for the sake of simplicity, the particular case of the msb of the first partial remainder, for which, according to (4.6), we have the following:

$$R_1[4] = A[4] \oplus c[4] \oplus \big(M[4] \oplus \overline{A[4] \oplus M[4]}\big) = \overline{c[4]} \tag{4.9}$$

Applying (4.7) to the same D cell for carry/borrow, results in:

$$\begin{aligned} c[5] &= A[4]c[4] \ \underline{or}\ c[4]\big(M[4] \oplus \overline{A[4] \oplus M[4]}\big) \ \underline{or}\ A[4]\big(M[4] \oplus \overline{A[4] \oplus M[4]}\big) \\ &= A[4]c[4] \ \underline{or}\ c[4]\overline{A[4]} \ \underline{or}\ A[4]\overline{A[4]} = c[4] \end{aligned} \tag{4.10}$$

It can be demonstrated that relations (4.9) and (4.10) are also applicable to the other stages of the array structure, and thus we have $R_{n-j+1}[4] = \overline{c[4]} = \overline{c[5]}$, which, if (4.8) is taken into account, leads us to $Q^*[4] = c[5] \oplus M[4]$. This Boolean equation justifies the layer of EX-OR gates which generates, in the most favorable way, as far as performance is concerned, the uncorrected bits of the quotient. Regarding the correcting circuit of the quotient Q, using modularity, we use five (generally n) D cells for configuring an adder, which adds, according to Fig. 4.9 and Fig. 4.11, 1 to the lsb of the quotient when $S \oplus M[4] = 1$.

As concerns the remainder-correcting circuit, between the inferior level of the array structure proper and the adder/subtracter -obtained with the same D cells- which enables the remainder corrected value to be obtained, the AND gates layer has been inserted, all such gates conditioned by the value $R_5[4] \oplus S$. Since $R_5[4]$ represents the uncorrected value of the last remainder msb, and according to Fig. 4.11, the value of divisor X passes through the AND gates only when this value has to be added or subtracted for correction. In order to configure this row of cells to perform subtraction in the appropriate cases, a circuit has been provided which implements the logic conditions from Fig. 4.9, and Fig. 4.11, when the correction is expressed by A := A − M. In this case, as well, in order to avoid the delay on a negation gate, in the synthesis the output $c[5]$ is used, i.e. $\overline{A[4]}$, corresponding to the D cell that generates the msb of the uncorrected remainder.

Referring, finally, to the performance and cost aspects of the non-restoring combinational divider from Fig. 4.13, we shall make a rough evaluation, considering the delay d on a logic gate, whatever its function. Thus, if we take into account the operations which can be executed in parallel, but without appealing to improvements of the pipeline arithmetic type, it results that the CLOCK period T has, for the general case, to exceed the value of $2(n^2 + 2n + 2)d$. In particular, for the case $n = 5$, the condition $T > 74d$ results. Otherwise, if the performance complexity is estimated by means of the specified terms, we appreciate that we have $O(n^2)$. Similarly, if cost is roughly estimated by means of the number of D cells, a complexity of $(n^2 + n)$ cells results.

4.3.2 Combinational Array Structure Based on Restoring Division

As mentioned before, we shall refer to the synthesis of a combinational array divider which enables the restoring division of integer operands without sign for which the overflow check and the divide-by-zero conditions have been tested by other means. The general character being still maintained, we shall consider that the operands' dimension is $n = 4$ bits.

We shall aim at a synthesis of the non-performing version of the restoring algorithm, which, in sequential manner, provided, at each step, the execution of a subtraction and the conservation of the partial remainder in order to avoid restoring it through addition. We shall adequately appeal to a division cell whose construction differs from that of Fig. 4.12, to enable the execution of the subtraction operation

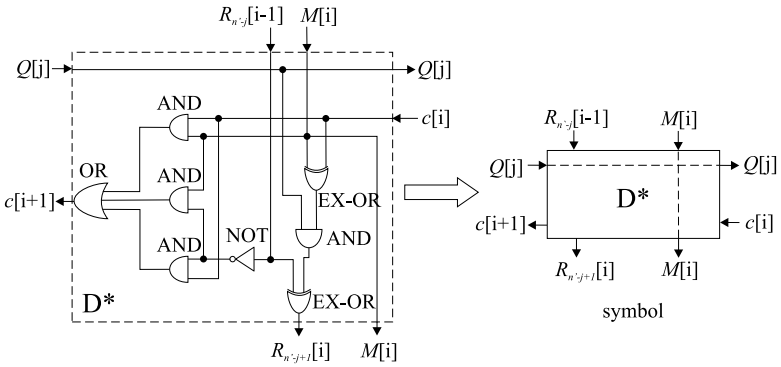


Fig. 4.14 Detailed diagram at the gate level and the symbol of a divider cell for a restoring combinational array division structure

and the inhibition of any other operation. Such a cell is presented in Fig. 4.14, which, to distinguish it from that of Fig. 4.12, will be denoted by D*, but for which we shall generally maintain the notations of the input and output variables. Mention should be made that $c[i]$ and $c[i+1]$ no longer represent carries, but only borrows. Thus, in this case, the equations homologous to (4.6) and (4.7) have the following forms:

$$R_{n'-j+1}[i] = R_{n'-j}[i-1] \oplus Q[j]\big(M[i] \oplus c[i]\big) \tag{4.11}$$

$$c[i+1] = \overline{R_{n'-j}[i-1]}M[i] \ \underline{or} \ \overline{R_{n'-j}[i-1]}c[i] \ \underline{or} \ c[i]M[i] \tag{4.12}$$

Mention should be made that unlike the relation (4.6) in which the uncorrected quotient bit $Q[j]$ intervenes, in (4.11) the variable $Q[j]$ intervenes representing a bit of the final quotient.

In relation (4.11), when $Q[j] = 0$, the current remainder becomes the next remainder, and when $Q[j] = 1$, the divisor assumed to be in register M will be subtracted from the current remainder, a case in which (4.11) together with (4.12) ensures the implementation of a binary subtracter.

Denoting the partial remainders in an ordered and unitary manner by taking over most of the notations used in Fig. 4.12, mention should be made, however, that certain modifications are required regarding Fig. 4.14. Thus, the range of integer values for index i is extended by one unit (it being, in case of Fig. 4.15, [0,4], except for the first level, where it is [0,3]) similar to the extensions by one rank of registers A and M and of the adder/subtracter from Fig. 4.7. Then, index j varies, in our case, only within the range of integers [0,3], because any bit of the quotient determines, according to (4.11), the operation executed in the given level, not in the next one, as in Fig. 4.13. The change referring to j requires, in order to maintain the already introduced conventions, a restriction of the value n, which, in this case, represents the number of ranks diminished by one unit, i.e. for Fig. 4.15, $n' = 3$, where n' substitutes n in the notations from the previous section.

Besides the specifications regarding the notations, we also mention certain differences of the structure from Fig. 4.15 (adaptations according to [Haye98]) as com-

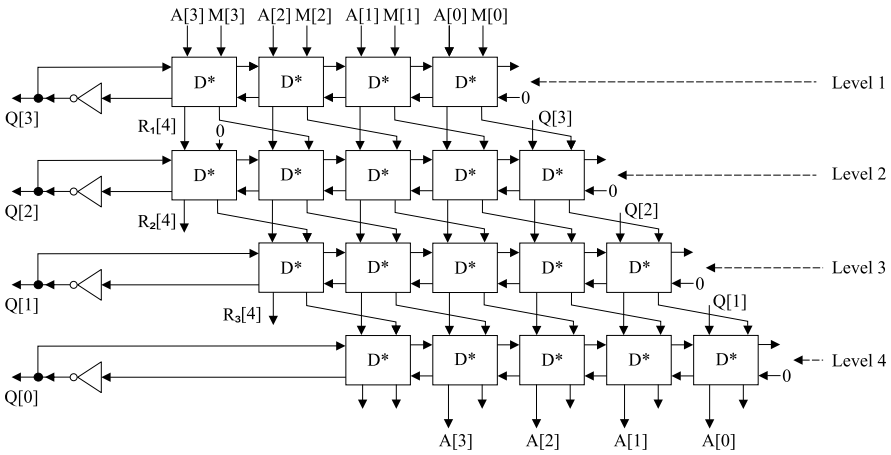


Fig. 4.15 Restoring combinational division structure for 4-bit unsigned integers

pared to that from Fig. 4.13, some of which have already been presented regarding the modifications of the indexes' values. Moreover, in Fig. 4.15, the quotient and remainder corrections are missing, the restoring algorithm not requiring any correction. Condition $c[0] = Q[j]$ is also missing from the structure given in Fig. 4.13, it being substituted by $c[0] = 0$, because in Fig. 4.15 the subtraction is executed directly and not through the addition of two's complement. Mention should also be made that the quotient bits represent, as expected, the negated "sign" binary digits, according to each logic level.

For instance, let us divide $Y = 72_{10} = 1001000_2$ by $X = 13_{10} = 1101_2$. First of all, the overflow check condition shows that $72 < 13 \cdot 15 + 12 = 207$ where, by adopting the representation on 4 bits, it results that values 15(1111), and 12 ($12 < X = 13$) represent the largest possible ones for the quotient, and for the remainder respectively. Figure 4.16 presents the binary configurations corresponding to the various levels of the division cells D* of the structure from Fig. 4.15. The partial remainders from the levels' outputs are given before (marked by "Level i") and after (marked by "Level i'") the signal stabilization. There can be observed the borrows generation ("Borrow" column) from the msb ranks of the partial remainders which establish the values of the quotient bits ($Q[j]$). There can also be seen the right-shift, at each level, of the divisor (marked by M). Finally, the results (the quotient and the remainder) are obtained which lead to the satisfaction of the division identity.

In this case, as well, we make rough quantitative estimates as regards performance and cost aspects, similar to those for the previous array structure. If the operations which are executed in parallel are taken into account, and the hypothesis according to which all the logic gates have the same delay, the CLOCK period T results, which has to exceed the value (for the general case of the same number n of the quotient and remainder ranks) of $2(n^2 + 3n - 1)d$. In particular, for the case $n = 4$, the condition $T > 54d$ results. Otherwise, if performance is estimated by

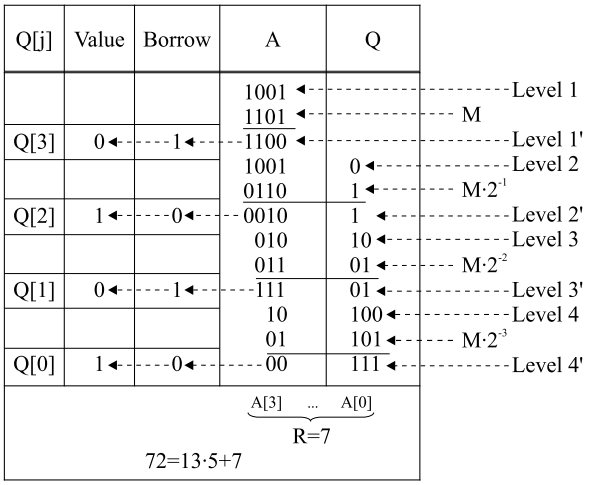


Fig. 4.16 Binary division example for unsigned integers using a restoring combinational array division structure

means of the already specified terms, we have a complexity which is comparable with that obtained for the non-restoring structure.

On the other hand, if cost is estimated, in the same rough manner, by taking into account the number of division cells D* and by generalizing the configuration from Fig. 4.15 for an n bit quotient and remainder, a synthesis results which requires $(n + (n + 1)(n - 1))$ cells, i.e. a complexity of $(n^2 + n - 1)$ cells.

4.4 SRT Procedures for Binary Division

4.4.1 Radix 2 SRT Procedure

As regards non-restoring division, at each step of the algorithm, a subtraction or addition operation is executed, as applicable. Let us select the quotient digits from a set $\{1, \overline{1}\}$, where 1 corresponds to a subtraction, and $\overline{1}$ corresponds to an addition. Under these circumstances, and temporarily ignoring the problem of corrections as regards fractional operands (according to Fig. 4.9), and taking into account the differences pointed out in (4.5), a recurrence relation of the type given by (4.3) can be applied. Using this relation, the quotient obtained with digits from the set $\{1, \overline{1}\}$ has to be transformed into a conventional value with digits belonging to the set $\{0, 1\}$. Let us analyse how the quotient conversion is done from the signed digit form, expressed with digits $q_i^* \in \{1, \overline{1}\}$, into the conventional form, expressed with binary digits $q_i \in \{0, 1\}$, when the latter is given by the two's complement. Consequently, we can see, at first, that between the digits q_i^* and q_i there is the obvious relation $q_i^* = 2q_i - 1$, and if quotient Q is assumed to be an integer, the signed digit form of

Q may undergo the following transformations:

$$Q = q_{n-1}^* q_{n-2}^* \dots q_i^* \dots q_1^* q_0^* = \sum_{i=0}^{n-1} q_i^* 2^i = \sum_{i=0}^{n-1} (2q_i - 1)2^i = 2\sum_{i=0}^{n-1} q_i 2^i - \sum_{i=0}^{n-1} 2^i \tag{4.13}$$

If in (4.13) we take into account the identity $\sum_{i=0}^{n-1} 2^i = 2^n - 1$, and rearrange the terms, then the following will be obtained:

$$Q = \sum_{i=0}^{n-1} q_i 2^{i+1} - 2^n + 1 = -2^n + q_{n-1}2^n + \sum_{i=0}^{n-2} q_i 2^{i+1} + 1$$

$$= -(1 - q_{n-1})2^n + \sum_{i=0}^{n-2} q_i 2^{i+1} + 1 \tag{4.14}$$

If we take into account that $1 - q_{n-1} = \overline{q_{n-1}}$, as well as relation (3.5) about the expression of an integer in two's complement, (4.14) may be written in the following form [Parh00]:

$$Q = \overline{q_{n-1}} q_{n-2} \dots q_i \dots q_1 q_0 1 \tag{4.15}$$

According to (4.15), quotient Q can be conventionally obtained by substituting, first of all, each value of $\overline{1}$ by one of 0, and then by complementing the msb of Q, and finally attaching a 1 bit to the right side of the lsb. Thus, the signed digit form of the quotient being obtained, the conversion to the conventional form will be simple.

On the other hand, an alternative to non-restoring division is the algorithm where for the 0 quotient digits only shifts (without any other operation) are executed, a procedure known as "the non-restoring division with shifting over 0s" [HePa03, Parh00, ErLa04]. The method is based on the observation that when the values of $2R_i$ belong to the interval $[-X, +X)$ (the value $(-X)$ corresponds to the next partial remainder for which $R_{i+1} = 0$), by adding or subtracting X, the sign of the next partial remainder changes anyway. In this case, for the current digit of the quotient (q_{n-1-i}, from relation (4.3)), we can choose value 0, a situation which requires only the shift of the partial remainder, without executing any other operation. Otherwise, if $0 \le 2R_i < 2X$, then $q_{n-1-i} = 1$ and $R_{i+1} := 2R_i - X$, similarly, if $(-2X) \le 2R_i < 0$, then $q_{n-1-i} = \overline{1}$ and $R_{i+1} := 2R_i + X$, and, finally, if $(-X) \le 2R_i < (+X)$, then $q_{n-1-i} = 0$, and no operation (NOP) will be executed. The Robertson diagram for the situation when the quotient is represented in signed-digit form is given in Fig. 4.17 [Parh00], where can be observed the overlapping intervals as regards the choosing of the quotient digits, making this process a redundant one. Thus, for a certain overlapping region, it does not matter which digit is chosen for the quotient ($\overline{1}$ or 0, and 0 or 1). This fact is a favorable one, because an "error" made in choosing is subsequently corrected, it being, therefore, tolerated [KoMu06]. Consequently, the comparisons provided by the rules for the selection of the quotient digits may be approximated, their exact, precise execution not being necessary. The previously mentioned tolerance regarding the choice of the quotient

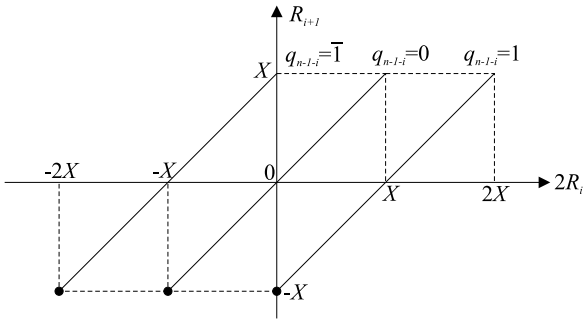


Fig. 4.17 Robertson diagram for the non-restoring division with a signed-digit quotient

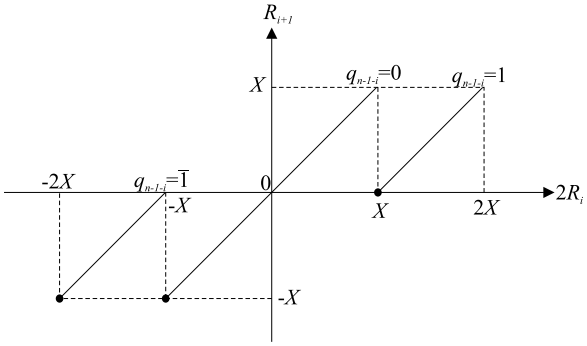


Fig. 4.18 Robertson diagram for the non-restoring division with shifting over 0s

digit urges the priority selection of digit 0 for the overlapped intervals, because, in this case, as shown before, it is not necessary to execute any arithmetic operation, except the shift. Due to this selection criterion, the above-described procedure may be changed, namely, if $(-X) \leq 2R_i < (+X)$, then, in the same way as before, $q_{n-1-i} = 0$ and NOP, but the setting of $q_{n-1-i} = 1$ and the subsequent subtraction $R_{i+1} := 2R_i - X$ take place only when $X \leq 2R_i < 2X$, while the setting of $q_{n-1-i} = \overline{1}$ and the subsequent addition $R_{i+1} := 2R_i + X$ take place only when $(-2X) \leq 2R_i < (-X)$. The new situation correspond to the non-restoring division with shifting over 0s procedure whose appropriately modified Robertson diagram is presented in Fig. 4.18 [Parh00], where changes that have been made as compared to the diagram corresponding to the conventional non-restoring procedure (Fig. 4.5b) can clearly be observed.

As has been observed in the above-presented method, certain steps consist only of shifts, a fact which can be employed in an asynchronous design option, which may take advantage of the fact that the adder is only selectively activated. Taking advantage of the steps in which the adder is not involved (concerning both addition and subtraction), an implementation version characterized by an average division speed superior to that of the non-restoring method can be obtained. There is, however, a major problem, namely to find out whether the value of the shifted partial remainder does or does not belong to the interval $[-X, +X)$. A possible solution consists in making trial subtractions, but the time required by these operations may

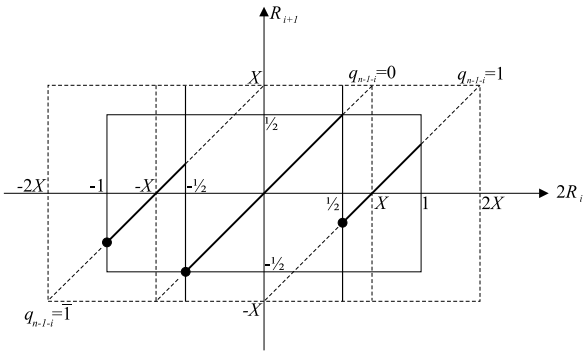


Fig. 4.19 Robertson diagram corresponding to the quotient digit selection for the SRT division

exceed the time saved by avoiding the arithmetic operations when $2R_i \in [-X, +X)$. Consequently another solution is needed, and it has been offered, independently, by D. Sweeney, J.E. Robertson and K.D. Tocher, in honour of whom the resultant algorithm has been called SRT [ErLa04, Parh00].

The SRT procedure will be presented, the general character being still maintained, by adopting certain hypotheses. Thus, we consider the operands to be subunitary fractional numbers, because the treatment of integer operands may be reduced to that which is specific for fractions, by applying appropriate scaling [Omon94]. Consequently, Y would belong to the interval $[-1, +1)$, but, to avoid quotient overflow, we proceed by restricting the value range for Y, i.e. $Y \in [-X, +X)$. Moreover, divisor X will be considered a normalized fractional number, according to the model of floating point operation, i.e. when taking into account its absolute value its msb is 1, which implies $|X| \geq 1/2$. In order to fulfill this hypothesis, an initial shift of X might be necessary, which requires additional shifts, namely, an initial shift of Y, as well as the final, recurrent, shift of R.

These stipulations being made, we shall reconsider now the problem of precise comparisons, on the entire length of the words, between $2R_i$ and X, and $(-X)$. But, due to the overlapping shown in the Robertson diagram from Fig. 4.17, precise comparisons are not strictly required, and thus, instead of making comparisons with reference to X and $(-X)$ they can be made regarding other values, which we call X^* and $(-X^*)$. Since $|X|$ is normalized, having the minimum value $(+1/2)$, the SRT algorithm, in standard form [Korn03, Parh00], assumes the value $X^* = +1/2$. Initially, the partial remainder represented by Y, which has been supposed to belong to the value range $[-X, +X)$, may not belong to the interval $[-1/2, +1/2)$, which requires the shift of Y by one position to the right. To compensate this initial shift, finally, the doubling of the quotient and the remainder will be required.

As mentioned before, the algorithm is based on the fact that, when $(+1/2) \leq 2R_i < +1$, we have $q_{n-1-i} = 1$ and $R_{i+1} = 2R_i - X$, similarly, when $(-1) \leq 2R_i < (-1/2)$, we have $q_{n-1-i} = \overline{1}$ and $R_{i+1} = 2R_i + X$, and when $(-1/2) \leq 2R_i < (+1/2)$, we have $q_{n-1-i} = 0$ and $R_{i+1} = 2R_i$ (NOP). Figure 4.19 [Parh00] presents the Robertson diagram corresponding to the new selection rules. It can be observed that over the dependences of the quotient digits from Fig. 4.18, marked with a broken line in Fig. 4.19, are superposed the variations corresponding to

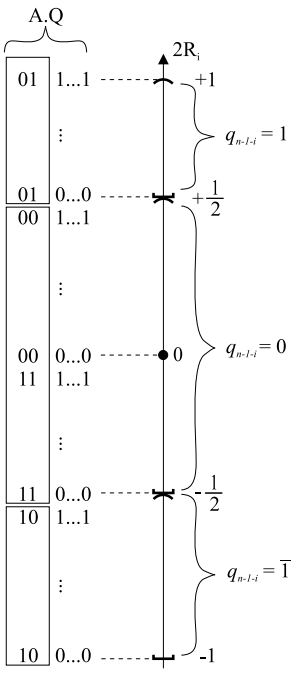


Fig. 4.20 Correspondence between the A.Q register contents and the values on the shifted partial remainders axis for the SRT procedure

the new procedure, marked with a solid line. Once the initial partial remainder Y has been adjusted in such a way (through rightshift) as to belong to the interval $[-1/2, +1/2)$, all the following partial remainders can be maintained within the value range $[-1/2, +1/2)$, which is delimited by the rectangle traced with a solid line in Fig 4.19.

Reverting to the approach introduced for the sequential device from Fig. 4.7, the remainder R_i is stored in the double register A.Q, its dimension being restricted by one bit at each step, so that the final remainder takes up only register A. Thus, in Fig. 4.20 are represented the contents corresponding to the double register A.Q and the axis of shifted partial remainders $(2R_i)$ are represented by the marking of the significand values (negatives values are represented in two's complement).

There are also delimited the value intervals associated with the current bit of the quotient, and which can be identified from the two most significant bits of A, the msb corresponding to the sign. Thus, when the given doublet is 01, we have $q_{n-1-i} = 1$, and when it is 10, we have $q_{n-1-i} = \overline{1}$; similarly, if the two bits are identical (00 or 11), we have $q_{n-1-i} = 0$.

Figure 4.21 presents, in the adopted hardware description language, and by using the same structure elements from Fig. 4.7, the code sequence corresponding to the description of one of the possible implementations, in sequential version, of the SRT algorithm. We shall only comment upon the differences as compared to what has already been presented regarding the conventional non-restoring algorithm. As can be seen from the very beginning, a companion counter (COUNTC) of the well-known iteration counter (COUNT) exists. The presence of COUNTC is required by

```
            divider3
            declare register A[7:0], Q[7:0], M[7:0], COUNT[3:0], COUNTC[2:0];
            declare bus INBUS[7:0], OUTBUS[7:0];
      BEGIN: COUNT :=0, COUNTC:=0,
      INPUT: A[7:0]:=INBUS[7:0];
             Q[7:0]:=INBUS[7:0];
             M[7:0]:=INBUS[7:0];
NORMALIZE: if M[7]=1 then go to TEST1,
             M[7:1]:=M[6:0], A.Q[7:1]:=A[6:0].Q, COUNTC:=COUNTC+1, go to NORMALIZE;
      TEST1: if (A[7] ex-nor A[6])=1 then
              A.Q[7:1]:=A[6:0].Q, COUNT:=COUNT+1, Q*[0]:=0, go to TEST2;
             if A[7]=1 then
               begin
                A.Q[7:1]:=A[6:0].Q, COUNT:=COUNT+1, Q*[0]:=1̄;
                A:=A+M, go to TEST2;
               end
             else
               begin
                A.Q[7:1]:=A[6:0].Q, COUNT:=COUNT+1, Q*[0]:=1;
                A:=A-M;
               end
      TEST2: if COUNT8≠1 then go to TEST1,
CORRECTION: if A[7]=1 then A:=A+M, Q:=Q-0.00...01;
      TEST3: if COUNTC≠0 then A[6:0]:=A[7:1], COUNTC:=COUNTC-1, go to TEST3;
     OUTPUT: OUTBUS:=Q;
             OUTBUS:=A;
        END:
```

Fig. 4.21 Description of the SRT division in one of the possible sequential implementation versions

the normalization process of divisor X, because the number of bits by which X and Y are shifted to the left has to be counted. This is the task of COUNTC, which counts the leading 0s eliminated during the leftshift process, until M[7] becomes 1. The content of COUNTC is needed at the end of the procedure, because the last remainder has to be shifted by the same number of bits, this time to the right, an operation indicated by label TEST3. Thus, COUNTC shall be provided with capacity both to increment and decrement of its contents, and register A shall be provided with bidirectional shifting capacity. Within the normalization operation, register Q (together with A) needs to be able to be left-shifted (in a similar way as in the device from Fig. 4.7), but register M, whose contents have been fixed in the previous approaches, shall also have the left-shifting capacity.

Before commenting upon the algorithm proper, which begins at TEST1, mention should be made that in Fig. 4.21 no reference is made regarding the overflow or divide-by-0 condition checking, which are supposed to be previously solved through adequate software routines. The description of the procedure is precisely based on Fig. 4.20. Thus, when the most significant two bits from A are identical, which is tested through the coincidence function of A[7] and A[6], the current bit of the quotient shall be set to 0.

We also add that, if the two bits A[7] and A[6] are different, the current bit of the quotient will be either 1 or $\overline{1}$, but the three values (0, 1 or $\overline{1}$) cannot be stored in a flip-flop, which is why in Fig. 4.21 we have set up a fictitious rank Q[0], denoted by

Q*[0]. The conversion from the signed digit form into the conventional binary one cannot be made by applying the solution based on the relation (4.15), because, now, instead of two digits with sign $(1, \overline{1})$, we have three $(0, 1, \overline{1})$. A possible solution to this problem consists in using two quotient Q registers where 1 bits are loaded in one of the registers in the positions corresponding to their occurrence in the quotient, while in the other register 1 bits corresponding to the occurrence in the quotient of $\overline{1}$ digits are loaded. Finally, after all the quotient bits have been determined, but before the result is corrected, indicated by the CORRECTION statement, the binary version of quotient Q will be obtained by subtracting the contents of the two registers where the positive and negative digits have been stored. Thus, according to this solution for signed digit-binary conversion, we have a supplementary activation of the adder/subtracter, which increases the total latency of the execution. This situation can be avoided by appealing to an algorithm which performs the conversion in a serial manner, as the quotient digits are generated, the final subtraction operation being unnecessary. This algorithm has been suggestively called an "on-the-fly" conversion algorithm [ErLa04], and we shall discuss it later. Finally, as regards the procedure proper, mention should be made that the distinction between the two doublets from the most significant two ranks of A, by means of which there is chosen the current digit of the quotient among $q_{n-1-i} = 1$ and $q_{n-1-i} = \overline{1}$, can be made using the sign bit A[7] of the partial remainder.

We also refer to the statement labeled CORRECTION, which has to be appealed when the last remainder is negative, i.e. having a sign opposed to that of the dividend, a step also met with in the conventional non-restoring procedure (Fig. 4.6). But now, besides the supplementary addition of X, the compensatory adjustment of the quotient value Q is also provided, by subtracting from it a binary unit (in the lsb position). In a manner similar to the correction recommended in the algorithm from Fig. 4.9, we have $Y = X(Q-1) + (R+X)$.

We shall refer below to the on-the-fly conversion problem [ErLa04]. Thus, if we consider a subunitary fractional quotient, and maintain the conventional notation of the indexes associated with its digits, for the signed digit vector $Q(j-1)$ made up of the most significant $(j-1)$ bits of Q, except the sign bit, there results:

$$Q(j-1) = q^*_{n-2}2^{-1} + q^*_{n-3}2^{-2} + \cdots + q^*_{n-j}2^{-(j-1)} = \sum_{i=1}^{j-1} q^*_{n-1-i}2^{-i} \quad (4.16)$$

where q^*_{n-i-1} represents the ith digit with sign of the quotient, the counting beginning from the left side with 1.

Let us consider now the next jth digit with sign q^*_{n-1-j}, and let us add it to $Q(j-1)$, then, according to its value, the following will be obtained:

$$Q(j) = \begin{cases} Q(j-1) + q^*_{n-1-j}2^{-j} & \underline{\text{if}}\ q^*_{n-1-j} = 0\ \underline{or}\ 1 \\ Q(j-1) - 2^{-(j-1)} + (2 - |q^*_{n-1-j}|)2^{-j} & \underline{\text{if}}\ q^*_{n-1-j} = \overline{1} \end{cases} \quad (4.17)$$

where the bars $||$ signify the absolute value.

But in (4.17), the subtraction $Q(j-1)-2^{-(j-1)}$ occurs, which may require the propagation of a borrow, it being, consequently, slow. This operation can be avoided by updating, for each digit of the quotient, a value representing the difference $Q_D(j-1)=Q(j-1)-2^{-(j-1)}$, an update which is made by taking into account (4.17) and the potential values of q^*_{n-1-j}, by means of the following operations:

$$\begin{aligned} Q_D(j) &= Q(j)-2^{-j} \\ &= \begin{cases} Q(j-1)+(q^*_{n-1-j}-1)2^{-j} & \underline{\text{if}}\ q^*_{n-1-j}=1 \\ Q_D(j-1)+(1-|q^*_{n-1-j}|)2^{-j} & \underline{\text{if}}\ q^*_{n-1-j}=\overline{1}\ \underline{or}\ 0 \end{cases} \end{aligned} \tag{4.18}$$

Starting from (4.17), the signed digit-binary conversion implies the execution of the computations given by:

$$Q(j)=\begin{cases} Q(j-1)+q^*_{n-1-j}2^{-j} & \underline{\text{if}}\ q^*_{n-1-j}=0\ \underline{or}\ 1 \\ Q_D(j-1)+(2-|q^*_{n-1-j}|)2^{-j} & \underline{\text{if}}\ q^*_{n-1-j}=\overline{1} \end{cases} \tag{4.19}$$

But the additions from (4.19) represent only concatenations, so that the time costly carry and borrow propagations are avoided. In terms of concatenations, the conversion algorithm is as follows:

$$Q(j)=\begin{cases} (Q(j-1),q^*_{n-1-j}) & \underline{\text{if}}\ q^*_{n-1-j}=0\ \underline{or}\ 1 \\ (Q_D(j-1),(2-|q^*_{n-1-j}|)) & \underline{\text{if}}\ q^*_{n-1-j}=\overline{1} \end{cases} \tag{4.20}$$

and

$$Q_D(j)=\begin{cases} (Q(j-1),(q^*_{n-1-j}-1)) & \underline{\text{if}}\ q^*_{n-1-j}=1 \\ (Q_D(j-1),(1-|q^*_{n-1-j}|)) & \underline{\text{if}}\ q^*_{n-1-j}=\overline{1}\ \underline{or}\ 0 \end{cases} \tag{4.21}$$

We also assume the initial conditions $Q(0)=Q_D(0)=0$ for a positive quotient. Figure 4.22 exemplifies the on-the-fly conversion for the particular case of the signed-digit quotient $Q^*=011\overline{1}0\overline{1}10=2^{-1}+2^{-2}-2^{-3}-2^{-5}+2^{-6}=01001110_2$.

Regarding the conversion implementation, it requires two registers, which will be denoted Q and Q_D, provided with leftshift capacity (Fig. 4.23), different from register Q from Fig. 4.21. In a possible solution, we suppose that the statements of $\underline{\text{begin}}$ $Q^*[0]=0, 1$ or $\overline{1}$ type from Fig. 4.21 exclusively determine the value of one of the flag flip-flops A for 0, B for 1, and C for $\overline{1}$ (a fact suggested by the controls on the asynchronous inputs of set and reset of the three flags, Fig. 4.23). Each of the registers Q and Q_D has a special multiplexing circuit attached to it which enables the loading (as applicable) of the contents of the two registers shifted to the left by one rank. Without loss of generality, the registers have been considered of eight ranks each, in the lsb positions the binary values which correspond to relations (4.19) and (4.18) being loaded. Thus, according to (4.19), in Q[0] is introduced q^*_{n-1-j} if this

Fig. 4.22 Quotient conversion example from signed-digit form into the binary conventional form by the on-the-fly procedure

j	q^*_{n-j-1}	$Q(j)$	$Q_D(j)$
0		0	0
1	1	0.1	0.0
2	1	0.11	0.10
3	$\overline{1}$	0.101	0.100
4	0	0.1010	0.1001
5	$\overline{1}$	0.10011	0.10010
6	1	0.100111	0.100110
7	0	0.1001110	0.1001101

bit is 0 or 1, and $(2 - |q^*_{n-1-j}|)$, if $q^*_{n-1-j} = \overline{1}$, when Q[0] is set to $2 - |\overline{1}| = 1$. Because when $q^*_{n-1-i} = 0$, Q[0] must be set to logical 0, and when q^*_{n-1-i} is 1 or $\overline{1}$, Q[0] must be set to at logical 1, it results that in the selection of a signed digit, the given input of the lsb rank of register Q is controlled by Q[0]= B *or* C. Similarly, in Q_D[0] there is introduced $(q^*_{n-1-j} - 1)$, if $q^*_{n-1-j} = 1$, rank Q_D[0] being set to $1 - 1 = 0$, the same way as in Q_D[0] there is introduced value $(1 - |q^*_{n-1-j}|)$, if q^*_{n-1-j} is $\overline{1}$ or 0, rank Q_D[0] being set to $1 - 0 = 1$, and on $1 - |\overline{1}| = 0$. Taking into account all these situations, for the lsb rank input of register Q_D, there results $Q_D[0] = A$.

As regards the validations of the parallel loadings in the two registers, with reference to Q, and based on relations (4.19) and (4.20), the contents of this register are shifted to the left when q^*_{n-1-j} is 0 or 1, namely when (A *or* B = 1), as it is loaded with the shifted contents from register Q_D when $q^*_{n-1-j} = \overline{1}$, i.e. when the variable C = 1 is set. Similarly, in Q_D, based on relations (4.18) and (4.21), the shifted contents from register Q is loaded when $q^*_{n-1-j} = 1$, i.e. when variable B = 1 is set, the same way as the shifted contents from the same register Q_D is loaded when q^*_{n-1-j} is $\overline{1}$ or 0, i.e. when (A *or* C = 1).

The diagram from Fig. 4.23 can easily be combined with the description from Fig. 4.21, resulting in a high-performance implementation which implies, for the quotient conversion, a worst delay of three logic gates at the most, as well as the set up of two flip-flops, at each digit of the quotient, avoiding the final subtraction whose borrow propagation may consistently degrade the performance of the entire operation.

These statements being made, let us consider the example from Fig. 4.24 in which we refer to the division of the binary equivalents corresponding to dividend $Y = 535_{10}$ by divisor $X = 7_{10}$, an operation after which quotient $Q = 19_{10}$ and remainder $R = 3_{10}$ result. Mention should be made that taking into account the restrictive conditions regarding the operands, we have $Y = 535/2^{16}$ and $X = 7/2^7$, so that the procedure starts with $Y \in [(-1/2), (+1/2))$ and $X \in [(+1/2), (+1))$. The

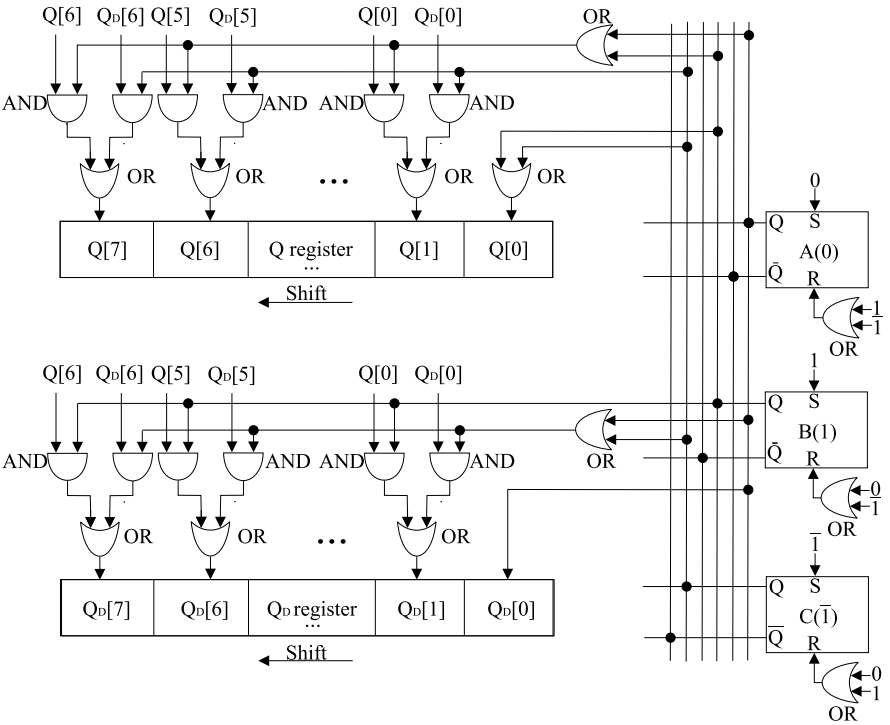


Fig. 4.23 Block diagram of the on-the-fly quotient conversion implementation from the signed-digit form into the binary conventional form

assurance of the initial conditions has implied the left-shift of the three registers A.Q and M by 4 bits, and, finally, the right-shift of the remainder from register A by the same number of bits.

Before analyzing the specific problems of the SRT radix 4 procedure, end our examination of the SRT radix 2 method by comparing it with the common version of the non-restoring method (Fig. 4.6 and Fig. 4.7), pointing out the following differences:

(a) Since the decision regarding the value of the quotient bit is taken according to the sign of the partial remainder, in the non-restoring method, registers A, M and, consequently the adder/subtracter, contain one more rank as compared to the same structural elements from the device which implements the SRT radix 2 procedure.
(b) The value of the current bit of the quotient is chosen, in the non-restoring procedure, as a function of the value of one bit (the sign bit), while in the SRT radix 2 procedure it is necessary to investigate the value of the most significant two bits corresponding to the partial remainders.
(c) As regards the final quotient, in the non-restoring procedure, it results directly, by chaining the values that have resulted by means of the signs of the partial re-

Fig. 4.24 SRT binary division example

A	Q	M
00000010	00010111	00001110
00100001	01110000	11100000
01000010	11100000	
10000101 -11100000 10100101	11000001	
01001011 +11100000 00101011	10000$1\bar{1}$	
01010111	00001$\bar{1}$0	
10101110 -11100000 11001110	0001$\bar{1}$01	
10011100	001$\bar{1}$010	
00111000 +11100000 00011000	01$\bar{1}$0101	
00110000	0$\bar{1}$01010	
00000011 $=\frac{3}{2^{16}}$	00100110 $=\frac{19}{2^7}$	

$$\frac{535}{2^{16}} = \frac{7}{2^7} \cdot \frac{19}{2^7} + \frac{3}{2^{16}} = \frac{133}{2^{14}} + \frac{3}{2^{16}}$$

mainders, while in the SRT radix 2 procedure extra work is necessary in the conversion of the signed digit form into the conventional binary one, either in "on-the-fly" manner, or by appealing to a supplementary activation of the adder/subtracter.

(d) Supposing that an asynchronous design style is considered [Poll90] which takes advantage, as regards time, of the adder/subtracter non-activation, the SRT radix 2 procedure is the more rapid than the non-restoring one, the more the particular binary configurations of the operands generate a large number of 0 bits for the quotient.

4.4.2 Radix 4 SRT Procedure

In order to arrive at the problems that are specific to the SRT radix 4 procedure, we shall first appeal to two elements. The first consists of the diagram known as the P-D plot [ErLa04, Parh03, Kore93], and sometimes as the PR-D plot [Omon94]. This represents a graphical means of investigating the process of the selection of quotient digits, being made up of a rectangular system of axes with divisor D associated with

the abscissa, and the shifted partial remainder P associated with the ordinate. The P-D plot [Korn05] is useful when, in the choice of the current quotient bit q_{n-1-i}, there are alternative values (as there exist, for instance, in Fig. 4.17) which means that the option for a certain value does not require full precision comparisons, but only low precision ones.

Thus, let us suppose that we want to perform the division operation in a number system with the general radix r, a case in which, by adapting (4.3), the following recurrence relation will be obtained:

$$R_{i+1} := r\,R_i - q_{n-1-i}\,X \tag{4.22}$$

On the other hand, as regards the quotient digits, when signed digits have been used in the conventional non-restoring method with $r = 2$, we had $q_{n-1-i} = \overline{1}$ when $(-2X) \leq 2R_i < 0$, and $q_{n-1-i} = 1$ when $0 \leq 2R_i < 2X$, which, by extension, when the radix is r, become:

$$q_{n-1-i} = \begin{cases} r-1 & \text{if } r\,R_i \in [(r-2)X, rX) \\ \quad\vdots & \\ 2 & \text{if } r\,R_i \in [X, 3X) \\ 1 & \text{if } r\,R_i \in [0, 2X) \\ \overline{1} & \text{if } r\,R_i \in [-2X, 0) \\ \overline{2} & \text{if } r\,R_i \in [-3X, -X) \\ \quad\vdots & \\ \overline{r-1} & \text{if } r\,R_i \in [-rX, -(r-2)X) \end{cases} \tag{4.23}$$

In this way, at each step, the arithmetic operation is either a subtraction, or an addition of the r multiples depending on whether the partial remainder is positive or negative.

If we denote by q_{max} the absolute maximum value which may be taken by the quotient digit, then we have $((r-1)/2) < q_{max} \leq (r-1)$, and if 0 is added to the set of quotient digits, as is needed for the SRT procedure, then (4.23) changes into [ErLa04]:

$$q_{n-1-i} = \begin{cases} q_{max} & \text{if } r\,R_i \in [(q_{max}-1)X, (q_{max}+1)X) \\ q_{max}-1 & \text{if } r\,R_i \in [(q_{max}-2)X, q_{max}X) \\ \quad\vdots & \\ 2 & \text{if } r\,R_i \in [X, 3X) \\ 1 & \text{if } r\,R_i \in [0, 2X) \\ 0 & \text{if } r\,R_i \in [-X, X) \\ \overline{1} & \text{if } r\,R_i \in [-2X, 0) \\ \overline{2} & \text{if } r\,R_i \in [-3X, -X) \\ \quad\vdots & \\ \overline{q_{max}-1} & \text{if } r\,R_i \in [-q_{max}X, -(q_{max}-2)X) \\ \overline{q_{max}} & \text{if } r\,R_i \in [-(q_{max}+1)X, -(q_{max}-1)X) \end{cases} \tag{4.24}$$

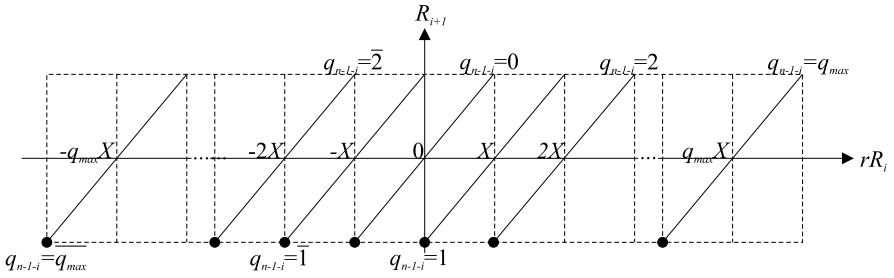


Fig. 4.25 Robertson diagram for a certain radix binary division

The Robertson diagram for (4.24) is presented in Fig. 4.25 [Parh00], where the overlapping zones corresponding to the quotient digits can be observed, giving it the already-mentioned redundant character. If, within this context, we define the redundancy factor ρ by $\rho = q_{max}/(r-1)$ [ErLa04], then $(1/2) < \rho \leq 1$, and the restriction $-\rho X \leq R_{i+1} < \rho X$ will act on the next partial remainder. Using this restriction, as well as (4.22), the range of variation tolerated for rR_i will be obtained, namely: $(-\rho + q_{n-1-i})X \leq rR_i < (\rho + q_{n-1-i})X$.

These remarks being made, the P-D plot is used to present graphically the variation zones for rR_i, with q_{n-1-i} increasing from unit to unit from the value $(-q_{max})$ to q_{max}.

For instance, let us consider the SRT radix 4 algorithm of minimum redundancy, i.e. the algorithm to which corresponds $q_{max} = 2$, having associated the Robertson diagram from Fig. 4.26, and the P-D plot from Fig. 4.27 [ErLa04, HePa03, Parh00]. Thus, on the basis of what has been presented above, for Fig. 4.26, we have $\rho = 2/3$, and then:

$$q_{n-1-i} = \begin{cases} 2 & \text{if } 4R_i \in [(-\frac{2}{3}+2)X, (+\frac{2}{3}+2)X) \\ 1 & \text{if } 4R_i \in [(-\frac{2}{3}+1)X, (+\frac{2}{3}+1)X) \\ 0 & \text{if } 4R_i \in [(-\frac{2}{3}+0)X, (+\frac{2}{3}+0)X) \\ \overline{1} & \text{if } 4R_i \in [(-\frac{2}{3}+\overline{1})X, (+\frac{2}{3}+\overline{1})X) \\ \overline{2} & \text{if } 4R_i \in [(-\frac{2}{3}+\overline{2})X, (+\frac{2}{3}+\overline{2})X) \end{cases} \tag{4.25}$$

Following the computations from (4.25), the range of variation of the remainders R_i corresponding to the various values of the digits of quotient q_{n-1-i} will be obtained:

$$q_{n-1-i} = \begin{cases} 2 & \text{if } R_i \in [+\frac{X}{3}, +\frac{2X}{3}) \\ 1 & \text{if } R_i \in [+\frac{X}{12}, +\frac{5X}{12}) \\ 0 & \text{if } R_i \in [-\frac{X}{6}, +\frac{X}{6}) \\ \overline{1} & \text{if } R_i \in [-\frac{5X}{12}, -\frac{X}{12}) \\ \overline{2} & \text{if } R_i \in [-\frac{2X}{3}, -\frac{X}{3}) \end{cases} \tag{4.26}$$

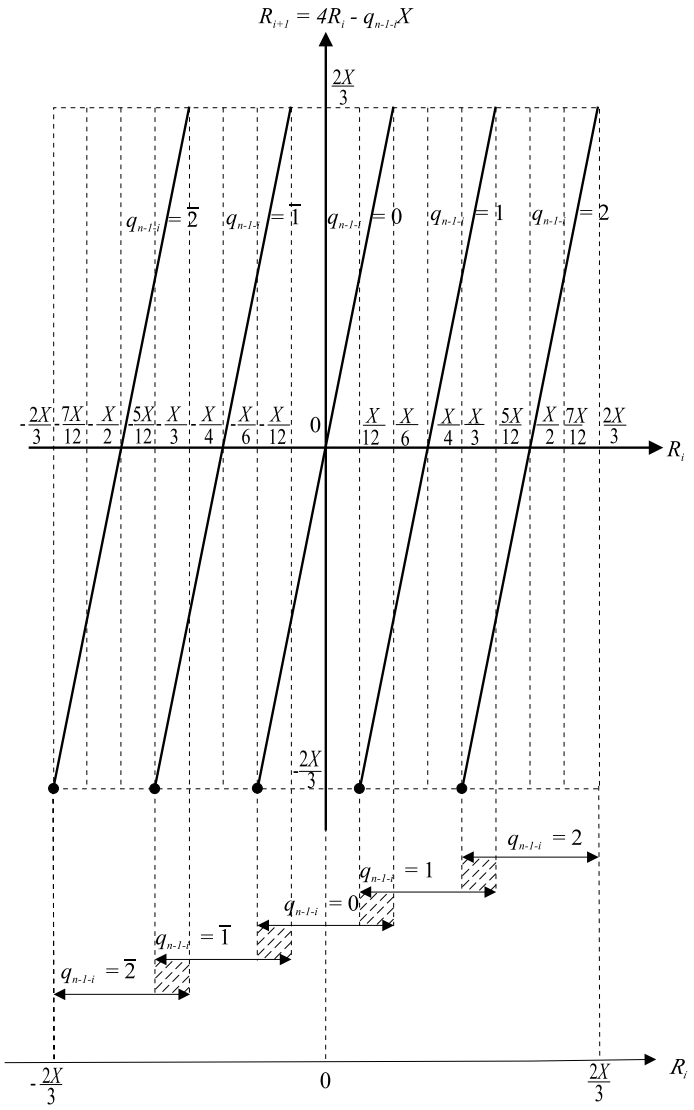


Fig. 4.26 Robertson diagram for the SRT radix-4 division of minimum redundancy

The lines corresponding to the dependences $R_{i+1} = f(R_i)$ (in the abscissa the representation of remainder R_i has been preferred, and not of the value $4R_i$, as required by the genuine Robertson diagram) for each possible digit of the quotient, with the variation intervals specified by (4.26), can be observed in Fig. 4.26. In the lower part of the figure, are highlighted (through hachure) the intervals of the remainder R_i, to which two values for the quotient digit correspond.

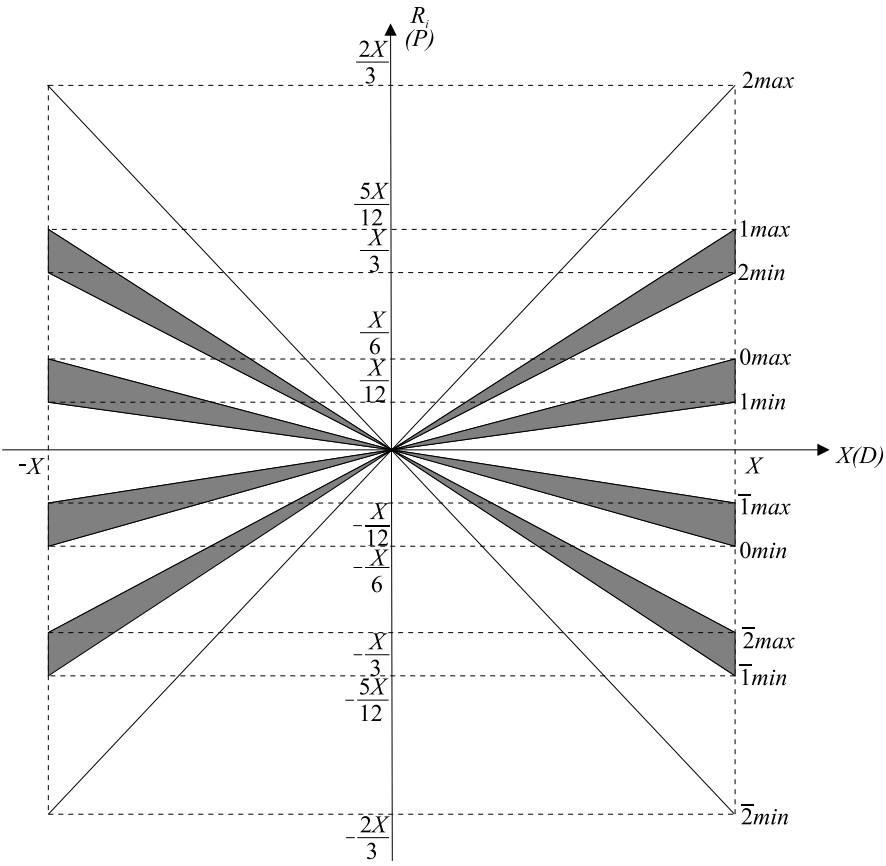


Fig. 4.27 P-D plot for the SRT radix-4 division of minimum redundancy

On the other hand, Fig. 4.27 presents the P-D plot modified to some extent, namely, on the P axis appears the partial remainder R_i in place of the shifted partial remainder $4R_i$, thus having, with the notations used, the $R_i - X$ plot for the same case SRT radix 4 with $q_{max} = 2$. The lines which delimit the zones of the remainder values corresponding to the possible digits of the quotient are marked with "max" (for the maximum values), and with "min" (for the minimum values). The overlapping zones are also marked (with hatched areas) to which correspond two values of the quotient digits. As regards these zones, by means of P-D plot a certain digit of the quotient can be selected. Moreover, this selection process does not require exact values for the remainders (whose computations require long latencies) and for the divisor, but only truncated values. Consequently, a second important problem consists of determining a sufficient number of bits which have to be exact for the remainders and for the divisor. Thus, we appeal to a staircase selection function, which is in principle similar to that in Fig. 4.28 [ErLa04, LaAn03, Parh00]. In this figure, divisor X is divided (following normalization) into intervals $[X_i, X_{i+1}]$, with

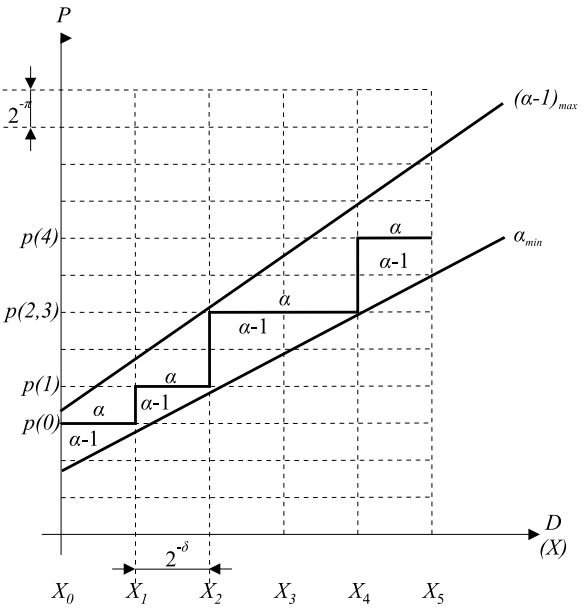


Fig. 4.28 Staircase selection function representation

$X_0 = 1/2$ and $X_{i+1} = X_i + 2^{-\delta}$, so that an interval is represented by the most significant δ fractional bits of X. Within each interval, the delimitation between the values of q_{n-1-i} and $(q_{n-1-i} - 1)$ for the quotient digit is made using $p(i)$ constants. They are equal to the values of the truncated remainders at the integer bits to which the most significant π fractional bits are added, and they correspond to one interval $[X_i, X_{i+1})$ (as is the case for $p(0)$, $p(1)$ or $p(4)$) or to several intervals (as is the case for $p(2, 3)$, to two intervals). Under the circumstances, to satisfy the containment and continuity conditions the selection of any digit $q_{n-1-i} > 0$ of the quotient shall satisfy the following correlation [ErLa04]:

$$(q_{n-1-i})_{min}\left(X_i + 2^{-\delta}\right) \leq p(i) \leq (q_{n-1-i} - 1)_{max}(X_i) \tag{4.27}$$

Similarly, the selection of any digit $q_{n-1-i} \leq 0$ of the quotient shall satisfy the following correlation:

$$(q_{n-1-i})_{min}(X_i) \leq p(i) \leq (q_{n-1-i} - 1)_{max}\left(X_i + 2^{-\delta}\right) \tag{4.28}$$

Both relations, (4.27) and (4.28), have to be fulfilled for all i values.

The design problem consists of finding the selection constants $p(i)$ and the division intervals so that π and δ will result with minimum values. Unfortunately, as can be graphically observed, at the reduction of δ, π increases, and vice versa. A possible optimization criterion could be the minimization of the sum $(\pi + \delta)$. Anyway, if the sign bit is also taken into account, the integer part of the partial remainders generally requires $(1 + \log_2(\rho r X))$ bits, i.e. there is an upper bound, because $\rho \leq 1$ and $X \in [-1, 1)$, which is equal to $(1 + \log_2 r)$ bits. In our particular case $(r = 4)$,

this implies the value 3 for the maximum number of bits corresponding to the representation of the integer part of the partial remainders [ErLa04].

On the other hand, in order to determine the minimum number δ of bits of the divisor, which is needed in the selection process, we observe that for two adjacent digits of the quotient, the values of the partial remainders corresponding to these digits are required to display an overlapping region. Thus, starting from (4.27), for a digit $q_{n-1-i} > 0$ (case $q_{n-1-i} \leq 0$ supports a similar approach starting from (4.28)) the overlapping region is generally guaranteed by the following condition:

$$(q_{n-1-i} - 1)_{max}(X_i) - (q_{n-1-i})_{min}\left(X_i + 2^{-\delta}\right) \geq 0 \tag{4.29}$$

But, if we take into account the range of variation tolerated for rR_i, and previously seen, namely $(-\rho + q_{n-1-i})X \leq rR_i < (\rho + q_{n-1-i})X$, relation (4.29) becomes:

$$(q_{n-1-i} - 1 + \rho)X_i - (q_{n-1-i} - \rho)\left(X_i + 2^{-\delta}\right) \geq 0 \tag{4.30}$$

Following certain obvious rearrangements of (4.30), we have:

$$(2\rho - 1)X_i \geq (q_{n-1-i} - \rho)2^{-\delta} \tag{4.31}$$

Condition (4.31) has to be fulfilled for all intervals of divisor X and for all $q_{n-1-i} = q_{n-1-i}$ digits of the quotient, the worst case being when the interval has the smallest value and q_{n-1-i} has the largest value. Since $X \geq (1/2)$ and $q_{n-1-i} \leq q_{max}$, but also $\rho = q_{max}/(r-1)$, (4.31) becomes:

$$2^{-\delta} \leq \frac{2\rho - 1}{2(q_{max} - \rho)} = \frac{2\rho - 1}{2\rho(r-2)} \tag{4.32}$$

But the use of the minimum value, given by (4.32), for δ, may lead to large numbers for π, i.e. many bits required for the shifted remainder. Limitation (4.32) is useful, because it reduces the number of alternatives to be taken into account, but when choosing the values for δ and π a decisive factor is represented by the technological peculiarities of the implementation [ErLa04].

Reverting to our case ($r = 4$, $q_{max} = 2$, and consequently $\rho = 2/3$), (4.32) gives us the required limitation on δ, i.e.:

$$2^{-\delta} \leq \frac{2\rho - 1}{2(q_{max} - \rho)} = \frac{1}{8} \quad \Rightarrow \quad \delta \geq 3 \tag{4.33}$$

According to (4.33), a truncated divisor of at least three bits is needed. But, in this case, a large value for π results. For instance, for $q_{n-1-i} = 2$, corresponding to the divisor part between 4/8 and 5/8, and taking into account (4.29) and (4.30), there results:

$$(q_{n-1-i} - 1)_{max}(X_i) = (q_{n-1-i} - 1 + \rho)X_i = \left(1 + \frac{2}{3}\right)\frac{4}{8} = \frac{5}{6} \tag{4.34}$$

$$(q_{n-1-i})_{min}\left(X_i + 2^{-\delta}\right) = (q_{n-1-i} - \rho)\left(X_i + 2^{-\delta}\right) = \left(2 - \frac{2}{3}\right)\frac{5}{8} = \frac{5}{6} \tag{4.35}$$

Otherwise, (4.34) and (4.35) show the presence of only one selection constant, which requires total precision, without truncation. This is the reason why we cannot work with the minimum value of 3 bits, but this number is increased to 4. Knowing now that $\delta = 4$, let us determine the minimum value of π, namely to determine those $p(i)$ constants which satisfy, for all q_{n-1-i} and X_i, the conditions (4.27) and (4.28), and, moreover, have the minimum number of bits.

Taking into account $q_{n-1-i} = 2$, i.e. the overlapping zone between $(q_{n-1-i})_{min} = 2_{min}$ and $(q_{n-1-i} - 1)_{max} = 1_{max}$ (Fig. 4.27), and because $\delta = 4$, $X_0 = 8/16$, $X_1 = 9/16, \ldots, X_7 = 15/16$, condition (4.27) requires value restrictions which will be estimated (with $\rho = 2/3$) for each interval $[X_i, X_i + 2^{-\delta})$ according to the following model:

- for $[X_0, X_1) = \left[\frac{8}{16}, \frac{9}{16}\right)$, we have $2_{min}\left(\frac{9}{16}\right) \le p(0) \le 1_{max}\left(\frac{8}{16}\right)$, i.e.

$$\left(2 - \frac{2}{3}\right)\frac{9}{16} \le p(0) \le \left(1 + \frac{2}{3}\right)\frac{8}{16} \text{ or } \frac{3}{4} \le p(0) \le \frac{5}{6}$$

$$\text{and we choose } p(0) = \frac{3}{4} = \frac{6}{8} \tag{4.36}$$

Executing computations similar to (4.36) for all $[X_i, X_i + 2^{-\delta})$ intervals, the following will be obtained:

- for $\left[\frac{9}{16}, \frac{10}{16}\right)$, we have $\frac{5}{6} \le p(1) \le \frac{45}{48}$ and we choose $p(1) = \frac{7}{8}$;
- for $\left[\frac{10}{16}, \frac{11}{16}\right)$, we have $\frac{11}{12} \le p(2) \le \frac{25}{24}$ and we choose $p(2) = 1 = \frac{8}{8}$;
- for $\left[\frac{11}{16}, \frac{12}{16}\right)$, we have $1 \le p(3) \le \frac{55}{48}$ and we choose $p(3) = 1 = \frac{8}{8}$;
- for $\left[\frac{12}{16}, \frac{13}{16}\right)$, we have $\frac{13}{12} \le p(4) \le \frac{5}{4}$ and we choose $p(4) = \frac{9}{8}$; (4.37)
- for $\left[\frac{13}{16}, \frac{14}{16}\right)$, we have $\frac{7}{6} \le p(5) \le \frac{65}{48}$ and we choose $p(5) = \frac{5}{4} = \frac{10}{8}$;
- for $\left[\frac{14}{16}, \frac{15}{16}\right)$, we have $\frac{5}{4} \le p(6) \le \frac{35}{24}$ and we choose $p(6) = \frac{5}{4} = \frac{10}{8}$;
- for $\left[\frac{15}{16}, \frac{16}{16}\right)$, we have $\frac{4}{3} \le p(7) \le \frac{25}{16}$ and we choose $p(7) = \frac{3}{2} = \frac{12}{8}$

From (4.36) and (4.37), it results that $p(i)$ constant values can be found with 3 fractional bits at the most (for $p(1) = 7/8$). The P-D plot diagram corresponding to the analyzed overlapping zone can be followed in Fig. 4.29 [ErLa04]. Similar investigations have to be made for the other overlapping zones $((0_{max}, 1_{min}), (\overline{1}_{max}, 0_{min})$

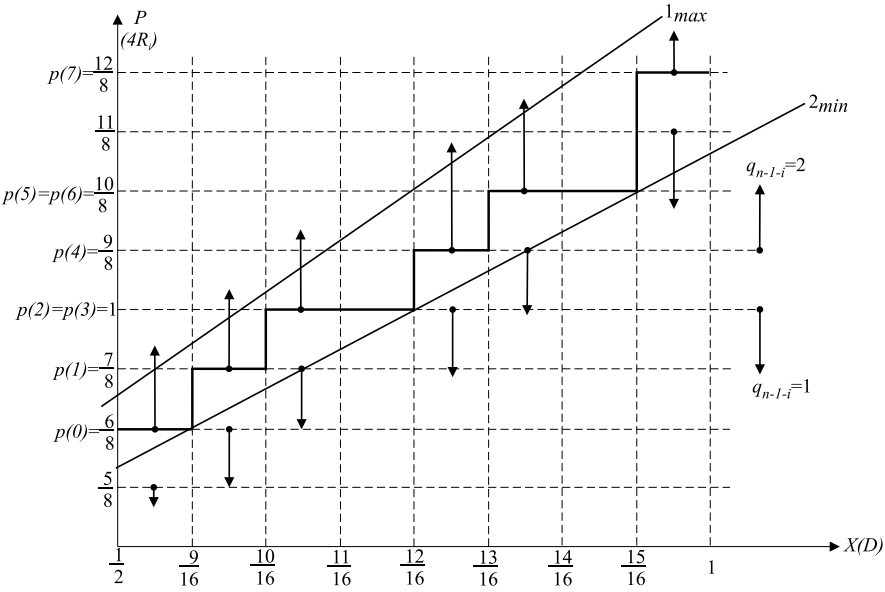


Fig. 4.29 P-D plot corresponding to the overlapping region between $(\alpha - 1)_{max} = 1_{max}$ and $\alpha_{min} = 2_{min}$

and $(\overline{2}_{max}, \overline{1}_{min})$), as well, (Fig. 4.27). The results of these analyses lead to the same value 3 for the maximum number of fractional bits required for the representation of the partial remainders [ErLa04, HePa03]. If these 3 bits are concatenated with the 3 bits corresponding to the integer part, it follows that it is sufficient to determine the most significant $\pi = 6$ bits of the partial remainders, which, together with the most significant $\delta = 4$ bits of the divisor, assure the correct execution of the quotient bits selection process.

The same result can be obtained by applying reasoning based on trial and error. Thus, let us suppose that, together with $\delta = 4$, $\pi = 5$ would be sufficient. Under the circumstances, let us consider the case of the shifted partial remainder $4R_i$. Taking into account these conditions, we will consider the particular case when the shifted partial remainder $4R_i$ has its value included in the interval $[(3/4), (4/4))$, thus having, in binary representation, the form $0.0011\ldots$ (the msb being assigned to the sign), and when the divisor X has a value in the interval $[(9/16), (10/16))$, thus having, in binary representation, the form $0.1001\ldots$ (the sign is ignored because the divisor is considered to be non-negative). By determining the unfavorable range of variation of $(R_i/X)^*$, the lower bound of the range is $(0.0011/0.1010) = (3/10)$ and the upper bound is $(0.100/0.1001) = (4/9)$. Returning to Fig. 4.26 and detailing the overlapping region corresponding to the digits 1 (for the upper bound $(R_i/X) = (5/12)$) and 2 (for the lower bound $(R_i/X) = (1/3)$), the intervals specified in Fig. 4.30 are obtained. In the figure are shown also the range of variation for $(R_i/X)^*$, represented by the interval $((3/10), (4/9))$, as well as the range of variation for (R_i/X) corresponding to the chosen constant, according to (4.37), for

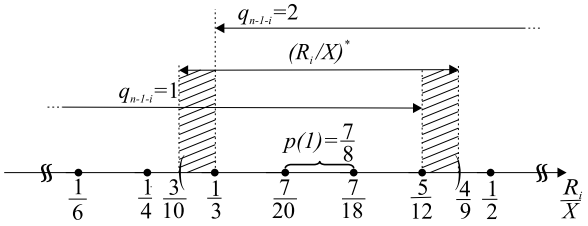


Fig. 4.30 Significant values resulting from the analysis of the overlapping region of the quotient digits 1 and 2

a divisor in the interval [(9/16), (10/16)), namely $p(1) = (7/8)$. From Fig. 4.30 it results that the range of variation ((7/20), (7/18)) corresponding to the constant $p(1) = (7/8)$ is entirely covered by the overlapping region of the digits 1 and 2, whereas the hatched regions of the $(R_i/X)^*$ interval exceed the overlapping area. It can be concluded that the selection process for the quotient bits, based on truncation to $\pi = 5$ bits of the partial remainders and to $\delta = 4$ bits of the divisor, does not benefit redundancy, being insufficiently precise.

By extending the truncation of the partial remainders to $\pi = 6$ bits, as previously presented, together with the $\delta = 4$ precision bits of the divisor, by means of computer-assisted investigation, the data presented in the table of Fig. 4.31 [HePa03] are obtained. These have been obtained through computer-aided investigation and comprise, for the divisor values given in Fig. 4.29, the selection of the current signed digit q^*_{n-1-i} of the quotient as a function of the value interval corresponding to the partial remainder R_i. The table has used integers from 8 to 15 for the fractional values from (1/2) to (15/16) of the divisor X. A similar interpretation can also be applied to the integers associated with the limits $-R_{imin}$ and R_{imax}—of the ranges of variation the partial remainders R_i. Thus, to the value (−12), from the first line of the table, corresponds the binary representation 110100 which is the two's complement for 101100 and which, in the interpretation with implicit binary point separating three bits at a time, corresponds to (−3/2). Under the circumstances, if, for instance, $X = (13/16)$ and R_i is within the interval from $R_{imin} = 101101$, which corresponds to (−19/8), to $R_{imax} = 110101$, which corresponds to (−11/8), then for the quotient digit the value $\overline{2}$ will be chosen.

Having made these remarks, let us, for instance, follow the SRT radix 4 procedure applied to the division of the binary equivalents corresponding to the integer decimal numbers $Y = 535_{10}$ and $X = 7_{10}$. Mention should be made that, since to the shifted partial remainder from register A it is possible to add or subtract the value $2X$, the registers A and M, as well as the adder/subtracter, are extended by one rank, while in the msb of M, the value 0 is permanently present. Thus, if we suppose that we work on the 8 bits standard length, A and M will contain 9 ranks (Fig. 4.32). Following the elimination of the 5 bits of 0 through the operand's left-shift, the procedure proper starts with the investigation of the most significant 6 bits of A, to which, in the already mentioned terms, the value (+8) corresponds. This value will have to be introduced in one of the $[R_{imin}, R_{imax}]$ intervals corresponding to $X = 14$ (the value 0 from the msb of M will be ignored). Since (+8) belongs to the interval [+3, +10], the first digit of the quotient becomes 1, and, from the left-shifted partial remainder (by two positions, because we operate in $r = 4$) is subtracted X,

X	R_i		q^*_{n-1-i}	X	R_i		q^*_{n-1-i}
	$R_{i\,min}$	$R_{i\,max}$			$R_{i\,min}$	$R_{i\,max}$	
8	-12	-7	$\overline{2}$	12	-18	-10	$\overline{2}$
8	-6	-3	$\overline{1}$	12	-10	-4	$\overline{1}$
8	-2	+1	0	12	-4	+3	0
8	+2	+5	1	12	+3	+9	1
8	+6	+11	2	12	+9	+17	2
9	-14	-8	$\overline{2}$	13	-19	-11	$\overline{2}$
9	-7	-3	$\overline{1}$	13	-10	-4	$\overline{1}$
9	-3	+2	0	13	-4	+3	0
9	+2	+6	1	13	+3	+9	1
9	+7	+13	2	13	+10	+18	2
10	-15	-9	$\overline{2}$	14	-20	-11	$\overline{2}$
10	-8	-3	$\overline{1}$	14	-11	-4	$\overline{1}$
10	-3	+2	0	14	-4	+3	0
10	+2	+7	1	14	+3	+10	1
10	+8	+14	2	14	+10	+19	2
11	-16	-9	$\overline{2}$	15	-22	-12	$\overline{2}$
11	-9	-3	$\overline{1}$	15	-12	-4	$\overline{1}$
11	-3	+2	0	15	-5	+4	0
11	+2	+8	1	15	+3	+11	1
11	+8	+15	2	15	+11	+21	2

Fig. 4.31 Table for the selection of the current signed digit value of the quotient digit depending on the divisor value and the value interval corresponding to the partial remainder

and then, the above-described procedures will be repeated. At a certain moment, for the remainder R_i the binary number 111001 results, which represents the two's complement for $100111 = -7$. This value belong to the interval $[-11, -4]$, consequently the quotient digit equal to $\overline{1}$ will be chosen. We also mention that the remainder $R_i = 000011 = +3$ corresponds both to the interval $[-4, +3]$, having associated the quotient digit 0, and to the interval $[+3, +10]$, having associated the quotient digit 1. Digit 0 is preferred, because it does not imply the activation of the adder/subtracter, with the corresponding time saving. Generally, when the choice

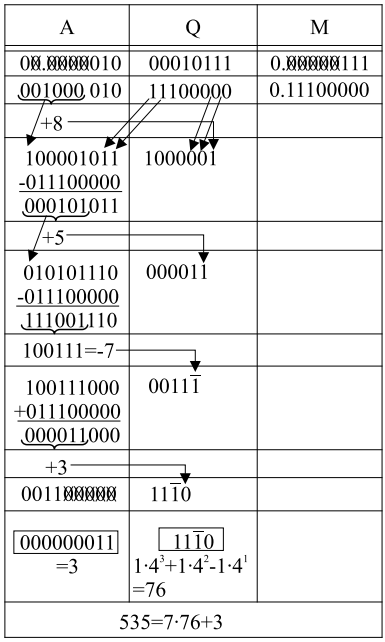


Fig. 4.32 SRT radix-4 binary division example

may be made among the digits 0 and 1, or $\overline{1}$ (and the table from Fig. 4.31 contains other such cases), it is recommended to choose digit 0. Finally, mention should be made that it is necessary to right-shift the last remainder by the same number (5) of bits initially used to bring divisor X into the interval $[(+1/2), (+1))$, and also that the quotient has been obtained in signed-digit form, wherefrom the conversion into the conventional binary form has to be made, as specified above.

There is one more aspect to be discussed, namely the obtaining of the partial remainder estimated value which enables the execution of the selection process [ErLa04, HePa03, Parh00]. The method's implementation's performance decisively depends on the speed with which the estimated partial remainder is generated. This problem can be efficiently solved by appealing to redundant representations of carry-save or signed-digit type, which are produced by fast adders and are carry-free. Thus, one of the methods to accelerate binary multiplication is based, as seen above, on the use of one or several carry-save adders by which sum and carry vectors of the word's length are rapidly generated, enabling the product efficient to be obtained. Such a carry-save adder can also be used in the implementation of binary division, it enabling the generation of the two vectors, sum and carry, for each of the partial remainders. Through the addition of these vectors by means of a carry adder the exact value of the partial remainders may be obtained, but this addition process requires substantial latencies, whatever the synthesis method used for the adder. Moreover, in the selection process of the quotient bits there is no need for exact values, i.e. of complete precision, of the partial remainders, but only for some truncated ones at the most significant, generally, π bits. Consequently, there will be taken into account only these π bits of the two vectors, and there will be obtained,

by means of them, the estimated, not the exact, value of the partial remainder, and based on this value the current bit of the quotient will be chosen. Obviously, this remainder can be obtained by appealing to a π bit carry propagation adder, for which more efficient synthesis solutions exist, the smaller the value of π is. But a solution with even better performance can be obtained by using a table of $(\gamma 2^{2\pi})$ bits capacity, where γ represents the number of bits required for the coding of all the quotient digits. In this table can be found all the combinations of the $(\pi + \pi)$ bits from the two vectors, i.e. sum and carry, with the associated values of the quotient bits. Thus, when $r = 4$, with $\pi = 6$ and $q_{max} = 2$, which implies $\gamma = 3$ bits for the coding of the 5 digits of the quotient, the table requires $2^{12} = 4096$ combinations, a value on 3 bits corresponding to each of them. As regards the implementation of such a table, it can be efficiently done through a Programable Logic Array (PLA), generally with 2π inputs. This solution is preferred, mainly when the table has a regular structure, to the alternative one based on a Read Only Memory (ROM) [HePa03]. Whatever the implementation, the use of a table requires, at each procedure iteration, a delay which, essentially, is caused by the so-called "table lookup" operation [Parh00].

Finally, as regards SRT radix 4 division, mention should be made that the Intel Pentium processor uses this algorithm, the implementation being based on a PLA. Within this context, we recall that in the first Pentium chips there was a failure which came to be known as the "division bug". Essentially, this failure appeared after the P-D plot generation, in the quotient lookup table transposition in the PLA, namely in the optimization of this implementation starting from the hypothesis that certain elements of the table will never be accessed. This design error materialized through the return, when reading from the PLA at the "optimized" locations, of digit 0 instead of +2 [HePa03, Parh00, ALMN05].

Consequently, generalizing the problems of radix r division, $r > 2$, it can be asserted that the reduction of the number of iterations is counterbalanced by the complexity of the quotient digit selection process. It can be somewhat simplified when we appeal to a redundant set of digits for the quotient, because truncated values of the remainder and of the divisor can be used. In [ErLa04] the increase of value r is investigated, and the conclusion is that the direct implementation of the selection function proves to be practical for $r = 8$, but, for $r = 16$, a suitable implementation is represented by the overlapping of two stages with $r = 4$.

4.5 Binary Division Based on Fast Convergence

The fundamental characteristic of the methods based on fast convergence consists of the fact that, starting from an initial approximation, they estimate a function which they improve in an iterative way. Unlike the previously studied algorithms, characterized by the recurrent obtaining of a quotient digit at each iteration, i.e. having a linear convergence, the new methods double the number of correct bits in the approximation at each iteration, presenting a quadratic convergence [HePa03, ErLa04, ObFl97]. Consequently, the number of iterations required to reach a certain accuracy is smaller in division by convergence.

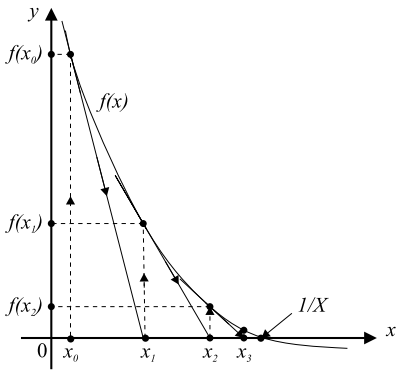


Fig. 4.33 Gradual search for the root $f(x) = 0$ corresponding to $1/X$

On the other hand, the conventional division methods, based on the recurrence of digits, imply, at each iteration, the selection of a digit, its multiplication by the divisor, and a subtraction or an addition, while in division by convergence, the principal operation, which is involved in each iteration, is multiplication. That is why these methods are also called multiplicative methods [ErLa04]. However, the multiplication operations require high precision, and thus the latency corresponding to an iteration, generally, is longer than that corresponding to the recurrence-based division methods.

As regards applicability, the methods of division by convergence prove attractive for the floating point units of a processor.

4.5.1 The Newton-Raphson Method

Division by the Newton-Raphson method obtains quotient $Q = Y/X$ by first determining the reciprocal value $1/X$, the result being then multiplied by dividend Y. Thus, the reciprocal $1/X$ will be computed by using the Newton-Raphson iteration for determining roots, in which a function obtains value 0. Let us consider the non-linear function $f(x)$ from Fig. 4.33 [HePa03], for which we gradually, namely iteration by iteration, look for the root $f(x) = 0$ corresponding to $1/X$. Thus, we start from an initial approximation x_0, and from point $(x_0, f(x_0))$, we draw the tangent to the curve which is the graphical representation of the function $f(x)$. The tangent crosses axis Ox at the abscissa point x_1, a value which represents a better approximation for $1/X$ than the initial one, (x_0). After repeating the above-described procedures, we shall successively obtain the values $x_2, x_3, \ldots$, and, given x_i, the next value x_{i+1} results by using the analytical equation of the tangent at point $(x_i, f(x_i))$, given by the following expression:

$$y - f(x_i) = f'(x_i)(x - x_i) \tag{4.38}$$

where $f'(x_i)$ represents the derivative of the function $f(x)$ with respect to x, evaluated for the abscissa x_i.

Making $y = 0$ in (4.38), we obtain, for the next better approximation, the expression given below:

$$x = x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{4.39}$$

This general method is applied to determine the reciprocal value $1/X$ by appealing to the function $f(x) = \frac{1}{x} - X$, whose root is $x = 1/X$. Under these circumstances, if the values $f(x_i) = \frac{1}{x_i} - X$ and $f'(x_i) = -\frac{1}{x_i^2}$ are introduced in (4.39), the following will be obtained:

$$x_{i+1} = x_i - \frac{\frac{1}{x_i} - X}{-\frac{1}{x_i^2}} = x_i(2 - x_i X) \tag{4.40}$$

From the computational point of view, each iteration requires, according to (4.40), two multiplications and a two's complementing step.

In order to demonstrate that this method has a quadratic convergence, we shall start from the error corresponding to the iteration i expressed by $\varepsilon_i = \frac{1}{X} - x_i$. Consequently, for the error corresponding to the iteration $(i+1)$ we have $\varepsilon_{i+1} = \frac{1}{X} - x_{i+1}$, which, if (4.40) is taken into account, leads us to:

$$\varepsilon_{i+1} = \frac{1}{X} - x_i(2 - x_i X) = X\left(\frac{1}{X} - x_i\right)^2 = X\varepsilon_i^2 \tag{4.41}$$

From (4.41), for $X \in [(+1/2), (+1))$, it results that $\varepsilon_{i+1} < \varepsilon_i^2$, thus demonstrating the quadratic convergence.

One of the important problems of this method is represented by the choice of the initial approximation x_0, because its accuracy determines the number of iterations required. Some of the alternatives for this choice are as follows [ErLa04]:

(a) The use of a constant whose value is independent from that of the divisor X.
(b) The use of an initial approximation of the form $x_0 = a - bX$, where a and b are constants. If b is a power of 2, then the implementation will be simple. If now $X \in [(+1/2), (+1))$, a good choice of the two constants is as follows: $a = 2{,}928$ and $b = 2$.
(c) The use of a lookup table which consists of a set of constants that represent truncated values of $1/X$, one for each interval of $1/X$. The determining of the number of bits in the truncated values has to be correlated with the resulting errors, an important requirement being error minimization.

Another problem specific to the Newton-Raphson method is represented by the computations' accuracy, more precisely that of the multiplications provided by (4.40). They are intended to be executed in complete precision, increasing the number of bits in the product at each iteration. If we denote by m the number of bits of the initial approximation x_0, divisor X being assumed to be of n bits, then the width of the two products corresponding to i iterations is given in the table from Fig. 4.34

i	x_i	x_iX	$x_{i+1}=x_i(2-x_iX)$
0	m	$m+n$	$2m+n$
1	$2m+n$	$2m+2n$	$4m+3n$
2	$4m+3n$	$4m+4n$	$8m+7n$
...			

Fig. 4.34 Table with the number of bits corresponding to a Newton-Raphson iteration as a function of the iteration's count

[ErLa04]. In the table, we can observe the excessive increase of the number of ranks for the intermediate and final results, which makes the implementation impossible. Consequently, the products are truncated or rounded, so that the effect of these operations over the final error is as small as possible. Implementations are based either on a floating point multiplier which enables rounded products to be obtained, of the type that will be presented in the next chapter, or a multiplier as a rectangular combinational array of mn capacity, similar to that of quadratic type presented in the previous chapter.

Having made these remarks, the binary division operation based on the Newton-Raphson method develops according to the following steps:

Step 1. If necessary, divisor X is scaled, being brought into the intended interval through the corresponding shift of both operands, and then, the first approximation x_0 of $1/X$ is chosen, for instance through table lookup.

Step 2. The computations provided by (4.40) are iterated, until a value x_{n-1} of sufficient accuracy is arrived at.

Step 3. $x_{n-1}Y$ is computed.

On the other hand, the Newton-Raphson method can stand at the basis of a procedure for the calculation of the square root [ErLa04]. Thus, we shall use the same relation (4.39), but this time we shall use the function $f(\sigma)=\sigma^2-Z$, one of whose roots is $\sigma=\sqrt{Z}$. Since $f'(\sigma)=2\sigma$, the following iteration will be obtained:

$$\sigma_{i+1}=\sigma_i-\frac{f(\sigma_i)}{f'(\sigma_i)}=\sigma_i-\frac{\sigma_i^2-Z}{2\sigma_i}=2^{-1}\left(\sigma_i+\frac{Z}{\sigma_i}\right) \tag{4.42}$$

According to (4.42), each iteration requires a division, an addition, and a right-shift. It would be favorable if we could substitute the division by multiplication and two's complementing operations, according to the model of the previous use of the Newton-Raphson method (refer to relation (4.40)). Thus, we shall first compute the reciprocal value of the square root, and finally multiply it by Z. Thus, we shall use function $f(\rho)=\frac{1}{\rho^2}-Z$, one of whose roots is $\frac{1}{\rho}=\sqrt{Z}$. Since $f'(\rho)=-\frac{2}{\rho^3}$, the following iteration will be obtained:

$$\rho_{i+1}=\rho_i-\frac{f(\rho_i)}{f'(\rho_i)}=\rho_i-\frac{\frac{1}{\rho_i^2}-Z}{-\frac{2}{\rho_i^3}}=2^{-1}\rho_i\left(3-\rho_i^2Z\right) \tag{4.43}$$

Regarding the subtraction operation from relation (4.43), we foresee its transformation into $3 - \rho_i^2 Z = 1 + (2 - \rho_i^2 Z)$, where $(2 - \rho_i^2 Z)$ represents the two's complement of product $\rho_i^2 Z$, which, it is already known, is obtained through the flipping of the bits of $\rho_i^2 Z$ (by one's complementing) and the addition of a binary unit to the lsb of the value thus obtained. Ignoring this binary unit, as well as that which has been added to the two's complement, we obtain the approximation of difference $(3 - \rho_i^2 Z)$ by complementing the bits of product $\rho_i^2 Z$ [ErLa04], thus greatly simplifying the assessment provided in (4.43).

The other considerations inserted in the presentation of the division operation can be similarly extended to the square root extraction, so that, at the above description of the steps of the algorithm, an essential change has to be made, i.e. iteration (4.40) shall be substituted by (4.43) in step 2.

4.5.2 Goldschmidt's Method

The characteristic of the second method based on fast convergence, sometimes called Goldschmidt's algorithm [HePa03, PiBr02], other times the multiplicative normalization method [ErLa04], or division by repeated multiplications [Parh00], consists of determining two multiplicative recurrences, of which one causes a convergence towards 1, and the other a convergence towards the intended function represented by the quotient Q of Y divided by X. Thus, in order to compute Q, we shall multiply both Y and X by a sequence of multiplication factors $r_0, r_1, \ldots, r_{p-2}$, i.e.:

$$Q = \frac{Y}{X} \cong \frac{Y r_0 r_1 \cdots r_{p-2}}{X r_0 r_1 \cdots r_{p-2}} \tag{4.44}$$

If now the choice of the values $r_0, r_1, \ldots, r_{p-2}$, is made so that the denominator from (4.44) tends towards 1, then the product from the numerator will converge towards Q. Consequently, the method starts from $y_0 = Y$ and $x_0 = X$ and computes, at each iteration, values of $y_{i+1} = r_i y_i$, and $x_{i+1} = r_i x_i$ types, so that the ratio $\frac{y_{i+1}}{x_{i+1}} = \frac{y_i}{x_i} = \cdots = \frac{Y}{X}$ remains constant. Choosing r_{p-2} so that x_{p-1} tends towards 1, y_{p-1} will tend towards Q. This process does not allow a remainder to be obtained, but, if necessary, it can be computed by using the relation $R = Y - XQ$, implying supplementary operations which consist of a multiplication and a subtraction.

A first problem regarding this method is the choice of r_i multipliers. To solve it, let us consider that divisor X is represented by a normalized fraction belonging to the interval $[(+1/2), (+1))$. Obviously, if this condition is not initially fulfilled, then we perform shifts on operands Y and X until the condition is satisfied. Then, $y_0 = Y$ and $x_0 = X$ will be set, and divisor X will be presented in the form $X = 1 - \varphi$, where $\varphi < 1$. Under these circumstances, if multiplier r_0 is chosen in such a way as to represent the two's complement of x_0, i.e. $r_0 = 2 - x_0 = 1 + \varphi$, then there results $x_1 = r_0 x_0 = (1 + \varphi)(1 - \varphi) = 1 - \varphi^2$. Then, to determine r_1, we appeal to the same two's complementing, so that $r_1 = 2 - x_1 = 1 + \varphi^2$, and, consequently,

$x_2 = r_1 x_1 = (1+\varphi^2)(1-\varphi^2) = 1-\varphi^4$, which is succeeded by $r_2 = 2 - x_2 = 1+\varphi^4$ etc. Obviously, since $\varphi < 1$, x_{p-1} will tend towards 1, and y_{p-1} will result through:

$$y_{p-1} = r_{p-2} y_{p-2} = \left(1+\varphi^{2^{p-2}}\right) y_{p-2} = \left(1+(1-X)^{2^{p-2}}\right) y_{p-2} \tag{4.45}$$

If the quantities $y_0, y_1, \ldots, y_{p-3}, y_{p-2}$ are gradually replaced in (4.45), the following expression will finally be obtained for y_{p-1}:

$$y_{p-1} = Y\left(1+(1-X)\right)\left(1+(1-X)^2\right)\left(1+(1-X)^4\right)\cdots\left(1+(1-X)^{2^{p-2}}\right) \tag{4.46}$$

A second problem of the Goldschmidt method is connected with the speed with which the denominator from (4.44) converges towards 1, or, in other words, the problem consists of determining the number of multiplications required to execute division. Thus, we find that $x_{i+1} = r_i x_i = (2 - x_i)x_i = 1 - (1 - x_i)^2$, i.e. $1 - x_{i+1} = (1 - x_i)^2$. In other words, if x_i is already close to 1 (for instance, $1 - x_i \leq \varepsilon$, ε representing the deviation), then x_{i+1} will be closer to 1 (for instance, $1 - x_{i+1} \leq \varepsilon^2$). This feature is known as quadratic convergence [Parh00], and it leads to a logarithmic number $(p - 1)$ of iterations. To justify this aspect, we recall that $X \in [+1/2, +1)$, thus $1 - x_0 \leq 2^{-1}$, then, through successive iterations, we have $1 - x_1 \leq 2^{-2}, 1 - x_2 \leq 2^{-4}, \ldots, 1 - x_{p-1} \leq 2^{-2^{p-1}}$. In case the word length is n bits, the closeness to 1 is limited by the value $(1 - 2^{-n})$. Consequently, the iteration can stop when 2^{p-1} is equal to or larger than n, wherefrom the number of iterations $p - 1 = \lceil \log_2 n \rceil$ results, with the same well known significance of the bars $\lceil\ \rceil$. Within the same context, we mention that, on operands of n bits, the $p - 1 = \lceil \log_2 n \rceil$ iterations require $(2p - 3)$ multiplications and $(p - 1)$ two's complementing operations [Parh00].

Certain statements have to be made regarding the initial approximation from which the algorithm starts. Thus, if φ is not close to 0, X being far from 1, it is recommendable to search, through table lookup, for an approximation X^* for the reciprocal value of X. Under these circumstances, the desired evaluation Y/X is substituted by YX^*/XX^* evaluation, which now has the denominator XX^* close to 1, ensuring a faster convergence.

Mention should be made that in this case, as well, the computations are affected by errors brought about by truncations and roundings, and sometimes errors compensation is necessary [HePa03, Kuli02].

Having made these statements, and taking into account the quadratic convergence given by the $(p - 1)$ iterations specified above, the binary division operation based on the Goldschmidt method develops according to the following steps:

Step 1. If necessary, X is scaled, being brought into the desired interval through the corresponding shift of both operands, and then, for instance through table lookup, the approximation X^* of $1/X$ is chosen.

Step 2. $y_0 = YX^*$ and $x_0 = XX^*$ are set.

Step 3. For $i = 0, 1, \ldots, p-2$, the computations given in the following loop are iterated.

Loop

$r_i = 2 - x_i;$
$y_{i+1} = r_i y_i;$
$x_{i+1} = r_i x_i;$

End loop

As with the Newton-Raphson method, the Goldschmidt procedure can be used for square root extraction. Thus, we start from $x_0 = Z$ and $y_0 = Z$, computing, at each iteration, values of $y_{i+1} = r_i y_i$, and $x_{i+1} = r_i^2 x_i$ types, so that the ratio $\frac{y_{i+1}^2}{x_{i+1}} = \frac{y_i^2}{x_i} = \cdots = Z$ remains constant. The evaluation of the square root S will be done by using a relation similar to (4.44), namely:

$$S = \frac{Y^2}{X} = \frac{Y^2 r_0^2 r_1^2 \cdots r_{p-2}^2}{X r_0^2 r_1^2 \cdots r_{p-2}^2} \tag{4.47}$$

Choosing r_{p-2} so that x_{p-1} tends towards 1, y_{p-1} will tend towards $\sqrt{Z}$. As with division, we now have to choose the multipliers r_i. This can be done by ensuring the quadratic convergence, namely if we have the deviations $\varepsilon_i = 1 - x_i$, and $\varepsilon_{i+1} = 1 - x_{i+1}$, then $\varepsilon_{i+1} = \varepsilon_i^2$ is required, i.e. $x_{i+1} = 1 - (1 - x_i)^2 = r_i^2 x_i$, wherefrom it results $r_i = \sqrt{2 - x_i}$. If we appeal to the Taylor serial expansion of the r_i function at the point $x_0 = 1$, and limit ourselves to the more significant terms, the following will be obtained for r_i:

$$r_i = 1 + 2^{-1}(1 - x_i) \tag{4.48}$$

Thus, r_i is obtained through the (one's) complementing of x_i, the fractional part shifting by one bit to the right, and a subsequent addition of a binary unit.

With these considerations the consistency step 3 from the Goldschmidt division algorithm changes into the following step for square root assessment:

Step 3*. For $i = 0, 1, \ldots, p-2$, the computations given in the following loop (r_0 is an initial approximation of $\sqrt{Z}$) are iterated

Loop

$r_i^2 = r_i r_i;$
$y_{i+1} = r_i y_i, x_{i+1} = r_i^2 x_i;$
$r_{i+1} = 1 + 2^{-1}(1 - x_{i+1});$

End loop

It can be observed that each iteration implies three multiplications, but two of them can be executed in parallel or in pipeline manner.

As regards the implementations, we shall limit our considerations to the division operation, but they can be extended without difficulties to the square root. In

both methods, Newton-Raphson and Goldschmidt, an initial step of table lookup is required, followed, in each iteration, by two successive multiplications and by the truncations of the intermediate results.

As concerns the choice of the initial approximation, the table lookup is the more efficient, the smaller is the table dimension. This requirement can be satisfied by appealing, for instance, to the storage of the reciprocal values for fewer points, and by using linear or higher order interpolation methods to compute, usually through a multiply-add operation, the initial approximation [Parh00, Ober99].

Division methods based on fast convergence usually require an efficient parallel multiplier built using a tree structure with CSAs. In the Goldschmidt algorithm, because in each division step two independent multiplications are executed by the same device, they can be overlapped through a solution based on an arithmetic pipeline with two stages. This solution cannot be applied to the Newton-Raphson method, because, according to iteration (4.40), the second multiplication with x_i requires the result of the first one [Parh00].

Since we want to finally characterize the two methods together, the Newton-Raphson and the Goldschmidt one, let us first mention that there is a correlation between them. Thus, we start from the particularization of relation (4.40) in the form $x_1 = x_0(2 - x_0X)$, and, by means of some obvious successive transformations, we obtain the following for x_2 and x_3:

$$\begin{aligned} x_2 &= x_1(2 - x_1X) = x_0(2 - x_0X)\left(1 + (1 - x_0X)^2\right) \\ x_3 &= x_2(2 - x_2X) = x_0(2 - x_0X)\prod_{i=1}^{2}\left(1 + (1 - x_0X)^{2^i}\right) \end{aligned} \tag{4.49}$$

where we have used the sign $\prod$ to denote the product of the parentheses corresponding to $i = 1$ and $i = 2$.

Then, let us apply inductive reasoning, considering valid the following relation:

$$x_n = x_0(2 - x_0X)\prod_{i=1}^{n-1}\left(1 + (1 - x_0X)^{2^i}\right) \tag{4.50}$$

Assuming that (4.50) is fulfilled, it is necessary to demonstrate the following extended relation:

$$x_{n+1} = x_0(2 - x_0X)\prod_{i=1}^{n}\left(1 + (1 - x_0X)^{2^i}\right) \tag{4.51}$$

Anyway, we have $x_{n+1} = x_n(2 - x_nX)$, and the parenthesis $(2 - x_nX)$ will be changed as follows:

$$
\begin{aligned}
2 - x_n X &= 2 - x_0 X (2 - x_0 X) \prod_{i=1}^{n-1} \left(1 + (1 - x_0 X)^{2^i}\right) \\
&= 2 - \left(1 - 1 + 2x_0 X - x_0^2 X^2\right) \prod_{i=1}^{n-1} \left(1 + (1 - x_0 X)^{2^i}\right) \\
&= 2 - \left(1 - (1 - x_0 X)^2\right) \prod_{i=1}^{n-1} \left(1 + (1 - x_0 X)^{2^i}\right) \\
&= 2 - 1 + (1 - x_0 X)^{2^n} = 1 + (1 - x_0 X)^{2^n}
\end{aligned} \tag{4.52}
$$

If (4.52) is replaced in (4.51), the following will result for x_{n+1}:

$$
\begin{aligned}
x_{n+1} &= \left(x_0 (2 - x_0 X) \prod_{i=1}^{n-1} \left(1 + (1 - x_0 X)^{2^i}\right)\right) \left(1 + (1 - x_0 X)^{2^n}\right) \\
&= x_0 (2 - x_0 X) \prod_{i=1}^{n} \left(1 + (1 - x_0 X)^{2^i}\right)
\end{aligned} \tag{4.53}
$$

Form (4.53) shows the validity of the anticipated relation (4.51), proving that for the Newton-Raphson method we certainly have the following value of quotient Q_{i+1}, obtained after the $(i + 1)$ iteration:

$$
Q_{i+1} = x_0 Y (2 - x_0 X)\left(1 + (1 - x_0 X)^2\right)\left(1 + (1 - x_0 X)^4\right) \cdots \left(1 + (1 - x_0 X)^{2^i}\right) \tag{4.54}
$$

On the other hand, according to (4.46), and after the same number $(i + 1)$ of iterations, the following will result for the Goldschmidt method corresponding to the value of quotient Q_{i+1}:

$$
Q_{i+1} = Y\left(1 + (1 - X)\right)\left(1 + (1 - X)^2\right)\left(1 + (1 - X)^4\right) \cdots \left(1 + (1 - X)^{2^i}\right) \tag{4.55}
$$

If in (4.54), $x_0 X$ is substituted by X, i.e. $x_0 = 1$, then (4.54) and (4.55) coincide, which means that both methods deliver the same sequence of Q_{i+1} values.

Following these observations, we shall refer to the two methods together, showing that they have the advantage of doubling the number of correct bits of the quotient at each iteration, presenting quadratic convergence. Moreover, they do not require dedicated hardware for the division operation, the multiplication device being sufficient, however with an increased control circuit complexity [HePa03].

On the other hand, the two methods have two drawbacks. The first consists of the fact that they do not allow the direct evaluation of the remainder. It can be obtained only through supplementary computations, as seen before in the Goldschmidt algorithm. This is the more unfavourable the more almost all high level languages imply remainder division with operations. The second disadvantage is connected

with a problem which will be developed in the next chapter, namely the problem of rounding. According to IEEE standards, the rounding operation shall be executed with certain exigencies regarding precision, which are only partly fulfilled by both multiplicative division methods [HePa03]. Moreover, within the same context of rounding, mention should be made that since the Newton-Raphson algorithm computes first the value $1/X$, which it subsequently multiplies by Y, it is possible that even when $1/X$ is correctly rounded, the result Y/X may not be correctly rounded. Let us, for instance, consider the division (in the more familiar decimal number system) of number $Y = 11$ by $X = 47$ working with $p = 2$ decimal numbers. Thus, for $1/47$ we obtain the value $0.021276\ldots$, which is rounded to 0.02. Multiplying it by 11, we will have 0.22 for the result of the division operation. However, dividing 11 by 47, results the value $0.234042\ldots$ which is rounded to 0.23, presenting a significant difference (0.01) as compared to that obtained using the Newton-Raphson multiplicative procedure.

Chapter 5
Functional Analysis and Synthesis of Floating Point Arithmetic Devices

5.1 Characteristics of the Floating Point Operation

5.1.1 Classification of Data Processing Units

By specifying from the beginning some elements of terminology, we show that, in compliance with many acknowledged literature landmarks, such as [HePa03, Stal99, Haye98], the Central Processing Unit (CPU), may be considered to be composed of two essential components, i.e. the so-called Program Control Unit (PCU) and the Data Processing Unit (DPU). On the one hand, the PCU is meant to capture, decode, and interpret that information which consists of the program's instructions, releasing the sequence of microoperations signals whose concatenation leads to the execution of each instruction and, implicitly, of the program. However it is not the PCU-specific range of problems that is the object of the analysis in this chapter, but that which corresponds to the other structural component represented by the DPU. As can be seen from its name, this latter unit is meant to process that part of information represented by data. Since these data can also be non-numerical ones, it is time to specify that our interest is mainly connected with DPUs which process numbers.

Essentially, a DPU contains the circuits for the implementation of arithmetic operations and logic functions, as well as a fast memory and a local control unit. A first criterion, decisive in the DPU's taxonomy, consists in the technological factor, on the basis of which there can be distinguished, on one hand, Arithmetic-Logic Units (ALUs), and, on the other hand, Arithmetic Processors (APs). The former are characteristic of that stage of technological development specific to medium scale integrated circuits and large scale integrated circuits when chips including more and more complex structures were created, starting with simple adders and subtracters up to multioperation units, that can be met within the ASIC families (Application-Specific Integrated Circuits) [ITRS01, Wake00]. On the other hand, the AP category belongs to the technological era of large scale and very large scale integration. They included in chips (which were, at first, independent) the circuitry of whole

M. Vlăduţiu, *Computer Arithmetic*,
DOI 10.1007/978-3-642-18315-7_5, © Springer-Verlag Berlin Heidelberg 2012

processors dedicated to the assessment of arithmetic and logic functions. They were available, at first, under the miniature form of some sequential logical circuits, and, after that, they received more and more of the CPU's proper attributes. As photolithographical procedures (which are employed in the technological process of integrated circuits manufacturing) become more refined, and due to the need to reduce delays on the wiring conductors (which, although microscopical, become "long" in the presence of the "vertical" increase of the signals frequency), AP's circuitry migrates from independent chips inside those of the processors, being identified as islands in the "ocean" represented by the silicon wafer.

Regarding ALUs, based on the technological criterion, they can be divided into the obsolete class of ALU bit slice circuits, and the conventional class of ALU components circuits. Since, at a certain moment, the packaging density of the components within a chip was limited to the integration of the circuits corresponding to a reduced number of bits (e.g., 4), the strategy of juxtaposing several such "slices" chips was used, the words being obtained through the concatenation of these slices [Poll90]. On the other hand, conventional ALUs can be divided, in their turn, into the older types with dedicated registers (such as, for instance, the accumulator), and those based on general registers.

The Arithmetic Processors (APs) can also be divided on the basis of the technological factor, but also on the basis of a functional one, into Peripheral Arithmetic Processors (PAPs) and Arithmetic Coprocessors (ACs). On the one hand, PAPs are independent of CPUs, being connected with them through a set of communication registers. Thus, a CPU executes instructions for data transfer, sending to the PAP's registers the set of operands, but also the control information, the latter being also treated as a data. The PAP decodes the control information, executes the operation, and finally places the result again in one of the communication registers, accessible to the CPU. Either on the initiative of PAP, through an interrupt signal, or on the initiative of the CPU, by consulting a state register, the CPU finds out that the PAP has finished the execution of the operation and takes over the result of the arithmetic operation from the PAP, through the same type of data transfer instructions. On the other hand, ACs are not independent, being "tailored" for a certain family of CPUs. Thus, each CPU is provided both with interfacing circuits with the ACs, ensuring the control of the connection between the two of them, and with special instructions meant to be executed by the AC. Usually, the CPU and the AC are tightly coupled, i.e. between the two of them there are direct communication lines which have the role of allowing their rapid entry into synchronism. This is because the AC is, usually, in a latent state of expectation, and has to be "awakened" when a coprocessor instruction is required to be executed. Technologically, the CPU and the AC were made up of two distinct chips, interconnected through a special socket, but at present the whole arithmetic and logic operation circuitry is integrated in the same silicon wafer corresponding to the whole processor.

Whether we deal with ALUs or APs, they can both be divided into two classes regarding the domain of values of the numbers on which they operate: a fixed point one, with a more restricted range of numbers, but also with simpler circuitry, and a floating point or mobile one, with a more extended range of numbers, but with

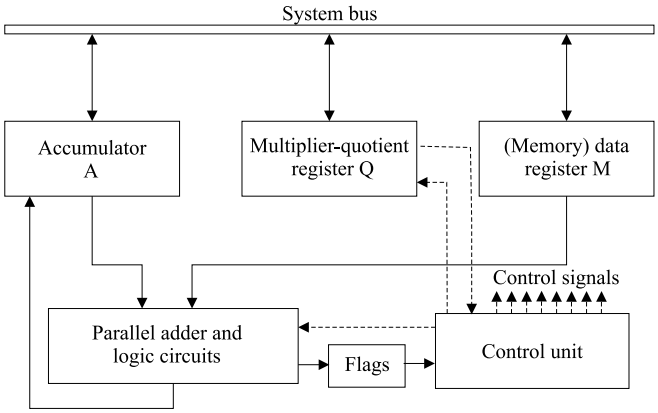


Fig. 5.1 Synthetic block diagram of an ALU

more complex circuitry, as well. In order to highlight the distinctive elements of the two classes, let us refer to the simpler case of an ALU with the synthetic diagram from Fig. 5.1 [Haye98], which covers the arithmetic operations treated in the previous chapters. There are recognized the three registers for which there have been maintained dedicated names, but which can be considered to belong to the general registers file. All three of them are bidirectionally connected to the system bus, while the M register, which houses one of the operands no matter the operation, is considered to belong to the memory unit. Besides the parallel adder with the associated logic circuits, the figure also contains the flags block, representing flip-flops which are set to 1 for the fulfillment of certain conditions, such as overflow, most significant bit (msb) carry, zero or negative result, etc. The control unit generates control signals (with dotted marking) whose concatenation ensures the implementation of the algorithms, which, for the sake of concrete rendering and without any loss of generality, are considered sequential. The typical way of using the registers consists of: $A := A + M$ for addition, $A := A - M$ for subtraction, $A.Q := Q \times M$ for multiplication, and $A.Q := A.Q/M$ for division. These operations are performed with numbers, but the operations can also be extended over some logic functions, that can be performed by using non-numerical operands, as well. One of their characteristics is the operation on bit pairs, involving the following typical usage of registers: $A := A$ *and* M for AND logic function, $A := A$ *or* M for OR logic function, $A := A$ *ex − or* M for EXCLUSIVE-OR logic function, $A :=$ *not* A for the complementation logic function. The functions' implementation is simple, implying wordgates on the whole words' length. Thus, in Fig. 5.2 is presented one of the possibilities of accomplishing the logic operations part of an ALU, such as the one presented in Fig. 5.1 [Haye98]. In registers A and M are kept operands X and Y, which are supplied, in accordance with the implemented function, to the gates of the logic level of the six wordgates. At their outputs are obtained the subfunctions which, negated by the wordgate from the second logic level, lead to the desired results. The four logic functions are controlled through the combination of values of

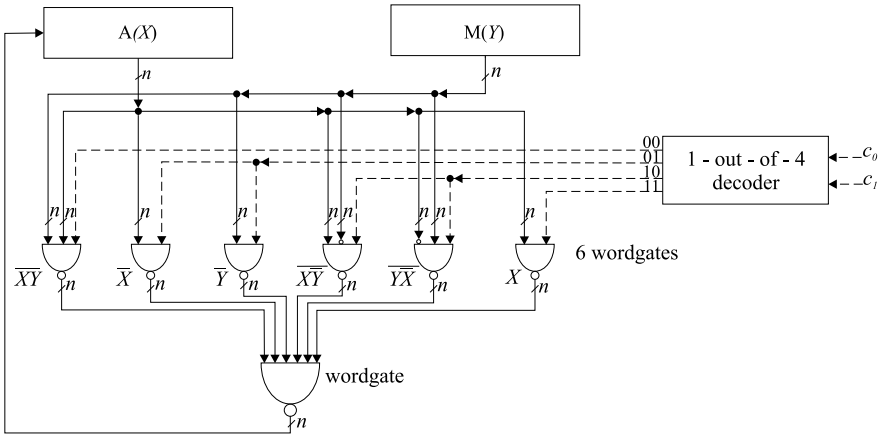


Fig. 5.2 Implementation version of the ALU part responsible for the logical operations

two control signals (c_0, c_1) that come from the CPU and which are decoded by the 1-out-of-4 decoder. Each output of the decoder controls the wordgates corresponding to a certain logic function (one wordgate for *and* and *not* and two wordgates for *or* and $ex-or$) and it determines, being activated on 1 (all the other outputs remain 0), the accomplishment of the given function. Mention should be made that at the wordgates' outputs from the first logic level there have been marked the subfunctions in terms of X and Y, the conditionings through the decoder's outputs being ignored (it can be observed how *or* and $ex-or$ functions are obtained by applying De Morgan laws).

With the observations made regarding the ALU logic subunit which, corroborated with the presentation from the previous chapters of the arithmetic subunit, enable the outline of the extent of ALU circuitry with fixed point operation, we shall highlight the differences between such a unit and a floating point one. They refer only to the arithmetic part, and, to highlight them, let us analyse the execution of the fundamental floating point operations in a more detailed way.

5.1.2 Problems Regarding Floating Point Operations

We shall refer, from the beginning, to the representation convention of floating point numbers, as well as to the system of notations we adher to. Thus, let us consider a number X which in the scientific notation is of the form $X = X_M B^{X_E}$, where B represents the base of the number system (implicitly it is equal to 2), and X_M and X_E are the fixed point numbers representing the mantissa and the exponent. It is known that the conventions of IEEE the 754 standard's format [BrO'H03] presents fields having variable numbers of bits, so we shall assume that a word used to represent a floating point number is made up, from left to right, of a bit assigned to

the sign, followed by e bits representing the exponent, and by m bits assigned to the mantissa. As concerns exponent X_E, this is a binary integer represented biased in an excess-$(2^{e-1}-1)$ code, which, particularized for the IEEE 754 standard on 32 bits with $e=8$, consists of the excess-127 code. Conventionally, in order to accept exceptions similar to those which correspond to the IEEE 754 standard, we also assume the value restriction $0 < X_E < 2^e - 1$. On the other hand, as far as mantissa X_M^* is concerned, it is represented by the fractional part of the so-called significand, of the form $X_M = 1.X_M^*$, which is a sign-magnitude binary number with the "hidden" integer bit (1). This "1" bit, meant to increase the representation precision, does not explicitly occur in the "packed" form of the number, as it appears as input data or as output results. For operation purposes, the floating point number has to be unpacked, which requires the corresponding insertion of the hidden bit. Mention should also be made that, as input data, and, also, as output results, floating point numbers occur in normalized form (implying the restriction $1 \leq |1.X_M^*| < 2$), but, within computations, they are usually operated on in their unnormalized form.

The representation conventions being established, the fundamental addition, subtraction, multiplication and division operations between floating point numbers $X = X_M 2^{X_E}$ and $Y = Y_M 2^{Y_E}$ are given by:

$$\begin{aligned} X + Y &= \left(X_M + Y_M 2^{Y_E - X_E}\right) 2^{X_E}, \quad \text{where } X_E \geq Y_E \\ X - Y &= \left(X_M - Y_M 2^{Y_E - X_E}\right) 2^{X_E}, \quad \text{where } X_E \geq Y_E \\ XY &= (X_M Y_M) 2^{X_E + Y_E} \\ Y/X &= (Y_M / X_M) 2^{Y_E - X_E} \end{aligned} \tag{5.1}$$

A first observation consists of the fact that in (5.1) only fixed point operations take place. Then, mention should be made that, although more complicated in fixed point arithmetic, multiplication and division are now simpler than addition and subtraction. The latter imply a spread out execution which is formed of the exponents comparison of the (through subtraction), then the corresponding alignment of the significand numbers and, finally, the carrying out of the operation proper. An example in the more familiar decimal system is as follows:

$$\begin{aligned} 3.158 \cdot 10^{10} + 7.936 \cdot 10^8 &= \left(3.158 + 7.936 \cdot 10^{8-10}\right) 10^{10} \\ &= (3.158 + 0.07936) 10^{10} = 3.23736 \cdot 10^{10} \end{aligned}$$

From (5.1), it results that, for the exponents, only additions and subtractions are performed, while for the significands, all the fundamental operations are performed, from addition to division. Obviously, an ALU structure of the type presented in Fig. 5.1 could cover all the above mentioned operations, but it would suppose their execution in a serial manner, this resulting in performance degradations. Therefore, the solution usually used for the arithmetic part of a floating point ALU consists of in appealing to a configuration with two subunits, one of them dedicated to the exponents, and the other one dedicated to the significands. The communication between the two of them can be ensured only by means of a bus, this solution being

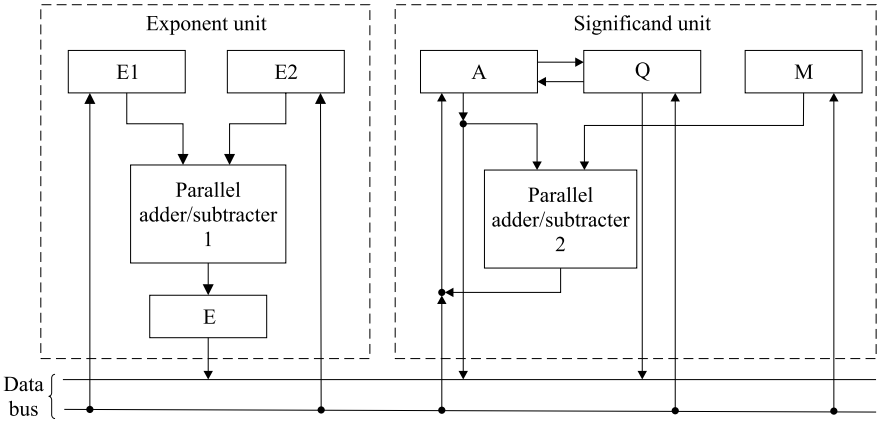


Fig. 5.3 Block diagram corresponding to the loosely coupled version of a floating point ALU

called "loosely coupled", or, in general, besides the bus lines, by means of a small number of direct connections, that assure a rapid but expensive communication, this solution being called "tightly coupled". In Fig. 5.3 is presented a block diagram that corresponds to the loosely coupled solution, while in the following section we will present a tightly coupled version for a floating point addition unit [Haye98]. The figure presents the exponent unit consisting of three registers, two of them (E_1 and E_2) for the input exponents (X_E and Y_E) and one (E) for the result exponent, and a parallel adder/subtracter (the parallel adder/subtracter 1), as well as the significand unit whose configuration is similar to that presented in Fig. 5.1.

But, floating point operation is faced, with a major problem, namely the rounding which is necessary to convert high precision values or results of intermediary computations with additional digits to formats of lower precision for memorizing purposes and/or output of final results. According to the IEEE 754 standard [Kaha97, ***08], there are four rounding modes [Parh00, EvSe00]: round to nearest even number, for short round to nearest, round toward 0 (inward), round toward $+\infty$ (upward), and round toward $-\infty$ (downward).

To familiarize the reader with the rounding modes, we shall consider, without loss of generality, the number X, with its integer and fraction parts in the following forms: $X = x_{n-1}x_{n-2}\ldots x_1x_0x_{-1}x_{-2}\ldots x_{-m}$, which has to be rounded to the integer value $X^* = x^*_{n-1}x^*_{n-2}\ldots x^*_1x^*_0$. The simplest method is represented by truncation or chopping, which consists of the giving up of the less significant m bits, the result being $X^*_1 = x_{n-1}x_{n-2}\ldots x_1x_0$. On the other hand, it is important to notice that the effect of truncation on numbers represented in sign-magnitude form is different from that on the numbers represented in two's complement form. The differences are highlighted by the representations from Fig. 5.4 [Parh00]. In Fig. 5.4a is shown the effect of truncation on a number represented in sign-magnitude form, and it can be observed that the magnitude of the truncated number, denoted by $chop(X)$, is always smaller, in its absolute value, than the non-truncated number X, a fact due to which the rounding type is called "round toward 0". By contrast, in Fig. 5.4b is

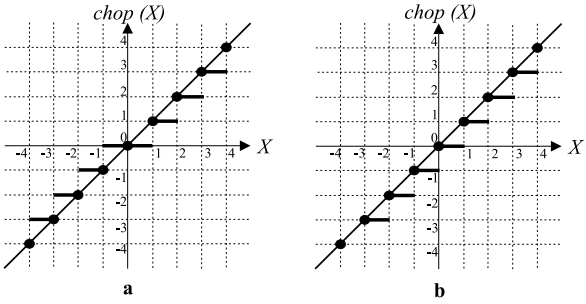


Fig. 5.4 Representations intended to mark out the "round toward 0" (**a**) and the "round toward $-\infty$" (**b**) rounding modes

shown the effect of truncation on a number in two's complement form, and it can be observed that the value of the truncated number, $chop(X)$, is always smaller than that of the non-truncated number X, a fact due to which the rounding type is called "round toward $-\infty$."

Regarding the "round to nearest" rounding mode of X, denoted as $rtn(X)$, if to a positive integer X is added a fractional part smaller than $(1/2)$, then the rounded value of X remains unchanged, while the addition to X of a fractional part equal or greater than $(1/2)$ determines the rounding of X to the following integer value greater by one unit, as highlighted in Fig. 5.5a [Parh00]. By modifying the sign of the landmark value $(1/2)$ (into $(-1/2)$), the previously enunciated rule can be extended over the whole range of numbers X with sign-magnitude representation (Fig. 5.5a). But, if the numbers are represented in two's complement, then a modification in Fig. 5.5 occurs only to the negative values of X, i.e. the dots will move from the ends on the right side of the bolded lines to those on the left side. Whatever the representation format, sign-magnitude or two's complement, the assigning of the value from the middle of the interval (the dots from Fig. 5.5a for instance) to a certain fractional part (in case of Fig. 5.5a, to the part greater than $(1/2)$),results in a certain imbalance which can create problems through accumulation of errors. In order to bring out the effect of this imbalance, let us suppose that we want to round number $X = x_{n-1}x_{n-2}\ldots x_1x_0x_{-1}x_{-2}$ to its integer value $X^* = x^*_{n-1}x^*_{n-2}\ldots x^*_1x^*_0$. Depending on the values of the two bits from the fractional part, we have the following four cases together with the associated ε errors: $x_{-1}x_{-2} = 00$, downward rounding (since $x_{-1} = 0$, consequently $x_{-1}x_{-2} < (1/2)$) and $\varepsilon = 0$; $x_{-1}x_{-2} = 01$, downward rounding (since $x_{-1} = 0$, consequently $x_{-1}x_{-2} < (1/2)$) and $\varepsilon = (-1/4)$; $x_{-1}x_{-2} = 10$, upward rounding and $\varepsilon = (1/2)$; $x_{-1}x_{-2} = 11$, upward rounding and $\varepsilon = (1/4)$. If the four cases are equiprobable, there is an average error $\varepsilon_{mean} = (1/8)$. If, in practice, it is proved that the probability to obtain the value corresponding to the middle of the range $(x_{-1}x_{-2} = 10)$ is greater than the others, this results in a greater average error with respect to that computed $(\varepsilon_{mean} = (1/8))$.

A way of surmounting the above problem is represented by always rounding to an even (or odd) integer, by rounding the middle values $(x_{-1}x_{-2} = 10)$ upward and downward with the same probability. In Fig. 5.5b [Parh00] is presented the rounding of X numbers, represented in sign-magnitude, to the even integers through the "round to nearest even" mode, values noted by $rtne(X)$. Regarding the numbers

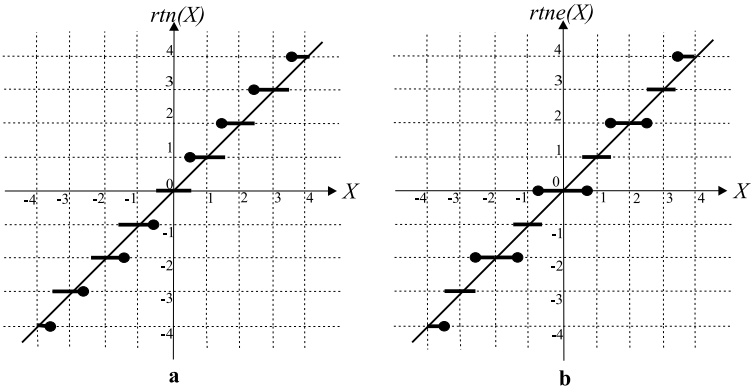
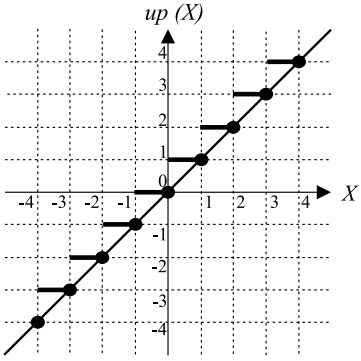


Fig. 5.5 Representations intended to mark out the "round to nearest" (**a**) and the "round to nearest even" (**b**) rounding modes

Fig. 5.6 Representation intended to mark out the "round toward $+\infty$" rounding mode



in the two's complement format, the dependence from Fig. 5.5b is unchanged, since, for instance, (-1.5) is rounded to (-2) corresponding to both representations. The IEEE 754 standard prefers the solving of the "halfway" cases by resorting to "round to nearest even" mode, and not "round to nearest odd" mode.

Finally, according to IEEE 754 standard, there is one more rounding mode, namely "round toward $+\infty$" (Fig. 5.6 [Parh00]), characterized by the fact that the rounded value of a number X, which is denoted by $up(X)$ (from upward-directed rounding), is obtained by always choosing the value of the integer upper end of the range to which X belongs.

In certain applications, such as those corresponding to interval arithmetic [ErLa04, Kuli02, Parh00], it is necessary that the computation errors be forced in a certain known direction. For instance, if we want to assess an upper bound for a certain quantity, then only the values which are greater than the correct one are accepted, which corresponds to upward rounding, i.e. "round toward $+\infty$". Obviously, in a similar way, if we want to assess a lower bound for a certain quantity, then only the values which are smaller than the correct one are accepted, which

$$\left.\begin{array}{l} X_M^* = x_{m-1}x_{m-2}.....x_ix_{i-1}.....x_1x_0 \\ Y_M^* = y_{m-1}y_{m-2}.....y_iy_{i-1}.....y_1y_0 \end{array}\right\} + \quad \text{before mantissa alignment}$$

$$\left.\begin{array}{l} X_M^* = x_{m-1}\,x_{m-2}.....x_ix_{i-1}.....x_1\,x_0 \\ Y_M^* = \longrightarrow\ y_{m-1}y_{m-2}.....\,y_{m-i}y_{m-i-1}y_{m-i-2}y_{m-i-3}y_{m-i-4}.....y_1y_0 \end{array}\right\} + \quad \begin{array}{l}\text{after mantissa}\\ \text{alignment}\\ X_E - Y_E = m\text{-}i\text{-}1\end{array}$$

$$Z_M^* = \overbrace{z_{m-1}}\,\overbrace{z_{m-2}}\ldots\ \overbrace{z_i}\,\overbrace{z_{i-1}}\ .....\ \overbrace{z_1}\,\overbrace{z_0}\quad \overbrace{g}\quad \overbrace{r}\quad \overbrace{s = y_{m-i-4}\underline{or}\ldots\underline{or}\,y_1}\,\underline{or}\,y_0$$

Fig. 5.7 Highlighting the *g, r* and *s* rounding bits

corresponds to, the previously mentioned downward rounding, i.e. "round toward $-\infty$" (Fig. 5.4b). Both rounding modes are applied in interval arithmetic.

Below, let us analyse in more details the performance of rounding [Kuli02]. Thus, if reference is made to addition/subtraction, according to (5.1), the operation proper is preceded by the previously mentioned comparison of exponents and the necessary alignment of the significands. If we suppose that the dimension of the mantissa field is of m bits and if we have, as in (5.1), $X_E \geq Y_E$, then, in the case when $X_E - Y_E \geq m$, the sum result will be equal to one of the operands, in our case to X. However, if $X_E - Y_E < m$, then, following the right shifting of an operand—in our case Y—there shall be executed the operation proper, which, in case it is executed in an exact mode, may require, in the worst case, an adder of $(2m - 1)$ bits, which represents a major disadvantage due to its cost, and still more to its performance. But the adopted solution is not based on the ideal algorithm consisting of the exact sum assessment and the subsequent rounding. An adder of only m bits will be used and a number of additional bits will be attached to the shifted operand and situated to the right of the non-shifted operand's least significant bit; these bits will be obtained during the alignment shift and will be used for rounding purposes. The rounding shall be executed without precision loss, relative to the operation execution according to the ideal algorithm. The number of additional bits to the m bits of the mantissa is three, noted by g, r and s. The first bit, situated, following the shift, to the right of the lsb of the shifted mantissa—denoted Y_M^*—, is called the guard bit and is denoted by g. It is the last bit which, during the process of alignment of the mantissas, leaves the m bits initially assigned to Y_M^*. Leaving aside the operands' signs, in Fig. 5.7 is presented the situation of the two mantissas before and after the alignment process, where it has been assumed that the right shift has been made with $X_E - Y_E = m - i - 1$ bits and g coincides with y_{m-i-2}.

The second bit, situated immediately to the right of g (in the example case from Fig. 5.7 it is y_{m-i-3}), is called the round bit and is denoted by r. It is the bit which leaves the m bits initially assigned to Y_M^* before g. Finally, the third bit, called the sticky bit s, has the value obtained as a result of an OR logic operation, executed, during the shift process, among the less significant, in our case, $(m - i - 3)$ bits of Y_M^* and, in general, among the bits that leave the m bits initially assigned to Y_M^*, except g and r.

If an alignment is executed to the right by one bit position, g will retain the shifted bit, "guarding" against precision loss. Following the process of alignment by two positions right shift, the significand of the shifted operand will have magnitude belonging to the value range $[0, (+1/2))$. Since the significand of the non-shifted operand is included, as is already known, within the $[(+1), (+2))$ range, the worst situation occurs when the difference between the non-shifted significand and the shifted one, belongs to the range $((+1/2), (+2))$. Consequently, firstly, it follows that, in order to normalize the result, this has to be shifted to the left by one binary position at the most, so that, secondly, the g bit is adequate to assure protection against precision loss in this case as well.

In case of normalizing the result with one binary position left shift, thus g becoming the mantissa's lsb, and when "round to nearest" rounding mode is applied, the rounding direction (upward or downward) of the resulting significand is determined by the value of the r bit. More precisely, if $r = 0$, i.e. the part eliminated through the alignment right shift is smaller than $(1/2)$, the rounding will be made downward, and if $r = 1$, the part eliminated through the alignment right shift is greater or equal to $(1/2)$, and the rounding is made upward. It also needs to be established whether the part eliminated through the alignment right shift is exactly equal to $(1/2)$ or not, because some of the rounding modes use this information. It is given by the s bit which is the result of an OR operation on all the bits from the right side of r and indicates the fact that, when $r = 1$ and $s = 1$, the magnitude of the eliminated part exceeds $(1/2)$, and when $r = 1$ and $s = 0$, the respective magnitude is exactly equal to $(1/2)$. In other words, if during the alignment right shift, 5 bits extend beyond the lsb of the non-shifted operand, then s will be set to the value that results following an OR operation on the 3 bits following the g and r positions.

Let us suppose that following the execution of an operation, such as an addition, the preliminary significand Z_M is obtained with the corresponding mantissa Z_M^*. Since Z_M can be obtained in unnormalized form, Z_M has to undergo the corresponding shifts for normalization purpose before passing to the rounding operation, with the compulsory exponent adjustments, accordingly. Following the normalization process, the bits of interest for the rounding are the round bit—situated immediately to the right of the lsb of the normalized significand—denoted by R, and the sticky bit—representing the result of an OR operation among all the bits situated to the right of the lsb of the normalized significand, except the first one (i.e. R)—denoted by S.

If all the potential normalization cases are taken into account, the expressions for R and S given in Fig. 5.8 can be obtained. Thus, the less significant bits (z_1 and z_0 mantissa of Z_M^* from Fig. 5.7 are presented before and after the one position right shift (Fig. 5.8a), the one position left shift (Fig. 5.8b), the two positions left shift (Fig. 5.8c), a case which also corresponds to several positions left shift, and the situation when Z_M is already normalized, its shifting not being necessary (Fig. 5.8d). It can be observed, for instance, that when Z_M^* is shifted during the normalization process by one bit to the right (Fig. 5.8a), R becomes equal to the lsb (z_0) of the non-shifted significand Z_M, and S is obtained through an OR operation executed between the old s, g and r bits.

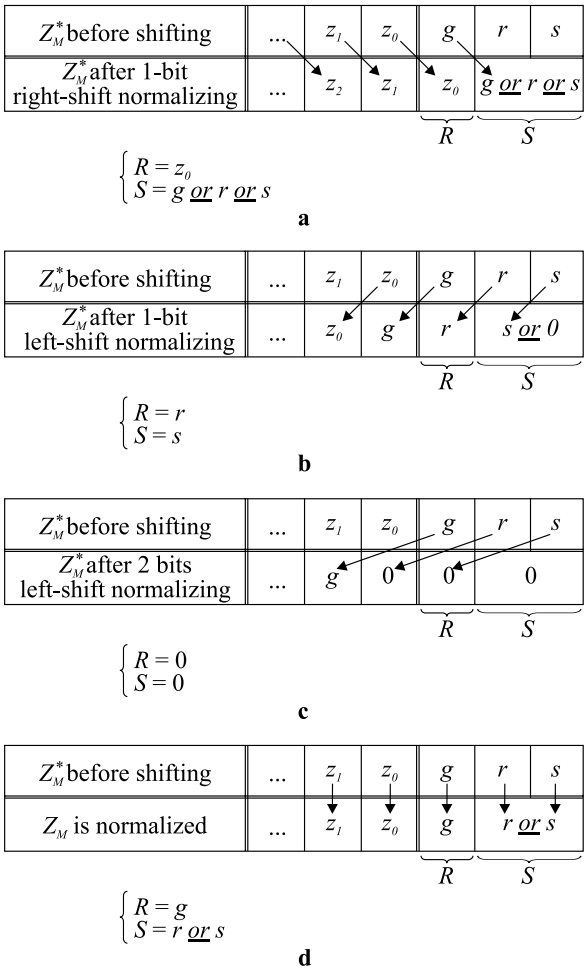


Fig. 5.8 Determining the values of the round bit R and the sticky bit S by taking into account potential cases of normalization

A special case is represented by the one in Fig. 5.8c where rounding is not necessary because the situation when Z_M is shifted, during normalization, by two or more bits to the left corresponds to the obtaining of the exact sum significand, with the conservation of full precision. For instance, as can be seen for the case f of Fig. 5.12, this special situation corresponds to the addition of two numbers of different signs and with near absolute values, when the alignment of the shifted operand implies only a one position right shift, a situation in which the r and s bits are 0. The analysis constructed for addition can be extended for other operations as well, for instance multiplication will be treated in Sect. 5.3.

These observations with regard to R and S bit adjustments being made, we shall pass to the effective implementation of the rounding modes. Thus, if we have a positive normalized result, then it can be rounded "to nearest even" by testing, first of all, the value of the R bit, which, if equal to 0, leaves the result unchanged. The same

Rounding Mode	$Z_{mn} \geq 0$	$Z_{mn} < 0$
toward $-\infty$		if $(R$ or $S = 1)$ then $Z_{Mn} - 1$
toward 0		
toward $+\infty$	if $(R$ or $S = 1)$ then $Z_{Mn} + 1$	
toward nearest even	if $(R(z_{0n}$ or $S)=1)$ then $Z_{Mn} + 1$	if $(R(z_{0n}$ or $S)=1)$ then $Z_{Mn} - 1$

Fig. 5.9 The conditions for the rounding operations

situation is obtained if the lsb of the normalized result, denoted by z_{0n}, and the S bit are both 0, since, if $z_{0n} = 0$, the normalized result is even, and it has to remain even, as well, according to Fig. 5.5b, when $R = 1$ and $S = 0$ too. Otherwise, by considering things in reversed order, if the logic condition $R(z_{0n}$ or $S) = 1$ is fulfilled, then for rounding purpose, a binary unit is added to the value of the normalized significand, which is denoted by Z_{Mn}, namely to the lsb position z_{0n} of Z_{Mn}. A similar conclusion is obtained regarding the same rounding mode for negative normalized significand numbers as well, but, in this case, the binary unit has to be subtracted from Z_{Mn} [Omon94].

On the other hand, if we refer to the rounding mode "toward $+\infty$", then, according to Fig. 5.6, the addition of a binary unit to the lsb position z_{0n} of the normalized significand Z_{Mn} takes place when either of the bits situated to the right of the z_{0n} are 1, namely when the logic condition R or $S = 1$ is fulfilled. The same logic condition need to be fulfilled when referring to the rounding mode "toward $-\infty$", when, according to Fig. 5.4b the binary unit have to be subtracted from Z_{Mn}. The synthesis of what has been presented with regarding the rounding modes corresponding to IEEE norms, enables us to obtain the rules given in the table from Fig. 5.9 [HePa03]. A blank field in the table means that the result obtained following normalization remains unchanged. Mention should also be made that, by performing the rounding according to the rules from Fig. 5.9, it is possible to obtain a carry-out (c_{out}) from the msb of the adder, which implies one more shifting, to the right, of the rounded significand, together with the corresponding adjustment of the exponent.

Besides the rounding problem, the floating point operation also confronts that of the exceptions handling. As far as this aspect is concerned, the IEEE standard defines five exceptions associated with overflow, underflow, divide-by-0, invalid operation, or inexact result [HePa03, Parh00]. On their occurrence, each of these exceptions sets up a flag and returns a special value of $\pm\infty$, ± 0 or NaN (not a number) type, with which the computations go on. In case such a special value occurs as operand in an arithmetic operation, its result is specified on the basis of some defined rules which are part of the standard. Thus, for instance, NaN + ordinary number = NaN, $(\pm\infty) \cdot$ ordinary number $= \pm\infty$ or ordinary number/ $(\pm\infty) = \pm 0$. As has already been seen, special codes are assigned for special values, thus enabling the propagation of the exceptions to the end of the computations, so as not to cause

stopping or aborting. The implementation of the exceptions mechanisms may include, for each of the above mentioned five types, one exception handling program, a so-called trap handler [ErLa04], which can be appealed to when it is authorized.

The overflow and underflow exceptions can easily be detected through diagrams after the exponents' adjustment. The overflow may occur only when the execution of a right shift, for normalization purposes, is required, or when a right shift is required by a rounding that resulted in a carry-out. On the other hand, the underflow may occur only when normalization requires left shifts.

As concerns divide-by-0, the flag associated with this exception is set to 1 if an ordinary number has to be divided by 0, a situation when, as a function of the sign of the number, $(+\infty)$ or $(-\infty)$ will be returned. If square root has to be performed on a negative operand, then the "invalid" flag will be set to 1, and, for the computations going on, NaN is returned. As a matter of fact, the invalid operation exception occurs not only for the square root of a negative number, but also on addition, when $(+\infty) + (-\infty)$ occurs, or on multiplication, when $0 \cdot \infty$ occurs, as well as on division, for $0/0$ or ∞/∞. Usually, the "invalid operation" exceptions return the NaN special value and they are treated by the unpacking and packing circuits from, respectively the packing circuits into the IEEE standard format.

Finally, the "inexact" exception, an unusual one, has to be signalled when the result of an operation or of a conversion cannot be exactly represented and has to be rounded. However, such situations occur quite frequently and therefore "inexact" is not really an exceptional condition. As an example, let us suppose that we have $m = 4$ bits assigned for the mantissa and that we have to multiply $(1.0011 \cdot 2^{-2})$ by $(1.1001 \cdot 2^{X_{Emin}})$ obtaining the product $11.1011011 \cdot 2^{X_{Emin}-3}$. Since $m = 4$ and, on the other hand, $R = 0$ and $S = 1$, the obtained rounded result is $0.0111 \cdot 2^{X_{Emin}}$, which determines the setting to 1 of the "inexact result" flag. In particular, it can also be observed that the obtained result is denormalized [ScST05]. As has already been presented, the denormalized numbers are defined without the hidden 1 bit and as having the smallest possible value for the exponent, being adopted to make the underflow effect less abrupt. Otherwise, when certain small values, which cannot be represented as normalized numbers, and, consequently, would have to be rounded to 0, are encountered during the computations they are represented, with a loss of precision, as denormalized numbers. Thus, we have a "gradual underflow" (sometimes called a "graceful underflow" [Parh00]), which requires special precautions when setting up the underflow exception flag. Mention should also be made of the fact that the implementation of the gradual underflow mechanism leads to performance and cost penalties, so that certain implementations do not accept denormalized numbers, choosing the faster, but less precise so-called "flush to zero" operation mode [ScST03].

A last observation, as far as exceptions are concerned, is made with reference to the detection of a zero result and its coding, because it cannot be represented as a number with normalized significand. In the IEEE standard format, zero consists of a sequence of 0 bits, except the sign one which may be positive (0) or negative (1).

Besides the problems of rounding and those of exception handling, there is also the problem of the representation precision. Thus, besides the formats on 32 bits,

denoted "single/short" precision, and on 64 bits, denoted "double/long" precision, the IEEE standard also defines extended formats [ErLa04, HePa03, Parh00]. They enable implementations with these formats to execute higher precision computations reducing the effect of accumulated errors. The two extended formats consist of "single-extended" with ≥ 11 bits for the exponent and with ≥ 32 bits for the significand (the bias is not specified, but the exponent range has to be included in the $[(-1022), (+1023)]$ range) and of "double-extended" with ≥ 15 bits for the exponent and with ≥ 64 bits for the significand (the bias is not specified, but the exponent has to be included in the $[(-16382), (+16383)]$ range). As far as extended formats are concerned, mention should be made that they prove their utility in the control of error propagation when several arithmetic operations are executed in sequence. Thus, let us suppose that we have to add several floating point numbers, a situation in which, to reduce computational errors, it is favorable to add, on the one hand, the positive values and, on the other hand, the negative values, and, finally, to subtract the two subtotals. If we have, for instance, to add a lot of numbers, it is possible to obtain overflow in the computation of one or both subtotals. But appealing to an extended format ("single extended" for simple precision operands, and "double extended" for double precision operands), the probability of overflow occurrence is greatly reduced [HePa03].

5.2 Floating Point Addition and Subtraction

5.2.1 Floating Point Addition and Subtraction Without Rounding

We prefer a gradual presentation of the problems regarding floating point addition and subtraction operations, not dealing, at the beginning, with aspects connected with rounding, as well as the unpacking of the operands from/packing the result into a standard format. Thus, we will assume a structure of the type presented in Fig. 5.3, in which we consider that the operands arrive in unpacked form (with explicit hidden bit), i.e. the bus dimension is of $(m + e + 2)$ bits, namely e for the exponent, m for the mantissa, one for the sign and one for the hidden bit. As compared to what was presented regarding the significand unit from Fig. 5.3, we also assume that not only the A register has the capacity of right-shifting (we suppose A can also support the left shift operation), but also the M register. The two registers, A and M, have each $(m + 2)$ bits, thus being able to store the significand numbers together with their signs. Register A has also, associated with it, a flag, `A_COUT`, which is implicitly set to 1, when a carry (c_{out}) is obtained from the msb of the adder/subtracter.

Regarding the exponent unit, the comparison of the contents of the two exponent registers, E1 and E2, each of them of e bits, is achieved using a subtraction, whose result is loaded in register E, also of e bits. This last register has the capacity to increment and decrement its content, operations encountered both in the alignment of the significand numbers, and in the result normalization, obviously, except on the occurrence of overflow or underflow situations. Mention should be made that in E is

```
                    adder/subtracter 1
                    declare register A[(m+1):0], M[(m+1):0], E1[(e-1):0], E2[(e-1):0],
                                     E[(e-1):0], A_COUT, ERROR;
                  declare bus INBUS[(e+m+1):0], OUTBUS[(e+m+1):0];
        BEGIN: A_COUT:=0, ERROR:=0,
        INPUT: E1:=INBUS(X_E), A:=INBUS(X_M);
               E2:=INBUS(Y_E), M:=INBUS(Y_M);
      COMPARE: E:=E1 - E2;
        ALIGN: if E<0 then A:=right-shift(A), E:=E+1,
                 go to ALIGN;
               if E>0 then M:=right-shift(M), E:=E-1,
                 go to ALIGN;
ADD/SUBTRACT: A:=A ± M, E:=max(E1, E2);
     OVERFLOW: if A_COUT=1 then begin
                 if E=E_MAX then go to ERROR,
                 A:=right-shift(A), E:=E+1, go to END; end
         ZERO: if A=0 then E:=0; go to END;
  NORMALIZE: if A is normalized then go to END;
 UNDERFLOW: if E>E_MIN then A:=left-shift(A), E:=E-1,
              go to NORMALIZE;
        ERROR: ERROR:=1;
          END:
```

Fig. 5.10 Description of the addition/subtraction without rounding implemented on a loosely coupled floating point ALU

loaded, following the alignment completion, the value of the greater exponent from among the contents of E1 and E2, which test is supposed to be done in a hardwired mode.

Following these remarks, Fig. 5.10 (adapted from [Haye98]) presents in the usual description language the code sequence corresponding to the addition/subtraction procedure, from which have been totally omitted the problems of rounding, all the bits which, through shifts, leave the registers being considered truncated, lost ones. Mention should be made of the fact that in the description, when loading the registers with the input operands (the INPUT statements), the INBUS bus bits that contribute to the exponent part and to that of the mantissa have not been detailed, only the operands being specified (X_E and X_M, and Y_E and Y_M, respectively). Also, the left and right shifting operations, as well as those for verifying that the result is normalized have not been explicitly presented. We have resorted, in the same way, to other simplifications as well, such as the exceptions that have been summed up only in some possible situations of overflow and underflow, as well as in zero result situations. Thus, one single flag, ERROR, is provided which is set to 1 when a right shift of the result significand in A is required and the exponent already has the maximum value that can be represented, i.e. E_{MAX}.

Referring to the procedure presented in Fig. 5.10, following the loading of the operands, and before the exponents comparison, it is recommended to perform a zero test on both significand operands, because if either is zero then the result may be anticipated without going through the whole procedure. As for the rest, the elementary operations are connected according to (5.1), namely that following the comparison—through subtraction—of the exponents, the alignment of the mantissas is performed, an operation for which an intervention in the procedure to stop

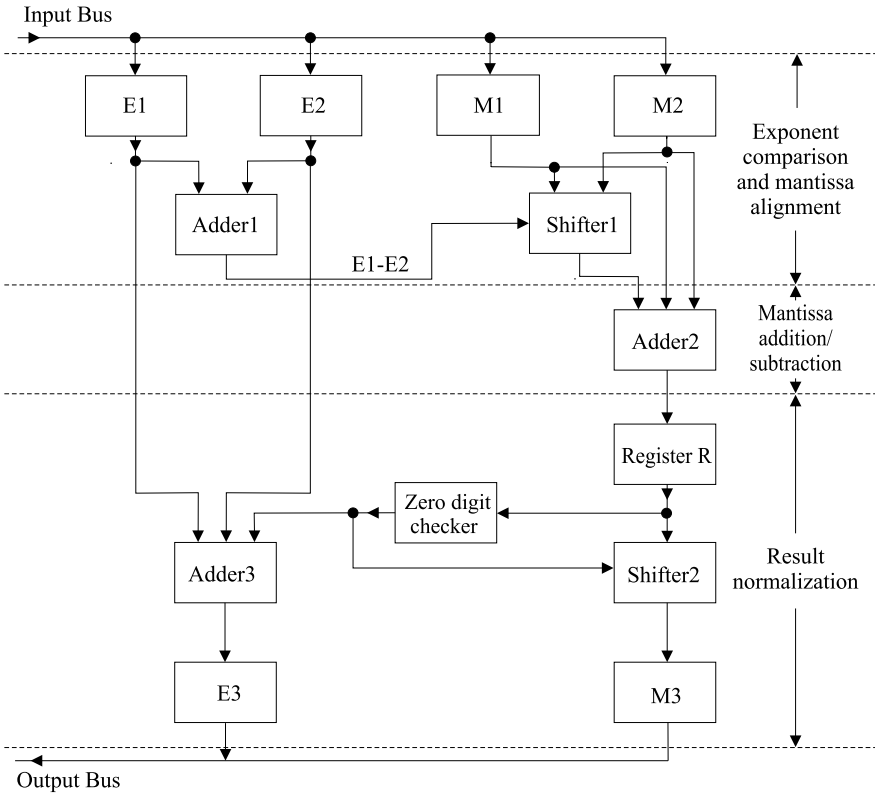


Fig. 5.11 Tightly coupled floating point adder/subtracter with hardware mechanisms for speeding up the addition/subtraction process

the shifts for alignment purposes is possible after $(m + 1)$ steps and not after $|E|$ steps. This is because we may have $|E| > (m + 1)$ and then the number of steps given by the difference $(|E| - m - 1)$ is useless, the shifted significand becoming, anyway, zero. The procedure being followed after the alignment of the significand numbers, the operation proper will be executed, which is followed by normalization of the result for its packing, with the necessary testing for exception cases. To this description of the addition/subtraction procedure, there corresponds, almost exactly, the block diagram of a famous computer in the age of medium and large capacity systems, of mainframe type, namely the IBM/S360 model 91. This machine includes two separate floating point units, one intended for addition and subtraction (add unit), and the other for multiplication and division (multiply/divide unit) [Haye98]. We shall refer only to the add unit which enables the addition/subtraction of numbers on 32 bits and on 64 bits. The reason for introducing the block diagram corresponding to this unit (Fig. 5.11 [Haye98]) in this chapter is to highlight some hardware mechanisms for speeding up the addition process, besides the fact that it represents a synthesis of what is shown in a more detailed way in Fig. 5.10.

Thus, the two exponents are stored in registers E1 and E2, and the difference between their contents, assessed using Adder 1, determines which of the mantissas (the notion of significand was not yet known), stored in registers M1 and M2, has to be shifted to the right for alignment purposes. Mention should also be made of the fact that the radix of the number system, corresponding to the whole product line to—which belongs the above—mentioned computer, is $r = 16$. If it results that $E1 - E2 = k$, because k is a hexadecimal number, the shift to the right has to be made by $4k$ binary positions. This whole process of mantissa alignment is accelerated through a combinational shifting circuit, denoted Shifter 1, which enables the simultaneous execution of this operation for the $4k$ bits. Then, the shifted mantissa is added to/subtracted from the non-shifted mantissa using Adder 2, representing an adder/subtracter on 56 bits (the value that results from the subtraction of the sign bit and of the 7 bits assigned to the exponent field from the dimension of the 64 bit format) implemented with several carry-lookahead levels. The preliminary sum/difference is temporarily stored in register R, whose content is investigated by the combinational circuit called the Zero digit checker, having the function to determine the number of hexadecimal digits equal to zero, the so-called leading zeros, that must be eliminated through the normalization process. The value established through this second structural element intended for the acceleration of addition, constituted by the Zero digit checker, determines, on the one hand, the number of hexadecimal positions by which the sum/difference preliminary mantissa from R has to be left shifted. On the other hand, the number determined by the Zero digit checker has to be subtracted, using Adder 3, from that found to be greater in the comparison of the exponents, thus completing the normalization process with reference to the exponent by storing its final value in E3. Regarding the mantissa from R, its left shifting by the number of positions established using the Zero digit checker is ensured by a third structural acceleration element represented by the shifting combinational circuit Shifter 2, at whose outputs the final mantissa is obtained, which is stored in register M3. Obviously, the three acceleration circuits, as well as the radix $r = 16$, determine some of the differences between the block diagram from Fig. 5.11 and the description from Fig. 5.10, to which there are added the direct communication lines between the exponent and the mantissa units, outlining a rather tightly coupled technical solution as compared to one where the communication is predominantly done via a bus.

5.2.2 Floating Point Addition and Subtraction with Rounding

Before presenting the addition procedure, we resume our examination, by using examples, of rounding problems. We recall that given, for instance, mantissa X_M^* of m bits and significand X_M with sign and having $(m + 2)$ bits, then the sum/difference result Z will have $(m + 6)$ bits, being of the form $z_c z_{h1} z_{h0} . z_{m-1} z_{m-2} \ldots z_1 z_0 grs$, where, from left to right, z_c represents the potential carry-out bit (A_COUT in Fig. 5.10), z_{h1} and z_{h0} represents the value resulting from the operation on the hidden and sign bits of the two significand numbers, $z_{m-1} z_{m-2} \ldots z_1 z_0$ represent the

m bits assigned to the mantissa and, finally, *g*, *r* and *s* represent the known guard, round and sticky bits used for rounding. As concerns these last three, in Fig. 5.12 are given illustrative examples of their use in the rounding process.

We can observe the comparison, it being the primary aim, between the results of the operations, first executed exactly, with infinite precision, and subsequently rounded (column "Operation with infinite precision and followed by rounding" in Fig. 5.12) and the results of the operation with direct rounding (column "Operation with rounding" in Fig. 5.12), executed on the bits length of an operand to which the three bits *g*, *r* and *s* are added. The examples from the "Operation with rounding" column particularly highlight the changes in the *g*, *r* and *s* bit positions. However, mention should be made that in order to obtain the *g*, *r*, and *s* rounding bits, the steps of the procedure to be described below were not applied ad litteram. Moreover, a top-down examination of the example operations, from case *a* to case *f*, highlights the fact that corresponding to a decreasing difference between the two floating point numbers, the positions of the rounding bits (*g*, *r* and *s*) change. But mention should be made that in order to obtain the *g*, *r*, and *s* rounding bits, the steps of the procedure that will be described below have not been applied ad litteram. The motivation for the presented examples, from case *a* to case *f*, was to render evident the mechanisms by which the relevant bits with respect to the rounding (*g*, *r*, and *s*) are modified as the difference between the two floating point numbers decreases. Nonetheless, all the results in the "Operation with rounding" column are consistent with those obtained by applying ad litteram the steps of the algorithm. In Fig. 5.12 there are also presented, in the last column, the ε errors consisting of the difference between the rounded result values and those corresponding to the exact operation (in the column "Operation with infinite precision and followed by rounding"), both of them computed in decimal, in accordance with the IEEE 754 standard's rules in order to find a systematic correlation between the decreasing difference of the operands and the error ε. As it can be observed, such a correlation does not exist.

In connection with the guard bit *g*, the example cases *a* to *e* highlight its necessity because the first bit of the intermediate result becomes 0 and, through the successive normalization operation, one position left shifting is required, thus *g* becomes the lsb of the normalized result. Instead of the old *g* bit we have, through left shifting, *r*, while *r* will be substituted by *s*.

Mention should be made that for the examples presented in Fig. 5.12 (except case *f* where the result is exact), "toward nearest even" rounding mode (Fig. 5.9) has been used. We now recall that *s* binary values have been obtained through OR operation of all the right-shifted (in the alignment process) significand bits (except the *g* and *r* bits) which are situated to the right of the non-shifted operand's lsb.

Hereinafter we point out that example *f* (Fig. 5.12) can be used to answer the hypothetical question whether only one *g* guard bit is sufficient. As the difference between the two operands decreases, it can be observed that in the first instance two 0 bits (subsequently, more 0 bits) occur in the most significant positions of the intermediate result, and, at a first analysis, it would require more than one guard bit. However, this is not confirmed because, when the operands are of close absolute values, the result representing the difference becomes exact (referring to the

Case	Example operation	Operation with infinite precision and followed by rounding	Operation with rounding	Error ε
a	$1.000 \cdot 2^{-6}$ $-1.011 \cdot 2^{-6}$	$1.000 - 0.000001011 = 0.111100101$ $1.111 + 1\ (r\ s) = 10.000 \rightarrow 1.000$ c_{out} $\left(\frac{1 - 11/512}{501/512}\right)$ $(1 \cdot 2^{-1} \cdot 2^{1})$	$1.000 - 0.000001 = 0.1111111$ $1.111 + 1\ (g\ r\ s) = 10.000 \rightarrow 1.000$	$1 - \frac{501}{512} = \frac{11}{512}$
b	$1.000 \cdot 2^{-5}$ $-1.011 \cdot 2^{-5}$	$1.000 - 0.00001011 = 0.1111001101$ $1.1110\ (r\ s) \rightarrow 1.111$ $\left(\frac{1 - 11/256}{245/256}\right)$ $\left((1+\frac{7}{8}) \cdot 2^{-1}\right)$	$1.000 - 0.000011 = 0.1111001$ $1.1110\ (g\ r\ s) \rightarrow 1.111$	$\frac{15}{16} - \frac{245}{256} = \frac{-5}{256}\ (= \frac{-10}{512})$
c	$1.000 \cdot 2^{-4}$ $-1.011 \cdot 2^{-4}$	$1.000 - 0.0001011 = 0.1110101$ $1.110\ (r\ s) + 1 = 1.111 \rightarrow 1.111$ $\left(\frac{1 - 11/128}{117/128}\right)$ $\left((1+\frac{7}{8}) \cdot 2^{-1}\right)$	$1.000 - 0.000101 = 0.1110101$ $1.110\ (g\ r\ s) + 1 = 1.111 \rightarrow 1.111$	$\frac{15}{16} - \frac{117}{128} = \frac{3}{128}\ (= \frac{12}{512})$
d	$1.000 \cdot 2^{-3}$ $-1.011 \cdot 2^{-3}$	$1.000 - 0.001011 = 0.1101101$ $1.101\ (r\ s) \rightarrow 1.101$ $\left(\frac{1 - 11/64}{53/64}\right)$ $\left((1+\frac{5}{8}) \cdot 2^{-1}\right)$	$1.000 - 0.001011 = 0.1101101$ $1.101\ (g\ r\ s) \rightarrow 1.101$	$\frac{13}{16} - \frac{53}{64} = \frac{-1}{64}\ (= \frac{-8}{512})$
e	$1.000 \cdot 2^{-2}$ $-1.011 \cdot 2^{-2}$	$1.000 - 0.01011 = 0.10101$ $1.010\ (r) \rightarrow 1.010$ $\left(\frac{1 - 11/32}{21/32}\right)$ $\left((1+\frac{1}{4}) \cdot 2^{-1}\right)$	$1.000 - 0.010110 = 0.1010100$ $1.010\ (g\ r\ s) \rightarrow 1.010$	$\frac{5}{8} - \frac{21}{32} = \frac{-1}{32}\ (= \frac{-16}{512})$
f	$1.000 \cdot 2^{-1}$ $-1.011 \cdot 2^{-1}$	$1.000 - 0.1011 = 0.01001$ $1.010 \rightarrow 1.010$ $\left(\frac{1 - 11/16}{5/16}\right)$ $\left((1+\frac{1}{4}) \cdot 2^{-2}\right)$	$1.000 - 0.101100 = 0.0101000$ $1.010\ (g\ r\ s) \rightarrow 1.010$	$\frac{5}{16} - \frac{5}{16} = 0$
g	$1.100 \cdot 2^{1}$ $+1.011 \cdot 2^{-1}$	$1.100 + 0.1011 = 10.00111$ $c_{out}\ 1.0001\ (r\ s) + 1 = 1.001 \rightarrow 1.001$ $\left(\frac{3/2 + 11/16}{35/16}\right)$ $\left((1+\frac{1}{8}) \cdot 2^{1}\right)$	$1.100 + 0.1011 = 10.00111$ $1.0001\ (r\ s) + 1 = 1.001 \rightarrow 1.001$	$\frac{9}{4} - \frac{35}{16} = \frac{1}{16}$

Fig. 5.12 Analysis of some edifying example cases for realizing the rounding process

example case *f* (Fig. 5.12), when *r* and *s* are 0 and *r* is shifted to the lsb position of the result, so that $\varepsilon = 0$ is obtained), and thus the rounding is no longer necessary. Consequently, in order to avoid loss of precision as compared to a result obtained exactly and then rounded, one guard *g* bit is necessary, but also sufficient.

The *g* example in (Fig. 5.12) highlights the possibility that when adding two significand numbers, carry (c_{out}) from the msb of the result may occur, which, for normalizing purposes, implies one position right shifting of the sum. But this determines a modification, in the reverse order as compared to the previous example cases, of the *g*, *r* and *s* bits, *r* being substituted by the lsb of the result, and the new *s* being obtained through an OR logic operation on the old *r* and *s* bits.

Passing below to the detailed description, in steps, of the floating point addition/subtraction algorithm, we will suppose *X* and *Y* numbers as input operands represented in compliance with the IEEE 754 standard.

Step 1 As the first operation, there will be performed, according to the previous presentation, the unpacking of the operands, which implies, for each operand, first of all, the separation of the sign, of the exponent, and of the mantissa with the explicit insertion of the hidden 1 bit of the significand, then the conversion of the operands to the internal format (the extended formats, single-extended or double-extended may be appealed to) and, finally, the testing for the special operands and for exceptions (for instance, the recognition of NaN inputs or of the adder bypassing cases when one or both operands are equal to 0).

Step 2 The computation of the difference between X_E and Y_E exponents follows in order to determine the quantity based on which the alignment operation of X_M and Y_M significands will be executed. To save circuitry, right shifting for alignment purposes is often performed for only one of the operands [HePa03, Parh00]. Thus, assuming that the only register used for the initial storage of Y_M has the presented preshifting capacity, then, if $X_E < Y_E$, the swapping of the operands is performed. In this way, the exponents' difference of $d = X - Y$ becomes ≥ 0 and the result exponent Z_E will be set to the value $Z_E = X_E$, according to (5.1).

Step 3 The operation in this step applies only in case the signs of the two operands, $sign(X)$ and $sign(Y)$, differ. This situation corresponds to the case when the content of one of the significand registers, let us assume the one provided with the capacity of right shifting, is substituted by its two's complement. By further discussing this aspect, we show that certain implementation solutions provide complementation logic in a selective mode (according to some authors [Parh00], if one operand is not preshifted, this is to shorten, in terms of time, the critical path). If, for instance, the operand is negative ($sign(X) = 1$) and only the register storing Y_M has complementation logic, where $sign(Y) = 0$, then Y_M is two's complemented and the sign of *X* is ignored. The result obtained is $(X - Y)$ instead of $(-X + Y)$, a fact which will be taken into account when establishing the sign of the final result. As concerns the described procedure, we assume that the register provided with preshifting capacity is also provided with complementation logic, the sign of the result being established,

when $sign(X) \neq sign(Y)$, at the end of the procedure, in step 9. If the signs of the operands coincide, then the common sign is ignored until the end when it is attached to the result.

Step 4 The register provided with complementation logic consists of $(m + 5)$ bits (1 (sign bit) + 1 (hidden bit) + m (bits assigned to the mantissa) + 3 (bits for rounding)). The significand from this register is right-shifted by $d = X_E - Y_E$ binary positions, an operation in which, if the two's complementation has taken place in the previous step, 1s instead of 0s will be introduced, evidently, through the left side of the register. As a result of this right shifting, in the first rounding bit, the guard bit, comes to be the last bit shifted out of the $(m + 2)$ bits of the register proper, in the second rounding bit, the r round bit, comes to be the bit which, through shifting, leaves the $(m+2)$ bits of the register before the bit that has become g, and in the third rounding bit, the sticky bit s, comes to be the result of the OR operation, assessed during shifting, all applied to the bits that leave the $(m + 2)$ bits of the register, except those which have become g and r.

Step 5 This step is dedicated to the computation of the preliminary result significand, Z_M, through the addition/subtraction of the $(m + 1)$ bits ((hidden bit) + m (bits assigned to the mantissa)) of the significand, possibly complemented (in step 2) and/or right-shifted (in step 3), to the $(m + 1)$ bits of the non-shifted significand. As a result of the preliminary addition/subtraction operation it is possible to obtain from the adder's msb a carry (c_{out}) which is required to be stored—for a subsequent test—in a flip-flop flag, to be attached to the memory ranks storing the result significand. Following the operation execution, it will be tested whether $sign(X) \neq sign(Y)$ when if, moreover, the msb of the result significand Z_M has become 1 without having generated carry-out, then Z_M is undoubtedly negative and has to be subject to an additional two's complementation. If the example cases a to f from Fig. 5.12 are transposed in terms of the described algorithm, it can be derived that any time c_{out} is generated, the msb of result Z_M is 0 (this msb may also be 1 if the first operand were, for instance, of the form 1.11...). The necessary condition for no c_{out} is for the msb of the complemented operand to be 0, but this implies no preshifting in step 4 or, in other words, $d = 0$. Moreover, when the conditions of the analysed case are observed, it can easily be seen out that the subtracted operand is, in its absolute value, greater than the minuend. Hence Z_M, which is negative, has to be substituted by its two's complement, to return to the expected sign-magnitude form (as is required by the IEEE 754 standard).

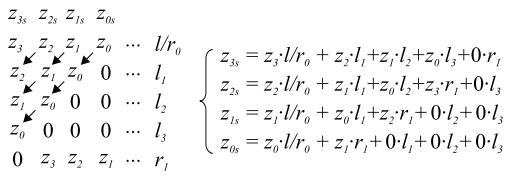
As concerns the adder implementation, a fast version of the carry-lookahead type is recommended that enables the execution of the operation in two's complement within a logarithmic time interval [Parh00]. The potential two's complement of Z_M obtained through the, in this work widely used, implementation of passing the complemented bits through the EXCLUSIVE-OR wordgate (which gates, having one of the inputs connected to 1, assures the one's complementing) applying them to an adder whose c_{in} input is also set to 1.

Step 6 This step is dedicated to pre-normalization operations on the result significand Z_M. This may not be the final normalization because, in a subsequent operation (step 8), carry-out may possibly be generated (refer to the example case *a*, Fig. 5.12), requiring an additional normalization operation.

Obviously, if $sign(X) = sign(Y)$ and if, in the addition/subtraction step, carry-out was generated (refer to the example case *g*, Fig. 5.12), then Z_M will be shifted by a binary position to the right, by introducing the carry-out bit as the msb of Z_M. It is also required to adjust—by increment—the value of the result exponent Z_E, as a consequence.

In case there is no agreement with the above described situation, Z_M shifts to the left until the normalized form of the significand is obtained. When the shift is made by one position to the left, the value of *g* will be introduced in the lsb of Z_M (refer to the example cases *a* to *e*, Fig. 5.12). If, for normalization purposes, it is necessary to shift Z_M to the left by two or more binary positions, then 0s will be introduced in the "tail" of *g* (refer also to case f, Fig. 5.12). Each of these situations requires the adjustment—by decrement—of the value of the result exponent Z_E, in the proper mode.

Below, we will refer to the implementation of the shifting mechanism for normalization purposes, which, in principle, is similar to that for the alignment of the significands, used in step 4. In this context, it is to be noted that the alignment mechanism is required to produce a shift to the right by *m* bits (we take into consideration the procedure's stopping condition when *d* is equal to or exceeds the length of the non-shifted operand), while the normalization mechanism has to produce either a shift to the right of one bit or a shift to the left by anything from 1 to *m* bits. Referring to the normalization shifter, one hardware implementation alternative could be the endowment of the result storage registers with bidirectional shift capabilities. However, the variable number of CLOCK pulses required, which can be as high as *m*, can become prohibitive for practical formats. This observation makes the above solution impractical in terms of the incurred costs. Another implementation option for the normalization shifter consists of using of two separate devices, one used for shifting to the right, the other for shifting to the left, but there is also the solution of combining the two functions by using a combinational shifter. In order to present the constructive characteristics of such a device, Fig. 5.13 presents the simplified case of a 4-bit register with ranks named as z_3 to z_0. The register's content can be shifted, as observed from Fig. 5.13a, by 1, 2 or 3 bits to the left, and by 1 bit to the right respectively. Each of these operations have an associated control signal, namely l_1, l_2 and l_3 for controlling the shift to the left by 1, 2 and 3 bits respectively, and r_1 for controlling the shift to the right by 1 bit, while l/r_0 corresponds to the binary value's conservation for a particular rank, for the case of that rank not being shifted. The Boolean equations corresponding to the shifted ranks z_{3s} to z_{0s} are depicted in Fig. 5.13b while a potential encoding of the selection signals for an implementation alternative using 5-to-1 multiplexers is given in Fig. 5.13c. Thus, from the three variables associated with the selection signals, *r* is used to distinguish the shift direction, with $r = 1$ for the generation of the signal $r_1 = 1$ which commands the 1 bit right shift and with $r = 0$ for the left shifts or for the no shift case. The left



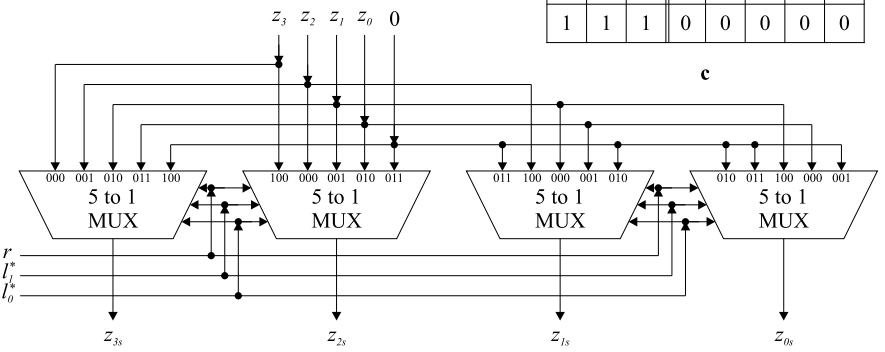
a

$$\begin{cases} z_{3s} = z_3 \cdot l/r_0 + z_2 \cdot l_1 + z_1 \cdot l_2 + z_0 \cdot l_3 + 0 \cdot r_1 \\ z_{2s} = z_2 \cdot l/r_0 + z_1 \cdot l_1 + z_0 \cdot l_2 + z_3 \cdot r_1 + 0 \cdot l_3 \\ z_{1s} = z_1 \cdot l/r_0 + z_0 \cdot l_1 + z_2 \cdot r_1 + 0 \cdot l_2 + 0 \cdot l_3 \\ z_{0s} = z_0 \cdot l/r_0 + z_1 \cdot r_1 + 0 \cdot l_1 + 0 \cdot l_2 + 0 \cdot l_3 \end{cases}$$

b

Inputs			Outputs				
r	l_1^*	l_0^*	r_1	l_3	l_2	l_1	l/r_0
0	0	0	0	0	0	0	1
0	0	1	0	0	0	1	0
0	1	0	0	0	1	0	0
0	1	1	0	1	0	0	0
1	0	0	1	0	0	0	0
1	0	1	0	0	0	0	0
1	1	0	0	0	0	0	0
1	1	1	0	0	0	0	0

c



d

Fig. 5.13 Implementation of a bidirectional normalization shifter in a 4-bit simplified version using multiplexers

shift amount is encoded by the binary combinations of the variables denoted by l_1^* and l_0^* (Fig. 5.13c), allowing the generation at each multiplexer's decoder output of the four signals: from $l_3 = 1$ (associated with a 3 bits left shift) to $l/r_0 = 1$ (when no shift is required). Using the content of the table in Fig. 5.13c, a potential implementation of the equations from Fig. 5.13b is represented by the technical solution using 5-to-1 multiplexers as presented in Fig. 5.13d. At each of the data inputs of the four multiplexers are marked the binary combinations for the triplet $(rl_1^*l_0^*)$ that open the multiplexer's channel for passing the input signal z_i (where $i = 0$ to 3) or 0 to the outputs z_{js} (where $j = 0$ to 3).

By extrapolating the suggested simplified solution in Fig. 5.13 to the number of bits employed by the IEEE 754 standard's formats it can be observed that this technical solution raises some difficult signal loading problems (see for example in Fig. 5.13d the fan-out corresponding to the z_0 and z_1 outputs of the result storage register). One possible solution to the loading problem can be achieved by the use of the multilevel multiplexers approach [Parh00], which, on the other hand causes additional latencies and increased power consumption. In any case, achieving a better performance/cost/power trade-off for a particular shifting device determines the number of levels for the multiplexers.

Another problem, specific to the shifting process, consists of the determining the number of bits to be shifted. As far as the shift for the alignment of the significands is concerned, problems occur only when the exponents difference d is negative, but by means of operands swapping, this can be avoided. Concerning the right shifting for normalization purposes, this is triggered, when $sign(X) = sign(Y)$, by the 1 status of the carry-out flag, set during the previous step. Obviously, if this flag is not set, and if the hidden bit is 1, no normalization shift is required. However, if the result from step 5 has leading 0s, these have to be eliminated through left shifting until the hidden position has a logic 1. From the technical point of view, the solution consists in a leading zero counter or, more elaborate, in the prediction of the leading 1 bit position of the result concurrently with the operation computation. In the latter case, the delay introduced by the leading 1 detector on the critical path is eliminated [ViLG06, ErLa04].

As a last remark connected with the technical solution for shifting, we show that the shifters, one for the alignment of the significand numbers, and the other for normalization, may be combined, for an economical implementation, in one device with bidirectional shifting capacity. If for performance purposes we aim at an arithmetic pipeline, then separate preshift and postshift devices are recommended [Parh00].

As also shown for the adding/subtraction algorithm without rounding (Fig. 5.10), through the right-shift for normalization purposes and the necessary incrementing of the result exponent Z_E, a larger than the maximal allowed biased value might be yielded (254 in single-precision, and 2046 in double-precision, respectively), which requires the signalling of the overflow exception status. Within the same context, if a denormalized value is obtained as the result, an amendment to the procedure is required, namely that the normalization operation through left shifting, with the corresponding Z_E exponent decrement, is carried out only until its biased value reaches the limit (1 being the still tolerated value). Under these circumstances, the msb of the result significand equaling 1 represents a normalized number, while the 0 msb stands for the case when the number becomes denormalized, which requires the signaling of this exception status. If following normalization, Z_E has obtained value 1 and, moreover, all Z_M bits have become 0, then the underflow exception status has been reached and it must be signalled appropriately [ScST05, Kore02].

Step 7 The operations in this step are dedicated to the adjustment of the values of round bit R and sticky bit S, with the aim of preparing the rounding in the following step.

Thus, if the result significand Z_M has been shifted, for normalization purposes in the previous step, by one position to the right, then, according to Fig. 5.8a, $R = z_0$ (R takes the value of the lsb of Z_M before shifting) and $S = g \ \underline{or} \ r \ \underline{or} \ s$, where g, r and s represent the values of the rounding bits before shifting (refer also to the example case g, Fig. 5.12).

But if the result significand Z_M did not require shifting for normalization purposes, then, according to Fig. 5.8d, $R = g$ and $S = r \ \underline{or} \ s$ where g, r and s are the old values of the rounding bits.

Fig. 5.14 Table for establishing the sign of the addition result for the cases when operands signs differ

swap	compl	*sign(X)*	*sign(Y)*	*sign(Z)*
Yes		+	-	-
Yes		-	+	+
No	No	+	-	+
No	No	-	+	-
No	Yes	+	-	-
No	Yes	-	+	+

If, in the normalization step, Z_M has been shifted by one position to the left, then, according to Fig. 5.8b, R and S maintain their old values, i.e. $R = r$ and $S = s$ (refer also to the example cases a to e, Fig. 5.12).

Finally, if Z_M has been shifted in the previous step by two or more positions to the left, then, according to Fig. 5.8c, $R = 0$ and $S = 0$ (refer also to the example case *f*, Fig. 5.12).

Step 8 This step is dedicated to the rounding of the result significand Z_M, which, having passed through normalization, will be denoted by Z_{Mn}. This operation is executed based on the table from Fig. 5.9, by using the R and S values determined in the previous step, and consists of the addition of a binary unit to the lsb position of Z_{Mn}. If the rounding causes a carry-out at the msb, then the new value of the significand has to be shifted to the right by one binary position and, also, the value of the result exponent Z_E has to be adjusted by incrementing. To this case there also applies the observation from step 6 related to the necessity to test the potential occurrence of the overflow exception status.

Step 9 The sign of the result will be added, in this step, to the finite values of the previously computed exponent and significand. Obviously, if the X and Y operands have the same sign ($sign(X) = sign(Y)$), this sign will also be the sign of the result ($sign(Z)$). If the signs differ ($sign(X) \neq sign(Y)$), and reference is made to addition, then the sign of the result ($sign(Z)$) will be obtained by means of the table from Fig. 5.14 [HePa03], as a function of the operands' signs and by taking into account whether swapping of the operands took place during step 2 (swap column) and whether two's complementation of the preliminary result significand Z_M took place during step 5 (compl column). The empty fields in the table represent the lack of any operation. A first comment concerning the table elements refers to the cases when swapping of the operands is needed, cases when $sign(Z)$ coincides with the sign of that operand of the two whose absolute value is greater, the operand whose significand will not be shifted in the alignment operation. On the other hand, when there is no swapping and no two's complementation in step 5, if reference is made, for instance, to the third row of the table, the significand of the operand Y will be two's complemented in step 3, and thus, it will have 0 in the hidden bit position.

Since the significand of the operand X has 1 in the hidden bit position and the complement of two is not performed in step 5, this means that in the addition c_{out} will be generated from the hidden bit position, the absolute value of the significand X_M being greater than that of the significand Y_M, therefore the sign of operand X (+) is preserved (this situation corresponds to the example cases *a* to *f*, Fig. 5.12).Through similar reasoning other result signs can be justified as presented by the table from Fig. 5.14.

Step 10 The result packing is performed during the last step of the procedure, which implies the removal of the hidden bit and the combination of the fields of the sign, exponent and significand, as well as testing for special values or exceptions (e.g. zero result, overflow or underflow). Special attention is dedicated to the conversion of the operands (at unpacking—step 1) and to the conversion of the results (at packing) between the ordinary and extended representation formats. Thus, the packing of a significand result an extended format, with several bits, into one with fewer bits, would require an additional rounding step. This might be avoided by using the procedure's rounding step to obtain the compressed result significand at the desired precision level [Parh00]. It is worth mentioning that if, following the normalization and rounding operations, the biased value of the exponent $Z_E \geq 1$ and if the hidden bit of the result significand Z_M is 1, then the packing is executed normally, with the omission of the respective 1 bit. However, if $Z_E = 1$ and the value of the bit from the hidden position is 0, the result is a denormalized number, and, in packing, its exponent field has to be set to 0.

Below, we illustrate the application of the steps of the presented algorithm on two examples described in parallel (Fig. 5.15). Both of them correspond to cases from Fig. 5.12, namely *c* and *g*, respectively. In example 1, we execute the passing, this time ad litteram, through the addition procedure. In example 2, as compared to case *g* from Fig. 5.12, the signs of the operands appear changed. Mention should be made that in Fig. 5.15, there appear some additional notations; they are obvious, besides those which have already been introduced, such as Y_{MC2} which represents the two's complement of significand Y_M, Y_{Ms} which represents the shifted value of Y_M for alignment purpose, as well as Y_{MC2s} which represents the two's complement shifted value of Y_M. Also, z_{0n} represents the lsb of the normalized result significand Z_{Mn}, a bit yielding the rounding mode "toward nearest even" used for the examples taken into account.

5.2.3 Speeding Up the Floating Point Addition/Subtraction Process

The problems related to the improvement of floating point addition and subtraction performance are tackled by analyzing the time-consuming operations from the algorithm described in the previous section. Among all the operations in the procedure, the additions and shifts appear as critical [HePa03] and need to be thoroughly studied.

Step	Example 1	Example 2
1	X=-1.011·2^{-4} → X_E=-4; X_M=1.011 +Y=+1.000·2^{0} → Y_E=0; Y_M=1.000	X=-1.011·2^{-1} → X_E=-1; X_M=1.011 +Y=-1.100·2^{0} → Y_E=0; Y_M=1.100
2	d=-4-0=-4 → swapping X=+1.000·2^{0} → X_E=0; X_M=1.000 Y=-1.011·2^{-4} → Y_E=-4; Y_M=1.011 d=0-(-4)=+4>0; Z_E=X_E=0	d=-1-0=-1 → swapping X=-1.100·2^{0} → X_E=0; X_M=1.100 Y=-1.011·2^{-1} → Y_E=-1; Y_M=1.011 d=0-(-1)=+1>0; Z_E=X_E=0
3	→ $sign(X)\neq sign(Y)$ → Y_{MC2}=0.101	→ $sign(X)=sign(Y)$ →
4	Y_{MC2s}=1.11101(0 or 1) → 1 g r s	Y_{Ms}=0.1011 → $r=s=0$ g
5	X_M = 1.000 + Y_{MC2s} = 1.111 Z_M =10.111 c_{out}	X_M= 1.100 + Y_{Ms}= 0.101 Z_M=10.001 c_{out} z_0
6	z_{0n} Z_{Mn}=1.110; Z_E=0-1=-1	Z_{Mn}=1.000; Z_E=0+1=+1 z_{0n}
7	R=r=1 S=s=1	R=z_0=1 S=g or r or s =1 or 0 or 0 =1
8	$R(z_{0n}$ or $S)$ = 1(0 or 1) = 1 → from Fig.5.9 → Z_{Mn}= 1.110+ 1 1.111	$R(z_{0n}$ or $S)$ = 1(0 or 1) = 1 → from Fig.5.9 → Z_{Mn}= 1.000+ 1 1.001
9	$swap$=Yes, $sign(X)$= -, $sign(Y)$= + → from Fig.5.14 → $sign(Z)$= +	$sign(X)=sign(Y)$= - → $sign(Z)$= -
10	Z= +1.111·2^{-1}	Z= -1.001·2^{+1}

Fig. 5.15 Examples of applying the steps of the floating point addition/subtraction algorithm with rounding

Thus, first the additions, which appear in steps 3 (in the assessment of two's complement), 5 (in the computation of the preliminary sum/difference and in the possible two's complementation of the preliminary result significand) and 8 (in the addition of a binary unit for rounding) will be considered. Apparently we might be confronted with four activations of the adder, but, as noted at the presentation for step 5, when the conditions which require the two's complementing of Z_M are fulfilled, we have $d = 0$, consequently, addition/subtraction are executed exactly, and the rounding from step 8 is no longer necessary. Otherwise, the presented algorithm implies, for the worst case, three activations of the adder at most. Anyway, these activations require a considerable latency, when taking into account the fact that they

are executed on $(m+1)$ bits, and have, according to the IEEE 754 standard, $m = 23$ bits for single-precision, and $m = 52$ bits for double-precision respectively.

On the other hand, if reference is made to the shifts, they are, obviously, the more critical as they are performed on more binary positions. As mentioned before, apparently, such shifts on multiple bits are required for step 4 (in the alignment of the significands), and for step 6 (dedicated to the preliminary normalization of the result significand), while in the final normalization after the rounding from step 8 only a shift by one position to the right may appear. Let us lay out this analysis as a function of the absolute value of the difference d of the exponents, namely, let us consider, first of all, the situation when $|d| \leq 1$ and consequently, in step 4, we have a shift—to the right—of one position at the most. Under these circumstances, in step 6 there may result a shift—to the left—of two (refer to the example case f from Fig. 5.12) or more binary positions. But if $|d| > 1$, in step 4 we have a shift—to the right—of several bits, a situation when in step 6, as has already been observed, a shift—to the left—takes place, of one bit at most (refer also to the example cases a to e from Fig. 5.12). To sums up, shifts by two or more binary positions cannot appear in both steps 4 and 6, but only in one of them.

A Concerning the presented situation, a first possibility to speed up the addition/subtraction process consists of employing the arithmetic pipeline approach, according to which the performance improvement comes from the throughput increase. This means that, by overlapping parts of the addition/subtraction process for various such concatenated operations, rather than executing these parts in sequence, more additions/subtractions can be executed within the same time interval. The application of the pipeline overlapped execution method to the operations implies the requirement that they be divided into suboperations assigned to some so-called stages or segments, whose time length shall be as balanced as possible and shall not produce resource or data conflicts according to the processor's execution model for overlapped instructions [HePa03]. A general diagram of a pipeline structure for arithmetic is presented in Fig. 5.16 [Haye98].

Each S_i stage has been assigned a latch register R_i, usually a multiword one, for data storage, and a processing unit C_i consisting, usually, of a combinational circuit. The R_i registers retain the partially processed results as they move forward through the pipeline, but they are also used as buffers between neighbouring stages to prevent the interference of the information. The status changes at R_i registers take place synchronously, under the control of a common CLOCK signal. Each R_i register obtains an input data set that comes from the previous stage S_{i-1} (except R_1, to which data from the external source is supplied), data that represent the results of some computations made by C_{i-1}, in the previous CLOCK cycle. The operation takes place on these registers using C_i in the current CLOCK cycle, forwarding the output data of the new processing to the following S_{i+1} stage. Thus, in each CLOCK cycle, each stage transfers the previous results to the following stage and computes a new set of output data. In other words, in each stage a part of the computations is yielded, and the final result is obtained after a set of operands goes through all the pipeline stages. An increase in the throughput is obtained when more concatenated

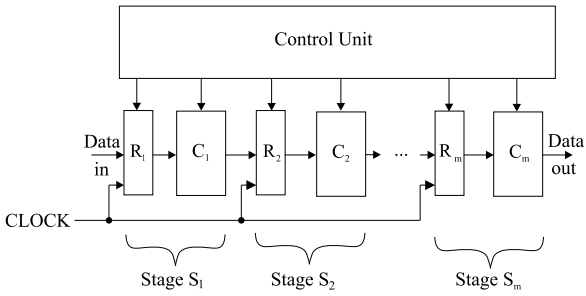


Fig. 5.16 Conceptual diagram of a pipelined structure intended for arithmetic

operations are executed, because a stage which, in the current CLOCK cycle, executes a specific processing task over a set of operands, becomes available to execute the same processing task over the next set of operands. Thus, at a certain moment, for the ideal case corresponding to no resource or data conflicts, up to m operations can be overlapped in the execution, m being considered the so-called pipeline depth.

Any operation that can be decomposed into a sequence of suboperations of approximately the same complexity becomes suitable, as has been seen in the case of multiplication with combinational arrays (Fig. 3.51), using a pipeline structure of the type presented in Fig. 5.16. If we refer to the addition procedure from the previous section, a hypothetical division of the suboperations into potential work stages would be that depicted in Fig. 5.17. Thus, in the allocation of stages, the worst case has been taken into account, i.e., for instance, to stage S_1 there have been assigned the compulsory operand unpacking and exponent comparison suboperations, but also, only in certain cases, if necessary, the operand swapping ones.

In stage S_2, the two's complementing of one of the operands is performed only when their signs differ, but significand shifting is necessary in all the cases, except when $d = 0$. In stage S_3, the adding/subtraction operation proper of the significand numbers is executed, while, in stage S_4, the result significand two's complementing is required only when this result is negative and the signs of the operands differ. By comparison, the sum/difference significand normalizing is, usually, a suboperation frequently resorted to. To the last stage, i.e. stage S_5, the potential rounding suboperations of the result significand, and the suboperations of its possible new normalization, as well as the compulsory packing of the result have been assigned.

Following the analysis of the way the addition/subtraction process suboperations have been assigned to the five stages, it can be noticed that in this case we have as balanced a length as possible for each stage when loaded with the most unfavorable (the greatest) number of suboperations. Furthermore, it is worth noticing that based on the available technology, a different grouping for the suboperations from that in Fig. 5.17 can also be considered, as shown, for instance, in [HePa03] implementation options for addition/subtraction units from some commercial chips.

In case of the so-called complete pipelining of the functional subunits (adders/subtracters, shifting devices, etc.), i.e. the adding/subtraction unit structure includes sufficient resources that the simultaneous execution of as many operations of a certain type as they are required can be fulfilled, then the ideal shifted overlapped concatenation can be executed as in Fig. 5.18. But if, for a certain functional subunit,

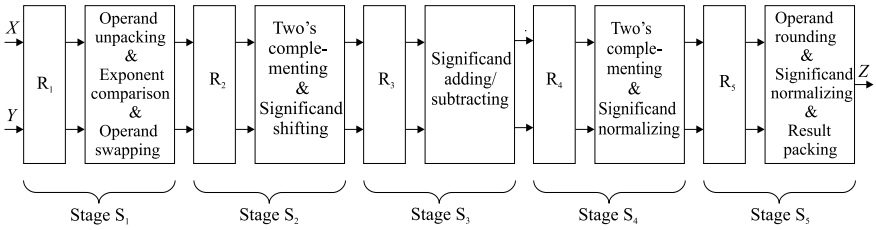


Fig. 5.17 Pipelined structure with a hypothetical partitioning on working stages of the floating point addition algorithm with rounding

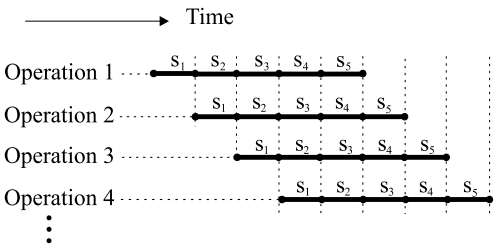


Fig. 5.18 Chaining with ideal shifted overlapping of the stages corresponding to an arithmetic pipeline structure

e.g. an adder, there are insufficient copies of this subunit, then the activity at the level of the corresponding subunit has to be serialized (it is said that there is a structural hazard), this causing inevitable degradation of the throughput, as compared to the ideal situation.

B Besides the solutions based on the arithmetic pipelining principle, the addition/subtraction process speedup can be accomplished by using the parallelism approach, particularly when reference is made to the simultaneous, parallel execution of two addition/subtraction processes [SeEv04, SeEv01]. Without losing generality, we will focus on addition, the adaptation to subtraction being easy to perform. Also, for precision purposes, we assume, regarding rounding, that the accepted mode is that "toward nearest even" (Fig. 5.9). As concerns the analysis of the acceleration based on parallelism, expressed as a function of the operands' signs and of the exponents' difference value, the following three cases have to be considered.

Before taking into consideration the analysis of these cases, an important observation should be made, that the following implementations are exclusively hardware solutions precisely in order to emphasize the dominance of the speed factor with respect to parallelization-based acceleration techniques. At the expense of implementation accuracy the proposed solutions consist of only combinational circuitry. The presentation concentrates on the problems associated with the parallelization of the procedure, and for the purpose of practical implementations a variety of combinational-sequential logic or hardware-software trade-offs can be resorted to. Mention should be made that the architectures to be presented in the following can be obtained by means of reconfiguration under the control of the command signals

appropriately generated by a control unit, signals which, for the sake of the circuits' clarity, were generally omitted.

B1 The case when the signs of the two operands, X and Y, are identical ($sign(X) = sign(Y)$).

The previously described algorithm shows that, for this situation, the two's complementing from steps 3 and 5 is avoided (refer to example 2, Fig. 5.15), consequently the adder that has to be employed in these suboperations is not activated, the activation being compulsory for the preliminary addition from step 5 and possible in the rounding step 8. The problem occurs in the addition from step 5 because, in this operation, carry-out may ($c_{out} = 1$) or may not ($c_{out} = 0$) be generated. Therefore, the position of the sum msb is not a priori known. For a more detailed consideration, let us consider the case, in Fig. 5.19, corresponding to which, for the sake of illustration, without loss of generality, the exponents' difference was assumed to be $d = 3$. However, mention should be made that, in general, d is a variable quantity and its range of possible values must be taken into consideration. In order to accomplish this, one possible solution can be represented by a combinational circuit for which the control is provided by the outputs of the decoder of the content of the register storing the value d. Reverting to the two alternatives, in Fig. 5.19a is presented the case when the c_{out} output of the parallel adder is 0, and in Fig. 5.19b respectively is presented the case when the c_{out} output of the parallel adder is 1. In both situations the rounding bits R and S can be generated in advance without needing to wait for the completion of the addition operation. In order to accomplish this, the structure takes advantage of the inputs c_{in} corresponding to the two adders, but in a different manner. More precisely, in Fig. 5.19a, the shifted significand Y_{Ms} (referring to all but the last bits y_2 to y_0 of the significand Y_M corresponding to the particular case) is added to the non-shifted significand X_M, so that when $c_{out} = 0$, the rounding bits can be determined in advance, namely $R = y_2$ and $S = y_1 \underline{or}\ y_0$. In this situation, by taking into consideration the general rounding function $R(z_{0n} \underline{or}\ S)$ (Fig. 5.9) and identifying the sum significand corresponding to this situation by Z_{M0}, the following is obtained:

$$Z_{M0} = X_M + Y_{Ms} + y_2(z_0 \underline{or}\ y_1 \underline{or}\ y_0)2^{-m} \tag{5.2}$$

where the multiplication by 2^{-m} suggests that the bit generated by rounding is added to the sum's lsb (to the bit z_0), because the significand's mantissa is subunitary.

On the other hand, when $c_{out} = 1$ (Fig. 5.19b), the context is to some extent more complex because in the rounding function, in this case, two bits of the sum take part, namely z_1 and z_0, and the sum requires to be shifted one position to the left. Taking into consideration these remarks the adder in Fig. 5.19b is divided into two parts, the one constituted by a HAC and intended for adding the least significant bits x_0 and—for this particular case—y_3, and the other part used for adding the X'_M and Y'_{Ms} segments representing the significand numbers X_M and Y_{Ms} without their lsb ranks. By taking into consideration the new particularities $R = z_0$ and $S = y_2 \underline{or}\ y_1 \underline{or}\ y_0$, and the rank (with the output z_1) to which the bit generated by rounding is applied,

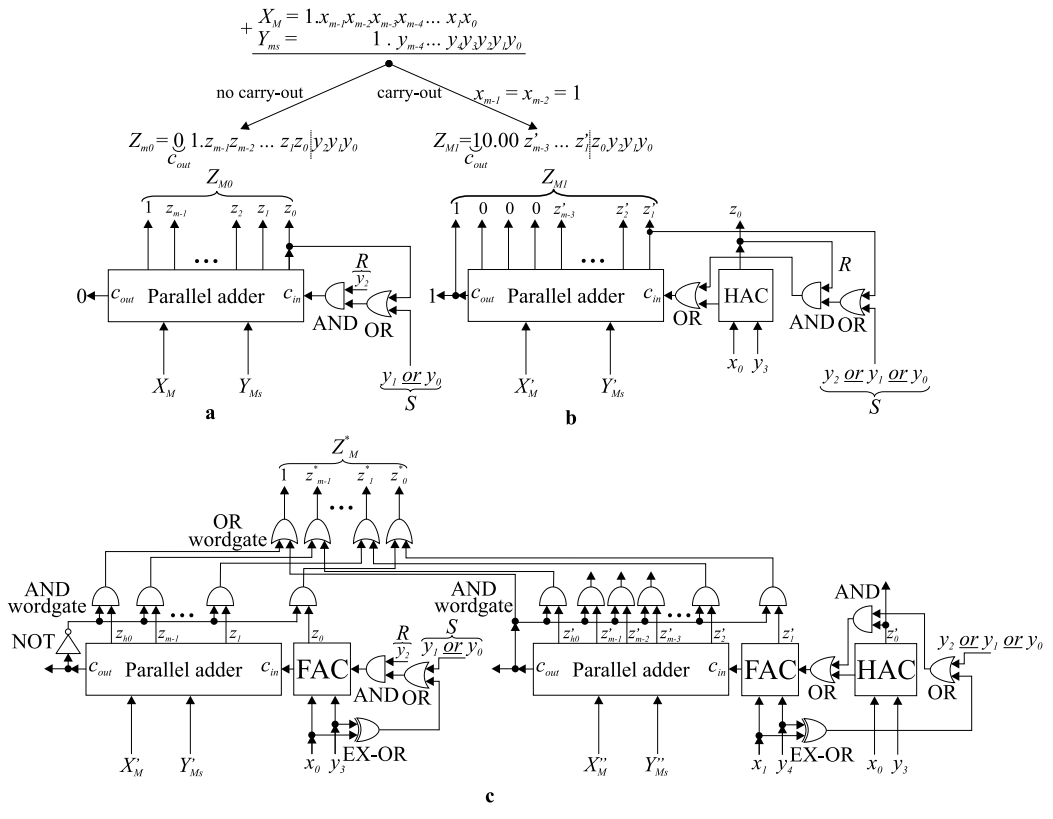


Fig. 5.19 Configuration of two adders with parallel operation for covering the case when the operands have the same sign

and labeling by Z_{M1} the sum significand corresponding to the case $c_{out} = 1$, the following is obtained:

$$Z_{M1} = X'_M + Y'_{Ms} + z_0(z_1 \underline{or}\ y_2 \underline{or}\ y_1 \underline{or}\ y_0)2^{-m+1} \tag{5.3}$$

The two sections of the adder in Fig. 5.19b are connected by an OR gate with the output connected to the c_{in} input of the parallel adder. The inputs for this gate are mutually exclusive since to one of the inputs is applied the Boolean sub-function $z_0(z_1 \underline{or}\ S)$, with $z_0 = x_0 \oplus y_3$, and to the other input is applied the Boolean sub-function $x_0 y_3$, so that when $x_0 y_3 = 1$ it follows that $z_0 = 0$ and when $z_0 = 1$ it results that $x_0 y_3 = 0$ respectively. In other words, the position of the respective OR gate allows for the correct covering of both the addition operation of the significands X_M and Y_{Ms}, as well as, in an anticipative manner, the possible rounding.

The circuits in Fig. 5.19a and Fig. 5.19b respectively were elaborated for the sole purpose of emphasizing the mechanisms covering the addition and anticipative rounding operations for the two distinct situations. The feedback connections intervening in both these circuits, which were expected to be combinational, can determine the apparition of oscillations intolerable for reliable functioning. As a consequence the transition to the cumulative version of the circuit in Fig. 5.19c was required, in which the lsb ranks of the adders in Fig. 5.19a and Fig. 5.19b were separated out. These design decisions were adopted in order to highlight the connection mechanism for the EXCLUSIVE-OR gates intended for doubling the respective gates of the FACs in order to avoid the feedback connections introduced into the circuits of Fig. 5.19a and Fig. 5.19b respectively. The two adders of Fig. 5.19c operate concurrently, and obtain two results, potentially different, of which only one is correct. The proper result is selected based on the value which results for c_{out}, namely, if this value is 0 the result of the left adder is chosen whereas if it is 1, the result of the right adder is chosen. Aside from some special cases, to be discussed below, the value of c_{out} coincides for the two adders. Thus, by marking the c_{out} value of the adder on the left side with c_{out0}, and with c_{out1} the c_{out} value of the adder on the right side, whenever $c_{out0} = c_{out1} = 0$, the control for selecting the result of the left-side adder is assured by satisfying the logical condition $(c_{out0} \odot c_{out1})\overline{c_{out0}} = 1$, where by $\odot$ we denote the coincidence function. By simplifying this logical condition we obtain: $\overline{c_{out0}} \cdot \overline{c_{out1}} = 1$. However this does not cover the, previously remarked, limit situations derived from the simplified case of adding operands $X = 1.010 \cdot 2^0$ $(= 5/4$ in decimal) and $Y = -1.010 \cdot 2^{-1}$ $(= -5/8$ in decimal). By making use of the described addition algorithm, the adder on the left of Fig. 5.19c provides the result significand consisting of only 1s, namely $Z_{M0} = 1.111$ $(= 15/8$ in decimal) and $c_{out0} = 0$, while the adder on the right provides the result significand consisting of a single 1 bit($c_{out1} = 1$) followed by 0s to the right, namely $Z_{M1} = 1.000 \cdot 2^1$ $(= 16/8$ in decimal). It can easily be seen that the deviation from the result's exact value is smaller for Z_{M0}, and this is the case for all cases similar to the one presented above since for all these cases the value of the rounding bit R is 0 for the adder on the right, and 1 for the adder on the left. These observations justify the decision of selecting the result significand Z_{M0} whenever one of the special

cases is encountered. In consequence the result is determined based on the logical condition $(c_{out0} \oplus c_{out1})\overline{c_{out0}} = \overline{c_{out0}} \cdot c_{out1} = 1$. By combining the effect of the two logical conditions covering the disclosed cases, it follows that the condition for delivering the left adder's result is $\overline{c_{out0}} = 1$, justifying the fact that the layer of AND gates allowing Z_{M0} (Fig. 5.19c) to pass is conditioned by variable $\overline{c_{out0}}$. On the other hand, when $c_{out0} = c_{out1} = 1$ the control for delivering the result Z_{M1} of the right-side adder (Fig. 5.19c) is assured by satisfying the logical condition $(c_{out0} \odot c_{out1})c_{out1} = c_{out0} \cdot c_{out1} = 1$. Since the case $c_{out0} = 1$ and $c_{out1} = 0$ can easily be shown to not occur, the condition $c_{out0} \cdot c_{out1} = 1$ can be further simplified, finally obtaining $c_{out1} = 1$. As a result the layer of AND gates, ensuring the delivery of Z_{M1}, has the variable c_{out1} supplied to it. In this manner, by ignoring the balancing of the connections' latencies, we obtain the united implementation of the two adders operating in parallel, for which the sum significand Z_M^* is delivered by the OR gates layer implementing the cumulative logical function:

$$Z_M^* = Z_{M0} \cdot \overline{c_{out0}} + Z_{M1} \cdot c_{out0} \tag{5.4}$$

As concerns the case $sign(X) = sign(Y)$, it can be concluded that by appealing to the simultaneous, parallel execution of the two additions, in terms of the computation's latency, the pessimistic interval required by the three adder activations, which results at first analysis, can be reduced to that corresponding to only one activation, representing a consistent acceleration of the operation execution.

B2 The case when the signs of the two operands, X and Y, differ ($sign(X) \neq sign(Y)$), but they have the same exponent ($X_E = Y_E$).

The previously described algorithm shows that, in this case, the two's complementing from step 3, as well as the preliminary addition from step 5 are compulsory, and this addition can be followed, under the condition that the sum's msb is 1 and there is no carry-out, by a new two's complementing, now of the resulting significand. Obviously, the sum results exactly, making the rounding from step 8 useless. However, this situation requires, for the worst case, three activations of the parallel adder, if we assume that this adder is used to obtain the two's complement by adding a 1 via the carry-in input to the value of the one's complement. This pessimistic situation can also be overcome by using two adders that operate simultaneously and execute the operations suggested in Fig. 5.20.

For simplicity, it is specified that the same notations X_M and Y_M have also been used for the signed significand numbers, and that the binary unit which is added to the lsb digits has been denoted by 1, without taking into account its weight. We also remark that on the left side (Fig. 5.20a) the $X_M + (-Y_M) = X_M + Y_{MC1} + 1$ operation is executed, and on the right side (Fig. 5.20b) the $(-X_M) + Y_M = X_{MC1} + Y_M + 1$ operation is executed, where X_{MC1} and Y_{MC1} represent the one's complement values obtained by significands X_M and Y_M passing through the EXCLUSIVE-OR wordgates, with the control signal c_1 set ($c_1 = 1$) by the control unit, when the condition $(d = 0 \& (sign(X) \oplus sign(Y) = 1))$ is fulfilled. By supplying $c_1 = 1$ to the c_{in} inputs of the two adders as well, the two's complements are yielded, so that we have the results of $X_M + (-Y_M)$, and $(-X_M) + Y_M$,

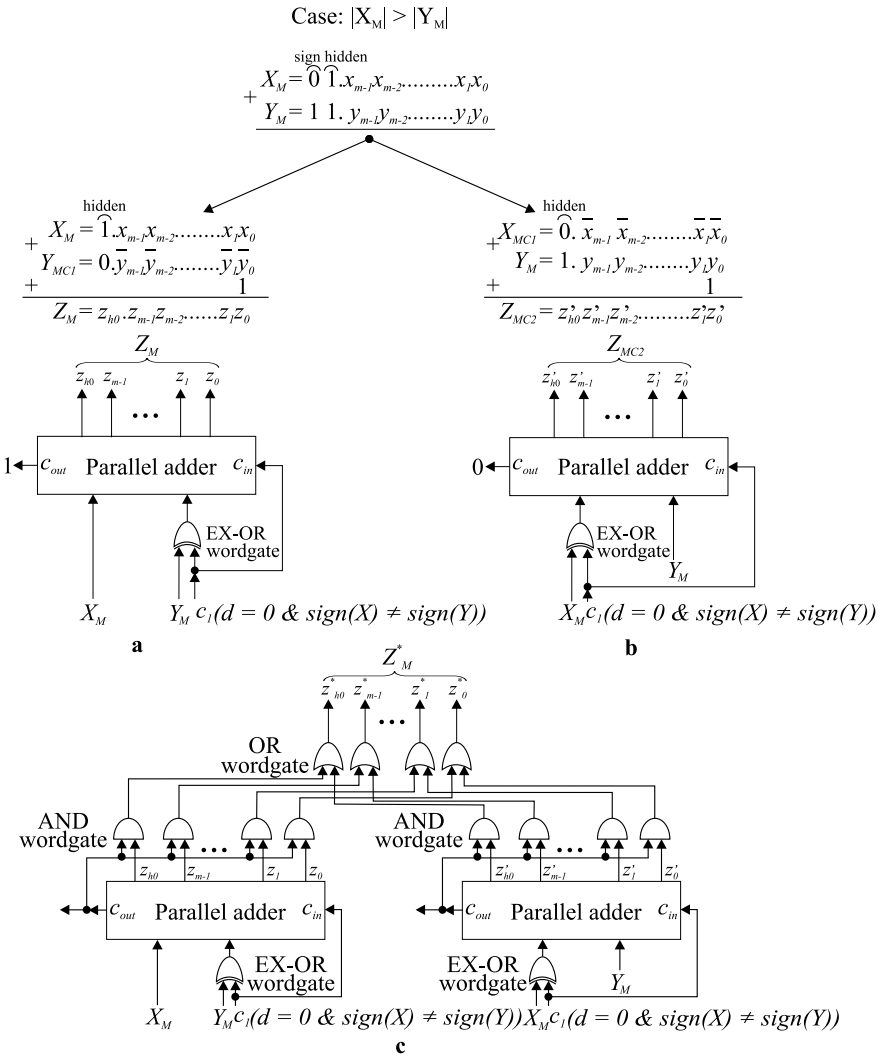


Fig. 5.20 Configuration of two adders with parallel operation for covering the case when the operands have different signs but have the same exponent value

additions which lead to the sums Z_M and Z_{MC2}, respectively, the latter representing the two's complement of the former. Thus, the two's complement from step 3 and the addition from step 5 have been compressed into only one operation, followed by the establishment of the result's sign. In addition to this, it is worth mentioning that one sum result, Z_M, is positive, namely, the one for which $c_{out} = 1$, and the other one, Z_{MC2}, is negative, namely, that for which $c_{out} = 0$. Z_{MC2} needs to be subjected to an additional two's complementing in order to obtain the desired sign-magnitude form of the result significand. Consequently, value Z_M is selected from

the adder satisfying $c_{out} = 1$. By taking into consideration the aspects mentioned, in Fig. 5.20c is presented the cumulative structure implying the parallel operation of the two adders, adders which also appear in Fig. 5.19c but which are now configured differently through the corresponding control signals provided by the control unit, signal, which, for the sake of circuits' clarity, were omitted. It is worth noting, also, the control of the AND gate layers associated with the adders by means of the c_{out} control signal derived from the adder itself. In this manner, toward the layer of OR gates, which delivers the final result significand Z_M^* at its outputs, can pass only that result significand for which $c_{out} = 1$.

The conclusion that can be drawn for this case is similar to that from B1, namely that the process of addition can be accelerated by using two adders (instead of one) which function simultaneously, there being required only the length of time corresponding to one activation of the adder instead of the worst case where the time length of three activations is needed.

B3 The case when the signs of the two operands, X and Y, differ ($sign(X) \neq sign(Y)$), the values of the exponents being also different ($X_E \neq Y_E$).

We suggest the splitting of the analysis corresponding to this situation as a function of the absolute value of the difference between the exponents values, namely $|d| = |X_E - Y_E|$.

B3a The subcase $sign(X) \neq sign(Y)$ and $|d| = 1$.

As far as this subcase is concerned, first of all, mention should be made that the two's complementing from step 3, as well as the preliminary addition from step 5 are compulsory. This is related to the rounding from step 8, which becomes unnecessary if the sum has two or more leading 0s (refer also to the example case *f*, Fig. 5.12); the bits involved in rounding become 0 due to the left-shift, thus the addition is executed exactly. There are situations when upward rounding might be necessary, namely when the leading bit 1 of the result coincides, as far as its position is concerned, with that of the operand significand not being shifted. Consequently, the result is normalized and needs no additional shifting. It is also the case when, continuing the same simplified analysis, the addition is performed, for example, on the operands $X = 1.110 \cdot 2^0 (= 7/4$ in decimal) and $Y = -1.001 \cdot 2^{-1}$ $(= 9/16$ in decimal), for which the non-rounded sum is 1.001. However since for this case $R = 1$, and the sum's lsb is also 1, the rounded result, obtained without shifting, is equal to $1.010 (=$ $5/4$ in decimal, as opposed to $19/16$ in decimal obtained by exact calculation).

Although the sticky bit S is 0, because R and z_0 (the lsb of the result) can be 1, according to Fig. 5.9, it is possible to have an upward rounding in step 8. For speedup purposes, let us cumulate, first of all, the operations from steps 3 and 5, computing, by activating the control signal c_2 $(|d| = 1 \& (sign(X) \neq sign(Y)))$, the difference between the significand numbers X_M and Y'_M (representing the significand Y_M less its lsb y_0), as presented in Fig. 5.21. Without loss of generality, it offers an example for the case when $sign(X) = 0$ and $sign(Y) = 1$. In the adder from Fig. 5.21a can be observed the situation in which the bit from the hidden position of the result significand Z_M has the logic value 1, the same as the homologous bit of the operand

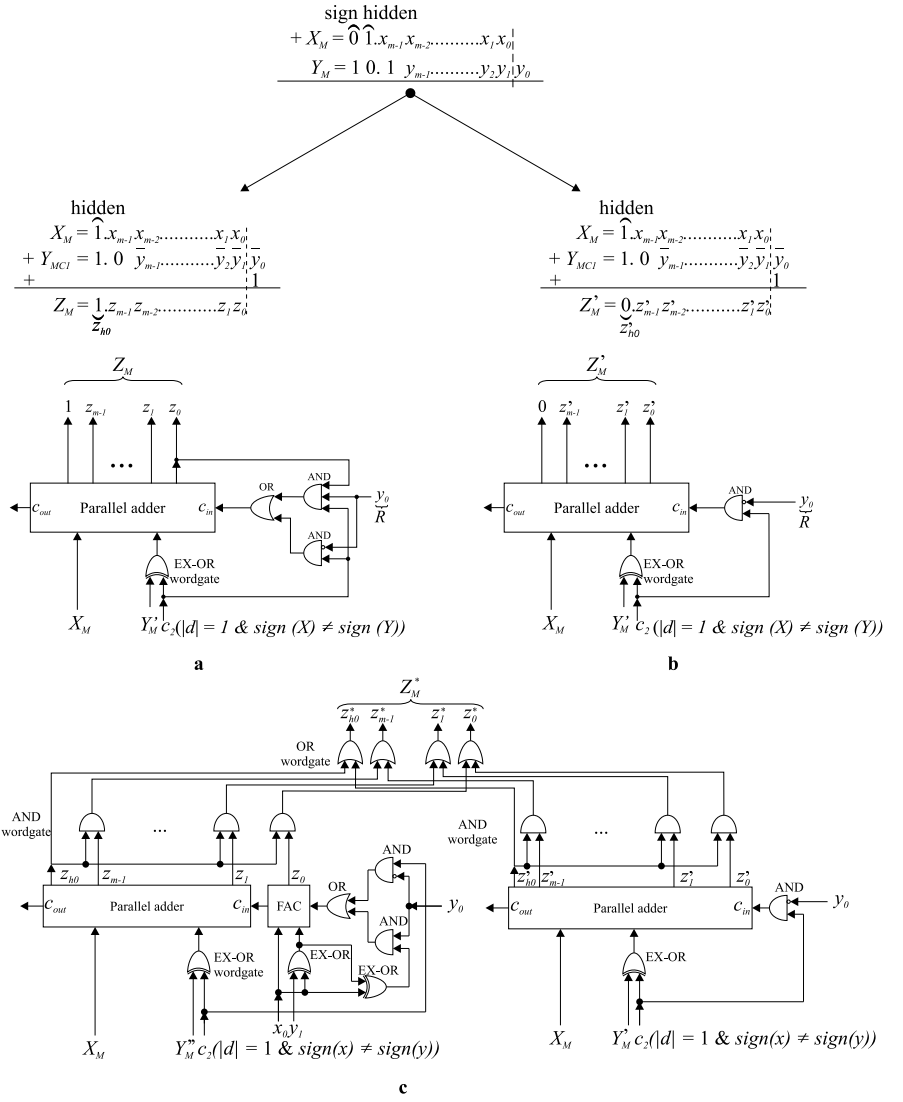


Fig. 5.21 Configuration of two adders with parallel operation for covering the case when the operands have different signs and the absolute value of their exponents difference is equal to 1

significand X_M, when, if we also have y_0 (alias R) $= z_0 = 1$, it results that upward rounding is necessary. This operation can be combined with the two operations from steps 3 and 5, all of them being executed during step 5, because Z_M normalization (step 6) is no longer necessary (there is a 1 in the hidden position z_{h0}) and neither is the adjustment of values R and S (step 7). Further remarks refer to the complementing operation from step 3, which is executed simultaneously with the preliminary

addition from step 5, consequently, the order of the execution of steps 3 and 4 from the algorithm described in the previous section is reversed. From the technical point of view, the combining of rounding with the other operations requires the logical circuit implemented on the adder input c_{in}, which executes the OR function between the AND operations between y_0 and z_0 (upward rounding) on the one hand, and $\overline{y}_0$ and c_2 (the addition of 1 to $\overline{y_0}$) on the other hand. The justification for using the OR gate is based on the fact that the terms $y_0 z_0$ and $\overline{y_0} c_2$ cannot have the logic value 1 simultaneously.

But, if the z_{h0} bit of sum's significand Z_M is 0, then the normalization step 6 cannot be omitted; furthermore, through the normalization R becomes 0 and the rounding doesn't have to be executed, the result being exact. Otherwise, no rounding is executed by the adder from Fig. 5.21b, and, after Z_M assessment and normalization, y_0 (alias R) becomes a bit of the significand Z_M.

For the result's delivery the cumulative structure of Fig. 5.21c was elaborated. For the case that $y_0 = 0$, the results supplied by the two adders coincide. However, if the results provided by the two adders differ, the correct result selection from the two results obtained in parallel is performed based on the value of their hidden bit z_{h0}. Accordingly, if $z_{h0} = 1$, the result delivered by the left-side adder is chosen (with Z_M being potentially upward rounded) for which the avoidance of oscillating behavior is realized by the supplementary EXCLUSIVE-OR gate implementing the function $x_0 \oplus \overline{y_3}$. The delivery of the result significand to the OR gates layer in order to become the final result significand, Z_M^*, is accomplished by conditioning the AND gate layer with the value of z_{h0}. On the other hand, provided that $z_{h0} = 0$, the result significand provided by the adder on the right is chosen (Z'_M is not rounded but needs to be normalized) for which the condition for penetration through the OR gates layer is realized by applying signal $\overline{z_{h0}}$ to the layer of AND gates associated with the adder. In this way, again, the three adder activations are reduced, for the described parallel solution, to one activation, with the corresponding decrease in computation time.

B3b The subcase $sign(X) \neq sign(Y)$ and $|d| > 1$.

As far as this subcase is concerned, again, the two's complementing from step 3 and the preliminary addition from step 5 are compulsory. Also, the conditions from Fig. 5.9 determines whether to execute the rounding from step 8 (refer also to the example cases *a* to *e*, Fig. 5.12). On the basis of the previous analyses, corresponding to this situation, the position of the leading bit 1 of the difference can be only one of two, namely, the hidden bit position of the non-shifted operand or the one immediately adjacent to the right. In order to accelerate the execution of the operation, we stipulate, in this case as well, the combining of the two's complementing from step 3 with the preliminary addition from step 5 through a similar solution to that from subcase B3a, this combined step being preceded by step 4 (of significand numbers aligning). First, let us analyze the effects of interchanging steps 3 and 4 on the values of the *g*, *r* and *s* rounding bits. Thus, regarding *s* it can easily be seen that, no modifications occur, *s* having the same value for both sign-magnitude and the two's complement representations. Things differ for *g* and *r*. In order to obtain their

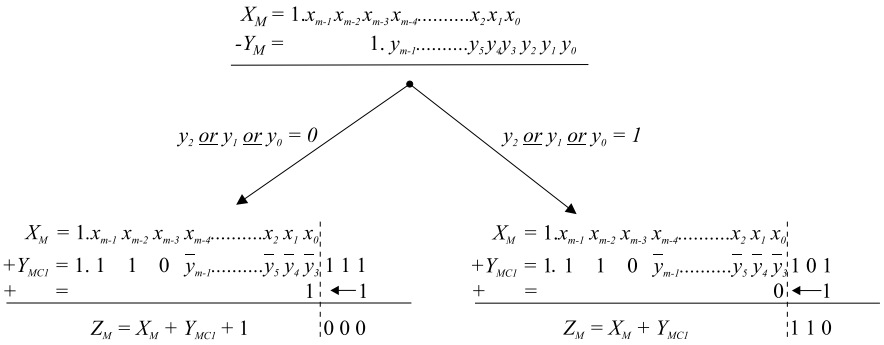


Fig. 5.22 Analysis of a subtraction example relevant for the case when the operands have different signs and the absolute value of their exponents difference is greater than 1

correct values, an additional investment consisting of the extension of the adder used in the Z_M computation by two bits to the right is required. First of all, some remarks need to be made in connection with this extension concerning the accomplishment of the example difference from Fig. 5.22. Mention should be made that, when all the binary digits of Y_M situated to the right of the $(m+1)$ binary digits of operand X_M are 0 (in Fig. 5.22, $y_2 = y_1 = y_0 = 0$), then the 1 bit, added to the bit $\overline{y_0} = 1$ of the one's complement (Y_{MC1}) corresponding to Y_M, propagates as carry, being added, ultimately, to the lsb pair out of the $(m+1)$ bits of the operands X_M and Y_{MC1}. Thus, the sum Z_M on $(m+1)$ bits, consists of $Z_M = X_M + Y_{MC1} + 1$. On the other hand, if one or more bits of Y_M, situated to the right of those $(m+1)$ bits of the operand X_M are 1 (in Fig. 5.22, we consider the case $y_1 = 1$ and $y_2 = y_0 = 0$), then the carry propagation, provoked by the addition to the bit $\overline{y_0} = 1$ of the one's complement (Y_{MC1}) corresponding to Y_M is blocked, so that to the lsb pair of the $(m+1)$ bits of the operands X_M and Y_{MC1}, a 0 is added, so that the sum, Z_M, on $(m+1)$ bits, becomes $Z_M = X_M + Y_{MC1}$.

Taking into account the previous observation, let us return to the adder extended with two binary positions to the right of the $(m+1)$ bits with the purpose of obtaining the correct values for the g and r rounding bits. For this, let us present the alternative situations from Fig. 5.23. We have supposed that Y_M, shifted for alignment purposes, is subtracted from X_M, for which we considered, without losing generality, the case when $g = y_3$, $r = y_2$ and $s = y_1 \ \underline{or}\ y_0$. Operand X_M is extended with two 0s to the right of the lsb position (x_0), becoming X_{Me}, and operand Y_M's one's complement has $\overline{g}$ and $\overline{r}$ bits, in the positions corresponding to the two 0s of X_{Me}, becoming Y_{MC1e}. The two alternative situations occur as a function of the value of the preliminary sticky bit, s, established correctly in step 4 of the presented addition algorithm. Thus, when $s = 0$, it can be observed that the addition of a 1 to $\overline{s}$ yields the propagation of a carry to the lsb of the adder, obtaining, in compliance with the addition from the left side of Fig. 5.22, the sum significand Z_{Me}, also extended by the two corresponding binary positions to the right, in the form $Z_{Me} = X_{Me} + Y_{MC1e} + 1$. Similarly, when $s = 1$, by adding a 1 to $\overline{s}$ no carry is generated to the adder's lsb, so that, according to the right side addition from Fig. 5.22,

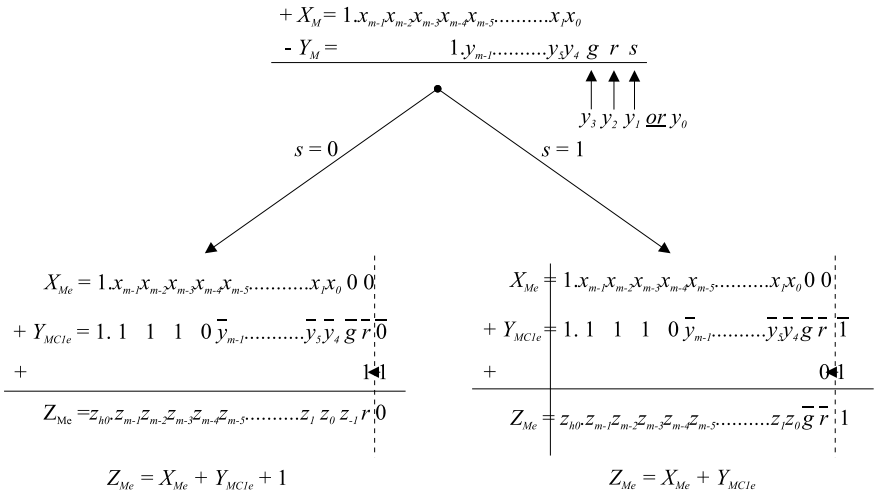


Fig. 5.23 Obtaining the correct values of the rounding bits g and r for the case when the operands have different signs and the absolute value of their exponents difference is greater than 1

Z_{Me} is obtained, this time, in the form $Z_{Me} = X_{Me} + Y_{MC1e}$. What is essential is the fact that, in both situations, the bits from Z_{Me}'s extended positions have correct values (z_{-1} and r when $s = 0$, $\overline{g}$ and $\overline{r}$ respectively, when $s = 1$), and thus, rounding can be executed in accordance with the presented algorithm's requirements.

With these amendments, let us try, for the sake of speeding up the addition process, to combine the three previously mentioned suboperations in one operation, by taking into account the two presented situations, namely when the z_{h0} (hidden) bit of Z_{Me} is 0 or 1 respectively. Thus, for $z_{h0} = 0$, in Fig. 5.24a and in Fig. 5.24b, the separate alternatives corresponding to $s = 0$ and $s = 1$ are presented, by making an analogy with Fig. 5.23. In both cases, the parallel adders have been extended to the right with two full adder cells (FAC) interconnected in ripple carry adder (RCA) manner. Since we assumed that z_{h0} is equal to 0, the final sum significand Z_M needs to be normalized through left shifting by one binary position. This suboperation can be avoided if we take the shifted bits of Z_M, and thus the hidden bit ($z_{h0} = 1$) moves one binary position to the right, thus the lsb of Z_M becomes z_{-1}. Before analyzing the two configurations, we select, again, the "toward nearest even" rounding mode (Fig. 5.9), with $R = z_{-2}$ and $S = s$, by taking into account the preliminary values of the g, r and s bits determined in the alignment step (step 4), which now precedes the significand Y_M complementing step (step 3). Under these conditions, the addition of a binary unit, for rounding purposes, to the least significant position of Z_M (z_{-1}), has to be executed only when $R(z_{-1}\ \underline{or}\ S) = 1$, i.e. when $z_{-2}(z_{-1}\ \underline{or}\ s) = 1$. With these stipulations, let us configure the parallel adders so that the complementing from step 3 and the addition from step 5, as well as the possible rounding from step 8, are covered by one activation. For Fig. 5.24a and Fig. 5.24b, the set of EX-OR gates will be noticed, which, in accordance with Fig. 5.23, enable Y_{MC1e} to be ob-

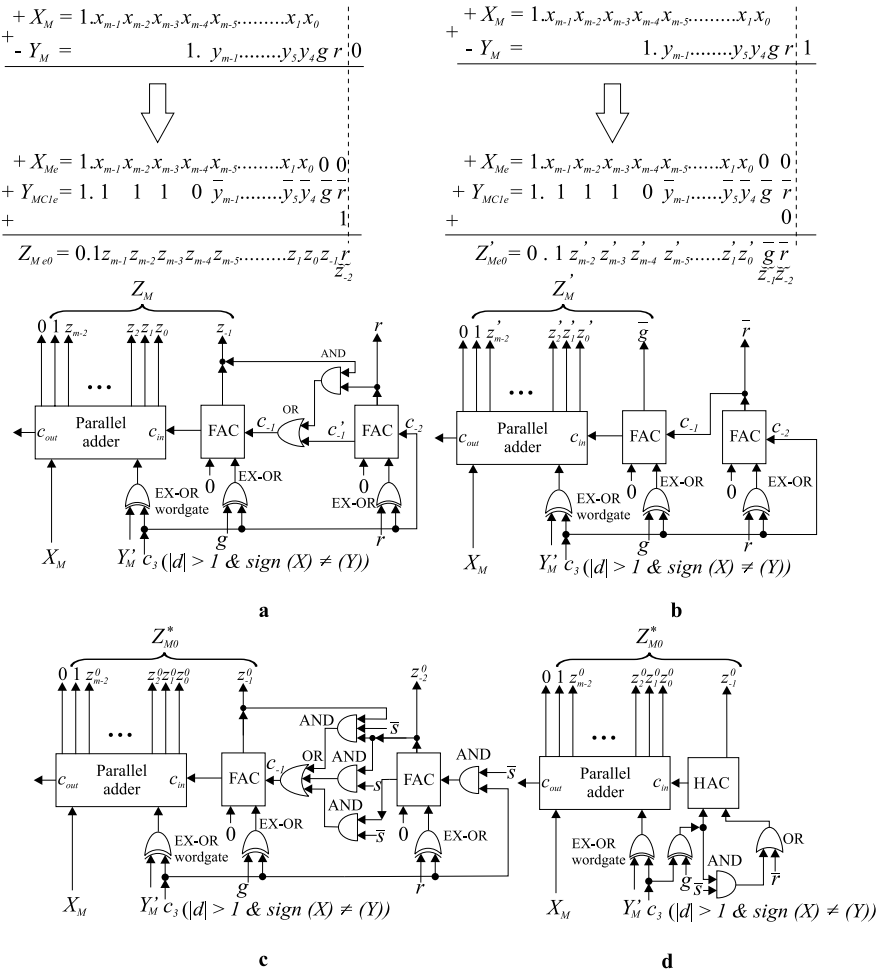


Fig. 5.24 Configuration of two adders with parallel operation for covering the case when the operands have different signs, the absolute value of their exponents difference is greater than 1 and the hidden bit is 0

tained (by Y'_M we have denoted only those bits of Y_M which are aligned with the bits of X_M and for which there has been provided the EX-OR wordgate) and which are all controlled by the c_3 signal that comes, when $|d| > 1$ and $sign(X) \neq sign(Y)$, from the control unit. The extended result significands were denoted by Z_{Me0} (for $z_{h0} = 0$ and $s = 0$) and Z'_{Me0} (for $z_{h0} = 0$ and $s = 1$) respectively, while the result significands were denoted by Z_{M0} and Z'_{M0} respectively.

The same c_3 signal is also applied, when $s = 0$ (Fig. 5.24a), to the carry-in input(c_{-2}) of the rightmost FAC, so that its sum and carry-out outputs become equal to $z_{-2} = r$, and $c'_{-1} = \overline{r}$, respectively. Since, after the signal stabilization,

it is impossible to have 1 at both inputs of the OR gate (because the AND gate is conditioned by r, and $c'_{-1} = \overline{r}$), we hereby justify the choice for logic synthesis of this gate type. Looking for a solution to eliminate the respective OR gate from the critical path, we can observe the fact that the AND gate output can be applied to that FAC input with z_{-1} sum output, where actually there is 0 (which is the reason why the two least significant FACs of the two adders are not substituted by HACs). Otherwise, the Boolean equation $c_{-1} = (z_{-1}r \ \underline{or} \ \overline{r})\overline{s}$ is applied to the carry-in input of the adder cell which generates the sum bit z_{-1}, when $s = 0$. On the other hand, the alternative right side circuit (Fig. 5.24b) corresponding to the case when $s = 1$, presents certain distinctive elements, namely that $c_{-2} = 0$ and, consequently, $z_{-2} = \overline{r}$ and $c'_{-1} = 0$, and the Boolean rounding function $R(z_{-1} \ \underline{or} \ S)$ becomes, since $S = s = 1$, equal to $R = r$. Otherwise, the Boolean equation $c_{-1} = \overline{r}s$ is applied to the carry-in input of the cell which generates the sum bit z_{-1}, when $s = 1$. Since the Boolean subfunctions with input c_{-1}, as it can easily be observed, can only take the logic value 1 one at a time, the synthesis that reunites the alternatives from Fig. 5.24a and Fig. 5.24b can use the OR gate from Fig. 5.24a provided with an additional input for the subfunction from Fig. 5.24b. Thus, the circuit represented in Fig. 5.24c will be obtained, while the conditioning with $\overline{s}$, and s, additionally occurs as the case requires.

Taking into account the necessity to avoid in the above-mentioned manner the oscillations of the circuit, the Boolean function applied to the carry-in input, labeled c_{-1}, belonging to the FAC with the z^0_{-1} output, is processed as follows:

$$c_{-1} = \big((g \oplus c_3)r \ \underline{or} \ \overline{r}\big)\overline{s} \ \underline{or} \ \overline{r}s = (g \oplus c_3)r\,\overline{s} \ \underline{or} \ \overline{r} = (g \oplus c_3)\overline{s} \ \underline{or} \ \overline{r} \tag{5.5}$$

Obviously, when $c_3 = 1$, c_{-1}, given by (5.5), becomes $c_{-1} = \overline{g}\,\overline{s} \ \underline{or} \ \overline{r}$. Using this last form, the implementation from Fig. 5.24d is obtained.

If we now take into account the alternative $z_{h0} = 1$, the essential distinctive aspect consists of the fact that the normalizing operation from step 6 is no longer necessary. Of course, the final sum significand Z_M has $z_{h0} = 1$ as msb and z_0 as lsb, requiring, relative to the circuits from Fig. 5.24a and Fig. 5.24b respectively, the modifications from Fig. 5.25a (corresponding to $s = 0$), and from Fig. 5.24b (corresponding to $s = 1$), depending on the value of the preliminary sticky bit s (determined, in advance, in step 4). Before analyzing the two configurations, we recall that we resort to the "toward nearest even" rounding mode (Fig. 5.9), with $R = z_{-1}$ and $S = z_{-2} \ \underline{or} \ s$. It is assumed that the preliminary values of the g, r and s bits have been determined in the alignment step (step 4) which now precedes the complementation step of the significand Y_M (step 3). Under these conditions, the addition of a binary unit, for rounding purposes, to the least significant position of $Z_M(z_0)$ has to be executed only when $R(z_0 \ \underline{or} \ S) = 1$, i.e. when $z_{-1}(z_0 \ \underline{or} \ z_{-2} \ \underline{or} \ s) = 1$. As before, we shall configure the parallel adders so that they cover, with only one time activation, both the complementing from step 3, and the addition from step 5, as well as the possible rounding from step 8. Similar to the circuits from Fig. 5.24, those from Fig. 5.25 also have the set of EX-OR gates with the same purpose, being controlled by the same c_3 control signal. In the situation from Fig. 5.25a, which corresponds to

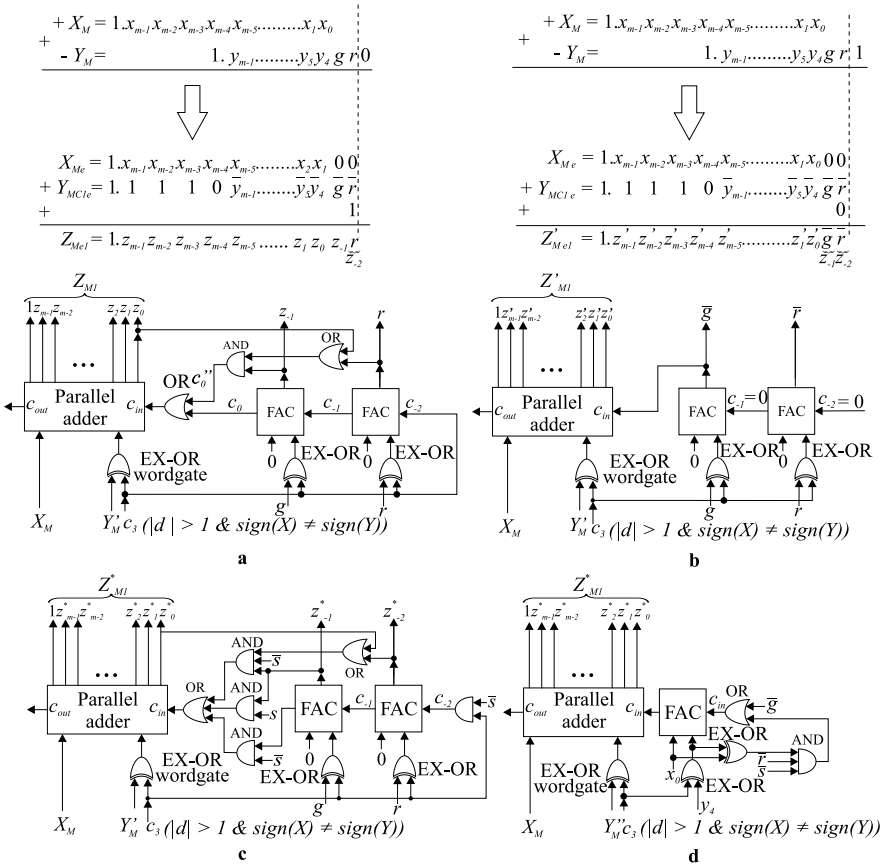


Fig. 5.25 Configuration of two adders with parallel operation for covering the case when the operands have different signs, the absolute value of their exponents difference is greater than 1 and the hidden bit is 1

the case when $s = 0$, c_3 is also applied, according to Fig. 5.23, to the carry-in input of the full adder cell (c_{-2}) in the rightmost position, so that its sum and carry-out outputs become equal to $z_{-2} = r$, and $c_{-1} = \overline{r}$ respectively, and, through the carry propagation to the following adder cell, consequently, $z_{-1} = \overline{g} \oplus \overline{r} = g \oplus r$ and $c'_0 = \overline{g}\,\overline{r}$ are obtained.

This last logic subfunction (c'_0) is passed to the parallel adder input c_{in}, and it ensures the simultaneous execution of the complementation of significand Y_M, and the preliminary addition, required by the algorithm in steps 3 and 5. However, it is mandatory to overlap the rounding operation of step 8, for which is generated, when $s = 0$, the following Boolean function:

$$c''_0 = z_{-1}(z_0 \underline{\,or\,} z_{-2} \underline{\,or\,} 0) = (g \oplus r)(z_0 \underline{\,or\,} r) \tag{5.6}$$

In Fig. 5.25a, the implementation of function (5.5) was realized by two gates, OR and AND. Since $(g \oplus r)r = \overline{g}r$, (5.6) can be rewritten in the following form:

$$c_0'' = (g \oplus r)z_0 \ \underline{or}\ \overline{g}r \tag{5.7}$$

It can easily be observed that the signals $c_0' = \overline{g}\,\overline{r}$ and c_0'', given by (5.7), cannot have the value 1 simultaneously, which justifies the selection out of several possible solutions of the one connecting to the c_{in} input of the parallel adder an OR gate with the inputs c_0' and c_0''.

On the other hand, for the alternative structure of Fig. 5.25b, according to Fig. 5.23, when $s = 1$, the complementation of Y_M, executed concurrently with the preliminary sum, prevents the control signal $c_3 = 1$ from being connected to the carry-in input of the rightmost FAC, in other words $c_{-2} = 0$. This in turn, determines $z_{-2} = \overline{r}$ and $c_{-1} = 0$, as well as the cascaded propagation, $z_{-1} = \overline{g}$ and $c_0' = 0$. By taking into consideration the rounding function $c_0'' = z_{-1}\ (z_0\ \underline{\text{or}}\ z_{-2}\ \underline{\text{or}}\ s)$, since $s = 1$ it follows that $c_0'' = z_{-1} = \overline{g}$, which is supplied to the c_{in} input of the parallel adder. By integrating the circuits in Fig. 5.25a and Fig. 5.25b, and by using the conditionality with $\overline{s}$, respectively with s, depending on the particular case, the cumulative structure of Fig. 5.25c is obtained. For this circuit, the OR gate connected to the c_{in} input of the adder has now, in addition to the two inputs from Fig. 5.25a, a third input, originating in Fig. 5.25b, but a logic value of 1 can appear, at a given moment, only on one of these inputs. The Boolean function implemented by the above-mentioned OR gate is therefore:

$$c_{in} = \big((g \oplus r)z_0^1 \ \underline{or}\ \overline{g}r\big)\overline{s} \ \underline{or}\ \overline{g}\,\overline{r}\,\overline{s} \ \underline{or}\ \overline{g}\,s \tag{5.8}$$

By executing on (5.8) some simple Boolean transformations, the following is obtained:

$$c_{in} = z_0^1\,\overline{r}\,\overline{s} \ \underline{or}\ \overline{g} \tag{5.9}$$

Provided that in (5.9) z_0^1 is substituted by $x_0 \oplus \overline{y_4}$ in order to eliminate the potential oscillating behavior which could be triggered by the feedback link of the z_0 connection, the following expression of interest for implementation is obtained:

$$c_{in} = (x_0 \oplus \overline{y_4})\overline{r}\,\overline{s} \ \underline{or}\ \overline{g} \tag{5.10}$$

By using the notation Y_M'' for the significand Y_M' excluding its least significant rank and by taking into consideration (5.10), the circuit of Fig. 5.25c is transformed into the structure of Fig. 5.25d.

In conclusion, for the case that $|d| > 1$ and $sign(X) \neq sign(Y)$, and the configuration being established under the control of the signal c_3, of the two results Z_{M0}^*, supplied by the circuit in Fig. 5.24d, and Z_{M1}^*, delivered by the circuit in Fig. 5.25d, it is selected as the correct result that one for which the value obtained for the bit z_{h0} corresponds. As a consequence the cumulative structure in Fig. 5.26 results, in which the resulting significand Z_M^* is equal to Z_{M0}^* when $z_{h0} = 0$, and is equal to Z_{M1}^* when $z_{h0} = 1$. Moreover, an analysis is required in order to cover also extreme

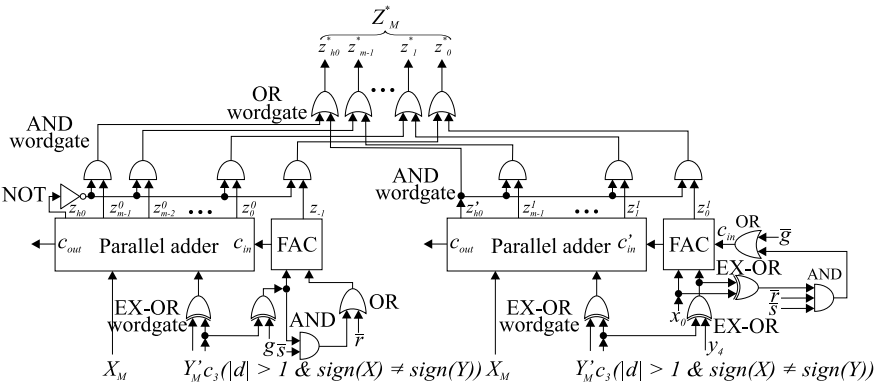


Fig. 5.26 Cumulative configuration of two adders with parallel operation for covering the case when the operands have different signs and the absolute value of their exponents difference is greater than 1

situations similar to those, exemplified in a simplified manner, such as the addition of the operands $X = 1.00101 \cdot 2^0$ ($= 37/32$ in decimal) and $Y = -1.01011 \cdot 2^{-3}$ ($= 43/256$ in decimal). The exact result of the operation is 0.11111101 ($= 253/256$ in decimal) whereas that obtained by applying the addition algorithm, adapted to the dimension of the example, is $1.11111 \cdot 2^{-1}$ ($= 63/64$ in decimal $= 252/256$). This time, the value propagated through the layer of AND gates attached to the adder on the left (Fig. 5.26) is 1.11111 with the corresponding "implicit" normalization, representing the correct result, while the value penetrating the AND gates layer attached to the adder on the right (Fig. 5.26) is 1.00000 ($= 1$ in decimal $= 256/256$). In the typical case only one of the AND gates layers is passed through by the sum result significand, either the one on the left (when $z_{h0} = 0$) or the one on the right (when $z_{h1} = 1$). For the particular extreme situations exemplified above, both AND gates layers are crossed by the sum result significand, but the final result Z^*_M is not affected, when taking into account the fact that it is obtained at the outputs of the OR gates layer. Furthermore, it can be noticed that the deviation from the exact result, as well as the deviation from the result obtained by applying the algorithm, is smaller for the sum significand obtained by the adder on the left, and this fact can be demonstrated without difficulty. In order to generalize the passing through the AND gates layer of a single result, and also in order to increase the structure's reliability, the AND gates attached to the adder on the right will have one of their inputs connected to the output of an additional AND gate having z_{h0} and z'_{h0} as its inputs, thus implementing the Boolean function $\overline{(z_{h0} \oplus z'_{h0})} z'_{h0} = z_{h0} \cdot z'_{h0}$.

Consequently, in this subcase as well, the three concatenated activations of an adder, are reduced, in terms of time, to only one activation of two adders that function in parallel, the same as in all the other analyzed situations. In an economic design version, the reconfiguration of the adders' circuits can be applied as a function of the operands' signs, and of the value of the exponents' difference. Thus, there

results a technical solution that may become a keen competitor for the option based on the pipeline approach.

5.3 Floating Point Multiplication and Division

If in fixed point arithmetic, multiplication and division represent operations obviously more complicated than addition and subtraction, in floating point the situation is reversed, therefore these two operations will be treated together. As has been mentioned from the beginning, relation (5.1) shows that the multiplication/division of two floating point operands, X and Y, consists of two fixed point operations, i.e. the multiplication/division of the significands, and the addition/subtraction of the exponents. A generic block diagram for both operations is given in Fig. 5.27 [Parh00], but it does not include the part of the circuits involved in the operation of special values (0, $\pm\infty$, etc.). Regarding the unpacking of the operands (X and Y) (unpack—Fig. 5.27), as well as the packing of the result (product P/quotient Q) (pack—Fig. 5.27), those methods used for addition/subtraction are still valid. Concerning the sign of result S_Z, it is obtained, in a simple way, by operating EXCLUSIVE-OR (EX-OR—Fig. 5.27) with the signs S_X and S_Y as operands, i.e. $S_Z = S_X \oplus S_Y$.

The problem of the operation on exponents consists in the fact that, in compliance with the IEEE 754 standard, their values X_E and Y_E are biased, representing numbers expressed in excess of a value of the bias which depends on the representation format's precision (with the value 127 at single-precision, and with the value 1023 at double-precision). In multiplication, when the two biased exponents, X_E and Y_E, have to be added in order to obtain the preliminary value of the result exponent, Z_E, it is necessary to subtract the bias from the sum $(X_E + Y_E)$ (to obtain a value which has the bias added only once). On the other hand, in division, when the two biased exponents are subtracted, the non-biased difference Z_E is obtained and, obviously, the value of the bias has to be added to the difference.

Thus, supposing that we adopt the IEEE 754 single-precision standard characterized by bias = 127, we shall consider the multiplication and the division of the non-standardized operands, $X_n = (-1)2^{-7}$ and $Y_n = (+1)2^5$. Referring to the biased, standardized values of the exponents, we have $X_E = -7 + 127 = +120$, and $Y_E = +5 + 127 = +132$, which leads to the operations from Fig. 5.28, where for the product and quotient results P and Q, indexes have been used, and the nonstandardized operands and nonstandardized results have been denoted by n, and the packed operands and the packed results have been denoted by p. We can observe the subtraction of the bias in the computation of the product exponent Z_{EP} (executed by the addition of the two's complement form of the bias), as well as the addition of the bias in the computation of the quotient exponent Z_{EQ} (executed by the addition of the sign-magnitude form of the bias).

The described operations are executed in the block Add/Substract exponents (Fig. 5.27), in connection with which we also mention that, regarding the implementation of the operations pointed out in Fig. 5.28, some simplifications to the

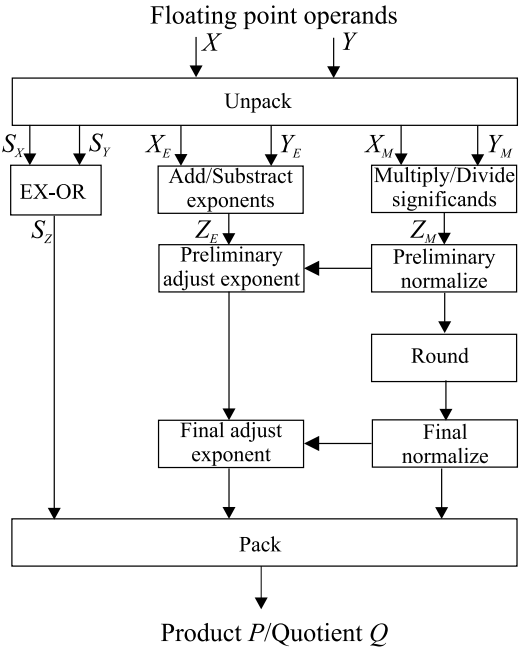


Fig. 5.27 Block diagram of a floating point multiplier/divider

circuitry are employed [ErLa04, Parh00]. Thus, the subtraction of the bias 127, in multiplication, is equivalent to the addition of a 1, by means of the c_{in} input on the exponents' adder and the subtraction from the sum of 128, which is equivalent to the switching of the msb of the product exponent Z_{EP}; in other words the operation $Z_{EP} = X_E + Y_E + 1 - 128$ is executed. On the other hand, the addition of the bias 127, in division, implies the accomplishment of the operation $Z_{EQ} = X_E - Y_E + 127 = X_E + Y_{EC2} + 127 = X_E + (Y_{EC1} + 1) + (128 - 1) = X_E + Y_{EC1} + 128$ where by Y_{EC1} is denoted the one's complement, and by Y_{EC2} the two's complement, of the exponent operand Y_E, between these two existing the known relationship $Y_{EC2} = Y_{EC1} + 1$. Otherwise, Z_{EQ} is obtained by adding Y_{EC1} (Y_{EC} is applied to an EX-OR wordgate) and by adding the value 128 through the switching of the msb of the sum $(X_E + Y_{EC1})$.

Regarding the significand numbers, X_M and Y_M, they are multiplied/divided in the most complex and the slowest part (Multiply/Divide significands—Fig. 5.27) of the entire multiplication/division device. Since $X_M \in [+1, +2)$ and $Y_M \in [+1, +2)$, the product of the two significand numbers without sign belongs to the $[+1, +4)$ range, just as the ratio of the two significand numbers without sign belongs to the $(+(1/2), (+2))$ range. Consequently, in order to obtain the result significand, normalization is required by one position right shifting of the product with the corresponding incrementing of the value of the preliminary exponent, and by one position left shifting of the quotient with the corresponding decrementing of the value of the preliminary exponent (Preliminary normalize & Preliminary adjust exponent—Fig. 5.27).

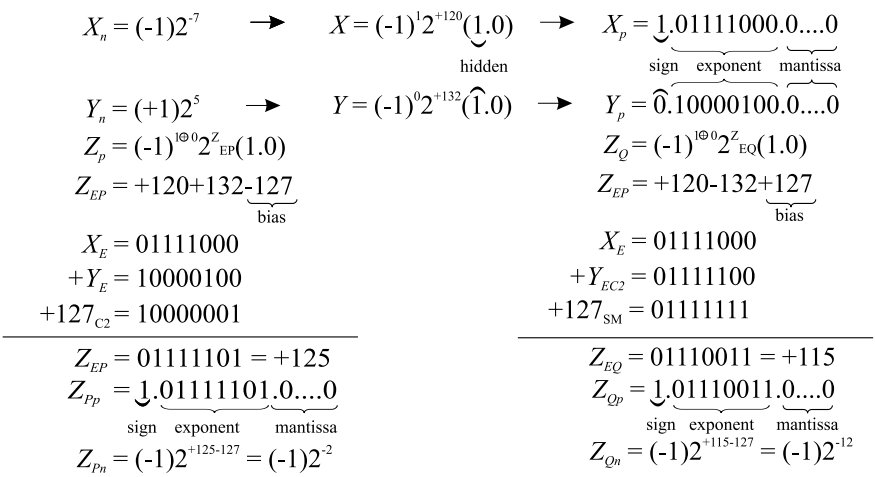


Fig. 5.28 Implementation examples for exponents addition/subtraction

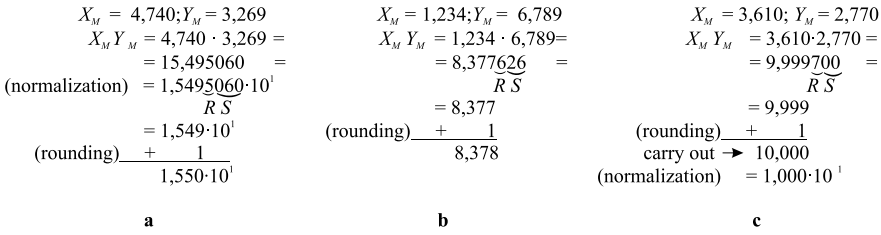


Fig. 5.29 Analysis of rounding situations for the floating point multiplication based on examples adopted from the decimal number system

Following the suboperations' flow, the described preliminary normalization is succeeded by rounding, which has specific aspects for the two operations. Thus, referring first to multiplication, we shall resort to some examples which will be adopted from the more familiar decimal number system. If we suppose, by analogy with the IEEE 754 standard, that we have a "hidden" decimal digit and three decimal digits that form the mantissa, i.e. $m = 3$, in Fig. 5.29 three potential situations for multiplication are presented. In case a, the product $X_M Y_M$ results in $((m + 1) + (m + 1)) = (2m + 2)$ decimal digits and it has first to be brought into the "normalized" form with one digit in the "hidden" position. The rounding suboperation follows, for which purpose, to the right of the $(m + 1)$ digits (value 1.549) the digits involved in the rounding are identified, R (round digit) and S (sticky digit). By analogy with what has been presented in binary, in Fig. 5.8a, these digits have the values $R = 5$ and $S \neq 0$ ($S = 0$, only when all the digits to the right of R are 0). Since $S \neq 0$, the rounding is made upwards by the addition of one unit to the least significant digit of the normalized "significand", according to the rounding model

of the "toward nearest even" mode from Fig. 5.9 ($S \neq 0$ is equivalent to $S = 1$ in binary; if S had been 0, we would have chosen, between 1.549 and 1.550, the even one, 1.550, i.e. the same result as those from Fig. 5.29, because the digit 9, the least significant as a result of the "normalization" is odd, which is equivalent to $z_{0n} = 1$ (Fig. 5.9) whereas if this digit were even, then z_{0n} would be 0). As previously described, the "significand" result has been obtained on the same number of digits as the operand "significands". On the other hand, case b is similar to the one already described, with the observation that the $X_M Y_M$ product results, according to the second possible alternative, in only $(2m + 1)$ digits, a situation in which the normalization, by right shifting as in case a, is no longer necessary. Consequently the rounding suboperation will follow (in a similar way to the one presented before), by identifying the R and S digits. Since, in this case, $R = 6$, having a value superior to 5 (which, being at the midpoint of the interval, requires the additional investigation of S), thus, the investigation of S becomes superfluous, and the upward rounding is compulsory. Mention should also be made that the position of the digit to which the rounding unit is added, relative to the point of the initial product $X_M Y_M$, differs between cases a and b. Finally, case c develops in a similar way to case b, including the rounding suboperation, in which, after adding the unit to the least significant decimal digit, carry out results for the most significant digit. This requires an additional normalization suboperation, this time through right shifting (Final normalization & Final adjust exponents—Fig. 5.27). Obviously, in both normalization cases, pre- and post-rounding, it is necessary to properly adjust the value of the result exponent.

Using as a model the examples given in decimal and extending our analysis to binary numbers, we specify that, in this case as well, the product, which can result in $(2m + 2)$ or $(2m + 1)$ bits, has to be rounded to $(m + 1)$ bits to correctly pack the result, in compliance with the IEEE 754 standard. As far as the number of the product bits is concerned, mention should be made that the product can be obtained, on the one hand, in the extended form, on $(2m + 2)$ bits, and subsequently, in the end, the rounding operation is applied. But, on the other hand, the additional bits can be gradually eliminated, as they are produced [Parh00]. Thus, we can imagine a multiplication device, e.g. a sequential one, of the type presented in Chap. 3, which results in a product on $(2m + 2)$ bits, with the more significant part in a register A (on $(m + 1)$ bits), and the less significant part in a register Q (on $(m + 1)$ bits), as product P is presented Fig. 5.30. Since product $X_M Y_M$ lies in the range of values $[(+1), (+4))$, the most significant two bits of the product, denoted by z_{h1} and z_{h0}, may have values differing from 0, a situation in which the lsb of the product (z_0) exceeds the capacity of the first result register (A), and it has to be retained in the msb position of the second result register (Q). Thus, the position of the guard bit g is occupied by z_0, and to the right of z_0, we have, in the usual manner, the round bit r and the rest of the $(m - 1)$ less significant bits of the product (from register Q), which have the role of the sticky bits s. On the other hand, because it is known that a sequential multiplication device generates the product bits in order, starting with the lsb and ending with msb, there is enough time to OR operate the less significant $(m - 1)$ bits, thus to finally obtain the compressed product form P_c from Fig. 5.30, which has only $(m + 1)$ bits for $z_{h1} z_{h0} z_{m-1} \ldots z_1$ to which are added the 3 bits for

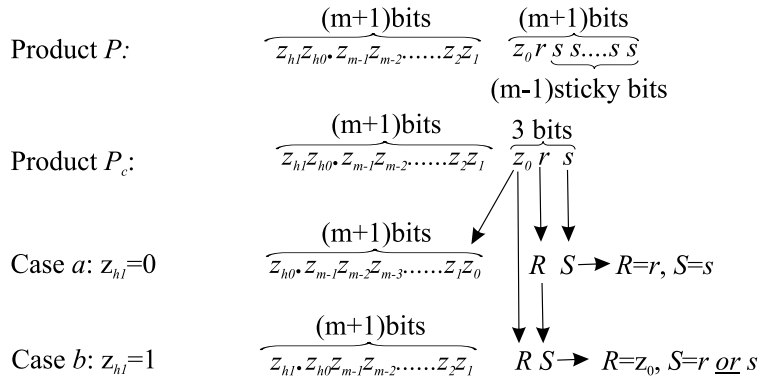


Fig. 5.30 Determining the rounding bits for floating point multiplication

$z_0 r s$, consequently, in total, $(m+4)$ bits. Whatever the form in which the product is obtained, P or P_c, to execute the rounding suboperation, it is necessary to identify the R and S bits, as well as the lsb of the result, bits which are used, as in Fig. 5.30, in the chosen rounding mode implementation. Thus, it is necessary to make a distinction between cases a and b from Fig. 5.30. Consequently, case a, when $z_{h1} = 0$, corresponds to examples b and c from Fig. 5.29, to which there corresponds an extended product on $(2m+1)$ bits, a situation in which, if we suppose the position of the register A msb to be hidden, we require, for normalization purposes, a one position left shifting of the result significand Z_M (as compared with the b and c examples from Fig. 5.29, a difference appears because in these examples the position of the most significant digits of the products are considered to be correct, without shifting, so they become "hidden"). Thus, the result significand is $Z_M = z_{h0} z_{m-1} z_{m-2} \ldots z_1 z_0$, and $R = r$ and $S = s$, where by s is denoted the cumulative sticky bit obtained following an OR operation, at the end of the multiplication of the significand numbers or during it, on the less significant $(m-1)$ bits of the preliminary product. At the end of the rounding suboperation, carry-out may be generated, similar to example c from Fig. 5.29, when a post-normalization, after rounding, is required, by a one position right shifting of the result significand, and the adjustment, through incrementing, of the exponent. On the other hand, case b from Fig. 5.30, with $z_{h1} = 1$, corresponds to example a from Fig. 5.29, when a product on $(2m+2)$ bits is obtained, and, to obtain the normalized form, is necessary to move the point by one position to the left, which is equivalent to a one position right shifting. Thus, the product significand results in the form $Z_M = z_{h1} z_{h0} z_{m-1} z_{m-2} \ldots z_2 z_1$ and, at the same time, it is necessary, for compensation purposes, to increment the exponent value, as well as to modify—as compared to the previous case—the values of the bits that are used in the rounding. Thus, $R = z_0$ and $S = r$ *or* s, and the lsb of the product significand becomes z_1. After establishing the values of the bits involved in rounding, the upward rounding will or will not be applied, as a function of the fulfillment of the conditions specified in the table from Fig. 5.9.

One more observation has to be made regarding signalling the overflow when the rounded result is too great and cannot be represented. Such a situation occurs in single-precision, when the non-biased exponent exceeds the value of 127. As the X_E and Y_E biased exponents have values within the range of integers $[(+1), (+254)]$, according to Fig. 5.28, the range of the tolerated values field for the exponent of the result product $Z_{EP} = X_E + Y_E - 127$ is between $(1 + 1 - 127) = -125$ and $(254 + 254 - 127) = +381$. Since the two numbers can be represented on 9 bits, it results that by using a 9 bit adder in the assessment of the result exponent, we can easily detect the exception status represented by overflow [HePa03].

Referring, more briefly, to the rounding suboperation that corresponds to division [ObFl97], we now recall that for the quotient values result within the $[(+1/2), (+2))$ range, which might require a one position left shift, for normalization. But in this case the same problems discussed for addition appear (refer to Fig. 5.8b), which, synthetically, require the concatenation to the quotient, to the right of the lsb digit, of two more bits having the roles of guard (g), and round (r) bits. Without developing this aspect, we also mention that, in those solutions for division for which a remainder is also generated, its final form is used to deduce the value of the sticky bit [Parh00]. Therefore, the division methods based on convergence have problems concerning rounding, as they do not generate a remainder.

Finally, we shall discuss some aspects regarding the speeding up of the multiplication and division operations. As far as the former is concerned [QuTF04], we start from the fact that the multiplication suboperation on the significands requires a rather high proportion of the total time required by the operation. The idea of not inserting the rounding step as a separate one, after the completion of the preliminary multiplication, but of including the rounding circuitry in the multiplier hardware, parallelizing the suboperations, appears natural. Thus, there is a favorable aspect, that has already been mentioned, according to which the bits involved in the rounding are produced early in the operation cycle. However, the necessity of normalization through one position right shifting comes to be known near to or at the end of multiplication. Since there are only two possibilities, namely, of the postshifting existing or not, after the model covered in detail for addition, two versions of the rounded product are generated at the same time, the correct option being selected in the final step. We can also compute in advance the adjusted exponent values for the two possibilities, the selection of the correct one being made at the moment when it is known whether the normalization postshifting is necessary or not, the last solution being applicable both for multiplication and for division [Parh00]. Alternatively, in multiplication, the rounding can be substituted by the more rapid truncation, but, compensatorily, it is necessary to inject some corrective terms during the operation [EvSe00, JPJH04].

Within the same context of the two operations' performance improvement, we insert the solution according to which increased throughput may be obtained, namely that based on the arithmetic pipeline approach [Poll90]. Thus, both operations consist of several stages or suboperations executed sequentially, and the structure from Fig. 5.27 enables the insertion of some latch devices separating the blocks pointed out in the diagram, so that the various stages can be superposed as presented in

Fig. 5.17, and in Fig. 5.18. Moreover, for the block which multiplies the significands, the idea of separating it into stages and of their superposed functioning can be perpetuated inside the block which is composed of a combinational array represented by a concatenation of CSA adders (Fig. 3.52).

A final observation regarding the implementation of the floating point multiplication and division units, as has resulted from the presentation of the corresponding devices for integers, is that they can have in common a great part of the circuitry [KaTa05, LaAn03], which happens mainly when the division of the significand numbers is performed through methods based on rapid convergence [PiBr02]. In this case, a relatively small quantity of hardware is needed in order to transform a floating point multiplication device into a floating point multiplication/division unit.

Appendix A
Hardware Description Elements

Hardware description practice uses dedicated languages such as VHDL (Very high speed integrated circuits Hardware Description Language), Verilog or System C [Wake00, Haye98], to mention only some of the more widely used ones. This appendix proposes to point out the complexity supported by the use of such a language, thus justifying the choice—for the description of the diagrams in this work—of a pseudo-language meant to make it easy to follow the diagrams' functional aspects.

Thus, choosing the VHDL language (IEEE 1076 standard), we adopt, as the objective of our description, the very simple circuit of a half_adder, some possible implementations of which are given in Fig. 2.5. The problem can be approached on two levels, the entity one, and the architecture one.

Regarding the entity part, this enables the description of the structural element at the highest level, as if the entity represented only one component, without detailing, in any way, its internal architecture. We are interested only in the interconnection interface with other external devices, as in the formal specification from Fig. A.1 [Haye98]. This confers on the adder the name of "half_adder", and names the interconnection signals, of the input and the output, accepted as ports. The difference between inputs and outputs is made by means of the key words in and out. The dimension of each input-output (IO) port, representing the number of signal lines associated with it, is specified through the key word bit, corresponding to one bit. In other words, the half_adder entity from Fig. A.1 has two inputs of one bit, called x and y, and two outputs of one bit, called *sum* and *carry*. The figure also contains the same information given in graphical form. Regarding this representation, the inputs are placed on the left side, and the outputs are placed on the right side, by convention, it not being necessary to use arrows to indicate the signals' transmission direction.

As concerns the architecture part, the VHDL language enables the specification of both the logical behavior, and the internal structure of the described element. We shall refer first to the behavior part, the description of the example half-adder being given in Fig. A.2 [Haye98]. The pair begin-end is used to comprise items which are related between them. From the behavioural point of view, the structure element is perceived as a primitive module or a "black box", whose internal structure is either

M. Vlăduţiu, *Computer Arithmetic*,
DOI 10.1007/978-3-642-18315-7, © Springer-Verlag Berlin Heidelberg 2012

```
entity half_adder is
        port (x,y: in bit; sum,carry: out bit);
end half_adder;
```

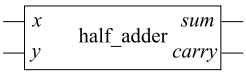


Fig. A.1 VHDL entity level description of a half adder cell

```
architecture behavior of half_adder is
begin
  sum <= x xor y;
  carry <= x and y;
end behavior;
```

Inputs		Outputs	
x	y	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Fig. A.2 VHDL architecture level description of the behaviour of a half adder cell

unknown or is of no interest. The functions corresponding to the two outputs, *sum* and *carry*, are specified through two Boolean equations, xor (EXCLUSIVE-OR) and and (AND), which are pre-defined within the VHDL language [Haye98]. The symbol $\Leftarrow$ is used for the signals assignment, and it indicates the fact that the value of the expression on the right side of the sign is assigned to the signal on the left side of the sign.Thus, $carry \Leftarrow x$ and y shows that to the *carry* signal the Boolean function AND between the variables x and y is assigned. But the language has a certain richness, i.e. it can express the same information in different ways. Thus, the previous statement corresponding to *carry* can be substituted by the following one: if $xy =' 11'$ then $carry \Leftarrow 1$ else $carry \Leftarrow 0$, which corresponds to the information specified graphically in the form of a truth table (Fig. A.2). The VHDL language also allows the communication of performance characteristics, and information about the signals timing respectively. For instance, to indicate that the carry signal appears after a time interval of 5 nsec since the arrival of inputs x and y, there exists the possibility of appealing to a statement structured as follows: $carry \Leftarrow x$ and y after 5 nsec.

Finally, when the interest is in the internal structure of the element, the description can be achieved in a manner similar to the entity part, by specifying, first of all, the components used in the practical implementation of the diagrams. Thus, for the example half-adder we have the description of the structural architecture from Fig. A.3 [Haye98]. There can be identified two types of components described by VHDL statements of component type, that have a similar form to the entity statements. They enable the specification of the component type name (xor_circuit and nand_gate, i.e. an EXCLUSIVE-OR circuit, and a NAND gate respectively), as well as the names and types of IO signals. The internal connections are specified by signal statements, such as the one bit *alpha* signal line (Fig. A.3). There follows the description of the architecture comprised between parentheses (begin-end), where all the copies of each component used in the implementation diagram are pre-

```
architecture structure of half_adder is
        component xor_circuit port(a, b:in bit; c:out bit); end component;
        component nand_gate port(d, e:in bit; f:out bit); end component;
        signal alpha: bit;
begin
        XOR: xor_circuit port map(a=>x, b=>y, c=>sum);
        NAND1: nand_gate port map(d=>x, e=>y, f=>alpha);
        NAND2: nand_gate port map(d=>alpha, e=>alpha, f =>carry);
end structure;
```

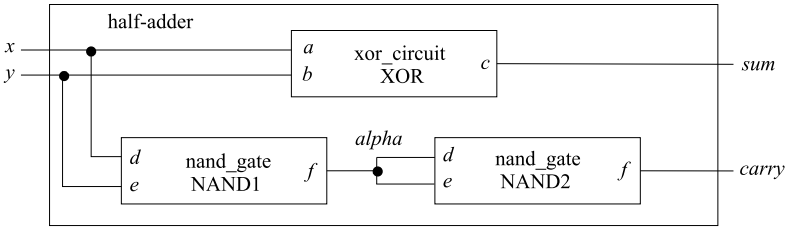
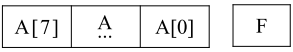


Fig. A.3 VHDL architecture level description of the structure of a half adder cell

Fig. A.4 Registers description using a declare type statement

```
declare register A[7:0], F;
```



sented, by specifying their individual name and their IO connections. This description, which also results from the graphical part given in Fig. A.3, corresponds to the wiring information which is, in fact, the netlist between the components. Thus, for instance, the NAND1 component is of the nand_gate type, and at its *d* and *e* inputs the variables *x* and *y* are supplied, and its output is represented by the *alpha* line.

The above VHDL description for the simple element consisting of a half-adder highlights the richness of the language but shows that ability is required in its handling, which, for most of the diagrams from the previous sections, raises follow up difficulties. Thus, we have appealed to a pseudo-language for the hardware description, which has been inspired by [Haye98], and whose essential characteristic consists in its simpleness. The main rules and conventions of the language used in the description of a large part of the diagrams from this work are as follows:

1. The structural elements consisting of registers are specified through declare register type statements, where the name, the number of ranks and the numbering of the ranks are specified. For instance, Fig. A.4 presents the declaration of two registers, A and F. Regarding register A, it has 8 ranks numbered between 0 and 7, with rank A[7] situated at the extreme left end, and rank A[0] situated at the extreme right end respectively. Regarding F, it represents a register made up of only one storage element (e.g. a flag), whose specification is made only through its name. On the other hand, the declaring of a double length register is also possible, it being given by the juxtaposition of the two names separated by a dot, such as register A.Q. These names are used mainly for shift registers, when the contents of one register is shifted into the other.

2. A similar description is also made for the bus lines, such a specification having the following form: declare bus IOBUS[15:0], i.e. the bus named IOBUS has 16 lines, numbered from 0 to 15. Within this context, we mention that the buses are bidirectional, but to make it easier to follow the informational flow, we consider, sometimes, that there are, in fact, two separate buses, INBUS and OUTBUS, which, actually, are merged.
3. In the description, the elementary operations (the microoperations) are expected to take place in the order provided by the procedure. The non-conflicting operations are separated only by a comma (,) this signifying that they can be issued by the same CLOCK pulse. The microoperations which lead to logic conflicts, if they are not synchronously executed, shall be delayed, i.e. controlled through successive CLOCK pulses, which requires, formally, their separation by a semicolon (;).
4. In order to assign a certain value or a value corresponding to an expression to a certain structure element or to a signal, the sign (:=) is used. For instance, through the A := statement, the initialization microoperation (the adjustment to 0) of the contents of register A is described, controlling, through the signal associated with the microoperation, all the RESET asynchronous inputs of the storage elements of which register A is configured. The above-described microoperation may, selectively, refer to only certain ranks of the register, when they have to be explicitly specified. On the other hand, the microoperation F := (Q[0]and M[7])or F assigns to flag F the value of the expression from the right side of the equality sign, requiring its previous assessment. In the expression there act the Boolean functions and and or, which are considered to be hardwired in the diagram in wiring form.
5. During the description, there can be used statements of unconditioned jump, of go to type, and of conditioned jump, implemented by constructions of if (condition) then type. The target of the jump can be indicated by using labels, which can also be used to aid the clarity of the procedure. An example is the statement if CM $\neq$ 0 then go to ADD, having the significance that when the contents of register CM is different from 0, a jump to label ADD is made, otherwise the sequential execution of the microoperations is respected.
6. Statements of type A[7] := Q[0] ex-or M[7], Q[0] := 0; are allowed, where the rank Q[0] is both read and written, apparently creating a conflicting situation. But the reading is considered to be executed on the rising edge of the CLOCK pulse, and the writing is considered to be executed on the falling one. The pulse period is assumed to be sufficiently long, so that the microoperations, separated only by commas, are not mutually disturbed.

Mention should also be made that, by convention, in the diagram representations a distinction between data paths and control paths has been made, namely for the former solid lines have been used, while for the latter, i.e. for the control signals, dotted lines have been used.

Appendix B
Control Units Synthesis Elements

We stipulate the presentation of certain synthesis methods which are suited to moderate size control units, of the kind encountered as local elementary operations (microoperations) sequencers, included in various devices, such as, for instance, the arithmetic ones described within the above sections. We insert in this appendix only some synthesis elements meant to make it possible to follow the generation of control signals which trigger microoperations that are specific to the arithmetic algorithms. Such local control units are encountered in very many of the block diagrams, starting with that provided in Fig. 3.6 for the sequential multiplication device for binary numbers represented in sign-magnitude, and up to the cumulative diagram from Fig. 5.1 corresponding to an independent arithmetic and logic unit. Without loss of generality, we shall actually refer to the control unit from Fig. 3.12, corresponding to the sequential multiplication device for binary numbers represented in two's complement through James Robertson's procedure.

For clarity reasons, we specify that we aim at the synthesis of the logic which sequences the control signals $c_0, \ldots, c_6$ (refer also to Fig. 3.7), according to the algorithm formally described in Fig. 3.11. As is already known [HePa03, Stal99, Poll90], the synthesis of the control units can in principle be performed through methods based on hardwired logic, and on microprogrammed logic respectively. We shall limit our considerations to the hardwired methods: more precisely, we shall refer to state-table methods, one-hot, and to sequence counter methods [Haye98]. They enable, unlike the computer aided design program products dedicated to large control units [DeMi94], the manual synthesis of control diagrams for the sequencers needed by arithmetic applications.

Roughly characterized, the state-table method allows the most economic design regarding the storage elements required by the synthesis to be obtained. But the combinational logic part is rather intricate and hard to follow and, consequently, the entire solution is not attractive as far as the potential troubleshooting of the diagrams is concerned. The other two methods have a heuristic character, being less rigorous than the previous one and leading to less economic solutions. But the diagrams are easy to follow, justifying, in many cases, the preference for these synthesis approaches [Haye98].

M. Vlăduţiu, *Computer Arithmetic*,
DOI 10.1007/978-3-642-18315-7, © Springer-Verlag Berlin Heidelberg 2012

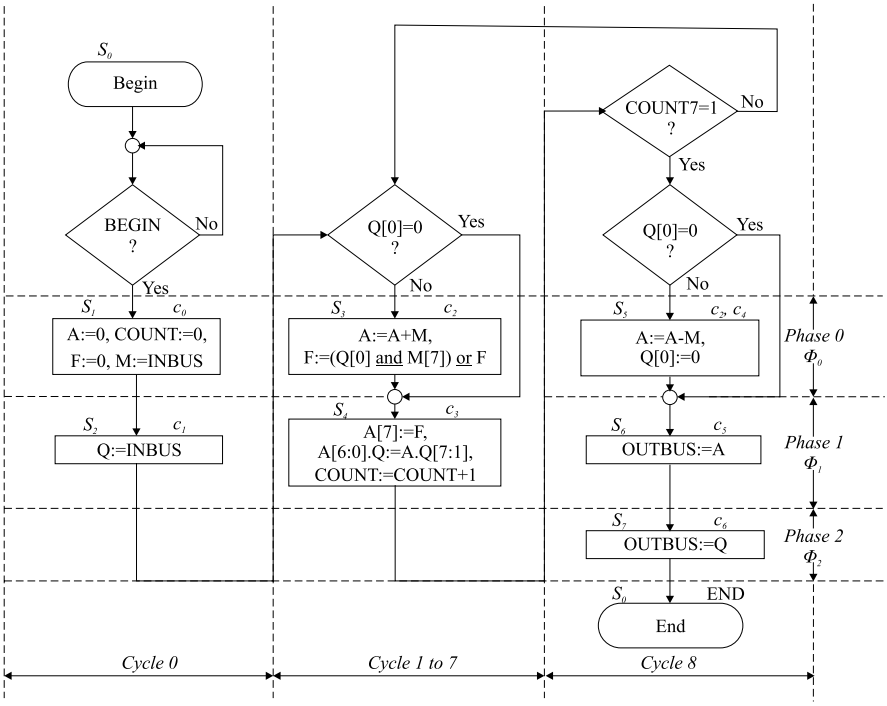


Fig. B.1 Flowchart description of the two's complement binary multiplication based on Robertson's procedure

The starting point of the three methods is common, and consists of the functional description of the algorithm, in our case an arithmetic one, in the form of a flow chart, preferred to a formal description. Thus, the procedure from Fig. 3.11 is presented in Fig. B.1 in its graphical description version [Haye98].

The commentaries regarding the flow chart shall be correlated with those made in Sects. 3.2 and 3.3, i.e. to the operative blocks (marked by rectangles in Fig. B.1) there correspond nonconflictual elementary operations (microoperations), separated by commas in Fig. 3.11, which can be triggered by one and the same control signal c_i. Besides the operative blocks, the flow chart also comprises some testing blocks for a certain condition fulfillment and/or for the activation of a certain state signal (marked by rhombuses in Fig. B.1), they being associated, in our particular case, with the signals BEGIN, Q[0], and COUNT7 (refer to Fig. 3.7). The third category of blocks is represented by the input (Begin) blocks and the output (End) blocks. On the other hand, through it is not compulsory, it is recommended that the blocks are structured in such a manner that they may be partitioned by attributing to a central body of the flow chart those operations that are repetitively executed. It is recommended that this body be flanked on the left by some preparatory blocks for the supply of input data, while on the right there are provided blocks meant to adjust and present the result data. Mention should be made that the elementary oper-

ations allocated to the blocks from the two flanks are executed once, not repeatedly. There are cases in which the requirement of structuring mentioned above cannot be satisfied, but it can be easily fulfilled by the blocks from Fig. B.1, where the repetitive operations corresponding to the cycles 1 to 7 are flanked by those which are executed once in the cycles 0 and 8 respectively. Following the recommended block structuring, or following an arbitrary one, to each operative blocks shall be attributed a so-called phase, the total number of phases being determined by the number of operative blocks corresponding to that part (cycle) of the flow chart with most such blocks. If the number of phases, thus established, exceeds, for a certain part (cycle) of the flow chart, the number of operative blocks, they are allocated to phases in an arbitrary way, or so that the phases may have allocated a balanced number of blocks (which will lead to a balanced loading of the circuits in implementations). Regarding the example from Fig. B.1, the maximum number (three) of phases (denoted from ϕ_0 to ϕ_2) corresponds to cycle 8, and the allocation of the operative blocks from the other cycles has been made only at the first two phases. Obviously, a more judicious assigning, as per the above specification, would have been to attribute one of the operative blocks from cycles 0 or 1 up to 7 to phase 2 (ϕ_2), balancing the phase signal loading (in the case from Fig. B.1, this problem is not critical, but it may become important as the size of the control unit increases).

Regarding the first of the synthesis methods, namely the state-table one, whose procedure steps have also been discussed in the serial adder design (Sect. 2.1), it starts by elaborating the so-called state-table. Thus, to each operative block there is assigned an internal state (denoted by S_i in Fig. B.1) of the future sequential circuit represented by the control unit. There is also an initial state denoted by S_0, corresponding to the Begin and End blocks. To each such internal state, considered as current, a line in the state-table is attributed, while to each potential primary input vector a column is assigned. Since, in the example considered, this vector contains three signals (BEGIN, Q[0], COUNT7—Fig. 3.7), the state-table will have $2^3 = 8$ columns and, incidentally, the number of internal states is also eight (from S_0 through to S_7).

The state-table may be elaborated in the more general form corresponding to a Mealy machine [Wake00, Yarb97], where the elements that are at the intersection of a line with a column are comprised of the next internal state and of the observable vector presented at the outputs of a machine that is in the current internal state corresponding to the line, to which, at the primary inputs, the vector corresponding to the column and a pulse of the CLOCK train is supplied. However, if the output vectors depend only on the current internal states, not on the input combinations, as well, the form of the state-table corresponds to a Moore machine [Wake00, Yarb97]. Thus, regarding the example of the control unit under consideration, the two forms of state-table are given in Fig. B.2a for a Mealy machine, and in Fig. B.2b for a Moore machine respectively. Mention should be made that, except in the case of state S_5, when two control signals (c_2 and c_4) are activated, at the other output vectors only one such signal is active.

Following the filling in—on the basis of the functional description through a flow chart (Fig. B.1)—of the state-table elements in one of the forms from Fig. B.2, for

State Code	Input Vector / State	(BEGIN, Q[0], COUNT7)							
(y_2, y_1, y_0)		000	001	010	011	100	101	110	111
000	S_0	S_0 / END	S_0 / END	S_0 / END	S_0 / END	S_1 / c_0	S_1 / c_0	S_1 / c_0	S_1 / c_0
001	S_1	S_2 / c_1	S_2 / c_1	S_2 / c_1	S_2 / c_1	S_2 / c_1	S_2 / c_1	S_2 / c_1	S_2 / c_1
010	S_2	S_4 / c_3	S_4 / c_3	S_3 / c_2	S_3 / c_2	S_4 / c_3	S_4 / c_3	S_3 / c_2	S_3 / c_2
011	S_3	S_4 / c_3	S_4 / c_3	S_4 / c_3	S_4 / c_3	S_4 / c_3	S_4 / c_3	S_4 / c_3	S_4 / c_3
100	S_4	S_4 / c_3	S_6 / c_5	S_3 / c_2	S_5 / c_2,c_4	S_4 / c_3	S_6 / c_5	S_3 / c_2	S_5 / c_2,c_4
101	S_5	S_6 / c_5	S_6 / c_5	S_6 / c_5	S_6 / c_5	S_6 / c_5	S_6 / c_5	S_6 / c_5	S_6 / c_5
110	S_6	S_7 / c_6	S_7 / c_6	S_7 / c_6	S_7 / c_6	S_7 / c_6	S_7 / c_6	S_7 / c_6	S_7 / c_6
111	S_7	S_0 / END	S_0 / END	S_0 / END	S_0 / END	S_0 / END	S_0 / END	S_0 / END	S_0 / END

a

State Code	Input Vector / State	(BEGIN, Q[0], COUNT7)								Outputs							
(y_2, y_1, y_0)		000	001	010	011	100	101	110	111	c_0	c_1	c_2	c_3	c_4	c_5	c_6	END
000	S_0	S_0	S_0	S_0	S_0	S_1	S_1	S_1	S_1	0	0	0	0	0	0	0	1
001	S_1	S_2	S_2	S_2	S_2	S_2	S_2	S_2	S_2	1	0	0	0	0	0	0	0
010	S_2	S_4	S_4	S_3	S_3	S_4	S_4	S_3	S_3	0	1	0	0	0	0	0	0
011	S_3	S_4	S_4	S_4	S_4	S_4	S_4	S_4	S_4	0	0	1	0	0	0	0	0
100	S_4	S_4	S_6	S_3	S_5	S_4	S_6	S_3	S_5	0	0	0	1	0	0	0	0
101	S_5	S_6	S_6	S_6	S_6	S_6	S_6	S_6	S_6	0	0	1	0	1	0	0	0
110	S_6	S_7	S_7	S_7	S_7	S_7	S_7	S_7	S_7	0	0	0	0	0	1	0	0
111	S_7	S_0	S_0	S_0	S_0	S_0	S_0	S_0	S_0	0	0	0	0	0	0	1	0

b

Fig. B.2 Mealy and Moore state tables corresponding to the local control unit of a sequential two's complement binary multiplier based on Robertson's procedure

the in general n states, the number $\lceil \log_2 n \rceil$ of the state variables y_j are determined, which enable the coding of these states, where the bars $\lceil\ \rceil$ signify the least integer which is larger or equal to the value between the bars. In the particular case analyzed $n = 8$ and $\log_2 8 = 3$, and thus three state variables (y_2, y_1, y_0) are sufficient. Since the eight internal states exhaust all the codes which can be formed with three

Inputs						Outputs												
BEGIN	Q[0]	COUNT7	y_2	y_1	y_0	c_0	c_1	c_2	c_3	c_4	c_5	c_6	*END*	J_2	K_2	J_1	K_1	J_0	K_0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	*d*	0	*d*	0	*d*
0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	*d*	1	*d*	*d*	1
0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	*d*	*d*	1	0	*d*
0	0	0	0	1	1	0	0	0	1	0	0	0	0	1	*d*	*d*	1	*d*	1
≈																			≈
0	1	1	1	0	0	0	0	1	0	1	0	0	0	*d*	0	0	*d*	1	*d*
≈																			≈
1	1	1	1	1	1	0	0	0	0	0	0	0	1	*d*	1	*d*	1	*d*	1

Fig. B.3 Excitation table corresponding to the local control unit of a sequential two's complement binary multiplier based on Robertson's procedure

variables, regarding our example, the coding (column "State code" in the tables from Fig. B.2) is achieved by associating, with each state, the code corresponding to the state index. But, there are, generally, recommendations for favorable codings, mainly when the number of combinations which can be generated for the state variables exceeds the number of internal states [Haye98].

The synthesis goes on with the choosing of the type of the storage elements, to each of them corresponding a state variable. Taking into account the state-table, as well as the characteristic equation of the chosen storage element, the design process goes on with the elaboration of the so-called excitation table. It has as inputs the union of the primary inputs subvector (in our case, BEGIN, Q[0] and COUNT7) with the state subvector (in our case, y_2 through to y_0), and it has as outputs the union of the observable outputs subvector (in our case, from c_0 through to c_6 and *END*) with the so-called excitation variables subvector, this latter representing the synchronous inputs of the storage elements. Thus, in case flip-flop JK is chosen, with its characteristic equation $w(t+1) = J\overline{w(t)}\ \underline{or}\ \overline{K}w(t)$ (where J and K are the synchronous inputs, and $w(t)$ and $w(t+1)$ represent the current and the next states, after the CLOCK pulse), a part of the excitation table (having in its complete form $2^6 = 64$ lines) is given in Fig. B.3, where the synchronous inputs, and the flip-flops' outputs have been denoted by (J_2, K_2), (J_1, K_1), (J_0, K_0), and (y_2, y_1, y_0), and a don't care logic value has been denoted by d.

Starting from the excitation table, for each output (from c_0 through to K_0—Fig. B.3) the Boolean equations are written in normal disjunctive form, thus con-

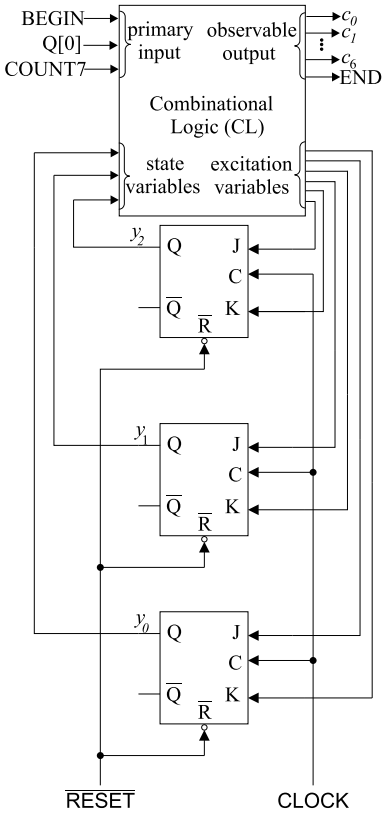


Fig. B.4 Conceptual diagram corresponding to the state table design version for the local control unit of a Robertson multiplier

necting, through an OR operation, all the canonical terms which correspond to the binary units (the don't care terms included) from the column of each output. In our case, there will result 14 such Boolean equations, one of which, for instance, for the control signal c_3, has the following form:

$$c_3 = \overline{BEGIN}\,\overline{Q[0]}\,\overline{COUNT7}\,\overline{y_2}\,y_1\,\overline{y_0}\ \underline{or}\ \overline{BEGIN}\,\overline{Q[0]}\,\overline{COUNT7}\,\overline{y_2}\,y_1\,y_0\ \underline{or}\ \ldots \quad \text{(B.1)}$$

To each Boolean equation of type (B.1) minimization methods are applied [Wake00, Yarb97], which enable economic expressions to be obtained, on the basis of which the practical implementation can be made. Thus, the combinational logic part results corresponding to the sequential circuit which is the control unit. Consequently, for our example we have the principle diagram from Fig. B.4. The storage elements, denoted from B_2 through to B_0, are controlled either synchronously, in cadence with the CLOCK, through the excitation variables, or asynchronously, through the $\overline{RESET}$ signal. Equation (B.1) type equations, following the minimization operation, stand at the basis of the technological implementation of the combinational logic (CL) circuit from Fig. B.4, a part of the control unit.

Commenting hereafter on the state-table, let us resume the synthesis activity from the stage at which the storage element was chosen, the choice being this time a flip-

flop of type D (whose characteristic equation is $w(t+1) = D$). Then the elaboration of the excitation table follows, which can be done rigorously according to the description given in Fig. B.3, but which can also be done by starting, for instance, from the state-table from Fig. B.2b and from the characteristic equation of the chosen storage element, and in a less rigorous, ad hoc manner. The motivation of such a procedure also consists in the fact that, in the performance achieved by modern manufacturing technology of integrated circuits regarding the packing density of the electronic components, the minimization criterion of the Boolean equations synthesis loses its importance, becoming secondary. Thus, for our example the ad hoc excitation table given in Fig. B.5 results. The generation of this table has been done in a minimized form by following of the transitions corresponding to the current internal states. It can be observed that the first line of the table from Fig. B.5 corresponds to the line S_1 (001) of the table from Fig. B.2, which, for any combination of the input variables (BEGIN, Q[0], COUNT7), performs a transition into the state S_2 (010); consequently, the respective variables can be omitted. The internal states from Fig. B.2b are marked, in Fig. B.5, in the order of the number of transitions into various next internal states which they bring about. In this way, in Fig. B.5, the last four lines are dedicated to the state S_4 which passes through four different next states (from S_3 through to S_6). Starting from the excitation table from Fig. B.5, the Boolean equations for outputs in a near minimal form can be deduced. Thus, for instance, for the control signal c_2, there results:

$$c_2 = \overline{y_2} y_1 y_0 \ \underline{or} \ y_2 \overline{y_1} y_0 \tag{B.2}$$

The equations of type (B.2) can, possibly, undergo certain supplementary minimization processings, being followed by the practical implementation executed in the manner stipulated in Fig. B.4.

Passing to the one-hot method, the synthesis starting point is represented by the same functional description flow chart from Fig. B.1. As mentioned above, the state-table method, sometimes also called the classic method [Haye98], is characterized by the fact that it minimizes the number of storage elements, but the structure of the combinational logic, also called random logic, is generally intricate, the post-execution maintenance being rather difficult to accomplish. Alternatively, the one-hot method is generally based on the assigning of a storage element to each state from the description flow chart (Fig. B.1). Consequently, a "waste" results, which is compensated by the rather simple structure of the random logic. At a certain moment, only one of the storage elements is in the logic state 1 (i.e. only one element is in "hot" state—one-hot), the others being in the logic state 0. There being, generally, a one-to-one correspondence between the number of states and that of the storage elements, this method can be favorably applied only to designs with a small number of states. The important characteristic of the synthesis consists of the fact that the equations for the next internal states, as well as those for the observable outputs, can be directly deduced from the functional description flow chart. Applying this method to our example, we shall use, as storage elements, flip-flops of type D, whose outputs (Q) will be denoted by B_0 through to B_7, and thus the states coding

Inputs						Outputs										
BEGIN	Q[0]	COUNT7	y_2	y_1	y_0	c_0	c_1	c_2	c_3	c_4	c_5	c_6	END	D_2	D_1	D_0
-	-	-	0	0	1	1	0	0	0	0	0	0	0	0	1	0
-	-	-	0	1	1	0	0	1	0	0	0	0	0	1	0	0
-	-	-	1	0	1	0	0	1	0	1	0	0	0	1	1	0
-	-	-	1	1	0	0	0	0	0	0	1	0	0	1	1	1
-	-	-	1	1	1	0	0	0	0	0	0	1	0	0	0	0
0	-	-	0	0	0	0	0	0	0	0	0	0	1	0	0	0
1	-	-	0	0	0	0	0	0	0	0	0	0	1	0	0	1
-	0	-	0	1	0	0	1	0	0	0	0	0	0	1	0	0
-	1	-	0	1	0	0	1	0	0	0	0	0	0	0	1	1
-	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0
-	0	1	1	0	0	0	0	0	1	0	0	0	0	1	1	0
-	1	0	1	0	0	0	0	0	1	0	0	0	0	0	1	1
-	1	1	1	0	0	0	0	0	1	0	0	0	0	1	0	1

Fig. B.5 Ad-hoc excitation table corresponding to the local control unit of a Robertson multiplier

will immediately result, on the basis of what has been specified above. Thus, to the state S_0 the state vector $(B_0, B_1, B_2, \ldots, B_7) = (1, 0, 0 \ldots, 0)$ corresponds, while to the state S_7 the state vector $(B_0, B_1, B_2, \ldots, B_7) = (0, 0, 0 \ldots, 1)$ corresponds. Based on the transitions between the operative blocks corresponding to the states from Fig. B.1, the Boolean equations for the synchronous inputs (D) of the state

storage element, as well as the Boolean equations for the observable outputs result:

$$\begin{aligned}
D_0 &= \mathrm{B}_0\,\overline{\mathrm{BEGIN}}\,\underline{or}\,\mathrm{B}_7 & c_0 &= \mathrm{B}_1\\
D_1 &= \mathrm{B}_0\,\mathrm{BEGIN} & c_1 &= \mathrm{B}_2\\
D_2 &= \mathrm{B}_1 & c_2 &= \mathrm{B}_3\,\underline{or}\,\mathrm{B}_5\\
D_3 &= \mathrm{B}_2\,\overline{\mathrm{Q[0]}}\,\underline{or}\,\mathrm{B}_4\,\mathrm{Q[0]}\,\overline{\mathrm{COUNT7}} & c_3 &= \mathrm{B}_4\\
D_4 &= \mathrm{B}_2\,\overline{\mathrm{Q[0]}}\,\underline{or}\,\mathrm{B}_4\,\overline{\mathrm{Q[0]}}\,\overline{\mathrm{COUNT7}}\,\underline{or}\,\mathrm{B}_3 & c_4 &= \mathrm{B}_5\\
D_5 &= \mathrm{B}_4\,\mathrm{Q[0]}\,\mathrm{COUNT7} & c_5 &= \mathrm{B}_6\\
D_6 &= \mathrm{B}_4\,\overline{\mathrm{Q[0]}}\,\mathrm{COUNT7}\,\underline{or}\,\mathrm{B}_5 & c_6 &= \mathrm{B}_7\\
D_7 &= \mathrm{B}_6 & END &= \mathrm{B}_0
\end{aligned} \tag{B.3}$$

In Eqs. (B.3), it can be observed that the control signal c_2, which is generated in S_3 and in S_5, has two terms, the other output equations having only one term represented by the storage element which is set up in the corresponding internal state. Based on Eqs. (B.3), the synthesis of the logic diagrams corresponding to the control unit results immediately, it being given, for our case, in Fig. B.6, where, for the implementation of Eqs. (B.3), AND and OR gates have been used.

As can be observed in Fig. B.6, the resulting random logic is simple, this representing an important advantage. However, the synthesis solutions obtained through the one-hot method suffer, mainly when there is a large number of states, because the synchronous application of the CLOCK to all the storage elements may present the undesired phenomenon of clock skew, due to the delays that can occur on long wires. This clock skew determines the delayed control of the storage elements, which leads to the diagram's unsteady and unreliable operation. This phenomenon is more noticeable for a variation of the one-hot method, known as the delay element method [Haye98].

Passing to the third version of wired synthesis, that based on the sequence counter method, mention should be made that this also enables a logic diagram whose random part is easy to follow to be obtained, but implies a larger investment in the storage elements as compared to the classical method of the state-table [Haye88]. The central structure component of the synthesis is the so-called sequence counter, which also gives its name to the method. Its function consists of the generation of so-called phase pulses, whose essential characteristic is that they are non-overlapping, and which are separated by a CLOCK train period. At block level, the sequence counter contains the components given in Fig. B.7a, and there can be identified the Start/Stop (S/S) flip-flop of type SR, the modulo-m counter, and the 1-out-of-m decoder. Thus, the modulo-m counter is initialized either when the S/S flip-flop changes to Stop (End is activated on input R), or on an external Reset signal, and then it counts the CLOCK pulses only when the S/S flip-flop changes to Start (Begin is activated on input S). Each state of the counter is decoded at a CLOCK pulse, so that at the decoder outputs, denoted from ϕ_0 through to ϕ_{m-1}, the phase pulses are obtained. Figure B.7b presents the symbol of the modulo-m sequence counter, and in Fig. B.7c are given the pulse trains from ϕ_0 through to ϕ_{m-1}, delayed by the period T of the CLOCK, and which are non-overlapping.

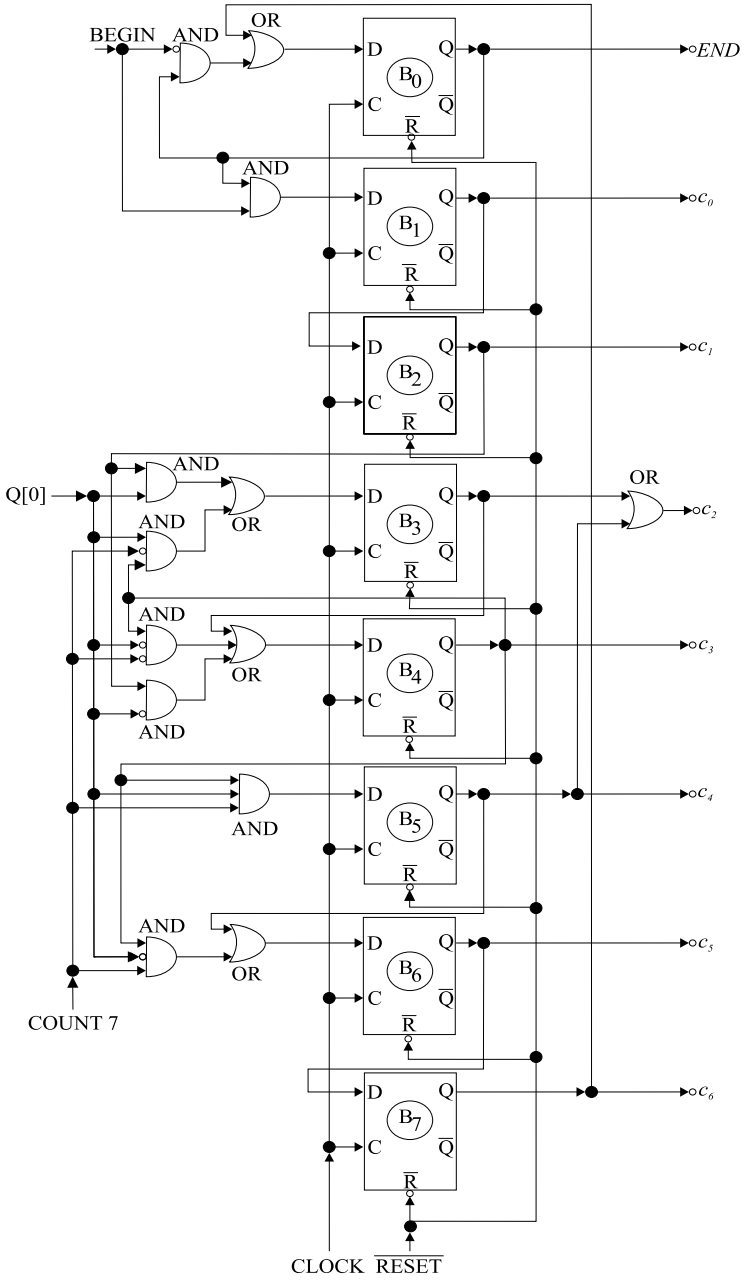


Fig. B.6 Detailed diagram at the gate level corresponding to the one-hot design version for the local control unit of a Robertson multiplier

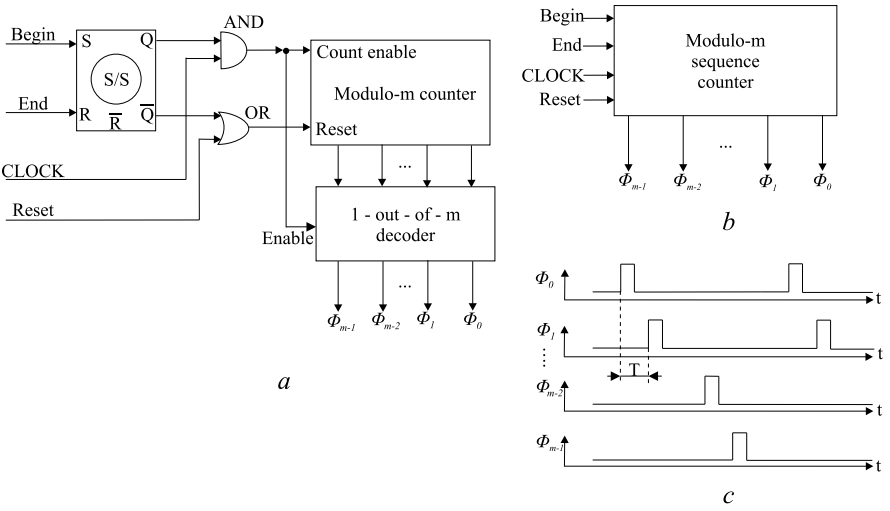


Fig. B.7 Structural elements of a sequence counter

The design activity starts by dimensioning the sequence counter. The number of delayed pulses is given by the number of phases identified on the functional flow chart (Fig. B.1). Regarding our example, we have only the phases from ϕ_0 through to ϕ_2, consequently, the counter from Fig. B.7a will be modulo 3, synthetized with two storage elements. The design goes on by the assigning—to each partition that resulted after the flow chart structuring—of an SR type flip-flop, so that, at a certain moment, only one of them is set, all the others being reset. In the flow chart from Fig. B.1, which is divided into the three recommended partitions, the three necessary SR flip-flops result.

Then, the synthesis stage of the random logic follows, which consists, essentially, of AND gates that must have as inputs a phase pulse and an output corresponding to the SR flip-flop associated with the flow chart partitions. As applicable, also among the inputs to the AND gates inputs may be external signals, which are primary inputs for the control unit. By means of AND gates the control signals representing the observable outputs are generated. If a certain control signal corresponds to two or more states, that signal is obtained through an OR gate. Applying what has been specified to the example of the control unit for the Robertson multiplication device, the design version from Fig. B.8 (after [Haye88]) results. The three SR flip-flops have been denoted by B_0 through to B_2, B_0 being set by the external signal BEGIN, which also starts the CLOCK counting by the modulo-3 sequence counter. For the resetting of B_0, at the same time as the setting of B_1, and for the subsequent resetting of B_1 and setting of B_2, AND gates controlled through the phase pulse ϕ_2 have been used. Incidentally, this is because in the functional flow chart operative blocks are not provided, consequently phase pulse ϕ_2 is not used. The AND gates level, which

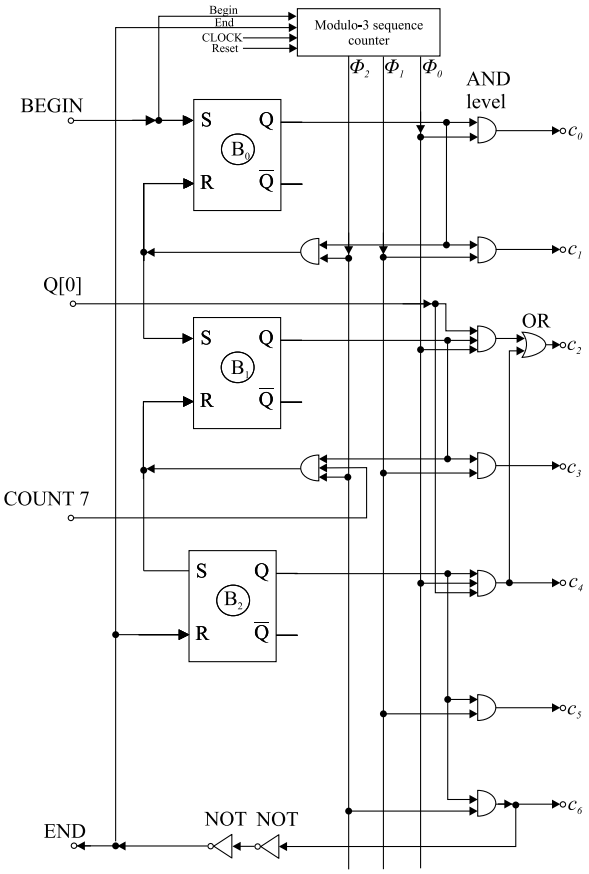


Fig. B.8 Detailed diagram at the gate level corresponding to the sequence counter design version for the local control unit of a Robertson multiplier

generates the control signals, implements the following logic equations

$$\begin{aligned} c_0 &= \mathrm{B}_0\,\phi_0 & c_4 &= \mathrm{B}_2\,\phi_0\,\mathrm{Q}[0] \\ c_1 &= \mathrm{B}_0\,\phi_1 & c_5 &= \mathrm{B}_2\,\phi_1 \\ c_2 &= \mathrm{B}_1\,\phi_0\,\mathrm{Q}[0] & c_6 &= \mathrm{B}_2\,\phi_2 \\ c_3 &= \mathrm{B}_1\,\phi_1\,\mathrm{Q}[0]\,\underline{or}\,\mathrm{B}_1\,\phi_1\,\overline{\mathrm{Q}[0]} = \mathrm{B}_1\,\phi_1 \end{aligned} \tag{B.4}$$

In their elaboration, the functional particularities of the controlled device have to be taken into account. Thus, in case of the equations (B.4), for c_2 and c_3, there has not been provided conditioning through $\overline{\mathrm{COUNT7}}$, because we have assumed that the CLOCK period covers the interval between the application of the incrementing signal c_3 to the iterations counter COUNT and the generation of the COUNT7 signal (Fig. 3.12). In this way, through COUNT7 there is ensured the output from the repetitive cycle, and the input in the final cycle, conditioning the AND gate which resets B_1 and sets up B_2. In case the values of the parameters used in the implementation do not warrant the hypothesis, the equations (B.4) undergo certain modifications. We should also like to mention the way the *END* signal is generated,

which, besides the signalling completion of the operation performed by to the exterior of the device, also determines the reset of the B_2 cycle storage element, and the reset of the sequence counter (through the reset of the S/S flip-flop). This signal has been obtained from c_6 through the delay assured by the serialization of two inverter gates, through which, it has been assumed, covers the time interval required to annihilate the effect determined by the signal c_6 (the bus delivery of the less significant part of the product). If this hypothesis is not confirmed by the catalogue data of the employed circuits, the delay shall be, again, lengthened by supplementary pairs of inverter gates.

The last comments are meant to highlight the flexibility of the design solution offered by the sequence counter method, as well as the requirement to correlate it with the technology of the circuits used in the implementation, an aspect which, as a matter of fact, is common also to the other synthesis methods.

References

[AbBF90] Miron Abramovici, Melvin Breuer, Arthur Friedman: "Digital System Testing and Testable Design" Computer Science Press, New York, 1990.

[ALMN05] Elisardo Antelo, Tomas Lang, Paolo Montuschi, Alberto Nannarelli: "Digit-Recurrence Dividers with Reduced Logical Depth" IEEE Trans. Comput., vol. 54, no. 7, 2005, pp. 837–851.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, Carl Landwehr: "Basic Concepts and Taxonomy of Dependable and Secure Computing" IEEE Trans. Dependable Secure Comput., vol. 1, no. 1, 2004, pp. 11–33.

[BrO'H03] Randal E. Bryant, David O'Hallaron: "Computer Systems. A Programmer's Perspective" Pearson Education International, Upper Saddle River, 2003.

[BoTi05] Nicolas Boullis, Arnaud Tisserand: "Some Optimizations of Hardware Multiplication by Constant Matrices" IEEE Trans. Comput., vol. 54, no. 10, 2005, pp. 1271–1282.

[COPR06] Gian Carlo Cardarilli, Marco Ottavi, Salvatore Pontarelli, Marco Re, Adelio Salsano: "Fault Localization, Error Correction, and Graceful Degradation in Radix 2 Signed Digit-Based Adders" IEEE Trans. Comput., vol. 55, no. 5, 2006, pp. 534–539.

[DaTa05] Albert Danysh, Dimitri Tan: "Architecture and Implementation of a Vector/SIMD Multiply-Accumulate Unit" IEEE Trans. Comput., vol. 54, no. 3, 2005, pp. 284–293.

[DeMi94] Giovanni De Micheli: "Synthesis and Optimization of Digital Circuits" McGraw-Hill International Editions, New York, 1994.

[EfVN03] Costas Efstathiou, Haridimos T. Vergos, Dimitris Nikolos: "Modulo $2^n \pm 1$ Adder Design Using Select-Prefix Blocks" IEEE Trans. Comput., vol. 52, no. 11, 2003, pp. 1399–1406.

[ErLa94] Miloš D. Ercegovac, Tomas Lang: "Division and Square Root: Digit Recurrence Algorithms and Implementations" Kluwer Academic, Dordrecht, 1994.

[ErLa04] Miloš D. Ercegovac, Tomas Lang: "Digital Arithmetic" Morgan Kaufmann, San Mateo, 2004.

[EvSe00] Guy Even, Peter-Michael Seidel: "A Comparison of Three Rounding Algorithms for IEEE Floating-Point Multiplication" IEEE Trans. Comput., vol. 49, no. 7, 2000, pp. 638–650.

[GoSA06] Mustafa Gok, Michael J. Schulte, Mark G. Arnold: "Integer Multipliers with Overflow Detection" IEEE Trans. Comput., vol. 55, no. 8, 2006, pp. 1062–1066.

[Haye88] John P. Hayes: "Computer Architecture and Organization" McGraw-Hill, New York, Second Edition, 1988.

[Haye98] John P. Hayes: "Computer Architecture and Organization" McGraw-Hill, New York, Third Edition, 1998.

[HePa94] John L. Hennessy, David A. Patterson: "Computer Organization and Design. The Hardware/Software Interface" Morgan Kaufmann, San Mateo, 1994.

DOI 10.1007/978-3-642-18315-7,

[HePa03] John L. Hennessy, David A. Patterson: “Computer Architecture. A Quantitative Approach” Morgan Kaufmann, San Mateo, Third Edition, 2003; Appendix H: Computer Arithmetic by David Goldberg.

[HuEr05] Zhijun Huang, Miloš D. Ercegovac: “High-Performance Low-Power Left-to-Right Array Multiplier Design” IEEE Trans. Comput., vol. 54, no. 3, 2005, pp. 272–283.

[ITRS01] International Technology Roadmap for Semiconductors-Interconnect, 2001.

[JPJH04] Jong-Chul Jeong, Woo-Chan Park, Woong Jeong, Tack-Don Han, Moon-Key Lee: “A Cost-Effective Pipelined Divider with a Small Lookup Table” IEEE Trans. Comput., vol. 53, no. 4, 2004, pp. 489–495.

[KaGa06] Jung-Yup Kang, Jean-Luc Gaudiot: “A Simple High-Speed Multipier Design” IEEE Trans. Comput., vol. 55, no. 10, 2006, pp. 1253–1258.

[Kaha97] W. Kahan: “Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic” October 1997, http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF.

[KaTa05] Marcelo E. Kaihara, Naofumi Takagi: “A Hardware Algorithm for Modular Multiplication/Division” IEEE Trans. Comput., vol. 54, no. 1, 2005, pp. 12–21.

[KeSc05] Robert D. Kenney, Michael J. Schulte: “High-Speed Multioperand Decimal Adders” IEEE Trans. Comput., vol. 54, no. 8, 2005, pp. 953–963.

[KoMu06] Peter Kornerup, Jean-Michel Muller: “Leading Guard Digits in Finite Precision Redundant Representations” IEEE Trans. Comput., vol. 55, no. 5, 2006, pp. 541–548.

[Kore93] Israel Koren: “Computer Arithmetic Algorithms” Prentice Hall International, Englewood Cliffs, 1993.

[Kore02] Israel Koren: “Computer Arithmetic Algorithms” A.K. Peters, Wellesley, Second Edition, 2002.

[Korn03] Peter Kornerup: “Revisiting SRT Quotient Digit Selector” Proc. 16th IEEE Symp. Computer Arithmetic, 2003, pp. 38–45.

[Korn05] Peter Kornerup: “Digit Selection for SRT Division and Square Root” IEEE Trans. Comput., vol. 54, no. 3, 2005, pp. 294–303.

[Kuli02] Ulrich W. Kulisch: “Advanced Arithmetic for the Digital Computer. Design of Arithmetic Units” Springer, Berlin, 2002.

[LaAn03] Tomas Lang, Elisardo Antelo: “Radix-4 Reciprocal Square Root and Its Combination with Division and Square Root” IEEE Trans. Comput., vol. 52, no. 9, 2003, pp. 1100–1114.

[Ober99] Stuart F. Oberman: “Floating Point Division and Square Root Algorithms and Implementation in the AMD-K7 Microprocessor” Proc. 14th Symp. Computer Arithmetic (ARITH 14), 1999, pp. 106–115.

[ObFl97] Stuart F. Oberman, Michael J. Flynn: “Division Algorithms and Implementations” IEEE Trans. Comput., vol. 46, no. 8, 1997, pp. 833–854.

[Oliv01] Mauro Olivieri: “Design of Synchronous and Asynchronous Variable-Latency Pipelined Multipliers” EEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 9, no. 2, 2001, pp. 365–376.

[Omon94] Amos R. Omondi: “Computer Arithmetic Systems. Algorithms, Architecture and Implementations” 1994. C.A.R. Hoare Series Editor.

[PaHe96] David A. Patterson, John L. Hennessy: “Computer Architecture. A Quantitative Approach” Morgan Kaufmann, Dordrecht, Second Edition, 1996; Appendix A: Computer Arithmetic by David Goldberg.

[Parh00] Behrooz Parhami: “Computer Arithmetic. Algorithms and Hardware Designs” Oxford University Press, London, 2000.

[Parh03] Behrooz Parhami: “Tight Upper Bounds on the Minimum Precision Required of the Divisor and the Partial Remainder in High-Radix Division” IEEE Trans. Comput., vol. 52, no. 11, 2003, pp. 1509–1514.

[PiBr02] Jose-Alejandro Piñeiro, Javier D. Bruguera: “High-Speed Double-Precision Computation of Reciprocal, Division, Square Root and Inverse Square Root” IEEE Trans. Comput., vol. 51, no. 12, 2002, pp. 1377–1388.

[Poll90] L. Howard Pollard: "Computer Design and Architecture" Prentice-Hall International, Englewood Cliffs, 1990.
[QuTF04] Nhon T. Quach, Naofumi Takagi, Michael J. Flynn: "Systematic IEEE Rounding Method for High-Speed Floating-Point Multipliers" IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 12, no. 5, 2004, pp. 511–521.
[RaCa06] Sridhar Rajagopal, Joseph R. Cavallaro: "Truncated On-line Arithmetic with Applications to Communication Systems" IEEE Trans. Comput., vol. 55, no. 10, 2006, pp. 1240–1252.
[RaFu89] T.R.N. Rao, E. Fujiwara: "Error-Control Coding for Computer Systems" Prentice-Hall International, Englewood Cliffs, 1989.
[RaPe96] Jan M. Rabaey, Massored Pedram: "Low Power Design Methodologies" Kluwer Academic, Dordrecht, 1996.
[RaTy98] Janusz Rajski, Jerzy Tyszer: "Arithmetic Built-In Self-Test for Embedded Systems" Prentice Hall, New York, 1998.
[ScST03] Eric M. Schwarz, Martin Schmookler, Son Dao Trong: "Hardware Implementations of Denominalized Numbers" Proc. 16th IEEE Symposium on Computer Arithmetic (Arith 16), 2003, pp. 70–78.
[ScST05] Eric M. Schwarz, Martin Schmookler, Son Dao Trong: "FPU Implementations with Denormalised Numbers" IEEE Trans. Comput., vol. 54, no. 7, 2005, pp. 825–836.
[SeMM05] Peter-Michael Seidel, Lee D. McFearin, David W. Matula: "Secondary Radix Recodings for Higher Radix Multipliers" IEEE Trans. Comput., vol. 54, no. 2, 2005, pp. 111–123.
[SeEv01] Peter-Michael Seidel, Guy Even: "On the Design of Fast IEEE Floating-Point Adders" Proc 15th IEEE Symposium on Computer Arithmetic (Arith 15), 2001, pp. 184–194.
[SeEv04] Peter-Michael Seidel, Guy Even: "Delay-Optimized Implementation of IEEE Floating-Point Addition" IEEE Trans. Comput., vol. 53, no. 2, 2004, pp. 97–113.
[Stal99] William Stallings: "Computer Organization and Architecture. Designing for Performance" Prentice Hall International, Englewood Cliffs, 1999.
[TaYY85] N. Takagi, N.H. Yasuura, S. Yajima: "High-Speed VSLI Multiplication Algorithm with a Redundant Binary Addition Tree" IEEE Trans. Comput., vol. 34, no. 9, 1985, pp. 789–796.
[VeEN02] Haridimos T. Vergos, Costas Efstathiou, Dimitris Nikolos: "Diminished-One Modulo $2^n + 1$ Adder Design" IEEE Trans. Comput., vol. 51, no. 12, 2002, pp. 1389–1399.
[ViLG06] Julio Villalba, Tomas Lang, Mario A. Gonzales: "Double-Residue Modular Range Reduction for Floating-Point Hardware Implementations" IEEE Trans. Comput., vol. 55, no. 3, 2006, pp. 254–267.
[Vlăd82] Mircea Vlăduţiu: "Tehnologie de ramură şi fiabilitate" Litografia Institutului Politehnic, Timişoara, 1982.
[Vlăd86] Mircea Vlăduţiu: "Tehnica testării sistemelor de calcul" Litografia Institutului Politehnic, Timişoara, 1986.
[VlPe94] M. Vlăduţiu, N. Petrakis: "Adapted Combinational Array for Exact Binary Division with Signed Operands" International Conference on Technical Informatics, Proceedings vol. 5, Timişoara, 1994, pp. 1–10.
[Wake00] John F. Wakerly: "Digital Design. Principles and Practices" Prentice-Hall, New York, 2000.
[Yarb97] John M. Yarbrough: "Digital Logic. Application and Design" West Publishing Company, Eagan, 1997.
[YeJe03] Wen-Chang Yeh, Chein-Wei Jen: "Generalized Earliest-First Fast Addition Algorithm" IEEE Trans. Comput., vol. 52, no. 10, 2003, pp. 1233–1242.
[***08] "IEEE Standard for Floating Point Arithmetic" http://ali.ayad.free.fr/IEEE_2008.pdf.