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

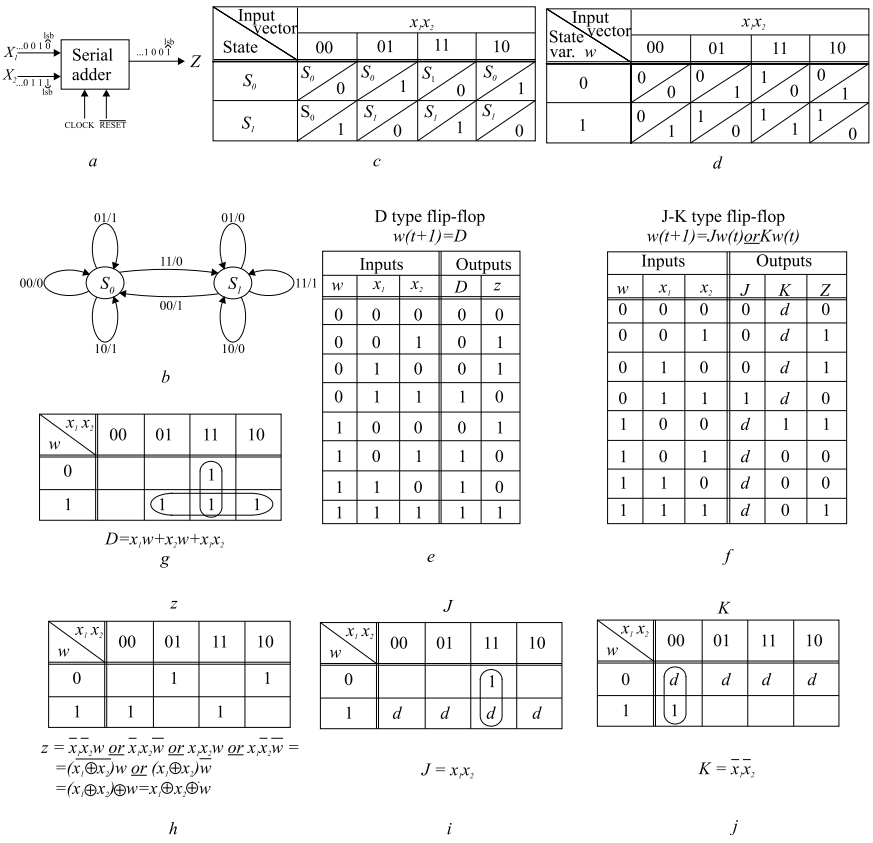


Fig. 2.1 Sequence of design steps for a serial adder

(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

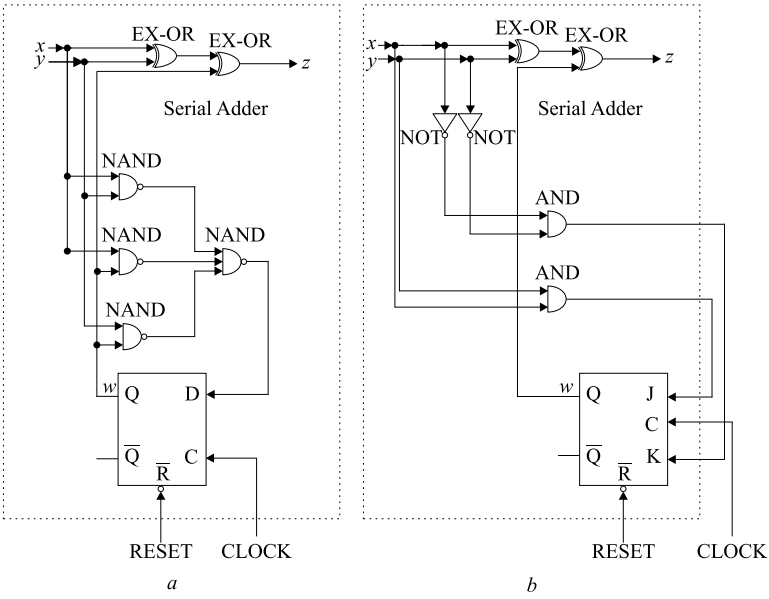


Fig. 2.2 Gate level implementation versions for a serial adder

(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 $D = wx$ *or* xy *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 $z = \overline{x}w\overline{y}$ or $\overline{w}y\overline{x}$ or $\overline{w}x\overline{y}$ 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.

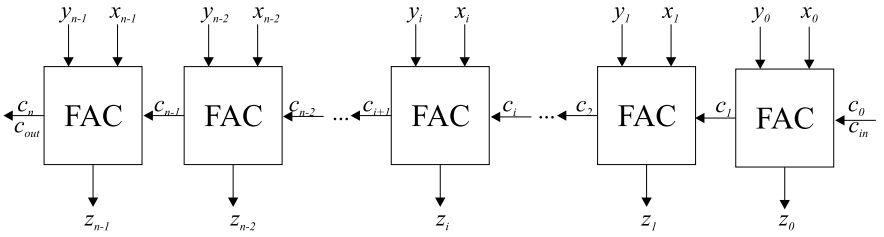


Fig. 2.3 Block diagram of a ripple carry adder

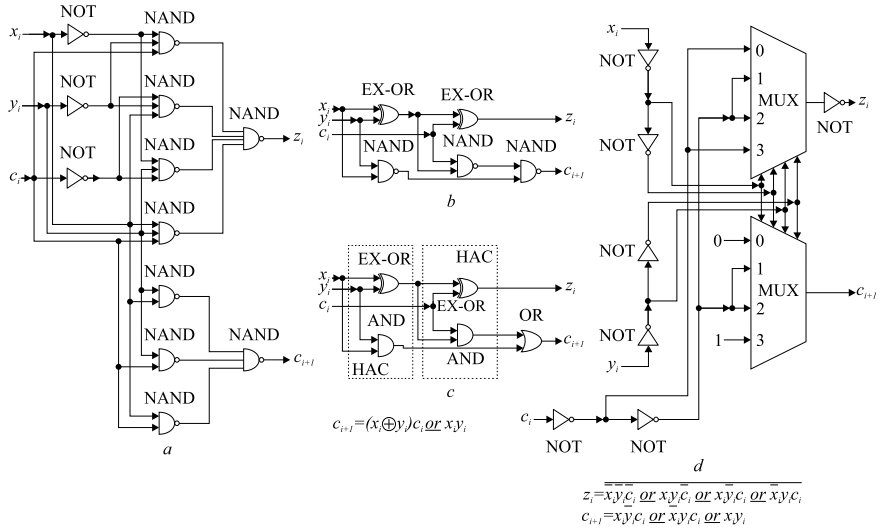


Fig. 2.4 Gate level implementation versions for a full adder cell

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

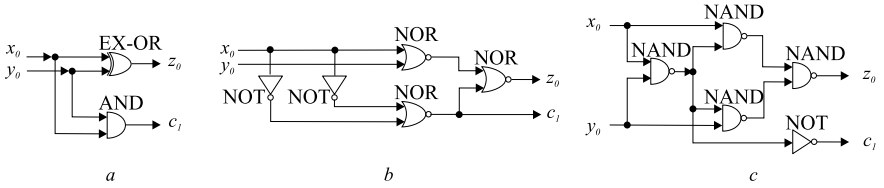


Fig. 2.5 Gate level implementation versions for a half adder cell

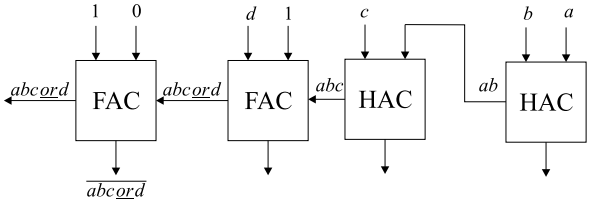


Fig. 2.6 Logical function implementation example using FACs and HACs

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:

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

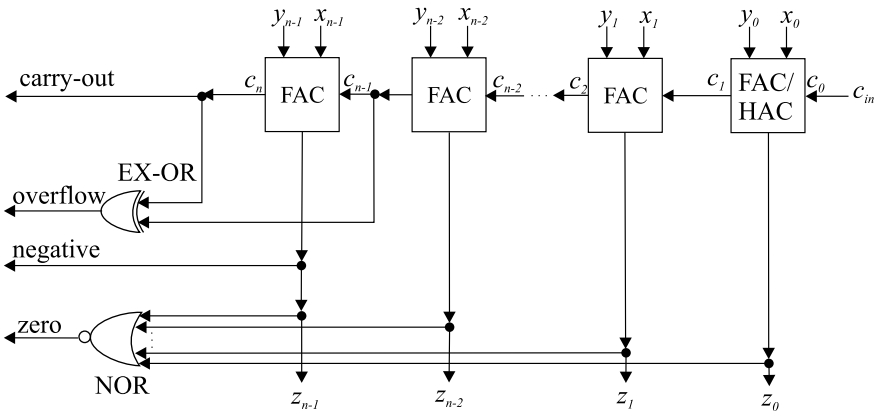


Fig. 2.7 Testing of exception conditions for an adder

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 $p_i = x_i \overline{y_i}$ *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

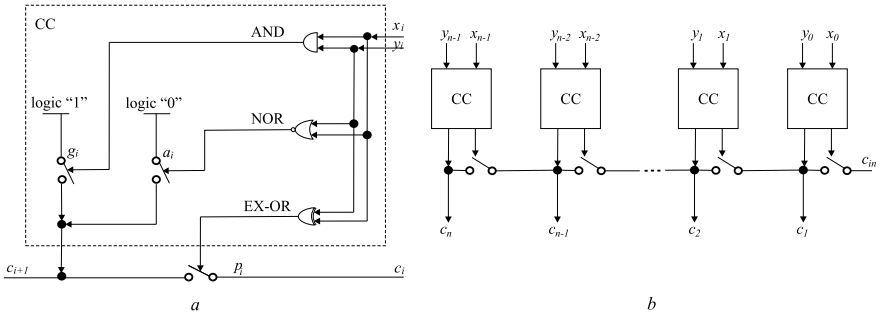


Fig. 2.8 Conceptual diagram of a Manchester adder cell and their interconnection

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

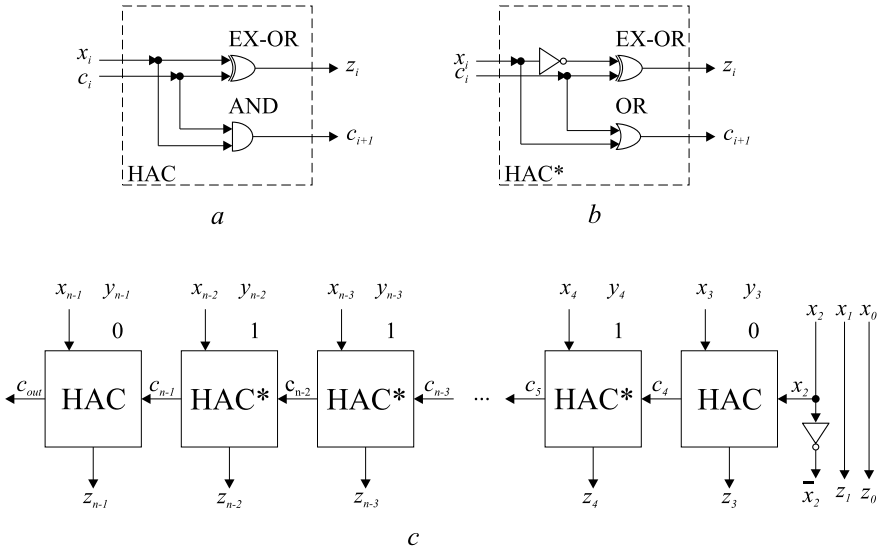


Fig. 2.9 Elements of addition of a constant to an operand

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

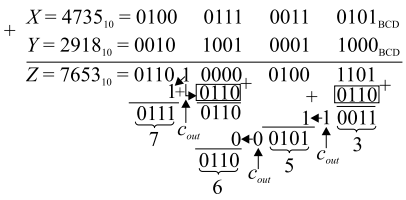


Fig. 2.10 BCD8421 code addition example

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.

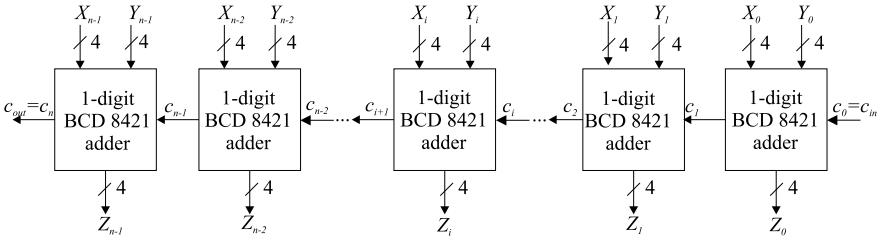


Fig. 2.11 RCA version of an BCD8421 decimal adder block diagram

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

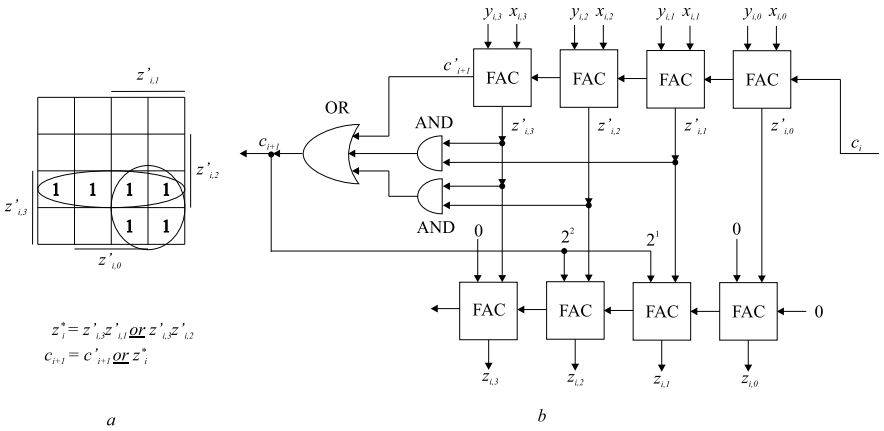


Fig. 2.12 Synthesis of a tetrad adder used in BCD8421 addition

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.

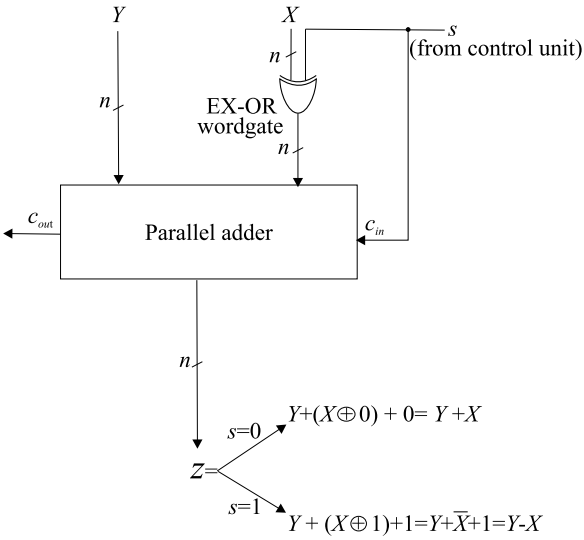


Fig. 2.14 Block diagram of a binary adder/subtracter

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

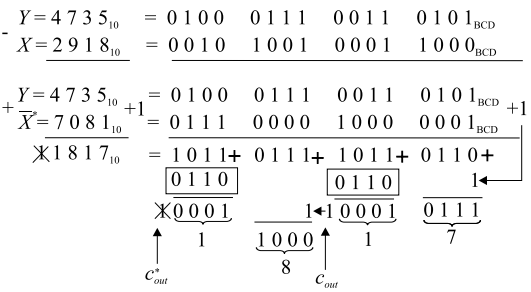


Fig. 2.17 Truth table for synthesis of a code translator for the nine's complement implementation

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 $\overline{X_i^*}$, $\overline{x_{i,3}^*} = \overline{x_{i,3}}\,\overline{x_{i,2}}\,\overline{x_{i,1}} = \overline{x_{i,3}}\ \underline{or}\ x_{i,2}\ \underline{or}\ x_{i,1}$, $\overline{x_{i,2}^*} = \overline{x_{i,3}}(x_{i,2} \oplus x_{i,1})$, $\overline{x_{i,1}^*} = \overline{x_{i,3}}x_{i,1}$ 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:

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

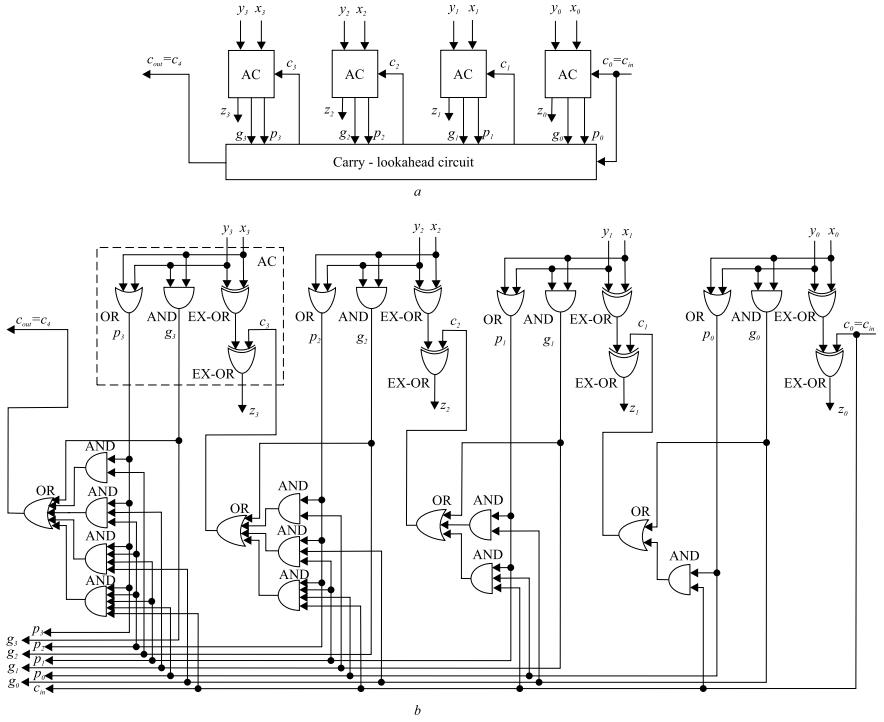


Fig. 2.18 Block diagram of a 4-bit conventional CLA and one of its gate level implementation versions

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. $c_1 = g_0 \underline{or}\ p_0c_0$, 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 $G_{0,1} = g_1 \underline{or}\ p_1g_0$ signifies the fact that the carry is generated in the block made up of the first two ranks, 0 and 1, and $P_{0,1} = p_1p_0$ 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

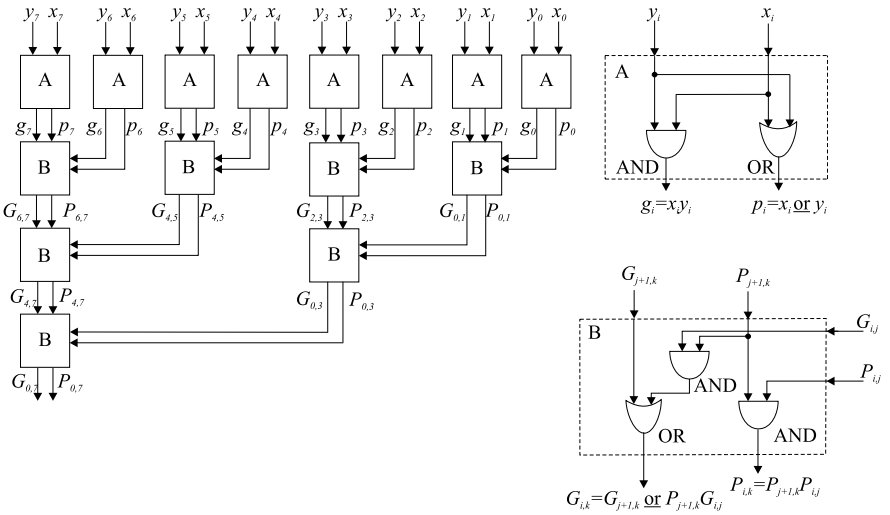


Fig. 2.19 Binary tree structure for building the generation and propagation functions of an 8-bit multilevel CLA

$P_{i,i} = p_i$, 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.

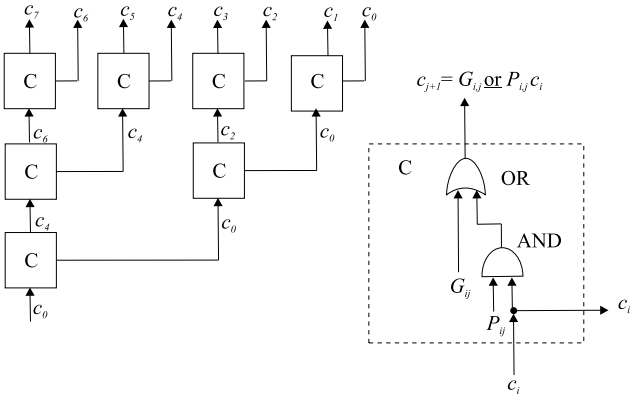


Fig. 2.20 Binary tree structure for building the carries of an 8-bit multilevel CLA

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,

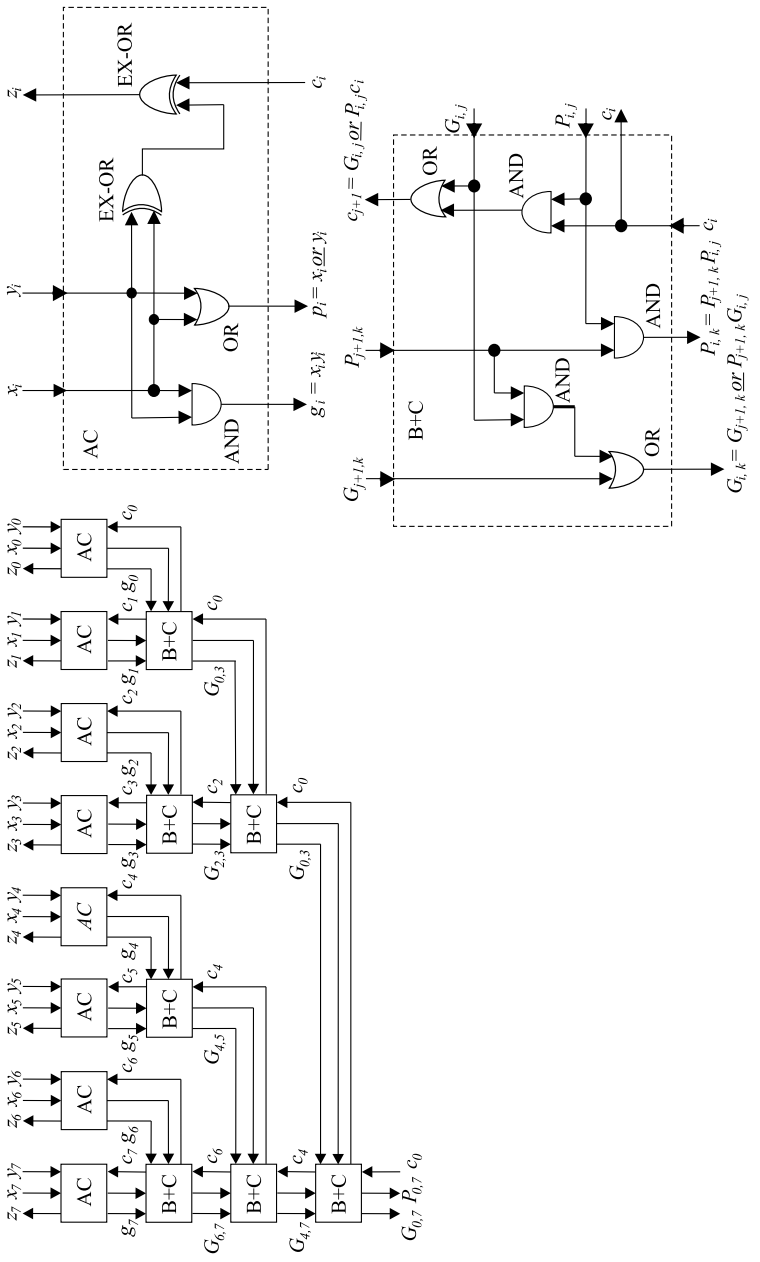


Fig. 2.21 Block diagram of an 8-bit multilevel CLA and gate level implementations of its cells

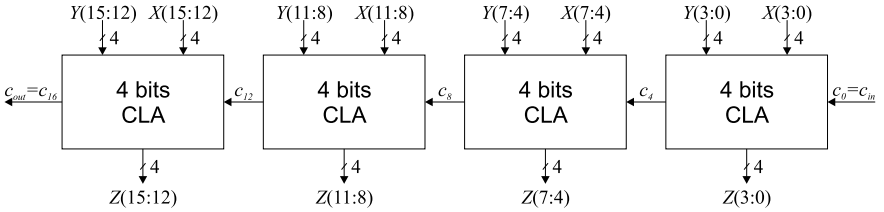


Fig. 2.22 Block diagram of a 16-bit hybrid adder consisting in connecting 4-bit CLA segments in a RCA manner

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

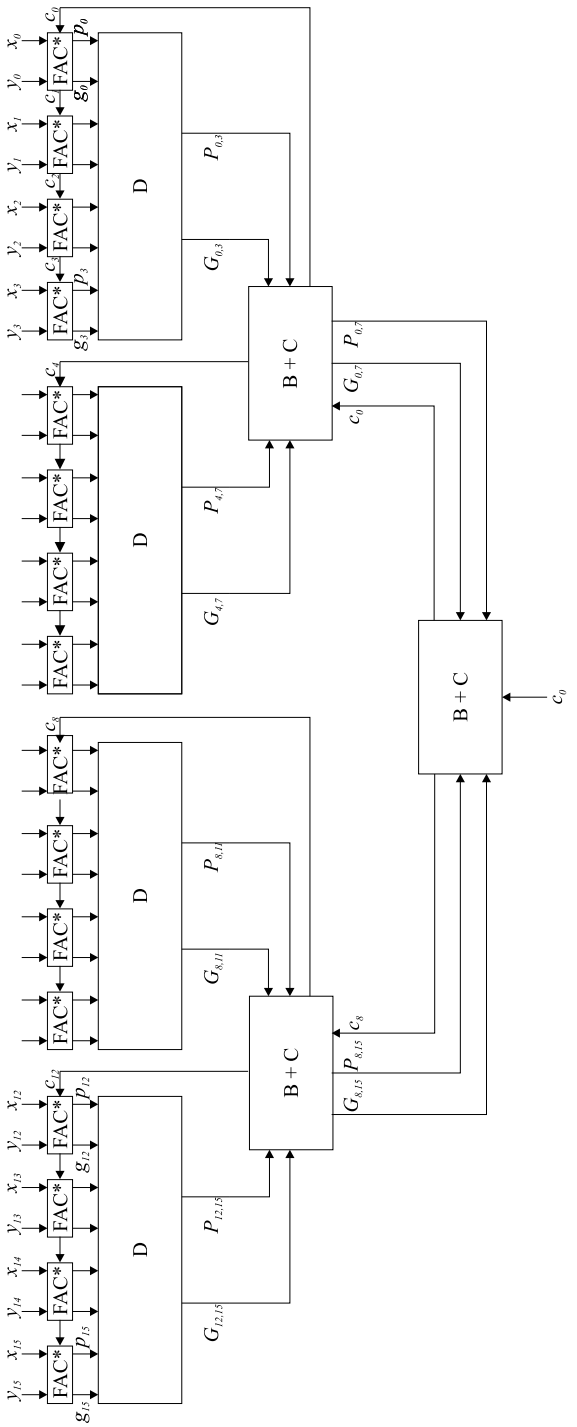


Fig. 2.23 Block diagram of a 16-bit hybrid adder consisting in connecting 4-bit RCA segments in a CLA manner

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

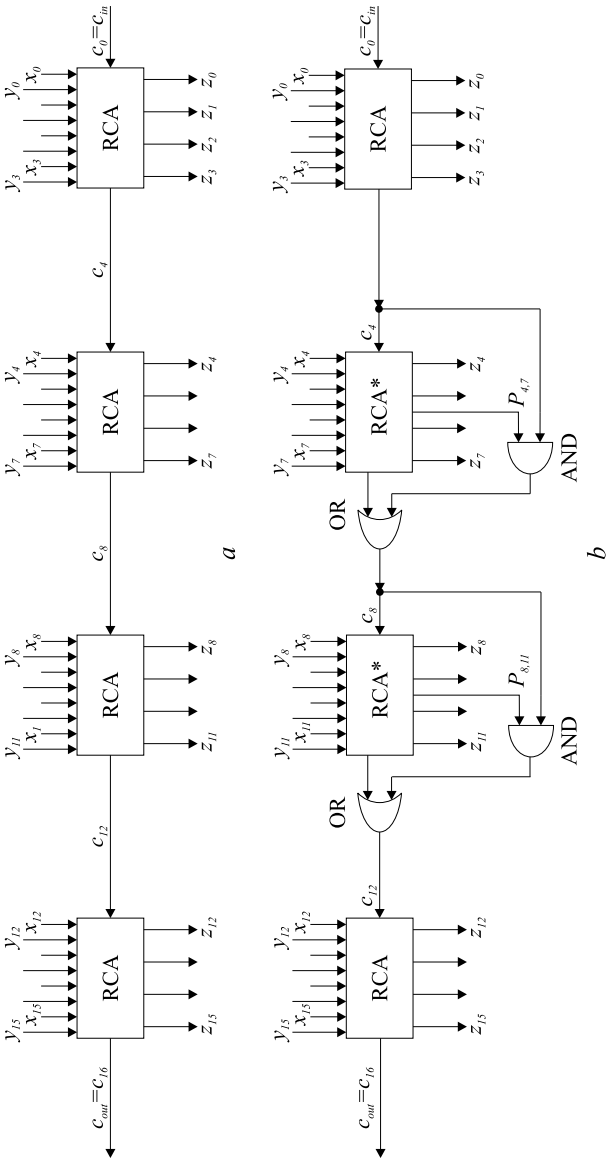


Fig. 2.24 Block diagram of a 16-bit CSkA

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

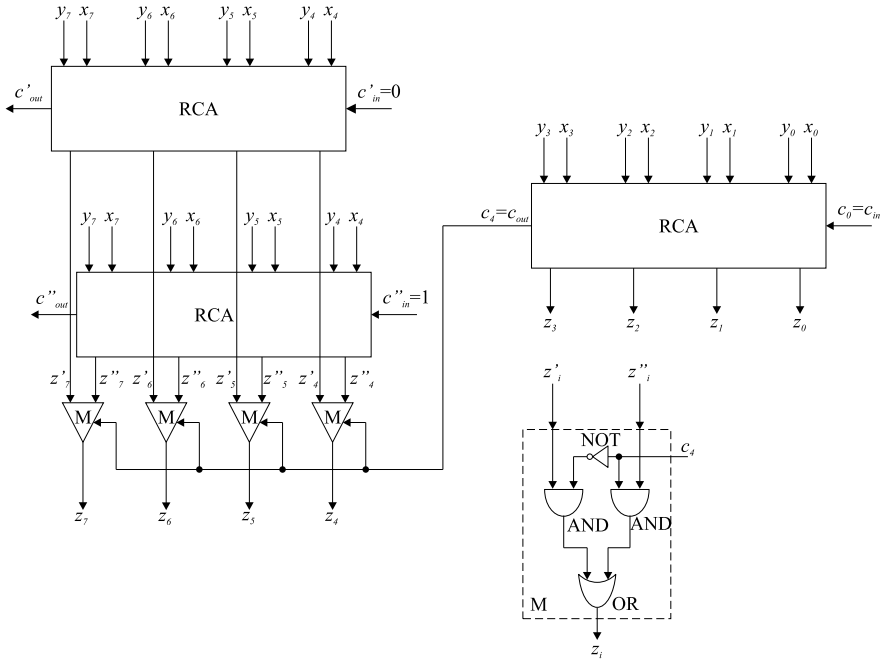


Fig. 2.25 Block diagram of an 8-bit CSeA

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

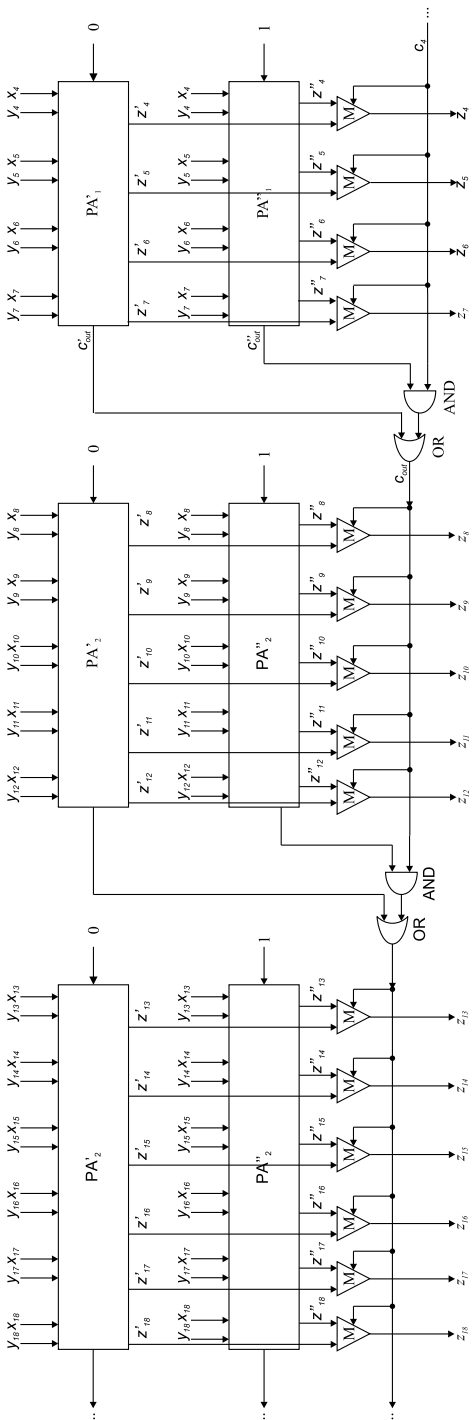


Fig. 2.26 Segments interconnection for a CSeA

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

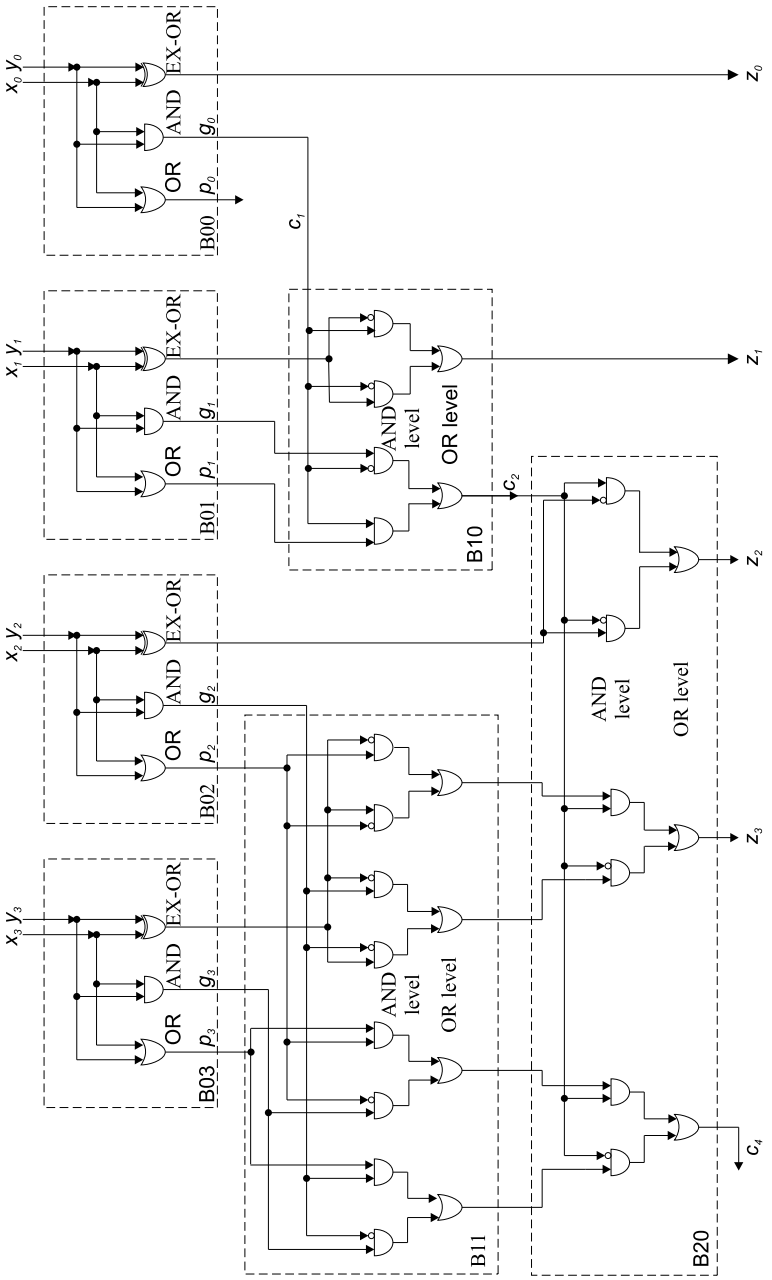


Fig. 2.28 Gate level synthesis for a 4-bit CSuA

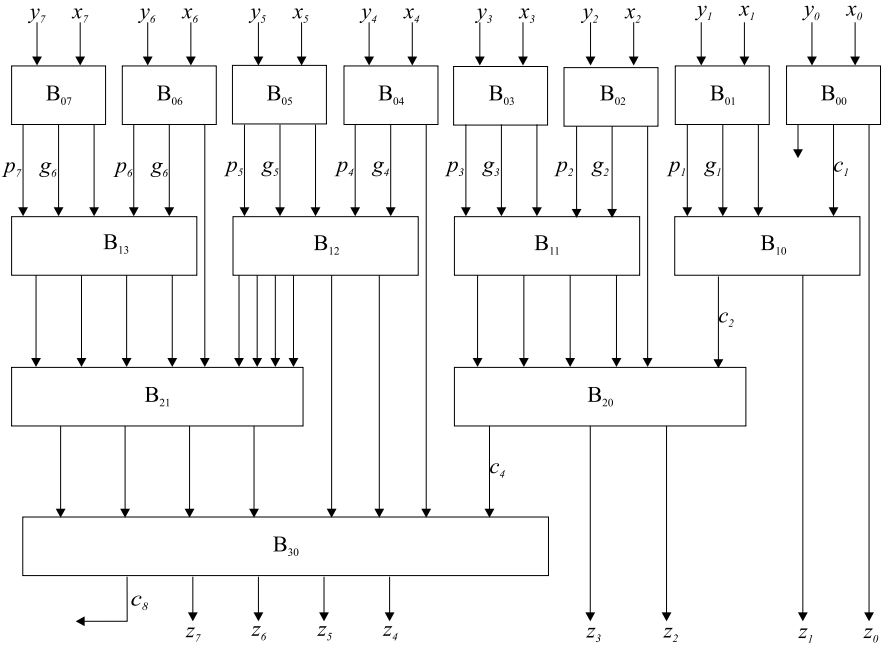


Fig. 2.29 Block diagram of an 8-bit CSuA

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.

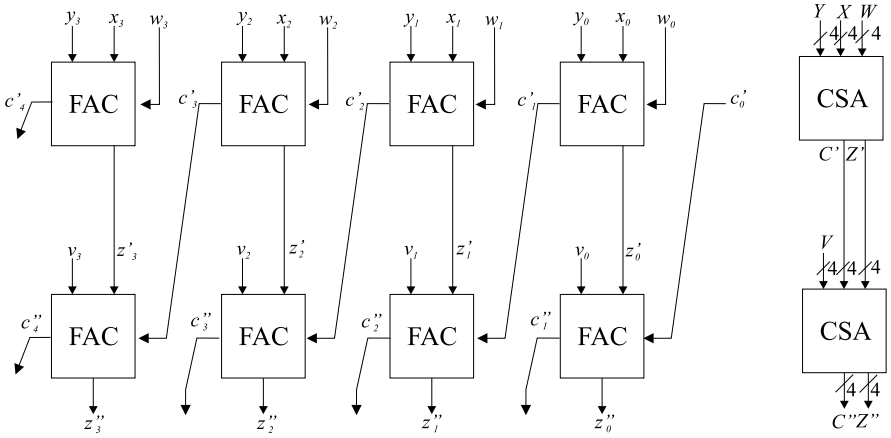


Fig. 2.31 FACs interconnections into a CSA structure

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

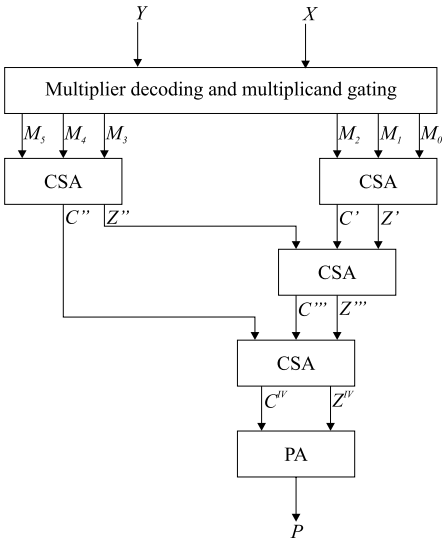


Fig. 2.32 Block diagram of a binary multiplier with CSA levels

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

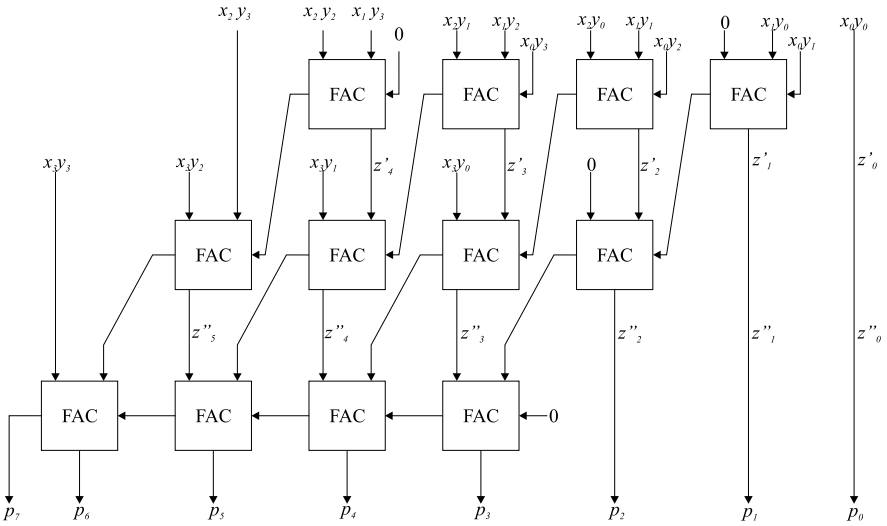


Fig. 2.33 Detailed FACs interconnections for a 4-bit operands binary multiplier having two CSA levels and an RCA level

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

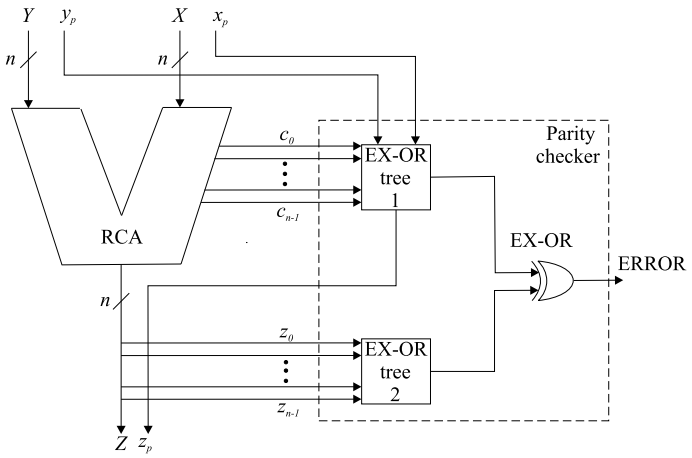


Fig. 2.35 Block diagram of a parity checked adder

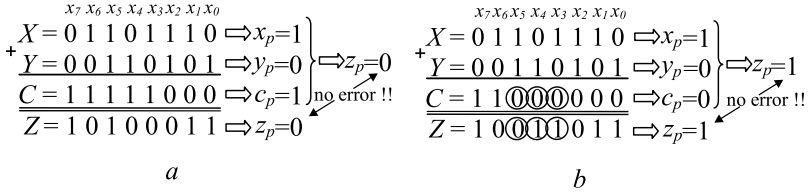


Fig. 2.36 RCA binary addition example with masked singular fault presence on the carry chain

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

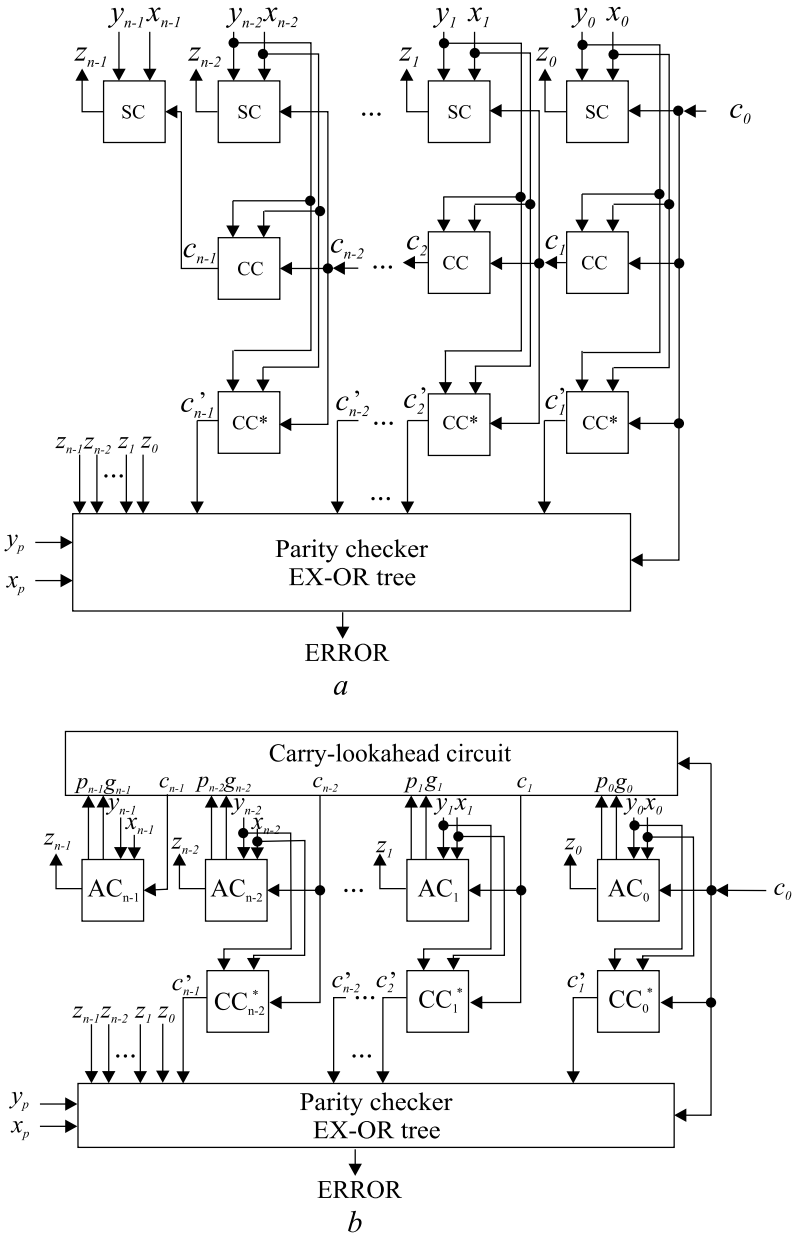


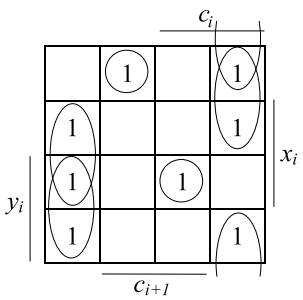
Fig. 2.37 Single fault detection solutions based on carry chain duplication

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

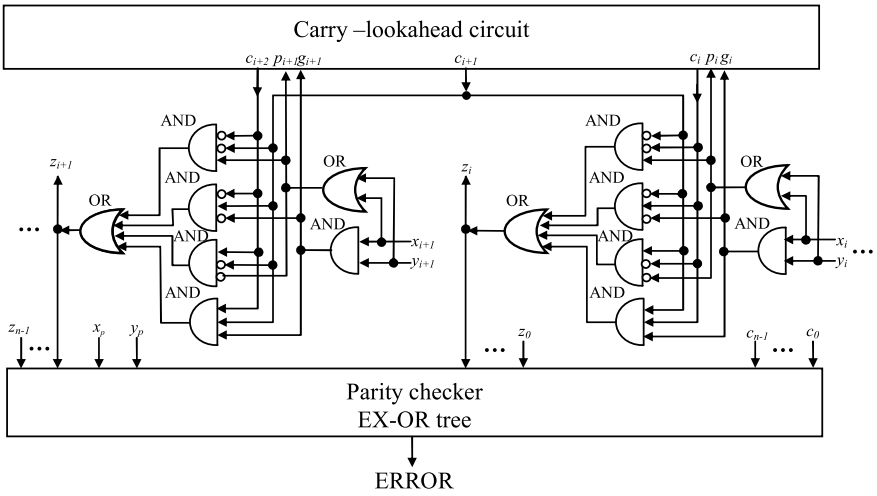


Fig. 2.40 Block diagram of a CDSA with gate level details for the ranks i and $i+1$

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.

http://www.springer.com/978-3-642-18314-0

Computer Arithmetic
Algorithms and Hardware Implementations
Vlăduţiu, M.
2012, VI, 270 p., Hardcover
ISBN: 978-3-642-18314-0