Chapter 7
Adders

Addition is a primitive operation for most arithmetic functions, so that FPGA vendors have dedicated a particular attention to the design of optimized adders. As a consequence, in many cases the synthesis tools are able to generate fast and cost-effective adders from simple VHDL expressions. Only in the case of relatively long operands can it be worthwhile to consider more complex structures such as carry-skip, carry-select and logarithmic adders.

Another important topic is the design of multi-operand adders. In this case, the key concept is that of carry-save adder or, more generally, of parallel counter.

Obviously, the general design methods presented in Chap. 3 (pipelining, digit-serial processing, self-timing) can be applied in order to optimize the proposed circuits. Numerous examples of practical FPGA implementations are reported in Sect. 7.9.

7.1 Addition of Natural Numbers

Consider two radix-B numbers

$$x = x_{n-1} \cdot B^{n-1} + x_{n-2} \cdot B^{n-2} + \cdots + x_1 \cdot B + x_0$$

and

$$y = y_{n-1} \cdot B^{n-1} + y_{n-2} \cdot B^{n-2} + \cdots + y_1 \cdot B + y_0,$$

where all digits x_i and y_i belong to $\{0, 1, \ldots, B-1\}$, and an input carry c_0 belonging to $\{0, 1\}$. An n-digit adder generates a radix-B number

$$z = z_{n-1} \cdot B^{n-1} + z_{n-2} \cdot B^{n-2} + \cdots + z_1 \cdot B + z_0,$$

J.-P. Deschamps et al., *Guide to FPGA Implementation of Arithmetic Functions*, Lecture Notes in Electrical Engineering 149, DOI: 10.1007/978-94-007-2987-2_7,
© Springer Science+Business Media Dordrecht 2012

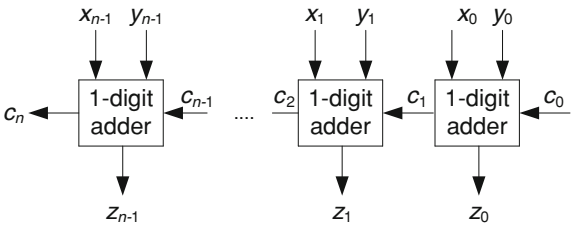

Fig. 7.1 *n*-digit adder

and an output carry c_n, such that

$$x + y + c_0 = c_n \cdot B^n + z.$$

Observe that $x + y + c_0 \leq 2(B^n - 1) + 1 = 2B^n - 1$, so that c_n belongs to $\{0, 1\}$.

The common way to implement an n-digit adder consists of connecting in series n 1-digit adders (Fig. 7.1). For each of them

$$x_i + y_i + c_i = c_{i+1} \cdot B + z_i,$$

where c_i and c_{i+1} belong to $\{0, 1\}$. In other words

$$z_i = (x_i + y_i + c_i) \bmod B,\ c_{i+1} = \lfloor (x_i + y_i + c_i)/B \rfloor.$$

The critical path is

$$(x_0, y_0, c_0) \rightarrow c_1 \rightarrow c_2 \rightarrow \cdots \rightarrow c_{n-1} \rightarrow (z_{n-1}, c_n),$$

so that the total computation time is approximately equal to $n \cdot T_{carry}$ where T_{carry} is the computation time of c_{i+1} in function of x_i, y_i and c_i.

In order to reduce T_{carry}, it is convenient to compute two binary functions p (*propagate*) and g (*generate*) of x_i and y_i:

$$p(x_i, y_i) = 1 \text{ if } x_i + y_i = B - 1, p(x_i, y_i) = 0 \text{ otherwise;}$$

$g(x_i, y_i) = 1$ if $x_i + y_i \geq B, g(x_i, y_i) = 0$ if $x_i + y_i \leq B - 2$, otherwise, any value. So, c_{i+1} can be expressed under the following way:

$$c_{i+1} = p(x_i,\ y_i) \cdot c_i + \overline{p(x_i,\ y_i)} \cdot g(x_i,\ y_i)$$

The last relation expresses that if $x_i + y_i = B - 1$, then c_{i+1} is equal to c_i; if $x_i + y_i \geq B$, then $c_{i+1} = 1$; if $x_i + y_i \leq B - 2$, then $c_{i+1} = 0$. The corresponding implementation is shown in Fig. 7.2. It is made up of two 2-operand combinational circuits that compute $p(x_i, y_i)$ and $g(x_i, y_i)$, and a multiplexer. In an n-digit adder (Fig. 7.1), all functions $p(x_i, y_i)$ and $g(x_i, y_i)$ are computed in parallel, so that the value of T_{carry} is practically equal to the multiplexer delay T_{mux}.

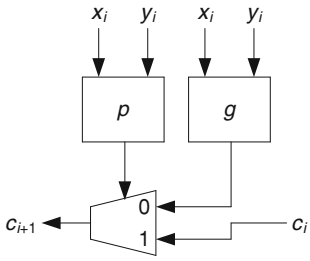

Fig. 7.2 Carry computation

7.2 Binary Adder

If $B = 2$, then $p(x_i, y_i) = x_i$ XOR y_i, and $g(x_i, y_i)$ can be chosen equal to x_i (or y_i). A complete n-bit adder is shown in Fig. 7.3. Its computation time is equal to

$$T_{adder}(n) = T_{xor} + (n-1) \cdot T_{mux} + \max\{T_{mux},\ T_{xor}\}, \tag{7.1}$$

and the delay from the input carry to the output carry is equal to

$$T_{carry-to-carry}(n) = n \cdot T_{mux}. \tag{7.2}$$

Comment 7.1
Most FPGA's include the basic components to implement the structure of Fig. 7.3, and the synthesis tools automatically generate this optimized adder from a simple VHDL expression such as

```
z <= x + y + c0;
```

7.3 Radix-2^k Adder

If $B = 2^k$, then $p(x_i, y_i) = 1$ if $x_i + y_i = 2^k - 1$, that is, if the k less significant bits of $s_i = x_i + y_i$ are equal to 1, and $g(x_i, y_i) = 1$ if $x_i + y_i \geq 2^k$, that is, if the most significant bit of s_i is equal to 1. The iterative cell of a radix-2^k adder is shown in Fig. 7.4. The critical path of the part of the circuit that computes $g(x_i, y_i)$ and $p(x_i, y_i)$ has been shaded. Its computation time is equal to $T_{adder}(k) + T_{and}$. An m-digit radix-2^k adder is equivalent to an n-bit adder with $n = m \cdot k$. The total computation time is

$$T_{adder}(n) = T_{adder}(k) + T_{and} + (m-1) \cdot T_{mux} + T_{half-adder}(k), \tag{7.3}$$

and the delay from the input carry to the output carry to

$$T_{carry-to-carry}(n) = m \cdot T_{mux}. \tag{7.4}$$

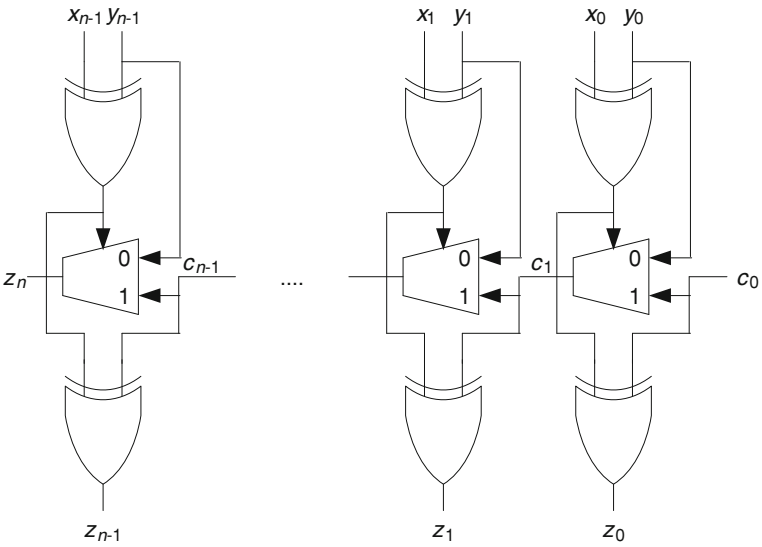

Fig. 7.3 *n*-bit adder

The following VHDL model describes the basic cell of Fig. 7.4.

```
s <= '0'&x + y;
t(1) <= s(0);
and_gates: FOR i in 1 TO k-1 GENERATE
  t(i+1) <= t(i) AND s(i);
END GENERATE;

p <= t(k);
WITH p SELECT c_out <= c_in WHEN '1', s(k) WHEN OTHERS;
z <= s(k-1 DOWNTO 0) + c_in;
```

An alternative way of computing $p = s_0 \cdot s_1 \cdot \ldots \cdot s_{k-1}$ is

```
t <= '0'&s(k-1 DOWNTO 0) + '1';
p <= t(k);
```

The corresponding circuit is a k-bit half adder that computes $t = (s \bmod 2^k) + 1$. The most significant bit t_k of t is equal to 1 if, and only if, all the bits of $(s \bmod 2^k)$ are equal to 1. As mentioned above (Comment 7.1) most FPGA's include the basic components to implement the structure of Fig. 7.3. In the particular case where $x = 0$, $y = s$ and $c_0 = 1$, the circuit of Fig. 7.4 is obtained. The apparently unnecessary XOR gates are included because there is generally no direct connection between the adder inputs and the multiplexer control inputs. Actually, the

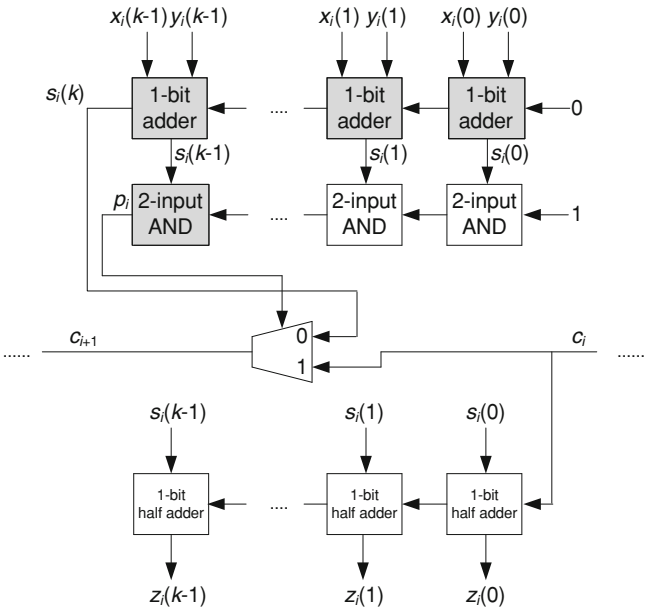

Fig. 7.4 Radix 2^k adder

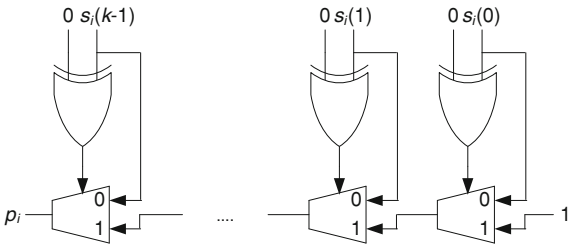

Fig. 7.5 FPGA implementation of a k-input AND

XOR gates are LUTs whose outputs are permanently connected to the carry-logic multiplexers.

A complete generic model *base_2 k_adder.vhd* is available at the Authors' web page and examples of FPGA implementations are given in Sect. 7.9.

According to (7.3), the non-constant terms of $T_{adder}(n)$ are:

- $m{\cdot}T_{mux}$,
- $k{\cdot}T_{mux}$ included in $T_{adder}(k)$ according to (7.1),
- $k{\cdot}T_{mux}$ included in $T_{half\text{-}adder}(k)$ according to (7.1).

Thus, the sum of the non-constant terms of $T_{adder}(n)$ is equal to $(2k + m){\cdot}T_{mux}$. The value of $2k + m$, with $m{\cdot}k = n$, is minimum when $2k \cong m$, that is, when $k \cong (n/2)^{1/2}$. With this value of k, the sum of the non-constant terms of $T_{adder}(n)$ is equal to $(8n)^{1/2}{\cdot}T_{mux}$. Thus, the computation time is $O(n)^{1/2}$ instead of $O(n)$.

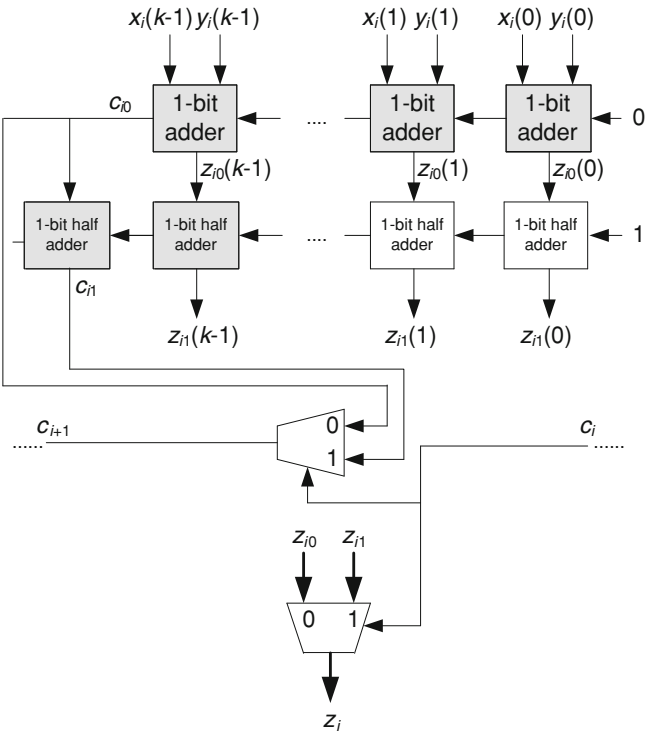

Fig. 7.6 Carry select adder

Comments 7.2

1. The circuit of Fig. 7.4 is an example of *carry-skip adder*. For every group of k bits, both the carry-propagate and carry-generate functions are computed. If the carry-propagate function is equal to 1, the input carry is directly propagated to the carry output of the k-bit group, thus skipping k bits.
2. A mixed-radix numeration system could be used. Assume that $n = k_1 + k_2 + \ldots + k_m$; then a radix

$$(2^{k_1},\ 2^{k_2}, \cdots, 2^{k_m})$$

representation can be considered. The corresponding adder consists of m blocks, similar to that of Fig. 7.3, whose sizes are k_1, k_2,…, and k_m, respectively. Nevertheless, within an FPGA it is generally better to use adders that fit within a single column. Assuming that the chosen device has r carry-logic cells per column, a good option could be a fixed-radix adder with $k \leq r$. In order to minimize the computation time, k must be approximately equal to $(n/2)^{1/2}$, so that n must be smaller than $2r^2$, which is a very large number.

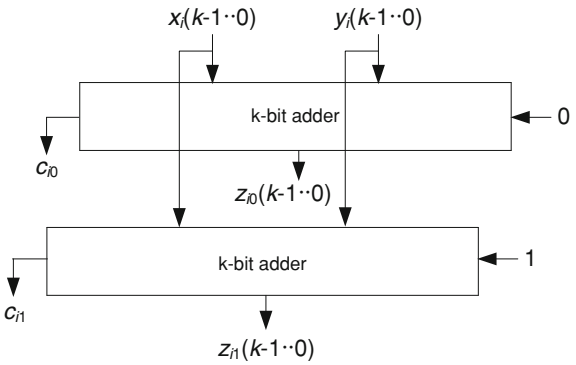

Fig. 7.7 Carry-select adder (second version)

7.4 Carry Select Adders

Another way of reducing the computation time of a radix-2^k adder consists in computing, at each step, the next carry and the output digit for both values of the input carry. The corresponding circuit is shown in Fig. 7.6.

The critical path of the part of the circuits that computes the two possible values of the next carry and of the output digit has been shaded. Its computation time is equal to $T_{adder}(k) + T_{adder}(2)$. The total computation time is ($n = m{\cdot}k$)

$$T_{adder}(n) = T_{adder}(k) + T_{half\_adder}(2) + (m - 1) \cdot T_{mux} + T_{mux}, \tag{7.5}$$

and the delay from the input carry to the output carry to

$$T_{carry-to-carry}(m \cdot k) = m \cdot T_{mux}. \tag{7.6}$$

The following VHDL model describes the basic cell of Fig. 7.6.

```
t0 <= '0'&x + y;
t1 <= '0'&x + y + '1';
c0 <= t0(k);
c1 <= t1(k);
z0 <= t0(k-1 DOWNTO 0);
z1 <= t1(k-1 DOWNTO 0);
WITH c_in SELECT c_out <= c0 WHEN '0', c1 WHEN OTHERS;
WITH c_in SELECT z <= z0 WHEN '0', z1 WHEN OTHERS;
```

A complete generic model *carry_select_adder.vhd* is available at the Authors' web page and examples of FPGA implementations are given in Sect. 7.9.

The non-constant term of $T_{adder}(n)$ is equal to $(k + m){\cdot}T_{mux}$. The minimum value is obtained when $k \cong m$, that is $k \cong (n)^{1/2}$. With this value of k, the non-constant term of $T_{adder}(n)$ is equal to $(4n)^{1/2}{\cdot}T_{mux}$. Thus, the computation time is $O(n)^{1/2}$ instead of $O(n)$.

Comment 7.3
As before (Comments 7.2) a mixed-radix numeration system could be considered.

As a matter of fact, the FPGA implementation of a half-adder is generally not more cost-effective than the implementation of a full adder. So, the circuit of Fig. 7.6 could be slightly modified: instead of computing c_{i0} and c_{i1} with a full adder and a half adder, two independent full adders of any type can be used (Fig. 7.7).

The following VHDL model describes the modified cell:

```
t0 <= '0'&x + y;
t1 <= t0 + '1';
c0 <= t0(k);
c1 <= t1(k);
z0 <= t0(k-1 DOWNTO 0);
z1 <= t1(k-1 DOWNTO 0);
WITH c_in SELECT c_out <= c0 WHEN '0', c1 WHEN OTHERS;
WITH c_in SELECT z <= z0 WHEN '0', z1 WHEN OTHERS;
```

The computation time of the modified circuit is

$$T_{adder}(n) = T_{adder}(k) + (m - 1) \cdot T_{mux} + T_{mux} = T_{adder}(k) + m \cdot T_{mux}. \qquad (7.7)$$

A complete generic model *carry_select_adder2.vhd* is available at the Authors' web page and examples of FPGA implementations are given in Sect. 7.9.

7.5 Logarithmic Adders

Several types of adders whose computation time are proportional to the logarithm of n have been proposed. For example: carry-lookahead adders ([1], Chap. 6), Ling adders [2], Brent-Kung prefixed adders [3], Ladner-Fischer prefixed adders [4]. Nevertheless, their FPGA implementations are generally not as fast as what could be theoretically expected. There are two reasons for that. On the one hand, the special purpose carry-logic included in most FPGAs is very fast, so that ripple-carry adders are fast. Their computation time is approximately equal to $a + b{\cdot}n$, where a and b are very small constants: a is the delay of a LUT and b is the delay of a multiplexer belonging to the carry logic. On the other hand, the structure of most logarithmic adders is not so regular as the structure of ripple-carry adders, so that they include long connections which in turn introduce long additional delays. The practical result is that, except for very great values of n, the adders described in Sects. 7.2–7.4 are faster and more cost-effective.

Obviously, any optimization method that allows the dividing up of an n-bit adder into smaller k-bit and m-bit adders, with $k{\cdot}m = n$, in such a way that

$$T_{adder}(n) \cong T_{adder}(k) + T_{adder}(m),$$

can be recursively used in order to generate a logarithmic adder. As an example, consider again a carry-select adder. According to (7.7)

$$T_{adder}(n) = T_{adder}(k) + m.T_{mux}.$$

Assume that $k = r{\cdot}s$. Then each k-bit adder (Fig. 7.7) can in turn be decomposed in such a way that

$$T_{adder}(k) = T_{adder}(r) + s.T_{mux},$$

so that the computation time of the corresponding 2-level carry-select adder is

$$T_{adder}(n) = T_{adder}(r) + (s + m) \cdot T_{mux},$$

where $n = r{\cdot}s{\cdot}m$. Finally, if $n = n_1{\cdot}n_2{\cdot}\ldots\ {\cdot}n_t$, then a ($t$-1)-level carry-select adder, whose computation time is equal to

$$T_{adder}(n_1 \cdot n_2 \cdot \ldots \cdot n_t) = T_{adder}(n_1) + (n_2 + \ldots + n_t) \cdot T_{mux} = O(n_1 + n_2 + \ldots + n_t),$$

can be generated.

Example 7.1
The following VHDL model describes an n-bit 2-level carry-select adder with $n = n_1{\cdot}n_2{\cdot}n_3$. First, define the basic cell *carry_select_step3*, in which two 1-level carry-select adders, with $k = n_1$ and $m = n_2$, are used:

```
first_adder: carry_select_adder2
GENERIC MAP(k => n1, m => n2)
PORT MAP(x => x, y => y, c_in => '0', z => z0, c_out => c0);
second_adder: carry_select_adder2
GENERIC MAP(k => n1, m => n2)
PORT MAP(x => x, y => y, c_in => '1', z => z1, c_out => c1);
WITH c_in SELECT c_out <= c0 WHEN '0', c1 WHEN OTHERS;
WITH c_in SELECT z <= z0 WHEN '0', z1 WHEN OTHERS;
```

The complete circuit is made up of n_3 basic cells:

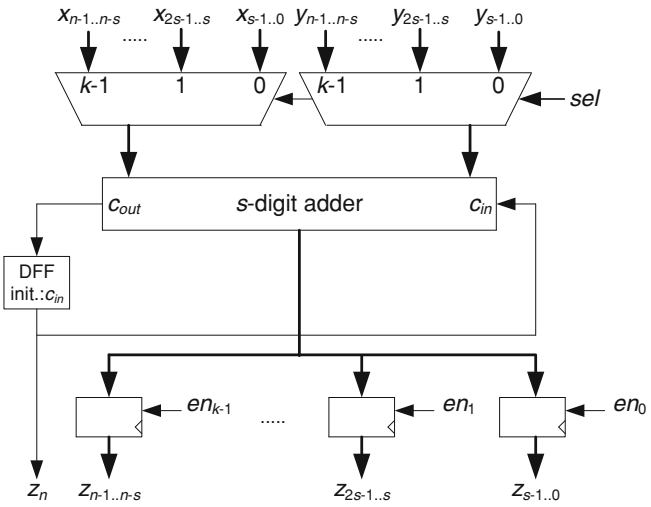

Fig. 7.8 Long-operand adder

```
carries(0) <= c_in;
iteration: FOR i IN 0 TO n3-1 GENERATE
  main_component:carry_select_step3
  GENERIC MAP(n1 => n1, n2 => n2)
  PORT MAP(x => x(n1*n2*i+n1*n2-1 DOWNTO n1*n2*i),
  y => y(n1*n2*i+n1*n2-1 DOWNTO n1*n2*i), c_in => carries(i),
  z => z(n1*n2*i+n1*n2-1 DOWNTO n1*n2*i),
  c_out => carries(i+1));
END GENERATE;
c_out <= carries(n3);
```

A complete generic model *carry_select_adder3.vhd* is available at the Authors' web page and examples of FPGA implementations are given in Sect. 7.9.

7.6 Long-Operand Adder

In the case of long-operand additions, the n-digit operands can be broken down into s-digit groups and the addition computed according to the following algorithm in which *natural_addition* is a procedure that computes

$$z_i = (x_i + y_i + c_i) \bmod B^s \text{ and } c_{i+1} = \lfloor (x_i + y_i + c_i) / B^s \rfloor,$$

where x_i, y_i and z_i are s-digit numbers, and c_i and c_{i+1} are bits.

Algorithm 7.1: Long-operand addition

```
c0 := cin;
for i in 0 .. n/s - 1 loop
  natural_addition(ci, x(i·s+s-1..i·s), y(i·s+s-1..i·s), ci+1,
    z(i·s+s-1..i·s));
end loop;
zn := cn/s;
```

The complete circuit (Fig. 7.8, with $k = n/s$) is made up of an s digit adder, connection resources (k-to-1 multiplexers) giving access to the s-digit groups, a D-flip-flop which stores the carries (c_i in Algorithm 7.1), an output register storing z, and a control unit whose main component is a k-state counter.

The following VHDL model describes the circuit of Fig. 7.8 ($B = 2$).

```
adder_in1 <= x(sel*s+s-1 DOWNTO sel*s);
adder_in2 <= y(sel*s+s-1 DOWNTO sel*s);
sum <= '0'&adder_in1 + adder_in2 + q;
adder_out <= sum(s-1 DOWNTO 0);
next_q <= sum(s);
registers: FOR i IN 0 TO k-1 GENERATE
  PROCESS(clk)
  BEGIN
    IF clk'EVENT AND clk = '1' THEN
      IF (update = '1') AND (sel = i) THEN
        z(i*s+s-1 DOWNTO i*s) <= adder_out;
      END IF;
    END IF;
  END PROCESS;
END GENERATE;
flip_flop: PROCESS(clk)
BEGIN
  IF clk'EVENT AND clk = '1' THEN
    IF load = '1' THEN q <= c_in;
    ELSIF update = '1' THEN q <= next_q; END IF;
  END IF;
END PROCESS;
```

A complete generic model *long_operand_adder.vhd* is available at the Authors' web page.

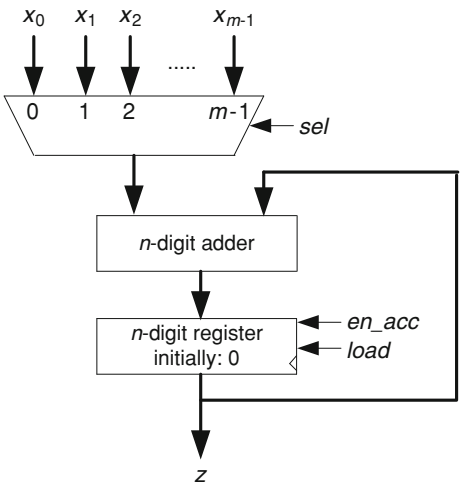

Fig. 7.9 Multioperand addition

7.7 Multioperand Adders

Consider m natural numbers x_0, x_1,…, x_{m-1}. A multioperand adder computes

$$z = x_0 + x_1 + \cdots + x_{m-1}. \tag{7.8}$$

7.7.1 Sequential Multioperand Adders

In order to compute (7.8), the following (obvious) algorithm can be used.

Algorithm 7.2: Basic multioperand addition

```
accumulator := 0;
for j in 0 .. m-1 loop
  accumulator := accumulator + xj;
end loop;
z := accumulator;
```

The corresponding sequential circuit (Fig. 7.9) is made up of an n-digit adder, an n-digit register, an m-to-1 n-digit multiplexer, and a control unit whose main component is an m-state counter.

The following VHDL model describes the circuit of Fig. 7.9 ($B = 2$). The $n{\cdot}m$-bit vector x is the concatenation of x_0, x_1,…, x_{m-1}.

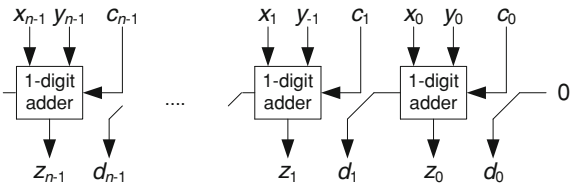

Fig. 7.10 Carry-save adder

```
adder_in1 <= x(sel*n+n-1 DOWNTO sel*n);
adder_out <= adder_in1 + accumulator;
z <= accumulator;
accumulator_register: PROCESS(clk)
BEGIN
  IF clk'EVENT AND clk = '1' THEN
    IF load = '1' THEN accumulator <= (OTHERS => '0');
    ELSIF en_acc = '1' THEN accumulator <= adder_out; END IF;
  END IF;
END PROCESS;
```

A complete generic model *multioperand_adder.vhd* is available at the Authors' web page.

The computation time of the preceding m-operand n-digit sequential adder is approximately equal to

$$T_{sequential}(m,n) \cong m \cdot T_{adder}(n). \tag{7.9}$$

In order to reduce the computation time, a *carry-save adder* can be used. The basic component is shown in Fig. 7.10: it consists of n 1-digit adders working in parallel. Given two n-digit numbers x and y, and an n-bit number c, it expresses the sum $(x + y + c)$ mod B^n under the form $z + d$, where z is an n-digit number and d an n-bit number. In other words, the carries are stored within the output binary vector d instead of being propagated (*stored-carry encoding*). As all cells work in parallel the computation time is independent of n.Let *CSA* be the function implemented by the circuit of Fig. 7.10, that is

$$CSA(x,\ y,\ c) = (z,\ d),$$

where

$$z_i = (x_i + y_i + c_i) \bmod B, d_i = \lfloor (x_i + y_i + c_i)\ /B \rfloor, \forall i \in \{0,\ 1, \ldots, n-1\}.$$

Assume that at every step of Algorithm 7.2 the value of *accumulator* is represented under the form $u + v$, where u is an n-digit number and v an n-bit number. Then, at step j, the following operation must be executed:

$$(u, v) := CSA\big(u, x_j, v\big).$$

The following formal algorithm computes z.

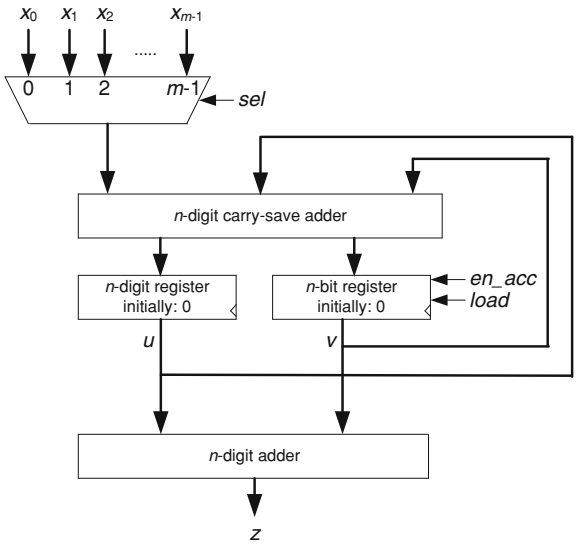

Fig. 7.11 Carry save adder

Algorithm 7.3: Multioperand addition with stored-carry encoding

```
u0 := 0; v0 := 0;
for j in 0 .. m-1 loop
  (uj+1, vj+1) := CSA(uj, xj, vj);
end loop;
z := um + vm;
```

The sequential circuit corresponding to Algorithm 7.3 (Fig. 7.11) is made up of an n-digit carry-save adder (Fig. 7.10), an n-digit register, an n-bit register, an m-to-1 n-digit multiplexer, a conventional n-digit adder implementing the last step of Algorithm 7.3, and a control unit whose main component is an m-state counter.The following VHDL model describes the circuit of Fig. 7.10 ($B = 2$). As before, x is the concatenation of x_0, x_1,…, x_{m-1}.

```
adder_in1 <= x(sel*n+n-1 DOWNTO sel*n);
next_v(0) <= '0';
csa: FOR i IN 0 TO n-2 GENERATE
  next_u(i) <= adder_in1(i) XOR u(i) XOR v(i);
  next_v(i+1) <=
    (adder_in1(i) AND u(i)) OR (adder_in1(i) AND v(i))
    OR (u(i) AND v(i));
END GENERATE;
```

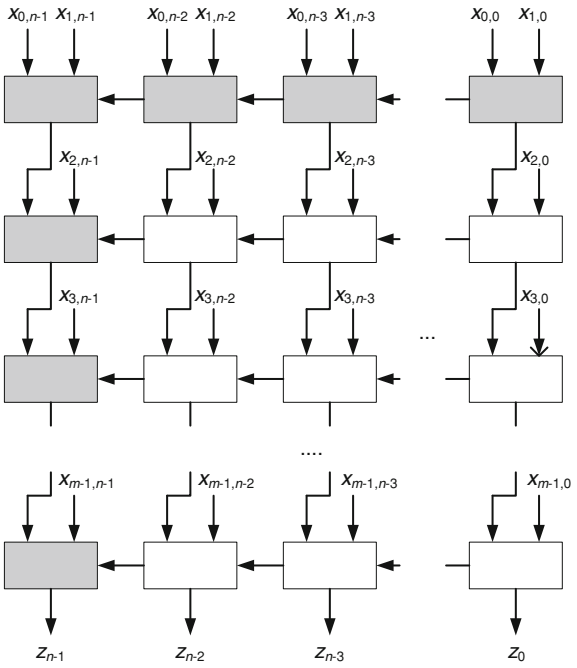

Fig. 7.12 Multioperand addition array

```
next_u(n-1) <= adder_in1(n-1) XOR u(n-1) XOR v(n-1);
z <= u + v;
accumulator_register: PROCESS(clk)
BEGIN
  IF clk'EVENT AND clk = '1' THEN
    IF load = '1' THEN
      u <= (OTHERS => '0'); v <= (OTHERS => '0');
    ELSIF en_acc = '1' THEN u <= next_u; v <= next_v; END IF;
  END IF;
END PROCESS;
```

A complete generic model *CSA_multioperand_adder.vhd* is available at the Authors' web page.

Taking into account that the computation time of the circuit of Fig. 7.10 is independent of the number n of digits, the computation time of the circuit of Fig. 7.10 is approximately equal to

$$T_{sequential\_csa}(m,n) \cong m \cdot T_{adder}(1) + T_{adder}(n). \tag{7.10}$$

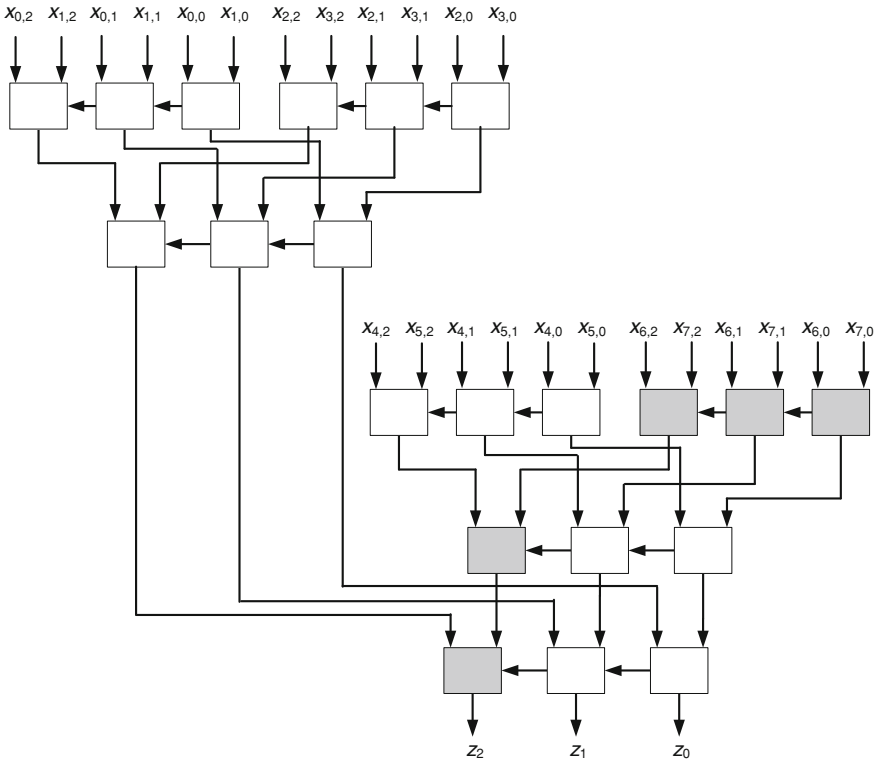

Fig. 7.13 Multioperand addition tree

7.7.2 Combinational Multioperand Adders

The combinational circuit that corresponds to Algorithm 7.2 is an iterative circuit made up of m-1 2-operand n-digit adders. If every adder is a simple ripple-carry adder, then the complete circuit is a 2-dimensional array made up of $(m\text{-}1)\cdot n$ one-digit adders, as shown in Fig. 7.12 in which one of the critical paths has been shaded. The corresponding computation time is equal to

$$T_{combinational}(m,\ n) = (m + n - 2) \cdot T_{adder}(1). \tag{7.11}$$

The following VHDL model describes the circuit of Fig. 7.12 ($B = 2$). As before, x is the concatenation of x_0, x_1,…, $x_{m\text{-}1}$.

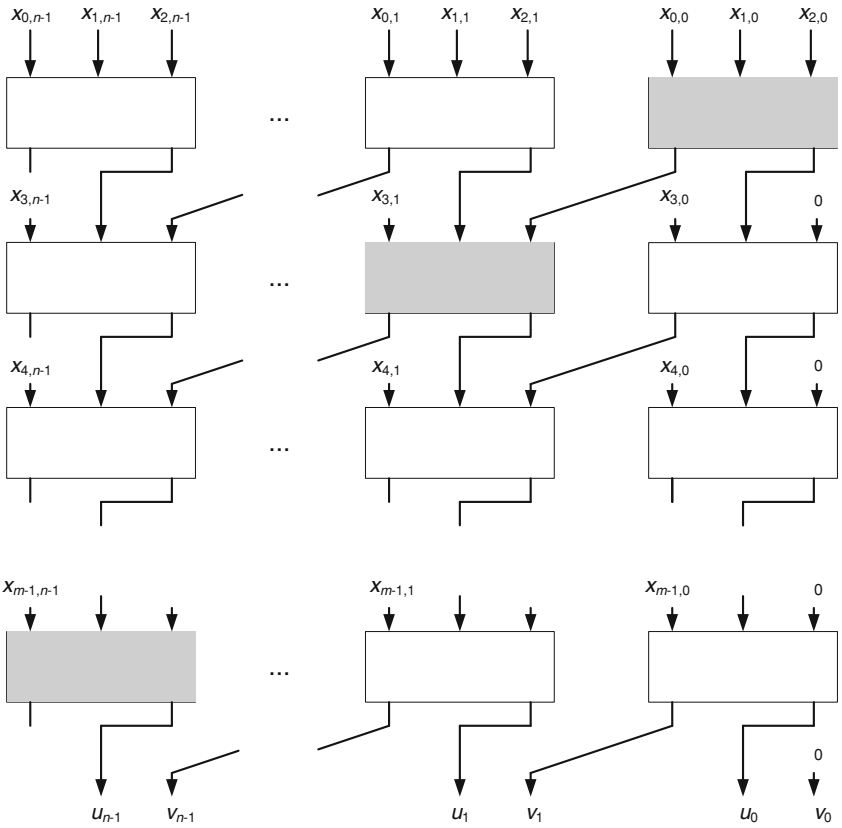

Fig. 7.14 Combinational carry-save adder

```
y(2*n-1 DOWNTO n) <= x(2*n-1 DOWNTO n) + x(n-1 DOWNTO 0);
iteration: FOR i in 2 TO m-1 GENERATE
  y(i*n+n-1 DOWNTO i*n) <=
    y(i*n-1 DOWNTO i*n-n) + x(i*n+n-1 DOWNTO i*n);
END GENERATE;
z <= y(m*n-1 DOWNTO m*n-n);
```

A complete generic model *comb_multioperand_adder.vhd* is available at the Authors' web page.

A most time-effective solution is a binary tree of 2-operand n-digit adders instead of an iterative circuit. An example, with $n = 3$ and $m = 8$, is shown in Fig. 7.13:

$$x_0 = x_{0,2}x_{0,1}x_{0,0}, x_1 = x_{1,2}x_{1,1}x_{1,0}, \ldots, x_7 = x_{7,2}x_{7,1}x_{7,0}.$$

The depth of the tree is equal to $\lceil log_2 m \rceil$ and its computation time (one of the critical paths has been shaded) is approximately equal to

$$T_{adder-tree}(m,n) \cong (n + \log_2 m - 1) \cdot T_{adder}(1). \tag{7.12}$$

The following VHDL model describes the circuit of Fig. 7.13 ($B = 2$).

```
y0 <= x0 + x1; y1 <= x2 + x3; y2 <= x4 + x5; y3 <= x6 + x7;
y4 <= y0 + y1; y5 <= y2 + y3;
z <= y4 + y5;
```

A complete generic model *eight_operand_adder.vhd* is available at the Authors' web page.

Another way to reduce the computation time, with an iterative architecture similar to that of Fig. 7.12, is to use the *carry-save* principle. An m-operand carry-save array (Algorithm 7.3) is shown in Fig. 7.14 (if $B > 2$, x_2 must be an n-bit number or an initial file that computes $x_0 + x_1 + 0$ must be added). The result is the sum of two n-digit numbers u and v. In order to get the actual result, an additional 2-operand n-digit adder is necessary for computing $u + v$ (last instruction of Algorithm 7.3). The corresponding computation time is equal to

$$T_{combinational\_csa}(m,n) = (m-2) \cdot T_{adder}(1) + T_{adder}(n). \tag{7.13}$$

The following VHDL model describes a 2-operand carry-save adder, also called 3-to-2 *counter* (Sect. 7.7.3). It corresponds to a file of the circuit of Fig. 7.14.

```
v(0) <= '0';
iteration: FOR i IN 0 TO n-2 GENERATE
  u(i) <= a(i) XOR b(i) XOR c(i);
  v(i+1) <=
    (a(i) AND b(i)) OR (a(i) AND c(i)) OR (b(i) AND c(i));
END GENERATE;
u(n-1) <= a(n-1) XOR b(n-1) XOR c(n-1);
```

The complete circuit is made up of m-2 3-to-2 counters:

```
first_row: three_to_two_counter GENERIC MAP(n => n)
PORT MAP(a => x(n-1 DOWNTO 0), b => x(n+n-1 DOWNTO n),
  c => x(2*n+n-1 DOWNTO 2*n), u => u(n-1 DOWNTO 0),
  v => v(n-1 DOWNTO 0));
iteration: FOR i IN 1 TO m-3 GENERATE
  following_rows: three_to_two_counter GENERIC MAP(n => n)
  PORT MAP(a => x((i+2)*n+n-1 DOWNTO (i+2)*n),
    b => u((i-1)*n+n-1 DOWNTO (i-1)*n),
    c => v((i-1)*n+n-1 DOWNTO (i-1)*n),
    u => u(i*n+n-1 DOWNTO i*n), v => v(i*n+n-1 DOWNTO i*n));
END GENERATE;
z <= u((m-3)*n+n-1 DOWNTO (m-3)*n) +
  v((m-3)*n+n-1 DOWNTO (m-3)*n);
```

A complete generic model *comb_CSA_mutioperand_adder.vhd* is available at the Authors' web page and examples of FPGA implementations are given in Sect. 7.9.

Comment 7.4
In all of the previously described multioperand adders, the operands, as well as the result, were assumed to be n-digit numbers. If all of the operands belong to the same range, and the result is known to be an n-digit number, whatever the value of the operands, then the operands can be represented with $(n–k)$ digits where $k \cong log_B m$, and the previously described circuits can be pruned.

7.7.3 Parallel Counters

Given two n-digit numbers x and y, and an n-bit number c, the carry-save adder of Fig. 7.10 allows the expression of the sum $(x + y + c)$ mod B^n under the form $z + d$, where z is an n-digit numbers and d an n-bit number. In other words, it reduces the sum of three digits x, y and c to the sum of two digits z and d. For that reason, it is also called a 3-to-2 *counter*.

This 3-to-2 counter can be used as a computation resource for reducing the sum of m digits $x_0, x_1,\ldots, x_{m-1}$ to the sum of two digits u and v as shown in Fig. 7.14. Thus, the circuit of Fig. 7.14 could be considered as an m-to-2 counter.

This type of construction can be generalized. As an example, consider an adder that computes the sum of 6 bits $x_0, x_1,\ldots, x_5$. The result, smaller than or equal to 6, is a 3-bit number. Thus, this 6-operand 1-bit adder computes

$$x_0 + x_1 + \cdots + x_5 = 4z_2 + 2z_1 + z_0 \tag{7.14}$$

and can be implemented by three 6-input Look Up Tables (LUT6) working in parallel:

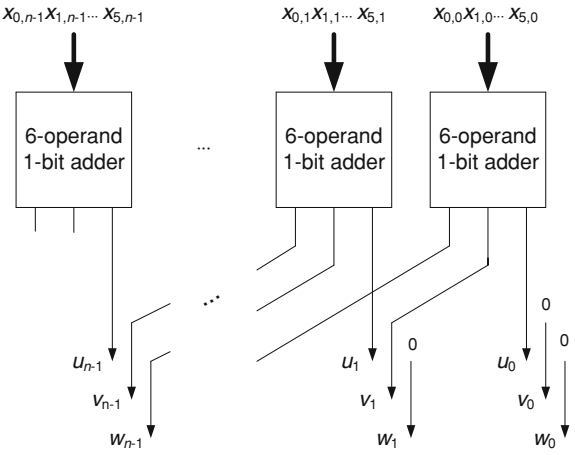

Fig. 7.15 6-to-3 counter

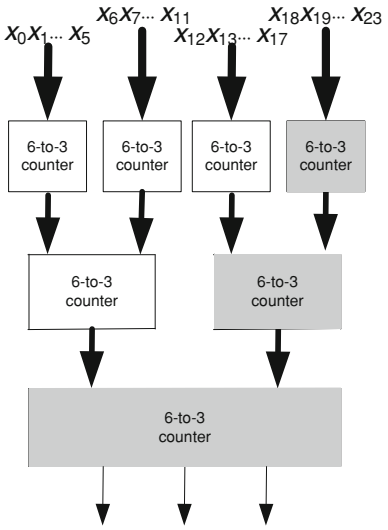

Fig. 7.16 24-to-3 counter

```
first_lut: lut6
GENERIC MAP(truth_vector => x"000101170117177f")
PORT MAP(a => x, b => z(2));
second_lut: lut6
GENERIC MAP(truth_vector => x"177e7ee87ee8e881")
PORT MAP(a => x, b => z(1));
third_lut: lut6
GENERIC MAP(truth_vector => x"6996966996696996")
PORT MAP(a => x, b => z(0));
```

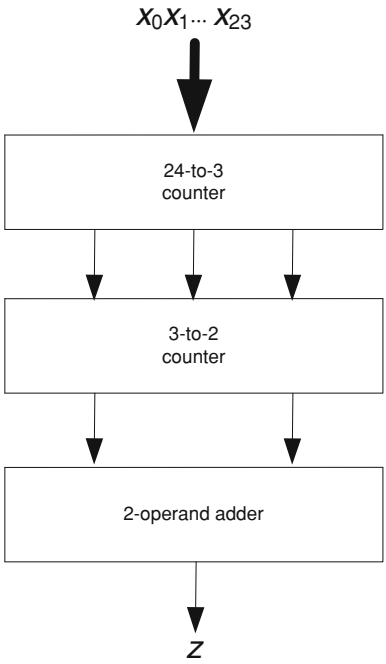

Fig. 7.17 24-operand adder

Then, by connecting in parallel *n* circuits of this type, a binary 6-to-3 counter is obtained (Fig. 7.15):

```
iteration1: FOR i IN 0 TO n-1 GENERATE
  x(i) <= x0(i)&x1(i)&x2(i)&x3(i)&x4(i)&x5(i);
END GENERATE;
iteration2: FOR i IN 0 TO n-1 GENERATE
  cells: six_to_three_counter_cell
  PORT MAP(x => x(i), z => z(i));
  u(i) <= z(i)(0);
END GENERATE;
v(0) <= '0';
iteration3: FOR i IN 1 TO n-1 GENERATE
  v(i) <= z(i-1)(1);
END GENERATE;

w(0) <= '0'; w(1) <= '0';
iteration4: FOR i IN 2 TO n-1 GENERATE
  w(i) <= z(i-2)(2);
END GENERATE;
```

The counter of Fig. 7.15 can in turn be used as a building block for generating

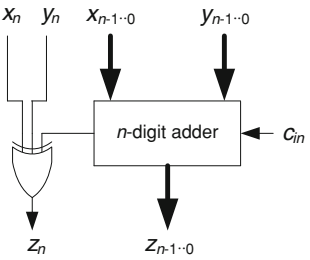

Fig. 7.18 Radix-*B B*'s complement adder

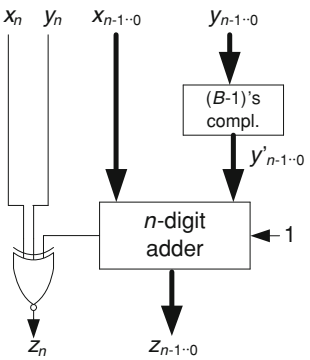

Fig. 7.19 Radix-*B B*'s complement subractor

more complex counters. As an example, the circuit of Fig. 7.16 is a 24-to-3 counter.

The computation time of the circuit of Fig. 7.16 is equal to $3T_{LUT6}$. More generally, a tree made up of 2^k-1 6-to-3 counters generates a $6{\cdot}2^{k-1}$-to-3 counter, with a computation time equal to $k{\cdot}T_{LUT6}$. In the case of Fig. 7.16, $k = 3$ and $6{\cdot}2^{k-1} = 24$.

Finally, with an additional 3-to-2 counter and an n-bit adder a 24-operand adder is obtained (Fig. 7.17). Complete VHDL models *six_to_three_counter.vhd* and *twenty_four_operand_adder.vhd* are available at the Authors' web page and examples of FPGA implementations are given in Sect. 7.9.

To summarize, an m-operand adder, with $m = 6{\cdot}2^{k-1}$, can be synthesized with 2^k-1 6-to-3 counters plus a 3-to-2 counter and an n-bit adder. Its computation time is

$$T(m,n) \cong k \cdot T_{LUT6} + T_{FA} + T_{adder}(n),$$

where $k = log_2 m + 1 - log_2 6 < log_2 m$.

Comment 7.5
More complex types of counters have been proposed (see, for example, Chap. 8 of [1], Chap. 3 of [5], Chap. 11 of [6]). Nevertheless, they do not necessarily give high performance FPGA implementations. As a matter of fact, in many cases the best FPGA implementations are based on relatively simple algorithms, to which correspond regular circuits that allow taking advantage of the special purpose carry

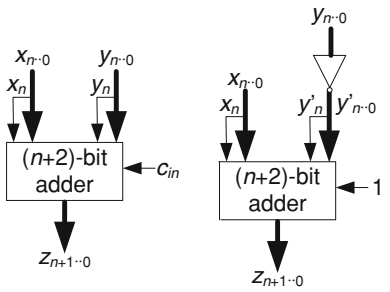

Fig. 7.20 2′s complement adder and subractor

logic circuitry, and permit the use of efficient design techniques such as pipelining and digit-serial processing.

7.8 Subtractors and Adder–Subtractors

Given two radix-B naturals x and y, the difference $z = x-y$ could be negative. So, the subtraction operation must be considered over the set of integers. A convenient way to represent integers is *B's complement*: the vector

$$x_n x_{n-1} x_{n-2} \ldots x_1 x_0, \text{ with } x_n \in \{0, 1\} \text{ and } x_i \in \{0, 1, \ldots, B-1\} \forall i < n,$$

represents

$$x = -x_n \cdot B^n + x_{n-1} \cdot B^{n-1} + x_{n-2} \cdot B^{n-2} + \cdots + x_1 \cdot B + x_0.$$

Thus, x_n is a *sign bit*: if $x_n = 0$, x is a non-negative integer (a natural), and if $x_n = 1$, x is a negative integer. The range of represented integers is

$$-B^n \leq x < B^n.$$

Let $x_n\ x_{n-1}\ x_{n-2} \ldots\ x_1\ x_0$ and $y_n\ y_{n-1}\ y_{n-2} \ldots\ y_1\ y_0$ be the B's complement representations of x and y. If the sum $z = x + y + c_{in}$, being c_{in} an initial carry, belongs to the interval $-B^n \leq z < B^n$, then z is represented by the vector $z_n\ z_{n-1}\ z_{n-2} \ldots\ z_1\ z_0$ generated by the mixed-radix adder of Fig. 7.18 (all radix-B digits but the most significant binary digits x_n, y_n and z_n).

If the difference $z = x-y$ belongs to the interval $-B^n \leq z < B^n$, then z is represented by the vector $z_n\ z_{n-1}\ z_{n-2} \ldots\ z_1\ z_0$, generated by the circuit of Fig. 7.19 in which y_i' is the (B-1)'s complement of y_i, $\forall i < n$.

The sum $z = x + y$ or the difference $z = x-y$ could lie outside the interval $-B^n \leq z < B^n$ (an *overflow* situation). In order to avoid overflows, both x and y should be represented with an additional digit. In the case of B's complement representations, *digit extension* is performed as follows:

$$x_n x_{n-1} x_{n-2} \ldots x_1 x_0 \rightarrow x_n w_n x_{n-1} x_{n-2} \ldots x_1 x_0, \text{ with } w_n = x_n \cdot (B-1).$$

For example, if $B = 10$ and $x = 249$, then x is represented by

Table 7.1 Binary adders

n	LUTs	Delay
32	32	2.25
64	64	2.98
128	128	4.44
256	256	7.35
512	512	13.1
1024	1024	24.82

Table 7.2 Radix-2^k n-bit adders

n	k	LUTs	Delay
32	4	88	2.92
64	4	176	3.11
64	5	143	3.05
64	8	152	3.64
64	16	140	4.95
128	8	304	3.85
128	12	286	4.73
128	16	280	5.04
256	16	560	5.22
256	12	572	4.98
256	13	551	4.99
512	16	1120	5.59
1024	16	2240	6.31
1024	22	2303	6.15
1024	23	2295	6.13
1024	32	2304	6.41

$$0249,\ 00249,\ 000249,\ \text{etc.}$$

If $B = 10$ and $x = -249$, then x is represented by

$$1751,\ 19751,\ 199751,\ \text{etc.}$$

Observe that if $B = 2$, then the *bit extension* operation amounts to repeating the most significant bit. In Fig. 7.20 a 2′s complement adder and a 2′s complement subtractor are shown. In both cases the comparison of bits z_{n+1} and z_n allows the detection of overflows: if $z_{n+1} \neq z_n$ then the result does not belong to the interval $-B^n \leq z < B^n$.

The following VHDL models describe the circuits of Fig. 7.20.

```
z <= (x(n)&x) + (y(n)&y) + c_in;
```

```
z <= (x(n)&x) - (y(n)&y);
```

Generic models *two_s_comp_adder.vhd* and *two_s_comp_subtractor.vhd* are available at the Authors' web page.

Table 7.3 *n*-bit carry-select adders

n	k	LUTs	Delay
32	6	84	4.83
32	8	72	3.99
32	4	60	3.64
64	8	144	6.06
64	16	199	4.17
64	4	120	4.03
128	12	417	5.37
128	16	399	4.87
256	16	799	5.69
256	32	783	5.26
256	13	819	5.64
512	16	1599	6.10
512	32	1567	6.09
512	23	1561	6.16
1024	16	3199	6.69
1024	64	3103	6.74
1024	32	3135	6.52

Table 7.4 *n*-bit carry-select adders (version 2)

n	k	Delay
32	8	3.99
256	16	5.69
512	32	6.09
1024	32	6.52

Table 7.5 *n*-bit adders with $n = n_1 \cdot n_2 \cdot n_3$

n	n_1	n_2	n_3	LUTs	Delay
256	16	4	4	1452	6.32
256	4	16	4	684	6.81
512	8	8	8	2120	10.20
512	4	16	8	1364	7.40
512	16	4	8	2904	7.33
1024	16	4	16	5808	10.33
1024	16	16	4	6242	7.79

7.9 FPGA Implementations

Several adders have been implemented within a Virtex 5-2 device. All along this section, the times are expressed in *ns* and the costs in numbers of Look Up Tables (LUTs) and flip-flops (FF's). All VHDL models as well as several test benches are available at the Authors' web page.

Table 7.6 Long-operand adders

n	s	k	FF	LUTs	Period	Total time
128	8	16	135	107	3.21	51.36
128	16	8	134	97	3.14	25.12
128	32	4	133	132	3.18	12.72
128	64	2	132	137	3.45	6.90
256	16	16	263	187	3.40	54.40
256	32	8	262	177	3.51	28.08
256	64	4	261	234	3.87	15.48
512	16	32	520	381	3.92	125.44
512	32	16	519	347	3.78	60.48
512	64	8	518	337	4.26	34.08
1024	16	64	1033	757	4.20	268.80
1024	32	32	1034	717	4.32	138.24
1024	64	16	1031	667	4.55	72.80
2048	32	64	2063	1427	4.45	284.80
2048	64	32	2056	1389	5.04	161.28

Table 7.7 Sequential multioperand adders

n	m	FF	LUTs	Period	Total time
8	4	12	23	2.25	9.00
8	8	13	32	2.37	18.96
16	16	22	90	2.71	43.36
16	8	21	56	2.57	20.56
32	32	39	363	3.72	119.04
32	16	38	170	3.09	49.44
32	64	40	684	3.89	248.96
64	64	72	1356	4.62	295.68
64	32	71	715	4.48	143.36
64	16	70	330	4.41	70.56

7.9.1 Binary Adder

The circuit is shown in Fig. 7.3. The synthesis results for several numbers n of bits are given in Table 7.1.

7.9.2 Radix 2^k Adders

The circuit is shown in Fig. 7.4. The synthesis results for several numbers $n = 2^k$ of bits are given in the Table 7.2. In these implementations, the carry propagation multiplexer *muxcy* has been explicitly instantiated within the VHDL description.

Table 7.8 Sequential carry-save adders

n	m	FF's	LUTs	Period	Total time
8	4	19	37	1.81	7.24
8	8	20	46	1.81	14.48
16	16	37	120	1.87	29.92
16	8	36	86	1.84	14.72
32	32	70	425	2.57	82.24
32	16	69	232	1.88	30.08
32	64	71	746	2.68	171.52
64	64	135	1482	2.69	172.16
64	32	134	841	2.61	83.52
64	16	133	456	1.9	30.40

Table 7.9 Multioperand addition array

n	m	LUTs	Delay
8	4	21	2.82
8	8	47	5.82
16	16	219	11.98
16	8	103	6.00
32	32	947	24.32
32	8	215	6.36
32	16	459	12.35
32	64	1923	47.11
64	64	3939	49.98
64	32	1939	25.04
64	16	939	13.07

7.9.3 Carry Select Adder

The circuit is shown in Fig. 7.6. The synthesis results for several numbers $n = m{\cdot}k$ of bits are given in Table 7.3.

The alternative circuit of Fig. 7.7 has also been implemented for several values of n. The results are given in Table 7.4.

7.9.4 Logarithmic Adders

The synthesis results for several numbers $n = n_1{\cdot}n_2{\cdot}n_3$ of bits are given in Table 7.5.

7.9.5 Long Operand Adder

The circuit is shown in Fig. 7.8. The synthesis results for several numbers $n = k{\cdot}s$ of bits are given in Table 7.6. Both the clock period T_{clk} and the total delay ($k{\cdot}T_{clk}$) are given.

Table 7.10 Combinational carry-save adder

n	m	LUTs	Delay
8	4	22	2.93
8	8	68	5.49
16	16	314	10.26
16	8	135	5.59
32	32	1388	20.03
32	8	279	5.95
32	16	649	10.65
32	64	2868	37.75
64	64	5844	39.09
64	32	2828	20.31
64	16	1321	11.35

Table 7.11 8-operand addition trees

n	LUTs	Delay
8	50	3.78
16	106	3.97
32	218	4.33
64	442	5.06

Table 7.12 24-operand adders based on 6-to-3 counters

n	LUTs	Delay
8	157	4.59
16	341	4.77
24	525	4.95
32	709	5.13
64	1445	5.86

7.9.6 Sequential Multioperand Adders

The circuit is shown in Fig. 7.9. The synthesis results for several numbers n of bits and m of operands are given in Table 7.7. Both the clock period T_{clk} and the total delay ($m \cdot T_{clk}$) are given.

The carry-save adder of Fig. 7.10 has also been implemented. The results are given in Table 7.8.

7.9.7 Combinational Multioperand Adders

The circuit of Fig. 7.12 has been implemented. The synthesis results for several numbers n of bits and m of operands are given in Table 7.9.

The carry-save adder of Fig. 7.14 has also been implemented. The results are given in Table 7.10.

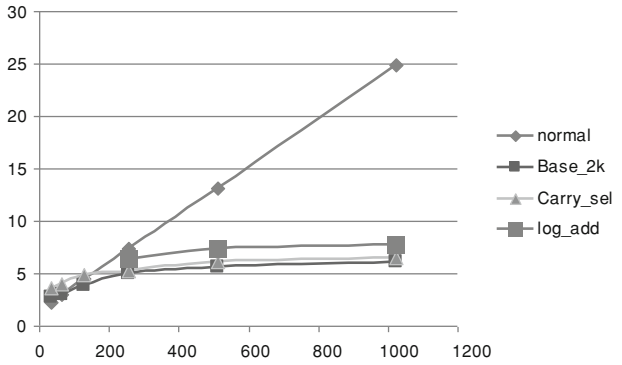

Fig. 7.21 Delay in function of the number of bits for several 2-operand adders

As an example of multioperand adddition trees (Fig. 7.13), several 8-bit adders have been implemented, with the results given in Table 7.11.

Examples of implementation results for 24-operand adders based on 6-to-3 counters (Fig. 7.17) are given in Table 7.12.

7.9.8 Comparison

A comparison between four types of 2-operand adders, namely binary (normal), radix-2^k, carry-select and logarithmic adders, has been done: Fig. 7.21 gives the corresponding adder delays (*ns*) in function of the number n of bits.

7.10 Exercises

1. Generate a generic model of a 2′s complement adder–subtractor with overflow detection.
2. An integer x can be represented under the form $(-1)^s \cdot m$ where s is the sign of x and m its magnitude (absolute value). Design an n-bit sign-magnitude adder–subtractor.
3. Design several n-bit counters, for example

   7-to-3,
   31-to-3,
   5-to-2,
   26-to-2.

4. Design a self-timed 64-bit adder with end of computation detection (*done* signal).

5. Generate several generic models of an *incrementer/decrementer*, that is, a circuit that computes $x \pm 1 \bmod m$ under the control of an *upb/down* binary signal.

References

1. Parhami B (2000) Computer arithmetic: algorithms and hardware design. Oxford University Press, New York
2. Ling H (1981) High-speed binary adder. IBM J Res Dev 25(3):156–166
3. Brent R, Kung HT (1982) A regular layout for parallel adders. IEEE Trans Comput C-31:260–264
4. Ladner RE, Fischer MJ (1980) Parallel prefix computation. J ACM 27:831–838
5. Ercegovac MD, Lang T (2004) Digital arithmetic. Morgan Kaufmann, San Francisco
6. Deschamps JP, Bioul G, Sutter G (2006) Synthesis of arithmetic circuits. Wiley, New York