Chapter 8
Multipliers

Multiplication is a basic arithmetic operation whose execution is based on 1-digit by 1-digit multipliers and multi-operand adders. Most FPGA families include the basic components for implementing fast and cost-effective multipliers. Furthermore, they also include optimized fixed-size multipliers which, in turn, can be used for implementing larger-size multipliers.

The basic multiplication algorithm is described in Sect. 8.1. Several combinational implementations are proposed in Sect. 8.2. They correspond to different types of multi-operand adders: iterative ripple-carry adders, carry-save adders, multi-operand adders based on counters, radix-2^k and mixed-radix adders. Sequential implementations are proposed in Sect. 8.3. They used the shift and add method implemented with either a ripple-carry adder or a carry-save adder. If integer operands are considered, several options are proposed in Sect. 8.4. A first method consists of multiplying B's complement integers as they are naturals; the drawback of this conceptually simple method is that the operands must be represented, and multiplied, with as many digits as the final result. Better options are a modification of the shift and add algorithm, multiplication of naturals followed by a post-correction, and the Booth algorithms. The last section describes a LUT-based method for implementing a constant multiplier, that is to say, circuits that compute $c \cdot y + u$, where c is a constant.

8.1 Basic Algorithm

Consider two radix-B numbers

$$x = x_{n-1} \cdot B^{n-1} + x_{n-2} \cdot B^{n-2} + \ldots + x_1 \cdot B + x_0 \text{ and}$$
$$y = y_{m-1} \cdot B^{m-1} + y_{m-2} \cdot B^{m-2} + \ldots + y_1 \cdot B + y_0,$$

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_8,
© Springer Science+Business Media Dordrecht 2012

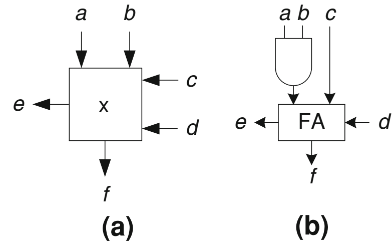

Fig. 8.1 1-digit by 1-digit multiplier. **a** Symbol, **b** internal structure ($B = 2$)

where x_i and y_i belong to $\{0, 1,\ldots, B-1\}$. An n-digit by m-digit multiplier generates a radix-B number

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

such that

$$z = x \cdot y.$$

A somewhat more general definition considers the addition of two additional numbers

$$u = u_{n-1} \cdot B^{n-1} + u_{n-2} \cdot B^{n-2} + \ldots + u_1 \cdot B + u_0 \text{ and}$$
$$v = v_{m-1} \cdot B^{m-1} + v_{m-2} \cdot B^{m-2} + \ldots + v_1 \cdot B + v_0,$$

so that

$$z = x \cdot y + u + v. \tag{8.1}$$

Observe that the maximum value of z is

$$(B^n - 1)(B^m - 1) + (B^n - 1) + (B^m - 1) = B^{n+m} - 1.$$

In order to compute (8.1), first define a 1-digit by 1-digit multiplier: given four B-ary digits a, b, c and d, it generates two B-ary digits e and f such that

$$a \cdot b + c + d = e \cdot B + f \tag{8.2}$$

(Fig. 8.1a).

If $B = 2$, it amounts to a 2-input AND gate and a 1-digit adder (Fig. 8.1b).

An n-digit by 1-digit multiplier made up of n 1-digit by 1-digit multipliers is shown in Fig. 8.2. It computes as

$$z = x \cdot b + u + d \tag{8.3}$$

where x and u are n-digit numbers, b and d are 1-digit numbers, and z is an $(n + 1)$-digit number. Observe that the maximum value of z is

$$(B^n - 1)(B - 1) + (B^n - 1) + (B - 1) = B^{n+1} - 1.$$

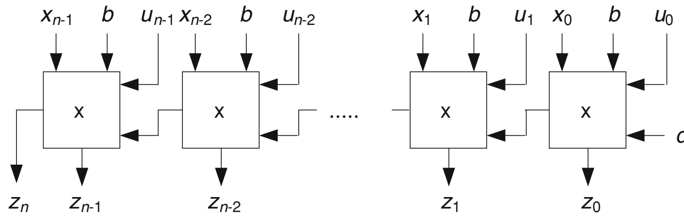

Fig. 8.2 *n*-digit by 1-digit multiplier

Using the iterative circuit of Fig. 8.2 as a computation resource, the computation of (8.1) amounts to computing the m n-digit by 1-digit products

$$\begin{aligned} z^{(0)} &= x \cdot y_0 + u + v_0, \\ z^{(1)} &= (x \cdot y_1 + v_1)B, \\ z^{(2)} &= (x \cdot y_2 + v_2)B^2, \\ &\ldots \\ z^{(m-1)} &= (x \cdot y_{m-1} + v_{m-1})B^{m-1}, \end{aligned} \tag{8.4}$$

and to adding them, that is

$$z = z^{(0)} + z^{(1)} + z^{(2)} + \ldots + z^{(m-1)} = x \cdot y + u + v. \tag{8.5}$$

For that, one of the multioperand adders of Sect. 7.7 can be used. As an example, if Algorithm 7.2 is used, then z is computed as follows.

Algorithm 8.1: Multiplication, right to left algorithm

```
accumulator := u;
for j in 0 .. m-1 loop
  accumulator := accumulator + (x·yj + vj)·B^j;
end loop;
z := accumulator;
```

8.2 Combinational Multipliers

8.2.1 Ripple-Carry Parallel Multiplier

The combinational circuit of Fig. 8.3 implements Algorithm 8.1 (with $n = 4$ and $m = 3$). One of its critical paths has been shaded. Its computation time is equal to

$$T_{multiplier}(n, m) = (n + 2m - 2) \cdot T_{multiplier}(1, 1). \tag{8.6}$$

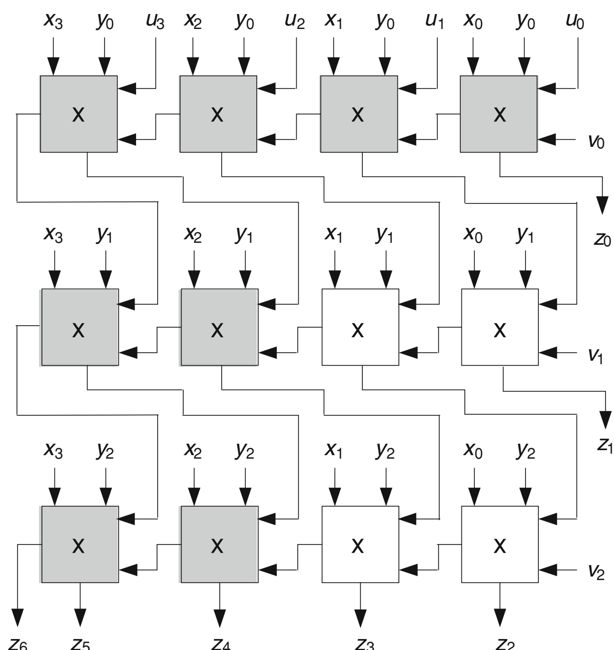

Fig. 8.3 Combinational multiplier

The following VHDL model describes the circuit of Fig. 8.3 ($B = 2$).

```
main_iteration: FOR i IN 0 TO m-1 GENERATE
  internal_iteration: FOR j IN 0 TO n-1 GENERATE
    f(i)(j) <= (x(j) AND y(i)) XOR c(i)(j) XOR d(i)(j);
    e(i)(j) <= (x(j) AND y(i) AND c(i)(j))
      OR (x(j) AND y(i) AND d(i)(j))
      OR (c(i)(j) AND d(i)(j));
  END GENERATE;
END GENERATE;
connections1: FOR j IN 0 TO n-1 GENERATE c(0)(j) <= u(j);
END GENERATE;
connections2: FOR i IN 1 TO m-1 GENERATE
  connections3: FOR j IN 0 TO n-2 GENERATE
      c(i)(j) <= f(i-1)(j+1);
  END GENERATE;
  c(i)(n-1) <= e(i-1)(n-1);
END GENERATE;
connections4: FOR i IN 0 TO m-1 GENERATE
  d(i)(0) <= v(i);
```

```
        connections5: FOR j IN 1 TO n-1 GENERATE
            d(i)(j) <= e(i)(j-1);
        END GENERATE;
      END GENERATE;
      outputs: FOR j IN 0 TO m-1 GENERATE z(j) <= f(j)(0);
      END GENERATE;
      z(m+n-2 DOWNTO m) <= f(m-1)(n-1 DOWNTO 1);
      z(m+n-1) <= e(m-1)(n-1);
```

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

8.2.2 Carry-Save Parallel Multiplier

A straightforward modification of the multiplier of Fig. 8.3, similar to the carry-save principle, is shown in Fig. 8.4. The circuit is made up of an n-by-m array of 1-by-1 multipliers, whose computation time is equal to $n \cdot T(1,1)$, plus an m-digit output adder. Its critical path has been shaded. Its computation time is equal to

$$T_{multiplier}(n,m) = n \cdot T_{multiplier}(1,1) + m \cdot T_{adder}(1) \leq (n+m) \cdot T_{multiplier}(1,1). \quad (8.7)$$

The following VHDL model describes the circuit of Fig. 8.4 ($B = 2$).

```
main_iteration: FOR i IN 0 TO m-1 GENERATE
  internal_iteration: FOR j IN 0 TO n-1 GENERATE
    f(i)(j) <= (x(j) AND y(i)) XOR c(i)(j) XOR d(i)(j);
    e(i)(j) <= (x(j) AND y(i) AND c(i)(j))
      OR (x(j) AND y(i) AND d(i)(j))
      OR (c(i)(j) AND d(i)(j));
  END GENERATE;
END GENERATE;
connections1: FOR j IN 0 TO n-1 GENERATE c(0)(j) <= u(j);
END GENERATE;
connections2: FOR i IN 1 TO m-1 GENERATE
  connections3: FOR j IN 0 TO n-2 GENERATE
      c(i)(j) <= f(i-1)(j+1);
  END GENERATE;
  c(i)(n-1) <= e(i-1)(n-1);
END GENERATE;
```

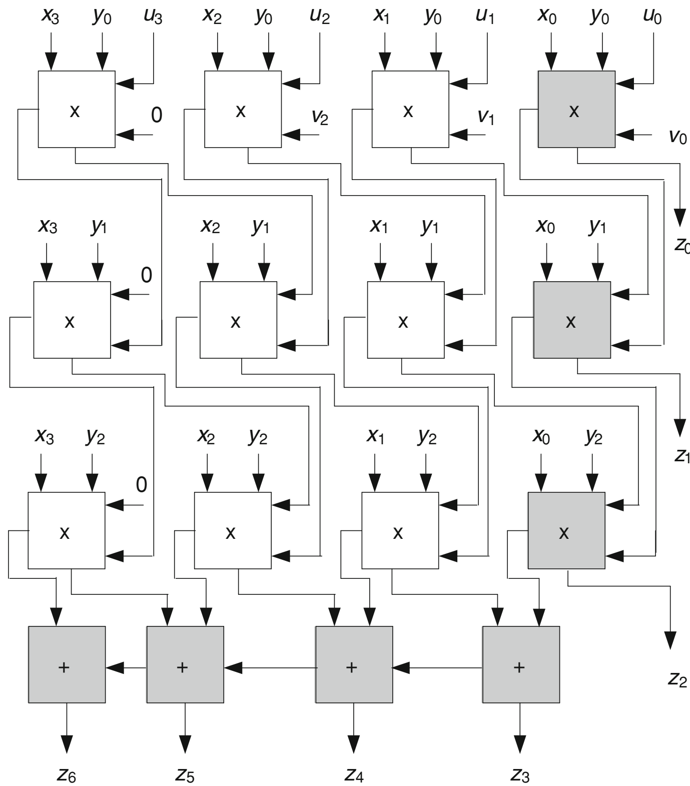

Fig. 8.4 Carry-save combinational multiplier

```
connections4: FOR i IN 0 TO m-1 GENERATE
  d(i)(0) <= v(i);
  connections5: FOR j IN 1 TO n-1 GENERATE
      d(i)(j) <= e(i)(j-1);
  END GENERATE;
END GENERATE;
outputs: FOR j IN 0 TO m-1 GENERATE z(j) <= f(j)(0);
END GENERATE;
first_operand<= f(m-1)(n-1 DOWNTO 1);
second_operand <= e(m-1);
z(n+m-1 DOWNTO m) <= first_operand + second_operand;
```

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

8.2.3 Multipliers Based on Multioperand Adders

A straightforward implementation of Eqs. (8.4) and (8.5) can also be considered (Fig. 8.5). For that, any type of multioperand adder can be used.

Example 8.1
Consider an n-bit by 7-bit multiplier. The 7-operand adder can be divided up into a 7-to-3 counter, a 3-to-2 counter and a ripple-carry adder. The complete structure is shown in Fig. 8.6 and is described by the following VHDL model:

```
yy0 <= (OTHERS => y(0));
...
yy6 <= (OTHERS => y(6));
w0 <= (x AND yy0) + ('0'&u) + v(0);
w1 <= (x AND yy1) + zero + v(1);
...
w6 <= (x AND yy6) + zero + v(6);
z0 <= "000000"&w0;
z1 <= "00000"&w1&'0';
...
z6 <= w6&"000000";
first_component: seven_to_three1 GENERIC MAP(n => n+7)
PORT MAP(x1 => z0, x2 => z1, x3 => z2, x4 => z3, x5 => z4,
x6 => z5, x7 => z6, y1 => x1, y2 => x2, y3 => x3);
second_component: csa GENERIC MAP(n => n+7)
PORT MAP(x1 => x1, x2 => x2, x3 => x3, y1 => y1, y2 => y2);
z <= y1 + y2;
```

A complete generic model *N_by_7_multiplier.vhd* is available at the Authors' web page.

Numerous multipliers, based on trees of counters, have been proposed and reported, among others the Wallace and Dadda multipliers (Wallace [4]; Dadda [3]). Nevertheless, as already mentioned before (Comment 7.5), 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 logic circuitry. To follow, an example of efficient FPGA implementation is described.

Consider the set of equations (8.4). If two successive steps are merged within an only step (loop unrolling), the new set of equations is:

$$
\begin{aligned}
z^{(1,0)} &= (x \cdot y_1 + v_1)B + x \cdot y_0 + u + v_0, \\
z^{(3,2)} &= [(x \cdot y_3 + v_3)B + (x \cdot y_2 + v_2)]B^2, \\
&\quad\cdots \\
z^{(m-1,m-2)} &= [(x \cdot y_{m-1} + v_{m-1})B + (x \cdot y_{m-2} + v_{m-2})]B^{m-2},
\end{aligned}
\tag{8.8}
$$

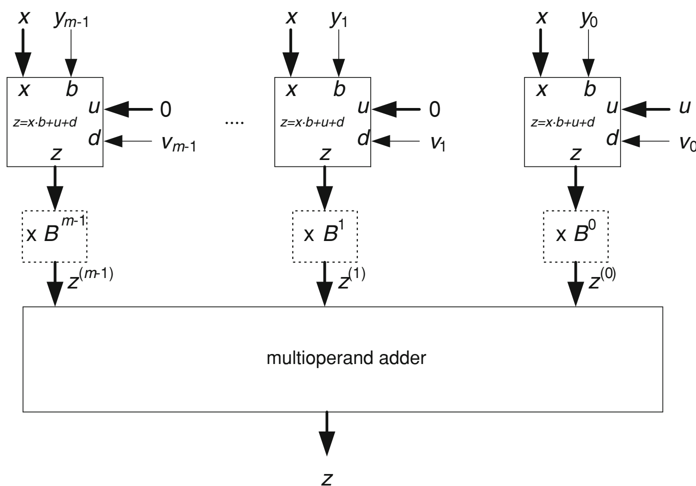

Fig. 8.5 Multiplier with a multioperand adder

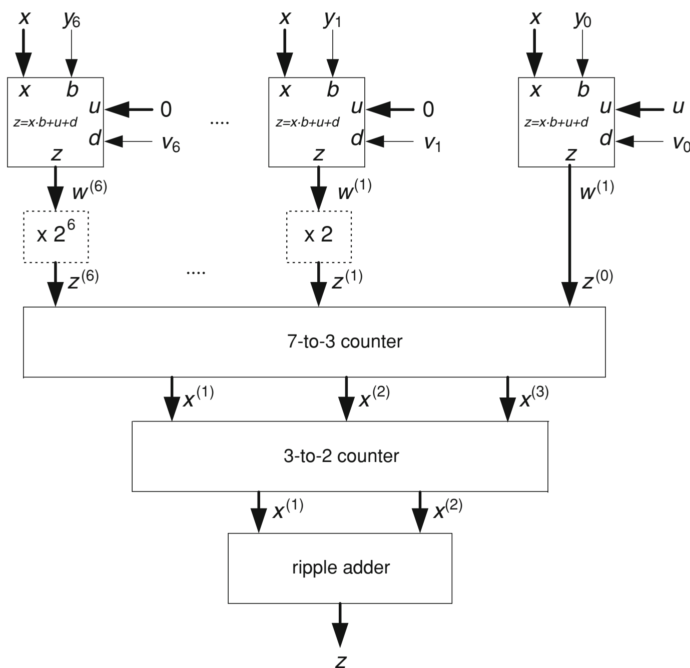

Fig. 8.6 An *n*-bit by 7-bit multiplier

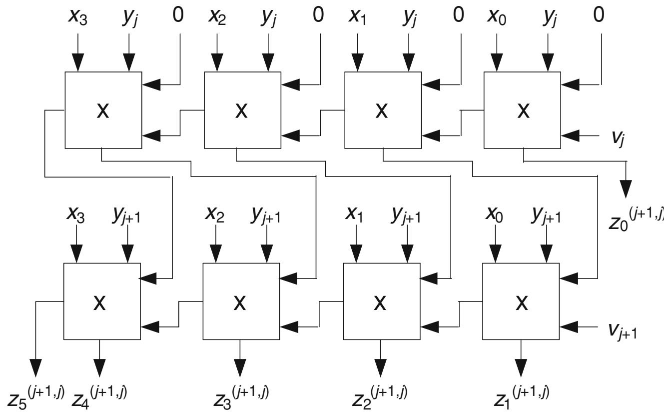

Fig. 8.7 4-digit by 2-digit multiplier

and the product is equal to

$$z = z^{(1,0)} + z^{(3,2)} + \ldots + z^{(m-1,m-2)}.$$

Assuming that $u = 0$, the basic operation to implement (8.8) is

$$z^{(i+1,i)} = \left(x \cdot y_{j+1} + v_{j+1}\right)B + \left(x \cdot y_j + v_j\right)$$

to which corresponds the circuit of Fig. 8.7 (with $n = 4$).

The circuit of Fig. 8.7 can be decomposed into $n + 1$ vertical slices of the type shown in Fig. 8.8a (with obvious simplifications regarding the first and last slices). Finally, if $B = 2$ and $v_j = 0$, the carries of the first line are equal to 0, so that the circuit of Fig. 8.8a can be implemented as shown in Fig. 8.8b.

Comment 8.1

Most FPGA's include the basic components for implementing the structure of Fig. 8.8b, and the synthesis tools have the capability to generate optimized multipliers from a simple VHDL expression, such as

$$z <= \text{ x} * \text{y};$$

Furthermore, many FPGA's also include fixed-size multiplier blocks.

8.2.4 *Radix-2^k and Mixed-Radix Parallel Multipliers*

The basic multiplication algorithm (Sect. 8.1) and the corresponding ripple-carry and carry-save multipliers (Sects. 8.2.1 and 8.2.2) have been defined for any radix-B. In particular, radix-2^k multipliers can be defined. This allows the synthesis of $n \cdot k$-bit by $m \cdot k$-bit multipliers using k-bit by k-bit multipliers as building blocks.

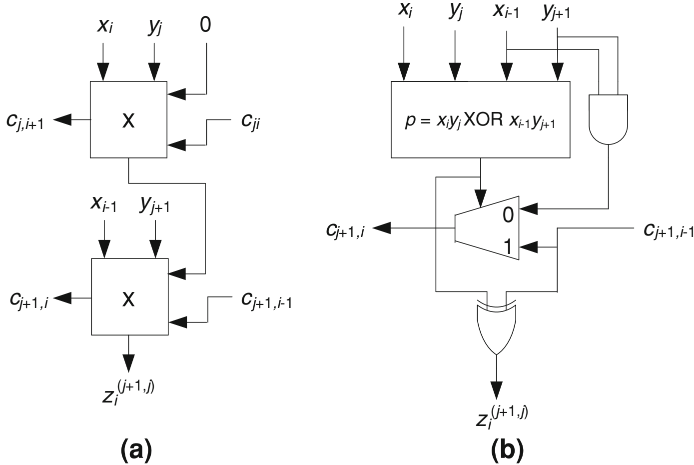

Fig. 8.8 Iterative cell of a parallel multiplier

The following VHDL model defines a radix-2^k ripple-carry parallel multiplier. The main iteration consists of $m \cdot n$ instantiations of any type of k-bit by k-bit combinational multiplier that computes $z = a \cdot b + c + d$ and represents z under the form $z_H \cdot 2^k + z_L$, where z_H and z_L are k-bit numbers:

```
main_iteration: FOR i IN 0 TO m-1 GENERATE
  internal_iteration: FOR j IN 0 TO n-1 GENERATE
    a_multiplier: k_by_k_parallel_multiplier
    GENERIC MAP(k => k)
    PORT MAP(a => x(j*k+k-1 DOWNTO j*k),
    b => y(i*k+k-1 DOWNTO i*k), c => c(i,j),
    d => d(i,j), zH => e(i,j), zL => f(i,j));
  END GENERATE;
END GENERATE;

--connections similar to those of the binary ripple-carry
--multiplier of Section 8.2.1

outputs1: FOR j IN 0 TO m-1 GENERATE
  z(j*k+k-1 DOWNTO j*k ) <= f(j,0);
END GENERATE;
outputs2: FOR j IN m TO m+n-2 GENERATE
  z(j*k+k-1 DOWNTO j*k ) <= f(m-1,j-m+1);
END GENERATE;
z(m*k+n*k-1 DOWNTO m*k+n*k-k) <= e(m-1,n-1);
```

A complete generic model *base_2k_parallel_multiplier.vhd* is available at the Authors' web page.

The stored-carry encoding can also be applied. Once again, the main iteration consists of $m \cdot n$ instantiations of any type of k-bit by k-bit multiplier, and the connections are similar to those of the carry-save multiplier of Sect. 8.2.2. A complete generic model *base_2k_csa_multiplier.vhd* is available at the Authors' web page.

A straightforward generalization of relations (8.2) to (8.5) allows defining mixed-radix combinational multipliers. First consider the circuit of Fig. 8.1a, assuming that

$$a, c \in \{0, 1, \ldots, B_1 - 1\}, \text{ and } b, d \in \{0, 1, \ldots, B_2 - 1\}.$$

Then

$$z = a \cdot b + c + d \leq (B_1 - 1) \cdot (B_2 - 1) + (B_1 - 1) + (B_2 - 1) = B_1 \cdot B_2 - 1,$$

so that z can be expressed under the form

$$z = e \cdot B_1 + f, \text{ with } e \in \{0, 1, \ldots, B_2 - 1\}, f \in \{0, 1, \ldots, B_1 - 1\}.$$

Then, consider the circuit of Fig. 8.2, assuming that x and u are n-digit radix-B_1 numbers, and b and d are 1-digit radix-B_2 numbers. Thus,

$$x \cdot b + u + d = z_n \cdot B_1^n + z_{n-1} \cdot B_1^{n-1} + \ldots + z_1 \cdot B_1 + z_0, \quad (8.9)$$

with

$$z_n \in \{0, 1, \ldots, B_2 - 1\} \text{ and } z_i \in \{0, 1, \ldots, B_1 - 1\}, \forall i \text{ in } \{0, 1, \ldots, n - 1\}.$$

Finally, given two n-digit radix-B_1 numbers x and u, and two m-digit radix-B_2 numbers y and v, compute

$$\begin{aligned} z^{(0)} &= x \cdot y_0 + u + v_0, \\ z^{(1)} &= (x \cdot y_1 + v_1)B_2, \\ z^{(2)} &= (x \cdot y_2 + v_2)B_2^2, \\ &\ldots \\ z^{(m-1)} &= (x \cdot y_{m-1} + v_{m-1})B_2^{m-1}. \end{aligned} \quad (8.10)$$

Then

$$z = z^{(0)} + z^{(1)} + z^{(2)} + \ldots + z^{(m-1)} = x \cdot y + u + v. \quad (8.11)$$

Consider the case where

$$B_1 = 2^{k_1}, \ B_2 = 2^{k_2}.$$

An easy way to define a VHDL model of the corresponding multiplier consists in first modelling a circuit that implements (8.9). The main iteration consists of n instantiations of any type of k_1-bit by k_2-bit combinational multiplier that computes

$$a \cdot b + c + d = z_H \cdot 2^{k_1} + z_L,$$

where z_H is a k_2-bit number and z_L a k_1-bit number:

```
first_cell: k1_by_k2_parallel_multiplier
GENERIC MAP(k1 => k1, k2 => k2)
PORT MAP(a => x(k1-1 DOWNTO 0), b => b,
c => u(k1-1 DOWNTO 0), d => d, zL => z(k1-1 DOWNTO 0),
zH => e(0));
iteration: FOR i IN 1 TO n-1 GENERATE
  other_cells: k1_by_k2_parallel_multiplier
  GENERIC MAP(k1 => k1, k2 => k2)
  PORT MAP(a => x(i*k1+k1-1 DOWNTO i*k1), b => b,
  c => u(i*k1+k1-1 DOWNTO i*k1), d => e(i-1),
  zL => z(i*k1+k1-1 DOWNTO i*k1), zH => e(i));
END GENERATE;
z(n*k1+k2-1 DOWNTO n*k1) <= e(n-1);
```

Then, it remains to instantiate m rows:

```
  first_row: MR_multiplier_row
  GENERIC MAP(n => n, k1 => k1, k2 => k2)
  PORT MAP(x => x, u => u, b => y(k2-1 DOWNTO 0),
    d => v(k2-1 DOWNTO 0), z => zzz(0));
    z(k2-1 DOWNTO 0) <= zzz(0)(k2-1 DOWNTO 0);
  next_rows: FOR i IN 1 TO m-1 GENERATE
    another_row: MR_multiplier_row
    GENERIC MAP(n => n, k1 => k1, k2 => k2)
    PORT MAP(x => x, u => zzz(i-1)(n*k1+k2-1 DOWNTO k2),
      b => y(i*k2+k2-1 DOWNTO i*k2),
      d => v(i*k2+k2-1 DOWNTO i*k2), z => zzz(i));
      z(i*k2+k2-1 DOWNTO i*k2) <= zzz(i)(k2-1 DOWNTO 0);
  END GENERATE;
  z(n*k1+m*k2 -1 DOWNTO m*k2) <= zzz(m-1)(n*k1+k2-1 DOWNTO k2);
```

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

The circuit defined by the preceding VHDL model is a bidirectional array similar to that of Fig. 8.3, but with more complex connections. As an example, with $k_1 = 4$ and $k_2 = 2$, the connections corresponding to cell (j, i) are shown in

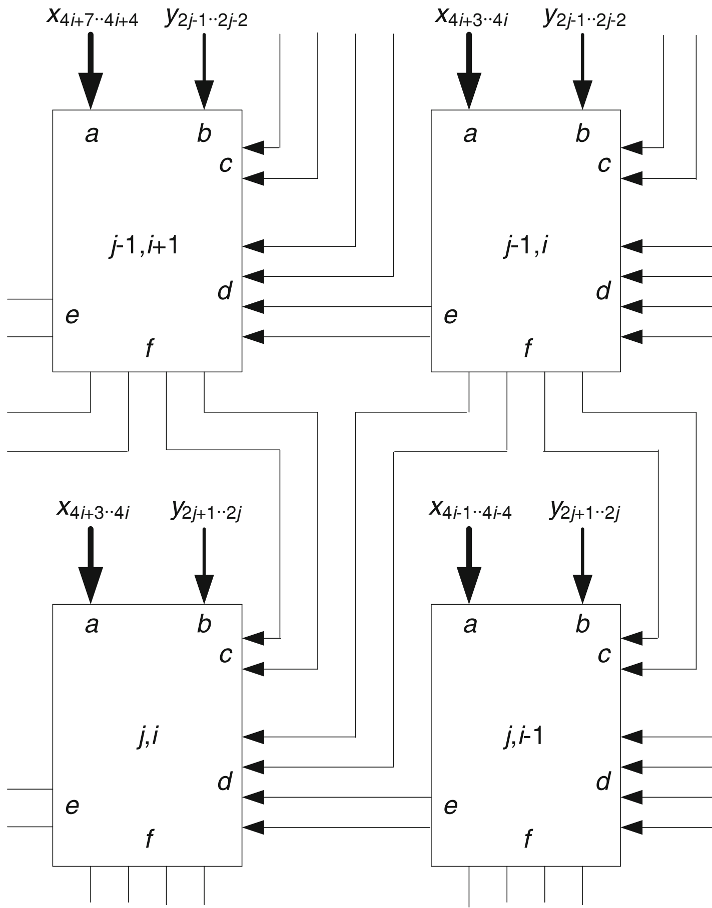

Fig. 8.9 Part of a 4*n*-bit by 2*n*-bit multiplier using 4-bit by 2-bit multiplication blocks

Fig. 8.9. As before, a stored-carry encoding circuit could also be designed, but with an even more complex connection pattern. It is left as an exercise.

8.3 Sequential Multipliers

8.3.1 Shift and Add Multiplier

In order to synthesize sequential multipliers, the basic algorithm of Sect. 8.1 can be modified. For that, Eq. (8.4) are substituted by the following:

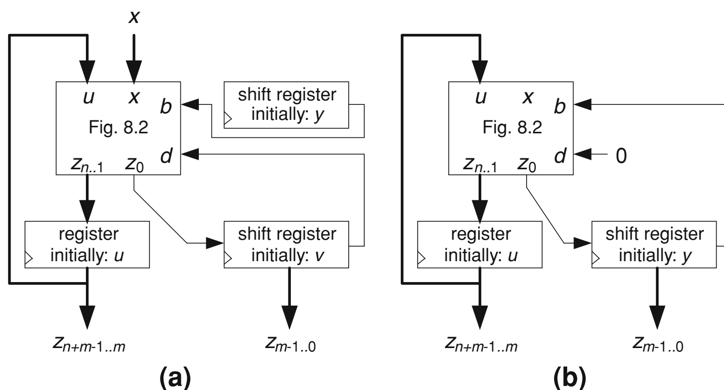

Fig. 8.10 Shift and add multipliers

$$
\begin{aligned}
z^{(0)} &= (u + x \cdot y_0 + v_0)/B, \\
z^{(1)} &= \left(z^{(0)} + x \cdot y_1 + v_1\right)/B, \\
z^{(2)} &= \left(z^{1)} + x \cdot y_2 + v_2\right)/B, \\
&\ldots \\
z^{(m-1)} &= \left(z^{(m-2)} + x \cdot y_{m-1} + v_{m-1}\right)/B.
\end{aligned} \tag{8.12}
$$

Multiply the first equation by B, the second by B^2, and so on, and add the so obtained equations. The result is

$$
\begin{aligned}
z^{(m-1)}B^m &= u + x \cdot y_0 + v_0 + (x \cdot y_1 + v_1)B + \ldots + (x \cdot y_{m-1} + v_{m-1})B^{m-1} \\
&= xy + u + v.
\end{aligned}
$$

Algorithm 8.2: Shift and add multiplication

```
accumulator := u;
for j in 0 .. m-1 loop
  accumulator := (accumulator + x·yj + vj)/B;
end loop;
z := accumulator·B^m;
```

A data path for executing Algorithm 8.2 is shown in Fig. 8.10a. The following VHDL model describes the circuit of Fig. 8.10a ($B = 2$).

```
carries(0) <= acc_0(0);
main_iteration: FOR i IN 0 TO n-1 GENERATE
  product(i) <=
    (x(i) AND int_y(0)) XOR acc_1(i) XOR carries(i);
  carries(i+1) <=
    (x(i) AND int_y(0) AND acc_1(i)) OR
    (x(i) AND int_y(0) AND carries(i)) OR
    (acc_1(i) AND carries(i));
END GENERATE;
product(n) <= carries(n);
z <= acc_1 & acc_0;
parallel_register: PROCESS(clk)
BEGIN
  IF clk'EVENT and clk = '1' THEN
    IF load = '1' THEN acc_1 <= u;
    ELSIF update = '1' THEN acc_1 <= product(n DOWNTO 1);
    END IF;
  END IF;
END PROCESS;
shift_register1: PROCESS(clk)
BEGIN
  IF clk'EVENT and clk = '1' THEN
    IF load = '1' THEN acc_0 <= v;
    ELSIF update = '1' THEN
      acc_0 <= product(0)&acc_0(m-1 DOWNTO 1);
    END IF;
  END IF;
END PROCESS;
shift_register2: PROCESS(clk)
BEGIN
  IF clk'EVENT and clk = '1' THEN
    IF load = '1' THEN int_y <= y;
    ELSIF update = '1' THEN
      int_y <= '0'& int_y(m-1 DOWNTO 1);
    END IF;
  END IF;
END PROCESS;
```

The complete circuit also includes an m-state counter and a control unit. A complete generic model *shift_and_add_multiplier.vhd* is available at the Authors' web page.

If $v = 0$, the same shift register can be used for storing both y and the least significant bits of z. The modified circuit is shown in Fig. 8.10b. A complete generic model *shift_and_add_multiplier2* is also available.

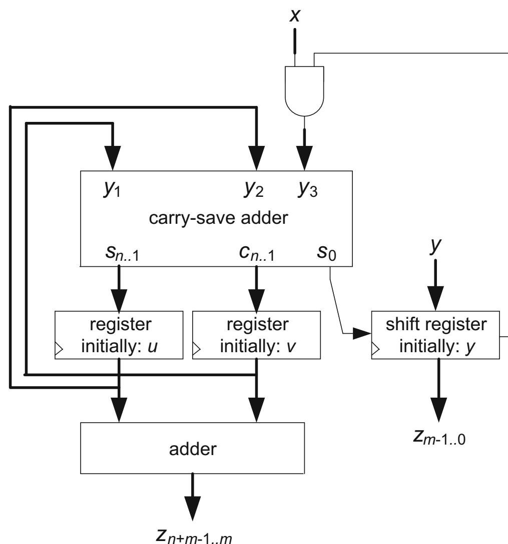

Fig. 8.11 Sequential carry-save multiplier

The computation time of the circuits of Fig. 8.10 is approximately equal to

$$T_{multiplier}(n, m) = m \cdot T_{multiplier}(n, 1) = m \cdot n \cdot T_{multiplier}(1, 1). \tag{8.13}$$

8.3.2 Shift and Add Multiplier with CSA

The shift and add algorithm can also be executed with stored-carry encoding. After m steps the result is obtained under the form

$$s_{n-1} B^{n+m-1} + (c_{n-2} + s_{n-2})B^{n+m-2} + \ldots + (c_0 + s_0)B^m + z_{m-1} B^{m-1} + \ldots \\ + z_1 B + z_0,$$

and an additional n-digit adder computes

$$s_{n-1} B^{n-1} + (c_{n-2} + s_{n-2})B^{n-2} + \ldots + (c_0 + s_0) \\ = z_{m+n-1} B^{n-1} + \ldots + z_{m+1} B + z_m.$$

The corresponding data path is shown in Fig. 8.11. The carry-save adder computes

$$y_1 + y_2 + y_3 = s + c,$$

where y_1, y_2 and y_3 are n-bit numbers, and s and c are $(n + 1)$-bit numbers. At the end of step i, the less significant bit of s is z_i, and the n most significant bits of s and c are transmitted to the next step:

```
xy <= '0'&(x AND y_i);
main_component: csa GENERIC MAP(n => n+1)
PORT MAP(y1 => xy, y2 => s , y3 => c, s => next_s,
  c => next_c);
register_s: PROCESS(clk) ...
register_c: PROCESS(clk) ...
shift_register: PROCESS(clk)
BEGIN
  IF clk'EVENT and clk = '1' THEN
    IF load = '1' THEN int_y <= y;
    ELSIF update = '1' THEN
      int_y <= next_s(0)& int_y(m-1 DOWNTO 1);
    END IF;
  END IF;
END PROCESS;
y_i <= (OTHERS => int_y(0));
z(m-1 DOWNTO 0) <= int_y(m-1 DOWNTO 0);
z(m+n-1 DOWNTO m) <= s(n-1 DOWNTO 0) + c(n-1 DOWNTO 0);
```

The complete circuit also includes an m-state counter and a control unit. A complete generic model *sequential_CSA_multiplier.vhd* is available at the Authors' web page. The minimum clock period is equal to the delay of a 1-bit by 1-bit multiplier. Thus, the total computation time is equal to

$$T_{multiplier}(n,m) = m \cdot T_{multiplier}(1,1) + T_{adder}(n) \leq (n+m) \cdot T_{multiplier}(1,1). \tag{8.14}$$

Comment 8.2

In *sequential_CSA_multiplier.vhd* the *done* flag is raised as soon as the final values of the adder inputs are available. A more correct control unit should raise the flag k cycles later, being $k \cdot T_{clk}$ an upper bound of the n-bit adder delay. The value of k could be defined as a generic parameter (Exercise 8.3).

8.4 Integers

Given four B's complement integers

$$x = x_n x_{n-1} x_{n-2} \ldots x_0,\ \ y = y_m y_{m-1} y_{m-2} \ldots y_0,\ \ u = u_n u_{n-1}\ u_{n-2} \ldots u_0,$$
$$v = v_m v_{m-1} v_{m-2} \ldots v_0,$$

belonging to the ranges

$$-B^n \leq x < B^n, -B^m \leq y < B^m, -B^n \leq u < B^n, -B^m \leq v < B^m,$$

then $z = x \cdot y + u + v$ belongs to the interval

$$-B^{n+m+1} \leq z < B^{n+m+1}.$$

Thus, z is a B's complement number of the form

$$z = z_{n+m+1}\, z_{n+m}\; z_{n+m-1} \ldots z_1 z_0.$$

8.4.1 Mod $2B^{n+m}$ Multiplication

The integer represented by a vector $x_n\, x_{n-1}\, x_{n-2} \ldots x_1\, x_0$ is

$$x = -x_n B^n + x_{n-1}\, B^{n-1} + x_{n-2}\, B^{n-2} + \ldots + x_1 B + x_0,$$

while the natural *natural*(x) represented by this same vector is

$$natural\,(x) = x_n B^n + x_{n-1}\, B^{n-1} + x_{n-2}\, B^{n-2} + \ldots + x_1 B + x_0.$$

As $x_n \in \{0,\ 1\}$, either *natural*(x) = x or *natural*(x) = x +$2B^n$. So,

$$natural(x) = x \bmod 2B^n.$$

The following method can be used to compute $z = x \cdot y + u + v$. First, represent the operands x, y, u and v with the same number of digits ($n + m + 2$) as the result z (digit extension, Sect. 7.8). Then, compute $z = x \cdot y + u + v$ as if x, y, u and v were naturals:

$$z = natural(x) \cdot natural(y) + natural(u) + natural(v) = natural(x \cdot y + u + v).$$

Finally, reduce z modulo $2B^{n+m+1}$. Assume that before the mod $2B^{n+m+1}$ reduction

$$z = \ \ldots + z_{n+m+1}\, B^{n+m+1} + z_{n+m}\, B^{n+m} + z_{n+m-1}\, B^{n+m-1} + \ldots + z_1 B + z_0;$$

then

$$z \bmod 2B^{n+m+1} = (\ldots + z_{n+m+1} \bmod 2)B^{n+m+1} + z_{n+m}\, B^{n+m} + z_{n+m+1}\, B^{n+m-1} + \ldots + z_1 B + z_0.$$

In particular, if B is even,

$$z \bmod 2B^{n+m+1} = (z_{n+m+1}\ 2)B^{n+m+1} + z_{n+m}\, B^{n+m} + z_{n+m-1}\, B^{n+m-1} + \ldots + z_1 B + z_0.$$

Example 8.2
Assume that $B = 10$, $n = 4$, $m = 3$, $x = 7918$, $y = -541$, $u = -7017$, $v = 742$, and compute $z = 7918{\cdot}(-541) + (-7017) + 742$. In 10's complement: $x = 07918$, $y = 1459$, $u = 12983$, $v = 0742$.

1. Express all operands with 9 digits: $x = 000007918$, $y = 199999459$, $u = 199992983$, $v = 000000742$.

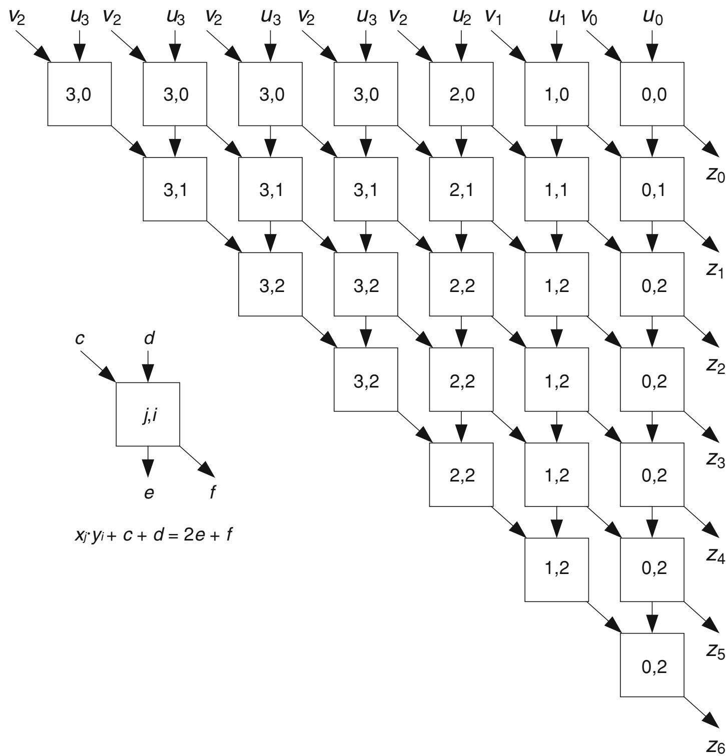

Fig. 8.12 Carry-save multiplier for integers ($n = 3$, $m = 2$)

2. Compute $x \cdot y + u + v$: 000007918·199999459 + 199992983 + 000000742 = 1583795710087.
3. Reduce 1583795710087 modulo $2{\cdot}10^8$: (1583795710087) mod $2{\cdot}10^8 = (7 \bmod 2){\cdot}10^8 + 95710087 = 195710087$.

The result 195710087 is the 10's complement representation of −4289913.

Thus, any multiplier for natural numbers can be used. As an example, an $(n + m + 2)$-digit by $(n + m + 2)$-digit carry-save multiplier could be used (Fig. 8.4). As the result is reduced modulo $2B^{n+m+1}$, only the rightmost part of the circuit is used (if B is even), so that there is no output adder, and the most significant digit is reduced mod 2. An example with $n = 3$ and $m = 2$ is shown in Fig. 8.12. The corresponding computation time is equal to

$$(n + m + 2) \cdot T_{multiplier}(1, 1). \tag{8.15}$$

This delay is practically the same as that of a carry-save combinational multiplier (8.7). Nevertheless, the number of 1-digit by 1-digit multiplication cells is equal to $1 + 2 + 3 + \ldots + (n + m + 2) = (n + m + 2)(n + m + 3)/2$ instead of $n \cdot m$.

A very simple way to generate a VHDL model consists of defining $(n + m + 2)$-bit representations of all operands and instantiating an $(n + m + 2)$-bit by $(n + m + 2)$-bit carry-save multiplier:

```
long_x(n+m+1 DOWNTO n+1) <= (OTHERS => x(n));
long_x(n DOWNTO 0) <= x;
long_u(n+m+1 DOWNTO n+1) <= (OTHERS => u(n));
long_u(n DOWNTO 0) <= u;
long_y(n+m+1 DOWNTO m+1) <= (OTHERS => y(m));
long_y(m DOWNTO 0) <= y;
long_v(n+m+1 DOWNTO m+1) <= (OTHERS => v(m));
long_v(m DOWNTO 0) <= v;
main_component: parallel_csa_multiplier
GENERIC MAP(n => n+m+2, m => n+m+2)
PORT MAP(x => long_x, u => long_u, y => long_y, v => long_v,
  z => long_z);
z <= long_z(n+m+1 DOWNTO 0);
```

Only $n + m + 2$ output bits of the carry-save multiplier are connected to output ports, and the synthesis program will prune the circuit accordingly.

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

To conclude, this approach is conceptually attractive because any type of multiplier for natural numbers can be used. Nevertheless, the cost of the corresponding circuits is very high.

8.4.2 Modified Shift and Add Algorithm

Consider again four B's complement integers

$$x = x_n\, x_{n-1}\, x_{n-2} \ldots x_0, y = y_m y_{m-1}\, y_{m-2} \ldots y_0,\; u = u_n\, u_{n-1}\; u_{n-2} \ldots u_0,$$
$$v = v_m\, v_{m-1}\, v_{m-2} \ldots v_0.$$

A set of equations similar to (8.12) can be defined:

$$\begin{aligned}
z^{(0)} &= (u + x \cdot y_0 + v_0)/B,\\
z^{(1)} &= \left(z^{(0)} + x \cdot y_1 + v_1\right)/B,\\
z^{(2)} &= \left(z^{1)} + x \cdot y_2 + v_2\right)/B,\\
&\ldots\\
z^{(m-1)} &= \left(z^{(m-2)} + x \cdot y_{m-1} + v_{m-1}\right)/B,\\
z^{(m)} &= \left(z^{(m-1)} - x \cdot y_m - v_m\right)/B.
\end{aligned} \tag{8.16}$$

Multiply the first equation by B, the second by B^2, and so on, and add the $m + 1$ so obtained equations. The result is

$$z^{(m)}B^{m+1} = u + x \cdot y_0 + v_0 + (x \cdot y_1 + v_1)B + \ldots + (x \cdot y_{m-1} + v_{m-1})B^{m-1} \\ - (x \cdot y_m + v_m)B^m \\ = xy + u + v.$$

Algorithm 8.3: Modified shift and add multiplication

```
accumulator := u;
for j in 0 .. m-1 loop
  accumulator := (accumulator + x·yj + vj)/B;
end loop;
accumulator := (accumulator - x·ym - vm)/B;
z := accumulator·B^(m+1);
```

In what follows it is assumed that $v_m = 0$, that is to say $v \geq 0$; so, in order to implement Algorithm 8.3, the two following computation primitives must be defined:

$$z = u + x \cdot b + d \tag{8.17}$$

and

$$z = u - x \cdot b, \tag{8.18}$$

where

$$-B^n \leq x < B^n, -B^n \leq u < B^n, 0 \leq b < B, 0 \leq d < B.$$

Thus, in the first case,

$$-B^{n+1} \leq z < B^{n+1},$$

and in the second case

$$-B^{n+1} + (B - 1) \leq z < B^{n+1},$$

so that in both cases z is an $(n + 2)$-digit B's complement integer and $natural(z) = z \bmod 2B^{n+1}$.

The first primitive (8.17) is implemented by the circuit of Fig. 8.13 and the second (8.18) by the circuit of Fig. 8.14. In both, circuit z_{n+1} is computed modulo 2.

As an example, the combinational circuit of Fig. 8.15 implements Algorithm 8.3 (with $n = m = 2$). Its cost and computation time are practically the same as in the case of a ripple-carry multiplier for natural numbers. It can be described by the following VHDL model.

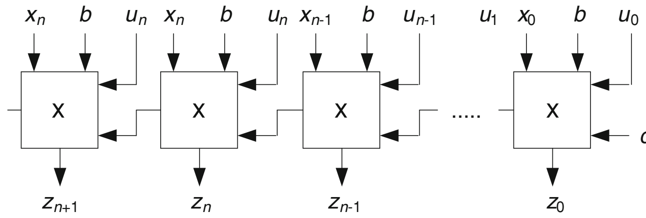

Fig. 8.13 First computation primitive

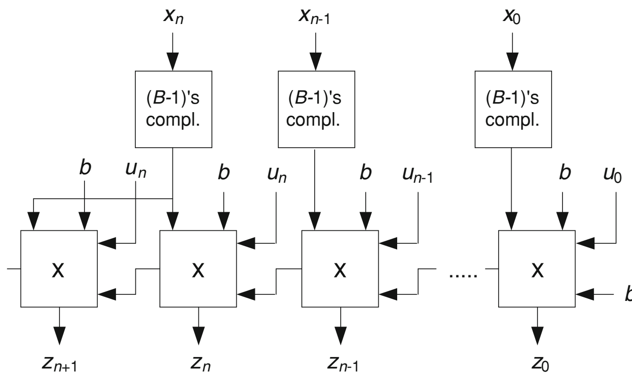

Fig. 8.14 Second computation primitive

```
main_iteration: FOR i IN 0 TO m-1 GENERATE
  internal_iteration: FOR j IN 0 TO n GENERATE
    f(i)(j) <= (x(j) AND y(i)) XOR c(i)(j) XOR d(i)(j);
    e(i)(j) <= (x(j) AND y(i) AND c(i)(j))
      OR (x(j) AND y(i) AND d(i)(j))
      OR (c(i)(j) AND d(i)(j));
  END GENERATE;
  f(i)(n+1) <= (x(n) AND y(i)) XOR c(i)(n+1) XOR d(i)(n+1);
  e(i)(n+1) <= (x(n) AND y(i) AND c(i)(n+1))
    OR (x(n) AND y(i) AND d(i)(n+1))
    OR (c(i)(n+1) AND d(i)(n+1));
END GENERATE;
last_row: FOR j IN 0 TO n GENERATE
  f(m)(j) <= (NOT(x(j)) AND y(m)) XOR c(m)(j) XOR d(m)(j);
  e(m)(j) <= (NOT(x(j)) AND y(m) AND c(m)(j)) OR
    (NOT(x(j)) AND y(m) AND d(m)(j)) OR
    (c(m)(j) AND d(m)(j));
END GENERATE;
```

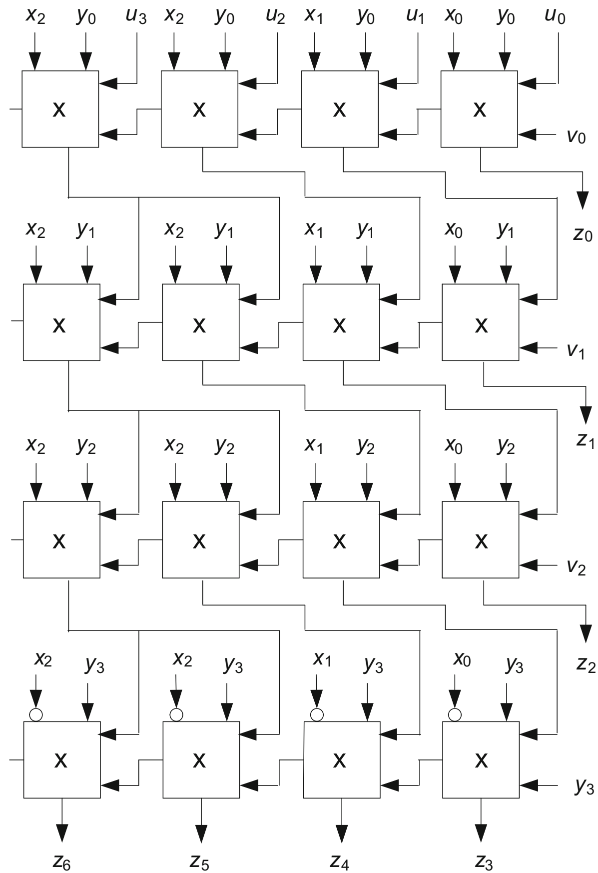

Fig. 8.15 Combinational multiplier for integers ($B = 2$, $m = n = 2$)

```
f(m)(n+1) <=
   (NOT(x(n)) AND y(m)) XOR c(m)(n+1) XOR d(m)(n+1);
e(m)(n+1) <= (NOT(x(n)) AND y(m) AND c(m)(n+1))
   OR (NOT(x(n)) AND y(m) AND d(m)(n+1))
   OR (c(m)(n+1) AND d(m)(n+1));
connections1: FOR j IN 0 TO n GENERATE c(0)(j) <= u(j);
END GENERATE;
c(0)(n+1) <= u(n);
connections2: FOR i IN 1 TO m GENERATE
  connections3: FOR j IN 0 TO n GENERATE
    c(i)(j) <= f(i-1)(j+1);
  END GENERATE;
  c(i)(n+1) <= f(i-1)(n+1);
END GENERATE;
```

```
connections4: FOR i IN 0 TO m-1 GENERATE
  d(i)(0) <= v(i);
  connections5: FOR j IN 1 TO n+1 GENERATE
    d(i)(j) <= e(i)(j-1);
  END GENERATE;
END GENERATE;
d(m)(0) <= y(m);
connections6: FOR j IN 1 TO n+1 GENERATE
  d(m)(j) <= e(m)(j-1);
END GENERATE;
outputs: FOR j IN 0 TO m GENERATE z(j) <= f(j)(0);
END GENERATE;
z(m+n+1 DOWNTO m+1) <= f(m)(n+1 DOWNTO 1);
```

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

The design of a sequential multiplier based on Algorithm 8.3 is left as an exercise.

8.4.3 Post Correction Multiplication

Given four B's complement integers

$$x = x_n\, x_{n-1}\, x_{n-2} \ldots x_0, y = y_m\, y_{m-1}\, y_{m-2} \ldots y_0, u = u_n u_{n-1}\, u_{n-2} \ldots u_0,$$
$$v = v_m v_{m-1}\, v_{m-2} \ldots v_0,$$

then $z = x \cdot y + u + v$, belonging to the interval $-B^{n+m+1} \leq z < B^{n+m+1}$, can be expressed under the form

$$z = (X_0 \cdot Y_0 + U_0 + V_0) + x_n \cdot y_m \cdot B^{n+m} - (x_n \cdot Y_0 + u_n) \cdot B^n - (y_m \cdot X_0 + v_m) \cdot B^n,$$

where X_0, Y_0, U_0 and V_0 are four naturals

$$X_0 = x_{n-1}\, x_{n-2} \ldots x_0, Y_0 = y_{m-1}\, y_{m-2} \ldots y_0,\; U_0 = u_{n-1}\, u_{n-2} \ldots u_0,$$
$$X_0 = v_{m-1} v_{m-2} \ldots v_1 v_0$$

deduced from x, y, u and v by eliminating the sign bits. Thus, the computation of z amounts to the computation of

$$Z_0 = X_0 \cdot Y_0 + U_0 + V_0,$$

that can be executed by any type of multiplier for naturals, plus a post correction that consists of several additions and left shifts.

If $B = 2$ and $u = v = 0$, then

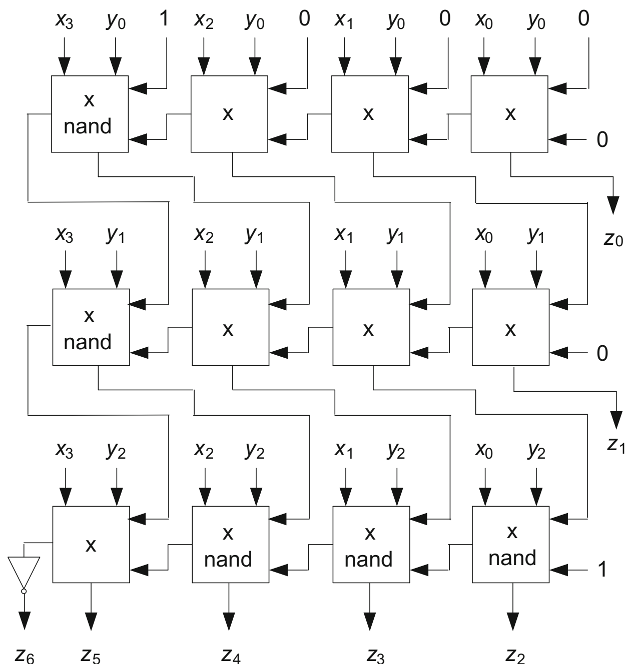

Fig. 8.16 Multiplier with post correction

$$z = x \cdot y = X_0 \cdot Y_0 + x_n \cdot y_m \cdot 2^{n+m} - x_n \cdot Y_0 \cdot 2^n - y_m \cdot X_0 \cdot 2^n.$$

The $(n + m + 2)$-bit 2's complement representations of $-x_n \cdot Y_0 \cdot 2^n$ and $-y_m \cdot X_0 \cdot 2^m$ are

$$(2^{m+1} + 2^m + \overline{(x_n \cdot y_{m-1})} \cdot 2^{m-1} + \ldots + \overline{(x_n \cdot y_0)} \cdot 2^0 + 1) \cdot 2^n \bmod 2^{n+m-2},$$

and

$$(2^{n+1} + 2^n + \overline{(y_m \cdot x_{n-1})} \cdot 2^{n-1} + \ldots + \overline{(y_m \cdot x_0)} \cdot 2^0 + 1) \cdot 2^m \bmod 2^{n+m-2},$$

so that the representation of $x_n \cdot y_m \cdot 2^{n+m} - x_n \cdot Y_0 \cdot 2^n - y_m \cdot X_0 \cdot 2^n$ is

$$(2^{n+m+1} + x_n \cdot y_m \cdot 2^{n+m} + \overline{(x_n \cdot y_{m-1})} \cdot 2^{n+m-1} + \ldots + \overline{(x_n \cdot y_0)} \cdot 2^n$$
$$+2^n + \overline{(y_m \cdot x_{n-1})} \cdot 2^{n+m-1} + \ldots + \overline{(y_m \cdot x_0)} \cdot 2^m + 2^m) \bmod 2^{n+m+2}.$$

A simple modification of the combinational multipliers of Fig. 8.3 and 8.4 allows computing $x \cdot y$, where x is an $(n + 1)$-bit 2's complement integer and y an $(m + 1)$-bit 2's complement integer. An example is shown in Fig. 8.16 $(n = 3,$

$m = 2$). The *nand* multiplication cells are similar to that of Fig. 8.1b, but for the substitution of the AND gate by a NAND gate [1].
The following VHDL model describes the circuit of Fig. 8.16.

```
main_iteration: FOR i IN 0 TO m-1 GENERATE
  internal_iteration: FOR j IN 0 TO n-1 GENERATE
    f(i)(j) <= (x(j) AND y(i)) XOR c(i)(j) XOR d(i)(j);
    e(i)(j) <= (x(j) AND y(i) AND c(i)(j)) OR
      (x(j) AND y(i) AND d(i)(j)) OR (c(i)(j) AND d(i)(j));
  END GENERATE;
END GENERATE;
first_column: FOR i IN 0 TO m-1 GENERATE
  f(i)(n) <= (x(n) NAND y(i)) XOR c(i)(n) XOR d(i)(n);
  e(i)(n) <= ((x(n) NAND y(i)) AND c(i)(n)) OR
    ((x(n) NAND y(i)) AND d(i)(n)) OR (c(i)(n) AND d(i)(n));
END GENERATE;
last_row: FOR j IN 0 TO n-1 GENERATE
  f(m)(j) <= (x(j) NAND y(m)) XOR c(m)(j) XOR d(m)(j);
  e(m)(j) <= ((x(j) NAND y(m)) AND c(m)(j)) OR
    ((x(j) NAND y(m)) AND d(m)(j)) OR (c(m)(j) AND d(m)(j));
END GENERATE;
f(m)(n) <= (x(n) AND y(m)) XOR c(m)(n) XOR d(m)(n);
e(m)(n) <= (x(n) AND y(m) AND c(m)(n)) OR
  (x(n) AND y(m) AND d(m)(n)) OR (c(m)(n) AND d(m)(n));

--connections similar to those of a multiplier for naturals
--(Fig.8.3)

outputs: FOR j IN 0 TO m GENERATE z(j) <= f(j)(0);
END GENERATE;
z(m+n DOWNTO m+1) <= f(m)(n DOWNTO 1);
z(m+n+1) <= NOT(e(m)(n));
```

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

8.4.4 Booth Multiplier

Given an $(m + 1)$-bit 2's complement integer $y = -y_m \cdot 2^m + y_{m-1} \cdot 2^{m-1} + \ldots + y_1 \cdot 2 + y_0$, define

$$y'_0 = -y_0 \text{ and } y'_j = -y_j + y_{j-1},\ \forall i \text{ in } \{1, 2, \ldots, m\},$$

so that all coefficients y_i' belong to {−1, 0, 1}. Then y can be represented under the form

$$y = y'_m \cdot 2^m + y'_{m-1} \cdot 2^{m-1} + \ldots + y'_1 \cdot 2 + y'_0,$$

the so-called Booth's encoding of y (Booth [2]. Unlike the 2's complement representation in which y_m has a specific function, all coefficients y_i' have the same function. Formally, the Booth's representation of an integer is the same as the binary representation of a natural. The basic multiplication algorithm (Algorithm 8.1), with $v = 0$, can be used.

Algorithm 8.4: Booth multiplication, z = x·y + u

```
accumulator := u;
for j in 0 .. m-1 loop
  accumulator := accumulator + x·yj'·2^j;
end loop;
z := accumulator;
```

The following VHDL model describes a combinational circuit based on Algorithm 8.4.

```
a(0) <= ((u(n)&u) - (x(n)&x)) WHEN y(0) = '1'
  ELSE u(n)&u;
z(0) <= a(0)(0);
main_iteration: FOR i IN 1 TO m GENERATE
  a(i) <= ((a(i-1)(n+1)&a(i-1)(n+1 DOWNTO 1)) - (x(n)&x))
    WHEN (y(i-1) = '0' AND y(i) = '1')
  ELSE ((a(i-1)(n+1)&a(i-1)(n+1 DOWNTO 1)) + (x(n)&x))
    WHEN (y(i-1) = '1' AND y(i) = '0')
  ELSE a(i-1)(n+1)&a(i-1)(n+1 DOWNTO 1);
  z(i) <= a(i)(0);
END GENERATE;
z(n+m+1 DOWNTO m+1) <= a(m)(n+1 DOWNTO 1);
```

A complete generic model *Booth1_multiplier.vhd* is available at the Authors' web page.

Higher radix Booth multipliers can be defined. Given an $(m + 1)$-bit 2's complement integer $y = -y_m \cdot 2^m + y_{m-1} \cdot 2^{m-1} + \ldots + y_1 \cdot 2 + y_0$, where m is odd, define

$$y'_0 = -2 \cdot y_1 + y_0, y'_i = -2 \cdot y_{2 \cdot i+1} + y_{2 \cdot i} + y_{2 \cdot i-1}, \forall i \, \text{in} \{1, 2, \ldots, (m-1)/2\},$$

so that all coefficients y_i' belong to {−2, −1, 0, 1, 2}. Then y can be represented under the form

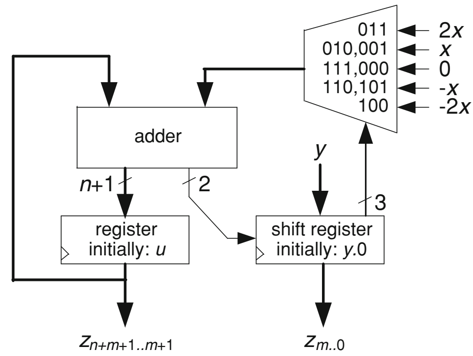

Fig. 8.17 Sequential radix-4 Booth multiplier

$$y = y'_{(m-1)/2} \cdot 4^{(m-1)/2} + y'_{(m-1)/2-1} \cdot 4^{(m-1)/2-1} + \ldots + y'_1 \cdot 4 + y'_0,$$

the so-called Booth-2 encoding of y.

Example 8.3
Consider the case where $m = 9$ and thus $(m-1)/2 = 4$. The 2's complement representation of -137 is 1101110111. The corresponding Booth-2 encoding is $-1\ 2\ -1\ 2\ -1$ and, indeed, $-4^4 + 2 \cdot 4^3 - 4^2 + 2 \cdot 4 - 1 = -137$.The basic radix-4 multiplication algorithm, with $v = 0$, can be used.

Algorithm 8.5: Radix-4 Booth multiplication, z = x · y + u

```
accumulator := u;
for j in 0 .. (m-1)/2 - 1 loop
  accumulator := accumulator + x·yj'·2^j;
end loop;
z := accumulator;
```

A sequential implementation is shown in Fig. 8.17. It includes a shift register whose content is shifted two positions at each step, a parallel register and an adder whose second operand is $-2x$, $-x$, 0, x or $2x$ depending on the three least significant bits ($y_{2\cdot i+1}$, $y_{2\cdot i}$, $y_{2\cdot i-1}$) of the shift register. At each step, two output bits are generated. Hence, the total computation time is equal to $(m + 1)/2 \cdot T_{clk}$, where T_{clk} must be greater than the computation time of an $(n + 3)$-bit adder. Thus,

$$T(n,m) \cong \frac{m+1}{2} \cdot T_{adder}(n+3).$$

With respect to a radix-2 shift and add multiplier (Sect. 8.2.1), the computation time has been divided by 2.

The following VHDL model describes the circuit of Fig. 8.17.

```
long_x <= x(n)&(x(n)&x); minus_x <= NOT(long_x)+1;
two_x <= x(n)&(x&'0'); minus_two_x <= NOT(two_x)+1;
zero <= (OTHERS => '0');
yyy <= acc0(2 DOWNTO 0);
WITH  yyy  SELECT  second_operand  <=  zero  WHEN  "000"|"111",
long_x WHEN "001"|"010", two_x WHEN "011",
  minus_two_x WHEN "100", minus_x WHEN OTHERS;
product <= acc1(n)&(acc1(n)&acc1) + second_operand;
z <= acc1 & acc0(m+1 DOWNTO 1);
parallel_register: PROCESS(clk)
BEGIN
  IF clk'EVENT and clk = '1' THEN
    IF load = '1' THEN acc1 <= u;
    ELSIF update = '1' THEN acc1 <= product(n+2 DOWNTO 2);
    END IF;
  END IF;
END PROCESS;
shift_register: PROCESS(clk)
BEGIN
  IF clk'EVENT and clk = '1' THEN
    IF load = '1' THEN acc0 <= y&'0';
    ELSIF update = '1' THEN
      acc0 <= product(1 DOWNTO 0)&acc0(m+1 DOWNTO 2);
    END IF;
  END IF;
END PROCESS;
```

The complete circuit also includes an $(m + 1)/2$-state counter and a control unit. A complete generic model *Booth2_sequential_multiplier.vhd* is available at the Authors' web page.

8.5 Constant Multipliers

Given an n-bit constant natural c and an m-bit natural y, the computation of $c \cdot y$ can be performed with any n-bit by m-bit multiplier whose first operand is connected to the constant value c. Then, the synthesis tool will eliminate useless components. In the case of FPGA implementations, an alternative method is to store the constant c within the LUTs.

Assume that the technology at hand includes k-input LUTs. The basic component is a circuit that computes $w = c \cdot b$, where b is a k-bit natural. The maximum value of w is

$$(2^n - 1)\left(2^k - 1\right) = 2^{n+k} - 2^k - 2^n + 1,$$

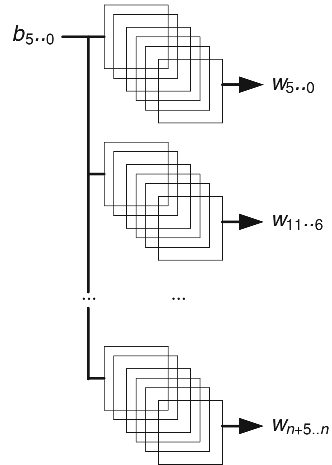

Fig. 8.18 LUT implementation of a *k*-bit by *n*-bit constant multiplier

so w is an $(n + k)$-bit number. The circuit is shown in Fig. 8.18, with $k = 6$. It is made up of $n + 6$ LUT-6, each of them being programmed in such a way that

$$w_{6j+5\ldots6j}(b) = [c_1 \cdot b]_{6j+5\ldots6j}.$$

Its computation time is equal to T_{LUT6}.
The following VHDL model describes the circuit of Fig. 8.18.

```
main_iteration: FOR i IN 0 TO n+5 GENERATE
  LUT_instantiation: lut6
  GENERIC MAP (truth_vector => LUT_definition(c)(i))
  PORT MAP(a => b, b => w(i));
END GENERATE;
```

The function *LUT_definition* defines the LUT contents.

```
TYPE vectors IS ARRAY (0 TO n+5)
  OF STD_LOGIC_VECTOR(0 TO 63);
FUNCTION LUT_definition(c: NATURAL) RETURN vectors IS
  VARIABLE zz: NATURAL;
  VARIABLE zzz: STD_LOGIC_VECTOR(n+5 DOWNTO 0);
  VARIABLE truth_vector: vectors;
BEGIN
  FOR i IN 0 to 63 LOOP
    zz := c*i;
    zzz := CONV_STD_LOGIC_VECTOR(zz, n+6);
      FOR j IN 0 TO n+5 LOOP
        truth_vector(j)(i) := zzz(j);
      END LOOP;
    END LOOP;
   RETURN truth_vector;
 END LUT_definition;
```

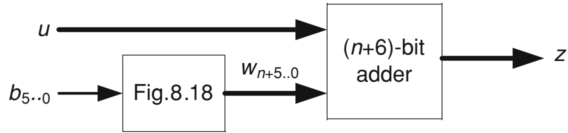

Fig. 8.19 Computation of $w = c \cdot b + u$

The circuit of Fig. 8.18 can be used as a component for generating constant multipliers. As an example, a sequential n-bit by m-bit constant multiplier is synthesized. First define a component similar to that of Fig. 8.2, with x constant. It computes $z = c \cdot b + u$, where c is an n-bit constant natural, b a k-bit natural, and u an n-bit natural. The maximum value of z is

$$(2^n - 1)\left(2^k - 1\right) + 2^n - 1 = 2^{n+k} - 2^k,$$

so it is an $(n + k)$-bit number. It consists of a k-bit by n-bit multiplier (Fig. 8.18) and an $(n + k)$-bit adder (Fig. 8.19).

Finally, the circuit of Fig. 8.19 can be used to generate a radix-2^k shift and add multiplier that computes $z = c \cdot y + u$, where c is an n-bit constant natural, y an m-bit natural, and u an n-bit natural. The maximum value of z is

$$(2^n - 1)(2^m - 1) + 2^n - 1 = 2^{n+m} - 2^m,$$

so z is an $(n + m)$-bit number. Assume that the radix-2^k representation of y is $Y_{m/k-1}\ Y_{m/k-2\ldots}\ Y_0$, where each Y_i is a k-bit number. The circuit implements the following set of equations:

$$\begin{aligned} z^{(0)} &= (u + c \cdot Y_0)/2^k, \\ z^{(1)} &= \left(z^{(0)} + c \cdot Y_1\right)/2^k, \\ z^{(2)} &= \left(z^{1)} + c \cdot Y_2\right)/2^k, \\ &\ldots \\ z^{(m/k-1)} &= \left(z^{(m/k-2)} + c \cdot Y_{m/k-1}\right)/2^k. \end{aligned} \tag{8.19}$$

Thus,

$$z^{(m/k-1)} \cdot \left(2^k\right)^{m/k} = u + c \cdot Y_0 + c \cdot Y_1 \cdot 2^k + \ldots + c \cdot Y_{m/k-1} \cdot \left(2^k\right)^{m/k-1},$$

that is to say

$$z^{(m/k-1)} \cdot 2^m = c \cdot y + u.$$

The circuit is shown in Fig. 8.20.

The computation time is approximately equal to

$$T \cong (m/k) \cdot (T_{LUT-k} + T_{adder}(n + k).$$

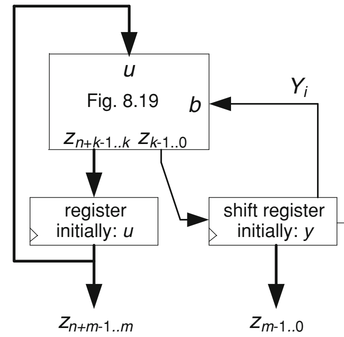

Fig. 8.20 *n*-bit by *m*-bit constant multiplier

The following VHDL model describes the circuit of Fig. 8.20 ($k = 6$).

```
main_component: digit_by_constant_multiplier
  GENERIC MAP(n => n, c => c)
  PORT MAP(b => acc_0(5 DOWNTO 0), u => acc_1, z => product);
z <= acc_1 & acc_0;
parallel_register: PROCESS(clk)
BEGIN
  IF clk'EVENT and clk = '1' THEN
    IF load = '1' THEN acc_1 <= u;
    ELSIF update = '1' THEN acc_1 <= product(n+5 DOWNTO 6);
    END IF;
  END IF;
END PROCESS;
shift_register: PROCESS(clk)
BEGIN
  IF clk'EVENT and clk = '1' THEN
    IF load = '1' THEN acc_0 <= y;
    ELSIF update = '1' THEN
      acc_0 <= product(5 DOWNTO 0)& acc_0(m-1 DOWNTO 6);
    END IF;
  END IF;
END PROCESS;
```

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

The synthesis of constant multipliers for integers is left as an exercise.

Table 8.1 Combinational multiplier

m	*n*	*LUTS*	*Delay*
8	8	96	13.29
16	16	384	28.26
32	16	771	36.91
32	32	1536	57.46
64	32	3073	74.12
64	64	6181	119.33

Table 8.2 Carry-save combinational multiplier

m	*n*	*LUTS*	*Delay*
8	8	102	8.05
16	16	399	15.42
32	16	788	16.99
32	32	1580	29.50
64	32	3165	32.08
64	64	6354	60.90

8.6 FPGA Implementations

Several multipliers have been implemented within a Virtex 5-2 device. Those devices include Digital Signal Processing (DSP) slices that efficiently perform multiplications (25 bits by 18 bits), additions and accumulations. Apart from multiplier implementations based on LUTs and FFs, more efficient implementations, taking advantage of the availability of DSP slices, are also reported. As before, the times are expressed in *ns* and the costs in numbers of Look Up Tables (LUTs), flip-flops (FFs) and DSP slices. All VHDL models as well as several test benches are available at the Authors' web page.

8.6.1 Combinational Multipliers

The circuit is shown in Fig. 8.3. The synthesis results for several numbers n and m of bits are given in Table 8.1.

A faster implementation is obtained by using the carry-save method (Fig. 8.4; Table 8.2).

If multipliers based on the cell of Fig. 8.8b are considered, more efficient circuits can be generated. It is the "by default" option of the synthesizer (Table 8.3).

Finally, if DSP slices are used, better implementations are obtained (Table 8.4).

Table 8.3 Optimized combinational multiplier

n	*m*	*LUTs*	*Delay*
8	8	113	5.343
16	16	435	6.897
32	16	835	7.281
32	32	1668	7.901
64	64	6460	11.41
64	32	3236	9.535
32	64	3236	9.535

Table 8.4 Combinational multiplier with DSP slices

n	*m*	*LUTs*	*DSPs*	*Delay*
8	8	0	2	4.926
16	16	0	2	4.926
32	16	77	2	6.773
32	32	93	4	9.866
64	64	346	12	12.86
64	32	211	6	11.76
32	64	211	6	11.76

Table 8.5 Radix-2^k parallel multipliers

m	*n*	*k*	$m \cdot k$	$n \cdot k$	*LUTs*	*Delay*
2	2	8	16	16	452	10.23
4	4	4	16	16	448	17.40
2	2	16	32	32	1740	12.11
4	4	8	32	32	1808	20.29
4	2	16	64	32	3480	15.96
4	4	16	64	64	6960	22.91

8.6.2 Radix-2^k Parallel Multipliers

Several $m \cdot k$ bits by $n \cdot k$ bits multipliers (Sect. 8.2.4) have been implemented (Table 8.5).

A faster implementation is obtained by using the carry-save method (Table 8.6).

The same circuits have been implemented with DSP slices. The implementation results are given in Tables 8.7, 8.8

8.6.3 Sequential Multipliers

Several shift and add multipliers have been implemented. The implementation results are given in Tables 8.9, 8.10. Both the clock period T_{clk} and the total delay $(m \cdot T_{clk})$ are given.

Table 8.6 Carry-save radix-2^k parallel multipliers

m	*n*	*k*	*m · k*	*n · k*	*LUTs*	*Delay*
2	2	8	16	16	461	8.48
4	4	4	16	16	457	10.09
2	2	16	32	32	1757	10.36
4	4	8	32	32	3501	11.10
4	2	16	64	32	1821	12.32
4	4	16	64	64	6981	14.93

Table 8.7 Radix-2^k parallel multipliers with DSPs

m	n	k	m · k	n · k	DSPs	LUTs	Delay
2	2	8	16	16	8	0	12.58
4	4	4	16	16	32	0	27.89
2	2	16	32	32	8	0	12.58
4	4	8	32	32	32	0	27.89
4	2	16	64	32	16	0	16.70
4	4	16	64	64	32	0	27.89

Table 8.8 Carry-save radix-2^k parallel multipliers with DSPs

m	*n*	*k*	*m · k*	*n · k*	*DSPs*	*LUTs*	*Delay*
2	2	8	16	16	8	15	9.90
4	4	4	16	16	32	15	17.00
2	2	16	32	32	8	31	10.26
4	4	8	32	32	32	31	17.36
4	2	16	64	32	16	63	11.07
4	4	16	64	64	32	63	18.09

Table 8.9 Shift and add multipliers

n	*m*	*FFs*	*LUTs*	*Period*	*Total time*
8	8	29	43	2.87	23.0
8	16	46	61	2.87	45.9
16	8	38	72	4.19	33.5
16	16	55	90	4.19	67.0
32	16	71	112	7.50	120.0
32	32	104	161	7.50	240.0
64	32	136	203	15.55	497.6
64	64	201	306	15.55	995.2

8.6.4 Combinational Multipliers for Integers

A carry-save multiplier for integers is shown in Fig. 8.12. The synthesis results for several numbers n and m of bits are given in Table 8.11.

Table 8.10 Sequential carry-save multipliers

n	*m*	*FFs*	*LUTs*	*Period*	*Total time*
8	8	29	43	1.87	15.0
16	8	47	64	1.88	15.0
16	16	56	74	1.92	30.7
32	16	88	122	1.93	30.9
32	32	106	139	1.84	58.9
64	32	170	235	1.84	58.9
64	64	203	268	1.84	117.8

Table 8.11 Carry-save mod 2^{n+m+1} multipliers

n	*m*	*LUTs*	*Delay*
8	8	179	12.49
8	16	420	18.00
16	8	421	20.41
16	16	677	25.86
32	16	1662	42.95
32	32	2488	55.69

Table 8.12 Modified shift and add algorithm

n	m	LUTs	Delay
8	8	122	15.90
8	16	230	27.51
16	8	231	20.20
16	*16*	*435*	*31.81*
32	16	844	39.96
32	32	1635	62.91

Table 8.13 Multipliers with post correction

n	*m*	*LUTs*	*Delay*
8	8	106	14.18
8	16	209	24.60
16	8	204	18.91
16	16	407	30.60
32	16	794	39.43
32	32	1586	62.91

Another option is the modified shift and add algorithm of Sect. 8.4.2 (Fig. 8.15; Table 8.12).

In Table 8.13, examples of post correction implementations are reported.

As a last option, several Booth multipliers have been implemented (Table 8.14).

Table 8.14 Combinational Booth multipliers

n	*m*	*LUTs*	*Delay*
8	8	188	13.49
8	16	356	25.12
16	8	332	13.68
16	16	628	25.31
32	16	1172	25.67
32	32	2276	49.09

Table 8.15 Sequential radix-4 Booth multipliers

n	*m*	*FFs*	*LUTs*	*Period*	*Total time*
8	9	25	58	2.90	26.1
8	17	34	68	2.90	49.3
16	9	33	125	3.12	28.1
16	17	42	135	3.12	53.0
32	17	58	231	3.48	59.2
32	33	75	248	3.48	114.8
64	33	107	440	4.22	139.1
64	65	140	473	4.22	274.0

8.6.5 Sequential Multipliers for Integers

Several radix-4 Booth multipliers have been implemented (Fig. 8.17). Both the clock period T_{clk} and the total delay ($m \cdot T_{clk}$) are given (Table 8.15).

8.7 Exercises

1. Generate the VHDL model of a mixed-radix parallel multiplier (Sect. 8.2.4).
2. Synthesize a 2n-bit by 2n-bit parallel multiplier using n-bit by n-bit multipliers as building blocks.
3. Modify the VHDL model *sequential_CSA_multiplier.vhd* so that the *done* flag is raised when the final result is available (Comment 8.2).
4. Generate the VHDL model of a carry-save multiplier with post correction (Sect. 8.4.3).
5. Synthesize a sequential multiplier based on Algorithm 8.3.
6. Synthesize a parallel constant multiplier (Sect. 8.5).
7. Generate models of constant multipliers for integers.
8. Synthesize a constant multiplier that computes $z = c_1 \cdot y_1 + c_2 \cdot y_2 + \ldots + c_s \cdot y_s + u$.

References

1. Baugh CR, Wooley BA (1973) A two's complement parallel array multiplication algorithm. IEEE Trans Comput C 31:1045–1047
2. Booth AD (1951) A signed binary multiplication technique. Q J Mech Appl Mech 4:126–140
3. Dadda L (1965) Some schemes for parallel multipliers. Alta Frequenza 34:349–356
4. Wallace CS (1964) A suggestion for fast multipliers. IEEE Trans Electron Comput EC-13:14–17