Chapter 3
Special Topics of Data Path Synthesis

Several important implementation techniques are presented in this chapter. The first one is pipelining, a very commonly used method in systems that process great volumes of data. Self-timing is the topic of the second section. To some extent it can be considered as an extension of the pipelining concept and is especially attractive in the case of very big circuits. The third section is an introduction to a circuit level, or even algorithm level, transformation known as "loop unrolling". It permits the exploration of different cost—performance tradeoffs, from combinational iterative circuits to completely sequential circuits. Finally, the last section tackles the problem of reducing the number of connection resources.

3.1 Pipeline

A very useful implementation technique, especially for signal processing circuits, is pipelining [1, 2]. It consists of inserting additional registers so that the maximum clock frequency and input data throughput are increased. Furthermore, in the case of FPGA implementations, the insertion of pipeline registers has a positive effect on the power consumption.

3.1.1 Introductory Example

Consider the introductory example of Sect. 2.3.1. The set of Eq. (2.10) can be implemented by a combinational circuit (option 1. of Sect. 2.3.1) made up of four carry-save adders, with a computation time equal to $3 \cdot T_{FA}$. That means that the minimum clock period of a synchronous circuit including this 7-to-3 counter should be greater than $3 \cdot T_{FA}$, and that the introduction interval between successive data inputs should also be greater than $3 \cdot T_{FA}$. The corresponding circuit is shown

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_3,

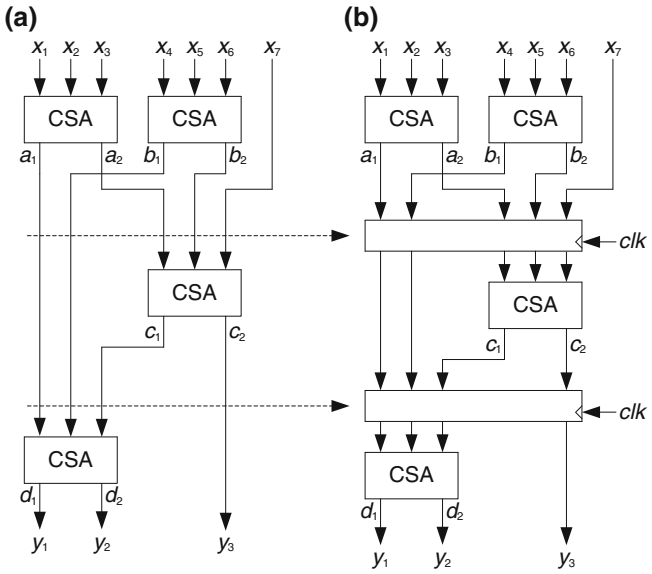


Fig. 3.1 **a** Combinational circuit. **b** Pipelined circuit

in Fig. 3.1a. As previously commented, this is probably a bad circuit because its cost is high and its maximum clock frequency is low.

Consider now the circuit of Fig. 3.1b in which registers have been inserted in such a way that operations scheduled in successive cycles, according to the ASAP schedule of Fig. 2.9a, are separated by a register. The circuit still includes four carry-save adders, but the minimum clock period of a synchronous circuit including this counter must be greater than T_{FA}, plus the set-up and hold times of the registers, instead of $3 \cdot T_{FA}$. Furthermore, the minimum data introduction interval is now equal to T_{clk}: as soon as a_1, a_2, b_1 and b_2 have been computed, their values are stored within the corresponding output register, and a new computation, with other input data, can start; at the same time, new computations of c_1 and c_2, and of d_1 and d_2 can also start. Thus, at time t, three operations are executed in parallel:

```
(a1(t), a2(t), b1(t), b2(t)) :=
(CSA(x1(t), x2(t), x3(t)), CSA(x4(t), x5(t), x6(t)));

(c1(t-1), y3(t-1)) := (CSA(a2(t-1), b2(t-1), x7(t-1)));

(y1(t-2), y2(t-2)) := (CSA(a1(t-2), b1(t-2), c1(t-2)));
```

To summarize, assuming that the set-up and hold times are negligible,

$$T_{clk} > T_{FA},\ latency = 3 \cdot T_{clk},\ r = T_{clk},$$

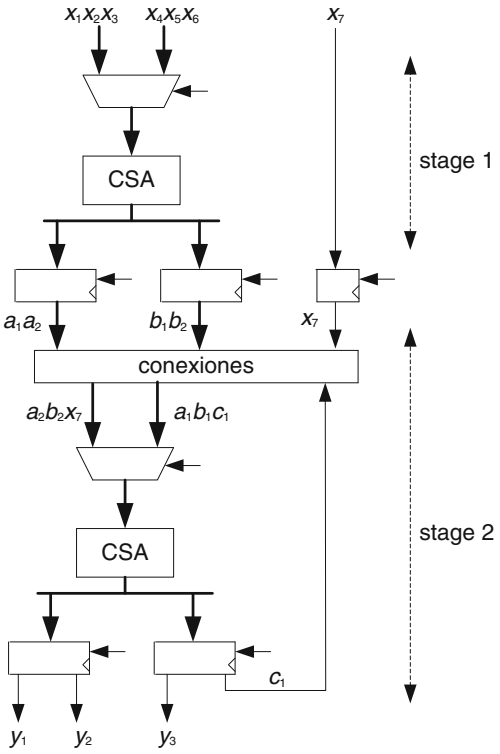


Fig. 3.2 Two-stage two-cycle implementation

where *latency* is the total computation time and r is the minimum data introduction interval.

Another implementation, based on the admissible schedule of Fig. 2.9c is show in Fig. 3.2. In this case the circuit is made up of two stages separated by a pipeline register. Within every stage the operations are executed in two cycles. During the first cycle the following operations are executed

```
stage 1: (a1(t), a2(t)) := CSA(x1(t), x2(t), x3(t));

stage 2 :
  (c1(t-1), y3(t-1)) := (CSA(a2(t-1), b2(t-1), x7(t-1)));
```

and during the second cycle, the following ones are executed

```
stage 1: (b1(t), b2(t)) := CSA(x4(t), x5(t), x6(t)) ;

stage 2 :
  (y1(t-1), y2(t-1)) := (CSA(a1(t-1), b1(t-1), c1(t-1)));
```

The circuit of Fig. 3.2 includes two carry-save adders instead of four, and its timing constraints are the following:

$$T_{clk} > T_{FA} + T_{multiplexor},\ latency = 4 \cdot T_{clk},\ r = 2 \cdot T_{clk}.$$

To summarize, the main parameters of a pipelined circuit are the following.

- *Latency* (also called *delay*, *response time*): total delay between the introduction of a new set of input data and the generation of the corresponding output results. It is equal to $n{\cdot}T_{clk}$ where n is the number of pipeline stages and T_{clk} the clock period.
- *Pipeline rate* (also called *pipeline period*): data introduction interval.
- *Throughput* (also called *speed*, *bandwidth*, *production*): number of input data processed per time unit. For great numbers of processed data, it is the inverse of the pipeline rate r.

Assuming that the combinational delay of stage number i is equal to t_i, and that the register set-up and hold times are negligible, the minimum clock period is the maximum of all t_i's. In the first example (Fig. 3.1a) $t_1 = t_2 = t_3 = T_{FA}$, while in the second example (Fig. 3.2) $t_1 = t_2 = T_{FA} + T_{multiplexor}$.

A very common situation is that of the first example (Fig. 3.1). An initial combinational circuit has a delay equal to C, so that it is able to process $1/C$ input data per time unit. It is divided up into n pipeline stages, all of them with the same delay C/n (balanced stages). Then

$$T_{clk} \cong C/n,\ r = 1/T_{clk} \cong n/C,\ latency = n \cdot T_{clk} \cong C,\ T(m) = n \cdot T_{clk} + (m-1) \cdot T_{clk},$$

where $T(m)$ is the time necessary to process m input data. Thus, the average number of input data processed per time unit is equal to $m/T(m) = m/(n + m - 1){\cdot}T_{clk} \cong mn/(n + m - 1){\cdot}C$. For great values of m, the number of input data processed per time unit is equal to n/C. Thus, with respect to the initial combinational circuit, the throughput has been multiplied by n.

The actual speedup factor is smaller if the connection and register delays are taken into account. Assume that those additional delays are equal to δ time units. Then, the minimum clock period is equal to $T_{clk} = C/n + \delta$, $r = 1/T_{clk} = n/(C + n\delta)$, $latency = n{\cdot}T_{clk} = C + n\delta$, $T(m) = (n + m{-}1){\cdot}T_{clk} = (n + m{-}1){\cdot}(C/n + \delta) \cong m{\cdot}(C/n + \delta)$, $m/T(m) \cong 1/(C/n + \delta) = n/(C + n\delta)$. Hence, the throughput increase factor is equal to $n{\cdot}C/(C + n\delta) = n/(1 + \alpha)$ where $\alpha = n\delta/C$.

3.1.2 Segmentation

Given a computation scheme and its precedence graph G, a *segmentation* of G is an ordered partition $\{S_1, S_2, \ldots, S_k\}$ of G. The segmentation is *admissible* if it respects the precedence relationship. This means that if there is an arc from $op_J \in S_i$ to op_M then either op_M belongs to the same segment S_i or it belongs to a different segment S_j with $j > i$. Two examples are shown in Fig. 3.3 in which the segments are separated by dotted lines.

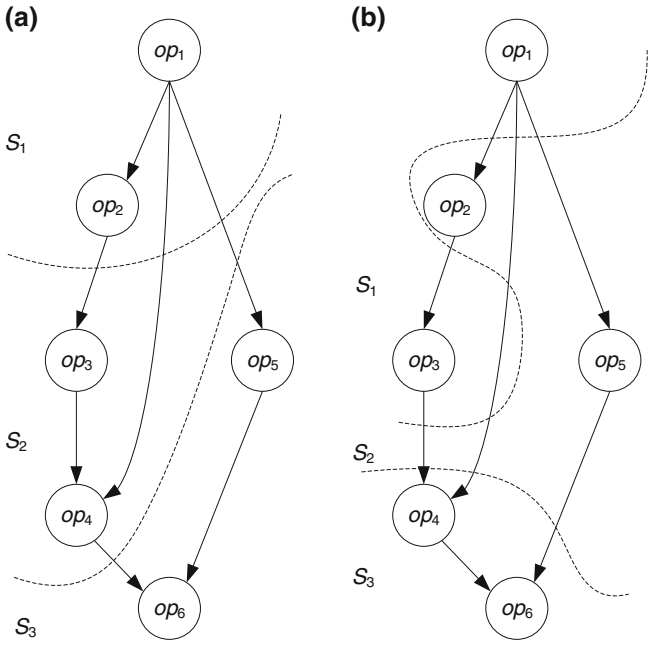


Fig. 3.3 **a** Admissible segmentation. **b** Non-admissible segmentation

The segmentation of Fig. 3.3a, that is $S_1 = \{op_1,\ op_2\}$, $S_2 = \{op_3,\ op_4\}$, $S_3 = \{op_5,\ op_6\}$, is admissible, while that of Fig. 3.3b, that is $S_1 = \{op_1,\ op_3\}$, $S_2 = \{op_2,\ op_5\}$, $S_3 = \{op_4,\ op_6\}$, is not (there is an arc $op_2 \rightarrow op_3$ from S_2 to S_1).

Once an admissible partition has been defined, every segment can be synthesized separately, using the same methods as before (scheduling, resource assignment). In order to assemble the complete circuit, additional registers are inserted: if an arc of the precedence graph crosses the line that separates segments i and $i + 1$, then a register must be inserted; it will store the output data generated by segment i that in turn are input data to segment $i + 1$. As an example, the structure of the circuit corresponding to Fig. 3.3a is shown in Fig. 3.4.

Assume that C_i and T_i are the cost and computation time of segment i. The cost and latency of the complete circuit are

$$C = C_1 + C_2 + \cdots + C_k + C_{registers} \text{ and } T = T_1 + T_2 + \cdots + T_k + T_{registers}$$

where $C_{registers}$ represents the total cost of the pipeline registers and $T_{registers}$ the additional delay they introduce. The time interval δ between successive data inputs is

$$\delta = max\{T_1, T_2, \ldots, T_k\} + T_{SU} + T_P \cong max\{T_1, T_2, \ldots, T_k\}$$

where T_{SU} and T_P are the set-up and propagation times of the used flip-flops.

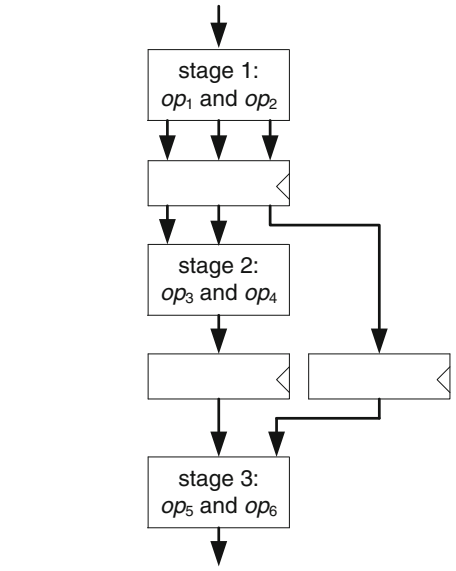


Fig. 3.4 Pipelined circuit

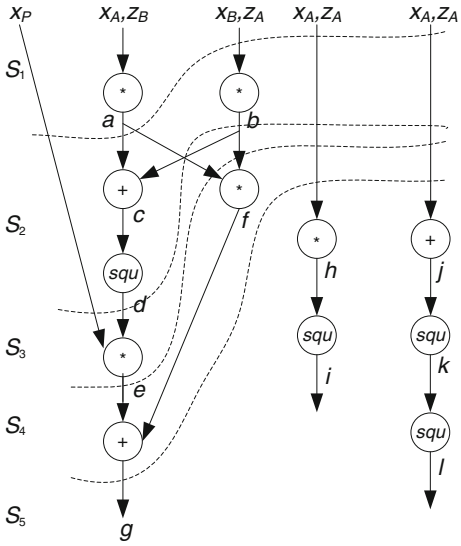


Fig. 3.5 An admissible segmentation

A second, more realistic, example is now presented. It corresponds to part of an Elliptic Curve Cryptography algorithm (Example 2.1).

Example 3.1
An admissible segmentation of the precedence graph of Fig. 2.10 is shown in Fig. 3.5.

The operations corresponding to each segment are the following.

- Segment 1:

$a = x_A \cdot z_B;$

- Segment 2:

$b = x_B \cdot z_A,$
$c = a + b,$
$d = c^2;$

- Segment 3:

$e = x_P \cdot d;$

- Segment 4:

$f = a \cdot b,$
$g = e + f;$

- Segment 5:

$h = x_A \cdot z_A,$
$i = h^2,$
$j = x_A + z_A,$
$k = j^2,$
$l = k^2;$

So, every segment includes a product over a finite field plus some additional 1-cycle operations (finite field additions and squares) in segments 2, 4 and 5. The corresponding pipelined circuit, in which it is assumed that the output results are g, d, l and i, is shown in Fig. 3.6.

A finite field product is a complex operation whose maximum computation time t_m, expressed in number of clock cycles, is much >1. Thus, the latency T and the time interval δ between successive data inputs of the complete circuit are

$$T \cong 5t_m \text{ and } \delta \cong t_m.$$

The cost of the circuit is very high. It includes five multipliers, three adders, four squarers and four pipeline registers. Furthermore, if used within the scalar product circuit of Sect. 2.5, the fact that the time interval between successive data inputs has been reduced ($\delta \cong t_m$) does not reduce the execution time of Algorithm 2.4 as the input variables x_A, z_A, x_B and z_B are updated at the end of every main loop execution.

As regards the control of the pipeline, several options can be considered. A simple one is to previously calculate the maximum multiplier computation time t_m and to choose $\delta > t_m + 2$ (computation time of segment 2). The control unit updates the pipeline registers and sends a *start* pulse to all multipliers every δ cycles. In the following VHDL process, *time_out* is used to enable the pipeline register clock every *delta* cycles and *sync* is a synchronization procedure (Sect. 2.5):

```
control_unit: PROCESS
BEGIN
  LOOP
    time_out <= '0'; start <= '0'; sync;
    IF reset = '1' THEN EXIT; END IF;
    start <= '1'; sync;
    FOR i IN 1 TO delta-3 LOOP
      sync;
    END LOOP;
    time_out <= '1'; sync;
  END LOOP;
END PROCESS;
```

In order to describe the complete circuit, five multipliers, three adders and four squarers are instantiated, and every pipeline register, for example the segment 1 output register, can be described as follows:

```
segment1: PROCESS
BEGIN
  WAIT UNTIL time_out = '1';
  xP1 <= xP; a1 <= a; xB1 <= xB; zA1 <= zA; xA1 <= xA;
END PROCESS;
```

A complete VHDL model *pipeline_DF2.vhd* is available at the Authors' web page.

Another interesting option could be a self-timed circuit (Example 3.4).

3.1.3 Combinational to Pipelined Transformation

A very common situation is the following: a combinational circuit made up of relatively small blocks, all of them with nearly equal delays, has been designed, and its computation time is equal to T seconds. If this combinational circuit is used as a computation resource of a synchronous circuit, then the clock cycle must be greater than T, and in some cases it could be an over extended time (a too low frequency). In order to increase the clock frequency, as well as to reduce the minimum time interval between successive data inputs, the solution is pipelining. As the combinational version already exists, it is no longer necessary to use the general method of Sect. 3.1.2. The combinational circuit can be directly segmented into stages.

Consider a generic example. The iterative circuit of Fig. 3.7 is made up twelve identical blocks, each of them with a maximum delay of t_{cell} seconds. The maximum propagation time of every connection is equal to $t_{connection}$ seconds. Thus, the computation time of this circuit is equal to $T = 6t_{cell} + 7t_{connection}$ (input and output connections included). Assume that this circuit is part of a synchronous

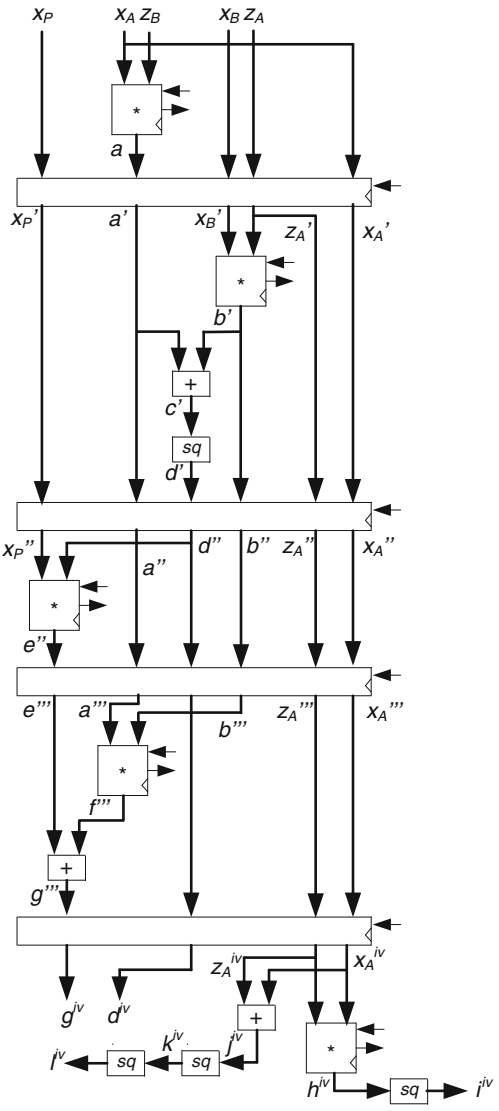


Fig. 3.6 Pipelined circuit ($\forall$ s: $s' = s(t-1)$, $s'' = s(t-2)$, $s''' = s(t-3)$, $s^{iv} = s(t-4)$)

circuit and that all inputs come from register outputs and all outputs go to register inputs. Then the minimum clock cycle T_{CLK} is defined by the following relation:

$$T_{CLK} > 6t_{cell} + 7t_{connection} + t_{SU} + t_P, \tag{3.1}$$

where t_{SU} and t_P are the minimum set-up and propagation times of the registers (Chap. 6).

If the period defined by condition (3.1) is too long, the circuit must be segmented. A 2-stage segmentation is shown in Fig. 3.8. Registers must be inserted in

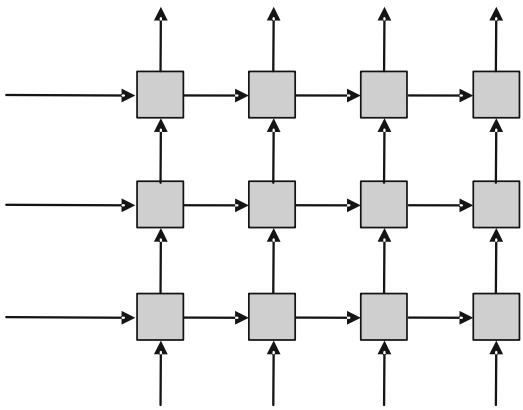


Fig. 3.7 Combinational circuit

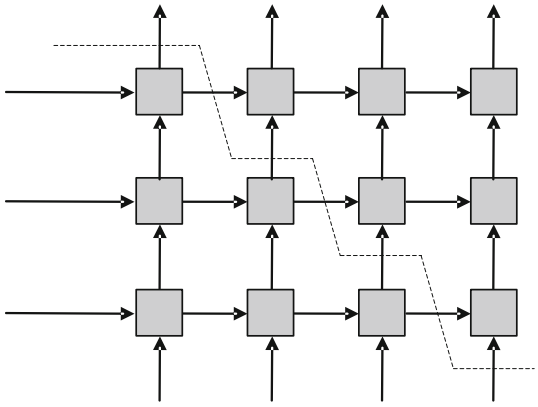


Fig. 3.8 2-stage segmentation

all positions where a connection crosses the dotted line. Thus, seven registers must be added. Assuming that the propagation time of every part of a segmented connection is still equal to $t_{connection}$, the following condition must hold:

$$T_{CLK} > 3t_{cell} + 4t_{connection} + t_{SU} + t_P. \tag{3.2}$$

A 5-stage segmentation is shown in Fig. 3.9. In this case, 32 registers must be added and the following condition must hold:

$$T_{CLK} > t_{cell} + 2t_{connection} + t_{SU} + t_P. \tag{3.3}$$

Consider a practical example.

Example 3.2
Implement a 128-bit adder made up of four 32-bit adders. A combinational implementation is described in Fig. 3.10. The computation time T of the circuit is equal to $4 \cdot T_{adder}$, where T_{adder} is the computation time of a 32-bit adder.

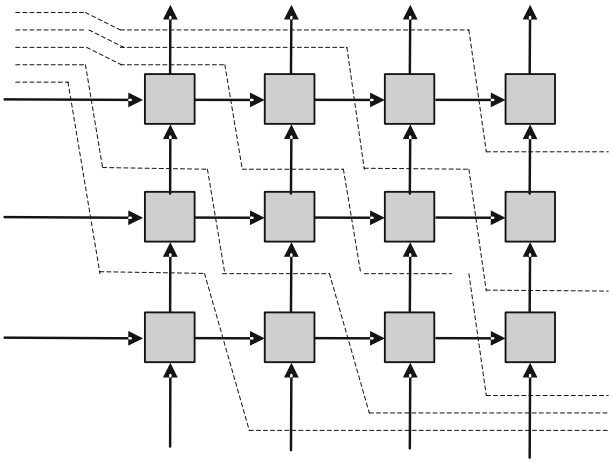


Fig. 3.9 5-stage segmentation

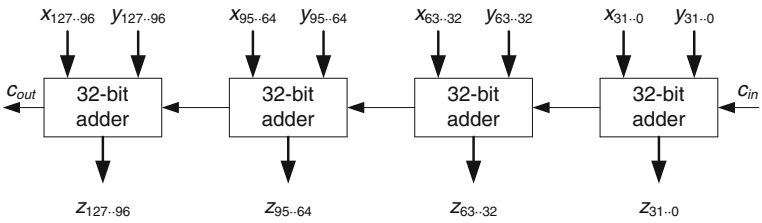


Fig. 3.10 128-bit adder

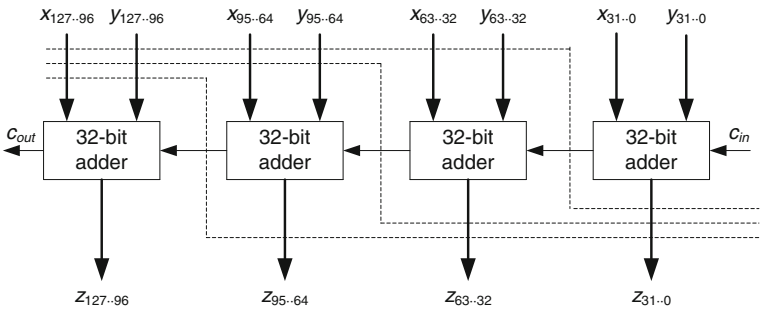


Fig. 3.11 4-stage segmentation

A 4-stage segmentation is shown in Fig. 3.11. Every stage includes one 32-bit adder so that the minimum clock cycle, as well as the minimum time interval between successive data inputs, is equal to T_{adder}. The corresponding circuit is shown in Fig. 3.12. In total, $(7{\cdot}32 + 1) + (6{\cdot}32 + 1) + (5{\cdot}32 + 1) = 579$ additional flip-flops are necessary in order to separate the pipeline stages.

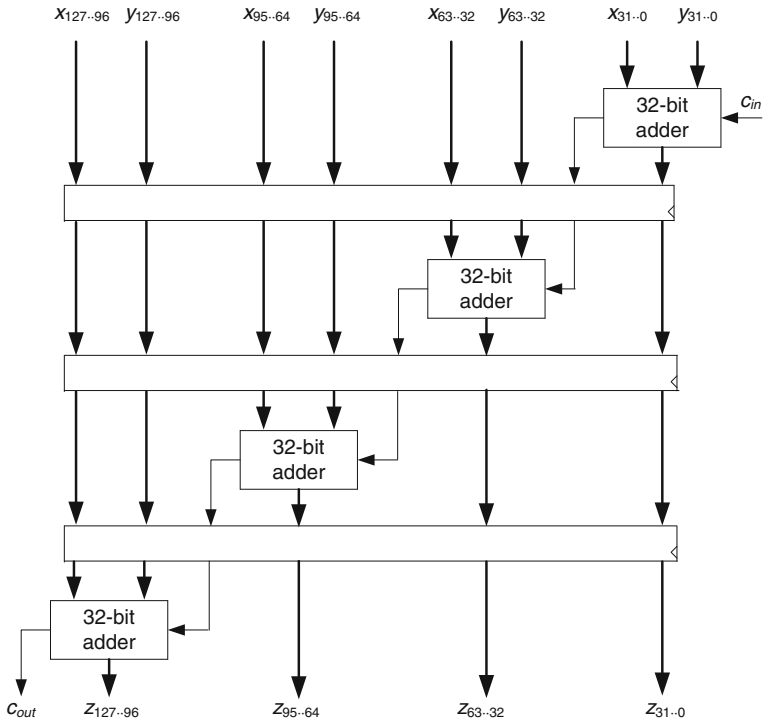


Fig. 3.12 Pipelined 128-bit adder

Comments 3.1

- The extra cost of the pipeline registers could appear to be prohibitive. Nevertheless, the basic cell of a field programmable gate array includes a flip-flop, so that the insertion of pipeline registers does not necessarily increase the total cost, computed in terms of used basic cells. The pipeline registers could consist of flip-flops not used in the non-pipelined version.
- Most FPGA families also permit implementing with LUTs those registers that do not need reset signals. This can be another cost-effective option.
- The insertion of pipeline registers also has a positive effect on the power consumption: the presence of synchronization barriers all along the circuit drastically reduces the number of generated spikes.

3.1.4 Interconnection of Pipelined Components

Assume that several pipelined circuits are used as computational resources for generating a new pipelined circuit. For example, consider a circuit that computes $g = a{\cdot}b$ and $f = (a{\cdot}b + c){\cdot}d$, and uses a 2-stage pipelined multiplier and a 3-stage

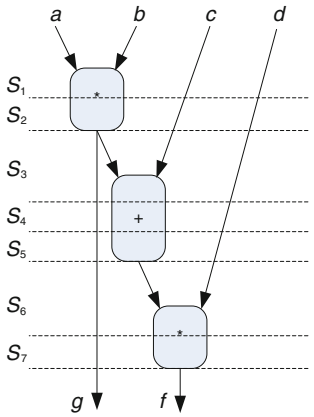


Fig. 3.13 Interconnection of pipelined components: scheduling

pipelined adder, both of them working at the same frequency $1/T_{clk}$ and with the same pipeline rate $r = 1$. The operations can be scheduled as shown in Fig. 3.13. Some inputs and outputs must be delayed: input c must be delayed 2 cycles, input d must be delayed 5 cycles, and output g must be delayed 5 cycles. The corresponding additional registers, which maintain the correct synchronization of the data, are sometimes called *skewing* (c and d) and *deskewing* (g) registers.

An alternative solution, especially in the case of large circuits, is *self-timing*.

As a generic example, consider the pipelined circuit of Fig. 3.14a. To each stage, for example number i, are associated a maximum delay $t_{MAX}(i)$ and an average delay $t_{AV}(i)$. The minimum time interval between successive data inputs is

$$\delta = max\{t_{MAX}(1), t_{MAX}(2), \ldots, t_{MAX}(n)\}, \tag{3.4}$$

and the minimum circuit latency T is

$$T = n \cdot max\{t_{MAX}(1), t_{MAX}(2), \ldots, t_{MAX}(n)\}. \tag{3.5}$$

A self-timed version of the same circuit is shown in Fig. 3.14b. The control is based on a *Request/Acknowledge* handshaking protocol:

- a *req_in* signal to stage 1 is raised by an external circuit; if stage 1 is free, the input data is registered ($ce = 1$), and an *ack_out* signal is issued;
- the *start* signal of stage 1 is raised; after some amount of time, the *done* signal of stage 1 elevates indicating the completion of the computation;
- a *req_out* signal to stage 2 is issued by stage 1; if stage 2 is free, the output of stage 1 is registered and an *ack_out* signal to stage 1 is issued; and so on.

If the probability distribution of the internal data were uniform, inequalities (3.4) and (3.5) would be substituted by the following:

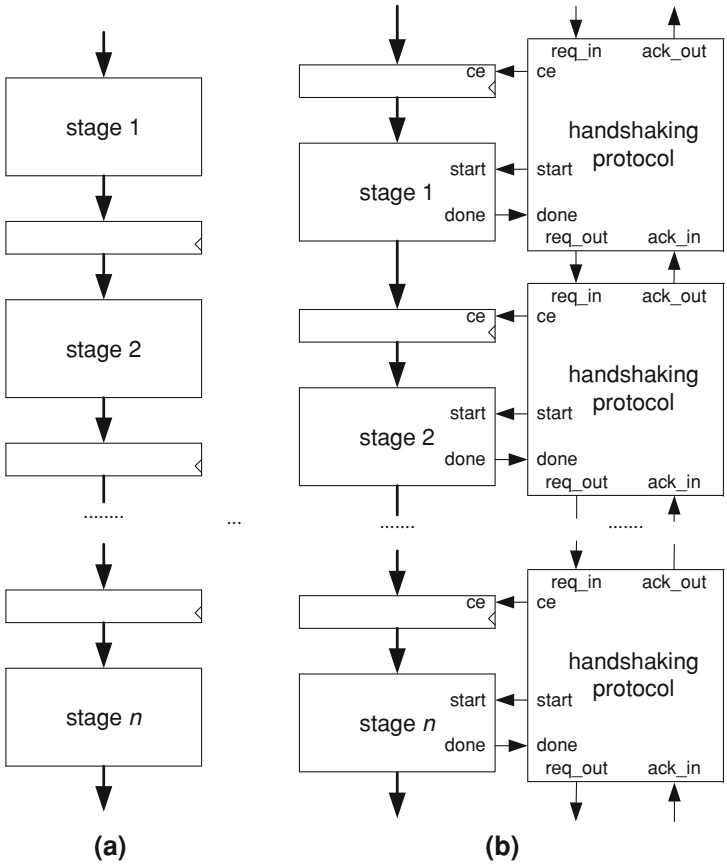


Fig. 3.14 **a** Pipelined circuit. **b** Self-timed pipelined circuit

$$\delta = max\{t_{AV}(1), t_{AV}(2), \ldots, t_{AV}(n)\}, \tag{3.6}$$

$$T = t_{AV}(1) + t_{AV}(2) + \cdots + t_{AV}(n). \tag{3.7}$$

Example 3.3
The following process describes a *handshaking protocol* component. As before, *sync* is a synchronization procedure (Sect. 2.5):

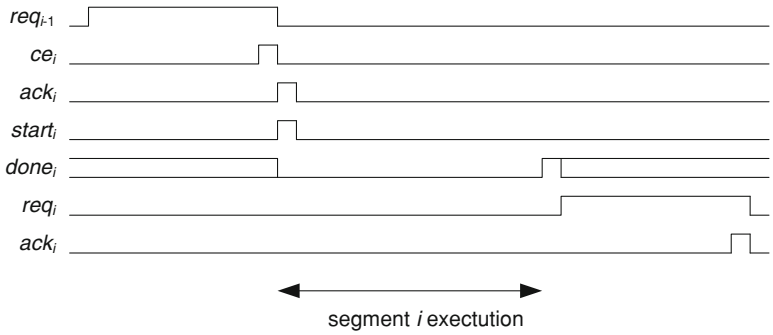


Fig. 3.15 Handshaking protocol

```
PROCESS
BEGIN
  ce <= '0'; start <= '0'; req_out <= '0'; ack_out <= '0';
  sync;
  LOOP
    IF reset = '1' THEN EXIT; END IF;
    IF req_in = '1'THEN
      ce <= '1'; sync;
      ce <= '0'; ack_out <= '1'; sync;
      ack_out <= '0'; start <= '1'; sync;
      start <= '0';
      WAIT UNTIL done = '1';
    req_out <= '1';
    WAIT UNTIL ack_in = '1';
    req_out <= '0'; sync;
  ELSE sync;
  END IF;
END LOOP;
END PROCESS;
```

The corresponding signals are shown in Fig. 3.15.

Example 3.4
Consider a self-timed version of the circuit of Example 3.1 (Fig. 3.6). In stages 1 and 3, the *done* signals are the corresponding *done* outputs of the multipliers. In stages 2, 4 and 5 an additional delay must be added. Stage 2 is shown in Fig. 3.16: a synchronous delay Δ, greater than the sum of the computation times of $c' = a' + b'$ and $d' = (c')^2$, has been added. A complete VHDL model *pipeline_ST.vhd* is available at the Authors' web page.

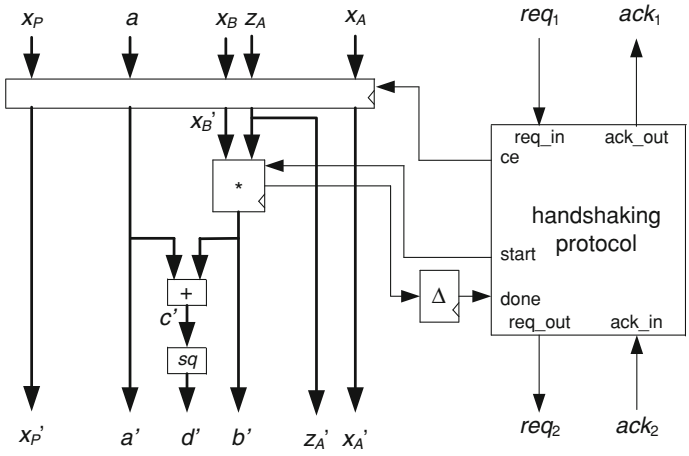


Fig. 3.16 Self-timed pipelined circuit: stage 2

Table 3.1 Redundant encoding

s	s_1	s_0
Reset or *in transition*	0	0
0	0	1
1	1	0

With regards to the generation of the *done* signal in the case of combinational components, an interesting method consists of using a redundant encoding of the binary signals (Sect. 10.4 of [3]: every signal s is represented by a pair (s_1, s_0) according to the definition of Table 3.1.

The circuit will be designed in such a way that during the initialization (*reset*), and as long as the value of s has not yet been computed, (s_1, s_0) = (0, 0). Once the value of s is known $s_1 = s$ and $s_0 = not(s)$.

Assume that the circuit includes n signals s_1, s_2,…, s_n. Every signal s_i is substituted by a pair (s_{i1}, s_{i0}). Then the *done* flag is computed as follows:

$$done = (s_{11} + s_{10}) \cdot (s_{21} + s_{20}) \ldots (s_{n1} + s_{n0}).$$

During the initialization (*reset*) and as long as at least one of the signals is *in transition*, the corresponding pair is equal to (0, 0), so that *done* = 0. The *done* flag will be raised only when all signals have a stable value.

In the following example, only the signals belonging to the critical path of the circuit are encoded.

Example 3.5

Generate an n-bit ripple-carry adder (Chap. 7) with end of computation detection. For this purpose, all signals belonging to the carry chain, that is c_0, c_1, c_2,…, c_{n-1}, are represented by the form (c_0, cb_0), (c_1, cb_1), (c_2, cb_2),…, (c_{n-1}, cb_{n-1}). During the initialization, all c_i and cb_i are equal to 0. When *reset* goes down,

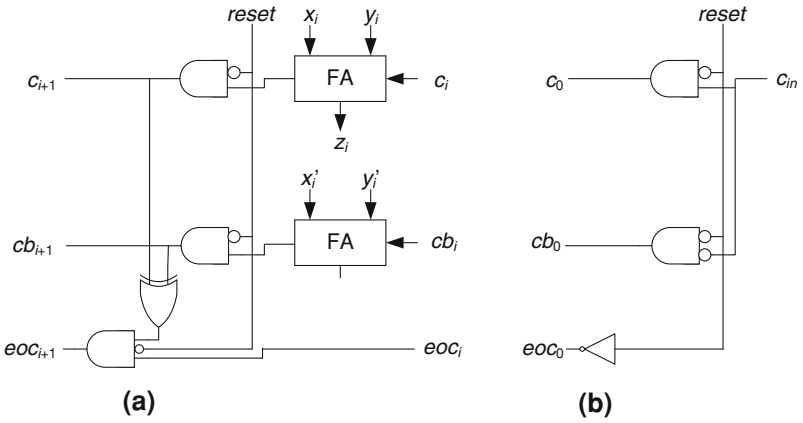


Fig. 3.17 **a** Iterative cell. **b** Initial cell

$$c_0 = c_{in}, \quad cb_0 = \overline{c_{in}},$$

$$c_{i+1} = x_i \cdot y_i + x_i \cdot c_i + y_i \cdot c_i, \quad cb_{i+1} = \overline{x_i} \cdot \overline{y_i} + \overline{x_i} \cdot cb_i + \overline{y_i} \cdot cb_i,$$
$$\forall i \in \{0, 1, \ldots n-1\}.$$

The end of computation is detected when

$$cb_i = \overline{c_i}, \quad \forall i \in \{0, 1, \ldots, n\}.$$

The following VHDL process describes the circuit:

```
  adder: PROCESS(reset, cy, cyb, eoc)
  BEGIN
  IF reset = '1' THEN cy <= (OTHERS => '0');
    cyb <= (OTHERS => '0'); eoc <= (OTHERS => '0');
  ELSE
    cy(0) <= c_in; cyb(0) <= NOT(c_in); eoc(0) <= '1';
    FOR i IN 0 TO n-1 LOOP
      cy(i+1) <= (x(i) AND y(i)) OR
        (x(i) AND cy(i)) OR (y(i) AND cy(i));
      cyb(i+1) <= (NOT(x(i)) AND NOT(y(i))) OR
        (NOT(x(i)) AND cyb(i)) OR (NOT(y(i)) AND cyb(i));
      eoc(i+1) <= eoc(i) AND (cy(i+1) XOR cyb(i+1));
    END LOOP;
  END IF;
END PROCESS;
z <= x XOR y XOR cy(n-1 DOWNTO 0);
c_out <= cy(n);
done <= eoc(n);
```

The corresponding circuit is shown in Fig. 3.17.

A complete model *adder_ST2.vhd* is available at the Authors' web page. In order to observe the carry chain delay, *after* clauses have been added (1 ns for

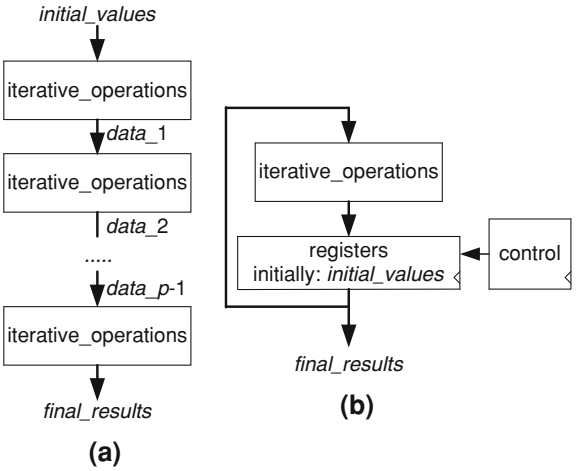


Fig. 3.18 Iterative algorithm implementation

c_{i+1} and cb_{i+1}, 0.2 ns for eoc_{i+1}). For synthesis purpose, they must be deleted.

3.2 Loop Unrolling and Digit-Serial Processing

Consider an iterative algorithm whose main operation consists of executing a procedure *iterative_operations*(*a*: *in*; *b*: *out*):

```
data0 := initial_values;
for i in 1 .. p loop
  iterative_operations(datai-1, datai);
end loop;
final_results := datap;
```

Assuming that a combinational component that implements *iterative_operations* has been previously developed, two straightforward implementations of the algorithm are shown in Fig. 3.18. The first one is an iterative combinational circuit whose cost and delay are

$$C_{combinational} = p \cdot C_{component},\ T_{combinational} < p \cdot T_{component}.$$

The second one is a sequential circuit whose main characteristics are

$$C_{sequential} = C_{component} + C_{registers} + C_{control},\ T_{clk} > T_{component},\ T_{sequentiall} = p \cdot T_{clk}.$$

An alternative option consists of a partial *unroll* of the "for" loop [1, 2]. Assume that $p = k{\cdot}s$. Then, s successive iteration steps are executed at each clock cycle.

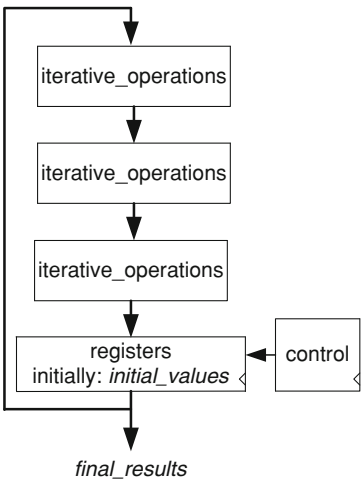


Fig. 3.19 Unrolled loop implementation ($s = 3$)

An example, with $s = 3$, is shown in Fig. 3.19. Obviously, the clock cycle, say T_{clk}', must be longer than in the sequential implementation of Fig. 3.18b (T_{clk}). Nevertheless, it will be generally shorter than $s{\cdot}T_{clk}$. On the one hand, the critical path length of s serially connected combinational circuits is generally shorter than the critical path length of a single circuit, multiplied by s. For example, the delay of an n-bit ripple-carry adder is proportional to n; nevertheless the delay of two serially connected adders, that compute $(a + b) + c$, is proportional to $n + 1$, and not to $2n$. On the other hand, the register delays are divided by s. Furthermore, when interconnecting several circuits, some additional logical simplifications can be performed by the synthesis tool. So,

$$\begin{aligned} C_{unrolled} &= s \cdot C_{component} + C_{registers} + C_{control},\ T_{unrolled} \\ &= (p/s) \cdot T'_{clk},\ \text{where } T'_{clk} < s \cdot T_{clk}. \end{aligned}$$

Example 3.6
Given two naturals x and y, with $x < y$, the following *restoring division algorithm* computes two fractional numbers $q = 0.q_{-1}\ q_{-2}\ \ldots\ q_{-p}$ and $r < y{\cdot}2^{-p}$ such that $x = q{\cdot}y + r$ and, therefore, $q \le x/y < q + 2^{-p}$:

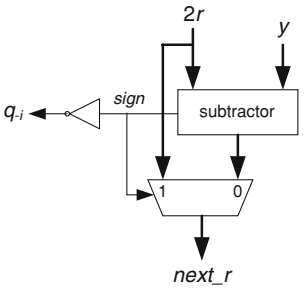


Fig. 3.20 Restoring algorithm (combinational component)

Algorithm 3.1: Restoring division algorithm

```
r0 := x;
for i in 1 .. p loop
  z := 2·ri-1 - y;
  if z < 0 then q-i := 0; ri := 2·ri-1;
  else q-i := 1; ri := z;
  end if;
end loop;
r := rp·2^-p;
```

The corresponding circuit is made up of a combinational component (Fig. 3.20), a register that stores the successive remainders r_0, r_1,…, r_p, a shift register that serially stores the quotient bits q_{-1}, q_{-2},…, q_{-p}, and a control unit. The combinational component can be defined as follows:

```
long_y <= '0'&y;
two_r <= r&'0';
dif <= two_r - long_y;
WITH dif(n) SELECT next_r <=
  dif(n-1 DOWNTO 0) WHEN '0',
  two_r(n-1 DOWNTO 0) WHEN OTHERS;
q_i <= NOT(dif(n));
```

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

An unrolled version, with $s = 2$, is made up of a combinational component consisting of two serially connected copies of the component of Fig. 3.20, a register that stores the successive remainders r_0, r_2, r_4,…, a shift register that stores the successive quotient bits q_{-1}, q_{-3}, q_{-5},…, another shift register that stores q_{-2}, q_{-4}, q_{-6},…, and a control unit. The combinational component can be defined as follows:

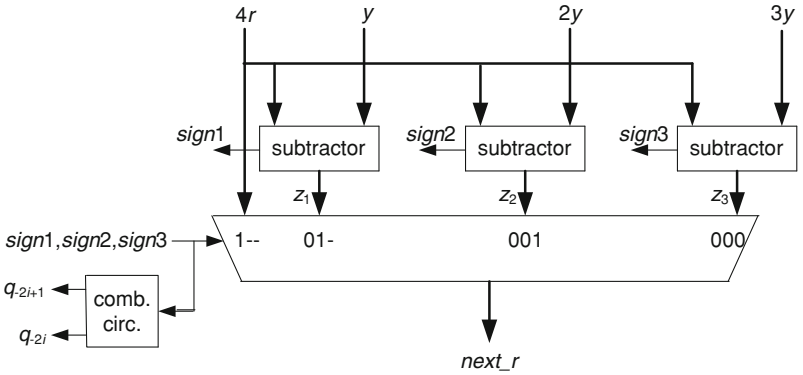


Fig. 3.21 Digit serial restoring divider ($D = 2$)

```
long_y <= '0'&y;
two_r_even <= r_even&'0';
dif_even <= two_r_even - long_y;
WITH dif_even(2*n) SELECT r_odd <=
  dif_even(2*n-1 DOWNTO 0) WHEN '0',
  two_r_even(2*n-1 DOWNTO 0) WHEN OTHERS;
two_r_odd <= r_odd&'0';
dif_odd <= two_r_odd - long_y;
WITH dif_odd(2*n) SELECT next_r <=
  dif_odd(2*n-1 DOWNTO 0) WHEN '0',
  two_r_odd(2*n-1 DOWNTO 0) WHEN OTHERS;
q_even <= NOT(dif_even(n)); q_odd <= NOT(dif_odd(n));
```

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

The first implementation (*restoring.vhd*) of Example 3.6 generates one quotient bit at each step, while the second one (*unrolled_divider.vhd*) generates two quotient bits at each step. So, as regards to the quotient generation, the first implementation could be considered as *bit-serial* and the second one as *digit-serial*, defining in this case a digit as a 2-bit number. This is a common situation in arithmetic function implementation: an algorithm processes data, or part of them, in a bit-serial manner; a modified version of this initial algorithm permits the processing of several bits, described as D, concurrently. The second implementation is called *digital-serial* and D is the digit size. This technique is used in many examples during the course of this book, in order to explore cost—performance tradeoffs: small values of D generate cost-effective circuits, while high values of D yield fast circuits.

Example 3.7
Consider again a restoring divider (Example 3.6). Algorithm 3.1 is modified in order to generate two quotient bits ($D = 2$) at each step.

Algorithm 3.2: Base-4 restoring division algorithm (p even)

```
r0 := x;
for i in 1 .. p/2 loop
  z1 := 4·r2i-2 - y;
  z2 := 4·r2i-2 - 2·y;
  z3 := 4·r2i-2 - 3·y;
  if z1 < 0 then q-(2i-1) := 0; q-2i := 0; r2i := 4·r2i-2;
  elsif z2 < 0 then q-(2i-1) := 0; q-2i := 1; r2i := z1;
  elsif z3 < 0 then q-(2i-1) := 1; q-2i := 0; r2i := z2;
  else q-(2i-1) := 1; q-2i := 1; r2i := z3;
  end if;
end loop;
r := rp/2^p;
```

The corresponding circuit is made up of a combinational component (Fig. 3.21), a register that stores the successive remainders r_0, r_2, r_4,…, a shift register that stores the successive quotient bits q_{-1}, q_{-3}, q_{-5},…, another shift register that stores q_{-2}, q_{-4}, q_{-6},…, a circuit that computes $3y$, and a control unit. The combinational component can be defined as follows:

```
z1 <= four_r - long_y;
z2 <= four_r - two_y;
z3 <= four_r - three_y;
signs <= z1(n+2)&z2(n+2)&z3(n+2);
WITH signs SELECT next_r <=
  four_r(n-1 DOWNTO 0) WHEN "111",
  z1(n-1 DOWNTO 0) WHEN "011",
  z2(n-1 DOWNTO 0) WHEN "001",
  z3(n-1 DOWNTO 0) WHEN OTHERS;
  digits: PROCESS(signs)
  BEGIN
    IF signs(2) = '1' THEN q_even <= '0'; q_odd <= '0';
    ELSIF signs(1) = '1' THEN q_even <= '0'; q_odd <= '1';
    ELSIF signs(0) = '1' THEN q_even <= '1'; q_odd <= '0';
    ELSE q_even <= '1'; q_odd <= '1';
    END IF;
  END PROCESS;
```

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

The components of Fig. 3.20 (initial restoring algorithm) and Fig. 3.21 (digit-serial restoring algorithm, with $D = 2$) have practically the same delay, namely the computation time of an n-bit subtractor, so that the minimum clock period of the corresponding dividers are practically the same. Nevertheless, the first divider needs p clock periods to perform a division while the second only needs $p/2$. So, in this example, the digit-serial approach practically divides by 2 the divider latency. On the other hand the second divider includes three subtractors instead of one.

Loop unrolling and digit-serial processing are techniques that allow the exploration of cost—performance tradeoffs, in searching for intermediate options between completely combinational (maximum cost, minimum latency) and completely sequential (minimum cost, maximum latency) circuits. Loop unrolling can be directly performed at circuit level, whatever the implemented algorithm, while digit-serial processing looks more like an algorithm transformation. Nevertheless it is not always so clear that they are different techniques.

3.3 Data Path Connectivity

In the data paths described in Chap. 2, multiplexers are associated with all the computation resource and register data inputs. With this structure, sets of operations such as $R_i := CR_j(\ldots)$ using different resources CR_j can be executed in parallel. In other words, this type of data path has *maximum connectivity*.

Assuming that the computation resources have at most p data inputs, another option is to add $p - 1$ registers acc_1, acc_2,…, acc_{p-1}, and to realize all the data transfers with two multiplexers: the first one connects all the register outputs and external signals to the first data input of every resource as well as to every register acc_i; the second one connects the resource data outputs and the first multiplexer output to the register data inputs. With this structure, an operation such as

$$R_i := CR_j\big(R_0, R_1, \ldots, R_{p-1}\big);$$

must be decomposed as follow:

$$acc_1 := R_1; acc_2 := R_2; \ldots acc_{p-1} := R_{p-1}; R_i := CR_j\big(R_0, acc_1, \ldots, acc_{p-1}\big);$$

Obviously, it is no longer possible to execute several operations in parallel. On the contrary, every operation is divided up into (at most) p steps. So, this option only makes sense if one resource of each type is used.

Example 3.8
Figure 3.22 shows a *minimum connectivity* data path for the example of Sect. 2.5. The operations of the first branch ($k_{m-i} = 0$) of Algorithm 2.4 are executed as follows:

```
acc := zB;
R := xA·acc;
acc := zA;
zB := xA+acc;
zB := zB^2;
zB := zB^2;
xB := xB·acc;
xA := xA·acc ;
acc := xB;
zA := R + acc;
zA := zA^2;
acc := xB;
xB := R·acc;
R := xA^2;
acc := zA;
xA := xP·acc ;
acc := xB;
xB := xA+acc;
xA := zB;
zA := R;
zB := zA;
```

This 2-*multiplexer structure* includes ten multiplexer inputs instead of thirty-two in Figs. 2.18 and 2.19, but does not allow the concurrent execution of compatible operations. Nevertheless, in this case, the total computation time is defined by the only time consuming operations, which are the five products $R := x_A \cdot acc$, $x_B := x_B \cdot acc$, $x_A := x_A \cdot acc$, $x_B := R \cdot acc$ and $x_A := x_P \cdot acc$, so that the latency of the circuit is still of the order of $5t_m$, t_m being the delay of a multiplier. In conclusion, the circuit of Fig. 3.22 has practically the same computation time as that of Figs. 2.18 and 2.19, and uses less multiplexing resources.

3.4 Exercises

1. Generate VHDL models of different pipelined 128-bit adders.
2. Design different digit-serial restoring dividers ($D = 3$, $D = 4$, etc.).
3. The following algorithm computes the product of two natural numbers x and y:

```
r := y;
for i in 0 to n loop
   r := 2·r + xi·y;
end loop;
```

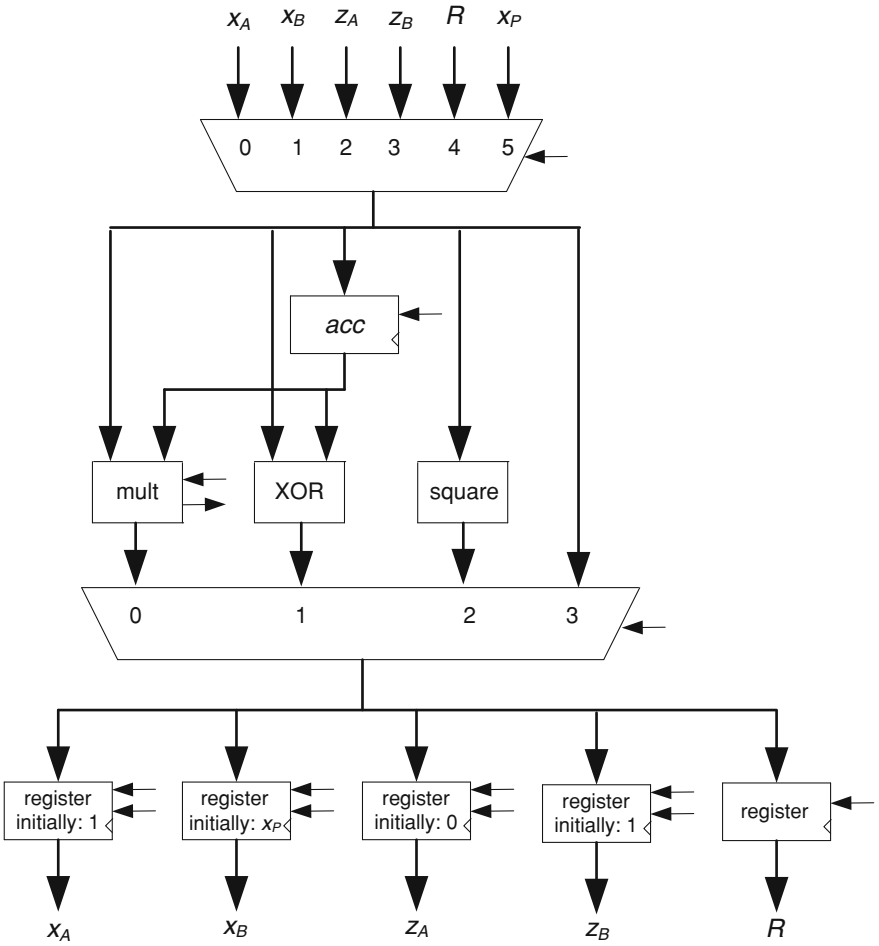


Fig. 3.22 Example of minimum connectivity data path

A computation resource *mult_and_add* that computes $2r + x_i{\cdot}y$, is available.

a. Define a combinational circuit using *mult_and_add* as a computation resource.
b. Define and compare several pipelined versions of the circuit.
c. Unroll the loop in several ways and synthesize the corresponding circuits.

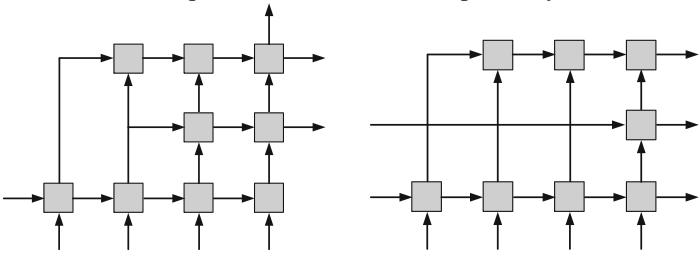
4. The following algorithm divides an integer x by a natural y, where $-y \leq x < y$, and generates a quotient $q = q_0{\cdot}q_1\ q_2 \ldots q_p \cong x/y$ (Chap. 9).

```
r0 := x;
for i in 1 .. p loop
  if ri-1 < 0  then ri := 2·ri-1 + y ; qi-1 := 0;
  else ri := 2·ri-1 - y ; qi-1 := 1; end if;
end loop;
q0 := 1 - q0; qp := 1; r := rp;
```

An adder-subtractor that computes $2r \pm y$, under the control of an *add/sub* variable, is available.

a. Define a combinational circuit.
b. Define and compare several pipelined versions of the circuit.
c. Unroll the loop in several ways and synthesize the corresponding circuits.

5. In the following combinational circuits, the delays of every cell and of every connection are equal to 5 ns and 2 ns, respectively.



For each circuit:

a. Compute the combinational delay.
b. Segment the circuit in two stages. How many registers must be added?
c. Segment the circuit in three stages. How many registers must be added?
d. What is the maximum number of segmentation stages?
e. Assume that the cutting of a connection generates two new connections whose delays are still equal to 2 ns, and that the registers have a propagation delay of 1 ns and a setup time of 0.5 ns. Which is the maximum frequency of circuits b. and c. ?

6. The following pipelined floating-point components are available: *fpmul* computes the product in 2 cycles, *fpadd* computes the sum in 3 cycles, and *fpsqrt* computes the square root in 5 cycles, all of them with a rate $r = 1$.

a. Define the schedule of a circuit that computes the distance d between two points (x_1, y_1) and (x_2, y_2) of the (x, y)-plane.
b. Define the schedule of a circuit that computes the distance d between two points (x_1, y_1, z_1) and (x_2, y_2, z_2) of the three-dimensional space.
c. Define the schedule of a circuit that computes $d = a + ((a - b)\cdot c)^{0.5}$.
d. In every case, how many registers must be added?

References

1. Parhami B (2000) Computer arithmetic: algorithms and hardware design. Oxford University Press, New York
2. De Micheli G (1994) Synthesis and optimization of digital circuits. McGraw-Hill, New York
3. Rabaey JM, Chandrakasan A, Nikolic B (2003) Digital integrated circuits: a design perspective. Prentice Hall, Upper Saddle River