

## Chapter 2

# Architecture of Digital Circuits

This chapter describes the classical architecture of many digital circuits and presents, by means of several examples, the conventional techniques that digital circuit designers can use to translate an initial algorithmic description to an actual circuit. The main topics are the decomposition of a circuit into Data Path and Control Unit and the solution of two related problems, namely scheduling and resource assignment.

In fact, modern Electronic Design Automation tools have the capacity to directly generate circuits from algorithmic descriptions, with performances—latency, cost, consumption—comparable with those obtained using more traditional methods. Those development tools are one of the main topics of [Chap. 5](#). So, it is possible that, in the future, the concepts and methods presented in this chapter will no longer be of interest to circuit designers, allowing them to concentrate on algorithmic innovative aspects rather than on scheduling and resource assignment optimization.

### 2.1 Introductory Example

As a first example, a “naive” method for computing the square root of a natural  $x$  is considered. The following algorithm sequentially computes all the pairs  $[r, s = (r + 1)^2]$  with  $r = 0, 1, 2$ , etc.:

```
r := 0; s := 1;
loop
  s := s + 2*(r+1) + 1;
  r := r + 1;
end loop;
```

Initially  $r = 0$  and thus  $s = 1$ . Then, at each step, the pair  $[r + 1, (r + 2)^2]$  is computed in function of  $r$  and  $s = (r + 1)^2$ :

$$(r + 2)^2 = ((r + 1) + 1)^2 = (r + 1)^2 + 2 \cdot (r + 1) + 1 = s + 2 \cdot (r + 1) + 1.$$

The same method can be used for computing the square root of  $x$ . For that, the loop execution is controlled by the condition  $s \leq x$ .

### Algorithm 2.1: Square root

```

r := 0; s := 1;
while s <= x loop
    s := s + 2*(r+1) + 1;
    r := r + 1;
end loop;
root := r;

```

The loop is executed as long as  $s \leq x$ , that is  $(r + 1)^2 \leq x$ . Thus, at the end of the loop execution,

$$r^2 \leq x < (r + 1)^2.$$

Obviously, this is not a good algorithm as its computation time is proportional to the square root itself, so that for great values of  $x$  ( $x \cong 2^n$ ) the number of steps is of the order of  $2^{n/2}$ . Efficient algorithms are described in [Chap. 10](#).

In order to implement Algorithm 2.1, the list of operations executed at each clock cycle must be defined. In this case, each iteration step includes three operations: evaluation of the condition  $s \leq x$ ,  $s + 2 \cdot (r + 1) + 1$  and  $r + 1$ . They can be executed in parallel. On the other hand, the successive values of  $r$  and  $s$  must be stored at each step. For that, two registers are used. Their initial values (0 and 1 respectively) are controlled by a common *load* signal, and their updating at the end of each step by a common *ce* (clock enable) signal. The circuit is shown in [Fig. 2.1](#).

To complete the circuit, a control unit in charge of generating the *load* and *ce* signals must be added. It is a finite state machine with one input *greater* (detection of the loop execution end) and two outputs, *load* and *ce*. A *start* input and a *done* output are added in order to allow the communication with other circuits. The finite state machine is shown in [Fig. 2.2](#).

The circuit of [Fig. 2.1](#) is made up of five blocks whose VHDL models are the following:

- computation of *next\_r*:

```
next_r <= r + 1;
```

- computation of *next\_s*:

```
next_s <= s + (next_r(n-2 DOWNT0 0) & '0') + 1;
```

(multiplying by 2 is the same as shifting one position to the right)

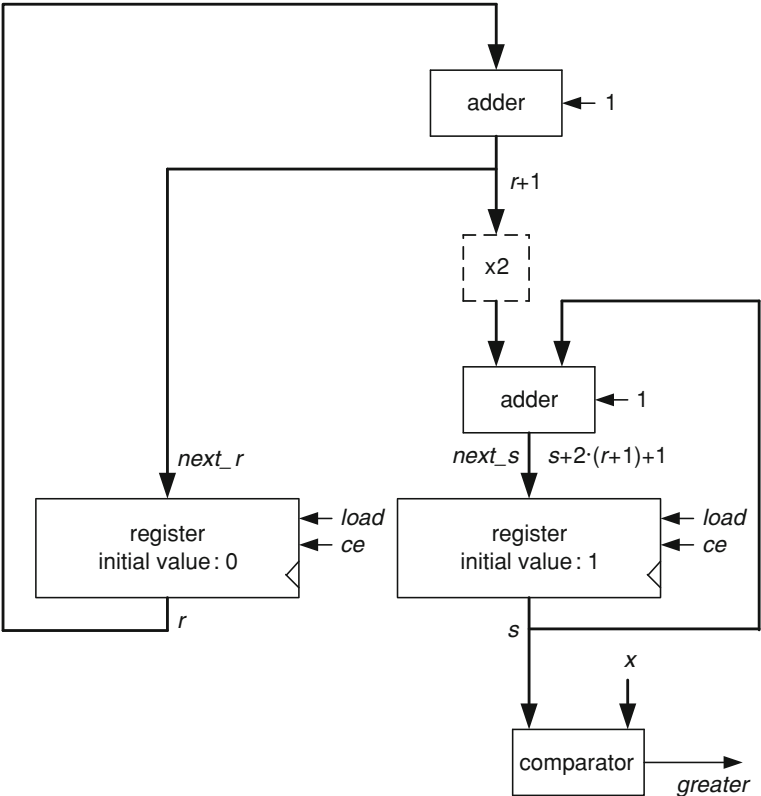
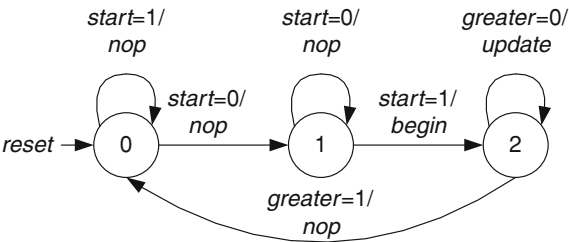


Fig. 2.1 Square root computation: data path

Fig. 2.2 Square root computation: control unit



	<i>ce</i>	<i>load</i>	<i>done</i>
<i>nop</i>	0	0	1
<i>begin</i>	0	1	0
<i>update</i>	1	0	0

- register *r*:

```

register_r: PROCESS(clk)
BEGIN
    IF clk'EVENT AND clk = '1' THEN
        IF LOAD = '1' THEN r <= (OTHERS => '0');
        ELSIF ce = '1' THEN r <= next_r;
        END IF;
    END IF;
END PROCESS;

```

- register *s*:

```

register_s: PROCESS(clk)
BEGIN
    IF clk'EVENT AND clk = '1' THEN
        IF LOAD = '1' THEN s <= CONV_STD_LOGIC_VECTOR(1,8);
        ELSIF ce = '1' THEN s <= next_s;
        END IF;
    END IF;
END PROCESS;

```

- end of loop detection:

```

greater <= '1' WHEN s > x ELSE '0';

```

The control unit is a Mealy finite state machine that can be modeled as follows:

- next state computation:

```

control_unit_next_state: PROCESS(clk, reset)
BEGIN
    IF reset = '1' THEN current_state <= 0;
    ELSIF clk'event AND clk= '1' THEN
        CASE current_state IS
            WHEN 0 => IF start = '0' THEN current_state <= 1;
                       END IF;
            WHEN 1 => IF start = '1' THEN current_state <= 2;
                       END IF;
            WHEN 2 => IF greater = '1' THEN current_state <= 0;
                       END IF;
        END CASE;
    END IF;
END PROCESS;

```

- output state computation:

```

control_unit_output: PROCESS(current_state, start, greater)
BEGIN
  CASE current_state IS
    WHEN 0 => ce <= '0'; load <= '0'; done <= '1';
    WHEN 1 => ce <= '0';
              IF start = '1' THEN load <= '1'; done <= '0';
              ELSE load <= '0'; done <= '1'; END IF;
    WHEN 2 => IF greater = '0' THEN ce <= '1';
              ELSE ce <= '0'; END IF;
              load <= '0'; done <= '0';
  END CASE;
END PROCESS;

```

The circuit of Fig. 2.1 includes three  $n$ -bit adders: a half adder for computing  $next\_r$ , a full adder for computing  $next\_s$  and another full adder (actually a subtractor) for detecting the condition  $s > x$ . Another option is to use one adder and to decompose each iteration step into three clock cycles. For that, Algorithm 2.1 is slightly modified.

### Algorithm 2.2: Square root, version 2

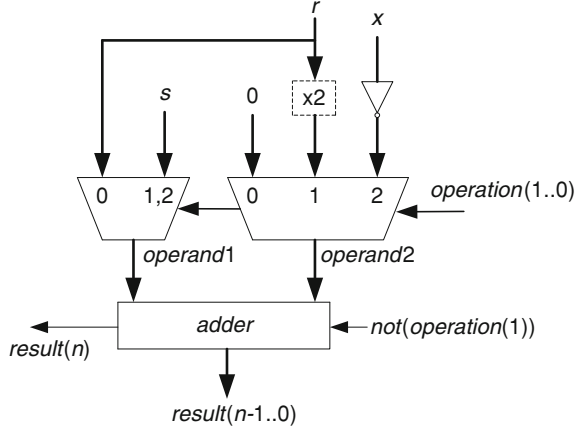
```

r := 0; s := 1;
if s > x then greater := true; else greater := false; end if;
while not(greater) loop
  --cycle 1:
  r := r + 1;
  --cycle 2:
  s := s + 2*r + 1;
  --cycle 3:
  if s > x then greater := true; end if;
end loop;

```

A circuit able to execute the three operations, that is  $r + 1$ ,  $s + 2 \cdot r + 1$  and evaluation of the condition  $s > x$  must be defined. The condition  $s > x$  is equivalent to  $s \geq x + 1$  or  $s + 2^n - 1 - x \geq 2^n$ . The binary representation of  $2^n - 1 - x$  is obtained by complementing the bits of the binary representation of  $x$ . So, the condition  $s > x$  is equivalent to  $s + not(x) \geq 2^n$ . Thus, the three operations amount to additions:  $r + 1$ ,  $s + 2 \cdot r + 1$  and  $s + not(x)$ . In the latter case, the output carry defines the value of *greater*. The corresponding circuit is shown in Fig. 2.3. It is an example of *programmable computation resource*: under the control of a 2-bit command *operation*, it can execute the three previously defined operations. The corresponding VHDL description is the following:

**Fig. 2.3** Square root computation: programmable computation resource



```

shifted_r <= r(n-2 DOWNT0 0) & '0';
complemented_x <= NOT(x);
WITH operation SELECT operand_1 <=
    r WHEN "00", s WHEN OTHERS;
WITH operation SELECT operand_2 <= zero WHEN "00",
    shifted_r WHEN "01", complemented_x WHEN OTHERS;
result <= '0' & operand_1 + operand_2 + NOT(operation(1));

```

The complete circuit is shown in Fig. 2.4.

A control unit must be added. It is a finite state machine with one input *greater* and five outputs *load*, *ce\_r*, *ce\_s*, *ce\_greater*. As before, a *start* input and a *done* output are added in order to allow the communication with other circuits. The finite state machine is shown in Fig. 2.5.

The building blocks of the circuit of Fig. 2.4 (apart from the programmable resource) are the following:

- register *r*:

```

register_r: PROCESS(clk)
BEGIN
    IF clk'EVENT AND clk = '1' THEN
        IF LOAD = '1' THEN r <= (OTHERS => '0');
        ELSIF ce_r = '1' THEN r <= result(n-1 DOWNT0 0);
        END IF;
    END IF;
END PROCESS;

```

- register *s*:

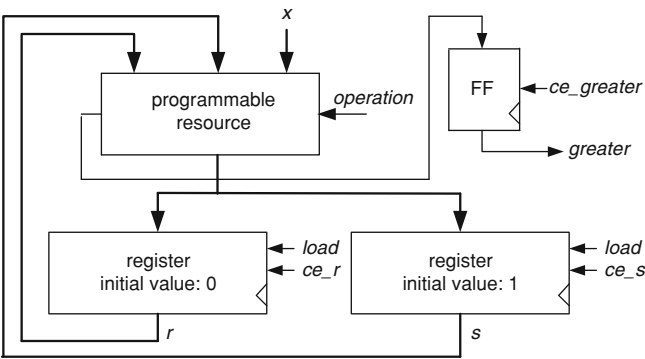


Fig. 2.4 Square root computation, second version: data path

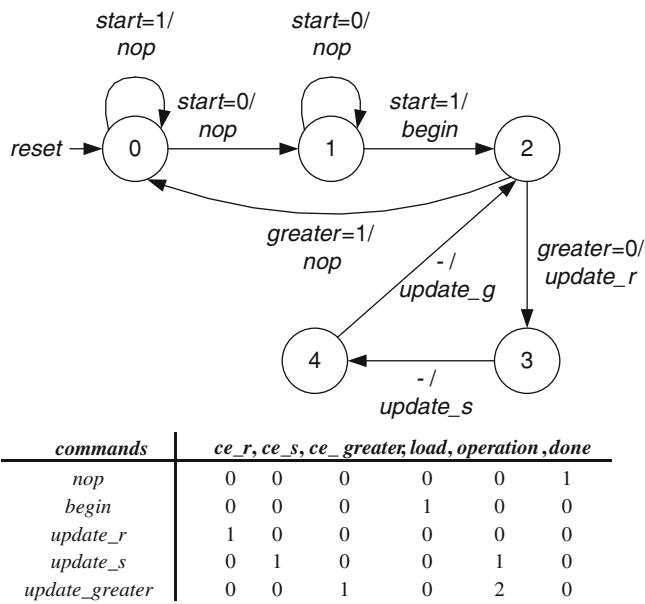


Fig. 2.5 Square root computation, second version: control unit

```

register_s: PROCESS(clk)
BEGIN
    IF clk'EVENT AND clk = '1' THEN
        IF LOAD = '1' THEN s <= CONV_STD_LOGIC_VECTOR(1,8);
        ELSIF ce_s = '1' THEN s <= result(n-1 DOWNT0 0);
        END IF;
    END IF;
END PROCESS;

```

- **flip-flop greater**

```

ff_greater: PROCESS(clk)
BEGIN
    IF clk'EVENT AND clk = '1' THEN
        IF LOAD = '1' THEN greater <= '0';
        ELSIF ce_greater = '1' THEN greater <= result(n);
        END IF;
    END IF;
END PROCESS;

```

The control unit is a Mealy finite state machine whose VHDL model is the following:

- **next state computation:**

```

control_unit_next_state: PROCESS(clk, reset)
BEGIN
    IF reset = '1' THEN current_state <= 0;
    ELSIF clk'event AND clk= '1' THEN
        CASE current_state IS
            WHEN 0 => IF start = '0' THEN
                        current_state <= 1; END IF;
            WHEN 1 => IF start = '1' THEN
                        current_state <= 2; END IF;
            WHEN 2 => IF greater = '1' THEN current_state <= 0;
                        ELSE current_state <= 3; END IF;
            WHEN 3 => current_state <= 4;
            WHEN 4 => current_state <= 2;
        END CASE;
    END IF;
END PROCESS;

```

- **output state computation:**



```

control_unit_output: PROCESS(current_state, start, greater)
BEGIN
  CASE current_state IS
    WHEN 0 => ce_r <= '0'; ce_s <= '0'; ce_greater <= '0';
              load <= '0'; operation <= "00"; done <= '1';
    WHEN 1 => ce_r <= '0'; ce_s <= '0'; ce_greater <= '0';
              operation <= "00";
              IF start = '1' THEN load <= '1'; done <= '0';
              ELSE load <= '0'; done <= '1'; END IF;
    WHEN 2 => IF greater = '0' THEN ce_r <= '1';
              ELSE ce_r <= '0'; END IF;
              ce_s <= '0'; ce_greater <= '0'; load <= '0';
              operation <= "00"; done <= '0';
    WHEN 3 => ce_r <= '0'; ce_s <= '1'; ce_greater <= '0';
              load <= '0'; operation <= "01"; done <= '0';
    WHEN 4 => ce_r <= '0'; ce_s <= '0'; ce_greater <= '1';
              load <= '0'; operation <= "10"; done <= '0';

  END CASE;
END PROCESS;

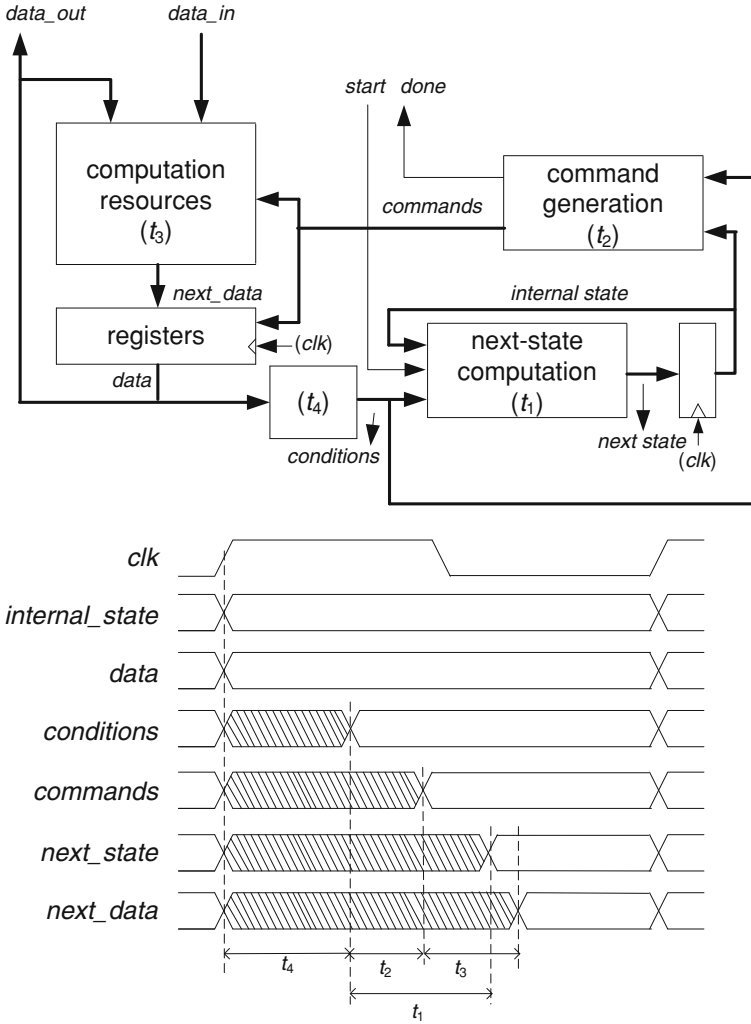
```

Complete VHDL models (*square\_root.vhd*) of both circuits (Figs. 2.1, 2.4) are available at the Authors' web page.

## 2.2 Data Path and Control Unit

The general structure of a digital circuit is shown in Fig. 2.6. It consists of a *data path* and a *control unit*. The data path (leftmost part of Fig. 2.6) includes computation resources executing the algorithm operations, registers storing the algorithm variables, and programmable connections (for example multiplexers, not represented in Fig. 2.6) between resource outputs and register inputs, and between register outputs and resource inputs. The control unit (rightmost part of Fig. 2.6) is a finite state machine. It controls the sequence of data path operations by means of a set of control signals (*commands*) such as clock enables of registers, programming of computation resources and multiplexers, and so on. It receives from the data path some feedback information (*conditions*) corresponding to the algorithm control statements (loop, if, case).

In fact, the data path could also be considered as being a finite state machine. Its internal states are all the possible register contents, the next-state computation is performed by the computation resources, and the output states are all the possible values of *conditions*. Nevertheless, the number of internal states is enormous and there is generally no sense in using a finite state machine model for the data path. However, it is interesting to observe that the data path of Fig. 2.6 is a Moore



**Fig. 2.6** Structure of a digital circuit: data path and control unit

machine (the output state only depends on the internal state) while the control unit could be a Moore or a Mealy machine. An important point is that, when two finite state machines are interconnected, one of them must be a Moore machine in order to avoid combinational loops.

According to the chronograms of Fig. 2.6, there are two critical paths: from the data registers to the internal state register, and from the data registers to the data registers. The corresponding delays are

$$T_{data-state} = t_4 + t_1 \quad (2.1)$$

and

$$T_{data-data} = t_4 + t_2 + t_3, \quad (2.2)$$

where  $t_1$  is the computation time of the next internal state,  $t_2$  the computation time of the commands,  $t_3$  the maximum delay of the computation resources and  $t_4$  the computation time of the conditions (the set up and hold times of the registers have not been taken into account).

The clock period must satisfy

$$T_{clk} > \max\{t_4 + t_1, t_4 + t_2 + t_3\}. \quad (2.3)$$

If the control unit were a Moore machine, there would be no direct path from the data registers to the data registers, so that (2.2) and (2.3) should be replaced by

$$T_{state-data} = t_2 + t_3 \quad (2.4)$$

and

$$T_{clk} > \max\{t_4 + t_1, t_2 + t_3\}. \quad (2.5)$$

In fact, it is always possible to use a Moore machine for the control unit. Generally it has more internal states than an equivalent Mealy machine and the algorithm execution needs more clock cycles. If the values of  $t_1$  to  $t_4$  do not substantially vary, the conclusion could be that the Moore approach needs more, but shorter, clock cycles. Many designers also consider that Moore machines are safer than Mealy machines.

In order to increase the maximum frequency, an interesting option is to insert a command register at the output of the command generation block. Then relation (2.2) is substituted by

$$T_{data-commands} = t_4 + t_2 \quad \text{and} \quad T_{commands-data} = t_3, \quad (2.6)$$

so that

$$T_{clk} > \max\{t_4 + t_1, t_4 + t_2, t_3\}. \quad (2.7)$$

With this type of *registered Mealy machine*, the commands are available one cycle later than with a non-registered machine, so that additional cycles must be sometimes inserted in order that the data path and its control unit remain synchronized.

To summarize, the implementation of an algorithm is based upon a decomposition of the circuit into a data path and a control unit. The data path is in charge of the algorithm operations and can be roughly defined in the following way: associate registers to the algorithm variables, implement resources able to execute the algorithm operations, and insert programmable connections (multiplexers) between the register outputs (the operands) and the resource inputs, and between the resource outputs (the results) and the register inputs. The control unit is a finite state machine whose internal states roughly correspond to the algorithm steps, the

input states are conditions (flags) generated by the data path, and the output states are commands transmitted to the data path.

In fact, the definition of a data path poses a series of optimization problems, some of them being dealt with in the next sections, for example: scheduling of the operations, assignment of computation resources to operations, and assignment of registers to variables. It is also important to notice that minor algorithm modifications sometimes yield major circuit optimizations.

## 2.3 Operation Scheduling

Operation scheduling consists in defining which particular operations are in the process of execution during every clock cycle. For that purpose, an important concept is that of *precedence relation*. It defines which of the operations must be completed before starting a new one: if some result  $r$  of an operation  $A$  is an initial operand of some operation  $B$ , the computation of  $r$  must be completed before the execution of  $B$  starts. So, the execution of  $A$  must be scheduled before the execution of  $B$ .

### 2.3.1 Introductory Example

A *carry-save adder* or *3-to-2 counter* (Sect. 7.7) is a circuit with 3 inputs and 2 outputs. The inputs  $x_i$  and the outputs  $y_j$  are naturals. Its behavior is defined by the following relation:

$$x_1 + x_2 + x_3 = y_1 + y_2. \quad (2.8)$$

It is made up of 1-bit full adders working in parallel. An example where  $x_1, x_2$  and  $x_3$  are 4-bit numbers, and  $y_1$  and  $y_2$  are 5-bit numbers, is shown in Fig. 2.7.

The delay of a carry-save adder is equal to the delay  $T_{FA}$  of a 1-bit full adder, independently of the number of bits of the operands. Let CSA be the function associated to (2.8), that is

$$(y_1, y_2) = \text{CSA}(x_1, x_2, x_3). \quad (2.9)$$

Using carry-save adders as computation resources, a 7-to-3 counter can be implemented. It allows expressing the sum of seven naturals under the form of the sum of three naturals, that is

$$x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 = y_1 + y_2 + y_3.$$

In order to compute  $y_1, y_2$  and  $y_3$ , the following operations are executed ( $op_1$  to  $op_4$  are labels):

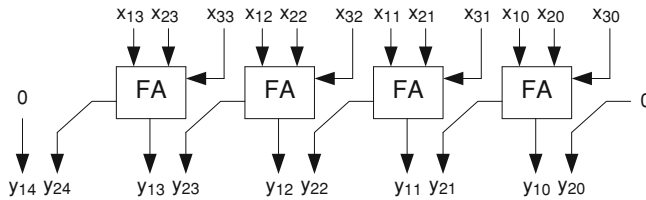


Fig. 2.7 Carry-save adder

$$\begin{aligned}
 op_1 : (a_1, a_2) &= \text{CSA}(x_1, x_2, x_3), \\
 op_2 : (b_1, b_2) &= \text{CSA}(x_4, x_5, x_6), \\
 op_3 : (c_1, c_2) &= \text{CSA}(a_2, b_2, x_7), \\
 op_4 : (d_1, d_2) &= \text{CSA}(a_1, b_1, c_1).
 \end{aligned} \tag{2.10}$$

According to (2.10) and the definition of CSA

$$\begin{aligned}
 a_1 + a_2 &= x_1 + x_2 + x_3, \\
 b_1 + b_2 &= x_4 + x_5 + x_6, \\
 c_1 + c_2 &= a_2 + b_2 + x_7, \\
 d_1 + d_2 &= a_1 + b_1 + c_1,
 \end{aligned}$$

so that

$$\begin{aligned}
 c_1 + c_2 + d_1 + d_2 &= a_2 + b_2 + x_7 + a_1 + b_1 + c_1 \\
 &= x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + c_1.
 \end{aligned}$$

Thus

$$c_2 + d_1 + d_2 = a_2 + b_2 + x_7 + a_1 + b_1 + c_1 = x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7$$

and  $y_1$ ,  $y_2$  and  $y_3$  can be defined as follows:

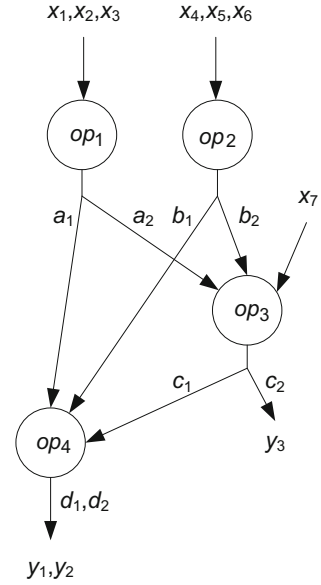
$$y_1 = d_1, y_2 = d_2, y_3 = c_2.$$

The corresponding precedence relation is defined by the graph of Fig. 2.8, according to which  $op_1$  and  $op_2$  must be executed before  $op_3$ , and  $op_3$  before  $op_4$ . Thus, the minimum computation time is equal to  $3 \cdot T_{FA}$ .

For implementing (2.10) the following options could be considered:

1. A combinational circuit, made up of four carry-save adders, whose structure is the same as that of the graph of Fig. 2.8. Its computation time is equal to  $3 \cdot T_{FA}$  and its cost to  $4 \cdot C_{CSA}$ , being  $C_{CSA}$  the cost of a carry-save adder. This is probably a bad solution because the cost is high (4 carry-save adders) and the delay is long (3 full-adders) so that the minimum clock cycle of a synchronous circuit including this 7-to-3 counter should be greater than  $3 \cdot T_{FA}$ .
2. A data path including two carry-save adders and several registers (Sect. 2.5). The computation is executed in three cycles:

**Fig. 2.8** Precedence relation of a 7-to-3 counter



- 1:  $(a_1, a_2, b_1, b_2) := (\text{CSA}(x_1, x_2, x_3), \text{CSA}(x_4, x_5, x_6))$ ;
- 2:  $(c_1, y_3) := \text{CSA}(a_2, b_2, x_7)$ ;
- 3:  $(y_1, y_2) := \text{CSA}(a_1, b_1, c_1)$ ;

The computation time is equal to  $3 \cdot T_{clk}$ , where  $T_{clk} > T_{FA}$ , and the cost equal to  $2 \cdot C_{CSA}$ , plus the cost of the additional registers, controllible connections and control unit.

3. A data path including one carry-save adder and several registers. The computation is executed in four cycles:

- 1:  $(a_1, a_2) := \text{CSA}(x_1, x_2, x_3)$ ;
- 2:  $(b_1, b_2) := \text{CSA}(x_4, x_5, x_6)$ ;
- 3:  $(c_1, y_3) := \text{CSA}(a_2, b_2, x_7)$ ;
- 4:  $(y_1, y_2) := \text{CSA}(a_1, b_1, c_1)$ ;

The computation time is equal to  $4 \cdot T_{clk}$ , where  $T_{clk} > T_{FA}$ , and the cost equal to  $C_{CSA}$ , plus the cost of the additional registers, controllible connections and control unit.

In conclusion, there are several implementations, with different costs and delays, corresponding to the set of operations in (2.10). In order to get an optimized circuit, according to some predefined criteria, the space for possible implementations must be explored. For that, optimization methods must be used.

### 2.3.2 Precedence Graph

Consider a *computation scheme*, that is to say, an algorithm without branches and loops. Formally it can be defined by a set of operations

$$op_J : (x_i, x_k, \dots) = f(x_l, x_m, \dots), \quad (2.11)$$

where  $x_i, x_k, x_l, x_m, \dots$  are variables of the algorithm and  $f$  one of the algorithm operation types (*computation primitives*). Then, the precedence graph (or *data flow graph*) is defined as follows:

- associate a vertex to each operation  $op_J$ ,
- draw an arc between vertices  $op_J$  and  $op_M$  if one of the results generated by  $op_J$  is used by  $op_M$ .

An example was given in Sect. 2.3.1 (operations (2.10) and Fig. 2.8).

Assume that the computation times of all operations are known. Let  $t_{JM}$  be the computation time, expressed in number of clock cycles, of the result(s) generated by  $op_J$  and used by  $op_M$ . Then, a schedule of the algorithm is an application  $Sch$  from the set of vertices to the set of naturals that defines the number  $Sch(op_J)$  of the cycle at the beginning of which the computation of  $op_J$  starts. A necessary condition is that

$$Sch(op_M) \geq Sch(op_J) + t_{JM} \quad (2.12)$$

if there is an arc from  $op_J$  to  $op_M$ .

As an example, if the clock period is greater than the delay of a full adder, then, in the computation scheme (2.10), all the delays are equal to 1 and two admissible schedules are

$$Sch(op_1) = 1, Sch(op_2) = 1, Sch(op_3) = 2, Sch(op_4) = 3, \quad (2.13)$$

$$Sch(op_1) = 1, Sch(op_2) = 2, Sch(op_3) = 3, Sch(op_4) = 4. \quad (2.14)$$

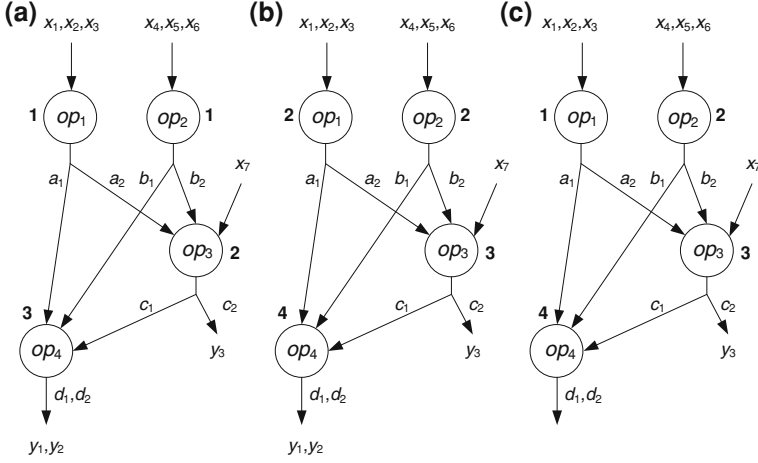
They correspond to the options 2 and 3 of Sect. 2.3.1.

The definition of an admissible schedule is an easy task. As an example, the following algorithm defines an ASAP (as soon as possible) schedule:

- initial step:  $Sch(op_J) = 1$  for all initial (without antecessor) vertices  $op_J$ ;
- step number  $n + 1$ : choose an unscheduled vertex  $op_M$  whose total amount of antecessors, say  $op_P, op_Q, \dots$  have already been scheduled, and define  $Sch(op_M) = \text{maximum}\{Sch(op_P) + t_{PM}, Sch(op_Q) + t_{QM}, \dots\}$ .

Applied to (2.10) the ASAP algorithm gives (2.13). The corresponding data flow graph is shown in Fig. 2.9a.

An ALAP (as late as possible) schedule can also be defined. For that, assume that the latest admissible starting cycle for all the final vertices (without successor) has been previously specified:



**Fig. 2.9** 7-to-3 counter: **a** ASAP schedule. **b** ALAP schedule. **c** Admissible schedule

- initial step:  $Sch(op_M) = \text{latest admissible starting cycle of } op_M \text{ for all final vertices } op_M$ ;
- step number  $n + 1$ : choose an unscheduled vertex  $op_J$  whose all successors, say  $op_P, op_Q, \dots$  have already been scheduled, and define  $Sch(op_J) = \min\{Sch(op_P) - t_{JP}, Sch(op_Q) - t_{JQ}, \dots\}$ .

Applied to (2.10), with  $Sch(op_4) = 4$ , the ALAP algorithm generates the data flow graph of Fig. 2.9b.

Let  $ASAP\_Sch$  and  $ALAP\_Sch$  be ASAP and ALAP schedules, respectively. Obviously, if  $op_M$  is a final operation, the previously specified value  $ALAP\_Sch(op_M)$  must be greater than or equal to  $ASAP\_Sch(op_M)$ . More generally, assuming that the latest admissible starting cycle for all the final operations has been previously specified, for any admissible schedule  $Sch$  the following relation holds:

$$ASAP\_Sch(op_J) \leq Sch(op_J) \leq ALAP\_Sch(op_J), \forall op_J. \quad (2.15)$$

Along with (2.12), relation (2.15) defines the admissible schedules.

An example of admissible schedule is defined by (2.14), to which corresponds the data flow graph of Fig. 2.9c.

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

### Example 2.1

Given a point  $P = (x_P, y_P)$  of an elliptic curve and a natural  $k$ , the scalar product  $kP = P + P + \dots + P$  can be defined [1, 2]. In the case of the curve  $y^2 + xy = x^3 + ax + 1$  over the binary field, the following formal algorithm [3] computes  $kP$ . The initial data are the scalar  $k = k_m - 1 \ k_m - 2 \dots k_0$  and the  $x$ -coordinate  $x_P$  of  $P$ . All the algorithm variables are elements of the Galois field  $GF(2^m)$ , that is, polynomials of degree  $m$  over the binary field  $GF(2)$  (Chap. 13).



**Algorithm 2.3: Scalar product, projective coordinates**

```

 $x_A := 1; z_A := 0; x_B := x_P; z_B := 1;$ 
for i in 1 .. m loop
   $u := (x_A \cdot z_B + x_B \cdot z_A)^2;$ 
   $v := x_P \cdot u + x_A \cdot x_B \cdot z_A \cdot z_B;$ 
  if  $k(m-i) = 0$  then
     $x_B := v;$ 
     $z_B := u;$ 
     $y := x_A^2 \cdot z_A^2;$ 
     $x_A := x_A^4 + z_A^4;$ 
     $z_A := y;$ 
  else
     $x_A := v;$ 
     $z_A := u;$ 
     $z := x_B^2 \cdot z_B^2;$ 
     $x_B := x_B^4 + z_B^4;$ 
     $z_B := z;$ 
  end if;
end loop;

```

In fact, the preceding algorithm computes the value of four variables  $x_A$ ,  $z_A$ ,  $x_B$  and  $z_B$  in function of  $k$  and  $x_P$ . A final, not included, step would be to compute the coordinates of  $kP$  in function of the coordinates of  $P$  ( $x_P$  and  $y_P$ ) and of the final values of  $x_A$ ,  $z_A$ ,  $x_B$  and  $z_B$ .

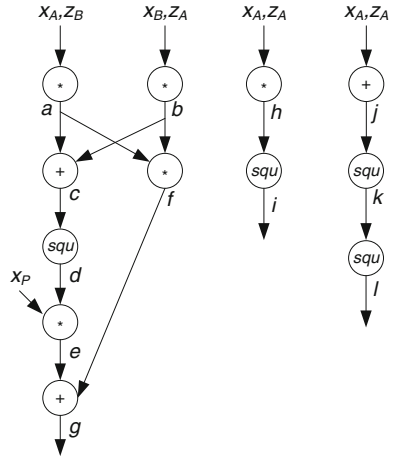
Consider one step of the main iteration of Algorithm 2.3, and assume that  $k_m - i = 0$ . The following computation scheme computes the new values of  $x_A$ ,  $z_A$ ,  $x_B$  and  $z_B$  in function of their initial values and of  $x_P$ . The available computation primitives are the addition, multiplication and squaring in  $GF(2^m)$  (Chap. 13).

```

op1:  $a = x_A \cdot z_B,$ 
op2:  $b = x_B \cdot z_A,$ 
op3:  $c = a + b,$ 
op4:  $d = c^2,$ 
op5:  $e = x_P \cdot d,$ 
op6:  $f = a \cdot b,$ 
op7:  $g = e + f,$ 
op8:  $h = x_A \cdot z_A,$ 
op9:  $i = h^2,$ 
op10:  $j = x_A + z_A,$ 
op11:  $k = j^2,$ 
op12:  $l = k^2.$ 

```

**Fig. 2.10** Example 2.1:  
precedence graph



The updated values of  $x_A$ ,  $z_A$ ,  $x_B$  and  $z_B$  are  $x_A = l$ ,  $z_A = i$ ,  $x_B = g$ ,  $z_B = d$ . The corresponding data flow graph is shown in Fig. 2.10. The operation type corresponding to every vertex is indicated (instead of the operation label). If  $k_m - i = 1$  the computation scheme is the same but for the interchange of indexes  $A$  and  $B$ .

Addition and squaring in  $GF(2^m)$  are relatively simple one-cycle operations, while multiplication is a much more complex operation whose maximum computation time is  $t_m \gg 1$ . In what follows it is assumed that  $t_m = 300$  cycles. An ASAP schedule is shown in Fig. 2.11. The computation of  $g$  starts at the beginning of cycle 603 so that all the final results are available at the beginning of cycle 604. The corresponding circuit must include three multipliers as the computations of  $a$ ,  $b$  and  $h$  start at the same time.

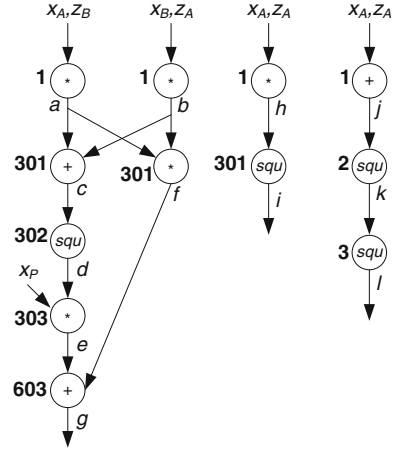
The computation scheme includes 5 multiplications. Thus, in order to execute the algorithm with only one multiplier, the minimum computation time is 1,500. More precisely, one of the multiplications  $e$ ,  $f$  or  $h$  cannot start before cycle 1,201, so that the next operation ( $g$  or  $i$ ) cannot start before cycle 1,501. An ALAP schedule, assuming that the computations of  $g$  and  $i$  start at the beginning of cycle 1,501, is shown in Fig. 2.12.

### 2.3.3 Optimization Problems

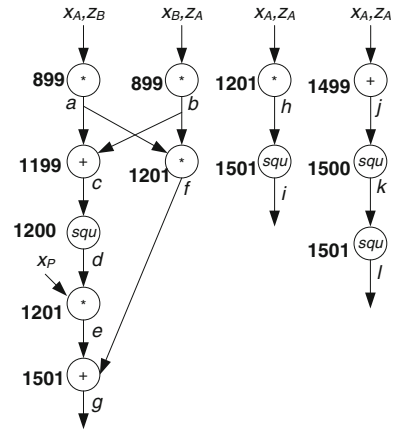
Assuming that the latest admissible starting cycle for all the final operations has been previously specified then any schedule, such that (2.12) and (2.15) hold true, can be chosen. This poses optimization problems. For example:

1. Assuming that the maximum computation time has been previously specified, look for a schedule that minimizes the number of computation resources of each type.

**Fig. 2.11** Example 2.1:  
ASAP schedule



**Fig. 2.12** Example 2.1:  
ALAP schedule with  
 $Sch(g) = 1501$



- Assuming that the number of available computation resources of each type has been previously specified, minimize the computation time.

An important concept is the *computation width*  $w(f)$  with respect to the computation primitive (operation type)  $f$ . First define the *activity intervals* of  $f$ . Assume that  $f$  is the primitive corresponding to the operation  $op_J$ , that is

$$op_J : (x_i, x_k, \dots) = f(x_l, x_m, \dots).$$

Then

$$[Sch(op_J), Sch(op_J) + maximum\{t_{JM}\}]$$

is an activity interval of  $f$ . This means that a resource of type  $f$  must be available from the beginning of cycle  $Sch(op_J)$  to the end of cycle  $Sch(op_J) + t_{JM}$  for all  $M$  such that there is an arc from  $op_J$  to  $op_M$ . An incompatibility relation over the set of activity intervals of  $f$  can be defined: two intervals are incompatible if they overlap. If two intervals overlap, it is obvious that the corresponding operations cannot be executed by the same computation resource. Thus, a particular resource of type  $f$  must be associated to each activity interval of  $f$  in such a way that if two intervals overlap, then two distinct resources of the same type must be used. The minimum number of computation resources of type  $f$  is the *computation width*  $w(f)$ .

The following graphical method can be used for computing  $w(f)$ .

- Associate a vertex to every activity interval.
- Draw an edge between two vertices if the corresponding intervals overlap.
- Color the vertices in such a way that two vertices connected by an edge have different colors (a classical problem of graph theory).

Then,  $w(f)$  is the number of different colors, and every color defines a particular resource assigned to all edges (activity intervals) with this color.

### Example 2.2

Consider the scheduled precedence graph of Fig. 2.11. The activity intervals of the multiplication are

$$a : [1, 300], b : [1, 300], h : [1, 300], f : [301, 600], e : [303, 602].$$

The corresponding incompatibility graph is shown in Fig. 2.13a. It can be colored with three colors ( $c_1$ ,  $c_2$  and  $c_3$  in Fig. 2.13a). Thus, the computation width with respect to the multiplication is equal to 3.

If the scheduled precedence graph of Fig. 2.12 is considered, then the activity intervals of the multiplication are

$$a : [899, 1198], b : [899, 1198], h : [1201, 1500], f : [1201, 1500], e : [1201, 1500].$$

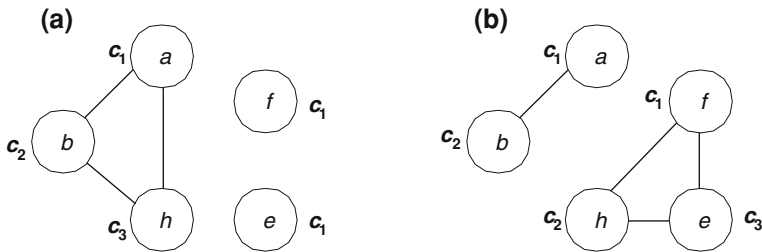
The corresponding incompatibility graph is shown in Fig. 2.13b. It can be colored with three colors. Thus, the computation width with respect to the multiplication is still equal to 3.

Nevertheless, other schedules can be defined. According to (2.15) and Figs. 2.11 and 2.12, the time intervals during which the five multiplications can start are the following:

$$a : [1, 899], b : [1, 899], h : [1, 1201], f : [301, 1201], e : [303, 1201].$$

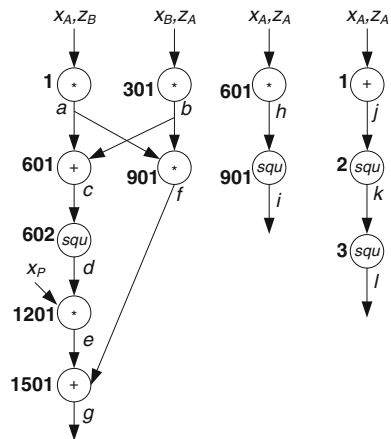
As an example, consider the admissible schedule of Fig. 2.14. The activity intervals of the multiplication operation are

$$a : [1, 300], b : [301, 600], h : [601, 900], f : [901, 1200], e : [1201, 1500].$$



**Fig. 2.13** ColoringComputation width: graph coloring

**Fig. 2.14** Example 2.1:  
admissible schedule using  
only one multiplier



They do not overlap hence the incompatibility graph does not include any edge and can be colored with one color. The computation width with respect to the multiplication is equal to 1.

Thus, the two optimization problems mentioned above can be expressed in terms of computation widths:

1. Assuming that the maximum computation time has been previously specified, look for a schedule that minimizes some cost function

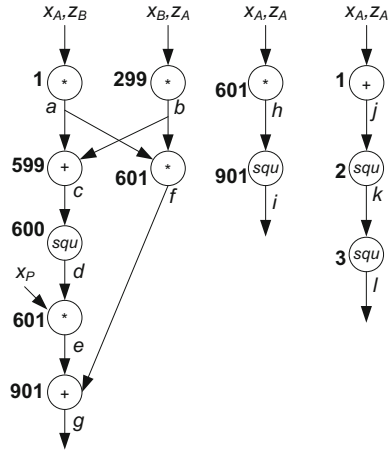
$$C = c_1 \cdot w(f^1) + c_2 \cdot w(f^2) + \dots + c_m \cdot w(f^m) \quad (2.16)$$

where  $f^1, f^2, \dots, f^m$  are the computation primitives and  $c_1, c_2, \dots, c_m$  their corresponding costs.

2. Assuming that the maximum computation width  $w(f)$  with respect to every computation primitive  $f$  has been previously specified, look for a schedule that minimizes the computation time.

Both are classical problems of scheduling theory. They can be expressed in terms of integer linear programming problems whose variables are  $x_{It}$  for all

**Fig. 2.15** Example 2.3:  
schedule corresponding to the  
first optimization problem



operation indices  $I$  and all possible cycle numbers  $t$ :  $x_{It} = 1$  if  $Sch(e_I) = t$ , 0 otherwise. Nevertheless, except for small computation schemes—generally tractable by hand—the so obtained linear programs are intractable. Modern Electronic Design Automation tools execute several types of heuristic algorithms applied to different optimization problems (not only to schedule optimization). Some of the more common heuristic strategies are *list scheduling*, *simulated annealing* and *genetic algorithms*.

### Example 2.3

The list scheduling algorithm, applied to the graph of Fig. 2.10, with  $t_m = 300$  and assuming that the latest admissible starting cycle for all the final operations is cycle number 901 (first optimization problem), would generate the schedule of Fig. 2.15. The list scheduling algorithm, applied to the same graph of Fig. 2.10, with  $t_m = 300$  and assuming that the computation width is equal to 1 for all operations (second optimization problem), would generate the schedule of Fig. 2.14.

## 2.4 Resource Assignment

Once the operation schedule has been defined, several decisions must be taken.

- The number  $w(f)$  of resources of type  $f$  is known, but it remains to decide which particular computation resource executes each operation. Furthermore the definition of multifunctional programmable resources could also be considered.
- As regards the storing resources, a simple solution is to assign a particular register to every variable. Nevertheless, in some cases the same register can be used for storing different variables.

A key concept for assigning registers to variables is the *lifetime*  $[t_I, t_J]$  of every variable:  $t_I$  is the number of the cycle during which its value is generated, and  $t_J$  is the number of the last cycle during which its value is used.

### Example 2.4

Consider the computation scheme of Example 2.1 and the schedule of Fig. 2.14. The computation width is equal to 1 for all primitives (multiplication, addition and squaring). The computation is executed as follows:

```
initial data:  $x_A, z_A, x_B, z_B$ 
cycle 1:  $j := x_A + z_A$ ; start  $x_A \cdot z_B$ ;
cycle 2:  $k := j^2$ ;
cycle 3:  $l := k^2$ ;
cycle 300:  $a := x_A \cdot z_B$ ;
cycle 301: start  $x_B \cdot z_A$ ;
cycle 600:  $b := x_B \cdot z_A$ ;
cycle 601:  $c := a + b$ ; start  $x_A \cdot z_A$ ;
cycle 602:  $d := c^2$ ;
cycle 900:  $h := x_A \cdot z_A$ ;
cycle 901:  $i := h^2$ ; start  $a \cdot b$ ;
cycle 1200:  $f := a \cdot b$ ;
cycle 1201: start  $x_P \cdot d$ ;
cycle 1500:  $e := x_P \cdot d$ ;
cycle 1501:  $g := e + f$ ;
final results:  $(x_A, z_A, x_B, z_B) := (l, i, g, d)$ ;
```

In order to compute the variable lifetimes, it is assumed that the multiplier reads the values of the operands during some initial cycle, say number  $I$ , and generates the result during cycle number  $I + t_m - 1$  (or sooner), so that this result can be stored at the end of cycle number  $I + t_m - 1$  and is available for any operation beginning at cycle number  $I + t_m$  (or later). As regards the variables  $x_A, z_A, x_B$  and  $z_B$ , in charge of passing values from one iteration step to the next (Algorithm 2.3), their initial values must be available from the first cycle up to the last cycle during which those values are used. At the end of the computation scheme execution they must be updated with their new values. The lifetime intervals are given in Table 2.1.

The definition of a minimum number of registers can be expressed as a graph coloring problem. For that, associate a vertex to every variable and draw an edge between two variables if their lifetime intervals are incompatible, which means that they have more than one common cycle. As an example, the lifetime intervals of  $j$  and  $k$  are compatible, while the lifetime intervals of  $b$  and  $d$  are not.

The following groups of variables have compatible lifetime intervals:

**Table 2.1** Lifetime intervals

$a$	[300, 901]
$j$	[1, 2]
$k$	[2, 3]
$l$	[3, <i>final</i> ]
$b$	[600, 901]
$h$	[900, 901]
$c$	[601, 602]
$d$	[602, <i>final</i> ]
$f$	[1200, 1501]
$i$	[901, <i>final</i> ]
$e$	[1500, 1501]
$g$	[1501, <i>final</i> ]
$x_A$	[ <i>initial</i> , 601]
$z_A$	[ <i>initial</i> , 601]
$x_B$	[ <i>initial</i> , 301]
$z_B$	[ <i>initial</i> , 1]

$z_B(\text{initial} \rightarrow 1), j(1 \rightarrow 2), k(2 \rightarrow 3), l(3 \rightarrow \text{final});$   
 $x_B(\text{initial} \rightarrow 301), b(600 \rightarrow 901), f(1200 \rightarrow 1501), g(1501 \rightarrow \text{final});$   
 $z_A(\text{initial} \rightarrow 601), c(601 \rightarrow 602), d(602 \rightarrow \text{final});$   
 $x_A(\text{initial} \rightarrow 601), h(900 \rightarrow 901), e(1500 \rightarrow 1501);$   
 $a(300 \rightarrow 901), i(901 \rightarrow \text{final}).$

Thus, the computing scheme can be executed with five registers, namely  $x_A, z_A, x_B, z_B$  and  $R$ :

```

cycle 1:  $z_B := x_A + z_A$ ; start  $x_A \cdot z_B$ ;
cycle 2:  $z_B := z_B^2$ ;
cycle 3:  $z_B := z_B^2$ ;
cycle 300:  $R := x_A \cdot z_B$ ;
cycle 301: start  $x_B \cdot z_A$ ;
cycle 600:  $x_B := x_B \cdot z_A$ ;
cycle 601:  $z_A := R + x_B$ ; start  $x_A \cdot z_A$ ;
cycle 602:  $z_A := z_A^2$ ;
cycle 900:  $x_A := x_A \cdot z_A$ ;
cycle 901:  $R := x_A^2$ ; start  $R \cdot x_B$ ;
cycle 1200:  $x_B := R \cdot x_B$ ;
cycle 1201: start  $x_P \cdot z_A$ ;
cycle 1500:  $x_A := x_P \cdot z_A$ ;
cycle 1501:  $(x_A, z_A, x_B, z_B) := (z_B, R, x_A + x_B, z_A)$  ;

```



## 2.5 Final Example

Each iteration step of Algorithm 2.3 consists of executing a computation scheme, either the preceding one when  $k_m - i = 0$ , or a similar one when  $k_m - i = 1$ . Thus, Algorithm 2.3 is equivalent to the following algorithm 2.4 in which sentences separated by commas are executed in parallel.

### Algorithm 2.4: Scalar product, projective coordinates (scheduled version)

```

 $x_A := 1; z_A := 0; x_B := x_p; z_B := 1;$ 
for i in 1 .. m loop
  if  $k_{m-i} = 0$  then
     $R := x_A \cdot z_B, z_B := x_A + z_A;$ 
     $z_B := z_B^2;$ 
     $z_B := z_B^2;$ 
     $x_B := x_B \cdot z_A;$ 
     $x_A := x_A \cdot z_A, z_A := R + x_B;$ 
     $z_A := z_A^2;$ 
     $x_B := R \cdot x_B, R := x_A^2;$ 
     $x_A := x_p \cdot z_A;$ 
     $x_B := x_A + x_B, x_A := z_B, z_A := R, z_B := z_A;$ 
  else
     $R := x_B \cdot z_A, z_A := x_B + z_B;$ 
     $z_A := z_A^2;$ 
     $z_A := z_A^2;$ 
     $x_A := x_A \cdot z_B;$ 
     $x_B := x_B \cdot z_B, z_B = R + x_A;$ 
     $z_B := z_B^2;$ 
     $x_A := R \cdot x_A, R := x_B^2;$ 
     $x_B := x_p \cdot z_B;$ 
     $x_A := x_B + x_A, x_B := z_A, z_B := R, z_A := z_B;$ 
  end if;
end loop;

```

The data processed by Algorithm 2.4 are  $m$ -bit vectors (polynomials of degree  $m$  over the binary field  $GF(2)$ ) and the computation resources are field multiplication, addition and squaring. Field addition amounts to bit-by-bit modulo 2 additions (XOR functions). On the other hand, VHDL models of computation resources executing field squaring and multiplication are available at the Authors' web page, namely *classic\_squarer.vhd* and *interleaved\_mult.vhd* (Chap. 13). The *classic\_squarer* component is a combinational circuit. The *interleaved\_mult* component reads and internally stores the input operands during the first cycle after detecting a positive edge on *start\_mult* and raises an output flag *mult\_done* when the multiplication result is available.

The operations executed by the multiplier are

$$x_A \cdot z_B, x_B \cdot z_A, x_A \cdot z_A, R \cdot x_B, x_P \cdot z_A, x_B \cdot z_B, R \cdot x_A, x_P \cdot z_B.$$

An incompatibility relation can be defined over the set of involved variables: two variables are incompatible if they are operands of a same operation. As an example,  $x_A$  and  $z_B$  are incompatible, as  $x_A \cdot z_B$  is one of the operations. The corresponding graph can be colored with two colors corresponding to the sets

$$\{x_A, x_B, x_P\} \text{ and } \{z_A, z_B, R\}.$$

The first set of variables can be assigned to the leftmost multiplier input and the other to the rightmost input.

The operations executed by the adder are

$$x_A + z_A, R + x_B, x_A + x_B, x_B + z_B, R + x_A, x_B + x_A.$$

The incompatibility graph can be colored with three colors corresponding to the sets

$$\{x_A, z_B\}, \{x_B, z_A\} \text{ and } \{R\}.$$

The first one is assigned to the leftmost adder input, the second to the rightmost input, and  $R$  to both inputs.

Finally, the operations realized by the squaring primitive are

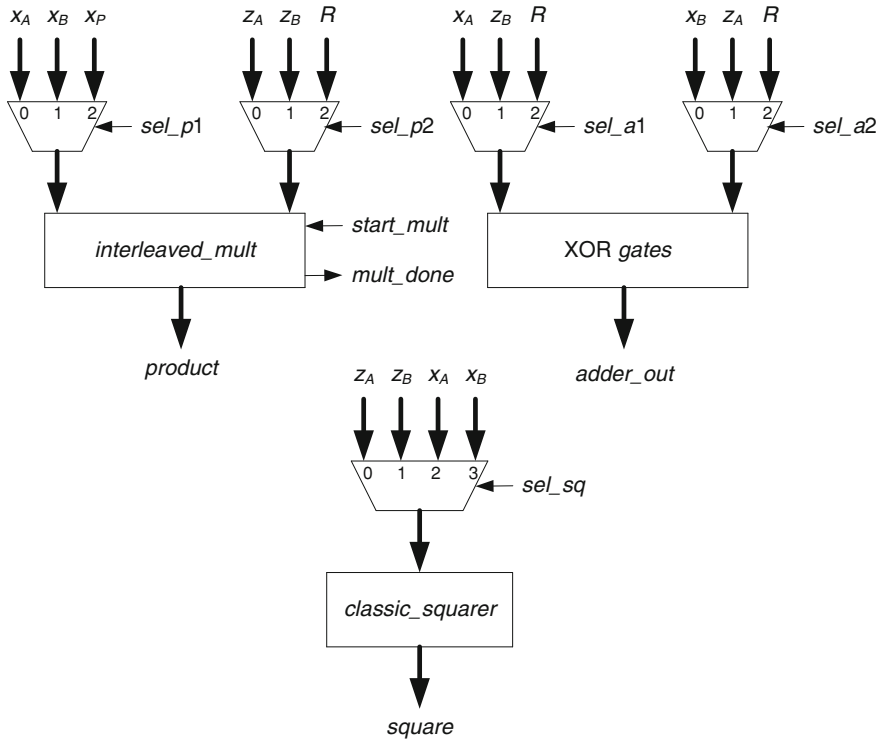
$$z_B^2, z_A^2, x_A^2, x_B^2.$$

The part of the data path corresponding to the computation resources and the multiplexers that select their input data is shown in Fig. 2.16. The corresponding VHDL model can easily be generated. As an example, the multiplier, with its input multiplexers, can be described as follows.

```
WITH sel_p1 SELECT
  mult1 <= xA WHEN "00", xB WHEN "01", xP WHEN OTHERS;
WITH sel_p2 SELECT
  mult2 <= zA WHEN "00", zB WHEN "01", R WHEN OTHERS;
a_mod_f_multiplier: interleaved_mult
PORT map (A => mult1, B => mult2, clk => clk,
  reset => reset, start => start_mult, Z => product,
  done => mult_done);
```

Consider now the storing resources. Assuming that  $x_P$  and  $k$  remain available during the whole algorithm execution, there remain five variables that must be internally stored:  $x_A$ ,  $x_B$ ,  $z_A$ ,  $z_B$  and  $R$ . The origin of the data stored in every register must be defined. For example, the operations that update  $x_A$  are

```
x_A := 1; x_A := x_A · z_A; x_A := x_P · z_A; x_A := z_B; x_A := x_A · z_B;
x_A := R · x_A; x_A := x_B + x_A;
```



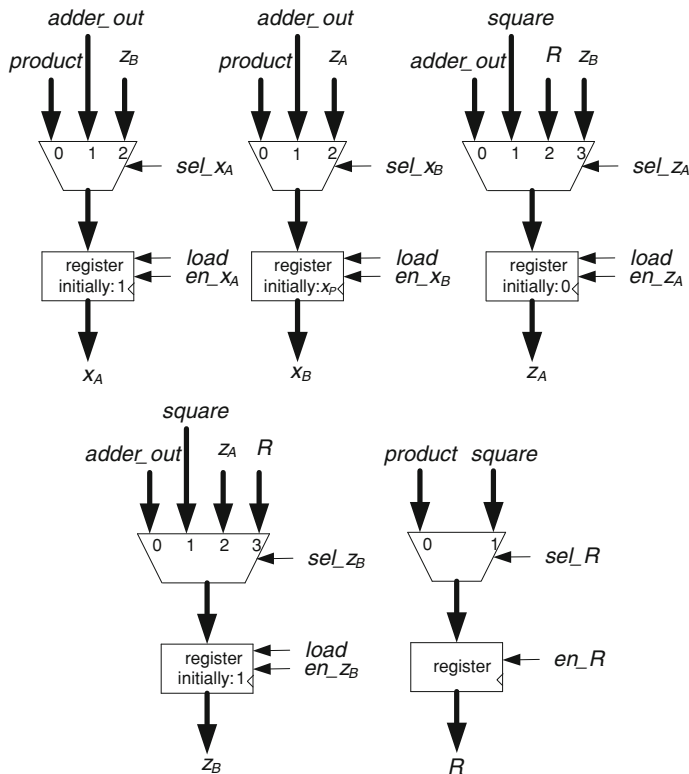
**Fig. 2.16** Example 2.4: computation resources

So, the updated value can be 1 (initial value), *product*, *adder\_out* or  $z_B$ . A similar analysis must be done for the other registers. Finally, the part of the data path corresponding to the registers and the multiplexers that select their input data is shown in Fig. 2.17. The corresponding VHDL model is easy to generate. As an example, the  $x_A$  register, with its input multiplexers, can be described as follows.

```

WITH sel_xA SELECT
  next_xA <=
    product WHEN "00", adder_out WHEN "01", zB WHEN OTHERS;
register_xA: PROCESS(clk)
BEGIN
  IF clk'EVENT AND clk = '1' THEN
    IF load = '1' THEN xA <= one;
    ELSIF en_xA = '1' THEN xA <= next_xA;
    END IF;
  END IF;
END PROCESS;

```



**Fig. 2.17** Example 2.5: data registers

A complete model of the data path *scalar\_product\_data\_path.vhd* is available at the Authors' web page.

The complete circuit is defined by the following entity.

```

ENTITY scalar_product IS
PORT (
    xP, k: IN STD_LOGIC_VECTOR(M-1 DOWNT0 0);
    clk, reset, start: IN STD_LOGIC;
    xA, zA, xB, zB: INOUT STD_LOGIC_VECTOR(M-1 DOWNT0 0);
    done: OUT STD_LOGIC
);
END scalar_product;
```

It is made up of

- the data path;
- a shift register allowing sequential reading of the values of  $k_m - i$ ;
- a counter for controlling the *loop* execution;

- a finite state machine in charge of generating all the control signals, that is *start\_mult*, *load*, *shift*, *en\_xA*, *en\_xB*, *en\_zA*, *en\_zB*, *en\_R*, *sel\_p1*, *sel\_p2*, *sel\_a1*, *sel\_a2*, *sel\_sq*, *sel\_xA*, *sel\_xB*, *sel\_zA*, *sel\_zB* and *sel\_R*. In particular, the control of the multiplier operations is performed as follows: the control unit generates a positive edge on the *start\_mult* signal, along with the values of *sel\_p1* and *sel\_p2* that select the input operands; then, it enters a wait loop until the *mult\_done* flag is raised (instead of waiting for a constant time, namely 300 cycles, as was done for scheduling purpose); during the wait loop the *start\_mult* is lowered while the *sel\_p1* and *sel\_p2* values are maintained; finally, it generates the signals for updating the register that stores the result. As an example, assume that the execution of the fourth instruction of the main loop, that is  $x_B := x_B \cdot z_A$ , starts at state 6 and uses identifiers *start4*, *wait4* and *end4* for representing the corresponding commands. The corresponding part of the next-state function is

```
WHEN 6 => current_state <= 7;
WHEN 7 => IF mult_done = '1' THEN
    current_state <= 8; END IF;
```

and the corresponding part of the output function is

```
WHEN 6 => command <= start4;
WHEN 7 => IF mult_done = '0' THEN command <= wait4;
    ELSE command <= end4; END IF;
```

- a command decoder ([Chap. 4](#)). Command identifiers have been used in the definition of the finite state machine output function, so that a command decoder must be used to generate the actual control signal values in function of the identifiers. For example, the command *start4* initializes the execution of  $x_B := x_B \cdot z_A$  and is decoded as follows:

```
WHEN start4 =>
    start_mult <= '1'; load <= '0'; shift <= '0';
    en_xA <= '0'; en_xB <= '0'; en_zA <= '0'; en_zB <= '0';
    en_R <= '0'; sel_p1 <= "01"; sel_p2 <= "00";
    sel_a1 <= "00"; sel_a2 <= "00"; sel_sq <= "00";
    sel_xA <= "00"; sel_xB <= "00"; sel_zA <= "00";
    sel_zB <= "00"; sel_R <= '0';
```

In the case of operations such as the first of the main loop, that is  $R := x_A \cdot z_B$ ,  $z_B := x_A + z_A$ , the 1-cycle operation  $z_B := x_A + z_A$  is executed in parallel with the final cycle of  $R := x_A \cdot z_B$  and not in parallel with the initial cycle. This makes the algorithm execution a few cycles (3) longer, but this is not significant as  $t_m$  is generally much greater than 3. Thus, the control signal values corresponding to the identifier *end1* are:

```

WHEN endl =>
  start_mult <= '0'; load <= '0'; shift <= '0';
  en_xA <= '0'; en_xB <= '0'; en_zA <= '0'; en_zB <= '1';
  en_R <= '1'; sel_p1 <= "00"; sel_p2 <= "00";
  sel_a1 <= "00"; sel_a2 <= "01"; sel_sq <= "00";
  sel_xA <= "00"; sel_xB <= "00"; sel_zA <= "00";
  sel_zB <= "00"; sel_R <= '0';

```

The control unit also detects the *start* signal and generates the *done* flag. A complete model *scalar\_product.vhd* is available at the Authors' web page.

### Comment 2.1

The *interleaved\_mult* component is also made up of a data path and a control unit, while the *classic\_squarer* component is a combinational circuit. An alternative solution is the definition of a data path able to execute all the operations, including those corresponding to the *interleaved\_mult* and *classic\_squarer* components. The so-obtained circuit could be more efficient than the proposed one as some computation resources could be shared between the three algorithms (field multiplication, squaring and scalar product). Nevertheless, the hierarchical approach consisting of using pre-existing components is probably safer and allows a reduction in the development times.

Instead of explicitly disassembling the circuit into a data path and a control unit, another option is to describe the operations that must be executed at each cycle, and to let the synthesis tool define all the details of the final circuit. A complete model *scalar\_product\_DF2.vhd* is available at the Authors' web page.

### Comment 2.2

Algorithm 2.4 does not compute the scalar product  $kP$ . A final step is missing:

```

if zB = 0 then xA := xP; yA := xP + yP;
else
  xA := xA / zA;
  xB := xB / zB;
  yA := ((xA + xP) [(xA + xP) (xB + xP) + xP2 + yP] / xP) +
  yP;
end if;
xR := xA; yR := yR;

```

The design of a circuit that executes this final step is left as an exercise.

## 2.6 Exercises

1. Generate several VHDL models of a 7-to-3 counter. For that purpose use the three options proposed in [Sect. 2.3.1](#).
2. Generate the VHDL model of a circuit executing the final step of the scalar product algorithm (Comment 2.2). For that purpose, the following entity, available at the Authors' web page, is used:

```
entity binary_algorithm_polynomials is
port(
  g, h: in std_logic_vector(M-1 downto 0);
  clk, reset, start: in std_logic;
  z: out std_logic_vector(M-1 downto 0);
  done: out std_logic
);
end binary_algorithm_polynomials;
```

It computes  $z = g \cdot h^{-1}$  over  $GF(2^m)$ . Several architectures can be considered.

3. Design a circuit to compute the greatest common divisor of two natural numbers, based on the following simplified Euclidean algorithm.

```
while a ≠ b loop
  if a > b then a := a - b;
  else b := b - a;
end loop;
```

4. Design a circuit for computing the greatest common divisor of two natural numbers, based on the following Euclidean algorithm.

```
while b > 0 loop
  (a, b) := (b, a mod b);
end loop;
gcd := a;
```

5. The distance  $d$  between two points  $(x_1, y_1)$  and  $(x_2, y_2)$  of the  $(x, y)$ -plane is equal to  $d = ((x_1 - x_2)^2 + (y_1 - y_2)^2)^{0.5}$ . Design a circuit that computes  $d$  with only one subtractor and one multiplier.
6. Design a circuit that, within a three-dimensional space, computes the distance between two points  $(x_1, y_1, z_1)$  and  $(x_2, y_2, z_2)$ .
7. Given a point  $(x, y, z)$  of the three-dimensional space, design a circuit that computes the following transformation.

$$\begin{bmatrix} x_t \\ y_t \\ z_t \end{bmatrix} = \begin{bmatrix} a_{11} & a_{21} & a_{31} \\ a_{21} & a_{22} & a_{32} \\ a_{31} & a_{32} & a_{11} \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

8. Design a circuit for computing  $z = e^x$  using the formula

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

9. Design a circuit for computing  $x^n$ , where  $n$  is a natural, using the following relations:  $x^0 = 1$ ; if  $n$  is even then  $x^n = (x^{n/2})^2$ , and if  $n$  is odd then  $x^n = x \cdot (x^{(n-1)/2})^2$ .
10. Algorithm 2.4 (scalar product) can be implemented using more than one *interleaved\_multiplier*. How many multipliers can operate in parallel? Define the corresponding schedule.

## References

1. Hankerson D, Menezes A, Vanstone S (2004) Guide to elliptic curve cryptography. Springer, New York
2. Deschamps JP, Imaña JL, Sutter G (2009) Hardware implementation of finite-field arithmetic. McGraw-Hill, New York
3. López J, Dahab R (1999) Improved algorithm for elliptic curve arithmetic in  $GF(2^n)$ . Lect Notes Comput Sci 1556:201–212



Guide to FPGA Implementation of Arithmetic Functions

Deschamps, J.-P.; Sutter, G.D.; Cantó, E.

2012, XVI, 472 p., Hardcover

ISBN: 978-94-007-2986-5