The design of embedded systems, that is, circuits designed for specific applications, is based on a series of decisions as well as on the use of several types of development techniques. For example:

- Selection of the data representation
- Generation or selection of algorithms
- Selection of hardware platforms
- Hardware–software partitioning
- Program generation
- New hardware synthesis
- Cosimulation, coemulation, and prototyping

Some of these activities have a close relationship with the study of arithmetic algorithms and circuits, especially in the case of systems including a great amount of data processing (e.g., ciphering and deciphering, image processing, digital signature, biometry).

1.1 NUMBER REPRESENTATION

When using general-purpose equipment, the designer has few possible choices concerning the internal representation of data. He must conform to some fixed
and predefined data types such as integer, floating-point, double precision, and character. On the contrary, if a specific system is under development, the designer can choose, for each data, the most convenient type of representation. It is no longer necessary to choose some standard fixed-point or floating-point numeration system. Nonstandard specific formats can be used. In Chapter 3 the main number representation methods will be defined.

1.2 ALGORITHMS

Every complex data processing operation must be decomposed into simpler operations — the computation primitives — executable either by the main processor or by some specific coprocessor. The way the computation primitives are used in order to perform the complex operation is what is meant by algorithm. Obviously, knowledge of algorithms is of fundamental importance for developing arithmetic procedures (software) and circuits (hardware). It is the topic of Chapters 4–8.

1.3 HARDWARE PLATFORMS

The selection of a hardware platform is based on the answer to the following question. How do we get the desired behavior at the lowest cost, while fulfilling some additional constraints? As a matter of fact, the concept of cost must be carefully defined in each particular case. It can cover several aspects: for example, the unit production cost, the nonrecurring engineering costs, and the implicit cost for a late introduction of the product to the market. Some examples of additional technical constraints are the size of the system, its power consumption, and its reliability and maintainability.

For systems requiring little data processing capability, microcontrollers and low-range microprocessors can be the best choice. If the computation needs are greater, more powerful microprocessors, or even digital signal processors (DSPs), should be considered. This type of solution (microprocessors and DSPs) is very flexible as the development work mainly consists in generating programs.

For getting higher performances, it may be necessary to develop specific circuits. A first option is to use a programmable device, for example, a field-programmable gate array (FPGA). It could be an interesting option for prototypes and small series. For greater series, an application-specific integrated circuit (ASIC) should be developed. ASIC vendors offer several types of products: for example, gate arrays, with relatively small prototyping costs, or standard cell libraries, integrating a complete system-on-chip (SOC) including processors, program memories, data memories, logic, macrocells, and analog interfaces.

A brief presentation of the most common hardware platforms is given in Chapter 9.
1.4 HARDWARE–SOFTWARE PARTITIONING

The hardware–software partitioning consists of deciding which operations will be executed by the central processing unit (the software) and which ones by specific coprocessors (the hardware). As a matter of fact, the platform selection and the hardware–software partitioning are tightly related operations. For systems requiring little data processing capability, the whole system is implemented in software. If higher performances are necessary, the noncritical operations, as well as control of the operation sequence, are executed by the central processing unit, while the critical ones are implemented within specific coprocessors.

1.5 SOFTWARE GENERATION

The operations belonging to the software block of the chosen partition must be programmed. In Chapters 4–8 the algorithms are presented in an Ada-like language that can easily be translated to C or even to the assembly language of the chosen microprocessor.

1.6 SYNTHESIS

Once the hardware–software partition has been defined, all the tasks assigned to the specific hardware (FPGA, ASIC) must be translated into circuit descriptions. Some important synthesis principles and methods are described in Chapter 10. The synthesis of arithmetic circuits, based on the algorithms of Chapters 4–8, is the topic of Chapters 11–15, and an additional chapter (16) is dedicated to the implementation of floating-point arithmetic.

1.7 A FIRST EXAMPLE

Common examples of application fields resorting to embedded solutions are cryptography, access control, smart cards, automotive, avionics, space, entertainment, and electronic sales outlets. In order to illustrate the main steps of the design process, a small digital signature system will now be developed (complete assembly language and VHDL code available).

1.7.1 Specification

The system under development (Figure 1.1) has three inputs,

- `character` is an 8-bit vector.
- `new_character` is a signal used for synchronizing the input of successive characters.
- `sign` is a control signal ordering the computation of a digital signature.
and two outputs,

- `done` is a status variable indicating that the signature computation has been completed,
- `signature` is a 32-bit vector, namely, the signature of the message.

The working of the system is shown in Figure 1.2: a sequence \( c_1, c_2, \ldots, c_n \) of any number \( n \) of characters (the message), synchronized by the signal `new_character`, is inputted. When the `sign` control signal goes high, the `done` flag is lowered and the signature of the message is computed. The `done` flag will be raised as soon as the signature \( s \) is available.

In order to sign the message two functions must be defined:

- a hash function associating a 32-bit vector (the `summary`) to every message, whatever its length;
- an encode function computing the signature corresponding to the summary.

The following (naive) hash function is used:

\[
\text{Algorithm 1.1 Hash Function}
\]

\[
\text{summary}:=0; \\
\text{while not(end_of_message) loop} \\
\quad \text{get(character);} \\
\quad a:=(\text{summary}(7 \text{ downto } 0)+\text{character}) \mod 256; \\
\quad \text{summary}(23 \text{ downto } 16):=\text{summary}(31 \text{ downto } 24); \\
\quad \text{summary}(15 \text{ downto } 8):=\text{summary}(23 \text{ downto } 16);
\]

![Figure 1.1 System under development.](image1.png)

![Figure 1.2 Input and output signals.](image2.png)
As an example, assume that the message is the following (every character can be equivalently considered as an 8-bit vector or a natural number smaller than 256, i.e. a base-256 digit; see Chapter 3):

12, 45, 216, 1, 107, 55, 10, 9, 34, 72, 215, 114, 13, 13, 229, 18.

The summary is computed as follows:

\[
\text{summary} = (0, 0, 0, 0),
\text{summary} = (12, 0, 0, 0),
\text{summary} = (45, 12, 0, 0),
\text{summary} = (216, 45, 12, 0),
\text{summary} = (1, 216, 45, 12),
\text{summary} = (119, 1, 216, 45),
\text{summary} = (100, 119, 1, 216),
\text{summary} = (226, 100, 119, 1),
\text{summary} = (10, 226, 100, 119),
\text{summary} = (153, 10, 226, 100),
\text{summary} = (172, 153, 10, 226),
\text{summary} = (185, 172, 153, 10),
\text{summary} = (124, 185, 172, 153),
\text{summary} = (166, 124, 185, 172),
\text{summary} = (185, 166, 124, 185),
\text{summary} = (158, 185, 166, 124),
\text{summary} = (142, 158, 185, 166).
\]

The final result, translated from the base-256 to the decimal representation, is

\[
\text{summary} = 142 \times 256^3 + 158 \times 256^2 + 185 \times 256 + 166 = 2392766886.
\]

The encode function computes

\[
\text{encode}(y) = y^x \mod m
\]

\(x\) being some private key, and \(m\) a 32-bit number. Assume that

\[x = 1937757177 \quad \text{and} \quad m = 2^{32} - 1 = 4294967295.\]
Then the signature of the previous message is

\[ s = (2392766886)^{1937757177} \mod 4294967295 = 37998786. \]

1.7.2 Number Representation

In this example all the data are either 8-bit vectors (the characters) or 32-bit vectors (the summary, the key, and the module \( m \)). So instead of representing them in the decimal numeration system, they should be represented in the binary or, equivalently, the hexadecimal system. The message is

0C, 2D, D8, 01, 6B, 37, 0A, 09, 22, 48, D7, 72, 0D, 0D, E5, 12.

The summary, the key, the module, and the signature are

\[
\begin{align*}
\text{summary} &= 8E9EB9A6, \\
\text{private key} &= 737FD3F9, \\
m &= FFFFFFFF, \\
s &= 0243D0C2.
\end{align*}
\]

1.7.3 Algorithms

The hash function amounts to a mod-256 addition, that is, a simple 8-bit addition without output carry. The only complex operation is the mod \( m \) exponentiation.

Assume that \( x, y, \) and \( m \) are \( n \)-bit numbers. Then

\[ x = x(0) + 2.x(1) + \ldots + 2^{n-1}.x(n - 1), \]

and \( e \) can be written in the form

\[ e = (((\ldots ((1^2.y)^{x(n-1)})^2.y^{x(n-2)})^2\ldots)^2.y^{x(1)})^2.y^{x(0)} \mod m. \]

The corresponding algorithm is the following (Chapter 8, Algorithm 8.14).

**Algorithm 1.2 Exponentiation**

```plaintext
e := 1;
for i in 1..n loop
  e := (e*e) mod m;
  if x(n-i)=1 then e := (e*y) mod m; end if;
end loop;
```

The only computation primitive is the modulo \( m \) product, which, in turn, is equivalent to a natural multiplication followed by a modulo \( m \) reduction, that is, an integer division by \( m \). The following algorithm (Chapter 8, Algorithm 8.5)
computes \( r = x \cdot y \mod m \). It uses two procedures: \textit{multiply}, which computes the product \( z \) of two natural numbers \( x \) and \( y \), and \textit{divide}, which generates \( q \) (the quotient) and \( r \) (the remainder) such that \( z = q \cdot m + r \) with \( r < m \).

\medskip

**Algorithm 1.3 Modulo \( m \) Multiplication**

\begin{verbatim}
multiply (x, y, z);
divide (z, m, q, r);
\end{verbatim}

A classical method for computing the product \( z \) of two natural numbers \( x \) and \( y \) is the \textit{shift and add} algorithm (Chapter 5, Algorithm 5.3). In base 2:

\medskip

**Algorithm 1.4 Natural Multiplication**

\begin{verbatim}
p(0):=0;
for i in 0..n-1 loop
    p(i+1):=(p(i)+x(i)\cdot y)/2;
end loop;
z:=p(n)\cdot (2**n);
\end{verbatim}

For computing \( q \) and \( r \) such that \( z = q \cdot m + r \) with \( r < m \), the classical restoring division algorithm can be used (Chapter 6, Algorithms 6.1 and 6.2). Given \( x \) and \( y \) (the operands) such that \( x < y \), and \( p \) (the desired precision), the restoring division algorithm computes \( q \) and \( r \) such that

\[ x \cdot 2^p = q \cdot y + r. \tag{1.1} \]

Within the exponentiation algorithm 1.2, the operands \( e \) and \( y \) are \( n \)-bit numbers. Furthermore, \( e \) is always smaller than \( m \), so that both products \( z = e \cdot e \) or \( z = e \cdot y \) are \( 2 \cdot n \)-bit numbers satisfying the relation

\[ z < m \cdot 2^n. \]

Thus by substituting \( x \) by \( z \), \( p \) by \( n \), and \( y \) by \( m \cdot 2^n \) in (1.1), the restoring division algorithm computes \( q \) and \( r' \) such that

\[ z \cdot 2^n = q \cdot (m \cdot 2^n) + r' \quad \text{with} \quad r' < m \cdot 2^n, \]

that is,

\[ z = q \cdot m + r \quad \text{with} \quad r = r' \cdot 2^{-n} < m. \]

The restoring algorithm is similar to the pencil and paper method. At every step the latest obtained remainder, say, \( r(i - 1) \), is multiplied by 2 and compared with the divider \( y \). If \( 2 \cdot r(i - 1) \) is greater than or equal to \( y \), then the new remainder is


\[ r(i) = 2r(i - 1) - y \] and the corresponding quotient bit is equal to 1. In the contrary case, the new remainder is \( r(i) = 2r(i - 1) \) and the corresponding quotient bit equal to 0. The initial remainder \( r(0) \) is the dividend.

**Algorithm 1.5  Restoring Division**

\[
\begin{align*}
& r(0) := z; \quad y := m * (2^n); \\
& \text{for } i \text{ in } 1..n \text{ loop} \\
& \quad \text{if } 2r(i-1) - y < 0 \text{ then } q(i) := 0; \ r(i) := 2r(i-1); \ \text{else} \\
& \quad \quad q(i) := 1; \ r(i) := 2r(i-1) - y; \ \text{end if;}
& \text{end loop;}
& r := r(n)/(2^n);
\end{align*}
\]

By merging Algorithms 1.4 and 1.5, the following modular product algorithm is obtained.

**Algorithm 1.6  Modular Product**

\[
\begin{align*}
& p(0) := 0; \\
& \text{for } i \text{ in } 0..n-1 \text{ loop} \\
& \quad p(i+1) := (p(i) + x(i)*y)/2; \\
& \text{end loop;}
& r(0) := p(n) * (2^n); \quad y := m * (2^n); \\
& \text{for } i \text{ in } 1..n \text{ loop} \\
& \quad \text{if } 2r(i-1) - y < 0 \text{ then } q(i) := 0; \ r(i) := 2r(i-1); \ \text{else} \\
& \quad \quad q(i) := 1; \ r(i) := 2r(i-1) - y; \ \text{end if;}
& \text{end loop;}
& r := r(n)/(2^n);
\end{align*}
\]

Observe that the multiplication of \( p(n) \) and \( m \) by \( 2^n \), as well as the division of \( r(n) \) by \( 2^n \) can be deleted. Then \( r(0) = p(n) \) is a \( 2^n \)-bit fixed-point number (Chapter 3) smaller than \( 2^n \) and the divider is equal to \( m \). The quotient \( q \) and the remainder \( r(n) \) satisfy the relation \( p(n) \cdot 2^n = q \cdot m + r(n) \) so that \( r = r(n) \).

**1.7.4  Hardware Platform**

For implementing this illustrative example, a prototyping board will be used, namely, an XSA-100 board from XESS Corporation. It includes an XC2S100 FPGA (Spartan-II family of Xilinx) integrating the complete digital signature system. The design environment includes virtual components (synthesizable VHDL models, Chapter 9), among others PicoBlaze, an 8-bit microprocessor, and its program memory ([XIL2002]).

**1.7.5  Hardware–Software Partitioning**

As mentioned above, the only complex operation is the computation of \( y^x \) modulo \( m \). All the other operations can be carried out by the processor. The corresponding system architecture is shown in Figure 1.3. It works as follows:
PicoBlaze reads the character input at address 0 and the command input at address 1, where

\[
\text{command} = 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 
\]

sign new_character.

- It computes the 32-bit summary and writes it, under the form of four separate bytes,

\[
\text{summary} = Y(3) \ Y(2) \ Y(1) \ Y(0),
\]

into four registers whose addresses are 3, 2, 1 and 0, respectively.

- A specific coprocessor receives the start signal from PicoBlaze at address 4, computes

\[
s = \text{(summary)}^{737FD3F9} \mod \text{FFFFFFF},
\]

and generates the done flag.

### 1.7.6 Program Generation

The program executed by PicoBlaze is made up of three parts (assembly language code available):

- reading of the new_character and sign input signals,
- reading of the character input and updating of the summary,
- writing of the summary and of the start command within the interface registers:
summary:=(0, 0, 0, 0);
start:=0;
loop
   --wait for command=0
   while command>0 loop null; end loop;
   --wait for command=1 (new_character) or 2 (sign)
   while command=0 loop null; end loop;
   if command=1 then
      a:=(summary(0)+character) mod 256;
      summary(0):=summary(1);
      summary(1):=summary(2);
      summary(2):=summary(3);
      summary(3):=a;
   elsif command=2 then
      Y(3):=summary(3);
      Y(2):=summary(2);
      Y(1):=summary(1);
      Y(0):=summary(0);
      start:=1;
      summary:=(0, 0, 0, 0);
      start:=0;
   end if;
end loop

1.7.7 Synthesis

The synthesis (complete VHDL code available) of the exponentiator block of Figure 1.3 is based on the algorithms of Section 1.7.3. A summary of the main principles for translating an algorithm to a circuit is given in Chapter 10. The data path of Figure 1.4 allows executing Algorithm 1.2. It includes:

- two 32-bit registers: a parallel register storing e, and a loadable shift register, initially storing x and allowing to successively read the value of x(n−1), x(n−2),...,x(0);

- a mod m multiplier with a start input signal and a done output flag;

- a 32-bit 2-to-1 multiplexer selecting either e or y as the second multiplier operand.

The complete circuit is described by the following VHDL model (including the control unit):

entity exponentiator is
   port ( 
      x, y, m: in std_logic_vector(n-1 downto 0);
      z: inout std_logic_vector(n-1 downto 0);
      clk, reset, start: in std_logic;
      done: out std_logic
   );
end exponentiator;
architecture circuit of exponentiator is
   component sequential_mod_mult..end component;
signal start_mult, sel_y, done_mult: std_logic;
signal reg_x, input_y, output_z: std_logic_vector(n-1 downto 0);
subtype step_number is natural range 0 to n;
signal count: step_number;
subtype internal_states is natural range 0 to 14;
signal state: internal_states;
begin
label_1: sequential_mod_mult port map(z, input_y, m, output_z, clk, reset, start_mult, done_mult);
with sel_y select input_y<=z when '0', y when others;
process (clk, reset)
begin
if reset='1' then
   state<=0; done<='0'; start_mult<='0'; count<=0;
elsif clk'event and clk='1' then
   case state is
   when 0=>if start='0' then state<=state+1; end if;
   (e)
end process;
end if;
end process;
end label_1;

Figure 1.4 Exponentiator.
when 1 => if start='1' then state<=state+1; end if;
when 2 => z<=conv_std_logic_vector(1, n);
    reg_x<x; count<=0; done<='0'; state<=state+1;
when 3 =>
    sel_y<='0'; start_mult<='1'; state<=state+1;
when 4 => state<=state+1;
when 5 => start_mult<='0'; state<=state+1;
when 6 =>
    if done_mult='1' then state<=state+1; end if;
when 7 => z<=output_z;
    if reg_x(n-1)='1' then state<=state+1;
    else state<=13; end if;
when 8 =>
    sel_y<='1'; start_mult<='1'; state<=state+1;
when 9 => state<=state+1;
when 10 => start_mult<='0'; state<=state+1;
when 11 =>
    if done_mult='1' then state<=state+1; end if;
when 12 => z<=output_z; state<=state+1;
when 13 => reg_x(0)<=reg_x(n-1);
    for i in 1 to n-1 loop reg_x(i)<=reg_x(i-1);
    end loop;
    count<=count+1; state<=state+1;
when 14 =>
    if count>=n then done<='1'; state<=0;
    else state<=3; end if;
end case;
end if;
end process;
end circuit;

1.7.8 Prototype

All the files (complete source files available) necessary for programming an XSA-100 board are included in the file section1_7.zip:

- exponentiator.vhd is the complete description of the exponentiation circuit (including the modular multiplier model);
- signatu.psm is the assembly language program;
- kpcsm.vhd is the PicoBlaze model;
- signatu.vhd is the program memory model generated from the assembly language program with kcpsm.exe (the PicoBlaze assembler released by Xilinx [XIL2002]).

In order to test the complete system, the circuit of Figure 1.5 has been synthesized. It is made up of:

- the circuit of Figure 1.3 including PicoBlaze, its program memory, the interface registers, and the exponentiator;
• a finite state machine generating the commands and characters corresponding to the example of Section 1.7.1;

• a circuit that interfaces the board with signals $d(7..0)$ controllable from the host computer ([XSA2002]):

  - $d(7)$ cannot be used,
  - $d(3..0)$ are used for selecting one of the outputs (out_0 to out_15) or inputs (in_0 to in_15),
  - $d(6..4)$ are control signals,

<table>
<thead>
<tr>
<th>$d(6..4)$</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>nop</td>
</tr>
<tr>
<td>001</td>
<td>write</td>
</tr>
<tr>
<td>010</td>
<td>read</td>
</tr>
<tr>
<td>011</td>
<td>reset</td>
</tr>
<tr>
<td>100</td>
<td>address strobe</td>
</tr>
</tbody>
</table>

![Figure 1.5 Prototype.](image-url)
in this application the write and address strobe commands are not used; when the read command is active, the hexadecimal representation of the 4-bit vector selected with $d(3..0)$ is displayed on the LED of the board;

- the 7-segment LED decoder.

The VHDL model of the circuit of Figure 1.5 (firma.vhd) is also included in section1_7.zip as well as the file describing the pin assignment (pines.ucf). The whole system (Figure 1.5) can be synthesized with ISE, the synthesis program of Xilinx, and downloaded to the XSA-100 board.

1.8 BIBLIOGRAPHY


