Chapter 9
Introduction to FPGA and HDL Design

Abstract The last chapter deals with the practical implementation in hardware of systems similar to the ones presented in previous chapters and tested by simulation only. The devices that host the projects are Field-Programmable Gate Arrays (FPGAs), inserted on commercially available boards and managed by Deeds and proprietary tools. A short description of the devices and the associated tools is presented. An original, hands-on introduction of the VHDL hardware description language is included. A few exercises of digital system design and prototyping complete the chapter.

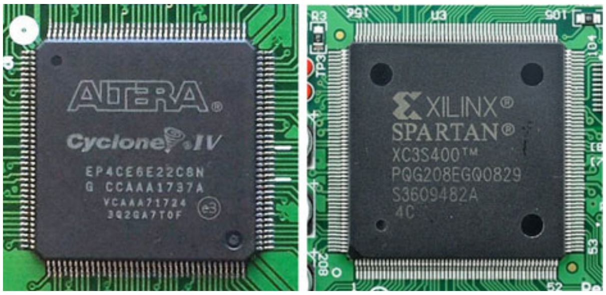
The reader that has successfully followed the book is now familiar with the issues at the bases of digital systems and able to practice with their design and simulation. Projects and examples presented in the previous chapters were targeted more to understanding and less to practical implementation, since the former is an essential skill for the designer, upon which the latter is based. Furthermore, circuit implementation is strongly dependent on the state of the art of microelectronic technologies and subject to a rapid evolution and inevitable obsolescence.

The networks presented in this chapter are similar, as for their approach and complexity, to the ones already studied, with the difference that the work will go all the way to physical implementation and testing.

9.1 Field-Programmable Gate Arrays

Physical implementation will be based on components called Field-Programmable Gate Array (FPGA). The figure in the next page shows the visual appearance of two FPGA, by *Intel/Altera FPGA* (ex *Altera Corporation*), on the left, and by *XilinX*, on the right.

G. Donzellini et al., *Introduction to Digital Systems Design*,
https://doi.org/10.1007/978-3-319-92804-3_9



An FPGA is a chip that contains a large quantity of basic logical elements, such as gates and flip-flops (and quite a lot of more complex circuits) that can be wired together to form the system thanks to a matrix of switches. The connections are created by using development tools provided by the FPGA's manufacturers, downloaded in the chip and kept alive in a memory.

FPGAs are the youngest child within the large family of programmable logic devices (PLDs), a term that designates all the chips that can be programmed, i.e., specialized for a given application by establishing or changing their inside connections, during production or in the field. PLD must not be confused with devices, as microcomputers, whose hardware is fixed and programming means execution of external instructions. PLD has changed deeply, since the 1980s, design and implementation of complex systems.

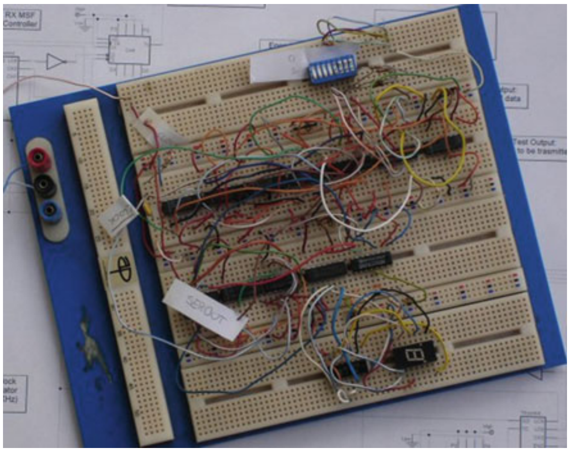
FPGAs have a great value for the educational field, too, since they lend themselves very well for the fast and inexpensive realization of working prototypes of systems designed for learning purposes.

9.1.1 System Prototyping and FPGA

In a not too distant past, circuit prototyping implied the connection (by soldered wires) of many discrete components. That process was extremely time consuming and very sensitive to mistakes in the connection or bad contacts with the wires to a point that it was not easy to understand if the malfunctioning of the system was due to design mistakes of faulty connections.

It was common the use, in the laboratories, of solderless breadboards, with a fixed grid of holes partially connected together, in which students inserted components, usually in the form of integrated circuits (ICs) and established connections through wires. The ICs made available gates, flip-flops, and a wide variety of combinational and sequential blocks: the circuit implementation was therefore the same, or very close, to the hardware structure designed with *Deeds*.

For instance, below is a picture of the breadboard of an 8-bit parallel to serial converter, where the system is built with standard ICs implementing the functions of gates, flip-flops, registers and counters.



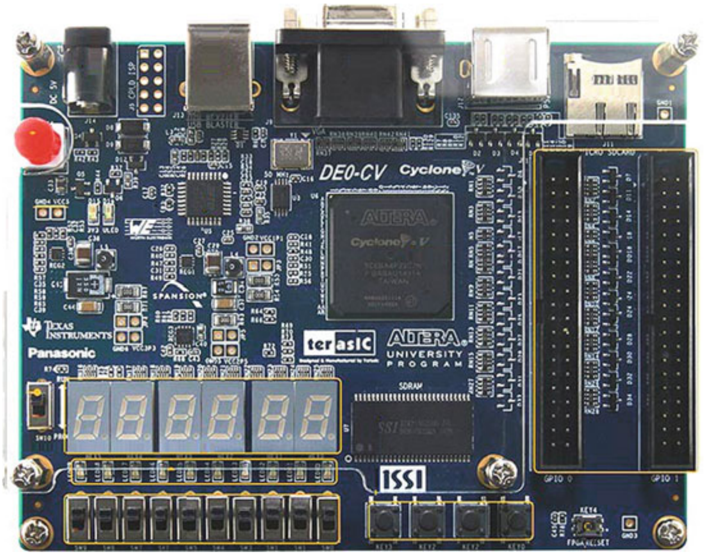
The problems with breadboarding digital systems are the same mentioned when presenting the traditional prototyping. There are a few advantages: it is easier to change connections and the risk that students burn their finger with the soldering iron disappears. The problem of faulty contacts is even worse than in the previous method. Solderless breadboards are still useful for rapid prototyping of system whose core is a FPGA or a microcontroller: the board can host simple interface or ancillary circuits around the core.

Nowadays are available FPGA-based prototyping board, especially suited for educational purposes. They include several input/output interfaces, allowing the implementation of system prototypes without the addition of components outside the board.

9.1.2 FPGA Board Examples

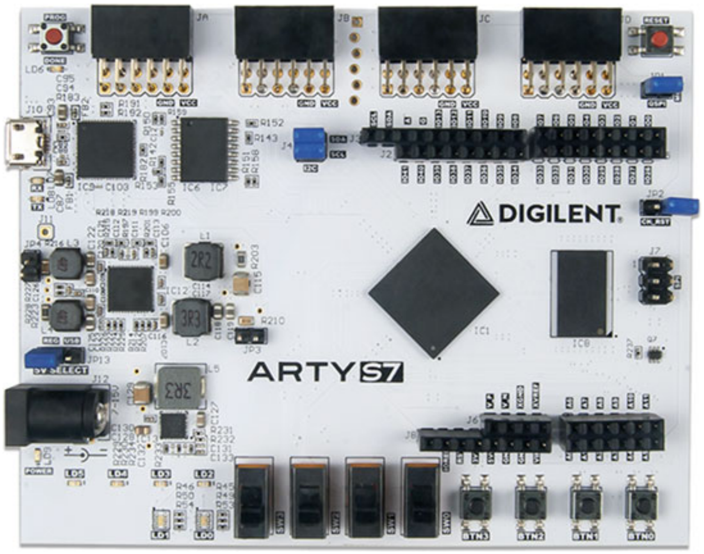
A wide variety of FPGA boards are commercially available, with performances continuously evolving. They are targeted to different applications, from simple and inexpensive boards ideal for educational purposes to complex, high-speed boards for professional designs. It is not our intention to go into the details of the boards. The experimenters or the designers can find on the market the most suitable for their application, paying attention to their performances, available software and, last but not least, budget.

As an example, we present here a general description of a few FPGA boards, suitable for the implementation of the systems developed in the book. It is worth to notice that each of them has the capacity to host much more complex systems and, therefore, to allow a natural transition toward professional design. The figure below shows the board DE0-CV, produced by *Terasic/Altera*. The DE0-CV is supported by *Deeds* environment.



On the board are available, as the picture shows, push-buttons, switches, LEDs, seven-segment displays, connectors, and other devices. The heart is the FPGA, the big black square in the center of the board, a device from *Intel/Altera FPGA*, member of the family "Cyclone® V FPGA." The chip contains a matrix of more than 40,000 logic units (the basic FPGA block that will be explained later in this chapter), 60,000 flip-flops, and a microprocessor *ARM CortexTM* dual-core, the same used in many mobile phones.

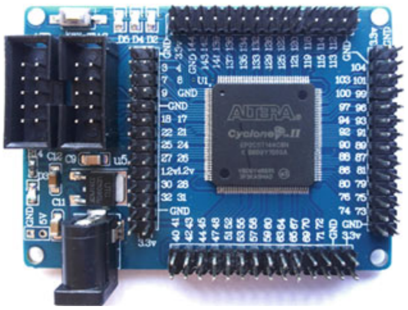
In the next page is another example: the ARTY S7-50 board, produced by *Digilent*, using an FPGA chip from the *Spartan®-7* family, produced by *XilinX*. The FPGA chip is placed at 45 degrees with respect to the board's boundaries.



As in the previous board, push-buttons, switches, LEDs, and other interfaces are available. In particular, there are connectors designed to host the input/output and additional boards (shields) originally designed for *Arduino* microcontrollers. The FPGA contains more than 52,000 basic combinational blocks and more than 65,000 flip-flops. No microprocessor is included.

Note: it is possible to program the FPGA to implement a processor, by using the chip's resources (logic elements and flip-flops). This is called "soft processor" since it is assembled by software. It behaves exactly as an "hard" processor, i.e., one built as such on the silicon.

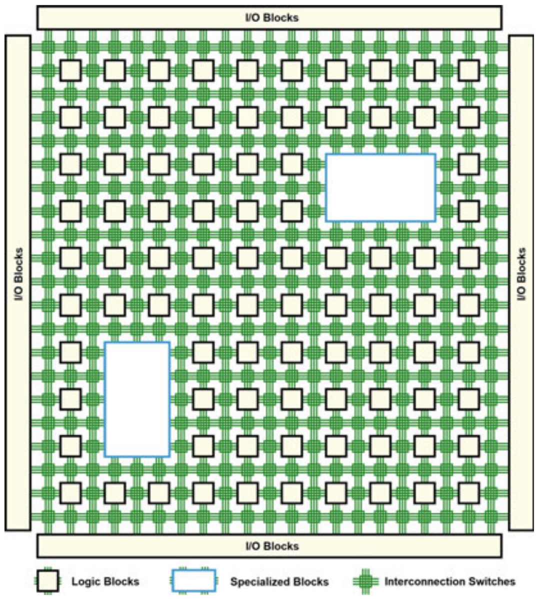
The last example is a very inexpensive FPGA board available online and supported by *Deeds*. It is based on an *Intel/Altera FPGA* chip "Cyclone® II." In addition to the four connectors placed around the chip, three LEDs and a push-button are available. The two 10-pin connectors on the left are used for programming the FPGA chip.



To implement our projects, which usually need more input/output devices, we must connect push-buttons, switches, displays, etc., to the four connectors. The chip is large enough to implement small microcomputers, like the ones available with *Deeds*. The resulting “soft processor” would use about one half of the 4,600 logic units and only 400 of the 4,600 flip-flop.

9.1.3 FPGA Architecture

FPGAs’ manufactures make available a wide variety of devices, classified by *families* varying by complexity and targeted to different application fields. Typical examples are audio/video signal processing, radar, automotive systems, and, generally speaking, all the applications that require high performances but do not have the volume to justify the cost of a full custom chip. In spite of the variety presented by the families, all FPGA devices have in common a basic architecture. An FPGA is essentially a large matrix of logic blocks, arranged by rows and columns, as in the figure below.

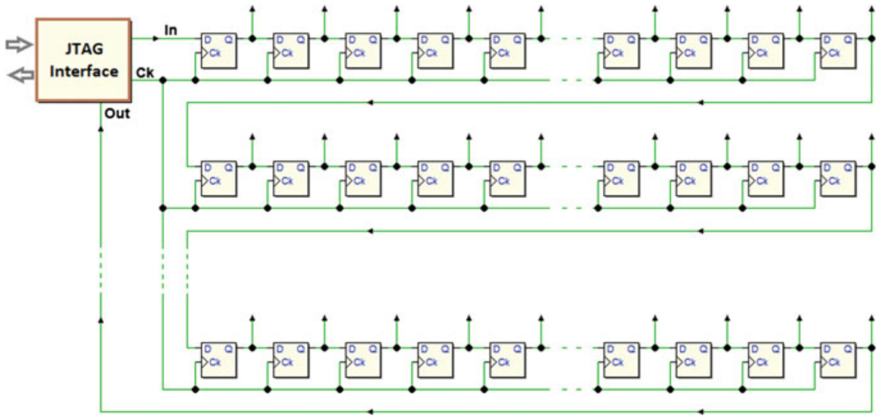


Each block contains one or more flip-flops and combinational networks. A matrix of programmable connections is spread through the chip, using the largest share of its area. Connections are, again, arranged by rows and columns: at every crossing electronic switches allow the individual junction of rows and columns. Such structure provides the interconnection among the blocks. Local submatrices may be available to improve the speed of communication among blocks physically close together.

Inside the matrix special blocks targeted to specific functions may be available, such as read/write memories (RAM), arithmetic circuits (very often multipliers), and others.

The matrix is surrounded along the four edges by other logic blocks (I/O Blocks), in charge of the interface of the chip with external devices.

The previous figure shows only the elements of the chip that are available for design, while hides the ones in charge of programming (configuring) logic blocks and connections, made of a very large number of flip-flops connected to form a shift register, as in the figure below.

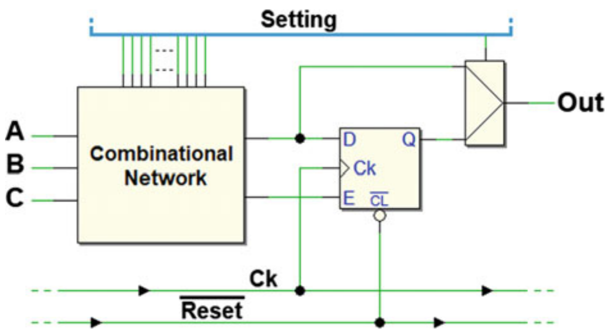


A particular synchronous serial port (JTAG interface, that we explain in the next section) is in charge to write the flip-flops during the FPGA programming phase. In normal operation, the flip-flops are not accessible.

The FPGA must be re-configured each time at power-up, since, as we know, a flip-flop cannot maintain information when power is off. Therefore, at power-up, a non-volatile memory in the board transfer programming information through dedicated pins. From the above, we understand that the FPGA may be re-programmed to perform a different function, using the JTAG interface.

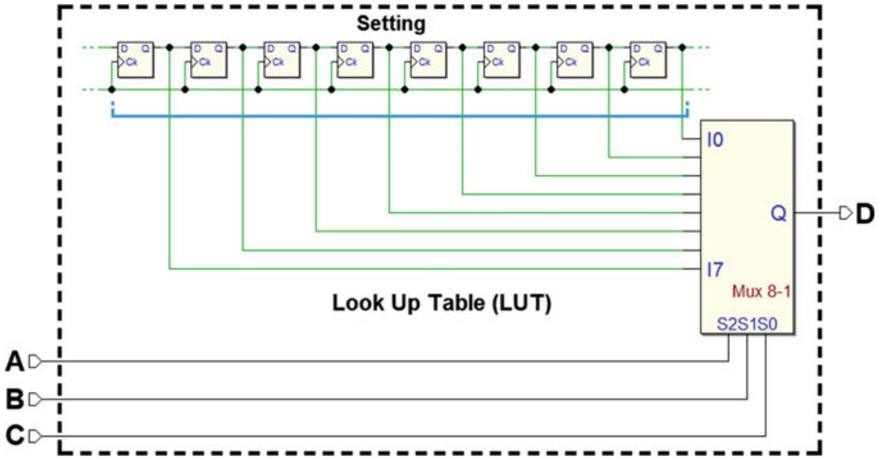
Logic Block

In the next figure, the schematic of a simplified FPGA *logic block* is represented. Basically, we have an E-type flip-flop, edge-triggered, driven by a combinational logic network.



The combinational network's operation is controlled by the configuration flip-flops (top of the figure), as well as the multiplexer on the right that allows the option of storing in the flip-flop the output of the combinational network.

It is worth to have a closer look at the combinational network, which is implemented as a Lookup Table (LUT). In Sect. 2.6.6, we have seen how to use a multiplexer as a configurable combinational network, by feeding into its inputs the values of the desired function, in the present case provided by the configuration flip-flops.



The multiplexer copies in its output D, according to the combination of A, B, and C, the values stored in the flip-flops, implementing the function simply by reading the LUT, which is defined at the time of FPGA configuration and does not change during its operations.

Note that a logic block could be more complex than the one just presented, containing also a full adder, XOR gates, other flip-flops, or multiple LUTs.

9.1.4 JTAG Programming

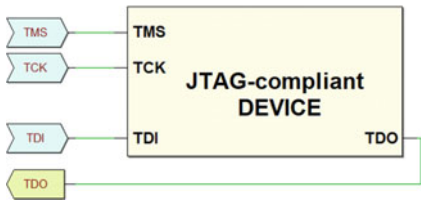
JTAG is the acronym of the consortium (Joint Test Action Group) that defined, at the end of the 1980s of last century a standard protocol for the functional test of integrated circuits, which later became the IEEE 1149.1 (*IEEE Standard Test Access Port and Boundary-Scan Architecture*). In the following, we will refer to it as JTAG (the term *Boundary-Scan* is sometimes used).

The version of the protocol released in 1994 added the possibility of programming memories, microcontrollers, and other devices. In addition, it allowed to perform the functional verification of the firmware and the possibility to activate automatic testing (Built-In Self-Test), defined by the component's manufacturer. A standard language to access components (*Boundary Scan Description Language*), by using JTAG, has been developed.

Nowadays, JTAG is the only procedure to access electronic systems, such as cell phones, tablets, wireless access points, and the like, for testing and troubleshooting.

In synthesis, the standard provides the possibility of blocking the normal operation of a system and disconnecting its clock, to switch to a modality in which the JTAG interface takes control of all the components' pins and the testing and programming circuits that may be inside the system itself.

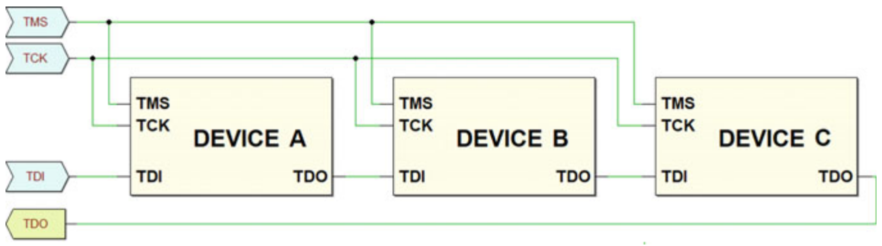
The physical JTAG interface is composed of a limited number of standard connections. The simplest set allows to communicate with the circuit using a few lines, as shown in the following figure.



Pin	Name	Function
TCK	*Test Clock*	Data Clock Pin
TMS	*Test Mode Select*	Mode Control and Operation Selection
TDI	*Test Data In*	Serial Data Input Pin (toward the device)
TDO	*Test Data Out*	Serial Data Output Pin (from the device)

All JTAG's signals are serial and synchronized by clock TCK (usually in the range 10–100 MHz). The activation of TMS signals to the system to enter the JTAG-compliant mode. Then, through the same line it is possible to perform the operation requested, using a state algorithm, the "JTAG State Machine" (not described here). The standard defines an optional Test Reset control (TRST) also, but its functionality can be obtained via TMS control, and often it is not used, as in the example above.

Moreover, the standard allows the series connection of the pin TDI and TDO of more than one device ("Daisy Chain" connection) in order to access all the JTAG-compliant devices in a board (shown in the next figure). An example of the power of this method is the possibility of performing a "Chain Integrity Test". Each JTAG-compliant device has its own ID code. All the ID codes can be read and checked against the ID of the design project, to verify if the JTAG chain is working as designed.

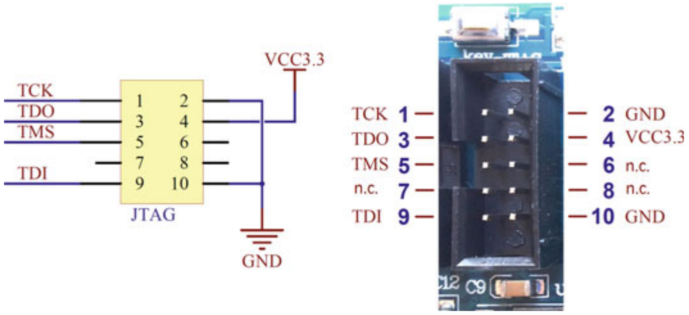


FPGA Programming

Many programmable devices, such as FPGA and PLD, are not designed to be JTAG-compliant for testing purposes only. They use JTAG for their programming.

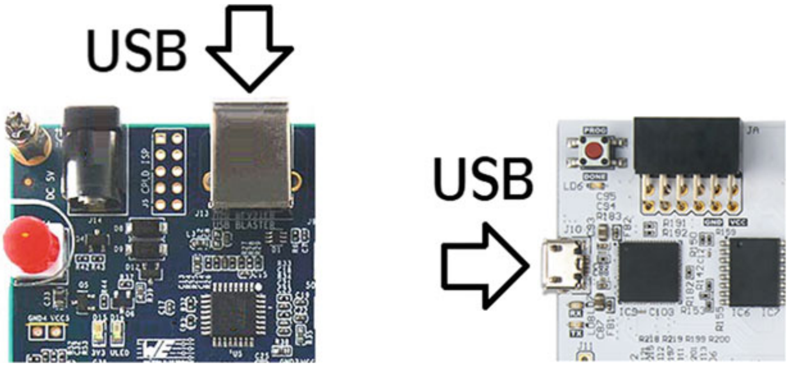
It must be stressed that FPGA is programmable after their insertion on the system's board. This fact provides several advantages, such as to simplify the programming phase, avoid the use of external programmers, update, and modify the networks implemented by the FPGA. This feature makes FPGA systems ideal for the implementation of prototypes, experimental circuits, including educational ones.

There are several standards for the JTAG physical interface. The simplest uses a 10-pin connector, as shown in the figure below that refers to the FPGA board with the *Intel/Altera FPGA* chip "Cyclone® II" already described.



In the next figure, an example of JTAG programmer with its 10-pin cable (right) and a standard USB cable for connection to the PC. The software is always provided by the FPGA manufacturer.

Often, especially in the most advanced boards, the programmer is built in with the board, available through a dedicated USB interface. This is the case of the first two boards described before.



9.1.5 FPGA Development Tools

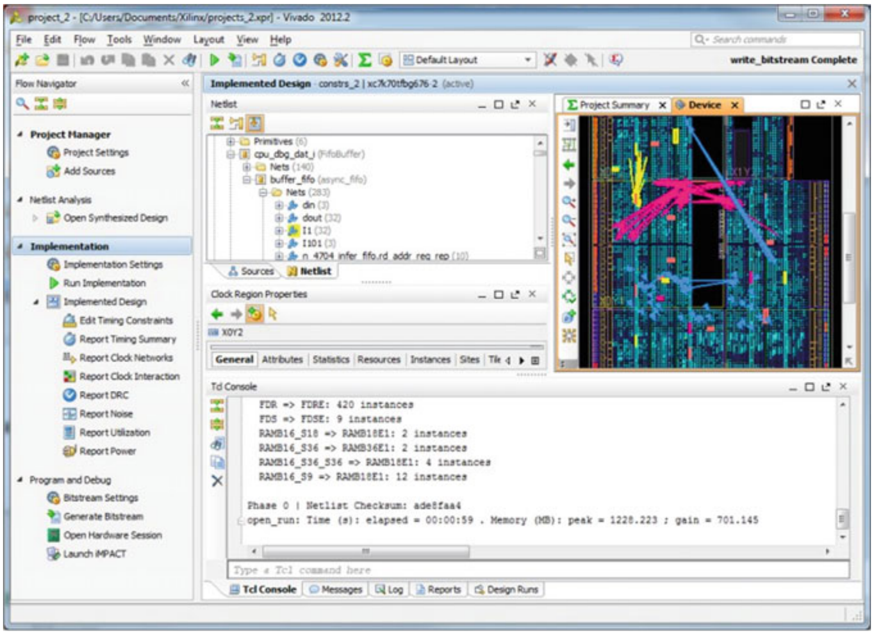
FPGA manufacturers offer (for a fee) proprietary development tools, targeted to professional designer and to the implementation of complex systems. The same manufacturers make available, usually free of charge, reduced versions of the same tools, usually targeted to the education field.

The obvious purpose is to publicize their products to future designers and, therefore, to influence their choices when they will be working. Plenty of documentation of such software tools is provided by the manufacturers.

In the following, we present a general view of what is available, free of charge, on the net. All the tools we mention offer similar features, such as schematic editor, source code editor, compiler, pin manager, optimization tools, and programmer and allow to design digital systems using logical schematics or Hardware Description Languages (HDL), such as Verilog, VHDL, or System C.

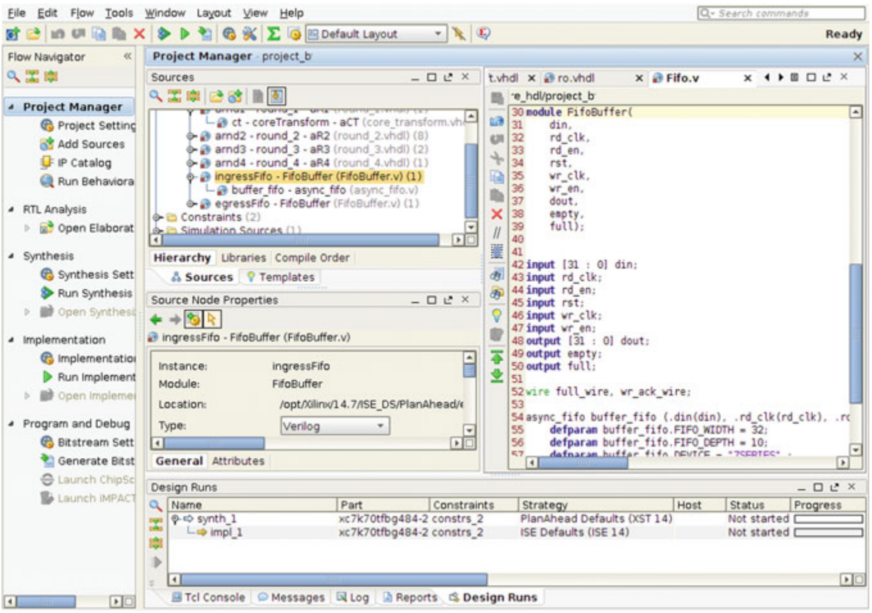
The large amount of features available, which make the professional's work more productive, produces in the beginner the impression of a difficult to manage complexity. In the following, we will see how *Deeds* allows to use FPGAs for a fast prototyping of our projects without going into the technicalities of the FPGA tools.

About boards based on *XilinX* devices, at the moment of writing two free tools are available: *Vivado® Design Suite HL WebPACK™* and *ISE® WebPACK™*. The screenshot below shows *Vivado®*, which is targeted to the most recent families of devices.



```
  FDR => FDRE: 420 instances
  FDS => FDSE: 9 instances
  RAMB16_S18 => RAMB18E1: 2 instances
  RAMB16_S36 => RAMB36E1: 2 instances
  RAMB16_S36_S36 => RAMB18E1: 4 instances
  RAMB16_S9 => RAMB18E1: 12 instances

Phase 0 | Netlist Checksum: ade8faa4
open_run: Time (s): elapsed = 00:00:59 . Memory (MB): peak = 1228.223 ; gain = 701.145
```

Below is *ISE®*. It supports project development on less recent FPGAs:



ct - coreTransform - aCT (core_transform.vhd)
arnd2 - round_2 - aR2 (round_2.vhdl) (8)
arnd3 - round_3 - aR3 (round_3.vhdl) (2)
arnd4 - round_4 - aR4 (round_4.vhdl) (1)
ingressFifo - FifoBuffer (FifoBuffer.v) (1)
buffer_fifo - async_fifo (async_fifo.v)
egressFifo - FifoBuffer (FifoBuffer.v) (1)
Constraints (2)
Simulation Sources (1)

ingressFifo - FifoBuffer (FifoBuffer.v)

Instance:	ingressFifo
Module:	FifoBuffer
Location:	/opt/Xilinx/14.7/ISE_DS/PlanAhead/
Type:	Verilog

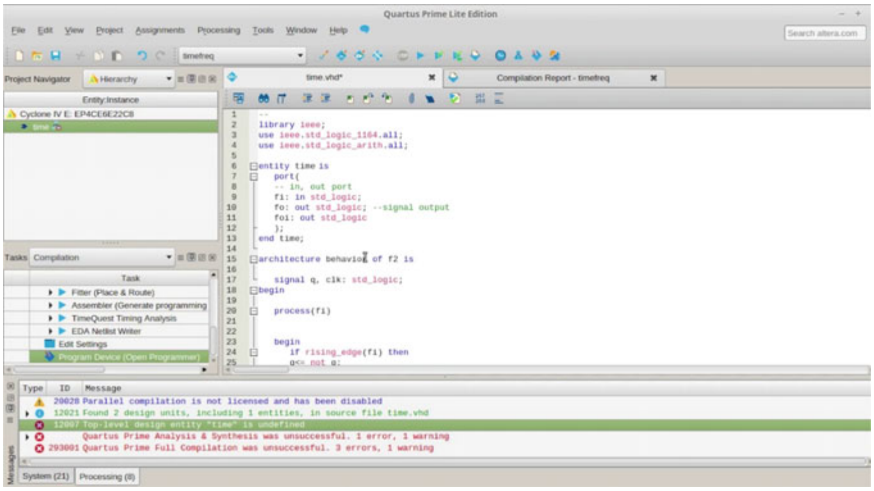
t.vhdl ro.vhdl Fifo.v

```
module FifoBuffer(
    din,
    rd_clk,
    rd_en,
    rst,
    wr_clk,
    wr_en,
    dout,
    empty,
    full);


input [31 : 0] din;
input rd_clk;
input rd_en;
input rst;
input wr_clk;
input wr_en;
output [31 : 0] dout;
output empty;
output full;

wire full_wire, wr_ack_wire;

async_fifo buffer_fifo (.din(din), .rd_clk(rd_clk), .r
    defparam buffer_fifo.FIFO_WIDTH = 32;
    defparam buffer_fifo.FIFO_DEPTH = 10;
```

Design Runs

Name	Part	Constraints	Strategy	Host	Status	Progress
synth_1	xc7k70tfbg484-2	constrs_2	PlanAhead Defaults (XST 14)		Not started	
impl_1	xc7k70tfbg484-2	constrs_2	ISE Defaults (ISE 14)		Not started	

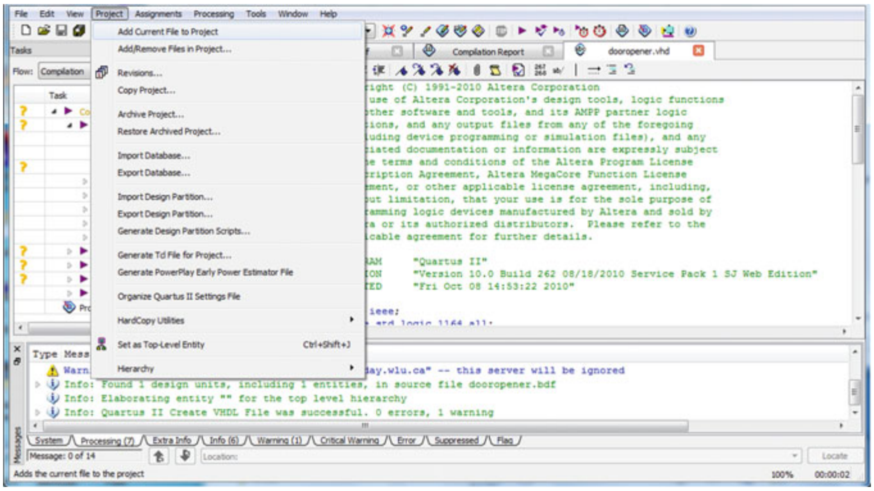
Intel/Altera FPGA provides two free tools: *Quartus® Prime Lite Edition*™ and *Quartus® II Web Edition*™. The next screenshot shows the mail window of the first one. It supports the most recent *Intel/Altera FPGA* families:


Quartus Prime Lite Edition
File Edit View Project Assignments Processing Tools Window Help
Search altera.com
timefreq
Project Navigator
Hierarchy
Entity:Instance
Cyclone IV E: EP4CE6E22C8
time
Tasks Compilation
Task
Fitter (Place & Route)
Assembler (Generate programming
TimeQuest Timing Analysis
EDA Netlist Writer
Edit Settings
Program Device (Open Programmer)
time.vhd*
Compilation Report - timefreq
Messages
System (21) Processing (8)

```
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity time is
   port(
   -- in, out port
   fi: in std_logic;
   fo: out std_logic; --signal output
   foi: out std_logic
   );
end time;

architecture behavior of f2 is

   signal q, clk: std_logic;
begin

   process(fi)


   begin
      if rising_edge(fi) then
      q<= not q;
```

```
Type  ID     Message
      20028  Parallel compilation is not licensed and has been disabled
      12021  Found 2 design units, including 1 entities, in source file time.vhd
      12007  Top-level design entity "time" is undefined
             Quartus Prime Analysis & Synthesis was unsuccessful. 1 error, 1 warning
      293001 Quartus Prime Full Compilation was unsuccessful. 3 errors, 1 warning
```

The older versions of the same software are called *Quartus® II*. Below is its main window with project management commands:

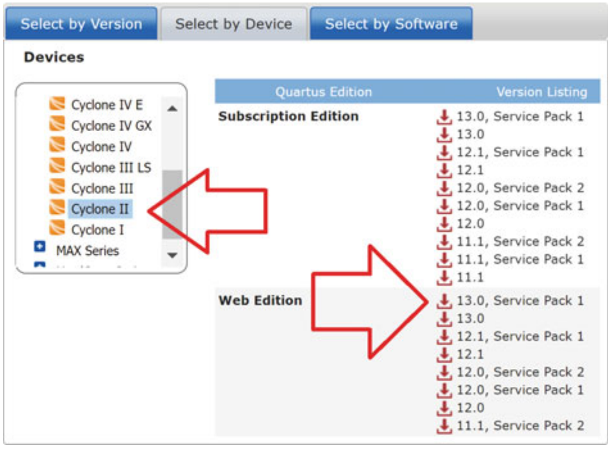


```
right (C) 1991-2010 Altera Corporation
 use of Altera Corporation's design tools, logic functions
other software and tools, and its AMPP partner logic
tions, and any output files from any of the foregoing
luding device programming or simulation files), and any
ciated documentation or information are expressly subject
e terms and conditions of the Altera Program License
cription Agreement, Altera MegaCore Function License
ement, or other applicable license agreement, including,
out limitation, that your use is for the sole purpose of
ramming logic devices manufactured by Altera and sold by
ra or its authorized distributors.  Please refer to the
icable agreement for further details.

RAM      "Quartus II"
ION      "Version 10.0 Build 262 08/18/2010 Service Pack 1 SJ Web Edition"
TED      "Fri Oct 08 14:53:22 2010"

 ieee;
```

```
Type Mess
  Warn.                                      day.wlu.ca" -- this server will be ignored
  Info: Found 1 design units, including 1 entities, in source file dooropener.bdf
  Info: Elaborating entity "" for the top level hierarchy
  Info: Quartus II Create VHDL File was successful. 0 errors, 1 warning
```

In the last few years, due to the increase in the number of families and the complexity of the chips, FPGA producers have put more efforts in developing and maintaining the tools that support the new families than in assuring their compatibility with older chips. In general, the less recent boards require the use of older version of development software. It is therefore necessary to pay attention in the choice of chips and tools, by studying the documentation available in the manufacturers' Web site.

Utilities called *Software Selector* associate the FPGA family to the corresponding software version. In the following figure, from the *Intel/Altera FPGA* Web site, we see that the *Cyclone*® *II* family is supported by a version *13.0 - ServicePack 1* (and older) of *Quartus*® *II Web Edition*™:

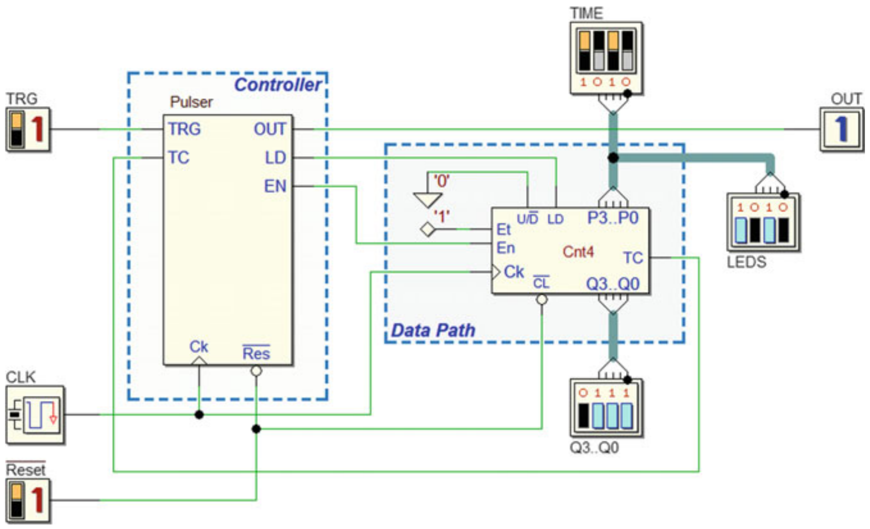


Quartus Edition	Version Listing
Subscription Edition	13.0, Service Pack 1
	13.0
	12.1, Service Pack 1
	12.1
	12.0, Service Pack 2
	12.0, Service Pack 1
	12.0
	11.1, Service Pack 2
	11.1, Service Pack 1
	11.1
Web Edition	13.0, Service Pack 1
	13.0
	12.1, Service Pack 1
	12.1
	12.0, Service Pack 2
	12.0, Service Pack 1
	12.0
	11.1, Service Pack 2

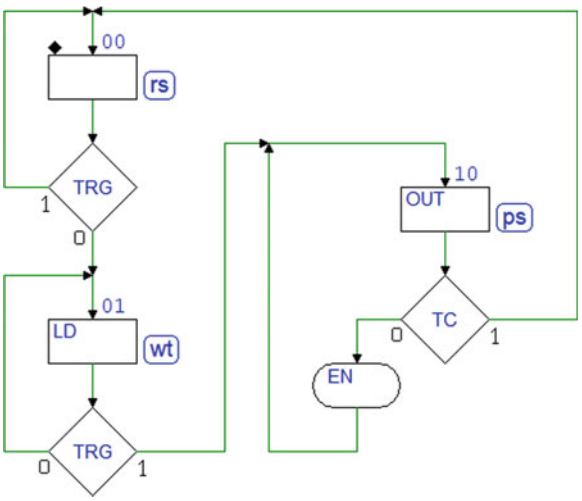
9.1.6 Deeds Support for FPGA

In the project development process, after design and simulation, a logical step is to test the network in a physical system. *Deeds* allows a fast implementation of prototypes on several FPGA boards commercially available.

To examine the path that goes from project to prototype, we use as an example a *Pulse Generator* similar to the ones seen on p. 372 in Chap. 8. We implement the project on the *Terasic/Altera* DE0-CV board presented before.



In this version, counter's outputs are visualized on a LED array, and pulse duration is set using 4 switches, with the aid of another LED array that shows the number while it is set. Controller's functionality has been extended by the addition of the line *EN* to enable counting. The FSM, in the reset state (rs), waits until the *TRG* input goes to zero.

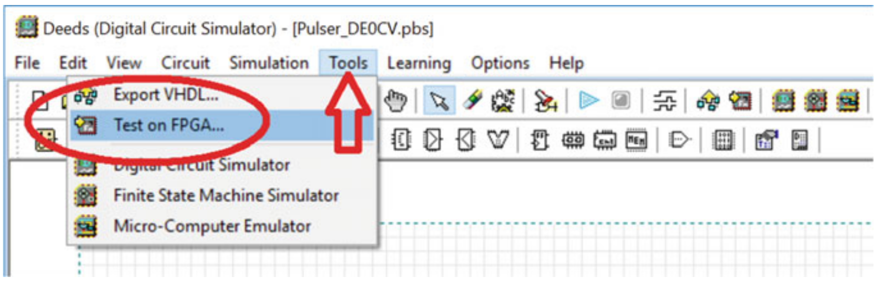


On next state (wt), the machine loads the counter and waits for a rising edge of *TRG*, after which goes to state (ps) that generates the pulse on *OUT*. In (ps) the counter in enabled until *TC* is activated and the FSM returns to the initial state.

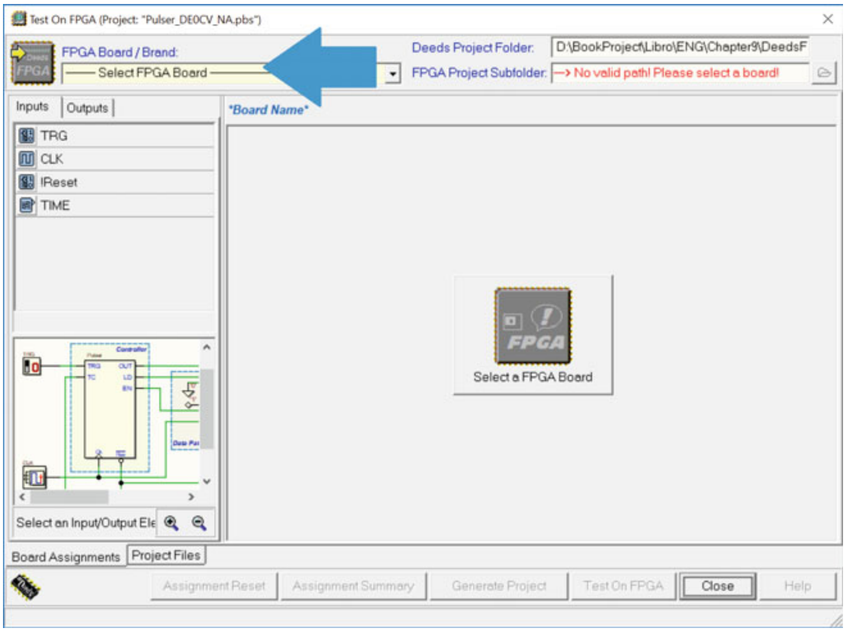
"Test On FPGA" Window

In the following, we go through all the steps of the process (in several projects available on the *Deeds*'s Web site most of the settings are already included in the files).

Let's assume we are done with the behavioral verification of the project, obtained by simulation, and are ready for the prototype implementation. We open the *"Test On FPGA"* window (see the *Tools* menu item).


Deeds (Digital Circuit Simulator) - [Pulser_DE0CV.pbs]
File Edit View Circuit Simulation Tools Learning Options Help
Export VHDL...
Test on FPGA...
Digital Circuit Simulator
Finite State Machine Simulator
Micro-Computer Emulator

In the *"Test On FPGA"* window firstly we select a FPGA board from the list box (blue arrow in the next figure):


Test On FPGA (Project: "Pulser_DE0CV_NA.pbs")

FPGA Board / Brand: —— Select FPGA Board ——

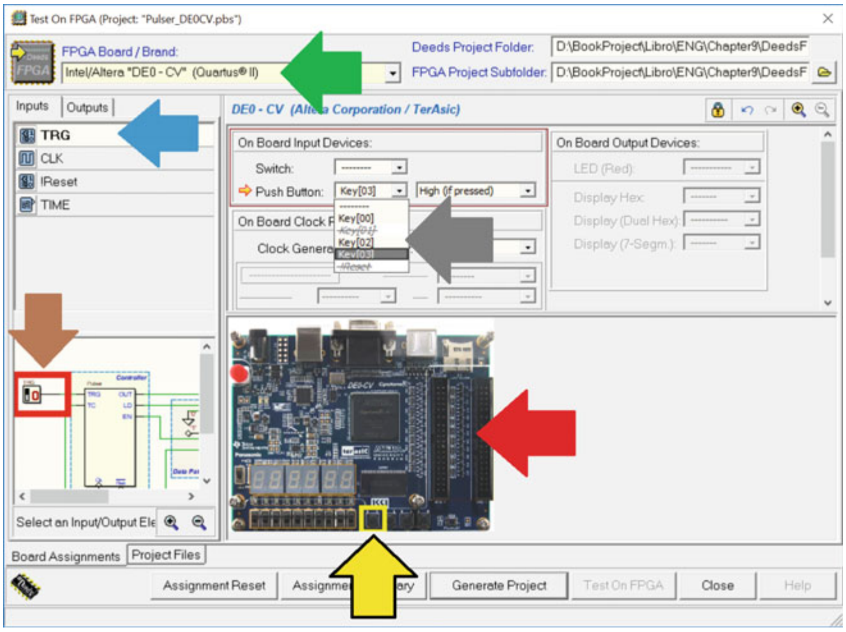
Deeds Project Folder: D:\BookProject\Libro\ENG\Chapter9\DeedsF

FPGA Project Subfolder: —> No valid path! Please select a board!

"Board Name"

Select a FPGA Board

After the selection of the board, in our case the DE0-CV (green arrow in the figure below), we see the picture of the physical board in use (red arrow).


Test On FPGA (Project: "Pulser_DE0CV.pbs")

FPGA Board / Brand: Intel/Altera "DE0 - CV" (Quartus® II)

Deeds Project Folder: D:\BookProject\Libro\ENG\Chapter9\DeedsF

FPGA Project Subfolder: D:\BookProject\Libro\ENG\Chapter9\DeedsF

DE0 - CV (Altera Corporation / TerAsic)

On Board Input Devices:

Switch: ---------

Push Button: Key[03] High (if pressed)

Resource Assignment

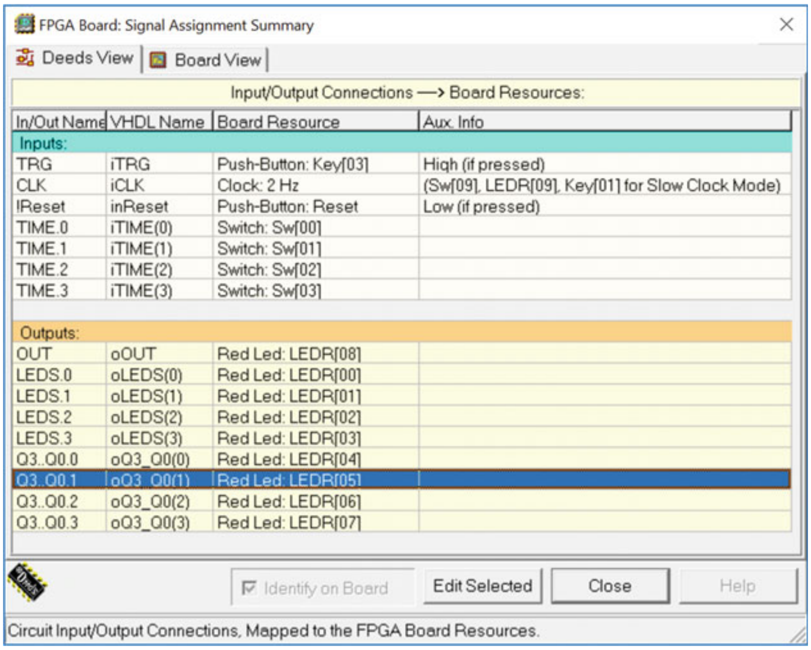
We need to associate the network's input and output components with the devices available on the FPGA board. We select, one by one, the input/output components of our network, either by clicking on the schematic (brown arrow) or in the list (blue arrow).

In the example of the previous figure, we point to the input *TRG* and the system shows only the board's resources that are compatible with our selection. In this case, we choose to associate to *TRG* the push-button "Key[03]".

While scrolling the list (gray arrow), each physical device is identified on the board inside a yellow frame (yellow arrow).

Assignment Summary

After the association of all inputs/outputs of our network to the corresponding physical resources of the board, we can check the "Assignment Summary" by clicking the button with the same name in the bottom of the window.



Input/Output Connections —> Board Resources:

In/Out Name	VHDL Name	Board Resource	Aux. Info
Inputs:			
TRG	iTRG	Push-Button: Key[03]	High (if pressed)
CLK	iCLK	Clock: 2 Hz	(Sw[09], LEDR[09], Key[01] for Slow Clock Mode)
!Reset	inReset	Push-Button: Reset	Low (if pressed)
TIME.0	iTIME(0)	Switch: Sw[00]	
TIME.1	iTIME(1)	Switch: Sw[01]	
TIME.2	iTIME(2)	Switch: Sw[02]	
TIME.3	iTIME(3)	Switch: Sw[03]	
Outputs:			
OUT	oOUT	Red Led: LEDR[08]	
LEDS.0	oLEDS(0)	Red Led: LEDR[00]	
LEDS.1	oLEDS(1)	Red Led: LEDR[01]	
LEDS.2	oLEDS(2)	Red Led: LEDR[02]	
LEDS.3	oLEDS(3)	Red Led: LEDR[03]	
Q3..Q0.0	oQ3_Q0(0)	Red Led: LEDR[04]	
Q3..Q0.1	oQ3_Q0(1)	Red Led: LEDR[05]	
Q3..Q0.2	oQ3_Q0(2)	Red Led: LEDR[06]	
Q3..Q0.3	oQ3_Q0(3)	Red Led: LEDR[07]	

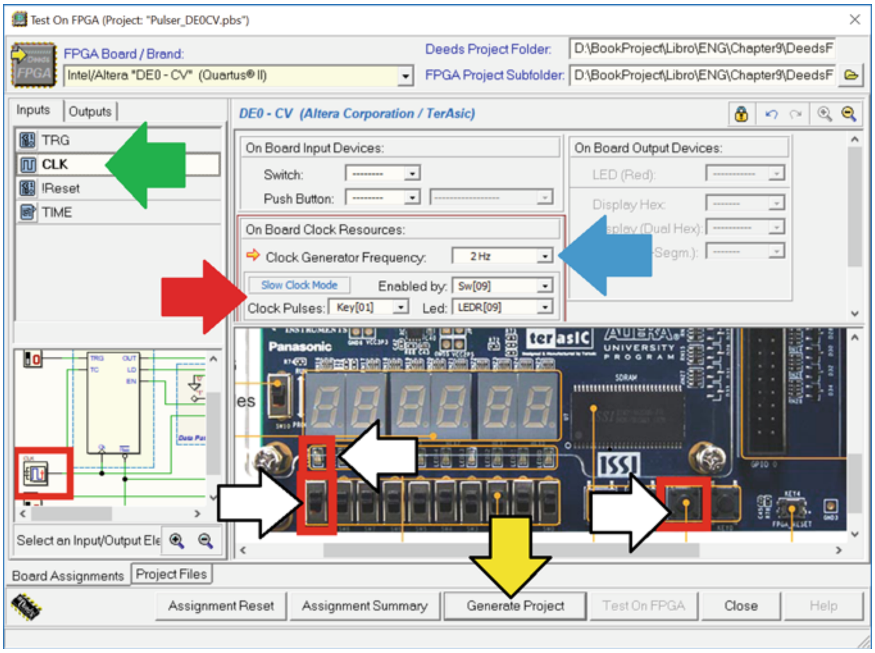
Circuit Input/Output Connections, Mapped to the FPGA Board Resources.

Clock Frequency Setting and Slow Clock Test Mode

The clock frequency on the board (Fc) can be defined at wish, independently of the one chosen in *Deeds*'s schematic. A frequency divider (not shown here) allows to scale downward the native clock frequency (50 MHz in the case of the DE0-CV board).

In our *Pulse Generator* project, the output pulse can last up to 16 clock cycles. Therefore, for visual testing, $Fc = 2$ Hz allows to generate pulses up to 8 seconds.

Deeds inserts a frequency divider by 25,000,000 between the board clock generator and the clock input of our network.


Test On FPGA (Project: "Pulser_DE0CV.pbs")

FPGA Board / Brand: Intel/Altera "DE0 - CV" (Quartus® II)

Deeds Project Folder: D:\BookProject\Libro\ENG\Chapter9\DeedsF

FPGA Project Subfolder: D:\BookProject\Libro\ENG\Chapter9\DeedsF

DE0 - CV (Altera Corporation / TerAsic)

Inputs: TRG, CLK, !Reset, TIME

On Board Clock Resources:

Clock Generator Frequency: 2 Hz

Slow Clock Mode — Enabled by: Sw[09]

Clock Pulses: Key[01] Led: LEDR[09]

It is possible to slow further down the clock to observe *step by step* the behavior of the network, using the *"Slow Clock Mode"* (see the area pointed by the red arrow, in the figure above).

We set the switch "Sw[09]" to activate it and the LED "LEDR[09]" to visualize the clock pulses sent to the circuit under test. Push-button "Key[01]" will be the manual command to generate clock pulses, one by one.

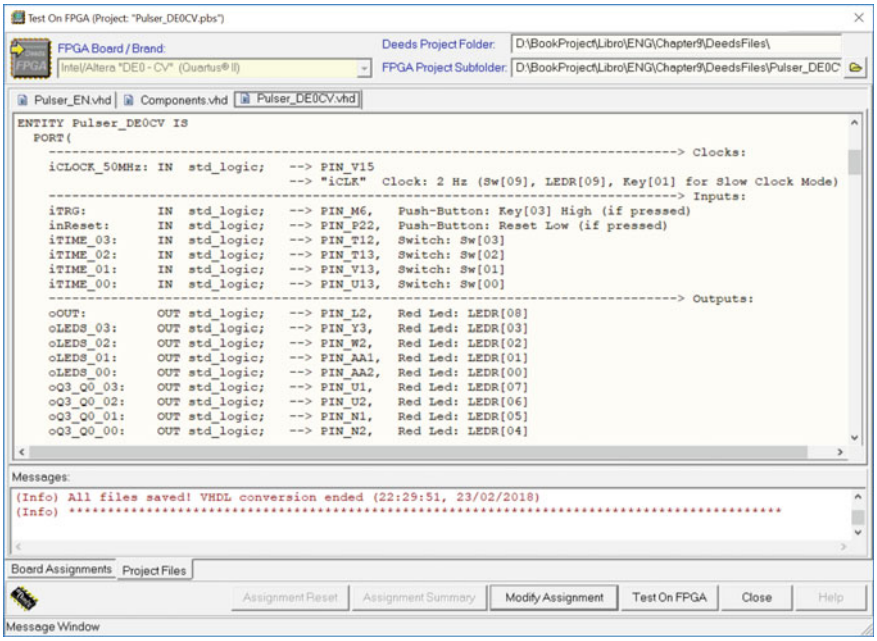
If switch "Sw[09]" is at '0' during the test, the clock works with the regular 2 Hz frequency. If at '1' the clock stops and, at every push of "Key[01]" a complete clock cycle is generated. If the same button is kept pressed, the pulse is repeated at the rate of about two cycles per second.

Project Generation

The time has come to translate our network, with all its associations, into a *Project* that can be opened in the tool corresponding to the FPGA used.

In our case, the board is the DE0-CV, so the tool is *Quartus*® *II Web Edition*™. We use the button "Generate Project" (see the yellow arrow in the figure above) to start the process.

The window now looks as it is shown in the following figure.

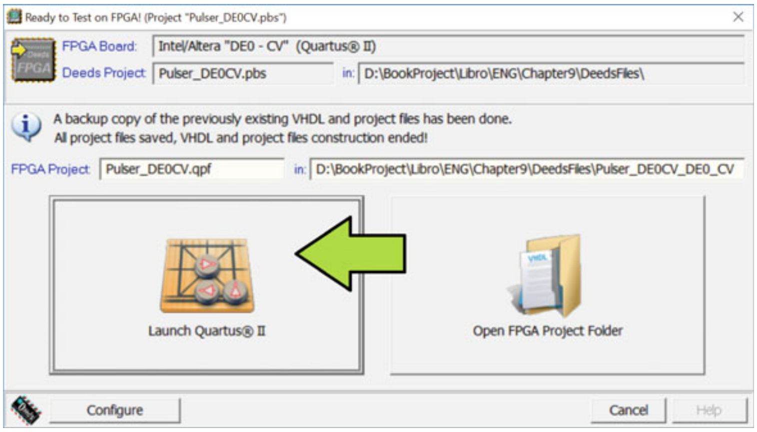


```
ENTITY Pulser_DE0CV IS
  PORT(
    ---------------------------------------------------------------------> Clocks:
    iCLOCK_50MHz: IN  std_logic;   --> PIN_V15
                                   --> "iCLK"  Clock: 2 Hz (Sw[09], LEDR[09], Key[01] for Slow Clock Mode)
    ---------------------------------------------------------------------> Inputs:
    iTRG:         IN  std_logic;   --> PIN_M6,   Push-Button: Key[03] High (if pressed)
    inReset:      IN  std_logic;   --> PIN_P22,  Push-Button: Reset Low (if pressed)
    iTIME_03:     IN  std_logic;   --> PIN_T12,  Switch: Sw[03]
    iTIME_02:     IN  std_logic;   --> PIN_T13,  Switch: Sw[02]
    iTIME_01:     IN  std_logic;   --> PIN_V13,  Switch: Sw[01]
    iTIME_00:     IN  std_logic;   --> PIN_U13,  Switch: Sw[00]
    ---------------------------------------------------------------------> Outputs:
    oOUT:         OUT std_logic;   --> PIN_L2,   Red Led: LEDR[08]
    oLEDS_03:     OUT std_logic;   --> PIN_Y3,   Red Led: LEDR[03]
    oLEDS_02:     OUT std_logic;   --> PIN_W2,   Red Led: LEDR[02]
    oLEDS_01:     OUT std_logic;   --> PIN_AA1,  Red Led: LEDR[01]
    oLEDS_00:     OUT std_logic;   --> PIN_AA2,  Red Led: LEDR[00]
    oQ3_Q0_03:    OUT std_logic;   --> PIN_U1,   Red Led: LEDR[07]
    oQ3_Q0_02:    OUT std_logic;   --> PIN_U2,   Red Led: LEDR[06]
    oQ3_Q0_01:    OUT std_logic;   --> PIN_N1,   Red Led: LEDR[05]
    oQ3_Q0_00:    OUT std_logic;   --> PIN_N2,   Red Led: LEDR[04]
```

```
(Info) All files saved! VHDL conversion ended (22:29:51, 23/02/2018)
(Info) ***********************************************************************************
```

Deeds generates several files that represent the translation of our network in VHDL code. VHDL will be introduced later in this chapter: for the moment is enough to know that the files describe in a textual format connections and behavior of all the components in our network.

When the *Project* has been generated a dialog windows appears:



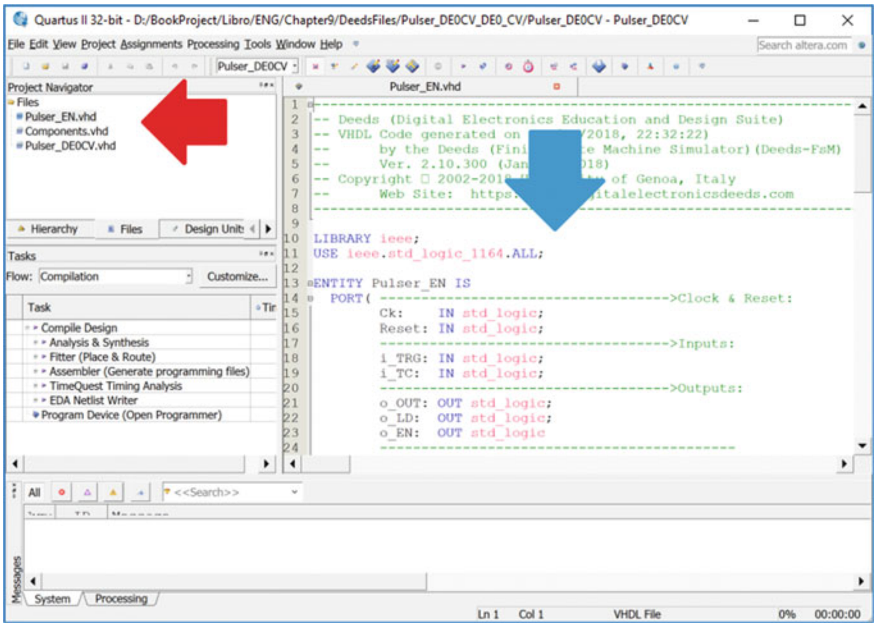
FPGA Board: Intel/Altera "DE0 - CV" (Quartus® II)
Deeds Project: Pulser_DE0CV.pbs in: D:\BookProject\Libro\ENG\Chapter9\DeedsFiles\

A backup copy of the previously existing VHDL and project files has been done.
All project files saved, VHDL and project files construction ended!

FPGA Project: Pulser_DE0CV.qpf in: D:\BookProject\Libro\ENG\Chapter9\DeedsFiles\Pulser_DE0CV_DE0_CV

The window confirms that the generation of the Project has been successful and allows either to launch *Quartus*® *II* (green arrow) or simply to access the project's file folder.

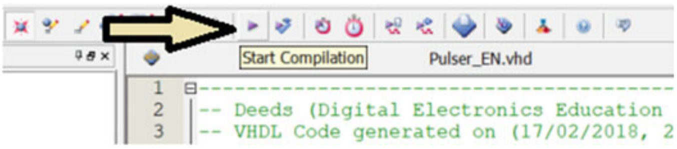
Quartus® *II* will open our project, as seen in the figure in the next page.


Quartus II 32-bit - D:/BookProject/Libro/ENG/Chapter9/DeedsFiles/Pulser_DE0CV_DE0_CV/Pulser_DE0CV - Pulser_DE0CV
File Edit View Project Assignments Processing Tools Window Help
Search altera.com
Pulser_DE0CV
Project Navigator
Files
Pulser_EN.vhd
Components.vhd
Pulser_DE0CV.vhd
Hierarchy
Files
Design Units
Tasks
Flow: Compilation
Customize...
Task
Compile Design
Analysis & Synthesis
Fitter (Place & Route)
Assembler (Generate programming files)
TimeQuest Timing Analysis
EDA Netlist Writer
Program Device (Open Programmer)
Pulser_EN.vhd
All
<<Search>>
Messages
System
Processing
Ln 1 Col 1 VHDL File 0% 00:00:00

```
-----------------------------------------------------------------
-- Deeds (Digital Electronics Education and Design Suite)
-- VHDL Code generated on      /2018, 22:32:22)
--      by the Deeds (Fini     e Machine Simulator)(Deeds-FsM)
--      Ver. 2.10.300 (Jan     018)
-- Copyright □ 2002-2018         of Genoa, Italy
--      Web Site:  https:        italelectronicsdeeds.com
-----------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY Pulser_EN IS
  PORT( ---------------------------------->Clock & Reset:
        Ck:     IN std_logic;
        Reset:  IN std_logic;
        ---------------------------------->Inputs:
        i_TRG:  IN std_logic;
        i_TC:   IN std_logic;
        ---------------------------------->Outputs:
        o_OUT:  OUT std_logic;
        o_LD:   OUT std_logic;
        o_EN:   OUT std_logic
        ----------------------------------
```

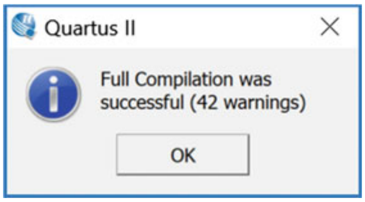
The red arrow points to the list of the generated VHDL files, while the blue to one of the files opened in the editor. It would be possible, at this point, to modify the VHDL code generated automatically by *Deeds*.

In fact, FPGA boards offer a wealth of possibilities from which students could take advantage, using the code generated by *Deeds* as a starting point. For our purposes, instead, it is enough to click "Start Compilation" to proceed with the FPGA configuration process.



```
-----------------------------------------
-- Deeds (Digital Electronics Education
-- VHDL Code generated on (17/02/2018, 2
```

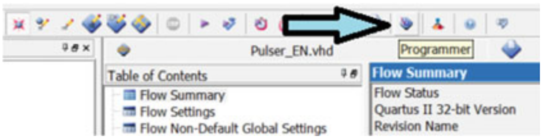
When the compilation is finished, a message reports a certain number of *warnings*. Warnings are not errors and, in our case, can be ignored.



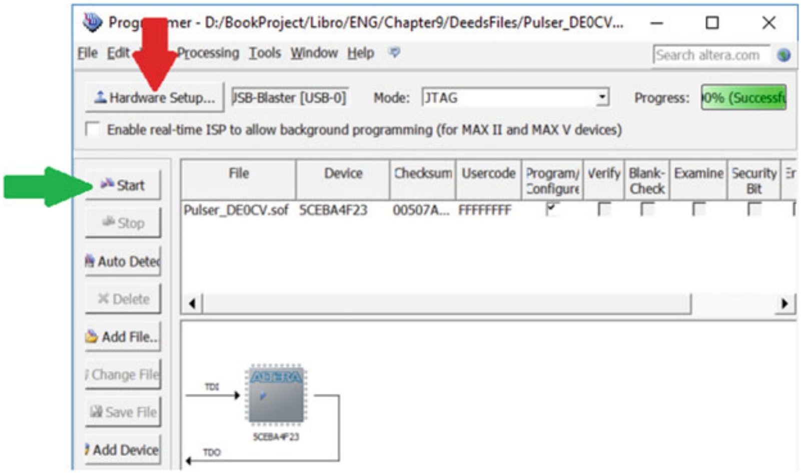
Full Compilation was successful (42 warnings)

Chip Programming

In the last phase, we must download the result of compilation on the board, using the "Programmer" module:



The green arrow points to the "Start" button that commences downloading. The red arrow points to the command that establishes communication between the PC hosting the tools and the FPGA board.



USB-Blaster [USB-0] Mode: JTAG Progress: 100% (Successful)

Enable real-time ISP to allow background programming (for MAX II and MAX V devices)

File	Device	Checksum	Usercode	Program/ Configure	Verify	Blank-Check	Examine	Security Bit
Pulser_DE0CV.sof	5CEBA4F23	00507A...	FFFFFFFF	✓				

The VHDL code generated in this example will be explained in the next section, that also will introduce the basic elements of the language.

9.2 Introduction to VHDL

As stated in the preface of this book, we presented so far digital design basics using a traditional schematic entry approach. This choice favors an intuitive and visual understanding of concepts and circuits and allows a "conscious" transition to *Hardware Description Languages* (HDL).

In fact, HDLs are the current industry standard to describe and design digital systems. A very large and complex digital system can be efficiently designed with a top-down methodology using HDLs. Schematics are still used at the board level, where it is necessary to describe the wiring to the other parts of the system, or to the external connections.

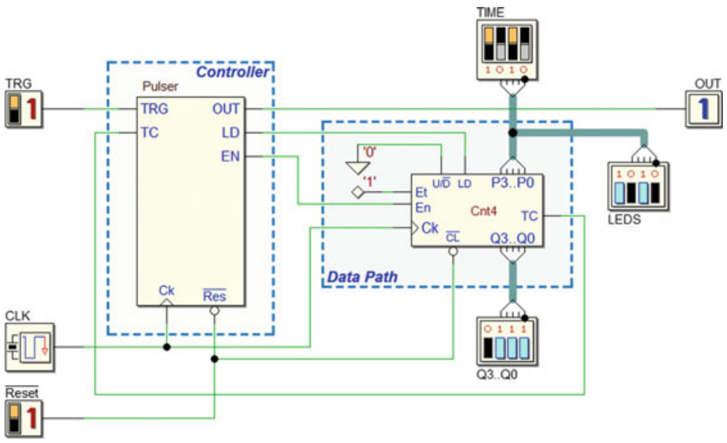
A HDL is a programming language that allows us to describe digital circuits, either in a behavioral or structural way. HDLs are often used for simulation.

Very high-speed integrated-circuit Hardware Description Language (VHDL) and Verilog are the most widespread and are supported by the majority of CAD tools available. Currently a great and growing attention is reserved to System C, mainly a set of C++ class libraries that provide the necessary modeling code to describe systems. Other HDLs to be mentioned are JHDL (*Java HDL*) or Active-HDL (from *Cypress Semiconductor*).

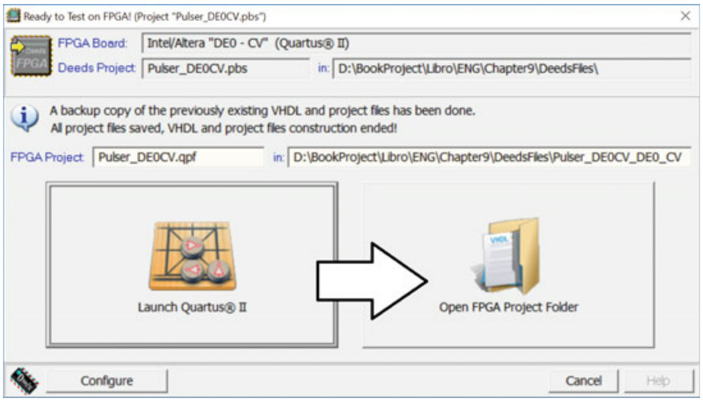
In this section, we introduce VHDL, starting from the example seen just before. Our presentation of the language is far from exhaustive since VHDL is used for many purposes and here we see only one of them, the *"VHDL for synthesis"*.

9.2.1 VHDL Code from Deeds

We refer to the schematic seen in the previous section (see below).



Apart from the connections (from inputs, to outputs and between blocks), the network contains two components, a controller (designed as FSM) and a 4-bit counter. Now, let's look at the files generated by *Deeds*, by going back in the chapter to the window that follows.

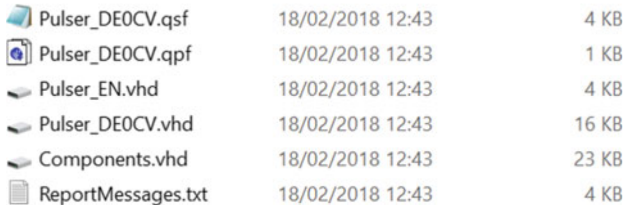

Ready to Test on FPGA! (Project "Pulser_DE0CV.pbs")

FPGA Board: Intel/Altera "DE0 - CV" (Quartus® II)
Deeds Project: Pulser_DE0CV.pbs in: D:\BookProject\Libro\ENG\Chapter9\DeedsFiles\

A backup copy of the previously existing VHDL and project files has been done.
All project files saved, VHDL and project files construction ended!

FPGA Project: Pulser_DE0CV.qpf in: D:\BookProject\Libro\ENG\Chapter9\DeedsFiles\Pulser_DE0CV_DE0_CV

This time we choose to press the big button on the right (“Open FPGA Project Folder”) to examine in more detail the files generated within the project.



Pulser_DE0CV.qsf	18/02/2018 12:43	4 KB
Pulser_DE0CV.qpf	18/02/2018 12:43	1 KB
Pulser_EN.vhd	18/02/2018 12:43	4 KB
Pulser_DE0CV.vhd	18/02/2018 12:43	16 KB
Components.vhd	18/02/2018 12:43	23 KB
ReportMessages.txt	18/02/2018 12:43	4 KB

The first two, in text format, define all the project’s parameters, such as the name of the FPGA chip and the correspondence between the input/output connections of our network and the FPGA physical connections (pins). The last one is a log file that serves for debugging purposes. We concentrate our interest on the files with extension “.vhd” (the complete code is available on in Appendix C).

The “Components.vhd” contains the behavioral description of all the components included in the *Deeds* project (except FSMs and microcomputers). In our case, excluding the FSM (the controller), and the I/O objects, only one component should be in this VHDL file, the 4-bit counter.

However, the file contains also another component (named *“ClockScaler”*, not described here in detail), which is generated by *Deeds* to implement the *“Slow Clock Mode”*. As introduced before, this is the component that provides to the network a clock whose frequency is scaled down to the value defined in the settings.

9.2.2 *Counter*

A quick look into the VHDL file allows us to see the following code, which describes the terminals of the counter.

```
ENTITY Counter4b IS
  PORT( Ck  : IN std_logic;
        nCL: IN std_logic;
        LD  : IN std_logic;
        ENP: IN std_logic;
        ENT: IN std_logic;
        UD  : IN std_logic;
        P3  : IN std_logic;
        P2  : IN std_logic;
        P1  : IN std_logic;
        P0  : IN std_logic;
        Q3  : OUT std_logic;
        Q2  : OUT std_logic;
        Q1  : OUT std_logic;
        Q0  : OUT std_logic;
        Tc  : OUT std_logic );
END Counter4b;
```

VHDL uses the *design entity* concept, and, in our example, "Counter4b" is an *entity*. This piece of code represents the *entity declaration unit* of the component and describes its external "PORT" interface, including all the terminals by name and type. For instance, Tc is defined as output ("OUT") and of type std_logic. The same type applies to all inputs and outputs of this example.

The type std_logic is used to define a digital signal that can assume the 0 and 1 values. While this may look obvious, really it is not since this type can assume nine different values, useful for simulation (for instance, X = Unknown).

The *entity declaration unit* must be followed by the *architecture body unit*, which represents the internal description of the *design entity*. It can describe its behavior, its structure, or a mix of both. For instance, in the following code example, we define an entity with two inputs (A, B) and one output (U).

```
ENTITY MyNand IS
  PORT( A,
        B: IN std_logic;
        U: OUT std_logic );
end MyNand;

ARCHITECTURE behavioral OF MyNand IS
BEGIN
  U <= not (A and B);
END behavioral;
```

The architecture part is described between the couple of keywords BEGIN .. END. The word "behavioral" is completely arbitrary (it should be the *private name* of the architecture, but it is not useful in this context). Instead, it is often used by designers to identify the style of the description (*behavioral* or *structural*).

With *behavioral description*, we mean that the entity is described by a function or an algorithm, without dealing directly with the components and connections (as in the example, where a Boolean function defines the output U). A *structural description*, instead, is the textual translation of connection among the blocks that form the logic network, in a way equivalent to traditional schematics. In any case, the two styles can be mixed, when convenient.

Let's return to our 4-bit counter, and consider the following code, extracted from its architecture description. We see an example of process:

```
ARCHITECTURE behavioral OF Counter4b IS
BEGIN
  Count4b: PROCESS( Ck, nCL, ENP, ENT, UD )
  variable aCnt: unsigned( 3 downto 0 );
  BEGIN
              -- omissis...

  END PROCESS;
END behavioral;
```

A process could resemble, at a first glance, a function as the ones that we can write in C, JAVA, or other *procedural languages*. The resemblance is only apparent: a process describes the entity's behavior as a *parallel process*.

There is not a *main program* that calls the processes: they are *"launched"* when at least one of the signals defined between parentheses (*sensitivity list*) changes. In our case, the process is executed if the value of at least one of inputs *Ck*, *nCL*, *ENP*, *ENT*, or *UD* changes, as it happens in a physical network.

Let's examine the code of the process that describes the 4-bit counter.

```
Count4b: PROCESS( Ck, nCL, ENP, ENT, UD )
variable aCnt: unsigned( 3 downto 0 );
BEGIN
  if    (nCL = '0') then  aCnt := (others =>'0');
  elsif (nCL = '1') then
    if (Ck'event) AND (Ck='1') then
      if    (LD = '1') then  aCnt := (P3 & P2 & P1 & P0);
      elsif (LD = '0') then
        if  (ENP = '1') and (ENT = '1')then
          if    (UD = '1') then
            if (aCnt < "1111") then  aCnt := aCnt + 1;
            else  aCnt := (others =>'0');
            end if;
          elsif (UD = '0') then
            if (aCnt > "0000") then aCnt := aCnt - 1;
            else  aCnt := (others =>'1');
            end if;
          else  aCnt := (others =>'X'); -- (UD: Unknown)
          END IF;
        elsif not((ENP ='0') or (ENT ='0') ) then
          aCnt := (others =>'X'); -- (ENP: Unknown)
        END IF;
      else  aCnt := (others =>'X'); -- (LD: Unknown)
      END IF;
    END IF;
  else  aCnt := (others =>'X'); -- (nCL: Unknown)
  END IF;
  --
  Tc <= ENT and ((aCnt(3) and aCnt(2) and
                  aCnt(1) and aCnt(0) and UD) or
             (not(aCnt(3) or  aCnt(2) or
                  aCnt(1) or  aCnt(0) or UD)));
  --
  Q3 <= aCnt(3);
  Q2 <= aCnt(2);
  Q1 <= aCnt(1);
  Q0 <= aCnt(0);
  --
END PROCESS;
```

A few preliminary observations:

(a) inputs Ck, nCL, ENP, ENT and UD (the ones that appear in the *sensitivity list*, line #1) are tested by *if-then-else* constructs (the keyword *elsif* is short for *else if*).
(b) line #2 defines a local variable *aCnt*, of type *unsigned 4-bit integer*, representing the counter state.
(c) the *aCnt := (others => '0')* construct (as at line #4) sets to zero all the variable's bits.
(d) in the same way, *aCnt := (others => 'X')* (for instance at line #18) declares that the state of the counter is *unknown* (notice that this fact is relevant only for simulation, not synthesis).
(e) line #6 calls the library function *Ck'event* that detects a level transition of Ck. The argument of the *if* will be true in the case of a *rising edge*.
(f) two consecutive dashes '- -' signal the beginning of a comment that ends with the line.

Let's interpret now the code in natural language. If the input clear nCL is '0', the counter's state is cleared asynchronously (line #4). If nCL is at '1', the system waits for a rising edge of clock (line #5,6) and, therefore, the part of code following this control will be evaluated only when a rising edge arrives. Because of the rising edge check, the compiler must instance memory elements to store $aCnt$ between two rising edges.

On the rising edge, the process controls the load LD input (line #7). If LD = '1', the value of the inputs $P3$, $P2$, $P1$, and $P0$ is loaded in $aCnt$. Notice that the four inputs are combined together by the operator '&' to form a *4-bit variable*. If, instead, LD = '0', the process controls inputs ENP and ENT (line #9), which must be at '1' at the same time to enable counting.

If this condition is verified, next line controls input *UD* that sets the counting direction (if *UD* = '1', the statement *aCnt := aCnt + 1* is executed, else the other *aCnt := aCnt - 1*. Since the counting must be cyclical, $aCnt$ is cleared (line #12) if *UD* = '1' and count has reached '1111'; else if *UD* = '0' and counter state $aCnt$ is '0000', next state is assigned to '1111' (line #16).

Notice that the controls for possible unknown input signals make a bit more complex the interpretation of the code.

Following next's state logic, we find (line #29) the Boolean expression that assigns the output Tc (terminal count). Such expression is outside the construct that depends on nCl and the Ck. The compiler will translate it in a combinational network, function of the counter's state and ENT and UD inputs.

Last, we find the assignments of the outputs $Q3..Q0$ that copy the bits of the state variable (lines #34..37). The special operator '<=' defines the assignment of a value to an output (inputs and outputs of a process are "SIGNALs" in VHDL jargon). Instead, in the previous lines of code the assignments to the variable $aCnt$ have been defined using the operator ':='.

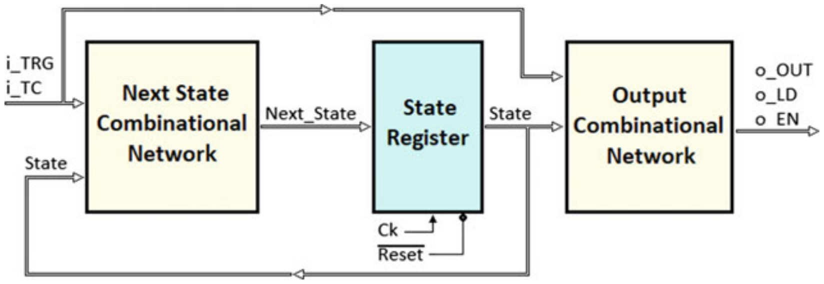
In the following, the difference between operators '<=' and ':=' will be considered. For now, it is enough to say that in VHDL the "SIGNALs" correspond to physical connections, while "VARIABLEs" concur to define the logic behavior of the network.

9.2.3 Finite State Machine

"Pulser_EN.vhd" describes the FSM with the same name but extension ".fsm" that has been designed as ASM chart. FSMs are exported in separated VHDL files (they are not included in the "Components.vhd"). Below is the "Pulser_EN" *entity declaration unit* with its input and output connections.

```
ENTITY Pulser_EN IS
  PORT( ----------------------------------->Clock & Reset:
        Ck:     IN std_logic;
        Reset:  IN std_logic;
        ----------------------------------->Inputs:
        i_TRG:  IN std_logic;
        i_TC:   IN std_logic;
        ----------------------------------->Outputs:
        o_OUT:  OUT std_logic;
        o_LD:   OUT std_logic;
        o_EN:   OUT std_logic
        -------------------------------------------
        );
END Pulser_EN;
```

Next figure describes the FSM in general terms, by the three blocks with their connections. To facilitate the identification of inputs and outputs, *Deeds* added to their names a prefix '*i_*' or '*o_*'.



The code of the *architecture body unit*, below, recalls the three blocks.

```
 1 ARCHITECTURE behavioral OF Pulser_EN IS
 2   TYPE states is ( state_rs,
 3                    state_wt,
 4                    state_ps,
 5                    dummy_11 );
 6   SIGNAL State,
 7          Next_State: states;
 8 BEGIN
 9   -- Next State Combinational Logic -------------
10   FSM: process( State, i_TRG, i_TC )
11   begin
12                    -- omissis--
13   end process;
14
15   -- State Register ------------------------------
16   REG: process( Ck, Reset )
17   begin
18                    -- omissis--
19   end process;
20
21   -- Outputs Combinational Logic ----------------
22   OUTPUTS: process( State, i_TRG, i_TC )
23   begin
24                    -- omissis--
25   end process;
26 END behavioral;
```

Note at line #6,7 the definition of *State* and *Next_State*, both declared as “SIGNAL”. The compiler will create physical connections for them.

State and *Next_State* are defined as “*states*”, an ordinal type declared in the previous line #2..5 that defines the state names (*Deeds* has renamed the states instanced in the ASM chart with a prefix “*state_*”).

The body of the code defines three processes, one for each of the blocks of the FSM general model (see lines #10, #16, and #21).

Let us first consider the *State Register* process (see below). The *sensitivity list* shows that the process is executed when a change of the input signals *Ck* and *Reset* occurs. The *state_rs* value is assigned asynchronously to *State* if the *Reset* input is ’0’, on the rising edge of the *Ck* input. If *Reset* = ’1’, *Next_State* is copied in *State*. This piece of code will compile as a parallel register that memorizes the FSM state. Note that “*rising_edge()*” is a library function.

```
-- State Register ----------------------------
REG: process( Ck, Reset )
begin
  if (Reset = '0') then
            State <= state_rs;
  elsif rising_edge(Ck) then
            State <= Next_State;
     end if;
end process;
```

The other two processes are compiled as combinational networks, because in their code all the outputs are completely specified as function of the inputs (see the next listing).

The *next state combinational logic* process declares *State*, *i_TRG*, and *i_TC* in the *sensitivity list*. The construct *case-when* is very similar to the C/C++ *switch* construct, or the Pascal *case*. In this code, the selector is *State*.

Trying to translate the code in natural language, we see that, if *State* is equal to *state_wt* (line #5), the value of the input *i_TRG* decides if *next_state* will be equal to *state_ps* or will remain equal to the current *state_wt*.

The other lines are very similar, except at line #23, that simply defines the default condition. If the current state could be different from the ones defined by design, the FSM will be forced into the reset state *state_rs*.

Note that if this last statement were missing, the compiler will understand that we would maintain memorized the outputs when none of the stated combination is there.

It will then generate a sequential circuit, instead of a combinational one. To be sure to avoid this, designers play it safe by inserting the "*when OTHERS*" clause even when not strictly necessary.

```
1  -- Next State Combinational Logic ------------
2  FSM: process( State, i_TRG, i_TC )
3  begin
4    CASE State IS
5      when state_wt =>
6                    if (i_TRG = '1') then
7                      Next_State <= state_ps;
8                    else
9                      Next_State <= state_wt;
10                   end if;
11     when state_ps =>
12                   if (i_TC = '1') then
13                     Next_State <= state_rs;
14                   else
15                     Next_State <= state_ps;
16                   end if;
17     when state_rs =>
18                   if (i_TRG = '1') then
19                     Next_State <= state_rs;
20                   else
21                     Next_State <= state_wt;
22                   end if;
23     when OTHERS =>
24                   Next_State <= state_rs;
25   END case;
26 end process;
```

The last process describes the *Output Combinational Network*, where the outputs *o_OUT*, *o_LD*, and *o_EN* are defined as function of *State*, with a *CASE* statement, in a way similar to the previous process. Note that in this example the clause "*when OTHERS*" is used too.

```
1  -- Outputs Combinational Logic ---------------
2  OUTPUTS: process( State, i_TRG, i_TC )
3  begin
4    -- Set output defaults:
5    o_OUT <= '0';
6    o_LD <= '0';
7    o_EN <= '0';
8
9    -- Set output as function of current state and input:
10   CASE State IS
11     when state_wt =>
12                 o_LD <= '1';
13     when state_ps =>
14                 o_OUT <= '1';
15                 if (i_TC = '0') then
16                   o_EN <= '1';
17                 end if;
18     when OTHERS =>
19                 o_OUT <= '0';
20                 o_LD <= '0';
21                 o_EN <= '0';
22   END case;
23 end process;
```

This piece of code allows us to get familiar with the usage of the operator '<=' in processes. For instance, pay attention to line #6, where we assign '0' to the signal *o_LD*.

Then, at line #12 the same output *o_LD* is set to '1', if *State* = *state_wt*. If this code were written in C/C++ or another procedural language, this would describe a sequence where *"before" o_LD* is reset, and *"after"*, set. But this is VHDL, and a process defines a parallel behavior in which the assignments to *signals* should be understood in a different way.

In this code, when the process *executes*, it produces a logic value on the outputs, and no delay is implicit there. The assignment *o_LD* <= '0' should be intended as a default output value for *o_LD*.

If no other logic condition applies, *o_LD* will be '0' on the process execution end. Otherwise, if a different condition will define *o_LD* at '1', this will be the value on the process execution end, without regard to the other assignment.

9.2.4 Top-Level Entity

In a VHDL project, the *top-level entity* represents the entire system and instances all the VHDL project entities defined in the project itself, in a *hierarchical* way. In the example that we have considered, the *top-level entity* is defined in the file "Pulser_DE0CV.vhd".

Deeds has generated it starting from the schematics, including in it all the needed references to the component and FSM entities, adding the description of all the connections between the blocks. Let's extract and comment a few excerpts from the code.

In the *entity declaration unit*, we find all the external connections of our circuit. *Deeds* added automatically some useful comment, just to remind the connections to the board devices.

In the example below, at line #7 the input *iTRG* has been connected, according to the user definitions, to the button 'Key[03]', through the 'PIN_M6' of the FPGA chip. Obviously, these comments are not relevant for the compiler, which will find these definitions embedded into the *Quartus*® *II* project file.

```
1  ENTITY Pulser_DE0CV IS
2    PORT(
3      iCLOCK_50MHz: IN std_logic; --> PIN_V15
4                                  --  "iCLK"  Clock: 2 Hz
5                                  --  Sw[09],LEDR[09],Key[01]
6                                  --  for Slow Clock Mode
7      iTRG:         IN std_logic; --> PIN_M6,
8                                  --> Push-Button: Key[03]
9      inReset:      IN std_logic; --> PIN_P22,
10                                 --> Push-Button: Reset
11     iTIME_03:     IN std_logic; --> PIN_T12,
12                                 --> Switch: Sw[03]
13     iTIME_02:     IN std_logic; --> PIN_T13,
14                                 --> Switch: Sw[02]
15             -- omissis
16     );
17 END Pulser_DE0CV;
```

At line #3, we see the connection to the 50-MHz native board clock. Internally, this is connected to the clock scaler circuit (note that the generated comment reminds the user setting of the clock frequency and the *Slow Clock Mode*).

All the other definitions have been omitted here to shorten the code listing and make it more readable. Indeed, the number of entries is considerable and depends on the board's resources in use.

Deeds instances all the relevant output connections, even if not directly used by our project, to switch off all the displays, LEDs, and other unused output devices during the circuit test.

The code of the *architecture body unit*, shown in the next listing, declares the components used by the entity, for instance, the counter "Counter4b" (line #3), the FSM "Pulser_EN" (line #8), and other ones (here omitted).

The architecture description is *"structural"*, so all the SIGNALs used are declared together with the mapping of all the connections among the components. For instance, at line #13, a SIGNAL of name 'S001' is declared (the code generation is automated, so *Deeds* assign a different name to each net using its internal netlist identifier).

```
ARCHITECTURE structural OF Pulser_DE0CV IS
                           -- omissis
  COMPONENT Counter4b IS
    PORT(                  -- omissis
          );
  END COMPONENT;

  COMPONENT Pulser_EN IS
    PORT(                  -- omissis
          );
  END COMPONENT;

  SIGNAL S001: std_logic;
                           -- omissis
  SIGNAL S017: std_logic;

BEGIN
                           -- omissis
END structural;
```

As an example, you see below the declaration of the *interface* of FSM component “Pulser_EN”, which was shortened in the previous listing.

```
COMPONENT Pulser_EN IS
  PORT( ------------------------>Clock & Reset:
        Ck:    IN std_logic;
        Reset: IN std_logic;
        ------------------------>Inputs:
        i_TRG: IN std_logic;
        i_TC:  IN std_logic;
        ------------------------>Outputs:
        o_OUT: OUT std_logic;
        o_LD:  OUT std_logic;
        o_EN:  OUT std_logic
        -------------------------
        );
END COMPONENT;
```

Let’s examine now the body of the architecture, shortened to focus the attention on the relevant elements. At line #3 and following, a few examples of usage of the ’<=’ operator , used outside processes, represent a simple *wired* connection.

```
 1 BEGIN
 2                        -- omissis --
 3    S005 <= iCLK;
 4    S007 <= inReset;
 5                        -- omissis --
 6    oQ3_Q0_00 <= S010;
 7    oQ3_Q0_01 <= S011;
 8                        -- omissis --
 9    S004 <= '0';
10    S001 <= '1';
11                        -- omissis --
12    C680: Counter4b PORT MAP( S005, S007, S003, S006,
13                              S001, S004, S017, S016,
14                              S015, S014, S013, S012,
15                              S011, S010, S009 );
16    C704: Pulser_EN PORT MAP( S005, S007, S002, S009,
17                              S008, S003, S006 );
18 END structural;
```

Line #3 means that the input *iCLK* is connected to the internal SIGNAL named 'S005'. In a similar way, at line #6 the internal net 'S010' is connected to the output *oQ3_Q0_00*. As an example of constant setting, the internal SIGNAL 'S004' is set to '0' (line #9).

The architecture body ends with the effective connection of the components to the internal nets. In the VHDL jargon, this operation is named *"mapping"* and is obtained here with the statement "PORT MAP".

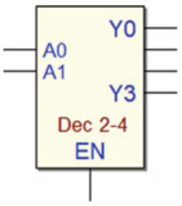
In this example, on line #17 an instance named 'C704' of the "Pulser_EN" component is connected to the internal SIGNALs declared in the arguments. Their order corresponds to that of the component declaration, so, for instance, its *Reset* terminal is connected to the SIGNAL 'S007'.

9.2.5 Other VHDL Examples

In the following, a few examples of VHDL code describe frequently used networks.

Decoder

A decoder activates the output corresponding to the input binary code, assuming the Enable input is active (as seen in Sect. 2.6.1). Below is the symbol of a $2 \rightarrow 4$ decoder and its *entity declaration unit*, as defined in *Deeds*.



```
ENTITY Decoder_2_4 IS
  PORT( A1: IN   std_logic;
        A0: IN   std_logic;
        EN: IN   std_logic;
        Y0: OUT  std_logic;
        Y1: OUT  std_logic;
        Y2: OUT  std_logic;
        Y3: OUT  std_logic );
END Decoder_2_4;
```

The PORT declaration lists inputs *A1*, *A0*, *EN*, and outputs *Y0..Y3*. Below is a possible component's description using the construct *with-select-when*.

In this case, we did not use a process to describe the behavior. The language allows a considerable freedom of choice; in this case, it is convenient to adopt a descriptive approach, basically similar to a truth table.

In the VHDL code generated by *Deeds*, below, line #2 defines *aNumber* as a *vector* of *three wires* (indexed as 2,1,0), which on line #3 gathers together the three inputs *EN*, *A1*, and *A0*.

The binary value of *aNumber*, i.e., the group of the three input wires, selects the active output.

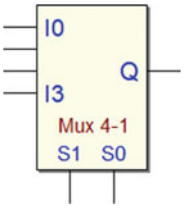
```
1  ARCHITECTURE behavioral OF Decoder_2_4 IS
2    SIGNAL aNumber: std_logic_vector( 2 downto 0 );
3  BEGIN
4    aNumber <= EN & A1 & A0;
5    with aNumber select
6      Y0 <= '0' when "000", '0' when "001",
7            '0' when "010", '0' when "011",
8            '1' when "100", '0' when "101",
9            '0' when "110", '0' when "111", 'X' when others;
10   with aNumber select
11     Y1 <= '0' when "000", '0' when "001",
12           '0' when "010", '0' when "011",
13           '0' when "100", '1' when "101",
14           '0' when "110", '0' when "111", 'X' when others;
15   with aNumber select
16     Y2 <= '0' when "000", '0' when "001",
17           '0' when "010", '0' when "011",
18           '0' when "100", '0' when "101",
19           '1' when "110", '0' when "111", 'X' when others;
20   with aNumber select
21     Y3 <= '0' when "000", '0' when "001",
22           '0' when "010", '0' when "011",
23           '0' when "100", '0' when "101",
24           '0' when "110", '1' when "111", 'X' when others;
25 END behavioral;
```

The rest of the code assigns a value to each output, on the bases of the selection dictated by *aNumber*. For instance, at lines #6..9, the output *Y0* is assigned to '0' for all the combinations of *aNumber*, except *when* it is equal to '100' (i.e., when *EN* = '1', *A1* = '0' and *A0* = '0').

Remember that the assignment operator '<=', when used outside the processes, represents a connection. The clause *'X' when others* is introduced only for simulation purposes, and it is not necessary for synthesis.

Multiplexer

As described in Sect. 2.6.2, a multiplexer selects which one of the inputs will be copied in the output, according to the binary value of the selection inputs. The 4 → 1 multiplexer symbol and the corresponding *entity declaration unit* follow.



```
ENTITY Multiplexer_4_1 IS
  PORT( I0: IN  std_logic;
        I1: IN  std_logic;
        I2: IN  std_logic;
        I3: IN  std_logic;
        S1: IN  std_logic;
        S0: IN  std_logic;
         Q: OUT std_logic );
END Multiplexer_4_1;
```

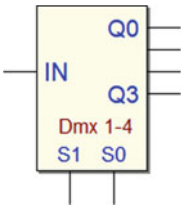
In this case, it is convenient to use the construct *when-else*. The behavior of the component is easily readable in the following code.

```
ARCHITECTURE behavioral OF Multiplexer_4_1 IS
BEGIN
 Q <= I0 when ((S1 = '0') and (S0 = '0')) else
      I1 when ((S1 = '0') and (S0 = '1')) else
      I2 when ((S1 = '1') and (S0 = '0')) else
      I3 when ((S1 = '1') and (S0 = '1')) else 'X';
END behavioral;
```

The output *Q* takes the value of one of the inputs *I0..I3*, according to the combinations of the input *S1* and *S0*, this time represented as logic expressions. For instance, *Q* copies *I2* if (*S1* = '1') and (*S0* = '0').

Demultiplexer

Input *IN* is copied on the output selected by *S1* and *S0* (see Sect. 2.6.3). An example of a 1 → 4 demultiplexer follows.



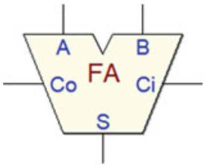
```
ENTITY Demultiplexer_1_4 IS
  PORT(  I: IN  std_logic;
        S1: IN  std_logic;
        S0: IN  std_logic;
        Q0: OUT std_logic;
        Q1: OUT std_logic;
        Q2: OUT std_logic;
        Q3: OUT std_logic );
END Demultiplexer_1_4;
```

A demultiplexer is the same as a decoder with enable input; therefore, the implementation of both is identical.

```
ARCHITECTURE behavioral OF Demultiplexer_1_4 IS
  SIGNAL aNumber: std_logic_vector( 2 downto 0 );
BEGIN
  aNumber <= I & S1 & S0;
  with aNumber select
    Q0 <= '0' when "000", '0' when "001",
          '0' when "010", '0' when "011",
          '1' when "100", '0' when "101",
            '0' when "110", '0' when "111", 'X' when others;
                    -- omissis ...
END behavioral;
```

Full Adder

The component, described in Sect. 3.9.2, adds two bits and the input carry and generates the sum and the output carry. On the right is the VHDL declaration of its terminals.



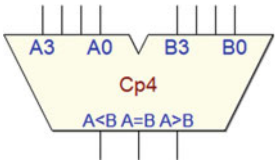
```
ENTITY Adder_Full IS
  PORT( CIN:  IN  std_logic;
        COUT:OUT  std_logic;
        A:    IN  std_logic;
        B:    IN  std_logic;
        S:    OUT std_logic );
END Adder_Full;
```

The construct *with-select-when* produces a list that reflects the component's truth table. *ABC*, defined as *vector* of three *signals*, groups together inputs *A*, *B*, and *CIN* (line #4).

```
1  ARCHITECTURE behavioral OF Adder_Full IS
2    SIGNAL ABC: std_logic_vector( 2 downto 0 );
3  BEGIN
4    ABC <= A & B & CIN;
5    --
6    with ABC select
7    S  <= '0' when "000",
8          '1' when "001",
9          '1' when "010",
10         '0' when "011",
11         '1' when "100",
12         '0' when "101",
13         '0' when "110",
14         '1' when "111",
15         'X' when others;
16   --
17   with ABC select
18   COUT  <= '0' when "000",
19            '0' when "001",
20            '0' when "010",
21            '1' when "011",
22            '0' when "100",
23            '1' when "101",
24            '1' when "110",
25            '1' when "111",
26            'X' when others;
27 END behavioral;
```

Magnitude Comparator

It is a component used to compare the magnitude of two unsigned integer numbers (encountered in Sect. 8.3.6). The figure shows the 4-bit version. On the right is the corresponding entity in VHDL, as generated by *Deeds*.



```
ENTITY Compar_4 IS
  PORT( A3:  IN   std_logic;
        A2:  IN   std_logic;
        A1:  IN   std_logic;
        A0:  IN   std_logic;
        B3:  IN   std_logic;
        B2:  IN   std_logic;
        B1:  IN   std_logic;
        B0:  IN   std_logic;
        MIN: OUT  std_logic;
        EQU: OUT  std_logic;
        MAJ: OUT  std_logic );
END Compar_4;
```

MIN is asserted when operand *A* is lower than operand *B*.

MAJ is set to one in the opposite case, and *EQU* when they are equal.

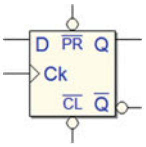
This description is easily readable also in the architecture description, shown below, where a process construct is used.

```
 1  ARCHITECTURE behavioral OF Compar_4 IS
 2  BEGIN
 3    Cmp4: PROCESS( A3, A2, A1, A0,
 4                   B3, B2, B1, B0 )
 5    variable A: unsigned( 3 downto 0 );
 6    variable B: unsigned( 3 downto 0 );
 7    BEGIN
 8      A := (A3 & A2 & A1 & A0);
 9      B := (B3 & B2 & B1 & B0);
10      --
11      if    (A > B) then MIN <= '0'; EQU <= '0'; MAJ <= '1';
12      elsif (A < B) then MIN <= '1'; EQU <= '0'; MAJ <= '0';
13      elsif (A = B) then MIN <= '0'; EQU <= '1'; MAJ <= '0';
14      else               MIN <= 'X'; EQU <= 'X'; MAJ <= 'X';
15      END IF;
16    END PROCESS;
17  END behavioral;
```

In the process body, two variables (*A* and *B*) group together the corresponding input wires (lines #8..9). In this way, we can compare in algebraic mode the variables and assert, or not, the output coherently (lines #11..13).

Flip-flop D-PET

Below, the VHDL description of a D-PET-type flip-flop, with $\overline{Clear}$ and $\overline{Preset}$.



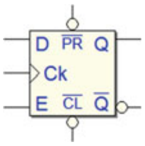
```
ENTITY DpetFF IS
  PORT(  D, Ck    : IN  std_logic;
         nCL, nPR: IN  std_logic;
         Q, nQ    : OUT std_logic );
END DpetFF;
```

Note that the *D* input is not in the sensitivity list (line #3), because a change of that input only will not modify the flip-flop state. Lines #5..7 evaluate the asynchronous inputs $\overline{Clear}$ and $\overline{Preset}$ (*nCL* and *nPR*), changing the flip-flop outputs independently of the clock. If they are not active (line #8), on the positive clock edge the output *Q* will copy the value of *D*.

```
1  ARCHITECTURE behavioral OF DpetFF IS
2  BEGIN
3    Dff: PROCESS( Ck, nCL, nPR )
4    BEGIN
5      if    (nCL='0') and (nPR='0') then Q <= 'X'; nQ <= 'X';
6      elsif (nCL='0') and (nPR='1') then Q <= '0'; nQ <= '1';
7      elsif (nCL='1') and (nPR='0') then Q <= '1'; nQ <= '0';
8      elsif (nCL='1') and (nPR='1') then
9        if (Ck'event) AND (Ck='1') THEN -- Positive Edge
10          Q <=  D;  nQ <= not D;
11       END IF;
12     else Q <= 'X'; nQ <= 'X';
13     END IF;
14   END PROCESS;
15 END behavioral;
```

Flip-Flop E-PET

The VHDL description of an E-PET-type flip-flop is obviously very similar to the one of the D-PET, with the addition of the input *E* condition.



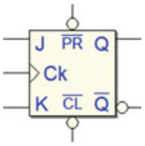
```
1 ENTITY EpetFF IS
2   PORT(  D, E, Ck: IN std_logic;
3              nCL, nPR: IN std_logic;
4              Q, nQ   : OUT std_logic );
5 END EpetFF;
```

At line #12, the flip-flop outputs are updated on the positive clock edge only if *E* is active. Note that the line immediately below is there only for simulation purposes and states that if the input *E* is unknown, the outputs will be too.

```
1  ARCHITECTURE behavioral OF EpetFF IS
2  BEGIN
3    Eff: PROCESS( Ck, nCL, nPR )
4    BEGIN
5      if    (nCL='0') and (nPR='0') then Q <= ‘X'; nQ <= ‘X';
6      elsif (nCL='0') and (nPR='1') then Q <= '0'; nQ <= '1';
7      elsif (nCL='1') and (nPR='0') then Q <= '1'; nQ <= '0';
8      elsif (nCL='1') and (nPR='1') then
9        if (Ck'event) AND (Ck='1') THEN -- Positive Edge
10         if       (E = '1') then Q <=  D;  nQ <= not D;
11         elsif not(E = '0') then Q <= 'X'; nQ <= 'X';
12         END IF;
13       END IF;
14     else Q <= 'X'; nQ <= 'X';
15     END IF;
16   END PROCESS;
17 END behavioral;
```

Flip-Flop JK-PET

The VHDL of JK-PET flip-flop is similar to the previous two.



```
1 ENTITY JKpetFF IS
2   PORT(  J, K, Ck: IN std_logic;
3          nCL, nPR: IN std_logic;
4          Q, nQ   : OUT std_logic );
5 END JKpetFF;
```

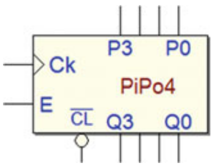
The presence of the JK *Toggle* modality needs a variable representing the state of the flip-flop (line #2).

```
1  ARCHITECTURE behavioral OF JKpetFF IS
2  BEGIN
3    JKff: PROCESS( Ck, nCL, nPR )
4      variable  OutQ: STD_LOGIC;
5    BEGIN
6      if    (nCL='0') and (nPR='1') then OutQ := '0';
7      elsif (nCL='1') and (nPR='0') then OutQ := '1';
8      elsif (nCL='1') and (nPR='1') then
9        if (Ck'event) AND (Ck='1') THEN
10         -- Positive Edge
11         if    (J = '0') AND (K = '1') THEN OutQ := '0';
12         elsif (J = '1') AND (K = '0') THEN OutQ := '1';
13         elsif (J = '1') AND (K = '1') THEN OutQ := not OutQ;
14         elsif not((J='0')AND(K='0'))  THEN OutQ := 'X';
15         END IF;
16       END IF;
17     else                                   OutQ := 'X';
18     END IF;
19     --
20     Q  <= (    OutQ);
21     nQ <= (not OutQ);
22     --
23   END PROCESS;
24 END behavioral;
```

On the positive clock edge, the J and K inputs are evaluated (lines #11..15) and the flip-flop state is updated. The output Q and $\overline{Q}$ are then updated on the end of the process code (lines #20..21).

Parallel Register

We have already encountered the VHDL of a parallel register, when we considered the *State Register* process of the FSM. Here is presented an example of a more general coding, defining a register with $Enable$ and $\overline{Clear}$ inputs.



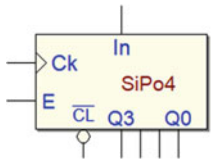
```
ENTITY PiPoE4 IS
  PORT( Ck  : IN std_logic;
        nCL: IN std_logic;
        E   : IN std_logic;
        P3  : IN std_logic;
        P2  : IN std_logic;
        P1  : IN std_logic;
        P0  : IN std_logic;
        Q3  : OUT std_logic;
        Q2  : OUT std_logic;
        Q1  : OUT std_logic;
        Q0  : OUT std_logic );
END PiPoE4;
```

To group together the register bits, we introduce the variable *aReg* (line #3). If $\overline{Clear}$ input is active (line #6), the variable is cleared. Else, on the positive clock edge, if the enable input *EN* is active (line #8 and 9), the input values $P3..P0$ are assigned to the variable. The statements at lines #18..21 assign to the register outputs the updated variable bits.

```
1  ARCHITECTURE behavioral OF PiPoE4 IS
2  BEGIN
3    RegPiPoE4: PROCESS( Ck, nCL )
4      variable aReg: std_logic_vector( 3 downto 0 );
5    BEGIN
6      if    (nCL = '0') then    aReg := (others =>'0');
7      elsif (nCL = '1') then
8        if (Ck'event) AND (Ck='1') THEN -- Positive Edge
9          if (E = '1') then
10                 aReg := (P3 & P2 & P1 & P0);
11         elsif not(E = '0') then
12                 aReg := (others =>'X');
13         END IF;
14       END IF;
15     else        aReg := (others =>'X');
16     END IF;
17
18     Q3 <= aReg(3);
19     Q2 <= aReg(2);
20     Q1 <= aReg(1);
21     Q0 <= aReg(0);
22
23   END PROCESS;
24 END behavioral;
```

Shift Register (S.I.P.O.)

An example of *Serial-Input Parallel-Output* shift register is shown on the figure. This 4-bit register has an *EN* input that enables the shifting. Data enters through the input *In* and shifts from *Q3* to *Q0*.



```
ENTITY SiPoE4 IS
  PORT( I   : IN std_logic;
        Ck  : IN std_logic;
        nCL : IN std_logic;
        E   : IN std_logic;
        Q3  : OUT std_logic;
        Q2  : OUT std_logic;
        Q1  : OUT std_logic;
        Q0  : OUT std_logic );
END SiPoE4;
```

The VHDL description style resembles that of the parallel register see before, with the difference that, on the positive clock edge (and if the enable *E* is active), the variable bits are assigned coherently with the required function.

```
1  ARCHITECTURE behavioral OF SiPoE4 IS
2  BEGIN
3    RegSiPoE4: PROCESS( Ck, nCL )
4    variable aReg: std_logic_vector( 3 downto 0 );
5    BEGIN
6      if    (nCL = '0') then aReg := (others =>'0');
7      elsif (nCL = '1') then
8        if (Ck'event) AND (Ck='1') THEN -- Positive Edge
9          if (E = '1') then
10           aReg := (I & aReg(3) & aReg(2) & aReg(1));
11         elsif not(E = '0') then
12           aReg := (others =>'X');
13         END IF;
14       END IF;
15     else aReg := (others =>'X');
16     END IF;
17     --
18     Q3 <= aReg(3);
19     Q2 <= aReg(2);
20     Q1 <= aReg(1);
21     Q0 <= aReg(0);
22     --
23   END PROCESS;
24 END behavioral;
```

On line #10, the new variable value is constructed joining the input *In* (*I* in the code) and the current bits in position 3, 2, and 1.

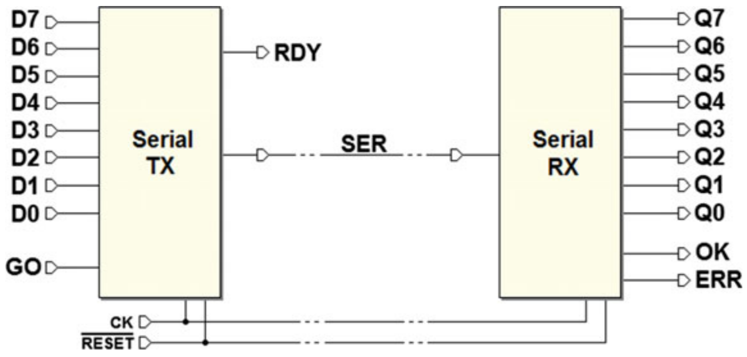
After the clock edge, we obtain the copy on the input in the higher bit, and the other ones shifted to the right.

9.3 FPGA Prototyping Exercises

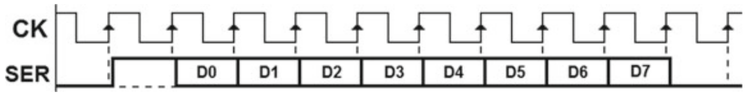
This section offers a few exercises of system project and prototyping on FPGA. Others are included in the *Deeds* Web site's *Learning Materials*.

9.3.1 Synchronous Serial Communication System (8-bit)

Complete the design of a *synchronous serial communication system*, and then implement it on a FPGA board.



The TX unit reads an eight-bit parallel data (*D0..D7*); when the input *GO* goes from 0 to 1, data is serialized and transmitted on the line *SER*, in the format shown in the following figure:



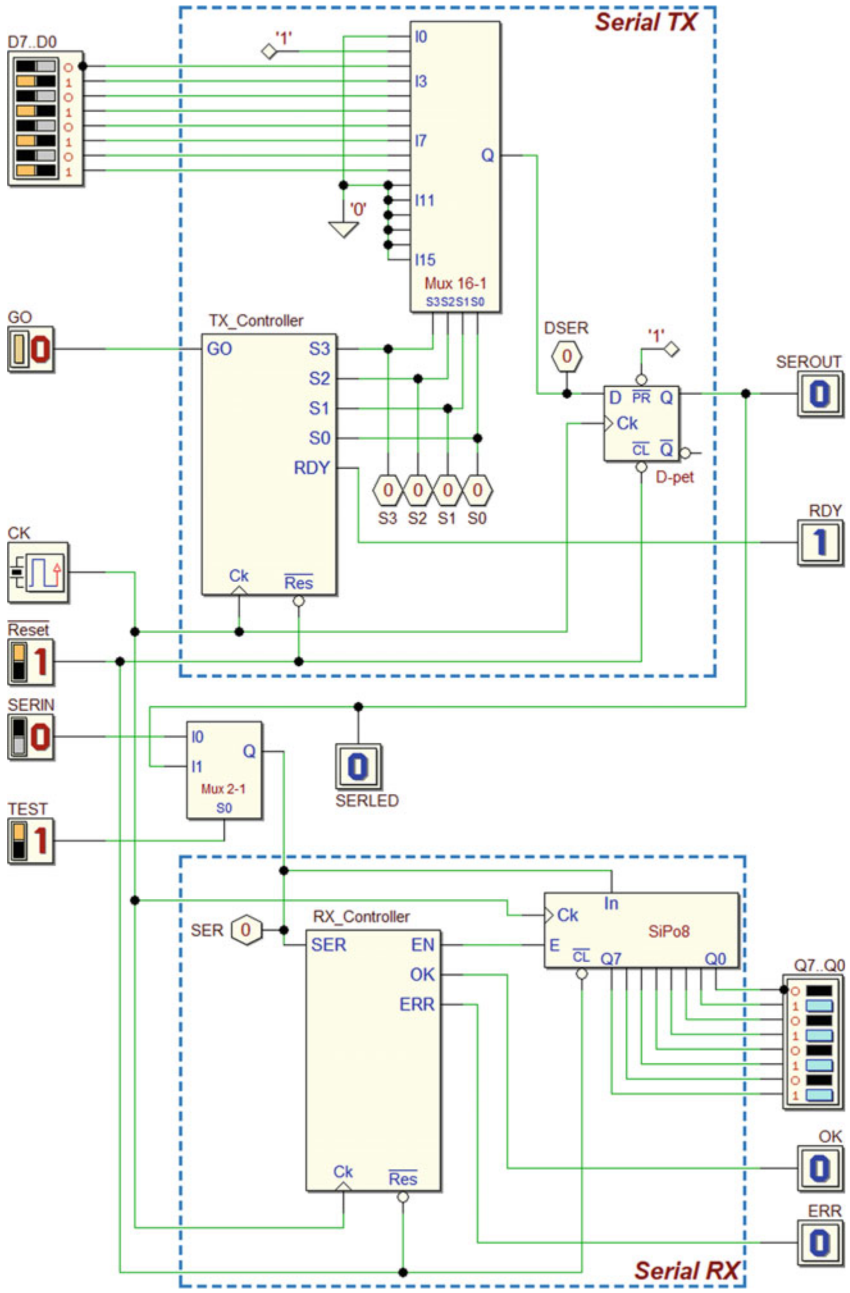
- The bit sequence is synchronous with the clock CK.
- The bit time duration is equal to one clock cycle.
- Each sequence begins with a start bit (high).
- Eight data bits follow (D0..D7).
- The bit packet ends with a Stop Bit (low).

The output *RDY* is activated when the transmitter is waiting for a *low to high transition* on the input *GO*.

The receiver RX waits for packets on the line *SER*. When a start bit is detected, the receiver processes the serial sequence and copies the *D0..D7* values on the outputs *Q0..Q7*.

The stop bit is evaluated. If correctly received, the system activates the *OK* output for the duration of a clock cycle, otherwise the receiver sets the *ERR* output, maintaining it active until *SER* returns to 0.

The proposed architecture is shown below (a template of the schematic and the controllers' FSM are available for downloading from *Deeds* Web site).



TX Design Guidelines

The TX circuit is divided into the usual structure *controller–datapath*, where the latter is composed of a multiplexer 16→1 and a D-PET flip-flop.

While waiting for the *GO* command, the transmitter generates a low level *(the idle state of the line)*.

On the positive edge of *GO*, it transmits sequentially on *SER* a '1' *(the start bit)*, the eight bits *D0..D7*, starting from *D0*, and finally a '0' *(the stop bit)*. Then, it waits for the next *GO* command.

A "Mux16-1" multiplexer (from *Deeds* library) provides the data, under the control of the FSM using the selection lines *S3..S0*. As shown in the previous figure, the multiplexer inputs are connected in the following order:

- $I0$ = low (the idle line);
- $I1$ = high (the start bit);
- $I2..I9$ = *D0..D7* (the data bits);
- $I10$ = low (the stop bit);
- $I11..I15$ = low (not used).

To grant the synchronicity of the output with the clock *CK*, the multiplexer output is fed to a D-PET flip-flop, which drives the output line *SER*.

RX Design Guidelines

The RX datapath is represented by an eight-bit shift register ("SiPo8"), used to deserialize and store the data received on *SER*.

On the rising edge of the clock, if the register enable input *E* is active, data shifts by a position ($In \rightarrow Q7\ \ldots \rightarrow Q1 \rightarrow Q0$). Otherwise, the outputs *Q0..Q7* remain unchanged.

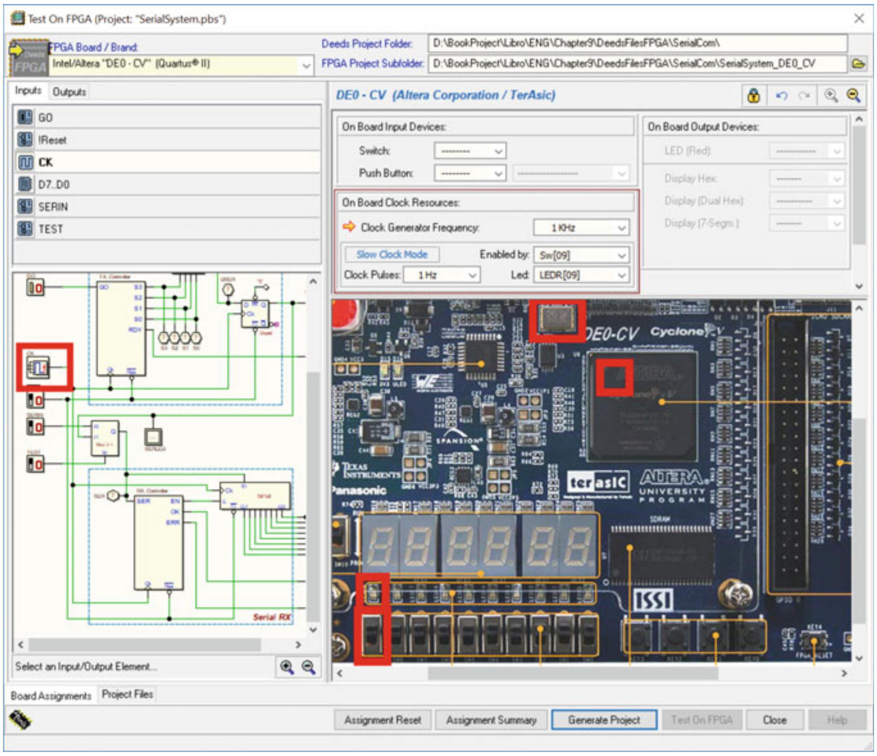
The controller synchronizes the shift operations, enabling the register when needed by activating the *EN* line. The controller waits for the start bit and then enables data shifting in the register, for each data bit.

Finally, the controller evaluates the stop bit, activating *OK* or *ERR*, according to the specifications.

Note that a multiplexer 2→1 was added between the transmitter and the receiver. The *TEST* input, when activated, allows us to connect the receiver directly to the transmitter to allow stand-alone testing.

Testing the Design on FPGA

If the circuit schematic template available on *Deeds* Web site is used, the I/O association for an *Terasic/Altera* DE0-CV board is already configured.


Test On FPGA (Project: "SerialSystem.pbs")

FPGA Board / Brand: Intel/Altera "DE0 - CV" (Quartus® II)

Deeds Project Folder: D:\BookProject\Libro\ENG\Chapter9\DeedsFilesFPGA\SerialCom\

FPGA Project Subfolder: D:\BookProject\Libro\ENG\Chapter9\DeedsFilesFPGA\SerialCom\SerialSystem_DE0_CV

Inputs: GO, !Reset, CK, D7..D0, SERIN, TEST

DE0 - CV (Altera Corporation / TerAsic)

On Board Input Devices: Switch: --------- ; Push Button: ---------

On Board Clock Resources: Clock Generator Frequency: 1 KHz

Slow Clock Mode — Enabled by: Sw[09]

Clock Pulses: 1 Hz — Led: LEDR[09]

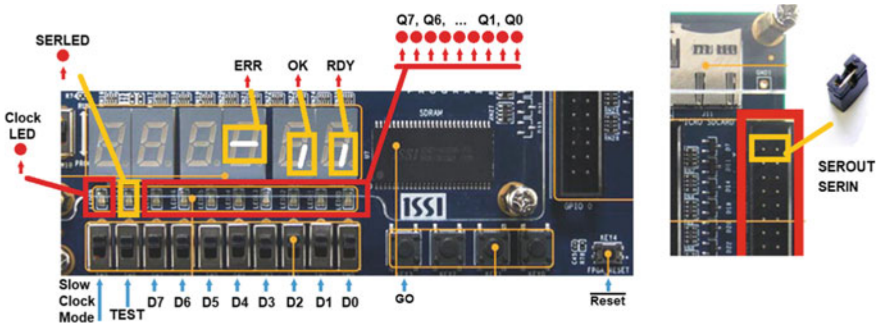
On Board Output Devices: LED (Red), Display Hex, Display (Dual Hex), Display (7-Segm.)

Select an Input/Output Element...

As visible above, the clock frequency set in the template is 1 KHz. It is possible to slow down the clock thanks to the *Slow Clock Mode* feature.

We set the switch "Sw[09]" to activate the mode, and the LED "LEDR[09]" visualizes the clock pulses. During the test, if the switch is at '0', the clock works at the regular frequency. If at '1', the clock slows down to 1 Hz, as defined in the setup.

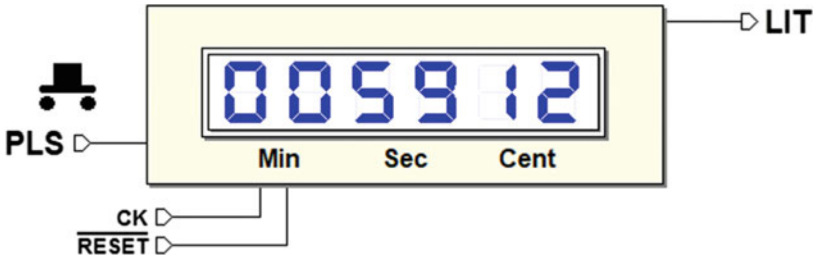
To interact with the board resources, it could be convenient to refer to the figure below, which summarizes the associations.



On right side of the figure are represented also the *SEROUT* and *SERIN* terminals, made available on the expansion header. They are useful to connect the transmitter to the receiver using a jumper or a cable (in this way, for example, we can test what happens if the line is disconnected during the test operations, or if the line is very long and exposed to external noise).

9.3.2 Digital Chronometer

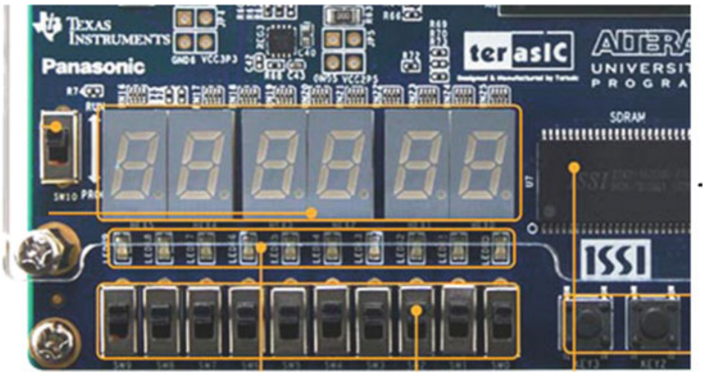
We design and test on FPGA a *Digital Chronometer* (see figure below). Requested resolution is one hundredth of a second, maximum time measurable about one hour. Time is displayed with six decimal figures, two for the minutes, two for the seconds, and two for the hundredths of second.



The system has a push-button (*PLS*) and a lamp bulb (*LIT*). At system reset all displays' digits are set to zero. *LIT* is on for all the time *PLS* is pushed.

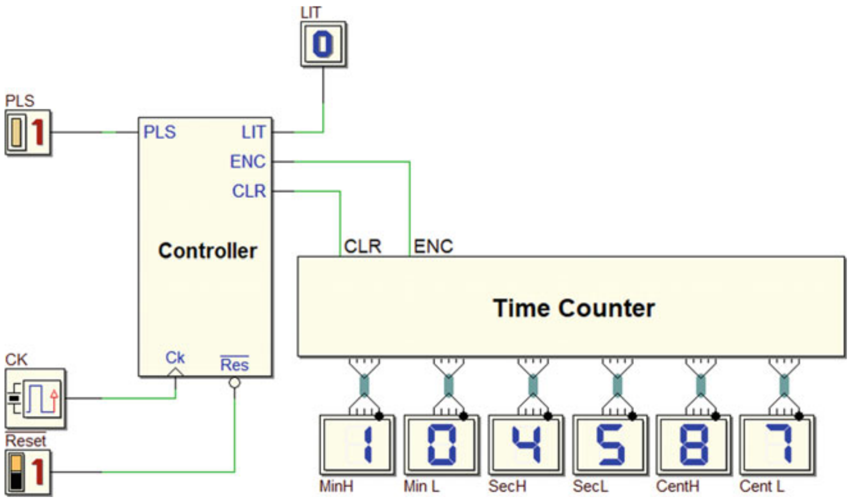
The time counting starts when *PLS* is pushed and then released. The second pressure on *PLS* stops counting, and the time elapsed can be read on the displays. The third pressure resets the display, and the timer waits for *PLS* to start a new counting sequence.

We assume to have available a clock *CK* with 100 Hz frequency and a FPGA board, such as the *Terasic/Altera* DE0-CV with six *seven-segment displays*.



Design Guidelines

We suggest to use for the chronometer the structure *controller–datapath*. The latter is composed by the *time counter* and the associated display. The former handles the push-button functionality, the lamp, and the counter controls.



Observe that the *time counter* needs only two control inputs, *CLR* (to clear its contents) and *ENC* (to enable/stop counting). The controller can be implemented as a FSM that reads the push-button through the *PLS* input, activates the lamp with the output *LIT*, and generates the control signals *CLR* and *ENC* for the time counter.

We suggest to design a controller FSM that tracks the *PLS* input value, acting on its level changes as required by the specifications.

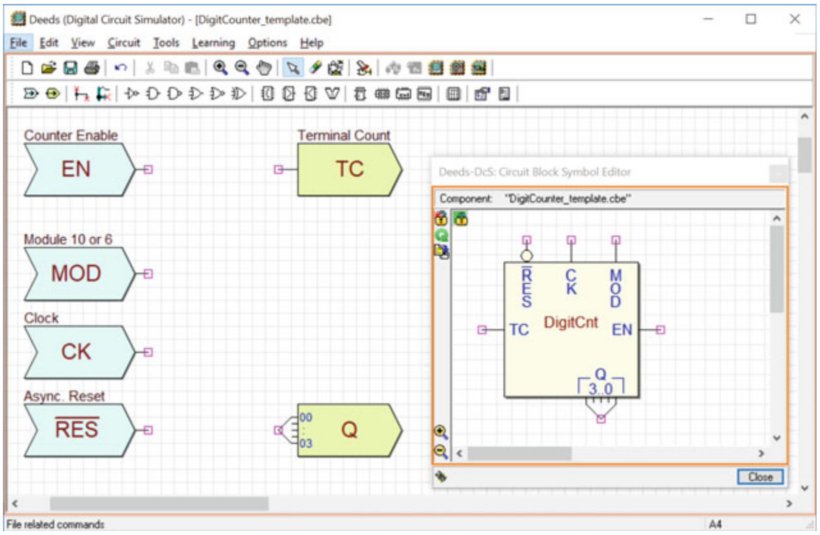
Time Counter

The time counter can be designed in many different ways. We suggest to separate the counter in six *Binary Coded Decimal* (BCD) elements, each for every digit, defining in *Deeds* a *Circuit Block Element* (CBE) block.

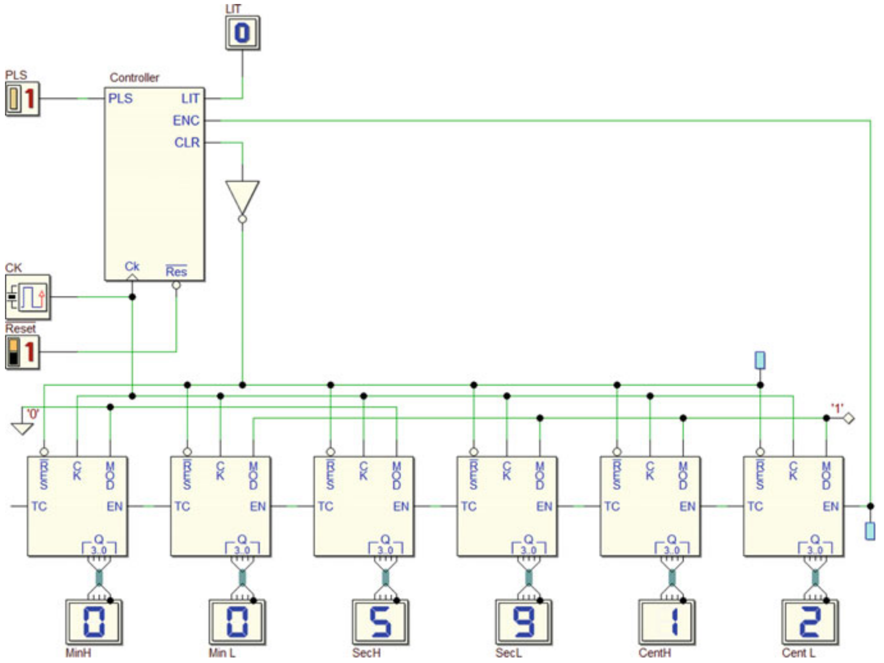
Following this approach, each digit will be driven by its own counter, receiving a count enable input *EN* from the component on the right side, and generating a *terminal count TC* to increment the digit placed on its left. Take also in account that the *seconds* and *minutes* should be counted *module 60*, so two of the blocks must generate *TC* on the '5' digit.

In the next figure, the CBE component to be completed, as available in the *digital contents* of this book, opened in the *Deeds-DcS* circuit editor.

The proposed CBE shows an added input *MOD* allowing to set the counting *module* (i.e., equal to 10 if *MOD* = *'1'*, or 6 if *MOD* = *'0'*). The CBE receives also, as usual, the inputs *CK* and $\overline{RES}$.


Deeds (Digital Circuit Simulator) - [DigitCounter_template.cbe]
File Edit View Circuit Tools Learning Options Help
Counter Enable
EN
Terminal Count
TC
Module 10 or 6
MOD
Clock
CK
Async. Reset
RES
Q
Deeds-DcS: Circuit Block Symbol Editor
Component: "DigitCounter_template.cbe"
RES CK MOD
TC DigitCnt EN
Q 3..0
Close
File related commands
A4

Following this *modular* approach, the overall schematic appears as follows.

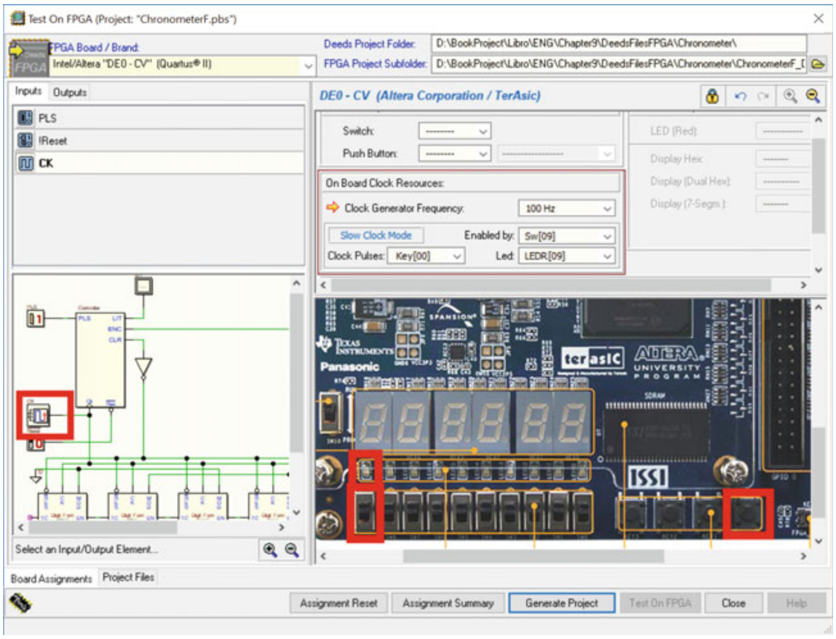


The CBE design, in turn, can be defined in different ways. A first, intuitive method can be to connect a 4-bit binary counter with a few gates around, to implement the module selection logic and the generation of *TC*. In this case, the approach must stand on the experience of the designers and their own creativity.

It is preferable to use a more systematic approach, designing the module in terms of FSM, therefore defining its behavior in an algorithmic way (as seen, for instance, in Sect. 7.2.2).

Testing the Chronometer on FPGA

A circuit to be completed is available on the *digital contents* of the book. If this file is used, the *I/O associations* are already configured (for an *Altera/Terasic* DE0-CV board). Following the specifications, the clock frequency is set to 100 Hz. The *“Slow Clock Mode”* setting allows us to feed the clock manually, for testing, pressing the button “Key[00]”, if the switch “Sw[09]” is set to ’1’. Normal operations will take place if the switch is left at ’0’.



Test On FPGA (Project: "ChronometerF.pbs")

FPGA Board / Brand: Intel/Altera "DE0 - CV" (Quartus® II)

Deeds Project Folder: D:\BookProject\Libro\ENG\Chapter9\DeedsFilesFPGA\Chronometer\

FPGA Project Subfolder: D:\BookProject\Libro\ENG\Chapter9\DeedsFilesFPGA\Chronometer\ChronometerF_[

DE0 - CV (Altera Corporation / TerAsic)

On Board Clock Resources:

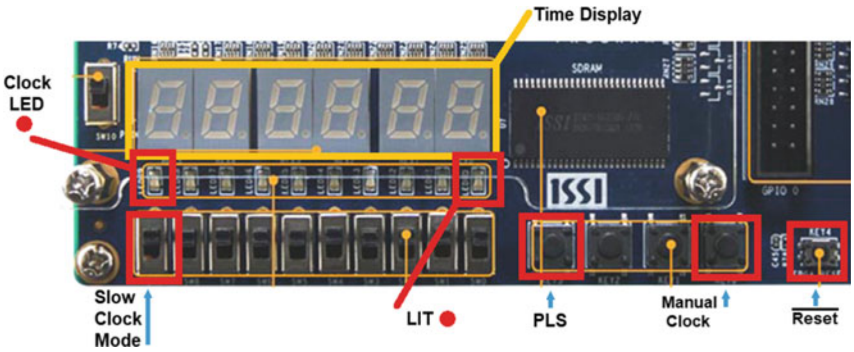
Clock Generator Frequency: 100 Hz

Slow Clock Mode — Enabled by: Sw[09]

Clock Pulses: Key[00] — Led: LEDR[09]

Select an Input/Output Element...

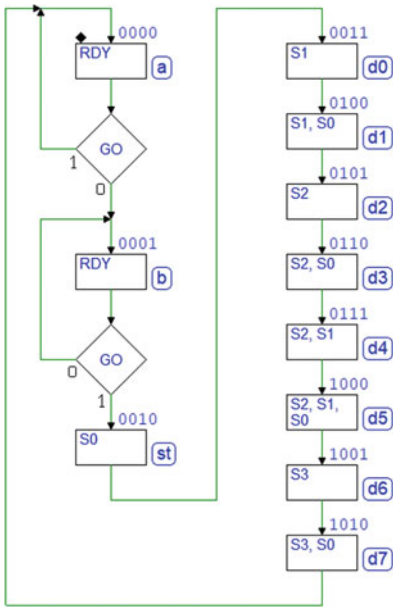
The following figure can be useful to interact with the board controls resources during the circuit test.



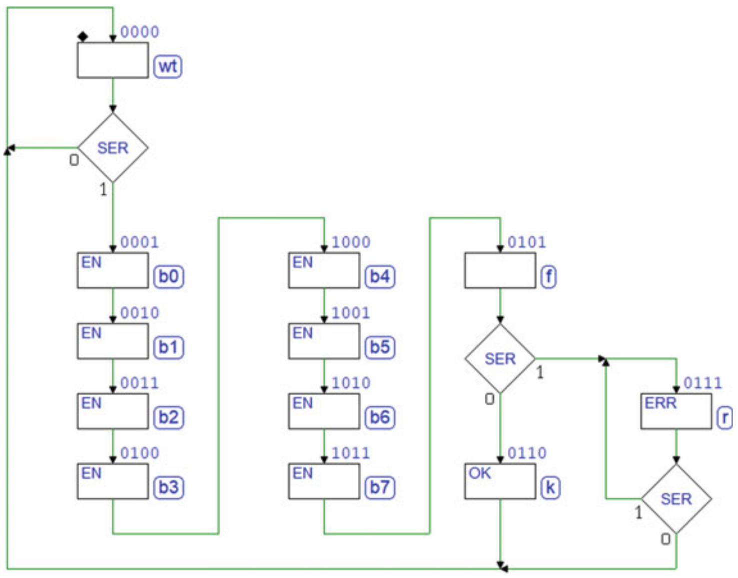
9.4 Solutions

9.4.1 Synchronous Serial Communication System (8-bit)

ASM Diagram of the Transmitter's Controller

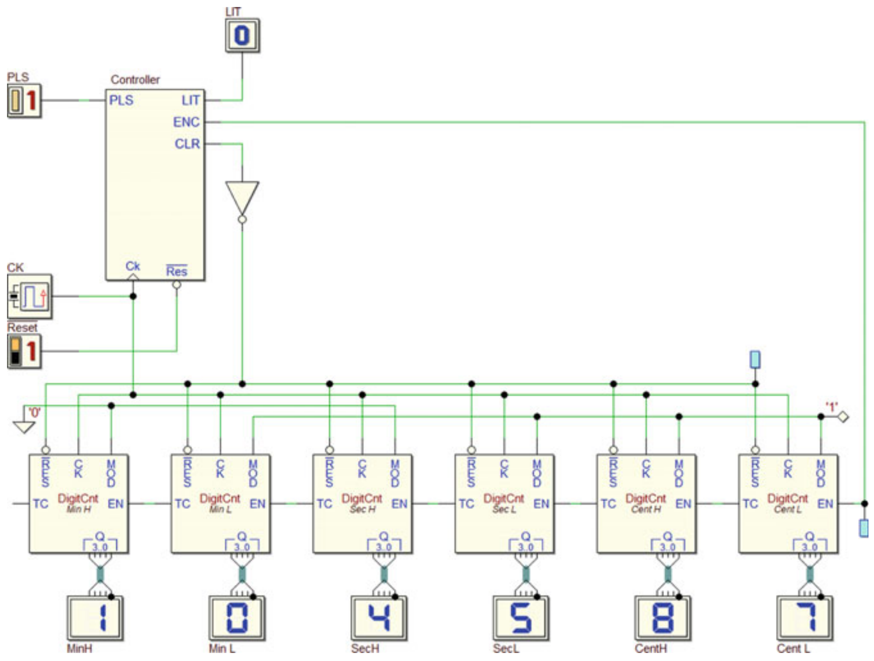


ASM Diagram of the Receiver's Controller

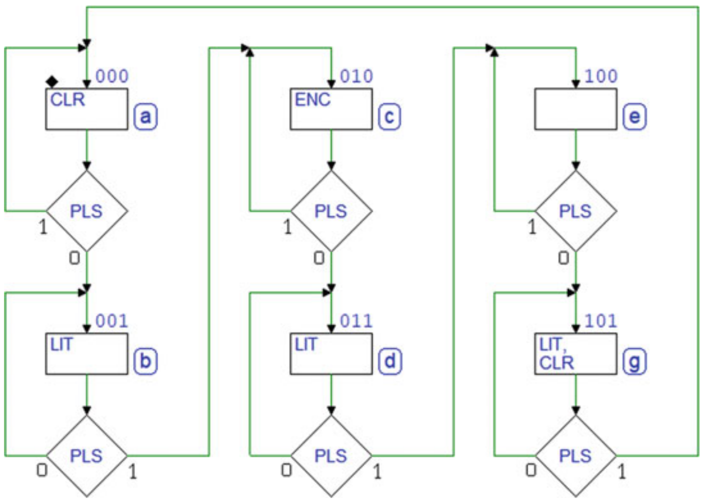


9.4.2 Digital Chronometer

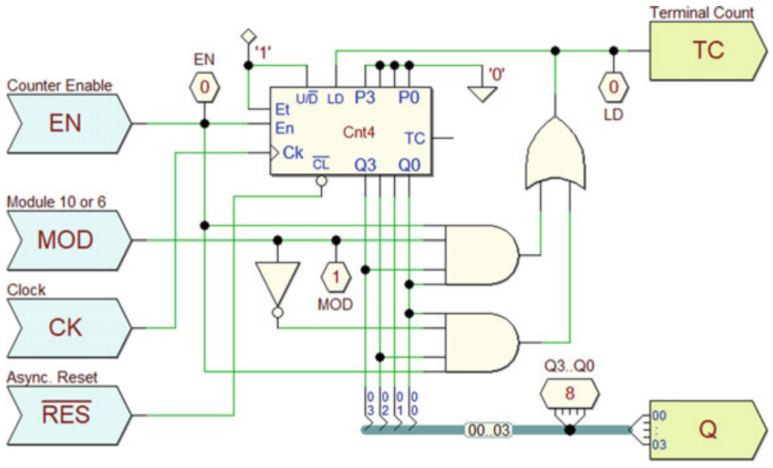
Schematic



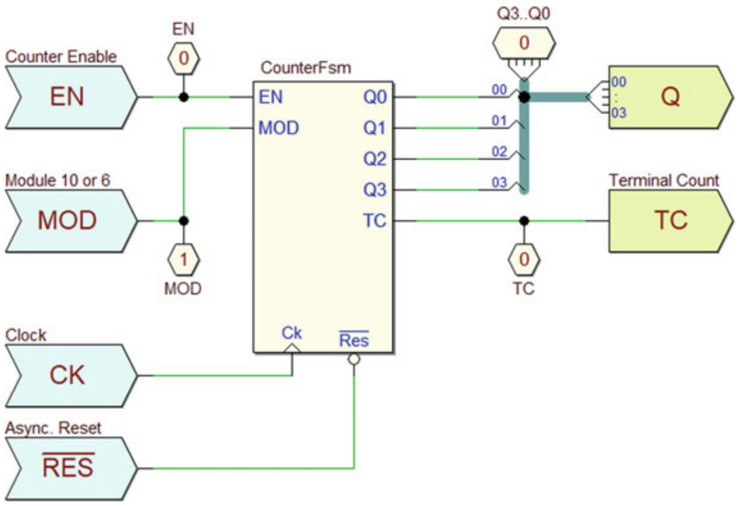
Controller ASM Diagram



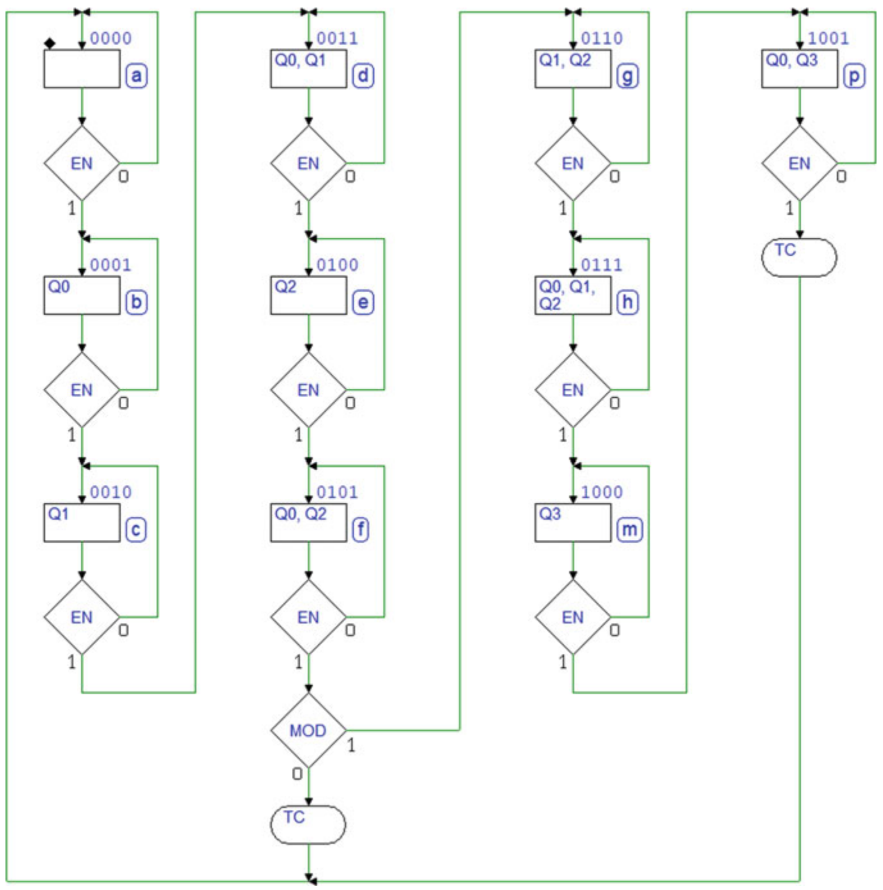
CBE Counter Module (Circuital Approach)



CBE Counter Module (Algorithmic Approach)



ASM Diagram (CBE Counter Module)


0000
a
EN
0
1
0001
Q0
b
0010
Q1
c
0011
Q0, Q1
d
0100
Q2
e
0101
Q0, Q2
f
MOD
TC
0110
Q1, Q2
g
0111
Q0, Q1, Q2
h
1000
Q3
m
1001
Q0, Q3
p
TC