5

Microprocessor systems on FPGA

Abstract In this chapter, we will learn what a Field Programmable Gate Array (FPGA) device is and how we can use it to physically implement what we have just simulated until now. In fact, Deeds supports both the simulation and deployment on FPGA of the microprocessor-based systems. After a brief introduction to the FPGA, we will present a few examples of FPGA-based experimentation boards that we will use to implement our projects. Then we will present several practical project examples (i.e. LED light dimmer, special sound effects, a musical box, a stepper motor controller, and an LCD display-based stopwatch). We will start from the specification, we will continue with the phases of conception, hardware design, and assembly language programming, and we will conclude by showing how quickly the Deeds allows their physical implementation on different FPGA boards.

5.1 Introduction to FPGAs

All the examples in the previous chapters were given with the aim to facilitate understanding of the architecture and programming techniques of microprocessor systems. These are essential capabilities for any designer/programmer but we have not yet focused on their practical implementation. Now, readers who have acquired the basics can work on programming and simulating.

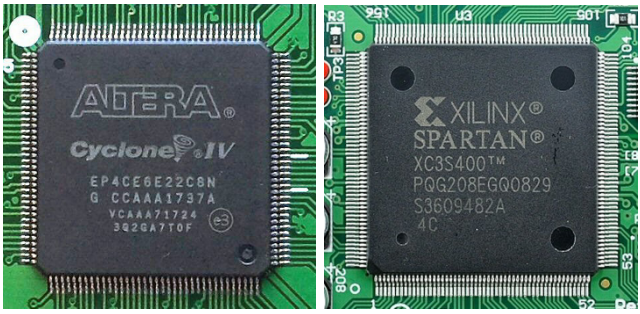
This chapter offers readers a process of experimentation. These examples are inspired by ones from the previous chapters in terms of approach and complexity. The difference here is that readers can work on them and test how they work in practice.

The physical implementation of a system depends greatly on the state of the art of the technologies in use and is subject to rapid changes and consequent obsolescence. A good understanding of the basic concepts from the previous chapters will allow designers/programmers to easily approach the changes coming along the way.

G. Donzellini et al., *Introduction to Microprocessor-Based Systems Design*,
https://doi.org/10.1007/978-3-030-87344-8_5

Right now, designers can take advantage of the programmable components called FPGAs[1], and a wide range of boards for prototypes called "FPGA boards", based on these components. All the practical examples in this chapter are made on FPGA boards.
An FPGA component contains a large number of basic logical elements (logical ports, flip-flops and more complex circuits), that can be connected through a network of connections to create a system. We choose the connections by using specific development software provided by the producers of the FPGA.

Their programming is loaded in the chip and kept in the internal memory. The following figure shows two examples of FPGA devices, produced by *Intel/Altera FPGA*[2], on the left, and by *Xilinx*[3], on the right.



FPGA components are the younger descendants of the the large family of PLDs[4], a term for all the chips that can be specialized for a specific application. Their connections can be internally established when the system is produced or "in the field", i.e., when it is already in use. Since the 1980s, PLD components have profoundly changed the world of complex system design.

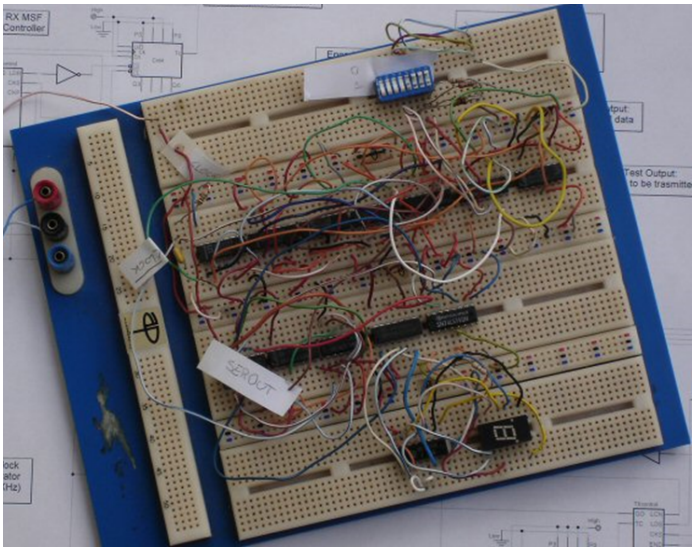
FPGAs are very valuable from an educational perspective as well. They are suited to developing prototypes designed quickly and economically for educational purposes. Finally, an FPGA component should not be confused with a microcomputer, where the hardware is pre-defined and programming consists of setting a sequence of instructions to be executed.

5.1.1 Creation of prototypes with FPGA

Not long ago, "prototyping" (creating a prototype) of a circuit required the connection of a large number of discrete components through soldered wires. This process was very time-consuming and quite prone to connection errors or bad wiring. It was often hard to tell if the system malfunction was due to design error or faulty connections.

[1] Field Programmable Gate Arrays
[2] https://www.intel.com/content/www/us/en/products/details/fpga.html
[3] https://www.xilinx.com/products/silicon-devices/fpga.html
[4] Programmable Logic Devices

In laboratories, it was very common to have boards with solderless connections called "breadboards" with a fixed grill made of internally connected holes. Students used them to insert components and the connections were stabilized by wires. For example, the following figure shows an 8-bit parallel-serial interface implemented on a breadboard. This was done with commercial integrated circuit connections that carry out the various logical functions required (logical ports, flip-flops, registers and counters).



The problems with this prototyping system are similar to those of the traditional soldered systems. Two advantages are the fact that it is easier to change the connections and there is no risk associated with using a soldering iron. A disadvantage is the problem of bad contacts due to wear and tear on the boards or oxidation of the wires.

Currently, solderless breadboards are very useful for quick prototyping of systems based on FPGAs. For example, simple interfaces or support circuits connected to the FPGA board in use can be mounted on a breadboard.

Many types of prototyping boards based on FPGA components are found on the market. They are developed for educational use and run from the simplest and cheapest to more complex versions. They include various types of interfaces in input and output, and allow for the creation of system prototypes, often without needing additional components aside from the board itself.

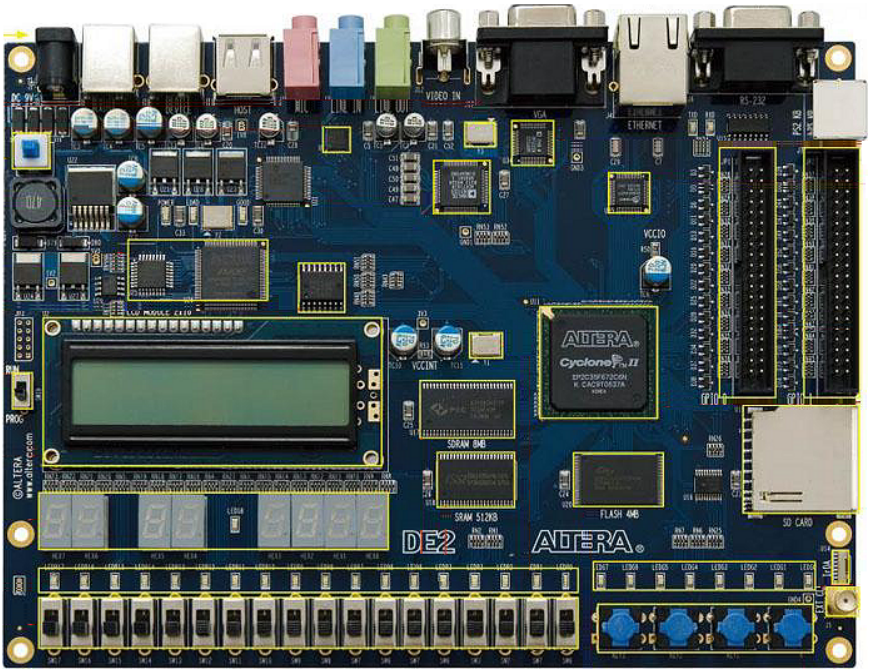
If it has the appropriate capacity, an FPGA component can be programmed so that it implements a microcalculator using the large number of logic elements at its disposal. When a processor is loaded on an FPGA, it is called a "soft-processor", in that its hardware is assembled from the software during the design development. A soft-processor behaves exactly like a "hard-processor", that is, like those physically manufactured on a chip.

5.1.2 Some examples of FPGA boards

Many models of FPGA boards are available commercially and they have constantly evolving features and capabilities. They are made for a broad spectrum of applications: from cheap, simple boards for educational use to fast, complex boards for professional use. Designers and experimenters can find the right board on the market for their applications, in terms of capabilities, software availability and budget.

As an example, we'll give a brief description of some FPGA boards suitable for implementing the microprocessor systems developed in the book. It should be noted that each of these has the capacity to host much more complex systems and could allow for a natural transition to professional design. Some of these boards will be described in more detail in Section 5.4, regarding elements on it that could be put to advantage in conjunction with the Deeds environment. For now, we will only give some general information about them.

The following figure shows the DE2 board, produced by *Terasic*[5] for *Intel/Altera FPGA*[6]. This board is supported by Deeds.



This was conceived for mid/high complexity digital circuit experimentation and includes numerous devices, from the simplest (switches, push-buttons, LED lights, seven-segment displays) to more complex (LCD matrix displays,

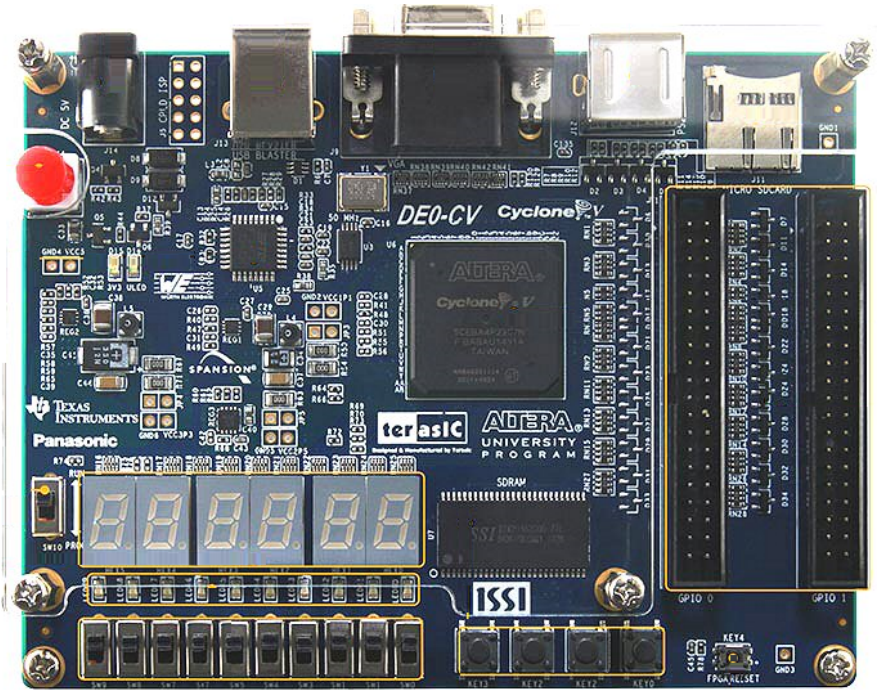
[5] https://www.terasic.com.tw/en/

[6] https://www.intel.com/content/www/us/en/products/details/fpga.html

Ethernet network interfaces, A- and B-type USB 2.0 interfaces, connectors for SD memory cards, analog audio inputs and outputs, analog video input, VGA video output, etc.).

The core of the board is the chip EP2C35 "Cyclone® II" family from *Intel/Altera FPGA*, which has more than 33,000 logic units (the basic logic block will be explained a but further ahead in Section 5.2). The DE2 board allows for the creation of simple, introductory projects (that use a small portion of its potential) to complex systems that can include specialized microcomputers and interfaces.

The following figure shows the DE0-CV board produced by *Terasic* for *Intel/Altera FPGA*. As the image shows, it has push-buttons, switches, LED lights, seven-segment displays, connectors and other devices, although fewer than the previous board had. This board is also supported by Deeds.

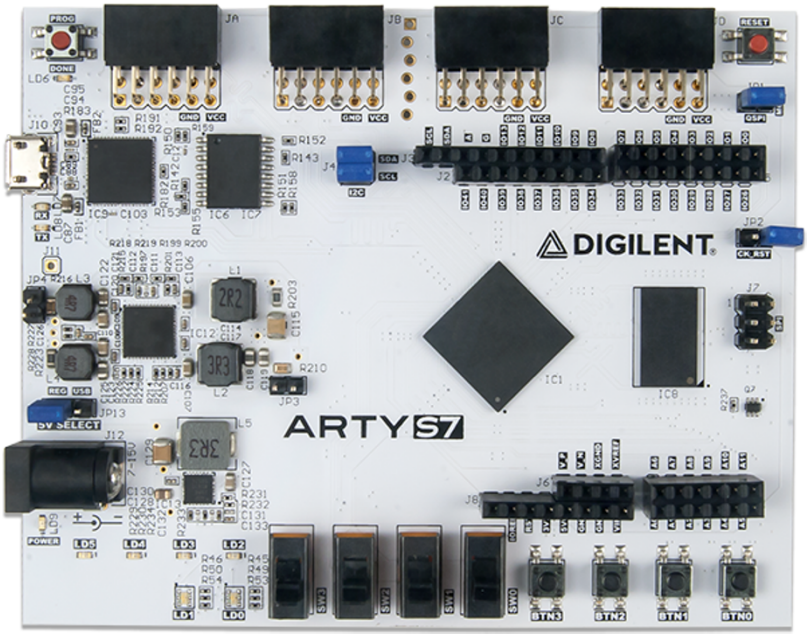


The core of the board is the big, black square in the center. This is the FPGA chip 5CEBA4, part of the "Cyclone® V" family, produced by *Intel/Altera FPGA*. This chip contains a matrix of about 49,000 logic units and 3,080 Kbits of integrated RAM.

Other, larger members of the same chip family include the *ARM Cortex*^{TM} dual-core microprocessor by ARM (Advanced RISC Machine)[7], which is used in many mobile phones.

[7] `https://www.arm.com/products/silicon-ip-cpu`

The following figure shows another example: the ARTY S7-50 board produced by *Digilent*[8], which uses the XC7S50 chip from the "Spartan®-7" family, by *Xilinx*[9]. The FPGA contains over 52,000 basic combinational blocks and over 65,000 flip-flops. The FPGA chip is the one positioned at a 45 degree angle on the board[10].



As in the previous example, this board has push-buttons, switches, LED lights and other interfaces. More specifically, we have connectors designed to host input and output boards ("shields"), which were originally designed for *Arduino* microcontroller boards[11].

The last example we will show on these pages is a very inexpensive FPGA board available online and supported by Deeds (see the following figure). It is based on a "Cyclone® II EP2C5T144C8" chip from *Intel/Altera FPGA*.

The user has four connectors positioned around the chip, only three LED lights and one push-button. The two 10-pin connectors on the left, however, are used to program the FPGA chip. For brevity's sake, we will refer to this board as "EP2C5".

To implement our designs, which often need multiple input and output devices, we need to use the four connectors to externally connect all the necessary push-buttons, switches, LED lights and displays.

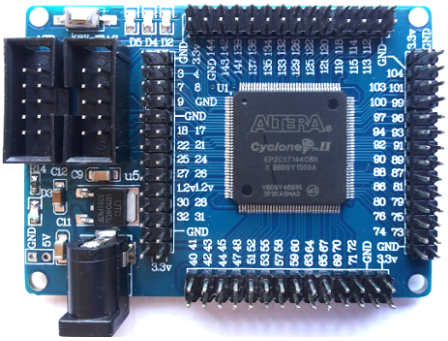
---

8 `https://store.digilentinc.com`

9 `https://www.xilinx.com/products/silicon-devices/fpga.html`

10 This board is not yet supported by the Deeds environment.

11 `https://www.arduino.cc/`



The chip is big enough to implement a microcomputer based on the Deeds DMC8 processor if it is configured with a small RAM and ROM (once the DMC8 is loaded on the FPGA component, it uses about half of the 4,600 logic units available and only 400 of the 4,600 flip-flops).

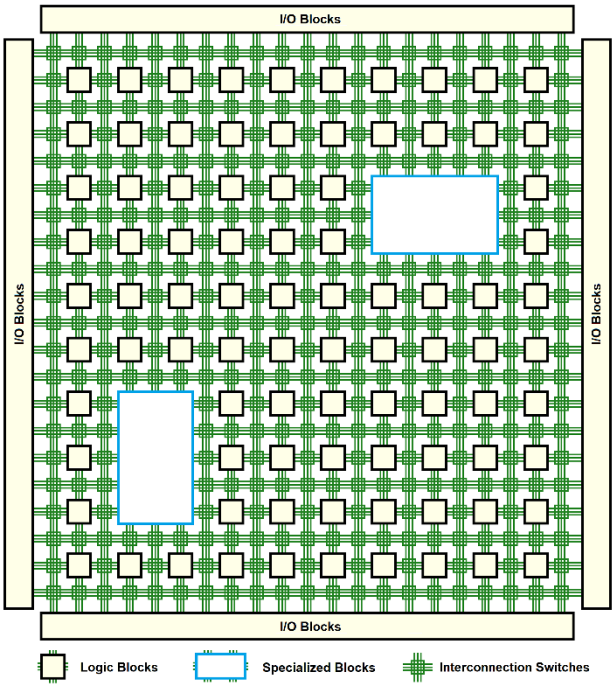
5.2 The architecture of FPGA components

FPGA makers offer a wide array of devices classified into "component families", that differ in their complexity and their fields of application.

Some typical examples are audio/video signal processing, radar, automobile systems and generally, all the applications that require high performance but don't have the volume to justify the cost of a "full custom" chip.

Despite the wide variety, all the devices have the same base architectural structure in common.

An FPGA chip is essentially a big matrix of logic blocks set in rows and columns as shown in the figure at the right.

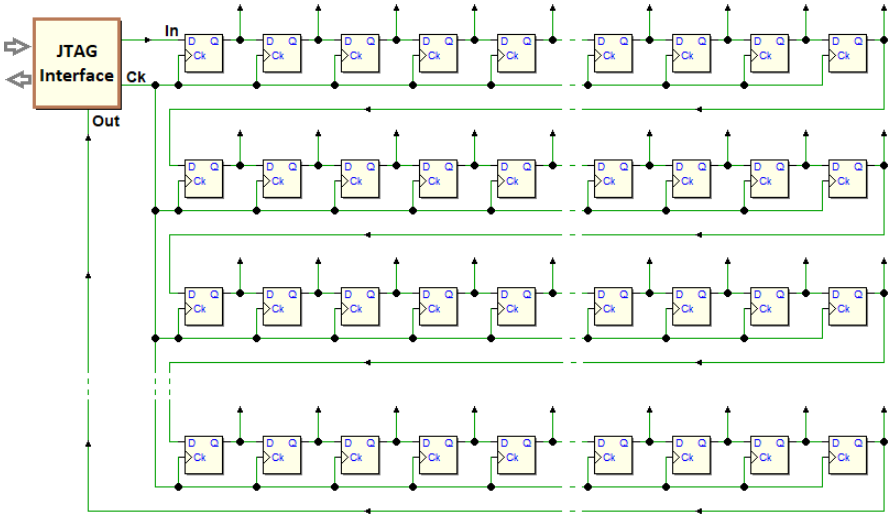


Every block contains one or more flip-flops and programmable combinational networks. A matrix of connections goes across the whole chip, using up most of its area.

At every intersection of the matrix, programmable electronic switches allow for the local connection between rows and columns, and the interconnections with the blocks. Local sub-matrices of connections can be made available to improve the speed of communication between nearby blocks.

There can also be special blocks inside the matrix intended for specific functions like read/write RAM memory, arithmetic circuits (often multipliers), etc. The matrix is also surrounded along its four edges, by input and output logic blocks responsible for the interface of the chip and external devices.

The previous figure shows only the elements of the chip that are available for the design. It does not show the memory elements that program the connections and configure the logic blocks. There is a high number of flip-flops connected in cascade, that make up a very long shift register as shown in the following figure.



A specific serial interface (called JTAG, which will be explained later) is tasked with writing the flip-flops in the programming phase of the FPGA chip (in the normal functioning, however, these flip-flops are not accessible, they cannot be used as part of our projects).

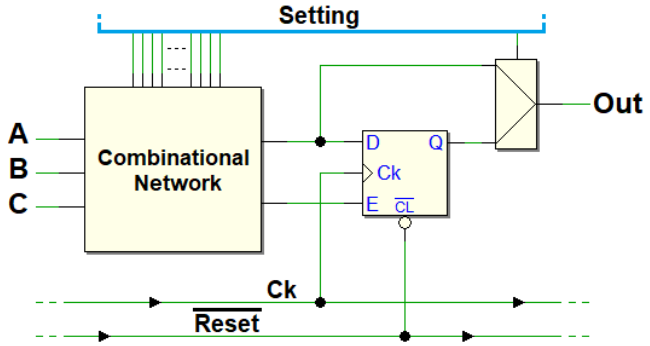
The FPGA component must be re-programmed every time the system is turned on[12]. To solve this problem, we add to the circuit board a Flash ROM memory chip (see Appendix A.1) that stores the programming information.

At every system power up, the Flash ROM will transfer the programming information into the FPGA, through dedicated pins.

[12] As we know, a flip-flop does not store information when the power is off.

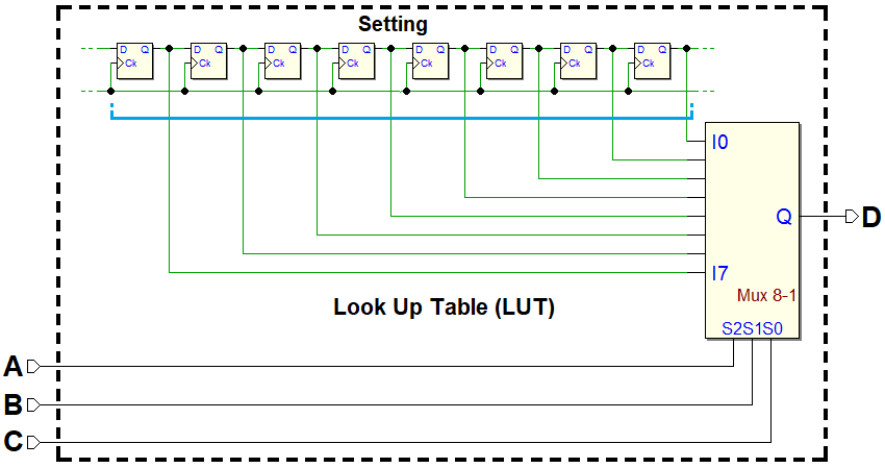
5.2.1 Logic blocks

The following figure shows the basic schematic of one generic "logic block" in an FPGA. The block has a type-E flip-flop (edge-triggered), driven by a combinational logic network.



The combinational network's functionality is controlled by the configuration shift register ("Setting", in the upper part of the figure), and also by the 2-to-1 multiplexer on the right, which gives us the option to use the flip-flop to register the output of the combinational network.

It might be interesting to more closely examine the combinational network, that is implemented in the form of an LUT (Look-Up Table). Its values are selected by a multiplexer driven by the network inputs (see the figure below).



During the FPGA chip programming operations, the values of the desired table are stored in the flip-flops of the above-mentioned shift register. Once they are stored, these values remain the same during the normal operations of the FPGA, and the multiplexer copies them on output D, according to the combination of A, B and C, giving us the requested Boolean function.

Notice that according to the FPGA family, the logic block might be more complex than what is presented here. For example, it might also contain an adder, XOR networks, other flip-flops and multiple combinational networks.

5.2.2 JTAG programming

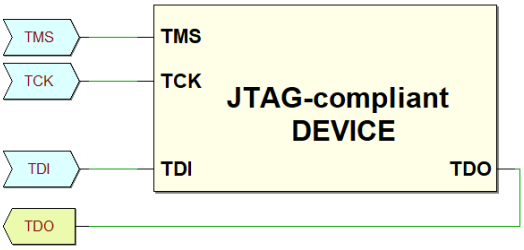
JTAG is the acronym for the (Joint Test Action Group) consortium that defined a standard protocol for the functional testing of integrated circuits toward the end of the 1980s. It was later published as IEEE 1149.1 ("IEEE Standard Test Access Port and Boundary-Scan Architecture"). In the following, this will be referred to as JTAG (the term "Boundary-Scan" is also used)[13].

The version of this protocol released in 1994 added the capacity to program memory, microcontrollers and other devices. It also made it possible to execute functional debugging of the firmware and activate automatic tests ("Built-In Self-Test"), defined by the maker of the component. To achieve this, a standard language ("Boundary Scan Description Language") was developed to access the components using the JTAG interface.

Currently, the JTAG interface is the only method used to access the internal hardware of electronic systems such as mobile phones, tablets, wireless "access points" etc. both for testing during production and for fault diagnostics.

Let's examine its basic operations. Briefly, the standard offers the possibility to stop the normal system operations and going to a modality where the JTAG interface is activated. The interface takes control of all the external pins of the components and the test and programming circuits in the system itself.

The physical JTAG interface is composed of a limited number of standard connections. The simplest set allows us to communicate with the circuit by using the TCK, TMS, TDI and TDO lines as shown in the following figure.



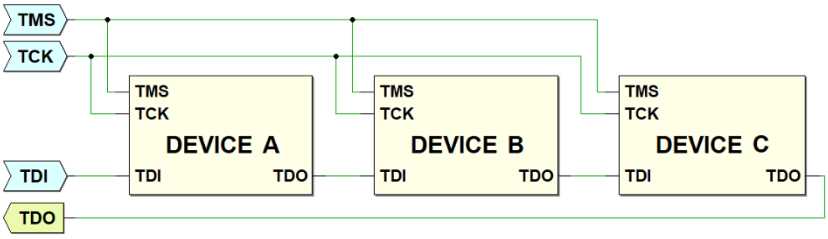
Pin	Name	Function
TCK	*Test Clock*	Data Clock Pin
TMS	*Test Mode Select*	Mode Control and Operation Selection
TDI	*Test Data In*	Serial Data Input Pin (toward the device)
TDO	*Test Data Out*	Serial Data Output Pin (from the device)

[13] `https://www.jtag.com`

All the signals of the JTAG interface are serial and synchronized by the clock TCK (usually in the $10 \,..\, 100\, MHz$ interval). Activating TMS signals the system to turn on the JTAG-compliant mode. So, through the same line, we can execute the required operation by using a specific state algorithm, the "JTAG State Machine" (not described here).

The standard also defines an optional control line, Test Reset (TRST), but its functionality can be obtained by controlling TMS following a specific sequence and it is often not used as in the example above.

The standard furthermore defines the chain connection of the TDI and TDO pins of more than one device so that we can access all the JTAG-compliant devices located on the same board (see the following figure).



This method makes it possible to run a "chain integrity test", for example. Each JTAG-compliant device has its own ID code. All the ID codes can be read and checked against the design project ID to see if the JTAG chain works as it should.

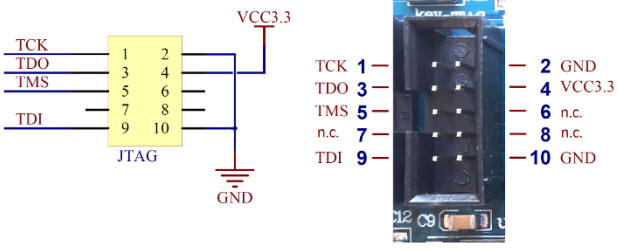
5.2.3 Devices for programming FPGAs

Many programmable devices like FPGAs use JTAG for programming, and not only for testing purposes.

It should be noted that FPGA chips are programmable after they have been soldered on the board. This brings many advantages including simplifying the programming phase, avoiding the use of external programmers and adding the possibility of updating and changing the networks programmed inside the chip. This all makes FPGA systems ideal for the implementation of prototypes and experimental circuits, including those for educational use.

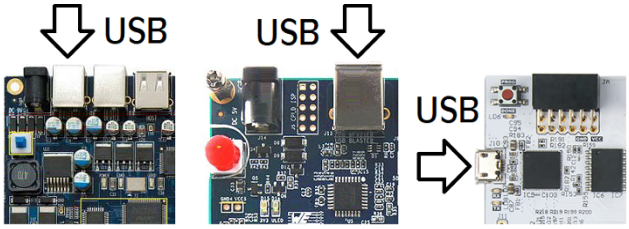
There are different standards for the physical JTAG interface.

The simplest uses a 10-pin connector, as shown in the figure on the right, which refers to an EP2C5 board.



In the following figure, we see a JTAG programmer and its 10-pin cable (on the right) as well as a standard USB cable to connect it to a PC. The software is provided by the maker of the FPGA.

Often, especially in more complete boards, the programmer is integrated and available through a dedicated USB interface. This is the case for the first three boards described before.



5.3 FPGA development tools

The makers of FPGAs sell proprietary development tools for professional designers to implement complex systems. The makers provide reduced versions of the same tools usually for free, to educators.

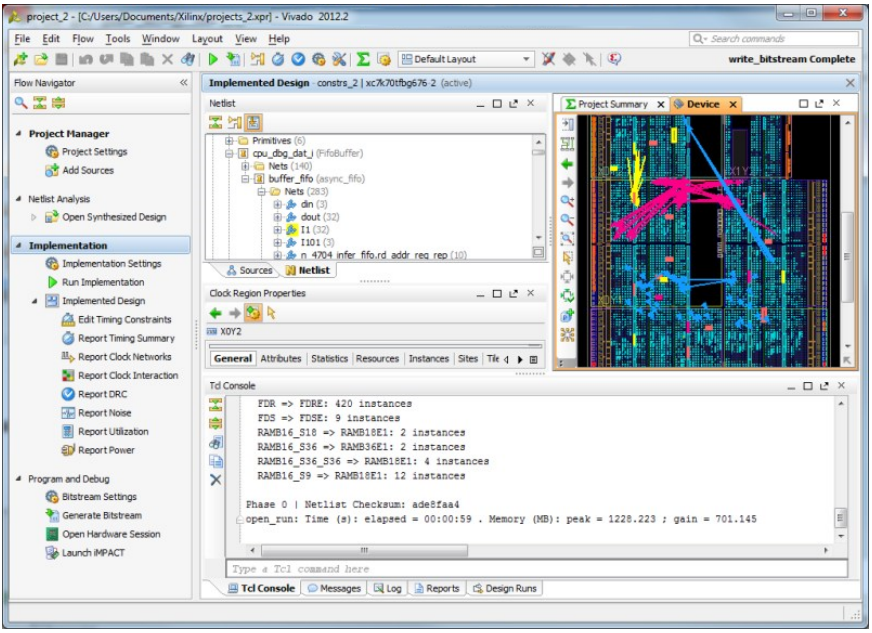
Clearly, the goal is to promote their products to future designers by influencing their choices later on in the world of work. This is why makers provide a great deal of documentation on their software.

In the following, we present a brief panorama of what is available for free online. All the tools mentioned offer similar functional capabilities, such as schematic editors, source code editors, compilers, pin planners and optimization tools. They make it possible to design digital systems by using logical schematics or languages to describe the hardware (HDL) such as the Verilog[14], the VHDL[15], or the System-C[16].

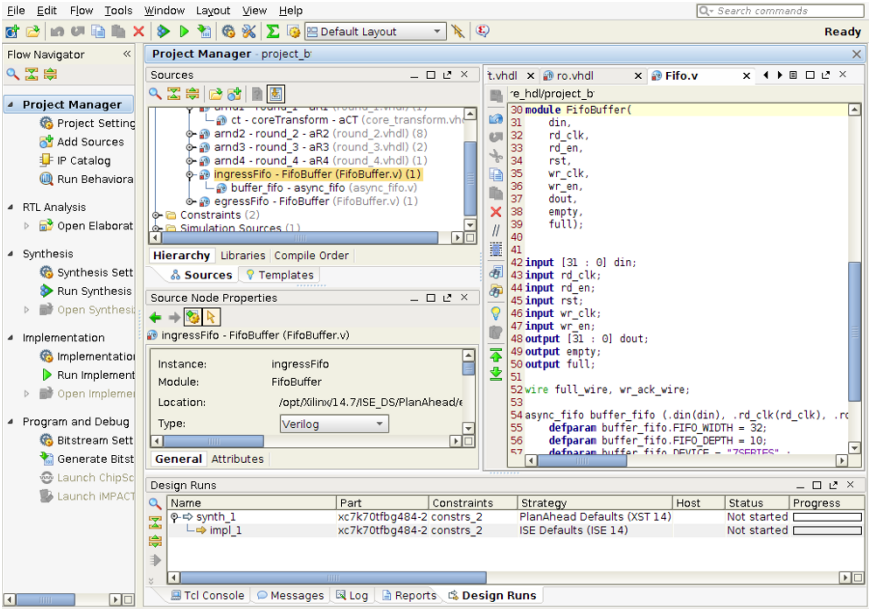
The large number of functions available that make a professional's work more productive, may give a beginner the impression of a complex arena that is difficult to manage. In the following, we will see that Deeds allows for the use of FPGAs for rapid prototyping of our projects without entering into the typical technicalities of professional tools.

[14] https://standards.ieee.org/standard/1800-2017.html
[15] https://standards.ieee.org/standard/1076_6-2004.html
[16] https://accellera.org/community/systemc

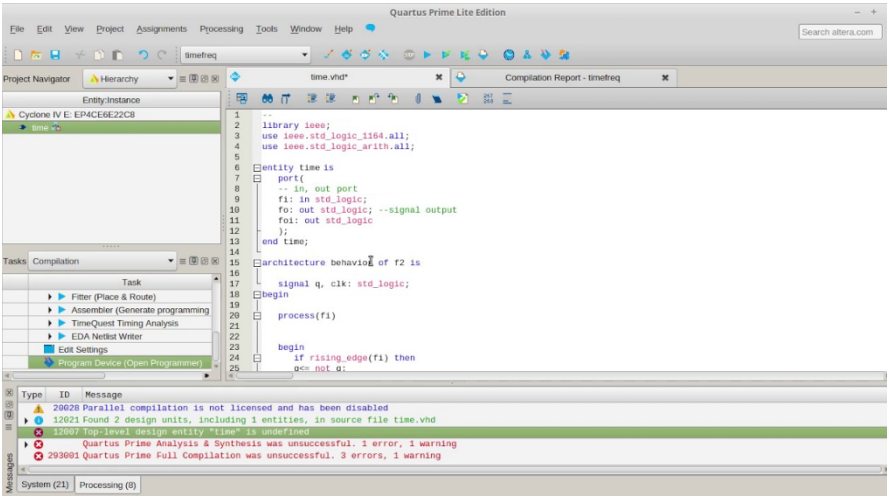
Regarding boards based on *Xilinx* FPGA devices, at the moment this book is being written, there are free tools available such as: *Vivado® Design Suite HL WebPACK™* and *ISE® WebPACK™*. The following screen-shot shows *Vivado®*, which is made for the most recent families of devices.



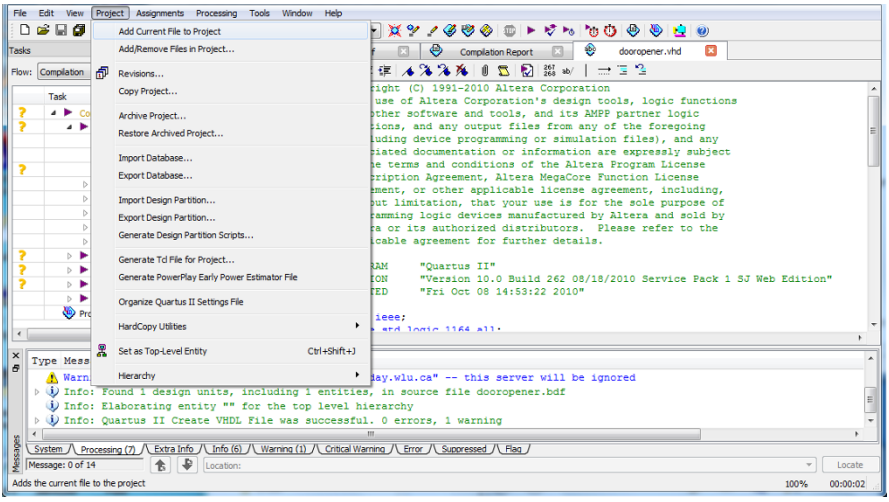
Below, the *ISE®*, which supports the less recent FPGA families.



Intel/Altera FPGA offers two free tools: *Quartus® Prime Lite Edition™* and *Quartus® II Web Edition™*. The following screenshot shows the main window of the first. It supports the most recent families of FPGAs.



The previous versions of this software are called *Quartus® II*. The main command window for project management is found below.

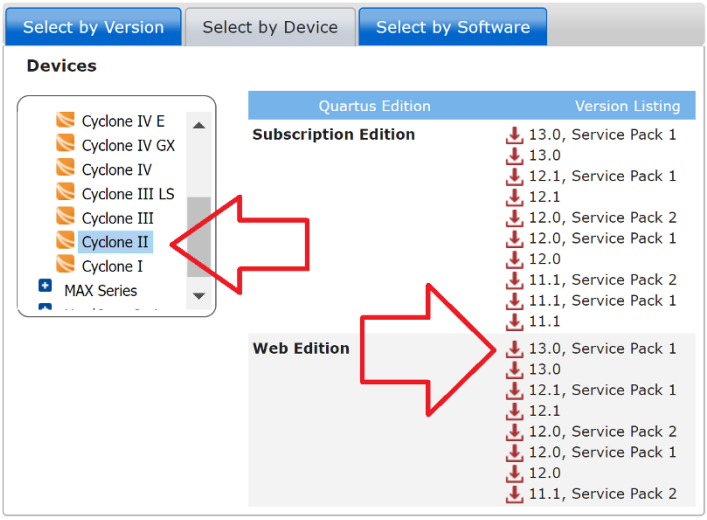


In recent years, due to the increase in the number of families and the complexity of chips FPGA makers have put greater effort into developing and maintaining newer generation tools rather than guaranteeing they would be compatible with older ones.

In general, less recent FPGA boards need to be used with older versions of software. Therefore, it is important to study the documentation on a producer's website before choosing chips and tools.

A useful function (*Software Selector*) associates an FPGA family to the corresponding software to use.

The screenshot below, from the *Intel/Altera FPGA* website shows that the *Cyclone® II* family is supported by version "*13.0 - ServicePack 1*" and previous versions from *Quartus® II Web Edition™*.



5.4 The FPGA boards used in the examples

5.4.1 The DE2 board

As introduced before, in Section 5.1.2, the Terasic/Altera board DE2[17] makes it possible to do simple, basic projects (using few of the boards resources), as well as systems including a microcomputer and its interfaces.

The DE2 board has various types of components[18]. Those interested in learning more about what it can do should consult the exhaustive documentation on the manufacturer's website[19], which also contains the user manual. In our examples, we use those components of the board that can be directly interfaced with the projects generated by Deeds.

For example, there are 18 slide switches (Sw17 .. Sw0) connected to 18 pins of the FPGA chip (see the following figure).

[17] http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=39&No=30

[18] https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=39&No=30&PartNo=2#section

[19] https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=39&No=30&PartNo=4#section

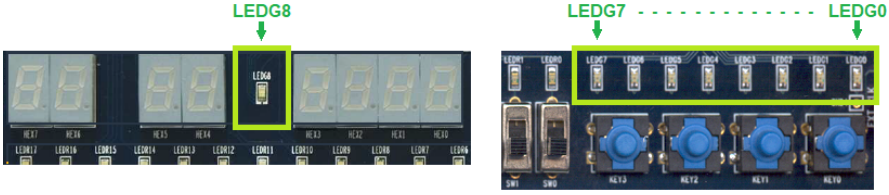
The manufacturer defined the switch connection so as to provide a '0' to the FPGA pins when their cursors are brought low, toward the edge of the board and a '1' when they are high.

On the bottom right we see 4 push-buttons (Key3 .. Key0), which are also connected to the FPGA input pins (see the following figure).

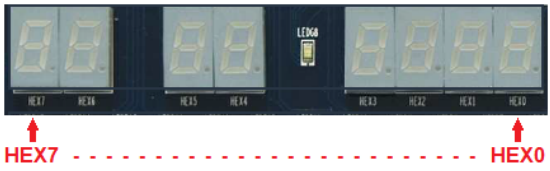
The push-button connections are set so as to generate a '0' on the corresponding FPGA chip pin when the push-button is pressed, and a '1' when it is in idle position.

Directly above the switches, there are 18 red LED lights (LEDR17 .. LEDR0), as indicated in the following figure.

There are also 9 green LED lights (see the following figure), 8 of which are placed on the push-buttons (LEDG7 .. LEDG0), and one (LEDG8) positioned near the seven-segment display.

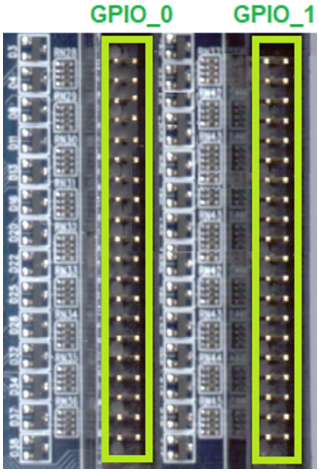


The eight seven-segment displays (HEX7 .. HEX0, see the following figure), can manage the individual segments by driving them directly (the native modality of the board).



Deeds allows for the use of the "decoded" displays in the library.

These components make it possible to use the board display by providing the 4-bit number in binary to show as a hexadecimal number.



The board also has two 40-pin "expansion connectors" (GPIO_0 e GPIO_1, see the figure on the right). The connector pins are directly connected to the FPGA chip and can be programmed individually as inputs or as outputs. See the above-mentioned user manual for more information on connectors' pinouts (a few power supply lines are also available on them).

Deeds allows the user to connect the connector pins freely to external networks, as we will see later in this chapter.

There is a special issue about clock generators on the board, where we find two crystal oscillators, one at 27 MHz, and the other at 50 MHz.

Deeds supports both generators. From the second one, it derives (by division) our projects' clockwork frequencies. Therefore, we can define clock frequencies of $50MHz$, $27MHz$, $10MHz$, $5MHz$, $2MHz$, $1MHz$, $500KHz$, $200KHz$, $100KHz$, etc. down to the lowest, $1Hz$).

5.4.2 The DE0-CV board

In this chapter, we will also use the Terasic/Altera DE0-CV board[20]. As mentioned in Section 5.1.2, this board makes it possible to develop projects of varying levels of complexity, similar to what we saw for the DE2 board. It has an FPGA chip of the newest generation "Cyclone® V" (which is more powerful) but has fewer switches, push-buttons, LED lights and displays.

The DE0-CV board also has a wide variety of components[21]. The manufacturer's website has exhaustive documentation[22]. The user manual is a good starting point to learn more about its components.

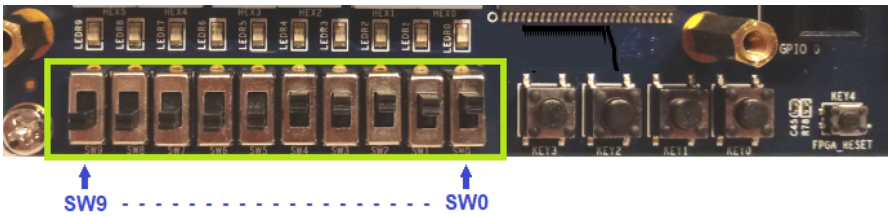
Now, let's look at components of the board that will allow us to interact with our networks. The figure below shows 10 slide switches (Sw9 .. Sw0), connected to 10 FPGA inputs.

[20] https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=167&No=921

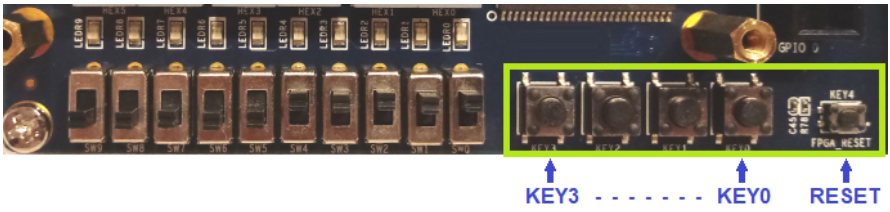
[21] https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=167&No=921&PartNo=2

[22] https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=167&No=921&PartNo=4

They are connected in order to provide the chip pins with a '0' when the cursor is low, toward the edge of the board and a '1' when we move it high.



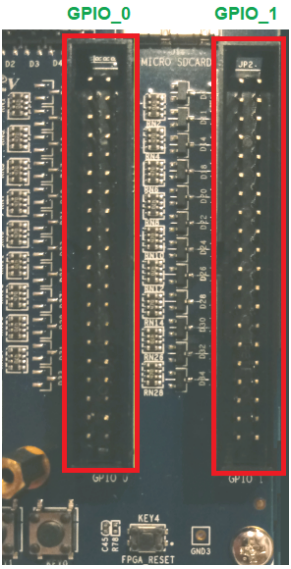
At the lower right hand side of the following figure, there are 5 push-buttons (Key3 .. Key0 e RESET), which are also connected to the input pins of the FPGA. The push-buttons generate a '0' on the corresponding pin of the FPGA chip when the push-button is pressed, and a '1' when it is not.



There are 10 red LED lights (LEDR9 .. LEDR0) in line above the switches, as shown in the figure below. There are no green LED lights on this board.

As shown in the following figure, there are six seven-segment displays. We can manage their individual segments. Deeds allows for the use of the "decoded" displays here as well (These are the same as those belonging to the DE2 board).

The board also has two 40-pin "expansion connectors" (GPIO_0 e GPIO_1, see the following figure), which are identical to those in the DE2 board.



The connectors, here as well, are directly connected to the FPGA pins and can be defined as inputs or outputs. We can also connect the connector pins to external networks.

See the board user manual for more information on connectors' pinouts (a few power supply lines are also available on them).

We see only one $50MHz$ quartz clock generator on this board. With the support of Deeds, we can use this to obtain the clock work frequencies of our projects ($50MHz$, $10MHz$, $5MHz$, $2MHz$, $1MHz$, $500KHz$, $200KHz$, $100KHz$, etc. down to $1Hz$).

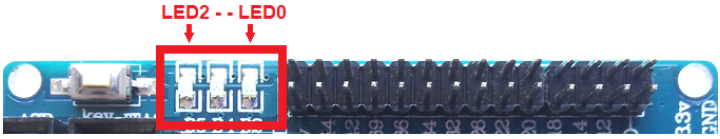
5.4.3 The EP2C5 board

The third board used in the examples of this book is called the "EP2C5". It is an economical, unbranded FPGA board that is easy to find online. It is based on the "Cyclone® II EP2C5T144C8" chip from *Intel/Altera FPGA*. As mentioned in Section 5.1.2, this board allows us to create networks that are smaller in size. Yet, we are still able to load a DMC8 microcomputer on it, if configured with a small sized ROM and RAM system.

Documentation for this board, including the electrical schematics can be found online[23]. It has no switches and offers the user only one push-button. There are only three red LED lights and no display. Essentially, any interface component that we would need would have to be connected to the board.

The one available push-button (KEY0) is shown in the following figure.

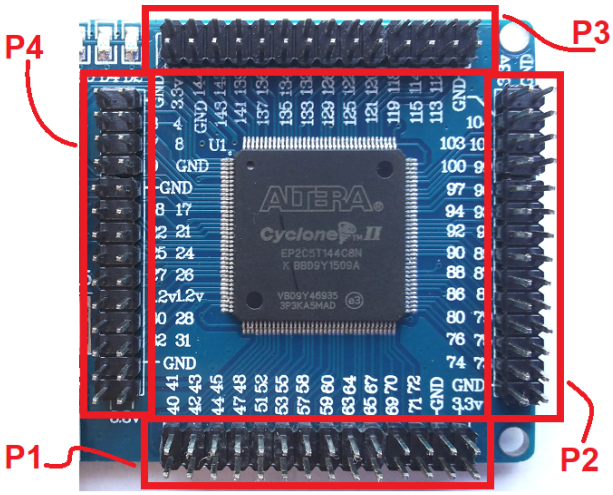
On the same side of the board, we find the three LED lights (LED2..LED0).



[23] http://land-boards.com/blwiki/index.php?title=Cyclone_II_EP2C5_Mini_Dev_Board

The board has four 28-pin connectors (P4, P3, P2 and P1, see the figure on the right) connected directly to the FPGA chip pins.

They can be set as inputs or outputs and given the scarcity of devices on the board, they'll necessarily be used to connect to external interfaces, as shown in Section 5.5.



The EP2C5 board has a 50*MHz* clock generator, like the boards we have studied before. For our projects, Deeds offers the use of the native clock frequency and many of its submultiples (those listed for the DE0-CV board).

5.5 Microprocessor system prototypes on FPGA

In this section, we will show how to implement a microprocessor system on FPGA that is designed through the Deeds software suite. In the examples, we will use the 3 FPGA boards described in Sections 5.1.2 and 5.4). In any case, the concepts presented here can be reused on every FPGA board supported by Deeds to create not only microprocessor systems but also any digital network.

5.5.1 The steps to take

To put a project created with Deeds on an FPGA board, we need to follow the procedure below, which is applicable for all the boards that Deeds supports. After simulating our system to check that it works correctly, we take the following steps:

— Associate the input and output devices on the Deeds schematic with those available on the chosen FPGA board,
— Launch the automatic conversion of the Deeds project in VHDL code,
— Program the FPGA with the *Quartus® II Web Edition™* software.

In the following, we will go through this procedure step by step to provide a practical example to refer to. First, we will present a microprocessor system to implement on FPGA. Then, we will provide an example for each board presented in Sections 5.1.2 and 5.4.

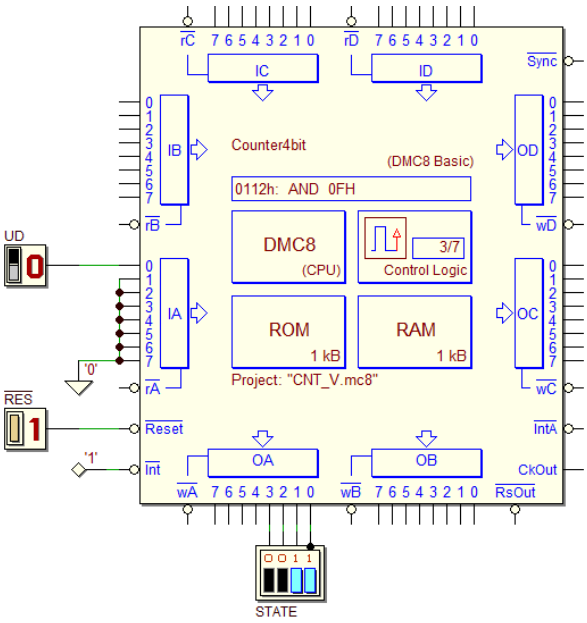
If none of these boards is available, we suggest following this procedure regardless since it is usable in all the boards supported by Deeds. By studying the DE2 and DE0-CV boards, we will learn to use the I/O devices already on the board, while studying the EP2C5 will show us how to connect input and output devices that are not on the board through the board's connectors.

5.5.2 A system to implement on FPGA: An example

To facilitate learning, we will begin with a simple example: emulating a 4-bit bidirectional binary counter.

The image on the right is a schematic that includes a "DMC8 microcomputer" component (in its basic version, introduced in Section 2.4.1), configured with 1 kB of ROM and 1 kB of RAM.

An input checks the direction of the count (UD), and a push-button is connected to the reset input $\overline{\text{RES}}$.



A bar of 4 LED lights (STATE) is connected to output port OA. The interrupt line is set at '1' since it is not used.

When UD is at '0', the count goes up; when it is at '1', it goes down. Notice the input port IA wiring, where we connected[24] the unused lines at '0'.

Now, let's look at the program loaded in the microcomputer. First, we see the label definition for input port CNTRL, where input UD is read from. Next, we see the label definition for output port STATE, where the internal state of the counter is displayed. Then we find the usual link to the reset.

```
CNTRL   EQU   00h      ; IA input port (UD control line, bit 0)
STATE   EQU   00h      ; OA output port (the counter output)
        ORG   0000h    ; link to the reset
        JP    0100h
        ORG   0100h
```

[24] If we don't connect the unused lines to '0' (or '1'), Deeds signals the reading of unknown values during the simulation.

Before entering the program's main loop, we zero output port STATE and register B, which contains the internal state of the counter. We do not use the Stack in the code, so we do not initialize the Stack Pointer.

```
INIT:   LD    A,00h        ; initialize the counter state
        OUT   (STATE),A    ; and the outputs to zero
        LD    B,A
```

Every time the main loop is executed, the UD input is first checked (by reading the input port CNTRL), so register B is then incremented or decremented.

```
MAIN:   IN    A,(CNTRL)    ; read the input port CNTRL
        BIT   0,A          ; check the value of input line UD
        JP    NZ,DN        ; jump and decrement the state if it is at '1'
UP:     INC   B            ; otherwise, increment the state
        JP    UPDATE       ; jump to update and display it
DN:     DEC   B            ; decrement the state
```

Then bits 7, 6, 5 and 4 are set to zero through bit masking to reduce the count to 4 bits. Finally, the new value of the count is shown on output port STATE.

```
UPDATE: LD    A,B          ; mask the bits in position 7,6,5 and 4
        AND   0Fh          ; to reduce the count to 4 bits
        LD    B,A          ; update the state in B
        OUT   (STATE),A    ; and copy it to the output lines
        JP    MAIN
```

Note that bit masking is not strictly necessary since the bits connected to the bar of LED lights are only those in positions 3, 2, 1 and 0, but it is done anyway to make the code more legible.

5.5.3 Implementing the network on an FPGA board

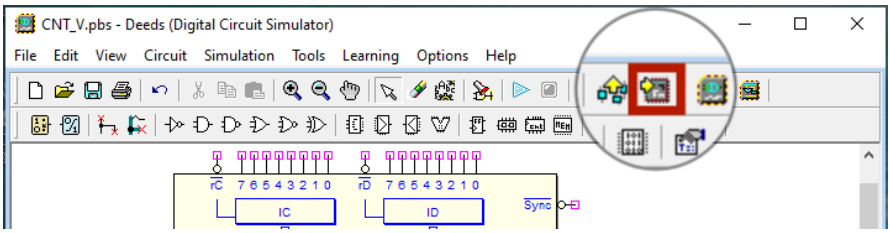
Now let's look at the procedure required for implementing the system above on an FPGA board. The following explanation goes more into detail on DE2, DE0-CV and EP2C5 boards[25].

Associating input and output devices

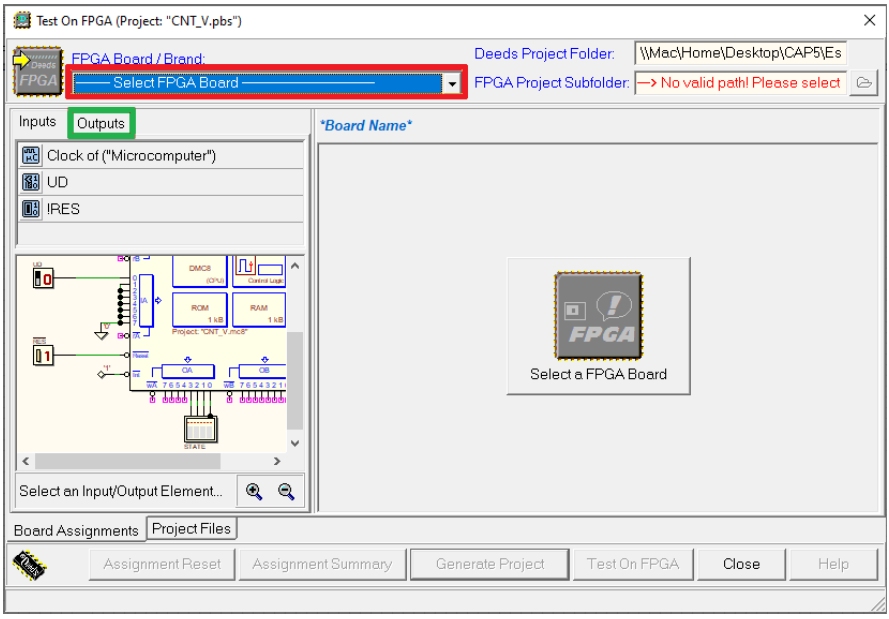
Let's study how to associate the input and output devices in the Deeds schematic with those available on the FPGA board.

The following figure shows the Deeds-DcS main page with our project open in the editor. We press the "Test on FPGA" button on the main tool bar (highlighted by the red box on the upper right hand side).

[25] There are extensive tutorials available about this procedure on the Deeds website.



A new window, shown in the following figure, will open. This allows us to associate the push-buttons, switches, clock generators, LED lights, etc. in our schematic with their respective components on the FPGA board. By using expansion connectors, we can connect other devices. In some boards, such as the EP2C5, this operation is necessary since the components available to the designer are not enough to create our system.



In the box at the left, we can see the input devices in the schematic. By clicking on "Outputs" (in the green box) we can see the output devices. When we click on the upper left hand side menu (in the red box), we can select the FPGA board where we want to implement the project defined in the schematic.

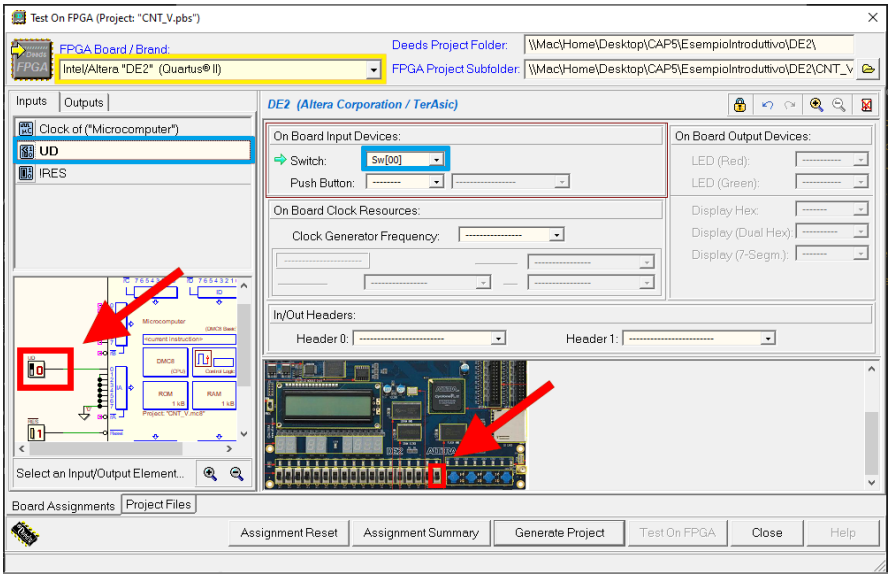
When we select an FPGA board, its inputs (or outputs, depending on the page selected) will be made available to the designer. When we click on one of the schematic's inputs (or outputs), the inputs (or outputs) of the board that we can associate with it will be shown.

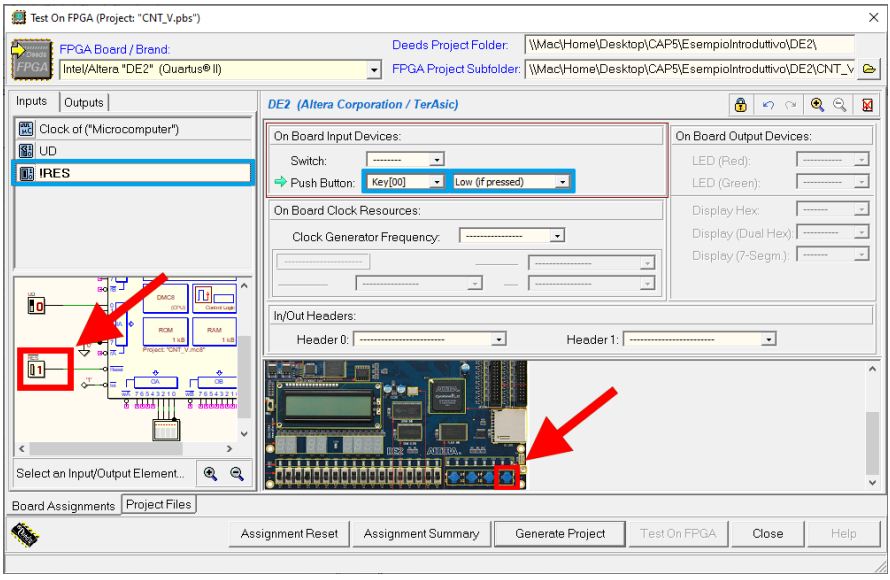
The following pages will show the associations chosen for each board.

5.5.4 Settings for the DE2 board

Associating input devices

When we select the DE2 board, the window below will appear. We have clicked on input UD in order to associate it with one of the switches on the DE2. Here, we have chosen the SW[00] switch, as shown in the blue rectangle.

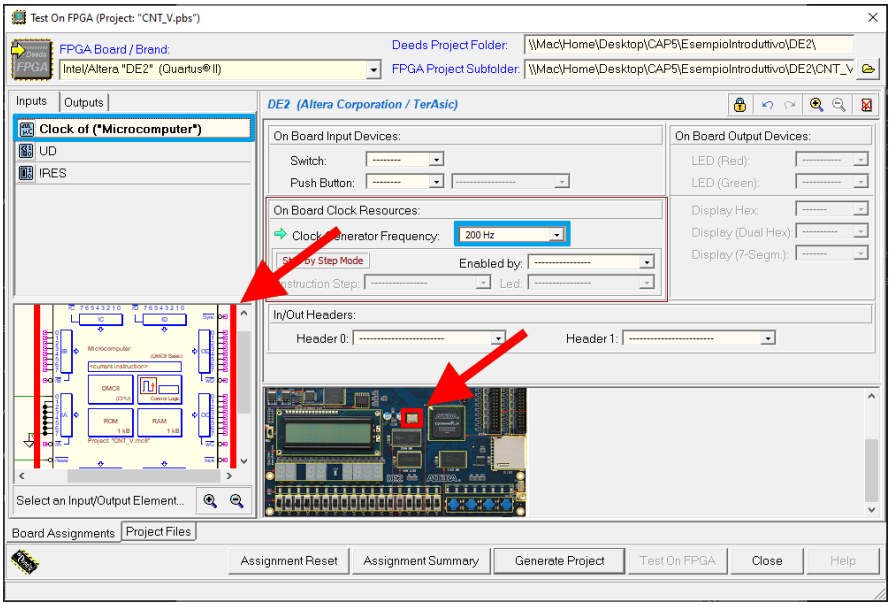




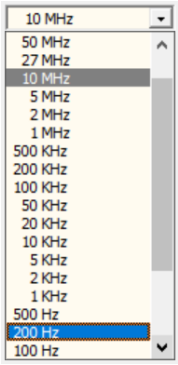
This window shows both the position of the component selected in the Deeds schematic and that of the one associated to it on the FPGA board, on the right. Both are highlighted in red (see the arrows).

The same procedure was followed for input $\overline{\text{RES}}$, which was associated to push-button KEY[00] and set to generate a low level when pressed (see the figure at the bottom of the opposite page).

We have chosen the DMC8 Microcomputer component in order to set its clock frequency (see the following figure).



Although the clock is fixed at 10 *MHz* in the simulation, when creating it on FPGA, we can select a lower or higher frequency clock source. See the figure at the right. However, if we choose a frequency that is different from 10 *MHz*, we need to consider the fact that the calculations for any delay loops will have to be redone for the new frequency.

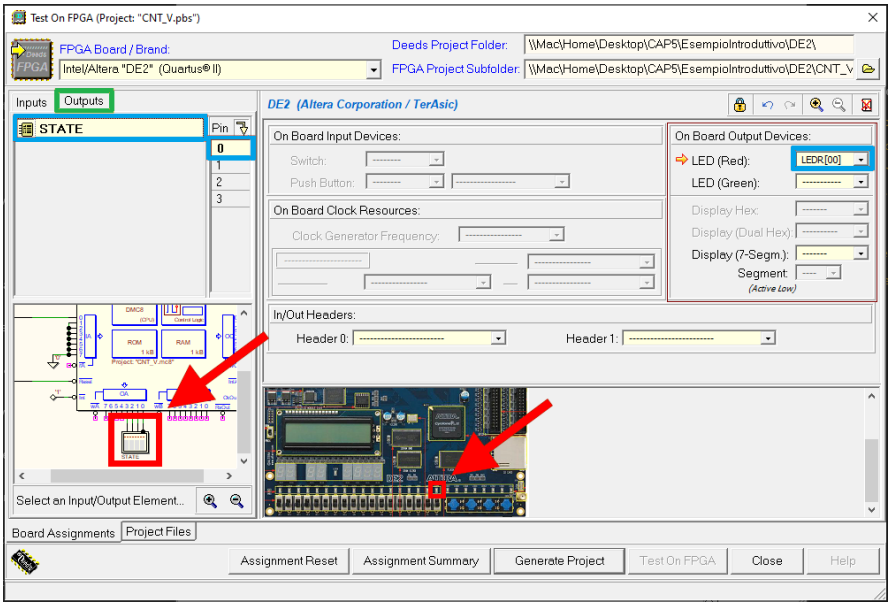


As shown in the blue rectangle in the center of the figure above, a 200 *Hz* clock was chosen in order to make human interaction with the device possible. There are other options for function checks on the clock for debugging, but they have been omitted here for simplicity's sake.

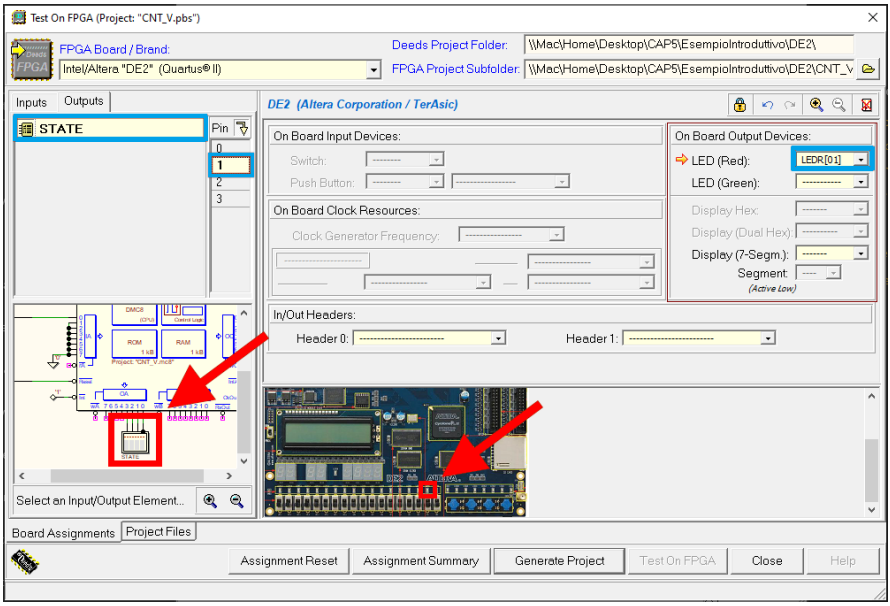
All the clock frequencies we can choose in the box ensure the proper operations of the microcomputer. As highlighted by the red arrows in the figure above, the clock generator on the board corresponds to the selected DMC8 component. This association is shown by the red outlines.

Associating output devices

Once the input devices are defined, we go on to associating the outputs by clicking on the "Outputs" palette outlined in green in the figure.



We select the STATE component (outlined in blue), then we associate the red LED lights available on the FPGA board to its four pins (one by one).

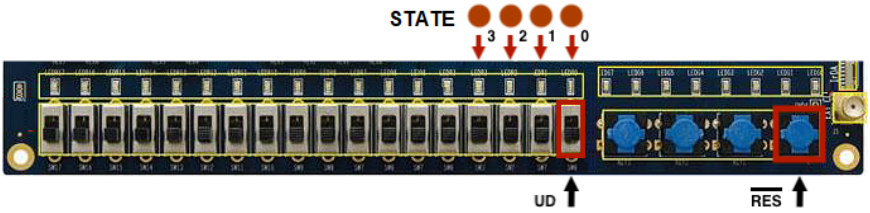


Now, we focus on the figures on the opposite page. In the window shown at the top, the STATE component and index pin 0 (the least significant bit) were selected (highlighted by the blue outline). Then we select the red LED light that we want to associate to it (LEDR[00]) using the menu at the upper right hand side (also outlined in blue).

In the example at the bottom, we select index pin 1 (the bit of weight 2) and associate LEDR[01] to it, as shown by the blue box. Using the same procedure, we then associate the LED lights belonging to STATE of weight 4 and 8 to the LED lights LEDR[02] and LEDR[03].

Notice that for output components as well, the window shows the device selected in the Deeds schematic and the FPGA device associated to it in a red box (as highlighted by the red arrows in the figure).

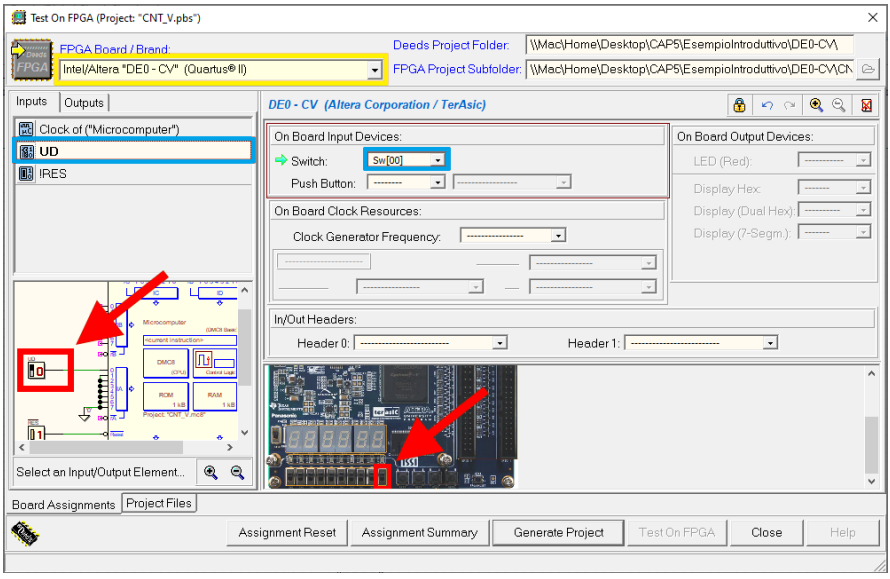
The following picture shows a useful overview of the function of the devices.



5.5.5 Settings for the DE0-CV board

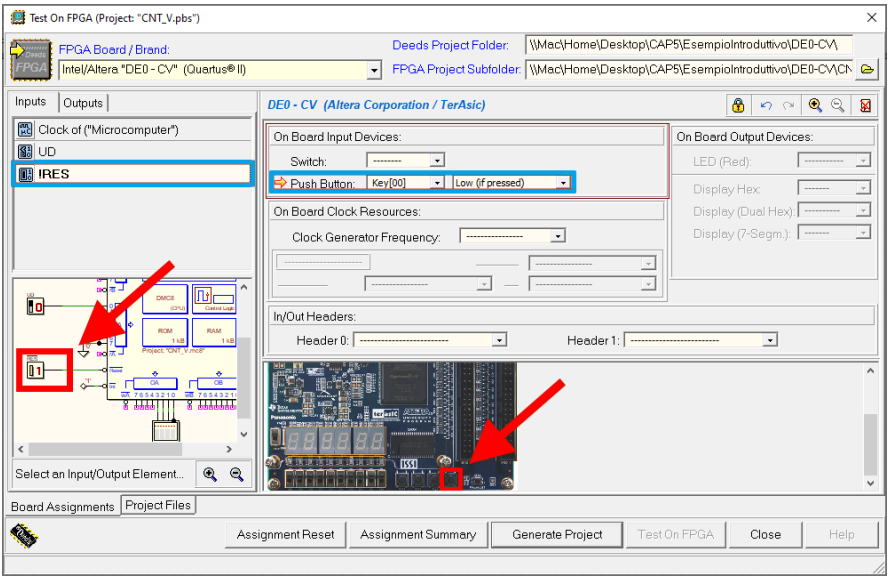
Associating input devices

When we select the DE0-CV board (see the yellow box), the window will appear as follows. With a click, we also select input UD.

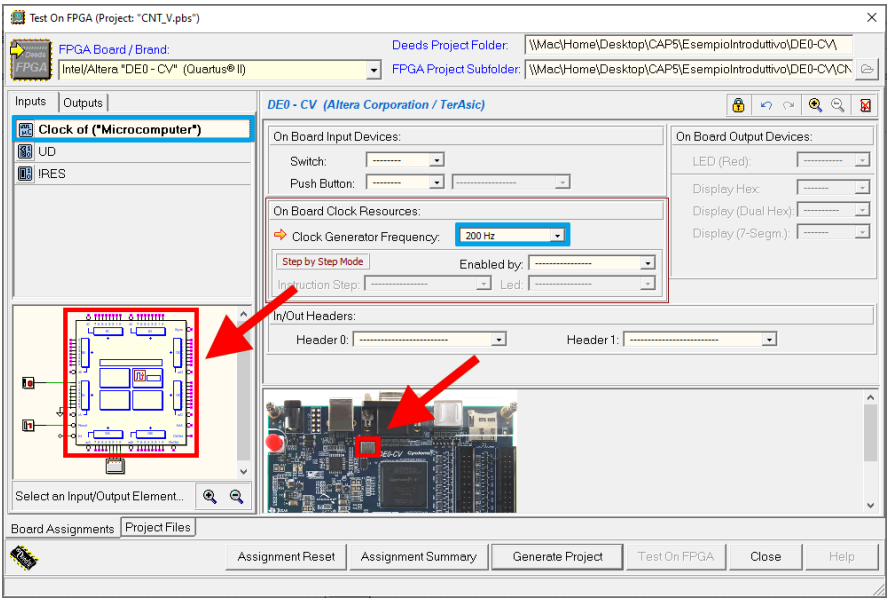


We have associated input UD to the SW[00] switch on the DE0-CV board, as shown by the red boxes and arrows in the figure.

We'll do the same thing (see the following figure) with input $\overline{\text{RES}}$: associate it to KEY[00] (set to generate a low level if pressed).

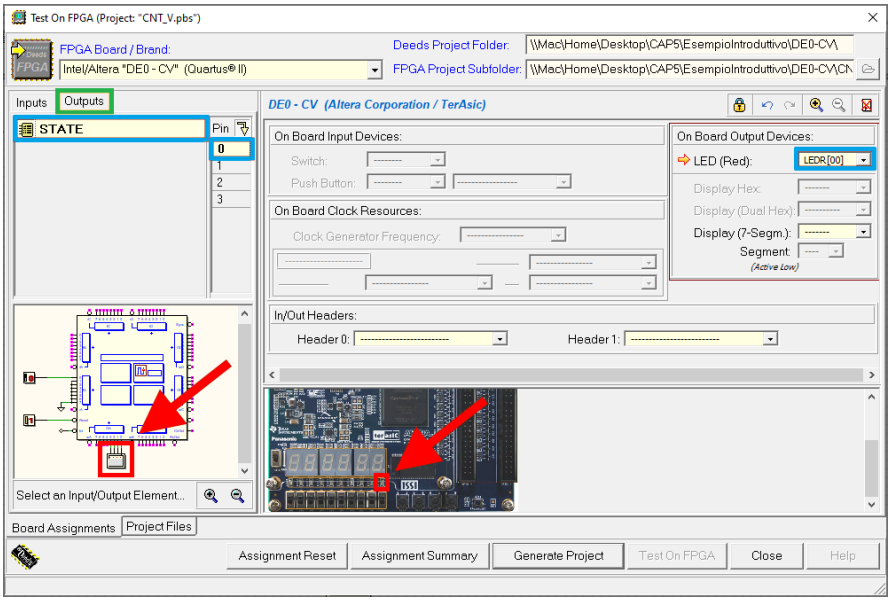


We have associated a 200 *Hz* clock source to the microcomputer to make the behavior of the device observable with the human eye (see more on the clock options in Section 5.5.4).



Associating output devices

Once the input devices are associated, we will deal with the output devices by clicking on the "Outputs" palette (see the green box in the figure).

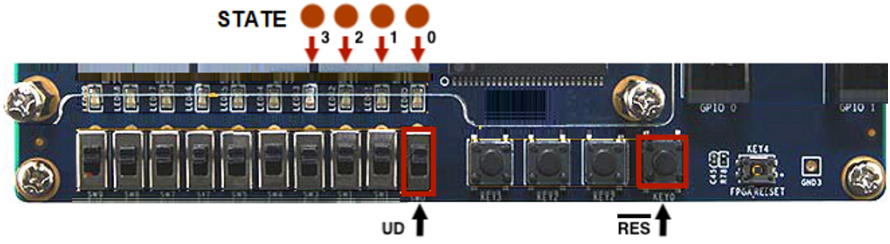


As before, the window lets us select the output device in the Deeds schematic and associate it to an output device on the FPGA board (as shown in the figure, in blue).

We follow the same procedure that we did for input devices and associate the four STATE outputs to the LED lights available on the FPGA board. We select STATE and the index of pin 0 (see the boxes on the upper left hand side of the window), then we use the list box control at the upper right hand side to associate that pin to the red LED light LEDR[00] on the board.

Once this is done, the Deeds component and the corresponding physical device will be marked with red boxes (indicated here by the arrows). We repeat the procedure for the other indexes and associate the bits of weight 2, 4 and 8, to the LED lights LEDR[01], LEDR[02] and LEDR[03].

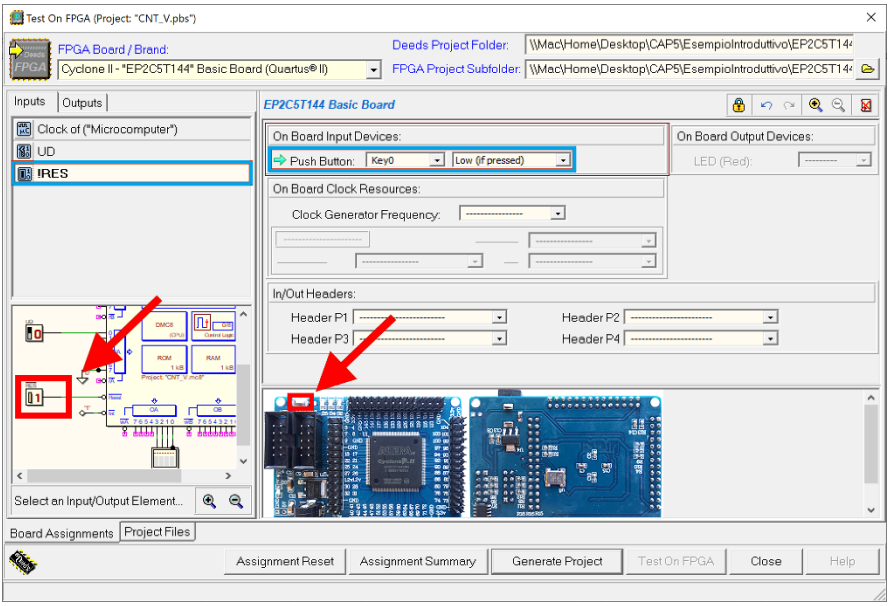
The following figure shows a summary of the associations, useful to test the system on the board.

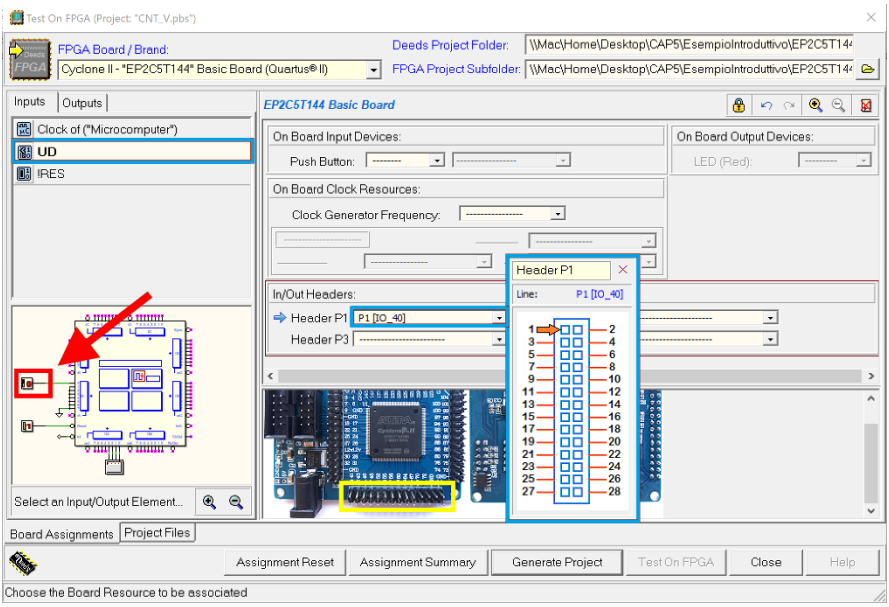


5.5.6 Settings for the EP2C5 board

Associating input devices

When we select the EP2C5 board, the following screen will appear. As considered in Section 5.4.3, the only input device on the board is the KEY0 push-button, which we will use for the reset input $\overline{\text{RES}}$.



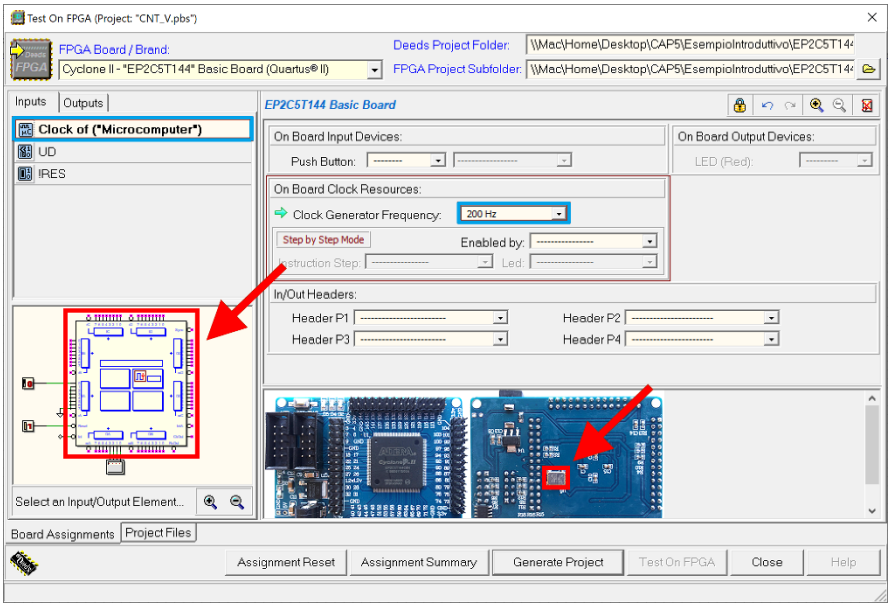


In the figure at the upper part of the opposite page, we selected line $\overline{\text{RES}}$ in the list of inputs on the left, then we associated it to that of the KEY0 push-button (see blue boxes). The arrows point out this association in the schematic and on the board.

To connect the remaining input and output devices, we need to rely on the expansion connectors available on the board.

The figure at the bottom of the opposite page shows the association between input line UD and connector pin P1 (chosen arbitrarily among those available). When we select the connector pin, the drawing of the connector appears with its pins numbered (as seen in the figure). This will be useful when we need to physically connect a wire to the pin because the red arrow points out its position and number.

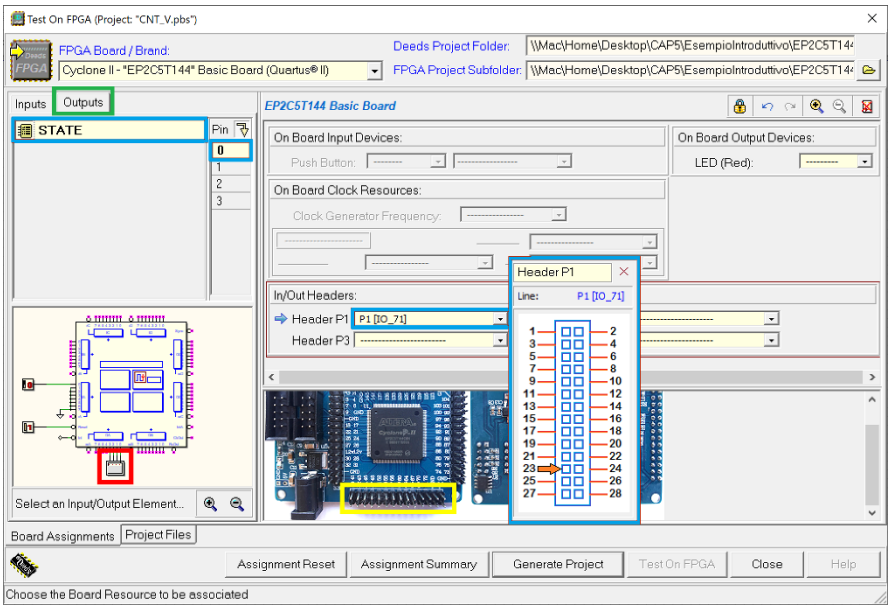
The clock generator is similar to that of the DE0-CV board (50 *Mhz*) and is managed in the same way by Deeds. As shown in the following figure, we assign a frequency of 200 *Hz* to the microcomputer to allow for interaction with the user (for more on this, see the clock options in Section 5.5.4).



Associating output devices

The following figure shows that when we click on "Outputs" (in the green box) we can go on to assign output devices.

On the screen in the blue boxes, we see that pin P1[IO_71] on connector P1, has been associated to index pin 0 on the STATE lines.

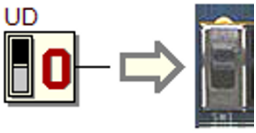


Pins P1[IO_63], P1[IO_53] and P1[IO_44] are assigned to index lines 1, 2 and 3, respectively. As before, when we select the connector pin, the drawing of the connector appears (see figure) with a red arrow indicating the position of the pin.

Connection to physical devices

Once we have assigned the pins through software, we need to physically connect the board and the input/output devices. In this subsection, we will give some practical directions on this subject that should be useful not only for this specific case but also for all the examples given.

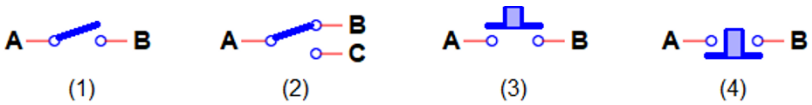
When we want to make an input component like an "Input Switch" correspond to a physical slide switch, the necessary electrical connections have already been set on boards such as the DE2 and the DE0-CV.



In the case where we need to reach the same goal by using an expansion connector pin, we should follow some simple rules to electrically connect it. We have already seen some points on this in the example in Section 4.7.7).

Switches and push-buttons are electromechanical devices and we need to transform their mechanical action into a two-level physical quantity that can be read by a logical device.

The following figure shows the electrical symbols of four of these single-pole devices (there are also other types). They are, from left to right: (1) an "on-off switch", (2) a "double throw switch", (3) a "normally open push-button" and (4) a "normally closed push-button".



(1) The "on-off switch" makes it possible to open or close an electric circuit between electrodes A and B. For example, to turn the light in a room on or off, we use this switch. The device has two stable positions, so the state that we set manually is kept and to change it, we have to push it again.

(2) The "double throw switch" allows us to re-route the current into two distinct connections (A $\leftrightarrow$ B, or A $\leftrightarrow$ C). This type of switch has two stable positions[26]. Here, for reasons of availability, we use this kind of switch but without connecting one of the two electrodes (B or C), so they will actually be used as on-off switches.

(3) The "normally open push-button" behaves like an on-off switch electrically, but it has only one mechanically stable position, that is it closes the contact if we press it and the return spring brings it back to the open position. Other devices that behave this way are keyboard keys, and the buttons on elevators, remote controls or televisions.

(4) The "normally closed push-button" behaves like the open one mechanically, but the contact remains closed when at rest and it opens only when we press it (we will not use this model in our examples).

The left side of the following figure shows the connections of an on-off switch to a connector pin, which is connected to an FPGA chip input (the NOT gate is purely illustrative and represents a logical input).

In the figure on the right, the electrical network is identical but we are connecting a push-button. The difference is only mechanical, in the way they are activated, as described before (and so, in the way they are used).

If the contact is open, the logical input is kept high by the "pull up" resistor[27]. When the contact is closed, however, the logical input is forced low by the electrical connection to the ground (Gnd).

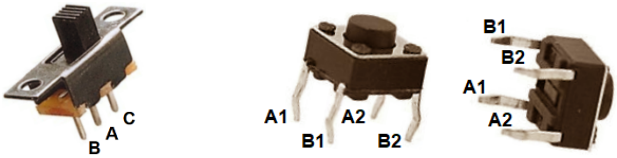
As discussed in the example in (Section 4.7.7), the mechanical properties of electromechanical devices make them susceptible to "mechanical contact

[26] Note that when the contact is moved, it is unconnected for an instant from either contact before closing on the chosen side.

[27] The 10 $K\Omega$ (Kilo-Ohm) value is used as an example and needs to be adapted to the electrical parameters of the input.

bounces". These are generally resolved by doing multiple reads through software. However, in our elemental example, we will ignore this problem for simplicity's sake.

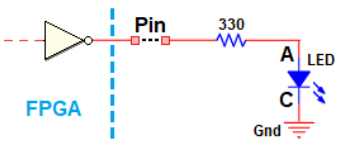
On the left of the following figure, we see a double throw slide switch, whose contact happens by sliding a cursor. The central contact A gets connected by sliding the cursor to contact B or C.



The movement is stable, that is, the cursor remains where we move it. In our examples, we will employ it as an on-off switch, using the contact pairs AB or AC arbitrarily.

In the middle and on the right, we have a "tactile" push-button (shown right side up and on its side). The internal contact is kept open by a spring, but it closes when pressed. When it is released, it opens again.

As the figure shows, the push-buttons often have 4 pins and we can choose those most convenient for connecting to the circuit. A1 and A2 are connected together internally, as are B1 and B2 (so for example we can use only the A1/B1 pair and ignore the other).



In the figure on the left, we see the connection between an LED light and a connector pin that comes from the FPGA (the NOT gate is there as a formality and represents the output of any logical component).

The LED light must be connected with the right polarity. In the figure, the anode is indicated by an A and the cathode by a C[28]. The resistance limits the working current[29] of the LED light.

LED lights on the market come in a wide variety of colors, shapes, sizes and power characteristics. The figure on the left shows a green LED light. Following convention, the longer terminal is always the anode (A) and the shorter is the cathode (C).

When the FPGA output is low, the tension generated by the logic gate is not enough to turn the light on. When the output is high, it can provide enough current to turn on a low-power light, suitable for our purposes.

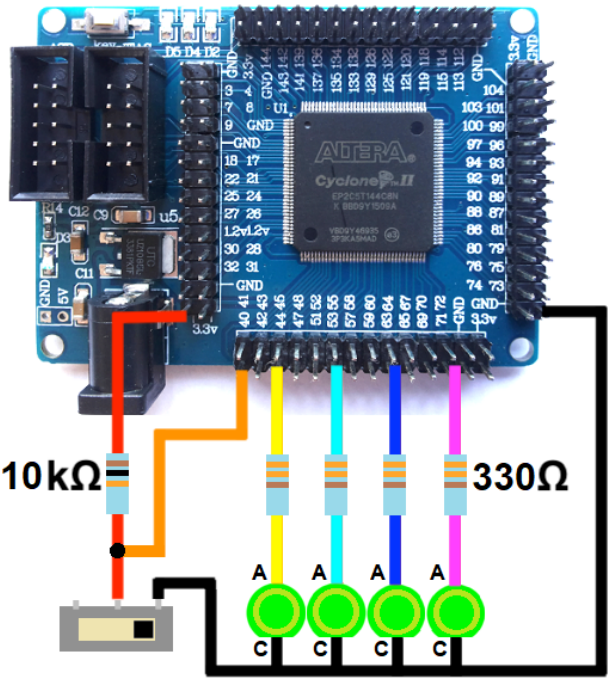
[28] The terms "anode" and "cathode", come from the field of Electrochemistry; the anode is at a higher tension than the cathode.

[29] The value of 330 Ω is just an example that can be applied here. It would be reduced according to the electrical parameters of the LED lights and the output.

Let's go back to our example. For ease of consultation, the figure below provides a table summarizing the associations that have been made (as reported in Deeds).

In/Out Name	VHDL Name	Board Resource	Aux. Info
Inputs:			
Ck of ("Microcomputer")	—	Clock: 200 Hz	
UD	iUD	Header 0: P1 [IO_40]	
!RES	inRES	Push-Button: Key0	Low (if pressed)
Outputs:			
STATE.0	oSTATE(0)	Header 0: P1 [IO_71]	
STATE.1	oSTATE(1)	Header 0: P1 [IO_63]	
STATE.2	oSTATE(2)	Header 0: P1 [IO_53]	
STATE.3	oSTATE(3)	Header 0: P1 [IO_44]	

The result of these connections will be similar to what we see in the figure on the right.



We have attached the central pin of the slide switch to input UD, that is pin P1[IO_40] of the connector at the bottom (the pin is simply reported as '40' on the silk-screen printing of the board).

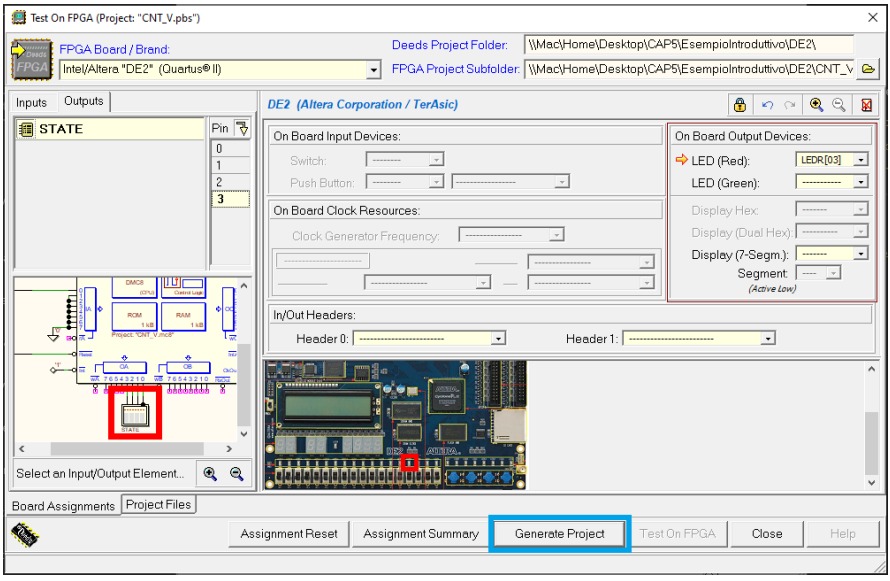
We have also connected one of the two wires of a 10 $K\Omega$ resistor to the same pin of the slide switch, while we brought the other one to the power supply (Vcc, 3.3V, pin on the connector at the left).

Either of the opposite pins of the switch will be connected to the ground (GND, pin on the connector at the right).

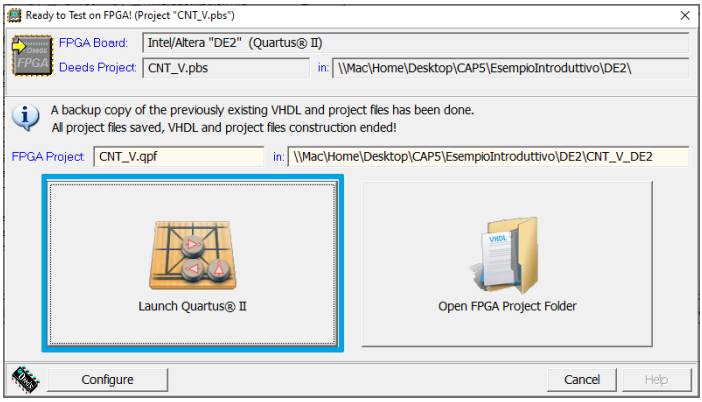
Finally, we connect the 4 LED lights to the outputs that we assigned to connector pins P1[IO_44], P1[IO_53], P1[IO_63] and P1[IO_71] at the bottom (shown as '44', '53', '63' and '71' on the silk-screen printing of the board). We connect these pins to the anodes of the LED lights, taking care to insert the 330 Ω resistor. The cathodes (the shorter terminals) will then be connected to the ground (GND, the same as the pin above).

5.5.7 Converting the Deeds project into VHDL

When we have finished associating the input and output devices of the board to the Deeds project, we generate the VHDL code by clicking on the "Generate Project" button (highlighted by the blue box in the figure below).



After the short time it takes for the VHDL code to be generated automatically, we get to the following window:

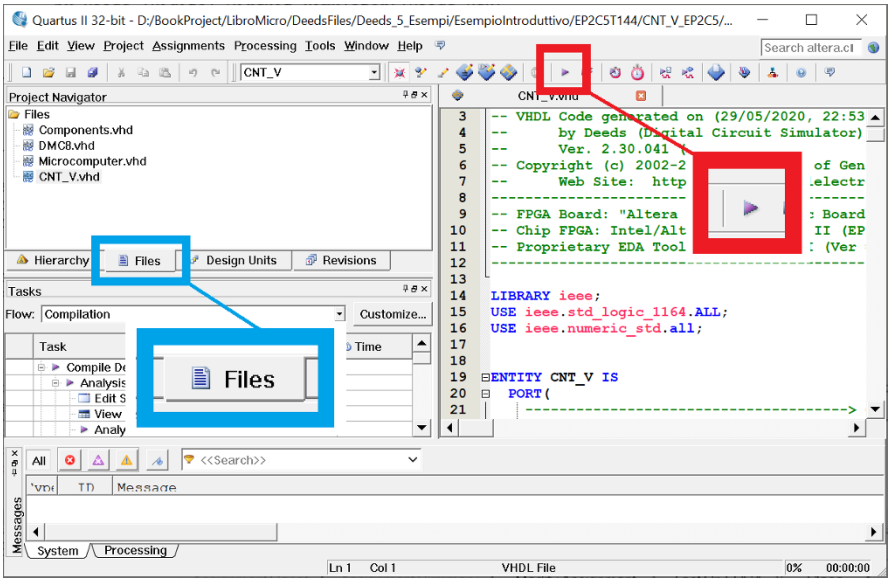


Clicking on the "Launch Quartus® II" button will run the software tool of the same name[30] (introduced in Section 5.3), with which we will go on to program the board.

[30] The Deeds website tutorials on using FPGAs, have useful information on installing the software.

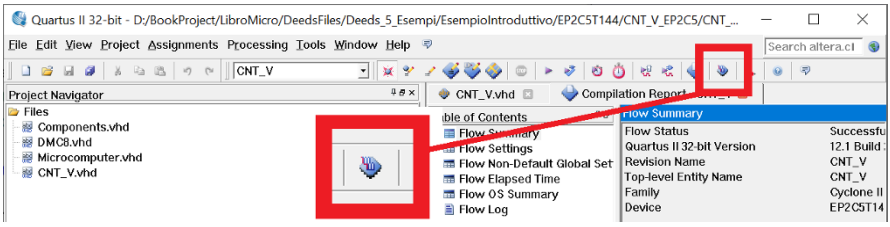
5.5.8 Programming the FPGA board

Once Quartus® II is open, the following window will appear. If we click on the "Files" command in the blue box in the figure, we can examine the VHDL files that come from the Deeds schematic and the association of the input/output components. This may be interesting for those who want to learn more, but it is not necessary for those who want to simply program the circuit on the FPGA board.

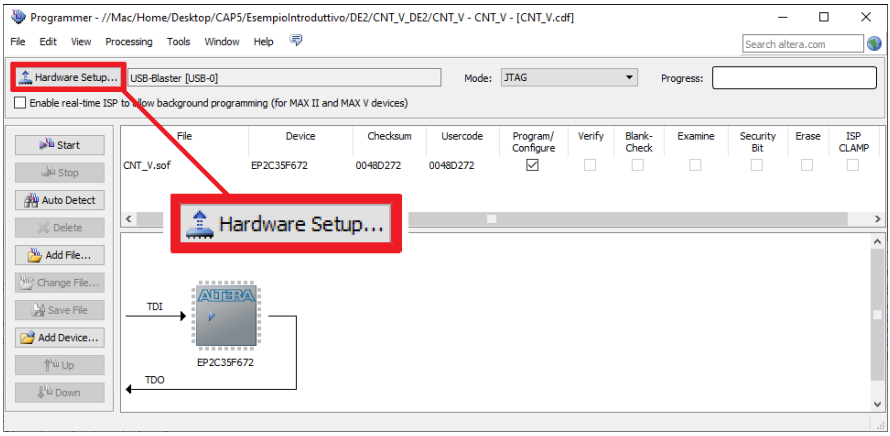


Before going on to actually programming the board, we need to process the program's VHDL file by pressing the "Compile" command (see the red box). After a few minutes, an overview on the project compilation will appear in the window. This indicates that the project is now ready to be loaded on the FPGA board. Usually, many warnings are generated but for the educational scope of these projects, we do not need to take them into account unless they are explicit error messages.

To load the compiled project on the FPGA board, we need to click on the "Programmer" command to open the programmer tool window (highlighted by the red box in the following figure).

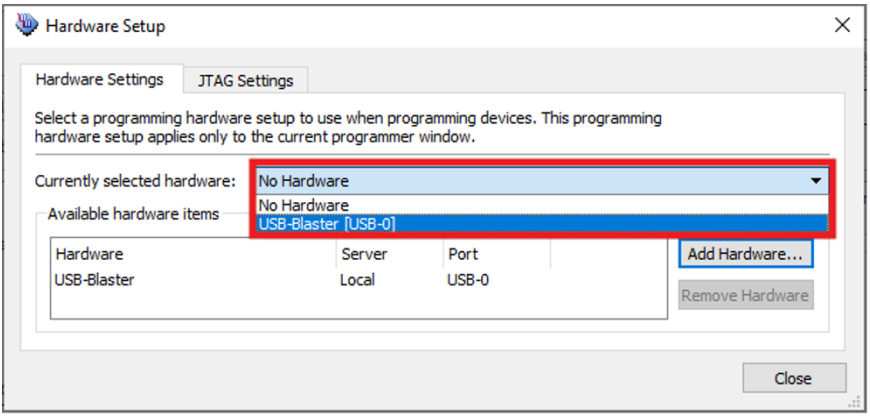


The programmer window will open (see the following figure).



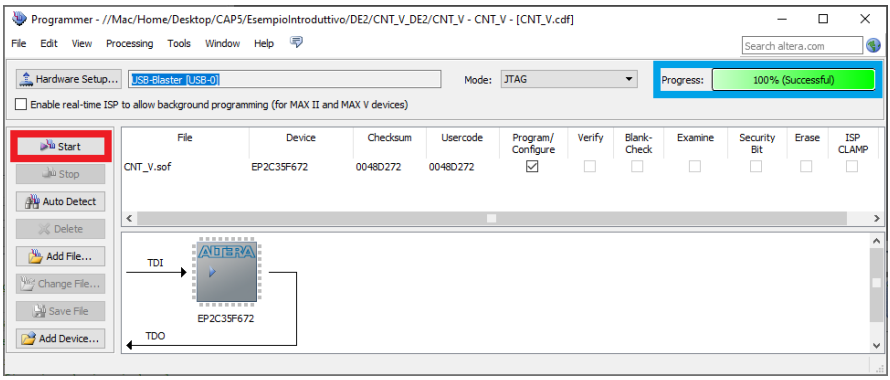
When we click on the "Hardware Setup" button (see the red box), a dialog box will open. This allows us to select the USB port where the programming hardware is connected.

In the "Hardware Setup" dialog box (see the figure below), when we open the "Currently Selected Hardware" drop-down list (in the red box) we can select (for example) "USB Blaster [USB-0]", the USB port recognized by the operating system as connected to the programming hardware in use[31].



When we close this window, we return to the programmer window (see the following figure), where we can launch the board programming by clicking on the "Start" button (outlined in red).

[31] If this option weren't there, we would have to check that the FPGA is connected to the PC and powered. If that option were still missing, we would have to check that the QUARTUS drivers were updated and installed correctly. For more on this, consult the tutorials on the Deeds website: `https://www.digitalelectronicsdeeds.com/learningmaterials/labtopics.html#fpgatutorial`



During the programming phase, we can see how far along we are by looking at the progress bar (see the blue outline in the figure). Once the process is over, we can test the functionality of our project on the FPGA board.

5.6 Project examples

As mentioned in the beginning of this last chapter, we offer some projects developed with the Deeds simulator that are easy to replicate and experiment with on an FPGA board.

Each project comes with an implementation on all three boards that we have shown in this chapter. This will allow the reader who has one of these boards on hand to immediately reproduce these projects.

All the projects shown here are available in their entirety on the Deeds website on the pages regarding this book. Readers can redesign and extrapolate on this material as they like, using the skills they have developed and also that bit of creativity that any future microprocessor system designer/programmer should have. Readers are encouraged to check that the system is working correctly, go through the steps shown here, and then to try to modify it in a creative way.

With the first example, we will learn to control the luminosity of an LED light and with the second we will attempt to create a light gadget. Then we will focus on producing sounds, first by creating a special effects generator and then a music box that can play a famous tune.

From these projects with fun applications, we will move on to more technical/industrial examples. We will first design and build a stepper motor controller and then we will provide two examples of the use of a small alphanumeric and graphical display that had been used in a very popular mobile phone.

As mentioned at the beginning of this chapter, the physical implementation of a system depends greatly on the technologies in use. Cutting edge technologies evolve and become obsolete rapidly, often after only a few years.

This is why it is important to focus on the programming techniques in the previous chapters. The device interfaces will certainly change over time, but the approaches we use to study them and the techniques we use to program them will remain. These approaches and techniques can be reused to design the systems of the future and they will help those who have mastered them to face technological changes with success.

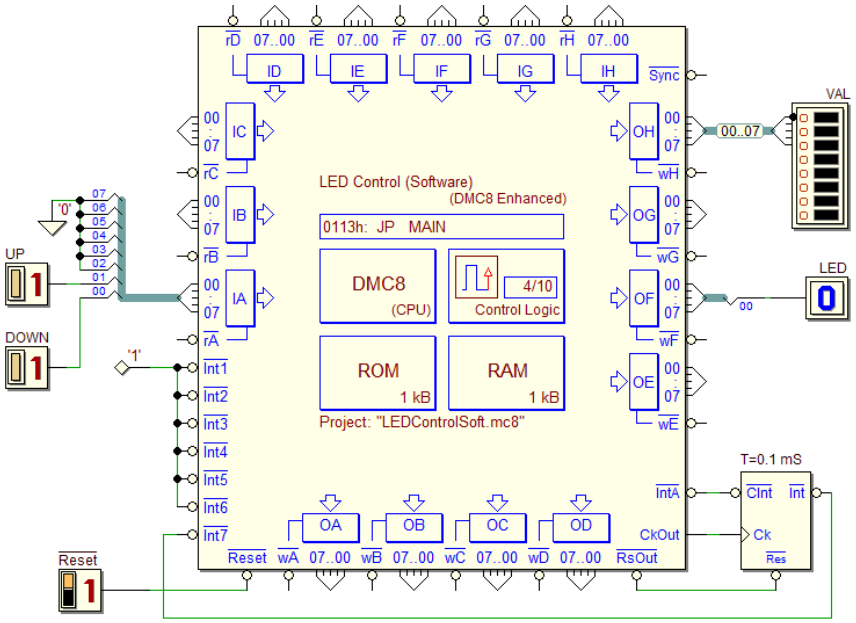
5.6.1 Light dimmer

We need to design and build a light dimmer that can progressively control an LED light from completely off to completely on and vice versa, through the PWM (Pulse Width Modulation) technique.

The dimmer needs an UP and a DOWN push-button. When UP is kept pressed, the light gets progressively brighter and when DOWN is pressed, progressively dimmer. If both buttons are pressed simultaneously, DOWN has priority. When the buttons are released, the level of brightness stays fixed.

5.6.1.1 The system (Version 1)

In this version (see the following figure), the system carries out the task requested by the specifications using a software-only approach.



The system is based on a "DMC8 Enhanced Microcomputer" (see Section 2.4.1) with an added interrupt timer.

The LED has been connected to bit 0 of port OF, while two lines of port IA are dedicated to reading the UP and DOWN buttons (when pressed, the buttons generate a low level). The schematic also shows an 8-LED display (connected to port OH) that we will only use for simulations to assess the state of the LED control.

The timer interrupts the processor every $100\,\mu S$. The requested task is executed exclusively by the interrupt handler, aside from the necessary initializations, which are executed by the main program.

We were introduced to the PWM technique in an example from Section 1.5.3.3. It makes it possible to generate programmable average voltage on one line thanks to a succession of fixed period pulses but variable duration. The average voltage that is generated depends on the ratio between the duration of the high part and the whole period.

Here, the PWM period is $25.6\,mS$ $(= 256 \cdot 100\,\mu S)$, that is 256 timer interrupts. The pulse duration is established by the 8-bit variable VALUE. If it has a value of zero (VALUE = 0) the LED light is off; if it has the maximum value (VALUE = 255), the light remains on for 255 calls and is turned off by the 256th.

The continual on/off flashing cannot be perceived by the human eye due to retinal persistence. This makes it so that our brains do not see variations but rather stable images whose brightness is regulated by the ratio VALUE/256.

The reading of the buttons should be confirmed by two consecutive reads $25.6\,mS$ apart to eliminate contact bounces (see Sections 4.7.7 and 5.5.6).

The program

First we define ports USER, PLED and PVAL (read the comments in the code), then we define three variables in the memory.

VALUE memorizes the PWM pulse duration (a number from 0 to 255), which controls the LED light. PUSER records the previous reading of the push-button port so the debouncing check can be done. TIME, on the other hand, is used to assess the time that passes in the period of the PWM output.

```
USER    EQU   00h       ; IA input port: user push-buttons
PLED    EQU   05h       ; OF output port: LED control
PVAL    EQU   07h       ; OH output port: value monitor

VALUE   EQU   0FC00h    ; LED control value
PUSER   EQU   0FC01h    ; the previous state of USER port
TIME    EQU   0FC02h    ; time counter
```

We define the link to the reset and the interrupt handler. Then, we initialize the Stack Pointer and the variables. We enable the interrupts and enter the

infinite loop MAIN where no operation is executed. Here, no operations are executed because everything will be done by the interrupt handler.

```
            ORG   0000h
            JP    START
            ORG   0038h
            JP    HINT
            ORG   0100h
START:      LD    SP,0FFFFh     ; initialize the Stack Pointer
            LD    A,00h
            LD    (VALUE),A     ; zero the LED control value
            OUT   (PLED),A      ; and the corresponding output port
            LD    (PUSER),A     ; no button previously pressed
            LD    (TIME),A      ; initialize also the time counter
            EI                  ; enable interrupts
MAIN:       JP    MAIN          ; main loop (empty)
```

Every $100\,\mu S$, the interrupt handler is launched. The code begins with the usual PUSH instructions and ends with the required POPs so that we can preserve the content of the registers used by the handler. This is done even though the main program is empty for now, leaving space for any future development of the code.

```
HINT:       PUSH  AF            ; save the used registers
            PUSH  BC
```

As we will see further on, the TIME variable is decremented in the exit code of the handler (that is every $100\,\mu S$). This means that TIME behaves like a cyclical 256 module down counter.

Since we need to turn the LED on for the duration defined by VALUE, we compare VALUE with TIME at every call. If TIME $\geq$ VALUE we jump to NOPULSE and put the output at '0'; if not, we activate the output at '1'.

In other words, the output goes to '1' when TIME, while decrementing, reaches VALUE. The output goes back to '0' when TIME starts back again from 255.

```
            LD    A,(VALUE)     ; copy the control value
            LD    B,A           ; to register B
            LD    A,(TIME)      ; compare the current time
            CP    B             ; with the control value
            JP    NC,NOPULSE
PULSE:      LD    A,00000001b   ; PWM = '1' , if TIME < VALUE
            JP    WPORT
NOPULSE:    LD    A,00000000b   ; PWM = '0', if TIME ≥ VALUE
WPORT:      OUT   (PLED),A      ; update the PWM output port
```

Cyclically, every 256 calls (25.6 mS), TIME is zeroed. So the next check verifies if it is now time to assess the push-buttons.

```
        LD      A,(TIME)        ; check if it is time
        CP      0               ; to assess the push-buttons' state
        JP      NZ,EXIT         ; EXIT if it is not
```

If 25.6 mS have gone by, we check the state of the push-buttons and compare that to the state saved the last time in PUSER. The new state is saved in this variable to be used for the next comparison and in register C.

```
        LD      A,(PUSER)       ; copy the previous port state
        LD      B,A             ; to register B
        IN      A,(USER)        ; read the current push-buttons' state
        CP      B               ; compare it with the previous state
        LD      (PUSER),A       ; save the new state in the variable
        LD      C,A             ; and in register C
        JP      NZ,EXIT         ; debouncing: exit if they are different
```

If the values are different we assume that the cause is a contact bounce or a transient, so we exit. If the values are equal, we validate the state that's just been read. Then, we go on to act on the basis of activation or push-buttons. DOWN has priority over UP, so we test it first and if it is pressed, we do not test the state of UP.

```
TESTDN: BIT     0,C             ; 'DOWN' is pressed? (it has priority)
        JP      NZ,TESTUP       ; jump to the other test if it is not
```

If DOWN is pressed (it is at '0'), we decrement the VALUE variable by 1 unless it is already at zero.

```
        LD      A,(VALUE)       ; get the control value
        CP      0               ; check if it is already at zero,
        JP      Z,EXIT          ; if it is so do not decrement, exit
        DEC     A               ; otherwise decrement the value
        LD      (VALUE),A       ; and save back it in memory
        JP      TESTV           ; jump because 'DOWN' has priority
```

If DOWN is not pressed, we check to see if UP is pressed. We go on if it is.

```
TESTUP: BIT     1,C             ; check if 'UP' is pressed
        JP      NZ,EXIT         ; jump to exit if it is not
```

UP is being pressed (it is at '0'), so we increment the VALUE variable by 1 unless it is already at the maximum value.

```
        LD      A,(VALUE)       ; get the control value
        CP      0FFh            ; check if it is at the maximum value,
        JP      Z,EXIT          ; if it is so do not increment, exit
        INC     A               ; otherwise increment the value
        LD      (VALUE),A       ; and save back it in memory
TESTV:  OUT     (PVAL),A        ; display the new value for test
```

Finally, as mentioned before, we decrement the TIME variable and then exit the interrupt handler. The count is cyclical, so if TIME is at zero, it returns to 255 by decrementing.

```
EXIT:   LD    A,(TIME)    ; count cyclically the time
        DEC   A
        LD    (TIME),A
        POP   BC          ; restore the used registers
        POP   AF
        EI                ; re-enable interrupts and
        RET               ; return to the interrupted program
```

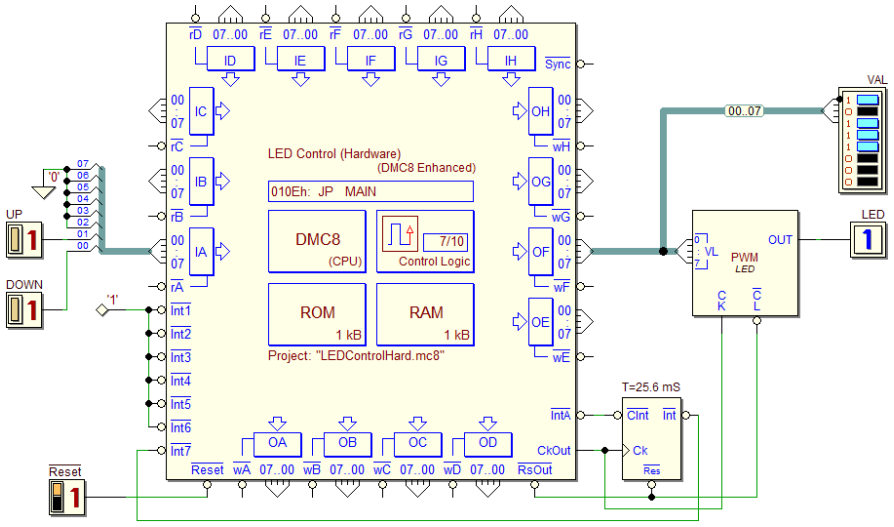
The version we have now analyzed can be changed if we introduce specialized hardware that automatically executes the generation of the PWM output, and therefore can remove that task from those of the microprocessor.

This hardware feature is integrated into microcontrollers. In version 2, which is discussed below, we will go further into the functionality of a specially designed PWM converter, which we add to the microcomputer.

5.6.1.2 The system (Version 2)

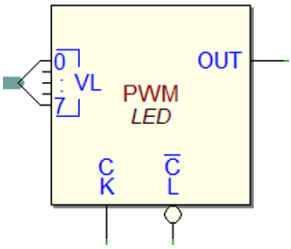
In this new version, we leave it to the microcomputer to handle the function of the push-buttons and the content of VALUE (see version 1 since we've taken some of the code from it).

The following figure shows that the microcomputer no longer controls the LED light directly. Rather, it copies the number in VALUE to the OF port, leaving the external PWM component to generate the correct pulse sequence. The component uses the 10 *MHz* processor clock.



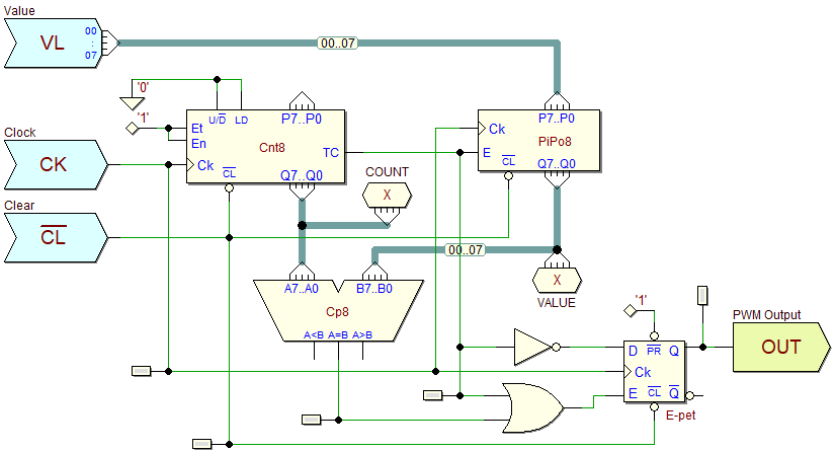
In this version, the timer interrupts the processor only to read the state of the push-buttons, which it does on a regular basis. Its interval is now defined as 25.6 *mS*, as per specifications.

The PWM component (see the figure at the right) achieves the same thing through hardware as version 1 does through software, but can do it much faster. More importantly, the added component allows us to save the processor's computational capacity. It accepts the number VL in the input and produces the corresponding PWM signal in the output, with a period 256 times the clock period.



With a 10 *MHz* clock, the PWM signal period is 25.6 μS (thousands of times faster than the software version).

The inside of the component is described in the following schematic. The 8-bit counter "Cnt8" cyclically produces the decreasing sequence from 255 to 0, which is brought to inputs A7..A0 of the magnitude comparator "Cp8".



Each time the counter reaches zero, its TC (Terminal Count) output enables loading the number VL in parallel register "PiPo8". The output of this is brought to the comparator's inputs B7..B0. The comparator signals the moment when the number generated by the counter becomes equal to the number set at input VL. When this happens, the OUT output (the PWM signal) is brought to '1'; it will be brought to '0' when the counter starts the count again from 255.

The program

The assembly code derives from that of version 1.

```
USER    EQU   00h      ; IA input port: user push-buttons
PLED    EQU   05h      ; OF output port: LED control
VALUE   EQU   0FC00h   ; LED control value
PUSER   EQU   0FC01h   ; the previous state of USER port
```

The TIME variable has been eliminated from the definitions since generating the PWM signal is no longer the job of the software. After the jumps to the start of the program and the interrupt handler, we initialize the Stack Pointer and the variables. The main program is similar to that of version 1.

```
        ORG   0000h
        JP    START
        ORG   0038h
        JP    HINT
        ORG   0100h
START:  LD    SP,0FFFFh     ; initialize the Stack Pointer
        LD    A,00h
        LD    (VALUE),A     ; zero the LED control value
        OUT   (PLED),A      ; and the corresponding output port
        LD    (PUSER),A     ; no button previously pressed
        EI                  ; enable interrupts
MAIN:   JP    MAIN          ; main loop (empty)
```

As mentioned before, the interrupt handler only manages debouncing and incrementing/decrementing the VALUE variable.

```
HINT:   PUSH  AF            ; save the used registers
        PUSH  BC
        LD    A,(PUSER)     ; copy the previous port state
        LD    B,A           ; to register B
        IN    A,(USER)      ; read the current push-buttons' state
        CP    B             ; compare it with the previous state
        LD    (PUSER),A     ; save the new state in the variable
        LD    C,A           ; and in register C
        JP    NZ,EXIT       ; debouncing: exit if they are different
```

With regard to this functioning, the code is identical to version 1. The DOWN push-button is checked first (priority).

```
TESTDN: BIT   0,C           ; 'DOWN' is pressed? (it has priority)
        JP    NZ,TESTUP     ; jump to the other test if it is not
```

If the button is pressed, the content of VALUE is decremented. If it is not already at the minimum value, then it jumps to the SEND label.

```
        LD    A,(VALUE)     ; get the control value
        CP    0             ; check if it is already at zero,
        JP    Z,EXIT        ; if it is so do not decrement, exit
        DEC   A             ; otherwise decrement the value
        LD    (VALUE),A     ; and save back it in memory
        JP    SEND          ; jump because 'DOWN' has priority
```

If DOWN is not pressed, the check moves to the state of the UP push-button. If UP is not pressed either, we exit the handler without changing VALUE.

```
TESTUP: BIT   1,C           ; check if 'UP' is pressed
        JP    NZ,EXIT       ; jump to exit if it is not
```

If, however, UP is pressed, the content of VALUE is incremented unless it is already at the maximum value. Then at the SEND label, we copy the new content of VALUE to the PLED port and pass this number to the PWM module.

```
        LD      A,(VALUE)     ; get the control value
        CP      0FFh          ; check if it is at the maximum value,
        JP      Z,EXIT        ; if it is so do not increment, exit
        INC     A             ; otherwise increment the value
        LD      (VALUE),A     ; and save back it in memory
SEND:   OUT     (PLED),A      ; send the value to the PWM generator
EXIT:   POP     BC            ; restore the used registers
        POP     AF
        EI                    ; re-enable interrupts and
        RET                   ; return to the interrupted program
```

5.6.1.3 Implementation on FPGA

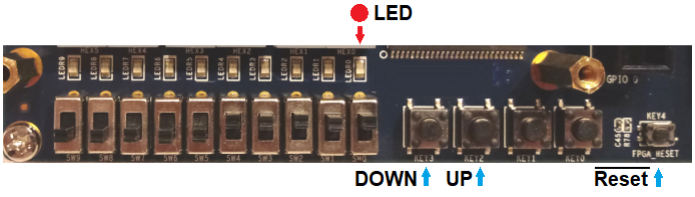
From the functional perspective, versions 1 and 2 carry out the same operations. To implement them on an FPGA board, however, we have used version 2, which is available on the Deeds website in the online content for this section. This allows interested readers to implement version 1 on their own. The following paragraphs will offer synoptic images that summarize the associations chosen for each board[32].

The DE2 board

The following figure provides a visual indication of the devices used on the DE2 board, which is useful when we test the system.

Scheda DE0CV

The assignment of devices for the DE0-CV board is very similar to that for the DE2.



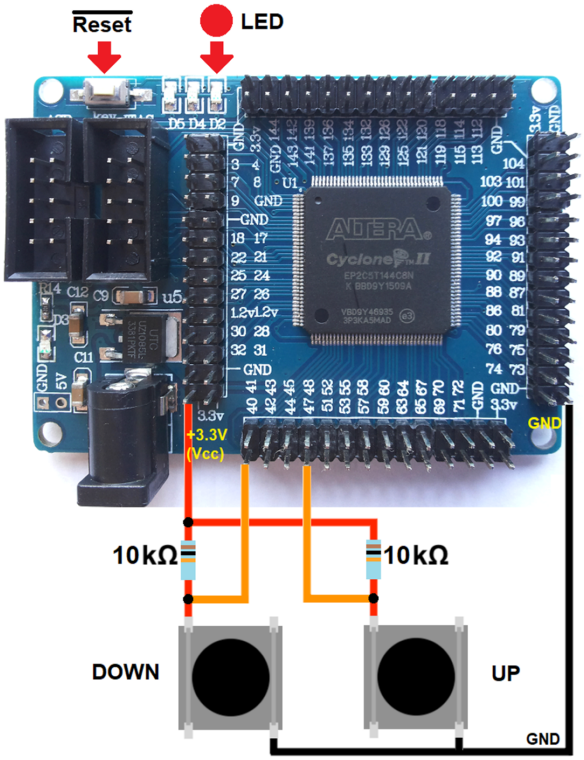
[32] For more information on the associations, see the online content for this section.

The EP2C5 board

For the EP2C5 board, since some of the connections have to be made by hand, it is a good idea to use the summary generated by Deeds (see the following figure). Notice that the outputs that were just set for the simulation have not been mapped.

In/Out Name	VHDL Name	Board Resource	Aux. Info
Inputs:			
Ck of ("LED	—	Clock: 10 MHz	
!Reset	inReset	Push-Button: Key0	Low (if pressed)
UP	iUP	Header 0: P1 [IO_40]	
DOWN	iDOWN	Header 0: P1 [IO_47]	
Outputs:			
!Int7	onInt7		
VAL.0	oVAL(0)		
VAL.1	oVAL(1)		
VAL.2	oVAL(2)		
VAL.3	oVAL(3)		
VAL.4	oVAL(4)		
VAL.5	oVAL(5)		
VAL.6	oVAL(6)		
VAL.7	oVAL(7)		
LED	oLED	LED (Red): LED0	

The following figure shows these connections and highlights the network for the two push-buttons to attach to the board.



5.6.2 LED gadget

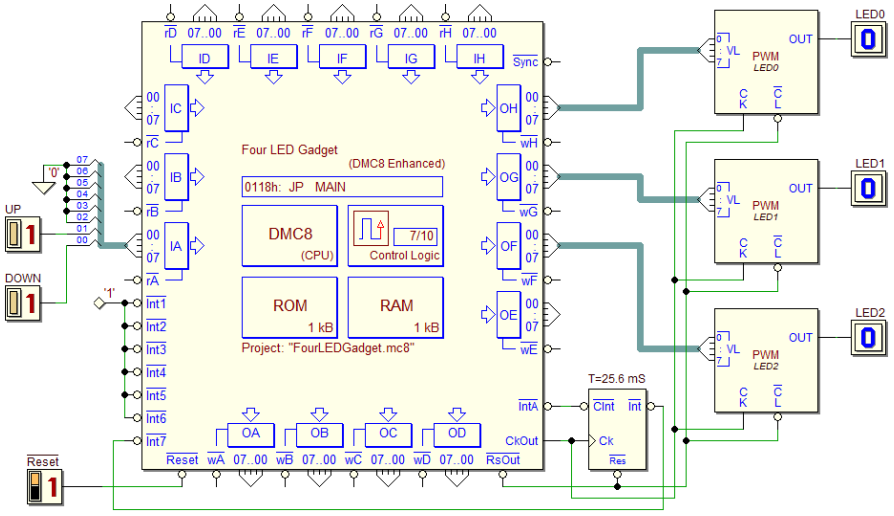
We need to design and build a small gadget that "rotates" the on/off state of three LED lights in a cyclical sequence. The three lights have to gradually turn on and off like a sine wave. The waves are offset from each other by 120 degrees.

The gadget has two push-buttons (UP and DOWN), that control the speed of rotation between a high and low point (that can be perceived by the human eye). When UP is kept pressed, the speed of rotation increases progressively; when DOWN is kept pressed, it decreases progressively. When neither button is being pressed, the speed remains the same. If both buttons are being pressed at the same time, DOWN has priority over UP.

5.6.2.1 The system

Some elements of this project are reminiscent of the previous example (see Section 5.6.1). The push-button handling is similar, so some of the code can be reused in this project. Also, to control the LEDs' brightness, it would be a good idea to use the PWM component described in the previous example.

As we can see in the following figure, a "DMC8 Enhanced Microcomputer" uses output ports OH, OG and OF to drive three PWM generators connected to LED lights 0, 1 and 2, respectively. The generators use the same 10 *MHz* processor clock.



Following the previous example, we use the two lines of the IA port to read the UP and DOWN buttons (when pressed, they generate a low level).

We also add a timer that will interrupt the processor every 25.6 *mS*, and will be used both for debouncing the reading of the buttons and to handle the on/off rotation of the LED lights.

The same time has been chosen as that of the previous example, but this is not critical and could be changed (in which case the speed of rotation of the lights would change as well).

The example from Section 4.7.3 also inspires the technique of generating outputs with a sinusoidal shape in this case. Here, we reuse the waveform table in Section 4.7.3 and the method to read it. Every time the handler is called, the read index is incremented by the size the user sets through the UP and DOWN buttons, thus controlling the resulting oscillation frequency.

The three sine waves that control the PWM modules are phase-shifted from each other by 120 degrees. So, the read index of the table is offset by a value corresponding to that angle for each light (as required by the specifications).

All the required tasks are carried out exclusively by the interrupt handler. The main program only does the necessary initializations of the ports and the variables used.

The program

First we define ports USER, PLED0, PLED1 and PLED2, then we declare the PHASE120 constant. This will allow us to offset the three sine waves by 120 degrees from each other (1/3 of a round angle but in 256ths).

```
USER      EQU   00h       ; IA input port: user push-buttons
PLED0     EQU   07h       ; OH output port: LED0
PLED1     EQU   06h       ; OG output port: LED1
PLED2     EQU   05h       ; OF output port: LED2
PHASE120  EQU   85        ; 85/256 = about 120 degrees
```

Among the variables, we have PUSER, which memorizes the state of the buttons (for debouncing checks). FREQ is the multiplication parameter of the oscillation frequency, which the user increments/decrements through the push-buttons. Finally, the ANGLE variable records the current angle of the generation of the sine wave related to LED0 (the others use the same angle, with the addition of the PHASE120 offset).

```
PUSER     EQU   0FC00h    ; USER port previous state
FREQ      EQU   0FC01h    ; frequency parameter (0..31)
ANGLE     EQU   0FC02h    ; current angle
```

Then we define the jumps to the program and the interrupt handler.

```
          ORG   0000h
          JP    START
          ORG   0038h         ; Int. 7
          JP    HINT7
          ORG   0100h
```

When the program is launched, we initialize the Stack Pointer, the variables and the output ports. The FREQ parameter is set at an intermediate value.

```
START:  LD    SP,0FFFFh     ; initialize the Stack Pointer
        LD    A,00h         ; zero the output ports
        OUT   (PLED0),A     ; LED0, LED1 and LED2
        OUT   (PLED1),A
        OUT   (PLED2),A
        LD    (ANGLE),A     ; zero the current angle
        LD    (PUSER),A     ; no button previously pressed
        LD    A,5           ; default frequency
        LD    (FREQ),A
```

Then we enable the interrupts and enter the main loop (which is empty since everything is done by the interrupt handler).

```
        EI                  ; enable interrupts
MAIN:   JP    MAIN          ; main loop (empty)
```

The interrupt handler is organized into different subprograms. The contents of the registers in use are saved at the beginning and then restored at the end. This allows us to leave the handler unchanged if functions are added to the main program (which is empty now).

```
HINT7:  PUSH  AF            ; save the used registers
        PUSH  BC
```

We copy the FREQ parameter to register B, and increment the ANGLE variable by this value. Every time the timer is called, that is, we move forward in reading the waveform table by using ANGLE as an index. The new value is also saved in register C.

```
        LD    A,(FREQ)      ; copy the frequency parameter
        LD    B,A           ; to register B
        LD    A,(ANGLE)     ; update the current angle
        ADD   A,B           ; by adding the parameter to it
        LD    (ANGLE),A
        LD    C,A           ; copy the new angle to register C, too
```

The angle is passed through A to the WAVEFORM function, that reads the value table and returns the corresponding sample of the function, which is then sent to the PWM component that drives LED0.

```
        CALL  WAVEFORM      ; read the sample from the table
        OUT   (PLED0),A     ; send it to the port of LED0
```

This same operation is repeated twice more for the other two lights, but with 120 degrees added to the current index (what we saved in register C).

```
        CALL  PHASE         ; shift the phase of 120 degrees
        CALL  WAVEFORM      ; read the sample from the table
        OUT   (PLED1),A     ; send it to the port of LED1
```

After we send the value to the PWM component that drives LED1, we deal with the one connected to LED2.

```
        CALL  PHASE       ; shift the phase of 120 degrees
        CALL  WAVEFORM    ; read the sample from the table
        OUT   (PLED2),A   ; send it to the port of LED2
```

Then we call UPDOWN, which assesses the state of the push-buttons and increments (or decrements) the FREQ parameter, and we exit the handler.

```
        CALL  UPDOWN      ; assess the state of push-buttons
        POP   BC          ; restore the saved registers
        POP   AF
        EI
        RET
```

The PHASE subprogram increments the current angle by 120 degrees and puts it back into register A (as we have seen, it is called before the WAVEFORM function in relation to LED lights 1 and 2).

```
PHASE:  LD    A,C         ; get the current angle from register C
        ADD   A,PHASE120  ; move it 120 degrees ahead
        LD    C,A         ; save back it in C and also return it in A
        RET
```

As mentioned before, the UPDOWN subprogram manages the buttons. It checks their state and compares that with what is saved in PUSER. The new state is then saved in this variable and in register C to be used in the next comparison. If the two states are different, we assume a transitory or contact bounce, so we exit the subprogram with the RET NZ instruction.

```
UPDOWN: LD    A,(PUSER)   ; copy the push-buttons previous state
        LD    B,A         ; to register B
        IN    A,(USER)    ; read the current push-buttons state
        CP    B           ; compare it with the previous
        LD    (PUSER),A   ; save back the new state in memory
        LD    C,A         ; and in register C
        RET   NZ          ; debouncing: if they are different, exit
```

If they are the same, the state is valid. We then go ahead and assess the DOWN button first (it has priority). If DOWN is pressed (= '0'), we decrement the FREQ parameter unless it is not already at the lowest value[33], and we return to the calling program.

```
TESTDN: BIT   0,C         ; is the DOWN button pressed?
        JP    NZ,TESTUP   ; (it has priority) jump if it is not
        LD    A,(FREQ)    ; get the previous parameter value
        CP    1           ; if it is already at the lowest value
        RET   Z           ; do not decrement it and exit,
        DEC   A           ; otherwise decrement
        LD    (FREQ),A    ; and save back it in memory
        RET
```

[33] The lowest value must be greater than 0, otherwise the advance of the angle stops.

If DOWN is not being pressed, we jump to TESTUP to check the UP button. If it is not being pressed either, we exit the function with the RET NZ instruction; if it is being pressed, we increment the FREQ variable (if it is not already at the highest value[34]), and we return to the calling program.

```
TESTUP:   BIT   1,C          ; is the UP button pressed?
          RET   NZ           ; exit if it is not
          LD    A,(FREQ)     ; get the previous parameter value
          CP    31           ; if it is already at the highest value
          RET   Z            ; do not increment it and exit,
          INC   A            ; otherwise increment
          LD    (FREQ),A     ; and save back it in memory
          RET
```

As mentioned before, the WAVEFORM function provides the value of the sine in function of the angle that we pass in the accumulator. The function uses the SINTAB table, which contains the values of the positive half cycle of the sine wave. The negative values are retrieved from the positive ones by two's complement. This is almost identical to what is used in the programming example in Section 4.7.3). Therefore, we'll bypass any explanation of the details on how it functions. The only difference is that the returned values are offset into the positive range (0..254) and constant +127 is added.

```
WAVEFORM: PUSH  HL           ; save register HL and BC
          PUSH  BC
          LD    C,A          ; save bit 7 of the angle in C, and mask
          AND   01111111B    ; it to avoid readings outside the table
          LD    HL,SINTAB    ; get the base address of the table
          ADD   A,L          ; add the index to it
          LD    L,A          ; to obtain the address of the location
          JP    NC,NoCarry   ; of interest in register HL
          INC   H
NoCarry:  LD    A,(HL)       ; get the value
          BIT   7,C          ; check if we are in second half wave
          JP    Z,Positive   ; if not, the value is positive
Negative: NEG                ; otherwise invert the sign of the value
Positive: ADD   A,127        ; move the samples in the range 0..254
          POP   BC           ; restore registers BC and HL
          POP   HL
          RET
```

The SINTAB table is defined in the ROM and has been calculated previously. It is identical to the one used in the example cited above. For convenience, part of it is re-printed here.

```
SINTAB:   DB    000          ; x = 0    (0 degrees)
          DB    003          ; x = 1
          DB    006          ; x = 2                    (cont.)
```

[34] The highest value was determined experimentally and can be changed.

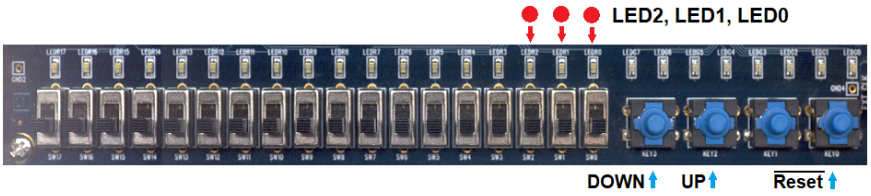
```
                ... omissis ...
        DB      088             ; x = 31
        DB      090             ; x = 32    (45 degrees)
        DB      092             ; x = 33
                ... omissis ...
        DB      127             ; x = 63
        DB      127             ; x = 64    (90 degrees)
        DB      127             ; x = 65
                ... omissis ...
        DB      092             ; x = 95
        DB      090             ; x = 96    (135 degrees)
        DB      088             ; x = 97
                ... omissis ...
        DB      006             ; x = 126
        DB      003             ; x = 127
        DB      000             ; x = 128   (180 degrees, not used)
```

5.6.2.2 Implementation on FPGA

The following sections will show images that summarize the connections chosen for each board[35], which are useful for testing the system.

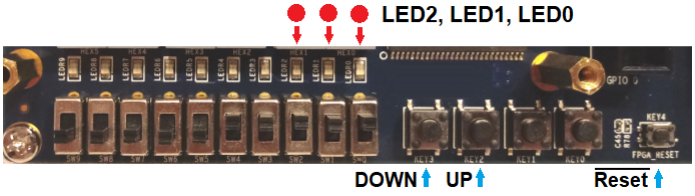
The DE2 board

The following figure shows the choice of devices for the DE2 board.



The DE0CV board

The assignment of devices for the DE0-CV board is very similar to that set for the DE2.



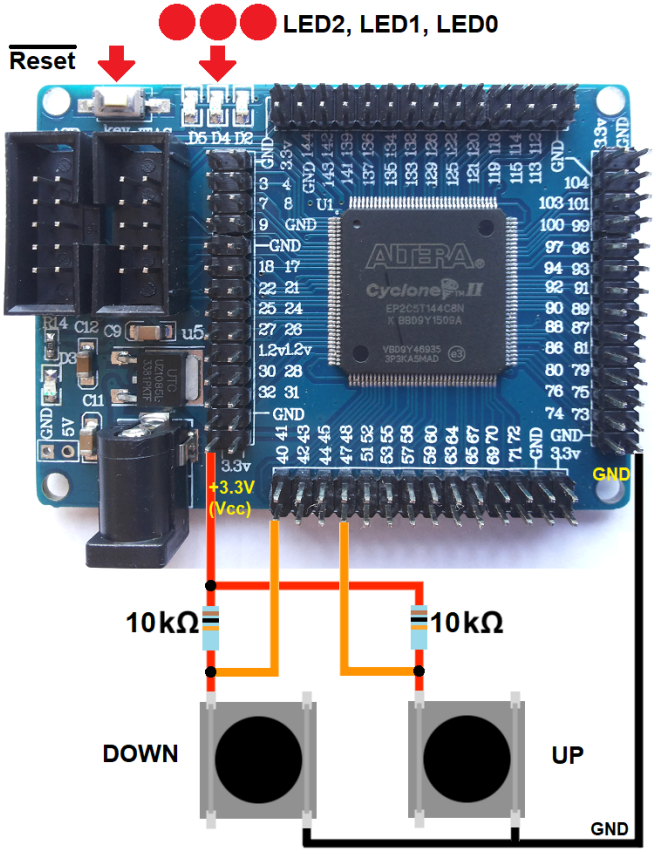
[35] For more information on the associations, see the online content for this section.

The EP2C5 board

The connections on the EP2C5 board require two external push-buttons to be connected. Among the resources on the board, we use the three red LED lights and the push-button (for reset). In any case, it should be useful to have access to the Deeds summary (see the following figure).

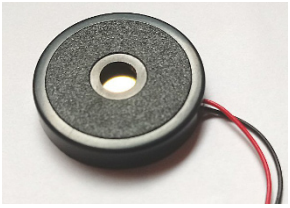
In/Out Name	VHDL Name	Board Resource	Aux. Info
Inputs:			
Clk of i"Four	—	Clock: 10 MHz	
!Reset	inReset	Push-Button: Key0	Low (if pressed)
DOWN	iDOWN	Header 0: P1 [IO_40]	
UP	iUP	Header 0: P1 [IO_47]	
Outputs:			
!Int7	onInt7		
LED0	oLED0	LED (Red): LED0	
LED1	oLED1	LED (Red): LED1	
LED2	oLED2	LED (Red): LED2	

Below is a photograph of the board with the physical connections to make for the two push-buttons superimposed on it. Pay attention to the connections of the 10 KΩ pull-up resistors (highlighted in red and orange).



5.6.3 Special sound effects

We want to design and build a sound effects generator that imitates the typical sounds of 1980s video games. We will use a small piezoelectric speaker (called a "buzzer", see the photo at the right) as an acoustic transducer. This speaker has the benefit of being able to directly connect to the logic circuit without the need of an amplifier.

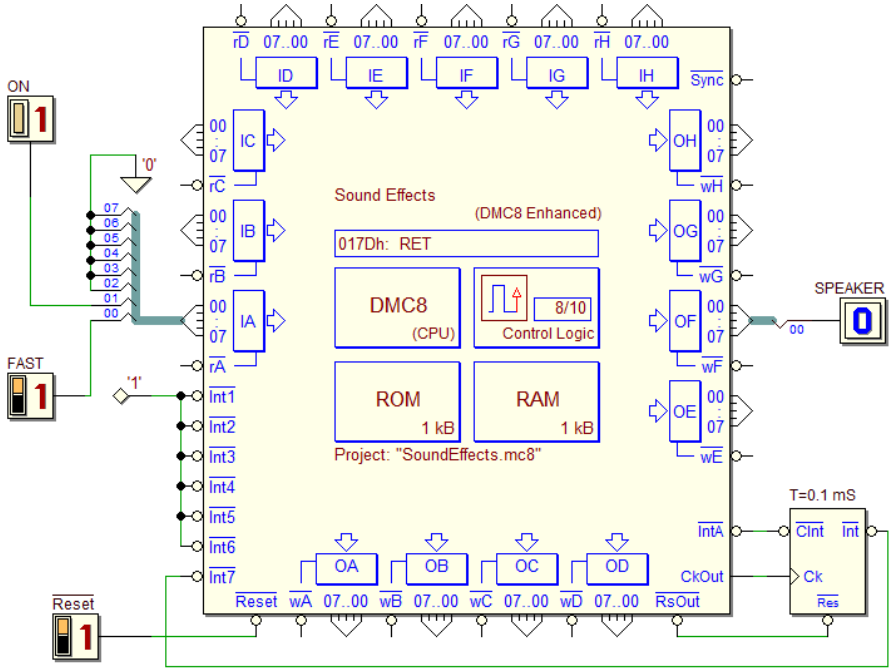


Piezoelectric materials are able to deform when an electrical field is applied. If the electrical field varies over time, we will get a transformation into acoustic vibrations.

5.6.3.1 The system

The generator requires a push-button (ON) and a switch (FAST). The sound is generated when we press the ON button. FAST allows us to select the type of effect we want (fast or slow pace).

The following figure shows the system, which uses the "DMC8 Enhanced Microcomputer" component with an added timer set to interrupt the processor every 100 μS.



Through the IA port, we read the state of ON and FAST (we will not do debouncing checks here since they are not required by the application).

The piezoelectric speaker is directly connected to the SPEAKER line (bit 0 of output port OF). Since the audio signal sent by the transducer is generated by a logic output, it will always have exactly two levels. To generate the variety of sounds that we expect, we need to work on the oscillation frequency of the logic levels and above all, on the variation of the frequency itself, i.e., on its "modulation".

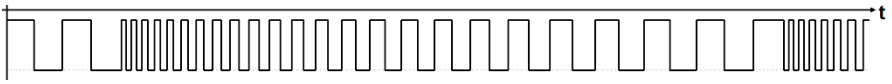
The main program acquires the inputs and translates them into two variables: PLAY and TMAX (used by the interrupt). PLAY enables sound generation and is obtained by the state of the ON button. TMAX, however, depends on the FAST switch. It contains the value used to re-initialize the time count that the frequency variation (fast or slow) and the type of effect generated depends on, as we shall see.

The interrupt handler primarily inverts the level of the SPEAKER output at the right time. To do this, it decrements a counter every time it is called and executes the inversion each time it zeroes. The count is therefore re-initialized with the value contained in the PERIOD variable.

If PERIOD contained a constant, the frequency of the output would be fixed. However, PERIOD is regularly incremented so it gradually lowers the frequency of the signal generated. This increment is cyclical (once it gets to the highest value, it restarts at the lowest).

This way, the frequency is modulated by a signal with a "descending sawtooth" trend. How quickly the frequency is decremented is in turn controlled by the TMAX variable, which as we remember, depends on the FAST switch.

Therefore, we get a fast or slow decline of the frequency generated that produces two types of sounds that are very different from each other from a psychoacoustic perspective[36]. The following timing diagram shows the rate of the signal in the output, which we obtain by setting FAST to '1'.



The program

We define the addresses of ports PSEL and PSOUND, and then the variables.

```
PSEL    EQU   00h     ; IA input port: user commands
PSOUND  EQU   05h     ; OF output port: audio output
```

The COUNT variable assesses the time that elapses between two output transitions. PERIOD, TMAX and PLAY have already been discussed. The SOUND variable is the software copy of the PSOUND output port. It records the last value written on the port in order to invert the value of the output line when requested.

[36] Obviously, we suggest creating the system and listening to the result!

TIME assesses the time elapsed between two variations of PERIOD and so allows us to modulate the frequency, making the two different modulations (fast or slow) possible.

```
COUNT   EQU   0FC00h   ; time counter (between two transitions)
PERIOD  EQU   0FC01h   ; requested time between two transitions
SOUND   EQU   0FC02h   ; software copy of the output
TIME    EQU   0FC03h   ; time counter (between two variations)
TMAX    EQU   0FC04h   ; requested time between two variations
PLAY    EQU   0FC06h   ; sound generation flag (ON/OFF)
```

The following constants determine the final result of the sound effect and can be changed as desired, even experimentally.

```
PHIGH   EQU   50       ; maximum time between two transitions
PLOW    EQU   7        ; minimum time between two transitions
TMLONG  EQU   255      ; time between variations (long and short)
TMSHORT EQU   25
```

We define the jumps to the main program and the interrupt handler.

```
        ORG   0000h
        JP    START
        ORG   0038h
        JP    HINT
        ORG   0100h
```

After the initialization of the Stack Pointer, the main program calls the CLEAR subprogram, which defines the default values of the variables and the output port. After that, we enable the interrupts.

```
START:  LD    SP,0FFFFh   ; initialize the Stack Pointer
        CALL  CLEAR       ; and all the variables and the output port
        EI                ; enable interrupts
```

The main program translates the state of the push-button and the switch into the proper values of variables PLAY and TMAX, as mentioned before. If the content of PLAY is not zero, the sound is generated. The FAST switch determines if the TMSHORT or the TMLONG constant is loaded in TMAX (corresponding to the quick or slow modulation, respectively).

```
MAIN:   IN    A,(PSEL)    ; read the user commands
        LD    B,A         ; save their state
        CPL               ; invert all the bits
        AND   00000010b   ; if the push-button is pressed, bit 1 = '1'
        LD    (PLAY),A    ; save the 'ON' command flag
        BIT   0,B         ; assess the line 'FAST' and assign
        LD    A,TMLONG    ; the update time of the time between
        JP    Z,LONG      ; two transitions (long or short time)
        LD    A,TMSHORT
LONG:   LD    (TMAX),A
        JP    MAIN
```

The subprogram below, CLEAR, initializes all the default values of the variables and the output port (for more on this, read the comments in the assembly code). As we have seen, it is called at the start of the main program, but the interrupt handler uses it as well, as we will see further on.

```
CLEAR:  LD    A,PHIGH       ; this subprogram initialize:
        LD    (PERIOD),A    ; 1) the time between two transitions
        LD    (COUNT),A     ; 2) and its counter
        LD    A,00h
        LD    (PLAY),A      ; 3) zero the play enable
        LD    (SOUND),A     ; 4) zero the output software copy
        OUT   (PSOUND),A    ; and the output port
        LD    A,TMLONG      ; define a default
        LD    (TIME),A      ; to the time count
        RET
```

The interrupt handler is launched by the timer every $100\,\mu S$. The only registers used in it are the accumulator and the flags, so we save them with a PUSH AF instruction. On exit, we restore its original value with the corresponding POP AF.

```
HINT:   PUSH  AF            ; save the used register
```

We immediately check if the generation is enabled. If it isn't, we exit without generating anything, but we re-initialize the variables in play by calling CLEAR (even though it is called all the time, this poses no problem).

```
        LD    A,(PLAY)      ; is the generation enabled?
        CP    0
        JP    NZ,SING       ; jump if it is, otherwise
        CALL  CLEAR         ; re-initialize all the variables and exit
        JP    EXIT
```

If generation is enabled, we jump to the SING label where we count the time that passed since the last inversion to check if it is time to invert the output again. If the count is not at zero yet, we jump to the UPDATE label.

```
SING:   LD    A,(COUNT)     ; assess the time passed
        DEC   A             ; since the last output inversion
        LD    (COUNT),A
        JP    NZ,UPDATE     ; jump if time has not elapsed
```

If the time count is at zero, we make it start again, but we take the new duration from the PERIOD variable (which may have been changed).

```
        LD    A,(PERIOD)    ; re-initialize the time counter
        LD    (COUNT),A
```

We do the level transition by inverting the bit in position 0 of the SOUND variable, which is then copied to output port PSOUND.

```
        LD      A,(SOUND)       ; invert the bit 0 of the output
        XOR     00000001b
        LD      (SOUND),A       ; save back the new state in memory
        OUT     (PSOUND),A      ; and copy it to the output port
```

As explained at the beginning, if PERIOD were a constant, the frequency in the output would not change. At the next UPDATE label, however, the new value of PERIOD is calculated (if the time has come to do it).

We decrement the TIME variable and, if it is not at 0, we exit the handler without updating PERIOD.

```
UPDATE: LD      A,(TIME)        ; during the generation, assess
        DEC     A               ; if it is time to change the duration
        LD      (TIME),A        ; of the next half-period
        JP      NZ,EXIT         ; exit if it is not
```

If TIME is zeroed, we re-initialize it with time TMAX.

```
        LD      A,(TMAX)        ; otherwise re-initialize the counter
        LD      (TIME),A        ; of the time to change the period
```

So, if TMAX contains the constant TMLONG, PERIOD will be changed less often, giving us the "slow" sound. If TMAX contains the constant TMSHORT, PERIOD will be changed more often, giving us the "fast" sound.

The following instructions produce a progressive increment in the PERIOD variable until it reaches its maximum: PHIGH. When it gets to PHIGH, PERIOD is re-initialized to the minimum value: PLOW.

```
        LD      A,(PERIOD)      ; assess the duration of the half-period
        CP      PHIGH           ; check if it has reached the maximum
        JP      NZ,INCP         ; jump if it is not
        LD      A,PLOW          ; otherwise restart the count from the
        LD      (PERIOD),A      ; minimum half-period
        JP      EXIT
INCP:   INC     A               ; increment the half-period by one
        LD      (PERIOD),A      ; save back the new value
```

Finally, we exit the handler and return to the interrupted program.

```
EXIT:   POP     AF              ; restore the saved registers
        EI                      ; re-enable interrupts
        RET                     ; return to the interrupted program
```

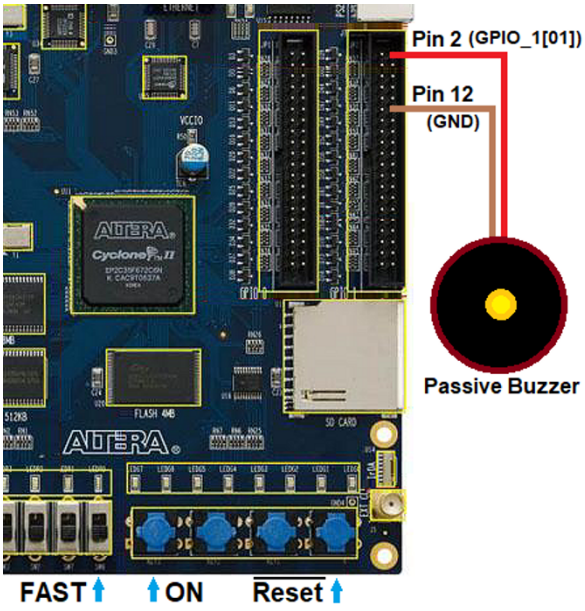
5.6.3.2 Implementation on FPGA

The following sections have figures that summarize the connections chosen for each board[37] used to test the system.

[37] For more information on the associations, see the online content for this section.

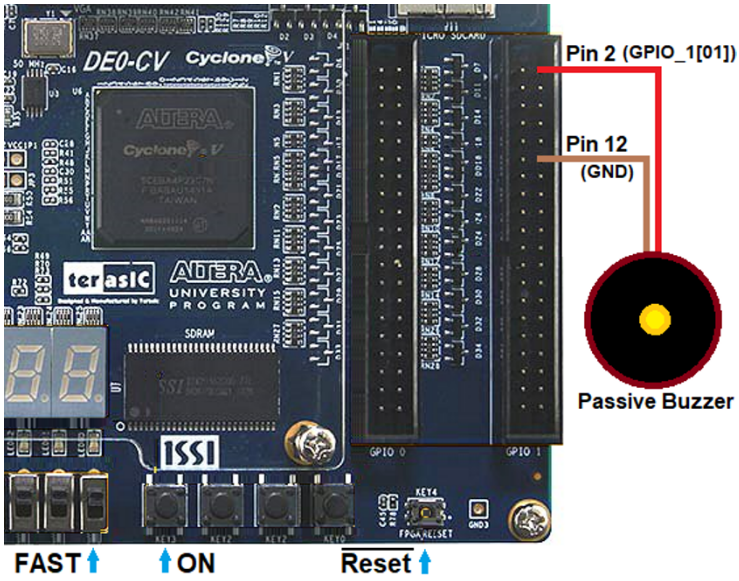
The DE2 board

The following figure shows the devices chosen for the DE2 board.



The DE0CV board

The assignment of devices for the DE0-CV board is very similar to that set for the DE2.

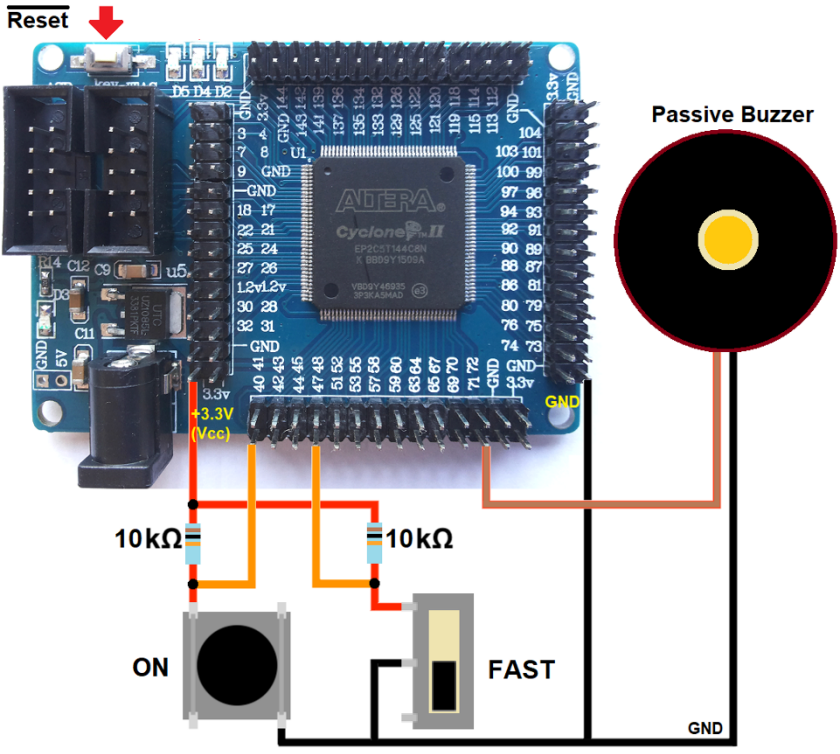


The EP2C5 board

The connections on the EP2C5 board require the connection of an external push-button (ON) and switch (FAST).
We will use the push-button already on the board for reset. The table below shows the summary of the connections as generated by Deeds.

In/Out Name	VHDL Name	Board Resource	Aux. Info
Inputs:			
Ck of ("Soun	—	Clock: 10 MHz	
!Reset	inReset	Push-Button: Key0	Low (if pressed)
FAST	iFAST	Header 0: P1 [IO_47]	
ON	iON	Header 0: P1 [IO_40]	
Outputs:			
!Int7	onInt7		
SOUND	oSOUND	Header 0: P1 [IO_71]	

Below is a photograph of the board with the physical connections to make for the ON button and the FAST switch. The connections of the 10 KΩ pull up resistors are highlighted in red and orange.



5.6.4 Music box

We need to build a music box that can cyclically play a short tune. The traditional music box is mechanical, box-shaped and decorated, and plays when the cover is lifted.

In our prototype, we will use the system reset button to simulate this behavior. We will make it work in the opposite sense from the usual. The reset will then be active with the push-button at rest. For us, "lift the cover" means pressing the reset button, and for this purpose, 'reset' should be called 'PLAY'.

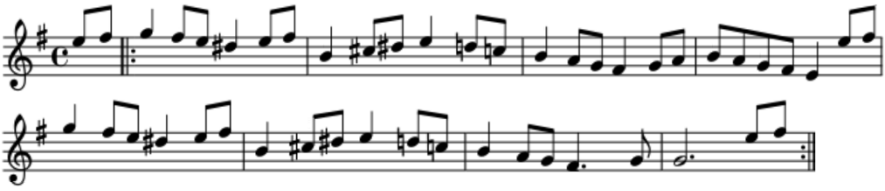
For the transducer, we can use the same piezoelectric device described in Section 5.6.3. Then we can connect it directly to the circuit with no need for an amplifier.

A horn loudspeaker like the one photographed at the left is a potential alternative. It functions along the same principle.

Like most piezoelectric acoustic transducers, it can be directly connected to the logic circuit.

It is affordable, easy to repair, and produces a decidedly higher volume and better sound quality.

Among all the tunes that could be played, we chose a part of Bourrée in E minor (BWV 996) from the great composer J.S.Bach[38]. Below is the musical notation for the first 8 measures, which will repeat cyclically.



To make it possible to set some variations to the sound, we have added three switches in our system. Two of the switches (OCT1 and OCT0) allow us to "transpose" the execution an "octave".

From a Physics perspective, this means that the note's frequency f_N is multiplied by a certain factor in function of the setting of OCT1 and OCT0 (see the table below).

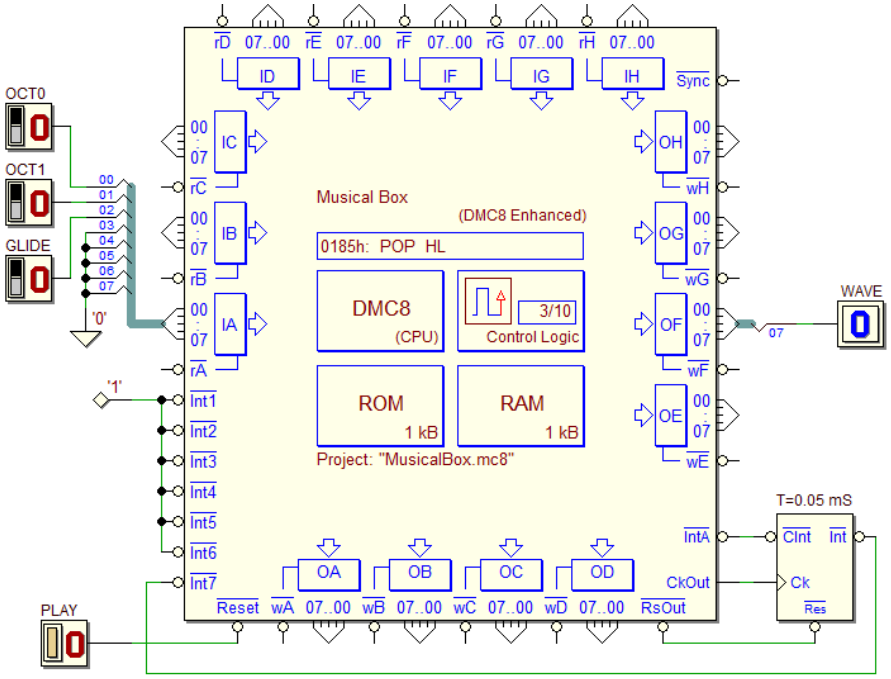
OCT1, OCT0	Frequency	Transposition
0 0	f_N	None
0 1	$f_N \cdot 2$	One octave above
1 0	$f_N \cdot 4$	Two octaves above
1 1	$f_N \cdot 8$	Three octaves above

[38] Johann Sebastian Bach (1685-1750) was a German composer and musician of the Baroque period. Originally composed for the lute, this song was covered in 1969 by Jethro Tull on the album "Stand Up".

The third switch (GLIDE) allows us to give a special touch to the tune by adding a "glide" effect between the notes. The glide between two consecutive notes consists in progressively raising or lowering the frequency from the first note to the second[39].

5.6.4.1 The system

The figure below shows the system, which uses a "DMC8 Enhanced Microcomputer" with an added timer (that interrupts every 50 μS) for the OCT0, OCT1 and GLIDE switches. The sound is generated on the output WAVE when the PLAY button is pressed, as described before.



Generating notes

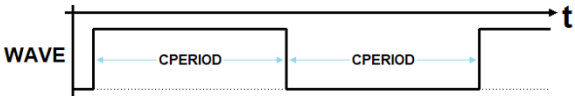
Before examining the assembly code, it is useful to describe the principle of operation behind the generation of notes. We'll approach the subject in steps, ignoring for the moment the specifications for the glide and transposing the octave.

Generating a note in our music box means producing a square wave signal, i.e., a two-level periodic wave on the WAVE output line.

[39] This can be done with the voice or various types of instruments (synthesizers, for example).

The notes are differentiated on the basis of their "pitch". In physical terms this means we simply need to control their frequency of oscillation[40].

Here, it makes sense to assess the half-period of the signal rather than the frequency because at the end of the half-period, we will have to invert the signal logical level on the output. We use a counter (COUNT), which is incremented by one every time the timer is called. When COUNT is equal to the half-period set in the CPERIOD variable, we invert the WAVE output and go back to counting from the top (see the following figure).



We should then define a table of constants, one for every note to generate (FTABLE). For a certain note, we consult the table and load the value of the desired half-period in the CPERIOD variable. The duration of the half-period T_H corresponding to the note's frequency f_N is:

$$T_H = \frac{1}{2 \cdot f_N}$$

Keeping in mind that here, time is marked by the interrupts (that come every $\Delta T = 50\,\mu S$), the number N_H to insert in the table for every note, is calculated by the following expression:

$$N_H = \frac{1}{2 \cdot \Delta T \cdot f_N}$$

Then we round off the result to the nearest integer number. To get a better approximation of the frequency generation, and thus an optimal pitch, we clearly need to reduce ΔT, but for our music box, the timer's $50\,\mu S$ are enough for an acceptable sound.

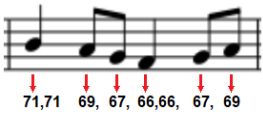
Reading the music

The sequence of notes to generate is read on the "musical score", which is just another table (MSCORE). MSCORE shows the codes of the notes that have to be executed one after the other.

[40] The frequency of notes is not universal but depends on the culture, the musical system and the historical period it is developed in. Just to cite a couple examples, within Europe, the Ancient Greeks tuned their instruments differently from those in the Baroque period, who in turn tuned them differently from the musicians of today. In modern "equal temperament" (introduced after the Baroque period), note frequencies are calculated starting from a reference point (the "central A note", generally defined as $440\,Hz$) and deriving the others by multiplying by a factor of $\sqrt[12]{2}$ (about 1.059). For example, the frequency of Bb, the note after A is $(440 \cdot \sqrt[12]{2}) \simeq 466.164$. Moving on to the note-by-note calculation, we do 12 multiplications and then get the frequency of A at the octave above, which is $880\,Hz$, simply double of the A at the beginning, since$(\sqrt[12]{2})^{12} = 2$.

The codes to insert in the table were chosen according to the numbering defined by the MIDI standard[41], which identifies notes with numbers from 1 to 127. The lowest note we generate is fourth octave C, which corresponds to MIDI 60. Then, to index the FTABLE half-period table as of 0 (whose first line corresponds to our note), we need to subtract 60 from the code that we get by reading MSCORE.

The MSCORE table is read in a timed fashion; the new note is read when the previous one 's time elapses. The notes are divided into "eighths" i.e., one eighth of the time of a whole "measure". Let's look below at the figure of a part of the score. We will insert two identical codes (=71) one after the other for note B, which must last for a quarter measure, then the codes for the notes A (=69), G (=67), and then two identical codes for $F_{\#}$ (=66), for a quarter measure, and so on.



In the program, a one eighth note duration is assessed by counting the interrupts and is defined by the NOTETIME constant (the larger it is, the slower the execution and vice versa). When a note must last more than an eighth on the score, multiple identical codes are inserted consecutively in the table.

Octave transposition

Now let's add the specification for octave transposition. If the time interval between one interrupt and another were a lot shorter, we would be able to afford to extend the table of half-periods FTABLE.

Here though, if we included three higher octaves (36 notes), we would be forced to insert the values of the progressively smaller half-periods, which become ever more approximated in proportion. This would cause a drastic decline in pitch quality because the resulting frequencies would be too discordant from the nominal frequencies.

An acceptable solution from a musical perspective consists in a multiplication parameter (O_{ct} = 1, 2, 4 or 8) for the expression:

$$N_H = O_{ct} \cdot \frac{1}{2 \cdot \Delta T \cdot f_N} \ ,$$

so as to obtain higher frequencies with the same N_H in the denominator:

$$f_N = O_{ct} \cdot \frac{1}{2 \cdot \Delta T \cdot N_H} \ .$$

We can obtain this result by changing the way in which we calculate the half-period.

[41] Musical Instrument Digital Interface, `https://www.midi.org/`

Each time the interrupt handler is called, we continue to increment the counter (COUNT), but by the amount dictated by the parameter O_{ct} (OCTAVE, in the program). When COUNT is greater than or equal to the half-period set in the CPERIOD variable, we invert the WAVE output and instead of zeroing COUNT, we make it equal to the difference (COUNT - CPERIOD).

This means we bring COUNT back to zero after (CPERIOD $\cdot$ O_{ct}) interrupt calls. This gives us a half-period only more or less what we want but that manages the pitch acceptably for our application.

The glide

Finally, we now add the glide between two consecutive notes. We need to take the CPERIOD variable (the current half-period used to calculate transition times) and place another PERIOD variable next to it, the "desired" half-period, that is, that of the next note.

Every time we take a new note from the MSCORE table, the corresponding half-period is calculated in PERIOD. If glide is enabled, rather than assigning the content of PERIOD directly to CPERIOD, we gradually increment (or decrement) CPERIOD in a time-controlled way until it reaches the value set in PERIOD. The resulting effect is that the frequency gradually shifts from one note to the next. As we will see, we can change the timing of this shift by changing the value of a constant.

The program

In the beginning, we define the input and output ports (PCTRL and PWAVE).

```
PCTRL     EQU   00h          ; IA input port: control inputs
PWAVE     EQU   05h          ; OF output port: square wave output
```

There are many variables so it is better to discuss them where they are used.

```
COUNT     EQU   0FC00h       ; half-period time counter
PERIOD    EQU   0FC01h       ; nominal duration of the half-period
CPERIOD   EQU   0FC02h       ; current duration of the half-period
WAVE      EQU   0FC03h       ; output's state
SINDEX    EQU   0FC04h       ; index in the musical score
CNOTE     EQU   0FC05h       ; current note in execution
OCTAVE    EQU   0FC06h       ; octave transposition
TIME      EQU   0FC07h       ; time count (16 bit)
GLITIME   EQU   0FC09h       ; glide duration counter
GLIDEON   EQU   0FC0Ah       ; glide mode On/Off flag
```

The NOTETIME constant determines the duration of an eighth note. The GLIDESET constant defines the duration of the glide between the notes (the shorter it is the less it is perceived).

```
NOTETIME  EQU   3700         ; duration of an eighth note
GLIDESET  EQU   90           ; duration of the glide
```

These constants can be changed as desired. What follow are the definitions of the jumps to the start of the program and the interrupt handler.

```
        ORG     0000h
        JP      START
        ORG     0038h
        JP      HINT
```

We allocate the FTABLE table in the ROM area that comes before the main program. As explained before, FTABLE contains the durations of the half-periods corresponding to each note in terms of units of time ($50\,\mu S$, the interval defined by the timer).

The comments of each line have the names of the notes, their nominal frequencies and their MIDI codes.

```
        ORG     00C0h           ; Note frequency table
FTABLE: DB      153             ; C4 = 261.626 Hz (MIDI: 60)
        DB      144             ; C#4 = 277.183 Hz (MIDI: 61)
        DB      136             ; D4 = 293.665 Hz (MIDI: 62)
        DB      129             ; Eb4 = 311.127 Hz (MIDI: 63)
        DB      121             ; E4 = 329.628 Hz (MIDI: 64)
        DB      115             ; F4 = 349.228 Hz (MIDI: 65)
        DB      108             ; F#4 = 369.994 Hz (MIDI: 66)
        DB      102             ; G4 = 391.995 Hz (MIDI: 67)
        DB      96              ; Ab4 = 415.305 Hz (MIDI: 68)
        DB      91              ; A4 = 440.000 Hz (MIDI: 69)
        DB      86              ; Bb4 = 466.164 Hz (MIDI: 70)
        DB      81              ; B4 = 493.883 Hz (MIDI: 71)
        DB      76              ; C5 = 523.251 Hz (MIDI: 72)
        DB      72              ; C#5 = 554.365 Hz (MIDI: 73)
        DB      68              ; D5 = 587.330 Hz (MIDI: 74)
        DB      64              ; Eb5 = 622.254 Hz (MIDI: 75)
        DB      61              ; E5 = 659.255 Hz (MIDI: 76)
        DB      57              ; F5 = 698.457 Hz (MIDI: 77)
        DB      54              ; F#5 = 739.989 Hz (MIDI: 78)
        DB      51              ; G5 = 783.991 Hz (MIDI: 79)
        DB      48              ; Ab5 = 830.609 Hz (MIDI: 80)
        DB      91              ; A5 = 880.000 Hz (MIDI: 81)
        DB      43              ; Bb5 = 932.328 Hz (MIDI: 82)
        DB      40              ; B5 = 987.767 Hz (MIDI: 83)
        DB      38              ; C6 =1046.502 Hz (MIDI: 84)
```

First of all, the main program initializes the Stack Pointer.

```
        ORG     0100h
START:  LD      SP,0FFFFh       ; initialize the Stack Pointer
```

Then it reads the first note found in the "score" (the MSCORE table, which we see at the end of the code), sets the execution and immediately after, calls the GPERIOD subprogram to take the corresponding half-period from the FTABLE table.

```
        LD      A,(MSCORE)      ; get the first MIDI note from the score
        LD      (CNOTE),A       ; and save it as current note
        CALL    GPERIOD         ; get the corresponding half-period
        LD      (CPERIOD),A     ; copy it to the current duration variable
        LD      A,00h           ; zero the counter of the half-period
        LD      (COUNT),A
```

We also zero output port PWAVE, its software copy WAVE, the read index for the notes from the MSCORE table and the octave transposition parameter (OCTAVE).

```
        LD      (WAVE),A        ; zero the wave output port
        OUT     (PWAVE),A
        LD      (SINDEX),A      ; zero the read index for the notes and
        LD      (OCTAVE),A      ; the octave transposition parameter
```

Before entering the main loop, we initialize the parameters for the glide and the metronome. After, we enable interrupts.

```
        LD      A,GLIDESET      ; initialize the glide duration counter
        LD      (GLITIME),A
        LD      A,0             ; set glide mode OFF
        LD      (GLIDEON),A
        LD      HL,1            ; initialize the metronome glide counter
        LD      (TIME),HL
        EI                      ; enable interrupts
```

The main loop continually acquires the switches (this type of application doesn't necessarily require a debouncing check so it has been omitted for simplicity's sake).

Based on the state of the switches, it first defines the GLIDEON variable, which determines if the notes have been generated with or without a glide effect.

```
MAIN:   IN      A,(PCTRL)       ; read the input switches' state
        LD      C,A             ; and copy it to register C
        AND     00000100b       ; assess the glide control: glide mode
        LD      (GLIDEON),A     ; is ON if GLIDEON is not zero
```

Then, based on the configuration of the OCT1 and OCT0 switches, we load 1, 2, 4 or 8 in the OCTAVE variable.

```
        LD      A,C             ; get the OCT1 and OCT0 bits from C,
        AND     00000011b       ; encode them in a number ranging
        INC     A               ; from 1 to 4 and copy it to register B
        LD      B,A             ;                                   (cont.)
```

```
        LD    A,00000001b    ; calculate (as power of 2) the octave
POWER:  DEC   B              ; transposition parameter
        JP    Z,SAVE         ; when finished, jump and save it
        SLA   A              ; multiply register A by two as many
        JP    POWER          ; times as specified by register B
SAVE:   LD    (OCTAVE),A     ; save the octave transposition parameter
        JP    MAIN           ; and repeat the main loop
```

The interrupt handler is called every $50\,\mu S$. As mentioned before, the shortest time possible was chosen to get the best approximation of the frequencies generated. We can verify, even with time simulation, that all the possible handler sequences are executed correctly within the time frame of $50\,\mu S$.

In order to understand the algorithms in the interrupt handler, it is important to note that it does different tasks all with the goal of generating sound.

These tasks produce results that are stored in variables to be then read and used by other modules, not necessarily written in subsequent order. In some cases, the results are not immediately used by a module executed right in the same interrupt call, but rather they are destined for a module that will be executed during the next interrupt.

At the start of the handler, we save the contents of the registers in use.

```
HINT:   PUSH  AF             ; save the registers in use
        PUSH  HL
        PUSH  BC
```

In the first part of the code, we check if a glide between notes is desired or not. Then we choose either the NOGLIDE, or the GLIDE subprogram. The NOGLIDE subprogram immediately assigns the half-period required by the current note for the generator. GLIDE, on the other hand, calculates the progressive approach from the current half-period to the half-period of the required note, little by little (as we will see further on).

```
        LD    A,(GLIDEON)    ; check if glide is enabled
        OR    A
        CALL  Z,NOGLIDE      ; OFF: assign directly the half-period
        CALL  NZ,GLIDE       ; ON : approach the desired half-period
```

The function IsBEAT tells us if it is time to take the next note from the score (it assesses if the time of one eighth note has elapsed, but the details will be addressed in the following pages).

```
        CALL  IsBEAT         ; is it time to read a new note?
        JP    NZ,NONEW       ; jump if it is not, otherwise
        CALL  NEXTNOTE       ; get the next note code and convert
        CALL  GPERIOD        ; it in the corresponding half-period
```

If it is time, we read the next note on the score and retrieve the corresponding half-period (this is an example of parameters that are not used immediately but will be used at the next call).

Whether a new note is taken or not, we check if the current code is a "rest note" (00h). If it is, we exit the handler because we do not generate transitions on the output during a rest note.

```
NONEW:  LD   A,(CNOTE)   ; check the current note code,
        OR   A           ; if we are during a rest note, exit
        JP   Z,EXIT
```

If it is not a rest note, we continue; we call the subprogram that generates the transitions of the output square wave (described a bit further on).

```
        CALL GENERATE    ; call the output transition generator
```

Then the handler restores the content of the registers, re-enables the interrupts and goes back to the interrupted program.

```
EXIT:   POP  BC          ; restore the saved registers
        POP  HL
        POP  AF
        EI               ; re-enable interrupts
        RET              ; return to the interrupted program
```

Now let's look at the subprograms that the interrupt handler calls.

The IsBEAT function counts time and if an eighth note has gone by, it gives authorization to take the next note to the calling program.

Specifically, it decrements the TIME variable (every $50\,\mu S$), and exits if the count is not zeroed. If it is zeroed, it reloads the NOTETIME constant[42] in the TIME counter and exits with the zero flag active.

```
IsBEAT: LD   HL,(TIME)     ; assess the increment of time
        DEC  HL            ; (by steps of 50 microseconds)
        LD   (TIME),HL
        LD   A,H
        OR   L             ; exit if it's not time to read the next note
        RET  NZ            ; on the score, otherwise
        LD   HL,NOTETIME   ; the time of an eight note has passed,
        LD   (TIME),HL     ; re-initialize the time counter
        RET                ; and return to the calling program
```

The NOGLIDE subprogram is called when we need to execute notes without glides. It simply copies the PERIOD half-period, (which had been retrieved before from the FTABLE table), directly to the CPERIOD variable, which will be used by the generator.

```
NOGLIDE: LD  A,(PERIOD)    ; use directly the requested
         LD  (CPERIOD),A   ; half-period value, without gliding
         RET
```

[42] To execute the piece more quickly we need to reduce the NOTETIME constant.

The GLIDE subprogram produces the glide by gradually moving the content of CPERIOD closer and closer to the desired half-period contained in PERIOD. As we have seen before, the note frequency will take a certain amount of time to move from one to the next so that the transition is clearly audible.

When we enter GLIDE, we immediately decrement the GLITIME ('glide time') variable, but if it is not zeroed yet, we simply leave the function.

```
GLIDE:  LD    A,(GLITIME)   ; assess the glide time
        DEC   A             ; decrementing the variable GLITIME
        LD    (GLITIME),A   ; exit if it is not time to modify the
        RET   NZ            ; half-period of the generated note
```

If it has been zeroed, we refresh the count by loading the GLIDESET constant in the GLITIME[43] variable.

```
        LD    A,GLIDESET    ; otherwise, re-initialize GLITIME
        LD    (GLITIME),A   ; to be able to restart the count
```

We check to see if the contents of the two variables are already equal, and if they are, we exit (the goal has been reached).

```
        LD    A,(CPERIOD)   ; copy the current half-period in B
        LD    B,A
        LD    A,(PERIOD)    ; get the requested half-period in A
        CP    B             ; compare the two values
        RET   Z             ; exit if they have become equal
```

Otherwise, the current half-period CPERIOD is still different from the requested PERIOD. So, we decide to increment or decrement CPERIOD by one based on the Carry flag in order to move the value closer to PERIOD.

```
        LD    A,B           ; move CPERIOD in A
        JP    C,GLIDEDN     ; jump if PERIOD < CPERIOD
GLIDEUP: INC  A             ; otherwise, increment CPERIOD
        LD    (CPERIOD),A
        RET
GLIDEDN: DEC  A             ; decrement CPERIOD
        LD    (CPERIOD),A
        RET
```

Now, let's examine the code of GENERATE, which inverts the output when it is time to do so. It increments COUNT by the amount in the OCTAVE variable (remember that it can contain 1, 2, 4 or 8). If the number in COUNT is not larger than CPERIOD, it is not time to invert the output yet.

```
GENERATE: LD  A,(CPERIOD)   ; get the current note half-period
        LD    B,A           ; and copy it to B
        LD    A,(COUNT)     ; then, copy the time counter to C
        LD    C,A                                   (segue)
```

[43] We can change the entity of the glide by redefining the GLIDESET constant.

```
        LD      A,(OCTAVE)      ; get the octave transposition (1, 2, 4, 8)
        ADD     A,C             ; add it to the time counter:
        LD      C,A             ; COUNT ← COUNT + OCTAVE
        CP      B               ; the count is larger than CPERIOD?
        JP      NC,INVERT       ; jump to INVERT if it is, otherwise
        LD      (COUNT),A       ; save back the COUNT variable
        RET                     ; and leave the function
```

If COUNT is larger than CPERIOD, we jump to INVERT, complement the state of the output and save a copy of that in the WAVE variable. Before that, however, we refresh the COUNT variable by subtracting CPERIOD from it (as described before).

```
INVERT: SUB     B               ; re-initialize the time counter to
        LD      (COUNT),A       ; the difference (COUNT - CPERIOD)
        LD      A,(WAVE)        ; do a transition on the output WAVE,
        XOR     10000000b       ; inverting the MSB of the software copy
        OUT     (PWAVE),A       ; and coping the new value to the port
        LD      (WAVE),A        ; save back the new port state
        RET
```

Now consider the reading of the musical score, that is, the MSCORE table (shown further on). In the MSCORE table we find:

— The codes for the *note* (MIDI, limited here to the interval of 60..84).
— A code for *rest note* (00h), which is not used in the song chosen here.
— A code for *refrain* (80h), which makes everything start from the top[44].

The table is read by using the SINDEX index, which was zeroed at the start. The index is added to the base address of the table. We obtain an address in the HL register that we use to take the code of the note from the table. The code is saved in the CNOTE variable.

```
NEXTNOTE: LD    HL,MSCORE       ; copy the address of the table in HL
          LD    A,(SINDEX)      ; get the index of the note code to read
          ADD   A,L             ; add it to the table base address,
          LD    L,A             ; to obtain the address of the note code
          JP    NC,GETCODE
          INC   H               ; (handle the carry, if any)
GETCODE:  LD    A,(HL)          ; get the note code from the table
          LD    (CNOTE),A       ; copy it to the CNOTE variable
```

Before incrementing the SINDEX index (in order to handle taking the next note from the table), we check to see if we have read the code for refrain.

```
        CP      80h             ; check if it is a code for refrain
        JP      NZ,NEXTIND      ; jump if it is not
```

[44] This is only in our simplified project. In real musical notation, "refrain" signs would only repeat a part of the song not everything from the top.

If we have read a code for refrain, we have gotten to the end of the music described by the score so we zero the index and jump back to NEXTNOTE, then take the first note in the table and go back to the calling subprogram.

```
        LD      A,0             ; this is a refrain, zero the index
        LD      (SINDEX),A
        JP      NEXTNOTE        ; and jump back to get the first code
```

If we have not read it, we increment the read index of the table and exit.

```
NEXTIND: LD     A,(SINDEX)      ; increment the index for the next time
        INC     A
        LD      (SINDEX),A
        RET                     ; and exit
```

The last subprogram to study is GPERIOD, which translates the code of the note into the value of the corresponding half-period. In GPERIOD, the initial check is to confirm that the code taken is for the rest note, in which case we exit (without producing a value for the half-period).

```
GPERIOD: LD     A,(CNOTE)       ; get the current code and
        OR      A               ; check if it is a code for rest note
        RET     Z               ; exit if it is (no matter the return value)
```

Then we check just to be sure that the code of the note is within the interval that we handle and if it is not, we exit.

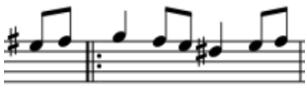
```
        SUB     60              ; calculate the index from the MIDI code
        RET     C               ; exit if the index is < 0 (not valid)
        CP      25              ; (A ≥ 25)? Cy = 1 if the index is < 25
        RET     NC              ; exit if ≥ 25 (not valid)
```

We continue if the index is valid (00..24). Then we add it to the base address of the table, to take the desired half-period.

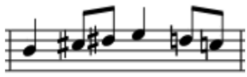
```
        LD      HL,FTABLE       ; get the table base address in HL
        ADD     A,L             ; add the index to that address
        LD      L,A
        LD      A,(HL)          ; get the period
        LD      (PERIOD),A
        RET
```

The following is the entire musical score for the song, which has been transcribed note by note in the MSCORE table.

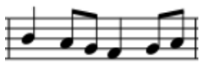
```
MSCORE: DB  76      ; E5        (measure 0)
        DB  78      ; F#5
        DB  79      ; G5        (measure 1)
        DB  79      ; G5
        DB  78      ; F#5
        DB  76      ; E5
        DB  75      ; Eb5
        DB  75      ; Eb5
        DB  76      ; E5
        DB  78      ; F#5
```



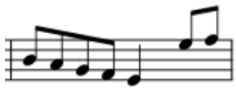
```
DB  71      ; B4      (measure 2)
DB  71      ; B4
DB  73      ; C#5
DB  75      ; Eb5
DB  76      ; E5
DB  76      ; E5
DB  74      ; D5
DB  72      ; C5
```



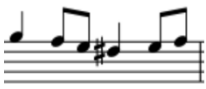
```
DB  71      ; B4      (measure 3)
DB  71      ; B4
DB  69      ; A4
DB  67      ; G4
DB  66      ; F#4
DB  66      ; F#4
DB  67      ; G4
DB  69      ; A4
```



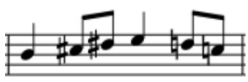
```
DB  71      ; B4      (measure 4)
DB  69      ; A4
DB  67      ; G4
DB  66      ; F#4
DB  64      ; E4
DB  64      ; E4
DB  76      ; E5
DB  78      ; F#5
```



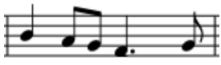
```
DB  79      ; G5      (measure 5)
DB  79      ; G5
DB  78      ; F#5
DB  76      ; E5
DB  75      ; Eb5
DB  75      ; Eb5
DB  76      ; E5
DB  78      ; F#5
```



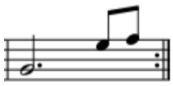
```
DB  71      ; B4      (measure 6)
DB  71      ; B4
DB  73      ; C#5
DB  75      ; Eb5
DB  76      ; E5
DB  76      ; E5
DB  74      ; D5
DB  72      ; C5
```



```
DB  71      ; B4      (measure 7)
DB  71      ; B4
DB  69      ; A4
DB  67      ; G4
DB  66      ; F#4
DB  66      ; F#4
DB  66      ; F#4
DB  69      ; A4
```



```
        DB  67      ; G4        (measure 8)
        DB  67      ; G4
        DB  67      ; G4
        DB  67      ; G4
        DB  67      ; G4
        DB  67      ; G4
        DB  80h     ; code for refrain
```



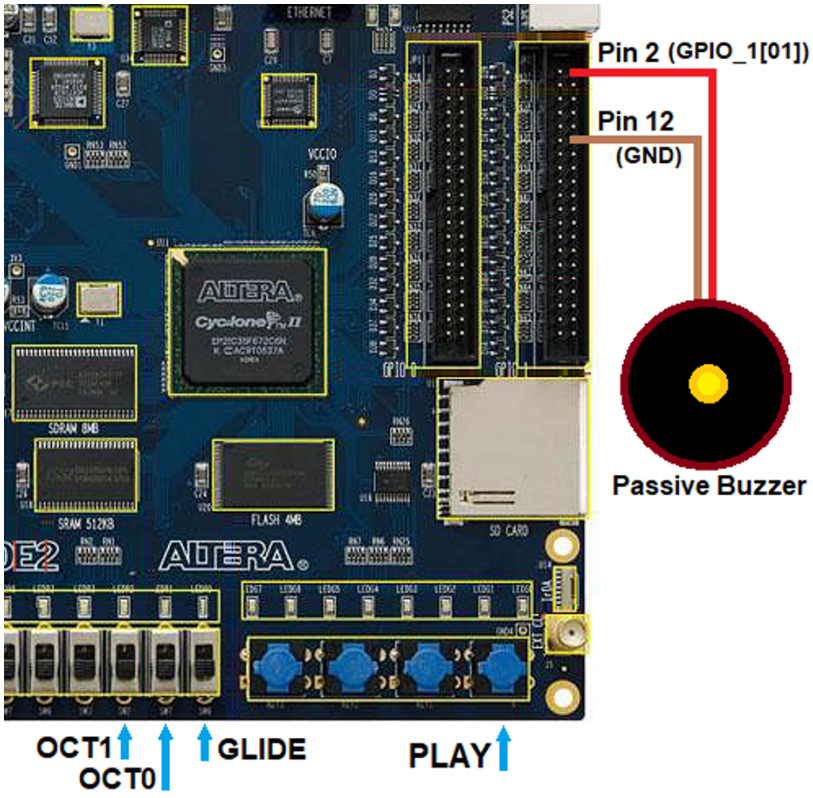
Note: the last two notes (F#5 and G5) are not in the table because they are already at the beginning of the song and will be played when we go back to the top.

5.6.4.2 Implementation on FPGA

The following sections have figures that summarize the connections chosen for each board[45], used to test the system.

The DE2 board

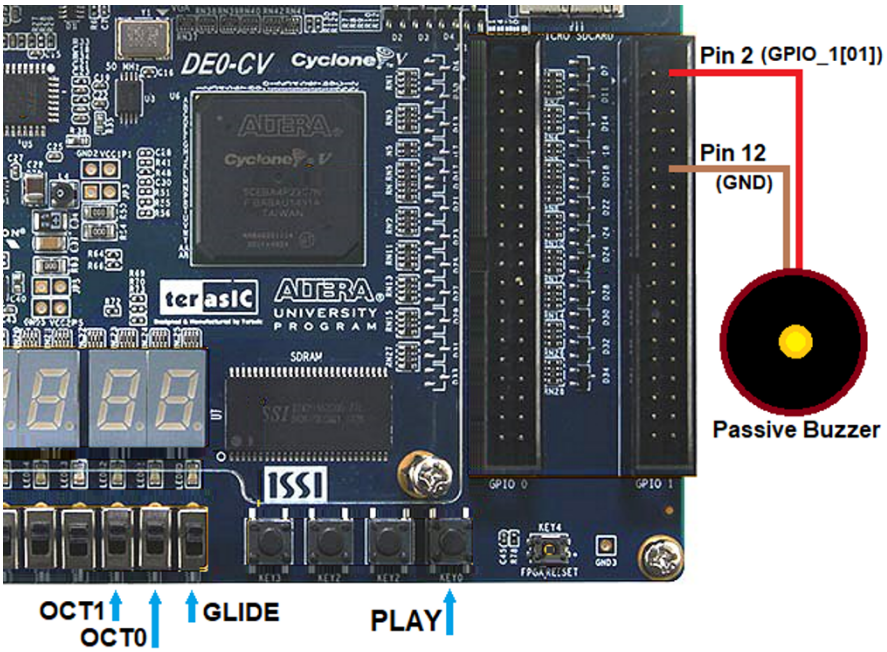
The following figure shows the devices chosen for the DE2 board.



[45] For more information on the associations, see the online content for this section.

The DE0CV board

The assignment of devices for the DE0-CV board is very similar to that set for the DE2.

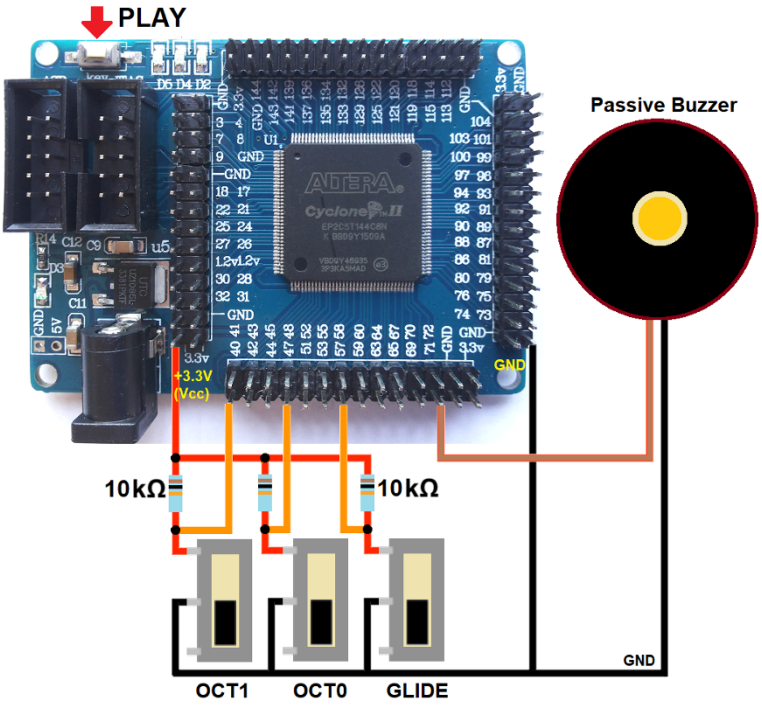


The EP2C5 board

The connections on the EP2C5 board require the external connection of three switches (OCT1, OCT0 and GLIDE). For the PLAY command, however, we will use the one push-button on the board. The table below shows the summary of the connections as generated by Deeds.

In/Out Name	VHDL Name	Board Resource	Aux. Info
Inputs:			
Ck of ("Musi	—	Clock: 10 MHz	
!Reset	inReset		
OCT0	iOCT0	Header 0: P1 [IO_57]	
GLIDE	iGLIDE	Header 0: P1 [IO_57]	
PLAY	iPLAY	Push-Button: Key0	High (if pressed)
Outputs:			
!Int7	onInt7		
WAVE	oWAVE	Header 0: P1 [IO_71]	

Below is a photograph of the board with the physical connections to make for switches OCT1, OCT0 and GLIDE. The connections of the three 10 KΩ pull up resistors are highlighted in red and orange.

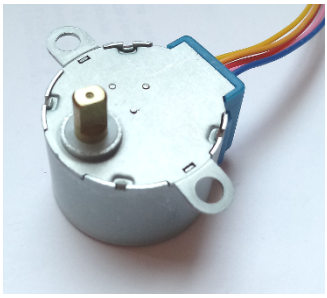


5.6.5 Stepper motor control

In this example, we will build a microprocessor-based system that continuously rotates a "stepper motor". After a brief introduction to stepper motor operations, we will analyze the system and implement it on FPGA.

The stepper motor

Here, we discuss the 28BYJ-48 component[46], shown in the figure on the right. It is easy to find and economical.

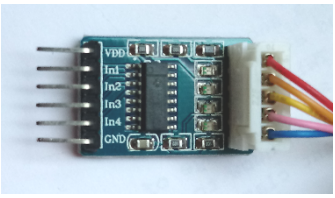


Stepper motors are electro-mechanical components that can execute small rotations of a predefined angle on command.

To move the motor we need electronic circuits whose operating principle goes beyond the scope of this book. We will use a small electronic "power" driver board as an interface between the motor and the output logic lines of the FPGA board we will use.

[46] https://datasheetspdf.com/pdf-file/1006817/Kiatronics/28BYJ-48/1
https://lastminuteengineers.com/28byj48-stepper-motor-arduino-tutorial

On the right, we have a photo of the electronic interface circuit that we will use.

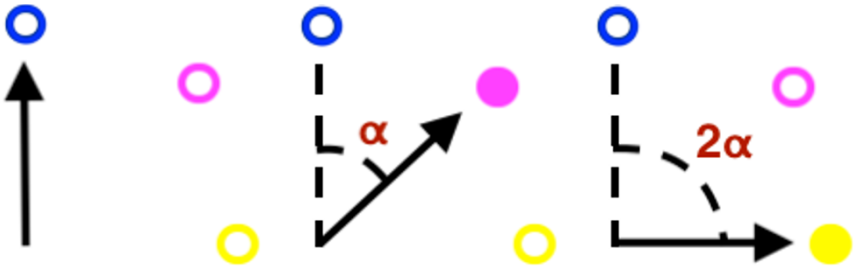


Its job is to allow for driving the lines of the motor, given that they require higher current and tension than those available on the typical outputs of an FPGA component.

Further on, the instructions to connect this board between the chosen FPGA and the lines of the motor will be explained.

The operating principle

On these pages, we will show an abstract model of the motor and focus on the logical and operative aspects. The physical and electro-mechanical aspects will not be dealt with here. We use an arrow to represent the position of the angle of the motor shaft at a given time (see the following figure).



Initial state First rotation Second rotation

In the initial state the arrow is pointing up. There is an electromagnet to the right of the arrow[47] (pink circle) that attracts the arrow when activated. Once the electromagnet is activated, a certain amount of time elapses and the arrow completes the rotation of a certain angle α, whose value depends on the physical parameters of the motor (in the figure, it is represented at 45° just for clarity's sake). Until the electromagnet is active, the arrow will continue to point at it.

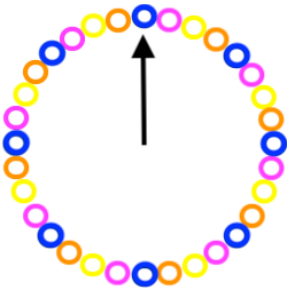
Now we activate another electromagnet (yellow circle in the figure) and make sure to deactivate the first one. As before, we will make the motor rotate again by attracting the arrow, giving us a total rotation of $2 \cdot \alpha$.

If we continue to activate the electromagnets along the circumference we can execute a full stepwise rotation of the motor. We proceed as before: each time, we activate the next electromagnet and deactivate the previous one. Clearly, if we invert the activation order of the electromagnets, we make the motor rotate in the opposite direction.

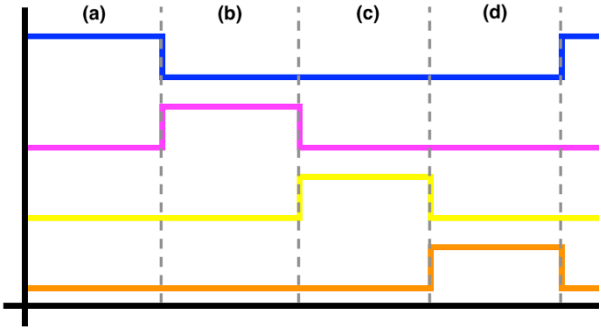
[47] An electromagnet produces an electric field "on command", with the passage of an electric current.

The stepper motor: an abstract model

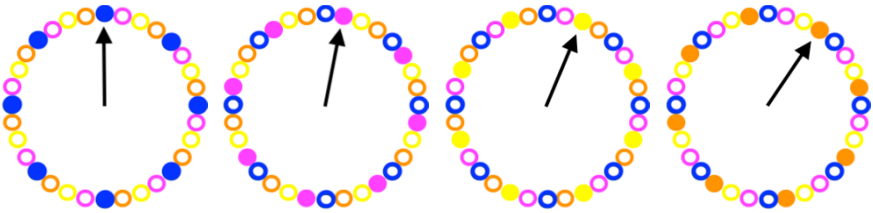
Now let's look at the following figure, an abstract model of the 28BYJ-48 stepper motor. We see 32 electromagnets grouped by color. Four inputs (active high) are available to the software programmer. Each input is connected to a different group of electromagnets (the four groups are represented by the colors blue, pink, yellow and orange).



To rotate the motor we individually activate the groups of electromagnets one after the other in the desired direction.



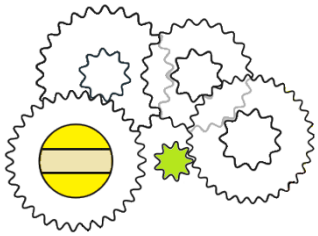
When we activate the blue, pink, yellow and orange electromagnets in sequence, the arrow rotates clockwise. When we change the sequence to orange, yellow, pink and blue, the arrow rotates counter-clockwise.



(a) Initial state (b) First rotation (c) Second rotation (d) Third rotation

In the operating mode described, we see that each step corresponds to a rotation of 11.25 degrees (360/32) of the motor tree.

The component has a mechanical reduction gear set (as sketched in the figure on the right).



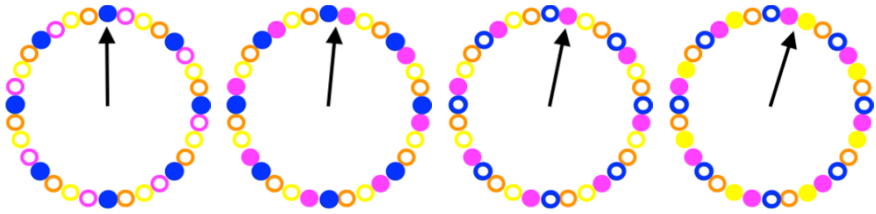
The gear on the motor shaft is shown in green. The gear set reduces the rotation by a ratio of about 1:64.

Each time a new electromagnet is activated in the sequence described, the external axis (shown in yellow) executes a rotation of 0.18 degrees downstream of the gear ratio.

Three different stepper motor control modes

We have already looked at the simplest control mode for the stepper motor. There are (at least) two other ways to drive this component.

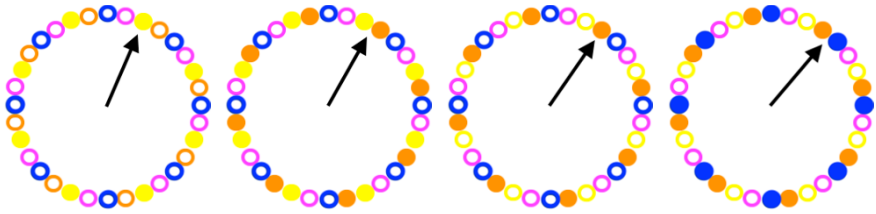
One makes it possible to rotate the arrow by an angle half the standard step (that is 0.09, taking into account the gear set) degrees, by activating not only the electromagnet at the right of the arrow but also the one on the left. This way, the arrow will be attracted between the two electromagnets, passing from (a) to (b).



(a) Initial state (b) First rotation (c) Second rotation (d) Third rotation

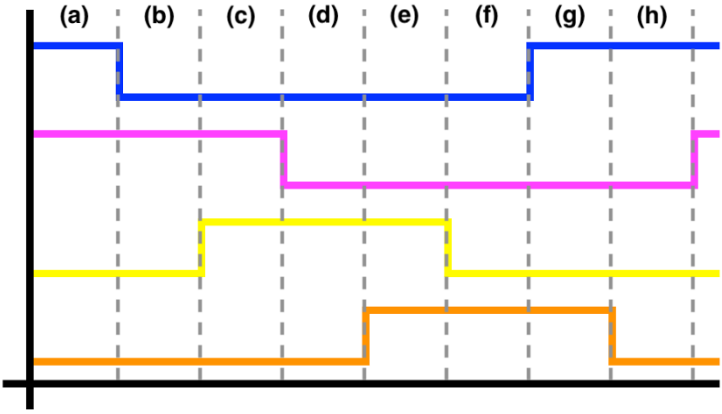
Then by deactivating the one on the left and leaving the one on the right active (c) we can execute a second 0.09 degree rotation. This way we get a complete step rotation, that is 0.18 degrees and we have doubled the motor's precision without changing the mechanics.

By continuing as shown in (d), (e), (f), (g), (h) and so on, we can make a whole round angle by 0.09 degree steps.

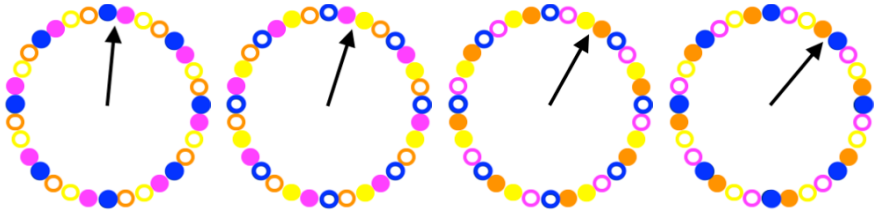


(e) Fourth rotation (f) Fifth rotation (g) Sixth rotation (h) Seventh rotation

So, the sequence to activate for a clockwise rotation is the following:

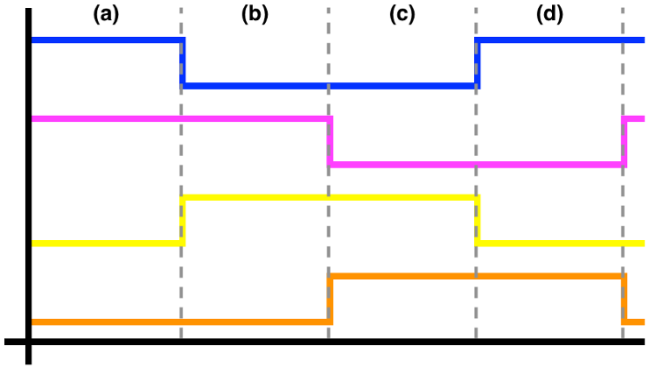


The other driving mode allows us to raise the angular momentum generated by the motor by activating not just the electromagnet closest to the arrow but also the next one, as shown in the following figure. It increases the strength of the attraction on the arrow.



(a) Initial state | (b) First rotation | (c) Second rotation | (d) Third rotation

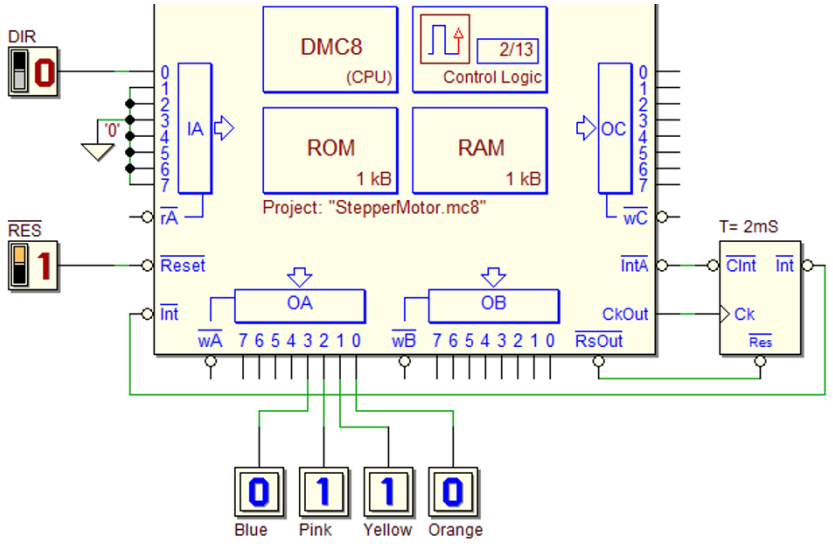
What follows is the timing diagram for a clockwise rotation.



This mode will be used in the following programming example.

5.6.5.1 The system

The control system for the stepper motor is based on the "DMC8 Microcomputer" component (see the following figure). Pin 0 of port IA is connected to switch DIR, which can control the motor's direction of rotation. If it is set at '1' it is counter-clockwise but clockwise if it is at '0'.



Output port OA drives the motor. Specifically, the blue, pink, yellow and orange control wires are driven by port bits 3, 2, 1 and 0, respectively. A timer activates interrupt requests $\overline{\text{Int}}$ every 2 *mS*. When the request is accepted a pulse on output $\overline{\text{IntA}}$ automatically deactivates line $\overline{\text{Int}}$.

Once it has executed the necessary initializations after system reset, the main program enters an infinite loop where it reads input port IA and memorizes the value in the DIR variable, which is read by the interrupt handler.

The interrupt handler updates the value of the blue, pink, yellow and orange commands in line with what is memorized in the DIR variable, which is updated by the main program. For every call, it must show a new value on output port OA for the sequence of commands needed to rotate the motor in the direction required by the DIR input.

The program

We declare the input port (CNTRL) and output port (MOTOR).

```
CNTRL    EQU    00h        ; IA input port: direction (bit 0)
MOTOR    EQU    00h        ; OA output port: motor control
```

We define the DIR and ANGLE variables (each one byte sized). The DIR variable stores the direction of rotation required. The use of ANGLE will be explained further on.

```
DIR     EQU   0FC00h      ; direction of rotation
ANGLE   EQU   0FC01h      ; index in the SEQUENCE table
```

We insert the jumps to the start of the program and the interrupt handler.

```
        ORG   0000h
        JP    START
        ORG   0038h
        JP    HINT
        ORG   0100h
```

At the start of the main program, as usual, we initialize the Stack Pointer, the variables used and the output ports. Then before entering the main loop, we enable interrupts.

```
START:  LD    SP,0FFFFh   ; initialize the Stack Pointer
        LD    A,00h       ; zero the variables DIR and ANGLE
        LD    (DIR),A
        LD    (ANGLE),A
        OUT   (MOTOR),A   ; and the output port MOTOR
        EI
```

As mentioned previously, the main loop does nothing more than update the required direction of rotation in the variable DIR.

```
MAIN:   IN    A,(CNTRL)   ; read the input port
        AND   00000001b   ; mask the unused bits
        LD    (DIR),A     ; update the rotation direction
        JP    MAIN
```

The interrupt handler HINT, which is called every 2 mS, updates the control lines of the motor.

The program saves the registers in use on the Stack and then reads the required direction of rotation from the DIR variable and decides whether to increment or decrement an index (ANGLE). As we will see in more detail further on, by incrementing the ANGLE index, we get a clockwise rotation and by decrementing it, we get a counter-clockwise rotation. The count is made cyclical on two bits (so that the value of ANGLE can assume only values from 0 to 3), and the resulting value is saved back in ANGLE.

```
HINT:   PUSH  AF
        LD    A,(DIR)     ; get the rotation direction
        OR    00000000b   ; modify the flags
        LD    A,(ANGLE)
        JP    Z,RIGHT     ; jump, or not, according to the direction
LEFT:   DEC   A           ; counter-clockwise rotation
        JP    UPDATE
RIGHT:  INC   A           ; clockwise rotation
UPDATE: AND   00000011b   ; make the count cyclical (on two bits)
        LD    (ANGLE),A
```

The CONTROL subprogram, analyzed further on, returns to the accumulator the configuration of commands that are needed to rotate the motor, given the ANGLE index in the accumulator. The configuration we get is then transferred to the MOTOR output port. Then we retrieve the saved registers and the handler re-enables the interrupts and returns the control to the interrupted program.

```
        CALL    CONTROL         ; get the motor commands from the table
        OUT     (MOTOR),A       ; write the commands to the driver
        POP     AF
        EI                      ; re-enable interrupts
        RET                     ; return to the interrupted program
```

What follows is the SEQUENCE table, which contains the sequence of commands needed to rotate the stepper motor clockwise, by driving it in the mode that generates the greatest angular momentum. If the table is read in reverse, it can also rotate the motor counter-clockwise.

```
SEQUENCE:   DB      00001100b   ; active commands: blue, pink
            DB      00000110b   ; active commands: pink, yellow
            DB      00000011b   ; active commands: yellow, orange
            DB      00001001b   ; active commands: orange, blue
```

The CONTROL subprogram reads the table. It takes an index passed through register A and returns the desired value back into the same register.

```
CONTROL:    PUSH    HL
            LD      HL,SEQUENCE ; add the index in register A
            ADD     A,L         ; to the table base address
            LD      L,A
            JP      NC,NOCARRY
            INC     H
NOCARRY:    LD      A,(HL)      ; get the desired value in A
            POP     HL
            RET
```

5.6.5.2 Implementation on FPGA

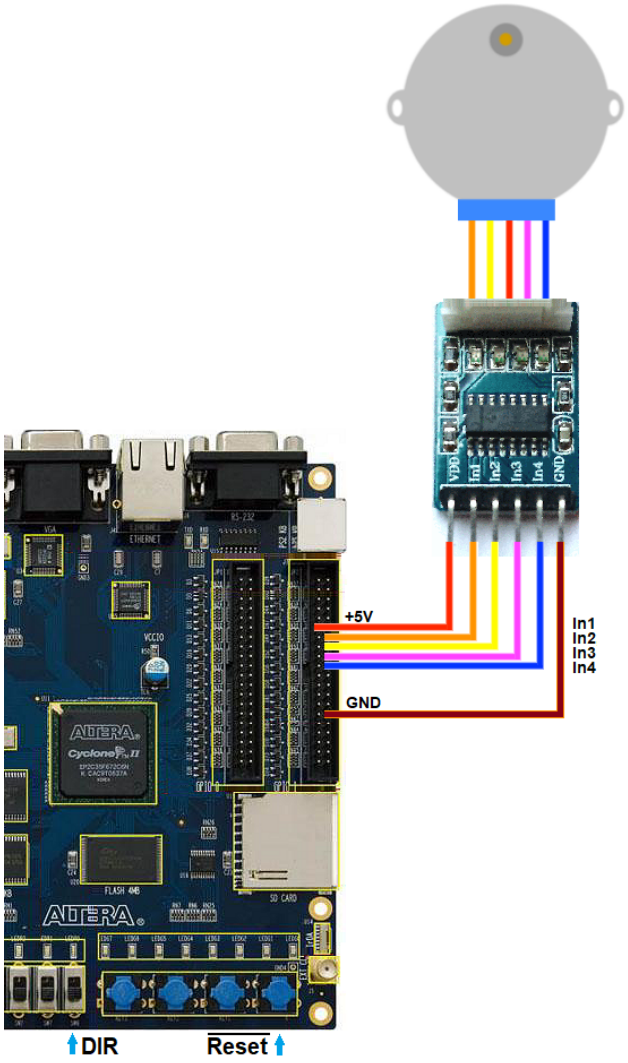
The following subsections have figures that summarize the connections chosen for each board, which are useful for testing the system.

The DE2 board

The following figures show the devices chosen for the DE2 board, both in table form (as generated by Deeds), and in terms of physical connections.

In/Out Name	VHDL Name	Board Resource	Aux. Info
Inputs:			
Ck of f"Micro	—	Clock: 10 MHz	
!RES	inRES	Push-Button: Key[00]	Low (if pressed)
DIR	iDIR	Switch: Sw[00]	
Outputs:			
Orange	oOrange	Header 1: GPIO_1[11]	
Yellow	oYellow	Header 1: GPIO_1[13]	
Blue	oBlue	Header 1: GPIO_1[17]	
Pink	oPink	Header 1: GPIO_1[15]	

Take care to also connect the power supply wires (+5V and GND) of the motor control board, as shown in the following figure.

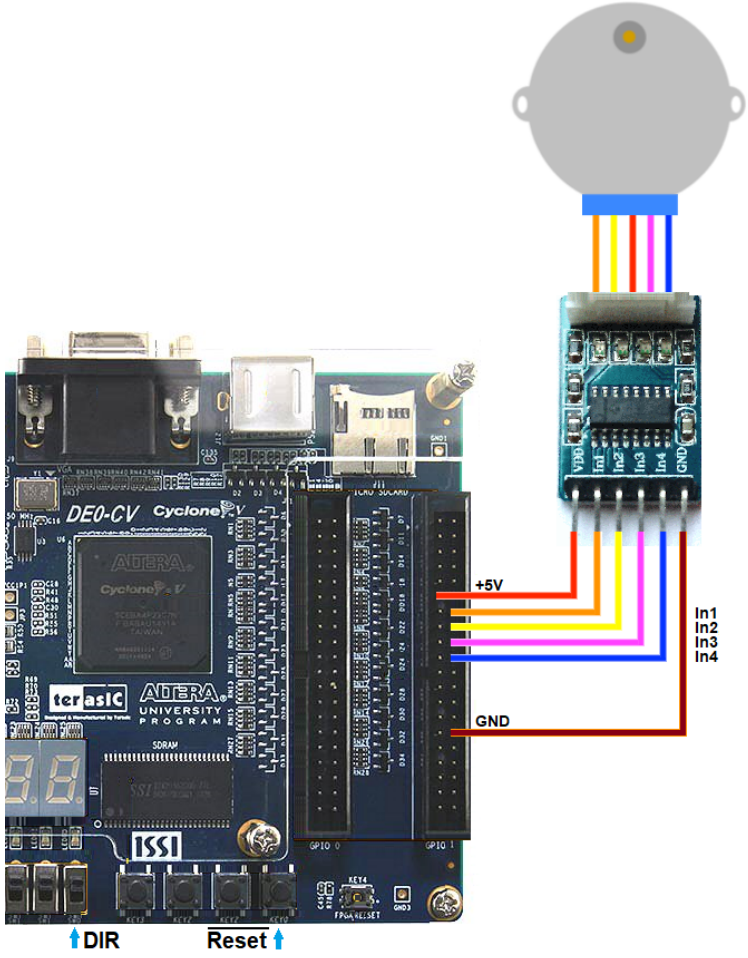


The DE0CV board

The assignment of devices for the DE0-CV board is very similar to that set for the DE2.

In/Out Name	VHDL Name	Board Resource	Aux. Info
Inputs:			
Ck of ("Microd —		Clock: 10 MHz	
!RES	inRES	Push-Button: Key[00]	Low (if pressed)
DIR	iDIR	Switch: Sw[00]	
Outputs:			
Orange	oOrange	Header 1: GPIO_1[11]	
Yellow	oYellow	Header 1: GPIO_1[13]	
Blue	oBlue	Header 1: GPIO_1[17]	
Pink	oPink	Header 1: GPIO_1[15]	

Likewise, pay close attention when connecting the motor control power supply (+5V e GND) as shown in the figure.

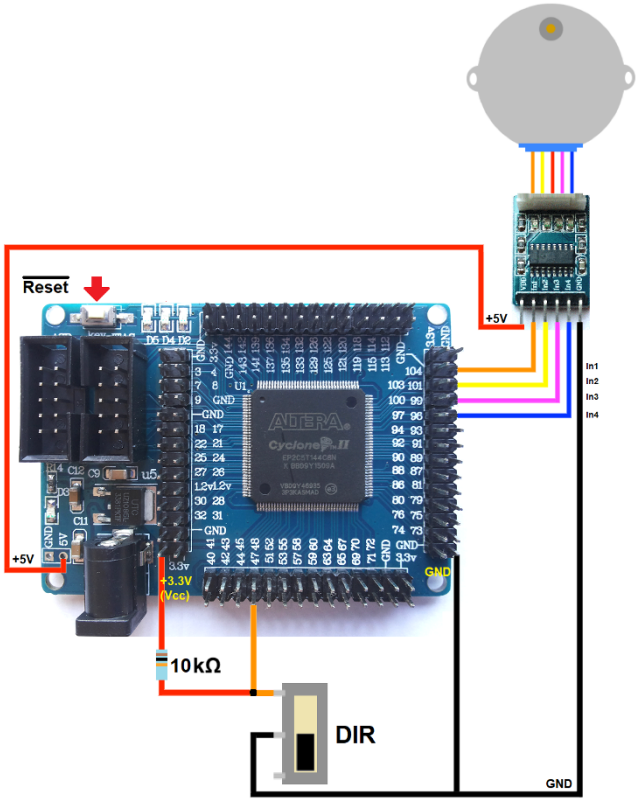


The EP2C5 board

The connections on the EP2C5 board require the connection of an external slide switch to manage the direction (DIR). We will use the only push-button on the board for manual system reset. The following table shows a summary of the connections as generated by Deeds.

In/Out Name	VHDL Name	Board Resource	Aux. Info
Inputs:			
Ck of ("Micrd	—	Clock: 10 MHz	
!RES	inRES	Push-Button: Key0	Low (if pressed)
DIR	iDIR	Header 0: P1 [IO_47]	
Outputs:			
Orange	oOrange	Header 1: P2 [IO_104]	
Yellow	oYellow	Header 1: P2 [IO_101]	
Blue	oBlue	Header 1: P2 [IO_96]	
Pink	oPink	Header 1: P2 [IO_99]	

What follows is the photo of a board with the physical connections to make for the DIR slide switch and the 10KΩ pull up resistors.



The motor's power supply can be connected by using the +5V on the board next to the power supply connector (as shown in the figure) even if there is no soldered pin.

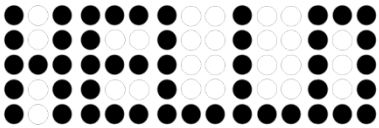
5.6.6 Using a liquid crystal display (LCD)

In this sample project, we will first introduce the interface of the LCD (Liquid Crystal Display) that was mounted on a "vintage" Nokia 5110 mobile phone[48] and many others. Then we will analyze and build a microprocessing system that can show the classic "Hello World" message on this display.

A brief introduction to graphic displays

Any single-color display like the one here, can be understood as an ordered collection of small point sources of light that can be controlled electronically. When they are lit they have only one color depending on their physical characteristics. They are commonly called "pixels" (from "PIcture - ELement"), in that they represent the part of the image that is the smallest and cannot be divided.

By turning some pixels on and others off, we can create any type of visual information (text, drawings, graphs, sheet music, photographs, etc.).



The term pixel does not only refer to single-color images; they can also be multi-colored.

If we put three small lights in red, green and blue (RGB) in the space for one single-color pixel, we get a tri-color display. By regulating the intensity of the three lights, we can represent most of the visible color spectrum.

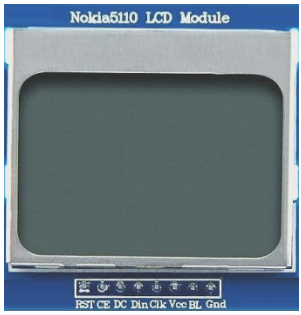
This is the color model currently used for screens. There are others, however, such as the RGBY which adds yellow to the red, green, blue combination, and the CMYK based on cyan, magenta, yellow and black, which is commonly used in printers.

The lights discussed here can be produced with different technologies like LED, plasma and others. The differences come in the physical characteristics of the displays, which have an effect on the range of colors displayed, the luminosity, energy consumption, durability, resistance to mechanical stress... the basic features of an electronic system.

Aside from production technology, the screens differ in the density of pixels per inch (PPI), the higher it is, the less we see the outline of the pixels. Above 300 PPI we cannot see the outline of the pixels at a distance of 10-12 cm. Another difference that is important from the commercial perspective is the width and length of the pixel array.

[48] https://it.wikipedia.org/wiki/Nokia_5110

The Nokia 5110 display



The display used in the Nokia 5110 is single-color and made up of an array of 48 × 84 liquid crystals for a total of 4032 pixels (the component is shown at the right).

The pixel array is managed by an electronic circuit inside the display, designed specifically by the manufacturer to handle all the physical details autonomously. This allows the system designer to deal only with some settings and what is shown on the display.

The electronic circuit in question is an integrated component called the PCD8544[49] made by Philips in the 1990s. It offers programmers a serial interface for handling internal parameters and for sending the content to show on the display. Before analyzing the communication interface in detail, we will discuss the structure of the chip's internal memory.

The structure of the PCD8544 chip's internal memory

Inside the PCD8544 component, we have an 8-bit, 48 x 84 location RAM memory that contains the program for the image on the screen. The figure at the top of the opposite page shows the relation between the position of the RAM's flip-flops and that of the liquid crystals.

Up front, we see the array of liquid crystals. In the background, we see the memory divided into 6 banks (numbered 0 to 5). Each bank corresponds to an 84 pixel-wide, 8 pixel-high strip (in total, 6 strips × 8 pixels = 48 vertical pixels).

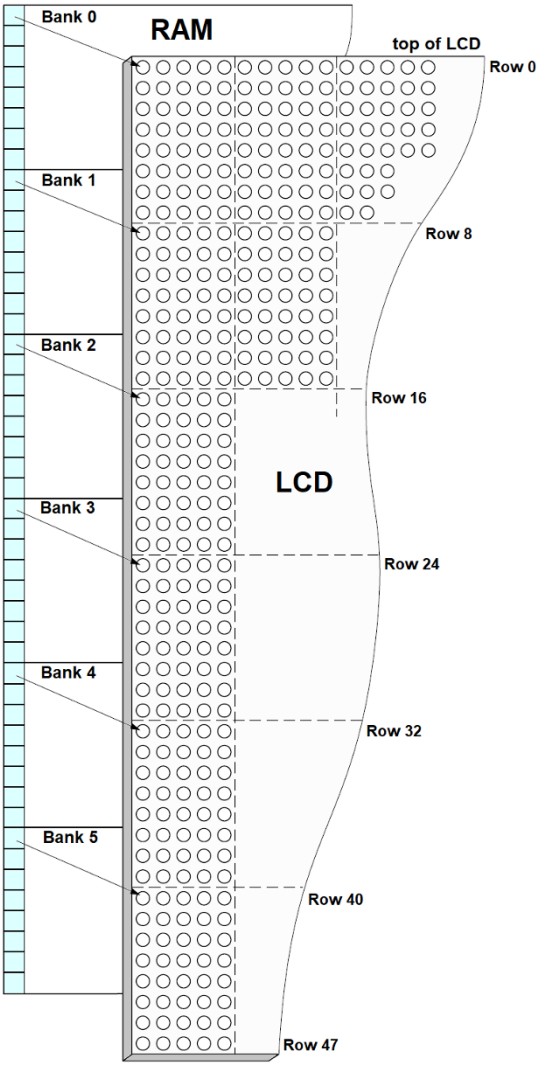
As an example, on the left of the figure, the flip-flops that make up the first location in every bank of the RAM are highlighted. An arrow shows the relation between the position of the first flip-flop of the memory location and the corresponding liquid crystal (a pixel).

Each flip-flop addresses a single liquid crystal. A logical '1' activates the liquid crystal to black. A '0' turns the crystal off. This allows the background color to pass through.

When the system is powered, the RAM contains random values and then needs to be initialized. To write the values in the RAM memory, the chip uses the data received on the serial line. Every byte that is sent writes 8 flip-flops, i.e., one RAM memory location.

The second figure on the same page shows the locations that make up the RAM memory.

[49] Datasheet PCD8544: `https://www.sparkfun.com/datasheets/LCD/Monochrome/Nokia5110.pdf`



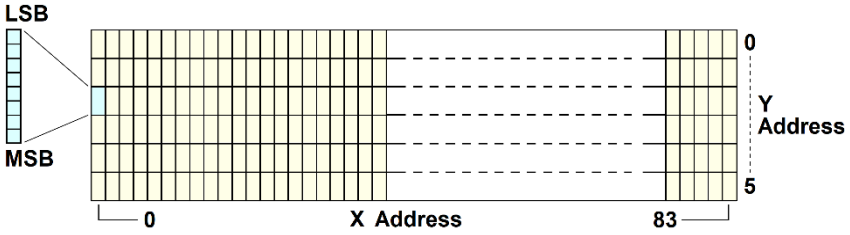
As shown in the figure at the bottom of this page, the RAM is organized into two dimensions (the X axis and Y axis).

The figure below also highlights the order in which the bits are organized in an individual location.

We start with the least significant bit, which is stored in the highest flip-flop and keep going down until we get to the most significant bit at the lowest flip-flop.

Consider again the figure at the left showing the relation of the flip-flop position to the corresponding liquid crystal. We see how the least significant bit of the first RAM location of Bank 0 drives the first liquid crystal of row 0. Continuing down we get to the most significant bit of the first RAM location of Bank 5, which drives the first crystal of row 47.

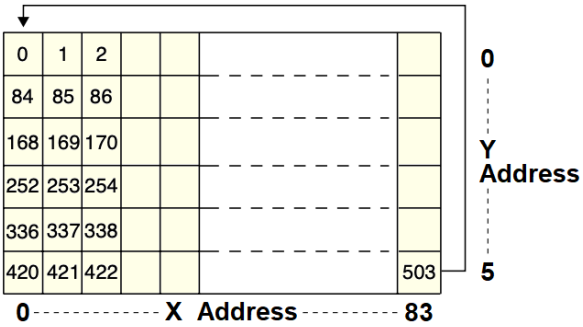
Two internal chip registers store the x and y coordinates of the memory cell currently in use. At system reset, they are both initialized to zero.



These registers can be set manually through the serial interface, although normally they are handled automatically by the chip according to the addressing method that's been chosen.

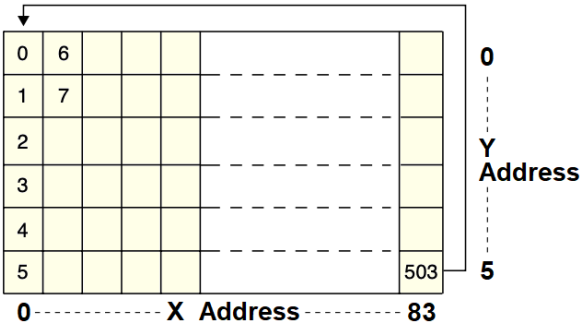
Addressing methods

There are two methods for addressing the RAM: "row addressing" and "column addressing". In row addressing (this will be used in our example and is shown in the following figure), the memory locations are addressed row by row from left to right, as we are used to writing with pen and paper.



After each write operation, the registers are automatically incremented. Once the last row is written, the process restarts cyclically from the beginning.

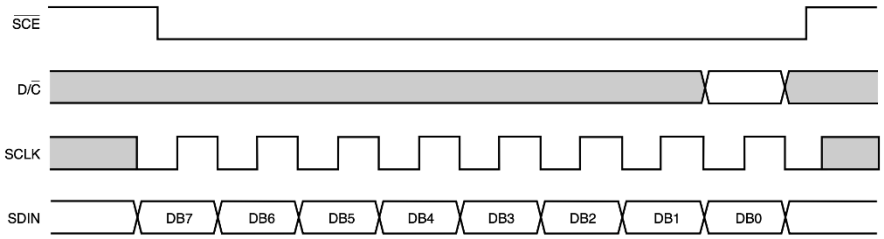
For column addressing (see the following figure), first, all six locations in the first column (having index zero) are written from top to bottom. Then we increment the column and repeat the process. Once the last column is written the process restarts cyclically from the beginning.



Now that we have analyzed the structure of the chip's internal memory, we can see how to interact with it.

Communication interface

The PCD8544 integrated circuit offers the designer a programming interface for the RAM memory based on five connections: SDIN, SCLK, D/$\overline{\text{C}}$, $\overline{\text{SCE}}$ and $\overline{\text{RES}}$. The interface provided requires serial synchronous communication with 8-bit words where the most significant bit is sent first (see the following figure).



The five lines control the serial data (SDIN), the communication clock (SCLK) and the format of the data (D/$\overline{\text{C}}$), the chip enable ($\overline{\text{SCE}}$) and the device reset ($\overline{\text{RES}}$).

Both the control words that act on the internal settings and the data to show on the display can be sent to the chip through the interface. The use of the D/$\overline{\text{C}}$ line allows us to make this distinction.

To start communication with the chip, we need to activate line $\overline{\text{SCE}}$. This enables the action of the SCLK line, the serial communication clock.

The chip acquires the logic value presented on line SDIN at every rising edge of line SCLK. A data packet sent on this serial line is made up of 8 bits, starting from the most significant. The maximum frequency of the communication clock is 4 *MHz*.

The D/$\overline{\text{C}}$ line defines the content of the packet. If D/$\overline{\text{C}}$ is set at '0' while the last bit of the packet (the least significant) is sent, the word will be interpreted as a command. Otherwise the word will be interpreted as information to write on the display.

Since this explanation is introductory, we have chosen not to show the complete set of chip commands. This section only contains the basic commands[50]. Therefore, we will continue with a presentation of a brief display initialization sequence and then with useful instructions for writing the display contents into its RAM memory.

Initializing the display

This section will present a short series of instructions to make the display operative. As explained, we have left some instructions aside, such as those for handling some physical details. Yet, these are not essential for the display to function in closed environments with temperature control.

After we reset the system and activate the $\overline{\text{RES}}$ line for a short time, we can begin to interact with the display.

The first thing to do is set the contrast[51] of the display. If we omit this step, the information displayed might not be visible to the human eye.

[50] For a complete analysis, please refer to the PCD8544 datasheet cited previously.
[51] The contrast is regulated through the tension the liquid crystals work on.

This parameter can assume values from 0 (minimum contrast) to 127 (maximum contrast). To set it, we must send the chip the following command, which orders the chip to pass to the "extended instruction set", used to define the physical parameters of the display.

D/$\overline{C}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	1	0	0	0	0	1

Once in the extended instruction set mode, we can set the contrast by sending the byte below:

D/$\overline{C}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	1	C6	C5	C4	C3	C2	C1	C0

The protocol requires the most significant bit at '1'. The remaining bits represent the selected contrast value. A good contrast value to set for closed rooms is 16 (C6..c0 = 0010000_2). Once the contrast is set, we can go back to the instructions' "normal mode" with the following byte.

D/$\overline{C}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	1	0	0	0	V	0

In this step, we can set the desired addressing mode on bit V (a '0' sets the mode for rows and a '1', the mode for columns). If we want to use the display to show text, it is best to use row addressing as suggested in the previous example. We can change the addressing mode at any time by using the same command.

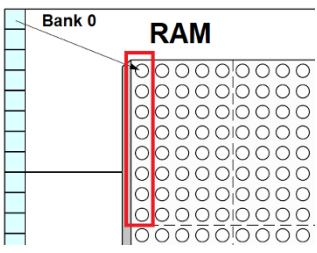
Lastly, we set the normal mode for the display and then send the data.

D/$\overline{C}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	0	1	1	0	0

Now we can finally move forward and send the data to be shown.

Transmitting the data to the display

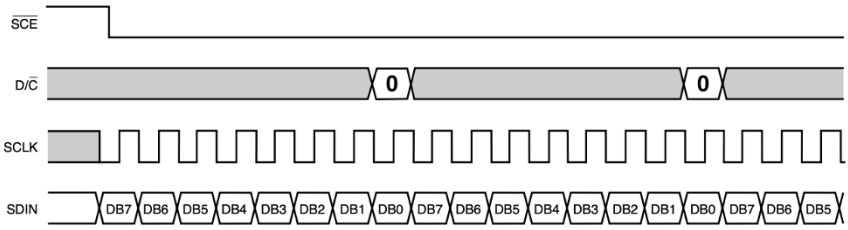
The system reset has zeroed registers X and Y, which point to the memory. Therefore the currently selected RAM cell is the first one in Bank zero, which drives the vertical strip of pixels at the upper right (see the figure at the right). At every write operation, registers X and Y are incremented according to the chosen addressing method with no need for external intervention. So as we send more data bytes, contiguous locations will be written.



At the beginning, the RAM contains unknown values, so it is best to initialize it at a known value by sending 504 (84 × 6) data bytes. The automatic

handling of write addresses makes it so that after 504 writes, the contents of register X and Y go back to the initial value (0 in this case).

To send the data, we need to select the "data mode" on the interface by setting the D/$\overline{\text{C}}$ line to '1'. At this point, we can start to send the bytes to the display, one by one. The following timing diagram shows the signals involved in sending two consecutive data bytes to the chip.



Changing the write flow of the RAM

If we wanted to skip to writing an arbitrary memory location, without following the established order of the selected writing modality, we could re-write the X and Y addressing registers. The next instruction writes register X, which addresses the X axis.

D/$\overline{\text{C}}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	1	X6	X5	X4	X3	X2	X1	X0

The following byte writes register Y, which addresses the Y axis.

D/$\overline{\text{C}}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	1	0	0	0	Y2	Y1	Y0

Once these registers are set, we can continue sending data, taking care to notify the interface by setting the D/$\overline{\text{C}}$ line to '1'.

Writing will start again as of the newly selected memory location. Registers X and Y will continue to be automatically managed by the chip according to the previously set addressing mode.

Other display operating modes

Aside from the normal mode, set at initialization, there are 3 others that we can select through the following command.

D/$\overline{\text{C}}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	0	1	D	0	E

The first mode turns off all the liquid crystals (D = '0', E = '0'), the second turns them all on (D = '0', E = '1') and the third inverts the colors of the display (D = '1', E = '1'). Normal mode is set by D = '1' and E = '0'.

Managing the display state

We can manage the display state by sending the following command byte:

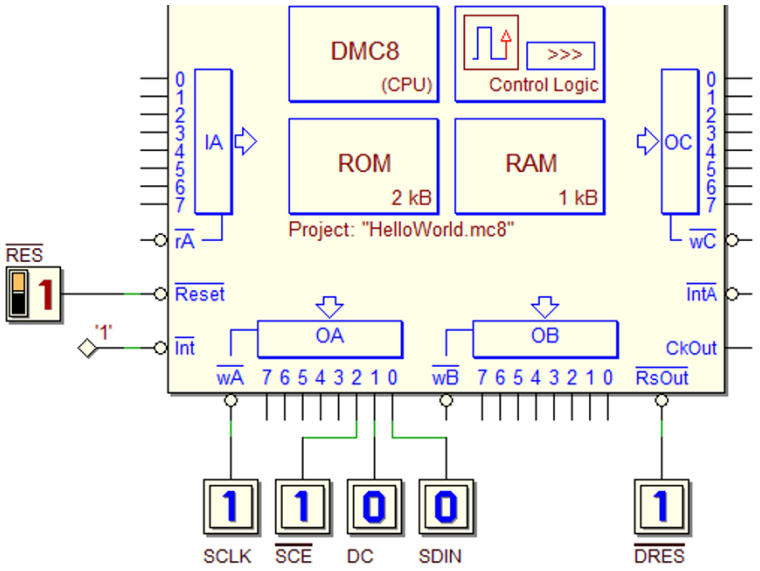
D/$\overline{C}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	1	0	0	PD	V	H

Bit PD ("Power Down") active high, turns off the display and leaves the internal state of the registers unaltered. To achieve minimum consumption we must fill the display RAM with zeroes. It is useful to keep a software copy of what was stored in the RAM before to be able to show it again the next time the display is turned on.

Bit V regulates the addressing mode. If it is set at '0' the "row addressing" mode is selected, otherwise the "column addressing" mode is used. Finally, bit H allows us to enter the "extended mode" of the instructions, when set at '1'. If it is set at '0' the normal instruction mode is selected.

5.6.6.1 The system

The system is based on the "DMC8 Microcomputer". At output port OA the display serial interface lines $\overline{\text{SCE}}$, DC and SDIN are connected to bits 2, 1 and 0 respectively. The port's $\overline{\text{wA}}$ signal strobe has been connected to the SCLK line, as the serial communication clock. The reset of the $\overline{\text{DRES}}$ display has been connected to the microcomputer's reset output $\overline{\text{RsOut}}$.



After the main program executes the necessary initializations, it shows the "Hello World!" message on the display and enters an infinite cycle where it is inactive.

The program

The address of the CNTRL output port is declared and this is followed by the link to the reset and the initialization of the Stack Pointer.

```
CNTRL   EQU   00h          ; IA port: SDIN (bit 0), DC (1), !SCE (2)
        ORG   0000h
        JP    0100h
        ORG   0100h
INIT:   LD    SP,0FFFFh    ; initialize the Stack Pointer
```

As seen in the previous section, before being able to see something on the display, we need to set the contrast. To set this parameter, we first need to use the "extended instruction" mode by sending the following command through the SENDC function, which will be analyzed later.

```
        LD    B,00100001b  ; PD = '0', V = '0', H = '1'
        CALL  SENDC        ; set the extended instruction mode
```

Then we send the next command, which sets the contrast to 8.

```
        LD    B,10001000b
        CALL  SENDC        ; set the contrast to 8
```

Finally we go back to normal instruction mode and set normal working mode.

```
        LD    B,00100000b  ; PD = '0', V = '0', H = '0'
        CALL  SENDC        ; go back to normal instruction mode
        LD    B,00001100b  ; D = '1', E = '0'
        CALL  SENDC        ; set normal working mode
```

The display's reset hardware has initialized all the internal registers to zero, so the pointers to the X and Y axes have been zeroed. We know that after reset, the state of the RAM is unknown. So the CLRSCR function is called to zero all the display's memory locations. This function leaves the state of the pointers intact.

```
MAIN:   CALL  CLRSCR       ; initialize all the RAM locations to zero
```

At this point, the address of the "Hello World!" string is loaded in index register IX. Then the X axis and Y axis where we want to start writing are loaded in registers B and C, respectively. After having loaded these parameters into the registers, we call the PRTSTR function, which shows the string on the display at the specified point (X,Y). Finally, the program enters an infinite loop where it is inactive, as per specifications.

```
        LD    IX,STR       ; copy the string's address to register IX
        LD    B,10         ; set X axis = 10 (through register B)
        LD    C,2          ; set Y axis = 2 (through register C)
        CALL  PRTSTR       ; finally, display the string
STOP:   JP    STOP         ; and enter in an inactive infinite loop
```

The CLRSCR function saves the used registers in the Stack then initializes the display RAM to zero. To do this, it enters a loop where it sends 504 data bytes through the SENDD function (analyzed below). After the saved registers are recovered, it goes back to the calling program.

```
CLRSCR:   PUSH  AF
          PUSH  BC
          PUSH  HL
          LD    HL,504      ; 84 x 6 = 504 RAM locations
          LD    B,0         ; set B = 0 to zero each location
CLRAM:    CALL  SENDD       ; of the display RAM
          DEC   HL          ; count the locations
          LD    A,L
          OR    H           ; set the flags according to HL value
          JP    NZ,CLRAM    ; exit if finished
          POP   HL
          POP   BC
          POP   AF
          RET
```

The next PRTSTR function shows on the display the string stored in memory at the address in register IX starting from the X axis indicated in B and the Y axis in C. After the used registers are saved as usual, the function sets the display's X and Y axes by calling the XSET and YSET subprograms analyzed below.

```
PRTSTR:   PUSH  AF
          PUSH  HL
          PUSH  BC
          PUSH  DE
          LD    A,B
          CALL  XSET        ; set X axis
          LD    A,C
          CALL  YSET        ; set Y axis
```

Then we read the first ASCII character of the string and make sure it is not equal to the string terminator (0). If it is, we exit with no further ado.

```
READ:     LD    A,(IX)      ; read the character pointed by IX
          CP    00h         ; check if it is the string terminator
          JP    Z,EXIT      ; exit if it is
```

If it is not equal, we go ahead and make sure the character we read is printable[52].

```
          CP    20h         ; if the character is not included
          JP    C,PINC      ; among the printable ones
          CP    7Fh
          JP    NC,PINC     ; jump to the label PINC
```

[52] That is can be shown and isn't a "control character", such as the previously mentioned terminator of the string.

If it is not printable, we jump to the PINC label, where we move on to reading the next character without showing anything.

The ASCII code that has been read is then adjusted to make index 0 coincide with the first printable character, to appropriately address the table containing the fonts[53], shown below. For now, it suffices to know that it is made up of 5 bytes for each printable code and it takes up a total of 480 bytes.

To get the address of the first byte representing the character graphically, we need to multiply the adjusted index by 5 to calculate the needed offset. Since there are more than 256 bytes making up the table, we have used the 16-bit HL register to process the index as described.

```
        SUB   20h          ; adjust the index excluding
                           ; the not-printable characters
        LD    L,A          ; copy the 8-bit index to HL
        LD    H,00h        ; and multiply it by 5
        ADD   HL,HL        ; x 2
        ADD   HL,HL        ; x 2
        ADD   A,L          ; +1
        JP    NC,NCY
        INC   H            ; if Carry, increment the high byte
NCY:    LD    L,A          ; now we have the offset in HL
```

Then, the resulting offset is added to the base address of the ASCII table, giving us the address of the first byte to send.

```
        LD    DE,ASCII     ; add the ASCII table base address
        ADD   HL,DE        ; to the offset
```

In a loop, all the data bytes are read and sent. We point to the data bytes using the HL register, which we increment on every loop repetition.

```
        LD    D, 5         ; set the number of bytes for each character
READ2:  LD    A,(HL)       ; read the byte to send (pointed by HL)
        LD    B,A
        CALL  SENDD        ; send data
        INC   HL           ; point to the next byte
        DEC   D            ; decrement the number of bytes
        JP    NZ,READ2     ; still to send
```

Finally, we go ahead and increment the IX register, to read the next character in the string on the next repetition as of the label READ.

```
PINC:   INC   IX           ; point to the next string character
        JP    READ         ; repeat loop as of READ
```

We only exit the subprogram and restore the used registers when we get to the terminator of the string.

[53] "Font", typically refers to typeface. Here, it refers to the sequence of bytes to send to the display that make up the characters in graphical terms.

```
EXIT:   POP   DE
        POP   BC
        POP   HL
        POP   AF
        RET
```

The XSET function sets the display's X axis pointer by using the value passed from the calling program to the accumulator. After saving the register in use, the function limits the value passed by the calling program to 83.

```
XSET:   PUSH  AF
        PUSH  BC
        CP    84          ; limit to 83 the value set for the X axis
        JP    C,XNCLP     ; if it is greater than 83 (when Carry = 0)
        LD    A,83
```

Then it sets bit 7 of the accumulator to '1', as required by protocol, and it sends everything to the display in the form of a command. In the end, we restore the previously saved registers and exit the function.

```
XNCLP:  SET   7,A         ; set bit 7 to '1',
        LD    B,A         ; as required by the protocol
        CALL  SENDC       ; send the command to the display
        POP   BC
        POP   AF
        RET
```

A similar operation is executed by the YSET function, which sets the address of the Y axis with the value passed to A, limiting it to 5.

```
YSET:   PUSH  AF
        PUSH  BC
        CP    6           ; limit to 5 the value set for the Y axis
        JP    C,YNCLP     ; if it is greater than 5 (when Carry = 0)
        LD    A,5
YNCLP:  SET   6,A         ; set bit 6 to '1',
        LD    B,A         ; as required by the protocol
        CALL  SENDC       ; send the command to the display
        POP   BC
        POP   AF
        RET
```

The SEND subprogram sends the byte in register B to the display in the form of a command if register C is set to 0, or in the form of data if it is not. We save the used registers on the Stack as usual, then counter E of the sent bits is initialized to 8.

```
SEND:   PUSH  AF
        PUSH  DE
        LD    E,8         ; initialize the bit counter to 8
```

After that, the $\overline{\text{SCE}}$ line is set to '0', while the DC line is set according to the content of register C.

```
            BIT   0,C
            JP    NZ,DATA       ; if C = '0'
COMMAND:    LD    A,00000000b   ; set DC = '0' (bit 1)
            JP    SLOOP         ; and go on to send a "command"
DATA:       LD    A,00000010b   ; otherwise send a "data byte" (DC = '1')
                                ; note that bit 2 (!SCE) is activated at '0'
```

We then go on to retrieve the most significant bit from register B and insert it in the bit in position 0, which commands data line SDIN. The configuration we get in the accumulator is copied on output port OA. Writing on the port causes the $\overline{\text{wA}}$ line to activate, which produces a clock pulse. On the clock rising edge, the data line is acquired by the peripheral.

```
SLOOP:      RLC   B             ; retrieve the next bit to send
            JP    NC,DRES       ; if it is high
DSET:       SET   0,A           ; set SDIN to '1' (on bit 0)
            JP    GO
DRES:       RES   0,A           ; otherwise set it to '0'
GO:         OUT   (CNTRL),A     ; send the bit on the serial line
```

This operation is repeated for all 8 bits in B. Then, we disable the display interface setting $\overline{\text{SCE}}$ to '1', restore the registers and exit the subprogram.

```
            DEC   E             ; decrement the number of bits to send
            JP    NZ,SLOOP      ; leave the loop if all bits have been sent

            LD    A,00000100b   ; set !SCE = '1' (bit 2)
            OUT   (CNTRL),A     ; to disable the display interface

            POP   DE
            POP   AF
            RET
```

The SENDD and SENDC versions of the SEND subprogram do nothing more than set the parameter of register C before calling it. SENDC puts C = 0 since it sends a command on the serial line. Because it sends a data byte on the serial line, however, SENDD puts C = 1.

```
SENDC:      PUSH  BC
            LD    C,0           ; send a command
            CALL  SEND
            POP   BC
            RET
SENDD:      PUSH  BC
            LD    C,1           ; send a data byte
            CALL  SEND
            POP   BC
            RET
```

What follows is the definition in the ROM memory of the message string to show and the table of the pixels corresponding to the ASCII characters (the table is shown only in part due to space restrictions).

```
STR:    DB    "Hello World!", 00h
ASCII:  DB    00h, 00h, 00h, 00h, 00h      ; 20 = (space)
        DB    00h, 00h, 5fh, 00h, 00h      ; 21 = !
        DB    00h, 07h, 00h, 07h, 00h      ; 22 = "
        DB    14h, 7fh, 14h, 7fh, 14h      ; 23 = #
        DB    24h, 2ah, 7fh, 2ah, 12h      ; 24 = $
        DB    23h, 13h, 08h, 64h, 62h      ; 25 = %
        DB    36h, 49h, 55h, 22h, 50h      ; 26 = &
        DB    00h, 05h, 03h, 00h, 00h      ; 27 = '
        DB    00h, 1ch, 22h, 41h, 00h      ; 28 = (
        DB    00h, 41h, 22h, 1ch, 00h      ; 29 = )
        DB    14h, 08h, 3eh, 08h, 14h      ; 2A = *
        DB    08h, 08h, 3eh, 08h, 08h      ; 2S = +
        DB    00h, 50h, 30h, 00h, 00h      ; 2C = ,
        DB    08h, 08h, 08h, 08h, 08h      ; 2D = -
        DB    00h, 60h, 60h, 00h, 00h      ; 2E = .
        DB    20h, 10h, 08h, 04h, 02h      ; 2F = /
        DB    3eh, 51h, 49h, 45h, 3eh      ; 30 = 0
        DB    00h, 42h, 7fh, 40h, 00h      ; 31 = 1
        DB    42h, 61h, 51h, 49h, 46h      ; 32 = 2
        DB    21h, 41h, 45h, 4bh, 31h      ; 33 = 3
        DB    18h, 14h, 12h, 7fh, 10h      ; 34 = 4
        DB    27h, 45h, 45h, 45h, 39h      ; 35 = 5
        DB    3ch, 4ah, 49h, 49h, 30h      ; 36 = 6
        DB    01h, 71h, 09h, 05h, 03h      ; 37 = 7
        DB    36h, 49h, 49h, 49h, 36h      ; 38 = 8
        DB    06h, 49h, 49h, 29h, 1eh      ; 39 = 9
        DB    00h, 36h, 36h, 00h, 00h      ; 3A = :
        DB    00h, 56h, 36h, 00h, 00h      ; 3B = ;
        DB    08h, 14h, 22h, 41h, 00h      ; 3c = <
        ...   .............                ; ...
        ...   .............                ; ...
```

5.6.6.2 Implementation on FPGA

The following sections have figures that summarize the connections chosen for each board, which are used to test the system.

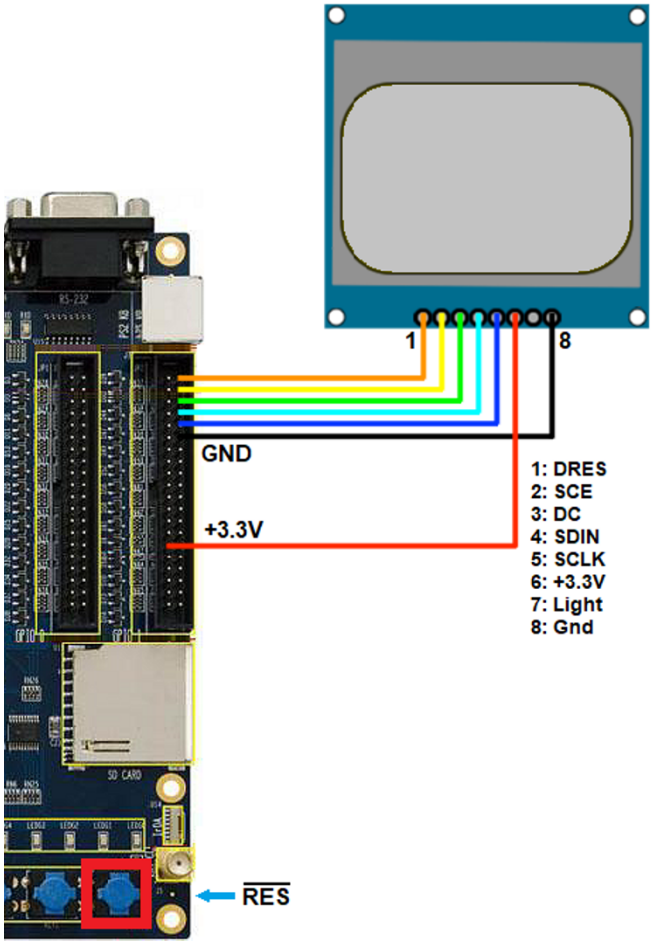
Note: the LCD components on the market are produced by different companies. They may differ in the position of the connections on the connector. If any component is unlike the type indicated in the following figures, it will be necessary to review the connections according to the maker's indications, which are often shown on the silk-screen printing of the board itself.

The DE2 board

The following figures show the devices chosen for the DE2 board, both in table form (as generated by Deeds), and in terms of physical connections.

In/Out Name	VHDL Name	Board Resource	Aux. Info
Inputs:			
!RES	inRES	Push-Button: Key[00]	Low (if pressed)
Ck of ("Micro	—	Clock: 10 MHz	
Outputs:			
!SCE	onSCE	Header 1: GPIO_1[03]	
DC	oDC	Header 1: GPIO_1[05]	
SDIN	oSDIN	Header 1: GPIO_1[07]	
SCLK	oSCLK	Header 1: GPIO_1[09]	
!DRES	onDRES	Header 1: GPIO_1[01]	

Take care to also connect the power supply wires (+3,3V and GND) of the display, as shown in the following figure.

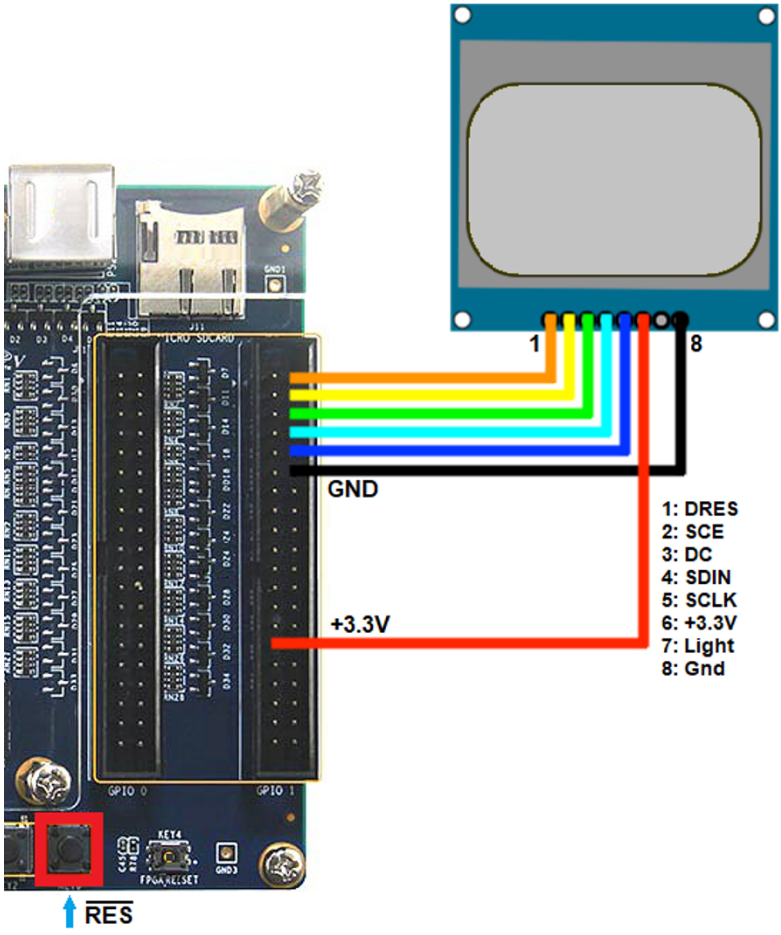


The DE0CV board

The assignment of devices for the DE0-CV board is very similar to that set for the DE2.

In/Out Name	VHDL Name	Board Resource	Aux. Info
Inputs:			
!RES	inRES	Push-Button: Key[00]	Low (if pressed)
Ck of ("Micro	—	Clock: 10 MHz	
Outputs:			
!SCE	onSCE	Header 1: GPIO_1[03]	
DC	oDC	Header 1: GPIO_1[05]	
SDIN	oSDIN	Header 1: GPIO_1[07]	
SCLK	oSCLK	Header 1: GPIO_1[09]	
!DRES	onDRES	Header 1: GPIO_1[01]	

As before, pay close attention when connecting the power supply for the display (+3.3V and GND) as shown in the figure.

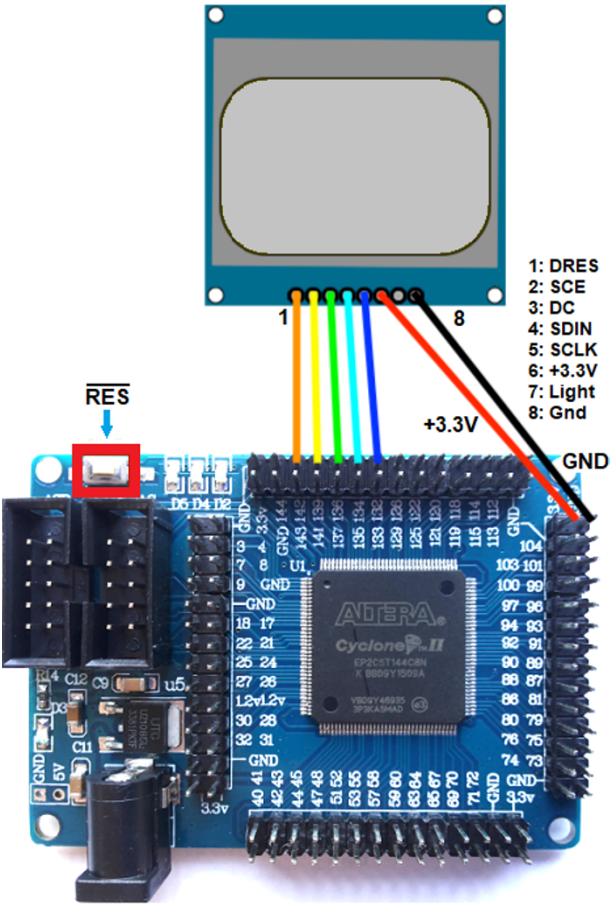


The EP2C5 board

We will use the only push-button on the board for manual system reset. The following table shows a summary of the connections as generated by Deeds.

In/Out Name	VHDL Name	Board Resource	Aux. Info
Inputs:			
!RES	inRES	Push-Button: Key0	Low (if pressed)
Ck of ("Micro	—	Clock: 10 MHz	
Outputs:			
!SCE	onSCE	Header 2: P3 [IO_139]	
DC	oDC	Header 2: P3 [IO_136]	
SDIN	oSDIN	Header 2: P3 [IO_134]	
SCLK	oSCLK	Header 2: P3 [IO_132]	
!DRES	onDRES	Header 2: P3 [IO_142]	

What follows is the photo of a board with the physical connections to make. Again, pay close attention when connecting the power supply for the display (+3.3V and GND) as shown in the figure.

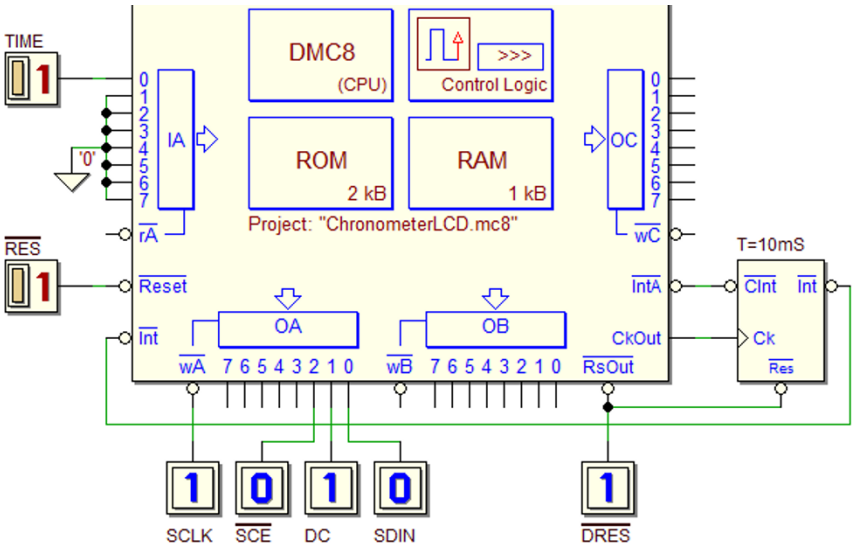


5.6.7 LCD stopwatch

In this project, we will analyze and build a digital stopwatch based on the DMC8 microcomputer that shows the time count on an LCD graphic display.

5.6.7.1 The system

This system, which is based on a "DMC8 Microcomputer" component, is shown in the following figure.



The TIME push-button is connected to the system through line 0 of input port IA. When it is pressed it generates a zero. The push-button also starts, pauses and restarts the time count. Pressing the $\overline{\text{RES}}$ push-button restarts the whole system and zeroes the count.

The counted time is shown on the LCD display of the Nokia 5110 phone used in the example in Section 5.6.6, much of whose code is borrowed to drive the component.

On output port OA, the display's interface lines $\overline{\text{SCE}}$, DC and SDIN are connected to bits 2, 1 and 0, respectively. The port's $\overline{\text{wA}}$ strobe signal is connected to the SCLK line (the serial communication clock). $\overline{\text{DRES}}$, the display reset, is connected to the microcomputer's reset output $\overline{\text{RsOut}}$.

A timer activates interrupt requests $\overline{\text{Int}}$ every 10 *mS*. When the request is accepted a pulse on output $\overline{\text{IntA}}$ automatically deactivates line $\overline{\text{Int}}$. The timer allows the system to precisely measure time.

The program

The code starts with the declaration of input port BUTTON and output port CNTRL. CNTRL drives the display's serial interface. Read the comments in the code.

```
BUTTON    EQU   00h        ; IA port: push-button (bit 0)
CNTRL     EQU   00h        ; OA port: SDIN (bit 0), DC (1), !SCE (2)
```

This is followed by the declaration of the variables. EN activates the count (when set at '1', the count is enabled, otherwise it is not).

BSTATE and PBSTATE store the current and previous states of the TIME button downstream of the debouncing checks.

```
EN        EQU   0FC00h     ; count enable
BSTATE    EQU   0FC01h     ; TIME push-button current
PSTATE    EQU   0FC02h     ; and previous states
```

Then the variables for the BCD (Binary Coded Decimal) time count are declared. BCD facilitates the translation of numbers into ASCII characters.

More specifically, CSEC counts the hundredths of second needed to make one tenth of a second elapse; DSEC counts tenths of a second; SEC1 and SEC2 count seconds and finally, MIN1 and MIN2 count minutes. Each of the variables above takes up one byte of memory.

```
CSEC      EQU   0FC03h     ; counter for the hundredths of second
MIN2      EQU   0FC04h     ; BCD most significant minutes digit
MIN1      EQU   0FC05h     ; BCD least significant minutes digit
SEC2      EQU   0FC06h     ; BCD most significant seconds digit
SEC1      EQU   0FC07h     ; BCD least significant seconds digit
DSEC      EQU   0FC08h     ; BCD tenths of a second digit
```

Finally, the STR variable is declared. It contains the string to show on the display. The format chosen for displaying the time is MM:SS:D, where MM represents the two digits for the minutes, SS the two digits for the seconds and D the digit for the tenths of a second.

The string is composed of 5 digits, two separators (':') and one character that terminates the string (00h). Therefore, it occupies a total of 8 bytes.

```
STR       EQU   0FC09h     ; string to show on the display
```

Registers B, C and D are dedicated to storing the intermediate readings of the debouncing check. B stores the last reading, C stores the second-to-last and D stores the third-to-last.

After the usual links to the reset and the interrupt handler, the Stack Pointer is initialized. The LCD component is initialized by the INITD subprogram that zeroes the graphic RAM and defines the working parameters.

```
        ORG   0000h
        JP    0100h
        ORG   0038h
        JP    HINT
        ORG   0100H
INIT:   LD    SP,0FFFFh     ; initialize the Stack Pointer
        CALL  INITD         ; and the LCD display
```

Then the variables and the registers used are zeroed.

```
        LD    A,00h         ; zero all the other variables
        LD    (EN),A
        LD    (BSTATE),A
        LD    (PSTATE),A
        LD    (CSEC),A
        LD    (MIN2),A
        LD    (MIN1),A
        LD    (SEC2),A
        LD    (SEC1),A
        LD    (DSEC),A
        LD    B,A           ; zero the registers in use
        LD    C,A
        LD    D,A
```

When the DISPLAY subprogram is called, a zeroed time count is shown on the display screen. The time count has been set to zero by the previous initialization. This subprogram's code will be analyzed next. After enabling the interrupts, we enter an empty main loop and wait for an interrupt request.

```
        CALL  DISPLAY       ; a zeroed time is shown on the display
        EI
MAIN:   JP    MAIN
```

The INITD subprogram initializes the LCD display, sets its physical parameters and zeroes the graphic RAM. The sequence of instructions is taken from the first rows of the code in the example in Section 5.6.6.

```
INITD:  PUSH  BC
        LD    B,00100001b   ; PD = '0', V = '0', H = '1'
        CALL  SENDC         ; set the extended instruction mode
        LD    B,10001000b
        CALL  SENDC         ; set the contrast to 8
        LD    B,00100000b   ; PD = '0', V = '0', H = '0'
        CALL  SENDC         ; go back to normal instruction mode
        LD    B,00001100b   ; D = '1', E = '0'
        CALL  SENDC         ; set normal working mode
        CALL  CLRSCR        ; initialize all the RAM locations to zero
        POP   BC
        RET
```

Every 10 mS the timer generates an interrupt, which causes the execution of its handler at the HINT label. Before acquiring the state of the TIME button, HINT shifts the previous readings in registers B and C into registers C and D, respectively. This makes room for the new reading that is stored in B (read the comments in the code).

```
HINT:   LD    D,C           ; second-to-last reading into third-to-last
        LD    C,B           ; last reading into second-to-last
        IN    A,(BUTTON)    ; get the current push-button state
        AND   00000001b     ; mask the bits not of our interest
        LD    B,A           ; copy the current state to B
```

The new reading is then compared to the previous ones. If they all show the same value, the current one is confirmed and saved in the BSTATE variable.

Before saving the new reading in the BSTATE variable, we need to transfer the previous content of BSTATE into PSTATE, where we store the reading that was confirmed before.

Then, a logic operation between the two values will allow us to detect the falling edges of the push-button line. To do this, the previous read confirmed is copied also to register L.

```
        CP    C             ; compare the state with the previous
        JP    NZ,TUPDATE
        CP    D
        JP    NZ,TUPDATE    ; if the reading is confirmed
        LD    E,A           ; copy it to register E
        LD    A,(BSTATE)    ; get the last confirmed reading
        LD    L,A           ; and copy it to register L
        LD    (PSTATE),A    ; and to the PSTATE variable
        LD    A,E
        LD    (BSTATE),A    ; save the confirmed current reading
```

We execute an XOR operation to invert the new reading and put the result in AND with the previous one. Then, if bit 0 is at '1', a falling edge is detected and the content of EN, which enables the time count, is inverted.

```
        XOR   00000001b     ; logic operation to detect a falling edge
        AND   L             ; on bit 0
        JP    Z,TUPDATE     ; if it is detected, go on and
        LD    A,(EN)        ; invert the enable variable (EN)
        CPL
        LD    (EN),A
```

Regardless of the result, we go on to evaluate the EN flag. If it is zero, we exit the handler without changing the state of the display. If it is not, we move on to the time count, starting from the hundredths of a second.

```
TUPDATE: LD   A,(EN)        ; check if the time count is enabled
        OR    A
        JP    Z,EXIT        ; if it is, go on, otherwise exit
```

Then the variable that counts hundredths of a second (CSEC) is incremented. If 10 hundredths have elapsed, we proceed by incrementing the tenths of a second and zeroing the hundredths. Otherwise, we exit without changing the state of the display.

```
LD    A,(CSEC)     ; increment the hundredths of a second
INC   A
LD    (CSEC),A
CP    10           ; if 10 hundredths have not elapsed,
JP    NZ,EXIT      ; jump and exit, otherwise
LD    A,0          ; zero the hundredths of a second
LD    (CSEC),A     ; and go on, incrementing the tenths
```

If the tenths of a second are at 10, it means we need to increment the second counter. We zero the tenths of a second counter and go on to increment the least significant digit of the seconds. Otherwise, we jump to update the state of the display and exit.

```
LD    A,(DSEC)
INC   A            ; increment the tenths of a second
LD    (DSEC),A
CP    10           ; if a second have not elapsed,
JP    NZ,EXITD     ; jump to update the display and exit
LD    A,0
LD    (DSEC),A     ; otherwise zero the tenths of a second
LD    A,(SEC1)     ; and increment the least significant digit
INC   A            ; of the seconds
LD    (SEC1),A
```

If the least significant digit of the seconds has gotten to 10, it means that it is time to increment the most significant digit and zero the least significant. If not, we jump to update the display and exit.

```
CP    10           ; if 10 seconds have not elapsed,
JP    NZ,EXITD     ; jump to update the display and exit
LD    A,0          ; otherwise zero the least significant
LD    (SEC1),A     ; digit of the seconds
LD    A,(SEC2)     ; and increment the most significant digit
INC   A
LD    (SEC2),A
```

If the most significant digit of the seconds has gotten to 6, we need to update the minutes, so we proceed by zeroing that number[54] and incrementing the minutes. If not, we update the state of the display and exit.

```
CP    6            ; if 60 seconds have not elapsed,
JP    NZ,EXITD     ; jump to update the display and exit
```

[54] The seconds' least significant digit was already zeroed and passed from 59 to 60.

```
        LD      A,0             ; otherwise zero the seconds (the least
        LD      (SEC2),A        ; significant digit was already zeroed)
        LD      A,(MIN1)        ; and increment the least significant digit
        INC     A               ; of the minutes
        LD      (MIN1),A
```

Through the same process we update the minutes: first, we increment the least significant digit and if it has not gotten to 10, we jump to update the display and exit. Otherwise, we zero it and increment the most significant digit. When this gets to 6, it is zeroed but the hours are not incremented since we will not see them[55]. Thus the count restarts from zero.

```
        CP      10              ; if 10 minutes have not elapsed,
        JP      NZ,EXITD        ; jump to update the display and exit

        LD      A,0             ; otherwise zero the least significant
        LD      (MIN1),A        ; digit of the minutes
        LD      A,(MIN2)
        INC     A               ; and increment the most significant digit
        LD      (MIN2),A
        CP      6               ; if 60 minutes have not elapsed,
        JP      NZ,EXITD        ; jump to update the display and exit

        LD      A,0             ; otherwise zero the minutes (the least
        LD      (MIN2),A        ; significant digit was already zeroed)
```

Finally, we update the display by calling the DISPLAY subprogram, which will be analyzed further on. Then we re-enable the interrupts and go back to the calling program.

```
EXITD:  CALL    DISPLAY         ; update the display
EXIT:   EI
        RET
```

The DISPLAY subprogram updates the time count on the LCD display. The individual decimal digits stored in the BCD code are first translated into the corresponding ASCII code. To do this, we add their value to the hexadecimal constant 30h (the first code in the ASCII table section representing the numbers). The resulting characters are then linked in a string with a colon (':') as a separator between the minutes and seconds and another between the seconds and tenths of seconds, giving us the classic MM:SS:D format.

The subprogram begins by saving the registers in use. Then it enters a counting loop where every variable representing time is read in the order they are stored in, starting from the most significant digit of the minutes and ending with the tenths.

```
DISPLAY:PUSH    AF              ; save the register in use on the Stack
        PUSH    DE
        PUSH    IX
        PUSH    IY
```

[55] With two extra variables, we could extend the code to count the hours.

Before entering the loop, register E is initialized at 5 (to count the number of digits used). Then the address of the STR string is assigned to index register IX (the chain of characters is stored in STR). IY is set at the address of the first digit of the measure to convert.

```
        LD    E,5          ; set the counter of the 5 digits
        LD    IX,STR       ; set the string address in register IX
        LD    IY,MIN2      ; set the address of the first digit in IY
```

In the loop, the current digit of the time measure, whose address is contained in IY is read so that its value can be translated into the corresponding ASCII code. The resulting character is then saved in the STR string, concatenated to the previous characters.

```
DLUP:   LD    A,(IY)       ; get the current digit and add 30h
        ADD   A,30h        ; to translate it into the corresponding
        LD    (IX),A       ; ASCII code and add it to the string
```

Let's use a little trick. If the loop counter is not zero and if it is even, it means that we are between the minutes and the seconds or between the seconds and the tenths so we need to insert a colon (':') to separate the groups of digits.

```
        LD    A,E          ; get the digit counter
        CP    0
        JP    Z,NOSEP      ; if this is not the last digit,
        BIT   0,A          ; and if the counter index is even
        JP    NZ,NOSEP
        INC   IX           ; add the separator character ':' to
        LD    A,3Ah        ; the string to separate minutes from
        LD    (IX),A       ; seconds and seconds from tenths
```

In any case, we update indexes IX and IY, and the loop counter. If all the digits have been processed, we add the terminator (00h) to the sting end.

```
NOSEP:  INC   IY           ; otherwise, update the pointers
        INC   IX           ; to the string and to the digit
        DEC   E            ; and decrement the digit counter
        JP    NZ,DLUP      ; if all the digits have passed,
        LD    A,00h        ; add the string terminator
        LD    (IX),A
```

Lastly, the address of the STR string is reloaded in IX and then passed to the PRTSTR subprogram, which shows it on the display at the coordinates indicated in registers B and C. For a description of the PRTSTR subprogram, refer to the example in Section 5.6.6.

```
        LD    IX,STR       ; copy the string's address to register IX
        LD    B,22         ; set X axis = 22 (through register B)
        LD    C,2          ; set Y axis = 2 (through register C)
        CALL  PRTSTR       ; display the string
```

Finally, we restore the registers used and exit.

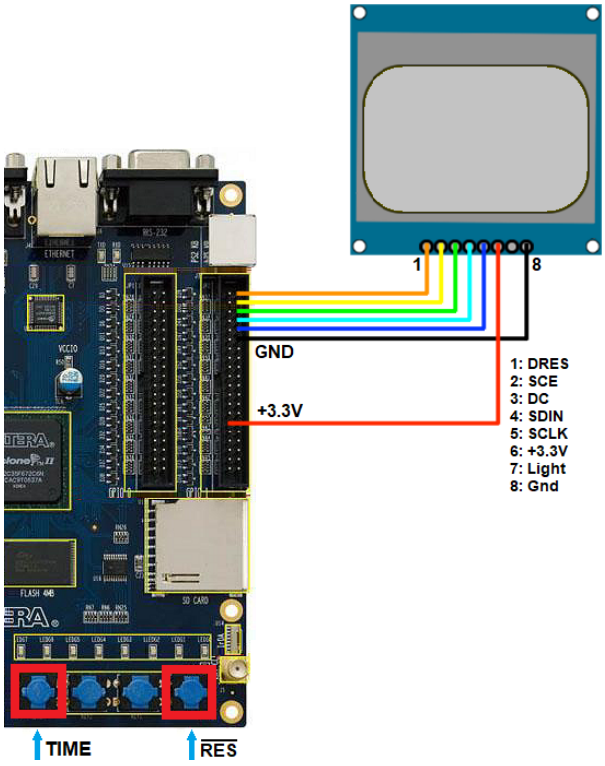
```
POP    IY
POP    IX
POP    DE
POP    AF
RET
```

5.6.7.2 Implementation on FPGA

Here, we show images that summarize the connections for each board, which are useful for testing the system (refer to the observations on Page 540 regarding the different versions of the LCD component).

The DE2 board

In/Out Name	VHDL Name	Board Resource	Aux. Info
Inputs:			
Ck of f"Chro	—	Clock: 10 MHz	
TIME	iTIME	Push-Button: Key[03]	Low (if pressed)
!RES	inRES	Push-Button: Key[00]	Low (if pressed)
Outputs:			
!SCE	onSCE	Header 1: GPIO_1[03]	
DC	oDC	Header 1: GPIO_1[05]	
SDIN	oSDIN	Header 1: GPIO_1[07]	
SCLK	oSCLK	Header 1: GPIO_1[09]	
!DRES	onDRES	Header 1: GPIO_1[01]	



The table above lists the devices chosen for the DE2 board (as generated by Deeds).

The figure on the left shows the physical connections to make between the board and the LCD component.

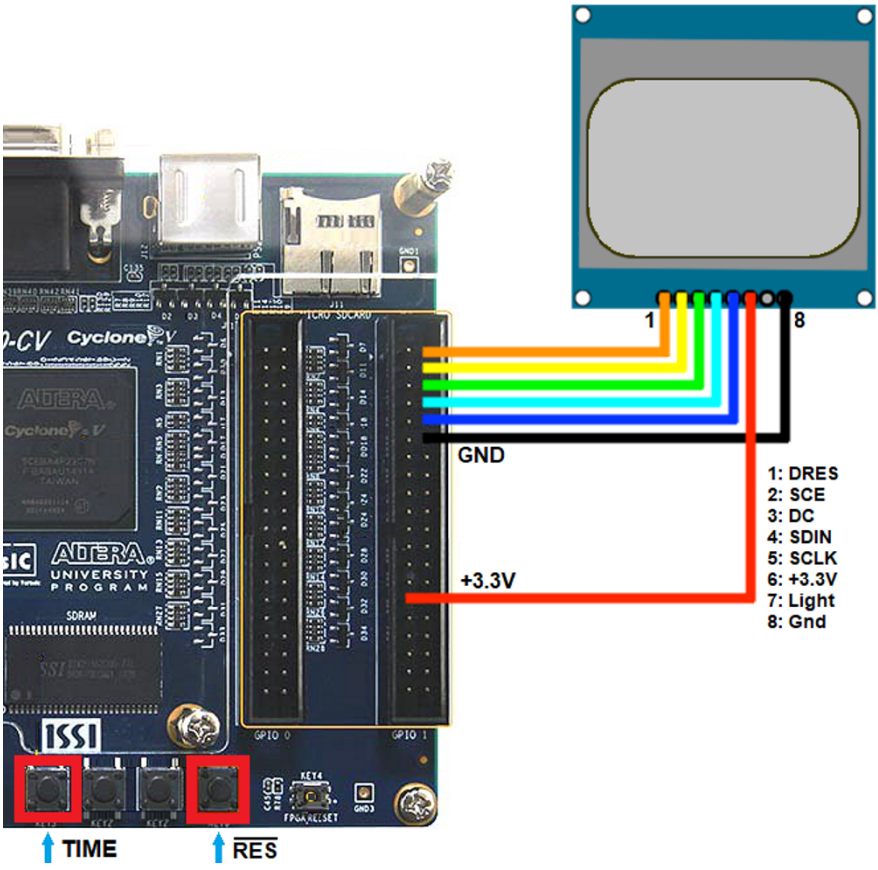
Take care to connect the display's power supply wires (+3,3V e GND) as well, as suggested in the figure.

The DE0CV board

The assignment of devices for the DE0-CV board is very similar to that set for the DE2.

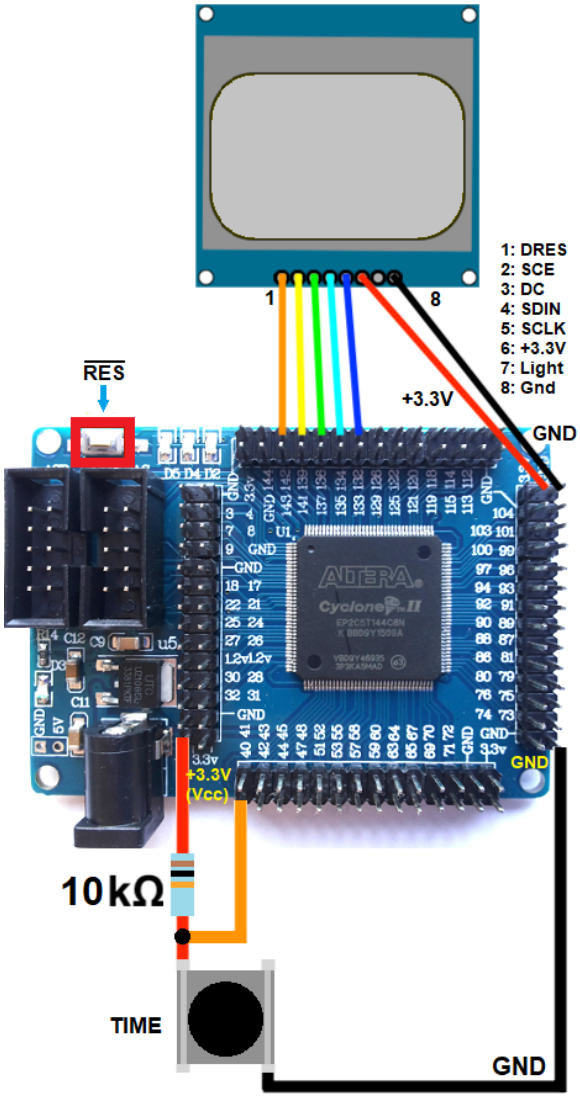
In/Out Name	VHDL Name	Board Resource	Aux. Info
Inputs:			
Ck of f"Chrot —		Clock: 10 MHz	
TIME	iTIME	Push-Button: Key[03]	Low (if pressed)
!RES	inRES	Push-Button: Key[00]	Low (if pressed)
Outputs:			
!SCE	onSCE	Header 1: GPIO_1[03]	
DC	oDC	Header 1: GPIO_1[05]	
SDIN	oSDIN	Header 1: GPIO_1[07]	
SCLK	oSCLK	Header 1: GPIO_1[09]	
!DRES	onDRES	Header 1: GPIO_1[01]	

As before, take care to connect the display's power supply wires (+3.3V and GND) as well, as shown in the following figure.



The EP2C5 board

In/Out Name	VHDL Name	Board Resource	Aux. Info
Inputs:			
Ck of ("Chro[—		Clock: 10 MHz	
TIME	iTIME	Header 0: P1 [IO_40]	
!RES	inRES	Push-Button: Key0	Low (if pressed)
Outputs:			
!SCE	onSCE	Header 2: P3 [IO_139]	
DC	oDC	Header 2: P3 [IO_136]	
SDIN	oSDIN	Header 2: P3 [IO_134]	
SCLK	oSCLK	Header 2: P3 [IO_132]	
!DRES	onDRES	Header 2: P3 [IO_142]	



The table above lists the devices chosen for the EP2C5 board (as generated by Deeds).

The figure on the left shows the physical connections to make between the EP2C5 board and the LCD display component.

We use the only push-button on the board to reset the system.

In the figure, the externally added TIME button and the 10K Ω pull-up resistor are highlighted.

Take care to also connect the power supply wires (+3,3V and GND), as shown in the figure.