Synthesizable VHDL Design for FPGAs

Eduardo Augusto Bezerra
Djones Vinicius Lettnin

Universidade Federal de Santa Catarina
Florianópolis, Brazil

August 2013

Contents

Chapter 1. Digital Systems, FPGAs and the Design Flow

This chapter introduces the target technology used in the laboratory sessions, and provides a step-by-step guide for the design and simulation of a digital circuit. At the end of this chapter, the students should be able:

- to have a basic understanding of the FPGA technology (internal blocks, applications);
- to understand the basic logic gates operation;
- to design a logic circuit using a schematic editor tool;
- to simulate and test a digital circuit;
- to understand the basic flow of an Electronic Design Automation (EDA) tool.

1.1 Digital Systems

Digital systems are composed of two basic components: Datapath and Control Unit. As shown in Figure 1.1, the control unit has as its main function the generation of control signals to the datapath unit (also known as "operational block"). The control signals "command" the desired operations in the datapath. Furthermore, the control unit may receive control inputs from the external environment; for instance, it can be a simple "start" or even an operation code ("opcode" in microprocessors). Finally, this unit can also generate one or more output control signals to communicate with other digital systems (e.g., "done, bus request, nack").

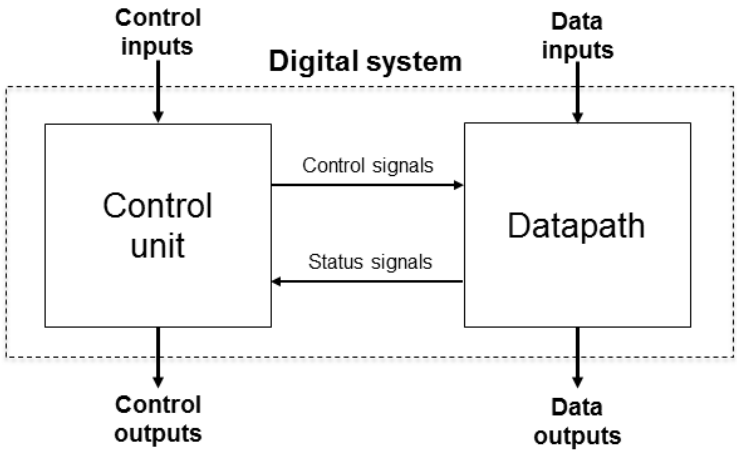

Figure 1.1. Digital system (adapted from [ManoKime99])

The datapath unit performs operations on data received, usually, from the external environment. The operations are performed in one or more steps, where each step takes a clock cycle. The datapath generates "status" signals (sometimes also called as "flags") that are used by the control block to define the sequence of operations to be performed. Examples of datapath blocks include:

- Interconnection network - wires, multiplexers, buses, and tri-state buffers;
- Functional units - adders, subtractors, shifters, multipliers, and Arithmetic and Logic Unit (ALU);
- Memory elements - registers, and Random Access Memory (RAM).

The datapath and control units are based on two types of circuits: combinational and sequential circuits.

Combinational circuits have I_m inputs and O_n outputs, as shown in Figure 1.2. In these circuits the output pins depend only on the values that are presented to the input pins and each output is defined by a different Boolean logic equation.

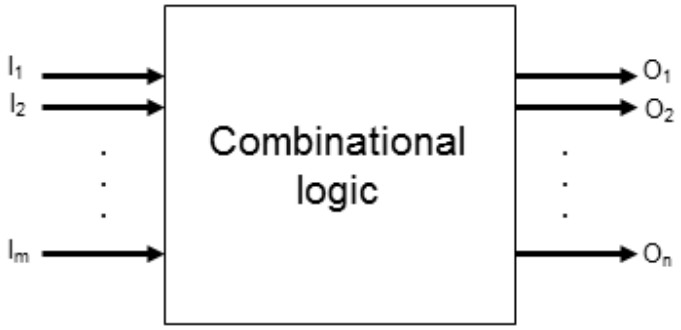

Figure 1.2. Combinational circuit

A combinational circuit can also be classified according to its application as:

- Interconnection circuit - e.g. multiplexers, demultiplexers, encoders, and decoders;
- Logical and arithmetic circuit - e.g. adders, subtractors, multipliers, shifters, comparators, and ALUs (circuits that combine more than one arithmetic or logical modules).

Sequential circuits have I_m inputs and O_n outputs. However, in these circuits the output pins depend not only on the values that are presented to the input signals (i.e. m signals), but they depend also on the current state stored in the memory module, as shown in Figure 1.3. The number of both next state (i.e. N_j) and current state signals (i.e. S_j) depend on the encoding style of a Finite State Machine (FSM) states, such as, sequential binary, gray code or one hot.

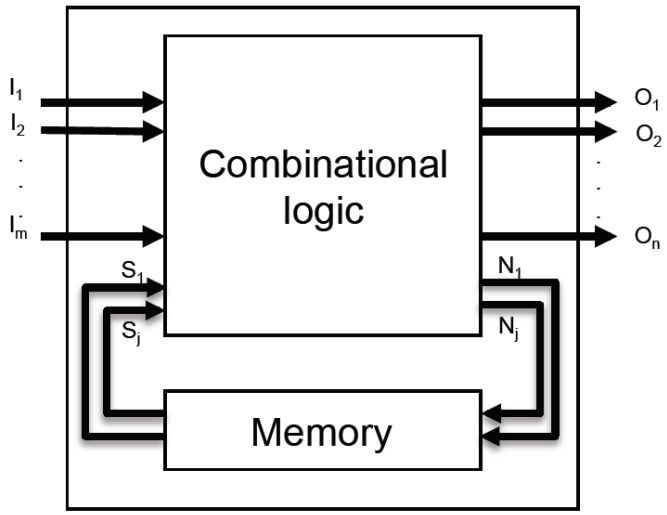

Figure 1.3. Sequential circuit.

1.2 Field Programmable Gate Array (FPGA)

The concept behind the Field Programmable Gate Array (FPGA) technology is better understood through a digital circuit design example.

Problem:

You are asked to design a circuit that implements the following logic function:

f(A, B) = A AND B

In this circuit, inputs *A*, *B*, and output *f* are all 4 bits wide.

Solution 1: Using a 7408 TTL device (Application-specific Integrated Circuit - ASIC solution)

The operations to be performed are:

f(1) = A(1) AND B(1)
f(2) = A(2) AND B(2)
f(3) = A(3) AND B(3)
f(4) = A(4) AND B(4)

The 7408 TTL device, shown in Figure 1.4, has four 1-bit AND gates. Therefore, a single 7408 chip is sufficient to implement the 4 bits function. In Figure 1.4, the *A* inputs are connected to pins 1, 4, 9 and 12. The *B* inputs are connected to pins 2, 5, 10 and 13. The *f* output can be obtained from pins 3, 6, 8 and 11. The chip must be placed (soldered) on a printed circuit board (PCB) or protoboard, and all pin connections should be made, including the power lines (Vcc and GND).

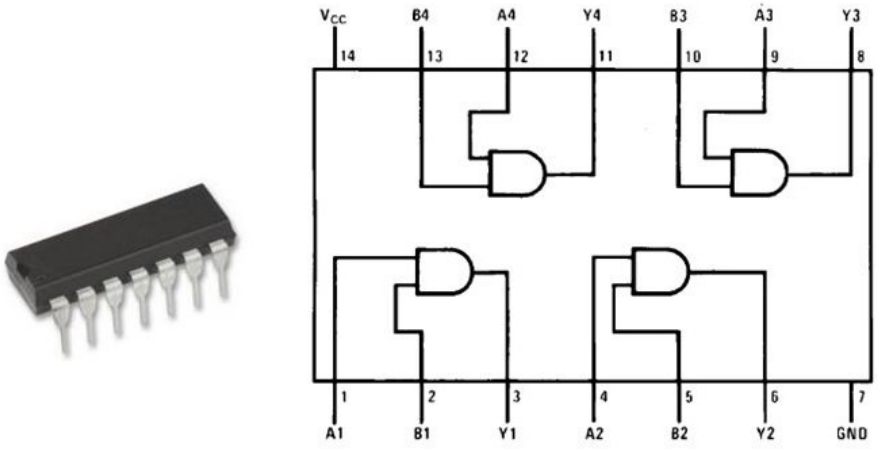

Figure 1.4. 7408 TTL chip

This is a nice solution for the proposed problem, but it has some drawbacks. TTL is an obsolete technology, presenting high-energy consumption, and the chip itself takes considerable space on the PCB. Another drawback is in case of changes or upgrades in the design requirements. In these situations, a recall may be needed in order to replace the components (7408 chip) by the new ones. Depending on the required changes, a full PCB redesign may be needed.

Solution 2: Microprocessor (software solution)

A microprocessor (or microcontroller) based system is not the best solution for this sort of "simple problem", but it has been included here in order to show the differences between the hardware and software approaches.

In a similar way to the first solution, a microprocessor also needs to be placed on a PCB, but it has more flexibility for changes in the system functionality. As the solution is implemented at the "software level", any changes in the design requirements may be performed in the system with no need for hardware or PCB modifications. A possible solution for the problem is shown in the C language program as follows:

```
int main(){
   int A, B, F;
   F = A & B;        // F = A AND B
}
```

However, this solution also has some drawbacks. A special C compiler may be required in order to have a 4-bits wide *int* type. Another difficulty would be to assign the inputs and outputs (A, B, F) to the microprocessor physical pins. Considering a standard C compiler, the developer would use *scanf* or *getc* input functions to obtain the *A* and *B* values, and *printf* to output the *F* result. Depending on the target systems, several adaptations would be necessary in order to associate the functions to the device's pins. In a microcontroller based C compiler, usually, special functions are provided to the programmer in order to access the device's pins.

Solution 3: FPGA

An FPGA device could be seen as an intermediary solution, between the hardware and the software approaches. In a similar way to the hardware (TTL chip) and the software (microprocessor chip) approaches, the FPGA chip also has to be placed on a PCB, and all its input and output pins should be connected. An important difference from the microprocessor approach is that the solution for the problem is not implemented in software. It is a hardware solution, similar to the 7408 chip, but with the same flexibility for changes as the microprocessor based approach. In case of bugs or design changes, a new circuit can be "placed" inside the FPGA, with no need for PCB modifications. A solution for the problem could be written in the VHDL language as follows:

```
library IEEE;
use ieee.std_logic_1164.all;
entity AND_VHDL is
   port ( A, B : in std_logic_vector (3 downto 0);
          F    : out std_logic_vector (3 downto 0)
   );
end entity;
architecture rtl of AND_VHDL is
begin
     F <= A AND B;
end rtl;
```

The microprocessor solution drawbacks are easily outwitted when using an FPGA. Different bit lengths for the inputs and outputs are defined by standard language constructions, as well as the assignment of physical pins.

However, an interesting advantage of Solution 3 (FPGA technology) is the flexibility for hardware changes, after the product has been completed. Assuming that a requirement mistake has been identified, and the circuit were supposed to perform an OR operation and not the implemented AND operation. In Solution 1, the 7408 chip would have to be replaced by the 7432 TTL device (four 1-bit OR gates), which means to remove the chip from the board, and to place the new one. In solutions 2 and 3, there will be no need for chips replacement. In Solution 2 the software would have to be changed and reloaded in the microprocessor's memory. In Solution 3, the VHDL code would have to be changed, using OR instead of AND, and a new FPGA configuration should be generated. It is important to notice that in Solution 3 a "new hardware" is generated, for the FPGA, while in Solution 2 a new piece of "software" is loaded in the microprocessor.

1.3 FPGA Internal Organization

An FPGA is a device (a chip) whose internal logic can be changed by the developer (user) after the chip has been manufactured – it is "field programmable". From the developer point of view, its functionality is similar to a microprocessor, as it is possible to change its operation according to the application requirements. In the ASIC approach, the

chip has its functionality defined by the manufacturer (permanently), and it cannot be changed. For instance, the 7408 device introduced in "Solution 1" has 4 1-bit AND gates, and this hardware arrangement is fixed and it is not possible to be modified. The microprocessor ("Solution 2"), also has a fixed hardware that cannot be changed (registers, ALU, …), but its functionality can be changed by the user at the "software level". The internal hardware of a microprocessor has been designed in order to execute instructions of a program, and different programs can be written according to the application requirements. An FPGA, on the other hand, has an internal logic that can be changed by the developer. The main similarity to a microprocessor is that the FPGA developer also uses a language to describe the application's functionality, and a tool to generate the new hardware to be configured inside the FPGA. However, as an FPGA is not a microprocessor, it has not been designed to execute a program written by the developer. The source code written by the developer is in fact a "hardware description", and it becomes the FPGA internal logic configuration.

In order to have its internal logic changed, the architecture of an FPGA chip employs "writable" technologies, such as SRAM or FLASH memories. FPGA devices have three main configurable components: the logic blocks; the routing elements; and the input/output (I/O) blocks. The three types of components are shown in Figure 1.5, where the I/O blocks are located in the periphery of the chip, and the logic blocks are the larger squares, interconnected by the routing elements.

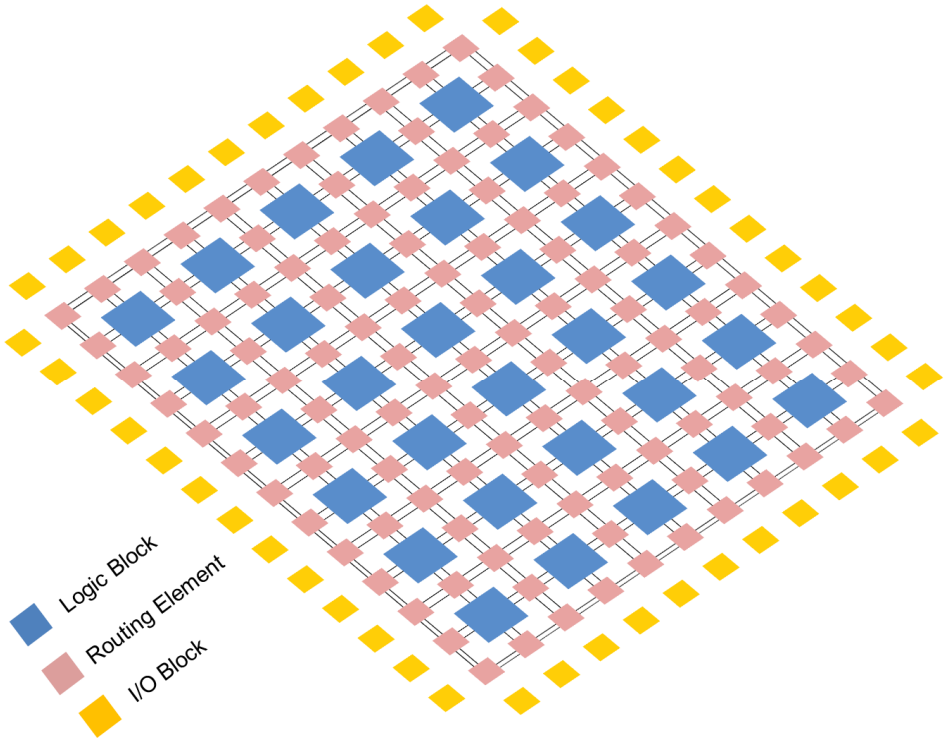

Figure 1.5. A typical FPGA internal components.

The “logic blocks” are configured in order to perform the logic operations required by the application, and so they are the “processing elements” of an FPGA. The logic blocks are interconnected through the configurable routing elements, which are used also for the connection to/from I/O blocks. The I/O blocks may be configured in order to function as input pins, output pins, or both. Figure 1.6 shows a possible hardware configuration for the Solution 3 described before. A single logic block is used to perform the AND operation. Eight I/O blocks are configured as input, and a path (routing) is configured between them and the AND logic block. Four I/O blocks are configured as output, and they are connected to the output of the logic block.

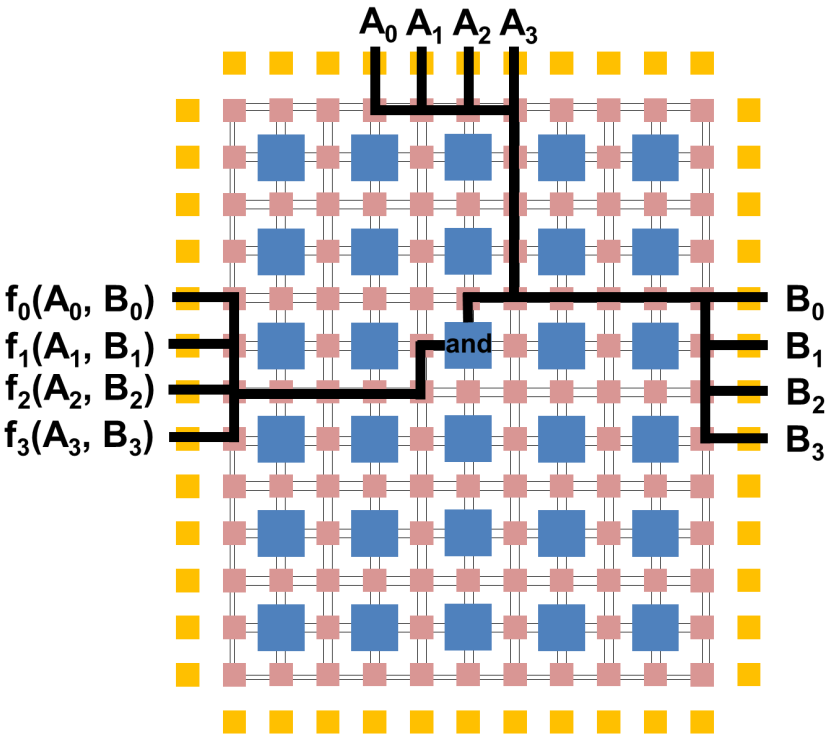

Figure 1.6. Possible FPGA configuration for Solution 3.

1.4 Configurable Logic Block

As stated before, FPGAs are made of three major configurable resources: Logic Blocks; Routing Elements; and I/O Blocks. The internal organization of all three elements is very interesting and deserves a more detailed explanation, but this is not the main goal of this book. However, in order to better understand the developing tools functionality, it is important to have a basic idea of how logic functions are implemented in FPGAs. Most FPGAs are based on look-up tables (LUTs), which can be used as “universal function generators”. An LUT is a configurable resource capable of implementing any n-input truth table. Figure 1.7 shows an abstract view of an LUT contents generation from a logic function.

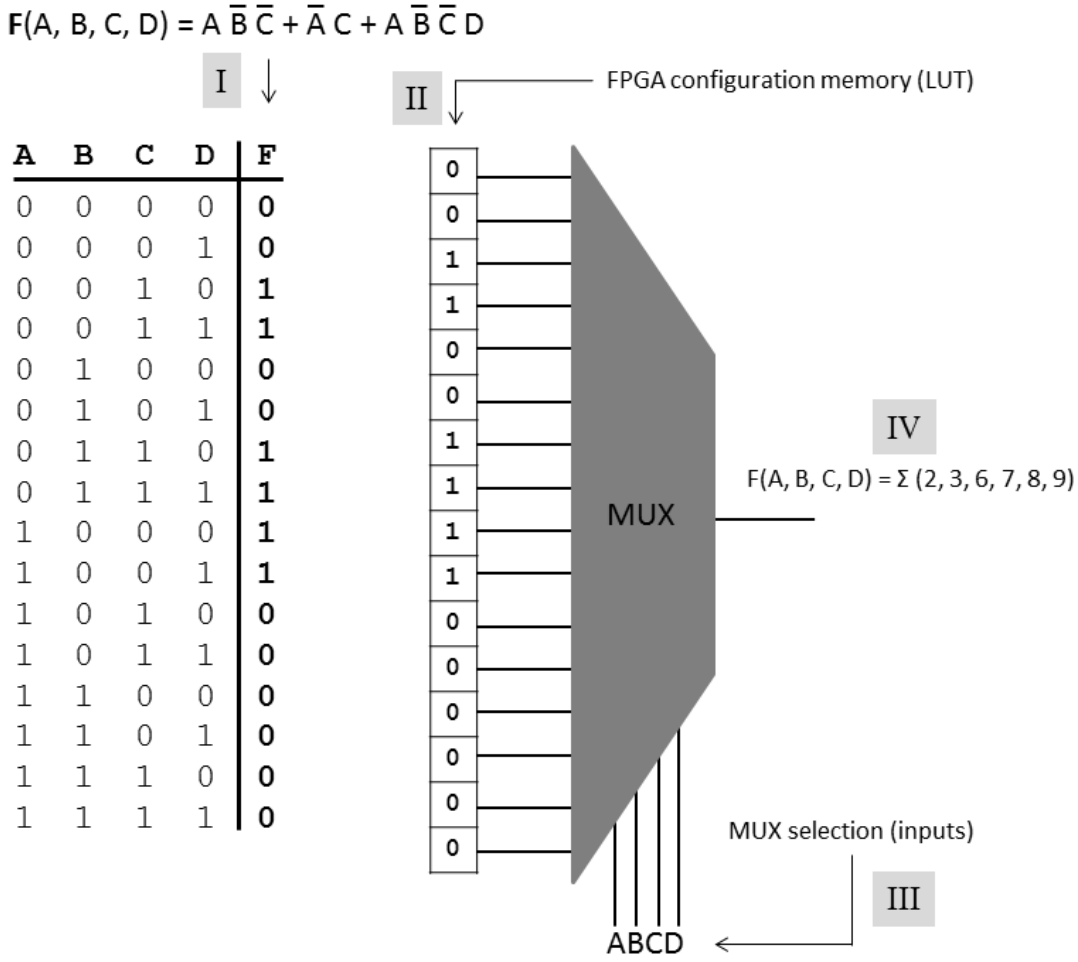

Figure 1.7. LUT working model.

In Figure 1.7, the logic function in the upper left hand corner is written in an input format accepted by the selected Electronic Design Automation (EDA) tool (e.g. schematic diagram, VHDL, Verilog, …). The EDA tool generates the truth table for the logic function (I). During the programming process, the results column of all truth tables is written to the FPGA configuration memory (II). More precisely, in a 4-input LUT based FPGA, a truth table for a 4-input logic function (A, B, C, D) is written to a LUT of a logic block. The FPGA does not need to have any physical logic gates, as only the results of a truth table are stored in the LUTs (FPGA configuration memory). In order to perform the equation, the inputs (A, B, C, D) are used as the MUX selection (III), and the provided output is the final result (IV). This functionality explains how Solution 3 described before can be implemented in an FPGA. As no logic gates are used to implement a logic function, each time a new functionality is required in a design, the circuit developer just needs to change the hardware description, and the EDA tool generates a new truth table to be used in the LUT configuration. An LUT is the building block of the Logic Block shown in Figure 1.5.

1.5 Electronic Design Automation (EDA) and the FPGA design flow

In order to develop a digital circuit targeting an FPGA device, it is compulsory to use Electronic Design Automation tools. Figure 1.8 shows a typical FPGA design flow, where several EDA tools are employed.

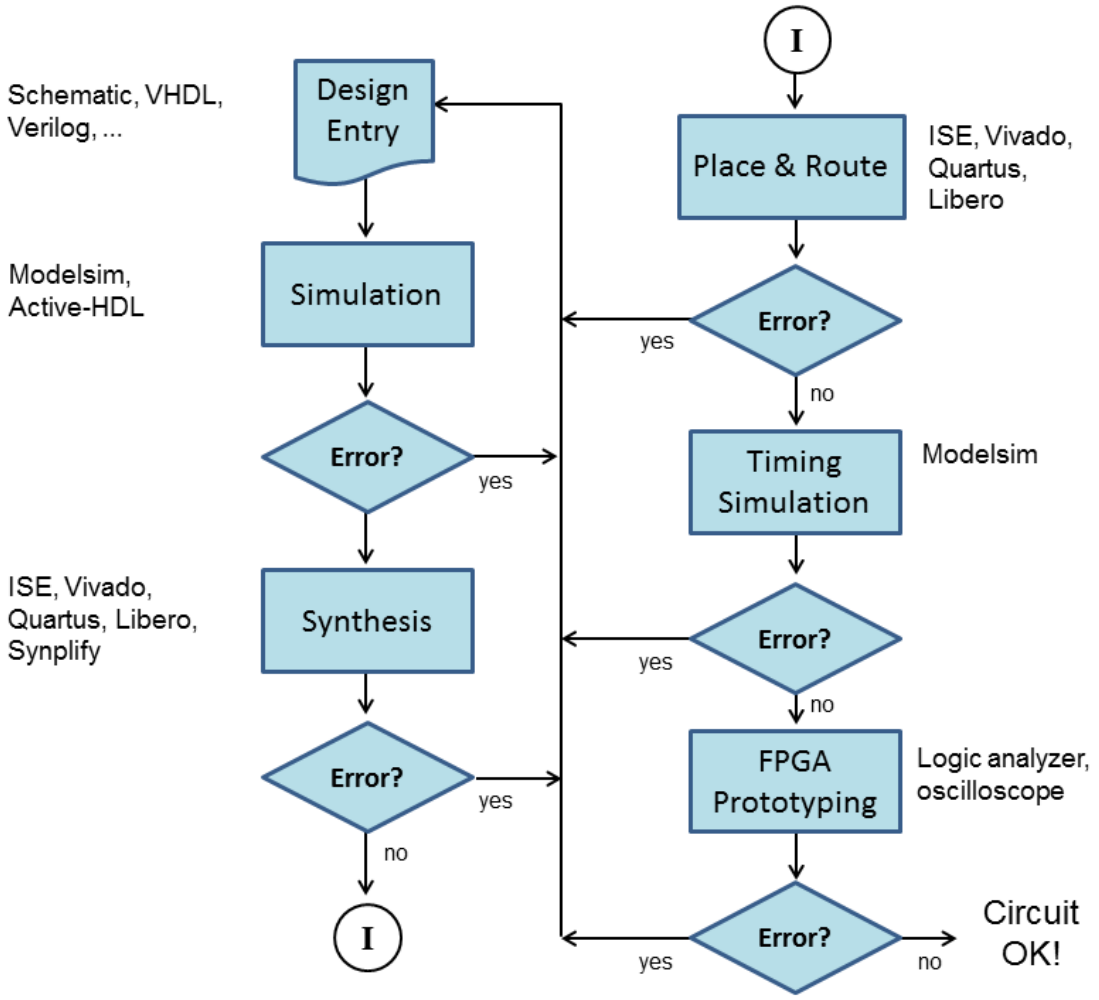

Figure 1.8. Typical FPGA design flow.

The first step is the "Design Entry", where the developer uses a design-editing tool to provide their circuits description. Usually, EDA tools take a number of description formats as, for instance, schematic diagrams, hardware description languages (HDLs), state machine diagrams, among others.

The next step in Figure 1.8 is the circuit simulation. There are several simulation tools available, and the most used method in all of them is the waveform analysis. The circuit's logic and states can be fully simulated using tools that provide a detailed waveform generation and analysis. It is important to notice that this is a "functional simulation", as at this stage of the design flow there is no timing information yet. The functional simulation process tests only the design's logic operation, as it has no further information regarding the internal circuit in the target device (FPGA) or routing path delays. This step

should be performed in order to verify the circuit functionality disregarding timing constraints and other performance parameters. In this step, in case of functional errors, the designer may change the circuit using the design entry tool, and perform new simulations until the design shows the expected behavior.

After have finished the functional simulation, the designer can perform the synthesis activity and a *netlist* representing the provided circuit is generated. This activity is split in two operations: the logic synthesis; and the physical synthesis. Traditionally, the logic synthesis role is to map the provided circuit to a set of gates (netlist) performing the same functionality as the original circuit description. This netlist has not yet any physical information such as power consumption or routing delays for the final physical implementation. The "design entry", the "functional simulation", and the "logic synthesis" are activities performed by a professional called the "Frontend Engineer".

The remaining activities shown in Figure 1.8 are performed by the "Backend Engineer". In FPGA designs, it is not unusual to have the same engineer performing both, frontend and backend, activities. The first backend activity in an FPGA design flow is the "placement and routing". At the end of the process, in case of errors the designer may go back to the design entry step in order to fix the problems pointed out during the synthesis. The physical synthesis output is a netlist and the respective physical design data. The design data, used in the timing simulation step, include performance figures such as maximum clock rate, area usage, power estimation, and throughput. The netlist has all the physical design information needed for the FPGA configuration, including: I/O pins, routing paths, and operational blocks for LUTs configuration, among others. In the physical synthesis step activities as placement, routing and circuits optimizations are accomplished. In contemporary EDA tools, the logic and the physical synthesis are implemented in a cooperative way, employing incremental algorithms. This strategy results in better results, as the logic synthesis is performed considering feedback information obtained from the physical synthesis.

In the "timing simulation" step, the designer employs the data provided by the physical synthesis in order to verify whether or not the physical design will meet the defined performance constraints. If the timing simulation results in acceptable figures, so there is a higher confidence that the physical circuit will work according to the requirements. However, only in the final step, that is, the FPGA prototyping, it is possible to have a better assurance of the circuit correctness. In this step, usually, the circuit is tested in a development board.

1.6 FPGA Devices and Platforms

Xilinx and Altera are the major players in the FPGA market. Their main business is the manufacture of FPGA devices, but they also develop EDA tools. The FPGA internal organization complexity is a motivation for these companies to produce their own physical synthesis tool. The FPGA manufacturer has, undoubtedly, the best understanding of a device's internal organization and, consequently, is the most suitable developer for a low level EDA tool. Another motivation for the FPGAs companies to develop their own physical synthesis tools is related to the confidential information regarding the device's internal organization. Holding back this information is important for a company as it may re-

sult in technology advantages over competitors. Other important players in the FPGA market include: Microsemi Corporation (formerly Actel); Lattice Semiconductor; and SiliconBlue Technologies.

These companies target very specific markets, where the most common applications are: networking (switches and routers); video and image processing; cryptography; space systems (satellites and deep space missions); avionics; defense; ASIC prototyping; audio equipment; medical solutions; motor control; high performance computing; and automotive systems. In order to attend the requirements of these applications, current FPGA devices have extra hardware resources as, for instance, DSP operators, RAM blocks, and even embedded microprocessors.

FPGA devices are used not only as basic components in the design of boards for a variety of applications, but also in the design of development and educational boards. Throughout this book, two educational boards are employed for the exercises prototyping:

- The Altera DE2 Development and Education board shown in Figure 1.9 has an Altera Cyclone II 2C35 FPGA, and several peripherals. The board has been designed for teaching digital systems classes, in laboratory-based courses.
- The Mercurio IV board has been developed by Macnica-DHW Engineering and it has an Altera Cyclone IV FPGA. It also has a set of peripherals, and was developed aiming university courses.

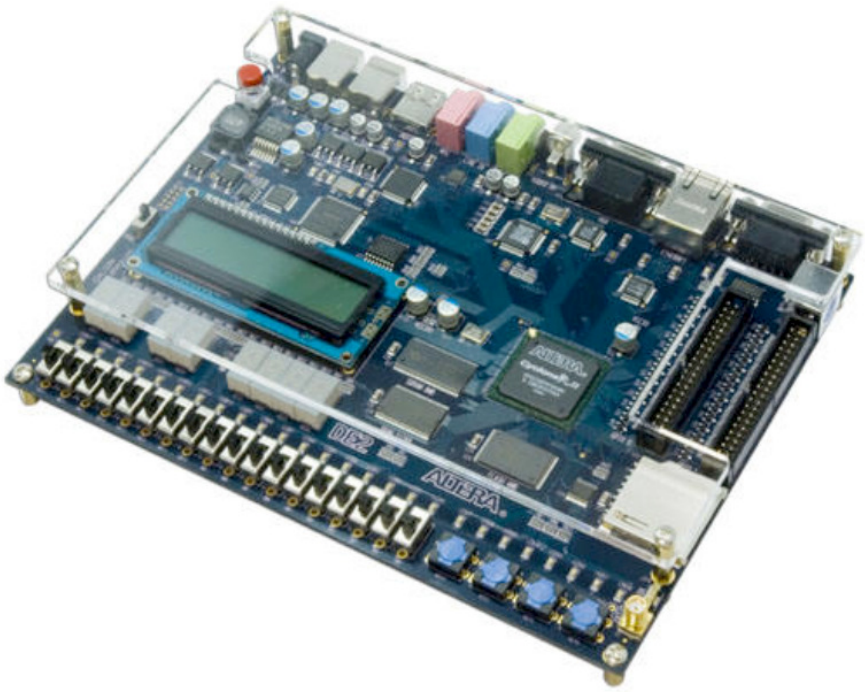

Figure 1.9. DE2 Development and Education board [Altera].

1.7 Writing Software for Microprocessors and VHDL Code for FPGAs

Applications for FPGAs and microprocessor based systems can be modeled and conceived at different levels of abstraction, and using different design entry formats. A compiler used to generate object code (machine code) for a microprocessor can have as input, for instance, C/C++ source code (computer program) and also other pieces of software and components available in libraries. A synthesis tool (hardware compiler) used to generate a netlist (bitstream, circuit) to an FPGA has as input a code written in an HDL (VHDL, Verilog, …), as well as components available in libraries. An important distinction here is that a piece of software written in the C language is at a level of abstraction many times higher than a hardware description written in VHDL. Even the assembly language can be considered as in a higher abstraction level than VHDL. The VHDL developer is in fact describing the behavior of a piece of hardware to be used to implement some desired functionality (application), while with the assembly language (or any other high level languages) the developer is describing the application's functionality that will be performed by an existing piece of hardware (microprocessor). The VHDL developer uses the language to design the hardware and the application to be performed, while a microprocessor programmer has to worry, basically, with the application functionality to be implemented.

Considering the low abstraction level of VHDL descriptions, an attractive approach used by software developers when targeting an FPGA implementation, is the use of software programming languages followed by the automatic conversion to VHDL. Problems appear when circuits with a certain level of optimization are expected. In this case programmer needs to use special keywords and special instructions to be used by the high level software compiler to produce a more optimized code. Depending on the design complexity, it may be harder to realize than developing the system straight in VHDL. Nested loops, the variety of data types, and some arithmetic operators are examples of restrictions for the translation. The main point is that a compiler for a high level language can translate the program to a low-level language (assembly language), which has similar instructions, making the translation process not too difficult. On the other hand, synthesis tools have to translate the high level programs to a much reduced set of low level "instructions". For instance, while a compiler can use a *je* (conditional jump in assembly) instruction as the target for the translation of a high level *if* constructor, a synthesis tool may have to use D flip-flops and multiplexers.

Over the years some "programming language" to "HDL" translators have been developed, but with no major impact in the market. Developers with a software background starting in the field of HDL design, usually try the way that appears to be the easiest one, which is the use of a language translator. As soon the design challenges start, and the developer must learn very specialized commands and constructions in order to produce a proper code suitable for the chosen translator, so there is a realization that learning a hardware description language may be a better option. Learning a new programming paradigm is not easy, but the syntaxes of languages like Verilog or VHDL are not so different, for instance, from the C language. In addition, a C developer has to use only a subset of the language, because of the synthesis tools limitations. For example, the translation of dynamic structures (pointers) from C to VHDL is not useful, as this class of data type is not suitable for static hardware generation.

1.8 Laboratory Assignment

The laboratory objectives are:

- practicing the use of basic logic gates in the design of a digital circuit;
- to use a design entry tool based on schematic editing;
- to have a first contact with an EDA tool;
- to have a basic understanding of a complete FPGA design flow, starting from the schematic design entry step, performing the circuit simulation, and prototyping it in an FPGA development board.

Logic Gates

Figure 1.10 shows the basic logic gates: OR; AND; XOR; and NOT. The gate functionality is also described through different textual representations, and by its truth table.

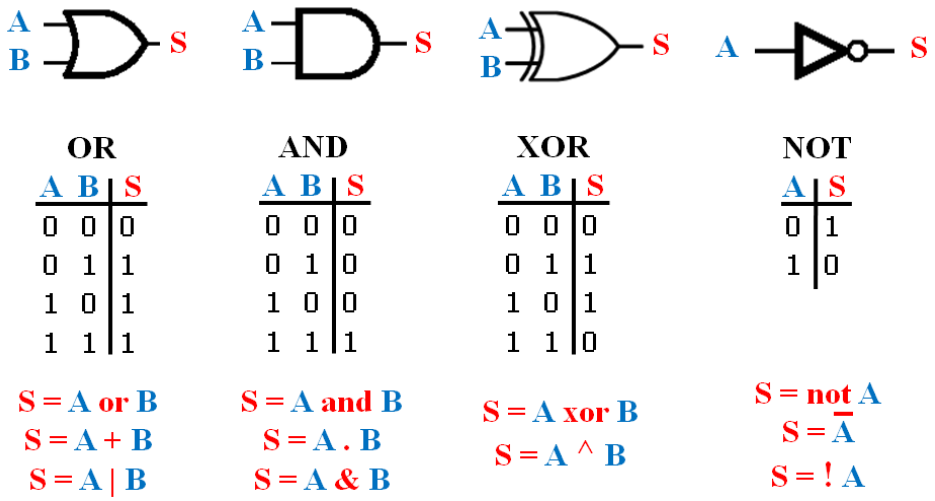

Figure 1.10. Basic logic gates.

A digital circuit can be designed as a collection of logic gates, connected in order to perform the required function. For instance, in Figure 1.11 it is shown the design of a circuit that implements an add operation. This circuit is called a “half-adder” and it is used for the summation of the least significant bits (LSB) of two n-bits operands, providing as a result the sum and a carry. Figure 1.11 shows the sum of two 4-bit values, and the half-adder circuit used to add their LSBs.

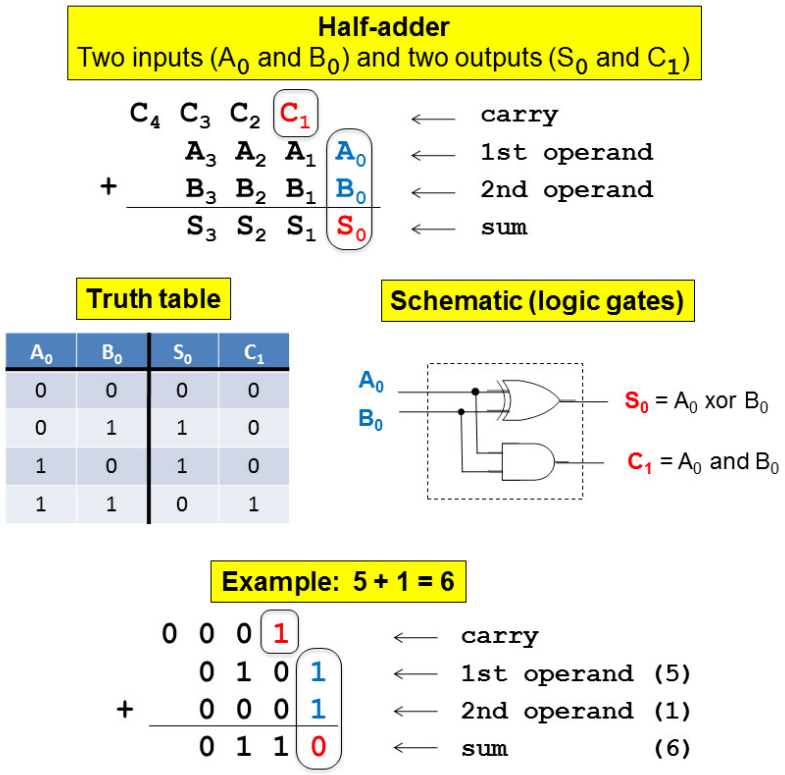

Figure 1.11. Half-adder circuit implemented with an AND gate and an XOR gate.

Laboratory Session

The tasks to be completed in this laboratory session, using Altera's Quartus II EDA tool, are as follows:

- Create the half-adder circuit shown in Figure 1.11, using the schematic design entry tool.
- Perform the synthesis (logic and physical).
- Perform the functional simulation.
- Fix simulation and synthesis errors.
- Prototype the half-adder circuit in the development board, in order to check its functionality.

For those who do not have the Quartus II software installed, there is a free version available on Altera's website at http://www.altera.com

Quartus II 12.1 was the latest version available when this book was written, and this version has been used in all laboratory sessions.

In this first laboratory session, the idea is to follow the step-by-step instructions in order to get used to the EDA tool facilities and resources. This tutorial can be used as a guide to the next chapters as well.

Step 1 – Creating a new project

Start Quartus II, close all initial windows, and open the menu *File –> New Project Wizard*, as shown in Figure 1.12.

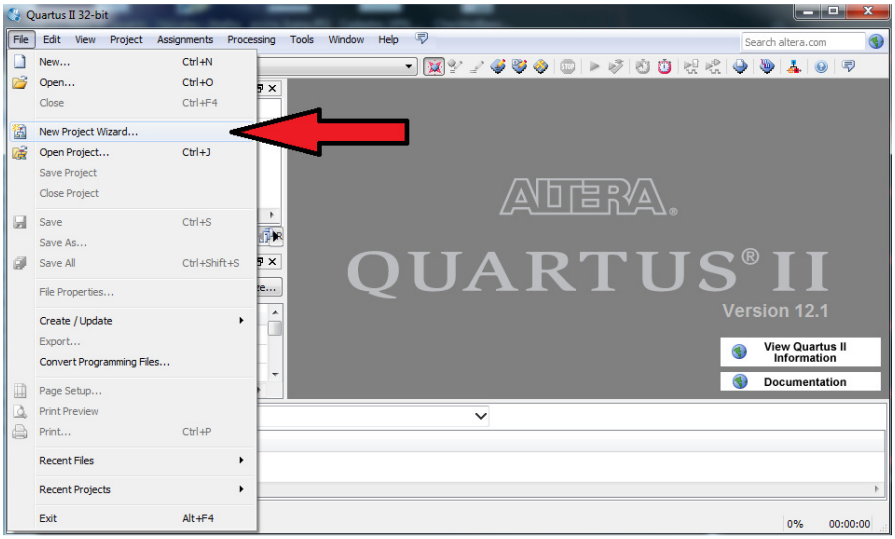

Figure 1.12. Quartus II – New Project Wizard.

Next, choose the project's folder and name. In a VHDL design, it is important to provide the top-level entity's name. Do not use names or paths with spaces or special characters (e.g., à, ç, …) in any of the text boxes.

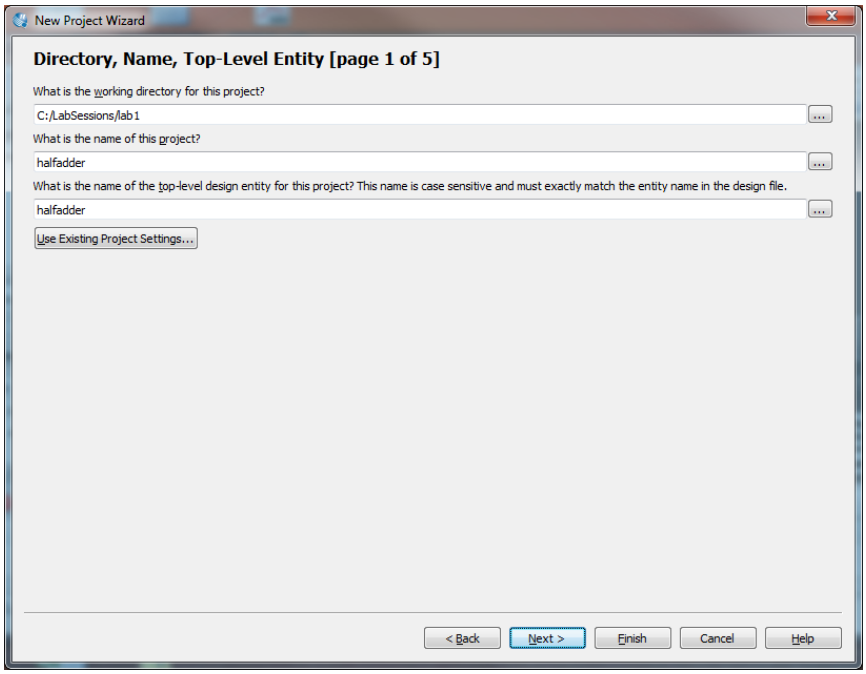

Figure 1.13. Directory, Name and Top-Level Entity.

In “page 2 of 5”, just click on the “Next” button. In “page 3 of 5”, choose the DE2’s FPGA, which is the Cyclone II (“Family”) EP2C35F672C6 (“Device”), as shown in Figure 1.14. In Figure 1.15, choose ModelSim-Altera as the simulation tool.

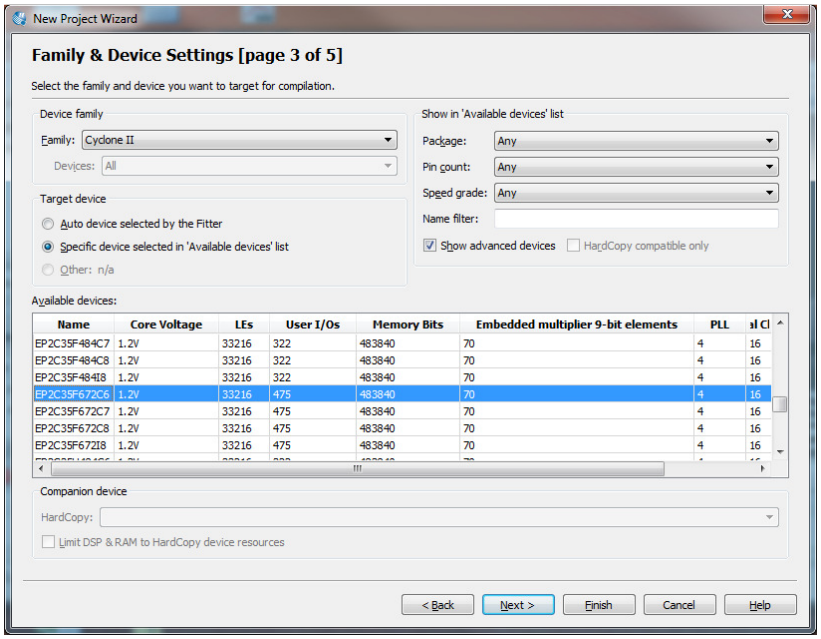

Figure 1.14. Selecting Altera FPGA - Cyclone II EP2C35F672C6.

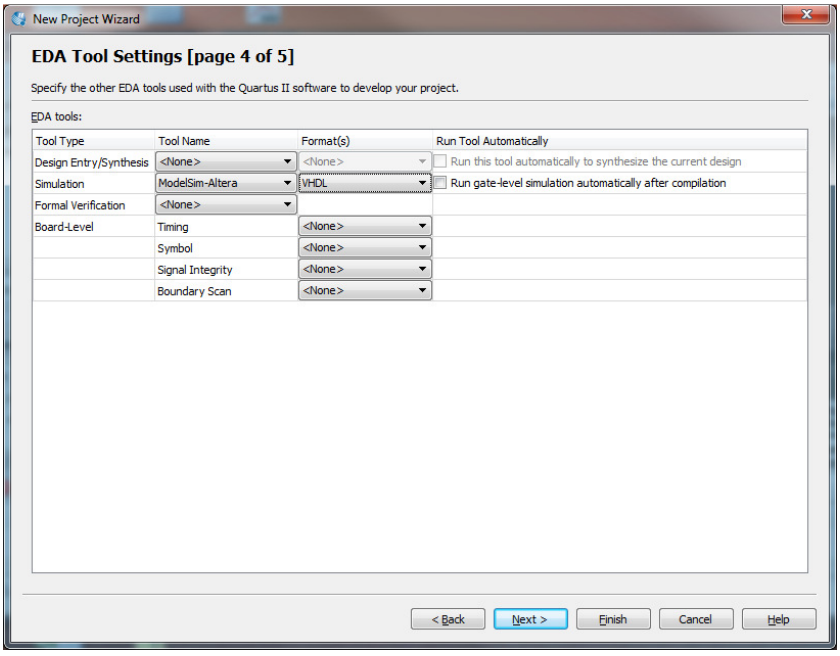

Figure 1.15. Selecting the Simulation Tool – ModelSim-Altera.

In “page 5 of 5” it is shown a summary of the new project. Just click “Finish”.

Step 2 – Design entry (schematic)

Chose *File –> New* and *Block Diagram/Schematic File*, as shown in Figure 1.16. This will open an empty editor, used to create the new circuit schematic.

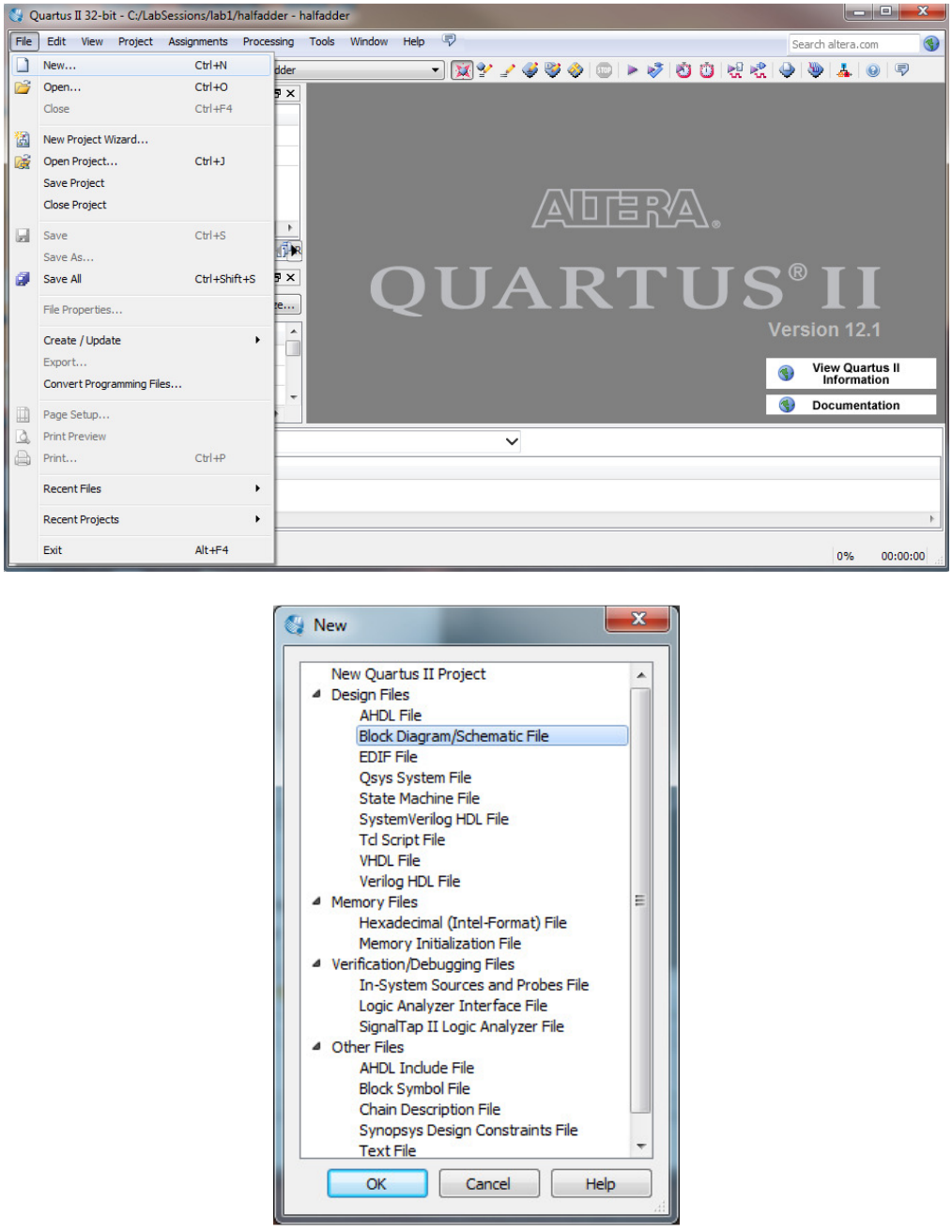

Figure 1.16. New Schematic File.

The schematic to be created should implement the half-adder circuit shown in Figure 1.11. The schematic editor tool has several pre-defined components stored in libraries. The logic gates are available in the Primitives/Logic library. To add a logic gates to the schematic, click on the “Symbol Tool” icon highlighted in Figure 1.17.

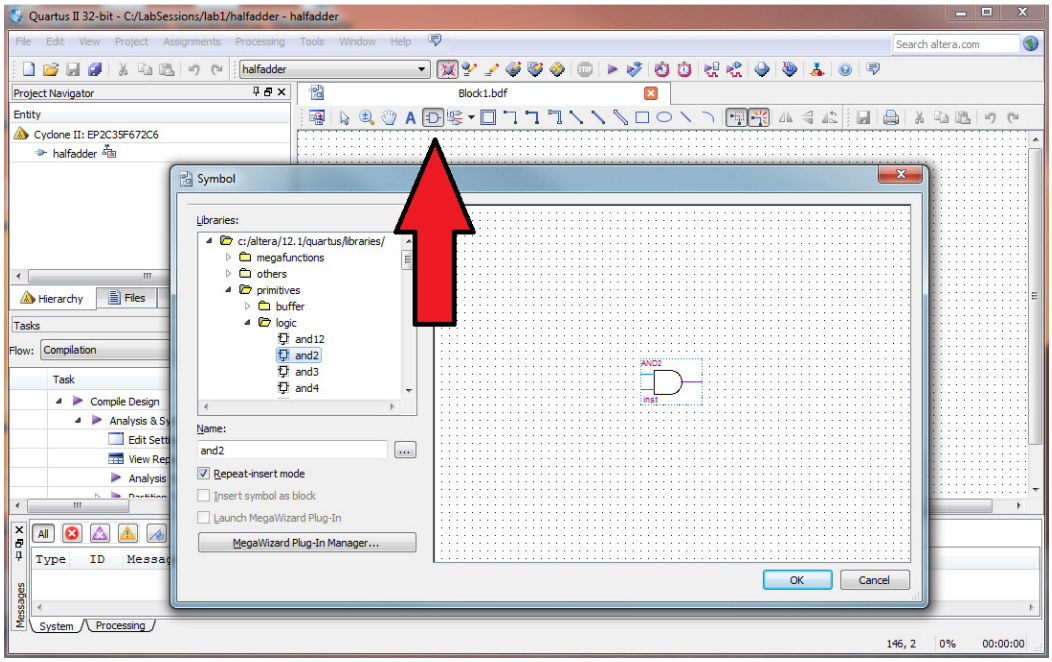

Figure 1.17. Symbol Tool icon.

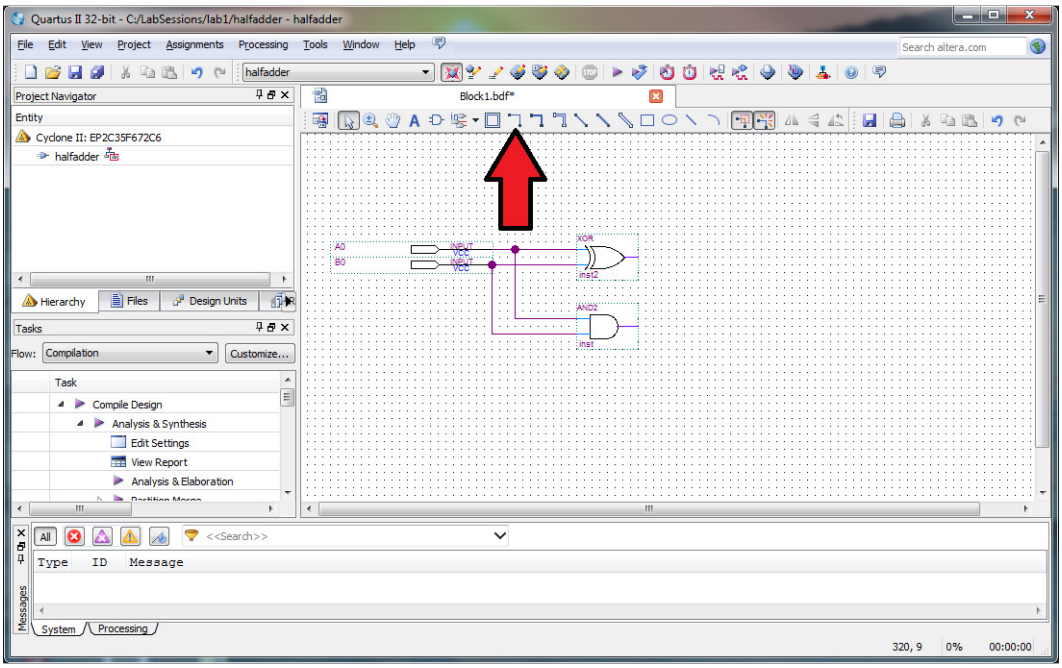

Figure 1.18. Schematic entry.

To connect the gates, use the "Orthogonal Node Tool" as indicated in Figure 1.18. The input and output pins, can be found in library Primitives/Pin, as shown in Figure 1.19. The pin names can be renamed, as shown in Figure 1.20, completing the half-adder design.

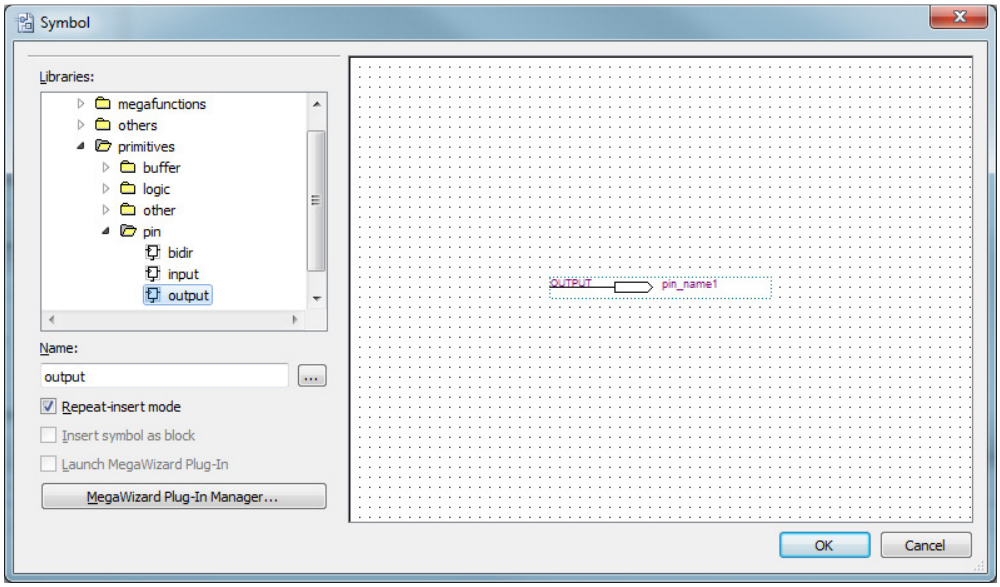

Figure 1.19. Input and output pins.

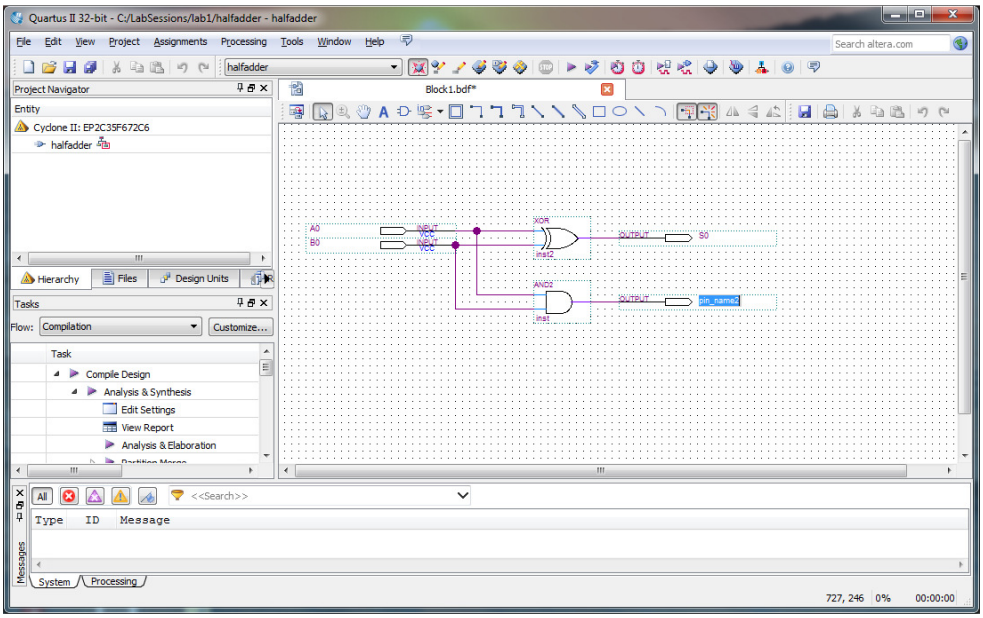

Figure 1.20. Renaming the input and output pins.

When done, save the schematic design: *File –> Save* (or just Ctrl-S). A window will open asking for the schematic's name, and for a place to save the design. The default name is *halfadder.bdf*, and the folder is *c:\LabSessions\lab1*.

Step 3 – Synthesis

In the synthesis step, an analysis is performed in the schematic design and a circuit is generated targeting the selected FPGA device. To start the synthesis, choose *Processing –> Start Compilation* (or just type Ctrl-L). The 11th icon from right to left in the menu bar in Figure 1.21 can also be used to start the synthesis process. At the end of the synthesis, in case of errors found, go back to the schematic design, and check for bad connections. In case of success, the next step is the simulation. Figure 1.22 shows a summary of the synthesis process, listing the four pins used, and two logic elements.

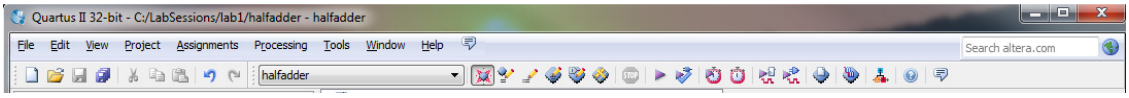

Figure 1.21. Quartus II menu bar.

Step 4 – Simulation

Quartus II (v. 10 and following) does not have its own simulator, and ModelSim, a third party software tool is used. The ModelSim-Altera tool is installed together with Quartus II, but before the first simulation it is recommended to verify that the tool is properly configured. In Quartus II, as shown in Figure 1.22, open *Tools –> Options*.

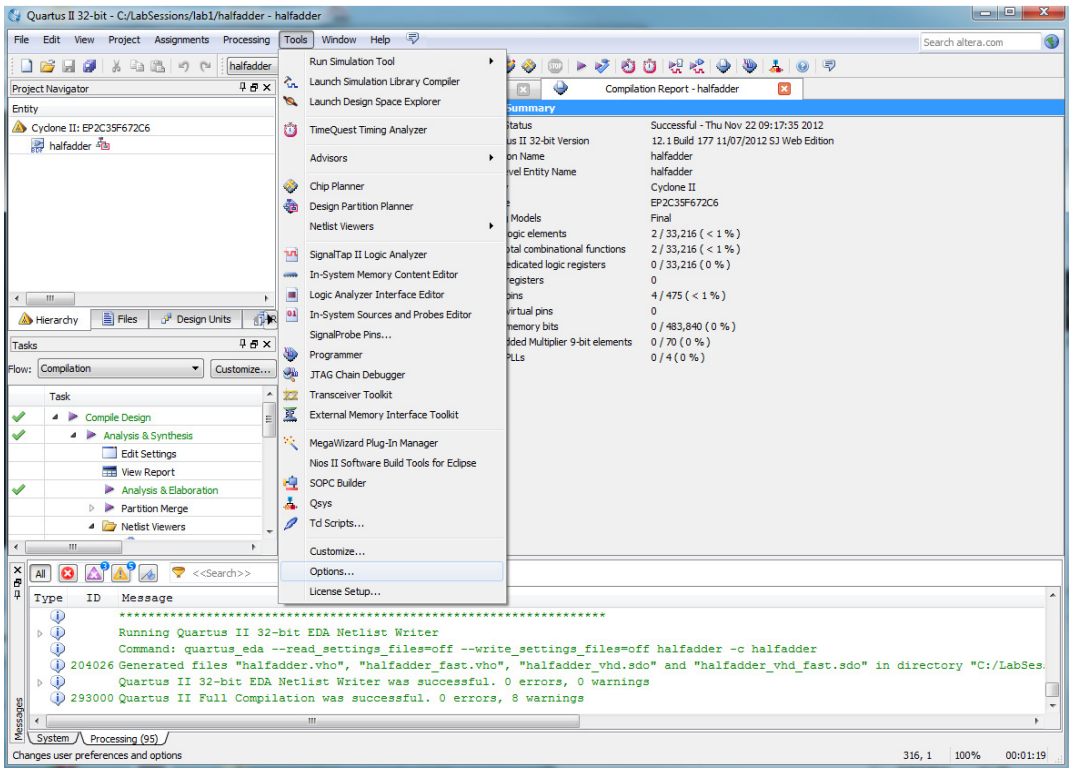

Figure 1.22. ModelSim-Altera installation in Quartus II.

Click on EDA Tool Options, and type ModelSim path in the text box shown in Figure 1.23.

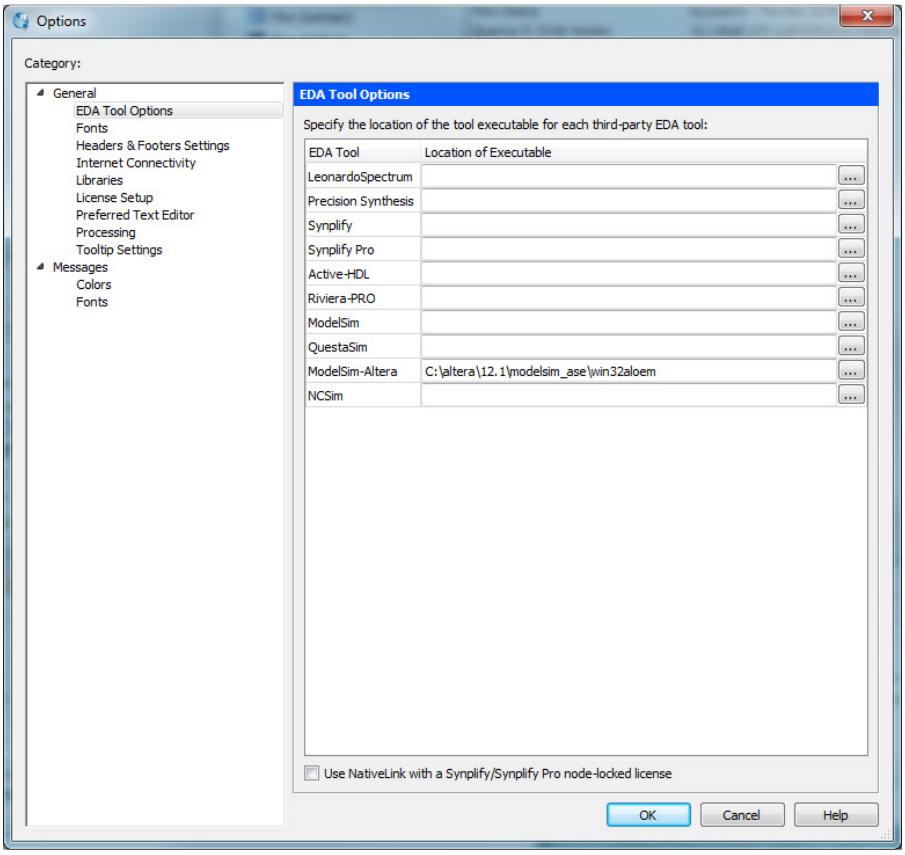

Figure 1.23. ModelSim-Altera path configuration.

To start the simulation tool, select *Tools –> Run Simulation Tool –> Gate Level Simulation* (or press the 6th icon from right to left shown in Figure 1.21). This will run ModelSim software.

When ModelSim starts, just close the presentation window, and initiate the simulation by putting ModelSim in "simulation mode": select *Simulate –> Start Simulation*. The "Start Simulation" window in Figure 1.24 will show. This window has several tabs, but for this functional simulation just the first tab (Design) will be used. The Design tab lists all designs ready for simulation, including the half-adder circuit that has been synthesized.

The target design is located in the "work" library. Open this library clicking on the "+" sign. As a next step, click on "halfadder Entity" to select this module for simulation, as shown in Figure 1.24. Click OK to start the simulation, and ModelSim will open several new windows on the left side of Figure 1.25. The "sim" window lists the hierarchical view of the circuit, and it is used in order to select the signals to be shown during the simulation. When the halfadder module is highlighted, the Objects window lists the signals that can be observed and changed in the simulation process. The signals of interest for the half-adder simulation include A0, B0, S0 and C1.

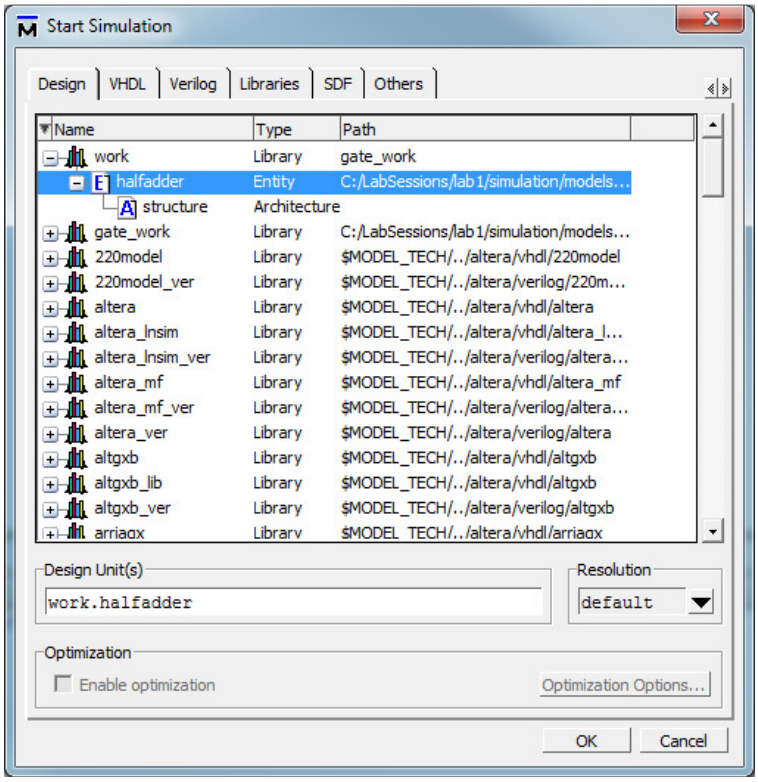

Figure 1.24. Start Simulation window.

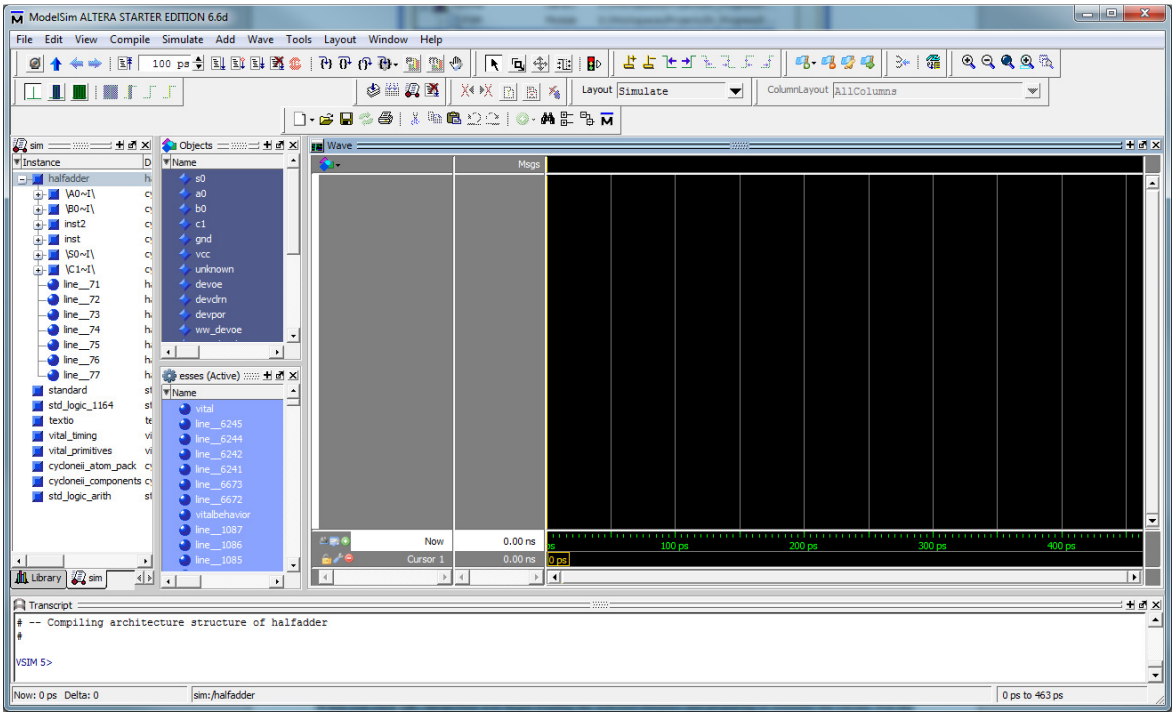

Figure 1.25. Simulation environment.

As shown in Figure 1.26, to add signals to the simulation window, select the signals in the Objects window using the mouse pointer, and with a right click select *Add –> To Wave –> Selected Signals* from the pop-up menu. After have added signals A0, B0, S0 e C1, they will show in a waveform window.

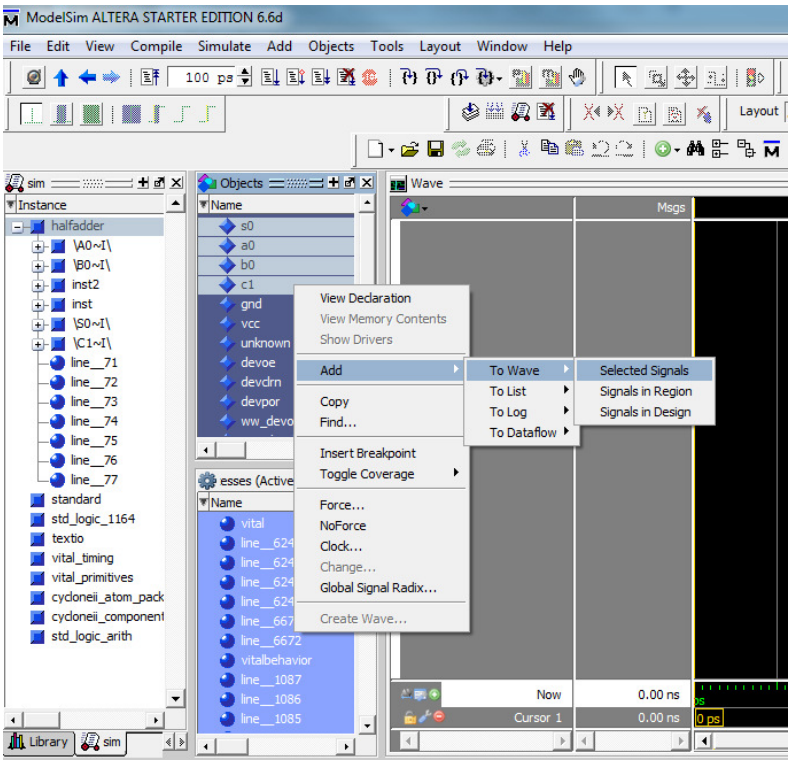

Figure 1.26. Selecting signals for simulation.

In ModelSim the simulation process can be done manually or through scripts. The script language is relatively intuitive, but for the basic circuits introduced in most laboratory sessions it would be overwhelming. The manual option is described next.

The waveform window can be undocked using the button in the upper right hand corner, as shown in Figure 1.27. To organize the signals (it is optional), place the mouse on the A0 signal, hold down the left button, and drag the signal to the desired position.

To set a signal to 0 or 1, right click on the signal and select the Force option from the pop-up menu. In Figure 1.27, the B0 signal is "forced" to 1 (one). Next, repeat the operation for signal A0, but forcing it to 0 (zero). Set the simulation run length to 100 ps, and press the Run button. This will make the simulation run for 100 ps. In Figure 1.27 there is an arrow pointing at the Run button. The remaining simulation buttons are:

- Restart simulation button – it is located on the left of "Run length" (100 ps);
- Run length text box - defines for how long the simulation will run;
- Run button - runs the simulation for the duration specified in the Run length text box;
- Continue run button - used to resume a simulation;
- Run –All button - used to start a simulation;
- Break button - used to interrupt a simulation.

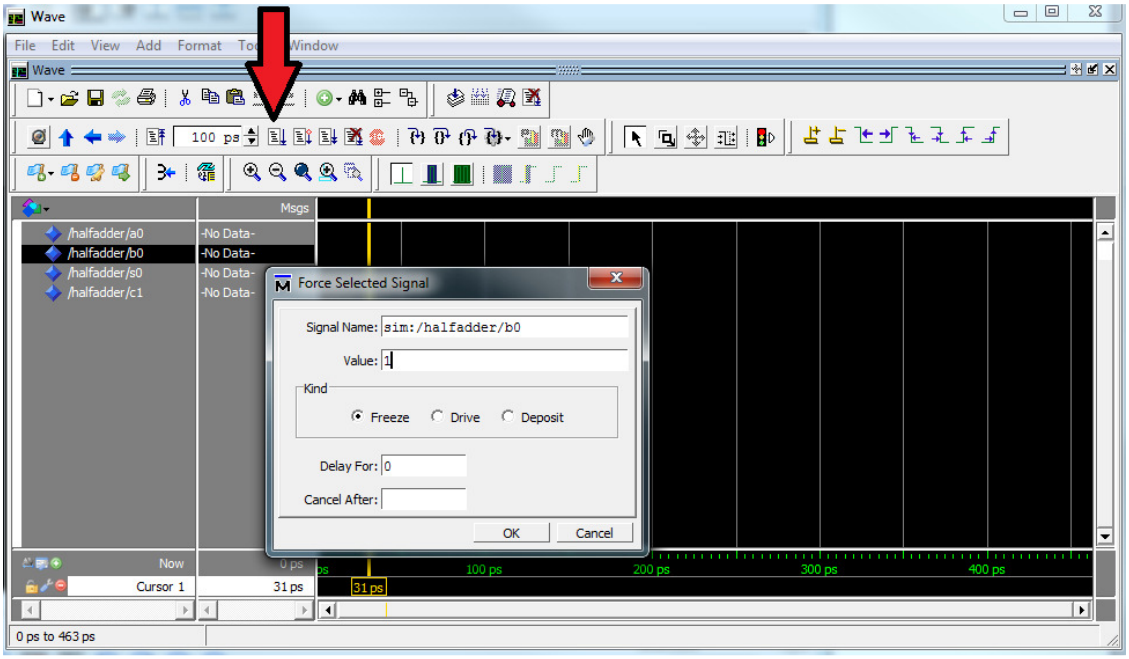

Figure 1.27. Waveform window.

In Figure 1.28, the input signals A0 and B0 are changed to 0 and 1, respectively, and the Run button is pressed again, making the simulation run for another 100 ps period.

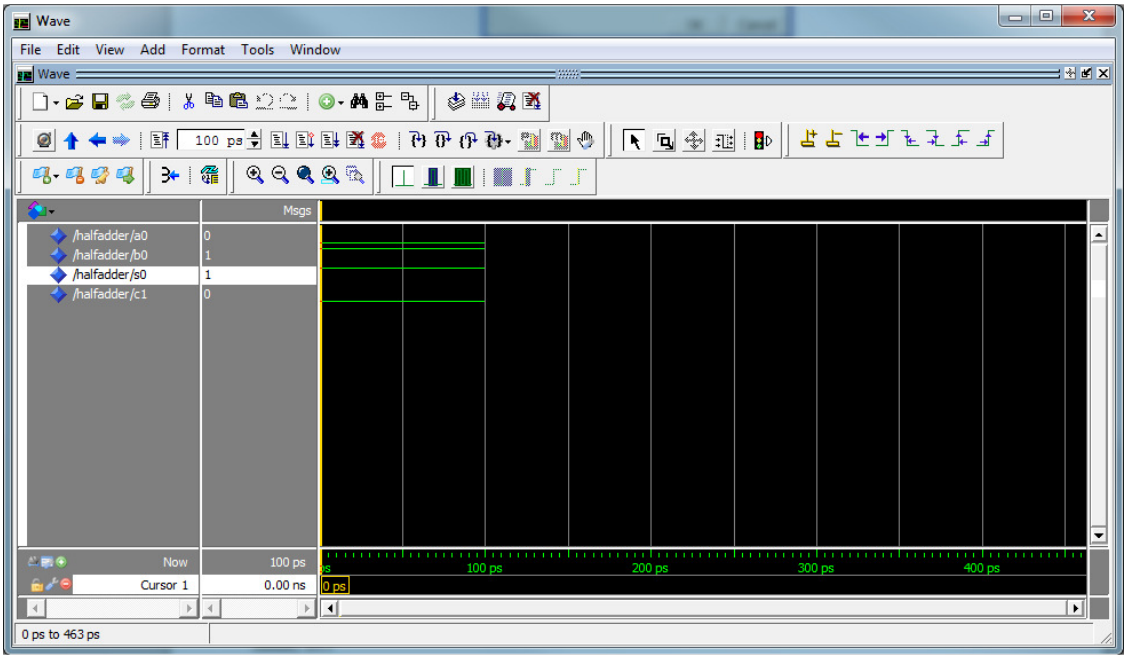

Figure 1.28. A0 = 0 and B0 = 1, resulting in S0 = 1 and C1 = 0.

Figure 1.29 shows the four possible inputs for A0 and B0, according to the truth table in Figure 1.11. The functional simulation shows that the half-adder design presents the expected behavior, but it does not considers time constraints which may result in unexpected delays and, consequently, in a faulty circuit. The timing simulation can be used to check this situation, but in this laboratory session it will not be executed.

The next step is the physical implementation of the circuit, using the FPGA board. Close (quit) all ModelSim windows, and go back to the Quartus II environment.

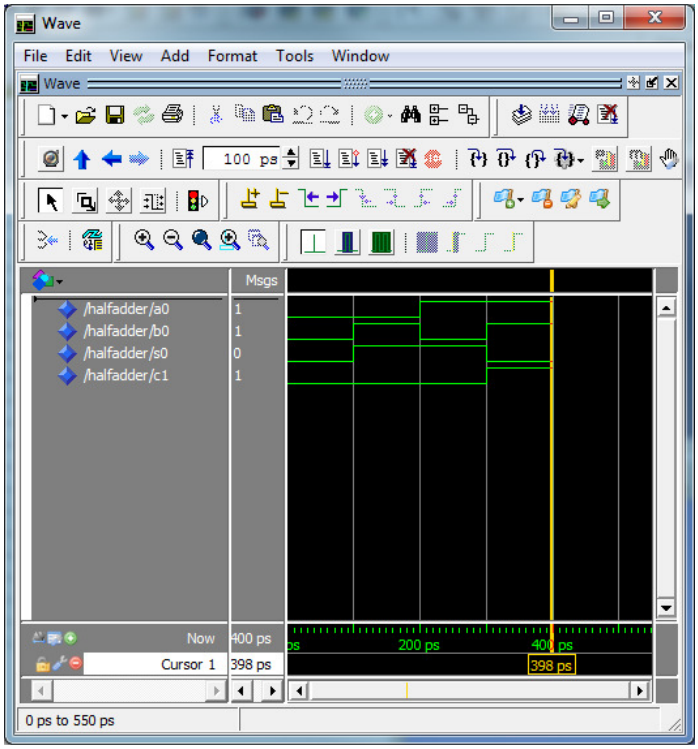

Figure 1.29. All four combinations for A0 and B0.

Step 5 – FPGA prototyping

Using Quartus II, choose menu *Assignments* –> *Pin Planner*. As shown in Figure 1.30 and in Figure 1.31, this tool is used to associate the input and output signals defined in the half-adder schematic design, to the actual FPGA pins. The list of FPGA pins and their connections to the DE2 resources (peripherals) can be found in the board's user manual available on the DE2 folder: DE2\DE2_user_manual\ DE2_UserManual.pdf.

In DE2 user manual there is a list of all available pins, with the respective connection between the FPGA pins and the board's peripherals. For instance, SW[0] and SW[1] switches are connected to the N25 and N26 FPGA pins (see DE2_UserManual.pdf Table 4.1 in page 28). The half-adder output signals can be shown on LEDR[0] (sum) and LEDR[1] (carry), which are connected to FPGA pins AE23 and AF23 (see DE2_UserManual.pdf Table 4.3 in page 29).

Using the Pin Planer tool, perform these pin assignments in the Location column, as shown in Figure 1.31, and close the Pin Planner.

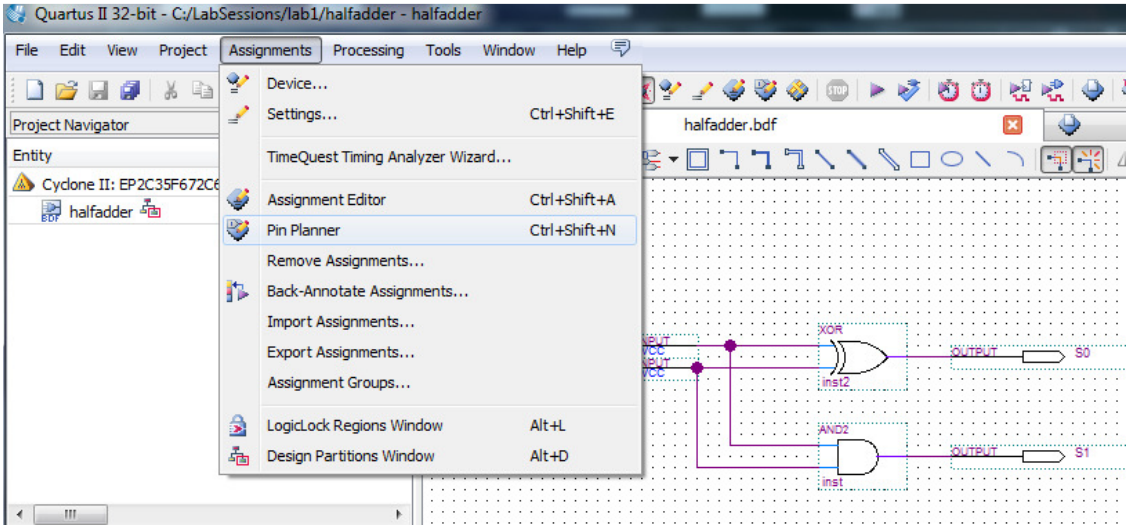

Figure 1.30. FPGA pin assignments.

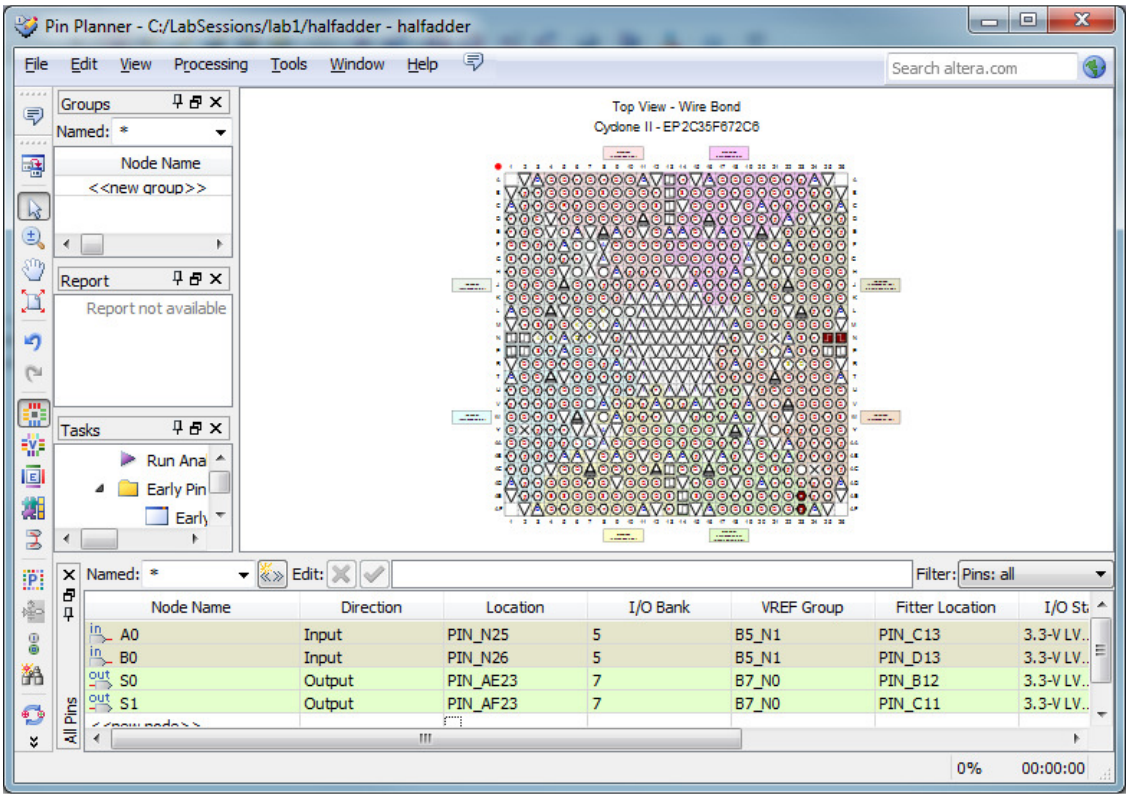

Figure 1.31. Half-adder pin assignment.

Perform a new synthesis (button Compile in Quartus II menu), in order to assign the physical pins to the design. It will generate the final netlist that can be downloaded to the FPGA.

After the user has powered on and connected the DE2 board to the USB (use the "blaster" USB connector), press the red button to switch on the board, and run the programming tool: Menu *Tools* –> *Programmer* (or 4th icon from right to left in Figure 1.21). The window in Figure 1.32 shows the Programmer tool interface. In order to perform a download of a configuration file to the FPGA, be sure to have the "USB-Blaster [USB-0]" message showing next to the "Hardware Setup" button, and that the configuration file "halfadder.sof" is listed in the File column.

The "Start" button is used to perform the download of the halfadder.sof to the FPGA configuration memory.

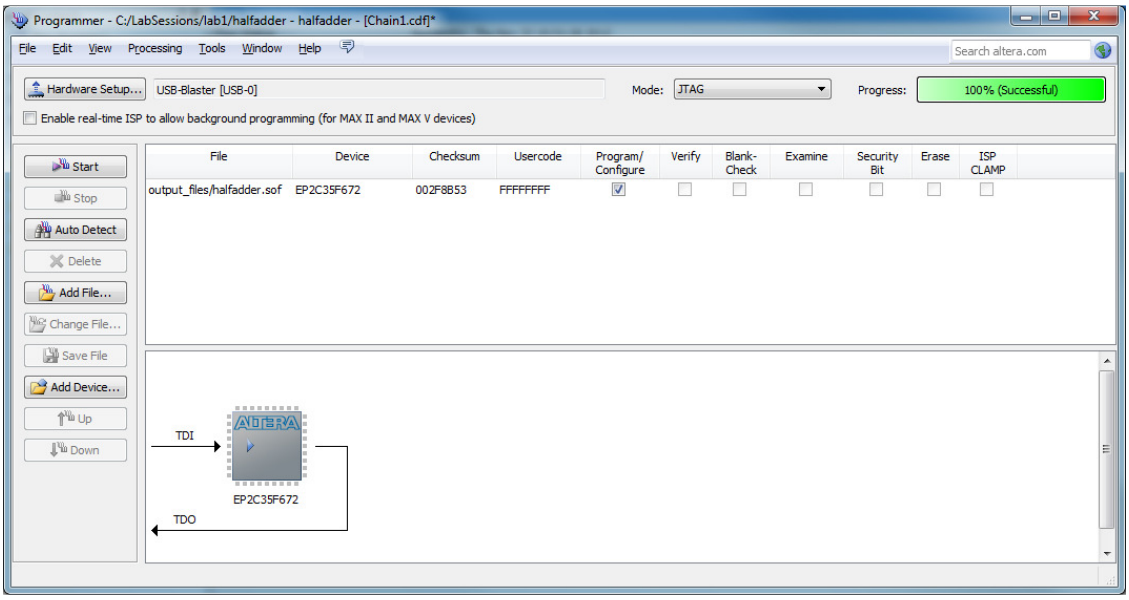

Figure 1.32. Quartus II FPGA programming tool.

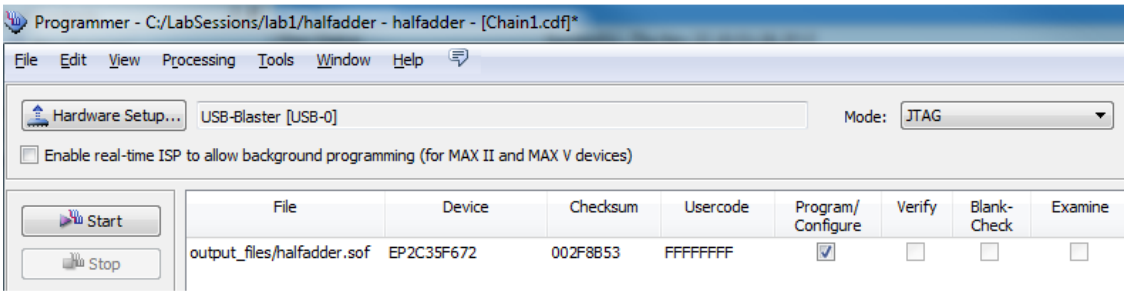

Figure 1.33. FPGA programmer.

Some problems that may happen regarding the FPGA Programmer tool are listed next:

- The USB port has not been found (DE2 board not detected). Solution: start "Devices and Printers" in Windows start menu and proceed with the new driver installation. The DE2 driver is located in the Quartus II folder: C:\altera\12.1\quartus\drivers. For Linux systems, follow the standard procedure for driver installation.
- The USB-Blaster [USB-0] is not shown. Solution: certify that the USB cable is connected to the "Blaster" input. Press the "Hardware Setup" button, double-click on "USB-Blaster", and close the Hardware Setup window.
- The halfadder.sof file is not shown. Solution: press the "Add File" button, go to c:\LabSession\lab1\output_files, and double-click on halfadder.sof

After have finished the download to the FPGA, in order to check the circuit functionality in the board, just switch inputs SW0 (A0) and SW1 (B0) "on" and "off", and make a comparison between the results showed in the red LEDs (Light-Emitting Diode) with the truth table shown in Figure 1.11.

A summary of the activities developed in this laboratory session is as follows:

1. **Project creation *File–> New Project Wizard***
2. **In "project wizard", follow <u>exactly</u> the steps listed in this tutorial. Any oversight may result in errors in the hardware generation for the FPGA.**
3. **Design Entry (schematic) *File –> New –> Block Diagram***
4. **Draw the half-adder schematic (Figure 1.11).**
5. **Compile the circuit (synthesis).**
6. **Perform the functional simulation using ModelSim (no timing information).**
7. **Perform the pin assignment, which is the interface between FPGA's internal signals (half-adder input/output signals) and the board switches and LEDs) – *Assignments – Assignment Editor (or Pin Planer)*.**
8. **Compile the circuit again, in order to generate the final circuit (with pin assignments) to be used in the FPGA. In the physical synthesis (place & route), the logic elements defined in the logic synthesis are placed using the physical resources found in the FPGA. In this process, all connections (routing) between logic elements are performed.**
9. **Temporal Analysis - Analysis of propagation delays of signals after physical synthesis resulting in a report consisting of the expected performance.**
10. **Programming – The synthesized circuit is downloaded to the FPGA - *Tools – Programmer. Hardware Setup – USB-Blaster. Start!***
11. **Testing the prototyped circuit - Figure 1.34 shows the power connector (1), the USB blaster port (2), the inputs SW(0) and SW(1), and the outputs LEDR(0) and LEDR(1). Switch the inputs on and off, and observe the outputs (sum and carry) in the LEDs, according to the truth table in Figure 1.11.**

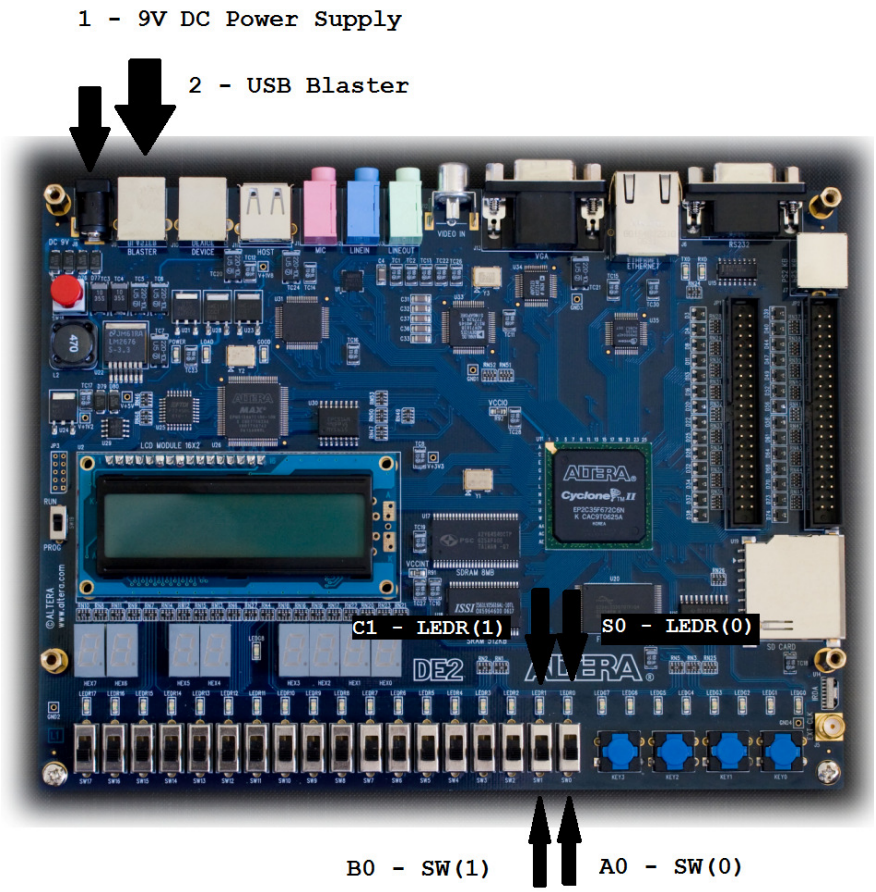

Figure 1.34. DE2 board.