Digital Design and Computer Architecture LU

Design Flow Tutorial

Florian Huemer, Florian Kriebel
{fhuemer, fkriebel}@ecs.tuwien.ac.at
Department of Computer Engineering
TU Wien

Vienna, March 8, 2023

Contents

1   Design Flow

Figure 1.1 outlines the Field Programmable Gate Array (FPGA) design flow, which we will use in this lab course. Table 1.1 summarizes the tools required for each of these steps which are either sophisticated software (e.g. a simulator) or a hardware devices (e.g. a logic analyzer or an oscilloscope).
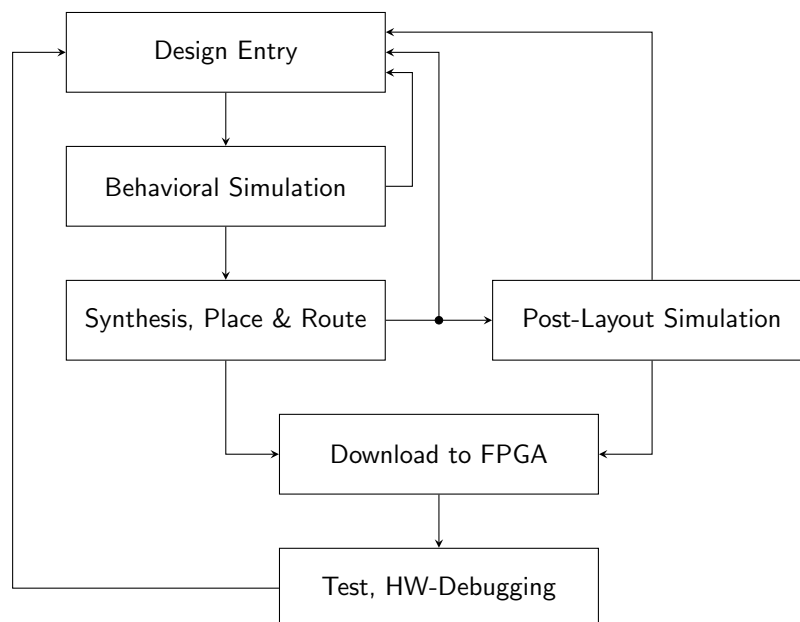

Figure 1.1: The design flow.

Table 1.1: Used tools.

Design Flow Step	Tool
Design Entry	Text editor, Questa/Modelsim, Quartus
Behavioral Simulation	Questa/Modelsim
Synthesis, Place&Route	Quartus
Post-layout Simulation	Questa/Modelsim
Download	Quartus (programmer)
Test & HW Debugging	Logic Analyzer/Signaltap

Design entry   During the design entry you will describe your FPGA design with a high-level hardware description language. In this lab course we will use VHDL[1] for this purpose.

Behavioral Simulation   The behavioral simulation is the most important step in verifying the correct functionality of your circuit description. First the simulator compiles your VHDL files and indicates syntax errors. If the compilation runs without errors, the behavior of the circuit design, more specifically the traces of external and internal signal over time, can be represented graphically.

---

[1]VHDL: Very high speed integrated circuit Hardware Description Language

In contrast to the *post-layout* simulation, the behavioral simulation does not take real signal and gate delays into account, but rather simulates an "initialized" version of the circuit.

Synthesis, Place & Route   During the synthesis step your VHDL circuit description is analyzed by the synthesis tool and transformed into a gate-level netlist. The netlist elements can then be placed on the FPGA's configurable cells and connected by suitable interconnect lines. Finally a bitstream file is generated containing the configuration information for the FPGA. In this lab course we are using Altera FPGAs, for which the Altera Quartus Design Software is used for the aforementioned steps[2].

Post-layout Simulation   The post-layout simulation is performed on the netlist generated in the synthesis step. Additionally, timing information, which are provided by the Place&Route-tool can be used for simulating the final circuit with correct signal timings. A post-layout simulation is useful for resolving timing problems in your circuit. For simple FPGA designs this type of simulation is usually not necessary.

Download   The download transfers the bitstream data to the target board for configuring the FPGA.

Test & HW Debugging   Finally, the configured FPGA can be tested. In some cases it might be necessary to attach a logic analyzer to the FPGA for verifying the correct functionality of the HW circuit or for debugging errors, which can not be reproduced during simulation.

1.1   Directory Structure

The source code provided for every task assignment is organized with the directory structure described in Table 1.2. We highly recommend you to maintain this structure and add new files you create in the designated subdirectories.

Table 1.2: Directory structure.

Directory Path	Contents
./src	VHDL design files
./tb	Source files for testbenches, Questa/Modelsim scripts
./scripts	Tcl scripts for simulation or generating Quartus projects
./quartus	Quartus project for target board
./modelsim_beh	Questa/Modelsim project for behavioral simulation
./modelsim_post	Questa/Modelsim project for post-layout simulation

---

[2]A free version of Quartus and Questa/Modelsim called ModelSim Altera can be downloaded form `https://www.altera.com/products/design-software/fpga-design/quartus-prime/overview.html`

2 Behavioral Simulation

To start Questa/Modelsim GUI enter the following command in the terminal:

```
vsim&
```

2.1 Creating a new Project

A new Questa/Modelsim project can be created by clicking the menu item *"File"* ⇒*"New"* ⇒*"Project..."*. Subsequently the dialog which can be seen in Figure 2.1a opens. Enter the project name and select the path of the behavioral simulation directory as *project location*. For the other settings, just use the default values.
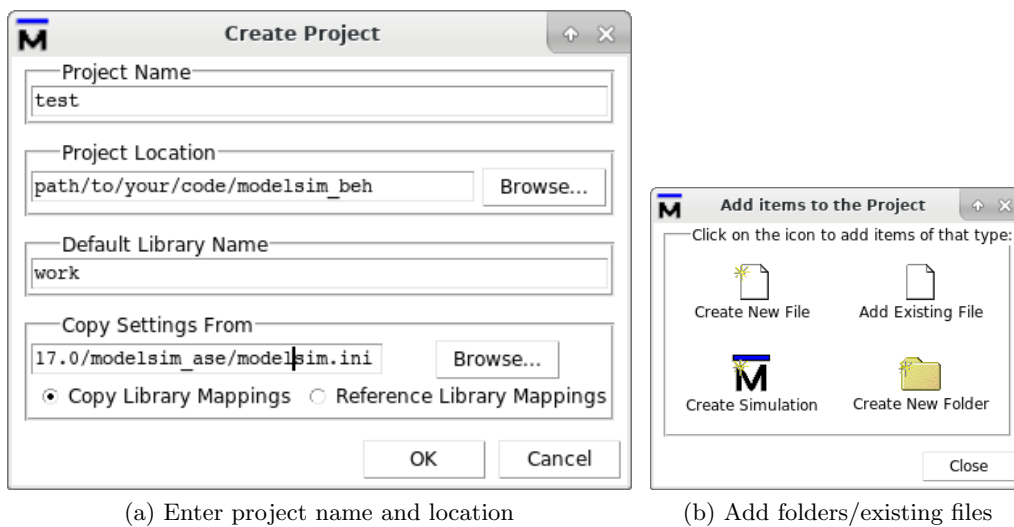

(a) Enter project name and location

(b) Add folders/existing files

Figure 2.1: Creating a new project.

In the next dialog, Questa/Modelsim allows you the add items like source files or folders to your project (Figure 2.1b). For a good structure of your project we recommend you to add the following folders:

- Source folders: Create one folder for every module/IP core.

- Simulation folder for adding all your testbench files.

- Script folder for Tcl script files.

These folders will not be created on the filesystem. They are just used in the Questa/Modelsim GUI for structuring the project files. Once you have created the necessary folders, you can add VHDL source files by clicking *"Add existing File"*. Figure 2.2 shows the dialog for adding files. Enter the file name, select the appropriate folder and make sure the option *"Reference from current location"* is checked. In Figure 2.3 you can see the project view after all folders and files have been added. Of course, you can also easily add or remove folders/files by right-clicking in the project tree. As you can see in Figure 2.3 the script folder is still empty. We will explain the use of Questa/Modelsim scripts later.
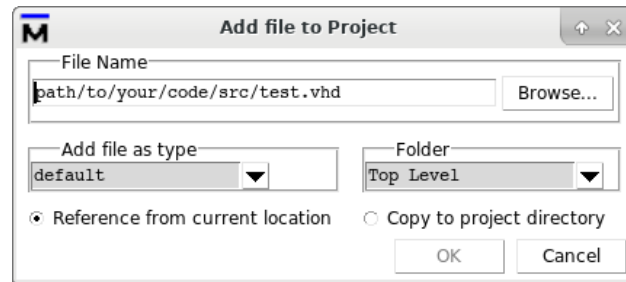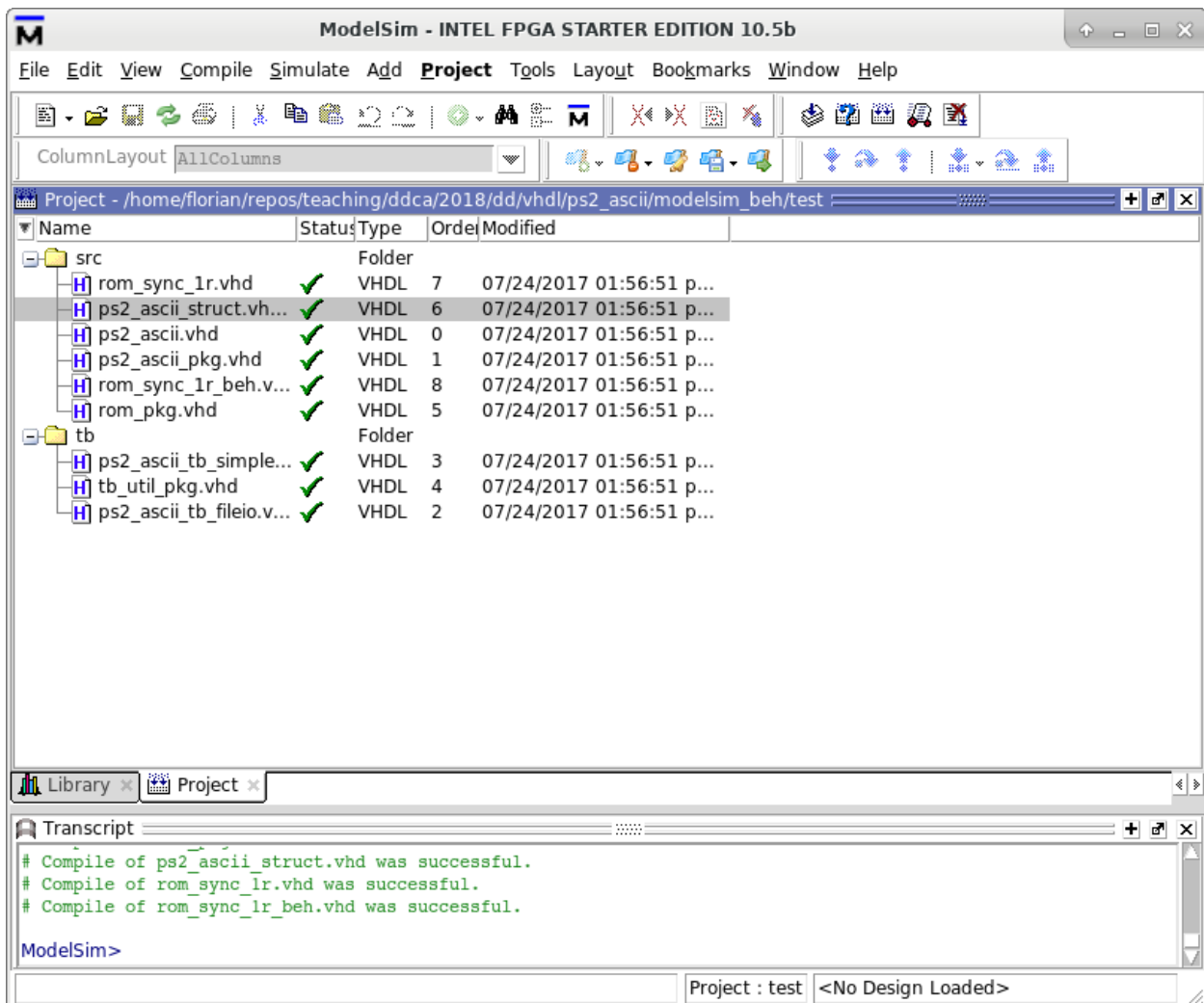
Figure 2.2: Adding a VHDL file

Figure 2.3: Questa/Modelsim project with files added

2.2   Compiling the VHDL Files

In the following we will use the PS/2 to ASCII converter as an example (refer to the IP cores manual for details on this core). Before the VHDL files can be analyzed and compiled it is necessary to setup the right compilation order. In a VHDL design consisting of multiple files there are usually some VHDL files, which depend on other VHDL files (e.g., a file containing constant declarations will

have to be compiled before other files using these constants). Fortunately, Questa/Modelsim is able
to resolve these build dependencies itself. Simply click on the menu item "*Compile*" ⇒"*Compile
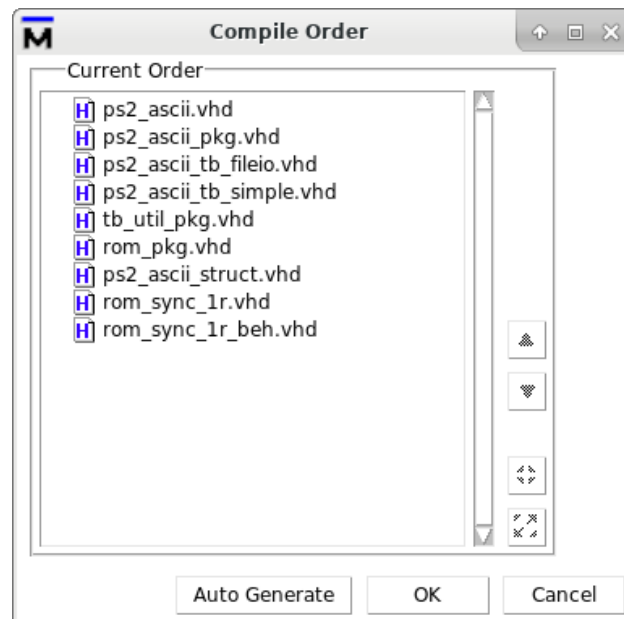Order...*". In the next dialog hit the button "*Auto Generate*" (see Figure 2.4).

Figure 2.4: Compile order

Once the file order is fixed, you can compile selected VHDL files with a right-click on the file
name or with the "*Compile*" button in the toolbar. You can also compile all source files of the
project at once by clicking on the menu item "*Compile*" ⇒"*Compile all*" or the corresponding
button in the toolbar.

2.3   Performing the Simulation

When all VHDL files have been compiled successfully, the behavioral simuation can be started.
Click on the menu item "*Simulate*" ⇒"*Start Simulation...*". In the next dialog, which can be seen
in Figure 2.5, simply select the name of the testbench and click "*Ok*".

Now the main window of Questa/Modelsim changes its appearance and shows a hierarchical
view of the design to be simulated. With the treeview on the left side of the main window you can
navigate through the component instances of your VHDL design. The *object inspector* on the right
side shows the input/output ports and the internal signals of the currently selected component.
The signals you want to be traced can now be added to the waveform viewer by selecting them in
the *objects list* as illustrated in Figure 2.6.

Upon adding the signals, the waveform viewer should open automatically. Otherwise you can
open it by clicking on the menu item "*View*" ⇒"*Wave*". You should see the signals you just added
on the left side of the waveform viewer (see Figure 2.7). In the following we will explain the most
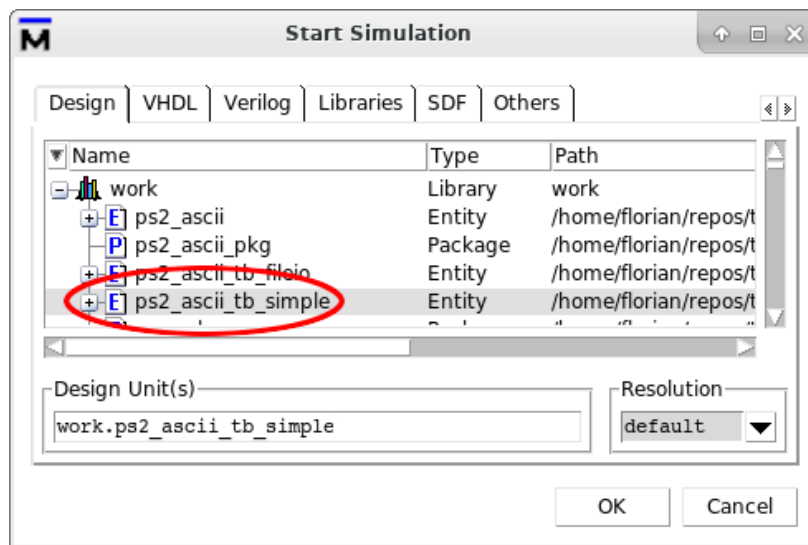important toolbars of the waveform viewer.
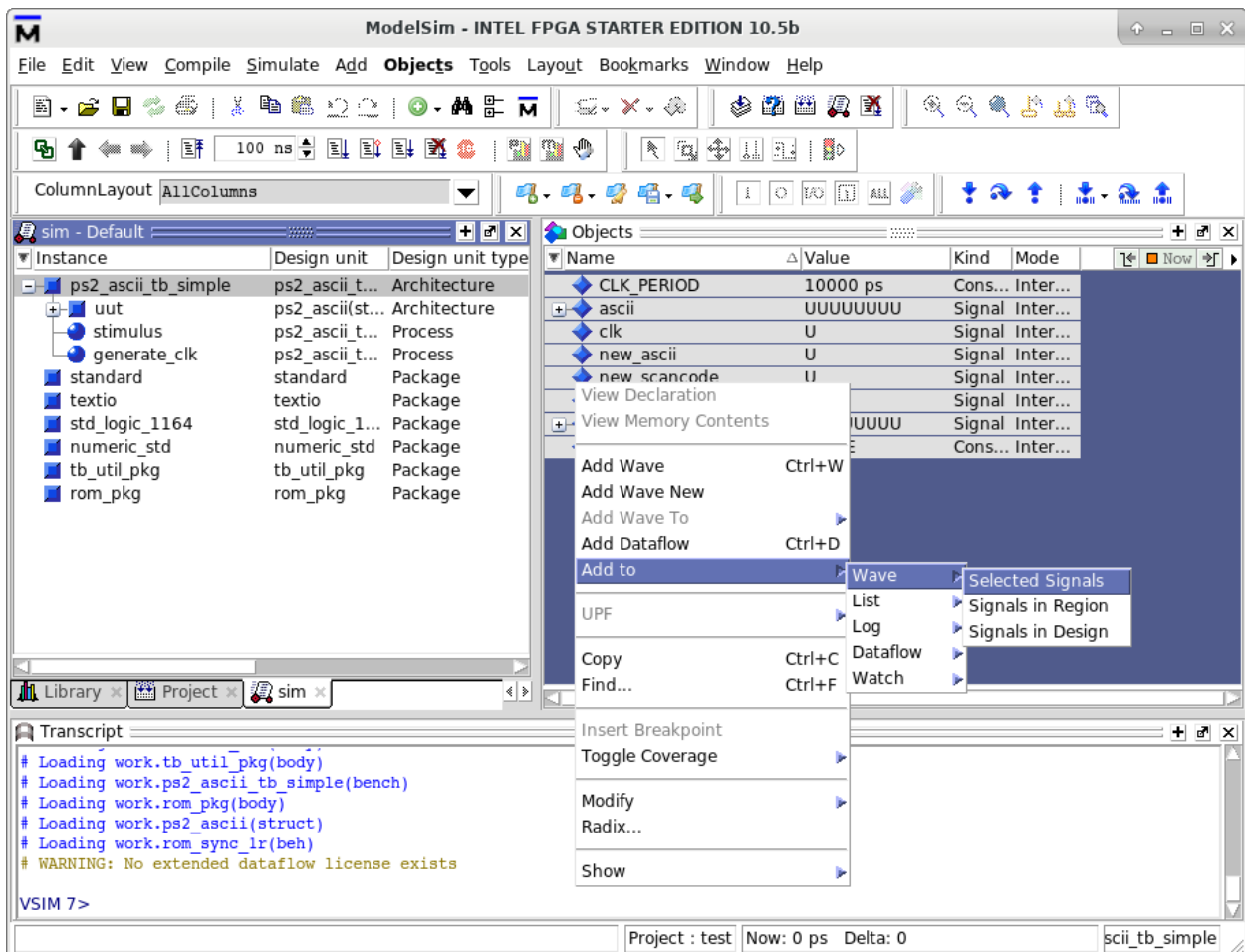
Figure 2.5: Starting the simulation

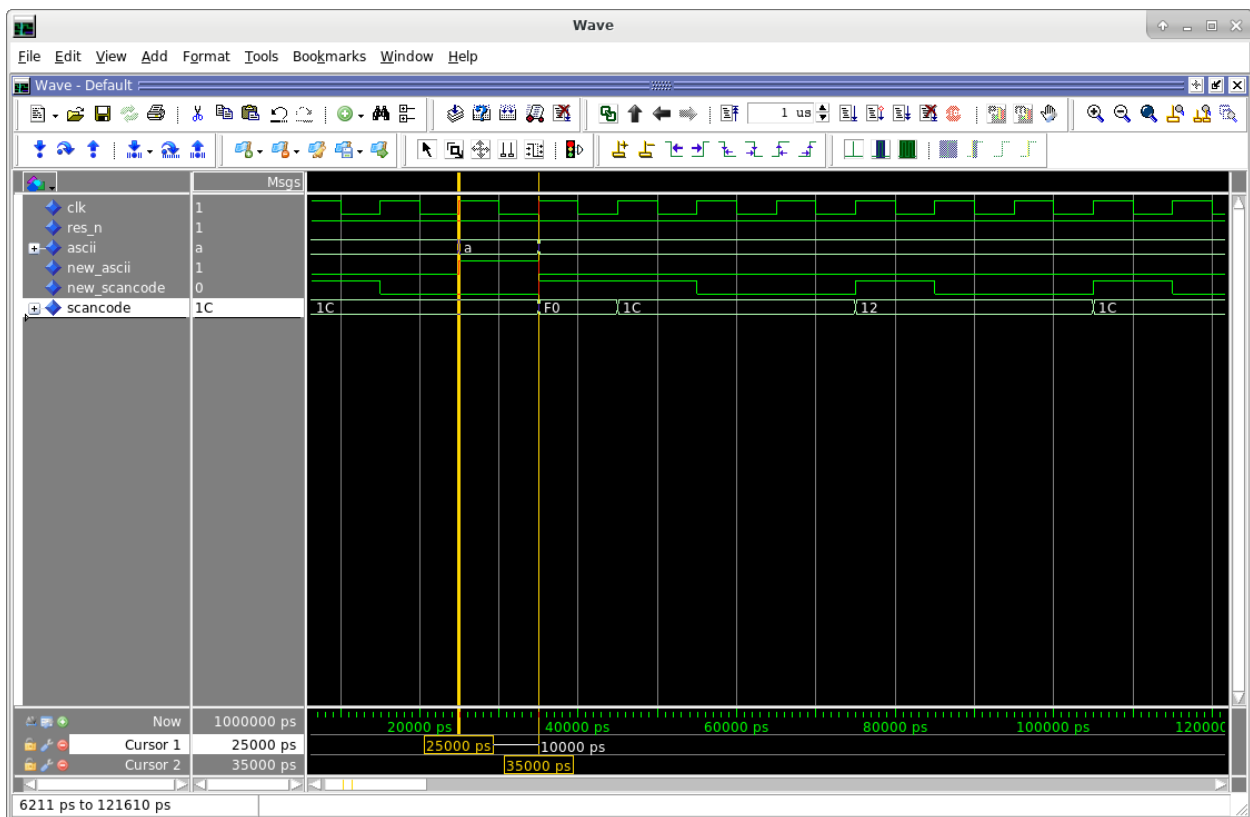Figure 2.6: Add signals to the waveform
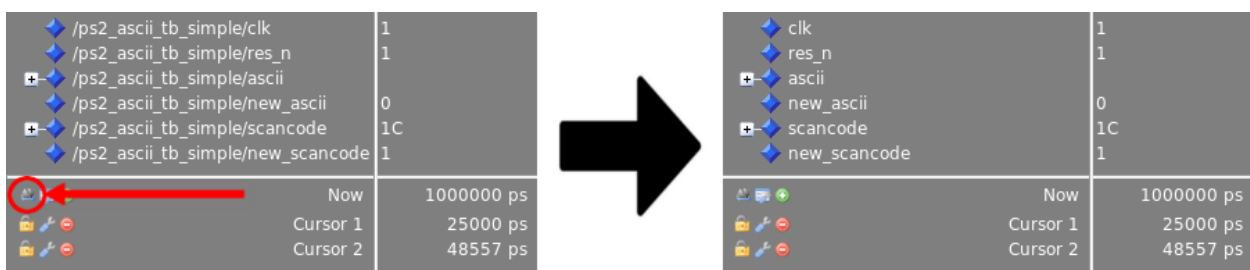
Figure 2.7: The waveform viewer

Figure 2.8: Changing the way signal names are displayed

- Simulation:
  To run the simulation, we first have to adjust the simulation time. This can be done with the small input field in the simulation toolbar. Use *ns, us* and *ms* to specify nano-, micro- and milliseconds, respectively. The button to the right of this input field then runs the simulation for the specified time. The button to the left resets the simulation and deletes all recorded simulation data. Figure 2.7 shows a simulation, which has been run for 1 μs, as can be seen in the statusbar.

- Zoom:
  This toolbar can be used to zoom in and out of the waveforms. The third button is particularity useful, since pressing it zooms to such a level that all recorded data is shown in the available space in the window. Should you ever get "lost" in your signal traces, this function offers you a quick method to reset the view.

- Cursors:
  This toolbar contains buttons for adding (first button), removing (second button) and positioning cursors on the signal traces of the waveform viewer. In Figure 2.7 two markers have been added for measuring the period of the clock signal drawn in the first line of the waveform. It can be seen that the clock period is $10ns$.

Another important feature of the waveform viewer is the possibility to change the display format of signal vectors. The values of a signal vector can be displayed in different representations such as binary, hexadecimal, decimal, ASCII, etc. Just right-click on the signal's name and choose the suitable representation from the "*Radix*" submenu. In Figure 2.7 the signal *scancode* is shown in hexadecimal representation while the signal *ascii* is interpreted as ascii character.

The default way signal names are displayed on the left side of the waveform viewer can be a little bit unclear (see the left side of Figure 2.8), since the full "path" of the signal through the design is used (i.e. here also the name of the testbench is included). This behavior can be changed by using the button marked in the figure.

2.4   Automation and Scripting

Often a simulation needs to run several times until the functionality of the circuit has been verified or an error has been corrected. Therefore it is highly recommended to automate the steps necessary for setting up the simulation.

2.4.1   Tcl Scripts

Fortunately, all the actions described above (compiling, starting the simulation, adding signals to the waveform viewer, running the simulation for a specified period of time, etc.) can be scripted with *Tcl* scripts. These scripts files have the file extension "*.do*". Listing 1 shows a short example for a Tcl script, which executes the tasks mentioned above.

```
1 # compile all source files
2 project compileall
3
4 # start simulation with testbench named "ps2_ascii_tb_simple"
5 vsim work.ps2_ascii_tb_simple
6
7 # add signals to waveform viewer
8 add wave -format logic /ps2_ascii_tb_simple/clk
9 add wave -format logic /ps2_ascii_tb_simple/res_n
```

```
10 add wave -format literal -radix ascii /ps2_ascii_tb_simple/ascii
11 add wave -format logic /ps2_ascii_tb_simple/new_ascii
12 add wave -format literal -radix hex /ps2_ascii_tb_simple/scancode
13 add wave -format logic /ps2_ascii_tb_simple/new_scancode
14
15 # run simulation
16 run 1 us
```

Listing 1: Tcl script for running a Questa/Modelsim simulation

All these commands can also be executed by running them in the command interface of Questa/Modelsim, usually located in the lower part of the window labeled *transcript*. Moreover, all commands you enter through the GUI also show up in this window as Tcl commands. This can be useful, when developing scripts.

   Tip: The waveform viewer offers a convenient function to save the current signal configuration to a Tcl script, to quickly restore the current view on a later simulation run (use "*File*" ⇒"*Save Format*").

   We recommend to create such a Tcl script for every testcase in the simulation directory, which also contains the corresponding testbenches. Then you can add the do-files to your Questa/Modelsim project just like you have added your other source files. In order to execute a script, right-click on the file in the Questa/Modelsim GUI and click on the "*Execute*" item in the context menu.

2.4.2 Questa/Modelsim Command Line Interface

You can also run Questa/Modelsim solely based on scripts, even without creating a project. Listing 2 shows a Tcl script that runs the compilation process of our sample project. The file paths used in this script must be relative to the directory, from where the script is called.

```
 1 vlib work
 2 vmap work work
 3
 4 #compilation order is important!
 5
 6 #compile dependencies first
 7 vcom -work work ../rom/src/rom_pkg.vhd
 8 vcom -work work ../rom/src/rom_sync_1r.vhd
 9
10 #compile source file of the design we want to test
11 vcom -work work src/ps2_ascii_pkg.vhd
12 vcom -work work src/ps2_ascii.vhd
13
14 #compile the testbench and its dependencies
15 vcom -work work tb/ps2_ascii_tb_simple.vhd
16 vcom -work work tb/tb_util_pkg.vhd
```

Listing 2: Tcl (shell) script calling the Questa/Modelsim compiler

   Using the command line interface of Questa/Modelsim this script can be called without the need to start the GUI:

```
vsim -c -do "do scripts/compile.do;quit -f"
```

The `-c` flag tells Questa/Modelsim to start in command line mode. The argument of `-do` allows you to directly pass commands to the Questa/Modelsim Tcl interpreter. Here our compilation script is executed use the `do` command. Afterwards Questa/Modelsim exits `quit -f`. If the quit command is omitted, Questa/Modelsim will not exit but rather wait for further input on the command line.

The script shown in Listing 2 can also be executed directly as a shell script. We will use this to create a convenient makefile-based build and simulation flow.

To start the simulator (and the waveform viewer) the following Tcl commands are used.

```
vsim ps2_ascii_tb_simple do scripts/wave.do run 100ns
```

First the `vsim` command initializes the simulation with a certain testbench entity (in this case `ps2_ascii_tb_simple`). See `vsim -help` for more information on arguments for this command. Next the waveform viewer is configured using the `wave.do` file as shown in Listing 1. Finally `run` starts the actual simulation. In this case we run the simulation for 100 ns (simulation time).

Using the following command the simulator can be started from a shell script:

```
vsim -do "vsim ps2_ascii_tb_simple; do scripts/wave.do; run 100ns"
```

Notice that we don't use the `-c` flag since we want the GUI to be started, the quit command is also omitted because we want the program to keep running after executing the simulation script. For text-based simulations without a graphical output it can, however, make sense to use `-c` and `quit`.

For convenience we provide you with a makefile template you can use during implementation of the lab exercises (Listing 3).

```
 1 #arguments to the compiler (-2008 selects the VHDL 2008 standard)
 2 VCOM_ARGS=-2008 -work work -suppress 1236
 3
 4 #additional arguments to vsim
 5 VSIM_ARGS=-msgmode both
 6
 7 #list the VHDL input files here (notice the escape character!)
 8 VHDL_FILES = \
 9   src/file1.vhd\
10   src/file2.vhd
11
12 #list the VHDL used during simulation (testbenches etc.)
13 TB_FILES = \
14   tb/snes_cntrl_tb.vhd
15
16
17 #the name of your testbench entity
18 TB = my_tb
19
20 # the desired simulation time
21 SIM_TIME = 100ns
22
23 #compile everything
24 compile:
25   rm -f log
26   vlib work | tee log
27   for i in \$(VHDL_FILES); do \
28     vcom \$(VCOM_ARGS) \$\$i | tee -a log;\
29   done;
30   for i in \$(TB_FILES); do \
31     vcom \$(VCOM_ARGS) \$\$i | tee -a log;\
32   done;
33   @echo "----------------------------------------------------------------"
34   @echo "--                  Error and Warning Summary                --"
35   @echo "----------------------------------------------------------------"
36   @cat log | grep 'Warning\|Error'
37
38 list_sources:
39   @for i in \$(VHDL_FILES); do \
40     echo \$\$i;\
```

```
41    done;
42
43 sim:
44    vsim -do "vsim \$(TB) \$(VSIM_ARGS); do wave.do; run \$(SIM_TIME)"
45
46 clean:
47    rm -f transcript
48    rm -f vsim.wlf
49    rm -f log
50    rm -fr work
```

Listing 3: Makefile

Now the compilation and simulator can be started with:

```
make compile
make sim
```

3   Synthesis and Place&Route

Altera Quartus can be started in a terminal with the following command:

```
quartus --64bit&
```

3.1   Creating a new Project

In order to create a new project go to the menu "*File*" ⇒"*New Project Wizard...*". The wizard lets you configure basic settings of your project. Figure 3.1 to Figure 3.4 show the settings necessary for creating a project for the FPGA board and the tool-chain of this lab course.

After you have successfully created a new project and properly added your VHDL files, you can run a first compilation by clicking the menu entry "*Processing*" ⇒"*Start*" ⇒"*Start Analysis & Synthesis*". Alternatively, you can click the corresponding icon in the toolbar.
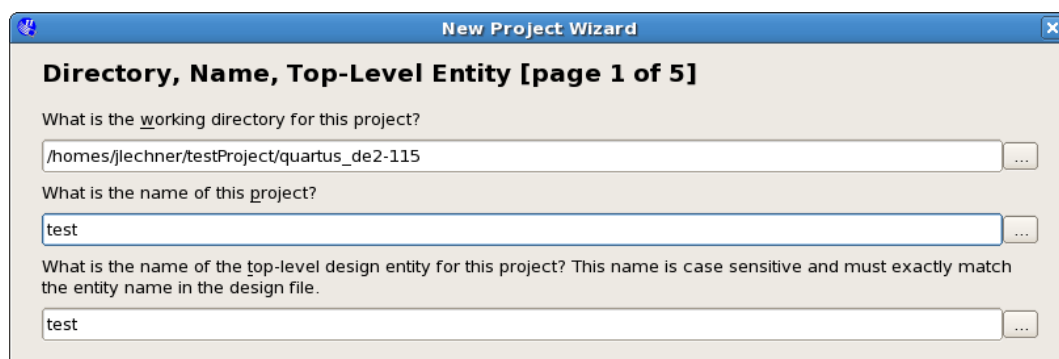
Figure 3.1: New project wizard – Step 1.
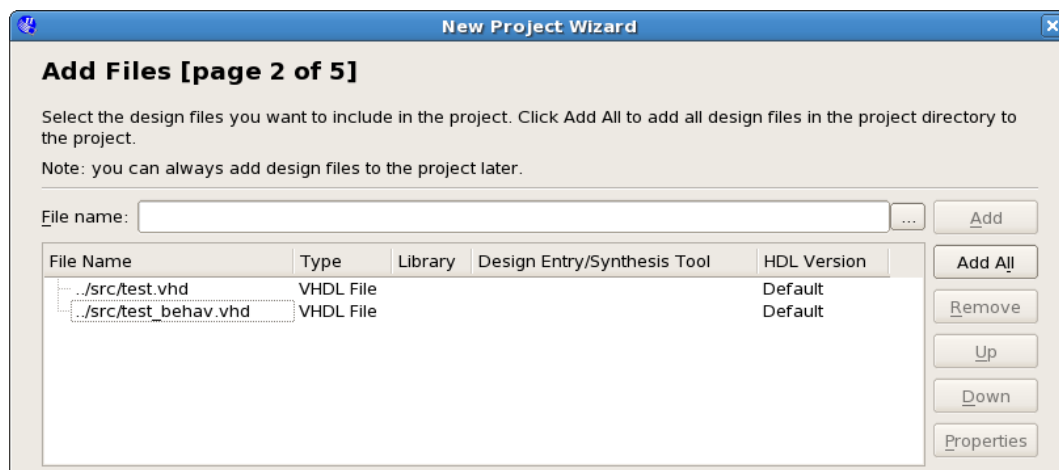
Figure 3.2: New project wizard – Step 2.

Figure 3.3: New project wizard – Step 3.

Figure 3.4: New project wizard – Step 4.

3.2   Compiler Settings

To change the compiler settings of a Quartus project open the dialog under "*Assignments*" ⇒"*Settings*" and select "*Compiler Settings*" in the list on the left side of the window (see Figure 3.5). Here you can select the optimization mode and change various other fine-grained synthesis and fitter settings. The VHDL version that is used for the Quartus project can be configured in the "*VHDL input*" sub-menu on the left.
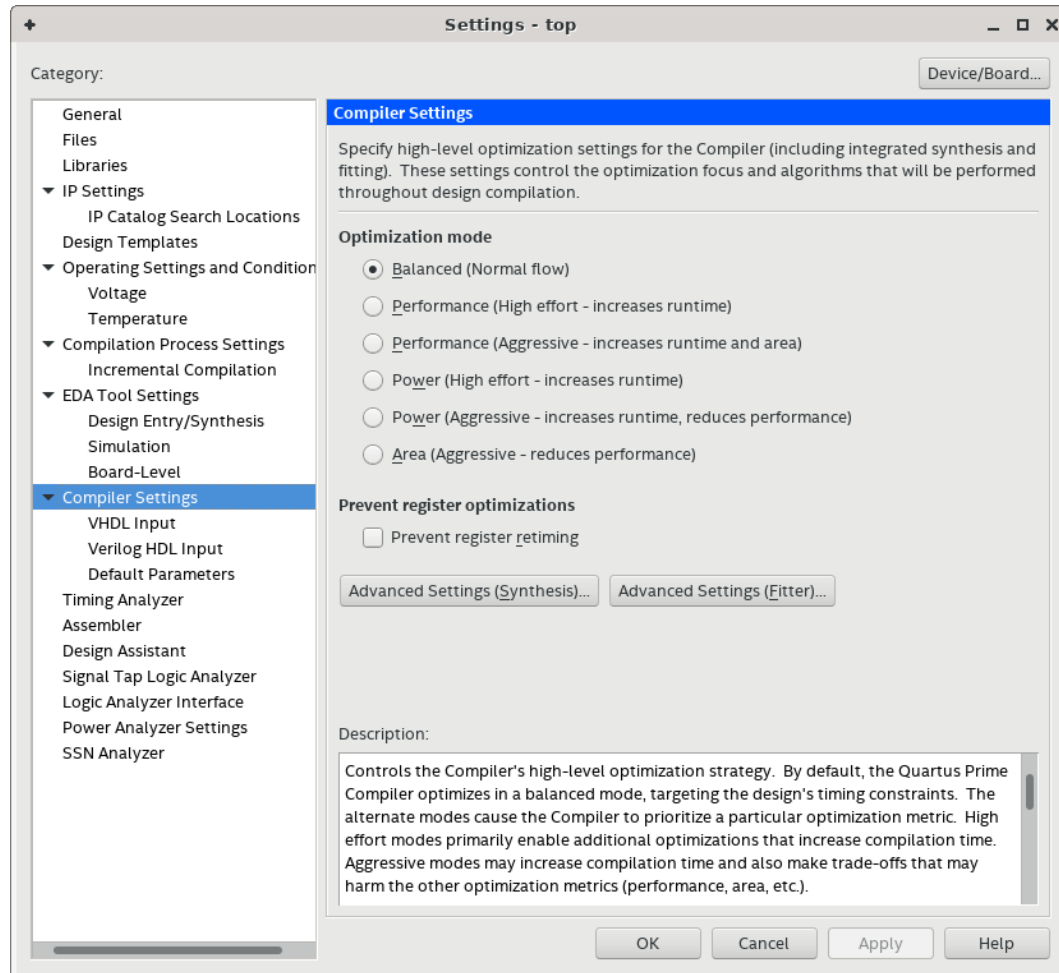
Figure 3.5: Quartus project compiler settings

3.3   Timinig Analyzer

The Timing Analyzer needs to be parametrized with a constraints file. The constraints, which need to be specified for a simple FPGA project are the clock period of the external oscillator. Furthermore we need to tell the timing analyzer to derive all PLL generated clocks automatically. A constraints file is created using the menu "*File*" ⇒"*New*". In the newly displayed dialog select "*Other Files*" ⇒"*Synopsys Design Constraints File*" and click OK (see Figure 3.6). Add the contents of Listing 4 to your newly created constraints file and save it in the src/quartus_de2-115 directory.
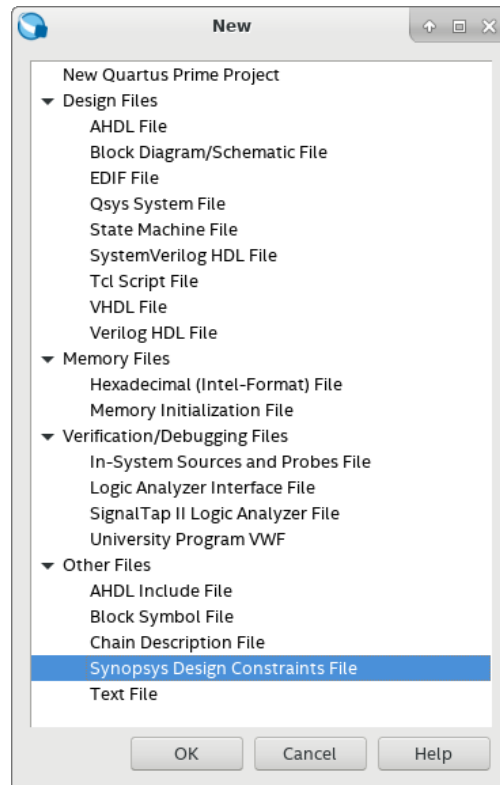
```
1 # Clock constraints
```

Figure 3.6: Add a new SDC file to the procject

```
2 create_clock -name "clk" -period 20.000ns [get_ports {clk}]
3
4 # Automatically constrain PLL and other generated clocks
5 derive_pll_clocks -create_base_clocks
6
7 # Automatically calculate clock uncertainty to jitter and other effects.
8 derive_clock_uncertainty
```

Listing 4: Design Constraints File

Add the constraints file to your project by clicking *"Project"* ⇒*"Add Current File to Project"* (the SDC file must be the current file in the Quartus editor).

3.4 Pin Mappings

Finally, we need to configure the pin assignments of your design (note that this can only be done after your VHDL files have been analyzed by a first compilation run as described above). In this task you assign the logical input and output ports of your design's top-level entity to the physical pins of the FPGA. Obviously, in most cases you can not randomly choose a pin. E.g., the reset port of your design needs to be assigned to exactly the pin, which is connected to the reset button on the FPGA board. The same is true for all other peripherial interfaces like VGA, RS232, key matrix, leds etc. The specific FPGA pin numbers for all the components can be found in the manual of the FPGA board.

In the Quartus user interface the pin assignment task is done with the so-called *Pin Planner* (menu *"Assignments"* ⇒*"Pin Planner"*). In the bottom half of the *Pin Planner* you can see a table listing the input and output ports of your design. The associated pin names need to be entered

in the location column of this table. Simply select one entry and start typing the name of the pin (e.g., M1 for the clock input).
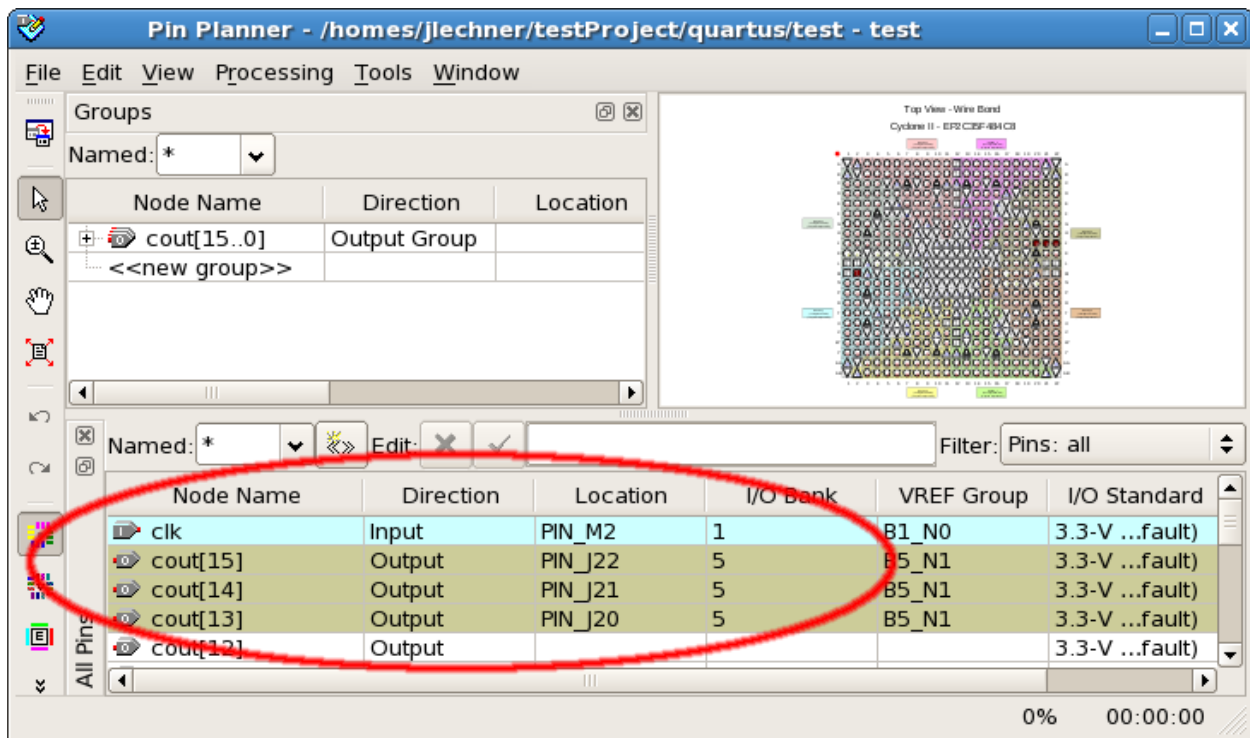

Figure 3.7: Assigning input and output pins.

Make sure *all* ports of your design are assigned to one FPGA pin. All other *unused* FPGA pins should be configured as tri-state input pins with weak pull-up resistors. This can be done in the device settings dialog (menu "*Assignments*" ⇒"*Device...*") where you need to click the button "*Device and Pin Options...*". In the following dialog, select the category "*Unused Pins*" and change the combo-box to "*As input tri-stated with weak pull-up*" (see Figure 3.8).

> Caution: The default setting "*As output driving ground*" should only be used if you really know what you are doing! Otherwise it might be harmful for some FPGA boards as it can cause short-circuits in certain circumstances.

Tip: If, during compilation, you get the error saying that a pin is doubly-assigned, check the pin's settings in "*Assignments*" ⇒"*Device*" ⇒"*Device and Pin Options*" ⇒"*Dual-Purpose-Pins*". For example, you may need to set the *nCEO* pin to *Use as regular I/O*.

Quartus is also able to import pin assignments from external files. If you have such a file you can import it by selecting "*Assignments*" ⇒"*Import Assignments...*". Now a dialog should pop up (see Figure 3.9) where the file can be selected. After clicking OK, the new pin assignments should show up in "Pin Planner".
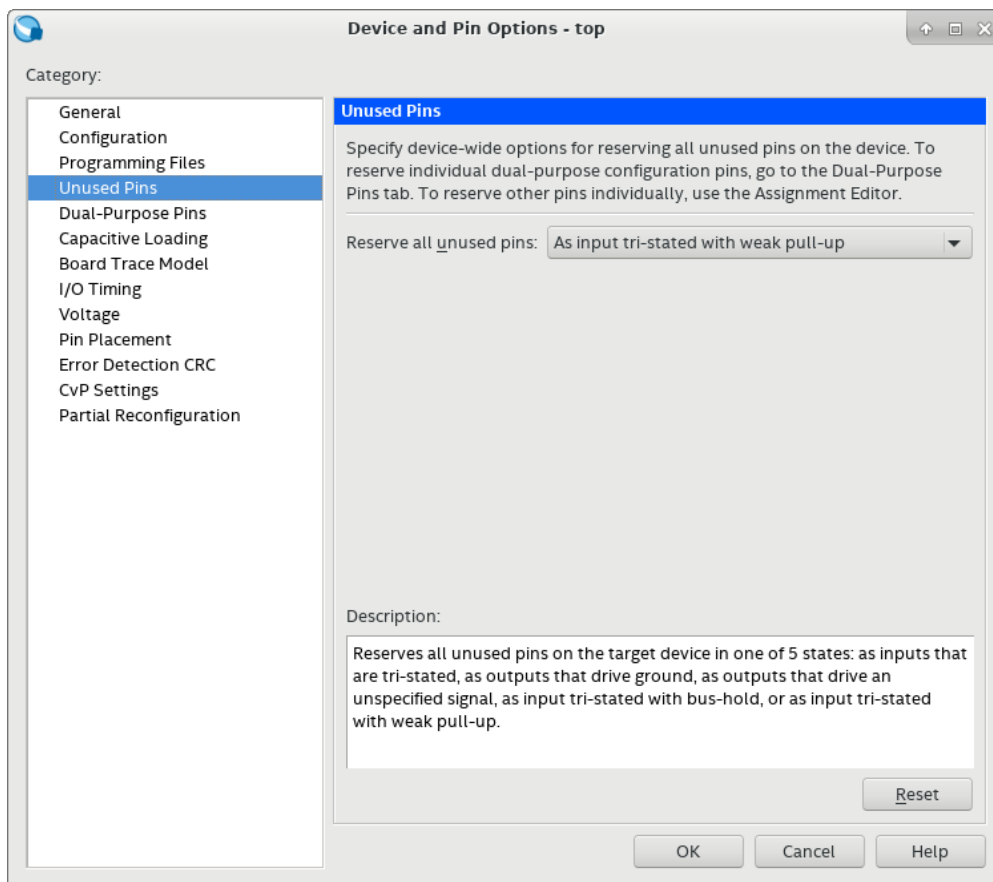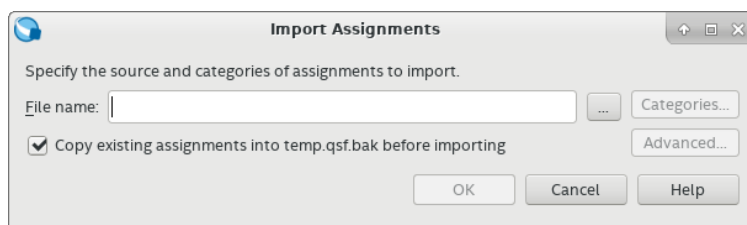
Figure 3.8: Unused pins setting

Figure 3.9: Import assignments dialog

3.5 Adding a PLL

Sometimes it is necessary to run the implemented circuit design with another clock frequency as provided by the external oscillator. Therefore Altera FPGAs include PLLs (phase locked loop). PLLs generate a stable clock signal which can be a rational multiple of the external reference clock.

3.5.1 PLL Generation in Quartus

Quartus provides a wizard, which allows you to create and configure a PLL component. This wizard can be started over the menu "*Tools*" ⇒"*IP Catalog*". On the right hand side of Quartus, a docked pane called IP Catalog will appear (see Figure 3.10). Open the tree "*Library*" ⇒"*Basic Functions*" ⇒"*Clocks; PLLs and Resets*" ⇒"*PLL*" and double click *ALTPLL*.
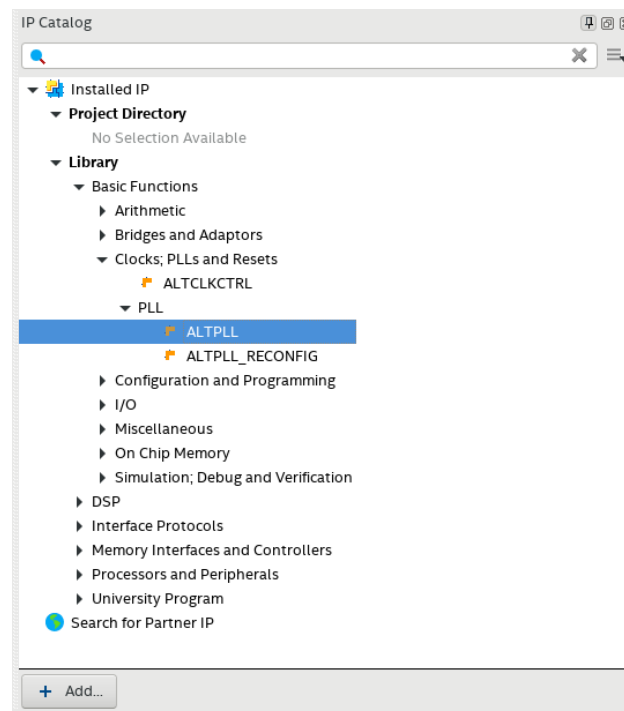

Figure 3.10: IP Catalog side pane in Quartus

In the opening dialog (Save IP Variation) enter the name and path for your PLL implementation file (e.g. /home/User/Projects/testPrj/src/quartus_de2-115/pll.vhd) and select VHDL as IP variation file type. Click OK.

Caution: If the wizard that shows up after you pressed OK looks like the one shown in Figure 3.11 you have to resize the window before you proceed (i.e. make the window larger such that the buttons in the lower right corner fit inside the window)! If you don't resize the window and click on any of the controls inside of the wizard, it may freeze and needs to be terminated.

Figure 3.12 to Figure 3.15 show how to generate a simple PLL configuration necessary for this lab course. The central settings are the frequency of the input clock, i.e., the external oscillator and the frequency of the PLL's output clock. As can be seen in the symbol in Figure 3.14, the customized PLL component only has two ports (input and output clock). The final screenshot shows the files which are generated by the wizard: The most important file is the VHDL file
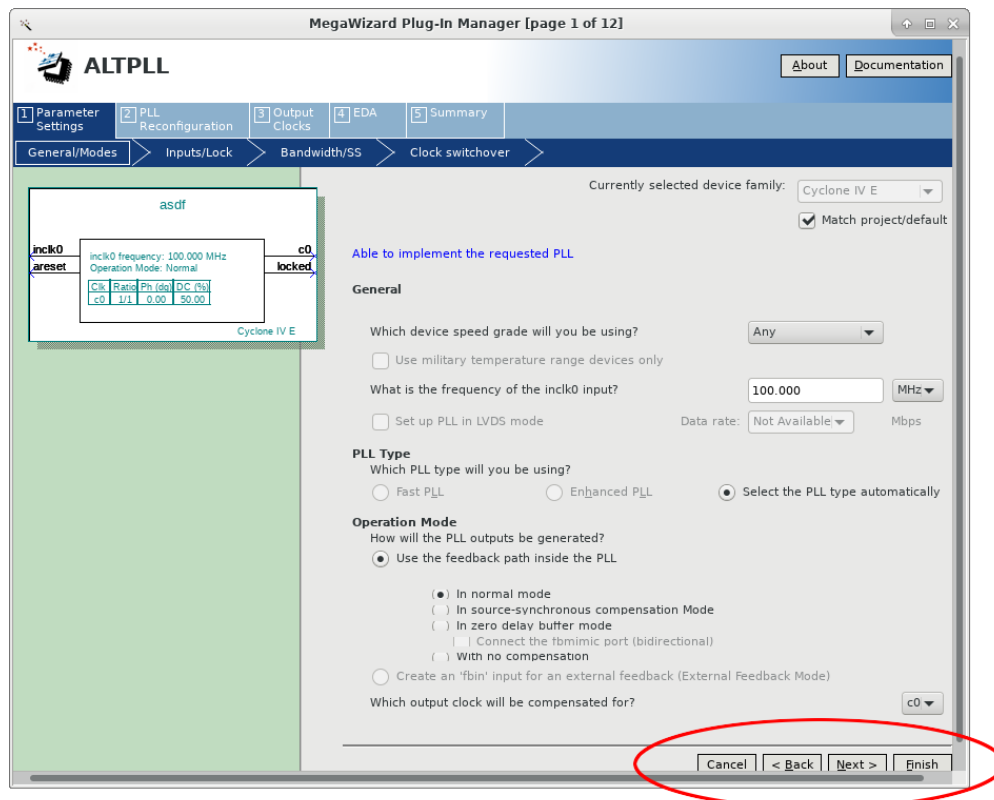
Figure 3.11: PLL Wizard Bug

containing the description of the PLL component based on your settings. Furthermore a cmp-file is generated, which provides the component declaration of the PLL. This declaration can be copied into your top-level design unit to be able to instantiate the PLL.

Finally a dialog box appears asking if you want to add the IP file to the Quartus project. Answer the dialog with "Yes". The PLL's VHDL file will then be automatically added to your project.

3.5.2   Simulation

It is of course possible to simulate designs containing enities created by the IP core wizard of Quartus. However, when using a PLL special care must be taken. Make sure that the simulation time in Questa/Modelsim is set to 1 ps! Otherwise the PLL will not work! For the graphical user interface this is shown in Figure 3.16.

If the simulation is started using the command line, use the -t command line argument:

```
vsim testbench -t ps [...  others args ...]
```

Figure 3.12: Creating a PLL - Step 1.

Figure 3.13: Creating a PLL - Step 2.
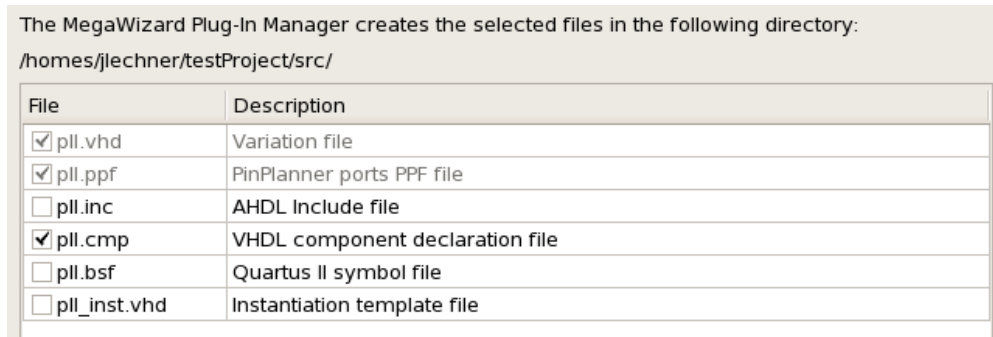

Figure 3.14: Creating a PLL - Step 3.
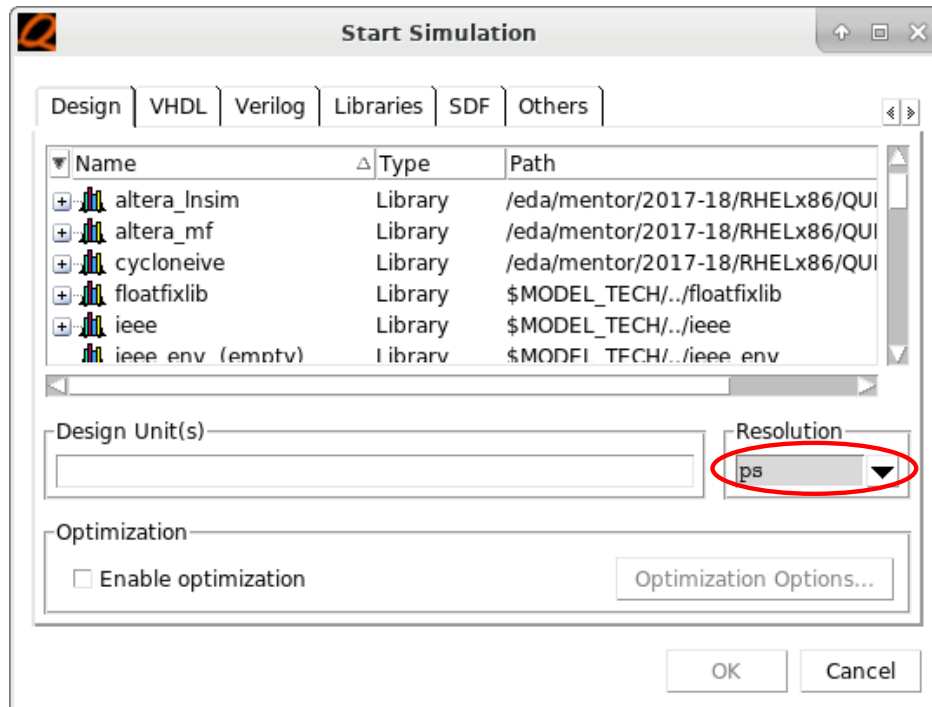
Figure 3.15: Creating a PLL - Step 4.

Figure 3.16: Setting the time resolution of the simulation

3.6 Full Compilation

Once the project has been set up correctly, a full compilation can be performed by clicking the menu entry "*Processing*" ⇒ "*Start Compilation*". Alternatively, you can click on the corresponding button in the toolbar. If the compilation runs successfully, Quartus generates the following files:

- Bitstream file for programming the FPGA (.sof)

- Netlist file for post-layout simulation (.vho)

- Timing file for post-layout simulation (.sdo)

The bitstream file can be found in the *output_files* subdirectory, while the simulation files are stored in the subdirectory *simulation/modelsim*. Detailed results of the compilation and the synthesized circuit are shown in the compilation report window (Figure 3.17). Carefully inspect all warnings produced during the compilation. A tidy VHDL design should have no critical warnings and also non-critcal warnings should be resolved whenever possible.

Figure 3.17: Compilation Report.

3.7 Export a Project Tcl Script - Optional

Alternatively to using the Quartus GUI, it is also possible to manage a Quartus project with a Tcl script. Quartus can generate such a script for an existing project and include all settings you have made so far. Simply click on the menu item "*Project*" ⇒ "*Generate Tcl File for Project...*" and store the script file in the source directory. The Tcl script can then be executed with the following shell command:

```
quartus_sh -t path/to/script.tcl
```

If you execute the above command in an empty directory, a new Quartus project is created in this directory with all settings as specified by the script. If you execute the command in a directory where the corresponding project already exists, the script will simply update the project settings. E.g., if you have modified the pin assignments in the Tcl script, you can update these settings in

the existing project by executing the script. This can even be done while the project is opened in Quartus – the new settings will be applied immediately. Listing 5 shows a short code snippet with some Tcl statements.

```
1  # use timequest
2  set_global_assignment -name USE_TIMEQUEST_TIMING_ANALYZER ON
3
4  # add project files
5  set_global_assignment -name VHDL_FILE ../src/subcomp_behav.vhd
6  set_global_assignment -name VHDL_FILE ../src/test_behav.vhd
7  set_global_assignment -name VHDL_FILE ../src/subcomp.vhd
8  set_global_assignment -name VHDL_FILE ../src/test.vhd
9
10 # pin assignments
11 set_location_assignment PIN_M1 -to clk
```

Listing 5: Tcl script for managing a Quartus project.

4   Postlayout Simulation

Similar to a behavioral simulation a postlayout simulation can also be executed using the GUI or the command line insterface.

4.1   Using the GUI

Create a new Questa/Modelsim project in the post-layout simulation directory as described in Section 2. But in contrast to the behavioral simulation the only design file you need to add for a postlayout simulation is the netlist file (.vho). This file is produced by Quartus during the full compilation and is located in the `simulation/modelsim` subdirectory of your Quartus project. Furthermore you need to add your testbench files and the simulation scripts to the Questa/Modelsim project. In Figure 4.1 you can see a simple post-layout simulation project in Questa/Modelsim.
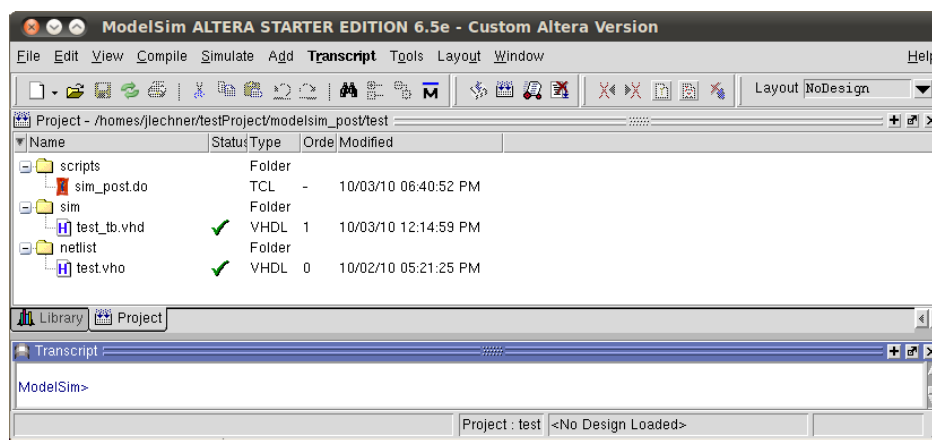
Figure 4.1: Questa/Modelsim project for a post-layout simulation.

Hit the "*Compile all*" button to compile the netlist and the testbenches. Then the simulation can be started with the menu "*Simulate*" ⇒ "*Start Simulation...*".
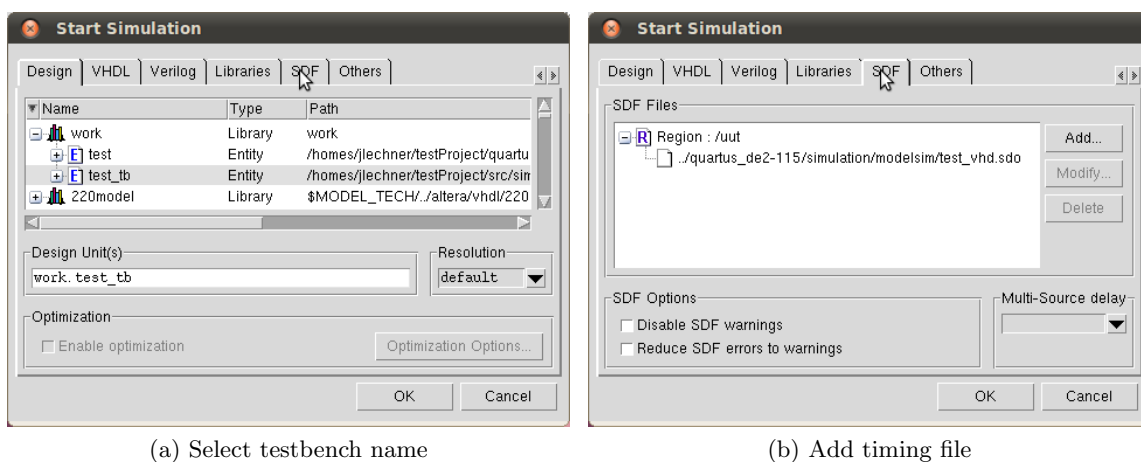
(a) Select testbench name   (b) Add timing file

Figure 4.2: Starting the simulation.

As with a behavioral simulation you have to specify the name of the testbench to be simulated (see Figure 4.2a). In case of a post-layout simulation additionally a timing file (.sdo) should be

added. This file is also located in the simulation subdirectory of your Quartus project. Click the tab "*SDF*" as illustrated in Figure 4.2b. and add the timing file to the list of SDF files. It is important to specify the correct *region* where the sdo file should be applied to. In most cases the testbench directly instantiates the the top-level entity of the design. Thus the region path would simply be the instance name of the design's component in the testbench file. E.g., consider the code snippet of a testbench shown in Listing 6. The instance name is "*uut*", therefore the region path for the timing file needs to be "*/uut*" (compare Figure 4.2b).

```
1 uut : test
2 port map (
3   clk   => clk,
4   reset => reset,
5   cout  => cout
6 );
```

Listing 6: Instantiation of the design unter test.

Adding signals to the waveform viewer and running the simulation for a specified time works exactly like when doing a behavioral simulation. The only difference is, that internal signals may have been renamed during the synthesis with Quartus or even removed due to optimization steps. Therefore it might be necessary to add debug output ports to the top-level entity and connect the internal signals to these debug ports. Then the internal signals can be traced using these debug ports.

4.2 Using Scripting

A post-layout simulation can also be automated very easily with the help of Tcl or shell scripts. As with the process described above it is important to first compile the required netlist file generated by Quartus and located in the `simulation/modelsim` directory in the quartus project folder. The crucial part is then to pass the timing file (i.e. the `sdo` file generated by Quartus) to the simulator command using the `-sdftyp` argument:

```
vsim -sdftyp /uut=/path/to/quartus_project/simulation/modelsim/test_vhd.sdo work.test_tb
```

The left side of the equality sign specifies the name of the instance to which the timing information file (right side) should be applied to.

The shell script in Listing 7 shows an example of a post-layout simulation script. It first compiles the relevant files and then start the simulator (in GUI mode).

```
1 VCOM_ARGS=-2008 -work work
2 SDO_FILE=/path/to/quartus_project/simulation/modelsim/[...].sdo
3 VHO_FILE=/path/to/quartus_project/simulation/modelsim/[...].vho
4
5 vlib work
6 vcom $VCOM_ARGS $VHO_FILE
7 vcom $VCOM_ARGS test_tb.vhd
8 vsim -do "vsim test_tb -sdftyp /uut=$SDO_FILE; do scripts/wave.do; run -all"
```

Listing 7: Shell script for running a post-layout simulation.

5 Download

Programming the FPGA can be easily done with the Quartus Programmer, which is pictured in Figure 5.1. Open the programmer over the menu entry "*Tools*" ⇒ "*Programmer*" or the corresponding button in the toolbar. Typically all settings are correct (compare with Figure 5.1) when the

window opens and you can simply start the programming routine by clicking the "*Start*" button. The bitstream file (.sof) generated during the compilation is downloaded over a parallel inferface and stored in the configuration SRAM of the FPGA. Since an SRAM is not a permanent memory, the configuration data is lost, if the board is disconnected from power. The FPGA then needs to be reprogrammed. If the .sof does not show up in the programmer window, you can manually add it. It is usually located in the `output_files` directory of the quartus project directory.

If the programmer window is not configured correctly or the programming procedure fails, click the button "*Auto Detect*" to check if the FPGA is properly connected to your PC. If the FPGA device appears, you can right click on the device symbol and assign the sof file using the menu item "*Change file*". Make sure "*Program/Configure*" is checked and hit "*Start*" to begin programming.
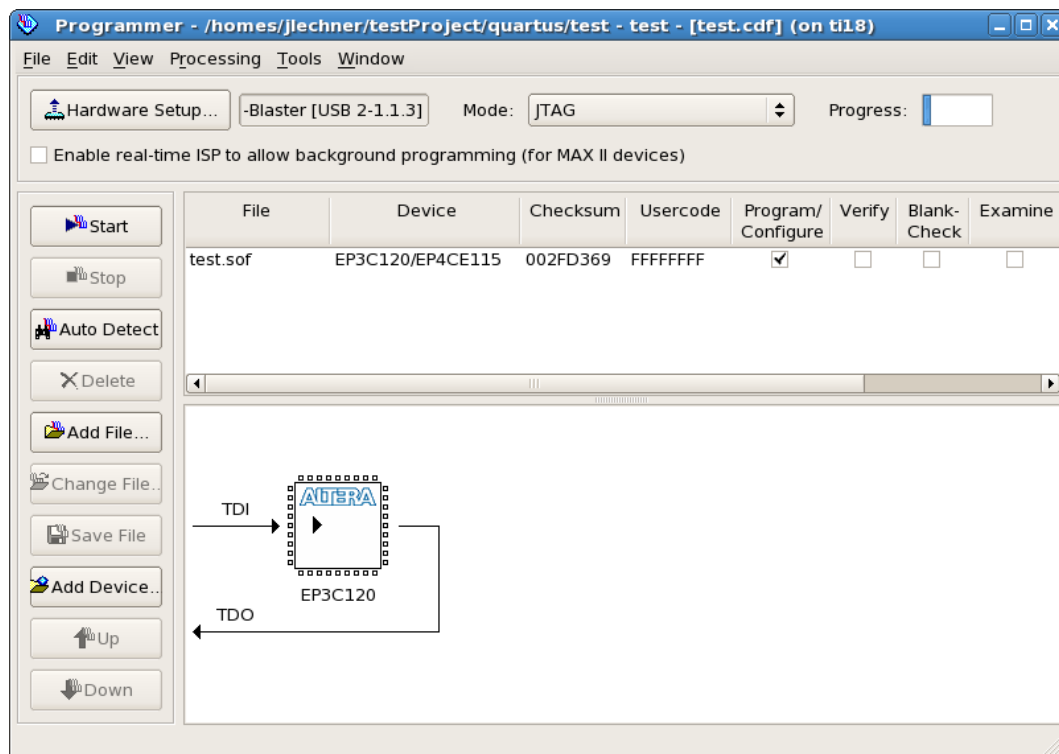

Figure 5.1: Programming the FPGA.

Caution: There is a bug in the JTAG server, which can cause the programmer to fail or freeze. In such a case perform the following routine:

- Switch off the FPGA board (if possible).

- Open a terminal and execute the command `killall jtagd`.

- Now switch it on and again execute the command `jtagconfig`. The Board should be detected and you should get an output simular to:

```
1 1) USB-Blaster [1-8.3]
2      020F70DD   10CL120(Y|Z)/EP3C120/..
```

It is now possible to run the programmer.

Revision History

Revision	Date	Author(s)	Description
23.0	09.03.2023	FH	Removed outdated logic analyzer section
22.0	03.03.2022	FH	Updated Quartus programmer section
21.0	08.03.2021	FH	Added "Compiler Settings" section (Quartus)
20.0	01.03.2020	FH	Improved scipting sections
19.0	01.03.2019	FH	Improved scipting and PLL sections
17.0	01.03.2017	FH, FK	Updated version
16.0	01.10.2016	JL, TP, RN	Initial Version

Author Abbreviations:

FH	Florian Huemer
FK	Florian Kriebel
JL	Jakob Lechner
JM	Jürgen Maier
RN	Robert Najvirt
TP	Thomas Polzer