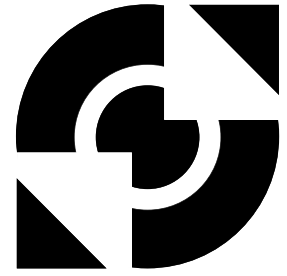


University of Twente

Faculty of Electrical Engineering,
Mathematics and Computer Science



VHDL tutorial

For internal use only

©E. Molenkamp

August 2012

Contents

1	Introduction	3
2	Simulation with ModelSim	4
2.1	Analyse/Compile	5
2.2	Simulate	7
2.2.1	Script file with the stimuli	10
2.2.2	Stimuli generation with VHDL	10
2.3	Simulation model	14
3	Synthesis with Quartus II	16
3.1	Start Quartus II	17
3.2	Create a new project	17
3.3	Top level of the design	20
3.4	Compile (=synthesize)	20
3.5	RTL viewer/Technology Map Viewer	21
4	Post simulation	22
5	Constraint file	24
6	Programming the <i>LiveDesign Evaluation kit</i>	25
Appendix A	Synthesis with Precision RTL	26
Appendix B	An alternative description for count	29
Appendix C	Verification of a design via simulation	30
Appendix D	ModelSim/QuestaSim if no project is used	32
Appendix E	Compiling Altera libraries with QuestaSim	33
Appendix F	Quartus II, License?	35
Appendix G	Quartus II USB Blaster (programmer)	36
Appendix H	Location of the software on the lab machines	37

1 Introduction

VHDL is the hardware description language used in this course. It is one of the languages used in many companies in Europe. Many tools are available for simulation and synthesis. We have chosen a toolset that can also be installed at home (no license required).

		Home	University
VHDL simulation	ModelSim-Altera Starter Includes post simulation libraries for Altera devices. (Windows/Linux) (Optimization is disabled in the free version.)	X	X (windows)
	QuestaSim Installed on the lab machines and supports PSL. For post simulation of a design the vendor libraries should be compiled first.		X
VHDL Synthesis	Quartus II		X
	Quartus II web edition (free)(windows/linux)	X	X (windows)
	Precision RTL Is a technology independent synthesis tool.		X

Table 1: tools used in the course

The tools to be used at home can be downloaded from:

https://www.altera.com/support/software/download/altera_design/quartus_we/dnl-quartus_we.jsp

A disadvantage is that probably newer versions will be available at the time you download the software. Hence, the tutorial may not be correct anymore at that time. We expect minor changes.

If you use the software on the lab machines then you have to know the location of the software (see Appendix H).

2 Simulation with ModelSim

In this tutorial a circuit is used that counts the number of ones in the input pattern.

```
1.  LIBRARY ieee;
2.  USE ieee.std_logic_1164.ALL;
3.  ENTITY count IS
4.    GENERIC (w : positive := 8);
5.    PORT (a : IN  std_logic_vector(w-1 DOWNT0 0);
6.          q : OUT integer RANGE 0 TO w);
7.  END count;
8.
9.  ARCHITECTURE behaviour OF count IS
10.    FUNCTION cnt (a:std_logic_vector) RETURN integer IS
11.      VARIABLE nmb : INTEGER RANGE 0 TO a'LENGTH;
12.    BEGIN
13.      nmb := 0;
14.      FOR i IN a'RANGE LOOP
15.        IF a(i)='1' THEN nmb:=nmb+1; END IF;
16.      END LOOP;
17.      RETURN nmb;
18.    END cnt;
19.  BEGIN
20.    q <= cnt(a);
21.  END behaviour;
```

Figure 1: behavioural description of count (the line numbers are not part off the VHDL code!)

Generic w , line 4, is a global constant with value 8. Input a of this design is w bits wide. The output q is an integer value. The width of the input is w therefore the number of ones must be between 0 and w (inclusive). A range constraint is added to the integer type. The range constraint is not necessary but it is used for documentation and will guide synthesis process.

There are many ways to count the number of ones in an array. In the architecture (figure 1) a function is declared that takes care of this. This function has as input an object of type *std_logic_vector*. A *std_logic_vector* is an unconstrained array; the length of this type is not (yet) known! The reason to use an unconstrained array as input is to make the design generic with respect to the width of the input. At the location of the function call, line 20, the range of the input is known.

The algorithm used in the function is straightforward. With a loop statement all elements of the inputs are examined. The only problem is: what are the vector indices? The attribute *'range* is used for this. If the function is called with an object that is declared as *std_logic_vector(5 to 36)* then within the function the *a'range* is replaced with *5 to 36*.

This design is simulated using ModelSim.

2.1 Analyse/Compile

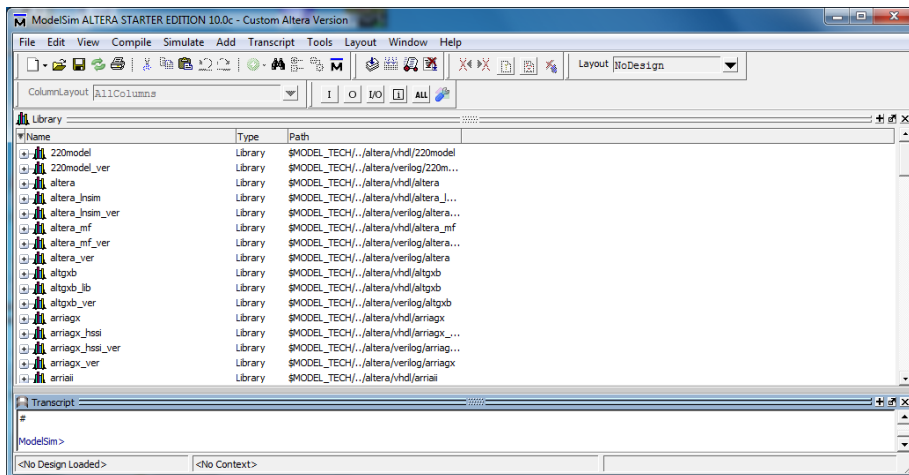
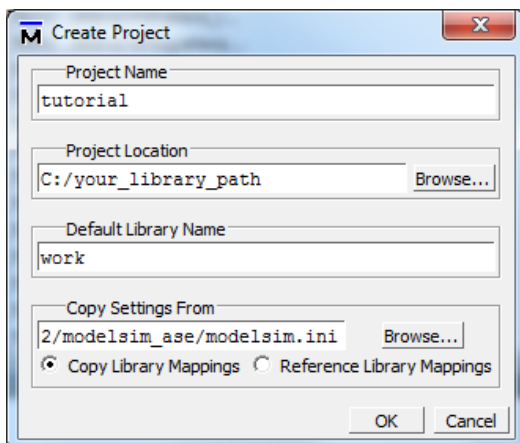


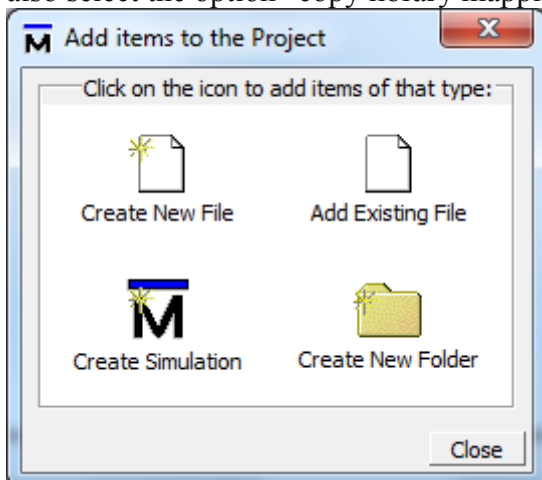
figure 2: ModelSim screen

ModelSim starts with the window shown in figure 2. In this tutorial a project is used. So we will first create a project:

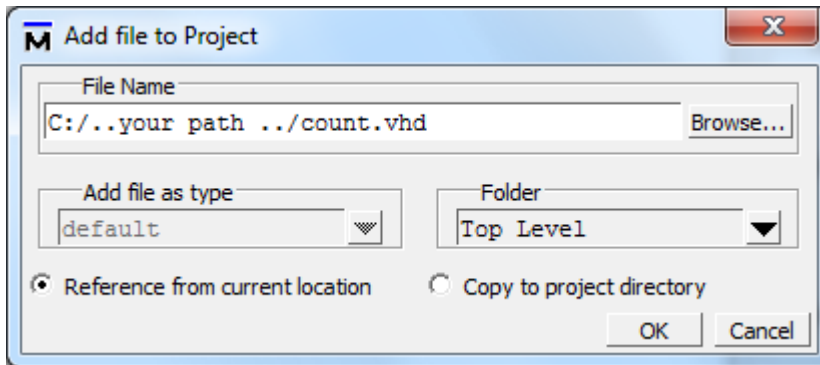
File→New→Project



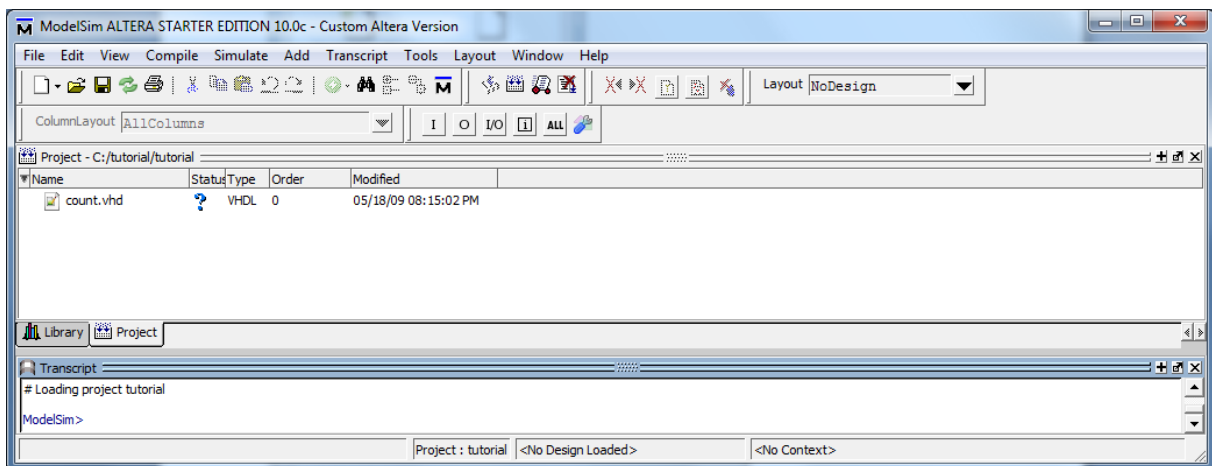
Choose a “project name” and “Project location”. The default library must not be changed, and also select the option “copy library mappings”.



Select “add existing file” and browse to the directory with the source file and select the file “counter.vhd”.



Click on OK
and close the “add items to the Project”



Right-click on the file “count.vhd” → compile → compile selected
The design is compiled into library *work*. You can verify this by clicking the tab *library*

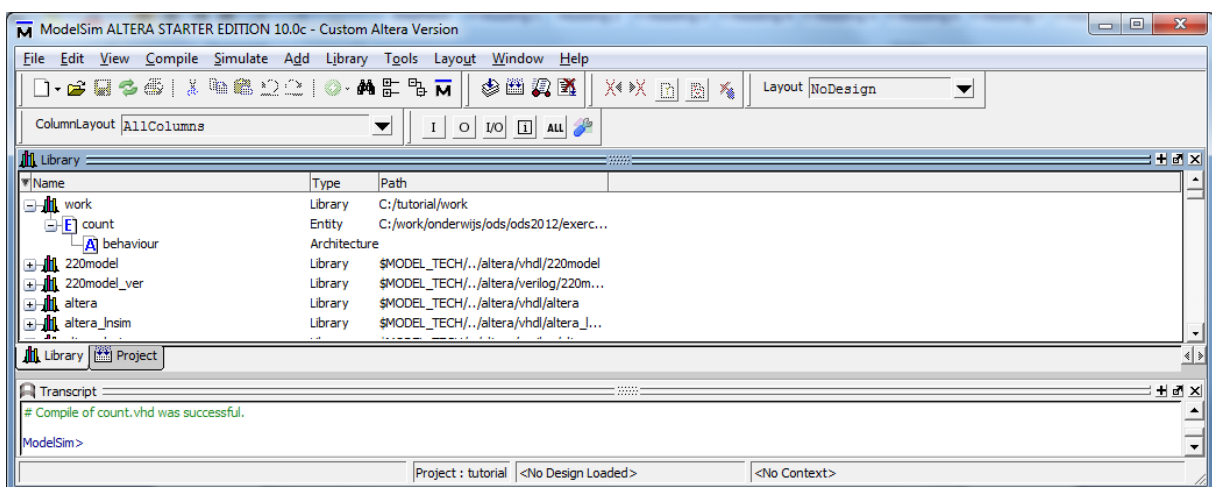


Figure 3: the result after compilation

The top window shows the libraries and the bottom window (“Transcript”) is used for commands entered by the user and for reporting information to the user. You can *dock* (as

shown above) and *undock* (a separate window) using the arrow in the upper right corner of a window.

By default an analysed VHDL design is stored in a library *work*.

Notes

- 1 The content of the library *work* is managed by ModelSim. Do not change the contents of this library outside ModelSim nor place your source files in that library!
- 2 If you want to delete a compiled design from the library right click the design (tab *library* should be selected) in the *GUI* and select *delete*. Your source file is still in the project.
- 3 If you want to remove a file from the project right click the file (tab *project* should be selected) in the *GUI* and select *remove from project*.
Important: if the file was compiled then the design units that were in the file are not removed from the library!
- 4 ModelSim uses the term *Compile* instead of *Analysis*.
- 5 For the VHDL object *signal* ModelSim uses also the term *object*.
- 6 For the VHDL object *variable* ModelSim uses also the term *local*. During debugging of a design locals are not visible by default. This can be changed:
tab View → select “locals”.

2.2 Simulate

Click with the right mouse button on the architecture name ‘behaviour’ and you can load your design in the simulator (or you can use menu *simulate* → *start simulation*).

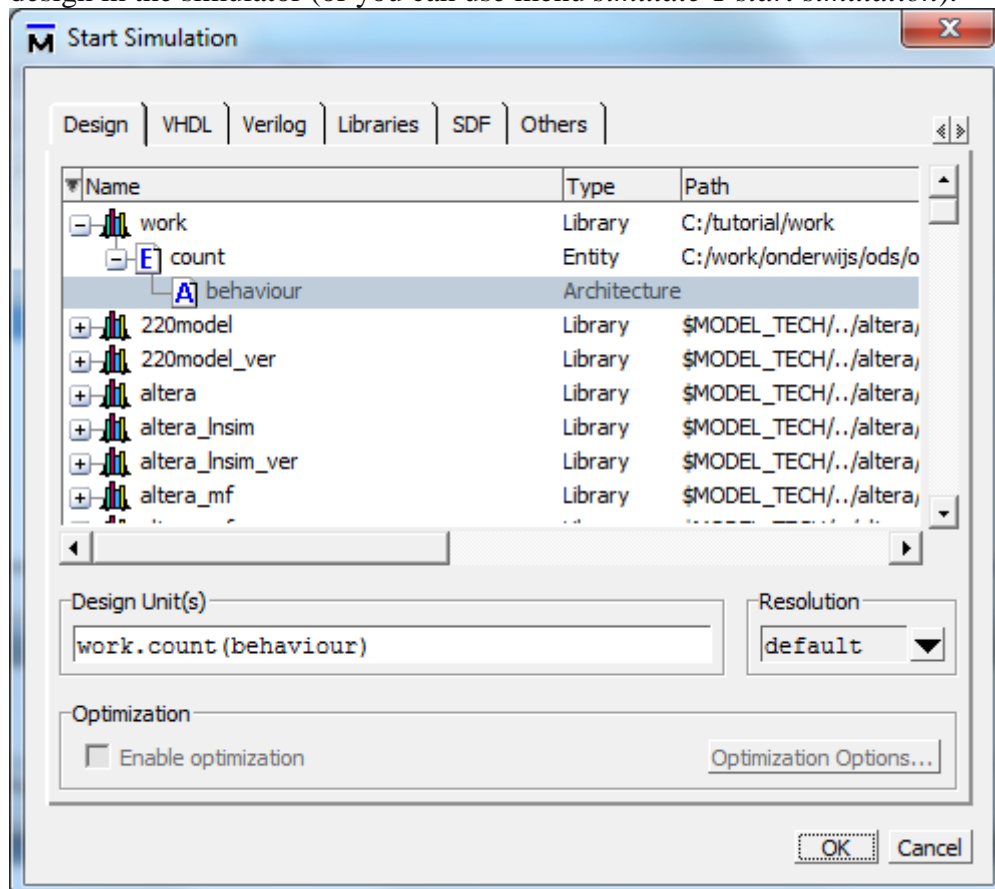
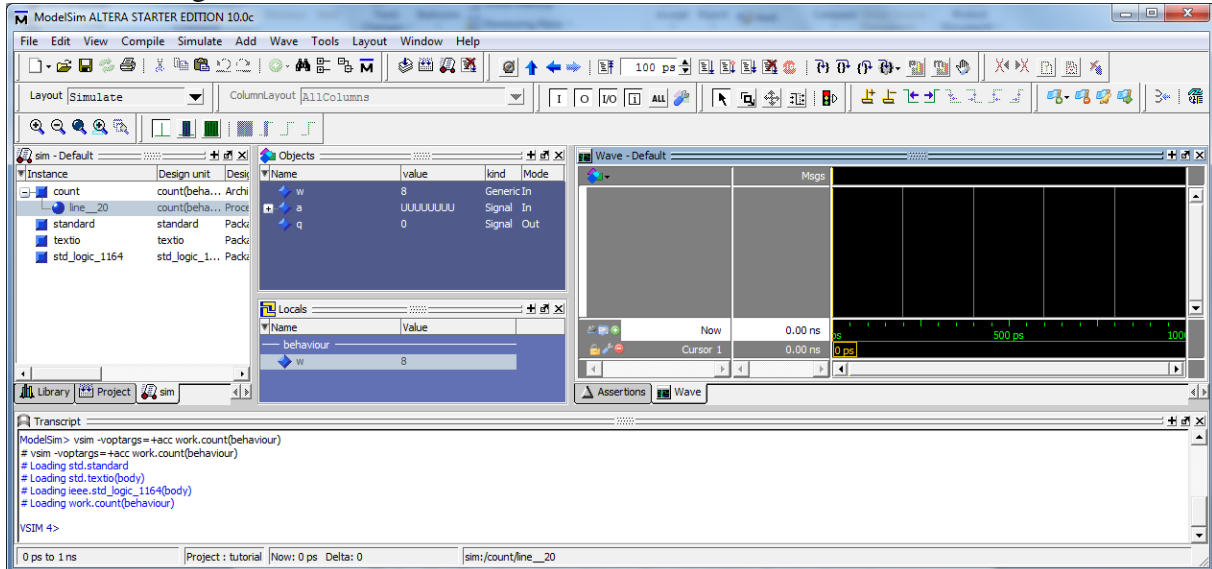


Figure 3a: Selection of the design that should be simulated

Note:

In the ModelSim-Altera starter edition “Enable optimization” is disabled. In case a licenced version of ModelSim/QuartaSim is used optimizations it is on by default. Optimization improves simulation speed but during debugging not all signals and variables are visible. **Therefore select ‘full visibility’ in the tab Optimization Options.**

Select the design and click OK



During simulation you probably like to see some waveforms therefore enter:

add wave * <return>

(In stead of * you may enter a list with the signal names separated with a comma).

Notes

1. If the signals *a* and *q* are not shown in the window *wave* you probably did not select *count* in the window *instance*. Select *count* and repeat the command “add wave *”
2. The windows *objects* and *locals* show respectively the signals and variables that are visible at the selected *instance*. Select *line__20* and you will also see the generic *w*.
3. You can also drag and drop objects and locals to the wave window.

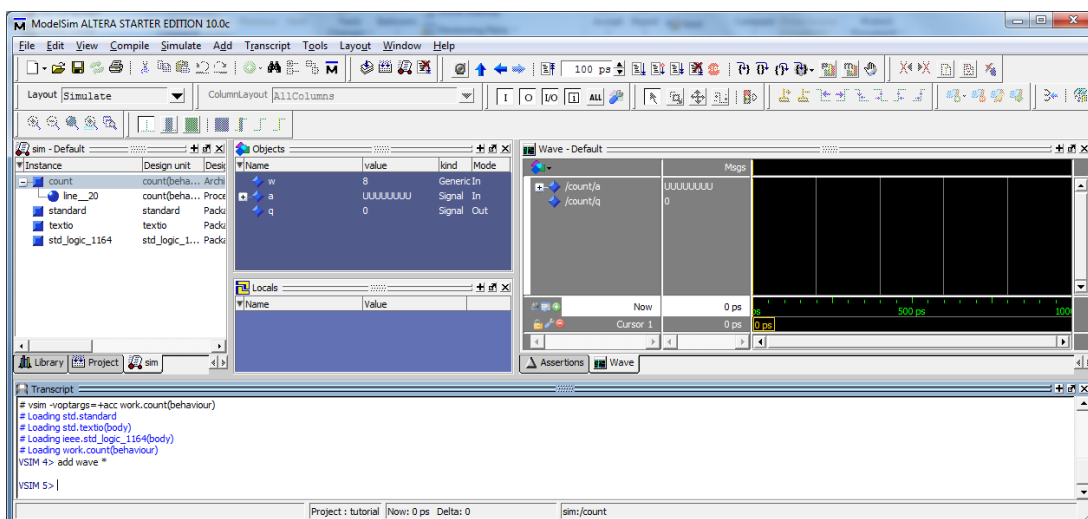


Figure 3b: Selection of the design that should be simulated

With the run command you perform a simulation:

run 200ns <enter>

Why are the input values 'U'?

With the force command you can apply an input pattern to *a*:

force a 01100011 <enter>

run 100ns <enter>

Try it with some other values for *a*.

You can assign multiple values to the input with:

force a 11111111, 00111111 10ns, 11110101 20ns <enter>

run 100ns <enter>

Try it.

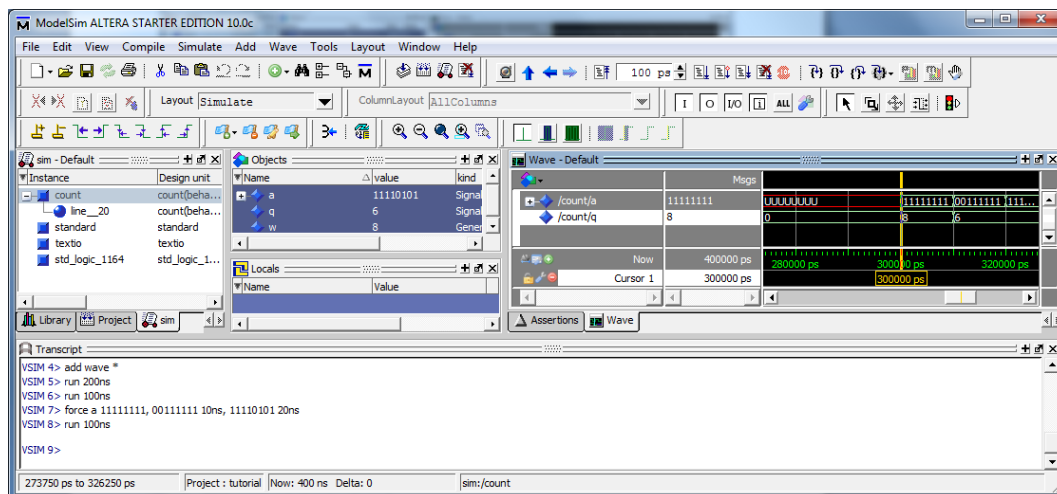
Note:

In VHDL descriptions a space is required between the number and the time unit. So “100ns” is not correct, it must be but “100 ns”. ModelSim will report a warning:

Warning: [4] <path>.<file>(<line number>): (vcom-1207) An abstract literal and an identifier must have a separator between them.

It is probably only a warning because the examples in the first VHDL standard (1987) did not have the spaces☹.

The user interface of ModelSim supports both.



Notes:

1. If the wave window is selected you can zoom in and out with the buttons:



- zoom in
- zoom out
- zoom full
- zoom in on active cursor. (Select a point in the wave window and a cursor appears.)

2. Multiple cursors are possible in the wave window. Right-click the bottom black line in the wave window. The time differences between the cursors are also shown.

2.2.1 Script file with the stimuli

A tool dependent solution to apply stimuli is using a script file. A simple example is given beneath. Create a file “demo.do” with the following contents:

```
force a 00011111
run 100 ns
force a 10100000
run 100 ns
```

In ModelSim this script file is executed with the command:
do demo.do <return>

Notes

- 1 In a synchronous design a clock signal is needed. Assume signal *clk* is the clock line. A repetitive pattern is created with the command:
force clk 0, 1 50 ns –repeat 100 ns
(this can also be used for other signals, not necessarily a clock.)
- 2 The ModelSim command “run –all” starts a simulation and will stop when no events are scheduled for the future. Do not use this command when a clock signal is generated with the method described in Note 1. Why not?
- 3 Commands that are entered in the transcript window can be written to a file with the command “write transcript < filename>”. This file can be used as a script file afterwards.

2.2.2 Stimuli generation with VHDL

Applying stimuli as presented in the previous section is tool dependent. It is also possible, and strongly advised to be tool independent, to generate stimuli using VHDL.

Finding test data for a design is not an easy task. In this example an exhaustive test is used when the width of the data vector is not too large.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
ENTITY testset IS
    GENERIC (w : positive := 8);
    PORT (data : OUT std_logic_vector(w-1 DOWNT0 0));
END testset;
```

```

ARCHITECTURE set1 OF testset IS
BEGIN
  PROCESS
  BEGIN
    data <= (others => '0');    -- all zero
    WAIT FOR 10 ns;

    data <= (others => '1');    -- all one
    WAIT FOR 10 ns;

    FOR i IN 0 to 2**w-1 LOOP
      data <= std_logic_vector(to_unsigned(i,w));
      EXIT WHEN i=20; -- an exhaustive test is not performed if w is large
      WAIT FOR 10 ns;
    END LOOP;
    WAIT; -- forever
  END PROCESS;
END set1;

```

Figure 4: simple test set.

Figure 4 shows a simple test set. It contains one process statement. It first generates all zeros, waits for 10 ns, then it generates all ones and it waits again. Of course an exhaustive test is possible. In the for-statement the loop parameter i (which is implicitly declared!) goes from 0 to 2^w-1 . This integer value is converted to a bit pattern (using binary coding; also called unsigned). For the conversion the function *to_unsigned* is used. This function converts the integer value i to an unsigned vector with length w . This function is located in a package *numeric_std* (this package is in library *ieee*). However in case the generic (\sim constant) w is large this is a time consuming task. Therefore in this example the loop is ended in case i is equal to 20. The process ends with *wait*. This means the process will not resume execution.

Background information: numeric_std

The package *numeric_std* declares two types:

- signed (twos complement representation) and
- unsigned (binary representation).

Both types are similar to type *std_logic_vector*; arrays with elements of type *std_logic*

```

variable sa,sb,sc : signed(2 downto 0);
variable ua,ub,uc : unsigned(2 downto 0);
variable a,b,c : std_logic_vector(2 downto 0);

```

If sa is “111” then it is interpreted as -1 (twos complement).

If ua is “111” then it is interpreted as 7.

Is a is “111” then no number interpretation is associated with it!

What is the result of the statement: $sa := sb + \text{“11”}$?

The operands do not have the same length. Since sb is of type signed the shortest vector is sign extended before the addition takes places.

In case the operands are of type unsigned the shortest vector is extended with zeros.

In case the operands are of type *std_logic_vector* you cannot perform an addition because no number interpretation is associated with this type.

VHDL is a strongly typed language therefore you can not write:

```
a := sa;
```

However the types are closely related. In that case a type conversion function can be used:

```
a := std_logic_vector(sa);
```

If you want the integer value of a vector you simply write:

```
integer_value := to_integer(sa);
```

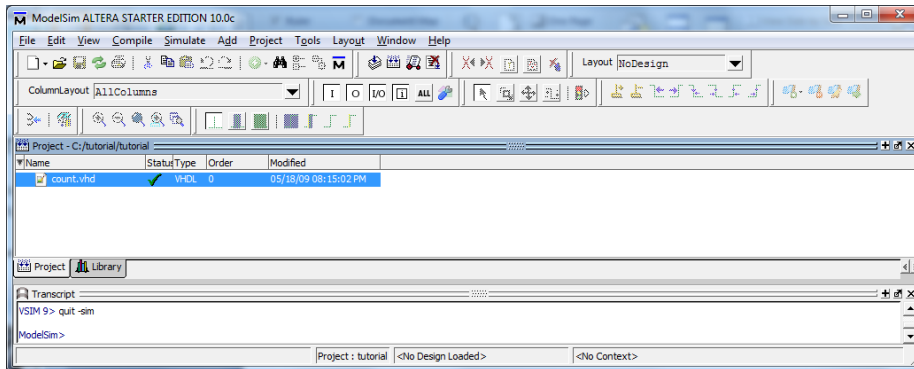
If you want to convert an integer to a vector you must add the length of the vector:

```
sa := to_signed(integer_value,3) or  
us := to_unsigned(integer_value,us'LENGTH)
```

The attribute LENGTH is used in the last example.

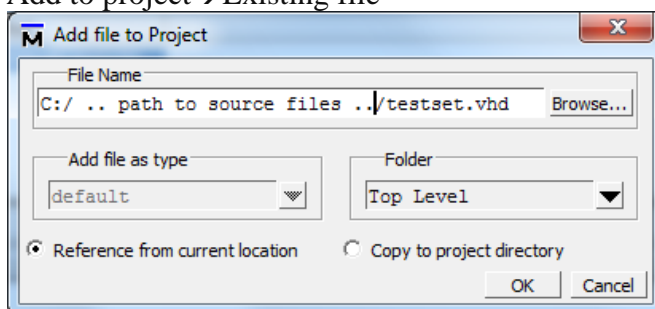
2.2.2.1 Add a design unit to the project

If the simulator is still active, end the current simulation via the simulate menu.



Perform the following steps to add *testset.vhd* to the project

1. Be sure the tab *project* is selected.
2. Right click in the project window.
3. Add to project → Existing file



4. Click on OK

Compile the design and perform a simulation. Since the design entity *testset* ends with a wait statement when all test patterns are applied you can use the command:

run -all <enter>

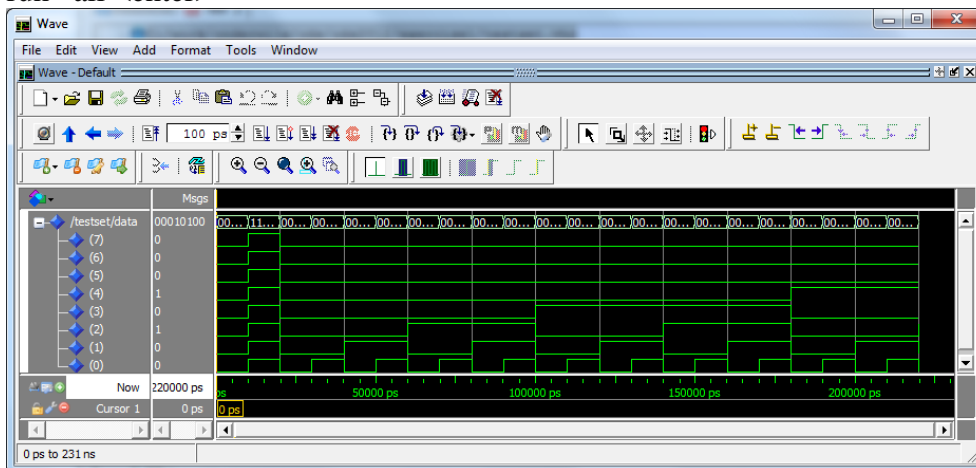


Figure 5: simulation result of the test set

2.2.2.2 Connect the test set with the design under test

Figure 6 shows the structural VHDL description that connects the design entity testset with design entity count. Add the file *testbench.vhd* to the project and simulate entity *testbench*.

Check that the length of the pattern is changed to 10 in the design (figure 6)!

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
ENTITY testbench IS
    GENERIC (width : positive := 10);
END testbench;

ARCHITECTURE structure OF testbench IS
    COMPONENT testset
        GENERIC (w : positive := 8);
        PORT (data : OUT std_logic_vector(w-1 DOWNT0 0));
    END COMPONENT;
    COMPONENT count
        GENERIC (w : positive := 8);
        PORT (a : IN std_logic_vector(w-1 DOWNT0 0);
              q : OUT integer RANGE 0 TO w);
    END COMPONENT;

    -- local connections
    SIGNAL stimuli : std_logic_vector(width-1 DOWNT0 0);
    SIGNAL output : integer;
BEGIN
    ts : testset
        GENERIC MAP (w => width)
        PORT MAP ( data => stimuli);
    dut : count
        GENERIC MAP (w => width)
        PORT MAP ( a => stimuli,
                  q => output);
END structure;
```

Figure 6: test bench

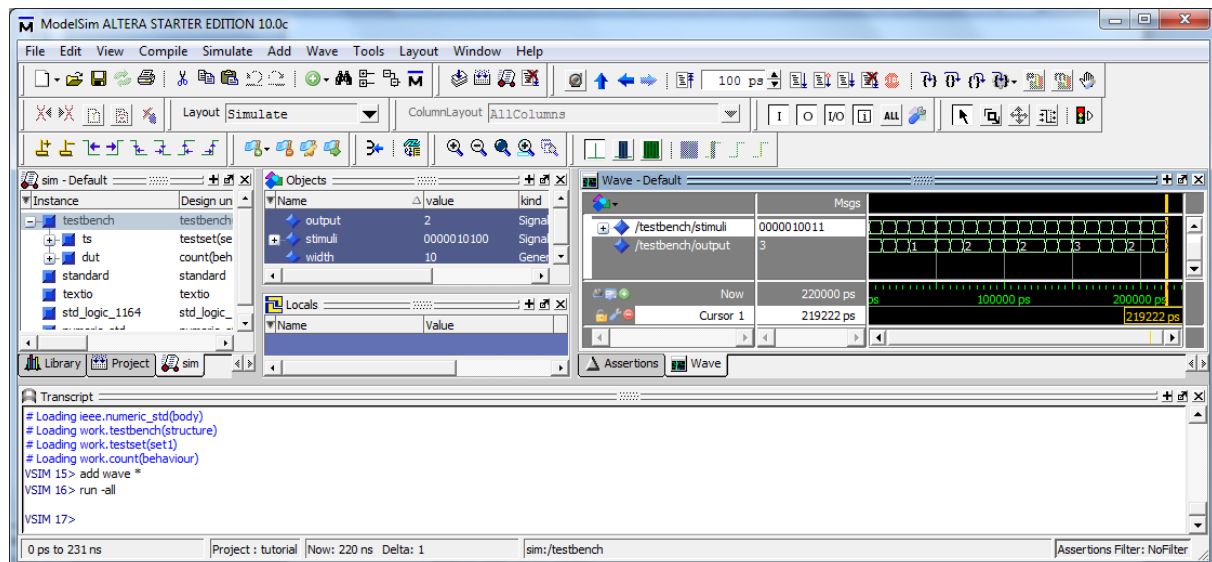
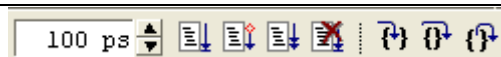




Figure 7: The design hierarchy is shown in the upper left window



With these buttons you can step through your design (e.g. to locate errors). The ‘step’ button  is often used. Then only one statement (concurrent or sequential) is executed. Also a source window is opened so you can see (the arrow) what the next statement to be executed.

“Step -Over”  is similar to the execution of a function/procedure in one step.

During debugging you often like to run your program to a certain point and perform a low level debugging from that point. Double click on the right of the line number of an executable line and a breakpoint appears.

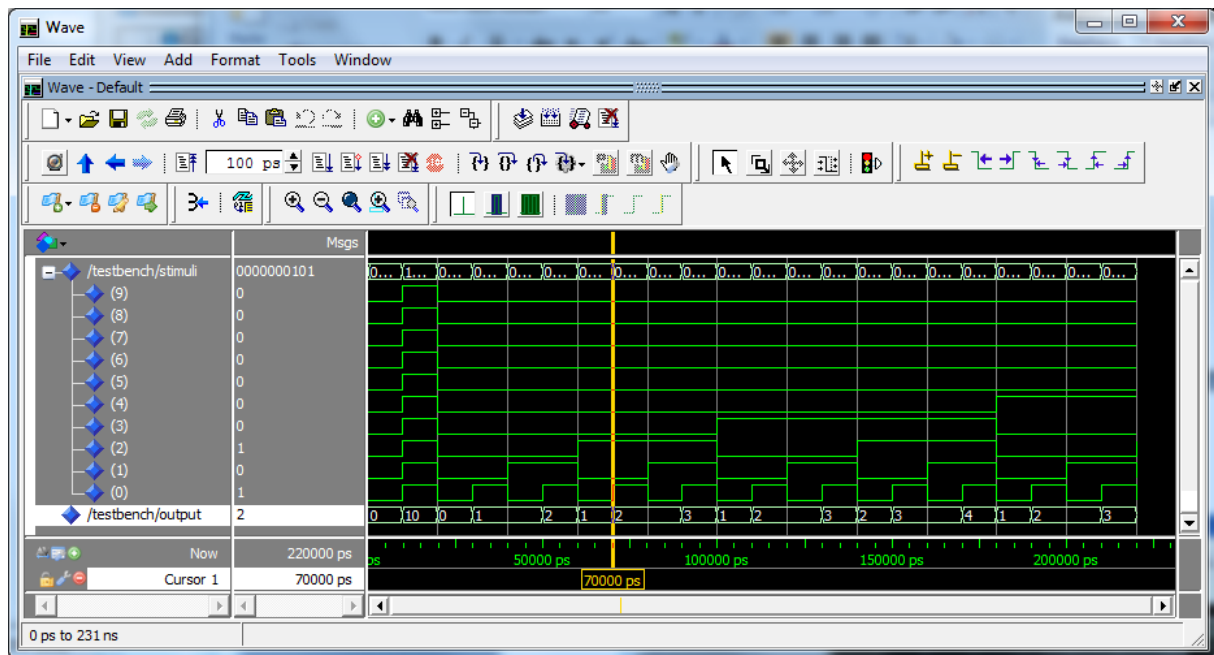


Figure 8: Simulation result

2.3 Simulation model

VHDL is a collection of concurrent statements. The order of the concurrent statements has no effect on the behaviour.

Concurrent statements can only communicate with each other using signals (I and also you should forget here the *shared variable*; don't use shared variables!). If you assign a value to a signal that signal value is not updated immediately. This means that all processes will use the same signal values; consequently the simulation is order independent w.r.t concurrent statements. If you assign a value to a variable that variable is updated immediately.

Users who are not that familiar with VHDL are often surprised by this update mechanism and the consequence it has for simulation and synthesis.

If you write:

```
y <= a after 10 ns;
```

The output *y* follows the input *a* with a delay of 10 ns. (More precise the signal *a* must be stable for 10 ns too.)

If you write:

```
y <= a;
```

The output is updated after a delta delay. Delta delays are not shown in the wave window. There can be an infinite number of delta delays before simulation time advances. ModelSim will report a warning when it has executed 1000 delta steps.

ModelSim can show the result of signals after a delta step. Repeat the previous simulation but (also) use the following command:

add list * <return>

Check that you really understand what is going on.

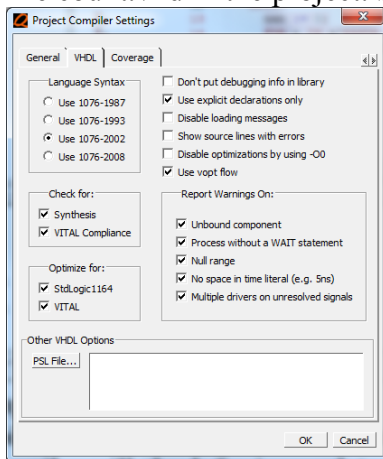
3 Synthesis with Quartus II

With Quartus II® a VHDL description is synthesized for Altera devices.

Read Appendix F if a *License Setup Required* appears.

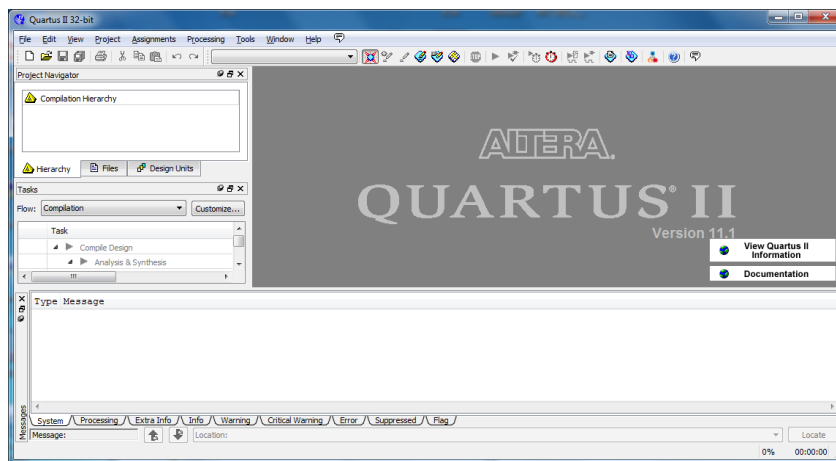
Notes

- 1 Although not required for VHDL it is advised that the name of the file is the same of the name of the entity.
- 2 In the VHDL description the pin locations of inputs and outputs are not specified (although it is possible).
- 3 Most synthesis tools do not handle the sensitivity list correctly. The synthesis tool assumes that the sensitivity list includes all signals read in the process. A mismatch between simulation and synthesis can occur if the concurrent statement does not model synchronous hardware.
ModelSim has an option that checks the sensitivity list at compile time. Right click the file count.vhd in the project. Select VHDL and mark “check for Synthesis”.



The constraint file (with the file extension *qsf*) should be in the same directory as the design. The constraint file contains the pin locations of the input and output signals. If no constraint file is added the software maps an input and output to a pin. For now will skip the constraint file (see chapter 5). **The constraint file should be in the directory before you start II if you want to program a device afterwards.**

3.1 Start Quartus II



Note

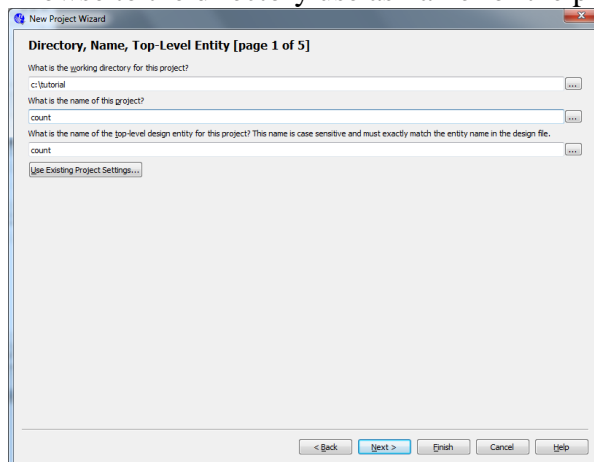
The first time you start Quartus II you can choose between Quartus II or MaxPlus look and feel. Choose: Quartus II. Next you are asked if you have purchased an IP library, which is probably not the case, so choose: run the Quartus II software.

3.2 Create a new project

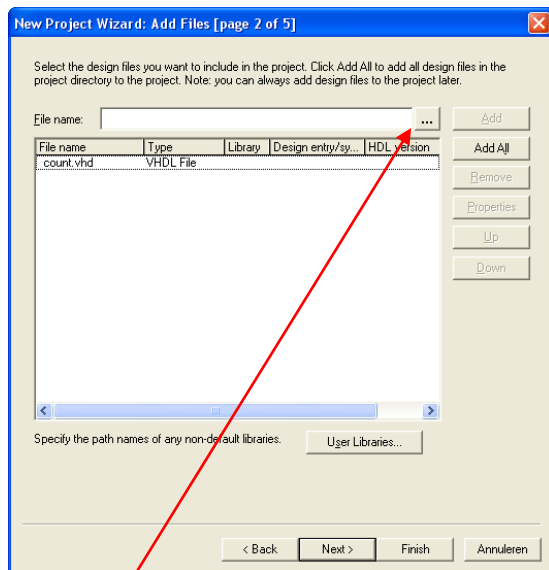
File→New Project Wizard

Skip the Introduction→Next

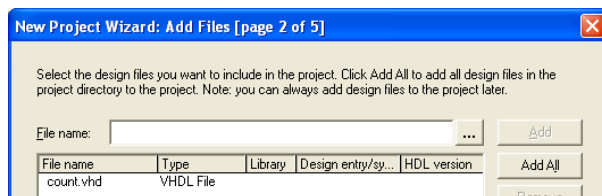
Browse to the directory use as name for the project: **count**



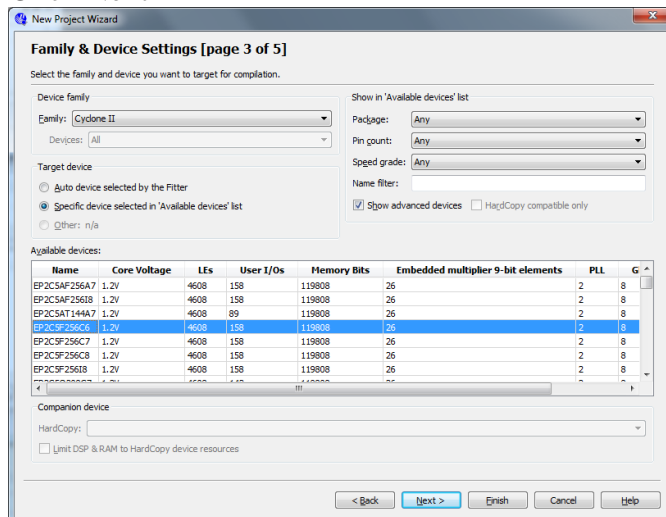
Click Next



Select the file(s) you want to include: count.vhd. Don't forget to click on the *add* button afterwards!



Click Next

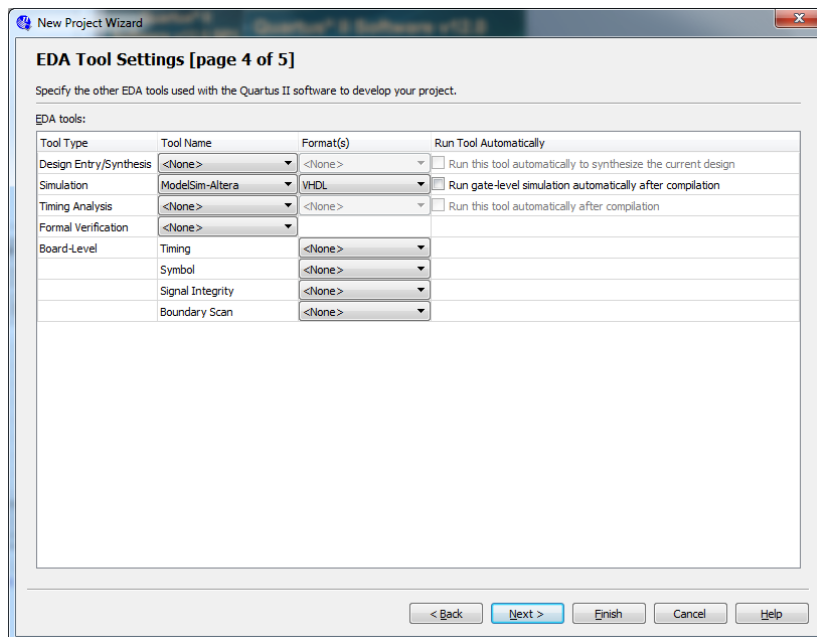


Select the device that is on the board. In this tutorial a Cyclone II device is selected.

Note

The latest web edition of Quartus II does not support Cyclone (Cyclone II is a newer device). The Altium Live Design board has device EP1C12F324C8. The licensed version in the lab supports Cyclone (and the web edition of Quartus II version 10).

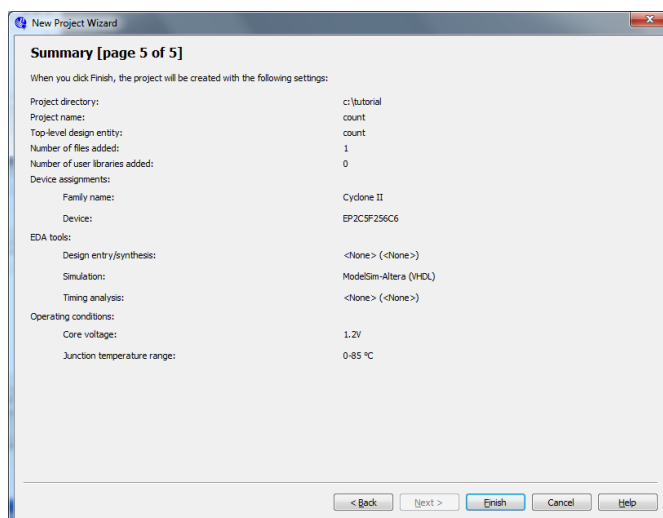
Click Next.



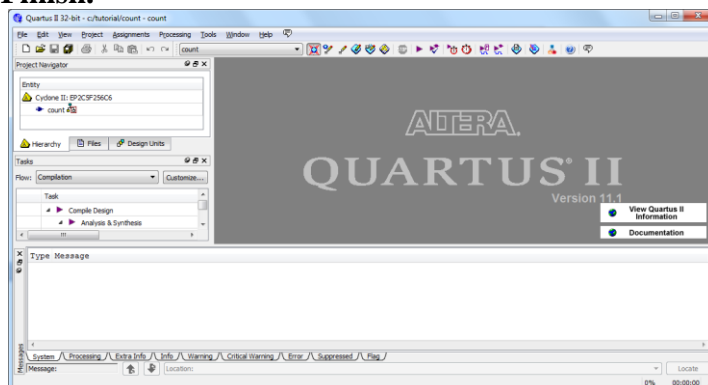
Simulation → Tool name: ModelSim-Altera and Format: VHDL

This allows post simulation of the design. This will be discussed in chapter 4.

“Next”

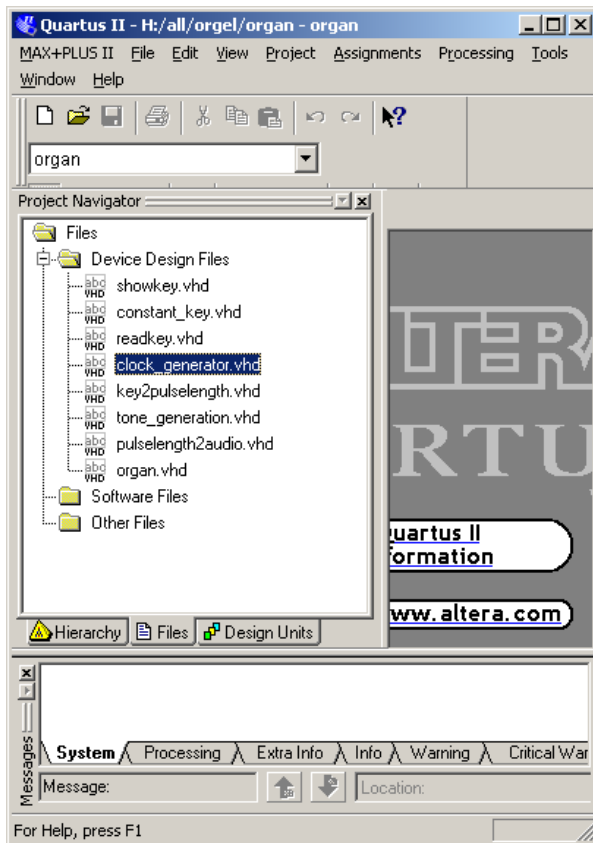


Finish.



3.3 Top level of the design

A design often has multiple design files. The top level of a design is the same as the name of the Quartus II project. In this tutorial it is *count*.



The picture above is from another design

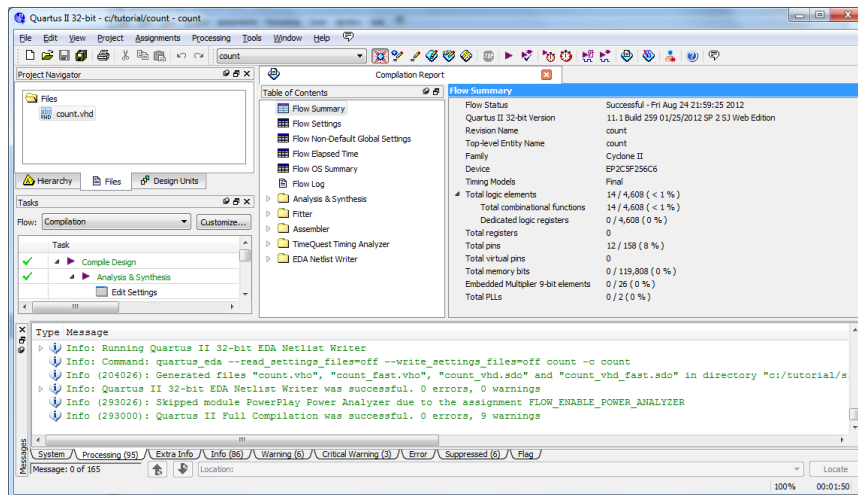
With the mouse right click on the file that contains the top level of the design. Now you can change the top level design.

3.4 Compile (=synthesize)

Processing → Start Compilation

Note

1. Only a subset of VHDL is synthesizable therefore it is possible that Quartus II, and other synthesis tools, generate error messages!
2. Synthesis tools generate many warnings! For some warnings you should be an experienced user.

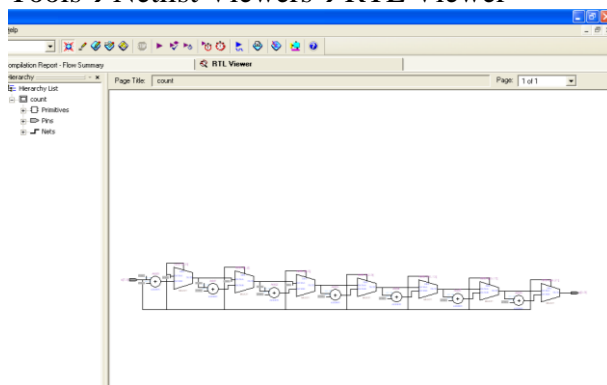


3.5 RTL viewer/Technology Map Viewer

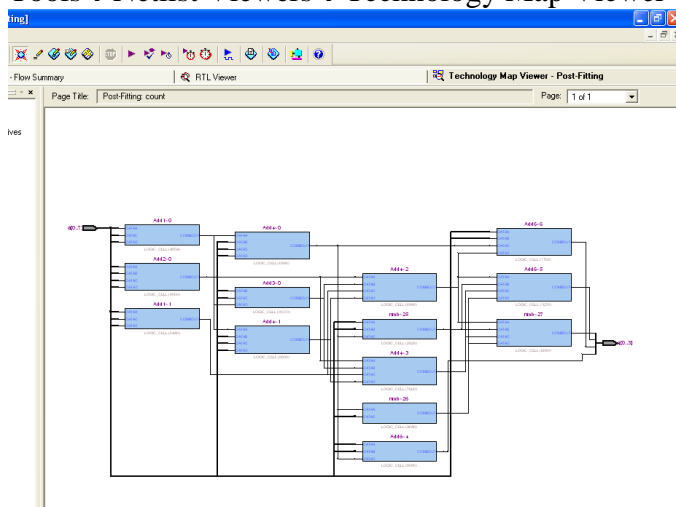
After synthesis a schematic can be generated:

- RTL view; this view is very close to the VHDL description.
- Technology view is an optimized result for the given technology.

Tools→Netlist Viewers→RTL Viewer



Tools→Netlist Viewers→Technology Map Viewer (post-mapping)



Notice that the loop structure in the VHDL description is visible in the RTL view whereas in the Technology Map view it is realized with an efficient tree like implementation.

4 Post simulation

ModelSim-Altera includes compiled versions of the Altera libraries that are required for post simulation. **For QuestaSim the Altera libraries have to be compiled first.** For more information see page 33.

After synthesis a post simulation can be performed with the real delays (as expected by Quartus II for the device). During simulation several constraints are checked, e.g. setup and hold times violations.

In the project directory the following directory is created:

<project_directory>\simulation\modelsim

In this directory there are the files:

- count.vho (VHDL output file)
- count_vhd.sdo (standard delay output file)

Note:

The names and extensions of these files depend on the synthesis tool.

A fragment of the generated VHDL file is (Quartus II version 11):

```
--
-- This VHDL file should be used for ModelSim-Altera (VHDL) only
--

LIBRARY CYCLONEII;
LIBRARY IEEE;
LIBRARY STD;
USE CYCLONEII.CYCLONEII_COMPONENTS.ALL;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE STD.STANDARD.ALL;

ENTITY count IS
  PORT (
    q : OUT STD.STANDARD.integer range 0 TO 8;
    a : IN std_logic_vector(7 DOWNT0 0)
  );
END count;
```

Note:

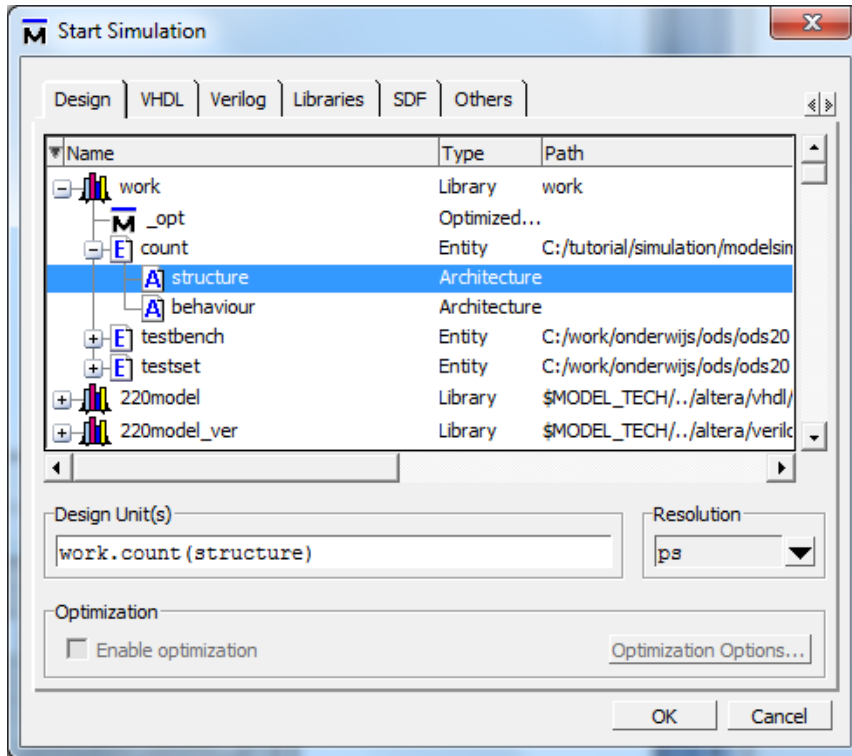
1. The port declaration of the generated post simulation model is tool dependent. A previous version of Quartus II always used types *std_logic* and *std_logic_vector*. E.g.

```
ENTITY count IS
  PORT (
    a : IN std_logic_vector(7 DOWNT0 0);
    q : OUT std_logic_vector(3 DOWNT0 0)
  );
END count;
```

This is closely related to the realization (4 pins are used for the integer range 0 to 8). A disadvantage is that if the generated post simulation model is verified using the testbench the mismatch between the types used in the component instantiation and those in the generated entity declaration will raise an error: some rewriting is necessary. Advanced users probably use type conversion functions in the *port map*. Often one of the guidelines in a company is that you must use *std_logic* and *std_logic_vector* at the top level entity. Then you never have a mismatch because all synthesis tools will use these types also in the generated post simulation model.

To perform a post simulation:

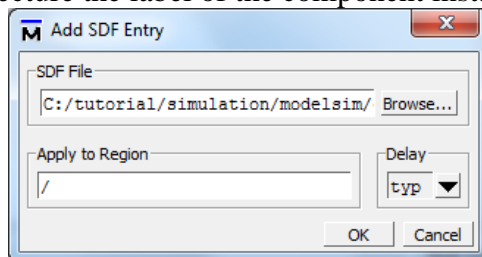
1. Add the file <project_directory>\simulation\modelsim\count.vho to the project.
2. Compile file count.vho
3. simulate→start simulation



- a. Tab design:
Select your design count(structure) and set the resolution to “ps” (The delays in the SDF files is in pico seconds).
 - b. Tab SDF
Select file <project_directory>\simulation\modelsim\count_vhd.sdo (via ADD)
4. Perform a simulation.

Notes:

1. You can also apply a region when the SDF file is selected. This is required in case the synthesized design is part of a larger system (e.g. a component instantiation). The path to the component is the region (e.g. if it is instantiated in the top-level architecture the label of the component instantiation that is used).



2. The default run length in ModelSim-Altera is 100 ps. The default run length can be any other value. Therefore explicit add the time unit in a script file; e.g. “run 100 ns” instead of “run 100”.

5 Constraint file

In the top-level entity description the input and output ports are enumerated. However the physical pin to which a port is connected has to be specified. This can be done in a constraint file.

An example of a constraint file for Quartus II is:

```
set_location_assignment PIN_E16 -to q[0]
set_location_assignment PIN_G14 -to q[1]
...
```

Fragment of the constraint file count.qsf

For Quartus II the name of the constraint file should be the same as the name of the top-level entity. The file should be located in the design directory.

Notes

1. During synthesis Quartus II will change the contents of the qsf file. **Your constraint file should be in the design directory before you start Quartus II.**
2. Precision RTL (see page 26), and other tools, often use the SDC (Synopsys Design Constraint) file also for the pin mapping. Precision RTL requires that the SDC file is explicitly included. The syntax is a little bit different:
 set_attribute -name pin_number -value E16 -port q(0)
 set_attribute -name pin_number -value G14 -port q(1)
3. Quartus II uses the SDC file for timing constraints. E.g. if a synchronous design has a clock with name *clk* and it operates at 100 MHz. This information is put in the synopsys design constraint file: <top_level_design_name>.sdc
 create_clock -period 10.000 -name clk [get_ports clk]

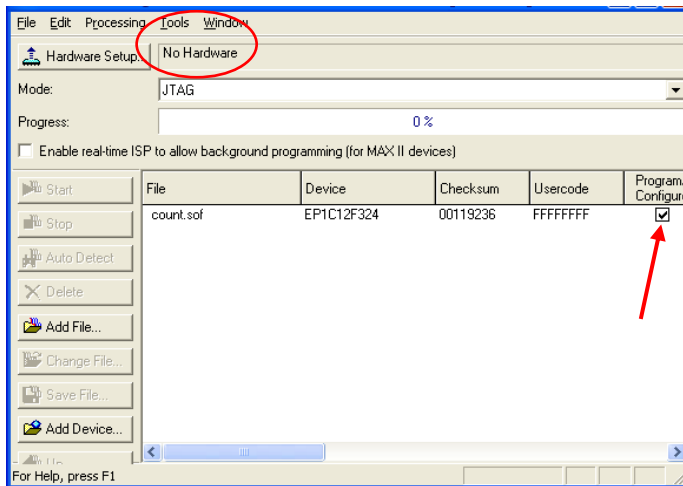
6 Programming the *LiveDesign Evaluation kit*

The constraint file with the pin mapping is required! See page 24.

If the the *Cyclone II FPGA Starter Development Kit* is used the first time the USB Blaster programming hardware is to be selected (see Appendix G).

To program the device perform the following steps:

1. Connect the programming cable to the parallel port of the PC
2. Connect the power supply
3. Program the device: Tools→Programmer and select the file count.sof. Select “program/Configure. Next: start.



Note

1. Sometimes the JTAG interface of the *LiveDesign kit*, used to program the device, cannot be found. Solution: disconnect and connect the power supply of the board.

Appendix A Synthesis with Precision RTL

Precision RTL is a technology independent synthesis tool. It can be used on top of Quartus II (for Altera devices), or ISE (for Xilinx devices), and more. It also can perform retiming: moving flip-flops in the design to increase the maximum clock frequency. For our course Precision RTL is not needed. However the tool generates an RTL schematic very quickly!

Start Precision RTL.

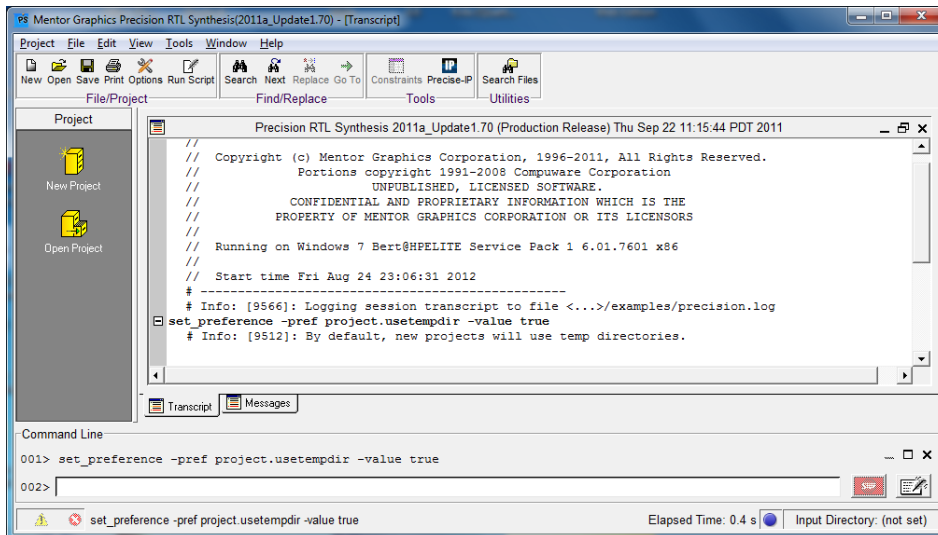


Figure 8: Precision start up screen

Click on “New Project” and browse to the folder that contains the source files (file *count.vhd*.) (choose a project name, and *create impl*(ementation) should be selected (default)).

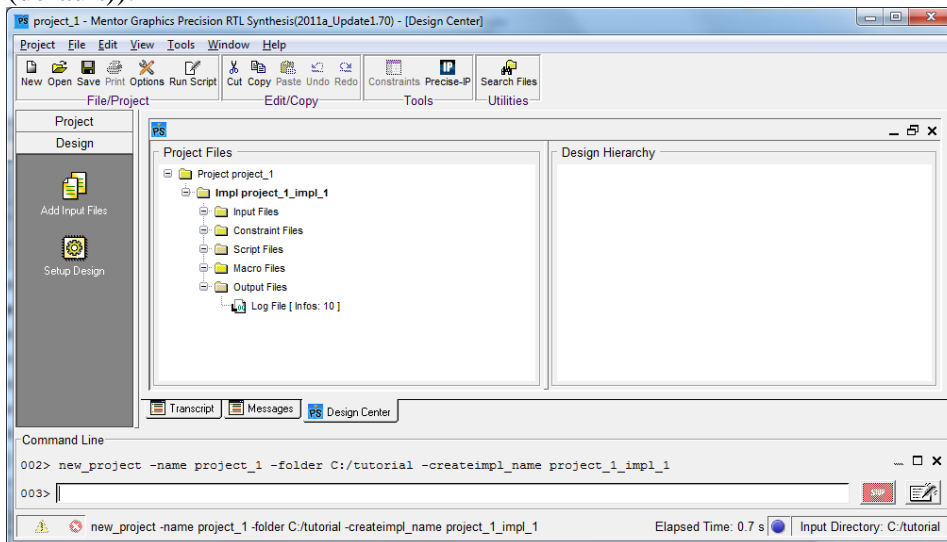
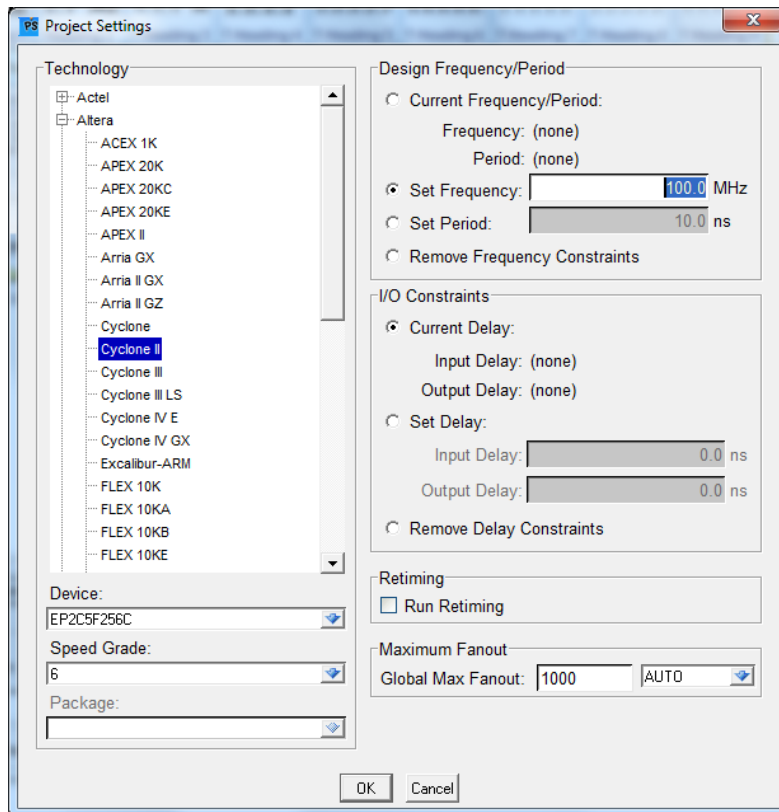


Figure 9: Precision “Design” tab

Setup Design (target device)

Click “Setup Design” and select device, e.g.



Click OK.

Input files

Click “Add Input Files”.

Note

The order is not important. The tool automatically reorders the files.

Select the design files and press OK.

Compile the design

Press the compile button.

The tab “Design analysis” is added at the lower left corner.
Select this tab, and choose “View RTL Schematic” (figure 10).

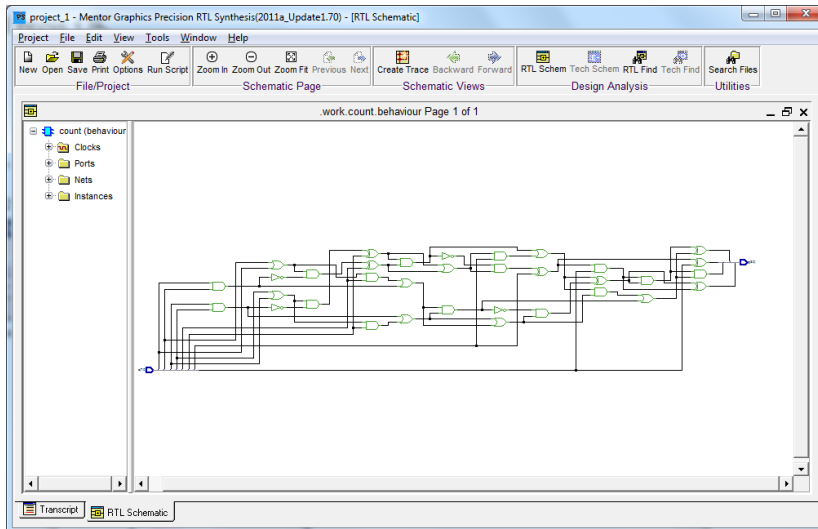


Figure 10: schematic view of the design

The compile step is an intermediate step (still technology independent).
 Tab Design→Synthesis will map the intermediate result to the chosen technology.

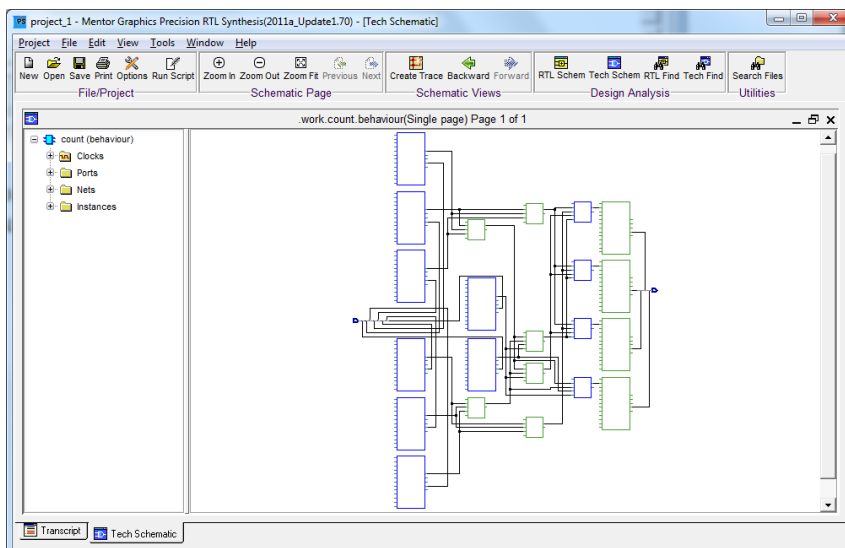


Figure 11: schematic view of the design with unbundled nets.

After the synthesis step a new tab “Quartus II” appears and you can execute Quartus II.

Note

If you had chosen a device from Xilinx then you can execute the Xilinx software ISE instead of Quartus II.

Appendix B An alternative description for count

The behavioral description of count (file count.vhd) was a straightforward and is a readable description. Synthesis tools nowadays support these kinds of descriptions, and often find smart implementations. The tree like implementation is used by the synthesis tool because it knows the properties of the addition operator (associative and commutative).

In case the tool does not use these properties you can force the tree like structure using a recursive description (figure 12).

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY count IS
    GENERIC (w : positive := 8);
    PORT (a : IN std_logic_vector(w-1 DOWNT0 0);
          q : OUT integer);
END count;

ARCHITECTURE recursive OF count IS
    -- count bit with a balanced tree approach
    FUNCTION count_bits(vec: std_logic_vector) RETURN integer IS
        CONSTANT n: natural := vec'LENGTH;
        CONSTANT v: std_logic_vector(1 TO n) := vec;
    BEGIN
        CASE n IS
            WHEN 0 => RETURN 0;
            WHEN 1 => IF v(1) = '1' THEN
                        RETURN 1;
                    ELSE
                        RETURN 0;
                    END IF;
            WHEN OTHERS => RETURN count_bits(v(1 to n/2)) -- 1
                                + count_bits(v(n/2+1 to n));
        END CASE;
    END count_bits;
BEGIN
    q <= count_bits(a);
END recursive;
```

Figure 12: recursion (file count_recursive_funct2.vhd)

Remember that the function count_bits, figure 12, has as input a std_logic_vector. This is an unconstrained array. Not only the length but also the left and right indexes are not known. Therefore the two constant declarations are used.

Synthesize this design and have a look at the RTL schematic.

¹ The division operator, with integer types, will always give an integer result. Hence 8/3 is 2.66666 and the integer result is 2.

Appendix C Verification of a design via simulation

Modelsim-Altera starter does not have support for *Code coverage* and *PSL*

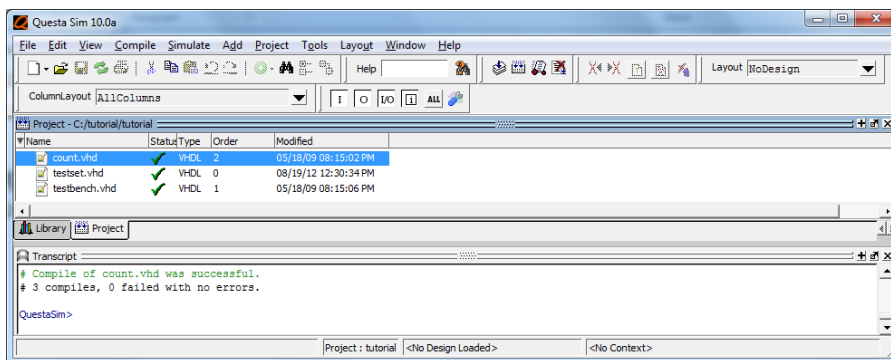
Code coverage

QuestaSim supports several type of code coverage's, e.g. statement coverage, branch coverage, condition coverage etc.

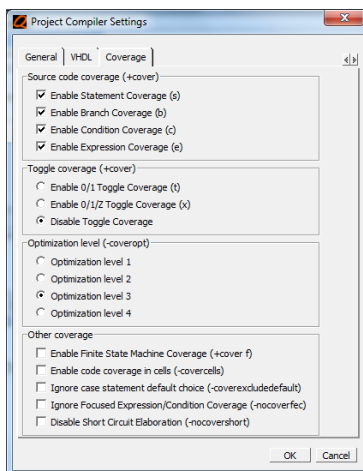
A detailed description of the code coverage features is found in chapter 17 (Help → Questa documentation - PDF Bookcase → Questa Sim manual (chapter 20 code coverage for QuestaSim 10.0a)).

This chapter only illustrates code coverage with the counter example. We are interested in all type of code coverage's in the design *count*. We start with the project with the behavioral description of count.

The relevant files, including the test environment, are: *count.vhd*, *testset.vhd* and *testbench.vhd*



Right click file *count.vhd* in tab *Project* and select *properties* and tab *coverage*.



In this tutorial all four *source code coverage* checks are selected. Click OK.

File count.vhd is to be compiled again (the coverage information is “added”):
Right click in project field → compile → compile all.

Now simulate the design by right click design unit testbench (tab *library* should be selected) and select: *Simulate with Coverage*.

Double click on *dut* and the lines that are not covered are marked with an **X**.

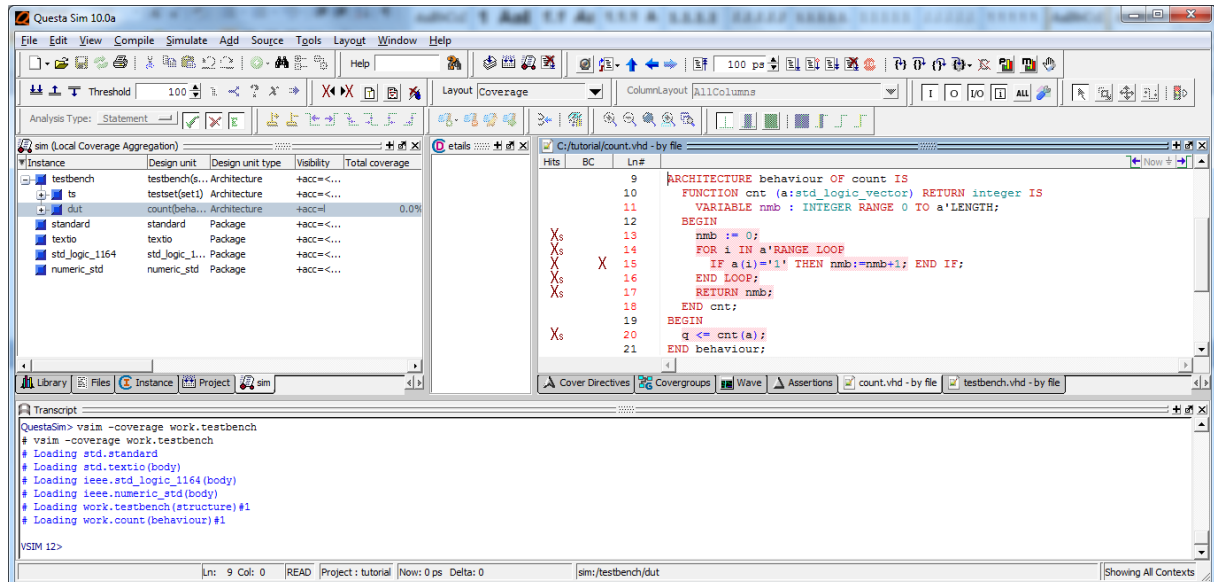
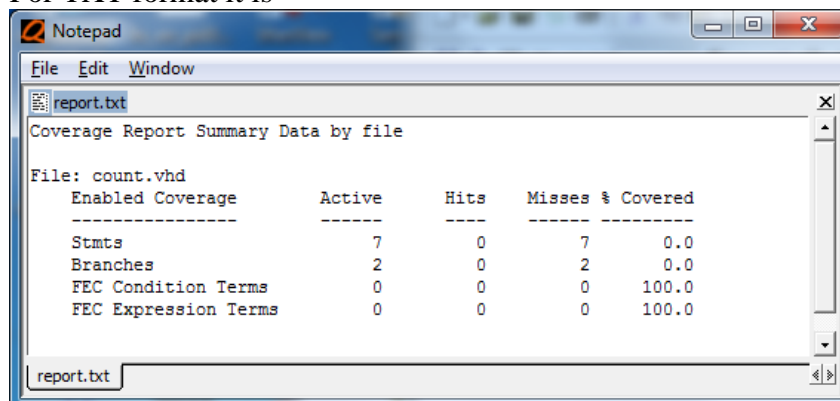


Figure 13: result of code coverage at the start of the simulation

Notes

1. If you select an 'X' with the mouse an explanation is shown.
2. A coverage report can also be generated:
Tools → Coverage report → <select your format>
For TXT format it is



Assertion Based Verification with PSL

To be done.

Appendix D ModelSim/QuestaSim if no project is used

In this tutorial a project is used. However, the use of a project is not compulsory. You can also enter commands in the transcript window (or the commands are in a script file) or use the *GUI*.

The counter example is used again, and the relevant files are:

```
<path>\count.vhd  
<path>\testset.vhd  
<path>\testbench.vhd
```

1. Start Modelsim-Altera (or QuestaSim)
(close the project if the last project is opened: File→Close)
2. File→Change Directory and browse to the directory with the source file (<path>) and click OK
3. vlib work <return>
The work directory is created. The location is <path>\work
The compiled design units are located in library WORK. Never place your source file in WORK, Never! (because before you know it they are deleted by the tooling or the student assistant ☺). The WORK directory is only for ModelSim.
4. vcom count.vhd <return>
If you want to check for code coverage in the design, see page 30, you could use the command: vcom count.vhd +cover <return>. That's easy!
5. vcom testset.vhd <return>
vcom testbench.vhd <return>
6. Have a look at the *GUI*. The compiled design are in library work. Via the GUI you can start a simulation (similar to the project style approach). You can also start the simulation via a command:
vsim testbench
(or if you want to simulate with code coverage type:
vsim -coverage testbench

Notes

1. If the previous commands are in a file you can easily reproduce a correct compilation (order) and simulation.
You can execute this script file in ModelSim with;
do <file name> <return>
2. The files should be compiled in a correct order. E.g. the file with the entity before the file that contains the architecture.
3. Delete a design unit (in library work).
vdel <name of design unit> <return>
4. If all design units are to be deleted
vdel -all <return>
Library work is deleted. So before you can compile a file you should first create library work (i.e. vlib work).
5. Hint: sometimes you have the feeling that something is totally wrong. Delete the work library (vdel -all) and compile the design files again (use a script file).
6. If you use the *GUI* then (often) the command is shown as command (command symbol #) in the transcript window. This command can also be used in a script file.

Appendix E Compiling Altera libraries with QuestaSim

The procedure in this appendix is not for ModelSim-Altera. ModelSim-Altera already includes compiled versions of the Altera libraries.

Instead of compiling the libraries with the procedure in this appendix you could also copy compiled versions into your home directory: P:/.

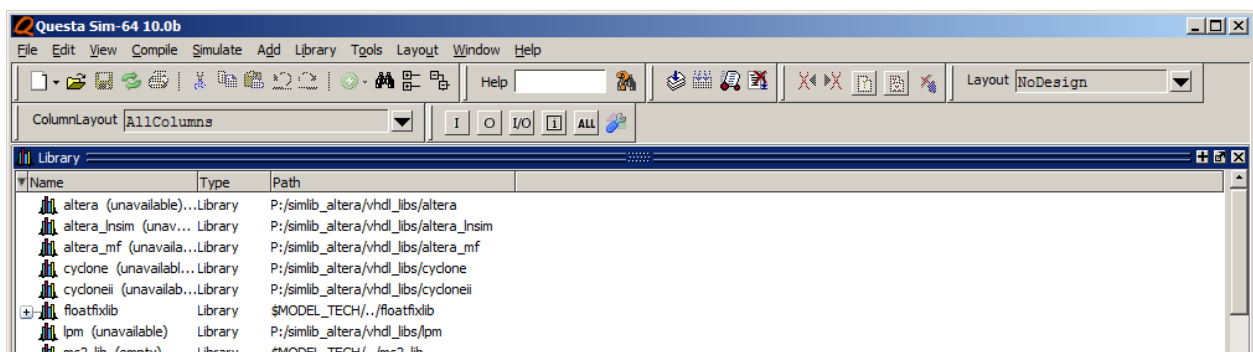
See for more information (read the readme):

http://www.cs.utwente.nl/~molenkam/ods/Postsim_lib_altera/

In QuestaSim (the licensed version installed on the lab machines) pre-compiled libraries of the Altera libraries are not installed. Therefore these libraries have to be compiled first before a post simulation can be performed.

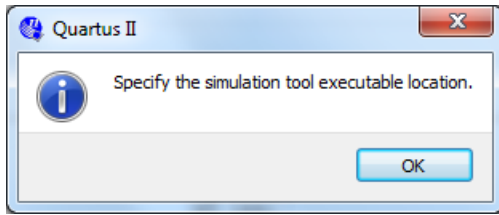
In courses we often use a Cyclone or Cyclone II device. The location of the libraries is stored in the file modelsim.ini. This file is write protected for non admin users. Therefore the locations of the libraries are already added in the file modelsim.ini: **P:\simlib_altera**

This is your personal directory, and you probably did not compile these libraries! That is also the reason why the libraries are “unavailable” in QuestaSim.



Compile the libraries Cyclone and Cyclone II

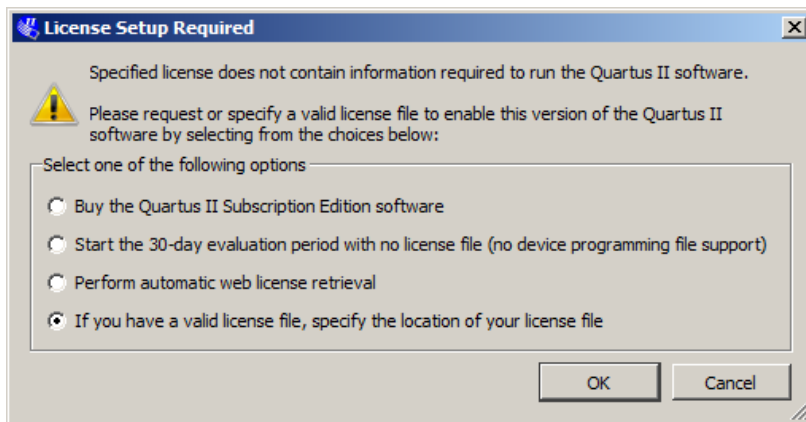
1. Start Quartus II
2. Tools→Launch Simulation Library Compiler
3. The settings are:
 - Tool name: QuestaSim
 - Executable location: C:/Programs/questasim64_10.0b/win64
(do not include “vsim.exe” and use a forward slashes (not back slashes) in the path)
 - Only select the families Cyclone and Cyclone II
 - Library language: select VHDL
 - Output directory: P:\simlib_altera
 - Select *Show all messages* and *Create log file*.
 - Start compilation



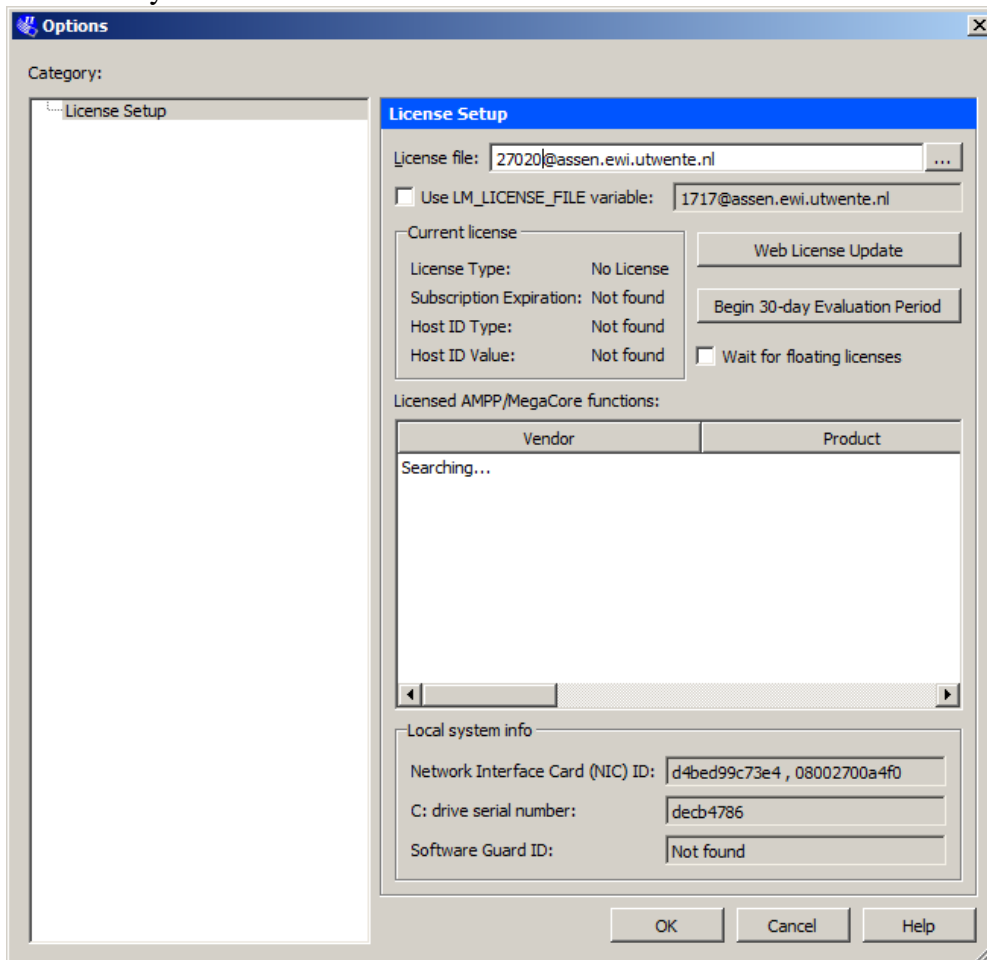
If you get this error message then check the path to the tool! You should use a forward slashes in the path. If this does not work: e.g. add a space and remove that space ... and now it probably works, I hope.

Appendix F Quartus II, License?

The first time Quartus II is executed you probably get the License Setup window.



Select that you have a valid license file.



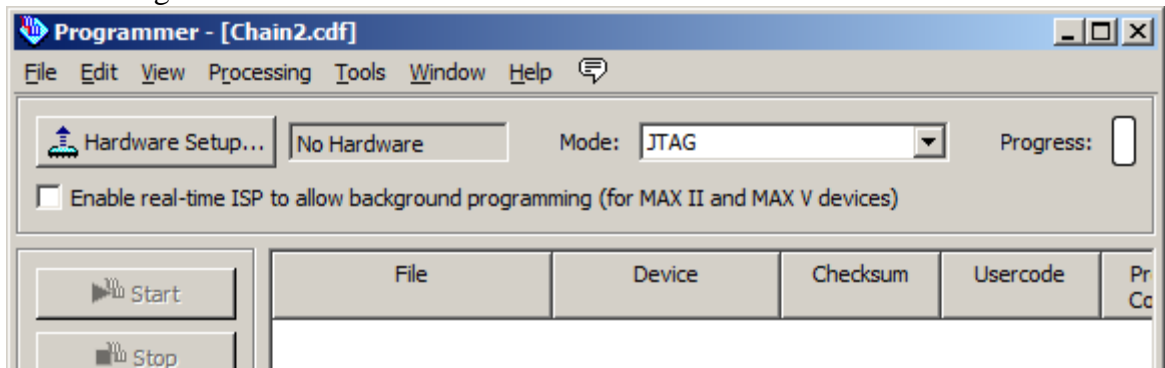
In the box license file write: **27020@assen.ewi.utwente.nl**
Click OK.

Appendix G Quartus II USB Blaster (programmer)

The procedure in this appendix is only required if you use a USB cable to program the device.

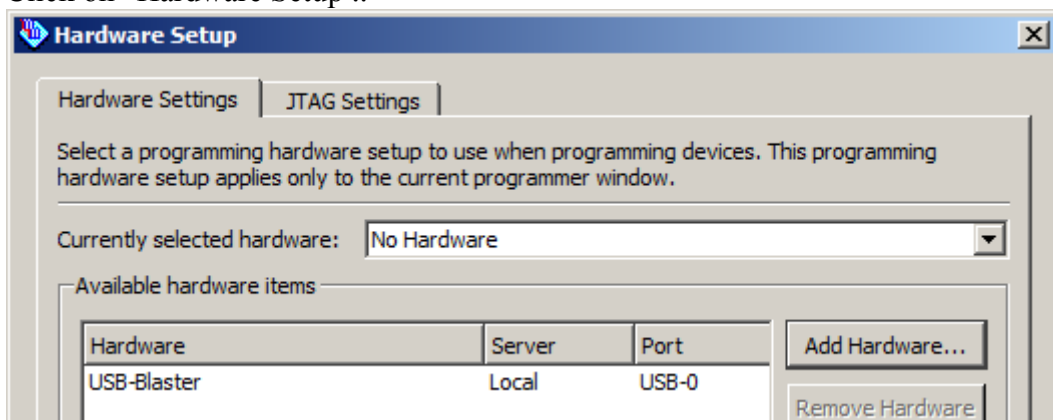
To program a device

1. Start Quartus II
2. Tools→Programmer

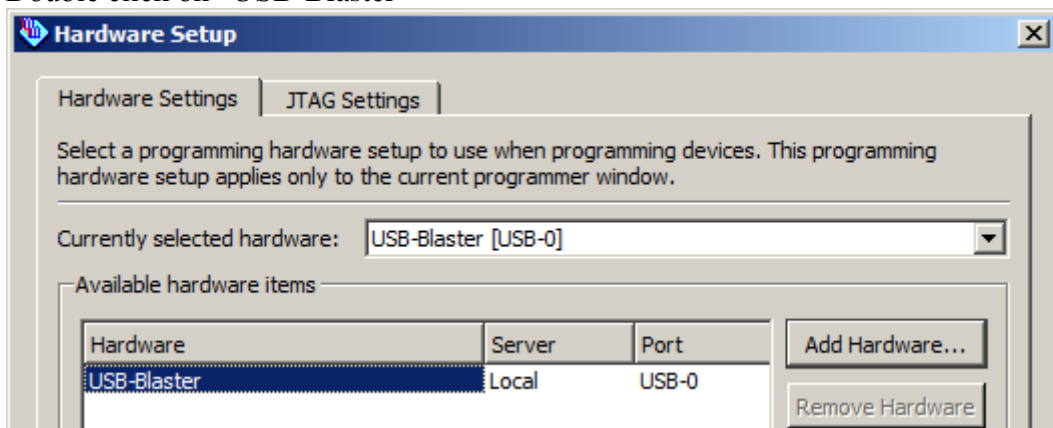


If your board uses a USB blaster, E.g. for programming the Cyclone II FPGA Starter Development Kit, then the first time the hardware is not found “No hardware”.

3. Click on “Hardware Setup ..”



4. Double click on “USB-Blaster”



5. Close the window

Appendix H Location of the software on the lab machines

Since you can use many programs on the lab machines there are no icons on the desktop. So the first time software is used you have to know the location of it.

You can then also create a short cut on the desktop.

The locations of the programs used in this tutorial:

1. QuestaSim
C:\Programs\questasim64_10.0b\win64\questasim.exe
2. ModelSim-Altera Starter
C:\Programs\altera\11.1\modelsim_ase\win32aloem\modelsim.exe
3. Quartus II
C:\Programs\altera\11.1\quartus\bin64\quartus.exe
4. Precision
C:\Programs\Mentor_Graphics\Precision_Synthesis 2011a_Update2.76\Mgc_home\bin\precision_gui.exe

Note:

The locations can be different on the lab machines. E.g. ICTS can use another policy with respect to the directories, or a new version of software is installed (often the version is included in the path).