HOGESCHOOL VOOR WETENSCHAP & KUNST | DE NAYER INSTITUUT
SINT-KATELIJNE-WAVER

DE NAYER Instituut
J. De Nayerlaan 5
B-2860 Sint-Katelijne-Waver
Tel. (015) 31 69 44
Fax. (015) 31 74 53
e-mail: ppe@denayer.wenk.be
ddr@denayer.wenk.be
tti@denayer.wenk.be
website: emsys.denayer.wenk.be

Basic Custom OpenRISC System Hardware Tutorial

Altera

Version 1.00

HOBU-Fund
Project IWT 020079

Title	:	Embedded systemdesign based upon Soft- and Hardcore FPGA's
Projectleader	:	Ing. Patrick Pelgrims
Projectassistants	:	Ing. Dries Driessens Ing. Tom Tierens

I Introduction

Purpose of this tutorial is to help you compose and implement a custom, OpenRISC based, embedded system in the eastiest way possible. Unexperienced users should be warned that the OpenRISC processor is quite difficult processor : the lack of a self-configuring embedded system-package and the more 'open' nature of the OpenRISC compared to other opensource processors like the Leon SPARC is one of the reasons for this.

Before proceeding, check if you have the following software and hardware:

Hardware:
- Linux-PC or Windows-PC
- Development board with Altera FPGA (minimum 8000LEs), UART and 6 available pins.

Software:
- Correctly built OpenRISC-GNU Toolchain
- Altera Quartus II 3.0 (or Quartus II Web edition) with service pack 2
- For Windows : WinCVS 1.2 (http://prdownloads.sourceforge.net/cvsgui/WinCvs120.zip)

If you experience problems building the OpenRISC-GNU Toolchain, there is also an OpenRISC Software tutorial available from our website (http://emsys.denayer.wenk.be).

The flow of implementing a custom, OpenRISC based, embedded system is:
A. retreive OpenRISC Platform HDL source-code
B. remove unnecessary components from source-code
C. adjust RAM module
D. synthesize and place & route OpenRISC based system
E. download
F. test the OpenRISC system

II Retreive Source Code

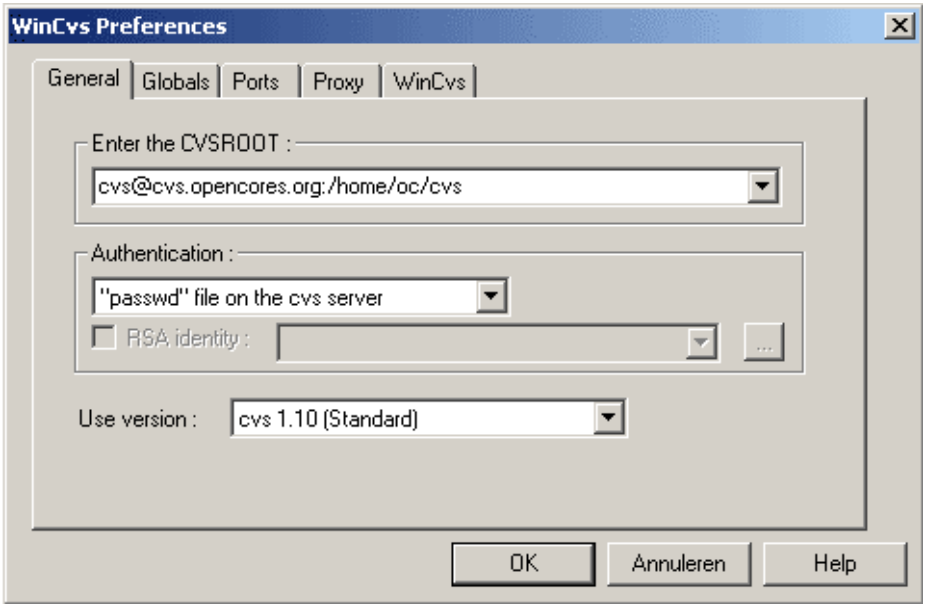
The OpenRISC Platform source code is available through a CVS server. To make sure that this tutorial doesn't get obsolete, the source code that will be downloaded is the same version that was used writing this tutorial.

IN WINDOWS:

Windows doesn't come with a cvs-client, so if you haven't installed wincvs yet, you need to download the CVS client WinCVS 1.2 (http://prdownloads.sourceforge.net/cvsgui/WinCvs120.zip). After downloading, install and open it.

First, configure WinCVS:
- goto menu item 'Admin' and click on 'Preferences'
- 'CVSROOT' should be cvs@cvs.opencores.org:/home/oc/cvs
- 'Authentication" needs to be "passwd file on the cvs server"
- disable "Checkout read-only" at the 'Globals' tab.

Figure 1: WinCVS settings



Next, goto the right location where you want to put the data downloaded from CVS: goto menu item 'View', then "browse location", click 'change' and choose your location.

Then you should login: goto menu item 'Admin' and click on 'Login'. When you're promted for a password, you can type anything.

Finally you can download the OpenRISC source code. Goto menu item 'Admin' and click on "Command Line...". Now type "cvs –z9 co –D 1/1/04 or1k/orp/orp_soc/rtl" in the command box.

To finish you just have to logout: goto menu item 'Admin' and click on 'Logout'

IN LINUX:

To begin, open a terminal so you have a prompt.

First, set the correct CVSROOT: type "export CVSROOT=:pserver:cvs@cvs.opencores.org:/home/oc/cvs"

Then go to the directory where you want to place the downloaded OR1K source code.

Next, login by typing "cvs login". When promted for a password, press enter.

Finally you can download the source code: just type "cvs –z9 co –D 1/1/04 or1k/orp/orp_soc/rtl"

To finish, just log out, by typing "cvs logout".

III Adjust Source Code

After retreiving the source code, comes the more complex part: adjusting the source code.
Keep in mind that we want to build a "basic custom OpenRISC system":

- 'Basic' because we only use a minimal set of peripherals: a debug-unit (input), onchip-RAM (for processing) and a UART (for output).
- 'Custom' because this tutorial isn't targeted for one or a few boards specifically. Everything is kept general so that anybody can implement the OpenRISC on his "Altera FPGA"-board.
- 'OpenRISC' because of the processor used.
- System': because the system we're building contains all the necessary components of an embedded system (input, processing and output).

1) Delete unnecessary files

In the 'Verilog' directory remove:

- the following directories: 'audio', 'ethernet', 'ethernet.old', 'or1200.old', 'ps2', 'ps2.old', 'svga' and 'uart16550.old'
- the file 'tdm_slave_if.v'.

2) Adjust 'xsv_fpga_defines.v'

Disable the 'define TARGET_VIRTEX' line by putting '//' before it.

3) Adjust 'xsv_fpga_top.v'

a. 'module xsv_fpga_top' part:

There are 4 groups of signals that are necessary (So if they're not present add them):

- 2 global signals (clk, rstn),
- 2 uart signals (uart_stx, uart_srx),
- 7 jtag debug signals (jtag_tvref, jtag_tgnd, jtag_tck, jtag_tms, jtag_trst, jtag_tdi, jtag_tdo),
- and chip enable signals to shut down any unused electronic ICs.

All other signals can be removed.

b. 'input and output' list:

For every signal that was used in the module part, you should define if it is either an input or an output:

- **7 inputs** : clk, rstn, tck, tms, tdi, trst, uart_srx
- **4 outputs** : tvref, tgnd, tdo, uart_stx

c. 'internal wires' list:

The following wires can remain: debug core wires, debug<->risc wires, RISC data & instruction master wires, "SRAM controller slave i/f" wires (for the onchip RAM), UART16550 wires, UART external wires, JTAG wires and your custom chip enable wires (so that you can assign vcc or gnd).

d. ‘assign’ part:

In the following part of the ‘xsv_fpga_top.v’ file, first insert an interface for a PLL-module because the clock speed is too high with most FPGA development boards:
So replace : “assign wb_clk = clk;”
By “clkdiv clkdiv_inst (
.inclk0 (clk),
.c0 (wb_clk));”

Remove “SRAM tri-state data”, “Ethernet tri-state”, “PS/2 Keyboard Tristate”, “Unused interrupts” and “Unused WISHBONE signals” (except for “assign wb_us_err_o = 1’b0;”)

Also do not forget to remove the “RISC Instruction address for Flash” part.

You can now add the necessary assignments for the chip enable signals and jtag_tvref/jtag_tgnd. Every assignment looks like “assign signal_to_be_assigned = 1’b0;” (or “1’b1;”)

e. ‘instantiations’ part:

Remove the following instantiations: “VGA CRT controller”, “Audio controller”, “Flash controller”, “Ethernet 10/100 MAC”, “PS/2 Keyboard Controller”, “CPLD TDM”.

The following adjustments have to be made into the remaining instantiations:
- change the ‘clk’ of ‘or1200_top’ in the divided clock ‘wb_clk’.
- the last connection ‘pic_ints’ of the ‘or1200_top’ instantiation should be replaced by “20’b0”
- “sram_top sram_top” should be replaced by “onchip_ram_top onchip_ram_top”
- all the external SRAM connections should be removed (‘// SRAM external’ part of ‘sram_top’)
- in the ‘uart_top’ instantiation the “.int_o (pic_ints[`APP_INT_UART])” must be replaced by “.int_o ()”
- in the Traffic cop instantiation, the following changes should be made:
  - MASTERS: Wishbone Initiators 0, 1 and 2 must be replaced by stubs (like Initiators 6 and 7)
  - SLAVES: Wishbone Targets 1, 2, 3, 4 and 6 must be replaced by stubs (like Targets 7, 8)
  - “.i4_wb_ack_o (wb_rdm_ack)” must be “.i4_wb_ack_o (wb_rdm_ack**_i**)”
  - and “.i5_wb_adr_i (wb_ri**f**_adr)” must be “.i5_wb_adr_i (wb_ri**m**_adr**_o**)”

4) Adjust ‘or1200_defines.v’

Several defines should be enabled or disabled:
- Enable “`define OR1200_ALTERA_LPM”
- Disable “`define OR1200_XILINX_RAMB4”
- Enable “`define OR1200_NO_DC”, “`define OR1200_NO_IC”, “`define OR1200_NO_DMMU” and “`define OR1200_NO_IMMU”
- Disable “`define OR1200_CLKDIV_2_SUPPORTED”
- Disable “`define OR1200_RFRAM_DUALPORT”
- Enable “`define OR1200_RFRAM_GENERIC”
- Disable “`define OR1200_DU_TB_IMPLEMENTED”

5) Adjust ‘or1200_pc.v’

Remove the line “.genpc_stop_refetch (1’b0)” completely.

IV Add new components

1) PLL component

Because most FPGA development boards contain clocks that are too high, these have to be divided so that the OpenRISC can run. You can always run your OpenRISC at a higher frequency, but usually 10 to 20 MHz is attainable on most FPGA's.

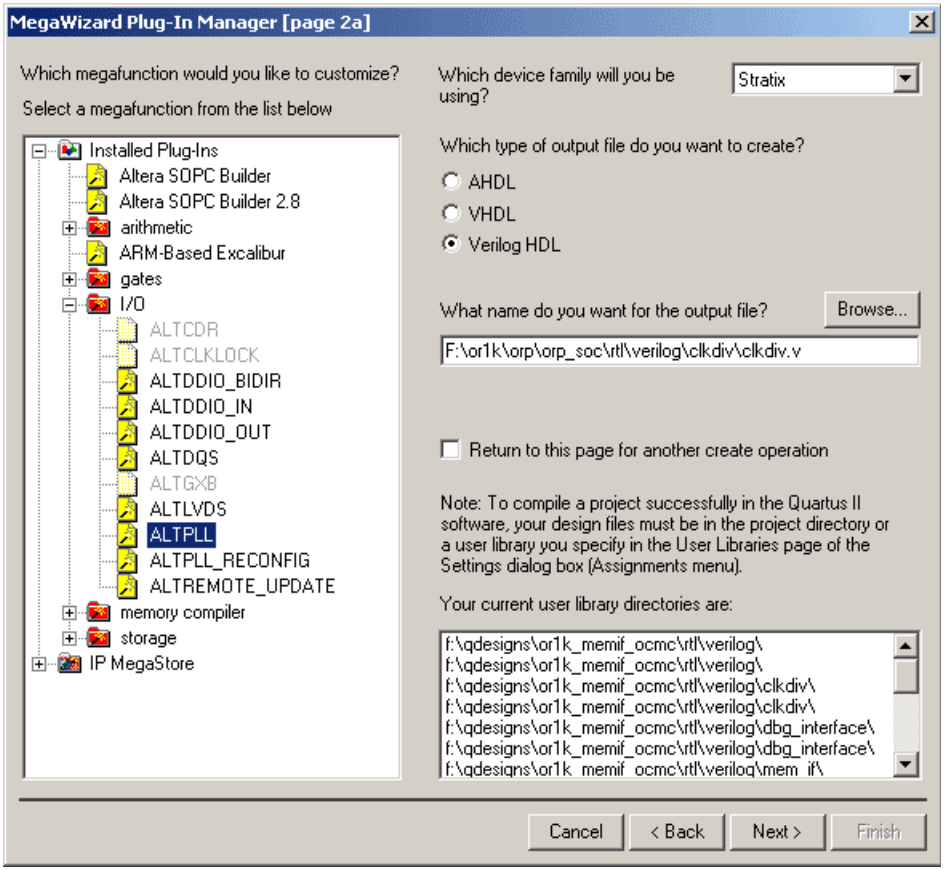
To add a PLL component, it first has to be generated using Altera Megawizard. So open Quartus II. Go to the menu item 'Tools' and then 'Megawizard plug-in wizard'.

One the first page of the wizard click "Create a new custom megafunction variation" and press "Next".

On the second page:

- select the correct FPGA family you want to use
- as output file, select 'Verilog HDL'
- browse to the location where you want to place the clockdivider and type in a name like in the example of figure 1.
- The only thing needed now is to select the megafunction you want to create: in the left screen select 'I/O' -> 'ALTPLL'
- You can now click 'Next'

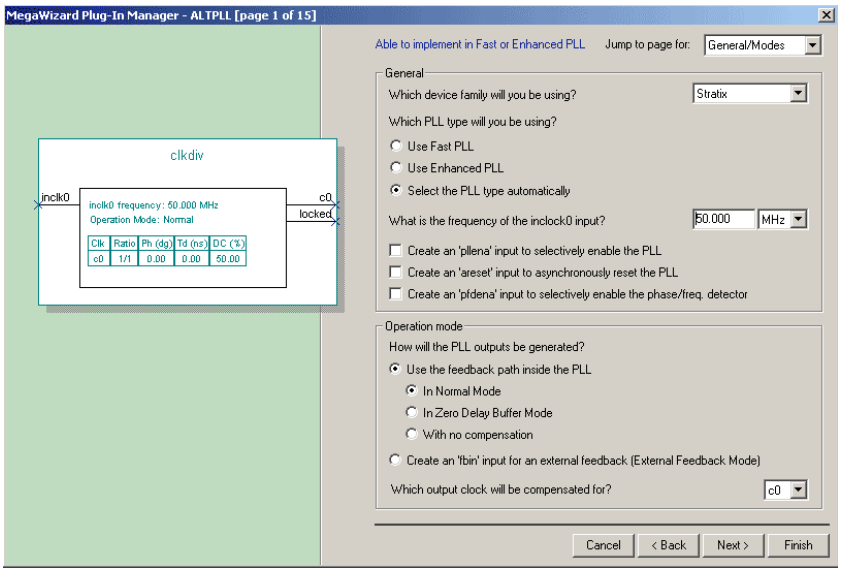
Figure 2: MegaWizard page 2



On the first page of the ALTPLL wizard:

- Check whether the correct FPGA family is selected.
- Correct the input frequency
- You can disable ‘areset’ and ‘pfdena’
- You can then click ‘Next’

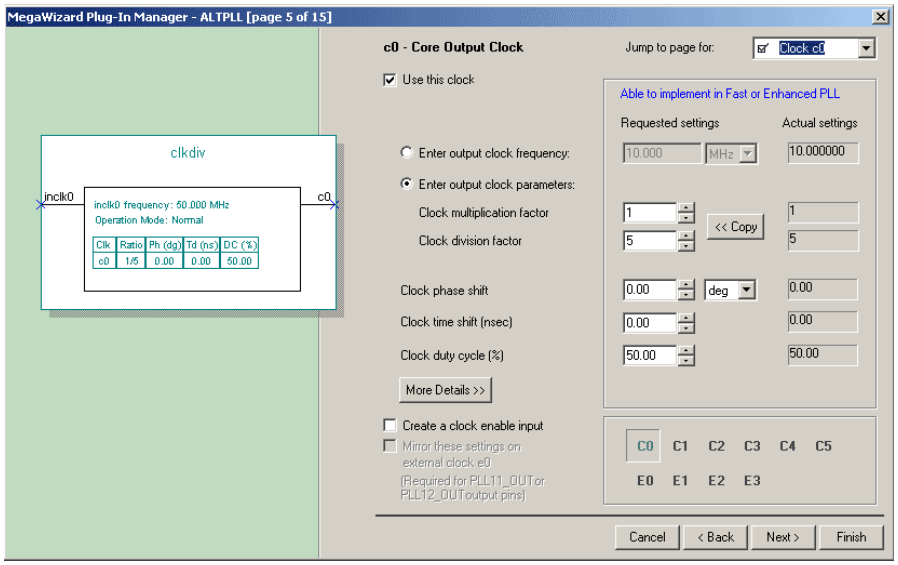
Figure 3: First ALTPLL page



On the second page, you can disable the ‘locked’ output option. Now you can proceed directly to page 5 (clock c0).

On page 5, you can either select an output frequency or a multiplication/division factor. After creating the correct clock frequency, you can press ‘Finish’.

Figure 4: Create a Quartus II project



Because this PLL component is ready-as-it-is to be included, it now only has to be included in the top-file of the Quartus project in a later stadium.

2) onchip RAM component

Now comes an important step: writing the onchip RAM component. A completely new component has to be written. The module consists of:

- a ‘module’ part
- a ‘parameter’ part
- an ‘input & output’ part
- a ‘wire’ part
- an ‘assignment’ part
- ‘read and write acknowledge processes’ part
- ‘altsyncram instantiation’ part

When taking in account timing and protocol of Wishbone bus and synchronous onchip RAM, the onchip RAM module eventually must look like (16kB or 131072 bits onchip RAM version):

```
module onchip_ram_top (
  wb_clk_i, wb_rst_i,

  wb_dat_i, wb_dat_o, wb_adr_i, wb_sel_i, wb_we_i, wb_cyc_i,
  wb_stb_i, wb_ack_o, wb_err_o
);

//
// Parameters
//
parameter           aw = 12;

//
// I/O Ports
//
input               wb_clk_i;
input               wb_rst_i;

//
// WB slave i/f
//
input  [31:0]       wb_dat_i;
output [31:0]       wb_dat_o;
input  [31:0]       wb_adr_i;
input  [3:0]        wb_sel_i;
input               wb_we_i;
input               wb_cyc_i;
input               wb_stb_i;
output              wb_ack_o;
output              wb_err_o;

//
// Internal regs and wires
//
wire          we;
wire [3:0]    be_i;
wire [aw-1:0]adr;
wire [31:0]   wb_dat_o;
reg           ack_we;
reg           ack_re;
```

```
//
// Aliases and simple assignments
//
assign wb_ack_o = ack_re | ack_we;
assign wb_err_o = wb_cyc_i & wb_stb_i & (|wb_adr_i[23:aw+2]);     // If Access to > (8-bit
leading prefix ignored)
assign we = wb_cyc_i & wb_stb_i & wb_we_i & (|wb_sel_i[3:0]);
assign be_i = (wb_cyc_i & wb_stb_i) * wb_sel_i;

//
// Write acknowledge
//
always @ (negedge wb_clk_i or posedge wb_rst_i)
begin
if (wb_rst_i)
    ack_we <= 1'b0;
  else
  if (wb_cyc_i & wb_stb_i & wb_we_i & ~ack_we)
    ack_we <= #1 1'b1;
  else
    ack_we <= #1 1'b0;
end

//
// read acknowledge
//
always @ (posedge wb_clk_i or posedge wb_rst_i)
begin
  if (wb_rst_i)
    ack_re <= 1'b0;
  else
  if (wb_cyc_i & wb_stb_i & ~wb_err_o & ~wb_we_i & ~ack_re)
    ack_re <= #1 1'b1;
  else
    ack_re <= #1 1'b0;
end

//
// change intended_device_family according to the FPGA device (Stratix or Cyclone)
//
       altsyncram     altsyncram_component (
                              .wren_a (we),
                              .clock0 (wb_clk_i),
                              .byteena_a (be_i),
                              .address_a (wb_adr_i[aw+1:2]),
                              .data_a (wb_dat_i),
                              .q_a (wb_dat_o));
       defparam
              altsyncram_component.intended_device_family = "Stratix",
              altsyncram_component.width_a = 32,
              altsyncram_component.widthad_a = 12,
              altsyncram_component.numwords_a = 4096,
              altsyncram_component.operation_mode = "SINGLE_PORT",
              altsyncram_component.outdata_reg_a = "UNREGISTERED",
              altsyncram_component.indata_aclr_a = "NONE",
              altsyncram_component.wrcontrol_aclr_a = "NONE",
              altsyncram_component.address_aclr_a = "NONE",
              altsyncram_component.outdata_aclr_a = "NONE",
              altsyncram_component.width_byteena_a = 4,
              altsyncram_component.byte_size = 8,
              altsyncram_component.byteena_aclr_a = "NONE",
              altsyncram_component.ram_block_type = "AUTO",
              altsyncram_component.lpm_type = "altsyncram";

endmodule
```

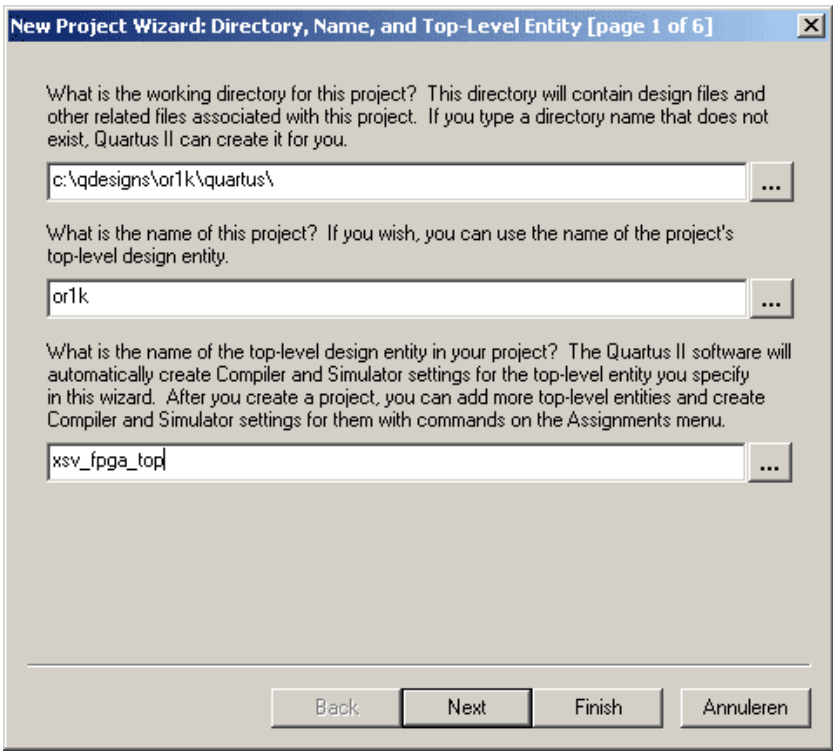
It's best to add this file to the source directory so that it can be easily added to the Quartus project.

V Synthesis, place & route, generating the bitstream

1) Start Quartus II (*Start → programs → Altera → Quartus II 3.0*)

2) Create a new quartus project (*File → new project wizard*) (figure 1)

Figure 5: Create a Quartus II project



3) Select a directory as a working directory. It's best to put it in a seperate directory than the source code. Then choose a projectname. The top level entity is 'xsv_fpga_top' if you haven't changed this name in the 'xsv_fpga_top.v' file. You can now click 'next'

4) Now you have to select the source files. When using Quartus II 3.0, the order in which the files are added isn't important. Just add all the 'debug-unit'-, 'openrisc'- and 'uart'-files (except for timescale.v). Also add of course the topfile of the PLL component and the onchip RAM. In the root of the source-files, add the 'xsv_fpga_top.v', 'xsv_fpga_defines.v' and 'tc_top.v'. After adding all these files, **don't** press 'next'.

5) You now have to **include the library pathnames**. Therefor click "User Library Pathnames...". Now add all the directories where the source files reside. Press 'OK' and 'Next' to proceed.

6) Select 'none' as design entry synthesis tool and click 'next'.

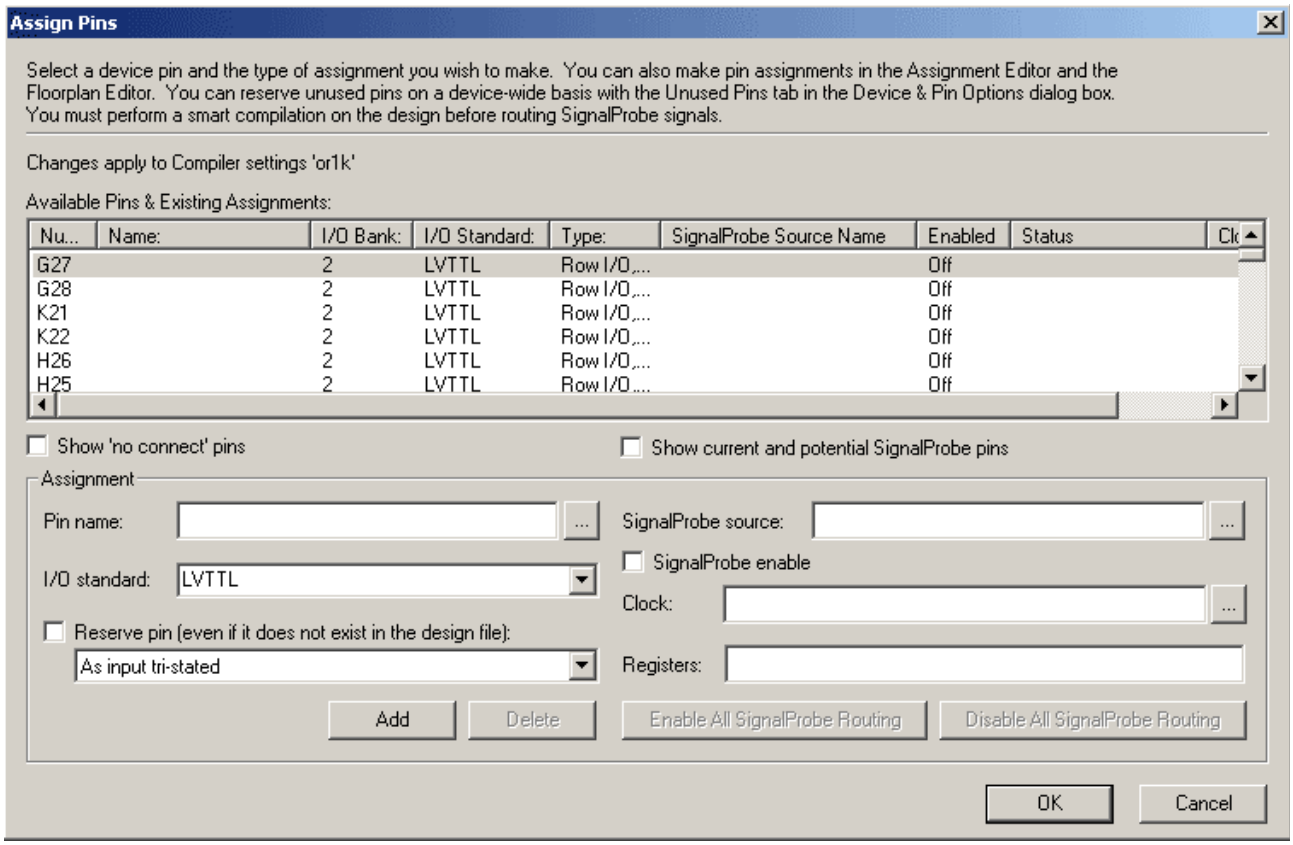
7) Select your component's target family, and select 'yes' to select the component itself directly. Click on 'next'

8) Select your target component, click on 'next'

9) Press on ‘finish’ to complete the project setup.

10) Now you have to assign the pins of the FPGA to the top-file ports. There are two ways to do this:
    a) *Graphical:* In the menu ‘assignments’, click “assign pins”. You will then get a list with all the physical FPGA pins. To assign a pin to a signal, select the physical pin and type the signalname in the “Pin name” box. To add the pin, press ‘Add’. Continue this process until all pins have been assigned.

Figure 6: graphically assigning pins



---

    b) *Text:* Close the Quartus project by clicking “close project” in the ‘File’ menu. With a text editor open the “<projectname>.csf” file in the project directory. Now to assign pins, add the following line in the “CHIP(projectname)” part: “<signal-name> : LOCATION = Pin_<location>;” After assigning all signals, you can save and go back to the Quartus project.

11) Start the compilation “processing → start compilation” to synthesise and place & route the design and create the bitstream.

12) In the “message”-window, look for the message indicating the achieved clock speed. It should be higher then the divided clock. Otherwise, adjust the division factor of the PLL component and re-compile.

VI Download and test the OpenRISC

1) Open the Quartus programmer (*Tools → Programmer*)
   a) Make sure you connected your download cable correctly (e.g. byteblaster,…), make sure your board is powered on

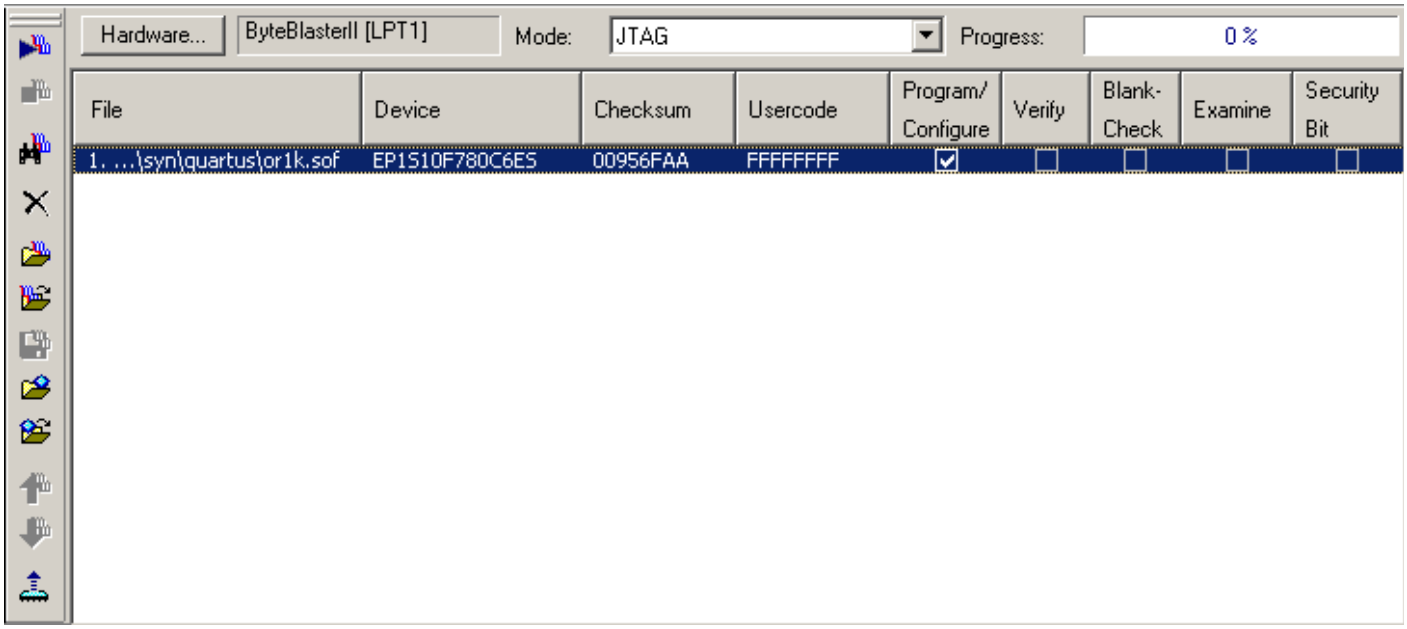
   b) If you haven't configured the Quartus programmer yet, push the 'Hardware...' button to configure your hardware setup. First "Add Hardware" and select the correct download cable. Then select the hardware in the list and press "Select Hardware".

   c) After the hardware configuration, push on the "auto detect" button (binocular icon). This will initialise the device-chain and display it.

   d) Remove the device that has to be programmed, and add the generated target file "<projectname>.sof" to the chain by pushing on the button "add file" (upper map icon). After the file has been added, make sure the order of the different devices is still correct.

   e) Make sure to mark the program/configure box of your device in the "Program/configure"-column (figure 7)

Figure 7: Quartus programmer



   f) Press on the "start" button (play icon) to download your design into the device.

Congratulations, you have now succesfully generated an OpenRISC based embedded system and downloaded it to an FPGA. To test your system by running software on the processor, follow the instructions of the software tutorial that you can find on our website (http://emsys.denayer.wenk.be).