© 2025 - ALSE
Author : Bertrand CUZEAU
Date : Feb 2025
Version : 2025.02d

Advanced Logic Synthesis for Electronics
A.L.S.E. - https://alse-fr.com

- ALSE -

Installing and Discovering Lattice Radiant

To contact ALSE, visit the
ALSE website (FPGA.fr)
or write to : info@alse-fr.com

Installing and Discovering Lattice Radiant

In this document, you will see how to quickly obtain, install and license Radiant, the Lattice FPGA Design tool. The steps are detailed for Windows, but Linux is perfect too for installing and using Radiant.

The Introduction to Radiant that follows the installation will help you create a first Lattice FPGA project and walk you through the main steps of the design flow.

If you plan to use Radiant after this introduction, we strongly recommend that you follow this free and very instructive 2+ hours Lattice course.

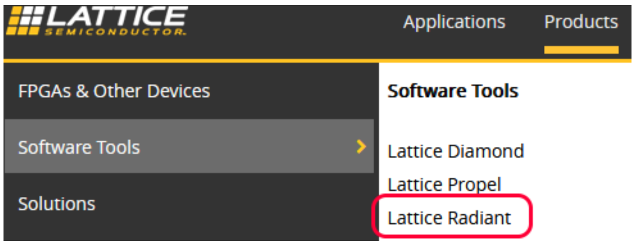
You can also find some short videos detailing design flow steps.

If you are relatively new to FPGA design, be aware that there is a lot of theory even behind a trivial example like this one. If you want to understand and master all you need, we have the right training courses for you.

Installing Radiant (2024.2)

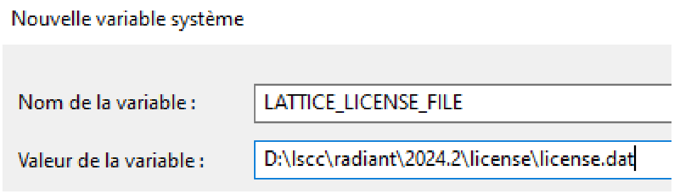
Here are the instructions to install Radiant (version 2024.2).

- Go to the Lattice web site.
- If you've not done so already :
  Register and **create an account**.
- Sign In.
- Products > Software Tools > Lattice Radiant
- In « Getting Started », **download** the Windows version (or the Linux version if it's your OS).



- You must accept the License Agreement.
  You will get a 2.9 GB zip file in your usual « download » directory.
- Extract the zip file at the same location : you will obtain an executable file for example `2024.2.0.3.4_Radiant.exe`.
  You can now delete the zip file.
- Launch the executable file.
- Select where you want to install Radiant (avoid too long path names and spaces in the name).
- Select the FPGA families you want to install.
  For the Free version of Radiant and for the purpose of this introduction, you can remove (uncheck) MachXO5 and Avant.
- Accept the License and launch the installation.
  Be patient : this will take some time...
- During or after the installation, you can sign in again if necessary, and request a Free License.
  Go to the Lattice Radiant Software section, and select « Request Node-Locked License ».
- Open a command shell and enter the command : `ipconfig /all` (or `ip addr` under Linux).
  Note the Physical Address of a fixed (and if possible active) network adapter.
  Enter this 12 characters address **without space nor ":" nor « - »** in the "Host NIC" box.
- Select all the IPs, agree with the License Agreement, and click *one time* on « Generate License ».
  After usually a few seconds, you should receive in your mailbox a `license.dat` file.
  Verify that this file contains a feature named "`LSC_RADIANT`" ! If not, try to request a new license.
  Copy it under the « license » folder of Radiant install, for example in `x:\lscc\radiant\2024.2\license`.

- For Windows : create a **`LATTICE_LICENSE_FILE`** System Variable pointing to the license :

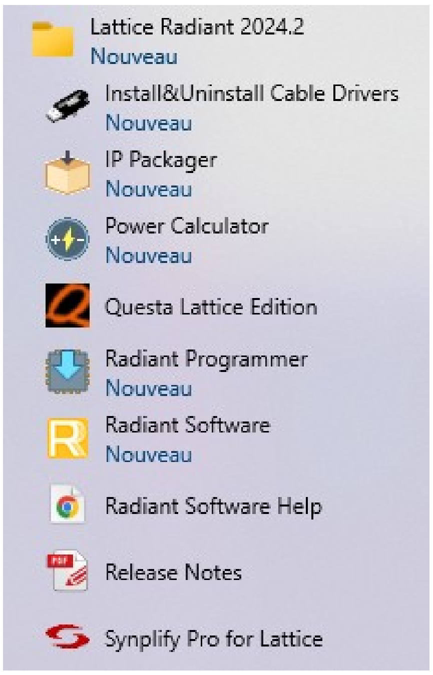

Nouvelle variable système

Nom de la variable : LATTICE_LICENSE_FILE

Valeur de la variable : D:\lscc\radiant\2024.2\license\license.dat

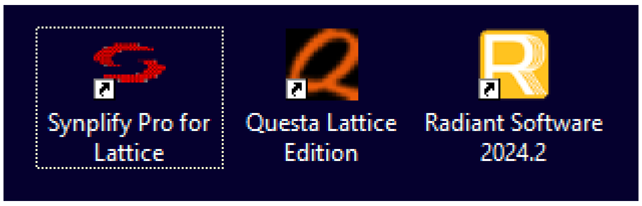
  (the actual path may be different for your installation)

  This above step is not necessary under Linux.
- Windows : Launch Radiant with its new icon on your Windows desktop.
  If it takes more than 30 seconds to launch, contact ALSE.



Extra steps **for Windows**

- In Windows Start Menu, drag and drop **Questa Lattice Edition** icon to your desktop close to the Radiant icon.
- Edit the icon property and modify « Start In » to point to the location of your tests (eg c:\Radiant_tests).
  Avoid too long path names and spaces in the path name.
- Do the same for **Synplify Pro for Lattice** (including "Start in").



With the Synplify Pro you can very quickly Synthesize any HDL code in a Lattice FPGA target.

With the QuestaSim icons, you can Simulate any HDL code.

With Radiant, you can implement the complete design flow and try your design on a Lattice FPGA kit or board.

Extra steps for **Linux**

- Make sure the **/bin** directory of the Radiant install is in your path,
  or create a "radiant" alias to launch Radiant from the bin directory.
- Open a command shell, and change to a directory that will hold your Radiant project like for example ~/Radiant_tests.
- Launch **`radiant`**. And verify that it does open correctly and reasonably quickly.

  NOTE ! The first time you launch Radiant under Linux, il will open the **License debugger** in which you will be able to specify where your license file is.

  Another way to help tools locate the license is to run this command :
  `export LM_LICENSE_FILE=/path/to/license.dat`
  before launching the tool.

It's a good idea to create simple ways to launch "**Synplify Pro for Lattice**" and "**QuestaSim Lattice Edition**" just as you did for Radiant.

Introduction to Radiant

Let's take a very simple design (it's the first exercise in our VHDL basic training).

To make this introduction more general, you will find in Appendix A both VHDL and SystemVerilog versions.

It comes with a very simple entity to be synthesized (logic.vhd or logic.sv), and a very simple testbench to simulate it (logic_tb.vhd or logic_tb.sv). Both files a reproduced in Appendix A so you don't need any archive.

In the following steps, we are going to see how to create a Lattice Radiant project with these two files, how to compile an FPGA, and how to simulate with QuestaSim Lattice Edition before and after Synthesis.

Create the three design files

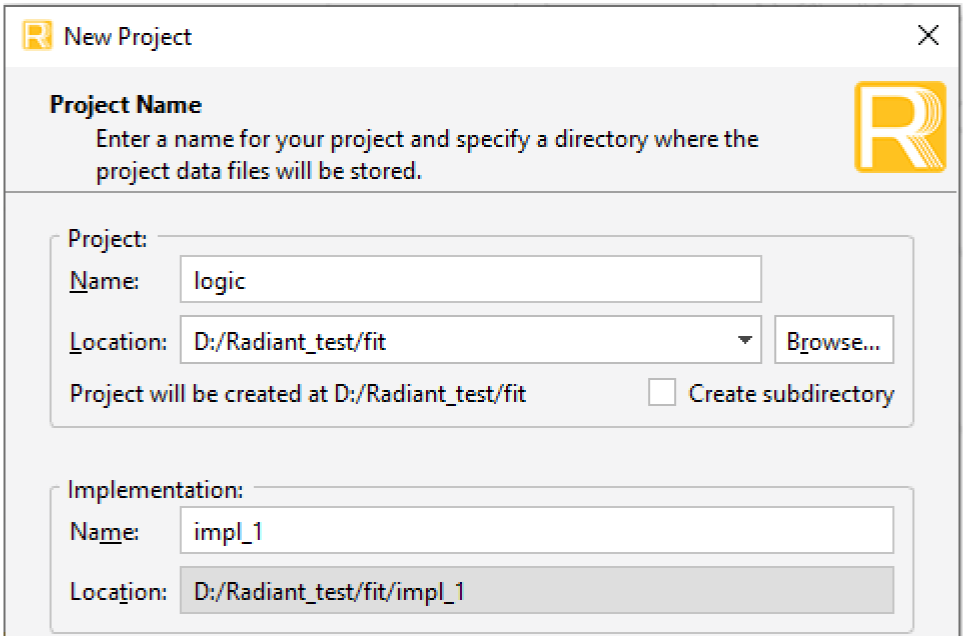
Make sure you create a directory in a simple path to host this test design, as suggested in the instalaltion. For example : `c:\Radiant_test` or `~/Radiant_test`.

- Use a text editor to create `logic.vhd` (or .sv) and copy-paste the contents from Appendix A.
- Use a text editor to create `logic_tb.vhd` (or .sv) and copy-paste the contents from Appendix A.
- Use a text editor to create `logic.sdc` and copy-paste this constraint below :

```
set_max_delay -from [get_ports IN*] -to [get_ports OUT*] 6
```

Create a new Radiant project

- Launch Radiant and wait until you have the main panel showing up and telling you that you have the latest version.
- Click on « New Project »
  Name : **logic**
  Location : xxxxx/**fit** (xxxxx is the directory you created in the previous step)
  uncheck "Create subdirectory".
  Implementation name = impl_1 (default)
  Implementation location= xxxxx**/fit/impl_1**


New Project

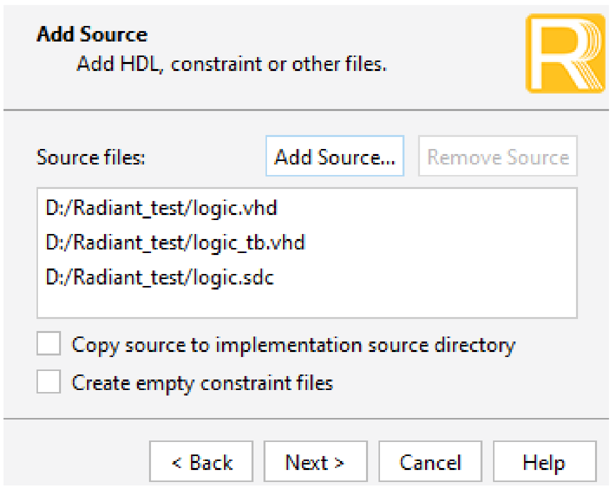
Project Name
Enter a name for your project and specify a directory where the project data files will be stored.

Project:
Name: logic
Location: D:/Radiant_test/fit
Project will be created at D:/Radiant_test/fit
☐ Create subdirectory

Implementation:
Name: impl_1
Location: D:/Radiant_test/fit/impl_1

- On the next panel, click on « Add source... », navigate up then into the source folder.
- Select the three files : `logic.vhd`, `logic.sdc`, and `logic_tb.vhd`.
  /!\ You may have to change the Input Files (...) box to see the sdc file.

- Do <u>**not**</u> « Copy source to implementation source directory »,
  Do <u>**not**</u> « Create empty constraint files »
  Next


Add Source
Add HDL, constraint or other files.

Source files:

D:/Radiant_test/logic.vhd
D:/Radiant_test/logic_tb.vhd
D:/Radiant_test/logic.sdc

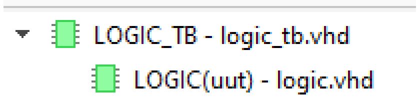
☐ Copy source to implementation source directory
☐ Create empty constraint files

- Select LFD2NX (Certus Nx) and the smallest LFD2NX-9, with other settings being default choices.
  The Part Number should be LFD2NX-9-9MG121C
  (in fact, it doesn't really matter for this lab).
- Next : Synthesis Tool = **Lattice LSE**
- On the final review page, click on « **Finish** ».

The project is created !

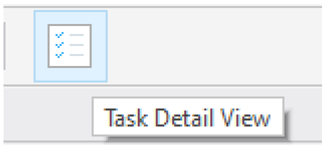
Project settings

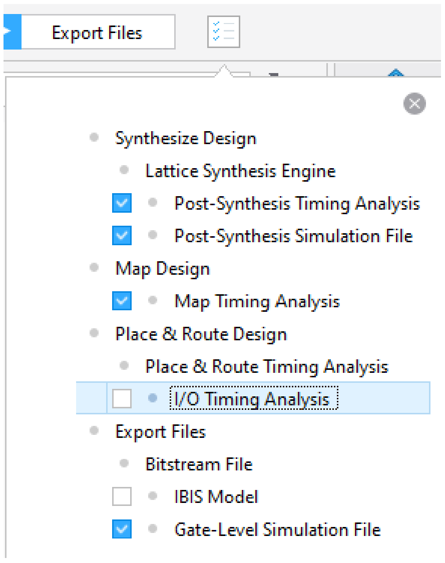
The left panel must be in « File List » mode (bottom tab).
You see that the top level is the testbench when it should be « LOGIC », and logic_tb.vhd is in bold characters.


LOGIC_TB - logic_tb.vhd
LOGIC(uut) - logic.vhd

This is not correct !
The reason is that we have added both the design file and the testbench *for design* and for simulation.
Let's fix this :

- In the Input files list, right click on **`logic_tb.vhd`** then select **Include for** > **Simulation**.
  Now, the top level is LOGIC and the active file is ../logic.vhd.
- For a real project we should have more complete timing constraints and also physical constraints like I/O pins assignments. For this lab, we have no board and the design has no clock…
- At this stage we can prepare for some simulations by generating the proper models.
  Open the **Task Detail View** and activate the generation of the simulation files :





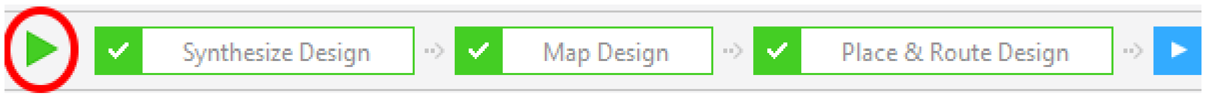
- Synthesize Design
  - Lattice Synthesis Engine
  - ☑ Post-Synthesis Timing Analysis
  - ☑ Post-Synthesis Simulation File
- Map Design
  - ☑ Map Timing Analysis
- Place & Route Design
  - Place & Route Timing Analysis
  - ☐ I/O Timing Analysis
- Export Files
  - Bitstream File
  - ☐ IBIS Model
  - ☑ Gate-Level Simulation File

Note that we also ask for various timing reports too.
In our constraints, we expressed that we wanted a maximum delay of 6 ns between our inputs and our outputs. We will see if this has been achieved.

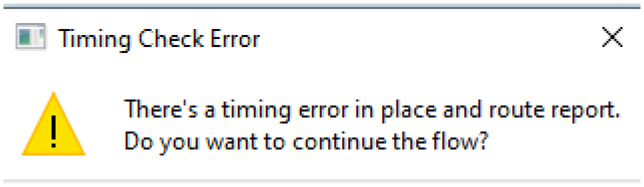
Project compilation

Now we can perform a full compilation of the project :

- ✔ Double click on the **Gate Level Simulation File** item in the Task View
- ✔ Or simply **click on the Arrow** on the left :



This process takes a bit of time and should end with a success and the green check marks above.
You should get a warning, you can safely **ignore** it at this point and continue the compilation.
We will understand the issue in the next step.



Timing Check Error

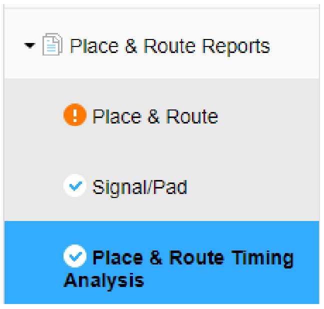
There's a timing error in place and route report.
Do you want to continue the flow?

Timings Analysis

Let's see if our 6 ns constraint has been met.

➢ Open the Place & Route Timing Analysis report.

➢ Scroll down :

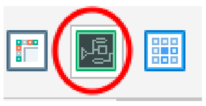


```
2.2  Endpoint slacks
-------------------------------------------------------
          Listing 4 End Points            |    Slack
-------------------------------------------------------
OUT1                                      |   -0.961 ns
OUT0                                      |   -0.941 ns
OUT2                                      |   -0.853 ns
OUT3                                      |   -0.821 ns
-------------------------------------------------------
                                          |
Setup # of endpoints with negative slack:|           4
                                          |
-------------------------------------------------------
```

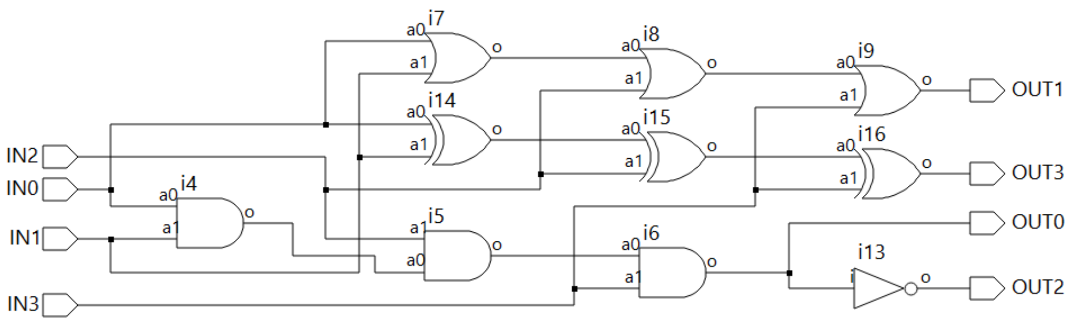
If you scroll further down, you'll see the details of the timing path, leading to a total delay of ~6.96 ns which is missing our objective by the negative slack of 0.961 ns.

Inspecting the Implementation results

Check the **Project Summary** (in the Reports Tab).
You should see that 4 Look Up Tables (4 inputs) and 8 I/Os were used.



Open the Netlist Analyzer.
Here is what you should see, this is the **RTL View** :



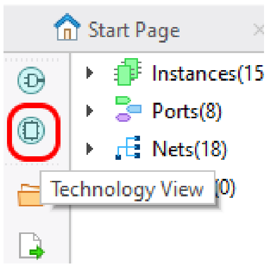
This is the translation of the boolean equations of VHDL code. We see 10 logic gates.

Keep in mind that, in general (for non-trivial design) the **RTL view is not providing useful information** at all about how and how efficiently you code was implemented. This is not better in Vivado or Quartus !
Never try to optimize your code based on what you see in the RTL View. This is a very common misconception.

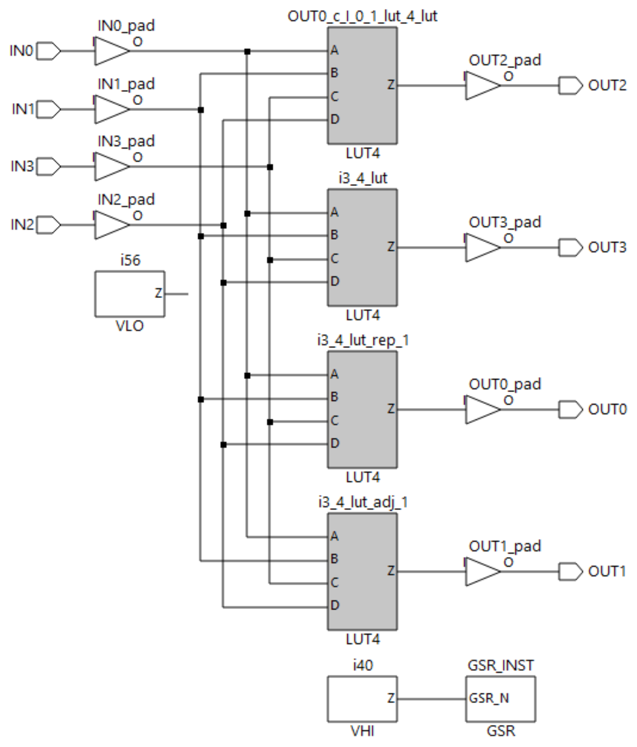
Let's see now how this logic has actually been implemented in the FPGA !

Will we find 10 gates ?

Switch to the **Technology View** by clicking on the button in the left tool bar.



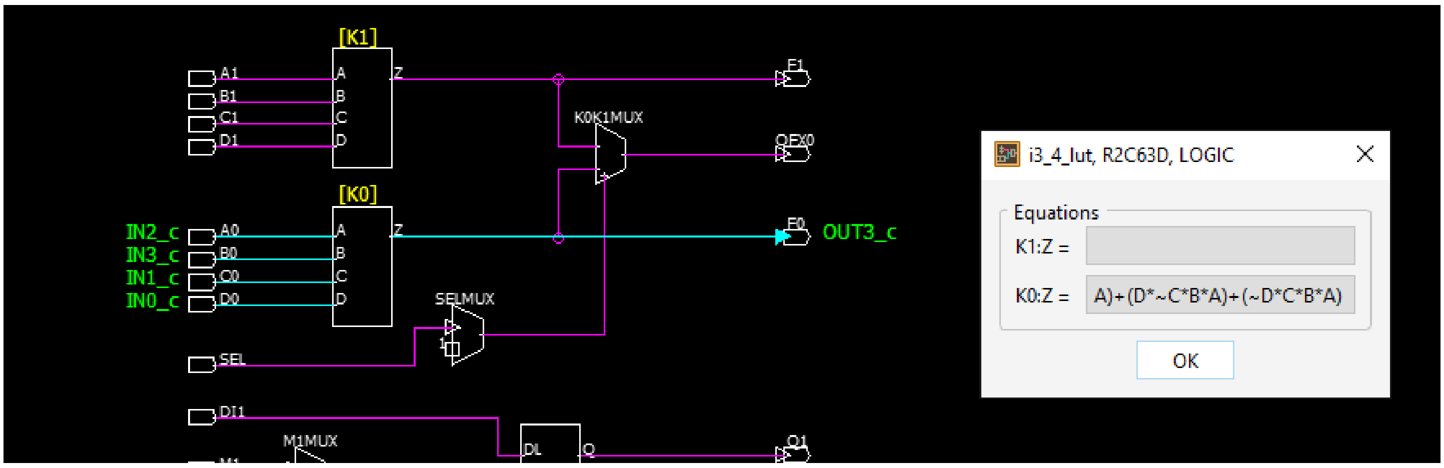
He is what you'll see :



We see the 4 x 4-inputs LUTs that were inferred.

We also see that the Global Set Reset (GSR) block is present but not used.

If you have time, you can open **Physical Designer**, select one of the LUTs and display the **logic block view**.



i3_4_lut, R2C63D, LOGIC

Equations

K1:Z =

K0:Z = A)+(D*~C*B*A)+(~D*C*B*A)

OK

We see the slice inside which only one LUT4 was used [K0], with the equation (partially) displayed.

You can also take a look at the routing (interconnects).

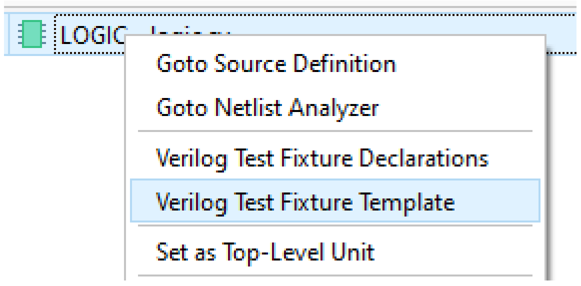
This is a very trivial design, so there isn't much more to see.

Verification by Simulation

Simulation is an essential part of the design flow and it will typically consume a lot of the total design time.
In our training courses, we show -even just rapidly in some courses- how to use Modelsim or Questasim efficiently. The steps below are just showing the integration of QuestaSim in Radiant.

Testbench creation

Radiant has a nice feature : it can create automatically a testbench skeleton.
Many other tools can do this, like EMACS (VHDL mode), or Teros-HDL to name a few HDL Editors.
You just have to right click on an RTL file in the file list and select "Test Fixture Template".

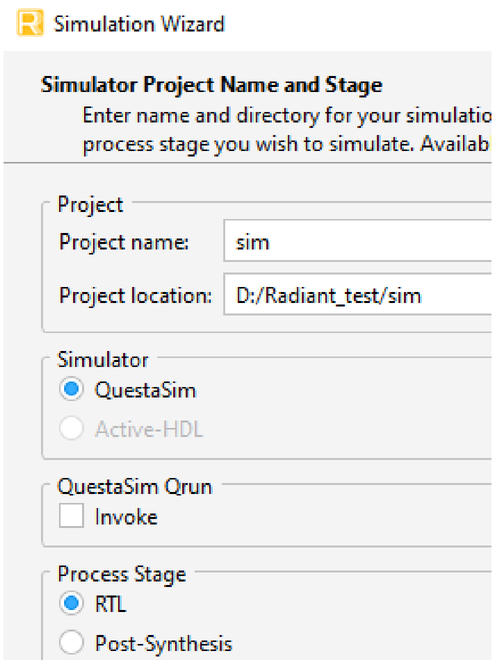


It creates a VHDL test bench for a VHDL entity, and a Verilog test fixture for a (System)Verilog module.

RTL Simulation

In this simple project, RTL- (or "source code-") simulation does simulate the generic code, so the simulation is therefore agnostic of the type of FPGA and makes no use of the vendors' simulation libraries.

- Launch the **Simulation Wizard**. Then Next.
- Project Name = **sim**
  Location = xxxxx/**sim**
  Do **not** invoke Qrun
  Process stage = **RTL**
  Confirm the creation of the directory.
- You should see the two files with the testbench second in the list, which is the correct order, so you could un-check "Automatically..."
  Next.
- The simulation Top module should be LOGIC_TB.
  Next
- Run Simulation Default Run = **0** (not the default),. The simulation will stop when the test bench is completed.
  Keep the other settings.
- Finish


Simulation Wizard

Simulator Project Name and Stage
Enter name and directory for your simulatio
process stage you wish to simulate. Availab

Project
Project name: sim
Project location: D:/Radiant_test/sim

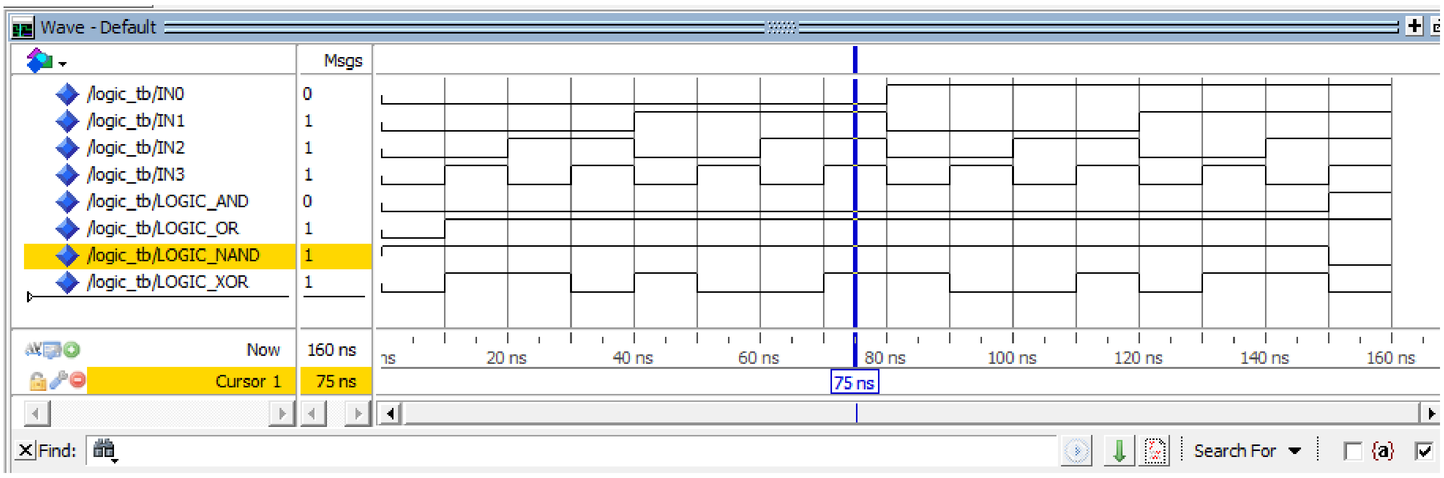
Simulator
[✓ QuestaSim]
Active-HDL

QuestaSim Qrun
☐ Invoke

Process Stage
[✓ RTL]
Post-Synthesis

Questasim must launch, compile, simulate and display the waveform below, automatically.

- If needed, perform a **zoom full** to visualize the entire simulation (mouse stroke to upper left).
  The waveform is reproduced next page.



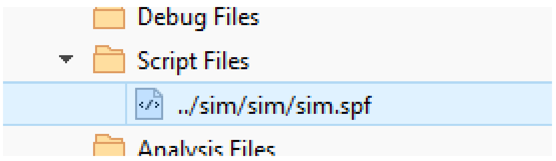
As we can see, there is zero delay between the input change and the output change.

Keep in mind that we simulate the HDL source code.

➢ Close QuestaSim and confirm.

Post-map simulation

It is also possible and sometimes valuable, to simulate the design after its synthesis and transformation ("mapping") into FPGA primitives. This is what "post-map" simulation is about..



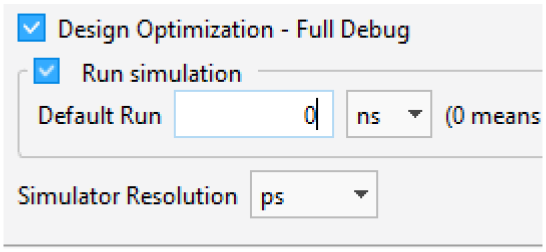
➢ Double click on the **sim.spf** file in the Script Files list
This reloads the Simulation wizard.

➢ Select "**Post-Synthesis**".

➢ Simulation Top Module = **logic_tb** (vhdl) or LOGIC_TB (sv)
Associated Instance = **/uut** (vhdl) or /dut (sv)

Note that if the VHDL test bench uses direct instantiation or some advanced syntax, the wizard's parser **will have errors** which you can override by entering manually the values above.
If you use a VHDL testbench that uses the component declaration, or if you use the SystemVerilog test bench, you won't have to enter this manually.

➢ Important ! Select **Simulator resolution = ps** You also need to enter "0" as "Default Run" duration.



☑ Design Optimization - Full Debug

☑ Run simulation

Default Run: 0 ns (0 means

Simulator Resolution: ps

The simulation should run and display exactly the same results as for the RTL simulation.

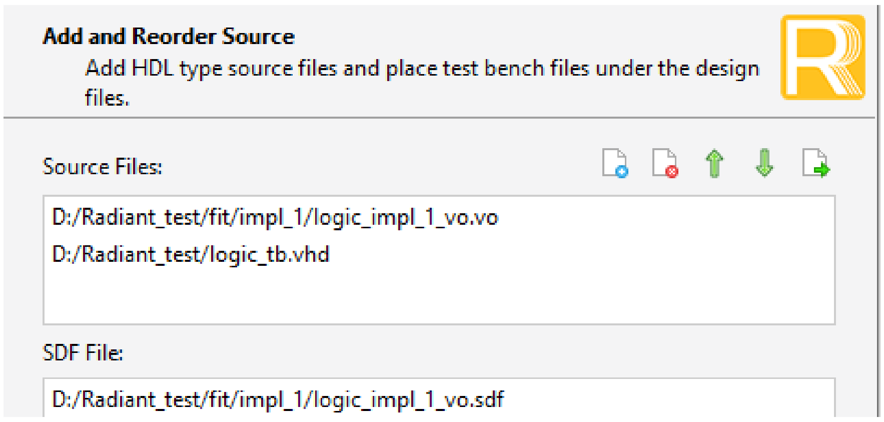
Post-Layout timing simulation

This type of simulation is rarely used for many reasons including : very slow, not useful (it does show delays which are never going to be the actual delays), and it is actually impossible to perform on some modern FPGA families from other vendors.

Important : keep in mind that **Timings are not verified by timing simulation !**
Timings are verified by Static Timing Analysis (STA).

But we can run post-layout timing simulation with Radiant if we want !

- Proceed as for Post-map simulation, but :
  Select "**Post-Route Gate-Level+Timing**"
- Verify that the proper sdf and Verilog output (vo) files appear in the correct order :



Add and Reorder Source
Add HDL type source files and place test bench files under the design files.

Source Files:

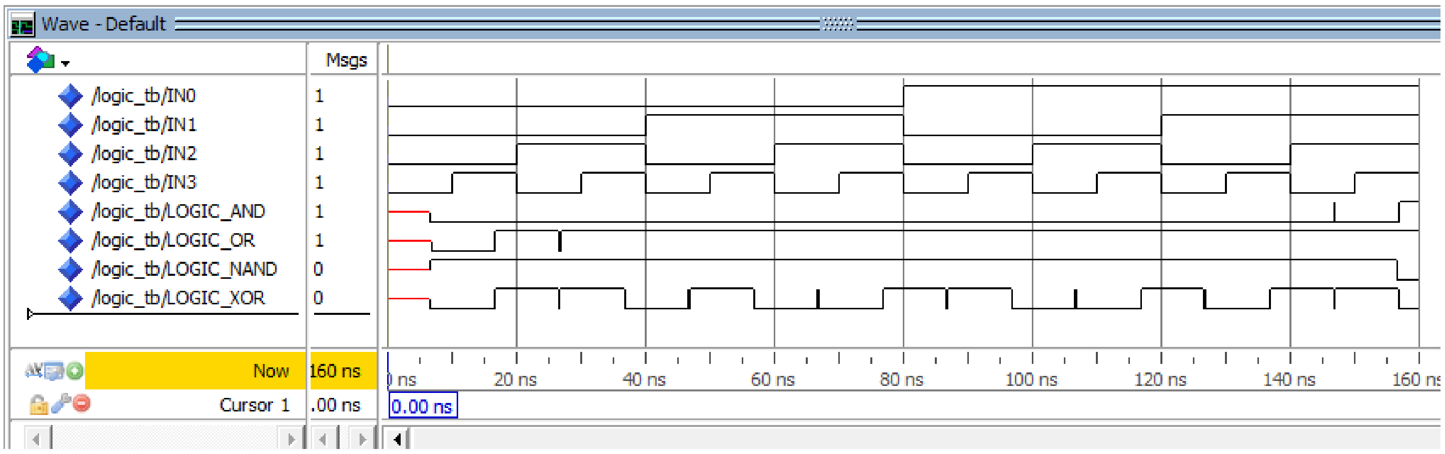
D:/Radiant_test/fit/impl_1/logic_impl_1_vo.vo
D:/Radiant_test/logic_tb.vhd

SDF File:

D:/Radiant_test/fit/impl_1/logic_impl_1_vo.sdf

- For this type of simulation, you need to verify that the « Associated instance » is the device under test name (eg /dut for SV or /UUT for the VHDL version).
- Do not forget to select **Simulator resolution = ps** You also need to enter "0" as "Default Run" duration.
- Run the simulation, then zoom-Full.
- You may see in the transcript some width violations : this is normal ! Combinatorial logic does generate glitches...

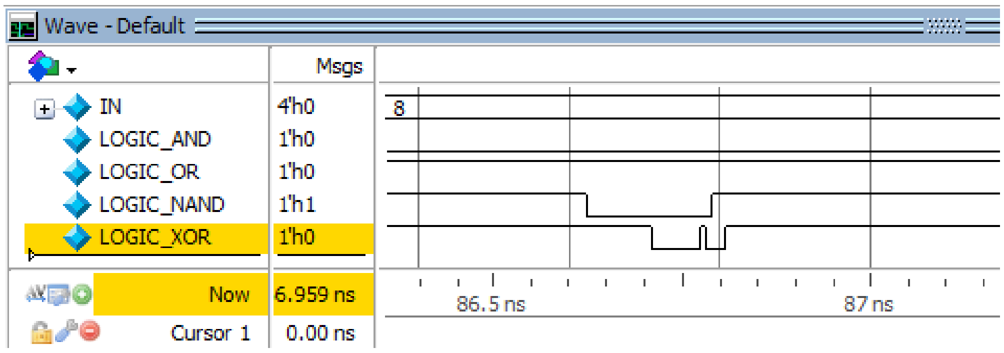
Here is the simulation waveform :



This time, the simulation show delays and glitches !

The delays shown are between input pins and output pins and we can measure around 7 ns which is consistent with the Timing Analysis report we have obtained after P&R.

And if you zoom in on glitches, you'll see details like :



THIS CONCLUDES THE INTRODUCTION TO RADIANT !

Appendix A
Design files

As explained, you can copy and paste from this document to create the two files you need to run the introduction to Radiant, either in VHDL or in SystemVerilog.

The first file (logic.vhd) is a very simple **VHDL** design involving 4 simple boolean equations.

```
-- logic.vhd
-- Very Simple design file
--
Library IEEE;
    use IEEE.std_logic_1164.all;

Entity LOGIC is
  port (
    IN0 : in  std_logic;
    IN1 : in  std_logic;
    IN2 : in  std_logic;
    IN3 : in  std_logic;
    OUT0: out std_logic;
    OUT1: out std_logic;
    OUT2: out std_logic;
    OUT3: out std_logic
  );
end entity LOGIC;

Architecture RTL of LOGIC is
Begin
  OUT0 <=      IN0 and IN1 and IN2 and IN3;
  OUT1 <=      IN0 or  IN1 or  IN2 or  IN3;
  OUT2 <= not (IN0 and IN1 and IN2 and IN3);
  OUT3 <=      IN0 xor IN1 xor IN2 xor IN3;
end architecture RTL;
```

The **SystemVerilog** version (logic.sv) is :

```
// logic.sv
// Very Simple design file (SystemVerilog)

module LOGIC (
    input  logic  IN0,  IN1,  IN2,  IN3,
    output logic OUT0, OUT1, OUT2, OUT3  );

assign OUT0 =   IN0 & IN1 & IN2 & IN3;
assign OUT1 =   IN0 | IN1 | IN2 | IN3;
assign OUT2 = ! OUT0;
assign OUT3 =   IN0 ^ IN1 ^ IN2 ^ IN3;

endmodule : LOGIC
```

Note: the name of the module "LOGIC" must not be in lower case to avoid a conflict with the "logic" type in SV. SystemVerilog is case-sensitive.

The second file is the **Timing Constraints** File (logic.sdc) :

```
# logic.sdc
# Constraint file for the all combinatorial design
set_max_delay -from [get_ports IN*] -to [get_ports OUT*] 6
```

The third file is a **simple test bench** to exercise the four inputs. The result must be observed in the simulation waveform (the test bench makes no effort to verify the outputs).

```
-- logic_tb.vhd
-- Very Simple Test Bench

Library IEEE;
    use IEEE.std_logic_1164.all;

Entity LOGIC_TB is end;

Architecture TEST of LOGIC_TB is
  signal IN0,IN1,IN2,IN3 : std_logic;
  signal LOGIC_AND : std_logic;
  signal LOGIC_OR  : std_logic;
  signal LOGIC_NAND: std_logic;
  signal LOGIC_XOR : std_logic;
Begin
  process  begin
    IN0<='0'; IN1<='0'; IN2<='0'; IN3<='0'; wait for 10 ns;
    IN0<='0'; IN1<='0'; IN2<='0'; IN3<='1'; wait for 10 ns;
    IN0<='0'; IN1<='0'; IN2<='1'; IN3<='0'; wait for 10 ns;
    IN0<='0'; IN1<='0'; IN2<='1'; IN3<='1'; wait for 10 ns;
    IN0<='0'; IN1<='1'; IN2<='0'; IN3<='0'; wait for 10 ns;
    IN0<='0'; IN1<='1'; IN2<='0'; IN3<='1'; wait for 10 ns;
    IN0<='0'; IN1<='1'; IN2<='1'; IN3<='0'; wait for 10 ns;
    IN0<='0'; IN1<='1'; IN2<='1'; IN3<='1'; wait for 10 ns;
    IN0<='1'; IN1<='0'; IN2<='0'; IN3<='0'; wait for 10 ns;
    IN0<='1'; IN1<='0'; IN2<='0'; IN3<='1'; wait for 10 ns;
    IN0<='1'; IN1<='0'; IN2<='1'; IN3<='0'; wait for 10 ns;
    IN0<='1'; IN1<='0'; IN2<='1'; IN3<='1'; wait for 10 ns;
    IN0<='1'; IN1<='1'; IN2<='0'; IN3<='0'; wait for 10 ns;
    IN0<='1'; IN1<='1'; IN2<='0'; IN3<='1'; wait for 10 ns;
    IN0<='1'; IN1<='1'; IN2<='1'; IN3<='0'; wait for 10 ns;
    IN0<='1'; IN1<='1'; IN2<='1'; IN3<='1'; wait for 10 ns;
    wait;
  end process;

  UUT: entity work.LOGIC(RTL)
  port map (
    IN0  => IN0,
    IN1  => IN1,
    IN2  => IN2,
    IN3  => IN3,
    OUT0 => LOGIC_AND,
    OUT1 => LOGIC_OR,
    OUT2 => LOGIC_NAND,
    OUT3 => LOGIC_XOR  );

end architecture TEST;
```

SystemVerilog version :

```
// logic_tb.sv
// Very Simple Test Bench (SystemVerilog)

module LOGIC_TB;
  logic [3:0] IN = 4'b0;
  logic LOGIC_AND, LOGIC_OR, LOGIC_NAND, LOGIC_XOR;

initial repeat (16) #10ns IN=IN+1'b1;

LOGIC dut (
    .IN0  (IN[0]     ),
    .IN1  (IN[1]     ),
    .IN2  (IN[2]     ),
    .IN3  (IN[3]     ),
    .OUT0 (LOGIC_AND ),
    .OUT1 (LOGIC_OR  ),
    .OUT2 (LOGIC_NAND),
    .OUT3 (LOGIC_XOR )  );
endmodule
```