

Embedded System Security – Practical Work

Sécurité des Systèmes Embarqués – Travaux Pratiques

Session 1 - Embedded system design and architecture: basic concepts

1. Working environment – OCAE room and tool access

1.1. Connection on PCs and workstations

Local connection on Unix PCs:

Your login will be "xph3sle6XX" where "XX" must be replaced by the number of your group (e.g. 01, 02, etc. – Note that the group number is specific to this course). The initial password will be provided by your teachers – **this password should be changed just after the first connection and the new password must be reminded by all the students in the group (with the yppasswd command; take care that yppasswd does not work on newer machines, and that using the command through ssh will show your passwords)**

Session termination:

At the end of the session, it is mandatory **to close the PC session**.

1.2 Disk space

Due to network policies, all the accounts have limited quotas. This limit is quite large, however keep in mind that you may **not** have **enough** space to keep all the projects stored or run large simulations. If you think that you may run out of space, when starting a project, **archive** the previous one on a **personal** media, such as a USB key. Alternatively, you can store your project on another a network path, such /tp-fmr, or locally, such as /tmp.

2. Global presentation of the session 1 work

2.1. Contents

The goal of this first part of the practical work is to understand the basics of a development flow for SoPC-based embedded systems. Many files are therefore delivered at the beginning of the work and the first task is to understand what is in these files, and the use of the tools, as explained in the next sections of this document. If you are already confident in this domain, you will go through the first part quickly.

2.2. Goals and expected report contents

The report is expected a couple of weeks after the last TP session and should summarize the work carried out, including the explanation of the parts added to the initial files. Problems encountered and solved during the work should be emphasized.

3. CAD environment on the workstations

3.1. Directories and project structure

The tutorial files are in the directory "Security_lab/AES". The tool configurations can be set up by running the following script:

```
cd Security_lab/AES
source script_InitTP.sh
```

A file called "modelsim.ini" also defines the access paths to the design libraries. The location of this file is defined using an environment variable. It is used by the VHDL compiler and simulator.

The files related to the project of the first session are in a subdirectory named *AES*. The files related to the circuit and testbench descriptions are hence in separated directories, each of them grouping a specific set of files:

- vhd: source files of the circuit and script for behavioral simulation of the circuit blocks
- bench: source files of the testbench and script for validation
- ise_support: script files for the simulation of the architecture within ISE
- vivado_support: template files for developing the Vivado project (peripheral interface, software application template).

Tool commands (e.g. library creation or removal, compilation ...) may be written once in a script file, that can be executed using the command "source".

Warning: all names must be coherent! It is also suggested not to include several entities in the same file, and to give the same name to the file and to the entity.

3.2. Tools and global simulation environment

All parts of the project will be developed in VHDL language.

The main tools used during the lab sessions (in addition to text editors) are:

- Modelsim: VHDL simulation
- ISE: Xilinx synthesis and place and route tool for programmable chips (FPGAs)
- Vivado: Xilinx embedded system environment for development boards with FPGAs

4. Simulation with ModelSim (versions 6.0 or above) as a stand-alone tool

4.1. Standard packages and documentation

Packages are available by using the following clauses in the VHDL source code:

```
library IEEE ;  
use IEEE.STD_LOGIC_1164.all ;  
or  
use IEEE.NUMERIC_STD.all ;
```

4.2. Loading and simulating a design unit

Before simulating (using Modelsim as a stand-alone tool), a library must first be created using the **vlib** command. Please note that some commands will create many file and folders, so it is advised to run them in a suitable folder to keep your home directory cleaner and more readable.

Example: vlib lib_Project

The files to be simulated must then be compiled into the library using the **vcom** command, specifying the target library with the option **-work**. The files must be compiled with respect to the circuit hierarchy, starting from the leaves of the hierarchical tree.

Example: vcom -work lib_Project <vhdl file>

The graphical simulation environment is started using the command **vsim -gui &**. All files in the current directory will be available from the simulation environment, so that simulation command files can be invoked in the simulator (do MyCommands.do). **Several scripts already exist in your project directory to help you running the first commands.** For example, the compilation scripts described above are already in the *vhd* and *bench* directory, stored in the files *scripts*. Try to run the scripts from the AES directory

```
source comp_aes.do  
source comp_bench.do
```

do not change the directory while compiling; otherwise, the libraries will not be able to see each other.

Simulations can be either behavioral or after synthesis or after placement and routing, using the same simulation environment (see respective sections for specificities), but relying on different libraries.

Loading the design unit

The "Workspace" window shows the project structure. Select the library corresponding to the testbench to be simulated (i.e. lib_Project_Bench) and load one of the configurations.

Loading a new simulation unit can be done with the menu *Simulate* → *Start simulation*. The time unit can be specified in this menu ("ns" by default). You can also run the simulation with the command **vsim <entity>:**

```
vsim lib_bench.test_core
```

Observing signals and waveforms

The "Objects" window opens when loading a simulation unit. This window shows the Inputs/Outputs of the block selected in the Workspace (signals defined in the entity), and can also be obtained from the menu *View* -> *Debug Windows*.

Source VHDL descriptions can be displayed from the "Files" section in the Workspace.

Signals to observe as waveforms are defined in the "Objects" window (pop-up menu called with a click on the right mouse button, then "Add to Wave"). The added signals can be:

"Selected signals": signals selected in the "Objects" window before activating the command.

"Signals in Region": entity signals.

"Signals in Design": all internal signals of the whole design.

From a command file or from the command line, the commands "add wave signal_name" or "add wave /*" can also be used to display either one signal or all the signals in the top-level entity.

In the waveform window, cursors can be placed at significant positions to measure durations. Additional cursors can be obtained from menu "Add -> Cursor".

Waveforms can be printed or saved in a Postscript file.

Stimuli definition

A testbench should normally define most of the simulation conditions, as it is the case for this tutorial. However, in some cases, simulation stimuli may be more easily defined directly in the simulation environment. The basic command is the "force" command (available in the pop-up menu in the "Objects" window, or from the command line).

Examples:

force x 000	forces value 000 on signal x from the current simulation time
force x(1) 1	forces at value 1 the first bit of signal x from the current simulation time
force clk 1 30, 0 80 -repeat 100	creates a waveform for signal clk with period 100, clk = 1 at time 30 and clk = 0 at time 80 within the period

Simulation execution

Simulation execution is started with the menu Simulate → Run.

The "Step" command allows step-by-step simulation with source code visualization.

The "run <duration>" command executes "Duration" time units (ex.: run 100). For this example, run for 4.2 us (run 4200 ns, for example).

Command file

A command file may be used either to define the simulation environment (signals to monitor, display options, etc.) or to reproduce some stimuli. It is recommended to call such a file with extension ".do".

The file can be executed from the command line with the command "**do file_name**".

4.3. Simulation after synthesis

The result of the synthesis step is saved as a netlist in VHDL or Verilog (VHDL is recommended here, to be easily able to use the same testbench as before synthesis).

In order to simulate this netlist, it is necessary to specify the simulation models of the cells in the library used during synthesis. This implies to define the cell library to be used in the netlist file, if not present

The testbench (and/or the simulation command file) used for behavioral simulations can be reused, by modifying the pointed descriptions in the configuration files (e.g. libraries in the "synth" directory rather than the "vhd" directory).

4.4. Simulation after placement and routing

After placement and routing, the parasitic data must be fed back in the simulation by means of an "sdf" file, specified in the menu "Simulate" (validate "Reduce SDF errors to warnings"). Post P&R simulation will be described more in detail in the next chapter.

5. Synthesis, placement and routing with ISE (version 14.7)

5.1. Documentation

The ISE tool is used to implement a design on an FPGA chip. The environment can be started using the `ise` command at the terminal prompt. The easiest way is to run the `ise` command from the directory `Security_lab/AES/synth/ise`. Run

```
ise &
```

If the environment variables are properly set up, the main application will pop up in a few seconds. An extensive online help can be invoked from the menu.

In the next sections it will be shown how to create a simple project implementing an AES encryption core.

5.2. Creating an ISE project

1. Open the ISE application
2. **Create a New Project:** choose the menu item

File → New Project...

This will start the *New Project Wizard*.

3. **New Project Wizard.** A new window pops up:

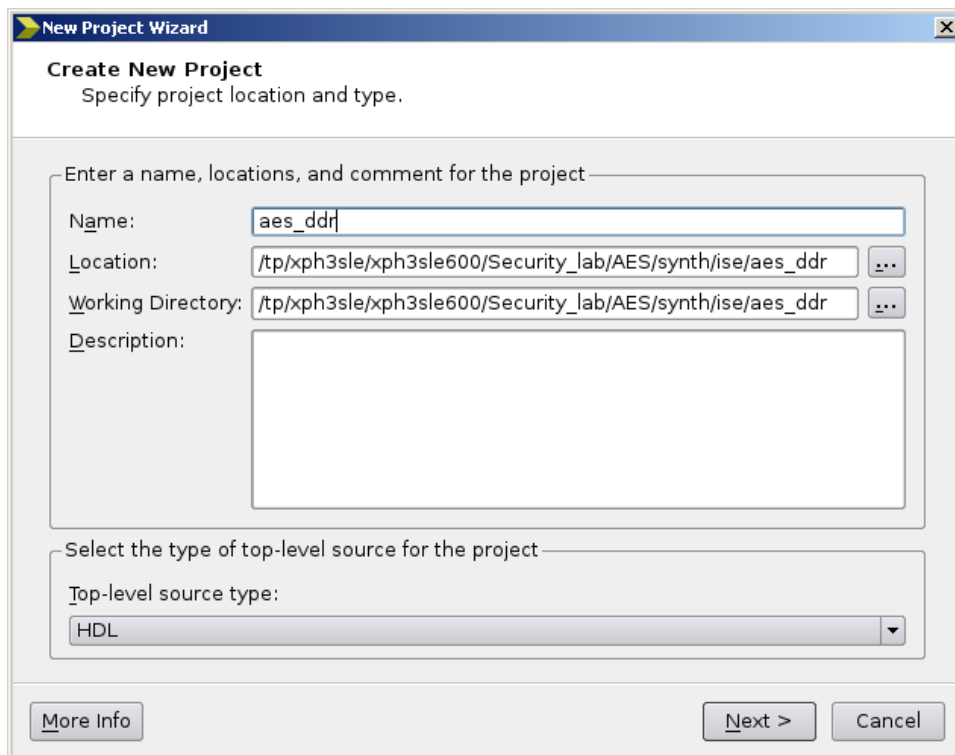


Figure 1: New ISE project wizard

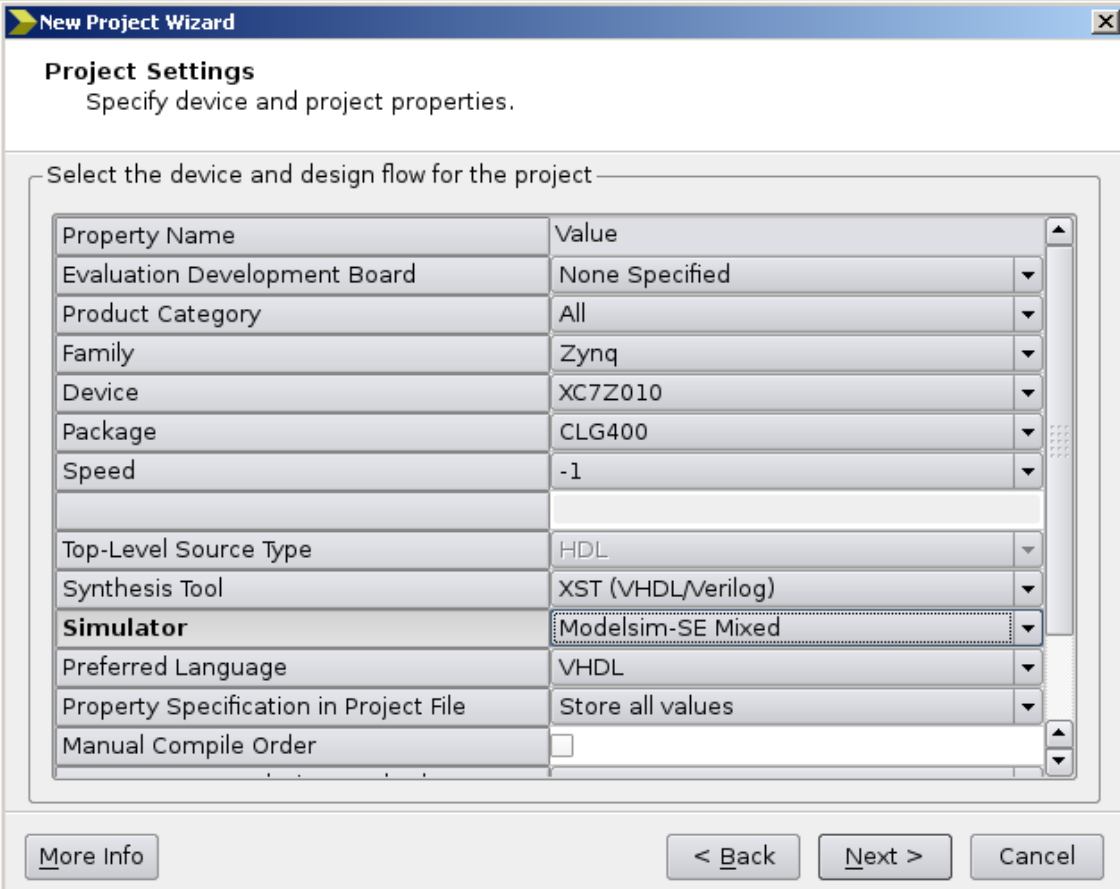
Write a name for the project in *Project Name* field and change the location of the project folder (a subfolder is created using the project name by default). The name you choose is not

mandatorily the one shown in Figure 1, but consider that changing this may force you to modify several parameters and files that are already provided you to start the project.

Leave the other fields at their default value. Click the *Next* button.

Note: it is advisable to avoid spaces in the name; rather, use points (“.”), underscores (“_”), or dashes (“-”).

4. The window is now entitled “*Device properties*”. Here you can choose the target implementation device: either choose the specifications of your target board or leave the default values. This can be changed anytime later: for the instant, **define** the chip parameters as shown in Figure 2; **set** also the *Synthesis* and *Simulation* tool as in the figure. For simulation, you can use either ModelSim or the integrated ISim simulator. Finish the wizard.



New Project Wizard
Project Settings
Specify device and project properties.

Select the device and design flow for the project

Property Name	Value
Evaluation Development Board	None Specified
Product Category	All
Family	Zynq
Device	XC7Z010
Package	CLG400
Speed	-1
Top-Level Source Type	HDL
Synthesis Tool	XST (VHDL/Verilog)
Simulator	Modelsim-SE Mixed
Preferred Language	VHDL
Property Specification in Project File	Store all values
Manual Compile Order	<input type="checkbox"/>

More Info < Back Next > Cancel

Figure 2: Device properties

5. Select the menu *Project* → *Add Sources*. You can add any already existing file to the project. The selected files do not need to be complete or error free: debugging and completion can be performed later, once the project is defined and open.

Add all the provided VHDL source files: the AES description (in the `vhd` directory, for both implementation and simulation) and the test bench (in the `bench` directory, only for simulation). Since you will use custom scripts for simulations, you may either leave the default value in the library field, or set a new one yourself as in the figure below.

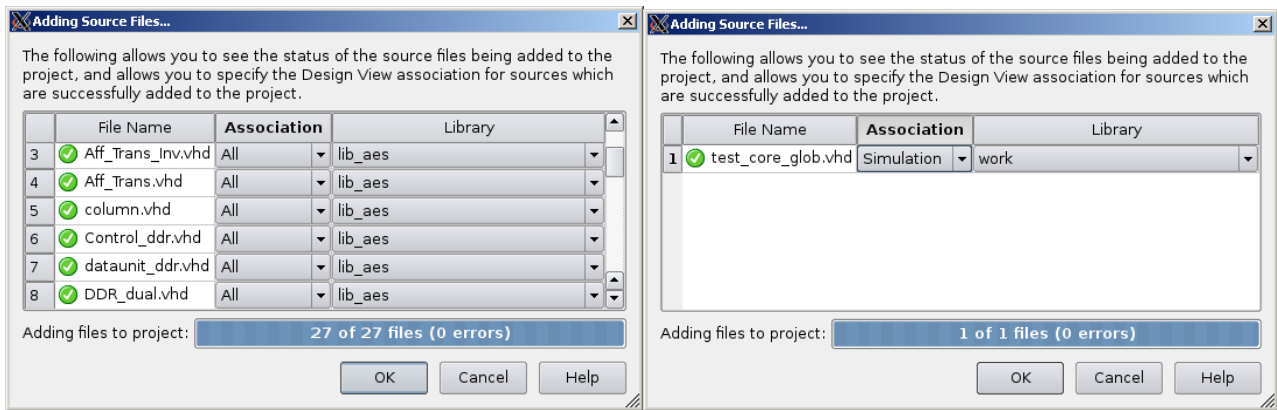


Figure 3: Add existing HDL sources

6. **Application interface.** The application window is divided into several areas which have different goals:

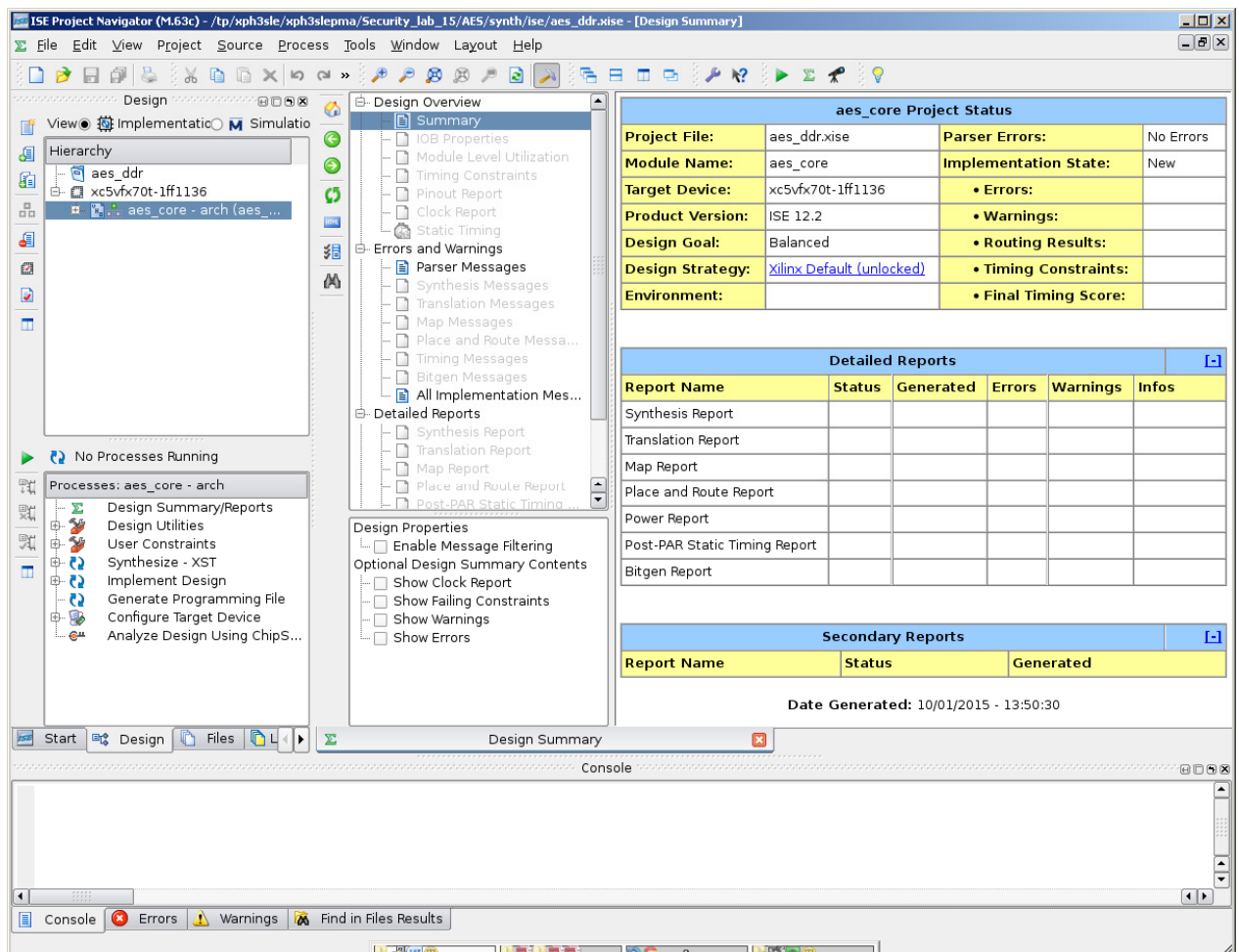


Figure 4: Main application window

In the top left part, there is the main project browser, which can be used to browse within the files and the library of the project. **Note that a menu at the top edge** allows selecting the current activity: synthesis or simulation at different levels. The root of the file hierarchy is

Embedded Systems Security – Sécurité des Systèmes Embarqués – SLE 3A – 2017/2018

the target device: double clicking allows modifying the target device and other configuration settings.

The middle left window lists the processes that can be started on the element selected in the file browser: for instance, for a synthesizable component it allows starting the synthesis, the mapping, the placing and routing and so on. The application tracks all the changes to files and settings: hence, if a process has already run and no changes to the source files and settings occurred, running it again must be explicitly ordered. Likely, after changes to sources or settings, the validity of the results expires and the process must be run again. If a specific process requires that others are run in advance, the application resolves all the dependencies by itself and executes all those required.

In the right part of the window there is the *Design Summary*. Here the results of the executed processes can be accessed quickly and examined. For instance, this window reports the device utilization rate and the number of errors/warning occurred during each process, which can be accessed easily through these links.

The bottom window displays the output of the current running process. During the process execution, it shows the current activity, warnings and errors.

7. **Pre-synthesis verification.** Before starting the implementation of the architecture, it is advisable to do a preliminary test of the design. This can be done by simulating the design at the *behavioral* level. A behavioral simulation takes into consideration only the functionality of a component and it does not require details about its actual implementation or internal structure. The behavioral simulation can run directly from the ISE application by selecting *Behavioral Simulation* in the drop down source filter and the test bench file, which should be the first element just below the target board name. The simulator can be started by executing the *Simulate* command:

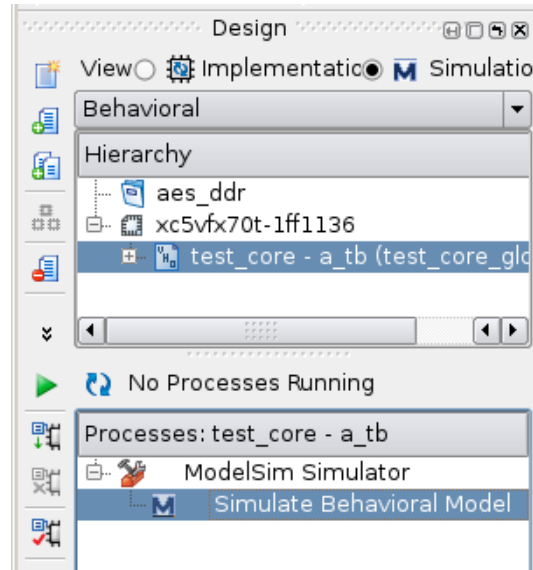


Figure 5: Behavioral simulation (pre-synthesis)

This command starts the simulator; when set appropriately, it also loads and compiles the source modules, and starts the simulation for a predetermined amount of time (usually 1 μ s). This is accomplished since the default behavior is to use the script file that is generated automatically. For our example, however, **open** the *Properties* window of the *Simulate Behavioral Model* command and **select the user script file** as shown in Figure 6.

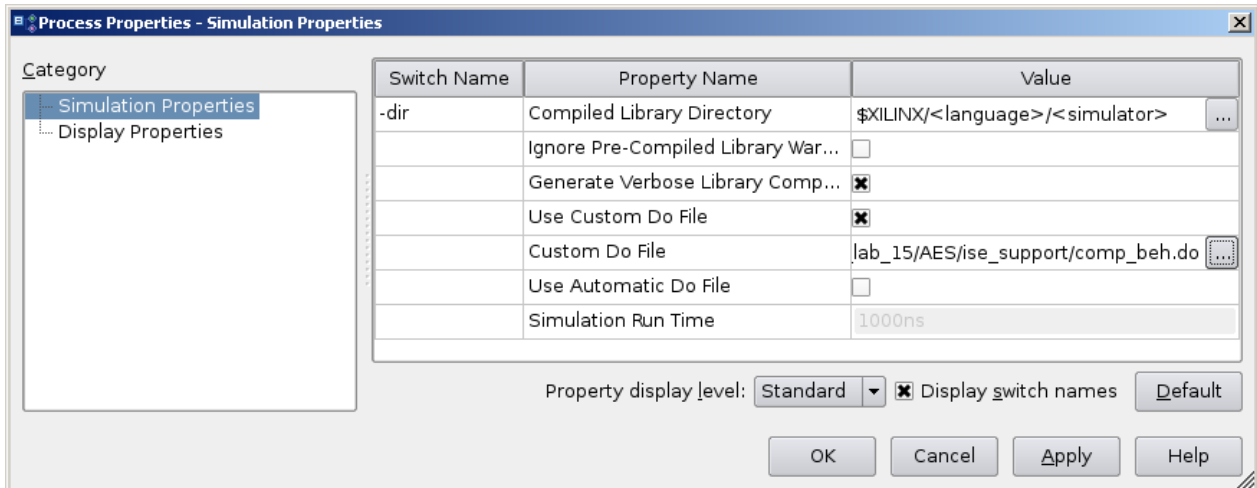


Figure 6: Parameters for behavioral simulation

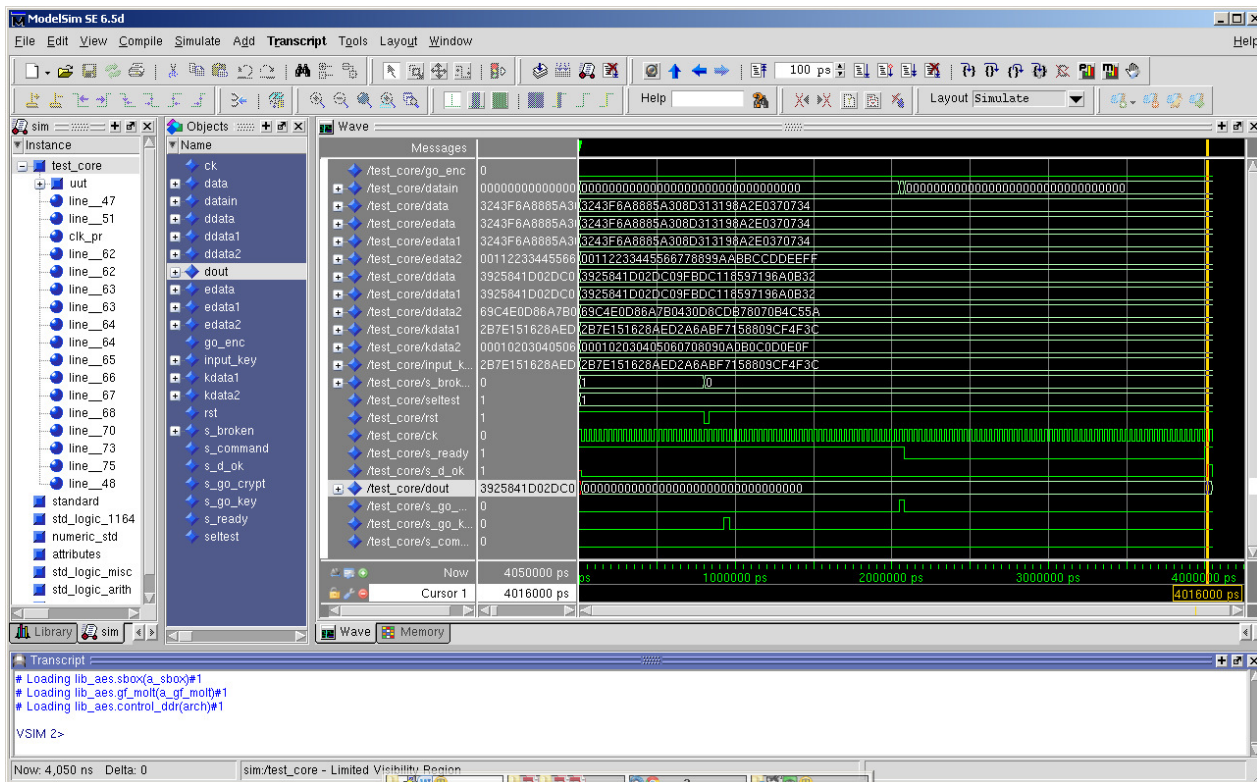


Figure 7: Behavioral simulation in ModelSim

It can be noted that the simulation environment shows all the signals defined in the test bench. Internal signals can be added and the simulation restarted to further explore the behavior of the system. It can be seen that many internal signals are still encoded in symbolic form (namely, the state signals), which is typical of behavioral simulation.

before launching the synthesis again. If there are no errors, then a green tick appears near the *Synthesize* command and the summary windows reports the results: device utilization rate, number of warnings, possible errors and auxiliary information. Now it is possible to proceed to verify the synthesized design by simulation.

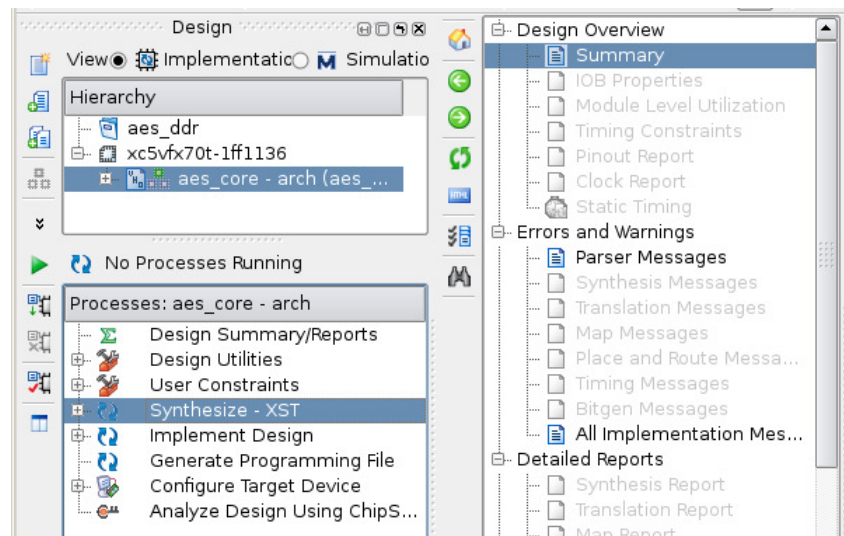


Figure 9: Starting the synthesis process

9. **Post-synthesis verification.** If the synthesis completed successfully, then it is advisable to simulate the result, in order to verify that it did not alter the expected timing of the signals. This requires that the design is synthesized and translated, and the simulation model is generated after the translation step (Figure 10). The simulation of the post-translate model can be started by selecting *Post-Translate Simulation* in the drop-down menu, and then running the command *Simulate....* In the project of the first session, open the *Process Properties* window of the *Simulate* command, and select the custom script file as in Figure 11. Verify that your design behaves as expected even after synthesis. Note that the representation, or the existence itself, of some signals has changed.

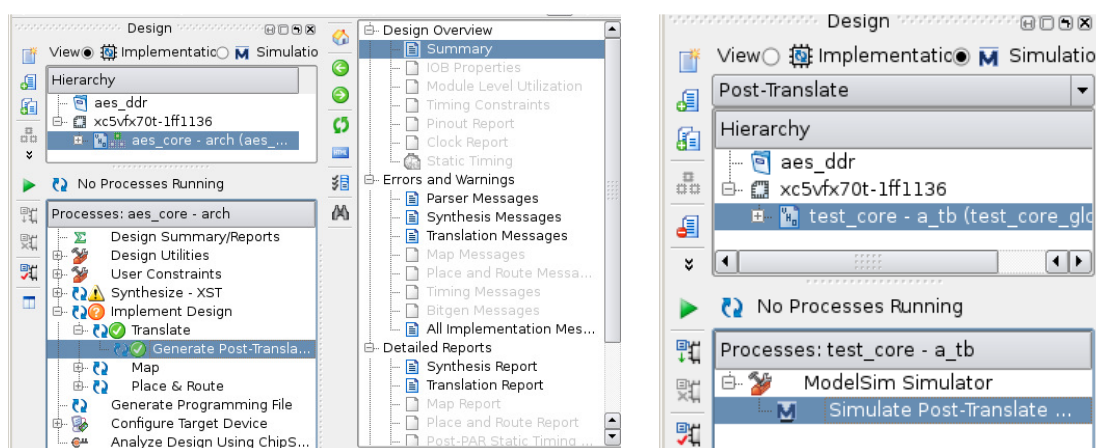


Figure 10: Starting the post-synthesis simulation

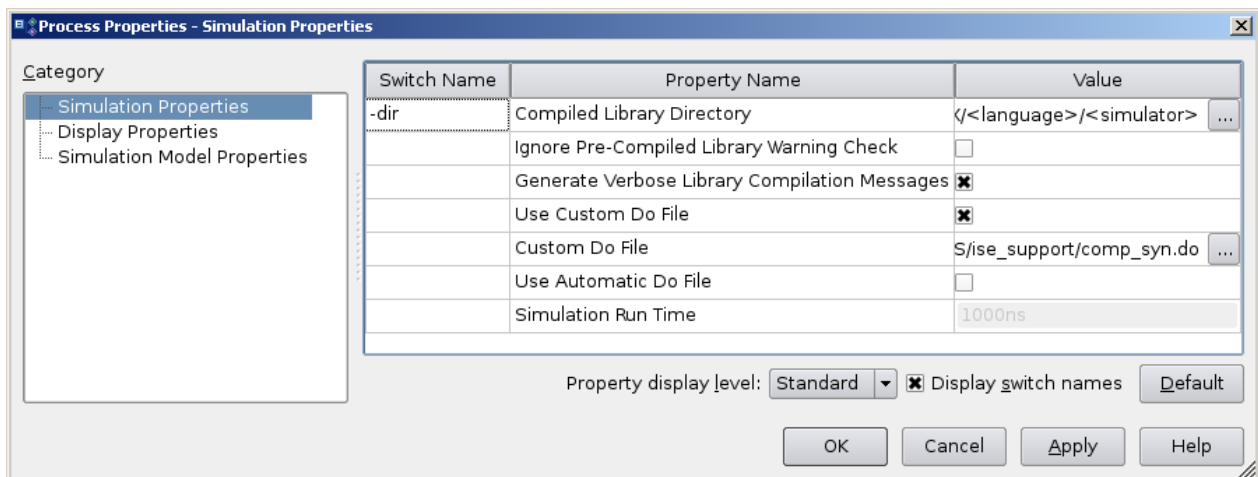


Figure 11: Parameters for the Post-Translate Simulation

10. **Mapping, Placing and Routing.** After synthesis, the FPGA workflow can be completed by running the *mapping* process and then running the *Place & Route* phase. This produces the actual bit stream that will be sent to the device board during the programming phase. Both phases can be verified by generating the appropriate simulation model and starting the simulator. The methodology is the same described for the post-synthesis simulation, provided that the correct commands are selected. However, in order to embed the device core in the EDK design flow, this is **not** required.

Note. The synthesis converts HDL code into a gate-level netlist (represented in the terms of the *UNISIM* component library, a *Xilinx* library containing basic primitives). The implementation stage is intended to translate netlist into the placed and routed FPGA design. During the translate phase the NGC/EDIF netlist is converted to an NGD netlist. The difference between them is in that NGC netlist is based on the UNISIM component library, designed for behavioral simulation, and NGD netlist is based on the SIMPRIM library. The netlist produced by the NGDBUILD program contains some approximate information about switching delays. During the map phase the SIMPRIM primitives from an NGD netlist are mapped on specific device resources: LUTs, flip-flops, BRAMs and other. The output file contains precise information about switching delays, but no information about propagation delays, since the layout hasn't been processed yet. Finally, Place and route defines how device resources are located and interconnected inside an FPGA. It is the most important and time consuming step of the implementation.

NOTE: The mapping, placing and routing phase are not required when embedding the synthesized core in an EDK project. More precisely, these steps can be completed only when IO ports of the top module are buffered, which is in contrasts with the integration of a netlist within EDK; **on the other hand, this option must NOT be selected in the *Synthesis Properties* window when synthesizing the core for EDK.**

6. Using Vivado (version 2017.1)

The Vivado tool is used to build complex embedded systems; the environment can be started using

`vivado &`

If the environment variables are properly set up, the main application will pop up in a few seconds.

1. **Create a project.** The user may start a new project (either by following a wizard or defining a new project from scratch) or open a previous project. Create a new project in the `synth/vivado` subdirectory: this will be an RTL project, no sources specified at the moment (but VHDL specified as preferred target language). Choose the Zybo board as target platform

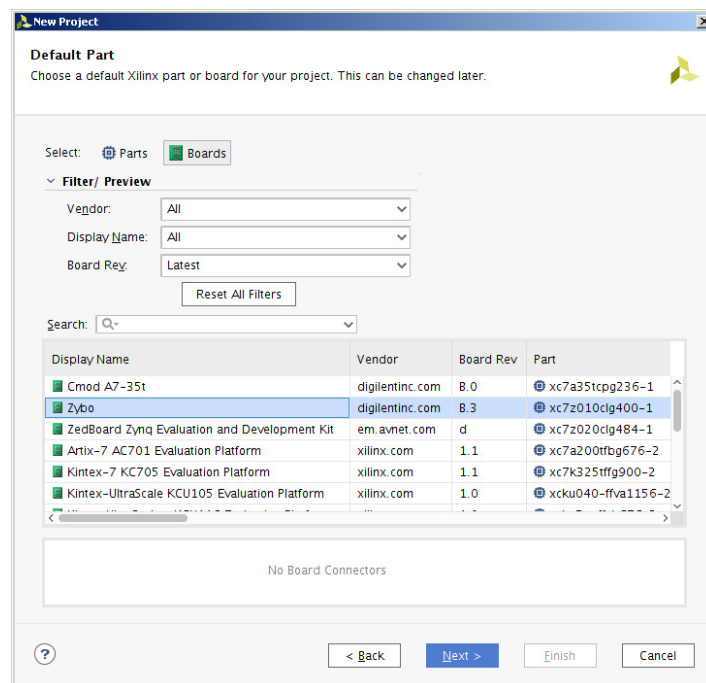


Figure 12: EDK wizard at startup

The project is currently empty. Create a new Block design from the IP Integrator. You will get an empty design to work in, where you can start adding or creating IPs. Use the *Add IP* button to add a MicroBlaze processor.

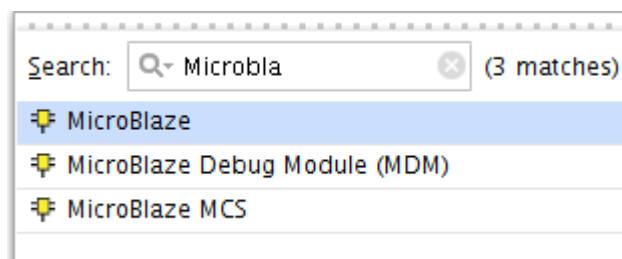


Figure 13: Adding a Microblaze processor to the system

The tool tries to help the user to automate as many tasks as possible. For instance, Block Automation is proposed when only the MicroBlaze is instantiated in the design. The tool asks for

some information, needed in order to build a sound design, then instantiate and connects the required modules.

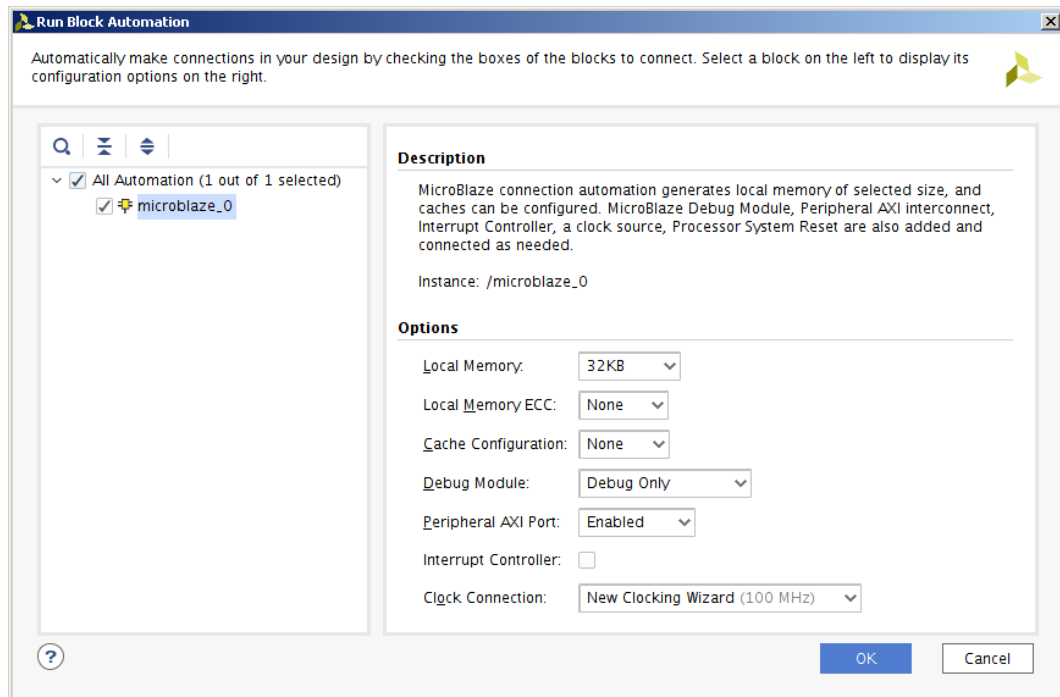


Figure 14: Block automation for MicroBlaze

The memory and a clock generator are instantiated.

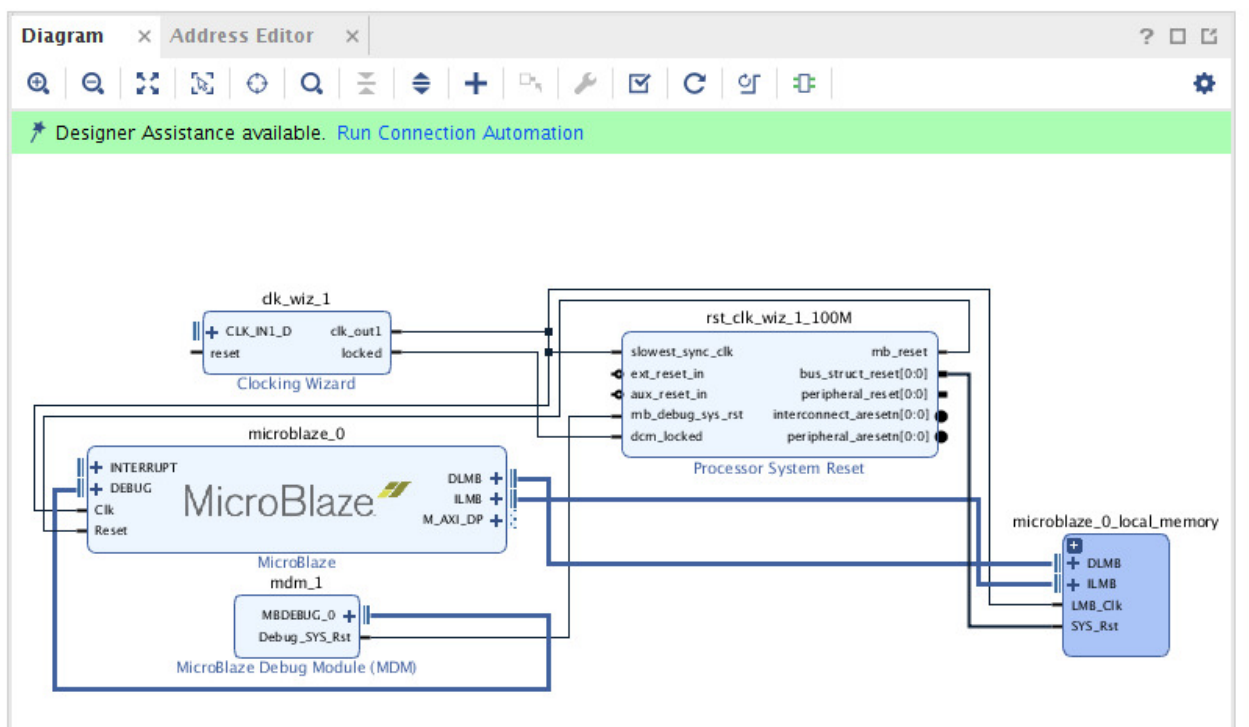


Figure 15: The basic μ Blaze system

At this time it is necessary to adapt some of the instantiated modules to the constraints of the project. Customize the clocking wizard: in Clocking Options, set the input clock as a single pin input at 125 MHz; in Output Clocks, add an additional clock output running at 40 MHz.

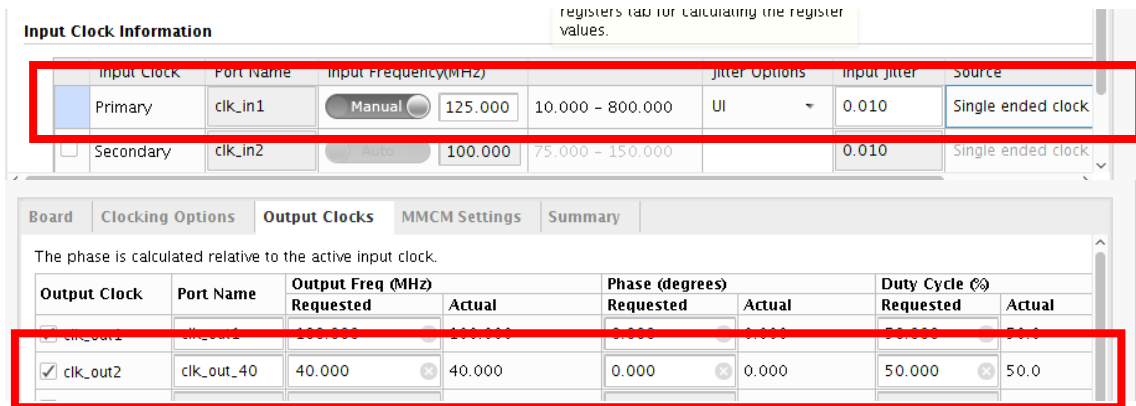
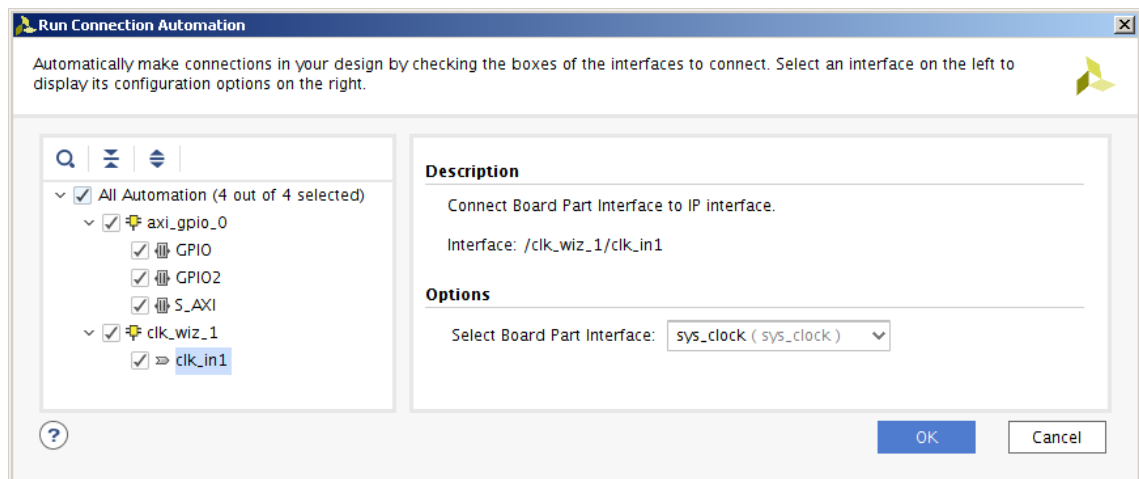


Figure 16: Clocking configuration

You need to add Constant blocks to generate a high signal for the external reset port of the Processor System Reset (active low), and a low signal for the reset port of the Clocking Wizard (active high): connect these signals to the corresponding ports. Finally, you need to instantiate (and configure) the GPIO module for the buttons and LEDs available on the board. Run the Connection Automation Wizard to complete (temporarily) the design.



2. **Create and Package a new IP.** Choose the corresponding item from the Tools menu. This will start the Wizard which enables the user to create his own peripherals and easily connect them to the AXI bus. The interface to the bus will be automatically generated.

Choose to create an AXI4 peripheral names “aes_dds”. The peripheral will be a slave on the AXI Lite bus, and will be equipped with **14 slave registers**. At the final step, choose to **edit immediately** the IP just defined. This will open the sub-project of the peripheral inside Vivado.

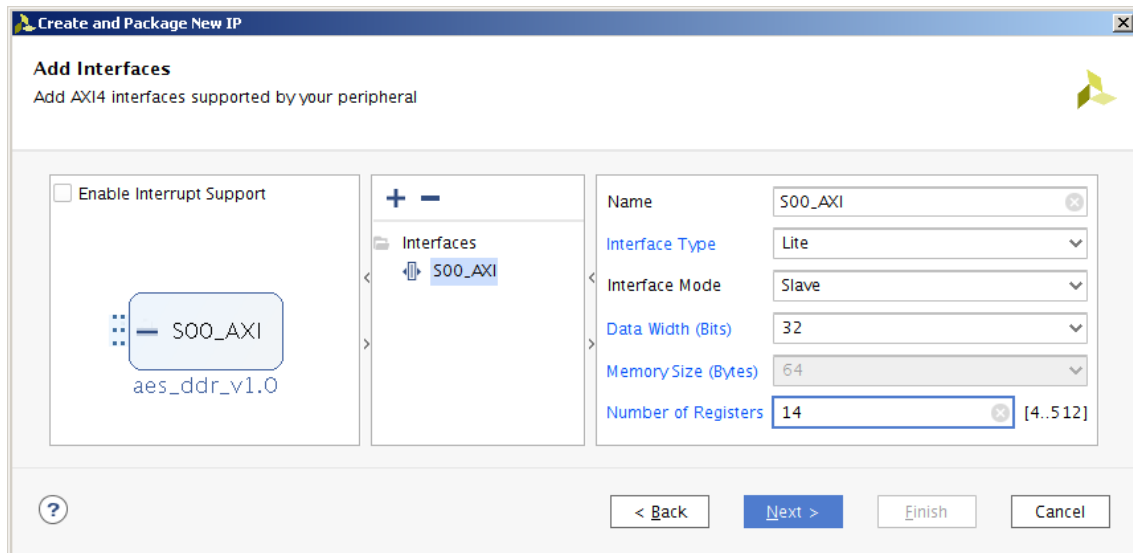


Figure 17: AES peripheral interface

The peripheral just created contains only the user addressable slave registers and the logic needed for the AXI interface. We need to add the IP for AES encryption, and the required glue logic.

First, add the netlist generated by ISE as an additional source: the corresponding item will appear in the source list, still not in the main hierarchy as it is not instantiated.

The netlist must be instantiated and made accessible. You need to modify the two source files that have been automatically generated. In the top module, you need to add an additional user port feeding a slower clock to the AES DDR, which is not able to work at the frequency of the AXI bus. This port must be added and passed also to the lower-level module. See the corresponding file in the `vivado_support` directory for details.

The lower level model requires larger interventions: the clock (as said above), the AES module, the glue logic; moreover, the access to the slave registers should be modified in order to meet the application constraints and avoid write conflicts. Again, check the corresponding source file provided in the `vivado_support` directory for details.

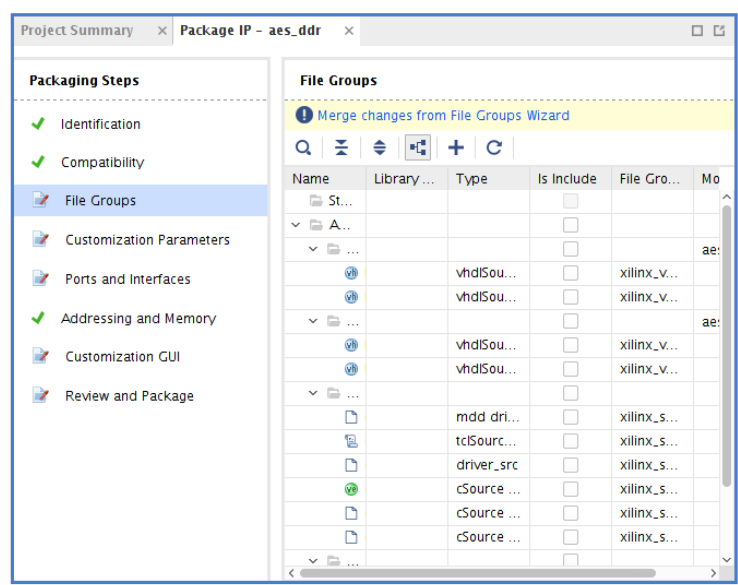


Figure 18: Repackaging the new peripheral

In the package IP window, review all the modified tabs and let the tool merge the changes. Feel free to ignore the warning in Ports and Interfaces, and then finally repackage the IP and close the sub-project.

You are now back to the main project, where you can add your new IP to the design and let the tool perform the connections. Take care to select the clock at 40 MHz to feed the slower clock of your custom peripheral.

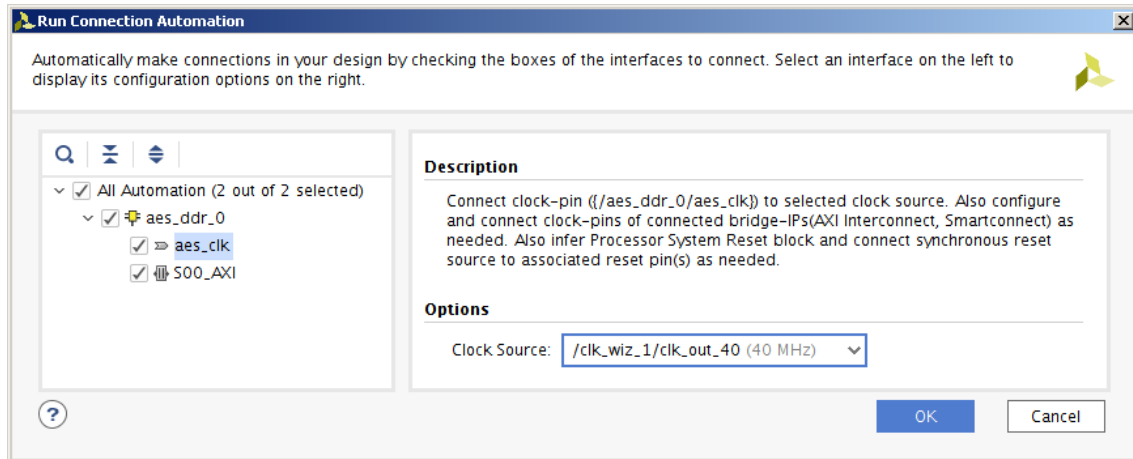


Figure 19: Connecting the AES clock

Check that the design does not contain any error by running the Validation Tool.

You might be already able to simulate your system, but there are two issues: there is no software defined to run on the CPU yet, and accessorially the AES IP is defined as a netlist, thus not having a behavioral simulation model.

3. **Synthesis and validation.** You are ready to build your system. Generate block design lets the tool create the VHDL source files describing your project; then, you can generate the VHDL wrapper which will be the top module of your system. Let the tool manage the generated files.

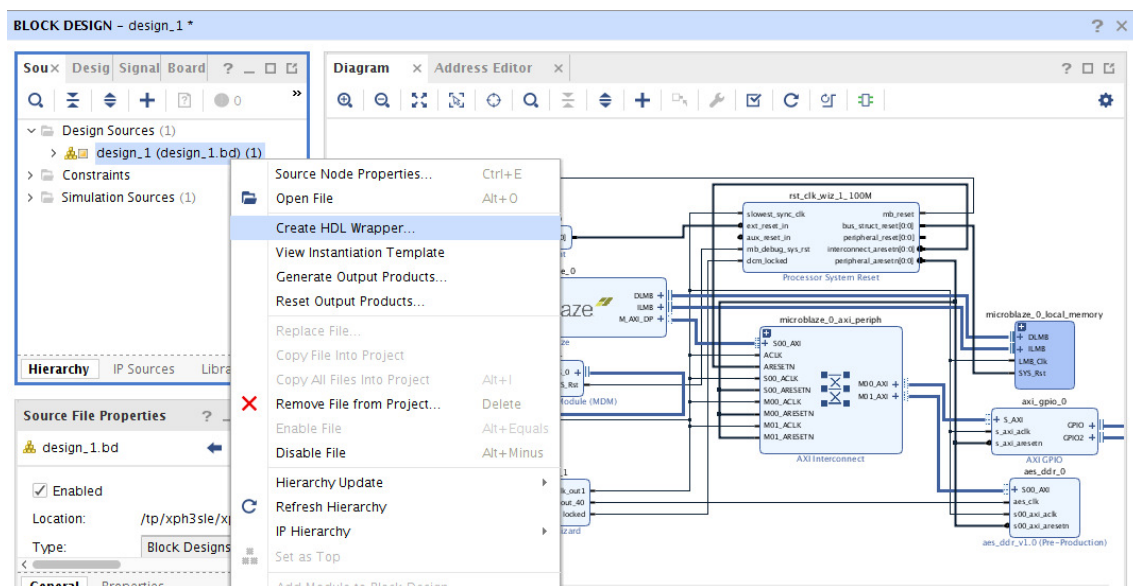
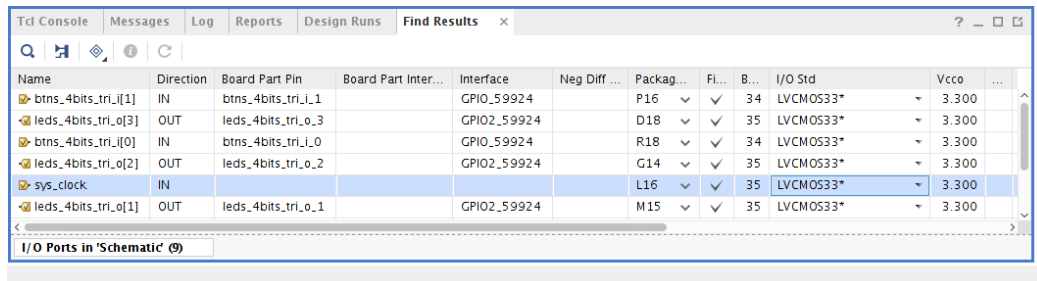


Figure 20: Generating the project wrapper

Having chosen a specific board at the beginning, almost everything is correctly set up. You just need to assign the correct system clock pin. Open the Elaborated Design, and ask the tool to show all the external interfaces by selecting Ports in the Schematic view.

All the pins are pre-assigned, except for the input clock. Select the pin **L16** and the **LVC MOS33** I/O standard.



Name	Direction	Board Part Pin	Board Part Inter...	Interface	Neg Diff ...	Packag...	Fl...	B...	I/O Std	Vcco	...
btms_4bits_tri_i[1]	IN	btms_4bits_tri_i_1		GPIO_59924		P16	✓	34	LVC MOS33*	3.300	
leds_4bits_tri_o[3]	OUT	leds_4bits_tri_o_3		GPIO2_59924		D18	✓	35	LVC MOS33*	3.300	
btms_4bits_tri_i[0]	IN	btms_4bits_tri_i_0		GPIO_59924		R18	✓	34	LVC MOS33*	3.300	
leds_4bits_tri_o[2]	OUT	leds_4bits_tri_o_2		GPIO2_59924		G14	✓	35	LVC MOS33*	3.300	
sys_clock	IN					L16	✓	35	LVC MOS33*	3.300	
leds_4bits_tri_o[1]	OUT	leds_4bits_tri_o_1		GPIO2_59924		M15	✓	35	LVC MOS33*	3.300	

Figure 21: I/O assignment

Save the modified constraints in a new file, which will be automatically added to the project. You can start the synthesis process in order to obtain the system's netlist.

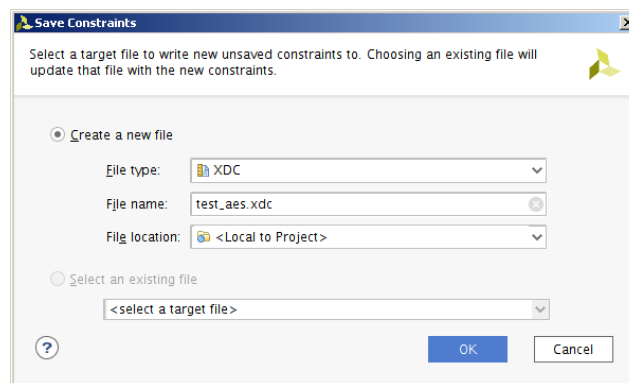


Figure 22: Saving constraints file

- Software application.** You may now Export the Hardware from the File>Export menu and then launch the SDK. Once open, this tool will present you some information about the current system, such as the CPU, the peripherals, and so on. Create a New Application Project, written in C language and running on the standalone OS. You may choose a pre-defined template, as well as an empty project.

Note: On some machines, this step may fail. You need to connect to older machines (such as ocaepc13, 14 and 19) and run the SDK tool manually. You can find an example script `xsdk.sh` in the main AES directory.

With the project created, you can now create a new source file where you can specify the code that will be executed by the CPU. You have a very simple example in the `vivado-support` directory: the `test_aes.c` file gives you an example of using the AES coprocessor, you can modify this file to verify your design. Build the application in order to generate an executable elf file, exit the SDK and go back to Vivado.

5. **Update the executable.** In order to simulate (and run the design on the board) you need to provide executable file of the software program, so select Tools>Associate ELF Files and pick the corresponding file

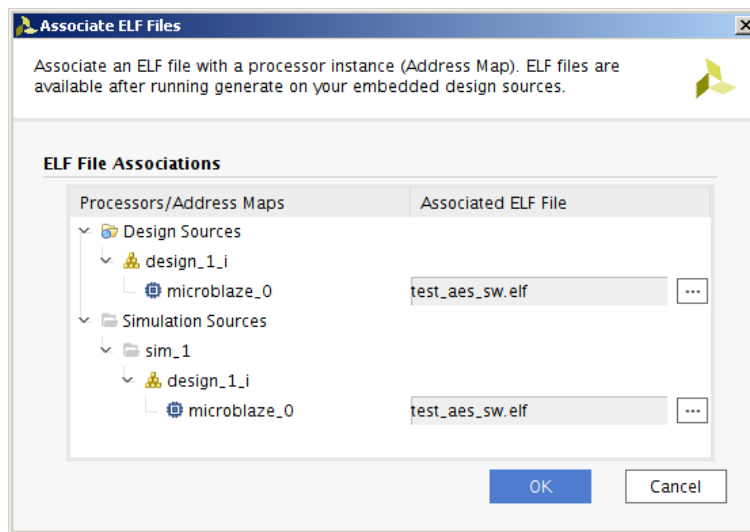


Figure 23: Setting the executable file

6. **Simulate.** In order to simulate your design, you need to write yourself the test bench providing the required stimuli. Hence, add a simulation source, and write the code to generate the 125 MHz clock input and instantiate the design. You can either run functional or timing post-synthesis simulation, depending on the speed and level of detail you are looking for. Depending on the choice made during the project setup, the simulation will be run either in the integrated simulator environment (Vivado simulator) or ModelSim; for the latter, the libraries need to be compiled and referenced in the modelsim.ini file. Load your test bench in the simulator, pick a few signals to show in the wave window (for instance, the slave registers and AES port of the custom peripheral), and run the simulation up to 13 μ s and look for the results.
7. **Implementation.** Once the hardware structure has been completely defined, the implementation process can start: this will run the mapping, placement, and routing processes. This may be quite a time-consuming step.
8. **Simulate.** Once the design is implemented, you can run post-P&R simulation to verify that all the constraints are satisfied. Run again the simulation, choosing now a lower level; as earlier, you can choose functional simulation for a faster run, or timing for a more precise description of the system's behavior.