GRLIB IP Library User's Manual

Oct 2025, Version 2025.2

Frontgrade Gaisler AB
Kungsgatan | SE-411 19 | Goteborg | Sweden
+46 31 7758650 | frontgrade.com/gaisler

Table of contents

1 Introduction 6
1.1 Scope 6
1.2 Other resources 6
1.3 Overview 6
1.4 Library organization 6
1.5 On-chip bus 6
1.6 Distributed address decoding 7
1.7 Interrupt steering 7
1.8 Plug&Play capability 7
1.9 Portability 8
1.10 Available IP cores 8
1.11 Versions 9
1.12 Licensing 9
2 Installation 10
2.1 Installation 10
2.2 Upgrading 10
2.3 Directory organization 10
2.4 Host platform support 11
2.4.1 Linux 11
2.4.2 Windows with Cygwin 11
2.5 Installation of simulation libraries 12
2.5.1 Installation of Altera libraries 12
2.5.2 Installation of Microsemi libraries 12
2.5.3 Installation of Xilinx libraries 13
2.5.4 Installation of DARE+ libraries 13
2.5.5 Installation of NanoXplore libraries 13
2.5.6 Installation of Lattice Radiant libraries 13
3 LEON/GRLIB quick-start guide 15
3.1 Introduction 15
3.2 Overview 15
3.3 Configuration 16
3.4 Simulation 16
3.5 Synthesis and place&route 17
3.6 Simulation of post-synthesis netlist 18
3.7 Board re-programming 18
3.8 Running applications on target 18
3.9 Flash PROM programming 19
3.10 Software development 19
4 Implementation flow 20
4.1 Introduction 20
4.2 Using Makefiles and generating scripts 20
4.3 File attributes 22
4.4 Simulating a design 23
4.4.1 Overview 23
4.4.2 GRLIB_SIMULATOR environment variable 23
4.5 Synthesis and place&route 25
4.6 Skipping unused libraries, directories and files 26
4.7 Encrypted RTL 28

4.8 Tool-specific usage ... 29
4.8.1 GNU VHDL (GHDL) ... 29
4.8.2 Cadence ncsim ... 30
4.8.3 Mentor FormalPro ... 32
4.8.4 Mentor Questa/ModelSim ... 33
4.8.5 Aldec Active-HDL ... 35
4.8.6 Aldec ALINT ... 37
4.8.7 Aldec Riviera ... 38
4.8.8 Synopsys VCS ... 39
4.8.9 Synthesis with Synplify ... 41
4.8.10 Synthesis with Mentor Precision ... 43
4.8.11 Actel/Microsemi Designer ... 45
4.8.12 Microsemi Libero ... 47
4.8.13 Altera Quartus ... 51
4.8.14 Xilinx ISE ... 53
4.8.15 Xilinx PlanAhead ... 56
4.8.16 Xilinx Vivado ... 58
4.8.17 Lattice Radiant ... 61
4.8.18 Lattice ISP Tools ... 63
4.8.19 Synthesis with Synopsys Design Compiler ... 64
4.8.20 Synthesis with Cadence RTL Compiler ... 64
4.8.21 eASIC eTools ... 65
4.8.22 NanoXplore NanoXmap and NanoXpython ... 66
4.9 XGrlib graphical implementation tool ... 68
4.9.1 Introduction ... 68
4.9.2 Simulation ... 68
4.9.3 Synthesis ... 69
4.9.4 Place & Route ... 69
4.9.5 Additional functions ... 69
5 GRLIB Design concept ... 70
5.1 Introduction ... 70
5.2 AMBA AHB on-chip bus ... 70
5.2.1 General ... 70
5.2.2 AHB master interface ... 71
5.2.3 AHB slave interface ... 72
5.2.4 AHB bus control ... 73
5.2.5 AHB bus index control ... 73
5.2.6 Support for wide AHB data buses ... 73
5.3 AHB plug&play configuration ... 75
5.3.1 General ... 75
5.3.2 Device identification ... 76
5.3.3 Address decoding ... 78
5.3.4 Cacheability ... 78
5.3.5 Interrupt steering ... 79
5.4 AMBA APB on-chip bus ... 80
5.4.1 General ... 80
5.4.2 APB slave interface ... 81
5.4.3 AHB/APB bridge ... 82
5.4.4 APB bus index control ... 82
5.5 APB plug&play configuration ... 83
5.5.1 General ... 83
5.5.2 Device identification ... 83
5.5.3 Address decoding ... 83
5.5.4 Interrupt steering ... 84
5.6 Endianness configuration ... 84
5.6.1 APB accesses ... 85
5.6.2 AMBA plug&play ... 85

5.7 GRLIB configuration package 85
5.8 Technology mapping 87
5.8.1 General 87
5.8.2 Memory blocks 88
5.8.3 Memory collision handling 89
5.8.4 Memory power-down optimizations 89
5.8.5 Pads 89
5.9 Scan test support 90
5.9.1 Overview 90
5.9.2 GRLIB support 91
5.9.3 Usage for existing cores 91
5.9.4 Usage for new cores 91
5.9.5 Configuration options 91
5.10 Support for integrating memory BIST 92
5.10.1 Syncram level 92
5.10.2 IP core level 92
5.10.3 Design level 93
5.11 GRLIB system test software 93
5.11.1 Introduction 93
5.11.2 Typical test software use 94
5.11.3 Test software reporting 94
5.11.4 Selecting the right test module 95
5.11.5 Standalone systest 95
6 GRLIB Design examples and FPGA board template designs 96
6.1 Introduction 96
6.2 Supported FPGA boards 97
6.3 LEON3MP - Generic multiprocessor system 99
6.4 LEON3ASIC - ASIC flow example design 100
6.4.1 Modification of GRLIB Scripts 100
6.4.2 RTL Simulation scripts 101
6.4.3 Synthesis scripts 102
6.4.4 Formal verification scripts 102
6.4.5 GTL Simulation scripts 102
6.5 Xilinx Dynamic Partial Reconfiguration Examples 102
6.6 Microsemi designs 103
6.6.1 Simulating from Libero v12.0 103
6.6.2 Libero projects with encrypted RTL on Windows 103
6.6.3 Using the template designs 103
7 Using netlists 108
7.1 Introduction 108
7.2 Mapped VHDL 108
7.3 Xilinx netlist files 108
7.4 Altera netlists 108
7.5 Known limitations 108
8 Extending GRLIB 109
8.1 Introduction 109
8.2 GRLIB organisation 109
8.2.1 Encrypted RTL 110
8.3 Adding an AMBA IP core to GRLIB 111
8.3.1 Example of adding an existing AMBA AHB slave IP core 111
8.3.2 AHB Plug&play configuration 112
8.3.3 Example of creating an APB slave IP core 113
8.3.4 APB plug&play configuration 114
8.4 Adding a design to GRLIB 115

8.4.1 Overview ... 115
8.4.2 Example: Adding a template design for Nexys4 ... 115
8.5 Using verilog code ... 121
8.6 Adding portabilty support for new target technologies ... 121
8.6.1 General ... 121
8.6.2 Adding a new technology ... 122
8.6.3 Encapsulation ... 122
8.6.4 Memories ... 122
8.6.5 Pads ... 124
8.6.6 Clock generators ... 124
8.7 Extending the xconfig GUI configuration ... 124
8.7.1 Introduction ... 124
8.7.2 IP core xconfig files ... 124
8.7.3 xconfig menu entries ... 126
8.7.4 Adding new xconfig entries ... 126
8.7.5 Other uses and limitations ... 128

1 Introduction

1.1 Scope

This document describes the GRLIB IP library infrastructure, organization, tool support and on-chip bus implementation.

1.2 Other resources

There are several documents that together describe the GRLIB IP Library and Frontgrade Gaisler's IP cores:

- GRLIB IP Core User's Manual (grip.pdf) - Describes specific IP cores provided with the GRLIB IP library. Also specifies which cores that are included in each type of GRLIB distribution.
- GRLIB-FT User's Manual (grlib-ft.pdf) - Describes the FT and FT-FPGA versions of the GRLIB IP library. The document is an addendum to the GRLIB IP Library User's Manual. This document is only available in the FT and FT-FPGA distributions of GRLIB.
- GRLIB FT-FPGA Xilinx Add-on User's Manual (grlib-ft-fpga-xilinx.pdf) - Describes functionality of the Virtex5-QV and Xilinx TMRTool add-on package to the FT-FPGA version of the GRLIP IP library. The document should be read as an addendum to the 'GRLIB IP Library User's Manual' and to the GRLIB FT-FPGA User's Manual. This document is only available as part of the add-on package for FT-FPGA.
- LEON/GRLIB Configuration and Development Guide (guide.pdf) - This configuration and development guide is intended to aid designers when developing systems based on LEON/GRLIB. The guide complements the GRLIB IP Library User's Manual and the GRLIB IP Core User's Manual. While the IP Library user's manual is suited for RTL designs and the IP Core user's manual is suited for instantiation and usage of specific cores, this guide aims to help designers make decisions in the specification stage.

1.3 Overview

The GRLIB IP Library is an integrated set of reusable IP cores, designed for *system-on-chip* (SOC) development. The IP cores are centered around a common on-chip bus, and use a coherent method for simulation and synthesis. The library is vendor independent, with support for different CAD tools and target technologies. A unique plug&play method is used to configure and connect the IP cores without the need to modify any global resources.

1.4 Library organization

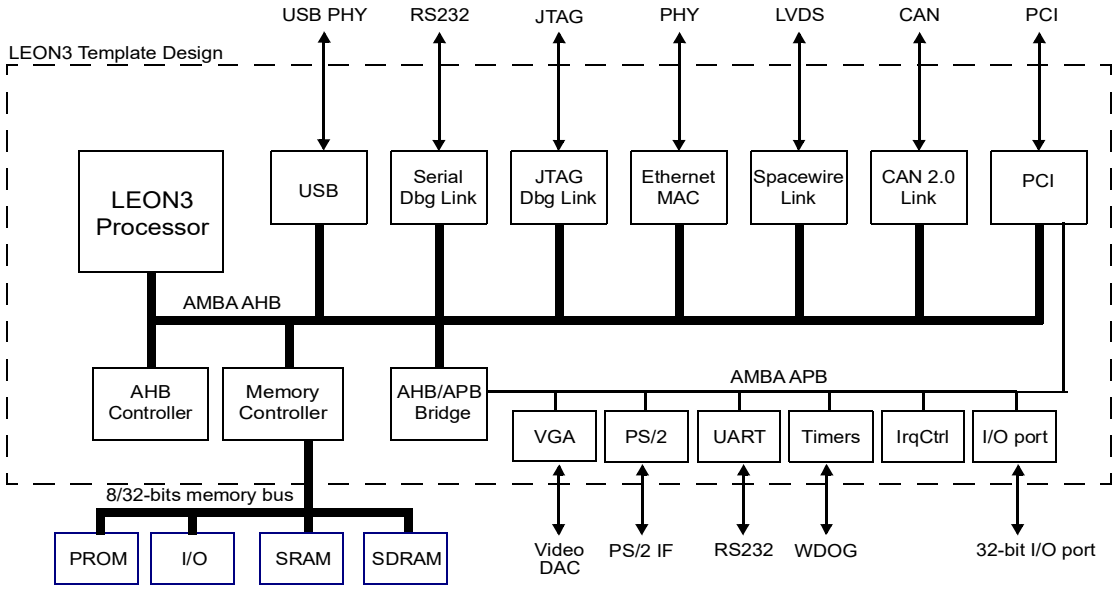
GRLIB is organized around VHDL libraries, where each major IP (or IP vendor) is assigned a unique library name. Using separate libraries avoids name clashes between IP cores and hides unnecessary implementation details from the end user. Each VHDL library typically contains a number of packages, declaring the exported IP cores and their interface types. Simulation and synthesis scripts are created automatically by a global makefile. Adding and removing of libraries and packages can be made without modifying any global files, ensuring that modification of one vendor's library will not affect other vendors. A few global libraries are provided to define shared data structures and utility functions.

GRLIB provides automatic script generators for the Modelsim, Ncsim, Aldec, Sonata and GHDL simulators, and the Synopsys, Synplify, Cadence, Mentor, Actel, Altera, Lattice, eASIC and Xilinx implementation tools. Support for other CAD tools can be easily be added.

1.5 On-chip bus

The GRLIB is designed to be 'bus-centric', i.e. it is assumed that most of the IP cores will be connected through an on-chip bus. The AMBA-2.0 AHB/APB bus has been selected as the common on-chip bus, due to its market dominance (ARM processors) and because it is well documented and can be used for free without license restrictions. The figure below shows an example of a LEON3 system

designed with GRLIB. The same concept is used for the complete LEON and NOEL lines of processors.



1.6 Distributed address decoding

Adding an IP core to the AHB bus is unfortunately not as straight-forward as just connecting the bus signals. The address decoding of AHB is centralized, and a shared address decoder and bus multiplexer must be modified each time an IP core is added or removed. To avoid dependencies on a global resource, distributed address decoding has been added to the GRLIB cores and AMBA AHB/APB controllers.

1.7 Interrupt steering

GRLIB provides a unified interrupt handling scheme by adding 32 interrupt signals to the AHB and APB buses. An AMBA module can drive any of the interrupts, and the unit that implements the interrupt controller can monitor the combined interrupt vector and generate the appropriate processor interrupt. In this way, interrupts can be generated regardless of which processor or interrupt controller is being used in the system, and does not need to be explicitly routed to a global resource. The scheme allows interrupts to be shared by several cores and resolved by software.

1.8 Plug&Play capability

A broad interpretation of the term ‘plug&play’ is the capability to detect the system hardware configuration through software. Such capability makes it possible to use software application or operating systems which automatically configure themselves to match the underlying hardware. This greatly simplifies the development of software applications, since they do not need to be customized for each particular hardware configuration.

In GRLIB, the plug&play information consists of three items: a unique IP core ID, AHB/APB memory mapping, and used interrupt vector. This information is sent as a constant vector to the bus arbiter/decoder, where it is mapped on a small read-only area in the top of the address space. Any AHB master can read the system configuration using standard bus cycles, and a plug&play operating system can be supported.

To provide the plug&play information from the AMBA units in a harmonized way, a configuration record for AMBA devices has been defined (figure 1). The configuration record consists of 8 32-bit words, where four contain configuration words defining the core type and interrupt routing, and four contain so called ‘bank address registers’ (BAR), defining the memory mapping.

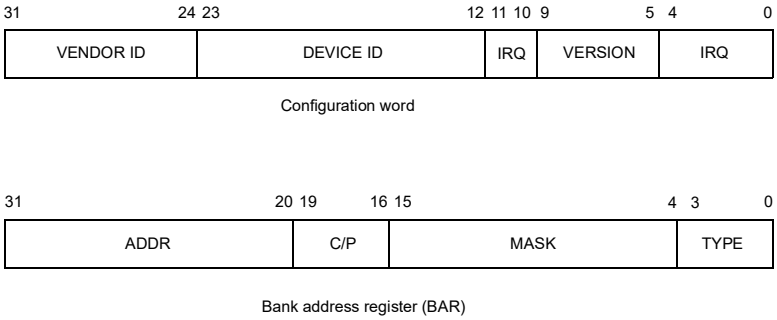


Figure 1. AMBA configuration record

The configuration word for each device includes a vendor ID, device ID, version number, and interrupt routing information. The BARs contain the start address for an area allocated to the device, a mask defining the size of the area, information whether the area is cacheable or pre-fetchable, and a type declaration identifying the area as an AHB memory bank, AHB I/O bank or APB I/O bank. The configuration record can contain up to four BARs and the core can thus be mapped on up to four distinct address areas.

1.9 Portability

GRLIB is designed to be technology independent, and easily implemented on both ASIC and FPGA technologies. Portability support is provided for components such as single-port RAM, two-port RAM, dual-port RAM, single-port ROM, clock generators and pads. The portability is implemented by means of virtual components with a VHDL generic to select the target technology. In the architecture of the component, VHDL generate statements are used to instantiate the corresponding macro cell from the selected technology library. For RAM cells, generics are also used to specify the address and data widths, and the number of ports.

1.10 Available IP cores

Please see the GRLIB IP Core User's Manual (GRIP, grip.pdf) for a list of IP cores included in the library.

1.11 Versions

A GRLIB release is identified by the name grib-*type*-*yyyy.q*-b*buildid.* The fields have the following meaning:

type - This describes the type of GRLIB distribution. The main types are com, ft-fpga, gpl and ft. The different distributions contain a different basic set of IP cores. The FT distributions contain support for enabling fault-tolerance features.

yyyy.q - This is a version number showing the year and the quarter of the release. This replaces the old version numbering system as of 2017.

buildid - This is the main identifier for the version of the IP cores. The build ID is incremented whenever a new GRLIB release is made that has changes to the IP cores. The build ID is also included in the system's plug&play information. The build ID may be used by software drivers to detect presence of features or to implement workarounds and should not be changed.

As described in section 1.8, the Plug&Play information also contains a version field for each IP core. This version field is typically updated when there are changes to the register interface or new features added. This is intended as an aid to software drivers. The main identifier for IP core version is the library build ID.

1.12 Licensing

The main infra-structure of GRLIB is released in open-source under the GNU GPL license. This means that designs based on the GPL version of GRLIB must be distributed in full source code under the same license. For commercial applications where source-code distribution is not desirable or possible, Frontgrade Gaisler offers low-cost commercial IP licenses. Contact sales@gaisler.com for more information or visit http://www.gaisler.com/.

2 Installation

2.1 Installation

GRLIB is distributed as a gzipped tar-file and can be installed in any location on the host system:

```
gunzip -c grlib-com-yyyy.q-bxxxx.tar.gz | tar xf -

or

tar xvf grlib-com-yyyy.q-bxxxx.tar.gz
```

NOTE: Do NOT use WinZip on the .tar.gz file, this will corrupt the files during extraction!

The distribution has the following file hierarchy:

```
bin                 various scripts and tool support files
boards              support files for FPGA prototyping boards
designs             template designs
doc                 documentation
lib                 VHDL libraries
netlists            Vendor specific mapped netlists
software            software utilities and test benches
verification        test benches
```

GRLIB uses the GNU ‘make’ utility to generate scripts and to compile and synthesis designs. It must therefore be installed on a UNIX system or in a ‘UNIX-like’ environment. Tested hosts systems are Linux and Windows with Cygwin.

2.2 Upgrading

When migrating from earlier GRLIB releases the steps below should be followed in order to minimize the number of possible conflicts when upgrading:

- The new package should be extracted in its own directory. Do not overwrite the existing GRLIB tree with the new package.
- Added designs and IP cores should be copied into the new tree.
- All existing scripts (file lists) should be removed and then re-generated using the appropriate make targets in the new GRLIB tree.
- The Changelog (available in the doc/ directory) of the new library should be studied. Check for new features or VHDL generics that have been deprecated compared to the previous version that was used.

2.3 Directory organization

GRLIB is organized around VHDL libraries, where each IP vendor is assigned a unique library name. Each vendor is also assigned a unique subdirectory under grlib/lib in which all vendor-specific source files and scripts are contained. The vendor-specific directory can contain subdirectories, to allow for further partitioning between IP cores etc.

The basic directories delivered with GRLIB under grlib-*yyyy.q*-b*xxxx*/lib are:

```
grlib               packages with common data types and functions
gaisler             Frontgrade Gaisler's components and utilities
tech/*              target technology libraries for gate level simulation
techmap             wrappers for technology mapping of macro cells (RAM, pads)
work                components and packages in the VHDL work library
```

Other vendor-specific directories are also delivered with GRLIB, but are not necessary for the understanding of the design concept. Libraries and IP cores are described in detail in separate documentation. Many of the tech/* directories are populated by performing simulation library installation. This is described in section 2.5.

2.4 Host platform support

GRLIB is designed to work with a large variety of platforms. The paragraphs below outline the supported environments. Other unix-based environments are likely to work but are not tested. As a baseline, the following host software must be installed for the GRLIB configuration scripts to work:

- Bash shell
- GNU make
- GCC
- Tcl/Tk (8.4 or later)
- patch utility
- X Windows graphical system (required for Tcl/Tk on Cygwin and Linux)

2.4.1 Linux

The make utility and associated scripts should work on most linux distribution. GRLIB is primarily developed on Linux hosts, and GNU/Linux is the preferred platform.

2.4.2 Windows with Cygwin

The make utility and associated scripts will work, although somewhat slow. Note that GCC and the make utility must be selected during the Cygwin installation. Cygwin troubleshooting:

- Some versions of Cygwin are known to fail due to a broken 'make' utility. In this case, try to use a different version of Cygwin or update to a newer make.
- Make sure that the paths to tools are set-up properly. For instance, for Xilinx ISE tools the *XILINX* environment variable must point at the installation of ISE. This can be checked in the Cygwin shell by typing **echo $XILINX**, which should lead to a print-out matching the Xilinx ISE installation. Example: *c:\Xilinx\13.2\ISE_DS\ISE* (path depends on ISE version and selected installation point) can be set from the Cygwin shell with the command:
export XILINX=c:\\Xilinx\\13.2\\ISE_DS\\ISE
- Paths to the EDA tools must be included in the *PATH* variable. It must be possible to invoke the tools by ussing their command on the Cygwin command line. For Xilinx tools, this can be tested by issuing a command such as **par**, which should result in the help text for Xilinx's place&route tool to be printed. If this does not work then the *PATH* variable must be set. Examples:
export PATH=$PATH:$XILINX/bin/nt
or
export PATH=$PATH:/cygdrive/Xilinx/13.2/ISE_DS/ISE/bin/nt
- In order to run the graphical configuration tools that come with GRLIB it is also required to install an X-server. The recommended option is the Cygwin X-server. The following packages shall also be selected using the Cygwin installer: xorg-server, xorg-server-common, xinit and all the packages found when filtering by "X11" in the search field. Inability to run the graphical tools or to work normally with them is very likely caused by a missing package in Cygwin. Another option is to install Tcl/Tk packages from another provider, such as ActiveState.
- With Cygwin's X server installed, the server should be started via the start menus's *Cygwin-X > XWin Server*. With the default setting this will bring up a terminal window with the proper initialization of the *DISPLAY* variable. In other terminal windows, the *DISPLAY* variable can be set with **export DISPLAY=:0**.
- In case **make xconfig** fails, try removing the file *lconfig.tk* from the template design directory. Then issue **make distclean** followed by **make xconfig**.
- It is recommended to extract the GRLIB file tree in your Cygwin user's home directory. Otherwise files may be generated in the wrong format (binary vs. text). See *http://cygwin.com/cygwin-ug-net/using-textbinary.html* for additional information.
- Tools, such as ModelSim, may generate Makefiles that contain paths with the character ':' in them. This will then lead to build failures. The GRLIB scripts attempt to detect and patch the generated Makefiles to avoid these failures. If you encounter errors such as *"*** No rule to make target ..."* then please send the file *make.work* from the template design directory together with the error output to support@gaisler.com. (NOTE: generating scripts under MSYS may not work and is NOT supported).
- For error errors involving *fork*, please see http://cygwin.com/faq-nochunks.html#faq.using.fixing-fork-failures.

- Cygwin sets the TZ variable. This variable must be set so that it corresponds to the timezone used by your license server. Otherwise you may experience problems with software such as Synplify. The issue can be resolved by issuing the command **unset TZ**.
- Some programs may generate files with DOS line endings "**\r\n**" which can cause incorrect behavior of the Cygwin shell (for example in for-loops over space separated strings). This issue can be resolved by running the command **set -o igncr** followed by **export SHELLOPTS**.

2.5 Installation of simulation libraries

Simulation libraries need to be installed to allow simulation of most template designs included in GRLIB. The simulation libraries are typically copied from the vendor EDA tool installation into GRLIB and can then be used with all the simulation tools. Some designs instead rely on prebuilt libraries, in this case it is documented in the design's README.txt file.

The descriptions in the subsections below install the simulation libraries globally for GRLIB. The steps only have to be performed once and it will apply to all designs. The commands described below can be performed from the root of the GRLIB tree if the variable **$GRLIB** has been set to point to the GRLIB base. Example:

export GRLIB=/home/user/grlib-com-2019.2-b4241

The commands can also be executed from within any template design directory under designs/.

2.5.1 Installation of Altera libraries

Altera libraries are copied from a Quartus II installation. The variable **$QUARTUS_ROOTDIR** needs to be set (note that it needs to include the quartus installation directory). Example:

export QUARTUS_ROOTDIR=/usr/local/altera/quartus13.1/quartus/

The Altera libraries are then installed with the command: **make install-altera**

Later version of Quartus may have discontinued support for some devices and the corresponding simulation libraries are then missing. This is reported by the installation script. For example, using Quartus II 13.1 the result will be:

```
bash-4.1$ make install-altera
installing tech/altera
installing tech/altera_mf
installing tech/cycloneiii
skipping tech/stratixii - not supported by Quartus II version
installing tech/stratixii
Altera library installation completed.
```

Using Quartus II 14.1 the result will be:

```
bash-4.1$ make install-altera
installing tech/altera
installing tech/altera_mf
skipping tech/cycloneiii - not supported by Quartus II version
skipping tech/stratixii - not supported by Quartus II version
skipping tech/stratixiii - not supported by Quartus II version
Altera library installation completed.
```

2.5.2 Installation of Microsemi libraries

Note: The GPL version of GRLIB does not support Microsemi devices.

Microsemi libraries are copied from a Libero IDE or Libero SoC installation. The variable **$LIBERO_ROOTDIR** needs to be set. Example:

export LIBERO_ROOTDIR=/usr/local/microsemi/Libero_SoC_v11.9

or (on Windows/Cygwin):

export LIBERO_ROOTDIR=/cygdrive/c/Micosemi/Libero_SoC_v11.9

The Microsemi libraries are then installed with the command: **make install-microsemi**

Libero SoC cannot be used for AX and RTAX devices. If the installation is performed with Libero SoC then it is expected that some Libraries are skipped. The same applies for Libero IDE that does not support new technologies.

The **make install-microsemi** command installs both source and pre-compiled versions of the simulation libraries. The source versions are patched when installed.

2.5.3 Installation of Xilinx libraries

Simulation libraries for Xilinx designs are installed in two different ways, depending on the target technology. For Xilinx devices built with Xilinx ISE, the base set of libraries are taken from a Xilinx ISE installation. The variable **$XILINX** needs to be set like it is from the ISE initialization scripts. Example:

export XILINX=/usr/local/xilinx/14.7/ISE_DS/ISE

The UNISIM libraries are then installed with the command: **make install-unisim**

In the case of designs targeting Xilinx 7-Series or Ultrascale/Ultrascale+ devices, a different approach needs to be followed, since the simulation libraries for these technologies are not included in Xilinx ISE and can not be simply copied from the installation. The variable **$XILINX** needs to be set like it is from the Vivado initialization scripts. Example:

export XILINX=/usr/local/xilinx/Vivado/2018.1/data

It is then possible to install the UNISIM and other libraries using the command:

make map_xilinx_7series_lib

This command will first use Xilinx Vivado to compile the simulation libraries and then map the libraries, e.g. in modelsim.ini.

Note: Take care to check the compatibility between the Xilinx Vivado version and the simulator version, since the compilation of the simulation libraries may fail if incompatible versions are used. Setting **GRLIB_VIVADO_COMPILE_VERBOSE** allows debugging of the compilation by removing the quiet flag from the compilation command.

Please also note that the variable **SKIP_SIM_TECHLIBS** needs to be set to 1 in the Makefile of the targeted design to avoid the script environment from including any simulation libraries that might have been installed using **make install-unisim**.

2.5.4 Installation of DARE+ libraries

Note: Only the FT versions of GRLIB support the DARE+ library.

DARE+ ASIC libraries version 5.x are copied from a DARE+ ASIC installation. The variable **$DARE_ROOTDIR** needs to be set. Example:

export DARE_ROOTDIR=/usr/local/dare/DesignKit_V5.5

The DARE+ libraries are then installed with the command: **make install-dare**

For DARE+ library simulation models to be included in the simulation the **make install-dare** needs to be performed before simulation scripts are created.

2.5.5 Installation of NanoXplore libraries

The NanoXplore simulation library for Modelsim is taken from the NanoXmap installation directory.

It is necessary to set the variable **NANOXLIBPATH** to the directory that contains nxLibrary.vhdp. For example:

export NANOXLIBPATH = /home/nanoxplore/nxmap-22.1.0.1/share/modelsim

The NanoXplore libraries are then installed with the command: **make install-nxlibrary**

Note that the simulation libraries must be installed before simulation scripts are created.

The library is removed from GRLIB using the command **make remove-nxlibrary**

2.5.6 Installation of Lattice Radiant libraries

The Lattice Radiant simulation library (Nexus family) for Modelsim can be automatically built using the make system provided and it uses files from Radiant installation folder. Right now, only simulations using ModelSim are supported.

It is necessary to set the variable **GRLIB_LATTICE_RADIANT** with the path to the Radiant installation folder. Example:

export GRLIB_LATTICE_RADIANT=/home/lscc/radiant/3.2

Also you need to set the variable **GRLIB_LATTICE_RADIANT_SIM_DEVICE** that indicates the nexus library you want to compile. Values that we support are **lifcl** (for Crosslink-NX and Certus-NX) and **lfcpnx** (CertusPro-NX). For example, for compiling libraries for Crosslink-NX (library lifcl):

export GRLIB_LATTICE_RADIANT_SIM_DEVICE=lifcl

This variable is usually set inside the local Makefile found in the design folder.
Please note that, as of Radiant 3.2, lifcl (Crosslink-NX) and lfd2nx (Certus-NX) are equivalent libraries. Internal Radiant scripts actually compile lfd2nx as lifcl. To keep the make system simple, the user is urged to use lifcl as value for **GRLIB_LATTICE_RADIANT_SIM_DEVICE** even in the case of designs targeting Certus-NX FPGAs.

To then compile the library, the following target needs to be run: **make install-radiant-simlibs**. This step can easily take 20 minutes.

Before simulating, the library needs to be mapped in the local modelsim.ini by issuing: **make map-radiant-simlibs**. If the library is not already compiled, the make system will take care of it before the mapping. The compiled library gets always mapped as **nexus_sim**: this allows us to be library-independent in out internal Lattice techmap.
Lattice IPs that are included in the design through the **LATTICE_IP** variable, are automatically generated when running **make map-radiant-simlibs**, which takes care of mapping the lattice libraries for the IP verilog file.

The compiled library is removed from GRLIB using the command: **make remove-radiant-simlibs**

These make targets are working only inside a Linux environment, while they won't work inside Windows/Cygwin.

3 LEON/GRLIB quick-start guide

3.1 Introduction

This chapter will provide a simple quick-start guide on how to implement a LEON3 or LEON4 system using GRLIB, and how to download and run software on the target system. Refer to chapters 4 - 7 for a deeper understanding of the GRLIB organization.

The template designs in GRLIB are typically named as *<processor>-<board vendor>-<board>*, where *<processor>* is either leon3, for LEON3/4 designs, leon5, for LEON5 designs, or noelv, for NOEL-V designs. A subset of the template designs instantiate a processor subsystem that can be configured to either include LEON3 or LEON4 (if included in GRLIB). This subsystem is named *gaisler.leon_dsu_stat_base* and is further described in the *LEON/GRLIB Configuration and Development Guide (guide.pdf)*. LEON5 and NOEL-V are both designed to be instantiated as parts of a dedicated subsystem, named *leon5sys* and *noelvsys*, respectivelyThe subsections below focus on LEON3 and are also applicable for LEON4, LEON5 and NOEL-V.

3.2 Overview

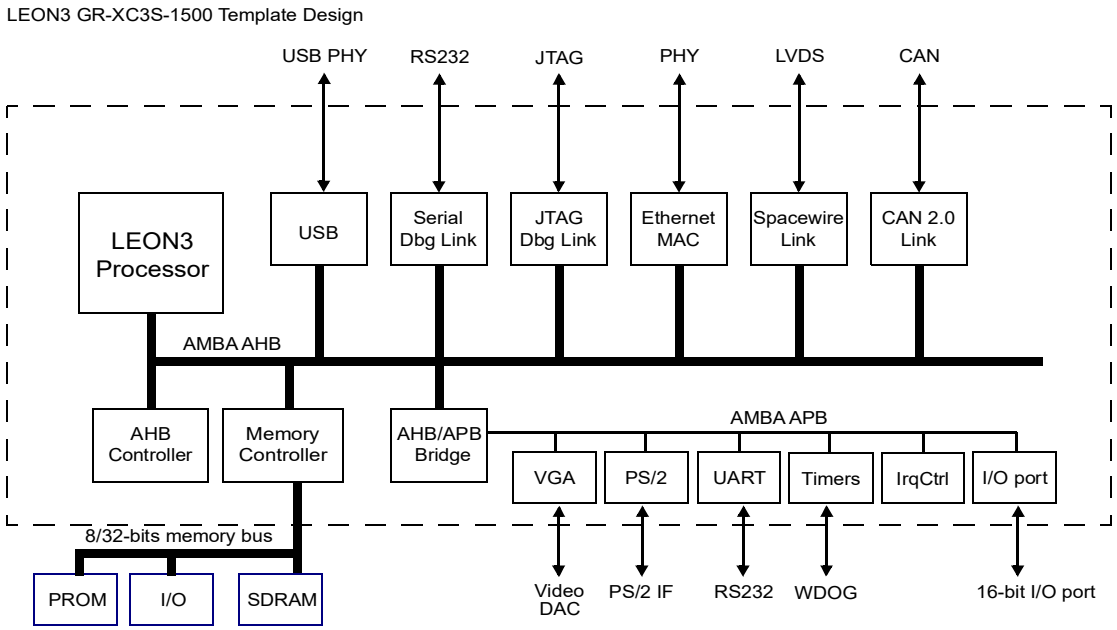
Implementing a LEON3 system is typically done using one of the template designs on the designs directory. For this tutorial, we will use the LEON3 template design for the GR-XC3S-1500 board. Implementation is typically done in three basic steps:

- Configuration of the design using xconfig
- Simulation of design and test bench
- Synthesis and place&route

The template design is located in `designs/leon3-gr-xc3s-1500`, and is based on three files:

- *config.vhd* - a VHDL package containing design configuration parameters. Automatically generated by the xconfig GUI tool.
- *leon3mp.vhd* - contains the top level entity and instantiates all on-chip IP cores. It uses config.vhd to configure the instantiated IP cores.
- *testbench.vhd* - test bench with external memory, emulating the GR-XC3S-1500 board.

Each core in the template design is configurable using VHDL generics. The value of these generics is assigned from the constants declared in config.vhd, created with the xconfig GUI tool.



3.3 Configuration

Change directory to designs/leon3-gr-xc3s-1500, and issue the command 'make xconfig' in a bash shell (linux) or cygwin shell (windows). This will launch the xconfig GUI tool that can be used to modify the leon3 template design. When the configuration is saved and xconfig is exited, the config.vhd is automatically updated with the selected configuration.

3.4 Simulation

The template design can be simulated in a test bench that emulates the prototype board. The test bench includes external PROM and SDRAM which are pre-loaded with a test program. The test program will execute on the LEON3 processor, and tests various functionality in the design. The test program will print diagnostics on the simulator console during the execution.

The following command should be give to compile and simulate the template design and test bench using Mentor ModelSim/QuestaSim or Aldec Riviera-PRO (simulator is selected based in the GRLIB_SIMULATOR environment variable, default is ModelSim):

```
make sim
make sim-launch
```

Make targets also exist for other simulators. See documentation of tools in this document or issue *make help* to view a list of available targets.

Some designs require that the environment variable GRLIB_SIMULATOR is set to the simulator to use in order for all parts of the design to be built correctly (in particular template designs for Xilinx devices that make use of the Xilinx MIG). Refer to the design's README.txt file and section 4.3 of this document for additional information.

A typical simulation log can be seen below.

```
$ make sim-run

VSIM 1> run -a
# LEON3 GR-XC3S-1500 Demonstration design
# GRLIB Version 1.0.15, build 2183
# Target technology: spartan3 ,  memory library: spartan3
# ahbctrl: AHB arbiter/multiplexer rev 1
# ahbctrl: Common I/O area disabled
# ahbctrl: AHB masters: 4, AHB slaves: 8
# ahbctrl: Configuration area at 0xfffff000, 4 kbyte
# ahbctrl: mst0: Frontgrade Gaisler        Leon3 SPARC V8 Processor
# ahbctrl: mst1: Frontgrade Gaisler        JTAG Debug Link
# ahbctrl: mst2: Frontgrade Gaisler        SpaceWire Serial Link
# ahbctrl: mst3: Frontgrade Gaisler        SpaceWire Serial Link
# ahbctrl: slv0: European Space Agency   Leon2 Memory Controller
# ahbctrl:       memory at 0x00000000, size 512 Mbyte, cacheable, prefetch
# ahbctrl:       memory at 0x20000000, size 512 Mbyte
# ahbctrl:       memory at 0x40000000, size 1024 Mbyte, cacheable, prefetch
# ahbctrl: slv1: Frontgrade Gaisler        AHB/APB Bridge
# ahbctrl:       memory at 0x80000000, size 1 Mbyte
# ahbctrl: slv2: Frontgrade Gaisler        Leon3 Debug Support Unit
# ahbctrl:       memory at 0x90000000, size 256 Mbyte
# apbctrl: APB Bridge at 0x80000000 rev 1
# apbctrl: slv0: European Space Agency   Leon2 Memory Controller
# apbctrl:       I/O ports at 0x80000000, size 256 byte
# apbctrl: slv1: Frontgrade Gaisler        Generic UART
# apbctrl:       I/O ports at 0x80000100, size 256 byte
# apbctrl: slv2: Frontgrade Gaisler        Multi-processor Interrupt Ctrl.
# apbctrl:       I/O ports at 0x80000200, size 256 byte
# apbctrl: slv3: Frontgrade Gaisler        Modular Timer Unit
# apbctrl:       I/O ports at 0x80000300, size 256 byte
# apbctrl: slv8: Frontgrade Gaisler        General Purpose I/O port
# apbctrl:       I/O ports at 0x80000800, size 256 byte
# apbctrl: slv12: Frontgrade Gaisler        SpaceWire Serial Link
# apbctrl:       I/O ports at 0x80000c00, size 256 byte
# apbctrl: slv13: Frontgrade Gaisler        SpaceWire Serial Link
# apbctrl:       I/O ports at 0x80000d00, size 256 byte
```

```
# grspw13: Spacewire link rev 0, AHB fifos 2x64 bytes, rx fifo 16 bytes, irq 11
# grspw12: Spacewire link rev 0, AHB fifos 2x64 bytes, rx fifo 16 bytes, irq 10
# grgpio8: 18-bit GPIO Unit rev 0
# gptimer3: GR Timer Unit rev 0, 8-bit scaler, 2 32-bit timers, irq 8
# irqmp: Multi-processor Interrupt Controller rev 3, #cpu 1
# apbuart1: Generic UART rev 1, fifo 1, irq 2
# ahbjtag AHB Debug JTAG rev 0
# dsu3_2: LEON3 Debug support unit + AHB Trace Buffer, 2 kbytes
# leon3_0: LEON3 SPARC V8 processor rev 0
# leon3_0: icache 1*8 kbyte, dcache 1*4 kbyte
# clkgen_spartan3e: spartan3/e sdram/pci clock generator, version 1
# clkgen_spartan3e: Frequency 50000 KHz, DCM divisor 4/5
#
# **** GRLIB system test starting ****
# Leon3 SPARC V8 Processor
#   CPU#0 register file
#   CPU#0 multiplier
#   CPU#0 radix-2 divider
#   CPU#0 floating-point unit
#   CPU#0 cache system
# Multi-processor Interrupt Ctrl.
# Generic UART
# Modular Timer Unit
#   timer 1
#   timer 2
#   chain mode
# Test passed, halting with IU error mode
# ** Failure: *** IU in error mode, simulation halted ***
#    Time: 1104788 ns  Iteration: 0  Process: /testbench/iuerr File: testbench.vhd
# Stopped at testbench.vhd line 338
VSIM 2>
```

The test program executed by the test bench consists of two parts, a simple PROM boot loader (prom.S) and the test program itself (*systest.c*). Both parts can be re-compiled using the *make soft* command. This requires that the BCC 1.0.x tool-chain is installed on the host computer. The BCC 1.0.x tool-chain by default includes AMBA plug&play scanning routines that are able to scan over AHB bridges. This is seldom required for system tests and simulation time is decreased by the default assignment of the environment variable *LDFLAGS* to *LDFLAGS=-qnoambapp*. The default assignment can be avoided by defining the *LDFLAGS* variable.

The simple PROM boot loader (i) contains code to initialize the processor, memory controller and other peripherals. If the file *prom.S* is missing from the template design folder then a default version located at *software/leon3/prom.S* will be used. Configuration constants used by *prom.S* are located in the file *prom.h*. If the memory controller in a design is changed, or the base address of main memory is moved, then *prom.h* and possibly *prom.S* may need to be updated to correctly initialize the new configuration. If the template design contains a link script or compiler flags in the Makefile these may also need to be updated when the memory is moved. If *prom.h* or *prom.S* are modified then *make soft* is required before the changes take effect.

The boot loader is designed for simulation only.

Note that the simulation is terminated by generating a VHDL failure, which is the only way of stopping the simulation from inside the model. An error message is then printed:

```
# Test passed, halting with IU error mode
# ** Failure: *** IU in error mode, simulation halted ***
#    Time: 1104788 ns  Iteration: 0  Process: /testbench/iuerr File: testbench.vhd
# Stopped at testbench.vhd line 338
```

This error can be ignored.

3.5 Synthesis and place&route

The template design can be synthesized with either Synplify, Precision or ISE/XST. Synthesis can be done in batch or interactively. To use Synplify in batch mode, use the command:

```
make synplify
```

To use Synplify interactively, use:

```
make synplify-launch
```

The corresponding command for ISE are:

```
make ise-map
```

and

```
make ise-launch
```

To perform place&route for a netlist generated with synplify, use:

```
make ise-synp
```

For a netlist generated with XST, use:

```
make ise
```

In both cases, the final programming file will be called ‘leon3mp.bit’. See the GRLIB User’s Manual chapter 3 for details on simulation and synthesis script files.

3.6 Simulation of post-synthesis netlist

If desired, it is possible to simulate the synthesized netlist in the test bench. The synplify synthesis tool generates a VHDL netlist in the file synplify/leon3mp.vhm. To re-run the test bench with the netlist, do as follows:

```
vcom synplify/leon3mp.vhm
vsim -c testbench
vsim> run -all
```

3.7 Board re-programming

The GR-XC3S-1500 FPGA configuration PROMs can be programmed from the shell window with the following command:

```
make ise-prog-prom
```

For interactive programming, use Xilinx Impact software. See the GR-XC3S-1500 Manual for details on which configuration PROMs to specify.

A pre-compiled FPGA bit file is provided in the bitfiles directory, and the board can be re-programmed with this bit file using:

```
make ise-prog-prom-ref
```

3.8 Running applications on target

To download and debug applications on the target board, the GRMON debug monitor is used. GRMON can be connected to the target using RS232, JTAG, ethernet, USB, PCI or SpaceWire. The most convenient way is probably to use JTAG.

Please refer to the GRMON2 User’s Manual for a description of the GRMON2 operations. The output below is an example of GRMON output after connecting to a system:

```
initialising ..............
 detected frequency:  40 MHz

 Component                            Vendor
 LEON3 SPARC V8 Processor             Frontgrade Gaisler
 AHB Debug UART                       Frontgrade Gaisler
 AHB Debug JTAG TAP                   Frontgrade Gaisler
 SVGA frame buffer                    Frontgrade Gaisler
 GR Ethernet MAC                      Frontgrade Gaisler
 AHB ROM                              Frontgrade Gaisler
 AHB/APB Bridge                       Frontgrade Gaisler
 LEON3 Debug Support Unit             Frontgrade Gaisler
 DDR266 Controller                    Frontgrade Gaisler
 Generic APB UART                     Frontgrade Gaisler
 Multi-processor Interrupt Ctrl       Frontgrade Gaisler
 Modular Timer Unit                   Frontgrade Gaisler
```

```
 Keyboard PS/2 interface              Frontgrade Gaisler
 Keyboard PS/2 interface              Frontgrade Gaisler
```

To download an application, use the ‘load’ command. To run it, use ‘run’ :

```
load stanford.exe
run
```

The console output will occur in the grmon window if grmon was started with -u, otherwise it will be send to the RS232 connector of the board.

3.9 Flash PROM programming

The GR-XC3S-1500 board has a 64 Mbit (8Mx8) Intel flash PROM for LEON3 application software. A PROM image is typically created with the MKPROM2 utility that can be downloaded from http://www.gaisler.com.

Once the PROM image has been created, the on-board flash PROM can be programmed through GRMON. The procedure is described in the GRMON manual, below is the required GRMON command sequence:

```
flash erase all
flash load prom.out
```

3.10 Software development

The LEON and NOEL line of processors are supported by several free software tool chains:

- Bare-C cross-compiler system (BCC)
- RTEMS cross-compiler system (RCC)
- Linuxbuild embedded linux

All these tool chains and associated documentation can be downloaded from www.gaisler.com.

In addition, both LEON and NOEL are supported by several commercial alternatives. Please contact Frontgrade Gaisler for additional information or see http://www.gaisler.com.

4 Implementation flow

4.1 Introduction

The following sections will describe how simulation and synthesis is performed using the GRLIB make system. It is recommended to try out the various commands on one of the template designs, such as designs/leon3mp.

4.2 Using Makefiles and generating scripts

GRLIB consists of a set of VHDL libraries from which IP cores are instantiated into a local design. GRLIB can be installed in a in a global location (such as on a network share that is used by several designers) and be used in read-only mode. Note that for some technologies it is possible to install vendor specific libraries into the GRLIB tree. In this case, write permission is required for the user that performs the library install.

All compilation, simulation and synthesis is done in a local design directory, using tool-specific scripts. The GRLIB IP cores (components) are instantiated in the local design by the inclusion of various GRLIB packages, declaring the components and associated data types.

A design typically contains of one or more VHDL files, and a local makefile:

```
bash$ ls -g mydesign
-rw-r--r--    1 users        1776 May 25 10:37 Makefile
-rw-r--r--    1 users       12406 May 25 10:46 mydesign.vhd
```

The GRLIB files are accessed through the environment variable GRLIB. This variable can either be set in the local shell or in a local makefile, since the 'make' utility is used to automate various common tasks. A GRLIB-specific makefile is located in bin/Makefile. To avoid having to specify the GRLIB makefile using the -f option, the local makefile should includes the GRLIB makefile:

```
GRLIB=../../grlib
include $(GRLIB)/bin/Makefile
```

Running 'make help' with this makefile will print a short menu:

```
$ make help

installation targets:    : see doc/grlib.pdf for library installation targets

 interactive targets:
  simulation:
   make avhdl-launch         : start Active-HDL GUI mode
   make avhdl                : compile design using Active-HDL gui mode
   make ncsim-launch         : start NCSim GUI
   make riviera-launch       : start RivieraPRO
   make vsim-launch          : start ModelSim/QuestaSim
   make vcs-launch           : start VCS
  verification:
   make alint-launch         : start Alint elaboration time linting
   make fpro-launch          : start FormalPro GUI
  implementation:
   make actel-launch         : start Actel Designer for current project
   make ise-launch           : start ISE project navigator for XST project
   make ise-launch-synp      : start ISE project navigator for synplify project
   make libero-launch        : start Microsemi Libero
   make planahead-launch     : start PlanAhead project navigator
   make quartus-launch       : start Quartus for current project
   make quartus-launch-synp  : start Quartus for synplify project
   make synplify-launch      : start Synplify
   make vivado-launch        : start Vivado project navigator
   make nanoxmap-launch      : start NanoXmap GUI
  other GRLIB targets:
   make xgrlib               : start GRLIB GUI

 batch targets:
  simulation:
   make vsimsa          : compile design using Active-HDL batch mode
   make riviera         : compile design using riviera
   make vsim            : compile design using modelsim
   make ncsim           : compile design using ncsim
   make ghdl            : compile design using GHDL
   make vcs-elab        : compile and elaborate design using VCS
  verification:
   make alint-comp      : alint compilation time linting
   make fm              : Formal equivalence check using Synopsys Formality
```

```
implementation:
 make actel           : synthesize with synplify, place&route Actel Designer
 make dc              : synthesize design usign Synopsys Design Compiler
 make ise             : synthesize and place&route with Xilinx ISE
 make ise-map         : synthesize design using Xilinx XST
 make ise-prec        : synthesize with precision, place&route with Xilinx ISE
 make ise-synp        : synthesize with synplify, place&route with Xilinx ISE
 make isp-synp        : synthesize with synplify, place&route with ISPLever
 make libero          : syntiesize, place&route and generate bit file with Microsemi Libero
 make planahead       : synthesize and place&route with Xilinx PlanAhead
 make precision       : synthesize design using precision
 make quartus         : synthesize and place&route using Quartus
 make quartus-map     : synthesize design using Quartus
 make quartus-synp    : synthesize with synplify, place&route with Quartus
 make synplify        : synthesize design using synplify
 make vivado          : synthesize and place&route with Xilinx Vivado
 make nanoxpython     : synthesize and place&route with NanoXplore NanoXmap
other GRLIB targets:
 make scripts         : generate compile scripts only
 make clean           : remove all temporary files
 make distclean       : remove all temporary files
```

Generating tool-specific compile scripts can be done as follows:

```
$ make scripts
$ ls compile.*
compile.dc  compile.ncsim  compile.synp  compile.vsim  compile.xst compile.ghdl
```

The local makefile is primarily used to generate tool-specific compile scripts and project files, but can also be used to compile and synthesize the current design. To do this, additional settings in the makefile are needed. The makefile in the design template grlib/designs/leon3mp can be seen as an example:

```
$ cd grlib/designs/leon3mp
$ cat Makefile
GRLIB=../..
TOP=leon3mp
BOARD=gr-pci-xc2v
include $(GRLIB)/boards/$(BOARD)/Makefile.inc
DEVICE=$(PART)-$(PACKAGE)$(SPEED)
UCF=$(GRLIB)/boards/$(BOARD)/$(TOP).ucf
QSF=$(BOARD).qsf
EFFORT=1
VHDLSYNFILES=config.vhd leon3mp.vhd
VHDLSIMFILES=testbench.vhd
SIMTOP=testbench
SDCFILE=$(GRLIB)/boards/$(BOARD)/default.sdc
BITGEN=$(GRLIB)/boards/$(BOARD)/default.ut
CLEAN=local-clean
include $(GRLIB)/bin/Makefile
```

The table below summarizes the common (target independent) ‘make’ targets:

TABLE 1. Common make targets

Make target	Description
scripts	Generate GRLIB compile scripts for all supported tools
xconfig	Run the graphic configuration tool (leon3/leon5/noelv designs)
clean/distclean	Remove all temporary files
xgrlib	Run the graphical implementation tool (see “XGrlib graphical implementation tool” on page 68)

Simulation, synthesis and place&route of GRLIB designs can also be done using a graphical tool called **xgrlib**. This tool is described further in chapter 4.9.

4.3 File attributes

The GRLIB script environment supports file attributes for the tool project creation to be added in the library structure. Attributes can be added to all files in an entire library through `libs.txt`; a directory through `dirs.txt`; or on single files through `vhdlsyn.txt`, `vhdlsim.txt`, `vlogsyn.txt` or `vlogsim.txt`.

Attributes are added after each other after the library/directory/file name and are separated by space or tab using the following structure:

```
<file> <attribute1>=<value1> <attribute2>=<value2>
```

Example:

```
$ cat lib/grlib/stdlib/vhdlsim.txt
stdio.vhd   vhdlstd=93
testlib.vhd vhdlstd=93
```

Attributes can also be added to project specific files included in the local variables: `VHDLSYNFILES`, `VHDLOPTSYNFILES`, `VHDLSIMFILES`, `VHDLIPFILES`, `VERILOGOPTSYNFILES`, `VERILOGSYNFILES` and `VERILOGSIMFILES`. To add attributes to a file included in these variables, the file name and its attributes should be enclosed in curly brackets.

Example:

```
VHDLSYNFILES=config.vhd {leon3mp.vhd vhdlstd=93}
```

The handling of the attributes depend on the tool. The following file attributes are supported:

TABLE 2. Supported file attributes

Attribute	Description	Values
vhdlstd	Specify the VHDL standard supported by a file	87, 93, 2002, 2008

4.4 Simulating a design

4.4.1 Overview

The ‘make scripts’ command will generate compile scripts and/or project files for the Model/QuestaSim, Riviera, NCsim, Xilinx and gHDL simulators. This is done by scanning GRLIB for simulation files according to the method described in section 8.2. These scripts are then used by further make targets to build and update a GRLIB-based design and its test bench. The local makefile should set the VHDLSYNFILES to contain all synthesizable VHDL files of the local design. Likewise, the VHDLSIMFILES variable should be set to contain all local design files to be used for simulation only. The variable TOP should be set to the name of the top level design entity, and the variable SIMTOP should be set to the name of the top level simulation entity (e.g. the test bench).

```
VHDLSYNFILES=config.vhd ahbrom.vhd leon3mp.vhd
VHDLSIMFILES=testbench.vhd
TOP=leon3mp
SIMTOP=testbench
```

The variables must be set before the GRLIB makefile is included, as in the example above.

All local design files are compiled into the VHDL work library, while the GRLIB cores are compiled into their respective VHDL libraries.

The following simulators are currently supported by GRLIB:

TABLE 3. Supported simulators

Simulator	Comments
GNU VHDL (GHDL)	version 0.25, VHDL only
Aldec Active-HDL	batch and GUI
Aldec Riviera	batch and GUI
Mentor Modelsim version	version 10.6 or later
Cadence NcSim	IUS-5.8-sp3 and later
Xilinx ISIM	ISE-14 or later
Xilinx XSIM	Vivado 2017.3 or later
Synopsys VCS	version 2017.03-SP2 or later

4.4.2 GRLIB_SIMULATOR environment variable

Some designs (including Xilinx 7-series designs and designs that use the Xilinx MIG or other components that require installation of special libraries such as SecureIP or SIMPRIMS) require that external tools are invoked in order to build the simulation libraries. In this case, the GRLIB infrastructure must be made aware of which simulator that will be used. This is done by setting the GRLIB_SIMULATOR variable. Table 4 lists allowed values for GRLIB_SIMULATOR.

TABLE 4. GRLIB_SIMULATOR values

Value	Comment
ALDEC	Aldec Riviera Pro or Aldec ActiveHDL
ALDEC_RWS	Aldec Riviera Pro Workspace (WS) flow, see section 4.8.7.
ModelSim	Mentor ModelSim SE or QuestaSim
ModelSim-PE	ModelSim PE
ModelSim-SE	Alias for ModelSim
Questa	Mentor QuestaSim
Xilinx	Xilinx XSim/ISim

The default value for GRLIB_SIMULATOR is *ModelSim*.

Frontgrade Gaisler AB
Kungsgatan | SE-411 19 | Goteborg | Sweden
+46 31 7758650 | frontgrade.com/gaisler

4.5 Synthesis and place&route

The **make scripts** command will scan the GRLIB files and generate compile and project files for all supported synthesis tools. For this to work, a number of variables must be set in the local makefile:

```
TOP=leon3mp
TECHNOLOGY=virtex2
PART=xc2v3000
PACKAGE=fg676
SPEED=-4
VHDLSYNFILES=config.vhd ahbrom.vhd leon3mp.vhd
SDCFILE=
XSTOPT=-resource_sharing no
DEVICE=xc2v3000-fg676-4
UCF=default.ucf
EFFORT=std
BITGEN=default.ut
```

The TOP variable should be set to the top level entity name to be synthesized. TECHNOLOGY, PART, PACKAGE and SPEED should indicate the target device parameters. VHDLSYNFILES should be set to all local design files that should be used for synthesis. SDCFILE should be set to the (optional) Synplify constraints file, while XSTOPT should indicate additional XST synthesis options. The UCF variable should indicate the Xilinx constraint file, while QSF should indicate the Quartus constraint file. The EFFORT variable indicates the Xilinx place&route effort and the BITGEN variable defines the input script for Xilinx bitfile generation.

The technology related variables are often defined in a makefile include file in the board support packages under GRLIB/boards. When a supported board is targeted, the local makefile can include the board include file to make the design more portable:

```
BOARD=gr-pci-xc2v
include $(GRLIB)/boards/$(BOARD)/Makefile.inc
SDCFILE=$(GRLIB)/boards/$(BOARD)/$(TOP).sdc
UCF=$(GRLIB)/boards/$(BOARD)/$(TOP).ucf
DEVICE=$(PART)-$(PACKAGE)-$(SPEED)
```

The following synthesis tools are currently supported by GRLIB:

TABLE 5. Supported synthesis and place&route tools

Syntesis and place&route tool	Recommended version
Altera Quartus	version 20.1 (Standard and Lite editions)
Cadence RTLC	version 6.1 (GRLIB is not continuously tested with this tool, feedback is appreciated)
Lattice Diamond	version 1.3 (GRLIB is not continuously tested with this tool, feedback is appreciated)
Lattice Radiant	version 3.2.1 and later
Mentor Leonardo Precision	2014 and later
Microsemi Libero*	Libero IDE version 9.2-SP2, Libero SoC 11.9 or later (tested with Libero SoC 12.5)
NanoXplore NanoXmap	NXmap-22.1.0.1 or later
Synopsys DC	2010.12 and later
Synplify	2015.03 and later
Xilinx ISE/XST** ***	version 14.7
Xilinx Vivado	2017.3 (see README.txt in template design) or later
Xilinx PlanAhead	version 14.7

* NOTE: A variable has to be added in the Makefile for Libero SoC 12.0, refer to table 45.
For simulation with Libero SoC 12.0 refer to section 6.6.1

** NOTE: The XST option -use_new_parser yes should NOT be used with GRLIB. The option is known to create bugs in the generated netlist when targeting Virtex-5 (verified with ISE13.2 and 14.7 that produce a design with a malfunctioning LEON3 cache controller).

*** NOTE: XST has been seen to cause issues with GRFPU-Lite. When synthesizing GRFPU-Lite, it is recommended to use Synplify for synthesis.

Note that the batch targets for invoking the synthesis tools typically do not depend on the complete file list. If one of the local design files is modified then the tool will typically be re-run on the whole design. If a design file in a GRLIB library is modified then it may be necessary to run the command 'make distclean' to remove the currently generated files in order to resynthesize the full design using the batch targets.

4.6 Skipping unused libraries, directories and files

GRLIB contains a large amount of files, and creating scripts and compiling models might take some time. To speed up this process, it is possible to skip whole libraries, directories or individual files from being included in the tool scripts. Skipping VHDL libraries is done by defining the constant LIBSKIP in the Makefile of the current design, before the inclusion of the GRLIB global Makefile.

To skip a directory in a library, the variable DIRSKIP should be used. All directories with the defined names will be excluded when the tool scripts are built. In this way, cores which are not used in the current design can be excluded from the scripts. To skip an individual file, the variable FILESKIP should be set to the file(s) that should be skipped. Below is an example from the a template design. All target technology libraries except unisim (Xilinx) are skipped, as well as cores such as PCI, DDR and SpaceWire. Care has to be taken to skip all dependent directories when a library is skipped.

```
LIBSKIP = core1553bbc core1553brm core1553brt gr1553 corePCIF \
          tmtc cypress ihp opencores spw
DIRSKIP = b1553 pcif leon2 leon2ft crypto satcan pci leon3ft ambatest \
                    spacewire ddr can usb ata
FILESKIP = grcan.vhd

include $(GRLIB)/bin/Makefile
```

By default, all technology cells and mapping wrappers are included in the scripts and later compiled. To select only one or a sub-set of technologies, the variable TECHLIBS can be set in the makefile:

```
TECHLIBS = unisim
```

The table below shows which libraries should added to TECHLIBS for each supported technology.

TABLE 6. TECHLIB settings for various target technologies

Technology	TECHLIBS defines
Xilinx (All)	unisim If TECHNOLOGY is set to Virtex2, Virtex4, Spartan3, Spartax3E or Spartan6 then the GRLIB infrastructure will automatically add *virtex* to *TECHLIBS*. *lib/techmap/virtex* contains mappings used for these technologies that depend on UNISIMS components that are not available in later Xilinx tools, such as Vivado.
Altera Stratix-II	altera altera_mf stratixii
Altera Cyclone-III	altera altera_mf cycloneiii
Altera Stratix-III	altera altera_mf stratixiii
Altera others	altera altera_mf
Actel/Microsemi Axcelerator	axcelerator
Actel/Microsemi Axcelerator DSP	axcelerator
Actel/Microsemi Proasic3/e3/3l	proasic3/proasic3e/proasic3l
Actel/Microsemi Fusion	fusion
Actel/Microsemi IGLOO2/Smart-Fusion2	igloo2/smartfusion2

TABLE 6. TECHLIB settings for various target technologies

Technology	TECHLIBS defines
Microsemi RTG4	rtg4
Microsemi Polarfire	polarfire
NanoXplore NG-Medium/NG-Large/NG-Ultra	nx
Lattice	ec
Lattice Certus-NX/CertusPro-NX/CrossLink-NX	nexus
Quicklogic	eclipsee
Atmel ATC18	atc18 virage
Atmel ATC18RHA	atc18rha_cell
eASIC 90 nm	nextreme
eASIC 45 nm	nextreme2
IHP 0.25	ihp25
IHP 0.25 RH	sgb25vrh
Aeroflex 0.25 RH	ut025crh
Aeroflex 0.13 RH	ut130hbd
Ramon 0.18 RH	rh_lib18t
STM C65SPACE	rhs65
UMC 0.18 um	umc18
UMC 0.18 um DARE	dare
TSMC 90 nm	tsmc90

Note that availability of technology mappings for the technologies listed above varies with type of GRLIB distribution. Contact Frontgrade Gaisler for details.

It is also possible to skip compilation of the simulation libraries (located in the *tech/* directory in the GRLIB file tree). This can be useful if prebuilt libraries should used since these may otherwise be overwritten when compiling the full GRLIB file list. In order to skip compilation of simulation libraries set:

```
SKIP_SIM_TECHLIBS=1
```

This will prevent files under *lib/tech/* from being built. Note that technology map files under *lib/techmap* may depend on libraries in *lib/tech/* and that any prebuilt libraries should be mapped before compiling the GRLIB files.

4.7 Encrypted RTL

GRLIB supports encrypted script generation to include encrypted RTL files. The information in this section is applicable if you have purchased GRLIB IP cores that are delivered as encrypted RTL. The open source (GPL) release of GRLIB does not include any encrypted RTL.

There are several different solutions for IP protection available from the EDA vendors. At the time of writing it is not possible to generate one encrypted RTL file that can be used with tools from all vendors. Because of this, encrypted RTL is delivered in several versions. All versions contain the same RTL but in different containers to be used with a specific EDA tool.

Currently the GRLIB script generation supports IP protection (encrypted RTL) for the following tools:

Aldec Riviera-PRO (key ALDEC015_001, for Riviera 2015.06 and later)

Cadence tools supporting Cadence IP protection (proprietary and IEEE-P1735)

Mentor Graphics tools with support for IEEE-P1735 (ModelSim version 6.6+, latest Precision)

Mentor Graphics FormalPro (Linux, tested with version 2015.1)

Microsemi (using key MSL-IP-KEY_RSA)

Synopsys Design Compiler, VCS etc with support for IEEE-P1735

Synopsys Spyglass with support for IEEE-P1735

Synopsys Synplify with support for IEEE-P1735 (version 2012.03 and later)

Xilinx ISE and Vivado

Please contact Frontgrade Gaisler to ensure that your EDA tools are capable of working with GRLIB and encrypted RTL. Specify which tools you will use at the time of order when placing an order for IP cores that are delivered as encrypted RTL.

The RTL source is not available for viewing and simulator views are restricted when using components that are delivered as encrypted RTL.

4.8 Tool-specific usage

4.8.1 GNU VHDL (GHDL)

GHDL is the GNU VHDL compiler/simulator, available from http://ghdl.free.fr/.

The complete GRLIB as well as the local design are compiled by **make ghdl**. The simulation models will be stored locally in a sub-directory (./gnu). A ghdl.path file will be created automatically, containing the proper VHDL library mapping definitions. A sub-sequent invocation of **make ghdl** will re-analyze any outdated files in the WORK library using a makefile created with 'ghdl --gen-makefile'. GRLIB files will not be re-analyzed without a **make ghdl-clean** first.

GHDL creates an executable with the name of the SIMTOP variable. Simulation is started by directly executing the created binary:

```
$ ./testbench
```

TABLE 7. GHDL make targets

Make target	Description
ghdl	Compile or re-analyze local design
ghdl-clean	Remove compiled models and temporary files
ghdl-run	Run test bench in batchmode

TABLE 8. GHDL scripts and files

File	Description
compile.ghdl	Compile script for GRLIB files
make.ghdl	Makefile to rebuild local design
gnu	Directory with compiled models
SIMTOP	Executable simulation model of test bench

The environment variables used by the GHDL flow are:

TABLE 9. Environment variables used by GHDL

Variable	Default value	Description
GRLIB	Set in template design	GRLIB root directory
SIMTOP	Set in template design	Simulation top-level entity
GHDL	ghdl	ghdl executable
GHDLIOPT	(empty)	Arguments supplied to GHDL -i
GHDLMOPT	-fexplicit --ieee=synopsys --mb-comments --warn-no-binding	Arguments supplied to GHDL -m
GHDLRUNOPT	--assert-level=error --ieee-asserts=disable	Arguments supplied when starting compiled SIMTOP binary with ghdl-run

4.8.2 Cadence ncsim

The complete GRLIB as well as the local design are compiled and elaborated in batch mode by **`make ncsim`**. The simulation models will be stored locally in a sub-directory (./xncsim). A cds.lib file will be created automatically, containing the proper VHDL library mapping definitions, as well as an empty hdl.var. Simulation can then be started by using **`make ncsim-launch`**.

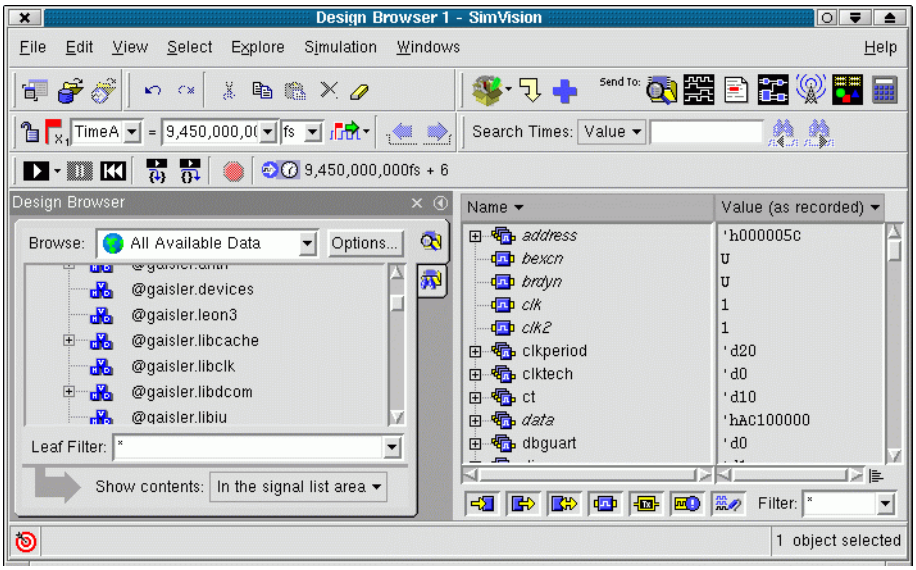


Figure 2. **Ncsim graphical user interface**

To rebuild the local design, run **`make ncsim`** again. This will use the ncupdate utility to rebuild out-of-date files. The tables below summarizes the make targets and the files creates by make scripts.

TABLE 10. Ncsim make targets

Make target	Description
ncsim	Compile or re-analyze GRLIB and local design
ncsim-clean	Remove compiled models and temporary files
ncsim-launch	Start modelsim GUI on current test bench
ncsim-run	Run test bench in batchmode

TABLE 11. Ncsim scripts and files

File	Description
compile.ncsim	Compile script for GRLIB files
make.ncsim	Makefile to rebuild GRLIB and local design
xncsim	Directory with compiled models

The environment variables used by the ncsim flow are:

TABLE 12. Environment variables used by ncsim

Variable	Default value	Description
GRLIB	Set in template design	GRLIB root directory
SIMTOP	Set in template design	Simulation top-level entity
NCVHDL	ncvhdl -nowarn DLCPTH -NOVITALCHECK -linedebug -v93 -nocopy-right -work	VHDL compile command
NCVLOG	ncvlog -nowarn DLCPTH -nocopyright -linedebug $(NCVLOGOPT) -work	Verilog compile command
NCVLOGOPT	(empty)	See use in Verilog compile command above
VHDLOPT	(empty)	Supplied together with command line defined by NCVHDL above

4.8.3 Mentor FormalPro

FormalPro can be launched with its GUI using **make fpro-launch**. The command line mode can be started using **make fpro-run**. In order to perform a sanity check on the flow and RTL design, **make fpro-launch-rtl2rtl** and **make fpro-run-rtl2rtl** can be used to perform verification using the same RTL file list for both A and B. The intended flow is to start FormalPro with **make fpro-launch** that will load the project RTL files a set A. The user will then need to specify the other design (B) to perform the equivalence check against using the GUI.

TABLE 13. FormalPro make targets

Make target	Description
fpro-launch	Start FormalPro in GUI mode and load RTL filelist as A
fpro-launch-rtl2rtl	Start FormalPro in GUI mode and load RTL filelist as A and B
fpro-run	Start FormalPro in CLI mode and load RTL filelist as A
fpro-run-rtl2rtl	Start FormalPro in CLI mode and load RTL filelist as A and B

TABLE 14. FormalPro scripts and files

File	Description
*TOP*_rtl_fpro.fl	FormalPro filelist of project RTL files. *TOP* in the filename is replaced with the top-level design name, typically *leon3mp* or *leon4mp*.

The environment variables used by the FormalPro flow are:

TABLE 15. Environment variables used by FormalPro

Variable	Default value	Description
TOP	Set in template design	Top-level entity

4.8.4 Mentor Questa/ModelSim

The complete GRLIB as well as the local design are compiled by **make vsim**. The compiled simulation models will be stored locally in a sub-directory (./modelsim). A modelsim.ini file will be created automatically, containing the necessary VHDL library mapping definitions. Running **make vsim** again will then use a vmake-generated makefile to check dependencies and rebuild out of date modules. Use of vmake to achieve incremental compile is unsupported with recent Modelsim library formats. When make vsim is used then the library format used by the simulator is forced by setting MTI_DEFAULT_LIB_TYPE to 0.

An other way to compile and simulate the library with modelsim is to use a modelsim project file. When doing **make scripts**, a modelsim project file is created. It is then possible to start vsim with this project file and perform compilation within vsim. In this case, vsim should be started with **make vsim-launch**. In the vsim window, click on the build-all icon to compile the complete library and the local design. The project file also includes one simulation configuration, which can be used to simulate the test bench (see figure below).

The modelsim.ini file of GRLIB sets the VHDL93 configuration to 93, resulting in this being the default VHDL standard used for compiling files. For files requiring other versions of the standard, this must be specified with the vhdlstd attribute.

The file compilation order generated by the GRLIB script environment relies on component declarations to be compiled before instantiations, to allow the compilation to match instances to a component without having compiled the actual entity. This is accentuated in the technology abstraction layer, which includes component instantiations of for all supported technologies - surrounded by VHDL generate statements. Having access to the component declarations for all technologies allows the compiler to proceed without knowing the generic settings, which will be applied at load time. For these reasons, GRLIB requires the default binding at load time, which is the default behavior of Modelsim. The modelsim.ini file of GRLIB does not set the bind at compile flag and the default vcom call does not apply the -bindAtCompile flag. If using another modelsim.ini or vcom call, care must be taken not to use binding at compile time or else vcom will issue the warning or error vcom-1253.

When launching vsim, the default vopt arguments in GRLIB are +acc and -nowarn 1. +acc is used to provide full debug access while -nowarn 1 suppresses warnings about potentially unbound components in the technology abstraction layer. These flags are only used if vsim options are not superseded by setting the VSIMOPT to a non-empty value.

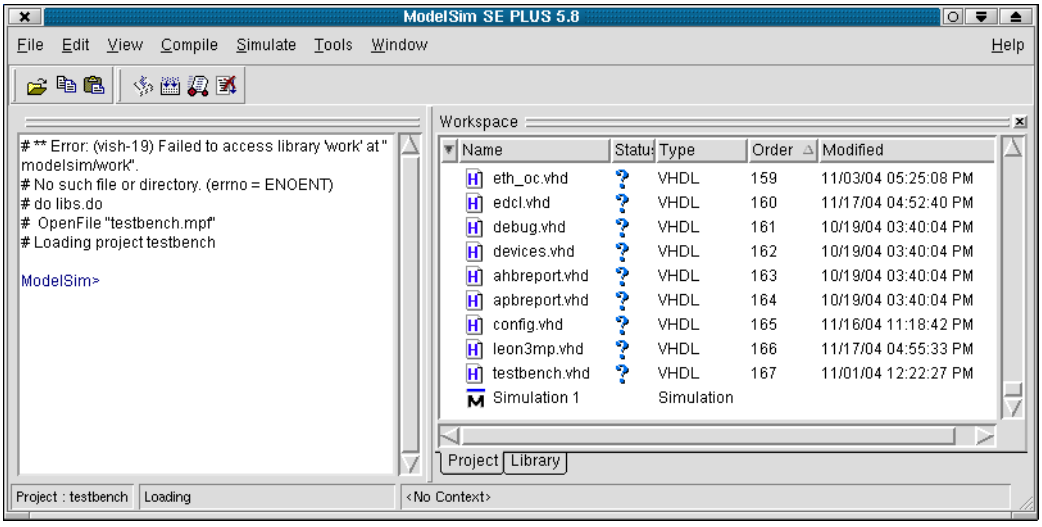


Figure 3. **Modelsim simulator window using a project file**

TABLE 16. Modelsim make targets

Make target	Description
vsim	Compile or re-analyze local design
vsim-clean	Remove compiled models and temporary files

TABLE 16. Modelsim make targets

Make target	Description
vsim-launch	Start modelsim GUI on current test bench
vsim-fix	Run after **make vsim** to fix problems with make in CygWin
vsim-run	Run test bench in batch mode

TABLE 17. Modelsim scripts and files

File	Description
compile.vsim	Compile script for GRLIB files
make.work	Makefile to rebuild GRLIB and local design
modelsim	Directory with compiled models
SIMTOP.mpf	Modelsim project file for compilation and simulation

The environment variables used by the Questa/Modelsim flow are:

TABLE 18. Environment variables used by Questa/Modelsim

Variable	Default value	Description
GRLIB	Set in template design	GRLIB root directory
SIMTOP	Set in template design	Top-level entity
SVLOG	$(VLOG)	svlog command used when compiling SystemVerilog files.
VCOM	vcom -quiet $(VCOMOPT) -93	vcom command used when compiling VHDL files. If the vhdstd attribute is used, the -93 flag will be replaced.
VCOMOPT	(empty)	Extra flags supplied to VCOM command, see above.
VHDLOPT	(empty)	Flags supplied to vcom when compiling VHDL files
VLOG	vlog -quiet $(VLOGOPT)	vlog command used when compiling Verilog files.
VLOGOPT	(empty)	Extra flags supplied to VLOG command, see above.
VSIMOPT	(empty)	If VSIMOPT is set then the value will be supplied as command line arguments when starting vsim using the make targets vsim-run and vsim-launch

The file attributes used by Questa/Modelsim are:

TABLE 19. File attributes used by Questa/Modelsim

Attribute	Default value	Description
vhdlstd	93	Version of the VHDL standard to use for each file

4.8.5 Aldec Active-HDL

The Active-HDL tool from Aldec can be used in the standalone batch mode (vsimsa.bat) and in the GUI mode (avhdl.exe, or started from Windows icon/menu).

The batch mode does not support waveforms and is generally not directly transferable to the GUI mode. The batch mode uses ModelSim compatible command line names such as *vlib* and *vcom*. To use the batch mode, one must ensure that these commands are visible in the shell to be used. Note that the batch mode simulator requires a separate license from Active-HDL.

In batch mode, the completed GRLIB as well as the local design are compiled by **make vsimsa**. The compiled simulation models will be stored locally in a sub-directory (./activehdl). A vsimsa.cfg file will be created automatically, containing the necessary VHDL library mapping definitions. The simulation can then be started using the Active-HDL *vsimsa.bat* or *vsim* command. The simulation can also be started with **make vsimsa-run**.

Another way to compile and simulate the library is with the Active-HDL GUI using a *tcl* command file. When doing **make avhdl**, the *tcl* command file is automatically created for GRLIB and the local design files. The file can then be executed within Active-HDL with do avhdl.tcl, creating all necessary libraries and compiling all files. The compiled simulation models will be stored locally in a sub-directory (./work). Note that only the local design files are directly accessible from the design browser within Active-HDL. The compilation and simulation can also be started from the cygwin command line with **make avhdl-launch**.

Note that it is not possible to use both batch and GUI mode in the same design directory.

Note that simulation libraries provided with GRLIB may collide with libraries that are automatically included by Active-HDL. In this case the user needs to determine if the GRLIBlibraries should be skipped or if the inclusion of Aldec's own libraries should be disabled in Active-HDL.

TABLE 20. Active-HDL make targets

Make target	Description
vsimsa	Compile GRLIB and local design
vsimsa-clean	Remove compiled models and temporary files
vsim-run	Run test bench in batch mode (must be compiled first)
avhdl	Setup GRLIB and local design
avhdl-clean	Remove compiled models and temporary files
avhdl-launch	Compile and Run test bench in GUI mode (must be setup first)

TABLE 21. Active-HDL scripts and files

File	Description
compile.asim	Compile script for GRLIB files (batch mode)
make.asim	Compile script for GRLIB files and local design (batch mode)
activehdl	Directory with compiled models (batch mode)
work	Directory with compiled models (GUI mode)
avhdl.tcl	Active-HDL *tcl* file for compilation and simulation (GUI mode)

The environment variables used by the Active-HDL flow are:

TABLE 22. Environment variables used by Active-HDL

Variable	Default value	Description
GRLIB	Set in template design	GRLIB root directory
SIMTOP	Set in template design	Top-level entity

TABLE 22. Environment variables used by Active-HDL

Variable	Default value	Description
ACOM	acom -quiet $(ACO-MOPT) -work	VHDL compile command
ACOMOPT	(empty)	Extra flags for VHDL compile command, see above
ALOG	ALOG = alog -quiet $(ALOGOPT) -work	Verilog and SystemVerilog compile command
ALOGOPT	(empty)	Extra flags for Verilog compile command, see above
VHDLOPT	(empty)	Flags supplied to acom when compiling VHDL files

The file attributes used by Active-HDL are:

TABLE 23. File attributes used by Active-HDL

Attribute	Default value	Description
vhdlstd	93	Version of the VHDL standard to use for each file

4.8.6 Aldec ALINT

The ALINT tool from Aldec can be used in the standalone batch mode and in the GUI mode.

TABLE 24. ALINT make targets

Make target	Description
alint-comp	Compilation time linting
alint-elab	Compilation time linting followed by elaboration time linting

The environment variables used by the ALINT flow are:

TABLE 25. Environment variables used by ALINT

Variable	Default value	Description
GRLIB	Set in template design	GRLIB root directory
TOP	Set in template design	Top-level entity
SVLOG	vlog -incr -quiet $(VLOGOPT)	svlog command used when compiling SystemVerilog files. These flags are appended with the flags -alint -alint_elabchecks -alint_avdb alint.avdb
VCOM	vcom -incr -nowarn ELAB1_0026 -nowarn COMP96_0259 -quiet $(VCOMOPT) -93	vcom command used when compiling VHDL files. These flags are appended with the flags -alint -alint_elabchecks -alint_avdb alint.avdb
VCOMOPT	(empty)	Extra flags supplied to VCOM command, see above.
VHDLOPT	(empty)	Flags supplied to vcom when compiling VHDL files
VLOG	vlog -incr -v2k5 -quiet $(VLOGOPT)	vlog command used when compiling Verilog files. These flags are appended with the flags -alint -alint_elabchecks -alint_avdb alint.avdb
VLOGOPT	(empty)	Extra flags supplied to VLOG command, see above.

4.8.7 Aldec Riviera

The Riviera tool from Aldec can be used in the standalone batch mode and in the GUI mode. The two modes are compatible, using the same compiled database.

In both modes, the complete GRLIB as well as the local design are compiled by **make riviera**.

If GRLIB_SIMULATOR is set to ALDEC_RWS then the compiled simulation models will be stored locally within a Riviera workspace in a sub-directory (./riviera_ws). If GRLIB_SIMULATOR is set to ALDEC then a legacy flow will be used, without creating the Riviera workspace. The recommended setting is GRLIB_SIMULATOR=ALDEC

Note that the workspace flow in GRLIB is currently experimental.

The standalone batch mode simulation can be started with **make riviera-run**. The GUI mode simulation can be started with **make riviera-launch**. Both of these targets require **make riviera** to be run first in order to compile the design.

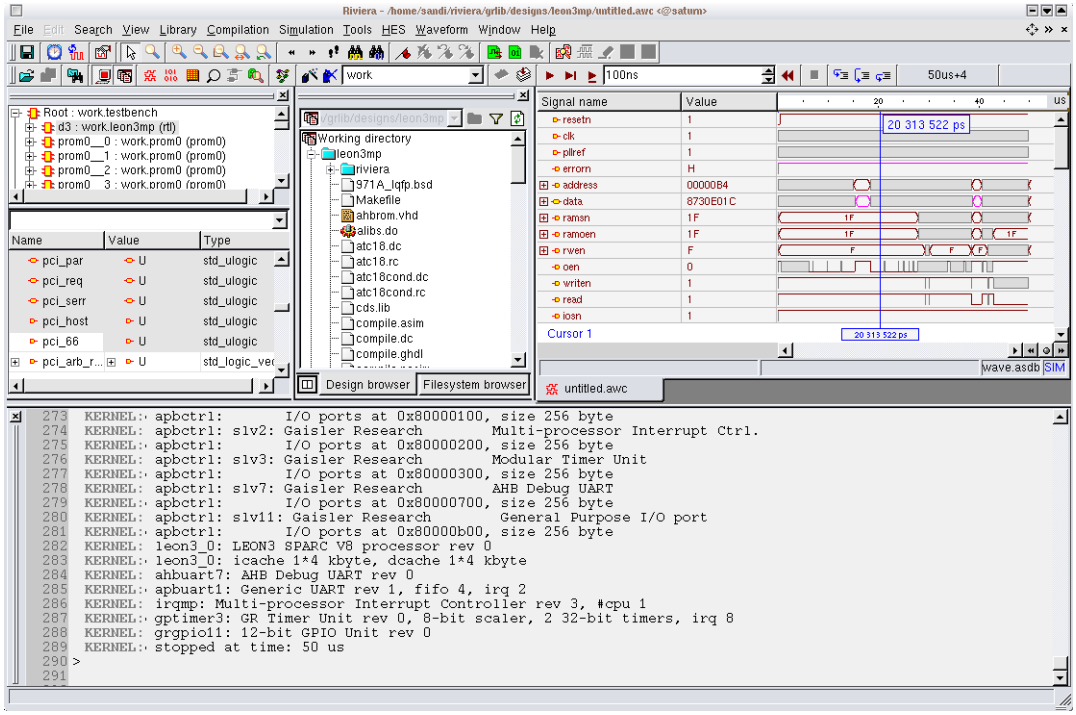


TABLE 26. Riviera make targets

Make target	Description
riviera	Compile GRLIB and local design
riviera-clean	Remove compiled models and temporary files
riviera-run	Run test bench in batch mode (must be compiled first)
riviera-launch	Run test bench in GUI mode (must be compiled first)

TABLE 27. Riviera scripts and files

File	Description
make.riviera	Riviera script for GRLIB_SIMULATOR=ALDEC
riviera_ws_create.do	Rivera script file for simulation (GUI mode)

The environment variables used by the Riviera flow are:

TABLE 28. Environment variables used by Riviera

Variable	Default value	Description
GRLIB	Set in template design	GRLIB root directory
SIMTOP	Set in template design	Top-level entity
SVLOG	vlog -incr -quiet $(VLOGOPT)	svlog command used when compiling SystemVerilog files.
VCOM	vcom -incr -nowarn ELAB1_0026 -nowarn COMP96_0259 -quiet $(VCOMOPT) -93	vcom command used when compiling VHDL files. If the vhdlstd attribute is used, the -93 flag will be replaced.
VCOMOPT	(empty)	Extra flags supplied to VCOM command, see above.
VHDLOPT	(empty)	Flags supplied to vcom when compiling VHDL files
VLOG	vlog -incr -v2k5 -quiet $(VLOGOPT)	vlog command used when compiling Verilog files.
VLOGOPT	(empty)	Extra flags supplied to VLOG command, see above.
VSIMOPT	(empty)	If VSIMOPT is set then the value will be supplied as command line arguments when starting vsim using the make targets vsim-run and vsim-launch

The file attributes used by Riviera are:

TABLE 29. File attributes used by Riviera

Attribute	Default value	Description
vhdlstd	93	Version of the VHDL standard to use for each file

4.8.8 Synopsys VCS

VCS can be run in either batch mode or GUI mode.

The design is built using **make vcs-elab**. The simulation models will be stored locally in a subdirectory (./simv).

The standalone batch mode simulation can be started with **make vcs-run**. The GUI mode simulation can be started with **make vcs-launch**. Both of these targets require **make vcs-elab** to be run first in order to compile the design. There's no support for running VCS through the make **sim-run** and **make sim-launch** targets.

When running VCS on a 64-bit Linux machine it may be necessary to set the VCS_TARGET_ARCH environment variable to linux64 to ensure that the correct executable is used.

TABLE 30. VCS make targets

Make target	Description
vcs-comp	Compile GRLIB and local design
vcs-elab	Elaborate compiled design
vcs-clean	Remove compiled models and temporary files

TABLE 30. VCS make targets

Make target	Description
vcs-run	Run test bench in batch mode (must be compiled first)
vcs-launch	Run test bench in GUI mode (must be compiled first)

TABLE 31. VCS scripts and files

File	Description
make.simv	Makefile to rebuild GRLIB and local design
simv	Directory with compiled models

The environment variables used by the VCS flow are:

TABLE 32. Environment variables used by VCS

Variable	Default value	Description
GRLIB	Set in template design	GRLIB root directory
SIMTOP	Set in template design	Simulation top-level entity
VCSELAB	-debug_access+all	Extra options for vcs commmand
VLOGANOPT	(empty)	Extra options for vlogan command
VHDLANOPT	(empty)	Extra options for vhdlan command

4.8.9 Synthesis with Synplify

The **make scripts** command will create a compile.synp file which contains Synplify tcl commands for analyzing all GRLIB files and a synplify project file called TOP_synplify.prj, where TOP will be replaced with the name of the top level entity.

Synthesizing the design in batch mode using the generated project file can be done in one step using **make synplify**. All synthesis results will be stored locally in a sub-directory (./synplify). Running Synplify in batch requires that it supports the -batch option (Synplify Professional). If the installed Synplify version does not support -batch, first create the project file and then run Synplify interactively. By default, the synplify executable is called 'synplify_pro'. This can be changed by supplying the SYNPLIFY variable to 'make':

```
make synplify SYNPLIFY=synplify_pro.exe
```

The synthesis script will set the following mapping option by default:

```
set_option -symbolic_fsm_compiler 0
set_option -resource_sharing 0
set_option -use_fsm_explorer 0
set_option -write_vhdl 1
set_option -disable_io_insertion 0
```

TABLE 33. Synplify make targets

Make target	Description
synplify	Synthesize design in batch mode
synplify-clean	Remove compiled models and temporary files
synplify-launch	Start synplify interactively using generated project file

TABLE 34. Synplify scripts and files

File	Description
compile.synp	Tcl compile script for all GRLIB files
TOP_synplify.prj	Synplify project file
synplify	Directory with netlist and log files

The environment variables used by the Synplify flow are:

TABLE 35. Environment variables used by Synplify

Variable	Default value	Description
GRLIB	Set in template design	GRLIB root directory
TOP	Set in template design	Top-level entity
FDCFILE		Constraint file(s). Added using the -fpga_constraint flag.
NETLISTTECH	Set depending on value of the TECHNOLOGY variable	If script generation finds EDIF netlists in the directory $(GRLIB)/netlists/xilinx/$(NETLISTTECH)/ then these files will be added to the Synplify project.
PACKAGE	(empty)	Selects target package. Typically provided by include of board Makefile in design Makefile.
PART	(empty)	Selects target part. Typically provided by include of board Makefile in design Makefile.

TABLE 35. Environment variables used by Synplify

Variable	Default value	Description
SDCFILE	(empty)	Constraint file(s). Added using the -constraint flag.
SPEED	(empty)	Selects target speed grade. Typically provided by include of board Makefile in design Makefile.
SYNFREQ	(empty)	Target clock frequency
SYNPLIFY	synplify_pro	Synplify Pro binary
SYNPOPT		Additional options can be set through the SYNPOPT variable in the Makefile: `SYNPOPT="set_option -pipe 0; set_option -retiming 1”`
SYNPVLOGDEFS	(empty)	If set then the value of this variable will be added to the Synplify project as `set_option -hdl_define -set "$(SYNPVLOGDEFS)"`
SYNPVLOGINC	(empty)	If set then the value of this variable will be added to the Synplify project as `set_option -include_path "$(SYNPVLOGINC)"`
SYNPVLOGSTD	(empty)	If set then the value of this variable will be added to the Synplify project as `set_option -vlog_std "$(SYNPVLOGSTD)"`
TECHNOLOGY	(empty)	Selects target technology. Typically provided by include of board Makefile in design Makefile.

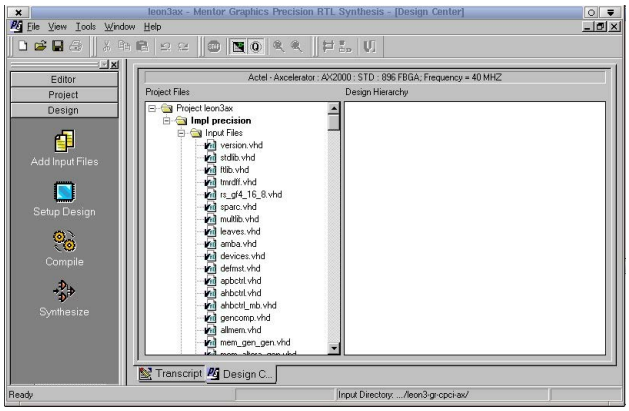
4.8.10 Synthesis with Mentor Precision

Note: GRLIB contains support for generating project files for Precision and starting the tool. Precision support is provided as-is and is not tested with the latest versions by Frontgrade Gaisler.

The **make scripts** command will create a TOP_precision.tcl file which contains tcl script to create a Precision project file. The project file (TOP_precision.psp) is created on the first invocation of Precision, but can also be created manually with **precision -shell -file TOP_precision.tcl**.

Synthesizing the design in batch mode can be done in one step using **make precision**. All synthesis results will be stored locally in a sub-directory (./precision). Precision can also be run interactively by issuing **make precision-launch**. By default, the Precision executable is called with 'precision'. This can be changed by supplying the PRECISION variable to 'make':

```
make precision PRECISION=/usr/local/bin/precision
```



The environment variable PRECISIONOPT can be set in to pass arguments to Precision. For example, to always start with RTL+ the following line can be added to the design Makefile:

```
PRECISIONOPT=-rtlplus
```

TABLE 36. Precision make targets

Make target	Description
precision	Synthesize design in batch mode
precision-clean	Remove compiled models and temporary files
precision-launch	Start Precision interactively using generated project file

TABLE 37. Precision scripts and files

File	Description
TOP_precision.tcl	Tcl compile script to create Precision project file
TOP_precision.psp	Precision project file
precision	Directory with netlist and log files

The environment variables used by the Precision flow are:

TABLE 38. Environment variables used by Precision

Variable	Default value	Description
GRLIB	Set in template design	GRLIB root directory
TOP	Set in template design	Top-level entity

TABLE 38. Environment variables used by Precision

Variable	Default value	Description
MANUFACTURER	(empty)	Selects target technology manufacturer. Typically provided by include of board Makefile in design Makefile.
MGCPACKAGE	(empty)	Selects target package. Typically provided by include of board Makefile in design Makefile.
MGCPART	(empty)	Selects target part. Typically provided by include of board Makefile in design Makefile.
MGCTECHNOL-OGY	(empty)	Selects target technology. Typically provided by include of board Makefile in design Makefile.
PRECISION	precision	Selects Precision binary
PRECISIONOPT	(empty)	Arguments when starting Precision
SPEED	(empty)	Selects target speed grade. Typically provided by include of board Makefile in design Makefile.
SYNFREQ	(empty)	Target clock frequency

4.8.11 Actel/Microsemi Designer

Actel Designer is used to place&route designs targeting Actel FPGAs. It does not include a synthesis engine, and the design must first be synthesized with synplify.

The **`make scripts`** command will generate a tcl script to perform place&route of the local design in batch mode. The tcl script is named TOP_designer.tcl, where TOP is replaced with the name of the top entity.

The command **`make actel`** will place&route the design using the created tcl script. The design database will be place in actel/TOP.adb. The command **`make actel-launch`** will load the edif netlist of the current design, and start Designer in interactive mode.

GRLIB includes a leon3 design template for the GR-CPCI-AX board from Pender/Gaisler. The template design is located designs/leon3-gr-cpci-ax. The local design file uses board settings from the boards/gr-cpci-ax directory. The leon3-gr-cpci-ax design can be used a template for other AX-based projects.

A template design can specify the variable *DESIGNER_LAYOUT_OPT* to override the switches passed to the *layout* command.

TABLE 39. Actel Designer make targets

Make target	Description
actel	Place&route design in batch mode
actel-clean	Remove compiled models and temporary files
actel-launch	Start Designer interactively using synplify netlist
actel-from	Create FROM memory simulation (from.mem) and programming (from.ufc) files from the input hex file (from.hex)

TABLE 40. Actel Designer scripts and files

File	Description
TOP_designer.tcl	Batch script for Actel Designer place&route

The environment variables used by the Designer flow are listed in the table below. Note that the input to designer is typically a netlist generated by Synplify. This means that the environment variables listed in section 4.8.9 also affect the flow.

TABLE 41. Environment variables used by Designer

Variable	Default value	Description
GRLIB	Set in template design	GRLIB root directory
TOP	Set in template design	Top-level entity
DESIGNER_LAYOUT_OPT	*-effort_level 5 -timing_driven -incremental "OFF"* for Axcelerator technology and *-timing_driven -incremental "OFF"* for others.	Passed to *layout* command
DESIGNER_PACKAGE	(empty)	Used with *set_device* to define the package as -package *"$DESIGNER_PINS $DESIGNER_PACKAGE"*

TABLE 41. Environment variables used by Designer

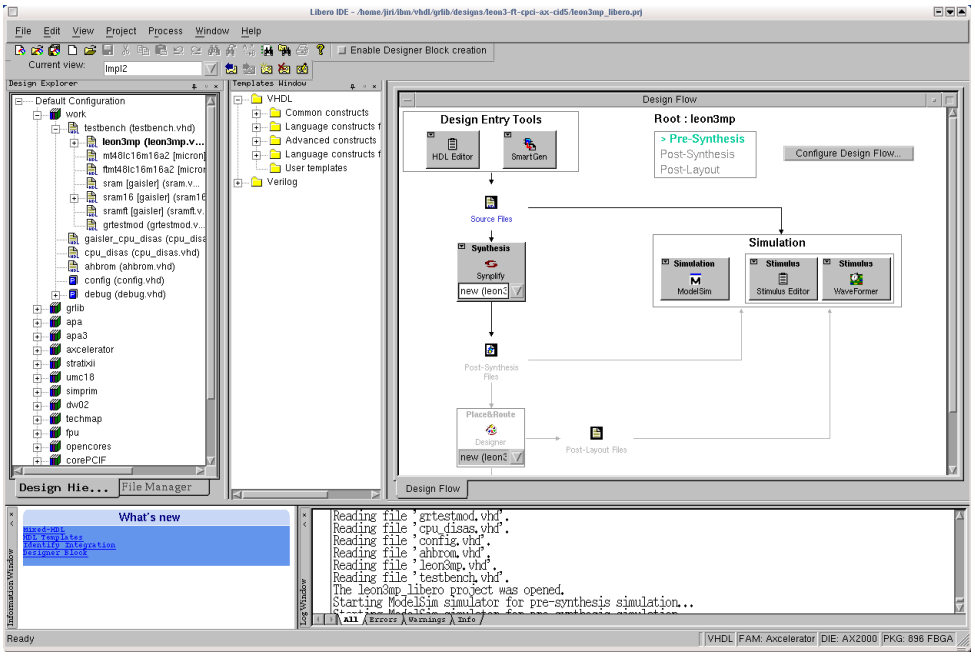
Variable	Default value	Description
DESIGNER_PART	Set to PART variable unless defined.	Used with *set_device* to define *-part*
DESIGNER_PINS	(empty)	Used with *set_device* to define the package as -package *"$DESIGNER_PINS $DESIGNER_PACKAGE"*
DESIGNER_RADEXP	(empty)	Used with *set_device* to define *-radexp*
DESIGNER_TECHNOLOGY	Set to TECHNOLOGY variable unless defined.	Used with *new_design* to define *-family*
DESIGNER_TEMPR	(empty)	Used with *set_device* to define *-temprange*
DESIGNER_VOLTAGE	(empty)	Used with *set_device* to define *-voltage*
DESIGNER_VOLTRANGE	(empty)	Used with *set_device* to define *-voltrange*
PDC	(empty)	PDC file
PDC_EXTRA	(empty)	Optional additional PDC file
SDC	(empty)	SDC file
SDC_EXTRA	(empty)	Optional additional SDC file
SPEED	(empty)	Used with *set_device* to define *-speed*
TECHNOLOGY	(empty)	Affects export format to produce both AFM-APS2 and prb when set to Axcelerator.

4.8.12 Microsemi Libero

Microsemi Libero is an integrated design environment for implementing Microsemi FPGAs. It consists of Microsemi-specific versions of Synplify and Modelsim, together with the Microsemi Designer back-end tool.

Using Libero to implement GRLIB designs is possible using recent versions of Libero IDE and Libero SoC. The **`make scripts`** command will create a Libero project file called TOP_libero.prj for Libero IDE and TOP_libero.tcl for Libero SoC. Libero can then be started with the command **`make libero-launch`**. Implementation of the design is done using the normal Libero flow.

Note that the GRLIB infrastructure generates a project file for Libero IDE that is then loaded when using **`make libero-launch`**. For Libero SoC, the GRLIB infrastructure generates a Tcl script that is executed within Libero the first time **`make libero-launch`** is run. This Tcl script generates the Libero SoC project. Subsequent invocations of **`libero-launch`** will then directly launch Libero SoC loading the generated Libero SoC project.



Note for Libero IDE: Note that when Synplify is launched from Libero the first time, the constraints file defined in the local Makefile are not included in the project, and must be added manually. Before simulation is started first time, the file testbench.vhd in the template design should be associated as stimuli file.

TABLE 42. Libero make targets

Make target	Description
scripts	Created libero project file
libero-launch	Create project file and launch libero
libero	Synthesize, P&R and generate programming file for design in batch mode
libero-prog-fpga	Program FPGA in batch mode
libero-from	Create FROM memory simulation (from.mem) and programming (from.ufc) files from the input hex file (from.hex)

TABLE 43. Libero scripts and files

File	Description
TOP_libero.prj	Libero IDE project file
TOP_libero.tcl	Script that generates Libero SoC project
TOP_libero_genbit.tcl	Script for generating programming file
TOP_libero_progfpga.tcl	Script for programming FPGA
TOP_libero/TOP.prjx	Libero SoC project file generated by running TOP_libero.tcl

The environment variables used by the Libero IDE flow are listed in the table below. The variables used for the Libero SoC flow is listed in a table further down.

TABLE 44. Environment variables used by Libero IDE

Variable	Default value	Description
GRLIB	Set in template design	GRLIB root directory
TOP	Set in template design	Top-level entity
SIMTOP	Set in template design	Top-level entity for simulation
SPEED	(empty)	Value assigned to Libero project file KEY VendorTechnology_Speed
DESIGNER_VOLTAGE	(empty)	Value assigned to Libero project file KEY VendorTechnology_DieVoltage
DESIGNER_TEMPR	(empty)	Value assigned to Libero project file KEY VendorTechnology_TEMPR
DESIGNER_VOLTRANGE	(empty)	Value assigned to Libero project file KEY VendorTechnology_VOLTR
DESIGNER_TECHNOLOGY	If unset then assigned from the variable TECHNOLOGY	Value assigned to Libero project file KEY VendorTechnology_FAMILY
LIBERO_DIE	(empty)	Value assigned to Libero project file KEY VendorTechnology_Die
LIBERO_PACKAGE	(empty)	Value assigned to Libero project file KEY VendorTechnology_Package
SDCFILE	(empty)	Added to Libero project synthesis file set. Added as synthesis constraints.
SDC	(empty)	Added to Libero project synthesis file set
PDC	(empty)	Added to Libero project synthesis file set

The environment variables used by the Libero SoC flow are listed in the table below.

TABLE 45. Environment variables used by Libero SoC

Variable	Default value	Description
GRLIB	Set in template design	GRLIB root directory
TOP	Set in template design	Top-level entity
SIMTOP	Set in template design	Top-level entity for simulation

TABLE 45. Environment variables used by Libero SoC

Variable	Default value	Description
TECHNOLOGY	(empty)	Used with *new_project* command to define *-family*. Also used to differentiate between script generation options for supported target devices.
PART	(empty)	Used with *new_project* command to define *-die*.
DESIGNER_PINS	(empty)	Used with *new_project* command to define *-package*.
DESIGNER_PACKAGE	(empty)	Used with *new_project* command to define *-package*.
SPEED	(empty)	Used with *new_project* command to define *-speed*.
DESIGNER_VOLTAGE	(empty)	Used with *new_project* command to define *-die_voltage*.
DESIGNER_PARTR	(empty)	Used with *new_project* command to define *-adv_options {PART_RANGE:}*
DESIGNER_TEMPR	(empty)	Used with *new_project* command to define *-adv_options {TEMPR:}*
DESIGNER_VOLTRANGE	(empty)	Used with *new_project* command to define *-adv_options* for *VCCI_1.2_VOLTR, VCCI_1.5_VOLTR, VCCI_1.8_VOLTR, VCCI_2.5_VOLTR, VCCI_3.3_VOLTR, VOLTR*
RAD_EXPOSURE	(empty)	Used with *new_project* command to define *-adv_options {RAD_EXPOSURE:}*, valid range is 0 to 300. Only applicable when TECHNOLOGY is RTG4.
IO_DEFT_STD	LVCMOS 2.5V	Used with *new_project* command to define *-adv_options {IO_DEFT_STD:}*
DESIGNER_RESTRICT-PROBEPINS	1	Used with *new_project* command to define *-adv_options {RESTRICT-PROBEPINS:}*
ENHANCED_CONSTRAINT_FLOW	(empty)	Used with *new_project* command to turn on enhanced constraint flow. Enhanced constraint flow is used when this variable is set to a non-empty value. Note that the timing constraint files in the template designs are adapted to classic constraint flow and have to be modified if this flow is enabled. If TECHNOLOGY is PolarFire or the Libero version used is 12.0 or newer only enhanced constraint flow is supported, then this variable does not have to be set.

TABLE 45. Environment variables used by Libero SoC

Variable	Default value	Description
DESIGNER_RTG4_SET_MIT-IGATION	Not used	Used with *project_settings* command to define *-enable_set_mitigation.* Only applicable when TECHNOLOGY is RTG4. If this variable is unassigned then the *-enable_set_mitigation* flag is not included with the *project_settings* command. If the variable is assigned then the flag is included in the project generation. Suitable values are 0 and 1.
GRLIB_SIMULATOR	ModelSim	GRLIB simulator selection, affects contents of generated Tcl file
LIBEROPRECOMPLIBDIR	(empty)	Used when GRLIB_SIMULATOR is set to ModelSim to define location of precompiled libraries
SF2SIMLIB_RIVIERA	(empty)	Used when GRLIB_SIMULATOR is not set to ModelSim to define location of precompiled libraries for Aldec Riviero-PRO simulator.
SDC	(empty)	SDC file added to Libero project
PDC	(empty)	IO PDC file added to Libero project
NDC	(empty)	NDC file added to Libero project. Used for synthesis only
FPPDC	(empty)	Floorplan PDC file added to Libero project
LIBERO_EXTRA_SDC	(empty)	List of additional SDC files to be added to project
SDCFILE	(empty)	SDC file added to Libero project. Used for synthesis only.
FDC	(empty)	FDC file added to Libero project. Used for synthesis only.
LIBERO_VERSION	Not used	If Libero version 12.0 or newer is used this variable has to be set to 12.

4.8.13 Altera Quartus

Altera Quartus is used for Altera FPGA targets, and can be used to both synthesize and place&route a design. It is also possible to first synthesize the design with synplify and then place&route with Quartus.

The **`make scripts`** command will generate two project files for Quartus, one for an EDIF flow where a netlist has been created with synplify and one for a Quartus-only flow. The project files are named TOP.qpf and TOP_synplify.qpf, where TOP is replaced with the name of the top entity.

The command **`make quartus`** will synthesize and place&route the design using a quartus-only flow in batch mode. The command **`make quartus-synp`** will synthesize with synplify and run place&route with Quartus. Interactive operation is achieved through the command **`make quartus-launch`** (quartus-only flow), or **`make quartus-launch-synp`** (EDIF flow). Quartus can also be started manually with **`quartus TOP.qpf`** or **`quartus TOP_synplify.qpf`**.

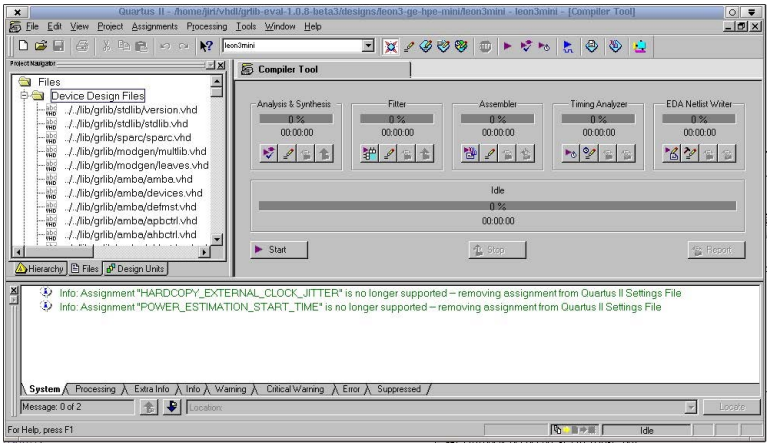


TABLE 46. Altera Quartus make targets

Make target	Description
quartus	Synthesize and place&route design with Quartus in batch mode
quartus-clean	Remove compiled models and temporary files
quartus-launch	Start Quartus interactively using Quartus-only flow
quartus-launch-synp	Start Quartus interactively using EDIF flow
quartus-map	Synthesize design with Quartus in batch mode
quartus-synp	Synthesize with synplify and place&route with Quartus in batch mode
quartus-prog-fpga	Program FPGA in batch mode

TABLE 47. Altera Quartus scripts and files

File	Description
TOP.qpf	Project file for Quartus-only flow
TOP_synplify.qpf	Project file for EDIF flow

The environment variables used by the Altera Quartus flow are listed in the table below.

TABLE 48. Environment variables used by Quartus

Variable	Default value	Description
GRLIB	Set in template design	GRLIB root directory
TOP	Set in template design	Top-level entity
QSF	(empty)	(optional) QSF file to be read into generated project
QSF_APPEND	(empty)	(optional) Additional QSF file to be read into generated project

4.8.14 Xilinx ISE

Xilinx ISE is used for Xilinx FPGA targets, and can be used to simulate, synthesize and place&route a design. It is also possible to first synthesize the design with synplify and the place&route with ISE. It is generally recommended to use the latest version of ISE. Simulation of GRLIB template designs using ISIM is supported as of ISE-13.2. The simulator is launched from the project navigator GUI.

The `make scripts` command will create an XML project file (TOP.xise), useful with ISE-11 and above. When executing **make ise-launch**, this XML will be used to launch the ISE project manager. Synthesis and place&route can also be run in batch mode (preferred option) using **make ise** for the XST flow and **make ise-synp** for synplify flow.

Many Xilinx FPGA boards are supported in GRLIB, and can be re-programmed using **make ise-prog-fpga** and **make ise-prog-prom**. The first command will only re-program the FPGA configuration, while the second command will reprogram the configuration proms (if available). Programming will be done using the ISE Impact tool in batch mode.

When simulating designs that depends on Xilinx macro cells (RAM, PLL, pads), a built-in version of the Xilinx UNSIM simulation library will be used. The built-in library has reduced functionality, and only contains the cells used in grlib. The full Xilinx UNISIM library can be installed using make install-unisim. This will copy the UNISIM files from ISE into grlib. A **make distclean** must first be given before the libraries can be used. It is possible to revert to the built-in UNISIM libraries by issuing **make remove-unisim**. To simulate designs using the Xilinx MIG memory controllers, the secureIP library must first be installed using **make install-secureip**. The Xilinx UNIMACRO library can also be installed/removed by using **make install-unimacro** and **make remove-unimacro**. Verilog versions of the above libraries can also be installed using the install targets with a **_ver** ending.

Note: to install the Xilinx UNISIM/SeureIP/UNIMACRO files, the variable XILINX must point to the installation path of ISE. The variable is normally set automatically during installation of ISE.

Note: Installation of secureip depends on the GRLIB_SIMULATOR setting to select encrypted models for either Aldec or Mentor tools. If the simulator is changed then **make install-secureip** must be rerun.

TABLE 49. Xilinx ISE make targets

Make target	Description
ise	Synthesize and place&route design with XST in batch mode
ise-prec	Synthesize and place&route design with Precision in batch mode
ise-synp	Synthesize and place&route design with Synplify in batch mode
ise-launch	Start project navigator interactively using XST flow
ise-launch-synp	Start project navigator interactively using EDIF flow
ise-map	Synthesize design with XST in batch mode
ise-prog-fpga	Program FPGA on target board using JTAG
ise-prog-fpga-ref	Program FPGA on target board with reference bit file
ise-prog-prom	Program configuartion proms on target board using JTAG
ise-prog-prom-ref	Program configuartion proms with reference bit file
install-unisim	Install Xilinx UNISIM libraries into GRLIB
remove-unisim	Remove Xilinx UNISIM libraries from GRLIB
install-secureip	Install Xilinx SecureIP files into GRLIB
remove-secureIP	Remove Xilinx SecureIP files from GRLIB
install-unimacro	Install Xilinx UNIMACRO files into GRLIB (requires install-unisim)
remove-unimacro	Remove Xilinx UNIMACRO files from GRLIB
install-unisim_ver	Install Verilog version of UNISIMS into GRLIB
install-xilinxcorelibs_ver	Install Verilog version of Xilinx CoreLibs into GRLIB
install-secureip_ver	Install Verilog version of SecureIP into GRLIB (secureip_ver)

TABLE 50. Xilinx ISE scripts and files

File	Description
compile.xst	XST synthesis include script for all GRLIB files
TOP.xst	XST synthesis script for local design
TOP.npl	ISE 8 project file for XST flow
TOP.ise	ISE 9/10project file for XST flow
TOP.xise	ISE 11/12/13/14 XML project file for XST flow
TOP_synplify.npl	ISE 8 project file for EDIF flow

ISE project properties:

The ISE project file is automatically generated based on settings in the current design's Makefile. Variables such as device, speed grade and so on are defined in the template design's Makefile, or taken from the board directory specified in the template design's Makefile. A few additional ISE properties can be set in the board or template design Makefile. If the variables are not assigned then a default value will be used. Table 51 below lists the ISE project properties that can be overriden by defining specific variables

As an example, to change the default simulator used by the ISE project to ModelSim the following definition can be added to the design's Makefile:

```
GRLIB_XIL_PN_Simulator=Modelsim-SE VHDL
```

Old and deprecated ISE versions:

The **make scripts** command also generates .npl project files for the ISE-8 project navigator, for both EDIF flow where a netlist has been created with synplify and for ISE/XST flow. The project navigator can be launched with **make ise-launch-synp** for the EDIF flow, and with **make ise-launch8** for the XST flow. The project navigator can also be started manually with **ise TOP.npl** or **ise TOP_synplify.npl**. The .npl files are intended to be used with ISE 6 - 8.

For ISE-9 and ISE-10, an .ise file will be generated using xtclsh when **make ise-launch** is given, or by **make TOP.ise**. Note that the Xilinx xtclsh application may operate very slowly.

The environment variables used by the ISE 11/12/13/14 flow are listed in the table below.

TABLE 51. Environment variables used by ISE 11/12/13/14

Variable	Default value	Description
GRLIB	Set in template design	GRLIB root directory
TOP	Set in template design	Top-level entity
SIMTOP	Set in template design	Simulation top-level entity
UCF	(empty)	UCF file to be read into generated project
PART	(empty)	Defines device
ISE11TECH	Technology selection. Set via ISETECH variable, which is in turn set from TECHNOLOGY variable if unset.	Defines device family
NETLISTTECH	Set depending on value of the TECHNOLOGY variable	The directory $(GRLIB)/netlists/xilinx/$(NETLISTTECH)/ will be added to the project's Macro Search Path.
ISEMAPOPT	(empty)	Map command line options
XSTOPT	(empty)	XST command line options

TABLE 51. Environment variables used by ISE 11/12/13/14

Variable	Default value	Description
EFFORT	(empty)	Place & Route Effort Level (Overall)
GRLIB_XIL_PN_Pack_Reg_Latches_into_IOBs	For Inputs and Outputs	Pack I/O Registers/Latches into IOBs
PACKAGE	(empty)	Defines package
GRLIB_XIL_PN_Simulator	ISim VHDL/Verilog	Simulator
SPEED	(empty)	Defines speed grade

4.8.15 Xilinx PlanAhead

Xilinx PlanAhead is supported for Xilinx devices and prototype boards to improve runtime and performance. The GRLIB enviroment allows the user to experiment with diffrent implementation options to improve design results via runtime option specificed in `$(GRLIB)/boards/$(BOARD)/Makefile.inc`. The Xilinx PlanAhead flow should be seen as an extension of GRLIB Xilinx ISE flow.

The **`make scripts`** command will create compile scripts for the PlanAhead tool, useful with ISE-14 and above. When executing **make planahead-launch**, the compile scripts will be used to launch the PlanAhead project manager. Synthesis and place&route can also be run in batch mode (preferred option) using **make planahead**.

Many Xilinx FPGA boards are supported in GRLIB, and can be re-programmed using **make ise-prog-fpga** and **make ise-prog-prom**. The first command will only re-program the FPGA configuration, while the second command will reprogram the configuration proms (if available). Programming will be done using the ISE Impact tool in batch mode.

It is possible to specify Bitgen options to be used in the PlanAhead flow. This is done via the **PLANAHEAD_BITGEN** environment variable. If this variable is set then the contents will be used to specify additional Bitgen options in the PlanAhead flow.

TABLE 52. Xilinx PlanAhead specific make targets

Make target	Description
planahead	Synthesize and place&route design with PlanAhead in batch mode
planahead-launch	Start project navigator interactively using planAhead flow
planahead-clean	Remove all planAhead generated project files

TABLE 53. Xilinx PlanAhead scripts and files

File	Description
compile.planahead	PlanAhead synthesis include script for all GRLIB files
planAhead.tcl	PlanAhead script for creating a PlanAhead project and to build the project.

The environment variables used by the PlanAhead flow are listed in the table below.

TABLE 54. Environment variables used by PlanAhead

Variable	Default value	Description
GRLIB	Set in template design	GRLIB root directory
TOP	Set in template design	Top-level entity
SIMTOP	Set in template design	Simulation top-level entity
DEVICE	(empty)	Defines part
PLANAHEAD_SIMSET	sim_1	Name of PlanAhead simset
GRLIB_XIL_PlanAhead_Simulator	If GRLIB_SIMULATOR is set to ModelSim then the PlanAhead simulator is set to Modelsim, otherwhise ISim.	Defines target simulator

TABLE 54. Environment variables used by PlanAhead

Variable	Default value	Description
PROTOBOARD	(empty)	If set then used for *set_property_board* Special cases also exists if protoboard is set to boards support in GRLIB. For example the value zedBoard will lead to zedBoard-specific files being imported.
NETLISTTECH	Set depending on value of the TECHNOLOGY variable	If the directory exists then files in $(GRLIB)/netlists/xilinx/$(NETLISTTECH)/ will be imported into the project.
GRLIB_XIL_PlanAhead_sim_verilog_define	(empty)	If set then the value is used with *set_property verilog_define* for the simulation fileset.
GRLIB_XILINX_SOURCE_MGMT_MODE	(empty)	If set, then used with *set_property source_mgmt_mode*
UCF_PLANAHEAD	(empty)	If set then the list of files are included using the *read_ucf* command.
PLANAHEAD_SYNTH_STRATEGY	TimingWithIOBPacking	Defines synthesis strategy
PLANAHEAD_IMPL_STRATEGY	ParHighEffort	Defines implementation strategy
PLANAHEAD_BITGEN	(empty)	Specified Bitgen options
CONFIG_MIG_DDR2	Define din template design configuration	If set to “y” and the file mig/user_design/par/mig.ucf exists then the file will be read using *read_ucf*.

4.8.16 Xilinx Vivado

Xilinx Vivado is the build flow for modern Xilinx FPGA devices and prototype boards, starting from the 7 series. The GRLIB environment allows the user to experiment with different implementation options to improve design results via runtime option specified in `$(GRLIB)/boards/$(BOARD)/Make-file.inc`.

The **`make scripts`** command will create compile scripts for the Vivado tool. When executing **make vivado-launch**, the compile scripts will be used to launch the Vivado project manager. Synthesis and place&route can also be run in batch mode (preferred option) using **make vivado**.

Many Xilinx FPGA boards are supported in GRLIB, and can be re-programmed using **make ise-prog-fpga** and **make ise-prog-prom**. The first command will only re-program the FPGA configuration, while the second command will reprogram the configuration proms (if available). Programming will be done using the ISE Impact tool in batch mode.

TABLE 55. Xilinx Vivado specific make targets

Make target	Description
vivado	Synthesize and place&route design with Vivado in batch mode
vivado-launch	Start project navigator interactively using Vivado flow
vivado-clean	Remove all Vivado generated project files
vivado-prog-fpga	Optional program target for faster programming of the FPGA Device. This target needs Xilinx EDK/SDK to be installed.
vivado-prog-fpga-ref	Program reference bitfile (requires installation of GRLIB bitfiles package)

TABLE 56. Xilinx Vivado scripts and files

File	Description
compile.vivado	Vivado synthesis include script for all GRLIB files
vivado.tcl	Vivado script for creating a PlanAhead project and to build the project.

The environment variables used by the Vivado flow are listed in the table below.

TABLE 57. Environment variables used by Vivado

Variable	Default value	Description
GRLIB	Set in template design	GRLIB root directory
TOP	Set in template design	Top-level entity
SIMTOP	Set in template design	Simulation top-level entity
DEVICE	(empty)	Defines part
VIVADO_SIMSET	sim_1	Name of Vivado simset
GRLIB_XILINX_SOURCE_MGMT_MODE	(empty)	If set, then used with *set_property source_mgmt_mode*
GRLIB_COMPILE_VIVADO_IP	(empty)	If not set, compile_simlibs is called with the -no_ip_compile flag. Set to compile all IPs supplied with Vivado.
GRLIB_XIL_Vivado_sim_verilog_define	(empty)	If set, then used with *set_property verilog_define* for the simulation fileset.

TABLE 57. Environment variables used by Vivado

Variable	Default value	Description
XDC	(empty)	List of files imported with *read_xdc*. Used both in synthesis and in implementation.
TCL	(empty)	List of Tcl scripts that are sourced after including source files and XDC files.
VIVADO_UCF	(empty)	List of files imported with *import_files*. Used both in synthesis and in implementation.
GRLIB_XIL_Vivado_Simulator	If unset then the simulator is selected based on GRLIB_SIMULATOR. Default is XSim.	Defines target simulator
PROTOBOARD	(empty)	If set then used for *set_property board_part*
CONFIG_MIG_7SERIES	(empty)	If set to “y” then automatically includes MIG files depending on template design configuration.
BOARD	(empty)	Used to locate MIG files when CONFIG_MIG_7SERIES is set.
VIVADO_MIG_AXI	(empty)	Used to locate MIG files when CONFIG_MIG_7SERIES is set.
AXI_64	(empty)	Used to locate MIG files when CONFIG_MIG_7SERIES is set.
AXI_128	(empty)	Used to locate MIG files when CONFIG_MIG_7SERIES is set.
DESIGN	(empty)	Used to generate MIG files when CONFIG_MIG_7SERIES is set.
CONFIG_GRETH_ENABLE	(empty)	Used to conditionally locate SGMII IP files in the same way as CONFIG_MIG_7SERIES
NETLISTTECH	Set depending on value of the TECHNOLOGY variable	If the directory exists then files in $(GRLIB)/netlists/xilinx/$(NETLISTTECH)/ will be imported into the project.
VIVADO_SYNTH_FLOW	Vivado Synthesis 2012	Used with *set_property flow* for the synthesis run.
VIVADO_SYNTH_STRATEGY	Vivado Synthesis Defaults	Used with *set_property strategy* for the synthesis run.
VIVADO_IMPL_STRATEGY	Vivado Implementation Defaults	Used with *set_property strategy* for the implementation run.
VIVADO_INCL_DIRS	(empty)	If set then the contents is supplied with the *-include_dirs* flag to the *synth_design* command.

TABLE 57. Environment variables used by Vivado

Variable	Default value	Description
CONFIG_GRHSSL_ENABLE	(empty)	If set to “y”, then automatically imports the technology-specific HSSL SerDes into the project, as long as the board folder contains a Xilinx IP called hssl_serdes.xci.
GRLIB_VIVADO_COMPILE_VERBOSE	(empty)	Debug functionality for *map_xilinx_7series_lib*. If set, the compilation of Xilinx simulation libraries will not be quiet, allowing visibility into the details of the compilation.

4.8.17 Lattice Radiant

Lattice Radiant is the build flow for Nexus family (NX) of Lattice FPGA devices (Crosslink-NX, Certus-NX and CertusPRO-NX) and evaluation boards. The GRLIB environment allows the user to experiment with different implementation options to improve design results via runtime option specified in `$(GRLIB)/boards/$(BOARD)/Makefile.inc`.

The **`make scripts`** command will create compile scripts for the Radiant tool. When executing **make radiant-launch**, the compile scripts will be used to launch the Radiant project manager: here the user can edit the project before further synthesis, place&route etc. To instead automatically synthesize, place&route and generate a bitstream directly in batch mode (i.e. no GUI interaction), then the user can execute **make radiant**. The system will take care of generating IPs and then run the needed steps for obtaining a bitstream.
This procedure works in Cygwin/Windows as well, provided that environment variables show below are correctly initialized in Cygwin.

About Lattice IPs: the make system copies, from the respective board folder (see BOARD variable below), configuration files of needed IP cores (see LATTICE_IP variable below). The system then automatically generates the IPs as they are setup in their respective configuration files.
To add a new IP in a way that it'll be picked up automatically by running **make radiant** or **make radiant-launch**, after opening the project in Radiant, the user first needs to create and instantiate the new IP inside the project (e.g. a "pll" inside a CertusPro project with name "pll_clock"). Then, go to Radiant installation folder and then inside "ip" folder: from here, the user needs to get the path to the IP that is being included (in our example: lifcl/pll). At this point, the user needs to move to the "lattice_ips" folder inside the respective board folder: here they need to recreate the path found inside the "ip" folder (in our example: boards/board_folder/lattice_ips/lifcl/pll/) and here (inside pll/) copy the generated configuration file (in our example: pll_clock.cfg). As last step, the LATTICE_IP variable in the design Makefile needs to be updated with the new IP, by adding the path to the IP followed by the IP name (in our example: lifcl/pll/pll_clock).
If the IPs were already generated, then the system will use the local cfg file and regenerate them every time **make radiant**, **radiant-launch**, **map-radiant-simlibs** are called.

TABLE 58. Lattice Radiant specific make targets

Make target	Description
radiant	Automatically setup, synthesize and generate the bitstream in batch mode
radiant-launch	Setup the project and start project navigator interactively using Radiant flow
radiant-clean	Remove all Radiant generated project files
radiant-ips	Generate the vhdl/verilog files for the required IPs
remove-radiant-ips	Remove the locally generated IPs files
install-radiant-simlibs	Compile Nexus library needed for simulation purposes (only with ModelSim)
map-radiant-simlibs	Edit modelsim.ini file to use the compiled libraries. If not yet compiled, then they'll be automatically compiled before editing modelsim.ini
remove-radiant-simlibs	Delete the compiled libraries from the GRLIB

TABLE 59. Lattice Radiant scripts and files

File	Description
radiant.tcl	Radiant script for creating a Radiant project and to build the project.

The environment variables used by the Radiant flow are listed in the table below.

TABLE 60. Environment variables used by Radiant

Variable	Default value	Description
GRLIB	Set in template design	GRLIB root directory
GRLIB_LATTICE_RADIANT	(empty)	Installation path of Radiant. It needs to be set as environment variable.
TOP	Set in template design	Top-level entity
SIMTOP	Set in template design	Simulation top-level entity
DEVICE	Automatically set	Defines part
BOARD	Set in template design	Board name as in boards folder
GRLIB_LATTICE_RADIANT_-SIM_DEVICE	Set in template design	FPGA library[a] to use for primitives simulation - allowed values are lifcl (Crosslink-NX and Certus-NX) and lfcpnx (CertusPRO-NX)
PDC	Set in template design	.pdc file containing constraints (pin mapping, clock freq. etc) for Mapping and Place&Route
LDC	Set in template design	.ldc file containing constraints (clock freq. etc) for Synthesis
LATTICE_IP	Set in template design	List of Lattice IPs that will be automatically generated from the config files saved in the chosen BOARD folder.

a. As explained in "Installation of Lattice Radiant libraries", the user is urged to use lifcl also for simulating designs featuring Certus-NX (lfd2nx) FPGAs

4.8.18 Lattice ISP Tools

Note: GRLIB contains support for generating project files for Lattice ISP and starting the tool. Lattice ISP support is provided as-is and is not kept up to date by Frontgrade Gaisler.

Implementing GRLIB design on Lattice FPGAs is supported with Synplify for synthesis and the Lattice ISP Lever for place&route. The **make isp-synp** commmand will automatically synthesize and place&route a Lattice design. The associated place&route script is provided in bin/route_lattice, and can be modified if necessary. Supported FPGA families are EC and ECP. On linux, it might be necessary to source the ISP setup script in order to set up necessary paths:

source $ISPLEVER_PATH/ispcpld/bin/setup_lv.sh

TABLE 61. Lattice ISP make targets

Make target	Description
isp-synp	Synthesize and place&route design with Sunplify in batch mode
isp-clean	Remove compiled models and temporary files
isp-prom	Create FPGA prom

4.8.19 Synthesis with Synopsys Design Compiler

The **make scripts** command will create a compile.dc file which contains Design Compiler commands for analyzing all GRLIB files. The compile.dc file can be run manually using 'dc_shell -f compile.dc'. A script for the local design is created automatically and called *TOP_dc.tcl* where *TOP* is the top entity name:

```
$ cat leon4mp_dc.tcl
sh mkdir synopsys
set objects synopsys
#set trans_dc_max_depth 1
#set hdlin_seqmap_sync_search_depth 1
#set hdlin_nba_rewrite false
set hdlin_ff_always_sync_set_reset true
set hdlin_ff_always_async_set_reset false
#set hdlin_infer_complex_set_reset true
#set hdlin_translate_off_skip_text true
set suppress_errors VHDL-2285
#set hdlin_use_carry_in true
source  compile.dc
analyze -f VHDL -library work config.vhd
analyze -f VHDL -library work ahbrom.vhd
analyze -f VHDL -library work clkgate.vhd
analyze -f VHDL -library work qmod.vhd
analyze -f VHDL -library work qmod_prect.vhd
analyze -f VHDL -library work leon4mp.vhd
elaborate leon4mp
```

The script can be run with dc_shell-xg-t via the command **make dc**. The created script will analyze and elaborate the local design. Compilation and mapping will not be performed, the script should be seen as a template only. The default script can be overriden by setting the *DCSCRIPT* variable. Additional command line flags can be passed to dc_shell-xg-t via the *DCOPT* variable.

4.8.20 Synthesis with Cadence RTL Compiler

Note: GRLIB contains support for generating project files for RTL Compiler and starting the tool. RTL Compiler support is provided as-is and is not tested with the latest versions by Frontgrade Gaisler.

The **make scripts** command will create a compile.rc file which contains RTL Compiler commands for analyzing all GRLIB files. The compile.rc file can be run manually using **rc -files compile.rc** or through **make rc**. A script to analyze and synthesize the local design is created automatically and called TOP.rc where TOP is the top entity name:

```
$ cat netcard.rc
set_attribute input_pragma_keyword "cadence synopsys g2c fast ambit pragma"
include compile.rc
read_hdl -vhdl -lib work netcard.vhd
elaborate netcard
write_hdl -generic > netcard_gen.v
```

The created script will analyze and elaborate the local design, and save it to a Verilog file. Compilation and mapping will not be performed, the script should be seen as a template only.

4.8.21 eASIC eTools

GRLIB support for eTools with eASIC Nextreme technology was discontinued in GRLIB version 1.1.0-b4109.

Support for the Nextreme2 technology and eTools 9 can be requested from Frontgrade Gaisler but is not included in any of the default GRLIB distributions. To work with eTools 9 the environment variable ETOOLS_N2X_HOME must be set to the eTools installation directory.

TABLE 62. eASIC Nextreme2 make targets

Make target	Description
import-easic-n2x	Imports eASIC RTL and IP libraries from eTools into GRLIB. Requires that the environment variable.
remove-easic-n2x	Removes eASIC RTL and IP libraries from GRLIB.
etools-n2x-init	Creates a eTools project file. Makes use of the environment variables TOP, DEVICE, PACKAGE, PNC, SDCFILE, and GRLIB_NHCPU. The last variable defines the number of available host CPUs.
etools-n2x-launch	Launch eTools DesignNavigator for the current project
etools-n2x-launch-no_iu	LauncheTools DesignNavigator for the current project in CLI mode.

The GRLIB technology map for eASIC Nextreme2 makes extensive use of eASIC's RAM and pad generators, and also of wrappers for the DDR2 PHY. When eASIC's IP library has been imported into GRLIB (via the *import-easic-n2x* make target), the normal technology map components (pads, memory, DDR2 PHY) can be used.

The GRLIB SYNCRAM* components map to both rFiles and bRAMs. The conditions for selecting between these RAM types may need to be adjusted for each design in order to not over-utilize one or the other. The selection between rFiles and bRAMs is made with the function *n2x_use_rfile(..)* that is defined in the file *lib/techmap/nextreme2/memory_n2x_package.vhd*.

The technology map also includes a clock generator map for eASIC PLLs. However it is strongly recommended to use eASIC's IP generators instead and directly instantiate the Nextreme2 PLLs in the design.

4.8.22 NanoXplore NanoXmap and NanoXpython

NanoXmap is the software provided by NanoXplore that uses the user RTL code and timing constraints to generate a bitstream downloadable in the NanoXplore FPGA. From the NanoXmap user manual:

NanoXmap is a graphical interface that allows user to view and compile an existing HDL project. To create a new project, the user has to use NanoXpython tool.

NanoXpython is a wrapper around Python executable that allows user to control nxmap software. As a wrapper, it fully supports Python syntax, structures and external modules.

For GRLIB designs synthesis using NanoXmap or NanoXpython the user has to set some variables in the Makefile:

DEVICE: Device variant. The value of this variable is passed to `project.setVariantName()` during project creation. The default value is NG-MEDIUM.

TIMING_DRIVEN: If set to 'Yes', algorithms are timing driven (Yes or No, default is Yes)

MAPPING_EFFORT: Effort for an optimized mapping (Low, Medium or High, default is High)

If the user does not set these variables, they will be assigned to their default value by the scripts.

Moreover, the user can define constraints to the nanoxplore tools by writing them in a .py file and then setting the variable **NXCONSTRAINTS** to the path of this file.

For example, in the file example_constraints.py:

```
project.createClock('getClockNet(clkm)', 'clkm', 40000, 0, 20000)
project.addFalsePath('getRegister(rst0|rstout','geRegiser(spw.swloop[0]')
```

And then in the Makefile:

```
NXCONSTRAINTS=example_constraints.py
```

To define the pins the user has to specify their location and their configuration in a file named <TOP>_pads.py. For example, for a top entity named leon3mp, the pads have to be specified in a file called leon3mp_pads.py, which should have this structure:

```
pads = {
'dsurx' : {'location': 'IOB0_D11N',  'type': 'LVCMOS_3.3V_8mA'},
'dsutx' : {'location': 'IOB0_D01N',  'type': 'LVCMOS_3.3V_8mA'},
'spw_rxs[0]'   : {'location': 'IOB12_D04P', 'type': 'LVDS_2.5V', 'differential' : True, 'termination' : '50', 'terminationReference': 'VT', 'turbo': True}
}
```

It is important to use the keyword `pads` at the top of the file, because the python script file will search for a file called <TOP>_pads.py and then import the variable pads from that file. If the file or the variable are not found, then the tool will automatically generate a pad file named <TOP>_generated_pads.py.

The **make scripts** command will generate two python files: one called <TOP>_nanoxmap.py and the other <TOP>_nanoxpython.py. The first file contains all the directives useful for both NanoXmap and NanoXpython (file lists, constraints and project options). The other file contains the commands for running synthesis, placing, routing and bit stream generation using NanoXpython.

The command **make nanoxmap-launch** will perform two actions: first it will run nanoxpython on the file <TOP>_nanoxmap.py. This will generate a nanoxmap project file named <TOP>_native.nxm. Then the command will launch the NanoXmap GUI on this file.

The command **make nanoxpython** will also generate the TOP_native.nxm file but then it will use the file TOP_nanoxpython.py to run consecutively synthesis, place and route and bit stream generation in batch mode (this is the preferred option). Note that currently, the bitstream generation can only be performed by NanoXpython and not through the NanoXmap GUI.

The generated bitstream will be saved in a file named TOP_bitfile.nxb. During the process also the files <TOP>_synthesized.nxm, <TOP>_placed.nxm and TOP_routed.nxm will be generated.

To program the FPGA it is possible to run the commands **make nx-prog-fpga,** which will search for a bitfile named TOP_bitfile.nxb and will load it on the device using the JTAG interface.

Using the command **make nx-prog-prom** it is also possible to write the bitfile in the SPI flash daughter board provided with the NanoXplore Dev Kit V2. The command first writes the EEPROM memory and then validates the memory content.

REMARK: To successfully use these make targets the user has to add to its PATH variable the paths to the installation directories of NanoXmap and NxBase2_cli. Moreover, in order to avoid running NxBase2 as root, the user must setup the Linux udev rules in order to access the board from Linux user space. Further information are available in the NxBase2 User manual.

TABLE 63. NanoXplore specific make targets

Make target	Description
nanoxpython	Synthesize, place&route and bistream generation with nanoxpython in batch mode
nanoxmap-launch	Start project navigator interactively using NanoXmap flow
nanoxmap-clean	Remove all NanoXpython generated project files
nx-prog-fpga	Optional program target for programming the FPGA Device through JTAG. This target needs NxBase2 to be installed.
nx-prog-prom	Writes the bitstream named into the SPI flash daughter board and then validates the memory content. This target needs NxBase2 to be installed

TABLE 64. NanoXmap scripts and files

File	Description
<TOP>_nanoxmap.py	Python file which includes all the project files and synthesys options. Used to create a .nxm project file
<TOP>_nanoxpython.py	Used to run the implementation flow through nanoxpython in batch mode.

The environment variables used by the NanoXpython flow are listed in the table below.

TABLE 65. Environment variables used by NanoXpython

Variable	Default value	Description
GRLIB	Set in template design	GRLIB root directory
TOP	Set in template design	Top-level entity
SIMTOP	Set in template design	Simulation top-level entity
DEVICE	NG-MEDIUM	Defines part
TIMING_DRIVEN	No	Algorithms are timing driven
MAPPING_EFFORT	High	Effort for an optimized mapping
NXCONSTRAINTS	Set in template design as the path to a .py file	The python directives regarding constraints or synthesys options listed in this file will be imported into the project file <TOP>_nanoxmap.py

4.9 XGrlib graphical implementation tool

4.9.1 Introduction

NOTE: Some template designs require commands to be issued to install special libraries or to generate parts of the design. These special commands are not available in XGrlib and must instead be given via the command line interface.

XGrlib serves as a graphical front-end to the makefile system described in the previous chapters. It is written in tcl/tk, using the Visual-tcl (vtcl) GUI builder. XGrlib allows to select which CAD tools will be used to implement the current design, and how to run them. XGrlib should be started in a directory with a GRLIB design, using **`make xgrlib`**. Other make variables can also be set on the command line, as described earlier:

```
make xgrlib SYNPLIFY=synplify_pro GRLIB="../.."
```

Since XGrlib uses the make utility, it is necessary that all used tools are in the execution path of the used shell. The tools are divided into three categories: simulation, synthesis and place&route. All tools can be run in batch mode with the output directed to the XGrlib console, or launched interactively through each tool's specific GUI. Below is a figure of the XGrlib main window:

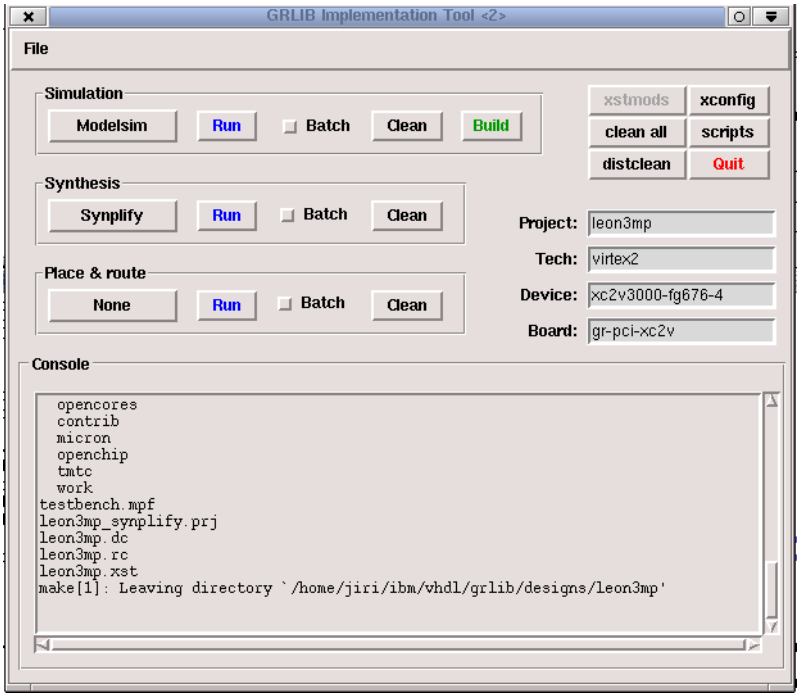


Figure 4. **XGrlib main window**

4.9.2 Simulation

The simulator type can be selected through the left menu button in the frame marked 'Simulation'. There are seven options available: modelsim, ncsim, GHDL, libero, riviera, active-hdl, and active-hdl batch. Once the simulator has been selected, the design can be compiled by pressing the green 'Build' button. The simulator can then be launched interactively by pressing the 'Run' button. If the 'Batch' check-button has been set, the 'Run' button will run the default test bench in batch mode with the output displayed in the console frame. The 'Clean' button will remove all generated file for the selected tool.

Note: on windows/cygwin platforms, launching modelsim interactively can fail due to conflict of cygwin and modelsim tcl/tk libraries.

4.9.3 Synthesis

The synthesis tool is selected through the menu button in the frame labeled with 'Synthesis'. There are five possibilities: Synplify, Altera Quartus, Xilinx ISE/XST, Mentor Precision and Microsemi Libero. The 'Batch' check-button defines if synthesis will be run in batch mode or if the selected tool will be launched interactively. The selected tool is started through the 'Run' button.

If a tool is started interactively, is automatically loads a tool-specific project file for the current design. It is then possible to modify the settings for the project before synthesis is started. Only one tool should be started at a time to avoid I/O conflicts. The 'Clean' button in the 'Synthesis' frame will remove all generated file for the selected synthesis tool.

Note that the Libero toolchain actually performs both simulation, synthesis and place&route. I has been added to the 'Synthesis' menu for convenience.

4.9.4 Place & Route

Place & route is supported for three FPGA tool-chains: Actel Designer, Altera Quartus and Xilinx ISE. Selecting the tool-chain is done through the menu button in the frame labeled 'Place & Route'. Again, the 'Batch' check-button controls if the tool-chain will be launched interactively or run in batch mode. Note that the selection of synthesis tool affects on how place&route is performed. For instance: if synplify has been selected for synthesis and the Xilinx ISE tool is launched, it will use a project file where the edif netlist from synplify is referenced. If the XST synthesis tool has been selected instead, the .ngc netlist from XST would have been used.

The 'Clean' button in the 'Place&Route' frame will remove all generated file for the selected place&route tool.

4.9.5 Additional functions

Cleaning

The 'Clean' button in each of the three tool frames will remove all generated files for selected tool. This make it possible to for instance clean and rebuild a simulation model without simultaneously removing a generated netlist. Generated files for all tools will be removed when the 'clean all' button is pressed. This will however not removed compile scripts and project files. To remove these as well, use the 'distclean' button.

Generating compile scripts

The compile scripts and project files are normally automatically generated by the make utility when needed by a tool. They can also be created directly through the 'scripts' button.

Xconfig

If the local design is configured through xconfig (leon3/leon5/noelv systems), the xconfig tool can be launched by pressing the 'xconfig' button. The configuration file (config.vhd) is automatically generated if xconfig is exited by saving the new configuration.

FPGA PROM programming

The button 'PROM prog' will generate FPGA prom files for the current board, and program the configuration proms using JTAG. This is currently only supported on Xilinx-based boards. The configuration prom must be reloaded by the FPGA for the new configuration to take effect. Some boards has a special reload button, while others must be power-cycled.

5 GRLIB Design concept

5.1 Introduction

GRLIB is a collection of reusable IP cores, divided on multiple VHDL libraries. Each library provides components from a particular vendor, or a specific set of shared functions or interfaces. Data structures and component declarations to be used in a GRLIB-based design are exported through library specific VHDL packages.

GRLIB is based on the AMBA AHB and APB on-chip buses, which is used as the standard interconnect interface. The implementation of the AHB/APB buses is compliant with the AMBA-2.0 specification, with additional 'sideband' signals for automatic address decoding, interrupt steering and device identification (a.k.a. plug&play support). The AHB and APB signals are grouped according to functionality into VHDL records, declared in the GRLIB VHDL library. The GRLIB AMBA package source files are located in lib/grlib/amba.

All GRLIB cores use the same data structures to declare the AMBA interfaces, and can then easily be connected together. An AHB bus controller and an AHB/APB bridge are also available in the GRLIB library, and allows to assemble quickly a full AHB/APB system.

5.2 AMBA AHB on-chip bus

5.2.1 General

The AMBA Advanced High-performance Bus (AHB) is a multi-master bus suitable to interconnect units that are capable of high data rates, and/or variable latency. A conceptual view is provided in figure 5. The attached units are divided into master and slaves, and controlled by a global bus arbiter.

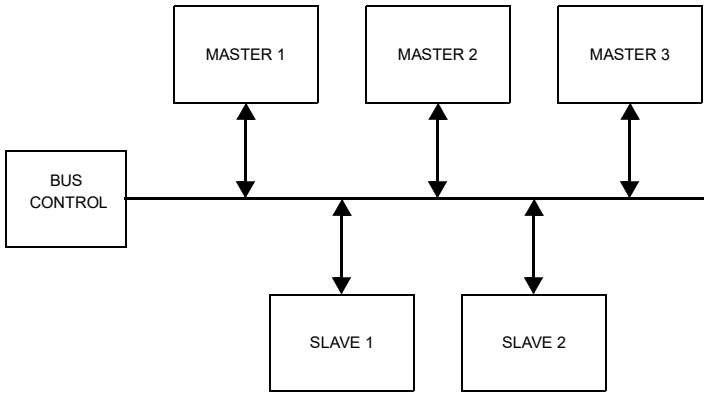


Figure 5. AMBA AHB conceptual view

Since the AHB bus is multiplexed (no tristate signals), a more correct view of the bus and the attached units can be seen in figure 6. Each master drives a set of signals grouped into a VHDL record called ahbmo. The output record of the current bus master is selected by the bus multiplexers and sent to the input record (ahbsi) of all AHB slaves. The output record (ahbso) of the active slave is selected by the bus multiplexer and forwarded to all masters. A combined bus arbiter, address decoder and bus multiplexer controls which master and slave are currently selected.

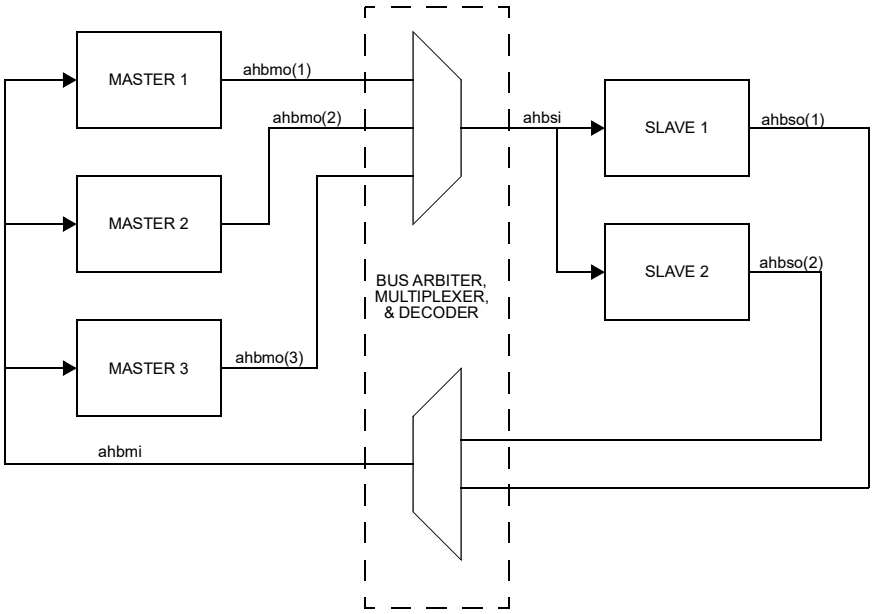


Figure 6. AHB inter-connection view

5.2.2 AHB master interface

The AHB master inputs and outputs are defined as VHDL record types, and are exported through the AMBA package in the GRLIB library:

```
-- AHB master inputs
  type ahb_mst_in_type is record
    hgrant  : std_logic_vector(0 to NAHBMST-1);       -- bus grant
    hready  : std_ulogic;                             -- transfer done
    hresp   : std_logic_vector(1 downto 0);           -- response type
    hrdata  : std_logic_vector(31 downto 0);          -- read data bus
    hirq    : std_logic_vector(NAHBIRQ-1 downto 0);   -- interrupt result bus
  end record;

-- AHB master outputs
type ahb_mst_out_type is record
    hbusreq           : std_ulogic;                         -- bus request
    hlock             : std_ulogic;                         -- lock request
    htrans            : std_logic_vector(1 downto 0); -- transfer type
    haddr             : std_logic_vector(31 downto 0); -- address bus (byte)
    hwrite            : std_ulogic;                         -- read/write
    hsize             : std_logic_vector(2 downto 0); -- transfer size
    hburst            : std_logic_vector(2 downto 0); -- burst type
    hprot             : std_logic_vector(3 downto 0); -- protection control
    hwdata            : std_logic_vector(31 downto 0); -- write data bus
    hirq              : std_logic_vector(NAHBIRQ-1 downto 0);-- interrupt bus
    hconfig           : ahb_config_type;            -- memory access reg.
    hindex            : integer range 0 to NAHBMST-1; -- diagnostic use only
  end record;
```

The elements in the record types correspond to the AHB master signals as defined in the AMBA 2.0 specification, with the addition of three sideband signals: HIRQ, HCONFIG and HINDEX. A typical AHB master in GRLIB has the following definition:

```
library grlib;
use grlib.amba.all;
library ieee;
use ieee.std_logic.all;

entity ahbmaster is
  generic (
    hindex : integer := 0);           -- master bus index
  port (
    reset   : in  std_ulogic;
    clk     : in  std_ulogic;
    ahbmi   : in  ahb_mst_in_type;    -- AHB master inputs
    ahbmo   : out ahb_mst_out_type    -- AHB master outputs
  );
end entity;
```

The input record (AHBMI) is routed to all masters, and includes the bus grant signals for all masters in the vector AHBMI.HGRANT. An AHB master must therefore use a generic that specifies which HGRANT element to use. This generic is of type integer, and typically called HINDEX (see example above).

5.2.3 AHB slave interface

Similar to the AHB master interface, the inputs and outputs of AHB slaves are defined as two VHDL records types:

```
-- AHB slave inputs
  type ahb_slv_in_type is record
    hsel      : std_logic_vector(0 to NAHBSLV-1);     -- slave select
    haddr     : std_logic_vector(31 downto 0);        -- address bus (byte)
    hwrite    : std_ulogic;                           -- read/write
    htrans    : std_logic_vector(1 downto 0);         -- transfer type
    hsize     : std_logic_vector(2 downto 0);         -- transfer size
    hburst    : std_logic_vector(2 downto 0);         -- burst type
    hwdata    : std_logic_vector(31 downto 0);        -- write data bus
    hprot     : std_logic_vector(3 downto 0);         -- protection control
    hready    : std_ulogic;                           -- transfer done
    hmaster   : std_logic_vector(3 downto 0);         -- current master
    hmastlock : std_ulogic;                           -- locked access
    hbsel     : std_logic_vector(0 to NAHBCFG-1);     -- bank select
hirq      : std_logic_vector(NAHBIRQ-1 downto 0); -- interrupt result bus
  end record;

-- AHB slave outputs
  type ahb_slv_out_type is record
    hready     : std_ulogic;                            -- transfer done
    hresp      : std_logic_vector(1 downto 0);          -- response type
    hrdata     : std_logic_vector(31 downto 0);         -- read data bus
    hsplit     : std_logic_vector(15 downto 0);         -- split completion
    hirq       : std_logic_vector(NAHBIRQ-1 downto 0); -- interrupt bus
    hconfig    : ahb_config_type;                       -- memory access reg.
    hindex     : integer range 0 to NAHBSLV-1;          -- diagnostic use only
  end record;
```

The elements in the record types correspond to the AHB slaves signals as defined in the AMBA 2.0 specification, with the addition of four sideband signals: HSEL, HIRQ, HCONFIG and HINDEX. A typical AHB slave in GRLIB has the following definition:

```
library grlib;
use grlib.amba.all;
library ieee;
use ieee.std_logic.all;

entity ahbslave is
  generic (
    hindex : integer := 0);           -- slave bus index
  port (
    reset   : in  std_ulogic;
    clk     : in  std_ulogic;
    abhsi   : in  ahb_slv_in_type;     -- AHB slave inputs
    ahbso   : out ahb_slv_out_type     -- AHB slave outputs
  );
end entity;
```

The input record (ahbsi) is routed to all slaves, and include the select signals for all slaves in the vector ahbsi.hsel. An AHB slave must therefore use a generic that specifies which hsel element to use. This generic is of type integer, and typically called HINDEX (see example above).

5.2.4 AHB bus control

GRLIB AMBA package provides a combined AHB bus arbiter (AHBCTRL), address decoder and bus multiplexer. It receives the ahbmo and ahbso records from the AHB units, and generates ahbmi and ahbsi as indicated in figure 6. The bus arbitration function will generate which of the ahbmi.hgrant elements will be driven to indicate the next bus master. The address decoding function will drive one of the ahbsi.hsel elements to indicate the selected slave. The bus multiplexer function will select which master will drive the ahbsi signal, and which slave will drive the ahbmo signal.

5.2.5 AHB bus index control

The AHB master and slave output records contain the sideband signal HINDEX. This signal is used to verify that the master or slave is driving the correct element of the ahbso/ahbmo buses. The generic HINDEX that is used to select the appropriate hgrant and hsel is driven back on ahbmo.hindex and ahbso.hindex. The AHB controller then checks that the value of the received HINDEX is equal to the bus index. An error is issued dunring simulation if a missmatch is detected.

5.2.6 Support for wide AHB data buses

5.2.6.1 Overview

The cores in GRLIB and the GRLIB infrastructure can be configured to support an AMBA AHB data bus width of 32, 64, 128, and in several cases 256 bits. The default AHB bus width is 32 bits and AHB buses with data vectors having widths over 32 bits will in this section be referred to as wide AHB buses.

Changing the AHB bus width can increase performance, but may also increase the area requirements of a design, depending on the synthesis tool used and the type of cores instantiated. Manual modification of the GRLIB CONFIG package is required to enable support for wide AHB buses. Alternatively, a local version of the GRLIB CONFIG package can be placed in the current template design, overriding the settings in the global GRLIB CONFIG package.

When modifying the system's bus width, care should be taken to verify that all cores have been instantiated with the correct options with regards to support for wide buses.

Note that the APB bus in GRLIB will always be 32-bits, regardless of the AHB data bus width.

5.2.6.2 Implementation of support for wide AHB buses

To support wide buses, the AHB VHDL records that specify the GRLIB AMBA AHB interface have their data vector lengths defined by a constant, CFG_AHBDW, defined in the GRLIB CONFIG VHDL package.

Using a wide AHB bus places additional requirements on the cores in a design; The cores should drive the extra positions in the AHB data vector in order to minimize the amount of undriven signals in the design, and to allow synthesis tool optimisations for cores that do not support AMBA accesses larger than word accesses. The cores are also required to select and drive the applicable byte lanes, depending on access size and address.

In order to minimize the amount of undriven signals, all GRLIB AHB cores drive their AHB data vector outputs via a subprogram, *ahbdrivedata(..)*, defined in the GRLIB AMBA VHDL package. The subprogram replicates its input so that the whole AHB data vector is driven. Since data is present on all byte lanes, the use of this function also ensures that data will be present on the correct byte lanes.

The AMBA 2.0 Specification requires that cores select their data from the correct byte lane. For instance, when performing a 32-bit access in a system with a 64-bit wide bus, valid data will be on positions 63:32 of the data bus if bit 2 of the address is 0, otherwise the valid data will be on positions 31:0. In order to ease adding support for variable buses, the GRLIB AMBA VHDL package includes subprograms, *ahbread*(...)*, for reading the AMBA AHB data vectors, hereafter referred to as AHB read subprograms. These subprograms exists in two variants; The first variant takes an address argument so that the subprogram is able to select the valid byte lanes of the data vector. This functionality is not always enabled, as will be explained below. The second variant does not require the address argument, and always returns the low slice of the AHB data vector.

Currently the majority of the GRLIB AHB cores use the functions without the address argument, and therefore the cores are only able to read the low part of the data vector. The cores that only read the low part of the AHB data vector are not fully AMBA 2.0 compatible with regard to wide buses. How-

ever, this does not affect the use of a wide AHB bus in a GRLIB system, since all GRLIB cores places valid data on the full AHB data vector. As adoption of wide buses become more widespread, the cores will be updated so that they are able to select the correct byte lanes.

The GRLIB AHB controller core, AHBCTRL, is a central piece of the bus infrastructure. The AHB controller includes a multiplexer of the width defined by the AMBA VHDL package constant AHBDW. The core also has a generic that decides if the controller should perform additional AMBA data multiplexing. Data multiplexing is discussed in the next section.

5.2.6.3 AMBA AHB data multiplexing

Almost all GRLIB cores drive valid data on all lanes of the data bus, some exceptions exist, such as the cores in the AMBA Test Framework). Since the *ahbdrivedata(..)* subprogram duplicates all data onto the wider bus, all cores will be compliant to the AMBA standard with regards to placing valid data on the correct lane in the AHB data vector.

As long as there are only GRLIB cores in a design, the cores can support wide AHB buses by only reading the low slice of the AHB data vectors, which is the case for most cores, as explained in the section above. However, if a core that only drives the required part of the data vector is introduced in a design there is a need for support to allow the GRLIB cores to select the valid part of the data.

The current implementation has two ways of accomplishing this:

Set the ACDM generic of AHBCTRL to 1. When this option is enabled the AHB controller will check the size and address of each access and propagate the valid part of the data on the entire AHB data bus bus. The smallest portion of the slice to select and duplicate is 32-bits. This means that valid data for a a byte or halfword access will not be present on all byte lanes, however the data will be present on all the required byte lanes.

Set the CFG_AHB_ACDM constant to 1 in the GRLIB CONFIG VHDL package. This will make the AHB read subprograms look at the address and select the correct slice of the incoming data vector. If a core uses one of the AHB read subprograms that does not have the address argument there will be a failure asserted. If CFG_AHB_ACDM is 0, the AHB read subprograms will return the low slice of the data vector. With CFG_AHB_ACDM set to 1, a core that uses the subprograms with the correct address argument will be fully AMBA compliant and can be used in non-GRLIB environments with bus widths exceeding 32 bits.

Note that it is unnecessary to enable both of these options in the same system.

5.2.6.4 IP cores with support for wide buses

Several cores in the IP library make use of the wide buses, see the core documentation in the GRLIB IP Cores User's Manual to determine the state of wide bus support for specific cores. Most cores in GRLIB can be used in a system with wide AHB buses, however they do not all exploit the advantages of a wider bus and cores such as trace buffers may not allow tracing of the full data width. Please see the IP core documentation for supported bus widths.

5.2.6.5 GRLIB CONFIG Package

The GRLIB configuration package contains a constant the controls the maximum allowed AHB bus width in the system, see section 5.6.

5.2.6.6 Issues with wide AHB buses

A memory controller may not be able to respond all access sizes. With the current scheme the user of the system must keep track of which areas that can be accessed with accesses larger then word accesses. For instance, if SVGACTRL is configured to use 4WORD accesses and the designs has a DDR2SPA core and a MCTRL core in the system, the SVGACTRL will only receive correct data if the framebuffer is placed in the DDR2 memory area.

Special care must be taken when using wide buses so that the core specific settings for wider buses matches the intended use for the cores. Please see the IP core documentation for supported bus widths.

5.3 AHB plug&play configuration

5.3.1 General

The GRLIB implementation of the AHB bus includes a mechanism to provide plug&play support. The plug&play support consists of three parts: identification of attached units (masters and slaves), address mapping of slaves, and interrupt routing. The plug&play information for each AHB unit consists of a configuration record containing eight 32-bit words. The first word is called the identification register and contains information on the device type and interrupt routing. The last four words are called bank address registers, and contain address mapping information for AHB slaves. The remaining three words are currently not assigned and could be used to provide core-specific configuration information.

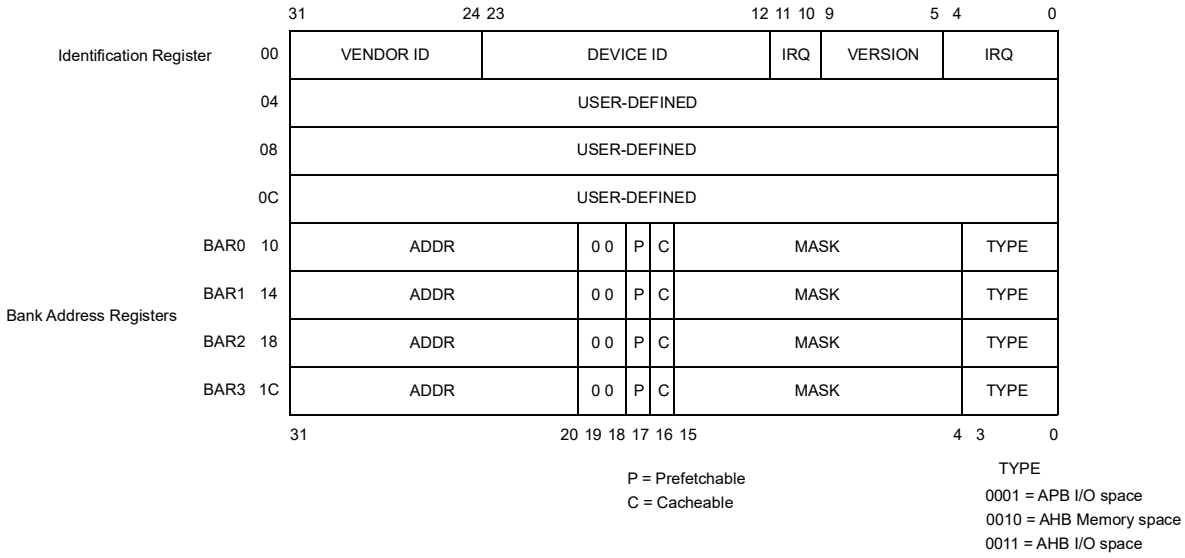


Figure 7. AHB plug&play configuration layout

The plug&play information for all attached AHB units appear as a read-only table mapped on a fixed address of the AHB, typically at 0xFFFFF000. The configuration records of the AHB masters appear in 0xFFFFF000 - 0xFFFFF800, while the configuration records for the slaves appear in 0xFFFFF800 - 0xFFFFFFFC. Since each record is 8 words (32 bytes), the table has space for 64 masters and 64 slaves. A plug&play operating system (or any other application) can scan the configuration table and automatically detect which units are present on the AHB bus, how they are configured, and where they are located (slaves).

The top four words of the plug&play area (0xFFFFFFF0 - 0xFFFFFFFF) may contain device specific information such as GRLIB build ID, a (SoC) device ID and the endianness of the system. If present, this information shadows the bank address registers of the last slave record, limiting the number of slaves on one bus to 63. All systems that use the GRLIB AHB controller have the library's build ID in the least significant half-word, and a (SoC) device ID in the most significant half-word, of the word at address 0xFFFFFFF0. The endianness is reported via the least significant bit of the word at address 0xFFFFFFF4 (0b: Big endian; 1b: Little Endian). The contents of the top four words is described in the AHB controller's IP core manual.

The configuration record from each AHB unit is sent to the AHB bus controller via the HCONFIG signal. The bus controller creates the configuration table automatically, and creates a read-only memory area at the desired address (default 0xFFFFF000). Since the configuration information is fixed, it can be efficiently implemented as a small ROM or with relatively few gates. A debug module, present within the AHB bus controller, can be used to print the configuration table to the console during simulation, which is useful for debugging. A typical example is provided below:

```
VSIM 1> run
.
.
# LEON3 Actel PROASIC3-1000 Demonstration design
# GRLIB Version 1.0.16, build 2460
# Target technology: proasic3 ,  memory library: proasic3
# ahbctrl: AHB arbiter/multiplexer rev 1
# ahbctrl: Common I/O area disabled
# ahbctrl: AHB masters: 2, AHB slaves: 8
# ahbctrl: Configuration area at 0xfffff000, 4 kbyte
# ahbctrl: mst0: Frontgrade Gaisler        Leon3 SPARC V8 Processor
# ahbctrl: mst1: Frontgrade Gaisler        AHB Debug UART
# ahbctrl: slv0: European Space Agency   Leon2 Memory Controller
# ahbctrl:       memory at 0x00000000, size 512 Mbyte, cacheable, prefetch
# ahbctrl:       memory at 0x20000000, size 512 Mbyte
# ahbctrl:       memory at 0x40000000, size 1024 Mbyte, cacheable, prefetch
# ahbctrl: slv1: Frontgrade Gaisler        AHB/APB Bridge
# ahbctrl:       memory at 0x80000000, size 1 Mbyte
# ahbctrl: slv2: Frontgrade Gaisler        Leon3 Debug Support Unit
# ahbctrl:       memory at 0x90000000, size 256 Mbyte
# apbctrl: APB Bridge at 0x80000000 rev 1
# apbctrl: slv0: European Space Agency   Leon2 Memory Controller
# apbctrl:       I/O ports at 0x80000000, size 256 byte
# apbctrl: slv1: Frontgrade Gaisler        Generic UART
# apbctrl:       I/O ports at 0x80000100, size 256 byte
# apbctrl: slv2: Frontgrade Gaisler        Multi-processor Interrupt Ctrl.
# apbctrl:       I/O ports at 0x80000200, size 256 byte
# apbctrl: slv3: Frontgrade Gaisler        Modular Timer Unit
# apbctrl:       I/O ports at 0x80000300, size 256 byte
# apbctrl: slv7: Frontgrade Gaisler        AHB Debug UART
# apbctrl:       I/O ports at 0x80000700, size 256 byte
# apbctrl: slv11: Frontgrade Gaisler        General Purpose I/O port
# apbctrl:       I/O ports at 0x80000b00, size 256 byte
# grgpio11: 8-bit GPIO Unit rev 0
# gptimer3: GR Timer Unit rev 0, 8-bit scaler, 2 32-bit timers, irq 8
# irqmp: Multi-processor Interrupt Controller rev 3, #cpu 1
# apbuart1: Generic UART rev 1, fifo 1, irq 2
# ahbuart7: AHB Debug UART rev 0
# dsu3_2: LEON3 Debug support unit + AHB Trace Buffer, 1 kbytes
# leon3_0: LEON3 SPARC V8 processor rev 0
# leon3_0: icache 1*2 kbyte, dcache 1*2 kbyte
```

5.3.2 Device identification

The Identification Register contains three fields to identify uniquely an attached AHB unit: the vendor ID, the device ID, and the version number. The vendor ID is a unique number assigned to an IP vendor or organization. The device ID is a unique number assigned by a vendor to a specific IP core. The device ID is not related to the core's functionality. The version number can be used to identify (functionally) different versions of the unit.

The vendor IDs are declared in a package located at lib/grlib/amba/devices.vhd. Vendor IDs are provided by Frontgrade Gaisler. The following ID's are currently assigned:

Vendor	ID
Frontgrade Gaisler	0x01
Pender Electronic Design	0x02
European Space Agency	0x04
Astrium EADS	0x06
OpenChip.org	0x07
OpenCores.org	0x08
Various contributions	0x09
DLR	0x0A
Eonic BV	0x0B
Telecom ParisTech	0x0C
DTU Space	0x0D

TABLE 66. Vendor ID assignment

Vendor	ID
Barcelona Supercomputing Center	0x0E
Radionor	0x0F
Gleichmann Electronics	0x10
Menta	0x11
Sun Microsystems	0x13
Movidia	0x14
L3 Technologies	0x15
Orbita	0x17
Siemens AG	0x1A
Synopsys	0x21
NASA	0x22
NIIET	0x23
JHUAPL	0x24
JHUAPL	0x25
JHUAPL	0x26
JHUAPL	0x27
SemiBlocks B.V.	0x28
NEC Corporation	0x29
Honeywell International	0x2A
S3	0x31
Thales Alenia Space	0xA5
Recore Systems BV	0xA6
ÅAC Microtec	0xAA
UC Berkeley	0xAB
Microsemi/Actel Corporation	0xAC
Beyond Gravity	0xAD
Applecore	0xAE
TU Braunschweig C3E	0xC3
CBK PAN	0xC8
Caltech	0xCA
Ceton	0xCB
SSTL	0xD6
Embeddit	0xEA
NASA GSFC	0xFC
AZST	0xFE

TABLE 66. Vendor ID assignment

Vendor ID 0x00 is reserved to indicate that no core is present. Unused slots in the configuration table will have Identification Register set to 0. IP cores added to GRLIB must only use vendor ID 0x09 to prevent that the user IP core is detected as an IP core from another vendor. Vendor IDs for organizations can be requested via e-mail to support@gaisler.com.

5.3.3 Address decoding

The address mapping of AHB slaves in GRLIB is designed to be distributed, i.e. not rely on a shared static address decoder which must be modified as soon as a slave is added or removed. The GRLIB AHB bus controller, which implements the address decoder, will use the configuration information received from the slaves on HCONFIG to automatically generate the slave select signals (HSEL). When a slave is added or removed during the design, the address decoding function is automatically updated without requiring manual editing.

The AHB address range for each slave is defined by its Bank Address Registers (BAR). Address decoding is performed by comparing the 12-bit ADDR field in the BAR with part of the AHB address (HADDR). There are two types of banks defined for the AHB bus: AHB memory bank and AHB I/O bank. The AHB address decoding is done differently for the two types.

For AHB memory banks, the address decoding is performed by comparing the 12-bit ADDR field in the BAR with the 12 most significant bits in the AHB address (HADDR(31:20)). If equal, the corresponding HSEL will be generated. This means that the minimum address range occupied by an AHB memory bank is 1 MByte. To allow for larger address ranges, only the bits set in the MASK field of the BAR are compared. Consequently, HSEL will be generated when the following equation is true:

```
((BAR.ADDR xor HADDR[31:20]) and BAR.MASK) = 0
```

As an example, to decode a 16 MByte AHB memory bank at address 0x24000000, the ADDR field should be set to 0x240, and the MASK to 0xFF0. Note: if MASK = 0, the BAR is disabled rather than occupying the full AHB address range.

For AHB I/O banks, the address decoding is performed by comparing the 12-bit ADDR field in the BAR with 12 bits in the AHB address (HADDR(19:8)). If equal, the corresponding HSEL will be generated. This means that the minimum address range occupied by an AHB I/O bank is 256 Byte. To allow for larger address ranges, only the bits set in the MASK field of the BAR are compared. Consequently, HSEL will be generated when the following equation is true:

```
((BAR.ADDR xor HADDR[19:8]) and BAR.MASK) = 0
```

The 12 most significant bits in the AHB address (HADDR(31:20)) are always fixed to 0xFFF, effectively placing all AHB I/O banks in the 0xFFF00000-0xFFFFEFFF address space. As an example, to decode an 4 kByte AHB I/O bank at address 0xFFF24000, the ADDR field should be set to 0x240, and the MASK to 0xFF0. Note: if MASK = 0, the BAR is disabled rather than occupying the full AHB I/O address range.

The AHB slaves in GRLIB define the value of their ADDR and MASK fields through generics. This allows to choose the address range for each slave when it is instantiated, without having to modify a central decoder or the slave itself. Below is an example of a component declaration of an AHB RAM memory, and how it can be instantiated:

```
component ahbram
  generic (
    hindex : integer := 0;                          -- AHB slave index
    haddr  : integer := 0;
    hmask  : integer := 16#fff#);
  port (
    rst    : in  std_ulogic;
    clk    : in  std_ulogic;
    ahbsi  : in  ahb_slv_in_type;                    -- AHB slave input
    ahbso  : out ahb_slv_out_type);                  -- AHB slave output
end component;

ram0 : ahbram
  generic map (hindex => 1, haddr => 16#240#, hmask => 16#FF0#)
  port map (rst, clk, ahbsi, ahbso(1));
```

An AHB slave can have up to four address mapping registers, thereby decode four independent areas in the AHB address space. HSEL is asserted when any of the areas is selected. To know which particular area was selected, the ahbsi record contains the additional bus signal HBSEL(0:3). The elements in HBSEL(0:3) are asserted if the corresponding to BAR(0-3) caused HSEL to be asserted. HBSEL is only valid when HSEL is asserted. For example, if BAR1 caused HSEL to be asserted, the HBSEL(1) will be asserted simultaneously with HSEL.

5.3.4 Cacheability

In processor-based systems without an MMU, the cacheable areas are typically defined statically in the cache controllers. The LEON processors build the cachebility table automatically during synthesis, using the cacheability information in the AHB configuration records. In this way, the cacheability settings always reflect the current configuration.

For systems with an MMU, the cacheability information can be read out by from the configuration records through software. This allows the operating system to build an MMU page table with proper cacheable-bits set in the page table entries.

5.3.5 Interrupt steering

GRLIB provides a unified interrupt handling scheme by adding 32 interrupt signals (HIRQ) to the AHB bus, both as inputs and outputs. An AHB master or slave can drive as well as read any of the interrupts.

The output of each master includes all 32 interrupt signals in the vector ahbmo.hirq. An AHB master must therefore use a generic that specifies which HIRQ element to drive. This generic is of type integer, and typically called HIRQ (see example below).

```
component ahbmaster is
  generic (
    hindex : integer := 0;            -- master index
    hirq : integer := 0);           -- interrupt index
  port (
    reset   : in  std_ulogic;
    clk     : in  std_ulogic;
    hmsti   : in  ahb_mst_in_type;     -- AHB master inputs
    hmsto   : out ahb_mst_out_type     -- AHB master outputs
  );
end component;

master1 : ahbmaster
  generic map (hindex => 1, hirq => 1)
  port map (rst, clk, hmsti, hmsto(1));
```

The same applies to the output of each slave which includes all 32 interrupt signals in the vector ahbso.hirq. An AHB slave must therefore use a generic that specifies which HIRQ element to drive. This generic is of type integer, and typically called HIRQ (see example below).

```
component ahbslave
  generic (
    hindex : integer := 0;            -- slave index
    hirq : integer := 0);           -- interrupt index
  port (
    rst     : in  std_ulogic;
    clk     : in  std_ulogic;
    hslvi   : in  ahb_slv_in_type;     -- AHB slave inputs
    hslvo   : out ahb_slv_out_type);   -- AHB slave outputs
end component;

slave2 : ahbslave
  generic map (hindex => 2, hirq => 2)
  port map (rst, clk, hslvi, hslvo(1));
```

The AHB bus controller in the GRLIB provides interrupt combining. For each element in HIRQ, all the ahbmo.hirq signals from the AHB masters and all the ahbso.hirq signals from the AHB slaves are logically OR-ed. The combined result is output both on ahbmi.hirq (routed back to the AHB masters) and ahbsi.hirq (routed back to the AHB slaves). Consequently, the AHB masters and slaves share the same 32 interrupt signals.

An AHB unit that implements an interrupt controller can monitor the combined interrupt vector (either ahbsi.hirq or ahbmi.hirq) and generate the appropriate processor interrupt.

5.4 AMBA APB on-chip bus

5.4.1 General

The AMBA Advanced Peripheral Bus (APB) is a single-master bus suitable to interconnect units of low complexity which require only low data rates. An APB bus is interfaced with an AHB bus by means of a single AHB slave implementing the AHB/APB bridge. The AHB/APB bridge is the only APB master on one specific APB bus. More than one APB bus can be connected to one AHB bus, by means of multiple AHB/APB bridges. A conceptual view is provided in figure 8.

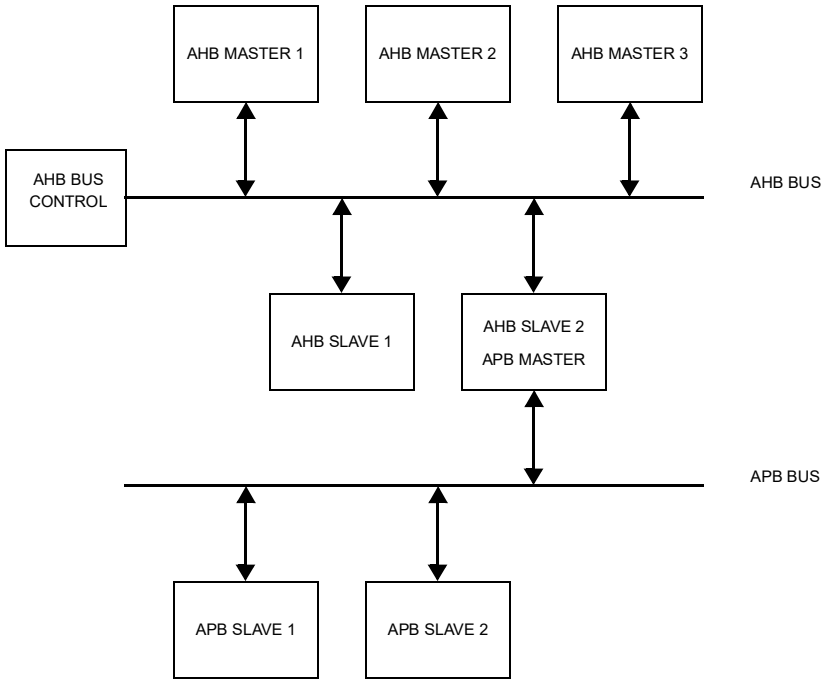


Figure 8. AMBA AHB/APB conceptual view

Since the APB bus is multiplexed (no tristate signals), a more correct view of the bus and the attached units can be seen in figure 9. The access to the AHB slave input (AHBI) is decoded and an access is made on APB bus. The APB master drives a set of signals grouped into a VHDL record called APBI which is sent to all APB slaves. The combined address decoder and bus multiplexer controls which slave is currently selected. The output record (APBO) of the active APB slave is selected by the bus multiplexer and forwarded to AHB slave output (AHBO).

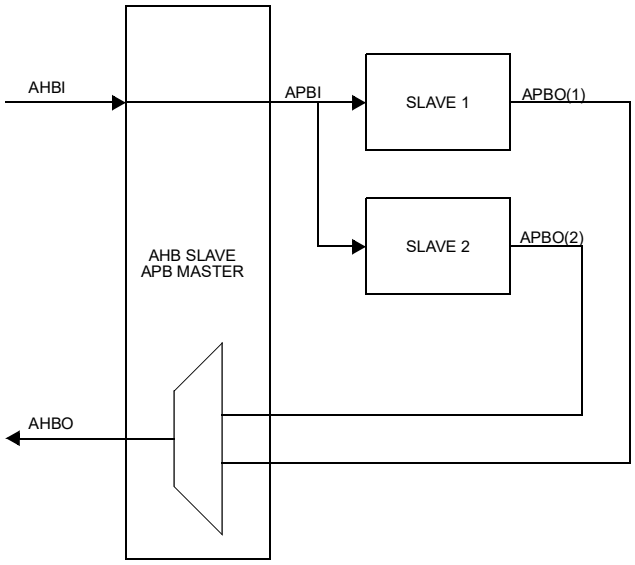


Figure 9. APB inter-connection view

5.4.2 APB slave interface

The APB slave inputs and outputs are defined as VHDL record types, and are exported through the TYPES package in the GRLIB AMBA library:

```
-- APB slave inputs
  type apb_slv_in_type is record
    psel    : std_logic_vector(0 to NAPBSLV-1);       -- slave select
    penable : std_ulogic;                             -- strobe
    paddr   : std_logic_vector(31 downto 0);          -- address bus (byte)
    pwrite  : std_ulogic;                             -- write
    pwdata  : std_logic_vector(31 downto 0);          -- write data bus
    pirq    : std_logic_vector(NAHBIRQ-1 downto 0);   -- interrupt result bus
  end record;

-- APB slave outputs
  type apb_slv_out_type is record
    prdata   : std_logic_vector(31 downto 0);           -- read data bus
    pirq     : std_logic_vector(NAHBIRQ-1 downto 0);    -- interrupt bus
    pconfig  : apb_config_type;                         -- memory access reg.
    pindex   : integer range 0 to NAPBSLV -1;           -- diag use only
  end record;
```

The elements in the record types correspond to the APB signals as defined in the AMBA 2.0 specification, with the addition of three sideband signals: PCONFIG, PIRQ and PINDEX. A typical APB slave in GRLIB has the following definition:

```
library grlib;
use grlib.amba.all;
library ieee;
use ieee.std_logic.all;

entity apbslave is
  generic (
    pindex  : integer := 0);           -- slave bus index
  port (
    rst     : in  std_ulogic;
    clk     : in  std_ulogic;
    apbi    : in  apb_slv_in_type;     -- APB slave inputs
    apbo    : out apb_slv_out_type     -- APB slave outputs
  );
end entity;
```

The input record (APBI) is routed to all slaves, and include the select signals for all slaves in the vector APBI.PSEL. An APB slave must therefore use a generic that specifies which PSEL element to use. This generic is of type integer, and typically called PINDEX (see example above).

5.4.3 AHB/APB bridge

GRLIB provides a combined AHB slave, APB bus master, address decoder and bus multiplexer. It receives the AHBI and AHBO records from the AHB bus, and generates APBI and APBO records on the APB bus. The address decoding function will drive one of the APBI.PSEL elements to indicate the selected APB slave. The bus multiplexer function will select from which APB slave data will be taken to drive the AHBI signal. A typical APB master in GRLIB has the following definition:

```
library IEEE;
use IEEE.std_logic_1164.all;
library grlib;
use grlib.amba.all;

entity apbmst is
  generic (
    hindex  : integer := 0;            -- AHB slave bus index
  );
  port (
    rst     : in  std_ulogic;
    clk     : in  std_ulogic;
    ahbi    : in  ahb_slv_in_type;     -- AHB slave inputs
    ahbo    : out ahb_slv_out_type;    -- AHB slave outputs
    apbi    : out apb_slv_in_type;     -- APB master inputs
    apbo    : in  apb_slv_out_vector   -- APB master outputs
  );
end;
```

5.4.4 APB bus index control

The APB slave output records contain the sideband signal PINDEX. This signal is used to verify that the slave is driving the correct element of the AHBPO bus. The generic PINDEX that is used to select the appropriate PSEL is driven back on APBO.PINDEX. The APB controller then checks that the value of the received PINDEX is equal to the bus index. An error is issued during simulation if a mismatch is detected.

5.5 APB plug&play configuration

5.5.1 General

The GRLIB implementation of the APB bus includes the same type of mechanism to provide plug&play support as for the AHB bus. The plug&play support consists of three parts: identification of attached slaves, address mapping, and interrupt routing. The plug&play information for each APB slave consists of a configuration record containing two 32-bit words. The first word is called the identification register and contains information on the device type and interrupt routing. The last word is the bank address register (BAR) and contains address mapping information for the APB slave. Only a single BAR is defined per APB slave. An APB slave is neither prefetchable nor cacheable.

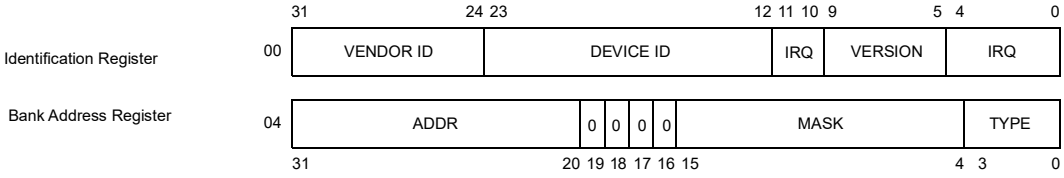


Figure 10. APB plug&play configuration layout

All addressing of the APB is referenced to the AHB address space. The 12 most significant bits of the AHB bus address are used for addressing the AHB slave of the AHB/APB bridge, leaving the 20 least significant bits for APB slave addressing.

The plug&play information for all attached APB slaves appear as a read-only table mapped on a fixed address of the AHB, typically at 0x---FF000. The configuration records of the APB slaves appear in 0x---FF000 - 0x---FFFFF on the AHB bus. Since each record is 2 words (8 bytes), the table has space for 512 slaves on a signle APB bus. A plug&play operating system (or any other application) can scan the configuration table and automatically detect which units are present on the APB bus, how they are configured, and where they are located (slaves).

The configuration record from each APB unit is sent to the APB bus controller via the PCONFIG signal. The bus controller creates the configuration table automatically, and creates a read-only memory area at the desired address (default 0x---FF000). Since the configuration information is fixed, it can be efficiently implemented as a small ROM or with relatively few gates. A debug module, present within the APB bus controller, can be used to print the configuration table to the console during simulation, which is useful for debugging

5.5.2 Device identification

The APB bus uses same type of Identification Register as previously defined for the AHB bus.

5.5.3 Address decoding

The address mapping of APB slaves in GRLIB is designed to be distributed, i.e. not rely on a shared static address decoder which must be modified as soon as a slave is added or removed. The GRLIB APB master, which implements the address decoder, will use the configuration information received from the slaves on PCONFIG to automatically generate the slave select signals (PSEL). When a slave is added or removed during the design, the address decoding function is automatically updated without requiring manual editing.

The APB address range for each slave is defined by its Bank Address Registers (BAR). There is one type of banks defined for the APB bus: APB I/O bank. Address decoding is performed by comparing the 12-bit ADDR field in the BAR with 12 bits in the AHB address (HADDR(19:8)). If equal, the corresponding PSEL will be generated. This means that the minimum address range occupied by an APB I/O bank is 256 Byte. To allow for larger address ranges, only the bits set in the MASK field of the BAR are compared. Consequently, PSEL will be generated when the following equation is true:

```
((BAR.ADDR xor HADDR[19:8]) and BAR.MASK) = 0
```

As an example, to decode an 4 kByte AHB I/O bank at address 0x---24000, the ADDR field should be set to 0x240, and the MASK to 0xFF0. Note that the 12 most significant bits of AHBI.HADDR are

used for addressing the AHB slave of the AHB/APB bridge, leaving the 20 least significant bits for APB slave addressing.

As for AHB slaves, the APB slaves in GRLIB define the value of their ADDR and MASK fields through generics. This allows to choose the address range for each slave when it is instantiated, without having to modify a central decoder or the slave itself. Below is an example of a component declaration of an APB I/O unit, and how it can be instantiated:

```
component apbio
  generic (
    pindex : integer := 0;
    paddr  : integer := 0;
    pmask  : integer := 16#fff#);
  port (
    rst    : in  std_ulogic;
    clk    : in  std_ulogic;
    apbi   : in  apb_slv_in_type;
    apbo   : out apb_slv_out_type);
end component;

io0 : apbio
  generic map (pindex => 1, paddr => 16#240#, pmask => 16#FF0#)
  port map (rst, clk, apbi, apbo(1));
```

5.5.4 Interrupt steering

GRLIB provides a unified interrupt handling scheme by also adding 32 interrupt signals (PIRQ) to the APB bus, both as inputs and outputs. An APB slave can drive as well as read any of the interrupts. The output of each slave includes all 32 interrupt signals in the vector APBO.PIRQ. An APB slave must therefore use a generic that specifies which PIRQ element to drive. This generic is of type integer, and typically called PIRQ (see example below).

```
component apbslave
  generic (
    pindex : integer := 0;            -- slave index
    pirq   : integer := 0);           -- interrupt index
  port (
    rst    : in  std_ulogic;
    clk    : in  std_ulogic;
    apbi   : in  apb_slv_in_type;     -- APB slave inputs
    apbo   : out apb_slv_out_type);   -- APB slave outputs
end component;

slave3 : apbslave
  generic map (pindex => 1, pirq => 2)
  port map (rst, clk, pslvi, pslvo(1));
```

The AHB/APB bridge in the GRLIB provides interrupt combining, and merges the APB-generated interrupts with the interrups bus on the AHB bus. This is done by OR-ing the 32-bit interrupt vectors from each APB slave into one joined vector, and driving the combined value on the AHB slave output bus (AHBSO.HIRQ). The APB interrupts will then be merged with the AHB interrupts. The resulting interrupt vector in available on the AHB slave input (AHBSI.HIRQ), and is also driven on the APB slave inputs (APBI.PIRQ) by the AHB/APB bridge. Each APB slave (as well as AHB slave) thus sees the combined AHB/APB interrupts. An interrupt controller can then be placed either on the AHB or APB bus and still monitor all interrupts.

5.6 Endianness configuration

The majority of the GRLIB IP cores have been developed for LEON systems based on the SPARC architecture. Since the SPARC architecture is big endian, that is the default endianness for GRLIB IP cores. Cores that does not support a change of endianness therefore supports big endian.

To support both little endian systems and big endian systems, the GRLIB CONFIG VHDL package contains a constant to configure the endianness of the system, see section 5.6. The implementation of how the constant affects each IP core differ and it is important to read the IP core manual to understand how the core behaviour changes with endianness.

In some cases having a single configuration constant for all GRLIB IP is not flexible enough, and therefore an additional scheme has been implemented where the endianness configuration is fed as a generic into the ahb controller of the bus and then distributed to the IP cores via sideband signals in the AMBA records. The default value of this generic is equal to the constant in the GRLIB configuration package which makes this scheme backward compatible for systems using only the constant, however not all IP cores support the sideband signal scheme. Note that this does still not allow mixing

big-endian and little-endian cores on the same AHB bus, these need to be on completely separated AMBA buses.

In summary:

- For purely big-endian systems (systems using LEON processor, or processor-less) all GRLIB IP will support big-endian operation (with possible exception of NOEL-V specific peripherals) and no special consideration for endian-ness needs to be taken in the design.
- For purely little-endian systems (systems using NOELV processor, or processor-less), a local GRLIB configuration package needs to exist for the design with the endian-ness setting set to little endian and the IP cores used need to be checked (in the respective IP core manual) to support little-endian operation.
- For systems containing both big-endian and little-endian subsystems, the ahbctrl instances in the design should have the ahbendian generic set and IP cores used need to be checked (in the respective IP core manual) to support endian-ness configuration via the AMBA sideband signals.

Registers and descriptors are intended to be accessed using 32-bit accesses by software. This will allow the same code to work regardless of endianness. For data buffers fetched by DMA engines, endianness behavior varies between IP cores. In most cases, data buffers are considered byte buffers and will be consistent if accessed by software as a byte array. When using other access sizes to access descriptors and data in memory, the endianness of the system will become visible to software and has to be managed on software level. Consult the IP core documentation for information about how each IP core handles system wide endianness.

It's recommended to test running software through simulation or FPGA prototyping to ensure the endianness of the system is fully understood.

5.6.1 APB accesses

Accesses to registers mapped on the APB are always 32 bit wide and no endianness conversions are done. Consult the manual of each IP core to determine the layout of registers.

5.6.2 AMBA plug&play

The endianness of the system can be found in the plug&play area.

The AMBA plug&play areas should be read using 32-bit accesses in order to get consistent data.

5.7 GRLIB configuration package

The location of the global GRLIB CONFIG package is in *lib/grlib/stdlib/config.vhd*. This file contains the settings for the wide AHB buses, as described in the previous sections, and some additional global parameters.

This package can be replaced by a local version by setting the variable GRLIB_CONFIG in the Makefile of a template design to the location of an alternative version. When the simulation and synthesis scripts are built, the alternative CONFIG package will be used instead of the global one. The the variable GRLIB_CONFIG is modified, the scripts have to be re-built for the new value to take effect.

The GRLIB configuration package contains the constants listed in table 67.

Constant	Description
CFG_AHBDW	Selects the maximum AHB data width to be used in the system. Note that some cores may not take advantage of the full bus width. A subset of IP cores may not support bus sizes larger than 128 bits, leading to build errors.

TABLE 67. GRLIB configuration package constants

Constant	Description
CFG_AHB_ACDM	Enable AMBA compliant data multiplexing in cores that support this.
GRLIB_CONFIG_ARRAY	Array of configuration values that enable different types of functionality in the library. The available values together with short descriptions can be seen in the file *lib/grlib/stdlib/config_types.vhd*. The available settings are also described in table 68.

TABLE 67. GRLIB configuration package constants

GRLIB_CONFIG_ARRAY(Constant)	Description
grlib_debug_level	Controls (simulation) debug output from TECHMAP layer
grlib_debug_mask	
grlib_techmap_strict_ram	Defines if struct RAM TECHMAP should be used. Otherwise small (shallow) RAMs may be mapped to inferred technology. Not supported by all target technologies.
grlib_techmap_testin_extra	Expand testin vector to SYNCRAM components with additional bits (value defines number of additional bits).
grlib_sync_reset_enable_all	Add synchronous reset to all registers (requires support in instantiated IP cores). Synchronization registers will not have resets added. Note that IP cores may have VHDL generics that override the library settings.
grlib_async_reset_enable	Add asynchronous reset to all registers (requires support in instantiated IP cores, see IP core manual). This option must not be enabled together with *grlib_sync_reset_enable_all*. Asynchronous reset will not be used for synchronization registers and for registers where the reset state depends on external input signals. Note that IP cores may have VHDL generics that override the library settings.
grlib_syncramft_autosel_disable	Disables automatic override of ECC implementation in syncramft wrappers (GRLIB-FT only).
grlib_syncram_selftest_enable	Enables data monitors on syncram blocks.
grlib_external_testoen	Disable testoen multiplexing in IP cores. Not supported by all IP cores.
grlib_amba_inc_nirq	Increase maximum number of interrupts Total number of interrupt lines: (32 + *grlib_amba_inc_irq**32) Note: Should be left at 0 at present time. > 32 interrupts is not supported by all IP cores and it is not supported by the GRLIB plug&play version 0.
grlib_little_endian	Change the endianness of the system to little endian for IP cores that support this, see IP core manual. Note that IP cores may have VHDL generics that override the library settings.

TABLE 68. GRLIB configuration array description

IP core support for settings such as *grlib_sync_reset_enable_all*, *grlib_async_reset_enable* and *grlib_little_endian* is described for each IP core in the GRLIB IP Core User's Manual in the Implementation / Reset section.

Additional technology specific constants are documented in GRLIB-FT User's Manual (grlib-ft.pdf).

5.8 Technology mapping

5.8.1 General

GRLIB provides portability support for both ASIC and FPGA technologies. The support is implemented by means of encapsulation of technology specific components such as memories, pads and clock buffers. The interface to the encapsulated component is made technology independent, not relying on any specific VHDL or Verilog code provided by the foundry or FPGA manufacturer. The interface to the component stays therefore always the same. No modification of the design is therefore required if a different technology is targeted. The following technologies are currently supported by the TECHMAP.GENCOMP package:

```
constant inferred    : integer := 0;
constant virtex      : integer := 1;
constant virtex2     : integer := 2;
constant memvirage   : integer := 3;
constant axcel       : integer := 4;
constant proasic     : integer := 5;
constant atc18s      : integer := 6;
constant altera      : integer := 7;
constant umc         : integer := 8;
constant rhumc       : integer := 9;
constant apa3        : integer := 10;
constant spartan3    : integer := 11;
constant ihp25       : integer := 12;
constant rhlib18t    : integer := 13;
constant virtex4     : integer := 14;
constant lattice     : integer := 15;
constant ut25        : integer := 16;
constant spartan3e   : integer := 17;
constant peregrine   : integer := 18;
constant memartisan  : integer := 19;
constant virtex5     : integer := 20;
constant custom1     : integer := 21;
constant ihp25rh     : integer := 22;
constant stratix1    : integer := 23;
constant stratix2    : integer := 24;
constant eclipse     : integer := 25;
constant stratix3    : integer := 26;
constant cyclone3    : integer := 27;
constant memvirage90 : integer := 28;
constant tsmc90      : integer := 29;
constant easic90     : integer := 30;
constant atc18rha    : integer := 31;
constant smic013     : integer := 32;
constant tm65gpl     : integer := 33;
constant axdsp       : integer := 34;
constant spartan6    : integer := 35;
constant virtex6     : integer := 36;
constant actfus      : integer := 37;
constant stratix4    : integer := 38;
constant st65lp      : integer := 39;
constant st65gp      : integer := 40;
constant easic45     : integer := 41;
constant cmos9sf     : integer := 42;
constant apa3e       : integer := 43;
constant apa3l       : integer := 44;
constant ut130       : integer := 45;
constant ut90        : integer := 46;
constant gf65        : integer := 47;
constant virtex7     : integer := 48;
constant kintex7     : integer := 49;
constant artix7      : integer := 50;
constant zynq7000    : integer := 51;
constant rhlib13t    : integer := 52;
constant saed32      : integer := 53;
constant dare        : integer := 54;
constant igloo2      : integer := 55;
constant smartfusion2: integer := 55;
constant rhs65       : integer := 56;
constant rtg4        : integer := 57;
constant stratix5    : integer := 58;
constant memrhs65b   : integer := 59;
constant ultrascale  : integer := 60;
```

```
constant kintexu     : integer := 60;
constant virtexu     : integer := 60;
constant polarfire   : integer := 61;
constant nx          : integer := 62;
constant nx          : integer := 62;
constant dare65t     : integer := 63;
constant gf22        : integer := 64;
constant ultrascalep : integer := 65;
constant virtexup    : integer := 65;
constant kintexup    : integer := 65;
constant rhs28       : integer := 66;
constant techres1    : integer := 67;
```

Each encapsulating component provides a VHDL generic (normally named TECH) with which the targeted technology can be selected. The generic is used by the component to select the correct technology specific cells to instantiate in its architecture and to configure them approriately. This method does not rely on the synthesis tool to inferring the correct cells.

For technologies not defined in GRLIB, the default “inferred” option can be used. This option relies on the synthesis tool to infer the correct technology cells for the targeted device.

A second VHDL generic (normally named MEMTECH) is used for selecting the memory cell technology. This is useful for ASIC technologies where the pads are provided by the foundry and the memory cells are provided by a different source. For memory cells, generics are also used to specify the address and data widths, and the number of ports.

The two generics TECH and MEMTECH should be defined at the top level entity of a design and be propagated to all underlying components supporting technology specific implementations.

5.8.2 Memory blocks

Memory blocks are often implemented with technology specific cells or macrocells and require an encapsulating component to offer a unified technology independent interface. The TECHMAP library provides such technology independent memory component, as the synchronous single-port RAM shown in the following code example. The address and data widths are fully configurable by means of the generics ABITS and DBITS, respectively.

```
component syncram
  generic (
    memtech  : integer := 0;                          -- memory technology
    abits    : integer := 6;                          -- address width
    dbits    : integer := 8);                         -- data width
  port (
    clk      : in  std_ulogic;
    address  : in  std_logic_vector((abits -1) downto 0);
    datain   : in  std_logic_vector((dbits -1) downto 0);
    dataout  : out std_logic_vector((dbits -1) downto 0);
    enable   : in  std_ulogic;
    write    : in  std_ulogic);
end component;
```

This synchronous single-port RAM component is used in the AHB RAM component shown in the following code example.

```
component ahbram
  generic (
    hindex : integer := 0;                          -- AHB slave index
    haddr  : integer := 0;
    hmask  : integer := 16#fff#;
    memtech : integer := 0;                          -- memory technology
    kbytes  : integer := 1);                         -- memory size
  port (
    rst     : in  std_ulogic;
    clk     : in  std_ulogic;
    hslvi   : in  ahb_slv_in_type;                   -- AHB slave input
    hslvo   : out ahb_slv_out_type);                 -- AHB slave output
end component;

ram0 : ahbram
  generic map (hindex => 1, haddr => 16#240#, hmask => 16#FF0#,
               tech => virtex, kbytes => 4)
  port map (rst, clk, hslvi, hslvo(1));
```

In addition to the selection of technology (VIRTEX in this case), the size of the AHB RAM is specified in number of kilo-bytes. The conversion from kilo-bytes to the number of address bits is performed automatcally in the AHB RAM component. In this example, the data width is fixed to 32 bits and requires no generic. The VIRTEX constant used in this example is defined in the TECHMAP.GENCOMP package.

The following generic memory components are available in the TECHMAP library:

Name	Description	Ports
syncram	Single port synchronous memory	1xRW
syncram_2p	Two-port (one read, one write) synchronous memory	1xR,1xW
syncram_dp	True dual-port synchronous memory	2xRW
regfile_3p	Three port (two read, one write) synchronous memory	2xR,1xW

TABLE 69. Memory wrapper types in techmap library

5.8.3 Memory collision handling

There are corner cases where behavior of synchronous RAM macrocells can differ slightly between technologies, and the techmap library has some features to identify and sometimes emulate this for specific technologies.

For double-ported types of RAM, technologies differ in how the memory block handles simultaneous read and write to the same address. In the techmap library, the following types of behaviors are supported with respect to simultaneous read and write:

- The write succeeds but the read performed the same time will return undefined data.
- The memory location becomes corrupted (written with undefined data). Simultaneous read and write must be avoided.
- The write succeeds and the read returns the new data just written (write-first or write-through behavior).

To specify which of the above behaviors is the provided by the underlying cells, there are two capability vectors in the gencomp package for each type of memory called syncram_2p_write_through and syncram_2p_dest_rw_collision.

Certain IP (in particular, the LEON3 processor) depends in some cases on having the write-first type of behavior on its memories. When instantiating the syncram_2p wrapper inside the IP core, it is possible to specify this requirement through a generic wrfst on the syncram_2p component. If the underlying memory cells have a more restrictive behavior, the syncram_2p contains logic to detect and/or avoid the collision and emulate the write-first behavior.

5.8.4 Memory power-down optimizations

If the technology-specific memory blocks are designed so that the read data bus is guaranteed to hold its last read value when the memory block is disabled regardless of the value on the address bus, then certain optimizations can be enabled in the processor to disable the memories more often. This is identified by a capability vector in the gencomp package called syncram_readhold.

When enabling this feature, it is important to ensure the read-hold guarantee is upheld regardless of value on the address bus. For example, it is common to simulate large memory blocks using many smaller ones and selecting read data based on the highest bits of the read address that came in the previous cycle. If there are registers in the technology specific layer that hold the previous cycle's read address, then those registers must be held at their current value when the memory is disabled.

If the readhold setting is disabled (0), then the IP cores will have to feed back the previous cycles address and keep the memory enabled when it wants the memory to keep its current read data on the output.

5.8.5 Pads

As for memory cells, the pads used in a design are always technology dependent. The TECHMAP library provides a set of encapsulated components that hide all the technology specific details from the user. In addition to the VHDL generic used for selecting the technology (normally named TECH), generics are provided for specifying the input/output technology levels, voltage levels, slew and driving strength. A typical open-drain output pad is shown in the following code example:

```
component odpad
  generic (
```

```
    tech     : integer := 0;
    level    : integer := 0;
    slew     : integer := 0;
    voltage  : integer := 0;
    strength : integer := 0
  );
  port (
    pad      : out std_ulogic;
    o        : in  std_ulogic
  );
end component;

pad0 : odpad
  generic map (tech => virtex, level => pci33, voltage => x33v)
  port map (pad => pci_irq, o => irqn);
```

The TECHMAP.GENCOMP package defines the following constants that to be used for configuring pads:

```
-- input/output voltage

constant x18v      : integer := 1;
constant x25v      : integer := 2;
constant x33v      : integer := 3;
constant x50v      : integer := 5;

-- input/output levels

constant ttl       : integer := 0;
constant cmos      : integer := 1;
constant pci33     : integer := 2;
constant pci66     : integer := 3;
constant lvds      : integer := 4;
constant sstl2_i   : integer := 5;
constant sstl2_ii  : integer := 6;
constant sstl3_i   : integer := 7;
constant sstl3_ii  : integer := 8;

-- pad types

constant normal    : integer := 0;
constant pullup    : integer := 1;
constant pulldown  : integer := 2;
constant opendrain: integer := 3;
constant schmitt   : integer := 4;
constant dci       : integer := 5;
```

The slew control and driving strength is not supported by all target technologies, or is often implemented differently between different technologie. The documentation for the IP core implementing the pad should be consulted for details.

5.9 Scan test support

5.9.1 Overview

Scan test is a method for production testing digital ASICs. A test mode is added to the design that changes all flip-flops in the design to shift registers that can be set and read out serially. This is implemented partially in RTL code and partially in the implementation flow.

In a typical GRLIB ASIC, a number of signals are added for scan test. All signals except testen are usually muxed with other slow I/O signals so only one pin has to be added to the design.

The signals added are:

testen - Enables test mode (top-level pin)

scanen - Muxes flip-flop data inputs to previous in chain instead of normal function

testoen - Controls all output-enables in test mode

testrst - Controls all async-resets in test mode

scanin - Scan chain inputs

scanout - Scan chain outputs

The top level of the design adds the testen signal to the port list and muxes in the scanen, testoen and testrst signals. The scanin and scanout signals are not handled at the RTL level.

At the RTL level, the test signals are connected to any hard macro that needs them, such as block RAM:s and PLL:s. Also testoen and testrst are handled fully at source code level. The RTL also contains logic so that all flip-flops are directly clocked by an input clock pin when test mode is enabled.

During synthesis, the synthesis tool implements registers using special "scan flip-flops" containing the necessary muxing for the scan chain. The actual scan chain connections are not derived until after placement, so the scan order can be selected to minimize routing.

5.9.2 GRLIB support

To support scan test methods, GRLIB distributes the *testen*, *scanen*, *testoen* and *testrst* signals via the AHB and APB bus records. The signals are supplied into the AHB controllers which will pass them on to the AHB bus records. The APB controller will in turn forward them to the APB bus records. This way all IP cores connecting to an AHB or APB bus have access to the test signals without having to add extra input ports for them.

The GRLIB IP cores supporting scan test signals have a generic called *scantest* to enable this functionality. For historical reasons, this generic is on some IP cores called *scanen* or *testen* instead. Cores which use the scan signals include LEON3, MCTRL and GRGPIO.

The techmap layer handles certain test mode features. The clkgate component will automatically enable (pass through) the clock when test mode is enabled. The various syncram wrappers will disable the RAM:s during shifting (when scanen and testen are high).

The syncram techmaps have an input vector called *testin*, containing *testen*, *scanen*, plus two extra technology-dependent bits. The AMBA records contain a *testin* element that can be passed on directly to the syncram. The tech dependent bits can be set using the testsig input signal to the AHB controller. More bits can be added to the vector if necessary via a local GRLIB configuration option.

5.9.3 Usage for existing cores

For using the scan test support with existing cores in GRLIB, the test signals need to be supplied to the AHB controller and the scan test support needs to be enabled in the IP cores.

5.9.4 Usage for new cores

For adding scan test support to an IP core, a couple of changes may be needed.

- A generic called scantest should be added that enables scan test support. If the core does not have any AHB or APB interfaces, you will also need to add explicit inputs for any test signals that you need to implement the below.
- If the core has asynchronous resets, these should be tied to testrst when testen is high. This is usually done by a statement such as:

```
arst <= testrst when scantest/=0 and ahbsi.testen='1' else lrst;
```

- If the core controls output enables going directly to pads, these should be tied directly to testoen when testen is high.
- If you invert or divide clocks internally, these should be bypassed in test mode so all flip-flops are clocked by the same edge on the incoming clock:

```
lnclk <= not clk;
stgen: if scantest /= 0 generate
  m1: clkmux
    generic map (tech => tech)
    port map (io => lnclk, i1 => clk, sel => ahbsi.testen, o => nclk);
end generate;
nstgen: if scantest = 0 generate
  nclk <= lnclk;
end generate;
```

- Pass on the scantest generic and test signals to any submodules, techmap instances and hard macros that need them.

5.9.5 Configuration options

Certain options in the GRLIB configuration record (section 5.6) controls above features:

The *testin* vector to the syncrams can be enlarged from the default width of 4 (*testen*, *scanen*, and two custom inputs) to allow more design/technology-specific signals to be passed into the memory wrap-

pers. This is done by setting the *grlib_techmap_testin_extra* option to a nonzero value. This will widen also the AMBA records' *testin* field to accommodate the extra bits.

In some designs, the *testoen* connection to the output enables is done above the IP core level. For example such multiplexing may be included in the pads or in the boundary scan cells of the technology. The option *grlib_external_testoen* turns off the *testoen* multiplexing in some IP cores to remove the redundant logic. This is only implemented in some IP cores in the library. For IP where it has not been implemented, using this will then result in redundant *testoen* logic but should still be functionally correct.

5.10 Support for integrating memory BIST

GRLIB provides some infrastructure intended to support integrating memory BIST for ASIC designs directly at the RTL source level. Inserting at source level rather than at netlist level has several advantages, for example MBIST logic gets included in equivalence checking, MBIST execution can be simulated also at source level and a simplified implementation flow.

The support is divided into multiple layers, described below. Note that the IP core and top level layers are not included in all releases of GRLIB.

5.10.1 Syncram level

The syncram wrappers have two vectors called *customin* and *customout*, plus a *customclk* input. The width of the vectors is controlled by a *custombits* generic. These vectors can be used to communicate with the BIST for that RAM block.

The syncram wrapper converts the variable-width *customin/out* vectors into fixed-width zero-padded custominx and customoutx vectors, which can then be used by the mapping for a specific technology:

```
custominx(custominx'high downto custombits) <= (others => '0');
custominx(custombits-1 downto 0) <= customin;
customout <= customoutx(custombits-1 downto 0);
```

Note that if the mapping for a technology drives *customoutx*, it must also set the syncram_has_customif entry in gencomp.vhd, otherwise the *customout* vector is driven with all-zero to avoid undriven signal warnings in synthesis:

```
nocust: if syncram_has_customif(tech)=0 generate
    customoutx <= (others => '0');
end generate;
```

Some mappings, such as syncrambw and syncramft, may in some cases instantiate multiple syncram blocks internally. For such mappings, the *customin/out* vectors' widths is multiplied by the maximum number of sub-instances in order to provide a unique in/out vector for each block. Depending on how many blocks are actually instantiated, the top part of the vector may be unused (only the (custombits * Nsyncrams) lowest bits are used).

5.10.2 IP core level

Where this is supported, the IP core collects the *customin/customout* vectors of the instantiated syncrams into an array or record and propagates this to ports on the IP called *mtesti* and *mtesto*. The *customclk* is propagated to an input called *mtestclk*.

The *custombits* generic is not propagated but is set fixed in the IP to the constant *memtest_vlen*, defined in techmap/gencomp/gencomp.vhd. In gencomp.vhd, types *memtest_vector* and *memtest_vector_array* are also declared so this does not have to be done for every IP:

```
constant memtest_vlen: integer := 16;
subtype memtest_vector is std_logic_vector(memtest_vlen-1 downto 0);
type memtest_vector_array is array(natural range <>) of memtest_vector;
```

Below is an example to illustrate how this is integrated in an IP core:

```
type ipcore_memtest_type is record
  data_buffers: memtest_vector_array(0 to 5);
  control_ram: memtest_vector_array(0 to 1);
end record;
constant ipcore_memtest_none : ipcore_memtest_type := (
  (others => (others => '0')), (others => (others => '0')));

entity ipcore is
```

```
port(
...
mtesti    : in ipcore_memtest_type := grpci2_memtest_none;
mtesto    : out ipcore_memtest_type;
mtestclk  : in std_ulogic := '0'
);
end;

architecture rtl of ipcore is
begin
...
buf0 : syncram
generic map (..., custombits => memtest_vlen)
port map (... ,
customin => mtesti.data_buffers(0), customout => mtesto.data_buffer(0),
customclk => mtestclk);

end;
```

5.10.3 Design level

At the design top level, the different memtest records need to be combined together and interfaced to the design. How this is done depends on the exact details on the design and the MBIST implementation so it can not be completely standardized. This section describes one possible approach.

One way to do this is to create a shift register for each memory block, tie all shift registers in the design in series, and access it from the JTAG TAP. To do this, the syncram mapping is designed so that the *customin* bit 0 to each syncram is used as a serial data in, and its *customout* bit 0 is used as a serial data out. In order to tell which "slots" in the *memtest* record are actually occupied, bit 1 of the customout vector is used as a "present" indicator, driven by constant 1 when there is a real memory inside it. The JTAG clock is passed as *mtestclk/customclk*, and the JTAG control signals (update/shift/capture) can be passed either as extra bits on *customin* or using the additional bits of the *testin* interface (described in section 5.9)

The chaining can be done using VHDL procedures similar to the below:

```
  procedure chain_memtest(i: memtest_vector_array; o: out memtest_vector_array;
di: std_ulogic; do: out std_ulogic) is
    variable r: memtest_vector_array(0 to i'length-1);
    variable d: std_ulogic;
  begin
    r := (others => (others => '0'));
    d := di;
    for x in r'range loop
      r(x)(0) := d;
      if i(x)(1)='1' then
        d := i(x)(0);
      end if;
    end loop;
    o := r;
    do := d;
    mo := m;
  end procedure;

  process(mbist_tdi, mtesto_ip1, mtesto_ip2)
    variable di,do: std_ulogic;
    variable vi_ip1: ipcore1_memtest_type;
    variable vi_ip2: ipcore2_memtest_type;
  begin
    di := mbist_tdi;
    do := '0';
    chain_memtest(mtesto_ip1.data_buffers, vi_ip1.data_buffers, di, do);
    di := do;
    chain_memtest(mtesto_ip1.control_ram,  vi_ip1.control_ram,  di, do);
    di := do;
    chain_memtest(mtesto_ip2.data_buffers, vi_ip2.data_buffers, di, do);
    di := do;
    chain_memtest(mtesto_ip2.data_buffers, vi_ip2.data_buffers, di, do);
    di := do;
    mbist_tdo <= do;
  end process;
```

5.11 GRLIB system test software

5.11.1 Introduction

GRLIB contains test software that is intended to be run on a LEON processor in simulation. The collection of test software contains tests for both the LEON processors and peripheral units.

The test software is intended as a system level sanity check that verifies that IP cores have been correctly connected to the system. It does not provide in general full coverage of the IP cores. Higher coverage is achieved through the use of standalone test benches. For communication interfaces, some of the test software requires simulation models of external functions, such as transceivers. In this case the test software is also used to check that an off-chip interface has been correctly connected.

5.11.2 Typical test software use

As described in section 3.4, the test program executed by the test bench consists of two parts, a simple PROM boot loader (*prom.S*) and the test program itself (*systest.c*). Both parts can be re-compiled using the make soft command. This requires that the BCC 1.0.x toolchain is installed on the host computer. The BCC 1.0.x toolchain by default includes AMBA plug&play scanning routines that are able to scan over AHB bridges. This is seldom required for system tests since the test functions take the register area base addresses as inputs. Simulation time is decreased by the default assignment of the environment variable *LDFLAGS* to *LDFLAGS=-qnoambapp*. The default assignment can be avoided by defining the LDFLAGS variable in the template design Makefile.

The simple PROM boot loader (*prom.S*) contains code to initialize the processor, memory controller and other peripherals. If the file *prom.S* is missing from the template design folder then a default version located at *software/leon3/prom.S* will be used. Configuration constants used by *prom.S* are located in the file *prom.h*. If the memory controller in a design is changed, or the base address of main memory is moved, then *prom.h* and possibly *prom.S* may need to be updated to correctly initialize the new configuration. If *prom.h* or *prom.S* are modified then *make soft* is required before the changes take effect.

The boot loader is designed for simulation only.

Note that the simulation is terminated by generating a VHDL failure, which is the only way of stopping the simulation from inside the model. An error message is then printed:

```
# Test passed, halting with IU error mode
# ** Failure: *** IU in error mode, simulation halted ***
# Time: 1104788 ns Iteration: 0 Process: /testbench/iuerr File:
testbench.vhd
# Stopped at testbench.vhd line 338
```

5.11.3 Test software reporting

The system test software reports status messages on the simulation console such as:

```
# **** GRLIB system test starting ****
# LEON3 SPARC V8 Processor
# CPU#0 register file
# CPU#0 multiplier
# CPU#0 radix-2 divider
# CPU#0 floating-point unit
# CPU#0 cache system
```

This output is generated by calling the functions:

```
void report_start(void)
void report_end(void)
int report_device(int dev);
int report_subtest(int dev);
int fail(int dev);
```

One way of generating the output would be to use the accelerated UART tracing that can be enabled for GRLIB's APBUART and use printf() to output the status messages. In order to reduce simulation time, the test output is instead generated by a test module. This test module monitors write accesses and performs different functions based on the addresses and values written by the processor. The available test modules include:

- AHBREP - AMBA slave test module. Connects as an AHB slave to the AHB bus
- SDRTESTMOD - SDRAM test module. Connects to external SDRAM interface and monitors it for test module output.

- GRTESTMOD - Connects to system via a memory controllers memory-mapped IO interface.

The default address for the test module is at offset 0x20000000. This is the address traditionally used for memory-mapped IO in LEON/GRLIB systems. When the system test software is built with default settings the use of the report* functions will cause write operations to the memory area at 0x20000000. If nothing, or if another IP core, is mapped at this address then the test program is likely going to fail and at least will not show any output. The base address for the test module can be changed by defining *GRLIB_REPORTDEV_BASE*. This can be done in a template design Makefile, through the *BOPT* variable:

```
BOPT=-DGRLIB_REPORTDEV_BASE=0xD0000000
```

The default behaviour for the test program is to use 32-bit accesses. In case the GRTESTMOD is used and is connected to a 16-bit wide interfaces then the test software can be built to use half-word accesses instead through the define *GRLIB_REPORTDEV_WIDTH*. As an example, for a 16-bit wide memory-mapped IO interface with base address 0xD0000000, the following *BOPT* settings should be used:

```
BOPT=-DGRLIB_REPORTDEV_BASE=0xD0000000 -DGRLIB_REPORTDEV_WIDTH=16
```

5.11.4 Selecting the right test module

Most designs should use either GRTESTMOD or AHBREP. If the design has a memory controller with memory-mapped IO then using GRTESTMOD at the test bench level has the benefit of providing some test coverage for the memory-mapped IO interface. In addition to this, having the test module in the test bench allows it to be used for both RTL and netlist simulations.

For designs that lack suitable external interfaces, the AHBREP module can be instantiated in the design top-level. The AHBREP module cannot be synthesized and should be put within pragma statements. Note that this module will not be included in a design's netlist and another test module needs to be used for netlist simulations.

5.11.5 Standalone systest

The system tests can also be built in standalone mode. In this case the reporting functions:

```
void report_start(void)
void report_end(void)
int report_device(int dev);
int report_subtest(int dev);
int fail(int dev);
```

Will be replaced by versions that do not depend on the presence of a test module. Reporting will instead be used through calls to printf(). Note that tests that depend on simulation models of, for example, external transceivers will not function in standalone mode.

6 GRLIB Design examples and FPGA board template designs

6.1 Introduction

GRLIB contains a number of template design, both generic and tailored for specific development boards. The template design examples described in the following sections are provided for the understanding of how to integrate the existing GRLIB IP cores into a design. The documentation for the various IP cores should be consulted for details.

A cross-reference between FPGA development boards and template design can be found under section 6.2. Availability of template designs varies depending on type of GRLIB distribution (COM, FT, FT-FPGA, GPL).

Documentation on how to add a new template design to GRLIB is found in section 8.2.

6.2 Supported FPGA boards

FPGA Vendor	FPGA Board	Template design name
AMD/Xilinx	Avnet Spartan3-1500 board	leon3-avnet-3s1500
AMD/Xilinx	Avnet Virtex4 Evaluation board	leon3-avnet-eval-xc4vlx25, leon3-avnet-eval-xc4vlx60
AMD/Xilinx	Digilent Virtex2pro XUP board	leon3-digilent-xup
AMD/Xilinx	Digilent Basys3 board	leon3-digilent-basys3
AMD/Xilinx	Digilent Nexys 3 board	leon3-digilent-nexys3
AMD/Xilinx	Digilent Nexys 4 board	leon3-digilent-nexys4
AMD/Xilinx	Digilent Nexys 4 DDR board	leon3-digilent-nexys4ddr
AMD/Xilinx	Digilent Nexys Video board	leon3-digilent-nexys-video
AMD/Xilinx	Digilent Spartan3 Starter board	leon3-digilent-xc3s1000
AMD/Xilinx	Digilent Spartan3E Development board	leon3-digilent-xc3s1600e
AMD/Xilinx	Digilent Spartan6 Atlys board	leon3-digilent-atlys
AMD/Xilinx	Digilent XC7Z020	leon3-digilent-xc7z020
AMD/Xilinx	Digilent Arty A7 board	noelv-digilent-arty-a7
AMD/Xilinx	Nuhorizons Spartan3-1500 board	leon3-nuhorizons-3s1500
AMD/Xilinx	Pender/Gaisler GR-XC3S1500/2000 board	leon3-gr-xc3s-1500
AMD/Xilinx	Pender/Gaisler GR-PCI-XC2V3000 board	No longer supported.
AMD/Xilinx	Pender/Gaisler GR-CPCI-XC2V6000 board	No longer supported
AMD/Xilinx	Pender/Gaisler GR-CPCI-XC4VLX100/200 board	leon3-gr-cpci-xc4v
AMD/Xilinx	Pender/Gaisler GR-PCI-XC5VLX50/110 board	leon3-gr-pci-xc5v
AMD/Xilinx	Pender/Gaisler GR-XC6S-LX75 Spartan6 board	leon3-gr-xc6s
AMD/Xilinx	Pender/Gaisler GR-CPCI-XC7K board	leon3-gr-cpci-xc7k
AMD/Xilinx	Xilinx ML401 / ML402 / ML403 / ML501 / ML505 / ML506 / ML507 / ML510 boards	leon3-xilinx-ml40x, leon3-xilinx-ml403, leon3-xilinx-ml501, leon3-xilinx-ml50x, leon3-xilinx-ml510
AMD/Xilinx	Xilinx Spartan3A DSP-1800 Starter Platform	leon3-xilinx-xc3sd-1800
AMD/Xilinx	Xilinx SP601 Spartan6 Evaluation kit	leon3-xilinx-sp601
AMD/Xilinx	Xilinx SP605 Spartan6 Evaluation kit	leon3-xilinx-sp605
AMD/Xilinx	Xilinx ML605 Virtex-6 Development board	leon3-xilinx-ml605
AMD/Xilinx	Xilinx AC701 Artix-7 Evaluation kit	leon3-xilinx-ac701
AMD/Xilinx	Xilinx VC707 Virtex-7 Evaluation kit	leon3-xilinx-vc707 leon5-xilinx-vc707 noelv-xilinx-vc707
AMD/Xilinx	Xilinx KC705 Kintex Evaluation kit	leon3-xilinx-kc705 leon5-xilinx-kc705
AMD/Xilinx	Xilinx Zynq ZC702	leon3-xilinx-zc702
AMD/Xilinx	Xilinx KCU105 Kintex UltraScale Evaluation Kit	leon3-xilinx-kcu105 leon5-xilinx-kcu105 noelv-xilinx-kcu105

FPGA Vendor	FPGA Board	Template design name
AMD/Xilinx	ZTEX USB-FPGA Module 1.11	leon3-ztex-ufm-111
AMD/Xilinx	ZTEX USB-FPGA Module 1.15	leon3-ztex-ufm-115
Intel/Altera	Altera Stratix-II Development boardS	leon3-altera-ep2s60-sdr, leon3-altera-ep2s60-ddr, leon3-altera-ep2sgx90-av
Intel/Altera	Altera Cyclone-III Starter Kit	leon3-altera-ep3c25
Intel/Altera	Altera Cyclone-III Multimedia board	leon3-altera-ep3c25-eek
Intel/Altera	Altera CycloneV E Development kit	leon3-altera-c5ekit leon5-altera-c5ekit
Intel/Altera	Altera Stratix-III FPGA Development kit	leon3-altera-ep3sl150
Intel/Altera	Arrow BE Micro SDK Cyclone IV board	leon3-arrow-bemicro-sdk
Intel/Altera	TerASIC DE-4 Development and Education board	leon3-terasic-de4
Intel/Altera	TerASIC DE2-115 Cyclone-IV board	leon3-terasic-de2-115
Intel/Altera	TerASIC DE2 Cyclone II board	leon3-altera-de2-ep2c35
Intel/Altera	TerASIC DE0-Nano board	leon3-terasic-de0-nano
Intel/Altera	TerASIC SoCKit	leon3-terasic-sockit
Lattice	CertusPro-NX Evaluation Board	leon3-lattice-lfcpnx-evn leon5-lattice-lfcpnx-evn noelv-lattice-lfcpnx-evn
Lattice	CertusPro-NX Versa Board	leon3-lattice-lfcpnx-versa-evn leon5-lattice-lfcpnx-versa-evn noelv-lattice-lfcpnx-versa-evn
Lattice	Certus-NX Versa Evaluation Board	leon3-lattice-lfd2nx-versa-evn
Lattice	CrossLink-NX Evaluation Board	leon3-lattice-lifcl-40-evn
Lattice	Gaisler GR740-MINI	-
Microsemi	Actel Fusion Advanced Development kit	leon3-actel-fusion
Microsemi	Actel ProASIC3L Starter Kit	leon3-actel-proasic3l
Microsemi	Actel CoreMP7 Developers Kit	leon3-actel-proasic3
Microsemi	Microsemi IGLOO2 Evaluation Kit	leon3-microsemi-m2gl-eval-kit
Microsemi	ProASIC3 MCC-C Board	leon3-gr-mcc-c
Microsemi	GR-CPCI-AX board	leon3-rtax-cid*
Microsemi	RTG4 Development Kit	leon3-microsemi-rtg4-devkit
Microsemi	SmartFusion2 Evaluation Kit	leon3-microsemi-m2s090ts-eval-kit
Microsemi	SmartFusion2 Advanced Development Kit	leon3-microsemi-m2s150ts-adv-kit
Microsemi	PolarFire Evaluation Kit	leon3-microsemi-polarfire-eval-kit
Microsemi	PolarFire Splash Kit	leon3-microsemi-polarfire-splash-kit leon5-microsemi-polarfire-splash-kit noelv-microsemi-polarfire-splash-kit
NanoXplore	NG-Medium Development Kit	leon3-nanoxp-medium
NanoXplore	-	leon3-nanoxp-ultra

6.3 LEON3MP - Generic multiprocessor system

The LEON3MP design example described in this section is a multi-processor system based on LEON3MP. The design is based on IP cores from GRLIB. Only part of the VHDL code is listed hereafter, with comments after each excerpt. The design and the full source code is located in grlib/designs/leon3mp.

```
entity leon3mp is
  generic (
    ncpu    : integer := 1;
```

The number of LEON3 processors in this design example can be selected by means of the NCPU generic shown in the entity declaration excerpt above.

```
signal leon3i : l3_in_vector(0 to NCPU-1);
signal leon3o : l3_out_vector(0 to NCPU-1);
signal irqi   : irq_in_vector(0 to NCPU-1);
signal irqo   : irq_out_vector(0 to NCPU-1);
signal l3dbgi : l3_debug_in_vector(0 to NCPU-1);
signal l3dbgo : l3_debug_out_vector(0 to NCPU-1);
```

The debug support and interrupt handling is implemented separately for each LEON3 instantiation in a multi-processor system. The above signals are therefore declared in numbers corresponding to the NCPU generic.

```
signal apbi   : apb_slv_in_type;
signal apbo   : apb_slv_out_vector := (others => apb_none);
signal ahbsi  : ahb_slv_in_type;
signal ahbso  : ahb_slv_out_vector := (others => ahbs_none);
signal ahbmi  : ahb_mst_in_type;
signal ahbmo  : ahb_mst_out_vector := (others => ahbm_none);
```

The multiple LEON AMBA interfaces do not need any special handling in this example, and the AHB master/slave are therefore declared in the same way as in the previous example.

```
----------------------------------------------------------------------
---  LEON3 processor and DSU ---------------------------------------
----------------------------------------------------------------------
  cpu : for i in 0 to NCPU-1 generate
    u0 : leon3s                 -- LEON3 processor
      generic map (hindex => i, fabtech => FABTECH, memtech => MEMTECH,
                   fpu => fpu, dsu => dbg, disas => disas,
                   pclow => pclow, tbuf => 8*dbg,
                   v8 => 2, mac => 1, nwp => 2, lddel => 1,
                   isetsize => 1, ilinesize => 8, dsetsize => 1,
                   dlinesize => 8, dsnoop => 0)
      port map (clkm, rstn, ahbmi, ahbmo(i), ahbsi, leon3i(i), leon3o(i));

    irqi(i)         <= leon3o(i).irq;
    leon3i(i).irq   <= irqo(i);
    leon3i(i).debug <= l3dbgi(i);
    l3dbgo(i)       <= leon3o(i).debug;
  end generate;
```

The multiple LEON3 processors are instantiated using a generate statement. Note that the AHB index generic is incremented with the generate statement. Note also that the complete AHB slave input is fed to the processor, to allow for cache snooping.

```
  dcomgen : if dbg = 1 generate
    dsu0 : dsu                  -- LEON3 Debug Support Unit
      generic map (hindex => 2, ncpu => ncpu, tech => memtech, kbytes => 2)
      port map (rstn, clkm, ahbmi, ahbsi, ahbso(2), l3dbgo, l3dbgi, dsui, dsuo);

    dsui.enable <= dsuen;
    dsui.break  <= dsubre;
    dsuact      <= dsuo.active;

    dcom0: ahbuart              -- Debug UART
      generic map (ahbndx => NCPU, pindex => 7, paddr => 7)
      port map (rstn, clkm, dui, duo, apbi, apbo(7), ahbmi, ahbmo(NCPU));

    dui.rxd <= dsurx;
    dsutx <= duo.txd;
  end generate;
```

There is only one debug support unit (DSU) in the design, supporting multiple LEON3 processors.

```
  irqctrl0 : irqmp -- interrupt controller
    generic map (pindex => 2, paddr => 2, ncpu => NCPU)
    port map (rstn, clkm, apbi, apbo(2), irqi, irqo);
```

There is also only one interrupt controller, supporting multiple LEON3 processors.

To prepare the design for simulation with ModelSim, move to the grlib/designs/leon3mp directory and execute the 'make vsim' command.

```
$ make vsim
```

To simulate the default design execute the 'vsim' command.

```
$ vsim -c leon3mp
```

```
Simulate the first 100 ns by writing 'run'.
# LEON3 Demonstration design
# GRLIB Version 0.10
# Target technology: virtex ,  memory library: virtex
# ahbctrl: AHB arbiter/multiplexer rev 1
# ahbctrl: Common I/O area at 0xfff00000, 1 Mbyte
# ahbctrl: Configuration area at 0xfffff000, 4 kbyte
# ahbctrl: mst0: Frontgrade Gaisler        Leon3 SPARC V8 Processor
# ahbctrl: mst1: Frontgrade Gaisler        AHB Debug UART
# ahbctrl: slv0: European Space Agency   Leon2 Memory Controller
# ahbctrl:       memory at 0x00000000, size 512 Mbyte, cacheable, prefetch
# ahbctrl:       memory at 0x20000000, size 512 Mbyte
# ahbctrl:       memory at 0x40000000, size 1024 Mbyte, cacheable, prefetch
# ahbctrl: slv1: Frontgrade Gaisler        AHB/APB Bridge
# ahbctrl:       memory at 0x80000000, size 16 Mbyte
# ahbctrl: slv2: Frontgrade Gaisler        Leon3 Debug Support Unit
# ahbctrl:       memory at 0x90000000, size 256 Mbyte
# ahbctrl: slv6: Frontgrade Gaisler        AMBA Trace Buffer
# ahbctrl:       I/O port at 0xfff40000, size 128kbyte
# apbmst: APB Bridge at 0x80000000 rev 1
# apbmst: slv0: European Space Agency   Leon2 Memory Controller
# apbmst:       I/O ports at 0x80000000, size 256 byte
# apbmst: slv1: Frontgrade Gaisler        Generic UART
# apbmst:       I/O ports at 0x80000100, size 256 byte
# apbmst: slv2: Frontgrade Gaisler        Multi-processor Interrupt Ctrl.
# apbmst:       I/O ports at 0x80000200, size 256 byte
# apbmst: slv3: Frontgrade Gaisler        Modular Timer Unit
# apbmst:       I/O ports at 0x80000300, size 256 byte
# apbmst: slv7: Frontgrade Gaisler        AHB Debug UART
# apbmst:       I/O ports at 0x80000700, size 256 byte
# ahbtrace6: AHB Trace Buffer, 2 kbytes
# gptimer3: GR Timer Unit rev 0, 16-bit scaler, 2 32-bit timers, irq 8
# apbictrl: Multi-processor Interrupt Controller rev 1, #cpu 1
# apbuart1: Generic UART rev 1, irq 2
# ahbuart7: AHB Debug UART rev 0
# dsu2: LEON3 Debug support unit + AHB Trace Buffer, 2 kbytes
# leon3_0: LEON3 SPARC V8 processor rev 0
# leon3_0: icache 1*1 kbyte, dcache 1*1 kbyte
```

6.4 LEON3ASIC - ASIC flow example design

The LEON3ASIC design example provides a set of self-documented reference scripts for synthesis and verification of the generated netlist via formal verification and pre-layout GTL simulation. The LEON3ASIC synthesis and verification scripts serves as a guideline for developing and integrating your synthesis scripts into GRLIB. The design and scripts is located in grlib/designs/leon3asic.

The LEON3ASIC synthesis scrips include options to support different ASIC technology libraries via GRLIB TECHMAP structure, Insertion of SCAN and BIST and different synthesis options to in prove quality and timing of the LEON3ASIC netlist. Build options is set in build script dc.tcl except for the ASIC library which is set in **config.vhd** or **make xconfig**.

6.4.1 Modification of GRLIB Scripts

Selected TECH and MEMTECH generics are used for selecting the overall technology and the memory technology. TECH and MEMTECH generics needs to be passed on to synthesis and verification scripts in order for the scripts to select and compile correct ASIC technology library. The LEON3ASIC reference design make use of the pre-processing feature in Makefile scripts to extract the information from **config.vhd** by adding the following lines to the LEON3ASIC design **Makefile**:

```
TECHLIBS = $(shell grep FABTECH config.vhd | grep -o "[^ ]*$$"  | sed -e 's/;//g')
inferred grdware dware secureip unisim
```

```
DCOPT = -x "set argv [lindex [list $(TECHLIBS)] 0]; set top $(TOP)"
DCSCRIPT=dc.tcl

FMOPT = -x "set argv [lindex [list $(TECHLIBS)] 0]; set top $(TOP)"
FMSCRIPT=fm.tcl

VSIMOPT= -t ps -L work -L $(TECHLIBS) -novopt -i $(SIMTOP)
VSIMGTLOPT=$(VSIMOPT) -do ./gtl.do  -sdfmax /$(SIMTOP)/$(TOP)=./synopsys/
$(TOP)_$(grtechlib).sdf
```

Only the variable *VSIMGTLOPT* are local and the variables *DCOPT, DCSCRIPT, FMOPT, FMSCRIPT* and *VSIMOPT* are all integrated GRLIB variables.

6.4.2 RTL Simulation scripts

To compile and simulate the default design, move to the grlib/designs/leon3asic directory and execute the GRLIB command ‘vsim’ command.

```
$ make vsim
$ make vsim-launch
```

Simulate the first 100 ns by writing ‘run’.

```
# LEON3 ASIC Demonstration design
# GRLIB Version 1.3.2, build 4137
# Target technology: dare      , memory library: dare
# ahbctrl: AHB arbiter/multiplexer rev 1
# ahbctrl: Common I/O area disabled
# ahbctrl: AHB masters: 1, AHB slaves: 1
# ahbctrl: Configuration area at 0xfffff000, 4 kbyte
# ahbctrl: mst0: Aeroflex Gaisler        AHB-to-AHB Bridge
# ahbctrl: slv0: Aeroflex Gaisler        AHB/APB Bridge
# ahbctrl:       memory at 0x80000000, size 1 Mbyte
# ahbctrl: AHB arbiter/multiplexer rev 1
# ahbctrl: Common I/O area disabled
# ahbctrl: AHB masters: 6, AHB slaves: 8
# ahbctrl: Configuration area at 0xfffff000, 4 kbyte
# ahbctrl: mst0: Aeroflex Gaisler        LEON3 SPARC V8 Processor
# ahbctrl: mst1: Aeroflex Gaisler        AHB Debug UART
# ahbctrl: mst2: Aeroflex Gaisler        JTAG Debug Link
# ahbctrl: mst3: Aeroflex Gaisler        GRSPW2 SpaceWire Serial Link
# ahbctrl: mst4: Aeroflex Gaisler        GRSPW2 SpaceWire Serial Link
# ahbctrl: mst5: Aeroflex Gaisler        GRSPW2 SpaceWire Serial Link
# ahbctrl: slv0: European Space Agency   LEON2 Memory Controller
# ahbctrl:       memory at 0x00000000, size 512 Mbyte, cacheable, prefetch
# ahbctrl:       memory at 0x20000000, size 512 Mbyte
# ahbctrl:       memory at 0x40000000, size 1024 Mbyte, cacheable, prefetch
# ahbctrl: slv1: Aeroflex Gaisler        AHB-to-AHB Bridge
# ahbctrl:       memory at 0x80000000, size 256 Mbyte
# ahbctrl: slv2: Aeroflex Gaisler        LEON3 Debug Support Unit
# ahbctrl:       memory at 0x90000000, size 256 Mbyte
# ahbctrl: slv3: Aeroflex Gaisler        AHB/APB Bridge
# ahbctrl:       memory at 0xa0000000, size 1 Mbyte
# apbctrl: APB Bridge at 0xa0000000 rev 1
# apbctrl: slv0: European Space Agency   LEON2 Memory Controller
# apbctrl:       I/O ports at 0xa0000000, size 256 byte
# apbctrl: slv2: Aeroflex Gaisler        Multi-processor Interrupt Ctrl.
# apbctrl:       I/O ports at 0xa0000200, size 256 byte
# apbctrl: slv10: Aeroflex Gaisler        GRSPW2 SpaceWire Serial Link
# apbctrl:       I/O ports at 0xa0000a00, size 256 byte
# apbctrl: slv11: Aeroflex Gaisler        GRSPW2 SpaceWire Serial Link
# apbctrl:       I/O ports at 0xa0000b00, size 256 byte
# apbctrl: slv12: Aeroflex Gaisler        GRSPW2 SpaceWire Serial Link
# apbctrl:       I/O ports at 0xa0000c00, size 256 byte
# apbctrl: slv15: Aeroflex Gaisler        AHB Status Register
# apbctrl:       I/O ports at 0xa0000f00, size 256 byte
# apbctrl: APB Bridge at 0x80000000 rev 1
# apbctrl: slv1: Aeroflex Gaisler        Generic UART
# apbctrl:       I/O ports at 0x80000100, size 256 byte
# apbctrl: slv3: Aeroflex Gaisler        Modular Timer Unit
# apbctrl:       I/O ports at 0x80000300, size 256 byte
# apbctrl: slv6: Aeroflex Gaisler        General Purpose I/O port
# apbctrl:       I/O ports at 0x80000600, size 256 byte
# apbctrl: slv7: Aeroflex Gaisler        AHB Debug UART
# apbctrl:       I/O ports at 0x80000700, size 256 byte
# apbctrl: slv9: Aeroflex Gaisler        Generic UART
# apbctrl:       I/O ports at 0x80000900, size 256 byte
# apbctrl: slv13: Aeroflex Gaisler        AMBA Wrapper for OC I2C-master
```

```
# apbctrl:       I/O ports at 0x80000d00, size 256 byte
# apbctrl: slv14: Aeroflex Gaisler        SPI Controller
# apbctrl:       I/O ports at 0x80000e00, size 256 byte
# grspw12: Spacewire link rev 0, AHB fifos 2x64 bytes, rx fifo 16 bytes, irq 12
# grspw11: Spacewire link rev 0, AHB fifos 2x64 bytes, rx fifo 16 bytes, irq 11
# grspw10: Spacewire link rev 0, AHB fifos 2x64 bytes, rx fifo 16 bytes, irq 10
# ahbstat15: AHB status unit rev 0, irq 1
# spictrl14: SPI controller, rev 5, irq 14
# i2cmst13: AMBA Wrapper for OC I2C-master rev 3, irq 13
# grgpio6: 16-bit GPIO Unit rev 2
# gptimer3: GR Timer Unit rev 0, 12-bit scaler, 4 32-bit timers, irq 6
# irqmp: Multi-processor Interrupt Controller rev 3, #cpu 1, eirq 0
# apbuart9: Generic UART rev 1, fifo 4, irq 3, scaler bits 12
# apbuart1: Generic UART rev 1, fifo 4, irq 2, scaler bits 12
# ahbjtag AHB Debug JTAG rev 2
# ahbuart7: AHB Debug UART rev 0
# dsu3_2: LEON3 Debug support unit + AHB Trace Buffer, 1 kbytes
# leon3_0: LEON3 SPARC V8 processor rev 3: iuft: 0, fpft: 0
# leon3_0: icache 1*4 kbyte, dcache 1*4 kbyte
```

6.4.3 Synthesis scripts

The LEON3ASIC design synthesis script **dc.tcl** has been tested in Design Compiler H-2013.03-SP5. The **dc.tcl** script calls the generated GRLIB script for compilation and elaboration. Script name and location can be modified via the GRLIB variable DSCRIPT.

To synthesize the LEON3ASIC design, move to the grlib/designs/leon3asic directory and execute the GRLIB 'dc' command:

```
$ make dc
```

The synthesis script calls the scripts **timing.tcl** for general timing constraints, **report.tcl** to report timing and design exceptions found during synthesis and ASIC technology setup and timing scripts are located in the directory **grlib/designs/leon3asic/grtechscripts**.

For every ASIC technology a setup and timing script is required. The setup script **grtechscripts/<techmap_name>_setup.tcl** specify the ASIC library location and which cells to use during the synthesis. The timing script **grtechscripts/<techmap_name>_timing.tcl** specify clocks, timing margin and operation condition to be used for ASIC technology.

6.4.4 Formal verification scripts

The LEON3ASIC design formal verification script **fm.tcl** has been tested using Design Compiler H-2013.03-SP5 and Formality H-2013.03-SP5. Script name and location can be modfied via the GRLIB variable *FMSCRIPT*.

To run equivalence check execute the GRLIB ‘fm’ command:

```
$ make fm
```

6.4.5 GTL Simulation scripts

To simulate the synthesis netlist using the testbench the ASIC vendor library simulation models needs to integrated into the GRLIB or as in the LEON3ASIC reference design a new separate target for compiling the ASIC vendor library is used.

To GTL simulation execute the local LEON3ASIC design ‘gtl-vsim-launch’ command:

```
$ make gtl-vsim-launch
```

6.5 Xilinx Dynamic Partial Reconfiguration Examples

Examples of how to create dynamically reconfigurable systems on Xilinx FPGAs are included in several GRLIB template designs. The following documents describe the design flow and IP cores:

doc/dprc/qsg/dprc_qsg.pdf - DPRC and Partial Reconfiguration Design Flow - Quick Start Guide

doc/dprc/ug/dprc_ug.pdf - IP core documentation for FPGADynamic Reconfiguration controller with DMA AHB interface.

The following template designs contain example instantiation of the DPRC IP core:

leon3-digilent-nexys4ddr

leon3-gr-cpci-xc4v

leon3-xilinx-vc707

Please note that the use of partial reconfiguration requires a special license feature from Xilinx.

6.6 Microsemi designs

Corresponding documentation exists for RTG4 and the RTG4 Development kit in the GRLIB-FT User's Manual (doc/grlib-ft.pdf).

Note that using the PolarFire designs require a different version of Libero; Libero SoC PolarFire or Libero SoC v12.0 or newer. These versions are similar to Libero SoC with the major difference that only enhanced constraint flow is supported.

The subsections below describe how to generate a design down to programming the FPGA. Examples of connecting to the design with GRMON and initializing it can be found in the template designs README.txt file.

6.6.1 Simulating from Libero v12.0

When using Libero version 12.0 no stimulus files can be added to the project due to an issue in the tool. The testbench is therefore added as a design file and has to be added to the simulation flow manually. After issuing the command make libero-launch, right-click on *Simulation* in the *Design Flow* tab and choose *Organize Input Files > Organize Source Files* and add the testbench to the *Associated Source Files*. Double-click on *Simulation* to start the simulation and generate a run.do file. A warning will appear that no stimulus files are detected, select Yes to continue. The simulation will start without using the testbench and in order to use the testbench the generated run.do file has to be modified. The file is located in /leon3mp_libero/simulation and the last lines have to be changed. Remove -gSIM_PA5M300T=0 and replace presynth.leon3mp with presynth.testbench for the vsim command. Also uncomment the two last lines regarding the wave.do file and the run command. The simulation can then be started by issuing the command **do run.do** in ModelSim.

6.6.2 Libero projects with encrypted RTL on Windows

There is currently an issue with Microsemi's Libero tool when running on Windows and including encrypted RTL in a Libero project. This issue makes it impossible to open a Libero project with HDL language set to VHDL-93 when including encrypted files in the project. Therefore, VHDL2008 has to be used when opening the project. However, using VHDL2008 will cause simulation and synthesis errors for some of the files included in the GRLIB.

Because of this issue the HDL language has to be set manually in the Libero GUI when using Windows. The language is set from *Project > Project settings > Design flow* where VHDL-93 has to be selected. This has to be done every time the project is opened. When saving the project the language is set to VHDL-93 and if the project is opened as it is, Libero will terminate. By running the command **make libero-fix-vhdllang** before opening the project the language is changed to VHDL2008 and can then be opened with Libero whereupon the language has to be changed manually again.

6.6.3 Using the template designs

The steps below are described based on Libero v11.8-SP3. The flow is expected to be similar in later versions of Libero SoC and for Libero SoC PolarFire.

All the template designs contains one or more pregenerated Microsemi IP. Usage of these are enabled via xconfig under the menu *Synthesis > Use pregenerated Microsemi IP*. If the option is enabled the generated vhdl and verilog files are used in the project. The pregenerated IP has been built using Libero 11.7 SP1 or Libero PolarFire 2.3. If the xconfig option *Synthesis > Use pregenerated Microsemi IP* is set to n before the Libero project is generated the pregenerated IP is included in the project as a SmartDesign and can be configured by the user. Note that if the SmartDesign option is used the IP has to be generated through the Libero GUI and the commands **make libero** and **make libero-prog-fpga** cannot be used directly. Simulation of the design cannot be run either before generating the IP.

6.6.3.1 Step 1: Installation of simulation libraries

The Libero installation includes precompiled simulation libraries for Modelsim. If Modelsim is the target simulator, set LIBERO_ROOTDIR environment variable to point to Libero installation directory (i.e., the one that includes Libero/, Model/, Synplify/). Example:

```
export LIBERO_ROOTDIR=/home/user/Libero_v11.8
```

In order to copy Microsemi simulation libraries for Modelsim into GRLIB tree, issue **make install-microsemi-precomp**. They can be removed by issuing **make remove-microsemi-precomp**.

If the target simulator is Riviera, precompiled simulation libraries can be downloaded from the Microsemi website. After extracting the libraries, set SF2SIMLIB_RIVIERA environment variable to point to the directory including the precompiled files. In addition, set GRLIB_SIMULATOR variable to “ALDEC”, as in the following example:

```
export SF2SIMLIB_RIVIERA=/home/user/precompiled_libraries
export GRLIB_SIMULATOR=ALDEC
```

In this case, **make install-microsemi-precomp** is not needed.

6.6.3.2 Step 2: Configure design

In the design directory issue **make xconfig** to setup the LEON/NOEL system configuration and then **make soft** to compile prom.S and systest.c in order to generate the boot code and system test program SREC files needed for simulation (i.e., prom.srec and ram.srec).

6.6.3.3 Step 3: Generate scripts

Issue **make scripts**. This will generate a Tcl script (TOP_libero.tcl) that will be used in Libero SoC to build the design project including all the necessary design files.

6.6.3.4 Step 4: Start Libero SoC

Issue **make libero-launch**. This will create the design project (executing TOP_libero.tcl) and will launch the graphical user interface of Libero SoC.

During this step, errors and/or warnings on imported files (as in the following snippet) can be safely ignored:

```
Error: 'version' cannot be set as root.
Error: 'config_types' cannot be set as root.
Error: 'config' cannot be set as root.
Error: 'config' cannot be set as root.
Error: 'config' cannot be set as root.
Error: 'config' cannot be set as root.
Error: The components module is defined in multiple files. Duplicate modules are
not supported.
Select the file you want to use from the Design Hierarchy.
Error: The cpu_disas module is defined in multiple files. Duplicate modules are not
supported.
Select the file you want to use from the Design Hierarchy.
Error: The config module is defined in multiple files. Duplicate modules are not
supported.
Select the file you want to use from the Design Hierarchy.
```

All Libero SoC project files will be placed under <GRLIB path>/designs/leon3-microsemi-.../TOP_libero directory. In addition, when issuing **make libero-launch**, ram.srec (generated during step 2) will be copied into <GRLIB path>/designs/leon3-microsemi-.../TOP_libero/simulation for usage in simulation.

6.6.3.5 Step 5: Simulator setup

In Libero GUI, setup the simulation tool profile to start Modelsim SE (non-Microsemi version) or Riviera. This can be accomplished by clicking on *Project > Tool Profiles > Simulation > Add new profile*. The profile must be setup in order to include the directory of Modelsim SE (or Riviera) executable. Simulation will only work with Modelsim SE or Riviera. Modelsim Microsemi Edition (available with Libero SoC) cannot be used since it does not support mixed-language simulations.

6.6.3.6 Step 6: Generate pregenerated IP

Note that this step is only necessary if the option to use pregenerated IP is not selected with xconfig.

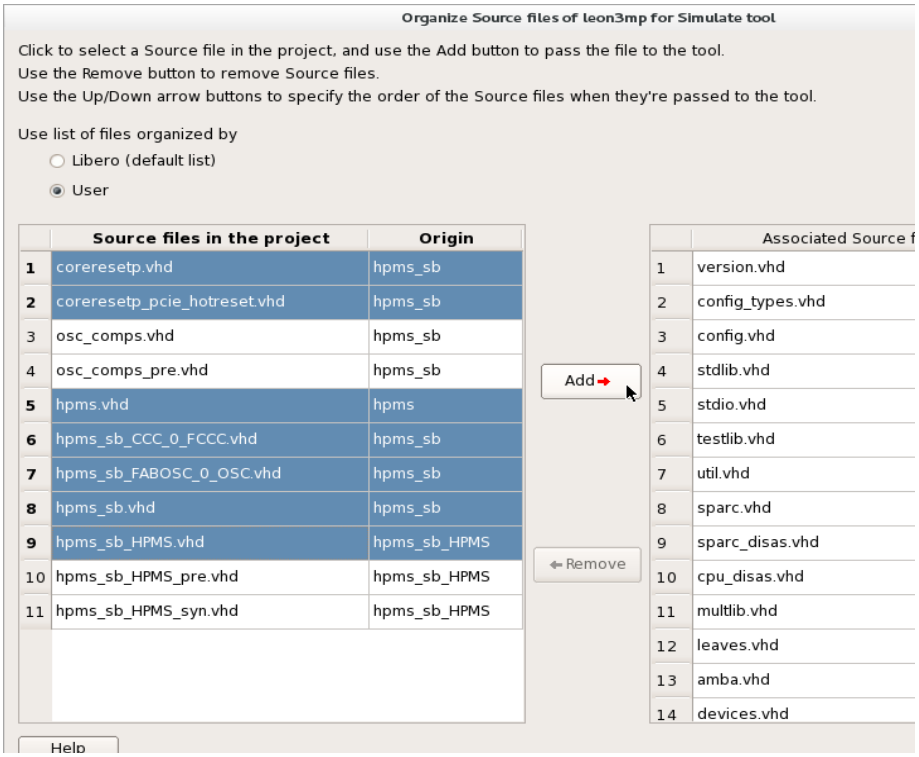
To configure the pregenerated IP, in the “Design Hierarchy” tab, double-click on the SmartDesign components to open them. The configurations can be changed by the user as long as they do not introduce any additional I/Os to the component.

The IGLOO2 designs use a HPMS subsystem whilst the Smartfusion2 designs use a MSS subsystem. The PolarFire designs uses several different clock generators and a memory controller. For each component click on Generate Component button in the SmartDesign toolbar.

6.6.3.7 Step 7: Add generated component files to simulation

Note that this step is only necessary if the option to use pregenerated IP is not selected with xconfig.

Before simulating the design, the generated files from the SmartDesign have to be added to the simulation flow. In the Design Flow window, right-click on Simulate, under Verify Pre-Synthesized Design, and select *Organize Input Files > Organize Source Files*. In the Organize Source Files window, select the files associated to the generated components (left side), and add them to the simulation sources (right side) by clicking on the Add button (see figure below).



For each design there are some files that must not be added to the simulation. For IGLOO2 designs these are:

· osc_comps.vhd;

· osc_comps_pre.vhd;

· hpms_sb_HPMS_pre.vhd;

· hpms_sb_HPMS_syn.vhd.

For Smartfusion2:

· osc_comps.vhd;

· osc_comps_pre.vhd;

· mss_sb_MSS_pre.vhd;

· mss_sb_MSS_syn.vhd.

For PolarFire:

· CoreDDR_TIP_SYN.v.

If SERDES is enabled in the IGLOO2 or Smartfusion2 design, then two additional files should not be added:

· igl_serdes_epcs_SERDES_IF_0_SERDESIF_pre.vhd;

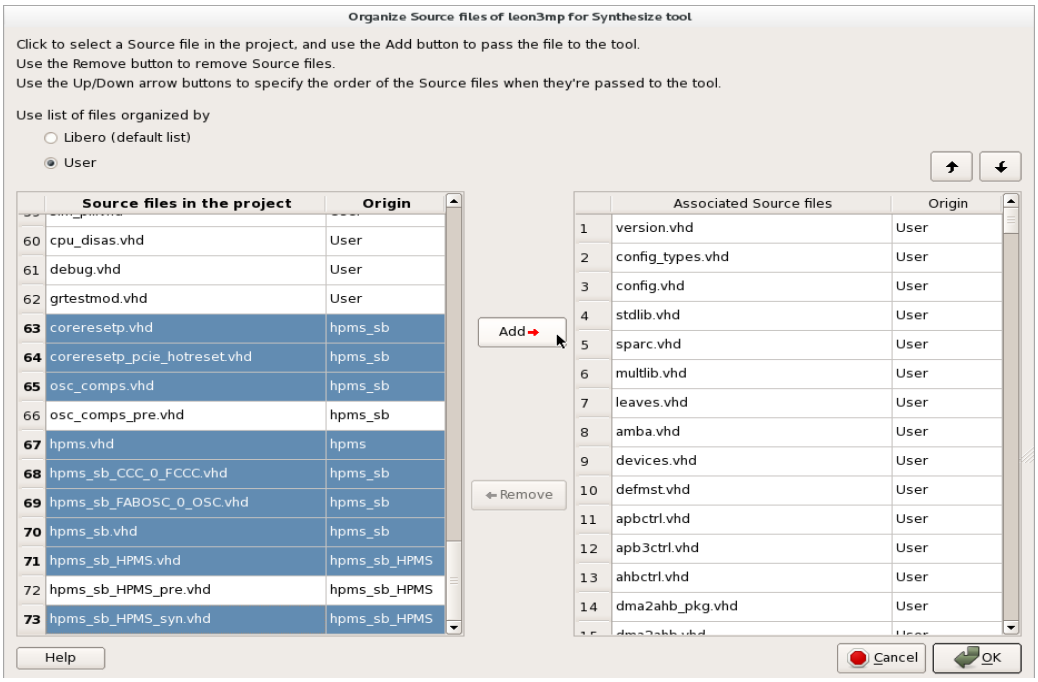
· igl_serdes_epcs_SERDES_IF_0_SERDESIF_syn.vhd.

Note: if the target simulator is Riviera, issue **make patch-simriviera** from the design directory in a terminal before launching simulation from Libero SoC.

6.6.3.8 Step 8: Add generated component files to synthesis

Note that this step is only necessary if the option to use pregenerated IP is not selected with xconfig.

The same process must be performed before synthesis as for the simulation. In the Design Flow tab of Libero SoC, right-click on Synthesize and select *Organize Input Files > Organize Source Files*. On the left side of Organize Source Files window, select files associated with the generated components and add them to the synthesis sources (right side) by clicking on the Add button (see figure below).



For each design there are some files that must not be added to the simulation. For IGLOO2 designs these are:

· osc_comps_pre.vhd;

· hpms_sb_HPMS_pre.vhd.

For Smartfusion2:

· osc_comps_pre.vhd;

· mss_sb_MSS_pre.vhd.

For PolarFire:

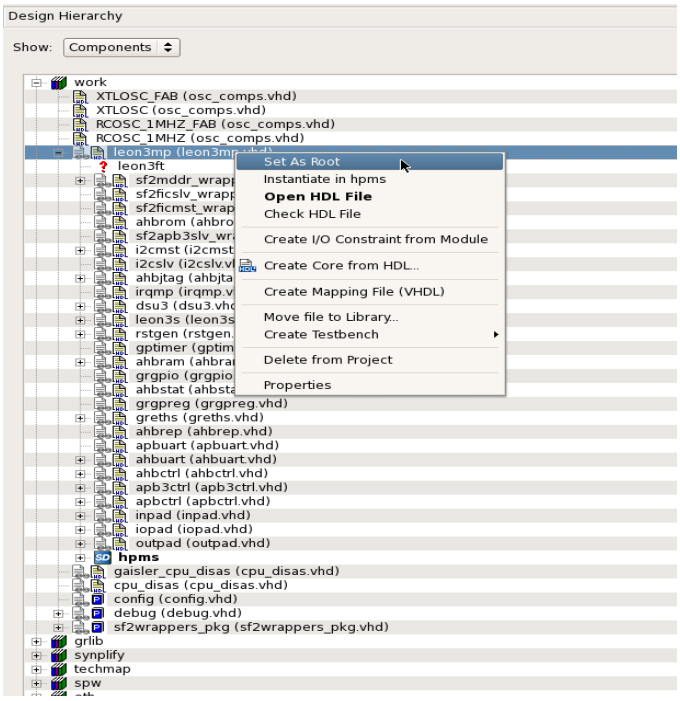
· CoreDDR_TIP_SIM.v.

If SERDES is enabled in the IGLOO2 or Smartfusion2 design, then one additional files should not be added:

· igl_serdes_epcs_SERDES_IF_0_SERDESIF_pre.vhd.

6.6.3.9 Step 9: Change root of project

In the Design Hierarchy window of Libero SoC, right-click on the top entity leon3mp and set it as root (this step is not required if leon3mp is already root).



6.6.3.10 Step 10

With the previous steps completed, the implementation flow in Libero SoC can be executed by running:

I. Synthesis;

II. Compile (this step does not exist for Libero SoC PolarFire);

III. Place and Route;

IV. Verify Timing (in this step carefully check for setup/hold violations in the generated reports);

V. Generate Bitstream;

VI. Run Program Action.

Simulation of the pre-synthesized design can be launched double-clicking on Simulate under Verify Pre-Synthesized Design (in the Design Flow tab).

7 Using netlists

7.1 Introduction

GRLIB supports the usage of mapped netlists in the implementation flow. The netlists can be included in the flow at two different points; during synthesis or during place&route. The netlists can have two basic formats: mapped VHDL (.vhd) or a technology-specific netlist format (.ngo, .vqm, .edf). The sections below outline how the different formats are handled.

GRLIB IP cores such as GRSPW, GRSPW2. GRFPU, GRFPU-lite, LEON3FT and GR1553B that were traditionally available only as netlists are provided as encrypted RTL instead of netlist format. The main remaining use for netlists are for GRFPU/GRFPU-lite evaluation. Some IP cores, such as GRPCI2, may have parts of the IP core in netlist format in order to simplify constraints and timing closure.

7.2 Mapped VHDL

A core provided in mapped VHDL format is included during synthesis, and treated the same as any RTL VHDL code. To use such netlist, the core must be configured to incorporate the netlist rather than the RTL VHDL code. This can be done in the **xconfig** configuration menu, or by setting the 'netlist' generic on the IP core. The benefit of VHDL netlists is that the core (and whole design) can be simulated and verified without special simulation libraries.

7.3 Xilinx netlist files

To use Xilinx netlist files (.ngo or .edf), the netlist should be placed in the 'netlists/xilinx/tech' directories. During place&route, the ISE mapper will look in this location and replace and black-boxes in the design with the corresponding netlist.

A special case exists for GRFPU and GRFPU-lite netlists. In GRLIB distributions that lack FPU source code, the netlist version of the selected FPU core will always be instantiated. When the design is simulated a VHDL netlist will be used (if available) and when the design is synthesized an EDIF netlist will be used. This is done in order to speed up synthesis. Parsing and performing synthesis on VHDL netlists is time consuming and using an EDIF netlist instead decreases the time required to run the tools.

Some tool versions have bugs that prevent them from using EDIF netlists. In order to work around such issues, convert the EDIF netlist to a .ngo netlist using the *edif2ngd* application in the ISE suite. After a netlist has been converted to .ngo format the EDIF version can be removed from the library.

7.4 Altera netlists

To use Altera netlist files (.vqm), the netlist should be placed in the 'netlists/altera/tech' directories, or in the current design directory. During place&route, the Altera mapper will look in these location and replace and black-boxes in the design with the corresponding netlist. Note that when using .vqm files, the 'netlist' generic on the cores should NOT be set.

A special case exists for GRFPU and GRFPU-lite netlists. In GRLIB distributions that lack FPU source code, the netlist version of the selected FPU core will always be instantiated. When the design is simulated a VHDL netlist will be used (if available) and when the design is synthesized a .vqm netlist will be used. This is done in order to speed up synthesis and due to the synthesis tools not always being able to handle VHDL netlists correctly.

7.5 Known limitations

Some tool versions have bugs that prevent them from using EDIF netlists. In order to work around such issues, convert the EDIF netlist to a .ngo netlist using the *edif2ngd* application in the ISE suite. After a netlist has been converted to .ngo format the EDIF version can be removed from the library

When synthesizing with Xilinx XST, the tool can crash when the VHDL netlist of GRFPU is used. This is not an issue with recent GRLIB versions since the VHDL netlists are currently only used for simulation.

8 Extending GRLIB

8.1 Introduction

GRLIB consists of a number of VHDL libraries, each one providing a specific set of interfaces or IP cores. The libraries are used to group IP cores according to the vendor, or to provide shared data structures and functions. Extension of GRLIB can be done by adding cores to an existing library, adding a new library and associated cores/packages, adding portability support for a new target technology, adding support for a new simulator or synthesis tool, or adding a board support package for a new FPGA board.

8.2 GRLIB organisation

The automatic generation of compile scripts searches for VHDL libraries in the file lib/libs.txt, and in lib/*/libs.txt. The libs.txt files contains paths to directories containing IP cores to be compiled into the same VHDL library. The name of the VHDL library is the same as the directory. The main libs.txt (lib/libs.txt) provides mappings to libraries that are always present in GRLIB, or which depend on a specific compile order (the libraries are compiled in the order they appear in libs.txt):

```
$ cat lib/libs.txt
grlib
tech/atc18
tech/apa
tech/unisim
tech/virage
fpu
gaisler
esa
opencores
```

Relative paths are allowed as entries in the libs.txt files. The path depth is unlimited. The leaf of each path corresponds to a VHDL libary name (e.g. ‘grlib’ and ‘unisim’).

Each directory specified in the libs.txt contains the file dirs.txt, which contains paths to sub-directories containing the actual VHDL code. In each of the sub-directories appearing in dirs.txt should contain the files vhdlsyn.txt and vhdlsim.txt. The file vhdlsyn.txt contains the names of the files which should be compiled for synthesis (and simulation), while vhdlsim.txt contains the name of the files which only should be used for simulation. The files are compiled in the order they appear, with the files in vhdlsyn.txt compiled before the files in vhdlsim.txt.

The example below shows how the AMBA package in the GRLIB VHDL library is constructed:

```
$ ls lib/grlib
amba/  dirs.txt  modgen/  sparc/  stdlib/  tech/  util/

$ cat lib/grlib/dirs.txt
stdlib
ftlib
util
sparc
modgen
amba
atf
dftlib
generic_bm

$ ls lib/grlib/amba
ahbctrl.vhd amba.vhd apbctrl.vhd vhdlsyn.txt

$ cat grlib/lib/grlib/amba/vhdlsyn.txt
amba.vhd
devices.vhd
apbctrl.vhd
apbctrlx.vhd
apbctrldp.vhd
apbctrlsp.vhd
apb3ctrl.vhd
```

```
ahbctrl.vhd
ahbxb.vhd
dma2ahb_pkg.vhd
dma2ahb.vhd
ahbmst.vhd
ahblitm2ahbm.vhd
ambaprot.vhd
```

The libraries listed in the grlib/lib/libs.txt file are scanned first, and the VHDL files are added to the automaticaly generated compile scipts. Then all sub-directories in lib are scanned for additional libs.txt files, which are then also scanned for VHDL files. It is therefore possible to add a VHDL library (= sub-directory to lib) without having to edit lib/libs.txt, just by inserting into lib.

When all libs.txt files have been scanned, the dirs.txt file in lib/work is scanned and any cores in the VHDL work library are added to the compile scripts. The work directory must be treated last to avoid circular references between work and other libraries. The work directory is always scanned as does not appear in lib/libs.txt.

8.2.1 Encrypted RTL

If the GRLIB library includes IP cores that are distributed as encrypted RTL, then the files with encrypted RTL are not listed in the vhdlsyn.txt file described in the previous section. Due to tool incompatibilities, some tools have a separate copy of the encrypted RTL. The contents of the encrypted containers is identical. The duplication is made since encrypted RTL for one tool may cause errors in other tools if included in all tools' file lists.

All files that should be encrypted within a GRLIB directory are concatenated into one file before encryption. This results in one encrypted file per directory per tool. The list below lists the file names that correspond to vhdlsyn.txt for encrypted RTL and the naming convention used for the encrypted containers.

GRLIB also supports IEEE P1735 encryption. In this case a single container will be provided for several tools, which simplifies project file creations and tool usage. IEEE P1735 encryption is currently supported for Synopsys Synplify, Mentor Modelsim/Questasim, Aldec Riviera, Microsemi tools and Mentor Veloce.

TABLE 70. Encrypted RTL

Tool	File corresponding to vhdlsyn.txt	Naming convention used for encrypted RTL
Aldec Riviera	vhdlmtie.txt	mtie_<directory>.vhd
Cadence tools	vhdlcdse.txt	<directory>.vhdp
Mentor Model/QuestaSim	vhdlmtie.txt	mtie_<directory>.vhd
Synopsys Synplify	vhdlsynpe.txt	synpe_<directory>.vhd
Synopsys Design Compiler	vhdldce.txt	<directory>.vhd.e
Xilinx Vivado	vhdlxile.txt	xile_<directory>.vhd
Xilinx ISE	vhdlise.txt	xise_<directory>.vhd
Synopsys Synplify, Mentor Modelsim/ Questasim, Alderc Riviera, Microsemi tools, Mentor Veloce	vhdlp1735.txt	p1735_<directory>.vhd

Files listed in the tool-specific vhdlsyn.txt file will only be added to the file list for that specific tool. For example, files listed in vhdlxile.txt will only be added to Xilinx Vivado projects.

8.3 Adding an AMBA IP core to GRLIB

8.3.1 Example of adding an existing AMBA AHB slave IP core

An IP core with AMBA interfaces can be easily adapted to fit into GRLIB. If the AMBA signals are declared as standard IEEE-1164 signals, then it is simple a matter of assigning the IEEE-1164 signal to the corresponding field of the AMBA record types declared in GRLIB, and to define the plug&play configuration information, as shown in the example hereafter.

The plug&play configuration utilizes the constants and functions declared in the GRLIB AMBA 'types' package, and the HADDR and HMASK generics.

Below is the resulting entity for the adapted component:

```
library ieee; use ieee.std_logic_1164.all;
library grlib; use grlib.amba.all;

entity ahb_example is
  generic (
    hindex :     integer := 0;
    haddr :     integer := 0;
    hmask :     integer := 16#fff#);
  port (
    rst     : in  std_ulogic;
    clk     : in  std_ulogic;
    ahbsi   : in  ahb_slv_in_type;
    ahbso   : out ahb_slv_out_type);
end;

architecture rtl of ahb_example is

-- component to be interfaced to GRLIB
component ieee_example
    port (
      rst       : in  std_ulogic;
      clk       : in  std_ulogic;
      hsel      : in  std_ulogic;                          -- slave select
      haddr     : in  std_logic_vector(31 downto 0);       -- address bus (byte)
      hwrite    : in  std_ulogic;                          -- read/write
      htrans    : in  std_logic_vector(1 downto 0);        -- transfer type
      hsize     : in  std_logic_vector(2 downto 0);        -- transfer size
      hburst    : in  std_logic_vector(2 downto 0);        -- burst type
      hwdata    : in  std_logic_vector(31 downto 0);       -- write data bus
      hprot     : in  std_logic_vector(3 downto 0);        -- protection control
      hreadyi   : in  std_ulogic;                          -- transfer done
      hmaster   : in  std_logic_vector(3 downto 0);        -- current master
      hmastlock : in  std_ulogic;                          -- locked access
      hreadyo   : out std_ulogic;                          -- transfer done
      hresp     : out std_logic_vector(1 downto 0);        -- response type
      hrdata    : out std_logic_vector(31 downto 0);       -- read data bus
      hsplit    : out std_logic_vector(15 downto 0));      -- split completion
end component;

 -- plug&play configuration
  constant HCONFIG: ahb_config_type := (
    0      => ahb_device_reg (VENDOR_EXAMPLE, EXAMPLE_AHBRAM, 0, 0, 0),
    4      => ahb_membar(memaddr, '0', '0', memmask), others => X"00000000");

begin
  ahbso.hconfig <= HCONFIG; -- Plug&play configuration
  ahbso.hirq <= (others => '0'); -- No interrupt line used
  -- original component
  e0: ieee_example
    port map(
      rst, clk, ahbsi.hsel(ahbndx), ahbsi.haddr, ahbsi.hwrite, ahbsi.htrans, ahbsi.hsize,
     ahbsi.hburst, ahbsi.hwdata, ahbsi.hprot, ahbsi.hready, ahbsi.hmaster,
     ahbsi.hmastlock, ahbso.hready, ahbso.hresp, ahbso.hrdata, ahbso.hsplit);
end;
```

The files containing the entity *ahb_example* the entity for *ieee_example* should be added to GRLIB by listing the files in a *vhdlsyn.txt* file located in a directory that will be scanned by the GRLIB scripts, as described in section 8.2. The paths in *vhdlsyn.txt* can be relative, allowing the VHDL files to be placed outside the GRLIB tree. The entities and packages will be compiled into a library with the same name as the directory that holds the *vhdlsyn.txt* file.

In the *ahb_example* example, the core does not have the ability to assert an interrupt. In order to assert an interrupt, an AHB core must drive the *hirq* vector in the *ahb_slv_out_type* (or *ahb_mst_out_type*) output record. If the core is an APB slave, it should drive the *apb_slv_out_type* record's *pirq* vector. Position *n* of hirq/pirq corresponds to interrupt line *n*. All unused interrupt lines must be driven to '0'.

8.3.2 AHB Plug&play configuration

As described in section 5.3, the configuration record from each AHB unit is sent to the AHB bus controller via the HCONFIG signal. From this information, the bus controller automatically creates the read-only plug&play area.

In the *ahb_example* example in the previous section, the plug&play configuration is held in the constant *HCONFIG*, which is assigned to the output *ahbso.hconfig*. The constant is created with:

```
-- plug&play configuration
 constant HCONFIG : ahb_config_type := (
   0      => ahb_device_reg (VENDOR_EXAMPLE, EXAMPLE_AHBRAM, 0, 0, 0),
   4      => ahb_membar(memaddr, '0', '0', memmask), others => X"00000000");
```

The *ahb_config_type* is an array of 32-bit vectors. Each position in this array corresponds to the same word in the core's plug&play information. Section 5.3.1 describes the plug&play information in the following way: The first word is called the identification register and contains information on the device type and interrupt routing. The last four words are called bank address registers, and contain address mapping information for AHB slaves. The remaining three words are currently not assigned and could be used to provide core-specific configuration information.

The AMBA package (*lib/grlib/amba/amba.vhd*) in GRLIB provides functions that help users create proper plug&play information. Two of these functions are used above. The *ahb_device_reg* function creates the identification register value for an AHB slave or master:

```
ahb_device_reg (vendor, device, cfgver, version, interrupt)
```

The parameters are explained in the table below:

TABLE 71. ahb_device_reg parameters

Parameter	Comments
vendor	Integer Vendor ID. Typically defined in *lib/grlib/amba/devices.vhd*. It is recommended that new cores be added under a new vendor ID or under the contrib vendor ID.
device	Integer Device ID. Typically defined in *lib/grlib/amba/devices.vhd*. The combination of vendor and device ID must not match any existing core as this may lead to your IP core being initialized by drivers for another core.
cfgver	Plug&play information version, only supported value is 0.
version	Core version/revision. Assigned to 5-bit wide field in plug&plat information.
interrupt	Set this value to the first interrupt line that the core drives. Set to 0 if core does not make use of interrupts.

If an IP core only has an AHB master interface, the only position in *HCONFIG* that needs to be specified is the first word:

```
constant hconfig : ahb_config_type := (
  0 => ahb_device_reg ( venid, devid, 0, version, 0),
  others => X"00000000");
```

If an IP core has an AHB slave interface, as in the *ahb_example* example, we also need to specify the memory area(s) that the slave will map. Again, the *HCONFIG* constant from *ahb_example* is:

```
-- plug&play configuration
 constant HCONFIG : ahb_config_type := (
   0      => ahb_device_reg (VENDOR_EXAMPLE, EXAMPLE_AHBRAM, 0, 0, 0),
   4      => ahb_membar(memaddr, '0', '0', memmask), others => X"00000000");
```

The last four words of *ahb_config_type* (positions 4 - 7) are called bank address registers (BARs), and contain memory map information. This information determines address decoding in the AHB controller (AHBCTRL core). Address decoding is described in detail under section 5.3.3. When creating an AHB memory bank, the *ahb_membar* function can be used to automatically generate the correct layout for a BAR:

```
ahb_membar(memaddr, prefetch, cache, memmask)
```

To create an AHB I/O bank, the *ahb_iobar* function can be used:

```
ahb_iobar(memaddr, memmask)
```

The parameters of these functions are described in the table below:

TABLE 72. ahb_membar/ahb_iobar parameters

Parameter	Comments
memaddr	Integer value propagated to BAR.ADDR
memmask	Integer value propagated to BAR.MASK
prefetch	Std_Logic value propagated to prefetchable field (P) in bank address register. Only applicable for AHB memory bars (*ahb_membar* function).
cache	Std_Logic value propagated to cacheable field (C) in bank address register. Only applicable for AHB memory bars (*ahb_membar* function).

An AHB slave can map up to four address areas (it has four bank address registers). Typically, an IP core has one AHB I/O bank with registers and zero or several AHB memory banks that map a larger memory area. One example is the GRLIB DDR2 controller (DDR2SPA) that has the following HCONFIG:

```
constant hconfig : ahb_config_type := (
   0 => ahb_device_reg ( VENDOR_GAISLER, GAISLER_DDR2SP, 0, REVISION, 0),
   4 => ahb_membar(haddr, '1', '1', hmask),
   5 => ahb_iobar(ioaddr, iomask),
   others => zero32);
```

Position four, the first bank address register, defines an AHB memory bank which maps external DDR2 SDRAM memory. Position five, the second bank address register, defines an AHB I/O bank that holds the memory controller's register interface. On this core, the *haddr*, *hmask*, *ioaddr* and *iomask* values are set via VHDL generics.

For IP cores that map multiple memory areas, there is no need for the IP core to decode the address in order to determine which bank that is accessed. The AHB controller decodes the incoming address and selects the correct AHB slave via the HSEL vector. The AHB controller also indicates which bank that is being accessed via the HMBSEL vector, when bank *n* is accessed HMBSEL(*n*) will be asserted.

8.3.3 Example of creating an APB slave IP core

The next page contains an APB slave example core. The IP core has one memory mapped 32-bit register that will be reset to zero. The register can be read or written from register address offset 0. The core's base address, mask and bus index settings are configurable via VHDL generics (*pindex*, *paddr*, *pmask*). The *paddr* and *pmask* VHDL generics are propagated to the APB bridge via the *apbo.pconfig* signal and the index is propagated via the *apbo.pindex* signal. These values are then used by the APB bridge to generate the APB address decode and slave select logic.

Example of APB slave IP core with one 32-bit register that can be read and written:

```
library ieee; use ieee.std_logic_1164.all;
library grlib; use grlib.amba.all; use grlib.devices.all;
library gaisler; use gaisler.misc.all;

entity apb_example is
  generic (
    pindex   : integer := 0;
    paddr    : integer := 0;
    pmask    : integer := 16#fff#);
  port (
    rst    : in  std_ulogic;
    clk    : in  std_ulogic;
    apbi   : in  apb_slv_in_type;
    apbo   : out apb_slv_out_type);
end;

architecture rtl of apb_example is

  constant REVISION : integer := 0;

  constant PCONFIG : apb_config_type := (
    0 => ahb_device_reg (VENDOR_ID, DEVICE_ID, 0, REVISION, 0),
    1 => apb_iobar(paddr, pmask));

  type registers is record
```

```
    reg : std_logic_vector(31 downto 0);
  end record;

  signal r, rin : registers;

begin

  comb : process(rst, r, apbi)
    variable readdata : std_logic_vector(31 downto 0);
    variable v        : registers;
  begin
    v := r;

    -- read register
    readdata := (others => '0');
    case apbi.paddr(4 downto 2) is
      when "000" => readdata := r.reg(31 downto 0);
      when others => null;
    end case;

    -- write registers
    if (apbi.psel(pindex) and apbi.penable and apbi.pwrite) = '1' then
      case apbi.paddr(4 downto 2) is
        when "000" => v.reg := apbi.pwdata;
        when others => null;
      end case;
    end if;

    -- system reset
    if rst = '0' then v.reg :=  (others => '0'); end if;

    rin <= v;
    apbo.prdata <= readdata; -- drive apb read bus
  end process;

  apbo.pirq <= (others => '0');         -- No IRQ
  apbo.pindex <= pindex;                -- VHDL generic
  apbo.pconfig <= PCONFIG;              -- Config constant

-- registers
  regs : process(clk)
  begin
    if rising_edge(clk) then r <= rin; end if;
  end process;

-- boot message

-- pragma translate_off
    bootmsg : report_version
        generic map ("apb_example" & tost(pindex) &": Example core rev " & tost(REVISION));
-- pragma translate_on

end;
```

The steps required to instantiate the *apb_example* IP core in a system are:

- Add the file to a directory covered by the GRLIB scripts (via *libs.txt* and *dirs.txt*)
- Add the file to *vhdlsyn.txt* in the current directory
- Modify the example to use a unique vendor and device ID (see creation of PCONFIG constant)
- Create a component for the *apb_example* core in a package that is also synthesized.
- Include the package in your design top-level
- Instantiate the component in your design top-level

For a complete example, see the General Purpose Register (GRGPREG) IP core located in *lib/gaisler/misc/grgpreg.vhd*. That core is very similar to the example given in this section. The GRGPREG core has a component declaration in the grlib.misc package located at *lib/gaisler/misc/misc.vhd*. Note that both of these files are listed in the *vhdlsyn.txt* file located in the same directory.

8.3.4 APB plug&play configuration

APB slave plug&play configuration is propagated via the *apb_slv_out_type* record's *pconfig* member. The configuration is very similar to that of an AHB slave. The main difference is that APB slaves only have one type of BAR and each APB slave only has one bank. The creation of the PCONFIG array in the previous section looked like:

```
constant PCONFIG : apb_config_type := (
    0 => ahb_device_reg (VENDOR_ID, DEVICE_ID, 0, REVISION, 0),
    1 => apb_iobar(paddr, pmask));
```

The *ahb_device_reg* function has been described in section 8.3.2. The *apb_iobar* function takes the same arguments as the *ahb_iobar* function, also described in section 8.3.2.

8.4 Adding a design to GRLIB

This section explains how to add a new design to GRLIB for users who do not have access to an already supported FPGA board. In this design, the majority of the configuration is hard-coded into the top-level design file. The disadvantage of the method described is the loss of the convenience that the xconfig GUI provides.

8.4.1 Overview

This example is based on the leon3-minimal design in the designs/ directory, but the principles can be applied to LEON5 and NOEL-V designs as well. It can be used to create a minimalistic system for a new FPGA board with low effort. The design includes basic cores like the LEON3 CPU, AMBA bus, memory controller and serial communication interfaces. However, the included memory controller might have to be replaced with one that is compatible with the RAM type on the target board. The serial communication interfaces available in this design are JTAG and UART. The GRMON debug monitor can connect to the design through any of these interfaces.

A minimal GRLIB design requires that at least four files. They should be placed in a new directory ../designs/<design name>.

Makefile	Local makefile for the design. Sets variables for synthesis and calls the main GRLIB makefile.
config.vhd	Design configuration parameters. Generated through xconfig.
leon3mp.vhd	Top level VHD file. The CPU and bus peripherals are instantiated here
leon3mp.ucf	Xilinx constraint file. Maps input/output ports in the top level to pins on the FPGA.

The design example further down covers how to create and modify these files for a board that has a Xilinx FPGA. The Xilinx ISE synthesis workflow is used in the example and is valid for the majority of Xilinx FPGAs.

The first goal in the implementation process is to get a design that it is possible to connect to with GRMON. To achieve this the leon3mp.vhd can mostly be left untouched, but a config.vhd and Makefile needs to be created and is covered in detail in the example. The next step is to replace or configure the memory controller in order to make accesses the on board RAM possible. This guide only covers in detail how to access on-board SRAM.

In order to also be able to simulate the design, the files listed below are required.

testbench.vhd	Testbench VHD file for simulation. Contains an instantiation of leon3mp.vhd and peripherals that are connected to the FPGAs pins like RAM/ROM.
prom.srec	Boot prom for the simulation that starts the program in sram.srec
sram.srec	Contains a test program
wave.do	Adds signals to simulator wave window.

Performing a simulation increases the probability of a successful implementation on the FPGA. When a simulation is performed the AMBA bus controller will check for violations, e.g. if two masters have the same index. It is also suitable to set up a simulation environment in order to test if the memory controller is correctly configured.

8.4.2 Example: Adding a template design for Nexys4

This section describes how to use the leon3-mininal design example to create a basic design for a board. The process covered here will make it possible to connect to the design from GRMON and to execute programs in a LEON3 CPU. The Digilent Nexys 4 broad is used as an example.

The first step is to generate a config.vhd file that has a configuration that matches the FPGA. The easiest way is to run "make xconfig" in "../designs/leon3mp/" and then copy over the config.vhd to the design directory (e.g ../designs/leon3-minimal). In the xconfig GUI under "Synthesis" set "Target technology" to the FPGA type. For the Nexys4 "Xilinx-Artix7" is selected. The other parameters in

the xconfig GUI are hardcoded in the top design directly. Changing them in xconfig will therefore have no effect.

Second, the UCF constraint file should be created or downloaded. In most cases it is delivered with the FPGA documentation. Name it leon3mp.ucf and place it in the leon3-mininal design directory.

Creating the Makefile

The "Makefile" file is required in order for the make scripts and synthesis tools to compile the right VHDL files and create a configuration file for the correct FPGA. The structure of the Makefile example below is aimed specifically at Xilinx FPGAs for Xilinx ISE Synthesis. Other tools and FPGAs from other vendors do require extra parameters to be set.

In order to make this example work with another FPGA the parameters TECHNOLOGY, PART, PACKAGE and SPEED have be changed. The possible values of these parameters can be looked up in Xilinx ISE under Project -> Design Properties. The parameters corresponding name in the ISE GUI is written as a comment.

```
GRLIB=../..                             # Path to the root folder of GRLIB
TOP=leon3mp                             # The entity name of the top design
TECHNOLOGY=Artix7                       # The FPGA Family. These are listed in ISE
                                        # under Project -> Design Properties.
PART=XC7A100T                           # FPGA device name
PACKAGE=csg324                          # FPGA package
SPEED=-2                                # FPGA speed grade (-1 is the slowest)
DEVICE=$(PART)-$(PACKAGE)$(SPEED)       # Combined device name

UCF=$(TOP).ucf                          # The filename of the ucf file in the design's
                                        # directory
EFFORT=high                             # Effort level for Map and Place-and-Route
VHDLSYNFILES=config.vhd ahbrom.vhd \    # The VHDL files that are in the design's directory
             leon3mp.vhd
VHDLSIMFILES=testbench.vhd              # The VHDL file containing the testbench
SIMTOP=testbench                        # The entity name of the test bench top design
CLEAN=soft-clean
TECHLIBS = unisim                       # unisim is used for Xilinx FPGAs

# Libraries, directories and files in GRLIB that should not be compiled for this design
LIBSKIP = core1553bbc core1553brm core1553brt gr1553 corePCIF \
                      tmtc ihp usbhc spw
DIRSKIP = b1553 pci/pcif leon2 leon2ft crypto satcan pci leon3ft ambatest can \
                      usb grusbhc spacewire ascs slink hcan \
                      leon4 leon4v0 l2cache pwm gr1553b iommu
FILESKIP = grcan.vhd

include $(GRLIB)/bin/Makefile           # Starts the main GRLIB Makefiles
include $(GRLIB)/software/leon3/Makefile
```

Practice used in other designs

The other designs that are included in GRLIB have their Makefile separated into two files. One in a board directory in boards/ and one in a design directory in designs/. The boards directory is intended to hold properties that can be shared between multiple designs for that specific board. E.g. the variables TECHNOLOGY, PART, PACKAGE, SPEED and DEVICE are instead defined in the Makefile.inc in the boards directory. The naming convention used for the design directories is (CPU)-(manufacturer)-(board), and the naming convention for the boards directories is (manufacturer)-(board)-(FPGA).

A board directory will often contain the files listed.

Makefile.inc	Makefile that sets variables that concern device and board organization.
default.ut	FPGA Program file generation parameters for Xilinx FPGAs. The available parameters can be found in the Xilinx ISE GUI in the "Generate Programming File" properties.
prom.cmd	Command file used with iMPACT to program the proms on the board
fpga.cmd	Command file used with iMPACT to program the FPGA directly
prom-usb.cmd	PROM programming over USB
leon3mp.ucf	Constraints file (can be placed in design directory)
default.sdc	Constraints file for Synplify (can be placed in design directory)

In the Makefile in the design directory the variables like TECHNOLOGY, PART, PACKAGE, SPEED and DEVICE are instead replaced with an include of the Makefile.inc in the board directory.

```
BOARD=digilent-nexys4-xc7a100t                # Directory name specific to an FPGA board
include $(GRLIB)/boards/$(BOARD)/Makefile.inc # Includes the Makefile.inc for the borad
```

If there exists a constraints file in the board directory it is still possible to use a constraints file that is local to a particular design. If the UCF variable points to the UCF file in the board directory is it is assigned UCF=$(GRLIB)/boards/$(BOARD)/$(TOP).ucf. In order to use the local UCF in the design directory the variable is instead assigned UCF=$(TOP).ucf.

The cmd files are scripts for iMPACT and can be generated by running it as a GUI. In the directory from where iMPACT was started a file "_impact.cmd" is created upon exit. It will contain the commands that where executed in the GUI mode session and might require some cleanup. The cmd files can not be overridden locally for a specific design and have to be placed in the boards directory.

Description of leon3mp.vhd

This section explains the leon3mp.vhd example file that exists in the LEON3-MINIMAL design and the modifications have to be done to it.

The entity declaration in this leon3mp.vhd example contains the minimal number of generics and ports. The four generics specify the technology used and are assigned in the generated config.vhd file.

```
entity leon3mp is
  generic (
    fabtech  : integer := CFG_FABTECH;
    memtech  : integer := CFG_MEMTECH;
    padtech  : integer := CFG_PADTECH;
    clktech  : integer := CFG_CLKTECH);
```

A minimal design needs input/output signals for at least clock, reset and communication links. In addition, extra signals are required in order to access external RAM and boot-(EEP)ROM that vary between different boards and memory types. All these signals have to be mapped to the correct FPGA pins in the leon3mp.ucf file. Either the signals have to be renamed in the ucf file or in leon3mp.vhd.

```
port (
    clk             : in    std_ulogic;        -- FPGA main clock input

    -- Buttons & LEDs
    btnCpuResetn    : in    std_ulogic;        -- Reset button
    Led             : out   std_logic_vector(15 downto 0);

    -- Onboard Cellular RAM
    RamOE           : out   std_ulogic;
    RamWE           : out   std_ulogic;

    RamAdv          : out   std_ulogic;
    RamCE           : out   std_ulogic;
    RamClk          : out   std_ulogic;
    RamCRE          : out   std_ulogic;
    RamLB           : out   std_ulogic;
    RamUB           : out   std_ulogic;

    address         : out   std_logic_vector(22 downto 0);
    data            : inout std_logic_vector(15 downto 0);

    -- USB-RS232 serial interface
    RsRx            : in    std_logic;
    RsTx            : out   std_logic
  );
end;
```

After the port mapping follows the signal and constant declaration section. There are four constants declared that are used to set the frequency of the LEON3 CPU and system bus.

```
constant clock_mult : integer := 10;       -- Clock multiplier
constant clock_div  : integer := 20;       -- Clock divider
constant BOARD_FREQ : integer := 100000;   -- Clock input frequency in KHz
constant CPU_FREQ   : integer := BOARD_FREQ * clock_mult / clock_div; -- CPU freq in KHz
```

On most boards the FPGAs input clock frequency is within 50 - 200 MHz. The Nexys4 board has an input clock that is 100 MHz that enters through the "clk" input signal. Therefore the BOARD_FREQ constant is set to 100 000 kHz.

In this example the LEON3 CPU clock frequency is scaled to half the input clock frequency by setting the clock multiplier to 10 and divider to 20. It is recommended to keep the system frequency low at this stage in the development process in order to avoid a malfunctioning design because of timing errors. The synthesis tool produces a warning in case of a timing error, but the bit file is still generated.

The frequency conversion is carried out in the "clkgen" IP-core that instantiates a DCM, PLL or an equivalent clock generator that is suitable for the FPGA. However, the valid intervals of the multiplier

and divider parameters vary between different FPGAs, but the parameters suggested here are likely to be valid in many cases. The new clock (50 MHz) is assigned to the "clkm" signal.

```
clkgen0 : clkgen
    generic map (fabtech, clock_mult, clock_div, 0, 0, 0, 0, 0, BOARD_FREQ, 0)
    port map (clk, gnd, clkm, open, open, open, open, cgi, cgo, open, open, open);
```

The btnCpuResetn signal originates from a button on the board and does therefore contain glitches. Therefore the rstgen IP-core is used to create a clean reset signal named rstn. The signal that is output when a button is pressed varies between FPGA boards. The reset button on the Nexys4 board produces a low value when pressed, and therefore the "acthigh" generic is set to 0. If it is uncertain how the button on the board behaves and GRMON does not connect it can be attempted to hold the reset button while trying to connect again.

```
rst0 : rstgen generic map (acthigh => 0)  -- Change to 1 if reset button is act high
    port map (btnCpuResetn, clkm, lock, rstn, rstraw);
```

The easiest way to connect to the board is through a serial interface like RS-232 and/or JTAG. On Xilinx FPGAs JTAG is the easiest since it is just to instantiate the ahbjtag core and the Xilinx tools will connect the input/output signals. When creating a Xilinx design the tck, tms, tdi and tdo are dummy signals, but have to be assigned for other FPGA manufacturers. In order for GRMON to connect through JTAG an argument needs to be passed to it that depends on the JTAG vendor (e.g "-digilent", "-xilusb" or "-jtag"). Refer to the GRMON manual for more details.

```
ahbjtag0 : ahbjtag generic map(tech => fabtech, hindex => 3)
    port map(rstn, clkm, tck, tms, tdi, tdo, ahbmi, ahbmo(3),
             open, open, open, open, open, open, open, gnd);
```

One other option is to use a serial connection which requires one input and one output signal from the FPGA. The RsRx signal is for receiving and RsTx signal is for transmission. The RsRx and RsTx signals are assigned to the internal signals (dui.rxd and duo.txd) through pads. Each of the duo.txd and duo.txd signals can also be mapped to leds in order to get visual feedback when there is activity.

```
dcom0 : ahbuart generic map (hindex => 1, pindex => 4, paddr => 7)
    port map (rstn, clkm, dui, duo, apbi, apbo(4), ahbmi, ahbmo(1));
  dsurx_pad : inpad generic map (tech  => padtech) port map (RsRx, dui.rxd);
  dsutx_pad : outpad generic map (tech => padtech) port map (RsTx, duo.txd);
```

At this stage it is suitable to test if it is possible to connect to the FPGA with GRMON through either JTAG or RS-232. Create the bitstream by running "make ise" and program the FPGA. When GRMON successfully connects the remaining work is to get the on board memory working. In the introduction chapter in the GRLIB IP Core User's Manual, there is a table of available memory controllers and their function. Since the configuration differs between various kinds of memories, the method is explained by using the SRAM implementation as an example.

The first step would be to instantiate a memory controller. The Nexys4 has a 16-bit wide SRAM and therefore the MCTRL is instantiated. The generic that controls where the SRAM is mapped in address space is left at the default address 0x40000000. This is the recommended address since it is where the binaries are uploaded by default.

```
sr1 : mctrl
    generic map (hindex => 5, pindex => 0, paddr => 0, rommask => 0,
                 iomask => 0, ram8 => 0, ram16 => 1,srbanks=>1)
    port map (rstn, clkm, memi, memo, ahbsi, ahbso(5), apbi, apbo(0), wpo, open);

  memi.brdyn  <= '1';
  memi.bexcn  <= '1';
  memi.writen <= '1';
  memi.wrn    <= "1111";
  memi.bwidth <= "01";       -- Sets data bus width for PROM accesses.

  -- Bidirectional data bus
  bdr  : iopadv generic map (tech => padtech, width => 8)
    port map (data(7 downto 0), memo.data(23 downto 16),
              memo.bdrive(1), memi.data(23 downto 16));
  bdr2 : iopadv generic map (tech => padtech, width => 8)
    port map (data(15 downto 8), memo.data(31 downto 24),
              memo.bdrive(0), memi.data(31 downto 24));

  -- Out signals to memory
  addr_pad : outpadv generic map (tech => padtech, width => 23) -- Address bus
    port map (address, memo.address(23 downto 1));
  oen_pad : outpad generic map (tech => padtech)   -- Output Enable
    port map (RamOE, memo.oen);
  cs_pad : outpad generic map (tech => padtech)    -- SRAM Chip select
    port map (RamCE, memo.ramsn(0));
  lb_pad : outpad generic map (tech => padtech)
    port map (RamLB, memo.mben(0));
  ub_pad : outpad generic map (tech => padtech)
    port map (RamUB, memo.mben(1));
  wri_pad : outpad generic map (tech => padtech)   -- Write enable
```

```
    port map (RamWE, memo.writen);
```

The memory data bus is bidirectional and therefore iopads controled by the MCTRL must be used. The MCTRL has one record that contains incoming signals into the core (memi) and one record that contains outgoing signals (memo). The memo.bdrive signal decides if the data bus is read into memi.data or is driven with value in memo.data. Further details about the MCTRL and its signals can be found in the GRLIB IP Core User's Manual.

When it comes to the memo signals it is likely that some SRAM chips will not require all the memo signals. E.g. other chips might not require the mben signals. There can also be a difference in how the address bus functions on different boards. Since the Nexys4 board has a 16 bit wide memory bus accesses are done in 2 byte blocks. The LSB address bit in the memo.address is therefore not assigned to the address bus. However another board could have an 8 bit PROM and a 32 bit SRAM and would therefore require the LSB address bit in order to access the PROM.

After the memory controller has been added the design it is suggested to do a simulation. Then create a new configuration file and program the FPGA. The first goal when trying to implement memory access is to be able to write to the memory and detect that something changed from before. In this development phase it is suitable to use long memory latencies in order to ensure that a failure is not related to incorrect timings.

It is possible to set the various timings for the MCTRL core through GRMON. Since in this example the MCTRL is used together with SRAM the read and write latency of the SRAM can be set by passing "-ramrws 3" and "-ramwws 3" as arguments when starting GRMON.

The memory contents can be shown in GRMON with the command "mem 0x40000000" and written with "wmem 0x40000000 0x12345678". If it appears that the data in the memory is changing but is irregular it is suggested to zero out all the memory using "wash 0x40000000 0x410000000" in GRMON. Thereafter perform one write and observe. If the data changes at the right address but is incorrect it is likely that the timing is wrong. If the data instead appears partially correct but is spread out over multiple words in memory the addressing is likely to be incorrect.

One other RAM alternative is to use the block RAM on the FPGA by instantiating the AHBRAM IP-core. The maximum size might range from 100 kB up to a few MB depending on the amount of block RAM available. The Nexys4 boards FPGA has 512 kB of block RAM in total, which is sufficient for many applications.

Simulation test bench

A testbench is provided in the LEON3-MINIMAL design directory. This section describes what areas of the simulation have to be modified to match different FPGA boards and how a test bench in the GRLIB is constructed in general.

The major advantage of setting up a simulation is the ability to find errors in the design before attempting the time consuming generation of the FPGA bitstream. A successful simulation will not guarantee that the FPGA design works but will increase the probability of a successful hardware implementation. See the implementation flow chapter in this document on how to compile and start a simulation with your simulation software.

Having a simulation for a design makes it possible to test that the memory controller is set up correctly and that input and output signals from the FPGA design are assigned with the correct function. Although if an input or output signal in the top level design is incorrectly mapped in the constraints file, the error will not be detected through simulation. Some types of miss configurations and incorrect signal assignments in the FPGA design will also be detected. For example at the simulation start the various bus controllers in the system will generate and error if any of the masters or slaves have colliding bus indexes or if slaves address mapping overlap.

The test bench is defined in the testbench.vhd file that is provided in the design directory. In it the top level design from the leon3mp.vhd file is instantiated together with on board peripherals like simulation models for SRAM. For examples how to use other RAM simulation models than SRAM refer to the test benches from other designs.

```
d3 : entity work.leon3mp
    generic map (fabtech, memtech, padtech, clktech, disas, dbguart, pclow)
    port map (
      clk     => clk,
      btnCpuResetn => rstn,

      -- PROM
      address   => address(22 downto 0),
      data      => data(31 downto 16),
```

```
      RamOE     => oen,
      RamWE     => writen,
      RamCE     => RamCE,

      -- AHB Uart
      RsRx      => dsurx,
      RsTx      => dsutx,

      -- Output signals for LEDs
      led       => led
      );

-- Memory Simulation Models
sram0 : sram
  generic map (index => 4, abits => 24, fname => sdramfile)
  port map (address(23 downto 0), data(31 downto 24), RamCE, writen, oen);

sram1 : sram
  generic map (index => 5, abits => 24, fname => sdramfile)
  port map (address(23 downto 0), data(23 downto 16), RamCE, writen, oen);
```

By default a test bench in the design folder execute a small system test program in the LEON processor. Upon simulation start the SRAM is loaded with a binary from an SREC file, usually named "ram.srec", which contains a test program. The file name is not assigned directly the to SRAM model but rather through a constant named sdramfile or sramfile for convenience. It is possible to execute most other binaries in simulation too as log as the binary is contained in an SREC file. The other binary can then be simulated by changing the sdramfile constant to point its SREC file.

Since the Nexys4 has a 16 bit wide data bus two 8-bit SRAM models are instantiated. Their index generic is set to four and five, which sets the SRAM models to behave appropriately for a 16-bit wide data bus. For a 32 bit data bus four SRAM models would be instantiated with their indexes assigned between zero and three. An 8 bit wide data bus would require one SRAM model instantiation that has its index generic set to six. Examples of all these configuration can be found in test benches for other designs.

Before the program in RAM is executed the processor boots from a ROM. It contains a small initialization program that clears registers and setups design specific configuration. This process is used to configure the LEON system simulation. However, when running on the design on the FPGA a PROM is not required since the configuration can be applied though GRMON.

The ROM can be instantiated in two ways depending on if the FPGA board has on board PROM or not. If there is no on board PROM the ROM is instantiated as an AHB slave with the AHBROM IP core in the leon3mp.vhd. The ROM is thus also instantiated in the FPGA design. Since there is no on board PROM on the Nexys4 the AHBROM method is used in the example directory.

```
brom : entity work.ahbrom
    generic map (hindex => 6, haddr => CFG_AHBRODDR, pipe => CFG_AHBROPIP)
    port map ( rstn, clkm, ahbsi, ahbso(6));
```

If there is a PROM on board it is added to the testbench.vhd and accessed though the same address and data bus as the SRAM. The PROM is also instantiated with the SRAM simulation model since the PROM read accesses are performed in the same way as for SRAM. The SRAM simulation model that is used as a PROM is instead loaded with the "prom.srec" file.

Before it is possible to generate the ram.srec, prom.srec and ahbrom.vhd it is necessary to have valid prom.h and systest.c files in the design directory, which are provided. The systest.c file contains the main function which then calls different test modules. In this test bench example it does only perform a basic test and does not require modifications.

The prom.h file contains constants that are applied to various configuration registers in the LEON system during the boot. At this stage the MCTRL memory controller is being configured to properly access the SRAM. The data written into the MCTRL registers is defined by the constants MCFG1, MCFG2 and MCFG3 and correspond to three of the memory controllers registers. The SRAM is configured through the MCFG2 constant and is used to set the data bus width and data access latency etc. The register is described in further detail in the GRLIB IP Core User's Manual. In order to configure other memory controllers and memory types it might be necessary to add or modify a constant in prom.h.

The generation of the sram.srec and prom.srec files is done be by running "make soft". To generate the AHBROM IP core run "make ahbrom.vhd", which will create the ahbrom.vhd file.

Within the testbench.vhd there is a section that asserts the processor's error signal, which indicates if the CPU entered the error state. In the leon3mp top level design this signal is assigned to the on board led(3) and made active high. If the led(3) signal ever goes high the simulation will immediately stop.

If an error occurs because of miss configured RAM the AHB address bus (ahbsi.haddr) will give a hint when and at what address a faulty data access occurred.

```
led(3) <= 'L';            -- ERROR pull-down
  error <= not led(3);

  iuerr : process
  begin
    wait for 5 us;
    assert (to_X01(error) = '1')
      report "*** IU in error mode, simulation halted ***"
      severity failure;
  end process;
```

Within the leon3mp top level design a test reporting unit is instantiated. When the simulation runs, the test reporting unit will print to the console whether the various test modules in the test program succeed or not. Notice that the --pragma translate on/off will remove the unit from the hardware synthesis but will leave it in the simulation.

```
--pragma translate_off
  test0 : ahbrep generic map (hindex => 4, haddr => 16#200#)
    port map (rstn, clkm, ahbsi, ahbso(4));
--pragma translate_on
```

8.5 Using verilog code

Verilog does not have the notion of libraries, and although some CAD tools supports the compilation of verilog code into separate libabries, this feature is not provided in all tools. Most CAD tools however support mixing of verilog and VHDL, and it is therefore possible to add verilog code to the work library. Adding verilog files is done in the same way as VHDL files, except that the verilog file names should appear in *vlogsyn.txt* and *vlogsim.txt*.

The basic steps for adding a synthesizable verilog core are:

- Create a directory and add it to *libs.txt* and *dirs.txt* as described in section 8.2, or use an existing directory.
- List the verilog files in a *vlogsyn.txt* file located in the selected directory
- Create a VHDL component declaration for the verilog top-level

In case the verilog IP core will be instantiated directly in the design, the component can be added to a package. This package can then be referenced in the design's top-level and the verilog core can be instantiated using the VHDL component.

In case the verilog IP core has an AMBA interface, it will likely require wrapping in order to add the GRLIB AMBA plug&play signals. To do this, the procedure described in section 8.3.1 can be used, where the *ieee_example* component declaration would be the VHDL component for the verilog IP core.

As mentioned above, all CAD tools may not support compiling verilog code into a library. Should the strategy above not work, another option is to list the verilog files in the *VERILOGSYNFILES* variable defined in the (template) design's Makefile and to create the VHDL component of the verilog IP core in the design's top-level.

Other issues that may arise include propagation problems of VHDL generics to Verilog parameters (issues crossing the language barrier). Many tools handle propagation of integer and string values correctly. Should there be any problems, it is recommended to change the Verilog code to remove the parameters.

Preliminary SystemVerilog support is available in selected tools, namely Mentor Graphics ModelSim, Altera Quartus II and Synopsys Synplify. SystemVerilog files should be added to *svlogsyn.txt* and *svlogsim.txt* in a way analogous to the one used for regular Verilog files described above. SystemVerilog simulation and synthesis is still experimental.

8.6 Adding portabilty support for new target technologies

8.6.1 General

New technologies to support portability can be added to GRLIB without the need to modify any previously developed designs. This is achieved by technology independent encapsulation of components such as memories, pads and clock buffers. The technology mapping is organized as follows:

- A VHDL library with the technology simulation models is placed in lib/tech/library

- Wrappers for memory, pads, PLL and other cells are placed under lib/techmap/library
- All 'virtual' components with technology mapping are placed in lib/techmap/maps
- Declaration of all 'virtual' components and technologies is made in lib/techmap/gencomp/gencomp.vhd

An entity that uses a technology independent component needs only to make the techmap.gencomp package visible, and can then instantiate any of the mapped components.

8.6.2 Adding a new technology

A new technology is added in four steps. First, a VHDL library is created in the lib/tech/library location. Secondly, a package containing all technology specific component declarations is created and the source code file name is added to the 'vhdlsyn.txt' or 'vlogsyn.txt' file. Third, simulation models are created for all the components and the source file names are added to the 'vhdlsim.txt' or 'vlogsim.txt' file. A technology constant is added to the GENCOMP package defined in the TECHMAP library. The library name is not put in lib/libs.txt but added either to the FPGALIBS or ASICLIBS in bin/Makfile.

The technology library part is completed and the components need to be encapsulated as described in the next section. As an example, the ASIC memories from Virage are defined in the VIRAGE library, located in the lib/tech/virage directory. The component declarations are defined in the VCOMPONENTS package in the virage_vcomponents.vhd file. The simulation models are defined in virage_simprims.vhd.

8.6.3 Encapsulation

Memories, pads and clock buffers used in GRLIB are defined in the TECHMAP library. The encapsulation of technology specific components is done in two levels.

The lower level handles the technology dependent interfacing to the specific memory cells or macro cells. This lower level is implemented separately for each technology as described hereafter.

For each general type of memory, pad or clock buffer, an entity/architecture is created at the lower level. The entity declarations are technology independent and have similar interfaces with only minor functional variations between technologies. The architectures are used for instantiating, configuring and interfacing the memory cells or macro cells defined for the technology.

A package is created for each component type containing component declarations for the aforementioned entities. Currently there is a separate memory, pad and clock buffer package for each technology. The components in these packages are only used in the higher level, never directly in the designs or IP cores.

The higher level defines a technology independent interface to the memory, pad or clock buffer. This higher level is implemented only once and is common to all technologies.

For each general type of memory, pad or clock buffer, an entity/architecture is created at the higher level. The entity declarations are technology independent. The architectures are used for selecting the relevant lower level component depending on the value of the `tech` and `memtech` generics.

A package is created for each component type containing component declarations for the aforementioned entities. Currently there is a separate memory, pad and clock buffer package. The components declared in these packages are used in the designs or by other IP cores. The two level approach allows each technology to be maintained independently of other technologies.

8.6.4 Memories

The currently defined memory types are single-port, dual-port, two-port and triple-port synchronous RAM. The encapsulation method described in the preceding section is applied to include a technology implementing one of these memory types.

For example, the ASIC memory models from Virage are encapsulated at the lower level i the `lib/techmap/virage/mem_virage_gen.vhd` file. Specifically, the single-port RAM is defined in the VIRAGE_SYNCRAM entity:

```
entity virage_syncram is
  generic (
    abits    :     integer := 10;
    dbits    :     integer := 8 );
  port (
    clk      : in  std_ulogic;
    address  : in  std_logic_vector(abits -1 downto 0);
```

```
    datain   : in  std_logic_vector(dbits -1 downto 0);
    dataout  : out std_logic_vector(dbits -1 downto 0);
    enable   : in  std_ulogic;
    write    : in  std_ulogic);
end;
```

The corresponding architecture instantiates the Virage specific technology specific memory cell, e.g. hdss1_256x32cm4sw0 shown hereafter:

```
architecture rtl of virage_syncram is
  signal d, q, gnd : std_logic_vector(35 downto 0);
  signal a : std_logic_vector(17 downto 0);
  signal vcc : std_ulogic;
  constant synopsys_bug : std_logic_vector(37 downto 0) := (others => '0');
begin

  gnd <= (others => '0'); vcc <= '1';
  a(abits -1 downto 0) <= address;
  d(dbits -1 downto 0) <= datain(dbits -1 downto 0);
  a(17 downto abits) <= synopsys_bug(17 downto abits);
  d(35 downto dbits) <= synopsys_bug(35 downto dbits);
  dataout <= q(dbits -1 downto 0);
  q(35 downto dbits) <= synopsys_bug(35 downto dbits);

  a8d32 : if (abits = 8) and (dbits <= 32) generate
    id0 : hdss1_256x32cm4sw0
      port map (a(7 downto 0), gnd(7 downto 0),clk,
                      d(31 downto 0), gnd(31 downto 0), q(31 downto 0),
                      enable, vcc, write, gnd(0), gnd(0), gnd(0), gnd(0), gnd(0));
  end generate;
  ...
end rtl;
```

The `lib/techmap/virage/mem_virage.vhd` file contains the corresponding component declarations in the MEM_VIRAGE package.

```
package mem_virage is
  component virage_syncram
  generic (
    abits    :     integer := 10;
    dbits    :     integer := 8 );
  port (
    clk      : in  std_ulogic;
    address  : in  std_logic_vector(abits -1 downto 0);
    datain   : in  std_logic_vector(dbits -1 downto 0);
    dataout  : out std_logic_vector(dbits -1 downto 0);
    enable   : in  std_ulogic;
    write    : in  std_ulogic);
  end component;
  ...
end;
```

The higher level single-port RAM model SYNCRAM is defined in the `lib/techmap/maps/syncram.vhd file`. The entity declaration is technology independent:

```
entity syncram is
  generic (
    tech     :     integer := 0;
    abits    :     integer := 6;
    dbits    :     integer := 8 );
  port (
    clk      : in  std_ulogic;
    address  : in  std_logic_vector((abits -1) downto 0);
    datain   : in  std_logic_vector((dbits -1) downto 0);
    dataout  : out std_logic_vector((dbits -1) downto 0);
    enable   : in  std_ulogic;
    write    : in  std_ulogic);
end;
```

The corresponding architecture implements the selection of the lower level components based on the MEMTECH or TECH generic:

```
architecture rtl of syncram is
begin
  inf : if tech = infered generate
    u0 : generic_syncram generic map (abits, dbits)
         port map (clk, address, datain, dataout, write);
  end generate;
  ...
  vir  : if tech = memvirage generate
    u0 : virage_syncram generic map (abits, dbits)
         port map (clk, address, datain, dataout, enable, write);
  end generate;
  ...
end;
```

The `lib/techmap/gencomp/gencomp.vhd` file contains the corresponding component declaration in the GENCOMP package:

```
package gencomp is
  component syncram
  generic (
    tech     :     integer := 0;
    abits    :     integer := 6;
    dbits    :     integer := 8);
  port (
    clk      : in  std_ulogic;
    address  : in  std_logic_vector((abits -1) downto 0);
    datain   : in  std_logic_vector((dbits -1) downto 0);
    dataout  : out std_logic_vector((dbits -1) downto 0);
    enable   : in  std_ulogic;
    write    : in  std_ulogic);
  end component;
  ...
end;
```

The GENCOMP package contains component declarations for all portable components, i.e. SYNCRAM, SYNCRAM_DP, SYNCRAM_2P and REGFILE_3P.

8.6.5 Pads

The currently defined pad types are in-pad, out-pad, open-drain out-pad, I/O-pad, open-drain I/O pad, tri-state output-pad and open-drain tri-state output-pad. Each pad type comes in a discrete and a vectorized version.

The encapsulation method described in the preceding sections is applied to include a technology implementing these pad types.

The file structure is similar to the one used in the memory example above. The pad related files are located in `lib/techmap/maps`. The `lib/techmap/gencomp/gencomp.vhd` file contains the component declarations in the GENCOMP package.

8.6.6 Clock generators

There is currently only one defined clock generator types named CLKGEN.

The encapsulation method described in the preceding sections is applied to include a technology implementing clock generators and buffers.

The file structure is similar to the one used in the memory example above. The clock generator related files are located in `lib/techmap/maps`. The CLKGEN component is declared in the GENCOMP package.

8.7 Extending the xconfig GUI configuration

8.7.1 Introduction

Each template design has a simple graphical configuration interface that can be started by issuing *make xconfig* in the template design directory. The tool presents the user with configuration options and generates the file *config.vhd* that contains configuration constants used in the design.

The subsections below describe how to create configuration menus for a core and then how to include these new options in xconfig for an existing template design.

8.7.2 IP core xconfig files

Each core has a set of files that are used to generate the core's xconfig menu entries. As an example we will look at the GRGPIO core's menu. The xconfig files are typically located in the same directory as the core's HDL files (but this is not a requirement). For the GRGPIO core the xconfig files are:

```
$ ls lib/gaisler/misc/grgpio.in.*

lib/gaisler/misc/grgpio.in
lib/gaisler/misc/grgpio.in.h
lib/gaisler/misc/grgpio.in.help
lib/gaisler/misc/grgpio.in.vhd
```

We will start by looking at the *grgpio.in* file. This file defines the menu structure and options for the GRGPIO core:

```
    bool 'Enable generic GPIO port              ' CONFIG_GRGPIO_ENABLE
    if [ "$CONFIG_GRGPIO_ENABLE" = "y" ]; then
      int 'GPIO width             '   CONFIG_GRGPIO_WIDTH 8
      hex 'GPIO interrupt mask    '   CONFIG_GRGPIO_IMASK 0000
    fi
```

The first line defines a boolean option that will be saved in the variable *CONFIG_GRGPIO_ENABLE*. This will be rendered as a yes/no question in the menu. If this constant is set to yes ('y') then the user will be able to select two more configuration options. First the width, which is defined as an integer (int), and the interrupt mask which is defined as a hexadecimal value (hex).

The GUI has a help option for each item in the menu. When a user clicks on the help button a help text can be optionally displayed. The contents of the help text boxes is defined in the file that ends with *.in.help*, in this case *grgpio.in.help*:

```
GPIO port
CONFIG_GRGPIO_ENABLE
  Say Y here to enable a general purpose I/O port. The port can be
  configured from 1 - 32 bits, whith each port signal individually
  programmable as input or output. The port signals can also serve
  as interrupt inputs.

GPIO port witdth
CONFIG_GRGPIO_WIDTH
  Number of bits in the I/O port. Must be in the range of 1 - 32.

GPIO interrupt mask
CONFIG_GRGPIO_IMASK
  The I/O port interrupt mask defines which bits in the I/O port
  should be able to create an interrupt.
```

As can be seen above, each help entry consists of a topic, the name of the variable used in the menu and the help text.

The two remaining files (*grgpio.in.h* and *grgpio.in.vhd*) are used when generating the *config.vhd* file for a design. *config.vhd* typically consists of a set of lines for each core where the first line decides if the core should be instantiated in the design and the following lines contain configuration options. For the GRGPIO core, the file *grgpio.in.vhd* defines that the following constants should be included in *config.vhd*:

```
-- GPIO port
  constant CFG_GRGPIO_ENABLE : integer := CONFIG_GRGPIO_ENABLE;
  constant CFG_GRGPIO_IMASK  : integer := 16#CONFIG_GRGPIO_IMASK#;
  constant CFG_GRGPIO_WIDTH  : integer := CONFIG_GRGPIO_WIDTH;
```

In the listing above, we see a mix of VHDL and the constants defined in the menus (see listing for *grgpio.in* above). The value we select for *CONFIG_GRPIO_ENABLE* will be assigned to the VHDL constant *CFG_GRGPIO_ENABLE*. In the menu we defined *CONFIG_GRGPIO_IMASK* as a hexadecimal value. The VHDL notation for this is to enclose the value in *16#..#* and this is done for the *CFG_GRGPIO_IMASK* constant.

When exiting the xconfig tool, the *.in.vhd* files for all cores will be concatenated into one file. Then a pre-processor will be used to replace all the variables defined in the menus (for instance *CONFIG_GRGPIO_ENABLE*) into the values they represent. In this process, additional information is inserted via the *.in.vhd.h files*. The contents of *grgpio.in.h* is:

```
#ifndef CONFIG_GRGPIO_ENABLE
#define CONFIG_GRGPIO_ENABLE 0
#endif
#ifndef CONFIG_GRGPIO_IMASK
#define CONFIG_GRGPIO_IMASK 0000
#endif
#ifndef CONFIG_GRGPIO_WIDTH
#define CONFIG_GRGPIO_WIDTH 1
#endif
```

This file is used to guarantee that the *CONFIG_* variable always exist and are defined to sane values. If a user has disabled *CONFIG_GRGPIO_ENABLE* via the configuration menu, then this variable and all the other GRGPIO variables will be undefined. This would result in a *config.vhd* entry that looks like:

```
-- GPIO port
  constant CFG_GRGPIO_ENABLE : integer := ;
  constant CFG_GRGPIO_IMASK  : integer := 16##;
  constant CFG_GRGPIO_WIDTH  : integer := ;
```

... and lead to errors during compilation. This is prevented by *grgpio.in.h* above, where all undefined variables are defined to sane values. It is also possible to place additional intelligence in the *.in.h* file where dependencies between variables can be expressed in ways that would be complicated in the menu definition in the *.in* file.

8.7.3 xconfig menu entries

The menu entries to include in xconfig is defined for each template design in the file *config.in*. As an example we will look at the *config.in* file for the design *leon3-gr-xc3s-1500*. In *designs/leon3-gr-xc3s-1500/config.in* we find the entry for the GRGPIO port (described in the previous section) as part of one of the submenus:

```
mainmenu_option next_comment
  comment 'UART, timer, I/O port and interrupt controller'
    source lib/gaisler/uart/uart1.in
    if [ "$CONFIG_DSU_UART" != "y" ]; then
      source lib/gaisler/uart/uart2.in
    fi
    source lib/gaisler/leon3/irqmp.in
    source lib/gaisler/misc/gptimer.in
    source lib/gaisler/misc/grgpio.in
  endmenu
```

These lines will create a submenu named *UART, timer, I/O port and interrupt controller* and under this submenu include the options for the two UART cores, interrupt controller, timer unit and GPIO port. When the *.in* file for a core is specified in *config.in*, the xconfig tool will automatically also use the corresponding *.in.h* and *.in.vhd* files when generating the *config.vhd* file.

8.7.4 Adding new xconfig entries

In this section we will extend the menu in the *leon3-gr-xc3s-1500* design to include configuration options for one additional core. Note that adding xconfig entries does not include IP core HDL files in the list of files to be synthesized for a design. See section 8.3 for information on adding the HDL files of an IP core to GRLIB.

When we start, the *config.in* file for *leon3-gr-xc3s-1500* has the following contents around the inclusion of GRGPIO:

```
mainmenu_option next_comment
  comment 'UART, timer, I/O port and interrupt controller'
    source lib/gaisler/uart/uart1.in
    if [ "$CONFIG_DSU_UART" != "y" ]; then
      source lib/gaisler/uart/uart2.in
    fi
    source lib/gaisler/leon3/irqmp.in
    source lib/gaisler/misc/gptimer.in
    source lib/gaisler/misc/grgpio.in
  endmenu
```

and the *config.vhd* file has the following entries (also just around the GRGPIO port):

```
-- GPIO port
  constant CFG_GRGPIO_ENABLE : integer := 1;
  constant CFG_GRGPIO_IMASK : integer := 16#0000#;
  constant CFG_GRGPIO_WIDTH : integer := (8);

-- Spacewire interface
....
```

The core that we will add support for is the I2C2AHB core. We start by making copies of the existing configuration files for the GRGPIO core (described in section 8.7.2) and modify them for I2C2AHB. The resulting files are listed below:

i2c2ahb.in:

```
bool 'Enable I2C to AHB bridge ' CONFIG_I2C2AHB
   if [ "$CONFIG_I2C2AHB" = "y" ]; then
     bool 'Enable APB interface   ' CONFIG_I2C2AHB_APB
     hex  'AHB protection address (high) ' CONFIG_I2C2AHB_ADDRH 0000
     hex  'AHB protection address (low) ' CONFIG_I2C2AHB_ADDRL 0000
     hex  'AHB protection mask (high) ' CONFIG_I2C2AHB_MASKH 0000
     hex  'AHB protection mask (low) ' CONFIG_I2C2AHB_MASKL 0000
```

```
    bool 'Enable after reset  ' CONFIG_I2C2AHB_APB
    hex  'I2C memory address ' CONFIG_I2C2AHB_SADDR 50
    hex  'I2C configuration address ' CONFIG_I2C2AHB_CADDR 51
  fi
```

i2c2ahb.in.help:

```
GRLIB I2C2AHB core
CONFIG_I2C2AHB
  Say Y here to enable I2C2AHB

CONFIG_I2C2AHB_APB
  Say Y here to configure the core's APB interface

CONFIG_I2C2AHB_ADDRH
  Defines address bits 31:16 of the core's AHB protection area

... and so on ..
```

i2c2ahb.in.vhd:

```
-- I2C to AHB bridge
  constant CFG_I2C2AHB        : integer := CONFIG_I2C2AHB;
  constant CFG_I2C2AHB_APB     : integer := CONFIG_I2C2AHB_APB;
  constant CFG_I2C2AHB_ADDRH   : integer := 16#CONFIG_I2C2AHB_ADDRH#;
  constant CFG_I2C2AHB_ADDRL   : integer := 16#CONFIG_I2C2AHB_ADDRL#;
  constant CFG_I2C2AHB_MASKH   : integer := 16#CONFIG_I2C2AHB_MASKH#;
  constant CFG_I2C2AHB_MASKL   : integer := 16#CONFIG_I2C2AHB_MASKL#;
  constant CFG_I2C2AHB_RESEN   : integer := CONFIG_I2C2AHB_RESEN;
  constant CFG_I2C2AHB_SADDR   : integer := 16#CONFIG_I2C2AHB_SADDR#;
  constant CFG_I2C2AHB_CADDR   : integer := 16#CONFIG_I2C2AHB_CADDR#;
  constant CFG_I2C2AHB_FILTER  : integer := CONFIG_I2C2AHB_FILTER;
```

i2c2ahb.in.h:

```
#ifndef CONFIG_I2C2AHB
#define CONFIG_I2C2AHB 0
#endif
#ifndef CONFIG_I2C2AHB_APB
#define CONFIG_I2C2AHB_APB 0
#endif
#ifndef CONFIG_I2C2AHB_ADDRH
#define CONFIG_I2C2AHB_ADDRH 0
#endif
#ifndef CONFIG_I2C2AHB_ADDRL
#define CONFIG_I2C2AHB_ADDRL 0
#endif
#ifndef CONFIG_I2C2AHB_MASKH
#define CONFIG_I2C2AHB_MASKH 0
#endif
#ifndef CONFIG_I2C2AHB_MASKL
#define CONFIG_I2C2AHB_MASKL 0
#endif
#ifndef CONFIG_I2C2AHB_RESEN
#define CONFIG_I2C2AHB_RESEN 0
#endif
#ifndef CONFIG_I2C2AHB_SADDR
#define CONFIG_I2C2AHB_SADDR 50
#endif
#ifndef CONFIG_I2C2AHB_CADDR
#define CONFIG_I2C2AHB_CADDR 51
#endif
#ifndef CONFIG_I2C2AHB_FILTER
#define CONFIG_I2C2AHB_FILTER 2
#endif
```

Once we have the above files in place, we will modify *designs/leon3-gr-emaxc3s-1500/config.in* so that I2C2AHB is also included. The resulting entries in config.in looks like:

```
mainmenu_option next_comment
  comment 'UART, timer, I/O port and interrupt controller'
    source lib/gaisler/uart/uart1.in
    if [ "$CONFIG_DSU_UART" != "y" ]; then
      source lib/gaisler/uart/uart2.in
    fi
    source lib/gaisler/leon3/irqmp.in
    source lib/gaisler/misc/gptimer.in
    source lib/gaisler/misc/grgpio.in
    source lib/gaisler/misc/i2c2ahb.in
  endmenu
```

Where the inclusion of *i2c2ahb.in* is made just before the *endmenu* statement.

We can now issue *make xconfig* in the template design directory to rebuild the graphical menu:

```
user@host:~/GRLIB/designs/leon3-gr-xc3s-1500$ make xconfig
make main.tk
make[1]: Entering directory `/home/user/GRLIB/designs/leon3-gr-xc3s-1500'
gcc -g -c ../../bin/tkconfig/tkparse.c
gcc -g -c ../../bin/tkconfig/tkcond.c
gcc -g -c ../../bin/tkconfig/tkgen.c
gcc -g tkparse.o tkcond.o tkgen.o -o tkparse.exe
./tkparse.exe config.in ../..  > main.tk
make[1]: Leaving directory `/home/user/GRLIB/designs/leon3-gr-xc3s-1500'
cat ../../bin/tkconfig/header.tk main.tk ../../bin/tkconfig/tail.tk > lconfig.tk
chmod a+x lconfig.tk
```

As can be seen from the output above, the change of *config.in* triggered a re-build of *tkparse.exe* and *lconfig.tk*. *tkparse.exe* is used to parse the *.in* files and *lconfig.tk* is what is executed when issuing *make xconfig*. In order to rebuild *tkparse.exe* the system must have a working copy of the GNU C compiler installed.

Under some circumstances the menus may not be rebuilt after *config.in* has been modified. If this happens try to issue *touch config.in* or remove the file *lconfig.tk*.

Now that the xconfig menus have been re-built we can check under *Peripherals > UART, timer, I/O port and interrupt controller* to see our newly added entries for the I2C2AHB core. Once we save and exit the xconfig tool a new *config.vhd* file will be generated that now also contains the constants defined in i2c2ahb.in.vhd:

```
-- GPIO port
  constant CFG_GRGPIO_ENABLE : integer := 1;
  constant CFG_GRGPIO_IMASK : integer := 16#0000#;
  constant CFG_GRGPIO_WIDTH : integer := (8);

-- I2C to AHB bridge
  constant CFG_I2C2AHB : integer := 0;
  constant CFG_I2C2AHB_APB : integer := 0;
  constant CFG_I2C2AHB_ADDRH : integer := 16#0#;
  constant CFG_I2C2AHB_ADDRL : integer := 16#0#;
  constant CFG_I2C2AHB_MASKH : integer := 16#0#;
  constant CFG_I2C2AHB_MASKL : integer := 16#0#;
  constant CFG_I2C2AHB_RESEN : integer := 0;
  constant CFG_I2C2AHB_SADDR : integer := 16#50#;
  constant CFG_I2C2AHB_CADDR : integer := 16#51#;
  constant CFG_I2C2AHB_FILTER : integer := 2;

-- Spacewire interface
```

These constants can now be used in all files that include the *work.config* VHDL package.

8.7.5 Other uses and limitations

There is nothing IP core specific in xconfig. Local copies of configuration files (**.in**) can be created in the template design directory to create constants that are used to control other aspects of the design and not just IP core configuration.

The graphical interface provided by xconfig can ease configuration but the tool has several limitations that designers must be aware of:

1. When configuration options are saved and xconfig is exited, the *config.vhd* file is overwritten.

2. When a core is disabled, the present configuration is not restored when the core is re-enabled.

3. The tool does not provide a good solution for multiple instances of the same core.

The last item means that xconfig can not be used to configure two separate instances of the same core (unless the cores should have the exact same configuration, if this is the case the same set of *config.vhd* constants can be used in several instantiations). It is not possible to just include the same *.in* file several times in *config.in*. This will lead to constants with the same name being created in *config.vhd*. One option is to make a local copy of a core's configuration files (**.in**) and place them in the template design directory. The local copies can then be edited to have all their variable names changed (for instance by adding a *2* to the end of the variable names) and a reference to the local files can be added to *config.in*. This way a separate set of menu items, that will affect a separate set of constants in config.vhd, can be included.

Frontgrade Gaisler AB
Kungsgatan 12
411 19 Göteborg
Sweden
www.frontgrade.com/gaisler
sales@gaisler.com
T: +46 31 7758650

Frontgrade Gaisler AB, reserves the right to make changes to any products and services described herein at any time without notice. Consult the company or an authorized sales representative to verify that the information in this document is current before using this product. The company does not assume any responsibility or liability arising out of the application or use of any product or service described herein, except as expressly agreed to in writing by the company; nor does the purchase, lease, or use of a product or service from the company convey a license under any patent rights, copyrights, trademark rights, or any other of the intellectual rights of the company or of third parties. All information is provided as is. There is no warranty that it is correct or suitable for any purpose, neither implicit nor explicit.

Copyright © 2025 Frontgrade Gaisler AB