GRLIB IP Library User’s Manual

Version 1.0.22

Jiri Gaisler, Sandi Habinc

Copyright Aeroflex Gaisler, 2010

Table of contents

1 Introduction

1.1 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.

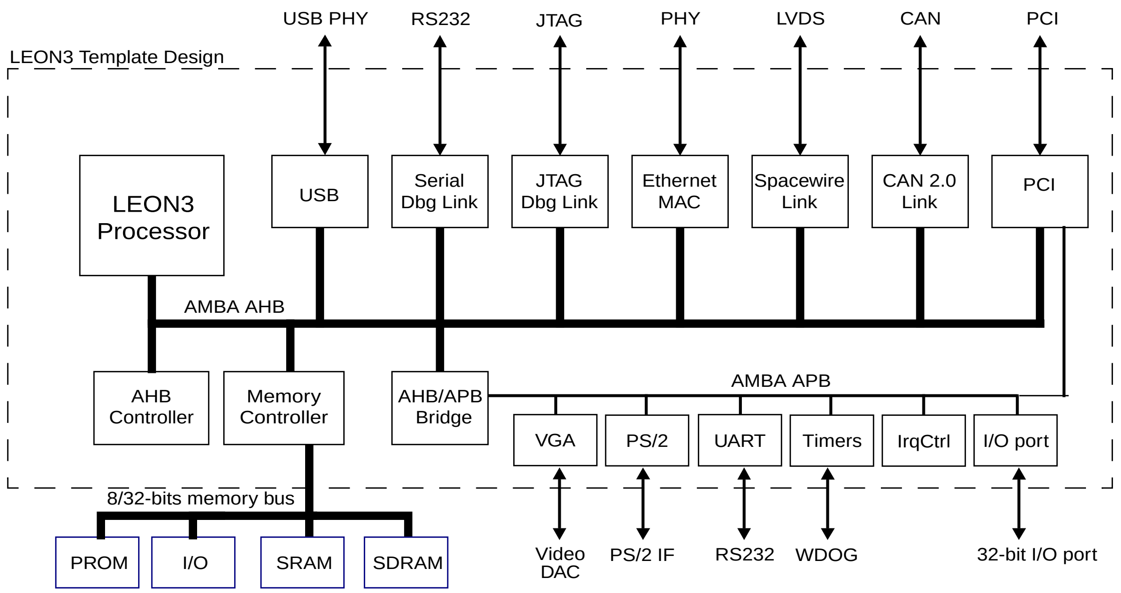
1.2 Library organization

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, and Xilinx implementation tools. Support for other CAD tools can be easily be added.

1.3 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:



1.4 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.

1.5 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.

1.6 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.

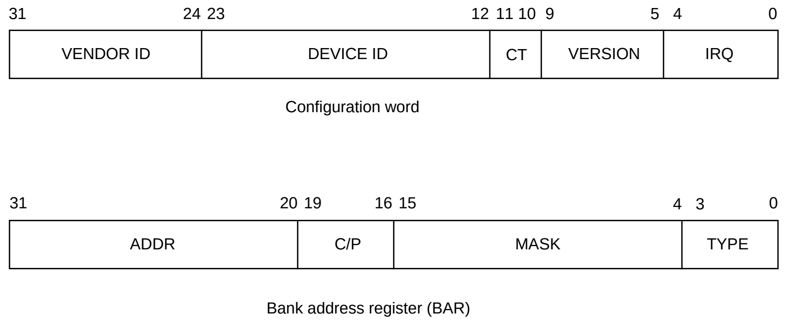


Figure 1. AMBA configuration record

The configuration word for each device includes a vendor ID, device ID, version number, and interrupt routing information. A configuration type indicator is provided to allow for future evolvement of the configuration word. 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.

1.7 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.

1.8 Available IP cores

The library includes cores for AMBA AHB/APB control, the LEON3 SPARC processor, IEEE-754 floating-point unit, AHB/ABH bridge, 32-bit PC133 SDRAM controller, DDR1/2 controllers,, 32-bit PCI bridge with DMA, 10/100/1000 Mbit ethernet MAC, CAN-2.0 controller, USB-2.0 host and device controllers, 8/16/32-bit PROM/SRAM controller, 32-bit SSRAM controller, 32-bit GPIO port, timer unit, interrupt controller, PS/2 interface, VGA controller and many other legacy cores. Memory generators are available for Actel, Altera, Atmel, Eclipse, Lattice, UMC, Artisan, Virage and Xilinx.

1.9 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, Gaisler Research offers low-cost commercial IP licenses. Contact sales@gaisler.com for more information or visit http://www.gaisler.com/ .

3 Installation

3.1 Installation

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

```
gunzip -c grlib-gpl-1.0.22-bxxxx.tar.gz | tar xf -

or

unzip grlib-gpl-1.0.22-bxxxx.zip
```

NOTE: Do NOT use unzip 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.

3.2 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-1.0.x/lib are:

`grlib`	`packages with common data types and functions`
`gaisler`	`Gaisler Research’s components and utilities`
`tech/*`	`target technology libraries for gate level simulation`
`techmap`	`wrappers for technology mapping of marco 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.

3.3 Host platform support

GRLIB is design to work with a large variety of hosts. The paragraphs below outline the hosts tested by Gaisler Research. Other unix-based hosts 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

3.3.1 Linux

The make utility and associated scripts will work on any recent linux distribution. GRLIB is primarily developed on linux hosts, and linux is the preferred platform.

3.3.2 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. Warning: some versions of Cygwin are know 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.

2 LEON3 quick-start guide

2.1 Introduction

This chapter will provide a simple quick-start guide on how to implement a leon3 system using GRLIB, and how to download and run software on the target system. Refer to chapters 4 - 6 for a deeper understanding of the GRLIB organization.

2.2 Overview

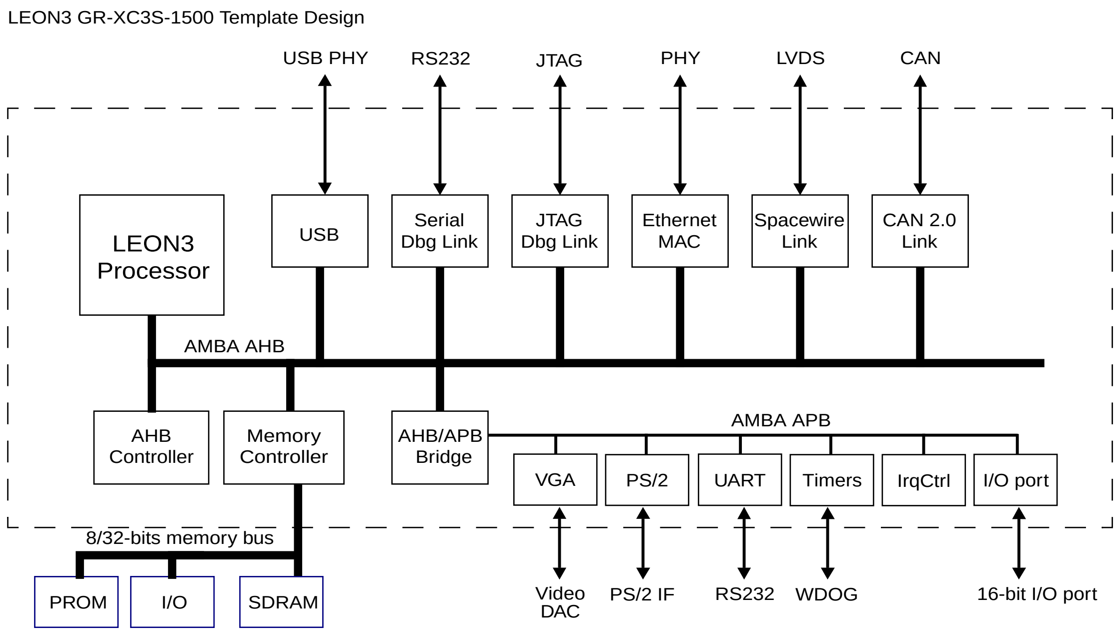
Implementing a leon3 system is typically done using one of the template designs on the designs directory. For this tutorial, we will use the LEON3 template design for the GR-XC3S-1500 board. Implementation is typically done in three basic steps:

- 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.



2.3 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.

2.4 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:

```
make vsim
vsim testbench
```

A typical simulation log can be seen below.

```
$ vsim testbench

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: Gaisler Research        Leon3 SPARC V8 Processor
# ahbctrl: mst1: Gaisler Research        JTAG Debug Link
# ahbctrl: mst2: Gaisler Research        SpaceWire Serial Link
# ahbctrl: mst3: Gaisler Research        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: Gaisler Research        AHB/APB Bridge
# ahbctrl:       memory at 0x80000000, size 1 Mbyte
# ahbctrl: slv2: Gaisler Research        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: Gaisler Research        Generic UART
# apbctrl:       I/O ports at 0x80000100, size 256 byte
# apbctrl: slv2: Gaisler Research        Multi-processor Interrupt Ctrl.
# apbctrl:       I/O ports at 0x80000200, size 256 byte
# apbctrl: slv3: Gaisler Research        Modular Timer Unit
# apbctrl:       I/O ports at 0x80000300, size 256 byte
# apbctrl: slv8: Gaisler Research        General Purpose I/O port
# apbctrl:       I/O ports at 0x80000800, size 256 byte
# apbctrl: slv12: Gaisler Research        SpaceWire Serial Link
# apbctrl:       I/O ports at 0x80000c00, size 256 byte
# apbctrl: slv13: Gaisler Research        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 tool-chain is installed on the host computer.

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.

2.5 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.

2.6 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
```

2.7 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
```

2.8 Running applications on target

To download and debug applications on the target board, GRMON debug monitor is used. GRMON can be connected to the target using RS232, JTAG, ethernet or USB. The most convenient way is probably to use JTAG. GRMON can the Xilinx parallel port cable programming cable and or the Platfrom USB cable. See the GRMON manual for details. To connect using the parallel port cable, do:

```
grmon -jtag -u
```

This should print the configuration of the target board:

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

 Component                            Vendor
 LEON3 SPARC V8 Processor             Gaisler Research
 AHB Debug UART                       Gaisler Research
 AHB Debug JTAG TAP                   Gaisler Research
 SVGA frame buffer                    Gaisler Research
 GR Ethernet MAC                      Gaisler Research
 AHB ROM                              Gaisler Research
 AHB/APB Bridge                       Gaisler Research
 LEON3 Debug Support Unit             Gaisler Research
 DDR266 Controller                    Gaisler Research
 Generic APB UART                     Gaisler Research
 Multi-processor Interrupt Ctrl       Gaisler Research
 Modular Timer Unit                   Gaisler Research
 Keyboard PS/2 interface              Gaisler Research
 Keyboard PS/2 interface              Gaisler Research
```

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.

2.9 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 sparc-elf-mkprom utility provided with the BCC tool chain. The suitable mkprom parameters for the GR-XC3S-1500 board are:

```
sparc-elf-mkprom -romws 4 -freq 40 -col 9 -nosram -sdram 64 -msoft-float -baud 38400
```

Note that the -freq option should reflect the selected processor frequency, which depends on the clock generator settings. If the processor includes an FPU, the -msoft-float switch can be omitted.

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
```

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 is designed to reside in a global location and to be used in read-only mode. 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

 interactive targets:

 make avhdl-launch         : start active-hdl gui mode
 make riviera-launch       : start riviera
 make vsim-launch          : start modelsim
 make ncsim-launch         : compile design using ncsim
 make sonata-launch        : compile design using sonata
 make actel-launch-synp    : 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 quartus-launch       : start Quartus for current project
 make quartus-launch-synp  : start Quartus for synplify project
 make synplify-launch      : start synplify
 make xgrlib               : start grlib GUI

 batch targets:

 make avhdl        : compile design using active-hdl gui mode
 make vsimsa       : compile design using active-hdl batch mode
 make riviera      : compile design using riviera
 make sonata       : compile design using sonata
 make vsim         : compile design using modelsim
 make ncsim        : compile design using ncsim
 make ghdl         : compile design using GHDL
 make actel        : synthesize with synplify, place&route Actel Designer
 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 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 precision    : synthesize design using precision
 make synplify     : synthesize design using synplify
 make scripts      : generate compile scripts only
 make clean        : remove all temporary files except scripts
 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 designs)
clean	Remove all temporary files except scripts and project files
distclean	Remove all temporary files
xgrlib	Run the graphical implementation tool (see page 29)

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 “XGrlib graphical implementation tool” on page 38.

4.3 Simulating a design

The ‘make scripts’ command will generate compile scripts and/or project files for the modelsim, ncsim and ghdl simulators. This is done by scanning GRLIB for simulation files according to the method described in “GRLIB organisation” on page 75. 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 simulator are currently supported by GRLIB:

TABLE 2. Supported simulators

Simulator	Comments
GNU VHDL (GHDL)	version 0.25, VHDL only
Aldec Active-HDL	batch and GUI
Aldec Riviera	
Mentor Modelsim version	version 6.1e or later
Cadence NcSim	IUS-5.8-sp3 and later
Synphony-EDA Sonata	verison 3.1 or later, VHDL only

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

The following synthesis tools are currently supported by GRLIB:

TABLE 3. Supported synthesis and place&route tools

Syntesis and place&route tool	Recommended version
Actel Designer/Libero	version 8.6
Altera Quartus	version 6.0 and later
Cadence RTLC	version 6.1 and later
Lattice ispLEVER	version 5.1
Mentor Leonardo Precision	2009a.138
Synopsys DC	2007.3
Synplify	version 8.9 and later
Xilinx ISE/XST	version 10.3, 11.4

4.5 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, 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 leon3-digilent-xc3s1000 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 openchip micron hynix cypress ihp gleichmann 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 4. TECHLIB settings for various target technologies

Technology	TECHLIBS defines
Xilinx (All)	unisim simprim
Altera Stratix-II	altera altera_mf stratixii
Altera Cyclone-III	altera altera_mf cycloneiii
Altera others	altera altera_mf
Actel Axcelerator	axcelerator
Actel Axcelerator DSP	axcelerator
Actel Proasic3	proasic3
Lattice	ec
Quicklogic	eclipsee
Atmel ATC18	atc18 virage
Atmel ATC18RHA	atc18rha_cell
eASIC 90 nm	nextreme
IHP 0.25	ihp25
IHP 0.25 RH	sgb25vrh
Aeroflex 0.25 RH	ut025crh

TABLE 4. TECHLIB settings for various target technologies

Technology	TECHLIBS defines
Ramon 0.18 RH	rh_lib18t
UMC 0.18 um	umc18
TSMC 90 nm	tsmc90

Note that some technologies are not availble in the GPL version. Contact Gaisler Research for details.

4.6 Tool-specific usage

4.6.1 GNU VHDL (GHDL)

GHDL is the GNU VHDL compiler/simulator, available from http://ghdl.free.fr/. It is used mainly on linux hosts, although a port to windows/cygwin has recently been reported.

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 5. GHDL make targets

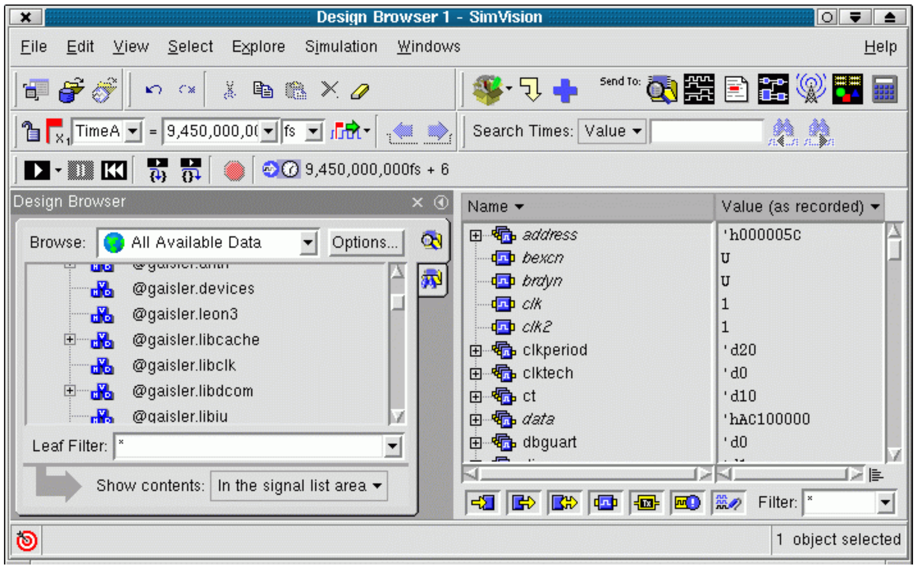
Make target	Description
ghdl	Compile or re-analyze local design
ghdl-clean	Remove compiled models and temporary files
ghdl-run	Run test bench in batchmode

TABLE 6. 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

4.6.2 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`**.



Name	Value (as recorded)
address	'h000005C
bexcn	U
brdyn	U
clk	1
clk2	1
clkperiod	'd20
clktech	'd0
ct	'd10
data	'hAC100000
dbguart	'd0

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 7. 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

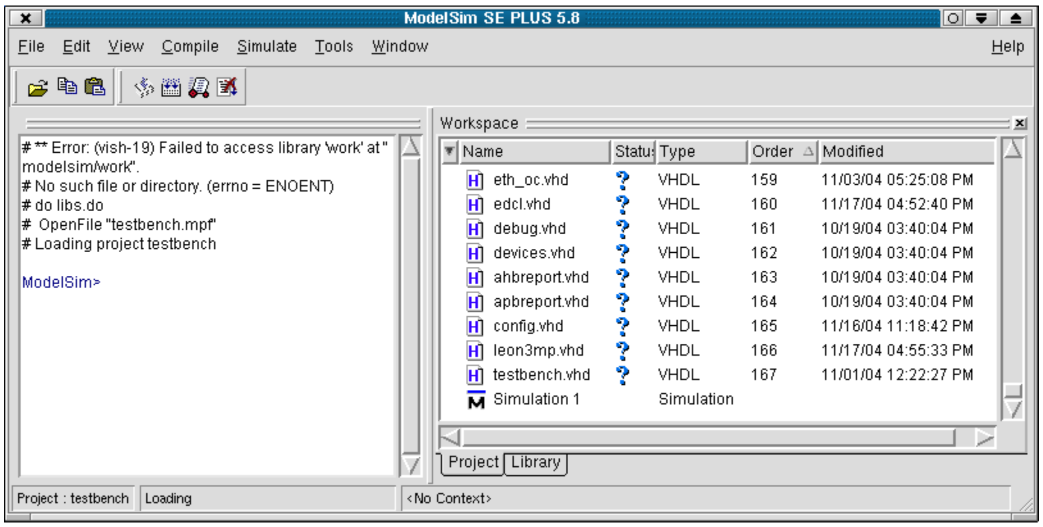
TABLE 8. Ncsim scripts and files

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

4.6.3 Mentor 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..

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).



```
# ** Error: (vish-19) Failed to access library 'work' at "
modelsim/work".
# No such file or directory. (errno = ENOENT)
# do libs.do
#  OpenFile "testbench.mpf"
# Loading project testbench

ModelSim>
```

Name	Status	Type	Order	Modified
eth_oc.vhd	?	VHDL	159	11/03/04 05:25:08 PM
edcl.vhd	?	VHDL	160	11/17/04 04:52:40 PM
debug.vhd	?	VHDL	161	10/19/04 03:40:04 PM
devices.vhd	?	VHDL	162	10/19/04 03:40:04 PM
ahbreport.vhd	?	VHDL	163	10/19/04 03:40:04 PM
apbreport.vhd	?	VHDL	164	10/19/04 03:40:04 PM
config.vhd	?	VHDL	165	11/16/04 11:18:42 PM
leon3mp.vhd	?	VHDL	166	11/17/04 04:55:33 PM
testbench.vhd	?	VHDL	167	11/01/04 12:22:27 PM
Simulation 1		Simulation		

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

TABLE 9. Modelsim make targets

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

TABLE 10. 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

4.6.4 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.

TABLE 11. 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 12. Active-HDL scripts and files

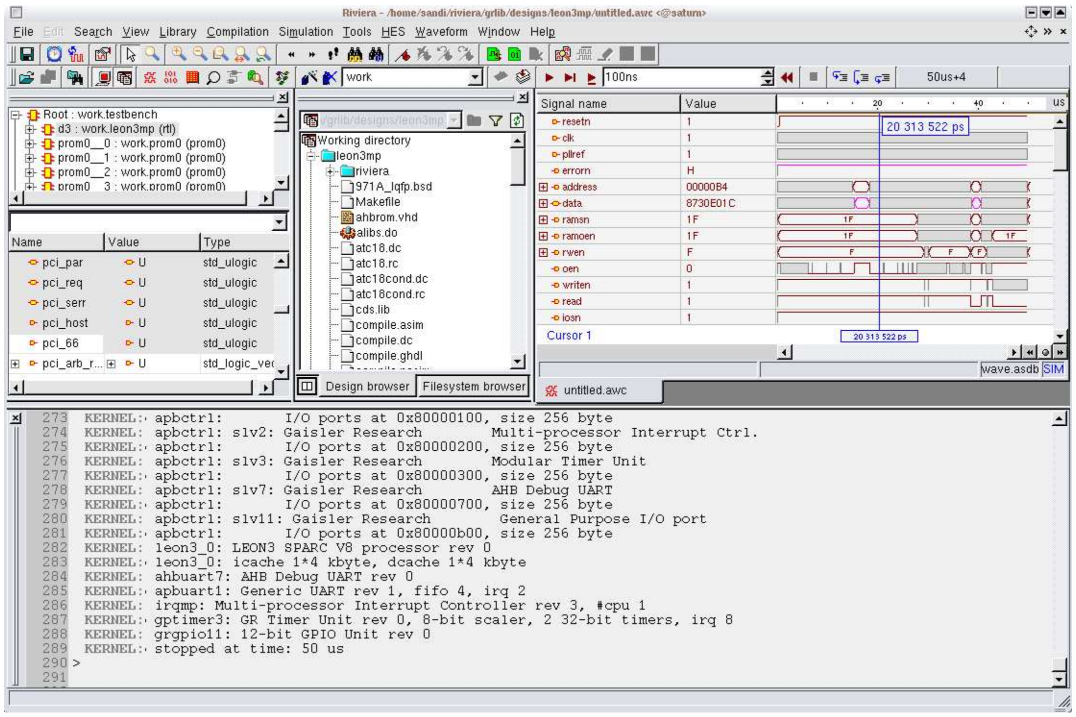
File	Description
compile.asim	Compile script for GRLIB files (batch mode)
make.asim	Compile script for GRLIB files and local design (batch mode)
activehdl	Directory with compiled models (batch mode)
work	Directory with compiled models (GUI mode)
avhdl.tcl	Active-HDL *tcl* file for compilation and simulation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

4.6.5 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 completed GRLIB as well as the local design are compiled by **make riviera**. The compiled simulation models will be stored locally in a sub-directory (./riviera). A vsimsa.cfg file will be created automatically, containing the necessary VHDL library mapping definitions.

The standalone batch mode simulation can be started with **make riviera-run**. The GUI mode simulation can be started with **make riviera-launch**.



```
KERNEL: apbctrl:        I/O ports at 0x80000100, size 256 byte
KERNEL: apbctrl: slv2: Gaisler Research        Multi-processor Interrupt Ctrl.
KERNEL: apbctrl:        I/O ports at 0x80000200, size 256 byte
KERNEL: apbctrl: slv3: Gaisler Research        Modular Timer Unit
KERNEL: apbctrl:        I/O ports at 0x80000300, size 256 byte
KERNEL: apbctrl: slv7: Gaisler Research        AHB Debug UART
KERNEL: apbctrl:        I/O ports at 0x80000700, size 256 byte
KERNEL: apbctrl: slv11: Gaisler Research        General Purpose I/O port
KERNEL: apbctrl:        I/O ports at 0x80000b00, size 256 byte
KERNEL: leon3_0: LEON3 SPARC V8 processor rev 0
KERNEL: leon3_0: icache 1*4 kbyte, dcache 1*4 kbyte
KERNEL: ahbuart7: AHB Debug UART rev 0
KERNEL: apbuart1: Generic UART rev 1, fifo 4, irq 2
KERNEL: irqmp: Multi-processor Interrupt Controller rev 3, #cpu 1
KERNEL: gptimer3: GR Timer Unit rev 0, 8-bit scaler, 2 32-bit timers, irq 8
KERNEL: grgpio11: 12-bit GPIO Unit rev 0
KERNEL: stopped at time: 50 us
>
```

TABLE 13. Riviera make targets

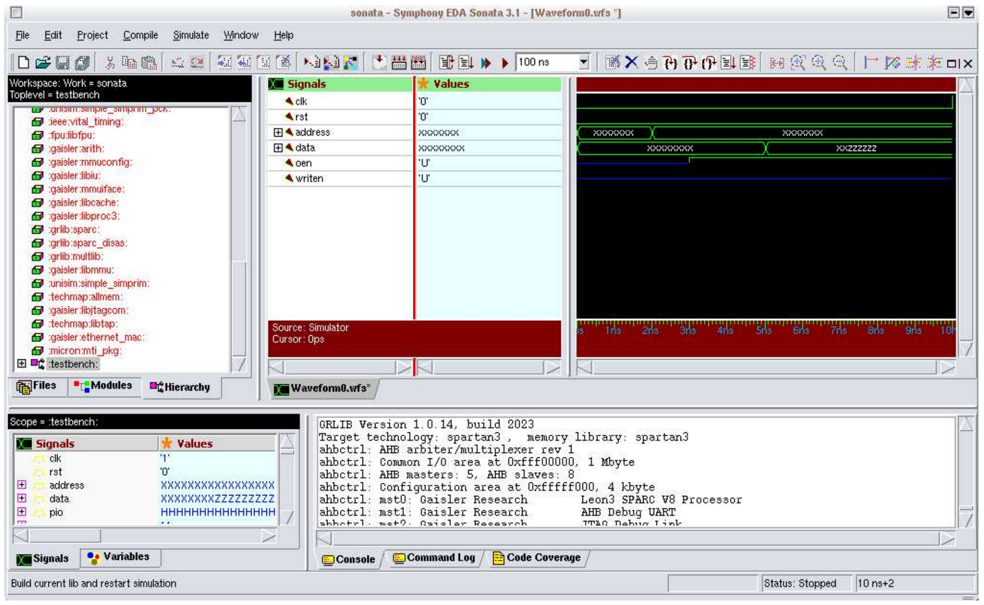
Make target	Description
riviera	Compile GRLIB and local design
riviera-clean	Remove compiled models and temporary files
riviera-run	Run test bench in batch mode (must be compiled first)
riviera-launch	Run test bench in GUI mode (must be compiled first)

TABLE 14. Riviera scripts and files

File	Description
riviera	Directory with compiled models
riviera.do	Rivera script file for simulation (GUI mode)

4.6.6 Symphony-EDA Sonata

The complete GRLIB as well as the local design are compiled by **make sonata**. The compiled simulation models will be stored locally in a sub-directory (./sonata). A symphony.ini file will be created automatically, containing the necessary VHDL library mapping definitions. To run the Sonata simulator in GUI mode, do **make sonata-launch** or start Sonata using the crated sonata.sws project file. Sonata can also be run in batch mode, with **make sonata-run**. The VHDL work library will be mapped on library ‘sonata’, as ‘work’ is reserved and cannot be used.



```
GRLIB Version 1.0.14, build 2023
Target technology: spartan3 ,  memory library: spartan3
ahbctrl: AHB arbiter/multiplexer rev 1
ahbctrl: Common I/O area at 0xfff00000, 1 Mbyte
ahbctrl: AHB masters: 5, AHB slaves: 8
ahbctrl: Configuration area at 0xfffff000, 4 kbyte
ahbctrl: mst0: Gaisler Research        Leon3 SPARC V8 Processor
ahbctrl: mst1: Gaisler Research        AHB Debug UART
ahbctrl: mst2: Gaisler Research        JTAG Debug Link
```

TABLE 15. Sonata make targets

Make target	Description
sonata	Compile GRLIB and local design
sonata-clean	Remove compiled models and temporary files
sonata-run	Compile GRLIB and run test bench in batch mode
sonata-launch	Compiler GRLIb and run test bench in GUI mode

TABLE 16. Riviera scripts and files

File	Description
sonata	Directory with compiled models
symphony.ini	Sonata library mapping for batch simulation
sonata.sws	Sonata project file for GUI version

4.6.7 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
```

Additional options can be set through the SYNPOPT variable in the Makefile:

```
SYNPOPT="set_option -pipe 0; set_option -retiming 1”
```

Note that the Synplify tool does have some bugs, which can cause the generation of corrupt netlist for large designs. Currently, the most stable version seems to be 8.9.

TABLE 17. 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 18. Synplify scripts and files

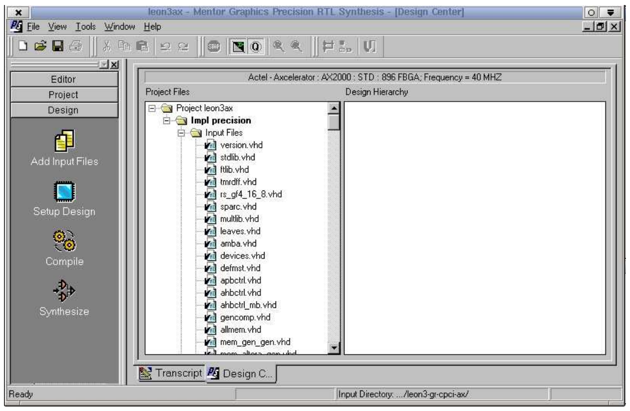
File	Description
compile.synp	Tcl compile script for all GRLIB files
TOP_synplify.prj	Synplify project file
synplify	Directory with netlist and log files

4.6.8 Synthesis with Mentor Precision

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
```



Actel - Axcelerator : AX2000 : STD : 896 FBGA; Frequency = 40 MHZ

```
Project leon3ax
  Impl precision
    Input Files
      version.vhd
      stdlib.vhd
      ftlib.vhd
      tmrdff.vhd
      rs_gf4_16_8.vhd
      sparc.vhd
      multlib.vhd
      leaves.vhd
      amba.vhd
      devices.vhd
      defmst.vhd
      apbctrl.vhd
      ahbctrl.vhd
      ahbctrl_mb.vhd
      gencomp.vhd
      allmem.vhd
      mem_gen_gen.vhd
```

Ready   Input Directory: .../leon3-gr-cpci-ax/

TABLE 19. 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 20. 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

4.6.9 Actel 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.

GRLIB also includes a leon3 template design for the Actel CoreMP7 board (Proasic3-1000). It is located in designs/leon3-actel-coremp7.

TABLE 21. 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

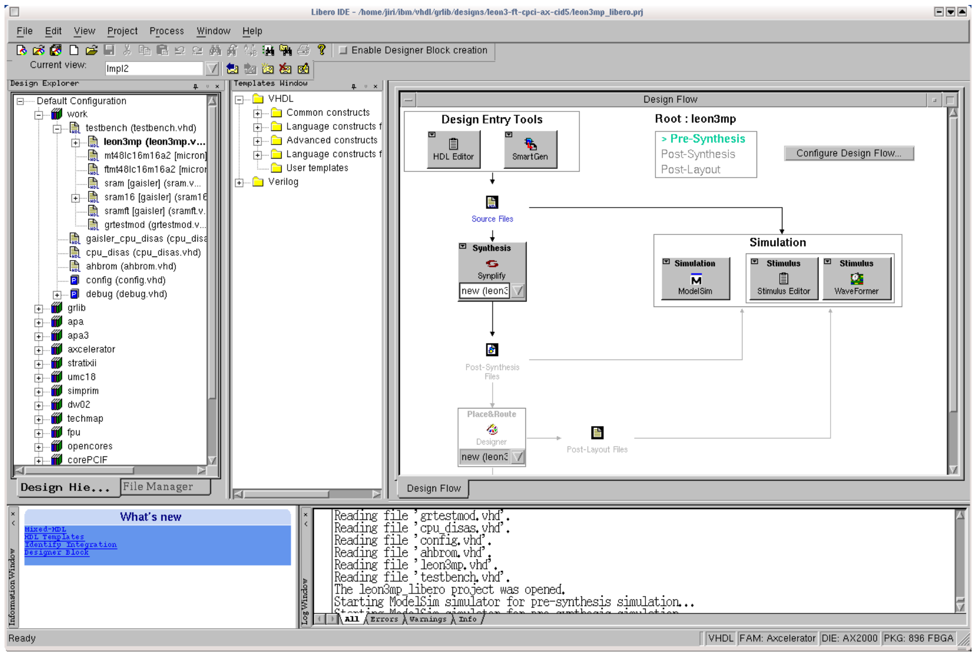
TABLE 22. Actel Designer scripts and files

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

4.6.10 Actel Libero

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

Using Libero to implement GRLIB designs is possible using Libero-8.1 and later versions. The **`make scripts`** command will create a Libero project file called TOP_libero.prj. Libero can then be started with **`libero TOP_libero.prj`**, or by the command **`make libero-launch`**. Implementation of the design is done using the normal Libero flow.



```
Reading file 'grtestmod.vhd'.
Reading file 'cpu_disas.vhd'.
Reading file 'config.vhd'.
Reading file 'ahbrom.vhd'.
Reading file 'leon3mp.vhd'.
Reading file 'testbench.vhd'.
The leon3mp_libero project was opened.
Starting ModelSim simulator for pre-synthesis simulation...
```

Note that when synplify is launched from Libero the first time, the constraints file defined in the local Makefile is 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 stimulify file.

TABLE 23. Libero make targets

Make target	Description
scripts	Created libero project file
libero-launch	Create project file and launch libero

TABLE 24. Libero scripts and files

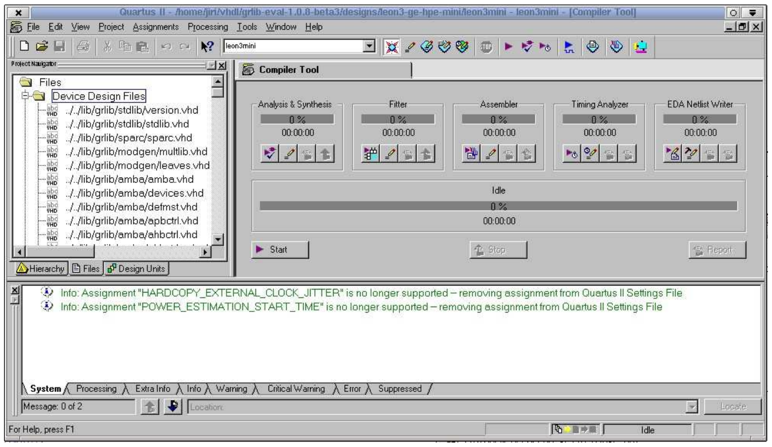
File	Description
TOP_libero.prj	Libero project file

4.6.11 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`**.



```
../../lib/grlib/stdlib/version.vhd
../../lib/grlib/stdlib/stdlib.vhd
../../lib/grlib/sparc/sparc.vhd
../../lib/grlib/modgen/multlib.vhd
../../lib/grlib/modgen/leaves.vhd
../../lib/grlib/amba/amba.vhd
../../lib/grlib/amba/devices.vhd
../../lib/grlib/amba/defmst.vhd
../../lib/grlib/amba/apbctrl.vhd
../../lib/grlib/amba/ahbctrl.vhd
```

```
Info: Assignment "HARDCOPY_EXTERNAL_CLOCK_JITTER" is no longer supported -- removing assignment from Quartus II Settings File
Info: Assignment "POWER_ESTIMATION_START_TIME" is no longer supported -- removing assignment from Quartus II Settings File
```

GRLIB includes a leon3 template design for the HPE-Mini board from Gleichmann Electronics. The template design is located designs/leon3-ge-hpe-mini. and uses board settings from the boards/ge-hpe-mini directory. The leon3-ge-hpe-mini design can favorably be used a template for other Altera-based projects. It is essential that Quartus version 6.0 or later is used.

TABLE 25. 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 26. Altera Quartus scripts and files

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

4.6.12 Xilinx ISE

Xilinx ISE is used for Xilinx 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 the place&route with ISE.

The **make scripts** command will create a compile.xst file which contains XST commands for analyzing all GRLIB files. The compile.xst can be run manually using xst -**ifn** compile.xst or through **make ise-map**. An XST script to analyze and synthesize the local design will be created automatically, and called TOP.xst. To synthesize and place&route in one go, use **make ise**.

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.np**l 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**. The Xilinx xtclsh operate very slowly, so generation of the .ise file will take 10 - 20 minutes (!).

For ISE-11, an XML project file is created (TOP.xise). The ISE-11 project navigator can be started using the .xise file directly, which is much faster than generating a corresponding .ise file with xtclsh. When executing **make ise-launch**, the version of the ISE installation will be automatically detected and the project manager will be launched with the appropriate project file.

It is generally recommended to use the latest version of ISE (11.4 at the time of writing). The XST option '-fsm_extract no' should be used to avoid possible netlist corruption by the FSM compiler.

Several 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 uninstall-unisim**. Note: to install the Xilinx UNISIM files, the variable XILINX must point to the installation path of ISE. The variable is normally set automatically during installation of ISE. To compile the Xilinx UNISIM libraries with modelsim, the switch -explicit must be given to vcom. This is done by setting the variable **VCOMOPT=-explicit** in the local Makefile.

TABLE 27. 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-clean	Remove compiled models and temporary files
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	Re-program FPGA on target board using JTAG
ise-prog-fpga-ref	Re-program FPGA on target board with reference bit file

TABLE 27. Xilinx ISE make targets

Make target	Description
ise-prog-prom	Re-program configuartion proms on target board using JTAG
ise-prog-prom-ref	Re-program configuartion proms with reference bit file
install-unisim	Install Xilinx UNISIM libraries into grlib
uninstall-unisim	Remove Xilinx UNISIM libraries from grlib

TABLE 28. 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 XML project file for XST flow
TOP_synplify.npl	ISE 8project file for EDIF flow

4.6.13 Lattice ISP Tools

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. The template design leon3-hpe-mini-lattice is a Lattice ECP design which can beused to test the implementation flow. 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 29. 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

4.6.14 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' or through **make dc**. A script to analyze and synthesize the local design is created automatically and called TOP.dc where TOP is the top entity name:

```
$ cat netcard.dc
sh mkdir synopsys
hdlin_ff_always_sync_set_reset = true
hdlin_translate_off_skip_text = true
include compile.dc
analyze -f VHDL -library work netcard.vhd
elaborate netcard
write -f db -hier  netcard -output netcard.db
quit
```

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

Synopsys DC is also not without bugs, and the usage of version 2003.06.SP1 or 2006.06.SP3 is strongly recommended.

The **make scripts** command will also create a top-level synthesis script for dc_shell-xg-t. The file will be called TOP_dc.tcl. It is recommended to use the dc_shell-xg-t shell and ddc file format, arther then the older db format. This allows a single-pass top-down synthesis of large designs without running out of memory.

4.6.15 Synthesis with Cadence RTL Compiler

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.

4.6.16 eASIC eTools

Implementing a GRLIB design on eASIC's nextreme technology is supported with the eASIC eTools. The **make etools-init** command will create and initialize a directory called cdb in the current working directory. The **make etools-launch** starts the eTools eX-checker, which reads in the design and extracts information about PADs and PLLs. After make etools-launch have been executed the **make etools-wizard** command can be used to start the eTools eWizard for placement of PADs and PLLs.

The make targets require that a number of parameters (variables) is set in the makefile. Consult table 31 for information about these parameters.

Make sure that the architecture of the top entity has a unique name, else the tools will not be able to identify the top design correctly.

TABLE 30. eASIC eTools make targets

Make target	Description
etools-init	Creates a cdb directory and initializes its contents.
etools-launch	Calls eTools eX checker, which extract PAD and PLL information from the design.
etools-wizard	Launches the eTools eWizard that is used for PAD and PLL placement.

TABLE 31. Makefile parameters

Parameter	Description
ETOOLS_TOP_HDL	Set to vhdl if the top HDL file is written in VHDL else set it to verilog
ETOOLS_PNC	The PNC is provided by eASIC
ETOOLS_DEVICE	The device to be targeted
ETOOLS_PACKAGE	The package to be used for the device
ETOOLS_HOME	Path to the eTools installation directory
MAGMA_HOME	Path to Magma tools (Set to . if Magma is not available)

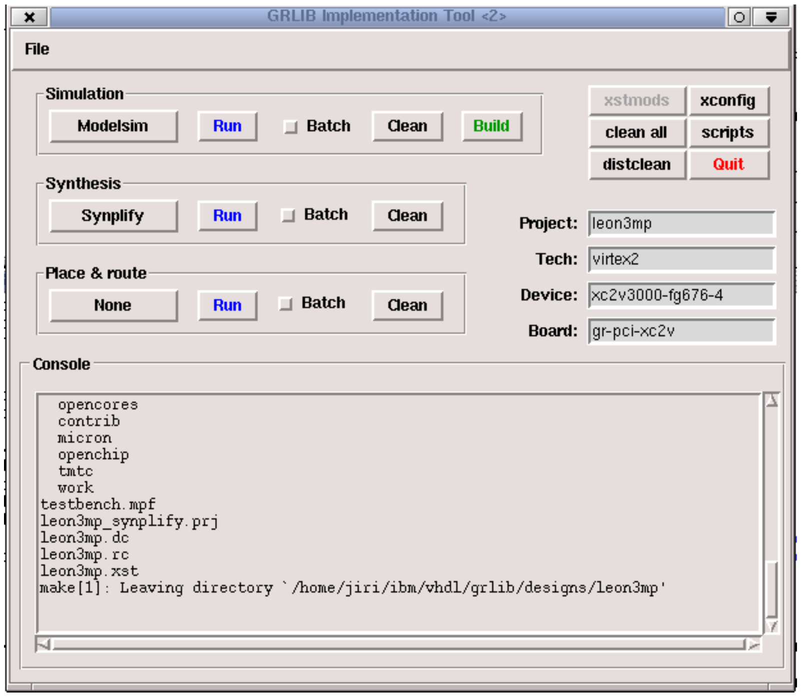
4.7 XGrlib graphical implementation tool

4.7.1 Introduction

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:



Project: leon3mp
Tech: virtex2
Device: xc2v3000-fg676-4
Board: gr-pci-xc2v

```
  opencores
  contrib
  micron
  openchip
  tmtc
  work
testbench.mpf
leon3mp_synplify.prj
leon3mp.dc
leon3mp.rc
leon3mp.xst
make[1]: Leaving directory `/home/jiri/ibm/vhdl/grlib/designs/leon3mp'
```

Figure 4. **XGrlib main window**

4.7.2 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. We are trying to resolve this issue.

4.7.3 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 Actel 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 tool actually performs both simulation, synthesis and place&route. I has been added to the ‘Synthesis’ menu for convenience.

4.7.4 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.

4.7.5 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 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.

The following sections will describe how the AMBA buses are implemented and how to develop a SOC design using GRLIB.

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.

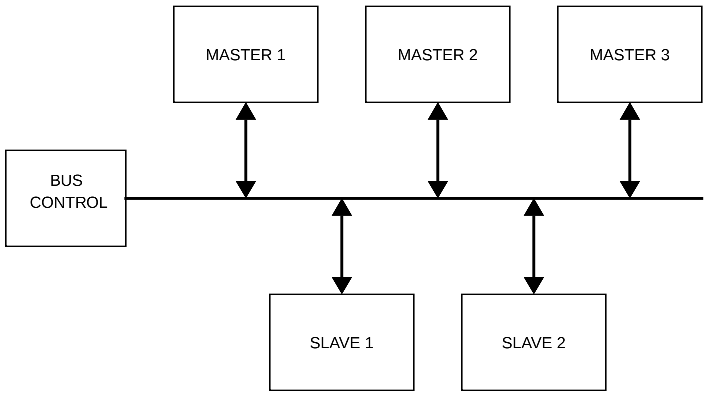


Figure 5. AMBA AHB conceptual view

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 HMSTO. 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.

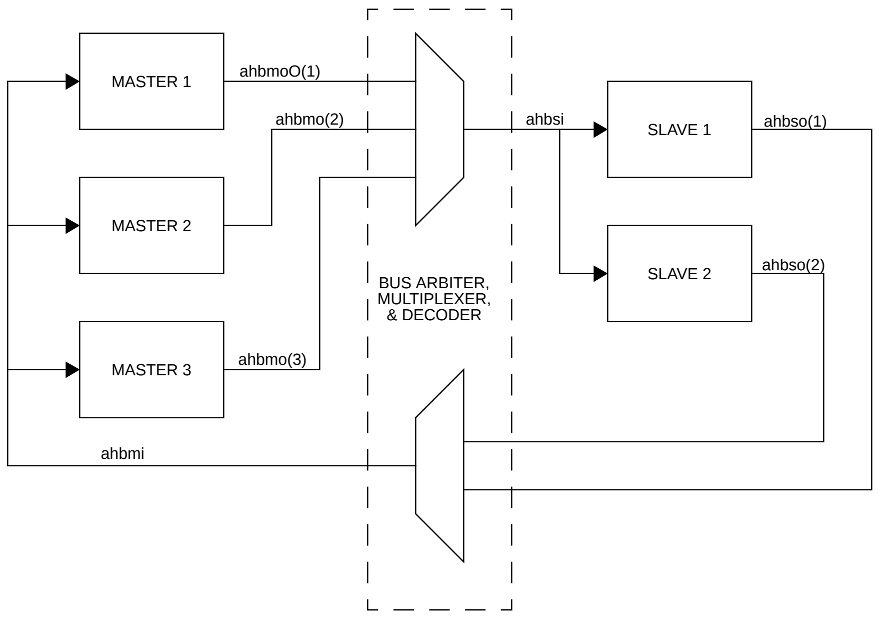


Figure 6. AHB inter-connection view

5.2.2 AHB master interface

The AHB master inputs and outputs are defined as VHDL record types, and are exported through the TYPES package in the GRLIB AMBA 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
    hcache  : std_ulogic;                             -- cacheable
    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 four sideband signals: HCACHE, 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;
    hmsti   : in  ahb_mst_in_type;     -- AHB master inputs
    hmsto   : out ahb_mst_out_type     -- AHB master outputs
  );
end entity;
```

The input record (HMSTI) is routed to all masters, and includes the bus grant signals for all masters in the vector HMSTI.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
    hcache    : std_ulogic;                           -- cacheable
    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
    hcache     : std_ulogic;                           -- cacheable
    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 five sideband signals: HBSEL, HCACHE, 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;
    hslvi   : in  ahb_slv_in_type;     -- AHB slave inputs
    hslvo   : 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.

5.3 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.

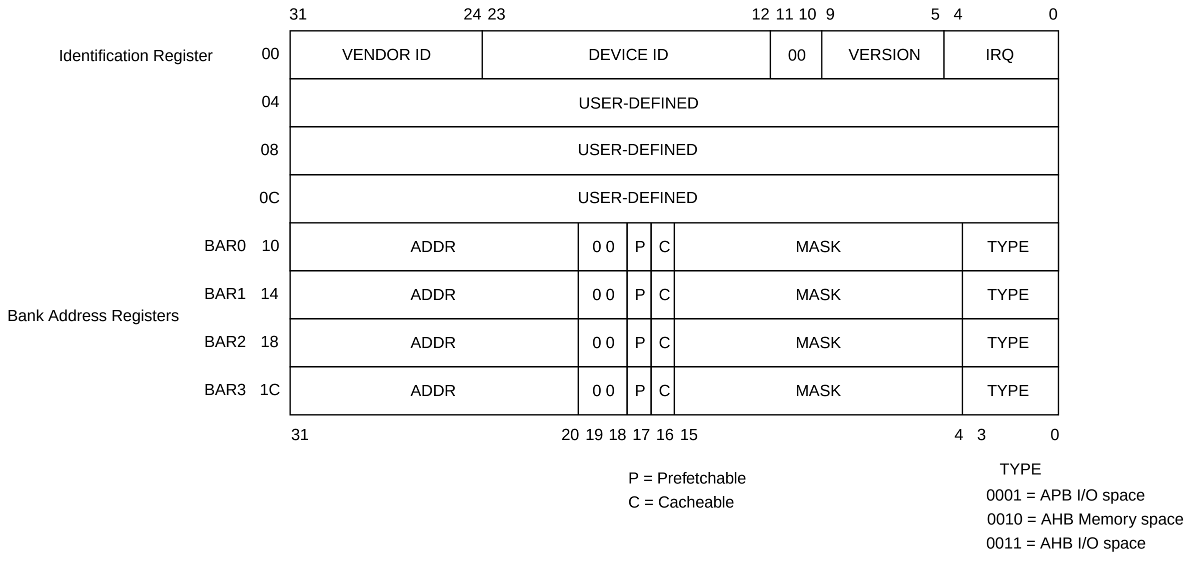


Figure 7. AHB plug&play configuration layout

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 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 (ahbreport) in the WORK.DEBUG package 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: Gaisler Research        Leon3 SPARC V8 Processor
# ahbctrl: mst1: Gaisler Research        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: Gaisler Research        AHB/APB Bridge
# ahbctrl:       memory at 0x80000000, size 1 Mbyte
# ahbctrl: slv2: Gaisler Research        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: Gaisler Research        Generic UART
# apbctrl:       I/O ports at 0x80000100, size 256 byte
# apbctrl: slv2: Gaisler Research        Multi-processor Interrupt Ctrl.
# apbctrl:       I/O ports at 0x80000200, size 256 byte
# apbctrl: slv3: Gaisler Research        Modular Timer Unit
# apbctrl:       I/O ports at 0x80000300, size 256 byte
# apbctrl: slv7: Gaisler Research        AHB Debug UART
# apbctrl:       I/O ports at 0x80000700, size 256 byte
# apbctrl: slv11: Gaisler Research        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 in each vendor library, usually called DEVICES. Vendor IDs are provided by Gaisler Research. The following ID's are currently assigned:

Vendor	ID
Gaisler Research	0x01
Pender Electronic Design	0x02
European Space Agency	0x04
Astrium EADS	0x06
OpenChip.org	0x07
OpenCores.org	0x08
Eonic BV	0x0B
Radionor	0x0F
Gleichmann Electronics	0x10
Menta	0x11
Sun Microsystems	0x13
Movidia	0x14
Orbita	0x17
Siemens AG	0x1A
Actel Corporation	0xAC
Caltech	0xCA
Embeddit	0xEA

TABLE 32. 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.

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;
    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#)
  port map (rst, clk, hslvi, hslvo(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 LEON3 processor builds 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.

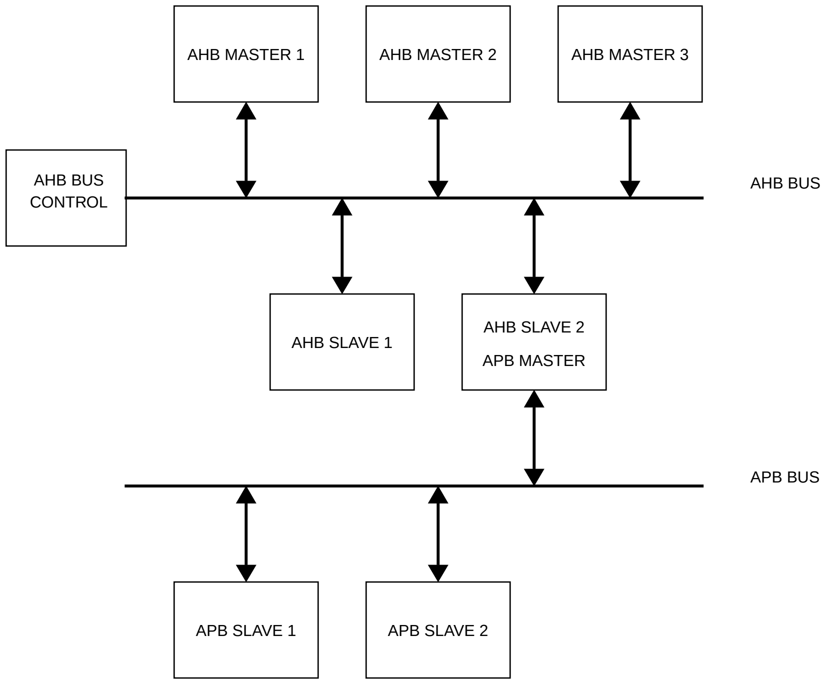


Figure 8. AMBA AHB/APB conceptual view

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).

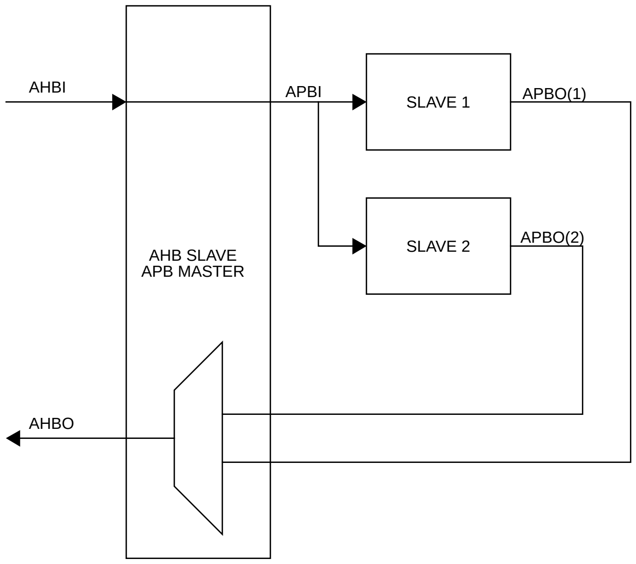


Figure 9. APB inter-connection view

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.

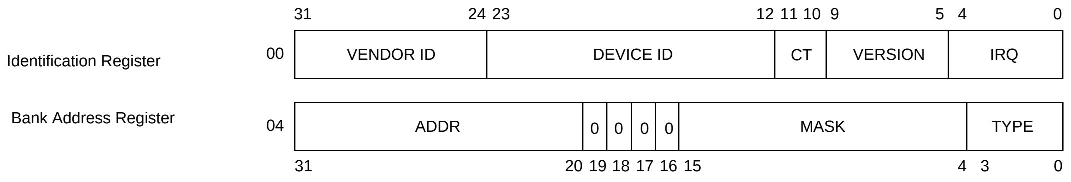


Figure 10. APB plug&play configuration layout

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 special reporting module (apbreport) is provided in the WORK.DEBUG package of Grlib which can be used to print the configuration table to the console during simulation.

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 Technology mapping

5.6.1 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 axdsp       : integer := 33;
constant spartan6    : integer := 35;
constant virtex6     : integer := 36;
```

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 instantiatein 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.

5.6.2 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 automatically 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.

5.6.3 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.

5.7 Scan test support

To support scan test methods, the GRLIB AHB and APB bus records include four extra signals: testen (test enable), scanen (scan enable), testoen (bidir control) and testrst (test reset). Scan methodology requires that all flip-flops are controllable in test mode, i.e. that they are connected to the same clock and that asynchronous resets are connected to the test reset signal. Bi-directionla or tri-state outputs should also be controllable. The four test signals are driven from the AHB bus controller (ahbctrl), where they are defined as optional inputs. The test signals are the routed to the inputs of all AHB masters and slaves (ahbmi and ahbsi records). The APB master (apbctrl) routes the test signals further to all APB slaves using the apbi record. In this way, the scan test control signals are available in all AMBA cores without additional external connections.

Cores which use the scan signals include LEON3, MCTRL and GRGPIO.

6 GRLIB Design examples

6.1 Introduction

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.

6.2 NetCard

The NetCard design example described in this section is a simple PCI to Ethernet bridge. The design is based on IP cores from GRLIB, including the GRPCI PCI bridge and the GRETH Ethernet MAC. The VHDL code of the design is listed in its full hereafter, but has been split into sections to allow for explanations after the source code. The design is located in grlib/designs/netcard.

```
library ieee;
use ieee.std_logic_1164.all;
library grlib;
use grlib.amba.all;                  -- AMBA AHB/APB components
library techmap;
use techmap.gencomp.all;             -- technology
use grlib.stdlib.all;                -- utilities
library gaisler;
use gaisler.uart.all;                -- AMBA AHB/APB UARTs
use gaisler.misc.all;                -- miscellaneous
use gaisler.pci.all;                 -- PCI
use gaisler.net.all;                 -- network cores
use work.config.all;                 -- design configuration
```

The GRLIB and GAISLER VHDL libraries are used for this design. Only the most important packages are explained. The AHB bus controller and the AHB/APB bridge components are defined in the GRLIB.AMBA package. The technology selection is defined in the TECHMAP.GENCOMP package.

```
entity netcard is
  generic (
    fabtech   : integer := CFG_FABTECH;
    memtech   : integer := CFG_MEMTECH;
    padtech   : integer := CFG_PADTECH;
    clktech   : integer := CFG_CLKTECH
  );
```

The TECH and MEMTECH generics are used for selecting the overall technology and the memory technology. It is possible to include optionally a debugger and a PCI signal tracer. It is possible to select the functionality of the PCI bridge, either as target only or as combined initiator/target.

```
port (
    resetn     : in    std_ulogic;
    clk        : in    std_ulogic;

    dsutx      : out   std_ulogic;                    -- DSU tx data
    dsurx      : in    std_ulogic;                    -- DSU rx data

    emdio      : inout std_logic;                     -- ethernet
    etx_clk    : in    std_logic;
    erx_clk    : in    std_logic;
    erxd       : in    std_logic_vector(3 downto 0);
    erx_dv     : in    std_logic;
    erx_er     : in    std_logic;
    erx_col    : in    std_logic;
    erx_crs    : in    std_logic;
    etxd       : out   std_logic_vector(3 downto 0);
    etx_en     : out   std_logic;
    etx_er     : out   std_logic;
    emdc       : out   std_logic;

    pci_rst    : in    std_ulogic;                    -- PCI
    pci_clk    : in    std_ulogic;
    pci_gnt    : in    std_ulogic;
    pci_idsel  : in    std_ulogic;
    pci_lock   : inout std_ulogic;
    pci_ad     : inout std_logic_vector(31 downto 0);
    pci_cbe    : inout std_logic_vector(3 downto 0);
    pci_frame  : inout std_ulogic;
    pci_irdy   : inout std_ulogic;
    pci_trdy   : inout std_ulogic;
    pci_devsel : inout std_ulogic;
    pci_stop   : inout std_ulogic;
    pci_perr   : inout std_ulogic;
    pci_par    : inout std_ulogic;
    pci_req    : inout std_ulogic;
    pci_serr   : inout std_ulogic;
    pci_irq    : out   std_ulogic;
    pci_host   : in    std_ulogic;
    pci_66     : in    std_ulogic);
end;
```

The interface ports of the design are all defined as standard IEEE 1164 types.

```
architecture rtl of netcard is
  signal apbi  : apb_slv_in_type;
  signal apbo  : apb_slv_out_vector := (others => apb_none);
```

Local signal declarations for the APB slave inputs and outputs. The outputs are contained in a vector and each APB slave drives it own element. Note that a default value is given to the APB output vector in the architecture declarative part. This is generally not supported for synthesis, but all synthesis tools supported by GRLIB generate all-zero values which makes the outcome determistic. If this design style is not accepted by a tool or user, the unused entries in the vector should be assigned the default value explicitly in the architecture statement part.

```
  signal ahbsi : ahb_slv_in_type;
  signal ahbso : ahb_slv_out_vector := (others => ahbs_none);
```

Local signal declarations for the AHB slave inputs and outputs. The outputs are contained in a vector, and each AHB slave drives it own element.

```
  signal ahbmi : ahb_mst_in_type;
  signal ahbmo : ahb_mst_out_vector := (others => ahbm_none);
```

Local signal declarations for the AHB masters inputs and outputs. The outputs are contained in a vector, and each AHB masters drives it own element.

```
  signal clkm, rstn, pciclk : std_ulogic;
  signal cgi   : clkgen_in_type;
  signal cgo   : clkgen_out_type;
  signal dui   : uart_in_type;
  signal duo   : uart_out_type;
  signal pcii  : pci_in_type;
  signal pcio  : pci_out_type;
  signal ethi  : eth_in_type;
  signal etho  : eth_out_type;
  signal irqn  : std_logic;
```

The rest of the local signal declarations are used for the clock generation, debugger, PCI and Ethernet interfaces.

```
begin

----------------------------------------------------------------------
---  Reset and Clock generation  -------------------------------------
----------------------------------------------------------------------
  cgi.pllctrl <= "00";
  cgi.pllrst  <= resetn;
  cgi.pllref    <= '0';

  clkgen0 : clkgen      -- clock generator
    generic map (clk_mul => 4, clk_div => 2, pcien => pci, tech => tech)
    port map (clk, pci_clk, clkm, open, open, open, pciclk, cgi, cgo);
  rst0 : rstgen         -- reset generator
    port map (resetn, clkm, cgo.clklock, rstn);
```

The clock generator can be implemented using technology specific cells, which is controlled by the CLKTECH generic.

```
----------------------------------------------------------------------
---  AHB CONTROLLER --------------------------------------------------
----------------------------------------------------------------------
  ahb0 : ahbctrl        -- AHB arbiter/multiplexer
    port map (rstn, clkm, ahbmi, ahbmo, ahbsi, ahbso);
```

The GRLIB GAISLER AHB bus controller is used for implementing the AHB arbiter, address decoder and multiplexer. All AHB master and slave inputs/outputs are route through the controller.

```
----------------------------------------------------------------------
---  ETHERNET ---------------------------------------------------------
----------------------------------------------------------------------
e0 : greth generic map(hindex => log2x(CFG_PCI),
                       pindex => 0, paddr => 11, pirq => 11, memtech => memtech)
     port map( rst => rstn, clk => clk, ahbmi => ahbmi, ahbmo => ahbmo(log2x(CFG_PCI)),
                       apbi => apbi, apbo => apbo(0), ethi => ethi, etho => etho);
```

The GRETH Ethernet interface is an AHB master and an APB slave. The generic hindex defines its AHB master number and the generic pindex defines its APB slave index. Note that hindex and the index used for selecting the correct element in the AHBMO vector must be the same. The same applies to pindex and apbo.. The two indices have no relation to the address mapping of the slave. The address of the APB bank is specified by the paddr generic, and in this case its starting address will be 0x80000B00. The IRQ generic specifies that the device will generate interrupts on interrupt vector element 11.

```
      emdio_pad : iopad generic map (tech => padtech)
      port map (emdio, etho.mdio_o, etho.mdio_oe, ethi.mdio_i);
      etxc_pad : clkpad generic map (tech => padtech, arch => 1)
                       port map (etx_clk, ethi.tx_clk);
      erxc_pad : clkpad generic map (tech => padtech, arch => 1)
                       port map (erx_clk, ethi.rx_clk);
      erxd_pad : inpadv generic map (tech => padtech, width => 4)
                       port map (erxd, ethi.rxd(3 downto 0));
      erxdv_pad : inpad generic map (tech => padtech)
                       port map (erx_dv, ethi.rx_dv);
      erxer_pad : inpad generic map (tech => padtech)
                       port map (erx_er, ethi.rx_er);
      erxco_pad : inpad generic map (tech => padtech)
                       port map (erx_col, ethi.rx_col);
      erxcr_pad : inpad generic map (tech => padtech)
                       port map (erx_crs, ethi.rx_crs);

      etxd_pad : outpadv generic map (tech => padtech, width => 4)
                       port map (etxd, etho.txd(3 downto 0));
      etxen_pad : outpad generic map (tech => padtech)
                       port map ( etx_en, etho.tx_en);
      etxer_pad : outpad generic map (tech => padtech)
                       port map (etx_er, etho.tx_er);
      emdc_pad : outpad generic map (tech => padtech)
                       port map (emdc, etho.mdc);

  irqn        <= ahbso(3).hirq(11);

  irq_pad : odpad generic map (tech => padtech, level => pci33)
  port map (pci_irq, irqn);
```

All Ethernet interface signals are mapped pads with tech mapping, selecting the appropriate pads for the selected target technology. A pad is explicitly instantiated for the interrupt output, ensuring that an open-drain output with PCI33 levels is being used.

```
----------------------------------------------------------------------
---  AHB/APB Bridge  -------------------------------------------------
----------------------------------------------------------------------
  apb0 : apbctrl -- AHB/APB bridge
    generic map (hindex => 0, haddr => 16#800#)
    port map (rstn, clkm, ahbsi, ahbso(0), apbi, apbo );
```

The GRLIB AHB/APB bridge is instantiated as a slave on the AHB bus. The HINDEX generic specifies its index on the AHB slave bus, and the HADDR generic specifies that the corresponding APB bus address area will be starting from AHB address 0x80000000.

```
----------------------------------------------------------------------
---  AHB RAM  --------------------------------------------------------
----------------------------------------------------------------------
  ram0 : ahbram
```

```
    generic map (hindex => 2, haddr => 0, hmask => 16#FFF#,
                 tech => memtech, kbytes => 8)
    port map (rstn, clkm, ahbsi, ahbso(2));
```

A local RAM is implemented as a slave on the AHB bus. The technology selection is made with the MEMTECH generic. The size is specified to be 8 kbytes with the KBYTES generic, and the memory is located at address 0x00000000 as specified by HADDR. The HMASK generic allocates a minimum 1 Mbyte address space on the AHB bus.

```
----------------------------------------------------------------------
---  PCI   -----------------------------------------------------------
----------------------------------------------------------------------
  pp : if pci /= 0 generate
    pci_gr0 : if pci = 1 generate
      pci0 : pci_target
        generic map (hindex => 0,
                     device_id => 16#0210#, vendor_id => 16#16E3#)
        port map (rstn, clkm, pciclk, pcii, pcio, ahbmi, ahbmo(0));
    end generate;
    pci_mtf0 : if pci = 2 generate
      pci0 : pci_mtf
        generic map (memtech => memtech, hmstndx => 0,
                     fifodepth => 6, device_id => 16#0210#,
                     vendor_id => 16#16E3#, hslvndx => 1,
                     pindex => 6, paddr => 2, haddr => 16#E00#,
                     ioaddr => 16#400#, nsync => 2)
        port map (rstn, clkm, pciclk, pcii, pcio, apbi, apbo(2),
                  ahbmi, ahbmo(0), ahbsi, ahbso(1));
    end generate;
    pci_trc0 : if pcitrc /= 0 generate
      pt0 : pcitrace
        generic map (memtech => memtech, pindex => 3,
                     paddr => 16#100#, pmask => 16#f00#)
        port map (rstn, clkm, pciclk, pcii, apbi, apbo(3));
    end generate;
    pcipads0 : pcipads
      generic map (tech)
      port map (pci_rst, pci_gnt, pci_idsel, pci_lock, pci_ad, pci_cbe,
                pci_frame, pci_irdy, pci_trdy, pci_devsel, pci_stop,
                pci_perr, pci_par, pci_req, pci_serr, pci_host, pci_66,
                pcii, pcio);
  end generate;
```

If the PCI interface is implemented as a target only, the device is only implemented as a master on AHB. This option does not require any on-chip memory and no technology selection is required. The PCI device and vendor ID is specified by means of generics.

For an initiator/target PCI interface, the device is implemented as both master and slave on AHB. This option implements on-chip memory for which the technology is selected with the MEMTECH generic. The size of the memory is selected with the FIFODEPTH generic and it is located at 0xE0000000 as specified by HADDR. The I/O bank of the device is located at AHB address 0x40000000. This option also implements a APB slave, and the PINDEX generic is used for specifying its APB bus number.

Not shown in this example is that there are several other generics specified for the PCI IP cores for which default values are being used. What should be noted is that most of the generics are hard coded in this example, not allowing the design to be changed by means of top level entity generics.

The pads for the PCI interface are implemented in the PCIPADS component, which only uses the TECH generic since the signal levels are already determined.

As an option, a PCI signal trace buffer can be included in the design. The trace buffer samples PCI signal activity and stores the data in a local on-chip memory. The trace buffer is accessible as an APB slave I/O bank of 4 kBytes at AHB address 0x80010000 as specified by the PADDR and PMASK generics. The 0x800 part of the address is specified by the AHB/APB bridge HADDR generic as explained above.

```
----------------------------------------------------------------------
---  Optional DSU UART -----------------------------------------------
----------------------------------------------------------------------
  dcomgen : if dbg = 1 generate
    dcom0: ahbuart   -- Debug UART
      generic map (ahbndx => 2, apbndx => 1, apbaddr => 1)
      port map (rstn, clkm, dui, duo, apbi, apbo(1), ahbmi, ahbmo(2));
```

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

An option debug support unit serial interface can be included in the design. The DSU acts as an AHB master and as an APB slave.

```
-----------------------------------------------------------------------
---  Boot message  ----------------------------------------------------
-----------------------------------------------------------------------
-- pragma translate_off

  apbrep : apbreport                  -- APB reporting module
  generic map (haddr => 16#800#)
  port map (apbo);

  ahbrep : ahbreport                  -- AHB reporting module
  port map (ahbmo, ahbso);

  x : report_version
  generic map (
   msg1 => "Network Card Demonstration design",
   msg2 => "GRLIB Version " & tost(LIBVHDL_VERSION/100) &
                        "." & tost(LIBVHDL_VERSION mod 100),
   msg3 => "Target technology: " & tech_table(tech) &
         ",  memory library: " & tech_table(memtech),
   mdel => 1
  );
-- pragma translate_on
end;
```

Finally, a component is added to the design which generates a report during simulation regarding the GRLIB version and technology selections. The component is not included in synthesis, a indicated by the pragma usage.

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

```
$ vsim -c netcard
```

Simulate the first 100 ns by writing ‘run’.

```
# Ethernet/PCI Network Card Demonstration design
# GRLIB Version 1.0.15, build 2194
# Target technology: virtex2  ,  memory library: virtex2
# ahbctrl: AHB arbiter/multiplexer rev 1
# ahbctrl: Common I/O area disabled
# ahbctrl: AHB masters: 3, AHB slaves: 4
# ahbctrl: Configuration area at 0xfffff000, 4 kbyte
# ahbctrl: mst0: Gaisler Research        Fast 32-bit PCI Bridge
# ahbctrl: mst1: Gaisler Research        GR Ethernet MAC
# ahbctrl: mst2: Gaisler Research        AHB Debug UART
# ahbctrl: slv0: Gaisler Research        AHB/APB Bridge
# ahbctrl:       memory at 0x80000000, size 1 Mbyte
# ahbctrl: slv1: Gaisler Research        Fast 32-bit PCI Bridge
# ahbctrl:       memory at 0xe0000000, size 256 Mbyte
# apbctrl: APB Bridge at 0x80000000 rev 1
# apbctrl: slv0: Gaisler Research        GR Ethernet MAC
# apbctrl:       I/O ports at 0x80000b00, size 256 byte
# apbctrl: slv1: Gaisler Research        AHB Debug UART
# apbctrl:       I/O ports at 0x80000100, size 256 byte
# apbctrl: slv3: Gaisler Research        32-bit PCI Trace Buffer
# apbctrl:       I/O ports at 0x80010000, size 64 kbyte
# apbctrl: slv6: Gaisler Research        Fast 32-bit PCI Bridge
# apbctrl:       I/O ports at 0x80000200, size 256 byte
# ahbuart1: AHB Debug UART rev 0
# pci_mtf1: 32-bit PCI/AHB bridge  rev 0, 2 Mbyte PCI memory BAR, 64-word FIFOs
# greth1: 10/100 Mbit Ethernet MAC rev 01, EDCL 0, buffer 0 kbyte 8 txfifo
# clkgen_virtex2: virtex-2 sdram/pci clock generator, version 1
# clkgen_virtex2: Frequency 25000 KHz, DCM divisor 2/2
```

The report shows that the Xilinx Virtex-2 technology is used for pads, clock generation and memories. The PCI initiator/target bridge is implemented, and the optional PCI trace buffer is included.

Generics can be provided as command line arguments to ‘vsim’. It is simple to simulate an ASIC instead of an Xilinx Virtex-2 implementation.

```
$ vsim -gtech=6 -gmemtech=3 -gclktech=0 -c netcard
```

Simulate the first 100 ns by writing ‘run’.

```
# Ethernet/PCI Network Card Demonstration design
# GRLIB Version 1.0.15, build 2194
# Target technology: atc18,  memory library: virage
# ahbctrl: AHB arbiter/multiplexer rev 1
# ahbctrl: Common I/O area disabled
# ahbctrl: AHB masters: 3, AHB slaves: 4
# ahbctrl: Configuration area at 0xfffff000, 4 kbyte
# ahbctrl: mst0: Gaisler Research        Fast 32-bit PCI Bridge
# ahbctrl: mst1: Gaisler Research        GR Ethernet MAC
# ahbctrl: mst2: Gaisler Research        AHB Debug UART
# ahbctrl: slv0: Gaisler Research        AHB/APB Bridge
# ahbctrl:       memory at 0x80000000, size 1 Mbyte
# ahbctrl: slv1: Gaisler Research        Fast 32-bit PCI Bridge
# ahbctrl:       memory at 0xe0000000, size 256 Mbyte
# apbctrl: APB Bridge at 0x80000000 rev 1
# apbctrl: slv0: Gaisler Research        GR Ethernet MAC
# apbctrl:       I/O ports at 0x80000b00, size 256 byte
# apbctrl: slv1: Gaisler Research        AHB Debug UART
# apbctrl:       I/O ports at 0x80000100, size 256 byte
# apbctrl: slv3: Gaisler Research        32-bit PCI Trace Buffer
# apbctrl:       I/O ports at 0x80010000, size 64 kbyte
# apbctrl: slv6: Gaisler Research        Fast 32-bit PCI Bridge
# apbctrl:       I/O ports at 0x80000200, size 256 byte
# ahbuart1: AHB Debug UART rev 0
# pci_mtf1: 32-bit PCI/AHB bridge  rev 0, 2 Mbyte PCI memory BAR, 64-word FIFOs
# greth1: 10/100 Mbit Ethernet MAC rev 01, EDCL 0, buffer 0 kbyte 8 txfifo
```

The report shows that the ACT18 technology is used for pads and Virage technology for the memories.

6.3 LEON3MP

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: Gaisler Research        Leon3 SPARC V8 Processor
# ahbctrl: mst1: Gaisler Research        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: Gaisler Research        AHB/APB Bridge
# ahbctrl:       memory at 0x80000000, size 16 Mbyte
# ahbctrl: slv2: Gaisler Research        Leon3 Debug Support Unit
# ahbctrl:       memory at 0x90000000, size 256 Mbyte
# ahbctrl: slv6: Gaisler Research        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: Gaisler Research        Generic UART
# apbmst:       I/O ports at 0x80000100, size 256 byte
# apbmst: slv2: Gaisler Research        Multi-processor Interrupt Ctrl.
# apbmst:       I/O ports at 0x80000200, size 256 byte
# apbmst: slv3: Gaisler Research        Modular Timer Unit
# apbmst:       I/O ports at 0x80000300, size 256 byte
# apbmst: slv7: Gaisler Research        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
```

7 Core-specific design information

7.1 LEON3 double-clocking

7.1.1 Overview

To avoid critical timing paths in large AHB systems, it is possible to clock the LEON3 processor core at an inter multiple of the AHB clock. This will allow the processor to reach higher performance while executing out of the caches. This chapter will describe how to implement a LEON3 double-clocked system using the LEON3-CLK2X template design as an example.

7.1.2 LEON3-CLK2X template design

The LEON3-CLK2X design is a multi frequency design based on double-clocked LEON3 CPU core. The LEON3 CPU core and DSU run at multiple AHB frequency internally, while the AHB bus and other AHB components are clocked by the slower AHB clock. Double clocked version of the interrupt controller is used, synchronizing interrupt level signals between the CPU and the interrupt controller.

The design can be configured to support different ratios between CPU and AHB clock such as 2x, 3x or 4x. If dynamic clock switching is enabled, an glitch-free clock multiplexer selecting between the fast CPU clock and the slower AHB clock is used to dynamically change frequency of the CPU core (by writing to an APB register).

7.1.3 Clocking

The design uses two synchronous clocks, AHB clock and CPU clock. For Xilinx and Altera technologies the clocks are provided by the *clkgen* module, for ASIC technologies a custom clock generation circuit providing two synchronous clocks with low skew has to be provided.

An AHB clock qualifier signal, identifying end of an AHB clock cycle is necessary for correct operation of the double-clocked cores. The AHB clock qualifier signal (HCLKEN), indicating end of an AHB clock cycle, is provided by the *qmod* module. The signal is generated in CPU clock domain and is active during the last CPU clock cycle during low-phase of the AHB clock. Figure 11 shows timing for CPU and AHB clock signals (CPUCLK, HCLK) and AHB clock qualifier signal (HCLKEN) for clock ratios 2x and 3x.

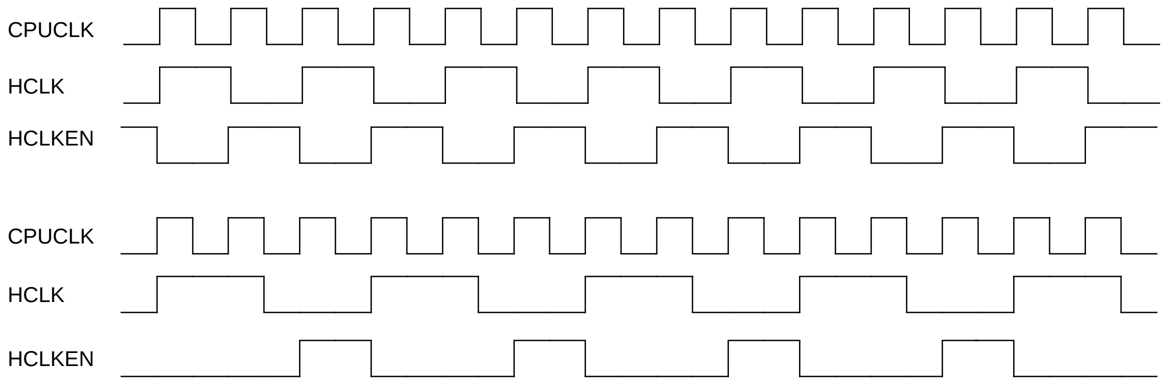


Figure 11. Timing diagram for CPUCLK, HCLK and HCLKEN

7.1.4 Multicycle Paths

Paths going through both CPU and AHB clock domains have propagation time of one AHB clock cycle, and should be marked as multicycle paths with following exceptions:

Start point	**Through**	**End point**	**Propagation time**
leon3s2x core			
CPUCLK	ahbi	CPUCLK	N CPUCLK
CPUCLK	ahbsi	CPUCLK	N CPUCLK
CPUCLK	ahbso	CPUCLK	N CPUCLK
HCLK	irqi	CPUCLK	1 CPUCLK
CPUCLK	irqo	HCLK	1 CPUCLK
CPUCLK		u0_0/p0/c0/sync0/r[*] (register)	1 CPUCLK
dsu3_2x core			
CPUCLK	ahbmi	CPUCLK	N CPUCLK
CPUCLK	ahbsi	CPUCLK	N CPUCLK
	dsui	CPUCLK	1 CPUCLK
r[*] (register)		rh[*] (register)	1 CPUCLK
irqmp2x core			
r2[*] (register)		r[*] (register)	1 CPUCLK

* N is ratio between CPU and AHB clock frequency (2, 3, ...)

Sample DC script defining multicycle paths and exceptions is provided in the design directory (*dblclk.dc*).

Figure 12 shows synchronization of AHB signals starting in HCLK clock domain and ending in CPUCLK domain (inside the double clocked cores LEON3S2X and DSU3_2X). These AHB signals are captured by registers in CPUCLK domain at the end of AHB clock cycle, allowing propagation time of 2 or more CPUCLK cycles (one HCLK cycle). The end of the AHB clock cycle is indicated by the AHB clock qualifier signal HCLKEN. One of the inputs of the AND gate in figure below is connected to the clock qualifier signal HCLKEN ensuring that the value of the signal AHBI is latched into R2 at the end of AHB cycle (HCLKEN = ‘1’). The value of signal AHBI is not valid in the CPUCLK clock domain if the qualifier signal HCLKEN is low. In this case, the AND gate will be closed and the value of the signal AHBI will not propagate to register R2.

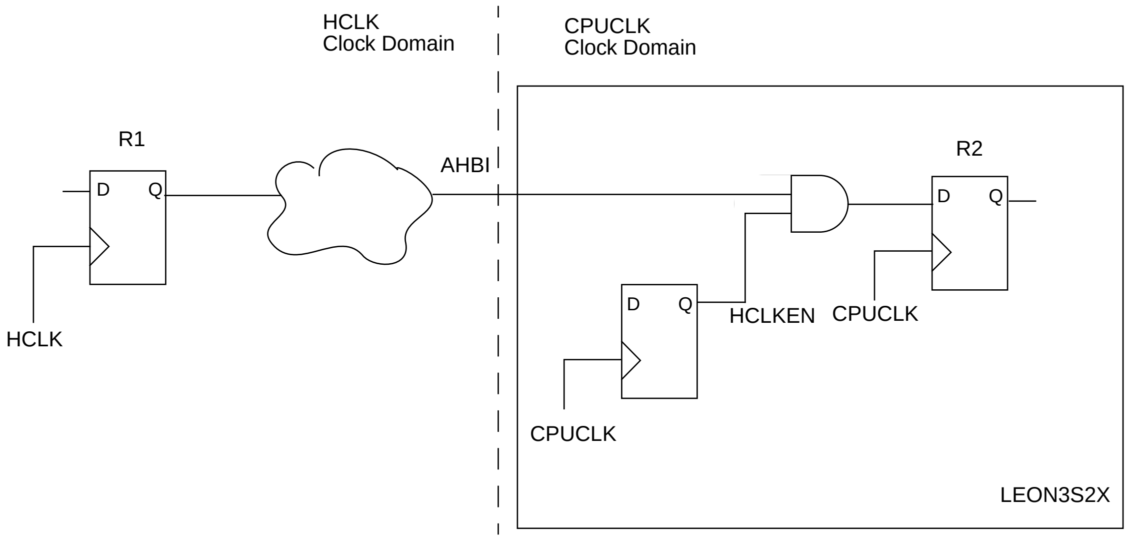


Figure 12. Synchronization between HCLK and CPUCLK clock domains

Synchronization of AHB signals going from the double clocked cores to the AHB clock domain is shown if figure 13. The AND gate is open when CPU (or DSU) performs an AHB access (AHBEN = ‘1’). When the AND gate is open, the signal AHBO will be stable during the whole AHB cycle and its value propagates to the HCLK clock domain (AHB bus). When CPU does not perform AHB access (CLKEN = ‘1’) the AND gate is closed (AHBEN = ‘0’) disabling propagation of signal AHBO to the HCLK clock domain.

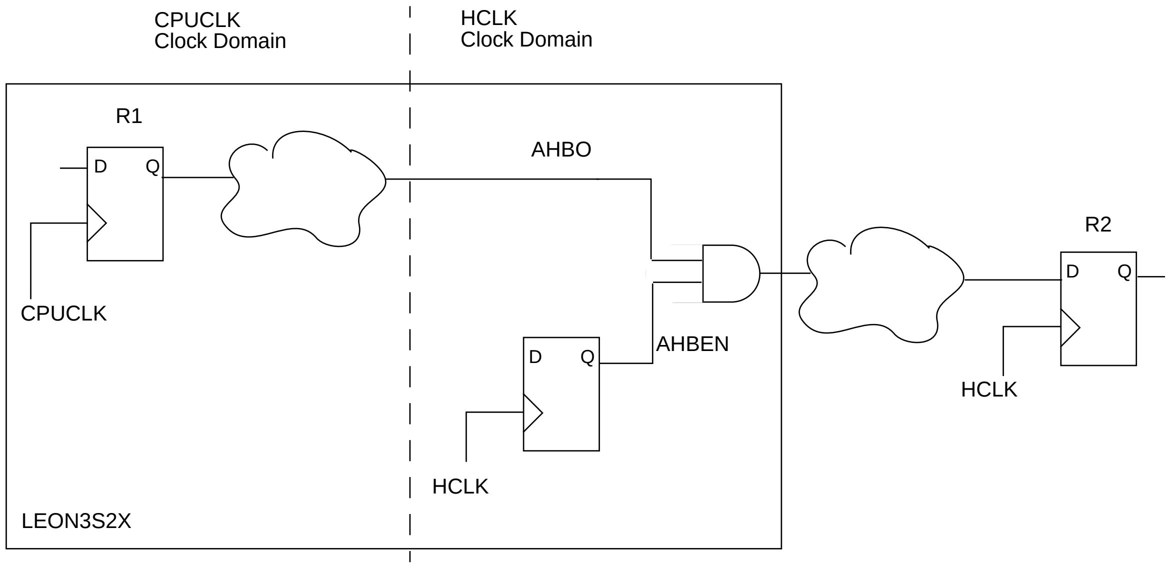


Figure 13. Synchronization between CPUCLK and HCLK clock domains

The AND gates in figures 12 and 13 are 2-input clock AND gates. Synthesis tool should not optimize these AND gates. Sample DC-script puts ‘don’t-touch’ attribute on these cells to prevent optimization.

The multicycle constraints for the GRLIB double clocked cores are typically defined by start clock domain, intermediate points and end clock domain. Although FPGA synthesis tools provide support for multicycle paths, they do not provide or have limited support for this type of multicycle constraints (start clock domain, intermediate points, end clock domain). This limitation results in over-constrained FPGA designs (multicycle paths become single cycle) which are fully functional and suitable for FPGA prototyping.

7.1.5 Dynamic Clock Switching

An optional clock multiplexer switching between the CPU and AHB clocks and providing clock for double-clocked cores can be enabled. The clock multiplexer is used to dynamically change frequency of the CPU core, e.g. CPU can run at lower AHB frequency during periods with low CPU load and at twice the AHB frequency during periods with high CPU load.

The clock switching is controlled by writing to the *qmod* modules APB register (default address 0x80000400), bit 0: writing ‘1’ will switch to the CPU clock and writing ‘0’ will switch to the AHB clock.

The clock multiplexer is glitch-free, during clock switching the deselected clock is turned-off (gated) before the selected clock is enabled and selected.

Dynamic clock switching is available for Xilinx and generic technologies.

7.1.6 Configuration

xconfig

Clock ratios 2x, 3x and 4x between CPU and AHB clock are supported. Clock ratio 2x is supported for all technologies, ratios 3x and 4x are supported for ASIC technologies. Dynamic clock switching is available for Xilinx and ASIC technologies.

leon3s2x

Double-clocked LEON3 core is configured similarly to standard LEON3 core (leon3s) through VHDL generics. An additional VHDL generic *clk2x* is set to ((clock ratio - 1) + (8 * *dyn*)) where *dyn* is 1 if dynamic clock switching is enabled and 0 if disabled.

qmod

Local *qmod* module generates AHB clock qualifier signal and optionally controls dynamic clock switching. The module is configured through VHDL - generics defining clock ratio (*clkfact*), dynamic clock switching (*dynfreq*) and address mapping of modules APB register (*pindex, paddr, pmask*).

irqmp_2x

VHDL generic *clkfact* should be set to clock ratio between CPU and AHB clocks.

7.2 GRUSBHC - USB 2.0 Host Controller

7.2.1 Overview

This section describes the USB host controller testbench included in the commercial release of GRLIB. For a description of the host controller's functionality, configuration options, and register interfaces please see the GRLIB IP Core User's Manual.

7.2.2 Testbench

The USB host controller testbench, *usbhc_tb.vhd*, is located under *verification/grusbhc* and comes with a test package and two transceiver dummy models, one for the ULPI interface and one for the UTMI+ interface. These dummy models are not fully functional behavioral models of transceivers, instead they provide the functionality necessary to test the GRUSBHC. They have also been extended so that they can partially act as a USB device and can be controlled from the testbench. The testbench can control what PID a device should answer with, if a transaction should time out and transceiver timing. The testbench does not have support for other behavioural models of transceivers. The transceiver models are controller by side band signals and not by transisitions on a modeled USB bus.

When simulating, the core is instantiated inside the testbench and can be configured through constants and generics the same way that it is configured for synthesis. The user is able to choose if a test should be performed by setting the corresponding constant for that test to true or false.

In order to reduce simulation time, the dummy models are implemented to transfer data faster than over a real USB link when operating in full- and low-speed mode. The data in simulation is normally sent in high-speed (one byte per 60 MHz clock cycle), however the protocol for communicating with the transceiver is the same and the functionality is therefore not affected. The dummy models can be forced to transfer data in real timing by setting their *realtiming* input signals, however this setting will cause some tests to fail. Many of the tests are written with the simulation timing in mind and may assume that more data will be transferred than possible on a real link.

The default is that the RTL model of the core is used, but if a VHDL netlist is available for the current configuration it can be simulated by setting the *netlist* generic to a 1 and the *tech* generic to the appropriate technology. More information about using netlists can be found in section .

7.2.2.1 Enhanced Host Controller tests

Most of the enhanced host controller tests are derived from the EHCI Compliance test specification. Although the tests are named after their models in the compliance specification they seldom perform exactly the same patterns. The naming should be seen as a guide to which functionality that is exercised. Some tests are more thorough and others are a relaxed variant. The constants to enable and disable EHC tests are named DO_EHC_TESTNAME_TEST, where TESTNAME matches one, or a combination of, EHCI Compliance tests. To fully understand the different tests please refer to the EHCI Compliance test specification and the comments for each test in *usbhc_tb.vhd*.

7.2.2.2 Universal Host Controller tests

The tests verifies that the UHCI registers and data structures are handled correctly after different events, such as device connect, resume signaling and transmission errors. A brief description of each UHC test is given in *usbhc_tb.vhd*. As mentioned above, the dummy models are faster at transferring data than a real transceiver. Therefore it is recommended to set UHC_BLO to a larger value than the default in order to minimize the risk for buffer underruns.

7.2.2.3 Example run

This section describes how to simulate the host controller with the provided testbench. First the core is configured by setting the constants in *usbhc_tb.vhd*. In this example a core with an enhanced controller, one universal controller, and two ports with ULPI interface is chosen.

UHC_BLO is increased from its default value as mentioned above. The rest of the constants are left at their default values.

```
constant INST_EHC : integer := 1;   -- Instantiate EHC
constant INST_UHC : integer := 1;   -- Instantiate UHC
constant UHC_BLO  : integer := 8;   -- UHC Buffer Limit OUT
constant NPORTS   : integer := 2;   -- Number of ports
constant UTMTYPE  : integer := 2;   -- 0 = UTMI+ 16-bit, 1 = UTMI+ 8-bit, 2 = ULPI
constant NCC      : integer := 1;   -- Number of companion controllers
constant NPCC     : integer := 2;   -- Number of ports per companion controller
```

In this example just a few tests are performed, and constants for all the tests not performed are set to false, the selected test for this run are:

```
constant DO_UHC_SUSPRESP_TEST  : boolean := INST_UHC = 1 and NPORTS > 1; -- UHC suspend
resume test
constant DO_UHC_SCHEDULE4_TEST : boolean := INST_UHC = 1; -- Queue exception
constant DO_UHC_SCHEDULE7_TEST : boolean := INST_UHC = 1; -- LS data through FS hub
constant DO_EHC_CAPREG_TEST    : boolean := INST_EHC = 1; -- Test initialization of EHC
constant DO_EHC_TD16_TEST      : boolean := INST_EHC = 1 and INST_UHC = 1; -- Port routing
test
constant DO_EHC_TD22_TEST      : boolean := INST_EHC = 1; -- Port change detect
constant DO_EHC_TD2829210_TEST : boolean := INST_EHC = 1; -- Port suspend and resume
constant DO_EHC_TD319320_TEST  : boolean := INST_EHC = 1; -- High bandwidth isochronous
transaction
constant DO_EHC_TD323324_TEST  : boolean := INST_EHC = 1; -- Split isochronous transaction
```

To run the testbench in ModelSim command line mode, issue the commands:

```
make vsim
vsim -c -quiet usbhc_tb
```

Output from the simulation is shown below. Note that the simulation always ends with a failure but that the testbench reports if all tests passed or if there were failed tests:

```
# ------------------------------------------------
# USB Host Controller Testbench Initialization
# ------------------------------------------------
# Core has byte swapped little endian registers
# Core uses byte swapped little endian for in-memory descriptors
# ------------------------------------------------
# Performing consistency check of generics
# ------------------------------------------------
# Initializing universal host controller
# ------------------------------------------------
# UHC Test: Test suspend and resume
# UHC Test: Test suspend and resume: PASSED
# UHC Test: Schedule 4 - Queue Exceptions
# UHC Test: Schedule 4: PASSED
# UHC Test: Schedule 7  - FS/LS Test
# UHC Test: Schedule 7: PASSED
# ------------------------------------------------
# Initializing test of Enhanced Host Controller
# ------------------------------------------------
# EHC Test: Checking capability registers
# EHC Test: Check capability registers: PASSED
# EHC Test: EHCI Compliance Test 1.6 started
# EHC Test: TD.1.6: PASSED
# EHC Test: EHCI Compliance Test 2.2 started
# EHC Test: TD.2.2: PASSED
# EHC Test: EHCI Compliance Test 2.8/2.9/2.10 started
# EHC Test: TD.2.8/2.9/2.10: PASSED
# EHC Test: EHCI Compliance Test 3.3 started
# EHC Test: TD.3.3: PASSED
# EHC Test: EHCI Compliance Test 3.5 started
# EHC Test: TD.3.5: PASSED
# EHC Test: EHCI Compliance Test 3.6/3.7 started
# EHC Test: TD.3.6/3.7: PASSED
# EHC Test: EHCI Compliance Test 3.8 started
# EHC Test: TD.3.8: PASSED
# EHC Test: EHCI Compliance Test 3.9 started
# EHC Test: TD.3.9: PASSED
# EHC Test: EHCI Compliance Test 3.19/3.20 started
# EHC Test: TD.3.19/3.20: PASSED
# EHC Test: EHCI Compliance Test 3.23/3.24 started
# EHC Test: TD.3.23/3.24: PASSED
# End of USB Host Controller test. All tests PASSED
# Testbench will now end simulation with a failure:
# ** Failure: Test of USB Host Controller ended
```

7.3 I^{2}C master and slave

7.3.1 Testbench

The I2C testbench, *i2c_tb.vhd*, is located under *verification/i2c* in the commercial version of GRLIB. The testbench instantiates one I2CMST core and three I2CSLV cores. The testbench's primary focus is to test the I2CSLV cores. The I2CMST core is used to generate I^{2}C stimuli and its functionality is excercised in the process. The I2CSLV cores are configured as:

I2CSLV0: I^{2}C slave with hardcoded 7 bit address
I2CSLV1: I^{2}C slave with programmable address, no 10-bit support
I2CSLV2: I^{2}C slave with programmable address and 10-bit support

The testbench contains the following tests:

- Test I2CMST reset values - This test reads all the I2CMST registers and verifies that they match the documented reset values.
- Test I2CSLV reset values - This test reads all the registers on all three I2CSLVs and verifies that they match the documented reset values.
- Test I2CSLV addressing - This test uses the I2CMST to address the slaves and transfers single bytes.
- Test I2CSLV interrupts - Tests that the I2CSLV core correctly generates interrupt requests for REC, TRA and NAK events.
- Test I2CSLV RMODE, TMODE, TV and TAV - This tests performs read and write transfers to I^{2}C slaves with 7- and 10-bit addresses. RMODE 0 and 1 is exercised as well as TMODE 0 and 1 together with TV and TAV. The test also verifies core behavior when RMODE 1 is set with an occupied receive register and when TMODE 1 is set with an empty transmit register.
- Test I2CMST arbitration lost - Verifies that I2CMST detects collisions, releases the bus and asserts the arbitration lost bit. This test may trigger bus monitors since the master will generate a short SCL cycle when it releases SCL.

Each test can be enabled or disabled by setting boolean constants. All tests are enabled by default. The designer sets the system frequenecy and the wanted SCL frequency by changing the value of the constants *SYSFREQ_HZ* and *SCLFREQ_HZ* respectively. These constants are defined at the top of *i2c_tb.vhd*. The I2CMST clock prescale value is calculated automatically.

A ModelSim .do file named i2c.do is included which initializes the waveform window. To start the simulation in ModelSim command line mode, issue the command *make vsim* and *vsim -c -quiet i2c_tb.*

7.4 SPI controller

7.4.1 Testbench

The SPICTRL testbench, available in the commercial version of GRLIB, is located under *verification/spictrl*. The testbench is described in *spictrl_tb.vhd* and contains the following tests:

- Test reset values
- Test register attributes
- Test slave select register
- Test loop mode
- Test all (legal) combinations of LEN, REV, CPOL and CPHA
- Test back-to-back transfers
- Test DIV16, PM and CLOCKGAP
- Test LR, NE, NF, OV and UN events
- Test multiple master error (mode fault)
- Test SPISEL always low

Each test can be enabled or disabled by setting boolean constants. All tests are enabled by default. The testbench uses a core with a FIFO depth of 2, this can be changed with the constant FDEPTH, defined at the top of *spictrl_tb.vhd*.

Two instantiations of the core are used, one as a slave and one as a master. The testbench directory contains a ModelSim do-file, *wave.do*. that creates a ordered waveform window. To run the testbench with modelsim in command line mode, issue the commands *make vsim* and *vsim -c -quiet spictrl_tb.*

8 Using netlists

8.1 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.

8.2 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 then the RTL VHDL code. This can be done in the xconfig configuration menu, or by setting the 'netlist' generic on core. The benefit of VHDL netlists is that the core (and whole design) can be simulated and verified without special simulation libraries.

The following Gaisler cores support the VHDL netlist format: GRFPU, GRFPU-Lite, GRSPW, LEON3FT. The netlists are available for the following technologies:

- Xilinx: GRFPU, GRFPU-Lite, GRSPW, GRUSBHC
- Actel: GRSPW, LEON3FT
- Altera: GRFPU

The Gaisler netlists have the following default configurations:

- GRFPU-Lite: simple FPC controller
- GRSPW: 16 word AHB FIFO, 16 byte TX/RX FIFO, no RMAP support, RMAP CRC support enabled. FIFO protection can be enabled/disabled through xconfig.
- LEON3FT: 8 reg windows, SVT, no hardware MUL/DIV, 2 watch-points, power-down enabled, 8 Kbyte icache with 32 bytes/line, 4 Kbyte dcache with 16/byte/line, GRFPU-Lite enabled
- GRUSBHC: One port configuration with only universal controller, one port with only enhanced controller, one port with both controllers, and two ports with both controllers. All other generics are set to their default values (see GRUSBHC section of *grip.pdf*).

Contact Gaisler Research if other settings or technologies for the netlists are required. Netlists can also be delivered for other cores, such as GRETH_GBIT.

Note that when implementing Xilinx systems, the standard (non-FT) LEON3 core is always used, even if LEON3FT is selected in xconfig. This allows the user to change the parameters to the core since the standard version of LEON3 is provided in source code.

8.3 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. Note that when using .ngo or .edf files, the 'netlist' generic on the cores should NOT be set.

8.4 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.

8.5 Known limitations

When synthesizing with Xilinx XST, the tool can crash when the VHDL netlist of GRFPU is used. In this case, use the .ngo netlist instead.

9 Extending GRLIB

9.1 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.

9.2 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 util sparc modgen amba tech

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

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

9.3 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.

9.4 Adding an existing AMBA IP core to GRLIB

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, the IRQNDX generic, and the MEMADDR and MEMMASK generics.

Below is the resulting entity for the adapted component:

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

entity ahb_example is
  generic (
    ahbndx  :     integer := 0;
    memaddr :     integer := 0;
    memmask :     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;

begin
  -- 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");

  -- 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;
```

9.5 Adding portabilty support for new target technologies

9.5.1 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.

9.5.2 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/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.

9.5.3 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.

9.5.4 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/tech/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/tech/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/gaisler/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/tech/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.

9.5.5 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 `grlib/lib/tech/techmap/maps`. The `grlib/lib/tech/techmap/gencomp/gencomp.vhd` file contains the component declarations in the GENCOMP package.

9.5.6 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 grlib/lib/tech/techmap/maps. The CLKGEN component is declared in the GENCOMP package.