POLITECNICO DI TORINO

Master Degree in Computer Engineering

Master Thesis

Validation and Test of SoC Devices

Supervisors
Matteo Sonza Reorda
Edgar Ernesto Sanchez Sanchez

Candidate
Alessandro Ciraci

December 2018

Contents

List of Figures

List of Tables

1 Introduction

Nowadays, integrated circuits (ICs) are becoming more and more complex, integrating many functions on a single silicon chip, reaching transistor counts up to tens of billions. As this complexity grows, the verification and testing steps become even more relevant in the IC development flow. These two phases have the important role of detecting bugs and defects in the design and in this Master Thesis, we will explore the feasibility of the use of a software application as a verification and testing suite for a System-on-Chip, focusing in particular on the verification and testing of an I/O peripheral of the SoC.

Verification is aimed at verifying that the implemented design respects its specifications and accomplishes the desired task; it is hence aimed at detecting design flaws and bugs injected during development.
Many verification techniques have been developed during the years, such as Universal Verification Methodology (UVM), formal verification and, in its simplest form, simulation. All these techniques rely on configuring the Device Under Test, then sending stimuli to the DUT and comparing the response with a known good vector. In this Master Thesis, we will focus on simulation as a verification technique, as it is the easiest to set up and run. To measure the goodness of a simulation as validation method, a metric is needed. This metric is called coverage and can be divided into few subgroups:

- **Statement coverage**, that is the percentage of statements (lines of HDL code) that have been covered by the stimuli.
- **Branch coverage**, that is the percentage of branches from `if` and `case` statements that have been explored during the simulation.
- **Expression coverage**, derived from evaluations of 1-bit valued expressions in the right-hand-side of assignments.
- **Condition coverage**, derived from decisions followed through `if` and ternary (`<condition> ? <true> : <false>`) statements.
- **FSM (Finite State Machine) coverage**, divided in three subcategories:
  - State coverage, that is the proportion of the number of FSM states reached in simulation.

  - Transition coverage, that is the proportion of the number of the FSM's allowed transitions followed in simulation.

- **Toggle coverage**, that is the percentage of toggles (transition from high to low and viceversa) that have happened on toggle nodes (wires and input/output ports of modules).

On the other hand, **testing** is performed during the manufacturing process and ensures that the final product is not affected by hardware defects, detecting any production flaw.
Testing relies on the generation of patterns that can excite and propagate defects in such a way that those defects manifest themselves as faults or misbehaviors on an output of the device. Nowadays, to increase test efficiency and reduce test time, many techniques have been introduced and they can all be grouped under the definition of *Design-for-Testability*; this techniques introduce dedicated hardware with the sole task of testing.
This does not change, though, the fact that patterns are needed in order to test a device, and to measure the goodness of these patterns, the **fault coverage** metric is used. This is calculated as the percentage of faults detected by the test over the total number of faults.

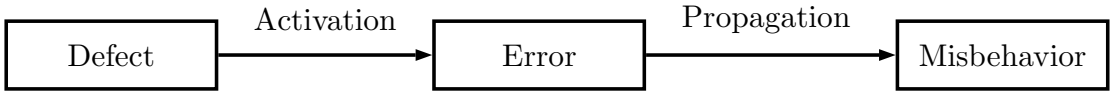

Figure 1.1: Propagation of a defect to a misbehavior

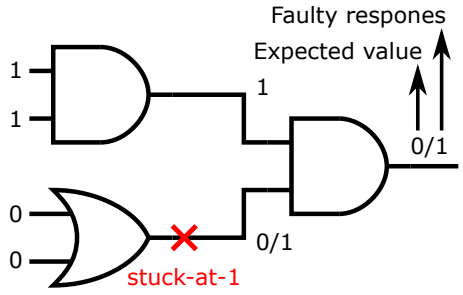

Figure 1.2: Example of stuck-at fault (*stuck-at-1*) and its propagation

As previously stated, the aim of this Master Thesis is to find if a software application can be used for both verification and testing of a SoC. This application

should be a simple mock-up of a real application, performing some input/output operations, elaborating some data and outputting the obtained results.

Document organization This document is divided into five chapters. The first one (this chapter) is a brief description of the project itself, with a general introduction to the verification and test steps in IC development. In the second chapter, the LEON3 SoC is described, including the core architecture and its peripherals, with a closer focus on the UART. It also contains a short list of other SoC candidates that were analyzed at the beginning of this project. The third chapter describes the software development steps, starting from the initial environments and their setup, then describing the steps followed during the development stage of this project. In the fourth chapter, the results are presented, comparing them with one another and explaining the reasons behind their differences. Finally, the last chapter is a conclusion that summarizes the entire project and its results and leaves some notes for future works.

2 LEON3 System-on-Chip

The first step for this Master Thesis work is the selection of a System-on-Chip on which to perform the coverage analysis. A **System-on-Chip** (SoC) is a highly integrated device, that includes one or more computing cores (such as CPUs and/or GPUs) and peripherals like timers, communication interfaces (Ethernet, UART, CAN, SPI, I^2C), input/output ports (GPIO) and even analog devices, such as Digital to Analog Converter (DAC) or Analog to Digital Converter (ADC).

The requisites needed for a suitable SoC are:

- The presence of at least an input/output peripheral (GPIO, Ethernet, etc.).
- The presence of an application that utilizes said peripheral.
- The existence of a ready-to-use environment, possibly with scripts to take care of compilations and simulations.

The chosen SoC is the Cobham Gaisler LEON3. The LEON3 is a highly configurable SoC that includes a 32-bit processor, based on the SPARC V8 architecture, originally developed by Sun Microsystems. The LEON3 package comes with a wide range of peripherals, making it particularly suitable for SoC designs and hence this Thesis work.

The processor and its peripherals are freely available on Cobham Gaisler website (`https://www.gaisler.com/index.php/downloads/leongrlib`) under GNU General Public Licence (GPL). The package includes both the LEON3 processor and the GRLIB IP library (see section 2.3). The package also provides many configurations for development; most of these options target FPGA development (whether on a development board or standalone), but it also provides an ASIC target, containing scripts for setup, synthesis and implementation on few supported technology libraries. This last configuration is the base for all development described in this Master Thesis.

2.1 LEON3 Architecture

As previously mentioned, the LEON3 is a highly configurable SoC, that can include multiple cores and several peripherals, as it can be seen in figure 2.1. These

peripherals are connected via AMBA 2.0 (Advanced Microcontroller Bus Architecture), an open-source bus architecture developed by ARM. More precisely, the peripherals that need a high bandwidth are connected to the high-speed bus, the AHB, while lower throughput peripherals are connected to the low-speed bust, the APB.

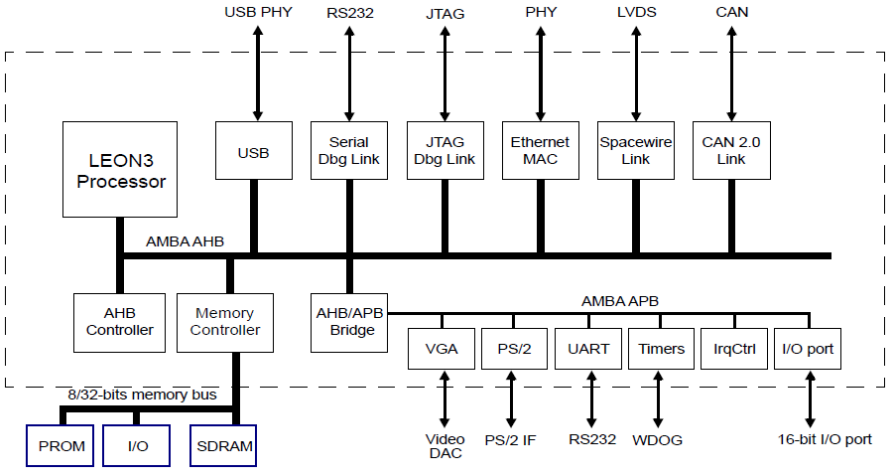

Figure 2.1: Architecture diagram of the LEON 3 SoC [1]

2.2 LEON3 Processor Core

The LEON3 processor is a 7-stage 32-bit CPU compliant with the SPARC V8 architecture and supports the V8e extension. The SPARC (Scalable Processor ARChitecture) is a Reduced Instruction Set Computer (RISC) architecture originally developed by Sun Microsystems for its Sparc server processors. It has an Harvard architecture, with two separate instruction and data caches.

[1]Image taken from the *LEON/GRLIB Guide*: *GRLIB IP Library User's Manual* [2]

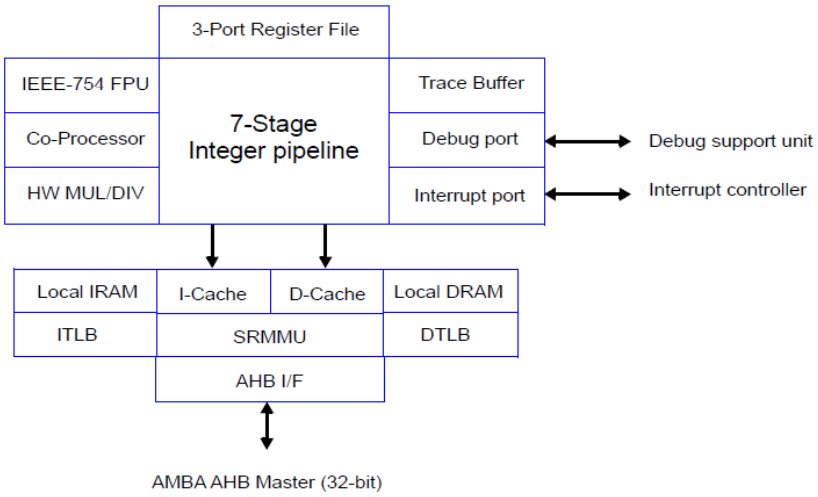

Figure 2.2: Block diagram of the LEON 3 Processor Core [2]

The SPARC architecture includes the following features [4]:

- 32bit linear address space.
- Few instructions with a simple format; 32bit wide, aligned with 32bit boundaries in memory.
- Only three instruction formats, with uniform placement of opcode and register address fields.
- LOAD/STORE memory access and I/O.
- Few addressing modes (either "register + register" or "register + immediate").
- Windowed register file, with 32 visible registers at a single time (8 global, 24 local); the window is changed whenever a procedure call or a return happens.
- Floating-point register file.
- Multiprocessor synchronization instructions.
- Coprocessor instruction set, for easy integration of new instructions.

[2]Image taken from the *LEON/GRLIB Guide*: *GRLIB IP Core User's Manual* [3]

The LEON3 processor is fully customizable via the graphical configuration tool provided (see figure 2.3) and allows the user to tailor the processor and the entire SoC to his needs.

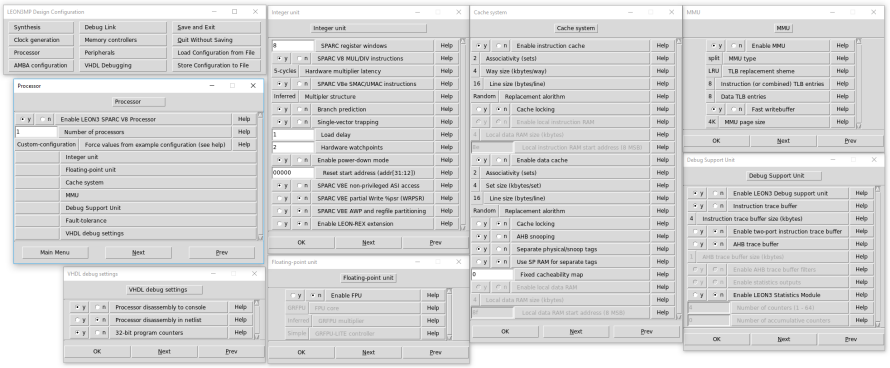

Figure 2.3: Graphical configuration tool for the LEON3 processor

2.3 LEON3 Peripherals

The Gaisler Cobham library package (GRLIB) includes several peripherals to be integrated with their processor core, with an AMBA interface(either AHB or APB). These peripherals include:

- Memories (both volatile and non-volatile)
- Serial communication interfaces (UART, SPI, I2C, CAN)
- SpaceWire link interface
- Ethernet interface
- Encryption cores
- Timers
- GPIOs

All these peripherals can be enabled or disabled, as well as configured, via the graphical configuration tool provided by Cobham Gaisler, exactly like the processor core.

2.3.1 LEON3 UART Peripheral (APBUART)

The APBUART peripheral provides a Universal Asynchronous Receiver-Transmitter interface for serial communication; it communicates with the processor core via AMBA 2.0 APB. The UART peripheral supports data frames up to 8 data bits, one optional parity bit (even or odd) and one or two stop bits. To generate the bit-rate, each UART has a programmable 12-bit clock divider. Two FIFOs (configurable via the graphical configuration tool) can be used for data transfer between the APB bus and the UART peripheral [3].

The peripheral also supports the detection of three different kind of errors:

- **Parity error**, signaling that the received data is corrupted.
- **Overrun error**, signaling that data may have been lost due to an overwrite on one of the two FIFOs.
- **Frame error**, signaling the presence of a data packet of the wrong size.

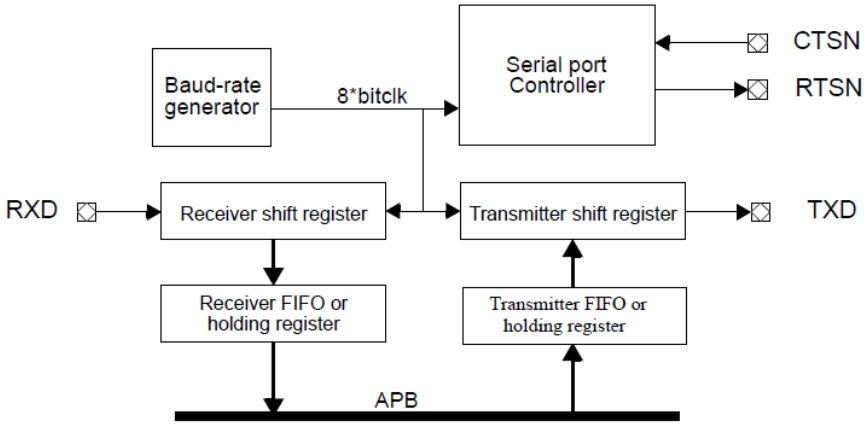

Figure 2.4: Block diagram of the LEON 3 APBUART peripheral [3]

The processor can interact with the peripheral via five memory-mapped 32-bit registers. For more details on the memory map, refer to the *LEON/GRLIB Guide*: *GRLIB IP Core User's Manual* [3].

[3]Image taken from the *LEON/GRLIB Guide*: *GRLIB IP Core User's Manual* [3]

APB address offset	Register
0x00	UART Data register
0x04	UART Status register
0x08	UART Control register
0x0C	UART Scaler register
0x10	UART FIFO Debug register

Table 2.1: LEON 3 APBUART register map

UART Data register When written, it queues the written data in the transmission FIFO; when read, it returns the first available data from the receiver FIFO.

UART Status register Contains status informations on the peripheral, for example the receiver and transmission FIFO status (full, empty, …), error status (parity, overrun, frame), data ready and transmitter empty.

UART Control register Contains the configuration fields of the peripherals, such as interrupt enable, debug modes, parity enable and polarity, transmitter and receiver enable.

UART Scaler register Contains the value for the scaler to generate the baud rate for the UART transmission and reception.

UART FIFO Debug register Is a debug register that allows direct access to the transmitter and receiver FIFOs.

2.4 LEON3 Software and OS support

Along with the GRLIB package, Cobham Gaisler provides two different software environments for their LEON3 processor.

- The **Bare-C Cross-Compiler** System (BCC) for LEON2/3/4
- The **RTEMS LEON/ERC32 Cross-Compiler** System (RCC)

Bare-C Cross Compiler The Bare-C Cross-Compiler (BCC) is "a cross-compiler for LEON2, LEON3 and LEON4 processors. It based on the GNU compiler tools, the `newlib` C library and a support library for programming LEON systems. The

cross-compiler allows compilation of C and C++ applications" [5]. It also includes some example applications.

RTEMS Cross Compiler The RTEMS LEON/ERC32 GNU cross-compiler (RCC) is a multi-platform development system, based on the GNU compiler tools [6]. The RCC package contains the following tools and packages:

- GCC C/C++ compiler.
- GNU binary utilities, with support for the LEON CASA/UMAC/SMAC instructions.
- RTEMS real-time kernel with LEON2, LEON3, LEON4 and ERC32 support (see section 3.1.2).
- `newlib` standalone C library.
- GDB SPARC cross debugger.

It also includes useful application examples that have been used as starting point for the software development described in chapter 3.

2.5 Alternative systems

Here's a short list of other candidates that have been analyzed for this Master Thesis and the reasons why they were discarded.

2.5.1 OpenRISC SoCs

Initially, the **OpenRISC** architecture was considered as a good candidate, due to its widespread use in both academic and professional fields. Another advantage is the licensing: the hardware design is licensed under the GNU Lesser General Public Licence (LGPL), while models and firmware are licensed under GNU General Public Licence (GPL).

Toolchain support is quite wide, since the community has ported the GNU toolchain to allow support of software development in both `C` and `C++`. Through this toolchain, the processor supports several widely adopted libraries, such as newlib, uClibc, musl and glibc. The OpenRISC architecture is also supported by the Linux kernel since version 3.1 and by several RTOS, including RTEMS, FreeRTOS and eCos. [7]

After an in-depth research on OpenRISC based SoCs, two candidates were found.

FuseSoC - ORPSoC The most promising one was FuseSoC, derived from the older project ORPSoC, a package manager that includes several peripherals (in the form of IPs) and a set of build tools for HDL (Hardware Description Language) code. The OpenRISC Reference Platform System-on-Chip (ORPSoC) project was designed as a reference SoC implementation based on the OpenRISC 1200 core; this SoC has been implemented on several FPGAs and there have been some commercial products derived from it.
This project provided then the necessary tools to simulate the resulting system on a various assortment of simulation suits, like GHDL, Isim, Verilator and ModelSim [8].

MiSoC Another OpenRISC implementation that has been analyzed was MiSoC. This package offers an OpenRISC CPU core, the `mor1kx`, as well as several peripherals, such as UART, GPIO and timers. It also supports built-in targets for a few FPGA development boards, although no built-in asic target support is provided [9].

Unfortunately, neither FuseSoc or MiSoC have readily available software applications that take advantage of the many peripherals they integrate; for this reason both projects were discarded as candidates for this Master Thesis.

2.5.2 PULPino

The *Parallel Ultra-Low Power* platform (PULP) is an open source hardware project based on the RISC-V core architecture. One of the developed products is the PULPino microcontroller, that includes a single-core RISC-V processor and several peripherals, connected to an AMBA AXI bus (see figure 2.5).

Unfortunately this project is quite recent, hence software availability is very scarce. At the beginning of this Master Thesis, no software application suitable for this project was available to run on the PULPino microcontroller, so this platform was discarded.

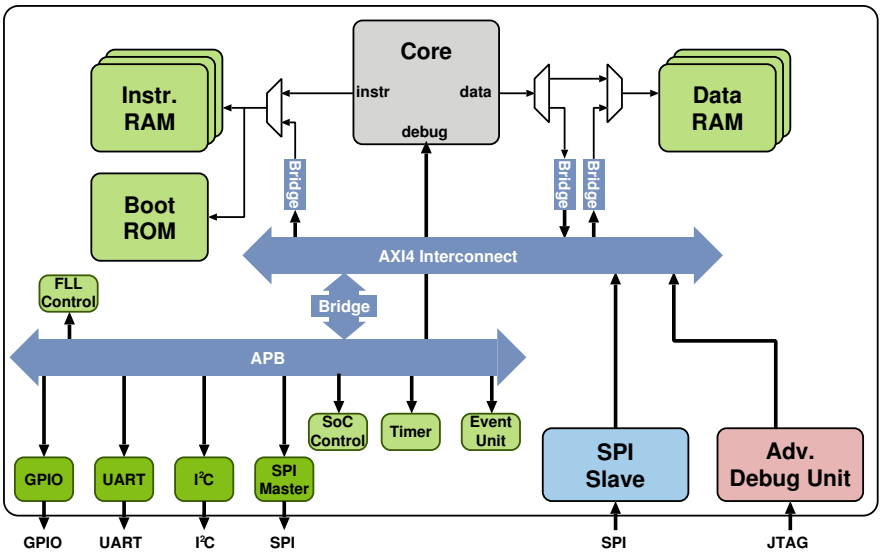

Figure 2.5: Architecture diagram of the PULPino SoC [4]

2.5.3 Ogg-on-a-Chip Project

One of the most promising devices taken into consideration was the Ogg-on-a-Chip project. Developed as a master thesis in 2005, it is a media player on a SoC, integrating both the audio decoder and the physical audio interface. [11]

The project is based on the LEON 2 System-on-Chip, from Gaisler Research. This SoC contains a SPARC v8 compliant processor core, instruction and data caches, an optionalMemory Management Unit (MMU), a Direct Memory Access (DMA), an Advanced Microcontroller Bus Architecture (AMBA) Advanced High-performance Bus (AHB) from ARM, an AMBA Advanced Peripheral Bus (APB) from ARM and several peripherals like Ethernet, GPIOs, timers, watchdogs and communication interfaces (UART, SPI, I^2C, CAN, …).

The project also includes an application to run onto the SoC, utilizing the RTEMS Operating System (see paragraph 3.1.2). This application implements a simple audio player that receives an Ogg encoded audio file (already loaded in memory), decodes it and sends it to the physical interface.

[4]Image taken from the *PULPino Datasheet* [10]

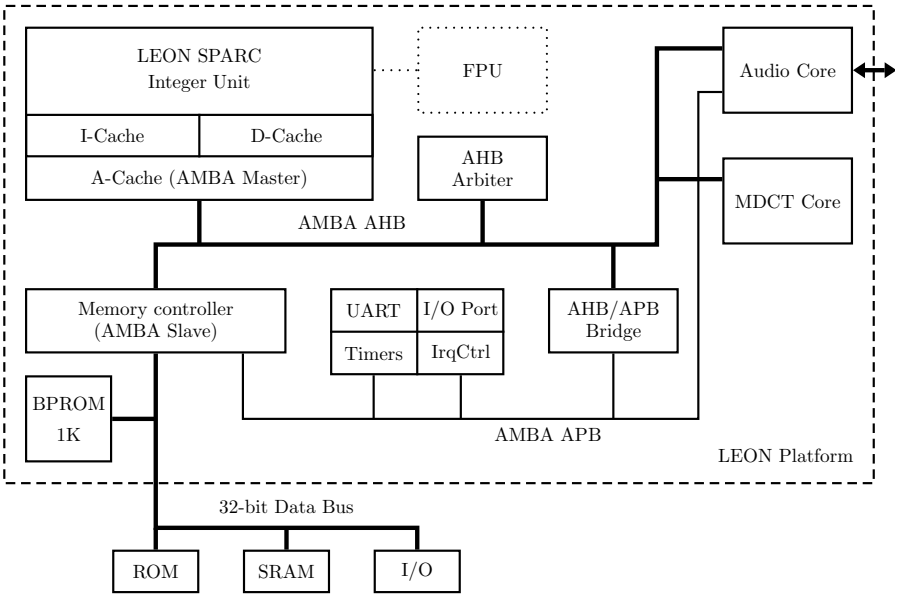

Figure 2.6: Architecture diagram of the Ogg-on-a-Chip SoC [5]

Although very promising on paper, the project was discarded, as neither the compiled software or the provided hardware would work correctly. This is most likely due to the age of the project, as the tools have changed greatly in these past 10 years and the processor used for development has been discontinued by Cobham Gaisler.

[5]Image taken from "Design of an Audio Player as System-on-a-Chip using an Open Source Platform" [12]

3 Software development

In this chapter we will describe the steps performed to obtain the results of this Thesis work. The project has been developed on two main platforms.

Local environment A laptop computer, manufacturer Hewlett-Packard (HP), model ProBook 450 G1, processor Intel i7-4702MQ, 16 GiB of RAM, operating system Windows 10 Pro 64-bit.
On the local environment, the available tools were:

- **Model*Sim*** SE-64 10.5, revision 2016.02
- **Cygwin** (kernel CYGWIN_NT-10.0; version 2.9.0(0.318/5/3))
- **RTEMS Cross-Compiler** (RCC) environment for Cygwin

Remote server A remote server of Politecnico di Torino was used to perform the synthesis of the APBUART peripheral, since it provides the required tool (Design Compiler).

3.1 Development tools and environments

This Master Thesis work relies on two hardware development tools and a specific software environment. In this section, both the tools and the software environment will be described.

3.1.1 Tools

In order to synthesize and simulate the netlist, two tools are required: Design Compiler and ModelSim. These two softwares were chosen due to their widespread use in commercial applications, support in the GRLIB environment and availability.

ModelSim The **Model*Sim*** suite by Mentor Graphics is a simulation, debug and verification platform for validating FPGA and SoC designs. This application is able to read any HDL source file and simulate the described logic, emulating its behavior at a boolean logic level. Obviously this makes Model*Sim*, as any other simulator on the market, one of the most important tools in the IC design flow.

ModelSim also simplifies RTL and gate level debugging, as it offers the capability of displaying the RTL code or the netlist as a schematic containing either behavioral blocks or gates respectively and their interconnections, thus allowing the user to track data flow through the design.

The ModelSim suite is divided into three main applications:

- `vcom` and `vlog`: a VHDL and Verilog compilators respectively that compile HDL source code for later simulation.
- `vopt`: an optimization tool that elaborates the compiled modules and optimizes the design for simulation.
- `vsim`: the simulator itself.

Mentor Graphics offers the ModelSim PE Student Edition, a free Windows version of the software, downloadable from their website (`https://www.mentor.com/company/higher_ed/modelsim-student-edition`).

Design Compiler The **Design Compiler** tool suite by Synopsys is an RTL synthesis and optimization suite aimed at the generation of a netlist from any RTL source. A netlist is a boolean gate level description of the design, that is a model of the integrated circuit built using only logic gates available in a vendor library.

The synthesis flow comprises several steps:

- **Analysis**: the RTL source code (VHDL and/or Verilog files) gets loaded in libraries by Design Compiler, while checking for syntax errors.
- **Elaboration**: the loaded modules get elaborated by Design Compiler, generating a top level design specified when calling the elaboration command.
- **Mapping**: the elaborated circuit gets mapped to one or more technology libraries, following user defined constraints for timings and power consumption.

After elaboration and mapping steps there are several optional optimization steps, aimed at reducing area, power consumption and increasing the operating frequency.

For this Thesis work, the mapping step has been skipped, as no tech library was available. As a result, the generated netlist only uses generic Synopsys gates, that do not belong to any tech library. Moreover, no optimization has been used on the synthesis flow of the APBUART peripheral.

3.1.2 Software environment

The software environment for this Master Thesis is enclosed in the RTEMS Cross-Compiler package, provided by Cobham Gaisler. This package is meant for software development on the LEON3 platform and includes both the RTEMS operating system and the GCC compiler; additionally it is possible to obtain the OS source code as well, allowing kernel and library customization.

The RCC environment also provides some software application examples for the LEON3, as well as a compilation makefile. The latter was extended to support the new applications developed.

Listing 3.1: Makefile for the software applications

```
HELLO_PROGS = # List of all software applications for this Master Thesis

all: leon-hello

build_hello: $(addprefix $(OUTDIR),$(HELLO_PROGS))

# Application specific targets - repeated for each program
$(OUTDIR)rtems-hello: rtems-hello.c $(CONFIG_DEPS) | $(OUTDIR)
  $(CC) $(CFLAGS) $< -o $@
```

The output of this compilation is an executable file, that can not be read by the simulation testbench. For this reason, another script was developed. This script reads an input ELF file and generates a SREC file; this format contains the binary data of the input ELF formatted as ASCII test and can be read by the memory model of the LEON3 during simulation.

Listing 3.2: ELF to SREC conversion script

```
#!/bin/bash
BIN_PATH="/opt/rtems-4.10-mingw/src/samples/bin/hello"
for file in $BIN_PATH/*; do
```

```
  if [ $(file ${file} | cut -d' ' -f2) == "ELF" ]; then
    sparc-rtems-objcopy -O srec $file $file.srec
  fi
done
rm -rf $BIN_PATH/srec
mkdir $BIN_PATH/srec
mv $BIN_PATH/*.srec $BIN_PATH/srec/.
```

RTEMS Operating System The **Real-Time Executive for Multiprocessor Systems** (RTEMS) Operating System, formerly Real-Time Executive for Missile Systems, and then Real-Time Executive for Military Systems, is a free open-source Real-Time Operating System (RTOS) designed for embedded systems [13]. It supports the POSIX API and it is widely used in space, medical and networking applications.

GNU Compiler Collection The **GNU Compiler Collection** (GCC) is one of the most, if not the most, widely used C/C++ compilers. Originally named GNU C Compiler, it was intended as the C compiler for the GNU operating system. Nowadays it has been ported to numerous platforms and supports countless target architectures.
The specific version included in the RCC package supports by default the different processors developed by Cobham Gaisler: LEON2, LEON3, LEON4 and ERC32. It also allows tuning of the code for higher performance, depending on the configuration of the processor.

3.2 Hardware environment setup

The hardware development environment is completely enclosed in the GRLIB package by Cobham Gaisler. The organization of this environment can be seen in the figure 3.1.

```
grlib
├── bin ..............  Contains scripts and makefiles for simulation and synthesis.
├── boards ...........  Contains environments for development board deployment.
├── design ...........  Contains environments for specific FPGA targets.
│   └── leon3-asic ...  Contains the environment (scripts and configurations) for
│                       ASIC development.
├── doc ..............  Contains the GRLIB and GRIP documentation.
├── lib ..............  Contains the RTL libraries and source code for the LEON3
│                       CPU and IPs.
├── scriptgenwork ...   Contains additional configuration files for the graphical con-
│                       figuration program.
└── software .........  Contains example C source code and precompiled applica-
                        tions for peripheral testing and verification.
```

Figure 3.1: Directory structure of the GRLIB library

The hardware environment required heavy configurations and modification. The first step was to configure the LEON3 SoC, enabling or disabling peripherals and features. The configuration used for this Master Thesis has two APBUART peripherals enabled, with both receiver and transmitter FIFO depth of 4 words. Of these two peripherals, APBUART1 will be used for all software test, while APBUART2 will be left untouched. For more details on the configuration of the LEON3 SoC, refer to appendix A.

3.2.1 Makefile and scripting

Two makefiles are provided for the `leon3-asic` configuration: a general one, located in the `grlib/bin` directory, and a specific one, that can be found in the `grlib/design/leon3-asic` directory.

The first makefile contains a set of generic targets for simulation, valid for all configurations; this makefile also defines several environment variables and paths required for simulation with all supported simulators. A few targets were added to this makefile, in order to support more simulation modes and simplify interactions with the simulator. Here's a list of the added targets:

- `vsim-run-disas`: launches `vsim` and starts simulation with console disassembly of the software running on the LEON3 core.
- `vsim-launch-disas`: launches `vsim` in graphical interface mode with console disassembly of the software running on the LEON3 core.

- `vsim-launch-nogui`: launches `vsim` in terminal mode without console disassembly of the software running on the LEON3 core.
- `vsim-launch-nogui-disas`: launches `vsim` in terminal mode without console disassembly of the software running on the LEON3 core.

To enable the disassembly on the ModelSim terminal, the parameter `-Gdisas=1` must be used while launching `vsim`; likewise, to disable the disassembly is sufficient to add the parameter `-Gdisas=0`.

The second makefile specifies configuration-dependent variables and targets for both simulation and synthesis. Here is a list of changes performed to the local makefile:

- Added coverage support for compilation (defining environment variables `VCOMOPT+= -cover sbceft` and `VLOGOPT+= -cover sbceft`).
- Added coverage support for simulation: added option `-coverage` in the `VSIMOPT` variable.
- Added targets for application selection.
- Added `select-rtl-sim` and `select-synth-sim` targets, selecting between the RTL model and the netlist of the APBUART peripheral.
- Added `simulate-all` target, to launch a sequence of simulations of all relevant applications.

Finally, the last script modified is `make.vsim`, an automatically generated file that contains only a single make target, `vsim`. This target compiles all HDL source files to allow simulation using ModelSim. Since this project requires the addition of few hardware designs, both the UART loopback module (`uart_loopback.vhd`) and the synthesized netlists of the two APBUART peripherals need to be added to this file.

Listing 3.3: ModelSim compilation script additions

```
1 vsim:
2   ...
3   # APBUART peripheral netlists and wrapper
4   vcom -quiet -cover sbceft -93 -work work apbuart_component.vhd
```

```
vcom -quiet -cover sbceft -93 -work work apbuart_syn.vhd
vcom -quiet -cover sbceft -93 -work work apbuart.vhd
vcom -quiet -cover sbceft -93 -work work leon3core.vhd
...
# UART loopback testbench module
vcom -quiet -cover sbceft -93 -work work uart_loopback.vhd
vcom -quiet -cover sbceft -93 -work work testbench.vhd
```

3.2.2 Simulation scripts and commands

Here follows a list of operations to perform in order to launch a simulation in the GRLIB package. For more details, refer to the *LEON/GRLIB Guide*: *GRLIB IP Library User's Manual* [2].

Listing 3.4: Setup and simulation flow

```
cd grlib/design/leon3-asic

# Launching the LEON3 configuration tool (alternative: make xconfig)
make xgrlib

# Compiling simulation source files
make vsim

# Selecting application and launching simulation
make sim_hello # Target selects the application and sets the environment
make vsim-run  # Alternative: make vsim-launch launches the GUI
```

The simulator receives its commands from a file, specified on the command line, `runsim.do`. This file contains the sequence of operations that the simulator has to perform and they depend on the type of simulation. For this master thesis, two simulation scripts have been written.

Listing 3.5: RTL simulation script

```
# sourcing application-specific setup
do setup.do

# running simulation
run -all
```

```
6
7  coverage report \
8    -code sbceft \
9    -all        \
10   -detail     \
11   -instance /testbench/d3/core0/leon3core0/ua1/apbuart1_inst \
12   -file report_rtl/${SOURCE_NAME}/app_toggle_cov_uart1.txt
13
14 coverage report \
15   -code sbceft \
16   -all        \
17   -detail     \
18   -instance /testbench/d3/core0/leon3core0/ua2/apbuart2_inst \
19   -file report_rtl/${SOURCE_NAME}/app_toggle_cov_uart2.txt
```

Listing 3.6: Netlist simulation script

```
1  # sourcing application-specific setup
2  do setup.do
3
4  # running simulation
5  run -all
6
7  coverage report \
8    -code t     \
9    -all        \
10   -detail     \
11   -instance /testbench/d3/core0/leon3core0/ua1/apbuart1_inst \
12   -file report_synth/${SOURCE_NAME}/app_toggle_cov_uart1.txt
13
14 coverage report \
15   -code t     \
16   -all        \
17   -detail     \
18   -instance /testbench/d3/core0/leon3core0/ua2/apbuart2_inst \
19   -file report_synth/${SOURCE_NAME}/app_toggle_cov_uart2.txt
```

3.2.3 UART Loopback module

In order to provide stimuli to the APBUART peripheral, an additional VHDL block called `uart_loopback` was developed. More specifically, this module reads the data sent by the SoC, saves it in an internal buffer and then sends it back to the LEON3. It also supports error injection in the receiver stream and it can be turned on and off via a configuration parameter.

The loopback module has several configuration options:

- `baud`: Baud rate period of the UART peripheral.
- `hbaud`: Half of the baud rate period.
- `t_idle`: Idle time to wait between data reception and data transmission.
- `t_overrun`: Time to wait before the overrun error test.
- `t_parity`: Time to wait before the parity error test.
- `t_frame`: Time to wait before the frame error test.
- `bits`: Size of the receive/transmit buffer; if equal to zero, the block is disabled.
- `par_en`: Enable/Disable parity.
- `par_o_ne`: Parity polarity (0: even; 1: odd).
- `tx`: Transmitter enable.
- `overrun_error`: Overrun error test enable.
- `parity_error`: Parity error test enable.
- `frame_error`: Frame error test enable.

This block is instantiated in the testbench, one for each APBUART peripheral, and it is connected to the RX and TX ports of the LEON3 SoC. The block is modeled as a simple sequence of operations that can be enabled or disabled via the different configuration options listed above.

3.2.4 APBUART peripheral synthesis

To allow correct measurements of coverage, the APBUART peripheral had to be synthesized using Design Compiler. Here is shown the synthesis script, that follows the standard flow. It starts by defining three main libraries (`grlib`, `gaisler` and `work`). Then, it analyzes the source files, associating them to the corresponding library. Finally, it elaborates the UART peripheral two times with different parameters and exports the elaborated netlists into a single file, `apbuart_syn.vhd`. These two generated designs correspond to the two APBUART peripherals in the LEON3 SoC and their configuration parameters were taken from the source code of the SoC itself.

Listing 3.7: Synthesis script for the APBUART peripheral

```
1  set source_path "./src"
2
3  # Read design
4  file mkdir synopsys
5  file mkdir synopsys/grlib
6  file mkdir synopsys/gaisler
7  file mkdir synopsys/work
8  define_design_lib grlib -path synopsys/grlib
9  define_design_lib gaisler -path synopsys/gaisler
10 define_design_lib work -path synopsys/work
11 analyze -f VHDL -library grlib ${source_path}/grlib/stdlib/version.vhd
12 analyze -f VHDL -library grlib
       ${source_path}/grlib/stdlib/config_types.vhd
13 analyze -f VHDL -library grlib ${source_path}/grlib/stdlib/config.vhd
14 analyze -f VHDL -library grlib ${source_path}/grlib/stdlib/stdlib.vhd
15 analyze -f VHDL -library grlib ${source_path}/grlib/amba/amba.vhd
16 analyze -f VHDL -library grlib ${source_path}/grlib/amba/devices.vhd
17 analyze -f VHDL -library gaisler ${source_path}/gaisler/uart/uart.vhd
18 analyze -f VHDL -library work ${source_path}/gaisler/uart/apbuart.vhd
19
20 # Elaborating UART instances
21 elaborate apbuart -parameter
       "pindex=1,paddr=1,pirq=2,console=0,fifosize=4"
22 elaborate apbuart -parameter
       "pindex=9,paddr=9,pirq=9,console=0,fifosize=4"
23
```

```
# Generating VHDL netlist
write_file -format vhdl -output synopsys/output/apbuart_syn.vhd
    "apbuart_pindex1_paddr1_console0_pirq2_fifosize4
    apbuart_pindex9_paddr9_console0_pirq9_fifosize4"
```

The synthesized netlists are exported in a VHDL format file, containing both UART peripherals and the VHDL models of the gates used in the designs. To simplify integration with the processor core, two wrappers have been written, with the same interface of the RTL model of the APBUART peripheral, but instantiating the synthesized netlists instead.

Listing 3.8: APBUART wrapper example

```
entity apbuart1 is
  generic (
    pindex  : integer := 0;
    paddr   : integer := 0;
    pmask   : integer := 16#fff#;
    console : integer := 0;
    pirq    : integer := 0;
    parity  : integer := 1;
    flow    : integer := 1;
    fifosize : integer range 1 to 32 := 1;
    abits   : integer := 8;
    sbits   : integer range 12 to 32 := 12
  );
  port (
    rst   : in std_ulogic;
    clk   : in std_ulogic;
    apbi  : in apb_slv_in_type;
    apbo  : out apb_slv_out_type;
    uarti : in uart_in_type;
    uarto : out uart_out_type
  );
end;

architecture wrapper of apbuart1 is

  component apbuart_pindex1_paddr1_console0_pirq2_fifosize4 is
  port( ... );
```

```
28   end component apbuart_pindex1_paddr1_console0_pirq2_fifosize4;
29
30   signal pindex_s : std_logic_vector(4 downto 0);
31
32 begin
33
34   apbo.pindex <= to_integer(unsigned(pindex_s));
35
36   apbuart_syn1 : apbuart_pindex1_paddr1_console0_pirq2_fifosize4
37   port map( rst, clk, apbi.psel, apbi.penable, apbi.paddr, apbi.pwrite,
       apbi.pwdata, apbi.pirq, apbi.testen, apbi.testrst, apbi.scanen,
       apbi.testoen, apbi.testin, apbo.prdata, apbo.pirq, apbo.pconfig(0),
       apbo.pconfig(1), pindex_s, uarti.rxd, uarti.ctsn, uarti.extclk,
       uarto.rtsn, uarto.txd, uarto.scaler, uarto.txen, uarto.flow,
       uarto.rxen
38   );
39
40 end architecture wrapper;
```

In order to support the synthesized netlist, two copies of `leon3core.vhd` were generated:

- `leon3core_rtl`, that instantiates the two APBUART peripherals as RTL models.
- `leon3core_synth`, that instantiates the two APBUART peripherals as synthesized netlists.

As previously stated, no tech library was used during synthesis. Due to this choice, Synopsys could not infer any library specific gate, leaving a netlist composed of GTECH (Generic Synopsys Technological Library) gates. This generic library does not contain any sequential element, though, so Design Compiler instantiated the `SYNOPSYS_GENERIC_SEQUENTIAL_ELEMENT`. This sequential elements are as generic as possible, including both synchronous and asynchronous clear and set lines. This is a very unrealistic implementation, as real tech gates would only include either the synchronous or the asynchronous ones, not both. For this reason, a simplified sequential element (`SIMPLIFIED_SEQUENTIAL_ELEMENT`) was created, with a reduced port count to more resemble standard Flip-Flops (see appendix B for more details).

During software development, two bug in the synthesized design were found. The first bug blocked the incoming data from the RX pin of the peripheral to reach the internal shift register, making the entire receiver logic unusable. This bug was solved by fixing the input structure from the RX pin, comparing it to the expected behavior of the RTL simulations.
The second bug discovered was on the internal clock/baud rate generator, which would get corrupted due to a non initialized register. This bug was solved by manually adding a reset signal to the affected register, controlled by the master reset of the peripheral.

3.3 Coverage analysis

ModelSim allows the evaluation of several coverage metrics during simulation. These metrics, already introduced at the beginning of this document, are described in ModelSim's User Manual as follows [14].

- **Statement coverage**, that counts the execution of each statement on a line individually, even if there are multiple statements in a line.
- **Branch coverage**, that counts the execution of each conditional `if/then/else` and `case` statement and indicates when a true or false condition has not executed.
- **Condition coverage**. that analyzes the decision made in `if` and ternary statements and can be considered as an extension to branch coverage.
- **Expression coverage**, that analyzes the expressions on the right hand side of assignment statements, and is similar to condition coverage.
- **Toggle coverage**, that counts each time a logic node transitions from one state to another.
- **FSM coverage**, that counts the states, transitions, and paths within a finite state machine.

Additionally, ModelSim can calculate both condition and expression coverages in several different ways.

- **Focused Expression Coverage** (FEC), that measures the coverage for each input of an expression. In FEC, an input is considered covered only when other

inputs are in a state that allow it to control the output of the expression. Furthermore, the output must be seen in both 0 and 1 states while the target input is controlling it. If these conditions occur, the input is said to be fully covered. The final FEC coverage number is the number of fully covered inputs divided by the total number of inputs.

- **User Define Primitive** (UDP), that measures the coverage as number of hit rows of an UDP table. A UDP table (similarly to Verilog's own UDP tables) describes the full range of behavior for a given expression, where each row corresponds to a coverage bin. If the conditions described by a row are observed during simulation, that row is said to be hit; all rows in the UDP table must be hit for UDP coverage to reach 100
- **Sum-of-Products**, that checks that each set of inputs that satisfies the expression (results in a `1`) must be exercised at least once, but not necessarily independently.
- **Basic Sub-Condition**, that checks that each subexpression has been both true and false.

The method chosen for this Master Thesis is the first one, the FEC, since it provides the widest coverage of the design, as well as being the default method used by ModelSim.

To evaluate these metrics and save them into a file, the following command has been used after the end of the simulation, to obtain the final results of the test.

Listing 3.9: Coverage report command

```
coverage report
  -code <s|b|c|e|f|t>
  [-recursive]
  [-detail]
  [-all]
  -instance <instance_name>
  -file <output_filename>
```

In order to get a measure of the goodness of the software application as verification test, all coverage metrics were evaluated during RTL simulation. This is a

common use case for these metrics, as they are widely used for simulation-based verification, with the difference that in typical verification flows a dedicated testbench is developed in order to maximize these metrics.

For the netlist simulations, ideally the best metric to evaluate the test goodness would have been the fault coverage. Unfortunately this metric can only be evaluated with the use of fault simulators provided with ATPG tools, such as Tessent or TetraMAX. Since these tools would require a significant amount of work to create a suitable setup, the fault coverage has been "approximated" using the toggle coverage statistic calculated on the synthesized peripheral. This approximation is possible because of the similarity between toggle coverage and the stuck-at fault model.

This fault model describes defects as wires of a netlist that are stuck at a fixed value, independently of their driver. This means that, to detect a stuck-at fault, the faulty wire needs to be driven to the opposite value of the fault (this corresponds to exciting the fault); the "result" then needs to be propagated to an output that can be observed, in such a way that the misbehavior can be observed and the fault is detected. This means that, in order to detect a fault, the test must be able to toggle every wire both high and low; hence the toggle coverage can be used as an approximation of the fault coverage, showing which nets are toggled by the test. On the other hand, toggle coverage does not take into account the error propagation through the netlist, as these can be easily blocked and obscured by other signals; this means that toggle coverage is the upper limit of the stuck-at fault coverage.

3.4 Software applications

A key phase of this Master Thesis is the generation, simulation and analysis of a set of software applications to run on the LEON3 SoC. As a first step, the `hello_world` program provided by the RCC package is simulated, collecting coverage figures from both UART peripherals. From this data, a baseline for future tests can be drawn, showing coverage for a basic peripheral utilization (APBUART1) and the coverage figure for no active use at all (APBUART2).

Iterating on these results, many software applications were developed in order to increase all coverage figures. In the following subsections are described the incremental steps that have been followed to reach the final results, as well as the

reasons behind every choice.

Here is a list of the five applications that have been used in this Master Thesis.

- `hello_world`, a simple application that prints the string "Hello World" to console (see section 3.4.1).
- `write_compact`, an optimized write test that is aimed at reducing the test time, without reducing coverage with respect to the baseline (see section 3.4.2).
- `write_exhaustive`, an exhaustive test that is aimed at increasing coverage at the cost of a much greater test time (see section 3.4.2).
- `read_write`, a test that excites both receiver and transmitter logic (see section 3.4.3).
- `error_injection`, a test that is aimed at covering the blocks left untouched by the previous test by causing error conditions (see section 3.4.4).

3.4.1 Starting point - "Hello World"

Since the APBUART peripheral had been selected for this analysis, the first software to be simulated was the `hello_world` program. This classic program boots the SoC, launching the OS and configuring the peripherals, then sends the string "Hello World" to the console via the APBUART1 peripheral.

Due to its simplicity, this program is meant just as a baseline and starting point for following tests, that can be built up from this common origin; as such the first optimization is to remove the 100 ms wait and the second print to the debug console.

Listing 3.10: Baseline test: `hello_world`

```
1  // Main task - entry point after OS boot
2  rtems_task Init(
3    rtems_task_argument ignored
4  )
5  {
6    printf( "Hello World\n" );
7    //rtems_task_wake_after(100);
8    //printk( "Hello World over printk() on Debug console\n\n" );
9    exit( 0 );
10 }
```

3.4.2 Transmitter subsystem test

After running the baseline test, two optimizations were developed to improve the test on the transmission logic: `write_compact` and `write_exhaustive`.

The first optimization (`hello_compact`) is intended to reduce the test time by replacing the "Hello World" string with a more test-focused one. By using the four value string `{0xF0, 0x0F, 0xAA, 0x55}`, full utilization of the FIFO can still be achieved, while exciting any possible combination of transitions on the transmitter logic.

Listing 3.11: Transmitter test: `write_compact`

```
// Main task - entry point after OS boot
rtems_task Init(
  rtems_task_argument ignored
)
{
  char string[5] = {0xf0, 0x0f, 0xaa, 0x55, 0x00};

  // Transmitting the 4 characters string
  // (last character is the string terminator)
  printf( "%s", string );
  exit( 0 );
}
```

On the other hand, the second optimization (`write_exhaustive`) is aimed at testing the transmission of every possible value for the data, which corresponds to a total of 256 (2^8) possible values.

Listing 3.12: Transmitter test: `write_exhaustive`

```
// Main task - entry point after OS boot
rtems_task Init(
  rtems_task_argument ignored
)
{
  char string[256];
  uint8_t i;

```

```
  for(i = 0x00; i < 0xFF; i++){
  string[(int) i]=(char) (((uint8_t) 0xFF) - i);
  }

  // Transmitting the 256 characters string
  // (last character is the string terminator)
  printf( "%s", string );
  exit( 0 );
}
```

3.4.3 Receiver subsystem test

The next step is to test the receiver logic and, for this purpose, the `read_write` application was developed. This software needs to read back the string it sent, so the UART loopback module provides this functionality.

Initially, the software read was implemented using the standard input/output function `scanf`, but the application would not perform any read in simulation, getting stuck in an infinite wait for data. A second attempt using the POSIX `read` function gave the same result.

To work around this issue, the final application developed to test the receiver logic relied on low-level register accesses to check for data availability and reading the data itself.

Listing 3.13: Receiver/Transmitter test: `read_write`

```
// Main task - entry point after OS boot
rtems_task Init(
  rtems_task_argument ignored
)
{
  int  i;
  char buf1[5] = {0xf0, 0x0f, 0xaa, 0x55, 0x00};
  char buf2[5] = {0x00, 0x00, 0x00, 0x00, 0x00};

  // Register addresses declaration ...

  printf("%s", buf1);
```

```
13    fflush(NULL);
14
15    // Waiting for read
16    for(i = 0; i < 5000; i++); // busy-waiting loop: ~1ms
17
18    // Printing status register contents
19    printf("%d", (unsigned int) *uart_stat_reg);
20    fflush(NULL);
21    for(i = 0; i < 500; i++); // busy-waiting loop: ~100us
22
23    // Reading data from RX FIFO
24    for(i = 0; i < 4; i++) {
25      buf2[3 - i] = (char) (*uart_data_reg & 0x000000ff);
26    }
27
28    // Printing received data
29    printf("%s", buf2);
30    fflush(NULL);
31
32    exit(0);
33  }
```

3.4.4 Additional improvements

The analysis on the read test results exposed some logic blocks with low coverage. These blocks were mainly the interrupt generation logic and the error detection systems. To test the latter, the software and the loopback module in the testbench both needed to generate error conditions:

- **Parity error**: to trigger a parity error, the loopback module sends one of the data packets with its parity bit inverted, thus triggering a parity error.
- **Overrun error**: to trigger an overrun error, the software first disables the transmitter via register write, then writes eight values to the transmission FIFO; this generates an overrun error condition, as words get overwritten in the FIFO.
- **Frame error**: to trigger a frame error, the loopback module keeps the RX line of the SoC low for the equivalent time of 16 clock cycles, violating the frame size of a UART packet.

As can be seen in the code below, this application starts by disabling the transmitter. In this way, an overrun error can be triggered by writing eight values to the 4-word transmitter FIFO. After the error is triggered and the second set of data is written, the transmitter is enabled, so that the write operation takes place. Then the software waits for available data to be read back from the UART peripheral. At this point, the UART loopback module also triggers both the parity and frame errors as described above. Finally, the status register is read and subsequently cleared by the application and the received data is printed one last time.

Listing 3.14: Read/write test with error triggering: `error_injection`

```
// Main task - entry point after OS boot
rtems_task Init(
  rtems_task_argument ignored
)
{
  int  i;
  char buf1[5] = {0xf0, 0x0f, 0xaa, 0x55, 0x00};
  char buf2[5] = {0x00, 0x00, 0x00, 0x00, 0x00};

  // Register addresses declaration ...

  // Disabling Receiver and Transmitter
  (*uart_ctrl_reg) &= ~(uart_rx_en_mask | uart_tx_en_mask);

  // Filling transmit buffer
  for (i = 0; i < 4; i++) {
    (*uart_data_reg) = buf1[i];
  }

  // Writing again on transmit buffer, triggering an overrun error
  for (i = 0; i < 4; i++) {
    (*uart_data_reg) = buf1[i];
  }

  // Enabling transmission and reception
  (*uart_ctrl_reg) |= (uart_rx_en_mask | uart_tx_en_mask);

  // Waiting for read
  for(i = 0; i < 10000; i++); // busy-waiting loop: ~2ms
```

```
30
31    // Printing status register contents
32    printf("%d", (unsigned int) *uart_stat_reg);
33    fflush(NULL);
34    (*uart_stat_reg) = 0; // Clearing error bits
35    for(i = 0; i < 500; i++); // busy-waiting loop: ~100us
36
37    // Reading back data from external module
38    for(i = 0; i < 4; i++) {
39      buf2[3 - i] = (char) (*uart_data_reg & 0x000000ff);
40    }
41
42    // Transmitting received data
43    printf("%s", buf2);
44    fflush(NULL);
45
46    exit(0);
47  }
```

4 Results

In this chapter, the results obtained during this Master Thesis will be presented and analyzed. To define a baseline for the following tests, the coverage measured on the second UART peripheral APBUART2 is analyzed. This peripheral is left untouched by all tests and can be considered as a minimum coverage baseline, obtained by just booting the SoC.

Coverage metric	Active nodes	Hits	Misses	Coverage [%]
Statement	219	104	115	47.4 %
Branches	157	46	111	29.2 %
Conditions	12	1	11	8.3 %
Expressions	52	0	52	0.0 %
FSMs - States	0	0	0	100.0 %
FSMs - Transitions	0	0	0	100.0 %
Toggle	328	117	211	35.6 %

Table 4.1: Coverage metrics for APBUART2 - RTL simulation

Coverage metric	Active nodes	Hits	Misses	Coverage [%]
Toggle	6226	1,599	4,627	25.6 %

Table 4.2: Coverage metrics for APBUART2 - Netlist simulation

As it can be seen in tables 4.1 and 4.2, these results are low, as expected from an inactive peripheral. Another interesting point that can be extrapolated from this data is the absence of any Finite State Machine in the design, as suggested by the zero active nodes. This means that FSM coverage is of no meaning for this specific peripheral and can be dropped from subsequent analysis.

4.1 `hello_world` application results

Next, we establish a baseline on an active peripheral by analyzing the results of the `hello_world` application.

Coverage metric	Active nodes	Hits	Misses	Coverage [%]	Increase[1]
Statement	219	139	80	63.4 %	16.0 %
Branches	157	75	82	47.7 %	18.5 %
Conditions	12	3	9	25.0 %	16.7 %
Expressions	52	9	43	17.3 %	17.3 %
Toggle	328	146	182	44.5 %	8.9 %

Table 4.3: Coverage metrics for the `hello_world` test - RTL simulation

Coverage metric	Active nodes	Hits	Misses	Coverage [%]	Increase[1]
Toggle	6,226	3,180	3,046	51.0 %	25.4 %

Table 4.4: Coverage metrics for the `hello_world` test - Netlist simulation

Another useful metric that has been measured is the elapsed time to complete these tests. More specifically, two times were measured.

- **Simulation time**, which is the internal time of the simulator, defining the data delays and clock periods of the simulated design; this time is usually expressed in fractions of seconds (in this Master Thesis, milliseconds).
- **Execution time**, which is the real time that the simulator took to complete its simulation; it is usually expressed in minutes or even hours, depending on the complexity of the simulation.

	Simulation type	
Time	RTL simulation	Netlist simulation
Simulation Time	10.860 ms	10.869 ms
Execution Time	19 min 42 s	21 min 45 s

Table 4.5: Simulation and execution time of the `hello_world` test

4.2 `write_compact` application results

In order to reduce both simulation and execution time, a second, optimized application was developed. Here the results of this test case are presented. As it can

[1]This coverage increase is measured with respect to the inactive peripheral baseline

be seen in the tables below, the coverage results are almost identical to the original `hello_world` application, with the desired reduction in simulation and execution time.

Coverage metric	Active nodes	Hits	Misses	Coverage [%]	Increase[2]
Statement	219	139	80	63.4 %	0.0 %
Branches	157	75	82	47.7 %	0.0 %
Conditions	12	3	9	25.0 %	0.0 %
Expressions	52	9	43	17.3 %	0.0 %
Toggle	328	146	182	44.5 %	0.0 %

Table 4.6: Coverage metrics for the `write_compact` test - RTL simulation

Coverage metric	Active nodes	Hits	Misses	Coverage [%]	Increase[2]
Toggle	6,226	3,117	3,109	50.0 %	−1.0 %

Table 4.7: Coverage metrics for the `write_compact` test - Netlist simulation

	Simulation type	
Time	RTL simulation	Netlist simulation
Simulation Time	9.808 ms	9.814 ms
Execution Time	18 min 12 s	20 min 12 s

Table 4.8: Simulation and execution time of the `write_compact` test

The difference in toggle coverage between the two netlist simulations is due to a slightly lower coverage of the transmission FIFO, caused by the lower volume of data transmitted from the peripheral. As will be shown in the next sections, this drop in coverage is recovered with the subsequent tests.

4.3 `write_exhaustive` application results

As an attempt to increase transmitter coverage, this third application, called `write_exhaustive`, was developed. Here the results obtained with this application are listed.

[2]These coverage increase are measured with respect to the `hello_world` test results

Coverage metric	Active nodes	Hits	Misses	Coverage [%]	Increase[3]
Statement	219	139	80	63.4 %	0.0 %
Branches	157	75	82	47.7 %	0.0 %
Conditions	12	3	9	25.0 %	0.0 %
Expressions	52	9	43	17.3 %	0.0 %
Toggle	328	146	182	44.5 %	0.0 %

Table 4.9: Coverage metrics for the `write_exhaustive` test - RTL simulation

Coverage metric	Active nodes	Hits	Misses	Coverage [%]	Increase[3]
Toggle	6,226	3,245	2,981	52.1 %	1.1 %

Table 4.10: Coverage metrics for the `write_exhaustive` test - Netlist simulation

	Simulation type	
Time	RTL simulation	Netlist simulation
Simulation Time	40.412 ms	40.425 ms
Execution Time	61 min 35 s	60 min 56 s

Table 4.11: Simulation and execution time of the `write_exhaustive` test

The data shows that this test achieves little to no coverage gains with respect to the `write_compact` test, with a severe penalty in both execution time (now increased by three times) and simulation time (now increased by four times). For this reason, the `write_compact` application is chosen as base for developing the subsequent tests.

4.4 `read_write` application results

Next, we will analyze the results obtained from the `read_write` application, that targets the receiver logic by performing read operations. The expected result is a noticeable increase in all coverage metrics, as this test excites parts of the design previously left untested.

[3]These coverage increase are measured with respect to the `hello_world` test results

Coverage metric	Active nodes	Hits	Misses	Coverage [%]	Increase[4]
Statement	219	164	55	74.8 %	11.4 %
Branches	157	105	52	66.8 %	19.1 %
Conditions	12	3	9	25.0 %	0.0 %
Expressions	52	14	38	26.9 %	9.6 %
Toggle	328	148	180	45.1 %	0.6 %

Table 4.12: Coverage metrics for the `read_write` test - RTL simulation

Coverage metric	Active nodes	Hits	Misses	Coverage [%]	Increase[4]
Toggle	6,226	4,426	1,800	71.0 %	20.0 %

Table 4.13: Coverage metrics for the `read_write` test - Netlist simulation

	Simulation type	
Time	RTL simulation	Netlist simulation
Simulation Time	11.296 ms	11.310 ms
Execution Time	21 min 53 s	23 min 27 s

Table 4.14: Simulation and execution time of the `read_write` test

While on the netlist simulation we get a 20 % toggle coverage increase as expected, the coverage gains in the RTL simulations are not quite as good. This is most likely due to the coding style adopted to design the peripheral itself and its high level of configurability, that forces the test to cover every possible configuration.

4.5 `error_injection` application results

The `error_injection` application was developed to cover the test misses of the previous application, exploring the least tested corners of the design. For this reason, a significant increase in branch coverage is expected, with moderate increases in all other metrics.

[4]These coverage increase are measured with respect to the `hello_world` test results

Coverage metric	Active nodes	Hits	Misses	Coverage [%]	Increase[5]
Statement	219	174	45	79.4 %	16.0 %
Branches	157	116	41	73.8 %	26.1 %
Conditions	12	4	8	33.3 %	8.3 %
Expressions	52	19	33	36.5 %	19.2 %
Toggle	328	149	179	45.4 %	0.9 %

Table 4.15: Coverage metrics for the `error_injection` test - RTL simulation

Coverage metric	Active nodes	Hits	Misses	Coverage [%]	Increase[5]
Toggle	6,226	4,697	1,529	75.4 %	24.4 %

Table 4.16: Coverage metrics for the `error_injection` test - Netlist simulation

	Simulation type	
Time	RTL simulation	Netlist simulation
Simulation Time	11.994 ms	12.009 ms
Execution Time	22 min 47 s	25 min 4 s

Table 4.17: Simulation and execution time of the `error_injection` test

As can be seen in the tables above, the expectations are fully met, with an increase of branch coverage of 26.1 % over the Hello World baseline and a 7.0 % over the receiver test. This test also yields significant increases in both condition and expression coverage metrics, with an increase of respectively 8.3 % and 9.6 % with respect to the receiver test.

This last test still leaves some areas of the design untested. One of this areas is the reset logic, that is only activated at power on and can not be controlled via software. The other untested area is the interrupt generation logic. The APBUART peripheral is able to generate an interrupt for many different events, such as errors or empty/full FIFOs. In order to test this logic, the interrupts must be enabled in the peripheral and the software must implement the corresponding interrupt service routine (ISR). Then, all interrupt conditions must be independently triggered and the event must be observed in software, when the ISR is called.

[5]These coverage increase are measured with respect to the `hello_world` test results

5 Conclusion

In conclusion, in this Master Thesis, a System-on-Chip (SoC) was chosen to perform a feasibility analysis for the use of a software application in verification and testing steps. The chosen SoC was the LEON3 and the analysis was conducted on one of its peripherals, the APBUART. At this point, software applications were developed and tested in order to improve results.

From the obtained data, it is possible to conclude that the use of applications as verification and test patterns is possible, depending on the specific situation. For example, software applications can be used as early verification patterns during the development of a new peripheral for an existing SoC. In this use-case, simulation platform for the SoC would be already available, as well as software compilation suites.

On the other hand, testing results were too low to justify the use of software applications as production test patterns (where is required a minimum fault coverage of 90%+), especially considering that the toggle coverage figures are an overestimation of the achievable fault coverage. Said that, these software applications might be used as a useful starting point for functional test stimuli in those cases where the IC requirements prevent the use of DfT techniques or when DfT must be complemented by other test steps to increase the real defect coverage (e.g., in System Level Test).

Future developments can be aimed at increasing coverage on the currently analyzed peripheral and/or expand the experiment on other peripherals. A clear example might be the extension of the currently developed software to support interrupts from the UART peripheral. This would rely on the integrated IRQ controller and it would make use of the RTEMS operating system Interrupt Service Routines, of which an example application is already provided with the RCC package.

Appendices

A LEON 3 Configuration

LEON 3 Configuration:

- Synthesis
  - Target technology: SAED32
  - Memory Library: SAED32
  - Infer RAM: NO
  - Infer pads: NO
  - Disable asynchronous reset: YES
  - Enable scan support: NO
  - Enable JTAG boundary scan: NO
- Clock generation:
  - Clock generator: SAED32-PLL
  - Use PCI clock as system clock: NO
- Processor:
  - Enable LEON3 SPARC V8 Processor: YES
  - Number of processors: 1
  - Integer unit:
    - SPARC register window: 8
    - SPARC V8 MUL/DIV instructions: YES
    - Hardware multiplier latency: 5-cycles
    - SPARC V8e SMAC/UMAC instructions: YES
    - Multiplier structure: Inferred
    - Branch prediction: YES
    - Single-vector trapping: YES
    - Load delay: 1
    - Hardware watchpoints: 2

    * Enable power-down mode: YES
    * Reset start address (addr[31:12]): 00000
    * SPARC V8E non-privileged ASI access: YES
    * SPARC V8E partial Write %psr (WRPSR): NO
    * SPARC V8E AWP and register file partitioning: NO
    * Enable LEON-REX extension: NO
- Floating-point unit:
    * Enable FPU: NO
- Cache system:
    * Enable instruction cache: YES
    * Associativity (sets): 2
    * Way size (kbytes/way): 4
    * Line size (bytes/line): 16
    * Replacement algorithm: Random
    * Cache locking: NO
    * Enable data cache: YES
    * Associativity (sets): 2
    * Way size (kbytes/way): 4
    * Line size (bytes/line): 16
    * Replacement algorithm: Random
    * Cache locking: NO
    * AHB snooping: YES
    * Separate physical/snoop tag: YES
    * Use SP RAM for separate tags: NO
    * Fixed cacheability map: 0
- MMU
    * Enable MMU: YES
    * MMU type: Split
    * TLB replacement scheme: LRU
    * Instruction (or combined) TLB entries: 8
    * Data TLB entries: 8
    * Fast writebuffer: YES

        * MMU page size: 4K
    - Debug Support Unit
        * Enable LEON3 Debug support unit: YES
        * Instruction trace buffer: YES
        * Instruction trace buffer size (kbytes): 4
        * Enable two-port instruction trace buffer: NO
        * AHB trace buffer: NO
        * Enable LEON3 Statistics Module: NO
    - Fault-tolerance: this feature isn't supported on the free version of the LEON3
    - VHDL debug settings:
        * Processor disassembly to console: YES
        * Processor disassembly in netlist: NO
        * 32-bit program counters: YES
- AMBA configuration:
    - Default AHB master: 0
    - Round-robin arbiter: YES
    - AHB split-transaction support: NO
    - Enable full plug&play decoding: NO
    - I/O area start address (haddr[31:20]): FFF
    - AHB/APB bridge address (haddr[31:20]): 800
    - Enable AMBA AHB monitor: NO
    - Write trace to simulation console: NO
- Debug Link:
    - Serial Debug Link (RS232): NO
    - JTAG Debug Link: YES
    - Ethernet Debug Communication Link (EDCL): NO
- Memory controllers:
    - LEON2 memory controller:

        * Enable LEON2 memory controller: YES
        * 8-bit PROM/SRAM bus support: YES
        * 16-bit PROM/SRAM bus support: NO
        * 5th SRAM chip-select: NO
        * SDRAM controller: YES
        * Separate address and data buses: NO
        * Enable page burst operation: NO
    - Enable AHB Status Register: YES
    - Number of correctable-error slaves: 1
- Peripherals:
    - Spacewire:
        * Enable Spacewire links: NO
    - Ethernet:
        * Gaisler Research 10/100/1000 Mbit Ethernet MAC: YES
        * Enable 1000 Mbit support: NO
        * AHB FIFO size (words): 8
    - SPI:
        * SPI memory controller:
            · Enable SPI memory controller: NO
        * SPI controller(s):
            · Enable SPI controller(s): YES
            · Number of SPI controllers: 1
            · Slave select lines: 6
            · FIFO depth (2N): 4
            · Enable slave select registers: YES
            · Enable automatic slave select: NO
            · Support automated transfers: NO
            · Support open drain mode: NO
            · Support three-wire protocol: NO
            · Maximum supported word length: 0 (up to 32-bit)
            · Use SYNCRAM for rx and tx queues: NO

        - SPI protocols: Standard
    - CAN:
        - Enable multi-core CAN interface: YES
        - Number of CAN cores: 1
        - CAN I/O area start address (haddr[19:8]): C00
        - Interrupt number: 13
        - Enable separate interrupts: NO
        - Enable synchronous reset: NO
        - Enable FT FIFO memory: NO
    - UARTs, timers and irq control:
        - Enable console UART: YES
        - UART1 FIFO depth: 4
        - Enable secondary UART: YES
        - UART2 FIFO depth: 4
        - Enable LEON3 interrupt controller: YES
        - Enable secondary interrupts: NO
        - Enable Timer Unit: YES
        - Number of timers (1 - 7): 4
        - Scaler width (2 -16): 12
        - Timer width (2 - 32): 32
        - Timer unit interrupt: 8
        - Separate interrupts: NO
        - Watchdog enable: YES
        - Initial watchdog time-out value: FFFFF
        - Enable generic GPIO port: YES
        - GPIO width: 16
        - GPIO interrupt mask: FE
        - Enable I2C master: YES
- VHDL Debugging:
    - Accelerated UART tracing: NO

B Difference between sequential elements

In this appendix, it is shown the difference in complexity between the two interfaces the original `SYNOPSYS_BASIC_SEQUENTIAL_ELEMENT` and the modified `SIMPLIFIED_SEQUENTIAL_ELEMENT`.

```
entity SYNOPSYS_BASIC_SEQUENTIAL_ELEMENT is
  generic (
    ac_as_q  : integer;
    ac_as_qn : integer;
    sc_ss_q  : integer
  );
  port(
    clear       : in std_logic;
    preset      : in std_logic;
    enable      : in std_logic;
    data_in     : in std_logic;
    synch_clear  : in std_logic;
    synch_preset : in std_logic;
    synch_toggle : in std_logic;
    synch_enable : in std_logic;
    next_state  : in std_logic;
    clocked_on  : in std_logic;
    Q           : buffer std_logic;
    QN          : buffer std_logic
  );
end SYNOPSYS_BASIC_SEQUENTIAL_ELEMENT;
```

```
entity SIMPLIFIED_SEQUENTIAL_ELEMENT is
  generic (
    ac_as_q  : integer;
    ac_as_qn : integer;
    sc_ss_q  : integer
  );
  port(
    synch_enable : in std_logic;
    next_state  : in std_logic;
    clocked_on  : in std_logic;
    Q           : buffer std_logic;
    QN          : buffer std_logic
  );
end SIMPLIFIED_SEQUENTIAL_ELEMENT;
```

Acronyms

ADC Analog to Digital Converter. 11

AHB Advanced High-performance Bus. 13

AMBA Advanced Microcontroller Bus Architecture. 13

APB Advanced Peripheral Bus. 13

CAN Controller Area Network. 11, 13

DAC Digital to Analog Converter. 11

DMA Direct Memory Access. 13

FPGA Field Programmable Gate Array. 12

GCC GNU Compiler Collection. 15

GNU GNU's Not Unix. 11

GPIO General Purpose Input/Output. 11, 13

GPL GNU General Public Licence. 11

HDL Hardware Description Language. 11

I^2C Inter-Integrated Circuit. 11, 13

IC Integrated Circuit. 5

IP Intellectual Property. 11

ISA Instruction Set Architecture. 11

LGPL GNU Lesser General Public Licence. 11

MMU Memory Management Unit. 13

ORPSoC OpenRISC Reference Platform System-on-Chip. 11

RCC RTEMS Cross-Compiler. 17

RTEMS Real-Time Executive for Multiprocessor Systems. 11, 14, 15

RTOS Real-Time Operating System. 11, 15

SoC System-on-Chip. 11–14

SPI Serial Peripheral Interface. 11, 13

Bibliography

[1] Cobham Gaisler. (2018). LEON/GRLIB download page, [Online]. Available: https://www.gaisler.com/index.php/downloads/leongrlib (visited on 11/18/2018).

[2] Cobham Gaisler, *LEON/GRLIB Guide: GRLIB IP Library User's Manual*, Nov. 2017. [Online]. Available: http://www.gaisler.com/products/grlib/grlib.pdf (visited on 12/29/2017).

[3] Cobham Gaisler, *LEON/GRLIB Guide: GRLIB IP Core User's Manual*, Dec. 2017. [Online]. Available: http://www.gaisler.com/products/grlib/grip.pdf (visited on 12/29/2017).

[4] SPARC International Inc., *The SPARC Architecture Manual: Version 8*, 1992. [Online]. Available: https://www.gaisler.com/doc/sparcv8.pdf (visited on 12/02/2018).

[5] Cobham Gaisler, *Bare-C Cross-Compiler: User manual*, 2018. [Online]. Available: https://www.gaisler.com/doc/bcc2.pdf (visited on 12/02/2018).

[6] Cobham Gaisler, *RTEMS Cross Compiler: User manual*, 2018. [Online]. Available: https://www.gaisler.com/anonftp/rcc/doc/rcc-1.2.pdf (visited on 12/02/2018).

[7] Wikipedia. (Jan. 9, 2018). OpenRISC, [Online]. Available: https://en.wikipedia.org/wiki/OpenRISC (visited on 01/28/2018).

[8] O. Kindgren. (Dec. 28, 2017). FuseSoC GitHub page, [Online]. Available: https://github.com/olofk/fusesoc (visited on 01/04/2018).

[9] M-Labs. (Nov. 11, 2018). MiSoC GitHub page, [Online]. Available: https://github.com/m-labs/misoc (visited on 12/01/2018).

[10] A. Traber and M. Gautschi, *PULPino Datasheet*, Jun. 9, 2017. [Online]. Available: http://www.pulp-platform.org/wp-content/uploads/2017/08/datasheet.pdf (visited on 02/04/2018).

[11] L. Azuara and P. Kiatisevi, "Design of an Audio Player as System-on-a-Chip," Master's thesis, University of Stuttgart, Jul. 2002. [Online]. Available: http://oggonachip.sourceforge.net (visited on 12/29/2017).

[12] L. Azuara, R. Dorsch, P. Kiatisevi, and H. J. Wunderlich, "Design of an Audio Player as System-on-a-Chip using an Open Source Platform," in *IEEE International Symposium on Circuits and Systems*, (May 23–26, 2005), Kobe, Japan: IEEE, Jul. 25, 2005, ISBN: 0-7803-8834-8. DOI: 10.1109/ISCAS.2005.1465242.

[13] Wikipedia. (Sep. 26, 2017). RTEMS Wikipedia page, [Online]. Available: https://en.wikipedia.org/wiki/RTEMS (visited on 01/05/2018).

[14] Mentor Graphics, *ModelSim PE User's Manual: Software version 10.2g*, 2015. [Online]. Available: https://documentation.mentor.com/en/docs/201508002/modelsim_pe_user/pdf (visited on 11/17/2018).