**Chapter 5**

# System verification, testing, and results

The architecture outlined in the design and development section has been carefully tested to ensure that the user logic prototype works as expected. This chapter elaborates on the methodologies that were used to perform the verification, testing, and resulting conclusions.

The chapter opens with a discussion on the verification work done on the user logic design through the use of software simulations. Methodologies in the design process to reduce the probability of bugs appearing in the design are also discussed. The chapter continues with a description of the acquisition system that was designed for the hardware tests and verifications. The focus is then moved to hardware device validation and testing.

## 5.1 Functional verification

Functional verification is the most important part of the design, it allows to verify that the system operates as expected. The moment the firmware is loaded on the FPGA, it can be very difficult to determine the cause of any unexpected issues, as there are no simple ways to observe the internal operation of an FPGA, unless specific debugging features have been implemented for this purpose.

A testbench code is create to encapsulates the design-part to be tested. Its purpose is to direct the overall testing effort, as well as to provide stimulation to the device under test, and to verify automatically the correctness of its response. It is essential to have tests that verify the complete operation of a design, including all corner cases, to minimize the risk of discovering issues after implementation. Code/test coverage tracking helps in this regard, where the tool collects information on which statements have been executed in the code, which branches have been taken, which states that have been covered or transitioned between, and which bits in a register have been toggled. Code coverage only verifies the correctness of what has been implemented and can be generally considered a quantitative measure of the design code, but it does not necessarily verify that all the specified functionality has been implemented or tested. The design intent from the specifications must, in addition, be formulated as testable statements that can direct the testing to verify that all features have been covered, generally referred to as functional driven verification.

To detect illegal transactions and signalling in the code, the design uses assertions, which are small pieces of simulation code inserted in the design code that reports when certain conditions occur. The benefit of including assertions is that they report both when testing a module standalone and when it is tested together with other modules as part of a bigger hierarchy. With assertions, interfaces between modules can more easily be verified when testing a bigger hierarchy, and the origin of the error is easier to pinpoint without the need to trace an error from a top-level output back to its point of origin.

The testbenches in this design use a combination of directed testing, where the specific design intent and specifications are verified, and random testing, where the input stimulus is randomized to exercise

|  |  |
| --- | --- |
| 2 | System verification, testing and results |

the bulk of the code coverage metrics. Testing the complete behaviour of a sub-module from the topmost level of the design hierarchy is often complicated by both the limited set of input controls available from the top level and the increased simulation time that is required to test the larger amount of code. Due to this, it is generally preferable to test the sub-module in isolation as well. However, since testing in isolation might not capture all the intricacies of the inter-module communications in the design, it is still important to try to verify as much as possible from the top level, particularly since the top-level testbench is used for verification of the post place and route code including timing verification. For the v2 design, the top-level testing was primarily done using functional directed testing, but more randomized tests where added for v3 as issues were discovered during hardware testing of v2 that were not caught during simulation, because the functional coverage was not exhaustive or high enough.

### 4.1.1 Module-based testbenches

Several module-based testbenches have been created to verify the functionality of most of the user logic modules.

Multi clock generator

TTC data generator

GBT data generator

Data extraction

Zero suppression

Data reformat

Data transmission

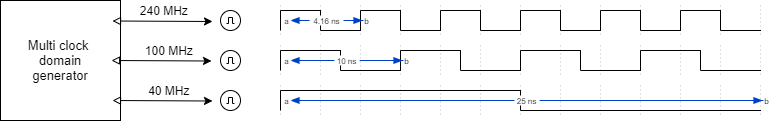
10x E-link checker

10x Reformat checker

|  |  |
| --- | --- |
| 2 | System verification, testing and results |

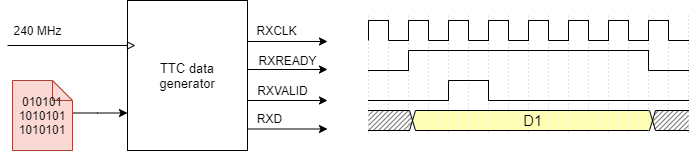
#### 5.1.1.1 Multi clock generator

Due to the difference in clock domains from different systems interfacing with user logic, a multi clock generator sub-module is designed to drive the simulation by providing clock signals to different sub-modules of the testbench. As shown in the figure below, the clock generator provides three different clock signals such as: the TTC clock domain (240MHz), the avalon MM clock domain (100 MHz) and the GBT clock domain (40MHz).



#### 5.1.1.2 TTC data generator

The TTC data generator sub-module emulates the TTC protocol described in chapter 3. Additionally, it generates trigger data extracted from a text file that contains real data collected during an acquisition. This sub-module is designed to operate in both continuous and triggered modes. The scenario is described in Figure 1.



#### 5.1.1.3 GBT data generator

#### 5.1.1.4 Data extraction verification

#### 5.1.1.5 Zero suppression verification

#### 5.1.1.6 Data reformat verification

#### 4.1.1.7 Data transmission verification

### 4.1.2 System-based testbenches

The testbench for the top-level design is sketched in figure 4.3. The testbench can be set up for testing of a single SAMPA or for testing of two SAMPAs in daisy-chained mode by changing a parameter at compile time. When testing only a single SAMPA, the boxes marked in grey are excluded from the simulation to increase the simulation speed. By changing another parameter at compile time, it is possible to run the simulations with gate level code, both with or without back annotated timing. A wrapper encapsulates the SAMPA code to model some of the analogue behaviours, like for instance the I 2C tristate driver and the SLVS enable signals.

## 4.2 Hardware verification

To verify the various performance specification of the SAMPA, a continuous Data Acquisition (DAQ) system has been developed. The FPGA-based DAQ system is specially designed for the qualification of the SAMPA and is designed around an Altera SocKit [76] evaluation board containing an Altera Cyclone V System-on- Chip FPGA [77], which has a built-in dual-core ARM1 Cortex-A9 microprocessor unit. The SAMPA is mounted on a separate mezzanine board connected to the development board through a high-speed connector. Control of the SAMPA and the FPGA board is handled through a Universal Asynchronous Receiver/ Transmitter (UART) connection from the controlling computer. Data packets from the SAMPA are acquired by the FPGA and transmitted verbatim to the readout computer via Gigabit Ethernet. A block diagram of the system design is shown in figure 4.4.

The intention of the DAQ system is to provide an easily customizable, standalone platform that is both compact and easily deployable to multiple testing groups for analogue behaviour qualification, digital verification, testing of the radiation tolerance and mass production testing. For the radiation tolerance testing, the intention is to verify the device performance in regards to SEUs, SELs, and functional interrupts. For the mass production, the goal is to filter out bad devi-ces before mounting them on front-end cards. DAQ systems and testbenches for the intention of qualifying of ASICs are commonly designed for a single or a limited set of tests. The system presented here is on the other hand designed as a common versatile platform and framework to interface with the SAMPA ASIC for a multitude of tests.

### 4.2.1 User logic integration

### 4.2.2 FPGA firmware compilation

The FPGA compilation is performed

### 4.2.3 Timing analysis

### 4.2.4 Readout electronics configuration

#### 4.2.5 CRU configuration

|  |  |
| --- | --- |
| 2 | System verification, testing and results |

#### 4.2.6 GBTx programmer

#### 4.2.7 LTU configuration

### 4.2.8 DAQ

### 4.2.9 Data collection

## 4.3 Results