High-Quality and High-Throughput Random Number Generation in FPGAs

Student
Cristian KLEIN

Supervisor
Conf. Dr. Ing. Octavian CRET

Cluj-Napoca
June, 2008
High-Quality and High-Throughput Random Number Generation in FPGAs

Absolvent: Cristian KLEIN

- Continutul proiectului: Introduction, Context, Implementation, Results and Conclusions
- Locul documentației: École normale supérieure de Lyon
- Consultanti: Conf. Dr. Ing. Octavian CRET
- Data emiterii temei: 20.02.2008
- Termen de predare: 20.06.2008
# Contents

1 Introduction ................................................. 8  
   1.1 Theme of this thesis .................................... 8  
   1.2 Field-Programmable Gate Array ............................... 8  
      1.2.1 Structure of the FPGAs ................................. 9  
      1.2.2 FPGA Software Development ........................... 11  
   1.3 Random Numbers ........................................... 12  
      1.3.1 Pseudo-Random Numbers Generators .................... 12  
      1.3.2 True-Random Number Generators ....................... 13  
      1.3.3 Test Batteries ........................................ 15  
      1.3.4 Normal Random Number Generators .................... 15  

2 Context .................................................. 18  
   2.1 Random-Number Test Batteries .............................. 18  
      2.1.1 DieHard .............................................. 18  
      2.1.2 TestU01 ............................................... 18  
   2.2 TRNGs in FPGAs ........................................... 21  
      2.2.1 Digital Randomness in FPGA ........................... 21  
      2.2.2 RO-based Jitter Sampling ............................. 22  
      2.2.3 Fixed-Frequency Jitter Sampling ....................... 23  
   2.3 Integer Arithmetic ........................................ 24  
      2.3.1 GNU Multiple Precision ................................. 24  
   2.4 Floating-Point Arithmetic .................................. 25  
      2.4.1 Floating-Point Numbers ................................. 25  
      2.4.2 Rounding Modes ...................................... 27  
      2.4.3 Arithmetic Operations ................................ 28  
      2.4.4 MPFR ................................................. 29  
      2.4.5 Minimax Polynomial ............................ 31  
      2.4.6 Sollya ................................................. 31  
      2.4.7 HOTBM .............................................. 32
## CONTENTS

2.5 FloPoCo ........................................ 35
   2.5.1 Signal Class .................................. 36
   2.5.2 Operator Hierarchy .............................. 38
   2.5.3 Target Hierarchy ................................. 38

3 Implementation .................................... 39
   3.1 TRNG ........................................... 39
      3.1.1 Creating ROs in VHDL .......................... 39
      3.1.2 RO-based Jitter Sampling ....................... 40
      3.1.3 Fixed-Frequency Jitter Sampling ............... 42
      3.1.4 High-Bandwidth Measurement ................. 43
      3.1.5 Speeding up the TRNG ......................... 44
   3.2 FloPoCo ......................................... 46
      3.2.1 FloFP ....................................... 46
      3.2.2 Automated Test Bench Generation ............... 50
      3.2.3 HOTBM Operator ................................. 54
      3.2.4 FPLog Operator ................................ 55

4 Results and Discussions ....................... 56
   4.1 TRNG ........................................... 56
      4.1.1 RO-based Jitter Sampling ....................... 56
      4.1.2 Fixed-Frequency Jitter Sampling ............... 56
   4.2 High-Speed TRNG ................................ 58

5 Conclusions .................................... 59
## List of Figures

1.1 Structure of an FPGA .................................................. 10
1.2 Structure of a Configurable Logic Block (CLB) .................. 10
1.3 Probability Distribution Function for Normally Distributed Functions .......... 16
1.4 Diagram of the Box Muller Transform .............................. 16

2.1 Example C Code Illustrating the Usage of the Rabbit Battery. .......... 20
2.2 The Jitter Model ...................................................... 21
2.3 Ring Oscillator (RO) .................................................. 22
2.4 RO-based Jitter Sampler .............................................. 22
2.5 RO-based Jitter Sampling Cycle ..................................... 23
2.6 Fixed-Frequency Jitter Sampling Working Principle .................. 24
2.7 Example of Using the GMP \texttt{mpz\_class} class .................... 26
2.8 Floating-point numbers stored as in IEEE 754 for single-precision (SP) and double-precision (DP) .................................................. 27
2.9 Floating-point Rounding Modes ...................................... 28
2.10 Corner case, when rounding to nearest is impossible due to error boundary. .. 29
2.11 Example of using \texttt{libsollya} ...................................... 33
2.12 Architecture of a HOTBM Generated Circuit ....................... 34
2.13 FloPoCo’s Floating-Point format .................................... 37
2.14 Part of FloPoCo’s Operator Hierarchy ............................... 37

3.1 Using the VHDL „KEEP“ attribute .................................. 40
3.2 Variants of Waveforms for the \texttt{R0} and \texttt{E0} Signals we tried in our control .. 41
3.3 TRNG Scheme ......................................................... 42
3.4 High-throughput Measurement Scheme ................................ 43
3.5 FSM State Diagram ................................................... 45
3.6 High-Speed TRNG ..................................................... 46
3.7 Conversion to \texttt{mpfr\_t} ............................................. 48
3.8 Conversion from \texttt{mpfr\_t} ......................................... 49
3.9 Example test case interface for an operator .......................... 53
List of Tables

1.1 Output stream improvement using the simple XOR method ........................................ 14
2.1 Values of the exception field and their meaning ....................................................... 37
4.1 Parameters for High Quality TRNG ........................................................................ 58
List of Abbreviations

CLB ............ Configurable Logic Block, page 9
FF ............... Flip-flop, page 9
FP ............... Floating-point, page 25
FPGA ............ Field-Programmable Gate Array, page 8
LUT .............. Lookup Table, page 9
PRNG ............ Pseudo-Random Number Generator, page 13
RNG ............ Random Number Generator, page 12
RO .............. Ring Oscillator, page 22
TRNG ............ True-Random Number Generator, page 13
VHDL ............ Very-High-Speed Integrated Circuits Hardware Description Language, page 11
Chapter 1

Introduction

„Random numbers should not be generated with a method chosen at random.”

Donald E. KNUTH

1.1 Theme of this thesis

This thesis proposes to study the implementation of random-number generators in FPGAs.

On one hand, it will emphasise on implementing a pure-digital TRNG in FPGA. In contrast to using external components, such as analog-to-digital converters to sample some kind of noise, we wanted to research the quite novel domain of generating noise inside the FPGA itself. This subject belongs to the CryptoRand project lead by Conf. Dr. Alin SUCIU. Part of this work constitutes the subject of a scientific paper that has been accepted for publication at the ICCP 2008 conference.

On the other hand, it will present the FloPoCo project developed at ENS-Lyon by a team lead by Prof. Florent de Dinechin, its goal and our contributions to it. Implementing normally distributed random numbers on FPGAs with arbitrary precision is still a hot topic, especially if one wants to automate this task. The FloPoCo project aims at creating various standard and non-standard arithmetic operators for FPGA, which will ease the task of generating uniform-to-normal probability distribution converters.

1.2 Field-Programmable Gate Array

Field-Programmable Gate Array (FPGA) are general-purpose integrated circuits with a large number of programmable logic blocks and interconnection grid. The programmable logic blocks contain either simple gates like AND, OR, XOR, or complex lookup tables and flip-flops. The interconnection grid can arbitrarily connect the inputs and outputs of the programmable logic, thus, suppose it has enough resources, an FPGA can implement any digital circuit.
FPGAs are often compared with application-specific integrated circuits (ASIC). The latter are integrated circuits which are designed and built for a single, particular use. FPGAs have the advantage that they are reconfigurable, which allow fixing bugs, and applying updates at any time. However, this flexibility comes at a high price, FPGAs being two orders of magnitude slower than ASICs implemented using the same technology.

1.2.1 Structure of the FPGAs

While the structure of FPGAs are in principle the same, when going into details they greatly differ from vendor to vendor, or from one family to the other. The following are the elements typically found in Xilinx FPGAs (figure 1.1):

**Configurable Logic Block (CLB)** are programmable logic blocks, which contain an \( n \)-input lookup table (LUT) whose output is connected to a storage element (figure 1.2). The LUT is completely configurable and can implement any \( n \)-bit boolean function. Various settings can be applied to the storage element, such as whether it should behave like a flip-flop (FF) or a latch, whether the reset should be synchronous or asynchronous, whether it should be initialized to „0‟ or „1‟.

The Xilinx Spartan3E FPGAs (XC3S500) has CLBs with 4-inputs (\( n = 4 \)), each two being packed into a slice. The total number of slices is around 10,000.

**Local switching boxes** make the connection between adjacent CLBs or between CLBs and the long-distance routing wires. They introduce small propagation delays.

**Long-distance routing wires** make the connection between distant CLBs. They are responsible for the biggest part of the delay in an FPGA.

**Switching boxes** connect long-distance routing wires, in order to complete the interconnection between CLBs.

**BlockRAMs** are included in FPGAs in order to save CLBs, as RAM is a very common FPGA construction.

**DSP blocks** (Digital Signal Processing blocks) accelerate fixed-width integer addition and multiplication, while at the same time reducing the number of CLBs required for implementing arithmetic on the FPGA.
1.2. FIELD-PROGRAMMABLE GATE ARRAY

CHAPTER 1. INTRODUCTION

Figure 1.1: Structure of an FPGA

Figure 1.2: Structure of a Configurable Logic Block (CLB)
1.2. FIELD-PROGRAMMABLE GATE ARRAY

1.2.2 FPGA Software Development

The development of software for FPGA differs a lot from microprocessor software development. The source code consists in one or more hardware modules organised in a hierarchical manner. Each such module can either be described by drawing their scheme, or by using a hardware description language such as VHDL or VeriLog. Hardware description languages offer the following advantages:

- Modules can be described structurally, as a composition of elementary modules, or behaviourally, highlighting what they do and not how they do it.
- Components can be programmed as generic entities, which means that their whole implementation depends on a compile-time parameter. For example, a single VHDL module can implement a shift register for any width.

In order to be loaded on a FPGA, the source code has to go through the following stage:

**Synthesis** This stage consists in reading the VHDL description of the circuit in order to extract all its logic structure. If the components of the circuit are described behaviourally, it is the task of the synthesis tool to deduce the necessary gates for realising the specified behaviours. This feature is very complicated and for many behavioural constructs hard-to-solve or even impossible.

Another role of the synthesiser is to optimise the logic of the circuit, that is, find the minimal number of gates which correctly implement the boolean functions of the circuit. Finally, the synthesiser produces a netlist, which consists in a list of logic elements and the connection between them.

**Mapping** The previously generated netlist of the circuit is adapted to the logical primitives available on the target FPGA. This stage converts the generic structural description of the netlist into a specific elements available on the target FPGA. The base elements are not anymore logic gates, but configurable logic cells of the FPGA.

**Placement and Routing** The mapping has identified the necessary FPGA resources to implement the circuit, so the task of the placement and routing stage is to find the concrete locations on the surface of the FPGA and connect these resources together. It is possible to constrain the routing to specific timing requirements, or fix the location of certain elements. For example, one may specify the input / output pin assigned to certain signals inside the circuit. It is even possible to manually place or route all or some part of the FPGA. There routing constraints are usually stored in a user constraint file (UCF).
After this phase the number of required slices and the maximum delay of certain signals is known. Based on the latter, one may compute the maximum frequency at which the circuit implemented on the FPGA may work error-free.

**Configuration** In this phase, a so-called *bitstream* is generated, which is the binary blob understood by the FPGA. It is generated in such a way that all logic of the FPGA is configured according to the results of the placement and routing phase.

The last thing to do is to upload the bitstream into the FPGA. This is usually done using a serial interface such as JTAG. The FPGA will immediately change its behaviour to the one we wanted to implement. Most FPGAs are volatile by nature, so if one wants the FPGA to retain its behaviour after a power loss, one has to load the bitstream to the Flash memory located on the same board as the FPGA.

If one wants to test a certain hardware design, it is not necessary to do it on the FPGA itself. There are many simulation tools which are able to compile a VHDL design and graph the values of the signals on the time axis. Simulation can either be done on the initial code source, on the post-mapped circuit, or on the post-placed-and-routed model.

In order to automate testing, one can generate a so-called *test bench*, which is a waveform file or a VHDL file, which assigns values to the input of the *unit under test* (UUT) and verifies the outputs.

### 1.3 Random Numbers

A random number sequence consist of number for which the following statements hold:

- They respect a given distribution, usually a uniform distribution.
- They are unpredictable, that is, the probability that a given number will be next is given only by the above chosen distribution.

Starting from the uniform distribution, one may generate any desired distribution, therefore we will restrict our discussion to these types of random numbers.

Random number generators (RNG) are programs or hardware devices that output random numbers. They can be of two types: *pseudo-random number generators* and *true-random number generators*.

#### 1.3.1 Pseudo-Random Numbers Generators

*Pseudo-random number generators* (PRNG) are generators for which a person who designed the system, or has access to its internal state can predict the next random number. The system
1.3. RANDOM NUMBERS

is a deterministic Finite State Machine, whose evolution can usually be described based on an arithmetic formula which determines its transition from a given internal state to another state, while outputting a random number based on a portion of the state. All PRNGs require an initial state (also called seed), which determines the sequence of numbers which will be generated.

Truthfully speaking, PRNG are not random number generators. As von Neumann would put it

Whoever considers arithmetic for random number generation is in a state of sin.

However, PRNGs have statistical properties which make their output indistinguishable from a random-number generator.

Regarding the predictability of their output, for some applications, such as for Monte-Carlo simulations [1], it is desirable for reproducibility and debugging reasons to be able to generate the same sequence of random numbers as often as one wants. Due to the large quantity of random numbers used, storing the random numbers is often unfeasible. Instead, a pseudo-random number generator is employed, for which only the seed is stored.

Other random number generators are hard-to-predict and even considered cryptographically secure (such as Blum Blum Shub [2]), assuming that they are well seeded. The importance of well seeding a pseudo-random number generator has recently been highlighted in a Debian security vulnerability [3].

In both applications, PRNGs are preferred because they have a very high bandwidth.

1.3.2 True-Random Number Generators

True-random number generators (TRNG) are generators whose output cannot be predicted, not even by the person who designed them. They are based either:

- on a microscopic event which is thought to be random, such as thermal noise [4], atmospheric noise [5], atomic decay [6], photoelectric effect or other quantum phenomena
- on a macroscopic phenomenon such as rolling dices, playing cards, etc.

Although macroscopic methods are deterministic, that is, they precisely obey Newtonian laws, chaos theory suggest that the final state of the system is greatly influenced by the initial microscopic details, which are impossible to determine in practice.

Because of their unpredictability, one would be tempted to use only TRNGs in cryptography. This is however often impossible due to their smaller throughput. Therefore they are often used to seed cryptographically secure PRNGs. Many operating system harvest entropy from sources such as network activity and disk activity[7][8]. Although they are not completely unpredictable, due to the complexity of the computer, it is impossible for an adversary to gain any usable information, especially if he hasn’t physical access to the computer.
1.3. RANDOM NUMBERS

CHAPTER 1. INTRODUCTION

Unbiasing TRNGs

Taking their output as a stream of bits, TRNGs are often biased, that is, although they are unpredictable, the probability of outputting a „0” is different than that of outputting a „1”. Two methods are well known for correcting this problem:

Neumann method consists in looking at consecutive bits. If the two bits are equal, the bit pair is dropped. If they differ, only the first bit is retained. Suppose the probability of „0” is \( p \), obviously the probability of „1” will be \( 1 - p \). After processing, the probability of „0” becomes \( p'_0 = p \cdot (1 - p) \) (the probability of the first bit being „0”, multiplied by the probability of the second bit being „1”), while the probability of „1” becomes \( p'_1 = (1 - p) \cdot p \) (the probability of the first bit being „1”, multiplied by the probability of the second bit being „0”). We observe that the two probabilities are equal.

While the Neumann method guarantees that the output stream will have the same probability for „0” as for „1”, it reduces the bandwidth of the system depending on how biased the initial TRNG was. In the best case, when the two probabilities are equal from start, the bandwidth is halved by the fact that out of the 4 combination of the input bits, only 2 are accepted. The bandwidth is halved again, because out of the pair of bits that is accepted, only one bit is outputted. Therefore, the output bandwidth is at best 25% of the input bandwidth.

Simple XOR consists in taking two consecutive bits and XOR-ing them. Assuming the same notation as above, after one round of XOR, the new probabilities will become:

\[
p'_0 = p \cdot p + (1 - p) \cdot (1 - p) = 2 \cdot p^2 - 2 \cdot p + 1
\]

\[
p'_1 = p \cdot (1 - p) + (1 - p) \cdot p = 2 \cdot p - 2 \cdot p^2
\]

Although the probabilities are not equal, we can observe by taking experimental values that this processing greatly improves the random stream (table 1.1). The advantage of this method is that each time it only halves the bandwidth.

<table>
<thead>
<tr>
<th>( p )</th>
<th>( p'_0 )</th>
<th>( p'_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.55</td>
<td>0.505</td>
<td>0.495</td>
</tr>
<tr>
<td>0.54</td>
<td>0.5032</td>
<td>0.4968</td>
</tr>
<tr>
<td>0.53</td>
<td>0.5018</td>
<td>0.4982</td>
</tr>
<tr>
<td>0.52</td>
<td>0.5008</td>
<td>0.4992</td>
</tr>
<tr>
<td>0.51</td>
<td>0.5002</td>
<td>0.4998</td>
</tr>
</tbody>
</table>

Table 1.1: Output stream improvement using the simple XOR method.
1.3.3 Test Batteries

It is impossible to absolutely guarantee that a stream is random. Instead, one can increase the confidence that a certain stream is random, by passing it through many tests of statistical randomness. In principle, the random numbers are the input to some arithmetic processing, which, on the assumption that the numbers are random, should output a value contained in a certain interval. The output value is then normalised between 0 and 1 (assigning 0 to lower-than-expected values and 1 to higher-than-expected values), thus obtaining the so-called p-value.

In order to understand the concept we will illustrate a few trivial but useful tests. These tests are in no way exhaustive. If these tests are passed, it doesn’t mean that the input stream was random, however, failing these tests does mean that the stream was not random. Non-trivial tests which are considered state-of-the-art and their usage will be presented in section 2.1. We will assume that the random numbers are generated as a stream of random bits.

**Bit Distribution** Counting the number of bits of „0” and „1”, we should obtain about the same numbers.

**n-bit Distribution** Grouping the stream into n-bits and counting each possible value, we should obtain the same numbers.

1.3.4 Normal Random Number Generators

Generally all random-number generators output uniformly distributed random-numbers, that is, the probability of the occurrence of each number is the same. In some applications, it is desirable to have normally distributed random-numbers, that is, their probability density function must be equal to:

\[
\frac{1}{\sigma \cdot \sqrt{2\pi}} \exp \left( -\frac{(x - \mu)^2}{2\sigma^2} \right)
\]

where \( \mu \) represents the mean and \( \sigma^2 \) the variance. The formed shape is often called the bell curve, due to its resemblance with a bell (figure 1.3).

**Acceptance-Rejection Methods**

The first class of methods to obtain a normal distribution out of a uniform one are the so-called rejection sampling methods or acceptance-rejection methods. The most used method from this class is the Ziggurat algorithm [9]. These methods, consist in choosing only certain values from the input random number sequence, so that after eliminating them, the random number sequence has a normal distribution.

These methods are often used because they have lower computational requirements. For example, the Ziggurat algorithm only requires computing the logarithm of a number, which is a
very costly computation, in 2.5% of the time. This computational luxury comes at the expense of reduced bandwidth.

We believe that this class of methods are not suitable for custom hardware architectures. First, the output numbers don’t arrive at a constant rate. This would require slowing down the computation that uses the random numbers and implementing a FIFO system so that random numbers are constantly available. Great care has to be taken not to fill up the FIFO as this would induce additional sampling, which might change the probability distribution function.

Secondly, since we have already wasted hardware resources on implementing the logarithm operator we could just as well use it all the time. Also, if the logarithm operator is pipelined, we would have to add delay registers for the short path, which would use additional hardware resources.

**Box-Muller Method**

The Box-Muller method transforms a pair of independent uniformly distributed random variables into a pair of independent normally distributed variables (figure 1.4). It does not reject any input random numbers, thus having the same output bandwidth as the input bandwidth, while
1.3. RANDOM NUMBERS

CHAPTER 1. INTRODUCTION

requiring more expensive computation.

Suppose $u_0$ and $u_1$ are two independently uniform random variables belonging to the interval $(0, 1]$. Let,

\[
\begin{align*}
  z_0 &= \sqrt{-2 \ln u_0} \cdot \sin(2\pi u_1) \\
  z_1 &= \sqrt{-2 \ln u_0} \cdot \cos(2\pi u_1)
\end{align*}
\]

then $z_0$ and $z_1$ are two independently normal random variables, with $\mu = 0$ and $\sigma^2 = 1$.

Implementing the Box-Muller method, requires square-root, logarithm, sine and cosine operators. While this method is not novel, it has only been recently put in practice on FPGAs. [10] presents an implementation of this algorithm on FPGAs and does a detailed error analysis, however, the output precision is fixed at 16 bits. The task of creating an arbitrary-precision Box-Muller transformation, with automatic precision augmentation and pipelining is still an open subject.
Chapter 2

Context

2.1 Random-Number Test Batteries

In order to validate the fact that our TRNG works well, we used two test batteries. In what follows we will describe their working principle and how to use them.

2.1.1 DieHard

DieHard [11] is an application for testing statistical randomness of a sequence of bytes written in ANSI C. Its input consists in a binary file which should have between 10MB and 12MB. It is very important to note that DieHard assumes each byte of the file is uniformly distributed in [0, 255]. Failure to comply with these requirements will return results as if the random number stream was of a poor-quality.

DieHard runs several tests (often in several iterations with slightly different parameters) and returns $p$-values for each one of them. $p$-values of 0 or 1 mean the test has failed. If several tests have failed, one may assume the random stream stored in the input file is of poor quality.

One of the biggest disadvantages of DieHard is the fact that it does not display a summary of all tests and displaying $p$-values is not consistent. Some values are written in a table, others inline, which makes filtering the output impossible. One has to manually scroll through the output of DieHard to count the number of $p$-values which are out-of-range.

2.1.2 TestU01

TestU01 [12] is a library written in C for generation and analysis of random numbers. The functions exported by the library are grouped in four modules: u, s, b, f.

Functions starting with „u” create uniform pseudo-random number generators. All fields relevant for a generator are stored in a unif01_Gen structure, which represents an instance
of a generator. A generator is supposed to implement two functions, GetU01, which returns a double-precision floating-point number uniformly distributed in $[0,1)$ and GetBits, which returns a 32-bit number uniformly distributed in the whole range.

The library includes a lot of example generators such as *Linear Congruent Generators* (LCGs), *multiple recursive generators* (MRGs). One may also implement his own generator, for example a generators which retrieves random number from an external hardware generator.

In our case, we wanted to be able to run multiple tests on the same random number sequence, so we used the ufile generator. The ufile_CreateReadBin function creates a generator which reads a binary file, just like DieHard.

**Functions starting with „s”** run statistical tests on the output of a previously created generator. Each such function outputs on stdout information such as:

- name of the generator
- host on which the generator ran
- name of the test
- parameters of the test
- results of the test

The quantity of information that each test outputs can be changed by setting variables from the swrite module.

There is also a special module called scatter which outputs LaTeX and GnuPlot code for generating a 2-dimensional scatter of the output of the generator. $N$ double-precision floating-point numbers in the interval $[0,1)$ are requested, which are given as coordinates to an $t$-dimensional hypercube. The dimensions of the hypercube are then arbitrarily projected on the $x$ and $y$ axis, optionally cropping them to $x \in [x_l, x_h]$ and $y \in [y_l, y_h]$. All the variable mentioned above may be customised. The scatter_PlotUnif function reads the parameters from a file, scatter_PlotUnif1 receives all the data in its parameters and scatter_PlotUnifInterac gets the data interactively from the user.

**Functions starting with „b”** run a series of tests (also called batteries) against a certain generators. The reason for having batteries is that the statistical tests receive many parameters, which often increase the computation time exponentially. The motivation behind batteries is to give the user an easy access to a set of tests (together with parameters) which allow the user to choose between thoroughness of the tests and time spend testing.

Another advantage of test batteries is that they output a final report, which includes the number of failed tests, the name of the tests that failed and their mean $p$-value. In TestU01 a test is considered passed if $p \in [0.001, 0.9990]$. 
#include <sys/types.h>
#include <sys/stat.h>

#include "unif01.h"
#include "bbattery.h"

int main(int argc, char *argv[]) {
    if (argc < 2)
        return 1;

    char *filename = argv[1];

    struct stat st;
    if (stat(filename, &st) == -1)
        return 1;

    bbattery_RabbitFile(argv[1], 8*st.st_size);
    return 0;
}

Figure 2.1: Example C Code Illustrating the Usage of the Rabbit Battery.

Some smaller test batteries have functions which directly receive as parameter a filename. Other, bigger tests require a very big quantity of random numbers and are not suitable for receiving their input from a file.

The batteries which we ran for our tests are alphabit (which is tailored for hardware RNG), rabbit (a fairly thorough test for any RNG) and NIST (which is equivalent with the NIST Random Number Test Suite). As TestU01 is a library and not a program, one has to write C or C++ to be able to use the above batteries. An example code is given in figure 2.1.

TestU01’s alphabit and rabbit test batteries are considered a lot better than DieHard, which has been confirmed by our experiments.

Function starting with „f” are used to systematically tests families of random number generators. They won’t be presented here as they are out of the scope of this thesis.
2.2 TRNGs in FPGAs

2.2.1 Digital Randomness in FPGA

In contrast with the ideas presented above, our intention was to create a design which would generate a stream of true-random numbers using only resources available on the FPGA. The clear advantage would be that every FPGA could be used out-of-the-box from applications requiring a true-random number source.

True-random numbers can obviously not be constructed without having a physical phenomenon which manifests a certain factor of uncertainty. In FPGAs jitter can appear in certain conditions, whose sampling can generate entropy.

In essence, due to various noise sources such as that induced by the power supply but also by nearby components, the behaviour of „demanding a 0 or 1” from the transition slope of an output is unpredictable. This is caused by the fact that each technology defines a low (L) threshold, which is the upper limit for voltages which represent a logic 0, and a high (H) threshold which is the lower limit of logic 1. Output behaviour between these two values is not well defined. This can be modelled as if the output of the component would have a perfectly vertical slope, but the time of the transition is unknown and can range from the beginning until the end of the real slope (figure 2.2).

In order to produce jitter, TRNGs employ one or more ring oscillators (RO) (figure 2.3). These are composed of a ring of odd number of inverting elements and an arbitrary number of delaying elements. The simplest RO is composed of a single inverter and a buffer. The output of a RO is never stable and does transitions from 0 to 1 and back to 0 with a frequency given by the propagation delay of the constituting elements. Due to the above described phenomena, the period of an oscillation will not be constant, because it will vary by a small amount each time.
This is the manifestation of jitter and the source of entropy which our TRNG will collect.

### 2.2.2 RO-based Jitter Sampling

This design is based on the idea presented in [13] and consists in two ROs each sampling the others output. The sampled RO is connected to the \( D \) pin of a flip-flop, while the sampling RO is connected to the „clk” pin (figure 2.4). Small differences in oscillation frequency and jitter will create uncertainty when counting the number of sampling RO cycles in which the output of the FF is „0” or „1” (\( S_0 \) signal in figure 2.5). A FF acting as a 1-bit counter, driven by the sampling RO, will store whether the last transition happened odd or even number of cycles ago (signal \( C_0 \)). This is the random bit which we are about to return to the user of the TRNG, after we stabilised it with another FF (\( \text{RandomBit} \) signal). Finally, will use one more FF to signal the user when a new random bit is available (\( \text{BitReady} \) signal). The user will have to set the \( \text{ReadAck} \) signal, when he consumed the previously returned random bit.
This design required a control unit, connected to the clkS, S0, R0 and E0 signals, which:

- resets the 1-bit counter after returning a random bit, to avoid correlations between consecutive random bits;
- disables the clock enable CE pin of the two output flip-flops, for few cycles after a random bit has been returned.

The latter is necessary, because jitter also appears in the sampling RO and therefore S0 might transition several times when the edges of clk0 and clkS are close. Unfortunately, the exact behaviour of the control unit is not fully specified in the paper which describes this TRNG. In section 3.1.2 we will describe several controller versions we have tried in our implementation.

Also, this design’s output is biased and must be post-processed using one of the methods described in section 1.3.2.

We chose to implement this design due to the fact that the whole stream before the post-processing phase is random (although it might be biased a little bit). We also favoured this design, because, if there is some kind of predictable jitter (such as coming from the power source), both ROs are influenced the same way which should cancel out at the sampler.

#### 2.2.3 Fixed-Frequency Jitter Sampling

We also chose to implement [14] which uses multiple ROs whose outputs are XOR-ed. A flip-flop whose clock is driven by a fixed frequency will sample the combined output of the ROs. The obtained stream will hit both jitter zones (our source of entropy) and flat zones (which are highly predictable) (figure 2.6). A post-processing phase is required which consists in a resilience function [15]. In essence, the function takes an m-bit input, out of which n-bits are known to be random (but we can’t determine which ones) and outputs n-bits which are known to be random. For n = 1, the simplest resilience function is to xor all the input bits. Suppose
all but one bits are deterministic, but the probability of a 0 or 1 value of one bit are equal, the output of the xor will also have equal probability of being 0 or 1.

2.3 Integer Arithmetic

2.3.1 GNU Multiple Precision

The GNU Multiple-Precision Library (GMP) [16] is a free library for arbitrary precision arithmetic. It operates on the following sets:

- signed integers ($\mathbb{Z}$)
- rational numbers ($\mathbb{Q}$)
- floating-point numbers (often denoted $\mathbb{F}$)

There is no practical limit to the precision except the ones implied by the available memory in the machine GMP runs on. GMP has a rich set of functions, and the functions have a regular interface.

GMP aims to be faster than any other bignum library for all operand sizes. Some important factors towards this end are:

- GMP uses the full word of the underlying machine to implements its operations.
- GMP uses different algorithms for different operand sizes. The algorithms that are fastest for really big numbers are seldom fastest for small numbers. For example, the Karatsuba algorithm is known to speed up multiplication only for very large numbers.
• Highly optimized assembly code for the most important inner loops, specialized for different processors.

For FloPoCo we used the signed integer function from GMP to store large signal values. GMP offers the advantage of having fast implementation for operations which are commonly used when simulating hardware modules, such as: addition, multiplication, shifting. The storage capacity of the result is automatically augmented to store the result without overflow.

In FloPoCo, we directly map the binary signal values to GMP integer, which allows us to use the negative range for special values, such as *don’t care* or *don’t know*.

Fortunately, as FloPoCo is written in C++, we can benefit from GMP++, a C++ wrapper around the GMP functions. The C++ library exports classes which overload operators directly mapped to GMP functions. This enables us to write arithmetic statements which use big integers without regarding the fact that they aren’t true integers.

An example code of how to use GMP++ is illustrated in figure 2.7). The code must be compiled using `g++ main.cpp -lgmpxx -lgmp`.

### 2.4 Floating-Point Arithmetic

#### 2.4.1 Floating-Point Numbers

A radix-\(b\) floating-point (FP) number \(x\) is a number of the form

\[
(-1)^s \times m \times b^e,
\]

where \(b\) is the *radix* of the floating-point system, \(m\) is the *significand* of \(x\), \(s\) is the *sign* and \(e\) is its *exponent*.

The usual way to store floating-point numbers on a computer is to keep \(m\) as a fixed-point number, \(e\) as a biased integer, \(s\) as a single bit, having \(b = 2\).

The most common representation for floating-point numbers is the IEEE 754 standard, which defines two formats: *single-precision* and *double-precision* (figure 2.8). Equation 2.1 and 2.2 give the formula of converting normalised IEEE single/double-precision numbers into the value they represent.

\[
x_{sp} = (-1)^s \times 2^{e-127} \times 1.f \tag{2.1}
\]

\[
x_{dp} = (-1)^s \times 2^{e-1023} \times 1.f \tag{2.2}
\]

Floating-point numbers are represented using a fixed number of digits and an exponent, thus allowing the fraction point to „float”. In contrast to *fixed-point* numbers, whose fraction point is in a fixed position, floating-point numbers have the following advantages:
```cpp
#include <streamio>
#include <gmpxx.h>

int main (int argc, char *argv[]) {
    mpz_class a, b, c;

    /* Convert to bigint */
    a = 1234;
    b = "-5678";

    /* Various arithmetics */
    c = a+b;
    cout << "sum is " << c << "\n";
    cout << "product is " << a * b << "\n";

    /* Common mathematic functions */
    cout << "absolute value is " << abs(c) << "\n";

    /* Shifting */
    cout << "c div 2 is " << (c >> 1) << "\n";

    /* Fits in 64 bits? */
    if (c < (mpz_class(1) << 64))
        cout << "fits in 64 bits\n";

    /* Get last 8 bits of the product*/
    cout << ((a*b) & 0xFF);

    /* Display base 3 representation */
    cout << a.get_str(3);

    return 0;
}
```

Figure 2.7: Example of Using the GMP `mpz_class` class
They have a much larger range.

- They can represent much smaller numbers.
- They can represent special values, such as $\pm 0$, $\pm \infty$ and $\pm NaN$ (not a number) – note that these numbers usually have special encoding.
- They have a guaranteed minimum relative error on a wide domain of values.

Floating-point numbers have been created to approximate real-number arithmetics, however, due to the fact that they are stored on a bounded storage space, they are not even able to represent rational numbers.

An interesting, yet annoying problem occurs when converting decimal numbers to floating-point numbers (which are usually not in base 10). Non-repeating decimal numbers must often be converted to repeating-digit numbers, when converting them to another base, and vice-versa. As floating-point numbers have limited storage, they cannot precisely store these numbers.

### 2.4.2 Rounding Modes

When converting real numbers to floating-point numbers, one may choose any of the following four rounding mode (figure 2.9):

- **Round toward $-\infty$:** $\nabla(x)$ is the largest floating-point number (possibly $-\infty$) less than or equal to $x$;
- **Round toward $+\infty$:** $\triangle(x)$ is the smallest floating-point number (possibly $+\infty$) greater than or equal to $x$;
- **Round toward zero:** $z(x)$ is the closest floating-point number to $x$ that is no greater in magnitude than $x$ (it is equal to $\nabla(x)$ if $x \geq 0$, and to $\triangle(x)$ if $x \leq 0$);
- **Round to nearest:** $\circ(x)$ is the floating-point number that is the closest to $x$. A tiebreaking rule must be chosen when $x$ falls exactly halfway between two consecutive floating-point numbers. The most common rule employed is to choose the floating-point number, for which the last bit in the mantissa is 0. This helps avoid systematic errors, as some errors tend to cancel out each other.

---

Figure 2.8: Floating-point numbers stored as in IEEE 754 for single-precision (SP) and double-precision (DP)
2.4. FLOATING-POINT ARITHMETIC

The above rounding modes are also called correct rounding. Rounded towards $\pm\infty$ is used in interval arithmetic, while rounding towards nearest is commonly used for all other computations.

In some cases, it is impossible or computationally infeasible to compute the correctly rounded value of a certain function. We say that the system does faithful rounding, if it returns any of the two neighbor values of the real values. More formally, $y$ being the real number, the result $f(y) \in \{\nabla(y), \Delta(y)\}$.

2.4.3 Arithmetic Operations

All basic arithmetic operations (such as $+$, $-$, $\cdot$, $\div$) are defined in the following way:

1. Take the two input floating-point numbers ($x$ and $y$) and convert them to their real number representation.
2. Apply the arithmetic operation in the set of real numbers.
3. Convert the real result into the floating-point result, with respect to the given rounding-mode.

When thinking about mathematics it is often difficult to understand the usage of the special numbers (especially signed zeros). The purpose if these numbers is the following:

- Signed zeros are the result of an operation whose value is too small to be represented in the given floating-point format (an underflow occurred) or the result is truly zero (for example when substractiong two equal numbers). If the real value was negative, $-0$ is chosen, else $+0$.

- $\pm\infty$ are the result of an operation whose value is too big to be represented in the given numbering system (an overflow occurred) or the result is truly infinity (for example when computing $1/0$. If the real value was negative, $-\infty$ is chosen, else $+\infty$.

Figure 2.9: Floating-point Rounding Modes
2.4. FLOATING-POINT ARITHMETIC

Figure 2.10: Corner case, when rounding to nearest is impossible due to error boundary.

- $\pm NaN$ are the result of an operation which received an input value outside the domain it was defined. Examples include square rooting a negative number, logarithm of a negative number, $\pm \infty$, etc.

Due to the fact that elementary functions (such as $\sin(x)$, $\cos(x)$, $e^x$ etc.) can only be computed with a given error boundary (see 2.4.5), it is more difficult to compute their correctly rounded value. The problem is caused by the fact that the error bound might include the halfway between two floating-point numbers, in which case it is impossible to choose the rounded to nearest value (figure 2.10). Similar cases occur for other rounding modes, when the error boundary includes a floating point number.

IEEE 754 standard only requires these functions to be faithfully rounded, while the revision of this standard, IEEE 754r, will most likely require these functions to be correctly rounded.

2.4.4 MPFR

MPFR is a portable C library for arbitrary-precision binary floating-point computation with correct rounding, based on GNU Multi-Precision Library. The computation is both efficient and has a well-defined semantics. It copies the good ideas from the ANSI/IEEE-754 standard for fixed-precision floating-point arithmetic. More precisely, its main features are:

- Support for special numbers: zero, infinities and NaN.

- Each number has its own precision (in bits since MPFR uses radix 2). The floating-point results are correctly rounded to the precision of the target variable, in any of the four IEEE-754 rounding modes.

- MPFR implements all mathematical functions from C99: the logarithm and exponential in natural base, base 2 and base 10, the log(1+x) and exp(x)-1 functions (log1p and expm1), the six trigonometric and hyperbolic functions and their inverses, the gamma, zeta and error functions, the arithmetic geometric mean, the power ($x^y$) function. All those functions are correctly rounded over their complete range.

Two important things to note when using MPFR are:

- MPFR does not track the precision of the result.

29
• (Currently) it is impossible to determine the sign of the stored zero.

In what follows we will describe the most used functions, their parameters and their behaviour, in order for the reader to better understand the section which follow:

**mpfr_t** is an opaque data type that represents an MPFR value.

**mpfr_init** allocates storage space for the given **mpfr_t** and initialises it to the default precision.

**mpfr_inits** the multi-argument version of **mpfr_init**. This function receives a variable number of **mpfr_t** which are to be initialised. The last argument must be zero.

**mpfr_init2** receives as a parameter both and **mpfr_t** and an integer **n**. The given **mpfr_t** is allocated storage space and initialised to the precision of **n**-bits.

**mpfr_clear** frees storage space of the given **mpfr_t**. It may not be used anymore without first initialising it.

**mpfr_clears** is the multi-argument version of **mpfr_clear**. It receives a variable number of arguments representing the **mpfr_t**s for which storage space must be released. The last argument must be zero.

**mpfr_get_exp** gets the exponent of the given **mpfr_t** assuming the mantissa would be in \([1/2, 1)\).

All of the following functions have a last argument which is the rounding mode. One may choose between the values: **GMP_RRND** for rounding towards \(-\infty\), **GMP_RNDU** for rounding towards \(+\infty\), **GMP_RNDZ** for rounding towards zero and **GMP_RNDN** for rounding to nearest.

**mpfr_set_d** sets the value of the given **mpfr_t** to the value of the given double argument. If necessary rounding is done as specified in the last parameter.

**mpfr_get_z** returns a GMP **mpz_t** (used to store signed integers) representing the given **mpfr_t**. The result is correctly rounded as specified in the last parameter.

**mpfr_set_z** sets the given **mpfr_t** to the value of the given **mpz_t**, rounding if necessary.

**mpfr_add** adds the second and the third argument and stores the result in the first argument. If the result argument does not have enough precision to store the exact result, rounding is done as specified in the last argument.

**mpfr_mul** behaves the same as **mpfr_add** but multiplies the operands instead of adding them.
mpfr_sin computes the sine of the second argument and stores it in the first argument. The result is correctly rounded as specified in the last argument.

mpfr_exp behaves the same as mpfr_sin but return the exponential instead of the sine.

mpfr_mul_2si(mpfr_t rop, mpfr_t op1, long int x, mp_rnd_t rnd)
    multiplies op1 by \(2^x\), eventually doing rounding as specified by rnd, and stores the result in rop. If rop and op1 are the same, this function just adds \(x\) to the exponent. This function is very useful to „shift” bits of an mpfr_t.

mpfr_div_2si does the same as mpfr_t only operating with \(-x\).

### 2.4.5 Minimax Polynomial

Given an interval \([a, b]\) and a function \(f\), the minimax polynomial is the polynomial \(p\), of the given degree \(n\), which approximates \(f\) with the smallest maximum absolute (or relative) error. Using the notion of infinite norm, we can say that \(p\) is the \(n\)-degree polynomial that minimises the infinite norm of \(f - p\).

Note that the minimax polynomial is a good approximation of a given function only for the given interval. The error outside that interval can be arbitrary large.

One may compute the minimax polynomial, using the Remez algorithm.

### 2.4.6 Sollya

Sollya is a tool environment for safe floating-point code development. It is particularly targeted to the automated implementation of mathematical floating-point libraries. Amongst other features, it offers a certified infinite norm and a fast Remez algorithm[18].

In contrast with other tools such as Mapple, sollya is a numeric tools, design to compute a certain result with the given precision, augmenting the precision of the internal computations as necessary. However, Sollya is only able to work with a single variable.

All functionality of Sollya can also be used as a library called libsollya. At the time of this writing libsollya is not officially supported by the developers, due to the fact that the interface is still in an early stage and it exposes to much of the internal working of Sollya.

Nevertheless, we required the functionality it provided. We used our own version of libsollya.h, which replaces all Sollya data types with opaque ones, so that we are as protected as possible from changes which happen inside libsollya.

As there is currently no documentation related to how to use libsollya, in what follows we will briefly describe its main data structures, functions and also give a usage example.
2.4. FLOATING-POINT ARITHMETIC

Using libsollya

At the heart of libsollya is the sollya_note_t data structure, which represents an expression tree. In order to create an expression tree one may use make* functions (such as makeVariable, makeConstant, makeAdd etc.) to recursively create the tree. Another option is to call the parseString function whose argument is a Sollya-like string expression. After all processing has been finished, one should call free_memory to free storage space used by the expression tree.

Another libsollya data types is sollya_chain_t which is used to store single-linked lists, containing any element. To create or add elements to a sollya_chain_t one may call the addElement function. The first and tail functions allow a Prolog-like iteration of the chain. When one has finished using the chain, its memory can be freed by calling freeChain.

After an expression tree has been created, one may call the following functions:

- differentiate returns a sollya_node_t representing the derivative of the given argument.
- infnorm returns the infinite norm of a function on a certain range.
- taylor returns a polynomial representing the Taylor expansion of the given function.
- remez returns the minimax polynomial of the given function on a given interval.
- evaluateConstantExpression returns the MPFR value of a given constant expression (i.e. one that has no variable).
- evaluateFaithful does a faithful evaluation of an expression, also receiving the value of the variable.

Whenever libsollya returns a polynomial, it stores it in a complicated horner-scheme evaluation tree. To easily get the coefficients of the polynomial, one may use the getCoefficients functions. Note that the coefficients of the polynomial are returned as sollya_note_t, so if one is interested in their numeric values evaluateConstantExpression should be called.

Figure 2.11 shows a short example of how to use libsollya.

2.4.7 HOTBM

HOTBM [19] is a method for creating hardware circuit which evaluates a given function. The problem it solves can be expressed as follows: Given a function \( f \) defined on a finite input interval \( I \subset \mathbb{R} \), and two positive integers \( w_I \) and \( w_O \) which specify the length in bits of the input and output words respectively, build a hardware circuit which will compute an approximation \( \tilde{f} \) of the function \( f \) on the interval \( I \). Without loss of generality, we can take \( I = [0; 1) \) and scale
```c
#include "sollya.h"

int main(int argc, char *argv[])
{
    /* Convert parameters to MPFRs */
    mpfr_t ia, ib, prec, mpErr;
    mpfr_init(ia, ib, prec, mpErr, NULL);
    mpfr_set_d(ia, 0.0, GMP_RNDN);
    mpfr_set_d(ib, 1.0, GMP_RNDN);
    mpfr_set_d(prec, 1.0e-12, GMP_RNDN);

    /* Create input functions and monoms list */
    sollya_node_t f = parseString("sin(x*Pi/4)"");
    sollya_node_t w = parseString("1");
    sollya_chain_t monoms = makeIntPtrChainFromTo(0, d);

    /* Call remez */
    sollya_node_t nRemez = remez(f, w, monoms, ia, ib, &prec, 165);

    /* Extract coefficients */
    int degree;
    sollya_node_t *nCoef;
    mpfr_t *coef;
    getCoefficients(&degree, &nCoef, nRemez);
    coef = new mpfr_t[degree+1];
    int i;
    for (i = 0; i <= degree; i++)
    {
        mpfr_init(coef[i]);
        evaluateConstantExpression(coef[i], nCoef[i], 165);
    }

    /* Compute the error */
    sollya_node_t nDiff = makeSub(f, nRemez);
    uncertifiedInfnorm(mpErr, nDiff, ia, ib, 501, 165);
}
```

Figure 2.11: Example of using libsollya
f such as \( f(\mathcal{I}) = [0; 1) \). Therefore, we will write the input word \( X \) as \( X = .x_1x_2\ldots x_{w_I} \), and similarly the output word \( Y = \hat{f}(X) = .y_1y_2\ldots y_{w_O} \). We also want our evaluation operator to guarantee the accuracy of the result. As rounding to nearest is impractical, we choose to ensure faithful rounding: \( \epsilon = \max_{X \in \mathcal{I}} |f(X) - \hat{f}(X)| < 2^{-w_O} \).

### Piecewise polynomial approximation

The method we present here is based on a piecewise polynomial approximation: The input interval \( \mathcal{I} = [0; 1) \) is regularly split in several sub-intervals \( \mathcal{I}_i = [i \cdot 2^{-\alpha}; (i + 1) \cdot 2^{-\alpha}) \). These sub-intervals are addressed by the \( \alpha \) most significant bits of \( X \), and we approximate \( f \) on each of them by a degree \( n \) polynomial \( P_i \). Each polynomial \( P_i \) is computed using a minimax scheme, and therefore minimizes the maximum error entailed by this approximation.

As the sub-intervals are addressed by \( \alpha \) bits from \( X \), we can split the input word in two sub-words \( A = .a_1a_2\ldots a_{\alpha} \) and \( B = .b_1b_2\ldots b_\beta \) of length \( \alpha \) and \( \beta = w_I - \alpha \) respectively. This gives: \( X = A + B \cdot 2^{-\alpha} \).

Thus, to compute \( \hat{f}(X) \), we need to evaluate the polynomial \( P_A(B \cdot 2^{-\alpha}) \), which we will write \( P(A, B \cdot 2^{-\alpha}) \) to simplify the notations. We expand the polynomial to obtain:

\[
 P(A, B \cdot 2^{-\alpha}) = K_n(A) \cdot B^n \cdot 2^{-n\alpha} + \ldots + K_1(A) \cdot B \cdot 2^{-\alpha} + K_0(A) \quad (2.3)
\]

The HOTBM method evaluates separately each term (or monomial) \( T_k(A, B) = K_k(A) \cdot B^k \cdot 2^{-k\alpha} \) for \( k \) ranging from 0 to \( n \). A final summation of all the terms then effectively computes the approximated function \( \hat{f}(X) \) (figure 2.12).
Computing the terms

There are several methods to evaluate a term $T_k(A, B)$, and we chose to implement two of them in our work, as described in the following paragraphs.

Simple ROM  The first and the simplest method is to extensively compute all the possible values for the term and tabulate these values in a table addressed by $A$ and $B$, or only by $A$ for $T_0(A, B) = K_0(A)$.

Power-and-multiply  A second method consists in first computing $B^k$ by using a powering unit, and then multiplying the result by $K_k(A)$.

Yet several implementation choices remain. The powering unit can either be a simple table addressed by $B$ where all the possible values of $B^k$ are stored, or a specialized *ad-hoc* unit which first generates then adds all the partial products required to compute $B^k$.

Moreover, the product of $B^k$ by $K_k(A)$ can be spread on several multipliers by splitting the word $B^k = p_1p_2\ldots p_{k\beta}$, which is of length $k\beta$, in $m_k$ sub-words $S_{k,j}$, as in the multipartite method. Spreading the product will allow us to optimize separately each multiplier.

Finally, another choice raised by this method is, for each product $Q_{k,j}(A, S_{k,j}) = K_k(A) \cdot S_{k,j}$, whether to use a table addressed by $A$ and a multiplier, or a single but larger table addressed by $A$ and $S_j$.

Exploiting symmetry  A change of variable in Eq. 2.3 gives a new expression for the polynomial approximation $P(A, B \cdot 2^{-\alpha})$, such that all the terms are symmetric with respect to the middle of the sub-interval $I(A)$: $P(A, B \cdot 2^{-\alpha}) = K'_n(A) \cdot (B - \Delta)^n \cdot 2^{-n\alpha} + \ldots + K'_1(A) \cdot (B - \Delta) \cdot 2^{-\alpha} + K'_0(A)$, where $\Delta = \frac{1}{2}(1 - 2^{-\beta})$. This transformation allows us to compute the terms only on one half of the sub-interval and deduce the values for the other half by symmetry at the expense of a few XOR gates.

### 2.5 FloPoCo

*FloPoCo* is a framework for creating arithmetic operators for FPGAs. Its high-level goals are:

- Arithmetic operators for FPGA should be *easy to create* with *tunable input / output precision* and *automatic pipelining*. As delays and other parameters greatly vary between FPGA families, FloPoCo should be able to generate operators which are *fine-tuned for several FPGA targets*.

- The generated operators should be *easy to combine and reuse* in more complex designs. For example, the shifter and the integer multiplier should be reused in the floating-point
multiplier, instead of the FP multiplier having its own version. This not only shortens development, but also reduces testing time, by reusing components which are already known to work well. While reusing a component, the pipelines and the precision should be automatically adjusted.

- All operators should have automated test-case generation, which would validate the operator using the equivalent software arithmetic. Several classes of tests should be generated, such as exhaustive or random tests.

In contrast to its predecessor, FPLibrary [20] which is a VHDL library of arithmetic operators for FPGAs, FloPoCo is a generator of VHDL code. This design choice has been made for the following reasons:

- VHDL libraries are unpleasant to install, especially if one has to track dependencies between modules located in different files.

- It takes a lot of work to pipeline an operator in VHDL and the obtained pipelined operator is not flexible. It cannot adapt to different speed requirements or FPGAs characteristics. It is impossible to dynamically insert pipeline stages.

- It is impossible to do design space exploration using only VHDL. This is particularly useful for table-based methods, such as HOTBM, which try several variants and only choose the smallest design.

FloPoCo is written in C++ which means it can benefit from well known object-oriented design patterns. At its heart it is composed of the Signal class, the Operator hierarchy and the Target hierarchy.

### 2.5.1 Signal Class

The Signal class represents an input / output or internal signal. Its purpose is to abstract details such as generating the VHDL code needed to define or use it. FloPoCo signals can be in two formats: regular and floating-point.

FloPoCo floating-point format is presented in figure 2.13. The exception field \( \exp \) is 2-bit wide and stores what kind of value all the other fields represent (table 2.1). The sign bit \( \text{s} \) is a 1-bit field, which is 0 for positive numbers and 1 for negative numbers. The exponent and the fraction field have a variable sizes and have similar meanings as in IEEE 754. The FloPoCo format, however, does not support subnormal numbers.
2.5. FLOPOCO

Figure 2.13: FloPoCo’s Floating-Point format

Table 2.1: Values of the exception field and their meaning

<table>
<thead>
<tr>
<th>exc</th>
<th>s</th>
<th>exponent</th>
<th>fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>wE</td>
<td>wF</td>
</tr>
</tbody>
</table>

±0
normalised number
±\infty
±NaN

Figure 2.14: Part of FloPoCo’s Operator Hierarchy
2.5.2 Operator Hierarchy

Operator is an abstract class which all FPGA operators must inherit.

The constructor of a child class explores the design space and computes all the required internal parameters, while the output_vhdl() method outputs the stream (in a printf-like manner) representing the VHDL code generated for implementing that operator.

The Operator class also contains a few helper methods to avoid duplicate work across operators:

- **add_input, add_FP_input, add_output, add_FP_output** adds input / output signals to the operator begin created.
- **output_vhdl_entity** outputs the VHDL entity declaration, with the previously added input / output signals.
- **add_signal, add_register** adds internal signals to the operator begin created.
- **output_vhdl_signal_declarations** outputs VHDL code to declare the previously added internal signals.
- **get_pipeline_depth, set_pipeline_depth** retrieve / store the number of pipeline levels this operator has.

In order to generate a test bench for a given operator, the class which generates it must implement the getTestIOMap, fillTestCase methods. Their usage is described in detail in section 3.2.2.

Also, it is very important to call the set_pipeline_depth method in the constructor of a class, since the test bench generator has to know how much the outputs will be delayed.

2.5.3 Target Hierarchy

Target is an abstract class which contains attributes and methods to store FPGA-specific information. When tuning FloPoCo to a new FPGA family, one has to create a new class which inherits Target.

Currently FPGA characteristics fall into two categories:

- **Architecture-related**: lut_inputs, suggest_submult_size
- **Delay-related**: lut_delay, adder_delay

The Target class also holds global options related to how the operators should be generated:

- **set_pipelined** sets whether the operators should be pipelined.
- **set_frequency** sets the target frequency for pipelining.
- **use_hard_multipliers** set whether the design should use DSP block or not.
Chapter 3

Implementation

3.1 TRNG

In order to achieve random numbers using TRNG, we tried to practically implement and validate both RO-based Jitter Sampling and Fixed-Frequency Jitter Sampling.

3.1.1 Creating ROs in VHDL

Both design have ROs, so our first goal was to create a VHDL component which would implement a ring oscillator with a parametrised length.

We first studied what resources are available in the FPGA to create ring oscillators. The main building block of the FPGA, the CLB are the only ones that actually contain logic, and are interconnected by a network of routing wires. The CLB contains a LUT, an invertor and a memory element which can be either used as a latch or as a flip-flop. The output of latch / FF goes directly out of the CLB into the interconnection network. Two CLBs are grouped together in a slice, however in order to connect the output of one latch / FF to the other CLB in the same slice, the wire has to exit the slice, go through the interconnection network and reenter the CLB. From Xilinx’s reports we noticed that the main delays in FPGA come from latches and routing. The invertor induces a negligible delay. Another interesting thing we noticed is that during the mapping phase, a GLOBAL_LOGIC1 signal is created which provides logic „1” for all the CLBs that require it.

Having the knowledge above, we chose to have a single inverter at the beginning of the chain and a variable number of latches as delay components (like in figure 2.3). A single inverter allows us to create ROs which both even and odd number of latches. By default Xilinx’s synthesis tool optimises out all but one latch, due to the fact that they seem redundant from its viewpoint. In order to prevent this, we must set the „keep” attribute[21] of the \texttt{d} bus which interconnects the latches (figure 3.1).
attribute keep : string;
attribute keep of d : signal is "true";

Figure 3.1: Using the VHDL „KEEP” attribute

This tells the synthesis and mapping tool that we want the individual d signals not to be absorbed into a CLB. Each of them must pass through the interconnection network, which forces the tools to map the redundant latches to CLBs.

To make sure that the inverter does not add more delay, we added the not keyword directly into the port map of the first latch, without assigning it a signal. This has the effect that the inverter and the first latch are mapped to the same CLB.

3.1.2 RO-based Jitter Sampling

Creating Identical ROs

For this design, we created the ROs as described in 3.1.1. The challenging part was to make the two ROs identical, which requires to manually place and route a few CLBs and wires. In Xilinx 9.2i this is achieved by following these steps:

1. Load the project in Xilinx.

2. In the Process toolbox chose Implement Design → Map → Manually Place & Route Design (FPGA Editor). The FPGA Editor will open up.

3. In the Array1 window, zoom to a CLB on which you want to place one RO.

4. In the List1 window, select the loopback wire of the RO you want to place (in our case trng0/roS/ros[0]/b).

5. Drag that item on a CLB in the Array1 window.

6. Scroll to an adjacent CLB and repeat operations 4–5.

7. If your ROs has multiple CLBs, place all CLBs similarly, click Tools → Route → Unroute All, then route each wire from each RO interleaved, by right-clicking the wires and choosing Auto Route.

To make sure that both ROs have the same feedback delay:

8. Click on the loopback wire of one RO, then press F2.

9. The Net Properties window pops up. Click on the Pins tab and note the signal delay.
10. Repeat steps 8–9 for the other RO and check whether the noted delays are equal.

To save the routing constraints in an UCF file:

11. In the Tools menu, click on Directed Routing Constraints.

12. While pressing the CTRL key, click the feedback wires of the ROs to select them.

13. Choose UCF filename and click OK.

The contents of the created UCF file can be pasted into the projects UCF file. This method has the advantage, that as long as one does not touch the TRNG, one may modify the rest of the circuit, without creating unequal frequency ROs.

**Controller Circuit**

Due to the lack of documentation in the original article, we have tried 4 versions of the controller circuits, representing all mixtures for the two behaviours of the E0 and RO signal (figure 3.2).

For E0 we tried the following versions: E0 is cleared on the rising-edge of S0 and set after:

(a) $n \text{ clkS}$ cycles after it was first cleared;
(b) $n \text{ clkS}$ cycles after it was last cleared.

For RO we tried the following versions:

(i) R0 is set / cleared on every rising-edge of S0;
(ii) R0 is set / cleared on every rising-edge of S0, but not sooner that $n$ cycles.

Unfortunately, none of them was able to create a high-speed, high-quality random number generator. With long ROs (more than 10 latches) we had some success creating a high-quality TRNG (even without controller circuit), however it worked only at less then 3KB/s.

![Variants of Waveforms for the R0 and E0 Signals we tried in our control](image-url)
3.1.3 Fixed-Frequency Jitter Sampling

The schema of the fixed-frequency jitter sampling TRNG is presented in figure 3.3. In what follows we will describe a few implementation choices which we took.

**Sampler**

We chose to give the whole TRNG circuit the same interface as the one used by [13], to which we added an input clock signal (figure 3.3). The BitReady output signal is high when the TRNG has a new random bit, which will appear at the RandomBit pin. When the external circuit has stored the random bit, it will acknowledge the TRNG by rising the ReadAck pin.

Although our particular TRNG is synchronous, all three signals are assumed to be asynchronous, both inside the TRNG and the external circuitry that connects to it. We took this decision for two purposes: first, we wanted to be able to use a RO’s output as the sampling clock, which would make the TRNG truly asynchronous, and secondly, we wanted to used the very same design to test future TRNG, which might be asynchronous in nature.

**Resilience Function**

Contrary to the design employed by others, we chose as the resilience function a simple XOR of $2^r$-bits (where $r$ is a generic parameter). We did this because we feared that using a more complex resilience function may hide possible defects in our TRNG, which we obviously want to avoid. Moreover, some resilience functions (such as cyclic codes) are implemented using shift registers and XORs which might act as PRNG. We specifically want to test how well the TRNG works with minimal post-processing. Using the TRNG to seed a PRNG (although a possibly weak one) is against the purpose of our paper.
3.1.4 High-Bandwidth Measurement

It was very important for us to validate the TRNG at its maximum speed. We feared that the output interface from the FPGA to the computer (where the random bits are collected and analysed), whether RS232 or USB, would do additional sampling of the (possible partially) random stream. This would return more optimistic results compared to the TRNG being used only inside the FPGA.

In order to achieve this, we created a design which would first fill a 16 Kbit BlockRAM with TRNG output, then transfer this to the output interface (figure 3.4). We think that this is very close to how a TRNG would be used in a FPGA cryptographic application: the cypher gets values from the entropy buffer and while the algorithm proceeds, the TRNG fills back the entropy buffer.

The design is able to handle burst transfers from the TRNG. The data-in port of the RAM is directly connected to the TRNGs output. The control signals of the address counter and the write-enable port of the RAM are directly connected to the BitReady port of the TRNG, provided the FSM is in the FillRAM state. A separate circuit is used to drive the ReadAck port of the TRNG which sets it to 1 at the very next clock rising edge, exactly when the RAM has stored the random bit.

The FSM which controls this circuit has 8 states (figure 3.5). The first, Idle is the state in which the FSM is set immediately after reset. Transition is made immediately to the PrepareFillRAM state, which resets the address counter. Next, the FillRAM state allows the counter to increase and the RAM to store values when a new random bit is ready. The FSM stays in this state until the RAM is filled (i.e. the RAM address counter wraps around). The
next three states (ReadRAM, ShiftIn, CheckSR) serialise the bits stored in the RAM into a byte for being transmitted to the UART module. The same counter is used to control the address of the RAM, but it is only incremented in the ShiftIn state. Finally, when a byte is complete (i.e. the Serialiser sets the ready port to 1) the FSM will wait for the UART to complete the previous transmission (WaitUART), then dispatch the data (UARTSend). If there is more data to transmit (i.e. RAM address counter is non-zero) then the FSM will transition to the ReadRAM state, serialising the next byte. If the whole contents of the RAM has been transmitted, it will be freshly filled with random numbers, by jumping to the PrepareFillRAM state.

For TRNGs which had an unpredictable bandwidth, we also wanted to measure the bandwidth. In order to achieve this, we inserted a bandwidth counter, which would be reset in the PrepareFillRAM state and incremented on every clock while the FSM is in the FillRAM state. In all other states, the bandwidth counter would store the number of clock cycles ($fcc$), which where required to fill the RAM. Using the following formula, one can determine the throughput of the TRNG:

$$th = \frac{s \cdot f}{fcc}$$ (3.1)

where $th$ is the throughput, $s$ is the size of the RAM and $f$ is the frequency at which the clock of the bandwidth counter is connected.

### 3.1.5 Speeding up the TRNG

FPGAs are becoming large enough to allow massive pipelining of arithmetic operands and compute one result per clock. In some applications it might be desirable to generate random numbers at the maximum frequency of the FPGA. In the above design, both the resilience function (characterised by $r$) and the sample clock divider ($d$) lower the frequency of the TRNG. While we could set $d$ to zero, so that the sampling clock is set to maximum, we can never set $r$ to zero, while at the same time obtain good quality random numbers.

First solution which would come to one’s mind is to use multiple parallel TRNGs and multiplex their outputs. Suppose the sample clock divider is equal to zero, each TRNG would output one bit each $2^r$ cycles. This means that we would need $2^r$ TRNGs for generating one random bit on each FPGA clock. While this solution would surely work (due to the fact that by interleaving truly random streams one obtains another truly random stream), we wanted to find a design that would minimise the resource utilisation.

Our idea is that we require the resilience function because not all our bits are sampled from jitter. The same would apply if we would XOR bits coming from different samplers. This way, we would save $2^r$ counters, FFs and AND gate and replace them with one big XOR (figure 3.6).
3.1. TRNG

CHAPTER 3. IMPLEMENTATION

Figure 3.5: FSM State Diagram
3.2 FloPoCo

3.2.1 FloFP

FloFP is a class which has been designed in order to:

- abstract encoding / decoding of the FloPoCo Floating-Point format
- abstract arithmetic operations (such as addition, multiplication) with correct rounding, on this format
- abstract elementary functions evaluation such as Exponential and Logarithm on this format
- reduce the size of the test bench generation code and maximise code reuse
- implement iteration on this format at the ULP level

In order to mimic the FloPoCo floating-point format, all FloFP constructors receive the parameters $w_E$ and $w_F$, representing the width of the exponent respectively the width of the mantissa, which are stored in private members. The value of a FloFP is stored in four distinct mpz_class members, each for the exception, sign, exponent and mantissa. This has several advantages:

- FloFP can scale to arbitrary large $w_E$ and $w_F$ values
- expensive bit operations are not necessary, as commonly used fields, such as exception can be accessed directly
- all FloPoCo representable values can also be represented as FloFP (note that, for example, mpfr_t does not distinguish between $-0$ and $+0$

In order to make the class as easy-to-use as possible, most operations are implemented as overloaded C++ operators, as detailed in Operators.
The core of the FloFP class are the two methods which convert between mpfr_t and FloPoCo format, getMPFR(mpft_t) and operator=(mpfr_t).

Converting to mpfr_t

The following are the steps to convert a FloPoCo number to an mpfr_t (figure 3.7):

1. Check the exception bits and use the mpfr_set_nan and mpfr_set_inf functions to assign NaN or ±∞.

   In the case of zero, one must note the lack of sign of zero in MPFR. If the sign of zero must be preserved (for example for subtraction, multiplication etc.), it is best to use mpfr_set_d to store one, then scale the exponent using mpfr_mul_2si thus obtaining a very low value, which will almost always underflow when converting back to FloPoCo format.

   In our implementation we used ZERO_EXPONENT equal to −1.000.000.000 in order to provide a safe margin, as mpfr_t does not work well for exponents which are close to INT_MIN. This proved to be sufficient for all our operators, except the floating-point logarithm, for which we specifically request the conversion function to use the real MPFR zero.

2. Should the exception bits be „01”, this means that our FloFP stores a number. To convert it, we first set the precision of the resulting mpfr_t to at least \( w_F + 2 \) bits using mpfr_set_prec and store the mantissa into it.

3. The obtained mpfr_t must be divided by \( 2^{w_F} \) using mpfr_div_2si.

4. The hidden „1” is added using mpfr_add_ui.

5. The unbiased exponent is computed, by subtracting \( 2^{w_E} - 1 \) from the FloPoCo exponent.

   This exponent will be given as parameter to mpfr_mul_2si, to scale the mpfr_t accordingly.

6. If the sign bit is „1”, we negate the so far obtained mpfr_t using mpfr_neg.

Converting from mpfr_t

The process of converting from mpfr_t to FloPoCo floating-point format is almost the reverse of converting from mpfr_t (figure 3.8):
1. We have to check whether the mpfr_t is a special value, such as NaN, 0 or ±∞, using the mpfr_nan_p, mpfr_zero_p and mpfr_inf_p functions. If we hit any special values, we set the exception bits accordingly. For ±∞ we can use the function mpfr_sgn to also set the sign bit.

2. We get the sign, using mpfr_sgn, then compute the absolute value, using mpfr_abs.

3. We extract the exponent of the mpfr_t. The mpfr_get_exp function returns the exponent as if the significand would be in the interval \([0, \frac{1}{2})\). As FloPoCo’s significand is in the interval \([0, 1)\), we have to subtract 1 to obtain the unbiased FloPoCo exponent. We use mpfr_div_2si giving it the mentioned exponent as parameter to obtain the significand, as a number in \([0, 1)\).

4. We subtract the hidden 1.

5. We multiply it with \(2^{w_F}\) using mpfr_mul_2si. So far, we have obtained an mpfr_t whose integer part represents the mantissa of the FloPoCo floating-point number.

6. This next step is critical. We want to obtain the integer representation of the mantissa and store it in an mpz_class. We can either do round-to-nearest (useful for correctly rounded operators), or round-down (for operators with faithful rounding). In case we do round-to-nearest, we must take care of the fact that the mantissa might overflow its allocated \(w_F\) bits. In case this happens, we will increment the exponent and set the mantissa to zero.

7. Finally, we must bias the exponent by adding \(2^{w_E} - 1\) to it and check whether it fits in \(w_E\) bits. Should the mpz_class exponent is negative, an underflow occurred and the exception bits will be set to „00“, representing 0. On the contrary, if the exponent is positive, but does not fit in \(w_E\) bits, an overflow occurred and the exception bits will be set to „10“, representing ∞.

Figure 3.7: Conversion to mpfr_t

1) \(\begin{array}{c}0 \end{array}1\begin{array}{c}1 \end{array}1010010011\)
2) \(\begin{array}{c}0010110011\end{array}\)
3) \(\begin{array}{c}0010110011\end{array}\)
4) \(\begin{array}{c}10010110011\end{array}\)
5) \(-6\begin{array}{c}1001011.0011\end{array}\)
6) \(-1001011.0011\)
3.2. FLOPOCO

CHAPTER 3. IMPLEMENTATION

Figure 3.8: Conversion from mpfr_t

Operators and Methods

In the following lines, we will enumerate the C++ operators and methods which are implemented in the FloFP class and highlight a few implementation details for each one of them.

- The equal operators is used to assign the value of one FloFP object to another, possibly having different \( w_E \) and \( w_F \). In order to handle precision loss with correct rounding, overflow and underflow, this C++ operator calls `getMPFR()` to obtain a lossless representation of the source FloFP, then calls `operator=(mpfr_t)` to do the conversion back to FloFP’s private members.

- The addition and multiplication operators are implemented by obtaining lossless mpfr_t representations of the inputs, calling `mpfr_add` or `mpfr_mul`, then calling `operator=(mpfr_t)` to do the conversion back to FloFP’s private members. The computed FloFP has enough precision to store the exact result. This allows to do correct rounding and handle underflow/overflow in the equal operator, when the precision of the destination FloFP is known.

- \( \exp, \log \) It is impossible to compute and exact value of these two functions, because most of their values are irrational. Therefore, the method we used for addition and multiplication, i.e. postponing rounding until we assign the computed value to the final FloFP, won’t work. Luckily, for FloPoCo, we only require the functions to return the same \( w_E \) and \( w_F \) as the inputs, so we can handle rounding/underflow/overflow, directly in these methods. The equal operators won’t essentially do anything, as \( w_E \) and \( w_F \) of the source and destination already match.

Figure 3.9 shows an example code of using the FloFP class.
3.2.2 Automated Test Bench Generation

In complex hardware designs, each unit has an associated test bench, in order to catch implementation errors as soon as possible and to make sure that no problems occur while integrating this component with the other ones. The test bench is basically a module, used inside a simulator, which instantiates the Unit Under Test (UUT) and feeds it with input values, while checking the outputs. Should the value of any output not match the expected outputs, the whole UUT has failed the tests. The developer can then check what the exact time, input values, actual output values and expected output values were when the UUT failed so he can debug the problem.

Due to the nature of FloPoCo, we wanted to automatically generate test benches for each operator. Clearly it would have been a waste of disk space and time, to pregenerate test cases for each operator, considering the fact that a separate test-bench is required for each value of their parameters. On the other hand, having each operator generate its own test bench would have been a poor design choice. Therefore, we had to design a testing infrastructure which would maximise reusage while at the same time maintain flexibility.

Requirements

Before designing our test infrastructure, we must observed the following requirements:

1. First, there are two data-types which we want to test: usual signals and FloPoCo floating-point signals. The former should be tested for any input value and an output value is correct if and only if all bits of the output value are equal to the expected output value. For the latter, due to the fact that the exponent and mantissa bits have no significance for zero, infinity and not-a-number (exception bits set to „00“, „10“ respectively „11“), there is no need to test all input value combinations. A good test bench, should separately tests the non-number values (for both positive and negative sign) and as many number values as possible (i.e. exception set to „01“). When validating output values, exponent and mantissa bits should be ignored for non-numbers.

2. Second, there are a few operators which only offer faithful rounding (i.e. the output floating-point value can be any of the two closest values to the real number). The test bench should be able to accept multiple possible expected output values. We noted however, that the maximum number of expected output values is always known and usually very small.

3. Next, the final goal of FloPoCo will be to pipeline all operators. The test bench should be able to retrieve the pipeline depth from the operator and automatically delay the comparison of the output values with the required number of clocks.
4. Also related to pipelining, there should be some easy method for the developer to link the
time at which an error occurred, with the input value and expected output values. Using
the error’s time, the developer can easily ask the simulation tool what the actual output
value was. In no case should the developer have to subtract the clock period multiplied
by the pipeline depth to know the values of the inputs.

5. The generated test bench should be as efficient as possible, allowing the testing of as
many values as possible. A test bench should consume as little disk space as possible.

Design

Unfortunately, requirements 4 and 5 are in contradiction. Therefore, we chose to implement
two test bench generators:

Basic Test Bench puts the whole test bench into the FloPoCo generated VHDL file. Simulation
time, input values and expected output values can be easily linked by looking at the
comments inside the VHDL file. This test bench is not space efficient, as input values are
repeated multiple times, once as actual VHDL statements and once as comments. Also,
this solution does not allow too many test cases, not even given infinite time. For exam-
ple, ModelSim considers the whole test bench as a single module and tries to compile /
optimise it as such, while keeping all data structures in memory, instead of reading the
test cases sequentially during simulation. However, this test bench is a very good start for
eliminating most errors.

Soak Test Bench puts the test bench into a separate test vector file. Only minimum VHDL
code is written to the FloPoCo generated file. This allows to simulate many test cases,
without being a burden on memory. In order to maximise space efficiency, input and
expected output values are written in hex to the test vector file and no other comments
are present. This however, makes debugging the UUT, especially pipelined ones, very
difficult. This test bench can be used after an operator passes all basic test bench tests and
one wants to raise the level of confidence of the correct working of the operator.

Both of these test benches are implemented as classes (TestBench and BigTestBench)
inherting Operator. As in all other operators, the output_vhdl() method generates the
relevant VHDL code. Also in output_vhdl(), BigTestBench generates the test vector
file.

Related to the operator-dependent part, we chose the interface so as to be test bench agnostic
and as fast as possible. The speed constraint can easily be met by avoiding data copying, i.e.
transmitting all values by reference.
• `getTestIOMap()` returns an object of type `TestIOMap`, which abstracts the input and output signals which are interesting for the test case. Besides storing the `Signal` object which contains the name, width, type (input / output) of the signal, `TestIOMap` also stores the maximum number of expected output values for each output signal.

• `fillTestCase(mpz_class a[])` receives an array of `mpz_class` with references to all input values and all expected output values. This method must read the input values, do the necessary computations and change the expected output values accordingly. Should it not care about a given expected output value, it should leave it unmodified. It is the task of the test bench to initialise outputs with a special `mpz_class` value to represent „don’t care”. The values in the array are in the order their corresponding signal were return by `getTestIOMap()`, with the note that multiple expected output values occupy consecutive array positions.

Note that we use `mpz_class` to represent signal values, which means that our design can be used for arbitrary large signal widths.

Figure 3.9 presents an example code for implementing these two methods for an operator with faithful rounding (note that the rounded down and rounded up value appears in every test case).

For `TestBench`, in order to abstract the VHDL code generation, the `TestCase` helper class has been implemented. Input values are added using the `addInput` method and expected outputs are added using the `addExpectedOutput` method. The `getInputVHDL` generates the VHDL code for assigning values to the input signals, while the `getExpectedOutputVHDL` generates the relevant VHDL assertions. The `TestBench` generates separate processes for assigning input values and verifying output values, so delaying the assertions due to pipelining is as easy as adding a `wait for` statement in the output verification process.

In `BigTestBench` in order to meet the speed requirements, all `TestIOMap` data returned by the UUT is cached in several arrays:

• `a_in[j]` – true if the j-th value is an input signal
• `a_isFP[j]` – true if the j-th value is a FloPoCo floating-point signal
• `a_w[j]` – the width in bits of the j-th value

In order to read hex values from the test vector file, we used the `hread` VHDL function. This function receives three parameters:

1. `buf` – the string buffer to parse, in our case a line previously read by the `readline` function
2. `s` – the signal to which the parsed value shall be assigned

In order to read hex values from the test vector file, we used the `hread` VHDL function. This function receives three parameters:

1. `buf` – the string buffer to parse, in our case a line previously read by the `readline` function
2. `s` – the signal to which the parsed value shall be assigned
TestIOMap FPExp::getTestIOMap()
{
    TestIOMap tim;
    tim.add(*get_signal_by_name("x"));
    tim.add(*get_signal_by_name("r"), 2);
    return tim;
}

void FPExp::fillTestCase(mpz_class a[])
{
    mpz_class& svX = a[0];
    mpz_class& svRD = a[1];
    mpz_class& svRU = a[2];

    FloFP fpX(wE, wF), fpR(wE, wF);
    fpX = svX;
    fpR = fpX.exp();

    svRD = fpR.getRoundedDownSignalValue();
    svRU = fpR.getRoundedUpSignalValue();
}

Figure 3.9: Example test case interface for an operator
3.2. FLOPOCO

3. good – a boolean which is true if no error occurred while parsing, otherwise false

Unfortunately, the `hread` function proved to be very picky. The signal given as parameters must have the width a multiple of four. Also, the string stored in the buffer (so in our case in the test vector file) has to be padded to exactly represent the signal which is about to be read. Therefore, for each input / output signal of the UUT, there is a separate variable which has the width rounded to the next multiple of four and another variable which returns whether the value was well parsed. A false „good” for an input value is considered an error, however, in case this happens for an output value, it means we retrieved a „don’t care” which is stored in the test vector file as the unparsable string „N/A”.

There is a array of `mpz_class[]` used as a circular buffer to store the last pipeline depth input and expected values. The UUT’s `fillTestCase` method is given the reference to the current item, to avoid data copying.

Also to avoid data copying, there is a preallocated buffer which holds the string representation of the next `mpz_class` which will be output to the test vector file. This buffer is twice the size of the longest hex representation of all inputs / outputs, plus two, so that it can not only store the newline and zero-terminator, but can also be sufficiently padded, without moving data. The `mpz_get_str` function is called in such a way, that the unpadded signal value is always stored from the middle of the buffer.

Both test bench generators output and use the `fp_equal` VHDL function to compare FloPoCo floating-point signals. For `TestBench`, where VHDL literals are given directly to `fp_equal`, a cast has to be made in order to transform the literal into the proper `std_logic_vector` with `downto` direction. A `subtype` is generated for each width of the floating-point signals as indicated in the `TestIOMap`.

3.2.3 HOTBM Operator

The HOTBM method was previously implemented as a separate generator program, having the input function hard-coded. Our goal was to make the input function a command-line argument and then integrate HOTBM into FloPoCo.

The first step was accomplished by using `libsollya`. The command-line string was parsed using `parseString` and transformed into a expression tree. Benefiting from the well-implemented Remez algorithm in `libsollya`, we also chose to remove the previous HOTBM minimax code with a call to the `remez` function.

For the second goal, a wrapper class (obviously called `HOTBM`) has been created. This class contains most of the code that was previously located in HOTBM’s `main` function, except command-line processing which has been moved to FloPoCo’s `main.cpp`. The `set_pipeline_depth` method is called with argument 0, as the HOTBM operator is not
yet pipelined.

For generating the automatic test case, the `getTestIOMap` and `fillTestCase` methods have been implemented. `getTestIOMap` only returns two signals, $x$ the fixed-point input of $w_I$ bits and $r$ the sign + fixed-point output of $w_O+1$ bits. Note that $r$ has cardinality two, as HOTBM only guarantees faithful rounding. `fillTestCase` converts the input `mpz_class` into an `mpfr_t` with enough precision, then uses `libsollya`'s `evaluateFaithful` function. The result is rounded to $-\infty$ and converted back to fixed-point ($r$). The output `mpz_class`es are assigned the previously computed value $r$ and $r+1$, thus the test bench will receive both values valid for faithful rounding.

### 3.2.4 FPLog Operator

The floating-point logarithm operator was also implemented as a separate generator program and had to be integrated into FloPoCo.

First, a wrapper class called `FPLog` has been created, which contains most of the code of the old generator's `main` function. `getTestIOMap` returns only two signals $x$ and $r$ (with cardinality two, as FPLog only guarantees faithful rounding), both being of type FloPoCo floating point. The `fillTestCase` method uses the `FloFP` class to compute the logarithm. By calling `getRoundedDownSignalValue` and `getRoundedUpSignalValue` one gets both values which are valid for faithful rounding.

Another aspect which has to be changed was the name of the packages which the FPLog generated. The old generator was „alone“ in a single file, so it could choose any names it wanted. In FloPoCo however, multiple operators (even of the same type) must coexist without name clashes. FPLog’s code was carefully revised to prepend the `unique_name` of the current FPLog to all package names and entities.
Chapter 4

Results and Discussions

4.1 TRNG

4.1.1 RO-based Jitter Sampling

We have been unable to produce a stable, usable, high-quality TRNG, based on the principle of sampling one RO with another RO. Even if some workaround could be found, building this kind of TRNG implies a lot of effort. This is due to the fact that the ROs must have the same frequency, which cannot be achieved unless the FPGA is manually placed and routed.

4.1.2 Fixed-Frequency Jitter Sampling

Using the principle of sampling the output of multiple XORed ROs with a fixed frequency, we have been able to produce a stable, reliable, high-quality TRNG, with very little effort.

A very important practical aspect of the TRNG is to know the influence of its generic parameters on the quality of its output. We also wanted to test practically what is the smallest number of FPGA resources which are required for this TRNG. We used the DieHard[11] and the TestU01[12] (NIST, Rabbit and Alphabit battery) suites to test the quality of the TRNG output. We only considered parameters for which the output of the TRNG passed all tests, i.e. all DieHard p-values are different from 0 or 1, and TestU01 prints „All tests were passed”. All files which we downloaded had at least 10 MB, due to limitations in DieHard. Interestingly, the TestU01 library proved to be a lot more sensitive than DieHard.

The proposed TRNG has the following generic parameters: number of ring oscillators \( n \), length of ring oscillators \( l \), sampling frequency divisor \( 2^d \) and resilience function input width \( 2^r \).

The first two aspects in which we were interested is the throughput and the amount of resources this design uses. The throughput can be easily computed as the output rate of the TRNG is the input clock frequency, divided first by the clock divider, then the resilience function. The
4.1. TRNG CHAPTER 4. RESULTS AND DISCUSSIONS

The formula is:

\[ b = \frac{f}{2^r * 2^d} \] (4.1)

where \( f \) is the input clock frequency of the TRNG and \( b \) is the throughput in bps.

The amount of resources can also be easily estimated. Each RO uses \( l \) CLBs. The xor stage is synthesised as a tree of LUTs. Due to the fact that the slice of a Spartan 3E device contains 4-bit input LUTs the number of CLBs is \( \lceil \frac{n-1}{3} \rceil \). The clock divisor uses approximately \( \frac{d}{4} \) CLBs. The counter uses about \( \frac{r}{4} \) CLBs, while the and stage at its output uses \( \lceil \frac{r-1}{3} \rceil \). The other components (sampler FF, resilience XOR and FF, acknowledge circuit) use 3 CLBs. Therefore the total number of used CLBs \( (C) \) is:

\[ C = l + \left\lceil \frac{n-1}{3} \right\rceil + \frac{d}{4} + \frac{r}{4} + \left\lceil \frac{r-1}{3} \right\rceil + 3 \] (4.2)

During our experiments we concluded that the quality of the output random bit stream increases with the increase of \( d \), \( r \) and \( n \). As the number of ring oscillators \( (n) \) increases and because the ring oscillators don’t have the exactly same frequency, the signal after xoring them will be composed of much more jitter than flat zones. This means that the sampler will return much more non-deterministic bits compared to the amount of deterministic bits. The more input bits the resilience function has the more non-deterministic bits will be xored with the deterministic bits, which in effect will increase the chance of the TRNG to output a truly random bit.

Regarding the clock divider, if \( d \) is too small (even comparable to the frequency of the ring oscillators), the sampler tends to hit the same flat zone or return the same non-deterministic bit several times. The resulting correlated bits can of course be eliminated in the resilience stage, provided that \( r \) is large enough. We can clearly see that the well-known throughput vs. resources conflict also holds in case of this TRNG.

We haven’t found any significant influence of \( l \) on the quality of the random numbers. This might be due to the fact that while each delay element increases the output period of the ring oscillators, it also increases the amount of jitter, so the percentage of the jitter after the sampling stage remains roughly the same. Although one is tempted to use ring oscillators with the minimum length, we recommend to use \( l \geq 3 \) to make sure that the system does not remain without jitter in extreme conditions such as sudden temperature variations.

In our experiments the parameters values presented in table 4.1 created a TRNG which passed all tests, while minimizing the number of ring oscillators. Please note that in case one wants to be absolutely sure that the TRNG will output high-quality random numbers, higher values should be used for \( r \) or, if bandwidth is an issue, \( n \).
4.2. HIGH-SPEED TRNG

We have also extended this design for high-speed, so that random bits are output at the frequency of the FPGA with minimal resources, while maintaining its high-quality. We have practically validated the fact that good quality random numbers are generated using the above concept, for 8 samplers and 20 ROs / sampler. Interestingly, the number of samplers required is equal to the number of bits which enters the resilience function in the design presented in figure 3.3.

Note however, what for the mentioned values, we used 160 ROs, eight times more. An interesting question is whether this number of ROs could be used to generate a random bit stream, without using a resilience function \((d = 0 \text{ and } r = 0 \text{ in figure 3.3})\). We have practically shown that this is not possible, as explained in [22]. In essence, the probability of sampling a random bit increases with the number of ROs, but never reaches 1. The small percent of the resulting correlated bits is enough to make the TRNG fail quality tests. The difference between the two circuits is illustrated in figure 4.1.

Table 4.1: Parameters for High Quality TRNG

<table>
<thead>
<tr>
<th>(d)</th>
<th>(r)</th>
<th>(n)</th>
<th>(l)</th>
<th>throughput (Kbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>20</td>
<td>3</td>
<td>12500</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>10</td>
<td>3</td>
<td>6250</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>10</td>
<td>3</td>
<td>3125</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>195</td>
</tr>
</tbody>
</table>

Figure 4.1: High-Level Comparison between the two „Equal Throughput” Designs:
(a) High-speed Design (figure 3.6);
(b) Low-speed Design with \(r = 0\) and \(d = 0\) (figure 3.3)

4.2 High-Speed TRNG
Chapter 5

Conclusions

We have tested several implementations of TRNG on FPGAs which can be found in current literature. We have shown how a simple yet of high-quality and high-throughput TRNG can be implemented on a low-end Xilinx Spartan 3E FPGA and presented the main implementation issues one might encounter. We have also discussed the various parameters of the TRNG and the influence they have on the output of the circuit, which we have not found in similar studies. We believe that this work has paved the way to implementing secure cryptographic applications in low-end FPGAs, without requiring any external component.

We have also implemented easy to customise FPGA arithmetic, required for arbitrary precision normal random number generation. We have added the FPLog and HOTBM operators to FloPoCo, which, together with a planned floating-point square root operator will enable the implementation of the Box-Muller method on FPGAs.

However, work in this direction is far from being complete. The following steps must be complete before we can claim we reached our target:

- pipeline the HOTBM operator;
- use HOTBM inside FPLog for generating smaller tables;
- pipeline FPLog;
- implemented floating-point square root operator (FPSqrt);
- do a precision requirement analysis and combine these operators into a new one (Box-Muller).
Bibliography


