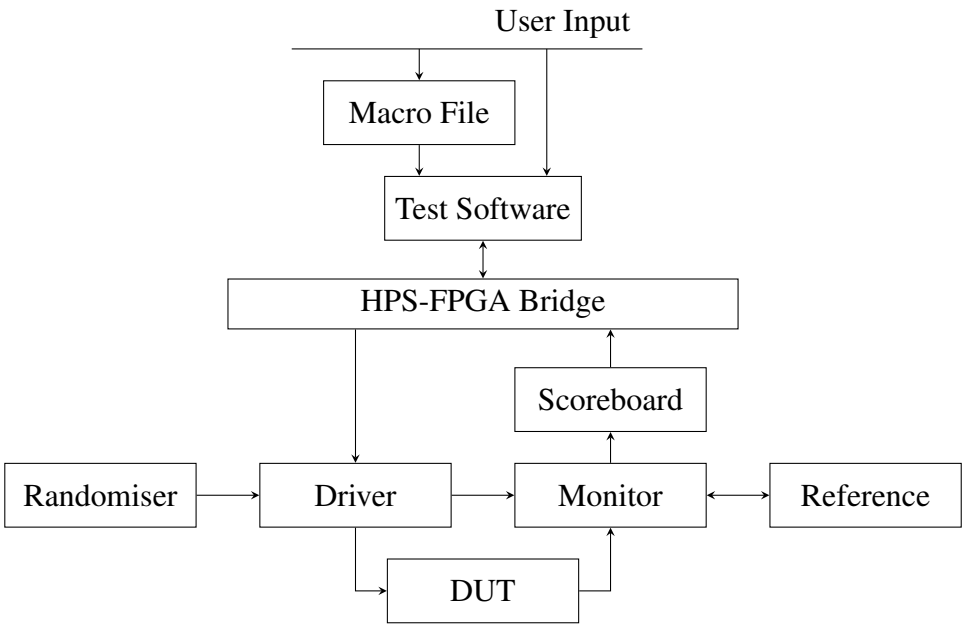
Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report



Project Title:	**An Extensible Framework for At-speed Evaluation of Arithmetic Hardware**
Student:	**Zifan Wang**
CID:	**01077639**
Course:	**EEE4**
Project Supervisor:	**Dr. James J. Davis**
Second Marker:	**Dr. Christos Bouganis**

Acknowledgements

I would like to express my gratitude to my supervisor, Dr. James Davis, for his continuous support at every stage of the project. As someone that started with limited experiences in FPGAs, he skilfully guided me through the engineering process with his impressive technical knowledge and great patience. In report writing, he also provided extremely detailed and timely feedback that I could never thank enough for.

Abstract

In this project, we aim to provide a customisable evaluation framework for arithmetic hardware. Initially, the project was conceived to perform at-speed testing of a set of newly designed high-radix online arithmetic units. The key benefits of this exotic configuration are its high throughput and ability to fail gracefully. As such, the testbench is designed to have a high maximum bandwidth and a method of monitoring the precision of the errors. However, we soon realised that in general, researchers would build their own ad-hoc testbench on FPGAs when experimenting with a new operator design. To improve the efficiency of their research and developments, we propose a flexible evaluation system for arithmetic units with this project. The framework includes both the testing software and the hardware architecture to minimise work for the user. Using a Cyclone V SoC development board, the system is implemented to demonstrate that it is indeed easy to setup and use, and can be modified to accommodate a variety of testing situations. Performance-wise, the implementation is stable at 400MHz on FPGA, resulting in a 4 orders of magnitude acceleration compared to testing by software simulation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the right number representation system, it is possible to perform arithmetic operations MSD first. Consequently, these online arithmetic operators are attractive for hardware implementation in both serial and parallel forms. When computing digits serially, they can be chained such that subsequent operations begin before the preceding ones complete. Parallel implementations tend to be most sensitive to failure in their LSDs, making them more friendly to overclocking than their LSD first counterparts, for which the opposite is true. In the past, online operators have typically been implemented in binary. Although Radix-2 modules are the simplest to design and have the shortest cycle time per digit, they have the highest online delay and require the largest number of cycles to complete calculations [20]. As such, the choice of binary is not absolute.

The initial goal of this project is thus to build a testbench that can investigate the operators' suitability for FPGA implementation and examine the resultant tradeoffs between performance, area and power. However, after some time researching and working on the project, we realised that the testbench can be extended to a more general testing framework with some effort. This makes the project much more meaningful in the long term, as researchers working on other arithmetic units can also utilise this testbench after some configuration. The focus of the project thus shifted to delivering a customisable and extensible verification system while retaining the at-speed testing capabilities needed for the starting goal.

In this report, we will first discuss the motivations of investigating high-radix online arithmetic hardware on FPGAs in chapter 1. Following which the design of the evaluation framework will be put forth in chapter 4, and the design process of each individual module will be examined in detail in chapter 5 and 6. After determining the preferred designs, we will present how each module was built on the FPGA development board in chapter 7, 8 and 9. With the implementation complete, the framework itself will be evaluated to see if it fulfilled its purposes in chapter 10. The results of this test will be subsequently analysed in chapter 11, and the report will conclude in chapter 12 before the proposing a few ideas for further improvement of the product in chapter 13.

Chapter 2

Background Research

2.1 Online Arithmetic

Traditional arithmetic operators have two common characteristics. Firstly, their order of operation may be different depending on the operation itself. A traditional adder, parallel or serial, generates its answers from the LSD to the MSD. A traditional divider design, on the other hand, generates its answer from the MSD to the LSD [3] [16].

Due to this inconsistency, arithmetic operators may be forced to compute word-by-word, waiting for all digits to finish in the previous operator before the next can start [23]. Therefore, if a divider follows an adder, the divider has to wait until the adder has completed its computation before it can begin its own.

The other commonality of traditional designs is that their precisions are specified at design-time. Once built, a 32-bit adder always adds 32 bits together; adding 16-bit numbers usually involves masking the unused bits. A possible way of making it less inefficient would be using SIMD instructions [6], splitting a large register into a few smaller ones, to execute the same instruction on them in parallel. This, however, has the tradeoff of being harder to program, and the applications must have sufficient parallelism to exploit.

Online arithmetic does not suffer from the first issue as it performs all arithmetic operations from MSD first [8] [9]. Furthermore, pipelining can be used with online serial arithmetic operators. Thus the output digit of an earlier operation can be fed into the next operator before the earlier one completes its computation.

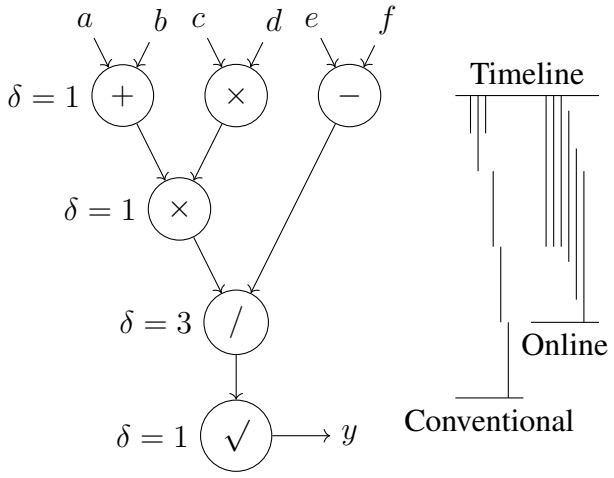


Figure 2.1: Computing $y = \sqrt{(a+b)cd/(e-f)}$ with serial online operators [8]

As illustrated in Figure 2.1, while each individual operation may take longer than its conventional counterpart, online arithmetic can provide a speedup if the operators are chained in serial. In addition to the tradeoff in time, individual online arithmetic operators also use more memory. To perform all computation from the MSD to the LSD, the use of a redundant number system is compulsory. However, this redundancy also has its advantage in making the operators scalable. The time required per digit can be made independent of the length of the operands [21].

A recently proposed architecture allows the precision of online arithmetic to be controlled at runtime [23]. Traditionally, this runtime control was restricted due to the parallel adders present in the multipliers and dividers. This architecture reuses a fixed-precision adder and stores residues in on-chip RAM. As such, a single piece of hardware can be used to calculate to any precision, limited only by the size of the on-chip RAM.

The way online arithmetic alleviates the second problem of fixed precision falls out directly from its MSD-first nature. Suppose the output of a conventional ripple adder is sampled before it has completed its operation. In this case, the lower digits would have been completed, but the carry would not have reached the higher ones. This means the error on the result would be significant, as the top bits were still undetermined [17]. However, if the output of a parallel online adder is sampled before its completion, the lower bits would be the undetermined ones. This means the error of the operation would be small. With overclocking, online arithmetic operators fail gracefully, losing their precision gradually from the lowest bits first. Thus, it allows for a runtime tradeoff between precision and frequency [18].

2.2 High-radix Arithmetic

Conventional designs of arithmetic operators use binary representations. The additional concerns of high-radix operators did not provide justifiable improvements as clock speed of processors kept in-

creasing. In recent years, the clock speed increase effectively ended, and semiconductor dies shrunk to extremely small sizes. This means the relative processing time available in a clock period increased. This enabled and drove the desire for accomplishing more per clock cycle, and high-radix arithmetic is one of them. It has been shown that, high-radix offers power savings and/or reasonable speedups to the arithmetic operations [4] [2] [5].

However, the savings are not without trade-offs. If the radix chosen is not a power of 2, then this trade-off can become unfavourable if the specification requires much I/O and little computation. This is because the overhead of radix conversion would be significant [22]. It is also unwise to use high-radix representations when the numbers are unusually small, thus making the savings offered by the high-radices negligible [4]. The radix also cannot be too high, as the time in a clock period is still limited. If there are too many logic gates for the signal to propagate through, it might become the critical path and slow down the overall design.

The construct of FPGAs might make high-radices more attractive than it is on ICs. As most FPGAs contain small fast carry-ripple adders or hardened carry propagate logic, high-radix adders may be able to exploit them to obtain significant speedups [11].

2.3 Summary

Using high-radix number representations for online arithmetic is a relatively novel concept. While there has been some research with similar premises [13] [14], there are sufficient motivation to implement custom operators made for high-radix online arithmetic on an FPGA. This will provide empirical results on the method, and will hopefully reveal practical insights along the way.

Chapter 3

Project Requirements

This project started as a part of a larger project investigating the effect of using high-radix number representation with online arithmetic operators. The overarching aim involves implementing such a system on an FPGA and quantifying its performance improvements. This is achieved through two individual projects split from the enveloping project. One shall design the arithmetic operator modules, while the other shall design a system to test and evaluate these operators. This project was thus conceived as the evaluation system.

As the project progressed, we soon realised that the system does not have to be only for the specific online arithmetic units, and can be made into a customisable framework that can fit a variety of designs. While there have been many similar performance analyses done on hybrid SoCs before, each of them used their own, usually ad hoc, testbench design [17] [12]. As such, most testbenches are not designed to be scalable or portable, serving only what they are built for.

With this, the aim of the project was shifted to become more of a extensible framework to test arithmetic units, from a single-purpose specialised testbench. However, the framework must still retain the features requires to test the original online designs. The characteristic for online arithmetic units, as discussed in the last chapter, is that they can be fast, and most interestingly they fail gracefully with overclocking. Therefore, the goal of the project is now to design an evaluation framework with the following requirements.

- It should be capable of testing at speed, which means that it can stress test the design a high maximum frequency. Since the process in which an online arithmetic unit degrades as the clock frequency increases, this number must also be controllable during testing.
- It should be able to provide information regarding the precision of the DUT output.
- It should be flexible, so that it can be customised to test different arithmetic hardware;
- It should be user-friendly, so the testbench can be adapted by many users to exploit the flexibility of the framework.

Chapter 4

System-level Design

4.1 Testbench Architecture

The proposed architecture of the verification system shown in Figure 4.1. The hardware side of this structure is inspired by that of an agent in Universal Verification Methodology (UVM). Before UVM, integrated circuit designs were verified with methodologies developed independently by simulator vendors such as Cadence, Mentor Graphics, and Synopsys. In an effort to unify for greater efficiency, the standards organisation of the Electronic Design Automation (EDA) industry, Accellera, established UVM with support from multiple vendors. It provided a common structure for verification, with class libraries that made building and running a testbench a significantly smoother experience. The agent is a container in UVM that emulates and verifies DUTs [24]. While this project is in no position to achieve what UVM has done, I do hope that this testbench would have an easily modifiable structure that will make the process of testing similar future designs slightly simpler.

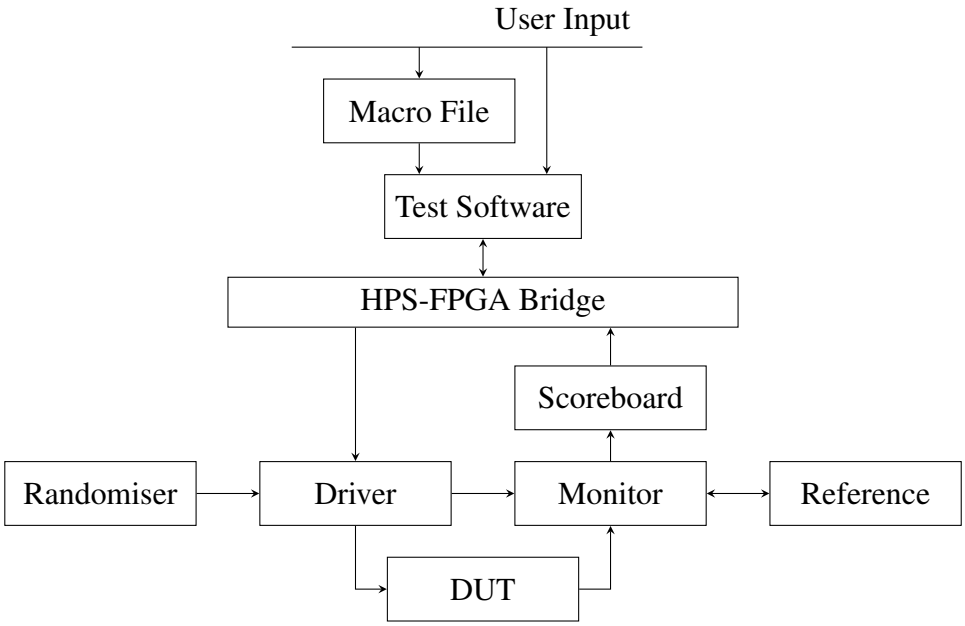


Figure 4.1: Block diagram of the testbench design

The test software running on the HPS will read instructions from either a user-written macro file, or straight from the user through command line, and sends the corresponding commands to the hardware. The driver pulls a stream of random data generated by the randomiser, and converts them to meaningful test inputs according to specification. The test output will be watched by the monitor, reporting the results to the scoreboard, which keeps track of them. The monitor make uses of reference designs functionally identical to the DUT, which allows identification of false outputs from the DUT. Multiple instances of the reference designs means multiple test data can be processed in parallel, so that the reference design can have a relaxed frequency requirement. The scoreboard collects the results from the monitor and writes them to memory locations accessible from the other side of the bridge. These memory locations are read by the test scripts, providing results and other useful information to the user.

A major advantage of taking inspiration from the UVM agent structure is its modularity and thus customisability. Each module has an intuitive purpose and features can be added or removed without much effort. This fits well with the goal of having an extensible framework. Since we needed at-speed testing capabilities, we have modified the one-to-one monitor reference connection, to a one-to-many connection by adding a distributor in between. Because we also needed precision measurements, the scoreboard was augmented with that function without affecting other parts of the design.

The design and implementation of each individual hardware and software module will be elaborated in detail in the following chapters.

4.2 Hardware Choice

The system itself will be built on a Cyclone V SX SoC Development Board from Intel [33].

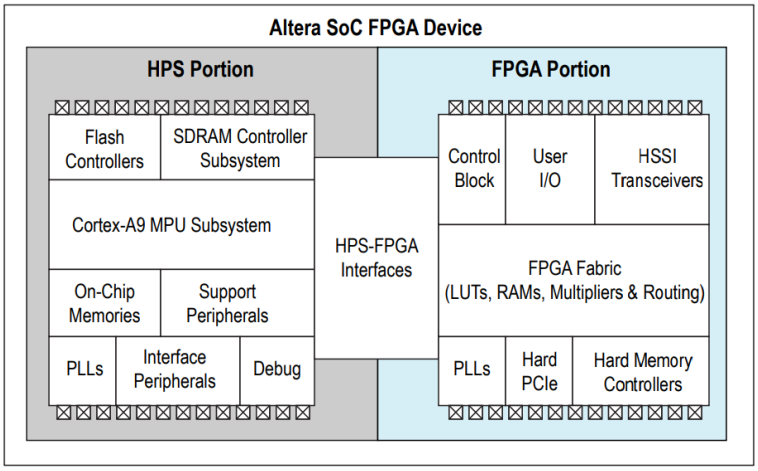


Figure 4.2: Structure of the System-on-Chip

The 5CSXFC6D6F31C6N SoC has an Arm Cortex-A9 MPCore accompanied by Intel's 28nm FPGA fabric [25]. The FPGA is necessary for implementing the hardware design and obtaining empirical results for the project. While an FPGA without an embedded CPU would be enough for this project to work, having a Hard Processor System (HPS) on the same chip is useful as the test software can run on it. The HPS is a separate piece of hardware that distinguishes itself from a soft processor, such as the Nios II — a processor programmed onto the FPGA itself. With this additional capacity, a better user interface can thus be constructed with more detailed, on-the-fly control of the FPGA. This means setting up the testbench will only require programming the design into the FPGA, followed by running the test script on the HPS. The product will thus be self-contained. It will be more accessible as no additional setup is required by the user.

It should be noted that Xilinx offers similar boards as well. Its Zynq SoC family has a very comparable structure as they too integrate the software programmability of an Arm processor with the hardware possibility of an FPGA. For example, similar to the Cyclone V SX, Zynq-7000S features an Arm Cortex-A9 coupled with a Xilinx 28nm FPGA [38]. As such, a board like the ZedBoard [39] could be just as viable for this project.

As there are very few significant functional differences between the two brands, I shall initially explore with the Intel board, simply for its availability and my familiarity with their development tools. Due to the architectural differences between the logic elements between Xilinx and Altera FPGAs [19], the performances on the two boards are not necessarily identical. Once the project has progressed to a point where the system design is mature and tested, the Xilinx alternative can be explored as an extension.

4.3 Software Choice

The software choice follows closely with the hardware choice in this project. To develop for Intel FPGAs, Quartus has to be used. The version picked is arbitrary as there are not many functional differences between the versions that will be critical to the project. As Quartus Prime 16.0 is the version installed in the computers in the department, I will use the same version simply for convenience. This naturally means the hardware system will be built with the system integration tool that comes with Quartus — Qsys.

The Qsys software is designed to be used for integrating different hardware modules into a system. As such, it will be used as the interface for the two parallel projects.

In terms the language, we have opted for Verilog for this project. Since the target DUT will be written in Verilog/VHDL, it makes sense to also use a HDL instead of a HLS language. The preference of Verilog over VHDL is made simply because I was more familiar with Verilog.

Other than the hardware design tools, there is some freedom of choice on the HPS side of the project. The test will be built with Python, which will be running on an Ubuntu system that is installed on the HPS. This choice is made as there are previous unrelated projects on the same development board, which means a lot of time can be saved on tedious setup works such as getting an operating system booting.

Git is used as the version control system for this project. Two repositories on GitHub holds all files related to this project.

Files related to the implementation of the system are available at:

`https://github.com/MerelyLogical/arithmetic-testbench`

Files related to this report are available at:

`https://github.com/MerelyLogical/final-report`

Chapter 5

Hardware Design

5.1 Providing Test Data

In order for the driver to stress the DUT, the verification system must perform at a much higher frequency than the expected frequency of the DUT. Assuming the DUT is to run at 300MHz, to fully explore the effect of overclocking, the testbench must be able to run faster than this frequency. Since it is still the design phase, we shall err on the side of caution and assume a very ambitious frequency margin of +100%, we will obtain an aspiring target frequency of 600MHz. Assuming a data width of 32-bit, the target data transfer rate is then estimated to be 19.2Gbps. With this rough estimate, we can start considering different design options.

5.1.1 HPS-FPGA Bridge

As the testing is to be controlled by the HPS, the HPS-FPGA bridge will be the immediate bottleneck if the test data is to flow from HPS to FPGA. The HPS-FPGA bridge on the development board supports 128-bit numbers and is clocked at 133MHz [30]. Multiplying the two numbers gets us 17.0Gbps, which as an extremely optimistic estimate, still falls short from the requirement. The bridge can only send one word per cycle if it is in burst transfer mode, so to fully use the potential of the bridges, we have to produce the logic that will carry all the data from the bridge to the testbench on FPGA. Since the testbench will be running with variable frequencies, and the numbers do not have to be always 32-bit, unpacking them and crossing them to another clock domain at-speed would be a challenging task. Thus, it is not be sensible for the HPS to send out data during runtime.

5.1.2 Off-chip DDR SDRAM

Another thought may be to first populate the off-chip DDR SDRAM on the FPGA side, then feed that data to the DUT during test. This is already much faster than passing the data directly from the HPS. The

1GB, 32-bit wide DDR3 on the FPGA side is rated at 400MHz. With double rate transfer, this gives a maximum transfer rate of 25.6Gbps.

Although using the off-chip RAM may theoretically achieve the targets, it still has its disadvantages. Firstly, the process of filling up the memory takes time. Thus, the testing would be broken up into bursts, with time in between for checking results and filling in new data. The complexity of the SDRAM interface also requires an SDRAM controller to be used to manage SDRAM refresh cycles, address multiplexing and interface timing. These all add up to significant access latency. While it could be overcome with burst and pipelined accesses, it would further complicate the SDRAM controller. A controller is provided by Intel [27], but it would consume a non-negligible amount of the limited FPGA resources while adding unnecessary complexities to the design. Customising or building a new SDRAM controller to fit this project is possible, but needlessly time-consuming.

5.1.3 On-chip Memory

The on-chip memory is much faster and simpler to use. In comparison, this memory is implemented on the FPGA itself, and thus needs no external connections for accesses. It has higher throughput and lower latency than the SDRAM. The target board has 768.9kB of on-chip memory. With the maximum operating frequency at 315MHz, and the memory transactions can be pipelined, so we have one transaction per clock cycle. Therefore we can get an estimated bandwidth of 242Gbps [36]. With an on-chip FIFO accessed in dual-port mode, the write operations at one end and the read operations at the other end can happen simultaneously. This feature is useful as tests are prepared and fed into the DUT, or when test results are collected and fed to the monitor.

On-chip memory is not without its drawbacks. It is volatile like SDRAM and very limited in capacity. SDRAMs can store about 1GB, while on-chip memory could only hold a few MB [26] for most FPGA boards. Volatility is not exactly of concern in this project, but its small capacity means not much test data can be held before it needs more fed in.

5.1.4 Distributed RAM / Registers

On-chip memory has a minimum latency of 1 clock cycle as the R/W access gets processed. If an even faster memory is desired, we can use LUTs or registers clocked at higher rates to store them. This option would eliminate the latency but takes up much more FPGA resources. The capacity is even more limited as LUTs are usually used for logic. There will be a significant amount of data generated during testing, and the testbench should be as lightweight as possible to allow flexibility in the DUTs. As such, distributed RAM will not be used in this project for data transfer. Registers will still be used as they are essential for many other purposes.

5.1.5 Real Time Data Generation

As seen from the analysis, the best design option here should be able to exploit the benefits of on-chip memory, and circumvent the drawback of buffering testing data generated from the HPS. Generating testing data at runtime, on the FPGA will be such a method. As arithmetic operators have a vast set of valid inputs, it is necessary to have cost-effective test generation.

A good choice here is to use random testing. With relatively low effort, random testing can provide significant coverage and discover relatively subtle errors [7]. The main drawback of random testing is the possible lack of coverage for corner cases, for which the usual solution is to provide handwritten tests to complement it. However, as the main goal of this testbench is gauging the performance of the module, and not necessarily verifying the correctness of the module, having uncovered testing holes is acceptable during stress testing. In the next module, the driver, we will allow manual test inputs to so that the user can cover these holes.

5.2 Randomiser

Linear-feedback shift registers (LFSR) are a reliable way of generating pseudorandom numbers quickly with low cost [10]. Fulfilling the design requirements, they will thus form the starting point of data generation. While it is possible for data generated to be invalid as inputs to the DUT, this should not be the case for most arithmetic units. Even if this is the case, they can be dealt by the filter in the driver. On the other hand, LFSRs go through every single possible value except for one before repeating themselves in a loop, so it is more efficient than a purely random data set. The one impossible value can be covered manually, and knowing that there is an impossible value from the randomiser can be turned into a design advantage later on when we make the driver.

Following this approach, the software would only need to configure the generation at the beginning, and test data no longer needs to pass through the HPS-FPGA bridge. Thus, the testbench can provide fast and constant data to stress the DUT.

5.2.1 LFSR Configurations

While LFSRs are simple hardware modules, there are still a few design options we should explore before implementing them. To compare, we can examine an 8-bit LFSR with taps on bits [7,5,4,3].

In a Fibonacci LFSR, the taps are pulled and fed into a cascade of XOR gates. The output of the final XOR gate is then the lowest bit of the next random number. The higher bits are obtained by one left bitwise shift.

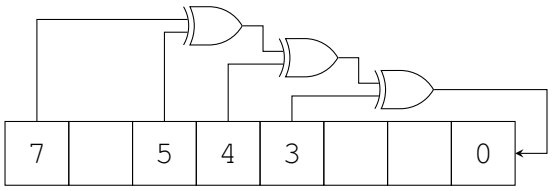


Figure 5.1: Fibonacci Configuration

In a Galois LFSR, the new bits in the taps are obtained by a XOR operation between the lowest bit and the bit on the left of each tap. The highest bit is simply the previous lowest bit, and all other bits are obtained by one right bitwise shift.

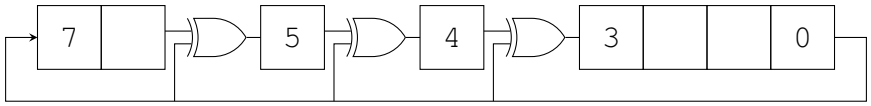


Figure 5.2: Galois Configuration

Other LFSR configurations such as Xorshift [15] exist, but they are mostly designed and optimised as pieces of software, thus being less appropriate for this design.

By examining the two configurations, we can see that Fibonacci LFSRs have to XOR multiple bits together through a cascade of 2-input XOR gates, or a single XOR gate with multiple inputs. On the other hand, Galois LFSRs have multiple 2-input XOR gates working independently. On an FPGA, the cascade of gates is usually implemented with a LUT, so while using LUTs with only 2 open inputs might give some minor improvements over LUTs with 3 or 4 inputs, this increased delay of the Fibonacci configuration will not be significant.

In terms of implementation, Fibonacci LFSRs are slightly easier to code if width reconfigurability is desired. However, building a configurable Galois LFSR is only slightly more complex.

As such, we need to take a step back and examine the overall structure of the randomiser to help us make this decision.

5.2.2 Randomiser Structure

A horizontally structured randomiser uses all bits in the LFSR as output.

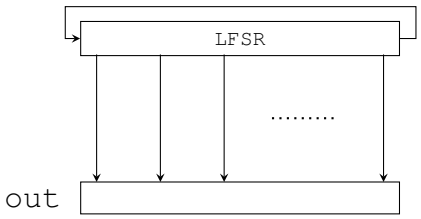


Figure 5.3: Horizontal Structure

A vertically structured randomiser uses multiple LFSRs, and combines one bit from each LFSR for its output.

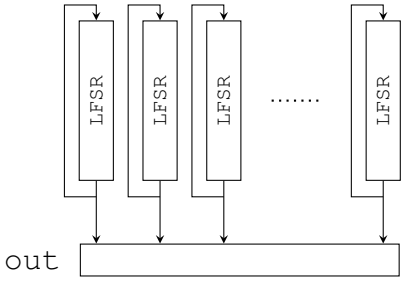


Figure 5.4: Vertical Structure

The horizontal option is easy to construct, but changing the width of the output value requires writing another wider LFSR since the tap positions would change. The vertical options is much more scalable as more or fewer LFSR can be instantiated depending on the required output width. As the widths of individual LFSRs are not related to the width of the output, a series of cheap, 2-tap LFSRs can be used for it, making the earlier point of additional delay for Fibonacci LFSRs a non-issue.

However, the vertical structure is not without downsides. Each LFSR needs a unique seed for its initialisation, making increasing the width not completely automatic unless we also build something that generates these seeds. More importantly, the structure reduces the test efficiency introduced by LFSRs. A single LFSR will go through every non-zero value before repeating itself, the vertically arranged randomiser will have early repeats unless we picked the seeds, lengths and taps of these LFSRs we great care.

If we allow early repeats, then the horizontal structure can be easily scaled. This is achieved by building a long LFSR and taking a truncated version of the output value when fewer bits are required for the tests.

Overall, there is no real immediate advantage of using the vertical structure. Truncation can provide the configurability in the horizontal structure, which means the slight advantage of the Fibonacci LFSRs in its ease to write is nullified. By sacrificing the no early repeat feature of LFSRs, making the randomiser as a horizontal Galois LFSR becomes the most attractive option, and will be chosen as the randomiser design in this project. However, if the randomiser is to be revisited in the future with more time budget, then doing a vertical randomiser might produce a more robust design.

5.3 Driver

The driver should have two main types of operation when feeding data into the DUT. One is the stress testing mode, where the driver tries to push a new piece of data into the DUT at every clock tick. The alternative is a slow manual mode, where the driver reads from the HPS-FPGA bridge and changes its

output to the DUT whenever a new test point is specified by the software. The stress testing mode will expose the DUT to as many random test points as possible in the test duration. The manual mode is used when the user has a special interest in a limited list of inputs.

5.3.1 Stress Testing Mode

The vanilla way of providing data to the DUT is to for the driver to simply instantiate the same number of randomisers as the number of inputs of the DUT. Then the randomisers' outputs can be directly connected to the inputs of the DUT.

While this fast and cheap method fulfils most of the requirements of the testbench, it suffers from a few issues. One, there is no user control for the test data. If we consider the LFSR pseudo-random number sequence to be unpredictable, then there is not much the user can do to influence the test inputs.

5.3.2 Input Filtering

To introduce some level of non-trivial user control in the test data without losing speed, a filtering system is included in the driver design. The filtering system needs to be fast since this is still a stress test, yet it should provide as much utility as possible to the users.

A possible design here is to allow the user to specify a maximum and a minimum bound to the value of the input data. Each output from the randomiser is compared to these values and either sent forward if they passed or replaced if they failed the comparisons. Additional logic needs to be written to produce the data for replacements. If a higher level of control is desired, there can also be a list of invalid inputs within the bounds of validity. However, the latency can get high if the list is long and the comparisons can get slow if the high or low bound is irregular in its binary form. The replacement system can also get complex if the bounds are strict.

A better alternative to achieve this is a bit manipulation system. The user can force individual bits in the data to be cleared or set. This is less flexible than the first design, but with some tricks the user is still able to perform a great level of input control. Having only odd or even test inputs will be trivial under this system. To set maxima or minima for the test data, the user can simply set or clear the higher bits. Certainly this imposes a strong preference to arithmetic units with regular binary representations, but many designs for high-radix arithmetic units, including the ones that spawned this project, use a radix that is a power of 2. As such, the binary manipulation system will mostly be helpful for the user to filter out uninteresting test points or to focus in on more meaningful ones, and thus is sufficient for this implementation of the testbench.

5.3.3 Manual Input Mode

As discussed in the randomiser section, random testing is surprisingly useful, but it does have their limits. If the user has a list of inputs that will trigger key logic paths in their design, they should be able to investigate them under this framework. This would serve as a good compliment to the random testing. The manual input mode is designed for this scenario. In this mode, the user first provides a list of numbers when configuring the test in software, and enables the manual input mode. Then, the test software will read through this list and write them to a set of memory locations on the HPS-FPGA bridge. A simple transfer protocol will be used for the driver to read these locations and then forward them to the DUT. Due to the limitations of the bridge, the DUT cannot be fully saturated with these data. As such, each manual test input will be repeatedly sent to the DUT before the next one becomes available. The scoreboard will be frozen for all repeated inputs since in the subsequent inputs the DUT will have additional time to stabilise its output.

5.3.4 Synchronised Inputs to Monitor

In addition to controlling what gets sent to the DUT, the driver has the responsibility to ensure that the monitor receives the test output from the DUT and the test inputs from the driver at the same time. After going through the filtering logic, the stream of test inputs will not only be sent to the DUT, but also sent to a shift register before reaching the monitor. The shift register will provide the delay required for the DUT to finish its operations. Since the delay in number of cycles from input to output should be consistent and known by the user, the length of this delay can be configured before compiling the testbench.

5.4 Monitor

Another concern in the system design is of the different clock domains that must exist in the testbench. The purpose of the reference design is to be correct, so the timing requirements should be relaxed. This would hopefully make the reference relatively easy to produce and difficult to make mistakes on. Thus, it is not sensible to require the reference design to run as fast as the DUT, but they need to process the same data as the monitor needs to know if the DUT outputs are correct. Therefore, there needs to be a way in the monitor to allow the reference design to keep up with the DUT. Since the rest of the testbench needs to carry the DUT data and is does not have much complex logic, it is convenient to clock everything at the DUT, and have a slow domain that surrounds the reference design.

We considered three possible solutions.

5.4.1 Partial Monitors

A lightweight idea is to implement a partial checker instead of a full model inside the monitors. For example, to check an adder, the monitor can just ask the reference to add the lower few bits instead of the full input.

This should provide a reasonable speedup, but it has a critical flaw. If there are too many bits being checked in the reference, there will not be enough speedup for the reference to keep up with the DUT. If there are too little bits being checked, the monitor would have a large error rate, as it would not be able to know if the higher bits are correct or wrong. Therefore, this method will not be experimented.

5.4.2 Lazy Monitors

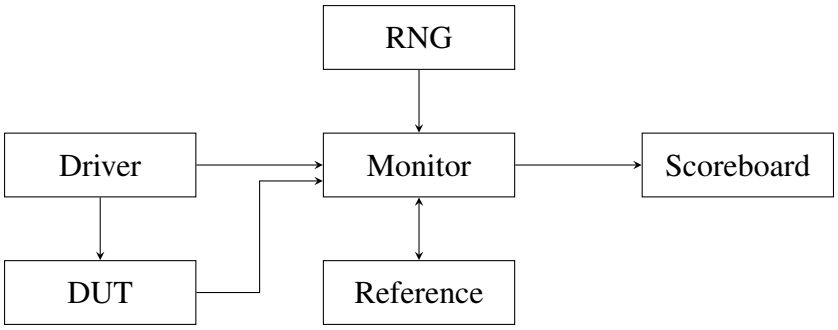


Figure 5.5: Structure of a Lazy Monitor

A more scalable alternative is to have the monitor only check a selection of data sets. For example, if the monitor is programmed to check every third test point, statistically it will make little difference to the final result, but the reference now has three times the time to calculate the correct answer. In the unlikely case the DUT is aware of this and only produce correct outputs on every third operation, this process can be randomised too.

This method can be extended if the DUT get fast simply by skipping more checks, and it has the full information when it detects an error. However, this method needs the extra logic in the random controller, making the monitor slightly more complex than it probably should be.

5.4.3 Parallel Monitors

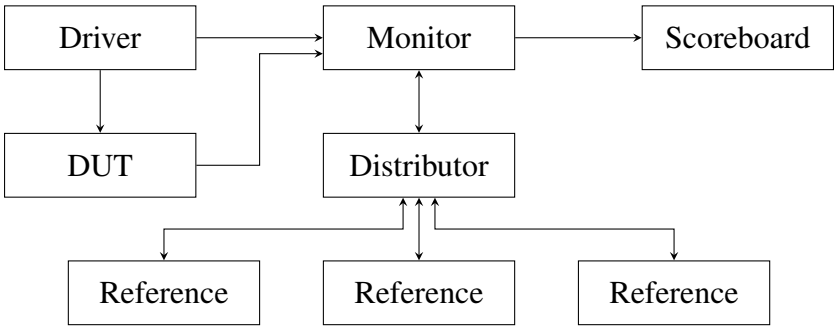


Figure 5.6: Structure of a Parallel Monitor

As the test data is uniform, the monitor can be parallelised to make use of multiple instances of the reference design. These reference-containing sub-monitors are connected to a distributor. The distributor has logic that gives data to all references in a round-robin manner. The results from each sub-monitor are then collected back to the monitor and fed to a single scoreboard.

This does not have data dependency on a random controller, and it can fully guarantee the correctness of the DUT. It is also scalable as more sub-monitors can be added if the DUT gets faster compared to the reference design. As a downside, this method takes up the most FPGA resources to implement as it scales.

Comparing across the three methods, the parallel monitors will be used for this project, as it offers the best functionalities.

5.5 Scoreboard

One of the simplifications has to do with the connection from the monitor to the scoreboard. Previously, there was no guarantee that the scoreboard would be synchronous with the sub-monitors producing the results of the tests. This necessitated an event-driven system, which would reduce the amount of traffic produced by the sub-monitors, and allows the scoreboard to keep track of more test data points than it normally could. Now that the scoreboard runs on the fast clock, the monitor can simply produce one piece of information for each data point, and the scoreboard would be able to handle it.

Since the precision of results coming from the DUT is of interest to us, one possible use of this bandwidth is to pass on the precision of each test data point to the scoreboard. The scoreboard can then produce more detailed statistics from the test set. Instead of only counting how many points are correct and how many are wrong, it will also be able to determine more interesting values such as the maximum and minimum precision of a test set.

These output values will be stored in registers which are exposed and can be read by the HPS. A section in the *Further Work* chapter discusses how this statistics gathering process can be improved.

Chapter 6

Software Design

6.1 Interface Design

The software should provide a layer of abstraction so that the user can run tests without worrying about too many technical details. The abstraction needs to be intuitive, but it should not compromise on the level of control given to the user. Based on the hardware design, a list of items that should be controllable by the user at the software level is drafted.

1. Operating mode (manual or auto)
2. Manual inputs in manual mode
3. Bit set/clear control in auto mode
4. Test duration
5. Phase-locked Loops (PLL) frequency

All other variables such as the exact memory locations accessed and the binary values being read and written should be hidden away by the interface. For example, while the `freeze` signal in the scoreboard is exposed to the HPS-FPGA Bridge, the user will not have to manually assert the signal. The signal should be automatically asserted by the software before results are collected for the scoreboard, ensuring the scoreboard registers are not still counting during the read process. Allowing manual control of this signal would only require unnecessary effort from the user, thus it should be one of the items abstracted away by the software. However, the code still needs to be constructed in a way such that an expert user attempting to modify the bridge interface can to do so without much trouble.

Two options were considered here.

6.1.1 Configuration File

One possible arrangement is to set up a configuration file for the software. We can have a YAML file that lists all the default values for the controls. These values can then be edited by the user to their liking. YAML was chosen due to its Python-like appearance and its easiness to read and edit. Python, the language which the software is written in, also has good support for parsing YAML files with the `pyyaml` library.

```
mode: auto
bitset:
  a: 00000001
  b: 00000000
bitclr:
  a: 00000000
  b: 00000001
input:
  - a: 00000000
    b: 00000000
freq: 200
runtime: 60
```

Figure 6.1: Config File 1

```
mode: manual
bitset:
  a: 00000000
  b: 00000000
bitclr:
  a: 00000000
  b: 00000000
input:
  - a: deadf00d
    b: fadef00d
  - a: feedf00d
    b: cafef00d
freq: 100
runtime: 5
```

Figure 6.2: Config File 2

Looking at the configuration file excerpts, Figure 6.1 will make the test run in auto mode for 60 seconds at 200MHz. Input `a` will always be odd, and input `b` will always be even. Figure 6.2, on the other hand, will make the test run in manual mode, in which the testbench will go through the list of inputs stated under the input key. Once the list is exhausted, the test will terminate.

This method is great for setting up one or two tests, but, to scale this up to series of tests, the user may have to generate a collection of such configuration files. The test software can then scan a folder instead of just a single file and run all tests.

If there happened to be significant extensions to the interface in the future, the user will also have to go through many configuration options that may be irrelevant to the test case. If we provide For example, Figure 6.2 has default values for `bitset` and `bitclr`, but the user needs to understand if they can ignore these values. This increases the cognitive load unnecessarily, reducing the user-friendliness of the design.

6.1.2 Read-eval-print Loop

Alternatively, the software can be structured into a read-eval-print loop (REPL). This is a simple interactive command line interface, where the program reads a single user expression, evaluates it, and prints out a response. The software then waits for the next user input, and so the loop continues until the user decides to stop.

```
> mode auto
> bitset a 00000001
> bitclr b 00000001
> freq 200
PLL Configured to 200.00MHz
> run 60
Results: ...
```

Figure 6.3: CLI Excerpt 1

```
> mode manual
> input a 00010001
> input b 000a000c
> input a feedf00d
> input b cafef00d
> freq 100
PLL Configured to 100.00MHz
> run 1
Results: ...
```

Figure 6.4: CLI Excerpt 2

This allows the user to intuitively command the software. Figure 6.3 and Figure 6.4 show possible commands the user may use to achieve the same effect as the previous option with configuration files. The user can now go directly to the option that needs to be changed, and, if a mistake with the configuration was found when looking at the test results, the user can immediately make adjustments within the software's command line interface (CLI). This means the user can be more exploratory, as REPL is more interactive and direct than having to reset, edit a file, run again, and then debug.

In order to scale this option, the software could be easily modified to also accept a macro file as well. Instead of typing in individual commands in the CLI, the user can enter all the necessary commands as lines of a file, and feed that into the software. The macro file can then be scaled and automated so series of tests can be easily run with a single instruction from the user.

Being the more user-friendly and more scalable option, this option will be implemented in the next phase of the project.

Chapter 7

System Implementation

7.1 Project Hierarchy

Programming the FPGA to communicate with the HPS is no trivial task. Luckily, there exists a golden system reference design (GSRD) [37] for the board in use for this project. Unfortunately, support for certain versions of Quartus are missing from the GSRD download database, including the version used for this project, 16.0. While the design can be opened with a different version of the software, it causes a series of conflicts usually related to using IP cores that have changed over the iterations. To circumvent this issue cleanly, GSRD version 14.1 was downloaded and compiled on a separate install of Quartus II 14.1. This allowed the reference design to be studied in detail, and the sections required for this project to be rebuilt with Quartus Prime 16.0.

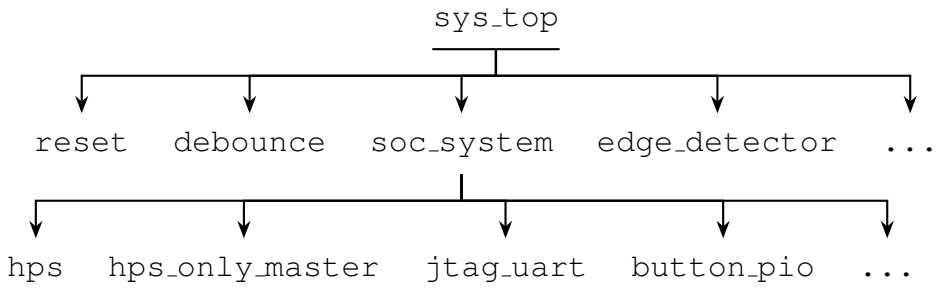


Figure 7.1: Hierarchy of the Golden Reference Design

By examining Figure 7.1, the structure of the reference design, we see that it has a top-level wrapper called `sys_top`, which instantiates the Qsys system `soc_system` and a few IP blocks that handle the low-level hardware controls on the development board. This Qsys system is of the most interest to us as it contains the module `hps`. In `hps`, there are 3 ports named `h2f_axi_master`, `h2f_axi_slave`, and `h2f_lw_axi_master`, corresponding to the bridges exposed by the HPS for connections [30].

With this knowledge, we can insert the testbench design into this hierarchy by having it wrapped into

a Qsys module, with an open port that works as a slave to the AXI bridge. As the traffic passing through the HPS-FPGA bridge is minimal in our design, the lightweight bridge will be used for its simplicity.

As the testbench only requires a list of registers to be sparingly read from and written to, the logic required for the signals on the Avalon slave interface can be handwritten in the module `text_wrapper` according to the interface specifications [35] without much trouble.

By following the naming conventions, the signals allow Qsys Component Editor to automatically detect the Avalon slave from this module during analysis. This saves the trouble of editing the `_hw.tcl` file.

Since Qsys was chosen as the user interface during design, it makes sense to also put the DUT and the reference design at this level in the hierarchy. This means that the user only needs to use Qsys to swap in new designs. As such, other than handling the interface to the HPS, the wrapper module also exposes conduits that connect to the DUT and the references designs.

In addition, the PLL and its reconfiguration IP cores are also instantiated at this level. This is convenient as Qsys is designed for integrating such IP cores.

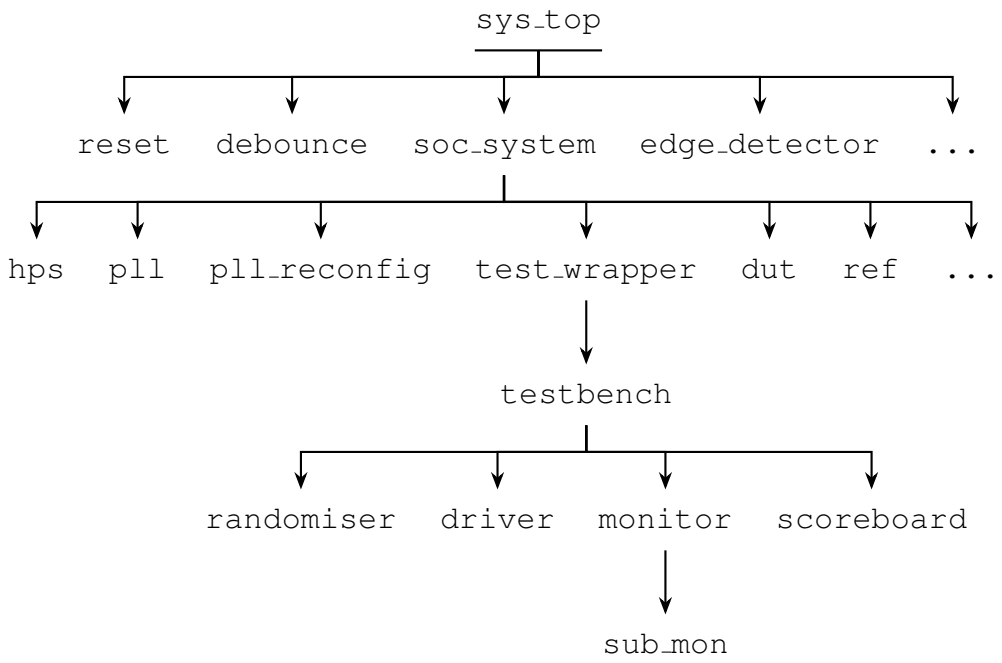


Figure 7.2: Hierarchy of the Full Hardware System

The main section of the testbench is instantiated in the wrapper as a separate module, which is named `testbench`. This module instantiates and connects all main components of the testbench. While this module seems unnecessary, being another wrapper in a bigger wrapper, it does provide 2 advantages.

Firstly, it simplifies the development cycle of the framework. During the compilation of a Qsys system, all relevant files are copied into a folder and the system is transformed into a Verilog file that has its direct dependencies contained in that folder. While this is arguably a benefit in terms of dependency man-

agement, it makes the development more difficult, since whenever something is changed in the interface between testbench modules, the entire `soc_system` needs to be recompiled to update the dependencies. If forgotten, this can cause confusion as the testbench could still be the old version even though a new compilation at the `sys_top` level has been performed.

Secondly, it makes the simulation of the testbench more straightforward. Verifying the correctness of the Avalon slave in the wrapper is important, but there is no need to go through the interface protocol whenever a new input signal is desired in testbench simulations. Having the `testbench` module allows direct manipulation and examination of the signals in and out of the testbench, without worrying about the HPS in simulation.

Chapter 8

Hardware Implementation

8.1 Randomiser

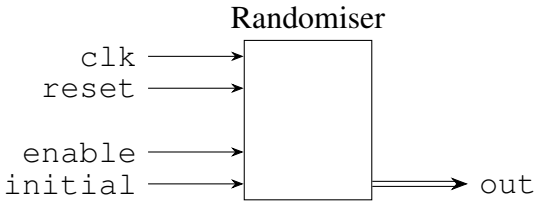


Figure 8.1: Randomiser Block Diagram

Implementing the randomiser is straightforward. A possible set of taps for a 32-bit Galois LFSR is [32, 30, 26, 25]. Referring back at Figure 5.2 on page 17, the logic is to XOR the bits left of the taps with bit 0, and simply right shift all other bits. For the driver to control the randomiser, an enable signal and an initial signal are added as input in addition to the clock and reset. The `initial` signal seeds the LFSR.

8.2 Driver

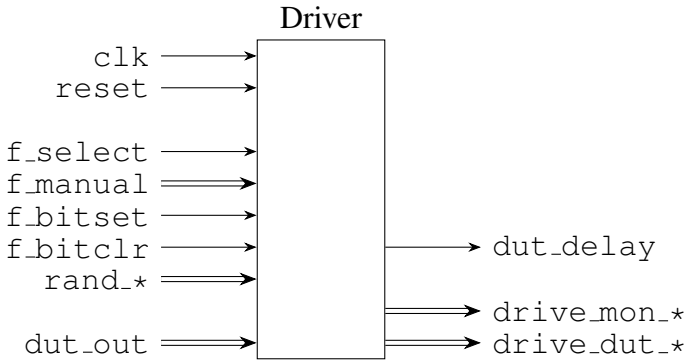


Figure 8.2: Driver Block Diagram

The filter select signal `f_select` selects the mode of operation of the driver. When it is set, the driver will read values from `f_manual` and feed them to the output. Otherwise, the driver will take the output of the randomisers at `rand_*`, setting and clearing specific bits according to `f_bitset` and `f_bitclr`.

The output is immediately sent to the DUT from the ports `drive_dut_*`. The output is also delayed for a number of cycles before being sent to the monitor from the ports `drive_mon_*`. This delay is the cycle delay of the DUT, which is known and thus can be configured by the user before compiling the testbench.

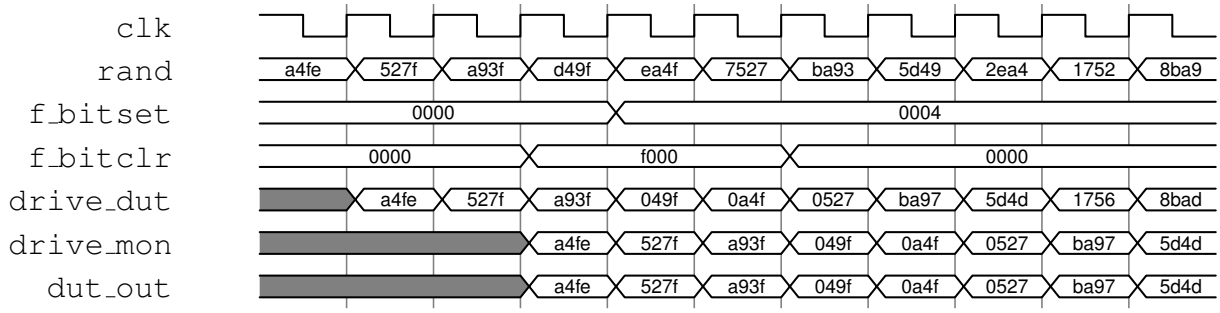


Figure 8.3: Driver Waveform

Figure 8.3 shows how the waveform of the implemented driver looks when operating in auto mode. It should be stated that all waveforms in this report have been obtained from simulation, and are verified to be correct. For clarity, unnecessary signals and signal values have been omitted.

In the example waveform, we assume the design is a simple 1-input 1-output module where the input is passed to the output after 2 cycles. The driver passes `rand` to `drive_dut` after a cycle. When `f_bitclr` is `0xf000`, the top 4 bits of the output are set to 0, and, when `f_bitset` is `0x0004`, bit 2 of the output is set to 1. It should be noted that, if the same bit is set and cleared by the user, `f_bitclr` takes priority. This is an arbitrary choice, and is noted in the user guide.

The driver delays the output to monitor by 2 cycles, and we can see that this aligns it the output from the DUT.

8.3 Monitor

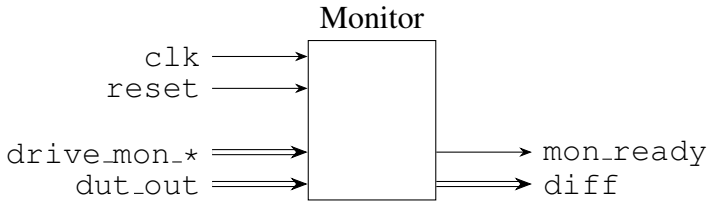


Figure 8.4: Monitor Block Diagram

The monitor takes DUT inputs from the driver, and distributes them to a few sub-monitors. Each sub-monitor containing a reference design then produces the correct results `mon_out` from the inputs with a relaxed time budget. The monitor then checks the difference between the reference output, `mon_out`, and the DUT output, `dut_out`, with XOR gates. This results in `diff`, where each bit set to 1 indicates an incorrect bit in the DUT output. `mon_ready` will be set after the distributor has completed an entire round, where the first meaningful `diff` value becomes available.

As the number of sub-monitors and the width of the tested unit are parametrised, the design of the distributor was not straightforward. An one-hot counter is used to determine the currently active sub-monitor. Since the sub-monitors are clocked `NUM_SUB_MON` times more slowly than the DUT, an array of `NUM_SUB_MON` registers each of size `WIDTH` is also created for each input and output of the design. These serve as the interface between the sub-monitors and the rest of the design.

8.3.1 Sub-monitors

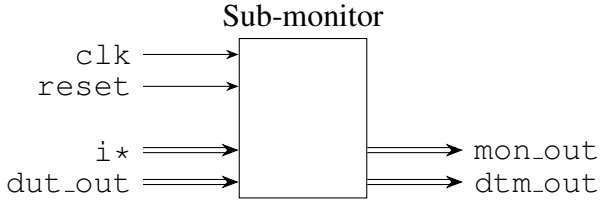


Figure 8.5: Sub-monitor Block Diagram

The current sub-monitor is a part of the monitor module in the architecture design that interfaces with the reference module. In addition to connecting to the reference inputs and outputs, the sub-monitors also handle the delay of the reference module. To accomplish this, there is an extra signal, `dtm_out`, which has the same value as `dut_out`, but delayed by the number of cycles that the reference design needs

to complete its operation, thus aligning with `mon_out`. The other signals are directly connected to the reference module.

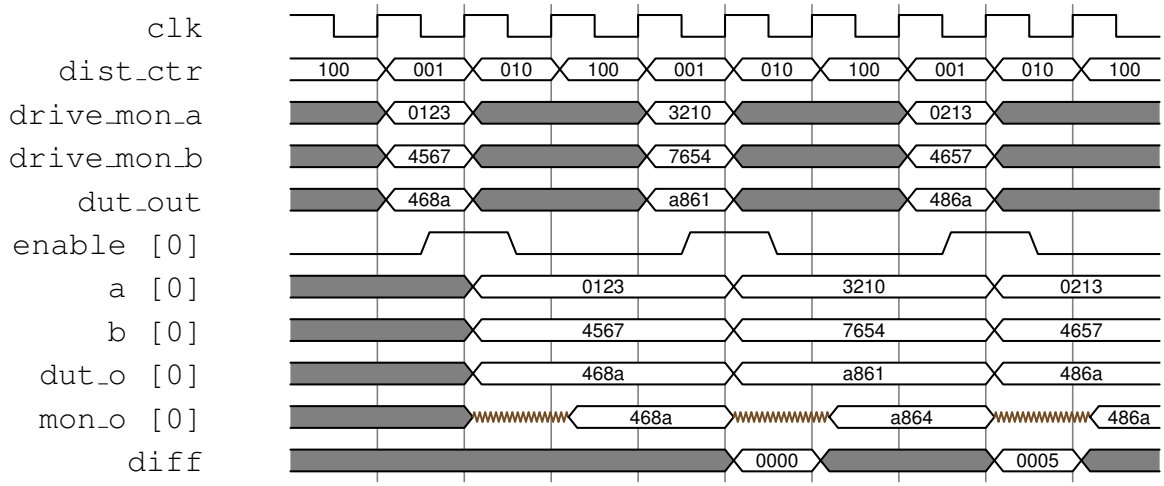


Figure 8.6: Monitor Waveform

Figure 8.6 shows the waveform of a monitor with `NUM_SUB_MON` = 3. The reference adder is a 1-cycle adder, but it must be clocked at a frequency slower than that of the DUT. With 3 sub-monitors, the width of `dist_ctr` is 3, and its lowest bit corresponds to sub-monitor 0, which is shown in detail in this figure. The sub-monitors are driven on the same `clk`, but it also has an `enable` signal that slows down the logic in the sub-monitor. As such, the I/O values are copied into the register arrays and held for 3 cycles. Within this time, the reference design completes its operation, fixing the value on `mon_o`. When the cycle of `dist_ctr` goes a full cycle and the `clk` ticks again with `enable` high, this value is collected back and XOR'ed to form the final output of the monitor, `diff`. In the example of Figure 8.6, the second result was an error on the DUT, as it gave `0xa861` while the reference answer was `0xa864`. This means that bits 0 and 2 were different, and `diff` is thus `0x0005`.

The other sub-monitors all work identically but each 1 DUT cycle later than the last. This allows the reference design to run more slowly than the DUT, but still providing a constant stream of `diff` values at the monitor output, as designed.

8.4 Scoreboard

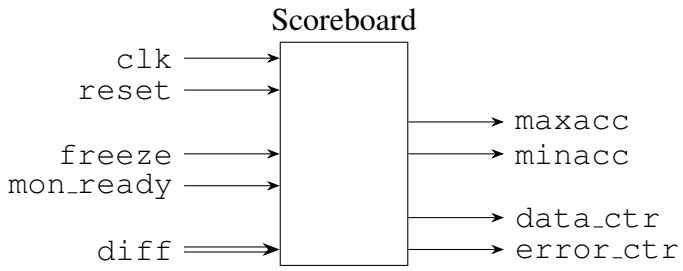


Figure 8.7: Scoreboard Block Diagram

This scoreboard tracks the number of valid test points presented with `data_ctr` and the number of errors within them with `error_ctr`. The external input `freeze` is exposed to the software to stop all counting in the scoreboard. As the current HPS-FPGA bridge setup only allows sequential reads to the FPGA registers, the freeze signal is necessary to ensure that the values do not change within a single set of read commands from the software.

Another implication of the current bridge setup is that there is no simple way of getting all of the `diff` values out to the HPS for statistics calculations. This limitation of the current implementation, and how it might be improved in the future will be discussed in the *Further Work* chapter. Therefore, the hardware will have to do some simple statistics.

Since we are interested in how the precision of the DUT degrades as the frequency increases, two signals are created to record the maximum and the minimum precision of the DUT output. Calculating the precision with the `diff` signal means counting the number of leading zeros (`CLZ`), as zeros indicate correct bits.

As the current implementation is limited to a maximum width of 32, the easiest way of doing `CLZ` quickly is by padding zeros after the number to 32 bits, and then using a large lookup table with don't cares. Figure 8.8 shows how they may be done in Verilog for up to 8 bits. This precision signal is named `acc`.

```
reg [3:0] acc;
wire [7:0] padded_diff;
assign padded_diff = {diff, {8-WIDTH{1'b0}}};

always @(posedge clk)
  casex(padded_diff)
    8'b1xxxxxxx: acc <= 4'h0;
    8'b01xxxxxx: acc <= 4'h1;
    8'b001xxxxx: acc <= 4'h2;
    8'b0001xxxx: acc <= 4'h3;
    8'b00001xxx: acc <= 4'h4;
    8'b000001xx: acc <= 4'h5;
    8'b0000001x: acc <= 4'h6;
    8'b00000001: acc <= 4'h7;
    8'b00000000: acc <= 4'h8;
    default:     acc <= 4'h0;
  endcase
```

Figure 8.8: Verilog code for up to 8 bits CLZ

To keep track of the minimum precision, register `minacc` is first initialised to the maximum, and then, for each smaller value observed, it will take on its smaller value. The comparison logic here is relatively expensive in this fast testbench design. To mitigate this, we can pipeline the comparison and write additional logic so that the results would become consistent a few cycles after the freeze signal has been asserted. Then we can add a ready signal in the scoreboard to indicate to the HPS when the values can be safely read. However, a much easier way is to offload these operations to the HPS, so there is great incentive in the future to build a better data transfer method to HPS.

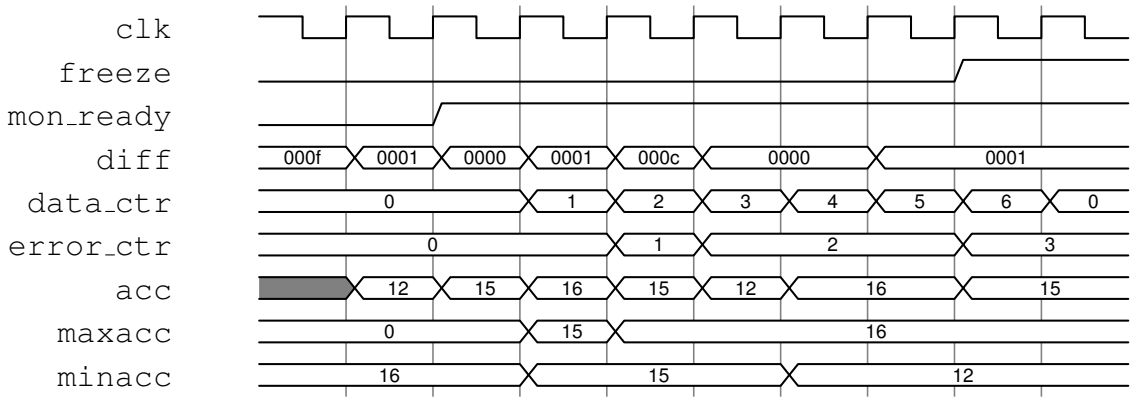


Figure 8.9: Scoreboard Waveform

Figure 8.9 shows an example waveform. The counters and the extrema trackers change values only if `mon_ready && !freeze`.

8.5 Wrappers

Figure 8.11 provides a detailed look at how the individual hardware components are wired together to form the `testbench` module. The DUT is shown as internal for clarity and it is the case during simulation testing of the testbench, but it should be understood that the DUT module is external in actual use. The reference module similarly, is implied to be contained within the sub-monitor module, but this can be external during hardware use.

Figure 8.10 shows the block diagram when both the reference module and the DUT is external. The only differences this will make to Figure 8.11 is that instead of connecting to the DUT, `drive_dut_*` will be exported as `dut_in_*`, and `dut_out` will be exported as `dut_out` at the wrapper level. Similarly, for the reference module, `ref_in_*` and `ref_out` are the exported signal from the sub-monitor.

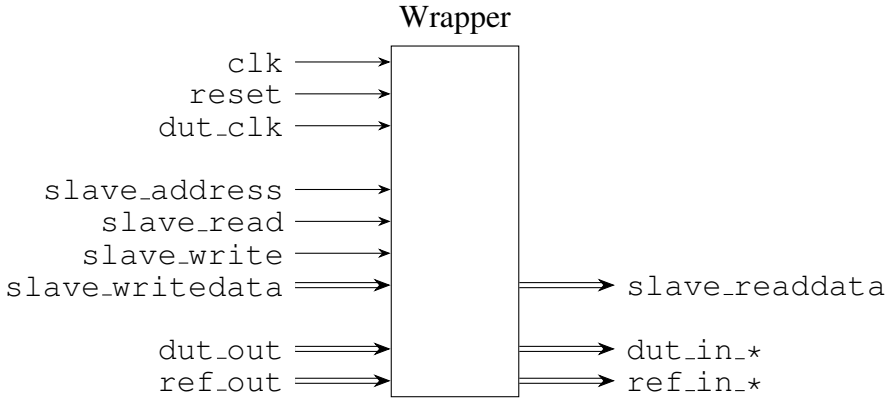


Figure 8.10: Test Wrapper Block Diagram

The inputs and outputs of this module are contained within `test_wrapper`, which handles the AXI communications when coupled with the `hps` module. This allows the software to access the testbench, completing the overall system implementation. In addition to the AXI interface, the conduits to external design and reference modules, we should also see that the wrapper has two clock inputs, since the AXI interface is clocked differently to the rest of the testbench.

All I/O signals are given an address on the HPS-FPGA Bridge. They are listed as follows in Table 8.1. The prefix `I`, `O`, or `D` indicates that the register is write only, read only, or read/write, respectively. Each register name follows that of the corresponding signal closely, except for `reset`, `enable`, and `freeze`, which have been collected into one `D_CTRL` register. They are bit 0, 1, and 2 of the register, respectively.

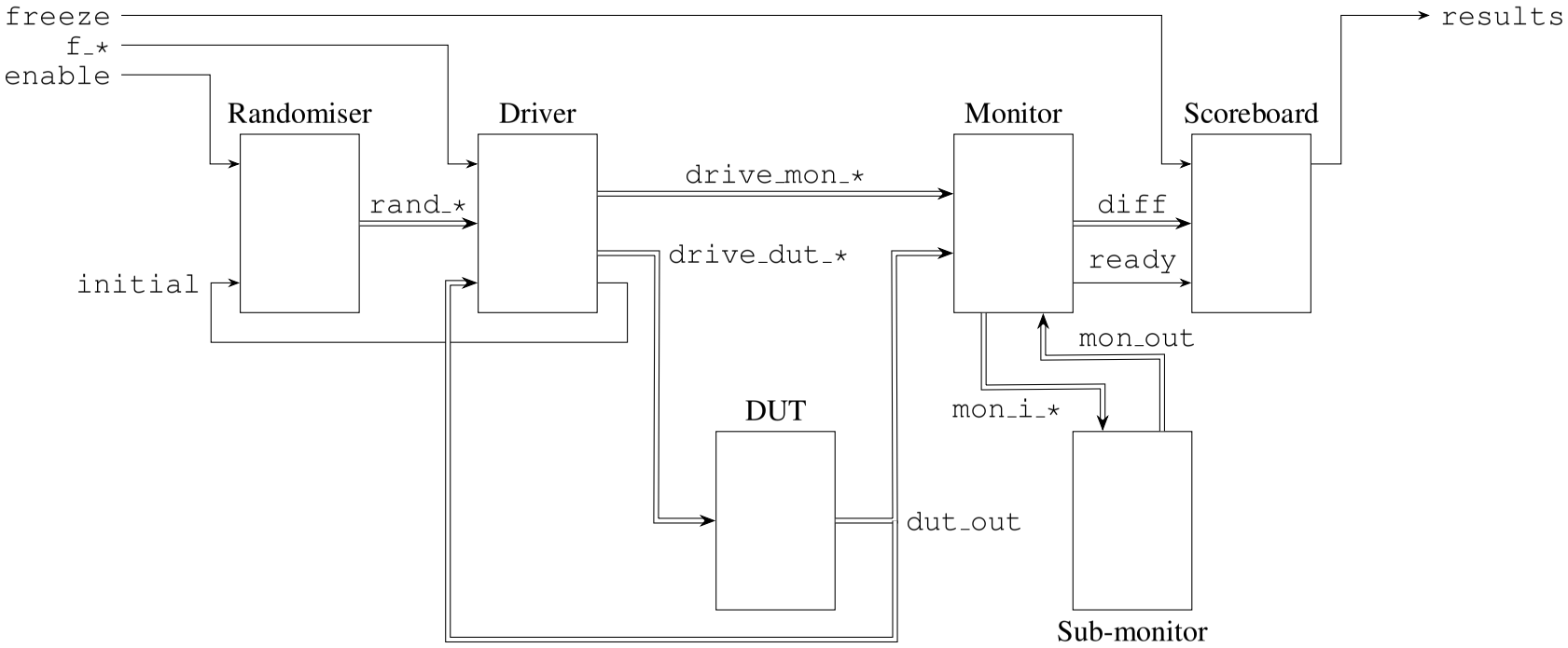


Figure 8.11: Block diagram of the implemented testbench

Register	Location
D_CTRL	6'h00
O_SYSVER	6'h04
I_FSELECT	6'h10
I_FMANUAL_A	6'h14
I_FMANUAL_B	6'h18
I_FBITSET_A	6'h1C
I_FBITSET_B	6'h20
I_FBITCLR_A	6'h24
I_FBITCLR_B	6'h28
O_DUTDELAY	6'h2C
O_DATCTR	6'h30
O_ERRCTR	6'h34
O_MAXACC	6'h38
O_MINACC	6'h3C

Table 8.1: Memory Locations in the Test Wrapper

The listed values are relative addresses. As specified in the manual, the lightweight bridge is physically at 0xFF20_0000 [30]. As the golden reference design already uses some of the lower values in this bridge, an offset address of 0x0010_0000 was given to the test wrapper. For example, the physical address of O_SYSVER is 0xFF30_0004. The PLL configuration also shares the same bridge, so it was given an offset address of 0x0011_0000.

Chapter 9

Software Implementation

9.1 Accessing the FPGA

The interfaces are mapped onto the physical memory in the HPS, thus they can be accessed by opening `/dev/mem` with the `mmap` Python library in the software. The 4 bytes of binary data from the FPGA also need to be packed or unpacked with the `struct` library, as they need to be interpreted as unsigned long integers in little-endian format. The read and write function are defined in a class called `axi`. The base of this function was provided to me by my supervisor.

Since the design is to abstract away direct interactions with the memory locations in the test wrapper, another class called `wrapper` was created to serve as a collection of useful read and write functions. Similarly, a class called `pll` was created to serve the same purpose, but for the PLL reconfiguration module. The initial version of the `pll` class was provided to me by my supervisor, but it was then modified to provide more flexibility in frequency control.

A brief summary of the PLL configuration is as follows. There are 3 reconfigurable stages from the input frequency f_{in} to the output frequency f_{out}, called M, N, and C. M is a multiplier, and N, and C are dividers, or, in an equation, $f_{\text{out}} = f_{\text{in}} \times \frac{M}{N \times C}$. Each stage can be individually bypassed, and there are multiple C stages in a single PLL to provide multiple output frequencies.

Converting from the desired divisor or multiplier to binary write data is not trivial, but it can be written into a function as all the rules are stated in the user manual [31]. After writing in binary data in the required format, the software waits for the hardware to finish configuring by watching another memory location, and then returns the frequency set. This may deviate from the desired frequency due to hardware limitations on resolution from the PLL.

To illustrate the abstraction, we can examine what happens when the user writes the command `reset` to reset the entire system.

First, the software calls the `cleanreset` function in class `wrapper`. This calls a function to set the reset bit before clearing the same bit. It also clears the enable signal and the freeze signal. It then

writes all zeros to all driver filter control registers, completing the wrapper-side reset. The software then calls a function in class `pll`, which sets the PLL frequency back to the default value, thus fully resetting the whole system.

9.2 Read-eval-print Loop

To set up a REPL, a list of expressions that the user is allowed to give to the program is defined as in Table 9.1. These commands are made to be intuitive to the user, yet retaining all functional control of the system.

Command	Explanation
`reset`	Resets the system and test results.
`version`	Prints the system version.
`freq <speed>`	Sets the clock speed to the specified value in MHz. Prints the actual frequency configured.
`mode <m|a>`	Choose between manual and auto test mode.
`manual <a|b> <hex>`	Give input in manual mode.
`bitset <a|b> <hex>`	Force bits to be 1 in auto mode.
`bitclr <a|b> <hex>`	Force bits to be 0 in auto mode.
`run <time>`	Runs the test for the duration specified in seconds. Prints the results at the end of the test.
`exit`	Exits the REPL.

Table 9.1: Commands accepted in test REPL

As user friendliness is a major concern in this project, the user inputs are not assumed to be always syntactically correct. Therefore, the raw inputs are first sent through a series of checks to make sure they are valid. This means the command must be in the list, they must be supplied with the correct number of arguments, and all arguments are in a valid form. If any of the tests failed, the command will not ran and a helpful error message will be printed. Otherwise, the command will be passed with a parser and translated to read and write instructions to the FPGA.

A debug feature is also provided in the software, so the user can choose to run the program in a sandbox first before running it on actual hardware. This is done with the built-in constant `__debug__` in Python. If the user decides to run the script in debug mode with `python ./run_test.py`, the script will not actually write or read anything in the FPGA, but print out the address of all locations that it will be writing or reading in a real run. To actually run the test, the user should use `python -O ./run_test.py` to disable debug mode.

9.3 Automation

The initial disadvantage of a REPL program is that it requires inputs from the users every time they wishes to do something. This can quickly get tedious, so to counter this issue, we have set up an automation system. Now the users can write all the commands that they wants to run into a file, and the script will parse them the same as in REPL mode line-by-line.

This is implemented with a check for `argv` after the program starts. If there is an argument provided when the program is called, as in `python -O ./run_test.py test.do`, then the program will jump to the automated mode, and if there is no arguments, the program will start in REPL mode. This syntax should be familiar to users, as this is how the `python` command and many other command line program works.

Chapter 10

Testing

10.1 Functional Correctness

10.1.1 Simulation

During implementation, the modules were simulated with ModelSim to verify their correctness. Initially we followed the data path of randomiser, driver, monitor, sub-monitor, and scoreboard, each module was attached and simulated. As the randomiser was a perfect source for setting up random tests for the following module, this process was relatively straightforward once the randomiser was ascertained to work correctly. Once the system design became stable, the testbench module can be simulated as a whole. Since RTL simulation is cheap, this is done after any design addition or change thereafter. As writing hardware can be a lot less intuitive than writing software, the simulations have prevented many errors from going on to the hardware.

10.1.2 FPGA Testing

After each major functional addition, the framework is tested as whole on the development board. This means that it is integrated with a test DUT, usually an adder, and the entire system is synthesised and programmed on the FPGA. A basic test script was written early on to for this system level testing. The FPGA testing serves to confirm the functional correctness shown by the software simulations.

However, the confirmatory nature of the FPGA tests stopped after the number of sub-monitors instantiated by the monitor was made configurable. The test results no longer agrees with simulation results. To discover the cause of this discrepancy, another debug method was suggested by my supervisor.

10.1.3 Post-fit Simulation

Previously, the simulation was done with modules files before synthesis. In order to be closer to hardware, we can synthesise and fit the design with first, then with the *EDA Netlist Writer*, a massive Verilog file can be generated containing the entire design. This can then be fed into ModelSim for a more accurate simulation.

With this, a signal was found to be off by a clock cycle so we fixed it in the design. However, this was not sufficient, as the FPGA test results still differ from that of the post-fit simulation. The next proposed debug method by my supervisor is to use *SignalTap*, which can probe signals inside of the FPGA, allowing us to see the actual waveforms in hardware. Being a much more time-consuming method, we decided to first take a closer look at the design code before committing to it. Among other issues, my supervisor was able to identify a clock which I have handled rather carelessly. After spending some time tidying up this clock, the system again behaved correctly as predicted by the simulation.

10.2 Maximum Frequency

As an important benchmark of the framework, the maximum frequency is closely monitored during the implementation process. This is done with *TimeQuest Timing Analyser*, which is a tool that can provide an estimate on the maximum frequency the testbench and run safely on. If this value is lower than what is required, the tool can also provide a list of the slowest offending signal paths for us to optimise on.

During FPGA tests, the physical maximum frequency the board can achieve is measured with a script that will run the same test repeatedly with increasing frequencies. These are useful as the software estimations are usually more conservative than what is possible on hardware.

10.3 Out-of-the-box Testing

10.3.1 Introduction

In an out-of-the-box(OOTB) test, the product is delivered to test users as a packaged box. Its unpacking process, in which the users sets up and uses the product, is then observed to study the intuitiveness of the design. This testing method is used in this project as one of the key determinant for the success of this project is how convenient it is for users to configure and make use of the framework.

For this project, we have made up a scenario where the users have designed an adder with a suspected error that they wish to ascertain. The user guide [Appendix A], the testbench, and a flawed adder design was provided to the test volunteers. An extra piece of logic was added to an adder design, so that if both

inputs are odd, the adder will produce an incorrect output. The test volunteers are made aware of the fact that there might be an error relating to the parity (evenness) of the inputs.

Unfortunately, due to time constraints and the limited number of people with Quartus knowledge that I have access to, this test was only performed by me first and one volunteer after. Nevertheless, this process still proved to be helpful with many design flaws being identified, and a list of possible improvements being drawn out. The possible improvements will be discussed in the *Further Work* chapter, while the identified flaws will be evaluated within this chapter.

10.3.2 Hardware Configuration

For the hardware portion, the test user were given adders of width 16 which is expected to work up to 3 times as fast as the reference. He was able to correctly modify the default values for `WIDTH` where is 32, and for `NUM_SUB_MON`, where the default is 2. However, there seemed to be an issue with Qsys where the widths of ports are not always updated when the module parameters are modified. As he had not much familiarity with Qsys, and was not expected to have any knowledge of the inner workings of the `test_wrapper` module, the test had to be interrupted. To fix this, we deleted and re-added both the `test_wrapper` and the `dut_adder` module, modifying their their default values before their were placed into the *System Contents* window. A note was also added in the user guide.

After dealing with the minor obstacle, he was able to generate the HDL from Qsys, and compile the full design with Quartus without issues.

10.3.3 Using the Test Software

The uploading and the programming procedure was smooth, the volunteer followed the guide and successfully primed the development board for testing.

He then had to read the command list of the testing software and devise a plan to pinpoint the relationship between the parity of the inputs and the correctness of the design output. At this point, he suggested that while some of the commands were explained succinctly, the commands on manipulating the inputs were not explained in sufficient detail and left him somewhat confused. As such, I explained the details and improved the user guide accordingly.

While explaining, the volunteer quickly realised that he can use the `bitset` and `bitclr` commands to fix the evenness of the inputs. However, in doing so he typed in a series of instructions that the test software never had to deal with during initial testing. This revealed a bug in the software where the frequency of the PLL output is not well defined if the software issued a reset command after setting the frequency.

After a second interruption to fix this bug, the test was able to continue. With some experimenting in the REPL interface, the user was able to correctly conclude that the design gives incorrect values only

when both inputs are odd. Test automation was also tested, in which the volunteer had no trouble getting the software to run with his command list file.

10.3.4 Testing Results

Despite the two intermissions, the OOTB was complete within 2 hours and reportedly a smooth experience for the test volunteer. This showed that the framework is reasonably intuitive and user-friendly. However, the test also highlighted a few major places where the project can do better in.

Over the entire duration of the OOTB testing process, a significant amount of time was wasted fiddling with Quartus and Qsys GUIs. It was also the step most where the framework is the most prone to user mistakes. For example, a misclick in the Qsys *System Contents* window may only cause an error during the synthesis of the entire project, and for someone not familiar with Qsys, this error may take a very long time to be identified and fixed. As such, having a way to automate the hardware configuration process would be a great improvement to the usability of the product. Furthermore, a unified configuration system would be even better, as the user will not have to figure out what customisation options have to be done in the GUIs before compile, and what can still be manipulated during in the testing software. A method of accomplishing both will be proposed in the *Further Work* chapter.

Another key improvement arising from the OOTB testing was on the readability of the user guide. It is difficult to produce a guide that will be suitable to readers in all skill levels. Therefore, while this one test has made it slightly better, feedback from more users and especially users with different levels of experience and knowledge is definitely still needed.

Chapter 11

Evaluation

11.1 Product Metrics

11.1.1 Robustness

As planned in the interim report and the *Project Requirement* chapter, we will use 3 metrics to evaluate the performance of the final product. First, the maximum stress of which the testbench can provide without failing is a good metric. This can be quantitatively measured by the maximum data throughput across the DUT, and the maximum frequency that the DUT can be running where the testbench remains reliable. A robust testbench with a higher maximum frequency can reveal a wider picture in the performance of the DUT. This would hopefully allow more insights to be gained regarding the DUT, or it could mean that the testbench can be used for future designs that may be faster than the current one.

To measure this, we can run *TimeQuest* on the compiled design to obtain software estimations of the speed of the design. The worst case scenario is a 1100mV model running at 85°C. Under this condition, the restricted f_{max} is reported as 394.01MHz. However, software estimations are usually conservative, so hardware tests were run to complement the results. After compiling the design on to a FPGA, it was observed that the test was reasonably stable at 400MHz, but breaks frequently at 425MHz.

Although this did not reach the initial goal of 600MHz set in the design phase of the project, it is still high enough to be capable of testing a wide range of designs. This number can be further optimised by pipelining the slowest signal paths and reducing the latency in each cycle. The initial f_{max} estimation before the frequency optimisation was at 210MHz. Because it is a relatively time consuming task with diminishing returns, and does not provide any functional improvement to the implementation, it was dropped for more important work once the milestone of 400MHz was reached.

Overall this is still a satisfactory result, as the frequency still comfortably higher than that of the arithmetic unit, which was assumed to run at 300MHz. Just to compare the hardware acceleration, we timed a software simulation of the same testbench. It managed to process 42000 data points in a second,

which is equivalent to 42kHz, making the FPGA roughly 10000 times faster than software.

11.1.2 Flexibility

As the framework is designed to be extensible and widely applicable, the flexibility of the testbench is also vital to the product's performance. This can be measured by the number of configurable parameters that it has, and the range of which these parameters can be adjusted to.

Item	Reconfigurability	Explanation
`WIDTH`	$\leq$32 bits	Design I/O width
`NUM_SUB_MON`	$\geq$2	Varies time constraint ratio between DUT and reference
f_{dut}	$\leq$400MHz	Frequency of the DUT
DUT delay	All values	The input to output delay of the DUT
bitset/bitclr	All values	Forces bits in test data in auto mode
manual	All values	Sets test data in manual mode
time	All values	Sets test duration
design I/O	2 in 1 out	Number of inputs and outputs supported for design

Table 11.1: Configurable and Fixed Options

Most of these items have been discussed in detail in the previous chapters. The entry on test duration shows that the test data generation and transfer system design removes the upper limit in how long the test can continuously run. This is useful if the user is interested in stressing the FPGA at a high temperature.

While the testbench implementation already allows for a great variety of DUTs to be tested, there are still featural limitations that may become deal breakers for some users.

1. The driver filtering in auto mode is limited to bit-wise manipulation.
2. The DUT can only have 1 or 2 inputs and 1 output.
3. The FPGA has no way of transferring all `diff` data out to the HPS at-speed.
4. The testbench was built with a 32 bits maximum width in mind. This is reflected on how the LFSRs in the randomiser have 32 bits, how the `CLZ` operation in scoreboard is implemented with a lookup table, and how we chose the light-weight AXI Bridge, which has a fixed width at 32 bits [30].

Looking at the lists, we can conclude that the product is reasonably flexible, but more importantly, as a prototype, the implementation has demonstrated the flexibility and the extensibility of the framework design. How we might loosen these limitations and exploit more of the potential from the framework design will be discussed in the *Further Work* chapter.

11.1.3 User-friendliness

The ease of use of the testbench can be another evaluation point. Having a plethora of knobs and switches makes a powerful testbench, but no one would want to use the testbench if the effort to understand and start working with it is overwhelming. As such the framework also needs to be critiqued for its user-friendliness.

The product has been designed and built with the users in mind every step of the way, and the usability has been studied with the OOTB test in the Testing chapter. In hardware, we have exposed the DUT and the reference conduits from the test wrapper to allow easy swapping of different test designs with Qsys. All configurable parameters are made editable from the same interface of Qsys. In software, the REPL is built so that the users can interact intuitively with the FPGA. We have also created a debug mode for the users since we understand that debugging hardware can be a convoluted process, and presenting insightful information to the users can be greatly beneficial.

From the design perspective, the framework is modular with each module having one obvious main purpose. This means the expert users can easily modify the implementation if additional functionalities are desired. This is the same for both hardware and software.

Nevertheless, the product still has potential to serve the users better. The users are still required to use Qsys and Quartus to enter a few parameters and make a few connections before compiling the hardware. They are then tasked to upload the test software and program the FPGA, before being able to start running tests with the REPL or macro files. In all, they have to perform inputs in 4 different locations, and as shown during the OOTB test, if a mistake made early in the process may not surface until the very end.

11.2 Project Metrics

Aside from evaluating the product itself, we should also have a brief analysis at the project level. We first examine at the project plan proposed in the interim report. 8 tasks were laid out onto a timeline in section 5.1. 2 were already complete by the submission of interim report, leaving 3 core tasks and 3 extension tasks to do until the submission of this report. The 3 core tasks were done on time, but there was a major delay during the first extension task, which was titled *Configurable Modules*. Instead of taking one and a half weeks as planned, it took more than a month to complete. The details and the resolution of the problem were discussed in section 10.1.2 and 10.1.2.

Since we have planned in slacks for each task, this delay did not hit the project as hard as it could have without. The next task, *Handling Failures* was cut short, so while the basics features such as providing accuracies of failed outputs and statistical data was built, the added reconfigurability for the verbosity of the statistics was not made available so the output of the testbench is always the same. The initial reasoning for providing varying verbosity was that we were worried that additional logic would slow

down the system, so the user will have the choice between having faster tests or having more detailed results. As we have limited the maximum width of the implementation at 32, the additional logic was built with lookup tables and other components that did not slow down the system as much. The final task, *Interactive UI* was fully complete as an interactive command line interface was built. In addition, it also had automation capabilities with macro files, which was a feature beyond what was planned for this task in the previous report.

As stated in the interim report section 6.2.2, there will always be more potential for further work. The list of limitations and potential improvements of the product was thus within expectation, and should not be considered as a failure on the project level. Therefore, we can claim that the project was reasonably successful on both the management and the execution level.

Chapter 12

Conclusion

In this project, we have proposed an extensible framework for at-speed testing of arithmetic hardware. Using the proposed architecture, we then implemented a testbench to demonstrate its utility and customisability. The modular implementation was verified separately and together with simulations and test runs on FPGA, following which we have estimated the performance of the complete testbench with software tools and hardware benchmarks. These tests showed that the testbench is stable at 400MHz, and thus 4 orders of magnitude faster than a similar test in software simulation. To study its user-friendliness, we have also conducted an out-of-the-box test. A volunteer with no previous knowledge of the project successfully configured the packaged product and obtained desired results without major issues within 2 hours, showing that the testbench can indeed be helpful to a wide range of users.

Chapter 13

Further Work

13.1 Unified User Interface

13.1.1 Expanded Software Design

A major point of feedback obtained during the OOTB test is that the set up process of the testbench is tedious and error-prone. To counter this, we can give more responsibility to the software. Since Qsys and Quartus both provide binary executable files for running in command line without GUIs, the hardware configuration process can be entirely scripted and packaged within the testing software.

The software can first ask the user for values all the hardware configuration variables such as `WIDTH` and `NUM_SUB_MON`. With these values, the software can edit the `.qsys` system file to make the connections and adjust the parameters. `.qsys` files are written in standard XML, which has ample support in Python, so parsing and editing a small part of it should be feasible with some effort.

Then the software can call Qsys to generate the synthesisable RTLs, which can be compiled with another call to Quartus. Establishing a connection to the development board should be possible in software as well. In the environment of the current implementation, the development board is connected with ssh, and files are uploaded with rsync. Programming the FPGA is done with a shell script ran on the HPS, so the entire configuration process can be automated with software. From the perspective of a user, they would have done everything in a single piece of software, saving the confusion and trouble in the current set up.

13.1.2 Additional Customisation

As stated during evaluation, there exist a number of parameters that should be configurable in the testbench implementation. The testbench should not be arbitrarily limited by a width of 32, or a maximum number of input ports of 2. To fix this, we have to write highly parametrised code. However, writing this kind of code in pure Verilog can get cumbersome fast with its generate loops.

These loops does not support changing the names of register in different iterations, which forces the use of multi-dimensional arrays of registers. They are simple enough to access if we only need to access them in one direction. For example, getting the second register in the array is simple, but if we want to do an operation to the bit 2 of every register in the array, the code gets complex and even worse if there are more than 2 dimensions. However, each new configurable parameter introduces at least a new dimension in the design, which makes the code unwieldy. This also makes verifying the code with simulation difficult, since nested generate loops and register arrays always appear together in one direction, quickly clustering up the limited space for waveforms. Preprocessing with macro syntaxes such as ticks and double ticks may work on one simulator but not the other, and could cause trouble in synthesis as well. This is not great for usability as portability and wide support is of great importance to this project.

Since we would already have a software doing all the hardware configuration at this point, we propose writing a small script that generates the Verilog files for synthesis as a part of the software. This means once generated, everything becomes flattened Verilog code, and easy to handle. This preprocessor does not have to be very powerful, it just needs to unroll for loops with variable number of iterations. All loop sizes will be provided by the user before flattening, so it should not be too difficult.

Once completed, this would make the process of adding new configurable parameters much easier for both developers and users. Developers will have an easier time writing RTL and verifying them in simulation, and the user can just enter any additional parameters into the same piece of software with the same interface as before.

13.2 Software Test Statistics

Another limitation for the current testbench is for how it processes the test results. Since there is no implemented way of transferring results at-speed from the FPGA to the HPS, the precision data collected in the scoreboard module had to be turned into two registers, `maxacc` and `minacc` for the HPS to read. This is a great waste of potential as the HPS can be much better at processing the precision data and providing them in an insightful manner to the user. The precision of a 32-bit number can be one of 33 values, so it can be conveyed in 6 bits without additional compression. They would still fill up any on chip memory fast. At 400MHz, it still fills up 300MB of space in a second even if we assume byte alignment is not an issue.

There are two possible solutions here. On one hand we could send as much as possible through the standard HPS-FPGA bridge and drop the rest and hope that it would not affect the overall statistics. On another we could save them up on the 1GB RAM since it is rated at 3.2GB/s. Then we can test in bursts so that the RAM has time to send out its stored data.

Either way, a tradeoff is necessary as expected. Therefore as planned in the interim report, we also need to have a switch that turns verbose statistics on or off.

13.3 Automatic Delay Reconfiguration

13.3.1 Delay Tester

One of the many parameter values users need to specify, before compiling the hardware, is the output delay of the DUT in number of clock cycles. Since having one less thing the user has to specify is one step towards a more convenient system, a delay tester is built to test the idea of on-the-fly reconfiguration of expected DUT delay in the driver. It counts the number of cycles for the DUT to produce its output as `dut_delay`, but there are a few limitations in its design, so the logic of dynamically configuring the driver with `dut_delay` was not complete.

The delay tester is built with a simple FSM.

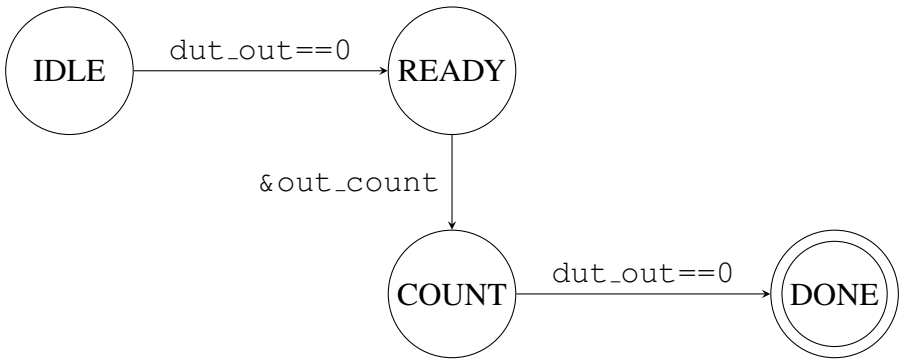


Figure 13.1: Delay Tester FSM

We first start a counter called `out_count`. Whenever this reaches all 1's, the driver output is set to some value with a known safe DUT output. In this example, the safe output value is 0, which means when the delay tester is active, the driver will not generate any input to the DUT that will result in 0 on `dut_out`. The delay tester can then test the DUT delay with this safe value, since it knows when the DUT received the safe input, and can count the number of cycles until the DUT gives the safe output.

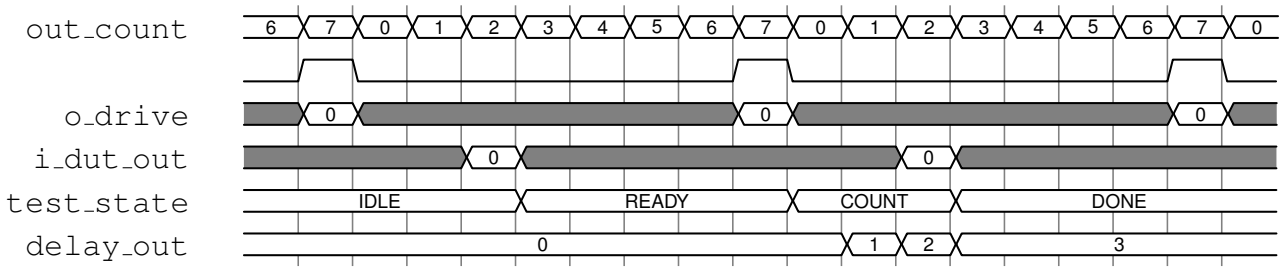


Figure 13.2: 3-bit Delay Tester Waveform

The FSM starts in state IDLE. When the first 0 output is detected from the DUT, the FSM enters the READY state. It now knows that it can enter the COUNT state when the `out_count` becomes all 1's again, triggering the next safe test input. The FSM leaves the COUNT state for the DONE state

when the safe output of 0 is detected. The delay tester process is now complete and `out_count` can be deactivated.

With this, the DUT's delay in clock cycles is the same as the number of cycles that the FSM stayed in state COUNT. The delay counter `delay_out` increments itself every cycle if the FSM is in that state. When the FSM enters the DONE state, we the value of the delay counter is the delay of the DUT. With a 3-bit counter as shown in the timing diagram, it can measure this delay for up to 8 clock cycles. Longer delays can be measured by extending the width of `out_count`.

13.3.2 Limitations

First, there has to be a safe value for the delay tester to use. For adders and multipliers, this is relatively simple as LFSRs cannot generate zero as inputs and the output can only be zero if the both or one input is zero, respectively. This is possible with subtracters as well since the two LFSRs will not produce the same number at the same time if they were seeded differently. However, as soon as the ALU starts dealing with overflows, IEEE floating numbers, or any non-standard number representation system, there would be no guaranteed safe value without user input. If the user have to input safe values for the delay tester to use, then if defeats the purpose of saving one parameter for the user to configure.

The addition of this function also slowed down the $f_{\max}$ to 315.06MHz. As such, the idea of having an on-the-fly delay reconfiguration system was put on hold, as it might not be worth the engineering effort to save the user one single input.

Bibliography

[1] I. Ahmed, S. Zhao, J. Meijers, O. Trescases and V. Betz, "*Automatic BRAM Testing for Robust Dynamic Voltage Scaling for FPGAs*", *Int. Conf. on Field-Programmable Logic and Applications*, 2018.

[2] A Amin, W. Shinwari, "*High-Radix Multiplier-Dividers: Theory, Design, and Hardware*", *IEEE Trans. Comput.*, vol. 1, no.8, 2008.

[3] R.P. Brent, "*A Regular Layout for Parallel Adders*", *IEEE Trans. Comput.*, vol. C-31, pp. 260-264, 1982.

[4] B. Catanzaro, and B. Nelson, "*Higher Radix Floating-Point Representations for FPGA-Based Arithmetic*", *Proceedings of the 51st Annual Design Automation Conference*, 2005.

[5] L. Chen, F. Lombardi, P. Montuschi, J. Han and W. Liu, "*Design of Approximate High-Radix Dividers by Inexact Binary Signed-Digit Addition*", *Proceedings of the on Great Lakes Symposium on VLSI*, 2017.

[6] R. Duncan, "*A Survey of Parallel Computer Architectures*", *Computer*, vol. 23, pp. 5-16, 1990.

[7] J.W. Duran, "*An Evaluation of Random Testing*", *IEEE Trans. on Software Engineering*, vol. SE-10, no. 4, pp. 438-444, 1984.

[8] M.D. Ercegovac, "*On-line Arithmetic: An Overview*", *28th Annual Technical Symposium*, pp. 86-93, Internaltional Society for Optics and Photonics, 1984.

[9] M.D. Ercegovac, and T. Lang, "*Digital Arithmetic*", Morgan Kaufmann, 2003.

[10] S. Hazwani, et al, "*Randomness Analysis of Pseudo Random Noise Generator Using 24-bits LFSR*", *Fifth Int. Conf. on Intelligent Systems, Modelling and Simulation*, 2014.

[11] P. Kornerup, "*Reviewing High-Radix Signed-Digit Adders*", *IEEE Trans. Comput.*, vol.64, no. 5, pp. 1502-1505, 2015.

[12] H. Li, J.J. Davis, J. Wickerson and G.A. Constantinides, "*ARCHITECT: Arbitrary-precision Constant-hardware Iterative Compute*", *Int. Conf. on Field-Programmable Technology*, 2017.

[13] T. Lynch, and M.J. Schulte, "*A High Radix On-line Arithmetic for Credible and Accurate Computing*", *Journal of Universal Computer Science*, vol. 1, no. 7, pp. 439-453, 1995.

[14] T. Lynch, and M.J. Schulte, "*Software for High Radix On-line Arithmetic*", *Reliable Computing*, vol. 2, no. 2, pp. 133-138, 1996.

[15] G. Marsaglia, "*Xorshift RNGs*", *Journal of Statistical Software*, 2003.

[16] H.R. Srinivas, and K.K. Parhi, "*High-Speed VLSI Arithmetic Processor Architectures Using Hybrid Number Representation*", *J. of VLSI Sign. Process.*, vol. 4. pp. 177-198, 1992.

[17] K. Shi, D. Boland, and G.A. Constantinides, "*Accuracy-Performance Tradeoffs on an FPGA through Overclocking*", *Proc. Int. Symp. Field-Programmable Custom Computing Machines*, pp. 29-36, 2013.

[18] K. Shi, D. Boland, E. Stott, S. Bayliss, and G.A. Constantinides, "*Datapath Synthesis for Overclocking: Online Arithmetic for Latency-Accuracy Trade-offs*", *Proceedings of the 13th Symposium on Field-Programmable Custom Computing Machines*, pp. 1-6, ACM, 2014.

[19] O. Šćekić "*FPGA Comparative Analysis*", *University of Belgrade*, 2005.

[20] A.F. Tenca, and M.D. Ercegovac, "*Design of high-radix digit-slices for on-line computations*", 2007.

[21] K.S. Trivedi, and M.D. Ercegovac, "*On-line Algorithms for Division and Multiplication*", *IEEE Trans. Comput.*, vol. C-26, no. 7, pp. 667-680, 1977.

[22] P. Whyte, "*Design and Implementation of High-radix Arithmetic Systems Based on the SDNR/RNS Data Representation*" *Edith Cowan University*, 1997.

[23] Y. Zhao, J. Wickerson, and G.A. Constantinides, "*An Efficient Implementation of Online Arithmetic*", *Int. Conf. on Field-Programmable Technology*, 2016.

[24] Accellera Systems Initiative, "*Universal Verification Methodology 1.2 User's Guide*", 2015.

[25] Altera Corporation, "*Cyclone V SoC Development Board Reference Manual*", 2015.

[26] Altera Corporation, "*Memory System Design*", *Embedded Design Handbook*, 2010.

[27] Altera Corporation, "*Introduction to Altmemphy IP*", *External Memory Interface Handbook: Reference Material*, vol. 3, 2012.

[28] Altera Corporation, "*Phase-Locked Loop Basics, PLL*".

[29] Altera Corporation, "*Creating Qsys Components*", 2018.

[30] Altera Corporation, "*Cyclone V Hard Processor System Technical Reference Manual*", 2018.

[31] Altera Corporation, "*Implementing Fractional PLL Reconfiguration with Altera PLL and Altera PLL Reconfig IP Cores*".

[32] Imperial College "*An Ethics Code*", *Imperial College Research Ethics Committee*, 2013.

[33] Intel Corporation, "*Cyclone V SoC Development Kit and Intel SoC FPGA Embedded Development Suite*".

[34] Intel Corporation, "*Introduction to Intel FPGA IP Cores*", 2018.

[35] Intel Corporation, "*Avalon Interface Specifications*", 2018.

[36] Intel Corporation, "*Cyclone V Device Overview*", 2018.

[37] RocketBoards.org, "*GSRD 14.1 User manual*", 2015.

[38] Xilinx, Inc, "*Zynq-7000 All Programmable SoC*", 2018.

[39] Xilinx, Inc, "*ZedBoard (Zynq Evaluation and Development) Hardware User's Guide*", 2012.

Appendix A

User Guide Provided to Test Volunteers

A.1 Introduction

This project provides an extensible framework for at-speed evaluation of arithmetic hardware. It is currently only implemented and tested with the following environment:

Item	Version
Hardware	Cyclone V SX SoC development board
HPS System	Ubuntu 16.04.6 LTS (GNU/Linux 3.10.31 armv7l)
HPS Python	2.7.12
Quartus Prime	16.0.0.211

Table A.1: Tested Environment

To download this implementation, use
`git clone https://github.com/MerelyLogical/arithmetic-testbench.git`
It includes both hardware and software files.

A.2 Configuring Hardware

The hardware needs to be configured to fit the design before compiled and uploaded onto the development board. This guide provides the steps to this configuration with Quartus' GUI in a beginner-friendly way. It should be noted that this configuration can also be done by editing the `.qsys` file directly. The summary of this section is to first integrate the user's design into the Qsys system as a component. Then the system is compiled into synthesisable HDL. The whole system is then synthesised fitted, and then assembled into an `.sof` file. In order to upload this to the FPGA through the HPS later, it is converted to an `.rbf`, but this is optional if a different method of programming the FPGA is desired.

1. Open Quartus 16.0
2. File → Open Project... → `cy5-systest.qpf`.
3. Tools → Qsys
4. File → Open... → `soc_system.qsys`. (The open dialogue may pop-up automatically on start up of Qsys.)

5. In the IP Catalog window, click New..., which will bring out the component editor. This turns your designed components into Qsys components, which will then be integrated into the Qsys system.
6. Enter basic information of your design in the *Component* tab of the component editor.
7. In the *Files* tab, add your design files, select the top-level module and click on *Analyze Synthesis Files*.
8. Module parameters and be set in the *Parameters* tab.
9. The *Signals & Interfaces* tab should have detected all ports of your design. First ensure that the clock and the reset inputs are detected and categorised in the interface list correctly. Then to allow connection to the rest of the testbench, move the two inputs and the one output of your design into a *Conduit* interface. Associate the clock and the reset signal to the conduit. Then name the two inputs a and b, and name the output as out.
10. Finish... → Yes, Save → Yes, save before refresh.
11. The created component should now show up in the *IP Catalog*.
12. Add the component to the system, enter the desired parameter values before clicking on *Finish*.
13. Connect the signals of your design using the *System Contents* window of Qsys. The clock should be connnected to the `outclk0` signal of the component `pll_dut`. The reset signal should be connected to the `clk_reset` signal of the component `clk_ref_50M`. The conduit should be connected to the opposing conduit in the `test_wrapper` module.
14. Click on the `test_wrapper` module and configure its parameters where necessary. `NUM_SUB_MON` determines how many sub-monitors are spawned to run in parallel. `WIDTH` should match the width of your design's I/O width.
15. Fix any outstanding errors in the *Messages* window if preset.
16. Click on Generate HDL...
17. Ensure *Create HDL design files for synthesis* is not on none.
18. Generate the HDL files.
19. Click on Finish to close Qsys.
20. Quartus should prompt you to add the `.qip` file and the `.sip` file. Follow them by going to Project → Add/Remove Files to Project...
21. After making sure that the two Qsys files are included, start the compilation by going to Processing → Start Compilation.
22. By the end of the compilation, the assembler should have produced `output_files/cy5-systest.sof`. We need this in a different format, so go to File → Convert Programming Files...
23. Select `.rbf` as the output type.
24. Click on the only entry in the input files table, and click on Add File... to add the produced `.sof` file.
25. The hardware setup is completed after obtaining the `.rbf` file.

A.3 Setting up the Board

By the end of this sections, we should have the FPGA programmed and the software uploaded to the HPS, ready to start running the tests. In this guide we will be using rsync to upload the `/scripts` folder and the `.rbf` file into the board, and programming the FPGA from the HPS, but other methods are equally as valid.

1. Copy the `.rbf` file into `/scripts` and upload it with `rsync`.

   ```
   rsync -rtuv ./ ___@_____.ee.ic.ac.uk:/home/_____/scripts
   ```

2. Connect to the board. (This can be done by using PuTTY on Windows or a ssh command on Linux.)

   ```
   ssh ___@_____.ee.ic.ac.uk
   ```

3. Program the FPGA.

   ```
   cd /home/_____/scripts
   ./program_fpga.sh output_file.rbf
   ```

A.4 Running Tests

```
python -O ./run_test.py [file]
```

If no file is provided, the software will enter a REPL, where the commands available are as follows. If a file is provided, the software will read each line and execute the commands with the same syntax requirements, except for `exit` which will not work in a file.

Command	Explanation
`reset`	Resets the system and test results.
`version`	Prints the system version.
`freq <speed>`	Sets the clock speed to the specified value in MHz. Prints the actual frequency configured.
`mode <m|a>`	Choose between manual and auto test mode.
`manual <a|b> <hex>`	Give input in manual mode.
`bitset <a|b> <hex>`	Force bits to be 1 in auto mode.
`bitclr <a|b> <hex>`	Force bits to be 0 in auto mode.
`run <time>`	Runs the test for specified duration in ms. Prints the results at the end of the test.
`exit`	Exits the REPL.

Table A.2: Commands accepted in test REPL

Notes:

- Arguments in angle brackets are required. Arguments in square brackets are optional. Vertical bars separate all possible options.
- Frequency configuration is done by a PLL which has limited granularities. As such the actual frequency may differ from the desired frequency.

- Argument <a|b> is used to select which input this command will apply to.
- <hex> needs to be in the format ^(0x)?[0-9a-fA-F]{1,8}$. In words, it takes 1 to 8 digits of case-insensitive hexadecimal digits. No other characters including space or underscore are allowed, base prefix 0x is unnecessary but allowed.
- Under the current hardware environment, a safe range for <speed> in MHz is from 50 to 400.
- When the same bit is set and cleared, clear always take priority.