Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report (DRAFT)

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**

Abstract

In this project, we aim to provide an customisable evaluation framework for arithmetic hardware. Initially, the project is conceived to perform at-speed testing of a set of newly designed high-radix online arithmatic 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.

Contents

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 has the shortest cycle time per digit, it has the highest online delay and requires the largest number of cycles to complete calculations [20]. As such, the choice of binary is not absolute. In this project, I will explore high-radix online operators, investigating their suitability for FPGA implementation and examining the resultant tradeoffs between performance, area and power.

2 Background

3 Project Specification

3.1 Project Organisation

This project is 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, vertically split from the enveloping project. One shall design the arithmetic operator modules, while the other shall design a system from the top-level to test and evaluate these operators. This project deals with the system-level issues.

As this project progresses in parallel with the designing of the operator modules, it is necessary to decouple the two projects so that, being individual projects, they can be evaluated individually. The success of one project should not be restricted by the status of the other. To this end, the goal of the system-level design is more focussed on its functionalities and robustness. This relationship and its effect on the evaluation will be examined further in the evaluation chapter of this report.

To ensure the two products will work together once they are both complete, a common interface is agreed upon. The interface will be done using Qsys. The unit-level project will build different operators, which can have varying arithmetic functions and designs. These can be packaged into individual Qsys modules, as adders, multipliers, or dividers. Alternatively they can also be delivered as a single module taking two operands and an instruction that is one of the four basic arithmetic operations. These will then become the DUTs of the testbench.

3.2 Deliverables

At the end of the project, the system should be able to perform the following:

1. Connect to the arithmetic modules as its input;
2. Generate and run tests on these modules;
3. Vary the frequency of the FPGA;

4. Evaluate its performance.

3.3 Hardware Choice

The system itself will be built on a Cyclone V SX SoC Development Board from Intel [32].

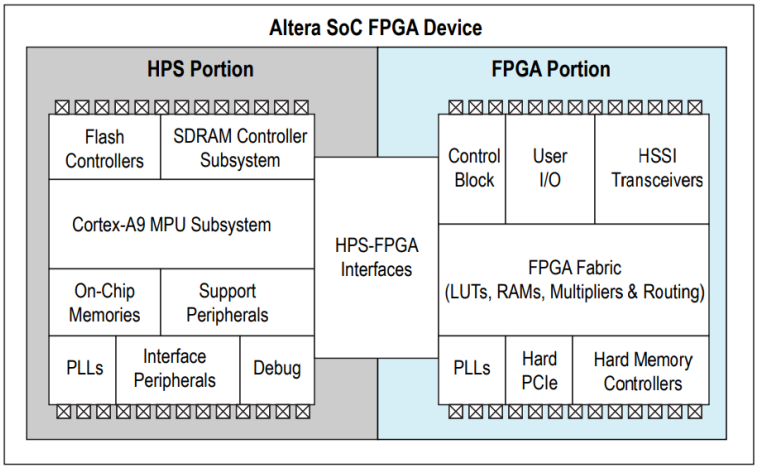

Figure 1: Structure of the System-on-Chip

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 will be enough for this project to work, having an 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 for 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 [36]. As such, a board like the ZedBoard [37] 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.

3.4 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.

While an HLS language could be used, in this design it suffers from a few problems and does not offer enough benefits to justify its use. Usually HLS is preferred for developing complex algorithms, because compilers can optimise them into RTL much better than humans. However, the resulting RTL would be unreadable, making directly controlling or debugging at the hardware level nearly impossible. The interfaces require detailed control of the actual hardware and the rest of the testbench has a lot of control path work and direct manipulation on the data bits. It is therefore not worth it to use HLS and as such, this design will be written in 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. A list of repositories on GitHub holds all files related to this project. Readme files on the repositories and the commit histories will serve as digital logbooks to this project.

4 System-level Design

4.1 Testbench Architecture

The design of the verification system is the major engineering challenge of this project. 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 testbench are not designed to be scalable or portable, serving only what they are built for. In this project, I shall use a generic structure inspired by that of an agent in Universal Verification Methodology (UVM).

Before UVM, integrated circuit designs were verified with methodologies developed independently by stimulator 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.

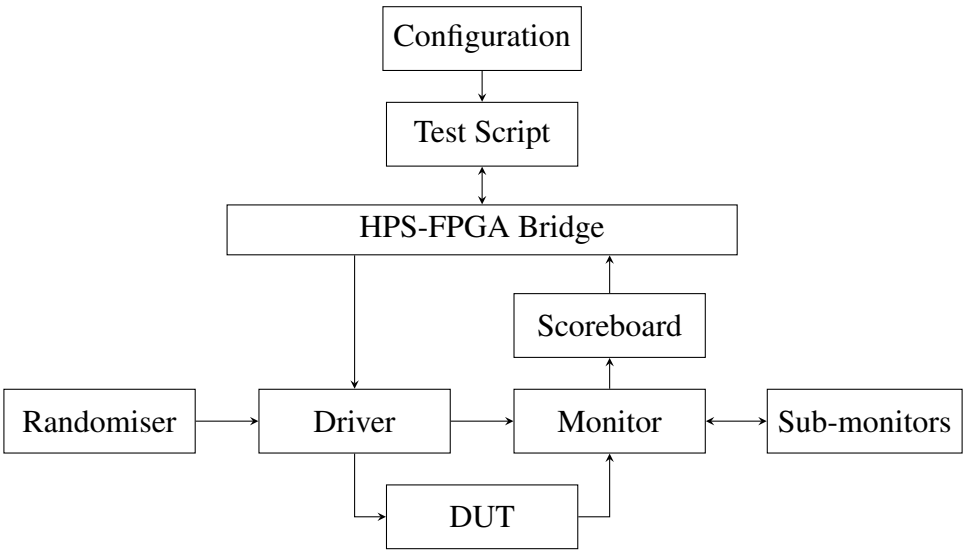

Figure 2: Block diagram of the testbench design

The test script running on the HPS will read from a configuration file, and sends the corresponding test configuration to the driver. The driver then pulls a stream of random data generated by the randomiser, and convert them to meaningful test inputs according to specification. The test output will be watched by the monitor, reporting any interesting event to the scoreboard, which keeps track of them. Sub-monitors contain reference designs functionally identical to the DUT, which the monitor uses to identify false outputs from the DUT. Multiple sub-monitors allow parallel processing, so that the reference design can have a relaxed frequency requirement. The scoreboard tallies the events 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.

4.2 User Interface

For this evaluation framework to be meaningful, it has to attract users by being easy to configure and run. To achieve this,

5 Hardware Design Choices

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 at double the frequency. This gives an ambitious target frequency of 800MHz. Assuming a data width of 32-bit, the target data transfer rate is then estimated to be 25.6Gbps. 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. While the HPS can easily generate test data with a piece

of software, there is a large amount of overhead as data crosses from one architecture to another. This overhead exists in the form of both decreased bandwidth and increased delay. 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 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 piplined 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 memory transactions can also be piplined, giving one transaction per clock cycle. 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 have store about 1GB, while on-chip memory could only hold a few MB [26]. 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 a even faster memory is desired, we can use LUTs or registers 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. As the project progress, special tests could be written and run separately with a relaxed timing restriction to cover the holes. It should be noted that certain corner cases may represent critical paths in the design. To combat this, the testbench provides the option to run handwritten inputs alongside random tests.

5.2 LFSR Randomiser

LFSRs 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 flip side, LFSRs go through every single possible value except for one before repeating itself 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 bit [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.

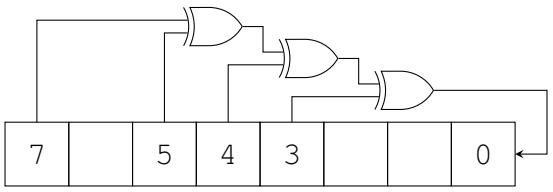

Figure 3: Fibonacci Configuration

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.

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

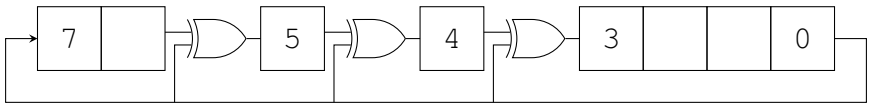

Figure 4: Galois Configuration

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 XOR gates working independently. On an FPGA, the cascade of gates is usually implemented with a LUT, so while limiting LUT input to 2 might have some minor improvements, this increased delay of the Fibonacci configuration should not be obvious.

In terms of implementation, Fibonnaci LFSRs are slightly easier to code if width configurability 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.

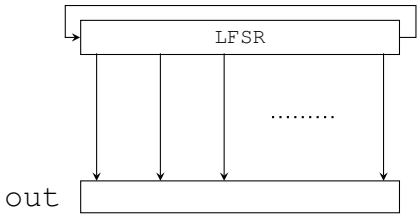

Figure 5: Horizontal Structure

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

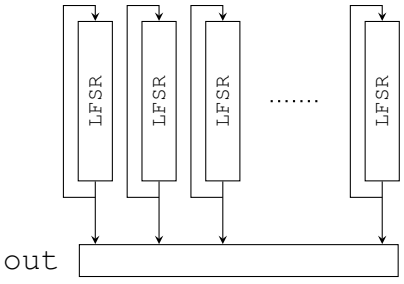

Figure 6: Vertical Structure

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 less 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 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.

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

As such, there is no real advantage of using the vertical structure. With truncation providing the configurability in the horizontal structure, the slight advantage of the Fibonacci LFSRs in its ease to write is nullified. Having the slight advantage in terms of speed for Galois LFSRs, they will be chosen as the randomiser design in this project.

5.3 Driver

The output of the driver is connected to two modules. It feeds test input to the DUT and the monitor.

5.3.1 Filtering Input

One driver focusses on fast stress tests, The other allows handwritten tests to coexist with random tests. They can be switched in software.

5.3.2 Synchronised Monitor Inputs

The driver has the responsibility in ensuring the monitor receives the test output from the DUT and the test inputs at the same time. After going through the filtering logic, the stream of test input 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 number of cycle delay 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 on the FPGA. At a minimum, there need to be two clock domains: one surrounds the DUT and another supports the rest of the control logic around the DUT. These clock frequencies can be generated with PLLs, which are provided as IP Cores in the Quartus software [28]. A clock tree will distribute them to the individual modules. Data crossing clock domains will be fed through FIFOs to prevent loss.

The proposed structure will have the bulk of the control logic running in a separate clock domain to the DUT. Only an interface with FIFOs will be running in synchronicity with the DUT. Therefore, the test controls can run at a slower frequency without bottlenecking the system, allowing the DUT to be stressed further. The problem now is to ensure the monitor can handle the stream of DUT output coming in at a higher frequency that it is running at. As the monitor needs to calculate the correct data before it can check if the DUT output is correct, it cannot keep up with the speed of the DUT. This report consider three alternatives.

5.4.1 Partial Monitors

A lightweight idea is to implement a parity checker instead of a full model inside the monitors. For example, to check an adder, the monitor can just check if the final bit with a LUT acting as a XOR gate.

Although this is reasonably fast, it cannot be extended once the DUT is faster than a parity calculation followed by a comparison. More critically, it provides no additional information once the DUT fails, and it has a 50% rate of ignoring an error. If this is to be solved by increasing the number of bits checked, the problem returns back to its initial state. Thus this method will not be experimented.

5.4.2 Lazy Monitors

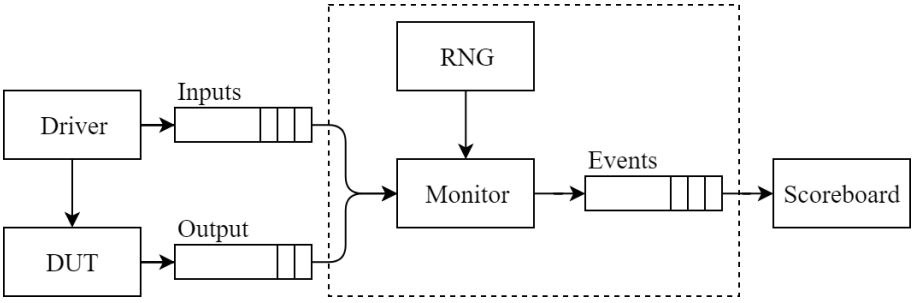

Figure 7: Structure of a Lazy Monitor

An 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. In 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

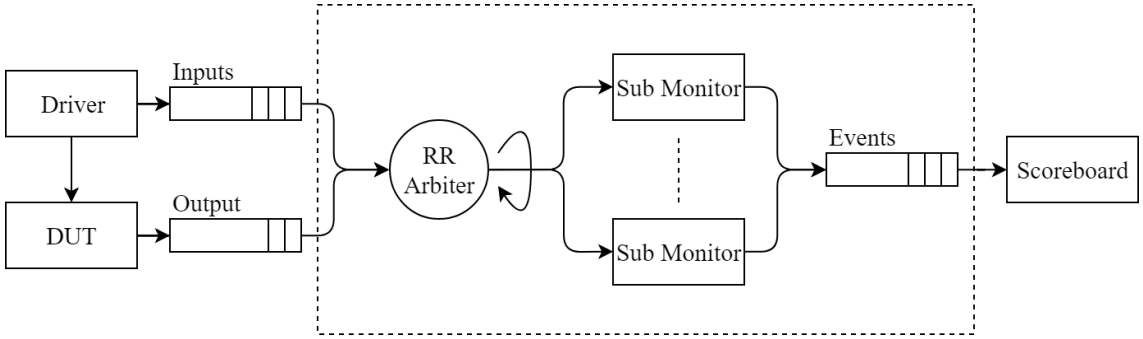

Figure 8: Structure of a Parallel Monitor

As the test data is uniform, the monitor can be parallelised in to a number of sub-monitors. The sub-monitors is connected to a arbiter that is connected to three FIFOs. The FIFOs are the inputs and the output of the DUT. A round robin arbiter distributes the data to the sub-monitors equally. The results from each sub-monitor are then sent to a single scoreboard. To avoid potential hazards, the output from the sub-monitors will be buffered before processed by the 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 it the DUT fills up its output buffer. 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 built first for this project, as it offers the best functionalities. Although unlikely, if a fatal issue arises in this design, or if the testbench needs to be more lightweight, then the lazy monitor will be used as the alternative.

5.5 Scoreboard

If the monitor detects an interesting event such as an error, it will send a message to the scoreboard. The scoreboard has counters tracking these events, which are exposed and can be read by the HPS.

The software can run statistics to provide further insights to the user.

6 Software Design Choices

6.1 Code Structure

7 Hardware Implementation

7.1 Project Hierarchy

Before any engineering work was done towards the final product, a small module was built to learn the environment. This module needed to be simple yet covered enough grounds to provide as much learning during the process as possible without taking up too much actual development time towards the product. As the greatest unfamiliarity was with the interaction across the HPS-FPGA bridge, a simple hardware accelerated adder was made for this training.

7.1.1 FPGA Side

Programming the FPGA to communicate with the HPS is no trivial task. Luckily, there exists a golden system reference design [35] for the board in use for this project. Unfortunately, support for certain versions of Quartus are missing from the GSRD download database, including the one 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.

From the perspective of the FPGA, The HPS exposes three bridges for connections [30]. As this is a relatively simple task, the lightweight bridge was used. Module `altera_hps` exposes the master of this bridge as `h2f_lw_axi_master`. Next, the actual hardware adder was built and integrated as a hardware module in Qsys with a matching interface. A simple adder can produce its result after one clock cycle. This greatly simplifies the logic required for the Avalon slave interface. The logic for the control and data signals were thus written according to the interface specifications [34]. Following the naming conventions for the signals allows Qsys Component Editor to automatically detect the Avalon slave from this module at analysis. This saves the troubles of editing the `_hw.tcl` file. To experiment with module configuration, the adder was designed with variable width.

The adder was then instantiated and connected to the rest of the system with two clicks in Qsys. From there, Qsys could generate the HDL for the entire system, which was then compiled to a bitstream file. With the bitstream ready, the work now shifted to the HPS.

7.2 Randomiser

7.3 Driver

7.3.1 Delay Tester

I built a delay tester to find out 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.

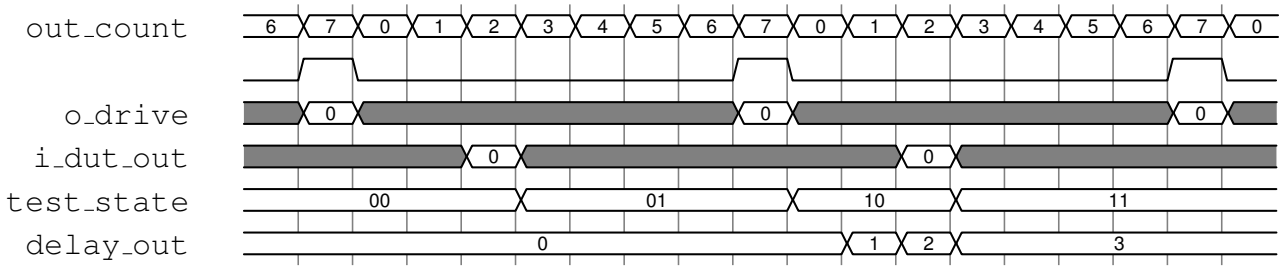

Figure 9: 3-bit Delay Tester FSM

Testing with 0 is safe since LFSR will never output 0.

7.3.2 Switching system

7.4 Monitor

7.4.1 Sub Monitors

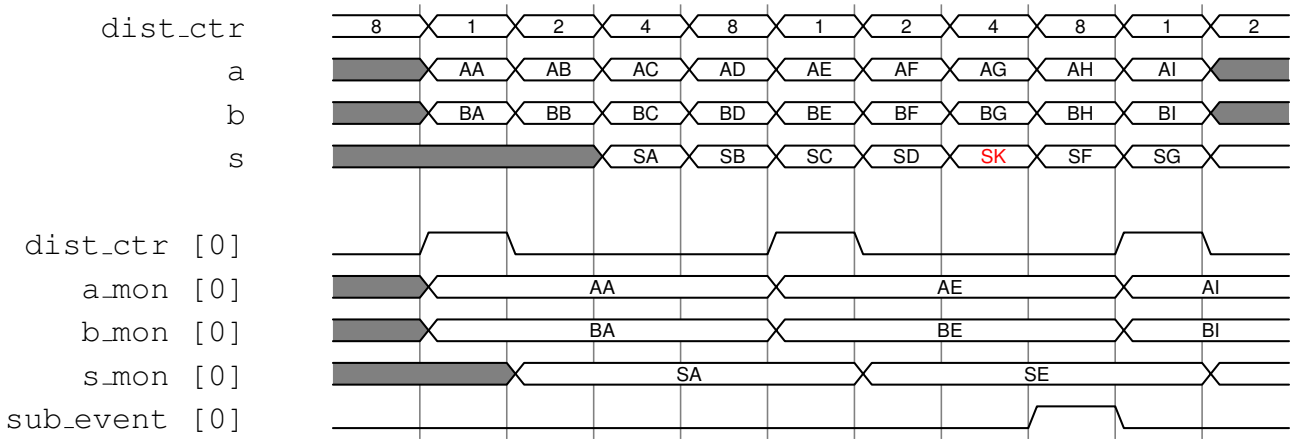

Figure 10: Distributed Monitoring System

7.5 Scoreboard

8 Software Implementation

8.0.1 HPS Side

The HPS runs Ubuntu and a Bash script has been written to load the bitstream onto the FPGA. Next, a program was written in Python to test the hardware design from the HPS. The interfaces are mapped onto the physical memory, thus they can be accessed by opening `/dev/mem`. Checking against the specifications [30], the lightweight master is at `0xFF20_0000`. Qsys allocates the memory spaces of modules relatively, so when it reports that the adder has been placed at `0x0010_0000`, it is physically at `0xFF30_0000`. The adder was designed to have its two inputs at `0x00` and `0x10` and its output at `0x20`, which were assigned by Qsys relatively to `0xFF30_0000`.

With the memory mapping understood, the program was designed to closely mirror this relative relationship between the modules using classes. For example, the adder defines its output at `0x20`, but its read and write methods are inherited from an AXI class that brings it to the correct physical address by adding the address of the bridge defined in it. This parallel between software and hardware should be helpful as the product gets more complex.

To verify what I have built and learnt was correct, 1000 add operations were executed separately with and without the hardware acceleration of the FPGA. The results were compared and confirmed that this training module worked. While called hardware acceleration, the FPGA actually had worse performance than the HPS in this testing case. The CPU is reasonably efficient in calculating additions, while calculating on the FPGA requires the adder I/O data going through the HPS-FPGA bridge. This incurs a significant overhead, thus slowing it down.

9 System Integration

9.1 Qsys

10 Testing

11 Results

12 Evaluation

12.1 Product Metrics

12.1.1 Robustness

The next few measures looks at 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. As the main quantitative metric, this will be a vital indicator of the project's success.

12.1.2 Flexibility

The flexibility of the testbench is also vital to the product's performance. The testbench should suitable for a range of DUTs. This will allow the testbench to be used for future experiments. The flexibility can be measured by the number of configurable parameters that it has, and the range of which these parameters can be adjusted to.

12.1.3 User-friendliness

The ease of use of the testbench can be another evaluation point. On the hardware side, the verification system can be packaged into a Qsys module. Given the DUT is also a module with an agreed interface, it can easily be connected in Qsys for testing. On the software side, a user-friendly interface could be built. A usable command line interface may be good enough, but a simple graphic interface could make the tests much more visual and informative.

The interface could also provide information on the failure in the DUT. A better testbench will provide more insightful details when the DUT fails. This would make debugging or evaluating the design much simpler. Along with the GUI, this project has many optional extensions that would be discussed further in section 6.2. After the main goal of the project being met, the number of optional functions implemented becomes a good measure of the progress of the project.

12.1.4 Note

A noteworthy point in evaluation is regarding the progress of the sister project. The purpose of the testbench is to verify and stress arithmetic designs. If these designs are not available near the end of this project, it would be difficult to empirically prove the capabilities of the testbench and its surrounding system. Since the project comprises a verification system, the results from the benchmarks should not be used for evaluation of this project. It is not impossible, as there are still substitutes for them. For functional purposes, standard off-the-shelf arithmetic modules could be used in-lieu. For other purposes, it is possible to have a model done before the actual design starts in the paired project. While this would allow this project to progress, it would be extra work for the other project. In all, it would be nice to have a solid arithmetic module completely to run in this testbench, but without one, the system can still be built and completed, albeit generating less useful data towards the overall aim of the project.

12.2 Project Metrics

12.2.1 Implementation Plan

One natural way of measuring the success of the project is to look at the actual progress and compare it to the implementation plan. It should be noted that no plan is perfect, so some deviation is allowed. However, if there is significant delay from the plan, there must be justifications given.

12.2.2 Fallbacks and Extensions

The fallbacks and extensions were detailed in the implementation plan. The progression of these extension tasks can be used as a point of evaluation to the project. This is already reflected in the product metrics, as 6.1.2 and 6.1.3 examines mainly the success of the extension tasks, while only 6.1.1 analyses the achievements of the core task. However, this does not mean that this project will be completely sealed if all extension tasks are complete before the final deadline, there is always more potential for future work.

In the very unlikely case that the project progresses to such a state, new ideas such as dynamic voltage scaling will be drafted and evaluated.

13 Conclusion

14 Further Work

15 User Guide

A A Brief Introduction to Online Arithmetic on FPGA

A.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.

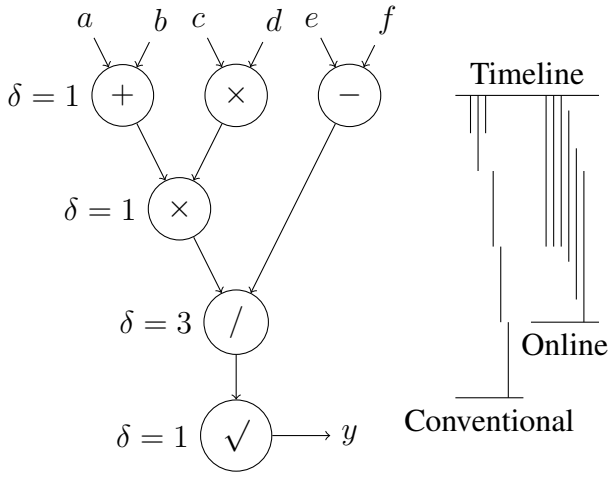

Figure 11: Computing $y = \sqrt{(a+b)cd/(e-f)}$ with serial online operators [8]

As illustrated in figure 11, 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 uses 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 run-time [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].

A.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 increasing. 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 saving 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 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 is 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 FPGAs contain small fast carry-ripple adders, high-radix adders may be able to exploit them to obtain significant speedups [11].

A.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], We take a more direct approach with this project by implementing 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.

Furthermore, benchmarking this exotic arithmetic system with popular FPGA applications such as neural networks would be interesting, as there is not much precedence for it.

References

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

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

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

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

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

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

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