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**

Chapter 1

Hardware Implementation

1.1 Project Hierarchy

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 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.

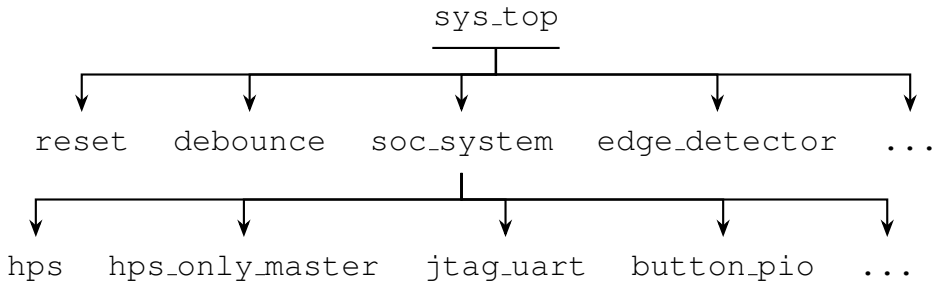

Figure 1.1: Hierarchy of the Golden Reference Design

By examining the structure of the reference design, we see that it has a top-level wrapper called the `sys_top`, which instantiates the Qsys system `soc_system` and a few IP blocks that handles 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`, cooresponding 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 and written to, the logic required for the signals on the Avalon slave interface can be handwritten according to the interface specifications [34] without much trouble.

By following the naming conventions 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. This is done in the module `text_wrapper`.

Since Qsys is chosen as the user interface, 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 connects to the DUT and the references designs.

In addition, the PLL and its reconfiguration module are also instantiated at this level since as packaged IP cores, Qsys is designed for such integrations.

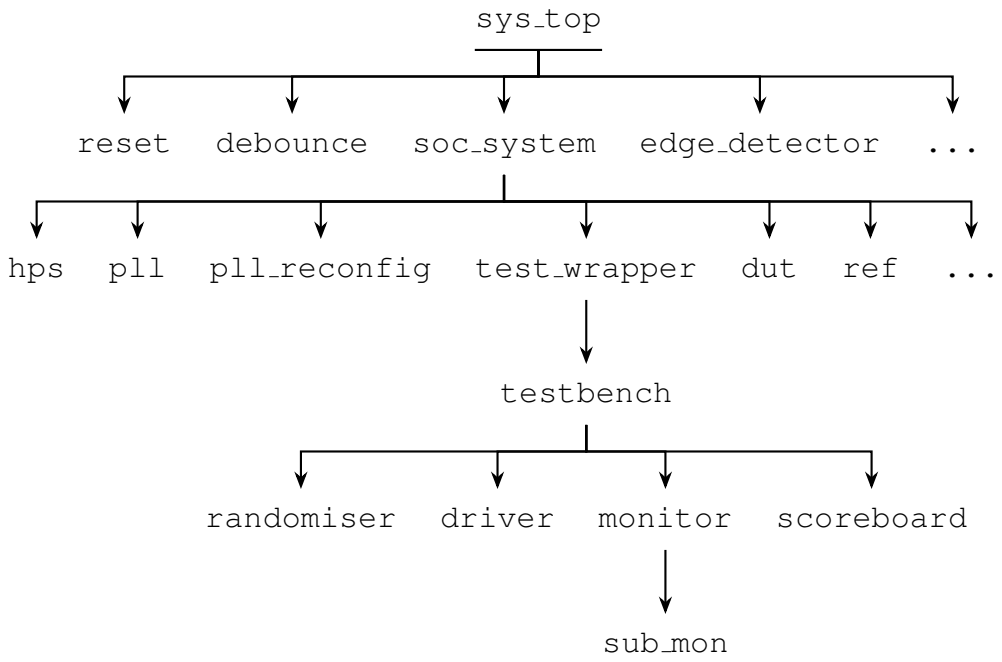

Figure 1.2: Hierarchy of the Full Hardware System

The main section of the testbench is instantiated in the wrapper as a separate module aptly 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 however, 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 rendered as a Verilog file that has its direct

dependencies contained in that folder. While this is arguably a benefit in terms of dependency management, 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, it can cause confusions as the testbench could be still on the old iteration even though a new compilation at the `sys_top` level has been performed.

Secondly, it makes the simulation of the testbench more straight forward. 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 side of things in simulations.

1.2 Randomiser

Implementing the randomiser is straight forward. A possible set of taps for a 32-bit Galois LFSR is [32, 30, 26, 25]. The logic is then to XOR the bits left of the taps with bit 0, and simple right shift for all other bits. For driver to control the randomiser, an enable signal and an initial signal is added as input in addition to clock and reset. The initial signal seeds the LFSR.

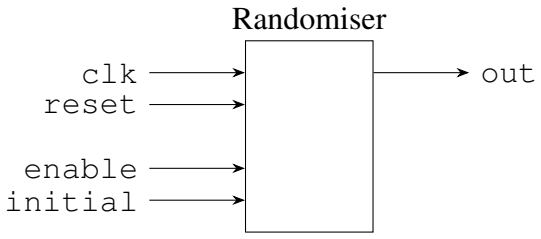

Figure 1.3: 3-bit Delay Tester FSM

1.3 Driver

1.3.1 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.

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

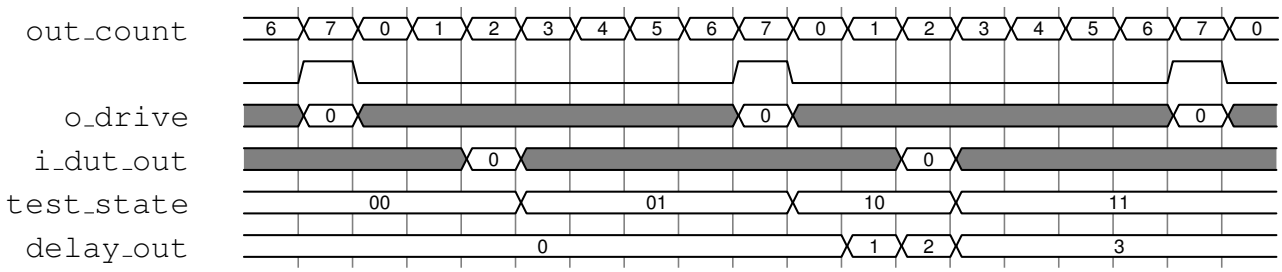

Figure 1.4: 3-bit Delay Tester FSM

1.3.2 Switching system

1.4 Monitor

1.4.1 Sub Monitors

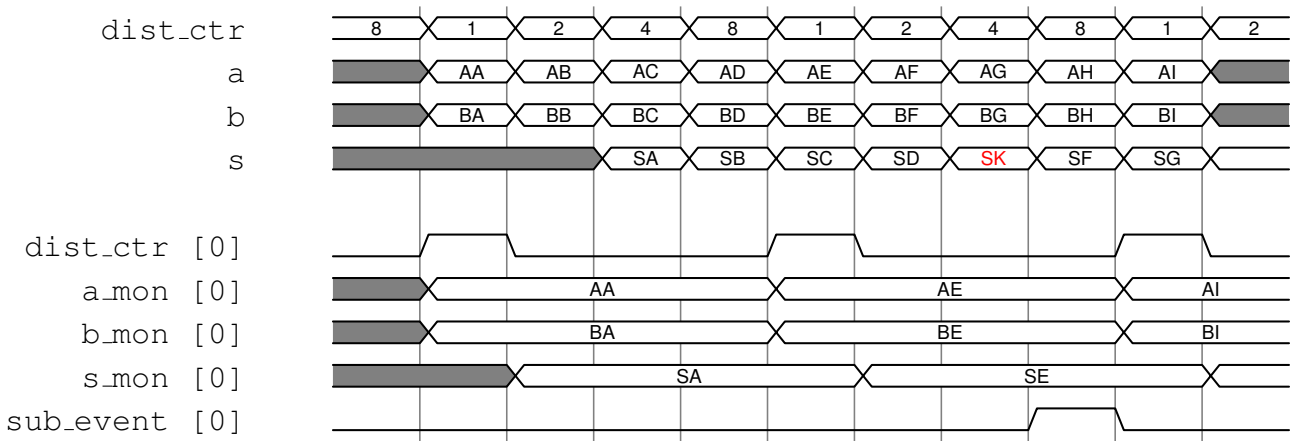

Figure 1.5: Distributed Monitoring System

1.5 Scoreboard

Chapter 2

Software Implementation

2.0.1 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.

2.0.2 newstuff

Most interactions through the HPS-FPGA bridge involves writing and reading different memory locations.

the software is mainly a collection of functions, each performs a series of reads and writes. This

means the final test procedure would be easier to

Chapter 3

System Integration

3.1 Qsys

Appendix 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.

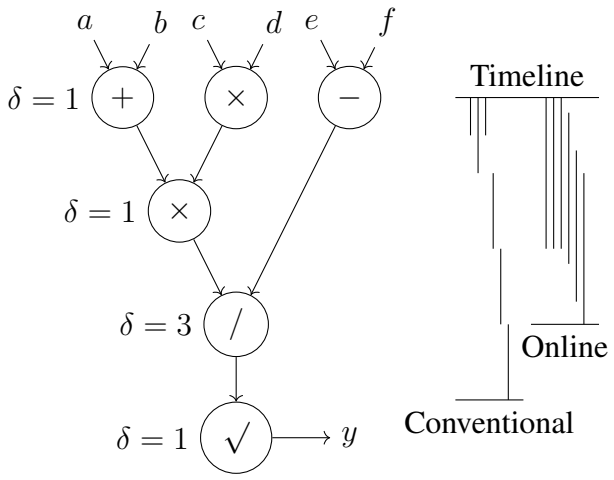

Figure A.1: Computing $y = \sqrt{(a+b)cd/(e-f)}$ with serial online operators [8]

As illustrated in figure A.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 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 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].

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 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 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.

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] 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.