Graceful Degradation on FPGAs: A High-Radix Online Approach

Joshua A. Laney
Bachelor of Science

A Thesis submitted in fulfilment of requirements for the degree of
Master of Science
Analogue and Digital Integrated Circuit Design
of Imperial College London

Department of Electrical and Electronic Engineering
Imperial College London
August 27, 2021

Abstract

In this thesis I test the overclocking performance and degradation of high-radix online arithmetic on an Intel Cyclone V FPGA. This is motivated by the notion that high-radix implementations can better utilize the FPGA micro-architecture than the more traditional radix-2. Data is collected on radix-2,4,8 and 16 online adders of equivalent widths from 16 to 128-bits. Radix-2 and radix-4 online multipliers of equivalent widths from 8 to 64-bits are also tested. Additionally, the results are compared to traditional arithmetic implementations for control. It is found that for online adders of sufficiently high width, radix-4 exhibits more desirable performance and degradation characteristics than radix-2. This trend does not continue for radix-8, where performance steeply degrades. Due to the core characteristics of online multiplication, specifically partial product generation and digit selection, it is found that higher radices provide no benefit over radix-2. As such higher-radix online implementations are only advisable in arithmetic pipelines consisting largely of addition.

Acknowledgment

I would like to thank Dr. James Davis for his clear guidance, patience, and impactful contributions throughout this learning experience. Dr. He Li and Professor Ercegovac for their help on the mathematics. Dr. Judith, Hyrum, and William Laney for their unwavering support and advice. Dr. Peter Wolfe for enabling my studies. Adam Turflinger for serving as my rubber duck and sage compatriot. And all my old friends back home, and new friends in London, for keeping me grounded and reminding me to have some fun.

On a final, more personal note, this thesis serves as the culmination of 18 uninterrupted years of schooling. With my education having earned the right to vote, I intended to cease my formal learning, and transition to applying myself in industry. With that said, I am without doubt that the lessons learned in the completion of this research will serve invaluable throughout my life. Beyond the directly applicable skills like Verilog, timing constraints, and the Quartus tool chain, I have honed my skills in critical decision making, analysis, articulation, and ownership of complex, multifaceted systems. As such, I proudly present this thesis not only for its contributions to the space of digital arithmetic, but as the flag atop this personal peak of life achievement.

Contents

Abstract **2**

Acknowledgment **3**

Contents **4**

List of Figures **6**

List of Tables **7**

Abbreviations **8**

Chapter 1. Introduction **9**

Chapter 2. Online Arithmetic **11**

2.1 Addition 12

2.2 Multiplication 15

Chapter 3. Experimental Setup **19**

3.1 Testing Strategy 19

3.2 Arithmetic IP Development and Verification 22

3.2.1 Adder 22

3.2.2 Multiplier 22

3.3 Hardware Test Bench 23

3.3.1 Framework 23

3.3.2 Architecture 24

3.3.3 Constraints and Debugging 25

Chapter 4. Results and Analysis **28**

4.1 Addition 30

4.1.1 Speed 30

4.1.2 Degradation 32

4.1.3 Area 37

4.2 Multiplication 38
4.2.1 Speed 38
4.2.2 Degradation 40
4.2.3 Area 41
4.3 Meta-analysis 42

Chapter 5. Conclusion **46**

Bibliography **49**

List of Figures

2.1 Online adder of general radix. Red arrows illustrate a critical path. Figure adapted the Digital Arithmetic textbook [1] 13
2.2 Radix-2 Online adder. Red arrows illustrate a critical path. Figure adapted from the Digital Arithmetic textbook [1] 14
2.3 Online Multiplier, figure inspired by Kan Shi [2] 17

3.1 High level block diagram of hardware test bench. Note the orange $PLL/2$ line contains both the 0 and 180 degree phase clocks for visual clarity 25
3.2 Timing related modifications made to the test bench to account for multiple clock domains 27

4.1 Comparison between the RTL usage of radix-2 and radix-4 online adders of 4-bit equivalent width 31
4.2 Comparison between the average MRE behavior of adders at different equivalent widths . 33
4.3 Comparison between the average MRE behavior of adders at radix-2 and radix-4, set to the same scale. 34
4.4 The average absolute error and average MRE of the measured 64-bit equivalent adders . 35
4.5 Additional performance metrics of the 128-bit equivalent adders. 36
4.6 Comparison between the average MRE behavior of multipliers at different equivalent widths 40
4.7 Comparison between the counts of illegal digits of multipliers at different equivalent widths 41
4.8 MRE of 64-bit control adders generated with different seeds 44

List of Tables

4.1 Width and radix of tested adders 28
4.2 Width and radix of tested multipliers 29
4.3 Measured F_{real} of adders. Frequencies given in MHz. 30
4.4 Tool reported number of LUTs required by each tested adder. 37
4.5 Area-Delay product of tested adders. Values given in LUT-μs. 37
4.6 Measured F_{real} of multipliers. Frequencies given in MHz. 38
4.7 Number of pipeline stages and latency of tested multipliers. Given in stages, μs 38
4.8 Tool reported number of LUTs required by each tested multiplier. The control multipliers are given in LUTs, DSPs. DSP usage is 0 for all online multipliers. 41
4.9 Area-Delay product of tested multipliers. Values given in LUT-μs. The control multipliers are given in LUTs-μs, DSPs-μs. DSP usage is 0 for all online multipliers. 42
4.10 Statistical F_{real} of various arithmetic IP generated with 10 different seeds. All frequencies reported in MHz 43

Abbreviations

ALM: Adaptive Logic Module

ASIC: Application Specific Integrated Circuit

BIST: Built in Self Test

BRAM: Block Random Access Memory

DSP: Digital Signal Processing

DUT: Design Under Test

FPGA: Field Programmable Gate Array

HDL: Hardware Description Language

HPS: Hard Processing System

IP: Intellectual Property

LFSR: Linear Feedback Shift Register

LSD: Least Significant Digit

LUT: Look Up Table (specifically the FPGA block and not the general concept)

MISR: Multiple Input Signature Register

MSD: Most Significant Digit

PLL: Phase-locked Loop

STA: Static Timing Analysis

TCU: Test Control Unit

Chapter 1

Introduction

FIELD programmable gate arrays (FPGA) are a promising platform for embedded, latency-sensitive applications. These ever-shrinking latency requirements, however, have begun to conflict with the maximum clock speeds guaranteed by (typically overly conservative) static timing analysis (STA) tools, F_{STA}, making some applications impossible to implement on the target device. These tools add considerable overhead to mitigate the possibility of short term effects such as noise, and long term effects such as full circuit breakdown impacting the performance of the design. As such, FPGAs can be overclocked (running it at a clock speed faster than that reported by STA) to a real maximum frequency, F_{real}, before degradation occurs. Furthermore, past research has theorized and proven that by overclocking an FPGA even beyond this point, a reduction in latency can be achieved at the cost of accuracy within the data [3] [4]. This hit in accuracy comes as arithmetic blocks fail, causing bit errors in the results. In traditional arithmetic blocks, propagating carry signals from the critical path in the system. These carry chains cause bit errors to occur at the most significant digits (MSD) of the output, resulting in a maximal reduction in accuracy.

Online arithmetic solves this issue by reversing the order of the arithmetic operation, resulting in a graceful degradation of the output beginning in the least significant digits (LSD). This unique style of arithmetic works on the principle of representing inputs and outputs with redundant digit sets, constraining carry propagation to a constant value

independent of bit length [5]. When utilized in overclocked FPGAs, online arithmetic has been shown to reduce the error (increase the accuracy) when compared to traditional methods [2]. This makes it particularly useful for implementing low-latency, error-insensitive applications such as real-time controllers and signal processing [6] [7].

While in principle the use of online arithmetic provides clear performance gains, the physical design of FPGA soft logic results in inefficient implementations. These inefficiencies motivated previous attempts at tailoring online architectures to FPGA platforms, reducing resource utilization and increasing latency [8] [9]. In these previous optimization attempts, the architectures operate on numbers encoded only in radix-2. In a 2015 paper, Dr. Peter Kornerup shows that in the case of application specific integrated circuits (ASIC), radix-2 encoding provides the fastest implementation [10]. However, Kornerup goes on to hypothesize that higher radix recodings could result in performance gains in the specific case of FPGAs due to the presence of highly optimized, traditional adder and digital signal processing (DSP) architectures within the fabric.

In this Master's thesis I set out to find an answer to Kornerup's hypothesis on the effects of high-radix online arithmetic on the performance of overclocked FPGAs. This includes comparisons of both online addition and multiplication across multiple radices, and word lengths. These implementations are assessed based on their measured, true, maximum clock speed (F_{real}), as well as the 'gracefulness of degradation', or how quickly errors in the result grow as the F_{real} is exceeded. These results will also be compared to a control of traditional 2's complement arithmetic implementations.

Before continuing it is worth taking time to clearly define what I mean by 'online arithmetic'. In its most traditional and original definition, online arithmetic refers solely to *serial* MSD-first implementations. However, within the Circuits and Systems group within Imperial College London, online arithmetic is commonly used to also describe *parallel* MSD-first implementations. This whole class of MSD-first units can also be referred to as 'signed digit' arithmetic, as first defined by Dr. Algirdas Avizienis in 1961 [11]. Here it is important to note that this paper focuses on the implementation of parallel MSD-first arithmetic, and all three descriptors will be used interchangeably.

Chapter 2

Online Arithmetic

ONLINE arithmetic, as originally introduced by Avizienis, and extensively studied and written about by Dr. Milos Ercegovac, hinges on the concept of redundancy [1] [11] [12]. In this form of arithmetic, digits are permitted to take on both negative and positive values, and form what is referred to as a redundant digit set. This project exclusively uses maximally redundant digit sets as defined in Eq. 2.1, where x is the digit, and r is the radix. While the encoding of the digits themselves is ultimately up to the designer, for this project each digit is encoded in 2's complement format. I chose to use 2's complement under the assumption that at higher radices, this encoding would best facilitate the use of the FPGA's hard-baked adders.

$$x \in \{-r+1, ..., -1, 0, 1, ..., r-1\} \tag{2.1}$$

Taking these digits, a full number, X, can then be represented as given in Eq. 2.2.

$$X = \sum_{i=0}^{i=N} x_i \times r^i \tag{2.2}$$

From this, the definition of redundant becomes clear: the same value can be represented with multiple different encoding. For example in radix-10, the encoding $21\bar{8}1_{10}$ is equivalent to $3\bar{9}\bar{7}\bar{9}_{10}$, which in turn both have the non-redundant value of 2021. This ability to represent the same value multiple ways is the key to enabling MSD-first operations

to work. Best guesses can be made for the higher significant digits, and then the digits of lower significance can be used to correct for those guesses, resulting in the correct value.

2.1 Addition

In the general case, online addition is a two-step process. First, intermediate encodings are calculated based on the algorithm in Eq. 2.3 with the constraint of Eq. 2.5 [1]. These intermediate encoding are then added together to result in the sum, seen in Eq. 2.4. The purpose of first generating these intermediate encodings is to assure no carrying will occur in the second summation step. As such, the carry chain is reduced to a constant value of 1-digit place, as illustrated in Fig 2.1. This is in contrast to traditional arithmetic where a right to left carry chain of length equal to the width of the addends is required. An especially advantageous property of this addition is that beyond even MSD-first, all resultant digits are calculated at the same time in parallel.

$$(t_{i+1}, w_i) = \begin{cases} (0, x_i + y_i) & \text{if } -a+1 \leq x_i + y_i \leq a-1 \\ (1, x_i + y_i - r) & \text{if } x_i + y_i \geq a \\ (-1, x_i + y_i + r) & \text{if } x_i + y_i \leq -a \end{cases} \tag{2.3}$$

$$s_i = t_i + w_i \tag{2.4}$$

$$a = r - 1 \geq \frac{(r+1)}{2} \tag{2.5}$$

As it turns out, radix-2 addition violates the constraint given in Eq. 2.5. To compensate for this fact an additional intermediate step must be added where the addition is 'double recoded' into an intermediate digit set of $\{-2, -1, 0, 1\}$ before than being transformed back into the fully redundant digit set of $\{-1, 0, 1\}$, as described in Eq. 2.6 and Eq. 2.7. In terms of implementation, this results in a carry propagation of 2-digit places as seen in Fig. 2.2, resulting in more complexity than higher radices, but still clearly

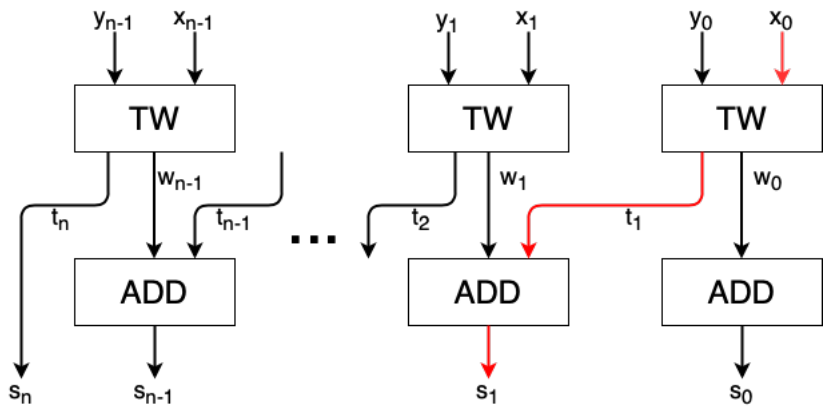


Figure 2.1: Online adder of general radix. Red arrows illustrate a critical path. Figure adapted the Digital Arithmetic textbook [1]

advantageous when compared to traditional methods.

$$(h_{i+1}, z_i) = \begin{cases} (0, x_i + y_i) & \text{if } x_i + y_i \leq 0 \\ (1, x_i + y_i - 2) & \text{if } x_i + y_i \geq 0 \end{cases} \tag{2.6}$$

$$(t_{i+1}, w_i) = \begin{cases} (0, z_i + h_i) & \text{if } z_i + h_i \geq 0 \\ (-1, z_i + h_i + 2) & \text{if } z_i + h_i \leq 0 \end{cases} \tag{2.7}$$

Another quirk of this online addition is what I call 'false overflow'. Take for example the radix-10 addition of 400_{10} and 500_{10}. Using Eq. 2.3 and Eq. 2.4, the sum is found to be $1\bar{1}00_{10}$, which has an equivalent value of 900. Here, the sum produced is in redundant form, and has overflown in width by 1-digit, but an equally valid encoding for this sum is 900_{10}, in which the width remains unchanged from addends. The result grows in width due solely to the requirements of the online algorithm, and not out of a necessity for increased data representation like in traditional addition. How this false overflow is dealt with comes down to an application specific design decision, but I will present three main techniques. The first is to simply accept false overflow as a possibility, and have sums grow by 1-digit after every addition. This solution makes particular sense if adders must be able to accommodate traditional overflow as well, and is what I developed for the experiments covered in this thesis. Note that false overflow will never result in 2 additional digits of

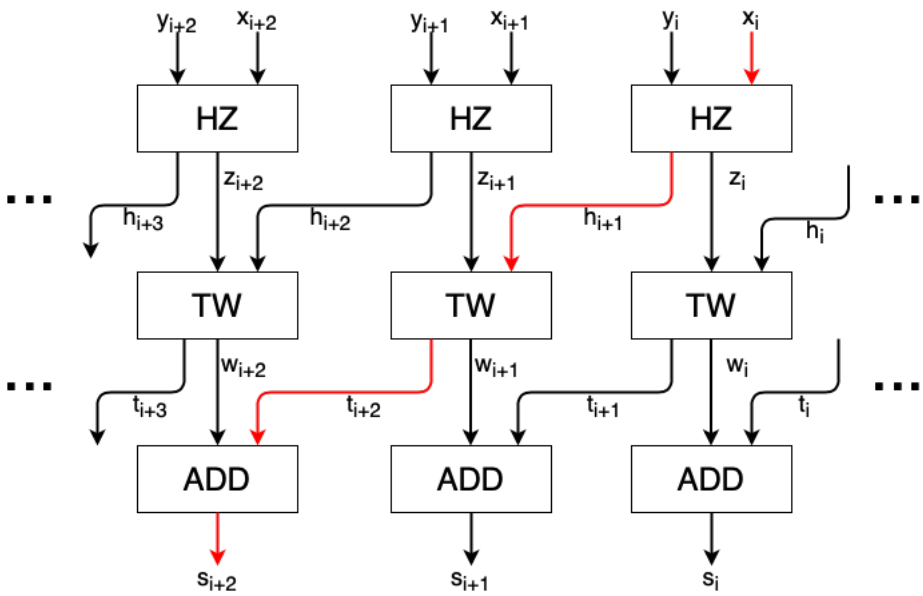


Figure 2.2: Radix-2 Online adder. Red arrows illustrate a critical path. Figure adapted from the Digital Arithmetic textbook [1]

growth, so having the sum 1-digit wider than the addends will capture all possible results. The second solution is to simply further constrain the input to the adder to ensure false overflow never occurs. This is done by making sure the MSDs of the addends adhere to the $t_i = 0$ case of Eq. 2.3. The final option is to 'compress' the output back into the original width. This requires additional logic to convert the overflown, redundant representation into a different, narrower redundant representation of the same value. In the case of parallel online addition, this introduces a left to right N-digit carry chain, degrading overall performance (but maintaining MSD-first operation). In the case of serial online addition this operation is practically free, as the carry gets buffered between each clock cycle.

While a variety of research has been done to improve the designs of these online adders, both in general [13] [14] and specifically for FPGAs [9], I opted to implement the most basic architecture for the sake of simplicity and overall fairness within the experiment.

2.2 Multiplication

Online multiplication is expectedly somewhat more involved than addition, and is built on the sequential, residual algorithm, outlined in Alg. 1 (adapted from Ercegovac [15]) that itself includes multiple online addition operations. The details of the mathematics behind this algorithm are out of the scope of this thesis, but the implications of the algorithm as presented will be thoroughly explored [5]. For all multipliers tested, I use an online delay of $\delta = 3$, following the lead of Chapter 9 of the Digital Arithmetic textbook [15].

Algorithm 1 Online multiplication of radix-r

1. [Initialize]
  $X[-3] = Y[-3] = w[-3] = 0$
  for $j = -3$ to -1 **do**
    $v[j] = r \cdot w[j] + (X[j] \cdot y_{j+4} + Y[j+1] \cdot x_{j+4})r^{-3}$
    $w[j+1] \leftarrow v[j]$
  end for
2. [Recurrence]
  for $j = 0$ to $N - 1$ **do**
    $v[j] = r \cdot w[j] + (X[j] \cdot y_{j+4} + Y[j+1] \cdot x_{j+4})r^{-3}$
    $p_{j+1} = SELM(\widehat{v[j]})$
    $w[j+1] \leftarrow v[j] - p_{j+1}$
    $P_{out} \leftarrow p_{j+1}$
  end for

For simplicity of understanding, the multiplication is considered to be fixed-point, with the input factors and output product being confined to the range of $(-1, 1)$. The internal residual vectors $w[j]$ and $v[j]$ are considered to also include 3 integer digits. $X[j]$ and $Y[j]$ are subsets of the multiplier and multiplicand, following the definition in Eq. 2.8. While $w[j]$ and $v[j]$ share similar notation, they simply indicate the jth iteration of the vector.

$$X[j] = \sum_{i=1}^{i=j+3} x_i * r^{-i} \tag{2.8}$$

Single digits are denoted by x_j, y_j, and p_j, where x_1 is the MSD and x_N is the LSD. As such, $X[j] \cdot y_{j+4}$ can be thought of as a partial product. N is the precision of the multiplication, and is chosen to be the digit-width of the input factors. The algorithm

produces the single precision result P_{rough}. The full, double-precision result, P_{full}, of the multiplication can then be computed with Eq 2.9.

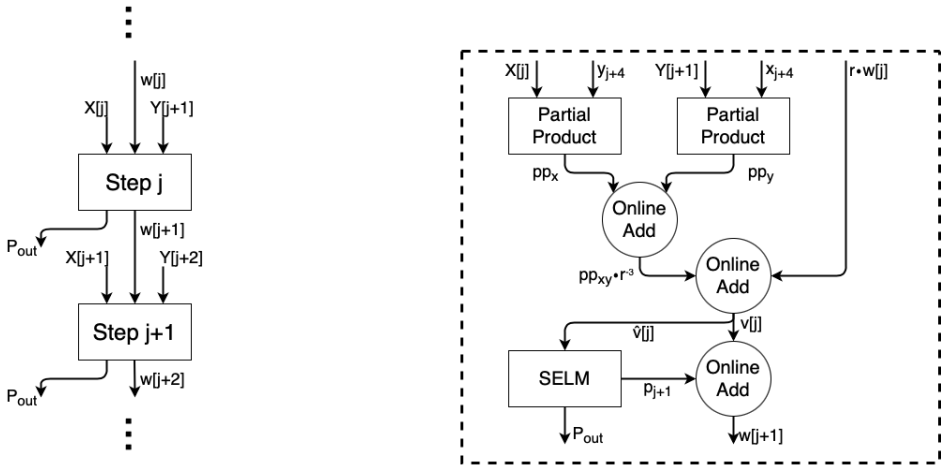
$$P_{full} = P_{rough} + w[N-1] * r^{-3} \tag{2.9}$$

$SELM(\widehat{v[j]})$ is the selection function, where $\widehat{v[j]}$ is a subset of $v[j]$ consisting of all its integer bits, and the first two fractional bits. For online multiplication of generalized radix, $SELM$ is done through rounding as described in Eq. 2.10.

$$SELM(\widehat{v[j]}) = \begin{cases} -a & \text{if } (\widehat{v[j]} + \frac{1}{2}) \leq a \\ \lfloor \widehat{v[j]} + \frac{1}{2} \rfloor & \text{if } -a < (\widehat{v[j]} + \frac{1}{2}) < a \\ a & \text{if } (\widehat{v[j]} + \frac{1}{2}) \geq a \end{cases} \tag{2.10}$$

Ercegovac developed Alg. 1 under the assumption of serial operation and presented an implementation in the radix-2 case only. This thus required me to adapt the algorithm for parallel execution. While this serial-to-parallel adaptation has been done before, and even optimized for FPGA execution, all previous attempts found are strictly radix-2 implementations (whereas my implementation works for a parameterized radix) [8] [9] [2]. So, following the same reasons as my online adder, I opted to develop my own, most basic, online multiplier implementation. Fig 2.3 depicts the high level architecture developed, built primarily off of online adders, bit shifts, and lookup tables.

Within my radix parameterization and data flow parallelization of the online multiplication algorithm, a few key inefficiencies arise not present in the starting point, serial, radix-2 version. Perhaps most impactful, the properties of radix-2 lend themselves to greatly simplified partial product and $SELM(\widehat{v[j]})$ operations. With a digit set of $\{-1, 0, 1\}$, the partial product in radix-2 becomes a simple negate, clear, or pass through operation. In higher radices, with larger digit sets, this efficiency is lost. The partial product must either be calculated with conventional multiplication (taking the form of DSPs within the FPGA), or a lookup table, which expands in size exponentially as radix increases. Similarly, efficiency can be found in $SELM(\widehat{v[j]})$ for radix-2. This function, as



(a) High level signal flow

(b) Internal components of a given stage.

Figure 2.3: Online Multiplier, figure inspired by Kan Shi [2]

given in Eq. 2.11, can be calculated directly from the value of $\widehat{v[j]}$, and removes the need for rounding, and the addition of $+\frac{1}{2}$. For the rounding selection function required in the case of higher radix there are two, less optimal options. Either a hard-coded lookup table, or a costly, multiplication based, conversion from the redundant representation to the true value of $\widehat{v[j]}$ is required. Again this selection lookup table grows exponentially with radix.

$$SELM(\widehat{v[j]}) = \begin{cases} -1 & \text{if } \widehat{v[j]} \leq \frac{-3}{4} \\ 0 & \text{if } \frac{-1}{2} \leq \widehat{v[j]} \leq \frac{1}{4} \\ 1 & \text{if } \widehat{v[j]} \geq \frac{1}{2} \end{cases} \tag{2.11}$$

These selection function issues disappear in the serial version of the algorithm, as conversion from redundant encoding to a 2's complement encoding of the value becomes cost free [16]. Converting the inputs from redundant form to conventional form also allows a more optimized adder architecture to be used, replacing the multiple online adders in the parallel case with a single $4:2$ adder in the serial case [15]. This adder has the benefit for not producing the previously discussed false overflow, allowing $w[j]$ and $v[j]$ to only require 2 integer bits, greatly reducing the complexity of any required lookup tables. Additionally, this reduced $w[j]$ allows the double precision output to be formed through a

cost free append operation, as opposed to the addition employed in my implementation.

Based on the analysis of these inefficiencies, and recorded online-adder results, I decided to implement and test only the radix-2 and radix-4 versions of the multiplier using hard coded lookup tables. The implementation presented with this thesis is parameterized to also accept higher radix configurations for completeness, but the partial product and selection function rely on the more cumbersome multiplication and conversion methods described. These methods greatly degrade overall speed performance. Additionally, based on provisional F_{STA} results, I added pipelining to the multiplier between each unrolled-iteration block to maximize speed. These buffering registers are placed between each 'stage' block in Fig. 2.3a.

Chapter 3

Experimental Setup

AS stated in Chapter 1, my overall goal of this research was to assess the maximal speed and degradation of high radix online arithmetic blocks compared to their more studied radix-2 counterparts on FPGAs. To achieve this goal I developed hardware implementations of the arithmetic operators (for simplicity given time constraints, only adders and multipliers are tested), a hardware test bench to test these arithmetic blocks, and software scripts to control the test bench and synthesize the data. All hardware HDL and simulation test benches were written in Verilog. Test control scripts were written in Python and Bash Script, data synthesis was done in MATLAB, and some software simulation was done in C++. The hardware designs were developed in the Intel Quartus development environment [17]. The targeted FPGA was the Cyclone V SoC Development Kit [18].

3.1 Testing Strategy

The on-board testing strategy roughly followed a UVM architecture [19]. The test controller generated randomized input sequences, injected them into the design under test (DUT), recorded the output, and compared this output to the expected results. I conducted these tests without targeting a specific application, so completely randomized input vectors were used. An application specific recreation of this experiment could constrain

the input vectors to achieve more precise results. The results of these comparisons then influenced what future tests the controller decided to run. I initially considered developing a full-fledged Built in Self Test (BIST), complete with on board linear feedback shift register (LFSR) random input generation, and multiple input signature register (MISR) output generation [20] [21], but ultimately decided instead to take a simpler, software-controlled approach. While the software-controlled approach I employed is no doubt less efficient than a BIST architecture (taking about 24hours for complete data collection), any gains in testing efficiency would have been greatly offset by the large increase in developmental overhead. With that said, the overall testing strategy for each arithmetic design was as follows:

1. Program FPGA with correct DUT
2. Verify successful programming by running arithmetic test at low clock speed (10MHz)
3. Adjust clock speed with binary search to find maximum working clock speed, (F_{real})
4. Collect data at sampled frequencies from 10MHz below F_{real} to maximum test bench frequency of 600MHz
5. Calculate and record various degradation and error metrics
6. Repeat steps 1-6 until statistically adequate data has been collected

After this, the designs were then sampled starting 10MHz below the measured F_{real}, in steps of 1MHz for the first 100MHz. Steps of 5MHz were then used for the next 100MHz, steps of 10MHz for the subsequent 100MHz, and finally steps 25MHz for any remaining frequencies until the test bench maximal frequency of 600MHz was reached. This was done to speed up data collection, under the assumption that the most relevant and interesting behavior would be close in frequency to F_{real}. Running an individual iteration of an arithmetic test followed a procedure of:

1. Generate random input vectors and expected output vectors in software

2. Load input vectors from the HPS (hard processing system) into FPGA BRAM (hard-baked memory blocks in the FPGA fabric)

3. Run vectors through DUT storing output back in DRAM

4. Offload output vectors to HPS

5. Compare expected output to generated output

Following the lead of past research in the area [2] [9], and suggested standards from [22], I calculated a variety of metrics at each sampled DUT frequency. Measuring F_{real} gave a baseline for the performance of each DUT. Minimum, maximum, and average error (Eq. 3.1), maximum and average absolute error (Eq. 3.2), and maximum and average mean relative error (MRE) (Eq. 3.3) at each sampled frequency revealed trends in the 'gracefulness' of each design's degradation. Note all averages were calculated using the arithmetic mean, and that measurements of MRE were excluded from the data when the expected value was 0.

$$Error = Recorded - Expected \tag{3.1}$$

$$Error = abs(Recorded - Expected) \tag{3.2}$$

$$Error = \frac{abs(Recorded - Expected)}{Expected} \tag{3.3}$$

In addition to these statistics, a count of the number of 'illegal digits' was recorded. The occurrence of illegal digits stem from the 2's complement encoding of the redundant digit sets. In the case of radix-2, the 2's complement binary vector 10_b, which has a value of -2 is a possible outcome when an error within the DUT occurs, but is outside the radix-2 digit set of $\{-1, 0, 1\}$. These illegal digits become an issue when attempting to convert a redundant number back into its conventional value form. The decision for how to handle these cases ultimately comes down to the designer, but options include: treating

all illegal digits as having a value of 0, treating all illegal digits as having the next closest legal value (in the case of radix-2 this would be -1), or simply accepting the illegal value as is (in the case of radix-2 the digit with binary vector of 10_b would yield an equivalent value to the redundant number $\bar{1}0_2$, or binary vector 1100_b). For the purposes of this project I opted to treat illegal digits as having a value of 0, my logic being 0 is only one bit-flip away from an illegal digit at any radix, and it will prevent the magnitude of test vectors from erroneously growing by large amounts (although it is not lost on me that a somewhat opposite and equally compelling argument can be made for selecting the closest legal value instead).

3.2 Arithmetic IP Development and Verification

3.2.1 Adder

Using the Digital Arithmetic textbook by Ercegovac as a guide [1], I first implemented a radix-2 online adder of parameterized digit width in Verilog with double-recoding. I then wrote a radix-4 version, and adapted it to be of parameterized radix (radix-R) and digit width. I finally combined the radix-2 and radix-R to produce my final implementation. This implementation was then validated with a simple Verilog test bench simulation that generated random input vectors and confirmed the output vectors with expected values.

3.2.2 Multiplier

The multiplier was a fair bit more difficult for me to implement, and a variety of issues, as outlined in Section 2.2 were encountered. Using the Digital Arithmetic textbook and descriptions from past papers [15] [23] [9], I first implemented a serial C++ model of the algorithm. After validating its execution with a randomly generating input C++ test bench, I ported the code into a parallel Verilog implementation. This implementation uses a hierarchy of modules, including the previously written online adder, a partial product module, a module that implements a single step/iteration of Alg. 1, and a top level module tying the whole design together. I then developed Verilog validation test bench

simulations, derived from the code for the online adder, to ensure the proper function of both the partial product module, and the overall online multiplier design.

3.3 Hardware Test Bench

Before I could run tests and collect execution and degradation data on the arithmetic units, I first had to develop a hardware test bench to actually perform said tests. This task proved to an extent more difficult than the design of the online arithmetic modules themselves, and absorbed the majority of my time completing this project. The complexity of this hardware test bench came from the fact that the experimental question being answered in this thesis required (already high-speed) designs to be tested at clock speeds beyond their breaking point. This meant that the key paths within the test bench itself had to be able to operate at frequencies higher than any test frequency imposed on the DUT. This was essential to ensure the recorded data captured the degradation of the arithmetic module under test, and not the degradation behavior of the test bench. The difficulty of this task was then further exacerbated by the fact that FPGAs are designed with clear industry applications in mind, and not an experimental research project requiring reliable operation across a wide (and dynamically controlled) frequency range. These differences required an additional level of creativity and lots of massaging the somewhat square tools into the rounder hole of the experimental setup.

3.3.1 Framework

As mentioned at the start of this chapter, I developed this project targeting the Cyclone V SoC Development Kit. The FPGA was configured to run a modified version of Ubuntu 16.04.7 on its hard processing system, and the board was networked and completely accessible and programmable remotely over SSH. Additionally a TP-Link smart plug was used to allow remote power cycling in the event of crippling errors. The FPGA fabric was configured by uploading a raw binary file ($.rbf$) of a design to the HPS, and programmed via HPS drivers. Data was transferred between the fabric and the HPS using the lightweight HPS-to-FPGA interface, and Avalon memory mapped interfaces. Details

on this memory mapping can be found in the HPS manual [24]. The configuration of the HPS, FPGA programming script, memory mapped python drivers, and even a basic Quartus Platform Designer (formally Qsys) structure were provided to me at the start of this research project, and were adapted from previous work [25] [26].

3.3.2 Architecture

To achieve my hardware testing goals I developed the test bench detailed in Fig 3.1. The key elements of this test bench are the test control unit (TCU), the dynamic phase-locked loop (PLL), and the double-pumped RAM structure surrounding the DUT. The whole test bench is interconnected with Avalon busses, which pass data and control information between the various hardware blocks and the HPS. The TCU is a module of my own design that is responsible for starting and stopping the flow of data through the DUT, and providing the correct read and write addresses to the RAMs. The dynamic PLL is a set of provided IP blocks from Intel and were simply dropped into the design and configured in 32-bit fractional mode. The DP RAM modules are wrappers around true dual port M10K block RAMs, where one port is connected to the HPS through the memory mapped Avalon bus, and the other port is connected to the DUT and controlled by the TCU.

In order to ensure the test bench was capable of operating at speeds beyond the breaking point of the DUT, I had to adopt a double-pumped architecture. Initial timing analysis estimates put the F_{STA} of a radix-2, 15-digit online adder at around 500MHz, but the data sheet reported F_{max} of the M10K block RAMs available on the Cyclone V is only 315MHz [27]. Double-pumping servers as an effective method to compensate for this disparity. While typically employed for the purposes of resource reduction through resource sharing [28] [29], I used double-pumping to unlock higher operating speeds. Double-pumping is where a hardware module is fed by two, 180 degree out of phase, clocks at half the clock speed of the module itself. This results in the module being fed data at full-speed, while only needing the supporting hardware to be capable of running at half-speed. Thus, with double-pumping, my test bench uses 6 different block RAMs, 4 feeding

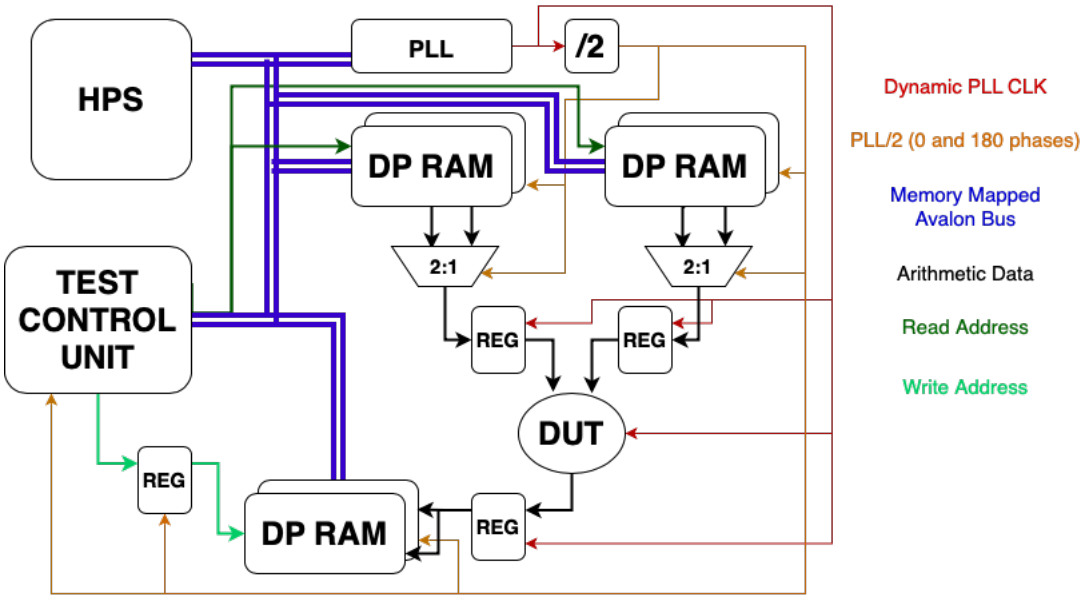


Figure 3.1: High level block diagram of hardware test bench. Note the orange $PLL/2$ line contains both the 0 and 180 degree phase clocks for visual clarity

the data, and 2 receiving the data. This enables a theoretical maximum test bench speed of 630MHz. To provide some conservative overhead to this number, I did not run the test bench faster than 600MHz.

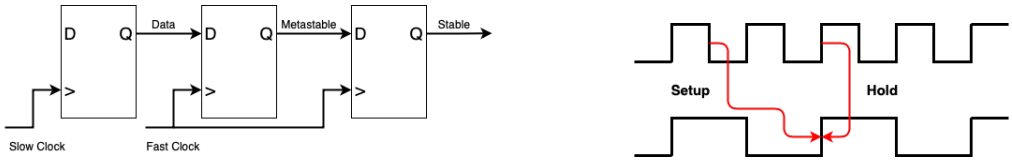
I used the Timing Analyzer tool within Quartus in order to verify the DUT was the critical path within the system. The results of these timing reports then influenced an iterative design cycle to further optimize the speed of the test bench. The most significant architecture change from these Timing Anazlyer results was the addition of pipelining registers to break up long control and data paths external to the DUT, as pictured in Fig. 3.1. Additional modifications were also made to the timing constraints file, synthesis settings, and fitter settings to help guide Quartus to generating an optimal output programming file.

3.3.3 Constraints and Debugging

My moderately non-standard use of a dynamic PLL, and the addition of double-pumping, greatly complicated the design process and initially introduced a small litany of issues in need of solving.

When implementing the control code for the dynamic PLL, I found inconsistencies and missing information in the documentation's descriptions of allowable register values [30] [31]. While the documentation indicates certain combinations of register values are valid, Quartus will throw errors when attempting to implement them as initial conditions. Typically this would not be that large of an issue, as most use cases of the dynamic PLL have the outputted frequencies configured between a number of predetermined settings. These settings can easily be manually checked in Quartus before hard coding. My application on the other hand required the precise configuration of an arbitrary number of unknown frequencies across a wide range, requiring me to find a viable, calculable solution for the configuration register values. Upon further research and searching on Intel's forums, I found threads confirming that Intel purposely withholds the full calculations and constraints Quartus uses when verifying PLL settings [32] [33]. Therefore, in order to confirm that the PLL was actually generating the clock speeds I expected, I built a quick secondary test bench. This test bench reports the value of a counter connected to the PLL after letting it count for a known number of reference clock cycles.

Another major roadblock I faced was getting the DUT to be reported as the critical path by the Timing Analzyer. Initially I was seeing no change in the setup-time critical path when adding pipelining registers to the long paths within the system. The setup-time seemed independent of the number of pipelining registers, and predominately from interconnects between FPGA resources. I cracked this puzzling behaviour from a comment in a StackOverflow post suggesting hold-time could be the culprit [34]. Analyzing the hold-time behavior of the system revealed that Quartus had gotten confused due to the introduction of multiple clock domains within the system due to the double-pumping. I had initially overlooked this fact since all the clocks used are perfect multiples of each other, with clear phase relationships. By adding double-flopping to control signals to prevent metastability [35] (Fig. 3.2a), following Example 23 from the Intel Quartus Prime Timing Analyzer Cookbook [36], and referencing a handy lecture on multicycle path constraints found online from Ben-Gurion University of the Negev [37] (Fig. 3.2b), I was able to alleviate the majority of the timing issues.



(a) Double-flopping to prevent meta-stability. Adapted from NANDLand [35]

(b) Fast-to-slow multicycle path. Adapted from BGU lecture [37]

Figure 3.2: Timing related modifications made to the test bench to account for multiple clock domains

A final, somewhat unfixed, bug in the system is that sometimes upon programming the FPGA, all tests will fail regardless of clock speed, and the DUT will produce outputs seemingly unrelated to the input test vectors. I theorize this issue is caused by the PLL failing to establish a proper phase lock, disrupting the phase relationships between the clocks, and thus crippling the phase sensitive double-pumping setup. All my initial attempts at fixing this issue failed however. In the name of saving time and keeping on track to collect actual experimental data, I implemented a brute force software fix. Immediately after programming, I run a test at a very low clock speed (10MHz). If this fails I simply continue to re-program the FPGA with the same programming file until it works (typically requiring only 1 to 2 reprogramming attempts, and never more than 10). While a hacky workaround, the bug was effectively eliminated, and shows no indication of having an impact on the recorded results.

Chapter 4

Results and Analysis

WITH the test bench built, and a testing procedure in place, I collected data across a range of widths and radices for the adder and multiplier. The tests run on the adder are detailed in Table 4.1, and the multiplier tests are detailed in Table 4.2. In both cases, standard, power of 2 widths were chosen for the control module. These were then compared to online modules with digit widths capable of capturing the equivalent amount of data. In this way, the performance of like-to-like drop in replacements can be quantified. Note at some radices the data range did not map perfectly to equivalent digit widths. In this case, the lowest digit width capable of capturing at least as much data as the control was used. I refer to these equivalently sized adders by the width of their corresponding control.

	Control	R-2	R-4	R-8	R-16
Digit Width	16	15	8	5	4
	32	31	16	11	–
	64	63	32	21	–
	128	127	64	43	–

Table 4.1: Width and radix of tested adders

As digit width increases, so too does the block RAM utilization of a single vector. However, in order to keep the test bench as constant as possible across all tested interactions, I kept the amount of allocated block RAM unchanged, regardless of digit width. While maintaining consistency, this presented the problem that it was not possible to store

	Control	R-2	R-4
Digit Width	8	7	4
	32	31	15
	64	63	32

Table 4.2: Width and radix of tested multipliers

the same number of test vectors on the FPGA for each tested width. For example, 2048 vectors can be stored for a 15-digit radix-2 adder, but only 512 vectors can be stored for the 63-digit radix-2 case. To compensate for this, I increased the number of trials run on a given implementation as the digit width increased, so the overall number of vectors tested remained the same. Using the previous example, while the 15-digit radix-2 test was run 10 times, the 63-digit, radix-2 test was run 40 times.

While best efforts were made within the physical design and overall test strategy to minimize the influence of the test bench on the DUT, the stochastic nature of the fitter was (and is) unavoidable. Specifically, the target clock frequency given to the tools has a large effect on placement and routing, and the Timing Analyzer reported critical path. The goal was to find the correct target frequency that produced a design where the DUT was as optimal as possible, but still slower than the test bench. Ideally to accomplish this, each tested design should be compiled at the tool's estimated maximum frequency (not it's true, overclocked maximum frequency), and it should be confirmed that the Timing Analzyer considers the DUT to be the critical path with adequate margin. With 26 individual designs being tested, and compile times on the order of 30 minutes, the iterative, knob turning adjustments on the fitter required to perfectly tune each design would have taken unreasonably long. Ideally, the use of separate block compilation, such as the methods recommended by Xiao et al. [38], could have been used. This would allow for the test bench and DUTs to be tuned separately and combined. Instead, what I assumed to be one of the fastest DUTs (the 15-digit, raidx-2 adder) was tuned to these standards (a target clock speed of 333MHz was found to produce the best results), and then this tuning was used for all other designs of the same class. A similar process was carried out on the radix-2 7-digit multiplier, for a target frequency of 250MHz. While the ability to compare truly,

maximally optimal implementations using this method is somewhat degraded, a level of fairness in the tools implementation of the designs is still preserved. A deeper exploration of the effects of this design decision is presented in Section 4.3.

4.1 Addition

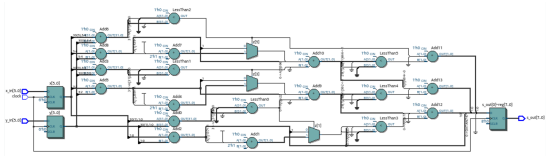
4.1.1 Speed

The measured F_{real} of each adder is reported in Table 4.3. From these results a few key observations can be made. In all widths, at least one online adder implementation reached higher speeds before failing than the control. Additionally, at almost all widths, the radix-2 adder outperformed those of higher radices in this regard. This is especially true at low widths, but this disparity decreases as width increases.

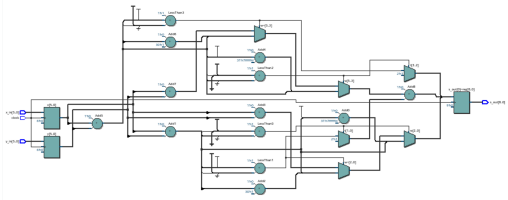
		Control	R-2	R-4	R-8	R-16
Equivalent Control Width	16	556	575	512	365	280
	32	483	533	437	367	–
	64	328	499	529	328	–
	128	277	381	350	323	–

Table 4.3: Measured F_{real} of adders. Frequencies given in MHz.

The general trends found in this data align with expectations. At low widths, the complexity of the addition is not sufficiently high enough to benefit from the speed benefits of the FPGA's hard-baked adders. This is why the radix-2 implementation, which relies on many, very narrow (2-bit) adders sees the best performance. As the width grows however, so does the number of basic fabric blocks (adaptive logic modules, ALMs) required for radix-2. This increase in ALMs then requires increasingly complex, and decreasing optimal, routing to connect them all together. The combination of these two effects lead to the quickly degrading performance. In contrast, the high radices, and particularly radix-4, depend on fewer and wider hard baked-adders for the individual digit additions, allowing for more efficient packing, and less slack required for interconnects. This ultimately causes the speed performance to degrade at a more gradual rate. Perhaps more impactful



(a) Tool generated RTL view of a radix-2 3-digit online adder



(b) Tool generated RTL view of a radix-4 2-digit online adder

Figure 4.1: Comparison between the RTL usage of radix-2 and radix-4 online adders of 4-bit equivalent width

than the width of the adders, recall that radix-2 addition requires double-recoding, adding another layer of logic, and another block of carry to the operations. Examining the tool generated RTL views (Fig. 4.1) illustrates this disparity in resource usage between radix-2 and radix-4 implementations of equivalent widths. Here an equivalent width of 4-bits is used to keep the figures to a reasonable size while still illustrating the point. Note at wider equivalent widths the disparity in resources is even larger.

These observations are further supported by a comparison between the control and radix-4 performance. In the lower width cases, both adders are implemented with hard-baked adder blocks, but the radix-4 has an additional logic layer on top to allow for its redundant behavior. This extra logic leads to the relative reduction in the radix-4 performance when compared to the control at these widths. As the width of the addition grows however, so does the carry chain in the control adder. In contrast, the carry chains of the radix-4's hard-baked adders remain constant, as they are employed only to add individual digits together, and are independent of width — it is the lookup table (LUT) based logic layer that handles carries across the digits in this architecture. As such, at higher widths the radix-4 adder pulls ahead of the control. This characteristic of width insensitivity in the high radix adders is further illustrated by the performance of the radix-8 implementations. Here, even more of the addition operation is sectioned off to constant length, hard-baked adders, with less LUT overhead. Again, while inefficient and unnecessary at low width, as the complexity of the required ALMs and their connectivity grows, the consistent use of many, parallel, short adders becomes increasingly advantageous.

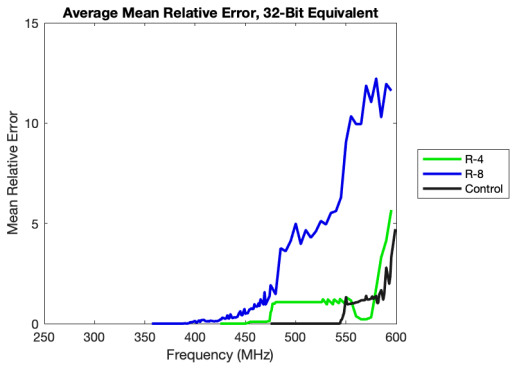
Another way to express this same notion of differing rates of growing complexity it to examine the space efficiency of the data storage. Returning to the discussion of illegal digits from Chapter 3, it can be seen that higher radices use their allocated bits more efficiently. For any given power of 2 radix with digits represented redundantly in 2's complement, only one bit combination is considered illegal. In radix-2, where each digit is represented in 2-bits, this accounts for 25% of the possible bit vectors. In radix-4, they make up 12.5%, and in radix-8 only 6.25%. At the low complexity of the lower widths this is not particularly impactful, but as width increases so too do the inefficiencies of the radix-2 representation. Note that while 0% of the control adder bit representation can be considered illegal, its sharp drop in F_{real} performance is due to the growing carry chains of the hard-baked adders. In radix-4 and radix-8 these carry chains never exceed 3-bits and 4-bits respectively, while in the control the carry chain is equal to the width of the operation.

From this F_{real} data, the recorded value for the 64-bit equivalent radix-4 adder (the 32-digit radix-4 adder) stands as a slight outlier. This is due to the fact that all these designs are being run at overclocked speeds, and their exact performance is sensitive to the exact fitter settings. The stochastic nature of this will be further explored in Section 4.3.

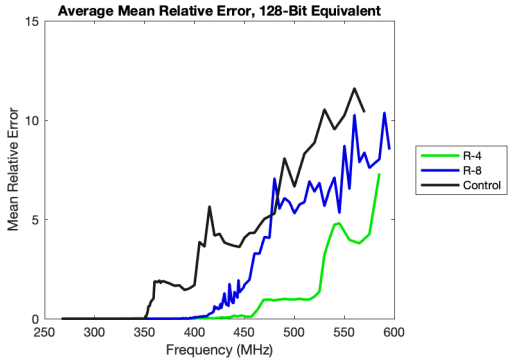
4.1.2 Degradation

The data collected on the degradation of the adders paints a less clear picture than the F_{real} measurements, but meaningful conclusions can still be made. Note, that in the case of the 16-bit equivalent adders, the measured F_{real} comes as close as 25MHz of the maximum test bench speed of 600MHz. This gives very little room for a statistically relevant sweep of the degradation. As such, this data is mostly excluded from the analysis.

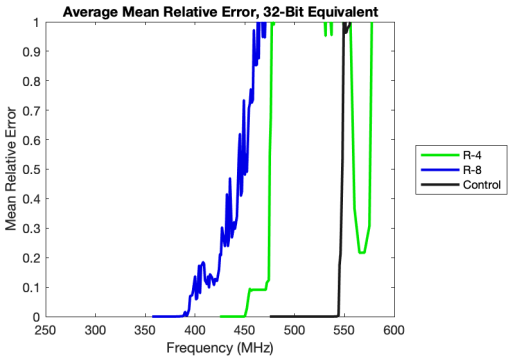
Examining the mean relative errors of the 32-bit equivalent, and 128-bit equivalent adders in Fig. 4.2, the expected degradation behavior of the online adders can be seen. In the adder, all output digits are computed in parallel. This gives each digit place a roughly equal chance of being the critical path and failing first. As such, this results typically



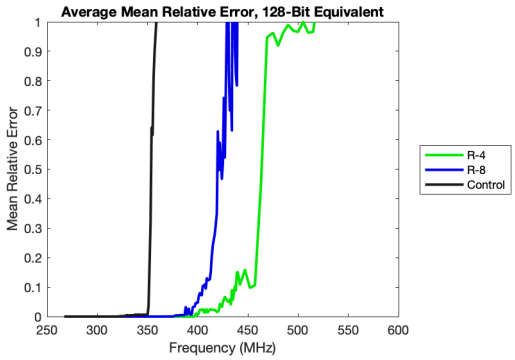
(a) Average MRE of the 32-bit equivalent width adders



(b) Average MRE of the 128-bit equivalent width adders



(c) Average MRE below 1 of the 32-bit equivalent width adders

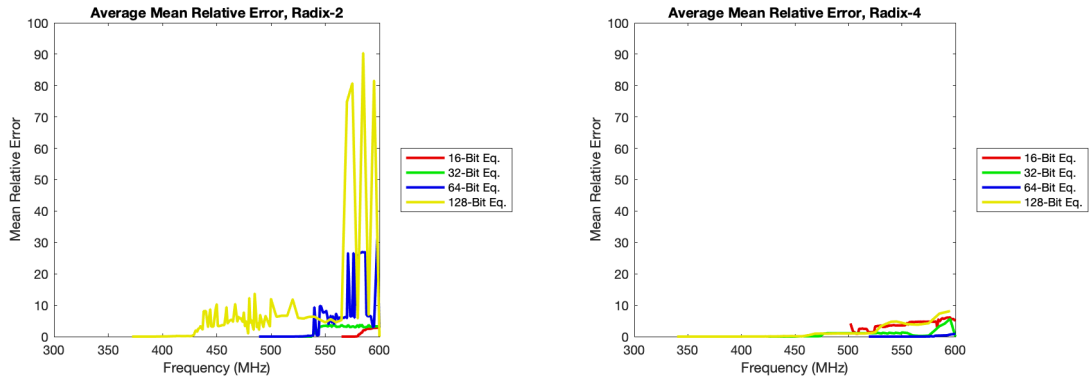


(d) Average MRE below 1 of the 128-bit equivalent width adders

Figure 4.2: Comparison between the average MRE behavior of adders at different equivalent widths

in short cliffs in degradation as a higher significant digit fails, followed by stability as digits of lower significance fail. Then once a path of an even higher significant digit fails, the degradation jumps up again. This is especially true of the radix-4 adder, and in the 128-bit equivalent case (Fig. 4.2b). After exceeding the F_{real}, the mean relative error of the online adders remains relatively low before hitting some critical failure frequency and quickly climbing. In contrast the MRE of the control jumps and continues to climb as soon as F_{real} is exceeded.

This behavior of the control versus the online adders is perhaps better illustrated in Fig. 4.2c and Fig. 4.2d. Here the graphs have been focused to only the behavior below an MRE of 1, displaying the values of MRE closer to what would be acceptable in an actual application. While in all cases the adders experience a 'cliff-edge', the online adders have

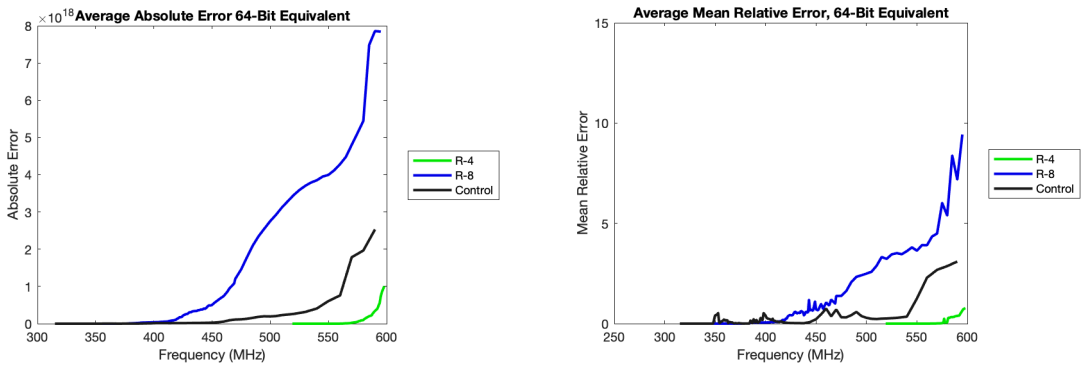


(a) Average MRE of various radix-2 adders (b) Average MRE of various radix-4 adders

Figure 4.3: Comparison between the average MRE behavior of adders at radix-2 and radix-4, set to the same scale.

a short span of low MRE as digits of lower significance fail, before spiking to their first failed digit of higher significance. In the case of the 128-bit equivalent radix-4 adder in Fig. 4.2d, the MRE remains low for a span of about 50MHz beyond F_{real} before spiking. The control on the other hand spikes immediately after F_{real} is exceeded. The fact that this cliff-edge behavior in the control does not continue all the way to the maximum MRE can be explained by checking the path slack in the Timing Analyzer. While the MSD path has the shortest slack as expected, the slack of some other paths throughout the adder are only longer by within a tenth of a nano-second. Since degradation has a level of randomness to it, the failing of these intermediate adder paths interspersed with the failing of the high-significance adder paths explains the more gradual growth in MRE. Overall however, the degradation of the online adders is more gradual. This advantageous property of the online adders is the graceful degradation of which the title of this thesis gets its name.

While the graphs in Fig. 4.2 display typical and expected degradation curves, not all the degradation recorded in the experiment was found to be as graceful. This is particularly true of the online radix-2 behavior, as shown across all tested widths in Fig. 4.3, and compared to the radix-4 behavior in the same scale. It can be seen that at frequencies just beyond F_{real}, the radix-2 adders (Fig. 4.3a) behave within expectations, but as the frequency becomes larger, and the adders wider, the trend of the degradation



(a) Average abs. error

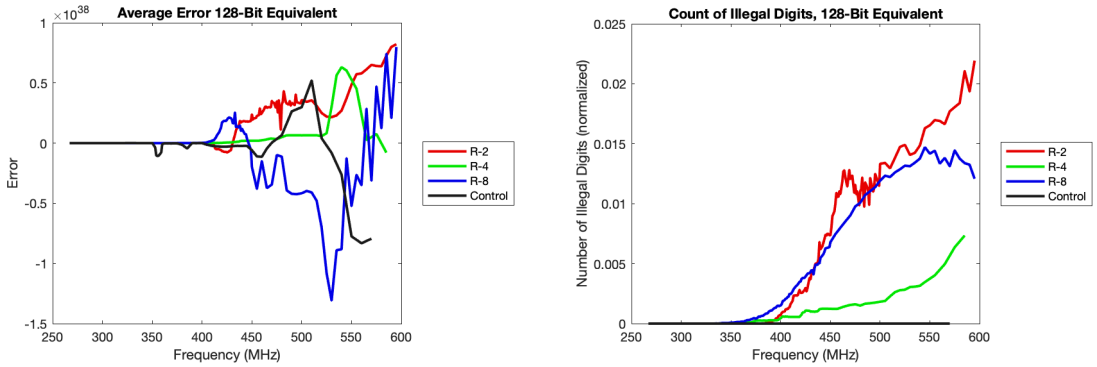
(b) Average MRE

Figure 4.4: The average absolute error and average MRE of the measured 64-bit equivalent adders

becomes increasingly unstable. This is in stark contrast to the radix-4 behavior in Fig. 4.3b. Here (easier to observe in Fig. 4.2) the degradation follows a discernible upward trend as frequency increases, and this trend maintains more independence to the width of the adder. These observations further support the analysis made in Section 4.1.1 about the growing complexity of the radix-2 adder as width increases, and how that complexity becomes increasingly difficult to map into FPGA fabric.

Another interesting result worth discussion is the 64-bit control's MRE response. This is seen in Fig. 4.4, and compared to the absolute error for reference. In this case, the error growth of the control followed a very advantageous curve, especially compared to the radix-8 adder. This somewhat puzzling disparity in results is most likely explained by a particularly optimal implementation by the fitter, and will be further explored in 4.3. In reference to Fig. 4.4a, remember the widths of each adder tested are not identical, but instead the closest equivalent width in terms of data capabilities. As such, while radix-8 exhibits significantly worse absolute error, it is also operating over a larger possible range of input and output values. Mean relative error attempts to correct for this disparity.

Using the 128-bit equivalent adders as a frame of reference due to the wide frequency range over which data is collected, some other interesting metrics are worth noting (Fig. 4.5). Examining Fig. 4.5a, a slight positive trend in the value of overall error can be seen in the radix-2 and radix-4 adders. One possible explanation for this positive trend



(a) Average error

(b) Normalized count of illegal digits

Figure 4.5: Additional performance metrics of the 128-bit equivalent adders.

could be due to how the value of illegal digits is handled within the test bench (assigning a value of 0). By assigning the illegal digits a value of -1 instead, a slight negative trend would be expected. As seen in the case of radix-8, these trends are not particularly strong, and are of course influenced by randomness in exactly how the adder fails each time it is overclocked. Additionally, again remember that the different radices are not exactly comparable within the same equivalent width, so the magnitude of the error must be, to an extent, disregarded.

A count of the occurrences of illegal digits (normalized to the total number of digits tested) is shown in Fig. 4.5b. From this radix-2 experiences the highest number of illegal digits as expected and discussed in Section 4.1.1. Unexpectedly however, radix-8 experiences a trend more similar to radix-2 than to radix-4, despite having a much lower percentage of its possible bit combinations be illegal. This generally worse behavior in the case of radix-8, especially when compared to radix-4, can also be observed in all the other error metrics I have reported on in this section. I suspect the reason for this is due to larger digit-wise hard-baked adders employed on the FPGA fabric. These adders have longer carry chains, and thus fail more readily than narrower ones used in radix-4. A secondary reason could also come down to the specific and stochastic placement that was used in testing. Overall however, the data indicates that radix-8 adopts the negative behavior of traditional adders, along with the added downsides of the LUT layer and

radix-2 behavior. Radix-4 on the other hand seems to strike a more maximal balance between the two.

4.1.3 Area

While the primary focus of this thesis is on the speed of the online arithmetic modules, it is useful to also quickly examine resource utilization when determining the benefits of one radix over another. Overall, the tool reported area (Table 4.4), and calculated area-delay product (Table 4.5), further support the analysis I've made so far.

		Control	R-2	R-4	R-8	R-16
Equivalent Control Width	16	17	58	54	88	93
	32	33	122	110	202	–
	64	65	250	222	392	–
	128	129	506	446	810	–

Table 4.4: Tool reported number of LUTs required by each tested adder.

		Control	R-2	R-4	R-8	R-16
Equivalent Control Width	16	0.031	0.101	0.105	0.241	0.332
	32	0.068	0.229	0.252	0.551	–
	64	0.198	0.501	0.420	1.195	–
	128	0.466	1.328	1.274	2.508	–

Table 4.5: Area-Delay product of tested adders. Values given in LUT-μs.

At low widths, while radix-2 still requires more resources than radix-4, the difference is overshadowed by better overall performance, leading to a lower (more advantageous) area-delay product. As width increases beyond $32 - bits$, the resource utilization of radix-2 grows at a faster rate than radix-4, resulting in radix-4 having a better area-delay product. The full extent of radix-8's resource utilization is surprising, and clearly explains the much worse performance in speed and degradation than the radix-2 and radix-4. Based on these results, it seems the added width to the hard-baked adders in radix-8 exceeds a critical width within the optimized fabric of the FPGA, requiring a large number of additional LUTs to compensate. The magnitude of this increase in resources is made even more significant when considering the fact that because a radix-8 encoding is more

space efficient, the digit-width of radix-8 adders is lower than the digit-widths of the other adders in an equivalent width class.

Also from this data comes the expected fact that the control adders use the fewest resources, and see the best area-delay products by a wide margin. This fact enforces the notion that online arithmetic is best applicable in applications where speed is the primary concern, and adequate resources are available.

4.2 Multiplication

4.2.1 Speed

		Control	R-2	R-4
Equivalent	8	446	269	151
Control	32	265	215	128
Width	64	148	201	129

Table 4.6: Measured F_{real} of multipliers. Frequencies given in MHz.

		Control	R-2	R-4
Equivalent	8	0, 0.000	9, 0.033	6, 0.040
Control	32	2, 0.008	33, 0.153	17, 0.133
Width	64	6, 0.041	65, 0.323	34, 0.264

Table 4.7: Number of pipeline stages and latency of tested multipliers. Given in stages, μs

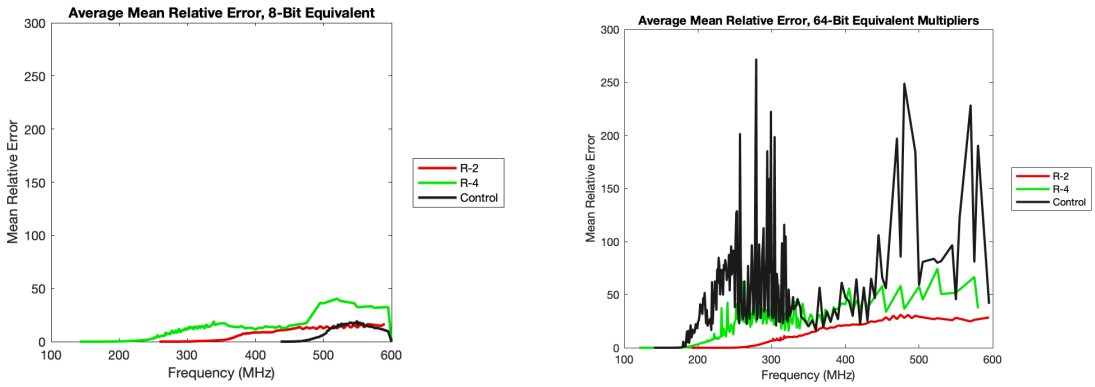
In order to properly examine the measured performance of the multipliers (Table 4.6), the pipelining applied to each design must first be considered. As stated in Section 2.2, pipelining was added between each iterative stage of the online multiplier. This results in each multiplier having a different number of pipelining stages, regardless of equivalent width (Table 4.7). While this could be seen as hindering the ability to properly compare multipliers across the same equivalent width, I argue that the method in which I pipelined is optimal, and the addition of more stages would not result in improved F_{real}. While more than optimal pipelining would be acceptable, as extra stages will not affect F_{real} or the degradation behavior, they are ultimately unnecessary. Note that the control was pipelined

by iteratively adding more stages until the maximum tool estimated performance, F_{STA}, was reached.

Examining the F_{real} of each tested multiplier, as reported in Table 4.6, produces less compelling results for the benefits of higher radix on FPGA than the adder results. Here, radix-2 is found to have a notably higher F_{real} than radix-4 in all cases. Additionally, the speed of radix-2 only exceeds that of the control once a sufficiently high width (64-bit equivalent) is reached. In general these two results align with the analysis done on the multiplier architecture itself in Section 2.2.

As noted, the online multiplier algorithm developed comes from a design originally meant for serial execution. This results in long logic chains that must be mapped and connected throughout the FPGA fabric. The control on the other hand utilizes the built-in DSP blocks, purpose meant for these simple multiplications. At narrow widths, these DSP blocks perform extremely optimally. But, as the width increases, and more DSP blocks need to be chained together to complete the multiplication, the benefit of these optimizations is lost. Thus, at these wider widths, the online implementation, with MSD-first carry chain, achieves better relative performance.

From the architecture, the recorded speed capabilities of the radix-2 over the radix-4 also make sense. The implementation details of these multipliers is much more similar than in the case of the adders. While it is the case that each multiplier utilizes multiple adders of their respective radix, the largest difference comes in the critical path forming partial product and digit selection operations. In both radices, and for both operations, hard-coded look up tables are used. However, due to the larger digit set, the radix-4 look up tables are significantly larger than their radix-2 counterparts. This requires more resources, and leads to more fitting inefficiencies, leading to an overall reduction in performance. Interestingly, as with the adder, as the width increases, the performance of the radix-2 multiplier decreases at a faster rate than the radix-4. I suspect that these trends are being caused by the differences in the online adders used, for the same reasons as discussed in Section 4.1.1. If these trends continue, it could be the case that in very wide multiplications, the radix-4 performs better than the radix-2.



(a) Average MRE of the 8-bit equivalent width multipliers

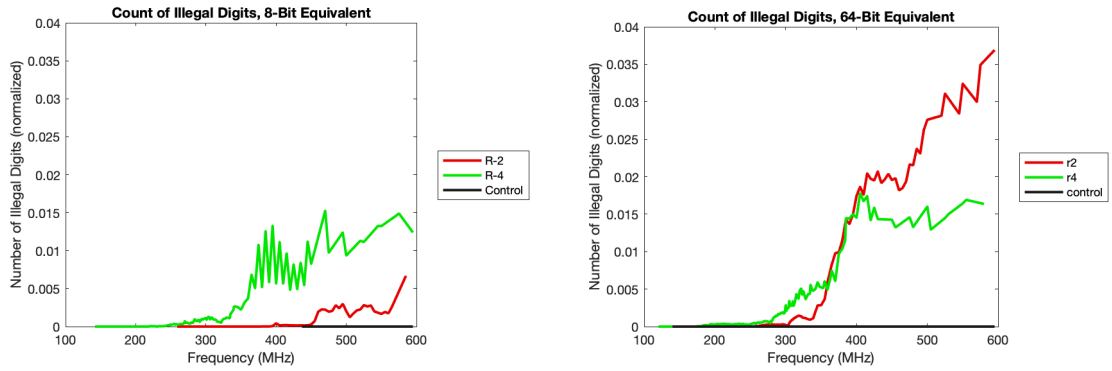
(b) Average MRE of the 64-bit equivalent width multipliers

Figure 4.6: Comparison between the average MRE behavior of multipliers at different equivalent widths

4.2.2 Degradation

In terms of degradation, the benefit of online multiplication over the control is even clearer. Additionally much of the unexpected behavior observed in the adder does not occur. Fig. 4.6 shows the MRE of both the 8-bit equivalent and 64-bit equivalent multipliers. In both cases the radix-2 multiplier experiences a more graceful degradation than the radix-4. This is again, most likely due to the simplified look up tables of the radix-2 multiplier for partial product generation and digit selection. In the 8-bit equivalent case (Fig. 4.6a), the control adder also experiences a somewhat graceful degradation. This can be explained by the fact that at 8-bits, the multiplication is narrow enough to fit entirely in one DSP block. While carry chains still span the multiplication's width, they are physically short and highly optimized within the block. Contrasting this the 64-bit case (Fig. 4.6b) , where multiple DSP blocks are required, the MRE of the control grows quickly with instability similar to the high-width radix-2 online adders, with the majority of its recorded levels higher than either of the online implementations. This 64-bit equivalent case clearly highlights the advantage of online methods when overclocking.

The most unexpected behavior of the multipliers comes from the count of illegal digits at the 8-bit equivalent as shown Fig. 4.7. In this low-width case, radix-4 experienced a much higher number of illegal digits than radix-2. Similar in behavior to the radix-8



(a) Normalized count of illegal digits of the 8-bit equivalent width multipliers

(b) Normalized count of illegal digits of the 64-bit equivalent width multipliers

Figure 4.7: Comparison between the counts of illegal digits of multipliers at different equivalent widths

adder, this is again in spite of the fact that the radix-4 illegal digits make up a smaller percentage of possible digit encodings. An explanation for this could simply be that at these narrow widths, the radix-4 fitting produced by the tools is particularly inefficient. Additionally, the large disparity between the radix-2 and radix-4 F_{real} means a larger relative span of the radix-4 degradation was recorded before the maximum test bench frequency was reached. Looking at the high end of the recorded frequency spectrum, the growth in illegal digit count of radix-2 outpaces that of radix-4. Similar behavior can be observed in the 64-bit equivalent response (Fig 4.7b), with the radix-4 count eventually being surpassed by the radix-2 count at around the 350MHz mark. Overall the illegal digit count data from these wider-width multiplies aligns with expectations.

4.2.3 Area

		Control	R-2	R-4
Equivalent	8	0, 1	807	1153
Control	32	46, 3	11026	9501
Width	64	278, 9	38690	36945

Table 4.8: Tool reported number of LUTs required by each tested multiplier. The control multipliers are given in LUTs, DSPs. DSP usage is 0 for all online multipliers.

As with the adders, it is worth examining the area usage (Table 4.8), and area-

		Control	R-2	R-4
Equivalent	8	0.0, 0.002	3.0	7.6
Control	32	0.2, 0.011	51.3	74.2
Width	64	1.9, 0.061	142.7	286.4

Table 4.9: Area-Delay product of tested multipliers. Values given in LUT-μs. The control multipliers are given in LUTs-μs, DSPs-μs. DSP usage is 0 for all online multipliers.

delay product (Table 4.9) of the multipliers. From the general LUT usage in Table 4.8, an interesting trend between radix-2 and radix-4 occurs. At the 8-bit equivalent width, the radix-4 implementation requires more resources than radix-2, despite its digit-width being lower (4-digits in radix-4 versus 7-digits in radix-2). This is likely due to much larger lookup tables needed for the radix-4 partial product generation and digit selection. As the width increases however, the growing disparity in digit-width, and the growing resource utilization of the radix-2 adders within the multiplier, result in radix-2 requiring more overall resources. Despite this, the area-delay factor of radix-2 remains much lower than that of radix-4 in all cases. Again, this observed benefit of the radix-2 can be attributed to its simpler lookup tables. Note in all cases the control multipliers saw much better resource utilization and are-delay products. This again emphasizes careful consideration of design priorities when implementing online arithmetic.

4.3 Meta-analysis

FPGA tool sets are designed to produce results with guaranteed operation below a given target clock frequency. This is done through a complex, pseudorandom fitting (also called place and route) algorithm that is highly sensitive to initial conditions. Given there are no timing violations, while the actual programming files produced by the fitter may differ for the same design, the designs will perform identically at speeds below the target. However, when overclocking these designs, or running designs that fail STA, the randomness of the fitting algorithm *can* impact the actual performance. Given the same design, but different starting conditions, the fitter will produce different results, which will then break down in different ways, and at different frequencies. In the context of my experiment, this means

the designs I test could be a particularly optimal or non-optimal fitting, skewing the overall analysis and answer to my hypothesis. Thus, to get a better idea of the impact of this randomness, I generate multiple copies of a subset of the tested designs, with each copy using a different initial fitter seed value. I then measure the change in F_{real} as a result of this different seeding, as reported in Table 4.10. Note that none of these designs passed STA.

	Adder					Multiplier
	64-Bit Eq.					32-Bit Eq.
	Control	R-2	R-2	R-4	R-8	R-2
Target Frequency (MHz)	333	333	1000	333	333	250
Seed=1 Tool-reported F_{STA} (MHz)	227	318	348	327	246	151
Minimum (MHz)	326	400	486	390	334	209
Maximum (MHz)	410	510	511	532	367	221
Average (MHz)	379	473	495	491	350	217
Standard Deviation (MHz)	32	39	13	42	9	2

Table 4.10: Statistical F_{real} of various arithmetic IP generated with 10 different seeds. All frequencies reported in MHz

From Table 4.10, it becomes clear that the stochastic nature of the tools can lead to widely varying overclocked performance. This appears to be especially true of the simpler designs, with the control adder, radix-2 adder and radix-4 adder all seeing large frequency ranges and standard deviations. The more logically complex radix-8 adder, and radix-2 multiplier on the other hand exhibit more reasonable levels of insensitivity to the fitter. It makes sense that the simpler designs see a wider range of results, as the tools have more options in how to place and route these designs while trying to meet the timing constraints before eventually settling on a solution.

As mentioned at the beginning of this chapter, I decided to compile all the adder designs at the same target frequency of 333MHz, and multipliers at 250MHz. I did this to generally promote fairness across the tests, with slight advantage being given to the low-width designs these target frequencies were tailored for. Performing seed testing on an adder with the much higher target frequency of 1000MHz indicates the logic behind this decision to be somewhat flawed. By giving the fitter a completely impossible target frequency, the tools will have to exert a maximal amount of effort to attempt to satisfy

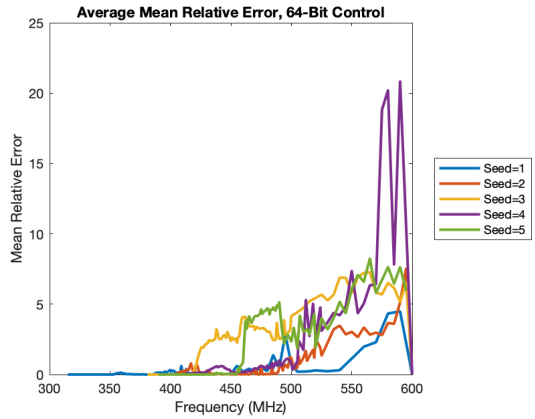


Figure 4.8: MRE of 64-bit control adders generated with different seeds

timing constraints. As seen from the seed testing, this maximal effort leads to a tighter range of possible outcomes. Additionally, the data indicates that these possible outcomes trend towards the more optimal implementation.

In Section 4.1.2, I briefly comment on the notably good, and unexpected, degradation of the 64-bit control adder from Fig. 4.4b. Based on this result, I collected MRE data for the adder generated with different starting seeds, as plotted in Fig. 4.8. From these results it appears the specific implementation I test in my main data collection, seed=1, is indeed particularly optimal with regards to its degradation. While there is of course variance between the other seeds, these MRE trends more closely align with the control MRE behavior found at other widths. Following similar logic, and supported by the results found in Table 4.10, it seems that the high performance of the 64-bit equivalent radix-4 adder recorded in the main data collection also happens to be a particularly optimal fitting of the design.

Based on these findings, an added level of nuance and care must be taken when forming conclusions about the data gathered in this thesis. Due to the wide variance in performance based on random chance from the fitting algorithm, there is no guarantee that the data collected is not from an outlier implementation. Therefore, notions that one width/radix/design is absolutely better than another must be critically examined. Instead, the safer approach is to analyze and draw conclusions from the general trends of the data. As the tools cannot guarantee performance tolerances in these overclocked

designs, this task falls to the designer. Based on the results of these seed tests, it is clear that the designer must do the due diligence to properly explore the implementation space, understand the risks, and ensure their application can tolerate the random error that can occur.

Chapter 5

Conclusion

THIS thesis set out to explore the question of if high-radix online arithmetic operators could outperform the more traditional radix-2 implementations on FPGAs. The motivation for this exploration being the idea that the hard-baked blocks within the FPGA's micro-architecture would be better utilized in the higher radix cases. Through a combination of on-board testing, as well as theoretical analysis of the online algorithm and architectures, I have found there is no clear answer, but the choice of what radix to use is instead application dependent.

Firstly, data widths must be sufficiently wide to justify the use of any online operators as opposed to more traditional methods. This is especially true of the multiplier, but extends to the adder as well. Working with online operators requires a certain level of developmental overhead and complication, as well as the actual performance costs of converting the data between redundant and conventional representations. As such, I cannot recommend the use of online operators for data widths below about 32-bit equivalent in adders, and below 64-bit equivalent for the multipliers. Additionally the arithmetic chain of operations must be sufficiently long enough to justify the extra processing needed in conversion.

In some cases, a strong argument can be made for the use of radix-4 over radix-2. For the online adders, at widths greater than 64-bit equivalent, the radix-2 and radix-4 implementations achieve comparable F_{real}. In general though, the radix-4 adders exhibited

a more graceful and predictable degradation than their radix-2 counterparts. Additionally, due to the nature of the number representation, radix-4 is more space efficient with its handling of the data, allowing for the added benefit of area improvements. For the online multipliers however, the advantage of radix-4 is lost. Here radix-2 is the only clear choice. These facts therefore support the assertion that radix-4 only becomes worthwhile in the case of addition heavy arithmetic pipelines. In all other cases, the use of the more traditional radix-2 makes more sense. Furthermore, the performance benefits seen by the radix-4 implementations due to the FPGA architecture are lost at even higher radices - in no case is the use of a radix beyond 4 advisable.

The work done in this thesis is not without flaws, and future improvement and extensions to the experiment stand to be made. As explored in 4.3, overclocked designs are susceptible to randomness in the fitting process. This randomness could have led to unintentionally skewed results. To obtain more robust results, this randomness must be better accounted for. This can be done through better constraining of the design before fitting (such as setting a higher target frequency), full data collection across a large set of seeds, and inter-die testing by running the compiled designs across multiple physical instances of the same FPGA. Moreover, the experiment was exclusively run on a Cyclone V architecture, and different architectures or chips from different manufacturers could produce different results.

Another limiting factor of the experiment is the maximum frequency of the test bench itself. In my case, this frequency is only slightly higher than the fastest designs tested. This prevents meaningful degradation data to be collected for these designs, and even opens the possibility that due to fitter randomness the test bench is in fact failing before the DUT (I do not believe this to be the case for the data I collected). Expanding the F_{real} of the test bench would allow for more data collection, and a larger safety margin in measurements. This could be done in a variety of ways, including expanding the double-pumping BRAM into quad-pumping, and/or replacing the RAM address counters in the TCU with faster, LFSR blocks.

Throughout the experiment, only the most basic implementations of the arithmetic

operators are tested. This is true of both the online architectures, and the controls. While FPGA optimized versions have been developed for traditional and radix-2 implementations, higher radix has yet to receive the same treatment. The development and testing of these optimizations, especially ones that actively target the FPGA fabric for higher radix, have the potential to yield a more positive result to the research question. This is particularly pertinent to the multiplication algorithm. Here even simple improvements like the use of 3-input online adders instead of two 2-input online adders stand to be made.

Lastly, this project focused solely on addition and multiplication. Online implementations exist for division as well as more complex mathematical operations such as the sum of squares. As such, high-radix implementations of these operators deserve to be tested too. Beyond just that, it is very rarely the case that a single arithmetic operation is required. Instead, it is much more common to see full arithmetic pipelines utilizing a series of arithmetic blocks. The testing and comparisons of these full pipelines is the ultimate test to determine the usefulness of higher-radix, and would be a worthy extension of the work I have done.

Bibliography

[1] M. D. Ercegovac and T. Lang, *Digital arithmetic.* Morgan Kaufmann, 2003, ch. 2, pp. 97–112.

[2] K. Shi, D. Boland, E. Stott, S. Bayliss, and G. A. Constantinides, "Datapath synthesis for overclocking: Online arithmetic for latency-accuracy trade-offs," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2014, pp. 1–6.

[3] S. I. Association, "International Technology Road Map for Semiconductors (2007)," 2007.

[4] K. Shi, D. Boland, and G. A. Constantinides, "Accuracy-performance tradeoffs on an fpga through overclocking," in *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, 2013, pp. 29–36.

[5] K. S. Trivedi and M. D. Ercegovac, "On-line algorithms for division and multiplication," in *1975 IEEE 3rd Symposium on Computer Arithmetic (ARITH)*, 1975, pp. 161–167.

[6] M. Dimmler, A. Tisserand, U. Holmbeg, and R. Longchamp, "On-line arithmetic for real-time control of microsystems," *IEEE/ASME Transactions on Mechatronics*, vol. 4, no. 2, pp. 213–217, 1999.

[7] R. Galli and A. F. Tenca, "Design and evaluation of online arithmetic for signal processing applications on FPGAs," in *Advanced Signal Processing Algorithms, Architectures, and Implementations XI*, F. T. Luk, Ed., vol. 4474, International Society for Optics and Photonics. SPIE, 2001, pp. 134 – 144. [Online]. Available: https://doi.org/10.1117/12.448642

[8] K. Shi, D. Boland, and G. A. Constantinides, "Efficient fpga implementation of digit parallel online arithmetic operators," in *2014 International Conference on Field-Programmable Technology (FPT)*, 2014, pp. 115–122.

[9] Y. Zhao, J. Wickerson, and G. A. Constantinides, "An efficient implementation of online arithmetic," in *2016 International Conference on Field-Programmable Technology (FPT)*, 2016, pp. 69–76.

[10] P. Kornerup, "Reviewing high-radix signed-digit adders," *IEEE Transactions on Computers*, vol. 64, no. 5, pp. 1502–1505, 2015.

[11] A. Avizienis, "Signed-digit numbe representations for fast parallel arithmetic," *IRE Transactions on Electronic Computers*, vol. EC-10, no. 3, pp. 389–400, 1961.

[12] M. D. Ercegovac, "On-Line Arithmetic: An Overview," in *Real-Time Signal Processing VII*, K. Bromley, Ed., vol. 0495, International Society for Optics and Photonics. SPIE, 1984, pp. 86 – 93. [Online]. Available: https://doi.org/10.1117/12.94401

[13] G. Jaberipur and S. Gorgin, "An improved maximally redundant signed digit adder," *Computers and Electrical Engineering*, vol. 36, no. 3, 2010. [Online]. Available: https://doi.org/10.1016/j.compeleceng.2009.12.002

[14] S. Gorgin and G. Jaberipur, "A family of high radix signed digit adders," in *2011 IEEE 20th Symposium on Computer Arithmetic*, 2011, pp. 112–120.

[15] M. D. Ercegovac and T. Lang, *Digital arithmetic.* Morgan Kaufmann, 2003, ch. 9, pp. 502–534.

[16] Ercegovac and Lang, "On-the-fly conversion of redundant into conventional representations," *IEEE Transactions on Computers*, vol. C-36, no. 7, pp. 895–897, 1987.

[17] "Quartus Prime Lite Edition (20.1.1)," Intel Corporation. [Online]. Available: https://fpgasoftware.intel.com/20.1.1/?edition=lite&platform=linux

[18] *Cyclone V Device Handbook*, Jul 2020.

[19] *Universal Verification Methodology (UVM) 1.2 User's Guide*, Oct 2015.

[20] V. Agrawal, C. Kime, and K. Saluja, "A tutorial on built-in self-test. I. Principles," *IEEE Design Test of Computers*, vol. 10, no. 1, pp. 73–82, 1993.

[21] V. P. Nelson, "Built-In Self Test." [Online]. Available: https://www.eng.auburn.edu/~nelson/courses/elec5250_6250/slides/Test_BIST.pdf

[22] J. Liang, J. Han, and F. Lombardi, "New metrics for the reliability of approximate and probabilistic adders," *IEEE Transactions on Computers*, vol. 62, no. 9, pp. 1760–1771, 2013.

[23] K. Shi, "Design of approximate overclocked datapath," Ph.D. dissertation, Imperial College London, 2015.

[24] *Cyclone V Hard Processor System Technical Reference Manual*, Jul 2021.

[25] J. J. Davis, "PRiME_OpenCL," 2017. [Online]. Available: https://github.com/PRiME-project/PRiME_OpenCL

[26] ——, "KAPow for OpenCL," 2019. [Online]. Available: https://github.com/PRiME-project/KOCL

[27] *Cyclone V Device Datasheet*, Nov 2019.

[28] H. E. Yantir, S. Bayar, and A. Yurdakul, "Efficient implementations of multi-pumped multi-port register files in fpgas," in *2013 Euromicro Conference on Digital System Design*, 2013, pp. 185–192.

[29] A. Canis, J. H. Anderson, and S. D. Brown, "Multi-pumping for resource reduction in fpga high-level synthesis," in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2013, pp. 194–197.

[30] *Altera Phase-Locked Loop (Altera PLL) IP Core User Guide*, Jun 2017.

[31] *Implementing Fractional PLL Reconfiguration With Altera PLL and Altera PLL Reconfig IP Cores*, Oct 2019.

[32] "Cyclone IV Reconfiguration parameter setting," Intel community. [Online]. Available: https://community.intel.com/t5/Programmable-Devices/Cyclone-IV-PLL-Reconfiguration-parameter-setting/m-p/228071/highlight/true#M65427

[33] "pll reconfiguration and CP/LF parameters," Intel community. [Online]. Available: https://community.intel.com/t5/Programmable-Devices/pll-reconfiguration-and-CP-LF-parameters/m-p/44278/highlight/true#M11331

[34] P. Burch, "Altera Cyclone V: Timing issues with routing (interconnect)," Electrical Engineering Stack Exchange. [Online]. Available: https://electronics.stackexchange.com/q/207359

[35] "What is metastability in an fpga?" [Online]. Available: https://www.nandland.com/articles/metastability-in-an-fpga.html

[36] *Intel®Quartus®Prime Timing Analyzer Cookbook*, Nov 2018.

[37] G. Rahav, "Multicycles Exception Between Two Synchronous Clock Domains." [Online]. Available: https://www.ee.bgu.ac.il/~digivlsi/slides/Multicycles_6_2.pdf

[38] Y. Xiao, D. Park, A. Butt, H. Giesen, Z. Han, R. Ding, N. Magnezi, R. Rubin, and A. DeHon, "Reducing fpga compile time with separate compilation for fpga building blocks," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, 2019, pp. 153–161.