Imperial College of Science, Technology and Medicine
Department of Electrical and Electronic Engineering

Hardware for Arbitrary-precision Iterative Numerical Algorithms

He Li

Supervised by Prof. George A. Constantinides and Dr. James J. Davis

Submitted in part fulfilment of the requirements for the degree of Doctor of Philosophy of Imperial College London and the Diploma of Imperial College London

Abstract

Many algorithms feature an iterative loop that converges to the result of interest. The numerical operations in such algorithms are generally implemented using finite-precision arithmetic, either fixed- or floating-point, most of which operate least-significant digit first. This results in a fundamental problem: if, after some time, the result has not converged, is this because we have not run the algorithm for enough iterations or because the arithmetic in some iterations was insufficiently precise? There is no easy way to answer this question, so users will often over-budget precision in the hope that the answer will always be to run for a few more iterations. I propose a fundamentally new approach: with the appropriate arithmetic able to generate results from most-significant digit first, this work shows that fixed compute-area hardware can be used to calculate an arbitrary number of algorithmic iterations to arbitrary precision, with both precision and approximant index increasing in lockstep. Consequently, datapaths constructed following my principles demonstrate efficiency over their traditional arithmetic equivalents where the latter's precisions are either under- or over-budgeted for the computation of a result to a particular accuracy. Further efficiency gains are realisable by inferring the superfluous digits within iterative calculations. Use of forward error analysis allows us to infer insignificant least-significant digits for stationary iterative methods. Their lack of computation is guaranteed not to affect the ability to reach a solution of any accuracy. Exploitation of most-significant digit-first arithmetic additionally enables us to declare certain digits to be identical at runtime for any iterative methods. Specific to stationary iterative methods, digit stability is inferred by combining the knowledge of identical digits shared between successive approximants with matrix conditioning. Both allow the skipping of MSD calculation. Versus arbitrary-precision iterative solvers without the optimisations I detail herein, up-to 470× performance speedups and 22× memory savings are achieved for the evaluated benchmarks.

Declaration of Originality

I herewith certify that the work presented in this thesis is my own. All material in the thesis which is not my own work has been properly referenced and acknowledged.

Copyright Declaration

Acknowledgements

First and foremost, I would like to express my biggest gratitude to my supervisors, George Constantinides and James Davis, for their continuous support and guidance during my PhD study. This thesis would not have been possible without their remarkable vision in academic research, as well as their brilliant supervision. They have shared their knowledge and ideas, always been patient and constructive with my thoughts, and more importantly, nurtured my confidence in my early research career. It has been a great pleasure and a fruitful journey working with them.

It was also an honour to have had invaluable and insightful discussions with my examiners, Geoff Merrett and David Thomas, during the viva. I am grateful for those at the Imperial CAS group—Christos Bouganis, Peter Cheung, John Wickerson, Wiesia Hsissen, Ben Chua, Shane Fleming, Junyi Liu, Nadesh Ramanathan, Jiang Su, Kan Shi, Hilda Xue—who have encouraged and helped me throughout my PhD marathon.

I would also like to thank my colleagues and friends in the FPGA and arithmetic community, especially Milos D. Ercegovac from UCLA for his helpful suggestions, and my Master's supervisor Qiang Liu at Tianjin University for encouraging me to pursue my studies at Imperial. I am also grateful for the support of the China Scholarship Council.

Last but not least, I want to give my special thanks to my parents, Guangbiao Li and Xiangling Chen, my parents-in-law, Cheng Pang and Hongjing Zhou, and my wife, Yaru Pang, for their endless love, wisdom and support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In numerical analysis, an algorithm executing on the real numbers, $\mathbb{R}$, is often expressed as a conceptually infinite iterative process that converges to a result. This is illustrated in a general form by the equation

$$\boldsymbol{x}^{(k+1)} = f\big(\boldsymbol{x}^{(k)}\big),$$

in which the computable real function $f \in \big(\mathbb{R}^N \to \mathbb{R}^N\big)$ is repeatedly applied to an initial approximation $\boldsymbol{x}^{(0)} \in \mathbb{R}^N$. The true result, $\boldsymbol{x}^*$, is obtained as k approaches infinity, *i.e.*

$$\boldsymbol{x}^* = \lim_{k\to\infty} \Pi\big(\boldsymbol{x}^{(k)}\big),$$

where the operator Π denotes the projection of the variables of interest since the result may be of lower dimensionality than N. Examples of this template include classical iterative methods such as the Jacobi and Newton's methods, as well as others including gradient descent methods [72].

In practice, these calculations are often implemented using finite-precision approximations such as that shown in Algorithm 1, wherein $\mathbb{FP}_P$ denotes some finite-

Algorithm 1 Generic finite-precision iterative algorithm.

```
Require: $\hat{\boldsymbol{x}}^{(0)} \in \mathbb{FP}_P^N$, $\hat{f} \in \left(\mathbb{FP}_P^N \to \mathbb{FP}_P^N\right)$
1: for $k = 0$ to $K - 1$ do
2:     $\hat{\boldsymbol{x}}^{(k+1)} \leftarrow \hat{f}\left(\hat{\boldsymbol{x}}^{(k)}\right)$
3: end for
Assert: $\left\|\Pi\left(\hat{\boldsymbol{x}}^{(K)}\right) - \boldsymbol{x}^*\right\| < \eta$
```

precision datatype, P is a measure of its precision (usually word length), $\hat{f}$ is a finite-precision approximation of f and η is an accuracy bound. The problem with this implementation lies in the coupling of P and iteration limit K. Generally, this algorithm will *not* be able to ensure that its assertion passes, and when it fails we are left with no knowledge as to whether K should be increased or if all computations need to be thrown away and the algorithm restarted with a higher P instead.

As a simple demonstration of this problem, suppose we wish to compute the following iteration

$$x^{(k+1)} = {}^1\!/_8 - {}^1\!/_7 \cdot x^{(k)} \tag{1.1}$$

starting from zero.

When performing this arithmetic using a standard approach in either software or hardware, we must choose a single, fixed precision for our calculations before beginning to iterate. Fig. 1.1a shows the order in which the digits are calculated when the precision is fixed to eight decimal places: approximant-by-approximant, least-significant digit (LSD) first. Choosing the right precision *a priori* is difficult, particularly with respect to hardware implementation. If it is too high, the circuit may be unnecessarily slow and power-hungry, while, if it is too low, the criterion for convergence may never be reached.

Iterating exactly

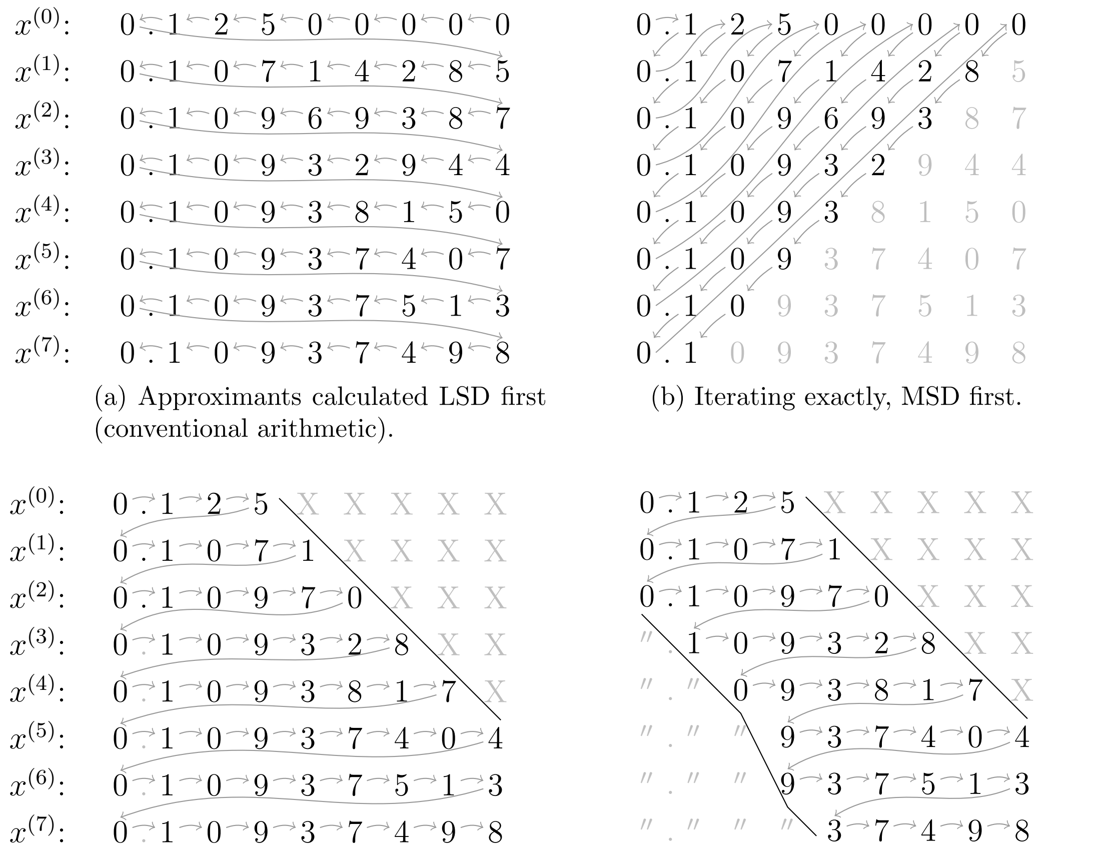
The first proposal for arbitrary-precision iterative computation, as will be presented in Chapter 3, avoids the need to answer the aforementioned question entirely. As illustrated in Fig. 1.1b, the digits are calculated in a zig-zag pattern, sweeping through iterations and decimal places simultaneously. The longer we compute, the more accurate our result will be; the computation can terminate whenever the result is accurate enough. This avoids the need to fix the precision beforehand, but requires the ability to calculate from most-significant digit (MSD) first, a facility provided through the use of *online arithmetic* [37]. While general-purpose processors featuring traditional, LSD-first arithmetic units exhibit inefficiency for the realisation of online arithmetic, field-programmable gate arrays (FPGAs) represent excellent platforms for the implementation of such MSD-first operations [73, 88, 104, 105, 122].

Don't-care digit elision

As originally formulated, iterating exactly is somewhat inefficient since the triangular shape traced out results in the computation of more digits than is actually needed. While the generation of all LSDs shown is strictly necessary to achieve exact computation in every iteration, iterative algorithms do not ordinarily require this to achieve convergence. Low-significance digits of early approximants are generally unimportant, thus they are called *don't-care* digits. In Chapter 4, a don't-care digit elision technique is presented for implementing stationary iterative calculations to arbitrary accuracies. This refines the exact iterative calculation by avoiding unnecessary computations of don't-care digits, arriving at a digit pattern shown in Fig. 1.1c. By purposefully allowing rounding error, yet carefully bounding the amount introduced in each approximant, arbitrary-accuracy results can still be obtained while generating up-to 6.6 fewer digits than iterating exactly.

Don't-change digit elision

As exemplified in Figs 1.1b and 1.1c, high-significance digits of later approximants lie


$x^{(0)}$: 0 . 1 2 5 0 0 0 0 0
$x^{(1)}$: 0 . 1 0 7 1 4 2 8 5
$x^{(2)}$: 0 . 1 0 9 6 9 3 8 7
$x^{(3)}$: 0 . 1 0 9 3 2 9 4 4
$x^{(4)}$: 0 . 1 0 9 3 8 1 5 0
$x^{(5)}$: 0 . 1 0 9 3 7 4 0 7
$x^{(6)}$: 0 . 1 0 9 3 7 5 1 3
$x^{(7)}$: 0 . 1 0 9 3 7 4 9 8
X
″

(a) Approximants calculated LSD first (conventional arithmetic).

(b) Iterating exactly, MSD first.

(c) MSD first with don't-care digit elision.

(d) MSD first with don't-change and don't-care digit elision.

Fig. 1.1. Alternative digit-calculating strategies for the solution of $x^{(k+1)} = {}^{1}\!/_{8} - {}^{1}\!/_{7} \cdot x^{(k)}$. Arrows show the order of digit generation. Fig. 1.1a shows the conventional way to converge a solution while Fig. 1.1b illustrates how the proposed arbitrary-precision iterative solver works to iterate exactly. Figs. 1.1c and 1.1d show arbitrary-precision iterative solvers with don't-care (X-marks) digit elision only and with don't-change (″-marks) and don't-care digit elision. The solid lines represent the boundaries of the regions.

in the bottom-left corner; digits of later approximants are generally identical to ones in the previous approximant, thus they are called *don't-change* digits. In Chapter 5, two theoretical analyses are proposed to infer don't-change digits. To begin, a generic analysis is introduced by exploiting online arithmetic's digit dependency to determine identical MSDs. Thereafter, a specialised analysis is introduced by using interval and forward error analyses to prove that digits of high significance will become stable during stationary iterative calculations. By detecting the presence and avoiding the recomputation of these digits, as well as the elision of don't-care ones, we end up with a corridor of digits, which is shown in Fig. 1.1d.

The proposed architecture, coined ARCHITECT (for **Ar**bitrary-precision **C**onstant-**h**ardware **Ite**rative **C**ompu**t**e), is the first to allow the runtime adaption of both precision and iteration count for iterative algorithms implemented in hardware. Several research questions that manifest from ARCHITECT's principles are addressed in this thesis:

- Can a fixed-sized datapath be used to compute results to arbitrary precision while still achieving comparable performance versus conventional arithmetic equivalents?
- Can the locations of unimportant LSDs in early approximants be inferred, excluding them from calculation to increase performance, while still obtaining arbitrary-accuracy results?
- How is it possible to avoid recomputing digits of later approximants that are identical to those of previous approximants?

To achieve these goals, the novel contributions of this thesis are as follows.

Hardware Architecture

- The first fixed-compute-resource hardware for iterative calculation capable of producing arbitrary-precision results after arbitrary numbers of iterations. (Chapter 3)
- An optimised mechanism for digit-vector storage based on a Cantor pairing function to facilitate simultaneously increasing precision and iteration count. (Chapter 3)

- To complement digit elision strategies, enhanced memory-addressing schemes are proposed, leading to greater performance and higher achievable result accuracy for a given memory budget. (Chapters 4 & 5)

Theoretical Analysis

- A theorem for the optimal rate of LSD growth per iteration within stationary iterative methods, thereby enabling the preclusion of don't-care digit computation. With the appropriate preconditions, this is proven to have no bearing on the chosen method's ability to reach a solution of any accuracy. (Chapter 4)
- Theoretical analysis of MSD stability within any online arithmetic-implemented iterative method, facilitating runtime detection of don't-change digits. (Chapter 5)
- Using interval and forward error analyses, a theorem for the rate of stable MSD growth within the approximants produced by any stationary iterative method. (Chapter 5)

Benchmarking and Evaluation

- A library of arbitrary-precision operations, enabling the construction of datapaths for iterative methods to solve linear and nonlinear equations. (Chapter 3)
- Exemplary hardware implementations of the proposals for the computation of both linear (Jacobi method) and nonlinear (Newton) iterations. (Chapters 3, 4 & 5)
- Qualitative and quantitative performance and scalability comparisons against traditional LSD-first and state-of-the-art online arithmetic FPGA implementations. (Chapters 3, 4 & 5)

In order to evaluate ARCHITECT, two widely used iterative algorithms were implemented in hardware—the Jacobi method (to solve systems of linear equations) and Newton's method (for the solution of nonlinear equations)—following the aforementioned principles. Jacobi and Newton were chosen to exemplify a large class of iterative methods with linear and quadratic convergence properties, respectively. Our emphasis herein is to focus on a small number of methods and treat them in depth.

1.1 Thesis Outline

The remainder of this thesis is organised in the following manner.

- **Chapter 2** highlights the theoretical background and reviews the work relevant to this thesis. Since this thesis is focussed upon arbitrary precision iterative computation, an overview of iterative methods for linear and nonlinear equations is initially presented. Following this, state-of-the-art arbitrary-precision computing techniques are surveyed. Finally, some necessary background of redundant number systems and online arithmetic are introduced, along with a detailed discussion of the benefits and limitations of online arithmetic.
- **Chapter 3** presents a fixed-area hardware architecture for arbitrary-precision iterative computing. A Cantor pairing function is employed to allow the simultaneous increase of both iteration indices and precision, enabling the traversal of two-dimensional iteration-precision space in an unconventional fashion. A library of arbitrary-precision operators is introduced. Exemplary hardware implementations are constructed for the computation of both linear (Jacobi method) and nonlinear (Newton) iterations.

- **Chapter 4** investigates how to determine don't-care digits with stationary iterative methods and avoid their computations. Forward error analysis is used to infer the locations of don't-care digits. I demonstrate that the lack of don't-care digit computation is guaranteed not to affect the ability to reach a solution of any accuracy. Performance speedups and memory footprint reductions are achieved versus iterative solvers without this optimisation.

- **Chapter 5** investigates how to infer the presence and elide the computation of don't-change digits for iterative calculations. I first employ online arithmetic's computation dependencies to infer identical MSDs through the runtime comparison of consecutive approximants. This method is unable, however, to guarantee that digits will stabilise, *i.e.* never change in any future iteration. Therefore, I also address this shortcoming by using interval and forward error analysis to prove that digits of high significance will become stable within stationary iterative methods. I formalise the relationship between matrix conditioning and the rate of growth in MSD stability, using this information to converge to desired results more quickly.

- **Chapter 6** concludes the thesis and suggests possible fruitful avenues for future research, such as optimal digit trajectories for iterative computation and methods for combining the arbitrary-precision computation with high-level synthesis (HLS).

1.2 Published Work

The original contributions of this thesis have been published in the following peer-reviewed conference papers and journal articles as follows.

- He Li, J. Davis, J. Wickerson and G. A. Constantinides. ARCHITECT: Arbitrary-precision Constant-hardware Iterative Compute, in IEEE International Conference on Field-Programmable Technology (FPT), 2017. (**Best Paper Presentation Award**)

- He Li, J. Davis, J. Wickerson and G. A. Constantinides. Digit Elision for Arbitrary-accuracy Iterative Computation, in IEEE Symposium on Computer Arithmetic (ARITH), 2018.

- He Li, J. Davis, J. Wickerson and G. A. Constantinides. ARCHITECT: Arbitrary-precision Hardware with Digit Elision for Efficient Iterative Compute. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2019.

- He Li, I. Mcinerney, J. Davis and G. A. Constantinides. Digit Stability Inference for Iterative Methods using Redundant Number Representation. IEEE Transactions on Computers, 2020.

Chapter 2

Background

The aim of this thesis is to construct a hardware architecture for arbitrary-precision iterative calculations using online arithmetic. In this chapter, some related work and concepts of iterative methods, arbitrary-precision computing, redundant number systems and online arithmetic are presented. In general, I restrict the discussions to four main research topics:

- Section 2.1 presents the relevant background of iterative methods for linear and nonlinear equations, including the basic concepts, iterative method classifications and an overview of state-of-the-art hardware implementations of different iterative methods on FPGAs.

- Section 2.2 discusses the motivation to pursue arbitrary-precision computing, and provides an overview of state-of-the-art arbitrary-precision software packages and hardware architectures.

- Section 2.3 presents the concepts of redundant number representation and discusses an overview of arithmetic circuits using redundant number systems.

- Section 2.4 reviews the concepts of online arithmetic algorithms, provides a brief survey of online arithmetic-based applications and discusses the benefits and limitations of online arithmetic.

2.1 Iterative Methods for Linear and Nonlinear Equations

The problem of solving systems of linear and nonlinear equations is among the most important in many scientific and engineering fields such as applied mathematics, physics, computer science, finance and astronomy [26]. The idea of using iterative methods to solve these problems has been put forward and has become one of the widely used techniques for hundreds of years [47]. A typical iterative method produces successive approximate answers to converge to the solution of the problem, with a relatively simple calculation per iteration [47].

Iterative methods can be classified by their rates of convergence. For example, the Jacobi method is a classical iterative algorithm with linear convergence, which is usually used to solve systems of linear equations [66]; Newton's method is a classical algorithm with quadratic convergence and forms the basis of most modern optimisation algorithms [69].

2.1.1 Basic Concepts

To begin, a brief overview of vector and matrix norms is presented as they are usually used for termination criteria and error analysis of iterative methods. A vector norm $\|\boldsymbol{x}\|$ can be considered as the magnitude of a vector $\boldsymbol{x} \in \mathbb{R}^n$, if satisfies three properties:

$$\begin{aligned}\|\boldsymbol{x}\| &\le 0 \quad \text{with equality only if } \boldsymbol{x} = \boldsymbol{0} \\ \|a\boldsymbol{x}\| &= |a|\|\boldsymbol{x}\| \\ \|\boldsymbol{x}+\boldsymbol{y}\| &\le \|\boldsymbol{x}\| + \|\boldsymbol{y}\| \quad \text{(triangle inequality)},\end{aligned}$$

where $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$ and a is a scalar.

A matrix norm of an $m \times n$ matrix $\boldsymbol{A}$ is defined as a mapping from $\mathbb{R}^{m\times n}$ to $\mathbb{R}$, such that the following three properties hold:

$$\begin{aligned}\|\boldsymbol{A}\| &\le 0 \quad \text{with equality only if } \boldsymbol{x} = \boldsymbol{0} \\ \|a\boldsymbol{A}\| &= |a|\|\boldsymbol{A}\| \\ \|\boldsymbol{A}+\boldsymbol{B}\| &\le \|\boldsymbol{A}\| + \|\boldsymbol{B}\| \quad \text{(triangle inequality)},\end{aligned}$$

where $\boldsymbol{A}, \boldsymbol{B} \in \mathbb{R}^{m\times n}$.

For square matrices (*i.e.*, $m = n$), matrix norms that satisfy the additional following condition are called *sub-multiplicative* [113]:

$$\|\boldsymbol{AB}\| \le \|\boldsymbol{A}\|\|\boldsymbol{B}\|.$$

For example, the l_∞-matrix norm for an $n \times n$ matrix is sub-multiplicative [24]:

$$\|\boldsymbol{A}\|_\infty = \max_{0\le i\le n-1} \sum_{j=0}^{n-1} |a_{ij}| \quad l_\infty\text{-matrix norm.}$$

This sub-multiplicative matrix norm will be used for my don't-care digit analysis in Chapter 4 and digit stability analysis in Chapter 5.

2.1.2 Iterative Methods for Linear Equations

Iterative methods for the solution of linear equations play a significant role in many scientific computing scenarios, such as computational fluid mechanics, economic modelling and oil reservoir modelling [16]. Iterative methods for linear equations can generally be classified into two types, stationary and nonstationary methods (*e.g.,* Krylov methods) [65]. Versus nonstationary ones, stationary iterative methods are classical, simple to implement and can serve as preconditioners for more advanced methods, such as the preconditioned conjugate gradient (PCG) method [115]. Therefore, we review stationary iterative methods in detail in the following.

Stationary Iterative Methods

In numerical linear algebra, a straightforward way to solve a system $\boldsymbol{Ax} = \boldsymbol{b}$ is to transform it into a linear fixed-point iteration of the form

$$\boldsymbol{M}\boldsymbol{x}^{(k+1)} = \boldsymbol{N}\boldsymbol{x}^{(k)} + \boldsymbol{b} \tag{2.1}$$

with $\boldsymbol{A} = \boldsymbol{M} - \boldsymbol{N}$ and $\boldsymbol{M}$ non-singular [51]. Defining iteration matrix $\boldsymbol{G} = \boldsymbol{M}^{-1}\boldsymbol{N}$, (2.1) can also be written as

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{G}\boldsymbol{x}^{(k)} + \boldsymbol{M}^{-1}\boldsymbol{b}. \tag{2.2}$$

Achievement of convergence requires that $\boldsymbol{G}$'s spectral radius $\rho(\boldsymbol{G}) < 1$. Such *stationary iterative methods* are widely used in the approximate solution of nonlinear [82], differential [67] and integral equations [121]. They also play a significant role in multigrid theory [83]; multigrid methods commonly serve as preconditioners for many other iterative algorithms [8].

Some well known stationary iterative methods include the Jacobi, Gauss-Seidel and successive over-relaxation (SOR) methods [52]. Let us now analyse the iteration matrix $\boldsymbol{G}$ for different stationary iterative methods. For example, consider the Jacobi method to solve $\boldsymbol{Ax} = \boldsymbol{b}$. This method uses the matrix splitting

$$\boldsymbol{M} = \boldsymbol{D}, \quad \boldsymbol{N} = -\left(\boldsymbol{L} + \boldsymbol{U}\right),$$

where $\boldsymbol{D}$, $\boldsymbol{L}$ and $\boldsymbol{U}$ are the diagonal, (strict) lower-triangular and upper-triangular parts of $\boldsymbol{A}$, respectively [65]. The iteration matrix of the Jacobi method is

$$\boldsymbol{G}_{\mathrm{J}} = -\boldsymbol{D}^{-1}\left(\boldsymbol{L} + \boldsymbol{U}\right).$$

For the Gauss-Seidel method, its iteration matrix is characterised as a new iteration with matrix splitting

$$\boldsymbol{M} = \boldsymbol{D} + \boldsymbol{L}, \quad \boldsymbol{N} = -\boldsymbol{U},$$

and the iteration matrix

$$\boldsymbol{G}_{\mathrm{GS}} = -\left(\boldsymbol{D} + \boldsymbol{L}\right)^{-1}\boldsymbol{U}.$$

To accelerate the Gauss-Seidel method, researchers modified it by introducing a relaxation parameter ω, leading to the successive overrelaxation (SOR) method with an iteration matrix

$$\boldsymbol{G}_{\mathrm{SOR}} = \left(\boldsymbol{D} + \omega\boldsymbol{L}\right)^{-1}\left(\left(1-\omega\right)\boldsymbol{D} - \omega\boldsymbol{U}\right).$$

Note that the Gauss-Seidel method is a special case ($\omega = 1$) of the SOR method. The performance of the SOR method can be improved by a good choice of ω, even though the choice is non-trivial [120].

Other Methods

There are other types of iterative method to solve linear equations, such as Krylov subspace methods. Unlike stationary iterative methods, Krylov methods do not require a fixed iteration matrix $\boldsymbol{G}$ [66]. Examples of Krylov subspace methods include the conjugate gradient method, the generalised minimum residual method (GMRES) and the minimum residual method (MINRES) [66].

2.1.3 Iterative Methods for Nonlinear Equations

Newton's method is a classical iterative method to solve systems of nonlinear equations. The algorithm was created by Sir Issac Newton, who formulated the result in 1669, and was later improved by Joseph Raphson in 1690. Hence, the algorithm is also occasionally called the Newton-Raphson method [21].

Newton's method is a root-finding algorithm, commonly employed to approximate the zeros of a real-valued function f. The $(k+1)^{\text{th}}$ approximant is given by

$$x^{(k+1)} = x^{(k)} - \frac{f\big(x^{(k)}\big)}{f'\big(x^{(k)}\big)},$$

where f' is the first derivative of f. If it converges, then the final solution that Newton's method converges to depends on the initial guess $x^{(0)}$.

Given a differentiable function f, formally the Newton iteration function $g(x)$ is defined as

$$g(x) = x - \frac{f(x)}{f'(x)}.$$

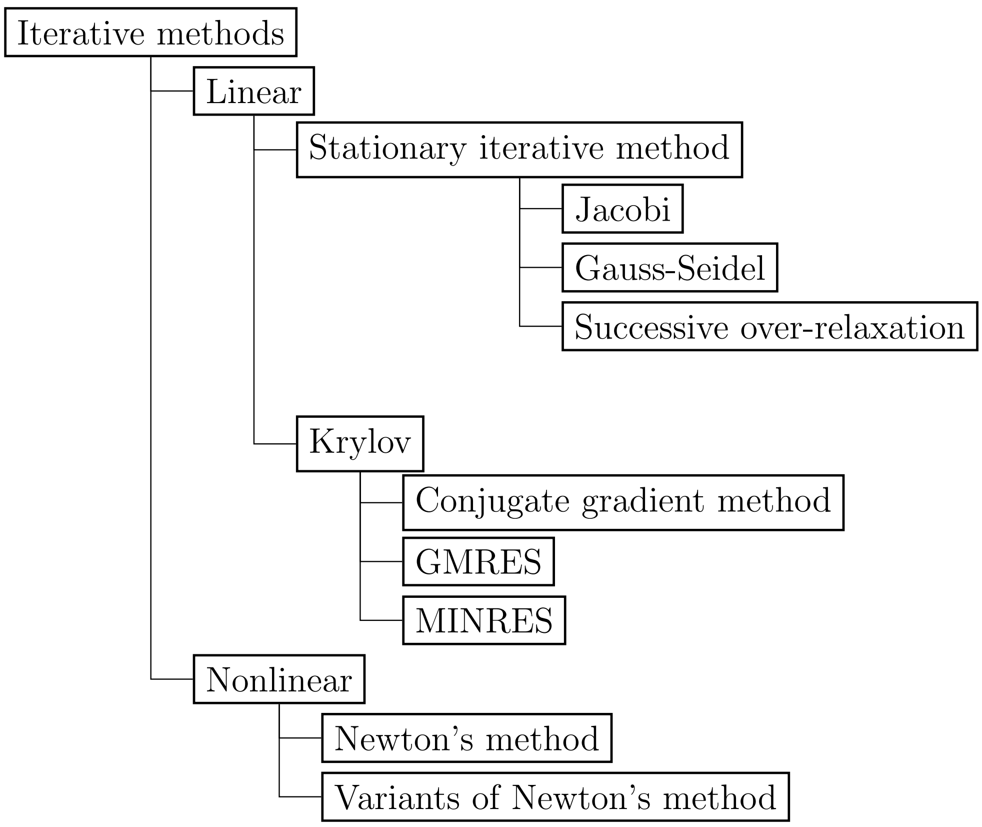

Iterative methods
Linear
Stationary iterative method
Jacobi
Gauss-Seidel
Successive over-relaxation
Krylov
Conjugate gradient method
GMRES
MINRES
Nonlinear
Newton's method
Variants of Newton's method

Fig. 2.1. A brief overview of iterative methods.

Then

$$\begin{aligned} x^{(1)} &= g\big(x^{(0)}\big) \\ x^{(2)} &= g\big(x^{(1)}\big) = g\big(g\big(x^{(0)}\big)\big) = g^2\big(x^{(0)}\big), \end{aligned}$$

and, in general

$$g\big(x^{(k+1)}\big) = g^{k+1}\big(x^{(0)}\big),$$

where the notation $g^{k+1}(\cdot)$ means applying $g(\cdot)$ up to $k+1$ times. If the initial guess $x^{(0)}$ is sufficiently close to the solution x^* and $g'(x^*) = 0$, Newton's method converges quadratically [21, 60].

In some cases, such as when the root multiplicity is greater than one, Newton's method only converges linearly [21]. There are several variants of Newton's method, such as the chord method, Broyden's method [66] and the Quasi-Newton (QN) method [29]. Some such variants have been used for neural network training [76, 77]. Fig 2.1 presents an overview of different categories of iterative methods to solve linear and nonlinear equations.

2.1.4 Hardware Implementations of Iterative Methods

Given that iterative methods are particularly popular in scientific computing, machine learning, optimisation and many other numerical application areas, interest in their acceleration with FPGAs is growing [86]. Recent studies have demonstrated that FPGAs are promising platforms for the acceleration of the Jacobi [107], Newton's [78], conjugate gradient [101] and MINRES [10] methods.

Implementations relying on traditional arithmetic—whether digit-serial or -parallel—enforce compile-time determination of precision. For digit-parallel designs, this affects their area and input/output (I/O) bandwidth requirements, whereas for digit-serial, it is one of the factors affecting algorithm runtime. Since digit-parallel arithmetic's I/O throughput requirements grow with precision, digit-serial arithmetic has inherently lower I/O consumption. Runtime tuning of precision in iterative calculations was enabled through the use of online arithmetic in recent work [122], however unrolling was necessary in order to implement an iterative algorithm's loop; area therefore scaled with the desired number of iterations. As shown in Table 2.1, the work presented in this thesis stands apart from these alternatives by enabling the runtime selection of both factors affecting accuracy.

TABLE 2.1
COMPARISON OF ARITHMETIC PARADIGMS FOR ITERATIVE ALGORITHMS.

Name	Area scales with		Runtime scales with	
	Precision	Iteration limit	Precision	Iteration limit
Digit-parallel	✔	✘	✘	✔ unbounded
Digit-serial	✘	✘	✔ bounded	✔ unbounded
Zhao *et al.* [122]	✘	✔	✔ unbounded	✘
This work	✘	✘	✔ unbounded	✔ unbounded

2.2 Arbitrary-precision Computing

Applications requiring very high precision have become increasingly popular in recent years, especially in the current scientific computing community [94]. For example, today, hundreds of digits of precision are required in atomic system simulations and electromagnetic scattering theory calculations [5], while Ising integrals and elliptic function evaluation need thousands of digits [4]. In experimental mathematics, precisions higher than double precision are required [39]. Poisson equation computations frequently require results to tens or hundreds of thousands of digits precision [6]. Researchers in the climate modelling community demonstrated that high-precision arithmetic can solve numerical inaccuracy in certain loops of climate modeling programs [7]. Standard numeric datatypes, such as double- or even quadruple-precision floating point numbers, are no longer sufficient in an increasing number of scenarios.

2.2.1 Software Libraries

Many software libraries have been developed for arbitrary-precision arithmetic. The *de facto* standard is MPFR (Multiple-Precision Floating Point with Correct Rounding), which guarantees correct rounding to any requested number of bits, selected before each operation is executed [89]. Other state-of-the-art arbitrary-precision computation packages include MPFI (Multiple Precision Floating-point Interval) [100], the complex MPFR extension MPC [30] and Arb (Arbitrary-Precision Midpoint-Radius Interval Arithmetic) [59]. Ndour *et al.* proposed a RISC-V instruction set architecture (ISA) extension with variable precision floating-point capabilities [93].

2.2.2 Hardware Implementations

Hardware architectures of high-precision operations have been put forward recently, in particular those within iterative algorithms [86]. FPGAs provide flexibilities not available on other platforms, allowing for the implementation of bespoke designs with many precision and performance tradeoffs [43]. Libraries including FloPoCo [28] and VFLOAT [41], alongside proprietary vendor tools, facilitate the creation of custom-precision arithmetic IP cores. Although they provide the designer with many options to suit particular frequency, latency and resource usage requirements, precision is determined at compile time and therefore remains fixed during operation. Mixed-precision iterative solvers, in which precisions can be selected from a set as required at runtime, have also been proposed [56, 79, 107]. A dual-mode double- and quadruple-precision architecture has also been designed; this was based on the Taylor series expansion [56]. Liu *et al.* proposed a novel FPGA-based Markov Chain Monte Carlo construction that exploited the mixed-precision support of FPGAs in order to accelerate the computation of likelihood functions [79]. Sun *et al.* proposed an FPGA-based mixed-precision linear solver: as many operations as possible are performed in low precision before switching to a slower, higher-precision mode for the later iterations [107]. The Unum “universal number” is an arithmetic and binary representation format analogous to floating point [46]. Bacco *et al.* pro-

Table 2.2
Comparison of arbitrary-precision arithmetic techniques.

Name	Level	Precision set per calculation	Iteration limit set per calculation
MPFR [89], *etc.*	Software	Before	During
FloPoCo [28], *etc.*	Hardware	Before	During
Mixed-precision [56, 79, 107]	Hardware	Before	During
SMURF [9]	Hardware	Before	During
Zhao *et al.* [122]	Hardware	During	Before
This work	Hardware	During	During

posed SMURF—a Unum-based variable-precision floating-point unit—implemented as a coprocessor of a RISC-V RocketChip core [46]. Zhao *et al.*'s work enables arbitrary-precision computation but, as mentioned previously, requires compile-time determination of iteration count [122].

Apart from Zhao *et al.*'s proposal, each of the aforementioned proposals requires precision—or precisions—to be determined *a priori*. In many cases, this is not a trivial task; making the wrong choice often means having to throw the calculations already done away and starting from scratch with higher precision, wasting both time and energy. In our work, we are particularly interested in hardware architectures that allow precision to be increased over time without having to restart computation or modify the circuitry. Table 2.2 presents a side-by-side comparison of these techniques and their features with this work, the only entry supporting the determination of precision and iteration count *after each calculation has commenced.*

When implementing arbitrary-precision arithmetic operators as fundamental units in datapaths on FPGAs, different number systems can affect the overall computation performance. In the following, an overview of arithmetic circuits with different number systems is presented.

2.3 Arithmetic Circuits with Different Number Systems

The performance of many custom hardware systems is predominantly dependent upon the speed of their underlying arithmetic operators [116]. When these employ conventional, nonredundant number representations, carry propagation is the primary factor jeopardising their latency. The introduction of redundancy, however, often allows operator execution time to be shortened due to the reduction—and sometimes even elimination—of carry chains [55]. For example, a redundant number system is the setting for the well-known Sweeney-Robinson-Tocher (SRT) division algorithm [20, 117], used in the Intel Pentium processor [50].

2.3.1 Traditional Number Systems

Before introducing redundant number systems, some basic concepts of traditional number systems must be reviewed. Traditional number systems are nonredundant, and can be characterised by the radix r and precision p [97]. Given a radix r, a digit can be selected from an integer set S to represent a number. For a nonredundant number system, we have

$$S = \{0, 1, 2, \cdots, r-1\}.$$

A number x in the range $[0, 1)$ is then $\sum_{i=1}^{p} x_i r^{-i}$, where $x_i \in S$ for all i. The digit x_i is the i^{th} MSD of this number. Every value within such a system has a unique representation.

2.3.2 Redundant Number Systems

In a redundant number system, the representation associated with a value is no longer unique [3]. In other words, multiple representations may be linked to the same value. This property allows more flexibility in the choice of representations for each value. Given a radix r, each digit can represent more than r values.

Signed-digit Number System

As a widely used redundant number system, the signed-digit number system was designed for the purpose of performing totally parallel addition [45]. The computation time of a parallel addition is therefore independent of the operand data width, since the chains of carry propagation are eliminated [81].

For a signed-digit number representation, digit values are selected from the set

$$S_{\text{sd}} = \{-\rho_1, \cdots, -1, 0, 1, \cdots, \rho_2\},$$

where $\rho_1 + \rho_2 + 1 \geq r$ and $\rho_1, \rho_2 \geq 0$. The digit set S_{sd} can be symmetric or asymmetric with different levels of redundancy. For example, a radix-4 redundant digit set can be formed in one of the following cases.

- Symmetric with minimal redundancy ($\rho_2 = \rho_1 = {}^{r}/_{2}$, assuming even r):

$$\{-2, -1, 0, 1, 2\}.$$

- Symmetric with maximal redundancy ($\rho_2 = \rho_1 = r - 1$):

$$\{-3, -2, -1, 0, 1, 2, 3\}.$$

Table 2.3
Classical representation of signed-digit numbers.

x_i^+	x_i^-	$x_i = x_i^+ - x_i^-$
1	0	1
1	1	0
0	0	0
0	1	-1

- Asymmetric ($\rho_2 \neq \rho_1$): *e.g.*

$$\{-1, 0, 1, 2, 3\}.$$

Note that S_{sd} is possible with over redundancy when $\rho_1 + \rho_1 > r$ [102].

In this thesis, we selected a radix-2 symmetric signed-digit set with maximal redundancy, since it is a standard redundant number system in hardware implementation [37].

2.3.3 Hardware Implementations

For the *de facto* standard radix-2 signed-digit number representation, the i^{th} digit of a number x, x_i, lies in $\{-1, 0, 1\}$. In hardware, each x_i corresponds to a pair of bits, x_i^+ and x_i^-, selected such that $x_i = x_i^+ - x_i^-$. The classical coding of this signed-digit number system is listed in Table 2.3.

Use of redundant number systems allows us to accelerate arithmetic circuits on hardware [2,27,40,64,110], in particular those with signed-digit representation. Although these arithmetic circuits necessitate the use of redundant number representations, data can be efficiently converted between non-redundant and redundant forms using well known on-the-fly conversion techniques [37].

Fast multipliers have been designed using signed-digit representation for partial product generation [27] and partial product reduction [64]. We further review state-of-the-art research employing signed-digit representation for other operations.

- *Addition.* Residue number system (RNS) arithmetic has attracted significant attention for high-speed arithmetic operations [58]. Timarchi *et al.* proposed a signed-digit residue number system (SD-RNS) to eliminate the carry propagation in RNS arithmetic. To achieve further performance increase, they used high-radix signed-digit (HRSD) coding to build an efficient HRSD-RNS adder.

- *Constant vector multiplication.* As a basic operation in filtering and convolution, constant vector multiplication can be implemented using multiplication-free techniques by adding and subtracting a series of power-of-two results [40]. Related hardware implementations have been investigated for many years. Fan *et al.* recently optimised a signed-digit constant vector multiplication by reducing the computational complexity of their multiplier-free technique [40].

- *Division.* Effort has been made to design fast and low-cost multipliers and adders. However, there have only been a few promising studies for division acceleration [54, 56, 63, 80]. Digit-recurrence division is the most widely used in today's high-performance microprocessors, since it presents a good balance between performance, area and energy consumption [17]. Bruguera presented a latency- and area-efficient radix-64 digit-recurrence division by overlapping simpler radix-4 iterations [18]. Amanollahi and Janeripur employed redundant number representations for partial remainders and quotient digits in radix-16 division algorithms [2]. By doing so, fast carry-free computation of the next partial remainder is achieved and less number of divisor multiples are required.

We will present an arbitrary-precision division architecture in Chapter 3.

The majority of applications using redundant number systems focus on fast arithmetic designs [55]. There are several other specialised applications, such as MSD-first computation [37] and function evaluation [32].

While the majority of existing applications employing such representations focus on speed and efficiency, the work I describe in Chapter 5 is the first to use redundancy in order to infer digit stability within iterative algorithms.

2.4 Online Arithmetic

Achieving arbitrary-precision computation with fixed hardware requires MSD-first input consumption and output generation. A suitable proposal for this, widely discussed in the literature, is online arithmetic [37]. By employing redundancy in the number representations of online arithmetic, allowing less-significant digits to correct errors introduced in those of higher significance, all online operators are able to function in an MSD-first fashion.

Recently, Ercegovac has presented a brief overview of the properties of MSD-first arithmetic [34]. To provide a detailed background of online arithmetic, this section introduces online arithmetic algorithms and surveys related hardware implementations of online arithmetic. Finally, the benefits and limitations of online arithmetic are discussed.

2.4.1 Basic Algorithms

Online operators are classically serial, however efficient digit-parallel (unrolled) implementations targetting FPGAs have been developed as well [104]. We make use of both digit-serial and -parallel online operators in this thesis.

Of particular significance to the material presented in this thesis is the concept of *online delay*. When performing an online operation, the digits of its result are generated at the same rate as its input digits are consumed, but the result is delayed by a fixed number of digits, denoted δ. That is, the first (*i.e.* most-significant) q digits of an operator's result are wholly determined by the first $q + \delta$ digits within each of its operands [37]. The value of δ is operation-specific, but is a small integer determined by the redundancy factor and the radix [37].

Online Addition

A classic online adder makes use of full adders and registers to add digits of inputs x and y, presented serially as x_{in} and y_{in}, as shown in Figure 2.2 (left), from most to least significant [37]. Digits of z start to appear at serial output z_{out} after two clock cycles; this is the online delay of the adder, denoted δ_+. Duplication of the serial adder P times and the removal of its registers lead to the creation of a P-digit parallel online adder devoid of online delay, as shown in Figure 2.2 (right) [37]. Crucially, while carry digits are presented at the least-significant end of the adder and generated at the most, there is no carry chain; the critical path lies across two full adders [104]. This indicates the adder's suitability for the construction of more complex online operators.

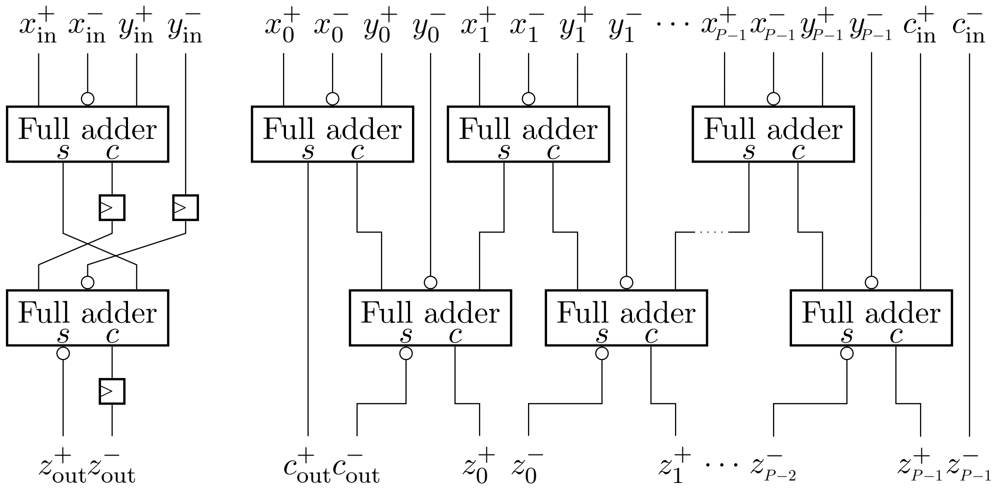

x^+_{in} x^-_{in} y^+_{in} y^-_{in}
x^+_0 x^-_0 y^+_0 y^-_0 x^+_1 x^-_1 y^+_1 y^-_1 $\cdots$ x^+_{P-1} x^-_{P-1} y^+_{P-1} y^-_{P-1} c^+_{in} c^-_{in}
Full adder
s c
z^+_{out} z^-_{out}
c^+_{out} c^-_{out}
z^+_0 z^-_0 z^+_1 $\cdots$ z^-_{P-2} z^+_{P-1} z^-_{P-1}

Fig. 2.2. Radix-2 online adders. Left: serial. Right: parallel.

Online Multiplication

Algorithm 2 illustrates classical radix-2 online multiplication of signed operands x and y, with product z in the range $(-1, 1)$ [37]. Digit vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ are assembled from digits of inputs x and y over time from most-significant first; $\|$ represents concatenation performed such that

$$\boldsymbol{x} = \sum_{i=0}^{j} x_i 2^{-i-1}, \quad \boldsymbol{y} = \sum_{i=0}^{j} y_i 2^{-i-1}$$

during cycle j. Digit-selection function $\text{sel}_\times$ serves to determine the digits of output z [37]. This is defined to be

$$\text{sel}_\times(\boldsymbol{v}) = \begin{cases} 1 & \text{if } \boldsymbol{v} \geq 1/2 \\ 0 & \text{if } -1/2 \leq \boldsymbol{v} < 1/2 \\ -1 & \text{otherwise.} \end{cases}$$

z_j is produced at cycle $j + 3$ since $\delta_\times$ is 3. P-digit online addition lies at the heart of the algorithm; due to its fixed width, the hardware that implements Algorithm 2

Algorithm 2 Radix-2 online multiplication.

```
Inputs: serially presented digits $x$, $y$
1: $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w} \leftarrow 0$
2: for $j = 0$ to $P + 2$ do
3:     $\boldsymbol{y} \leftarrow \boldsymbol{y} \parallel y_j$
4:     $\boldsymbol{v} \leftarrow 2\boldsymbol{w} + 2^{-3}(\boldsymbol{x}y_j + \boldsymbol{y}x_j)$
5:     $z_{j-3} \leftarrow \mathrm{sel}_{\times}(\boldsymbol{v})$
6:     $\boldsymbol{w} \leftarrow \boldsymbol{v} - z_{j-3}$
7:     $\boldsymbol{x} \leftarrow \boldsymbol{x} \parallel x_j$
8: end for
Output: serially generated digits $z$
```

can multiply to a precision of at most P, which must be fixed in advance.

Algorithm 3 Radix-2 online division.

```
Inputs: serially presented dividend $x$, divisor $y$
1: $\boldsymbol{y}, \boldsymbol{w}, \boldsymbol{z} \leftarrow \mathbf{0}$
2: for $j = 0$ to $P + 3$ do
3:     $\boldsymbol{y} \leftarrow \boldsymbol{y} \parallel y_j$
4:     $\boldsymbol{v} \leftarrow 2\boldsymbol{w} + 2^{-4}(x_j - \boldsymbol{z}y_j)$
5:     $z_{j-4} \leftarrow \mathrm{sel}_{\div}(\boldsymbol{v})$
6:     $\boldsymbol{w} \leftarrow \boldsymbol{v} - z_{j-4}\boldsymbol{y}$
7:     $\boldsymbol{z} \leftarrow \boldsymbol{z} \parallel z_{j-4}$
8: end for
Output: serially generated quotient $z$
```

Online Division

The process of classical radix-2 online division is shown in Algorithm 3, in which dividend x and divisor y are used to produce quotient z. In contrast to Algorithm 2, division requires the formation of digit vector $\boldsymbol{z}$ since all prior output digits are needed for the calculation of $\boldsymbol{v}$, while updates to $\boldsymbol{w}$ require the full history of y. Online division therefore has more complex computation dependencies than multiplication. Its digit-selection function, $\mathrm{sel}_{\div}$, is

$$\mathrm{sel}_{\div}(\boldsymbol{v}) = \begin{cases} 1 & \text{if } \boldsymbol{v} \geq 1/4 \\ 0 & \text{if } -1/4 \leq \boldsymbol{v} < 1/4 \\ -1 & \text{otherwise.} \end{cases}$$

z_j is produced at cycle $j+4$ since $\delta_{\div}$ is 4.

2.4.2 Related Work in Online Arithmetic

Online Arithmetic Operations

There is a long history of development of online arithmetic operators, such as fixed-point [111], floating-point [118, 119] and complex number system [19] multiplication and division, as well as MSD-first Coordinate Rotation DIgital Computer (CORDIC) [48, 108], *etc.* Recently, hardware acceleration of online algorithms have been increasingly popular, in particular those with efficient mapping [104], multiple operands [62, 114], high radices [61], variable precision [109] and overclocking [103]. Table 2.4 presents a summary of different online arithmetic implementations. FPGAs represent an appropriate platform for realising online arithmetic due to their flexible fabrics, devoid of the costs associated with application-specific integrated circuit (ASIC) implementation. Shi *et al.* presented an efficient digit-parallel implementation of online addition and multiplication [104] and explored the accuracy-performance tradeoff facilitated through overclocking [103]. Multi-operand multiplication [62] was designed recently, resulting in savings in resource usage and interconnection complexity versus a LSD-first arithmetic equivalent. Moreno *et al.* also presented a multi-operand online addition with an efficient conversion of multi-

TABLE 2.4
SUMMARY OF ONLINE ARITHMETIC IMPLEMENTATIONS.

Name	Operation	Technical feature
Joseph & Devanathan [61]	$\times$	High radix
Joseph & Devanathan [62]	$\times$	Multi-operand multiplication
Shi *et al.* [103]	$+, \times$	Overclocking
Shi *et al.* [104]	$+, \times$	Digit-parallel implementation
Moreno *et al.* [114]	$+$	Multi-operand and multi-format addition
Zhao *et al.* [122]	$+, \times, \div$	Arbitrary precision
This work	$+, \times, \div$	Arbitrary precision and iteration

format data [114]. Joseph and Devanathan built a high-radix online multiplier and demonstrated throughput boost and online delay reduction versus classical radix-2 online multipliers, with the sacrifice of a growing power-delay product [61]. Zhao *et al.* presented a novel architecture for arbitrary-precision online operations through hardware reuse [122].

Applications Using Online Arithmetic

There has been some work using online arithmetic in various application domains. The most popular is digital signal processing (DSP), named *online* signal processing [71]. Online arithmetic-implemented digital filters have been explored, such as the finite impulse response (FIR) filter [42, 44], infinite impulse response (IIR) filter [12, 36], discrete Fourier transform (DFT) [70, 71], fast Fourier transform (FFT) [84, 92] and discrete cosine transform (DCT) [70, 71]. These designs are able to produce their MSDs first and allow the subsequent computations to commence earlier, which can achieve high performance thanks to digit-serial designs.

Online arithmetic is also used in communication systems to truncate computations dynamically for power-area product savings [98, 99]. Rajagopal and Cavallaro employed this MSD-first feature for sign detection in communication systems [99]. Computation can be terminated when the first non-zero MSD arrives, while with conventional LSD-first arithmetic, the MSD is produced at the end of the entire computation. Therefore, online arithmetic provides easier sign detection.

Another important application based on online arithmetic is function evaluation [13–15, 31, 32, 38]. Ercegovac proposed an MSD-first function evaluation method, named the E-method, for polynomials and some rational functions [31]. The E-method solves diagonally dominant linear systems and generates one digit of each of the elements of their solution vectors in one iteration, starting from the MSD first [32]. Nicolas *et al.* overcame the limitation of diagonal dominance required for the E-method by changing the variables of a rational function, calling this family of rational functions E-fractions [13, 15]. For high-precision evaluation of polynomials, Ercegovac *et al.* implemented the E-method on FPGAs to demonstrate that FPGAs provide good flexibility of digit-serial and -parallel implementations [38]. Recently, Brisebarre *et al.* presented an automatic circuit generator for the E-method [14], incorporated into the FloPoCo framework [28].

There are also several other applications using online arithmetic, such as MSD-first approximate computing [106], arbitrary precision arithmetic [122], general profile search [88] and iterative numerical calculation [75].

2.4.3 Advantages and Disadvantages of Online Arithmetic

Given various applications using online arithmetic, I now summarise the benefits of online arithmetic.

- *Computing while Communicating.* MSD-first computation allows us to overlap multiple operations in datapaths to reduce the computation latency. This property can compensate for possible performance inefficiencies due to digit serial propagation.

- *Elimination of carry propagation.* Carry propagation is a performance bottleneck in conventional LSD-first arithmetic, and delays are proportional to precision. As presented previously in Section 2.3.2, online arithmetic uses redundant number systems, therefore long carry chains are eliminated for adders.

- *Iterating exactly.* With some architectural optimisations, online arithmetic can compute results to arbitrary precision [122]. A novel hardware architecture for arbitrary-precision iterative computation will be discussed in Chapter 3. As for conventional LSD-first arithmetic, precision must be fixed before starting to iterate, choosing the right precision *a priori* is non-trivial, particularly with respect to hardware implementation.

- *Runtime determination of precision.* Similarly to the previous point, online arithmetic allows computation to dynamically terminate when the required precision has been reached. Traditional LSD-first arithmetic operators cannot ever determine its precision at runtime, since their precision must be fixed at compile time.

- *Inference of stable most-significant digits.* In standard numerical iterative computing, approximate answers aim to approach the real solution with successive iterations. MSD-first computing allows us to infer stable digits in iterative calculations. The E-method [32] and the theoretical analysis in Chapter 5 demonstrate this property.

Despite the aforementioned benefits, online arithmetic still has some limitations that researchers have been putting effort into addressing for many years [97]. Decades ago, Muller discussed some characterizations of functions computatble in online arithmetic [90]. In general, the limitations of online arithmetic can be categorised into three aspects.

- *Overhead due to redundant number representations.* Just as in redundant number systems, a common concern in online arithmetic is that additional hardware resources are required. Area-efficient implementations of online arithmetic have been explored, such as that using modern FPGA's architectures [102].
- *Conversion to and from traditional representations.* For pure online arithmetic systems, conversion to and from traditional representations is not required. However, if online operators are used to generate intermediate results, this conversion is necessary. For example, Ercegovac and Lang proposed an *on-the-fly* technique that allows data to be efficiently converted between non-redundant and redundant forms [35].
- *Initial delay.* Online delay is an important characteristic of online operators. Classical digit-serial online operators produce output digits at the same rate as the consumption of operands, with a delay of a fixed number of digits: δ. Techniques have been proposed to reduce online delay, such as composite online algorithms [1,33,96], high-radix designs [61] and multi-operand operators [62, 114].

In the next chapter, we will see how ARCHITECT is designed by exploiting online arithmetic, and how it scales and outperforms traditional LSD-first and state-of-the-art online arithmetic equivalents.

Chapter 3

Arbitrary-precision Constant-hardware Iterative Compute

3.1 Overview

In this chapter, ARCHITECT-I, a novel hardware architecture for iterative computation is proposed. ARCHITECT-I is able to calculate an arbitrary number of iterations to arbitrary precision. A novel digit-vector storage scheme is described by considering two-dimensional indices: precision and iteration count. Given the computation dependencies of online arithmetic, a digit-scheduling pattern is then proposed within the precision-iteration space. How to construct ARCHITECT-I operations such as addition, multiplication and division is presented thereafter. The proposed hardware architecture is then evaluated on FPGAs for the computation of some iterative algorithm benchmarks.

The technical contributions of this chapter are as follows.

- The first fixed-compute-resource hardware architecture for iterative calculation capable of producing arbitrary-precision results after arbitrary numbers of iterations.
- An optimised mechanism for digit-vector storage based on a Cantor pairing function to facilitate simultaneously increasing precision and iteration count.
- A library of ARCHITECT-I operators able to compute from most-significant to least-significant digits, iteratively refining computations to any precision.
- Exploitation of digit-parallel online addition to decrease datapath latency.
- Exemplary hardware implementations of my proposals for the computation of both linear (Jacobi method) and nonlinear (Newton) iterations.
- Qualitative and quantitative performance and scalability comparisons against traditional and state-of-the-art online arithmetic FPGA implementations. Datapaths constructed following the proposed principles demonstrate efficiency over their traditional arithmetic equivalents where the latter's precisions are either under- or over-budgeted for the computation of a result to a particular accuracy. Versus arbitrary-precision iterative solvers without the optimisations detailed herein, I achieve up-to $1.1\times$ performance speedups for the evaluated benchmarks.

3.2 Notation

In this chapter, (K, P) is defined as the target result that computes at least K iterations and to at least P-digit precision.

A digit vector is denoted by a bold symbol $\boldsymbol{x}$, and will be stored in registers. For a P-digit number, $\boldsymbol{x} = \sum_{i=1}^{P} x_i r^{-i}$, where x_i is the i^{th} MSD.

P digits are stored by U-digit chunks, therefore $\lceil P/U \rceil$ is the number of such chunks that constitute a P-digit number. The chunk index is denoted by c and the digit index within a U-digit chunk is denoted by u.

The approximant of an iterative method at iteration $k \in \mathbb{N}_{>0}$ is denoted $\boldsymbol{x}^{(k)}$, with its j^{th} element (a scalar value) denoted $x_j^{(k)}$.

Where an approximant consists of signed-digit numbers, $x_{ji}^{(k)}$ is the i^{th} MSD of the j^{th} element of $\boldsymbol{x}^{(k)}$.

A matrix is represented by a bold capital symbol $\boldsymbol{X}$.

3.3 Proposed ARCHITECTure

Using classic online operators as a starting point, I now describe the construction of constant compute-resource hardware capable of performing iterative computation to increasing precision over time. This concept is called ARCHITECT-I.

3.3.1 Digit-vector Storage

Classic online operators make use of registers to store digit vectors. When implementing Algorithms 2 and 3 on page 28 in hardware, for example, P-digit registers are needed for $\boldsymbol{x}$ and $\boldsymbol{y}$. To compute to an arbitrary precision p instead, this is unsuitable; RAM must be used for digit-vector storage to avoid both under- and over-budgeting register resources. p is separated into two dimensions: one fixed, U, that determines the RAM width, and a second variable, $n = \lceil p/U \rceil$, representing the number of these 'chunks' that constitute each p-digit number. For digit index i, where $0 \leq i < p$, we define chunk index $c = \lfloor {}^{i}\!/\!_{U} \rfloor$ and chunk digit index $u = i \bmod U$ such that $i = Uc + u$. When performing iterative calculations, digit vectors exist for each step, thus their indexing requires three variables: $c \in [0, n)$, $u \in [0, U)$ and approximant index k. The relationships between c, n, u, U and overall digit index i are shown visually for a single p-digit number in Figure 3.1.

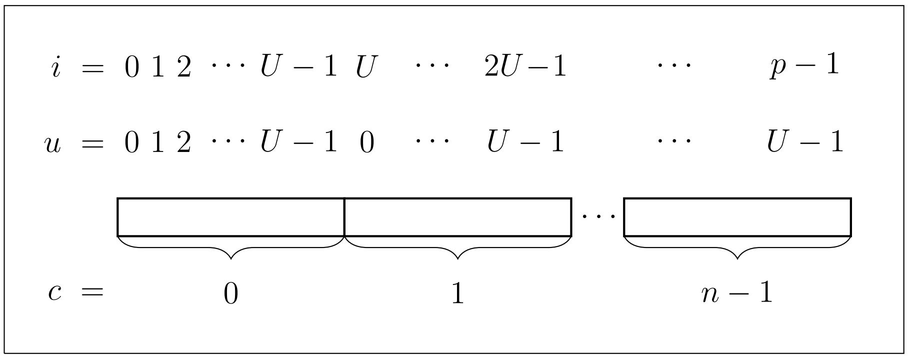

$i = 0\ 1\ 2\ \cdots\ U-1\ U\ \cdots\ 2U-1\ \cdots\ p-1$
$u = 0\ 1\ 2\ \cdots\ U-1\ 0\ \cdots\ U-1\ \cdots\ U-1$
$c =$ 0 1 $n-1$

Fig. 3.1. Indexing of digits and chunks within a p-digit number. i indexes all digits, while those of each of its n chunks, indexed c, are indexed with u.

Since ARCHITECT-I requires k and i to both vary non-monotonically as time progresses, as was shown in Fig. 1.1b on page 4, it is necessary to uniquely encode a one-to-one mapping from two-dimensional approximant and chunk index pair (k, c) into one-dimensional time. ARCHITECT-I uses a Cantor pairing function (CPF) [22], a bijection from $\mathbb{N}^2$ onto $\mathbb{N}$, for this purpose, defined to be

$$\mathrm{cpf}(k, c) = \frac{(k + c)\,(k + c + 1)}{2} + c. \tag{3.1}$$

The function's bijectivity is crucial for ARCHITECT-I. Unlike classic row- or column-major indexing, the injectivity of the CPF allows both dimensions to grow

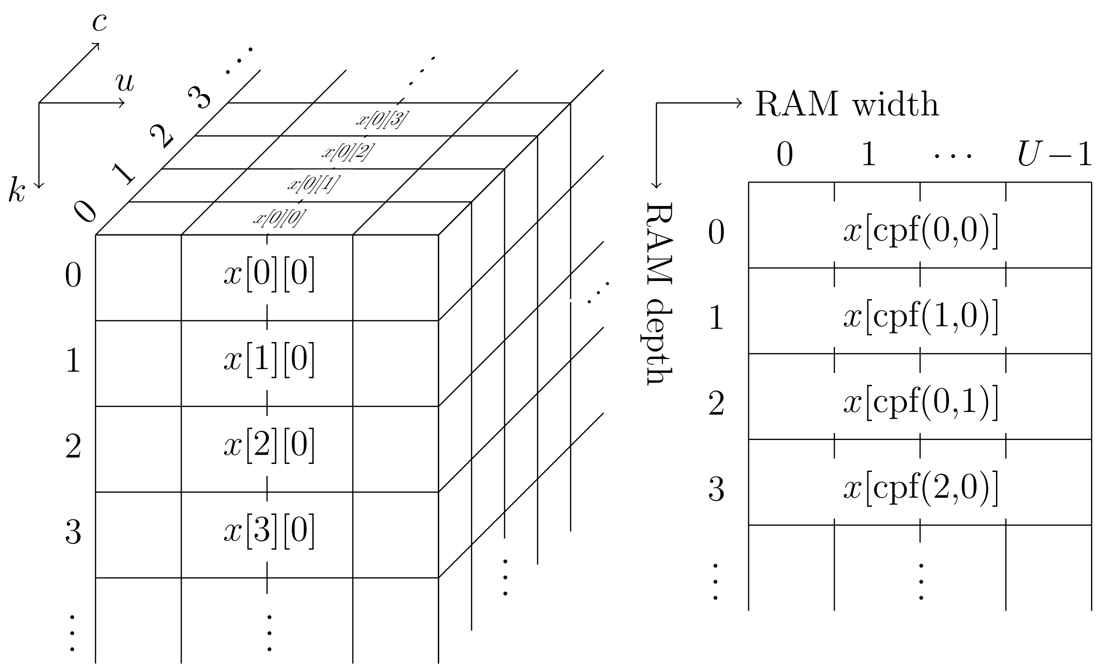


Fig. 3.2. Operation of our Cantor pairing function, showing the transformation of a three-dimensional array growing with both approximant and chunk indices k and c to a structure growing only in a single dimension.

without bound while providing a unique result for every (k, c). The operation of our CPF is demonstrated visually in Fig. 3.2; what is conceptually a three-dimensional array indexed as (k, c, u) becomes a two-dimensional array indexed by $(\text{cpf}(k, c), u)$ instead, thereby suiting the 'flat' nature of RAM. The function's surjectivity ensures that every $\text{cpf}(k, c)$ is produced by some (k, c), thus enabling the most efficient use of the available memory.

3.3.2 Arbitrary-precision Operators

Multiplication

We are now in a position to rewrite Algorithm 2 on page 28 such that it can compute results to arbitrary precision. These transformed steps are shown in Algorithm 4. Most importantly, a new loop has been introduced; this iterates over the n pairs

Algorithm 4 Radix-2 ARCHITECT-I multiplication.

```
Inputs: serially presented multiplicand $x$, multiplier $y$; approximant index $k$, precision $p$
 1: $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w} \leftarrow \boldsymbol{0}$
 2: for $j = 0$ to $p + 2$ do
 3:     $\boldsymbol{y}[\mathrm{cpf}(k, \lfloor j/U \rfloor)][j \bmod U] \leftarrow y_j$
 4:     for $c = \lfloor j/U \rfloor$ to $0$ do
 5:         $\boldsymbol{v}[\mathrm{cpf}(k,c)] \leftarrow 2\boldsymbol{w}[\mathrm{cpf}(k,c)] + 2^{-3}(\boldsymbol{x}[\mathrm{cpf}(k,c)]y_j + \boldsymbol{y}[\mathrm{cpf}(k,c)]x_j)$
 6:         if $c > 0$ then
 7:             $\boldsymbol{w}[\mathrm{cpf}(k,c)] \leftarrow \boldsymbol{v}[\mathrm{cpf}(k,c)]$
 8:         end if
 9:     end for
10:     $z_{j-3} \leftarrow \mathrm{sel}_\times(\boldsymbol{v}[\mathrm{cpf}(k,0)])$
11:     $\boldsymbol{w}[\mathrm{cpf}(k,0)] \leftarrow \boldsymbol{v}[\mathrm{cpf}(k,0)] - z_{j-3}$
12:     $\boldsymbol{x}[\mathrm{cpf}(k, \lfloor j/U \rfloor)][j \bmod U] \leftarrow x_j$
13: end for
Output: serially generated product $z$
```

of p-digit numbers' chunks (Fig. 3.1), most-significant first, to facilitate arbitrary-precision multiplication with a U-digit online adder. Digit vectors $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{v}$ and $\boldsymbol{w}$ are now indexed in two dimensions, corresponding to standard RAM addressing denoted as [word][digit]. Where a digit index is not given, all U digits of that word are accessed simultaneously.

Division

The equivalently transformed version of Algorithm 3 on page 28 is shown in Algorithm 5. Mirroring the increased complexity of classic online division over multiplication, here, two accumulation loops are needed: one for the calculation of $\boldsymbol{v}$, as for multiplication, and a second for $\boldsymbol{w}$. Consequently, $n-1$ more cycles are required for the computation of an output digit in ARCHITECT-I division than multiplication.

Particular care is required for digit alignment in online division since input operands need to be bounded such that the output range is $(-1, 1)$ [37]. The normalisation

Algorithm 5 Radix-2 ARCHITECT-I division.

```
Inputs: serially presented dividend $x$, divisor $y$; approximant index $k$, precision $p$
 1: $\boldsymbol{y}, \boldsymbol{w}, \boldsymbol{z} \leftarrow 0$
 2: for $j = 0$ to $p + 3$ do
 3:     $\boldsymbol{y}[\mathrm{cpf}(k, \lfloor j/U \rfloor)][j \bmod U] \leftarrow y_j$
 4:     for $c = \lfloor j/U \rfloor$ to 0 do
 5:         $\boldsymbol{v}[\mathrm{cpf}(k, c)] \leftarrow 2\boldsymbol{w}[\mathrm{cpf}(k, c)] + 2^{-4}(x_j - \boldsymbol{z}[\mathrm{cpf}(k, c)]y_j)$
 6:     end for
 7:     $z_{j-4} \leftarrow \mathrm{sel}_{\div}(\boldsymbol{v}[\mathrm{cpf}(k, 0)])$
 8:     for $c = \lfloor j/U \rfloor$ to 0 do
 9:         $\boldsymbol{w}[\mathrm{cpf}(k, c)] \leftarrow \boldsymbol{v}[\mathrm{cpf}(k, c)] - z_{j-4}\boldsymbol{y}[\mathrm{cpf}(k, c)]$
10:     end for
11:     $\boldsymbol{z}[\mathrm{cpf}(k, \lfloor j/U \rfloor)][j \bmod U] \leftarrow z_{j-4}$
12: end for
Output: serially generated quotient $z$
```

of quotients following online division ordinarily necessitates variable $\delta_{\div}$ [112]. To avoid this, a lower bound on the magnitude of the divisor can maintain a fixed online delay [95]. This has been analysed by Trivedi and Ercegovac who bounded the divisor magnitude within $[1/r, 1)$ for online-division algorithms [111]. For experimentation, digit alignment can be guaranteed across iterations through the appropriate selection of initial inputs.

3.3.3 Digit-scheduling Pattern

Given a generic online delay δ made up of latencies from a pipeline (or replicated pipelines operating in parallel) of one or more operators implementing the body of an iterative algorithm, restrictions are imposed on the order in which digits can be calculated. δ impacts us in two ways:

- Calculation of the first output digit requires the prior input of the first $\delta + 1$ input digits. Thereafter, each subsequent output digit requires one additional input digit in order to be computed.

- The i^{th} output digit is generated δ cycles after the i^{th} input digit is presented.

In general, digits of the same approximant can be calculated indefinitely, while those across iterations must be sequenced such that they obey these δ-imposed limitations. When scheduling digit $z_i^{(k)}$'s generation, we must ensure that

$$t\left(z_{i+1}^{(k)}\right) > t\left(z_i^{(k)}\right), \quad t\left(z_i^{(k+1)}\right) > t\left(z_{i+\delta}^{(k)}\right)$$

for all approximant indices $k \geq 1$ and digit indices $i \geq 0$, where t is the time at which a generation event occurs.

While there is the freedom to trade off between iteration count and precision within the bounds of these dependencies, ARCHITECT-I always assumes a mapping from the current to the next digit of the form depicted in Fig. 3.3. The groups of digits shown, each δ in size, are processed 'downwards' and 'leftwards,' with slope dependent on δ and control snapping back to the first approximant once digit position $i = 0$ has been reached. Fixing the granularity of digit generation to δ allows for control path simplification—as will be elaborated upon in Section 3.3.4—and limits transitions between approximants. The latter is beneficial since, as will be explained in Section 3.3.6, switching between approximants leads to the incursion of performance penalties under some circumstances.

3.3.4 Control Logic

Given a particular (k, i), ARCHITECT-I can compute the subsequent index pair, (k', i'), needed to realise a scheduling pattern such as the one shown in Figs 3.3 with the finite-state machine (FSM) depicted in Fig. 3.4.

The states' functionality is as follows.

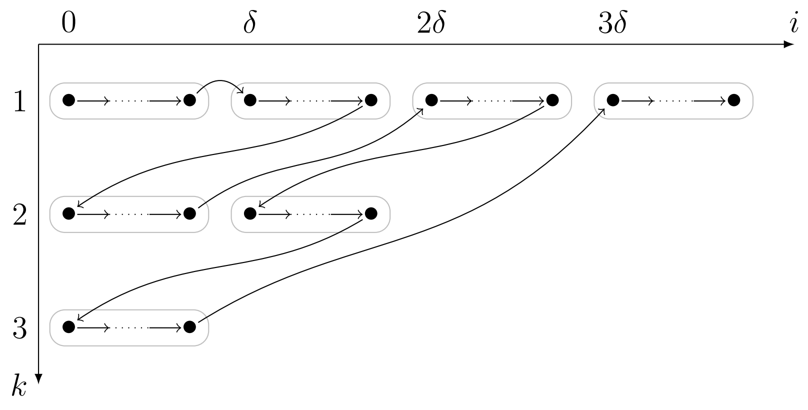


Fig. 3.3. Proposed digit generation pattern for generic iterative computation using online operators.

- *Digit generation*: Manages the propagation and storage of δ-digit groups across iterations. When remaining within this state, only digit index i must be evaluated to determine changes needed to k and i.

- *Accumulation*: When $i < U$, ARCHITECT-I is able to perform i-digit additions in single cycles. However, when $i \geq U$, $\lfloor i/U \rfloor$ additional cycles must be consumed to compute all U-digit chunks. Since ARCHITECT's multiplication and division operators have dissimilar accumulation functionality, as was explained in Section 3.3.2, the number of clock cycles consumed by each is different. The throughput of the datapath as a whole is determined by the slowest operator. In Fig. 3.4, counter γ sequences the return to the digit generation state. Since i is variable, this loop cannot be unrolled.

3.3.5 Accuracy Bounds

Let us assume the existence of a target result defined by its iteration index and precision (K, P). To reach it, we are required to compute for at least K iterations and to at least P-digit precision. Note that ARCHITECT-I does not necessitate its users to specify K or P up-front, while other approaches require either one or

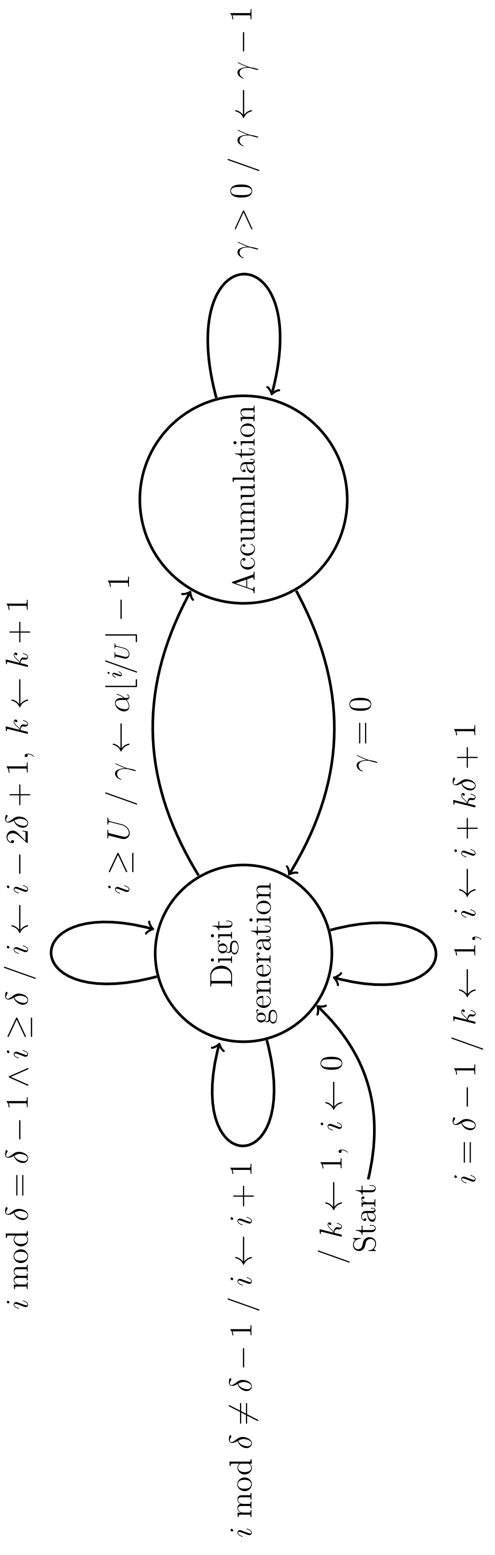

$\gamma > 0 \;/\; \gamma \leftarrow \gamma - 1$
Accumulation
$i \geq U \;/\; \gamma \leftarrow \alpha\lfloor i/\upsilon \rfloor - 1$
$\gamma = 0$
Digit generation
$i \bmod \delta = \delta - 1 \land i \geq \delta \;/\; i \leftarrow i - 2\delta + 1,\; k \leftarrow k + 1$
$i = \delta - 1 \;/\; k \leftarrow 1,\; i \leftarrow i + k\delta + 1$
$i \bmod \delta \neq \delta - 1 \;/\; i \leftarrow i + 1$
Start
$/\; k \leftarrow 1,\; i \leftarrow 0$

Fig. 3.4. FSM for digit computation scheduling. Transition edges are labelled with conditions and actions separated by slashes (/). If the datapath consists only of adders, the accumulation state is never entered. Otherwise, $\alpha = 2$ if the datapath contains one or more dividers, and is 1 in all other cases. Termination occurs either on demand or following memory exhaustion.

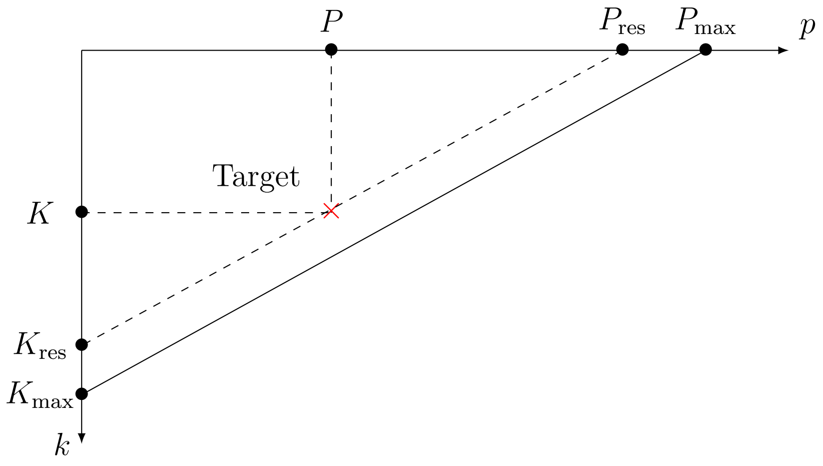


Fig. 3.5. How the final precision and iteration count $(K_{\text{res}}, P_{\text{res}})$ are constrained by the desired result (K, P) and the available memory $(K_{\text{max}}, P_{\text{max}})$.

both of these—usually P—to be determined before beginning to iterate.

As shown in Fig. 3.5, the number of iterations resulting from computation to target (K, P) is defined as K_{res} and the final precision of the first approximant—always the most precise—as P_{res}. K_{res} is bounded to no more than K_{max}, while P_{res} is similarly bounded by P_{max}, both of which are determined by the size of the available memory. The latter therefore determines the maximum approximant index and precision—and consequently accuracy—that can be reached through the use of this approach. Thus, if higher accuracy is required, more memory must be instantiated.

Upon completion, the precision of approximant k will be

$$p^{(k)} = \begin{cases} \delta\left(\left\lceil \frac{P}{\delta} \right\rceil + K - k\right) & \text{if } k < K \\ P & \text{if } k = K \\ \delta(K_{\text{res}} - k) & \text{otherwise,} \end{cases}$$

where K_{res} can be geometrically deduced, based on computation patterns such as that shown in Fig. 3.3, to be

$$K_{\text{res}} = \begin{cases} \lceil \frac{P}{\delta} \rceil + K - 1 & \text{if } P > \delta \\ K & \text{otherwise} \end{cases}$$

and $P_{\text{res}} = p^{(1)}$.

For each arbitrary-precision digit vector to be stored, $K_{\max}$ and $P_{\max}$ are fixed by RAM depth D (in U-digit words). Analysis of the pairing function in (3.1) allows us to derive

$$P_{\max} = U\Big(1 + \Big\lfloor {}^{3}\!/_{2}\Big(\sqrt{1 + {}^{8}\!/_{9}D} - 1\Big)\Big\rfloor\Big),$$

$$K_{\max} = \begin{cases} \frac{P_{\max}}{U} + 1 & \text{if } D \geq \left(\frac{P_{\max}}{U} + 1\right)\frac{P_{\max}}{2U} \\ \frac{P_{\max}}{U} & \text{otherwise.} \end{cases}$$

3.3.6 Compute Time

Given a particular target (K, P), and hence a certain K_{res} and P_{res}, we can calculate the number of clock cycles required to compute the desired result. This total time T (clock cycles) can be broken down into the following three components such that $T = T_{\text{init}} + T_{\text{gen}} + T_{\text{sa}}$.

- *Initial online delay*: The computation must wait δ clock cycles before each approximant's result begins to appear, thus the delay across all iterations is simply

$$T_{\text{init}} = \delta K_{\text{res}}.$$

- *Digit generation*: Across all iterations performed, the total time for digit generation is either

$$T_{\text{gen}} = \sum_{k=0}^{K_{\text{res}}-1} p^{(k)}\big(2n^{(k)} - 1\big) - Un^{(k)}\big(n^{(k)} - 1\big) - \delta,$$

if the datapath contains dividers, or

$$T_{\text{gen}} = \sum_{k=0}^{K_{\text{res}}-1} n^{(k)}\left(p^{(k)} - \frac{U\big(n^{(k)} - 1\big)}{2}\right) - \delta$$

if it contains multipliers. $n^{(k)} = \lceil p^{(k)}/U \rceil$ and represents the number of chunks within the given approximant upon termination of the algorithm. In the case that the datapath contains only adders,

$$T_{\text{gen}} = \sum_{k=0}^{K_{\text{res}}-1} p^{(k)} - \delta.$$

$p^{(0)}$ and $n^{(0)}$ are the numbers of digits and chunks, respectively, that must be read from the initial guess.

- *Digit-serial addition*: Recall that a serial online adder has $\delta_+ = 2$. When switching between iterations, adders, if present, require two cycles to recalculate the preceding approximant's residuals in order to produce a new digit [37]. This ensures that the calculated digit aligns with its truncated digit vectors. For this,

$$T_{\text{sa}} = \beta\big(K_{\text{res}}^2 - K_{\text{res}} + 2K - 2\big), \tag{3.2}$$

where β is the number of serial adders present along the highest-online delay path within the circuit.

3.3.7 Digit-parallel Addition Optimisation

It is possible to eliminate the final T component in Section 3.3.6, resulting in $T_{\text{sa}} = 0$, by using three-digit parallel online adders in place of serial ones. I store consecutive digit-vector words in alternating memory banks for speed. By ensuring that RAM width $U > 1$, *i.e.* that each word contains at least two digits, we can always read the three contiguous digits required by these adders in a single cycle. No additional memory is needed for this optimisation.

3.4 Benchmarks

In order to evaluate ARCHITECT-I, two widely used iterative algorithms were implemented—the Jacobi method (to solve systems of linear equations) and Newton's method (for the solution of nonlinear equations)—in hardware following the aforementioned principles. Jacobi and Newton were chosen to exemplify a large class of iterative methods with linear and quadratic convergence properties, respectively.

3.4.1 Jacobi Method

The Jacobi method seeks to solve the system of N linear equations $\boldsymbol{Ax} = \boldsymbol{b}$. If $\boldsymbol{A}$ is decomposed into diagonal and remainder components such that $\boldsymbol{A} = \boldsymbol{D} + \boldsymbol{R}$, $\boldsymbol{x}$ can be computed through the repeated evaluation of

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{D}^{-1}\left(\boldsymbol{b} - \boldsymbol{R}\boldsymbol{x}^{(k)}\right),$$

or, expressed in element-wise fashion,

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i \in [0,N)} a_{ij} x_j^{(k)} \right) \quad \forall i \in [0,N)\,,$$

where k is the approximant index. Since $\boldsymbol{D}$'s only non-zero elements lie along its diagonal, $\boldsymbol{D}^{-1}$ is trivial to calculate. Note that $\boldsymbol{x}^{(k+1)}$ relies only upon the previously calculated value $\boldsymbol{x}^{(k)}$; the calculation can therefore be parallelised by computing each $x_i^{(k+1)}$ independently across the different values of i. A convergence criterion, $\|\boldsymbol{A}\boldsymbol{x}^{(k)} - \boldsymbol{b}\| < \eta$, can be used in order to determine if the solution has been found to great enough accuracy.

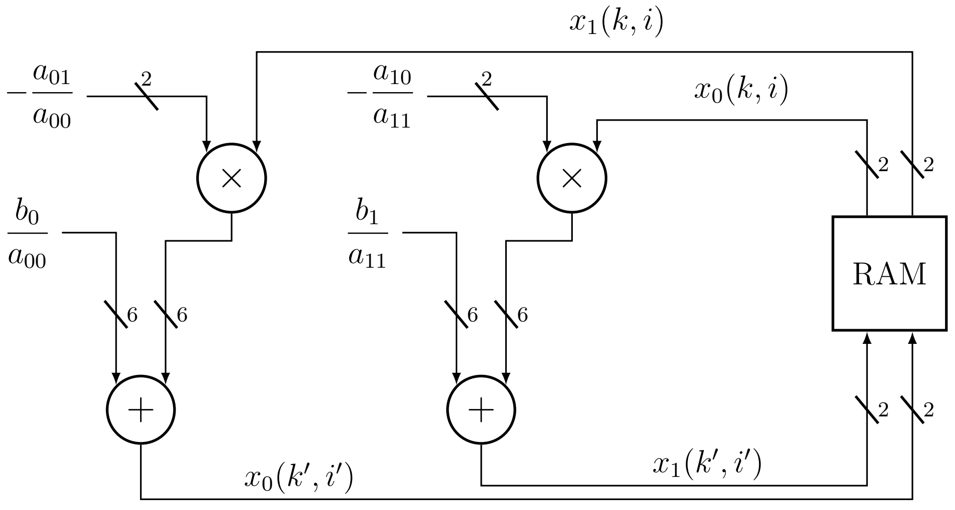
Such a system is guaranteed to be solvable by the Jacobi method when $\boldsymbol{A}$ is strictly diagonally dominant, *i.e.* if the condition $|a_{ii}| > \sum_{j \neq i} |a_{ij}|$ holds for all i. Although strict diagonal dominance is not a necessity in every case, we assume this condition to always be satisfied for simplicity.

A metric used to quantify the sensitivity of a particular linear system to error is the *condition number* of $\boldsymbol{A}$ [87],

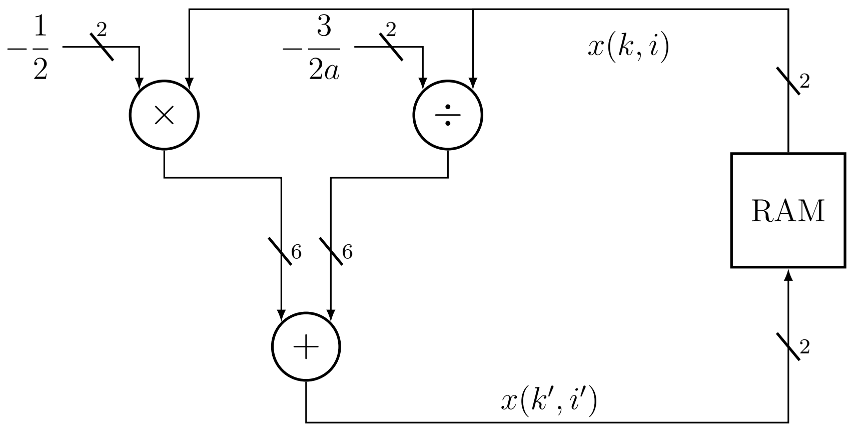
$$\kappa(\boldsymbol{A}) = \|\boldsymbol{A}\| \left\|\boldsymbol{A}^{-1}\right\|.$$

Perturbations in $\boldsymbol{x}^{(k)}$, caused by rounding, lead to errors in $\boldsymbol{x}^{(k+1)}$ whose magnitude is dependent, in part, on $\kappa(\boldsymbol{A})$; a high condition number indicates that $\boldsymbol{A}$ is sensitive to error and therefore ill-conditioned [25]. It is expected to need at least ζ additional digits of precision in order to compute a system with $\kappa(\boldsymbol{A}) = 2^\zeta$ than would be required if $\kappa(\boldsymbol{A})$ were 1 [23].

Without loss of generality, Jacobi solvers with matrix size $N = 2$ are implemented as a toy example, depicted in Fig. 3.6a, and features ARCHITECT-I numerical



(a) Jacobi method ($\delta = 3$).



(b) Newton's method ($\delta = 4$).

Fig. 3.6. ARCHITECT-I benchmark datapaths. Adders, multipliers and dividers are arbitrary-precision radix-2 signed-digit online operators. Use of three-digit adders reduces online delay by 2 over their serial equivalents.

operators as described in Section 3.3.2. Jacobi solvers with $N > 2$ could have been built with additional multipliers and adders, but this is not the emphasis—demonstrating arbitrary-accuracy iterative calculation—of this work. Note that runtime division is unnecessary since $\boldsymbol{A}$ and $\boldsymbol{b}$ are constants.

3.4.2 Newton's Method

Newton's method is a root-finding algorithm, commonly employed to approximate the zeroes of a real-valued function f. The iterative process is

$$x^{(k+1)} = x^{(k)} - \frac{f\left(x^{(k)}\right)}{f'\left(x^{(k)}\right)},$$

where f' is the first derivative of f. Assuming that $f(x) = 0$ is soluble and $f'(x^*) \neq 0$, convergence is quadratic if the initial guess $x^{(0)}$ is sufficiently close to a solution x^* [60].

The implemented datapath is shown in Fig. 3.6b, again with ARCHITECT-I operators, as a second case study. This can solve equations of the form $f(x) = ax^2 - 3 = 0$:

$$x^{(k+1)} = \frac{x^{(k)}}{2} + \frac{3}{2ax^{(k)}}. \tag{3.3}$$

Since the solution of $f(x) = 0$ is irrational for some choices of a (*e.g.* 1), we consider this to be a particularly good showcase of ARCHITECT-I's arbitrary-precision capabilities.

3.5 Evaluation

Theoretical analysis was conducted and experiments were performed to investigate how ARCHITECT-I scales and performs versus competing arithmetic implementations, both traditional (LSD-first) and online, using the Jacobi and Newton's methods as benchmarks. Performance is evaluated in terms of latency, which, for all implementations considered in this thesis, is the multiplicative inverse of throughput.

The closest study to this work is that presented by Zhao *et al.* [122], which ARCHI-

TECT-I compares against directly. For comparison against traditional arithmetic, parallel-in serial-out (PISO) operators were chosen to be implemented since ARCHITECT-I operates in a similar digit-serial fashion. PISO sits at the midpoint between fully serial (SISO) and parallel (PIPO) in terms of area and performance [57]. With increase in precision P—which, for traditional arithmetic, can solve problems requiring precision *up to* P—PISO suffers less from area growth and operating frequency $f_{\max}$ degradation than PIPO [85] while also being dramatically faster than SISO [68]. While we focus exclusively on hardware implementations, the limitations revealed for PISO apply equally to software libraries since precision must be chosen prior to iterative algorithmic commencement.

3.5.1 Complexity Analysis

Table 3.1 presents the results of asymptotic complexity analysis—in terms of circuit size, memory requirements and latency—performed for ARCHITECT-I and its competitors. For PISO, we assume the repeated evaluation of an iterative expression using datapaths composed of standard numeric operators. For each arithmetic, we further assume latency-optimal datapath implementations featuring minimal-depth adder (for Jacobi) and multiplier (Newton) trees. Complexities for Zhao *et al.*'s implementation were derived from analytical expressions provided by the authors [122].

Since we have chosen to analyse latency-optimised datapaths, area scales with the required number of multipliers (Newton) and adders (Jacobi), which themselves grow quadratically with N. For PISO, area also scales linearly with the width of its input operands, controlled by P, while the size of Zhao *et al.*'s implementations instead scales linearly with the number of iterations to be performed, K. The area of an ARCHITECT-I implementation, however, scales with neither K nor P, since

Table 3.1
Complexities of iterative solver implementations.

	Area	Memory	Solve time
PISO	$\mathcal{O}(N^2P)$	$\mathcal{O}(NP)$, $\mathcal{O}(P)^1$	$\mathcal{O}(\log(N)KP)$
Zhao *et al.* [122]	$\mathcal{O}(N^2K)$	$\mathcal{O}(N^2KP)$	$\mathcal{O}(P(\log(N)K+P))$
ARCHITECT-I	$\mathcal{O}(N^2)$	$\mathcal{O}\big(N^2(K+P)^2\big)$	$\mathcal{O}\Big(\frac{(\log(N)K+P)^3}{\log(N)}\Big)$

^{1} N-dimensional Jacobi method, Nth-order Newton's method.

the same arithmetic operators compute every approximant, to any precision, for the chosen iterative method.

As with area, a PISO implementation's memory footprint scales linearly with P; for the Jacobi method, scaling is also linear in N due to the size of the computed vector. Both Zhao *et al.*'s implementations and ARCHITECT-I require residue storage within their multipliers and dividers; memory occupancy therefore scales with area for the arbitrary-precision architectures. For the former, the use of memory also scales with P as residues in online multiplication and division are stored to the same precision as its input data. Since ARCHITECT-I effectively collapses approximant and precision indices into a single dimension via its CPF, the memory requirements for each operator are determined by the maximum value of (3.1) during computation to the target (K, P), thus they scale quadratically with $K + P$.

PISO's latency grows linearly with K and P, but logarithmically with N due to our aforementioned choice of adder (and multiplier) structures. Zhao *et al.*'s speed is bottlenecked by the growth of precision—quadratically—as well as the frequency of pipeline flushes, which grows as $\mathcal{O}(\log(N)KP)$ [122]. For ARCHITECT-I, given that each datapath's highest cumulative online delay δ is logarithmically related to N, its latency complexity can be determined by solving for T_{gen} in Section 3.3.6. Note that T_{gen} dominates T_{init} in all cases and $T_{\text{sa}} = 0$ due to the use of digit-parallel adders.

At first glance, it appears that ARCHITECT-I behaves more poorly than its competitors in terms of memory use and solve time when scaled. These complexities, however, do not take fundamental limitations of the alternatives into account. In particular, exact computation to a particular (K, P) is rarely possible with P-digit LSD-first arithmetic due to rounding errors introduced in earlier approximants; only MSD-first architectures are capable of producing exact results for every approximant. Additionally, they do not account for ARCHITECT-I's unique ability to compute results to *any* required accuracy, effectively allowing the necessary (K, P) to be determined, on a problem-by-problem basis, at runtime. In contrast, a PISO implementation's precision is always bounded, while the same is true of iteration count for Zhao *et al.*'s proposal. In the remainder of this section, we explore empirically the implications of these issues.

3.5.2 Experimental Particulars

Experiments were performed to investigate how ARCHITECT-I scales and performs versus competing arithmetic implementations, both traditional (LSD-first) and online, using the Jacobi and Newton's methods as benchmarks.

A Xilinx Virtex UltraScale FPGA (XCVU190-FLGB2104-3-E) was targetted for all experiments detailed henceforward, with implementation performed using Vivado 16.4. The correctness of results obtained in hardware was verified via comparison against those produced by golden models executed in software. Fig. 3.7 captures the experimental process.

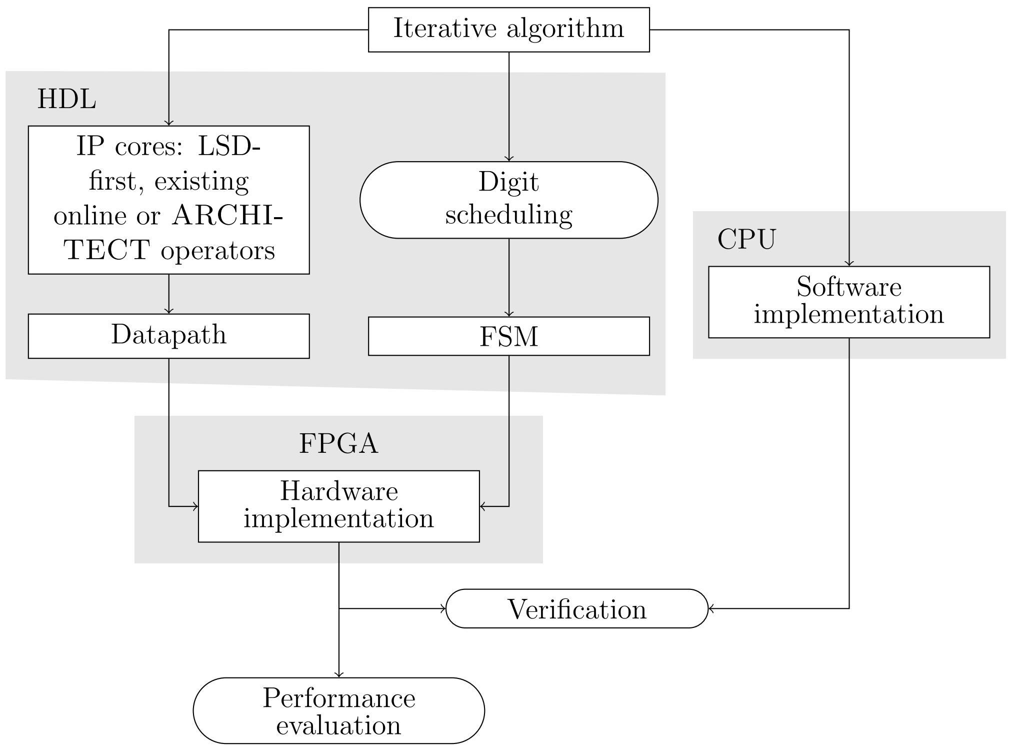


Fig. 3.7. Experimental setup for the evaluation of ARCHITECT-I.

3.5.3 Empirical Performance Comparison

To evaluate performance for the Jacobi method, systems were considered in which

$$\boldsymbol{A}_m = \begin{pmatrix} 1 & 1-2^{-m} \\ 1-2^{-m} & 1 \end{pmatrix}, \quad \boldsymbol{b} = \begin{pmatrix} b_0 \\ b_1 \end{pmatrix}, \quad \boldsymbol{x}^{(0)} = \mathbf{0},$$

with b_0 and b_1 randomly selected from a uniform distribution in the range $[0, 1)$. As m increases, the condition number $\kappa(\boldsymbol{A}_m)$ also increases, indicating that a higher precision P will be required to generate a result of great enough accuracy. I set accuracy bound $\eta = 2^{-6}$ and experimentally determined that the most ill-conditioned matrix requiring $P = 32$, a commonly encountered traditional arithmetic data width, to solve the associated system was that with $m = 25$, therefore I limited the experiments to $m \in [0, 25]$. I postulate that ARCHITECT-I should 'win', *i.e.* compute the required result in less time, versus PISO either when the latter's precision P is high and $\boldsymbol{A}_m$ is well conditioned or when P is too low for an ill-conditioned $\boldsymbol{A}_m$ to allow convergence at all. For ARCHITECT-I, a RAM size $(U, D) = (8, 2^{10})$ is used. Latencies were calculated using frequencies taken from Section 3.5.5.

Fig. 3.8a captures the latency ratio between ARCHITECT-I and PISO with a fixed precision of 32 bits (LSD-32) necessary to compute results for matrices with low m. Here, PISO can be said to have over-budgeted precision; results take longer to compute than had a smaller P been chosen in advance. For the most well conditioned matrices ($m \lessapprox 0.022$), ARCHITECT-I takes less time to reach the target (K, P). For larger m, however, the opposite is true: the lower-indexed iterations' results are computed to greater accuracy than those with PISO, taking more time. Had a lower choice of P been made for PISO, ARCHITECT-I would have been at a disadvantage for the more well conditioned matrices, but it would also have been able to compute the results of systems featuring ill-conditioned matrices that PISO could not. As shown in Fig. 3.8c, with $P = 8$ (LSD-8), ARCHITECT-I can solve systems with $m > 2$, where PISO's precision is under-budgeted; here, even if PISO ran indefinitely it would never be able to converge to an accurate-enough solution. By ways of conclusion, therefore, ARCHITECT-I requires less time to generate results than PISO either when P is small and convergence is fast, or when P is too large for PISO to ever converge.

Newton's method in (3.3) was implemented with $a \in [1, 2^{31}]$. As a increases, ${}^{3}\!/\!_{2a}$ decreases, thus greater precision will be required for its representation. Calculations are performed under termination condition $\left|f\left(x^{(k+1)}\right)\right| < \eta$, with η again set to 2^{-6}. $a \in [1, 2^{31}]$ was chosen since, to solve $f(x)$ with $a = 2^{31}$, the worst-case precision requirement was again $P = 32$.

Figs 3.8b and 3.8d show the performance of ARCHITECT-I-based Newton's method

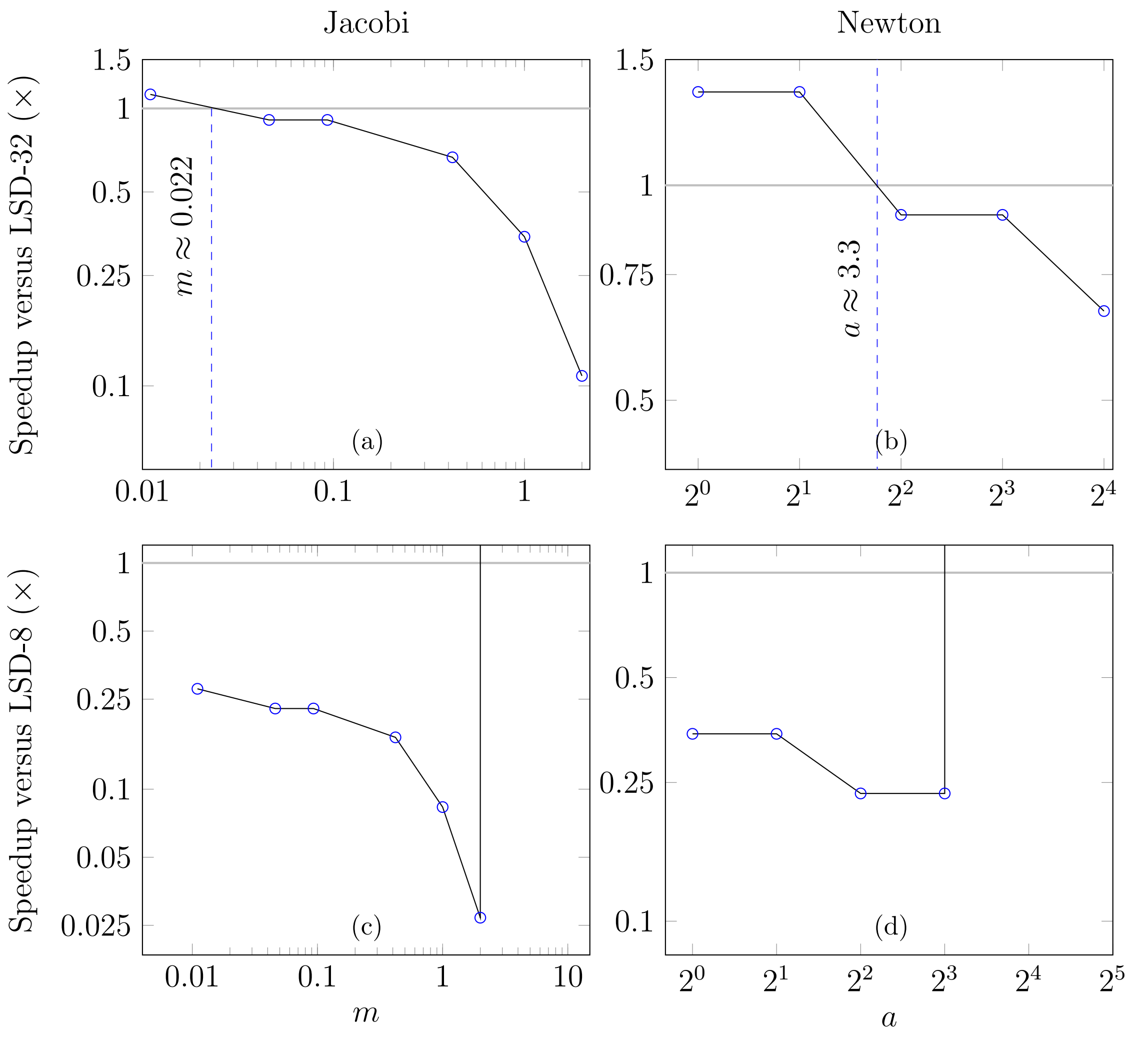


Fig. 3.8. Performance comparisons of our proposal against conventional LSD-first arithmetic for the Jacobi and Newton's methods. (a) and (b) show how the conditioning of input matrix $\boldsymbol{A}_m$ (Jacobi) and input value a (Newton) affect the solve time of our proposal compared to LSD-32. ARCHITECT-I computes more quickly than LSD-32 when $m \lessapprox 0.022$ for Jacobi and $a \lessapprox 3.3$ for Newton. (c) and (d) show that, even though our proposal leads to a slowdown compared to LSD-8, there are nevertheless points—at $m > 2$ (Jacobi) and $a > 8$ (Newton)—whence LSD-8 does not converge at all, hence our speedup is effectively infinite.

benchmark versus 32-bit and 8-bit PISO in the same form as Figs 3.8a and 3.8c. The results achieved for Newton's method are broadly similar to those for Jacobi. ARCHITECT-I requires $a \lessapprox 3.3$ to beat LSD-32 in terms of compute time, while only our proposed iterative solver can solve systems with $a > 8$ when PISO has $P = 8$. Identical conclusions regarding under- and over-budgeted precisions can therefore be drawn for Newton's method.

3.5.4 Scalability Analysis

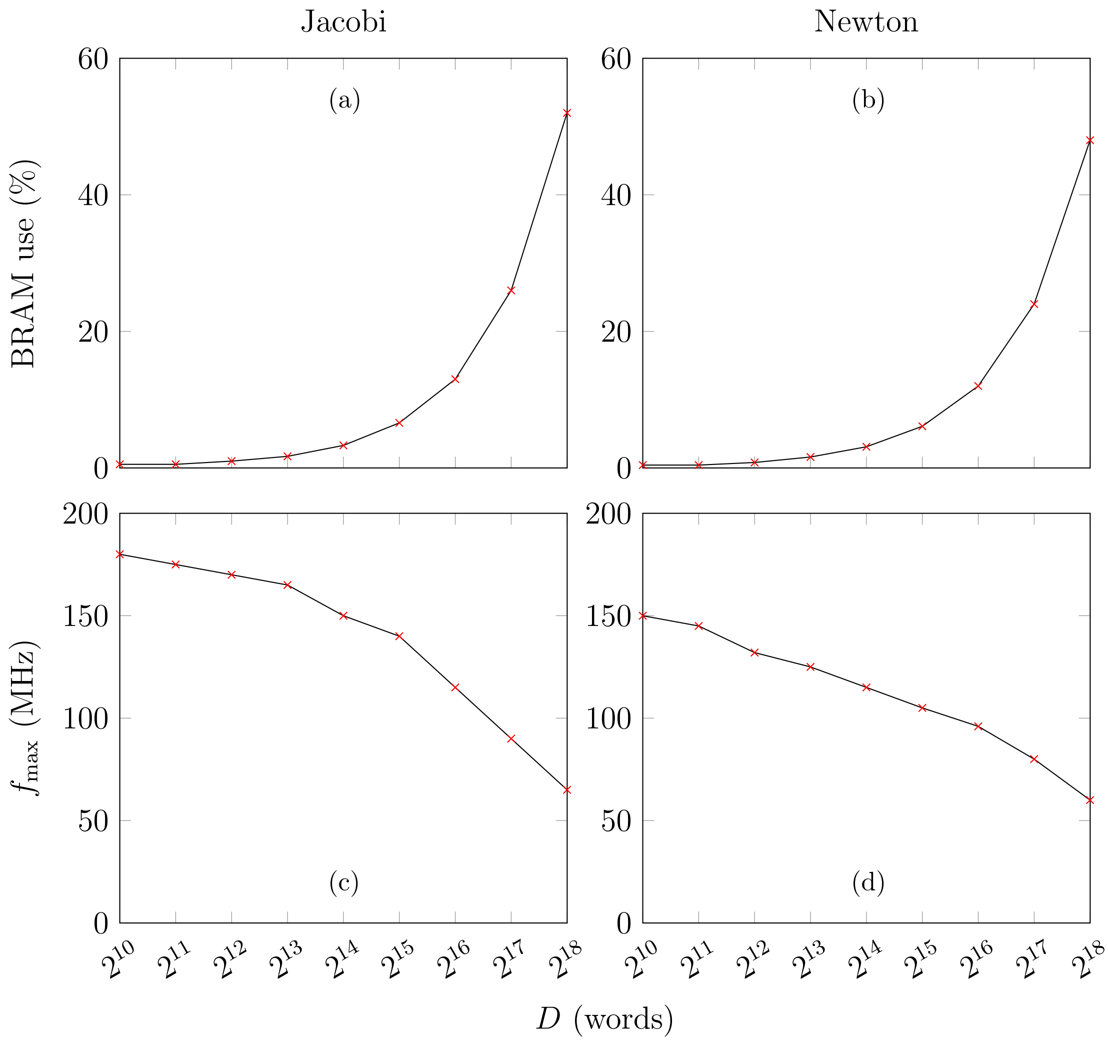


Fig. 3.9. Resource use and performance of ARCHITECT-I Jacobi and Newton benchmarks versus RAM depth D. Area is reported in terms of BRAMs only; LUT and FF use were below 1% for all design points.

Implementation results are presented in Fig. 3.9 for both Jacobi and Newton's method benchmarks, including area and maximum operating frequency $f_{\max}$. Each of the four plots features D, the RAM depth used for storage of each digit vector,

TABLE 3.2
AREA-SPEED TRADEOFF VIA SELECTION OF RAM WIDTH U.

	U	LUTs (%)	FFs (%)	BRAMs (%)	$f_{\max}$ (MHz)	Addition latency (cycles)
Jacobi	8	0.12	0.030	0.53	180	$\lceil p^{(k)}/8 \rceil$
	64	0.49	0.083	1.9	115	$\lceil p^{(k)}/64 \rceil$
Newton	8	0.12	0.024	0.42	155	$2\lceil p^{(k)}/8 \rceil - 1$
	64	0.34	0.035	1.5	110	$2\lceil p^{(k)}/64 \rceil - 1$

on the x-axis, and RAM width U was 8 in all cases. Lookup table (LUT) and flip-flop (FF) use is not shown since the numbers are insignificant compared to those of on-chip block RAM (BRAM)—from 0.12% to 0.29% for LUTs and 0.030% to 0.064% for FFs for the smallest ($D = 2^{10}$) and largest ($D = 2^{18}$) Jacobi designs implemented, and from 0.12% to 0.30% (LUTs) and 0.024% to 0.055% (FFs) for the Newton datapath. Memory use grows with D, as expected; the higher K_{res} and P_{res} one wishes to be able to reach, the more RAM must be instantiated. With 52% and 48% of BRAMs allocated for the Jacobi and Newton methods, respectively, both benchmarks can reach $K_{\max} = 724$ and $P_{\max} = 5784$. The small increases in non-memory resources noted can be attributed to the additional control logic and multiplexing required to address larger memories. The $f_{\max}$ plots show that my implementations are able to run at between 180 MHz (Jacobi) and 155 MHz (Newton), for the smallest D tested, to around 60 MHz for the largest of both benchmarks with a fixed RAM width U.

ARCHITECT-I gives its users the freedom to trade off area and computation time directly by varying RAM width U. When U is changed, so are the widths of the parallel online adders used in the datapath; while a design with narrower adders is just as able to compute a particular result as one capable of performing wider additions, it will also consume more clock cycles in return for demanding lower resource use. Comparisons between $U = 8$ and $U = 64$ with the same D, in this case 2^{10}, are shown in Table 3.2 to exemplify this for both of the benchmarks. Note that the accumulation latency for Newton's method is higher than Jacobi's due to the former's use of division; as was explained in Section 3.3.2, division requires more cycles to produce each output digit than are required for multiplication.

As was discussed in Section 3.3.6, the use of parallel online addition allows us to avoid the recalculation of the previous iterations' residuals, leading to the elimination of T_{sa} expressed in (3.2) on page 46. Figure 3.10 illustrates the performance speedup due to our architectural optimisations for both Jacobi (—+—) and Newton (—○—). Performance is improved for higher η since it leads to clock cycle savings when switching between iterations. For higher-accuracy cases, this optimisation does not contribute much to solve time speedup, however. This makes sense since, as η falls, more iterations are required to achieve convergence, thus more cycles are required for the production of each new each digit. This also affords much greater opportunity for digit-elision optimisations I will discuss in future chapters.

3.5.5 Quantitative Comparison

In order to compare the resource use and f_{max} of ARCHITECT-I against its competitors, assume that iterative solvers wish to compute to particular (K, P) targets. It is emphasised that, since ARCHITECT-I iterates exactly while LSD-first arithmetic-based solvers do not, latency cannot be fairly compared when considering computation to a particular (K, P).

The computation targets were chosen to set $(100, 2^{11})$ (for the Jacobi method) and $(10, 2^{11})$ (Newton). Thus, at their 100th and 10th iterations, respectively, a result

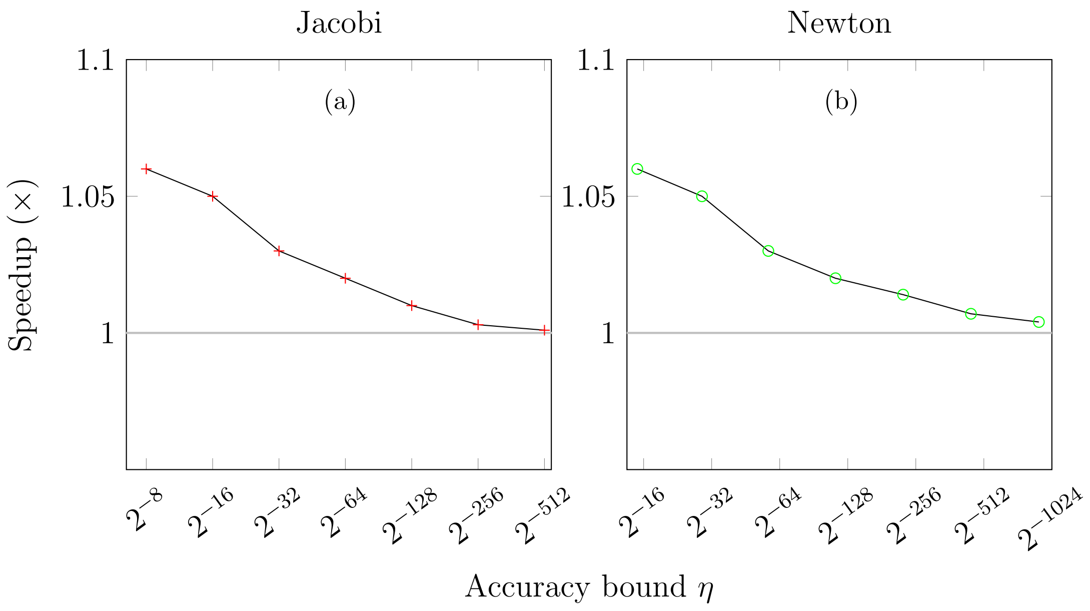


Fig. 3.10. Latency reduction using parallel online adders.

with 2048-digit precision is expected to be achieved. Fewer iterations were targetted for Newton's method due to its quadratic convergence for the test cases described in Section 3.4.2. Using $U = 8$, for ARCHITECT-I, the resultant iteration counts and precisions for the two methods are $(K_{\text{res}}, P_{\text{res}}) = (509, 2545)$ (Jacobi) and $(351, 2106)$ (Newton). To successfully perform computation to (K, P), we must ensure that $K_{\text{max}} \geq K_{\text{res}}$ and $P_{\text{max}} \geq P_{\text{res}}$. It can be determined that, by setting RAM depth $D = 2^{17}$, iterative solvers are able to reach $K_{\text{max}} = 512$ and $P_{\text{max}} = 4088$, which satisfies these requirements for both benchmarks.

Fig. 3.11 presents a side-by-side comparison of the architectures implemented following the principles presented herein and those using PISO operators as well as the online implementation published by Zhao *et al.* [122]. Most strikingly, the latter demonstrates area inefficiency, with resource use scaling linearly with iteration count K; ARCHITECT-I consumes 75× fewer LUTs and 77× fewer FFs than Zhao *et al.*'s proposal requires to execute 100 iterations of the Jacobi method. When executing 10 iterations of Newton's method, these factors are 11 and 18, respectively. f_{max}

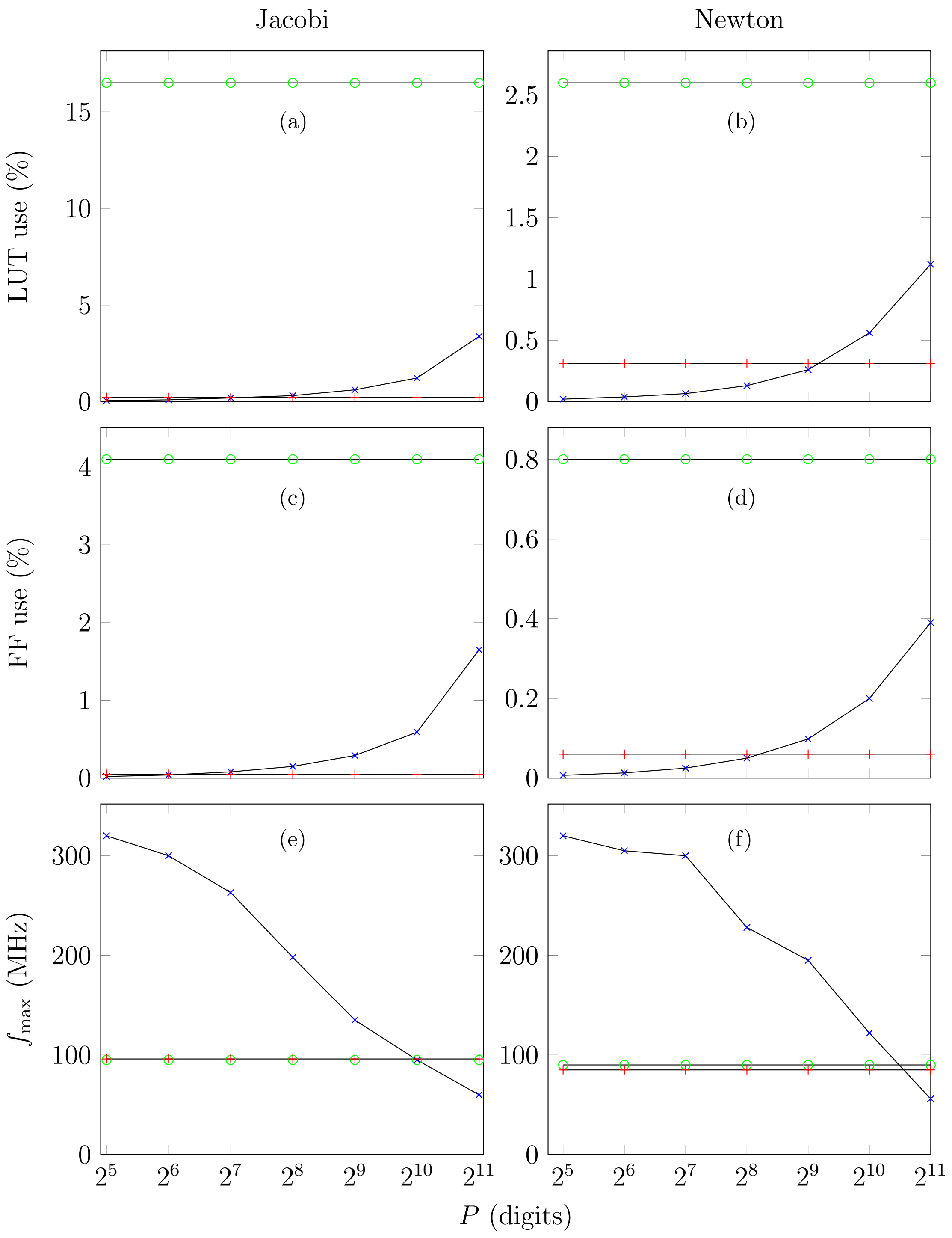


Fig. 3.11. Resource use and performance comparison of Jacobi and Newton's method implementations using Zhao *et al.*'s (—⊖—), PISO (—×—) and our (—+—) approaches versus required result precision P.

is comparable between the two since the underlying arithmetic is largely equivalent, although ARCHITECT-I's is slightly superior. For PISO, while its $f_{\max}$ is initially much higher—over 300 MHz for $P = 2^4$—than ARCHITECT-I's, it falls as P increases. Taking Newton's method as an example, with a high precision requirement, such as 2^{11} digits, ARCHITECT-I is able to outperform its PISO counterpart in terms of $f_{\max}$ by a factor of 1.6. Corresponding decreases in LUT and FF use were also found: when computing to $P = 2^{10}$, again for Newton, ARCHITECT-I consumes 2.4× and 4.4× fewer of each than PISO, while for 2^{11} these factors increase to 4.9 and 8.7. Similar conclusions can be made for the implementation of the Jacobi method. Since the proposed designs are able to calculate to any $K \leq K_{\max}$ and $P \leq P_{\max}$, their areas and $f_{\max}$es are constant.

3.6 Conclusion

This chapter proposed the first hardware architecture capable of executing iterative algorithms to produce results of arbitrary accuracy by combining increasing iteration count with precision while using constant compute resources. This technique is named ARCHITECT-I. This proposal employs online arithmetic to generate its results MSD first and a Cantor pairing function within its digit-storage mechanism to facilitate the simultaneous growth of iteration count and precision. I also proposed the replacement of serial online adders within iterative datapaths with parallel equivalents, facilitating latency reduction and consequent improvements in throughput.

ARCHITECT-I was evaluated on FPGAs using the Jacobi and Newton's methods in order to verify its accuracy and establish its scalability and efficiency. Experimental results showed that datapaths constructed from ARCHITECT-I operators are superior to their traditional arithmetic equivalents in scenarios where the latter's precisions are either overly high (*e.g.* LSD-32) for the problems being solved or too low (*e.g.* LSD-8) for results to converge at all. These benchmarks showcased the key advantage of the approach: removing the burden of having to determine and fix the precisions of arithmetic operators in advance. A single ARCHITECT-I datapath is able to compute results to any accuracy, with the only limit being imposed by the size of the available RAM, while LSD-first solvers have their precisions upper-bounded at compile-time.

Experimental results revealed 15× LUT and 31× FF reductions over 2048-bit conventional parallel-in serial-out arithmetic, along with 75× LUT and 77× FF decreases versus the state-of-the-art online arithmetic implementation, when executing 100 Jacobi iterations. For Newton's method run for 10 iterations, these factors were 4.9, 8.7, 11 and 18, respectively. Versus ARCHITECT-I with the proposed parallel addition optimisations disabled, ARCHITECT-I was able to achieve up-to 1.1 × reduction for Jacobi and Newton's methods.

Use of most-significant digit-first arithmetic additionally allows computation to terminate at any less-significant digit place at runtime. Since low-significance digits of early approximants are generally unimportant, avoiding their calculation in subsequent iterations can increase performance and decrease memory footprints. The next chapter will theorise the presence and avoid the computation of these unimportant LSDs, resulting in a more efficient computation pattern as shown in Fig. 1.1c on page 4.

Chapter 4

Don't-care Digit Elision

4.1 Overview

In the previous chapter, I proposed ARCHITECT-I which achieves exact numeric computation by using online arithmetic to allow the refinement of results from earlier iterations over time, without rounding error. ARCHITECT-I has a key drawback, however: often, many more digits than strictly necessary are generated, with this problem exacerbating the more accurate a solution is sought.

Significant efficiency gains are realisable by exploiting the fact that not all approximants contribute equally to an algorithm's overall error. Low-significance digits of early approximants are often unimportant, thus we call them 'don't-care' digits. ARCHITECT-I with don't-care digit elision sacrifices the feature of exact numeric computation, but can still converge to a solution to arbitrary accuracy.

In this chapter, the locations of don't-care digits are inferred within stationary iterative calculations by exploiting forward error analysis. Their lack of computation is guaranteed not to affect the ability to reach a solution of any accuracy. Versus

ARCHITECT-I without digit elision, the illustrative hardware implementation, named ARCHITECT-II, achieves a geometric mean 21× speedup in the solution of a set of representative linear systems through the avoidance of redundant digit calculation. For the computation of high-precision results, an up-to 25× memory requirement reduction is also obtained over the same baseline.

The following novel contributions are made in this chapter:

- A theorem for the optimal rate of LSD growth per iteration within stationary iterative methods, thereby enabling the avoidance of don't-care digit computation.
- A theorem showing that arbitrary-accuracy solutions can be achieved when omitting don't-care digits during stationary iterative calculations.
- An exemplary hardware implementation of the new proposal using the Jacobi method.
- Performance evaluations of the demonstrative architecture, drawing comparison against ARCHITECT-I and conventional fixed-precision equivalents. A mean 4.1× reduction is observed in the number of digits generated to solve a set of representative linear systems, showing a geometric mean 21× speedup over ARCHITECT-I.

4.2 Notation

The approximant of an iterative method at iteration $k \in \mathbb{N}_{>0}$ is denoted $\boldsymbol{x}^{(k)}$, while its exact result is $\boldsymbol{x}^*$, with their j^{th} element denoted $x_j^{(k)}$ and x_j^*, respectively.

A matrix is represented by a bold capital symbol $\boldsymbol{X}$. The infinity norm of either a matrix or a vector is given by $\|\bullet\|_\infty$.

Let λ be an eigenvalue of an $n \times n$ matrix $\boldsymbol{X}$. The spectral radius of $\boldsymbol{X}$ is denoted by $\rho(\boldsymbol{X}) = \max_{1\le i\le n} |\lambda_i|$, where $\boldsymbol{X}$ has eigenvalues λ_i.

4.3 Theoretical Analysis

Let us now turn to the issue of don't-care digit elision. This term is used to refer to low-significance digits in earlier approximants which do not prohibit the chosen iterative method's convergence. Herein, a novel don't-care digit analysis is presented, applicable to any stationary iterative method: Jacobi, Gauss-Seidel, successive over-relaxation, *etc.*

Consider a linear system $\boldsymbol{Ax} = \boldsymbol{b}$, where $\boldsymbol{A} \in \mathbb{R}^{N\times N}$. Given the definition of stationary iterative methods expressed in (2.2) in Section 2.1.2, approximant by approximant, we have

$$\begin{aligned}
\boldsymbol{x}^{(1)} &= \boldsymbol{G}\boldsymbol{x}^{(0)} + \boldsymbol{M}^{-1}\boldsymbol{b} \\
\boldsymbol{x}^{(2)} &= \boldsymbol{G}^2\boldsymbol{x}^{(0)} + \boldsymbol{G}\boldsymbol{M}^{-1}\boldsymbol{b} + \boldsymbol{M}^{-1}\boldsymbol{b} \\
&\vdots \\
\boldsymbol{x}^{(k+1)} &= \boldsymbol{G}^{k+1}\boldsymbol{x}^{(0)} + \sum_{i=0}^{k} \boldsymbol{G}^i\boldsymbol{M}^{-1}\boldsymbol{b} \qquad (4.1)
\end{aligned}$$

starting from some initial guess $\boldsymbol{x}^{(0)}$.

Lemma 4.1 (Higham [51]). *If* $\rho(\boldsymbol{G}) < 1$, *then* $\sum_{i=0}^{\infty} \boldsymbol{G}^i = (\boldsymbol{I} - \boldsymbol{G})^{-1}$.

Thus,

$$\lim_{k\to\infty} \boldsymbol{G}^k \boldsymbol{x}^{(0)} = \boldsymbol{0}. \tag{4.2}$$

Applying Lemma 4.1 a second time, this observation allows us to conclude from (4.1) that

$$\lim_{k\to\infty} \sum_{i=0}^{k} \boldsymbol{G}^i \boldsymbol{M}^{-1} \boldsymbol{b} = \boldsymbol{x}^*.$$

Hence, (2.1) on page 13 will converge to $\boldsymbol{x}^*$ for any choice of $\boldsymbol{x}^{(0)}$.

Lemma 4.2 (Higham [51])**.** *$\boldsymbol{x}^*$ is a fixed point of the iteration, i.e. $\boldsymbol{x}^* = \boldsymbol{G}^{k+1}\boldsymbol{x}^* + \sum_{i=0}^{k} \boldsymbol{G}^i \boldsymbol{M}^{-1}\boldsymbol{b} \ \ \forall k$.*

Introducing rounding error $\boldsymbol{\epsilon}_k$, as we propose to via truncation of each approximant, (2.2) on page 13 becomes

$$\hat{\boldsymbol{x}}^{(k+1)} = \boldsymbol{G}\hat{\boldsymbol{x}}^{(k)} + \boldsymbol{M}^{-1}\boldsymbol{b} + \boldsymbol{\epsilon}_{k+1}$$

or, expressed per approximant,

$$\begin{aligned}
\hat{\boldsymbol{x}}^{(1)} &= \boldsymbol{G}\hat{\boldsymbol{x}}^{(0)} + \boldsymbol{M}^{-1}\boldsymbol{b} + \boldsymbol{\epsilon}_1 \\
\hat{\boldsymbol{x}}^{(2)} &= \boldsymbol{G}^2\hat{\boldsymbol{x}}^{(0)} + \boldsymbol{G}\boldsymbol{M}^{-1}\boldsymbol{b} + \boldsymbol{G}\boldsymbol{\epsilon}_1 + \boldsymbol{M}^{-1}\boldsymbol{b} + \boldsymbol{\epsilon}_2 \\
&\vdots \\
\hat{\boldsymbol{x}}^{(k+1)} &= \boldsymbol{G}^{k+1}\hat{\boldsymbol{x}}^{(0)} + \sum_{i=0}^{k} \boldsymbol{G}^i \boldsymbol{M}^{-1}\boldsymbol{b} + \sum_{i=0}^{k} \boldsymbol{G}^i \boldsymbol{\epsilon}_{k+1-i} \qquad (4.3)
\end{aligned}$$

from some finite-precision initial guess $\hat{\boldsymbol{x}}^{(0)}$.

Defining computation error $\boldsymbol{e}^{(k)} = \boldsymbol{x}^* - \hat{\boldsymbol{x}}^{(k)}$, subtraction of (4.3) from the equality given in Lemma 4.2 results in

$$\boldsymbol{e}^{(k+1)} = \boldsymbol{G}^{k+1}\boldsymbol{e}^{(0)} - \sum_{i=0}^{k} \boldsymbol{G}^{i}\boldsymbol{\epsilon}_{k+1-i},$$

wherein $\boldsymbol{e}^{(k)}$ captures errors due to the finiteness of both the iteration count *and* the precision. We wish to minimise this value. Since we cannot minimise $\boldsymbol{e}^{(k)}$ directly, we seek to minimise its upper bound instead. Taking norms,

$$\begin{aligned} \left\|\boldsymbol{e}^{(k+1)}\right\|_\infty &\leq \left\|\boldsymbol{G}^{k+1}\boldsymbol{e}^{(0)}\right\|_\infty + \sum_{i=0}^{k} \|\boldsymbol{G}\|_\infty^{i} \left\|\boldsymbol{\epsilon}_{k+1-i}\right\|_\infty \\ &\leq \left\|\boldsymbol{G}^{k+1}\boldsymbol{e}^{(0)}\right\|_\infty + \sum_{i=0}^{k} \|\boldsymbol{G}\|_\infty^{i} r^{-d_{k+1-i}} \end{aligned} \tag{4.4}$$

where $\|\boldsymbol{G}\|_\infty < 1$ and we ensure that $\|\boldsymbol{\epsilon}_{k+1-i}\|_\infty \leq r^{-d_{k+1-i}}$ by controlling the precision of each approximant's computation, expressed as a number of radix-r digits d_j.

For neatness, let $g(i)$ denote the maximum error introduced in approximant i:

$$g(i) = \|\boldsymbol{G}\|_\infty^{i} r^{-d_{k+1-i}}. \tag{4.5}$$

Defining $\boldsymbol{d}$ to be a column vector whose j^{th} element is the number of digits used to represent each element of the j^{th} approximant, and assuming an available 'budget' of total digits D for computation, we aim to find

$$\begin{aligned} \min_{\boldsymbol{d}} f(\boldsymbol{d}) &= \sum_{i=0}^{k} g(i) \\ \text{subject to } h(\boldsymbol{d}) &= -D + \sum_{i=0}^{k} d_{i+1} = 0, \end{aligned} \tag{4.6}$$

thereby determining the optimal allocation of the available digits per approximant.

Theorem 4.1 (The optimal error distribution is uniform)**.** *The optimisation in* (4.6) *is achieved when $g(i)$ is a constant independent of i.*

Proof. Via the Karush-Kuhn-Tucker conditions [11], the optimal $\boldsymbol{d}$, $\boldsymbol{d}^*$, is obtained when

$$\begin{aligned} \nabla f(\boldsymbol{d}^*) + \lambda \nabla h(\boldsymbol{d}^*) &= \mathbf{0} \\ h(\boldsymbol{d}^*) &= 0 \end{aligned}$$

for some multiplier λ. We have

$$\nabla f(\boldsymbol{d}) = \begin{pmatrix} \left(\|\boldsymbol{G}\|_\infty^k \ln r^{-1}\right) r^{-d_1} \\ \left(\|\boldsymbol{G}\|_\infty^{k-1} \ln r^{-1}\right) r^{-d_2} \\ \vdots \\ \left(\|\boldsymbol{G}\|_\infty^0 \ln r^{-1}\right) r^{-d_{k+1}} \end{pmatrix}$$

$$\nabla h(\boldsymbol{d}) = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix},$$

i.e.

$$-\left(\|\boldsymbol{G}\|_\infty^i \ln r\right) r^{-d_{k+1-i}} + \lambda = 0.$$

Therefore, the optimisation in (4.6) is achieved when $g(i) = \lambda/\ln r$, a constant independent of i, as required. □

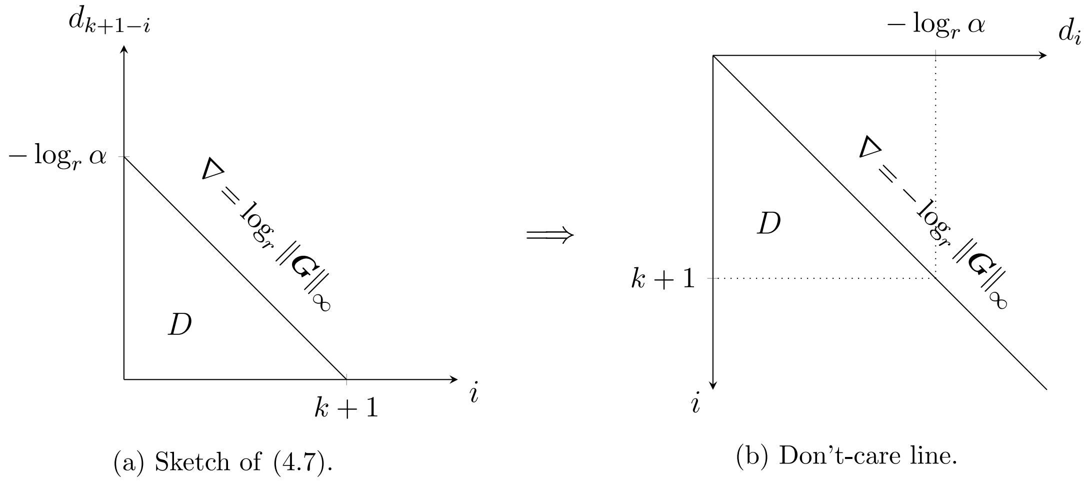


Fig. 4.1. Deriving the gradient of the don't-care line. Figure 4.1b was arrived at by transforming d_{k+1-i}, featured in Figure 4.1a, into d_i, after which it was rotated clockwise by 90° to match the presentation used in Figure 1.1. Since d_i does not feature k, Figure 4.1b's x-intercept (here, the origin) can be chosen arbitrarily.

Setting $g(i) = \alpha$ and taking logs, we obtain

$$d_{k+1-i} = i \log_r \|\boldsymbol{G}\|_\infty - \log_r \alpha. \tag{4.7}$$

I present the transformation of this function to the required don't-care line graphically in Figure 4.1. From Figure 4.1b, we can infer that the optimal gradient of the don't-care line is $-\log_r \|\boldsymbol{G}\|_\infty$, and is therefore independent of D and α. The line's x-intercept is analogous to the precision with which one wishes to begin computation, so is user-defined.

Let us now analyse the limit of $\left\|\boldsymbol{e}^{(k)}\right\|_\infty$ when the proposed don't-care line is used during stationary iterative computation.

Theorem 4.2 (Error can be arbitrarily minimised)**.** $\lim_{k\to\infty} \left\|\boldsymbol{e}^{(k)}\right\|_\infty = 0.$

Proof. Applying (4.7) to (4.4) leads to

$$\begin{aligned}
\left\|\boldsymbol{e}^{(k+1)}\right\|_\infty &\leq \left\|\boldsymbol{G}^{k+1}\boldsymbol{e}^{(0)}\right\|_\infty + \sum_{i=0}^{k} \|\boldsymbol{G}\|_\infty^i \, r^{-(i \log_r \|\boldsymbol{G}\|_\infty - \log_r \alpha)} \\
&= \left\|\boldsymbol{G}^{k+1}\boldsymbol{e}^{(0)}\right\|_\infty + \sum_{i=0}^{k} \|\boldsymbol{G}\|_\infty^i \, \|\boldsymbol{G}\|_\infty^{-i} \, \alpha \\
&= \left\|\boldsymbol{G}^{k+1}\boldsymbol{e}^{(0)}\right\|_\infty + \alpha \sum_{i=0}^{k} 1 \\
&= \left\|\boldsymbol{G}^{k+1}\boldsymbol{e}^{(0)}\right\|_\infty + (k+1)\alpha
\end{aligned} \tag{4.8}$$

Given that α is a constant independent of $i \in [0, k]$, from (4.5) we know that

$$\alpha = g(0) = r^{-d_{k+1}}. \tag{4.9}$$

Applying this to (4.8) results in

$$\left\|\boldsymbol{e}^{(k+1)}\right\|_\infty \leq \left\|\boldsymbol{G}^{k+1}\boldsymbol{e}^{(0)}\right\|_\infty + \frac{k+1}{r^{d_{k+1}}}. \tag{4.10}$$

From (4.2), we can infer that

$$\lim_{k\to\infty} \boldsymbol{G}^k = \mathbf{0}. \tag{4.11}$$

Combining (4.10) and (4.11), the limit of the computation error is

$$\lim_{k\to\infty} \left\|\boldsymbol{e}^{(k+1)}\right\|_\infty \leq \lim_{k\to\infty} \frac{k+1}{r^{d_{k+1}}}.$$

Via the Stolz–Cesàro theorem [91],

$$\lim_{k\to\infty} \frac{k+1}{r^{d_{k+1}}} = \lim_{k\to\infty} \frac{k+2-(k+1)}{r^{d_{k+2}} - r^{d_{k+1}}}.$$

As was illustrated in Fig. 4.1, the slope of the proposed don't-care line is only dependent upon $\|\boldsymbol{G}\|_\infty$. Since $\|\boldsymbol{G}\|_\infty \in (0,1)$, the gradient of don't-care lines, *i.e.* $-\log_r \|\boldsymbol{G}\|_\infty$ is always positive. Therefore, $\left(r^{d_{k+1}}\right)_{k\geq 1}$ is strictly monotone and divergent. From (4.7) and (4.9), we know that

$$\begin{aligned} r^{d_{k+2}} &= r^{-\log_r \|\boldsymbol{G}\|_\infty - \log_r r^{-d_{k+1}}} \\ &= r^{\log_r \|\boldsymbol{G}\|_\infty^{-1} + d_{k+1}} \\ &= \frac{r^{d_{k+1}}}{\|\boldsymbol{G}\|_\infty}. \end{aligned} \tag{4.12}$$

Thus,

$$\begin{aligned} \lim_{k\to\infty} \frac{k+1}{r^{d_{k+1}}} &= \lim_{k\to\infty} \frac{1}{\frac{r^{d_{k+1}}}{\|\boldsymbol{G}\|_\infty} - r^{d_{k+1}}} \\ &= \lim_{k\to\infty} \frac{\|\boldsymbol{G}\|_\infty}{r^{d_{k+1}}\left(1 - \|\boldsymbol{G}\|_\infty\right)} \end{aligned} \tag{4.13}$$

which, since $\|\boldsymbol{G}\|_\infty < 1$ and $r^{d_{k+1}} \to \infty$ as $k \to \infty$, is zero.

□

Therefore, iterative calculations performed using the proposed don't-care line are guaranteed to converge to a solution of any accuracy, as required.

4.4 Implementation

Armed with the analysis presented in Section 4.3, we are now in a position to design suitable control and storage infrastructure to support the efficient generation of digits within ARCHITECT implementations.

4.4.1 Control Logic

For every element of $\hat{\boldsymbol{x}}^{(k)}$, $\hat{x}_j^{(k)}$, two factors determine the number of its digits that must be calculated:

- *The initial guess.* Assume that Figure 4.1's don't-care line is placed such that the minimum number of digits needed to represent $\hat{x}_j^{(0)}$ exactly are used. Thus, each approximant requires a minimum of

  $$\chi = -\log_r \mathrm{ULP}\left(\hat{x}_j^{(0)}\right)$$

  digits for its representation, where ULP returns the unit in the last place of its argument.

- *Don't-care LSDs.* The don't-care line's gradient is indicative of the number of additional digits to calculate per approximant. Hence, within approximant k, we *do* care about

  $$\omega(k) = \lceil -k \log_r \|G\|_\infty \rceil$$

  more digits than were contained in the initial guess. $\omega(k)$ is always rounded towards $+\infty$ to ensure that we do not inadvertently neglect to calculate any required LSDs.

Combining these, we arrive at

$$\beta_j(k) = \chi + \omega(k)$$

where $\beta_j(k)$ reveals the number of digits to produce per approximant.

A parameterisable finite-state machine (FSM) was designed to sequence digit propagation through a pipeline of online operators. This is fundamentally different to the

FSM used in ARCHITECT-I, I hence modified it, with implications for memory usage. Beginning with the consumption of approximant $k = 0$ (the user-supplied initial guess), it sweeps through $\beta_j(k)$ digits per approximant, incrementing k after each calculation.

From a user's perspective, no additional information needs to be supplied to take advantage of don't-care digit elision. The system to be solved, defined by $\boldsymbol{A}$ and $\boldsymbol{b}$, along with $\hat{\boldsymbol{x}}^{(0)}$, are all that is required.

4.4.2 Memory

Since the proposed don't-care analysis tells us the number of digits needed to be computed per approximant *a priori*, approximants are able to be computed, in full, sequentially. This was demonstrated visually in Figure 1.1d. Since the number of memory words (chunks) required for each approximant grows over time, With this don't-care line, each approximant k calculated by ARCHITECT-II can overwrite its previous one, leading to a significant reduction in memory requirement versus ARCHITECT-I.

Output digit storage is not our only memory-related concern. Datapath following ARCHITECT's principle also requires residue storage within its multipliers and dividers which were presented in Section 3.3.2. As a result of these sources of increasing storage burden, the size of the instantiated memories will determine the maximum precision and number of iterations (and consequently accuracy) to which one can compute. The arbitrary-accuracy claim is subject to the availability of sufficient memory to solve the particular problem at hand.

4.5 Evaluation

Hardware performance evaluations were conducted to compare ARCHITECT-II against ARCHITECT-I and conventional LSD-first equivalents. For the latter, datapaths are composed of parallel-in, serial-out (PISO) arithmetic operators for which precision must be set at compile time, as was done to evaluate ARCHITECT-I. PISO functions in a similar digit-serial fashion to those used for ARCHITECT-I and this work.

Compared to iterative computation architectures constructed using conventional arithmetic (LSD-first) operators, we expect ARCHITECT-II to compare favourably due to:

- *There being no need to determine, or fix, precision in advance.* This is the arbitrary-precision feature I achieved in both ARCHITECT-I and ARCHITECT-II.
- *Don't-care LSD elision.* The theoretical don't-care analysis facilitates the growth of precision over time. This allows us to focus exclusively on digits known to be of value, thereby increasing efficiency. This is difficult to achieve in LSD-first architectures, in which every approximant must ordinarily be calculated to a maximum (worst-case) precision.

Versus ARCHITECT-I, we hypothesise that ARCHITECT-II will achieve greater performance thanks to:

- *Don't-care digit elision.* Beyond the benefits outlined above, additional performance gains can be obtained through digit generation avoidance in MSD-first architectures. It is expected that don't-care digit avoidance will significantly improve performance effectively since digit generation requires more time the lower the significance of the digit. This is a property of the hardware used for operators which are themselves inherently iterative, *e.g.* multipliers.

- *A more efficient digit computation pattern.* As exemplified in Figure 1.1b, ARCHITECT-I refines earlier approximants as needed in order to reach further into the iteration-precision space. Since the don't-care line is monotone, there is no need to revisit earlier iterations, as was demonstrated in Figure 1.1d.

- *Significantly lower memory usage.* Closely related to the previous point, the simplified computation pattern permits us to discard older approximants and their residues. This allows us to do away with ARCHITECT-I's Cantor paring function, with memory use scaling only with precision.

All of our hardware implementations targetted a Xilinx Virtex UltraScale FPGA (XCVU190-FLGB2104-3-E), with Vivado 16.4 used for compilation. Results obtained in hardware were compared for correctness against a golden software model. The Jacobi method, a well known stationary iterative algorithm, was used as a case study for the proposals presented in this chapter.

4.5.1 Performance Comparison

To begin, we investigated how the conditioning of $\boldsymbol{A}$ affected the performance of ARCHITECT-I and ARCHITECT-II relative to conventional LSD-first arithmetic. Mirroring the experiments conducted to evaluate ARCHITECT-I, a set of

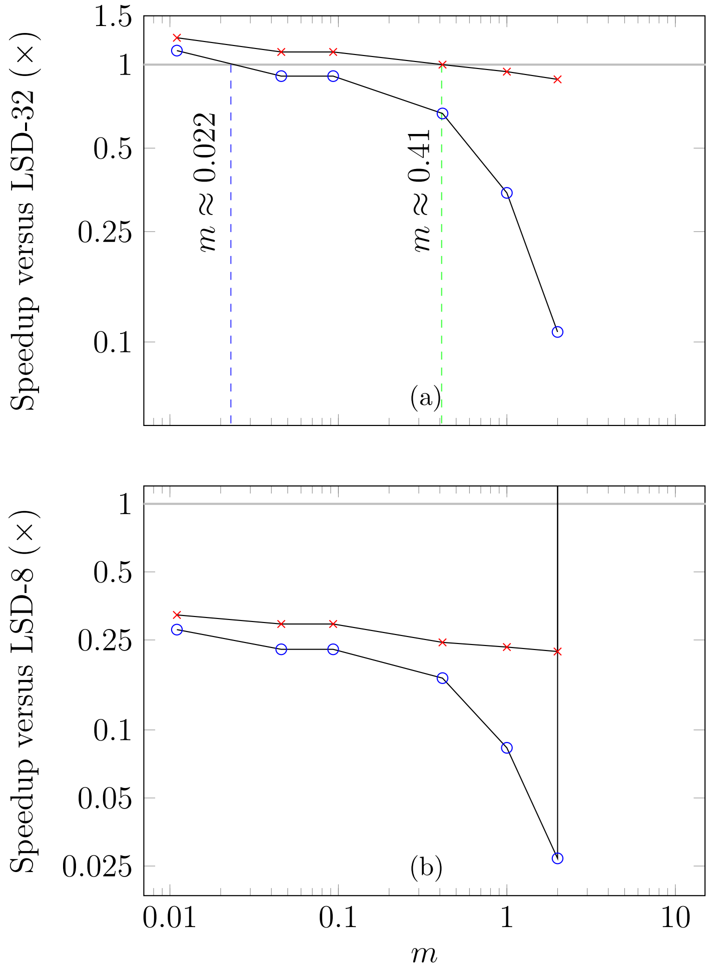


Fig. 4.2. A comparison of ARCHITECT-I and ARCHITECT-II against conventional LSD-first arithmetic equivalents. (a) shows how the conditioning of input matrix $\boldsymbol{A}_m$ affects the solve time of both ARCHITECT-I (—o—) and ARCHITECT-II (—×—) compared to LSD-32. ARCHITECT-I computes faster than LSD-32 only when $m \lessapprox 0.022$, but ARCHITECT-II beats LSD-32 when $m \lessapprox 0.41$. (b) shows that even though both ARCHITECT-I and ARCHITECT-II lead to a slowdown compared to LSD-8, there is nevertheless a point—at $m > 2$—whence LSD-8 does not converge at all, hence ARCHITECT's speedups are infinite.

linear systems $\boldsymbol{A}_m\boldsymbol{x} = \boldsymbol{b}$ were used with

$$\boldsymbol{A}_m = \begin{pmatrix} 1 & 1 - 2^{-m} \\ 1 - 2^{-m} & 1 \end{pmatrix},$$

elements of $\boldsymbol{b}$ selected from a uniform distribution bounded to $[0, 1)$ and we calculated the accuracy bound $\left\|\boldsymbol{A}\boldsymbol{x}^{(k)} - \boldsymbol{b}\right\|_\infty < \eta$ with $\eta = 2^{-6}$. Figure 4.2 illustrates the speedup in solve time of ARCHITECT-II with and ARCHITECT-I without don't-care digit elision. Note that speedups below unity are slowdowns.

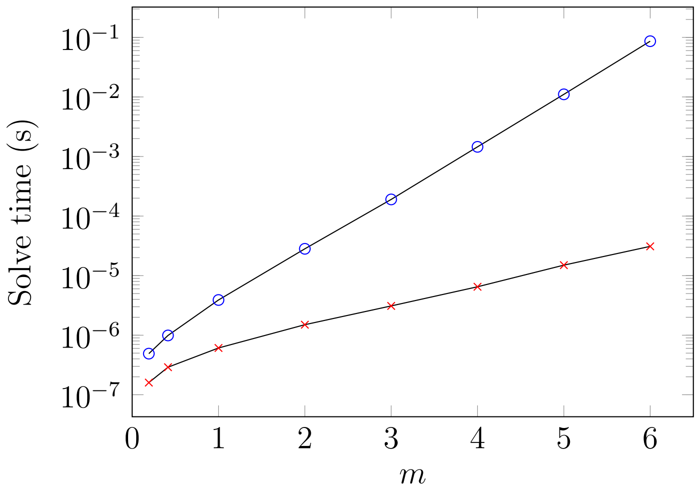


Fig. 4.3. How the conditioning of $\boldsymbol{A}_m$ affects the performance speedups of ARCHITECT-I (—○—) and ARCHITECT-II (—×—).

Figure 4.2a compares ARCHITECT-I and ARCHITECT-II against LSD-first arithmetic implementation with a fixed precision of 32 bits (LSD-32). This precision is over-budgeted for the solution of well conditioned matrices, hence it can be seen that, when m is small, both ARCHITECT-I and -II can compute more quickly. ARCHITECT-I requires $m \lessapprox 0.022$ in order to beat LSD-32, whereas ARCHITECT-II only requires $m \lessapprox 0.41$. Figure 4.2b demonstrates that if LSD-first arithmetic is given an under-budgeted precision of just eight bits (LSD-8), then only arbitrary-precision iterative solvers can solve ill-conditioned systems with $m > 2$. Even if LSD-8 could run indefinitely, it would never be able to converge to an accurate-enough result. The emphasis here is that, however optimised a conventional implementation is, if its precision is sufficiently over- or under-budgeted, an ARCHITECT implementation will eventually outperform it.

To further evaluate the performance of the new proposal, Figure 4.3 presents a side-by-side comparison of solving different linear systems using ARCHITECT-I and ARCHITECT-II. The results show that, generally, as m increases, more time is required in order to achieve a sufficiently accurate result. Most strikingly, ARCHITECT-II demonstrates high efficiency with corresponding decreases in computation time being found versus ARCHITECT-I. When computing a well conditioned matrix with $m = 0.19$, our design is 2.8× faster than ARCHITECT-I, while for $m = 6$ it is some 2500× faster. This proposal outperformed ARCHITECT-I in all tested cases, with a geometric mean 21× speedup obtained.

4.5.2 Scalability Comparison

To evaluate the scalability of ARCHITECT-II, Figure 4.4 shows how the requested accuracy, controlled by η, affects solve time, the number of computed digits and the corresponding memory requirement.

It can be seen from Figure 4.4a that ARCHITECT-I requires increasingly more time to reach a solution than ARCHITECT-II as the requested accuracy increases. With low accuracy requirements, such as $\eta = 2^{-4}$, this method is 2.3× faster than ARCHITECT-I. In the case of high accuracy, such as when computing to $\eta = 2^{-256}$, the optimised method is 48× faster. From a more fundamental perspective, Figure 4.4b shows the relationship between the requested accuracy and the total number of digits calculated. Thanks to don't-care digit elision, ARCHITECT-II calculates 1.2× fewer digits than ARCHITECT-I with $\eta = 2^{-4}$, increasing to 6.6× fewer digits when $\eta = 2^{-256}$.

Finally, Figure 4.4c shows the minimum number of on-chip memory blocks that need to be instantiated in order for ARCHITECT-I and ARCHITECT-II to reach particular accuracies. For lower-accuracy cases ($\eta \geq 2^{-32}$), both designs require

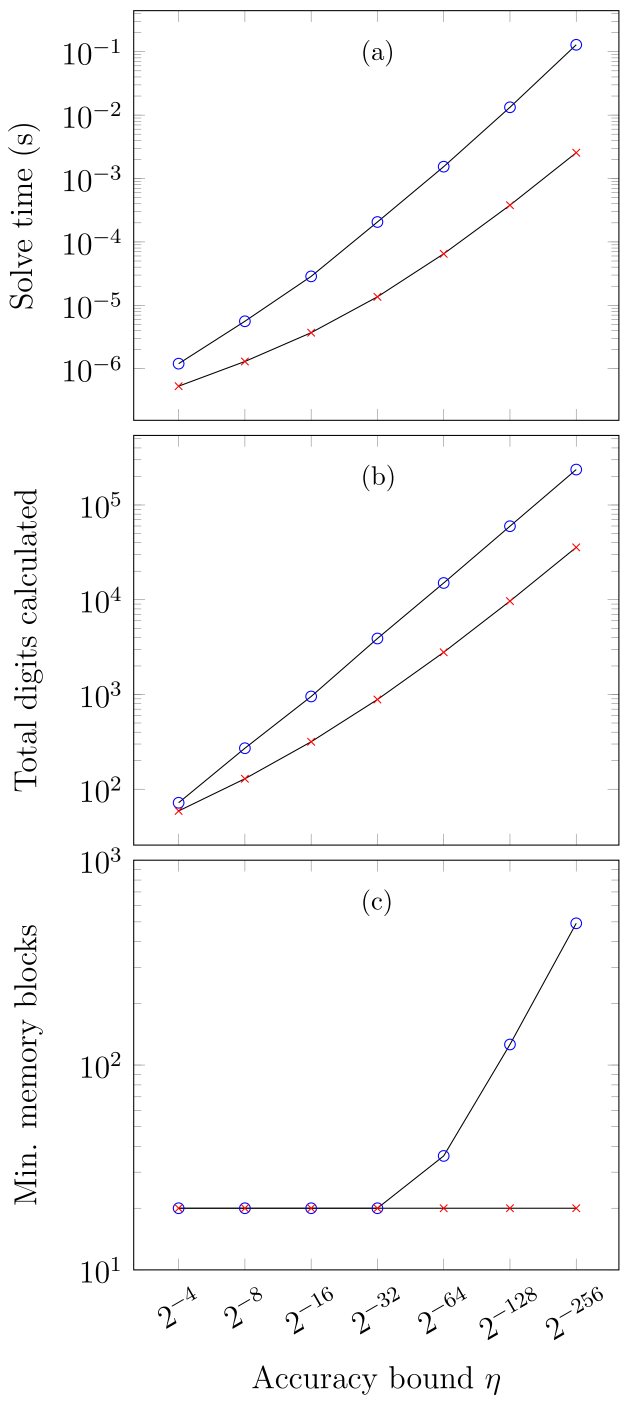


Fig. 4.4. How the requested accuracy bound η affects (a) the solve time, (b) the total number of digits calculated and (c) the minimum memory requirement for ARCHITECT-I (—○—) and ARCHITECT-II (—×—), with $m = 1$.

approximately the same amount of memory, where the constant quantity is due to the granularity of the FPGA memory blocks. The advantages over ARCHITECT-

II's memory addressing explained in Section 4.4.2 come to the fore with higher accuracy requirements. For the lowest η tested, 2^{-256}, I observed a $25\times$ memory reduction.

4.6 Conclusion

In this chapter, a new methodology ARCHITECT-II was proposed for the creation of iterative numeric solvers in hardware. Efficiency over ARCHITECT-I was achieved by identifying unimportant—don't-care—digits, excluding them from calculation. For the identification of don't-care digits, a theorem was proposed for stationary iterative methods allowing many low-significance digits to be ignored without impacting upon the solver's ability to reach a result of any accuracy.

The proposed method was evaluated using the Jacobi method. Versus ARCHITECT-I, experimental results showed a mean $4.1\times$ reduction in the number of digits generated in order to solve a set of differently conditioned matrices to the same accuracy. In those cases, elision of redundant digit calculation led to a geometric mean $21\times$ speedup. The monotonicity of the don't-care line allows us to eliminate the need to revisit and refine earlier approximants, leading to an up-to $25\times$ memory footprint reduction. Making more efficient use of a given-sized memory enables us to advance much deeper into the iteration-precision space than ARCHITECT-I allowed. Finally, versus a fixed-precision Jacobi solver constructed from conventional LSD-first arithmetic operators, ARCHITECT-II was shown to be able to solve more difficult linear systems than ARCHITECT-I did, and that less solve time is required when a system is well conditioned.

In the future, one possible research direction is to extend the proposed don't-care digit analysis to other iterative algorithms, including Krylov subspace methods such as conjugate gradient descent. Exploring the possibility of don't-change digit analysis is also very interesting, since convergence in iterative computation can be considered as agreement in MSDs. Don't-change digit analysis can be either generic or algorithm-specific, and this allows us to do away with restarting each approximant from the most significant digit every iteration, thereby achieving further performance improvements and memory reductions. I am particularly keen to see if it is possible to obtain the same rates of growth in MSD stability and LSD significance, thus achieving parallel don't-change and don't-care lines. Doing so would enable the creation of efficient, fixed compute-resource hardware with no bounds on accuracy, and may allow us to generalise the E-method [32].

Chapter 5

Don't-change Digit Elision

5.1 Overview

I have demonstrated that don't-care digit elision leads to performance increases and memory footprint reductions for stationary iterative calculations. On top of the benefits of don't-care digit elision, further efficiency gains are realisable by exploiting the fact that convergence in iterative computation can be considered as agreement in MSDs. This behaviour is easily found in standard numerical computing: a few new correct digits may be generated in each approximant. However, we always recompute all of the digits in each iteration, even the don't-change ones, *i.e.* digits of later approximants that are identical to ones in the previous approximant. In this chapter, we ask "why do we waste time and energy to recompute them?"

Two theoretical analyses are presented to infer don't-change digits. To begin, a generic analysis is introduced exploiting online arithmetic's digit dependencies to determine identical MSDs. With the knowledge that some D MSDs are common to approximants k and $k+1$, the first $D-\delta$ MSDs in approximant $k+2$ are guaranteed to be identical to its predecessors. This allows the generation of those digits to be skipped, increasing efficiency. This technique is applicable to any online arithmetic-implemented iterative calculation, however it is unfortunately unable to guarantee that digits will stabilise, *i.e.* never change in any future iteration.

Thereafter, a specialised analysis is introduced for stationary iterative methods, combining the knowledge of MSDs shared between successive approximants with matrix conditioning to infer digit stability. Interval and forward error analyses are used to prove that digits of high significance will become stable. I analyse the relationship between system matrix conditioning and the rate of growth in MSD stability, using this information to converge to the desired results more quickly.

The contributions of this chapter are as follows.

- Theoretical analysis to determine the position of identical digits within any online arithmetic-implemented iterative method.

- Using interval and forward error analyses, a theorem for the rate of stable MSD growth within the approximants produced by any stationary iterative method.

- Theoretical comparison of two proposals and the E-method, allowing the skipping of MSD calculation. Benefits and limitations of these methods are discussed.

- Empirical comparisons against the traditional and previous arbitrary-precision iterative solvers on FPGAs, demonstrating performance speedups and memory savings when employing don't-change digit elision.

5.2 Notation

We assume the use of a fixed-point radix-r signed-digit number representation system with maximal redundancy. Digits therefore belong to the digit set $\{-r+1, -r+2, \cdots, r-2, r-1\}$.

A scalar value is denoted by a normal symbol x. For convenience, we assume that all redundantly represented numbers have $|x| < 1$, and can thus be expressed as $x = \sum_{i=1}^{D} x_i r^{-i}$, where x_i is the i^{th} MSD of a D-digit number.

A vector is represented by a bold symbol $\boldsymbol{x}$, with its j^{th} element denoted x_j. Where a vector consists of signed-digit numbers, x_{ji} is the i^{th} MSD of the j^{th} element of $\boldsymbol{x}$.

The approximant of an iterative method at iteration $k \in \mathbb{N}_{>0}$ is denoted $\boldsymbol{x}^{(k)}$, while its exact result is $\boldsymbol{x}^*$. The algorithm residual of an iterative method at iteration k is $\boldsymbol{s}^{(k)} = \boldsymbol{x}^{(k)} - \boldsymbol{x}^*$.

A matrix is represented by a bold capital symbol $\boldsymbol{X}$. The infinity norm of either a matrix or a vector is given by $\|\bullet\|_\infty$.

5.3 Method-agnostic Don't-change Digit Elision

5.3.1 Theoretical Analysis

Thanks to the use of online arithmetic, when advancing downwards in our iteration-precision space, we can avoid the recalculation of don't-change digits. This is generally not possible in LSD-first architectures, in which carries can propagate from LSD to MSD. Don't-change digit elision is guaranteed to be an error-free transformation: no approximation is induced through its application.

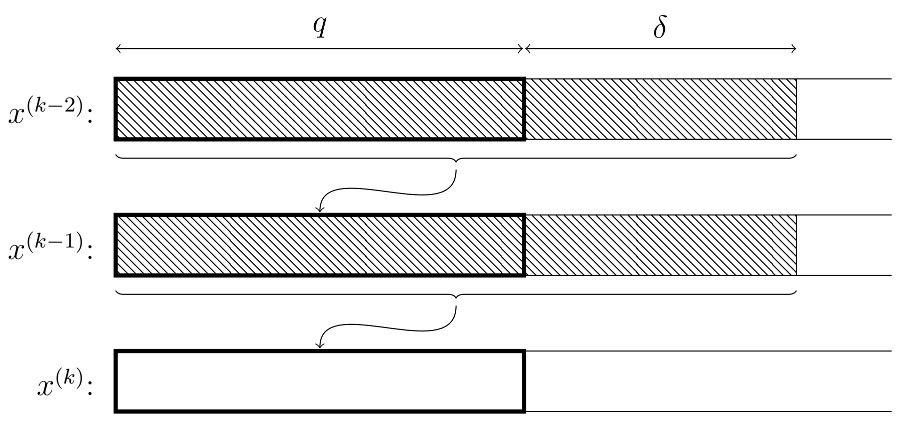


Fig. 5.1. A proof sketch showing why it is sound to omit don't-change digits. If the two hatched regions contain the same $q + \delta$ digits, the three thick boxes are guaranteed to contain the same q digits, hence $x^{(k)}$'s calculation can begin at digit $q + 1$.

The concept behind this optimisation is straightforward. Before beginning to calculate the digits of approximant k, we examine the digits of the previous two approximants. If these approximants are equal in their most-significant $q + \delta$ digits, it is guaranteed that approximant k will be equal to its two predecessors in its first q digits. Hence, we do not calculate them, thus we can skip directly to digit q's generation.

The soundness of this optimisation can be justified by appealing to the digit dependencies of online arithmetic. Fig. 5.1 provides some graphical intuition. Given that each approximant depends only on the value of its immediate predecessor, and recalling the definition of online delay from Section 2.4, we emphasise that the first q digits of one approximant depend only upon the first $q + \delta$ digits of the previous approximant [37]. Hence, if approximants $k - 2$ and $k - 1$ are equal in their first $q + \delta$ digits, approximant k is guaranteed to be equal to them in its first q digits.

5.3.2 Implementation

Online delay-based don't-change digit elision is applicable to any online arithmetic-based iterative calculation. By detecting the presence and avoiding the recomputation of don't-change digits, ARCHITECT-III, *i.e.* ARCHITECT-I (Fig. 1.1b) with don't-change elision, produces digit patterns such as that shown in Fig. 5.2. This increases efficiency while having no bearing on the chosen iterative method's ability to reach a result of any accuracy.

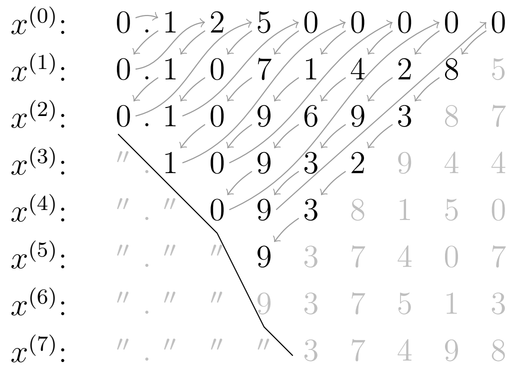

$x^{(0)}$: 0 . 1 2 5 0 0 0 0 0
$x^{(1)}$: 0 . 1 0 7 1 4 2 8 5
$x^{(2)}$: 0 . 1 0 9 6 9 3 8 7
$x^{(3)}$: " . 1 0 9 3 2 9 4 4
$x^{(4)}$: " . " 0 9 3 8 1 5 0
$x^{(5)}$: " . " " 9 3 7 4 0 7
$x^{(6)}$: " . " " 9 3 7 5 1 3
$x^{(7)}$: " . " " " 3 7 4 9 8

Fig. 5.2. Example of a digit-calculating pattern of ARCHITECT-III, for the solution of $x^{(k+1)} = 1/8 - 1/7 \cdot x^{(k)}$. Arrows show the order of digit generation.

Suitable control and storage infrastructure were designed to support the efficient generation of digits within iterative calculations. During the generation of approximant k, we compare digits on-the-fly with those generated for approximant $k - 1$, previously stored in RAM. Based on the number of digits found to be identical, we store a pointer indicating whence approximant $k + 1$'s, *i.e.* the *next* approximant's, generation should begin. Pointer storage requires a small amount of extra memory but, as will be elaborated upon in Section 5.3.3, this overhead is small and amortised out the more RAM is instantiated for digit vector storage.

As a result of the introduction of don't-change digit elision, ARCHITECT-III's scheduling pattern becomes dynamic. Fig. 5.3 shows an example. This is similar to Fig. 3.4 for ARCHITECT-I, but, due to the identification of the third approxi-

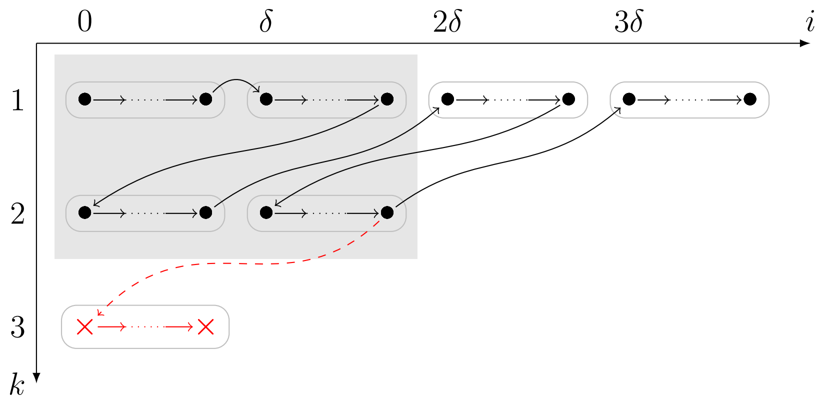


Fig. 5.3. Digit generation pattern with don't-change digit elision. Groups of digits in the shaded region were found to be identical at runtime, allowing computation of the first group to be skipped in the subsequent iteration. Dashed lines are scheduled paths not taken and $\times$s are digits therefore elided.

mant's first group of MSDs as identical, we can advance into the iteration-precision space more quickly than had we not elided these digits, thereby increasing compute efficiency.

Along with increased performance, the elision of don't-change digits also enables us to increase memory efficiency. Let ϕ be the number of digits guaranteed to be identical within the current approximant (k), as determined through the runtime comparison of MSDs within the preceding two approximants. We can substitute

$$\text{cpf}(k, \hat{c}) = \frac{(k+\hat{c})\,(k+\hat{c}+1)}{2} + \hat{c},$$

for (3.1) on page 37, where $\hat{c} = \left\lfloor {}^{(i-\phi)}/_{U} \right\rfloor$. By doing so, identical digits no longer need to be recomputed or stored. In common with (3.1), this optimised strategy guarantees no memory wastage through the surjectivity of its mapping from approximant and chunk indices to memory addresses.

Control Logic

Given a particular (k, i), *i.e.* the i^{th} digit at approximant k, we can compute the subsequent digit to realise scheduling patterns such as those shown in Fig. 5.3 with the finite-state machine (FSM) depicted in Fig. 5.4. The state transition conditions are similar to the FSM for ARCHITECT-I, as was discussed in Section 3.3.4, but suits don't-change digit elision, for which ϕ must be considered.

5.3.3 Evaluation

In order to evaluate ARCHITECT-III, Jacobi and Newton were chosen as benchmarks to exemplify a large class of iterative methods with linear and quadratic convergence properties, respectively. Their datapaths are shown in Figs.3.6a and 3.6b. ARCHITECT-III implementations featured all of the previously described optimisations: online delay-based don't-change digit elision and its related memory-addressing and digit-scheduling schemes.

A Xilinx Virtex UltraScale FPGA (XCVU190-FLGB2104-3-E) was targetted for all experiments detailed henceforward, with implementation performed using Vivado 16.4. The correctness of results obtained in hardware was verified via comparison against those produced by golden models executed in software.

Performance Improvement Breakdown

Analysis was conducted to investigate how the elision of don't-change digits improves the performance and memory efficiency over ARCHITECT-I. Overall, Figs 5.5a and 5.5b show that solve time can be reduced significantly versus ARCHITECT-I. As expected, the speedups versus ARCHITECT-I we observed for Newton's

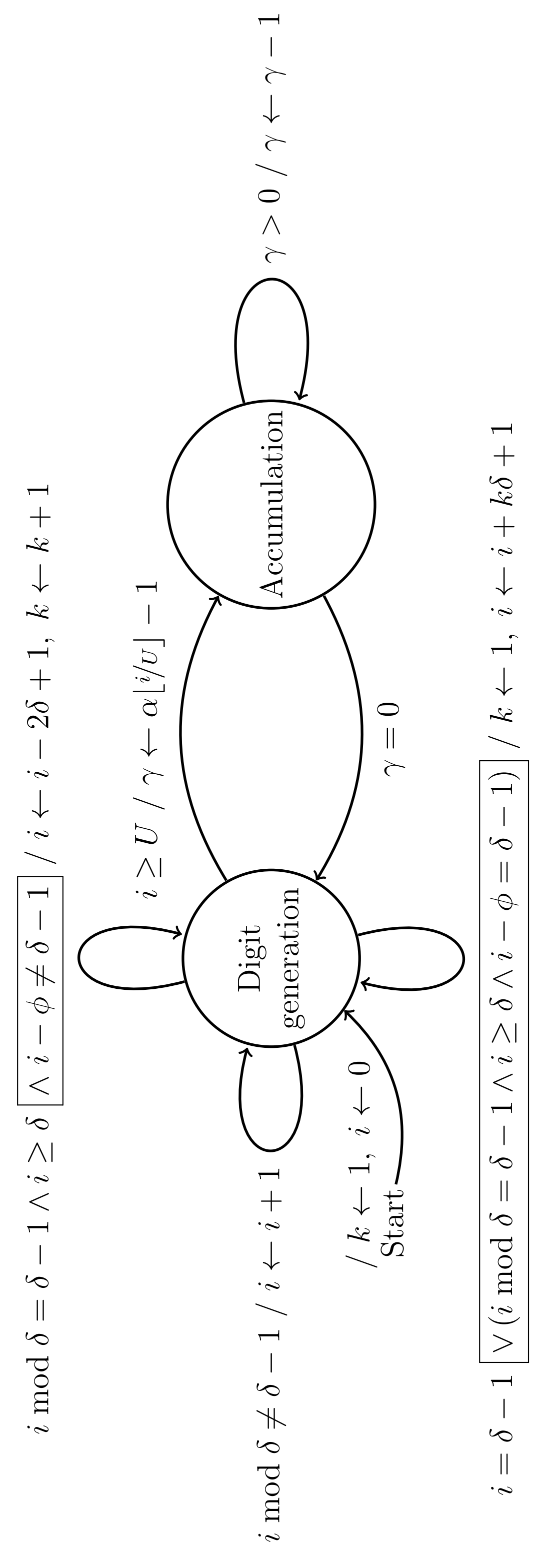

Accumulation
Digit generation
$\gamma > 0$ / $\gamma \leftarrow \gamma - 1$
$i \geq U$ / $\gamma \leftarrow \alpha\lfloor i/v \rfloor - 1$
$\gamma = 0$
$i \bmod \delta = \delta - 1 \wedge i \geq \delta \boxed{\wedge\, i - \phi \neq \delta - 1}$ / $i \leftarrow i - 2\delta + 1$, $k \leftarrow k + 1$
$i \bmod \delta \neq \delta - 1$ / $i \leftarrow i + 1$
Start
/ $k \leftarrow 1$, $i \leftarrow 0$
$i = \delta - 1 \boxed{\vee\, (i \bmod \delta = \delta - 1 \wedge i \geq \delta \wedge i - \phi = \delta - 1)}$ / $k \leftarrow 1$, $i \leftarrow i + k\delta + 1$

Fig. 5.4. FSM for digit computation scheduling with don't-change digit elision only. Transition edges are labelled with conditions and actions separated by slashes (/). Boxed conditions apply when don't-change digit elision is active; they are otherwise ignored, as was shown in Fig. 3.4 on page 43.

method were far more significant than those for Jacobi: up to 16× for the former. Far fewer don't-change digits are detected and elided during computation than for the quadratic-convergence Newton's method, hence the less-significant latency reductions seen in Fig. 5.5a than Fig. 5.5b. The subtle jump present in Fig. 5.5a is due to the δ-digit granularity of elision.

Relatively low performance improvements were expected for the Jacobi benchmark due to the method's linear convergence. For higher-accuracy cases, don't-change digit elision does not contribute much to solve time speedup. This makes sense since, as η falls, more iterations are required to achieve convergence, thus more cycles are required for the production of each new digit.

Figs 5.5c and 5.5d show the memory efficiency improvements afforded through the use of don't-change digit elision for both benchmarks. I present these as the ratio of the number of BRAM blocks that must be instantiated on the targetted FPGA for the solution of equations to particular accuracies for ARCHITECT-I and -III. For lower-accuracy cases ($\eta \geq 2^{-32}$), both designs require approximately the same amount of memory, although ARCHITECT-III is inferior due to the overheads involved in comparison and subsequent elision. For highest-accuracy cases, we observed up-to 1.5× and 1.9× memory savings for the Jacobi and Newton's methods, respectively. There are particularly high-accuracy cases—$\eta \geq 2^{-874}$ for Jacobi and $\eta \geq 2^{-7169}$ for Newton—ARCHITECT-I cannot reach before it exhausts its available memory, while ARCHITECT-III can. The advantages of this scheme and its efficient memory addressing therefore come to the fore with higher accuracy requirements.

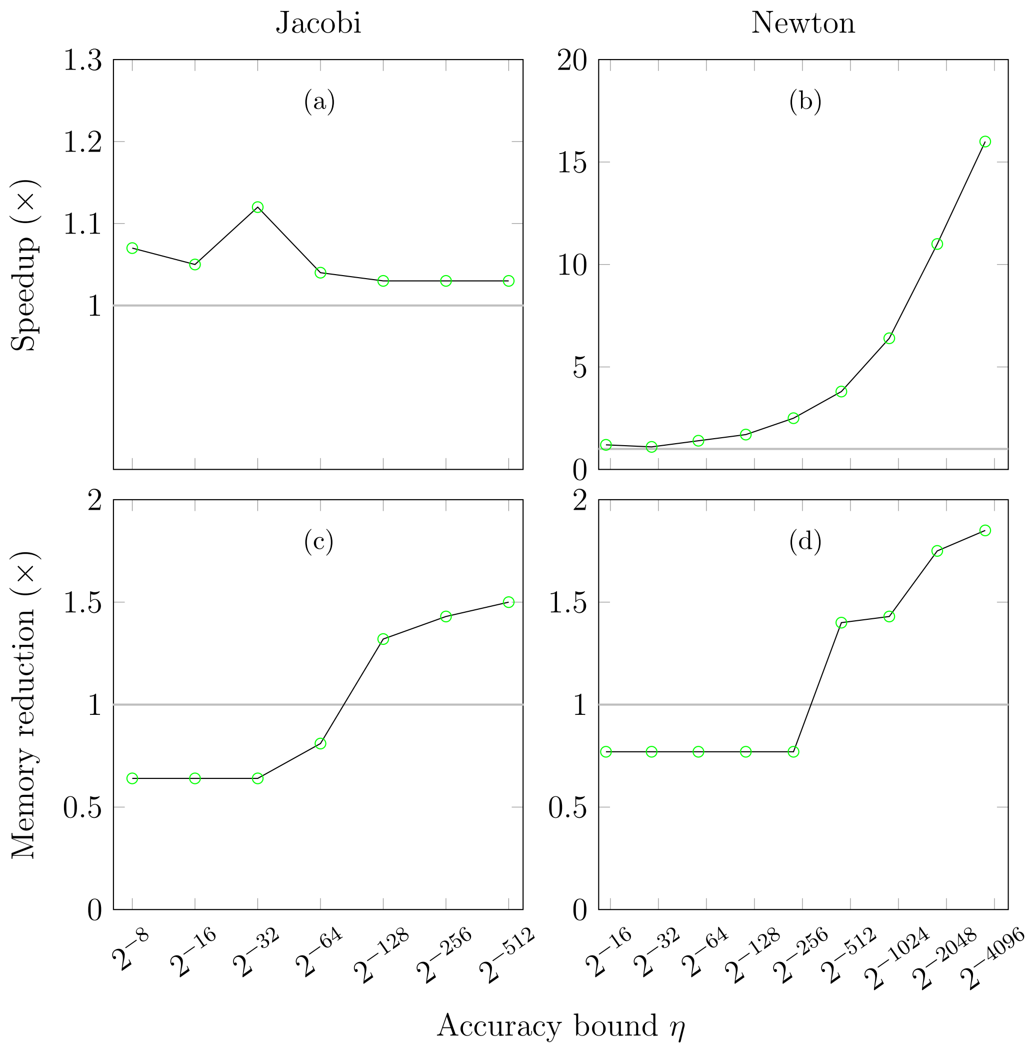


Fig. 5.5. Solve time speedup for the (a) Jacobi and (b) Newton's methods featuring don't-change digit elision versus ARCHITECT-I. (c) and (d) show the corresponding memory requirement reductions for Jacobi and Newton, respectively, facilitated through digit elision.

Area and Frequency Comparison

Further experiments were performed to investigate how ARCHITECT-III scales and performs in terms of area and maximum operating frequency $f_{\max}$ versus ARCHITECT-I.

For the largest case ($D = 2^{18}$) in Jacobi solvers, ARCHITECT-III consumes 1.1× more LUTs and 1.2× more FFs versus ARCHITECT-I. These factors for Newton's method with the same D are 1.5 and 1.4, respectively. The small increases in LUTs and FFs can be attributed to the runtime-detection and MSD-elision logic.

For the same reason, the critical path delays of ARCHITECT-III are higher than ARCHITECT-I's, resulting in a drop in $f_{\max}$ for both Jacobi and Newton implementations. $f_{\max}$ drops from 180 MHz to 135 MHz for the smallest ($D = 2^{10}$) Jacobi design implemented, and from 150 MHz to 120 MHz for the smallest ($D = 2^{10}$) Newton datapath.

5.3.4 Combining Don't-change and Don't-care Digit Elision

Given the presented don't-care elision for stationary iterative methods in Chapter 4, let us now take advantage of don't-change and don't-care digit elision to support the efficient generation of digits within an MSD-first iterative solver. This is named ARCHITECT-IIIa, with computation patterns such as that illustrated in Fig. 1.1d.

Control Logic

For every element of $\hat{\boldsymbol{x}}^{(k)}$, $\hat{x}_j^{(k)}$, three factors determine the number of its digits that must be calculated: (i) the number of digits of initial guess χ, (ii) the number of additional digits to calculate per approximant $\omega(k)$ indicated by the don't-care line's gradient (4.7) on page 70 and (iii) the number of MSDs that are guaranteed to be stable within approximant k, $\phi(k)$. χ and $\omega(k)$ have been described in Section 4.4.1. $\phi(k)$ is obtained through the comparison of successive approximants, as was discussed in Section 5.3.1. A counter is all that is required to implement this.

Combining these, we arrive at

$$\beta_j(k) = \begin{cases} \chi + \omega(k) & \text{if } k < 2 \\ \chi - \phi(k) + \omega(k) & \text{otherwise,} \end{cases}$$

where $\beta_j(k)$ reveals the number of digits to produce per approximant. The index of the first digit to produce within each approximant is given by $\phi(k)$.

I designed a parameterisable finite-state machine (FSM) to sequence digit propagation through a pipeline of online operators. Beginning with the consumption of approximant $k = 0$ (the user-supplied initial guess), it sweeps through $\beta_j(k)$ digits per approximant, incrementing k after each calculation. Once $k \geq 2$, don't-change digits start to be evaluated, shifting the start of each approximant's calculation by $\phi(k)$ digits away from the MSD.

From a user's perspective, no additional information needs to be supplied to take advantage of digit elision. The system to be solved, defined by $\boldsymbol{A}$ and $\boldsymbol{b}$, along with $\hat{\boldsymbol{x}}^{(0)}$, are all that is required.

Memory

To enable don't-change digit elision, we require access to *one* previously computed approximant. This is compared with the current approximant (k) to infer the number of don't-change digits in the yet-to-be-computed approximant $k+1$. It is therefore safe for us to overwrite approximant $k-2$ with approximant k. A single and flat memory is segmented into two halves: one for even approximants, and one for odd. Therefore, approximant k's memory bank is selected by simply evaluating $k \bmod 2$.

Output digit storage is not our only memory-related concern. Don't-change digit elision requires the ability to start computation from arbitrary digit indices, thus we must store the internal *residues* of earlier iterations' operations.

5.3.5 Performance Improvement

In order to compare the performance of ARCHITECT-IIIa against that of ARCHITECT-II with don't-care digit elision only, I used the Jacobi method and experimented with linear systems of the form, as was evaluated for ARCHITECT-II (Section 4.5),

$$\boldsymbol{A}_m = \begin{pmatrix} 1 & 1 - 2^{-m} \\ 1 - 2^{-m} & 1 \end{pmatrix}, \quad \boldsymbol{b} = \begin{pmatrix} b_0 \\ b_1 \end{pmatrix}, \quad \boldsymbol{x}^{(0)} = \boldsymbol{0}, \tag{5.1}$$

with b_0 and b_1 randomly selected from a uniform distribution in the range $[0, 1)$. I used the termination criteron $\|\boldsymbol{A}_m\boldsymbol{x} - \boldsymbol{b}\| < \eta$, with $\eta \in (0, 1]$. The conditioning of $\boldsymbol{A}_m$ is controlled via $m \geq 0$, and convergence is always guaranteed, since $\boldsymbol{A}_m$ is strictly diagonally dominant for all m.

Figure 5.6 shows how the requested accuracy, controlled by η, affects the solve time and the number of computed digits. We can see from Figure 5.6a that ARCHITECT-II requires increasingly more time to reach a solution than ARCHITECT-IIIa as the requested accuracy increases. In the case of high accuracy, such as when computing to $\eta = 2^{-256}$, ARCHITECT-IIIa is 4.1× faster than ARCHITECT-II. With low accuracy requirements, such as $\eta \leq 2^{-18}$, both designs consume similar solve time, even though ARCHITECT-IIIa is slightly slower than ARCHITECT-II. This makes sense since ARCHITECT-IIIa's $f_{\max}$ is slightly inferior principally due to reductions caused by the introduction of don't-change digit elision logic. From

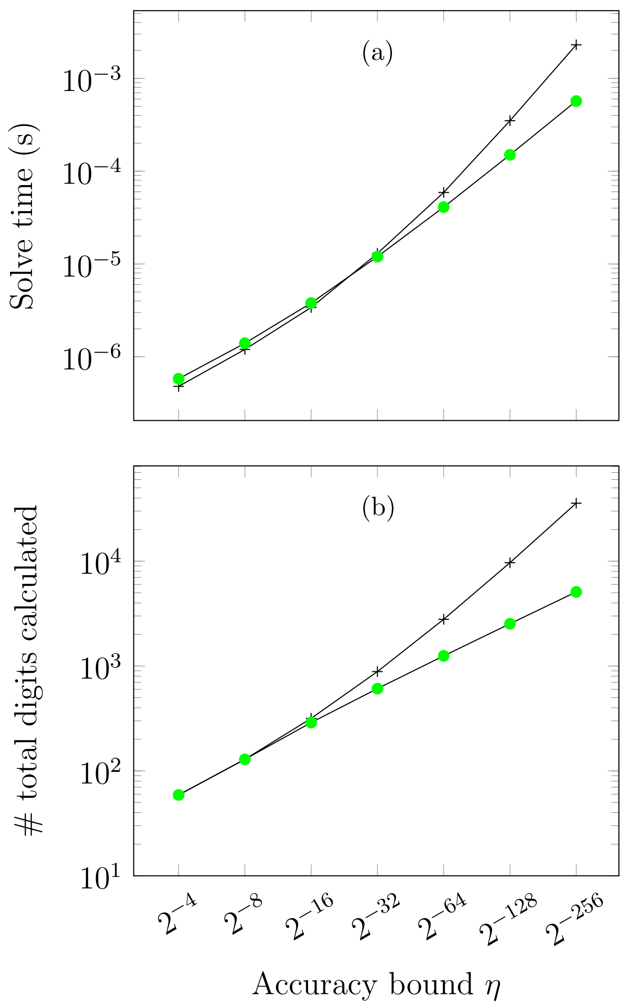


Fig. 5.6. How the requested accuracy bound η affects (a) the solve time, (b) the total number of digits calculated for ARCHITECT-II (—+—) and ARCHITECT-IIIa (—●—), with $m = 1$.

a more fundamental perspective, Figure 5.6b shows the relationship between the requested accuracy and the total number of digits calculated. ARCHITECT-IIIa calculates the same number of digits as ARCHITECT-II with $\eta = 2^{-4}$, increasing to 7.0× fewer digits when $\eta = 2^{-256}$.

The gap between the don't-change plus don't-care and don't-care-only lines widens as η reduces, indicating that consideration of don't-change digits becomes more important with higher accuracy requirements. This makes sense since, as η falls, more iterations are required to achieve convergence, thus affording more opportunity for advanced don't-change digit elision.

5.4 Specialised Don't-change Digit Elision for Stationary Iterative Methods

In the method-agnostic don't-change digit elision, given some D identical MSDs between approximants k and $k+1$, the number of identical MSDs that will also appear in approximant $k+2$ can be deduced. Since that number is always smaller than D, however, this technique is unfortunately unable to infer digit stability. If we concentrate on a particular class of algorithms, such as stationary iterative methods, the position of stable digits can be determined.

In this section, digit-stability analysis is presented for stationary iterative methods to avoid the recomputation of these stable digits. I demonstrated efficiency over iterative solvers using the generic don't-change digit elision, as will be discussed in Section 5.4.3.

Also enabling the inference of digit stability, Ercegovac's E-method produces the digits of its results from MSD first, one more per iteration [32], as exemplified in Fig. 5.7. The E-method, however, is a specialised Jacobi iteration and imposes strict conditions on its inputs: particularly a well conditioned iteration matrix $\boldsymbol{G}$ [32]. In contrast to the E-method, the proposed digit-stability analysis is applicable to any stationary iterative method and, as also shown in Fig. 5.7, holds for both well and ill-conditioned $\boldsymbol{G}$. With particularly well conditioned matrices, we can predict the generation of more than one stable digit per iteration.

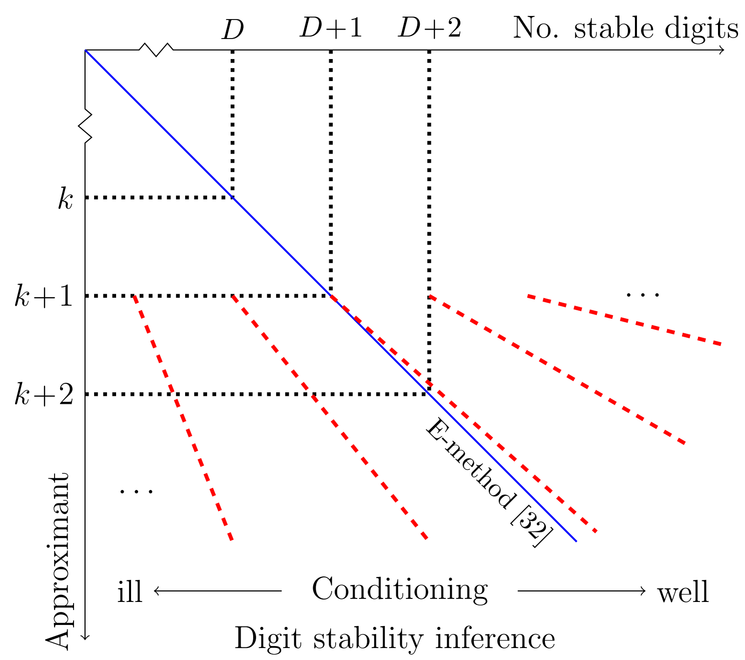


Fig. 5.7. A sketch of guaranteed digit stability. The E-method produces one new digit of lower significance per iteration; these, whose boundary is represented by the solid blue line, therefore remain stable across all future approximants. With knowledge that approximants k and $k+1$ share D identical MSDs, the proposed digit-stability analysis is able to infer the numbers of stable digits within the $k+1^{\text{th}}$ and all future approximants. As shown by the dashed red lines, these are dependent upon the conditioning of the iteration matrix $\boldsymbol{G}$.

5.4.1 Theoretical Analysis

Assume that a stationary iterative method is used to solve a linear system $\boldsymbol{Ax} = \boldsymbol{b}$. Further assume that the inequality $\|\boldsymbol{G}\|_\infty < 1$ holds for iteration matrix $\boldsymbol{G}$ [1]. If approximants to $\boldsymbol{x}^*$ are vectors of maximally redundant signed-digit numbers, knowledge of the number of identical MSDs in any two successive approximants $\hat{k}-1$ and $\hat{k}$ allows us to declare that subsets of MSDs in all approximants $k \geq \hat{k}$ will never change. The key steps in the derivation that follows are:

1. **Lemma 5.2**: If it is known that D MSDs of successive approximants' elements

[1]We adopt the infinity-norm in the analysis that follows since digit stability is ensured through bounds on worst-case perturbations of $\boldsymbol{x}^{(k)}$. In the common case of Hermitian matrices, $\rho(\boldsymbol{G}) = \|\boldsymbol{G}\|_2$ and $\|\boldsymbol{G}\|_2 \leq \|\boldsymbol{G}\|_\infty$, thus a bound on $\|\boldsymbol{G}\|_\infty$ corresponds to a bound on $\rho(\boldsymbol{G})$ [53]. Finally note that, although we present our analysis in a general setting, its application is intended for methods where $\|\boldsymbol{G}\|_\infty$ is readily computable, such as Jacobi.

are identical, we can bound the magnitude of the algorithm residue based on D and $\boldsymbol{G}$.

2. **Lemma 5.3**: Given a particular residue bound, we prove that a quantity of the current and future approximants' MSDs can never change.

3. **Theorem 5.1**: Bringing Lemmas 5.2 and 5.3 together, we infer the minimum number of permanently identical MSDs per approximant based on D and $\boldsymbol{G}$.

Let us begin by formally defining the meaning of digit stability within the approximants of an iterative algorithm.

Definition 5.1 (Digit stability). *The D MSDs of an approximant $\hat{k}$ are said to be stable iff*

$$x_i^{(k)} = x_i^{(\hat{k})} \quad \forall k > \hat{k},\ i \in \{1, 2, \cdots, D\}.$$

Our choice of number system means that we can append digits to a number x to form a new number, $\tilde{x}$, representing any value within a symmetric interval around x. We call such numbers *consistent* in the values they represent.

Definition 5.2 (Digit consistency). *Let x be a D-digit number represented using maximally redundant digits. Further let y be a second number, also maximally redundant, comprising any finite number of digits. y is said to be consistent with x iff*

$$y \in \left(x - r^{-D},\ x + r^{-D}\right).$$

Lemma 5.1 (Representation interval). *Let x be a maximally redundant D-digit number. If additional digits are appended to x to form a new number, $\tilde{x}$, then $\tilde{x}$ is consistent with x.*

Proof. By definition,

$$x = \sum_{i=1}^{D} x_i r^{-i}.$$

Since $\tilde{x}$ contains $\tilde{D} > D$ digits, with its D MSDs the same as those in x,

$$\begin{aligned}\tilde{x} &= \sum_{i=1}^{D} x_i r^{-i} + \sum_{i=D+1}^{\tilde{D}} \tilde{x}_i r^{-i} \\ &= x + \sum_{i=D+1}^{\tilde{D}} \tilde{x}_i r^{-i}.\end{aligned}$$

The digit extrema in a maximally redundant number system are $-(r-1)$ and $r-1$. We can thus deduce that

$$\begin{aligned}\tilde{x} &\in \left[x - \sum_{i=D+1}^{\tilde{D}} (r-1)\, r^{-i},\ x + \sum_{i=D+1}^{\tilde{D}} (r-1)\, r^{-i}\right] \\ &= \left[x - r^{-D} + r^{-\tilde{D}},\ x + r^{-D} - r^{-\tilde{D}}\right] \\ &\subset \left(x - r^{-D},\ x + r^{-D}\right),\end{aligned}$$

and so, per Definition 5.2, $\tilde{x}$ is consistent with x. □

Suppose now that we know—via runtime digit-by-digit comparison—that some D MSDs within successive approximants k and $k+1$ are identical. Given particular iteration matrix conditioning, we can bound the algorithm residue for approximant $k+1$.

Lemma 5.2 (Residue bound). *If the elements of $\boldsymbol{x}^{(k)}$ and $\boldsymbol{x}^{(k+1)}$ share a minimum of D identical MSDs, then*

$$\left\|\boldsymbol{s}^{(k+1)}\right\|_\infty < \frac{2\left\|\boldsymbol{G}\right\|_\infty}{1 - \left\|\boldsymbol{G}\right\|_\infty} r^{-D}.$$

Proof. Manipulation of (2.1) on page 13 allows us to deduce that

$$\begin{aligned}
\boldsymbol{M}\boldsymbol{x}^{(k+1)} &= \boldsymbol{N}\boldsymbol{x}^{(k)} + \boldsymbol{b} \\
\boldsymbol{M}\big(\boldsymbol{x}^{(k+1)} - \boldsymbol{x}^{(k)}\big) &= (\boldsymbol{N} - \boldsymbol{M})\,\boldsymbol{x}^{(k)} + \boldsymbol{A}\boldsymbol{x}^{*} \\
\boldsymbol{A}\boldsymbol{s}^{(k)} &= \boldsymbol{M}\big(\boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k+1)}\big).
\end{aligned}$$

Given that $\boldsymbol{A}^{-1} = \sum_{i=0}^{\infty} \boldsymbol{G}^{i}\boldsymbol{M}^{-1}$ [74], we therefore have

$$\boldsymbol{s}^{(k)} = \sum_{i=0}^{\infty} \boldsymbol{G}^{i}\big(\boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k+1)}\big).$$

Taking norms and recalling that $\|\boldsymbol{G}\|_{\infty} < 1$,

$$\begin{aligned}
\big\|\boldsymbol{s}^{(k)}\big\|_{\infty} &\leq \left\|\sum_{i=0}^{\infty} \boldsymbol{G}^{i}\right\|_{\infty} \big\|\boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k+1)}\big\|_{\infty} \\
&\leq \frac{1}{1 - \|\boldsymbol{G}\|_{\infty}} \big\|\boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k+1)}\big\|_{\infty}.
\end{aligned} \tag{5.2}$$

Let j be the index for which $x_j^{(k)}$ and $x_j^{(k+1)}$ are the successive elements sharing the fewest identical MSDs. We define the number of contiguous MSDs shared by the j^{th} elements as D. From Lemma 5.1 we know that

$$x_j^{(k)} \in \left(\sum_{i=1}^{D} x_{ji}^{(k)} r^{-i} - r^{-D},\ \sum_{i=1}^{D} x_{ji}^{(k)} r^{-i} + r^{-D}\right)$$

and

$$x_j^{(k+1)} \in \left(\sum_{i=1}^{D} x_{ji}^{(k+1)} r^{-i} - r^{-D},\ \sum_{i=1}^{D} x_{ji}^{(k+1)} r^{-i} + r^{-D}\right).$$

Since $x_{ji}^{(k)} = x_{ji}^{(k+1)}\ \forall i \in \{1, 2, \cdots, D\}$, we find that

$$\left|x_j^{(k)} - x_j^{(k+1)}\right| < 2r^{-D},$$

giving a bound on the vector norm of

$$\left\|\boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k+1)}\right\|_\infty < 2r^{-D}. \tag{5.3}$$

Transformation of (2.2) on page 13 reveals that

$$\begin{aligned}\boldsymbol{x}^{(k+1)} &= \boldsymbol{G}\boldsymbol{x}^{(k)} + \boldsymbol{M}^{-1}\boldsymbol{A}\boldsymbol{x}^* \\ &= \boldsymbol{G}\boldsymbol{x}^{(k)} + (\boldsymbol{I} - \boldsymbol{G})\,\boldsymbol{x}^* \\ \boldsymbol{s}^{(k+1)} &= \boldsymbol{G}\boldsymbol{s}^{(k)}.\end{aligned}$$

Taking norms,

$$\left\|\boldsymbol{s}^{(k+1)}\right\|_\infty \leq \|\boldsymbol{G}\|_\infty \left\|\boldsymbol{s}^{(k)}\right\|_\infty, \tag{5.4}$$

which, when combined with (5.2), results in

$$\left\|\boldsymbol{s}^{(k+1)}\right\|_\infty \leq \frac{\|\boldsymbol{G}\|_\infty}{1 - \|\boldsymbol{G}\|_\infty} \left\|\boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k+1)}\right\|_\infty.$$

Substitution of (5.3) then gives

$$\left\|\boldsymbol{s}^{(k+1)}\right\|_\infty < \frac{2\,\|\boldsymbol{G}\|_\infty}{1 - \|\boldsymbol{G}\|_\infty} r^{-D}. \tag{5.5}$$

□

Given a particular residue bound, our next task is to show that MSD stability is guaranteed within the current and future approximants.

Lemma 5.3 (Existence of digit stability)**.** *If the condition*

$$\left\|\boldsymbol{s}^{(\hat{k})}\right\|_\infty < r^{-D} \tag{5.6}$$

holds, then x_j^* is consistent with the $D-1$ MSDs of $x_j^{(k)}$ $\forall k \geq \hat{k}$ $\forall j$, and these MSDs are stable.

Proof. Convergence results on the algorithm ensure that there must exist an approximant $\hat{k}$ for which

$$x_j^* \in \left(x_j^{(\hat{k})} - r^{-D},\ x_j^{(\hat{k})} + r^{-D} \right) \quad \forall j.$$

From Lemma 5.1, we know that x_j^* is consistent with $x_j^{(\hat{k})}$.

Through repeated self-substitution of (5.4),

$$\left\| \boldsymbol{s}^{(k)} \right\|_\infty \leq \|\boldsymbol{G}\|_\infty^{k-\hat{k}} \left\| \boldsymbol{s}^{(\hat{k})} \right\|_\infty \tag{5.7}$$

which, given (5.6), means that

$$\left\| \boldsymbol{s}^{(k)} \right\|_\infty < \|\boldsymbol{G}\|_\infty^{k-\hat{k}} r^{-D}$$

and thus

$$\left| s_j^{(k)} \right| < \|\boldsymbol{G}\|_\infty^{k-\hat{k}} r^{-D} \quad \forall j.$$

For approximant k, therefore,

$$x_j^* \in I := \left(x_j^{(k)} - \|\boldsymbol{G}\|_\infty^{k-\hat{k}} r^{-D},\ x_j^{(k)} + \|\boldsymbol{G}\|_\infty^{k-\hat{k}} r^{-D} \right) \quad \forall j.$$

Let us consider how the perturbation of one or more of the $D-1$ MSDs in any approximant $k \geq \hat{k}$ would be inconsistent with the proof of convergence. Such a perturbation would produce a new interval, I'. If $x_j^* \notin I'$, such a new representation of x_j would invalidate the convergence, thus the $D-1$ MSDs of $x_j^{(k)}$ must be identical for all $k \geq \hat{k}$.

Consider an increase of the $D-1$th MSD by one unit, leading to a representation consistent with any value in

$$I' := \left(x_j^{(\hat{k})} - \|\boldsymbol{G}\|_\infty^{k-\hat{k}} r^{-D} + r^{-(D-1)}, x_j^{(\hat{k})} + \|\boldsymbol{G}\|_\infty^{k-\hat{k}} r^{-D} + r^{-(D-1)}\right) \quad \forall j.$$

Comparing the upper bound of I and the lower bound of I', we have

$$\begin{aligned} \min I' - \max I &= -2\, \|\boldsymbol{G}\|_\infty^{k-\hat{k}} r^{-D} + r^{-(D-1)} \\ &= \left(r - 2\, \|\boldsymbol{G}\|_\infty^{k-\hat{k}}\right) r^{-D}. \end{aligned} \tag{5.8}$$

Since $r \geq 2$ and $\|\boldsymbol{G}\|_\infty < 1$, (5.8) is strictly positive. This means that $I \cap I' = \emptyset$, and thus $x_j^* \notin I'$.

Clearly, a unit increase of *any* digit in $x_{ji}^{(k)}$ $\forall i \in \{1, 2, \cdots, D-1\}$ would lead to $I \cap I' = \emptyset$, violating the algorithm's convergence. A similar argument can be made for a unit decrease of $x_{ji}^{(k)}$ $\forall i \in \{1, 2, \cdots, D-1\}$. Thus, x_j^* is consistent with the $D-1$ MSDs of $x_j^{(k)}$ $\forall k \geq \hat{k}$ $\forall j$, and these MSDs are stable.

□

We are now able to bound the current and future iterations' residues and ensure that stable MSDs exist, but the relationship between these two features is currently missing. Combining Lemmas 5.2 and 5.3 will allow us to establish this, thereby providing a guaranteed minimum number of stable digits for the current and all future approximants.

Theorem 5.1 (Inference of digit stability)**.** *If $\boldsymbol{x}^{(\hat{k}-1)}$ and $\boldsymbol{x}^{(\hat{k})}$ share a minimum of D identical MSDs, then x_j^* is consistent with the $D + \left\lfloor \log_r \frac{1-\|\boldsymbol{G}\|_\infty}{2\|\boldsymbol{G}\|_\infty^{k-\hat{k}+1}} \right\rfloor - 1$ MSDs of*

$x_j^{(k)}$ $\forall k \geq \hat{k}$ $\forall j$, *and these MSDs are stable.*

Proof. Since the D MSDs of each element of approximants $\hat{k}-1$ and $\hat{k}$ are identical, we can apply Lemma 5.2 to approximants $\hat{k}-1$ and $\hat{k}$ to find that (5.5) holds for $\hat{k}$, *i.e.*

$$\left\| \boldsymbol{s}^{(\hat{k})} \right\|_\infty < \frac{2\left\|\boldsymbol{G}\right\|_\infty}{1-\left\|\boldsymbol{G}\right\|_\infty} r^{-D}.$$

Substituting this inequality into (5.7), we can deduce that

$$\begin{aligned}\left\| \boldsymbol{s}^{(k)} \right\|_\infty &< \left\|\boldsymbol{G}\right\|_\infty^{k-\hat{k}} \frac{2\left\|\boldsymbol{G}\right\|_\infty}{1-\left\|\boldsymbol{G}\right\|_\infty} r^{-D} \\ &= r^{-\left(D+\log_r \frac{1-\|\boldsymbol{G}\|_\infty}{2\|\boldsymbol{G}\|_\infty^{k-\hat{k}+1}}\right)} \\ &\leq r^{-\left(D+\left\lfloor \log_r \frac{1-\|\boldsymbol{G}\|_\infty}{2\|\boldsymbol{G}\|_\infty^{k-\hat{k}+1}} \right\rfloor\right)}.\end{aligned}$$

We can therefore apply Lemma 5.3 with this bound on $\left\| \boldsymbol{s}^{(\hat{k})} \right\|_\infty$, from which we are finally able to infer that x_j^* is consistent with the $D+\left\lfloor \log_r \frac{1-\|\boldsymbol{G}\|_\infty}{2\|\boldsymbol{G}\|_\infty^{k-\hat{k}+1}} \right\rfloor - 1$ MSDs of $x_j^{(k)}$ $\forall k \geq \hat{k}$ $\forall j$, and that those MSDs are stable. □

Examination of Theorem 5.1 allows us to understand the shapes of the stability regions seen in Fig. 5.7 for different $\|\boldsymbol{G}\|_\infty$. The relationship between the number of identical MSDs within approximants $\hat{k}-1$ and $\hat{k}$ and the quantity that stabilise by approximant $\hat{k}$ is controlled by $D+\left\lfloor \log_r \frac{1-\|\boldsymbol{G}\|_\infty}{2\|\boldsymbol{G}\|_\infty^{k-\hat{k}+1}} \right\rfloor - 1$ with $k=\hat{k}$, *i.e.* $D+\left\lfloor \log_r \frac{1-\|\boldsymbol{G}\|_\infty}{2\|\boldsymbol{G}\|_\infty} \right\rfloor - 1$. For the most well conditioned systems, *i.e.* those with low $\|\boldsymbol{G}\|_\infty$, $\log_r \frac{1-\|\boldsymbol{G}\|_\infty}{2\|\boldsymbol{G}\|_\infty}$ is more positive, while for particularly ill-conditioned systems it is more negative. This explains the leftward and rightward shifts present in Fig. 5.7 for high and low values of $\|\boldsymbol{G}\|_\infty$, respectively. The point at which D identical MSDs infer the presence of D stable digits within approximant $\hat{k}$ occurs when $\|\boldsymbol{G}\|_\infty = \frac{1}{2r+1}$. Beyond $\hat{k}$, we see a linear increase in $\log_r \frac{1-\|\boldsymbol{G}\|_\infty}{2\|\boldsymbol{G}\|_\infty^{k-\hat{k}+1}}$, and therefore in the number of stable digits, with k. This applies even for the most ill-conditioned systems; an increasing number of digits will therefore always stabilise over time.

5.4.2 Prototype Implementation

In order to evaluate the effectiveness of this digit-stability analysis, I built a hardware implementation based on the previous ARCHITECT-IIIa (Section 5.3.4), modified to allow the runtime inference, and subsequent avoidance of recalculation, of digits known to have *stabilised*. As the digits of approximant k are generated, their values are compared on-the-fly with those of previously generated approximant $k-1$, fetched from RAM. Once some $D > 0$ successive MSDs are found to be identical across all pairs of elements $\boldsymbol{x}_j^{(k-1)}$ and $\boldsymbol{x}_j^{(k)}$, we assign $\hat{k} \leftarrow k$ and, for all subsequent approximants, the generation of each approximant's first

$$\psi(k) = D + \left\lfloor \log_r \frac{1 - \|\boldsymbol{G}\|_\infty}{2\,\|\boldsymbol{G}\|_\infty^{k-\hat{k}+1}} \right\rfloor - 1$$

digits is skipped. Note that we do not need to calculate logarithms or perform exponentiation in hardware. Instead, we can use the more computationally efficient form

$$\psi(k) = D + \left\lfloor \alpha - \left(k - \hat{k} + 1\right)\beta \right\rfloor - 1, \tag{5.9}$$

where $\alpha = \log_r \frac{1-\|\boldsymbol{G}\|_\infty}{2}$ and $\beta = \log_r \|\boldsymbol{G}\|_\infty$ are constants that we precompute and feed in along with $\boldsymbol{A}$, $\boldsymbol{b}$ and $\boldsymbol{x}^{(0)}$.

Since the proposed digit-stability analysis is applicable to any stationary iterative methods, my prototype was a Jacobi method implementation, named ARCHITECT-IIIb. Like ARCHITECT-IIIa, ARCHITECT-IIIb excludes the don't-change and don't-care digits from calculation. By virtue of the novel digit-stability inference, ARCHITECT-IIIb can perform more efficiently against ARCHITECT-IIIa by skipping the calculation of MSDs known to have stabilised. The datapath of the Jacobi method is shown in Fig. 3.6a, which is identical in structure to that used in ARCHITECT-II and -IIIa, facilitating direct comparison.

5.4.3 Evaluation

There are three obvious comparison points for ARCHITECT-IIIb: ARCHITECT-IIIa with online delay-based MSD elision, the E-method [32] and the broad class of conventional, LSD-first iterative solvers. For the MSD-first methods, theoretical analysis (Section 5.4.3) was conducted to uncover the shortcomings of the prior art. Experiments (Section 5.4.3) were also performed to quantify the gains through the realisation of digit-stability analysis in hardware. For comparison against LSD-first arithmetic, I implemented datapaths composed of parallel-in, serial-out (PISO) operators of the same form that has been previously used to evaluate ARCHITECT-II and -IIIa. These operate in a similar digit-serial fashion, but require the compile-time determination of precision.

The hardware implementations all targetted a Xilinx Virtex UltraScale FPGA (part number XCVU190-FLGB2104-3-E) and were compiled using Vivado 16.4. All results obtained in hardware were verified against golden software models.

Theoretical Comparison

As was mentioned in Section 5.3.4, ARCHITECT-IIIa's MSD elision strategy is unable to infer the existence of stable digits. In the worst case, as shown in Table 5.1, we are forced to compute the values of δ *more* MSDs for *every* approximant, potentially wasting significant time and energy in doing so, when using this method. The hardware realisation of digit-stability analysis (*i.e.* ARCHITECT-IIIb) is actually simpler than its online delay-based predecessor (*i.e.* ARCHITECT-IIIa), leading to the multiple performance boosts I elaborate upon in Section 5.4.3. A benefit of online-delay based don't-change digit inference is its applicability to any iterative method.

The E-method, designed for the efficient evaluation of polynomial and rational functions, is the only existing work allowing the declaration of MSDs as stable across the approximants of an iterative algorithm [32]. Its MSD-first Jacobi solver produces one new, less-significant digit for each of the elements of its solution vector per iteration. To achieve this, the target linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ must fulfil a list of strict conditions. In particular: (i) $\|\boldsymbol{G}\|_\infty \leq {}^1\!/_{2r}$, *i.e.* a more restrictive requirement than strict diagonal dominance of $\boldsymbol{A}$, and (ii) $\|\boldsymbol{b}\|_\infty < 1$. (ii) is required since $\boldsymbol{b}$ forms the algorithm's initial internal residue, which must begin and remain bounded within $(-1, 1)^N$ in order to produce valid digits at each iteration.

As reflected in Table 5.1, digit-stability inference is far less restrictive than the E-method. The former holds for any stationary iterative method, while the E-method is a particular Jacobi implementation. Furthermore, we impose no restrictions upon the target system beyond $\|\boldsymbol{G}\|_\infty < 1$, meaning that users can realise the benefits of digit-stability even for very poorly conditioned matrices.

In order to achieve the same rate of stable MSD growth, solving (5.9) for $\beta = 1$ shows that the proposed analysis requires $\|\boldsymbol{G}\|_\infty = {}^1\!/_r$: double that for the E-method. This technique is thus able to achieve the E-method's growth rate for a wider range of differently conditioned matrices. With $\|\boldsymbol{G}\|_\infty < {}^1\!/_r$, we achieve a growth rate faster than the E-method's, while the opposite is true when $\|\boldsymbol{G}\|_\infty \in ({}^1\!/_r, 1)$. An advantage of the E-method over my proposals is that the former does not require knowledge of MSDs shared between approximants; the conditions enumerated above guarantee

Table 5.1
Properties of approaches for the inference of identical and stable MSDs in current and future approximants. To enable comparison, we assume that D MSDs of all elements of the most recently computed two approximants, $\hat{k}-1$ and $\hat{k}$, are known to be the same. For compactness, we abbreviate $\alpha = \log_r \frac{1-\|\boldsymbol{G}\|_\infty}{2}$ and $\beta = \log_r \|\boldsymbol{G}\|_\infty$ in the final row of the table.

Approach	Iterative method	Runtime detection	$\|\boldsymbol{G}\|_\infty$	$\|\boldsymbol{b}\|_\infty$	Guaranteed-stable digits in approximant $k \geq \hat{k}$	Guaranteed-identical digits between approximants k and $k+1$ $\forall k \geq \hat{k}$
E-method [32]	Jacobi	✗	$[0, 1/2r]$	$[0, 1)$	$D + k - \hat{k} + 1$	$D + k - \hat{k} + 1$
Online delay -based inference	Any	✓	—	$[0, \infty)$	0	$D - \delta\left(k - \hat{k} + 1\right)$
Digit-stability inference	Stationary	✓	$[0, 1)$	$[0, \infty)$	$D + \left\lfloor \alpha - \left(k - \hat{k} + 1\right)\beta \right\rfloor - 1$	$D + \left\lfloor \alpha - \left(k - \hat{k} + 1\right)\beta \right\rfloor - 1$

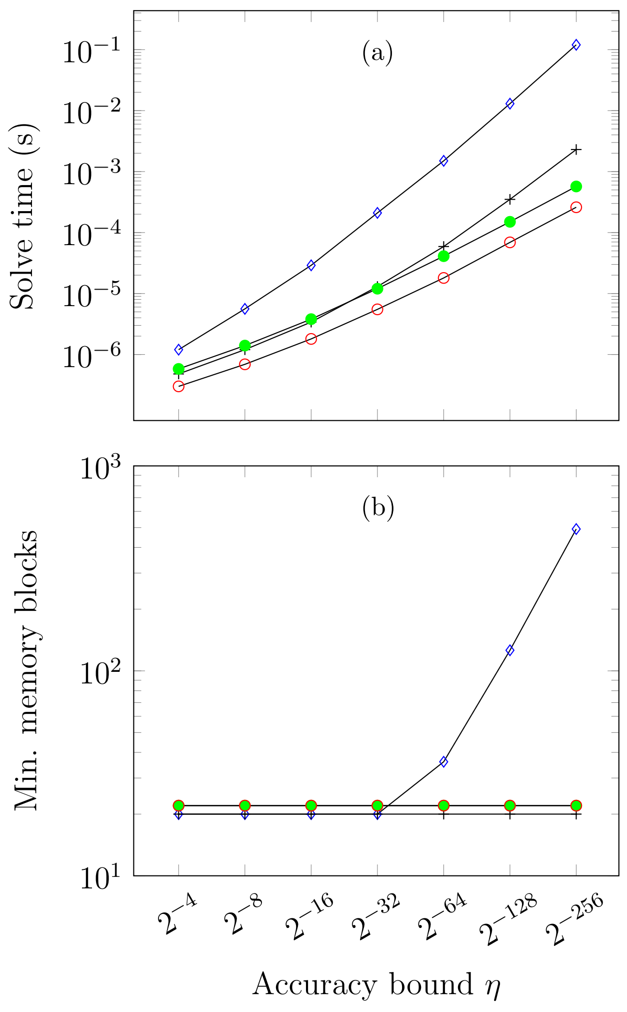


Fig. 5.8. How the requested accuracy bound η affects (a) solve time and (b) minimum memory requirement for ARCHITECT-I (—◊—), ARCHITECT-II (—+—), ARCHITECT-IIIa (—●—) and ARCHITECT-IIIb (—○—), with $m = 1$.

that digits will begin to stabilise immediately. However, it is trivial to implement logic to detect the existence of identical MSDs in successive approximants.

Empirical Evaluation

Experiments were conducted to evaluate the performance improvement of ARCHITECT-IIIb versus that of ARCHITECT-I, -II and IIIa, I used the Jacobi method, the same case study used in previous implementations, for direct comparison.

In Fig. 5.8, I demonstrate the scalability of the most efficient Jacobi solver *i.e.,* ARCHITECT-IIIb, featuring the elision of stable and don't-care digits. For these experiments, I fixed $m = 1$ in (5.1) and varied accuracy bound η. Fig. 5.8a shows that ARCHITECT-IIIb requires less time to reach a solution than ARCHITECT-I, ARCHITECT-II and ARCHITECT-IIIa for all the requested accuracies tested. When computing to $\eta = 2^{-256}$, ARCHITECT-IIIb achieves 470×, 8.8× and 2.2× speedups versus ARCHITECT-I, -II and -IIIa, respectively. With low accuracy requirements, such as when $\eta = 2^{-4}$, speedup factors are 4.0, 1.6 and 1.9, respectively.

In addition, the gaps between ARCHITECT-IIIa (─●─) and ARCHITECT-IIIb (─○─) and don't-care-only ARCHITECT-II (─+─) lines widen as η reduces. This indicates that the elision of don't-change digits achieves more solve-time savings, as η falls, since more iterations are required to compute.

For iterative solvers with both don't-change and don't-care optimisations enabled, ARCHITECT-IIIb achieves approximately constant solve time speedups over ARCHITECT-IIIa, showing the superiority of our digit-stability inference. Speedups ranged from 2.0× ($\eta = 2^{-4}$) to 2.2× ($\eta = 2^{-256}$). The saturation is due to properties of the arbitrary-precision arithmetic operators shared by both implementations, which require an increasing number of clock cycles to generate each digit as the significance of those digits decreases. As η falls, the increasing time per digit's generation begins to dominate the gains realised through the stable MSD elision.

Fig. 5.8b shows the minimum number of on-chip memory blocks that need to be instantiated in order for the arbitrary-precision iterative solvers to reach particular accuracies. The jaggedness in this plot is due to the granularity of the FPGA memory blocks, for which we only used whole numbers of BRAMs. For lower-accuracy cases ($\eta \geq 2^{-32}$), all of the designs require approximately the same amount of memory, although ARCHITECT-IIIa and -IIIb are slightly inferior due to don't-change detection overheads. The advantages of the memory addressing of ARCHITECT-IIIa and -IIIb explained in Section 5.3.4 come to the fore with higher accuracy requirements. For the lowest η tested, 2^{-256}, ARCHITECT-IIIa and -IIIb achieved a 22× memory reduction over ARCHITECT-I.

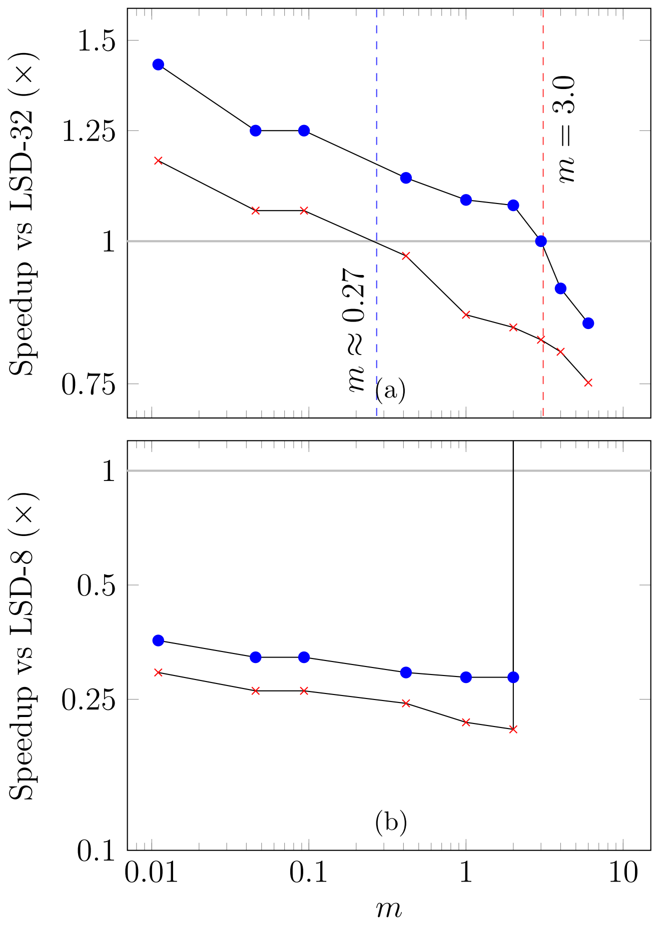


Fig. 5.9. How the conditioning of $\boldsymbol{A}_m$ affects the solve time of arbitrary-precision iterative solvers with both don't-change and don't-care digit elision, *i.e.* ARCHITECT-IIIa (—×—) and ARCHITECT-IIIb (—●—) *vs* LSD-first arithmetic with a fixed precision of (a) 32 and (b) 8 bits. ARCHITECT-IIIb computes more quickly than LSD-32 when $m < 3.0$, whereas ARCHITECT-IIIa can only beat LSD-32 when $m < 0.27$. (b) shows that both arbitrary-precision iterative solvers lead to an effectively infinite speedup when $m > 2$, since LSD-8 can never converge to accurate-enough results. While performance slowdowns were observed for $m \leq 2$, ARCHITECT-IIIb outperformed its predecessor in all cases, just as for LSD-32.

In Fig. 5.9a, we further compare our implementations against a Jacobi solver featuring LSD-first PISO arithmetic operators with a precision of 32 bits (LSD-32), a commonly encountered data width. For the solution of well conditioned linear systems—those with low m—LSD-32 has *over-budgeted* precision: results take longer to compute than had a lower precision been chosen instead. As a result, both ARCHITECT-IIIa and -IIIb compute more quickly than LSD-32 when $m < 0.27$. The benefits of our optimised MSD elision strategy become prominent with higher m. For $0.27 < m < 3.0$, ARCHITECT-IIIb outperforms its LSD-first competitor in terms of solve time, while ARCHITECT-IIIa does not.

Fig. 5.9b shows the results of the same experiments, but compared against an 8-bit LSD-first arithmetic implementation (LSD-8), also a commonly encountered data width [49]. Here, high m results in ill-conditioned systems, for which LSD-8 has *under-budgeted* precision. When $m > 2$, only our arbitrary-precision solvers can converge to results of great enough accuracy. In these cases, their performance speedups are effectively infinite. For $m \leq 2$, while both ARCHITECT-IIIa and -IIIb experience slowdowns *vs.* LSD-8, ARCHITECT-IIIb is faster than its competitor in all cases.

5.5 Conclusion

In this chapter, I proposed two don't-change digit elision methods. Firstly, an algorithm-agnostic don't-change digit elision was presented. Online arithmetic's digit dependencies were used to determine don't-change digits at runtime, allowing their recalculation in subsequent iterations and thereby increasing performance and decreasing memory footprints. I evaluated this technique on FPGAs using the Jacobi and Newton methods in order to verify its accuracy and establish its scalability and efficiency. These benchmarks showcased that iterative solvers with don't-change digit elision achieved up-to 16× performance speedups and 1.9× memory savings against ARCHITECT-I without any digit elision.

Even though online-delay based don't-change digit analysis is applicable to any iterative method, this technique is unable to guarantee that digits will stabilise.

To infer digit stability, a theorem specific to stationary iterative methods was presented to allow us to predict the rate of stable MSD growth across the approximants. With knowledge that some number of MSDs are common to two successive approximants, the proposed digit-stability analysis allows us to declare when, and which, MSDs in all future approximants will stabilise. The recomputation of these digits can thus be avoided, facilitating performance speedups. Unlike the E-method, this proposal holds, and is of benefit for linear systems of any conditioning.

As a classical stationary iterative method, the Jacobi method was used to evaluate the proposed digit-stability analysis. Combined with the don't-care digit elision, efficiency of ARCHITECT-IIIb was achieved over previous arbitrary-precision iterative solvers, with an up-to 470× speedup for the solution of a range of representative linear systems. Versus conventional arithmetic, I demonstrated ARCHITECT gains in cases where LSD-first solvers have precisions either too low (*e.g.* LSD-8) or too high (*e.g.* LSD-32) to suit the exemplary linear systems in this thesis.

In the future, I will extend our analysis to more iterative methods, including gradient descent and Krylov subspace methods. I foresee that MSD-first stochastic gradient descent with digit stability declaration would be of particular interest to the deep learning community. I am also keen to adapt my proposal to Newton's method, for which I expect to achieve substantial performance gains due to its quadratic convergence.

Chapter 6

Conclusion and Future Work

6.1 Summary of Contributions

Applications requiring arbitrary precision especially those featuring iterative algorithms that converge to solutions, are becoming increasingly popular. The precision of an iterative solver's datapath built using conventional arithmetic operators can be either overly high for the problem being solved or too low for the result to converge at all. To solve this problem, this thesis therefore asked:

- **Can a fixed-sized datapath be used to compute results to arbitrary precision while still achieving comparable performance versus conventional arithmetic equivalents?**

In Chapter 3, I proposed a constant-compute-resource hardware architecture named ARCHITECT that **iterates exactly** to arbitrary accuracies. I employed online arithmetic to start computation from MSD first and a Cantor pairing function for digit storage, allowing iteration count and precision to be increased simultaneously. A library of arbitrary-precision arithmetic operators—an ARCHITECT adder, multiplier and divider—was built to enable the construction of datapaths for different iterative methods. ARCHITECT was evaluated on FPGAs using the Jacobi and Newton methods in order to verify its accuracy and establish its scalability and suitability in numerical analysis. These benchmarks highlighted the key benefits of my approach: removing the burden of having to determine and fix the precision of arithmetic operators in advance. Experimental results revealed up-to 15× look-up table (LUT) and 31× flip-flop (FF) reductions over conventional parallel-in serial-out arithmetic, along with 75× LUT and 77× FF decreases versus the state-of-the-art online arithmetic implementation, when executing 100 Jacobi iterations. For Newton's method run for 10 iterations, these factors were 4.9, 8.7, 11 and 18, respectively.

The hardware presented in Chapter 3 achieves exact numeric computation by using online arithmetic to allow the refinement of results from earlier iterations over time, avoiding rounding error. However, this technique generates more LSDs than strictly necessary, since not all approximants contribute equally to an algorithm's overall error. I therefore asked:

- **Can the locations of unimportant LSDs in early approximants be inferred, excluding them from calculation to increase performance, while still obtaining arbitrary-accuracy results?**

In Chapter 4, I proposed the **don't-care digit elision** technique to address this question. I formalised the relationship between matrix conditioning and the number of digits required to be computed in each iteration via the forward error analysis. This technique is applicable to only stationary iterative methods which represent a large class of commonly used methods, such as Jacobi, Gauss-Seidel and successive over-relaxation (SOR). By analysing a bound on the rounding error that can be introduced in each approximant, I showed that a large number of don't-care digits can be elided while still allowing arbitrary accuracy to be obtained. First, a theorem for the optimal error distribution of approximants was presented. Forward error analysis was used to determine the number of additional LSDs to be computed per iteration. A second theorem was presented to confirm that arbitrary-accuracy solutions will still be achieved when eliding don't-care digits. This proposal was evaluated using the Jacobi method. For the solution of a set of representative linear quations, elision of don't-care digit calculation led to a geometric mean 21× speedup versus my vanilla arbitrary-precision solvers. Knowledge of don't-care digit locations allows us to eliminate the need to revisit and refine earlier approximants, leading to an up-to 25× memory footprint reduction.

Along with insignificant LSDs, in numerical iterative computation, later approximants generally feature stable MSDs. When using nonredundant number systems, cases arise where low-magnitude perturbations cause large numbers of digits to change between iterations via carry propagation. By introducing redundancy into our number representation, we can prevent the occurrence of such a scenario. LSDs are able to correct errors introduced in digits of higher significance, further allowing those MSDs to be declared as don't-change digits. This observation led to the third research question I aimed to address:

- **How is it possible to avoid recomputing digits of later approximants that are identical to those of previous approximants?**

In Chapter 5, two **don't-change digit elision** techniques were proposed. To begin, properties of online arithmetic were employed to allow the generation of those digits to be skipped in any iterative method, increasing efficiency. Versus my vanilla arbitrary-precision iterative solvers, I achieved up-to 16× performance speedups and 1.9× memory savings for the Jacobi and Newton benchmarks. A benefit of this proposal is its applicability to any iterative method, however, it is unfortunately unable to infer digit stability.

Use of interval and forward error analyses allows us to prove that digits of high significance will become stable when computing the approximants of systems of linear equations using stationary iterative methods. Unlike the E-method [32], the proposed digit-stability analysis holds for any stationary iterative method, and is of benefit to linear systems of any conditioning. I demonstrated efficiency over previous ARCHITECT and conventional (LSD-first) arithmetic implementations using a Jacobi solver with don't-care and stable digit elision. Versus the Jacobi solver using don't-care and the generic don't-change digit elision, I achieved speedups of 2.0-2.2× for the solution of a range of representative linear systems. Versus LSD-first solvers with over-budgeted precisions, my proposal was competitive when solving well conditioned matrices. Against LSD-first solvers with under-budgeted precisions, my proposal was competitive when solving ill-conditioned systems.

6.2 Future Work

I have identified several interesting directions for future work based on that conducted for this thesis.

(i) **Don't-care and digit-stability analyses for other iterative algorithms.** Since the present don't-care and digit-stability analyses are algorithm-specific, further analysis is required to similarly optimise other iterative algorithms. For example, I expect that arbitrary-precision stochastic gradient descent with don't-care and stable digit elision would be beneficial for neural network training. Newton's method with both optimisations enabled is expected to enable substantial performance gains due to its quadratic convergence.

(ii) **ARCHITECT in ASIC.** While I prototyped designs on FPGAs owing to the costs and lead times associated with full-custom implementation, I note that these devices are optimised for the implementation of conventional arithmetic operators. In particular, FPGAs' hardened carry chains suit the construction of fast LSD-first adders. At present, ARCHITECT cannot take advantage of such structures. I am confident that, should ARCHITECT see application-specific integrated circuit (ASIC) implementation, however, more competitive performance would be achievable.

(iii) **High-radix ARCHITECT.** Higher-radix ($r > 2$) online arithmetic could instead (or additionally) be employed to exploit high-performance adders, including on FPGAs, which I anticipate would lead to further speedups.

(iv) **Don't-care digit elision: time budget versus digit budget.** Given Theorem 4.1, don't-care digits can be excluded from iterative calculations. However, the generation time of each digit is different: the lower the significance of a given digit, the longer it takes to be computed. A time-minimising digit trajectory may be more beneficial than a digit-minimising one. Therefore, it would be interesting to minimise time for generation of all non-elided digits by changing the formulation of the proposed theoretical analysis.

(v) **Floating-point ARCHITECT.** The implementations presented and evaluated in this thesis utilise fixed-point arithmetic. ARCHITECT's principles are, however, generic, and can be employed for the construction of floating-point operators supporting arbitrary-precision mantissas. Arbitrary-precision computing using floating-point number representations would open fruitful research opportunities that deserve future investigations.

(vi) **High-level synthesis of ARCHITECT datapaths.** Finally, I envisage that the arbitrary-precision computation enabled by ARCHITECT could be combined with high-level synthesis (HLS) to enable faster hardware specialisation. Although FPGAs are suitable for the implementation of custom-precision datapaths, HLS tools only support finite-precision fixed-point or floating-point data types. It would be interesting to present a solution for arbitrary-precision arithmetic in HLS, allowing the use of arbitrary precisions for input arguments and return values.

6.3 Final Remarks

Through the work presented in this thesis, I have achieved significant advancement in the combination of online arithmetic and iterative algorithms to enable efficient arbitrary-precision iterative calculation in hardware. Several novel techniques for the elision of don't-care and don't-change digits were introduced for iterative calculations through the use of forward error analysis and redundant number systems. I demonstrated that hardware for arbitrary-precision iterative numerical algorithms can exhibit efficiency over traditional arithmetic equivalents where the latter's precisions are either under- or over-budgeted for the problem at hand. The contributions of this thesis can open up possibilities to conduct new avenues of research in digital computing, such as more advanced digit elision techniques, arbitrary-precision floating-point arithmetic and these combined with high-level synthesis.

Bibliography

[1] P. Adharapurapu and M. D. Ercegovac. A Composite Arithmetic Scheme for Evaluation of Multinomials. In *IEEE Asilomar Conference on Signals, Systems and Computers*, 2004.

[2] S. Amanollahi and G. Jaberipur. Energy-Efficient VLSI Realization of Binary64 Division with Redundant Number Systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(3), 2016.

[3] D. E. Atkins. Introduction to The Role of Redundancy in Computer Arithmetic. *IEEE Computer*, 8(6), 1975.

[4] D. H. Bailey. A Thread-safe Arbitrary Precision Computation Package. Technical report, `http://www.davidhbailey.com/dhbsoftware`, 2017.

[5] D. H. Bailey, R. Barrio, and J. Borwein. High-precision Computation: Mathematical Physics and Dynamics. *Applied Mathematics Computation*, 218(20), 2012.

[6] D. H. Bailey and J. M. Borwein. High-precision Arithmetic in Mathematical Physics. *Mathematics*, 3(2), 2015.

[7] D. H. Bailey, H. Yozo, X. S. Li, and B. Thompson. ARPREC: An Arbitrary Precision Computation Package. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2002.

[8] R. Barrett, M. W. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for The Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, 1994.

[9] A. Bocco, Y. Durand, and F. De Dinechin. SMURF: Scalar Multiple-precision Unum Risc-V Floating-point Accelerator for Scientific Computing. In *ACM Proceedings of the Conference for Next Generation Arithmetic*, 2019.

[10] D. Boland and G. A. Constantinides. An FPGA-based Implementation of the MINRES Algorithm. In *IEEE International Conference on Field Programmable Logic and Applications*, 2008.

[11] S. Boyd and L. Vandenberghe. *Convex Optimization.* Cambridge University Press, 2004.

[12] R. Brackert, M. D. Ercegovac, and A. Willson. Design of An On-line Multiply-add Module for Recursive Digital Filters. In *IEEE Symposium on Computer Arithmetic*, 1989.

[13] N. Brisebarre, S. Chevillard, M. D. Ercegovac, J.-M. Muller, and S. Torres. An Efficient Method for Evaluating Polynomial and Rational Function Approximations. In *IEEE International Conference on Application-Specific Systems, Architectures and Processors*, 2008.

[14] N. Brisebarre, G. Constantinides, M. Ercezovac, S.-I. Filip, M. Istoan, and J.-M. Muller. A High Throughput Polynomial and Rational Function Approximations Evaluator. In *IEEE Symposium on Computer Arithmetic*, 2018.

[15] N. Brisebarre and J.-M. Muller. Functions Approximable by E-fractions. In *IEEE Asilomar Conference on Signals, Systems and Computers*, 2004.

[16] A. M. Bruaset. *A Survey of Preconditioned Iterative Methods*, volume 328. CRC Press, 1995.

[17] J. D. Bruguera. Radix-64 Floating-Point Divider. In *IEEE Symposium on Computer Arithmetic*, 2018.

[18] J. D. Bruguera. Low Latency Floating-Point Division and Square Root Unit. *IEEE Transactions on Computers*, 2019.

[19] M. Brzicová, C. Frougny, E. Pelantová, and M. Svobodová. On-line Algorithms for Multiplication and Division in Real and Complex Numeration Systems. *arXiv:1610.08309*, 2016.

[20] N. Burgess. Radix-2 SRT Division Algorithm with Simple Quotient Digit Selection. *IET Electronics Letters*, 27(21), 1991.

[21] A. Burton. Newton's Method and Fractals. *Technical manuscript, Whitman College*, 2009.

[22] P. Cégielski and D. Richard. Decidability of The Theory of The Natural Integers with The Cantor Pairing Function and The Successor. *Theoretical Computer Science*, 257(1-2), 2001.

[23] E. Cheney and D. Kincaid. *Numerical Mathematics and Computing.* Nelson Education, 2012.

[24] E. W. Cheney and D. R. Kincaid. *Numerical Mathematics and Computing.* Cengage Learning, 2012.

[25] A. H.-D. Cheng. Multiquadric and its Shape Parameter—A Numerical Investigation of Error Estimate, Condition Number, and Round-off Error by Arbitrary Precision Computation. *Engineering Analysis with Boundary Elements*, 36(2), 2012.

[26] A. Cordero, J. R. Torregrosa, and M. P. Vassileva. Design, Analysis, and Applications of Iterative Methods for Solving Nonlinear Systems. *Nonlinear Systems: Design, Analysis, Estimation and Control*, 2016.

[27] X. Cui, W. Liu, X. Chen, E. E. Swartzlander, and F. Lombardi. A Modified Partial Product Generator for Redundant Binary Multipliers. *IEEE Transactions on Computers*, 65(4), 2015.

[28] F. de Dinechin and B. Pasca. Designing Custom Arithmetic Data Paths with FloPoCo. *IEEE Design & Test of Computers*, 28(4), 2011.

[29] J. E. Dennis, Jr and J. J. Moré. Quasi-Newton Methods, Motivation and Theory. *SIAM review*, 19(1), 1977.

[30] A. Enge. MPC-A Library for Multiprecision Complex Arithmetic with Exact Rounding. `http://mpc.multiprecision.org/`, 2010.

[31] M. D. Ercegovac. A General Method for Evaluation of Functions and Computations in A Digital Computer. University of Illinois at Urbana-Champaign, 1975.

[32] M. D. Ercegovac. A General Hardware-Oriented Method for Evaluation of Functions and Computations in a Digital Computer. *IEEE Transactions on Computers*, C-26(7), 1977.

[33] M. D. Ercegovac. On Digit-by-digit Methods for Computing Certain Functions. In *IEEE Asilomar Conference on Signals, Systems and Computers*, 2007.

[34] M. D. Ercegovac. On Left-to-right Arithmetic. In *IEEE Asilomar Conference on Signals, Systems, and Computers*, 2017.

[35] M. D. Ercegovac and T. Lang. On-the-fly Conversion of Redundant into Conventional Representations. *IEEE Transactions on Computers*, C-36(7), 1987.

[36] M. D. Ercegovac and T. Lang. Most-significant-digit-first and On-line Arithmetic Approaches for The Design of Recursive Filters. In *IEEE Asilomar Conference on Signals, Systems and Computers*, 1989.

[37] M. D. Ercegovac and T. Lang. *Digital Arithmetic.* Elsevier, 2004.

[38] M. D. Ercegovac, J.-M. Muller, and A. Tisserand. FPGA Implementation of Polynomial Evaluation Algorithms. In *Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing*, volume 2607. International Society for Optics and Photonics, 1995.

[39] N. Fabiano, J.-M. Muller, and J. Picot. Algorithms for Triple-word Arithmetic. *IEEE Transactions on Computers*, 68(11), 2019.

[40] C. Fan, Y. Niu, G. Shi, F. Li, F. Qi, X. Xie, and D. Jiao. An Improved Signed Digit Representation Approach for Constant Vector Multiplication. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 63(10), 2016.

[41] X. Fang and M. Leeser. Open-source Variable-precision Floating-point Library for Major Commercial FPGAs. *ACM Transactions on Reconfigurable Technology and Systems*, 9(3), 2016.

[42] J. S. Fernando and M. D. Ercegovac. On-line Arithmetic Modules for Recursive Digital Filters. In *IEEE Asilomar Conference on Signals, Systems & Computers*, 1992.

[43] G. Constantinides and A. Kinsman and N. Nicolici. Numerical Data Representations for FPGA-based Scientific Computing. *IEEE Design & Test of Computers*, 28(4), 2011.

[44] R. Galli and A. F. Tenca. Design and Evaluation of Online Arithmetic for Signal Processing Applications on FPGAs. *Advanced Signal Processing Algorithms, Architectures, and Implementations XI*, 4474, 2001.

[45] A. F. González and P. Mazumder. Redundant Arithmetic, Algorithms and Implementations. *Integration*, 30(1), 2000.

[46] J. L. Gustafson. *The End of Error: Unum Computing.* Chapman and Hall/CRC, 2017.

[47] L. A. Hageman and D. M. Young. *Applied Iterative Methods.* Courier Corporation, 2012.

[48] R. Hamill, J. V. McCanny, and R. L. Walke. Online CORDIC Algorithm and VLSI Architecture for Implementing QR-array Processors. *IEEE Transactions on Signal Processing*, 48(2), 2000.

[49] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, et al. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017.

[50] D. L. Harris, S. F. Oberman, and M. A. Horowitz. SRT Division Architectures and Implementations. In *IEEE Sympsoium on Computer Arithmetic*, 1997.

[51] N. J. Higham. *Accuracy and Stability of Numerical Algorithms.* SIAM, 2002.

[52] N. J. Higham and P. A. Knight. Componentwise Error Analysis for Stationary Iterative Methods. In *Linear Algebra, Markov Chains, and Queueing Models.* Springer, 1993.

[53] R. Horn and C. Johnson. *Matrix Analysis.* Cambridge University Press, 1985.

[54] M. Iştoan and B. Pasca. Flexible Fixed-point Function Generation for FPGAs. In *IEEE Symposium on Computer Arithmetic*, 2017.

[55] G. Jaberipur. *Arithmetic Circuits for DSP Applications.* Wiley Online Library, 2017.

[56] M. Jaiswal and H. So. Area-Efficient Architecture for Dual-Mode Double Precision Floating Point Division. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 64(2), 2017.

[57] K. Javeed, X. Wang, and M. Scott. Serial and Parallel Interleaved Modular Multipliers on FPGA Platform. In *IEEE International Conference on Field Programmable Logic and Applications*, 2015.

[58] W. K. Jenkins, M. A. Soderstrand, and C. Radhakrishnan. Historical Patterns of Emerging Residue Number System Technologies During The Evolution of Computer Engineering and Digital Signal Processing. In *IEEE Symposium on Circuits and Systems*, 2018.

[59] F. Johansson. Arb: Efficient Arbitrary-precision Midpoint-radius Interval Arithmetic. *IEEE Transactions on Computers*, 66(8), 2017.

[60] M. Joldes, J.-M. Muller, and V. Popescu. On The Computation of The Reciprocal of Floating Point Expansions Using An Adapted Newton-Raphson

Iteration. In *IEEE International Conference on Application-Specific Systems, Architectures and Processors*, 2014.

[61] G. B. Joseph and R. Devanathan. Design and Analysis of Online Arithmetic Operators for Streaming Data in FPGAs. *International Journal of Applied Engineering Research*, 11(3), 2016.

[62] G. B. Joseph and R. Devanathan. Algorithms for Multiplierless Multiple Constant Multiplication in Online Arithmetic. *Circuits, Systems, and Signal Processing*, 37(11), 2018.

[63] B. Jovanovic, R. Jevtic, and C. Carreras. Binary Division Power Models for High-level Power Estimation of FPGA-based DSP Circuits. *IEEE Transactions on Industrial Informatics*, 10(1), 2014.

[64] A. Kaivani and S. Ko. Floating-point Butterfly Architecture Based on Binary Signed-digit Representation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(3), 2015.

[65] C. T. Kelley. *Iterative Methods for Linear and Nonlinear Equations.* SIAM, 1995.

[66] C. T. Kelley. *Iterative Methods for Optimization.* SIAM, 1999.

[67] W. Koh, R. Ahmad, S. Jaaman, and J. Sulaiman. Pricing Asian Option by Solving Black-Scholes PDE using Gauss-Seidel Method. In *International Conference on Computing, Mathematics and Statistics*, 2019.

[68] A. Landy and G. Stitt. Revisiting Serial Arithmetic: A Performance and Tradeoff Analysis for Parallel Applications on Modern FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2015.

[69] K. Lange. *Optimization.* Springer, 2004.

[70] D. Lau, A. Schneider, M. D. Ercegovac, and J. Villasenor. FPGA-based Structures for On-line FFT and DCT. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, 1999.

[71] D. Lau, A. Schneider, M. D. Ercegovac, and J. Villasenor. A FPGA-based Library for On-line Signal Processing. *Journal of VLSI signal processing systems for signal, image and video technology*, 28(1-2), 2001.

[72] Y. LeCun, Y. Bengio, and G. Hinton. Deep Learning. *Nature*, 521(7553), 2015.

[73] H. Li, J. J. Davis, J. Wickerson, and G. A. Constantinides. ARCHITECT: Arbitrary-precision Constant-hardware Iterative Compute. In *IEEE International Conference on Field Programmable Technology*, 2017.

[74] H. Li, J. J. Davis, J. Wickerson, and G. A. Constantinides. Digit Elision for Arbitrary-accuracy Iterative Computation. In *IEEE Symposium on Computer Arithmetic*, 2018.

[75] H. Li, J. J. Davis, J. Wickerson, and G. A. Constantinides. ARCHITECT: Arbitrary-precision Hardware with Digit Elision for Efficient Iterative Compute. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2019.

[76] J. Liu and Q. Liu. Speed and Resource Optimization of BFGS Quasi-Newton Implementation on FPGA Using Inexact Line Search Method for Neural Network Training. In *IEEE International Conference on Field-Programmable Technology*, 2018.

[77] Q. Liu, J. Liu, R. Sang, J. Li, T. Zhang, and Q. Zhang. Fast Neural Network Training on FPGA Using Quasi-newton Optimization Method. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(8), 2018.

[78] Q. Liu, R. Sang, and Q. Zhang. FPGA-based Acceleration of Davidon-Fletcher-Powell Quasi-Newton Optimization Method. *Transactions of Tianjin University*, 22(5), 2016.

[79] S. Liu, G. Mingas, and C.-S. Bouganis. An Unbiased MCM FPGA-based Accelerator in The Land of Custom Precision Arithmetic. *IEEE Transactions on Computers*, 66(5), 2016.

[80] W. Liu, J. Li, T. Xu, C. Wang, P. Montuschi, and F. Lombardi. Combining Restoring Array and Logarithmic Dividers into An Approximate Hybrid Design. In *IEEE Symposium on Computer Arithmetic*, 2018.

[81] M. Lu. *Arithmetic and Logic in Computer Systems.* Wiley Online Library, 2004.

[82] M. Macklin, K. Erleben, M. Müller, N. Chentanez, S. Jeschke, and V. Makoviychuk. Non-smooth Newton Methods for Deformable Multi-body Dynamics. *ACM Transactions on Graphics*, 38(5), 2019.

[83] T. A. Manteuffel, J. Ruge, and B. S. Southworth. Nonsymmetric Algebraic Multigrid Based on Local Approximate Ideal Restriction (ℓ AIR). *SIAM Journal on Scientific Computing*, 40(6), 2018.

[84] R. McIlhenny and M. D. Ercegovac. On The Design of An On-line FFT Network for FPGA's. In *IEEE Asilomar Conference on Signals, Systems, and Computers*, 1999.

[85] M. R. Meher, C. C. Jong, and C.-H. Chang. A High Bit Rate Serial-serial Multiplier with On-the-fly Accumulation by Asynchronous Counters. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 19(10), 2011.

[86] Michele Benzi and Thomas M. Evans and Steven P. Hamilton and Massimiliano Lupo Pasini and Stuart R. Slattery. Analysis of Monte Carlo-accelerated Iterative Methods for Sparse Linear Systems. *Numerical Linear Algebra with Applications*, 24(3), 2017.

[87] E. K. Miller. A Computational Study of the Effect of Matrix Size and Type, Condition Number, Coefficient Accuracy and Computation Precision on Matrix-solution Accuracy. In *IEEE Antennas and Propagation Society International Symposium*, 1995.

[88] E. Mosanya and E. Sanchez. A FPGA-based Hardware Implementation of Generalized Profile Search Using Online Arithmetic. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 1999.

[89] MPFR. The GNU MPFR Library. `http://www.mpfr.org`, 2017.

[90] J.-M. Muller. Some Characterizations of Functions Computable in On-line Arithmetic. *IEEE Transactions on Computers*, 43(6), 1994.

[91] M. Muresan. *A Concrete Approach to Classical Analysis.* Springer, 2009.

[92] W. G. Natter and B. Nowrouzian. Digit-serial Online Arithmetic for High-speed Digital Signal Processing Applications. In *IEEE Asilomar Conference on Signals, Systems and Computers*, 2001.

[93] G. Ndour, T. T. Jost, A. Molnos, Y. Durand, and A. Tisserand. Evaluation of Variable Bit-width Units in A RISC-V Processor for Approximate Computing. In *ACM International Conference on Computing Frontiers*, 2019.

[94] Y. Nie, Y. Shen, Q. Chen, and Y. Xiao. Hybrid-Precision Arithmetic for Numerical Orbit Integration towards Future Satellite Gravimetry Missions. *Advances in Space Research*, 2020.

[95] S. F. Obermann and M. J. Flynn. Division Algorithms and Implementations. *IEEE Transactions on Computers*, 46(8), 1997.

[96] R. M. Owens. Compound Algorithms for Digit Online Arithmetic. In *IEEE Symposium on Computer Arithmetic*, 1981.

[97] R. M. Owens. Techniques to Reduce The Inherent Limitations of Fully Digit On-line Arithmetic. *IEEE Transactions on Computers*, 100(4), 1983.

[98] S. Rajagopal and J. R. Cavallaro. On-line Arithmetic for Detection in Digital Communication Receivers. In *IEEE Symposium on Computer Arithmetic*, 2001.

[99] S. Rajagopal and J. R. Cavallaro. Truncated Online Arithmetic with Applications to Communication Systems. *IEEE Transactions on Computers*, 55(10), 2006.

[100] N. Revol and F. Rouillier. Motivations for An Arbitrary Precision Interval Arithmetic and The MPFI Library. *Reliable computing*, 11(4), 2005.

[101] A. Roldao-Lopes, A. Shahzad, G. A. Constantinides, and E. C. Kerrigan. More Flops or More Precision? Accuracy Parameterizable Linear Equation Solvers for Model Predictive Control. In *IEEE Symposium on Field Programmable Custom Computing Machines*, 2009.

[102] K. Shi. Design of Approximate Overclocked Datapath. Imperial College London, 2015.

[103] K. Shi, D. Boland, and G. A. Constantinides. Accuracy-Performance Tradeoffs on An FPGA through Overclocking. In *IEEE International Symposium on Field-programmable Custom Computing Machines*, 2013.

[104] K. Shi, D. Boland, and G. A. Constantinides. Efficient FPGA Implementation of Digit Parallel Online Arithmetic Operators. In *IEEE International Conference on Field Programmable Technology*, 2014.

[105] K. Shi, D. Boland, E. Stott, S. Bayliss, and G. A. Constantinides. Datapath Synthesis for Overclocking: Online Arithmetic for Latency-accuracy Trade-offs. In *ACM/EDAC/IEEE Design Automation Conference*, 2014.

[106] A. Skaf, M. Ezzadeen, M. Benabdenbi, and L. Fesquet. On-Line Adjustable Precision Computing. In *IEEE International Conference on Design & Technology of Integrated Systems Iin Nanoscale Era*, 2019.

[107] J. Sun, G. D. Peterson, and O. O. Storaasli. High-performance Mixed-precision Linear Solver for FPGAs. *IEEE Transactions on Computers*, 57(12), 2008.

[108] N. Takagi, T. Asada, and S. Yajima. Redundant CORDIC Methods with A Constant Scale Factor for Sine and Cosine Computation. *IEEE Transactions on Computers*, 40(9), 1991.

[109] A. Tenca and M. Ercegovac. A High-Radix Multiplier Design for Variable Long-precision Computations. In *IEEE Asilomar Conference on Signals, Systems and Computers*, 1997.

[110] S. Timarchi, N. Akbarzadeh, and A. A. Hamidi. Maximally Redundant High-radix Signed-digit Residue Number System. In *IEEE International Symposium on Computer Architecture and Digital Systems*, 2015.

[111] K. S. Trivedi and M. D. Ercegovac. On-line Algorithms for Division and Multiplication. *IEEE Transactions on Computers*, 100(7), 2006.

[112] P. Tu and M. D. Ercegovac. Design of On-line Division Unit. In *IEEE Symposium on Computer arithmetic*, 1989.

[113] C. F. Van Loan and G. H. Golub. *Matrix Computations.* Johns Hopkins University Press, 1983.

[114] J. Villalba, T. Lang, and J. Hormigo. Radix-2 Multioperand and Multiformat Streaming Online Addition. *IEEE Transactions on Computers*, 61(6), 2011.

[115] C. Vuik. Krylov Subspace Solvers and Preconditioners. *ESAIM: Proceedings and Surveys*, 63, 2018.

[116] E. Wang, J. J. Davis, R. Zhao, H.-C. Ng, X. Niu, W. Luk, P. Y. K. Cheung, and G. A. Constantinides. Deep Neural Network Approximation for Custom Hardware: Where We've Been, Where We're Going. *ACM Computing Surveys*, 52(2), 2019.

[117] S. Waser and M. J. Flynn. Introduction to Arithmetic for Digital Systems Designers. 1982.

[118] O. Watanuki and M. D. Ercegovac. Floating-point On-line Arithmetic: Algorithms. In *IEEE Symposium on Computer Arithmetic*, 1981.

[119] O. Watanuki and M. D. Ercegovac. Error Analysis of Certain Floating-point On-line Algorithms. *IEEE Transactions on Computers*, C-32(4), 1983.

[120] D. M. Young. *Iterative Solution of Large Linear Systems.* Elsevier, 2014.

[121] D. Yuan and X. Zhang. An Overview of Numerical Methods for The First Kind Fredholm Integral Equation. *SN Applied Sciences*, 1(10), 2019.

[122] Y. Zhao, J. Wickerson, and G. A. Constantinides. An Efficient Implementation of Online Arithmetic. In *IEEE International Conference on Field Programmable Technology*, 2016.