ARCHITECT: Arbitrary-precision Hardware with Digit Elision for Efficient Iterative Compute

He Li, *Student Member, IEEE*, James J. Davis, *Member, IEEE*, John Wickerson, *Senior Member, IEEE*, and George A. Constantinides, *Senior Member, IEEE*

***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. We propose a fundamentally new approach: with the appropriate arithmetic able to generate results from most-significant digit first, we show 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 our 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. Use of most-significant digit-first arithmetic additionally allows us to declare certain digits to be stable at runtime, avoiding their recalculation in subsequent iterations and thereby increasing performance and decreasing memory footprints. Versus arbitrary-precision iterative solvers without the optimisations we detail herein, we achieve up-to 16× performance speedups and 1.9× memory savings for the evaluated benchmarks.**

***Index Terms*—Arbitrary-precision arithmetic, hardware architecture, online arithmetic, field-programmable gate array.**

I. 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 $f \in \left(\mathbb{R}^N \to \mathbb{R}^N\right)$ 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 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 Jacobi and successive over-relaxation, as well as others including gradient descent methods, the key algorithms in deep learning [1].

The authors are with the Department of Electrical and Electronic Engineering, Imperial College London, London, SW7 2AZ, United Kingdom. E-mail: {h.li16, james.davis, j.wickerson, g.constantinides}@imperial.ac.uk.

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: $\quad \hat{\boldsymbol{x}}^{(k+1)} \leftarrow \hat{f}\Big(\hat{\boldsymbol{x}}^{(k)}\Big)$
3: **end for**
Assert: $\left\|\Pi\Big(\hat{\boldsymbol{x}}^{(K)}\Big) - \boldsymbol{x}^*\right\| < \eta$

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-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.

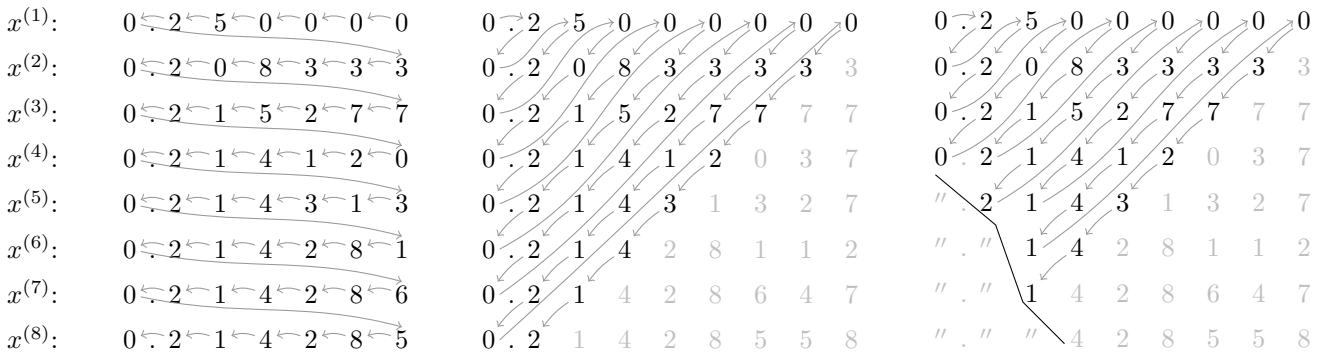
As a simple demonstration of this problem, suppose we wish to compute the toy iteration

$$x^{(k+1)} = 1/4 - 1/6 \cdot x^{(k)}$$

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. 1a shows the order in which digits are calculated when the precision is fixed to six 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.

Our proposal, illustrated in Fig. 1b, avoids the need to answer the aforementioned question entirely. The digits are calculated in a zig-zag pattern, sweeping through approximants 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* [2].



(a) Conventional arithmetic, approximants calculated LSD first.

(b) Our proposal, MSD first.

(c) Our proposal, MSD first with don't-change digit elision. ″-marks indicate don't-change digits, with the solid line representing the region's boundary.

Fig. 1. Digit-calculating strategies for the solution of $x^{(k+1)} = 1/4 - 1/6 \cdot x^{(k)}$ starting from $x^{(0)} = 0$. Arrows show the order of digit generation.

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 [3]–[5].

As originally formulated, this method is somewhat inefficient since the triangular shape traced out results in the computation of more digits than is actually needed. In the bottom-left corner lie high-significance digits of later approximants; these generally become stable over time, so we call them *don't-change* digits. By detecting the presence and avoiding the recomputation of these digits, we arrive at a digit pattern such as that shown in Fig. 1c. This increases efficiency while having no bearing on the chosen iterative method's ability to reach a result of any accuracy.

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. We make the following novel contributions:

- The first fixed-compute-resource hardware 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.
- Theoretical analysis of MSD stability within any online arithmetic-implemented iterative method, enabling the runtime elision of don't-change digits to obtain performance speedups and increase memory efficiency.
- Exemplary hardware implementations of our 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.

An earlier version of this work appeared in the proceedings of the 16th International Conference on Field-programmable Technology (FPT) [5]. This article combines that paper's material with the don't-change digit proposal taken from our 24th IEEE Symposium on Computer Arithmetic (ARITH) publication [6], extending both. In particular:

- We add an arbitrary-precision divider to our available operators, enabling the construction of datapaths for the calculation of irrational results with Newton's method.
- Changes to our digit elision technique, originally designed for linear-convergence algorithms, are made to suit the Newton method's quadratic convergence.
- To complement the new digit elision strategy, we propose an enhanced memory-addressing scheme, leading to greater performance and higher achievable result accuracy for a given memory budget.
- Finally, we exploit digit-parallel online addition to decrease datapath latency.

These optimisations allow us to obtain significant increases in throughput and memory efficiency over previous designs.

The implementations presented and evaluated in this article are fixed-point. ARCHITECT's principles are, however, generic, and could be employed for the construction of floating-point operators supporting arbitrary-precision mantissas.

II. Background

In scientific computing, machine learning, optimisation and many other numerical application areas, methods of iterative calculation are particularly popular and interest in their acceleration with FPGAs is growing [7]. Recent studies have demonstrated that FPGAs are promising platforms for the acceleration of the Jacobi [8], Newton's [9], conjugate gradient [10] and MINRES methods [11]. However, 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 bandwidth requirements, while for digit-serial it is one of the factors affecting algorithm runtime. Runtime tuning of precision in iterative calculations was enabled through the use of online arithmetic in recent work [4], however unrolling was necessary in order to implement the algorithm's loop; area therefore scaled with the desired number of iterations. As shown in Table I, ARCHITECT stands apart from these alternatives by enabling the runtime selection of both factors affecting result accuracy while keeping compute area constant.

TABLE I
COMPARISON OF ITERATIVE ARITHMETIC PARADIGMS.

	Area scales with		Runtime scales with	
	Prec.	Iter. limit	Prec.	Iter. limit
LSD-first, parallel	✔	✘	✘	✔ unbounded
LSD-first, serial	✘	✘	✔ bounded	✔ unbounded
Zhao *et al.* [4]	✘	✔	✔ unbounded	✘
ARCHITECT	✘	✘	✔ unbounded	✔ unbounded

TABLE II
COMPARISON OF ARBITRARY-PRECISION TECHNIQUES.

	Level	Prec. set per calc.	Iter. limit set per calc.
MPFR [16]	Software	Before	During
FloPoCo [18], *etc.*	Hardware	Before	During
Mixed precision [8], [20]	Hardware	Before	During
Zhao *et al.* [4]	Hardware	During	Before
ARCHITECT	Hardware	During	During

A. Arbitrary-precision Computing

Applications requiring very high precisions have become increasingly popular in recent years [12]. For example, today, hundreds of digits of precision are required in atomic system simulations and electromagnetic scattering theory calculations, while Ising integrals and elliptic function evaluation need thousands of digits [13]. In experimental mathematics, Poisson equation computations frequently require results to tens or hundreds of thousands of digits of precision [14]. Standard numeric datatypes, such as double- or even quadruple-precision floating-point, are therefore no longer sufficient in an increasing number of scenarios.

Many software libraries have been developed for arbitrary-precision arithmetic [15]–[17]. The *de facto* standard is MPFR, which guarantees correct rounding to any requested number of bits, selected before each operation is executed. Interest in the hardware acceleration of high-precision operations, in particular those within iterative algorithms, is growing [7]. FPGAs provide flexibility not available on other platforms, allowing for the implementation of bespoke designs with many precision and performance tradeoffs. Libraries including FloPoCo [18] and VFLOAT [19], alongside proprietary vendor tools, facilitate the creation of custom-precision arithmetic units. These provide designers with many options to suit particular frequency, latency and resource requirements. 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 [8]. A dual-mode (double- and quadruple-precision) architecture based on Taylor series expansion has also been implemented [20]. Zhao *et al.*'s work enables arbitrary-precision computation but, as mentioned earlier, necessitates compile-time determination of iteration count [4].

With the exception of Zhao *et al.*'s architecture [4], 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 doing so. In our work, we are particularly interested in hardware architectures which allow precision to be increased over time without having to restart computation or modify the circuitry. Table II presents a side-by-side comparison of these techniques and their features with ARCHITECT, the only entry supporting the determination of result precision and iteration count *after each calculation has commenced.*

B. 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 [2]. By employing redundancy in their number representation, allowing less-significant digits to correct errors introduced in those of higher significance, all online operators are able to function in MSD-first fashion. Online operators are classically serial, however efficient digit-parallel (unrolled) implementations targetting FPGAs have been developed as well [3]. We make use of both digit-serial and -parallel online operators in this work, employing the *de facto* standard radix-2 signed-digit number representation, wherein 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 $x_i = x_i^+ - x_i^-$. Data can be efficiently converted between non-redundant and redundant forms using well known on-the-fly conversion techniques [2].

Of particular significance to the material presented in this article 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 [2]. The value of δ is operator-specific, but is typically a small integer. When chaining operators to form a datapath, as we do, the total online delay is the highest cumulative delay through the complete circuit [4].

1) Addition: A classical 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 Fig. 2 (left), from most to least significant [2]. Digits of z start to appear at serial output z_{out} after two clock cycles, hence $\delta_+ = 2$. Duplication of such a serial adder P times and removal of its registers leads to the creation of a P-digit parallel online adder devoid of online delay, as shown in Fig. 2 (right). Crucially, while carry digits are presented at the least-significant end of the adder and generated at the most, there is no carry chain; independent of its word length, the critical path lies across two full adders. This demonstrates the adder's suitability for the construction of more complex online operators and indicates that its maximum frequency is independent of precision.

2) Multiplication: Algorithm 2 illustrates classical radix-2 online multiplication [2]: a process that operates in serial-

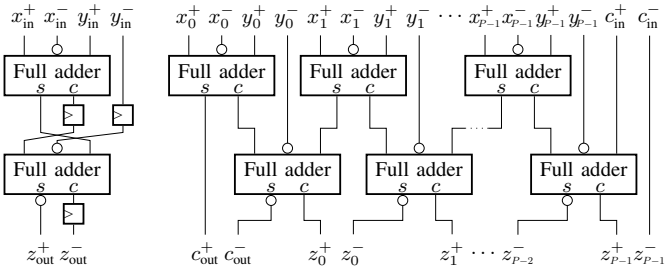


Fig. 2. Radix-2 online adders [2]. Left: Serial. Right: Parallel.

Algorithm 2 Radix-2 online multiplication [2].

Inputs: serially presented multiplicand x, multiplier y

$\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w} \leftarrow \boldsymbol{0}$
for $j = 0$ **to** $P + 2$ **do**
  $\boldsymbol{y} \leftarrow \boldsymbol{y} \parallel y_j$
  $\boldsymbol{v} \leftarrow 2\boldsymbol{w} + 2^{-3}(\boldsymbol{x}y_j + \boldsymbol{y}x_j)$
  $z_{j-3} \leftarrow \mathrm{sel}_{\times}(\boldsymbol{v})$
  $\boldsymbol{w} \leftarrow \boldsymbol{v} - z_{j-3}$
  $\boldsymbol{x} \leftarrow \boldsymbol{x} \parallel x_j$
end for

Output: serially generated product z

in, serial-out fashion. So-called digit vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ are assembled from digits of multiplicand x and multiplier y over time from the most significant first; $\parallel$ 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 $\mathrm{sel}_{\times}$ serves to determine the digits of product z. This is defined to be

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

z_j is produced at cycle $j + 3$ since $\delta_{\times} = 3$. Note that 'digits' z_j for $j < 0$ are ignored. P-digit online addition lies at the heart of the algorithm; due to its fixed width, hardware that implements Algorithm 2 can multiply to a precision of at most P digits, which must be fixed in advance.

3) 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.

Algorithm 3 Radix-2 online division [2].

Inputs: serially presented dividend x, divisor y

$\boldsymbol{y}, \boldsymbol{w}, \boldsymbol{z} \leftarrow \boldsymbol{0}$
for $j = 0$ **to** $P + 3$ **do**
  $\boldsymbol{y} \leftarrow \boldsymbol{y} \parallel y_j$
  $\boldsymbol{v} \leftarrow 2\boldsymbol{w} + 2^{-4}(x_j - \boldsymbol{z}y_j)$
  $z_{j-4} \leftarrow \mathrm{sel}_{\div}(\boldsymbol{v})$
  $\boldsymbol{w} \leftarrow \boldsymbol{v} - z_{j-4}\boldsymbol{y}$
  $\boldsymbol{z} \leftarrow \boldsymbol{z} \parallel z_{j-4}$
end for

Output: serially generated quotient z

III. Proposed ARCHITECTure

Using classical online operators as a starting point, we now describe the construction of constant-compute-resource hardware capable of performing iterative computation to increasing precision over time. We call this concept ARCHITECT.

A. Digit-vector Storage

Classical online operators make use of registers to store digit vectors. When implementing Algorithm 2 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; we must use random-access memory (RAM) for digit-vector storage to avoid both under- and over-budgeting register resources. We break p 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, independent digit vectors exist for each step, thus their indexing requires three variables: $c \in [0, n)$, $u \in [0, U)$ and approximant index k.

Since ARCHITECT requires k and i to both vary non-monotonically as time progresses, 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. We use a Cantor pairing function (CPF) [21], a bijection from $\mathbb{N}^2$ onto $\mathbb{N}$, for this purpose, defined to be

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

The function's bijectivity is crucial for ARCHITECT. Unlike classical row- or column-major indexing, the injectivity of the CPF allows both dimensions to grow without bound while providing a unique result for every (k, c). Its operation is demonstrated visually in Fig. 3; what is conceptually a three-dimensional array indexed as (k, c, u) becomes a two-dimensional array indexed by $(\mathrm{cpf}(k, c), u)$ instead, thereby suiting the 'flat' nature of RAM. The function's surjectivity ensures that every $\mathrm{cpf}(k, c)$ is produced by some (k, c), thus enabling the most efficient use of the available memory.

B. Arbitrary-precision Operators

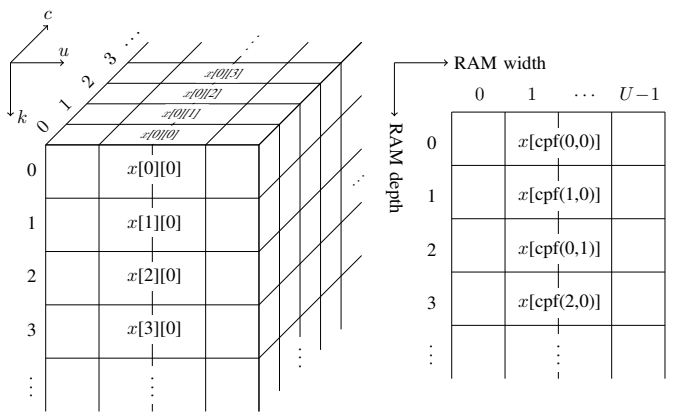


Fig. 3. Operation of our Cantor pairing function, showing the transformation of a three-dimensional array growing with both approximant and chunk indices k and c to a structure growing only in a single dimension.

Algorithm 4 Radix-2 ARCHITECT multiplication.

Inputs: serially presented multiplicand x, multiplier y; approximant index k, precision p

$\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w} \leftarrow \mathbf{0}$
for $j = 0$ **to** $p + 2$ **do**
  $\boldsymbol{y}[\text{cpf}(k, \lfloor j/U \rfloor)][j \bmod U] \leftarrow y_j$
  for $c = \lfloor j/U \rfloor$ **to** 0 **do**
    $\boldsymbol{v}[\text{cpf}(k,c)] \leftarrow 2\boldsymbol{w}[\text{cpf}(k,c)] + 2^{-3}(\boldsymbol{x}[\text{cpf}(k,c)]y_j + \boldsymbol{y}[\text{cpf}(k,c)]x_j)$
    if $c > 0$ **then**
      $\boldsymbol{w}[\text{cpf}(k,c)] \leftarrow \boldsymbol{v}[\text{cpf}(k,c)]$
    end if
  end for
  $z_{j-3} \leftarrow \text{sel}_{\times}(\boldsymbol{v}[\text{cpf}(k,0)])$
  $\boldsymbol{w}[\text{cpf}(k,0)] \leftarrow \boldsymbol{v}[\text{cpf}(k,0)] - z_{j-3}$
  $\boldsymbol{x}[\text{cpf}(k, \lfloor j/U \rfloor)][j \bmod U] \leftarrow x_j$
end for

Output: serially generated product z

1) Multiplication: We are now in a position to rewrite Algorithm 2 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 of p-digit numbers' chunks, 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.

2) Division: The equivalently transformed version of Algorithm 3 is shown in Algorithm 5. Mirroring the increased complexity of classical 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 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 $(-1, 1)$ [2]. The normalisation of quotients following online division ordinarily necessitates variable

Algorithm 5 Radix-2 ARCHITECT division.

Inputs: serially presented dividend x, divisor y; approximant index k, precision p

$\boldsymbol{y}, \boldsymbol{w}, \boldsymbol{z} \leftarrow 0$
for $j = 0$ **to** $p + 3$ **do**
  $\boldsymbol{y}[\text{cpf}(k, \lfloor j/U \rfloor)][j \bmod U] \leftarrow y_j$
  for $c = \lfloor j/U \rfloor$ **to** 0 **do**
    $\boldsymbol{v}[\text{cpf}(k,c)] \leftarrow 2\boldsymbol{w}[\text{cpf}(k,c)] + 2^{-4}(x_j - \boldsymbol{z}[\text{cpf}(k,c)]y_j)$
  end for
  $z_{j-4} \leftarrow \text{sel}_{\div}(\boldsymbol{v}[\text{cpf}(k,0)])$
  for $c = \lfloor j/U \rfloor$ **to** 0 **do**
    $\boldsymbol{w}[\text{cpf}(k,c)] \leftarrow \boldsymbol{v}[\text{cpf}(k,c)] - z_{j-4}\boldsymbol{y}[\text{cpf}(k,c)]$
  end for
  $\boldsymbol{z}[\text{cpf}(k, \lfloor j/U \rfloor)][j \bmod U] \leftarrow z_{j-4}$
end for

Output: serially generated quotient z

$\delta_{\div}$ [22]. To avoid this, we can maintain a fixed online delay by bounding divisor magnitude within $[1/r, 1)$ [23]. For experimentation, we can guarantee alignment across iterations through the appropriate selection of initial inputs.

C. Digit Computation Scheduling

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 we have the freedom to trade off between iteration count and precision within the bounds of these dependencies, we always assume a mapping from current to next digit of the form depicted in Fig. 4. 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 III-E—and limits transitions between approximants. The latter is beneficial since, as will be explained in Section III-G, switching between approximants leads to the incursion of performance penalties under some circumstances.

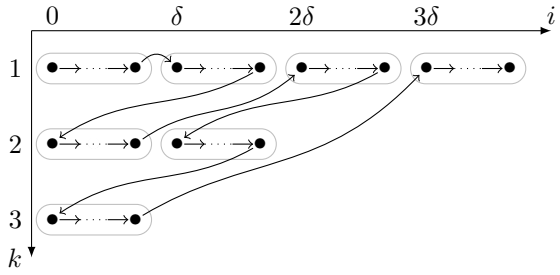


Fig. 4. Proposed digit generation pattern without don't-change digit elision for generic iterative computation using online operators.

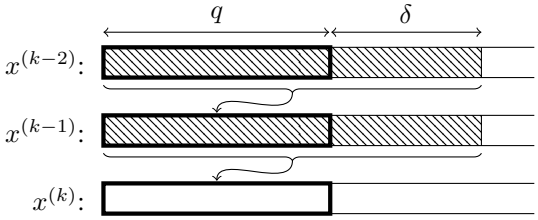


Fig. 5. A proof sketch showing why it is sound to omit don't-change digits. If the two hatched regions contain the same $q+\delta$ digits, the three thick boxes are guaranteed to contain the same q digits, hence $x^{(k)}$'s calculation can begin from digit index q.

D. Don't-change Digit Elision

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, *i.e.* those of later approximants that have stabilised. 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: it induces no approximation.

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 need to calculate them; 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 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 II-B, we emphasise that the first q digits of one approximant depend only upon the first $q+\delta$ digits of the previous approximant [2]. 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.

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 equal, 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 V-F, this overhead is small and amortised out the more RAM is instantiated for storing digit vectors. Since we have elected to process digits in groups of δ, it makes sense to also limit our don't-change digit elision to this granularity. We thus avoid the processing of entire groups of digits, where possible.

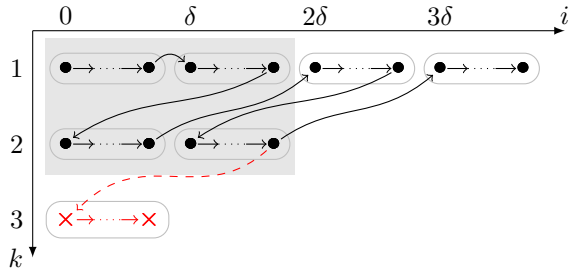


Fig. 6. Digit generation pattern with don't-change digit elision. Groups of digits in the shaded region were found to be identical at runtime, allowing computation of the first group to be skipped in the subsequent iteration. Dashed lines are scheduled paths not taken and ×s are digits therefore elided.

As a result of the introduction of don't-change digit elision, our scheduling pattern becomes dynamic. Fig. 6 shows an example. This is similar to Fig. 4, but, due to the identification of the third approximant's first group of MSDs as stable, we can advance into the iteration-precision space more quickly than had we not elided them, increasing compute efficiency.

Along with increased performance, the elision of don't-change digits also enables us to increase memory efficiency. Defining ψ as the number of digits guaranteed not to have changed within the current approximant, as determined through the runtime comparison of MSDs within the preceding two approximants, we can substitute

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

for (1), where $\hat{c} = \lfloor (i-\psi+1)/U \rfloor$. By doing so, stable digits no longer need to be recomputed or stored. In common with its predecessor, this optimised storage strategy guarantees no memory wastage through the surjectivity of its mapping from approximant and chunk indices to memory addresses.

E. Control Logic

Given a particular (k,i), we can compute the subsequent (k,i), (k',i'), needed to realise scheduling patterns such as those shown in Figs 4 and 6 with the finite-state machine (FSM) depicted in Fig. 7. Therein, we present state transition conditions both with and without don't-change digit elision functionality. When elision is enabled, the conditions shown in boxes are evaluated in addition to those outside.

The states' functionality is as follows.

- *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 without don't-change digit elision. When enabled, ψ must also be considered.
- *Accumulation*: Assuming that the constructed datapath contains at least one multiplier or divider, we must account for the variable latency of those operators. The throughput of the datapath as a whole is determined by

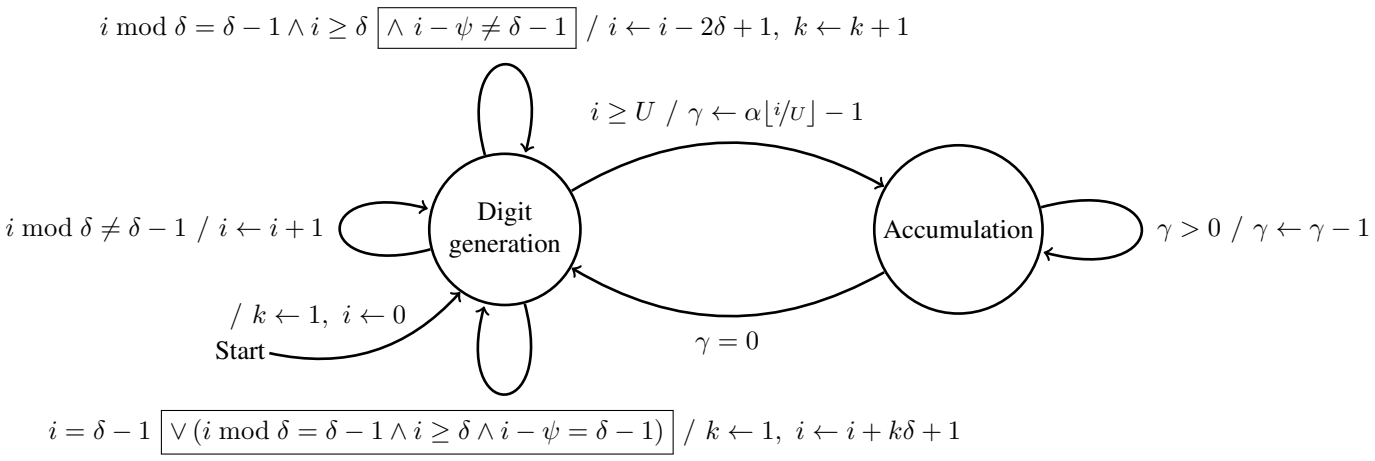


Fig. 7. FSM for digit computation scheduling. Transition edges are labelled with conditions and actions separated by slashes (/). If the datapath consists only of adders, the accumulation state is never entered. Otherwise, $\alpha = 2$ if the datapath contains one or more dividers, and is 1 in all other cases. Boxed conditions apply only when don't-change digit elision is active; they are otherwise ignored. Termination occurs either on demand or following memory exhaustion.

the slowest operator. Since ARCHITECT's multiplication and division operators have dissimilar accumulation functionality, as was explained in Section III-B, the number of clock cycles consumed by each is different. If the datapath contains at least one divider, advancement must be inhibited for $2\lfloor i/U \rfloor - 1$ cycles per generated digit. If it does not, but does contain at least one multiplier, this factor is $\lfloor i/U \rfloor - 1$ instead. Counter γ sequences the return to the digit generation state. Since i is variable, this loop cannot be unrolled. In the case that the datapath contains only adders, entry into this state never occurs.

F. Accuracy Bounds

Let us assume the existence of a target result defined by its approximant 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. We emphasise that ARCHITECT does not necessitate its users to specify K or P up-front, while other approaches require either one or both of these—usually P—to be determined before beginning to iterate. Since don't-change digits are identified at runtime, the analysis herein applies to ARCHITECT without digit elision. Enabling this optimisation will therefore increase the bounds that follow.

As shown in Fig. 8, we define the number of iterations resulting from computation to target (K, P) 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 our approach. Thus, if higher accuracy is required, more memory must be instantiated.

Upon termination, the precision of approximant k will be

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

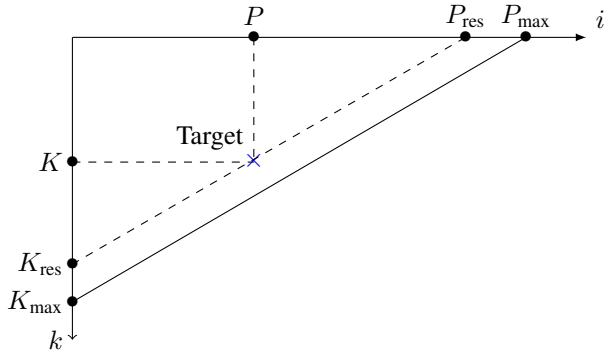


Fig. 8. How the final precision and iteration count ($K_{\text{res}}, P_{\text{res}}$) are constrained by the desired result (K, P) and the available memory ($K_{\text{max}}, P_{\text{max}}$).

where K_{res} can be geometrically deduced 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 our pairing function in (1) allows us to derive

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

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

G. 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. Let us first assume that don't-change digit elision is disabled. This total time T can be broken down into the following three components such that $T = T_1 + T_2 + T_3$.

- *Initial online delay*: We must wait δ clock cycles before each approximant's result begins to appear, thus the delay across all iterations is simply
$$T_1 = \delta K_{\text{res}}.$$
- *Digit generation*: Across all iterations performed, the total time for digit generation is either
$$T_2 = \sum_{k=0}^{K_{\text{res}}-1} p^{(k)}\Big(2n^{(k)}-1\Big) - Un^{(k)}\Big(n^{(k)}-1\Big) - \delta,$$
if the datapath contains one or more dividers, or
$$T_2 = \sum_{k=0}^{K_{\text{res}}-1} n^{(k)}\left(p^{(k)} - \frac{U\big(n^{(k)}-1\big)}{2}\right) - \delta$$
if it contains one or more 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 one or more adders,
$$T_2 = \sum_{k=0}^{K_{\text{res}}-1} p^{(k)} - \delta.$$
$p^{(0)}$ 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 [2]. This ensures that the calculated digit aligns with its truncated digit vectors. For this,
$$T_3 = \beta\big(K_{\text{res}}^2 - K_{\text{res}} + 2K - 2\big),$$
where β is the number of serial adders present along the highest-online delay path within the circuit.

When enabled, don't-change digit elision generally allows computation time to be reduced below T. Since the amount of achievable reduction is input-dependent, however, it is not practicable to determine such reductions analytically.

H. Digit-parallel Addition Optimisation

It is possible to eliminate the final T component in Section III-G, resulting in $T_3 = 0$, by using three-digit parallel online adders in place of serial ones. We 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.

IV. Benchmarks

In order to evaluate ARCHITECT, we implemented two widely used iterative algorithms—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. We chose Jacobi and Newton to exemplify a large class of iterative methods with linear and quadratic convergence properties, respectively. Except where otherwise stated, ARCHITECT implementations featured all of the previously described optimisations: don't-change digit elision, its related memory-addressing and digit-scheduling schemes and serial-to-parallel online adder substitutions.

A. 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
$$\boldsymbol{x}^{(k+1)} = \boldsymbol{D}^{-1}\Big(\boldsymbol{b} - \boldsymbol{R}\boldsymbol{x}^{(k)}\Big),$$
or, expressed in element-wise fashion,
$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j\neq i\in[0,N)} a_{ij}x_j^{(k)}\right) \forall i \in [0, N),$$
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 computed value of $\boldsymbol{x}$; the calculation can therefore be parallelised by computing each $x_i^{(k+1)}$ independently. A convergence criterion, $\left\|\boldsymbol{Ax}^{(k)} - \boldsymbol{b}\right\| < \eta$, can be used in order to determine if the solution has been found to great enough accuracy.

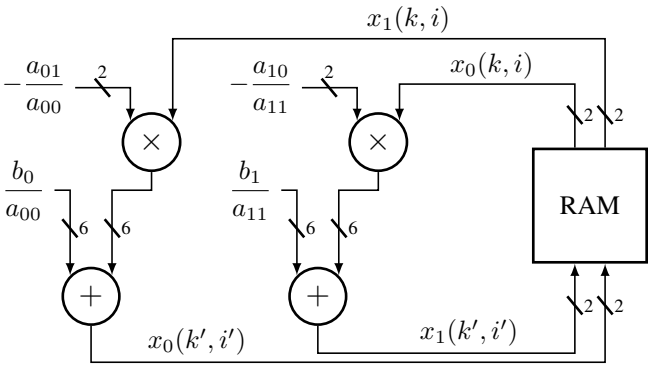
Such a system is guaranteed to be soluble when $\boldsymbol{A}$ is strictly diagonally dominant, *i.e.* if the condition $|a_{ii}| > \sum_{j\neq i}|a_{ij}|$ holds for all i. Although strict diagonal dominance is not a necessity in every case, we assume this condition to always be satisfied for simplicity.

A metric used to quantify the sensitivity of a particular linear system to error is the *condition number* of $\boldsymbol{A}$ [24], where
$$\kappa(\boldsymbol{A}) = \|\boldsymbol{A}\|\left\|\boldsymbol{A}^{-1}\right\|.$$
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]. We can expect to need at least ω additional digits of precision in order to compute a system with $\kappa(\boldsymbol{A}) = 2^\omega$ than required if $\kappa(\boldsymbol{A})$ were 1 [26].

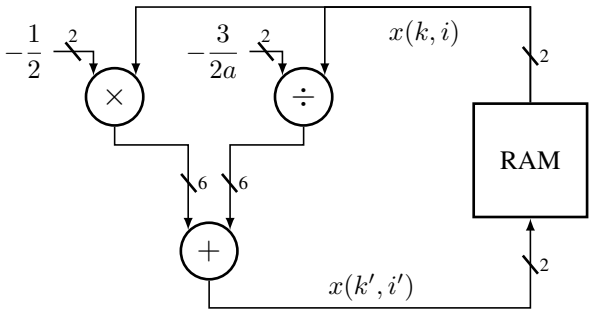
Without loss of generality, the datapath we developed to solve systems with dimensionality $N = 2$ is depicted in Fig. 9a, featuring ARCHITECT numerical operators as described in Section III-B. Jacobi solvers with $N > 2$ could have been built with additional multipliers and adders, but this is not the emphasis—demonstrating arbitrary-accuracy iterative calculation—of this work. Note that runtime division is unnecessary since $\boldsymbol{A}$ and $\boldsymbol{b}$ are constants and that simple rearrangement transforms subtraction into addition.

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



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



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

Fig. 9. ARCHITECT 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.

where f' is the first derivative of f. Assuming that $f(x) = 0$ is soluble and $f'(x)$ is Lipschitz continuous, convergence is quadratic if $x^{(0)}$ is sufficiently close to the solution [27].

We implemented the datapath shown in Fig. 9b, again with ARCHITECT operators, as a second case study. This can solve equations of the form $f(x) = ax^2 - 3 = 0$:

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

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's arbitrary-precision capabilities.

V. EVALUATION

We conducted theoretical analysis and performed experiments to investigate how ARCHITECT 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 article, is the multiplicative inverse of throughput.

The closest study to this work is that presented by Zhao *et al.* [4], which we compare against directly. For comparison against traditional arithmetic, we chose to implement parallel-in serial-out (PISO) operators since ARCHITECT 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 [28]. 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 [29] while also being dramatically faster than SISO [30]. 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.

TABLE III
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.* [4]	$\mathcal{O}(N^2K)$	$\mathcal{O}(N^2KP)$	$\mathcal{O}(P(\log(N)K+P))$
ARCHITECT	$\mathcal{O}(N^2)$	$\mathcal{O}\left(N^2(K+P)^2\right)$	$\mathcal{O}\left(\frac{(\log(N)K+P)^3}{\log(N)}\right)$

[1] N-dimensional Jacobi method, N^{th}-order Newton's method.

A. Complexity Analysis

In Table III, we present the results of asymptotic complexity analysis—in terms of circuit size, memory requirements and latency—performed for ARCHITECT 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 implementations were derived from analytical expressions provided by the authors [4].

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 implementation, however, scales with neither K nor P, since 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 require residue storage within their multipliers and dividers; memory occupancy therefore scales with area for the arbitrary-precision architectures. For the former, use of memory also scales with P as residues are stored to the same precision as its input data. Since ARCHITECT 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 (1) during computation to the target (K, P). They thus 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 $\mathcal{O}(\log(N)KP)$ [4]. For ARCHITECT, given that each datapath's highest cumulative

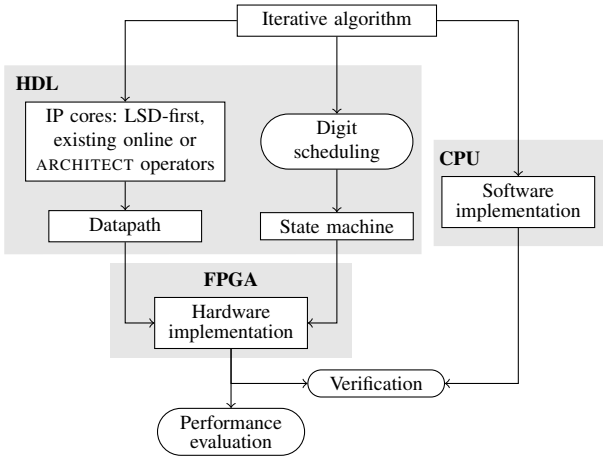


Fig. 10. Experimental setup for the evaluation of ARCHITECT.

online delay δ is logarithmically related to N, its latency complexity can be determined by solving for T_2 in Section III-G. Note that T_2 dominates T_1 in all cases and $T_3 = 0$ since we assume the use of digit-parallel adders.

At first glance, it appears that ARCHITECT behaves more poorly than its competitors in terms of memory use and solve time when scaled. We emphasise, however, that these complexities represent worst-case scenarios for ARCHITECT: optimisations including digit elision do not factor into its asymptotic behaviour but do significantly improve the average case. They also do not take fundamental limitations of the alternatives into account. In particular, exact computation to a given (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'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 empirically explore the implications of these issues.

B. Experimental Particulars

We targetted a Xilinx Virtex UltraScale FPGA (XCVU190-FLGB2104-3-E) 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. 10 captures our experimental process.

C. Qualitative Performance Comparison

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

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

with b_0 and b_1 randomly selected from a uniform distribution in the range $[0, 1)$. As m increases, condition number $\kappa(\boldsymbol{A}_m)$ also increases, indicating that higher precision P will be required to generate a result of great-enough accuracy. We 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 $m = 25$, so we limited our experiments to $m \in [0, 25]$. We postulate that ARCHITECT 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, we used RAM size $(U, D) = \left(8, 2^{10}\right)$. Latencies were calculated using frequencies taken from Section V-E.

Fig. 11a captures the latency ratio between ARCHITECT 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. For the most well conditioned matrices ($m \leq 0.15$), ARCHITECT takes less time to reach the target accuracy. For larger m, however, the opposite is true: lower-indexed approximants are computed to greater precision than those of PISO, taking more time. Had a lower choice of P been made for PISO, ARCHITECT 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. 11c, with $P = 8$ (LSD-8), ARCHITECT 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. We can conclude that ARCHITECT requires less time to generate results either when P is small and convergence is fast or when P is too large for PISO to ever converge.

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

Figs 11b and 11d show the performance of our ARCHITECT-based Newton's method benchmark versus 32-bit and 8-bit PISO in the same form as Figs 11a and 11c, respectively. The results achieved for Newton's method are broadly similar to those for Jacobi. ARCHITECT requires $a \leq 2.8$ 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.

D. Area & Frequency Scalability

Implementational results are presented in Fig. 12 for our 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, on the x-axis, and RAM width U was 8 in all cases.

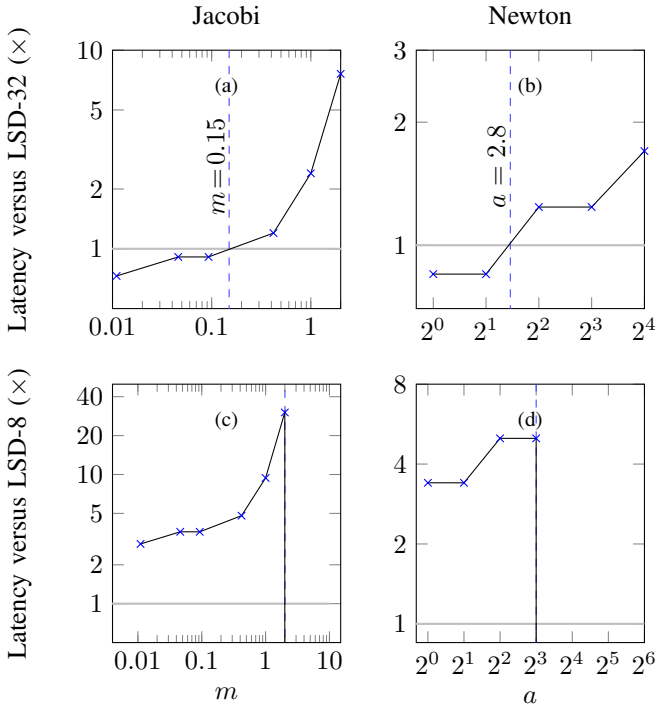


Fig. 11. Performance comparisons of our proposal against LSD-first arithmetic for the Jacobi and Newton's methods. (a) and (b) show how the conditioning of input matrix $\boldsymbol{A}_m$ (Jacobi) and input value a (Newton) affect the solve time of our proposal compared to LSD-32. ARCHITECT computes more quickly than LSD-32 when $m \leq 0.15$ for Jacobi and $a \leq 2.8$ 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.

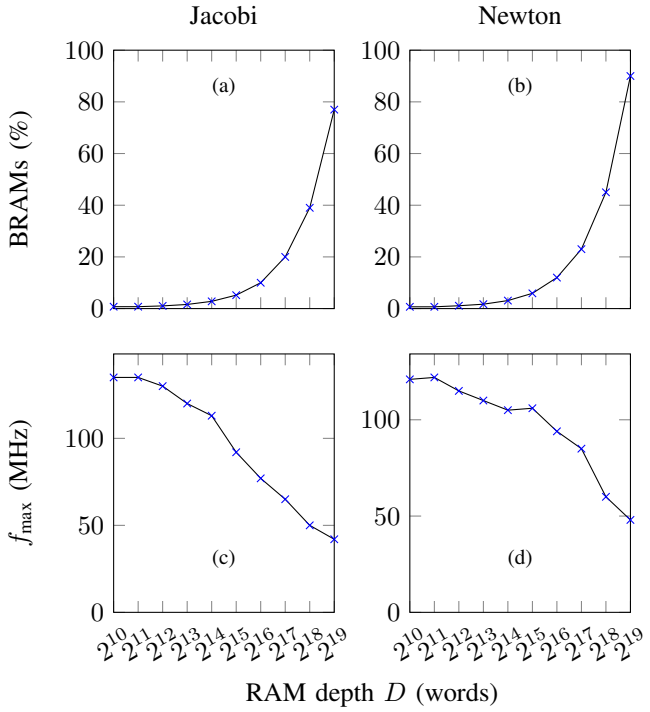


Fig. 12. Resource use and maximum clock rate of ARCHITECT Jacobi and Newton benchmarks versus RAM depth D. Area is reported in terms of BRAMs only; LUT and FF use were below 1% for all design points.

Lookup table (LUT) and flip-flop (FF) use are not shown since the numbers are insignificant compared to those of on-chip block RAM (BRAM)—from 0.17% to 0.66% for LUTs and 0.045% to 0.21% for FFs for the smallest ($D = 2^{10}$) and largest ($D = 2^{19}$) Jacobi designs implemented, and from 0.22% to 0.86% (LUTs) and 0.040% to 0.23% (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 90% and 77% of BRAMs allocated for the Jacobi and Newton methods, respectively, we can reach $K_{\text{max}} = 1023$ and $P_{\text{max}} = 8184$: the maxima for our targetted FPGA with power-of-two choices of D. 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 our implementations are able to run at between 120 MHz, for the smallest D tested, to around 50 MHz for the largest of both benchmarks. Subtle increases are due to compilation noise.

ARCHITECT 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 IV to exemplify this for both of our 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 III-B2, division requires more cycles to produce each output digit than are needed for multiplication. Table V shows the area breakdown and minimum clock period (critical path delay) for each of our individual arithmetic components for an example (U, D).

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

	U	LUTs	FFs	BRAMs	f_{max} (MHz)	Accumulation latency (cycles)
Jacobi	8	1827	964	28	121	$\lceil p^{(k)}/8 \rceil$
	64	6964	2551	88	93	$\lceil p^{(k)}/64 \rceil$
Newton	8	2316	866	26	120	$2\lceil p^{(k)}/8 \rceil - 1$
	64	6102	1710	83	95	$2\lceil p^{(k)}/64 \rceil - 1$

TABLE V
ARCHITECT OPERATOR FEATURES WITH RAM SIZE $(U, D) = (8, 2^{10})$.

	LUTs	FFs	BRAMs	Minimum clock period (ns)
+	4	3	–	2.0
×	250	141	4	5.0
÷	255	93	6	5.6

E. Quantitative Area & Frequency Comparison

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

We chose to set targets of $\left(100, 2^{11}\right)$ (for the Jacobi method) and $\left(10, 2^{11}\right)$ (Newton). Thus, at their 100^{th} and 10^{th} iterations, respectively, we wish to obtain a result with 2048-digit precision. Fewer iterations were targetted for Newton's method due to its quadratic convergence. Using $U = 8$, for ARCHITECT, 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 $P_{\text{max}} \geq P_{\text{res}}$. We can determine that, by setting RAM depth $D = 2^{17}$, we are able to reach $K_{\text{max}} = 512$ and $P_{\text{max}} = 4088$, which satisfies these requirements for both benchmarks.

Fig. 13 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.* [4]. Most strikingly, the latter demonstrates area inefficiency, with resource use scaling linearly with iteration count K; ARCHITECT consumes 57× fewer LUTs and 59× fewer FFs than Zhao *et al.*'s proposal requires in order to execute 100 iterations of the Jacobi method. When executing 10 iterations of Newton's method, these factors are 8.4 and 13, respectively. f_{max} is comparable between the two since the underlying arithmetic is largely equivalent, although ARCHITECT's is slightly inferior principally due to reductions caused by the introduction of don't-change digit elision logic. For PISO, we can see that, while its f_{max} is initially much higher—over 300 MHz for $P = 2^4$—than ARCHITECT's, it falls as P increases; the crossover occurs at $P \approx 1400$. Taking Newton's method as an example, with a high precision requirement, such as 2^{11} digits, ARCHITECT is able to outperform its PISO counterpart in terms of f_{max} by a factor of 1.5. Corresponding decreases in LUT and FF use were also found: when computing to $P = 2^{10}$, again for Newton, ARCHITECT consumes 1.8× and 3.3× fewer of each than PISO, while for 2^{11} these factors increase to 3.6 and 6.5. Similar conclusions can be made for our implementation of the Jacobi method. Since the proposed designs are able to calculate to any $K \leq K_{\text{max}}$ and $P \leq P_{\text{max}}$, their area and f_{max} are constant.

F. Performance Improvement Breakdown

We conducted further analysis to investigate how the elision of don't-change digits and use of parallel online adders individually improve the performance and memory efficiency of ARCHITECT. Overall, Figs 14a and 14b show that solve time can be significantly reduced when enabling these optimisations. As expected, don't-change digit elision leads to the majority of our design's efficiency savings over 'vanilla' ARCHITECT (that without digit elision or parallel addition).

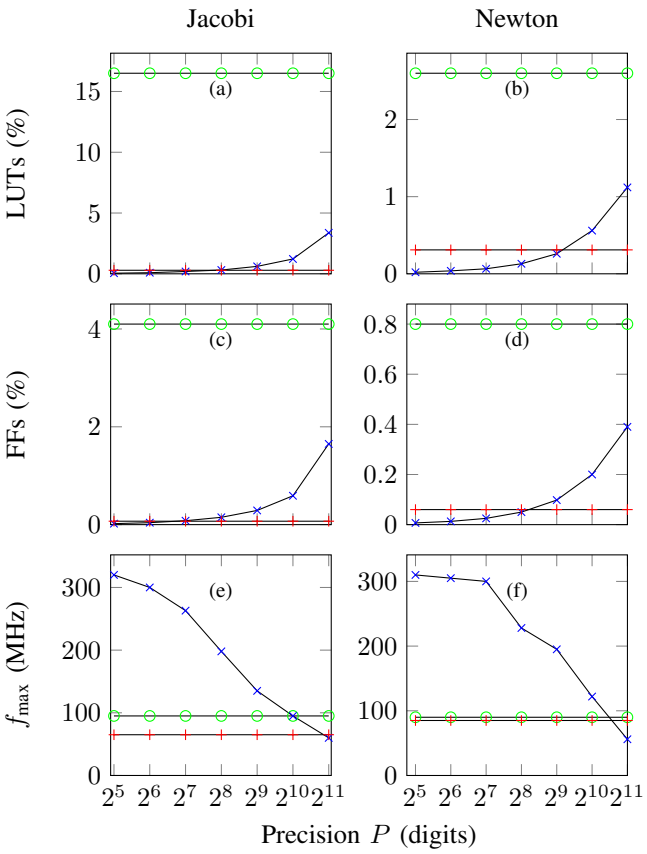


Fig. 13. Resource use and performance comparison of Jacobi and Newton's method implementations using Zhao *et al.*'s (—○—), PISO (—×—) and our (—+—) approaches versus required result precision P.

The gaps between the don't change-plus-parallel online addition and parallel addition-only lines widen as η is reduced, indicating that consideration of don't-change digits becomes more important with higher accuracy requirements. The subtle jump present in Fig. 14a is due to the δ-digit granularity of elision. With respect to using parallel online addition only, 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 digit. This also affords much greater opportunity for don't-change digit elision, however, hence the high overall speedups seen on the right-hand side of, in particular, Fig. 14b.

The speedups we observed for Newton's method were far more significant than those for Jacobi: up to 16× for the former. Relatively low performance improvements were expected for the Jacobi benchmark due to the method's linear convergence. 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. 14a than Fig. 14b.

Figs 14c and 14d show the memory efficiency improvements afforded through the use of don't-change digit elision for both

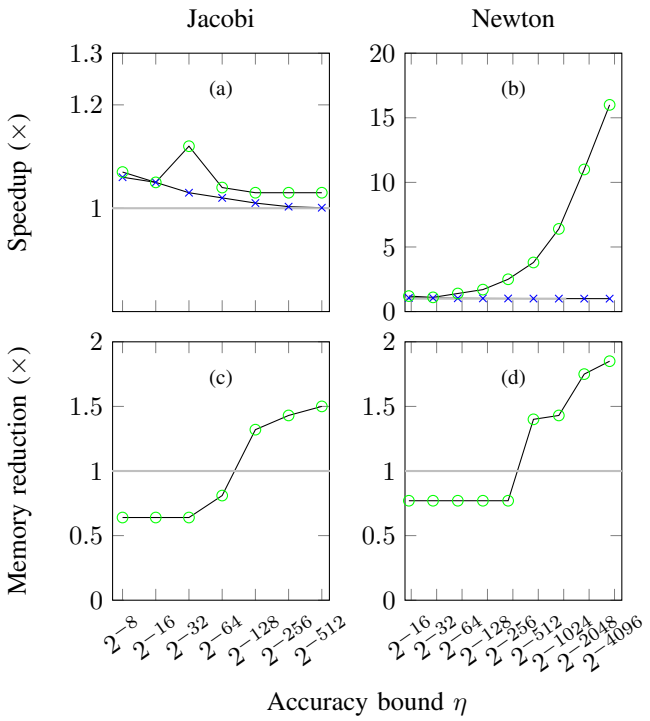


Fig. 14. Solve time speedup for (a) the Jacobi and (b) Newton's methods using both don't-change digit elision and parallel online addition (—○—) and parallel addition only (—×—) versus ARCHITECT with both optimisations disabled. (c) and (d) show the corresponding memory requirement reductions for Jacobi and Newton, respectively, facilitated through digit elision.

benchmarks. We present these as the ratio of the number of BRAM blocks that must be instantiated on our targetted FPGA for the solution of equations to particular accuracies with and without digit elision enabled. The jaggedness of these plots is due to the granularity of memories, for which we only used whole numbers of BRAMs. For lower-accuracy cases, both pairs of designs require approximately the same amount of memory, although that considering don't-change digits is slightly inferior due to the overheads involved in comparison and subsequent elision. However, don't-change digit elision allows our optimised Newton design to use the same amount of memory for $\eta \leq 2^{-512}$, while vanilla ARCHITECT starts to consume more memory when $\eta = 2^{-449}$. For the test cases we evaluated, we observed up-to 1.5× and 1.9× memory savings for the Jacobi and Newton's methods, respectively. Beyond those shown in Fig. 14, there are particularly high-accuracy cases—$\eta \geq 2^{-874}$ for Jacobi and $\eta \geq 2^{-7169}$ for Newton—vanilla ARCHITECT cannot reach before it exhausts its available memory, while that with digit elision can. The advantages of this scheme and its efficient memory addressing therefore come to the fore with higher accuracy requirements.

VI. Conclusion & Future Work

In this article, we 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. We named this technique ARCHITECT. Our 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. Using digit dependency analysis, we identified stable 'don't-change' digits across iterations, excluding them from calculation. This technique holds for any iterative method implemented using online arithmetic and was realised in hardware using simple runtime detection and digit-scheduling logic. We also proposed the replacement of serial online adders within iterative datapaths with parallel equivalents, facilitating latency reduction and consequent improvements in throughput.

We evaluated ARCHITECT on FPGAs using the Jacobi and Newton's methods in order to verify its accuracy and establish its scalability and efficiency. These benchmarks showcased the key advantage of our approach: removing the burden of having to determine and fix the precision of arithmetic operators in advance. By doing so, we showed that datapaths constructed from ARCHITECT operators are superior to their traditional arithmetic equivalents in scenarios where the latter's precision is either overly high for the problems being solved or too low for results to converge at all. A single ARCHITECT datapath is able to compute results to any accuracy, with the only limit being imposed by the size of the available RAM.

Our experiments revealed 12× LUT and 24× FF reductions over 2048-bit conventional parallel-in serial-out arithmetic, along with 57× LUT and 59× 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 3.6, 6.5, 8.4 and 13, respectively. Versus ARCHITECT with the proposed don't-change and parallel addition optimisations disabled, we were able to achieve up-to 16× decreases in solve time.

While we prototyped our designs on FPGAs owing to the costs and lead times associated with full-custom implementation, we 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. Our proposals cannot take advantage of such structures at present. We are confident that, should ARCHITECT see application-specific integrated circuit (ASIC) implementation, however, much more competitive performance would be achievable. Higher-radix ($r > 2$) online arithmetic could instead (or additionally) be employed to exploit high-performance adders, including on FPGAs, which we anticipate would also lead to speedups. We leave the exploration of such optimised implementations to future work.

Beyond this, we will extend our benchmarking to cover additional iterative algorithms, including Krylov subspace methods such as conjugate gradient descent. Finally, we envisage that the arbitrary-precision computation enabled by ARCHITECT can be combined with high-level synthesis to enable faster hardware specialisation.

Acknowledgements

The authors are grateful for the support of the United Kingdom EPSRC (grant numbers EP/P010040/1, EP/R006865/1 and EP/K034448/1), Imagination Technologies, the Royal Academy of Engineering and the China Scholarship Council.

Supporting data for this article are available online at `https://doi.org/10.5281/zenodo.3378800`.

REFERENCES

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep Learning," *Nature*, vol. 521, no. 7553, 2015.

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

[3] K. Shi, D. Boland, and G. A. Constantinides, "Efficient FPGA Implementation of Digit Parallel Online Arithmetic Operators," in *International Conference on Field Programmable Technology (FPT)*, 2014.

[4] Y. Zhao, J. Wickerson, and G. A. Constantinides, "An Efficient Implementation of Online Arithmetic," in *International Conference on Field Programmable Technology (FPT)*, 2016.

[5] H. Li, J. J. Davis, J. Wickerson, and G. A. Constantinides, "ARCHITECT: Arbitrary-precision Constant-hardware Iterative Compute," in *International Conference on Field Programmable Technology (FPT)*, 2017.

[6] ——, "Digit Elision for Arbitrary-accuracy Iterative Computation," in *IEEE Symposium on Computer Arithmetic (ARITH)*, 2018.

[7] M. Benzi, T. M. Evans, S. P. Hamilton, M. L. Pasini, and S. R. Slattery, "Analysis of Monte Carlo-accelerated Iterative Methods for Sparse Linear Systems," *Numerical Linear Algebra with Applications*, vol. 24, no. 3, 2017.

[8] J. Sun, G. D. Peterson, and O. O. Storaasli, "High-performance Mixed-precision Linear Solver for FPGAs," *IEEE Transactions on Computers*, vol. 57, no. 12, 2008.

[9] Q. Liu, R. Sang, and Q. Zhang, "FPGA-based Acceleration of Davidon-Fletcher-Powell Quasi-Newton Optimization Method," *Transactions of Tianjin University*, vol. 22, no. 5, 2016.

[10] 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 International Symposium on Field-programmable Custom Computing Machines (FCCM)*, 2009.

[11] D. Boland and G. A. Constantinides, "An FPGA-based Implementation of the MINRES Algorithm," in *International Conference on Field-programmable Logic and Applications (FPL)*, 2008.

[12] G. Constantinides, A. Kinsman, and N. Nicolici, "Numerical Data Representations for FPGA-based Scientific Computing," *IEEE Design & Test of Computers*, vol. 28, no. 4, 2011.

[13] D. H. Bailey, R. Barrio, and J. M. Borwein, "High-precision Computation: Mathematical Physics & Dynamics," *Applied Mathematics Computation*, vol. 218, no. 20, 2012.

[14] D. H. Bailey and J. M. Borwein, "High-precision Arithmetic in Mathematical Physics," *Mathematics*, vol. 3, no. 2, 2015.

[15] B. Serpette, J. Vuillemin, and J.-C. Hervé, "BigNum: A Portable and Efficient Package for Arbitrary-precision Arithmetic," Digital Paris Research Laboratory, Tech. Rep., 1989.

[16] MPFR, "The GNU MPFR Library," http://www.mpfr.org, 2017.

[17] F. Johansson, "Arb: Efficient Arbitrary-precision Midpoint-radius Interval Arithmetic," *IEEE Transactions on Computers*, vol. 66, no. 8, 2017.

[18] F. de Dinechin and B. Pasca, "Designing Custom Arithmetic Data Paths with FloPoCo," *IEEE Design & Test of Computers*, vol. 28, no. 4, 2011.

[19] X. Fang and M. Leeser, "Open-source Variable-precision Floating-point Library for Major Commercial FPGAs," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 9, no. 3, 2016.

[20] 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*, vol. 64, no. 2, 2017.

[21] P. Cégielski and D. Richard, "Decidability of the Theory of the Natural Integers with the Cantor Pairing Function and the Successor," *Theoretical Computer Science*, vol. 257, no. 1-2, 2001.

[22] P. Tu and M. D. Ercegovac, "Design of On-line Division Unit," in *IEEE Symposium on Computer Arithmetic (ARITH)*, 1989.

[23] S. F. Obermann and M. J. Flynn, "Division Algorithms and Implementations," *IEEE Transactions on Computers*, vol. 46, no. 8, 1997.

[24] 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.

[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*, vol. 36, no. 2, 2012.

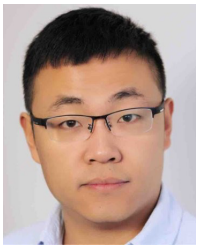
[26] E. Cheney and D. Kincaid, *Numerical Mathematics and Computing*. Nelson Education, 2012.

[27] C. T. Kelley, *Iterative Methods for Linear and Nonlinear Equations*. Society for Industrial and Applied Mathematics, 1995.

[28] K. Javeed, X. Wang, and M. Scott, "Serial and Parallel Interleaved Modular Multipliers on FPGA Platform," in *International Conference on Field-programmable Logic and Applications (FPL)*, 2015.

[29] 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*, vol. 19, no. 10, 2011.

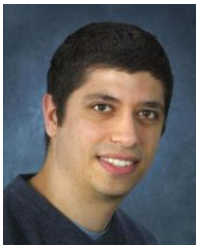
[30] A. Landy and G. Stitt, "Revisiting Serial Arithmetic: A Performance and Tradeoff Analysis for Parallel Applications on Modern FPGAs," in *IEEE International Symposium on Field-programmable Custom Computing Machines (FCCM)*, 2015.



He Li is a PhD student in the Department of Electrical and Electronic Engineering at Imperial College London. He received the MS degree from the Department of Microelectronics at Tianjin University in 2016. His main research interests are FPGA arithmetic, custom computing and hardware security. He received the Best Paper Presentation Award at FPT 2017.

James J. Davis is a Research Fellow in the Department of Electrical and Electronic Engineering's Circuits and Systems group at Imperial College London. He received a PhD in Electrical and Electronic Engineering from Imperial College London in 2016. His research is focussed on the exploitation of FPGA features for cutting-edge applications, driving up performance, energy efficiency and reliability. Dr Davis serves on the technical programme committees of the four top-tier reconfigurable computing conferences (FPGA, FCCM, FPL and FPT) and is a multi-best paper award recipient. He is a Member of the IEEE and the ACM.

John Wickerson received a PhD in Computer Science from the University of Cambridge in 2013. He is a Lecturer in the Department of Electrical and Electronic Engineering at Imperial College London. His research interests include high-level synthesis, the design and implementation of programming languages and software verification. He is a Senior Member of the IEEE and a Member of the ACM.



George A. Constantinides received the PhD degree from Imperial College London in 2001. Since 2002, he has been with the faculty at Imperial College London, where he is currently Professor of Digital Computation and Head of the Circuits and Systems research group. He was General Chair of the ACM/SIGDA International Symposium on Field-programmable Gate Arrays in 2015. He serves on several programme committees and has published over 200 research papers in peer-refereed journals and international conferences. Prof. Constantinides is a Senior Member of the IEEE and a Fellow of the British Computer Society.