ARCHITECT: **Ar**bitrary-precision Constant-**h**ardware **Ite**rative Compu**te**

He Li, James J. Davis, John Wickerson and George A. Constantinides
Department of Electrical and Electronic Engineering
Imperial College London, London, SW7 2AZ, United Kingdom
{h.li16, james.davis, j.wickerson, g.constantinides}@imperial.ac.uk

***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: armed 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 iteration index increasing in lockstep. Thus, 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. For the execution of 100 iterations of the Jacobi method, we obtain a 1.60× increase in frequency and 15.7× LUT and 50.2× flip-flop reductions over a 2048-bit parallel-in, serial-out traditional arithmetic equivalent, along with 46.2× LUT and 83.3× flip-flop decreases versus the state-of-the-art online arithmetic implementation.**

I. Introduction & Motivation

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, $\boldsymbol{r}$, is obtained as k approaches infinity, *i.e.*

$$\boldsymbol{r} = \lim_{k\to\infty} \Pi\Big(\boldsymbol{x}^{(k)}\Big)$$

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].

In practice, these calculations are often implemented using finite-precision approximations such as that shown in Algorithm 1, where $\mathbb{FP}_P$ denotes some finite-precision datatype, P is a measure of its precision (usually bit-width) and $\hat{f}$ is a finite-precision equivalent of f. 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.

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: $\left\|\Pi\Big(\hat{\boldsymbol{x}}^{(K)}\Big) - \boldsymbol{r}\right\| < \eta$

As a simple demonstration of this problem, suppose we wish to solve the toy equation

$$\begin{pmatrix} 1 & 0.5 \\ 0.5 & 1 \end{pmatrix} \begin{pmatrix} x \\ x \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

for x using the Jacobi method [2]. This necessitates computing

$$x^{(k+1)} = 1 - 0.5x^{(k)}$$

starting from $x^{(0)} = 0$, which yields the sequence 0, 1, 0.5, 0.75, 0.625 and so on.

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. Figure 1 (left) shows the order in which the digits are calculated when the precision is fixed to four decimal places: iteration-by-iteration, least-significant digit (LSD) first. Choosing the right precision *a priori* is difficult. If it is too high, we waste time. For instance, it is unnecessary to calculate $x^{(4)}$ to beyond three decimal places because every further digit will be 0. However, if the precision is too low, the sequence may never converge; if all calculations were truncated at the decimal point, we would instead obtain the sequence 0, 1, 0, 1, 0, *etc.*

Our proposal, illustrated in Figure 1 (right), avoids the need to answer the aforementioned question entirely. The digits are calculated in a diagonal pattern, sweeping through iterations and decimal places simultaneously. From the pattern, we can infer that 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

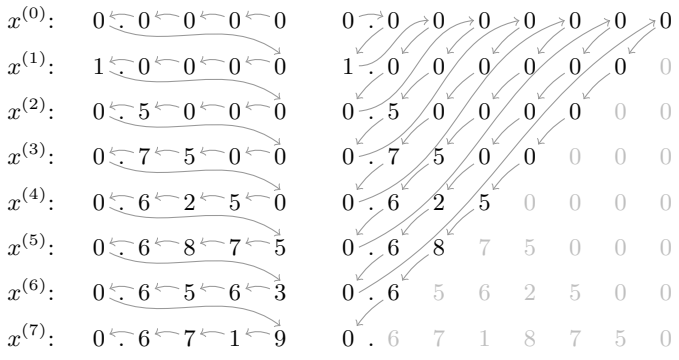

Fig. 1. Alternative digit-calculating strategies for the same result. Left: iteration-by-iteration, LSD first. Right: our proposal, MSD first.

beforehand, but requires the ability to calculate from most-significant digit (MSD) first, a facility provided through the use of *online arithmetic* [3]. While general-purpose processors featuring traditional, LSD-first arithmetic units exhibit inefficiency for the realisation of online arithmetic, FPGAs represent ideal platforms for the implementation of such MSD-first operations thanks to their completely flexible fabrics.

The proposed architecture, coined ARCHITECT, is the first to allow the runtime adaption of both precision and iteration count for iterative algorithms implemented in hardware. More specifically, we make the following novel contributions:

- 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.
- Qualitative and quantitative performance and scalability comparisons against traditional and state-of-the-art online arithmetic FPGA implementations.

The designs presented and evaluated in this paper 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 [4]. Implementations relying on traditional arithmetic—whether digit-serial or -parallel—enforce compile-time determination of precision; for digit-parallel designs this affects their area, while for digit-serial it is one of the factors affecting algorithm runtime. Realtime tuning of precision in iterative calculations was enabled through the use of online arithmetic in recent work [5], 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 ARITHMETIC PARADIGMS FOR ITERATIVE ALGORITHMS.

Name	Area scales with		Runtime scales with	
	Prec.	Iter. limit	Prec.	Iter. limit
Digit-parallel	✔	✘	✘	✔ unbounded
Digit-serial	✘	✘	✔ bounded	✔ unbounded
Zhao *et al.* [5]	✘	✔	✔ unbounded	✘
ARCHITECT	✘	✘	✔ unbounded	✔ unbounded

TABLE II
COMPARISON OF ARBITRARY-PRECISION ARITHMETIC TECHNIQUES.

Name	Level	Precision set per calc.	Iteration limit set per calc.
MPFR [9]	Software	Before	During
FloPoCo [10], *etc.*	Hardware	Before	During
Mixed-precision [12]	Hardware	Before	During
Zhao *et al.* [5]	Hardware	During	Before
ARCHITECT	Hardware	During	During

A. Arbitrary-precision Arithmetic

Calculations requiring very high precision are of increasing prevalence; in many cases, double- or even quadruple-precision floating point are insufficient [6]. Simulations of supernovae and electromagnetic scattering necessitate hundreds of digits, while thousands are now used for ordinary differential equations [7]. Poisson equation and Riemann zeta function computations frequently operate to tens or hundreds of thousands of digit precisions [8].

Many software libraries have been developed for arbitrary-precision arithmetic. The *de facto* standard is MPFR, which guarantees correct rounding to any requested number of bits [9]. Arbitrary-precision operations implemented on FPGAs have seen increasing attention in recent years; they provide flexibilities not available on other platforms, allowing for the implementation of bespoke designs with many precision and performance tradeoffs. Libraries including FloPoCo [10] and VFLOAT [11], 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. 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 [12]. Zhao *et al.*'s work enables arbitrary-precision computation but, as mentioned previously, requires compile-time determination of iteration count [5]. Table II presents a side-by-side comparison of these techniques and their features with ARCHITECT, the only entry supporting the determination of precision and iteration count *after each calculation has commenced*.

B. Online Arithmetic

Online arithmetic operators achieve left-to-right (MSD-first) computation through the use of redundancy in their number

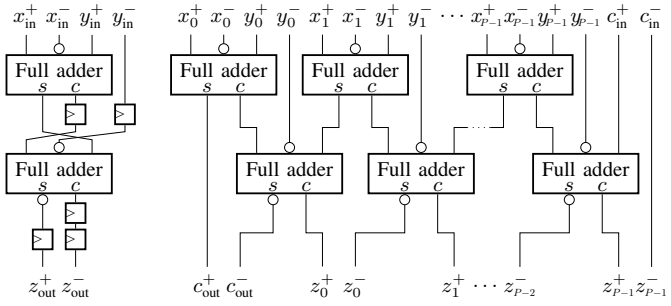

Fig. 2. Radix-2 online adders. Left: serial. Right: parallel.

representation [3]. With the radix-2 signed-digit representation we employ, the i^{th} digit of a number x, x_i, lies in $\{-1, 0, 1\}$ [13]. In hardware, each x_i corresponds to a pair of bits, x_i^+ and x_i^-, selected such that $x_i = x_i^+ - x_i^-$.

1) 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 (left), from most to least significant [3]. 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_{OA}. Duplication of the 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 Figure 2 (right) [3]. 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 [14]. This indicates the adder's suitability for the construction of more complex online operators and that its maximum frequency is independent of precision [5].

2) Online Multiplication: Algorithm 2 illustrates classic radix-2 online multiplication: a process that operates in serial-in, serial-out fashion [3]. 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

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

during cycle j. Digit-selection function *sel* [5], which can be implemented with a four-digit multiplexer, serves to determine the digits of output z. This is defined to be

$$\text{sel}(\boldsymbol{v}) = \begin{cases} 1 & \text{if } \boldsymbol{v} \geq 1/2 \\ 0 & \text{if } -1/2 \leq \boldsymbol{v} \leq 1/4 \\ -1 & \text{otherwise} \end{cases}$$

z_j is produced at cycle $j+3$ since $\delta_{\text{OM}} = 3$. A 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, which must be fixed in advance.

III. Proposed ARCHITECTure

Using classic 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.

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: $\quad \boldsymbol{y} \leftarrow \boldsymbol{y} \parallel y_j$
4: $\quad \boldsymbol{v} \leftarrow 2\boldsymbol{w} + 2^{-3}(\boldsymbol{x}y_j + \boldsymbol{y}x_j)$
5: $\quad z_{j-3} \leftarrow \text{sel}(\boldsymbol{v})$
6: $\quad \boldsymbol{w} \leftarrow \boldsymbol{v} - z_{j-3}$
7: $\quad \boldsymbol{x} \leftarrow \boldsymbol{x} \parallel x_j$
8: **end for**

Output: serially generated digits z

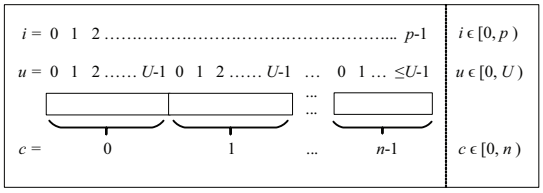

Fig. 3. Indexing of digits and chunks within a p-digit number. i indexes all digits, while those of each of its n chunks, indexed c, are indexed with u.

A. Digit-vector Storage

Classic 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, however, this is unsuitable; we must use 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. When performing iterative calculations, independent digit vectors exist for each step, thus their indexing requires three separate variables: iteration k, chunk $c \in [0, n)$ and chunk digit $u \in [0, U)$. The relationships between c, n, u, U and overall digit index i are shown visually for a single p-digit number in Figure 3.

Since ARCHITECT requires iteration index k and precision p to both vary non-monotonically as time progresses, it is necessary to uniquely encode a one-to-one mapping from two-dimensional iteration and chunk index pair (k, c) into one-dimensional time. We use a Cantor pairing function (CPF), a bijection from $\mathbb{N}^2$ onto $\mathbb{N}$, for this purpose, defined to be

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

The function's bijectivity means that it is *injective* and *surjective*, both crucial properties for ARCHITECT. Unlike classic row- or column-major indexing, CPFs' injectivity allows both dimensions to grow without bound while providing a unique result for every (k, c). The operation of our CPF is demonstrated visually in Figure 4; 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. Its surjectivity ensures that every $\text{cpf}(k, c)$ is produced by some (k, c), thus enabling the most efficient use of the available memory.

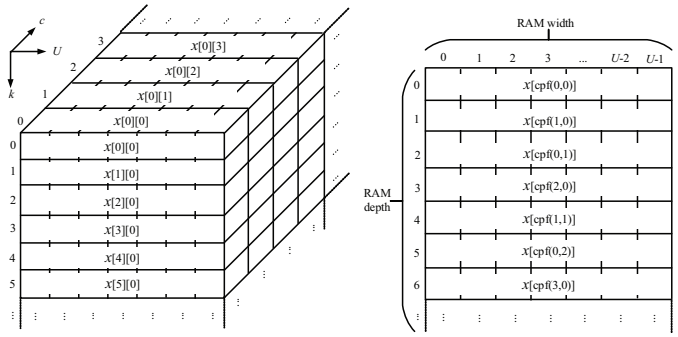

Fig. 4. Operation of our Cantor pairing function, showing the transformation of a three-dimensional array growing with both iteration and chunk indices k and c (left) to a structure growing only in a single dimension (right).

Algorithm 3 Loop body of arbitrary-precision radix-2 online multiplication, performed p times for p-digit precision.

Inputs: digit x_{in}, digit y_{in}, iteration index k, number of chunks n, chunk digit index u

1: $\boldsymbol{y}[\text{cpf}(k, n-1)][u] \leftarrow y_{\text{in}}$
2: **for** $c = n-1$ **to** 0 **do**
3: $\quad \boldsymbol{v}[\text{cpf}(k,c)] \leftarrow 2\boldsymbol{w}[\text{cpf}(k,c)] + 2^{-3}(\boldsymbol{x}[\text{cpf}(k,c)]y_{\text{in}} + \boldsymbol{y}[\text{cpf}(k,c)]x_{\text{in}})$
4: $\quad$ **if** $c > 0$ **then**
5: $\quad\quad \boldsymbol{w}[\text{cpf}(k,c)] \leftarrow \boldsymbol{v}[\text{cpf}(k,c)]$
6: $\quad$ **end if**
7: **end for**
8: $z_{\text{out}} \leftarrow \text{sel}(\boldsymbol{v}[\text{cpf}(k,0)])$
9: $\boldsymbol{w}[\text{cpf}(k,0)] \leftarrow \boldsymbol{v}[\text{cpf}(k,0)] - z_{\text{out}}$
10: $\boldsymbol{x}[\text{cpf}(k,n-1)][u] \leftarrow x_{\text{in}}$

Output: digit z_{out}

We are now in a position to rewrite the body of Algorithm 2 such that it can compute results to arbitrary precision. These transformed steps are shown in Algorithm 3. 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 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.

B. 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 are calculated across iterations. δ impacts us in two ways:

- As exemplified in Algorithm 2, 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 within the same iteration can be calculated indefinitely, while those lying across iterations must be sequenced such that they obey these δ-imposed limitations. The scheduling of output digit $z_i^{(k)}$ generation must therefore obey

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

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

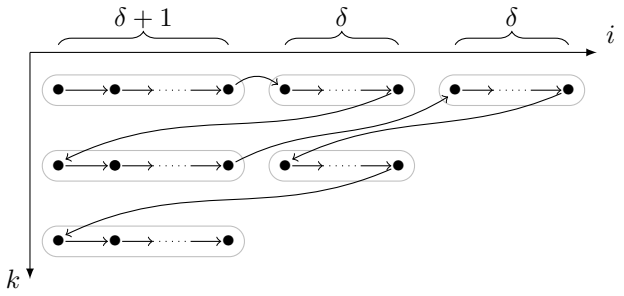

Fig. 5. Digit pattern for generic iterative computation using online operators, prioritising iteration count over precision.

While we have the choice of whether to prioritise iteration count or precision within these bounds, in this paper we always advance iteration index k as soon as possible. Thus, an appropriate mapping from the current to next digit respecting the aforementioned dependences is depicted in Figure 5. The figure reveals two distinct groupings of digits: we call the blocks of $\delta+1$ MSDs of each iteration *group 1*, while the sets of δ digits that follow each belong to *group 2*. $\delta+1$ digits make up each group 1 since output digit generation always lags input presentation by δ cycles; the computation of the $(k+1)^{\text{th}}$ iteration's first digit requires the prior generation of the k^{th} iteration's first $\delta+1$ digits, themselves necessitating the k^{th} iteration's first $2\delta+1$ input digits to compute. Following the calculation of the group 1 digits within each iteration, group 2s are processed 'downwards' and 'leftwards,' with slope dependent on δ and control snapping back to the first iteration once digit position $i = 0$ has been reached.

Given a particular (k, n, u), we can compute the subsequent (k', n', u') to realise this pattern with the finite-state machine (FSM) depicted in Figure 6, whose functionality is as follows.

- *Group 1 control*: Manages the propagation and storage of $\delta+1$ digits at the start of each new iteration. Since RAM width U is always greater than δ, a limitation we impose, the number of chunks $n = 1$ in group 1, thus digits can always be computed in successive clock cycles.
- *Group 2 control*: For δ digits' propagation and storage.
- *Iterative addition*: When $p \leq U$, ARCHITECT is able to perform p-digit additions in single clock cycles. However, when $p > U$, once each U-digit chunk has been computed, normal computation in group 2 must be halted for $n-1$ cycles to calculate one digit of the iterative algorithm's output [5]. Note that this loop cannot be unrolled since p is variable.

C. Required, Resultant and Maximum Iterations & Precision

In order to reach a result of required accuracy defined by its iteration index and precision $(K_{\text{req}}, P_{\text{req}})$, we must compute

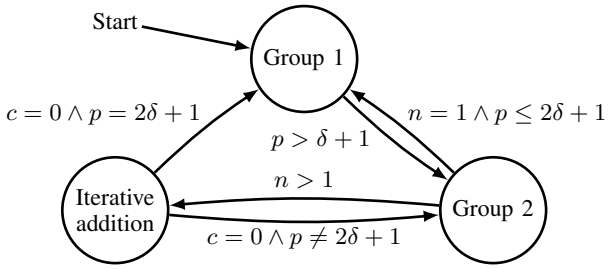

Fig. 6. ARCHITECT FSM for digit computation scheduling. Termination occurs either on demand or when the allocated RAM has been exhausted.

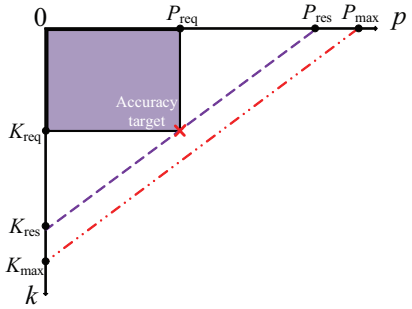

Fig. 7. Relationships of $(K_{\text{req}}, P_{\text{req}})$, K_{res}, P_{res}, K_{max} and P_{max}.

for at least K_{req} iterations and to at least P_{req}-digit precision. As shown in Figure 7, we define the number of iterations resulting from computation to accuracy target $(K_{\text{req}}, P_{\text{req}})$ as K_{res} and the precision of the first iteration's result—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 allocated memory.

From the pattern shown in Figure 5, we can deduce that the precision of the result in the k^{th} iteration, $p^{(k)}$, is given by

$$p^{(k)} = \begin{cases} \delta\left(\left\lceil \frac{P_{\text{req}}-1}{\delta} \right\rceil + K_{\text{req}} - 1 - k\right) + 1 & \text{if } k < K_{\text{req}} - 1 \\ P_{\text{req}} & \text{if } k = K_{\text{req}} - 1 \\ \delta\left(K_{\text{res}} - 1 - k\right) + 1 & \text{otherwise} \end{cases}$$

where K_{res} can be derived from $(K_{\text{req}}, P_{\text{req}})$ as

$$K_{\text{res}} = \begin{cases} \left\lceil \frac{P_{\text{req}}-1}{\delta} \right\rceil - 1 + K_{\text{req}} & \text{if } P_{\text{req}} > 1 \\ K_{\text{req}} & \text{if } P_{\text{req}} = 1 \end{cases}$$

and $P_{\text{res}} = p^{(0)}$, thus

$$P_{\text{res}} = \delta\left(\left\lceil \frac{P_{\text{req}} - 1}{\delta} \right\rceil + K_{\text{req}} - 1\right) + 1$$

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's results allow us to derive

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

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

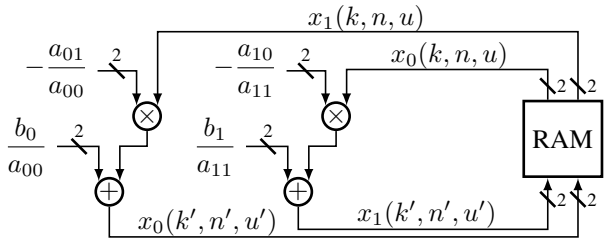

Fig. 8. Jacobi method datapath, highlighting digit-vector storage.

IV. BENCHMARK: JACOBI METHOD

In order to evaluate ARCHITECT, we implemented a widely used iterative algorithm, the Jacobi method, in hardware following the aforementioned principles. The Jacobi method seeks to solve the system of N linear equations $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$. If $\boldsymbol{A}$ is decomposed into diagonal and remainder components such that $\boldsymbol{A} = \boldsymbol{D} + \boldsymbol{R}$, $\boldsymbol{x}$ can be computed as

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

or, expressed in element-wise fashion, as

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

where k is the iteration 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 of $\boldsymbol{x}$; the calculation can therefore be parallelised by computing each $x_i^{(k+1)}$ independently. A convergence criterion, $\|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}\| < \eta$, is used in order to determine whether or not 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 in this paper for simplicity.

A metric used to quantify the sensitivity of a particular linear system to error is the *condition number* of $\boldsymbol{A}$ [15], 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 and therefore ill conditioned [16]. We can expect to need at least β additional digits of precision in order to compute a system with $\kappa(\boldsymbol{A}) = 2^{\beta}$ than would be required if $\kappa(\boldsymbol{A})$ were 1 [2].

Without loss of generality, the datapath developed to solve systems with dimensionality $N = 2$ is depicted in Figure 8, featuring ARCHITECT numerical operators described in Section III. 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 simple rearrangement transforms subtraction into addition.

A. Computation Time

Given a particular accuracy target $(K_{\text{req}}, P_{\text{req}})$, 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 can be broken down into the following three components such that $T = T_1 + T_2 + T_3$.

- *Digit generation in groups 1 and 2*: The latency for a single output digit's computation via iterative addition increases with the number of chunks within the given iteration, $n^{(k)}$. Across all iterations performed, this is

$$T_1 = \sum_{k=0}^{K_{\text{res}}-1} n^{(k)} \left(p^{(k)} - \frac{U\left(n^{(k)} - 1\right)}{2} \right)$$

  where $n^{(k)} = \left\lceil p^{(k)} / U \right\rceil$.
- *Initial online delay*: We must wait δ clock cycles before each iteration's result begins to appear, thus the delay across all iterations is simply

$$T_2 = \delta K_{\text{res}}$$

  where δ is the combined online delay of all operators within the datapath. For our Jacobi benchmark, $\delta = 5$.
- *Serial online adder*: Our datapath includes a serial online adder with $\delta_{OA} = 2$. When switching between iterations, the adder requires two cycles to recalculate the preceding iteration's residuals in order to produce a new digit [3]. Therefore, the online delay ensures that the calculated digit aligns with its truncated digit vector. For this,

$$T_3 = K_{\text{res}}^2 - K_{\text{res}} + 2K_{\text{req}} - 2$$

V. Evaluation

Experiments were performed to investigate how ARCHITECT scales and performs versus competing arithmetic implementations, both traditional (LSB-first) and online, using the Jacobi method as a case study. A Xilinx Virtex UltraScale FPGA (XCVU190-FLGB2104-3-E) was targetted, with compilation performed using Vivado 16.4. The correctness of results obtained in hardware was verified via comparison against those produced by a golden model executed in software.

The closest work to ARCHITECT is that presented by Zhao *et al.* [5], which we compare against directly. For comparison against traditional arithmetic, we chose to implement parallel-in, serial-out (PISO) operators. PISO sits at the midpoint between fully serial (SISO) and parallel (PIPO) in terms of area and performance [17]. With increase in precision P—which, for traditional arithmetic, can solve problems for which $P_{\text{req}} \leq P$—PISO suffers less from area growth and operating frequency f_{max} degradation than PIPO [18] while also being dramatically faster than SISO [19]. While we focus exclusively on hardware implementations here, the limitations revealed for PISO apply equally to software libraries since precision must be chosen prior to the iterative algorithm's commencement.

A. Qualitative Performance Comparison

To evaluate performance, 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, suggesting that higher precision P_{req} 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 $P_{\text{req}} = 32$, 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. Note that, for ARCHITECT, we used RAM size $(U, D) = \left(64, 2^{10}\right)$ and that reported latencies used frequencies taken from Section V-C.

The changes in $\kappa(\boldsymbol{A}_m)$ and P_{req} by m are shown in Figure 9 (left and mid-left, respectively). We can see that $\kappa(\boldsymbol{A}_m)$ increases exponentially with m, while P_{req} scales approximately linearly. Figure 9's mid-right plot captures the latency ratio between ARCHITECT and 32-bit PISO necessary to compute results for matrices with low m. Here, PISO can be said to have over-budgeted precision; $P > P_{\text{req}}$ and, therefore, results take longer to compute than had a smaller P been chosen in advance. In *region A*, for the most well-conditioned matrices, ARCHITECT takes less time to reach the target $(K_{\text{req}}, P_{\text{req}})$, while in *region B* 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 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 Figure 9 (right) with $P = 8$, ARCHITECT can extend into *region C*, 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 conclude, therefore, that ARCHITECT requires less time than PISO to generate results either when P_{req} is small and convergence is fast or when P_{req} is too large for PISO to ever converge.

B. Area & Frequency Scalability

Our implementational results are presented in Figure 10, including area, maximum operating frequency f_{max} and the corresponding limits on iteration count K_{max} and precision P_{max}. Each of the three plots features D, the RAM depth used for storage of each digit vector, on the x-axis. LUT and flip-flop (FF) usage are not shown since the numbers are insignificant compared to BRAM—from 0.0771% to 0.344% for LUTs and 0.0198% to 0.0392% for FFs for the smallest ($D = 2^{10}$) and largest ($D = 2^{18}$) designs implemented. Memory usage 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. The small increases in non-RAM resources noted can be attributed to the additional control logic and multiplexing required to address larger memories. With 56.4% of BRAMs allocated on our target FPGA, we were able to reach $K_{\text{max}} = 724$ and $P_{\text{max}} = 5784$. The f_{max} plot shows that

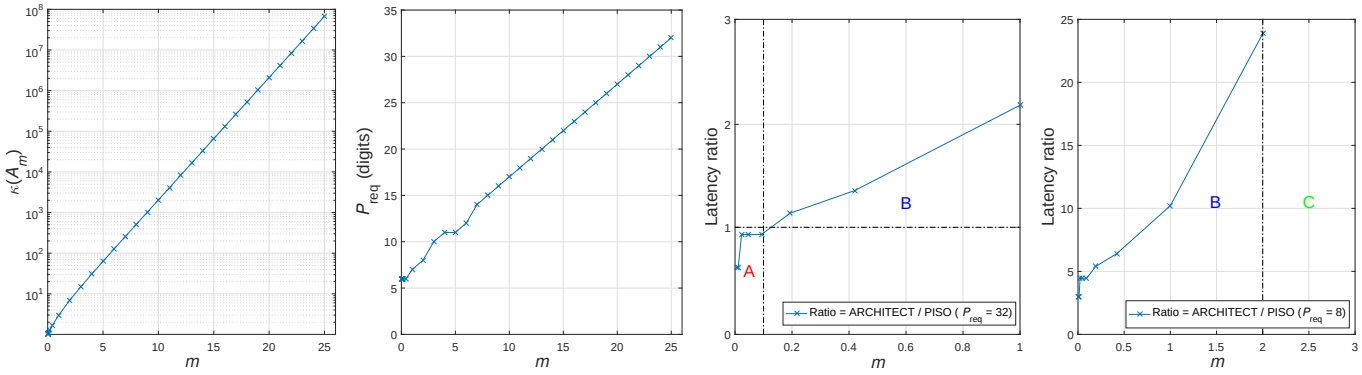

Fig. 9. Qualitative comparison between ARCHITECT and traditional arithmetic (PISO). Across a collection of matrices, higher m leads to larger $\kappa(\boldsymbol{A}_m)$, for which results with greater precision P_{req} are required to solve the linear system $\boldsymbol{A}_m\boldsymbol{x} = \boldsymbol{b}$.

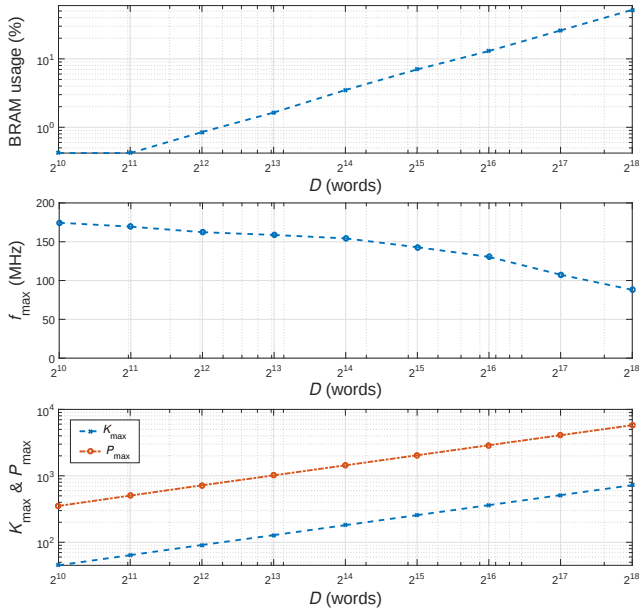

Fig. 10. Resource usage and performance of ARCHITECT Jacobi benchmark versus RAM depth D. Area is reported in terms of BRAMs only; LUT and FF usage were below 1% for all design points.

our implementations are able to run at between 180 MHz, for the smallest D tested, to just below 100 MHz for the largest.

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 usage. A comparison between $U = 8$ and $U = 64$ with the same D, in this case 2^{10}, is shown in Table III to exemplify this.

C. Quantitative Area & Frequency Comparison

In order to compare the resource usage and f_{max} of ARCHITECT against its competitors, we now assume that we wish

TABLE III
AREA-SPEED TRADEOFF VIA SELECTION OF U.

Metric	$U = 8$	$U = 64$
LUTs	828 (0.0771%)	2475 (0.231%)
FFs	408 (0.0198%)	936 (0.0436%)
BRAMs	16 (0.423%)	58 (1.54%)
f_{max} (MHz)	180	175
Iterative addition latency (cycles)	$\lceil p^{(k)}/8 \rceil$	$\lceil p^{(k)}/64 \rceil$

to compute a result to an accuracy target of $(K_{\text{req}}, P_{\text{req}}) = (100, 2^{11})$ using the Jacobi method. Thus, at its 100$^{\text{th}}$ iteration, we wish to obtain a result with 2048-digit precision. Using $U = 8$, for ARCHITECT, the resultant iteration count $K_{\text{res}} = 509$ and precision $P_{\text{res}} = 2546$ and, to successfully perform computation to $(K_{\text{req}}, P_{\text{req}})$, 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.

Figure 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.* [5]. Most strikingly, the latter demonstrates area inefficiency, with resource usage scaling linearly with iteration count K_{req}; ARCHITECT consumes 46.2× fewer LUTs and 83.3× fewer FFs than Zhao *et al.*'s proposal requires in order to execute 100 iterations of the Jacobi method. f_{max} is comparable between the two since the underlying arithmetic is largely equivalent, although ARCHITECT's is slightly superior. For PISO, we can see that, while its f_{max} is initially much higher—up to 322 MHz for $P_{\text{req}} = 2^4$—than ARCHITECT's, it falls as P_{req} increases; the crossover occurs at $P_{\text{req}} \approx 800$. With high precision requirements, such as 2^{10}- and 2^{11}-digits, ARCHITECT is able to outperform its PISO counterpart in terms of f_{max} by factors of 1.17 and 1.60, respectively. Corresponding decreases in LUT and FF usage were also found; when computing to $P_{\text{req}} = 2^{10}$, ARCHITECT consumes 6.87× and 25.3× fewer of each than PISO, while for 2^{11} these factors increase to 15.7

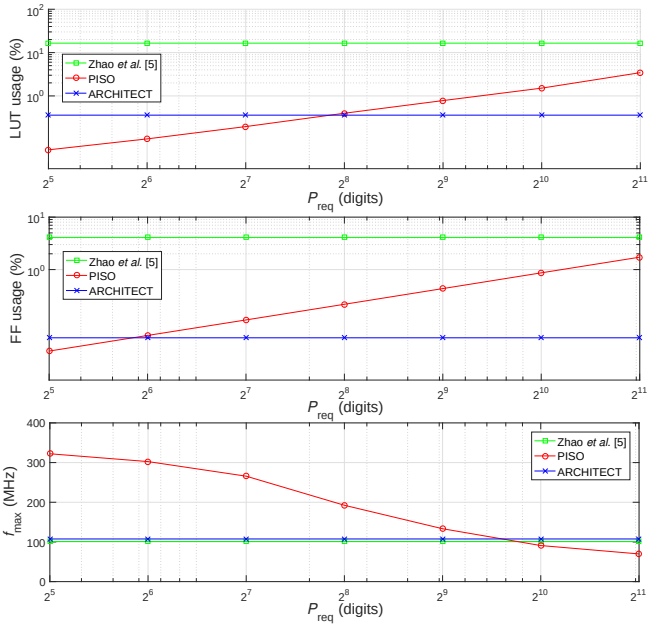

Fig. 11. Resource usage and performance comparison of competing Jacobi designs versus required precision P_{req}.

and 50.2. Since the proposed design is able to calculate to any $(K_{\text{req}}, P_{\text{req}})$ up to $(100, 2^{11})$, its area and f_{max} are constant.

VI. Conclusion & Future Work

In this paper, we proposed the first hardware architecture capable of executing an iterative algorithm to produce results of arbitrary accuracy by combining increasing iteration count with precision while utilising constant compute resources. We named this technique ARCHITECT, for **Ar**bitrary-precision **C**onstant-**h**ardware **Ite**rative **C**ompu**t**e. ARCHITECT 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.

We evaluated ARCHITECT using the Jacobi method in order to establish its accuracy, scalability and suitability in numerical analysis. This benchmark showcased the key advantage of our approach: removing the burden of having to determine and fix the precisions of arithmetic operators in advance. By doing so, we showed that datapaths constructed from arbitrary-precision ARCHITECT operators are superior to their traditional arithmetic equivalents in scenarios where either the latter's precisions are overly high for the problems being solved or too low for results to converge at all. A single ARCHITECT datapath, meanwhile, is able to compute results to any accuracy, with the only limit being imposed by the available RAM. Finally, our experiments showed that ARCHITECT is capable of achieving a 1.60× operating frequency boost and 15.7× LUT and 50.2× FF reductions over 2048-bit parallel-in serial-out arithmetic, along with 46.2× LUT and 83.3× FF decreases versus the state-of-the-art online arithmetic implementation, when executing 100 Jacobi iterations.

In the future, we will develop more efficient computation patterns, focussing on the avoidance of the recomputation of 'don't change' digits in algorithms' later iterations while also leaving 'don't care' digits in earlier iterations uncomputed. We will extend our benchmarking to cover additional iterative algorithms. Finally, we envisage that the arbitrary-precision computation enabled by ARCHITECT can be combined with high-level synthesis to enable faster hardware specialisation.

Acknowledgements

This work was supported by the EPSRC (grant numbers EP/P010040/1 and EP/K034448/1), Imagination Technologies and the Royal Academy of Engineering. Supporting data for this paper are available online at `https://doi.org/10.5281/zenodo.998249`.

References

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

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

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

[4] 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.

[5] Y. Zhao, J. Wickerson, and G. A. Constantinides, "An Efficient Implementation of Online Arithmetic," in *International Conference on Field-programmable Technology*, 2016.

[6] 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.

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

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

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

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

[11] 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.

[12] 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.

[13] B. Parhami, "On the Implementation of Arithmetic Support Functions for Generalized Signed-digit Number Systems," *IEEE Transactions on Computers*, vol. 42, no. 3, 1993.

[14] K. Shi, D. Boland, and G. A. Constantinides, "Efficient FPGA Implementation of Digit Parallel Online Arithmetic Operators," in *International Conference on Field-programmable Technology*, 2014.

[15] 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*, vol. 2, 1995.

[16] 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.

[17] 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*, 2015.

[18] 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 Systems*, vol. 19, no. 10, 2011.

[19] 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.