An Efficient Implementation of Online Arithmetic

Yiren Zhao *, John Wickerson †, George A. Constantinides ‡
Department of Electrical and Electronic Engineering, Imperial College London, SW7 2AZ, United Kingdom
yiren.zhao13, j.wickerson, g.constantinides@imperial.ac.uk

***Abstract*—We propose the first hardware implementation of standard arithmetic operators – addition, multiplication, and division – that utilises constant compute resource but allows numerical precision to be adjusted arbitrarily at run-time. Traditionally, precision must be set at design-time so that addition and multiplication, which calculate the least significant digit (LSD) of their results first, and division, which calculates the most significant digit (MSD) first, can be chained together. To get around this, we employ *online operators*, which are always MSD-first, and thus allow successive operations to be pipelined. Even online operators require precision to be fixed at design-time because multiplication and division traditionally involve parallel adders. To avoid this, we propose an architecture, which we have implemented on an FPGA, that reuses a fixed-precision adder and stores residues in on-chip RAM. As such, we can use a single piece of hardware to perform calculations to any precision, limited only by the availability of on-chip RAM. For instance, we obtain an 8x speed-up, compared to the parallel-in-serial-out (PISO) fixed-point method, when executing 100 iterations of Newton's method at a precision of 64 digits, while the product of circuit area and latency stays comparable.**

I. Introduction

Traditional fixed-point arithmetic operators, both parallel and serial, require precision to be specified at design-time. These operations fall into two categories; some, such as addition and multiplication, proceed from the least significant digit (LSD) to the most significant digit (MSD), while others, such as division, compute from the MSD to the LSD. As a result of this difference, fixed-point arithmetic is sometimes forced to perform word-by-word computations: operators may need to stall until all digits in a word have been calculated. For example, if a 32-bit multiplication feeds a 32-bit division for serial fixed-point arithmetic, the divider would only be able to take the input once all 64 bits of the intermediate product have been calculated by the multiplier.

Online arithmetic unifies all arithmetic operations in an MSD-first fashion [2]. Existing implementations of parallel and serial online arithmetic require precision to be confirmed at design-time [3]. Nonetheless, as illustrated in Figure 1, serial online arithmetic supports pipelining: an output digit can be streamed into the next operator before all the digits in that word have been generated.

In our work, we focus on serial online operators. Inheriting from classic online arithmetic, we employ the MSD-first computation order to reduce the number of computation clock cycles compared to serial fixed-point arithmetic [3]. On top of this, our novel hardware design can provide results with arbitrary precision at run-time. The algorithms of classic online multiplication and division compute residue terms using parallel online adders and produce a digit of product/quotient

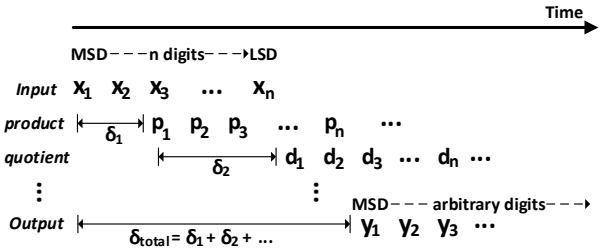

Fig. 1: Datapath of chained online operators; δ represents online delay; input **x** is used to produce a product **p**, which is then used to produce a quotient **q**; **y** is the final result.

based on these partial sums [13]. Classic architectures for serial online operators, although producing results in a digit-by-digit fashion, require precisions to be specified at design-time because they use parallel online adders on these residue terms. Our new design reuses a fixed-precision parallel adder and stores residues in on-chip RAM. This hardware reuse enables results to be produced with arbitrary precision but consumes only a fixed amount of computational hardware. For any given iterative algorithm with a known number of iterations, we could connect online operators and generate results to any precision at run-time with constant hardware costs, and the maximum achievable precision is only limited by the availability of on-chip RAM.

In Section III, we give an overview of online arithmetic, and discuss our optimized hardware architecture. In Section V, we summarize online arithmetic's performance on Newton's method both empirically and analytically to demonstrate its advantages on iterative operations with many iterations and high precision requirements. At a low iteration count and a precision of 64 digits, empirical results suggest our implementation of online arithmetic is 2.95x faster than parallel-in-serial-out (PISO) arithmetic. On the other hand, we demonstrate analytically that our design is also preferable if iteration count is large: we obtain a 1.7x speed increase at an iteration count of 100 but only 8 digits precision.

We make the following contributions in this paper:

- The first architecture of online arithmetic where the compute resource utilization does not grow with precision even for multiplication and division, opening the door to run-time tuning of arithmetic precision.
- A quantitative analysis of the overhead of online arithmetic on a modern FPGA device.
- A demonstration that for high precision or high iteration counts, online arithmetic is superior to standard fixed-point arithmetic.

II. Related work

In the past, ASIC designs of parallel online operators have proven to be advantageous in speed but to consume more hardware [4]. An example is the PasteUp system, which is a VLSI implementation of radix-4 online arithmetic [5]. Online arithmetic can be viewed as a method for trading an increase in circuitry for a reduction in latency.

Recently, Shi *et al*. have shown parallel online arithmetic to have an additional advantage due to overclocking [11]. Traditional fixed-point arithmetic is vulnerable to timing violations since errors always accumulate from LSD to MSD. Shi *et al*. have shown that online arithmetic is generally more resistant to timing errors at overcloking. The accumulation of timing error is circumvented because of the MSD-first computation pattern. There is also work focusing on optimizing architectures of parallel online operators on FPGAs to reduce their area overhead [10]. Researchers have demonstrated methods of mapping an online adder block to a four 6-input LUTs' slice on Xilinx Virtex-6 FPGAs utilizing the built-in carry resources. In addition, there is an efficient implementation of online multiplication between a digit and a digit-vector on FPGAs using only one logic element. These low-level hardware optimizations significantly reduce the area overhead of parallel online arithmetic [10].

Parallel online arithmetic requires more hardware resources compared to the traditional fixed-point approach [10]. To mitigate this area overhead of parallel online arithmetic, one practical solution is to use serial online arithmetic instead. Serial online arithmetic consumes more computational clock cycles than parallel online arithmetic but requires less circuitry. Serial online arithmetic has been exploited in control and signal processing, and has shown promising performance in scenarios requiring high-speed computations [1], [8]. Serial online arithmetic has been found to satisfy the needs of real-time control systems, where small circuitry, low power consumption and high computation speed are required [1]. In signal processing, analog-to-digital and digital-to-analog converters consume digits in a MSD-first fashion, and Natter *et al*. have shown the potential to embed online arithmetic behind these converters to increase processing speed [8]. However, these implementations are all application-specific and follow the classic online architecture requiring *a priori* determination of precision.

III. An overview of Online Arithmetic

Online arithmetic has two unique features. Firstly, the digits are taken from the MSDs of input operands and the resulting output always contains an initial delay, named the *online delay* and denoted by δ. Secondly, online arithmetic employs a redundant number representation system. To simplify matters, in this paper, all implementations of online operators only involve signed digit (SD) representation on a redundant digit set of $\{-1, 0, 1\}$. Using SD representation, we write x_j to represent the jth digit of a number, and $x_j = x_j^+ - x_j^-$, where x_j^+ and x_j^- are both single bits.

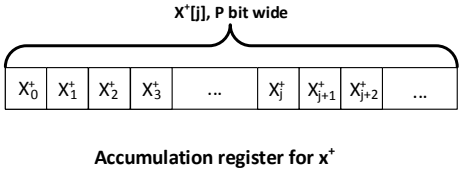

Fig. 2: Accumulation register for storing upper half of a single digit-vector $x^+[j]$.

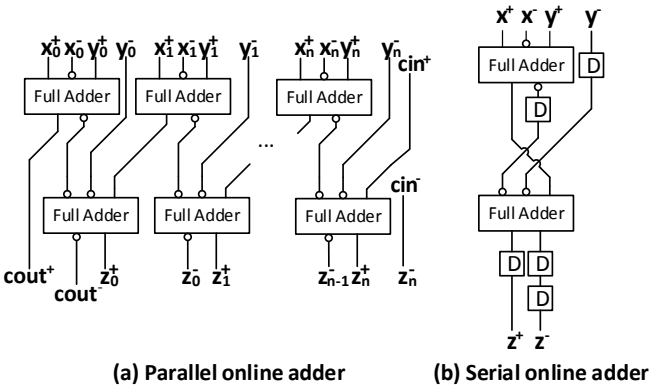

Fig. 3: Online adders, both serial and parallel.

In classic online arithmetic, all operands have a fixed width. For each given P-digit operand, we apply a left-to-right indexing system for its serial computation to the jth iteration. We apply this indexing on inputs ($x[j]$ and $y[j]$) and output ($z[j]$) of online operators. Since we applied a radix-2 system in this paper, $r = 2$ in Equation (1). Letting P be the required full precision, at current computational precision j, where $-\delta \leq j \leq P - 1$, we have:

$$x[j] = \sum_{i=1}^{j+\delta} x_i r^{-i},\ y[j] = \sum_{i=1}^{j+\delta} y_i r^{-i},\ z[j] = \sum_{i=1}^{j+\delta} z_i r^{-i} \quad (1)$$

A number, for example $x[j]$, is represented as a digit-vector as defined in Equation (1). This digit-vector consists of two bit-vectors, namely $x^+[j]$ and $x^-[j]$. In classic online arithmetic architectures, two accumulation registers are employed to store these two bit-vectors. In Figure 2, we show how one accumulation register stores a bit-vector $x^+[j]$. At each iteration j, a new digit is pushed in and the registers start to fill from the MSDs. The widths of these accumulation registers are pre-defined at design-time [2]. In Section IV-A, we present a new method of storing digit-vectors that avoids specifying the precision P at design-time.

A. Online addition

Addition is an important basic arithmetic element and parallel online adders are used for constructing online multipliers and online dividers. Serial online addition makes use of full adders as well as registers to add two redundant digits (Figure 3(b)). A serial online adder contributes two cycles of online delay ($\delta_{add} = 2$) [3]. If we duplicate this serial adder n times and take out the registers, we obtain an n-digit parallel online adder without any online delays [3]. As

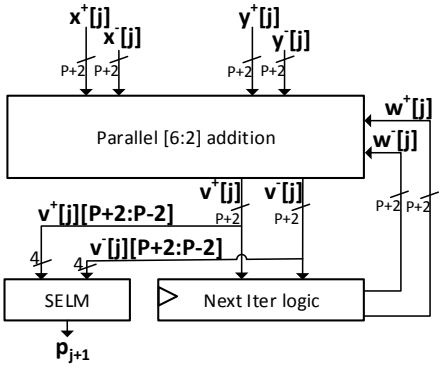

Fig. 4: Classic online multiplier. [6:2] addition represents an adder structure that is able to input 3 digit-vectors and output 1 digit-vector. The width of $P+2$ takes account of 2 cycles of online delays.

shown in Figure 3(a), this parallel online adder is expected to have a critical path crossing two full adders [14]. The fixed critical path indicates a ripple-carry free addition, and therefore the main advantage of a parallel online adder is that its maximum running frequency is independent of its precision requirements [10]. This unique feature enables our designs of online multiplication and division to have a constant f_{max} that is independent of precision. In this paper, the width of a parallel adder is defined as the unrolling width, denoted by U, and is decoupled from the run-time precision requirement P. In our implementation of online multiplication and division, this unrolling width is determined to be 64 digits ($U = 64$).

B. Online multiplication

Algorithm 1 defines the entire multiplication process. Online multiplication is an iterative operation, and produces two intermediate residues, respectively named $v[j]$ and $w[j]$. Meanwhile, for each iteration, a corresponding product digit, p_{j+1}, is produced by a selection function. We implement this selection process as a 4-digit hardware multiplexer, including 2 integer digits and 2 fractional digits. The selection boundaries are based on the $SELM$ function (Equation 2) [3].

Algorithm 1 Online Multiplication

1: Initialize:
$x[-\delta_{mul}] = y[-\delta_{mul}] = v[-\delta_{mul}] = w[-\delta_{mul}] = 0$

2: **for** j **in** $\{-\delta_{mul}, -\delta_{mul}+1, \ldots, P-1\}$ **do**
3: $v[j] = 2w[j] + (x[j]y_{j+1} + y[j+1]x_j) \times 2^{-3}$
4: $p_{j+1} = SELM(v[j])$
5: $w[j+1] = v[j] - p_{j+1}$
6: **end for**

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

In the classic hardware architecture of online multiplication (Figure 4), we represented a digit-vector using two bit-vectors. For example, we used $w^+[j]$ and $w^-[j]$ to represent digit-vector $w[j]$. For producing digit-vector $x[j]$ and $y[j]$, the classic design uses accumulation registers (Figure 2) to store serial input digits (x_{j+1} and y_{j+1}) and transfer them into digit-vectors ($x[j]$ and $y[j]$). In Figure 4, the Next Iteration Logic block is clocked: it holds the $v[j]$ value in a register and computes new $w[j]$ at every positive clock edge. Meanwhile, a combinational logic follows the $SELM$ equation (Equation (2)) to compute the product digit from $v[j]$.

The addition in Algorithm 1 Line 3 is implemented as a parallel online addition (Figure 3(a)) following the classic approach [12]. Figure 4 describes the classic approach of designing an online multiplier. Because of the pre-defined widths of the parallel adder and registers, this classic multiplier is able to achieve multiplication to a precision of at most P, and this precision has to be declared at design-time.

C. Online division

In division, we use x and d to denote numerator and divisor respectively. The algorithm for online division (Algorithm 2) is similar to multiplication, however, there are inherent differences in terms of selection function ($SELD$ and $SELM$) and recurrence paths.

Algorithm 2 Online Division

1: Initialize:
$n[-\delta_{div}] = d[-\delta_{div}] = v[-\delta_{div}] = w[-\delta_{div}] = 0$

2: **for** j **in** $\{-\delta_{div}, -\delta_{div}+1, \ldots, P-1\}$ **do**
3: $v[j] = 2w[j] + (x_{j+1} - q[j]d_{j+1}) \times 2^{-4}$
4: $q_{j+1} = SELD(v[j])$
5: $w[j+1] = v[j] - q_{j+1}d[j+1]$
6: **end for**

$$SELD(v[j]) = \begin{cases} 1 & \text{if } v[j] \geq 1/4 \\ 0 & \text{if } -1/4 \leq v[j] \leq 1/8 \\ -1 & \text{if } v[j] < -1/2 \end{cases} \quad (3)$$

The selection process requires more precise threshold values down to $1/8$, so that five digits (two integer digits and three fractional digits) need to be the input of SELD multiplexer to produce one quotient digit q_{j+1}. There is an additional recurrence path in online division – the quotient vector joins the calculations that produce $v[j]$. In order to achieve this, we employ two additional accumulation registers for storing $q[j]$ so that the produced quotient digit is immediately stored. Again, the classic online divider has precision P to be defined at design-time, and the maximum achievable precision of the quotient is P.

IV. Optimized multiplication and division

In order to construct a class of online operators computing results to any precision at run-time, we firstly need to consider two constraints imposed by the classic online operators. Firstly, the accumulation registers holding digit-vectors have limited storage. Secondly, the parallel online adders in both

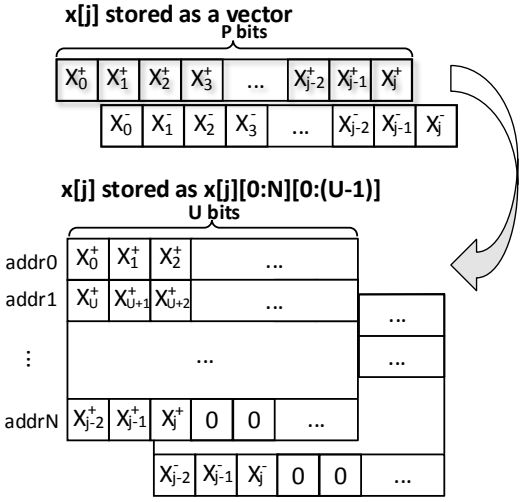

Fig. 5: Different storage mechanisms. Classic implementation stores digit-vectors in fixed-width registers, whereas our implementation reshapes these vectors and stores them into on-chip RAM.

multiplication and division have fixed lengths pre-defined at design-time. In order to overcome the first issue, we store digit-vectors in on-chip RAM of FPGAs (Section IV-A). We introduce a new notation for the remapped digit-vectors, and in Section IV-B, we propose a new hardware architecture to solve the second constraint.

A. Storing digit-vectors

In Section III, a digit-vector is defined as a combination of two bit-vectors. At the top of Figure 5, we illustrate two accumulation registers used for storing $x[j]$. The first constraint of a classic online architecture is that all digit-vectors require pre-defined widths. To avoid overflow in accumulation registers. we can iterate serial online operators at most P times, because this efficiently produces P digits and fills up the registers.

We propose an alternative storage mechanism using on-chip RAM. This storage mechanism exploits the large size of on-chip RAM on modern FPGAs, and the maximum precision is determined by the amount of on-chip RAM available. We define U to be the unrolling width and $N = \lfloor P/U \rfloor$ to be the depth. The unrolling width is a parameter that designers could control to fit the width of any given block RAMs. At the bottom of Figure 5, we define N to be the depth of on-chip RAM. Mathematically, we define the following new representation of $x^+[j]$:

$$x^+_{new}[j][0:N][0:(U-1)] = x^+_{classic}[j][0:(P-1)] \quad (4)$$

In the classic design, a bit-vector of $x^+_{classic}[j][0:(P-1)]$ is two dimensional: it has time index j and also physical width of P. We add another dimension to this bit-vector to make $x^+_{new}[j][0:N][0:(U-1)]$ – a three dimensional vector with a time index j, a physical width U and a physical depth of $N+1$. In the following sections, all digit-vectors are rearranged to fit into this new representation.

B. Reuse of parallel adder

1) MSD logic: A novel hardware architecture with efficient reuse of a parallel online adder is illustrated in Figure 6.

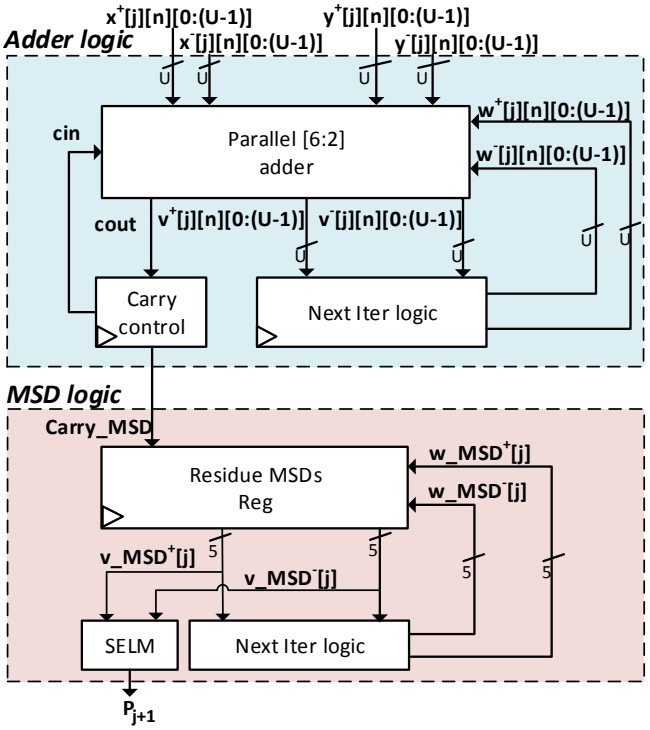

Fig. 6: Optimized online multiplier with reuse of adder; in adder logic, a notation of $v^+[j][n][(0:(U-1)]$ represents a U-bit-vector for computing the jth output at the nth iterative addition, this is the new representation introduced in Equation (4). In MSD logic, $v\_MSD^+[j]$ represents a 5-bit-vector for computing the jth output.

The underlying logic is to compute the parallel summation iteratively, if the width of digit-vectors P is larger than the unrolling width of parallel online adders U. The example shown is an online multiplier, and the architecture can be broken down into two parts (Figure 6): MSD logic (red, bottom) and Adder logic (blue, top).

The MSD logic links tightly with the algorithm of online multiplication. The algorithm indicates that the upper 5 MSDs of $v[j]$ remain unchanged unless a valid carry is propagated from the LSDs. Intuitively, we break down the residues of classic online multiplier into two parts to avoid unnecessary additions. In this case, the upper part, named $v\_MSD[j]$, stays constant until the iterated additions of LSDs propagate a valid carry that affects the MSDs. Notice we denoted $v\_MSD[j]$ in terms of $v^+\_MSD[j]$ and $v^-\_MSD[j]$, because the digit-vector consists of two 5-bit bit-vectors. The Next Iter logic is a combinational logic computing the next cycle input to the MSD register. At the end, following the same procedure in classic online architecture, we compute p_{j+1} based on the $SELM$ function. Division employs the same MSD logic. In division, the separated MSDs have a width of 6 digits due to the multiplication by 2^{-4} in Algorithm 2.

2) Adder logic: In order to introduce the iterative addition logic, we firstly need to define a few parameters. The adder logic works consistently with the vector storage methodology. To add U-digit wide data, with respect to Figure 6, we define U to be the width of the online parallel adder. At a particular

computation time, we define the run-time width of $x[j]$, $y[j]$, $v[j]$ and $w[j]$ to be P, but in a departure from the classic architecture, we allow this P to increase at run-time if the user requires a larger precision. We thus define $P = NU + u$, where $u \in [0 : (U-1)]$. Similarly, we use n to denote the current writing address of on-chip RAM, mathematically, and we have $n \in [0 : N]$.

In Equation (4), we introduced a new representation of digit-vectors, and thus write $x[j][0 : N][0 : (U-1)]$ to denote a digit-vector. At a particular clock cycle, n and j are well-defined, and thus we only process computation with respect to $x[j][n][0 : (U-1)]$. This is a one-dimensional digit-vector with width U and it consists of two bit-vectors, namely $x^+[j][0 : N][0 : (U-1)]$ and $x^-[j][0 : N][0 : (U-1)]$. In the adder logic, we perform U-digit wide iterative additions on these one-dimensional vectors.

The underlying logic of adder reuse is summarized in the flowchart of Figure 7, consistently, we have Figure 6 to illustrate the hardware architecture. We then introduce each hardware block illustrated in Figure 6.

The first block is the Next Iter logic block, it computes values of $w[j][n][0 : (U-1)]$ from $v[j][n][0 : (U-1)]$ in a similar fashion to the MSD logic obeying Algorithm 1, however, this Next Iter logic for adder logic is clocked, because it involves storing digit-vector $v[j][0 : N][0 : (U-1)]$ in on-chip memory.

The second hardware block, Parallel $[6 : 2]$ adder, is a key hardware unit. The control flow of utilizing this fixed-width parallel adder follows Figure 7. We compute the total number of addition iterations: $N = \lfloor P/U \rfloor$. The value of N has two underlying meanings. Firstly, N corresponds to the depth of on-chip RAM used for storing digit-vectors, as defined in Section IV-A. Secondly, it serves as a control logic. We initially assign $n = N$, this n serves both as an address pointer for on-chip RAM and a control signal for parallel adder. If n is non-zero, we keep adding vectors starting from LSDs. In other words, if using $x[j][n][0 : U]$ as an example again, we start addition from $x[j][N][0 : U]$ to $x[j][0][0 : U]$ at every clock cycle. As shown in Figure 7, we decrease the value of n at every iteration ($n = n - 1$). This iterative procedure will terminate once n reaches zero.

The classic online architecture computes internal parallel addition consuming only one clock cycle when it is fully pipelined, however, our suggested hardware computes the same addition using $N + 1$ cycles. Intuitively, in order to obtain a constant hardware utilization and be able to track to arbitrary precisions at run-time, our proposed architecture spends multiple clock cycles in parallel online addition if the required run-time precision is wider than the width of parallel online adders U.

The third hardware block introduced is the Carry control block. During this iterative addition, the Carry control block in Figure 6 stores carry outs and outputs carry ins. When we are in the iterative process, by setting an enable signal, the carry out of the online parallel adder is stored in a register and carry in would take this stored value at the next clock cycle

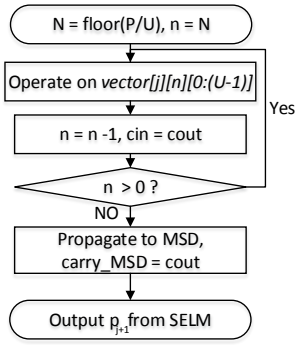

Fig. 7: Adder logic flow chart.

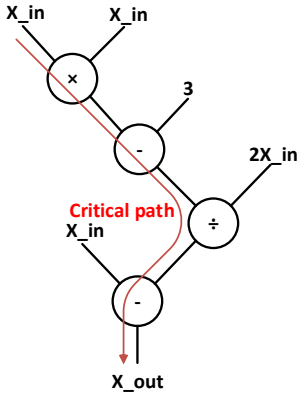

Fig. 8: Datapath of a single iteration of Newton's method.

(*cin* = *cout*). At the same time, the value of *carry_MSD* is kept to be zero, because the carry propagation has not reached the MSDs yet. When the process exits from the iterative loop, it states that a valid carry could be propagated to the upper 5 MSDs. We enable the carry to update the MSDs by setting *carry_MSD* = *cout* but keep *cin* = 0. This control logic is summarized as a flow chart in Figure 7.

After describing all important hardware blocks, we notice the iteration count j grows with the required run-time precision P. When entering a new jth iteration, we store new digits into the on-chip RAM, and P grows by one. This indicates the unique advantage of our system – it is able to compute results to arbitrary precision at run-time.

In division, the process of adder reuse is the same, but we have to take care of an extra recurrence path (Algorithm 2). Division uses two $[4 : 2]$ parallel addition units and these adders employ the same adder reuse technique as multiplication.

C. Analysis of clock cycles

To fully visualize the consumption of clock cycles by our online operators, we construct a relationship between number of computation clock cycles and run-time precision requirements. In order to characterize the analytic expressions of clock cycles, we recall definitions of following parameters:

- Unrolling width : U
- Required precision at output: P
- Computation clock cycles: C
- Online delay: δ

In addition, using existing parameters, we compute $N = \lfloor P/U \rfloor$. $N+1$ is the number of iterations required for finishing the iterative addition, and is also the depth of on-chip RAM used for storing digit-vectors. The expression for online adder is straightforward, we use subscript OA for the online adder:

$$C_{OA} = \delta_{OA} + P \tag{5}$$

For the multiplier, the number of clock cycles spent in computing each digit varies depending on the unrolling width U. Similarly, we use OM to denote online multiplication:

$$C_{OM} = \delta_{OM} + (P - NU)(N + 1) + \sum_{i=1}^{N} iU \tag{6}$$

We use OD to denote online division:

$$C_{OD} = \delta_{OD} + (P - NU)(N + 1) + \sum_{i=0}^{N} iU \qquad (7)$$

As explained in Section IV-B2, the number of iterative additions required increases segmentally with run-time precision P, and we characterize this using $\sum_{i=0}^{N} iU$.

Using equations developed in this section, given any numerical precision at run-time, we are able to tune operators to reach that precision by controlling the number of computational clock cycles. The maximum precision is limited only by the amount of on-chip RAM available. This effectively opens the door to tuning arithmetic precision at run-time.

V. Evaluation

To experimentally explore how pipelining behaves when online operators are chained together, we implement an iterative algorithm using online operators. Algorithm 3 recalls Newton's method for finding $\sqrt{3}$.

Algorithm 3 Newton's method with five iterations

1: Initialize: $x_0 = 2, f(x) = x^2 - 3, f'(x) = 2x$
2: **for** j **in** $\{0, \ldots, 4\}$ **do**
3: $\quad x_{j+1} = x_j - \frac{f(x_j)}{f'(x_j)}$
4: **end for**

This is a class of iterative algorithm where we expect online operators to have better performance than traditional fixed-point, because it involves chained division and multiplication (Figure 8). We expect carries to travel in non-unified directions for fixed-point arithmetic, and calculations are stalled in traditional fixed-point if the carry propagation direction reverses. In online arithmetic, the uniform carry propagation avoids such stalls. In this case, we unrolled the Newton's method to its 5th iteration to examine the behavior of different arithmetics when the precision required increases.

A. Empirical evaluation

For an implementation of Newton's method, we use the Altera Cyclone IV FPGA (EP4CGX150DF31I7AD) as the hardware platform and collect results after place and route in Quartus II. Both serial fixed-point arithmetic and serial online arithmetic are implemented. The correctness of results has been verified by co-simulation in Matlab.

The fixed-point multiplier and divider normally require a trade-off between speed and circuitry when choosing them to be parallel or serial. Parallel-in-serial-out (PISO) fixed-point operators sit at the intermediate point between fully serial and fully parallel. In this case study, we decided to implement PISO fixed-point multiplier and divider for two reasons. Firstly, with an increasing precision requirement, PISO fixed-point arithmetic suffers less from area growth and f_{max} degradation compared to parallel-in-parallel-out (PIPO) multiplication and division [7], but is significantly faster than serial-in-serial-out (SISO) arithmetic at moderate sampling rate [7]. Secondly, multiplication and division's serial outputs can be pipelined by successive SISO fixed-point adders.

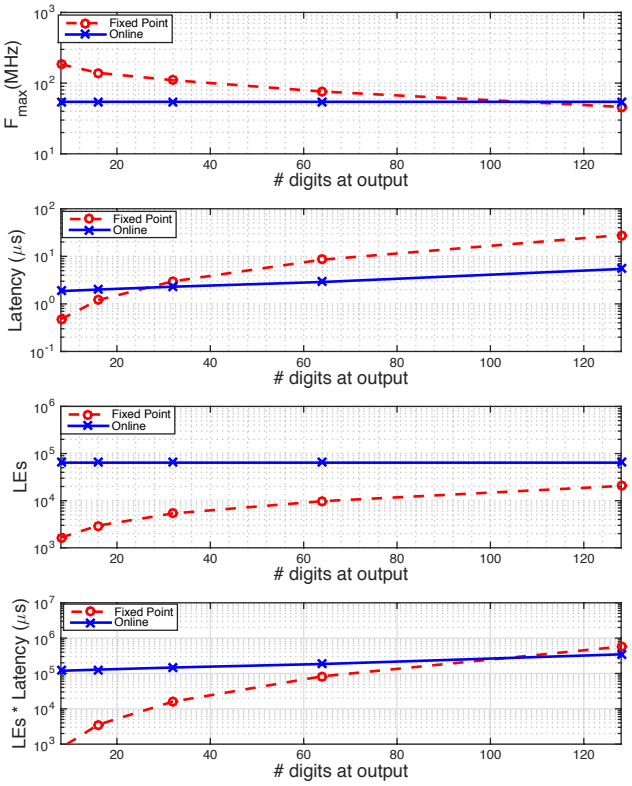

Fig. 9: Comparison between online and fixed-point arithmetics on Newtons Method in terms of f_{max}, latency and number of LEs used.

Our experimental results are illustrated in Figure 9. PISO fixed-point's running frequency is higher at low precision but becomes comparable to online arithmetic at high precisions. Also, online arithmetic has a constant f_{max} for all precisions.

In terms of latency, the initial latency of online arithmetic is higher due to the accumulation of online delays, but it grows less dramatically compared to fixed-point. In the second plot of Figure 9, at a precision of 30 digits, online has a similar latency to fixed-point, but has 2.95x and 5.16x speed increases when the precision is at 64 digits and 128 digits respectively.

With respect to hardware utilization, the complex circuitry of online arithmetic makes use of more circuit area than fixed-point. However, comparing our design with the classic approach, this area overhead is restricted for two main reasons. Firstly, the hardware usage stays constant for our design and it displays a comparable Logic elements (LEs) usage to PISO fixed-point arithmetic at high precisions. Secondly, we are able to trade-off the hardware usage with latency for our proposed design by varying the parameter U. This effectively varies the length of the parallel online adder – a design with smaller parallel online adder takes more clock cycles to produce the same output but consumes less circuitry. In addition, in the third plot of Figure 9, we observe a trend that online arithmetic is closing the gap of hardware overhead compared to PISO fixed-point by consuming only constant circuitry. It should be noted that PISO fixed-point multipliers and dividers contain internal parallel fixed-point adders, and these adders cause an increase of hardware utilization when precision grows.

Finally, we compute the product of the number of LEs and the latency. For computing the Newton's method to its fifth iteration, online arithmetic outperforms PISO fixed-point arithmetic by 1.66x at a precision of 128 digits in terms of LEs $\times$ Latency. This case study demonstrates that, for a low number of iterations but at high precision, our implementation of online arithmetic is superior to fixed-point.

B. Analytical evaluation

In the previous section, we compared performance of online arithmetic with fixed-point arithmetic on an algorithm with a fixed number of unrolled iterations. The complexity of arithmetic operations is proportional to the number of unrolled iterations – there are more chained arithmetic operations with more unrolled iterations. We are thus interested in the performance of online arithmetic at various numbers of unrolled iterations. For the implementation of Newton's method, it is possible to characterize the consumed clock cycles analytically using expressions in Section IV-C. We adapt the symbols defined in previous sections, and also define additional parameters. Subscript ON represents Newton's method implemented by online arithmetic, and K represents the number of iterations that we choose to compute for Newton's method.

$$C_{ON} = (\delta_{OM} + \delta_{OA} + \delta_{OA} + \delta_{OD})K + 4NK + (P - NU)(N+1) + \sum_{i=0}^{N} iU$$

$$= 21K + 4NK + (P - NU)(N+1) + \sum_{i=0}^{N} iU \quad (8)$$

In our analytic formula, based on the design of online arithmetic operators, specific values of online delays are defined: $\delta_{OA} = 3$, $\delta_{OM} = 7$ and $\delta_{OD} = 8$. The online delays of multiplication and division are larger than classic implementation, because reads and writes of on-chip RAM cost extra cycles. In this expression, an extra term $4NK$ is introduced. This is because the increment of N stalls all $4K$ operators on the critical path (Figure 8). For example, operators take one cycle to compute a result when $N = 1$; if $N = 2$, this effectively breaks previous pipelining and all successive operators have to wait for one extra clock cycle. Using the same unrolling width ($U = 64$) as before, these $4NK$ increments are notable when precisions are at multiples of U in Figure 10 and Figure 11, because N is defined as $\lfloor P/U \rfloor$ in Section IV-C.

Similarly, we can consider an analytical expression for our design of fixed-point arithmetic. This design includes PISO multipliers and dividers but SISO adders. It takes advantage of pipelined multiplication and subtraction, however, digits are stalled before and after division because of the reverse of the carry propagation direction.

$$C_{PISO} = (2 + 2P)K = 2PK + 2K \quad (9)$$

For SISO arithmetic, we characterize its clock cycles in a similar fashion. Classic design of SISO multipliers and dividers costs P^2 cycles for inputs with a precision of P [6]:

$$C_{SISO} = (2 + 2P^2)K = 2P^2K + 2K \quad (10)$$

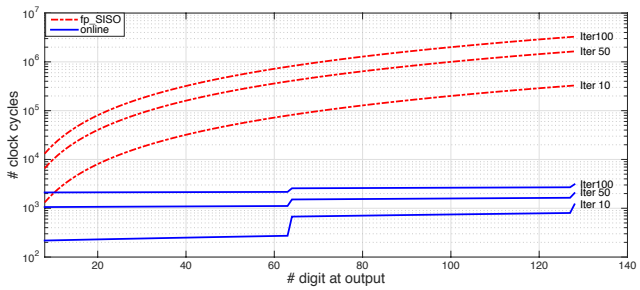

Fig. 10: Number of clock cycles for computing Newton's method with 10 iterations, 50 iterations and 100 iterations, comparing online arithmetic with SISO fixed-point.

In Equation (9) and Equation (10), there exist two subtractions for each unrolled iteration of Newton's method. We spend one clock cycle on each of the subtractions illustrated in Figure 8, because we can stream multiplication results into successive SISO adders.

These three analytic expressions fully characterize how different arithmetic would perform when solving Newton's method. The number of computational clock cycles is determined by both the required precision and the number of unrolling iterations of Newton's method.

SISO operators normally run at high frequencies [8], however, the clock cycle advantage of our implementation of online operators offsets SISO operators' high running frequency. In Figure 10, we compare serial online arithmetic with SISO fixed-point arithmetic. For iterating 100 times, at a precision of 16 digits, the number of clock cycles of online is one order of magnitude lower, but this difference increases to three orders of magnitude when precision is at 64 digits. With an increasing number of unrolling iterations, the difference in clock cycles between online and SISO fixed-point increases. Although our FPGA implementation of online arithmetic only runs at a relatively low frequency (Figure 9), a constant difference in running frequencies would not offset the increasing difference in clock cycles. In addition, SISO fixed-point is merely used for computations at moderate frequency, we thus give a more in-depth comparison between online and PISO fixed-point.

Figure 11 compares PISO fixed-point with online arithmetic in terms of computation clock cycles and latency at various high numbers of iterations. The first plot indicates that, at high number of iterations, serial online operators compute results with the same precision by using fewer computational clock cycles. The second plot takes into account the f_{max} disadvantage of online arithmetic, but still shows that online has a better performance than fixed-point if the required precision is more than 20 digits. At the same time, from the second plot, we observe online arithmetic is more advantageous at high iteration numbers. In this case, at the 10th iteration, online arithmetic outperforms PISO fixed-point at a precision of 20, but it outperforms fixed-point at a precision of 16 when number of iteration is 100. Both plots have online arithmetic showing an upward stepping trend, this is because of the $4NK$ term in Equation (8). The analytical results suggest that, benefiting from the MSD-first computation pattern, online

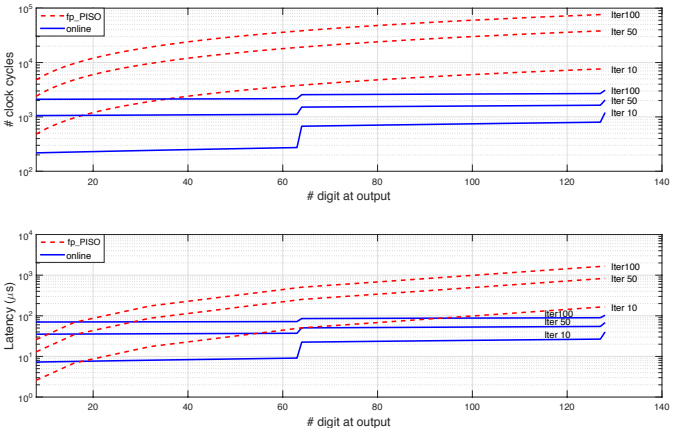

Fig. 11: Number of clock cycles and latency for computing Newton's method with 10 iterations, 50 iterations and 100 iterations, comparing online arithmetic with PISO fixed-point.

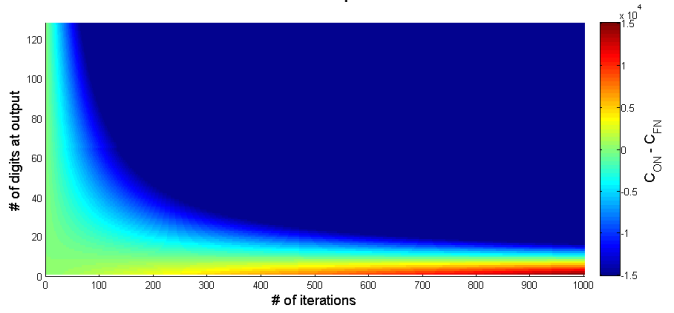

Fig. 12: Difference in clock cycles for computing Newton's method to different iterations and different precision. The number of computed iterations of Newton's method is on the x axis, and the number of digits produced is on the y axis. The color of this heatmap indicates the performance in terms of difference in number of computation clock cycles. C_{ON} represents the number of clock cycles needed by online arithmetic, and C_{FN} represents clock cycles consumed by PISO fixed-point. 'Hotter' region indicates online arithmetic is having a worse performance, in contrast, a 'colder' regions means online arithmetic outperforms PISO fixed-point arithmetic.

arithmetic is preferable for algorithms with many iterations.

To fully analyze how online arithmetic performs at high number of iterations and high precision requirement, we plot a heatmap in Figure 12. The heatmap suggests online arithmetic is preferable for iterative algorithms requiring high precision or a high number of iterations. In this case, we only consider the clock cycles, because online operators in our design have a constant f_{max}, and researchers have demonstrated possibilities for classic online operators to run at higher clock frequency than fixed-point arithmetic [9].

Online arithmetic has similar performance to fixed-point at a precision requirement of 8 digits for all numbers of unrolling iterations. At an iteration number of 100, online arithmetic outperforms fixed-point by 1.7x, 3x and 8x if required precisions are at 16 digits, 32 digits and 64 digits.

VI. Conclusion

In this paper, we exploit a novel method of computing numerical results to an arbitrary precision at run-time. Using this method, we are able to tune the precision of numerical solutions of iterative algorithms at run-time with only constant hardware cost.

This method relies on new hardware architectures for online multiplication and online division. Our optimized online multiplier and divider efficiently reuse their internal parallel addition logic. They store digit-vectors in on-chip memory and thus the maximum run-time precision is limited only by the size of on-chip RAM on FPGAs. Besides, our hardware architecture also allows the user to trade-off hardware consumption with computational clock cycles, where the f_{max} and hardware utilization of optimized operators stays constant with increasing precision requirements. By comparing our design with PISO fixed-point arithmetic, we demonstrated both empirically and analytically that online arithmetic is superior to fixed-point arithmetic in solving iterative algorithms with a high precision requirement or a large number of iterations.

VII. Acknowledgments

The support of EPSRC grants EP/I020357/1 and EP/K034448/1, the Royal Academy of Engineering and Imagination Technologies is gratefully acknowledged. The author would also like to thank Junyi Liu and Jiang Su for their support. Data underlying this article can be accessed on zenodo at http://doi.org/10.5281/zenodo.159454, and used under the Creative Commons Attribution licence.

References

[1] Dimmler, A. Tisserand, U. Holmbeg, and R. Longchamp. On-line arithmetic for real-time control of microsystems. *Mechatronics, IEEE/ASME Transactions on*, 4(2):213–217, Jun 1999.

[2] M. D. Ercegovac. On-line arithmetic: An overview. In *28th Annual Technical Symposium*, pages 86–93. International Society for Optics and Photonics, 1984.

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

[4] R. Hartley and P. Corbett. Digit-serial processing techniques. *IEEE Transactions on Circuits and Systems*, 37(6):707–719, June 1990.

[5] M. J. Irwin and R. M. Owens. Digit-pipelined arithmetic as illustrated by the paste-up system: A tutorial. *Computer*, 20(4):61–73, Apr. 1987.

[6] M. Lehman, D. Senzig, and J. Lee. Serial arithmetic techniques. In *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I*, AFIPS '65 (Fall, part I), pages 715–725, New York, NY, USA, 1965. ACM.

[7] 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 VLSIS*, 19:1733–1745, 2011.

[8] W. G. Natter and B. Nowrouzian. Digit-serial online arithmetic for high-speed digital signal processing applications. In *Signals, Systems and Computers, 2001. Conference Record of the Thirty-Fifth Asilomar Conference on*, pages 171–176 vol.1, Nov 2001.

[9] J. Olivares, J. Hormigo, J. Villalba, I. Benavides, and E. L. Zapata. SAD computation based on online arithmetic for motion estimation. *Microprocessors and Microsystems*, 30(5):250–258, 2006.

[10] K. Shi, D. Boland, and G. A. Constantinides. Efficient FPGA implementation of digit parallel online arithmetic operators. FPT, 2014.

[11] K. Shi, D. Boland, E. Stott, S. Bayliss, and G. A. Constantinides. Datapath synthesis for overclocking: Online arithmetic for latency-accuracy trade-offs. DAC '14, NY, USA, 2014. ACM.

[12] K. S. Trivedi and M. D. Ercegovac. On-line algorithms for division and multiplication. In *ARITH, 1975 IEEE 3rd Symposium on*, pages 161–167. IEEE, 1975.

[13] K. S. Trivedi and M. D. Ercegovac. On-line algorithms for division and multiplication. *IEEE Trans. Comput.*, 26(7):681–687, July 1977.

[14] P. K.-G. Tu. *On-line Arithmetic Algorithms for Efficient Implementation*. PhD thesis, University of California Los Angeles, 1990.